TDT4265 Computer Vision and Deep Learning

Assignment 2

Written February 2019 By
Fredrik Veidahl Aaagaard

1: Softmax regression with backpropagation

Task 1: Backpropagation

We want to find the gradient used in the backpropagation algorithm

$$\begin{aligned}
\alpha\frac{\partial C}{\partial w_{ij}} &= \alpha a_i\delta_j = \alpha\frac{\partial C}{\partial z_j}x_i \\
&= \alpha(\sum_k \frac{\partial C}{\partial z_k}\frac{\partial z_k}{\partial z_j})x_i \\
&= \alpha(\sum_k \delta_k\frac{\partial z_k}{\partial z_j})x_i \\
&= \alpha(\sum_k \delta_k\frac{\sum(\partial w_{kj}a_j)}{\partial z_j})x_i \\
&= \alpha(\sum_k \delta_k\frac{\sum(\partial w_{kj}f(z_j))}{\partial z_j})x_i \\
&= \alpha(\sum_k \delta_k w_{kj}f'(z_j))x_i \\
&= \alpha(f'(z_j)\sum_k \delta_k w_{kj})x_i \\
&= \alpha\delta_j x_i \quad \text{where} \quad \delta_j = f'(z_j)\sum_k \delta_k w_{kj}
\end{aligned} \tag{1}$$

Task 2: Vectorizing computation

Update rules using matrix notation for:

Hidden layer to output layer:

$$\mathbf{W_{kj}} = \mathbf{W_{kj}} - \alpha\delta_{\mathbf{k}}\mathbf{a_j}^\top \quad \text{where} \quad \delta_{\mathbf{k}} = -(\mathbf{t_k} - \mathbf{y_k}) \tag{2}$$

Input layer to hidden layer:

$$\mathbf{W_{ji}} = \mathbf{W_{ji}} - \alpha\delta_{\mathbf{j}}\mathbf{x_i}^\top \quad \text{where} \quad \delta_{\mathbf{j}} = (\mathbf{W_{kj}^\top}\delta_{\mathbf{k}}) \odot \mathbf{f'(z_j)} \tag{3}$$

2: MNIST classification

Note: I forgot to label the axes on the graphs, but the y-axis should be loss and accuracy (depending on the plot), and the x-axis the iterations.

a)

A layered neural network with one hidden layer with 64 neurons was used in the training procedure. A sigmoid activation function was used for the hidden layer, while a softmax activation function was used for the output layer. 10% of the training set was used for validation, resulting in 54000 images in the training set, 6000 images in the vaildation set and 10000 images in the test set.

The weights were initialized uniformly s.t. $W_k, W_{ji} \in [-1, 1]$. Hyperparameters was chosen according to Table 1.

batch size	learning rate	max epochs
128	0.5	20

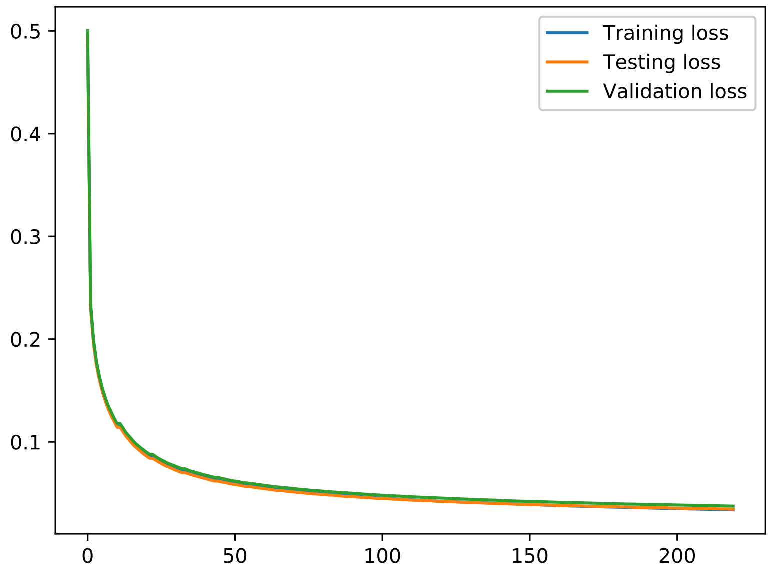
Table 1: Hyperparameters

b)

