1. Data Processing

1.1. The extracted dataset

1.1.1. Customer ID as first column/feature

The customer ID should not impact the likelihood the customer comes back to purchase. We do not expect a customer ID to be a critical feature, but it remains within the extracted dataset. We could go within the dataset to remove it or remove it during loading the dataset. I choose the latter option; it may not be the best option overall, but it is slightly more accessible than Excel for me.

```
import numpy as np
from sklearn import preprocessing
import numpy as np
import tensorflow as tf

raw_csv_data = np.loadtxt('Business_case_dataset.csv',delimiter=',')
# print(raw_csv_data)
np.random.shuffle(raw_csv_data)
# print(raw_csv_data)
unscaled_inputs_all = raw_csv_data[:,1:-1]
# print(unscaled_inputs_all)
targets_all = raw_csv_data[:,-1]
# print(targets_all)
```

Figure 1. Loading data.

We start loading to unscaled_inputs_all from the second column (index 1) to cut off the customer ID first column.

1.1.2. There are users who do not leave reviews.

Our audiobook store offers the option to leave a review/rating out of 10, but definitely customers are not forced to leave reviews. The proposed solution is that we can assign some neutral feeling. The customers who have left real reviews and ratings provide a gauge of an overall feeling. They might feel dissatisfied, happy, or indifferent. We assume that customers who did not leave reviews were indifferent in general with no compulsion to go leave reviews or ratings. We calculate that neutral feeling with the average of the existing ratings, which provides us with 8.91.

1.2. Preparing for model training

1.2.1. Feature scaling

Our values aren't exactly ridiculously large, but the features vary in their scale. It would be better for our model if transformed our features onto a standard scale. We can leave this to the library sklearn as the following code.

```
scaled_inputs = preprocessing.scale(unscaled_inputs_equal_priors)
```

Figure 2. Using sklearn library for feature scaling.

1.2.2. Balancing the data

Looking at the dataset, there are significantly more 0's than 1's for the target. Specifically, I found 2237 1's and 11847 0's. What this means is that more of the customers are found to have not come back in the next six months. However, the store wants to identify and target those users who are likely to come back. Leaving the data as is will probably lead the model to be disproportionately biased towards 0 where users won't come back, which is probably not ideal for the store relying on the model.

We balance the data by deleting some data in the majority class 0. The alternative option is to create synthetic data for our minority class 1, but it may be more disingenuous and not as accurate to create fake, similar customers who we say will definitely come back.

The code to delete is as follows in Figure 3.

```
num_one_targets = int(np.sum(targets_all))

zero_targets_counter = 0

indices_to_remove = []

for i in range(targets_all.shape[0]):
    if targets_all[i] == 0:
        zero_targets_counter += 1
        if zero_targets_counter > num_one_targets:
            indices_to_remove.append(i)

# Create two new variables, one that will contain the inputs, and one that will contain the targets.
# We delete all indices that we marked "to remove" in the loop above.
unscaled_inputs_equal_priors = np.delete(unscaled_inputs_all, indices_to_remove, axis=0)
targets_equal_priors = np.delete(targets_all, indices_to_remove, axis=0)
```

Figure 3. Deleting some of the majority class to balance data.

We also make sure that we don't just delete the same rows on every preprocessing run. We're also not sure if there is already an existing order in the extracted dataset or if the data has

already been randomized. If there is an inherent order, we may lose a representative sample. So, as in Figure 1, we do this by shuffling the data right after loading by doing np.random.shuffle(raw_csv_data).

