TDT4265 - A2

Martin Madsen, Sondre A Bergum

February 22, 2019

1 Softmax regression with backpropagation

1.1 Backpropagation

The update rule for weight w_{ji} is defined as:

$$w_{ji} := w_{ji} - \alpha \frac{\partial C}{\partial w_{ji}} \tag{1}$$

We want to show that the gradient of the cost function satisfies the following:

$$\alpha \frac{\partial C}{\partial w_{ji}} = \alpha \delta_j x_i \tag{2}$$

We first express the gradient as a mulplication of simpler partial derivatives by applying the chain-rule, solve these partials, then combine to form our result:

$$\alpha \frac{\partial C}{\partial w_{ji}} = \alpha \sum_k \frac{\partial C}{\partial z_k} \frac{\partial z_k}{\partial a_j} \frac{\partial a_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ji}} \tag{3}$$

$$\frac{\partial C}{\partial z_k} = \delta_k \tag{4}$$

$$\frac{\partial z_k}{\partial a_j} = \frac{\partial \sum_{j'} w_{kj'} a_{j'}}{\partial a_j} = w_{kj} \tag{5}$$

$$\frac{\partial a_j}{\partial z_j} = \frac{\partial f(z_j)}{\partial z_j} = f'(z_j) \tag{6}$$

$$\frac{\partial z_j}{\partial w_{ji}} = \frac{\partial \sum_{i'=0}^{d} w_{ji'} x_{i'}}{\partial w_{ji}} = x_i \tag{7}$$

$$\Rightarrow \alpha \frac{\partial C}{\partial w_{ji}} = \alpha \delta_j x_i \tag{8}$$

$$\textit{where: } \delta_j = f'(z_j) \sum_k w_{kj} \delta_k \tag{9}$$

To summarize this gives us the two update rules for weight w_{ji} and w_{kj}:

$$w_{ji} := w_{ji} - \alpha \delta_j x_i \tag{10}$$

$$w_{kj} := w_{kj} - \alpha \delta_k a_j \tag{11}$$

For furhter explanation of notation see the accompanying assignment text.

1.2 Vectorize computation

We want to vectorize our update rule as this is more computationally efficient than nesting for-loops. In some of the following equations the expressions are augmented with dimensional information to convince the reader that dimensions are consistent, we apologize for using a somewhat sloppy notation. The following sizes are used to denote dimensions:

- I = 785 = number of pixels in a picture + bias
- J = 64 = number of nodes in hidden layer
- K = 10 = number of classes [numbers 0-9]
- N = 32 = batch size

From this we structure our matrices as the following:

- $\mathbf{X} \in R^{N\times I}$
- $\mathbf{W}_{ji} \in R^{J\times I}$
- $\mathbf{W}_{kj} \in R^{K\times J}$

We define the vectorized activation of the hidden layer as $\mathbf{a}$ and the vectorized activation of the output layer as $\mathbf{y}$. The parameter to the activation function of the hidden layer we denote $\mathbf{z}_j$ while the parameter to the activation function of the output layer we denote $\mathbf{z}_k$.

The handed out starter code defines the vectorized $\mathbf{z}$ as `"z = X.dot(W.T)"`. We have derived our vector calculations to stay consistent with this choice. This leads to the following matrix-sizes:

$$\mathbf{z}_j = \mathbf{X}\mathbf{W}_{ji}^T \qquad \dim(\mathbf{z}_j) = R^{N\times I}R^{I\times J} = R^{N\times J} \tag{12}$$
$$\mathbf{a} = f(\mathbf{z}_j) \qquad \dim(\mathbf{a}) = \dim(\mathbf{z}_j) = R^{N\times J} \tag{13}$$
$$\mathbf{z}_k = \mathbf{a}\mathbf{W}_{kj}^T \qquad \dim(\mathbf{z}_k) = R^{N\times J}R^{J\times K} = R^{N\times K} \tag{14}$$
$$\mathbf{y} = f(\mathbf{z}_k) \qquad \dim(\mathbf{y}) = \dim(\mathbf{z}_k) = R^{N\times K} \tag{15}$$

next is the vectorization of δ_k, denoted Δ_k. The vectorized form of the targets, $\mathbf{t}$, will have the same dimensions as $\mathbf{y}$.

