Authors: Brandon Manville and Cameron Bronson
Date: December 8, 2023

An Exploration of Neural Networks Using PyTorch

Abstract

In this report, we delve into Neural Networks, their implementation using PyTorch in Python, and their comparative effectiveness across various datasets. Neural Networks, while widely applicable in fields like image processing, natural language processing, and financial predictions, pose a challenge as a black-box classification technique. Our objective was to assess Neural Networks against traditional classification methods, utilizing Kaggle datasets along with generated data for evaluation.

Our study reveals that Neural Networks often almost matched, matched, or outperformed the best traditional models across datasets. However, this came at the cost of significant computational intensity, with Neural Networks taking over six hours to execute compared to traditional methods, which completed within thirty minutes.

While acknowledging the commendable performance of properly trained and tuned Neural Networks on real-world datasets, our findings caution against blindly using them. Their computational demands, hyperparameter optimization challenges, and difficulty interpreting results highlight the need for a more balanced approach. Our subsequent sections detail the methodology behind constructing Neural Networks (Materials and Methods), present dataset-specific results (Results), and delve into the implications of our findings (Discussion).

The report not only underscores the promising performance of Neural Networks but also raises important considerations regarding their practicality and trade-offs in application.

Introduction

Neural Networks are a type of machine learning that attempts to build a model that is "inspired by the human brain, mimicking the way that biological neurons signal to one another" (1, IBM). Brains are made up of billions of neurons that carry information. Each neuron fires information when it is provided with a stimulus; this carries the information down the chain of neurons. Neural networks behave similarly. The input data is stored in 'neurons' where initially we store each feature in a neuron. We then perform a mapping of the data to another wall of neurons that can be smaller or larger than the original input vector. We continue this process of mapping neurons to neurons until we reach the last layer. We call this last layer the output layer, and we call the middle layers the hidden layers. As the data leaves the input layer and traverses the hidden layers, it will reach the output layer transformed into useful information.

The specifics of how a Neural Network operates is by applying linear transformations to the data and then applying activation functions that perform "non-linear transformation[s] to the input making it capable to learn and perform more complex tasks" (2, Tiwari, S.). Neural networks can have many layers of neurons or a few layers depending on the level of complexity of the data. As our data did not have very complex relationships within the features, our Neural Networks were all one to three layers. In this report by the way, when we say a model has two layers, we mean the model has one input layer, two hidden layers, and one output layer. This is a relatively standard amount of layers (3, Heaton, J.). Many different activation functions can be used depending on the task. We will talk about these more later. The last layer of the Neural Network is the output; the outputs are used to make predictions. In our code, the output layers always consisted of probabilities. The neuron with the highest probability would determine the class the data belonged to.

Our project's goal was to explore and implement Neural Networks in classification. Specifically, we were focused on how Neural Networks perform when compared with other classification methods we have discussed in the course so far. The methods we compared Neural Networks to were LDA, QDA, Naive Bayes, and Random Forests. We chose to use LDA, QDA, and Naive Bayes as they all perform well for their respective use cases (i.e. Naive Bayes performs well if the features are independent and identically distributed like Gaussian Distributions). We chose Random Forests as it generally performs well in many scenarios. We predicted, given enough computational resources and data, Neural Networks would outperform these other methods in the majority of scenarios. We will talk extensively about the outcomes of the experiment in the results section and the pros and cons of Neural Networks when compared to other methods in the discussion sections.

Materials and Methods

Setup

In our exploration of Neural Networks, we used PyTorch to create the Neural Networks and Scikit-learn to implement the other methods. PyTorch is a package in Python that has methods to build, train, optimize, and test Neural Networks. There is another package that is commonly used to create Neural Networks, TensorFlow, but PyTorch is generally more beginner-friendly. We specifically used PyTorch version 2.1.1 with CUDA 1.8 enabled to utilize GPU acceleration to decrease runtime. Scikit-learn is a general data science package that we utilized to perform the LDA, QDA, Naive Bayes, and Random Forest regressions on the datasets. We used scikit-learn version 1.3 in our project.