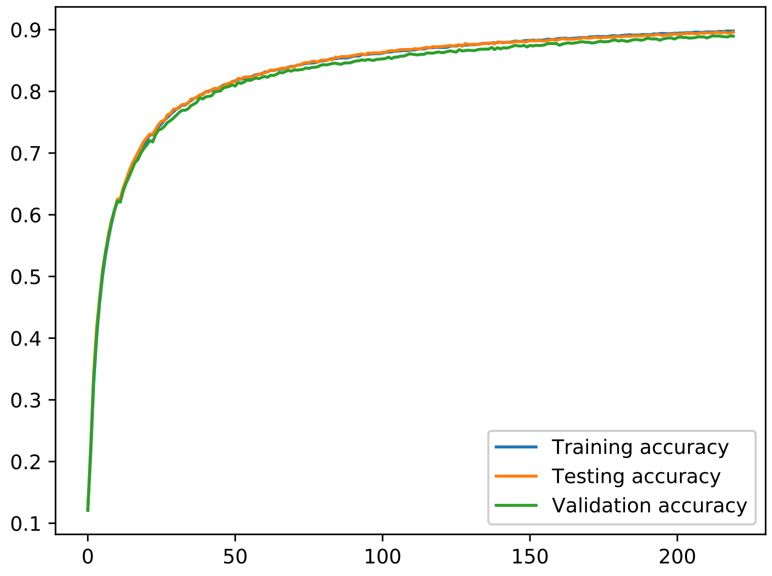
I tried implementing numerical approximation, but it didn't seem to work. I got a maximum error of 0.00245 before the program stopped since the maximum absolute difference was too big. I used the given $\epsilon = 10^{-2}$. Since I got about 90% accuracy and about 0.04 loss, I think my implementation of the gradient check might be wrong.

c)

We see from Figure 1 that without any tricks of the trade we get pretty good results. Using 64 hidden units and the hyperparameters given in Table 1 we get $\sim 89\%$ accuracy on the validation set.



(a) Loss



(b) Accuracy

Figure 1: Loss and accuracy of the sets without any tricks of the trade

3: Adding the "Tricks of the Trade"

When implementing these tricks of the trade I have chosen to let the hypervariables be the same as in Section 2, but I have chosen a batch size of 32 instead as this results in $\sim 92\%$ accuracy before implementing any tricks of the trade. I have then chosen to let these hypervariables be the same throughout this task as this makes it easier to see the direct impact of the different tricks. The following results (aswell as figures and Table 2) comes from stacking the tricks on top of eachother, and not testing them seperately.

a)

Shuffling training examples didn't seem to improve the accuracy (if at all). When running the program a couple of times I found it was generally faster, although this seems to wary quite alot. When it ran through more epochs it got better results (about 93% validation accuracy and 0.02 loss), while it sometimes early stopped after about 7 epochs with worse results (about 90% validation accuracy and 0.04 loss)

b)

The improved sigmoid function in Section 4.4 (LeCun et al., 2012) is $f(z) = 1.7159 \cdot \tanh\left(\frac{2}{3}z\right)$. To implement this we also need to find its derivative, which simply is $f'(z) = 1.7159 \cdot \frac{2}{3}(1 - \tanh^2(\frac{2}{3}z))$.

Implemeting the improved sigmoid function seems to dramatically increase training speed, but with the given hyperparameters we still only get about 92% validation accuracy.

c)

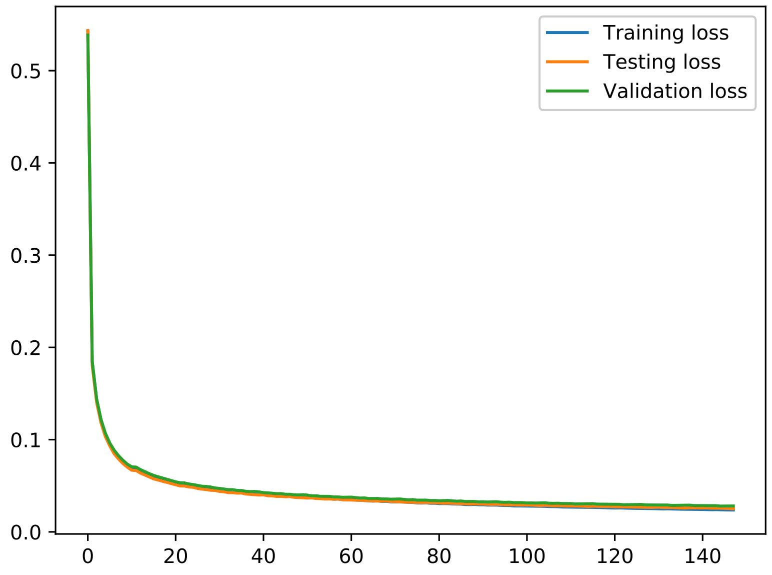
Initializing weigths from a normal distribution also improved the training dramatically. This time the training procedure wasn't only slightly faster, but we got a validation accuracy of about 96%! However, we see from Figure 4 that the graphs are not as smooth as they were.