1.2.2.1. Reshuffling

In the deletion algorithm in Figure 3, it really just deletes the all the 0's once we see a matching balance of 0's and 1's. However, depending on how the data was initially shuffled, this does not mean that the data can be evenly matched. This is because there are potentially 1's that get grouped together after the deletion, so we need to reshuffle. We do show as in the following code.

```
# Shuffle the indices of the data, so the data is not arranged in any way when we feed it.
shuffled_indices = np.arange(scaled_inputs.shape[0])
np.random.shuffle(shuffled_indices)

# Use the shuffled indices to shuffle the inputs and targets.
shuffled_inputs = scaled_inputs[shuffled_indices]
shuffled_targets = targets_equal_priors[shuffled_indices]
```

Figure 4. Reshuffling data after deleting for balance.

1.2.3. Dividing data into Training, Validation, and Testing

We have two general choices here: 80%, 10%, 10% respectively or 70%, 20%, 10% respectively. (Percentages are taken of the total data as we are dividing up the whole dataset). We want to have a nice balance of 0's and 1's, so that will be our goal. We'll start off with 80, 10, 10 as with code and result below.

```
training_split = 0.8
validation_split = 0.1

samples_count = shuffled_inputs.shape[0]

train_samples_count = int(training_split * samples_count)
validation_samples_count = int(validation_split * samples_count)

test_samples_count = samples_count - train_samples_count - validation_samples_count

train_inputs = shuffled_inputs[:train_samples_count]
train_targets = shuffled_targets[:train_samples_count]

validation_inputs = shuffled_inputs[train_samples_count:train_samples_count+validation_samples_count]
validation_targets = shuffled_targets[train_samples_count:train_samples_count+validation_samples_count]

test_inputs = shuffled_inputs[train_samples_count+validation_samples_count:]
test_targets = shuffled_targets[train_samples_count+validation_samples_count:]


# Print the number of targets that are 1s, the total number of samples, and the proportion for training, validation, and test (~50%).
print(np.sum(train_targets), train_samples_count, np.sum(train_targets) / train_samples_count)
print(np.sum(validation_targets), validation_samples_count, np.sum(validation_targets) / validation_samples_count)
print(np.sum(test_targets), test_samples_count, np.sum(test_targets) / test_samples_count)
```

```
1800.0 3579 0.5029337803855826
213.0 447 0.47651006711409394
224.0 448 0.5
```

Figure 5.a. Balance result of 80, 10, 10 split at bottom. Code at top.

```
1569.0 3131 0.5011178537208559
443.0 894 0.4955257270693512
225.0 449 0.5011135857461024
```

Figure 5.b. Balance result of 70, 20, 10.

From Figures 5.a and 5.b, we get a trade of more training data if we use 80, 10, 10 or more validation data 70, 20, 10. This was obvious if we looked at the change in percentages. When considering overfitting, I do definitely like having more validation data in the 70, 20, 10 split. However, I also would like to have more training data as in the 80, 10, 10 split, but the validation dataset is less than the test dataset. Instead, I want to try a nice medium like 75, 15, 10.

```
1681.0 3355 0.5010432190760059
332.0 671 0.4947839046199702
224.0 448 0.5
```

Figure 5.c. Balance result of 75, 15, 10.

Results from Figure 5.c seem more appealing that it slightly fits our wanted data amounts better and the sets are more balanced between 0's and 1's. Let's see what would happen if we try to hit more of a split like 1700 in training data and 300 in validation data. So, let's go with adjusting by one percent and try 76, 14, 10.

```
1709.0 3400 0.5026470588235294
304.0 626 0.48562300319488816
224.0 448 0.5
```

Figure 5.d. Balance result of 76, 14, 10.

This seems like a nicer roundoff, but the balance between 0's and 1's in the training and validation sets are worse. Our initial goal was to have almost an equal balance of 0's and 1's, so we'll go with the 75, 15, 10 split. We'll be using this data partition in the later steps.

2. Hyperparameter Tuning

2.1. The Model

It is reasonably unfeasible to exhaustively and manually search all possible hyperparameter combinations. Instead, we'll try to focus on fine-tuning one or a few hyperparameter(s) at a time.

Our initial settings for the model are as follows.