$$\delta_k = -(t_k - y_k) \quad \Rightarrow \Delta_k = -(\mathbf{t} - \mathbf{y}) \tag{16}$$
$$\dim(\Delta_k) = R^{N\times K} \tag{17}$$

We are now ready to state the first vectorized update rule for $\mathbf{W}_{kj}$ and the dimentions of its terms:

$$\mathbf{W}_{kj} := \mathbf{W}_{kj} - \alpha\Delta_k^T\mathbf{a} \tag{18}$$
$$R^{K\times J} = R^{K\times J} - R^1 R^{K\times N} R^{N\times J} \tag{19}$$

To vectorize the update rule for $\mathbf{W}_{ji}$ we start by vectorizing δ_j and denoting it Δ_j. The first term, $f'(z_j)$ is as follows:

$$f'(z_j) = a_j(1 - a_j) \tag{20}$$
$$\Rightarrow f'(\mathbf{z}_j) = \mathbf{a} \odot (\mathbf{1} - \mathbf{a}) \tag{21}$$
$$\dim(f'(\mathbf{z}_j)) = R^{N\times J} \odot (R^{N\times J} - R^{N\times J}) = R^{N\times J} \tag{22}$$

In the above equation $\odot$ is the Hadamard product, and **1** is a matrix filled with ones. The next factor to be vectorized is $\sum_k w_{kj}\delta_k$:

$$\sum_k w_{kj}\delta_k \Rightarrow \Delta_k \mathbf{W}_{kj} \tag{23}$$

$$\dim(\Delta_k \mathbf{W}_{kj}) = R^{N\times K} R^{K\times J} = R^{N\times J} \tag{24}$$

We can now form the expression for Δ_j:

$$\delta_j = f'(z_j)\sum_k w_{kj}\delta_k \tag{25}$$

$$\Rightarrow \Delta_j = f'(\mathbf{z}_j) \odot \Delta_k \mathbf{W}_{kj} \tag{26}$$

$$\dim(\Delta_j) = R^{N\times J} \odot R^{N\times J} = R^{N\times J} \tag{27}$$

This was the last piece of the puzzle and we are now ready to state the last update rule in vector form:

$$\mathbf{W}_{ji} := \mathbf{W}_{ji} - \alpha \Delta_j^T \mathbf{X} \tag{28}$$

$$R^{J\times I} := R^{J\times I} - R^1 R^{J\times N} R^{N\times I} = R^{J\times I} \tag{29}$$

To summarize; the vectorized update rules for the weights are expressed in equations 28 and 18.

2 MNIST Classification

2.1 a)

After loading the training and test sets we go on to normalize the pixels in the pictures so they take on values in the range [-1,1] instead of [0,255]. Then we perform the bias trick: appending a 1 to each sample, and change the labels to use one-hot encoding. We take the last 10% of the samples and split it out as a validation set. As a default we're using a batch size of 32 for batch training, learning rate of 0.5 and a maximum of 15 epochs. We're using cross entropy loss to calculate our loss and early stopping if the validation loss has increased consistently (typically 3 checks in a row). We check for this 10 times per epoch. We don't use any kind of regularization and normalize the loss by taking the average over all output values over the whole batch. We normalize the gradients over each batch by the same factor of normalization (batch size $\times$ classes $= 32 \cdot 10 = 320$ with the hyper-parameters stated above).

Our network consists of an input layer with 785 neurons, a hidden layer with 64 neurons and an output layer with 10 neurons. This gives us $785 \cdot 64 + 64 \cdot 10 = 50880$ parameters. We use the sigmoid function activation of the hidden layer and softmax for activation of the output layer.