d)

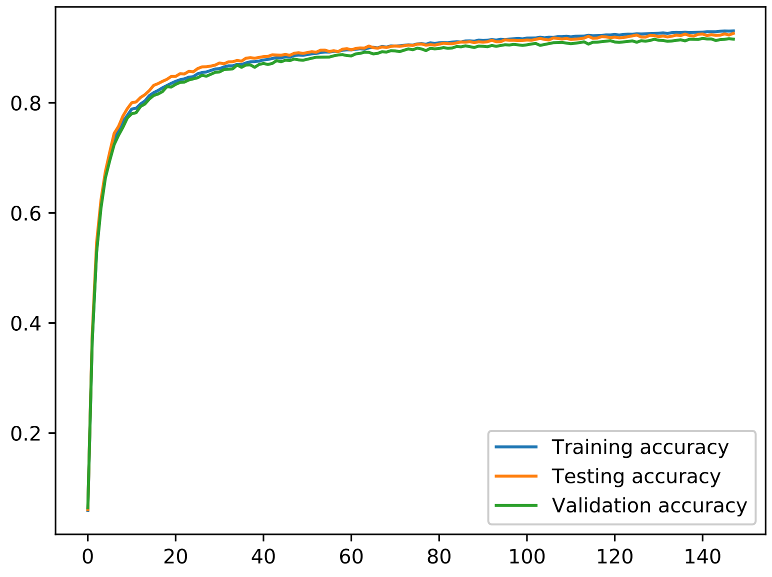
Adding momentum (classical, not Nesterov's) speeds up the training even more. The training can now become really fast, often early stopping after 1 or 2 epochs. This makes sense as the training speeds up as it goes on. We can see from Figure 5 that the graphs are now rather "jumpy" and not as smooth as they were, but this might be because we now have superfast training.

	No tricks	Shuffle	Improved sigmoid	Normal weights	Momentum
Epochs	20	10	5	4	2
Loss	0.03	0.03	0.03	0.02	0.02
Accuracy	92%	91%	91%	96%	94%

Table 2: Summary of the effects of the tricks.

