Deep Learning 1

375893

December 2020

1 Perceptron

The Perceptron is the simplest implementation of a neural network. I have adapted the code from per.R to be adapted to our problem and training data. We first construct the training data, which consists of a matrix of 1000×10 numbers drawn from a random probability with the `sample` function, set to be either 1 or -1. They are indeed $2^{10} = 1024$ different possibilities of input, and I therefore chose to represent our entire database with 1000 total input (for both training and testing), to see whether the Perceptron is able to learn the classification rule even though it is not presented with all of the data. We then split up our data according to a 80/20 split to create the training and testing datasets. We are now in measure to run the perceptron algorithm, which basically applies the learning rule:

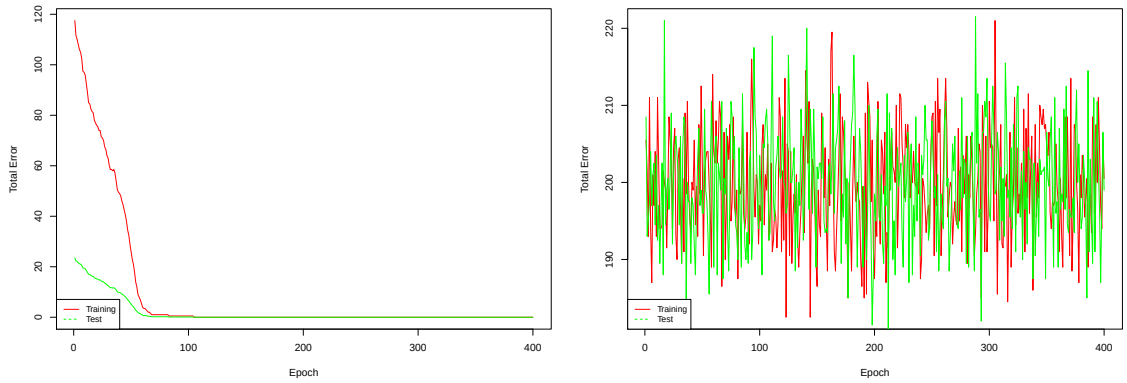
$$\Delta w = \begin{cases} -\epsilon x & \text{if } t < 0 \text{ and } \hat{y} > 0 \\ \epsilon x & \text{if } t > 0 \text{ and } \hat{y} < 0 \end{cases} \tag{1}$$

where t is the target and ŷ is the prediction given by $\hat{y} = \sum w_i x_i + b$. We can then represent the error at each epoch by the sum $\sum 0.5 \times (t - \hat{y})^2$, over all row in the dataset. After updating each weight according to the rule in 1, we predict labels for the test dataset and calculate the total error too.

1.1 Classifying the sum $\sum x_i$

We calculate the sum of each row of our dataset $\sum x_i$ with `rowSums` function in R, assigning a label 1 to positive and null values, and 0 to negative ones. We represent the total error for the training and testing dataset in Figure 1a.

After 100 epochs, the total loss goes down to 0, meaning the problem is solved by the perceptron. It is important to note here that we have deliberately put extreme values for our initial weights, for us to see the learning operate. If we do not do so, and assign weights with runif between 0 and 1, the learning essentially happens in one single epoch. We therefore multiply the runif vector by 99. For the same reason, the ϵ is set extremely low, at 0.01, to allow for slow learning, and better visualisation. The perceptron generalises to the testing dataset, with its loss function being essentially a scale down version of the loss from the testing data (as they are 200 vs 800 training data). This problem is quite easy to solve for the perceptron, as it is conceptually easy to see how it is possible to draw a hyper-plane able to separate our two classes in 10 dimensions. Indeed, by analogy with the 3D case, drawing a plane passing thought the origin and inclined on the y-z plane such that points with (1,1,1), (1,1,-1), etc are part of one class and points (-1,-1,1), (1, -1,



(a) Total Error for the sum

(b) Total Error for the product

Figure 1: Error per epoch for each task

-1) are not will do the trick. The final bias is b = -5.86266976 and the final weights are weights are [[5.80135002, 5.66868077, 4.53220123, 4.29796836, 6.47657652, 5.40859012, 5.75625067, 5.39578113, 5.16903275, 6.00299032]]. The neural net is assigning each input similar weights. The values indeed do not represent any physical importance, and each of them is as important in determining the total sum. The bias is negative, and of absolute similar value, insuring that the sum is indeed above 0 for a positive output. This is similar to doing an average of the 10 inputs, and insuring that the final result is positive or null.