2.2 b)

We implemented a numerical approximation to the gradient calculated in our gradient-descent function to verify our implementation. Below are a table listing the maximum absolute difference for weight w_{ji} and w_{kj} over the ten first training iterations of a untrained network. We use the same hyper-parameters as above. The gradient approximation used is the same as in the assignment text and our epsilon is $\epsilon = 0.01$ as suggested. Since our error is well within $\mathbf{O}(\epsilon^2)$ we conclude that our gradient implementations are correct.

Iteration No	Max error $\mathbf{W}_{ji}$	Max error $\mathbf{w}_{kj}$
1	2.09e-8	8.29e-8
2	2.37e-8	8.54e-8
3	3.22e-8	8.37e-8
4	2.28e-8	8.55e-8
5	1.77e-8	8.14e-8
6	2.84e-8	8.96e-8
7	2.52e-8	8.58e-8
8	1.99e-8	8.87e-8
9	2.32e-8	8.63e-8
10	2.04e-8	8.80e-08

Table 1: Maximum gradient error during the first 10 training iterations for task 2.b

2.3 c)

The training results for our net with the hyperparameters above are summarized in this section. The training, testing and validation losses during training are seen in figure 1, and the accuracy is seen in 2. The x-axis is training iterations for both figures. Our final validation loss and accuracy is summarized in table 2.

Last Loss	Last Accuracy
0.0259	0.924

Table 2: The last validation loss and validation accuracy after training in task 2.c

3 Adding the Tricks of the Trade

In this section we introduce some tricks to improve our network. The tricks are turned on one after another and the final validation accuracy and loss are presented for each added trick. In the end we summarize the our accuracies and losses for the whole training loop with all tricks turned on to compare with the plots from the previous section. For all of these ”tricks” we added a boolean variable to toggle the effect on and off in our code.

3.1 a)

Here we shuffle the training set for each epoch. This leads to different batch configuration between each epoch which should improve learning. Implemented as a function `shuffle_training_set()` in the code. The changes are summarized in table 3, we note that they are comparable to without shuffling. However it does impact learning rate as without shuffling the network takes about 7000 training iterations to reach an accuracy of 90%, while with shuffling it takes 6000 training iterations which is a slight improvement.

Last Loss	Last Accuracy
0.0273	0.920

Table 3: The last validation loss and validation accuracy after turning on shuffling

