NTNU
Norwegian University of
Science and Technology

TDT4265 - Computer Vision and Deep Learning

# Assignment 2

Jakob Løver
Einar Henriksen

February 14, 2019

# Contents

# 1 Softmax regression with backpropagation

## 1.1 Task 1.1 Backpropagation

We want to derive the update rule for $w_{ji}$:

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

This means we have to calculate $\frac{\partial C}{\partial w_{ji}}$:

$$\frac{\partial C}{\partial w_{ji}} = \sum_k \frac{\partial C}{\partial z_k} \frac{\partial z_k}{\partial a_j} \frac{\partial a_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ji}} \tag{2}$$

$$\frac{\partial C}{\partial z_k} = \delta_k \tag{3}$$

$$\frac{\partial z_k}{\partial a_j} = \frac{\partial}{\partial a_j} \sum_j w_{kj} a_j = w_{kj} \tag{4}$$

$$\frac{\partial a_j}{\partial z_j} = \frac{\partial}{\partial z_j} f(z_j) = f'(z_j) \tag{5}$$

$$\frac{\partial z_j}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} \sum_i w_{ji} x_i = x_i \tag{6}$$

$$\frac{\partial C}{\partial w_{ji}} = \delta_j x_i \tag{7}$$

$$\text{where } \delta_j = f'(z_j) \sum_k w_{kj} \delta_k \tag{8}$$

## 1.2 Task 1.2 Vectorize computation

$$w_{kj} = w_{kj} - \alpha \frac{\partial C}{\partial w_{kj}} = w_{kj} - \alpha \delta_k a_j \tag{9}$$

can be rewritten in matrix notation as:

$$\mathbf{W}_{kj} = \mathbf{W}_{kj} - \alpha \boldsymbol{\delta}_k \mathbf{a}_j^\mathsf{T} \tag{10}$$

Now we want to look at

$$w_{ji} = w_{ji} - \alpha f'(z_j) \sum_k w_{kj} \delta_k x_i \tag{11}$$

and rewrite it on vector form:

$$\delta_j = f'(z_j) \sum_k w_{kj} \delta_k \tag{12}$$

$$\boldsymbol{\delta}_j = \mathbf{f}'(\mathbf{z}_j) \circ \boldsymbol{\Sigma}_j \tag{13}$$

$$= \mathbf{F}'(\mathbf{z}_j) \boldsymbol{\Sigma}_j \tag{14}$$

Where $\mathbf{z}_j = \mathbf{x}_j \mathbf{w}_{ji}^{\mathsf{T}}$. This leads to

$$\mathbf{W}_{ji} = \mathbf{W}_{ji} - \alpha \mathbf{F}'(\mathbf{z}_j) \boldsymbol{\Sigma}_j \mathbf{x}_i^{\mathsf{T}} \tag{15}$$

# 2 MNIST Classification

## 2.1 Task a and c

The network was implemented in Python using the NumPy library to do matrix computations. It was designed with OOP in mind, so that a new layer could be added to the network by instantiating a new "Layer" object with the desired activation function, inputs, and neurons.

We used 50000 images for our training and validation. 10% were used for validation, which resulted in 45000 images for training and 5000 for validation. This is because the computer that was used ran out of RAM, and was unable to use the entire dataset. We used 64 neurons for our hidden layer, and a batch size of 128 for our mini-batch gradient descent algorithm. An early stopping algorithm was implemented using the provided skeleton code, where the network stops training after 3 iterations where the validation cost increases. We used a learning rate $\lambda = 1$, and set the maximum number of epochs to 20. The results are shown in figure 2.1. The values on the x-axis correspond to every 10. time the gradient descent algorithm was run, as this is how often we calculated accuracy and loss. For these hyperparameters, we ended up with an accuracy of about 90% on the validation set

## 2.2 Task b

Didn't implement numerical approximation of gradient.