- Input size = 10
- Output size = 2
- Output layer activation function = softmax
- # of hidden layers = 2
- Hidden Layer #1
    - 50 hidden nodes
    - Activation Function = relu
- Hidden Layer #2
    - 50 hidden nodes
    - Activation Function = relu
- Optimizer = ADAM
- Loss function = sparse_categorical_crossentropy
- Batch size = 100
- Maximum epochs (may not hit from early stopping) = 100
- Early stopping patience (i.e. tolerance for no improvement in loss) = 2
- Initializer = Xavier Glorot initialization (TensorFlow default)

We will be keeping the input, output layer, loss function, and initializer. The inputs and outputs come from our extracted data, which we will not change. We keep softmax on the output layer as there are two output nodes. We keep the loss function because we are doing classification and most likely using one-hot encoding given that we have only two categories, which are few and do not necessitate binary encoding. For the initializer, we'll just use the state-of-the-art Xavier Glorot that is the default setting from TensorFlow.

We will be adjusting the hidden layers, optimizer, batch size, and patience.

For the experiments, I will try running the same configuration of parameters multiple times - roughly around three times each. I will then report results from the highest accuracy run. I do so because each run of the model may yield different results since the initializer would likely pick a different point each time. We also don't really want to pivot ourselves to a point that we can't easily tell will be optimal.

2.2. Patience

Initial Test

Optimizer	Batch size	Patience	Hidden Layer #1 Activation	Hidden Layer #1 Size	Hidden Layer #2 Activation	Hidden Layer #2 Size	Test accuracy (%)
ADAM	100	2	relu	50	relu	50	91.74

First, I notice that the early stopping mechanism with patience 2 activates at epoch 16. The validation loss first got greater at epoch 15. This is an indication that we most likely start overfitting at the first time, but we don't have any information further to know if this could've been an intermediate increase in loss. To further test this, we'll try patience = 0, 1, 3, or 4 to hopefully see more information.

Patience Test

Optimizer	Batch size	Patience	Hidden Layer #1 Activation	Hidden Layer #1 Size	Hidden Layer #2 Activation	Hidden Layer #2 Size	Test accuracy (%)	Epochs where patience decrease
ADAM	100	0	relu	50	relu	50	92.19	12
ADAM	100	1	relu	50	relu	50	91.52	11, 12
ADAM	100	3	relu	50	relu	50	92.86	21, 29, 31
ADAM	100	4	relu	50	relu	50	92.41	18, 21, 22, 24

It seems the best choices are no patience, or patience = 3 or 4. Patience = 3 seems to be the best, but I think no patience is also acceptable. In patience = 1, we see a strictly increasing overfitting trend from the consecutive validation loss at the end. Cases like these mean having no tolerance and stopping at the first violation would be ideal to stop overfitting.

Experiments with patience = 3 or 4 showed that there may also be cases where intermediate epochs see the validation loss increase and decrease soon after. However, it seems that patience = 4 seems to trend towards overfitting at the end as the test accuracy actually decreased. We want to avoid that overfitting and allow some of the intermediate fluctuations, so we'll go with patience = 3 in later experiments. In addition, we can find the first patience violation and consider it as our test as if we had done patience = 0.

2.3. Batch Size

Our goal here for the audiobook store is the highest accuracy. We don't necessarily have a significant amount of data that it would greatly impact our speed. Let's try adjusting the batch size next.

The idea of batching is to update weights after each batch instead of each epoch. In doing so, we reach a minimum quicker with a little loss in our accuracy. Theoretically, if we use bigger batches, we might increase our accuracy. At the same time, smaller batches may be able to help reach the minimum quicker and stopping us immediately as we are about to overfit. Our initial value is 100, so let's try 20, 25, 50, 75, 150, 200, 250, 500, 1000, and 2000 (to cover the whole training dataset). We compare with an initial test.