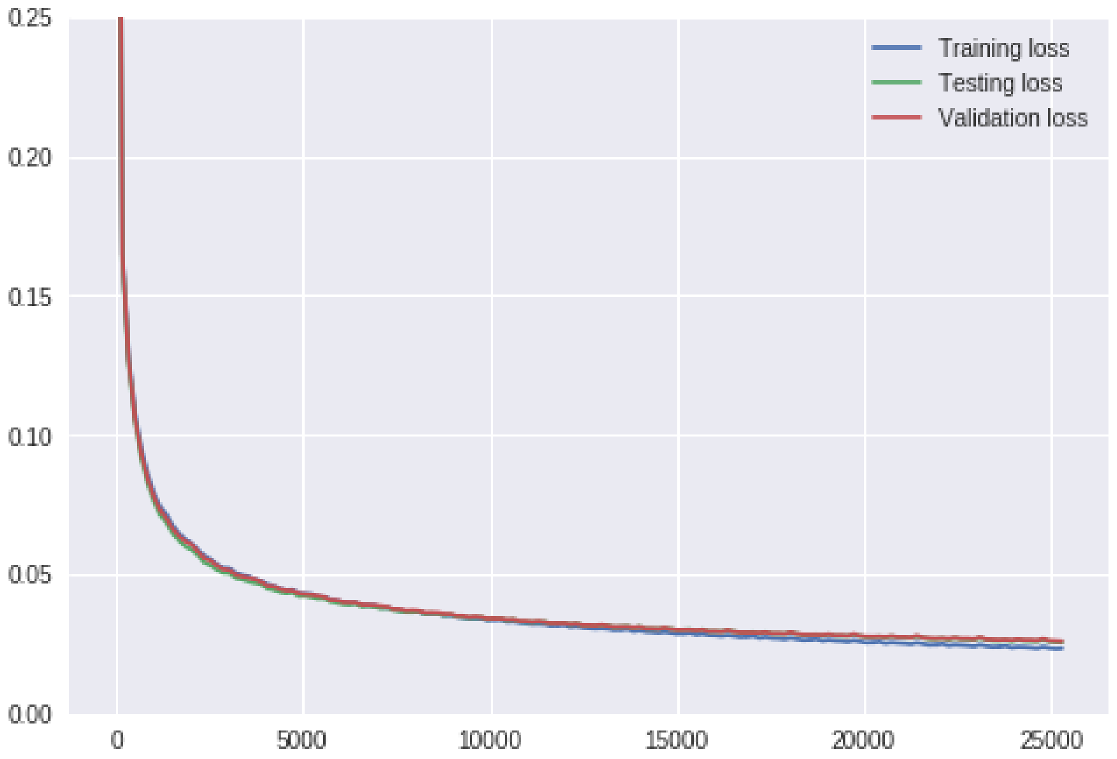

Figure 1: Losses for task 2.c

3.2 b)

We replace the sigmoid function with the improved sigmoid described in equation 30. We simply added to the `sigmoid(z)` function implementation. The results are summarized in table 4. We see that our final accuracy has decreased, this can be explained by our initial weights as they are uniformly distributed between $[0, 1]$ meaning that some weights are close to ± 1 where the improved sigmoid function perform badly. It also triggers the early stopping algortihm after about 7000 training iterations which shortens the training time.

$$f(z) = 1.7159tanh(\frac{2}{3}z) \tag{30}$$

Last Loss	Last Accuracy
0.0407	0.879

Table 4: The last validation loss and validation accuracy after training after turning on the improved sigmoid function

3.3 c)

We now initialize our weights as a normal distribution centered around 0 with a standard deviation of $\sigma = 1/\sqrt{785}$ for weight $w_j i$ and $\sigma = 1/\sqrt{64}$ for weight w_{kj}. The final validation loss and accuracy are summarized in table 7. This normalization enables the improved sigmoid to shine and our results are better than both the case with no tricks and when we add shuffling. It is implemented as the function `weight_initialization_normal_dist(input_units, output_units)`. It triggered the early stop algorithm after about 3750 training iterations, further shortening our training times and improving learning rate.

