Boston College

Department of Economics
Independent Study in Neural Networks

Extreme Event Forecasting: Using Artificial Neural Networks to Predict Bankruptcy in Poland

James LeDoux

Spring 2017

1 Introduction

Binary classification is among the most common and useful statistical learning tasks. Countless problems exist in which it is beneficial to some party to be able to form a reliable estimation of whether an event will occur. A small subset of such problems include predicting whether a stock will increase in price, whether an image contains a human face, and whether or not an email message is spam.

A particularly challenging and valuable-if-solved class of binary classification problem is that of extreme event forecasting. Imbalanced data, in which one class is significantly more commonly observed than another, is difficult to model, as a trained model will very seldom predict a high probability of the rare event occurring. While extreme outcomes are challenging to predict in a binary dependent variable model, they are also often quite valuable if predicted reliably. The odds of a patient contracting a rare disease, and of an early-stage company eventually having a public offering, are both examples of extreme events with either life-and-death consequences or significant financial award attached to them.

In this paper, I test the ability of neural network models to predict one specific type of extreme event: bankruptcy. Bankruptcy is the perfect example of a problem that is both rarely observed and valuable to predict. Further, we are limited by the number of companies that exist and go bankrupt, making it nearly impossible to gather additional data on this problem. Given the constraints of the rarity of this event's occurrence and the relatively small and restricted amount of data that exists on the matter, an improvement in the state of the art for this problem could be of interest to financial professionals and academics alike.

2 Data

The data used in this experiment were donated to the UCI Machine Learning Repository by Sebastian Tomczak. Tomczak donates data on the financial ratios of Polish companies, and a binary response variable for whether the company went bankrupt within a set number of years (Zieba, Tomczak, and Tomczak 2016). The donated data exists with varying time lags. For this study, I use only the five-year time lag, predicting the odds of a company going bankrupt within five years. This data consisted of 7027 observations, where each observation is a company. Of these 7027 companies, 271 (3.9%) went bankrupt within the five year period that followed the reporting of the financial ratios in this data. Despite having a large number of observations, the small number of positive classifications (bankruptcy=1) posed a significant challenge during learning.

The financial ratios in the data are reported in Table 1.

In order to make this data useable for analysis, a few preprocessing steps are necessary. Most importantly, one must deal with the outlier problem before a model will be able to learn from this data. Extreme outliers are a common problem with

Table 1: Independent Variables

ID	Description	ID	Description
X1	net profit / total assets	X33	operating expenses / short-term liabilities
X2	total liabilities / total assets	X34	operating expenses / total liabilities
X3	working capital / total assets	X35	profit on sales / total assets
X4	current assets / short-term liabilities	X36	total sales / total assets
X5	dropped for length	X37	(current assets - inventories) / long-term liabilities
X6	retained earnings / total assets	X38	constant capital / total assets
X7	EBIT / total assets	X39	profit on sales / sales
X8	book value of equity / total liabilities	X40	(current assets - inventory - receivables) / short-term liabilities
X9	sales / total assets	X41	total liabilities / ((profit on operating activities + depreciation) * (12/365))
X10	equity / total assets	X42	profit on operating activities / sales
X11	(gross profit + ext. items + fin. expenses) / tot. assets	X43	rotation receivables + inventory turnover in days
X12	gross profit / short-term liabilities	X44	(receivables * 365) / sales
X13	(gross profit + depreciation) / sales	X45	net profit / inventory
X14	(gross profit + interest) / total assets	X46	(current assets - inventory) / short-term liabilities
X15	(total liabilities * 365) / (gross profit + depreciation)	X47	(inventory * 365) / cost of products sold
X16	(gross profit + depreciation) / total liabilities	X48	EBITDA (profit on operating activities - depreciation) / total assets
X17	total assets / total liabilities	X49	EBITDA (profit on operating activities - depreciation) / sales
X18	gross profit / total assets	X50	current assets / total liabilities
X19	gross profit / sales	X51	short-term liabilities / total assets
X20	(inventory * 365) / sales	X52	(short-term liabilities * 365) / cost of products sold)
X21	sales (n) / sales (n-1)	X53	equity / fixed assets
X22	profit on operating activities / total assets	X54	constant capital / fixed assets
X23	net profit / sales	X55	working capital
X24	gross profit (in 3 years) / total assets	X56	(sales - cost of products sold) / sales
X25	(equity - share capital) / total assets	X57	(current assets - inventory - ST liabilities) / (sales - GP - depreciation)
X26	(net profit + depreciation) / total liabilities	X58	total costs /total sales
X27	profit on operating activities / financial expenses	X59	long-term liabilities / equity
X28	working capital / fixed assets	X60	sales / inventory
X29	logarithm of total assets	X61	sales / receivables
X30	(total liabilities - cash) / sales	X62	(short-term liabilities *365) / sales
X31	(gross profit + interest) / sales	X63	sales / short-term liabilities
X32	(current liabilities * 365) / cost of products sold	X64	sales / fixed assets

financial data, particularly when dealing with distressed companies. A poor sales quarter or catastrophic loss of assets, for example, could move the denominator of one of the ratios in this data toward zero, causing a massive value for one or more features. To remedy this problem, I employ a strategy of data windorization as was performed in Fijorek and Grotowski (2012). This involves identifying observations that are more than 1.5 times the interquartile range outside one of a variable's outer quartiles, and replacing them with the upper (lower) quartile plus (minus) 1.5 times the interquartile range (Fijorek and Grotowski 2012). With extreme outliers modified, it was then possible for a model to identify signals of financial risk in the data.