Initial Batch Size of 100

Optimizer	Batch size	Patience	Hidden Layer #1 Activation	Hidden Layer #1 Size	Hidden Layer #2 Activation	Hidden Layer #2 Size	Test accuracy (%)
ADAM	100	3	relu	50	relu	50	92.86

Batch Size Test

Optimizer	Batch size	Patience	Hidden Layer #1 Activation	Hidden Layer #1 Size	Hidden Layer #2 Activation	Hidden Layer #2 Size	Test accuracy (%)	Consecutive validation loss increases at end	Special note
ADAM	20	3	relu	50	relu	50	92.41	3	Lowest validation loss of 0.2267 at epoch
ADAM	20	**1**	relu	50	relu	50	91.52	2	
ADAM	20	**0**	relu	50	relu	50	92.41	1	
ADAM	25	3	relu	50	relu	50	92.86	1	
ADAM	25	**2**	relu	50	relu	50	92.19	2	Lowest validation loss of 0.2309 at epoch 10/13, so try 0 patience
ADAM	25	**0**	relu	50	relu	50	92.63	1	
ADAM	50	3	relu	50	relu	50	92.63	2	
ADAM	50	**2**	relu	50	relu	50	91.74	1	
ADAM	75	3	relu	50	relu	50	92.19	2	
ADAM	75	**2**	relu	50	relu	50	91.74	1	
ADAM	150	3	relu	50	relu	50	93.08	1	Lowest validation loss of 0.2264 at epoch 23/26

ADAM	150	**0**	relu	50	relu	50	91.29	1	
ADAM	200	3	relu	50	relu	50	92.41	1	
ADAM	250	3	relu	50	relu	50	92.63	1	
ADAM	500	3	relu	50	relu	50	92.19	1	
ADAM	1000	3	relu	50	relu	50	91.74	1	
ADAM	2000	3	relu	50	relu	50	92.41	1	

Bolded, underlined cells indicate that we are testing with same parameters but changing patience to see if we can prevent overfitting such as in Figure 6.

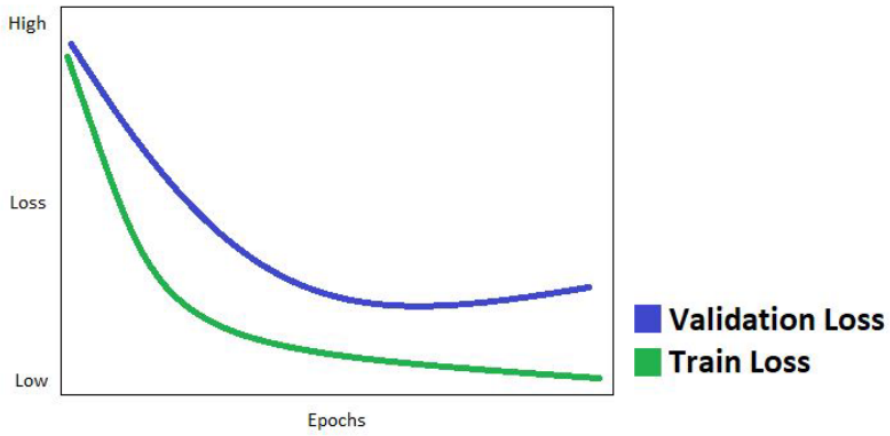

Figure 6. Example of overfitting model. This is what we want to avoid, especially with higher patience. Thus, we check the consecutive validation loss increases at end.

Overall in our batch size test, it seems that a batch size of 150 and patience of 3 yielded the highest test accuracy of 93.08%. Another decent setting seems to be a batch size of 25 and a patience of 3, tied with our initial test accuracy of 92.86%. Otherwise, the initial setting of 100 for batch size seemed to have been a luckily wise decision as it outperformed many of the other new tests.