1.2 Classifying the product $\prod x_i$

We calculate the product of each row of our dataset $\prod x_i$ with the `prod` function in R, assigning a label 1 to positive values, and 0 to negative ones. We represent the total error for the training and testing dataset in Figure 1b.

As we can see, the network does not manage to learn a classification rule, even on our training data. This is because the data represents essentially a XOR problem in multi-dimension. Indeed, limiting ourselves to n=2 for each input vector, we can see that (-1,-1) and (1,1) are classified as class 1 and (-1,1) and (1,-1) are classified as class 0. The 10D representation of our data is much harder to visualise, but will correspond to a similar XOR segmentation of 10D space whereby coordinates with even number of minuses will be considered as class 1 and those with odd number of minuses will be considered as class 0. This is even harder for the perceptron to do than the 2D classification, which itself cannot be solved by a perceptron.

2 Self Supervised Learning

We will use python to code an object oriented implementation of a neural net from scratch, using the `numpy` library. This approach will allow modularity, allowing to change the neural net architecture, loss function and transfer function as pleased. To do so, we create two classes:the `NeuralNet` class,

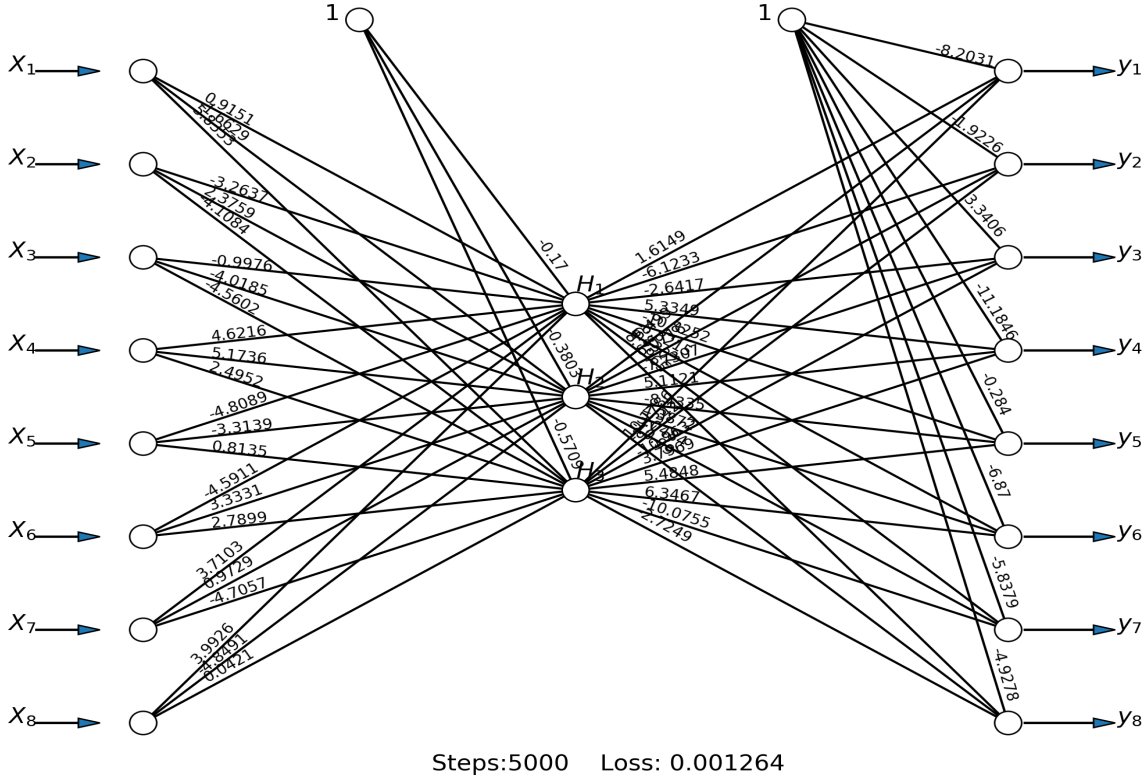


Figure 2: Final weights for the neural net

which itself will contain a list of staked `Layer` objects.

