Supervised Weather Forecasting with NN

Samuel Taliaferro
Virginia tech
Blacksburg, VA 24060
Samueltv250@vt.edu

Abstract

In this paper, I demonstrate the use of neural networks for predicting meteorological data and the fine-tuning of parameters to achieve optimal training results. I employed a feed-forward multi-layer neural network with Adam optimization to find the optimal weights that minimize the loss function. Adam optimization is an improved version of stochastic gradient descent that utilizes beta1 and beta2 values to control the decay of the moving average. Both forward and back propagation were utilized to update the gradients. By training multiple models with various learning rates and batch sizes and splitting the data into 75% for training and 25% for validation, I achieved an error rate of 16% for training and 13% for testing after fine-tuning.

I. INTRODUCTION

The main challenge in NWP is the highly nonlinear and complex nature of the atmosphere and ocean systems. Traditional methods, such as mathematical models, have limitations in accurately capturing these complex interactions. In recent years, the meteorology community has started exploring the use of deep learning models for NWP. These models have the ability to capture complex patterns in the data and provide more accurate predictions.

In this paper, I present a case study of using neural networks for predicting meteorological data. I focus on predicting whether or not it will rain in a given area based on the previous day's weather data indicators for that area. I employ a feed-forward multi-layer neural network with Adam optimization to find the optimal weights that minimize the loss function. I also demonstrate the importance of fine-tuning the parameters of the model, such as the learning rate and batch size, to achieve the best training results.

I show that by using my approach, I was able to achieve an error rate of 16% for training and 13% for testing. My results demonstrate the potential of using neural networks for NWP and the importance of parameter fine-tuning in achieving optimal performance.

II. THEORETICAL DESCRIPTION

The feed-forward neural network that I made employs the Adam optimization algorithm to optimize the weights of each node. Before the training begins, the training dataset is split into two-thirds for training and one-third for validation. During each iteration of training, the training data is randomly shuffled and separated into mini-batches.

The mini-batches are divided into equal batch sizes, which are a preset hyperparameter. Each batch is a set of samples drawn uniformly without replacement from the original training dataset. During every epoch, the training dataset is reshuffled and the batches are resampled from the training dataset.

Adam optimization is a combination of momentum gradient descent and root mean squared propagation. Gradient descent works by picking a random starting point for the weights and then calculating the gradient by making an initial prediction and calculating the error. Adam then uses momentum to accelerate the gradient descent towards the local minima. The first momentum is calculated by updating the previous first momentum with beta1, updating the gradient with 1-beta1, and adding these two together. The second momentum is calculated by updating the previous second momentum with beta2, squaring the gradient, and updating it with 1-beta2, and adding these two together.

When we take both equations from both optimizers and put them together, we can generate the Adam optimizer. This is represented in the following equation, where m represents the first momentum and v represents the second momentum:

$$m = beta1m + (1-beta1)grad$$
$$v = beta2v + (1-beta2)(grad)**2$$

Initially, v and m are both set to 0. The next step in this algorithm is calculating the bias-corrected first and second momentum. This is represented in the following equation, where mhat represents the first bias-corrected momentum, vhat represents the second bias-corrected momentum, and iter represents the iteration:

$$mhat = m/(1-(beta1iter))$$
$$vhat = v/(1-(beta2iter))$$

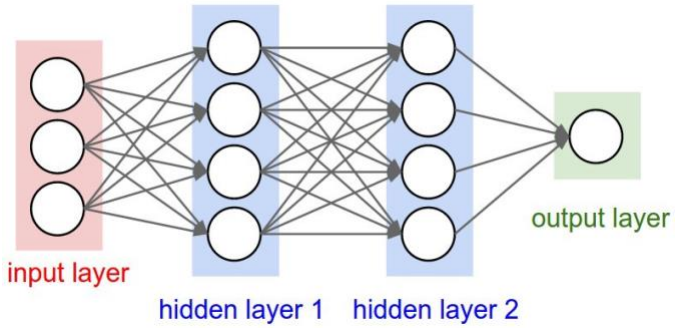
The next step in this algorithm updates the weights of the model. This is represented in the following equation, where mhat represents the first bias-corrected momentum, vhat represents the second bias-corrected momentum, e represents epsilon, alpha represents the learning rate, and w represents the weights:

$$w = w - alpha *(mhat / \sqrt{(vhat)} + e)$$

The model has four main parameters: alpha (the learning rate), beta1, beta2, and epsilon. The epsilon parameter is used to prevent division by zero, the alpha parameter determines the proportion that the weights are updated during each iteration, the beta1 parameter is the exponential decay for the first moment estimates, and the beta2 parameter is the exponential decay for the second moment estimate.

The hyper parameters I tuned during training where the batch size, the number of hidden nodes, the learning rate, and the number of epochs. If the learning rate is too big then the model will not converge but if the learning rate is too small then the model will take forever to converge, tuning the learning rate was a crucial part to making the model work properly. The model will stop training when the maximum epochs are reached or when the update magnitude is below a specified threshold, this is when the model converges and stops learning because the weights stop changing.

The actual neural network works by adding the previous neurons scaled by the weights of the connections and passing them through a activation function to calculate the output of the node. This is done for every layer until the output layer is reached, then the output nodes get added up to calculate the final target. My first layer has a sigmoid activation function while the second layer has a linear activation function.



The scaler used for this projects data is the standard scaler, this scaler produces sets that have a mean of zero and identity covariance. The scaler is uses to improve training and prevent extremely large numbers that could cause overflow or underflow in the calculations.

III. Implementation

The first step in implementing my model was to generate some functions that could help me evaluate its performance. Since the problem I was solving is a binary classification problem, generating a confusion matrix and a classification report containing the accuracy and error rate of each class seemed like a good metric to use. I generated a function called accuracy_score(Y, pred) that calculates the accuracy, error_rate(Y, pred) that calculates the error rate, confusion_matrix(Y, pred) that generates the binary confusion matrix for the prediction, and classification_report(Y, pred) that generates a report showing the accuracy and error rate for each class.

The function draw(N) generates a set of N random inputs x with their corresponding sine wave outputs y. I used the draw(N) function to generate my own training and validation set to determine if my neural network was correctly fitting to the training dataset. I also created a scaler() class that can be used to standardize or normalize input data.

The getAndCleanData() function opens the 'weatherAUS.csv' file located in the scripts directory and cleans the data by generating dummy variables for the target column "RainTomorrow" and extracting the numeric features ['Rainfall', 'MinTemp', 'MaxTemp', 'WindGustSpeed', 'Temp9am', 'WindSpeed9am', 'Humidity9am', 'Pressure9am', 'Temp3pm', 'WindSpeed3pm', 'Humidity3pm', 'Pressure3pm', 'RainToday']. These features are used as inputs to predict the target. The rows containing NaN are dropped and the inputs x and targets y are returned.

The main class for my model is called adamNN(self, hNodes, batchsz = 32, alpha= 1/1000) and it contains everything related to training, testing, and making predictions with a two-layer feed-forward neural network. The class is initialized with the number of hidden nodes, the batch size, and the learning rate. If the batch size and learning rate are not provided, the batch size will be set to 32 and the learning rate to 1/1000.

The model is trained by calling the train(self, input, target, maxEpochs = INFINITY) function that takes the input and target dataset and the maximum number of epochs wanted to train the model. If no argument is given for maxEpochs, the model will continue iterating through epochs until it converges. In lines 223-230, the training begins by declaring the beta1, beta2, and epsilon parameters. Then, in lines 250-260, the training dataset is randomly shuffled and divided into 1/3 for validation and 2/3 for training. In lines 280-283, the weights are initialized to a set of randomly distributed numbers with sigma = 5.