Datasets

The datasets we chose to use for this project were the Drinking/Smoking Dataset (1), the Diabetes Dataset (2), and the Powerlifting Dataset (3) (links can be found below). We used 'smoking_driking_dataset_Ver01.csv', 'diabetes_binary_health_indicators_BRFSS2015.csv', and 'openpowerlifting.csv' for our predictions. The Smoking/Drinking dataset contains various health markers, some binary and some continuous, and has binary variables on whether an individual drinks or smokes. Smoking originally was not a binary variable (but instead a trinary), so we turned it into binary. It has 990,000 samples. As the CSV name suggests, the Diabetes dataset consists of around 253,000 samples of binary health variables that are used to predict whether a person has diabetes or not. Finally, the Powerlifting dataset has around 1.4 million samples of people competing in various divisions, age classes, weight classes, etc. There were originally over 100 different types of placings, but we decided to just predict whether someone would win 1st, 2nd, 3rd, or not place.

In the datasets, we removed any data entries that were missing any key features. This was only noticeable on the powerlifting dataset as it had many incomplete entries, but the size of the dataset was so large that we were able to remove a large portion of the set whilst still having enough data to train and test a Neural Network on it. After this cleanup, the powerlifting dataset was left with around 400,000 samples. We also removed many features from the powerlifting dataset. For example, it had features "Deadlift1Kg, Deadlift2Kg, Deadlift3Kg, Deadlift4Kg, Best3DeadliftKg" which correspond to all the deadlifts done by an individual at a meet. But, the only deadlift that matters is "Best3DeadliftKg" since this is what is used in placings. Hence, the other deadlift variables can be deleted. Please refer to the PowerliftingCleanUp.py code to see all the cleanup we did for this dataset. As we will see later, this dataset was very hard to predict, so we also made some simulated datasets using Scikit-learn. These were made to further demonstrate the flexibility of Neural Networks.

Defining Datasets and Neural Networks

Our implementation of Neural Networks is based on the model from the PyTorch website (5, Welcome to PyTorch Tutorials). We used the Pandas package to read the CSVs, but PyTorch requires our data not to be stored as Pandas DataFrames but stored as Tensors Datasets that implement PyTorch Datasets. Luckily, torch.tensor() is an easy method that can convert NumPy arrays into tensors. Using this, we made a class of custom datasets that takes in NumPy input vectors and the corresponding labels as parameters. With our custom datasets built, we now need to define a Neural Network. To do this, there are two major components: the constructor and the forward function. The constructor simply calls a PyTorch superclass where the actual work is done. We also defined the constructor to take in a sequence; this is the Neural Network and the parts we call the layers and neurons. Even though at a high level we think of it like this, the sequence is just a sequence of linear transformations and activation functions. The forward function calls the sequence and sends the input data through the sequence. It takes the data from one layer of neurons to the next. Please refer to Figure 1 for how to define the datasets and Neural Networks and Figure 2 for an example of a sequence we would pass into the constructor. Figure 3 shows the overarching structure of a Neural Network. Figure 4 captures the essence of what happens at each neuron.

```python
#Builds a Dataset
class CustomDataset(Dataset):
    def __init__(self, x, y):
        super(CustomDataset, self).__init__()
        self.x = torch.tensor(x.values, dtype=torch.float32)
        self.y = torch.tensor(y, dtype=torch.float32)

    def __len__(self):
        return len(self.x)

    def __getitem__(self, index):
        return self.x[index], self.y[index]

#Defines a NeuralNetwork
class NeuralNetwork(nn.Module):
    #seq of layers
    def __init__(self, seq):
        super().__init__()
        self.seq = seq

    def forward(self, x):
        logits = self.seq(x)
        return logits
```

Figure 1

```
PS C:\Users\camro\Downloads> & C:/Users/camro/AppData/Local/Programs/Pytho
utation.py
Using seq=Sequential(
  (0): Linear(in_features=21, out_features=15, bias=True)
  (1): Linear(in_features=15, out_features=30, bias=True)
  (2): ReLU()
  (3): Linear(in_features=30, out_features=1, bias=True)
  (4): Sigmoid()
) and lr=1, we get 0.8603665159956551 accuracy using Neural Networks.
PS C:\Users\camro\Downloads>
```

