TDT4265 Assignment 2

Kristian Haga

February 2019

1 Task 1 at the end of the report

2 Task 2

2.1 2a-c,e - Training procedure

2.1.1 Pre-processing

Before the training procedure begins the solution pre-process data to increase the accuracy and the performance of the solution.

By normalizing the input data, vector of pixel values in the interval $[0, 255]$, the solution ensures that all pixel values is in the interval $[-1, 1]$. This is done with line:

```
X_train, X_test = (X_train / 127.5) - 1 , (X_test / 127.5) - 1
```

Normalizing the code avoids large input values to carry a heavier weight when calculating the gradient descent and such.

The solution then implement "The Bias Trick" with the function `bias_trick(X)` on the input. "The Bias Trick" shifts the activation function in the desired direction. This can be critical for successful learning.

To fit the Softmax 10 class activation function used for the output layer, the target data is "One-hot encoded" with the function `onehot_encode(Y)`.

To wrap up the pre-processing of the data set, we split our training data into a training and validation set with the function `train_val_split(X, Y, val_percentage)`. We do this split to avoid data snooping. Meaning the network learn to recognize the pictures values from seeing them before instead of recognizing the values of the pictures by their features

2.1.2 Training Loop

The training loop is implemented in the function `train_loop()`. The loop consist of two nested for-loops. An iteration in the outer loop represents an epoch. In each epoch the training loop iterates through every mini-batch of the training data. A mini-batch is a small subset of training data.

The inner loop represent one mini-batch gradient descent. The usage of mini-batches avoid that the network get stuck in a local minima or saddle point, while also achieving a speedy convergence compared to pure SGD.

After the solution creates a mini-batch from the training set, it performs one mini-batch gradient descent on the mini-batch.

In mini-batch gradient descent a subset, mini-batch, of the training set is used to calculate model error and the gradient. Here we take the average of the gradient found for each input when we update the weights once for each batch.

The solution finds the average gradient of a batch and updates the weights with the function

```
gradient_descent(X, targets,
   w_j, w_k, learning_rate, should_check_gradient)
```

Where X is the input batch, targets is the set of desired output and *should_check_gradient* is a Boolean which states if the gradient found by the function agrees with a numerical approximation.

To implement back propagation the solution first finds the gradient for the output layer, w_k. The output layer uses Softmax as the activation function, hence the change in error is $\delta_k = -(targets - outputs)$ as shown in Assignment 1. We then get the update rule:

$$w_k = w_k - \alpha \delta_k a_j \tag{1}$$

where α is the learning rate and a_j is the output of the activation function used in the hidden layer.

Before the solution applies the update rule above, the computed gradient $\delta_k * a_j$ is normalized by batch size and number of output classes. This is done by simply dividing the gradient by the product of the batch size and number of classes.

We then find the gradient $dw_j = \delta_j * X$ for the hidden layer. This layer uses the Sigmoid function as its activation function. Using the change in error δ_j found in Task 1.1 we get the update rule

$$w_j = w_j - \alpha \delta_j X \tag{2}$$

Also for the hidden layer the gradient is normalized before it is used in the update rule.

To implement the use of mini-batch gradient descent and to increase the performance of the solution, we calculate the updates of the weights and the gradients using vectorized code. As a concequence sums are efficiently calculated as a vector dot product, e.g. the sum in δ_k is found by doing `delta_k.dot(w_k)`.

2.1.3 Initialization and Hyperparameters

The weights are initialized as two matrices. The weights from the input layer to the hidden layer are represented by $u \times n$ matrix w_j, where u is the number of units in the hidden layer and n is the number of input values.