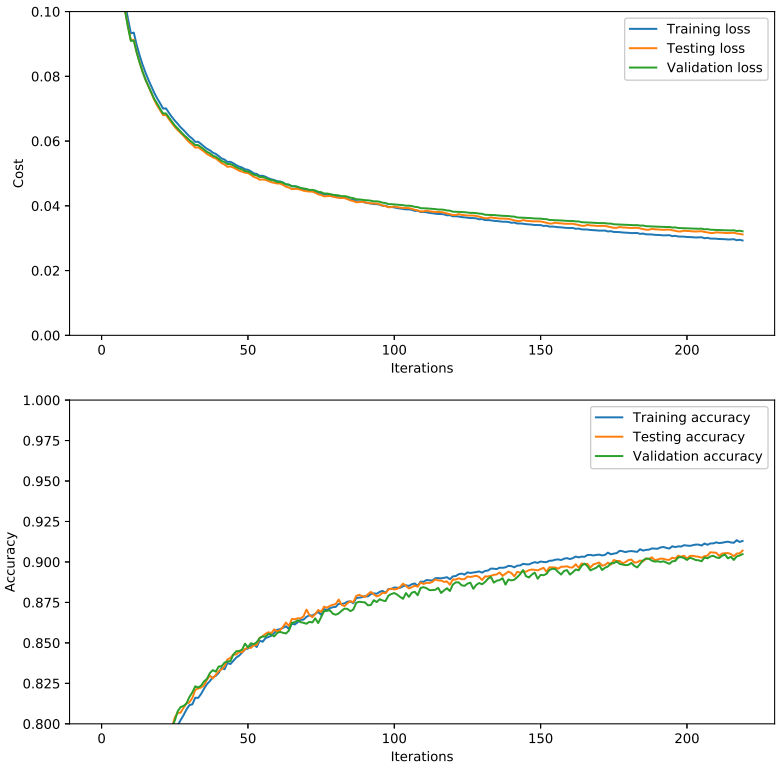


**Figure 2.1:** Accuracy and loss using backpropagation

# 3 Adding the "Tricks of the Trade"

## 3.1 Task a

When implementing random shuffling after every epoch, we did not see an improvement in number of iterations, but a slight increase in accuracy. The results are shown in 3.1

## 3.2 Task b

In this section, we implemented the improved sigmoid function as the activation function for the hidden layer. We added a small linear twisting term to the sigmoid function as described in LeCun et al. (2012), and ended up with the improved sigmoid

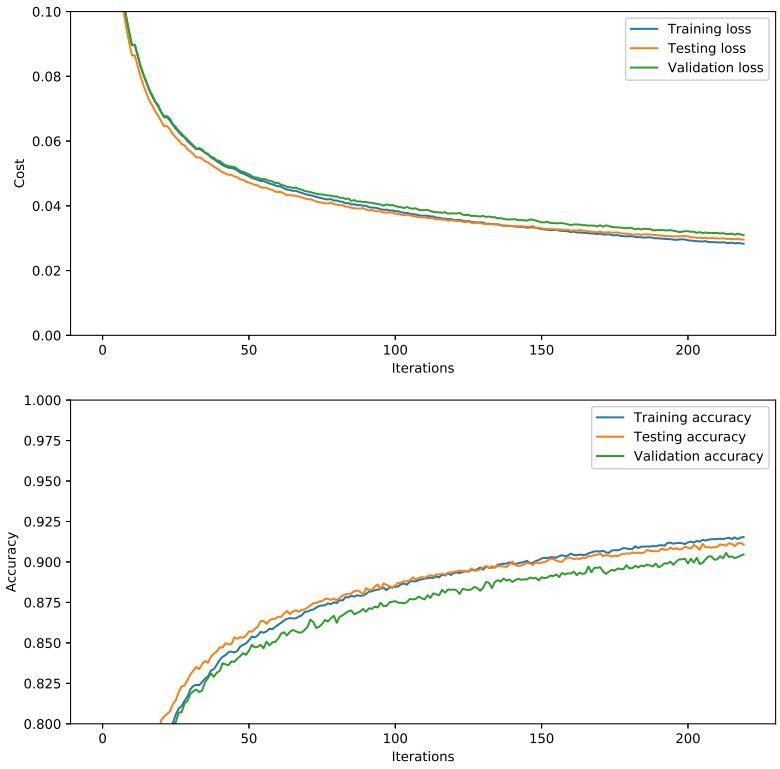


**Figure 3.1:** Accuracy and loss with random shuffling after each epoch

$$f(z) = 1.7159 \tanh \frac{2}{3} z + 0.001z \tag{16}$$

When using backpropagation, we also need the derivative of the activation function. This can be written as