Figure 2

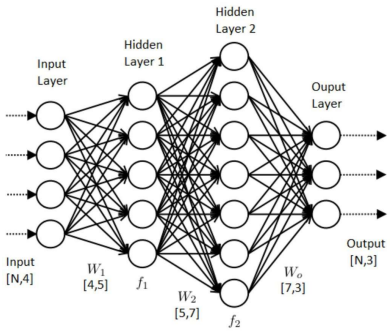


Figure 3

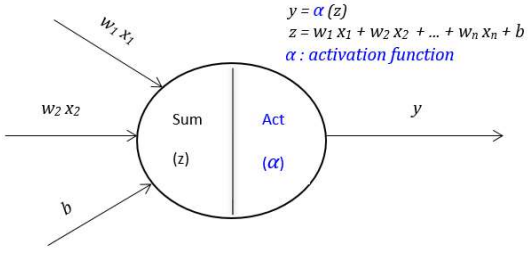


Figure 4

Making/Training a Neural Network

Next, we need to make and train the model. The model consists of an optimizer and a loss function. The loss function is the measure of the 'badness' of a model. We often wish to minimize the loss function since a higher loss function means we are predicting wrong. We chose two loss functions: Cross-Entropy Loss and Binary Cross-Entropy Loss (BCE Loss; a special case of Cross-Entropy Loss). Cross-Entropy Loss is used for multiclass classification problems while Binary Cross-Entropy Loss is used for binary classification problems (4, 365 Data Science). When dealing with multiclass classification, we need to use Softmax as our last activation function in the output layer. This will convert each entry in the output layer to a probability between (0,1). We make sure the output layer has as many neurons as there are classes. We then select the index with the highest probability and this index is the class we classify the input data to. Similarly, we use Sigmoid on the output of a binary classification task to turn the values into a number between (0,1). Our output layer will consist of only one neuron. If this neuron has a probability that is higher than 0.5, we classify it to class 1; else, it is classified to class 0 (class 1 is yes in our implementation and class 0 is no).

The optimizer's job is to try to minimize the loss function. Whenever we feed data through the model, the optimizer will look at the loss function and try to make changes to the model based on how poorly it performed on the data. These changes will hopefully make it so that the next time the data is fed through, there is less loss. These changes are made in terms of updating the 'weights' of the model. That is, updating the linear transformation used to transform the data. The optimizer we chose to use is the Adam optimizer (6, Adam). There are different hyperparameters to choose for this portion of the code.

Specifically, we need to choose the number of epochs, the learning rate, and the batch size. The batch size is how much data the model considers at one time. We kept the batch size at 64 for our model, so the model would run 64 samples through, evaluate the loss, and the optimizer would then backpropagate through the model and make changes to the linear transformations to minimize the loss. Then, it takes another 64 samples and does it again until the whole dataset is iterated through. The number of epochs is how many times we wish to go through the dataset. Going through the dataset too many times can lead to overfitting but too little can lead to underfitting. Finally, the learning rate is how big of a change we make at each step. "If your learning rate is set too low, training will progress very slowly as you are making very tiny updates to the weights in your network. However, if your learning rate is set too high, it can cause undesirable divergent behavior in your loss function" (7, Jordan, J.). These are all hyperparameters that need to be tuned to the data to increase the model accuracy. Figure 5 is making and training the model while Figure 6 is testing the model. Note that it is recommended to write model.eval() before testing the model. This does not matter for most Neural Networks, but sometimes it is necessary to take the model out of training mode and put it into evaluation mode.

```python
model = NeuralNetwork(best_seq).to(device)
loss_fn = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=best_lr)
num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    for batch in train_dataloader:
        inputs, targets = batch
        inputs, targets = inputs.to(device), targets.to(device)

        optimizer.zero_grad()
        outputs = torch.flatten(model(inputs))
        loss = loss_fn(outputs, targets)
        loss.backward()
        optimizer.step()
```

Figure 5

```python
with torch.no_grad():
    for batch in test_dataloader:
        inputs, targets = batch
        targets = torch.flatten(targets)
        inputs, targets = inputs.to(device), targets.to(device)

        outputs = model(inputs)
        outputs = torch.flatten(outputs)
        predicted = (outputs > 0.5).float()
        total += targets.size(dim=0)
        correct += (predicted == targets).sum().item()

accuracy = correct / total
```