Interpreting the results, I think an optimal choice for the batch size is neither extremely small nor extremely big. The reason why I didn't test 1 was because I knew it would be terrible because it would be harder to converge by noise from the many batch gradient estimates. However, I think what happened for 25 to perform well is that the noise may have pushed our estimate towards a global minimum (maybe not in the global minimum but closer). For 150, I think it was just comparatively more accurate steps towards the minima while retaining a little bit of the noise momentum. I think this may be why it performed better than its bigger batch size counterparts.

2.4. Optimizer and Learning Rate

On the topic of momentum, let's try changing the optimizer. We'll try four: SGD, AdaGrad, RMSProp, and ADAM.

Optimizer Test

Optimizer	Batch size	Patience	Hidden Layer #1 Activation	Hidden Layer #1 Size	Hidden Layer #2 Activation	Hidden Layer #2 Size	Test accuracy (%)	Epochs	Special note
SGD	150	3	relu	50	relu	50	91.29	169	
AdaGrad	150	3	relu	50	relu	50	89.96	>500	Took more than 500 epochs, we set max to 1000
AdaGrad	150	3	relu	50	relu	50	91.52	>1000	Exceeded 1000 epochs with decreasing trend in loss
AdaGrad	150	0	relu	50	relu	50	90.62	877	Set patience to 0 to see when AdaGrad would stop
RMSProp	150	3	relu	50	relu	50	92.41	20	
ADAM	150	3	relu	50	relu	50	93.08	23	

Of the four, we find that ADAM performs the best. The improvements from changing optimizers makes sense. SGD (Stochastic Gradient Descent) performed the worst, but we added an adaptive learning rate factor with AdaGrad. AdaGrad performed better; however, it took many more epochs to surpass SGD. However, AdaGrad gave the learning rate factor its full weight, whereas RMSProp distributes weight among the learning rate factor and previous influence.

Additionally, as shown in the table, the rate definitely does adapt much more efficiently. ADAM took all these and added a momentum factor, resulting in the most advanced optimizer among the four and offered the highest test accuracy.

Let's try adjusting the learning rate hyperparameter on ADAM.

Learning Rate Test

Learing rate	Optimizer	Batch size	Patience	Hidden Layer #1 Activation	Hidden Layer #1 Size	Hidden Layer #2 Activation	Hidden Layer #2 Size	Test accuracy (%)
0.001	ADAM	150	3	relu	50	relu	50	92.63
0.01	ADAM	150	3	relu	50	relu	50	91.74
0.1	ADAM	150	3	relu	50	relu	50	90.40
0.05	ADAM	150	3	relu	50	relu	50	93.30
0.005	ADAM	150	3	relu	50	relu	50	91.52
0.0005	ADAM	150	3	relu	50	relu	50	92.19

It seems that the best learning rate for our current model using ADAM would 0.05.

2.5. Hidden Layers

Finally, we adjust the hidden layers.

2.5.1. Preface Note

I have left the hidden layers as the last to be adjusted. The point of the hidden layers is to introduce non-linearity and be able to represent more complex models. However, there are evidently many different possibilites, especially considering permutations of hidden layers with their own sizes and activation functions. We also don't want to have too many hidden layers because that may cause dead neurons in those extra layers, negatively impacting how effective backpropagation will be. Here, I'll go with an incremental approach with one hidden layer at a time.

I conduct the previous experiments to better guide and try to better pinpoint improvements relevant to the changed hidden layer. Otherwise, we deal with many confounding variables at once. Nevertheless, they still have impact on each other; the patience, the batch size, and the optimizer all impact the test accuracy from which we assess improvement.

The previous experiments all have two hidden layers - each with 50 hidden nodes and relu as the activation function. I leave them as so since relu has supposedly “become the default activation function used across the deep learning community” (Ramachandran et al., 2017). All previous experiments have one or very few adjustments from a control that should not invalidate the test results and conclusion.

2.5.2. 1 Hidden Layer

For activation functions, we’ll mainly be looking at four of them: Sigmoid, TanH, ReLU, ELU.

