UNIVERSITY OF OKLAHOMA
GRADUATE COLLEGE

EVALUATING GAM-LIKE NEURAL NETWORK ARCHITECTURES FOR INTERPRETABLE MACHINE LEARNING

A THESIS
SUBMITTED TO THE GRADUATE FACULTY
in partial fulfillment of the requirements for the
Degree of
MASTER OF SCIENCE

By

WILLIAM BOOKER
Norman, Oklahoma
2019

EVALUATING GAM-LIKE NEURAL NETWORK ARCHITECTURES FOR INTERPRETABLE MACHINE LEARNING

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Dr. Amy McGovern, Chair

Dr. Dean Hougen

Dr. Christan Grant

Acknowledgements

I'd like to thank my committee chair, Dr. McGovern, for taking me under her wing and introducing me into the incredible world of machine learning. For keeping me on track, reviewing countless drafts (that I often provided five minutes before the deadline), and giving me every opportunity to succeed, I could not have asked for a better advisor.

I'd like to thank committee member, Dr. Hougen, his support over the years and for introducing me to academic research. When I came to OU four years ago, all I wanted to do was publish a paper and you went out of your way to give me that opportunity. Rio 2018! I still owe you a round of frisbee golf.

I'd like to thank Dr. Grant, for serving on this committee and providing a source of inspiration over the years. Hobby Robbie may not have been my most successful project, but it taught me a lot, and inspired me to push forward with my degree in Computer Science.

I'd like to thank my family, my dad Garry, my mom Lisa, and my brother Alex, for giving me this opportunity to learn and grow and for supporting me through all the ups and downs. I really couldn't have done it without you all.

Finally, I want to thank my best friends, Gabe, Trevor, and Catherine for looking out for me and keeping me sane throughout this whole process. Seriously, you all are the best.

Table of Contents

List Of Figures

Abstract

In many machine learning applications, interpretability is of the utmost importance. Artificial intelligence is proliferating, but before you entrust your finances, your well-being, or even your life to a machine, you'd really like to be sure that it knows what it's doing.

As a human, the best way to evaluate an algorithm is to pick it apart, understand how it works, and figure out how it arrives at the decisions it does. Unfortunately, as machine learning techniques become more powerful and more complicated, reverse-engineering is becoming more difficult. Engineers often choose to implement a model that is accurate rather than one that understandable. In this work, we demonstrate a novel technique that, in certain circumstances, can be both.

This work introduces a novel neural network architecture that improves interpretability without sacrificing model accuracy. We test this architecture in a number of real-world classification datasets and demonstrate that it performs almost identically to state-of-the-art methods. We introduce *Pandemic*, a novel image classification benchmark, to demonstrate that our architecture has further applications in deep-learning models.

Chapter 1

Introduction

In machine learning, increased accuracy is often paired with increased complexity. As technology continues to improve, new methods are being proposed to take advantage of recent advances in hardware and mathematics. Deep learning is supplanting shallow networks in fields such as image recognition (Szegedy et al. 2015) and machine translation (Bahdanau et al. 2014) while gradient boosted machines are replacing random forests in many tabular classification tasks (Friedman 2001). However, as machine learning techniques continue to improve, it is becoming increasingly difficult to understand each model's inner workings. While in some applications this opacity is not an issue, in high-risk sectors such as medicine, law, and finance, model justifiability remains an extremely important feature (Hastie et al. 2009).

Over the years, numerous techniques have been proposed to help unmask the underlying reasoning behind machine learning models. Some model-specific proposals take advantage of the method's underlying structure, such as impurity importance in tree-based methods (Louppe et al. 2013) or network pruning in artificial neural networks (Kingston et al. 2004). Model-agnostic methods have also been proposed, such as partial dependence plots (Friedman 2001), sequential selection (Zou and Hastie 2005), and permutation importance (Breiman et al. 1984) to help determine the effect of one input on the model's predicted

output. While these techniques help with improving interpretability, none succeed in fully exposing the complete inner working of these complex techniques.

A different approach is to create a model that's interpretable by design. The earliest machine learning methods, ordinary least squares linear regression (Galton 1886) and decision trees (Belson 1959), are good examples of this paradigm. While both techniques have severe limitations such as brittleness (Norton 1989) and strong assumptions (Hayashi 2000), both algorithms are still in common use, largely due to their simplicity and interpretability. Statistics research has focused heavily on this paradigm, developing extensions to linear regression such as weighted least squares (Suárez et al. 2017) and logistic regression (Nelder and Wedderburn 1972) to help address some of the concerns with ordinary least squares. These avenues have since coalesced into two paradigms, Generalized Linear Models (GLMs) (Nelder and Wedderburn 1972) and Generalized Additive Models (GAMs) (Hastie and Tibshirani 2007), which strike a nice balance between power and interpretability, but come with their own set of drawbacks, such as complex domain-specific parametric tuning and little implementation support from the machine learning community.

This thesis proposes a novel machine learning method that fuses the natural interpretability of GAMs with the flexibility and support of neural networks. We have developed the first known open source GAM-like implementation in a neural network framework and demonstrated that the network's natural flexibility allows the model to generalize better than traditional GAMs. Additionally, we have compared our model against other state-of-the-art classification techniques and demonstrated that we can achieve similar accuracy despite limitations on the network's predictive power. Finally, we investigate some of the advantages

and disadvantages of this this technique and discuss how it fits with other machine learning paradigms.

Chapter 2

Background

2.1 Overview

Machine learning is a sub-field of data analytics that uses semi-automated techniques to transform data into insights. Machine learning plays a major role in the modern world and forms the backbone for everything from image recognition (Davis 2018) to weather forecasting (McGovern et al. 2017). Today, there are thousands of different machine learning techniques that fall into a number of major categories. In this work, we are primarily interested in paradigm of supervised learning.

In its most simplistic form, *supervised learning* is the art of fitting functions to data. Given some known input vector x and an unknown response vector y, we would like to make a well-founded approximation of what that output vector is likely to be, $\hat{y}$. Ideally, this output $\hat{y}$ should be guided by how the inputs and the outputs have interacted in the past. While there are many different techniques for generating this prediction, most methods have three features in common: a function approximator, a loss function, and an optimizer.

Supervised models rely on the assumption that the outputs can be described as some function of the inputs $f(x)$ plus some unpredictable element σ, known as *noise* (equation 2.1).

$$y = f(x) + \sigma \tag{2.1}$$

A *function approximator* does its best to mimic this underlying function without getting thrown off by the noise. Different techniques implement this approximator in different ways, and each technique has its advantages and drawbacks.

Just as important as the function approximator is the *optimizer*, which determines how the approximator will learn from the data and how it will handle noise. A key element of an optimizer is the *loss function*, which measures the difference between a model's output and the actual output, a value known as the *residual*. The optimizer is the most important feature during model training, and more powerful approximators typically have more complex optimizers.

Two important features to consider when designing a model are its power and interpretability. *Power* describes how well the approximator can describe arbitrarily complex behavior, while *interpretability* describes how easy it is for a human to understand how the model came to its conclusion. More powerful models are more capable of approximating the underlying function, but are generally less interpretable (Du et al. 2018).

2.2 Linear Regression and Elastic Nets

Linear regression is the oldest and possibly simplest form of machine learning. The function approximator is a simple weighted sum of the input variables plus a bias (equation 2.2).

$$\hat{y} = w \cdot x + w_0 = \sum_{i=1}^{n} w_i x_i + w_0 \tag{2.2}$$

The most common loss function minimizes the sum of the squares of the residuals. This squaring has the effect of amplifying large residuals, forcing the model to account more heavily for outliers. In its most basic form, linear regression with this type of loss function is known as ordinary least squares regression (equation 2.3).

$$loss = (\hat{y} - y)^2 \tag{2.3}$$

This loss function has the added benefit in that it is analytically tractable, a unique feature among machine learning methods making ordinary least squares a highly efficient machine learning method. The weights that minimize this loss function can be found using the following equation (equation 2.4).

$$\hat{w} = (x^\top x)^{-1}(x^\top y) \tag{2.4}$$

Computationally cheap and remarkably effective, linear regression has been a benchmark of statistics since the late 19th century (Galton 1886). While useful in a number of applications, simple linear regression does have drawbacks that limit its usefulness in many scenarios.

The first issue with simple linear regression is that it tends to overfit when presented with a large number of features relative to the number of samples. This issue can be mitigated by adding a *regularizer* that penalizes large weights. The two most common regularization techniques are to punish on the absolute value of the weight, or L_1 norm (Santosa and Symes 1986), and to punish on the squared value of the weight, or the L_2 norm (Tikhonov and Arsenin

1977). Linear regression that employs just the L_1 punishment is called *Lasso Regression* (Tibshirani 1994), just the L_2 punishment is called *Ridge Regression*, and the technique that employs both punishments is called an *Elastic Net* (Zou and Hastie 2005). Each regularizer is weighted with a hyper-parameter that must be tuned during validation. Because elastic nets use two regularizers, the technique requires tuning of two hyper-parameters.

While regularizers can help solve the over-fitting issue, linear models still have limitations on the type of functions they can model. Most importantly, linear regression assumes that the underlying output variable can be represented as a simple linear combination of the inputs, something that's just not true for most real-world problems.

2.3 Tree-Based Methods

In order to model non-linear functions, we need to move away from simple linear regression, towards more powerful techniques. Decision trees are one of the oldest non-linear machine learning methods (Breiman et al. 1984), and variations on the technique are still common to this day (Natekin and Knoll 2013).

Similar to a flowchart, decision trees work by iteratively asking a sequence of yes/no questions that move the model towards a prediction (see Figure 2.1). This process segments the input space, effectively grouping elements that are perceived to be similar or share important characteristics. The model then takes an average of the output of these grouped elements. While initially fit by hand (Chisholm et al. 1968), decision trees became a staple of machine learning with

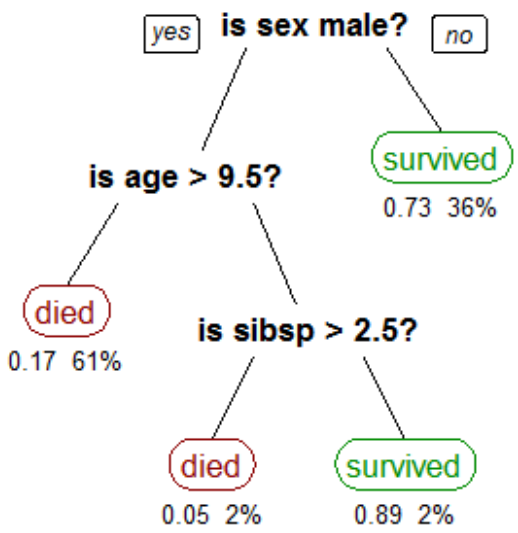


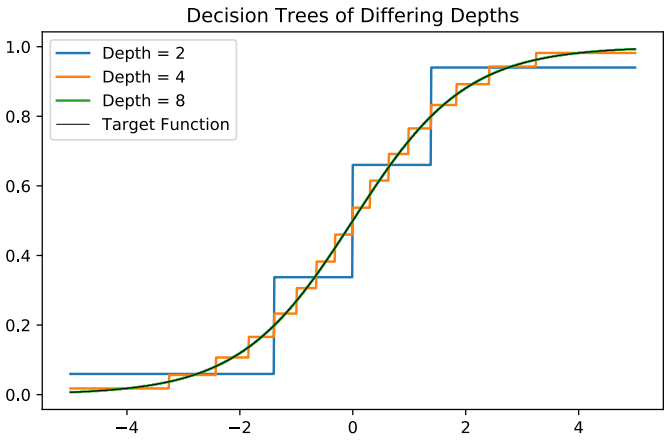
Figure 2.1: Example of a decision tree trained on the Titanic dataset (see below). Trunk nodes represent decisions to be made and leaf nodes represent the final prediction.

the development of an optimizer that could algorithmically determine variable splits using training data (Breiman et al. 1984).

This optimizer works in a greedy, top-down fashion, iteratively splitting the input space and calculating an impurity metric. While many such metrics exist, such as Gini impurity (Breiman et al. 1984), information gain (Kullback and Leibler 1951), and variance reduction (Breiman et al. 1984), they all effectively work to minimize the dissimilarity of the response variable in each group. When the branch with the smallest impurity is found, the tree splits on that variable and the process is repeated recursively for the two new groups.

The adoption of the decision tree was a significant step forward for the field of machine learning. Unlike linear regression, which is restricted to the realm of linear functions, decision trees can approximate any rational function. However, this assertion comes with an important caveat. Because decision trees are

Figure 2.2: Approximating a continuous sigmoid function with decision trees of varying depths. Greater depths provide better precision but also require more data to train.

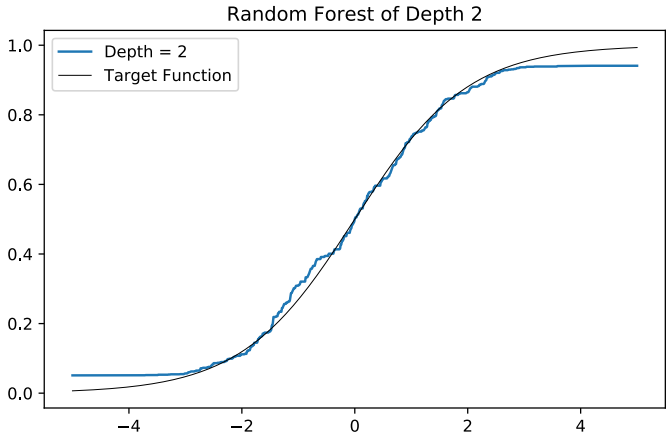


discrete entities, they require a large number of splits to accurately approximate continuous functions.

Every split approximately halves the number of data points in each bin, quickly burning through all available information and causing the method to overfit. As a result, the depth of decision trees must be kept low, making them more "brittle" and causing them to give poor estimates near the decision boundaries.

One solution to this issue is to ensemble many trees together, with each tree fit to a slightly different version of the data. This method is known as random forest regression (Ho 1995). Random forests employ two techniques to augment the data before training a tree: bootstrapping (Ho 1995) and bagging (Ho 2002). Bootstrapping randomly samples n points from the data with replacement while bagging takes a random sub-sample of the features. These modifications weight the data just enough to break symmetry and develop a large number of independent, predictive trees. By averaging the output of these trees, random forests create a smoother function without needing to add additional depth.