The weights from the hidden layer to the output layer are represented by $y \times u$ matrix w_k, where y is the number of output classes. All weights values in w_j, w_k are initially set to a random value in the interval $[-1, 1]$

Initial hyperparameter values are:

- Batch size = 128
- Learning rate = 0.9
- Units in hidden layer = 64
- Max epochs = 20

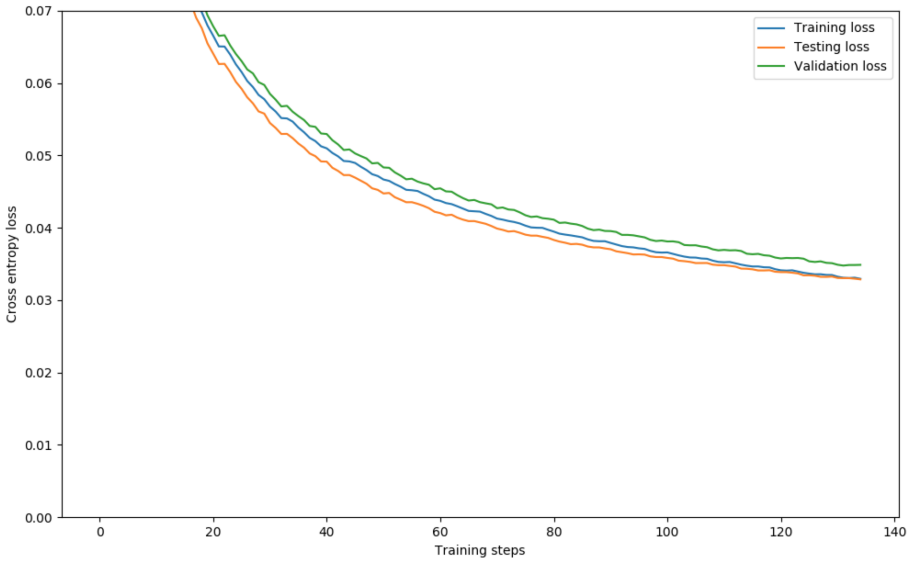
2.2 Numerical approximation

While checking the gradient, we experienced that some runs of the code resulted in the test failing by very little, while in other runs the code passed the test. This might be a result of the random initialization of the weights and that we checked our gradient after every epoch. The weights might not be well trained after only one epoch, resulting in the test failing.

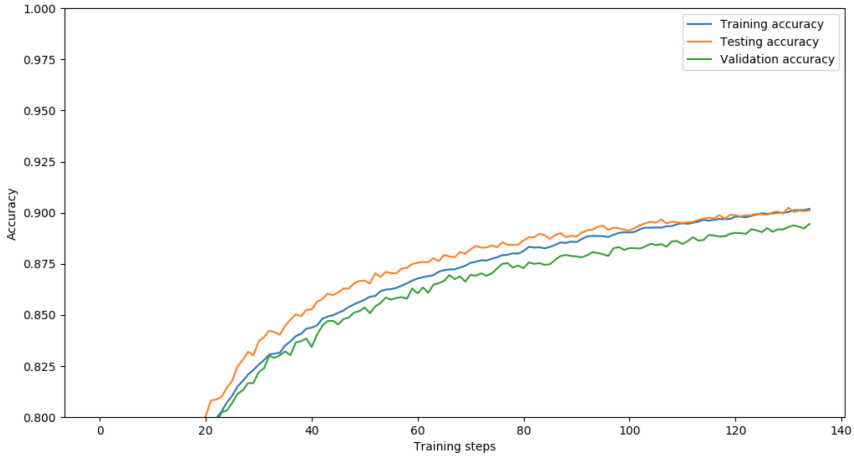
We measured maximum absolute error for the hidden layer to 0.00012 and for the output layer to 0.00021.

2.3 Results of training

2.3.1 Computed Cross Entropy Loss



2.3.2 Measured Accuracy



3 Task 3 - "Tricks of the Trade"

Note: In this section the learning rate was set to 0.5 to achieve smoother graphs.