We’ll begin with trying out just one hidden layer and see what happens.

1 Hidden Layer Test

Learning rate	Optimizer	Batch size	Patience	Hidden Layer #1 Activation	Hidden Layer #1 Size	Test accuracy (%)
0.05	ADAM	150	3	relu	10	91.07
0.05	ADAM	150	3	relu	50	91.52
0.05	ADAM	150	3	relu	100	91.52
0.05	ADAM	150	3	sigmoid	10	91.29
0.05	ADAM	150	3	sigmoid	50	91.29
0.05	ADAM	150	3	sigmoid	100	91.07
0.05	ADAM	150	3	tanh	10	91.07
0.05	ADAM	150	3	tanh	50	91.52
0.05	ADAM	150	3	tanh	100	91.52
0.05	ADAM	150	3	elu	10	90.85
0.05	ADAM	150	3	elu	50	91.07
0.05	ADAM	150	3	elu	100	91.07

For only one hidden layer, it seems that having many more nodes than inputs didn’t have much of an impact on the resulting test accuracy. It could be possible that some weights were set to near 0 and only the most prominent features were taken into consideration by the model. From this, let’s use either relu or tanh of size 50 or 100 for the first hidden layer.

2.5.3. 2 Hidden Layers

Let's try two hidden layers. In our initial setting, the two hidden layers had the same size and ReLU activation function. The ReLU activation function has a Dying ReLu problem, so let's try ELU that doesn't have that problem.

2 Hidden Layers - ELU Test

Learning rate	Optimizer	Batch size	Patience	Hidden Layer #1 Activation	Hidden Layer #1 Size	Hidden Layer #2 Activation	Hidden Layer #2 Size	Test accuracy (%)
0.05	ADAM	150	3	relu	50	elu	10	91.74
0.05	ADAM	150	3	relu	50	elu	50	90.85
0.05	ADAM	150	3	relu	50	elu	100	90.40
0.05	ADAM	150	3	relu	100	elu	10	91.52
0.05	ADAM	150	3	relu	100	elu	50	91.29
0.05	ADAM	150	3	relu	100	elu	100	91.52
0.05	ADAM	150	3	elu	50	elu	10	91.07
0.05	ADAM	150	3	elu	50	elu	50	90.62
0.05	ADAM	150	3	elu	50	elu	100	90.85
0.05	ADAM	150	3	elu	100	elu	10	92.65
0.05	ADAM	150	3	elu	100	elu	50	91.52
0.05	ADAM	150	3	elu	100	elu	100	91.52

The best so far remains the test with 0.05 learning rate, but another contender is the first hidden layer having ELU and size 100 and the second layer with ELU and size 10.

Of course, there are many more possibilities that can be tested. However, rather than try every possible combination, I would like to try process of elimination of what activation functions might not fit well together. While I recognize that a hidden layer node takes input from all node outputs in the previous layer, we'll be able to prune the search space to be at least more reasonable.

Sigmoid and TanH have the Vanishing Gradient problem that the adjustment during backpropagation for further layers may "vanish" or too small to be effective. This means that maybe we should try not to put them near the output layers. ReLU doesn't seem to be as

affected by this Vanishing Gradient problem, so it can go near the output. We may even consider ELU to fix the Dying ReLU problem.

Let's look at the ranges. Sigmoid is nearly linear near 0, centered at 0.5, and output is (-1, 1). TanH seems to nearly linear near 0, centered at 0, and output is also (-1, 1). ReLU looks to be a piecewise function that appears linear at ranges separated by the cusp at 0. Its range is (0, positive infinity). ELU, on the other hand, looks exponential as in the name and is differentiable across all inputs. Its range seems to be (-1, positive infinity). Taking these into account, it seems that - for example - putting ReLU after Sigmoid or TanH may not be too good if the potential negative output is pushed to 0.