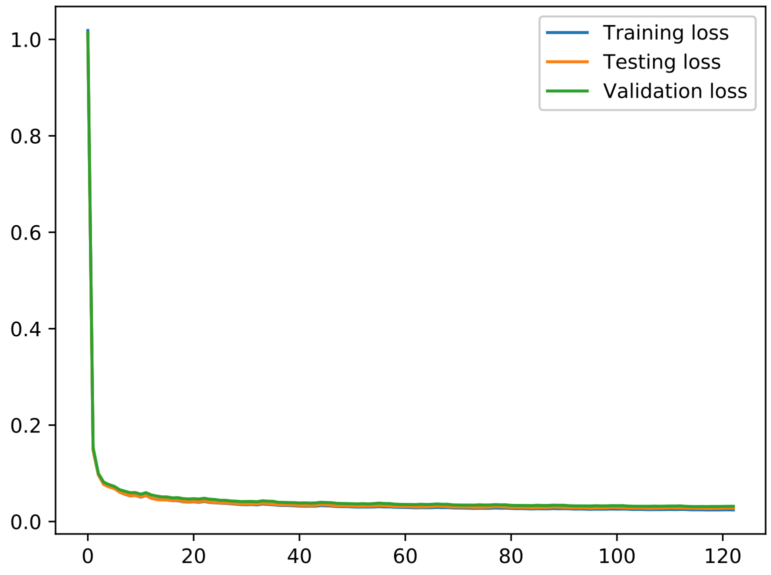


(a) Loss

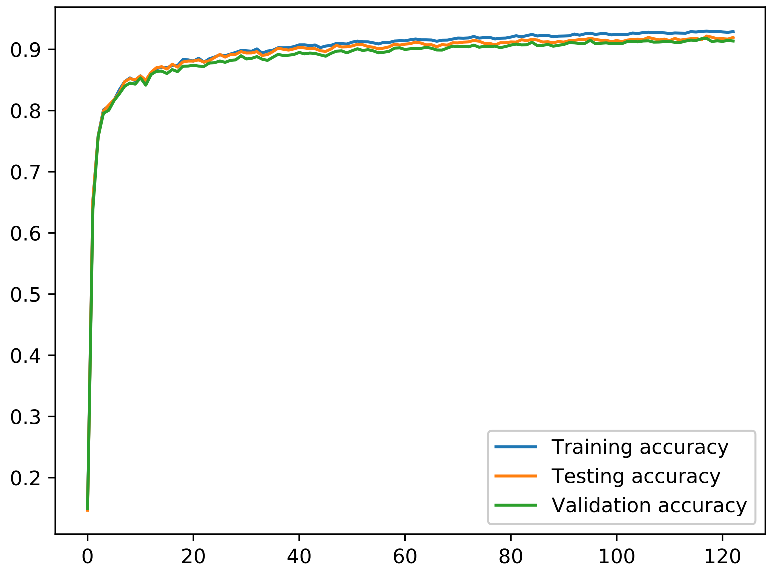


(b) Accuracy

Figure 2: Loss and accuracy of the sets with shuffling

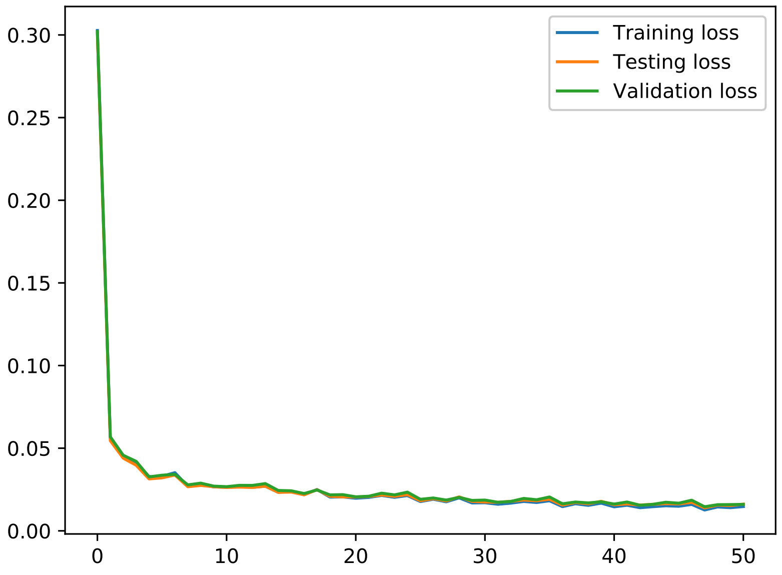


(a) Loss

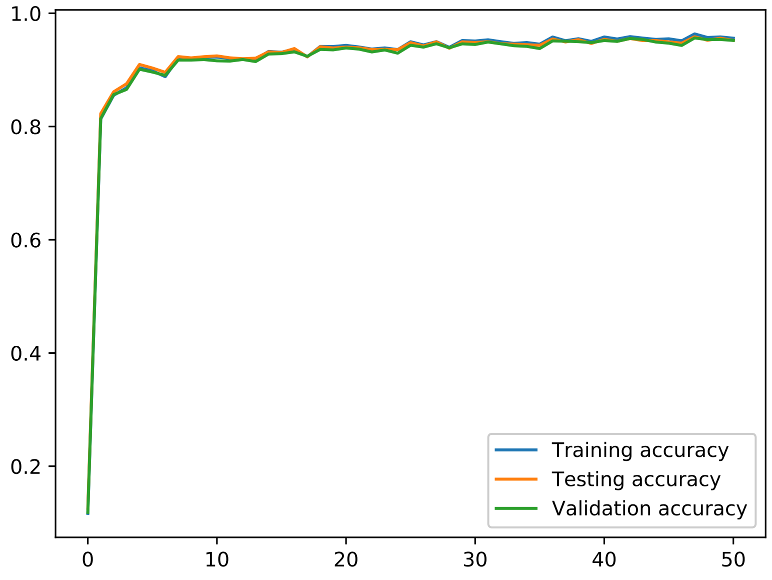


(b) Accuracy

Figure 3: Loss and accuracy of the sets with imroved sigmoid and the previous tricks