2.1 The `Layer` class

We first create an empty parent object, that contains the skeleton of what we want our `Layer` class to be. We want each layer to be modular, producing an array with all the output from an input one. The class therefore contains two methods: `forward_propagation` and `backward_propagation`. We will use matrix algebra to calculate all ouputs Y from intputs X in forward mode, and all derivatives of the Error E with respect to the input X, $\frac{\partial E}{\partial X}$ for a given $\frac{\partial E}{\partial Y}$ in reverse mode. We need two different type of Layers that will inherit from this parent class:

1. **The Fully connected layer:** This layer contains in total $w_{i,j}$ weights, with i the total number of input nodes and j the total number of output nodes. The value of each output

neuron is calculated using the formula $y_j = b_j + \sum x_i w_{i,j}$ We use matrix algebra to do this in one single operation for every output: $Y = XW + B$. The backward propagation applies the three following formulas:

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} W^t \quad \frac{\partial E}{\partial W} = X^t \frac{\partial E}{\partial Y} \quad \frac{\partial E}{\partial B} = \frac{\partial E}{\partial Y} \tag{2}$$

2. **The Activation layer:** This layer does a one to one connection between input an ouput nodes, and applies the activation function $Y = f(X)$ for the forward mode, and the backward formula: $\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} \odot f'(X)$, where $\odot$ denotes the element wise multiplication of two matrices.

2.2 The `NeuralNet` class

The `NeuralNet` contains a list storing each successive `Layer` objects. For our exercise, this will constitute of a first `FCLayer` with 8 input nodes and 3 output ones. We then `add` a `ActivationLayer` with a the sigmoid activation function. We add a second `FCLayer` with 3 input nodes and 8 output ones, implementing the sigmoid activation function too. We specify that we will use the mean squared error loss function to the `use` method. We can not train our network, using the `fit` method, to `x_train`, which contains the 8 different unit vectors, as input and output. We represent the final learned weights in Figure 2.

2.3 Influence of the learning rate

The neural net is able to solve the self-assignment problems in around 5000 epochs, with our initial learning rate, arbitrarily set to 0.5, with a final error or 0.001264. As an example, here is the prediction for the first input vector: [0.91584637, 0.00000128, 0.00003554, 0.03905836, 0.0416504, 0.00920911, 0.00001883, 0.05046158]. As can be seen on Figure 3, higher learning rates will converge faster to the solution, with the final error at epoch 3000 being 0.039636 for a learning rate of 0.2 compared to 0.009009 for a learning rate of 0.95. They is however diminishing return, and is limited benefits of setting the learning rate above 0.8

2.4 Influence of the momentum α

We can visualise the impact of varying α for a set learning rate in Figure 4. A bigger momentum will speed up the learning process, and this is more obvious for a smaller learning rate. As we can see on the graphs, a bigger momentum means that the curve "picks up some speed" when the gradient starts falling down (around the 500 epoch mark in the first graph).

2.5 How the Neural Net learns

Looking at the values of the weights in Figure 2, bias as well as the output of each input vector in the hidden units doesn't tell us very much about how the NN has "learned". We could, as a human think of a very simple and elegant way to code in 3 neurons the the 8 initial vectors, indeed, as $2^3 = 8$, we could think of simply representing a binary representation of our input vectors in the hidden unit. The NN however seems to take on a different but in a sense similar strategy. Looking at weights in the first layer, it seems that each has a different permutation of the negative weights. This allows our neural net to differentiate between our 8 vectors.