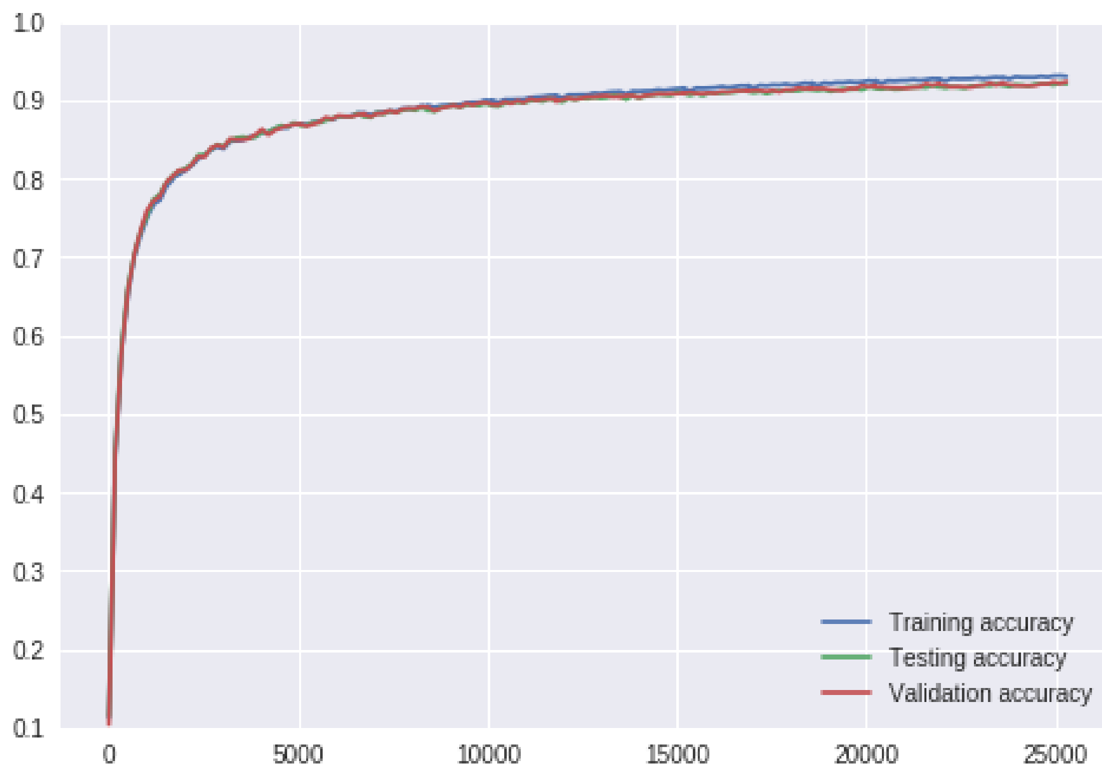

Figure 2: The accuracy of task 2.c

Last Loss	Last Accuracy
0.0193	0.942

Table 5: The last validation loss and validation accuracy after training after turning on normal distributed initial weights.

3.4 d)

Now we implement the last trick, momentum. We chose the standard momentum algorithm described in equation 31 and not the Nesterov momentum. This changes our update rule for each weight and is implemented as a change in our gradient descent function, `gradient_descent(...)`. The best accuracies and losses for the validation set obtained with this trick added is summarized in table 6. The momentum coefficient we used was $\mu = 0.9$. Note that we had to lower our learning rate to $\alpha = 0.1$ to compensate for the momentum. This is gave us the best results for this network topology. The figures 3 and 4 show the losses and accuracies during training. One important observation is that we have greatly improved our initial training as we are able to climb to an accuracy of about 90% in just a couple hundred training iterations. We do however not trigger early stopping until the 10th epoch. Also, the fact that early stopping triggered a lot later than usual in this exact training run gave us exceptionally good results. Usually early stopping would trigger earlier when we tested our code, but we chose to include this result as it shows the potential of our network. Further tweaking to the early stopping algorithm could lead to more consistent results.

$$v := \mu v - \alpha \frac{C}{\partial w} \tag{31}$$

$$w := w + v \tag{32}$$

Last Loss	Last Accuracy
0.0107	0.970

Table 6: The last validation loss and validation accuracy after training after turning on momentum.