$$f'(z) = \frac{1.14393}{(\cosh \frac{2}{3} z)^2} + 0.001 \tag{17}$$

We did see an improvement in iterations when $\lambda = 1$, but we changed the learning rate to $\lambda = 0.5$ for this task because the loss function and accuracy was oscillating too much. With the new learning rate, the loss function was much more stable, but took longer to converge. Therefore, we observed no shorter learning time, and a slight decrease in accuracy. Results are shown in figure 3.2

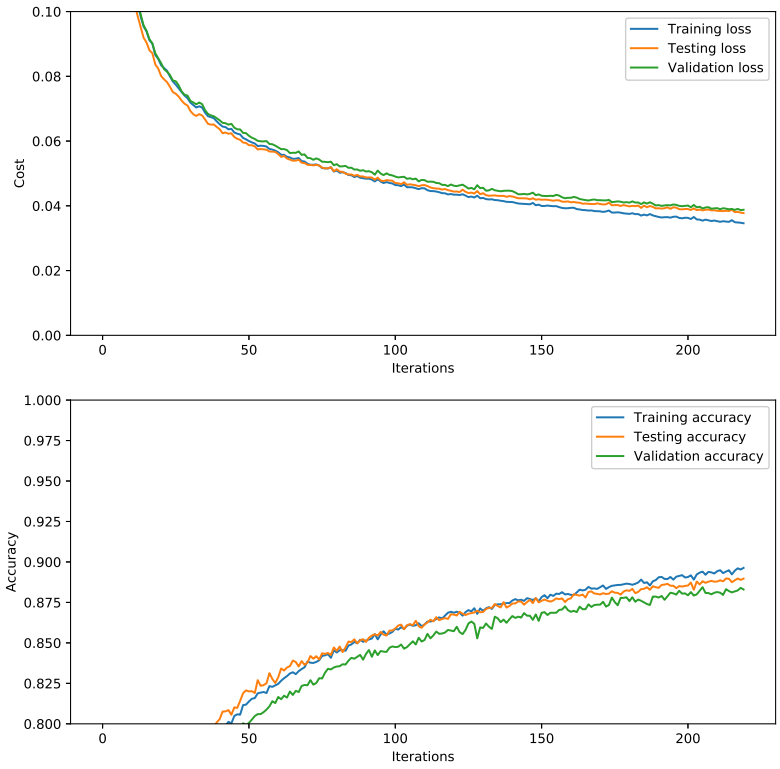


**Figure 3.2:** Accuracy and loss with improved sigmoid

## 3.3 Task c

For this task, we implemented a new way of generating the initial weights. In the previous tasks, the weights were initialized using a uniform random distribution. Now we initialize the weights by using a normal distribution. such that

$$W \sim \mathcal{N}(\mu, \sigma^2) \tag{18}$$

where $\mu = 0$, $\sigma = \frac{1}{\sqrt{n}}$, and $n$ is the number of inputs to the neuron.

This addition to our network significantly sped up the learning process. As can be seen in figure 3.3, the amount of iterations needed by the gradient descent algorithm was cut by more than half, and the accuracy increased to about 95%.