Additional to the windorization procedure, I also scale all features to mean zero and unit variance. Last, I impute missing values with their column means.

3 Performance Metric

Due to the nature of imbalanced data, accuracy is no longer an appropriate metric to optimize. Even a naive classifier that never predicts bankruptcy, for example, would achieve an accuracy score of almost 97%. For this reason, I measure my results using the receiver operating characteristic (ROC). The ROC curve is created by plotting true positive rate against false positive rate at varying classification thresholds. The benefit of using this statistic is that the area under the ROC curve (AUC score) will not give a high score to a naive classifier with imbalanced data. This score is bounded

between 0.5 and 1, where a higher score indicates a better model. A model that has learned nothing from the data will score close to 0.5, where a perfect model scores close to 1.0. Intuitively, the AUC score represents the probability that a randomly chosen positive example will be ranked higher than a randomly chosen negative sample by the model (Tape, n.d.).

4 Neural Networks

4.1 Theoretical Underpinnings

Deep feedforward networks, also called multilayer perceptrons (MLPs), were the first class of deep learning model developed in the 1970s. The goal of a feedforward network is to approximate a function f^*, defining a mapping $y = f(x;\theta)$ that learns the best parameters θ for approximating f^* (goodfellow et al. 2017).

The "network" part of the name comes from the fact that a neural network is typically comprised of several stacked functions, commonly referred to as layers. Given functions f^1, f^2, and f^3, for example, a feedforward network could be defined as $f^3(f^2(f^1))$, where the output of f^1 is fed into f^2, which is then fed into f^3, which in turn produces the network's output. The initial layer of this network, which receives the input data, is called the input layer. The final layer, producing the output, is referred to as the output layer. All layers in-between these two are referred to as hidden layers (goodfellow et al. 2017). In the case of binary classification tasks such as that being attempted in this paper, the final layer will be a sigmoid function that takes the output of the model's penultimate layer and feed it through a function bounded between zero and one.

Similar to other machine-learned models, a feedforward network learns by gradient descent; that is, by iteratively updating its weights according to the gradient of a cost function that it attempts to optimize. The key difference between gradient descent in neural networks and in simpler algorithms is that the stacked nature of a neural network's layers and weights means that additional steps are required in order to compute the gradients with respect to each layer's weights. The learning procedure for a feedforward network, in brief, is:

1 **Feed Forward:** propagate the model's input through the network by multiplying each layer's input by the layer's weight vector w^l and passing this through an activation function σ, obtaining output $\hat{y}$

2 **Calculate Cost Function:** given model output $\hat{y}$ and target value y, calculate the model's cost function $C(\hat{y}, y)$, where C can be the mean squared error, accuracy, cross entropy, or any other cost function.

3 **Backpropagate the Error:** for each layer $l = L-1, L-2, ..., 2$ compute the error $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma' z^l$, where σ is the activation function and z^l is the weighted

input to the neurons in layer l. Effectively, what this is doing is taking the known error at layer $l+1$ and propagating it backward through the weights and activation functions of the previous layers of the network. This allows us to calculate the gradient of the cost function $\frac{\partial C}{\partial w^l_{jk}}$ where l represents a layer and k represents an individual weight (Nielsen 2015).

4 **Update Weights:** with the gradient of the cost function now calculated with respect to each hidden unit, perform the standard gradient descent step of either adding or subtracting the product of the model's learning rate and weights to the pre-existing weights at each layer. Whether this is an addition or subtraction step depends on whether the cost function is being maximized or minimized.

For an in-depth discussion of the backpropagation algorithm, see Nielsen (2015). Many different specifications of these models exist, some of which will be discussed in the following section.

5 Hyperparameter Tuning and Model Architectures

I test a variety of different network architectures for this extreme-event binary classification task, tuning the models' respective hyperparameters in order to achieve optimal performance. In line with universal approximation theorem's claim that the size and quantity of layers affects model accuracy, I begin by testing model size and depth (Hornik et al. 1989). I test three different between-layer activation functions as well, as these play an important role in learning. In order to achieve better out-of-sample generalization, I test two different regularization strategies: Dropout and batch normalization. Finally, in an attempt to achieve an efficient learning procedure, I also test various optimization procedures, including stochastic gradient descent and the adaptive learning rate algorithms Adam and RMSprop. Gaining an idea of what might be successful from these tests, I run a final parameter search over the architectures and parameters that have shown promise in order to see which model specification will be most successful.

5.1 Layer Size

Universal approximation theorem shows that neural networks are universal function approximators. This means that, given a sufficient number of neurons, a neural network can achieve any desired level of accuracy on a dataset (Hornik et al. 1989).