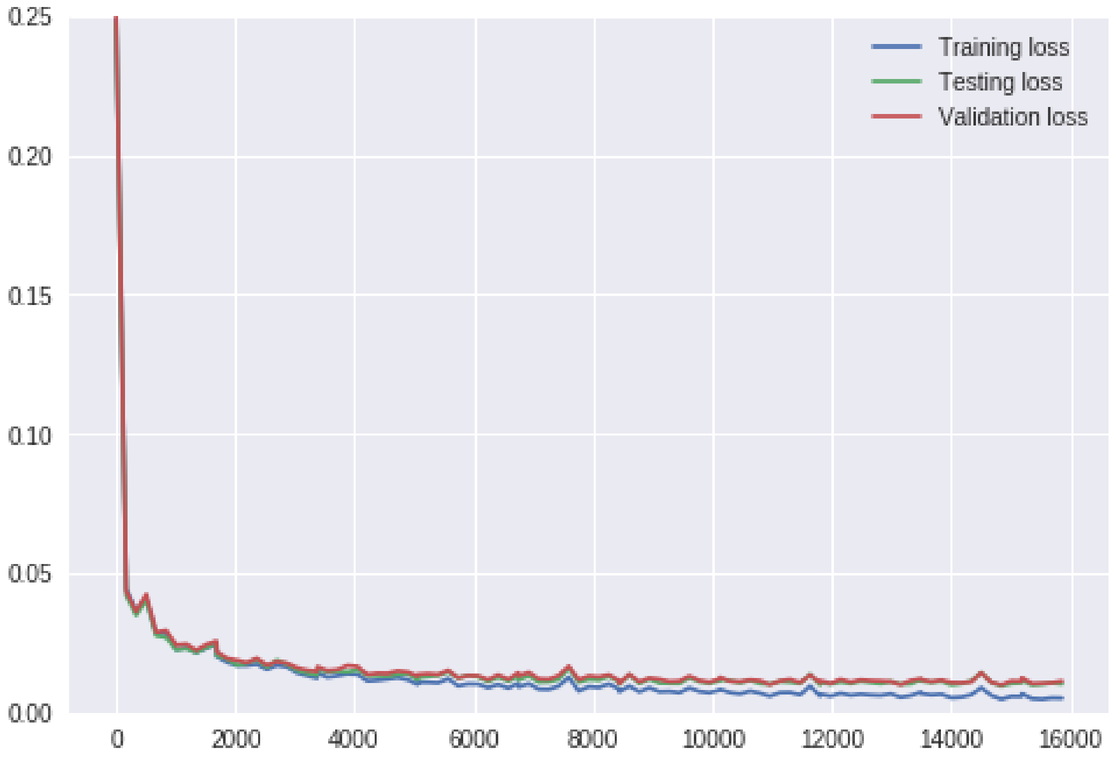

Figure 3: Losses for task 3.d

4 Experiment with network topology

4.1 a)

As we decrease the number of hidden units the performance drops consistently, but it also learns the task in fewer training iterations. When the number of hidden units equals the number of output neurons, the accuracy has dropped to about 90%. As we decrease the number of hidden units further, the performance plummets and with only 1 hidden unit the nework has an accuracy of about 20%. The fact that the performance plummets when the number of hidden neurons is reduced below the number of output neurons is very intuitive. At this point the hidden layer is limiting the networks capabilities rather than improving it.

4.2 b)

As we increase the number of hidden neurons, the learning speed decreases, but the performance increases slightly up to a point, it seems. If we increase too much the performance decreases again and the network gets over trained very quickly. Lowering the learning rate limits this problem, but then it takes very long to actually train the network to the same or even lower performance than with a lower number of hidden neurons. Not in term of more training iterations, but the iterations themselves takes a lot longer because of the increase in computational cost.