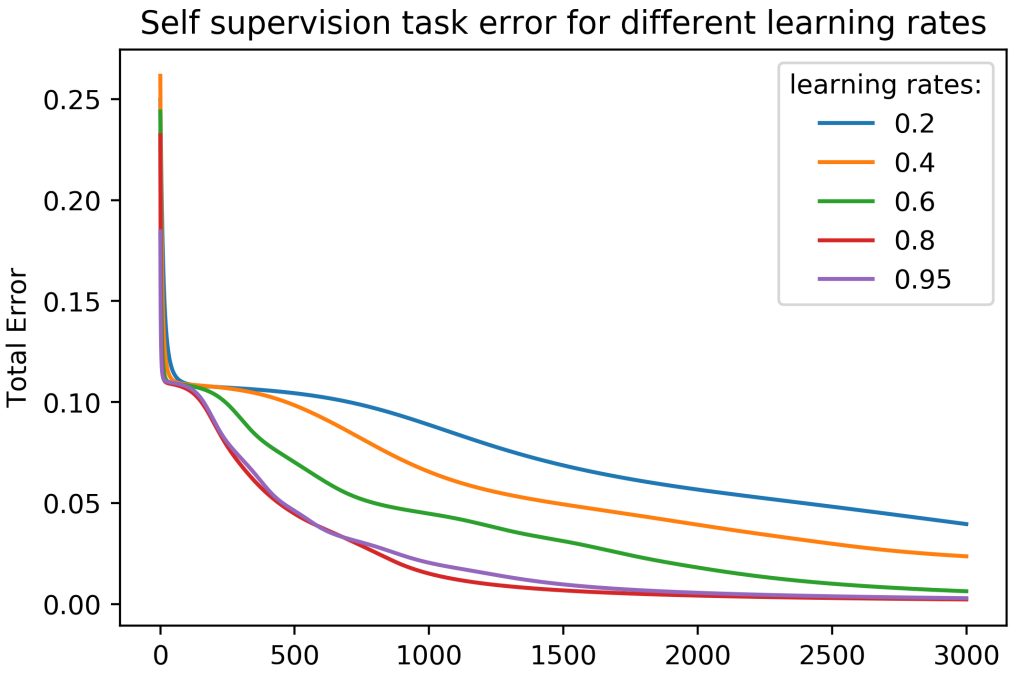


Figure 3: Effect of learning rate on the loss function

3 MNIST classification

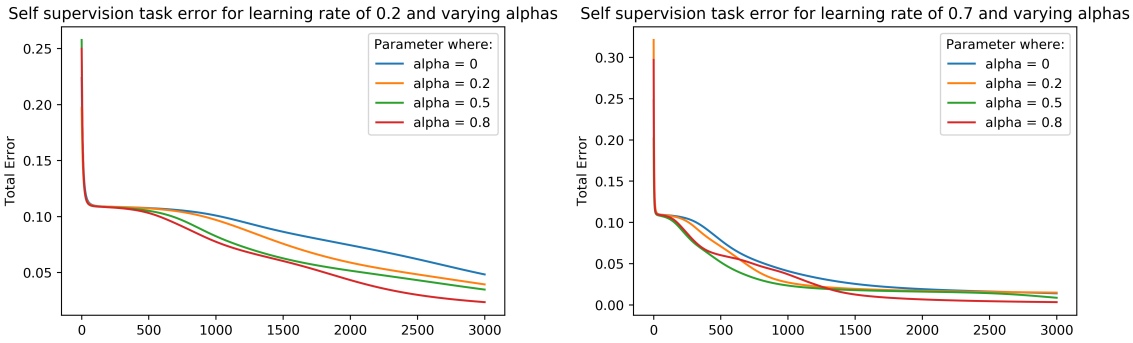
We use the Keras library under python to classify the MNIST database. We will work with the categorical cross entropy loss function.

3.1 Influence of layer depth

We vary the number of nodes in a single layer model, between 32, 256, 1024 and 2048, and we visualise the learning process for the first 4 epochs in Figure 5. As can be seen, increasing the number of nodes improve the accuracy, from around 88% with 32 nodes to 95% with 2048 nodes.

3.2 Influence of number of layers

Increasing the number of layers has more nuanced effect. Indeed, by continually staking layers of 32 nodes, we see improved performance on the first five epochs up until a total of three layers (Figure 5b). However, when we have four layers, we see that the learning fails to generalise as the training accuracy increases but the validation one does not. This could be due to multiple reasons. The first possible reason to investigate is over-fitting, meaning that the network fails to generalise the learning process. Neural nets are prone to over-fitting because of the very large number of tunable



(a) Varying alpha for learning rate of 0.2

(b) Varying alpha for learning rate of 0.2

Figure 4: Influence of momentum on learning

parameters. However, this does not seem to be the case here as the network was fine for the other layered implementation. The problem here is bit more subtle. It is due to the small number of nodes per layer. This was not an issue when we had less layers, but adding more and more layers here will compress our initial 784 pixels picture to only 32 nodes, and so repeatedly. By analogy, we could think of compressing multiple times a jpeg. To increase our network accuracy, we must have a good blend of both depth and width. Moreover, the bigger the network architecture, the more epoch of training will the network need to get to good solutions. We have recapitulated results of a combination of width and depth in Table 1, where each network has been trained on Epoch = layers$\times$10.