Adding weights to a neural network, however, is not a free lunch. First, while it is theoretically possible to approximate any function with this class of model, the number of weights required to do so may be prohibitively large. Neural models become

increasingly challenging and slow to train as weights are added, so this becomes problematic as the number of neurons per layer enters the thousands.

Second, universal approximation theorem only applies to in-sample accuracy. While a function can be approximated to an arbitrary degree of accuracy in-sample, this does not guarantee strong out of sample performance. A perfectly fit model within-sample, after all, is likely overfit.

For these reasons, it is important to test for the optimal number of weights per layer. Here I test the out of sample accuracy and AUC score of a single-layer model with layer sizes ranging between 4 and 1024 weights. All models had only one hidden layer, and were trained using the Adam optimizer (see section 5.6) with ReLU activation functions (section 5.2) and the binary cross-entropy loss function (Table 2).

Table 2: Relationship between Layer Size and Model Performance

Number of Neurons	Accuracy (%)	AUC
4	97.12	0.68
8	96.94	0.65
16	97.22	0.69
32	97.15	0.67
64	97.29	0.73
128	97.58	0.73
256	97.54	0.73
512	97.51	0.73
1024	97.58	0.72

While the smaller layer sizes of 4 through 32 nodes per layer appear to be too little, showing sub-73% AUC scores, all layer sizes above these values showed similar levels of success. Viewing the how the accuracy and loss scores changed during training, we can see that the model does not approach the ability to seriously overfit the data until it reaches 64 units per hidden layer (Figures 1 - 6). The ability to overfit is a desirable quality to have in a model, because this means that it is not missing out on accuracy due to under-fitting. Over-fitting in such models can be combatted by early stopping, regularization, or both. For this reason, a model should have at least 64 units per hidden layer when trained on this data set. It is worth noting, however, that adding nodes significantly increases training time.

5.2 Number of Layers

Adding layers to a model allows it to learn increasingly complex, nonlinear patterns in the data. An early example of this from the literature is the XOR problem. The "exclusive or", otherwise referred to as XOR, is a binary classification problem where, given two values x and y, a point is only classified as true when one, but not both values equal one. Minsky and Papert (1969) show that a single-layer neural network

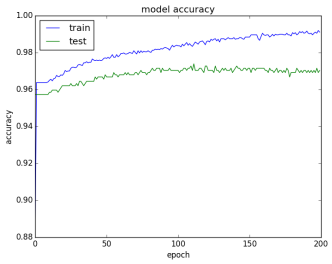

Figure 1: Training and Test Accuracy with 16 Neurons

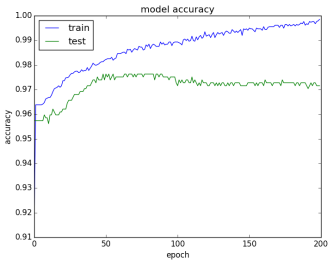

Figure 2: Training and Test Accuracy with 32 Neurons

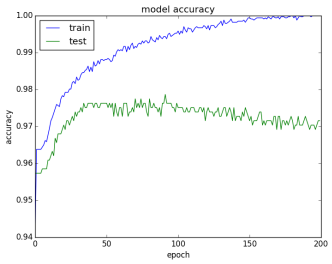

Figure 3: Training and Test Accuracy with 64 Neurons

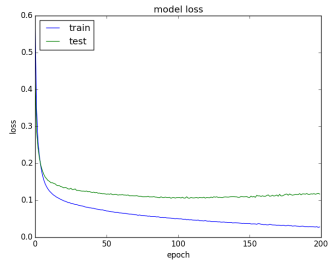

Figure 4: Training and Test Loss with 16 Neurons

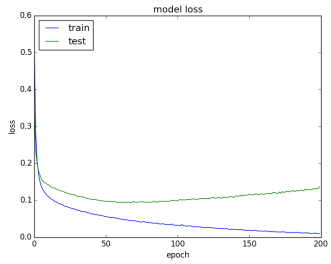

Figure 5: Training and Test Loss with 32 Neurons

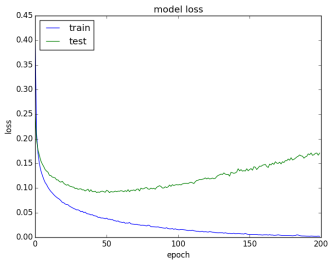

Figure 6: Training and Test Loss with 64 Neurons

is incapable of solving this problem, as the classes are not linearly separable. Adding additional layers to such a model, however, introduces the nonlinearity required in order to solve this problem.

While not all deep learning tasks are as simple as this two-dimensional classification example, the XOR problem serves as a minimal example of how adding layers to a network may help to solve complex tasks. Adding layers to a network allows a model to learn more complex interactions between features, providing more of an opportunity for the model to learn the underlying structure of the training data.

For this reason, I test the impact of adding layers to a simple neural network. Using 64 weights per layer, I test models with between one and eight hidden layers (Table 3). All models for this test use ReLU activation functions, the Adam optimizer, and the binary cross entropy cost function.