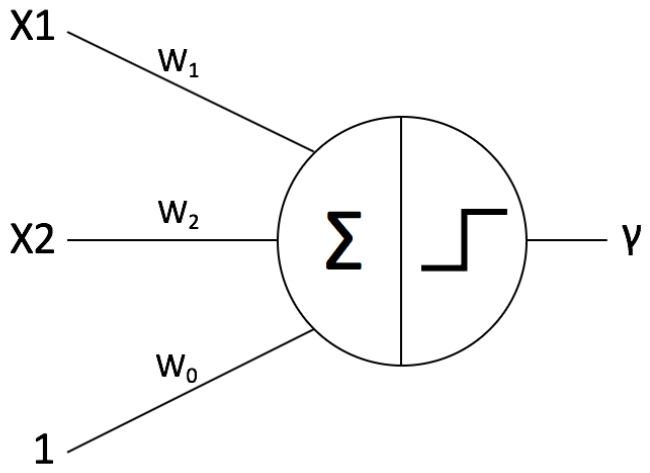
Figure 2.3: Approximating a continuous sigmoid function with a random forest of depth 2. Compared to decision trees of similar depth, the random forest provides much smoother, and therefore more accurate predictions of the underlying function without requiring additional training data.



One further improvement to random forests is the Gradient Boosted Machine (GBM) (Friedman 2001). A sort of addition to random forests, GBMs add weight to misclassified samples during training, causing future trees to focus more heavily on difficult regions in the solution space. This modification can often increase performance, by refocusing the model's power on difficult-to-model regions, even if there are relatively few samples in that area. The downside though, is that the trees must be trained sequentially rather than in parallel, slowing the time it takes to fit a model.

Note that each improvement on the standard decision tree decreases the model's overall interpretability. Where decision trees can be read like a flowchart, random forests are an average of lots and lots of trees, making it more difficult to parse out every combination of outcomes. On the other hand, random forests have a very clear-cut notion of variable importance, with more common splits being more important to the outcome. This analysis is somewhat more difficult

Figure 2.4: An example of a perceptron with inputs x_1 and x_2 and a step function activation. The bias is represented by w_0 and the output is represented by γ.



in GBMs due to the non-trivial nature of how the samples are weighted before each tree is trained.

2.4 Artificial Neural Networks

Another powerful machine learning technique is the artificial neural network. Neural networks are a modular approach to function approximation. The base unit is a weighted sum fed in to a non-linear activation function called a *perceptron*.

To get the output of a perceptron, the inputs plus a bias are weighted according to vector w. These weighted values are then summed together and fed as the input to a non-linear activation function f. The purpose of this activation is to allow the perceptron to expand beyond linear functions. Many different activations exist, and the optimal choice is still a point of open research (Duch and Jankowski 2000) (Xu et al. 2015). Historically, the most common activation

function has been the sigmoid (Minsky and Papert 1969) but there is increasing interest in other activations, including the rectified linear unit (Glorot et al. 2011), the leaky-rectified linear unit (Maas et al. 2013), and the exponential linear unit (Clevert et al. 2015).

While potentially more powerful than linear regression, the perceptron is still highly limited in the types of functions it can approximate. For instance, the output of a single perceptron with sigmoid activation will always be sigmoidal. We can get around this limitation by stacking many perceptrons together in parallel. In fact, it has been proven (Cybenko 1989) that a weighted sum of a large number of perceptrons can approximate any rational function. This mesh of perceptrons is known as a *neural network* and has three layers: one input layer containing the features, one output layer containing the predictions, and at least one hidden layer containing the perceptrons.

However, while possible to approximate any rational function with this architecture, it is not always easy to do so. Complex functions often require a large number of hidden nodes, which in turn require large amounts of data and computational resources. One remedy for this issue to stack multiple perceptron layers one after the other (Rumelhart et al. 1985), allowing each layer of the network to learn simpler functions which generalize more effectively, before combining them together into the full result (see Figure 2.5). This effect can speed network convergence, reduce training time, and can radically increase accuracy without requiring additional data.

In high-dimensional input spaces where input proximity is meaningful (images or speech for instance) further improvements can be achieved through the use of *convolutions* (Fukushima 1980). A convolutional network runs a sequence

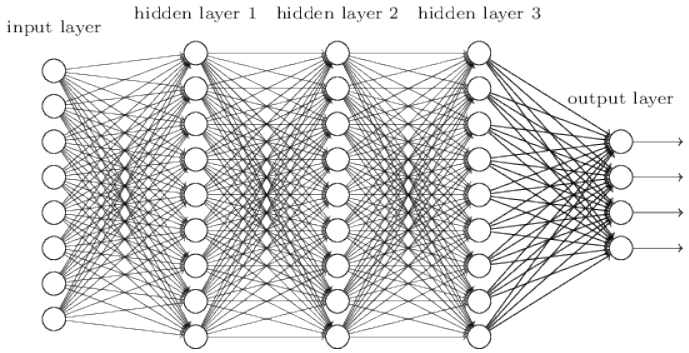


Figure 2.5: Example of a muli-layer neural network. Each node in the graph represents a summation and an activation, which each edge represents a weight. Image credit Nielsen (2015).

of filters over an image to create the inputs for the next layer. While less powerful than a traditional dense layer, this technique significantly reduces the total number of connections by ensuring that only spatially-related information is included in the transformation. This reduction allows convolutional networks to effectively train more complex structures on less input data.

Networks with a large number of convolutional or dense layers are known as *deep networks*, and have produced state-of-the-art results in long-standing computational challenges, including image recognition (Davis 2018), machine translation (Bahdanau et al. 2014), and weather forecasting (McGovern et al. 2017).

Unlike logistic regression, which is analytically tractable, and decision trees, which have a linear-time algorithm for determining splits, artificial neural networks are trained using the *gradient descent algorithm*. Mathematically, the objective of gradient descent is to minimize a loss function, which for simplicity, we will consider to be the sum of the squared residuals as in linear regression (equation 2.5).

$$loss = \sum_{i=0}^{n} (y_i - \hat{y}_i)^2 \tag{2.5}$$

Because there does not exist an analytical solution to this optimization problem, we are forced to use numerical methods to approximate a solution. By relating the weights of the network to the loss function and computing a derivative, it is possible to determine which way to adjust the weights to lower the overall error of the network (equations 2.6 and 2.7).

$$loss_i = (y_i - f(w^\top x))^2 \tag{2.6}$$

$$\frac{\partial_{loss_i}}{\partial_{w_j}} = -2(y_i - f(w^\top x))f'(w^\top x)x_j \tag{2.7}$$

A similar formula can be derived for deeper layers by invoking additional iterations of the chain rule. By iteratively pushing the weights a small amount, the network reduces in error until it reaches an equilibrium state, in which it is considered *fit* to the data.

With the success of the deep learning paradigm, neural networks have become ubiquitous in the field of machine learning. Powerful libraries including Keras (Chollet 2019), TensorFlow (Martin et al. 2015), and PyTorch (Paszke et al. 2017), make it easy to train and integrate neural networks on a range of systems and hardware. TensorFlowJS (TensorFlow 2019) allows neural networks to be embedded in web frameworks while other extensions provide support for statistical software like R (Allaire 2019). Years of research in the field have developed highly efficient methods and optimizers to train networks faster on larger amounts of data (Kingma and Ba 2014). These successes and widespread

support have made neural networks one of the most common methods for modeling in industry, spanning fields from weather forecasting (Lagerquist et al. 2017) to ecology (Chilson et al. 2019).

Like random forests and GBMs, neural networks suffer from a lack of interpretability. While architecture-specific improvements have been suggested to simplify (Olden and Jackson 2002) or visualize (Özesmi and Özesmi 1999) the transformations inside a network, none of these solutions fully resolve the interpretability issue. As a result, researchers have started looking to other tools to help explain a model's output.

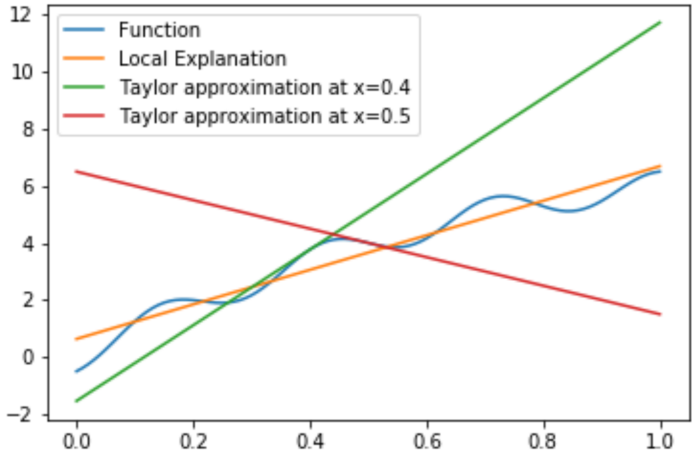
2.5 Local Approximation

One important interpretability technique is to approximate a model over a small domain rather than approximating the entire function. In other words, our objective is to fit a linear model such that it is a faithful representation of the larger model on some domain. This approach works well, as most models behave linearly on a small enough scale, allowing for relatively simple interpretations.

The most obvious local linear approximator is a first-order Taylor polynomial centered at a desired point. This approach simply fits a line to the slope of the model at a given point. While this technique works, it can be extremely misleading. Taylor polynomials are only guaranteed to be accurate when very close to initial point, which may not be a good indicator of what the model is doing on a more reasonable interval.

A better approach is to fit a linear model over a neighborhood that can be defined by the investigator. This linear model will be more stable than the Taylor approximation, and will provide better approximations, so long as the

Figure 2.6: Example of Taylor Polynomials failing to approximate the neighborhood behavior of a non-linear function. (Plumb et al. 2019)



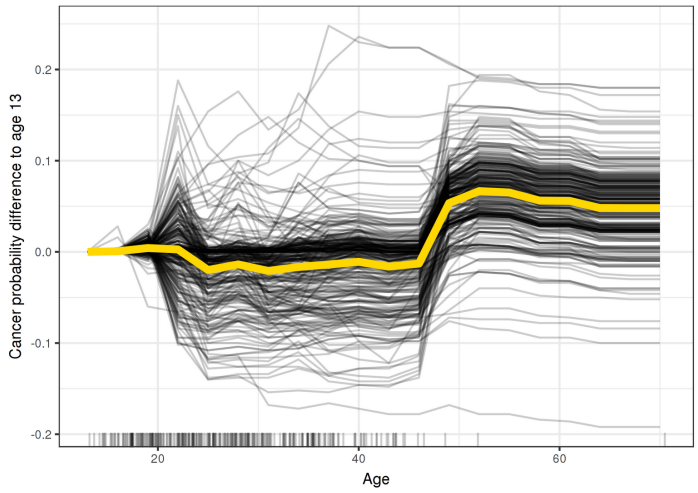
neighborhood selected is guaranteed to be approximately linear. While there are different ways of addressing this problem, one approach is to apply regularizers to the model during training, guaranteeing the model to be approximately linear in some neighborhood (Plumb et al. 2019).

Local approximation techniques are still an open area research (Ribeiro et al. 2016; Plumb et al. 2018) and initial results have been promising. However, these approaches only cover local approximations. In large input spaces, it is infeasible to test every potential point for a local approximation, so we instead look to other methods to explain the global behavior of an arbitrary model.

2.6 Global Approximation Methods

Partial dependence plots (PDP) and Individual Conditional Expectation (ICE) are two of the most popular model-agnostic methods for interpreting the global behaviors of machine learning methods. Both work by varying one input x across its full range of possible values while holding all other inputs constant

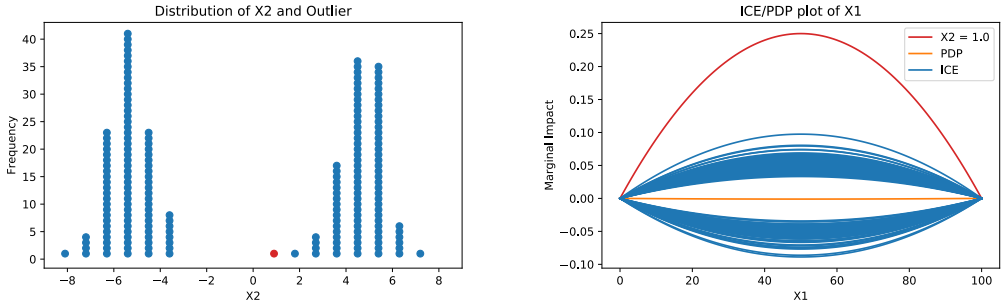
Figure 2.7: Example of a PDP plot layered on top of an ICE plot. Black lines represent ICE and the thick yellow line represents the PDP plot. (Molnar 2019)



and plotting the response. In ICE this process is repeated for each data point and all plots are shown. In PDP, the mean response is plotted for each value of x. While these techniques do work, they are far from perfect.

ICE plots track every observation across the range of all possible values x. While fine for a small sample, as the number of observations grow, ICE plots become more complex and difficult to read. While sub-sampling can help with this complexity, it runs the risk of leaving out important behaviors. Furthermore, in ICE plots its often unclear which line correlates with observation, somewhat dampening the overall interpretability.

PDP plots suffer from the opposite issue. While simple and easy to read, they rely on several strong assumptions which can lead to poor approximators for the underlying model. The most important assumption for PDP plots is the that of input independence, as correlations in the input space will result in unrealistic values being included in the average. For instance, a PDP plot investigating the impact of input x_1 on output $E(y|x_1, x_2)$ will provide biased results if x_1 and x_2 are highly correlated, as some values of x_2 will be significantly less likely given a



(a) Dot plot of variable x_2. (b) Joint ICE and PDP plot of variable x_1.

Figure 2.8: Example of how ICE/PDP plots can lead to deceptive results in under-defined input regions. In this instance, x_2 is distributed bimodally, with peaks at positive and negative five. These peaks correspond to two sets of parabolic curves in the ICE plot and a trivial PDP curve. Aside from the PDP plot being unhelpful, the dot and ICE plots strongly suggest that the behavior of x_1 is stable on the interval [-8,8]. However, an unexpected inverse causes outlier point one (red) to exhibit aberrant behavior. Actual function: $y = \frac{-x_1^2+0.25}{x_2}$

known value of x_1, but will still be included in the PDP average. Furthermore, because they take an average of a set of observations, PDP plots do not address the impact of joint terms in the model, and therefore miss a significant portion of model behavior.

