UNIVERSITY OF OKLAHOMA
GRADUATE COLLEGE

LATENT POTENTIAL IN MIXTURE-OF-EXPERTS FOR IMPROVING INTERPRETABILITY, TRAINING PROCESSES, AND DYNAMIC INFERENCE

A DISSERTATION
SUBMITTED TO THE GRADUATE FACULTY
in partial fulfillment of the requirements for the
Degree of
DOCTOR OF PHILOSOPHY

by ZACKARY HOYT
Norman, Oklahoma
2026

Downloaded from SHAREOK. Persistent Handle URL: https://hdl.handle.net/11244/342820

LATENT POTENTIAL IN MIXTURE-OF-EXPERTS FOR IMPROVING INTERPRETABILITY, TRAINING PROCESSES, AND DYNAMIC INFERENCE

A DISSERTATION APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Dean Hougen (Chair)

Dr. Chao Lan

Dr. Amy McGovern

Dr. Justin Metcalf (Graduate College Representative)

©Copyright by ZACKARY HOYT 2026

All Rights Reserved.

Acknowledgments

To my wonderful wife, Dr. Rachel E. Cook,

Your intellect, compassion, and aspirations have been inspirational. In these last couple of years, you have lifted me up whenever I stumbled, cheered for me when I moved forward, and endured my endless ramblings. You have patiently waited for me to finish this dissertation so we could finally take time to travel the world together, and I am looking forward to each and every day of sharing beautiful sunsets, brilliant waters, and happy memories together.

To my amazing family and friends,

Your pride in me and hope for my success have been a phenomenal source of support from which I have drawn a great deal of encouragement. You have been there for me, no matter the distance between us or the time apart. I am incredibly fortunate to have you all in my life. Thank you again for all you have said and done to help me along this journey.

Table of Contents

Acknowledgments iv

List of Tables ix

List of Figures xii

Abstract xvii

1 Introduction 1
- 1.1 Overview 1
- 1.2 Motivation 2
- 1.3 Baselines 2
- 1.4 Interpretability 3
- 1.5 Transfer learning 5
- 1.6 Computational Optimization 5
- 1.7 Layout 6

2 Background 8
- 2.1 Overview 8
- 2.2 Deep Learning 8
  - 2.2.1 AI/ML 8
  - 2.2.2 Neural Networks 10
  - 2.2.3 Neural Architectures 20
- 2.3 Summary 34

3 Mixture-of-Experts (MoE) 35
- 3.1 Introduction 35
- 3.2 Brief History 36
- 3.3 Motivation 39
- 3.4 Baseline Architecture 42
- 3.5 Experts Mixture 43
  - 3.5.1 Assumptions 44
  - 3.5.2 Encoding 44
  - 3.5.3 Routing Selection 45
  - 3.5.4 Output Approach 46
  - 3.5.5 Error Gradients 47

3.5.6 Sparse Routing over Batches . . . 48
3.6 Routing Methods . . . 50
3.7 Metrics . . . 55
3.8 MoE Routing Overhead . . . 58
3.9 Summary . . . 59

4 PyTorch Model Factory . . . 60
4.1 Overview . . . 60
4.2 Framework Motivation . . . 61
4.2.1 TMF CNN Example . . . 64
4.2.2 TMF EncDec Example . . . 66
4.3 Common Model Specification . . . 68
4.4 Feed-Forward Models . . . 69
4.4.1 Dense Feed-Forward Neural Networks . . . 69
4.4.2 Convolutional Feed-Forward Neural Networks . . . 70
4.4.3 Residual Convolutional Feed-Forward Neural Networks . . . 71
4.5 Encoder–Decoder and Autoencoder Models . . . 72
4.5.1 Convolutional Autoencoders . . . 73
4.5.2 UNet-Style Autoencoders . . . 74
4.5.3 Variational Convolutional Autoencoders . . . 74
4.5.4 Residual Variational Convolutional Autoencoders . . . 75
4.6 Mixture-of-Experts Model Definition . . . 76
4.6.1 Router Encoder and Expert Inputs . . . 76
4.6.2 Routing Result Layout . . . 77
4.6.3 Expert Stack and Aggregation . . . 78
4.6.4 Router Variants . . . 79
4.7 Summary . . . 80

5 Baseline Experiments . . . 82
5.1 Overview . . . 82
5.2 Classification Datasets . . . 82
5.2.1 MNIST . . . 83
5.2.2 CIFAR-10 . . . 84
5.2.3 CIFAR-100 . . . 85
5.2.4 Transformations . . . 85
5.3 Classifier Baselines . . . 87
5.3.1 MNIST . . . 91
5.3.2 CIFAR-10 . . . 96
5.3.3 CIFAR-100 . . . 100
5.3.4 Summary . . . 104
5.4 Autoencoder Baselines . . . 104
5.4.1 MNIST . . . 107
5.4.2 CIFAR-10 . . . 111
5.4.3 CIFAR-100 . . . 112
5.4.4 Summary . . . 117

5.5 MoE Baselines . . . 118
5.5.1 MNIST . . . 121
5.5.2 CIFAR-10 . . . 125
5.5.3 CIFAR-100 . . . 130
5.5.4 Summary . . . 133
5.6 MoE Scalability Experiments . . . 134
5.6.1 Summary . . . 137

6 Interpretability . . . 139
6.1 Overview . . . 139
6.2 Latent Interpretability . . . 140
6.2.1 MNIST . . . 147
6.2.2 CIFAR-10 . . . 156
6.2.3 CIFAR-100 . . . 163
6.2.4 Summary . . . 167
6.3 Routing Attention . . . 169
6.3.1 MNIST Attention-Routing Results . . . 171
6.3.2 CIFAR-10 Attention-Routing Results . . . 177
6.3.3 CIFAR-100 Attention-Routing Results . . . 180
6.3.4 Summary . . . 182
6.3.5 Latent Encodings . . . 183
6.4 Conclusions . . . 185

7 Transferring Latent Learning . . . 187
7.1 Overview . . . 187
7.2 Experiment Setups . . . 189
7.2.1 Pretraining Sources and Transfer Modes . . . 189
7.2.2 Updated Baseline Configurations . . . 190
7.2.3 Evaluation and Matched Baseline Comparison . . . 191
7.3 MNIST . . . 191
7.3.1 Classifier-Pretrained Router . . . 191
7.3.2 Autoencoder-Pretrained Router . . . 193
7.4 CIFAR-10 . . . 196
7.4.1 Classifier-Pretrained Router . . . 196
7.4.2 Autoencoder-Pretrained Router . . . 198
7.5 CIFAR-100 . . . 200
7.5.1 Classifier-Pretrained Router . . . 200
7.5.2 Autoencoder-Pretrained Router . . . 201
7.6 Cross-Dataset Comparison . . . 202
7.7 Summary . . . 203

8 Dynamic Inference Paths . . . 205
8.1 Overview . . . 205
8.2 Complexity Bias . . . 206
8.3 Approach . . . 207

8.4 Experiment Setups . . . 208
8.5 Results . . . 209
8.5.1 MNIST . . . 209
8.5.2 CIFAR-10 . . . 212
8.5.3 CIFAR-100 . . . 215
8.6 Conclusions . . . 216

9 Conclusions . . . 217
9.1 Overview . . . 217
9.2 Latent Interpretability . . . 218
9.3 Transferring Autoencoder Learning to Routers . . . 220
9.4 Dynamic Inference . . . 220
9.5 Future Work . . . 222
9.6 Final Remarks . . . 223

Reference List . . . 224

List of Tables

5.1 CIFAR-10 class index mapping 85

5.2 CIFAR-100 superclass and class mapping 86

5.3 MNIST classifier baseline results 92

5.4 CIFAR-10 classifier baseline results 96

5.5 CIFAR-100 classifier baseline results 100

5.6 MNIST autoencoder baseline results 108

5.7 CIFAR-10 autoencoder baseline results 111

5.8 CIFAR-100 autoencoder baseline results 112

5.9 MNIST MoE results for hardening schedules 122

5.10 MNIST MoE results for constant top-k schedules 123

5.11 CIFAR-10 MoE results with $P_e = 8$ 126

5.12 CIFAR-10 MoE results with $P_e = 16$ 127

5.13 CIFAR-100 MoE baseline test results 130

5.14 MNIST MoE scalability experiment results 136

6.1 MNIST attention-routing results with a VAE router encoder 171

6.2 MNIST attention-routing results with a UNet router encoder 174

6.3 CIFAR-10 attention-routing MoE test results 177

6.4 CIFAR-100 attention-routing MoE test results 180

7.1 Transfer-learning source and expert configurations 190

7.2 MNIST classifier-pretrained adaptive transfer results 192

7.3 MNIST classifier-pretrained frozen transfer results 192

7.4 MNIST autoencoder-pretrained adaptive transfer results 193

7.5 MNIST autoencoder-pretrained frozen transfer results 194

7.6 CIFAR-10 classifier-pretrained adaptive transfer results 197

7.7 CIFAR-10 classifier-pretrained frozen transfer results 197

7.8 CIFAR-10 autoencoder-pretrained adaptive transfer results 198

7.9 CIFAR-10 autoencoder-pretrained frozen transfer results 199

7.10 CIFAR-100 classifier-pretrained adaptive transfer results 200

7.11 CIFAR-100 autoencoder-pretrained adaptive transfer results 201

7.12 Average epochs for the transfer-learning experiments 202

8.1 Dynamic-inference experiment configurations 208

8.2 MNIST width-doubled experts dense-mixture training results 209

8.3 MNIST width-doubled experts top-1 path training results 210

8.4 MNIST depth-doubled experts dense-mixture training results 211

8.5 MNIST depth-doubled experts top-1 path training results 211

8.6 CIFAR-10 width-doubled experts dense-mixture training results 212

8.7 CIFAR-10 width-doubled experts top-1 path training results 213

8.8 CIFAR-10 depth-doubled experts dense-mixture training results 214

8.9 CIFAR-10 depth-doubled experts top-1 path training results 214

8.10 CIFAR-100 depth-doubled residual experts dense-mixture training results . . 215

List of Figures

2.1 Biological neuron example 10

2.2 Artificial neuron example 11

2.3 Comparison of nonconvex and convex surfaces 16

2.4 Linear layer example 22

2.5 Convolution layer example 24

2.6 Convolution filters example 25

2.7 Transposed-convolution layer example 28

2.8 Residual skip connection example 33

3.1 Classical MoE architecture diagram 37

3.2 Histogram of MoE research publications (1992-2012) in 2012 survey paper . 38

3.3 Graph MoE architecture 41

3.4 Chain-of-Experts architecture 42

3.5 Baseline MoE architecture 43

3.6 Routing algorithms visualization 52

4.1 Example Conv2D FFNN 62

4.2 Example Conv2D UNet 66

5.1 Example samples from the MNIST dataset 83
5.2 Example samples from the CIFAR-10 dataset 84
5.3 Example samples from the CIFAR-100 dataset 86
5.4 Illustration of the conv2d neural network architecture 87
5.5 Plot detailing *loss* and *acc1* metrics for the MNIST classifier baselines 94
5.6 Plot detailing *loss* and *acc5* metrics for the MNIST classifier baselines 94
5.7 Plot detailing *loss* and *acc1* metrics for the CIFAR-10 classifier baselines . . 98
5.8 Plot detailing *loss* and *acc5* metrics for the CIFAR-10 classifier baselines . . 98
5.9 Plot detailing *loss* and *acc1* metrics for the CIFAR-100 classifier baselines . 102
5.10 Plot detailing *loss* and *acc5* metrics for the CIFAR-100 classifier baselines . 103
5.11 Example samples from the MNIST dataset (repeated) 108
5.12 MNIST VAE reconstruction examples 109
5.13 MNIST VAE latent projection with lower capacity 110
5.14 MNIST VAE latent projection with greater capacity 110
5.15 Example samples from the CIFAR-10 dataset (repeated) 112
5.16 CIFAR-10 VAE reconstruction examples 113
5.17 CIFAR-10 VAE latent projection with lower capacity 114
5.18 CIFAR-10 VAE latent projection with greater capacity 114
5.19 Example samples from the CIFAR-100 dataset (repeated) 115
5.20 CIFAR-100 residual VAE reconstruction examples 115
5.21 CIFAR-100 residual VAE latent projection with lower capacity 116
5.22 CIFAR-100 residual VAE latent projection with greater capacity 116
5.23 MNIST MoE baseline summaries over K with respect to each routing algorithm 125

5.24 CIFAR10 MoE baseline summaries over K with respect to each routing algorithm . . . 129

5.25 CIFAR100 MoE baseline summaries over K . . . 132

5.26 MNIST MoE scalability experiments details . . . 135

6.1 Example 2D SVD illustration . . . 142

6.2 SVD projection example . . . 143

6.3 t-SNE projection example . . . 144

6.4 UMAP projection example . . . 145

6.5 Grid of 2D SVD projections for MNIST latent encodings . . . 147

6.6 Example MNIST latent encodings using 2D SVD projection learned by a MoE router . . . 148

6.7 Example MNIST expert specializations via acc1 confusion matrix diagonals per expert . . . 149

6.8 Example MNIST latent encodings using 2D SVD projection learned by a MoE router . . . 150

6.9 Example MNIST expert specializations via acc1 confusion matrix diagonals per expert . . . 150

6.10 Grid of 2D t-SNE projections for MNIST latent encodings . . . 152

6.11 Grid of 2D UMAP projections for MNIST latent encodings . . . 152

6.12 Example MNIST latent encodings using 2D t-SNE projection learned by a MoE router . . . 153

6.13 Example MNIST latent encodings using 2D UMAP projection learned by a MoE router . . . 153

6.14 Example MNIST latent encodings using 2D t-SNE projection learned by a MoE router . . . 154

6.15 Example MNIST latent encodings using 2D UMAP projection learned by a MoE router . . . 154

6.16 CIFAR-10 `tK2t1` latent encodings using a 2D SVD projection 156

6.17 CIFAR-10 `tK2tk` latent encodings using a 2D SVD projection 157

6.18 CIFAR-10 expert specializations under `tK2t1` 158

6.19 CIFAR-10 expert specializations under `tK2tk` 158

6.20 CIFAR-10 `tK2t1` latent encodings using a 2D t-SNE projection 159

6.21 CIFAR-10 `tK2tk` latent encodings using a 2D t-SNE projection 160

6.22 CIFAR-10 `tK2t1` latent encodings using a 2D UMAP projection 161

6.23 CIFAR-10 `tK2tk` latent encodings using a 2D UMAP projection 161

6.24 CIFAR-100 latent encodings using a 2D SVD projection 163

6.25 CIFAR-100 expert specializations over superclass groups 164

6.26 CIFAR-100 latent encodings using a 2D t-SNE projection 165

6.27 CIFAR-100 latent encodings using a 2D UMAP projection 166

6.28 MoE architecture with encoder-decoder routing 169

6.29 VAE example decodings for MNIST attention routing 172

6.30 UNet example decodings for MNIST attention routing 173

6.31 CIFAR-10 UNet example decodings for attention routing 178

6.32 CIFAR-100 UNet example decodings for attention routing 181

6.33 Example CIFAR-10 autoencoder encoding (SVD2D projection) 183

6.34 Example CIFAR-10 autoencoder decodings 184

6.35 Example CIFAR-10 MoE attention-based encoder projections (SVD2D) . . . 184

7.1 Example MNIST MoE trained with a fixed pre-trained classifier-router (Linear) 195

7.2 Example MNIST MoE trained with a fixed pre-trained classifier-router (RBF) 196

8.1 CIFAR-10 imbalanced experts mixture capabilities example 206

Abstract

In the field of artificial intelligence and machine learning (AI/ML), neural networks are a common and powerful statistical modeling tool. The neural architecture of such models describes how information is processed and decisions are made. The *Mixture-of-Experts* neural architecture is a metacognitive methodology applying the divide-and-conquer strategy to AI/ML. A *router* model learns "learns how to learn" where it defines subproblem decision boundaries (divide), whereby cognitive submodels (experts) are specialized (conquer). Thus, experts collectively provide a solution to the full problem space and can be passed samples independently or collaborate together.

The research questions explored in this dissertation explore leveraging the router and its understanding of domain meta-characteristics to address issues such as neural interpretability, training efficiency, and computation optimization. This work applies the MNIST, CIFAR-10, and CIFAR-100 image classification problem spaces to evaluate approaches to these issues and simultaneously provide metrics for common benchmarking datasets.

Interpretability is explored through the lenses of various 2D projection algorithms and attention-based routing to improve human understanding of model decisions, such as how subproblem regions are formed, which input features are relevant, and the behavioral relationships between experts. Training efficiency is addressed through transfer learning, where pretrained autoencoder and classifier models are adapted as router models, though experimental results fail to prove these strategies effective. Computation optimization is approached here as a complexity imbalance, where some subproblems are more difficult than others. Scaling up experts to handle increasingly complex subproblems enables dynamic inference, improving overall system performance without further increasing computational costs when handling simpler inputs.

Chapter 1

Introduction

1.1 Overview

Deep learning is a powerful subfield within the domain of artificial intelligence and machine learning (AI/ML) where neural networks are layered to form deep, cognitive models capable of effectively modeling complex problem spaces (Russell and Norvig, 2020, pg. 750). In traditional deep learning, a single model is used to handle the entire applied problem space. This is a fairly standard approach and can be scaled to billions of learnable parameters (Cheng et al., 2018) or even trillions of parameters (Sharda et al., 2025). The *Mixture-of-Experts* (MoE) architecture, by contrast, is a metacognitive deep learning methodology that details a divide-and-conquer strategy (Jordan and Jacobs, 1993). In MoE, there are two components: a metacognitive *router* model that learns subproblem space decision boundaries that break down a task (divide) and a set of cognitive *expert* models that then specialize across those partitions (conquer). The metacognitive nature of the router initially led to questions about how such learned representations could be further utilized. Thus, this dissertation research focuses on exploring how the MoE methodology can be leveraged to address common deep learning issues, such as interpretability, training efficiency, and computational optimization.

The following sections outline the motivation for the research, then explore each topic, including the hypothesis/hypotheses for evaluating the approaches examined and a summary of the contribution.

1.2 Motivation

Intuitively, an individual trying to master multiple domains will be less effective than distributing that task across multiple persons. The MoE methodology applies the same concept to deep learning, though importantly shaped by the router's ability to define how those tasks should be distributed.

Deep learning models, or just "models," have been rapidly growing both technologically and in size to solve increasingly difficult challenges (Bellalta et al., 2024; Suvarna et al., 2026). For example, *large language models* (LLMs), which are massive natural language processing (NLP) neural networks, are being scaled up to become increasingly powerful (Lin, 2022; Jiang et al., 2026). Relatedly, the MoE methodology has been found to improve the scalability of larger models, such as large language models (LLMs) (Cai et al., 2025; Li et al., 2026). However, many problem spaces do not require such staggering model sizes. Because smaller models provide faster inference speeds, they can be beneficial for embedded systems such as radar tracking and denoising (Stringer et al., 2024; Dolinger, 2025), object detection and classification (Naeem et al., 2013; Soumyadeep et al., 2023; Che, 2025; Wang et al., 2025a), and other real-time signal processing challenges.

Thus, by narrowing the problem spaces to small-to-mid-sized spaces, the intent is to prioritize the relevance of results for them. Additionally, this enabled faster iterative design, testing, and evaluation to consider more broadly which approaches are effective or ineffective, and to test against a larger number of conditions, such as *routing algorithms* that define how a router weights expert selection.

1.3 Baselines

This work explores the MoE methodology on the MNIST (LeCun et al., 1998) and CIFAR-10 and CIFAR-100 (Krizhevsky, 2009) datasets to contextualize the research within small-to-

mid-sized models. Convolutional neural network (CNN) models are leveraged as both router and expert models, with the CIFAR-100 tests additionally using residuals to accommodate the increased model size required for that level of complexity (He et al., 2016). Baseline experiments are performed across a range of single-model classifiers (referred to simply as "classifiers"), and MoE classifiers are trained to benchmark behaviors and provide a basis for comparison to later experiments. Additionally, autoencoder baselines, whereby an encoder-decoder is trained to encode the input to a latent (salient) representation and then decode it into the original image, are trained for use in select experiments.

The MoE baselines are comparable to the classifier baselines, with moderately improved parameter efficiency but no obvious changes in effectiveness. One limiting factor is that the baselines were trained with a patience-timeout hyperparameter, which produced fairly consistent results between the classifier and MoE baselines and was thus assumed to be effective. Notably, the MoE baselines did not report an obvious difference when adding/removing experts. However, later sparse ablation testing demonstrated that extensive training is required to properly specialize all expert models, at which point they became increasingly effective as well. This remains a limiting factor in the reported MoE benchmarks.

1.4 Interpretability

One of the primary drawbacks of neural networks is that their behaviors are opaque (Räuker et al., 2023; Garouani et al., 2024). This fundamentally means it is difficult to explain, predict, or otherwise understand their decisions, which is broadly categorized here as issues of "interpretability." The MoE methodology inherently also suffers from these issues since it is built on neural networks. Some prior works have explored directly integrating interpretability into the MoE methodology (Yang et al., 2025). However, this approach does not leverage the *latent space* by which the router represents high-level features about the problem space. **Research question**: How can the interpretability of Mixture-of-Experts systems be im-

proved by analyzing the latent representations learned by routing mechanisms?

This research question is then broken into two hypotheses:

Hypothesis 1: The latent space learned by a router encodes meaningful subproblem structure that can be analyzed to improve interpretability through relational and geometric methods.

Hypothesis 2: The router can integrate attention-based mechanisms to emphasize important characteristics of input data for human interpretability and potentially improve performance by sharing more information with each expert.

Contributions: Interpretability framework using latent clustering, as well as hyperparameters for controlling cluster distributions. Additionally, a novel routing architecture using an attention-based mechanism to highlight input signal-feature importance.

Interpretability is explored in two parts. The first is via visualization methods of the latent space and soft-attention mechanisms emphasizing relevant input features for classification. The second is through a novel architecture that integrates attention into the router to emphasize important features in the image classification process. Visualization methods primarily rely on 2D projections using singular-value decomposition (SVD), T-distributed stochastic neighbor embedding (t-SNE), and uniform manifold approximation and projection (UMAP) (McInnes et al., 2020). This helps contextualize how the router defines subproblem boundaries, what features are useful for grouping inputs, and the relations between experts. Results for the visualization experiments find that visualization methods can be useful, but are impeded by more complex problem spaces and likely require tuning to be more meaningful. Results for the attention-based routing experiments find the approach works well for emphasizing important features.

1.5 Transfer learning

Transfer learning is an effective approach for improving the training process and reducing the time it takes for a model's performance to converge (Zhuang et al., 2020a). While training time is a general issue across deep learning, one drawback of the MoE methodology is the added overhead of the router and the learnable parameters across all experts (Mi et al., 2026), which increases training time. This is additionally worsened by the router needing to learn optimal decision boundaries before experts can fully converge. Thus, MoE stands to benefit greatly from reduced training time.

Research question: How can the training process of MoE architectures be optimized to reduce computational cost while preserving or improving expert performance?

Hypothesis 3: Pre-training a router can reduce overall training cost and enable efficient transfer learning by providing a stable latent reference frame for training and specializing the expert models.

Contribution: Experimental data indicating, at the tested scales, transfer learning and pre/post-trained routers are not notably more effective/efficient.

The approach here is to leverage pre-trained autoencoder and classifier models on the same-dataset problem spaces as routing baselines, for example, taking a MNIST autoencoder and using its encoder component as a router for an MoE system. Additionally, it is tested whether allowing the new router to learn or use a fixed representation continually is more effective. However, experimental results do not find any of the approaches to be effective/consistent.

1.6 Computational Optimization

Within a problem space, it is unlikely that all problems therein will have uniform complexity. For example, in the context of image classification, some classes may interfere more strongly

than others. Consider the MNIST dataset when comparing digits '1,' '7,' and '8'. Intuitively, it would make sense to confuse the '1' and '7' with each other more than the '8'. Thus, if there are imbalanced complexities across the dataset, then there are likely experts who may underperform if their specialization entails such regions. For a single-model approach, if some baseline performance is needed across each class, then typically the approach is either to increase model parameters to better model the full problem space or to re-weight the learning priority to achieve a more uniform approach. MoE offers more modular flexibility, where potentially only a subset of the system needs to be up-scaled. This enables dynamic inference, where large, complex (slow) problems are solved by larger experts and the small, simple problems by the smaller (fast) experts.

Research question: How can routing strategies be defined to dynamically control computational depth and resource usage on a per-input basis?

Hypothesis 4: Effective dynamic inference can be achieved by pre-training a MoE system, then up-scaling under-performing experts to improve system performance while using minimal resources to handle simpler problems.

Contribution: Novel dynamic-inference cost approach using variable-weighted experts to balance improving performance with keeping lower computational costs.

The approach here is to swap out experts with upscaled versions while keeping the routing decisions constant. Experiment results demonstrate that, in general, this can be a very effective approach. Still, it requires some design consideration in how the experts are upscaled, with some upscaling strategies not consistently effective.

1.7 Layout

The remainder of the dissertation will cover the background and methodologies used in the experiments, then the baseline experiments, followed by each of the topics discussed above, and finally a conclusions chapter summarizing the work. Notably, the background is split

across two chapters, with Chapter 2 – Background detailing AI/ML at a high level and Chapter 3 – Mixture-of-Experts (MoE) going into the MoE methodology in more detail. Chapter 4 – PyTorch Model Factory begins preparing an explanation of the underlying methodologies by introducing a library developed for this project: the Torch Model Factory (TMF), which is used to describe different model configurations and provide a high-level understanding of the implementations. Chapter 5 – Baseline Experiments details the baseline classifier, autoencoder, and MoE experiments and results. Chapter 6 – Interpretability, Chapter 7 – Transferring Latent Learning, and Chapter 8 – Dynamic Inference Paths then detail the three explored topics, experiments, hypothesis/hypotheses tests, and evaluations. Finally, Chapter 9 – Conclusions summarizes all of the work, findings, and results.

Chapter 2

Background

2.1 Overview

This chapter provides technical background on relevant deep learning methodologies to establish a foundation for the remainder of this work. Starting with basic concepts, such as what artificial neural networks (or simply neural networks) are and how they learn. Then common neural architectures and optimization methodologies. Finally, example problem spaces and applications are presented to demonstrate the usefulness of these methodologies.

2.2 Deep Learning

This section broadly describes the topology of the AI/ML field and how deep learning is situated within it. Then, it introduces neural networks and their basic features, as well as how to optimize them. Finally, it concludes by describing neural architectures and supporting operations used throughout the work.

2.2.1 AI/ML

The field of artificial intelligence and machine learning (AI/ML) is large and complex. While this dissertation focuses primarily on the subfield of deep learning, it is useful to briefly describe what AI/ML encompasses at a high level.

The field of *artificial intelligence* (AI) fundamentally studies how to build intelligent entities (Russell and Norvig, 2020, pg. 1). *Intelligence* can be broadly described as the

capability for an entity to make a decision, such as which action to perform, based on the context of an input, such as an environment. An extension of the field introduces the aspect of *machine learning* (ML), which then adds the capability to adapt the entity to learn or adapt itself beyond the point at which it is initially built (Russell and Norvig, 2020, pg. 2). Despite ML being a subfield of AI, they are commonly joined together for clarity as AI/ML.

An example subfield of ML includes *genetic algorithms* where entities "evolve" through reproduction between more successful entities to reinforce positive traits and "mutate" to attempt further improvements (Russell and Norvig, 2020, pg. 116). The field of *deep learning* is another field within the domain of ML. It is more relevant for this work, whereby biologically inspired *artificial neural networks* are formed and learn how to solve problems by performing an action and then observing the results. Notably, such entities are more commonly referred to as *models*, since such neural networks are statistical modeling tools that attempt to minimize/maximize an error/score, respectively, corresponding to some objective function.

AI/ML encapsulates the fields of *natural language processing* (NLP), *knowledge representation*, *automated reasoning*, *computer vision*, and *simulation*, and *robotics* (Russell and Norvig, 2020, pg. 2). Each of these is a deep field in and of itself. Furthermore, they frequently intersect, such as applying ML to computer vision tasks (Wang et al., 2025b; Malik et al., 2025), NLP (Sharma, 2021; Kumar et al., 2024), robotics (Arulkumaran et al., 2017; Kaur and Bawa, 2022), and knowledge representation (Vaswani et al., 2017).

Fundamentally, AI is the study of designing agents that can make optimal decisions (Russell and Norvig, 2020, pg. 34). This leads to technical challenges such as defining optimality and designing agents with sufficient intelligence to act accordingly. Additional challenges include *interpretability* and *explainability*, which allow humans to understand why an AI agent makes a decision. Some AI methods, such as *decision trees*, are inherently interpretable (Russell and Norvig, 2020, pg. 711-712, 719). However, other AI agents, such as neural networks, can obscure the decision-making process through the complex internal

mechanisms used to compute a solution (Zhang et al., 2021b).

Bias is another important issue in AI, where an agent fails to fit its behavior appropriately to the environment (Russell and Norvig, 2020, pg. 654-656). This can occur for several reasons. For example, an agent may be unable to model a complex environment sufficiently and may *underfit*, failing to capture important patterns in the environment (Russell and Norvig, 2020, pg. 655). Conversely, an agent may *overfit* by specializing too strongly to a subset of the environment and failing to generalize to the rest of it (Russell and Norvig, 2020, pg. 655). In AI/ML contexts, this often occurs when a model overfits the *training data* from which it learns its behavior. Learned bias can also arise from external sources: if the learning method itself is biased, then the model's learned behavior can also be biased (Hoyt et al., 2025).

2.2.2 Neural Networks

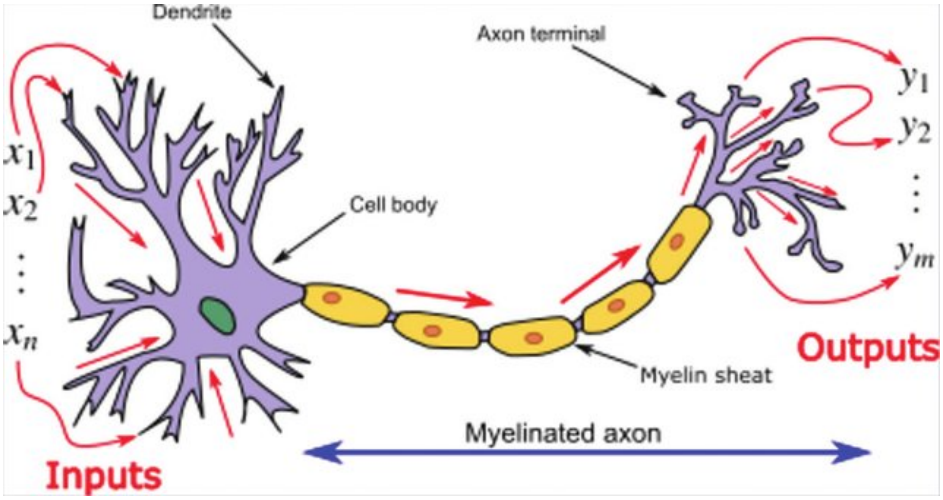


Figure 2.1: Biological Neuron Example (Fahmi et al., 2025, Figure 2): This visualization illustrates how biological neural networks inspire artificial neural networks. Biological neural networks contain interconnected neurons that transmit signals from synapses to other neurons' dendrites through complex electrochemical processes (Russell and Norvig, 2020, pg. 12). An artificial neural network emulates this by passing the input vector X to the dendrites and producing the output vector Y at the axon terminal, which can then be fed to another neural layer/dendrite.

Artificial neural networks are an AI/ML methodology inspired by biological neural net-

works, as depicted in Figure 2.1. The motivation behind the methodology is that biological brains solve complex problems by processing information across massive networks of neurons; therefore, emulating that behavior may provide an effective methodology for training AI/ML agents. This originated in the *perceptron*, which maps some input vector $\mathbf{x}$ with length n by some binary function f_{binary} given a set of weight parameters w to some output o.

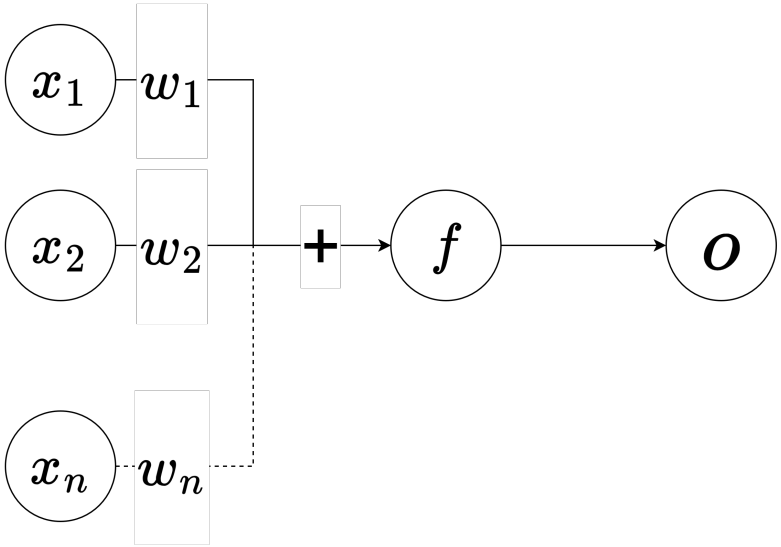


Figure 2.2: Artificial Neuron Example (Linear): The artificial neuron takes in signal $\mathbf{x}$, which is a feature vector with length n. Each feature $\mathbf{x_i}$ is weighted by a learned parameter $\mathbf{w_i}$ in a linear transform, where the result is summed and mapped by an activation function f to 1 or 0.

The following equation gives the perceptron.

$$o = f(\mathbf{x}; \mathbf{w}) = f_{\text{binary}} \left(\sum_{i=1}^{n} \mathbf{x_i} \mathbf{w_i} \right) \tag{2.1}$$

2.2.2.1 Activation Functions

The binary function here provides an important point of nonlinearity. If the output is linearly determined, then the model can only reliably solve linear problems. While the binary function satisfies this nonlinear requirement, more common activation functions include *ReLU*,

sigmoid, and *tanh* (Russell and Norvig, 2020, pg. 752). The term *activation* originates, once again, from the biological neuron: a signal is propagated across the axon and to the synapse only if the input signal *activates* the neuron. The following equations define these activation functions.

$$\mathbf{y} = \text{ReLU}(\mathbf{x}) = \max(0, \mathbf{x}) \quad \mathbf{y} \in [0, x) \tag{2.2}$$

$$\mathbf{y} = \sigma(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{x}}} \quad \mathbf{y} \in (0, 1) \tag{2.3}$$

$$\mathbf{y} = \tanh(\mathbf{x}) = \frac{e^{\mathbf{x}} - e^{-\mathbf{x}}}{e^{\mathbf{x}} + e^{-\mathbf{x}}} \quad \mathbf{y} \in (-1, 1) \tag{2.4}$$

While all three are common activation functions, ReLU is especially prominent because of its simplicity and effectiveness (Sára Pusztaházi et al., 2024). However, the output range of an activation function can be more directly useful in some domains than in others. For example, the sigmoid activation produces a signal in $(0, 1)$, making it useful as a likelihood estimate (Singh et al., 2023).

2.2.2.2 Optimization

The immediate challenge is then how to learn the weights, or parameters, of a neural network. The simple, brute-force approach would be to search the entire parameter space for an optimal parameter setting. However, neural-network parameters are usually continuous values. Even if these values are discretized into finite ranges, the search would still scale exponentially with the number of parameters, making it increasingly infeasible as the model size grows.

At very small parameter scales, a more efficient search may be possible through methodologies such as *genetic algorithms*. Genetic algorithms are inspired by biological processes of natural selection and evolution (Floreano and Mattiussi, 2008, pg. 33) (Russell and Norvig, 2020, pg. 111-117) and can be effective for small optimization problems. However, the

exponential scaling of the parameter search space is a problem for large neural networks. Tangentially, some works use genetic algorithms to identify neural-network architectures instead of directly learning all model parameters, providing higher-level architectural control while relying on more efficient learning methods to train the resulting model parameters (Cătrună and Iancu, 2023).

Instead of random or exhaustive searches, the more common approach is to use local gradient information to estimate the direction and magnitude of parameter updates. This is done through *backpropagation*, which efficiently computes loss gradients for neural-network parameters and is discussed in the following subsections.

2.2.2.2.1 Programming Libraries. Backpropagation, partial derivatives, and the chain rule become increasingly complex as model size and architectural complexity grow. Most programming frameworks used for AI/ML research support *automatic differentiation*, providing libraries that automatically and efficiently compute loss gradients (Russell and Norvig, 2020, 756). This makes neural-network training more accessible and allows greater research emphasis on higher-level topics such as architectural design and application.

This work therefore reviews the basic concepts, keeping the focus on higher-level AI/ML concepts.

2.2.2.2.2 Gradient Descent. For neural networks, the more common solution is to use backpropagation to compute parameter gradients with respect to an *objective function* (Ruder, 2017). The objective function describes the priorities of the agent and what it should be doing (Russell and Norvig, 2020, pg. 110). In neural network optimization, the objective is commonly defined using a *loss function*, often averaged over training examples. The loss function measures a differentiable error that relates the model's current behavior to some desired behavior. This error can then be differentiated to identify an *error gradient*, which identifies the local direction of steepest increase in loss; gradient descent updates parameters in the opposite direction to reduce the loss. This method is known as *gradient descent*, and

it is a common approach for training neural networks.

Let the input signal be denoted by x, the model parameters by θ, the activation or output function by f, and the loss function by $\mathcal{L}$. A simplified linear case is given below.

$$z = \theta x, \tag{2.5}$$

$$\hat{y} = f(z) = f(\theta x), \tag{2.6}$$

$$\ell = \mathcal{L}(\hat{y}, y). \tag{2.7}$$

Here, z is the pre-activation model output, $\hat{y}$ is the output prediction, and ℓ is the scalar loss.

Backpropagation computes the gradient of the loss with respect to the model parameter θ. The chain rule gives the following expression.

$$\frac{\partial \ell}{\partial \theta} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial \theta} \tag{2.8}$$

The component derivatives are given below.

$$\frac{\partial \hat{y}}{\partial z} = f'(z), \tag{2.9}$$

$$\frac{\partial z}{\partial \theta} = x, \tag{2.10}$$

Substituting these derivatives gives the following gradient.

$$\frac{\partial \ell}{\partial \theta} = \frac{\partial \ell(\hat{y}, y)}{\partial \hat{y}} f'(z) x. \tag{2.11}$$

Note, however, the loss gradient is commonly denoted as $\nabla_\theta \mathcal{L}(\theta)$. Here, ∇_θ refers to the gradient in the parameter space of θ, taken with respect to the loss evaluated over those parameters, $\mathcal{L}(\theta)$.

$$\nabla_\theta \mathcal{L}(\theta) \equiv \frac{\partial \ell}{\partial \theta} \tag{2.12}$$

This loss gradient $\nabla_\theta \mathcal{L}(\theta)$ describes a direction in which the model parameters can move to decrease the loss. However, this becomes a complex form of local search: the local slope and direction of improvement are known, but the distance to the optimum and whether the local optimum is globally best are not known (Russell and Norvig, 2020, pg. 111-112).

Thus, neural networks are typically trained over several epochs, iteratively exploring the optimization space. At each optimization step, a loss gradient is computed and the model *learns* with respect to a *learning rate* parameter η. This parameter controls the magnitude of the parameter update, helping prevent overshooting while still allowing reasonable progress.

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta) \tag{2.13}$$

However, there is a question of whether the solution found through gradient descent is optimal.

While an error/loss gradient details how model parameters should be moved to reduce error/loss, a major issue for this optimization process is that nonlinear neural-network loss functions are typically *nonconvex*, making the entire optimization space also nonconvex (Goodfellow et al., 2016, pg. 171). Figure 2.3 illustrates this distinction by comparing a convex surface with a nonconvex surface. In a convex optimization space, the objective has a globally consistent basin, so local descent directions reliably lead toward the global optimum. In a nonconvex space, by contrast, the surface may contain multiple local minima, local maxima, saddle points, and irregular regions. Therefore, the local gradient may identify a direction of immediate improvement without guaranteeing that the resulting solution is globally optimal.

Consider that the search space for the optimal solution is within a real space of dimensionality equal to the number of parameters in θ. Assuming there are N parameters, this

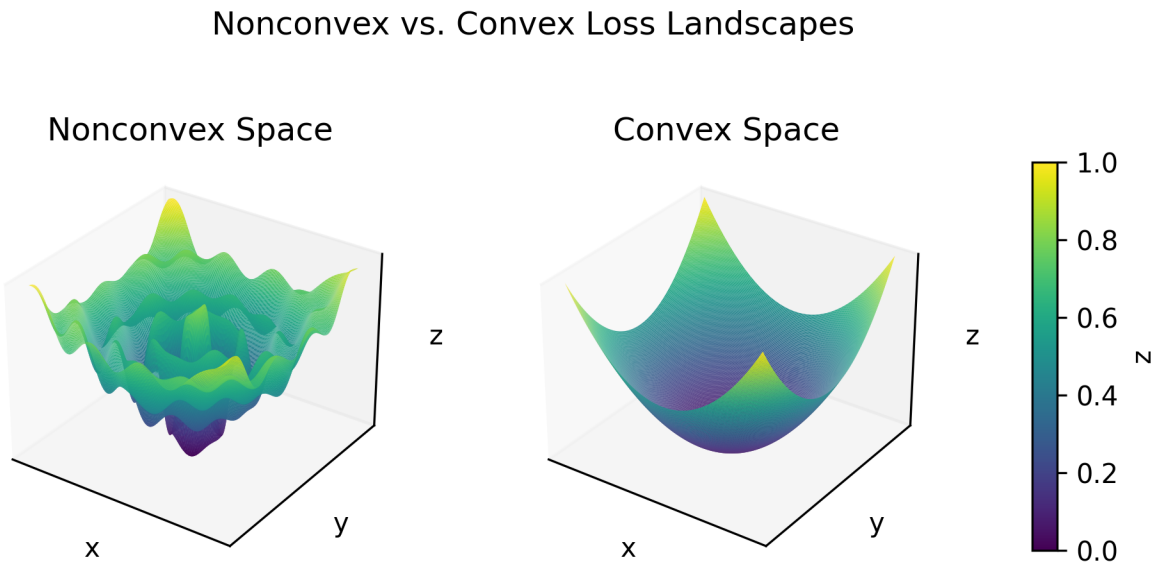


Figure 2.3: Comparison of nonconvex (left) and convex (right) 3D surfaces. The convex surface contains a single smooth basin, while the nonconvex surface contains multiple peaks, valleys, and local extrema. The nonconvex visualization is a low-dimensional representation of an optimization space for a nonlinear neural network, demonstrating the complexity of the optimization challenge.

gives a space of $\mathbb{R}^N$. While this search space is large and inefficient for methods such as genetic algorithms to explore, its high dimensionality can make it easier for gradient-based approaches to find a path toward a reasonable solution, even if that solution is not globally optimal (Welch et al., 2025).

Though this method of optimization is very effective, there still are several issues, such as computational complexity. These are discussed in the following subsections.

2.2.2.2.3 Stochastic Gradient Descent. Note that full gradient descent is not the usual way in which parameters are optimized in neural networks given a training dataset $\mathcal{D}$ with M samples, where each sample is a tuple between the input $\mathbf{x_i}$ and target $\mathbf{y_i}$: $\{(\mathbf{x_1}, \mathbf{y_1}), ..., (\mathbf{x_M}, \mathbf{y_M})\} = \mathcal{D}$. Then, the empirical loss over the full dataset can be written as follows.

$$\mathcal{L}(\theta) = \frac{1}{M} \sum_{i=1}^{M} \ell(f_\theta(\mathbf{x_i}), \mathbf{y_i}) \tag{2.14}$$

Here, f_θ is the model parameterized by θ, and ℓ is the loss for a single example.

A classical gradient descent update computes the gradient over the entire dataset before applying one optimization step, as shown below.

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta) = \theta - \eta \frac{1}{M} \sum_{i=1}^{M} \nabla_\theta \ell(f_\theta(\mathbf{x_i}), \mathbf{y_i}) \tag{2.15}$$

However, this can be computationally expensive for large datasets, since every update requires evaluating the loss and gradient across the full training set. *Stochastic gradient descent* (SGD) partially addresses this issue by estimating the full gradient using only one randomly (uniformly) sampled training example at a time. For a randomly selected example $(\mathbf{x_i}, \mathbf{y_i})$, the SGD update is then defined below.

$$(\mathbf{x_i}, \mathbf{y_i}) \sim \text{Uniform}(\mathcal{D}) \qquad \theta \leftarrow \theta - \eta \nabla_\theta \ell(f_\theta(\mathbf{x_i}), \mathbf{y_i}) \tag{2.16}$$

Here, $\mathbf{x_i}, \mathbf{y_i}$ are uniformly sampled from the dataset $\mathcal{D}$.

This update is much cheaper to compute per optimization step. Still, it is also noisier because the gradient from one sample may not point in the same direction as the gradient over the full dataset. In expectation, when the sampling is uniformly random, the stochastic gradient provides an estimate of the full gradient.

$$\mathbb{E}_i \left[\nabla_\theta \ell(f_\theta(\mathbf{x_i}), \mathbf{y_i}) \right] = \nabla_\theta \mathcal{L}(\theta) \tag{2.17}$$

Thus, SGD trades a more expensive but stable full-dataset gradient for a cheaper but noisier gradient estimate. This noise can cause the loss to fluctuate during training and can lead to less stable learning behaviors (Ruder, 2017). However, common extensions such as momentum can help keep SGD moving in a more consistent direction (Ruder, 2017). Such optimization variations are common to provide fine-tuning and improvements, often when there is an underlying need for certain behavioral characteristics. Such variations are discussed later in this chapter.

Notably, the pure form of SGD described above is less common than *mini-batch gradient descent*. Mini-batch gradient descent samples a batch $B \subset \mathcal{D}$ (when sampled without replacement) of size $|B|$, then averages the loss gradients over that batch, as given below.

$$\theta \leftarrow \theta - \eta \frac{1}{|B|} \sum_{(\mathbf{x_i}, \mathbf{y_i}) \in B} \nabla_\theta \ell(f_\theta(\mathbf{x_i}), \mathbf{y_i}) \tag{2.18}$$

This provides a middle ground between full gradient descent and pure SGD. The mini-batch gradient is less noisy than a single-sample estimate, but it is much cheaper than computing the gradient over the entire training dataset. Therefore, when SGD is discussed in modern deep learning, it often refers to mini-batch SGD, where models are trained over randomly sampled batches of data instead of individual samples or the full dataset (Ruder, 2017).

2.2.2.2.4 Adam Optimizer. A more commonly used variation of the SGD optimizer is the *adaptive moment estimation* (*Adam*) optimizer (Kingma and Ba, 2017). Adam is

a first-order stochastic optimization method that extends SGD. It applies momentum and adaptive learning-rate scaling by maintaining moving averages of both the gradient and the squared gradient. The first moment estimates the mean direction of the gradient, while the second moment estimates its uncentered variance. These estimates are then used to adapt the effective learning rate for each parameter individually. Notably, this allows Adam to combine the benefits of momentum-based optimization with per-parameter learning rate scaling, often making it more stable and less sensitive to the initial learning rate than plain SGD. As a result, the Adam optimizer is a popular choice in deep learning (Zhang, 2018; Ji, 2024) and is often used as the default optimization methodology. However, while Adam often converges more quickly than plain SGD, it is not universally more stable and does not always generalize better (Gupta et al., 2021; Manataki et al., 2021).

Adam has further inspired a family of similar optimizers. Such Adam-style optimizers can be grouped by which part of Adam they modify: the momentum estimate, the adaptive second-moment scaling, or the parameter decay rule.

AdamW decouples weight decay from the gradient update, making regularization more consistent under adaptive scaling (Loshchilov and Hutter, 2019). AMSGrad alters the second-moment estimate by using a non-decreasing maximum to improve convergence behavior (Reddi et al., 2018), while AdaMax, initially presented alongside Adam, replaces the second-moment scaling with an infinity-norm variant (Kingma and Ba, 2017). Nadam modifies the momentum term using Nesterov-style lookahead (Li et al., 2021), and RAdam rectifies Adam's adaptive step size during early training to reduce instability (Liu et al., 2021). AdaBound constrains Adam's effective learning rates so that training gradually behaves more like SGD (Luo et al., 2019), AdaBelief changes the variance estimate to track unexpected gradient deviations (Zhuang et al., 2020b), and LAMB adds layer-wise update scaling for large-batch training (You et al., 2020).

2.2.3 Neural Architectures

The previous section introduced neural networks, activation functions, and optimization. This section extends that discussion by additionally introducing neural architecture layers: the structural choices used to organize layers, process input signals, and support learning. Different architectures provide different inductive biases, allowing them to attenuate, emphasize, or learn certain input characteristics more effectively than others. For example, in image processing, a network that does not consider spatial feature properties may have more difficulty generalizing spatial patterns than an architecture that explicitly processes local spatial structure.

The research in this work builds heavily on neural architectural exploration. Therefore, this section focuses on the architectural components used later and provides the context needed to understand how more complex models function. First, basic neural networks are discussed, including *linear/dense* layers, *convolutional neural networks* (CNNs), and *residuals*. Linear/dense layers provide a basic fully connected formulation, CNNs are well-suited for spatial pattern recognition, and residual connections can shorten gradient paths through a network.

This section then considers convolutional *encoder-decoder* architectures, which encode input signals into a latent space and then decode that information, optionally transforming it in the process. A specific variation of the encoder-decoder methodology is the *autoencoder*, which attempts to reconstruct the original input signal from a salient latent representation. For these encoder-decoder and autoencoder architectures, this work examines basic convolutional implementations, *variational autoencoders* (VAEs) (Kingma and Welling, 2019; Razghandi et al., 2022), and *UNets* (Siddique et al., 2021).

Finally, this section covers auxiliary layers and operations with useful properties for neural networks, including *dropout*, *normalization*, *pooling*, and *flatten* functions.

2.2.3.1 Neural Network Layers

This section introduces the primary learnable layers used in the architectures discussed later. It first describes linear or dense layers and then convolutional layers, emphasizing their computations, parameterization, and suitability for vector and spatially structured inputs, respectively.

2.2.3.1.1 Linear Layers. A linear, or dense, neural network layer applies an affine transformation to an input vector. Although such layers are commonly called linear layers, the inclusion of a bias term makes the transformation affine instead of strictly linear. For an input $\mathbf{x} \in \mathbb{R}^{d_{\text{in}}}$, where d details some dimensionality, a dense layer produces the following output.

$$\mathbf{h} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{2.19}$$

Here, $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ is a learned weight matrix, $\mathbf{b} \in \mathbb{R}^{d_{\text{out}}}$ is a learned bias vector, and $\mathbf{h} \in \mathbb{R}^{d_{\text{out}}}$ is the resulting representation. The bias term allows the transformation to shift the output instead of forcing it to pass through the origin. With a bias term included, this layer has $d_{\text{out}}d_{\text{in}} + d_{\text{out}}$ learnable parameters.

Figure 2.4 illustrates the matrix-vector computation performed by a dense layer. Each row of the weight matrix defines the learned coefficients for one output unit, so every output entry is formed by combining all input features and then adding the corresponding bias term.

Dense layers are useful when the input is naturally represented as a vector or when previous architectural components have already extracted a compact representation. For example, convolutional feature maps, discussed in the next section, are often flattened or pooled before being passed to dense layers for classification. Dense layers are also commonly used in latent spaces and output logit heads.

However, because each output unit is connected to each input unit, dense layers can require many parameters for high-dimensional inputs. They also do not explicitly encode

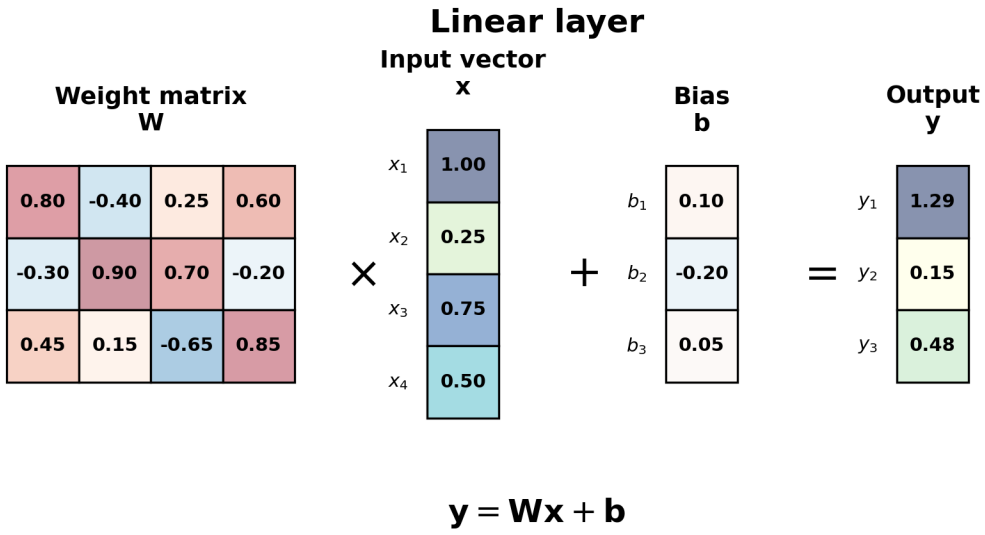


Figure 2.4: Linear layer example. Visualization of a dense layer applied to an input vector. The learned weight matrix $\mathbf{W}$ is multiplied by the input vector $\mathbf{x}$, the learned bias vector $\mathbf{b}$ is added, and the result is the output vector. This illustrates the affine transformation $\mathbf{h} = \mathbf{W}\mathbf{x} + \mathbf{b}$, where each output entry is computed from all input features.

spatial locality, channel structure, weight sharing, or translation-related structure. As a result, when the input signal has meaningful spatial organization, dense layers may be less parameter efficient than alternative architectures that specifically leverage those features (Bishop and Bishop, 2024, pg. 172-173).

2.2.3.1.2 Convolutional Layers. Convolutional layers process structured signals using local *kernels*, or *filters*, with shared parameters (Chauhan et al. 2018, pg. 287; Bishop and Bishop 2024). Rather than learning an independent weight for every input-output connection, a convolutional layer applies the same learned kernel across different spatial locations. This parameter sharing allows the layer to detect similar local patterns regardless of where they occur in the input.

For a two-dimensional input tensor $\mathbf{X} \in \mathbb{R}^{C_{\text{in}} \times H \times W}$ with C_{in} input channels, a convolutional layer with C_{out} output channels produces output feature maps $\mathbf{Y}$. Omitting padding,

stride, and dilation for notational simplicity, the operation can be written as follows.

$$Y_{c_{\text{out}},i,j} = b_{c_{\text{out}}} + \sum_{c=1}^{C_{\text{in}}} \sum_{u=0}^{k_h-1} \sum_{v=0}^{k_w-1} K_{c_{\text{out}},c,u,v} X_{c,i+u,j+v} \tag{2.20}$$

Here, $\mathbf{K} \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times k_h \times k_w}$ is the learned kernel tensor, $\mathbf{b} \in \mathbb{R}^{C_{\text{out}}}$ is the learned bias vector, and k_h and k_w denote the kernel height and width. With unit stride and no padding, the spatial indices i and j range over positions where the kernel lies entirely inside the input. In this case, the output has spatial size $(H - k_h + 1) \times (W - k_w + 1)$.

Note that Equation 2.20 actually describes cross-correlation instead of mathematical convolution, since the kernel is not flipped. However, this operation is commonly referred to as convolution in neural network literature and software libraries.

Furthermore, *padding*, *stride*, and *dilation* are often added to control the spatial size and receptive field of the resulting feature maps (Paszke et al., 2017, torch.nn.Conv2d). Padding expands the dimensions of the input, with various fill strategies such as zeros, random noise, interpolating the edges, reflecting the input, or wraparound. Stride determines the step size used to convolve the input with the kernel iteratively; how many pixels to skip between convolutions. Dilation determines the spacing between the kernel points in the output. The output shape of a CNN can therefore be expressed as follows.

$$S_{\text{out}} = \left\lfloor \frac{S_{\text{in}} + 2 * \text{padding_size}_S - \text{dilation}_S(\text{kernel_size}_S - 1) - 1}{\text{stride_size}_S} + 1 \right\rfloor \tag{2.21}$$

Here, S is an arbitrary axis. Note that some configurations such as kernel_size = 3, padding_size = 1, stride_size = 1, and dilation = 0 produce a convolved output whose size along that axis is identical to the input.

Such convolutions are useful for creating deeper networks without reducing the spatial information available. Typically, when downsampling the signal size, stride_size = 2 is used since it halves the number of pixels along that axis when the number of pixels is even, the number of convolution steps, and thus halves the output size. Alternative forms of down-

sampling include pooling layers, discussed later in Section 2.2.3.3, which also incorporate stride sizes to similar effect.

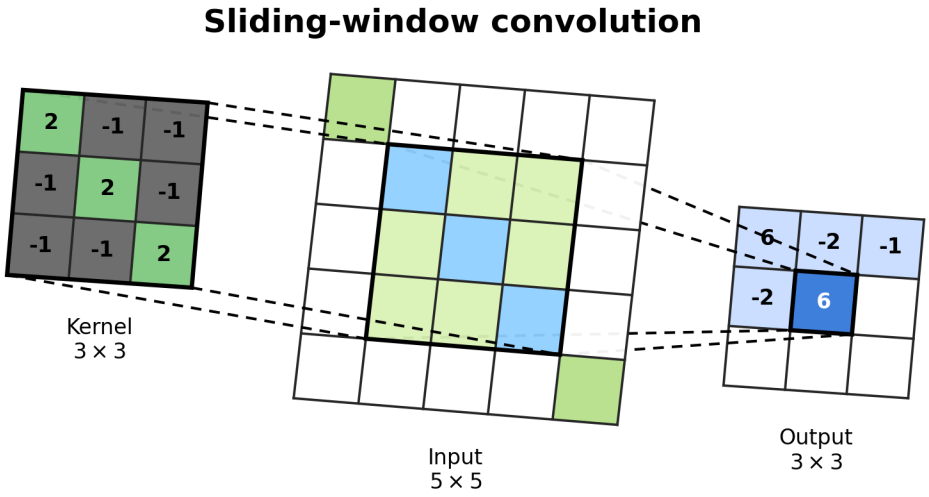


Figure 2.5: Convolution layer example. Sliding-window convolution applied to a two-dimensional input. A shared 3×3 kernel is applied to a local 3×3 input patch, producing one value in the output feature map. As the kernel moves across the input, the same weights are reused at each spatial location, generating the remaining output values. Notably, the output indicates the sliding window has already covered the first row and the first column of the second row.

Figure 2.5 illustrates the local receptive-field structure of a convolutional layer. At each spatial position, the kernel overlaps a small input patch, the corresponding entries are multiplied and summed, and the result is written to one location in the output feature map. This sliding-window operation demonstrates the key form of parameter sharing in convolutional layers: the same kernel weights are reused across spatial positions instead of learning separate weights for each location. This allows translational invariance within the image space, where an object can be moved to an arbitrary position so long as it is equally captured (not cut off) and the convolution, ignoring other image features, will similarly model that into the feature map, likely then also spatially offset therein. Note that this does not support all linear transforms, such as skewing and rotation.

Parameter sharing gives convolutional layers an architectural bias toward learning local

spatial patterns. This is especially useful for image processing, where nearby pixels are often strongly related, and the same visual pattern may occur in different locations (Bishop and Bishop, 2024, pg. 291). Lower convolutional layers often learn localized features, while deeper layers can combine these features into more abstract representations.

Compared with dense layers applied directly to high-dimensional spatial inputs, convolutional layers can substantially reduce the number of learned parameters while preserving spatial structure. However, this locality assumption can also be a limitation. A small convolutional kernel only observes a local neighborhood at each layer, so broader context must be captured through depth, pooling, larger kernels, dilation, or other architectural mechanisms.

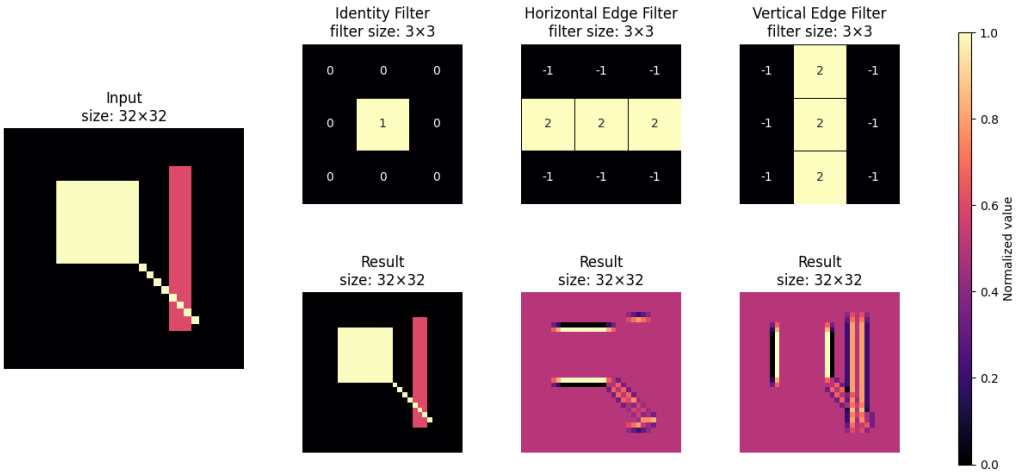


Figure 2.6: Convolution filters example. Visualization of simple convolutional filters applied to a two-dimensional input image. The input image is shown once on the left, the top row details the identity, horizontal-edge, and vertical-edge kernels, and the bottom row details the corresponding filtered outputs.

Figure 2.6 visualizes how simple convolutional filters transform a two-dimensional image by applying local weighted patterns across spatial locations. The input image is shown on the left, while the top row displays three 3×3 kernels and the bottom row details the corresponding output feature maps. The identity filter preserves the original image structure, whereas the horizontal- and vertical-edge filters emphasize edge structures with different spatial orientations.

2.2.3.2 Encoder-Decoders

Encoder-decoder architectures are neural networks that first compress (encode) an input signal into a salient representation (latent space) and then expand, reconstruct, or transform that representation into an output signal (Zhai et al., 2018). In general, an encoder E_ϕ maps an input $\mathbf{x}$ into a latent representation $\mathbf{z}$, and a decoder D_ψ maps that latent representation to an output $\hat{\mathbf{y}}$. These mappings are defined below.

$$\mathbf{z} = E_\phi(\mathbf{x}), \qquad \hat{\mathbf{y}} = D_\psi(\mathbf{z}) \tag{2.22}$$

Here, ϕ and ψ denote the learned encoder and decoder parameters, respectively. The latent representation $\mathbf{z}$ is often lower-dimensional so that it captures salient information from the original input.

In convolutional encoder-decoder architectures, the encoder commonly uses pooling or strided convolutions to reduce spatial resolution while increasing the number of feature channels. This process allows the encoder to summarize local spatial patterns into progressively more abstract features. The decoder works in the other direction, increasing spatial resolution and converting the latent representation back into an output with the desired shape. Depending on the task, the output may be a reconstruction of the input, a transformed image, a segmentation mask, or another structured signal.

2.2.3.2.1 Autoencoders. An autoencoder is a specific encoder-decoder architecture trained to reconstruct its own input. In this case, the desired output is $\mathbf{x}$ itself. The model output is given below.

$$\hat{\mathbf{x}} = \mathbf{y} = D_\psi(E_\phi(\mathbf{x})) \tag{2.23}$$

The autoencoder objective below encourages the decoded output $\hat{\mathbf{x}}$ to be close to the original input $\mathbf{x}$.

$$\mathcal{L}_{\text{AE}}(\phi, \psi) = \ell(\hat{\mathbf{x}}, \mathbf{x}) = \ell(D_\psi(E_\phi(\mathbf{x})), \mathbf{x}) \tag{2.24}$$

Here, $\ell(\cdot)$ is typically a reconstruction loss such as mean squared error for continuous-valued inputs or binary cross-entropy for normalized binary-like inputs (Khade et al., 2023).

The usefulness of an autoencoder is its ability to both compress information to a meaningful representation and then decode that into an approximate reconstruction. The greater the compression, the more useful it is. However, if a latent space is too small, then it is likely too much information will be lost. Conversely, a latent space so large that barely, if any, compression is needed is not a useful one.

2.2.3.2.2 Up-sampling with Transposed-Convolutions. A decoder typically needs to increase spatial resolution after the encoder has compressed the input. One common learned approach is the transposed convolution, sometimes informally called a de-convolution (Dumoulin and Visin, 2018). Additionally, despite the name, a transposed convolution is not the mathematical inverse of a convolution; it is a learnable operation that maps a lower-resolution feature map to a higher-resolution feature map by applying trainable kernels in a way that expands spatial dimensions. Effectively a reversed-convolution of sorts.

For an input feature map with spatial size $H_{\text{in}} \times W_{\text{in}}$, kernel size $k_h \times k_w$, stride $s_h \times s_w$, padding $p_h \times p_w$, dilation $d_h \times d_w$, and output padding $o_h \times o_w$, the spatial output size of a transposed convolution is commonly given by the following expressions.

$$H_{\text{out}} = (H_{\text{in}} - 1)s_h - 2p_h + d_h(k_h - 1) + o_h + 1, \tag{2.25}$$

$$W_{\text{out}} = (W_{\text{in}} - 1)s_w - 2p_w + d_w(k_w - 1) + o_w + 1 \tag{2.26}$$

Notably, the stride can be used to increase spatial resolution while the learned kernel weights determine how features are distributed into the expanded output.

Figure 2.7 illustrates how a transposed convolution expands spatial resolution. Unlike a standard convolution, where a local input patch is reduced to one output value, a transposed convolution maps each input value to a spatial patch in the output. The selected input cell scales the kernel weights, and those scaled values are added into the corresponding

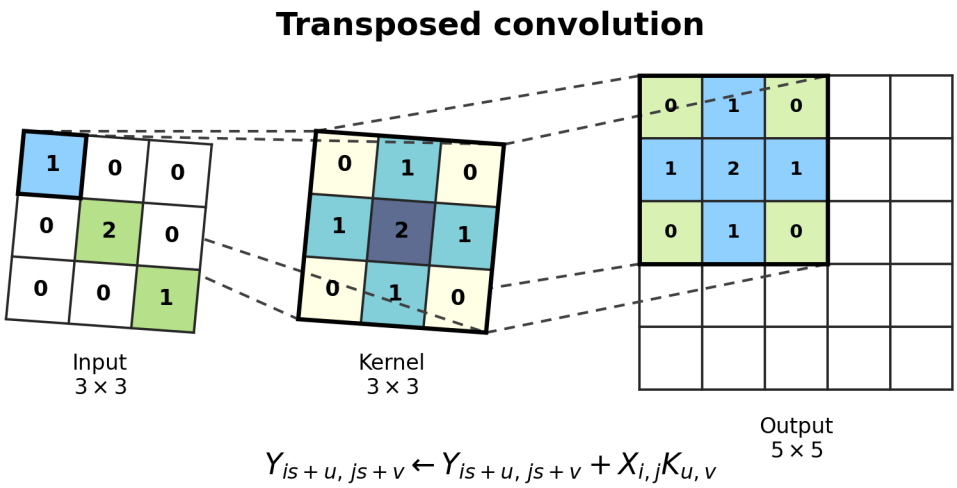


Figure 2.7: Transposed-convolution layer example. Visualization of a transposed convolution operation. A low-resolution input value scales a learned kernel, and the resulting kernel-shaped contribution is added into a larger output feature map. The highlighted output region details the spatial patch affected by the selected input cell, while the displayed output values show the accumulated contributions up to that step.

output locations. When multiple input cells contribute to overlapping output regions, their contributions are summed, allowing the decoder to learn how lower-resolution feature maps should be distributed into a higher-resolution representation. However, this up-sampling methodology is known for creating artifacts through how the kernels are applied to the output (Odena et al., 2016; Aitken et al., 2017).

