NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4265 - Computer Vision and Deep Learning

Assignment 2

Kolbjørn Kelly
Sander Endresen

19th February 2021

Contents

1 Task 1

1.1 1a) Backpropagation

To show that

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

we have to show that $\frac{\partial C}{\partial w_{ji}} = \delta_j x_i$. Applying the chain rule, we get

$$\frac{\partial C}{\partial w_{ji}} = \frac{\partial C}{\partial z_j} \frac{\partial z_j}{\partial w_{ji}}, \tag{1}$$

where the first term is given by definition, namely

$$\frac{\partial C}{\partial z_j} = \delta_j. \tag{2}$$

Differentiating $z_j = \sum_i w_{ij} x_i$ w.r.t. its incoming weights, w_{ji}, we get

$$\frac{\partial z_j}{\partial w_{ji}} = \sum_i x_i. \tag{3}$$

As we eventually want to find the change in the cost function w.r.t. one given weight, we only consider one instance of the summation above. This might be a severe abuse of mathematical notation, but the point should be clear. Inserting (2) and (3) into (1), we get

$$\frac{\partial C}{\partial w_{ji}} = \delta_j x_i, \tag{4}$$

and thus,

$$w_{ji} := w_{ji} - \alpha \delta_j x_i. \tag{5}$$

To show that

$$\delta_j = f'(z_j) \sum_k w_{kj} \delta_k, \tag{6}$$

we first apply the chain rule to obtain

$$\delta_j = \frac{\partial C}{\partial z_j} = \sum_k \frac{\partial C}{\partial z_k} \frac{\partial z_k}{\partial z_j}, \tag{7}$$

where

$$\frac{\partial C}{\partial z_k} = \delta_k \tag{8}$$

by definition.

Differentiating $z_k = \sum_j w_{kj} a_j = \sum_j w_{kj} f(z_j)$ w.r.t. z_j, we get

$$\frac{\partial z_k}{\partial z_j} = \sum_k w_{kj} f'(z_j). \tag{9}$$

Inserting this to equation 7, we get

$$\delta_j = \sum_k w_{kj} f'(z_j) \delta_k = f'(z_j) \sum_k w_{kj} \delta_k \tag{10}$$

1.2 1b) Vectorized computation

The update rule for the last hidden layer to the output layer is given by

$$w_{kj} := w_{kj} - \alpha \frac{\partial C}{\partial w_{kj}} = w_{kj} - \alpha \delta_k a_j. \tag{11}$$

In vector form, this equation can be written as

$$\mathbf{W}^k := \mathbf{W}^k - \alpha \boldsymbol{a}^j {\boldsymbol{\delta}^k}^T, \tag{12}$$

Where the superscript k and j is used to denote the layer. Here, α denotes the learning rate, and is a fixed, scalar value. $\boldsymbol{a}^j$ is a vector containing the activations from the last hidden layer. Hence it has dimension Jx1, where J is the number of units in the last hidden layer. $\boldsymbol{\delta}^k$ is a vector containing the error in the output layer, and is given to be

$$\boldsymbol{\delta}^k = -(\boldsymbol{y}^k - \hat{\boldsymbol{y}}^k) \tag{13}$$

where $\boldsymbol{y}^k$ is a vector conatining the labels provided, and $\hat{\boldsymbol{y}}^k$ is the output from the last layer. Both these vectors have dimension Kx1, where K is the number of units in the output layer. Hence, $\boldsymbol{\delta}^k$ is also of dimension Kx1. Transposing it yields the dimension 1xK.

Multiplying a vector of dimension Jx1 with a vector of dimension 1xK, yields a matrix of dimension JxK. Multiplying this matrix with a scalar does not alter the dimension. Hence the product $\alpha \boldsymbol{a}^j {\boldsymbol{\delta}^k}^T$ has the same dimension as $\mathbf{W}^k$, namely JxK, where J is the number of units in the last hidden layer, and K is the number of units in the output layer.