Finally, neither method describes how the model will extrapolate to under-defined regions in the input space. For instance, if our underlying function is $y = x_1 x_2^2$ and we're looking at an ICE plot of x_2, where we have only seen large positive and negative values of x_1, the figure will contain two parabolic structures, one for positive values of x_1 and one negative values. While useful for known values, the plot provides no information on what the model would do if x_1 were equal to zero.

This blind spot is a major issue in real-world applications, as under-defined regions are the most likely to break a well-fit model in a live environment.

The fundamental issue with each of these methods is that there is no feasible two-dimensional representation sufficient to explain the complexity of an arbitrary machine learning model. While possible to generalize the behavior to some degree, there is always a risk of encountering some portion of the input space that was not covered by the interpretation scheme.

One solution is to simplify the model itself, such that the internal representations can be expressed exactly as a PDP plot, or a simple sum of arbitrary one-dimensional functions. While uncommon in machine learning, this approach is well-known in the field of statistics as a *Generalized Additive Model* (GAM).

2.7 Logistic Regression, GLMs, and GAMs

Before jumping in to GAMs, we first need to explore an important early development in the linear paradigm: logistic regression (Nelder and Wedderburn 1972). Linear regression has a number of drawbacks, but one of the most troublesome was the lack of support for classification problems. This issue with this class of problem is that the response always takes one of two values: zero or one. This feature violates the constant variance assumption of linear regression and leads to impossible behaviors, such as predictions with over one-hundred or below zero percent confidence.

Logistic regression solves this issue by piping the output from linear regression into a sigmoid (equation 2.8, bounding the predictions between zero and one. This simple step was a major breakthrough, as it proved that linear models could be expanded beyond linearly structured problems.

$$\frac{\partial_{loss_i}}{\partial_{w_j}} = -2(y_i - f(w^\top x))f'(w^\top x)x_j \tag{2.8}$$

Similar approaches were applied using other distributions, allowing the distribution of the response to take the form of a Poisson or Bernoulli, for instance. Methods of this type are known as *Generalized Linear Models* (Nelder and Wedderburn 1972) and use a *link* function to circumvent the constant variance assumption of linear regression.

Further improvements can be attained using Generalized Additive Models (GAMs) (Hastie and Tibshirani 2007) which take the additional step of delinearizing each of the inputs as well as outputs. This change significantly improves the predictive power of the model, allowing it to handle complex non-linear functions nearly as well as other machine learning techniques like random forests or neural networks.

GAMs work by passing each of the inputs through carefully selected *spline* functions, which delinearize the inputs before passing them into the model. By combining multiple non-linear functions together the model can effectively approximate arbitrary combinations of non-linear behavior as illustrated in Figure 2.9. This technique is effectively a form of feature engineering which significantly increases our input space (Molnar 2019). GAMs therefore employ a regularizer similar to elastic nets which help dampen the weights to prevent over-fitting.

The simplest and perhaps most common method of choosing splines is to manually apply non-linear functions to an input using available domain knowledge. A simple example of this approach is the introductory physics exercise of applying a quadratic spline to the displacement of a falling object to determine its acceleration from gravity. While effective, this approach is heavily labor intensive, requires significant domain knowledge, and is not guaranteed to be optimal unless there is some strong theoretical foundation for the spline choice.

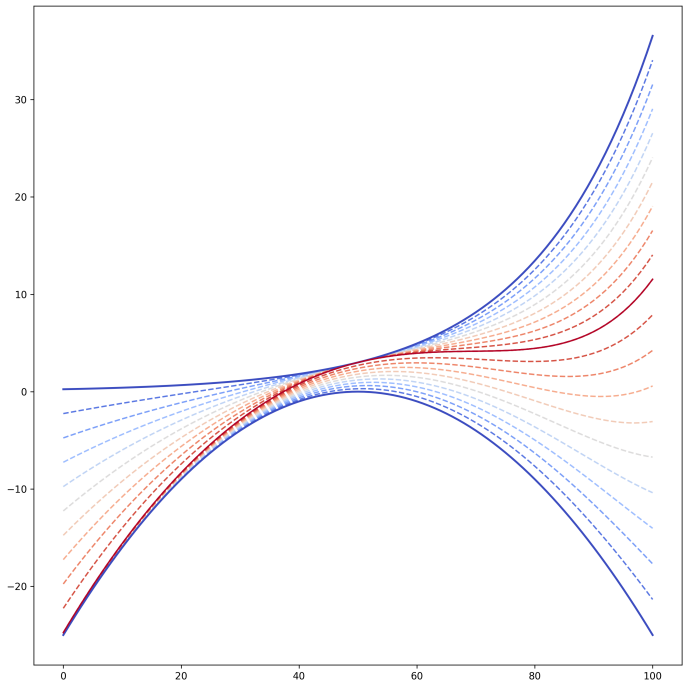


Figure 2.9: Example of how two non-linear splines can be added together to create new functions. The thick blue lines are an exponential spline and a polynomial spline. The thick red line is the new function generated by equal weighting of the two splines. The dashed lines represent intermediate weighting values.

A better option is to use machine learning to fit the spline using a special type of function known as a *b-spline* (de Boor 2006). B-splines, short for *open uniform quadratic basis splines*, are a cheap and effective method of fitting somewhat arbitrary one-dimensional functions. An extension of bézier curves, b-splines use polynomials to smoothly interpolate the values between a set of control points known as *knots*. Each section of the b-spline employs a set of overlapping quadratic bézier curves defined over a small subset of the domain. All knots are equidistant from one another (uniform) and additional knots are added on the ends to force the function to go all the way to the edge of the data (open). By weighting each of these piece-wise curves, we can fit a smooth non-linear approximation of the underlying data.

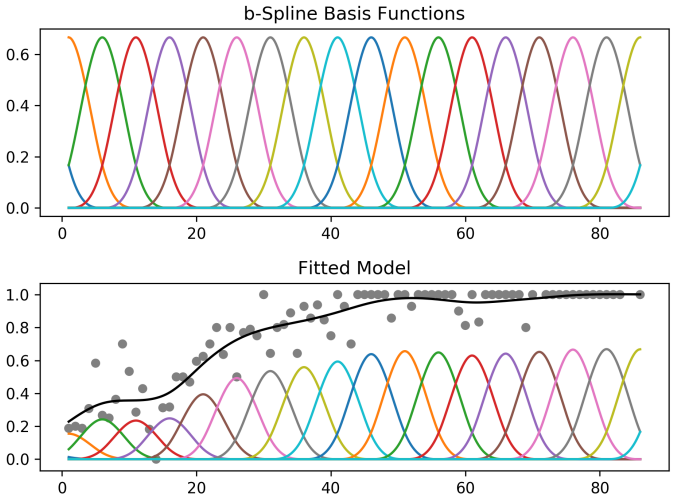


Figure 2.10: Example of how b-splines can be used to smoothly approximate a non-linear univariate function. (Servén and Brummitt 2018)

GAMs have a number of advantages when compared to other machine learning methods. Unlike simple linear regression and elastic nets, GAMs have non-linear predictive power, allowing them to be used on a wider range of problems. However, unlike random forests and neural networks, GAMs are highly interpretable. By computing the splines, weighting them, and summing them together, it becomes possible to visualize how each input affects the output. See for instance, how a combination of b-splines combines to form an interpretable non-linear function in Figure 2.10.

The output of a GAM can be expressed as a weighted sum of one-dimensional non-linearities (equation 2.9). Because there are no joint terms, these plots describe the complete behavior of the model. This is a useful feature where model interpretability is paramount, as it can provide information about how the model works and prevent errors in unforeseen edge cases or changing input spaces.

$$g(y) = f_1(w_1x_1) + f_2(w_2x_2) + ... + f(w_nx_n) \tag{2.9}$$

While extremely useful in high-interpretability situations, GAMs have several drawbacks that have slowed widespread adoption in the machine learning community. First, they are not a universal function approximator, as they cannot handle arbitrary joint functions of the inputs. Second, the splines are often difficult to tune, requiring specific domain knowledge to correctly fit in certain problems.

Despite heavy support in the statistical language, R, there is little support for GAMs in other languages. Python's leading GAM package, pyGAM (Servén and Brummitt 2018), is less supported and less frequently used when compared to other important machine learning packages like TensorFlow (Allaire 2019) and SciKit-Learn (Pedregosa et al. 2011).

However, what if we were to rebuild the GAM principles in modern machine learning frameworks like TensorFlow? Could neural networks be used to help solve the spline issues, and could we improve support for the method by piggybacking on the development of better supported frameworks?

Chapter 3

Methods

3.1 Network Architecture

Neural networks are often difficult to interpret due to complex interactions between multiple non-linear terms in their hidden layers. While it is easy to visualize curves in one or two dimensions, higher dimensional non-linearities quickly become confusing. By removing these joint interactions from the network, we can improve interpretability while only sacrificing a fraction of the network's predictive power.

Our architecture consists of n parallel sub-networks joined by an additive layer with sigmoid activation. Each sub-network has one input, one output, and one hidden layer containing m sigmoid activated nodes. The output of these sub-networks represent the delinearization of each of the input variables. Each non-linearity is independent and can be plotted in two-dimensions.

Conceptually, this network works in three steps. First, the sub-networks delinearize each of the inputs, mapping the initial values x to the more useful non-linear space x'. Second, the network weights each of the transformed x' values according to vector w and sums them together with a bias to generate score y. Third, y is passed through a sigmoid activation function (logistic function with steepness 1) to realize the bounded classification y'.

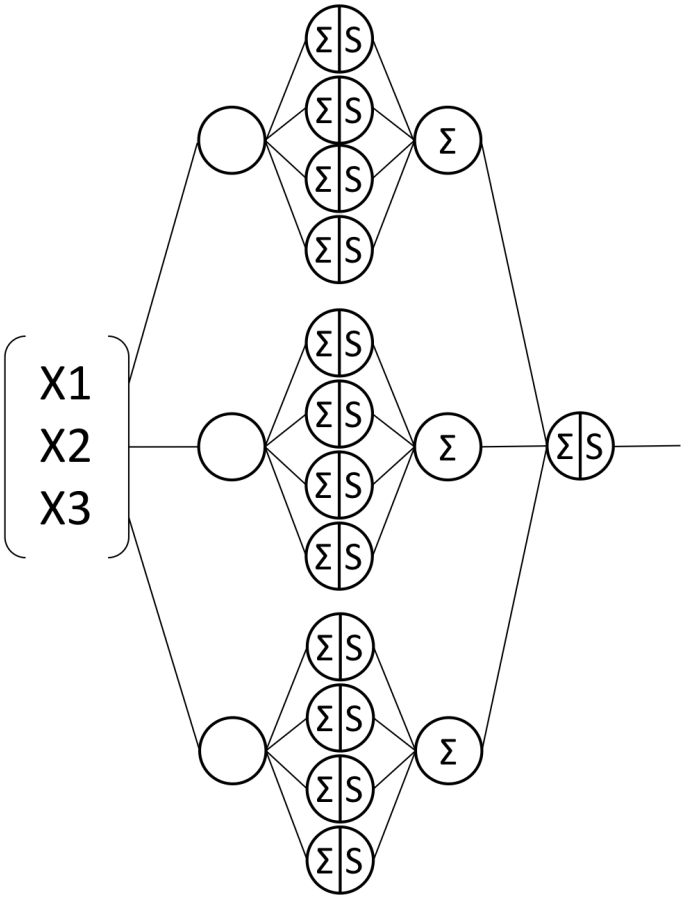


Figure 3.1: Example of our network architecture with n = 3 and m = 4. S represents a sigmoid activation.

$$y'_{net} = g(f_1(w_1x_1) + f_2(w_2x_2) + ... + f(w_nx_n)) \tag{3.1}$$

In theory, this methodology is identical to a GAM with b-splines and a sigmoid link function. In practice however, the sigmoid activations of the neural network can actually fit a wider range of one-dimensional functions than the b-spline GAM. Whereas the GAM is restricted to the polynomial-like equations defined by its splines, the neural network can allocate its hidden units to different regions of the input space. This feature gives the network more flexibility in its approximations, which can result in improved performance (see experiment 2).

We implemented our network in Keras (Chollet 2019) using an Adam optimizer (Kingma and Ba 2014) with a learning rate of 0.1. The number of marginal networks is always equivalent to the number of input variables. Each marginal network has one input, one output, and contains one hidden layer with eight sigmoid-activated hidden units. The outputs of the marginal networks are combined in a final dense layer with sigmoid activation.

3.2 Hypotheses

H1. Our network architecture is interpretable and will closely approximate GAM behavior.

H2. Our network architecture will outperform b-spline GAMs in real-world datasets where the optimal non-linearity is unimodal with high variance over a small domain.

H3. Our network architecture will fail on classification problems where joint terms are required, such as the XOR problem.

H4. Our network architecture will outperform logistic regression in all tested real-world classification datasets.

H5. Our network architecture will approach, but not surpass, the performance of GBMs on any tested real-world dataset.

H6. Our network architecture can be combined with deep networks to form more interpretable models in deep learning tasks.

3.3 Datasets

3.3.1 Real World

To demonstrate the effectiveness of our architecture in real-world classification problems, we compare our method to standard techniques on a number of common benchmark datasets. All datasets in this section have been well-studied in machine learning literature and were selected to represent a wide range of sample and feature space sizes. While our network architecture can be used in multi-class classification tasks, to standardize our comparison, we convert all non-binary classification datasets to binary classification problems by selecting the most difficult class to separate.

In all experiments, the data is randomly shuffled before being fed in to the model. We use 50% of the available data for training and 20% of the available

data for validation. A simple grid search is used to determine the best hyper-parameters for each model. The model is then refit on the combined training and validation data and test results are reported over the remaining 30%.

Wisconsin Breast-Cancer Dataset: (Street et al. 2005) The Wisconsin breast cancer dataset contains 569 observations of 31 features that are used to predict if a given tumor is malignant or benign. There are 212 malignant and 357 benign examples in the dataset. Feature examples include the tumor's mean texture, radius, and perimeter. The response is not linearly separable given the input features.

Sloan Sky Survey: (Blanton et al. 2017) The Sloan Sky Survey dataset contains 10,000 examples of 17 features used to predict if a given celestial object is a star, galaxy, or quasar. The dataset contains 4998 examples of galaxies, 4512 examples of stars, and 850 examples of quasars. Example features include red-shift, ascension angle, and u-band magnitudes. We reduce this dataset to a binary classification task by separating galaxies from other celestial objects.