Figure 6

Optimizing Neural Networks

Finally, for Neural Networks, we will talk about ways to tune the hyperparameters. First note, in our models, we kept the optimizer and batch sizes the same. This is not the best practice, but this was a rudimentary exploration of Neural Networks. Hence, we did not try to optimize every single feature. Plus, it would take significantly more computational power, which is a limiting factor we will see later in the discussion section. We did attempt to optimize the number of epochs, the learning rate, the number of layers, the number of neurons at each layer, and the activation functions. We took 1/10 of the data to train on and 1/40 of the data to test the accuracy when finding the hyperparameters. We then evaluated each model and saved the best hyperparameters according to the accuracy score which we deemed as the total number of correctly labeled samples divided by the total number of samples. With these best hyperparameters, we created a new model with all the data except for a portion of it which we used to test the model.

At first, we used the guess and check method for all four hyperparameters. That is, we picked an arbitrary value and made a model from it. We later realized we could optimize this by dynamically stopping the number of epochs once the model stopped improving or was improving at an extremely slow rate. Our specific implementation allowed for a max of 500 epochs, but it stopped earlier if there were 10 consecutive iterations of less than 0.005% accuracy improvement; it remembered the number of epochs with the highest accuracy. This allows for the model to train as long as it needs to without overtraining. We only used this for the generated datasets and the powerlifting dataset; we did not go back and rerun the other datasets with this optimization because of computation reasons. There is a similar way to dynamically optimize the learning rate, but we did not implement it. For the number of hidden layers and neurons, we (loosely) tried to follow the basic rules laid out in the “The Number of Hidden Layers” article (3, Heaton, J.). The input layer will have the size of the number of features (one neuron for each feature); the output layer will have the size of the number of classes we wish to classify the data to. Lastly, we tried randomly stacking different activation functions with these layers to see the

best results. For different activation functions and their equations, please see “Activation Functions in Neural Networks” (8, V7).

The Other Non-Neural Network Methods

For the other methods, we used the Scikit-learn (sklearn) package. For each, we used 5-fold cross-validation. We did not have this privilege for Neural Networks because of the computational power needed, but these methods were much simpler to make and run. We also tried to tune the Random Forests model based on a subset of the data. We tried to find the best number of estimators, maximum number of features, maximum depth, and the number of leaf nodes based on a subset of the data. We used around 100,000 samples for each of the real-world datasets to do this. We then built a full model based on all the data with the best hyperparameters and used 5-fold cross-validation.

Results

Smoking/Drinking

The first dataset that we tested was the Smoking/Drinking dataset. Since this was the first dataset, we tried very basic optimizations. For this dataset, we tried learning rates 0.01, 0.005, and 0.001, different amounts of stacking linear transformations with ReLU activation functions, and the number of neurons from 15 to 30 (where each layer had the same number). The goal of this dataset was to predict if someone was a smoker, a drinker, or either one based on some health information. Note that we left the number of epochs as 10 and this dataset had 22 features.

For the Smoking/Drinking dataset, the best learning rate and number of layers combination was a learning rate of 0.01 with three layers with either 22 or 23 neurons at each hidden layer. This setup yielded a smoking-alone prediction accuracy of 80.3%, a drinking-alone prediction accuracy of 72.5%, and a smoking or drinking accuracy of 75.6%. The best results for the other methods were Random Forests at 80.2% accuracy for smoking alone, LDA at 71.7% accuracy for drinking alone, and Random Forests at 75.2% accuracy for smoking or drinking. Even without complete optimization, Neural Networks outperformed the other classification methods in all categories. As we can see by the performance of LDA, this set is relatively linear. This shows Neural Networks can perform well on linear datasets. This is because Neural Networks are constructed from linear transformations of the features, allowing for the Neural Network to capture linear data.

Diabetes

The second dataset that we tested was the Diabetes dataset. The goal was to predict whether someone had diabetes based on some health measurements. Once again, we compared the results of Neural Networks with different learning rates, and different numbers of layers of neurons except this time with many different activation functions and different numbers of neurons. Note that we left the number of epochs as 10 and the dataset had 21 features.