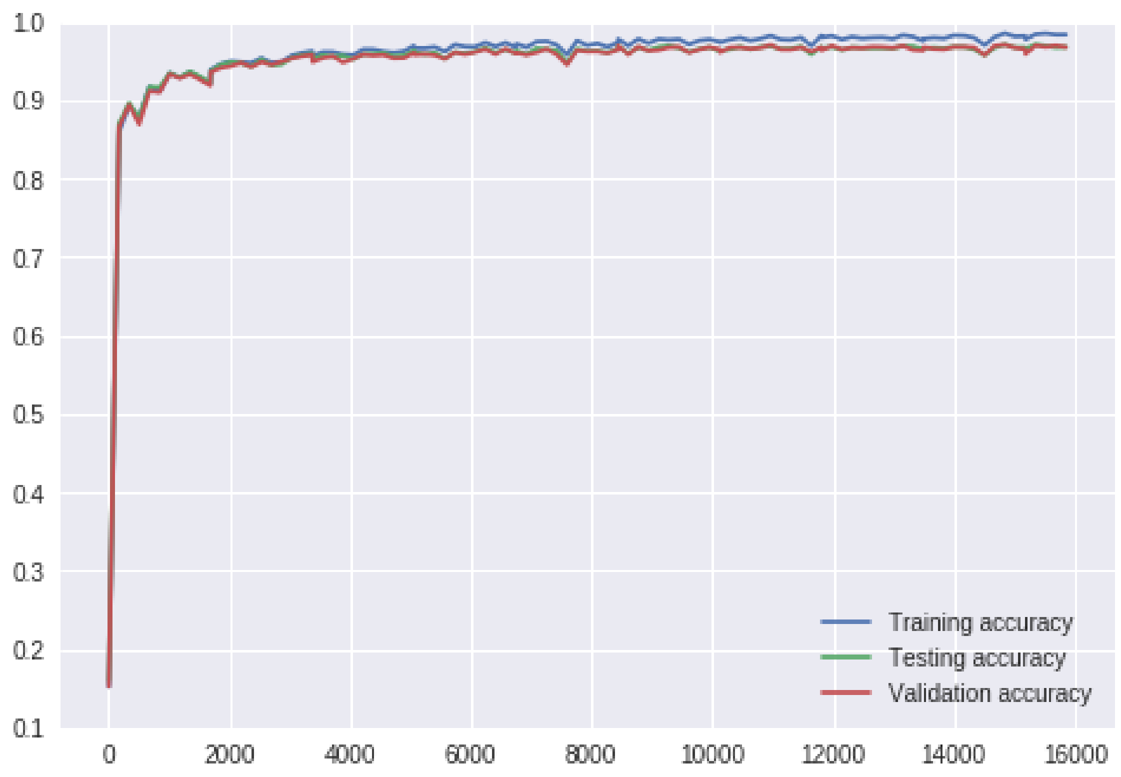

Figure 4: The accuracy of task 3.d

4.3 c)

The network used up until now in this assignment consists of an input layer with 785 neurons, a hidden layer with 64 neurons and an output layer with 10 neurons. This gives us $64 \cdot 785 + 10 \cdot 64 = 50880$ parameters. If we are to add another hidden layer, keep the number neurons in both hidden layers equal, and achieve about the same amount of parameters, we found that 59 neurons in each layer is the best fit. This gives us $785 \cdot 59 + 59 \cdot 59 + 59 \cdot 10 = 50386$ parameters.

Best Loss	Best Accuracy
0.0111	0.970

Table 7: Best loss and accuracy for the validation set with 2 hidden layers

Results with an extra layer can be found in table 7 and fig. 5 - 6, and were achieved with early stopping triggered after 5 consecutive increases to the loss. We see the results are comparable to the final network from task 3.

Because of the changes needed to be made to several of the functions in our code to run it with an additional layer, the implementation for this is found in a separate jupyter notebook.