Iris: (Fisher 1936) Iris is a famous dataset comparing different species of the Iris plant. It contains 150 observations of four input variables: sepal length, sepal width, petal length, and petal width. The objective is to correctly classify 50 examples of Iris setosa, Iris versicolor, and Iris virginica using the four input features. In our experiments, we converted this problem from a multi-class classification problem to a single-class classification problem by trying to separate Iris versicolor from the other two classes.

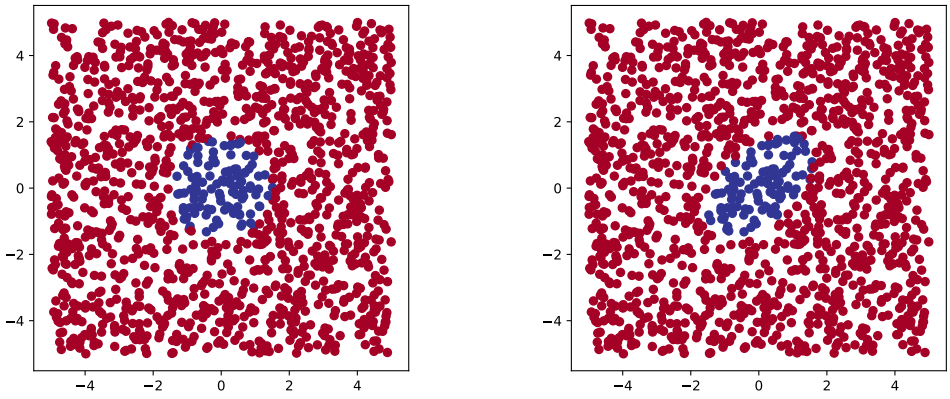
Titanic: (Stanford 2019) The Titanic dataset contains 1309 rows corresponding to the passengers who were on board the Titanic the night it sank. The objective is to use the 13 input features to predict which passengers survived and which ones died during the disaster. Feature examples include sex, age, and ticket fare. 500 passengers survived the wreck while 809 perished.

Wine: (Forina 1991) The wine dataset contains 178 instances of 13 variables corresponding to the attributes of different Italian wine samples. The original dataset contains 59, 71, and 48 examples of each of the output classes. We simplify this dataset to a binary classification task by taking the class with the most intermediate values in the first principle component.

3.3.2 Synthetic

In experiment 3, we employ several synthetic datasets to demonstrate the theoretical limitations of the GAM approach. These datasets are as follows.

Gaussian Islands: The Gaussian Islands are two synthetic datasets, *circle* and *ellipse*, containing 2000 observations of two features bounded between positive and negative five. A two-dimensional Gaussian distribution with mean of zero and variance of one is placed in the center of each grid. In the circle dataset, the covariance between x_1 and x_2 is zero while in the ellipse dataset, the covariance is 0.5. Each observation is randomly placed on the grid. If the value of the Gaussian distribution at that point is greater than one, the point is considered a true positive, if the value is less than one, the point is a true negative.



(a) Gaussian island with covariance of 0.0.(b) Gaussian island with covariance of 0.5.

Figure 3.2: Training data for the simulated Gaussian Islands dataset. Blue points are true positives, red points are true negatives.

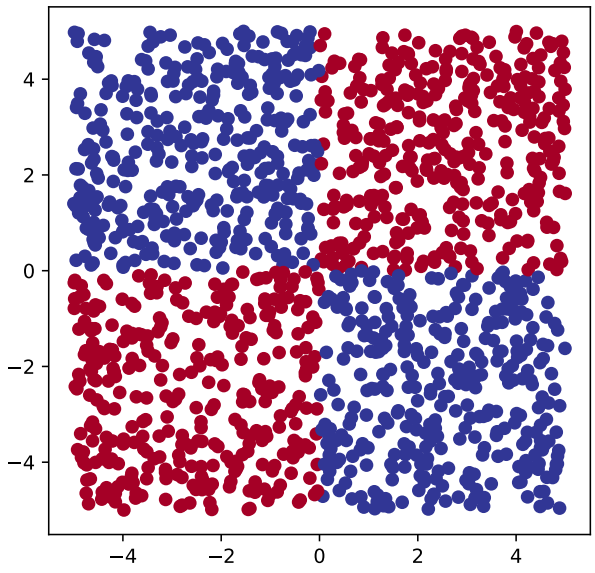
Simulated XOR: (Hara and Hayashi 2016) The simulated XOR dataset contains 2000 randomly selected observations of two features bounded between positive and negative five. If the product of the features is positive, the point is a true positive, if negative, the point is a true negative.

3.3.3 Pandemic

Pandemic is a novel image classification benchmark involving a fictional disease modeled on Yersinia pestis: the bacteria responsible for the black plague (Perry and Fetherston 1997). It is designed as a simple synthetic benchmark for our architecture in image classification tasks.

Most highly contagious diseases tend to follow a common track when they encounter an uninfected individual. First, upon contact with the new host, the contagion rapidly begins to multiply in the favorable environment. The host's immune system identifies the threat and begins to take measures to mitigate

Figure 3.3: Training data for the simulated XOR dataset. Blue points are true positives, red points are true negatives.



the disease. One of these measures is to rapidly bolster t-cell production, a type of white blood cell that will actively fight the contagion.

Depending on the severity of the disease and the frailty of the host, two outcomes are possible: the host's body mounts a rapid response and fights off the disease, or the contagion multiplies out of control, killing the host. These outcomes depend on how quickly the contagion multiplies and how quickly the body can produce an effective immune response. In our dataset, the immune response can be measured by proxy by taking a sample from an infection site and observing the relative frequencies of the bacteria and t-cells.

These samples consist of eight-hundred and seventy 120x120 black and white images containing a number of overlapping t-cells and bacteria. In these images, bacteria are modeled by a plus shape and t-cells are modeled by a square shape,

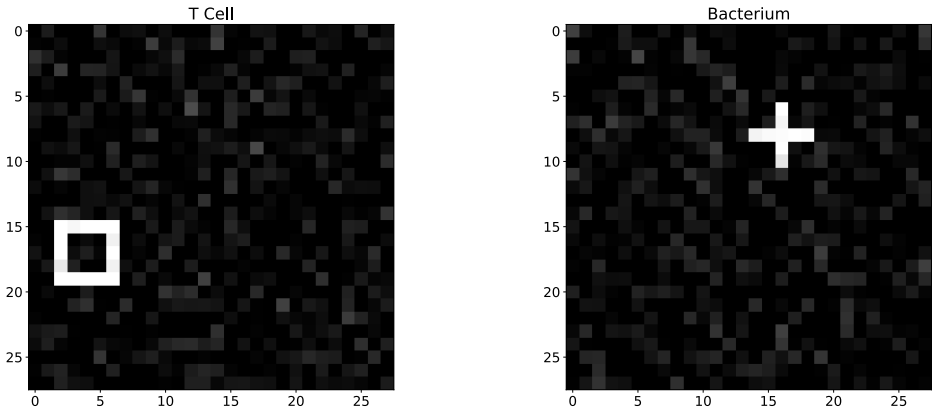


Figure 3.4: Examples of a t-cell (left) and a bacterium (right) in the pandemic dataset.

as illustrated in Figure 3.4. The total number of objects in any sample is bounded between zero and one-hundred.

The distributions of t-cells and bacteria are governed by equations 3.2, 3.3, and 3.4, represented graphically in Figures 3.5 and 3.6.

$$s(t) = \frac{1}{1 + e^{-t}} \tag{3.2}$$

$$i(t) = s(t - c_i) \tag{3.3}$$

$$c(t) = s(vt - c_v)(1 - i(t)) \tag{3.4}$$

Ostensibly, both the t-cell and bacterial distributions are modeled as time-lagged sigmoids, representing an exponential ramp-up in production before tapering as the host reaches capacity. The bacterial distribution is scaled by one minus the t-cell fraction, representing how the immune system's ramp-up up allows the body to fight off the infection.

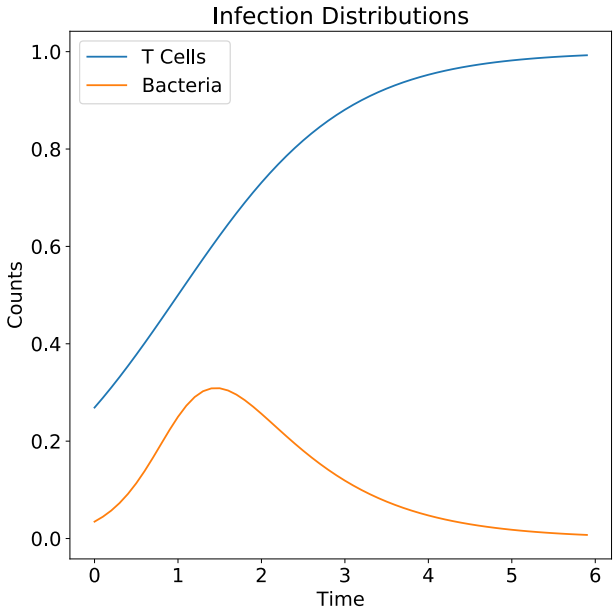


Figure 3.5: Plots of t-cell and bacteria distributions over time with $v = 3$. T-cell fraction increases throughout the simulation while the bacteria fraction peaks and then returns to zero.

Time-lag constants c_i and c_v are used to control the exact shape of the distributions, and for this experiment are set to one and three respectively. The viral rate v is bounded between two and four, with all values above three being considered fatal.

While these classes are easy to separate as-is, the sampling complication described in the following section makes accurate classification much more difficult, if not impossible. Specifically, the sampling approach adds significant noise to the dataset, thereby overlapping the two tails of the distribution and destroying most of the non-linear behavior. Therefore, we introduce two minor changes to the distribution to achieve slightly greater separation between fatal and non-fatal cases. First, we ignore all values v between 2.75 and 3.25 and second, we add lift constant of 0.15 to all fatal cases. Conceptually, this change

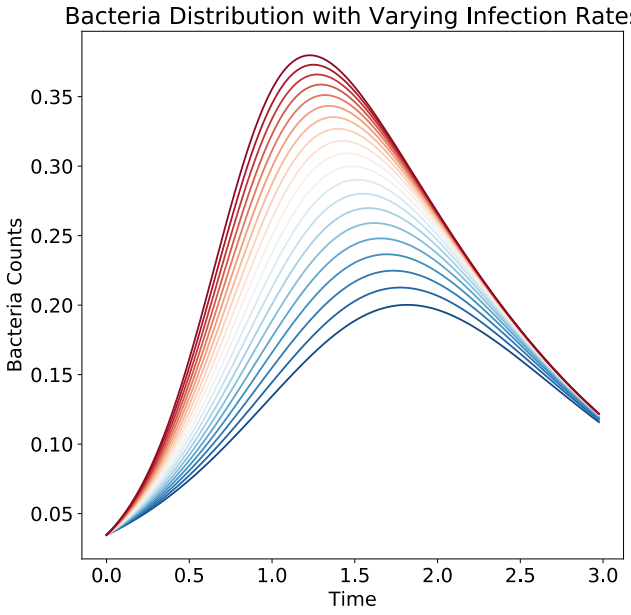


Figure 3.6: Bacteria distributions with v varying between two and four. Redder values correspond to higher values of v and are considered to be more lethal.

removes the most difficult edge cases from the dataset and adds a small amount of bacteria to the fatal cases.

A total of 870 samples are taken in grid fashion from this modified distribution by uniformly varying rate parameter v and time parameter t as shown in Figure 3.7. For testing, a new set of 870 images is generated from the same distribution.

An image is generated for each sample by scaling outputs i and c by fifty times the inverse of the maximum value for each distribution (equations 3.5 and 3.6).

$$i'_t = \frac{50 i_t}{\max(i)} \tag{3.5}$$

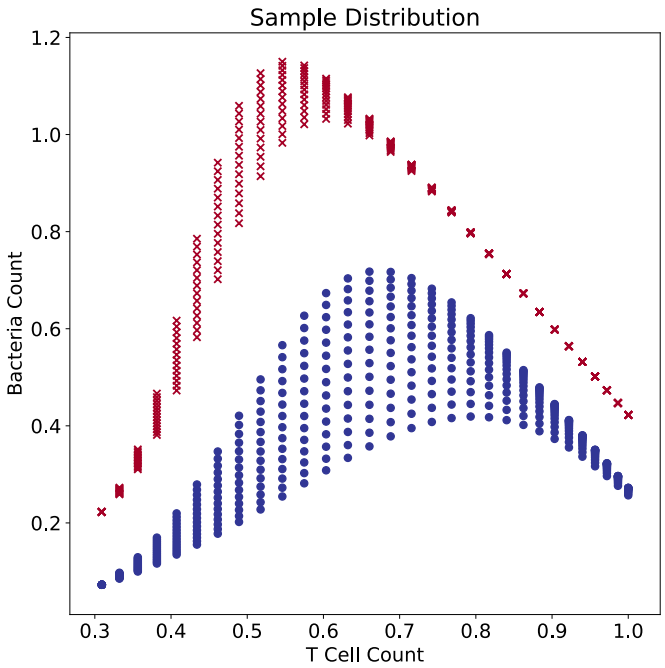


Figure 3.7: Final distribution of bacteria and t-cells from which we pull our image samples. Red x's are lethal while blue o's are non-lethal.

$$c'_t = \frac{50c_t}{\max(c)} \tag{3.6}$$

This step is a linear transformation that forces each of the values between zero and fifty. Objects are then added to the image in random locations with total frequencies corresponding to the transformed values i' and c'. Finally, a layer of uniform noise is applied across the image. An example of this process is shown in Figure 3.8.