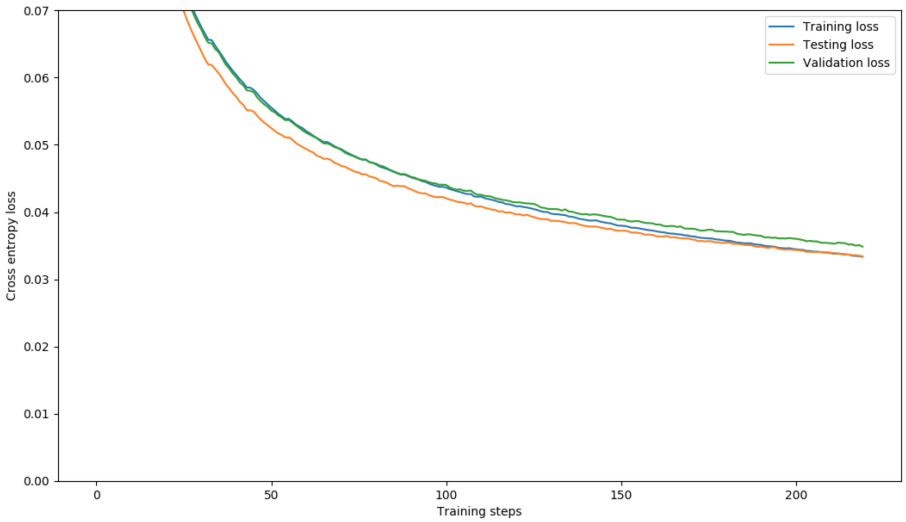
3.1 Shuffle training set

The function `shuffle_train_set(X, Y)` takes in the inputs X and targets Y in the training set and shuffles them. We do this because the network learn the fastest from unexpected data. Without shuffling the data the network may remember the order of the samples, hence their target values.

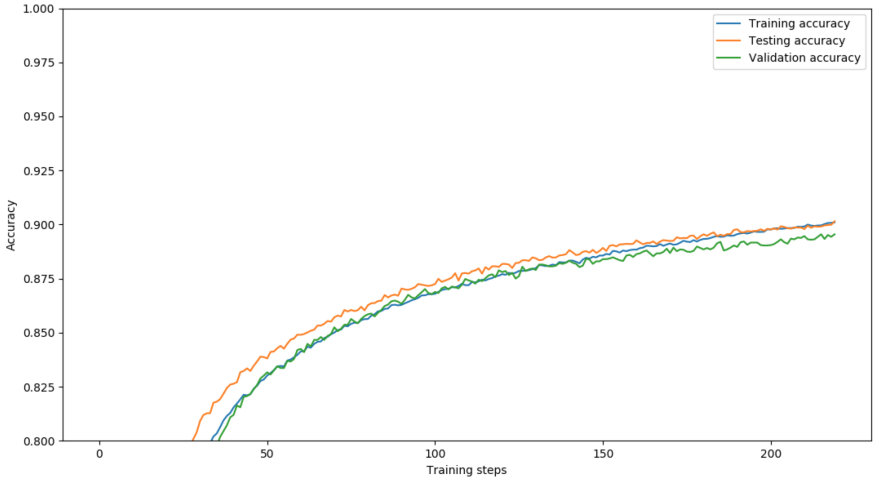
3.1.1 Results with shuffling

With shuffling of the training set after each epoch early stopping kicked in during the 20th epoch and the accuracy on the validation set was 0.9015.

Calculated Loss



Calculated Accuracy



3.2 Improved Sigmoid

Implemented the improved Sigmoid function

$$1.7159 \tanh(\frac{2}{3}x) \tag{3}$$

and its slope

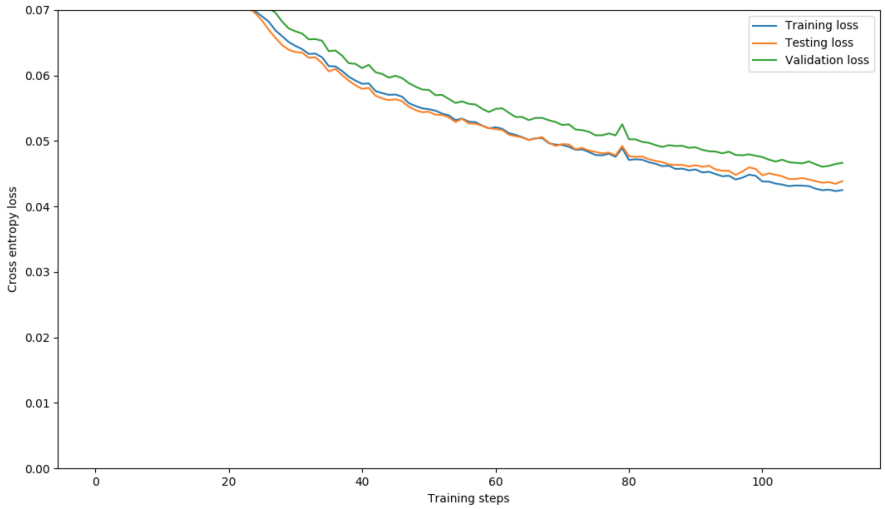
$$\frac{1.7159 * 2}{3}(1 - \tanh^2(\frac{2}{3}x)) \tag{4}$$

used for backpropagation.