2 Hidden Layers - Sigmoid and TanH Test

Learning rate	Optimizer	Batch size	Patience	Hidden Layer #1 Activation	Hidden Layer #1 Size	Hidden Layer #2 Activation	Hidden Layer #2 Size	Test accuracy (%)
0.05	ADAM	150	3	elu	100	tanh	10	91.07
0.05	ADAM	150	3	elu	100	tanh	50	90.40
0.05	ADAM	150	3	elu	100	tanh	100	91.52
0.05	ADAM	150	3	elu	100	sigmoid	10	91.29
0.05	ADAM	150	3	elu	100	sigmoid	50	90.62
0.05	ADAM	150	3	elu	100	sigmoid	100	93.08

With the exception of the last row with second hidden layer with 100 hidden nodes and sigmoid activation function, the other tests did not perform well. I think the reason might be similar to not using softmax in intermediate hidden layers because we lose variability of the data based on the output. The last row may have been lucky, but we can test this by adding another hidden layer and observing its behavior.

2.5.3. More Hidden Layers

So far, the best result from 2 hidden layers are as follows.

0.05	ADAM	150	3	elu	100	sigmoid	100	93.08
0.05	ADAM	150	3	relu	50	relu	50	93.30
0.05	ADAM	150	3	elu	100	elu	10	92.65

There could be a possibility that we are reaching the maximum that the complexity of the model can carry us. We have a good base, so let's test with adding more complexity/hidden layers instead of just one.

More Hidden Layers Test

HL = Hidden Layer

Learning rate	Optimizer	Batch size	Patience	HL #1 Activation	HL #1 Size	HL #2 Activation	HL #2 Size	HL #3 Activation	HL #3 Size	HL #4 Activation	HL #4 Size	Test accuracy (%)
0.05	ADAM	150	3	elu	100	sigmoid	100	relu	50	N/A	N/A	92.41
0.05	ADAM	150	3	elu	100	sigmoid	100	relu	25	N/A	N/A	92.63
0.05	ADAM	150	3	elu	100	sigmoid	100	elu	25	relu	50	92.19
0.05	ADAM	150	3	elu	100	sigmoid	100	elu	50	relu	25	90.86
0.05	ADAM	150	3	elu	100	sigmoid	100	elu	50	relu	100	91.96
0.05	ADAM	150	3	relu	50	relu	50	sigmoid	100	N/A	N/A	93.53
0.05	ADAM	150	3	relu	50	relu	50	sigmoid	100	elu	10	90.83
0.05	ADAM	150	3	relu	50	relu	50	sigmoid	150	N/A	N/A	91.74
0.05	ADAM	150	3	relu	50	relu	50	tanh	100	N/A	N/A	89.09
0.05	ADAM	150	3	relu	50	relu	50	elu	10	sigmoid	100	91.07
0.05	ADAM	150	3	elu	100	elu	10	relu	50	N/A	N/A	92.86
0.05	ADAM	150	3	elu	100	elu	10	sigmoid	100	relu	50	91.76
0.05	ADA	150	3	elu	100	elu	10	tanh	50	relu	50	94.6

	M											5
0.05	ADAM	150	3	elu	100	elu	10	tanh	100	relu	50	92.20
0.05	ADAM	150	3	elu	100	elu	10	relu	50	sigmoid	100	91.98

It looks like the best overall adjustments were the following with a test accuracy of 94.65%.
Learning rate = 0.05
Optimizer = ADAM
Batch size = 150
Early stopping patience = 3
Hidden Layers = 4
Hidden Layer #1:
    Activation Function = elu
    Size = 100
Hidden Layer #2:
    Activation Function = elu
    Size = 10
Hidden Layer #3:
    Activation Function = tanh
    Size = 50
Hidden Layer #4:
    Activation Function = relu
    Size = 50

References

Ramachandran, P., Zoph, B., & Le, Q. V. (2017). Searching for activation functions. https://doi.org/10.48550/ARXIV.1710.05941