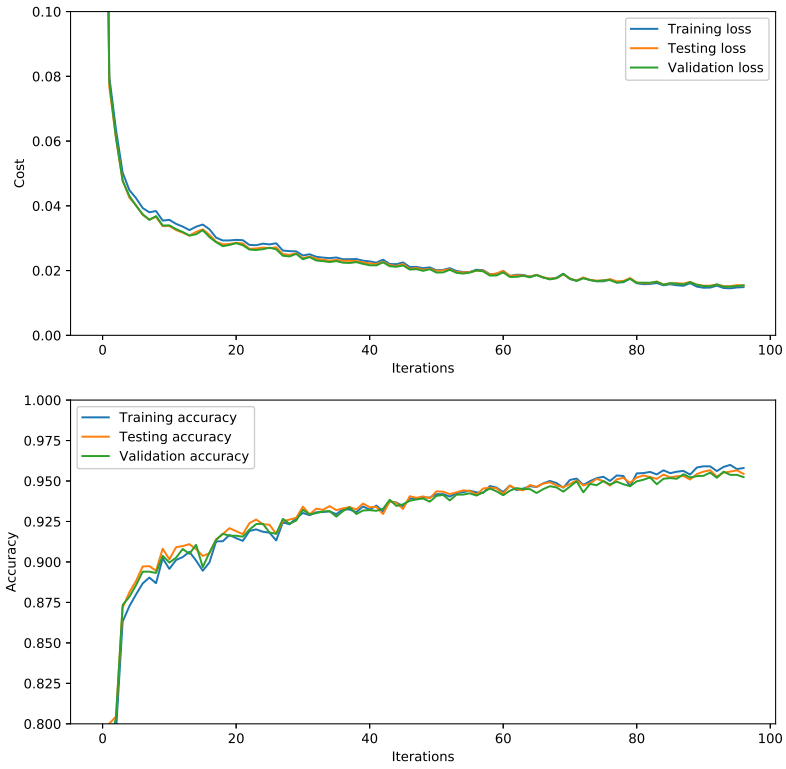


**Figure 3.3:** Accuracy and loss with normally distributed initial weights

## 3.4 Task d

LeCun et al. (2012) proposes adding a new term to the update step can speed up the learning process "[...] when the cost surface is highly nonspherical because it damps the size of the steps along directions of high curvature, thus yielding a larger effective learning rate along the directions of low curvature." This term is the previous gradient multiplied by an arbitrary constant. For this task, we used the constant $\mu_m = 0.9$ where the new update step can be written as

$$\Delta w_{t+1} = \alpha \frac{\partial C_{t+1}}{\partial w} + \mu_m \Delta w_t \tag{19}$$

Another increase in speed can be seen in figure 3.4. We did however see an ever so slight decrease in accuracy, but we believe this can be fixed by tuning the hyperparameters further.

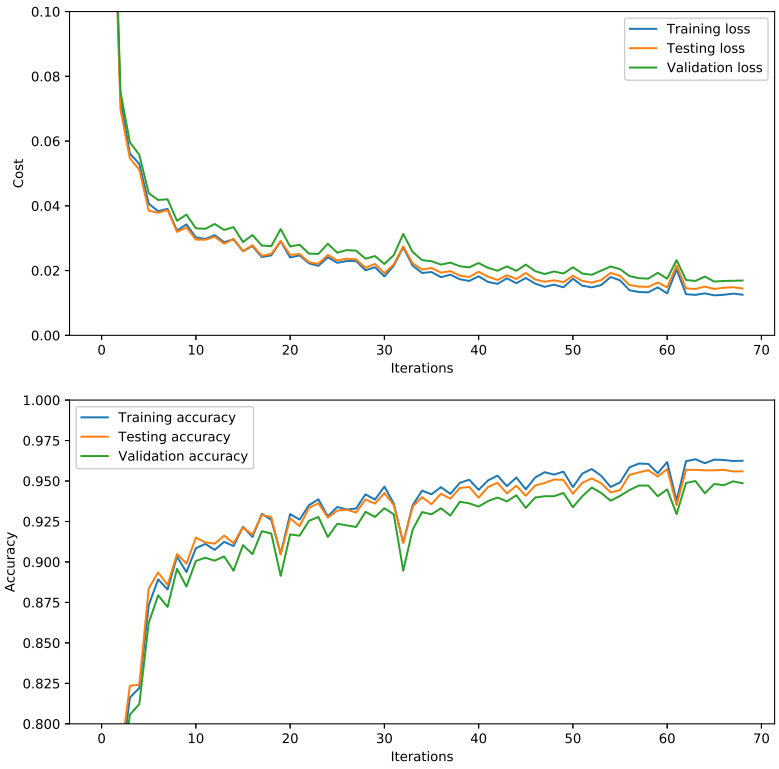


**Figure 3.4:** Accuracy and loss with added momentum term

# 4 Experiment with network topology

## 4.1 Task a

We tried to halve the number of hidden units to 32 neurons. We observed that the training procedure was around 10% faster, but the accuracy dropped by several percent. See figure 4.1