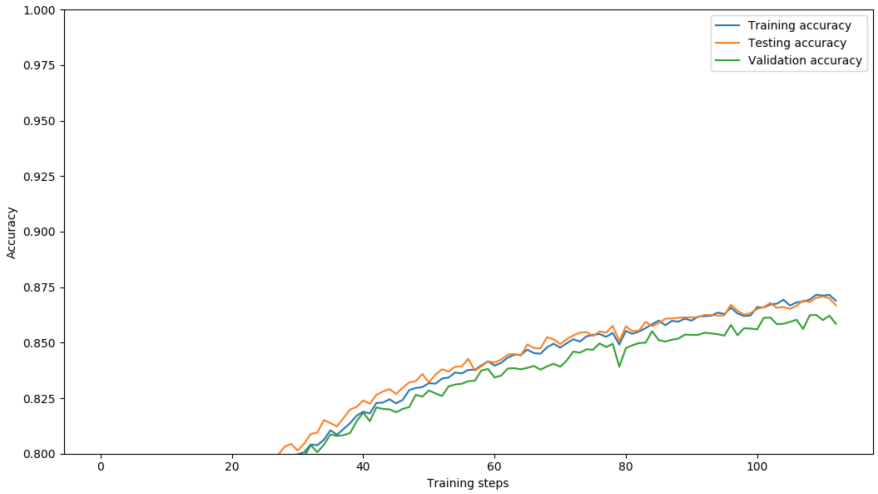
3.2.1 Results with improved Sigmoid

We observe that early stopping kicked in during the 11th epoch. This is nearly half of the training iterations needed compared with the standard Sigmoid. The accuracy of the network was 0.8702, which is lower.

Calculated Loss



Calculated Accuracy



3.3 Normal distributed initial weights

By choosing initial values for the weights such that the Sigmoid is primarily activated in the linear region we enhance the learning speed. Very small or large weights will result in small gradients and as a consequence make learning

slow.

This is achieved by randomly draw initial weight values from a normal distribution with mean zero and standard derivation σ:

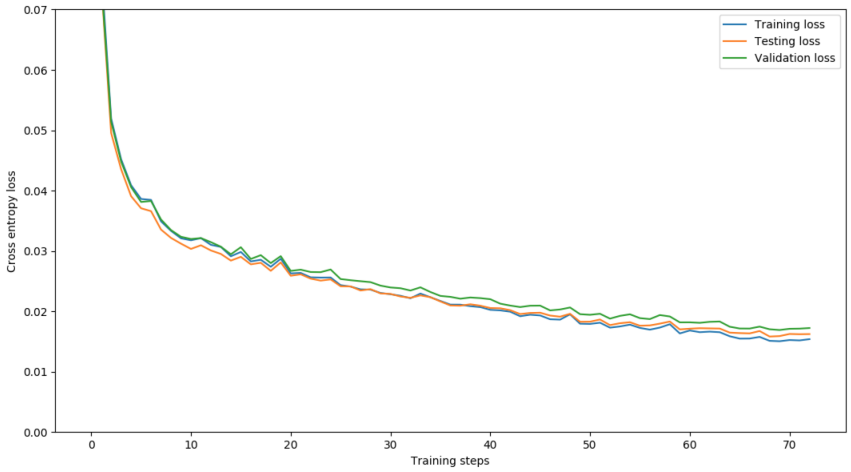
$$\sigma = m^{-\frac{1}{2}} \tag{5}$$

where m is the fan-in, e.i. the number of connections into a node. Here this will be the size of the input vector for the hidden layer and the number of units in the hidden layer for the output layer.