Nb of nodes	32	128	512
3 Layers	0.855	0.671	0.208
4 Layers	0.922	0.905	0.424
5 Layers	0.937	0.933	0.904

Table 1: Accuracy per neural net implementation.

It would therefore seem as though the best architecture consists of a 5 layers $\times$ 32 nodes. However, the results are also dependant on the time taken to train the network, itself dependant on the number of epoch and the architecture. In real world application, the computing resources and cost are also another limitation. The choice of right hyper-parameters is therefore not only an objective but a subjective one, depending on the time an resources available. It is highly likely that traing on more epoch will show the best performance for the 5 layers $\times$ 128 nodes, or 5 layers $\times$ 512 nodes.

3.3 Learning rate

Figure 6 represents the accuracy per epoch of a 4 $\times$ 32 neural net trained on 50 epochs. As we can see, this dataset works only for low learning rate, and becomes unstable with a learning rate

over 0.1. On the other hand, taking a learning rate too small will slow down learning. we select a learning rate of 0.001 for the rest of the exercise. (note how this learning rate is much smaller than in the previous exercise, reflecting also the complexity of the problem).

3.4 Optimisation algorithm choice

We test out 4 different learning algorithms, for our 4 $\times$ 32 neural net trained on 50 epochs: a stochastic gradient descent and three adaptive optimisers, (Adagrad, Adam and RMSprop) meaning they don't require a tuning of the learning rate value.

The results are shown in Figure 6. As we can see, stochastic gradient descent (SGD) and Adagrad perform the worst on our data, and both show delayed accuracy increase. This is probably due to the landscape of our loss function, that is not suited for both optimisers. Moreover, we observe that the SGD fluctuates a lot, and is a result of its definition, as the SGD updates are frequent and with a high variance. We used a momentum of 0.8 to train this network. Adagrad adapts the learning rate to the parameters performing small updates for frequently occurring features and large updates for the rarest ones. In this way, the network is able to capture information belonging to features that are not frequent, putting them in evidence and giving them the right weight. The problem of Adagrad is that it adjusts the learning rate for each parameter according to all the past gradients. So, the possibility of having a very small learning rate after a high number of steps — resulting from the accumulation of all the past gradients — is relevant[1]. This is precisely the problem we are encountering in our training. RMSprop improves Adagrad by introducing a history window which sets a fixed number of past gradients to take in consideration during the training. In this way, we don't have the problem of the vanishing learning rate. This works particularly well for our data, and we achieve a final accuracy value of 0.946. Finally, Adam adds to RMSprop, by storing an exponentially decaying average of past gradients, similarly to momentum for gradient descent. This optimiser shows a very slight improvement over RMSprop, with a final accuracy value of 0.950. It therefore seems that both Adam and RMSprop are appropriate for this dataset.

3.5 Final choice of hyper parameters and confusion matrix

Building from our previous observations, we test our final model with 5 layers $\times$ 32 nodes, and the Adam optimiser, optimised on 50 epochs. We represent the confusion matrix of this network in Figure 7, with the y axis representing the true labelled value, and the x representing the label assigned by our neural net. As we can see, the net is having particular trouble in identifying 9s, that are often misidentified as 4s. The second most misidentified pair is the 3 misidentified as a 5. This confusion matrix in the whole makes sense, and we would expect a human assigned the same task to have a similar confusion matrix. We have represented a few misidentified 9s on the figure. It is understandable how the second and third 9 could be misunderstood for a 4, however, to a human observer, the first error seems less evident. As such, the performance of our model can still be improved, particularly with the use of Convolutions Neural Nets, which are now the gold standard for image analysis.