Table 3 shows that adding layers to this model improves its performance until a certain point, and then begins to make it less accurate out of sample. This indicates that the reduced bias of the model is improving its performance when adding a second and third layer, but after this point it begins to overfit. This is the exact pattern one would expect due to the bias-variance tradeoff, because each added layer both reduces bias and increases variance. For this reason, adding layers to a model is not a catch-all solution. Here it appears that somewhere between three and five layers is optimal.

Table 3: Relationship between Number of Layers and Model Performance

Number of Layers	Accuracy (%)	AUC
1	96.94	0.63
2	97.12	0.65
3	96.94	0.67
4	96.91	0.62
5	96.94	0.69
6	96.87	0.62
7	96.79	0.64
8	96.86	0.62

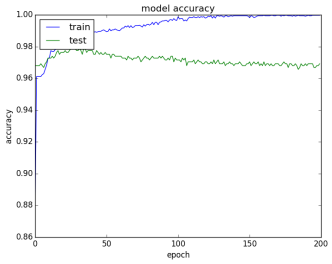

Figure 7: Training and Test Accuracy with 1 Layer

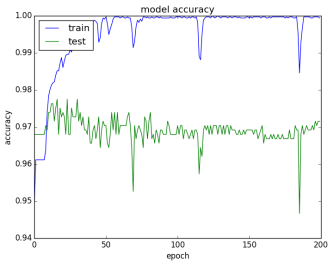

Figure 8: Training and Test Accuracy with 2 Layers

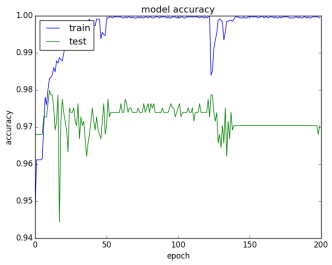

Figure 9: Training and Test Accuracy with 3 Layers

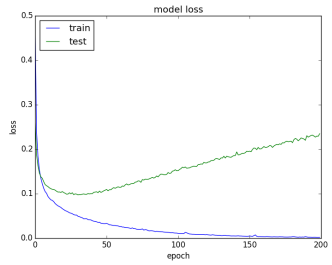

Figure 10: Training and Test Loss with 1 Layer

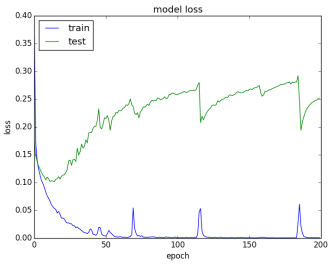

Figure 11: Training and Test Loss with 2 Layers

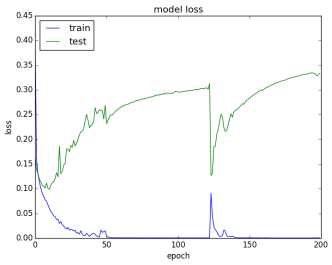

Figure 12: Training and Test Loss with 3 Layers

Figures 7 - 12 show that adding layers makes a model prone to achieve an optimal fit earlier-on during training. The training procedure also becomes more complicated, as can be seen by the noisier patterns followed by the accuracy and loss curves of the networks with more than one hidden layer. Both of these problems can be partially remedied by regularization, which is tested in a later section of this paper.

5.3 Activation Functions

An additional parameter important to a model's performance is its activation function. All weight-input products in a neural network are fed through a nonlinear

activation function, as otherwise the network would produce a linear decision boundary (Raschka 2016). Different activation functions, however, exhibit different performance characteristics, with some being preferable to others. Here I test three different activations on this classification task.

The first activation I test is the logistic sigmoid function. The sigmoidal activation was the original deep learning activation function, being a nonlinear function that is already recognized by most, if not all, in the statistical learning field. The function is defined as:

$$\sigma(X) = \frac{1}{1 + \exp(-X)}.$$

While this activation was common in the early years of deep learning, it has recently fallen out of favor for computational reasons. Because the function smooths out at its tails, its derivative is close to zero for most of the function's domain. As a result of this, computers sometimes face problems of numerical overflow with this function, where the gradient is rounded to zero. This quality can harm model performance during training, as saturated gradients slow the learning process (Karpathy 2016).

The second activation I test is the hyperbolic tangent function (tanh). The tanh activation is defined as:

$$\tanh(X) = \frac{\exp(X) - \exp(-X)}{\exp(X) + \exp(-X)},$$

or alternatively:

$$2\sigma(X) - 1.$$

This function is typically preferred to the logistic sigmoid for a variety of reasons. First, because it is bounded between -1 and 1, it will commonly output values close to zero, whereas the logistic sigmoid will more commonly output values close to 0.5 (Lecun et al. 1998). This quality helps the network decide which features are most important. Second, this function tends to have larger gradients than the logistic sigmoid if inputs to the network are normalized, and this is known to help models find optimal solutions quicker.