In line 300, the main training loop starts. In each epoch, the dataset is randomly shuffled and then the algorithm starts iterating through each batch to perform gradient descent. In line 329, I start making my prediction using the weights for each layer.

Each hidden node in the hidden layer is calculated by adding every input node with its corresponding layer 1 weight. The hidden nodes are passed through a sigmoid function to calculate the outputs to the hidden layer. In line 337, I start calculating the output node by adding all of the output nodes from the hidden layer and multiplying each output node by its corresponding layer 2 weight. The output node is passed through a linear function to produce the final output. In line 344, the difference from the prediction and the actual output is calculated to compute the gradient of each layer by backpropagation of delta.

In line 364, the Adam parameters are updated for both layers, and in line 379, the weights for both layers are updated using the Adam parameters. Once both weights are calculated, the error and accuracy are saved for the current weights, and I calculate the difference between the previous weights and the new weights to check if the weights have converged. If the update to the weights is 0, then the model has stopped learning, which means it has converged. If the model converges or the maximum number of epochs is reached, then I stop training, save, and return the accuracy history and weights for each layer.

In line 565, there is the trainAndScore() function. This function trains the model with the data extracted with the getAndCleanData() function. The data is split into 75% for training and 25% for testing. Once the model is trained, this function generates a classification report for the training and testing datasets to score the accuracy of the model. In line 624, the weights and scales used for the model are saved into a binary pickle file called "ModelAndScale.pickle". This function trains the neural network with the optimal parameters that were found during testing and fine-tuning.

IV. Reproducibility

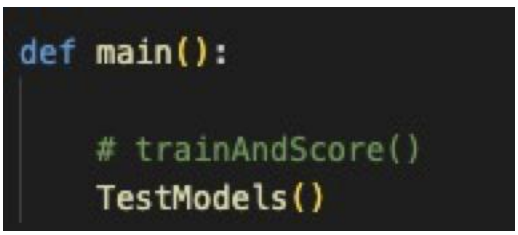
The model, code, and data used to train this model can be found in the following GitHub repository:

https://github.com/samueltv250/Adam-Neural-Network.git

In this repository, you will find the pre-trained model "ModelAndScale.pickle", the dataset containing 145,460 Australian weather observation "weatherAUS.csv", and the code for training and testing the model in the script file "NN_Adam.py". To run the "NN_Adam.py" script, you will need to have the following libraries installed on your computer: pickle, pandas, numpy, json, random, math, matplotlib, seaborn, tracemalloc, and time.

To view my final results you need to run the provided “NN_Adam.py” without doing any changes to the script with the pre-trained model called “ModelAndScale.pickle” and the dataset “weatherAUS.csv” in the same directory of the script, this will split the data in the same way that it was split during training, the script will then load the pre-trained model and create a classification report for the testing and training datasets, this will also plot the historic error and accuracy that the model had during training.

To reproduce my results by retraining the model, you need to uncomment the trainAndScore() function in the script.

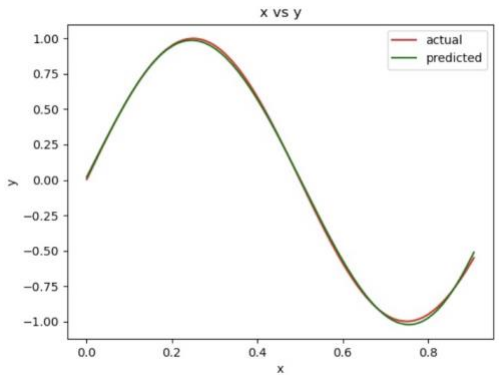


Doing this will train, score, and save the model with 75% of the data for training, 2 hidden nodes, batch size of 16, learning rate of 1/10000, and 90 epochs. The model will be saved with the name "ModelAndScale.pickle" in the same directory where the script is located. Beware that if you train a new model and the pre-trained model of the same name is also in the same directory, the old pre-trained model will be replaced by the new model.