2.2.3.2.3 VAEs. A variational autoencoder (VAE) extends the autoencoder framework by learning a probabilistic latent representation instead of a single deterministic latent vector (Kingma and Welling, 2019; Razghandi et al., 2022). Instead of producing only $\mathbf{z} = E_\phi(\mathbf{x})$, the encoder parameterizes an approximate posterior distribution over latent variables. The distribution is defined below.

$$q_\phi(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}\left(\boldsymbol{\mu} * \phi(\mathbf{x}), \mathrm{diag}(\boldsymbol{\sigma} * \phi^2(\mathbf{x}))\right) \tag{2.27}$$

Here, $\boldsymbol{\mu} * \phi(\mathbf{x})$ and $\boldsymbol{\sigma} * \phi^2(\mathbf{x})$ are learned functions of the input. A latent sample is then obtained using the reparameterization trick.

$$\mathbf{z} = \boldsymbol{\mu} * \phi(\mathbf{x}) + \boldsymbol{\sigma} * \phi(\mathbf{x}) \odot \boldsymbol{\epsilon}, \qquad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{2.28}$$

Because the noise term $\boldsymbol{\epsilon}$ is sampled independently of the model parameters, gradients can flow through the deterministic functions $\boldsymbol{\mu} * \phi(\mathbf{x})$ and $\boldsymbol{\sigma} * \phi(\mathbf{x})$ during backpropagation.

The VAE objective is commonly written as the negative evidence lower bound, combining a reconstruction term with a regularization term that encourages the approximate posterior to be close to a prior distribution, typically $p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$. The objective is given below.

$$\mathcal{L} * \mathrm{VAE}(\phi, \psi) = \mathbb{E} * q_\phi(\mathbf{z} \mid \mathbf{x}) \left[-\log p_\psi(\mathbf{x} \mid \mathbf{z}) \right] + D_{\mathrm{KL}} \left(q_\phi(\mathbf{z} \mid \mathbf{x}) \| p(\mathbf{z}) \right) \tag{2.29}$$

The first term penalizes poor reconstruction under the decoder likelihood $p_\psi(\mathbf{x} \mid \mathbf{z})$, while the second term is the Kullback-Leibler divergence (Kullback and Leibler, 1951) between the approximate posterior and the prior. For the diagonal Gaussian posterior above with a standard normal prior, this KL term has the following closed form.

$$D_{\mathrm{KL}} \left(q_\phi(\mathbf{z} \mid \mathbf{x}) \| \mathcal{N}(\mathbf{0}, \mathbf{I}) \right) = \frac{1}{2} \sum_j \left(\mu_j^2 + \sigma_j^2 - \log \sigma_j^2 - 1 \right) \tag{2.30}$$

This regularized latent space makes VAEs useful not only for reconstruction, but also for representation learning and generative modeling.

2.2.3.2.4 UNets. UNets are encoder-decoder architectures that add skip connections between corresponding encoder and decoder stages (Siddique et al., 2021). Like a standard convolutional encoder-decoder, a UNet first reduces spatial resolution through an encoder path and then increases spatial resolution through a decoder path. The key difference is that feature maps from earlier encoder stages are passed directly to decoder stages at matching

resolutions.

A simplified UNet decoder stage can be written as follows.

$$\mathbf{d}_\ell = G_\ell \left(\text{concat} \left(\text{up}(\mathbf{d}_{\ell+1}), \mathbf{e}_\ell \right) \right) \tag{2.31}$$

Here, $\mathbf{e}_\ell$ is an encoder feature map, $\mathbf{d}_{\ell+1}$ is a deeper decoder representation, $\text{up}(\cdot)$ denotes an up-sampling operation, $\text{concat}(\cdot)$ denotes channel-wise concatenation, and G_ℓ is a learned convolutional transformation.

These skip connections help preserve spatial detail that might otherwise be lost in the encoder bottleneck. This is especially useful for tasks where the output must be spatially aligned with the input, such as segmentation, reconstruction, and image-to-image translation.

2.2.3.3 Auxiliary Functions and Operations

In addition to the primary architectural layers described above, neural networks often use auxiliary functions that modify how information is represented, normalized, regularized, or reshaped as it moves through the model. These operations are important because they can improve optimization stability, reduce overfitting, control spatial resolution, or connect incompatible tensor shapes between different parts of a network. This section looks at such components used later in this work, namely *dropout*, *normalization*, and *pooling* layers.

2.2.3.3.1 Dropout. Dropout is a regularization method that randomly removes activations during training (Srivastava et al., 2014). It is one of the simplest and most effective forms of regularization and is widely used to help prevent overfitting (Bishop and Bishop, 2024, pg. 281).

Given hidden activations $\mathbf{h}$, inverted dropout can be written as follows.

$$\tilde{\mathbf{h}} = \frac{\mathbf{m} \odot \mathbf{h}}{1-p}, \quad m_i \sim \text{Bernoulli}(1-p) \tag{2.32}$$

Here, p is the dropout probability, $\odot$ denotes elementwise multiplication, and the factor $(1-p)^{-1}$ preserves the expected activation scale during training.

The motivation behind dropout is to prevent the network from relying too strongly on any individual activation or small group of activations (Goodfellow et al., 2016, pg. 251). Since different units are randomly removed at each training step, the remaining network must learn representations that are more robust to missing intermediate features. This can reduce overfitting, especially in dense layers or other high-capacity parts of a model. During evaluation, dropout is disabled so that the full learned representation is used.

Note that while $p = 0.5$ is a common configuration in linear neural networks (Hinton et al., 2012), such dropout can actually harm CNNs (Wu and Gu, 2015). Additionally, increasing dropout would likely lead to longer training times (Goodfellow et al., 2016, pg. 281). Effectively, while dropout can be powerful, it is not always necessary, and dropout strength can be reduced from the standard while still avoiding overfitting.

2.2.3.3.2 Normalization. Normalization layers rescale intermediate activations to improve training stability (Goodfellow et al., 2016, pg. 226). As activations move through a deep network, their distributions can shift due to changes in earlier learned parameters. This can make optimization more difficult because later layers must continually adapt to changing input statistics. Normalization addresses this by standardizing activations using estimated mean and variance statistics, often followed by learned scale and shift parameters. Thus, a given activation x is normalized as follows.

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}, \quad y_i = \gamma \hat{x}_i + \beta \tag{2.33}$$

Here, μ and σ^2 are the normalization statistics, ϵ is a small constant for numerical stability (avoiding divide-by-zero errors), and γ and β are learned affine parameters.

Different normalization methods differ primarily in how the statistics μ and σ^2 are computed. *Batch normalization* computes statistics across a mini-batch and is commonly used

in convolutional networks (Ioffe and Szegedy, 2015a). Batch normalization is particularly notable because it is commonly used to improve the rate of convergence, though the reasons are not fully clear (Russell and Norvig, 2020, pg. 768). *Layer normalization* computes statistics across features within each sample and is often useful when batch-level statistics are unstable or inappropriate (Sang et al., 2025). Other variants, such as *instance normalization* and *group normalization*, use different subsets of channels or spatial dimensions.

2.2.3.3.3 Pooling and Downsampling. Pooling layers reduce spatial resolution by summarizing local neighborhoods (Russell and Norvig, 2020, pg. 762). In convolutional architectures, this is often used to make feature maps smaller while preserving indicators of important local patterns. For a local region $\mathcal{R}_{i,j}$ around spatial location (i, j), max pooling (Zhou and Chellappa, 1988) is defined below.

$$Y_{i,j}^{\max} = \max_{(u,v)\in\mathcal{R}_{i,j}} X_{u,v} \tag{2.34}$$

Average pooling (Lin et al., 2014) is defined below.

$$Y_{i,j}^{\text{avg}} = \frac{1}{|\mathcal{R}_{i,j}|} \sum_{(u,v)\in\mathcal{R}_{i,j}} X_{u,v} \tag{2.35}$$

Max pooling retains the strongest activation in each region, while average pooling preserves the mean response.

Pooling provides several practical benefits. It reduces computational cost by shrinking feature maps, increases the effective receptive field of later layers, and can make representations less sensitive to small translations in the input (Bishop and Bishop, 2024, pg. 297).

2.2.3.3.4 Residuals. Residual connections are architectural components that pass information across one or more layers–similar to skip connections–by adding, concatenating, or otherwise combining an earlier representation with a later representation (He et al., 2016).

In the additive case, as in the original implementation (He et al., 2016), a residual block can be written as follows.

$$\mathbf{h}_{\ell+1} = \mathbf{h}_{\ell} + F_{\ell}\left(\mathbf{h}_{\ell}\right) \tag{2.36}$$

Here, $F_{\ell}(\cdot)$ denotes the learned transformation inside the residual block. Skip connections typically use identity mappings. When the input and output dimensions differ, a projection can give the residual path the appropriate shape, as shown below.

$$\mathbf{h}_{\ell+1} = P_{\ell}\left(\mathbf{h}_{\ell}\right) + F_{\ell}\left(\mathbf{h}_{\ell}\right) \tag{2.37}$$

Here, $P_{\ell}(\cdot)$ is typically a learned linear or convolutional projection.

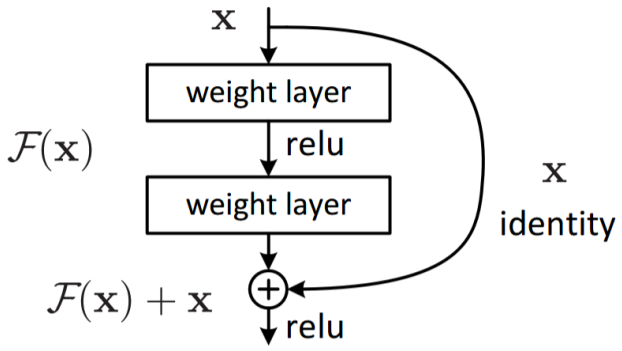


Figure 2.8: Residual skip connection example. Example block with residual connection skipping two layers (He et al., 2016, Figure 2).

Residuals are useful because they provide a shorter path for information and gradients to "move" through a network. As opposed to each block learning a complete transformation from scratch, the block can learn a residual correction relative to its input. This can make deeper feed-forward architectures easier to optimize and can help reduce degradation that may occur when additional layers are added (Duta et al., 2021; Nagpal et al., 2022).

2.3 Summary

This chapter introduces a technical background to deep learning to inform later approaches, experiments, and results.

It broadly covers AI/ML as a methodology, then introduces artificial neural networks, beginning with the perceptron formulation and extending to activation functions, loss functions, backpropagation, gradient descent, stochastic and mini-batch optimization, and adaptive optimizers such as Adam.

Next, it covered common neural-network layers and architectural components. Linear layers are presented as basic affine transformations, while convolutional layers are introduced as parameter-sharing operations well suited for spatially structured data such as images. Encoder-decoder architectures are then discussed as a framework for compressing input signals into latent representations and reconstructing or transforming them through a decoder. This includes autoencoders, transposed convolutions, VAEs, and UNets. Finally, auxiliary components such as dropout, normalization, pooling, and residual connections are described as practical mechanisms for regularization, stable optimization, dimensionality control, and improved gradient flow.

Chapter 3

Mixture-of-Experts (MoE)

3.1 Introduction

MoE is a broad methodology in which a metacognitive router model defines subproblem boundaries, and a set of cognitive expert models specialize in those defined regions. MoE is effectively a divide-and-conquer methodology, applying that basic computer science concept to neural networks. Instead of training a single cohesive model to generalize the entire solution space, a metacognitive model can identify efficient boundaries between subproblem regions (dividing) and route those tasks to cognitive submodels that specialize in those subproblem regions (conquering).

The concept is that by allowing for specialization, it is more efficient to train models. This then lends itself to improved system scalability and, potentially, improved inference speed. Furthermore, since the metacognitive capabilities learn subproblem characteristics, there is potential benefit to adapting to adjacent problem spaces with increased training efficiency over a random set of initial weights, because the metacognitive model can correlate that new problem space to an existing specialized cognitive submodel. Notably, in the context of MoE, the term for the metacognitive model is typically described as a "router". However, some literature uses the term *gating* model due to its origin in and continuing similarity to decision trees. For consistency, this work will continue using "router" as the term of choice here.

3.2 Brief History

In 1991, "Adaptive Mixtures of Local Experts" (Jacobs et al., 1991) was published in Neural Computation, presenting a supervised learning methodology built from multiple networks. Each network specialized in part of a larger problem. The framework used a divide-and-conquer optimization strategy, where simple gating networks routed inputs to dedicated expert networks that specialized in subregions of the problem space.

Notably, Jacobs et al. attributed the basis of their work to an earlier 1989 paper (Hampshire and Waibel, 1992). However, Jacobs et al. argued that Hampshire et al. used an error function that did not encourage localization, since it linearly combined outputs.

$$\text{loss}_{\text{linear}} = \|y - \sum_i p_i o_i\| \tag{3.1}$$

Here, y is the target, while p_i and o_i are the proportional contribution and output of expert i, respectively. Thus, the error function is the absolute difference between the linearly weighted prediction and the target.

In contrast, Jacobs et al. used a "stochastic" negative log-probability function, which has remained a popular choice in later works.

$$\text{loss}_{\text{stochastic}} = \sum_i p_i \|d - o_i\|^2 \tag{3.2}$$

For $p_j = \exp(x_j)/\sum_i \exp(x_i)$, x_j is the total weighted input received by output unit j of the gating network. The authors note that this approach decouples the experts from directly influencing each other. However, indirect weight coupling is through the routing function–still referred to as a gating function in their work–which defines subproblem-space boundaries. Notably, this approach is still a dense mixture; it combines the outputs of all experts.

"Hierarchical mixtures of experts and the EM algorithm" (Jordan and Jacobs, 1993) was

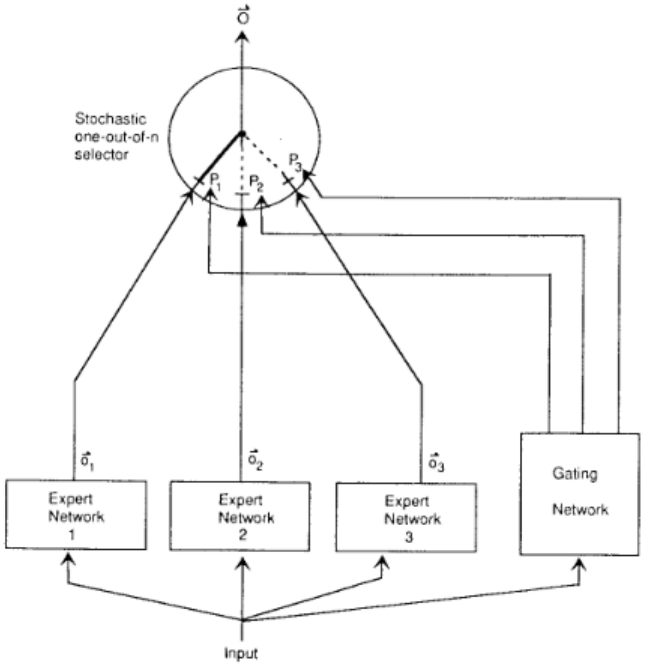


Figure 3.1: Classical MoE architecture diagram. System of expert and gating networks from (Jacobs et al., 1991, Figure 1).

published in the proceedings of the International Joint Conference on Neural Networks as an extension of the 1991 paper. This framework was referred to as a *hierarchical Mixture-of-Experts* (HME) and was based on decision trees (Breiman et al., 1984).

In 2012, "Twenty Years of Mixture-of-Experts" (Yuksel et al., 2012) summarized the prior two decades of related work, which had seen predominantly minor, incremental contributions. The survey covered many variations of the original HME framework, including common single-level Mixture-of-Experts frameworks. These single-level frameworks are occasionally referred to as *ME*, lacking a "hierarchical" component. However, the term MoE has become more common, encompassing both HMEs and MEs.

As noted earlier, Jacobs et al. used dense routing. Roughly 17 years later, the concept of sparsely routing inputs was introduced (Bo et al., 2008). This development addressed a central question for MoE: how can the added parameters and routing overhead of multiple expert networks be justified over simply increasing the size of a single network? Sparse routing helps answer this question by reducing the compute and memory required for each

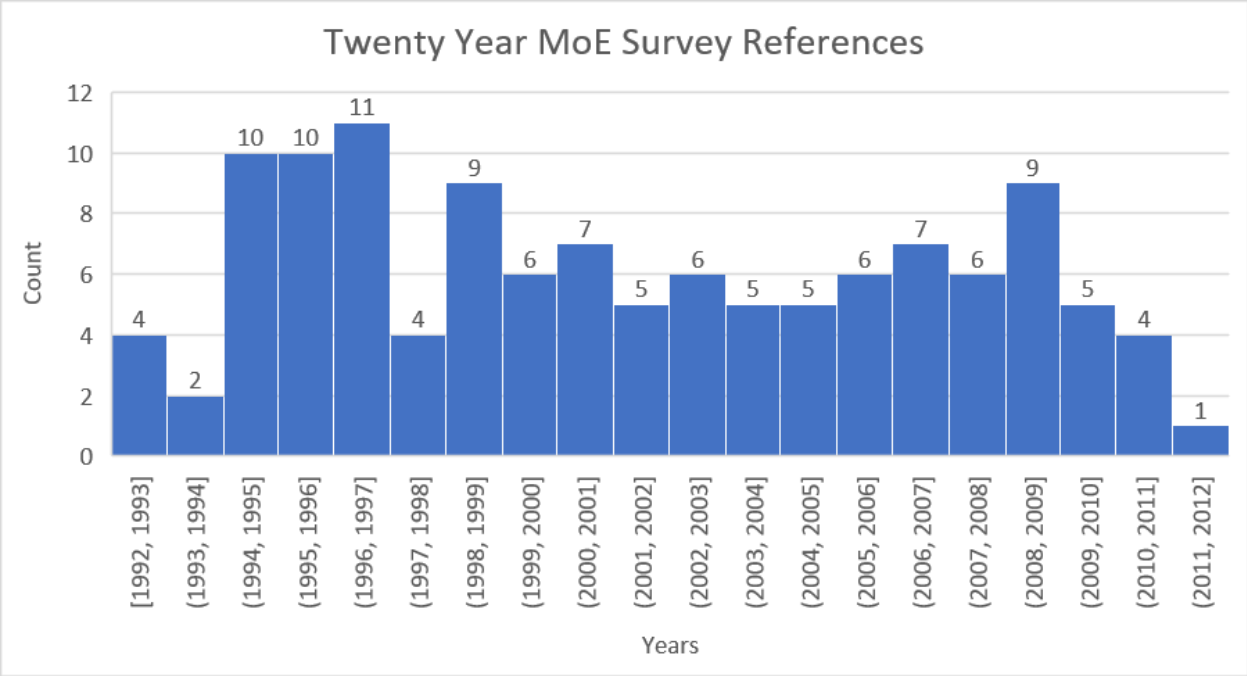


Figure 3.2: Histogram of MoE research publications (1992-2012). Publish dates provided by references in the "Twenty Years of Mixture-of-Experts" paper. Note that this does not include all MoE-related papers in the relevant period, only those identified by the survey.

inference, which later became one of the methodology's defining characteristics.

In 2017, "Attention is All You Need" (Vaswani et al., 2017), published at the 31st Conference on Neural Information Processing Systems, introduced the transformer architecture. Before transformers, sequential and token processing often depended on computationally inefficient recurrent neural networks (RNNs) and long short-term memory networks (LSTMs) (Hochreiter and Schmidhuber, 1997). Transformers made massive sequence models more viable, leading to later innovations such as large-language models (LLMs) (Brown et al., 2020) and stable diffusion models (Peebles and Xie, 2023).

Most recently, in 2025, "A Survey on Mixture-of-Experts in Large Language Models" (Cai et al., 2025) was published in the 37th volume of the Transactions on Knowledge and Data Engineering journal, which explored the prior 8 years of transformer-based LLMs. Here, there is substantial emphasis on *routing algorithms*, which is "a key feature to all sparse expert architectures, determin[ing] where to send [inputs]." This explores many other fine-

tuning and hyperparameter optimizations, because at the magnitude LLMs are operating, even marginal improvements can have substantial impacts.

3.3 Motivation

MoE uses a divide-and-conquer strategy in which a routing component divides the problem space and expert submodels specialize in the resulting subregions (Yuksel et al., 2012). This makes MoE relevant to several goals of this work. Although related approaches such as ensembles (Rincy and Gupta, 2020), bagging (Ali et al., 2022), and meta-learning (Vettoruzzo et al., 2024; Hartmann et al., 2019) also involve multiple models, collaborative submodels, or learned problem decompositions, MoE is distinguished by its explicit routing mechanism and sparse expert activation. In many cases, this appears as top-1 routing, though some MoE systems use top-2 or more general top-k inference (Huang et al., 2025). This routing structure motivates the central questions of this work: whether routers can improve interpretability, whether router pre-training can improve optimization, and whether routing can support dynamic inference costs in smaller systems.

3.3.0.0.1 H1–H2: Interpretability and Explainability The routing component of MoE is particularly relevant to interpretability because it learns high-level structure in the problem space and uses that structure to determine how samples should be processed. This raises the question of how routing behavior can be analyzed or modified to make the learned system more interpretable. This motivates the first two hypotheses, introduced in Chapter 1:

1. H1: The latent space learned by a router encodes meaningful subproblem structure that can be analyzed to improve interpretability through relational and geometric methods.

2. H2: The router can integrate attention-based mechanisms to indicate important characteristics of input data for human interpretability and potentially improve performance by sharing more information with each expert.

Interpretability, often discussed alongside *explainability*, is increasingly important as deep learning systems are deployed in more consequential settings (Garouani et al., 2024). Neural networks are commonly described as "black-box" models because their decision processes are difficult to inspect directly. This limits trust in safety- and security-critical applications such as medical technology, energy infrastructure, and autonomous vehicles, where systems may operate for extended periods with limited human oversight or otherwise have severe consequences in the event of an unexpected failure.

3.3.0.0.2 H3: Optimizing Training Processes. Training-process optimization is a central challenge in deep learning. Broadly, it asks how models can be trained more quickly, more efficiently, or more effectively. For MoE systems, this challenge is especially important because end-to-end training creates a feedback loop between the router and experts: changing the router changes which samples each expert receives, which changes how the experts learn, which can then change how the router should allocate future samples. Pre-training the router may reduce this instability by providing a fixed or partially stabilized latent reference frame for expert specialization. This motivates the third hypothesis:

H3: Pre-training a router, instead of end-to-end, can reduce overall training cost and enable efficient transfer learning by providing a stable latent reference frame for expert specialization.

3.3.0.0.3 H4: Dynamic Inference Costs. A final motivation comes from the observation that some problem spaces contain samples of varying difficulty. For example, in MNIST, some digits are substantially easier to classify than others. This raises a natural question: if some samples are less complex, can they be classified using fewer active parameters than samples requiring more complex models? Dynamic inference-cost paths may provide a way to scale computation according to sample difficulty, improving efficiency while preserving performance. This motivates the fourth hypothesis:

H4: Effective dynamic inference can be derived by pre-training a neural network to learn a

fixed routing strategy, then replacing under-performing experts with higher capacity experts to improve system performance while using minimal resources to handle simpler problems.

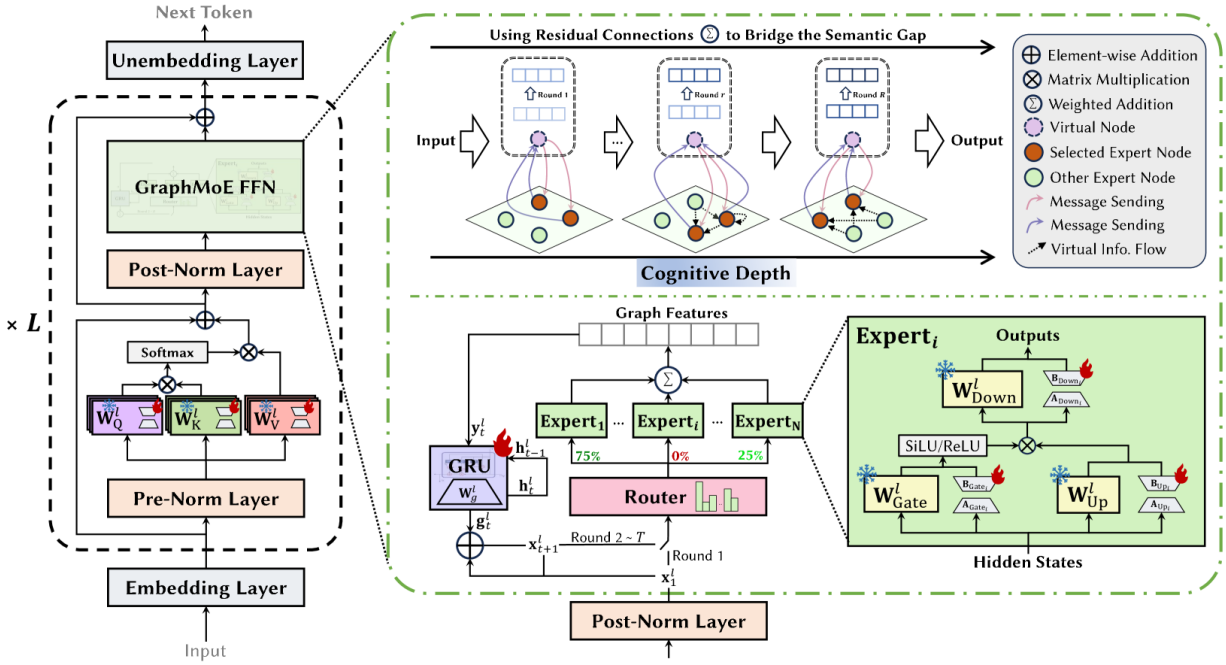


Figure 3.3: Graph MoE architecture illustration (Lv et al., 2026, Figure 1).

Some prior work has explored related ideas. The *GraphMoE* approach, shown in Figure 3.3, defines a graph of interconnected experts to augment cognitive depth (Lv et al., 2026). However, GraphMoE is applied to LLMs and uses a recurrent routing strategy. While effective in that setting, the focus of this work is smaller-scale applications in which dynamic inference cost is realized through predefined variable path sizes.

Similarly, the *Chain-of-Experts* approach, shown in Figure 3.4, achieves effective depth through sequential signal processing across multiple routing steps (Wang et al., 2025c). Tokens are passed through multiple experts over several *communication steps*, with each step further processing the signal. Like GraphMoE, this method is primarily designed for LLMs. It depends on token-specific logic, making it less directly applicable to smaller computer-vision problems such as object detection or recognition.

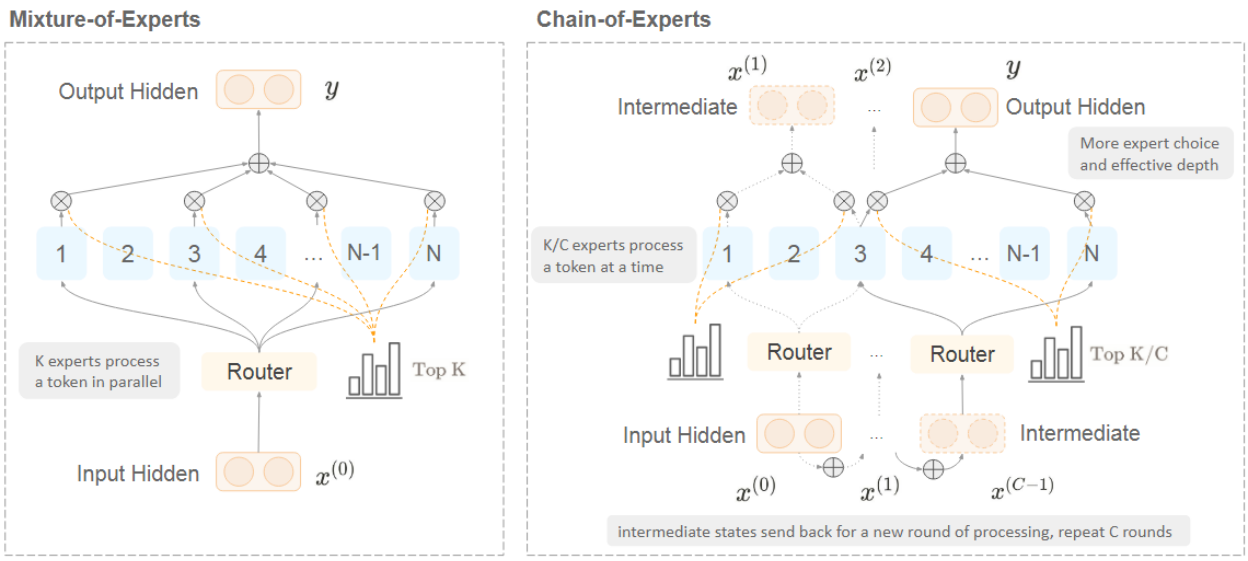


Figure 3.4: Chain-of-Experts architecture illustration (Wang et al., 2025c, Figure 2).

3.3.0.0.4 Summary. The motivating factors behind this MoE research are interpretability and explainability, training-process optimization, and dynamic inference costs. These motivations are grounded in long-standing challenges in deep learning: improving trust and safety, reducing training complexity, and increasing inference efficiency. Each motivation centers on the routing component, with the learned latent space serving as a key object of analysis for the hypotheses evaluated in this work.

3.4 Baseline Architecture

This section defines the baseline MoE architecture used here.

The baseline architecture is illustrated here by Figure 3.5. The router Φ defines the routing probabilities P. Then, given several experts to use k, P defines top-k routing probabilities P^k and experts selection Θ^k (shown in a single step). Note that P^k is re-normalized top-k routing probabilities to preserve the magnitude of the output signal. Each selected expert contributes to the top-k output signal $\hat{y}^k$, which is then aggregated through a weighted sum over the top-k routing probabilities to form the output signal $\hat{y}$. A loss function $\mathcal{L}$ evaluates $\hat{y}$ with respect to some target y and forms a loss gradient to update the router and

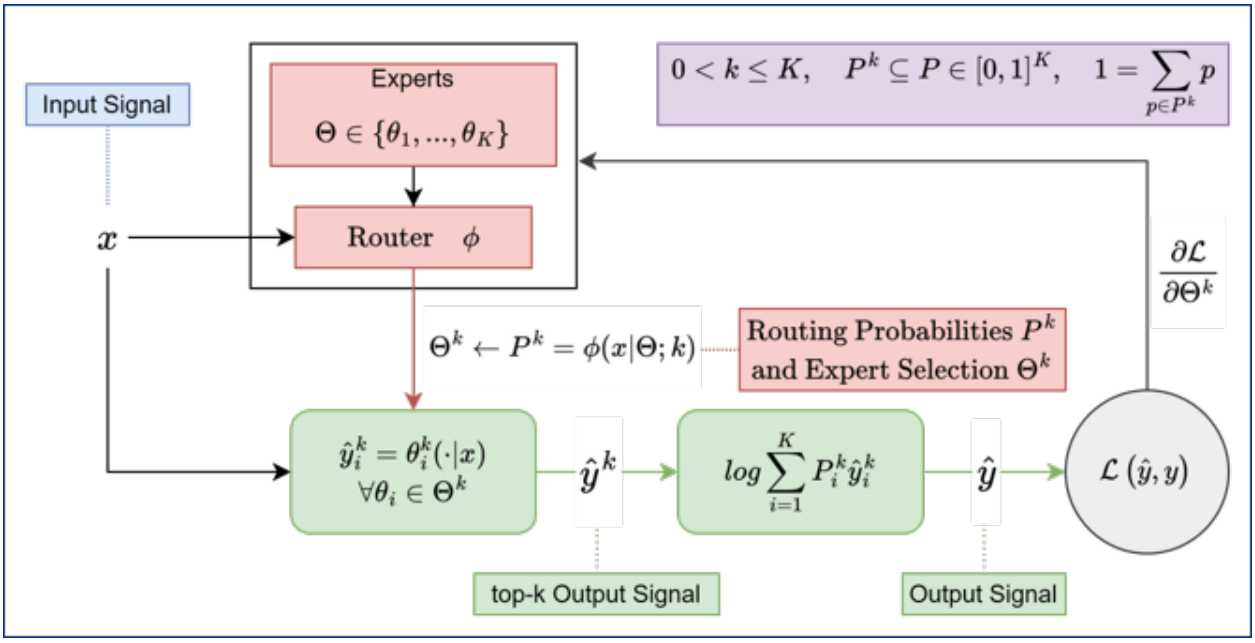


Figure 3.5: Baseline MoE architecture. It illustrates how input signal x is used to define the experts' mixture, how an output is derived from that mixture, and the update loop formed to optimize the MoE network.

selected experts.

The baseline architecture, illustrated in Figure 3.5, is based on the original design detailed in Figure 3.1. The figure additionally provides some detail on the operations performed and how an error gradient forms an update loop. However, there is one substantial variation: a top-k expert selection. When $k = K$, the MoE baseline matches the classical dense experts mixture, where every expert contributes to the output signal (weighted by the routing probabilities). However, for $k < K$ a sparse experts mixture is used, where only a sparse subset of experts are selected and updated. As previously noted, this top-k routing probabilities P^k is re-normalized to preserve the magnitude of the output signal.

3.5 Experts Mixture

This subsection details the abstracted routing methodology, how a top-k selection is formed, how an output is derived from the experts mixture, and how error gradients update the MoE

architecture.

Routing methodology is the focus of many research works. This considers a subset of those: linear, Cosine, Centroid, and RBF, which are detailed in later subsections.

In the baseline architecture used throughout this work, the router Φ maps an input signal $\mathbf{x}$ into a latent representation $\mathbf{z} \in \mathbb{L}^N$ and then converts that representation into K routing logits, one for each expert θ_i in the set of all experts Θ.

3.5.1 Assumptions

To begin with, consider the following constraints.

$$1 \leq k \leq K, \quad K > 1 \tag{3.3}$$

When referring to a top-k selection, it means selecting k experts. Thus, it stands to reason that $K = 1$ should be valid, where the routing is then trivial. In the general case, that statement is true. However, it adds some numerical instabilities later on that require conditional support. Furthermore, such a trivial case could then be substituted by a classical single-model approach, without routing overhead. Therefore, only cases of $K > 1$ are considered in this work.

3.5.2 Encoding

The encoding model ϕ encodes an input signal $\mathbf{x}$ into a representative latent vector $\mathbf{z}$ that encompasses salient features in a low-dimensional space.

$$\mathbf{z} = \phi(\mathbf{x}), \qquad \mathbf{z} \in \mathbb{L}^N \tag{3.4}$$

Here, N denotes the dimensionality of the latent space $\mathbb{L}$.

3.5.3 Routing Selection

The routing probabilities P define a top-k probability mixture P^k. These probabilities are defined as a function of the routing process, and both define the subset of experts that forms the top-k selection.

$$P^k = \Phi(\mathbf{x}|\Theta; k) \tag{3.5}$$

Here, Φ represents the full routing process and encapsulates a specific routing function that correlates experts with the latent encoding $\mathbf{z}$.

Notably, then, ϕ refers to just the sub-process within Φ that actually encodes the information. The encoding model ϕ, in conjunction with a routing methodology such as the four identified at the beginning of this section, then forms the full router Φ.

One detail that is omitted in the above derivation of P^k is that the routing probabilities are themselves defined as the softmax probabilities of the routing logits $r \in \mathbb{R}^K$.

$$P^k = \text{select}(\text{softmax}(r + \tau \cdot \epsilon); k), \quad \epsilon \sim \mathcal{N}(0, \sigma_r) \quad \tau \in [0, 1] \tag{3.6}$$

Here, the routing logits may be perturbed by Gaussian exploration noise controlled by σ_r to promote exploration and avoid early routing collapse (Yuksel et al., 2012). During inference and evaluation, such routing noise is excluded by setting $\sigma_r = 0$. Additionally, the select function selects the top-k experts based on the resulting probabilities and re-normalizes them to preserve magnitude.

$$1 = \sum_{p \in P^k} p \tag{3.7}$$

The routing probabilities then inform the set of selected experts Θ^k.

$$\Theta^k \leftarrow P^k \tag{3.8}$$

$$\Theta^k \subseteq \{\theta_1, ..., \theta_K\} \tag{3.9}$$

$$\tag{3.10}$$

3.5.4 Output Approach

Next, a prediction is informed by the top-k selection. Each selected expert i produces a component output signal $\hat{y}_i^k$.

$$\hat{y}_i^k = \theta_i^k(x), \quad \theta_i^k \in \Theta^k \tag{3.11}$$

The component outputs are then aggregated via a weighted summation of each expert's prediction and its routing probability to form a reduced output signal $\hat{y}$. Note that this applies only to inference, whereas training requires a log-mix. This is further explained in the subsequent subsection.

$$\hat{y} = \sum_{i=1}^{k} P_i^k \cdot \hat{y}_i^k \tag{3.12}$$

Note that in the top-1 case, the weighted aggregation can be effectively skipped, but the above formulation is still mathematically stable if it is not. In such a case, the summation term sums the single expert's result scaled by the trivial normalized routing probability $P^k = 1$.

$$\hat{y} = \sum_{i=1}^{k} P_i^k \cdot \hat{y}_i^k = P_1^k \cdot \hat{y}_1^k = \hat{y}_1^k, \quad k = 1 \tag{3.13}$$

Thus, while an ensemble of experts can contribute to a result, a single expert can be the sole model handling it. This is useful for minimizing inference path costs, since the number of parameters scales linearly with k as k expert models need to process each signal, with additional overhead for aggregating their individual results.

3.5.5 Error Gradients

There are two primary methods discussed here for computing error gradients in MoE:

1. Marginal negative log-likelihood over the experts' mixture.

2. Top-1 error gradients using classical, single-model optimization.

While there are other optimization methods (Yuksel et al., 2012; Jiang et al., 2026), the marginal log-likelihood is the classical approach (Jordan and Jacobs, 1993) and was chosen as the baseline; the other is given to explore the obvious idea of only training the best expert determined by the router.

If, in the inference path-cost setting, it is ideal to use only a single expert, then it may seem reasonable to train the model to maximize that exact policy. That is, use a top-1 approach in the training process, where the MoE output can then be treated identically to a single-model approach. In the problem space of image classification, common examples include cross-entropy.

However, it then becomes difficult for the router to distinguish between experts since, in such backpropagation of error gradients, the magnitude of confidence the router had in the selected expert is washed out as a constant scalar. This is due to the obvious issue that a single routing probability is used in the full, differentiable inference path, which thus does not provide context for the routing choice. As a result, the router is prone to effectively randomly sampling experts for each input, where all experts then converge to be generalist models instead of specialists.

The marginal negative log-likelihood mixes the routing probabilities with the expert outputs to produce a set of weighted logits. The classical form is given below.

$$\text{logits} = \log\left(p(k|x) * p_k(y|x)\right) \tag{3.14}$$

Here, $p(k|x)$ denotes the probability that an expert is selected given the input, and $p_k(y|x)$

denotes the probability that the expert selects an image class given the input.

For supervised classification, let $y \in 1, \ldots, O$ denote the target class associated with input signal $\mathbf{x}$. Each selected expert $\theta_i^k \in \Theta^k$ produces logits $\hat{\mathbf{y}}i^k$, which define the expert-conditional predictive distribution.

$$p_i^k(y \mid \mathbf{x}) = \text{softmax}(\hat{\mathbf{y}}i^k) * y \tag{3.15}$$

The top-k mixture likelihood marginalizes over the selected experts using the normalized routing probabilities P^k.

$$p(y \mid \mathbf{x}, \Theta^k, P^k) = \sum_{i=1}^{k} P_i^k \cdot p_i^k(y \mid \mathbf{x}) \tag{3.16}$$

The corresponding marginal negative log likelihood is then defined below.

$$\mathcal{L}\text{mNLL}(\mathbf{x}, y) = -\log\left(\sum_{i=1}^{k} P_i^k \cdot \text{softmax}(\hat{\mathbf{y}}i^k)y\right) \tag{3.17}$$

Equivalently, this quantity can be expressed in log-space.

$$\mathcal{L}\text{mNLL}(\mathbf{x}, y) = -\log \sum_{i=1}^{k} \exp\left(\log P_i^k + \log p_i^k(y \mid \mathbf{x})\right) \tag{3.18}$$

3.5.6 Sparse Routing over Batches

One technical implementation challenge is integrating sparse routing with batches of samples. As discussed in Section 2.2.2.2, the optimization process typically involves averaging backpropagation steps over a batch of samples. However, in sparse-routing, each sample could require a unique top-k selection of experts when $k < K$. As with many such technical challenges, there are likely multiple solutions. The approach used here, however, is to build a pseudo-dense set of output logits where the sparse selections update their respective logits

and the other logits are zeroed out. More details on specific implementations are discussed later, but this immediately details an essential but more complex interaction.

As discussed in Section 2.2.2.2, the optimization process typically involves averaging backpropagation steps over a batch of samples. However, in sparse routing, each sample may require a unique top-k selection of experts when $k < K$. Let a batch be denoted by $\mathcal{B} = \{\mathbf{x}_b\}_{b=1}^{B}$, and let Θ_b^k be the selected top-k expert set for sample $\mathbf{x}_b$. Then a binary selection mask is defined.

$$M_{j,b} = \mathbb{1}\{\theta_j \in \Theta_b^k\}, \qquad M \in \{0,1\}^{K \times B} \tag{3.19}$$

Thus, for each expert θ_j, only the subset of batch elements routed to that expert must be evaluated.

$$\mathcal{B}_j = \{\mathbf{x}_b \in \mathcal{B} : M_{j,b} = 1\} \tag{3.20}$$

The approach used here is to build a pseudo-dense tensor of expert logits $\tilde{\mathbf{Y}} \in \mathbb{R}^{K \times B \times O}$, where selected expert outputs update their corresponding entries and unselected entries are zeroed out.

$$\tilde{\mathbf{Y}}_{j,b} = \begin{cases} \theta_j(\mathbf{x}_b), & M_{j,b} = 1, \\ \mathbf{0}, & M_{j,b} = 0 \end{cases} \tag{3.21}$$

The corresponding sparse routing probabilities can then be represented over all K experts as zero-masked top-k probabilities.

$$\tilde{P}_{j,b}^k = M_{j,b} P_{j,b}^k, \qquad \sum_{j=1}^{K} \tilde{P}_{j,b}^k = 1 \tag{3.22}$$

This allows the batched mixture prediction to be written in a dense-looking form while still only evaluating the selected experts.

$$\hat{\mathbf{y}}_b = \sum_{j=1}^{K} \tilde{P}_{j,b}^k \tilde{\mathbf{Y}}_{j,b} \tag{3.23}$$

Equivalently, for the marginal likelihood objective, the unselected experts are excluded by assigning them zero routing mass.

$$\log p(y_b \mid \mathbf{x}_b) = \log \sum_{j=1}^{K} \exp\Big(\log \tilde{P}^k_{j,b} + \log \operatorname{softmax}(\tilde{\mathbf{Y}}_{j,b})_{y_b}\Big), \qquad \log 0 = -\infty \tag{3.24}$$

Thus, the implementation preserves a convenient dense tensor structure for batching while retaining sparse computational behavior over the selected expert paths.

3.6 Routing Methods

The router is the metacognitive component of the MoE system. It determines how an input signal is assigned across the available experts and therefore defines learned boundaries between subproblem regions (Jacobs et al., 1991; Yuksel et al., 2012; Shazeer et al., 2017). The router first encodes each input signal $\mathbf{x}_b$ into a latent representation.

$$\mathbf{z}_b = \phi(\mathbf{x}_b), \qquad \mathbf{z}_b \in \mathbb{L}^N \tag{3.25}$$

The router then computes one logit for each of the K experts. For a batch of size B, the routing logits can be written as follows.

$$R \in \mathbb{R}^{K \times B}, \qquad R_{i,b} = r_i(\mathbf{z}_b) \tag{3.26}$$

The routing probabilities are obtained by applying a softmax over the expert dimension.

$$P_{i,b} = \frac{\exp(R_{i,b})}{\sum_{j=1}^{K} \exp(R_{j,b})} \tag{3.27}$$

During training, Gaussian routing noise may be added to the logits before the softmax. During evaluation, this perturbation is omitted.

$$P_{i,b} = \text{softmax}_i(R_{\cdot,b} + \eta_{\cdot,b}), \qquad \eta_{i,b} \sim \mathcal{N}(0, \sigma_r^2) \tag{3.28}$$

For sparse routing, the router selects the top-k probabilities independently for each sample. Let S_b^k denote the selected expert indices for sample $\mathbf{x}_b$. The sparse, re-normalized routing probability is then defined below.

$$P_{i,b}^k = \frac{P_{i,b}\mathbb{1}\{i \in S_b^k\}}{\sum_{j \in S_b^k} P_{j,b}} \tag{3.29}$$

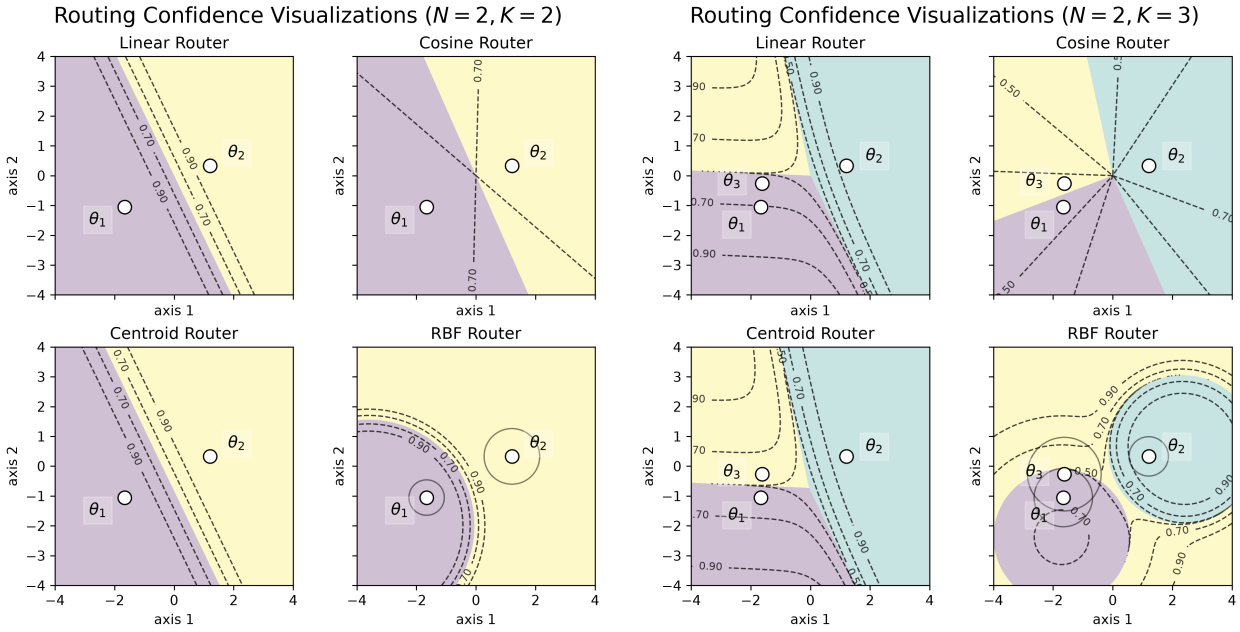
Thus, $P_{i,b}^k = 0$ for unselected experts and $\sum_{i=1}^K P_{i,b}^k = 1$. When $k = K$, no sparsification is required and $P^k = P$.

Figure 3.6 visualizes the routing algorithms explored in this section: Linear, Centroid, Cosine, and RBF. Note the similarity between Centroid and Linear, where effectively they only differ by a guaranteed bias term. Further note how the RBF allows for the most flexible usage of the space. Though looking ahead, for the baselines, all algorithms perform roughly evenly, the potential flexibility of the RBF makes it the default choice.

Figure 3.6 does detail some peculiarities with the contours (routing confidence) in the Cosine's routing space that may be observed. These are due to plotting software struggling to map the extremely narrow peaks defined in the space, making it prone to smoothing algorithms omitting those contours. This is mostly an issue for $K = 4$ and $K = 5$, when the space becomes significantly complex, however.

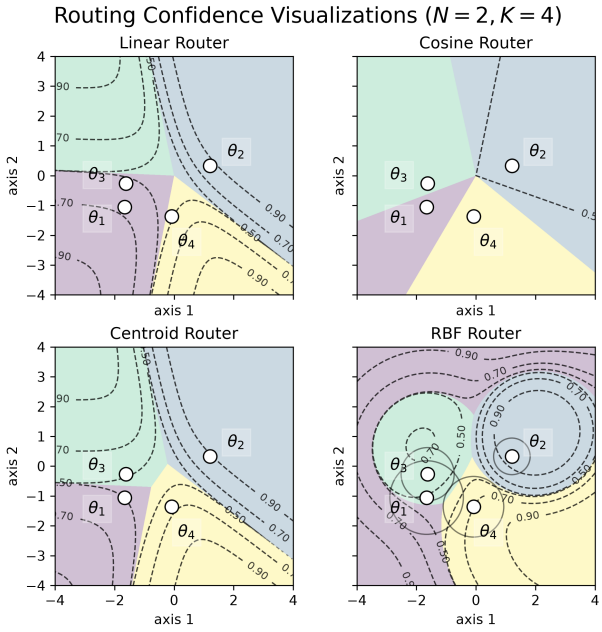
3.6.0.1 Linear Router

The linear router is the simplest routing strategy evaluated here. It computes expert preference using a linear score in the latent space. The router maintains a learned embedding

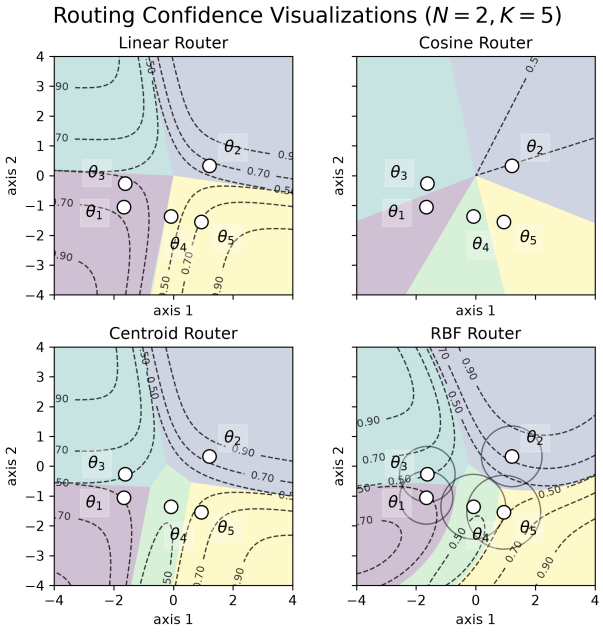


(a) $K = 2$

(b) $K = 3$



(c) $K = 4$



(d) $K = 5$

Figure 3.6: Visualization of the utilized routing algorithms across the common K values used here. Each circle represents the latent embedding of an expert, annotated by θ_i that marks it as submodel (expert) i. Contours detail routing probabilities: the strength at which an encoded sample on that line would be routed to the nearest expert submodel.

vector $\mathbf{e}_i \in \mathbb{L}^N$ for each expert, and the logit for expert i is defined below.

$$r_i(\mathbf{z}) = \mathbf{z}^\top \mathbf{e}_i \tag{3.30}$$

The linear router therefore determines expert preference through unnormalized latent-space scores followed by softmax and top-k selection. Pairwise comparisons between linear scores define its decision regions.

3.6.0.2 Cosine Router

The cosine router is a normalized variant of the linear router that evaluates routing preference using direction in latent space (Nguyen et al., 2025). The normalized latent representation and normalized expert embedding are defined below.

$$\bar{\mathbf{z}} = \frac{\mathbf{z}}{\max(\|\mathbf{z}\|_2, \epsilon)}, \qquad \bar{\mathbf{e}}_i = \frac{\mathbf{e}_i}{\max(\|\mathbf{e}_i\|_2, \epsilon)} \tag{3.31}$$

The cosine router then computes logits by cosine similarity.

$$r_i(\mathbf{z}) = \bar{\mathbf{z}}^\top \bar{\mathbf{e}}_i \tag{3.32}$$

This tests whether expert specialization is better explained by latent direction rather than raw activation magnitude. The small constant ϵ prevents numerical instability when normalizing near-zero vectors.

3.6.0.3 Centroid Router

The centroid router represents each expert by a learned latent centroid $\mathbf{c}_i \in \mathbb{L}^N$. It determines routing by the negative squared Euclidean distance from an encoded sample to each Centroid.

$$r_i(\mathbf{z}) = -\frac{\|\mathbf{z} - \mathbf{c}_i\|_2^2}{N} \tag{3.33}$$

The negative sign makes closer centroids produce larger logits, while normalization by N keeps the distance scale comparable across latent dimensions. Functionally, this makes the centroid router resemble a soft, learnable nearest-centroid classifier in the router latent space. The resulting top-1 boundaries form exclusive cluster regions, while top-k routing allows neighboring centroids to share samples.

Note that the Centroid algorithm can be decomposed into a Linear representation.

$$-\frac{1}{N}\left\|\mathbf{z}-\mathbf{c}_i\right\|^2 = -\frac{1}{N}\left(\mathbf{z}^\top\mathbf{z} - 2\mathbf{z}^\top\mathbf{c}_i + \mathbf{c}_i^\top\mathbf{c}_i\right) = \frac{2}{N}\mathbf{z}^\top\mathbf{c}_i - \frac{1}{N}\left\|\mathbf{c}_i\right\|^2 - \frac{1}{N}\left\|\mathbf{z}\right\|^2 \tag{3.34}$$

$$\frac{2}{N}\mathbf{z}^\top\mathbf{c}_i - \frac{1}{N}\left\|\mathbf{c}_i\right\|^2 = \mathbf{w}_i^\top\mathbf{z} + b_i, \qquad \mathbf{w}_i = \frac{2}{N}\mathbf{c}_i, \qquad b_i = -\frac{1}{N}\left\|\mathbf{c}_i\right\|^2 \tag{3.35}$$

However, also note that, unlike the Linear algorithm, the Centroid provides a bias term for each expert.

3.6.0.4 RBF Router

The RBF router extends the centroid router by adding a learned radial scale for each expert. Each expert therefore has both a centroid $\mathbf{c}_i$ and a learned log-width parameter ρ_i. The corresponding radial width is defined below.

$$\sigma_i = \exp(\rho_i) \tag{3.36}$$

The resulting RBF routing logit is defined below.

$$r_i(\mathbf{z}) = -\frac{1}{2\sigma_i^2}\|\mathbf{z}-\mathbf{c}_i\|_2^2 - N\rho_i \tag{3.37}$$

Here, σ_i controls the width of the region associated with expert i. This gives the router a Gaussian radial-basis interpretation, where routing preference is still based on distance to an expert center. Still, each expert can learn a different effective radius (Broomhead and Lowe, 1988).

The RBF router is useful when the assumption of equal-size expert regions is too restrictive. In the centroid router, all experts compete according to the same distance scale. In the RBF router, one expert may learn a narrower region while another learns a broader region. This allows greater routing flexibility when subproblem regions vary in size or density. The $-N\rho_i$ term acts as a dimensional log-scale penalty, preventing an expert from becoming broadly attractive solely by increasing its width.

3.7 Metrics

This section describes four metrics used to characterize routing behavior: entropy, switch-style load imbalance, soft load imbalance, and spread. These metrics are computed from the router probability matrix $\mathbf{P} \in \mathbb{L}^{K \times B}$, where K is the number of experts and B is the batch size. Each entry $p_{i,b}$ denotes the soft routing probability assigned to expert i for input sample b. For each sample b, the expert probabilities sum to one.

$$\sum_{i=1}^{K} p_{i,b} = 1. \tag{3.38}$$

3.7.0.0.1 Routing entropy. Routing entropy measures how uncertain the router is when assigning samples to experts. For a batch of samples, it is computed as the mean entropy of the per-sample routing distributions.

$$H(\mathbf{P}) = -\frac{1}{B} \sum_{b=1}^{B} \sum_{i=1}^{K} p_{i,b} \log(\max(p_{i,b}, \epsilon)) \tag{3.39}$$

Here, ϵ is a small numerical stability term that prevents evaluating the logarithm at zero. Lower entropy indicates more confident routing decisions, with probability mass concentrated on fewer experts. Higher entropy indicates less confident routing decisions, with probability

mass distributed more evenly across experts. When normalized, entropy is divided by $\log(K)$.

$$H_{\text{norm}}(\mathbf{P}) = \frac{H(\mathbf{P})}{\log(K)} \tag{3.40}$$

For $K > 1$, this bounds the entropy to $[0, 1]$, where 0 corresponds to fully concentrated routing and 1 corresponds to uniform routing.

3.7.0.0.2 Switch-style load imbalance. Switch-style load imbalance compares hard top-1 expert usage against the mean soft probability assigned to each expert. Let a_b denote the top-1 expert selected for sample b.

$$a_b = \arg\max_i p_{i,b} \tag{3.41}$$

The top-1 usage frequency for expert i is defined below.

$$f_i = \frac{1}{B} \sum_{b=1}^{B} \mathbb{1}[a_b = i] \tag{3.42}$$

Here, $\mathbb{1}[\cdot]$ is the indicator function. The mean soft routing probability assigned to expert i is defined below.

$$P_i = \frac{1}{B} \sum_{b=1}^{B} p_{i,b} \tag{3.43}$$

The switch-style load imbalance is then defined below.

$$\mathcal{L}_{\text{switch}} = K \sum_{i=1}^{K} f_i P_i \tag{3.44}$$

This metric is minimized when both hard usage frequencies and mean soft probabilities are balanced across experts. In the implemented form, f_i is treated as a detached coefficient, so the metric reflects hard top-1 routing behavior while gradients flow through P_i. When

normalized, the metric is shifted and rescaled.

$$\mathcal{L}_{\text{switch,norm}} = \frac{\mathcal{L}_{\text{switch}} - 1}{K - 1} \tag{3.45}$$

The non-normalized form has an ideal value of 1 under balanced use and increases as usage becomes more concentrated.

3.7.0.0.3 Soft load imbalance. Soft load imbalance measures the squared deviation of average soft usage from uniform usage. Using the same mean soft usage term P_i, this quantity is defined below.

$$\mathcal{L}_{\text{soft}} = \frac{1}{K} \sum_{i=1}^{K} \left(P_i - \frac{1}{K} \right)^2 \tag{3.46}$$

This metric is 0 when the average soft routing distribution is exactly uniform across experts. Larger values indicate that the router assigns more probability mass to some experts than others. Unlike switch-style load imbalance, this metric depends only on soft routing probabilities instead of top-1 expert selections.

3.7.0.0.4 Spread. Spread measures the range between the largest and smallest routing probabilities observed across the batch.

$$S(\mathbf{P}) = \max_{i,b} p_{i,b} - \min_{i,b} p_{i,b} \tag{3.47}$$

This is a simple descriptive, non-differentiable statistic useful for interpreting routing behaviors. A larger spread indicates that at least some routing decisions are highly concentrated relative to the smallest probabilities in the batch. A smaller spread indicates that routing probabilities are more evenly distributed across the set of experts. Notably, neither is necessarily better. However, routing collapse–when the router begins heavily preferring a specific subset of experts–is correlated with reduced effectiveness, and so is generally treated as less desirable than even routing (Yuksel et al., 2012). But moderate bias in the routing

preference is not necessarily problematic in the same way.

3.8 MoE Routing Overhead

Compared to baseline single-model classifiers, some obvious structural differences increase the learnable parameters and computational requirements. The MoE architecture includes extra overhead via the routing component in comparison to a typical classifier network. Additionally, while the MoE architecture can, as discussed before, form its outputs potentially as a mixture of multiple experts, such mixtures then further increase the computational requirements. Notably, a classifier network could be upscaled to approximate the parameter and computational requirements of the MoE system.

A flawed comparison would be evaluating the MoE in top-k, where $1 < k \leq K$, and comparing it to a feedforward network the same size as just one of the MoE's experts, except for some of the potential interpretability benefits MoE inherently provides, discussed later. More comparably, it should be done in the context of some consistent baseline such as the number of parameters along the inference path, which is the approach used in this work.

Notably, learnable parameters do not account for full computational efficiency. For example, the ReLU activation is a statically defined mapping function that nonetheless imposes additional computational costs on an inference call. Full computational efficiency is a difficult metric to derive, as it depends on hardware architectures/compiler-specific optimizations, device-based accelerations such as CPU versus CUDA operations, and precise implementation details in a work where such efficiencies are not the focus. The count of learnable parameters is easily computable, consistent across platforms, and largely informs the computational requirements of the model/system.

This method of evaluation further influences several architectural design decisions made to reduce bias in later evaluations and comparisons:

1. Experts are structurally identical to baseline classifier models.

2. Routing overhead is kept fairly minimal so a top-1 inference path cost can be reasonably compared to a classifier baseline.

This allows for comparisons between approximately similar parameter-count models. However, there will still be inequalities if an expert is identical to a classifier baseline and additionally has routing overhead in the top-1 inference path cost. Therefore, multiples of both at different parameter scales are trained, tested, and evaluated to extrapolate behaviors with increased confidence.

3.9 Summary

This section defined the baseline MoE architecture used in this work. The router maps each input into a latent representation, computes routing probabilities over experts, and optionally converts those probabilities into sparse top-k weights. The expert mixture then evaluates the selected experts and aggregates their predictions. The router families considered here–linear, Cosine, Centroid, and RBF–provide different geometric assumptions about how expert regions are organized in latent space. The accompanying routing metrics and cluster confusion score provide tools for evaluating confidence, load balance, Spread, and latent separation. Together, these components establish the architectural foundation for the baseline experiments and later dynamic-inference studies.

Chapter 4

PyTorch Model Factory

4.1 Overview

Chapter 2 provides a background on basic deep learning methodologies, including neural networks, CNNs, and encoder-decoder models. Chapter 3 extends the background by describing the MoE methodology, routing functions, and routing metrics. This chapter describes the implementation details and summarizes a library developed to facilitate the procedural construction of such models for experimentation. Furthermore, it presents a concise, high-level notation for a formulaic description of model architectures to understand and differentiate experimental models by their layouts quickly. For example, describing a CNN model as a 100k parameter model gives some insight, but it leaves many details blank, such as model height, layer widths, and downsampling block depths.

Therefore, while this chapter mainly focuses on technical details, it also provides explanations for later experiment descriptors that detail model architectures at a high level, which will be referenced in later chapters as well.

4.1.0.0.1 Programming language. *Python* (Python Software Foundation, 2021) provides a flexible programming environment for experimental machine learning research because it supports rapid prototyping, structured software organization, and interoperability with scientific computing libraries such as SciPy (Virtanen et al., 2020). In this work, Python is used to define neural network architectures, manage configurations, run training scripts, log metrics, aggregate results, and generate figures. Notably, Python 3.14 is used for this

research work.

4.1.0.0.2 PyTorch. *PyTorch* (Paszke et al., 2017) is the deep learning framework used here to build and optimize neural networks. It provides the `torch.nn.Module` abstraction, automatic differentiation, parameter serialization, batched dataset utilities, and standard neural network components such as convolutional layers, linear layers, normalization layers, and activation functions. The *PyTorch Model Factory* (TMF) builds on these PyTorch components while providing a more concise model-definition layer for producing architectural variations.

4.2 Framework Motivation

PyTorch is intentionally general. This flexibility is helpful, but it requires substantial additional development for more advanced, specialized tasks. This is a broadly accepted paradigm, with complementary tools designed to provide missing or specific capabilities. For example, the *PyTorch Lightning* (Falcon and The PyTorch Lightning team, 2019) provides a training framework with support for logging, metric aggregation, device management, and random-number generator (RNG) seeding. *TorchInfo* (Yep, 2020) provides model summaries and parameter-count diagnostics.

The TMF library was subsequently developed to support procedurally defining families of related neural architectures from compact, reusable specifications, facilitating easy perturbations and high-level insights. Additionally, it standardizes model interfaces through common base derivations to provide a consistent API to improve development.

The main experimental need is controllable layout and architectural perturbations. Many experiments evaluate and compare models across a range of parameters. Writing each of those models directly in PyTorch would make the experiment definitions long and error-prone through manual design. Instead, TMF makes some of the relevant architectural choices explicit, such as:

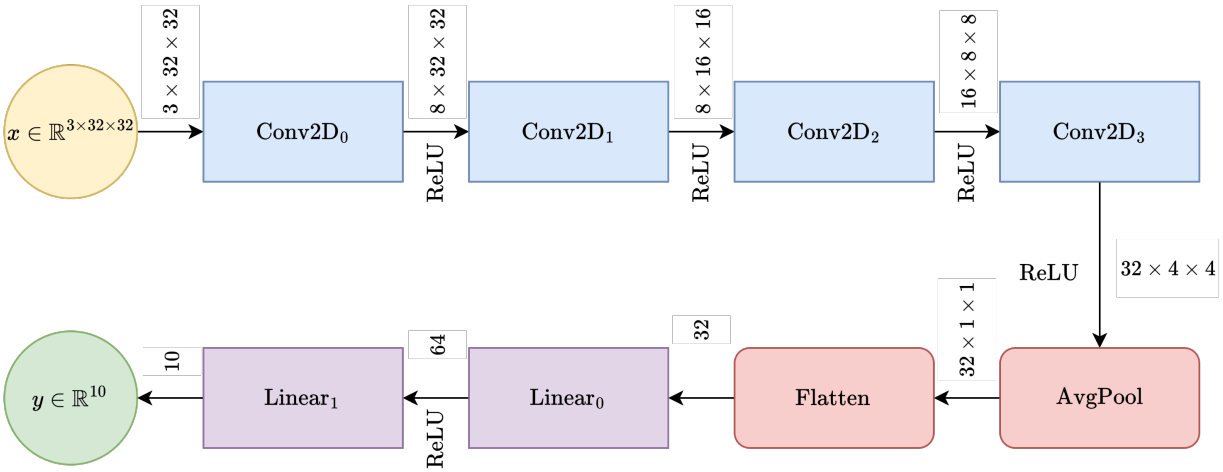


Figure 4.1: Example Conv2D FFNN: Example layout for a Conv2D FFNN with an expected input shape of $3 \times 32 \times 32$ and producing 10 output logits. Arrows indicate the pathway of the signal. Annotated dimensionalities indicate transmitted signal sizes from the originating layer. Each "Conv2D" layer is followed by a ReLU activation with downsampling on each Conv2D layer after the first. Spatial feature maps are collapsed using average pooling to a $32 \times 1 \times 1$ and then flattened to a feature vector of length 32 before going to the fully-connected classification head.

1. the input signal shape,
2. the model type,
3. the layout tuple that defines stage widths or channel counts,
4. the depth or residual settings that control repeated blocks,
5. the stage-local auxiliary layer choices, and
6. the output dimension required by the task or latent space.

Consider the simple CNN illustrated in Figure 4.1. Using PyTorch, the model can be defined as follows:

```
from torch import nn

model = nn.Sequential(
    nn.Conv2d(3, 8, kernel_size=3, padding=1),
```

```
    nn.ReLU(),
    nn.Conv2d(8, 8, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(8, 16, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(16, 32, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.AdaptiveAvgPool2d(output_size=1),
    nn.Flatten(),
    nn.Linear(32, 64),
    nn.ReLU(),
    nn.Linear(64, 10)
)
```

The input to this model is an image-like tensor with shape $C \times H \times W$, where C is the number of channels, H is the height, and W is the width. In PyTorch, batched image tensors are usually arranged as $B \times C \times H \times W$, where B is the batch size. For the CIFAR-10 input shape $3 \times 32 \times 32$, the first convolution maps the signal to 8 channels while preserving the spatial resolution. The later stride-2 convolutions reduce the spatial axes, producing a final convolutional feature map with 32 channels and spatial shape 4×4. Adaptive average pooling then reduces this to $32 \times 1 \times 1$; the flattening operation converts it to a length-32 vector, and the final dense head produces 10 class logits.

For more technical detail on convolutional layers, refer back to Chapter 2. Notably, the layer sequence must explicitly encode both the channel progression and the shape transition from a spatial tensor into a dense classifier head. A direct PyTorch definition is manageable for one model, but becomes more difficult to maintain when many nearby variants are required.