We found the highest accuracy to be 86.0% with a learning rate of 1, a two-layer Neural Network with no activation on the first layer and ReLU on the second, and 15 neurons on the first with 30 neurons on the second. For the other methods, our accuracy for LDA was 86.1%, which was slightly higher accuracy than Neural Network, our accuracy for Naive Bayes was 77.3%, our accuracy for QDA was 77.0%, and our accuracy for Random Forests was 86.5%, which is also higher than our Neural Network accuracy. Even though Neural Networks did not have the highest accuracy here, it only lost by a few tenths of a percent. With proper parameter tuning, it would most likely beat out both LDA and Random Forests

Powerlifting

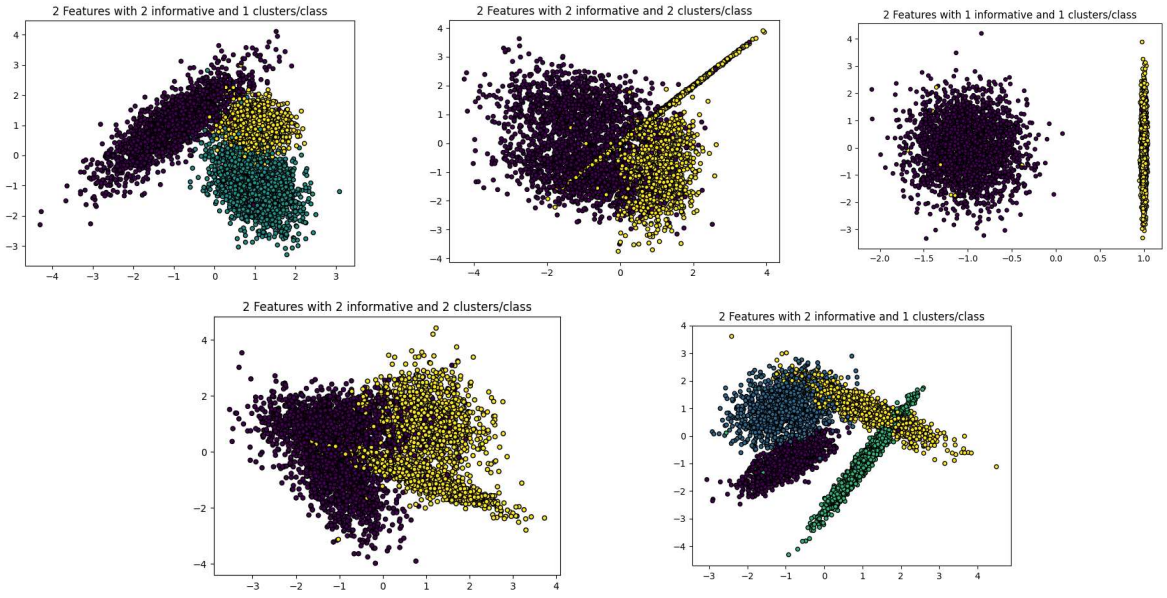
The third real-world dataset we tested was the Powerlifting dataset. This dataset had 11 features. After doing exploratory work on the first two datasets, our goal with this dataset was to use our knowledge to simply get the highest accuracy prediction from the Neural Network. We even tried changing the number of epochs from 5 to 10 to 15. This dataset was particularly

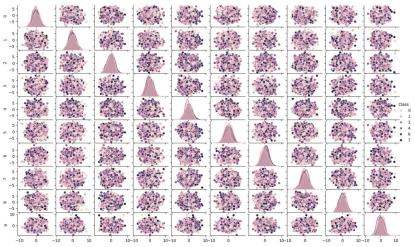
challenging as it is unpredictable (under 50% accuracy) using LDA, Naive Bayes, and Random Forests, so we had high hopes for Neural Networks. We were soon let down; Neural Network accuracy was also low coming in at 39.6%. This was done using a two-layer Neural Network, with a learning rate of 0.1, using SeLU activation function, and 12 neurons in the first layer and 6 neurons in the second. When trying to perform QDA on this dataset, we ran into issues that we could not resolve, so we only tested using LDA, Naive Bayes, and Random Forests. We achieved an accuracy of 43.8% on LDA, an accuracy of 42.5% on Naive Bayes, and 47.0% on Random Forests. All three of these methods outperformed our Neural Network, but they were also all sub-50 % accuracy. After the poor performance, we went back and tried the number of epochs optimization. It increased the accuracy to 40.1% which is an improvement but not by much.