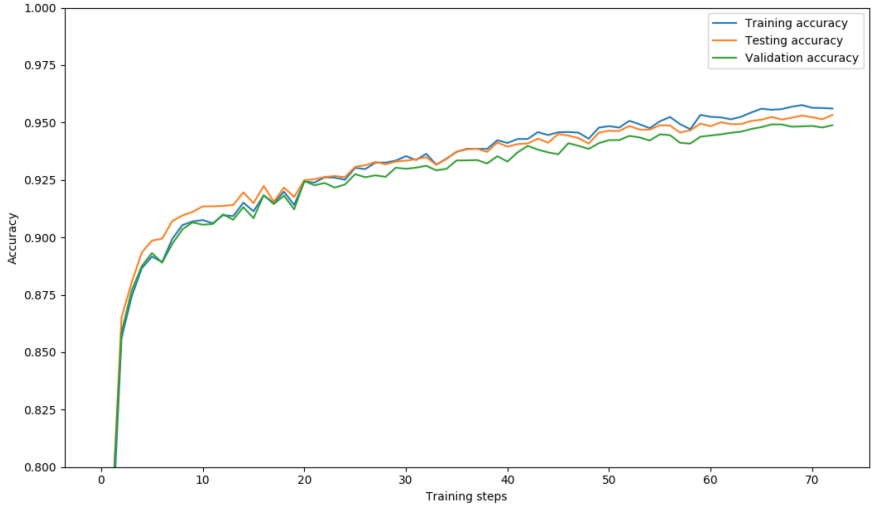
3.3.1 Results with normal distributed weights

We observe that early stopping kicked in during the seventh epoch and that the accuracy of the network was 0.9530. This is some improvement regarding learning speed and a significant improvement in accuracy. We did not expect such a large improvement.

Calculated Loss



Calculated Accuracy



3.4 Momentum

The momentum defines the step size taken towards the optimum. The goal of momentum is to speed up the training with a reduced risk of overshooting the optimum. Thus it is an improvement over choosing a large learning rate which would quickly converge towards a solution, but might overshoot the optimum.

Implementing momentum we get the update rules for weights w in the $t+1$ iteration:

$$w(t+1) = w(t) - \alpha dw(t+1) - \mu dw(t) \tag{6}$$

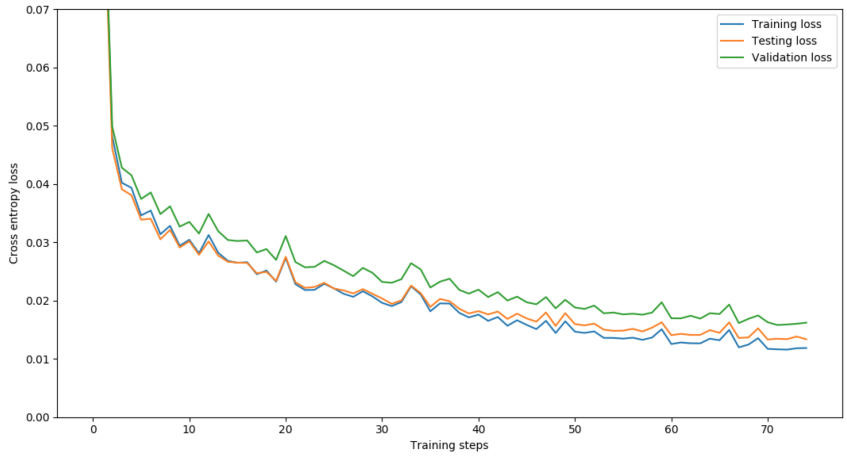
where α is the learning rate and μ is the momentum rate.

3.4.1 Results after implementing momentum

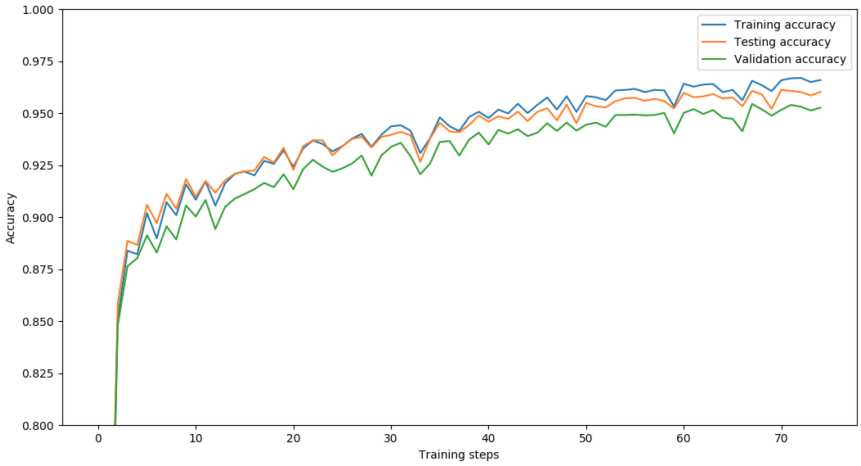
Adding the final "Tricks of the Trade" resulted in an early stopping during the seventh epoch. The solution achieved an accuracy of 0.9634 on the validation set, which we regard as a good result.