## 4.2 Task b

We tried to double the number of hidden units to 128 neurons. The gradient descent algorithm took much longer time to run each iteration. The amount of iterations needed was about the same as the previous task where we halved the number of units, but the accuracy was about the same as with 64 neurons. From figure 4.2 we noticed the cost stayed relatively flat at the start of the

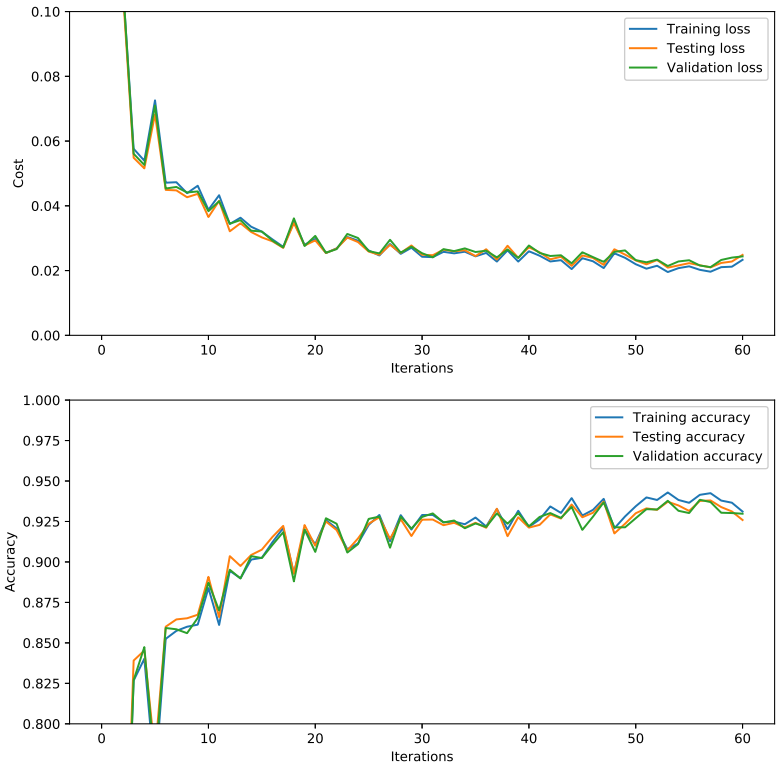


**Figure 4.1:** Accuracy and loss with 32 hidden units

training procedure. LeCun et al. (2012) mentions this can be fixed by tuning the twisting term added to the improved sigmoid function.

## 4.3 Task c

In this task, we add another layer to our network. The previous task operated with 50954 parameters, so we decided to set the number of neurons in both of the hidden layers to 59. This network operates with 50514 parameters, which is close to what the previous tasks used. This resulted in significant decrease in iterations, and a huge drop in accuracy. We know that adding more hidden layers may cause overfitting, so the accuracy decrease was expected. See figure 4.3.