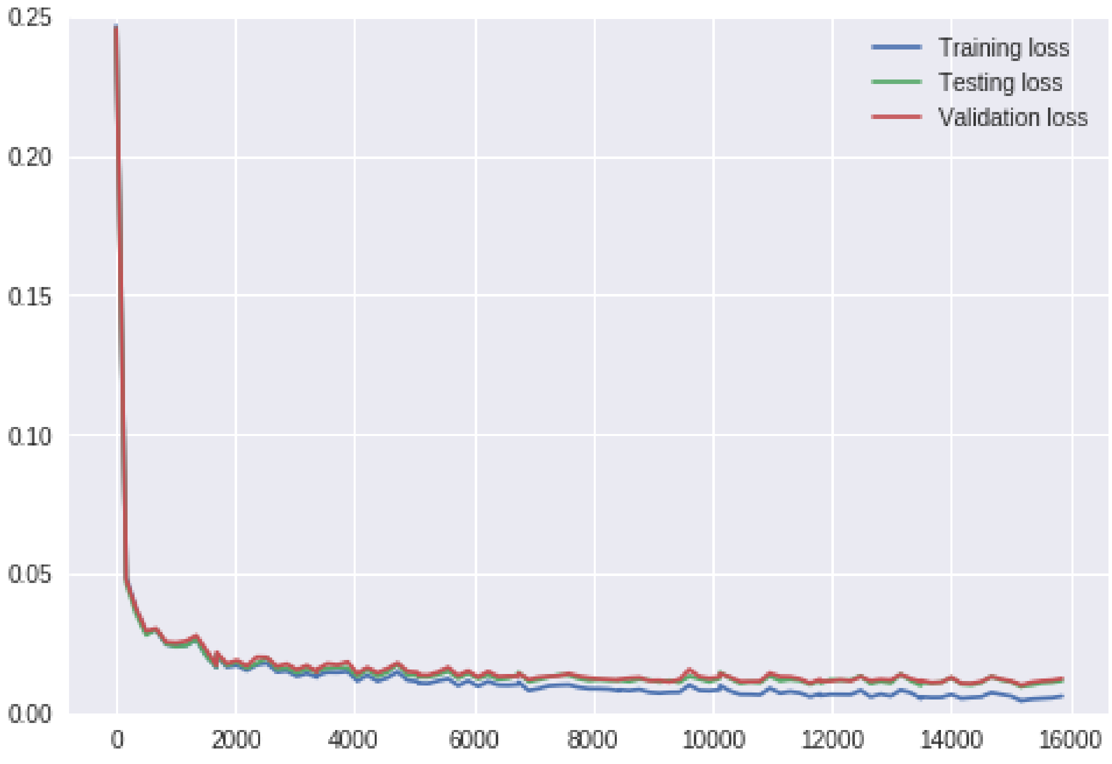

Figure 5: Losses for task 4.c

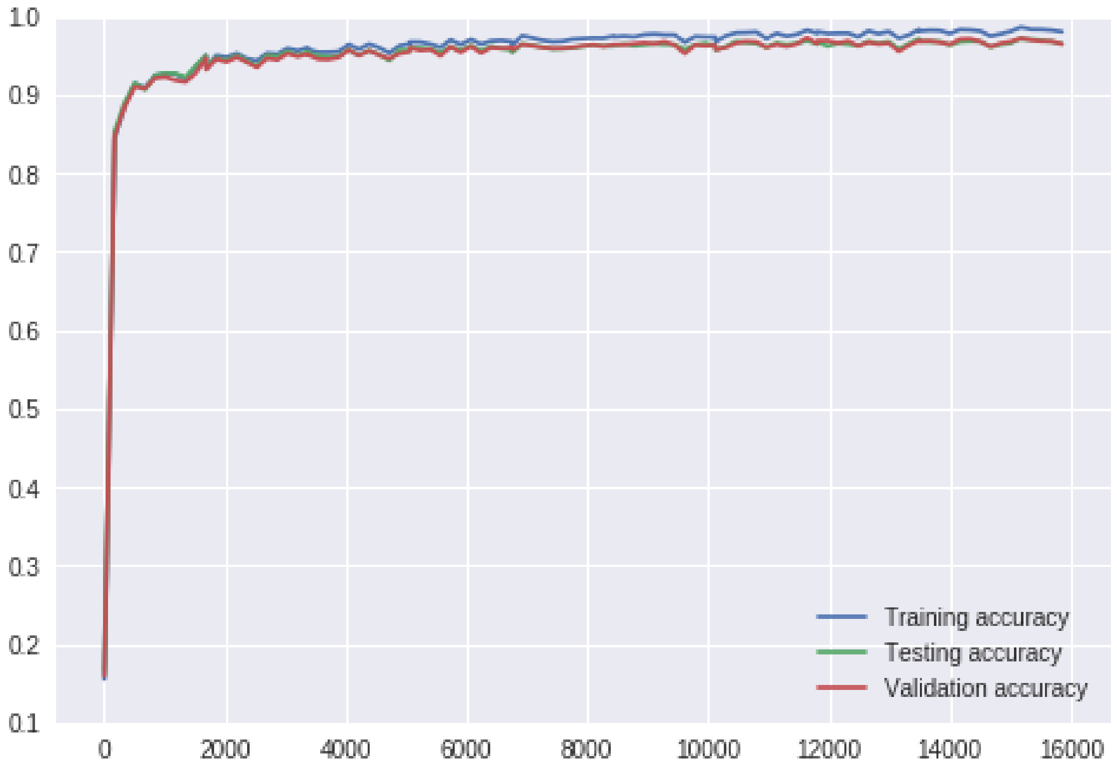

Figure 6: Accuracy for task 4.c