Overall the tricks have both improved learning speed and accuracy significantly.

Calculated Loss



Calculated Accuracy



4 Task 4

4.1 Results when halving the number of units in the hidden layer

When using 32 instead of 64 units in the hidden layer we observed a slight dropp in accuracy, on average 0.015, and no significant changes in learning speed. Early stopping occurred on average by 8 epochs, but we experienced a large variance in number of epochs needed. The number varied from 6 to 15.

When the number of units was set to 16 the learning speed increased drastically. Early stopping typically occurred in the third or fourth epoch. This came

at the cost of the accuracy which dropped to 0.92 on average. This is a drop of 0.04 from when we had 64 units.

These results suggest that the less complex network results in a underfitted model, but that the model is quickly learned.

4.2 Results when doubling the number of units in hidden layer

Naturally it was more time consuming to train the more complex model. Even when increasing the number of units to 256 did we observe any significant changes in number of epochs, but the accuracy was more or less the same as with 64 units in the hidden layer.

Even with 512 units did we experience any significant changes except the time consumption.

In theory by increasing the number of units we should experience that the model would be overfitted and poor results on unseen data as a consequence. The model should be less general and recognize the training set more than learning the general features of the different inputs.

4.3 Adding a layer

Complexity in neural networks are measured by the number of parameters:

$$\#parameters = \#weights + \#biases \tag{7}$$

The complexity of the network before adding a layer was given by:

$$complexity = (inputs + biases) * units_{hidden} + units_{hidden} * outputs \tag{8}$$

$$complexity = (784 + 1) * 64 + 64 * 10 = 50880 \tag{9}$$

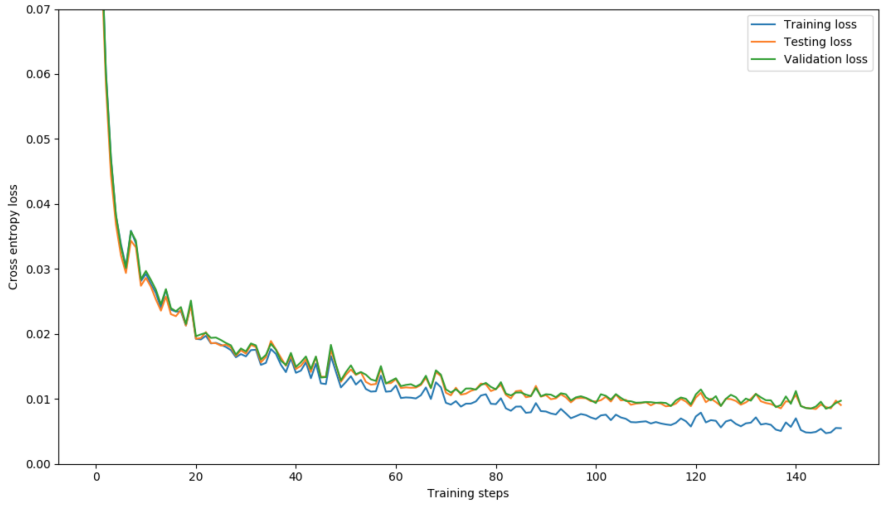
When adding a fully connected layer we get the complexity:

$$complexity = (inputs + biases) * u_{h_1} + u_{h_1} * u_{h_2} + u_{h_2} * outputs \tag{10}$$