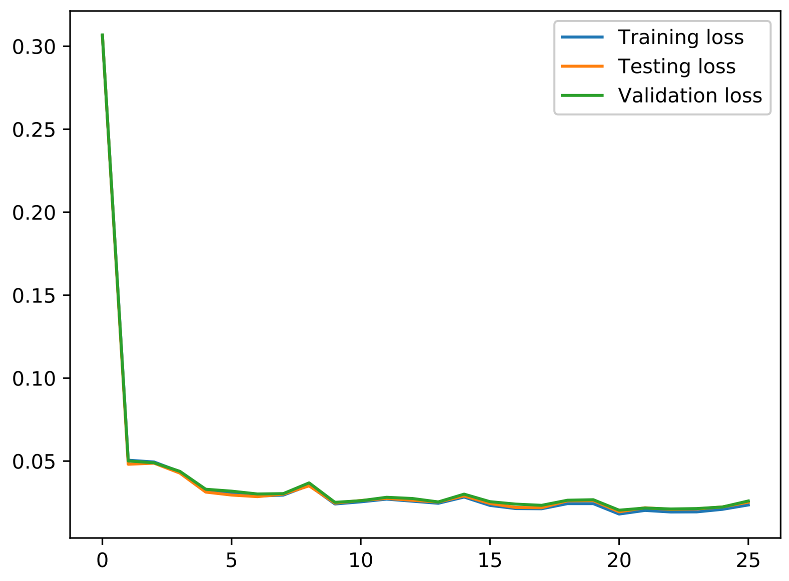


(a) Loss

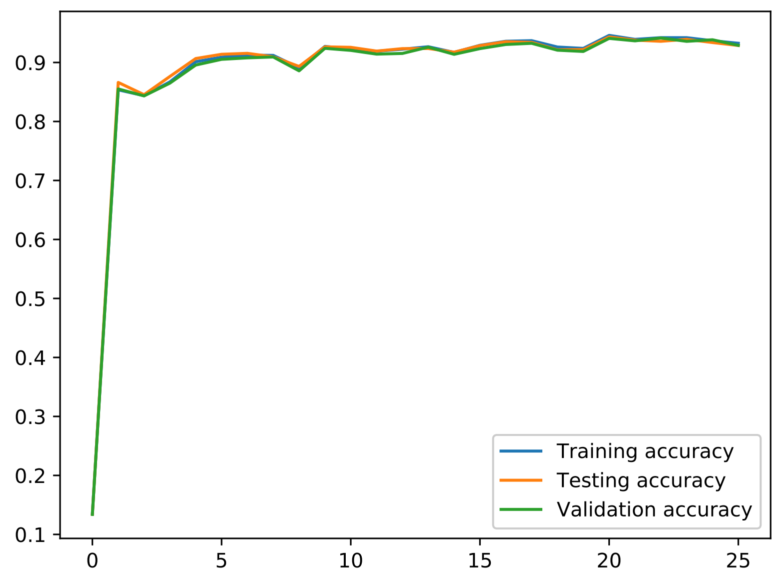


(b) Accuracy

Figure 4: Loss and accuracy of the sets with weights initialized from a normal distribution and the previous tricks



(a) Loss



(b) Accuracy

Figure 5: Loss and accuracy of the sets with weights initialized from a normal distribution and the previous tricks

4: Experiment with network topology

I have kept all the tricks from Task 3. (Assignment says to "Start with your final network from Task 4", but I assume this is just a typing error.)

a)

By halving the number of neurons in the hidden layer (from 64 to 32), training an epoch is faster. but the results seems to be slightly worse. That the program runs through an epoch faster makes sense as there are alot less weights that needs to be updated.

Halving again (to 16 neorons) made the results worse, as the training was now stagnating at about 90% accuracy. Halving yet again (to 8 neurons) the training was stagnating at about 80%.

b)

By doubling the number of neurons in the hidden layer (from 64 to 128), training an epoch is slower, but the results seems to be as good as with 64. Again it makes sense that the program runs through an epoch slower as there are more weight that needs to be updated.