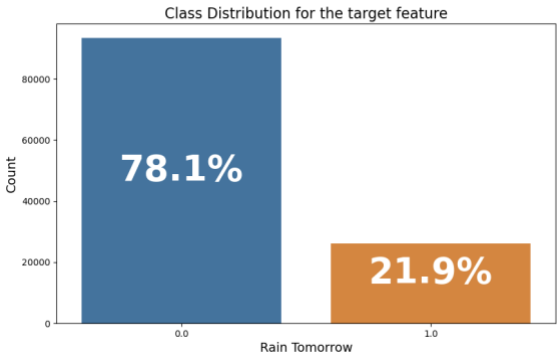
V. Results

To test the network, I generated a dataset of random inputs (x) and sinusoidal outputs (y) using the draw() function. The sigmoid activation function was implemented, and the Adam parameters were set to the recommended values of beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8, and alpha = 1/1000. A batch size of 256 was used, and the maximum number of epochs was not specified.

To determine the optimal number of hidden nodes, I tested the network using 1-10 hidden nodes and trained the model on the sinusoidal data until convergence was reached for each number of hidden nodes. The best results were obtained using 5 hidden nodes, which allowed for a very good fit for both the training and testing datasets



After successfully fitting the sigmoid function to the model, I began the data cleaning process. The goal of the project was to predict whether it would rain in a given location in Australia based on previous weather data, so I generated dummy variables for all of the yes/no columns in the dataset and dropped any non-numeric columns. The data was then divided into inputs and targets. The target was a binary column (1 for rain, 0 for no rain), and the inputs were the remaining numeric columns from the dataset. After cleaning the data, the binary distribution was as follows:



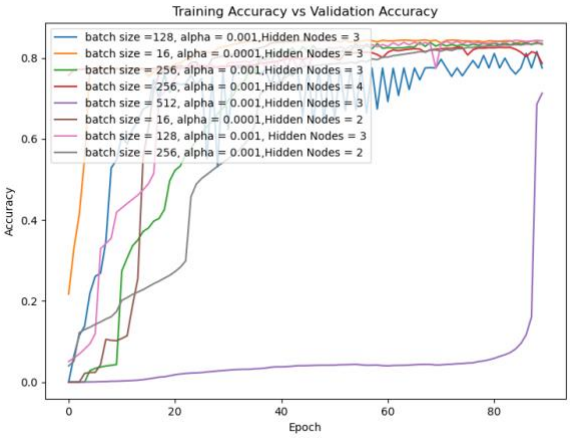
After verifying that there was no class imbalance in the dataset, I ran the model using the same parameters as before to train the sigmoid dataset. However, this resulted in very long training times because the model was not converging quickly enough. I managed to achieve an accuracy of 83% for training and 85% for testing, but the training time was more than a day.

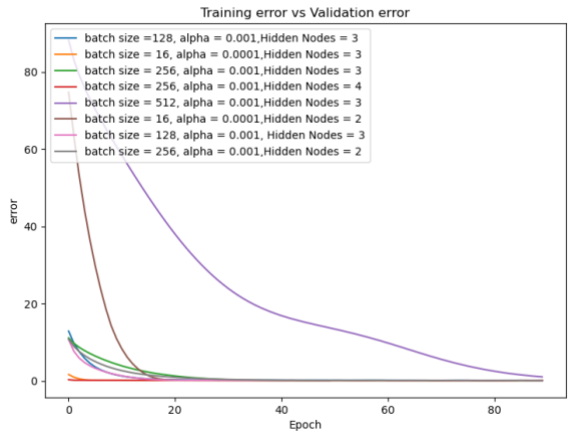
To improve the model's performance, I fine-tuned the parameters by setting a maximum number of epochs to 450 and training the model using 1-10 hidden nodes to determine the optimal number of hidden nodes for this data.

```
Training error rate = 0.19358533791523483
Testing error rate = 0.2268041237113402

Running model with 2 hidden nodes
Training error rate = 0.17048346055979643
Testing error rate = 0.12643678160919541

Running model with 3 hidden nodes
Training error rate = 0.1753889674681754
Testing error rate = 0.189873417721519

Running model with 4 hidden nodes
Training error rate = 0.16666666666666666
Testing error rate = 0.19718309859154928
```