We now look at the update rule for the input layer to the first hidden layer. We have

$$\begin{aligned} w_{ji} &= w_{ji} - \alpha \frac{\partial C}{\partial w_{ji}} \\ &= w_{ji} - \alpha \delta_j x_i. \end{aligned}$$

In matrix form this can be rewritten to

$$\mathbf{W}^j = \mathbf{W}^j - \alpha \boldsymbol{x} {\boldsymbol{\delta}^j}^T.$$

α is again a scalar value, denoting the learning rate. x denotes the input layer, and has dimension Ix1, where I is the number of inputs given to the network. $\boldsymbol{\delta}^j$ is the error for the first hidden layer, and is given by

$$\delta_j = f'(z_j) \sum_k w_{kj} \delta_k = f'(z_j) \odot (w_{kj} \cdot \delta_k) \tag{14}$$

which can be written in vector notation as

$$\boldsymbol{\delta}^j = \boldsymbol{f}'(\boldsymbol{z}^j) \odot (\boldsymbol{W}^k \cdot \boldsymbol{\delta}^k). \tag{15}$$

It has already been shown that $\boldsymbol{W}^k$ is of dimension JxK and that $\boldsymbol{\delta}^k$ is of dimension Kx1. Hence, their product is of dimension Jx1. $\boldsymbol{z}^j$ denotes the weighted sum for the first hidden layer, and is also of dimension Jx1. Hence, the slightly abusive mathematical notation $\boldsymbol{f}'(\boldsymbol{z}^j)$ denotes a vector of the same dimension (Jx1), with each entry differentiated w.r.t. the weighted sum. The Hadamard product $\boldsymbol{f}'(\boldsymbol{z}^j) \odot (\boldsymbol{W}^k \cdot \boldsymbol{\delta}^k)$ yields a vector of dimension Jx1, which is the dimension of $\boldsymbol{\delta}^j$. Its transpose then has dimension 1xJ.

Finally, the product $\alpha \boldsymbol{x} \boldsymbol{\delta}^{jT}$ is a product of a scalar, a Ix1 vector, and a 1xJ vector. The resulting matrix then has dimension IxJ, which is also the dimension of $\mathbf{W}^j$.

2 Task 2

2.1 2c) Softmax regression for mini-batch gradient descent for multi-layer network

The resulting multi-layer neural network is shown in Figure 1.

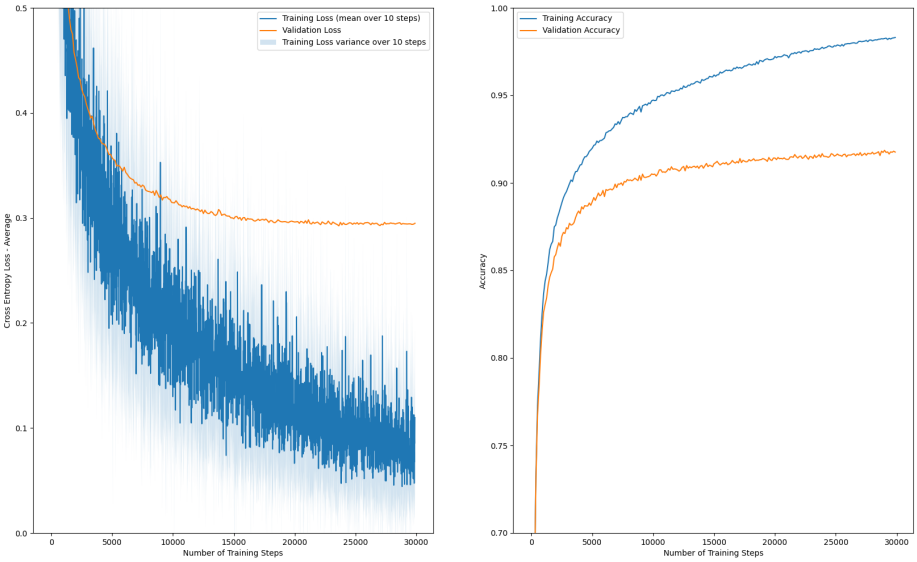

Figure 1: 2c) Multi-layer Neural Network

2.2 2d) Number of parameters in the network

For the sake of simplicity, we can think of the network as having two weight-matrices - one between the input- and hidden layer, and one between the hidden- and output layer. This gives a total of

$$\text{weights} = \text{weights}_{ji} + \text{weights}_{kj} = 785 * 64 + 64 * 10 = 50880 \tag{16}$$

parameters. Due to the bias-trick, the biases are included as the last input (the 785^{th} element).

3 Task 3

3.1 3a) Improved Weight Initialization

Figure 2 shows improved performance. The accuracy for the improved weight initialization reaches a stationary value earlier, indicating a greater rate of convergence. It is also more accurate in terms of both training and validation accuracy. There is a smaller discrepancy between the training- and validation accuracy, indicating better generalization and less overfitting. The early stopping also contributes to this. The main reason to initialize the weight in such manner, is to avoid saturation - meaning that the the weighted input to the activation function is either very small or very large. This makes the derivative of the sigmoid small, resulting in small gradients and slow learning. Initializing the weights to range the sigmoid's linear region also makes the network learn the linear part of the mapping before the unlinear part - reducing overfitting.