I tried doubling yet again (to 256), and this made the epoch incredibly slow. Table 3 summarizes the iterations per second for each of the different numbers of neurons tested.

Neurons	256	128	64	32	16	8
it/s	90	180	380	550	750	900

Table 3: Approximate iterations per second for different sizes of hidden layer

c)

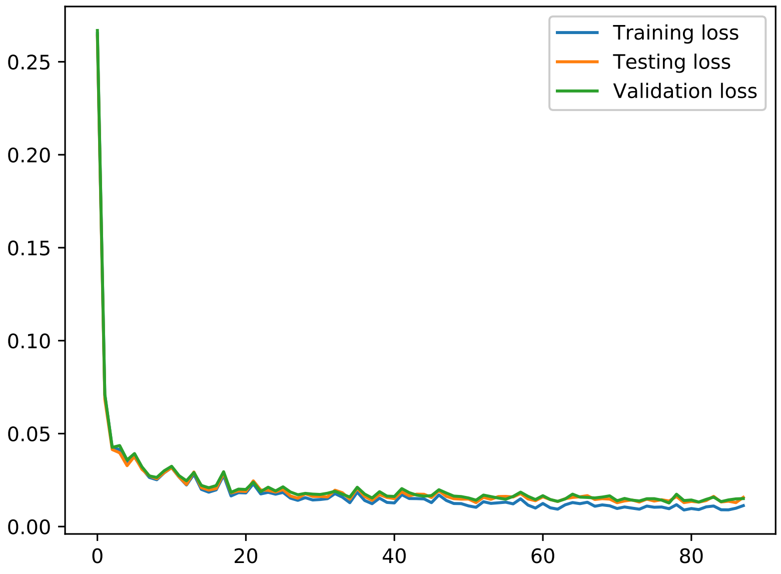
When implementing more hidden layers I chose to not do it in an object oriented fashion, as this would be much more work. Although, if I were to implement more than just one layer I would definitely do it that way.

In the previous task we have 784 neurons in the input layer, 64 in the hidden layer, and 10 in the output layer. This gives $64 \cdot 784 + 10 \cdot 64 = 50816$ weights and $64 + 10 = 74$ biases in the network resulting in a total of 50890 parameters. When implementing a second hidden layer the number of neurons in the input- and output layer stays the same. We calculate the number of neurons in each of the two layers, such that we have approximately as many parameters as in the first network, to be:

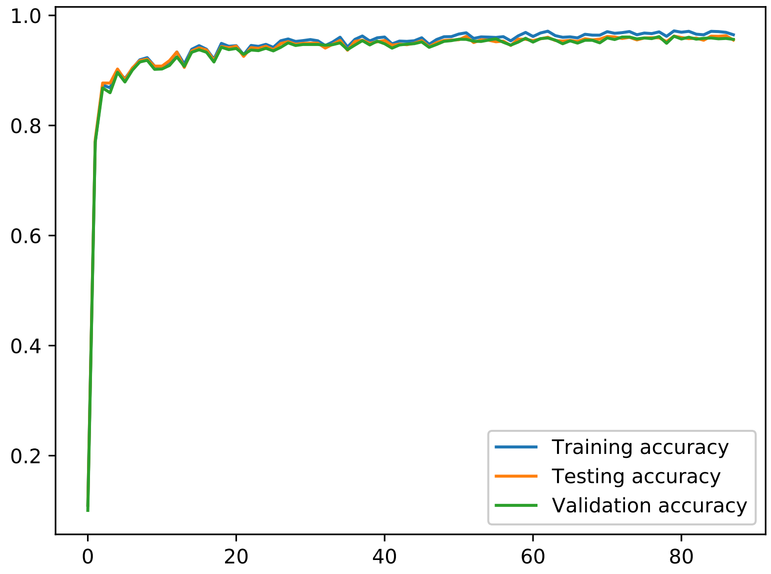
$$\begin{aligned}
(784x + x) + (x^2 + x) + (10x + 10) &= 50890 \\
x^2 + 796x - 50880 &= 0 \\
\text{Neurons} &= \frac{-796 \pm \sqrt{796^2 + 4 \cdot 50880}}{2} \\
\text{Neurons} &\approx 59.48
\end{aligned} \tag{4}$$

I settled on using 59 neurons in each of the hidden layers, resulting in a total of $(59 \cdot 784 + 59) + (59 \cdot 59 + 59) + (10 \cdot 59 + 10) = 50455$ parameters.