Intuitively, this dataset represents a two-part classification task where the model must first identify and count the number of each object in an image and then use those counts to classify it. While each task is relatively simple on its own, the combination of the two presents a more difficult hybrid challenge for most machine learning methods. The counting aspect requires image processing

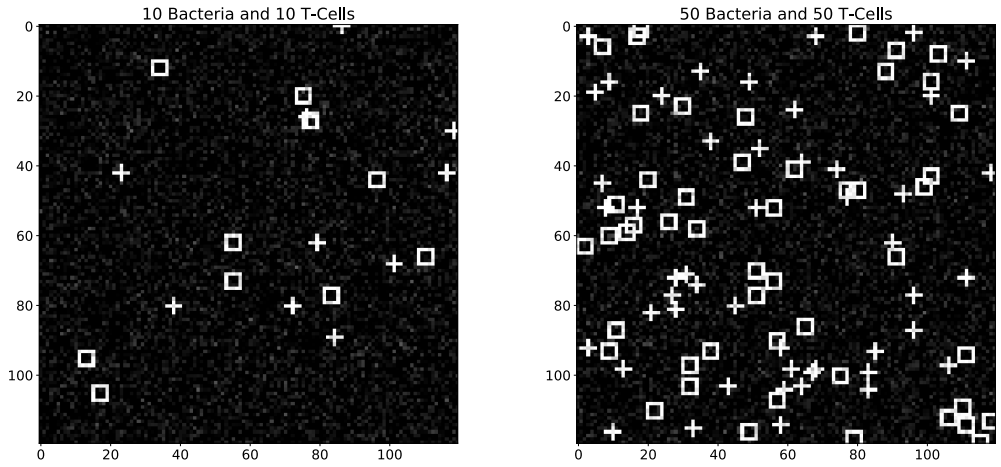


Figure 3.8: Example input images with 20 (left) and 100 (right) objects.

techniques, like convolutional networks, while the classification task requires a method with non-linear predictive power.

In this research, our objective is to correctly classify this dataset using a single model that is partially interpretable. The rationale being that a single model is both easier to deploy in the field as well as trainable in a single step, due to the existence of a gradient over the custom layer.

While other datasets exist that could demonstrate this architecture, Pandemic provides two useful features for our analysis. First, every Pandemic sample has an explicit two-dimensional latent space (bacteria and t-cell counts) that is easy to understand and to visualize. Most image classification tasks require the development of an implicit latent space using a technique like auto-encoders (Baldi 2012), adding unnecessary complexity and obfuscation to our analysis. Second, the Pandemic image classification task is very simple, allowing the experiment to be run on less powerful hardware. While extensions of this methodology are applicable on many datasets, Pandemic is designed to be faster to run and easier understand for the purposes of this demonstration.

3.4 External Libraries

PyGAM: (Servén and Brummitt 2018) PyGAM is the leading implementation of generative additive models in Python. It provides support for standard and logistic GAMs, as well as built-in support for b-splines. In this work, PyGAM is used as a baseline comparison for our neural network technique.

TensorFlow/Keras: (Chollet 2019) (Allaire 2019) TensorFlow is an open source neural network development package developed by Google. It is one of the most powerful and widely used neural networks frameworks available today. Keras is an abstraction layer included in the base TensorFlow package which makes it easier to build and train neural networks. All neural networks presented in this paper were built in Keras with a TensorFlow back-end.

SciKit-Learn: (Pedregosa et al. 2011) SciKit-Learn is the de facto standard framework for machine learning in Python. It provides implementations for logistic regression, random forests, and gradient boosted machines, which are used as benchmarks and points of comparison for our model.

Chapter 4

Experiments

In the following experiments, we test each of our hypotheses and discuss the results in the corresponding section.

Experiment 1: Tests our implementation against PyGAM on the ellipse dataset to examine how well our network approximates a standard GAM.

Experiment 2: Tests an instance in the Sloan Sky Survey dataset where our network outperforms traditional GAMs due to better marginal fitting.

Experiment 3: Tests our network against datasets that GAMs are incapable of fitting.

Experiment 4: Tests our network against a number of real-world datasets to see if the decrease in power is significant in real-world classification tasks.

Experiment 5: Tests the performance of our network as part of a larger convolutional net on the Pandemic dataset.

4.1 GAM Approximation Validation

4.1.1 Setup

Testing hypothesis **H1**. *Our network architecture is interpretable and will closely approximate GAM behavior.*

Experiment one is designed to empirically demonstrate that our neural network architecture will exhibit GAM-like behavior in real-world classification tasks. For this experiment, we compare our architecture against the PyGAM implementation with 10 b-splines on the ellipse dataset. We report the similarity of the predictions using cosine similarity and plots of the marginals are provided for visual analysis. Note, while Euclidean and cosine similarities provide similar results in this instance, we report the cosine similarity due to known issues with Euclidean distance in high-dimensional spaces (Domingos 2012).

4.1.2 Results

Network Accuracy	GAM Accuracy	Cosine Similarity
0.967	0.967	0.997

Table 4.1: Similarity of network and GAM predictions. Both methods arrived at approximately the same result.

4.1.3 Analysis

The results from this experiment support our hypothesis that we can generate GAM-like behavior using our neural network architecture. Both models generate a circular region centered at point (0,0) but fail to account for the correlated regions in quadrants I and III. The accuracy of both models is approximately

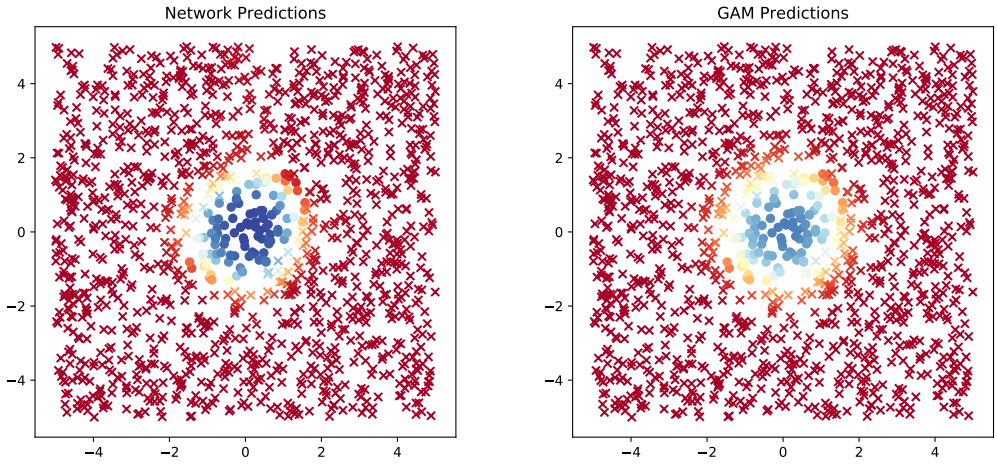


Figure 4.1: Model predictions for both our network and a GAM on the ellipse dataset. Color represents the model prediction, with blue being more positive, red being more negative, and yellow being approximately neutral. X's are true negatives and O's are true positives. Both model facilitate a positive, circular region centered around (0,0) with a ring of uncertainty corresponding to the elliptical section of the input space.

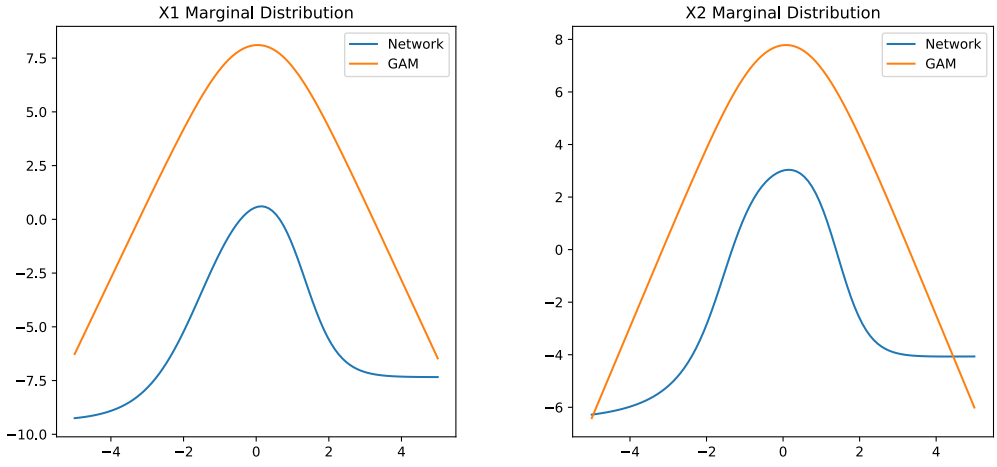


Figure 4.2: Marginal distributions for the x_1 and x_2 variables for both our network and a GAM in the ellipse dataset. Both methods produce symmetric, unimodal distributions centered at zero and negative at the extremes.

96% and the cosine similarity of the predictions is 0.997. These results suggest that our architecture is effectively equivalent to a GAM, and both techniques have roughly the same power.

One important caveat to this conclusion is the stark difference in how each model arrives at its predictions. While the GAM's marginal distributions favor polynomial-like functions, our architecture tended to produce Gaussian-like curves for its predictions. This difference stems from the mechanism by which each model fits its respective marginal functions. In GAMs, the b-spline technique is limited to smooth polynomial-like fits, while our neural network is built upon sigmoid non-linearities, which can produce any arbitrary function, but tend to favor a slope of zero near the end points. While this difference is often negligible, like in this dataset, there are instances where it can impact model accuracy.

4.2 GAMs vs. Networks

4.2.1 Setup

Testing hypothesis **H2**. *Our network architecture will outperform b-spline GAMs in real-world datasets where the optimal non-linearity is uni-modal with high variance over a small domain.*

Experiment two compares our architecture to the pyGAM implementation on the Sloan sky survey dataset. This dataset was selected for the red-shift variable, which is strongly predictive of a celestial object being a galaxy over a very narrow range of intermediate values. We expect that traditional GAMs will struggle to fit this variable effectively, as a large number of b-splines are required to effectively model the feature. Our method will have few issues fitting

this variable, as our neural network has additional flexibility to allocate power in this small domain.

We compare our architecture with eight hidden units against GAMs with an increasing number of splines. We then compare the f1-scores of the two models, as well as a selection of marginal distributions to determine which method provided superior results.

4.2.2 Results

Splines	5	10	15	20	25	30	35	40
F1 Score	0.929	0.939	0.951	0.961	0.969	0.971	0.973	0.975

Table 4.2: F1 scores on the stellar dataset for a GAM with an increasing number of b-splines. F1 scores, precision, recall, and accuracy increase with an increasing number of splines, but the model fails to converge beyond 40.

Hidden Units	5
F1 Score	0.986

Table 4.3: F1 scores on the stellar dataset for our network architecture. The neural network is more accurate than the GAM with only eight hidden units.

4.2.3 Analysis

Results from this experiment support our hypothesis that our architecture is more capable than traditional b-spline GAMs at fitting functions with high output variance over small univariate domains. While increasing the number of splines did help to improve the GAM's performance, it came at the cost of significantly increased complexity. Rather than five b-splines per feature, the best performing GAMs required upwards of forty, which significantly increased

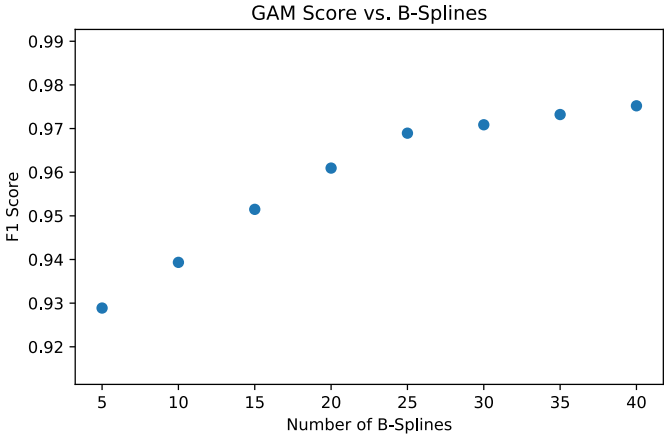


Figure 4.3: F1 scores on the stellar dataset for a GAM with an increasing number of b-splines. Model accuracy increases with an increasing number of splines but fails to converge beyond 40.

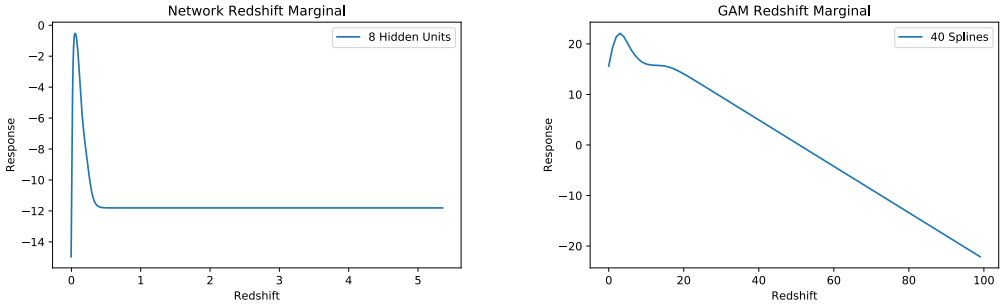


Figure 4.4: Marginal for the redshift variable in the network and GAM implementations. Commonly used as a proxy for distance, the peak in the network marginal corresponds to the high-density region for galaxies in the stellar dataset. The 40-spline GAM was unable to achieve a similar fit, and the slow decay at high redshift values decreases overall model performance.

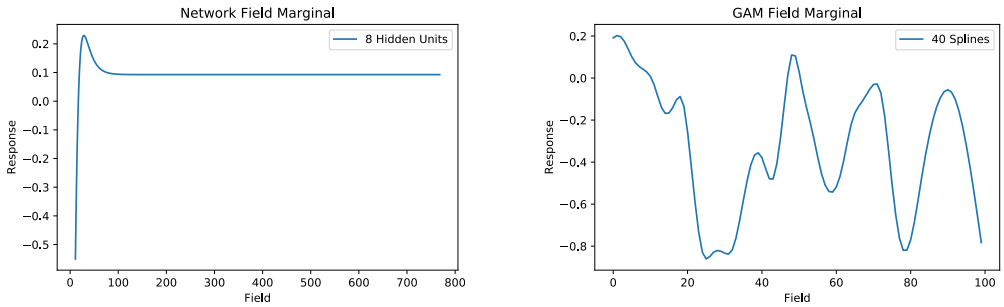


Figure 4.5: Marginals for the field variable in network and GAM implementations. Represents the location in the sky where the telescope is looking. A large number of b-splines has caused the GAM model to overfit the variable when compared to the network architecture.

training time, made the model less likely to converge, and decreased the interpretability of the remaining marginals. Furthermore, the model's accuracy could not be increased further by adding additional splines, at the model failed to converge when pushed beyond forty.

Even the 40-spline GAM model was unable to match the accuracy of our network architecture with just eight hidden units. Unlike GAMs, neural networks can transform the location of their non-linearities by modifying their input weights, allowing the model to better focus its limited power on more important regions. For this dataset, this ability was extremely important, as our method was able to concentrate its effort in the most important section of the input space, providing a smooth, sensible marginal distribution for the red-shift variable.