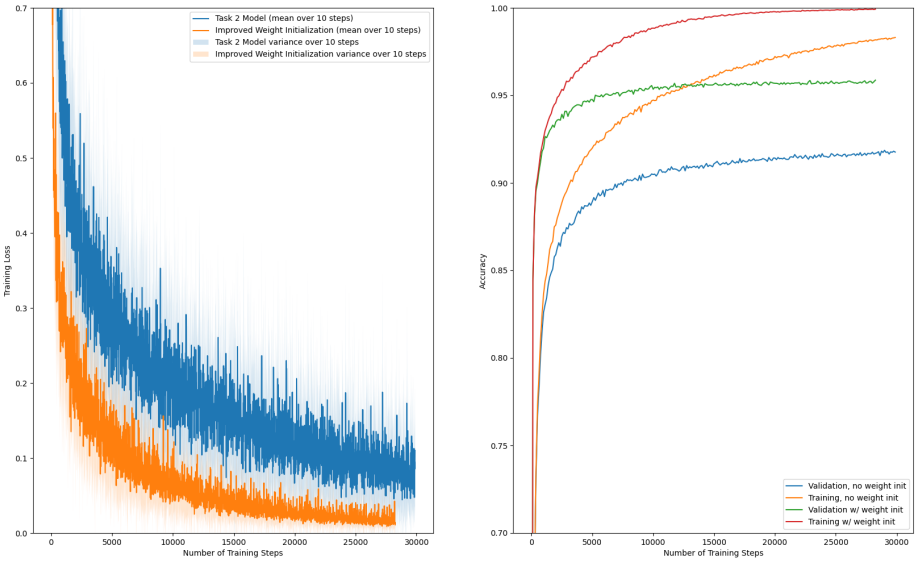

Figure 2: 3a) Model with and without weight initialization

3.2 3b) Improved Sigmoid

Figure 3 shows a major increase in convergence rate. The accuracy eventually ends up being approximately equal, but it takes less than half the number of training steps to achieve. The similarities between the validation accuracies indicates little to no signs of overfitting. The increase in convergence rate might come from the fact that the improved sigmoid is symmetric around zero. It is thus more likely to produce outputs closer to zero, avoiding saturation.

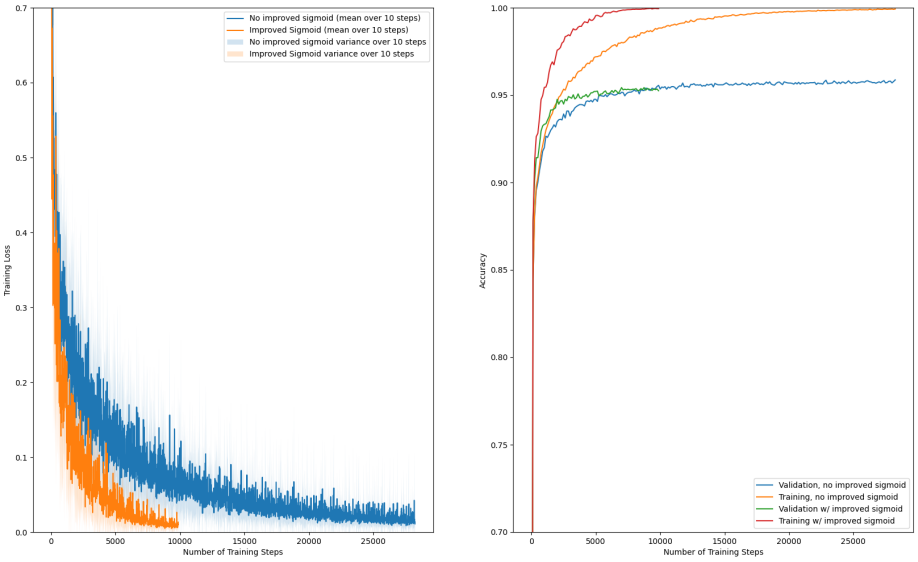

Figure 3: 3b) Model with and without improved sigmoid

3.3 3c) Momentum

Figure 4 shows minor differences in terms of convergence rate, accuracy and generalization/overfitting. In theory, momentum increases the step-length in directions of low curvature, thus increasing the convergence rate. Momentum might also step "through" cavities containing local minima, yielding a possibly global solution. This should, in theory, improve the final accuracy. In this case, however, it did not. This might be because the model actually found a good minimum in the first place, leaving the momentum a disturbing factor that does not actually contribute that much.