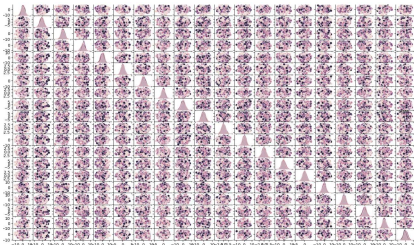
We will discuss these results further here instead of in the discussion section. We would anticipate that with more parameter tuning, the Neural Network would possibly outperform the other methods, but we have neither the time nor the computational power to properly tune a Neural Network. For these results alone, it took the Neural Network around 6 hours to run. Even with these optimizations, Neural Networks just might not perform that much better because of the dataset. It was incredibly difficult to predict as it had no way to denote the size of the competition. As one may expect, the amount of strength one needs to win a four-person event is much lower than the amount of strength needed to win a hundred-person event, but in our dataset, there was no way to recognize the number of competitors in each event. Additionally, there is no way to tell if a meet was a local, state, or national meet. Just as in any other sport, the talent it takes to win a local meet is much lower than to win a national meet.

Generated Datasets

As the powerlifting dataset was so unpredictable, we decided to create some datasets that we knew were predictable. We used Scikit-learn to make the datasets. The first datasets were simple, but the last two datasets had more complexity. We kept the number of epochs at 10 for the first five datasets and used the epoch optimization for the last two. The first five datasets had 5000 samples each and the sixth and seventh datasets had 100,000 and 200,000 samples, respectively.







Our specific results for each dataset are listed below. We will give a quick summary of this section right here for the readers who do not wish to dive into the details. Neural Networks demonstrated that it was able to handle linear and non-linear datasets very well. The first five datasets were linearly separable (as the images suggest), but the last two were not. Still, Neural Networks performed well though QDA did beat it out on the last two. Overall, Neural Networks and Random Forests had the best consistent results. The last takeaway from this section is how well the optimization for the number of epochs did. On the sixth dataset, it alone took the accuracy from 25.58% to 40.6%, and on the seventh dataset, it took the accuracy from 45.28% to 56.3%. These are both remarkable improvements.

Our first dataset has 3 classes with 2 features (2 informative) and 1 cluster per class. On this dataset, LDA's accuracy was 92.8%, Naive Bayes' accuracy was 95.4%, QDA's accuracy was 96.5%, and Random Forests' accuracy was 96.3%. We found the highest accuracy for Neural Networks at 96.1% with two layers with 8 neurons each, a learning rate of 0.01, and having only ReLU activation functions.

Our second dataset has 2 classes with 2 features (2 informative) and 2 clusters per class. On this dataset, LDA's accuracy was 86.7%, Naive Bayes' accuracy was 86.9%, QDA's accuracy was 86.2%, and Random Forests' accuracy was 90.8%. We found the highest accuracy for Neural Networks at 90.9% with two layers of 12 neurons and 6 neurons, respectively, a learning rate of 0.001, and having only ReLU activation functions.

Our third dataset has 2 classes with 2 features (1 informative) and 1 cluster per class. On this dataset, LDA's accuracy was 99.4%, Naive Bayes' accuracy was 99.4%, QDA's accuracy was 99.4%, and Random Forests' accuracy was 99.4%. We found the highest accuracy for Neural Networks at 99.2% with two layers of 8 and 4 neurons, respectively, a learning rate of 0.1, and having only SeLU activation functions. Note that this dataset is extremely separable and higher accuracy is probably not possible.

Our fourth dataset has 2 classes with 2 features (2 informative) and 2 clusters per class. On this dataset, LDA's accuracy was 85.9%, Naive Bayes' accuracy was 88.3%, QDA's accuracy was 92.3%, and Random Forests's accuracy was 92.9%. We found the highest accuracy for Neural Networks at 90.6% with one layer of 8 neurons, a learning rate of 0.001, and having only ReLU activation functions.

Our fifth dataset has 4 classes with 2 features (2 informative) and 1 cluster per class. On this dataset, LDA's accuracy was 90.2%, Naive Bayes' accuracy was 90.6%, QDA's accuracy was 90.2%, and Random Forests' accuracy was 92.8%. We found the highest accuracy for

Neural Networks at 92.6% with two layers both at 4 neurons, a learning rate of 0.01, and having only ReLU activation functions.