The use of marginal layers in the input network is a major improvement to univariate function approximation when compared to traditional b-spline GAMs. Our architecture is able to provide a better fit with less power thanks to additional non-linear flexibility. While b-splines are technically able to emulate any univariate function to an arbitrary degree of accuracy, in practice, this

functionality requires careful tuning and lots of guesswork on the part of the modeler. Our network handles this process automatically, making it generally superior to the b-spline approach. Further work with activation functions in this type of marginal architecture is likely to improve these results even further.

4.3 Architecture Limitations

4.3.1 Setup

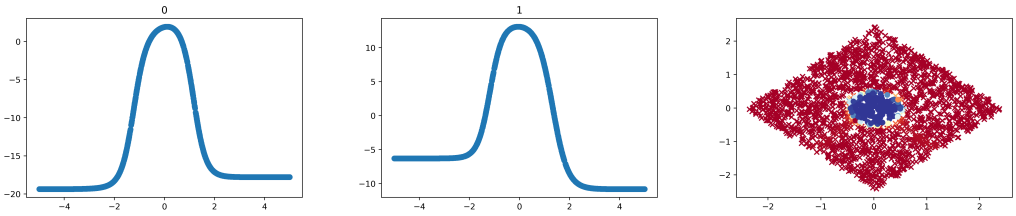
Testing hypothesis **H3**: *Our network architecture will fail on classification problems where joint terms are required, such as the XOR problem.*

Experiment three demonstrates the limitations of our architecture on two synthetic datasets: Gaussian islands and XOR. This test is designed to demonstrate the limitations of GAM-like architectures in instances where the response is a non-linear combination of univariate functions of the inputs.

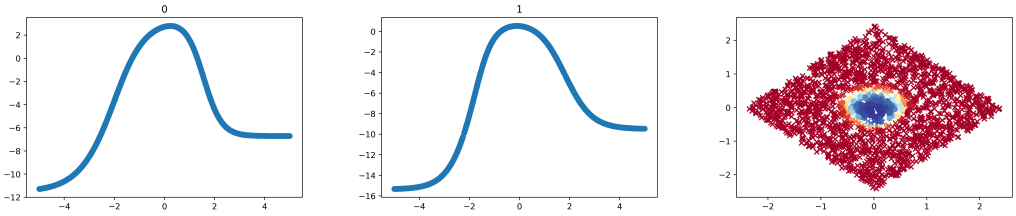
4.3.2 Results

Dataset	Logistic	GBM	Network
Circle	0.0000	0.8824	**0.9903**
Ellipse	0.0000	**0.8654**	0.8113
XOR	0.4357	**0.9983**	0.4618

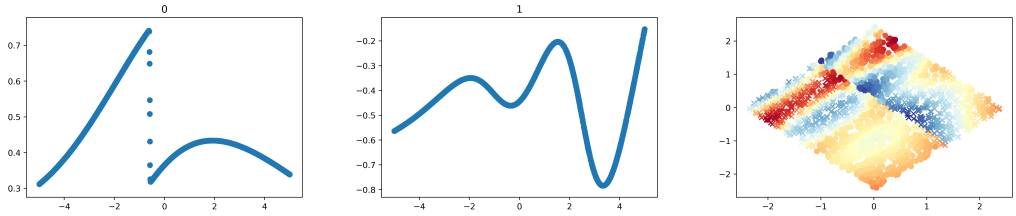
Table 4.4: Test set F1 scores for each of the models on each of the real-world datasets. Underlined values are statistically significant versus the worst-performing model. Bold values are statistically significant versus to both models. Statistical significance is measured at the 0.05 confidence level.



(a) Circle Dataset



(b) Ellipse Dataset



(c) XOR Dataset

Figure 4.6: Network predictions and selection of marginals for each dataset. In the first two columns (marginals), the x-axis represents the domain on the input variable and the y-axis represents the pre-sigmoid contribution to the response. In the third column, network predictions are presented as two-dimensional principle components plots. X's are true negatives and O's are true positives. Colors represent the network prediction, with blue being positive, red being negative, and yellow being neutral.

4.3.3 Analysis

Circle: True to its name, the circle dataset contains a circle of true positives encased in a field of true negatives. Our architecture performs well in this dataset, achieving an f1 score of 0.9903. GBMs perform slightly worse, with an f1 score 0.8824, due to mild over-fitting on the edges of the circle. This dataset

is not linearly separable, so logistic regression performed poorly, with an f1 score of 0.0000.

This dataset demonstrates one of the unexpected advantages of our power-reduced neural network architecture. By removing the joint terms from the network, we've reduced the number of weights our network needs to fit by an order of n when compared to an equivalently sized fully-connected layer. This reduction helps curb over-fitting, allowing the network to perform better without applying additional regularization constraints, such as dropout (Srivastava et al. 2014).

Ellipse: The ellipse dataset is similar to the circle dataset, but with a correlation of 0.5 between the x_1 and x_2 variables. Our network performance falls significantly to 0.8113 as the method is unable to handle correlations between input variables. GBMs perform almost identically with a score of 0.8654, while logistic regression performs poorly with an F1 score of 0.0000.

This dataset illustrates the major downside of our network architecture and other GAM-like approaches. While the network is capable of handling the circle, the addition of correlation term x_1x_2 creates a behavior that cannot be modeled, resulting in a major loss of accuracy. This issue applies to all behavior that cannot be modeled as a linear combination of univariate functions, and can lead to decreased accuracy, as in this example, or a complete failure to fit, as in the XOR dataset.

XOR: The XOR dataset mimics a famous neural networks problem, where the goal is to classify points in quadrants one and three as class one and points in quadrants two and four as class two. Our architecture is completely unsuited for this task, and achieves a similar f1-score to logistic regression. While possible

to model one of the four quadrants, joint terms are required to model both, so more powerful methods like GBMs will perform much better on this type of problem.

These limitations help explain why GAMs and GAM-like architectures have not risen to prominence in most mainstream machine learning applications. When compared to more powerful techniques like fully-connected neural networks or tree-based ensembles, this architecture seems woefully under-powered, failing to achieve valid predictions for even these very simple problems. However, the improved interpretability of this method still warrants attention, and the question remains: how much do joint terms matter in real-world classification tasks?

4.4 Real World Classification Tasks

4.4.1 Setup

Testing hypotheses **H4** and **H5**. *Our network architecture will outperform logistic regression in all tested real-world classification datasets. Our network architecture will approach, but not surpass, the performance of Gradient Boosted Machines on any real-world dataset.*

Experiment four is designed to test our network architecture against other state-of-the-art techniques in real world classification datasets in order to ascertain the importance of joint-terms in this class of problem. Our model is tested against SciKit-Learn logistic and gradient boosting classifiers in binary versions of the Wisconsin Breast Cancer, Sloan Sky Survey, Iris, Titanic, and Wine datasets. In all instances, we compare the f1-scores of all models on all datasets and run statistical significance tests on the results.

4.4.2 Results

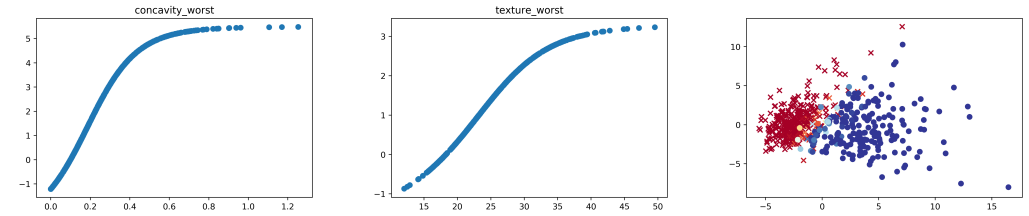
Dataset	Logistic	GBM	Network
Breast Cancer	0.3592	0.9383	0.9512
Sky Survey	0.6586	0.9871	0.9864
Iris	0.3478	**0.9333**	0.8966
Titanic	0.7231	0.7361	0.7200
Wine	0.9231	0.8718	0.9189

Table 4.5: Test set F1 scores for each of the models on each of the real-world datasets. Underlined values are statistically significant versus the worst-performing model. Bold values are statistically significant versus to both models.

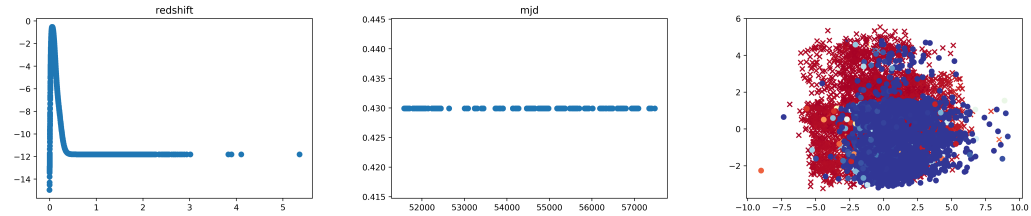
4.4.3 Analysis

Results from this experiment support our hypotheses that our network architecture outperforms logistic regression while falling slightly short of GBMs in most real world datasets. In three out of the five datasets tested, our architecture was statistically significantly better than logistic regression and in the remaining two, the results were not statistically significant either way. GBMs outperformed our architecture in the iris dataset, and there was no statistical difference in the remaining four.

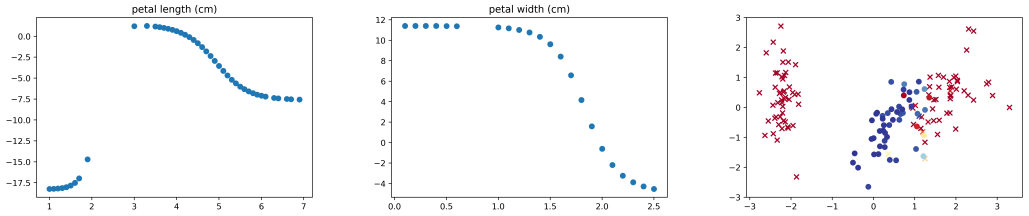
In each of the datasets, our network was able to pull out pertinent non-linear features and incorporate them into the network prediction. The lack of joint terms had surprisingly little impact on the accuracy of the network, as illustrated by the nearly identical performance of our network with the more powerful GBM. This result suggests that, for real-world classification problems, joint terms are often unnecessary, and it is possible to increase interpretability



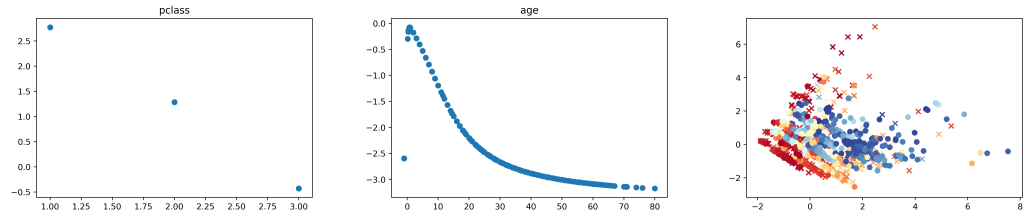
(a) Breast Cancer Dataset



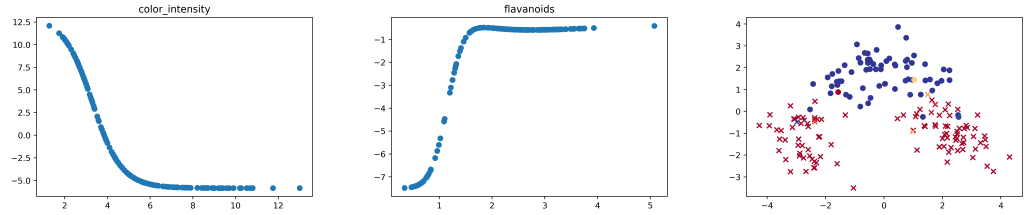
(b) Sloan Sky Survey Dataset



(c) Iris Dataset



(d) Titanic Dataset



(e) Wine Dataset

Figure 4.7: Network predictions and selection of marginals for each dataset. For a full explanation, see figure 4.6.

without sacrificing accuracy by employing a GAM-like delinearization approach in this class of problem.

A more thorough breakdown of each of the datasets is provided in the following section.

4.4.4 Sloan Sky Survey

In the Sloan Sky Survey dataset, logistic regression performed poorly with an f1 score of 0.6586. Our network performed much better with an f1 score of 0.9864, falling just short of the GBM's score of 0.9871. The score difference between the network and the GBM was not statistically significant. This dataset was the largest tested and contains very non-linear features, making it an ideal target for GBMs. Therefore, its promising that our model was able to keep pace, only missing the GBM's performance by a handful of instances.

Our architecture found a number of important non-linear features in this dataset, the most significant being red-shift. In astronomy, red-shift acts as a proxy for distance, and is intuitively one of the best ways to separate classes of astronomical objects. The relevant non-linearity exhibited a large spike at low-mid ranges of red-shift, suggesting that Galaxies occur most frequently at a distance between stars and quasars, which they do.

The network picked up on additional non-linear features such as declination angle, and a number of approximately linear features such as the g, i, and r bands. The network also zeroed out a number of unimportant variables, including the mjd, object id, and plate. This zeroing effect illustrates that our architecture is capable of performing automatic data-driven feature selection.

4.4.5 Breast Cancer

In the breast cancer dataset, the objective is to separate malignant and benign tumors using a set of input variables. This dataset is considerably smaller than the stellar dataset, but contains more highly correlated features. In this dataset, the our network outperformed the GBM with an f1 score of 0.9512 to 0.9383, however this difference was not statistically significant. The small sample size and large feature set made it more difficult not to overfit GBM, explaining the slightly lower score. Again, our target variable was not linearly separable, so logistic regression performed poorly with a f1 score of 0.3592, this result was statistically significant.

Our network found a number of important non-linear features, including concavity worst, texture worst, and symmetry worst. It also selected out a number of uninformative features, such as the patient id. This dataset demonstrates that the decrease in power from dropping the network's joint terms does not always negatively impact the performance of the model. On the contrary, by dropping the joint terms from the network, we made it significantly less able to overfit, which actually improved the network's performance.

4.4.6 Small Datasets

In addition to the two larger datasets above, we also tested our method on three smaller datasets: Iris, Titanic, and Wine. These datasets are all small-scale classification problems, containing only a few dozen samples in each category. Wine is notable for being almost linearly separable, and Titanic is notable for being very noisy.