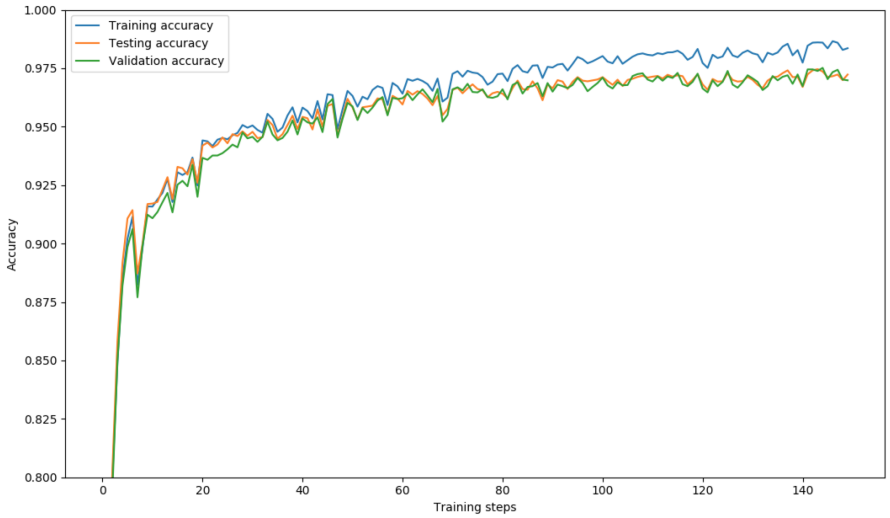
where u_{hi} is the number of units in hidden layer i. Thus to have the same complexity after adding a layer, the number of units in the hidden layers must be reduced. Reducing the units in the hidden layers to 59 we get a complexity of 50 386.

We did experience some improvement in the accuracy and at best measured an accuracy on the validation set of 0.97. The learning speed increased a lot, although the model is not more complex. This was a surprise.

Calculated Loss with 59 units in hidden layer



Calculated Accuracy 59 units in hidden layer



5 Task 5 - Bonus

5.1 Dropout

The solution implements the regularization mechanism dropout with the functions `get_dropout_index(w, p)`, `do_dropout(w, drop_index)` and `scale_for_dropout(w, p)`, where w is the weights and p is the percentage of units to drop.

When performing dropout the network choose not to update some of the weights in an iteration of gradient descent. This is to avoid overfitting by penalizing the loss function, such that the network performs better on unseen data.

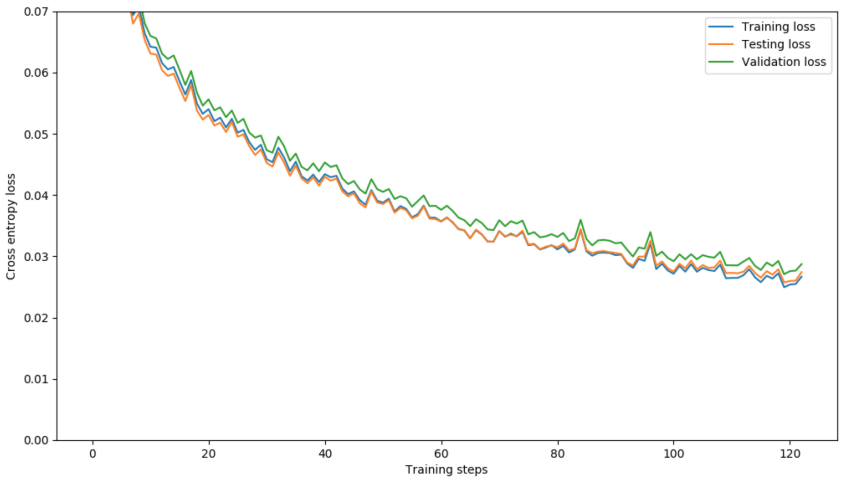
To assure that the same weights are dropped when feeding an input through the network and when doing backpropagation, the solution finds which weights to drop with `get_dropout_index(w, p)`. Which indices to drop is found using an binomial distribution with success probability p. This returns a matrix of same shape as the input weights with 1's and 0's. If an element is zero, then the corresponding weight is dropped.

To actually perform the dropout `do_dropout(w, drop_index)` element wise multiply the weights w with the index matrix of same shape.