4.2.0.0.1 Auxiliary layers. The example omits normalization and dropout for simplicity. Normalization layers, such as batch normalization, can stabilize and accelerate training

and can reduce vanishing or exploding gradient behavior (Ioffe and Szegedy, 2015b). Dropout randomly removes signal components during training and is often used to improve robustness and reduce overfitting (Sierra et al., 2025). These layers are included in the experimental model definitions, but the examples initially omit them to make the architecture layout easier to see.

4.2.1 TMF CNN Example

The same model can be expressed in TMF as a model type plus a layout:

```
import torch_model_factory as tmf


model = tmf.Conv2D_FFNN(
    input_shape=(3,32,32),
    layout=(8, (8, 16, 32), (64, 10)),
    vectorize_fc_input=True
)
relu_factory = tmf.ActivationLayerFactory.ALL("ReLU")
relu_factory_nonfinal = tmf.ActivationLayerFactory.NONFINAL("ReLU")
model.initialize({
    "input": tmf.LayerFactories(activation=relu_factory),
    "hidden": tmf.LayerFactories(activation=relu_factory),
    "fc": tmf.LayerFactories(activation=relu_factory_nonfinal)
})
```

Note the layout tuple given in the above example.

$$\text{layout} = (8, (8, 16, 32), (64, 10)) \tag{4.1}$$

For `Conv2D_FFNN`, the first entry defines the input-stage output channel count, the second entry defines the hidden convolutional channel sequence, and the third entry defines the dense classifier sequence. Thus, the model can be read as follows: first map the input image

to 8 channels, then pass through hidden convolutional blocks with channel counts 8, 16, and 32, then pass the vectorized representation through dense layers of width 64 and 10. The final value 10 is the output dimension O for CIFAR-10 classification.

The flag `vectorize_fc_input=True` means that the final convolutional tensor is reduced by adaptive average pooling before the dense head. This conforms to the PyTorch example and makes the first dense layer depend on the final channel count instead of on the full product $C \cdot H \cdot W$. This kind of spatial aggregation can substantially reduce the number of learnable parameters in the classifier head, though reducing the signal capacity can also remove information (Yu et al., 2019). However, adaptive average pooling can still be effective at aggregating spatial features while preserving channel-level information (Abdu-Aguye et al., 2020).

The initialization dictionary assigns auxiliary-layer behavior to the named stages of the model. For the example above, all convolutional layers receive ReLU activations, while the dense classifier uses ReLU only for nonfinal layers.

Consider testing two CNN models: the original one and one where every hidden channel count is doubled. In a direct PyTorch definition, the layer sequence would need to be duplicated and edited. In TMF, the model type and initialization pattern are fixed while only the layout changes. This is the role of the layout tuple throughout the framework.

For example, many experiments describe a type of layouts using a base parameter scale P, a height H, and an output size O:

```
def build_layout(P:int, H:int, O:int):
    return (P, tuple(P << i for i in range(H)), (O,))
```

Here, P controls the feature scale, H controls the model's height (number of hidden blocks), and O is determined by the task. In a classification problem, O is the number of classes. In a router encoder, O may instead be the router latent dimension N.

4.2.2 TMF EncDec Example

Encoder-decoder (EncDec) models are where the procedural generation is most useful. Notably, such models can have more complicated output signal shapes, making it more complex to design a framework that ensures interfacing components are compatible. In an autoencoder, the model compresses an input into a latent representation and reconstructs an output with the same shape as the input. In a UNet-style encoder-decoder model, the output may instead be a transformed spatial signal, such as a segmentation map (Siddique et al., 2021). In both cases, the model definition must coordinate encoder downsampling, latent vectorization, decoder up-sampling, and final reconstruction shape.

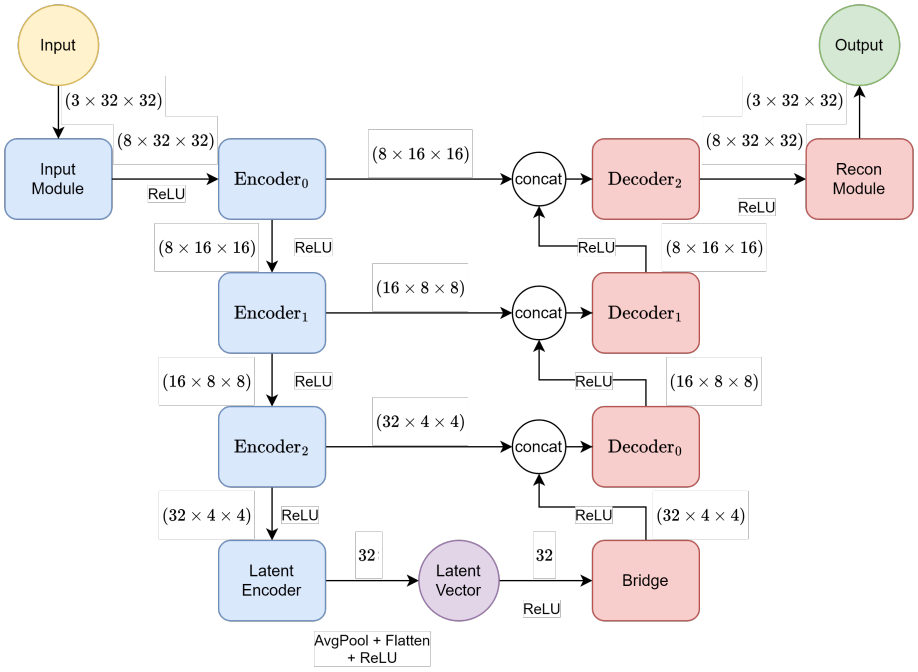


Figure 4.2: Example Conv2D UNet: input_shape $= (3, 32, 32)$, layout $= (8, (8, 16, 32), (32,))$. Arrows indicate the pathway of the signal. Annotated dimensionalities indicate transmitted signal sizes from the originating layer. The "Input Module" and "Recon Module" are given as same-size Conv2D layers that perform some pre/post-processing to the signal. Note the "Latent Encoder" produces a vector signal, collapsing the spatial axes to reduce parameters. Similarly, the "Bridge" component performs the opposite role, restructuring the latent vector back into a spatial signal. Per the UNet design, each "Encoder" passes its residuals to the equal-depth "Decoder", concatenating signals with the prior decoder/bridge module.

A UNet model with the layout shown in Figure 4.2 can be defined as follows:

```
import torch_model_factory as tmf


model = tmf.Conv2D_UNet(
    input_shape=(3,32,32),
    layout=(8, (8, 16, 32), (32,)),
    vectorize_latent_input=True
)
relu_factory = tmf.ActivationLayerFactory.ALL("ReLU")
relu_factory_nonfinal = tmf.ActivationLayerFactory.NONFINAL("ReLU")
model.initialize({
    "input": tmf.LayerFactories(activation=relu_factory),
    "encoder": tmf.LayerFactories(activation=relu_factory),
    "latent": tmf.LayerFactories(activation=relu_factory_nonfinal),
    "bridge": tmf.LayerFactories(activation=relu_factory),
    "decoder": tmf.LayerFactories(activation=relu_factory),
    "reconstruction": tmf.LayerFactories(activation=relu_factory_nonfinal)
})
```

The layout has the same three-part form as the feed-forward example, though different contexts within the autoencoder architecture. For `Conv2D_UNet`, the first entry gives the input-module channel count, the second entry gives the encoder channel sequence, and the third entry gives the latent dense sequence. The decoder and reconstruction stages are not separately listed in the tuple because they are derived from the encoder and input stages, roughly mirroring the layout in reverse. This keeps the specification compact while still defining the complete spatial path from input to latent representation and back to output.

This chapter uses the terms *input*, *encoder*, *latent*, *bridge*, *decoder*, and *reconstruction* to distinguish stages that are often grouped in the literature, grouping components by shared functionality.

4.3 Common Model Specification

All TMF models used here inherit the same high-level idea: a model has an input shape and a layout. The input shape specifies the expected sample shape before batching. For images, the sample shape is written below.

$$\text{input_shape} = (C, H, W) \tag{4.2}$$

Dense or already-vectorized signals may use a one-dimensional shape. The layout then specifies the sequence of widths or channel counts used by the selected model type.

For feed-forward models, the layout uses the following tuple.

$$\text{layout} = (\text{input_layout}, \text{hidden_layout}, \text{fc_layout}) \tag{4.3}$$

For autoencoder-style models, the layout uses the following tuple.

$$\text{layout} = (\text{input_layout}, \text{encdec_layout}, \text{latent_layout}) \tag{4.4}$$

The same tuple position can therefore mean different things depending on the model type.

The local behavior of generated stages is specified through stage names. Feed-forward models use stages such as `"input"`, `"hidden"`, and `"fc"`. Autoencoder models use `"input"`, `"encoder"`, `"latent"`, `"bridge"`, `"decoder"`, and `"reconstruction"`. These stage names allow the same architecture geometry to be paired with different activation, normalization, and dropout choices. The implementation's ordering places activation before normalization and dropout after normalization. Batch normalization was originally introduced before activation (Ioffe and Szegedy, 2015b), but post-activation normalization has also been studied (Ergen et al., 2022), with benchmarks indicating improved performance in some settings (Mishkin et al., 2017). Pooling, when included in convolutional groups, is placed directly

after the convolution, which is common in convolutional architectures (Rafiq et al., 2022; Simonyan and Zisserman, 2015).

The following sections describe the model types in terms of how these specifications define the model layout.

4.4 Feed-Forward Models

The feed-forward models share the following staged form.

$$f(x) = f_{\text{fc}} \circ f_{\text{hidden}} \circ f_{\text{input}}(x) \tag{4.5}$$

An omitted stage is treated as an identity map. The model type determines whether the stages are dense or convolutional and how the signal is vectorized before the final dense head.

4.4.1 Dense Feed-Forward Neural Networks

The dense feed-forward model, `Dense_FeedForwardNN`, first converts each input sample into a vector. If the input has shape $s_1 \times \cdots \times s_n$, the default vectorization step produces the following vector representation.

$$\text{vec}(x) \in \mathbb{R}^S \qquad S = \prod_{j=1}^{n} s_j \tag{4.6}$$

Here, S is the total number of scalar input features. For spatial inputs, the implementation also supports an adaptive-average-pooling vectorization mode that first reduces the spatial dimensions to 1×1 and then flattens the result. This is the dense-model analog of the vectorization used in the convolutional CNN example.

The dense layout uses the following tuple.

$$(\text{input_layout}, \text{hidden_layout}, \text{fc_layout}) \tag{4.7}$$

If `input_layout` is present, the vectorized input is first mapped to that width. The `hidden_layout` tuple defines the sequence of hidden widths, and `block_depth` determines how many dense layer groups are repeated for each hidden width. The `fc_layout` tuple defines the final dense output chain. For classification, its final entry is the number of classes.

The following composition summarizes a dense model.

$$\hat{y} = \mathrm{FC}\left(\mathrm{Hidden}\left(\mathrm{Input}\left(\mathrm{vec}(x)\right)\right)\right) \tag{4.8}$$

Each named component is defined by its corresponding portion of the layout.

4.4.2 Convolutional Feed-Forward Neural Networks

The convolutional feed-forward model, `Conv2D_FFNN`, uses the same layout tuple, but the input and hidden stages operate on tensors with shape $C \times H \times W$. The first layout entry gives the channel count produced by the optional input module. The hidden-layout tuple gives the channel count after each hidden convolutional block.

$$\text{hidden_layout} = (C_1, C_2, \ldots, C_m) \tag{4.9}$$

By default, the hidden pathway changes the channel count at each block and downsamples spatial dimensions between hidden blocks. The first hidden block preserves spatial resolution, while later blocks downsample. This matches the common pattern where early features are extracted before repeated spatial reduction.

The parameter `block_depth` controls how many convolutional groups are used inside each hidden block. With `block_depth=1`, a hidden block is a single channel-transition block. With `block_depth=2`, each hidden block applies two convolutional groups at the same target channel count before moving to the next block. Conceptually, each hidden block follows this

pathway.

$$X_{i-1} \to \text{Conv2D}_{i,1} \to \cdots \to \text{Conv2D}_{i,d_i} \to X_i \tag{4.10}$$

Here, d_i is the block depth for hidden block i.

After the hidden convolutional pathway, the final dense stage receives either the full flattened feature map or a channel vector produced by adaptive average pooling. When `vectorize_fc_input=True`, the transition is defined below.

$$u_{\text{fc}} = \text{Flatten}\left(\text{AdaptiveAvgPool2d}_{1\times1}(X_m)\right) \tag{4.11}$$

In this case, the classifier head receives only the final channel count. When the flag is false, the full final feature map is flattened.

The following stage process therefore defines the convolutional feed-forward model.

$$\hat{y} = \text{FC}\left(v\left(\text{ConvHidden}\left(\text{ConvInput}(x)\right)\right)\right) \tag{4.12}$$

Here, v denotes the selected vectorization operation.

4.4.3 Residual Convolutional Feed-Forward Neural Networks

The residual convolutional model, `Conv2D_Res`, keeps the same feed-forward layout but changes the hidden pathway by inserting residual additions inside the convolutional hidden blocks. The extra parameter `residuals_skip_length` determines how often a residual connection is applied within each block. If it is omitted, each block uses its full block depth as the residual interval. If a tuple is provided, each tuple entry corresponds to the matching hidden block.

For a hidden block, the residual update has the following form.

$$X \leftarrow X + \pi_i(X_{\text{residual}}) \tag{4.13}$$

Here, the projection π_i is used only when the residual tensor and current tensor require different channel counts or spatial sizes. This makes the residual version a controlled variant of `Conv2D_FFNN`: the layout, output dimension, and final classifier head are the same, while the hidden computation gains skip-connected pathways.

4.5 Encoder–Decoder and Autoencoder Models

The encoder-decoder models use the following layout tuple.

$$\text{layout} = (\text{input_layout}, \text{encdec_layout}, \text{latent_layout}) \tag{4.14}$$

The first entry defines the channel count after the input module, and the second entry defines the encoder channel sequence. The third entry defines the dense latent sequence, whose final entry is the latent dimension. From this specification, the decoder channel sequence and reconstruction pathway are inferred by reversing the encoder-side shape progression.

The common staged structure is shown below.

$$\text{input} \to \text{encoder} \to \text{latent} \to \text{bridge} \to \text{decoder} \to \text{reconstruction} \tag{4.15}$$

The encoder reduces the spatial signal into a compact representation. The latent stage maps the encoded tensor into a dense vector. The bridge maps that vector back into a spatial tensor compatible with the decoder. The decoder decodes the spatial resolution, and the reconstruction stage maps the decoder output to the required output channel count.

4.5.1 Convolutional Autoencoders

The base convolutional autoencoder, `Conv2D_AE`, implements the basic downsampling-upsampling pathway.

$$z = \text{Encode}(x) \tag{4.16}$$

$$\hat{x} = \text{Decode}(z) \tag{4.17}$$

Here, z is the latent representation and $\hat{x}$ is the reconstruction. The encoder channel sequence is determined by `encdec_layout`. The block-depth parameter determines how many convolutional groups are used for each encoder and decoder block. The vectorization parameter determines whether the latent module receives the full encoded feature map or an adaptive-average-pooled channel vector.

The bridge is an architectural transition between dense latent space and spatial decoder space. It maps the latent vector back to the final encoder spatial size so that the decoder reverses the encoder pathway. In the notation of this chapter, the base autoencoder can be read as the following composition.

$$\hat{x} = \text{reconstruction}\,(\text{Decoder}\,(\text{Bridge}\,(\text{Latent}\,(\text{Encoder}\,(\text{input}(x)))))) \tag{4.18}$$

This expression describes the model layout without requiring the reader to track each internal builder operation.

During an autoencoder training step, the reconstruction loss is based on the mean-squared error between x and $\hat{x}$.

$$\mathcal{L}_{\text{recon}} = \frac{1}{B}\sum_{b=1}^{B}\|x_b - \hat{x}_b\|_2^2 \tag{4.19}$$

When an autoencoder is used as a router encoder, this reconstruction loss is returned as an auxiliary router loss while the latent representation is used for routing.

4.5.2 UNet-Style Autoencoders

The `Conv2D_UNet` model uses the same layout as `Conv2D_AE`, but preserves encoder outputs as skip tensors that are concatenated into the decoder. If s_i denotes the output of encoder block i, then decoder block i receives the corresponding reversed skip tensor.

$$X_{\text{dec},i} = \text{DecoderBlock}_i\left(\text{Concat}_C\left(s_{m-i+1}, X_{\text{dec},i-1}\right)\right) \tag{4.20}$$

Here, m is the number of encoder blocks, and Concat_C denotes concatenation across the channel dimension. Because each decoder block receives both the decoder signal and an encoder skip signal, the decoder input channel count is effectively doubled at those stages.

This model type is helpful when the reconstruction or output should preserve spatial details from earlier encoder stages. In the current implementation, direct decoding from a latent vector is not the public interface for `Conv2D_UNet`, because the decoder decodes the skip tensors collected during encoding. The forward pass therefore encodes the input, stores the skip tensors, and decodes with those skips.

4.5.3 Variational Convolutional Autoencoders

The `Conv2D_VAE` model keeps the same encoder-decoder layout but changes the latent interpretation. Instead of producing a single latent vector directly, the latent stage produces a mean vector and a log-variance vector.

$$(\mu, \ell) = \text{Split}(\text{VAELatent}(u_{\text{latent}})) \tag{4.21}$$

A latent sample is then obtained through reparameterization.

$$z = \mu + \varepsilon \odot \exp\left(\tfrac{1}{2}\ell\right) \qquad \varepsilon \sim \mathcal{N}(0, I) \tag{4.22}$$

During evaluation, the implementation can optionally disable reparameterization and use $z = \mu$ instead. This provides a deterministic latent signal for analysis or routing.

The VAE objective combines reconstruction behavior with a KL-divergence term. Let $\mathcal{L}_{\text{mse}}$ denote the reconstruction mean-squared error over the batch. The KL term is defined below.

$$\mathcal{L}_{\text{kld}} = \text{mean}\left[-\tfrac{1}{2}\sum\left(1 + \ell - \mu^2 - \exp(\ell)\right)\right] \tag{4.23}$$

The implementation also uses a learned scalar decoder log-variance ℓ_{dec}. If D is the number of scalar entries in one input sample, the implemented loss terms are defined below.

$$\sigma_{\text{dec}} = \exp\left(\tfrac{1}{2}\ell_{\text{dec}}\right) \tag{4.24}$$

$$\mathcal{L}_{\text{recon}} = D\log(\sigma_{\text{dec}}) + \frac{D}{2\sigma_{\text{dec}}^2}\mathcal{L}_{\text{mse}} \tag{4.25}$$

$$\mathcal{L}_{\text{vae}} = \mathcal{L}_{\text{recon}} + \mathcal{L}_{\text{kld}} \tag{4.26}$$

Notably, the final latent width still represents the intended latent dimension, but the internal VAE latent layer produces twice that width so it can represent both μ and ℓ.

4.5.4 Residual Variational Convolutional Autoencoders

The `Conv2D_ResVAE` model extends `Conv2D_VAE` by adding residual connections inside both the encoder and decoder paths. Its layout still follows the same tuple form.

$$(\text{input_layout}, \text{encdec_layout}, \text{latent_layout}) \tag{4.27}$$

The residual-skip-length parameter determines how often residual additions occur inside each encoder and decoder block. This gives the experiments a VAE-style router or representation model with deeper paths while retaining the same compact layout notation used by the other encoder-decoder models.

4.6 Mixture-of-Experts Model Definition

Chapter 3 defines the MoE architecture mathematically. In the implementation framework, a complete MoE model is specified by composing three parts:

1. a router encoder model ϕ,
2. a router logit function that maps latent encodings to expert probabilities, and
3. a sequence of expert models $\Theta = \{\theta_1, \ldots, \theta_K\}$.

The TMF wrapper `ExpertsMixture` then coordinates routing, sparse expert evaluation, and output aggregation.

A MoE configuration therefore includes the expert count K, the router latent dimension N, the expert output dimension O, the router model type and layout, the router type, the expert model type and layout, and the sparse inference value k. This aligns with the notation from Chapter 3: the router produces probabilities over K experts, the top-k selection defines the active expert subset, and the selected expert outputs are aggregated to produce the final prediction.

4.6.1 Router Encoder and Expert Inputs

The router begins by encoding an input sample x into a latent vector.

$$z = \phi(x) \qquad z \in \mathbb{L}^N \tag{4.28}$$

The encoder ϕ may be a feed-forward model or an autoencoder-style model. If it is a feed-forward model, the router receives only the latent representation. If it is an autoencoder, the router receives both the latent representation z and a reconstruction $\hat{x}$. The reconstruction can optionally modify the signal sent to the experts.

The supported expert-input modes are defined as follows.

$$T_{\text{none}}(x, \hat{x}) = x \tag{4.29}$$

$$T_{\text{add}}(x, \hat{x}) = x + \hat{x} \tag{4.30}$$

$$T_{\text{concat}}(x, \hat{x}) = \text{Concat}_C(x, \hat{x}) \tag{4.31}$$

The `none` mode sends the original input to the experts. The `add` mode sends a reconstruction-augmented input. The `concat` mode concatenates the original input and reconstruction along the channel dimension, which doubles the expert input channel count for image-like data. This detail matters for model definition because the expert layout must match the expert input shape produced by the router.

4.6.2 Routing Result Layout

For a batch of size B, the router computes a logit matrix with the following shape.

$$R \in \mathbb{R}^{K \times B} \tag{4.32}$$

Applying softmax over the expert dimension gives the dense routing probabilities $P \in \mathbb{R}^{K \times B}$. The implementation stores these values in an expert-major layout, matching the notation used in Chapter 3. Top-k selection produces a sparse routing matrix with the following shape.

$$P^k \in \mathbb{R}^{K \times B} \tag{4.33}$$

Unselected expert/sample pairs have probability zero, and each sample's selected probabilities are re-normalized to sum to one.

The `RoutingResult` object stores the expert input tensor, latent encodings, logits, dense probabilities, top-k probabilities, top-k indices, sparse top-k probabilities, and analogous top-1 values. This storage layout is relevant because the MoE evaluates only the expert/sample

pairs that have nonzero sparse routing probability.

4.6.3 Expert Stack and Aggregation

The expert stack is written as a sequence of K models.

$$\Theta = (\theta_1, \theta_2, \ldots, \theta_K) \tag{4.34}$$

Each expert maps the expert input signal to an output vector of size O. In image-classification experiments, O is the number of classes. The MoE wrapper verifies that the number of expert modules equals the router's K.

Given a routing result, the implementation builds an expert-logit tensor with the following shape.

$$Y_{\text{sparse}} \in \mathbb{R}^{K \times B \times O} \tag{4.35}$$

For expert i, only the samples selected for that expert are evaluated. Unselected entries are zero and do not contribute to the sparse mixture because their corresponding routing probabilities are zero. The top-k output can be written as follows.

$$\hat{y}_b = \sum_{i=1}^{K} P_{i,b}^{k} \theta_i(x_{e,b}) \tag{4.36}$$

Here, $x_{e,b}$ is the expert input for sample b. When $k = 1$, this reduces to the selected expert's prediction. When $k = K$, every expert contributes, and the result is the dense mixture.

For marginal negative log-likelihood training over selected experts, the stored expert logits are converted into expert log-probabilities and aggregated with the log routing probabilities.

$$\log p_{\text{MoE}}(y \mid x_b) = \operatorname{logsumexp}_{i=1}^{K} \left(\log P_{i,b}^{k} + \log p_i(y \mid x_b) \right) \tag{4.37}$$

When $P_{i,b}^{k} = 0$, the corresponding log routing probability is treated as $-\infty$, so unselected

experts are excluded from the mixture.

4.6.4 Router Variants

The router type determines how the latent representation z is converted into logits. The expert layouts and expert model type do not need to change when the router type changes.

The *linear router* associates each expert with a learned embedding $e_i \in \mathbb{L}^N$ and computes the following score.

$$r_i(z) = z^\top e_i \tag{4.38}$$

The *cosine router* normalizes the latent vector and expert embeddings before computing similarity.

$$r_i(z) = \left\langle \frac{z}{\|z\|_2}, \frac{e_i}{\|e_i\|_2} \right\rangle \tag{4.39}$$

The *centroid router* represents each expert by a learned centroid c_i and routes according to negative squared distance.

$$r_i(z) = -\frac{\|z - c_i\|_2^2}{N} \tag{4.40}$$

This router also supports initialization by applying K-means to sampled latent encodings. The *RBF router* extends the centroid router by learning a width parameter σ_i for each expert.

$$r_i(z) = -\frac{\|z - c_i\|_2^2}{2\sigma_i^2} - N \log \sigma_i \tag{4.41}$$

These are the same routing functions discussed in Chapter 3; this chapter records how they are selected as interchangeable implementation components within the same MoE specification.

A typical procedural MoE construction therefore has the following form:

```
import torch_model_factory as tmf

N, K, O = 32, 4, 10

```

```
encoder_factories = ...
encoder = tmf.Conv2D_VAE(...).initialize(encoder_factories)

router = tmf.CentroidRouter(
    model=encoder,
    N=N, K=K,
    signal_modifier="concat"
)

experts = []
for _ in range(K):
    experts.append(tmf.Conv2D_FFNN(...).initialize(expert_factories))

model = tmf.ExpertsMixture(router=router, experts=experts, O=O)
```

Where an ellipsis is used to abbreviate a component already described previously.

This example demonstrates the overall process used by the experiments. First, define a router encoder from an input shape and layout. Second, select a router function and provide N and K. Third, instantiate K experts from the expert model type and expert layout. Finally, wrap the router and experts into `ExpertsMixture`. The resulting architecture is compactly described by its router encoder layout, router type, expert layout, number of experts, sparse top-k value, and output dimension.

4.7 Summary

TMF provides a consistent way to define the model types used throughout this work. Its central abstraction is the model specification: an input shape, a layout tuple, a model type, and a small number of type-specific controls. Dense, convolutional, residual, autoencoder, UNet, VAE, residual VAE, and MoE architectures are all defined through this same pattern.

The chapter intentionally emphasizes model layout over layer-builder mechanics. The

builder utilities provide the PyTorch modules required by each stage. Still, the experimental interpretation depends primarily on the layout: how the signal moves from input to hidden representation, from hidden representation to latent representation, from latent representation back to a spatial signal, or from router encoding to selected experts. This layout-focused description provides the architectural vocabulary used for the baselines and experiments that follow.

Chapter 5

Baseline Experiments

5.1 Overview

This chapter details the baseline experiments and analysis to contextualize later experiments testing the presented hypotheses. Baselines describe an image classification problem space on the MNIST (LeCun et al., 1998) and CIFAR-10 and CIFAR-100 (Krizhevsky, 2009) datasets. This chapter further covers the AI/ML methodologies used, including applied problem spaces, neural architecture definitions, loss and auxiliary loss functions, and training processes. Finally, baseline experimentation results are presented here. These baseline references and insights will be used to evaluate the hypotheses, such as evaluating simple classifier networks and contrasting their performance with a full MoE system.

5.2 Classification Datasets

The experimental problem spaces are selected to provide a gradual progression in difficulty while remaining representative of small-to-mid-sized image classification tasks. MNIST is used as an initial baseline and proof of concept, since its low visual complexity and well-understood structure make it suitable for validating core MoE behavior, routing dynamics, and expert specialization under relatively controlled conditions. CIFAR-10 then serves as the primary small-scale analysis task, introducing substantially greater visual variability, more complex feature structure, and a more realistic challenge for evaluating whether MoE methods provide meaningful benefits beyond simple benchmark settings. Finally, CIFAR-

100 is used to examine performance in a mid-sized problem space, where the larger number of classes and finer inter-class distinctions place greater pressure on both the routing mechanism and the expert models.

5.2.1 MNIST

MNIST is a grayscale handwritten digit dataset containing ten classes, one for each digit from 0 through 9. Its low-dimensional visual structure and limited background complexity make it well suited for establishing baseline behavior in the proposed MoE systems. In this work, MNIST serves primarily as a proof-of-concept dataset for observing whether the router and experts can learn meaningful specialization under relatively simple conditions. When normalization is applied, the dataset uses a mean of 0.1307 and a standard deviation of 0.3081. The implemented training pipeline also supports random cropping with padding for robustness to small spatial offsets. An example of the dataset structure and visual style is shown in Figure 5.1.



Figure 5.1: Example samples from the MNIST dataset (test partition), illustrating the relatively simple grayscale handwritten digit classification task.

5.2.2 CIFAR-10

CIFAR-10 is a small-scale color image classification dataset containing ten object classes drawn from natural images. Compared to MNIST, it introduces substantially greater visual diversity through variation in color, texture, pose, background content, and object appearance. This makes it a more realistic small-scale benchmark for evaluating whether MoE methods provide benefits in a setting where the classification boundaries are less structurally uniform. In the implemented configuration, CIFAR-10 may be trained with random cropping at 32×32 using padding of 4, together with random horizontal flipping, and uses normalization constants with mean $(0.4914, 0.4822, 0.4465)$ and standard deviation $(0.2470, 0.2435, 0.2616)$. These transformations are intended to improve robustness to small spatial translations and mirror-symmetric viewpoint variation while maintaining the semantic content of the images. An example of the dataset and its characteristic visual complexity is shown in Figure 5.2.

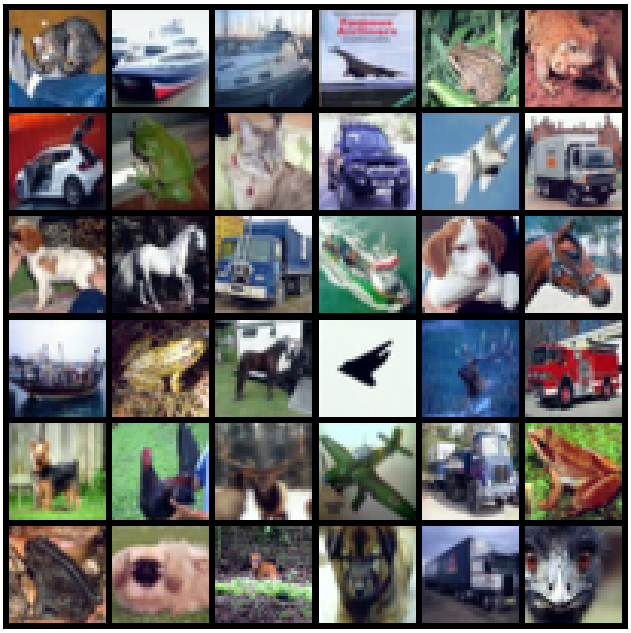


Figure 5.2: Example samples from the CIFAR-10 dataset (test partition), showing a more visually diverse natural-image classification task across ten classes.

Table 5.1 gives the CIFAR-10 class-index mapping in a compact two-row layout.

Table 5.1: CIFAR-10 class index mapping.

Index	Class	Index	Class	Index	Class	Index	Class	Index	Class
0	airplane	1	automobile	2	bird	3	cat	4	deer
5	dog	6	frog	7	horse	8	ship	9	truck

5.2.3 CIFAR-100

CIFAR-100 extends the CIFAR image setting to a substantially larger and finer-grained classification problem, containing one hundred classes instead of ten. While the image scale is small, the increased number of categories and the greater similarity between many classes make the task substantially more demanding. In this study, CIFAR-100 is used to examine whether the proposed MoE approaches are effective as the problem space grows in complexity and requires more refined routing and expert specialization. Its preprocessing follows the same general pattern as CIFAR-10, with optional random cropping at 32×32 using padding of 4, optional random horizontal flipping during training, and normalization using mean $(0.5071, 0.4867, 0.4408)$ and standard deviation $(0.2675, 0.2565, 0.2761)$. These transformations provide mild invariance to translation and left-right orientation while retaining the fine-grained visual distinctions necessary for classification. An example of the dataset is shown in Figure 5.3.

Table 5.2 lists the CIFAR-100 superclass grouping used to organize the one hundred fine-grained class labels. Note that the twenty superclasses are evenly split to match the number of classes and are grouped sequentially. E.g., the "aquatic mammals" details class indices 0-through-4 (0-index based), then "fish" indices 5-through-9 and so on.

5.2.4 Transformations

Normalization, random rotations, and random cropping are techniques shown to improve generalization by varying the features in the training data to prevent overfitting. Notably, this is similar to dropout layers randomly zeroing signals (turning off neural activations)

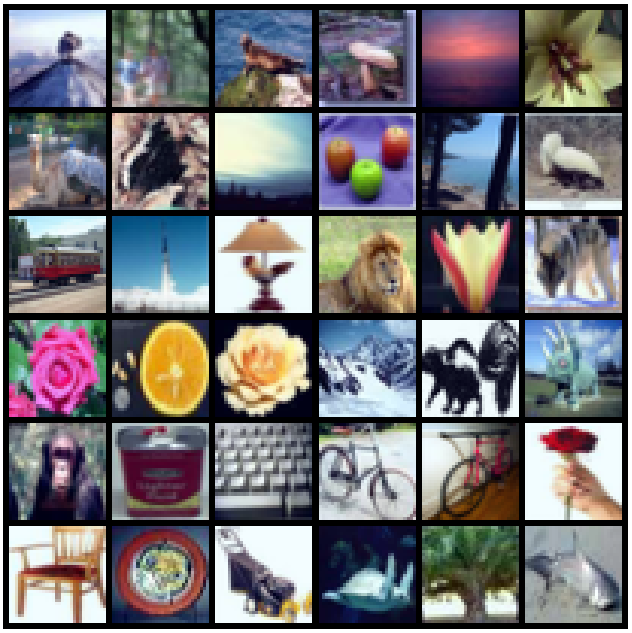


Figure 5.3: Example samples from the CIFAR-100 dataset (test partition), illustrating the increased class diversity and finer-grained natural-image classification challenge.

Table 5.2: CIFAR-100 superclass and class mapping.

Superclass	Classes
aquatic mammals	beaver, dolphin, otter, seal, whale
fish	aquarium fish, flatfish, ray, shark, trout
flowers	orchids, poppies, roses, sunflowers, tulips
food containers	bottles, bowls, cans, cups, plates
fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers
household electrical devices	clock, computer keyboard, lamp, telephone, television
household furniture	bed, chair, couch, table, wardrobe
insects	bee, beetle, butterfly, caterpillar, cockroach
large carnivores	bear, leopard, lion, tiger, wolf
large man-made outdoor things	bridge, castle, house, road, skyscraper
large natural outdoor scenes	cloud, forest, mountain, plain, sea
large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
medium-sized mammals	fox, porcupine, possum, raccoon, skunk
non-insect invertebrates	crab, lobster, snail, spider, worm
people	baby, boy, girl, man, woman
reptiles	crocodile, dinosaur, lizard, snake, turtle
small mammals	hamster, mouse, rabbit, shrew, squirrel
trees	maple, oak, palm, pine, willow
vehicles 1	bicycle, bus, motorcycle, pickup truck, train
vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

to encourage resilient behaviors that do not rely on specific features or signals. All of the evaluated datasets undergo normalization; however, only CIFAR-10 and CIFAR-100 are flipped/rotated. Notably, the MNIST dataset is a dataset of digits, which can be sensitive to such transforms. E.g., a "6" rotated 180-degrees becomes a "9" or a "3" horizontally mirrored is no longer a "3". The CIFAR datasets, by contrast, are more compatible with horizontal flipping because many natural-image object classes are recognizable under left-right reflection. However, a flipped image of a boat may typically involve replacing the ocean with the sky and the sky with the ocean. Contextually, this does not make sense, but invariably the object of relevance will still be a boat.

5.3 Classifier Baselines

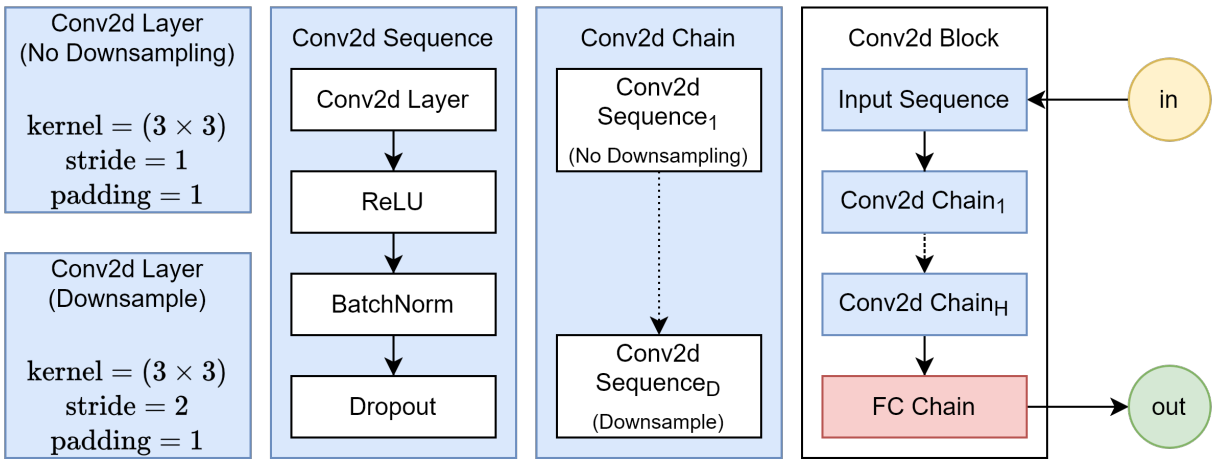


Figure 5.4: Illustration of the conv2d neural network architecture used to train the classifier baselines. The model is composed of H (model height) Conv2d blocks, where each block is itself composed of D (block depth) Conv2d chains. Each chain is a structured set of layers, detailing Conv2D, pooling, and regularization. Signals are fed into the model starting with the input sequence and flow through the model's conv blocks and into the fully-connected (FC) output head.

Models are implemented using TMF, as detailed in the previous chapter. In this work, TMF provides a procedural interface for defining repeated architectural patterns while pre-

serving a compact configuration format for experimental sweeps. Rather than specifying each convolutional block manually, the model definition separates the high-level architectural layout from the layer-level construction rules. Notably, the same implementation pattern can be reused across MNIST, CIFAR-10, and CIFAR-100 while varying only the model type and scaling parameters.

```
hyperparams = {
    "input": tmf.LayerFactories(
        activation=tmf.ActivationLayerFactory.ALL("ReLU"),
        normalization=tmf.NormLayerFactory.ALL(norm_mode="batch"),
        dropout=None
    ),
    "hidden": tmf.LayerFactories(
        activation=tmf.ActivationLayerFactory.ALL("ReLU"),
        normalization=tmf.NormLayerFactory.ALL(norm_mode="batch"),
        dropout=tmf.DropoutLayerFactory.ALL(dropout)
    ),
    "fc": tmf.LayerFactories()
}
```

This configuration exemplifies the patterned structure of the generated classifier models. The input and hidden convolutional stages use ReLU activations and batch normalization throughout, which provides a consistent nonlinear transformation and normalization scheme across all convolutional blocks. Dropout is omitted from the input stage and, when enabled, is applied through the hidden-stage layer factory. The fully connected stage is left without additional layer factories, so the final classifier projection maps the learned feature representation directly to the target class logits.

This separation between architecture layout and layer factory configuration supports controlled comparisons across model scales. The parameters P, H, and D determine the width, number of downsampling stages, and convolutional block depth, respectively, while the layer factory configuration determines the repeated operations used inside those stages.

As a result, the experiments can attribute performance differences primarily to architectural scaling choices rather than unrelated changes in activation functions, normalization layers, or regularization strategy.

Across datasets, the convolutional layers used ReLU activations and batch normalization, with variable dropout applied in the hidden layer configuration, dependent on the applied dataset: dropout = 0.1 for MNIST, dropout = 0.2 for CIFAR-10, and dropout = 0.5 for CIFAR-100.

Loss function: loss_function = CrossEntropyLoss

Optimizer configuration:

1. optimizer = AdamW
2. learning_rate = 3×10^{-4}
3. weight_decay = 5×10^{-2}
4. gradient_clipping = 1.0

Training configuration:

1. max_epochs = 1,000
2. batch_size = 256
3. train/val split = 70%/30%
4. monitor = val_loss
5. warmup = 100
6. patience_threshold = 30
7. min_delta = 1×10^{-3}

Dataset-specific label smoothing was used for the classifier baselines. MNIST was trained without label smoothing, CIFAR-10 used label smoothing of 0.05, and CIFAR-100 used label smoothing of 0.10. Across datasets, the convolutional layers used ReLU activations and batch normalization, with dropout of 0.2 applied in the hidden layer configuration. Convolutional layers, ReLU activations, batch normalizations, and dropouts are discussed in Chapter 2 (Background). Note then the *label smoothing* regularization, which helps soften predictions to reduce the magnitude of errors. Notably, this can help reduce overfitting on more complex datasets, such as CIFAR-100 (Zhang et al., 2021a).

The classifier baselines are standard multiclass image classification models. For each input image $\mathbf{x}_b$, the model produces a vector of class logits.

$$\hat{\mathbf{y}}_b = f_\theta(\mathbf{x}_b) \tag{5.1}$$

Here, $\hat{\mathbf{y}}_b \in \mathbb{R}^O$ and O is the number of target classes. The cross-entropy loss applies a softmax normalization over the output-class dimension.

$$p_{b,o} = \frac{\exp(\hat{y}_{b,o})}{\sum_{j=1}^{O} \exp(\hat{y}_{b,j})} \tag{5.2}$$

The resulting probabilities are used to evaluate the negative log-likelihood of the target class distribution. For a batch of B examples, the training objective is defined below.

$$\mathcal{L}_{\text{CE}} = -\frac{1}{B} \sum_{b=1}^{B} \sum_{o=1}^{O} \tilde{y}_{b,o} \log p_{b,o} \tag{5.3}$$

Here, $\tilde{y}_{b,o}$ is the target distribution for example b and output class o. When label smoothing is disabled, $\tilde{y}_{b,o}$ is a one-hot target. When label smoothing is enabled, the target distribution is modified to reduce overconfidence by distributing some of the mass across all classes, not just the target.

The classifier models are optimized with AdamW using weight decay. During each train-

ing step, the optimizer gradients are cleared, logits are computed for the current batch, cross-entropy loss is evaluated, backpropagation is performed, and gradient clipping is applied before the optimizer update. The learning rate is scheduled with cosine annealing over the full training run.

The primary metric reported for the classifier baselines is top-1 accuracy acc1. For a batch of B examples, top-1 accuracy is computed below.

$$\text{acc1} = \frac{1}{B}\sum_{b=1}^{B} \mathbb{1}\left[\arg\max_{o} \hat{y}_{b,o} = y_b\right] \tag{5.4}$$

Here, $\hat{y}_{b,o}$ is the logit assigned to output class o for example b, y_b is the target class label, and $\mathbb{1}[\cdot]$ is the indicator function. Thus, acc measures the fraction of examples for which the highest-scoring class is the correct class.

The acc5 metric details top-5 accuracy. Rather than requiring the highest-scoring class to be correct, acc5 checks whether the correct label appears among the model's five highest-scoring classes.

$$\text{acc5} = \frac{1}{B}\sum_{b=1}^{B} \mathbb{1}\left[y_b \in \text{TopK}(\hat{\mathbf{y}}_b, 5)\right] \tag{5.5}$$

Acc5 is useful for datasets with many classes, such as CIFAR-100, because it measures whether the model ranks the correct class among its most likely predictions, even when the final top-1 prediction is incorrect.

5.3.1 MNIST

Table 5.3 summarizes a sweep over convolutional feed-forward classifiers trained on MNIST, with each result averaged over three random seeds. The best-performing configuration observed is $\{P = 4, H = 4, D = 2\}$, which reaches 99.56% mean top-1 test accuracy with 19,046 learnable parameters. It also yields the lowest mean test loss, 0.0142, and a mean top-5 accuracy of 99.9981% (rounded to 1.0000 in the table). Notably, $\{P = 4, H = 4, D = 2\}$ also has the most learnable parameters, so that other configurations may be more parameter-efficient.

Table 5.3: MNIST classifier baseline results averaged over three trials. Each trial is identical except for the initial RNG seeding that controls initial values, train/val partition splits (but not split sizes), and how training samples are generated. P is the parameter scaling factor controlling feature-map width. H is the model height, defined as the number of downsample blocks. D is the block depth, defined as the number of convolutional layers per block. Loss, acc1, and acc5 metrics are rounded to 4 decimals.

FFNN	P	H	D	Params	Epochs	Loss	Acc1	Acc5
Conv2D	1	2	1	74	489.0	1.6877	0.3807	0.8975
Conv2D	1	2	2	125	189.3	2.0096	0.2232	0.7372
Conv2D	1	3	1	174	371.7	1.5029	0.5048	0.9284
Conv2D	1	3	2	377	281.3	1.4594	0.4895	0.9008
Conv2D	1	4	1	518	395.7	0.5798	0.8350	0.9882
Conv2D	1	4	2	1,313	489.0	0.2549	0.9245	0.9961
Conv2D	2	2	1	192	492.7	1.1798	0.5921	0.9715
Conv2D	2	2	2	384	269.7	1.4644	0.4149	0.9020
Conv2D	2	3	1	536	515.7	0.7591	0.7522	0.9860
Conv2D	2	3	2	1,320	505.7	0.2446	0.9269	0.9968
Conv2D	2	4	1	1,800	463.0	0.1324	0.9602	0.9984
Conv2D	2	4	2	4,920	337.7	0.0528	0.9828	0.9998
Conv2D	3	2	1	364	430.0	0.9697	0.6759	0.9717
Conv2D	3	2	2	787	334.3	0.8560	0.6896	0.9668
Conv2D	3	3	1	1,096	503.7	0.3027	0.9156	0.9954
Conv2D	3	3	2	2,839	460.0	0.0758	0.9766	0.9995
Conv2D	3	4	1	3,856	331.0	0.0584	0.9817	0.9997
Conv2D	3	4	2	10,831	280.7	0.0184	0.9941	1.0000
Conv2D	4	2	1	590	474.3	0.4870	0.8518	0.9932
Conv2D	4	2	2	1,334	493.7	0.3749	0.8851	0.9944
Conv2D	4	3	1	1,854	360.0	0.2113	0.9362	0.9975
Conv2D	4	3	2	4,934	369.3	0.0357	0.9888	0.9999
Conv2D	4	4	1	6,686	299.3	0.0323	0.9904	1.0000
Conv2D	4	4	2	19,046	155.0	0.0142	0.9956	1.0000

Model height is observed to be the most consistently beneficial architectural factor in this sweep. For every fixed combination of P and D, increasing H from 2 to 3 to 4 improves mean top-1 test accuracy. With $P = 2$ and $D = 2$, for example, accuracy rises from 41.49% at $H = 2$ to 92.69% at $H = 3$ and 98.28% at $H = 4$. Likewise, with $P = 4$ and $D = 2$, accuracy increases from 88.51% to 98.88% and 99.56% as H increases from 2 to 4.

The Similar parameter comparisons also suggest that allocating capacity to height can be more efficient than allocating it only to width. The $\{P = 1, H = 4, D = 2\}$ model uses 1,313 parameters and reaches 92.45% accuracy, whereas the wider but shallower $\{P = 4, H = 2, D = 2\}$ model uses slightly more parameters (1,334) but reaches only 88.51%. The same pattern appears for $D = 1$: $\{P = 2, H = 4, D = 1\}$ reaches 96.02% accuracy with 1,800 parameters, outperforming $\{P = 4, H = 3, D = 1\}$, which reaches 93.62% with 1,854 parameters. Block depth is less uniformly beneficial at low height: at $H = 2$, increasing D from 1 to 2 reduces accuracy for $P = 1$ and $P = 2$, but improves it for $P = 3$ and $P = 4$.

Note that for $H = 4$ there are 3 downsampling stages, since the first hidden block–after the input sequence–skips the downsampling step, as detailed in 5.4. For the MNIST images, which are $1 \times 28 \times 28$, this downsampling looks like $* \times 28 \times 28 \to * \times 14 \times 14 \to * \times 7 \times 7 \to * \times 3 \times 3$. Thus, some configurations like $H = 2$ likely struggle because the adaptive-pooling has a greater amount of information loss.

Figure 5.5 and Figure 5.6 illustrate the efficiency of each architectural approach by comparing the loss and model acc1 and acc5 metrics, respectively, over the number of learnable parameters. As a quick descriptor, the accuracy metrics (indicated by dashed lines) detail greater-parameter efficiencies when shifted to the left and/or above other points; reduced parameters for the same accuracies with fewer parameters are indicated by shifting the curve to the left, and improved accuracies with similar parameters are indicated by shifting the curve upwards.

There are two immediate observations: the generalized efficacy of each added parameter is exponentially reduced in terms of the accuracy it yields, and the distributions in both plots

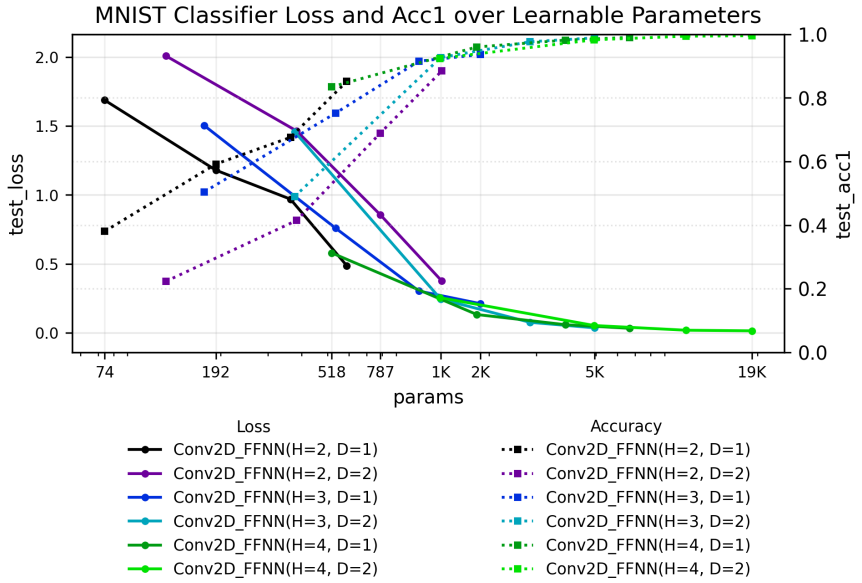


Figure 5.5: Twin-X plot detailing *loss* (categorical cross-entropy) and *acc1* (top-1 accuracy) metrics for the MNIST classifier baselines with respect to the count of learnable parameters. Line series are grouped by model heights H and downsampling block depths D, iterating over multiple parameter scaling factors P (see: Table 5.3). X-tick labels are rounded to the nearest representative integer value for readability.

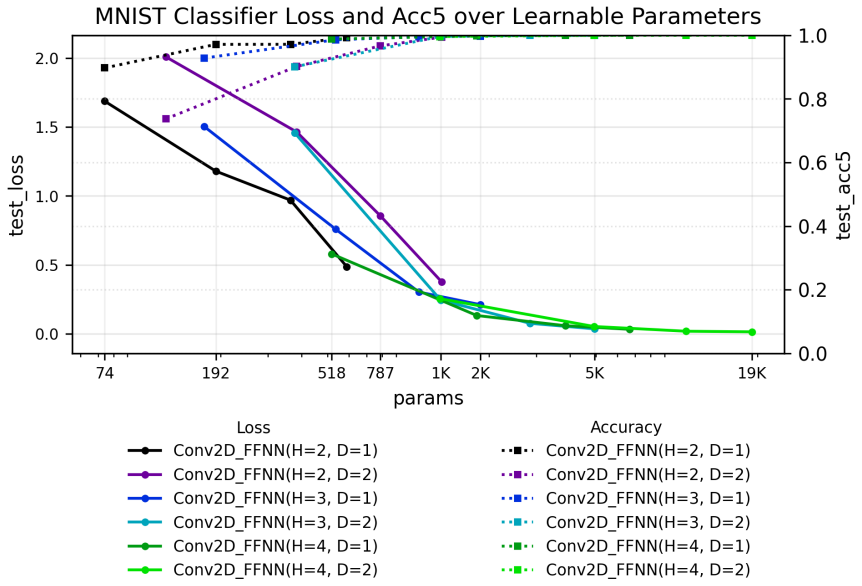


Figure 5.6: Twin-X plot detailing *loss* (categorical cross-entropy) and *acc5* (top-5 accuracy) metrics for the MNIST classifier baselines with respect to the count of learnable parameters. Line series are grouped by model heights H and downsampling block depths D, iterating over multiple parameter scaling factors P (see: Table 5.3). X-tick labels are rounded to the nearest representative integer value for readability.

of parameter efficiencies are fairly consistent. Thus, it can be observed both that reducing error requires an exponential number of parameters and that lower acc1 efficient models are also lower acc5 efficient models. These trends notably continue for later baselines on CIFAR-10 and CIFAR-100 as well.

Additionally, despite the architectural variations, performance and parameter efficiency begin to converge for all models around the 1k learnable parameters threshold. However, the $\{H = 2, D = 2\}$ model continues to have lower performance, consistent with its generally lower performance characteristics.

There is an interesting comparison between $\{H = 3, D = 1\}$ (blue line) and $\{H = 4, D = 1\}$ (green line) models. They take different approaches, with the former being slightly wider and the latter deeper, yet this results in approximately equal parameters between them at several intervals, such as around the 518 and $2k$ parameter markers. In both cases, the taller model is shown to be more parameter efficient, though in the later comparisons this could be owing to variance.

Looking ahead, the $\{P = 2, H = 3, D = 1\}$ model architecture is selected as the architecture for the experts in the MoE baselines owing to its relatively good parameter efficiency but not being so effective that reducing the error will be more difficult to observe.

5.3.2 CIFAR-10

Table 5.4: CIFAR-10 classifier baseline results averaged over three trials. Each trial is identical except for the initial RNG seeding that controls initial values, train/val partition splits (but not split sizes), and how training samples are generated. P is the parameter scaling factor controlling feature-map width. H is the model height, defined as the number of downsample blocks. D is the block depth, defined as the number of convolutional layers per block. All configurations use a dropout probability of 0.2 and no label smoothing. Loss, acc1, and acc5 metrics are rounded to 4 decimals of precision. Parameters above 100k parameters are rounded to the nearest 1k for clarity.

FFNN	P	H	D	Params	Epochs	Loss	Acc1	Acc5
Conv2D	8	2	1	2,178	478.3	1.3829	0.4980	0.9389
Conv2D	8	2	2	5,106	526.3	1.2993	0.5359	0.9458
Conv2D	8	3	1	7,010	497.3	1.2146	0.5687	0.9536
Conv2D	8	3	2	19,218	705.3	0.9746	0.6542	0.9699
Conv2D	8	4	1	25,890	552.0	0.9892	0.6519	0.9687
Conv2D	8	4	2	75,090	463.0	0.7209	0.7459	0.9839
Conv2D	16	2	1	7,802	423.3	1.1041	0.6111	0.9637
Conv2D	16	2	2	19,418	418.3	0.9592	0.6595	0.9723
Conv2D	16	3	1	26,682	498.7	0.8808	0.6911	0.9771
Conv2D	16	3	2	75,290	605.3	0.5258	0.8193	0.9915
Conv2D	16	4	1	101k	545.7	0.5873	0.7977	0.9901
Conv2D	16	4	2	298k	415.3	0.3782	0.8687	0.9959
Conv2D	32	2	1	29,418	498.7	0.8102	0.7193	0.9821
Conv2D	32	2	2	75,690	641.3	0.5290	0.8186	0.9925
Conv2D	32	3	1	104k	592.7	0.5198	0.8234	0.9928
Conv2D	32	3	2	298k	459.3	0.2421	0.9195	0.9982
Conv2D	32	4	1	401k	382.3	0.3405	0.8843	0.9968
Conv2D	32	4	2	1,185k	333.0	0.1462	0.9539	0.9992
Conv2D	64	2	1	114k	375.3	0.5762	0.8068	0.9912
Conv2D	64	2	2	299k	510.0	0.2484	0.9199	0.9980
Conv2D	64	3	1	411k	502.3	0.2704	0.9118	0.9977
Conv2D	64	3	2	1,186k	258.3	0.1155	0.9663	0.9993
Conv2D	64	4	1	1,594k	277.7	0.1630	0.9504	0.9991
Conv2D	64	4	2	4,729k	169.7	0.0935	0.9738	0.9994

Table 5.4 summarizes a sweep over convolutional feed-forward classifiers trained on CIFAR-10, with each result averaged over three random seeds. The best-performing configuration observed is $\{P = 64, H = 4, D = 2\}$, which reaches 97.38% mean top-1 test accuracy with 4,729k learnable parameters. It also yields the lowest mean test loss, 0.0935, and the

highest mean top-5 accuracy, 99.94%. As with the MNIST sweep, this configuration has the greatest number of learnable parameters, so it maximizes absolute performance rather than parameter efficiency.

Height, width, and block depth are all consistently beneficial in this sweep when the remaining architectural variables are similar. For every combination of P and D, increasing H from 2 to 3 to 4 improves mean top-1 accuracy. With $P = 16$ and $D = 2$, for example, accuracy rises from 65.95% at $H = 2$ to 81.93% at $H = 3$ and 86.87% at $H = 4$. Likewise, with $P = 64$ and $D = 2$, accuracy increases from 91.99% to 96.63% and 97.38%. Increasing width is similarly reliable: for $\{H = 3, D = 2\}$, accuracy rises from 65.42% at $P = 8$ to 81.93%, 91.95%, and 96.63% at $P = 16$, 32, and 64, respectively. Increasing D from 1 to 2 also improves accuracy and reduces loss for every tested combination of P and H.

Parameter-matched comparisons show, however, that allocating capacity exclusively to height is not always the most efficient choice for CIFAR-10. Near 75,000 parameters, $\{P = 8, H = 4, D = 2\}$ reaches 74.59% accuracy with 75,090 parameters, whereas $\{P = 16, H = 3, D = 2\}$ and $\{P = 32, H = 2, D = 2\}$ reach 81.93% and 81.86% with 75,290 and 75,690 parameters, respectively. The same pattern appears near 298k parameters: $\{P = 16, H = 4, D = 2\}$ reaches 86.87%, while $\{P = 32, H = 3, D = 2\}$ and $\{P = 64, H = 2, D = 2\}$ reach 91.95% and 91.99%. Similar comparisons hold for $D = 1$ experiments. For example, $\{P = 8, H = 4, D = 1\}$ reaches 65.19% with 25,890 parameters, whereas $\{P = 16, H = 3, D = 1\}$ reaches 69.11% with 26,682 parameters.

For $H = 4$, there are three downsampling stages because the first hidden block after the input sequence skips downsampling, as detailed in Figure 5.4. For CIFAR-10 images, which are $3 \times 32 \times 32$, the spatial progression is $* \times 32 \times 32 \to * \times 16 \times 16 \to * \times 8 \times 8 \to * \times 4 \times 4$. The consistently lower performance of the $H = 2$ configurations may therefore be partly attributable to performing less staged spatial aggregation before adaptive pooling. However, because increasing H also changes representation depth and parameter count, height could still be independent without further experimentation more strongly controlling for those.

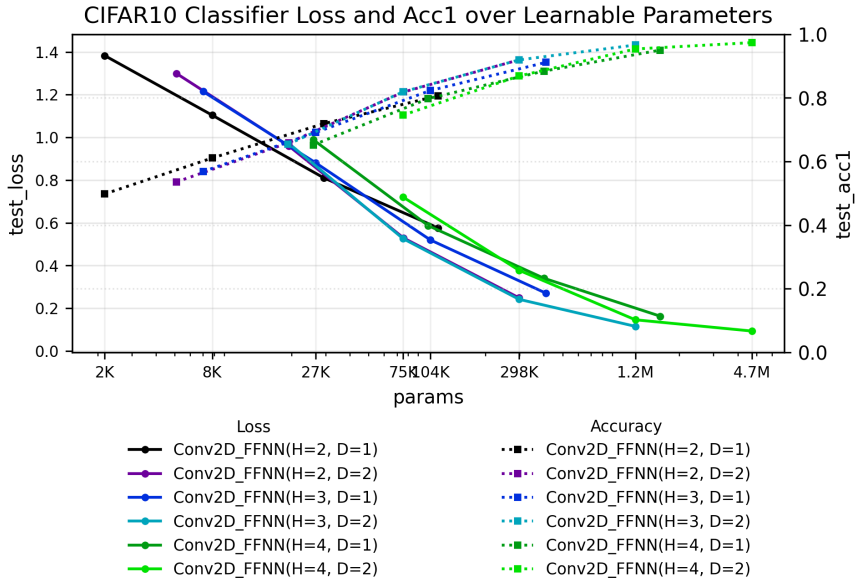


Figure 5.7: Dual-axis plot detailing *loss* (categorical cross-entropy) and *acc1* (top-1 accuracy) metrics for the CIFAR-10 classifier baselines with respect to the number of learnable parameters. Line series are grouped by model height H and downsampling block depth D, iterating over parameter scaling factors P (see Table 5.4). X-tick labels are rounded to representative integer values for readability.

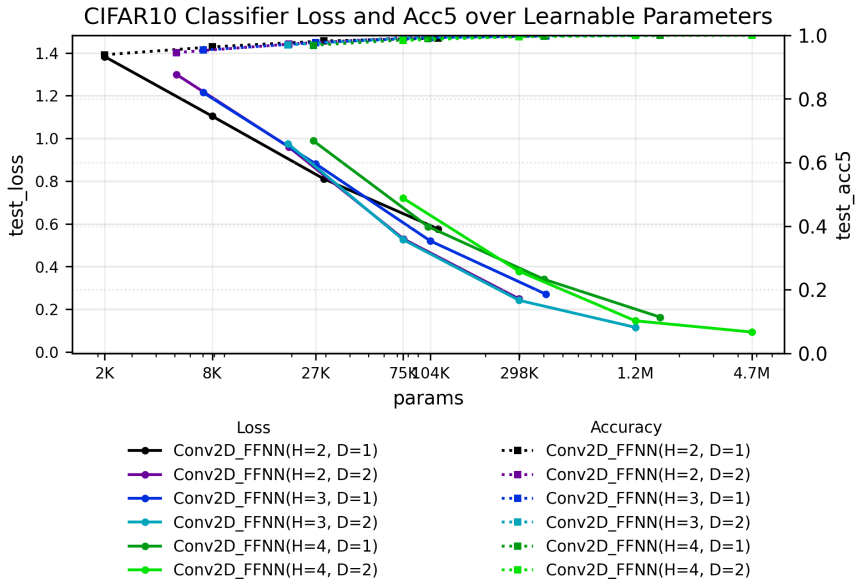


Figure 5.8: Dual-axis plot detailing *loss* (categorical cross-entropy) and *acc5* (top-5 accuracy) metrics for the CIFAR-10 classifier baselines with respect to the number of learnable parameters. Line series are grouped by model height H and downsampling block depth D, iterating over parameter scaling factors P (see Table 5.4). X-tick labels are rounded to representative integer values for readability.

Figures 5.7 and 5.8 illustrate the parameter efficiency of each architectural approach by plotting loss and accuracy against the number of learnable parameters. For a fixed accuracy, a point farther to the left is more parameter-efficient; for a fixed parameter count, a point farther upward is more accurate.

Consistent with the MNIST results, the plots show diminishing returns as parameter counts increase, except at higher parameter thresholds corresponding to increased problem-space complexity. Around 75,000 parameters, the strongest configurations reach approximately 81.9% top-1 accuracy. Around 298k parameters, the best result rises to approximately 92.0%, and around 1,186k parameters it reaches 96.63%. Increasing the parameter count from approximately 1,186k to 4,729k then improves top-1 accuracy by only about 0.75 percentage points, from 96.63% to 97.38%. Top-5 accuracy saturates even earlier: the strongest configurations near 75,000 parameters already exceed 99.1%, leaving top-1 accuracy as the more discriminating metric among the larger models.

The $\{H = 3, D = 2\}$ models are especially effective through the middle and upper portions of the parameter range. Near 75,000 parameters, $\{P = 16, H = 3, D = 2\}$ slightly outperforms the similarly sized $\{P = 32, H = 2, D = 2\}$ model and substantially outperforms $\{P = 8, H = 4, D = 2\}$. Near 1,186k parameters, $\{P = 64, H = 3, D = 2\}$ reaches 96.63%, compared with 95.39% for the nearly parameter-matched $\{P = 32, H = 4, D = 2\}$ model. Thus, although increasing height always helps at fixed P and D, medium-height models with wider feature maps appear generally more parameter-efficient.

Looking ahead, a $P = 8, H = 3, D = 4$ architectural approach is used for the expert baselines. This design choice is intended to make performance improvements easier to observe while simultaneously ensuring that a routing component can be relatively condensed. Additionally, as opposed to using $P = 16, H = 3, D = 2$, the selected architecture allows for more observable improvements made with respect to later dynamic inference experimentation where the width and/or depth of the model is doubled, giving configurations of $P = 16, H = 3, D = 4$ and $P = 8, H = 3, D = 8$.

5.3.3 CIFAR-100

Table 5.5: CIFAR-100 classifier baseline results averaged over three trials. Each trial is identical except for the initial RNG seeding that controls initial values, train/val partition splits (but not split sizes), and how training samples are generated. P is the parameter scaling factor controlling feature-map width. H is the model height, defined as the number of downsample blocks. D is the block depth, defined as the number of convolutional layers per block. All configurations use a dropout probability of 0.5 and label smoothing of 0.1. Loss, acc1, and acc5 metrics are rounded to 4 decimals of precision. Parameters above 100k parameters are rounded to the nearest 1k for clarity.

FFNN	P	H	D	Params	Epochs	Loss	Acc1	Acc5
Conv2D_Res	16	2	4	46,164	568.3	3.3425	0.2580	0.5606
Conv2D_Res	16	2	8	92,628	615.7	3.1995	0.2860	0.6021
Conv2D_Res	16	3	4	181k	666.3	2.8106	0.4014	0.7212
Conv2D_Res	16	3	8	375k	711.0	2.5224	0.4791	0.7887
Conv2D_Res	16	4	4	713k	758.3	2.3031	0.5428	0.8329
Conv2D_Res	16	4	8	1,498k	663.7	2.0664	0.6199	0.8817
Conv2D_Res	32	2	4	176k	665.7	2.7962	0.4056	0.7276
Conv2D_Res	32	2	8	361k	692.0	2.5074	0.4824	0.7991
Conv2D_Res	32	3	4	708k	773.3	2.1094	0.6144	0.8787
Conv2D_Res	32	3	8	1,484k	612.7	1.9056	0.6839	0.9168
Conv2D_Res	32	4	4	2,820k	656.0	1.6956	0.7428	0.9409
Conv2D_Res	32	4	8	5,958k	454.7	1.5704	0.8052	0.9615
Conv2D_Res	64	2	4	688k	774.7	2.1759	0.5968	0.8695
Conv2D_Res	64	2	8	1,427k	572.7	1.9571	0.6695	0.9115
Conv2D_Res	64	3	4	2,800k	638.3	1.5631	0.7993	0.9604
Conv2D_Res	64	3	8	5,901k	405.7	1.4964	0.8467	0.9731
Conv2D_Res	64	4	4	11.2M	504.3	1.2327	0.9096	0.9794
Conv2D_Res	64	4	8	23.8M	218.0	1.3963	0.8906	0.9772

Table 5.5 details a sweep over residual convolutional feed-forward classifiers trained on CIFAR-100, with each result averaged over three random seeds. The best-performing configuration observed is $\{P = 64, H = 4, D = 4\}$, which reaches 90.96% mean top-1 test accuracy with 11.2M learnable parameters. It also has the lowest mean test loss, 1.2327, and the highest mean top-5 accuracy, 97.94%. Notably, unlike in the MNIST and CIFAR-10 baselines, the largest model is not the best performing; the largest model is configured as $\{P = 64, H = 4, D = 8\}$ and uses 23.8M parameters, but reaches 89.06% top-1 accuracy.

While the accuracy difference is marginal, it does imply that the CIFAR-100 image classification problem space may have a lower ceiling than CIFAR-10 and MNIST, since doubling the number of parameters results in a plateau in performance.