[1] 7 tips to choose the best optimizer, https://towardsdatascience.com/7-tips-to-choose-the-best-optimizer-47bb9c1219e

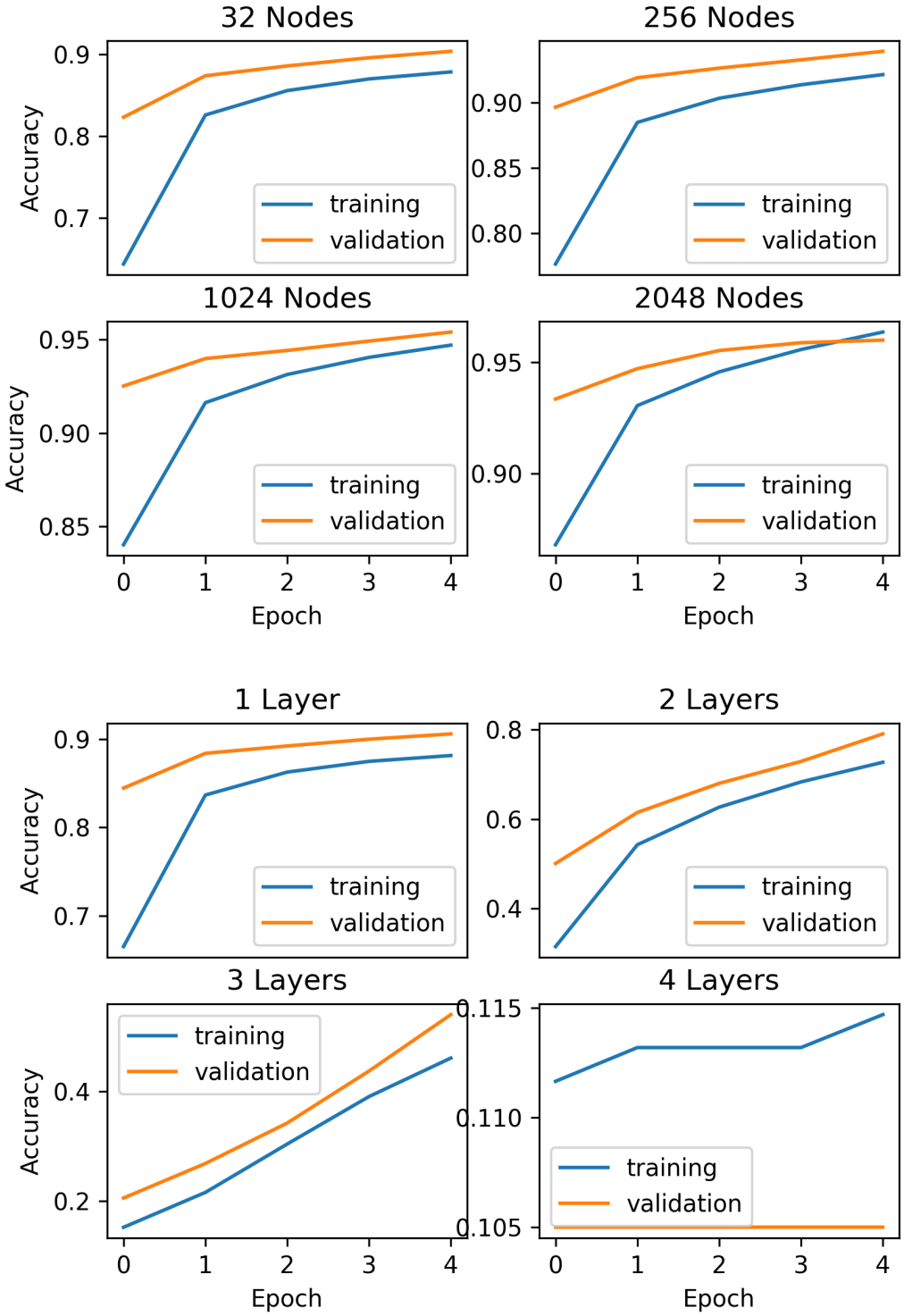


Figure 5: Total Error Function for the product per epoch

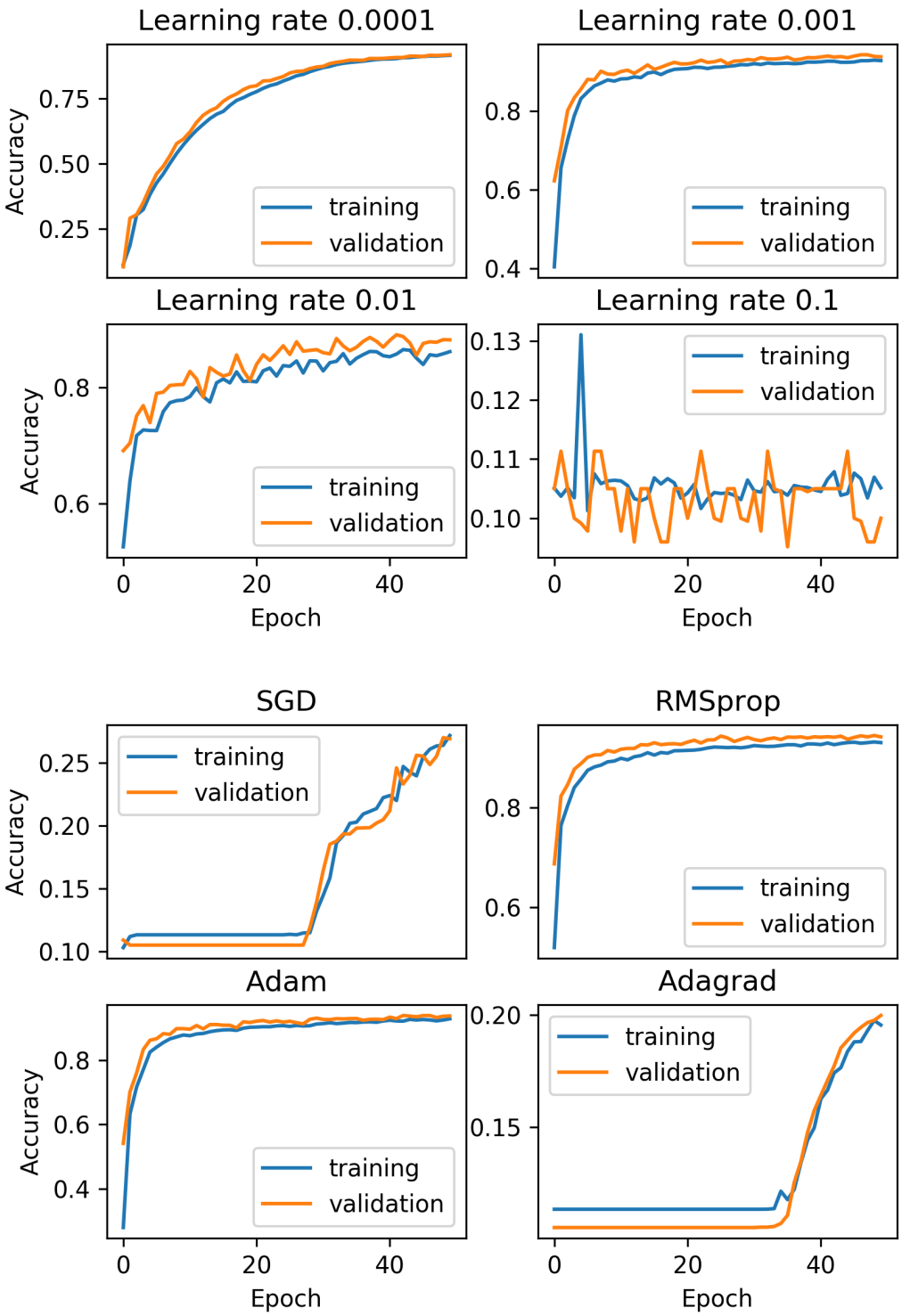


Figure 6: Varying Learning Rates and optimisation algorithm

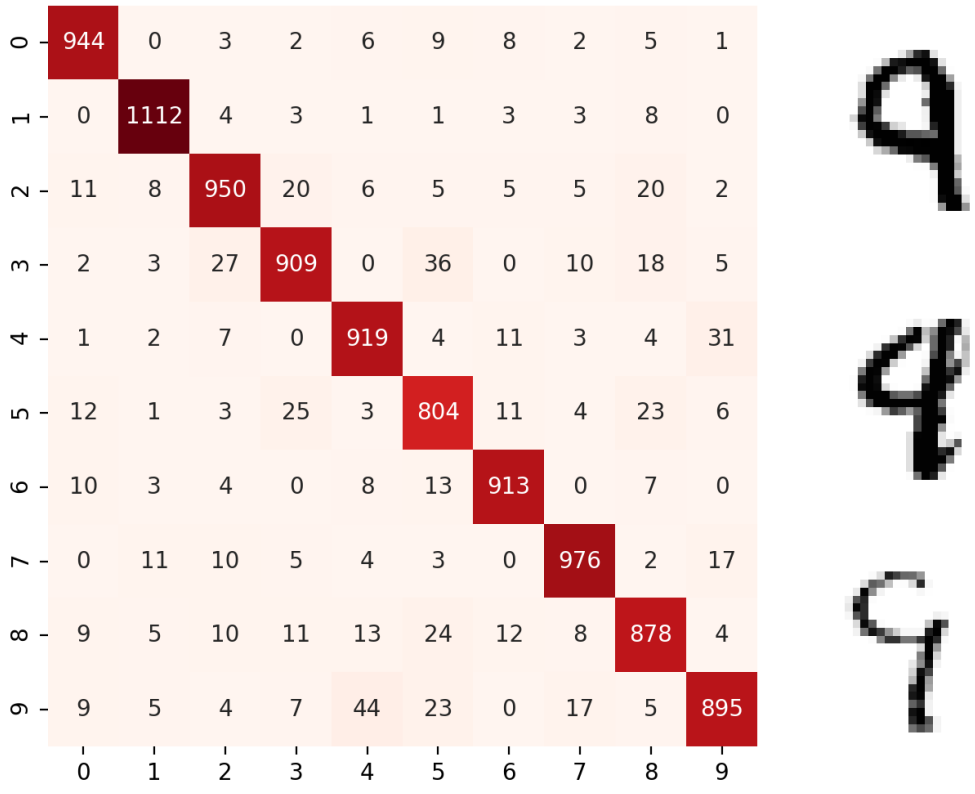


Figure 7: Confusion matrix of the final net, and 3 misidentified 9, falsly labelled as 4

4 Appendix: Code

4.1 Question 1

```
data = matrix(sample(c(-1,1), 10000, replace = TRUE), ncol = 10)

ninputs = nrow(data)

# we create two different databases, and we will apply the same algorithm to both
targets.sum = ifelse(rowSums(data) >=0,1,0)
targets.prod = ifelse(apply(data, 1, prod)>=0,1,0)

inputs = cbind( data[,1:10], rep(-1,ninputs))

## We separate our database into train and test datasets, 80/20.
split.perc = 0.8