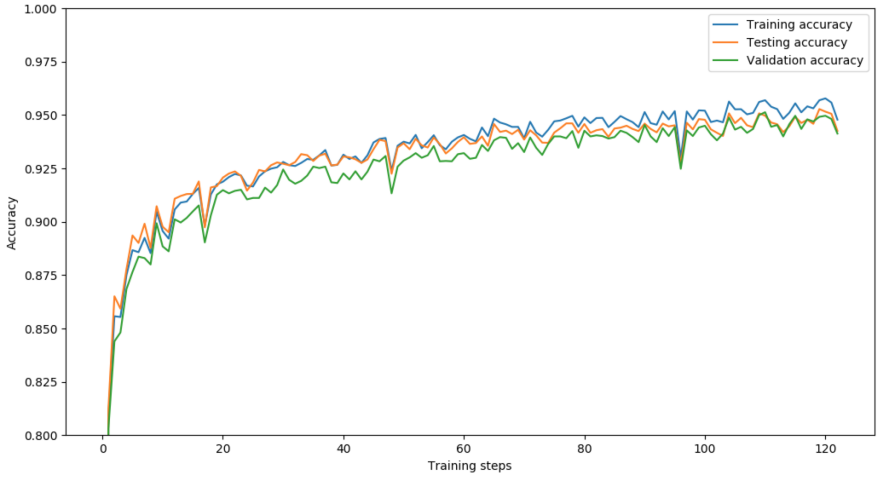
When testing we need to scale the activation of the weights in order to account for the missing activations during training. This is done by
`scale_for_dropout(w, p)`

Implementing dropout should increase the number of iterations before the network converge on a solution and decrease the difference between testing and validation/test loss and accuracy. This is confirmed in the figures below, but we also observe a loss inn the test accuracy compared to previous results. This is not desired. I might have done some mistakes in the implementation.

Calculated Loss



Calculated Accuracy



Problem 1.1

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

$$= w_{ji} - \alpha \, \delta_j \cdot a_i$$

Since the input layer do not have any activation function, the output of neuron x_i is simply the input, hence we get

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

We have the sigmoid function as our activation function in the hidden layer, hence

$$a_j = \text{sigmoid}(z_j) = \text{sigmoid}\left(\sum_{i=0}^{d} w_{ji} x_i\right) = f(z_j)$$

$\Rightarrow f'(z_j)$ is the slope of the hidden activation function.

From the problem we define $\delta_j = \frac{\partial C}{\partial z_j}$ as the change of the cost with regard to the activation.

δ_j is then the change in the activation, $f'(z_j)$, multiplied with the propagated error in the output layer, $\sum_K w_{kj} \delta_K$. Here δ_K is the change of error in the output layer. This "moves" the error backward through the activation function and network as a whole.

As a result $\delta_j = f'(z) \cdot \sum_K w_{kj} \delta_K$ and we get

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

More formal we have : $\frac{\partial C}{\partial w_{ji}} = \frac{\partial f(z_j)}{\partial z_j} \cdot \frac{\partial C}{\partial f(z_j)} \cdot \frac{\partial z_j}{\partial w_{ji}}$

where $\frac{\partial f(z_j)}{\partial z_j} = f'(z_j)$

$$\frac{\partial z_j}{\partial w_{ji}} = x_i$$

$$\frac{\partial C}{\partial w_j} = \sum_K w_{Kj} \underbrace{\left(-(t_K - y_K)\right)}$$

Change in error in output layer $= \delta_K$

We get $\frac{\partial C}{\partial w_{ji}} = f'(z_j) \cdot \sum_K w_{Kj}\, \delta_K \cdot x_i$

Problem 1.2

The error in the output layer K is $\delta_i^K = \frac{\partial C}{\partial f(z_i^K)} \cdot f'(z_i^K)$ for the ith output activation.

We can rewrite δ^K into a matrix based form as

$$\delta^K = \nabla_f C \odot f'(z^K)$$

where $\nabla_f C$ is a vector of the partial derivatives $\frac{\partial C}{\partial f(z_j^K)}$

We get the update rule $w_K := w_K - \alpha\left(\nabla_f C \odot f'(z^K)\right)$

For the hidden layer weights w_j we get the update rule

$$w_{ji} := w_{ji} - \alpha\left(f'(z_j) \sum_K w_{Kj}\, \delta_K\right)$$

$$= w_j - \alpha\left(f'(z_j) \odot w_K{}^T \delta_K\right)$$