The final activation function I test is quickly becoming one of the most commonly used: the Rectified Linear Unit (ReLU). This function computes:

$$\text{ReLU}(X) = \max(0, X),$$

making the function linear after $x = 0$ and equal to zero everywhere else. The main benefit of the ReLU is that it's quick to compute, speeding up stochastic gradient descent by up to a factor of six (Krizhevsky et al. 2012). The downside, however, is

that the unit can be fragile, with the units often "dying" and becoming stuck being equal to zero during training (Karpathy 2016).

The results of testing these three activation functions are shown in Table 4. Each model in this table was a network with three hidden layers of 128 neurons each.

Table 4: Activation Function Results

Activation Function	Accuracy (%)	AUC
Tanh	97.54	0.74
ReLU	97.54	0.73
Sigmoid	97.43	0.70

As expected, the sigmoidal units had the worst results of the three activation functions. The tanh and ReLU activations performed similar to one another, but the tanh activations had a smoother, more stable training curve than was observed in the ReLU model (Figures 13 - 18).

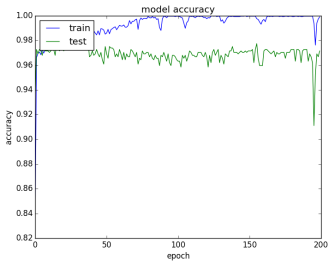

Figure 13: Training and Test Accuracy with Tanh Activations

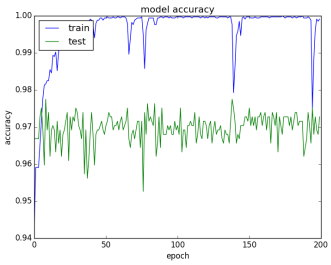

Figure 14: Training and Test Accuracy with ReLU Activations

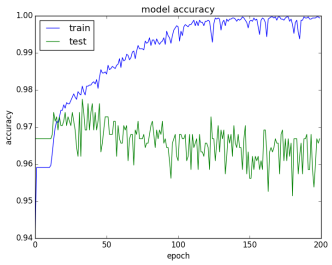

Figure 15: Training and Test Accuracy with Sigmoid Activations

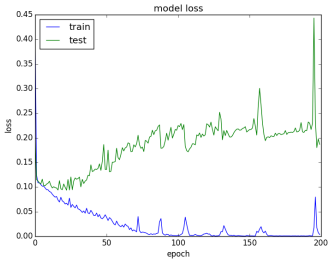

Figure 16: Training and Test Loss with Tanh Activations

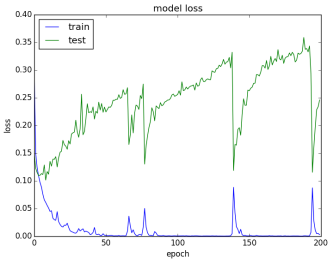

Figure 17: Training and Test Loss with ReLU Activations

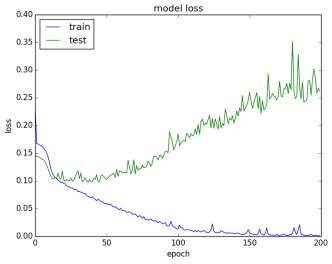

Figure 18: Training and Test Loss with Sigmoid Activations

5.4 Regularization: Dropout

My next consideration in model-building was regularization. Regularization works by trading increased bias for reduced variance, serving as a remedy to the overfitting

problem. A tried and true strategy for building successful neural networks is to use large, appropriately regularized models (goodfellow et al. 2017).

The first regularization strategy I test is Dropout. First introduced by Srivastava et al. (2014), Dropout provides a method for regularizing a broad family of models. This method works by "dropping out" a fixed percentage of nodes at random during each training iteration, forcing those weights to equal zero. The impact of dropping out nodes at random in this way is that it forces the network to learn representations of the data in its hidden layers that are robust to missing or modified information. If a network is being trained to recognize faces in images, for example, the network cannot learn to rely too heavily on an individual neuron that has learned to recognize individual features in the data such as noses or ears. If this individual neuron is required for the model's success and it is dropped out, the model will have poor performance. As a result of this constraint, models trained using a dropout strategy must either learn more robust representations of features (i.e. have multiple nodes serve the purpose of recognizing a nose on a face, rather than just one), or learn more general representations of the data that still hold predictive accuracy on previously unseen data with potentially different properties (i.e. being able to recognize a face without a nose).

If we consider each combination of non-dropped-out weights to be a unique model, this regularization strategy can then be viewed as a bagging method, where each iteration trains a unique model that shares parameters with every other model (goodfellow et al. 2017). Each iteration, borrowing some weights from the previous model and given a fresh set of constraints via its new set of dropped-out nodes, is encouraged to learn something slightly different than the previous. The end result is often a model with superior generalization to unseen data. Dropout increases a model's bias due to the constraints imposed by setting nodes equal to zero during training, but also decreases its variance for this same reason.

Here I test models using between-layer dropout of between 0 and 50% of nodes. All models consisted of three hidden layers, each with 1024 units (Table 5). All models used the Adam optimizer with hyperbolic tangent activation functions and the binary cross entropy cost function.