smp_size <- floor(split.perc * ninputs)
train <- inputs[1:smp_size,]
test <- inputs[(smp_size+1):ninputs,]

epsilon = 0.01


set.seed(100)


wts = runif(11)*99 ## start with a deliberately poor set of initial weights.


array.error.train = c() # we store the training errors
array.error.test = c() # we store the test errors
for (iteration in 1:400) {

  order = sample(smp_size);
  error.train = 0;
  error.test = 0;
  for (i in (ninputs-smp_size)){
    x = inputs[i+smp_size,]
    t = targets.prod[i+smp_size]
    y = sum ( x * wts )
    y = y > 0;
    error.test = error.test + (0.5*(t - y)^2); # we just calculate error

  }
  for (i in order) {
    x = train[i,]
    t = targets.prod[i]
```

```
    y = sum ( x * wts )
    y = y > 0;
    error.train = error.train + (0.5*(t - y)^2);
    dw = epsilon * ( t - y ) * x
    wts = wts + dw; # actual learning
  }

  # title =sprintf('Iteration %d error %.3f\n', iteration, error)
  # plotdata()
  # title(main=title)
  # Sys.sleep(0.1)
  array.error.train = c(array.error.train , error.train)
  array.error.test = c(array.error.test , error.test)

}


plot(1:length(array.error.train), array.error.train, type="l",col="red")
lines(1:length(array.error.test), array.error.test, type="o",col="green")
```

4.2 Question 2

Code was done in Jupyter notebook, and is available at the following github link `https://github.com/rw4917/dl-ass1/blob/main/Part2-NN%20from%20scratch.ipynb`. The repo also contains an appendix notebook used to draw the neural net in matplotlib, adapted from a git developped by @craffel and improved by @ljhuang2017.

4.3 Question 3

Code was done in Jupyter notebook, and is available at the following github link `https://github.com/rw4917/dl-ass1/blob/main/Part3-%20MNIST%20Classification.ipynb`