Our sixth dataset has 8 classes with 10 features (10 informative) and 5 clusters per class. On this dataset, LDA's accuracy was 28.5%, Naive Bayes' accuracy was 30.2%, QDA's accuracy was 46.8%, and Random Forests' accuracy was 39.4%. We found the highest accuracy for Neural Networks at 40.6% with one layer of 22 neurons, a learning rate of 0.01, 41 epochs, and having only Sigmoid activation functions. For reference, we took the accuracy from 25.58% before the epoch optimization to 40.6% after the optimization.

Our seventh and final dataset has 4 classes with 20 features (15 informative) and 8 clusters per class. On this dataset, LDA's accuracy was 40.2%, Naive Bayes' accuracy was 41.2%, QDA's accuracy was 58.1%, and Random Forests' accuracy was 52.8%. We found the highest accuracy for Neural Networks at 56.3% with one layer of 42 neurons, a learning rate of 0.01, 78 epochs, and having only Sigmoid activation functions. The epoch optimization took our accuracy from 45.28% to 56.3%.

Discussion

Coming into the project, we had high hopes that Neural Networks would be the end-all-be-all for classification problems. While Neural Networks is a powerful method that is good in many scenarios, it has its downfalls and can be beaten by other methods. This is true for any method. There is no one best method for all scenarios according to the "No Free Lunch Theorem" (9, IEEE Journals & Magazine). With this in mind, we will highlight our results and talk about the good and bad of Neural Networks.

In general, we found Neural Networks is capable of performing well in a variety of scenarios. We did not display every finding in the results section because that would make this document go towards 20-plus pages long, but we will talk about some general trends we ran across while making the Neural Networks. We found that one to two layers is a good place to start building a model in general. This was highlighted in the generated datasets as we tested many different size models and one to two layers always won. For the number of neurons, we found the best results were around the number of features + 1, the number of features, the number of features (possibly +1) times 2, or half the number of features. These of course are not set in stone and there were instances of these being broken, but this was a common occurrence amongst our models. This is also in the ballpark of what Heaton recommends (3, Heaton, J.), and other online sources such as MathStackExchange recommend starting around here, too. We found the number of epochs and learning rate to be highly variable depending on the datasets, and it is probably best to try to adjust these as the code runs instead of using the guess and check method. Finally, we recommend starting with the ReLU activation function. This function seemed to give consistently good results. We did not show the results for all the activation functions on each model, but we did see them when testing for the best one. It is safe to say most activation functions have very specific use cases and perform poorly on our datasets. However, ReLU tended to give good results across all the datasets.

As seen from the results, Neural Networks often performed the best or near the best. This was even with our poor optimizations. In the first models, we tried to guess and check the number of epochs and learning rate and still had great results. By dynamically changing these, changing the optimizer, the loss functions, the batch size, and building more models with many different layers of different sizes and activation functions, we think Neural Networks could have the highest accuracy on every dataset we used. Not to mention, we only tuned our hyperparameters on a training/testing split of 0.1/0.025 of the full data and used no cross-validation techniques. Using the whole dataset with cross-validation could also improve our accuracy. All this hyperparameter tuning can be a powerful asset that allows for the user to tune their model very well to their dataset, but we think this a double-edged sword.

We found Neural Networks to be very costly with not much reward on these datasets. On the Smoking/Drinking datasets, we spent around 2 hours running Neural Networks (no GPU). On the Diabetes dataset, we spent around 3 hours running Neural Networks (no GPU). On the Powerlifting datasets, we spent 3 hours running Neural Networks with a GPU and 6 hours running without a GPU. In comparison, it took LDA, Naive Bayes, and QDA around 1-2 minutes

each to run, and Random Forests ran in around 20 minutes. Remember, these non-Neural Network methods were done with 5-fold cross-validation and Random Forests had its hyperparameters tuned; Neural Networks was not done with any cross-validation and was tuned on a very small subset of the data. We did not keep exact track of how long the generated datasets took, but it was roughly the same ratios. This is the biggest drawback of Neural Networks we found. Another drawback was the high barrier to entry. It can feel overwhelming to start learning Neural Networks because there are so many considerations; methods like LDA and QDA have a small learning curve. Finally, Neural Networks is a black-box algorithm. When we get a result from LDA that says 95% accuracy, we can say with a reasonable amount of certainty the data is linearly separable. When we get 95% accuracy back with Neural Networks, we cannot generally say anything about the data except that it seems to be separable somehow without having to deep dive into the data, the hyperparameters chosen, and the way the model learned.