Table 5: Performance of Models using Dropout

Percent of Weights Dropped Out (%)	Accuracy (%)	AUC
0	97.54	0.75
10	97.40	0.75
20	97.23	0.71
30	97.44	0.73
40	97.51	0.74
50	97.80	0.75

The results of this testing shown in Table 5 demonstrate the value of regularizing a large model. Despite placing significant constraints on the model during training, the model with half of its units dropped out during each training iteration was able

to match the performance of the model using no dropout at all. While the zero and ten percent dropout models' relative success makes this a somewhat complicated set of results to interpret, we can see from the improvements shown from moving from 20 to 30, 30 to 40, and then 40 to 50% dropout that adding additional regularization strength can improve the performance of a large model.

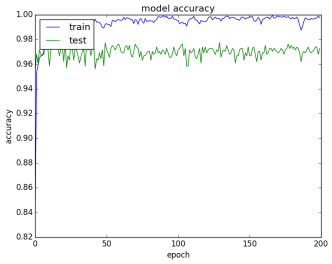

Figure 19: Training and Test Accuracy with 30% Dropout

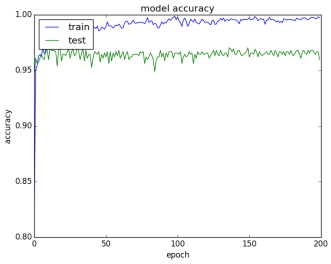

Figure 20: Training and Test Accuracy with 40% Dropout

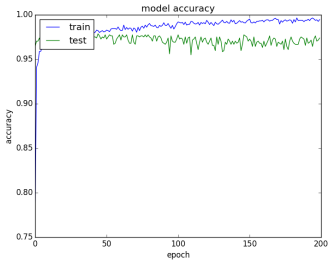

Figure 21: Training and Test Accuracy with 50% Dropout

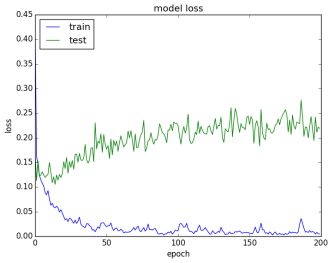

Figure 22: Training and Test Loss with 30% Dropout

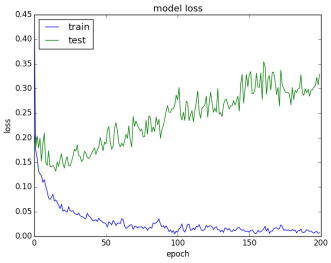

Figure 23: Training and Test Loss with 40% Dropouts

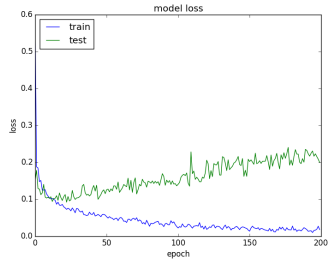

Figure 24: Training and Test Loss with 50% Dropout

Figures 19 through 24 show qualitatively the benefit of regularizing a neural network. First, we see that as additional dropout is added, the training and test accuracy curves move closer together. Second, we see that increasing the regularization of these models noticeably smooths out the test-loss curve, allowing loss to improve for a larger number of training steps and reducing the rate at which overfitting occurs as training continues past the optimal point. These are desirable qualities, indicating that dropping out 40 or 50% of weights during training is an effective strategy for this model.

5.5 Regularization: Batch Normalization

Part of the challenge of learning in neural models is the fact that the distribution of each layer's inputs is constantly changing during training, being dependent on both the outputs of prior layers in the network and the specific batch of data passing

through the training iteration. Ioffe and Szegedy (2015) refer to this problem as internal covariate shift. Batch normalization is an attempt to remedy this by normalizing the inputs to each layer before the data are passed through its affine transformation and activation function (Ioffe and Szegedy 2015).

Batch normalization is said to both reduce the number of training iterations required in order to find an optimal model and also improve out of sample accuracy (Ioffe and Szegedy 2015). Further, this method is also said to have a regularizing effect, improving generalization by making individual training iterations less dependent on the specific data drawn in that particular iteration (Ioffe and Szegedy 2015).