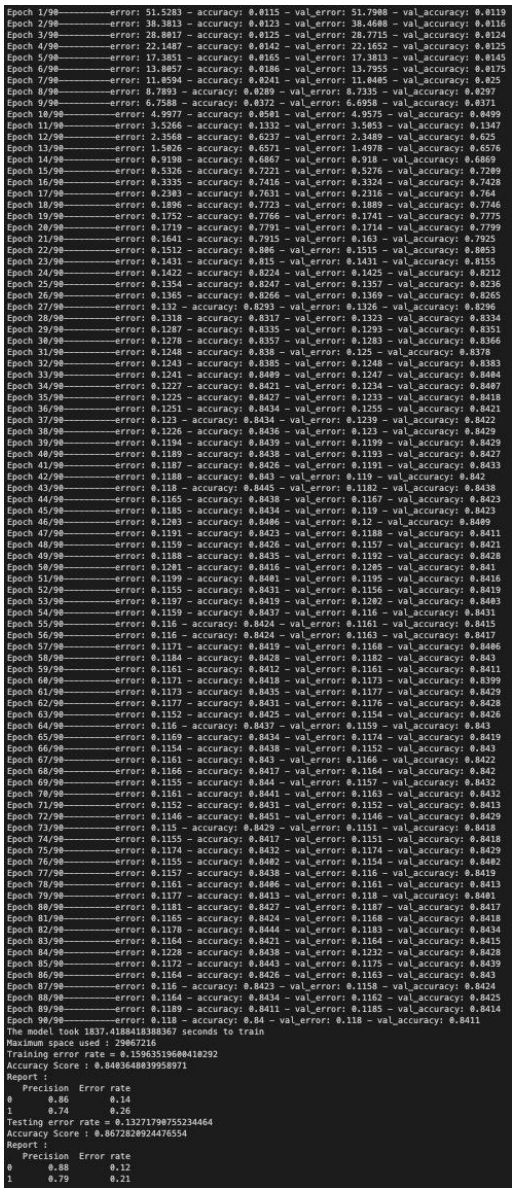
I found that using 2 hidden nodes gave the best accuracy for both classes, so I set the number of hidden nodes to 2. To reduce the training time, I made the batch size smaller and found that a batch size of 16 yielded the best results. I also experimented with the learning rate by increasing and decreasing it slightly to see how it affected the model's performance. As expected, the model trained slower with a smaller learning rate, and did not reach the minimum error when the learning rate was too high. I ultimately settled on a learning rate of 1/10000, which achieved the minimum error and reduced the training time.

Below is a plot demonstrating the accuracy and error I obtained during part of the fine-tuning using the standard scaler:



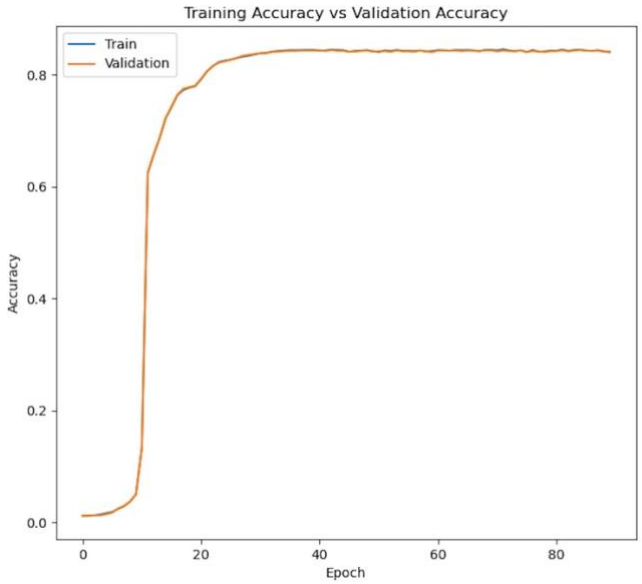


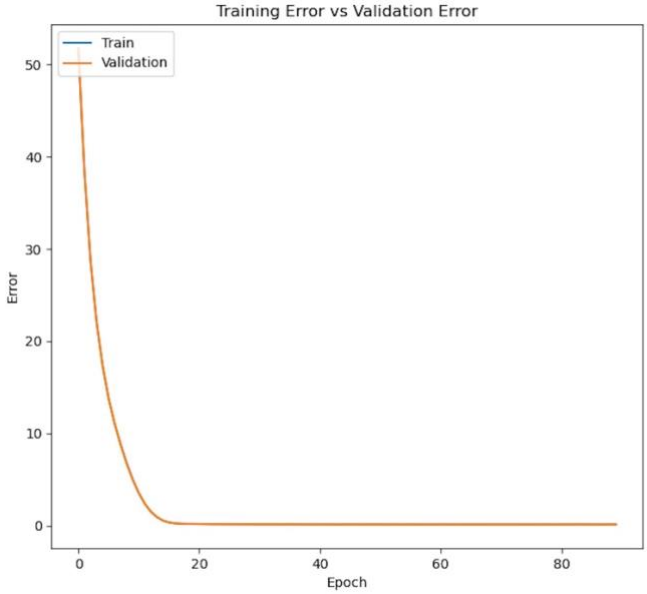
After testing multiple variations of the parameters and fine-tuning, I found that setting alpha = 1/10000, using 2 hidden nodes, and having a batch size of 16 gave the fastest and most accurate results with the dataset. The following is the output I obtained during training:



Using the chosen parameters, I was able to achieve an error rate of 16% for training and 13% for testing. This is a very good result, as the accuracy is not biased towards one class, meaning that the model is accurate at predicting whether it will rain or not tomorrow. The model trained in just under 30 minutes and used a maximum of 29 MB of memory.

The following is the history for the training and validation accuracy and error for my final, optimal model:





During the fine-tuning process, I trained over 20 models with different parameters, scales, and activation functions. I was able to obtain a good number of models that were very accurate, but the most challenging part of the project was finding a set of parameters that trained the model in a reasonable amount of time while still reaching the minimum error. I believe I found one of the optimal sets of parameters, as the model consistently reduced the error and reached a very accurate result in only 90 epochs.

VI. References

[1] Wyatt, C. (n.d.). *Lecture 26 Demo Code*. Lecture.

[2] *Adam*. Adam - Cornell University Computational Optimization Open Textbook - Optimization Wiki. (n.d.). Retrieved December 1, 2022, from https://optimization.cbe.cornell.edu/index.php?title=Adam

[3] Brownlee, J. (2021, January 12). *Gentle introduction to the adam optimization algorithm for deep learning*. MachineLearningMastery.com. Retrieved December 1, 2022, from https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[4] sahilambekar777. (2021, January 19). *Neural net pytorch (torch.nn) indepth for Beginner*. Kaggle. Retrieved December 1, 2022, from https://www.kaggle.com/code/sahilambekar777/neural-net-pytorch-torch-nn-indepth-for-beginner

[5] *Sklearn.preprocessing.StandardScaler*. scikit. (n.d.). Retrieved December 1, 2022, from https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

[6] Young, J. (2020, December 11). *Rain in Australia*. Kaggle. Retrieved December 1, 2022, from https://www.kaggle.com/datasets/jsphyg/weather-dataset-rattle-package?datasetId=6012