Iris: In the Iris dataset, our architecture significantly outperformed logistic regression while slightly under-performing the GBM. Both differences were statistically significant. The most important classification features were petal length and petal width, which both exhibited significant marginal non-linearities.

Titanic: In the Titanic dataset, all three methods performed similarly, with logistic regression and GBMs slightly outperforming the marginal network. The marginal network picked up the important input variables class, age, and sex, suggesting that women, children, and first-class passengers had the best chance of survival. This analysis corresponds with the real-world understanding of the disaster, where women, children, and first-class passengers were given priority in boarding the life boats.

Wine: In the wine dataset, logistic regression slightly outperformed our architecture with f1 score of 0.9231 to 0.9189. GBMs fared worse with an f1 score of 0.8718, but none of these differences are statistically significant. Most terms identified by our network are linear, while the remaining two are linear over a domain and zero elsewhere. This observation suggests that a linear model may be the best approach for this dataset. GBMs under-performed due to the large input dimension (12) over a small number of samples (130), making it more difficult to avoid over-fitting.

4.5 Deep(ish) Networks

4.5.1 Background

Testing hypothesis **H6**: *Our network architecture can be combined with deep networks to form more interpretable models in deep learning tasks.*

This experiment examines the feasibility of combining our network with deep learning techniques to create more a interpretable deep neural network. Thus far, we've considered our network as a standalone entity similar to a random forest or a GAM. However, neural networks are well-known for their modularity and our implementation is no exception. By removing the sigmoid activation from the end of our network, we can create a marginal layer that can be inserted into different positions in a network.

In many high-dimensional deep learning tasks, the initial layers of the network often construct high-level representations of the input space which can be embedded as a lower dimensional vector. Auto-encoders (Baldi 2012) are a particularly pertinent example of this phenomenon, as a type of network architecture that specializes in low-dimensional representations. Oftentimes, these low-dimensional representations are interpretable to a human user and can provide some insights into the final layers of the model.

For instance, in image processing tasks, stacked convolutional and pooling layers are used to extract spatial features from a network. These features embed a low dimensional representation that is passed to one or more dense layers for the final analysis. In many applications, it is common to take a pre-trained network, freeze the weights, cut the dense layers off the end, and replace them with a new ready-to-train set of dense layers (Pelka et al. 2018).

In the case of one dense layer, the post-convolutional phase of the network has linear power, potentially allowing for interpretation of the inputs, but limiting the functions the network can model. More dense layers can approximate more complex functions, but the network loses interpretability. In this experiment, we explore the potential of adding a marginal layer to the end of a convolutional network, allowing the network to generalize to more complex functions while maintaining a high level of interpretability.

4.5.2 Setup

In this experiment, we train our neural network classifier on the pandemic dataset in two phases. In the first phase, we fit a convolutional network, known as the counter, to the images and predict the number t-cells and bacteria present. The first four layers of the counter employ 3-by-3 convolutions with stride one and output depths of size six, twelve, twelve, and eight. The network then uses 112-by-112 average pooling to convert the image into a vector of size eight, which is scaled by 1000 to speed training, and passed into a dense layer of output size two. Table 4.6 provides additional network architecture details.

Layer Type	Output Shape	Number of Parameters
3-by-3 Convolution	(118, 118, 6)	60
3-by-3 Convolution	(116, 116, 12)	660
3-by-3 Convolution	(114, 114, 12)	1308
3-by-3 Convolution	(112, 112, 8)	872
112-by-112 Average Pooling	(1, 1, 8)	0
Scaling (1/1000)	(1, 1, 8)	0
Flatten	(8)	0
Dense	(2)	18

Table 4.6: Counter network architecture. This network is designed to count the number of bacteria and t-cells on screen and output the results to the experimental layers.

The counter is trained on all 870 network images using the default Adam optimizer and mean squared error loss for 1000 epochs. No regularization or dropout was necessary. Once trained, the network weights are frozen and the counter is used as the input for our two experimental networks.

The first experimental network is a simple dense layer and sigmoid activation attached to the end of the counter network. The second experimental network attaches a marginal layer of width 12 before the dense layer to delinearize the network outputs. Both approaches are highly interpretable after the counting layers and the objective of this experiment is to determine which is more successful at correctly classifying infection lethality.

Both experimental networks are fit on a new set of 870 images with the goal of correctly classifying lethality. Both networks are trained using an Adam optimizer with a learning rate of 0.1 and a binary cross-entropy loss function for 5000 epochs. After training, a new set of 870 images is generated, and ROC AUC and F1 scores are reported for both models. Scatter plots are also included for visual analysis.

4.5.3 Results

Network Layer	Dense	Marginal
F1 Score	0.766	0.927
ROC AUC	0.847	0.976

Table 4.7: F1 and ROC AUC scores for the two networks. The marginal layer outperforms the dense layer significantly in both metrics.

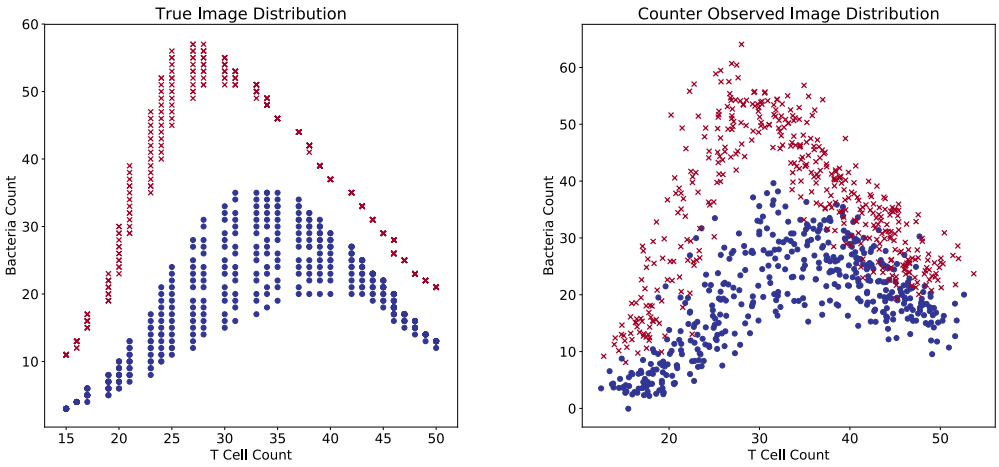


Figure 4.8: Differences between the true image distribution (left) and the distribution observed by the counter (right). While trends in the underlying distribution are still visible, imperfections in the counter network have caused the network outputs to wander from their real positions.

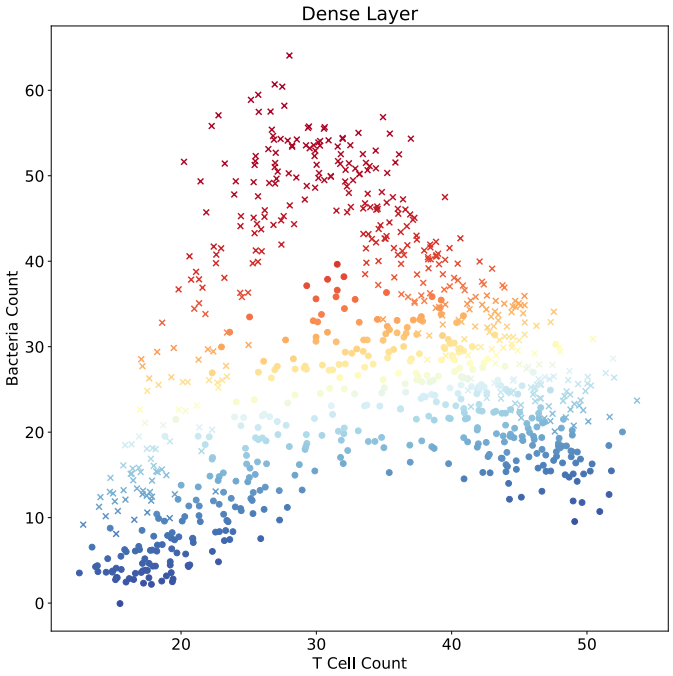


Figure 4.9: Final outputs for the single dense layer network. With an f1 score of 0.766 and a ROC AUC of 0.847, the single dense layer has missed the non-linearity in the counter's output and under-fit the data.

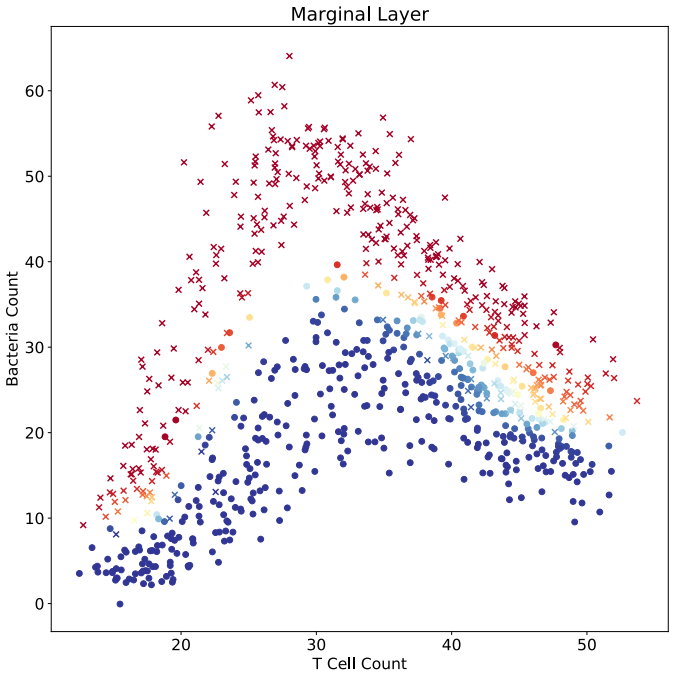


Figure 4.10: Final outputs for the marginal layer network. With an f1 score of 0.927 and a ROC AUC of 0.976, our marginal layer has correctly fit the non-linearity in the counter's output and provides a good model for the data.

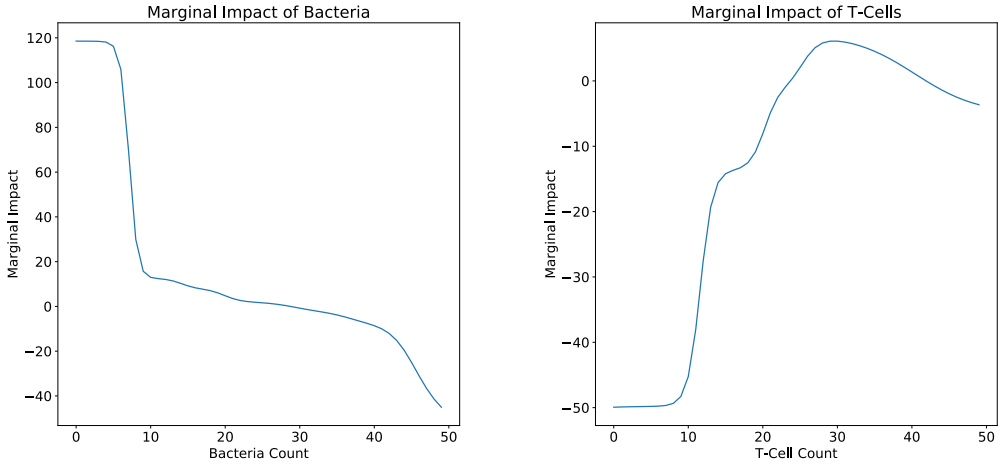


Figure 4.11: Marginal distributions for bacteria counts (left) and t-cell counts (right). The final layer of the network is behaving intuitively. Low bacteria counts radically increase the chance of survival while high bacteria counts radically lower it. In general, more bacteria is worse for survival. Low t-cell counts are bad for survival, but only if the bacteria counts are above ten. The positive marginal impact peaks around 30-35, where the infection is at its maximum point.

4.5.4 Analysis

This experiment supports our hypothesis that a marginal layer will outperform a single dense layer while still providing high transparency to the network outputs. As illustrated in Figure 4.9, the lower power of a single dense layer limits the network to a single linear classification boundary. This limitation greatly restricts the number of functions the network can approximate, and in this instance, this led to a sharp drop in accuracy.

The marginal layer performs much better, as illustrated in Figure 4.10, finding a smooth and effective classification boundary between the two sets of points. The model maintains high interpretability, as illustrated in Figure 4.11, behaving intuitively across the domain of both variables.

This experiment demonstrates that marginal layers can be effectively added into deeper networks and that these layers can improve interpretability without decreasing model accuracy. Because the marginal distribution is itself part of the network, it should be possible to further improve the accuracy of the model by unfreezing the initial weights and allowing the gradient to propagate through it. This change could provide minor improvements in instances where the pretrained model does not provide a perfect match for the outputs, or even to train the entire model completely from scratch.

Chapter 5

Conclusion

The contributions of this research are four-fold:

1. We have demonstrated that it is possible to improve interpretability in neural networks through the addition of GAM-like layers, and provided the first open source implementation to do so.

2. We have investigated some of the limitations of this architecture and demonstrated that those limitations often do not significantly impact performance in real-world classification tasks.

3. We have developed the Pandemic benchmark dataset for low-power small-sample image classification tasks.

4. We have demonstrated that marginal layers can be used effectively in conjunction with deeper neural networks.

This research has demonstrated a novel technique for improving interpretability in neural networks. By removing joint terms, we are able to simulate GAM-like behavior, providing results that closely match state-of-the-art techniques on many real-world classification problems. Our neural network architecture is a reliable and effective way to increase the power of logistic regression without sacrificing model interpretability. While the GAM paradigm does break down in

several contrived instances, our research has demonstrated that these situations are relatively uncommon in real-world tabular datasets.

In addition, we have shown that unlike traditional GAMs, our architecture can be applied within deeper networks. As illustrated in our pandemic experiment, applying marginal layers at the end of a deep network can help improve the transparency of the final output, while maintaining high levels of accuracy. The key to the technique is to find an effective and interpretable embedding that can be fed into a marginal layer that provides the final classification.

All the code used in this research is open sourced and publicly available online at https://github.com/zoox101/MarginalNetworks. The marginal layer code in Keras is also available at the end of this document.

Finally, like any machine learning technique, this architecture has a number of advantages and disadvantages compared to other approaches.

5.1 Advantages

Interpretability: Perhaps the most important advantage GAM-like networks is the strong interpretability of the outputs. Unlike dense networks, which are difficult to decode, the outputs from our architecture can be represented as a sum of one-dimensional functions. These one-dimensional functions are easy to visualize and interpret, making this architecture one of the most transparent available.