Table 6: Batch Normalization Results

	Accuracy (%)	AUC
No Batchnorm	97.01	0.76
Batchnorm	97.19	0.79

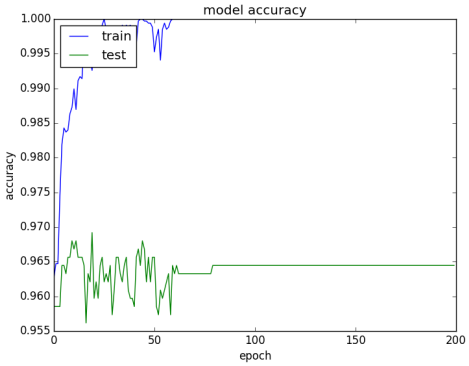

Figure 25: Training and Test Accuracy without Batch Normalization

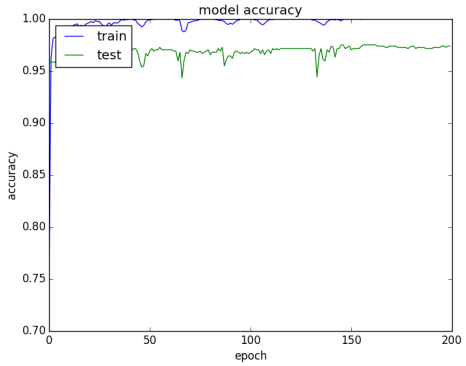

Figure 26: Training and Test Accuracy with Batch Normalization

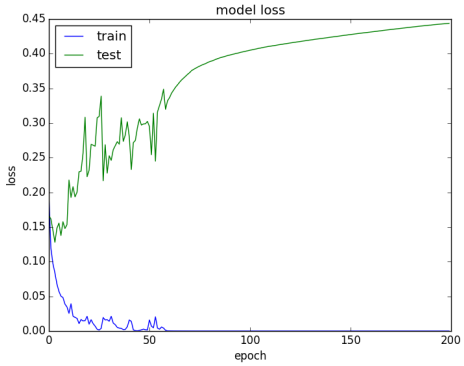

Figure 27: Training and Test Loss without Batch Normalization

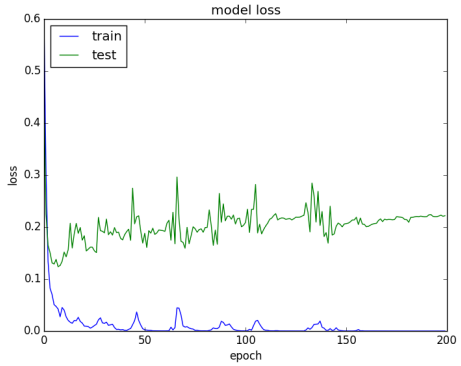

Figure 28: Training and Test Loss with Batch Normalization

The results in Table 6 show that batch normalization does indeed help when applied to this data. This strategy seems particularly appropriate for use with financial data because of the distribution-related problems discussed in Section 2.

5.6 Optimization: Adaptive Learning Rates

The final model-related strategy I test is learning rate procedure. While early neural networks were trained via either batch or stochastic gradient descent with either static or linearly cooled learning rates, many successful models today employ some form of adaptive learning rate algorithm. The reason for using these algorithms is that, while a larger learning rate is desirable early-on in training, a smaller learning rate becomes desirable as the model approaches a local optimum in its objective function so that it can hone in on the optimal point and avoid leaving this section of the parameter space. Neural networks' objective functions are highly nonconvex, so it is ideal for a model's learning rate to be highly flexible. For this reason, a variety of adaptive learning rate algorithms exist. For an overview of the available gradient descent optimization algorithms and how they work, see Ruder (2016).

Here I test a static learning rate, a decaying learning rate, and the popular adaptive learning rate algorithms Adam and RMSprop. The results of these tests are shown in Table 7.

Table 7: Optimizer Results

Optimizer	Accuracy (%)	AUC
SGD	97.36	0.72
SGD with Learning Rate Decay	97.47	0.72
Adam	97.79	0.78
RMSprop	97.79	0.79

Table 7 shows that smarter, adaptive learning rate algorithms can lead to significantly improved results. While the networks trained with stochastic gradient descent, both static and with learning rate decay, achieved AUC scores of 0.72, the models using the Adam and RMSprop optimizers achieved scores of 0.78 and 0.79 respectively. Turning to their training curves (Figures 29 - 34), Adam appears to have a slightly more reliable training process with less noise between training iterations.

5.7 Hardware

While not necessarily a model parameter, computer hardware is an important part of the training process. Because neural networks are slow to train compared to most other models, fitting large neural networks is often prohibitively slow on a personal computer. For this reason, I used the GPU-enabled devices in the Machine Learning Lab located in St. Mary's Hall. The NVIDIA Titan X GPU processors in these

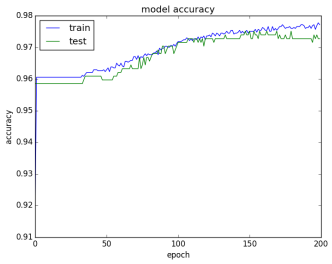

Figure 29: Accuracy with a Static Learning Rate (SGD)

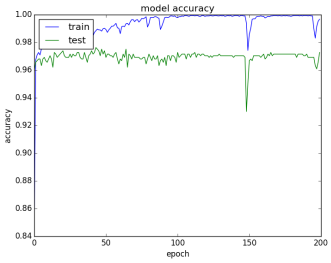

Figure 30: Accuracy with Adam Optimizer

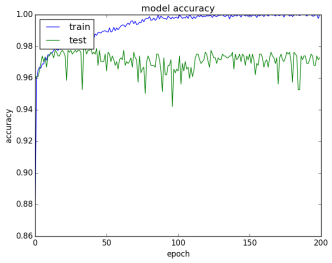

Figure 31: Accuracy with RMSprop Optimizer

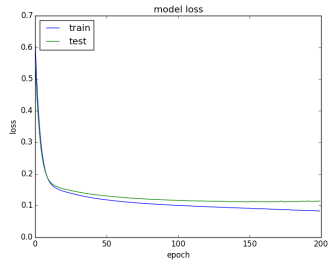

Figure 32: Loss with a Static Learning Rate (SGD)