Model height and feature-map width are generally observed to correspond to higher accuracy across the experiments. For each shared combination of P and D, increasing H from 2 to 3 to 4 raises mean top-1 test accuracy. With $P = 32$ and $D = 8$, for example, accuracy increases from 48.24% at $H = 2$ to 68.39% at $H = 3$ and 80.52% at $H = 4$. Likewise, with $P = 64$ and $D = 4$, accuracy rises from 59.68% to 79.93% and 90.96%. Width follows a similar pattern: for $\{H = 3, D = 8\}$, accuracy increases from 47.91% at $P = 16$ to 68.39% at $P = 32$ and 84.67% at $P = 64$.

Additional block depth is beneficial in most of the matched comparisons, although its effect appears less uniform near the largest parameter count. Increasing D from 4 to 8 improves top-1 accuracy in eight of the nine shared $\{P, H\}$ comparisons. For $\{P = 32, H = 3\}$, accuracy rises from 61.44% to 68.39%, while loss falls from 2.1094 to 1.9056. At $\{P = 64, H = 4\}$, however, accuracy decreases from 90.96% to 89.06% and loss increases from 1.2327 to 1.3963. This result may reflect optimization or regularization differences at this model size, but may also further corroborate the lower accuracy ceiling.

Parameter-matched comparisons suggest that medium-height configurations with greater width can be more efficient than taller, narrower alternatives. Near 700k parameters, $\{P = 16, H = 4, D = 4\}$ reaches 54.28% accuracy, compared with 61.44% for $\{P = 32, H = 3, D = 4\}$ and 59.68% for $\{P = 64, H = 2, D = 4\}$. A similar comparison appears around 1,500k parameters: $\{P = 16, H = 4, D = 8\}$ reaches 61.99%, $\{P = 32, H = 3, D = 8\}$ reaches 68.39%, and $\{P = 64, H = 2, D = 8\}$ reaches 66.95%. At larger parameter counts, $\{P = 64, H = 3, D = 4\}$ also reaches 79.93% with 2,800k parameters, compared with 74.28% for $\{P = 32, H = 4, D = 4\}$ with 2,820k parameters. These examples suggest that the relative allocation of width and height is important in addition to the total parameter count.

For $H = 4$, there are three downsampling stages because the first hidden block after the input sequence skips downsampling, as detailed in Figure 5.4. For CIFAR-100 images, which are $3 \times 32 \times 32$, the spatial progression is $* \times 32 \times 32 \to * \times 16 \times 16 \to * \times 8 \times 8 \to * \times 4 \times 4$. The lower performance of the $H = 2$ configurations may therefore be partly related to performing less staged spatial aggregation before adaptive pooling. Increasing H also changes representation depth and parameter count, so the independent contribution of downsampling would require more controlled comparisons.

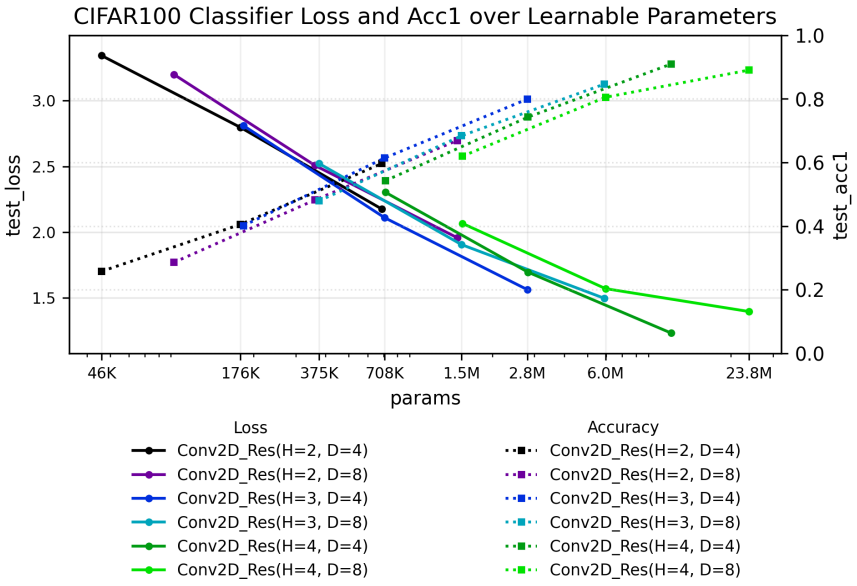


Figure 5.9: Dual-axis plot detailing *loss* (categorical cross-entropy) and *acc1* (top-1 accuracy) metrics for the CIFAR-100 classifier baselines with respect to the number of learnable parameters. Line series are grouped by model height H and downsampling block depth D, iterating over parameter scaling factors P (see Table 5.5). X-tick labels are rounded to representative integer values for readability.

Figures 5.9 and 5.10 illustrate the parameter efficiency of each architectural approach by plotting loss and accuracy against the number of learnable parameters. At a given accuracy, a point farther to the left uses fewer parameters; at a given parameter count, a point farther upward has greater accuracy.

As in the MNIST and CIFAR-10 results, the accuracy gains generally become smaller as parameter counts increase, although the architectural allocation of those parameters is

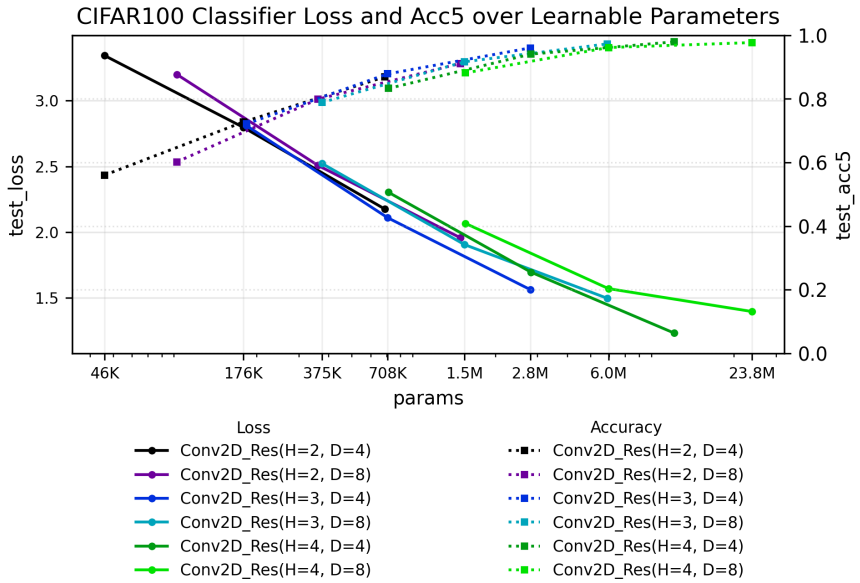


Figure 5.10: Dual-axis plot detailing *loss* (categorical cross-entropy) and *acc5* (top-5 accuracy) metrics for the CIFAR-100 classifier baselines with respect to the number of learnable parameters. Line series are grouped by model height H and downsampling block depth D, iterating over parameter scaling factors P (see Table 5.5). X-tick labels are rounded to representative integer values for readability.

relevant. The $\{P = 32, H = 3, D = 8\}$ configuration reaches 68.39% top-1 and 91.68% top-5 accuracy with approximately 1,480k parameters. Increasing to $\{P = 64, H = 3, D = 8\}$ raises these metrics to 84.67% and 97.31% with approximately 5,900k parameters. The $\{P = 64, H = 4, D = 4\}$ configuration then reaches 90.96% and 97.94% with approximately 11.2M parameters.

The comparisons between $H = 3$ and $H = 4$ also resemble those observed for CIFAR-10. Around 1,500k parameters, $\{P = 32, H = 3, D = 8\}$ exceeds the top-1 accuracy of $\{P = 16, H = 4, D = 8\}$ by 6.40 percentage points. Around 5,900k parameters, $\{P = 64, H = 3, D = 8\}$ reaches 84.67%, compared with 80.52% for the similarly sized $\{P = 32, H = 4, D = 8\}$ model. These comparisons suggest that the $H = 3$ approach can provide useful parameter efficiency when paired with sufficient width and block depth.

Looking ahead, the $\{P = 32, H = 3, D = 8\}$ architecture is used for the CIFAR-100 EM experts. In this sweep, it has 1,484k learnable parameters and reaches 68.39% top-1

and 91.68% top-5 accuracy. This provides a moderate expert size relative to the larger configurations and follows similar design decisions as CIFAR-10.

5.3.4 Summary

Baseline results detail that the generalized CNN architecture–and the TMF library by extension–is both capable and flexible as an approach for developing image classification models. It is demonstrated to effectively learn image classification across the MNIST, CIFAR-10, and CIFAR-100 problem spaces across multiple configurations, which further allow inner comparisons to consider what architectural traits are effective/ineffective. MNIST can be solved effectively with very small models, while CIFAR-10 benefits from increased channel scaling and deeper convolutional blocks. Finally, CIFAR-100 requires substantially more parameters and deeper models, but also potentially details a lower performance ceiling owing to the number of fine-detailed classes.

5.4 Autoencoder Baselines

Models are implemented using TMF, as detailed in the previous chapter. The staged encoder–decoder layout used for the autoencoder models is defined below.

$$\text{layout} = (\text{input_layout}, \text{encdec_layout}, \text{latent_layout}) \tag{5.6}$$

In the baseline configurations, this layout is parameterized by P, H, D, and N. The parameter P controls the initial feature-map width, H is the number of downsampling stages, D is the convolutional block depth, and N is the latent dimensionality. For an image-like sample shape (C, h, w), the implemented layout takes the following form.

$$\text{layout} = \big(P, (P, 2P, \ldots, 2^{H-1}P), (N,)\big) \tag{5.7}$$

This mirrors the compact configuration style used by the classifier baselines while replacing the feed-forward classification head with the encoder–latent–bridge–decoder–reconstruction pathway described in Chapter 4 and Chapter 2. For the VAE models, the final latent width N denotes the intended sampled latent dimension. In contrast, the internal variational latent layer parameterizes both the mean vector $\boldsymbol{\mu}$ and log-variance vector $\boldsymbol{\ell}$. The residual VAE variant keeps the same layout notation and adds residual connections inside the encoder and decoder blocks.

The autoencoder stages are configured through TMF layer factories. The input, encoder, and decoder stages use ReLU activations with batch normalization. The latent stage applies these operations only to non-final layers so that its final output can parameterize the variational distribution. The bridge uses ReLU activations, while the reconstruction stage applies dropout at its first layer.

```
hyperparams = {
  "input": tmf.LayerFactories(
    activation=tmf.ActivationLayerFactory.ALL("ReLU"),
    normalization=tmf.NormLayerFactory.ALL("batch")
  ),
  "encoder": tmf.LayerFactories(
    activation=tmf.ActivationLayerFactory.ALL("ReLU"),
    normalization=tmf.NormLayerFactory.ALL("batch")
  ),
  "decoder": tmf.LayerFactories(
    activation=tmf.ActivationLayerFactory.ALL("ReLU"),
    normalization=tmf.NormLayerFactory.ALL("batch")
  ),
  "latent": tmf.LayerFactories(
    activation=tmf.ActivationLayerFactory.NONFINAL("ReLU"),
    normalization=tmf.NormLayerFactory.NONFINAL("batch")
  ),
  "bridge": tmf.LayerFactories(
```

```
    activation=tmf.ActivationLayerFactory.ALL("ReLU")
  ),
  "reconstruction": tmf.LayerFactories(
    activation=tmf.ActivationLayerFactory.NONFINAL("ReLU"),
    dropout=tmf.DropoutLayerFactory.FIRST(dropout=0.2)
  )
}
```

For an input image $\mathbf{x}_b$, the encoder produces the mean and log-variance vectors shown below.

$$(\boldsymbol{\mu}_b, \boldsymbol{\ell}_b) = E_\phi(\mathbf{x}_b) \tag{5.8}$$

Both $\boldsymbol{\mu}_b$ and $\boldsymbol{\ell}_b$ lie in $\mathbb{R}^N$. During training, the latent representation is sampled as follows.

$$\mathbf{z}_b = \boldsymbol{\mu}_b + \boldsymbol{\epsilon}_b \odot \exp\left(\tfrac{1}{2}\boldsymbol{\ell}_b\right), \qquad \boldsymbol{\epsilon}_b \sim \mathcal{N}(\mathbf{0}, I) \tag{5.9}$$

The decoder maps the sampled latent representation to a reconstruction.

$$\hat{\mathbf{x}}_b = D_\psi(\mathbf{z}_b) \tag{5.10}$$

For evaluation, the mean vector is used as the latent representation.

The reconstruction term is a Gaussian negative log likelihood with a learned scalar decoder variance. Let n_{elem} denote the number of scalar entries in one input and let ℓ_{dec} denote the learned decoder log-variance. The learned decoder standard deviation is defined below.

$$\sigma_{\text{dec}} = \exp\left(\tfrac{1}{2}\ell_{\text{dec}}\right) \tag{5.11}$$

The reconstruction contribution for one sample is defined below.

$$\mathcal{L}_{\text{recon},b} = n_{\text{elem}} \log(\sigma_{\text{dec}}) + \frac{1}{2\sigma_{\text{dec}}^2} \sum_{j=1}^{n_{\text{elem}}} (x_{b,j} - \hat{x}_{b,j})^2 \tag{5.12}$$

The pixel-space mean squared error is reported separately using the following expression.

$$\text{recon_mse} = \frac{1}{Bn_{\text{elem}}} \sum_{b=1}^{B} \sum_{j=1}^{n_{\text{elem}}} (x_{b,j} - \hat{x}_{b,j})^2 \tag{5.13}$$

The latent regularization term is defined below.

$$\mathcal{L}_{\text{kld},b} = -\frac{1}{2} \sum_{j=1}^{N} \left(1 + \ell_{b,j} - \mu_{b,j}^2 - \exp(\ell_{b,j})\right) \tag{5.14}$$

The reported total loss is then defined below.

$$\mathcal{L}_{\text{VAE}} = \frac{1}{B} \sum_{b=1}^{B} (\mathcal{L}_{\text{recon},b} + \mathcal{L}_{\text{kld},b}) \tag{5.15}$$

The tables report the learned decoder standard deviation, total loss, reconstruction contribution, reconstruction MSE, and KL divergence. A negative total or reconstruction loss can occur because the learned Gaussian likelihood includes $n_{\text{elem}} \log(\sigma_{\text{dec}})$. It does not indicate a negative squared error; reconstruction MSE provides the direct pixel-error measurement. All experiments use a batch size of 256 and a 70%/30% training-validation split.

5.4.1 MNIST

The MNIST experiments use `Conv2D_VAE` with $D = 1$ and $N = 2$. It compares $H \in \{2, 3\}$ and $P \in \{2, 4, 8, 16\}$.

Increasing P lowers reconstruction MSE within both values of H. For $H = 2$, increasing P from 2 to 16 changes MSE from 0.5729 to 0.5107, while the parameter count changes from 926 to 15,346. For $H = 3$, the same width change reduces MSE from 0.5324 to 0.4369 and increases the parameter count from 1,141 to 52,409. The reconstruction improvement is consistent, but its parameter cost grows quickly.

The additional downsampling stage also lowers MSE at every matched width. The difference increases with P: the $H = 3$ result is approximately 7.1% lower at $P = 2$ and 14.5%

Table 5.6: MNIST VAE test results. P controls initial feature-map width, H is the number of downsampling stages, D is block depth, N is latent dimensionality, and Params is the learnable parameter count. Metrics are rounded to four decimal places.

Model	P	H	D	N	Params	σ_{dec}	Loss	Recon. Loss	MSE	KLD
VAE	2	2	1	2	926	0.7580	180.0	173.7	0.5729	6.4
VAE	4	2	1	2	1,690	0.7397	160.5	154.2	0.5451	6.3
VAE	8	2	1	2	4,514	0.7304	150.1	143.4	0.5303	6.7
VAE	16	2	1	2	15,346	0.7178	135.4	128.6	0.5107	6.8
VAE	2	3	1	2	1,141	0.7312	151.4	144.9	0.5324	6.6
VAE	4	3	1	2	3,713	0.7047	122.7	115.8	0.4943	6.9
VAE	8	3	1	2	13,609	0.6850	100.5	93.2	0.4666	7.3
VAE	16	3	1	2	52,409	0.6658	74.9	67.4	0.4369	7.5

lower at $P = 16$. The KL contribution changes by less than one point across most of the experiments, so the reduction in total loss is primarily associated with the reconstruction term.

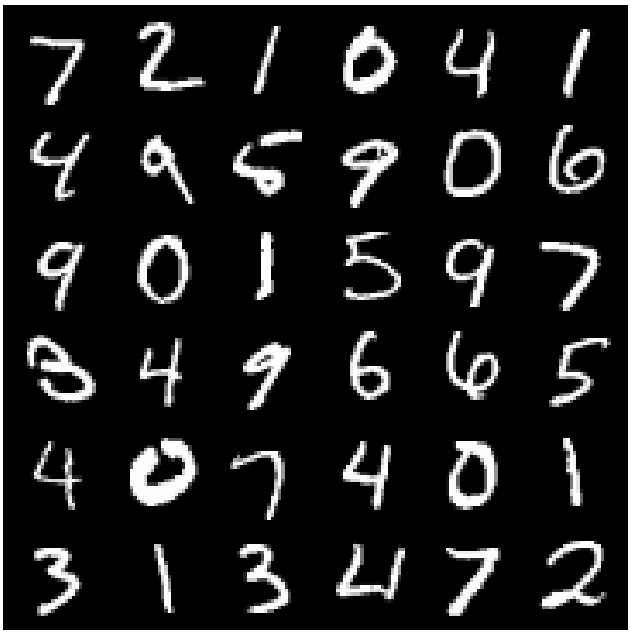
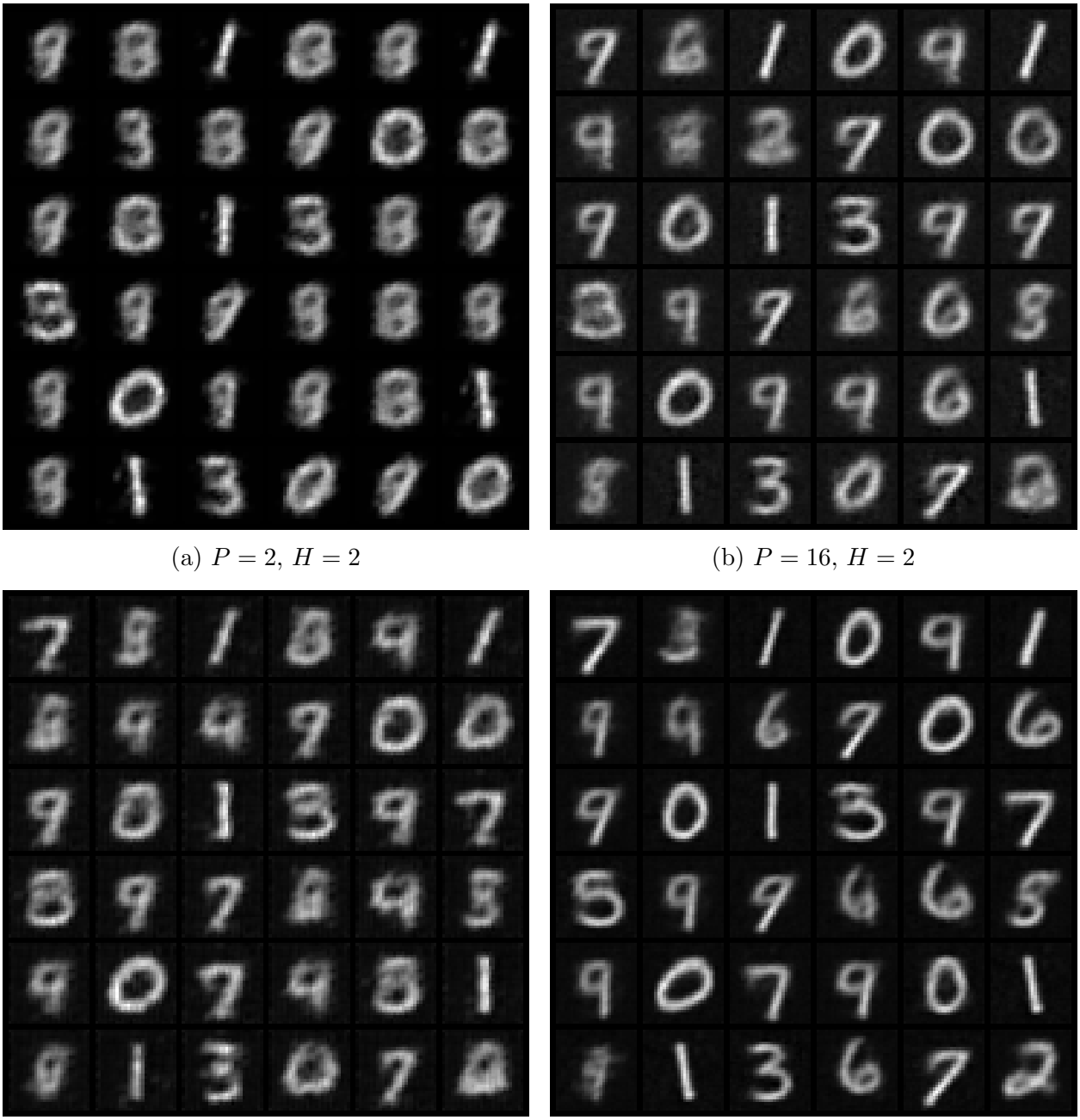


Figure 5.11: Example samples from the MNIST dataset (test partition) for comparison to the autoencoder's reconstructions.

Figures 5.13 and 5.14 show class-correlated organization in the two-dimensional latent space, together with overlap among several digits. Greater reconstruction capacity changes the geometry and class concentrations, but the reconstruction objective does not create a fully separated class representation.



(a) $P = 2$, $H = 2$

(b) $P = 16$, $H = 2$

(c) $P = 2$, $H = 3$

(d) $P = 16$, $H = 3$

Figure 5.12: MNIST reconstruction examples spanning the tested width and height configurations for fixed $(D, N) = (1, 2)$. Increasing width and adding a downsampling stage improve feature definition, although some samples remain blurred or incomplete.

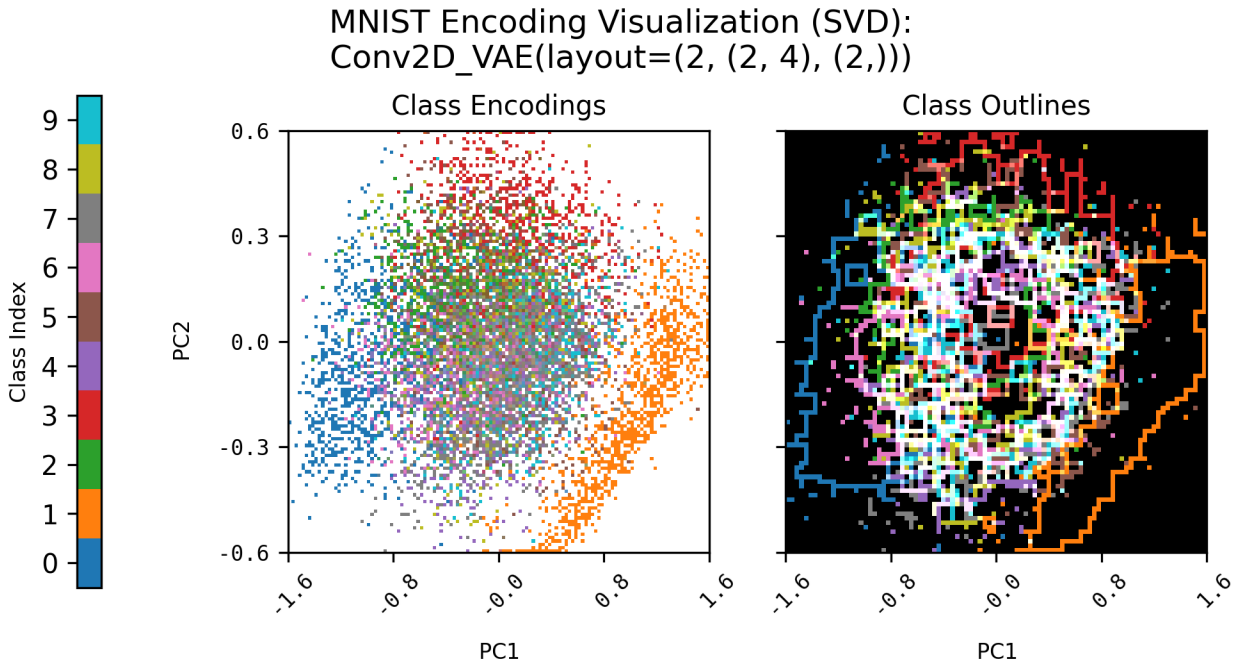


Figure 5.13: Two-dimensional SVD projection of the MNIST latent encodings for $(P, H, D, N) = (2, 2, 1, 2)$. The latent dimension is already two, so the projection does not discard latent coordinates.

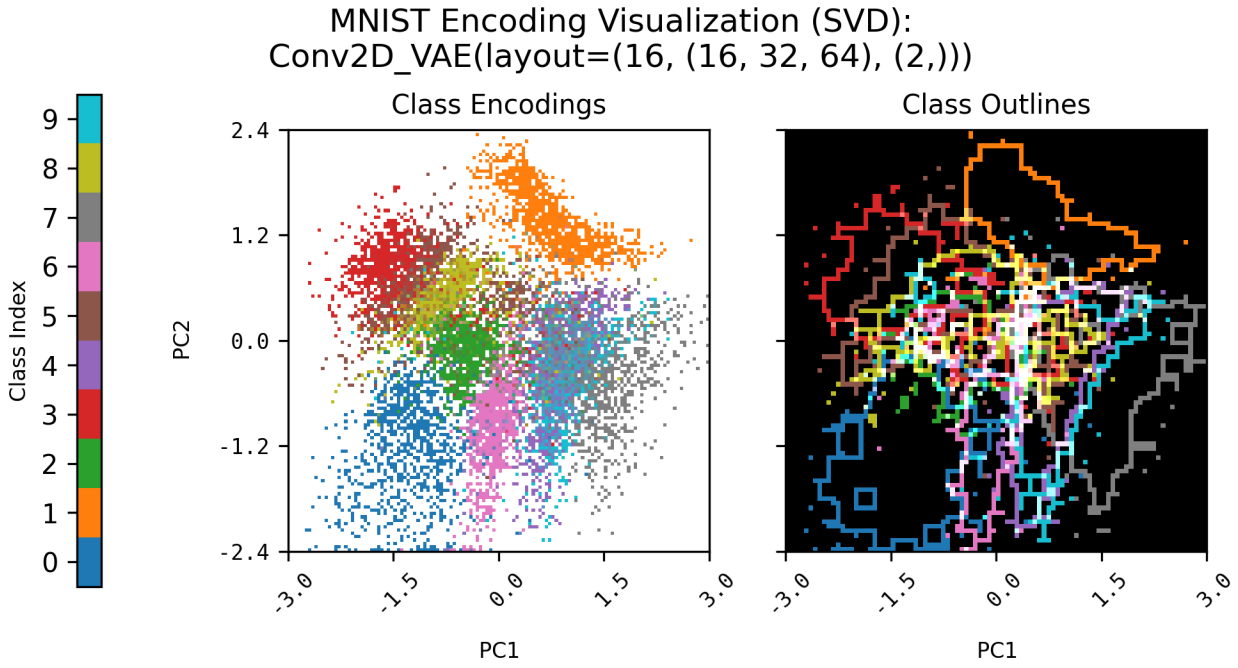


Figure 5.14: Two-dimensional SVD projection of the MNIST latent encodings for $(P, H, D, N) = (16, 3, 1, 2)$.

5.4.2 CIFAR-10

The CIFAR-10 experiments use `Conv2D_VAE` with $(H, D, N) = (3, 2, 8)$ and varies $P \in \{4, 8, 16, 32\}$.

Table 5.7: CIFAR-10 VAE test results with $(H, D, N) = (3, 2, 8)$. Metrics are rounded to four decimal places. Parameter counts above 100k are rounded to the nearest 1k for clarity.

Model	P	H	D	N	Params	σ_{dec}	Loss	Recon. Loss	MSE	KLD
VAE	4	3	2	8	8,468	0.6353	197.6	163.9	0.4093	33.7
VAE	8	3	2	8	30,452	0.6221	124.4	91.3	0.3904	33.1
VAE	16	3	2	8	117k	0.5950	11.5	-21.8	0.3626	33.3
VAE	32	3	2	8	459k	0.5826	-62.3	-96.2	0.3455	33.9

Reconstruction MSE decreases at each width increase, from 0.4093 at $P = 4$ to 0.3455 at $P = 32$. Over the same comparison, the parameter count increases from 8,468 to 459k. This is a 15.6% reduction in MSE for approximately 54 times as many parameters. The KL term stays close to 33 across the experiments, while the decoder standard deviation decreases from 0.6353 to 0.5826.

The reconstruction contribution becomes negative for $P = 16$ and $P = 32$. This follows from the learned Gaussian likelihood and the lower decoder variance; the corresponding MSE values are positive and continue to decrease.

The class-colored CIFAR-10 projections contain broad overlap at both widths. Increasing P improves the reconstruction metrics, but it does not produce a simple class partition in the first two SVD components. The encoder is optimized to retain information useful for reconstruction, and visually related classes can share many of those features.

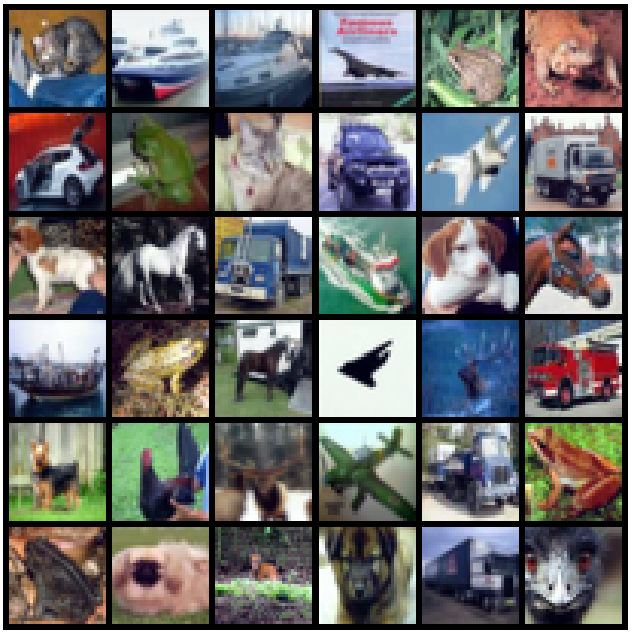


Figure 5.15: Example samples from the CIFAR-10 dataset (test partition) for comparison to the autoencoder's reconstructions.

5.4.3 CIFAR-100

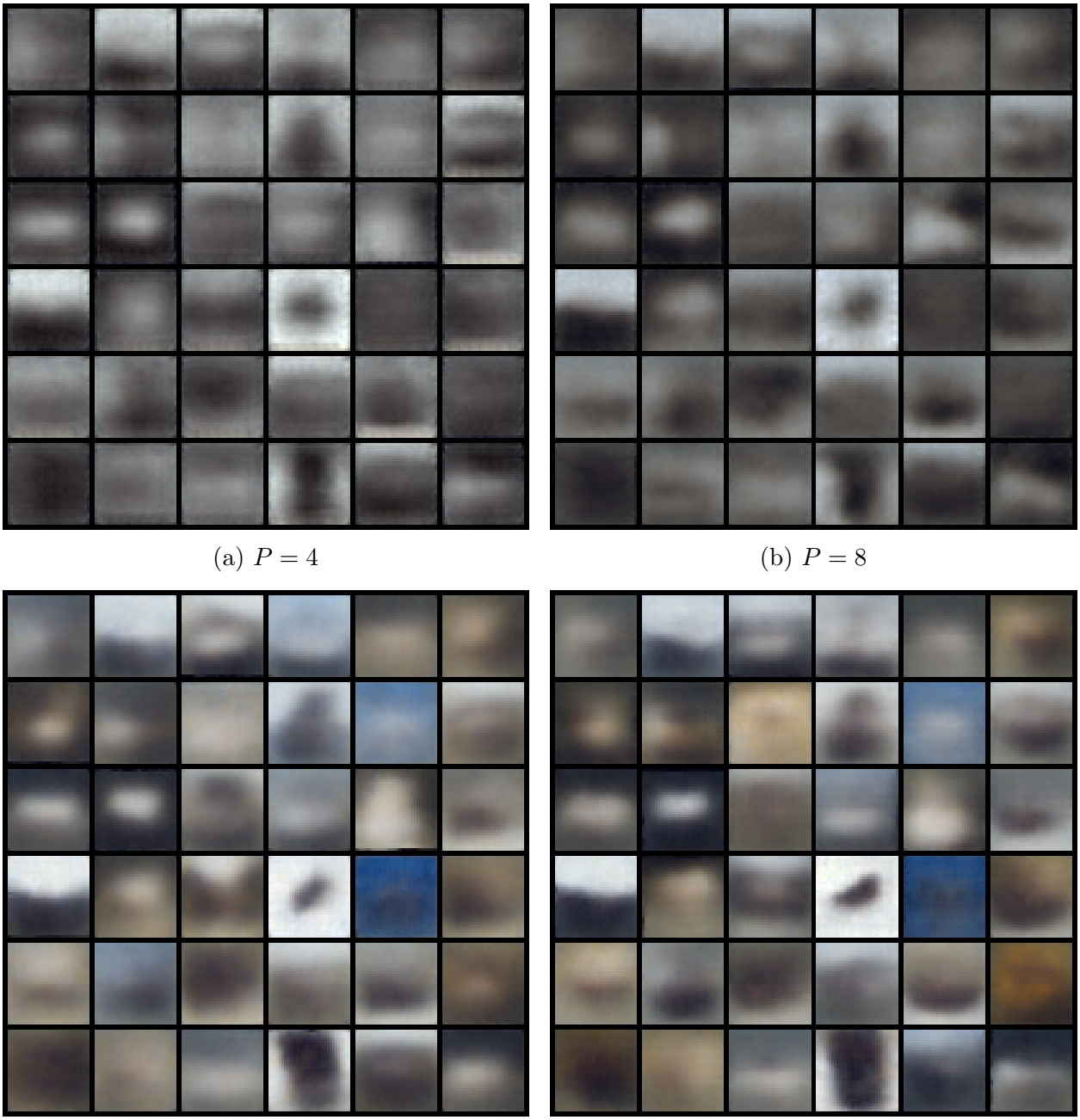
The CIFAR-100 experiments use `Conv2D_ResVAE` with $(H, D, N) = (3, 4, 32)$ and varies $P \in \{16, 32, 64\}$.

Table 5.8: CIFAR-100 residual VAE test results with $(H, D, N) = (3, 4, 32)$. Metrics are rounded to four decimal places. Parameter counts above 100k are rounded to the nearest 1k for clarity.

Model	P	H	D	N	Params	σ_{dec}	Loss	Recon. Loss	MSE	KLD
ResVAE	16	3	4	32	252k	0.3862	-1276.6	-1385.0	0.1493	108.4
ResVAE	32	3	4	32	986k	0.3736	-1376.6	-1484.0	0.1400	107.4
ResVAE	64	3	4	32	3,910k	0.3591	-1487.9	-1595.7	0.1302	107.8

Increasing P from 16 to 64 changes reconstruction MSE from 0.1493 to 0.1302, a reduction of approximately 12.8%. The parameter count increases from 252k to 3,910k, approximately 15.5 times larger. The KL term stays between 107.4 and 108.4, while the decoder standard deviation decreases from 0.3862 to 0.3591.

The CIFAR-100 MSE values are lower than the CIFAR-10 values in these tables, but the experiments use different model types, latent dimensions, block depths, and datasets. The



(a) $P = 4$ (b) $P = 8$

(c) $P = 16$ (d) $P = 32$

Figure 5.16: CIFAR-10 reconstruction examples across the tested widths for fixed $(H, D, N) = (3, 2, 8)$. Wider models preserve more color and coarse object structure, while fine textures and boundaries are still simplified.

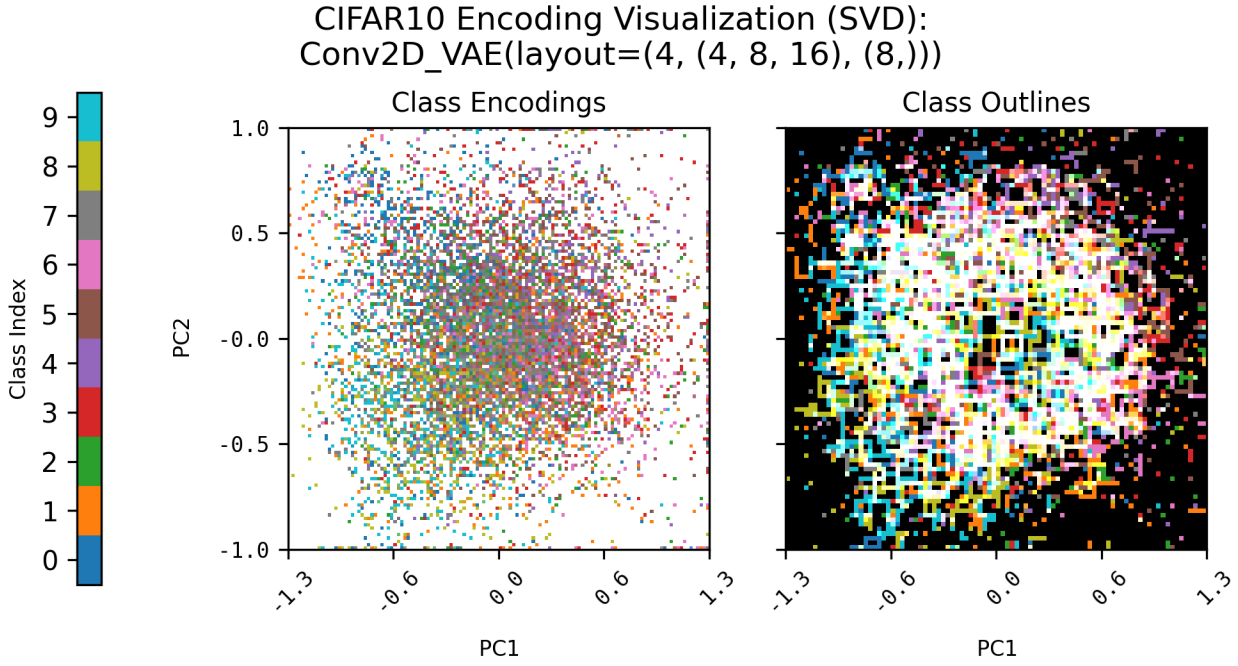


Figure 5.17: Two-dimensional SVD projection of the CIFAR-10 latent encodings for $(P, H, D, N) = (4, 3, 2, 8)$.

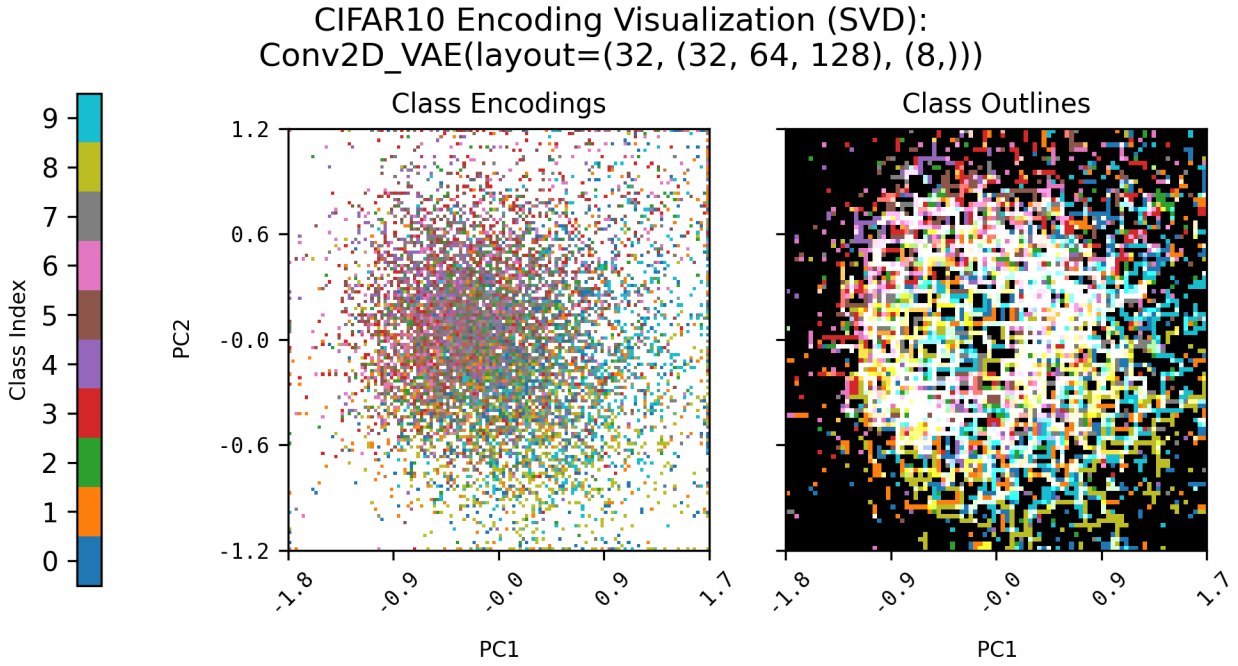


Figure 5.18: Two-dimensional SVD projection of the CIFAR-10 latent encodings for $(P, H, D, N) = (32, 3, 2, 8)$.

difference should therefore be treated as a property of the complete configurations, not as a direct measure of dataset difficulty.

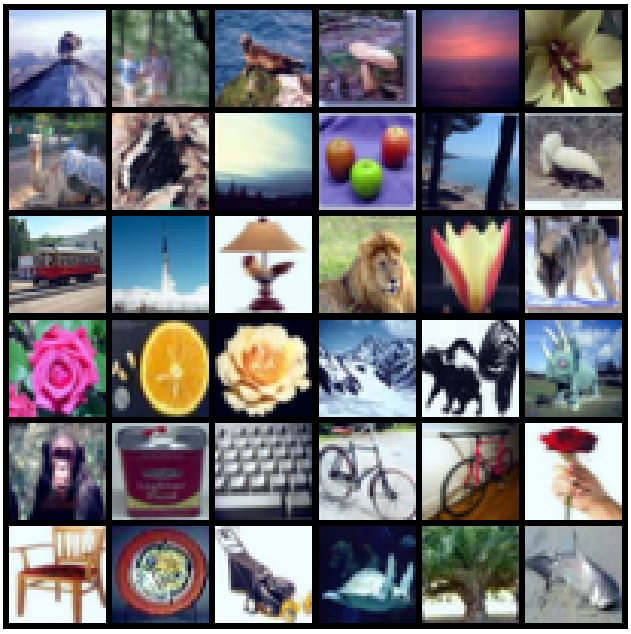
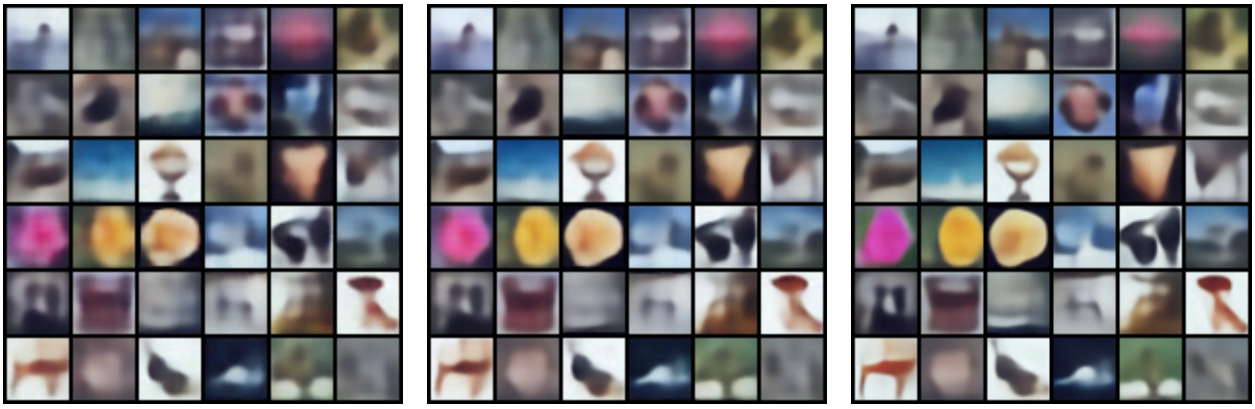


Figure 5.19: Example samples from the CIFAR-100 dataset (test partition) for comparison to the autoencoder's reconstructions.



(a) $P = 16$ (b) $P = 32$ (c) $P = 64$

Figure 5.20: CIFAR-100 reconstruction examples across the tested widths and fixed $(H, D, N) = (3, 4, 32)$. The residual models preserve coarse color and shape information, with gradual changes in detail as width increases.

The residual architecture supports low reconstruction MSE, but the first two SVD components do not organize the representation into distinct regions.

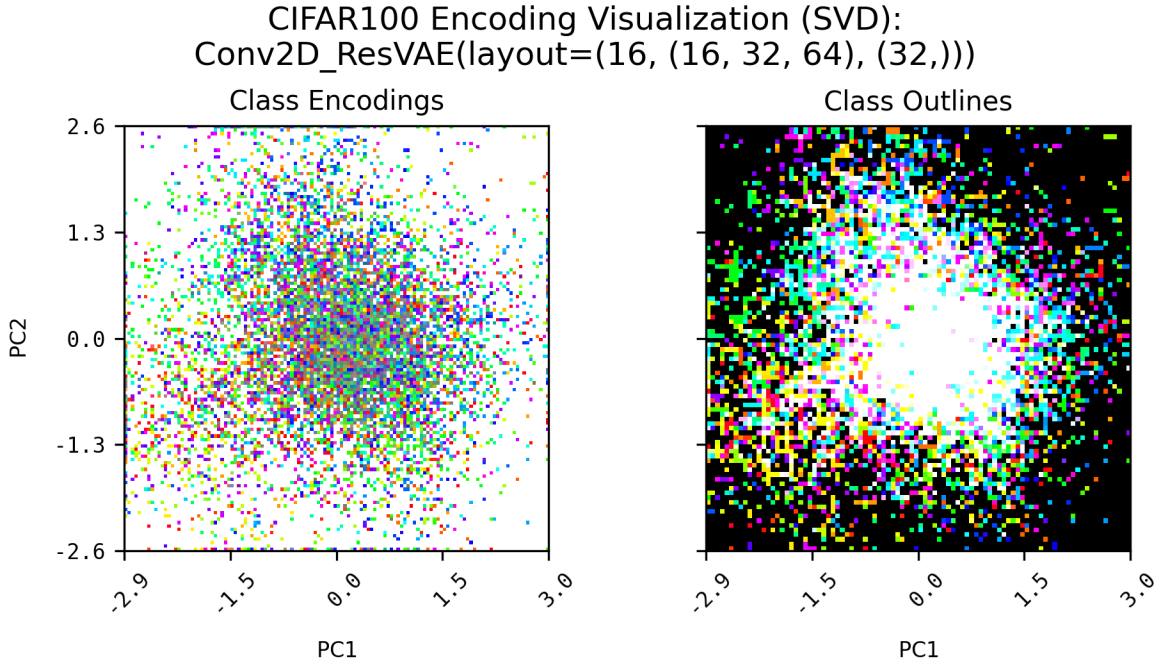


Figure 5.21: Two-dimensional SVD projection of the CIFAR-100 latent encodings for $(P, H, D, N) = (16, 3, 4, 32)$.

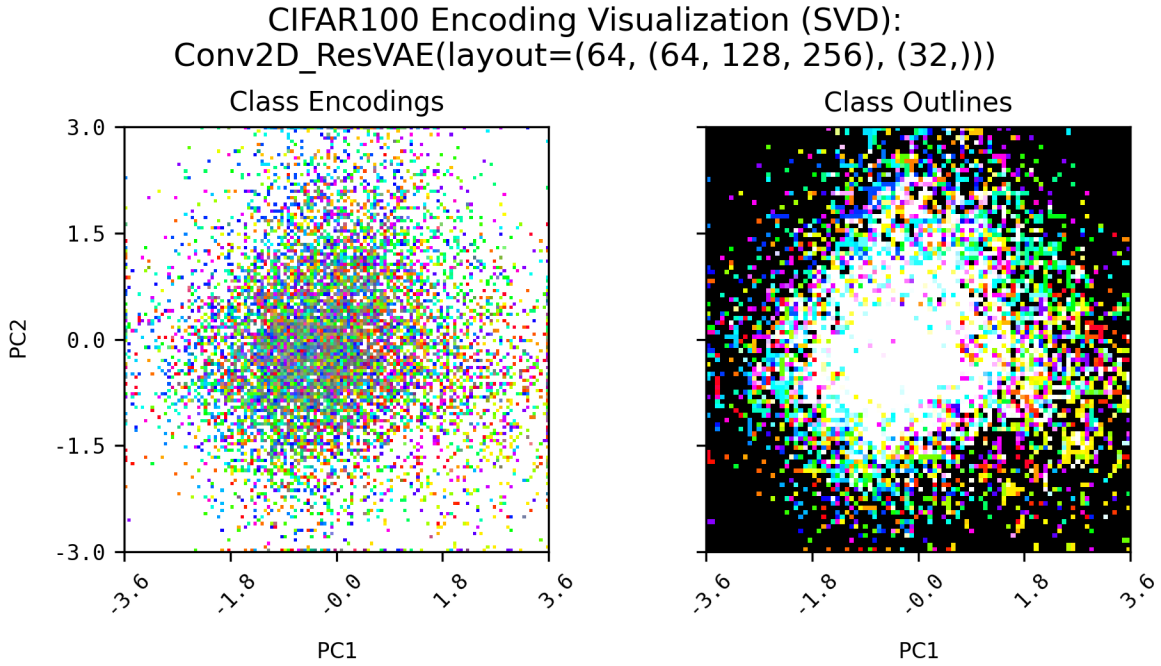


Figure 5.22: Two-dimensional SVD projection of the CIFAR-100 latent encodings for $(P, H, D, N) = (64, 3, 4, 32)$.

5.4.4 Summary

Across the three datasets, increased width lowers reconstruction MSE, and the additional MNIST downsampling stage also improves reconstruction at each matched width. The gains require much larger parameter counts: the widest CIFAR-10 model uses approximately 54 times the parameters of the narrowest model for a 15.6% MSE reduction. In comparison, the widest CIFAR-100 model uses approximately 15.5 times the parameters for a 12.8% reduction.

The learned Gaussian likelihood causes the reported reconstruction contribution to cross below zero in several CIFAR configurations. Reconstruction MSE stays positive and provides the more direct comparison of pixel error. The KL contribution changes less than the reconstruction term within each dataset, indicating that most of the total-loss change is associated with reconstruction.

MNIST shows clear class-correlated latent organization, while the CIFAR-10 and CIFAR-100 projections are more noisy. This difference is consistent with the reconstruction objective: it preserves visual information without explicitly aligning the latent representation to class boundaries. These baselines therefore provide reconstruction and latent-space references for the later routing experiments.

Across all three evaluated datasets, the results show that increasing capacity improves reconstruction quality, but reconstruction quality does not guarantee distinctive class clustering in the latent space. MNIST produces visible class-correlated latent structure, likely due to the simplicity of the input space and the fact that distinct digits have distinct features that naturally differentiate them. CIFAR-10 and CIFAR-100 produce noisier projected latent plots, indicating that unsupervised reconstruction features do not align cleanly with supervised class labels in more complex natural-image datasets. Notably, the reconstruction latent space may still require additional routing objectives or supervised signals before it becomes a reliable class-specialization space.

5.5 MoE Baselines

The MoE baseline experiments evaluate mixtures of structurally identical expert classifiers selected by one of four routing approaches: `CentroidRouter`, `CosineRouter`, `LinearRouter`, and `RBFRouter`. For each input image $\mathbf{x}_b$, the router encoder produces a latent representation $\mathbf{z}_b \in \mathbb{L}^N$, from which routing scores are computed over K experts. The experiments vary the routing approach, the number of experts, the expert width for CIFAR-10, and the routing optimization schedule. Unlike the classifier baselines, which are averaged over three trials, the MoE results reported in this section are single trials, with trends over increasing the number of experts to support the results.

The expert architectures follow the selections introduced in the classifier-baseline sections. MNIST uses `Conv2D_FFNN` experts with $\{P_e = 2, H_e = 3, D_e = 1\}$. CIFAR-10 uses `Conv2D_FFNN` experts with $\{P_e \in \{8, 16\}, H_e = 3, D_e = 4\}$. CIFAR-100 uses `Conv2D_Res` experts with $\{P_e = 32, H_e = 3, D_e = 8\}$. The associated router encoders are smaller than the experts: MNIST uses $\{P_r = 2, H_r = 2, D_r = 1, N = 2\}$; CIFAR-10 uses $\{P_r = P_e/2, H_r = 3, D_r = 2, N = 8\}$; and CIFAR-100 uses a residual encoder with $\{P_r = 16, H_r = 3, D_r = P_e/2, N = 32\}$. All experiments use $K \in \{2, 3, 4, 5\}$.

The datasets use a validation split of 0.3, consistent with the previous classifier baselines. CIFAR-10 and CIFAR-100 additionally use random cropping and horizontal flipping during training. The maximum training length is 3000 epochs with an initial warmup of 50 epochs. Batch sizes are 1024, 512, and 256 for MNIST, CIFAR-10, and CIFAR-100, respectively. Expert dropout probabilities are 0.1, 0.2, and 0.5, respectively. The MoE trainer uses no label smoothing in these experiments.

Four routing optimization modes are considered. The `t1` mode uses one selected expert throughout training, whereas `tK` retains all K experts. The `tK2t1` mode begins with K active experts and then hardens directly to one. The `tK2tk` mode uses staged reductions through intermediate values of k before reaching one. MNIST evaluates all four modes. CIFAR-

10 evaluates `tK2t1` and `tK2tk`, while CIFAR-100 evaluates `tK2t1`. For configurations with companion modes, the epoch selected by the `tK2t1` run is reused for the other schedules with patience disabled. This keeps the training length the same within each router, K, and expert-width comparison. Note that the `tK` is effectively a subset of the `tK2t1` approach, and `t1` is demonstrated by the MNIST experiments (and as described earlier in the MoE background) to be an ineffective strategy, so neither is used for CIFAR-10 and CIFAR-100. Notably, the `tK` approach is still very functional, but here is observed to largely approximate the `tK21` approach. The `tK2t1` approach is generally observed to be more effective than the `tK2tk` approach, so only the former is applied to the CIFAR-100 experiments.

For a selected expert set Θ_b^k, each expert produces logits $\hat{\mathbf{y}}_{i,b}^k$. The corresponding expert-conditional class probability is defined below.

$$p_i^k(y_b \mid \mathbf{x}_b) = \text{softmax}\left(\hat{\mathbf{y}}_{i,b}^k\right)_{y_b} \tag{5.16}$$

The optimized mixture loss is the marginal negative log likelihood over the active experts, as defined below.

$$\mathcal{L}_{\text{mNLL}}(\mathcal{B}) = -\frac{1}{B}\sum_{b=1}^{B}\log\left(\sum_{i\in\Theta_b^k} P_{i,b}^k p_i^k(y_b \mid \mathbf{x}_b)\right) \tag{5.17}$$

Here, $P_{i,b}^k$ denotes the normalized routing probability for expert i among the selected experts. This quantity is recorded as `loss`. The separately recorded `ce_loss` is the categorical cross-entropy of the top-1 selected expert and is used by the validation checkpoint and patience callbacks. These two quantities are equivalent when the evaluated route contains one expert, but they may differ when multiple experts are active.

Note that patience, where enabled, is set to 30 epochs without at least a 0.1% improvement in the monitored cross-entropy loss for the top-1 expert mixture selection. This loss is specified since most evaluations are done on a top-1 mixture, since that provides minimal inference costs and facilitates 1:1 comparisons with the classifier baselines.

The `t1_acc1` and `t1_acc5` metrics evaluate the top-1 selected expert. The `acc1` and

`acc5` metrics evaluate the active mixture. Consequently, the two metric pairs are identical for the reported `t1`, `tK2t1`, and `tK2tk` checkpoints, which finish with top-1 routing. They may differ for `tK`, where all experts are active. The tables report both the total learnable parameter count and *T1 Params*, the learnable parameters associated with the router, any routing projection, and one expert. Their relationship is defined below.

$$\text{T1Params} = \text{TotalParams} - (K-1)\text{ExpertParams} \tag{5.18}$$

The router and expert layer factories use ReLU activations. Router encoders use batch normalization, while the experts use group normalization and apply the dataset-specific dropout within hidden blocks. A simplified model construction is given below.

```
router_encoder = RouterEncoder(
    input_shape=dataset.input_shape,
    layout=router_layout,
    block_depth=Dr,
).initialize(hyperparams=router_encoder_hyperparams)

router = Router(
    model=router_encoder,
    N=N,
    K=K,
)

experts = [
    Expert(
        input_shape=router.expert_input_shape,
        layout=expert_layout,
        block_depth=De,
    ).initialize(hyperparams=expert_hyperparams)
    for _ in range(K)
]
```

```
21
22 em = ExpertsMixture(
23     experts=experts,
24     O=dataset.n_classes,
25     router=router,
26 )
```

5.5.1 MNIST

The MNIST experiments use a `Conv2D_FFNN` router encoder with $(P_r, H_r, D_r, N) = (2, 2, 1, 2)$ and `Conv2D_FFNN` experts with $(P_e, H_e, D_e) = (2, 3, 1)$. The router grid crosses the four routing approaches with $K \in \{2, 3, 4, 5\}$. Tables 5.9 and 5.10 separate the hardening schedules from the schedules that retain a constant number of active experts.

The highest observed MNIST full-mixture top-1 accuracy is produced by the Centroid router with $K = 4$ under `tK`, reaching acc1 = 0.9364 with 2,304 total parameters. Its top-1 path uses 696 parameters and reaches t1_acc1 = 0.9193. The Centroid router with $K = 5$ under `tK` has the lowest observed loss, 0.2237, and a similar full-mixture accuracy of 0.9363. The highest observed top-1 routed accuracy is 0.9230, obtained by the RBF router with $K = 5$ under `tK`. That configuration uses 703 parameters along the top-1 path and 2,847 parameters for the full mixture.

The routing schedule appears to account for more variation than the choice among the four router types in these runs. Averaged descriptively over the sixteen router and K combinations for each mode, `tK` reaches 91.83% full-mixture accuracy and 90.61% top-1 routed accuracy. The corresponding top-1 accuracies are 90.23% for `tK2t1`, 90.11% for `tK2tk`, and 84.39% for `t1`. The lower result for `t1` suggests that allowing multiple experts to participate earlier in training may be useful even when later inference selects only one expert. Because these are single runs, additional seeds would be needed to estimate how stable the schedule differences are.

Table 5.9: MNIST MoE test results for the `tK2t1` and `tK2tk` routing schedules. Each row reports one trial. Params includes the router and all experts; T1 Params reports the active learnable parameter count for top-1 inference. Loss and accuracy metrics are rounded to four decimals.

Router	Mode	K	Params	T1 Params	Epochs	Loss	T1 Acc1	T1 Acc5	Acc1	Acc5
Centroid	`tK2t1`	2	1,228	692	914	0.3412	0.9040	0.9926	0.9040	0.9926
Centroid	`tK2tk`	2	1,228	692	914	0.3455	0.9008	0.9926	0.9008	0.9926
Cosine	`tK2t1`	2	1,228	692	922	0.3420	0.8976	0.9936	0.8976	0.9936
Cosine	`tK2tk`	2	1,228	692	922	0.3609	0.8922	0.9939	0.8922	0.9939
Linear	`tK2t1`	2	1,228	692	987	0.3508	0.8938	0.9942	0.8938	0.9942
Linear	`tK2tk`	2	1,228	692	987	0.3195	0.9035	0.9944	0.9035	0.9944
RBF	`tK2t1`	2	1,230	694	912	0.3608	0.8957	0.9913	0.8957	0.9913
RBF	`tK2tk`	2	1,230	694	912	0.3361	0.9050	0.9930	0.9050	0.9930
Centroid	`tK2t1`	3	1,766	694	874	0.3038	0.9161	0.9930	0.9161	0.9930
Centroid	`tK2tk`	3	1,766	694	874	0.3235	0.9061	0.9927	0.9061	0.9927
Cosine	`tK2t1`	3	1,766	694	1417	0.3406	0.8974	0.9930	0.8974	0.9930
Cosine	`tK2tk`	3	1,766	694	1417	0.4151	0.8753	0.9906	0.8753	0.9906
Linear	`tK2t1`	3	1,766	694	837	0.3279	0.9053	0.9918	0.9053	0.9918
Linear	`tK2tk`	3	1,766	694	837	0.3396	0.8965	0.9932	0.8965	0.9932
RBF	`tK2t1`	3	1,769	697	842	0.3328	0.9002	0.9920	0.9002	0.9920
RBF	`tK2tk`	3	1,769	697	842	0.3314	0.9038	0.9927	0.9038	0.9927
Centroid	`tK2t1`	4	2,304	696	823	0.3046	0.9116	0.9936	0.9116	0.9936
Centroid	`tK2tk`	4	2,304	696	823	0.2787	0.9208	0.9943	0.9208	0.9943
Cosine	`tK2t1`	4	2,304	696	905	0.3685	0.8880	0.9919	0.8880	0.9919
Cosine	`tK2tk`	4	2,304	696	905	0.4112	0.8746	0.9907	0.8746	0.9907
Linear	`tK2t1`	4	2,304	696	1098	0.3252	0.9058	0.9922	0.9058	0.9922
Linear	`tK2tk`	4	2,304	696	1098	0.3141	0.9050	0.9948	0.9050	0.9948
RBF	`tK2t1`	4	2,308	700	1036	0.3014	0.9105	0.9916	0.9105	0.9916
RBF	`tK2tk`	4	2,308	700	1036	0.3233	0.9053	0.9925	0.9053	0.9925
Centroid	`tK2t1`	5	2,842	698	1052	0.3417	0.8965	0.9928	0.8965	0.9928
Centroid	`tK2tk`	5	2,842	698	1052	0.2955	0.9136	0.9940	0.9136	0.9940
Cosine	`tK2t1`	5	2,842	698	1123	0.3465	0.8955	0.9908	0.8955	0.9908
Cosine	`tK2tk`	5	2,842	698	1123	0.3762	0.8868	0.9883	0.8868	0.9883
Linear	`tK2t1`	5	2,842	698	1122	0.2962	0.9140	0.9923	0.9140	0.9923
Linear	`tK2tk`	5	2,842	698	1122	0.2784	0.9212	0.9927	0.9212	0.9927
RBF	`tK2t1`	5	2,847	703	1305	0.3234	0.9043	0.9909	0.9043	0.9909
RBF	`tK2tk`	5	2,847	703	1305	0.3064	0.9065	0.9907	0.9065	0.9907

Table 5.10: MNIST MoE test results for the `tK` and `t1` routing schedules. Each row reports one trial. Params includes the router and all experts; T1 Params reports the active learnable parameter count for top-1 inference. Loss and accuracy metrics are rounded to four decimals.

Router	Mode	K	Params	T1 Params	Epochs	Loss	T1 Acc1	T1 Acc5	Acc1	Acc5
Centroid	`tK`	2	1,228	692	914	0.3108	0.9027	0.9890	0.9060	0.9948
Centroid	`t1`	2	1,228	692	914	0.5125	0.8393	0.9913	0.8393	0.9913
Cosine	`tK`	2	1,228	692	922	0.3828	0.9024	0.9866	0.9050	0.9936
Cosine	`t1`	2	1,228	692	922	0.5213	0.8428	0.9899	0.8428	0.9899
Linear	`tK`	2	1,228	692	987	0.3236	0.8966	0.9868	0.9017	0.9953
Linear	`t1`	2	1,228	692	987	0.4835	0.8530	0.9918	0.8530	0.9918
RBF	`tK`	2	1,230	694	912	0.2992	0.9074	0.9873	0.9104	0.9941
RBF	`t1`	2	1,230	694	912	0.4989	0.8449	0.9915	0.8449	0.9915
Centroid	`tK`	3	1,766	694	874	0.2660	0.9148	0.9883	0.9209	0.9964
Centroid	`t1`	3	1,766	694	874	0.4778	0.8504	0.9929	0.8504	0.9929
Cosine	`tK`	3	1,766	694	1417	0.3505	0.8943	0.9850	0.9162	0.9960
Cosine	`t1`	3	1,766	694	1417	0.4512	0.8562	0.9932	0.8562	0.9932
Linear	`tK`	3	1,766	694	837	0.2851	0.9055	0.9850	0.9137	0.9959
Linear	`t1`	3	1,766	694	837	0.4969	0.8408	0.9921	0.8408	0.9921
RBF	`tK`	3	1,769	697	842	0.3556	0.8759	0.9799	0.8925	0.9950
RBF	`t1`	3	1,769	697	842	0.4973	0.8386	0.9929	0.8386	0.9929
Centroid	`tK`	4	2,304	696	823	0.2303	0.9193	0.9883	0.9364	0.9974
Centroid	`t1`	4	2,304	696	823	0.5234	0.8283	0.9918	0.8283	0.9918
Cosine	`tK`	4	2,304	696	905	0.3828	0.8958	0.9782	0.9184	0.9959
Cosine	`t1`	4	2,304	696	905	0.4968	0.8434	0.9918	0.8434	0.9918
Linear	`tK`	4	2,304	696	1098	0.2648	0.9083	0.9844	0.9185	0.9967
Linear	`t1`	4	2,304	696	1098	0.4746	0.8509	0.9929	0.8509	0.9929
RBF	`tK`	4	2,308	700	1036	0.2453	0.9190	0.9849	0.9261	0.9967
RBF	`t1`	4	2,308	700	1036	0.4796	0.8465	0.9933	0.8465	0.9933
Centroid	`tK`	5	2,842	698	1052	0.2237	0.9230	0.9893	0.9363	0.9973
Centroid	`t1`	5	2,842	698	1052	0.4424	0.8587	0.9941	0.8587	0.9941
Cosine	`tK`	5	2,842	698	1123	0.3667	0.8896	0.9819	0.9266	0.9967
Cosine	`t1`	5	2,842	698	1123	0.5265	0.8246	0.9909	0.8246	0.9909
Linear	`tK`	5	2,842	698	1122	0.2380	0.9207	0.9843	0.9310	0.9971
Linear	`t1`	5	2,842	698	1122	0.5116	0.8403	0.9919	0.8403	0.9919
RBF	`tK`	5	2,847	703	1305	0.2246	0.9230	0.9890	0.9332	0.9977
RBF	`t1`	5	2,847	703	1305	0.4841	0.8438	0.9930	0.8438	0.9930

Increasing K does not improve every router and schedule combination. There is, however, a notable increase in the descriptive mean full-mixture accuracy across all modes, from 88.72% at $K = 2$ to 89.58% at $K = 5$, together with a mean loss reduction from 0.3806 to 0.3489. The benefit is more visible for `tK`, where the full mixture can combine all expert predictions, than for the final top-1 schedules. Router ordering also varies by metric: the Centroid configuration gives the highest full-mixture accuracy and lowest loss, while the RBF configuration gives the highest top-1 accuracy and full-mixture top-5 accuracy.

Relative to the selected standalone MNIST classifier in Table 5.3, the highest top-1 MoE result increases accuracy from 75.22% for $\{P = 2, H = 3, D = 1\}$ to 92.30%. However, the learnable parameter count increases from 536 to 703. When adjusting for this efficiency, the nearest classifier baseline is $\{P = 3, H = 2, D = 2\}$ (787 params) with 69.0% acc1. However, as noted earlier, this is a less parameter-efficient approach and is not a fair comparison given the experts are designed to be somewhat parameter efficient. Therefore, a more appropriate comparison is the $\{P = 3, H = 3, D = 1\}$ (1096 params) with 91.6% acc1, which uses an identical but up-scaled architecture as the classifier expert. Notably, the best-performing MNIST MoE baseline does improve upon the $\{P = 3, H = 3, D = 1\}$ classifier baseline with fewer parameters, indicating improved scalability. However, that performance does not appear to scale in the defined training configuration and is consistent across K for most runs.