Using two hidden layers seems to have given a boost in performance, although it trained a little slower. The validation accuracy reached 96% again, and the validation loss reached 0.01. We see that the graph is also a little smoother (this may also just be that there are more iterations before early stopping).



(a) Loss



(b) Accuracy

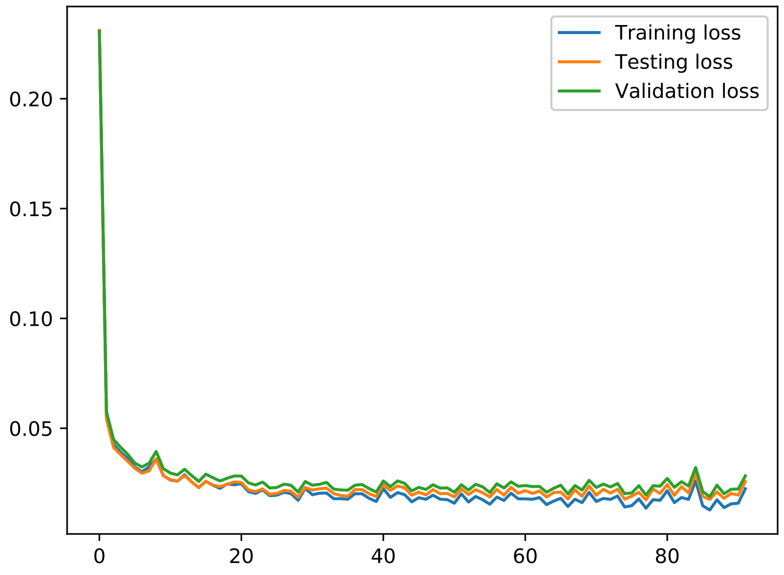
Figure 6: Loss and accuracy of the sets with all the tricks of the trade and 2 hidden layers

5: Bonus - ReLU implementation

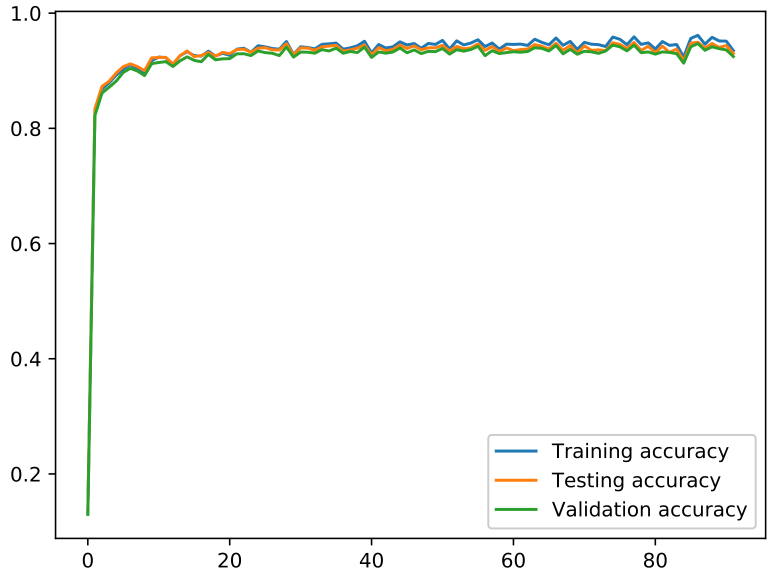
To further experiment with the network I chose to implement a ReLU activation function for the hidden layers. The ReLU function and its derivative is given in equation (5).

Figure 7 shows the loss and accuracy using ReLU as activation function. The performance of the network didn't really improve, as it converged to 93% validation accuracy and 0.02 validation loss. The convergence rate might have been slightly improved, although it is hard to tell.

$$f(z) = \begin{cases} z & z > 0 \\ 0 & z \leq 0 \end{cases} \quad , \quad f'(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases} \tag{5}$$



(a) Loss



(b) Accuracy

Figure 7: Loss and accuracy of the sets with all the tricks of the trade and a ReLU activation function for the hidden layers