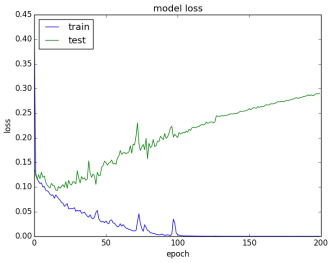

Figure 33: Loss with Adam Optimizer

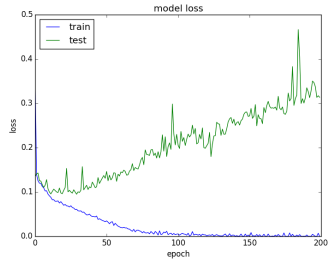

Figure 34: Loss with RMSprop Optimizer

devices allowed my models to train approximately 30 times faster than on my personal machine.

6 Model Selection

To select the best model, I take what has worked best in the prior sections of this paper and run a final parameter search. Here I test 144 different models, varying:

- Dropout percentage (20, 30, 40, or 50% of weights at each training iteration)
- Layer size (256, 512, or 1024 nodes in largest layer)
- Number of layers (3, 4, or 5 layers)
- Layer structure ($N \rightarrow N \rightarrow N$ layer structure vs. $N \rightarrow (N/2) \rightarrow (N/2)/2$ structure.)
- Activation function (ReLU vs. tanh)

Additional to these parameters being varied, each model uses batch normalization between hidden layers.

6.1 Results

Of the 144 models tested, the most successful was a model with five hidden layers, containing 1024, 512, 256, 128, and 64 nodes respectively. The model uses a 30% dropout rate and also uses batch normalization between each hidden layer. This model achieved a 98.51% accuracy score with an impressive AUC score of 0.83.

7 Discussion

The best model in this paper achieves an accuracy of 98.51% and an AUC score of 0.83. This is a considerable improvement to other prior results obtained on this data using similar neural models. Zieba, Tomczak, and Tomczak (2016), for example, see mean AUC scores between 0.54 and 0.70 for neural network models used on the same bankruptcy datasets used in this paper. The perceptrons used by Zieba, Tomczak, and Tomczak (2016) are simpler models, with only one hidden layer, so this paper in part shows the benefit of having a large, appropriately regularized model.

It is important to note, however, that non-neural models show even better performance on this data. Zieba, Tomczak, and Tomczak (2016) report AUC scores of up to 0.96 for gradient-boosted tree models, showing that such models are better suited to this particular task. The final takeaway of this project, then, is that while neural networks are not a cure-all solution to difficult classification tasks, they can still achieve strong out of sample performance if properly trained. By training large regularized models with normalization between layers, strong out of sample prediction with neural networks is indeed possible, even in highly imbalanced data sets.

References

[1] Zieba, Maciej, Sebastian K. Tomczak, and Jakub M. Tomczak *Ensemble boosted trees with synthetic features generation in application to bankruptcy prediction.* Expert Systems with Applications 58 (2016): 93-101. Web

[2] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville *Deep learning* Cambridge, MA: The MIT Press, 2017.

[3] Nielsen, Michael A. *Neural Networks and Deep Learning.* Determination Press, 2015.

[4] Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. *Multilayer feedforward networks are universal approximators.* Neural Networks 2.5 (1989): 359-66. Web.

[5] Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. *Dropout: A Simple Way to Prevent Neural Networks fromOverfitting.* Journal of Machine Learning Research 15 (2014). Web.

[6] Minsky, Marvin, Seymour Papert. *Perceptrons: An Introduction to Computational Geometry.* Cambridge, MA: The MIT Press, 1969.

[7] Raschka, Sebastian. *What is the Role of the Activation Function in a Neural Network?* KDnuggets Analytics Big Data Data Mining and Data Science. N.p., 2016. Web.

[8] Karpathy, Andrej. "Convolutional Neural Networks for Visual Recognition." CS231n Lecture Notes, 2016. Web.

[9] Lecun, Yann, Leon Bottou, Genevieve B. Orr, and Klaus -Robert Mller. *Efficient BackProp.* Lecture Notes in Computer Science Neural Networks: Tricks of the Trade (1998): 9-50. Web.

[10] Krizhevsky, Alex, Ilya Sutskever, Geoffrey Hinton. *ImageNet Classification with Deep Convolutional Neural Networks.* In NIPS, pp. 1106?1114, 2012

[11] Ruder, Sebastian. *An overview of gradient descent optimization algorithms.* Sebastian Ruder. Sebastian Ruder, 18 Apr. 2017. Web.

[12] Tape, Thomas. *Interpreting Diagnostic Tests.*— Likelihood Ratios. N.p., n.d. Web.

[13] Ioffe, Servey, Christian Szegedy. *Batch normalization: Accelerating deepnetwork training by reducing internal covariate shift.* InICML, 2015