Furthermore, note the tK runs generally outperform the $tK2t1$ runs. This is likely due to the resilience in the router; by hardening each expert with respect to the observable training partition, each expert likely becomes less capable of handling even marginal outliers in the unobserved evaluation partitions, despite using a secondary unobserved validation partition to determine early-stopping and best-fit states.

Figure 5.26 details that the `tK` observations account for most of the separation between top-1 routed and full-mixture accuracy. The final top-1 schedules overlap in their `t1` and mixture metrics by construction. Loss generally decreases for several `tK` configurations as K

increases, but this behavior is not shared by every router or optimization mode.

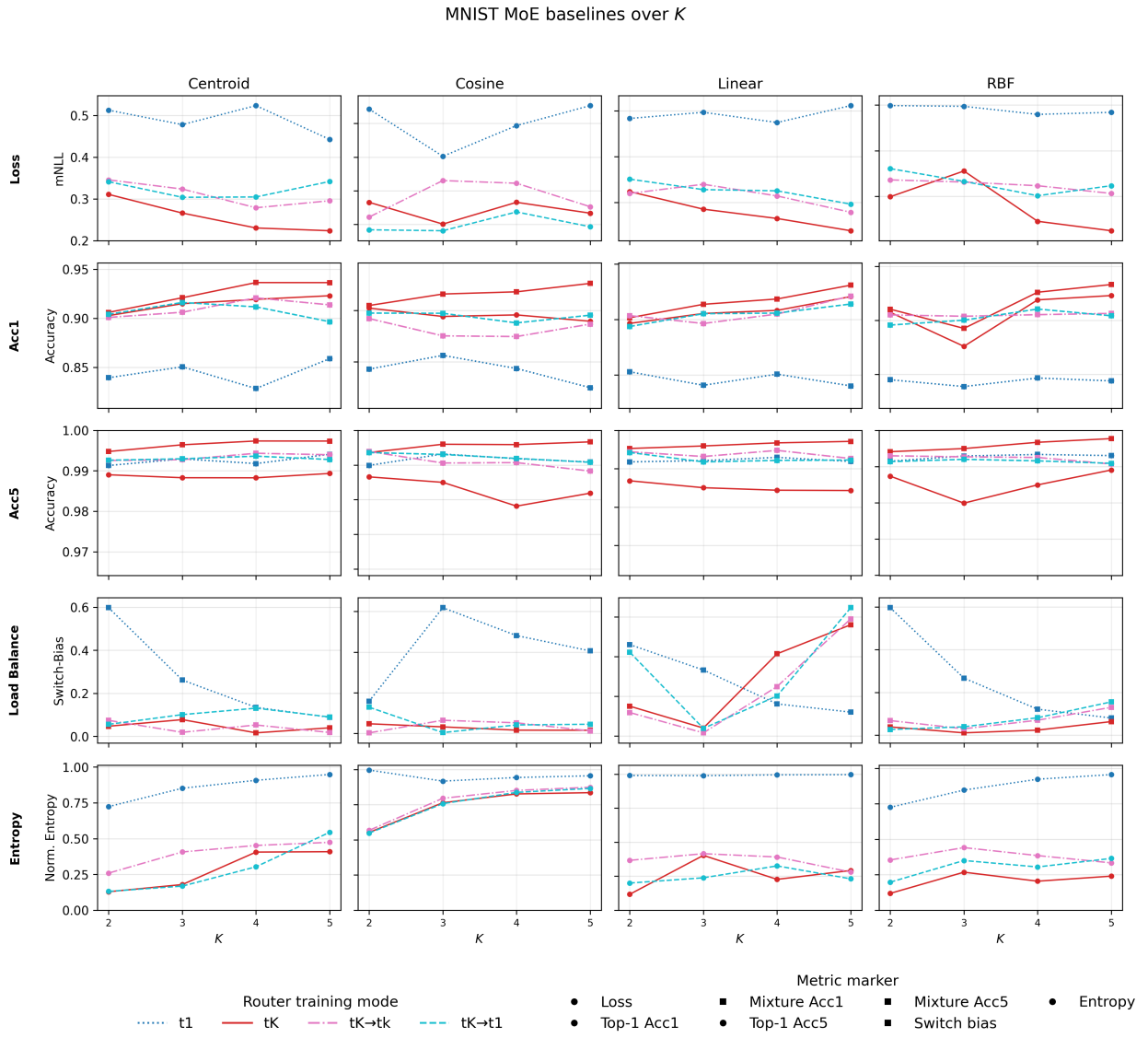


Figure 5.23: MNIST MoE test metrics over the number of experts K with respect to each routing algorithm.

5.5.2 CIFAR-10

The CIFAR-10 experiments use `Conv2D_FFNN` router encoders with $(P_r, H_r, D_r, N) = (P_e/2, 3, 2, 8)$ and experts with $(P_e, H_e, D_e) \in \{(8, 3, 4), (16, 3, 4)\}$. Both `tK2t1` and `tK2tk` finish with top-1 routing, so the reported `t1` and active-mixture accuracy metrics are identical. Tables 5.11 and 5.12 separate the two expert widths.

Table 5.11: CIFAR-10 MoE test results with $\{P_e = 8, H_e = 3, D_e = 4\}$ experts and $\{P_r = 4, H_r = 3, D_r = 2, N = 8\}$ router encoders. Each row reports one trial. Loss and accuracy metrics are rounded to four decimals. Parameter counts above 100k are rounded to the nearest 1k for clarity.

Router	Mode	K	Params	T1 Params	Epochs	Loss	T1 Acc1	T1 Acc5	Acc1	Acc5
Centroid	`tK2t1`	2	92,256	48,622	547	0.7226	0.7574	0.9763	0.7574	0.9763
Centroid	`tK2tk`	2	92,256	48,622	547	0.7589	0.7413	0.9741	0.7413	0.9741
Cosine	`tK2t1`	2	92,256	48,622	666	0.7457	0.7430	0.9795	0.7430	0.9795
Cosine	`tK2tk`	2	92,256	48,622	666	0.7439	0.7444	0.9788	0.7444	0.9788
Linear	`tK2t1`	2	92,256	48,622	630	0.7235	0.7600	0.9753	0.7600	0.9753
Linear	`tK2tk`	2	92,256	48,622	630	0.7454	0.7464	0.9755	0.7464	0.9755
RBF	`tK2t1`	2	92,258	48,624	756	0.6959	0.7660	0.9765	0.7660	0.9765
RBF	`tK2tk`	2	92,258	48,624	756	0.7187	0.7553	0.9775	0.7553	0.9775
Centroid	`tK2t1`	3	136k	48,630	596	0.7916	0.7339	0.9720	0.7339	0.9720
Centroid	`tK2tk`	3	136k	48,630	596	0.7772	0.7362	0.9741	0.7362	0.9741
Cosine	`tK2t1`	3	136k	48,630	612	0.7451	0.7455	0.9766	0.7455	0.9766
Cosine	`tK2tk`	3	136k	48,630	612	0.7661	0.7376	0.9771	0.7376	0.9771
Linear	`tK2t1`	3	136k	48,630	550	0.7633	0.7459	0.9720	0.7459	0.9720
Linear	`tK2tk`	3	136k	48,630	550	0.7901	0.7309	0.9704	0.7309	0.9704
RBF	`tK2t1`	3	136k	48,633	665	0.7446	0.7516	0.9730	0.7516	0.9730
RBF	`tK2tk`	3	136k	48,633	665	0.7612	0.7436	0.9719	0.7436	0.9719
Centroid	`tK2t1`	4	180k	48,638	488	0.7882	0.7375	0.9710	0.7375	0.9710
Centroid	`tK2tk`	4	180k	48,638	488	0.7795	0.7364	0.9701	0.7364	0.9701
Cosine	`tK2t1`	4	180k	48,638	525	0.8077	0.7280	0.9717	0.7280	0.9717
Cosine	`tK2tk`	4	180k	48,638	525	0.7901	0.7307	0.9732	0.7307	0.9732
Linear	`tK2t1`	4	180k	48,638	824	0.7387	0.7539	0.9714	0.7539	0.9714
Linear	`tK2tk`	4	180k	48,638	824	0.6850	0.7673	0.9745	0.7673	0.9745
RBF	`tK2t1`	4	180k	48,642	786	0.6868	0.7696	0.9756	0.7696	0.9756
RBF	`tK2tk`	4	180k	48,642	786	0.6942	0.7659	0.9759	0.7659	0.9759
Centroid	`tK2t1`	5	223k	48,646	494	0.8037	0.7341	0.9654	0.7341	0.9654
Centroid	`tK2tk`	5	223k	48,646	494	0.8002	0.7291	0.9714	0.7291	0.9714
Cosine	`tK2t1`	5	223k	48,646	561	0.8007	0.7225	0.9724	0.7225	0.9724
Cosine	`tK2tk`	5	223k	48,646	561	0.8076	0.7221	0.9716	0.7221	0.9716
Linear	`tK2t1`	5	223k	48,646	486	0.7908	0.7392	0.9651	0.7392	0.9651
Linear	`tK2tk`	5	223k	48,646	486	0.8342	0.7212	0.9654	0.7212	0.9654
RBF	`tK2t1`	5	223k	48,651	551	0.8021	0.7309	0.9691	0.7309	0.9691
RBF	`tK2tk`	5	223k	48,651	551	0.8157	0.7250	0.9664	0.7250	0.9664

Table 5.12: CIFAR-10 MoE test results with $\{P_e = 16, H_e = 3, D_e = 4\}$ experts and $\{P_r = 8, H_r = 3, D_r = 2, N = 8\}$ router encoders. Each row reports one trial. Loss and accuracy metrics are rounded to four decimals. Parameter counts above 100k are rounded to the nearest 1k for clarity.

Router	Mode	K	Params	T1 Params	Epochs	Loss	T1 Acc1	T1 Acc5	Acc1	Acc5
Centroid	`tK2t1`	2	364k	192k	485	0.3872	0.8709	0.9912	0.8709	0.9912
Centroid	`tK2tk`	2	364k	192k	485	0.4068	0.8613	0.9942	0.8613	0.9942
Cosine	`tK2t1`	2	364k	192k	318	0.4588	0.8444	0.9922	0.8444	0.9922
Cosine	`tK2tk`	2	364k	192k	318	0.4629	0.8418	0.9924	0.8418	0.9924
Linear	`tK2t1`	2	364k	192k	417	0.4289	0.8569	0.9921	0.8569	0.9921
Linear	`tK2tk`	2	364k	192k	417	0.4011	0.8648	0.9932	0.8648	0.9932
RBF	`tK2t1`	2	364k	192k	375	0.4219	0.8618	0.9895	0.8618	0.9895
RBF	`tK2tk`	2	364k	192k	375	0.4273	0.8553	0.9917	0.8553	0.9917
Centroid	`tK2t1`	3	537k	192k	455	0.4021	0.8692	0.9901	0.8692	0.9901
Centroid	`tK2tk`	3	537k	192k	455	0.4124	0.8618	0.9919	0.8618	0.9919
Cosine	`tK2t1`	3	537k	192k	544	0.3866	0.8704	0.9943	0.8704	0.9943
Cosine	`tK2tk`	3	537k	192k	544	0.3974	0.8672	0.9938	0.8672	0.9938
Linear	`tK2t1`	3	537k	192k	574	0.3780	0.8800	0.9898	0.8800	0.9898
Linear	`tK2tk`	3	537k	192k	574	0.3774	0.8725	0.9929	0.8725	0.9929
RBF	`tK2t1`	3	537k	192k	435	0.4172	0.8637	0.9899	0.8637	0.9899
RBF	`tK2tk`	3	537k	192k	435	0.4144	0.8618	0.9906	0.8618	0.9906
Centroid	`tK2t1`	4	709k	192k	591	0.3785	0.8760	0.9914	0.8760	0.9914
Centroid	`tK2tk`	4	709k	192k	591	0.3802	0.8744	0.9922	0.8744	0.9922
Cosine	`tK2t1`	4	709k	192k	431	0.4350	0.8560	0.9915	0.8560	0.9915
Cosine	`tK2tk`	4	709k	192k	431	0.4266	0.8562	0.9921	0.8562	0.9921
Linear	`tK2t1`	4	709k	192k	395	0.4325	0.8650	0.9867	0.8650	0.9867
Linear	`tK2tk`	4	709k	192k	395	0.4356	0.8549	0.9896	0.8549	0.9896
RBF	`tK2t1`	4	709k	192k	368	0.4557	0.8557	0.9858	0.8557	0.9858
RBF	`tK2tk`	4	709k	192k	368	0.4550	0.8503	0.9884	0.8503	0.9884
Centroid	`tK2t1`	5	882k	192k	611	0.3795	0.8814	0.9897	0.8814	0.9897
Centroid	`tK2tk`	5	882k	192k	611	0.3749	0.8763	0.9919	0.8763	0.9919
Cosine	`tK2t1`	5	882k	192k	539	0.4102	0.8641	0.9922	0.8641	0.9922
Cosine	`tK2tk`	5	882k	192k	539	0.3934	0.8673	0.9932	0.8673	0.9932
Linear	`tK2t1`	5	882k	192k	577	0.4059	0.8737	0.9884	0.8737	0.9884
Linear	`tK2tk`	5	882k	192k	577	0.4029	0.8671	0.9905	0.8671	0.9905
RBF	`tK2t1`	5	882k	192k	589	0.4299	0.8625	0.9882	0.8625	0.9882
RBF	`tK2tk`	5	882k	192k	589	0.3963	0.8698	0.9910	0.8698	0.9910

Expert width is associated with the largest accuracy difference in the CIFAR-10 results. Averaged over router type, K, and optimization mode, the $P_e = 8$ configurations reach 74.23% top-1 accuracy, while the $P_e = 16$ configurations reach 86.42%. However, the wider experts increase the top-1 active parameter count from approximately 48.6 thousand to 191.7 thousand, so a substantial increase in inference cost accompanies the accuracy increase. This exponential parameter cost to reduce accuracy is consistent with earlier classifier baselines.

The Centroid router obtains the highest observed top-1 accuracy with $P_e = 16$, $K = 5$, and `tK2t1`. It reaches 88.14% accuracy with 882k total parameters and 192k top-1 parameters. The same router, width, and K under `tK2tk` gives the lowest observed loss, 0.3749, with 87.63% accuracy. The highest observed top-5 accuracy is 99.43%, obtained by the Cosine router with $P_e = 16$, $K = 3$, and `tK2t1`. These differences indicate that the configuration selected by top-1 accuracy is not necessarily the one selected by loss or top-5 accuracy. However, the margins here are relatively close and likely subject to variance.

The two hardening schedules have similar descriptive averages. Across all CIFAR-10 rows, `tK2t1` reaches 80.53% top-1 accuracy and `tK2tk` reaches 80.11%. Among the $P_e = 16$ rows, the averages are 86.57% and 86.27%, respectively. Interestingly, however, the `tK2t1` routing optimization approach consistently results in significantly lower entropy, indicating that while accuracy does not significantly increase, there is more structure to how signals are routed across experts.

One potential issue with experimentation is that K is not associated with scalability in the results. For $P_e = 16$, the mean accuracy across routers and schedules is 85.71% at $K = 2$, 86.83% at $K = 3$, 86.11% at $K = 4$, and 87.03% at $K = 5$. For $P_e = 8$, the corresponding values are 75.17%, 74.07%, 74.87%, and 72.80%. Router averages are also close when both expert widths are combined: Linear, RBF, and Centroid routing reach approximately 80.62%, 80.56%, and 80.48%, while Cosine routing reaches 79.63%.

Figure 5.24 reflects the separation between the two expert widths more than a general change with K.

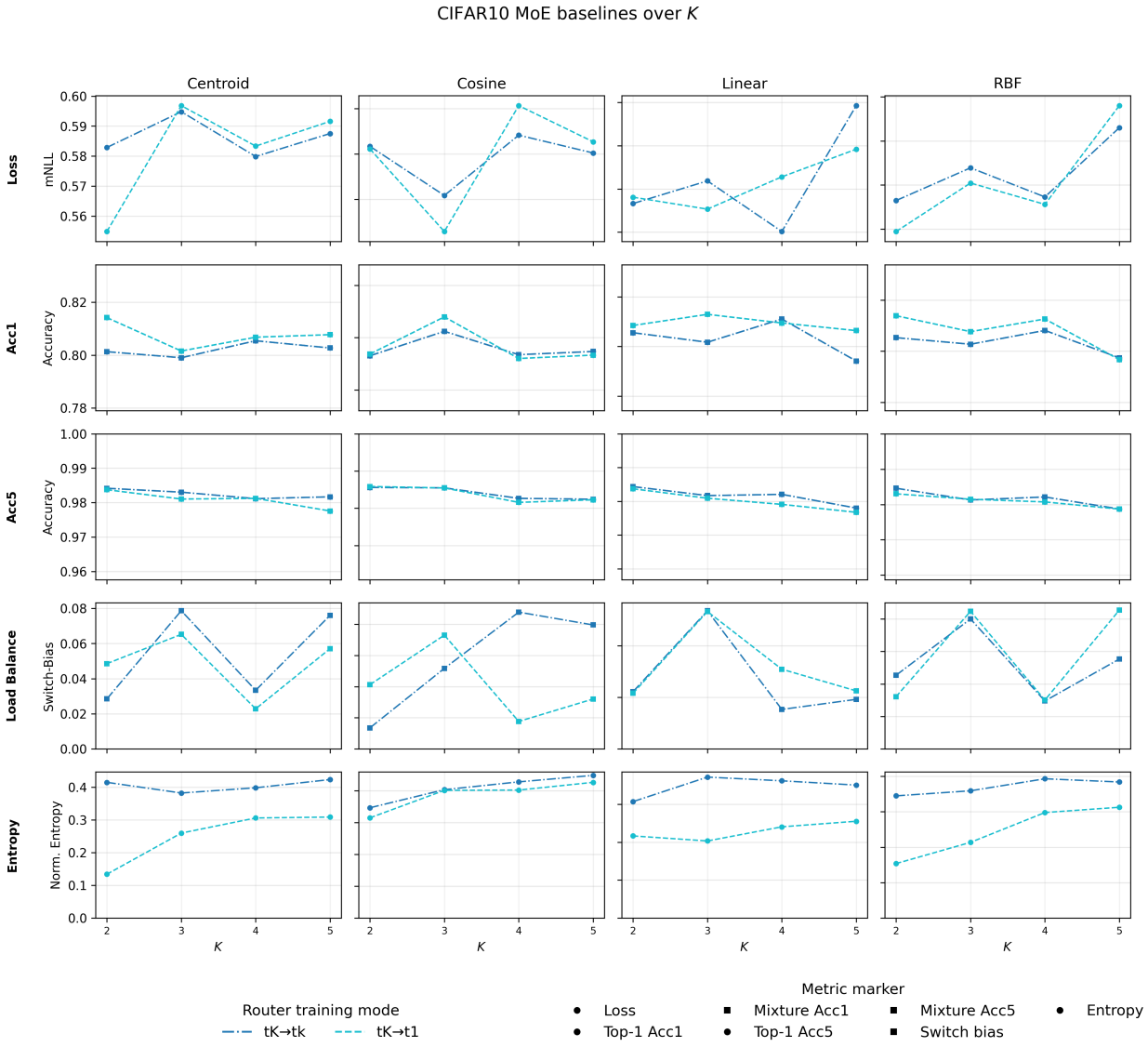


Figure 5.24: CIFAR10 MoE test metrics over the number of experts K with respect to each routing algorithm.

5.5.3 CIFAR-100

The CIFAR-100 experiments use a `Conv2D_Res` router encoder configured with $(P_r, H_r, D_r, N) = (16, 3, 4, 32)$ and `Conv2D_Res` experts configured with $(P_e, H_e, D_e) = (32, 3, 8)$. This is the $\{P = 32, H = 3, D = 8\}$ architecture selected for the CIFAR-100 EM experts. The residual blocks use a skip length of 2. Only the `tK2t1` schedule is evaluated, and the reported checkpoints use top-1 routing.

Table 5.13: CIFAR-100 MoE test results with $\{P_e = 32, H_e = 3, D_e = 8\}$ residual experts and a $\{P_r = 16, H_r = 3, D_r = 4, N = 32\}$ residual router encoder. All rows use `tK2t1` and report one trial. Loss and accuracy metrics are rounded to four decimals. Parameter counts above 100k are rounded to the nearest 1k for clarity.

Router	K	Params	T1 Params	Epochs	Loss	T1 Acc1	T1 Acc5	Acc1	Acc5
Centroid	2	3,145k	1,661k	406	0.9890	0.7310	0.9255	0.7310	0.9255
Cosine	2	3,145k	1,661k	422	1.0578	0.7095	0.9178	0.7095	0.9178
Linear	2	3,145k	1,661k	481	0.9623	0.7402	0.9296	0.7402	0.9296
RBF	2	3,145k	1,661k	511	0.9609	0.7378	0.9306	0.7378	0.9306
Centroid	3	4,629k	1,661k	349	1.0854	0.7171	0.9117	0.7171	0.9117
Cosine	3	4,629k	1,661k	348	1.1382	0.6998	0.9047	0.6998	0.9047
Linear	3	4,629k	1,661k	452	0.9922	0.7366	0.9221	0.7366	0.9221
RBF	3	4,629k	1,661k	430	0.9924	0.7324	0.9258	0.7324	0.9258
Centroid	4	6,113k	1,661k	413	1.0329	0.7389	0.9147	0.7389	0.9147
Cosine	4	6,113k	1,661k	426	1.1190	0.7076	0.9063	0.7076	0.9063
Linear	4	6,113k	1,661k	398	1.0036	0.7478	0.9187	0.7478	0.9187
RBF	4	6,113k	1,661k	406	0.9896	0.7422	0.9242	0.7422	0.9242
Centroid	5	7,597k	1,661k	473	0.9928	0.7503	0.9165	0.7503	0.9165
Cosine	5	7,597k	1,661k	369	1.1760	0.6977	0.8980	0.6977	0.8980
Linear	5	7,597k	1,661k	502	0.9936	0.7490	0.9177	0.7490	0.9177
RBF	5	7,597k	1,661k	399	1.0232	0.7379	0.9172	0.7379	0.9172

The highest observed CIFAR-100 top-1 accuracy is produced by the Centroid router with $K = 5$, reaching 75.03%. The Linear router gives similar values at $K = 5$ and $K = 4$, reaching 74.90% and 74.78%, respectively. The lowest observed loss and highest top-5 accuracy occur for the RBF router with $K = 2$, which reaches a loss of 0.9609, top-1 accuracy of 73.78%, and top-5 accuracy of 93.06%.

The number of experts has a limited and non-monotonic association with accuracy in this single trial. Averaged across routers, top-1 accuracy is 72.97% at $K = 2$, 72.15% at $K = 3$,

73.41% at $K = 4$, and 73.37% at $K = 5$. The mean loss is lowest at $K = 2$. Router-level averages also do not identify one approach across all metrics. Linear routing has the highest mean top-1 accuracy, 74.34%, and the lowest mean loss, while RBF routing has the highest mean top-5 accuracy. Centroid routing nevertheless provides the highest individual top-1 result.

The top-1 active parameter count approaches 1,660k as K increases, because inference uses the router and one expert. The total parameter count grows from approximately 3,140k at $K = 2$ to 7,600k at $K = 5$. The routing projection and router-specific parameters cause the small change in T1 Params across K.

The standalone $\{P = 32, H = 3, D = 8\}$ classifier in Table 5.5 reaches 68.39% top-1 accuracy with 1,484k parameters. The highest observed MoE top-1 result reaches 75.03% with 1,661k active parameters, an increase of approximately 6.64 percentage points in accuracy and 11.9% in active parameters. This comparison should again be interpreted cautiously because the classifier result is averaged over three trials and the MoE result is a single trial.s

Figure 5.25 details relatively small differences among several Linear, RBF, and Centroid observations, with the Cosine results generally lower in top-1 accuracy for this trial. The loss and top-5 plots place several $K = 2$ observations above configurations with more experts, which is consistent with the table and does not suggest that increasing K alone improves the evaluated metrics.

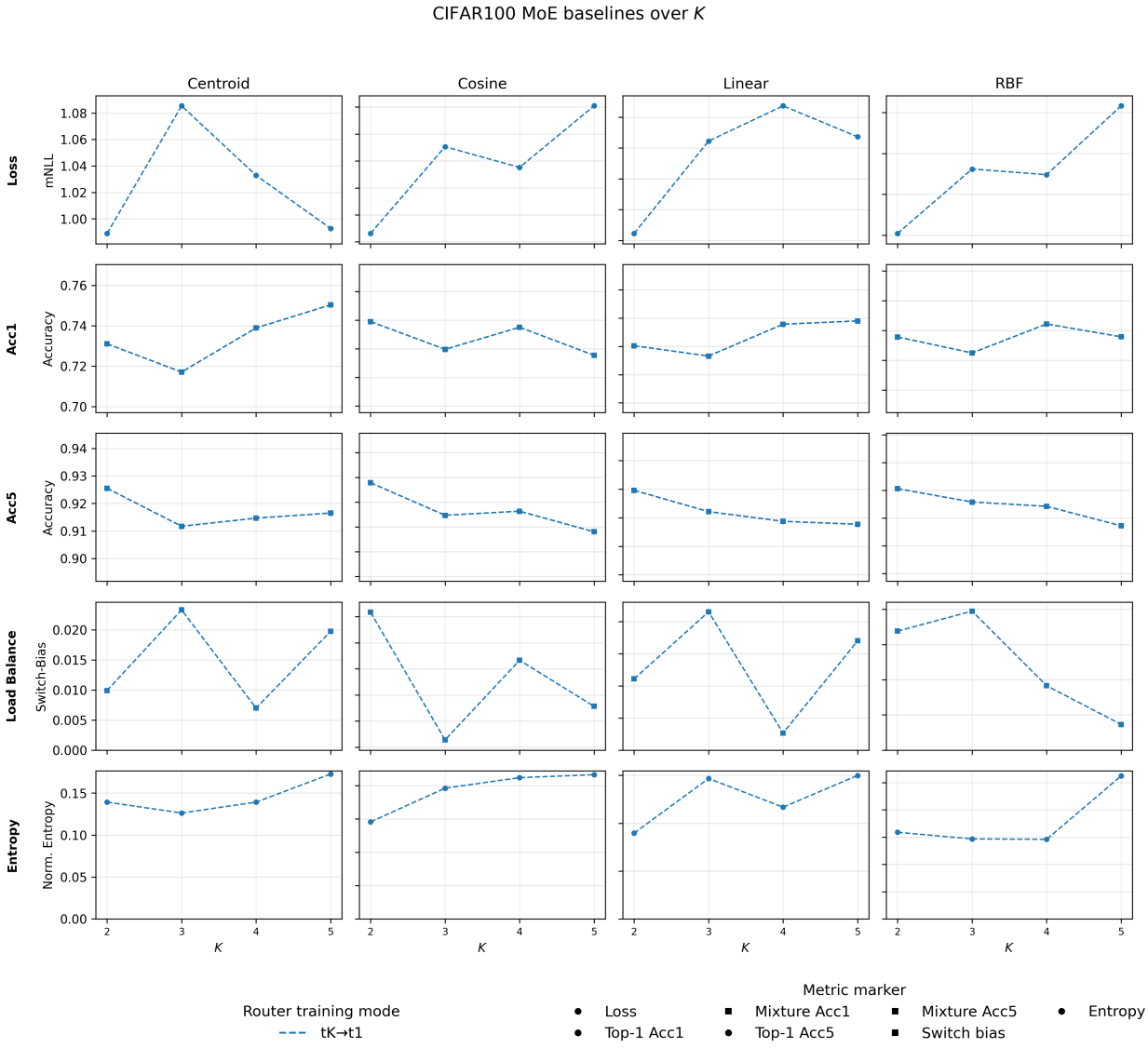


Figure 5.25: CIFAR100 MoE test metrics over the number of experts K with respect to each routing algorithm.

5.5.4 Summary

The margins by which the best-performing routers are decided are small and likely due to natural variance owing to initial weights and the training process. However, routing optimization approaches are clearer in terms of preferences. The `t1` runs have lower accuracy than the `tK`, `tK2t1`, and `tK2tk` runs on average. The CIFAR-10 comparison between direct and staged hardening gives smaller differences, with neither schedule improving every router, width, and K combination. CIFAR-100 includes only direct hardening, so no schedule comparison is available there.

Accuracy does not increase monotonically with K in the CIFAR-10 or CIFAR-100 results. The final top-1 schedules retain the lower active inference cost, although they do not receive an ensemble contribution at evaluation. The latter behavior is consistent with the properties of an MoE model outlined earlier in the background.

Given general inconsistencies across all routing algorithms, there is likely a limiting factor in the training process. The following section will attempt to explore extensions to the MoE baselines to verify potential issues. Looking ahead to the results, the issue is likely in the training length; despite the 30 epochs of patience, resulting in approximately 545 epochs (across all CIFAR-10 MoE baseline experiments), achieving scalable performance likely requires extending training to 2,000-to-3,000 epochs.

The selected MoE configurations improve on the corresponding standalone classifier results for MNIST and CIFAR-100 in these observations, even after accounting for the router in the active parameter count. Though only single experiments for each configuration are run, observations across multiple sequences can be used to establish confidence. Later experiments can use these results as references when evaluating routing regularization, transfer learning, expert specialization, and dynamic inference.

5.6 MoE Scalability Experiments

The previous MoE baselines do not show a consistent improvement in top-1 routed accuracy as the number of experts increases. One possible limitation is that the patience-based stopping criterion may end training before the router and experts have adapted to one another. A second possibility is that the router encoder is too small relative to either the number or capacity of the experts. The following MNIST experiments examine these possibilities by extending training and varying the relative sizes of the router and experts.

All experiments use the `RBFRouter`, the `tK` optimization mode, a batch size of 1024, expert dropout of 0.1, and no routing regularization. Training runs for 3000 epochs with patience disabled. Because `tK` retains all experts, the reported `t1_acc1` metric evaluates sparse top-1 routing while `acc1` evaluates the full active mixture. Scalability is considered primarily through the top-1 metric: an increase in `t1_acc1` as K grows, with only a small change in T1 Params, indicates that additional experts may improve sparse inference without requiring all expert paths to be evaluated.

Two router encoders and two expert architectures are considered. The smaller router uses $(P_r, H_r, D_r) = (2, 2, 1)$, while the larger router increases block depth to $(2, 2, 2)$. The smaller expert uses $(P_e, H_e, D_e) = (2, 2, 1)$ and has 192 learnable parameters; the larger expert uses $(2, 3, 1)$ and has 536 parameters. These components are combined into four approaches:

- `EMB+`: smaller router, larger experts, and $N = 2$. This repeats the RBF MoE baseline architecture with a longer training schedule.
- `SRLE`: smaller router, larger experts, and $N = 4$.
- `LRSE`: larger router, smaller experts, and $N = 4$.
- `LRLE`: larger router, larger experts, and $N = 4$.

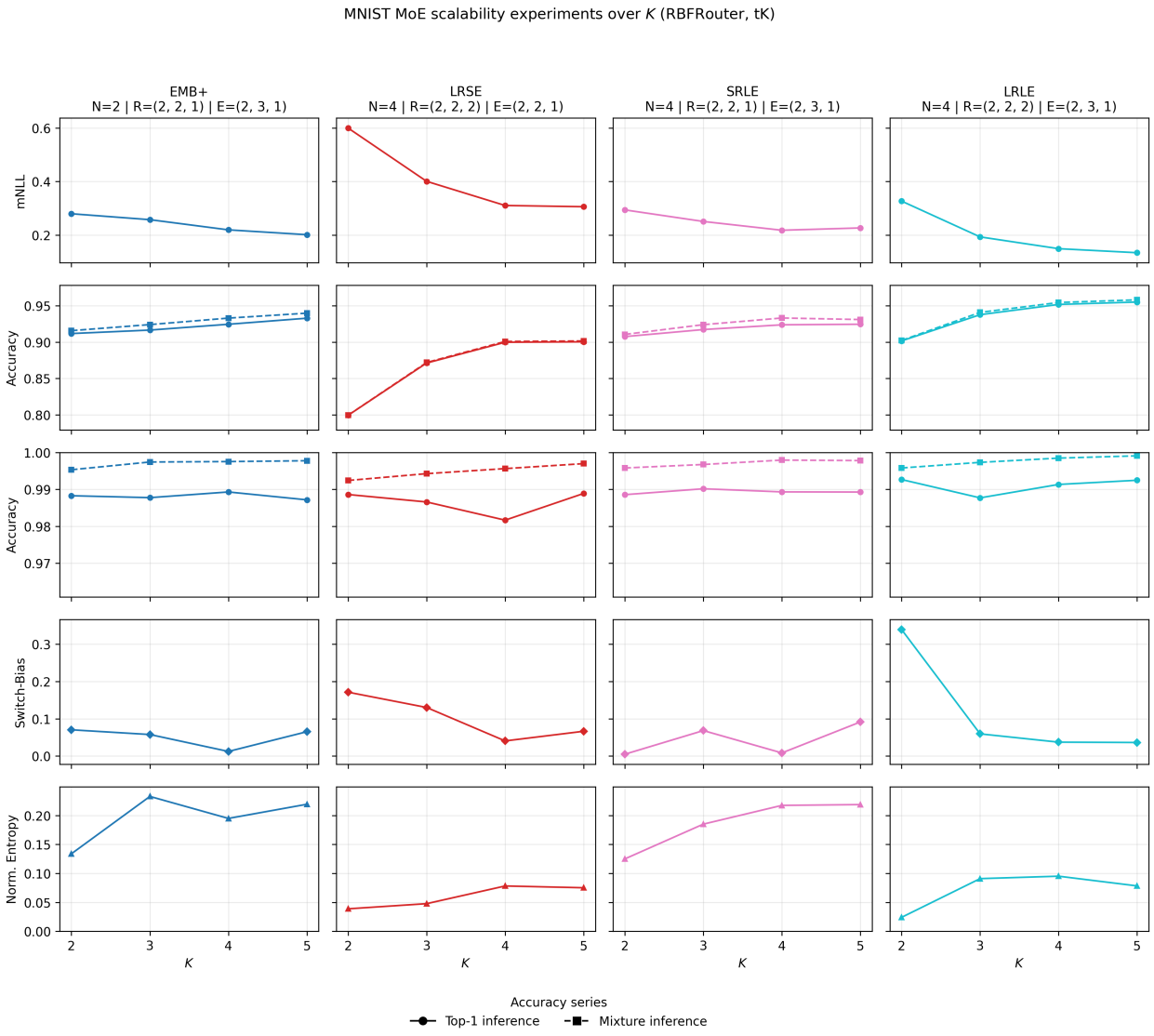


Figure 5.26: MNIST MoE test metrics over the number of experts K with respect to the improved scalability approach.

Table 5.14: MNIST test results for the MoE scalability experiments. Each row reports one trial after 3000 epochs of `tK` training with patience disabled. Params includes the router and all experts; T1 Params reports the learnable parameter count associated with the router and one expert. Loss and other metrics are rounded to four decimals.

Approach	K	Params	T1 Params	Loss	T1 Acc1	T1 Acc5	Acc1	Acc5	Entropy	Switch Bias
`EMB+`	2	1,230	694	0.2796	0.9117	0.9883	0.9157	0.9953	0.1339	0.0704
`EMB+`	3	1,769	697	0.2575	0.9165	0.9877	0.9240	0.9974	0.2329	0.0578
`EMB+`	4	2,308	700	0.2194	0.9244	0.9893	0.9329	0.9975	0.1950	0.0124
`EMB+`	5	2,847	703	0.2013	0.9327	0.9872	0.9397	0.9977	0.2194	0.0654
`SRLE`	2	1,244	708	0.2940	0.9075	0.9886	0.9105	0.9958	0.1251	0.0049
`SRLE`	3	1,785	713	0.2508	0.9172	0.9902	0.9239	0.9967	0.1851	0.0684
`SRLE`	4	2,326	718	0.2179	0.9237	0.9893	0.9330	0.9979	0.2175	0.0084
`SRLE`	5	2,867	723	0.2266	0.9245	0.9893	0.9309	0.9978	0.2188	0.0915
`LRSE`	2	748	556	0.6000	0.7996	0.9886	0.7997	0.9924	0.0389	0.1714
`LRSE`	3	945	561	0.4008	0.8713	0.9866	0.8720	0.9942	0.0478	0.1303
`LRSE`	4	1,142	566	0.3105	0.8997	0.9817	0.9008	0.9956	0.0783	0.0406
`LRSE`	5	1,339	571	0.3059	0.9003	0.9889	0.9015	0.9970	0.0753	0.0665
`LRLE`	2	1,436	900	0.3269	0.9016	0.9927	0.9022	0.9958	0.0242	0.3393
`LRLE`	3	1,977	905	0.1931	0.9376	0.9877	0.9407	0.9973	0.0910	0.0596
`LRLE`	4	2,518	910	0.1490	0.9516	0.9913	0.9543	0.9985	0.0952	0.0373
`LRLE`	5	3,059	915	0.1339	0.9549	0.9925	0.9579	0.9991	0.0784	0.0364

The `EMB+` results provide some support for the longer training schedule. Compared with the RBF `tK` rows in Table 5.10, the loss is lower at each value of K. Top-1 routed accuracy increases by approximately 0.43, 4.06, 0.54, and 0.98 percentage points for $K = 2$, 3, 4, and 5, respectively. Averaged over the four comparisons, this corresponds to an increase of approximately 1.50 percentage points in top-1 routed accuracy and 1.25 percentage points in full-mixture accuracy. The consistent loss reductions suggest that the patience-limited checkpoints may not fully describe the later training behavior.

Increasing the latent size from $N = 2$ to $N = 4$ without changing the router depth or expert architecture has comparatively little effect. The `SRLE` results are close to `EMB+` for $K = 2$, 3, and 4, and are lower at $K = 5$. Across the four values of K, `SRLE` has approximately 0.31 percentage points lower mean top-1 accuracy and 0.35 percentage points lower mean full-mixture accuracy.

The `LRSE` approach shows the largest relative change as experts are added. Its top-1 routed accuracy increases from 79.96% at $K = 2$ to 89.97% at $K = 4$ and 90.03% at $K = 5$. Over the same comparison, T1 Params increases only from 556 to 571, while the parameter count increases from 748 to 1,339. The loss decreases from 0.6000 to 0.3059. Most of the

accuracy increase occurs by $K = 4$, with little additional change from $K = 4$ to $K = 5$. The full-mixture accuracy at $K = 5$ is 90.15%, only about 0.12 percentage points above the top-1 result, suggesting that most of the observed improvement is retained when inference is restricted to one expert.

The `LRSE` results can also be compared with the standalone classifiers in Table 5.3. Its $K = 4$ configuration reaches 89.97% top-1 accuracy with 566 active parameters. The similarly sized $\{P = 4, H = 2, D = 1\}$ classifier uses 590 parameters and reaches 85.18% mean accuracy. The comparison suggests improved active-parameter efficiency for this trial, although the classifier result is averaged over three seeds while the MoE result uses one seed. Total parameters are also greater for the MoE because all four experts are retained.

The `LRLE` approach gives the highest observed accuracy in this set of experiments. Top-1 routed accuracy increases from 90.16% at $K = 2$ to 93.76%, 95.16%, and 95.49% at $K = 3$, 4, and 5. T1 Params changes from 900 to 915, while total parameters increases from 1,436 to 3,059 parameters. At $K = 5$, the full mixture reaches 95.79%, approximately 0.30 percentage points above the selected top-1 expert. Relative to `SRLE`, which uses the same experts and a smaller router, the larger router is lower by approximately 0.59 percentage points at $K = 2$ but higher by approximately 2.04 to 3.05 percentage points for $K = 3$ through $K = 5$.

The comparison between `LRSE` and `LRLE` also demonstrates that expert capacity is important. Both use the larger router, but replacing the 192-parameter experts with 536-parameter experts increases top-1 accuracy by approximately 10.20, 6.63, 5.19, and 5.46 percentage points for $K = 2$ through $K = 5$. The smaller experts therefore provide a clearer relative increase with K, while the larger experts provide the higher absolute accuracy.

5.6.1 Summary

Overall, the experiments suggest that training length and router capacity may both contribute to the limited scaling observed in the original MNIST baselines. Extending training significantly improves the longer-training reference across the tested values of K, while in-

creasing latent size alone has little observable effect. The larger router is associated with greater changes as experts are added, particularly for `LRSE` and `LRLE`. The `LRSE` configuration gives the clearest relative increase with K, whereas `LRLE` gives the highest accuracy.

Chapter 6

Interpretability

6.1 Overview

There are two related hypotheses about interpretability being tested here:

1. The latent space learned by a router encodes meaningful subproblem structure that can be analyzed to improve interpretability through relational and geometric methods.

2. The router can integrate attention-based mechanisms to emphasize important characteristics of input data for human interpretability and potentially improve performance by sharing more information with each expert.

These two hypotheses approach interpretability from related but distinct perspectives. The first hypothesis treats the router as a representational model, where it is explored how well the router can represent high-level feature characteristics. This leverages the natural property of latent spaces utilized by the MoE architecture to make the models more inherently interpretable. If the router learns meaningful subproblem boundaries, then those boundaries should be reflected in the geometry of this latent space. Furthermore, samples routed to the same expert should occupy coherent groupings, and class or object relationships may be visible through the organization, overlap, or separation of latent clusters.

The second hypothesis approaches interpretability through attention mechanisms that can indirectly inform what components of a signal are relevant to the mode. This is tested by integrating an encoder-decoder into the routing mechanism, such as a VAE or UNet. The

resulting auxiliary output of the decoder component can be evaluated to ascertain whether the router emphasizes semantically meaningful image features.

These tests evaluate whether MoE routers can provide interpretability beyond basic expert-selection statistics. The latent-space analysis examines the organization of routed subproblems, while the attention-based routing analysis examines the input features that appear relevant to the router's decision process. One challenge in evaluating these hypotheses, however, is that they encapsulate an intersection between quantitative results and qualitative characteristics–various statistical scoring and human understanding. Therefore, some of the final evaluations will have to rely on insights that are difficult to formalize.

6.2 Latent Interpretability

The router latent space provides a direct object of analysis for evaluating whether a MoE model has learned meaningful subproblem structure. As defined in Chapter 3, the router Φ contains an encoder ϕ that maps an input signal $\mathbf{x}$ into a latent representation.

$$\mathbf{z} = \phi(\mathbf{x}) \quad \mathbf{z} \in \mathbb{L}^N \tag{6.1}$$

Here, N denotes the router latent dimension. For a batch of B samples, the corresponding latent encodings can be collected into a sample-major matrix.

$$Z_{\mathbb{L}} = [\mathbf{z}_1^\top; \mathbf{z}_2^\top; \ldots; \mathbf{z}_B^\top] \in \mathbb{R}^{B \times N} \tag{6.2}$$

This is the same latent representation used by the router to compute logits, routing probabilities, and the sparse top-k expert mixture. Thus, analyzing $Z_{\mathbb{L}}$ provides a geometric view of the internal representation that determines expert selection.

For visualization, each encoded sample $\mathbf{z}_b$ can be colored by either the ground-truth class label y_b or by its routed expert assignment. Given dense routing probabilities $P \in \mathbb{R}^{K \times B}$,

the top-1 routed expert for sample b is defined as follows.

$$\theta_b = \underset{i \in \{1,\ldots,K\}}{\operatorname{argmax}} P_{i,b} \tag{6.3}$$

For sparse top-k routing, the selected expert set is defined as follows.

$$\Theta_b^k = \{i \mid P_{i,b}^k > 0\} \quad \Theta_b^k \subseteq \{1, \ldots, K\} \tag{6.4}$$

Class-colored plots show whether the router encoder preserves semantic class structure, while expert-colored plots show whether the routing function partitions that structure into coherent expert regions.

This work uses three projection methods for this diagnostic analysis: SVD, t-SNE, and UMAP. In each case, the projection maps the router latent matrix $Z_{\mathbb{L}}$ to a two-dimensional visualization matrix.

Figure 6.1 demonstrates how results are later illustrated using SVD as the basis for the example, though it is the same for t-SNE and UMAP projections otherwise. These figures illustrate both class and cluster (how samples are routed) encodings. They further identify regions of overlap between classes/clusters to mitigate the impact of only being able to observe points that are hidden behind others.

Notably, this figure details some expert embeddings corresponding to how the router determines the routing strength of each sample. Note that some later examples will detail figures that seemingly miss some of those embeddings. Given that these plots are a finite view of the space, it is possible for more remote embeddings to not appear within the captured view, but do exist beyond it. This is because the embeddings are learned and not fixed to the space. Regardless, the color-coded clusters will be able to inform which expert the samples are routed to.

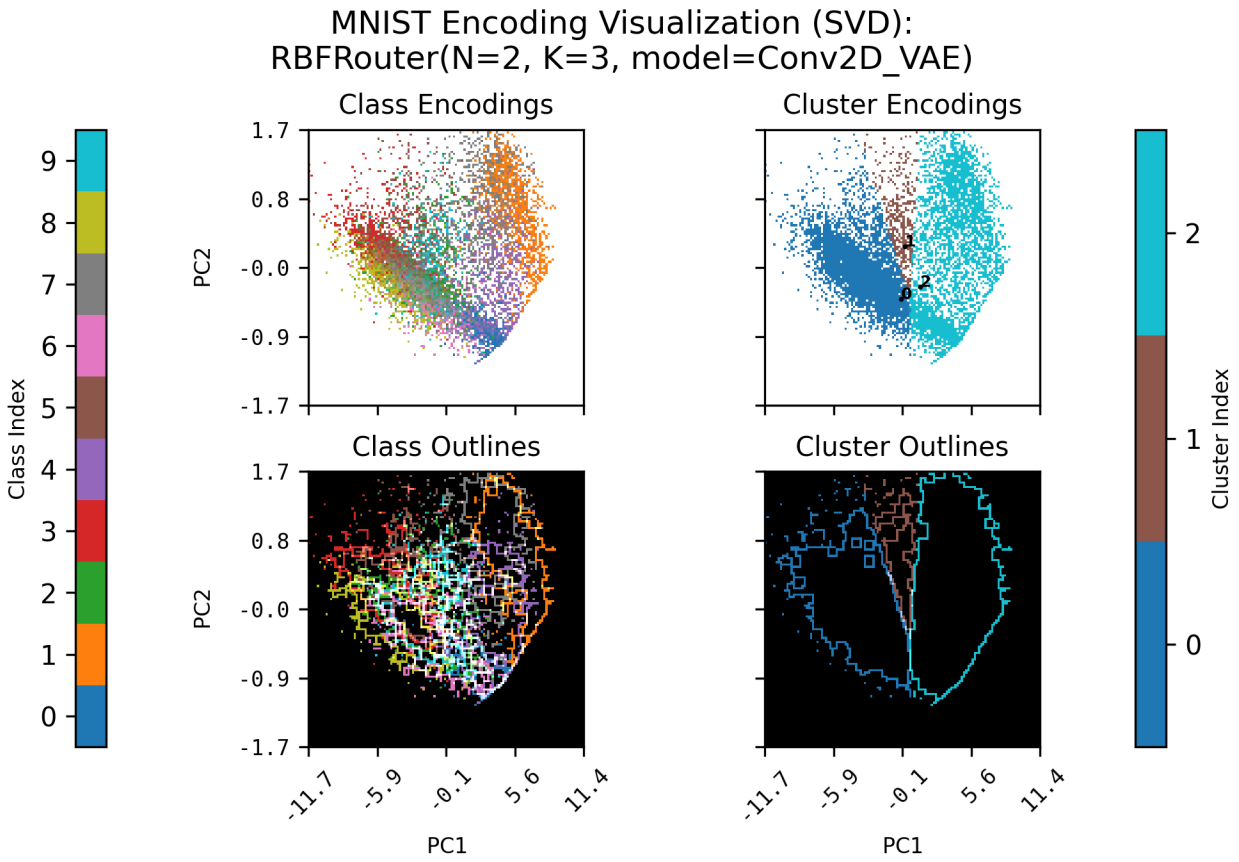


Figure 6.1: Example 2D SVD illustration. Left column details the class encoding features. The right column details the cluster encoding features. The top row is the plotted encodings. The bottom row is the class-cluster outlines to show overlapping regions. The colors in the encodings subfigures are additive, and the colors in the outline subfigures are multiplicative. This is a more stylistic choice done to indicate further overlap, instead of choosing an arbitrary class to obscure the points behind.

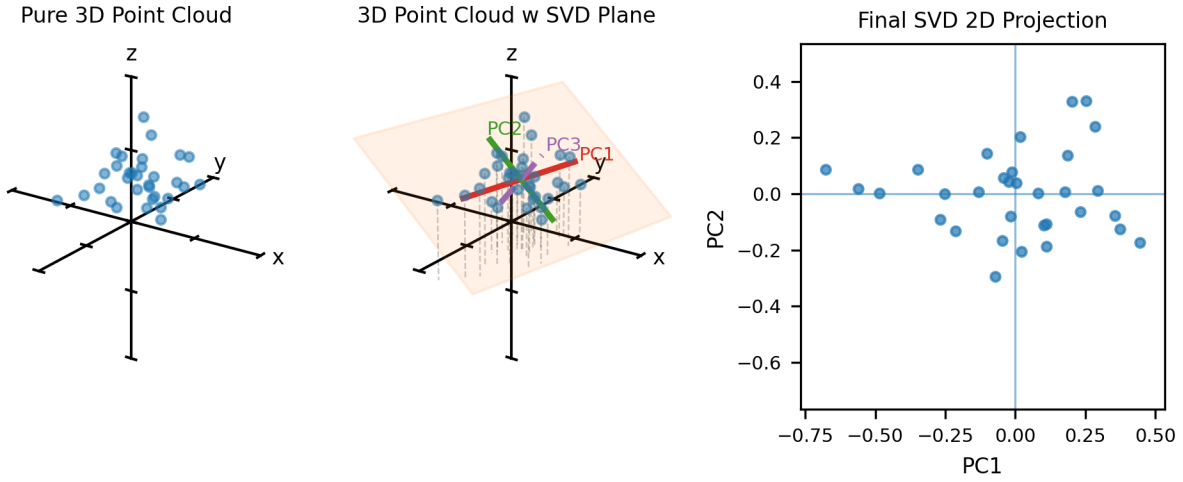


Figure 6.2: SVD projection example. The left panel shows the original 3D point cloud, the middle panel shows the same cloud with the SVD plane spanned by the first two principal component directions, and the right panel shows the resulting 2D projection. The example illustrates that the projection axes are learned from the geometry of the data; the resulting projection does not need to align with the original coordinate axes.

6.2.0.0.1 SVD. SVD provides a deterministic linear view of the router latent space.

$$\bar{\mathbf{z}} = \frac{1}{B}\sum_{b=1}^{B}\mathbf{z}_b \quad \widetilde{Z}_{\mathbb{L}} = Z_{\mathbb{L}} - \mathbf{1}\bar{\mathbf{z}}^{\top} \tag{6.5}$$

The centered latent matrix is decomposed as follows.

$$\widetilde{Z}_{\mathbb{L}} = U\Sigma V^{\top} \tag{6.6}$$

Here, the columns of V define orthonormal directions in the original latent space. The first two right singular vectors, $\mathbf{v}_1$ and $\mathbf{v}_2$, define the two-dimensional SVD projection as follows.

$$\mathbf{q}_b^{\mathrm{SVD}} = [(\mathbf{z}_b - \bar{\mathbf{z}})^{\top}\mathbf{v}_1,\ (\mathbf{z}_b - \bar{\mathbf{z}})^{\top}\mathbf{v}_2]^{\top} \in \mathbb{R}^2 \tag{6.7}$$

For MoE interpretability, this projection details a direct linear relationship to the router

latent space. If samples assigned to the same expert form distinct bands, lobes, or compact regions under SVD, then the router may be using strong global structure in $\mathbb{L}^N$ to form its expert mixture. This is particularly relevant for centroid and RBF routers, where learned centroids $\mathbf{c}_i \in \mathbb{L}^N$ define geometric regions in the latent space. SVD is therefore used as a baseline visualization because it is deterministic, computationally efficient, and comparatively easy to relate to the original latent coordinates.

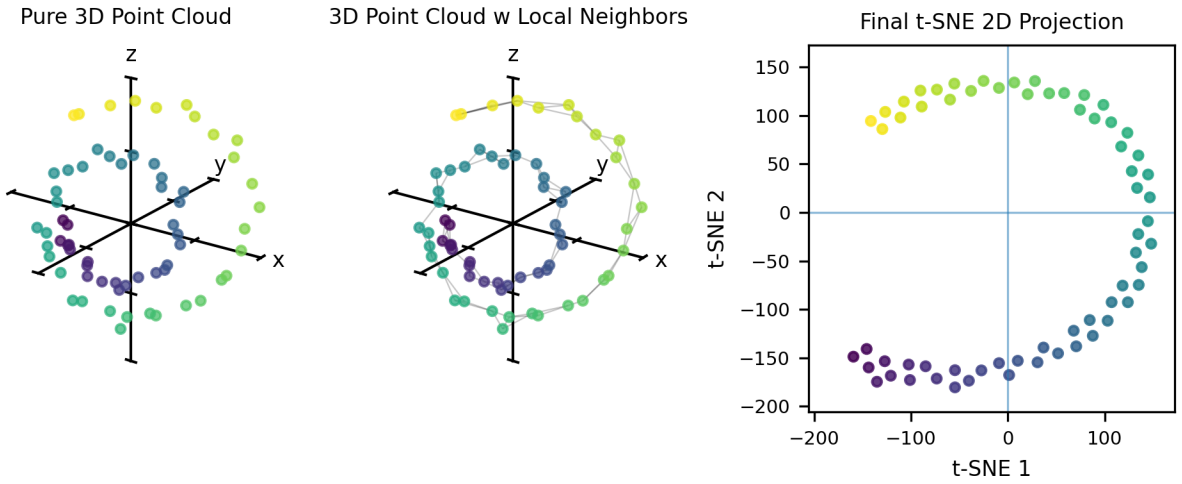


Figure 6.3: t-SNE projection example. The left panel shows the original 3D point cloud, the middle panel shows local neighborhood links in the original space, and the right panel shows the resulting 2D t-SNE embedding. Unlike SVD, t-SNE does not fit a projection plane; it constructs a nonlinear embedding that emphasizes local neighbor preservation.

6.2.0.0.2 t-SNE. t-SNE is a nonlinear visualization method that emphasizes local similarity. Rather than finding linear axes in $\mathbb{L}^N$, t-SNE constructs probability distributions over pairwise neighborhoods in the original latent space and in the low-dimensional embedding, then optimizes the embedding so those neighborhood relationships are similar. In this work, t-SNE is used to test whether routed samples that are near one another in the router latent space stay grouped after nonlinear projection.

The t-SNE methodology can be used to identify local expert specialization. If the expert-colored t-SNE plot encapsulates compact regions for a particular expert, then the router is

likely assigning that expert to a coherent local subproblem. If a single expert appears in several disconnected regions, that can identify if the expert is being used for multiple local modes. Conversely, if many expert colors are heavily mixed throughout the projection, then the routing function may be partitioning the latent space diffusely, forming visually separable expert regions.

The limitation is that t-SNE does not preserve global geometry. Large distances between separated clusters in the projected space do not necessarily imply large distances in $\mathbb{L}^N$. The result can also change with hyperparameters such as perplexity, initialization, and learning rate. For that reason, t-SNE is treated here as qualitative evidence of local routing structure.

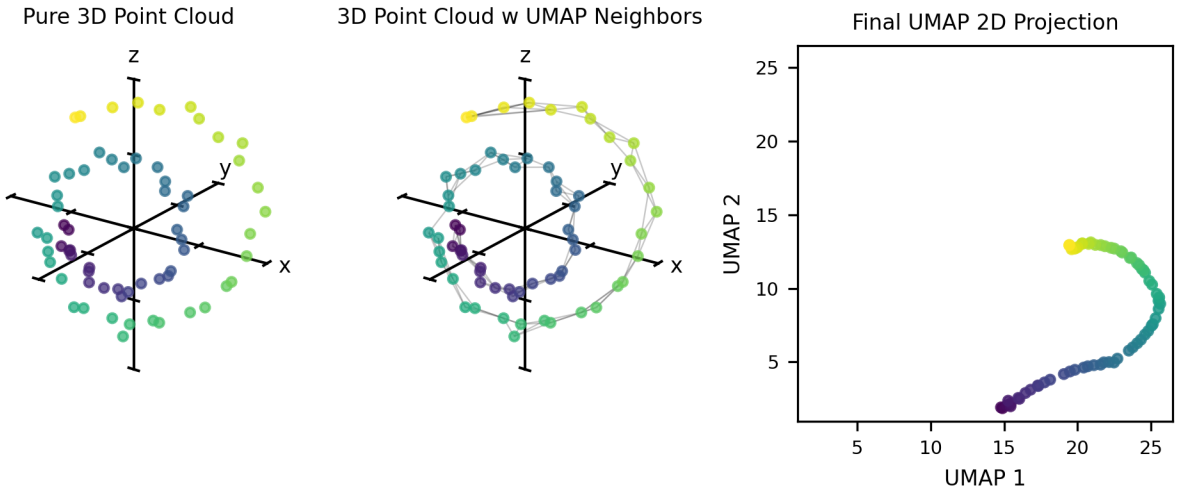


Figure 6.4: UMAP projection example. The left panel shows the original 3D point cloud, the middle panel shows the local neighborhood graph used to summarize the data manifold, and the right panel shows the resulting 2D UMAP embedding. UMAP, like t-SNE, is nonlinear, but its graph-based construction often provides a more stable view of neighborhood-scale manifold structure.

6.2.0.0.3 UMAP. UMAP is also a nonlinear visualization method, but it frames the projection problem through a neighborhood graph over the latent samples. In the original router latent space, nearby samples define local graph connections. UMAP then optimizes a low-dimensional embedding that preserves this graph structure as much as possible. In the

MoE setting, this makes UMAP especially relevant for evaluating whether the router has organized samples into connected regions that correspond to expert specialization.

Compared with t-SNE, UMAP details intermediate-scale structure. It can still reveal local clusters, but it may also preserve more of the coarse organization between nearby groups.

As with t-SNE, UMAP plots must be interpreted carefully. The projected coordinates are diagnostic. Hyperparameters such as neighborhood size and minimum distance can change the apparent separation of clusters. Thus, UMAP is used together with SVD and t-SNE: SVD provides a global linear baseline, t-SNE emphasizes local neighbor preservation, and UMAP provides a graph-based view of local and intermediate manifold structure.

6.2.0.0.4 Usage. Across all three methods, the focus is on whether the router latent representation $\mathbf{z} = \phi(\mathbf{x})$ organizes samples in a way that corresponds to meaningful subproblem structure. To encourage interpretability, a router should form a latent space in which expert assignments, class labels, and ambiguous samples can be analyzed geometrically. When class-colored and expert-colored projections show similar regions, the router may be specializing experts according to semantic class structure. When the two views differ, the router may instead be using lower-level visual similarity, sample difficulty, or other features that do not align directly with class identity.

These visualizations therefore provide qualitative support for the broader interpretability hypothesis. They do not prove that the router has learned a specific causal explanation for its decisions, but they do expose the relational structure used by the routing mechanism.

The class-colored projections show whether the router latent space preserves semantic class relationships. The expert-colored projections show whether the router partitions the latent space into coherent expert regions. Comparing the two makes it possible to distinguish between class-aligned routing and more general subproblem routing. For example, a router may assign several visually similar classes to the same expert, or it may split a single class

across multiple experts when that class contains multiple visual modes.

The most interpretable result occurs when the class-colored and expert-colored projections show related but not identical structure. This identifies that the router is forming subregions that reflect meaningful latent relationships within the data.

6.2.1 MNIST

Each visualization grid uses the same ordering. Rows correspond to router type, ordered as Linear, Cosine, Centroid, and RBF. Columns correspond to the number of experts, ordered as $K = 2$, $K = 3$, $K = 4$, and $K = 5$. All panels use the Conv2D_FFNN encoder ($P = 2, H = 2, D = 1$) with latent dimension $N = 2$ and Conv2D_FFNN experts ($P = 2, H = 3, D = 1$).

6.2.1.1 SVD

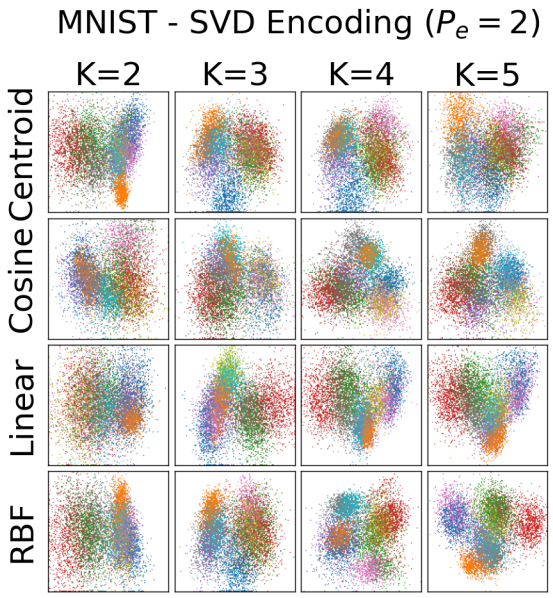


Figure 6.5: Grid of 2D SVD projections for MNIST latent encodings. Router uses a linear algorithm and a `tK` optimization approach with $P = 2, H = 2, D = 1, K = 2, N = 2$ and experts are $P = 2, H = 3, D = 1$. These figures use the class-encoding method and color map detailed in Figure 6.1, excluding the outline and cluster subfigures for simplicity.

Figure 6.5 provides a high-level overview of the 2D SVD projections for the learned latent

encodings over the MNIST dataset. Since the latent space here is 2D ($N = 2$), no features are obscured by high dimensionality.

Notably, similar router methodologies typically result in similar latent encoding behaviors. Note that in $K = 4$, the RBF router begins to form mini-circular clusters, likely encouraged to conform to the structures of some Gaussian mixture model as defined by the RBF, which are further strengthened by $K = 5$. Furthermore, in $K = 2$, the Centroid and RBF provide clear delineations between major groups, though interestingly the Linear router does not. This is likely because the Linear router does not provide any built-in bias, which may blur the edges of group boundaries, making them difficult to distinguish.

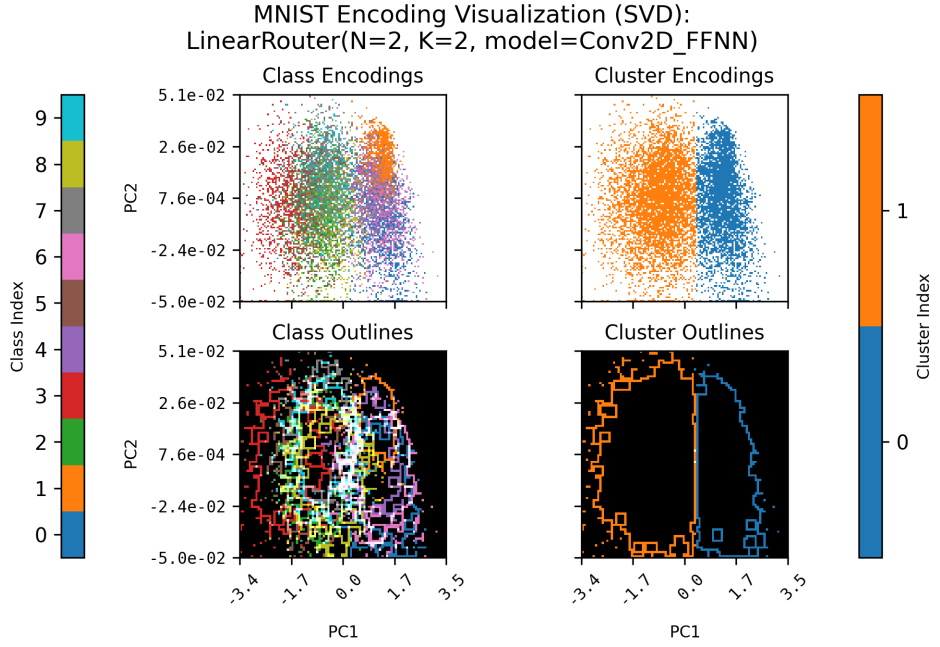


Figure 6.6: Example MNIST latent encodings using 2D SVD projection. Linear routing with a Conv2D encoding model ($P = 2, H = 2, D = 1$) trained using `tK` with Conv2D expert submodels ($P = 2, H = 3, D = 1$).

Note that in Figure 6.6 there are two experts. Expert 0 is allocated (by the router) samples that are encoded to the right side of the space, which includes classes 0, 1, 4, and 6. Expert 1 is allocated samples that are encoded to the left side, including the remainder of classes: 2, 3, 5, 7, 8, 9.

Figure 6.7 details the expert specializations via diagonals per expert submodel using the

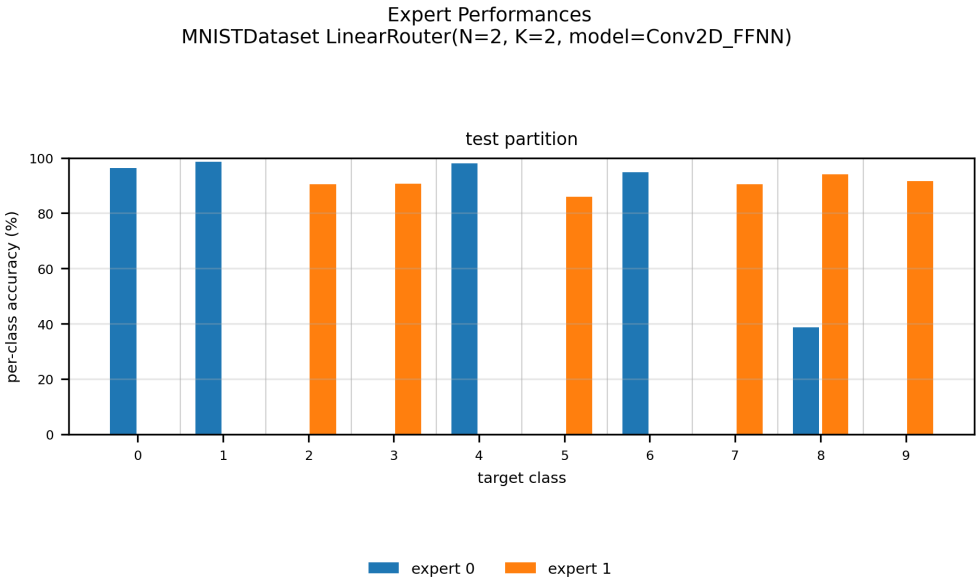


Figure 6.7: Example MNIST latent specializations corresponding to Figure 6.6. Specializations are identified by extracting the diagonals of each expert's confusion matrix (acc1) and plotted as a side-by-side bar plot to compare performances on labeled categories versus other experts.

t1 accuracy metrics. Note the specializations largely overlap with what is visually confirmed by the router. However, interestingly, expert 0 and 1 are both capable of more than just their partitions still. This is likely owing to the `tK` optimization process, which sees that all experts see all samples for the entire training process.

Note that in Figure 6.6 there are two experts. Expert 0 is allocated (by the router) samples that are encoded to the right side of the space, which includes classes 0, 1, 4, and 6. Expert 1 is allocated samples encoded to the left side, including the remaining classes: 2, 3, 5, 7, 8, 9.

Observe then in Figures 6.8 and 6.9, which use a `tK2t1` router optimization approach, that the specializations become significantly more exclusive. Additionally, expert 0 also specializes in class 7. This level of variation can be due to training variability or potentially different optimal routing strategies.