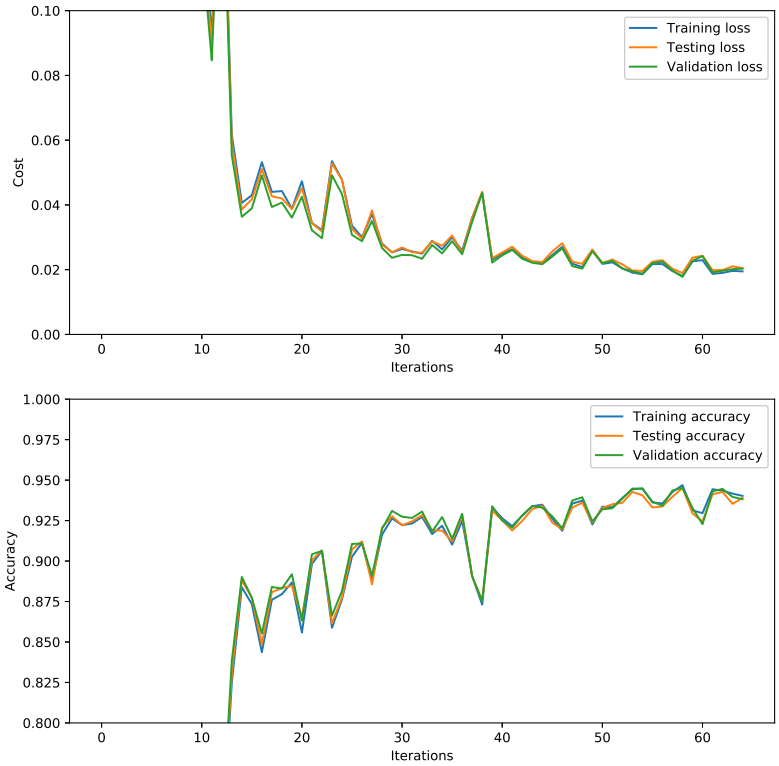


**Figure 4.2:** Accuracy and loss with 128 hidden units

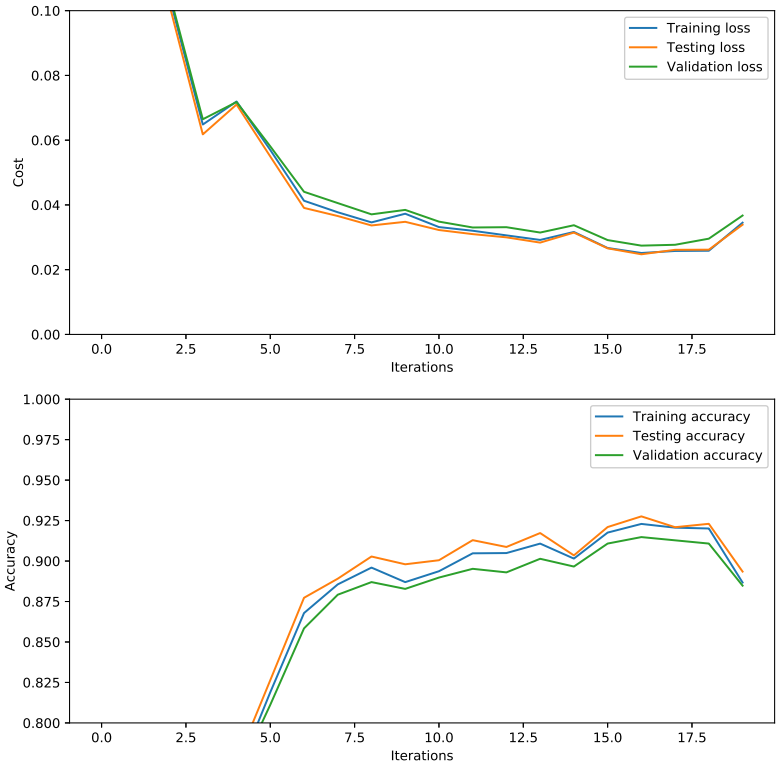


**Figure 4.3:** Accuracy and loss with two hidden layers with 59 neurons

# 5 Bonus

## 5.1 Implementing ReLU

For the bonus, we decided to implement the ReLU activation function with one hidden layer that consists of 64 neurons. The ReLU activation function was implemented as follows

**Listing 1:** ReLU activation function and its derivative

```
def relu(z):
    return np.maximum(0,z)

def relu_der(z):
    z[z <= 0] = 0
    z[z > 0] = 1
    return z
```

The ReLU algorithm was significantly faster to compute than the sigmoid functions.

# 6 Bibliography

LeCun, Y., Bottou, L., Genevieve, B. O., and Müller, K.-R. (2012). *Efficient backprop*. Springer.