To conclude, we will give some take-home recommendations for Neural Networks. Neural Networks is a computationally heavy, black-box, deep learning algorithm. It performs well in a variety of situations. It is a valuable tool to have but should not be the first used. For many datasets, Neural Networks is like swatting a fly with a sledgehammer. Many classical methods do a great job for a fraction of the cost. But, if classical methods are failing, Neural Networks is a viable option to try.

References

1. *What Are Neural Networks?*. IBM. (n.d.). https://www.ibm.com/topics/neural-networks.
2. Tiwari, S. (2023, February 17). *Activation Functions in Neural Networks*. GeeksforGeeks. https://www.geeksforgeeks.org/activation-functions-neural-networks/
3. Heaton, J. (n.d.). *The Number of Hidden Layers*. The Number of Hidden Layers | Heaton Research. https://web.archive.org/web/20140721050413/http://www.heatonresearch.com/node/707
4. *What Is Cross-Entropy Loss Function*?. 365 Data Science. (2023, June 15). https://365datascience.com/tutorials/machine-learning-tutorials/cross-entropy-loss/
5. *Welcome to PyTorch Tutorials*. Welcome to PyTorch Tutorials - PyTorch Tutorials 2.1.1+cu121 documentation. (n.d.). https://pytorch.org/tutorials/
6. *Adam*. Adam - Cornell University Computational Optimization Open Textbook - Optimization Wiki. (n.d.). https://optimization.cbe.cornell.edu/index.php?title=Adam
7. Jordan, J. (2023, March 5). *Setting The Learning Rate of Your Neural Network*. Jeremy Jordan. https://www.jeremyjordan.me/nn-learning-rate/
8. *Activation Functions in Neural Networks [12 types & use cases]*. V7. (n.d.). https://www.v7labs.com/blog/neural-networks-activation-functions
9. *No Free Lunch Theorems for Optimization*. IEEE Journals & Magazine. (n.d.). https://ieeexplore.ieee.org/document/585893

Datasets

1. The Drinking/Smoking Dataset - https://www.kaggle.com/datasets/sooyoungher/smoking-drinking-dataset
2. The Diabetes Dataset - https://www.kaggle.com/datasets/alexteboul/diabetes-health-indicators-dataset
3. The Powerlifting Dataset - https://www.kaggle.com/datasets/open-powerlifting/powerlifting-database/

Who Was Responsible for What Work

Brandon was responsible for implementing the code, interpreting the output from the code, and leading the project direction throughout. This is now Brandon typing. I wanted to do all the coding parts. I think it is safe to say that I enjoy coding a lot more than Cameron does, and I have a lot more experience coding than Cameron does. I had never tackled this big of a project, so I was very excited to get to do it. After I would type all the code (and thoroughly tested it on very small portions of the data), I would send it to Cameron to run on his computers. He had better hardware than me; plus, I needed my computer to continue working on the other code or just take a break from coding in general. Since I was the one who wrote the code, I understood the most about what the results meant, so I also played a big role in interpreting our results. Finally, I took the lead on the project overall. I told Cameron what stuff I would like him to do, and I did the rest. In this report, I was responsible for about half of the Introduction, Materials and Methods, and a big part of the Discussion section.

Cameron was responsible for doing the background research on the theoretical side of Neural Networks and helped with the computation of the Neural Networks. This is now Cameron typing. Brandon was very eager to implement the code and I was more than willing to take a step back on that stage, so I helped him with running the code that way he wouldn't need to take several multi-hour-long breaks. As Brandon was taking the lead on the programming and results, I focused on the background of Neural Networks and the introduction. I researched the overarching aspects of Neural Networks, mainly focusing on the structure, use cases, and limitations of a Neural Network. Additionally, I researched some of the key aspects of implementing our Neural Networks, like the Adam optimizer, ReLU activation function, and Binary Cross Entropy Loss function. On the report, I wrote the first drafts of the introduction, the results section, contributed to the discussion section, and wrote the abstract.