In general, while the visualizations of the latent space can visually convey which image classes are associated with which experts, that same information is also somewhat captured

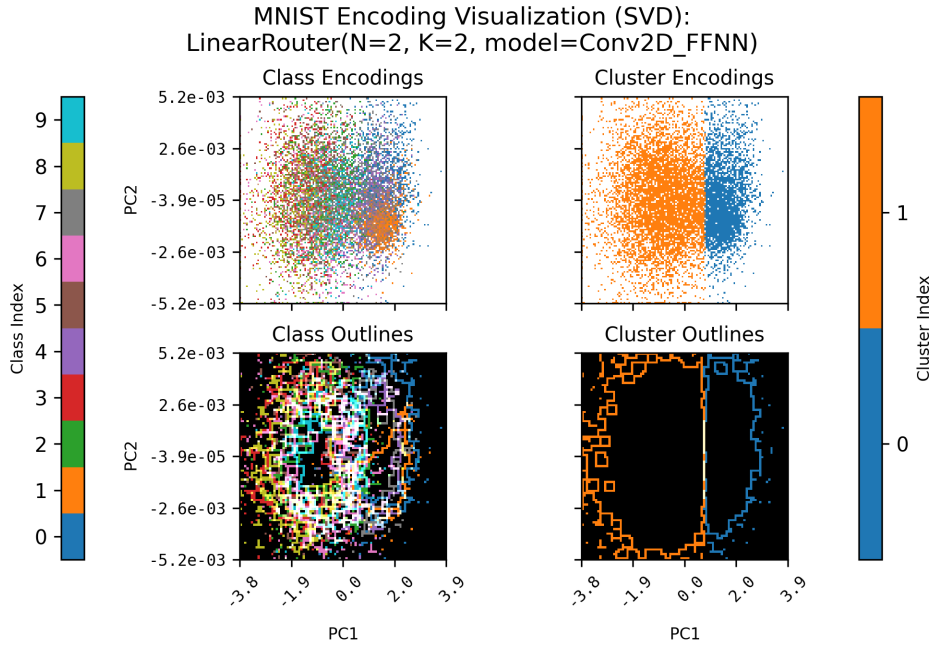


Figure 6.8: Example MNIST latent encodings using 2D SVD projection. Linear routing with a Conv2D encoding model ($P = 2, H = 2, D = 1, N = 2, K = 2$) trained using `tK2t1` with Conv2D expert submodels ($P = 2, H = 3, D = 1$).

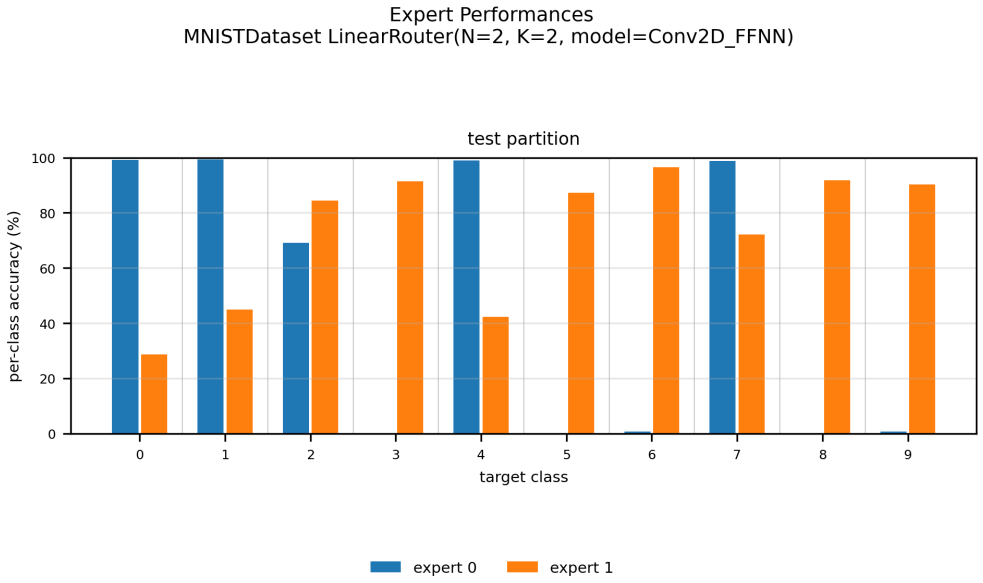


Figure 6.9: Example MNIST latent specializations corresponding to Figure 6.8. Specializations are identified by extracting the diagonals of each expert's confusion matrix (acc1) and plotted as a side-by-side bar plot to compare performances on labeled categories versus other experts.

by the confusion matrices by identifying which experts perform well on which image classes. Furthermore, forcing hardening may cause additional blur in the latent space, since experts can benefit from knowing a little just beyond their boundaries.

6.2.1.2 t-SNE and UMAP

The t-SNE and UMAP 2D projection methods struggle to provide meaningful interpretation of the learned routing behaviors. However, they do provide some insight into routing overlap that the SVD approach struggles with. This section, as well as the subsequent similar sections for CIFAR-10 and CIFAR-100, will present both approaches jointly, providing insight into their behaviors.

The usefulness of the information largely depends on what these projection methods do. They fundamentally map the inputs using their relational behaviors to form high-level representations of associativity. This works well, such as in the example figures in Figures 6.3 and 6.4, but these convey simple patterns with few points. These latent encodings here are composed of thousands of samples, forming significantly more complex behaviors in ways that are more difficult for t-SNE and UMAP to handle with any degree of readability. Thus, their contribution to interpretability is more in the association of clusters and embedded experts with each other, not the latent encoding and clustering of samples.

Figures 6.10 and 6.11 detail the t-SNE and UMAP projection examples. The inherent challenge with interpreting these is how they inherently squash information and distort it into messy strings. This high-level overview further corroborates the similarities between the t-SNE and UMAP approaches, though they still functionally differ.

Figures 6.12, 6.13, 6.14, and 6.15 detail the t-SNE and UMAP examples for the prior SVD examples given in Figures 6.6 and 6.8. While class-cluster correlations are difficult to establish, cluster-cluster correlations are guaranteed through how the algorithms map the embedded experts into the relational mapping. This can be potentially useful to identify when experts are fighting for resources–samples–or are otherwise distinct, potentially

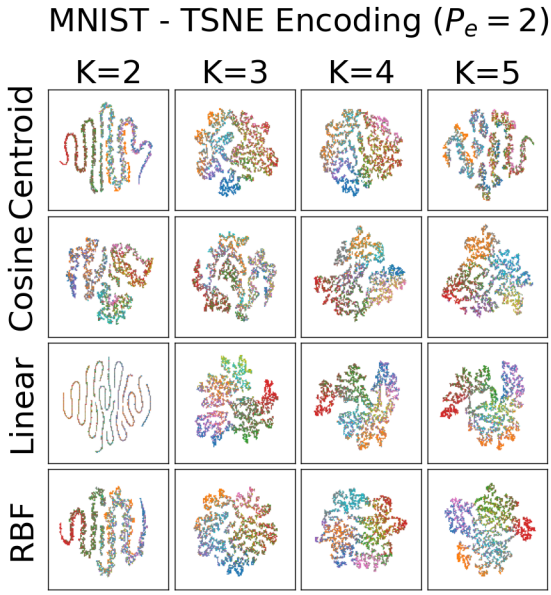


Figure 6.10: Grid of 2D t-SNE projections for MNIST latent encodings. Router uses a linear algorithm and a `tK` optimization approach with $P = 2, H = 2, D = 1, K = 2, N = 2$ and experts are $P = 2, H = 3, D = 1$. These figures use the class-encoding method and color map detailed in Figure 6.1, excluding the outline and cluster subfigures for simplicity.

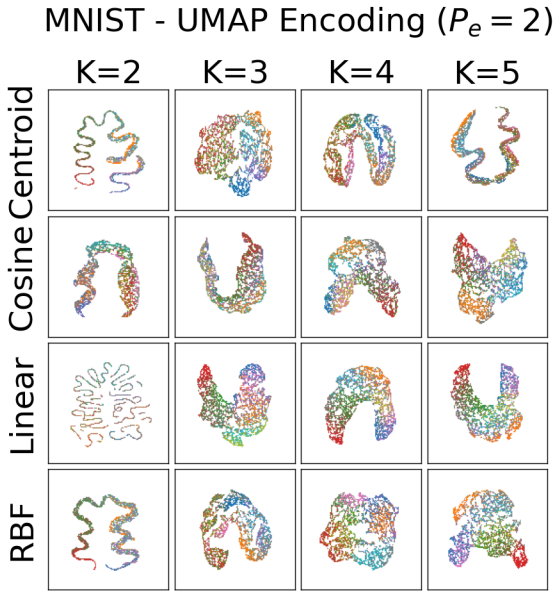


Figure 6.11: Grid of 2D UMAP projections for MNIST latent encodings. Router uses a linear algorithm and a `tK` optimization approach with $P = 2, H = 2, D = 1, K = 2, N = 2$ and experts are $P = 2, H = 3, D = 1$. These figures use the class-encoding method and color map detailed in Figure 6.1, excluding the outline and cluster subfigures for simplicity.

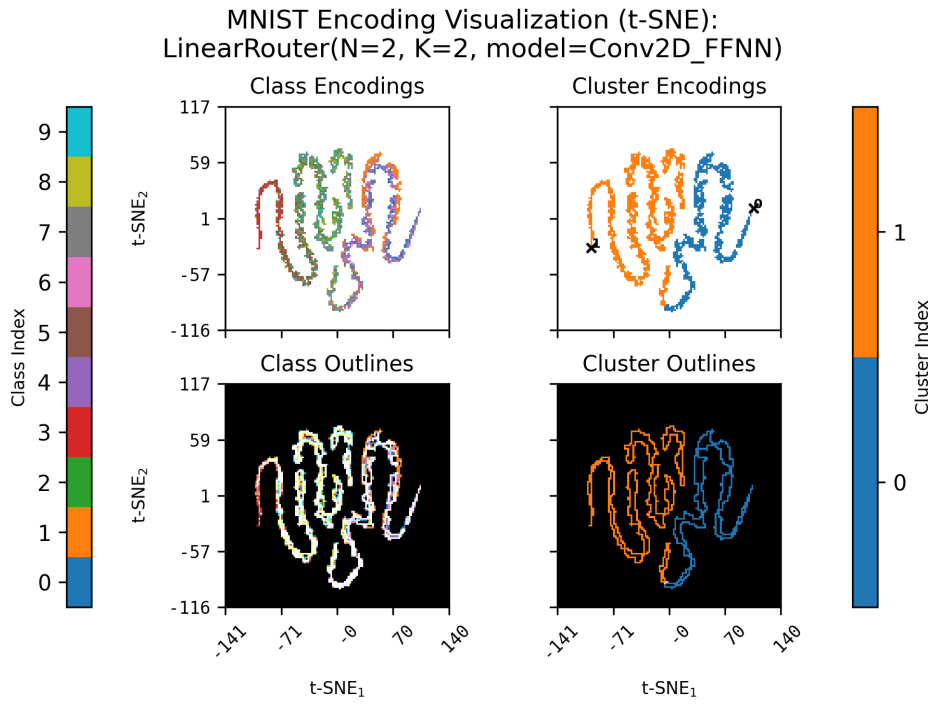


Figure 6.12: Example MNIST latent encodings using 2D t-SNE projection. Linear routing with a Conv2D encoding model ($P = 2, H = 2, D = 1, N = 2, K = 2$) trained using `tK` with Conv2D expert submodels ($P = 2, H = 3, D = 1$).

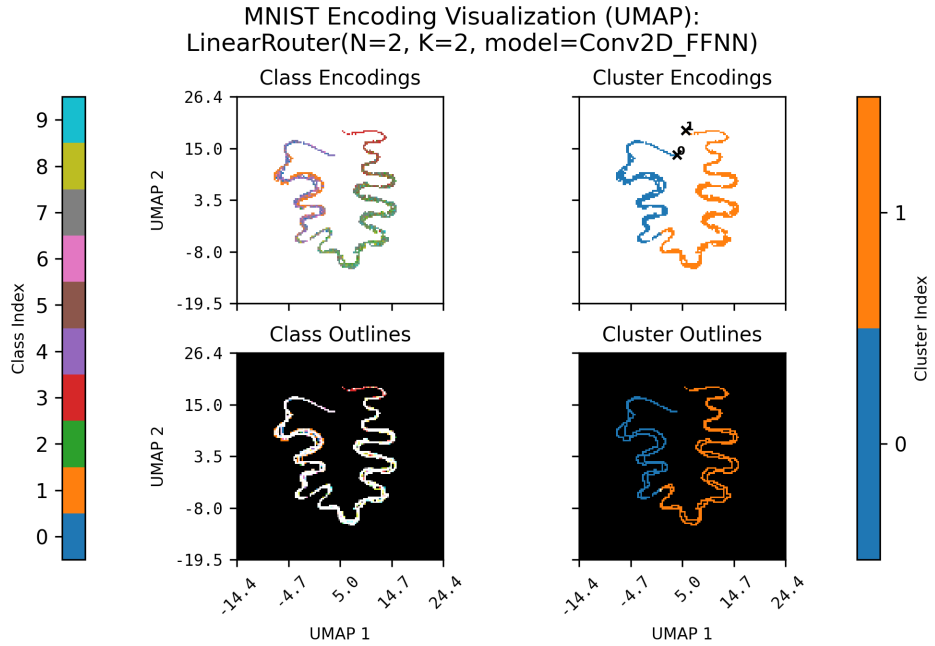


Figure 6.13: Example MNIST latent encodings using 2D UMAP projection. Linear routing with a Conv2D encoding model ($P = 2, H = 2, D = 1, N = 2, K = 2$) trained using `tK` with Conv2D expert submodels ($P = 2, H = 3, D = 1$).

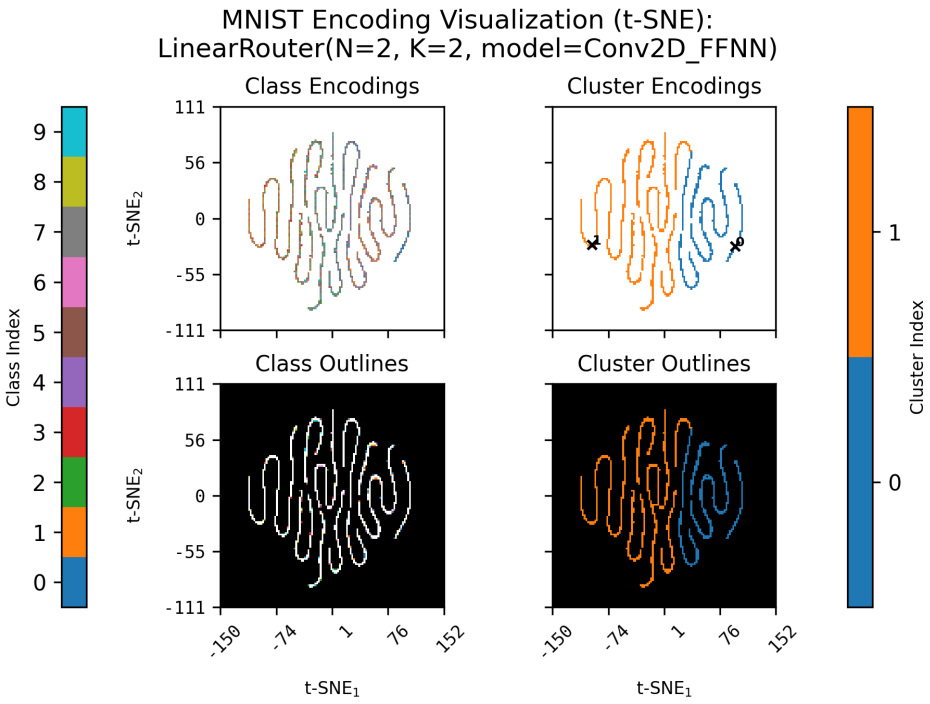


Figure 6.14: Example MNIST latent encodings using 2D t-SNE projection. Linear routing with a Conv2D encoding model ($P = 2, H = 2, D = 1, N = 2, K = 2$) trained using `tK2t1` with Conv2D expert submodels ($P = 2, H = 3, D = 1$).

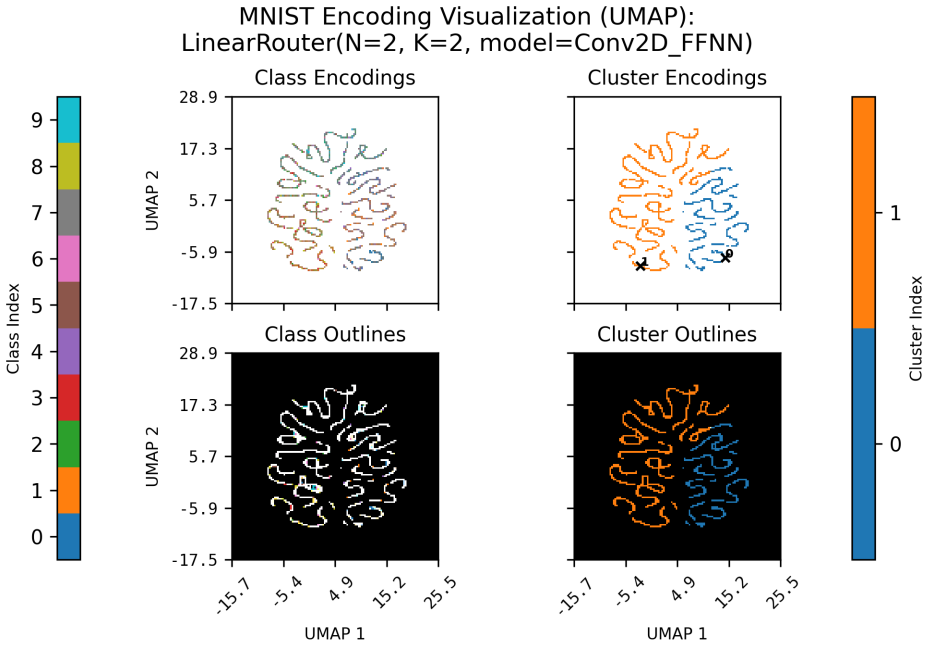


Figure 6.15: Example MNIST latent encodings using 2D UMAP projection. Linear routing with a Conv2D encoding model ($P = 2, H = 2, D = 1, N = 2, K = 2$) trained using `tK2t1` with Conv2D expert submodels ($P = 2, H = 3, D = 1$).

even unrelated to most of the encoded samples. Note that the `tK` embedded experts have much stronger proximity in the UMAP encoding than in any other, potentially indirectly informing us of their similar performances across several labels. However, t-SNE, by comparison, seems less useful since it does not convey any information with apparent meaning for interpretability.

6.2.2 CIFAR-10

The CIFAR-10 examples compare the `tK2t1` and `tK2tk` optimization approaches for a `LinearRouter` with $K = 2$ and latent dimension $N = 8$. The router encoder is a `Conv2D_FFNN` with $(P_r, H_r, D_r) = (4, 3, 2)$, and the experts are `Conv2D_FFNN` models with $(P_e, H_e, D_e) = (8, 3, 4)$. Both experiments use dropout 0.2. Unlike the MNIST examples, the learned latent space is eight-dimensional, so the SVD, t-SNE, and UMAP figures provide two-dimensional projections.

6.2.2.1 SVD

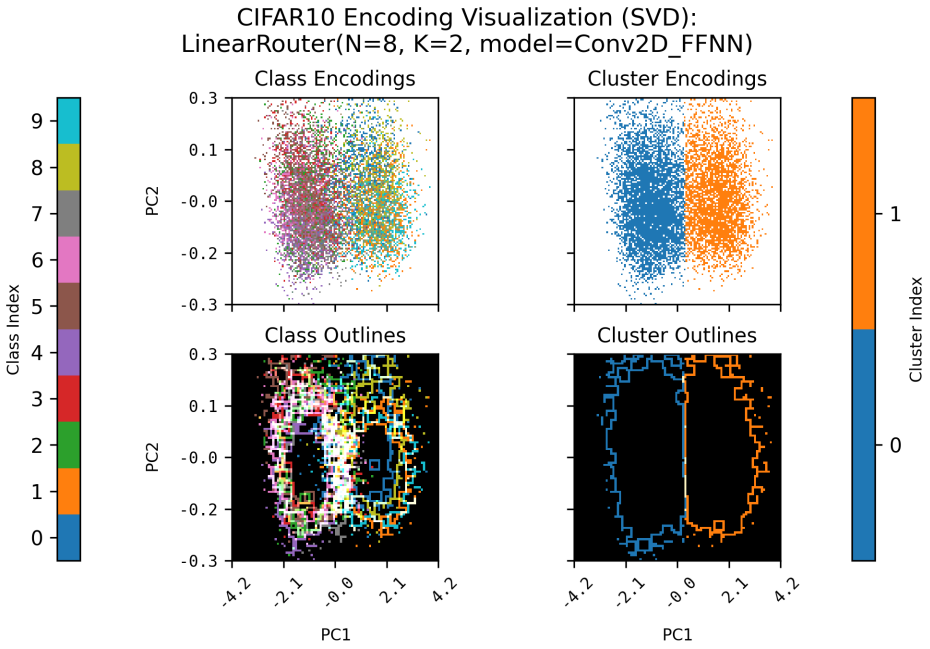


Figure 6.16: CIFAR-10 latent encodings using a 2D SVD projection for the `tK2t1` approach. The example uses a Linear router with $(P_r, H_r, D_r, K, N) = (4, 3, 2, 2, 8)$ and experts with $(P_e, H_e, D_e) = (8, 3, 4)$. The embedded experts are shown as star markers.

Figures 6.16 and 6.17 show a similar broad routing structure for both optimization approaches. The embedded experts are separated along the main diagonal direction in each projection, and the expert assignments form two partially overlapping regions between them. The `tK2t1` projection appears to have a somewhat clearer diagonal separation, while the

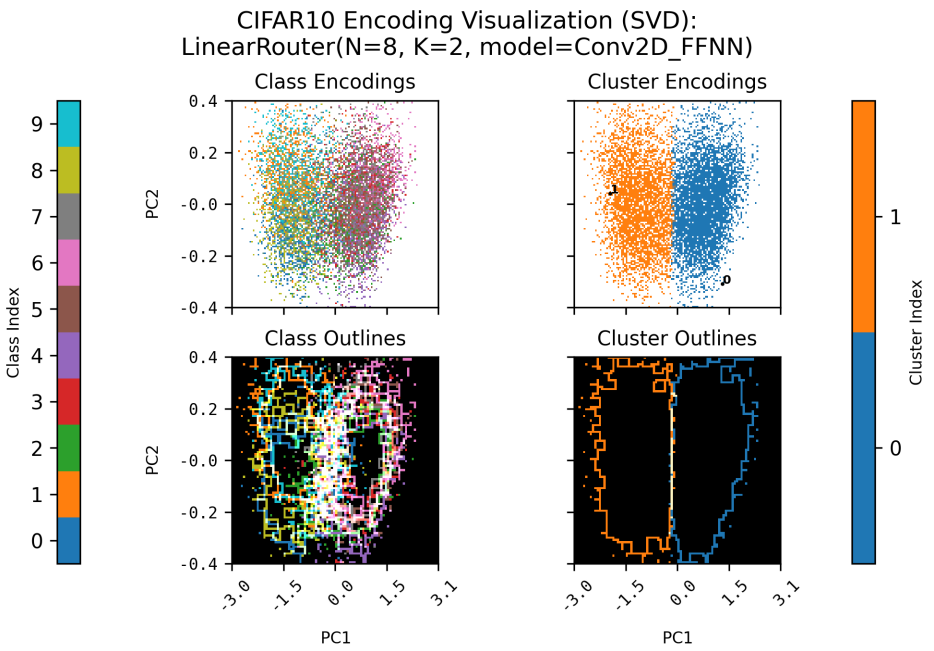


Figure 6.17: CIFAR-10 latent encodings using a 2D SVD projection for the `tK2tk` approach. The example uses a Linear router with $(P_r, H_r, D_r, K, N) = (4, 3, 2, 2, 8)$ and experts with $(P_e, H_e, D_e) = (8, 3, 4)$. The embedded experts are shown as star markers.

`tK2tk` projection has more mixing near the center. However, these figures only display the first two SVD components of the $N = 8$ latent representation, so some separation/structure may occur along dimensions that are not shown.

The class-colored points stay significantly mixed under both approaches. Some classes occupy more restricted portions of the projected space, but there is no simple division where each expert corresponds to a distinct set of CIFAR-10 labels. The router may instead be using lower-level visual characteristics that occur across several classes. This differs from the MNIST examples, where the class groups are more visibly separated in the latent space.

Figures 6.18 and 6.19 provide a more direct comparison of the learned expert specializations. With `tK2t1`, expert 0 performs better on classes 0, 1, 3, 6, and 8, while expert 1 performs better on classes 2, 4, 5, 7, and 9. The differences are largest for several classes such as 2, 4, 8, and 9, while the expert accuracies are closer for classes 3, 5, and 7.

The `tK2tk` results retain a similar allocation. Expert 0 is stronger on classes 0, 1, 6, and 8, while expert 1 is stronger on classes 2, 4, 5, 7, and 9. Class 3 changes from a modest

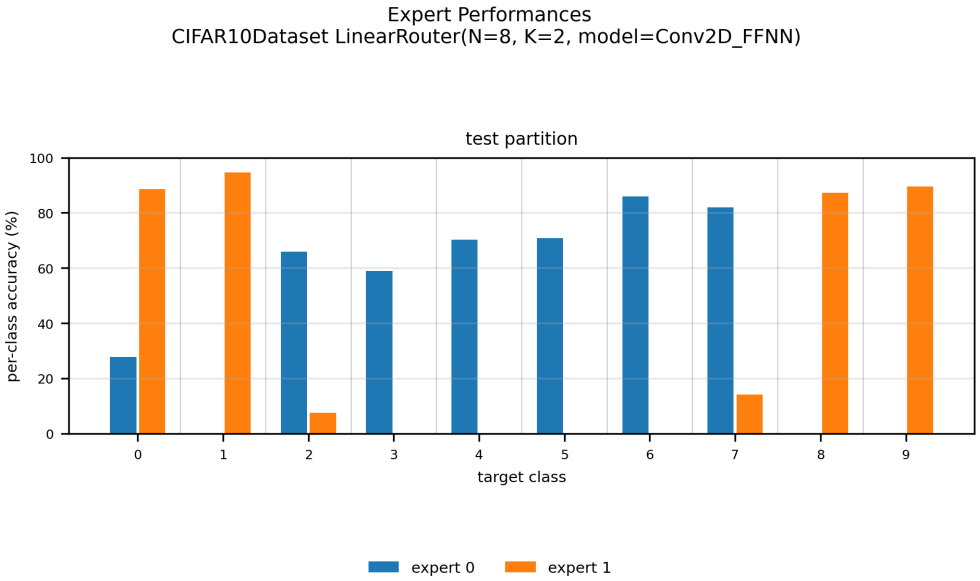


Figure 6.18: CIFAR-10 expert specializations corresponding to Figure 6.16. Bars report the diagonal values of each expert's top-1 confusion matrix under `tK2t1`.

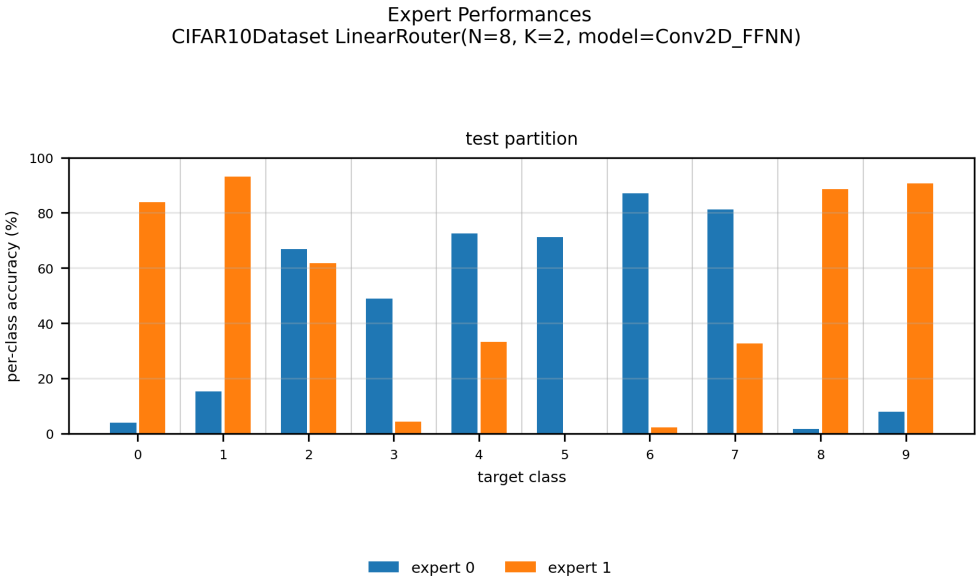


Figure 6.19: CIFAR-10 expert specializations corresponding to Figure 6.17. Bars report the diagonal values of each expert's top-1 confusion matrix under `tK2tk`.

advantage for expert 0 under `tK2t1` to a modest advantage for expert 1 under `tK2tk`. The staged hardening approach also produces somewhat larger differences for classes 4 and 5. Overall, the two approaches lead to related class-wise behaviors.

The specializations are not exclusive, since both experts retain meaningful accuracy across most categories. This may be useful when the routing boundary does not align directly with the class labels, because an expert can still classify samples routed near or beyond the region where it performs best.

6.2.2.2 t-SNE and UMAP

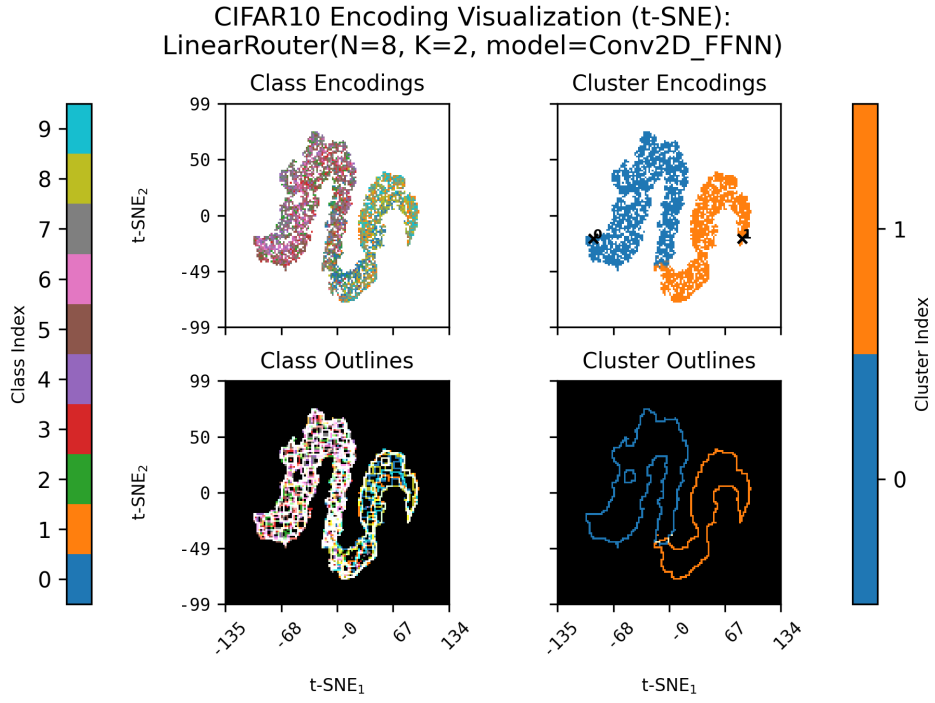


Figure 6.20: CIFAR-10 latent encodings using a 2D t-SNE projection under `tK2t1`. The class encoding is shown in the left panels and the corresponding expert assignments are shown in the right panels.

The t-SNE projections in Figures 6.20 and 6.21 form several curved and partially connected structures. The class-colored views show some local concentrations, but most regions contain samples from multiple labels. The expert-colored views show a broader division, with one expert more common across one side of the projected structure and the other more

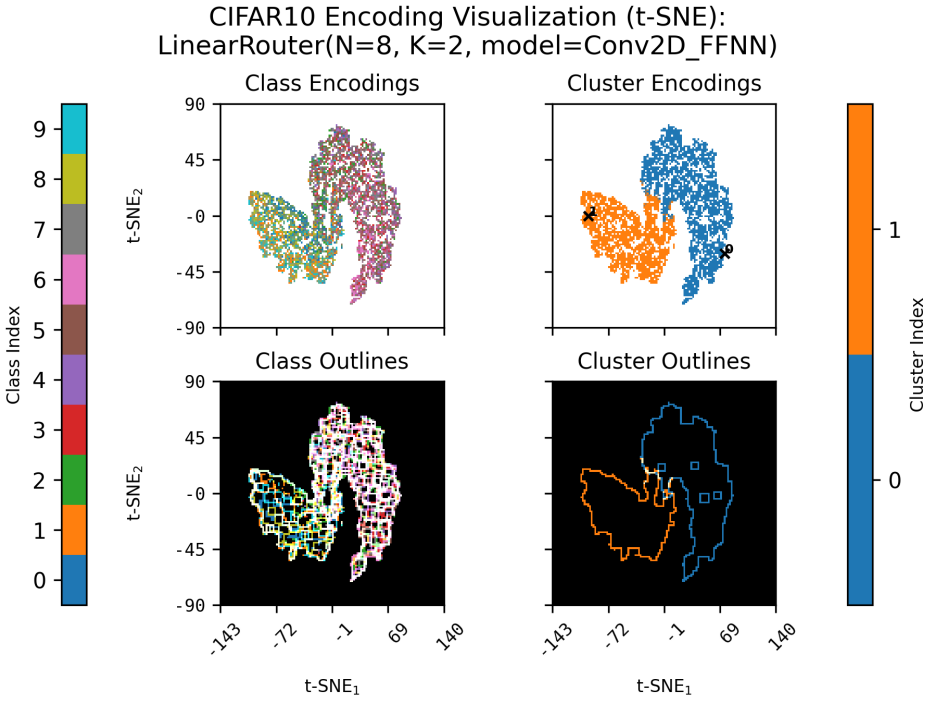


Figure 6.21: CIFAR-10 latent encodings using a 2D t-SNE projection under `tK2tk`. The class encoding is shown in the left panels and the corresponding expert assignments are shown in the right panels.

common across the opposite side. Both approaches still contain substantial overlap near the center and along several branches.

Figures 6.22 and 6.23 give more compact representations of the latent encodings. As in the SVD and t-SNE projections, the classes stay mixed, although some labels are more common within particular branches and outer regions. The expert-colored views show broader contiguous regions for each routing assignment. The `tK2t1` example appears to place the two embedded similarly far as the `tK2tk` does. However, note that their performance distributions are very similar across routing optimizations, lightly corroborating the earlier suggestion that the UMAP can indeed explain some inner-expert behaviors.

Across the three projection methods, `tK2t1` and `tK2tk` produce broadly similar routing organizations. Direct hardening appears somewhat more separated in the SVD and UMAP examples, while staged hardening changes some local assignments and class-wise expert strengths.

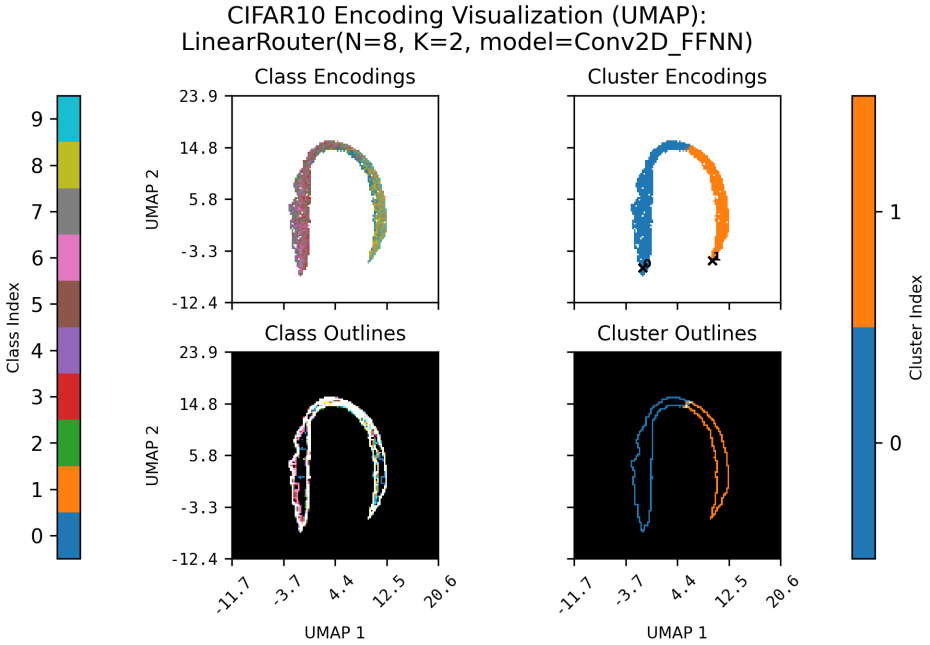


Figure 6.22: CIFAR-10 latent encodings using a 2D UMAP projection under `tK2t1`. The class encoding is shown in the left panels and the corresponding expert assignments are shown in the right panels.

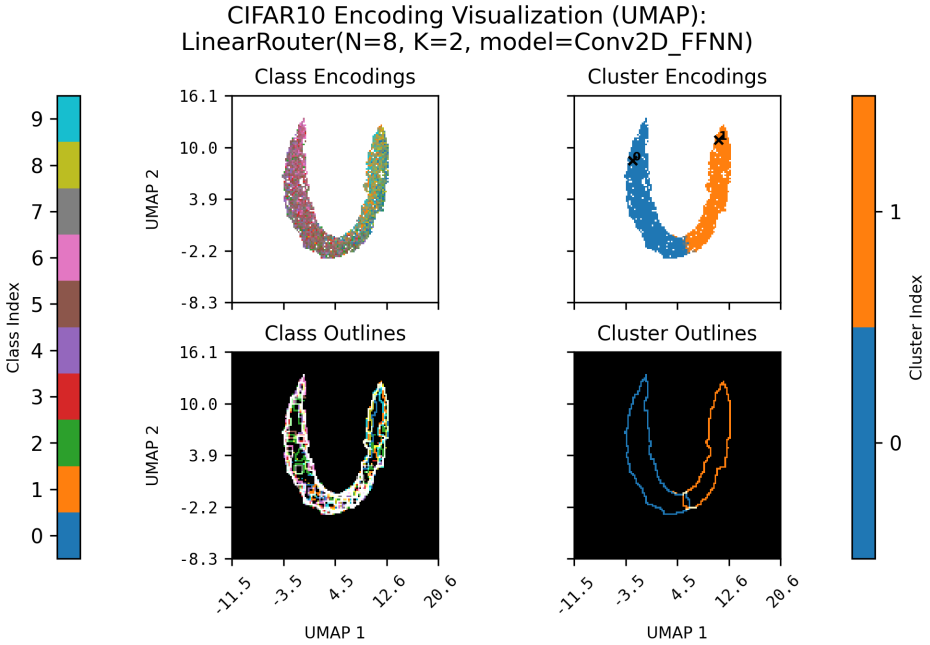


Figure 6.23: CIFAR-10 latent encodings using a 2D UMAP projection under `tK2tk`. The class encoding is shown in the left panels and the corresponding expert assignments are shown in the right panels.

Notably, the MoE router projections show more visible geometric structure than the autoencoder baselines even though the router encoder is substantially smaller than the autoencoders used in the reconstruction baseline. While the autoencoders are trained only through reconstruction, these router encoders are trained through the supervised mixture objective on a classification challenge. Thus, the difference is likely due to what information the router learns to leverage. In the reconstruction context, such fine-level detail, real-world classes can be very similar and thus do not naturally cluster along class boundaries. The class-colored projections are more mixed than the MNIST projections, but the expert-colored views often show coherent router partitions.

6.2.3 CIFAR-100

The CIFAR-100 example uses a `LinearRouter` with $K = 2$ and latent dimension $N = 32$. The router encoder is a `Conv2D_Res` model with $(P_r, H_r, D_r) = (16, 3, 4)$, and the experts are `Conv2D_Res` models with $(P_e, H_e, D_e) = (32, 3, 8)$. The model is trained using `tK2t1` with dropout 0.5. Because CIFAR-100 contains many fine-grained labels, the visualization groups the 100 classes into 20 superclass categories, with five dataset classes assigned to each displayed superclass index. The learned routing representation is 32-dimensional, so each figure is a two-dimensional projection and does not directly display the full latent space.

6.2.3.1 SVD

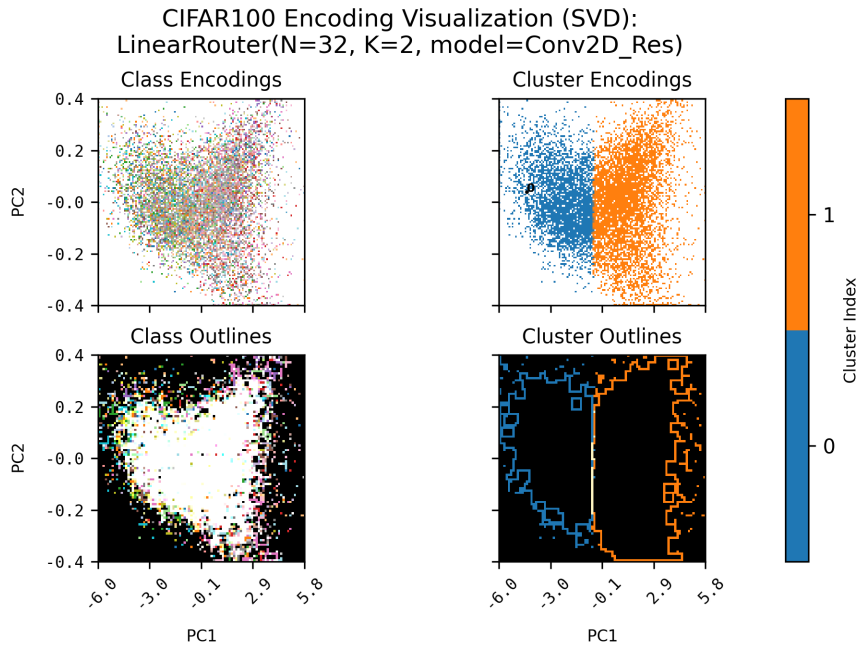


Figure 6.24: CIFAR-100 latent encodings using a 2D SVD projection. The example uses a Linear router with $(P_r, H_r, D_r, K, N) = (16, 3, 4, 2, 32)$ and residual experts with $(P_e, H_e, D_e) = (32, 3, 8)$. Class colors represent the 20 CIFAR-100 superclass groups, and the embedded experts are shown as star markers.

Figure 6.24 demonstrates broad overlap among the superclass-colored encodings. The first two SVD components do not cleanly divide the superclass groups. Some colors appear more frequently near particular edges of the projected distribution. By comparison, the

expert-colored view is divided mainly along the first component, with expert 0 assigned more often to the left side and expert 1 assigned more often to the right side. The assignments overlap near the center, which may correspond to inputs whose routing scores are similar for both experts.

This projection suggests that the router division is simpler than the superclass organization visible in two dimensions. The routing boundary may use combinations of features that cut across the semantic superclass labels without directly reproducing those labels. Since only two of the 32 latent dimensions are displayed, the degree of superclass organization in the complete representation cannot be determined from this figure alone.

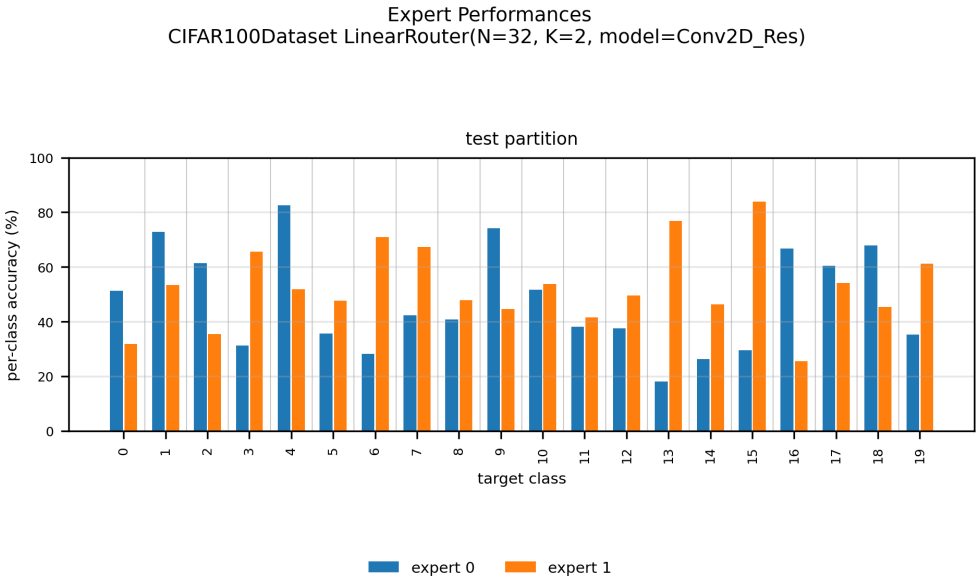


Figure 6.25: CIFAR-100 expert specializations corresponding to Figure 6.24. Bars report the diagonal values of each expert's top-1 confusion matrix after grouping the 100 dataset classes into 20 superclass categories.

Figure 6.25 demonstrates complementary performance across several superclass groups. Expert 0 has higher accuracy for groups such as 1, 2, 4, 9, 16, and 18, while expert 1 has higher accuracy for groups such as 3, 6, 7, 13, 15, and 19. Some of the larger differences occur for groups 4, 9, 13, 15, and 16. For example, expert 0 reaches approximately 83% accuracy on group 4 compared with approximately 52% for expert 1, whereas expert 1 reaches approximately 77% on group 13 compared with approximately 18% for expert 0.

Both experts also achieve nonzero accuracy across all displayed groups, and several groups have closer performances. The specialization is therefore not equivalent to assigning a disjoint set of superclasses to each expert. Instead, each expert appears better suited to some portions of the data while preserving broader classification ability.

6.2.3.2 t-SNE and UMAP

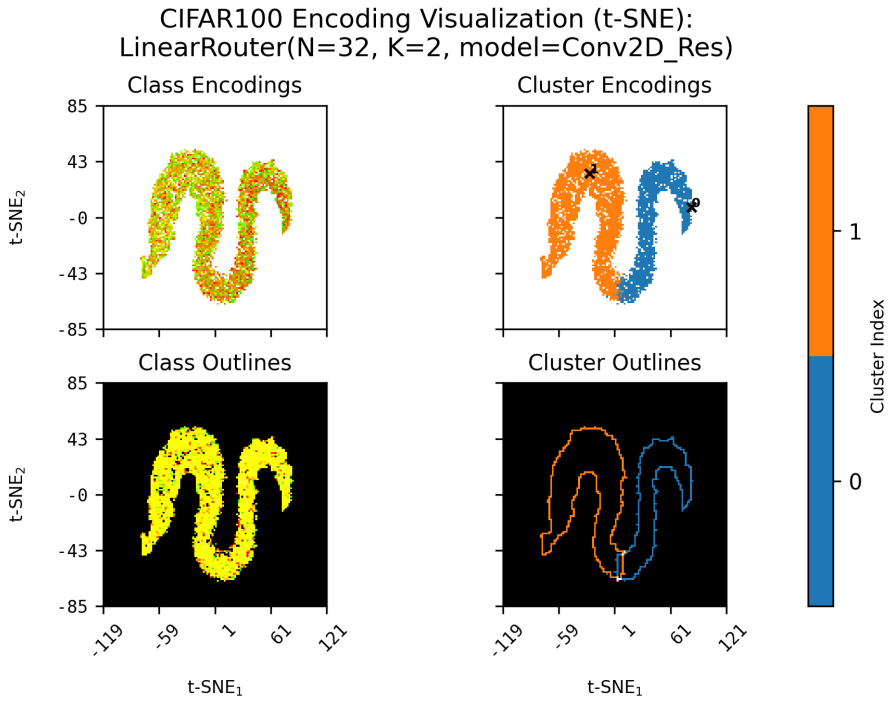


Figure 6.26: CIFAR-100 latent encodings using a 2D t-SNE projection. Class colors represent the 20 CIFAR-100 superclass groups, while the expert-colored panels show the corresponding top-1 router assignments.

The t-SNE projection in Figure 6.26 forms two curved structures connected near the center of the plot. The superclass colors are distributed throughout both structures, with only limited local concentrations visible at this scale. The expert-colored panels show a clearer division: expert 1 is associated more often with the left structure and expert 0 with the right structure, with an overlap region near their connection.

The two embedded experts are placed near different portions of the projected structures. Their apparent locations should be interpreted cautiously because t-SNE emphasizes local

neighborhoods and can alter global distances. The figure is more useful for showing the continuity and overlap of the two routed regions than for measuring the distance between the embedded experts.

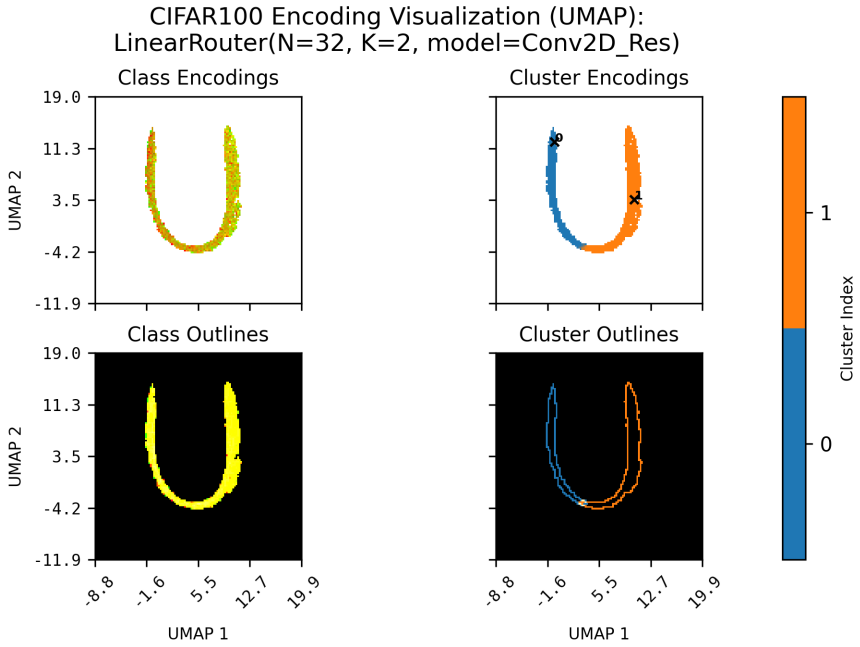


Figure 6.27: CIFAR-100 latent encodings using a 2D UMAP projection. Class colors represent the 20 CIFAR-100 superclass groups, while the expert-colored panels show the corresponding top-1 router assignments.

Figure 6.27 maps the encodings onto a compact U-shaped structure. The superclass-colored view demonstrates extensive overlap along the curve, while the expert-colored view divides the two arms more clearly. Expert 0 is associated mainly with the left arm and expert 1 with the right arm, with both assignments meeting near the lower bend. This organization resembles the broad division in the SVD projection, although the nonlinear UMAP transformation presents it as a curved manifold.

Across the SVD, t-SNE, and UMAP views, the superclass labels do not form distinct projected clusters, but the expert assignments produce coherent regions. This does not imply that the router has discovered the CIFAR-100 superclass hierarchy. Instead, the router appears to organize the representation according to features useful for directing samples between two experts. The confusion-matrix diagonals explain that this organization is as-

sociated with measurable differences in superclass performance, even though the routing regions do not correspond one-to-one with the displayed semantic groups.

Compared with the CIFAR-10 example, the CIFAR-100 class-colored projections provide less direct evidence of label-based grouping, which is plausible given the larger number of fine-grained categories. The expert-colored projections are easier to interpret because they reduce the visualization to the two decisions made by the router. This, in turn, further impedes interpretability, as no clear associativity can be made between the latent space and the experts. As with the CIFAR-10 results, the supervised mixture objective may encourage the encoder to emphasize distinctions that support expert selection.

6.2.4 Summary

The latent visualization experiments identify that router interpretability is highly subject to variance and is difficult to predict without other regularization encouraging more structured formulation. While there are some insights to gain into routing behaviors, they are partly redundant with a simple confusion matrix. Furthermore, one of the challenges remains that the measure of space is inherently unbounded. There is a soft correlation between the routing strength and where an expert is encoded with respect to all experts to whom that strength applies, but that is the extent. Nothing about how close/far a sample is to/from an embedded expert, independent of all other experts, is inherently meaningful since those embeddings can be moved arbitrary distances.

Notably, while the MNIST encodings had weaker class- and geometric-based structure than the autoencoder baselines, the CIFAR-10 and CIFAR-100 encodings were significantly stronger (albeit still noisy) and used significantly smaller encoders and latent dimensionality.

About the first hypothesis presented at the start of this chapter, these experiments partially support it: router latent spaces can encode meaningful subproblem structure, but this behavior is conditional on task complexity and router capacity. The level of interpretability provided is noteworthy, but not substantial depending on the complexity of the task and

quality of the encoder.

For future work, it may be interesting to use such feedforward encoding models pretrained by routing approaches as a basis for more interpretable autoencoders. Similarly, developing routing regularization to improve consistency may be an effective approach for more predictable behaviors. Additionally, identifying more advanced approaches to CIFAR-100 latent interpretability could be beneficial, such as further breakdowns along the supercategory types.

6.3 Routing Attention

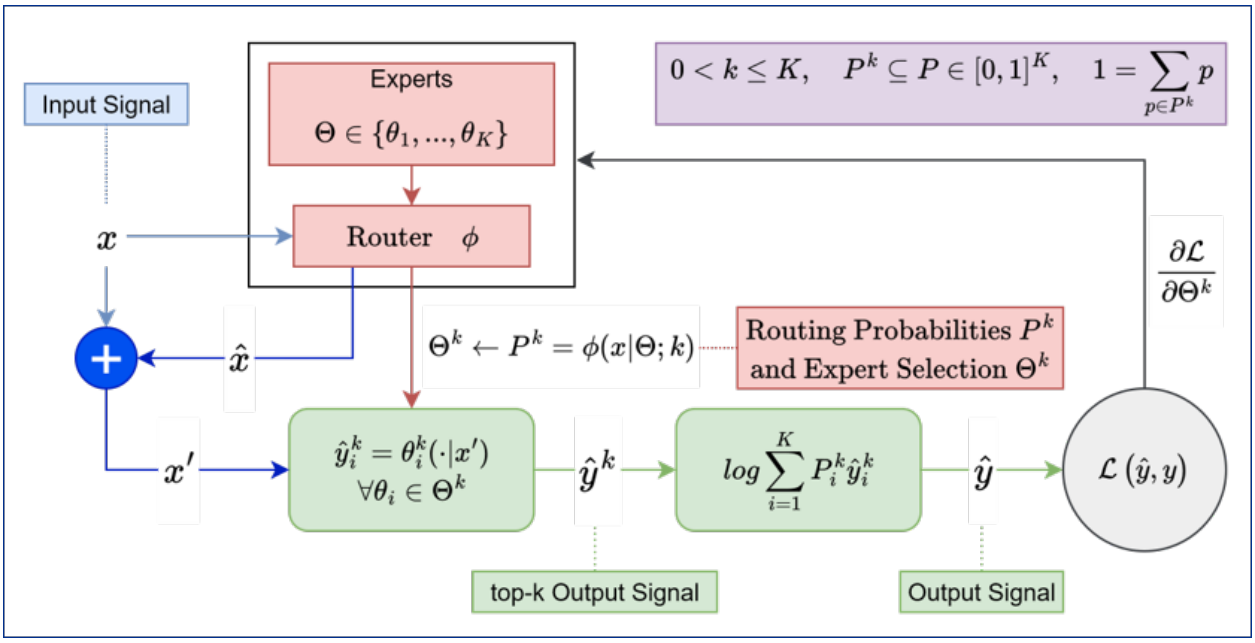


Figure 6.28: MoE architecture with encoder-decoder routing. The encoder-decoder router produces both routing parameters and an auxiliary decoded signal, allowing the routing process to be inspected visually.

The second interpretability approach adds an encoder-decoder pathway to the router. The latent visualizations in Section 6.2 describe the organization of the router representation, but they do not directly expose the spatial signal produced by the router pathway. In the attention-routing models, the router encoder produces a latent representation $\mathbf{z}_b$.

$$\mathbf{z}_b = R_{\text{enc}}(\mathbf{x}_b) \tag{6.8}$$

The paired decoder then produces a spatial output $\mathbf{r}_b$.

$$\mathbf{r}_b = R_{\text{dec}}(\mathbf{z}_b) \tag{6.9}$$

For the U-Net-based encoders, the decoder also receives the architecture's skip-connected features, which are omitted from the notation.

The decoded output is coupled to the classifier through the `signal_modifier`. The `add` setting modifies the expert input as follows.

$$\mathbf{x}'_b = \mathbf{x}_b + \mathbf{r}_b \tag{6.10}$$

The `concat` setting instead forms the expert input as follows.

$$\mathbf{x}'_b = [\mathbf{x}_b; \mathbf{r}_b] \tag{6.11}$$

The concatenation occurs along the channel dimension. Concatenation preserves the original image as a distinct part of the expert input, while addition directly changes its values. Because the decoded output participates in the expert computation, it receives classification gradients even when no explicit reconstruction penalty is applied.

The optimized loss is the marginal negative log likelihood used by the MoE baselines, with an optional normalized reconstruction term.

$$\mathcal{L} = \mathcal{L}_{\text{mNLL}} + \lambda_{\text{rl}} \frac{\mathcal{L}_{\text{recon}}}{|\mathbf{x}_b|} \tag{6.12}$$

The coefficient λ_{rl} is stored as `lambda_reconstruction_loss`. All experiments use `RBFRouter` with `tK2t1` optimization, and $K \in \{2, 3, 4, 5\}$. Since `tK2t1` finishes with one active expert, `t1_acc1` equals `acc1` in these results. Note the RBF router is used since it can approximate the Linear and Centroid, and is the most flexible of the approaches defined. Since no approach stands out in the baseline experiments, the RBF is thus used as the default representative.

MNIST compares VAE and UNet router encoders using both `add` and `concat`, with $\lambda_{\text{rl}} \in \{0, 10^{-3}\}$. CIFAR-10 and CIFAR-100 evaluate the UNet encoder using `concat` without an explicit reconstruction penalty. The router and expert dimensions follow the corresponding MoE baseline configurations described earlier.

6.3.1 MNIST Attention-Routing Results

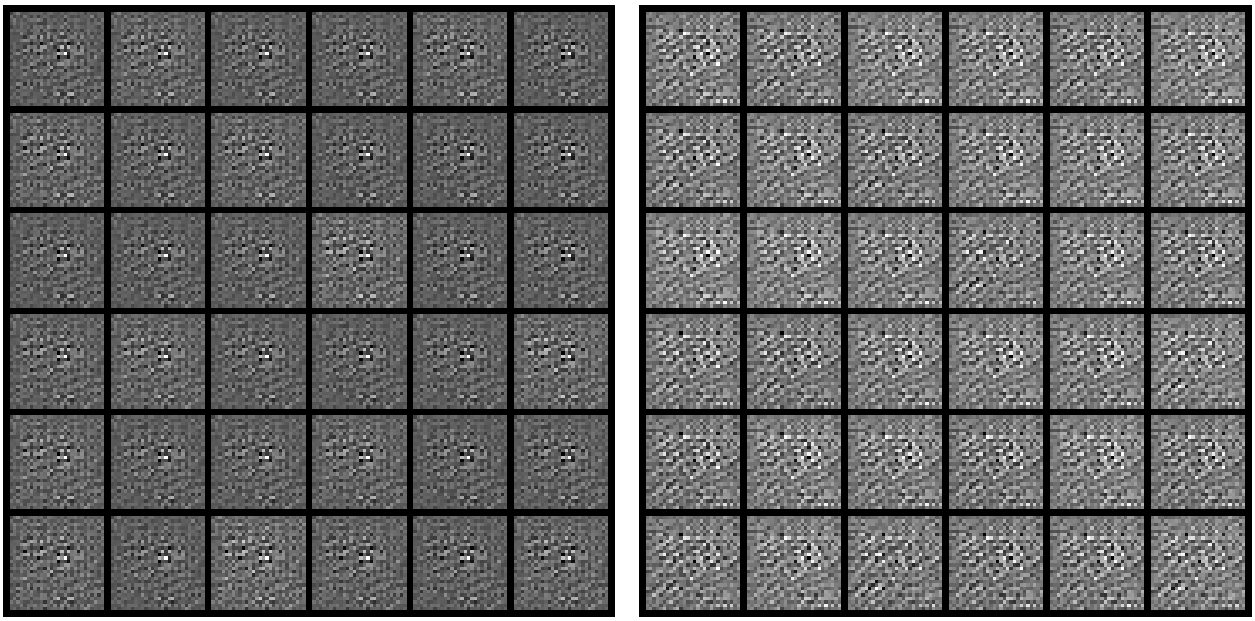
The MNIST experiments use router encoders with $(P_r, H_r, D_r, N) = (2, 2, 1, 2)$ and `Conv2D_FFNN` experts with $(P_e, H_e, D_e) = (2, 3, 1)$. The batch size is 1024, and expert dropout is 0.1. Tables 6.1 and 6.2 report every tested value of K. Horizontal dividers separate the modifier and reconstruction-weight setups.

Table 6.1: MNIST attention-routing results using a VAE router encoder. Each row reports one run. Params includes the router and all experts; T1 Params reports the active learnable parameter count for top-1 inference.

Modifier	λ_{rl}	K	Params	T1 Params	Loss	Acc1	Recon.	Entropy	Switch Bias
`add`	0	2	2,004	1,468	0.2289	0.9281	1.4314	0.2440	0.0094
`add`	0	3	2,543	1,471	0.1695	0.9488	8.3119	0.1199	0.9472
`add`	0	4	3,082	1,474	0.1713	0.9498	2.9491	0.1181	0.4094
`add`	0	5	3,621	1,477	0.1800	0.9445	3.2473	0.0760	0.4181
`add`	10^{-3}	2	2,004	1,468	0.2158	0.9317	1.1009	0.2461	0.0131
`add`	10^{-3}	3	2,543	1,471	0.2070	0.9389	1.9967	0.2106	0.6568
`add`	10^{-3}	4	3,082	1,474	0.1743	0.9500	1.4152	0.0897	0.4635
`add`	10^{-3}	5	3,621	1,477	0.1892	0.9431	1.2629	0.0749	0.4162
`concat`	0	2	2,040	1,486	0.1582	0.9519	0.7157	0.2535	0.8475
`concat`	0	3	2,597	1,489	0.1638	0.9502	0.6144	0.0806	0.6601
`concat`	0	4	3,154	1,492	0.1651	0.9506	0.6399	0.1505	0.3227
`concat`	0	5	3,711	1,495	0.1869	0.9445	0.7220	0.1057	0.3687
`concat`	10^{-3}	2	2,040	1,486	0.1740	0.9483	0.6911	0.2995	0.7211
`concat`	10^{-3}	3	2,597	1,489	0.1698	0.9494	0.5829	0.1098	0.5398
`concat`	10^{-3}	4	3,154	1,492	0.1803	0.9463	0.6579	0.1291	0.3381
`concat`	10^{-3}	5	3,711	1,495	0.1767	0.9452	0.7003	0.1535	0.3871

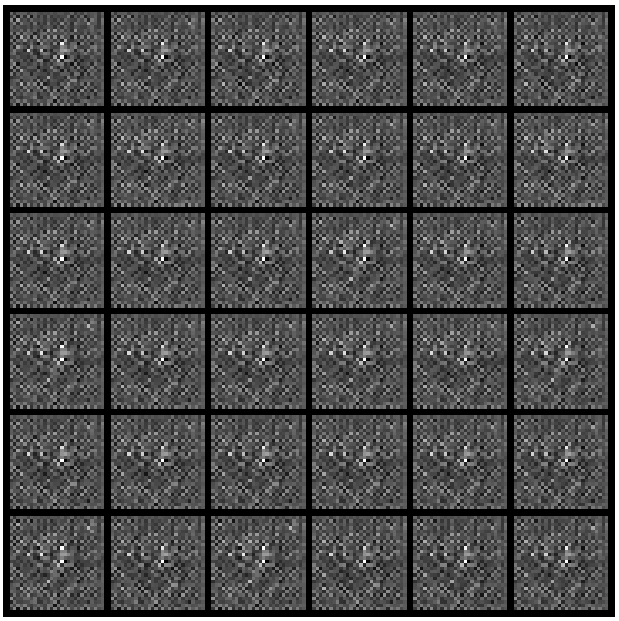
Figure 6.29 details representative decoded outputs from the VAE router for the four MNIST setups formed by crossing the `add` and `concat` modifiers with $\lambda_{\text{rl}} \in \{0, 10^{-3}\}$. In all four cases, the decoder does not produce clean digit reconstructions. Instead, the outputs are dominated by coarse high-frequency structure with only faint digit-like organization. Adding the reconstruction term appears to regularize the output and make its spatial organization more coherent, especially for the `concat` examples. However, it may lack enough parameters to model this relationship well.

Figure 6.30 details a clearer effect from the signal modifier than from the reconstruction-loss term. Under `add`, the decoded outputs are dominated by weak digit traces and high-

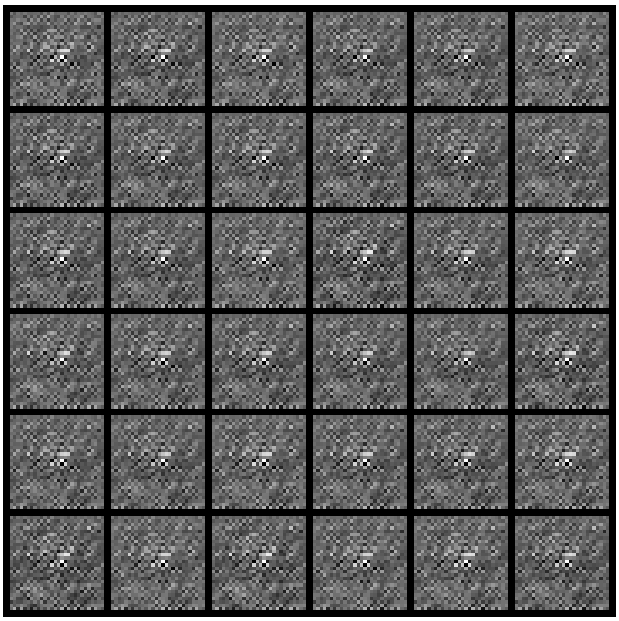


(a) `add`, $\lambda_{\text{rl}} = 0$

(b) `add`, $\lambda_{\text{rl}} = 10^{-3}$

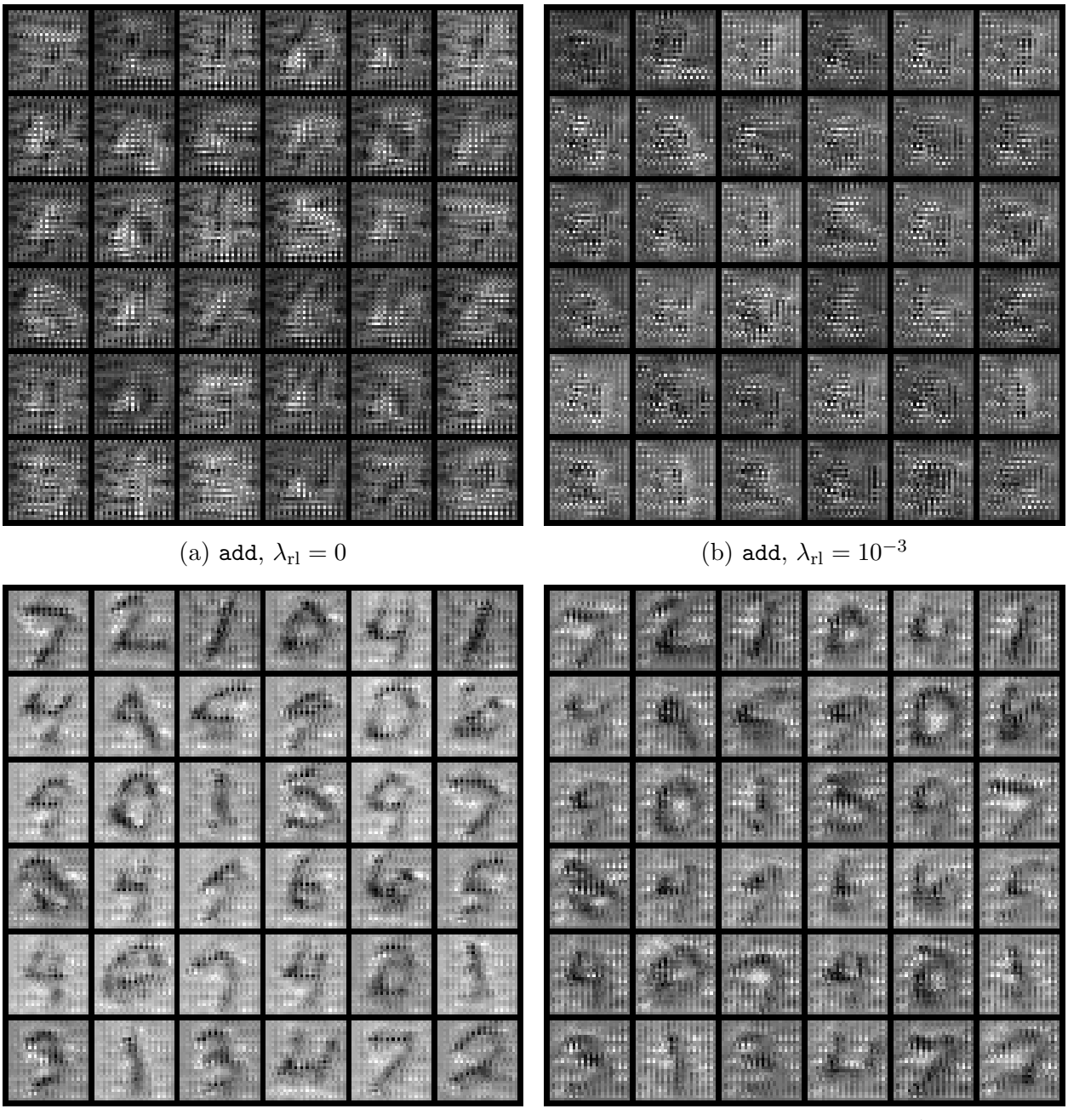


(c) `concat`, $\lambda_{\text{rl}} = 0$



(d) `concat`, $\lambda_{\text{rl}} = 10^{-3}$

Figure 6.29: Example decoded outputs from the VAE router on MNIST for $K = 2$. The four panels compare the `add` and `concat` signal modifiers with and without the reconstruction-loss term. Each small tile details one decoded output produced by the router decoder. Across these examples, the decoded signals are generally coarse and texture-like. Enabling $\lambda_{\text{rl}} = 10^{-3}$ tends to make the outputs more spatially organized, while the `concat` examples appear somewhat more structured than the `add` examples.