Overfitting Protection: Compared to dense networks, our architecture is less prone to overfitting, and thus requires less data to effectively train. Removing the joint connections reduces the network's degree of freedom by a factor of $n - 1$, making it harder for the network to exploit random noise.

Partial Automatic Feature Selection: Due partially to the decreased degrees of freedom and partially to a strong propensity to automatically zero outputs, this network architecture performs a basic form of feature selection during the marginal delinearization process. If a feature has no impact on classification, the delinearization step tends to push the marginal output to zero, as illustrated by the ID field in the sky survey example. This effect reduces the need for extensive data preprocessing before training the model.

5.2 Disadvantages

No Joint Terms: The obvious main drawback of this architecture is the lack of joint terms in the network, which breaks the assumptions required to make the network a universal approximator. While tests on real-world datasets suggest that this often has little impact on classification accuracy, there do exist situations, such as the XOR problem, that this architecture cannot handle.

Training Time: Training a neural network is always significantly slower than training tree-based methods. While our architecture performs better in some instances, traditional GAMs still train significantly faster and typically provide nearly identical results. We aim to improve on this issue in our future work.

Classification Only: In this work, we limited our experiments to classification tasks, as initial testing suggested that our architecture performs significantly worse than state-of-the-art on regression-type problems. This effect is likely caused by the increased impact of joint terms in regression tasks, as corroborating signals often have a diminishing effect on model output. However, additional work is needed to confirm or discount this hypothesis.

5.3 Future Work

While this work has demonstrated some of the advantages of this type of network architecture, this work has opened up at least two major avenues that I believe should be investigated in future research.

First, more investigation is needed to determine how well this technique applies to regression problems. I suspect that this type of network architecture will struggle in this domain, but to date no formal experiments have been run confirming that claim.

Second, more work needs to be done on the delinearization technique for this type of neural network. In our implementation, we rely on the implicit power of the network to force the delinearization of each of the network marginals. However, while a network of this type can theoretically model any arbitrary univariate function, it often has difficulty converging on the best one. Local minima cause issues for the network, and slow non-linear training times make convergence a pain. However, by implementing a version of the GAM b-spline delinearization technique within a marginal layer, I believe these training times can be improved tremendously.

Reference List

Allaire, J., 2019: R Interface to TensorFlow.
URL cran.r-project.org/web/packages/tensorflow/index.html

Bahdanau, D., K. Cho, and Y. Bengio, 2014: Neural Machine Translation by Jointly Learning to Align and Translate. *International Conference on Learning Representations*, International Conference on Learning Representations.

Baldi, P., 2012: Autoencoders, Unsupervised Learning, and Deep Architectures. *Proceedings of Machine Learning Research*, **27**, 37–49.

Belson, W. A., 1959: Matching and Prediction on the Principle of Biological Classification. *Applied Statistics*, **8**, 65.

Blanton, M. R., M. A. Bershady, and et al., 2017: Sloan Digital Sky Survey IV: Mapping the Milky Way, Nearby Galaxies, and the Distant Universe. *The Astronomical Journal*, **154**, 28–35.

Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone, 1984: *Classification and regression trees*. CRC Press, 1–358 pp.

Chilson, C., K. Avery, A. McGovern, E. Bridge, D. Sheldon, and J. Kelly, 2019: Automated detection of bird roosts using NEXRAD radar data and Convolutional Neural Networks. *Remote Sensing in Ecology and Conservation*, **5**, 20–32.

Chisholm, D. A., J. T. Ball, K. W. Veigas, and P. V. Luty, 1968: The Diagnosis of Upper-Level Humidity. *Journal of Applied Meteorology*, **7**, 613–619.

Chollet, F., 2019: Keras.
URL https://github.com/fchollet/keras

Clevert, D.-A., T. Unterthiner, and S. Hochreiter, 2015: Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *International Conference on Learning Representations*, International Conference on Learning Representations.

Cybenko, G., 1989: Mathematics of Control, Signals, and Systems Approximation by Superpositions of a Sigmoidal Function. *Math. Control Signals Systems*, **2**, 303–314.

Davis, T. B., 2018: *Real-Time Gesture Recognition With Mini Drones*. Ph.D. thesis, University of Oklahoma.

de Boor, C., 2006: *A Practical Guide to Splines*. Springer-Verlag, New York.

Domingos, P., 2012: A few useful things to know about machine learning. *Communications of the ACM*, **55**, 78.

Du, M., N. Liu, and X. Hu, 2018: Techniques for Interpretable Machine Learning. *CoRR*.

Duch, W. and N. Jankowski, 2000: Taxonomy of neural transfer functions. *IJCNN*, IEEE, 477–482.

Fisher, R. A., 1936: The Use of Multiple Measurements in Taxonomic Problems. *Annals of Eugenics*, **7**, 179–188.

Forina, M., 1991: UCI Machine Learning Repository: Wine Data Set. URL `archive.ics.uci.edu/ml/datasets/wine`

Friedman, J. H., 2001: Greedy function approximation: A gradient boosting machine. *Annals of Statistics*.

Fukushima, K., 1980: Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position. *Biological Cybernetics*, **36**, 193–202.

Galton, F., 1886: Regression Towards Mediocrity in Hereditary Stature. *The Journal of the Anthropological Institute of Great Britain and Ireland*, **15**, 246.

Glorot, X., A. Bordes, and Y. Bengio, 2011: Deep Sparse Rectifier Neural Networks. *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, G. Gordon, D. Dunson, and M. Dudík, eds., PMLR, Fort Lauderdale, FL, USA, volume 15 of *Proceedings of Machine Learning Research*, 315–323.

Hara, S. and K. Hayashi, 2016: Making Tree Ensembles Interpretable. *International Conference on Machine Learning*, International Conference on Machine Learning, New York.

Hastie, T. and R. Tibshirani, 2007: Generalized Additive Models. *Statistical Science*, **1**, 314–318.

Hastie, T., R. Tibshirani, and J. Friedman, 2009: *The Elements of Statistical Learning*. Springer.

Hayashi, F., 2000: *Econometrics*. Princeton University Press.

Ho, T. K., 1995: Random Decision Forests. *Proceedings of the 3rd International Conference on Document Analysis and Recognition*, AT&T Bell Laboratories, Montreal, QC, 278–282.

—, 2002: A Data Complexity Analysis of Comparative Advantages of Decision Forest Constructors. *Pattern Analysis & Applications*, **5**, 102–112.

Kingma, D. P. and J. Ba, 2014: Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*.

Kingston, G. B., H. R. Maier, and M. F. Lambert, 2004: A Statistical Input Pruning Method for Artificial Neural Networks Used in Environmental Modelling. *International Congress on Ecological Modeling Software*, volume 34.

Kullback, S. and R. A. Leibler, 1951: On Information and Sufficiency. *The Annals of Mathematical Statistics*, **22**, 79–86.

Lagerquist, R., A. McGovern, and T. Smith, 2017: Machine Learning for Real-Time Prediction of Damaging Straight-Line Convective Wind. *Weather and Forecasting*, **32**, 2175–2193.

Louppe, G., L. Wehenkel, A. Sutera, and P. Geurts, 2013: Understanding variable importances in forests of randomized trees. *Advances in Neural Information Processing Systems*, Curran Associates, Inc., volume 26, 431–439.

Maas, A. L., A. Y. Hannun, and A. Y. Ng, 2013: Rectifier Nonlinearities Improve Neural Network Acoustic Models. *Proceedings of Machine Learning Research*, **30**, 3.

Martin, A., A. Ashish, B. Paul, B. Eugene, C. Zhifeng, and et al., 2015: TensorFlow Large-Scale Machine Learning on Heterogeneous Systems.

McGovern, A., K. L. Elmore, D. J. Gagne, and et al., 2017: Using Artificial Intelligence to Improve Real-Time Decision-Making for High-Impact Weather. *Bulletin of the American Meteorological Society*, **98**, 2073–2090.

Minsky, M. and S. Papert, 1969: *Perceptrons; an introduction to computational geometry*. MIT Press, Cambridge.

Molnar, C., 2019: *Interpretable Machine Learning*. Github.
URL `christophm.github.io/interpretable-ml-book/`

Natekin, A. and A. Knoll, 2013: Gradient boosting machines, a tutorial. *Frontiers in Neurorobotics*, **7**, 21.

Nelder, J. A. and R. W. M. Wedderburn, 1972: Generalized Linear Models. *Source: Journal of the Royal Statistical Society. Series A (General)*, **135**, 370.

Nielsen, M. A., 2015: *Neural Networks and Deep Learning*. Determination Press, San Francisco.

Norton, S. W., 1989: Generating Better Decision Trees. Technical report, Siemens Corporate Research.

Olden, J. D. and D. A. Jackson, 2002: Illuminating the "black box": a randomization approach for understanding variable contributions in artificial neural networks. *Ecological Modelling*, **154**, 135–150.

Özesmi, S. L. and U. Özesmi, 1999: An artificial neural network approach to spatial habitat modelling with interspecific interaction. *Ecological Modelling*, **116**, 15–31.

Paszke, A., S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, 2017: Automatic differentiation in PyTorch. *NIPS-W*.

Pedregosa, F., G. Varoquaux, A. Gramfort, and Et al., 2011: Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.*, **12**, 2825–2830.

Pelka, O., F. Nensa, and C. M. Friedrich, 2018: Annotation of enhanced radiographs for medical image retrieval with deep convolutional neural networks. *PloS one*, **13**.

Perry, R. D. and J. D. Fetherston, 1997: Yersinia pestis–etiologic agent of plague. *Clinical microbiology reviews*, **10**, 35–66.

Plumb, G., M. Al-Shedivat, E. Xing, and A. Talwalkar, 2019: Regularizing Black-box Models for Improved Interpretability. *CoRR*, **abs/1902.0**.

Plumb, G., D. Molitor, and A. Talwalkar, 2018: Model Agnostic Supervised Local Explanations. *CoRR*, **abs/1807.0**.

Ribeiro, M. T., S. Singh, and C. Guestrin, 2016: "Why Should I Trust You?": Explaining the Predictions of Any Classifier. *International Conference on Knowledge Discovery and Data Mining*, ACM Press, New York, 1135–1144.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams, 1985: Learning Internal Representations by Error Propagation. Technical report, Institute for Cognitive Science.

Santosa, F. and W. W. Symes, 1986: Linear Inversion of Band-Limited Reflection Seismograms. *SIAM Journal on Scientific and Statistical Computing*, **7**, 1307–1330.

Servén, D. and C. Brummitt, 2018: pyGAM: Generalized Additive Models in Python.

Srivastava, N., G. Hinton, A. Krizhevsky, and R. Salakhutdinov, 2014: Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, **15**, 1929–1958.

Stanford, 2019: A Titanic Probability.
URL `web.stanford.edu/class/cs109/titanic.html`

Street, W. N., W. H. Wolberg, and O. L. Mangasarian, 2005: Nuclear feature extraction for breast tumor diagnosis. *Biomedical Image Processing and Biomedical Visualization*, SPIE, volume 1905, 861–870.

Suárez, E., C. M. Pérez, R. Rivera, and M. N. Martínez, 2017: Weighted Least-Squares Linear Regression. *Applications of Regression Models in Epidemiology*, John Wiley & Sons, Inc., Hoboken, NJ, USA, 117–128.

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, 2015: Going deeper with convolutions. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*.

TensorFlow, 2019: TensorFlow.js.
URL `https://www.tensorflow.org/js`

Tibshirani, R., 1994: Regression Shrinkage and Selection Via the Lasso. *Journal of the Royal Statistical Society, Series B*, **58**, 267–288.

Tikhonov, A. N. and V. Y. Arsenin, 1977: Solutions of Ill-Posed Problems. *SIAM Journal on Scientific and Statistical Computing*, **21**, 266–267.

Xu, B., N. Wang, T. Chen, and M. Li, 2015: Empirical Evaluation of Rectified Activations in Convolutional Network. *CORR*.

Zou, H. and T. Hastie, 2005: Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society*, **67**, 301–320.

Chapter 6
Appendix

Listing 6.1: Marginal Layer Code

```
from keras import backend as K
from keras.layers import Layer

#Marginal Layer Class for Keras
class MarginalLayer(Layer):

    #Saving input variables and initializing layer
    def __init__(self, hidden_units=12, **kwargs):
        self.hidden_units = hidden_units
        super(MarginalLayer, self).__init__(**kwargs)

    #Builds the marginal layer
    def build(self, input_shape):

        #Initializers
        winit = keras.initializers.RandomNormal(mean=1.0, stddev=0.05, seed=None)
        binit = keras.initializers.RandomNormal(mean=0.0, stddev=0.05, seed=None)
        oinit = keras.initializers.RandomNormal(mean=0.0, stddev=0.05, seed=None)

        #Creating input weight matrix
        self.win = self.add_weight(name = 'marginal_input_weight',
                                   shape = (input_shape[1], self.hidden_units),
                                   initializer = winit)

        #Creating the input bias matrix
        self.bin = self.add_weight(name = 'marginal_input_bias',
                                   shape = (input_shape[1], self.hidden_units),
                                   initializer = binit)

        #Creating the output weight matrix
        self.wout = self.add_weight(name = 'marginal_output_weight',
                                   shape = (input_shape[1], self.hidden_units),
                                   initializer = oinit)

        #Creating the output bias matrix
        self.bout = self.add_weight(name = 'marginal_output_bias',
                                   shape = (input_shape[1], self.hidden_units),
                                   initializer = binit)

        #Building layer
        super(MarginalLayer, self).build(input_shape)

    #Calls the layer on the input
    def call(self, x):

        #Marginal calculations
        a = K.expand_dims(x, axis=2)

        #Expanding win along axis 0 and multiplying it with variable a and adding a bias
        b = a * K.expand_dims(self.win, axis=0) + K.expand_dims(self.bin, axis=0)

        #Passing b through an activation, multiplying by wout, and adding a bias
        c = K.sigmoid(b) * K.expand_dims(self.wout, axis=0) + \
            K.expand_dims(self.bout, axis=0)

        #Adding lienar term back into the marginal
        c = K.concatenate([c, a], axis=2)

        #Summing along the second axis
        d = K.sum(c, axis=2, keepdims=False)

        #Returning marginal output
        return d

    #Computing the output shape
    def compute_output_shape(self, input_shape):
        return input_shape
```