(a) `add`, $\lambda_{\text{rl}} = 0$

(b) `add`, $\lambda_{\text{rl}} = 10^{-3}$

(c) `concat`, $\lambda_{\text{rl}} = 0$

(d) `concat`, $\lambda_{\text{rl}} = 10^{-3}$

Figure 6.30: Example decoded outputs from the UNet router on MNIST with $K = 2$. The panels compare the `add` and `concat` signal modifiers with $\lambda_{\text{rl}} \in \{0, 10^{-3}\}$. The `concat` outputs contain more recognizable digit structure than the corresponding `add` outputs. Enabling the reconstruction term changes the contrast and smoothness of the decoded signals, but its effect appears smaller than the difference between the two signal modifiers.

Table 6.2: MNIST attention-routing results using a UNet router encoder. Each row reports one run. Horizontal dividers separate the `add` and `concat` setups and the two reconstruction weights.

Modifier	λ_{rl}	K	Params	T1 Params	Loss	T1 Acc1	Recon.	Entropy	Switch Bias
`add`	0	2	2,101	1,565	0.1430	0.9577	0.0156	0.2740	0.0473
`add`	0	3	2,640	1,568	0.1384	0.9584	0.0127	0.1638	0.4686
`add`	0	4	3,179	1,571	0.1612	0.9542	0.0094	0.3265	0.5061
`add`	0	5	3,718	1,574	0.1772	0.9476	0.0032	0.2966	0.2635
`add`	10^{-3}	2	2,101	1,565	0.1296	0.9615	0.0109	0.3128	0.1913
`add`	10^{-3}	3	2,640	1,568	0.1665	0.9524	0.0069	0.1931	0.5384
`add`	10^{-3}	4	3,179	1,571	0.1366	0.9600	0.0054	0.1809	0.8645
`add`	10^{-3}	5	3,718	1,574	0.1526	0.9548	0.0061	0.2409	0.3287
`concat`	0	2	2,137	1,583	0.1391	0.9584	0.0026	0.1627	0.4973
`concat`	0	3	2,694	1,586	0.1030	0.9696	0.0010	0.1710	0.5696
`concat`	0	4	3,251	1,589	0.1175	0.9646	0.0013	0.2020	0.5314
`concat`	0	5	3,808	1,592	0.1404	0.9581	0.0014	0.1202	0.6688
`concat`	10^{-3}	2	2,137	1,583	0.1388	0.9603	0.0023	0.1910	0.5002
`concat`	10^{-3}	3	2,694	1,586	0.1277	0.9622	0.0011	0.1857	0.6000
`concat`	10^{-3}	4	3,251	1,589	0.1343	0.9600	0.0012	0.1235	0.6309
`concat`	10^{-3}	5	3,808	1,592	0.1122	0.9650	0.0013	0.1224	0.4966

frequency texture. This is especially visible with $\lambda_{\text{rl}} = 10^{-3}$, where much of the decoded signal has low contrast. The `concat` configurations produce more recognizable digit shapes across the sample grid. Their decoded signals emphasize features corresponding to their respective classes, such as contours and strokes. Enabling the reconstruction term changes the intensity and smoothness of the outputs without producing a comparably large change in their overall content.

Across the matched MNIST rows, `concat` produces higher top-1 accuracy than `add` in 13 of the 16 comparisons formed by holding encoder type, reconstruction weight, and K constant. Its descriptive accuracy is approximately 0.65 percentage points higher, and its loss is lower in 13 of the 16 comparisons. Concatenation adds 18 parameters to the active path because the experts receive the decoded signal through additional input channels. The effect is visible for both router encoders, although the size of the difference changes across K. This is intuitive because concat does not mutate the input signal, whereas add does. Thus, concat maximizes the amount of information given to the expert while the add distorts it with the learned attention. Additionally, the observed accuracy gains of the concat approach

are significant relative to how few parameters it costs over the add.

The reconstruction term has a smaller association with classification. Enabling $\lambda_{\text{rl}} = 10^{-3}$ increases top-1 accuracy in 8 of the 16 matched comparisons and decreases it in the other 8. Across all retained MNIST rows, the descriptive accuracy changes by less than 0.05 percentage points. The reconstruction diagnostic decreases in 13 of the 16 comparisons, with the largest reductions occurring for the VAE encoder. The auxiliary term therefore does not reliably impact the classification accuracy. This implies that the strength of which original features can be emphasized in the decoder can be safely tuned with minimal performance reduction.

The parameter comparison with the standard MoE baselines is less favorable than the accuracy comparison alone. The matched feedforward RBF baseline uses approximately 694–703 active parameters and has a descriptive top-1 accuracy of 90.26% across K. The VAE attention top-1 inference paths use approximately 1,468–1,495 active parameters and reach 94.51% descriptively, while the UNet paths use approximately 1,565–1,592 active parameters and reach 95.90%. The encoder-decoder paths improve accuracy, but they require a little more than twice the active parameter count. This is fairly in line with the exponential number of parameters required to grow accuracy observed in the earlier baselines. Note that a comparable classifier baseline is $\{P = 2, H = 3, D = 2\}$ (1320 params) with an average top-1 accuracy of 92.04%, which the attention-based routing approach is still fairly competitive with.

While the performance looks like an improvement, it is likely simply owing to the addition of parameters. This, however, provides two points of information. The first is that information can likely be passed from the router via the decoder to shape the expert's decisions better. This is because the only added parameters here are the decoder versus the MoE baselines, but performance is improved. The second is that by expanding an encoder-only router and/or the expert to an equal inference size, the expert either implicitly understands what the router discovered via the expert selection processor, or there is another axis of

improvement that the expert/router can independently grow alongside. In the latter case, this indicates a potential scalability opportunity. However, in the former, this reveals an interesting characteristic behind statistical priors and neural networks that may be worth exploring in the future.

Routing balance also varies across the modifier and reconstruction setups. Several high-accuracy rows have large switch-bias values, while other rows use the experts more evenly. Neither `concat` nor the reconstruction term consistently lowers switch bias. The decoded-signal design and the distribution of expert use therefore appear to be partly separate effects.

6.3.2 CIFAR-10 Attention-Routing Results

The CIFAR-10 experiments use router encoders with $(P_r, H_r, D_r, N) = (4, 3, 2, 8)$ and `Conv2D_FFNN` experts with $(P_e, H_e, D_e) = (8, 3, 4)$. The UNet encoder uses `concat`, $\lambda_{\text{rl}} = 0$, a batch size of 512, expert dropout of 0.2, random cropping, and horizontal flipping.

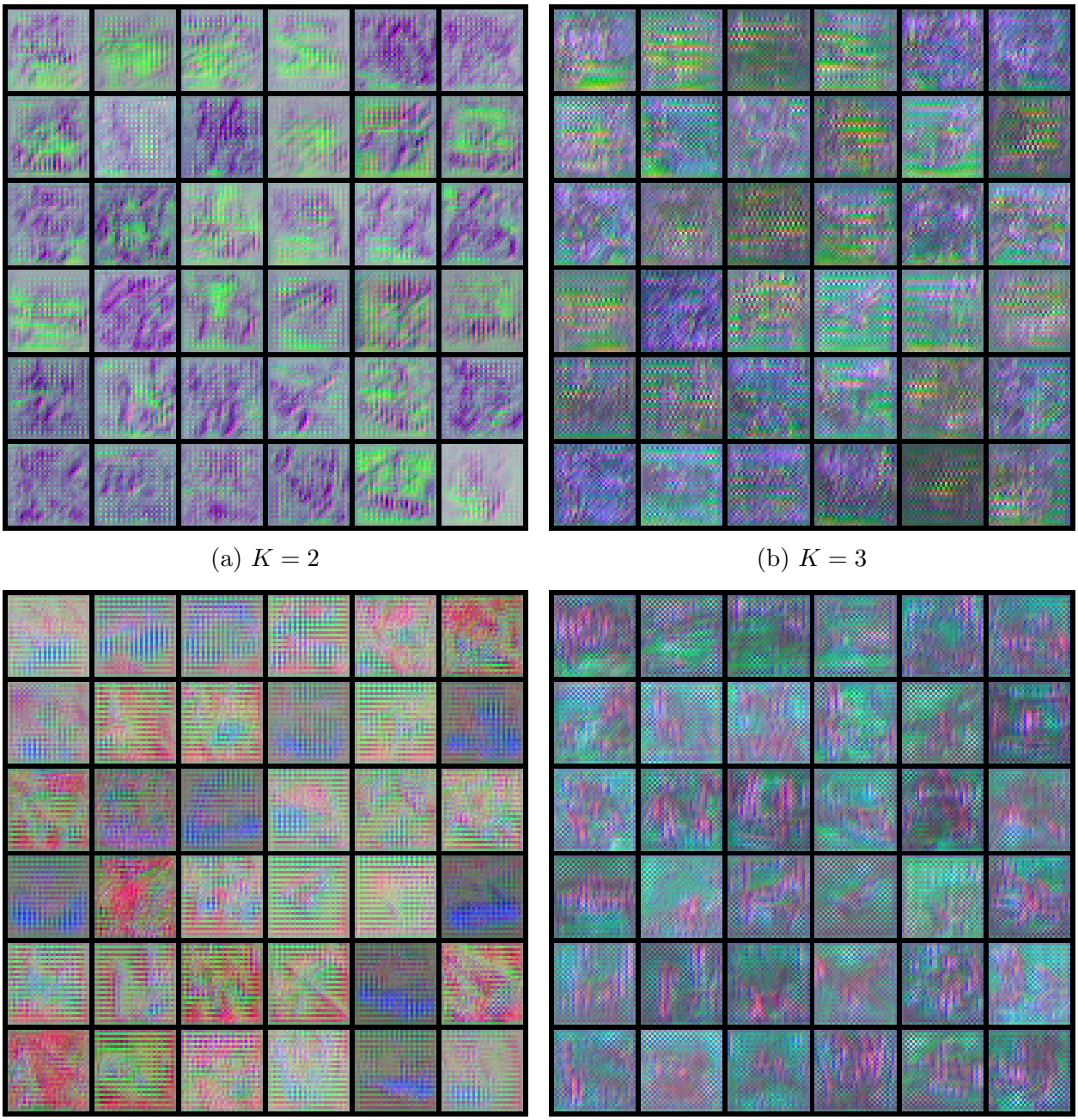
Table 6.3: CIFAR-10 attention-routing test results using the UNet router encoder. Each row reports one run.

Encoder	K	Params	T1 Params	Loss	T1 Acc1	Recon.	Entropy	Switch Bias
UNet	2	97,633	53,783	0.7959	0.7245	0.0004	0.5056	0.0541
UNet	3	141k	53,792	0.7709	0.7366	0.0004	0.4203	0.0078
UNet	4	185k	53,801	0.8130	0.7262	0.0005	0.4298	0.0457
UNet	5	229k	53,810	0.7541	0.7502	0.0004	0.4236	0.0483

Figure 6.31 details representative decoded outputs from the CIFAR-10 UNet router for the four tested expert counts. In contrast with the MNIST examples, these decoded signals contain rich color variation and strong local texture. The broad visual character of the outputs is fairly similar across $K = 2$ through $K = 5$, so increasing the number of experts does not appear to produce a simple qualitative shift in the decoded signal. This is consistent with the quantitative results, where changes in K alter classification performance, but do not by themselves indicate that the decoded outputs become more reconstruction-like or easier to interpret visually.

The CIFAR-10 experiment contains only the `concat` modifier with $\lambda_{\text{rl}} = 0$, so it does not provide a within-dataset modifier or reconstruction-term comparison. This is due to both how trivial it would be for a UNet to reconstruct the input if significantly encouraged to do so, and the lack of impact from applying the term in the MNIST experimentation. Across the four values of K, the UNet attention model reaches a descriptive top-1 accuracy of 73.44%, compared with 75.45% for the matched feedforward RBF MoE baseline. The attention model is lower at three values of K and higher at one.

The active UNet path uses approximately 53.8 thousand parameters, compared with approximately 48.6 thousand for the matched baseline, an increase of about 10.6%. Since the



(a) $K = 2$ (b) $K = 3$

(c) $K = 4$ (d) $K = 5$

Figure 6.31: Example decoded outputs from the CIFAR-10 UNet router using the `concat` modifier and $\lambda_{\text{rl}} = 0$, shown for $K \in \{2, 3, 4, 5\}$. Each small tile is a decoded output produced by the router decoder. Across all four expert counts, the outputs are highly textured and colorful, with visible stripe-like and edge-like structure.

attention model also has lower descriptive accuracy, it does not improve active-parameter efficiency over the matched MoE baseline in this experiment. The low reconstruction diagnostic indicates that the UNet decoder output is close to the input under the recorded metric, but this does not translate into a general accuracy increase.

Switch-bias values are low across the four rows, indicating that an obvious collapse does not explain the lack of improvement to one expert. The variation across K instead suggests that the added encoder-decoder pathway does not consistently provide a more useful routing signal for the selected CIFAR-10 expert architecture.

These results more firmly corroborate the insight in the MNIST experiments where the experts understand what the router learns, without requiring the router to pass it to them via the input signal. However, the routers are comparatively small, so it may be a capacity issue that would come at the cost of the total inference path length.

6.3.3 CIFAR-100 Attention-Routing Results

The CIFAR-100 experiments use residual router encoders with $(P_r, H_r, D_r, N) = (16, 3, 4, 32)$ and `Conv2D_Res` experts with $(P_e, H_e, D_e) = (32, 3, 8)$. The UNet encoder uses `concat`, $\lambda_{\text{rl}} = 0$, a batch size of 256, expert dropout of 0.5, random cropping, and horizontal flipping.

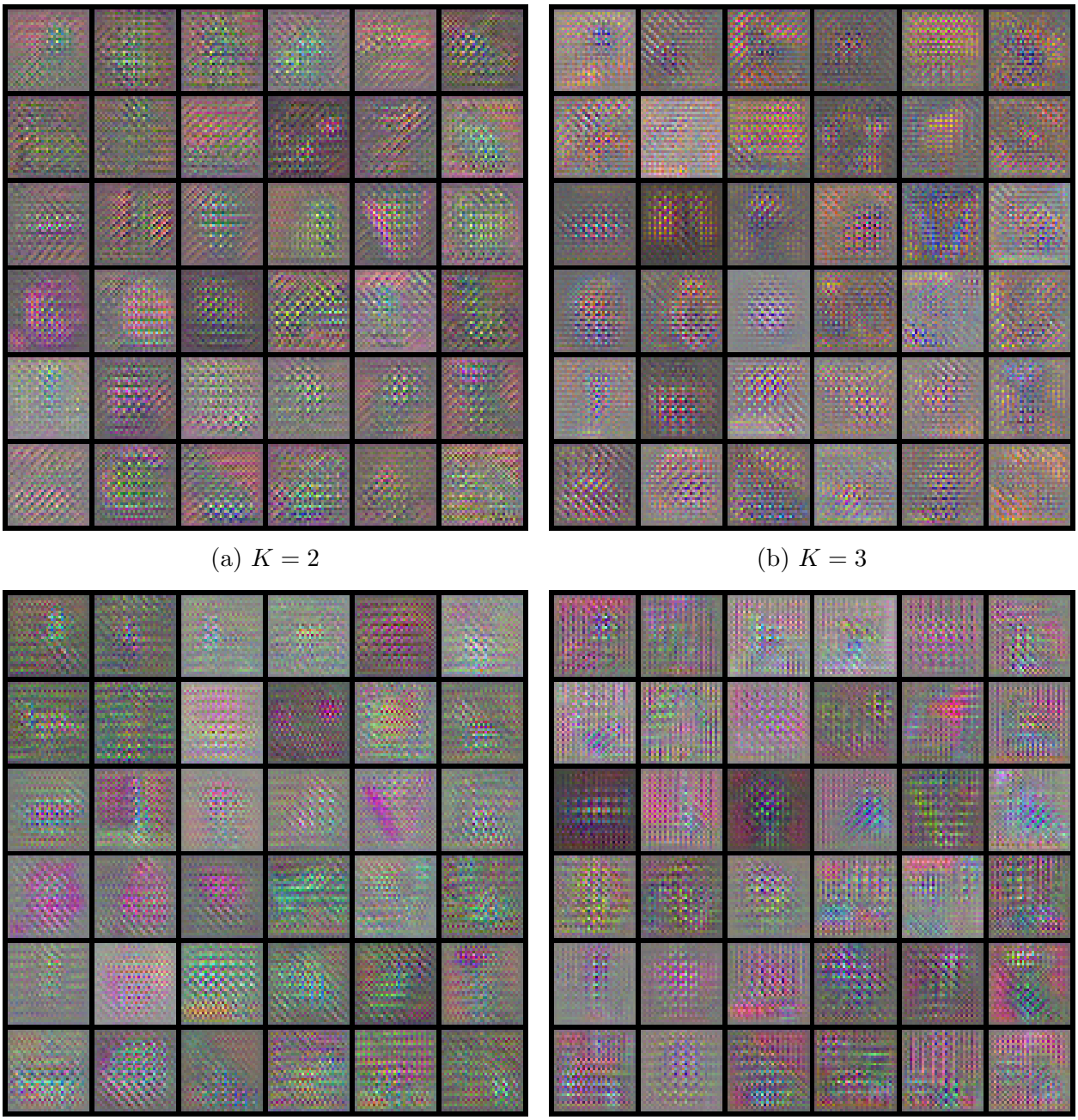
Table 6.4: CIFAR-100 attention-routing test results using the UNet router encoder. Each row reports one run.

Encoder	K	Params	T1 Params	Loss	T1 Acc1	T Acc5	Recon.	Entropy	Switch Bias
UNet	2	3,240k	1,755k	1.1734	0.6747	0.9116	0.0006	0.0889	0.9534
UNet	3	4,724k	1,755k	0.9345	0.7442	0.9382	0.0006	0.0684	0.8960
UNet	4	6,209k	1,755k	1.0105	0.7182	0.9307	0.0005	0.0647	0.8570
UNet	5	7,694k	1,755k	0.9754	0.7353	0.9338	0.0006	0.0894	0.8553

Figure 6.32 details representative decoded outputs from the CIFAR-100 UNet router for the four tested expert counts. As in the CIFAR-10 examples, the decoded signals are dominated by colorful local texture and directional patterning, rather than clean object-level structure. Many tiles contain repeated diagonal bands, checkerboard-like responses, and localized color contrasts, which suggests that the decoder is emphasizing visual components linked to classification instead of attempting to reconstruct the input image in an easily interpretable form. The overall appearance is fairly similar across $K = 2$ through $K = 5$, although the exact balance of color and contrast varies from panel to panel. Notably, resolution/attention improves as K increases, indicating that there may be a correlation between forcing the router to more strongly structure the space (accommodating more experts) and providing some inherent reconstruction/interpretability.

The CIFAR-100 experiment also uses only `concat` with $\lambda_{\text{rl}} = 0$. Across K, the UNet attention model reaches a descriptive top-1 accuracy of 71.81%, compared with 73.76% for the matched residual RBF MoE baseline. The attention result is lower at three values of K and higher at one.

The active attention path uses approximately 1.755 million parameters, while the matched baseline uses approximately 1.661 million, an increase of about 5.7%. As with CIFAR-10, the



(a) $K = 2$ (b) $K = 3$

(c) $K = 4$ (d) $K = 5$

Figure 6.32: Example decoded outputs from the CIFAR-100 UNet router using the `concat` modifier and $\lambda_{\text{rl}} = 0$, shown for $K \in \{2, 3, 4, 5\}$. Each small tile is a decoded output produced by the router decoder. Across all four expert counts, the outputs are highly textured and color-rich, with visible stripe-like, checkerboard-like, and edge-oriented structure.

added parameters do not produce a general accuracy increase, so the attention architecture is less parameter efficient across the tested rows. The comparison with the standalone residual expert is more favorable in absolute accuracy, but the MoE baseline already provides that comparison using a smaller active routing path.

The routing measurements also indicate uneven expert use. Switch bias ranges from 0.8553 to 0.9534, and normalized entropy is below 0.09 in every row. This concentration limits how much conditional capacity is used as K increases and complicates the interpretation of the decoded signal as evidence of broad expert specialization.

6.3.4 Summary

The MNIST comparison indicates that concatenation is generally more useful for classification than addition in the tested setups. It preserves the original image channels and supplies the decoded output separately, but it also increases the expert-input width. The experiment therefore combines two changes: how the signal is presented and how many input channels the expert receives.

The decoded signal is behaviorally connected to prediction because it is supplied to the experts. When visualized, the decoded signal highlights only portions of the image typically corresponding to the object. Though resolution is low, this is likely due to the inputs also being a lower resolution. Additionally, to keep the top-1 inference path costs down, smaller router models are used where upscaling them may further enhance interpretability.

The modifier comparison on MNIST provides the clearest gains from the new architecture. Concatenation usually improves top-1 accuracy and loss relative to addition, with a small increase in active parameters. The reconstruction term lowers the reconstruction diagnostic more consistently than it changes classification accuracy, so its main observed effect is on the decoded representation.

Compared with the matched MoE baselines, the attention models do not provide a general parameter-efficiency improvement. MNIST gains accuracy, but the active path is more than

twice as large. CIFAR-10 and CIFAR-100 use modestly larger active paths and have lower descriptive top-1 accuracy across the tested expert counts. The encoder-decoder router may still provide useful spatial outputs for interpretation, but those outputs do not consistently improve sparse classification efficiency.

As a result, this hypothesis is accepted. While the performance gains are still contested, the results demonstrate clear capabilities to emphasize important features.

6.3.5 Latent Encodings

Notably, using an autoencoder as a router has the added benefit of training such a model. While not strictly optimized for reconstruction, routing could be considered an adjacent objective function that may make a good basis for one.

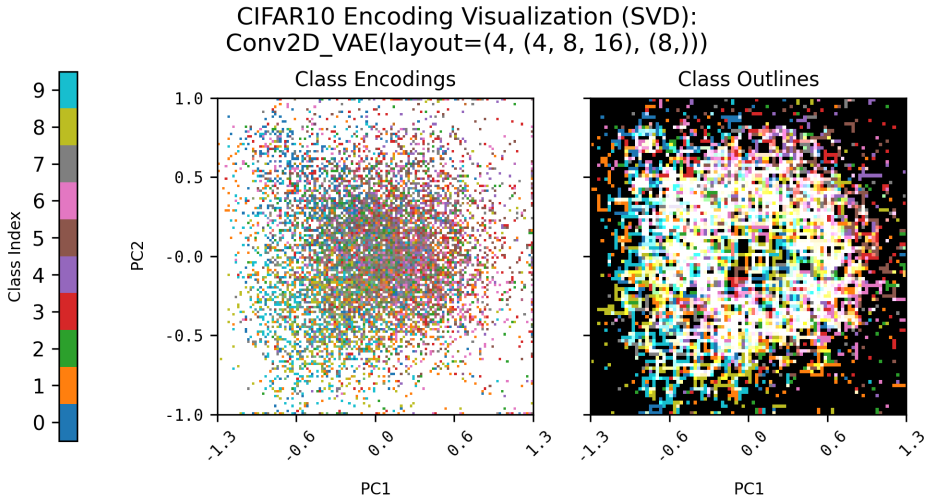


Figure 6.33: Example CIFAR-10 autoencoder encoding (SVD2D projection). Trained using a $\{P = 4, H = 3, D = 2, N = 8\}$ Conv2D_VAE autoencoder.

Consider Figures 6.33 and 6.34, which detail a CIFAR-10 autoencoder's encodings and decodings. Notably, neither is particularly well structured. The decodings are broad silhouettes with no color coordinate or fine features, and the latent space appears like random noise.

Then, consider Figure 6.35 in contrast, which is the projection of an autoencoder-router used in a MoE model. Furthermore, the decodings were given earlier in Figure 6.31d, which

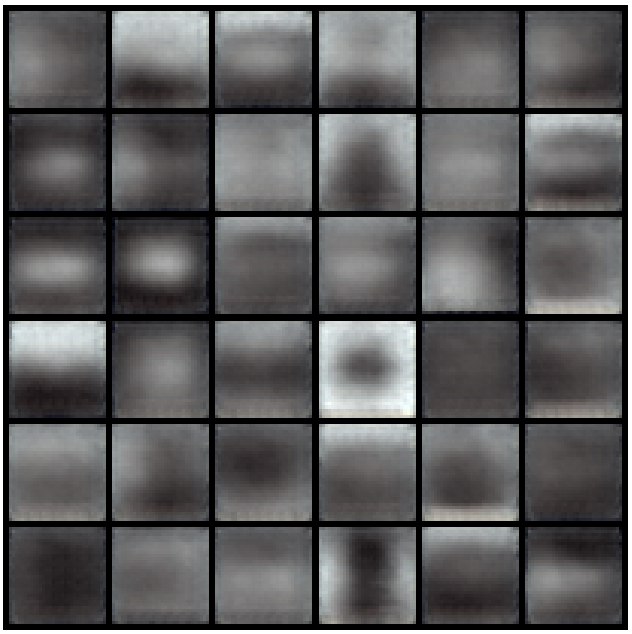


Figure 6.34: Example CIFAR-10 autoencoder decodings. Trained using a $\{P = 4, H = 3, D = 2, N = 8\}$ Conv2D_VAE autoencoder.

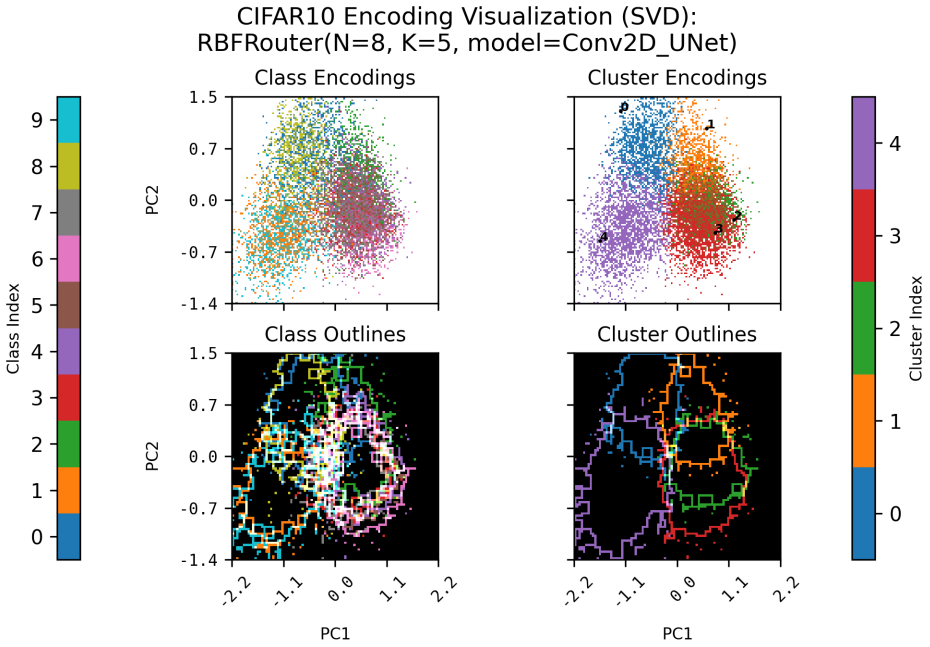


Figure 6.35: Example CIFAR-10 MoE attention-based encoder projections (SVD2D). Trained using a RBF router configured as $\{P = 4, H = 3, D = 2, K = 5, N = 8\}$ and Conv2D experts configured as $\{P = 8, H = 3, D = 2\}$.

provide significantly stronger matching contours and textures than the model trained as a pure autoencoder.

Thus, there may be future potential for leveraging such models as an auxiliary training mechanism.

6.4 Conclusions

The interpretability experiments produce mixed results. For latent interpretability, the results show that router encodings can provide meaningful geometric information, but only under certain conditions. In simple problem spaces such as MNIST, the router may learn a compressed representation that is sufficient for classification but not visually rich enough to support strong interpretability. This can result in flattened latent projections and expert starvation. In more complex problem spaces such as CIFAR-10, the router is more likely to learn a richer latent representation, producing clearer cluster structure and more meaningful expert-region separation.

These findings suggest that latent interpretability depends strongly on problem complexity, router capacity, and expert utilization. Higher-capacity routers can provide more resolution in the latent space, but this comes with an inference-cost tradeoff. For extremely simple datasets, increasing router capacity may be inefficient. For more complex datasets, the additional router capacity may be justified because the router must learn more detailed features to partition the problem space effectively.

The attention-based routing experiments show stronger promise. Across both simple and complex problem spaces, the encoder-decoder router produces visual signals that correspond to meaningful input characteristics. The UNet-based versions are especially effective because they preserve spatial resolution through skip connections. These results imply that attention-based or reconstruction-assisted routing can provide interpretable information that is more directly connected to the input image than latent-space projections alone.

The first hypothesis is weakly accepted. The router latent space can encode meaningful subproblem structure, but this behavior is conditional and depends on the interaction between dataset complexity, router capacity, and latent structure. Furthermore, while it can be used to interpret the type of features routing decisions are split along, the scale of distances is arbitrary, as evidenced by expert embeddings being inconsistently positioned with respect to their allocated clusters.

The second hypothesis is accepted. Attention-based routing provides a more consistent interpretability mechanism across parameter ranges and problem complexities, while also integrating the interpretable signal directly into the inference pathway. However, after considering upscaled baseline comparisons, no performance benefits are directly observed.

Chapter 7

Transferring Latent Learning

7.1 Overview

The hypothesis tested in this chapter is given as follows: pre-training a router can reduce overall training cost and enable efficient transfer learning by providing a stable latent reference frame for expert specialization.

This hypothesis follows from the idea that the router is an optimization bottleneck for an MoE system. Since all experts learn their specialization after the subproblem decision boundaries solidify, it follows that if the router is predefined—or at least more optimally initialized—it can reduce the computational resources required to train the model. Thus, instead of training a router and experts together from random initialization, the router can first be trained on some auxiliary objective function. The adapted latent space can then immediately more strongly inform subproblem decision boundaries, enabling the experts to learn to specialize and converge faster.

This approach is motivated by transfer learning being an effective approach for accelerating training processes (Taylor et al., 2014). Conceptually, a VAE learns how to encode inputs into a re-parameterized latent space that groups inputs (images) by their features for a probabilistic reconstruction. The encoder component of the VAE is explored as a pretrained router, which tangentially explores whether such an encoding objective is comparable to a routing process.

This chapter evaluates whether such transferred latent learning improves training effi-

ciency, routing stability, or final classification performance. The experiments compare three training strategies:

1. a baseline MoE trained end-to-end,
2. an MoE with a pre-trained router that is fixed during expert training, and
3. an MoE with a pre-trained router that is trainable during expert training.

The first case provides the reference behavior. The second case tests whether a fixed latent reference frame is sufficient for expert specialization. The third case tests whether pretraining gives the router a meaningful initialization while still allowing the routing space to adapt to the classification objective. Two variants are also tested, where the pre-trained model can either be an autoencoder that learns a reconstruction objective or a small classifier that learns a classification objective. In the latter, the logit space is interpreted as a latent space.

This approach is motivated by transfer learning being an effective approach for accelerating training processes (Taylor et al., 2014). Conceptually, a VAE learns how to encode inputs into a re-parameterized latent space that groups inputs (images) by their features for a probabilistic reconstruction. The encoder component of the VAE is explored as a pretrained router, which tangentially explores whether such an encoding objective is comparable to a routing process.

These experiments evaluate whether a pre-trained router can provide a meaningful latent organization for downstream expert specialization or otherwise improve the training process, such as through faster convergence. The experiments compare the fixed-router transfer rows in this chapter against the end-to-end MoE baselines reported in Section 5.5. The comparison focuses on the same evaluation quantities used in the baseline MoE section: top-1 routed path accuracy, dense-mixture accuracy, top-5 variants, loss, total parameter count, top-1 inference path cost, and the number of epochs it took for the model to converge.

7.2 Experiment Setups

The transfer experiments reuse the classifier, autoencoder, and MoE configurations reported in Chapter 5. The comparison changes only the router initialization source and whether the transferred encoder continues to update during MoE training. The experts, router type, expert count, sparse optimization procedure, and dataset processing follow the matched end-to-end MoE baseline.

7.2.1 Pretraining Sources and Transfer Modes

Two pretraining objectives are evaluated. Classifier transfer initializes the router encoder from a supervised classifier trained on the same dataset. Autoencoder transfer initializes it from the encoder component of a VAE or residual VAE. These sources differ both in objective and output structure: the classifier encoder is trained to support class prediction, while the variational encoder is trained to support probabilistic reconstruction and latent regularization.

For each transferred source, the router encoder is evaluated in two modes where results are reported:

1. **adaptive transfer**, where the transferred encoder continues to update with the MoE objective; and

2. **frozen transfer**, where the transferred encoder parameters are held constant while the routing head and experts are trained.

MNIST and CIFAR-10 include both modes for both pretraining sources. CIFAR-100 includes adaptive classifier and adaptive autoencoder transfer.

Let θ_s denote the transferred encoder parameters, θ_r the router parameters, and θ_e the

expert parameters. Both modes begin from

$$\theta_r^{(0)} \leftarrow \theta_s. \tag{7.1}$$

Adaptive transfer optimizes θ_r and θ_e jointly under the MoE objective. Frozen transfer optimizes the routing components outside the encoder together with θ_e, while the transferred encoder parameters are excluded from gradient updates.

In the adaptive frozen tests, only the encoder (or encoder-decoder for the autoencoder variants) weights are frozen; embedded experts are still learned so they can optimally adapt to the predefined environment.

7.2.2 Updated Baseline Configurations

Table 7.1 lists the source and expert architectures selected from the updated baseline experiments. All transfer experiments evaluate Centroid, Cosine, Linear, and RBF routers with $K \in \{2, 3, 4, 5\}$ and use `tK2t1` optimization. The classifier source uses the dataset class count as its output width. The autoencoder source uses the latent width selected for the corresponding autoencoder baseline.

Table 7.1: Classifier, autoencoder, and expert configurations used by the transferred-router experiments. The notation follows the baseline chapters: P controls feature-map width, H is model height, D is block depth, and N is autoencoder latent dimensionality.

Dataset	Classifier source	Autoencoder source	MoE expert
MNIST	`Conv2D_FFNN` $(P, H, D) = (2, 2, 1)$	`Conv2D_VAE` $(P, H, D, N) = (2, 2, 1, 2)$	`Conv2D_FFNN` $(P, H, D) = (2, 3, 1)$
CIFAR-10	`Conv2D_FFNN` $(P, H, D) = (4, 3, 2)$	`Conv2D_VAE` $(P, H, D, N) = (4, 3, 2, 8)$	`Conv2D_FFNN` $(P, H, D) = (8, 3, 4)$
CIFAR-100	`Conv2D_Res` $(P, H, D) = (16, 3, 4)$	`Conv2D_ResVAE` $(P, H, D, N) = (16, 3, 4, 32)$	`Conv2D_Res` $(P, H, D) = (32, 3, 8)$

MNIST uses expert dropout 0.1 and batch size 1024. CIFAR-10 uses expert dropout 0.2, batch size 512, random cropping, and horizontal flipping. CIFAR-100 uses expert dropout 0.5, batch size 256, random cropping, and horizontal flipping. No explicit routing regularizer is added, so the transfer comparison isolates the initialization source and encoder update mode.

7.2.3 Evaluation and Matched Baseline Comparison

The transferred models use the same marginal negative-log-likelihood objective as the updated MoE baselines. All experiments use the `tK2t1` router optimization and primarily report the top-1 metrics. All datasets report the Acc1 metric, where CIFAR-100 also reports the Acc5 metric.

For each transfer row, the matched baseline uses the same dataset, router type, expert architecture, K, and `tK2t1` procedure. The tables report

$$\Delta\text{Acc1} = \text{Acc1}_{\text{transfer}} - \text{Acc1}_{\text{baseline}}. \tag{7.2}$$

A positive value indicates that the transferred row exceeds its matched end-to-end baseline. Parameter efficiency is evaluated using the reported top-1 inference-path count,

$$\Delta P_{\text{T1}} = P_{\text{T1,transfer}} - P_{\text{T1,baseline}}. \tag{7.3}$$

The total parameter count includes the router and all experts. The top-1 count includes the router and one selected expert.

7.3 MNIST

The MNIST experiments use compact convolutional encoders and experts. Across the matched router and expert-count rows, the end-to-end MoE baseline averages 90.23% Acc1, 1,010.6 epochs, and 696 parameters on the reported top-1 inference path.

7.3.1 Classifier-Pretrained Router

Adaptive classifier transfer averages 89.98% Acc1, compared with 90.23% for the matched end-to-end MoE rows, and it exceeds the matched baseline in 7 of the 16 router–K compar-

Table 7.2: MNIST classifier-pretrained router results with the encoder updated during MoE training. ΔAcc1 is the difference from the end-to-end MoE baseline with the same router type and K. Params is the total stored parameter count, and T1 Params is the reported top-1 inference-path count.

Router	K	Params	T1 Params	Epochs	Loss	Acc1	ΔAcc1	Entropy	Switch Bias
Centroid	2	1,284	748	1251	0.3775	0.8892	-0.0148	0.1472	0.0009
Centroid	3	1,830	758	1219	0.3025	0.9093	-0.0068	0.1554	0.0159
Centroid	4	2,376	768	1058	0.2922	0.9124	+0.0008	0.1754	0.0291
Centroid	5	2,922	778	999	0.2895	0.9177	+0.0213	0.1759	0.0985
Cosine	2	1,284	748	952	0.3964	0.8786	-0.0190	0.5544	0.0054
Cosine	3	1,830	758	1009	0.3326	0.9023	+0.0049	0.7968	0.0213
Cosine	4	2,376	768	909	0.3619	0.8892	+0.0012	0.7964	0.0104
Cosine	5	2,922	778	991	0.3211	0.9063	+0.0108	0.8475	0.0072
Linear	2	1,284	748	996	0.3778	0.8870	-0.0068	0.1260	0.1715
Linear	3	1,830	758	1172	0.3693	0.8890	-0.0163	0.1711	0.0158
Linear	4	2,376	768	1166	0.3091	0.9089	+0.0030	0.1624	0.0801
Linear	5	2,922	778	1059	0.3187	0.9072	-0.0068	0.1428	0.0620
RBF	2	1,286	750	1215	0.3618	0.8898	-0.0059	0.1685	0.1785
RBF	3	1,833	761	1190	0.3630	0.8901	-0.0101	0.1818	0.1699
RBF	4	2,380	772	963	0.3057	0.9104	-0.0001	0.1127	0.2129
RBF	5	2,927	783	1071	0.2894	0.9096	+0.0054	0.1337	0.1158

Table 7.3: MNIST classifier-pretrained router results with the encoder held frozen during MoE training. ΔAcc1 is the difference from the end-to-end MoE baseline with the same router type and K. Params is the total stored parameter count, and T1 Params is the reported top-1 inference-path count.

Router	K	Params	T1 Params	Epochs	Loss	Acc1	ΔAcc1	Entropy	Switch Bias
Centroid	2	1,284	748	910	0.4204	0.8718	-0.0322	0.8902	0.1167
Centroid	3	1,830	758	1064	0.4128	0.8731	-0.0430	0.7557	0.1034
Centroid	4	2,376	768	965	0.4246	0.8660	-0.0456	0.8122	0.1515
Centroid	5	2,922	778	1116	0.3404	0.8961	-0.0003	0.7613	0.0710
Cosine	2	1,284	748	1022	0.4121	0.8712	-0.0263	0.9983	0.0024
Cosine	3	1,830	758	771	0.4263	0.8699	-0.0274	0.9829	0.0184
Cosine	4	2,376	768	871	0.3946	0.8789	-0.0091	0.9868	0.0211
Cosine	5	2,922	778	1256	0.3282	0.9005	+0.0050	0.9797	0.0075
Linear	2	1,284	748	1165	0.4199	0.8690	-0.0248	0.4218	0.0026
Linear	3	1,830	758	1531	0.4112	0.8776	-0.0277	0.2071	0.2274
Linear	4	2,376	768	1291	0.3721	0.8908	-0.0151	0.2912	0.1091
Linear	5	2,922	778	1040	0.3450	0.8978	-0.0162	0.2492	0.0857
RBF	2	1,286	750	1278	0.3898	0.8814	-0.0142	0.4104	0.2282
RBF	3	1,833	761	1356	0.3887	0.8863	-0.0139	0.3226	0.1751
RBF	4	2,380	772	1305	0.4504	0.8595	-0.0510	0.2928	0.3131
RBF	5	2,927	783	1234	0.3823	0.8833	-0.0209	0.2623	0.1477

isons. Frozen classifier transfer averages 87.96% Acc1 and exceeds the matched baseline in 1 comparison. The classifier-pretrained router adds 68 parameters to the reported top-1 path, approximately 9.8% above the matched baseline. This is a modest architectural increase, but the transfer results do not show a general accuracy gain. Adaptive transfer averages 1,076.2 epochs and frozen transfer averages 1,135.9 epochs, compared with 1,010.6 epochs for the matched baseline. The adaptive condition uses 65.7 more epochs on average, while the frozen condition uses 125.4 more. The transfer results do not show a general accuracy or training-speed gain.

7.3.2 Autoencoder-Pretrained Router

Table 7.4: MNIST autoencoder-pretrained router results with the encoder updated during MoE training. ΔAcc1 is the difference from the end-to-end MoE baseline with the same router type and K. Params is the total stored parameter count, and T1 Params is the reported top-1 inference-path count.

Router	K	Params	T1 Params	Epochs	Loss	Acc1	ΔAcc1	Entropy	Switch Bias
Centroid	2	2,002	1,466	962	0.3802	0.8892	-0.0148	0.1009	0.0263
Centroid	3	2,540	1,468	877	0.3154	0.9058	-0.0103	0.2266	0.0394
Centroid	4	3,078	1,470	836	0.3202	0.9103	-0.0013	0.2587	0.1348
Centroid	5	3,616	1,472	1003	0.3121	0.9121	+0.0156	0.3196	0.0603
Cosine	2	2,002	1,466	897	0.3353	0.9000	+0.0024	0.5806	0.0539
Cosine	3	2,540	1,468	1057	0.3805	0.8785	-0.0189	0.7893	0.0078
Cosine	4	3,078	1,470	1317	0.3452	0.8952	+0.0072	0.8316	0.0000
Cosine	5	3,616	1,472	888	0.3584	0.8919	-0.0036	0.8750	0.0050
Linear	2	2,002	1,466	1251	0.3239	0.9033	+0.0095	0.1252	0.0143
Linear	3	2,540	1,468	1225	0.3335	0.8993	-0.0060	0.2091	0.0314
Linear	4	3,078	1,470	1227	0.2819	0.9174	+0.0116	0.2432	0.1495
Linear	5	3,616	1,472	1069	0.3534	0.8934	-0.0206	0.3025	0.0945
RBF	2	2,004	1,468	610	0.4489	0.8664	-0.0293	0.1364	0.0527
RBF	3	2,543	1,471	1238	0.3568	0.8925	-0.0077	0.0891	0.4741
RBF	4	3,082	1,474	842	0.3607	0.8963	-0.0142	0.1775	0.1687
RBF	5	3,621	1,477	905	0.3477	0.8957	-0.0085	0.2592	0.1761

Adaptive autoencoder transfer averages 89.67% Acc1, compared with 90.23% for the matched end-to-end MoE rows, and exceeds the baseline in 5 comparisons–frozen autoencoder transfer averages 86.92% Acc1, with one matched improvement. Adaptive transfer is higher than frozen transfer in 14 of the 16 matched router–K comparisons. The VAE

Table 7.5: MNIST autoencoder-pretrained router results with the encoder held frozen during MoE training. ΔAcc1 is the difference from the end-to-end MoE baseline with the same router type and K. Params is the total stored parameter count, and T1 Params is the reported top-1 inference-path count.

Router	K	Params	T1 Params	Epochs	Loss	Acc1	ΔAcc1	Entropy	Switch Bias
Centroid	2	2,002	1,466	1206	0.3858	0.8848	-0.0192	0.1427	0.9524
Centroid	3	2,540	1,468	868	0.4263	0.8650	-0.0511	0.5613	0.5501
Centroid	4	3,078	1,470	793	0.5104	0.8355	-0.0761	0.7459	0.2943
Centroid	5	3,616	1,472	851	0.4550	0.8553	-0.0412	0.7935	0.2648
Cosine	2	2,002	1,466	748	0.4704	0.8563	-0.0413	0.9865	0.0018
Cosine	3	2,540	1,468	1173	0.4060	0.8768	-0.0205	0.9288	0.1129
Cosine	4	3,078	1,470	1529	0.3519	0.8958	+0.0078	0.9521	0.0971
Cosine	5	3,616	1,472	1156	0.4469	0.8612	-0.0344	0.9629	0.0861
Linear	2	2,002	1,466	1001	0.4857	0.8479	-0.0459	0.6863	0.3574
Linear	3	2,540	1,468	1111	0.3845	0.8806	-0.0247	0.7753	0.2687
Linear	4	3,078	1,470	748	0.4111	0.8722	-0.0336	0.8660	0.1631
Linear	5	3,616	1,472	1231	0.3833	0.8852	-0.0288	0.8955	0.1441
RBF	2	2,004	1,468	1105	0.4248	0.8777	-0.0180	0.1427	0.9524
RBF	3	2,543	1,471	827	0.4309	0.8608	-0.0394	0.5613	0.5501
RBF	4	3,082	1,474	1091	0.4390	0.8657	-0.0448	0.7459	0.2943
RBF	5	3,621	1,477	1003	0.3707	0.8862	-0.0180	0.7935	0.2648

router increases the reported top-1 path by 774 parameters, approximately 111.2% above the baseline path. Adaptive autoencoder transfer averages 1,012.8 epochs and frozen transfer averages 1,027.6 epochs, compared with 1,010.6 for the matched baseline. These correspond to increases of 2.2 and 17.0 epochs; marginal increases, but still increases. The additional encoder-decoder structure more than doubles the reported active parameter count without improving average Acc1 or reducing the average epoch count. The frozen VAE rows also have a higher average switch bias (0.3347) than the adaptive rows (0.0931), indicating less even dataset-level expert use.

Across the MNIST experiments, allowing the transferred encoder to update is more reliable than holding it frozen. Classifier pretraining is also more parameter efficient than VAE pretraining because its reported top-1 path is closer to the matched MoE baseline. Training speed does not improve on average: the baseline uses 1,010.6 epochs, adaptive classifier transfer uses 1,076.2, frozen classifier transfer uses 1,135.9, adaptive autoencoder transfer uses 1,012.8, and frozen autoencoder transfer uses 1,027.6. These results provide

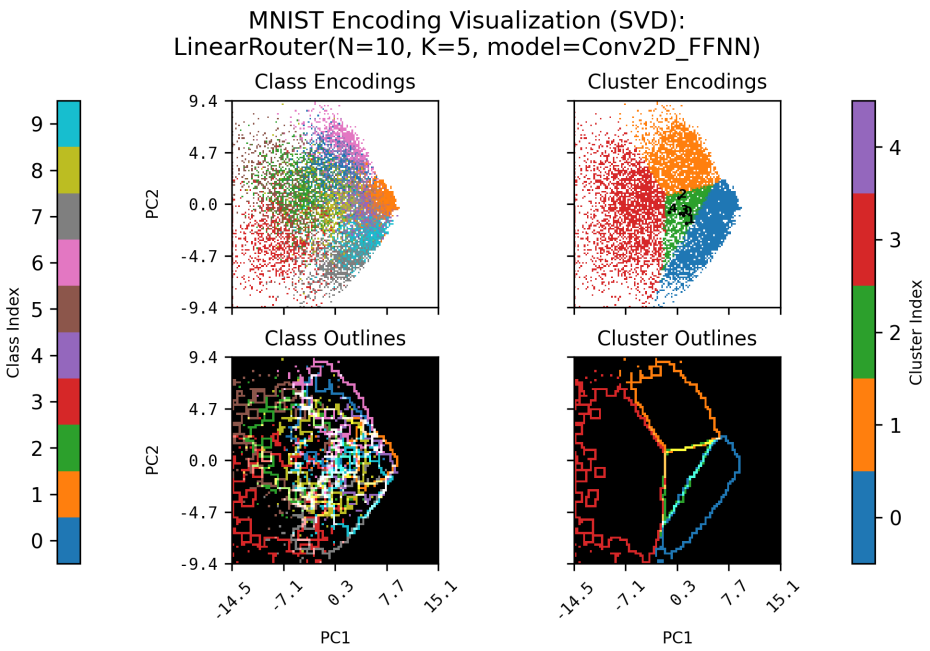


Figure 7.1: Example MNIST MoE trained with a fixed pre-trained classifier-router (Linear) $\{P = 2, H = 2, D = 1, N = 10\}$ with experts configured as $\{P = 2, H = 3, D = 1\}$.

limited support for transfer as an initialization method and little support for a frozen latent reference frame on MNIST.

Figures 7.1 and 7.2 demonstrate two separately trained MoE systems with the same router. One potential issue is noticeable here, however, where the expert embeddings struggle to move very far. This is likely because they receive few gradients with the rest of the router fixed, so a better approach may require allowing them to explore faster or even predefining them, such as through a clustering algorithm.

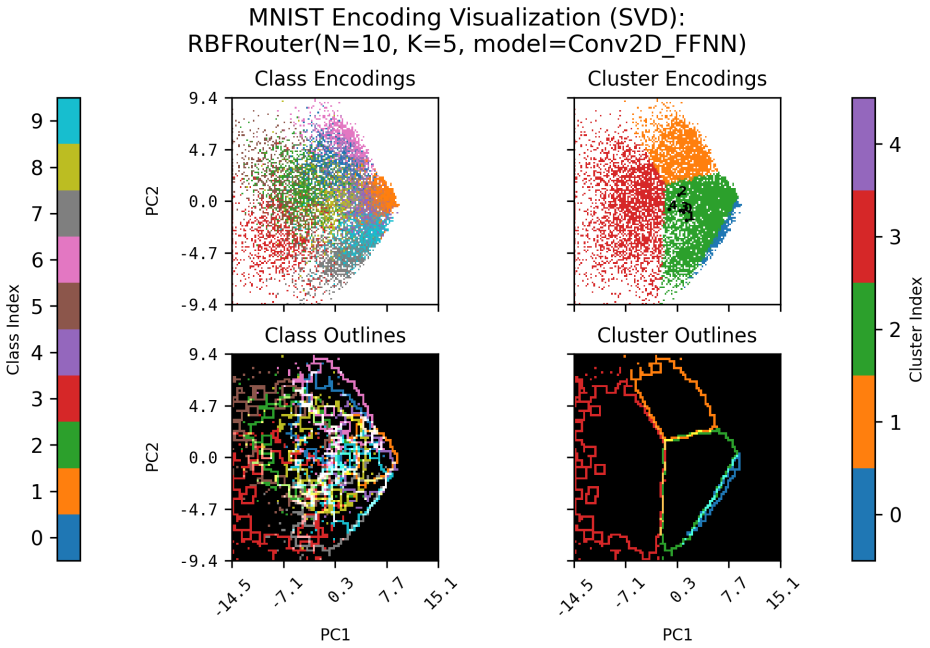


Figure 7.2: Example MNIST MoE trained with a fixed pre-trained classifier-router (RBF) $\{P = 2, H = 2, D = 1, N = 10\}$ with experts configured as $\{P = 2, H = 3, D = 1\}$.

7.4 CIFAR-10

The CIFAR-10 experiments use a `Conv2D_FFNN` router source with $(P, H, D) = (4, 3, 2)$ or a `Conv2D_VAE` source with $(P, H, D, N) = (4, 3, 2, 8)$. The matched experts use $(P, H, D) = (8, 3, 4)$. Across the matched rows, the end-to-end MoE baseline averages 74.49% Acc1, 608.6 epochs, and 48,635 parameters on the reported top-1 inference path.

7.4.1 Classifier-Pretrained Router

Adaptive classifier transfer averages 74.55% Acc1, compared with 74.49% for the matched end-to-end MoE rows, with 8 matched improvements. Frozen classifier transfer averages 73.66% Acc1 and exceeds the baseline in 5 comparisons.

The classifier-pretrained path adds only 41 reported top-1 parameters, approximately 0.08%. Adaptive classifier transfer averages 582.7 epochs, which is 25.9 fewer than the 608.6-epoch baseline average. Frozen classifier transfer averages 667.3 epochs, which is 58.8 more

Table 7.6: CIFAR-10 classifier-pretrained router results with the encoder updated during MoE training. ΔAcc1 is the difference from the end-to-end MoE baseline with the same router type and K. Params is the total stored parameter count, and T1 Params is the reported top-1 inference-path count.

Router	K	Params	T1 Params	Epochs	Loss	Acc1	ΔAcc1	Entropy	Switch Bias
Centroid	2	92,294	48,660	639	0.7184	0.7555	-0.0019	0.1523	0.0404
Centroid	3	136k	48,670	664	0.7139	0.7653	+0.0314	0.2698	0.0070
Centroid	4	180k	48,680	736	0.6934	0.7760	+0.0384	0.2731	0.0181
Centroid	5	223k	48,690	535	0.8175	0.7275	-0.0066	0.3242	0.0126
Cosine	2	92,294	48,660	672	0.7117	0.7571	+0.0141	0.5838	0.0197
Cosine	3	136k	48,670	585	0.7318	0.7519	+0.0064	0.7713	0.0223
Cosine	4	180k	48,680	456	0.8349	0.7140	-0.0140	0.8619	0.0180
Cosine	5	223k	48,690	548	0.8241	0.7209	-0.0016	0.8721	0.0037
Linear	2	92,294	48,660	580	0.7079	0.7634	+0.0034	0.1565	0.0381
Linear	3	136k	48,670	580	0.7495	0.7511	+0.0052	0.2978	0.0048
Linear	4	180k	48,680	517	0.7675	0.7447	-0.0092	0.2893	0.0132
Linear	5	223k	48,690	538	0.8281	0.7256	-0.0136	0.3161	0.0105
RBF	2	92,296	48,662	332	0.8016	0.7276	-0.0384	0.1664	0.0357
RBF	3	136k	48,673	678	0.7304	0.7598	+0.0083	0.2778	0.0681
RBF	4	180k	48,684	838	0.6992	0.7712	+0.0016	0.3006	0.0281
RBF	5	223k	48,695	425	0.8449	0.7169	-0.0140	0.3556	0.0197

Table 7.7: CIFAR-10 classifier-pretrained router results with the encoder held frozen during MoE training. ΔAcc1 is the difference from the end-to-end MoE baseline with the same router type and K. Params is the total stored parameter count, and T1 Params is the reported top-1 inference-path count.

Router	K	Params	T1 Params	Epochs	Loss	Acc1	ΔAcc1	Entropy	Switch Bias
Centroid	2	92,294	48,660	670	0.7165	0.7512	-0.0062	0.8695	0.2507
Centroid	3	136k	48,670	569	0.8054	0.7204	-0.0135	0.7954	0.0906
Centroid	4	180k	48,680	608	0.7822	0.7327	-0.0049	0.5971	0.1717
Centroid	5	223k	48,690	615	0.7680	0.7377	+0.0036	0.6818	0.1103
Cosine	2	92,294	48,660	616	0.7338	0.7434	+0.0003	0.9958	0.0016
Cosine	3	136k	48,670	719	0.7643	0.7323	-0.0133	0.9787	0.0066
Cosine	4	180k	48,680	711	0.7960	0.7236	-0.0044	0.9482	0.0228
Cosine	5	223k	48,690	814	0.7467	0.7408	+0.0183	0.9609	0.0139
Linear	2	92,294	48,660	701	0.7524	0.7412	-0.0188	0.4836	0.0455
Linear	3	136k	48,670	729	0.7516	0.7419	-0.0040	0.3307	0.1318
Linear	4	180k	48,680	738	0.7410	0.7471	-0.0068	0.1660	0.2235
Linear	5	223k	48,690	752	0.7549	0.7411	+0.0020	0.1990	0.1693
RBF	2	92,296	48,662	561	0.7695	0.7310	-0.0350	0.3253	0.5326
RBF	3	136k	48,673	599	0.7797	0.7306	-0.0210	0.3341	0.1464
RBF	4	180k	48,684	537	0.7941	0.7254	-0.0442	0.1103	0.2866
RBF	5	223k	48,695	738	0.7441	0.7452	+0.0143	0.1640	0.1890

than the baseline. The adaptive condition is the only reported transfer setting that reduces the average epoch count on CIFAR-10, although its accuracy changes are not consistent across every router and K.

7.4.2 Autoencoder-Pretrained Router

Table 7.8: CIFAR-10 autoencoder-pretrained router results with the encoder updated during MoE training. ΔAcc1 is the difference from the end-to-end MoE baseline with the same router type and K. Params is the total stored parameter count, and T1 Params is the reported top-1 inference-path count.

Router	K	Params	T1 Params	Epochs	Loss	Acc1	ΔAcc1	Entropy	Switch Bias
Centroid	2	95,752	52,118	656	0.7318	0.7526	-0.0048	0.1507	0.0608
Centroid	3	139k	52,126	634	0.7957	0.7362	+0.0024	0.3282	0.0086
Centroid	4	183k	52,134	693	0.7147	0.7682	+0.0307	0.2896	0.0184
Centroid	5	227k	52,142	755	0.7101	0.7649	+0.0308	0.3183	0.0442
Cosine	2	95,752	52,118	548	0.7777	0.7352	-0.0078	0.6293	0.0389
Cosine	3	139k	52,126	767	0.7701	0.7370	-0.0085	0.8237	-0.0008
Cosine	4	183k	52,134	590	0.8096	0.7208	-0.0072	0.8316	0.0104
Cosine	5	227k	52,142	628	0.7697	0.7348	+0.0123	0.8559	0.0229
Linear	2	95,752	52,118	644	0.7288	0.7558	-0.0042	0.1645	0.0280
Linear	3	139k	52,126	709	0.7410	0.7498	+0.0039	0.3315	0.0259
Linear	4	183k	52,134	708	0.7300	0.7555	+0.0016	0.2515	0.1369
Linear	5	227k	52,142	465	0.7764	0.7452	+0.0060	0.3004	0.0497
RBF	2	95,754	52,120	606	0.7112	0.7608	-0.0052	0.1529	0.0455
RBF	3	139k	52,129	705	0.7674	0.7401	-0.0114	0.4191	0.0127
RBF	4	183k	52,138	668	0.7405	0.7525	-0.0170	0.3544	0.0935
RBF	5	227k	52,147	485	0.7695	0.7462	+0.0153	0.3085	0.1073

Adaptive autoencoder transfer averages 74.72% Acc1, compared with 74.49% for the matched end-to-end MoE rows, and exceeds the baseline in 8 of the 16 matched comparisons. Frozen autoencoder transfer averages 71.83% Acc1 and does not exceed any matched baseline row. Adaptive autoencoder transfer is higher than the frozen condition in all 16 comparisons. The VAE source adds 3,496 parameters to the reported top-1 path, approximately 7.2%. Adaptive autoencoder transfer averages 641.3 epochs and frozen transfer averages 681.9 epochs, compared with 608.6 for the matched baseline. The two conditions use 32.8 and 73.3 more epochs on average. A higher active-path cost and longer training than classifier transfer accompany the small adaptive accuracy increase. Frozen autoencoder transfer also

Table 7.9: CIFAR-10 autoencoder-pretrained router results with the encoder held frozen during MoE training. ΔAcc1 is the difference from the end-to-end MoE baseline with the same router type and K. Params is the total stored parameter count, and T1 Params is the reported top-1 inference-path count.

Router	K	Params	T1 Params	Epochs	Loss	Acc1	ΔAcc1	Entropy	Switch Bias
Centroid	2	95,752	52,118	605	0.7691	0.7292	-0.0282	0.7987	0.4402
Centroid	3	139k	52,126	699	0.8043	0.7181	-0.0158	0.7814	0.2244
Centroid	4	183k	52,134	736	0.8101	0.7139	-0.0237	0.8559	0.0945
Centroid	5	227k	52,142	545	0.8997	0.6844	-0.0497	0.8814	0.0745
Cosine	2	95,752	52,118	691	0.7641	0.7316	-0.0115	0.9807	0.0531
Cosine	3	139k	52,126	556	0.7964	0.7228	-0.0228	0.9548	0.0930
Cosine	4	183k	52,134	846	0.7998	0.7166	-0.0114	0.9677	0.0438
Cosine	5	227k	52,142	556	0.8773	0.6927	-0.0298	0.9710	0.0390
Linear	2	95,752	52,118	620	0.7968	0.7203	-0.0397	0.5222	0.3872
Linear	3	139k	52,126	705	0.7724	0.7281	-0.0178	0.3226	0.3605
Linear	4	183k	52,134	826	0.7814	0.7263	-0.0276	0.5180	0.2159
Linear	5	227k	52,142	728	0.7874	0.7244	-0.0147	0.5390	0.2008
RBF	2	95,754	52,120	747	0.7475	0.7376	-0.0284	0.2818	0.8366
RBF	3	139k	52,129	668	0.7843	0.7231	-0.0284	0.2709	0.3986
RBF	4	183k	52,138	674	0.8115	0.7147	-0.0549	0.5132	0.1579
RBF	5	227k	52,147	708	0.8245	0.7084	-0.0225	0.5444	0.1404

has higher average entropy and switch bias, indicating less structured per-sample routing collapsing to a single expert.

CIFAR-10 provides evidence that transferred initialization can match the end-to-end MoE reference when the encoder continues to adapt. Both adaptive sources divide the 16 matched comparisons evenly between increases and decreases, and their average accuracies are close to the 74.49% baseline. The classifier source reaches 74.55% with almost no increase in the reported top-1 path and reduces the average epoch count from 608.6 to 582.7. The VAE source reaches 74.72%, but increases the active-path cost and the average epoch count to 641.3. Freezing either source lowers average accuracy and increases the average epoch count. Transfer therefore appears more useful as an adaptive initialization than as a permanent routing partition.

7.5 CIFAR-100

The CIFAR-100 experiments use residual classifier and residual VAE sources with $(P, H, D) = (16, 3, 4)$, with latent width $N = 32$ for the VAE. The experts use `Conv2D_Res` with $(P, H, D) = (32, 3, 8)$. Only adaptive transfer is reported. Across the matched rows, the end-to-end MoE baseline averages 72.97% Acc1, 424.1 epochs, and 1,661k parameters on the reported top-1 inference path.

7.5.1 Classifier-Pretrained Router

Table 7.10: CIFAR-100 classifier-pretrained router results with the encoder updated during MoE training. ΔAcc1 is the difference from the end-to-end MoE baseline with the same router type and K. Params is the total stored parameter count, and T1 Params is the reported top-1 inference-path count.

Router	K	Params	T1 Params	Epochs	Loss	Acc1	Acc5	ΔAcc1	Entropy	Switch Bias
Centroid	2	3,149k	1,665k	515	0.9211	0.7580	0.9365	+0.0269	0.0731	0.0106
Centroid	3	4,633k	1,665k	349	1.0613	0.7217	0.9170	+0.0046	0.0874	0.0794
Centroid	4	6,117k	1,665k	436	0.9892	0.7502	0.9213	+0.0113	0.1170	0.0302
Centroid	5	7,601k	1,665k	360	1.0780	0.7275	0.9081	-0.0228	0.1334	0.0288
Cosine	2	3,149k	1,665k	462	0.9728	0.7372	0.9253	+0.0276	0.5602	0.0035
Cosine	3	4,633k	1,665k	447	1.0354	0.7365	0.9162	+0.0367	0.7709	0.0292
Cosine	4	6,117k	1,665k	514	1.0334	0.7310	0.9152	+0.0234	0.8221	0.0270
Cosine	5	7,601k	1,665k	480	1.0744	0.7204	0.9106	+0.0227	0.8536	0.0038
Linear	2	3,149k	1,665k	453	0.9845	0.7349	0.9281	-0.0053	0.0695	0.0409
Linear	3	4,633k	1,665k	461	0.9317	0.7546	0.9326	+0.0180	0.0791	0.0795
Linear	4	6,117k	1,665k	271	1.2036	0.6894	0.9011	-0.0585	0.0894	0.0701
Linear	5	7,601k	1,665k	420	1.0029	0.7425	0.9208	-0.0065	0.0712	0.1590
RBF	2	3,149k	1,665k	782	0.9062	0.7511	0.9350	+0.0132	0.1594	0.0210
RBF	3	4,633k	1,665k	484	0.9346	0.7562	0.9282	+0.0237	0.0700	0.0697
RBF	4	6,117k	1,665k	335	1.0872	0.7171	0.9116	-0.0251	0.1084	0.0357
RBF	5	7,601k	1,665k	397	1.0024	0.7489	0.9198	+0.0110	0.0749	0.0987

Adaptive classifier transfer averages 73.61% Acc1, compared with 72.97% for the matched end-to-end MoE rows, and exceeds the matched baseline in 11 of the 16 comparisons. The reported top-1 path increases by 4,658 parameters, approximately 0.28%. Average switch bias decreases from 0.0915 in the baseline rows to 0.0492, suggesting more even aggregate expert use across this condition.

Adaptive classifier transfer averages 447.9 epochs, compared with 424.1 for the matched baseline, an increase of 23.8 epochs. The accuracy and parameter comparison are an improvement, while the average epoch count does not indicate faster training.

7.5.2 Autoencoder-Pretrained Router

Table 7.11: CIFAR-100 autoencoder-pretrained router results with the encoder updated during MoE training. ΔAcc1 is the difference from the end-to-end MoE baseline with the same router type and K. Params is the total stored parameter count, and T1 Params is the reported top-1 inference-path count.

Router	K	Params	T1 Params	Epochs	Loss	Acc1	Acc5	ΔAcc1	Entropy	Switch Bias
Centroid	2	3,220k	1,736k	372	1.0572	0.7129	0.9189	-0.0182	0.1102	0.3318
Centroid	3	4,704k	1,736k	498	0.9795	0.7437	0.9230	+0.0266	0.1862	0.0599
Centroid	4	6,188k	1,736k	344	1.1585	0.6973	0.9006	-0.0416	0.1975	0.0920
Centroid	5	7,672k	1,736k	329	1.1919	0.6985	0.8919	-0.0518	0.1881	0.0450
Cosine	2	3,220k	1,736k	494	0.9890	0.7258	0.9282	+0.0163	0.5527	0.4245
Cosine	3	4,704k	1,736k	354	1.1199	0.6949	0.9086	-0.0049	0.7177	0.2732
Cosine	4	6,188k	1,736k	307	1.2631	0.6672	0.8876	-0.0403	0.8180	0.1365
Cosine	5	7,672k	1,736k	486	1.1025	0.7115	0.9076	+0.0138	0.8617	0.0461
Linear	2	3,220k	1,736k	485	0.9592	0.7416	0.9287	+0.0014	0.1447	0.0533
Linear	3	4,704k	1,736k	508	0.9371	0.7572	0.9272	+0.0207	0.1165	0.2116
Linear	4	6,188k	1,736k	449	1.0013	0.7478	0.9196	-0.0001	0.1134	0.1009
Linear	5	7,672k	1,736k	472	0.9883	0.7468	0.9182	-0.0022	0.1499	0.0940
RBF	2	3,220k	1,736k	495	0.9721	0.7378	0.9291	-0.0000	0.0590	0.3878
RBF	3	4,704k	1,736k	443	1.0100	0.7350	0.9188	+0.0025	0.1145	0.2648
RBF	4	6,188k	1,736k	542	0.9149	0.7625	0.9278	+0.0204	0.1150	0.1312
RBF	5	7,672k	1,736k	470	0.9727	0.7548	0.9164	+0.0170	0.1750	0.0329

Adaptive residual-VAE transfer averages 72.72% Acc1, compared with 72.97% for the matched end-to-end MoE rows, with 8 matched improvements. The reported top-1 path increases by 75,204 parameters, approximately 4.5%. Its average switch bias (0.1678) is also higher than the baseline value (0.0915).

Adaptive residual-VAE transfer averages 440.5 epochs, compared with 424.1 for the matched baseline, an increase of 16.4 epochs. The residual VAE source therefore produces mixed classification changes with a larger parameter increase than the classifier source and no reduction in the average epoch count. Its reconstruction-oriented latent representation can support competitive rows, but it does not provide a general efficiency improvement over the matched end-to-end router.

The classifier-pretrained adaptive router gives greater improvement in CIFAR-100 accuracy and parameter count: it averages 73.61% Acc1 with 1,665k top-1 path parameters, compared with 72.97% and 1,661k parameters for the baseline. The residual-VAE source averages 72.72% with 1,736k top-1 path parameters. Neither source reduces the average epoch count: the classifier and residual-VAE conditions use 447.9 and 440.5 epochs, compared with 424.1 for the baseline.

7.6 Cross-Dataset Comparison

Table 7.12: Average number of epochs across router type and K for each reported training strategy. Lower values indicate fewer recorded training epochs. Dashes mark transfer experiments that were skipped based on prior results.

Dataset	End-to-End	Classifier Adaptive	Classifier Frozen	Autoencoder Adaptive	Autoencoder Frozen
MNIST	1,010.6	1,076.2	1,135.9	1,012.8	1,027.6
CIFAR-10	608.6	582.7	667.3	641.3	681.9
CIFAR-100	424.1	447.9	–	440.5	–

Three broad patterns are visible across the updated experiments. First, adaptive transfer is consistently more reliable than frozen transfer where both are evaluated. The adaptive classifier condition is higher than its frozen counterpart in all 16 MNIST comparisons and in 9 of 16 CIFAR-10 comparisons. Adaptive autoencoder transfer is higher in 14 of 16 MNIST comparisons and all 16 CIFAR-10 comparisons.

Second, classifier pretraining is generally more parameter efficient. Its reported top-1 overhead is 9.8% on MNIST, 0.08% on CIFAR-10, and 0.28% on CIFAR-100. The corresponding autoencoder overheads are 111.2%, 7.2%, and 4.5%. The VAE and residual-VAE routers carry more parameters without producing a consistent accuracy advantage over classifier initialization.

Third, Table 7.12 shows that transfer does not consistently shorten training. CIFAR-10 adaptive classifier transfer is the only condition with a lower average than its matched

baseline, decreasing from 608.6 to 582.7 epochs. MNIST adaptive autoencoder transfer is close to its baseline at 1,012.8 versus 1,010.6 epochs. The other reported conditions increase the average epoch count by between 16.4 and 125.4 epochs. Frozen transfer is slower on average in every reported source–dataset comparison.

7.7 Summary

The experiments test whether a pretrained router can provide a reusable latent reference frame for MoE expert specialization. The updated results provide partial support for pretrained initialization but do not support a general benefit from keeping the transferred representation frozen.

Classifier-pretrained adaptive routers give the more parameter-efficient transfer results. They stay close to the matched MoE baseline on MNIST and CIFAR-10 and improve the descriptive CIFAR-100 accuracy with a small active-path increase. Autoencoder-pretrained adaptive routers are also competitive in parts of the CIFAR-10 and CIFAR-100 experiments, but their larger reported router cost limits parameter efficiency. On MNIST, neither adaptive source improves the descriptive baseline result.

Frozen transfer is less reliable. Its descriptive accuracy is lower than adaptive transfer on MNIST and CIFAR-10, and the autoencoder-frozen condition also shows larger switch bias.

Overall, router pretraining can provide a useful initialization when the encoder stays trainable, particularly when the source objective is supervised classification. As noted, the embedded experts are still learned. This still weakens potential training-process optimizations, so future work may also look at predefining these. Such predefined embeddings could also be used for a dynamic expert allocation methodology, where an unsupervised clustering approach could map optimal decision boundaries and simultaneously determine how many experts are required.

The training-speed evidence is limited to CIFAR-10 adaptive classifier transfer, which

averages 582.7 epochs compared with 608.6 for the baseline. Every other reported transfer condition is equal to or slower than its baseline within the displayed averages. The VAE source also does not provide a general parameter-efficiency advantage over the updated end-to-end MoE baselines. The original hypothesis is therefore only partially supported: transferred latent learning can assist adaptation in selected settings, but a stable frozen latent reference frame is not consistently beneficial.

As a result, the router-transfer hypothesis is not accepted under these configurations. Pretraining the router does not reliably reduce total training cost while preserving or improving performance, or provide clear indications as to when it would for these conditions.

Chapter 8

Dynamic Inference Paths

8.1 Overview

The hypothesis tested in this chapter is given as follows: Effective dynamic inference can be achieved by pre-training a MoE system, then up-scaling under-performing experts to improve system performance while using minimal resources to handle simpler problems.

The motivation for this hypothesis is simple: performance is unlikely to be uniform across all subproblem spaces, and it would be ideal if improving accuracy on more complex/difficult subproblem spaces did not add additional computation costs to the simpler ones. This balances improving performance with a lower average inference speed than if the entire system were upscaled to account for the weakest part.

The dynamic inference approach explored in this chapter extends the MoE framework by allowing different experts to have different parameter counts, thereby allowing the system to assign inexpensive experts to lower-complexity regions of the problem space and larger experts to more difficult regions. Notably, these experiments detail a variation of transfer learning, where pretrained routers are again used. The difference between this and the previous chapter is that these pre-trained routers are actually trained to be routers. The hypotheses then hinge upon experts being scalable when routing is fixed.

This approach is specifically used because it avoids issues of routing collapse. In an end-to-end training process, having the Router realize that a subset of experts are more capable than the rest would likely lead to the Router over-relying on that single expert, thus not



Figure 8.1: CIFAR-10 imbalanced experts mixture capabilities example. Uses a Linear router ($K = 2, N = 8$) with a Conv2D_FFNN encoder ($P = 4, H = 3, D = 2$) and Conv2D_FFNN experts ($P = 8, H = 3, D = 4$), trained using a `tK2t1` optimization process.

developing any real specialization.

This chapter evaluates whether a fixed routing structure can be used as a stable partitioning of the input space, after which selected experts can be replaced with higher-capacity alternatives. The goal is not simply to maximize classification accuracy, but to improve the accuracy-to-cost tradeoff by increasing the parameter count only along inference paths that benefit from additional model capacity.

8.2 Complexity Bias

The dynamic-inference hypothesis assumes that routed subproblems can differ in difficulty. Some subproblem spaces may be represented adequately, while others may require larger models to handle them sufficiently. This chapter uses the term *complexity bias* for systematic differences in the capacity required across routed portions of the input distribution. A low-performing expert does not establish that its routed subproblem is intrinsically complex. Such biases are relative to the behavior of other experts.

For example, consider Figure 8.1, which details the per-expert performances on the

CIFAR-10 dataset. Notably,y expert 1 performs, on average, quite well across the labels it is dominant on: 0, 1, 8, and 9. Expert 0, by comparison, performs noticeably worse on average across its dominant labels: 2, 3, 4, 5, 6, and 7. Potentially, this is due to the 4-6 label split, where expert 0 has an easier time learning fewer labels. Or it could be because expert 1 has the more difficult labels, such as label 3, which has a $< 60\%$ acc1 score. Regardless of the situation, note the inherent imbalance in the distribution; performance across all labels is definitively non-uniform. Furthermore, average performance across experts is also not similar.

The theory of the approach is that by improving such under-performing experts, like expert 0 in Figure 8.1, the average system performance can be increased and simultaneously–with the benefit of MoE routing–the inference cost on solving those "simpler" labels routed to expert 1 is not increased to compensate. Thus, a balanced blend of optimization and computational efficiency can be achieved.

8.3 Approach

The experimental procedure has three stages:

1. train an end-to-end MoE baseline and retain its learned Router;
2. freeze the Router and construct larger replacement experts; and
3. optimize the replacement experts while preserving the original routing partition.

For an original expert configuration (P_e, H_e, D_e), the experiments consider

$$(P_e, H_e, D_e) \longrightarrow (2P_e, H_e, D_e) \tag{8.1}$$

for width expansion and

$$(P_e, H_e, D_e) \longrightarrow (P_e, H_e, 2D_e) \tag{8.2}$$

for depth expansion. MNIST and CIFAR-10 evaluate both transformations under `tK` and `t1` training. CIFAR-100 evaluates residual depth expansion under `tK`.

Because the Router is frozen, differences in routing entropy and switch bias mainly reflect the inherited partition and evaluation probabilities, not router adaptation during replacement training. Therefore, such routing metrics are disregarded in this section.

Note that because the Router is frozen, experts are effectively trained independently of each other. Thus, despite a full replacement of all experts, this approach can be reduced to only swapping out relatively lower-performing experts. Furthermore, when retraining the system, training will be significantly faster with sparse replacements, as there is neither a router to train nor additional learnable parameters from experts not being replaced.

8.4 Experiment Setups

All experiments use Centroid, Cosine, Linear, and RBF routers with $K \in \{2, 3, 4, 5\}$. Each row is matched to the `tK2t1` MoE baseline with the same dataset, Router, K, router encoder, and original expert architecture. Table 8.1 summarizes the replacement experiments.

Table 8.1: Router, original expert, and replacement expert configurations used in the dynamic-inference experiments. P controls feature-map width, H is model height, D is block depth, and N is router output dimensionality.

Dataset	Router encoder	Original expert	Replacement expert	Training mode
MNIST	`Conv2D_FFNN` $(P, H, D, N) = (2, 2, 1, 2)$	`Conv2D_FFNN` $(2, 3, 1)$	`Conv2D_FFNN` $(4, 3, 1)$	`tK`, `t1`
MNIST	`Conv2D_FFNN` $(P, H, D, N) = (2, 2, 1, 2)$	`Conv2D_FFNN` $(2, 3, 1)$	`Conv2D_FFNN` $(2, 3, 2)$	`tK`, `t1`
CIFAR-10	`Conv2D_FFNN` $(P, H, D, N) = (4, 3, 2, 8)$	`Conv2D_FFNN` $(8, 3, 4)$	`Conv2D_FFNN` $(16, 3, 4)$	`tK`, `t1`
CIFAR-10	`Conv2D_FFNN` $(P, H, D, N) = (4, 3, 2, 8)$	`Conv2D_FFNN` $(8, 3, 4)$	`Conv2D_FFNN` $(8, 3, 8)$	`tK`, `t1`
CIFAR-100	`Conv2D_Res` $(P, H, D, N) = (16, 3, 4, 32)$	`Conv2D_Res` $(32, 3, 8)$	`Conv2D_Res` $(32, 3, 16)$	`tK`

MNIST uses a batch size of 1024 and expert dropout of 0.1. CIFAR-10 uses a batch size

of 512, expert dropout of 0.2, random cropping, and horizontal flipping. CIFAR-100 uses a batch size of 256, expert dropout of 0.5, random cropping, and horizontal flipping.

For each row, the accuracy comparison is

$$\Delta\text{T1Acc1} = \text{T1Acc1}_{\text{replacement}} - \text{Acc1}_{\text{baseline}}. \tag{8.3}$$

The baseline has equal top-1 and mixture Acc1 after `tK2t1`, so one baseline value supports both comparisons.

8.5 Results

8.5.1 MNIST

The matched MNIST MoE rows average 90.23% Acc1, 1010.6 epochs, and 696 parameters on the reported top-1 path.

8.5.1.1 Width Expansion

Table 8.2: MNIST dynamic-inference results for width-doubled experts with dense-mixture training. Each row reports one Router and expert-count experiment. ΔT1 Acc1 is measured against the end-to-end MoE baseline with the same router type and K.

Router	K	Params	T1 Params	Epochs	Loss	T1 Acc1	Dense Acc1	ΔT1 Acc1
Centroid	2	3,864	2,010	702	0.1395	0.9527	0.9571	+0.0487
Centroid	3	5,720	2,012	897	0.1272	0.9560	0.9629	+0.0399
Centroid	4	7,576	2,014	350	0.1755	0.9403	0.9500	+0.0287
Centroid	5	9,432	2,016	445	0.1700	0.9324	0.9599	+0.0359
Cosine	2	3,864	2,010	748	0.1409	0.9565	0.9589	+0.0589
Cosine	3	5,720	2,012	577	0.1463	0.9530	0.9643	+0.0556
Cosine	4	7,576	2,014	724	0.1266	0.9553	0.9677	+0.0673
Cosine	5	9,432	2,016	254	0.3528	0.9111	0.9375	+0.0156
Linear	2	3,864	2,010	690	0.1398	0.9552	0.9588	+0.0614
Linear	3	5,720	2,012	687	0.1269	0.9564	0.9616	+0.0512
Linear	4	7,576	2,014	633	0.1365	0.9532	0.9615	+0.0474
Linear	5	9,432	2,016	604	0.1488	0.9487	0.9557	+0.0347
RBF	2	3,866	2,012	723	0.1388	0.9550	0.9594	+0.0593
RBF	3	5,723	2,015	577	0.1583	0.9453	0.9553	+0.0452
RBF	4	7,580	2,018	631	0.1190	0.9611	0.9683	+0.0506
RBF	5	9,437	2,021	639	0.1292	0.9589	0.9666	+0.0547

Table 8.3: MNIST dynamic-inference results for width-doubled experts with top-1 path training. Each row reports one Router and expert-count experiment. ΔT1 Acc1 is measured against the end-to-end MoE baseline with the same router type and K.

Router	K	Params	T1 Params	Epochs	Loss	T1 Acc1	Dense Acc1	ΔT1 Acc1
Centroid	2	3,864	2,010	581	0.1366	0.9582	0.9582	+0.0543
Centroid	3	5,720	2,012	546	0.1502	0.9528	0.9528	+0.0367
Centroid	4	7,576	2,014	350	0.1752	0.9459	0.9459	+0.0343
Centroid	5	9,432	2,016	445	0.1629	0.9517	0.9517	+0.0552
Cosine	2	3,864	2,010	596	0.1613	0.9497	0.9497	+0.0522
Cosine	3	5,720	2,012	577	0.1283	0.9610	0.9610	+0.0636
Cosine	4	7,576	2,014	570	0.1413	0.9576	0.9576	+0.0696
Cosine	5	9,432	2,016	254	0.2326	0.9286	0.9286	+0.0331
Linear	2	3,864	2,010	537	0.1634	0.9505	0.9505	+0.0567
Linear	3	5,720	2,012	437	0.1740	0.9472	0.9472	+0.0419
Linear	4	7,576	2,014	522	0.1416	0.9568	0.9568	+0.0509
Linear	5	9,432	2,016	558	0.1516	0.9541	0.9541	+0.0401
RBF	2	3,866	2,012	626	0.1311	0.9602	0.9602	+0.0645
RBF	3	5,723	2,015	577	0.1513	0.9555	0.9555	+0.0553
RBF	4	7,580	2,018	607	0.1221	0.9649	0.9649	+0.0544
RBF	5	9,437	2,021	639	0.1262	0.9618	0.9618	+0.0575

Width expansion improves top-1 accuracy in all 16 matched comparisons under both optimization modes. The `tK` rows average 94.95% top-1 Acc1 and 95.91% dense Acc1. Direct `t1` training averages 95.35% Acc1. These results compare with 90.23% for the matched baseline.

The replacement stage averages 617.6 epochs under `tK` and 526.4 under `t1`, compared with 1010.6 epochs for the complete baseline run. The larger expert path uses 2,014 parameters on average, compared with 696 for the baseline, an increase of approximately 189.4%.

The increased-width experiments detail universal improvements, with also greatly accelerated training speed consistent with the earlier assumptions. Note this further implies that a suitable pre-trained router could indeed improve training speeds, but the earlier approaches were insufficient. Notably, the `t1` approach outperforms the `tK` approach here in terms of acc1 by 0.40%, as well as consistently converging faster.

8.5.1.2 Depth Expansion

Depth expansion also improves every matched top-1 comparison. The `tK` rows average 95.47% top-1 Acc1 and 96.40% dense Acc1. The `t1` rows average 95.87% Acc1. The corre-

Table 8.4: MNIST dynamic-inference results for depth-doubled experts with dense-mixture training. Each row reports one Router and expert-count experiment. ΔT1 Acc1 is measured against the end-to-end MoE baseline with the same router type and K.

Router	K	Params	T1 Params	Epochs	Loss	T1 Acc1	Dense Acc1	ΔT1 Acc1
Centroid	2	2,796	1,476	607	0.1308	0.9566	0.9601	+0.0526
Centroid	3	4,118	1,478	475	0.1358	0.9527	0.9598	+0.0367
Centroid	4	5,440	1,480	745	0.1107	0.9575	0.9666	+0.0459
Centroid	5	6,762	1,482	384	0.1837	0.9281	0.9533	+0.0317
Cosine	2	2,796	1,476	677	0.1376	0.9579	0.9602	+0.0603
Cosine	3	4,118	1,478	762	0.1096	0.9619	0.9733	+0.0646
Cosine	4	5,440	1,480	589	0.1283	0.9663	0.9748	+0.0783
Cosine	5	6,762	1,482	449	0.1718	0.9412	0.9646	+0.0457
Linear	2	2,796	1,476	762	0.1269	0.9570	0.9623	+0.0633
Linear	3	4,118	1,478	548	0.1242	0.9563	0.9609	+0.0510
Linear	4	5,440	1,480	724	0.1024	0.9649	0.9709	+0.0590
Linear	5	6,762	1,482	751	0.1187	0.9561	0.9613	+0.0421
RBF	2	2,798	1,478	804	0.1150	0.9612	0.9663	+0.0656
RBF	3	4,121	1,481	441	0.1564	0.9423	0.9555	+0.0421
RBF	4	5,444	1,484	612	0.1145	0.9604	0.9679	+0.0499
RBF	5	6,767	1,487	648	0.1399	0.9552	0.9660	+0.0510

Table 8.5: MNIST dynamic-inference results for depth-doubled experts with top-1 path training. Each row reports one Router and expert-count experiment. ΔT1 Acc1 is measured against the end-to-end MoE baseline with the same router type and K.

Router	K	Params	T1 Params	Epochs	Loss	T1 Acc1	Dense Acc1	ΔT1 Acc1
Centroid	2	2,796	1,476	586	0.1346	0.9603	0.9603	+0.0563
Centroid	3	4,118	1,478	475	0.1493	0.9556	0.9556	+0.0396
Centroid	4	5,440	1,480	658	0.1042	0.9686	0.9686	+0.0570
Centroid	5	6,762	1,482	384	0.1791	0.9474	0.9474	+0.0509
Cosine	2	2,796	1,476	595	0.1472	0.9535	0.9535	+0.0559
Cosine	3	4,118	1,478	678	0.1240	0.9621	0.9621	+0.0647
Cosine	4	5,440	1,480	589	0.1274	0.9614	0.9614	+0.0734
Cosine	5	6,762	1,482	449	0.1444	0.9566	0.9566	+0.0610
Linear	2	2,796	1,476	762	0.1415	0.9560	0.9560	+0.0623
Linear	3	4,118	1,478	548	0.1460	0.9567	0.9567	+0.0514
Linear	4	5,440	1,480	712	0.1209	0.9631	0.9631	+0.0572
Linear	5	6,762	1,482	698	0.1187	0.9637	0.9637	+0.0497
RBF	2	2,798	1,478	701	0.1296	0.9616	0.9616	+0.0659
RBF	3	4,121	1,481	441	0.1559	0.9539	0.9539	+0.0538
RBF	4	5,444	1,484	612	0.1080	0.9680	0.9680	+0.0575
RBF	5	6,767	1,487	648	0.1589	0.9515	0.9515	+0.0472

sponding active path contains 1,480 parameters, approximately 112.7% above the baseline and lower than the 2,014-parameter width-expanded path.

The replacement stage averages 623.6 epochs under `tK` and 596.0 under `t1`. Both are lower than the 1010.6-epoch baseline average. Depth expansion gives the more favorable MNIST accuracy-to-parameter comparison because it reaches higher mean top-1 accuracy with fewer active parameters than width expansion. The shorter epoch count applies only to the replacement stage.

Similar to the increased-width experiments, the increased-depth experiments detail universal improvements, with also greatly accelerated training speed.

8.5.2 CIFAR-10

The matched CIFAR-10 MoE rows average 74.49% Acc1, 608.6 epochs, and 48,635 active top-1 parameters.

8.5.2.1 Width Expansion

Table 8.6: CIFAR-10 dynamic-inference results for width-doubled experts with dense-mixture training. ΔT1 Acc1 is measured against the end-to-end MoE baseline with the same router type and K. Dashes mark the unreported RBF result at $K = 2$.

Router	K	Params	T1 Params	Epochs	Loss	T1 Acc1	Dense Acc1	ΔT1 Acc1
Centroid	2	350k	177k	344	0.4423	0.8463	0.8514	+0.0889
Centroid	3	523k	178k	492	0.4293	0.8372	0.8549	+0.1034
Centroid	4	695k	178k	562	0.3962	0.8511	0.8717	+0.1136
Centroid	5	868k	178k	365	0.4983	0.8104	0.8341	+0.0763
Cosine	2	350k	177k	418	0.3954	0.8562	0.8654	+0.1131
Cosine	3	523k	178k	402	0.4243	0.8371	0.8567	+0.0916
Cosine	4	695k	178k	256	0.5163	0.7924	0.8354	+0.0644
Cosine	5	868k	178k	300	0.4967	0.7920	0.8401	+0.0694
Linear	2	350k	177k	405	0.4093	0.8584	0.8622	+0.0984
Linear	3	523k	178k	459	0.4102	0.8498	0.8638	+0.1039
Linear	4	695k	178k	427	0.4569	0.8323	0.8476	+0.0784
Linear	5	868k	178k	413	0.4991	0.8122	0.8354	+0.0731
RBF	2	–	–	–	–	–	–	–
RBF	3	523k	178k	453	0.4159	0.8506	0.8633	+0.0990
RBF	4	695k	178k	239	0.5866	0.7832	0.8037	+0.0137
RBF	5	868k	178k	673	0.3842	0.8567	0.8754	+0.1258

Table 8.7: CIFAR-10 dynamic-inference results for width-doubled experts with top-1 path training. ΔT1 Acc1 is measured against the end-to-end MoE baseline with the same router type and K.

Router	K	Params	T1 Params	Epochs	Loss	T1 Acc1	Dense Acc1	ΔT1 Acc1
Centroid	2	350k	177k	344	0.4373	0.8527	0.8527	+0.0953
Centroid	3	523k	178k	446	0.4616	0.8448	0.8448	+0.1109
Centroid	4	695k	178k	540	0.4286	0.8573	0.8573	+0.1198
Centroid	5	868k	178k	365	0.5073	0.8287	0.8287	+0.0946
Cosine	2	350k	177k	418	0.4398	0.8501	0.8501	+0.1070
Cosine	3	523k	178k	402	0.4524	0.8455	0.8455	+0.1000
Cosine	4	695k	178k	256	0.5560	0.8119	0.8119	+0.0838
Cosine	5	868k	178k	300	0.5880	0.7975	0.7975	+0.0750
Linear	2	350k	177k	405	0.4125	0.8615	0.8615	+0.1015
Linear	3	523k	178k	459	0.4378	0.8528	0.8528	+0.1068
Linear	4	695k	178k	427	0.4574	0.8458	0.8458	+0.0919
Linear	5	868k	178k	413	0.4704	0.8439	0.8439	+0.1048
RBF	2	350k	177k	396	0.4181	0.8597	0.8597	+0.0937
RBF	3	523k	178k	453	0.4126	0.8598	0.8598	+0.1082
RBF	4	695k	178k	239	0.5701	0.8086	0.8086	+0.0390
RBF	5	868k	178k	580	0.4150	0.8596	0.8596	+0.1287

Width expansion improves all 15 reported `tK` comparisons and all 16 `t1` comparisons. The `tK` rows average 83.11% top-1 Acc1 and 85.07% dense Acc1, compared with 74.35% for their matched baseline rows. Direct `t1` training averages 84.25% Acc1, compared with 74.49% for the full set of matched baselines.

The replacement stage averages 413.9 epochs under `tK`, compared with 598.7 for the corresponding baseline rows. Under `t1`, the averages are 402.7 and 608.6 epochs. The active path increases from approximately 48,635 to 178k parameters, an increase of 265.0%. Width expansion therefore raises the inference-path cost while improving accuracy and reducing the replacement-stage epoch count.

8.5.2.2 Depth Expansion

Depth expansion does not improve any of the 16 matched top-1 comparisons under either optimization mode. The `tK` rows average 65.53% top-1 Acc1 and 67.28% dense Acc1, while direct `t1` training averages 65.97%. Each is below the 74.49% baseline average.

The replacement stage averages 299.8 epochs under `tK` and 297.6 under `t1`, compared with 608.6 epochs for the baseline. The active path increases from approximately 48,635 to

Table 8.8: CIFAR-10 dynamic-inference results for depth-doubled experts with dense-mixture training. ΔT1 Acc1 is measured against the end-to-end MoE baseline with the same router type and K.

Router	K	Params	T1 Params	Epochs	Loss	T1 Acc1	Dense Acc1	ΔT1 Acc1
Centroid	2	190k	97,454	508	0.7956	0.7185	0.7220	-0.0389
Centroid	3	282k	97,462	328	0.9075	0.6676	0.6799	-0.0663
Centroid	4	375k	97,470	188	0.9743	0.6405	0.6544	-0.0970
Centroid	5	467k	97,478	263	0.9133	0.6562	0.6756	-0.0779
Cosine	2	190k	97,454	478	0.8723	0.6891	0.7016	-0.0540
Cosine	3	282k	97,462	219	1.0226	0.6349	0.6556	-0.1107
Cosine	4	375k	97,470	274	0.9170	0.6343	0.6949	-0.0937
Cosine	5	467k	97,478	289	1.0127	0.6130	0.6720	-0.1096
Linear	2	190k	97,454	382	0.8307	0.7027	0.7068	-0.0573
Linear	3	282k	97,462	398	0.8395	0.6926	0.7049	-0.0533
Linear	4	375k	97,470	156	1.0190	0.6206	0.6302	-0.1333
Linear	5	467k	97,478	178	0.9540	0.6466	0.6598	-0.0925
RBF	2	190k	97,456	539	0.7808	0.7215	0.7270	-0.0445
RBF	3	282k	97,465	390	0.8309	0.6984	0.7090	-0.0532
RBF	4	375k	97,474	113	1.0985	0.5942	0.6042	-0.1753
RBF	5	467k	97,483	94	1.1774	0.5534	0.5672	-0.1776

Table 8.9: CIFAR-10 dynamic-inference results for depth-doubled experts with top-1 path training. ΔT1 Acc1 is measured against the end-to-end MoE baseline with the same router type and K.

Router	K	Params	T1 Params	Epochs	Loss	T1 Acc1	Dense Acc1	ΔT1 Acc1
Centroid	2	190k	97,454	508	0.8233	0.7228	0.7228	-0.0346
Centroid	3	282k	97,462	328	0.9879	0.6682	0.6682	-0.0657
Centroid	4	375k	97,470	188	1.0433	0.6492	0.6492	-0.0884
Centroid	5	467k	97,478	263	1.0067	0.6613	0.6613	-0.0728
Cosine	2	190k	97,454	478	0.8953	0.6931	0.6931	-0.0499
Cosine	3	282k	97,462	219	1.0394	0.6414	0.6414	-0.1041
Cosine	4	375k	97,470	274	1.0662	0.6303	0.6303	-0.0977
Cosine	5	467k	97,478	289	1.0566	0.6330	0.6330	-0.0895
Linear	2	190k	97,454	382	0.8647	0.7085	0.7085	-0.0515
Linear	3	282k	97,462	398	0.9344	0.6883	0.6883	-0.0576
Linear	4	375k	97,470	156	1.1020	0.6289	0.6289	-0.1250
Linear	5	467k	97,478	178	1.0820	0.6395	0.6395	-0.0996
RBF	2	190k	97,456	520	0.8311	0.7197	0.7197	-0.0463
RBF	3	282k	97,465	374	0.9187	0.6935	0.6935	-0.0581
RBF	4	375k	97,474	113	1.1603	0.6093	0.6093	-0.1603
RBF	5	467k	97,483	94	1.2558	0.5685	0.5685	-0.1624

97,467 parameters, an increase of 100.4%. The shorter replacement stage is accompanied by lower accuracy.

8.5.2.3 Width and Depth Comparison

The two capacity changes produce different results under the same frozen routing partitions. Width expansion reaches 84.25% mean Acc1 under `t1`, while depth expansion reaches 65.97%. Width expansion uses approximately 178k active parameters, compared with 97,467 for depth expansion. For the selected CIFAR-10 architecture, increasing feature-map width is more effective than adding convolutional layers.

8.5.3 CIFAR-100

The matched CIFAR-100 MoE rows average 72.97% Acc1, 91.76% Acc5, 424.1 epochs, and 1,661k active top-1 parameters.

Table 8.10: CIFAR-100 dynamic-inference results for depth-doubled residual experts with dense-mixture training. Each row reports one Router and expert-count experiment. ΔT1 Acc1 is measured against the end-to-end MoE baseline with the same router type and K.

Router	K	Params	T1 Params	Epochs	Loss	T1 Acc1	Dense Acc1	T1 Acc5	Dense Acc5	ΔT1 Acc1
Centroid	2	6,248k	3,213k	223	0.9573	0.7257	0.7324	0.9136	0.9322	-0.0053
Centroid	3	9,284k	3,213k	168	1.1689	0.6673	0.6749	0.8850	0.9108	-0.0498
Centroid	4	12.3M	3,213k	151	1.2959	0.6273	0.6356	0.8646	0.8968	-0.1116
Centroid	5	15.4M	3,213k	25	1.6970	0.5193	0.5281	0.8003	0.8316	-0.2310
Cosine	2	6,248k	3,213k	285	0.8842	0.7462	0.7559	0.9196	0.9470	+0.0367
Cosine	3	9,284k	3,213k	186	1.2084	0.6631	0.6868	0.8666	0.9135	-0.0368
Cosine	4	12.3M	3,213k	137	1.5334	0.5897	0.6181	0.8248	0.8696	-0.1178
Cosine	5	15.4M	3,213k	143	1.5688	0.5614	0.6088	0.8008	0.8684	-0.1363
Linear	2	6,248k	3,213k	248	0.8533	0.7578	0.7612	0.9299	0.9456	+0.0176
Linear	3	9,284k	3,213k	203	1.0399	0.7007	0.7106	0.8965	0.9259	-0.0358
Linear	4	12.3M	3,213k	131	1.4247	0.5987	0.6047	0.8519	0.8742	-0.1492
Linear	5	15.4M	3,213k	101	1.5246	0.5662	0.5766	0.8285	0.8606	-0.1828
RBF	2	6,248k	3,213k	255	0.8826	0.7451	0.7488	0.9312	0.9424	+0.0073
RBF	3	9,284k	3,213k	263	0.8730	0.7471	0.7514	0.9296	0.9454	+0.0146
RBF	4	12.3M	3,213k	146	1.2487	0.6390	0.6440	0.8871	0.9027	-0.1032
RBF	5	15.4M	3,213k	102	1.6408	0.5337	0.5408	0.8205	0.8408	-0.2041

The depth-expanded residual experts average 64.93% top-1 Acc1 and 66.12% dense Acc1. Their mean top-1 and dense Acc5 values are 87.19% and 90.05%, compared with 91.76% for

the matched baseline. Top-1 Acc1 improves in 4 of the 16 matched rows, but the mean is below the baseline.

The replacement stage averages 172.9 epochs, compared with 424.1 for the baseline. The active path increases to 3,213k parameters, approximately 93.4% above the baseline. As on CIFAR-10, the reduced second-stage epoch count is accompanied by lower mean accuracy and a larger inference path.

8.6 Conclusions

The experiments support frozen-router expert replacement when the capacity change is compatible with the routed subproblems. On MNIST, both width and depth expansion improve every matched top-1 result and use fewer replacement-stage epochs than the end-to-end baseline.

CIFAR-10 distinguishes the two scaling choices. Width expansion improves every reported matched row, reaching 84.25% mean Acc1 under `t1` compared with 74.49% for the baseline. It also reduces the average replacement-stage duration from 608.6 to 402.7 epochs. Depth expansion reaches 65.97% and does not improve any matched row. The active path is larger in both cases, with width expansion carrying the greater parameter cost.

CIFAR-100 depth expansion gives mixed row-level results and lower mean accuracy. Across the three datasets, replacement-stage training is faster, but accuracy depends on how expert capacity is increased. The results support width expansion on MNIST and CIFAR-10 and depth expansion on MNIST. They also show that a pretrained router can be reused while experts with different capacities are trained for its existing partitions.

Therefore, the hypothesis is accepted. Upscaling individual models while maintaining a fixed router trained on a smaller, heterogeneous set is clearly a valid approach. There are some conditions, such as a proper upscaling strategy, that will impact results. However, overall, the methodology is demonstrated to work well.

Chapter 9

Conclusions

9.1 Overview

This dissertation investigates Mixture-of-Experts (MoE) neural architectures for small and mid-size image classification. A learned router partitions a problem space into subregions handled by specialized expert models.

The experimental framework evaluates classifier, autoencoder, and MoE models on MNIST, CIFAR-10, and CIFAR-100. Conventional classifiers provide strong baselines, while selected MoE configurations reach comparable or marginally improved results. Longer training produces clearer gains as experts are added, whereas increasing latent size alone has little effect. Router and expert convergence are therefore central to evaluating MoE scalability.

This work is organized around three research questions:

1. How can the interpretability of MoE systems be improved by analyzing the latent representations learned by routing mechanisms?

2. How can the training process of MoE architectures be optimized to reduce computational cost while preserving or improving expert performance?

3. How can routing strategies be extended to dynamically control computational depth and resource usage on a per-input basis?

Router latent spaces expose class structure, expert assignment, and routing confusion with clarity that varies by dataset complexity, router capacity, expert utilization, and pro-

jection method. Attention-based encoder–decoder routers provide direct input-level interpretation through spatial signals that emphasize image regions used by downstream experts. Their primary benefit is interpretability rather than consistent gains in accuracy or parameter efficiency. The first interpretability hypothesis is weakly accepted, with the second more strongly accepted.

Applying pre-trained classifiers and autoencoders as a router, for both frozen and unfrozen learning configurations, fails to converge faster reliably and typically lowers performance. Though results indicate some issue may be due to the approach, where the embedded experts struggle to update. The transfer-learning hypothesis, therefore, is not accepted.

Frozen-router expert replacement provides the clearest evidence for a useful MoE-specific capability. On MNIST, wider and deeper replacement experts improve every matched top-1 result and require fewer second-stage training epochs than the original end-to-end MoE runs. On CIFAR-10 and CIFAR-100, width scaling is substantially more effective than depth scaling. The dynamic-inference hypothesis is accepted: inherited routing partitions can support heterogeneous expert capacity when expansion is matched to the routed subproblem.

9.2 Latent Interpretability

The first major contribution is a framework for analyzing router representations and expert specialization. Router encodings are evaluated using SVD, t-SNE, and UMAP projections, together with routing frequencies and confusion behavior. These views make it possible to inspect whether samples are organized by class, which classes overlap, how expert regions are formed, and whether some experts are underused.

The latent-space results are mixed. MNIST autoencoders form clear class-correlated structure, while the smaller supervised routers can learn representations that are adequate for classification but visually compressed. CIFAR-10 and CIFAR-100 routers form stronger class- and expert-related structure than their reconstruction baselines despite using smaller

encoders and latent dimensions, although the resulting projections remain noisy. This contrast shows that reconstruction quality and routing interpretability are not equivalent objectives. A reconstruction model preserves information needed to reproduce an image, whereas a router is optimized to preserve information that helps divide the classification problem.

Learned latent units have no fixed physical scale, and expert embeddings can move without preserving a universal meaning for absolute distance. The projections are most useful for relative comparisons of cluster overlap, routing confusion, and expert utilization.

The first interpretability hypothesis is weakly accepted. Router latent spaces encode meaningful subproblem structure with strength determined by task complexity, router capacity, routing balance, and representation quality. The visualizations are most useful as diagnostics of cluster overlap, routing confusion, and expert utilization.

The attention-routing experiments provide a more direct form of interpretation. Encoder–decoder and UNet-style routers produce spatial outputs that are passed to the experts, which can also be visualized. These decoded signals emphasize portions of the image that are behaviorally connected to classification. Concatenating the decoded signal with the original image is generally more effective than additive modification on MNIST because it preserves the original channels while exposing the routed representation separately.

Attention routing primarily improves interpretation. MNIST attention models improve accuracy with substantially larger active paths, while the tested CIFAR-10 and CIFAR-100 models use modestly larger active paths and achieve lower descriptive top-1 accuracy than matched MoE baselines. The second interpretability hypothesis is accepted because the architecture consistently exposes meaningful spatial information that participates in inference.

9.3 Transferring Autoencoder Learning to Routers

The second research area evaluates whether pretrained encoders can reduce MoE training cost by supplying an initial routing representation. Both supervised classifier encoders and reconstruction-oriented autoencoder encoders are transferred, and the router is either allowed to adapt or held frozen.

Adaptive transfer is consistently more reliable than frozen transfer. A trainable router retains useful source features while adjusting its partition to expert specialization. Frozen routers generally produce lower accuracy, greater imbalance, or both. However, this may be due to frozen routers receiving limited updates to the embedded experts.

Adaptive classifier transfer on CIFAR-10 is the only reported condition with a lower mean epoch count than its matched end-to-end baseline, decreasing from 608.6 to 582.7 epochs. The remaining adaptive and frozen conditions are equal to or slower than their baselines, and frozen transfer is slower on average in every reported source–dataset comparison.

Thus, this research fails to support the transfer-learning hypothesis. Future transfer learning approaches should explore how to also effectively learn the embedded experts alongside the router, or identify how to update them while training more freely.

9.4 Dynamic Inference

The third contribution is a dynamic-inference strategy based on frozen-router reuse and variable expert capacity. A trained router preserves the learned subproblem boundaries while replacement experts are trained for the samples assigned to each route. Once the router is frozen, expert capacity can be changed without retraining the router. Selective replacement can therefore increase capacity on particular routes without increasing the active cost of every inference path.

MNIST provides the strongest support for this approach. Both width and depth ex-

pansion improve every matched top-1 comparison. Under direct top-1 training, the depth-expanded experts average 95.87% accuracy, compared with 90.23% for the matched end-to-end MoE baselines. Their replacement stage averages 596.0 epochs, compared with 1,010.6 epochs for the complete baseline runs. Depth expansion also uses approximately 1,480 active parameters, compared with 2,014 for width expansion, while producing the higher mean accuracy.

The CIFAR-10 results distinguish between width and depth expansion. Width-expanded experts improve all 16 matched comparisons under direct top-1 training, averaging 84.25% accuracy compared with 74.49% for the baseline. Their replacement stage averages 402.7 epochs, compared with 608.6 epochs for the baseline, while the active path increases from 48,635 to approximately 178k parameters. Depth-expanded experts average 65.97% accuracy and do not improve any matched comparison, despite a shorter replacement stage of 297.6 epochs and a smaller active path of 97,467 parameters.

CIFAR-100 depth expansion produces mixed row-level results and lowers the descriptive mean. The depth-expanded residual experts average 64.93% top-1 accuracy, compared with 72.97% for the matched baselines, and improve 4 of the 16 matched rows. Their replacement stage averages 172.9 epochs, compared with 424.1 epochs for the baseline, while the active path increases from approximately 1,661k to 3,213k parameters.

The dynamic-inference hypothesis is accepted. Frozen-router reuse permits larger experts to be trained in fewer replacement-stage epochs, but the accuracy benefit depends on the dataset and the form of capacity expansion. Width expansion is effective on MNIST and CIFAR-10, while depth expansion is effective on MNIST but not on the evaluated CIFAR-10 and CIFAR-100 configurations.

9.5 Future Work

The baseline and scalability results motivate longer and more structured MoE training studies. Additionally, identifying how to perform an early-stopping check better will be critical for long-running training to reduce required computational resources.

Latent interpretability would benefit from regularization that makes representations more stable across random seeds. These methods should be evaluated in the original latent space in addition to two-dimensional projections. Intervention-based tests, such as perturbing latent coordinates or decoded spatial signals and measuring changes in route selection, would also provide stronger causal evidence. Additionally, binding the latent spaces to some consistent unit would further improve interpretability, even across different runs, as distances would provide more well-defined relationships.

Attention routing should be revisited with lighter architectures. Vision-transformer or patch-based encoders may expose spatial importance without the full cost of a convolutional encoder–decoder. Additional work on establishing causal behaviors would further improve reliability, confidence, and overall interpretability.

The transfer-learning results suggest learning embedded experts alongside the pre-trained routers. This may also provide a means for dynamic expert sizing, allowing an intelligent decision about how many experts to emplace within the system. Reversing the transfer direction is also worth testing: a supervised router may provide an initialization for an autoencoder whose latent space preserves class distinction more effectively than a reconstruction-only baseline.

Dynamic inference should be extended from manual replacement to automatic capacity allocation. A controller could use expert-local loss, calibration error, routing frequency, latency, and memory constraints to determine which route should be widened, deepened, split, or left unchanged. Additionally, an interesting variation of the methodology can also include context-switching modes, where how fast an agent needs to respond can be factored

into which set of models to use.

9.6 Final Remarks

MoE provides a modular representation of a problem space that can be inspected, transferred, and selectively expanded. MoE systems remain worthwhile when their modularity directly serves the application: when subproblems differ in difficulty, when expert-local behavior must be examined, when routes can be reused, or when deployment constraints reward selective computation.

This dissertation contributes reproducible classifier, autoencoder, and MoE baselines; a model-factory framework for defining the tested architectures; latent and spatial methods for interpreting routing behavior; a comparative evaluation of classifier- and autoencoder-derived router transfer; and a dynamic-inference method based on frozen-router expert replacement. Routing partitions can be reused to train heterogeneous experts, while router pretraining alone provides limited reductions in total training cost.

Reference List

Abdu-Aguye, M. G., W. Gomaa, Y. Makihara, and Y. Yagi, 2020: Adaptive pooling is all you need: An empirical study on hyperparameter-insensitive human action recognition using wearable sensors. *2020 International Joint Conference on Neural Networks (IJCNN)*, 1–6, doi:10.1109/IJCNN48605.2020.9207082.

Aitken, A., C. Ledig, L. Theis, J. Caballero, Z. Wang, and W. Shi, 2017: Checkerboard artifact free sub-pixel convolution: A note on sub-pixel convolution, resize convolution and convolution resize. URL https://arxiv.org/abs/1707.02937, 1707.02937.

Ali, H. A., C. Mohamed, B. Abdelhamid, N. Ourdani, and T. E. Alami, 2022: A comparative evaluation use bagging and boosting ensemble classifiers. *2022 International Conference on Intelligent Systems and Computer Vision (ISCV)*, 1–6, doi:10.1109/ISCV54655.2022.9806080.

Arulkumaran, K., M. P. Deisenroth, M. Brundage, and A. A. Bharath, 2017: Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, **34 (6)**, 26–38, doi:10.1109/MSP.2017.2743240.

Bellalta, B., and Coauthors, 2024: Artificial Intelligence and Machine Learning. IEEE International Network Generations Roadmap – 2024 Edition, IEEE Future Networks. URL https://futurenetworks.ieee.org/images/files/pdf/INGR_2024_Edition/IEEE_INGR_AIML_2024-Edition-FINAL.pdf, edited by Baw Chng.

Bishop, C. M., and H. Bishop, 2024: *Deep Learning: Foundations and Concepts*. Springer Cham, doi:10.1007/978-3-031-45468-4, URL https://doi.org/10.1007/978-3-031-45468-4.

Bo, L., C. Sminchisescu, A. Kanaujia, and D. Metaxas, 2008: Fast algorithms for large scale conditional 3d prediction. *2008 IEEE Conference on Computer Vision and Pattern Recognition*, 1–8, doi:10.1109/CVPR.2008.4587578.

Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone, 1984: *Classification and Regression Trees*. 1st ed., Chapman and Hall/CRC, doi:10.1201/9781315139470.

Broomhead, D. S., and D. Lowe, 1988: Multivariable functional interpolation and adaptive networks. *Complex Syst.*, **2**, URL https://api.semanticscholar.org/CorpusID:3686496.

Brown, T., and Coauthors, 2020: Language models are few-shot learners. *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., Curran Associates, Inc., Vol. 33, 1877–1901, URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Cai, W., J. Jiang, F. Wang, J. Tang, S. Kim, and J. Huang, 2025: A survey on mixture of experts in large language models. *IEEE Transactions on Knowledge and Data Engineering*, **37 (7)**, 3896–3915, doi:10.1109/TKDE.2025.3554028.

Chauhan, R., K. K. Ghanshala, and R. Joshi, 2018: Convolutional neural network (cnn) for image detection and recognition. *2018 First International Conference on Secure Cyber Computing and Communication (ICSCCC)*, 278–282, doi:10.1109/ICSCCC.2018.8703316.

Che, Y., 2025: Real-time object detection and tracking using deep learning in autonomous driving systems. *2025 Asia Conference on Energy Conversion Systems and Power Electronics (AECSPE)*, 275–279, doi:10.1109/AECSPE66597.2025.00052.

Cheng, Y., D. Wang, P. Zhou, and T. Zhang, 2018: Model compression and acceleration for deep neural networks: The principles, progress, and challenges. *IEEE Signal Process. Mag.*, **35 (1)**, 126–136, doi:10.1109/MSP.2017.2765695, URL https://doi.org/10.1109/MSP.2017.2765695.

Cătrună, A.-E., and D.-T. Iancu, 2023: Automated neural network design using genetic algorithms. *2023 24th International Conference on Control Systems and Computer Science (CSCS)*, 113–120, doi:10.1109/CSCS59211.2023.00027.

Dolinger, G. M., 2025: Scalable multi-agent collaboration for radar tasks. URL https://sh areok.org/server/api/core/bitstreams/50bab2e0-a4bd-4328-aeea-3cb8560e0a4e/content.

Dumoulin, V., and F. Visin, 2018: A guide to convolution arithmetic for deep learning. URL https://arxiv.org/abs/1603.07285, 1603.07285.

Duta, I. C., L. Liu, F. Zhu, and L. Shao, 2021: Improved residual networks for image and video recognition. *2020 25th International Conference on Pattern Recognition (ICPR)*, 9415–9422, doi:10.1109/ICPR48806.2021.9412193.

Ergen, T., A. Sahiner, B. Ozturkler, J. Pauly, M. Mardani, and M. Pilanci, 2022: Demystifying batch normalization in relu networks: Equivalent convex optimization models and implicit regularization. URL https://arxiv.org/abs/2103.01499, 2103.01499.

Fahmi, A., A. Khan, D. Maqbool, and T. Abdeljawad, 2025: Circular intuitionistic fuzzy hamacher aggregation operators for multi-attribute decision-making. *Scientific Reports*, **15**, doi:10.1038/s41598-025-88845-0.

Falcon, W., and The PyTorch Lightning team, 2019: PyTorch Lightning. URL https://ww w.pytorchlightning.ai, doi:10.5281/zenodo.3828935.

Floreano, D., and C. Mattiussi, 2008: *Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies*. Intelligent Robotics and Autonomous Agents series, MIT Press, URL https://books.google.com/books?id=DSbgAAAAMAAJ.

Garouani, M., J. Mothe, A. Barhrhouj, and J. Aligon, 2024: Investigating the duality of interpretability and explainability in machine learning. *2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI)*, 861–867, doi:10.1109/ICTAI62512.2024.00125.

Goodfellow, I., Y. Bengio, and A. Courville, 2016: *Deep Learning*. MIT Press, http://www.deeplearningbook.org.

Gupta, A., R. Ramanath, J. Shi, and S. S. Keerthi, 2021: Adam vs. sgd: Closing the generalization gap on image classification. *OPT2021: 13th Annual Workshop on Optimization for Machine Learning*, URL https://api.semanticscholar.org/CorpusID:265106335.

Hampshire, J., and A. Waibel, 1992: The meta-pi network: building distributed knowledge representations for robust multisource pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **14 (7)**, 751–769, doi:10.1109/34.142911.

Hartmann, T., A. Moawad, C. Schockaert, F. Fouquet, and Y. Le Traon, 2019: Meta-modelling meta-learning. *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 300–305, doi:10.1109/MODELS.2019.00014.

He, K., X. Zhang, S. Ren, and J. Sun, 2016: Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778, doi:10.1109/CVPR.2016.90.

Hinton, G. E., N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, 2012: Improving neural networks by preventing co-adaptation of feature detectors. URL https://arxiv.org/abs/1207.0580, 1207.0580.

Hochreiter, S., and J. Schmidhuber, 1997: Long short-term memory. *Neural Computation*, **9 (8)**, 1735–1780, doi:10.1162/neco.1997.9.8.1735, URL https://doi.org/10.1162/neco.1997.9.8.1735, https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf.

Hoyt, Z., D. F. Hougen, B. Lavender, Z. Kastl, L. Golston, S. Nguyen, and J. Karch, 2025: Generalizing distributions of learned error and bias using empirical analysis. *2025 International Conference on Machine Learning and Applications (ICMLA)*, 588–594, doi:10.1109/ICMLA66185.2025.00086.

Huang, H., S. Lu, Y. Shan, H. Qu, F. Zhang, W. Guan, Q. Hong, and L. Li, 2025: Dynamic language group-based moe: Enhancing code-switching speech recognition with hierarchical routing. *ICASSP 2025 - 2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 1–5, doi:10.1109/ICASSP49660.2025.10890805.

Ioffe, S., and C. Szegedy, 2015a: Batch normalization: Accelerating deep network training

by reducing internal covariate shift. *Proceedings of the 32nd International Conference on Machine Learning*, F. Bach, and D. Blei, Eds., PMLR, Lille, France, Proceedings of Machine Learning Research, Vol. 37, 448–456, URL https://proceedings.mlr.press/v37/ioffe15.html.

Ioffe, S., and C. Szegedy, 2015b: Batch normalization: Accelerating deep network training by reducing internal covariate shift. URL https://arxiv.org/abs/1502.03167, 1502.03167.

Jacobs, R. A., M. I. Jordan, S. J. Nowlan, and G. E. Hinton, 1991: Adaptive mixtures of local experts. *Neural Computation*, **3 (1)**, 79–87, doi:10.1162/neco.1991.3.1.79.

Ji, C., 2024: A survey of neural network optimization algorithms. *2024 IEEE 4th International Conference on Data Science and Computer Application (ICDSCA)*, 1–7, doi:10.1109/ICDSCA63855.2024.10859435.

Jiang, F., C. Pan, L. Dong, K. Wang, M. Debbah, D. Niyato, and Z. Han, 2026: A comprehensive survey of large ai models for future communications: Foundations, applications, and challenges. *IEEE Communications Surveys Tutorials*, **28**, 4731–4764, doi:10.1109/COMST.2026.3660844.

Jordan, M., and R. Jacobs, 1993: Hierarchical mixtures of experts and the em algorithm. *Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan)*, Vol. 2, 1339–1344 vol.2, doi:10.1109/IJCNN.1993.716791.

Kaur, S., and G. Bawa, 2022: Learning robotic skills through reinforcement learning. *2022 3rd International Conference on Electronics and Sustainable Communication Systems (ICESC)*, 903–908, doi:10.1109/ICESC54411.2022.9885704.

Khade, R., K. Jariwala, and C. Chattopadhyay, 2023: Autoencoders in graphical document retrieval: Cost function influence and rotation impact. *2023 International Conference on Modeling, Simulation Intelligent Computing (MoSICom)*, 240–245, doi:10.1109/MoSICom59118.2023.10458779.

Kingma, D. P., and J. Ba, 2017: Adam: A method for stochastic optimization. URL https://arxiv.org/abs/1412.6980, 1412.6980.

Kingma, D. P., and M. Welling, 2019: An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, **12 (4)**, 307–392, doi:10.1561/2200000056, URL http://dx.doi.org/10.1561/2200000056.

Krizhevsky, A., 2009: Learning multiple layers of features from tiny images. Tech. rep., University of Toronto. URL https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

Kullback, S., and R. A. Leibler, 1951: On information and sufficiency. *The annals of math-*

ematical statistics, **22 (1)**, 79–86.

Kumar, S. P., M. Diwakar, J. S, P. Arthi, V. S. Pandi, and S. D, 2024: The application of machine learning to natural language processing: Modern advances in the study of human language. *2024 International Conference on Cybernation and Computation (CYBERCOM)*, 561–566, doi:10.1109/CYBERCOM63683.2024.10803211.

LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner, 1998: Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, **86 (11)**, 2278–2324, doi:10.1109/5.726791.

Li, J., S. Tripathi, L. Rastogi, Y. Lei, R. Pan, and Y. Xia, 2026: Optimizing mixture-of-experts inference time via model deployment and communication scheduling. *IEEE Transactions on Networking*, **34**, 2478–2497, doi:10.1109/TON.2025.3645806.

Li, S., D. Li, and Y. Zhang, 2021: Incorporating nesterov's momentum into distributed adaptive gradient method for online optimization. *2021 China Automation Congress (CAC)*, 7338–7343, doi:10.1109/CAC53003.2021.9727247.

Lin, H.-Y., 2022: Large-scale artificial intelligence models. *Computer*, **55 (5)**, 76–80, doi:10.1109/MC.2022.3151419.

Lin, M., Q. Chen, and S. Yan, 2014: Network in network. URL https://arxiv.org/abs/1312.4400, 1312.4400.

Liu, L., H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han, 2021: On the variance of the adaptive learning rate and beyond. URL https://arxiv.org/abs/1908.03265, 1908.03265.

Loshchilov, I., and F. Hutter, 2019: Decoupled weight decay regularization. URL https://arxiv.org/abs/1711.05101, 1711.05101.

Luo, L., Y. Xiong, Y. Liu, and X. Sun, 2019: Adaptive gradient methods with dynamic bound of learning rate. URL https://arxiv.org/abs/1902.09843, 1902.09843.

Lv, B., and Coauthors, 2026: Graphmoe: Amplifying cognitive depth of mixture-of-experts network via introducing self-rethinking mechanism. URL https://arxiv.org/abs/2501.07890, 2501.07890.

Malik, A., B. Attal, A. Xie, M. O'Toole, and D. B. Lindell, 2025: Neural inverse rendering from propagating light. *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 10 534–10 544, doi:10.1109/CVPR52734.2025.00985.

Manataki, M., A. Vafidis, and A. Sarris, 2021: Comparing adam and sgd optimizers to train alexnet for classifying gpr c-scans featuring ancient structures. *2021 11th International Workshop on Advanced Ground Penetrating Radar (IWAGPR)*, 1–6,

doi:10.1109/IWAGPR50767.2021.9843162.

McInnes, L., J. Healy, and J. Melville, 2020: Umap: Uniform manifold approximation and projection for dimension reduction. URL https://arxiv.org/abs/1802.03426, 1802.03426.

Mi, Z., Y. Chen, P. Zhao, X. Yu, H. Wang, Y. Wang, and S. Huang, 2026: Effective moe-based llm compression by exploiting heterogeneous inter-group experts routing frequency and information density. doi:10.48550/arXiv.2602.09316.

Mishkin, D., N. Sergievskiy, and J. Matas, 2017: Systematic evaluation of convolution neural network advances on the imagenet. *Computer Vision and Image Understanding*, doi:https://doi.org/10.1016/j.cviu.2017.05.007, URL http://www.sciencedirect.com/science/article/pii/S1077314217300814.

Naeem, H., J. Ahmad, and M. Tayyab, 2013: Real-time object detection and tracking. *INMIC*, 148–153, doi:10.1109/INMIC.2013.6731341.

Nagpal, P., S. A. Bhinge, and A. Shitole, 2022: A comparative analysis of resnet architectures. *2022 International Conference on Smart Generation Computing, Communication and Networking (SMART GENCON)*, 1–8, doi:10.1109/SMARTGENCON56628.2022.10083966.

Nguyen, H., P. Akbarian, T. Pham, T. Nguyen, S. Zhang, and N. Ho, 2025: Statistical advantages of perturbing cosine router in mixture of experts. URL https://arxiv.org/abs/2405.14131, 2405.14131.

Odena, A., V. Dumoulin, and C. Olah, 2016: Deconvolution and checkerboard artifacts. *Distill*, doi:10.23915/distill.00003, URL http://distill.pub/2016/deconv-checkerboard.

Paszke, A., and Coauthors, 2017: Automatic differentiation in pytorch. *NIPS-W*.

Peebles, W., and S. Xie, 2023: Scalable diffusion models with transformers. URL https://arxiv.org/abs/2212.09748, 2212.09748.

Python Software Foundation, 2021: Python 3.10. URL https://www.python.org/downloads/release/python-3100/, programming language.

Rafiq, M., S. S. Parihar, Y. S. Chauhan, and S. Sahay, 2022: Efficient implementation of max-pooling algorithm exploiting history-effect in ferroelectric-finfets. *IEEE Transactions on Electron Devices*, **69 (11)**, 6446–6452, doi:10.1109/TED.2022.3207114.

Razghandi, M., H. Zhou, M. Erol-Kantarci, and D. Turgut, 2022: Variational autoencoder generative adversarial network for synthetic data generation in smart home. *ICC 2022 - IEEE International Conference on Communications*, 4781–4786, doi:10.1109/ICC45855.2022.9839249.

Reddi, S. J., S. Kale, and S. Kumar, 2018: On the convergence of adam and beyond. *International Conference on Learning Representations*, URL https://openreview.net/forum?id=ryQu7f-RZ.

Rincy, T. N., and R. Gupta, 2020: Ensemble learning techniques and its efficiency in machine learning: A survey. *2nd International Conference on Data, Engineering and Applications (IDEA)*, 1–6, doi:10.1109/IDEA49133.2020.9170675.

Ruder, S., 2017: An overview of gradient descent optimization algorithms. URL https://arxiv.org/abs/1609.04747, 1609.04747.

Russell, S., and P. Norvig, 2020: *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, URL http://aima.cs.berkeley.edu/.

Räuker, T., A. Ho, S. Casper, and D. Hadfield-Menell, 2023: Toward transparent ai: A survey on interpreting the inner structures of deep neural networks. *2023 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, 464–483, doi:10.1109/SaTML54575.2023.00039.

Sang, Y., and Coauthors, 2025: Stream normalization for ctr prediction. *Proceedings of the Nineteenth ACM Conference on Recommender Systems*, Association for Computing Machinery, New York, NY, USA, 1086–1090, RecSys '25, doi:10.1145/3705328.3748093, URL https://doi.org/10.1145/3705328.3748093.

Sharda, J., P.-K. Hsu, and S. Yu, 2025: Training trillion-parameter llms with 3-d dram-based accelerator and co-packaged optical interconnects. *IEEE Transactions on Components, Packaging and Manufacturing Technology*, **15 (11)**, 2282–2289, doi:10.1109/TCPMT.2025.3613880.

Sharma, H., 2021: Improving natural language processing tasks by using machine learning techniques. *2021 5th International Conference on Information Systems and Computer Networks (ISCON)*, 1–5, doi:10.1109/ISCON52037.2021.9702447.

Shazeer, N., A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, 2017: Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. URL https://arxiv.org/abs/1701.06538, 1701.06538.

Siddique, N., S. Paheding, C. P. Elkin, and V. Devabhaktuni, 2021: U-net and its variants for medical image segmentation: A review of theory and applications. *IEEE Access*, **9**, 82 031–82 057, doi:10.1109/ACCESS.2021.3086020.

Sierra, R. L., G. Esposito, J.-D. Guerrero-Balaguera, J. E. R. Condia, and M. S. Reorda, 2025: Improving cnn runtime robustness against soft errors by dropout layer optimization. *2025 IEEE 26th Latin American Test Symposium (LATS)*, 1–6, doi:10.1109/LATS65346.2025.10963963.

Simonyan, K., and A. Zisserman, 2015: Very deep convolutional networks for large-scale image recognition. URL https://arxiv.org/abs/1409.1556, 1409.1556.

Singh, Y., M. Saini, and Savita, 2023: Impact and performance analysis of various activation functions for classification problems. *2023 IEEE International Conference on Contemporary Computing and Communications (InC4)*, Vol. 1, 1–7, doi:10.1109/InC457730.2023.10263129.

Soumyadeep, M., A. P. Singh, A. Sharma, and P. Kumar, 2023: Real-time object detection and tracking using deep learning. *2023 11th International Conference on Intelligent Systems and Embedded Design (ISED)*, 1–7, doi:10.1109/ISED59382.2023.10444534.

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, 2014: Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, **15 (56)**, 1929–1958, URL http://jmlr.org/papers/v15/srivastava14a.html.

Stringer, A., G. Dolinger, D. Hogue, L. Schley, and J. G. Metcalf, 2024: A metacognitive approach to adaptive radar detection. *IEEE Transactions on Aerospace and Electronic Systems*, **60 (1)**, 168–185, doi:10.1109/TAES.2023.3274101.

Suvarna, G., S. A. Khan, M. Garg, G. Karthikeyan, J. Giri, and M. Y. Al-Safarini, 2026: Machine learning and the future of ai: Revolutionizing industries with data-driven innovations. *2026 3rd International Conference on Emerging Trends in Engineering and Medical Sciences (ICETEMS)*, 1–7, doi:10.1109/ICETEMS66917.2026.11469719.

Sára Pusztaházi, L., G. Eigner, and O. Csiszár, 2024: Parametric activation functions for neural networks: A tutorial survey. *IEEE Access*, **12**, 168 626–168 644, doi:10.1109/ACCESS.2024.3474574.

Taylor, A., I. Dusparic, E. Galván-López, S. Clarke, and V. Cahill, 2014: Accelerating learning in multi-objective systems through transfer learning. *2014 International Joint Conference on Neural Networks (IJCNN)*, 2298–2305, doi:10.1109/IJCNN.2014.6889438.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, 2017: Attention is all you need. *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., Curran Associates, Inc., Vol. 30, URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Vettoruzzo, A., M.-R. Bouguelia, J. Vanschoren, T. Rögnvaldsson, and K. Santosh, 2024: Advances and challenges in meta-learning: A technical review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **46 (7)**, 4763–4779, doi:10.1109/TPAMI.2024.3357847.

Virtanen, P., and Coauthors, 2020: SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, **17**, 261–272, doi:10.1038/s41592-019-0686-2.

Wang, D., A. Qiu, Q. Zhou, and H. D. Schotten, 2025a: A survey on the role of artificial intelligence and machine learning in 6g-v2x applications. *2025 IEEE 11th World Forum on Internet of Things (WF-IoT)*, 1–7, doi:10.1109/WF-IoT64238.2025.11270555.

Wang, J., M. Chen, N. Karaev, A. Vedaldi, C. Rupprecht, and D. Novotny, 2025b: Vggt: Visual geometry grounded transformer. *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 5294–5306, doi:10.1109/CVPR52734.2025.00499.

Wang, Z., and Coauthors, 2025c: Chain-of-experts: Unlocking the communication power of mixture-of-experts models. URL https://arxiv.org/abs/2506.18945, 2506.18945.

Welch, S., S. Baskin, and P. Gundu, 2025: *The Welch Labs Illustrated Guide to AI*. No. v. 1, The Welch Labs Illustrated Guide to AI, Welch Labs LLC, URL https://books.google.com/books?id=W8yb0QEACAAJ.

Wu, H., and X. Gu, 2015: Towards dropout training for convolutional neural networks. *Neural Networks*, **71**, 1–10, doi:10.1016/j.neunet.2015.07.007, URL http://dx.doi.org/10.1016/j.neunet.2015.07.007.

Yang, X., C. Venhoff, A. Khakzar, C. S. de Witt, P. K. Dokania, A. Bibi, and P. Torr, 2025: Mixture of experts made intrinsically interpretable. URL https://arxiv.org/abs/2503.07639, 2503.07639.

Yep, T., 2020: torchinfo. URL https://github.com/TylerYep/torchinfo.

You, Y., and Coauthors, 2020: Large batch optimization for deep learning: Training bert in 76 minutes. URL https://arxiv.org/abs/1904.00962, 1904.00962.

Yu, Z., S. Dai, and Y. Xing, 2019: Adaptive salience preserving pooling for deep convolutional neural networks. *2019 IEEE International Conference on Multimedia Expo Workshops (ICMEW)*, 513–518, doi:10.1109/ICMEW.2019.00094.

Yuksel, S. E., J. N. Wilson, and P. D. Gader, 2012: Twenty years of mixture of experts. *IEEE Transactions on Neural Networks and Learning Systems*, **23 (8)**, 1177–1193, doi:10.1109/TNNLS.2012.2200299.

Zhai, J., S. Zhang, J. Chen, and Q. He, 2018: Autoencoder and its various variants. *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 415–419, doi:10.1109/SMC.2018.00080.

Zhang, C.-B., P.-T. Jiang, Q. Hou, Y. Wei, Q. Han, Z. Li, and M.-M. Cheng, 2021a: Delving deep into label smoothing. *IEEE Transactions on Image Processing*, **30**, 5984–5996, doi:10.1109/TIP.2021.3089942.

Zhang, Y., P. Tiňo, A. Leonardis, and K. Tang, 2021b: A survey on neural network inter-

pretability. *IEEE Transactions on Emerging Topics in Computational Intelligence*, **5 (5)**, 726–742, doi:10.1109/TETCI.2021.3100641.

Zhang, Z., 2018: Improved adam optimizer for deep neural networks. *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, 1–2, doi:10.1109/IWQoS.2018.8624183.

Zhou, Y., and R. Chellappa, 1988: Computation of optical flow using a neural network. *IEEE 1988 International Conference on Neural Networks*, 71–78 vol.2, URL https://api.semanticscholar.org/CorpusID:7292956.

Zhuang, F., Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, 2020a: A comprehensive survey on transfer learning. URL https://arxiv.org/abs/1911.02685, 1911.02685.

Zhuang, J., T. Tang, Y. Ding, S. Tatikonda, N. Dvornek, X. Papademetris, and J. S. Duncan, 2020b: Adabelief optimizer: Adapting stepsizes by the belief in observed gradients. URL https://arxiv.org/abs/2010.07468, 2010.07468.