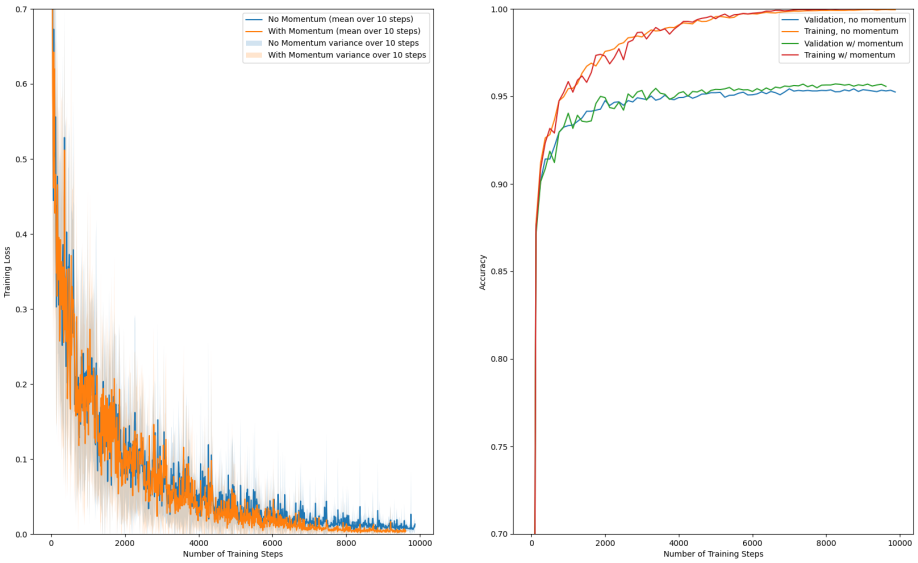

Figure 4: 3c) Model with and without momentum

4 Task 4

4.1 4a) 32 Hidden Units

The performance of the network with 32 hidden units is shown in Figure 5. Comparison with Figure 4 shows a decrease in training accuracy, and the magnitude of the validation loss indicate some degree of underfitting. This might be because the network is too simple to learn the complexity of the model. The convergence rate is, if anything, slightly lower. The smoothness of the validation loss curve might look strange, but it probably is a result from the fact that validation data is only stored after progressing 20% through the data set.

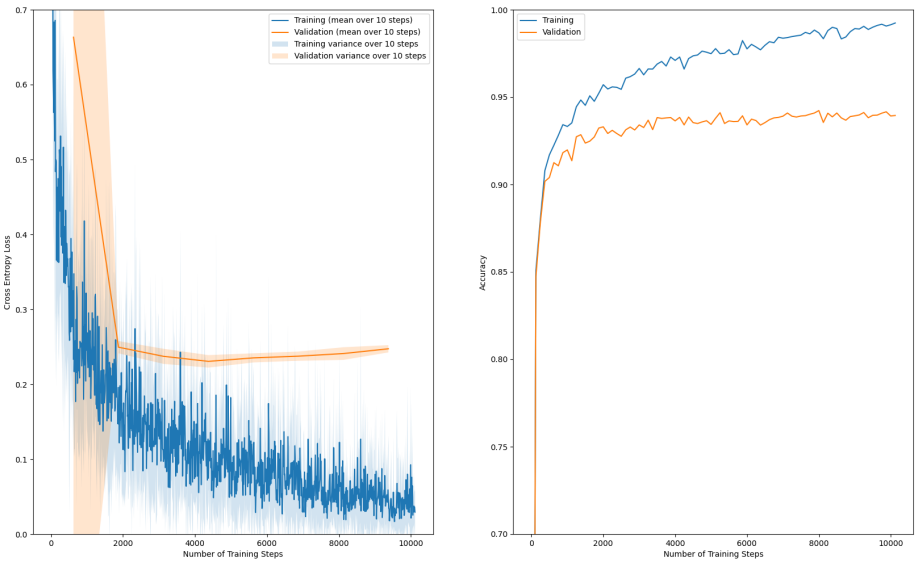

Figure 5: 4a) Network with 32 hidden units

4.2 4b) 128 Hidden Units

Figure 6 shows the network's performance with 128 hidden units. This network reaches approximately the same train accuracy as the one in Figure 4 (about 100%), while the validation accuracy is slightly higher. This is expected, as an increase of hidden layers increases the complexity which, in turn, increases training performance. The figure do, however, show that the training loss tends to zero, while the validation loss increases slightly. This is a classic indication of overfitting, and is to be expected when increasing the complexity of the network.

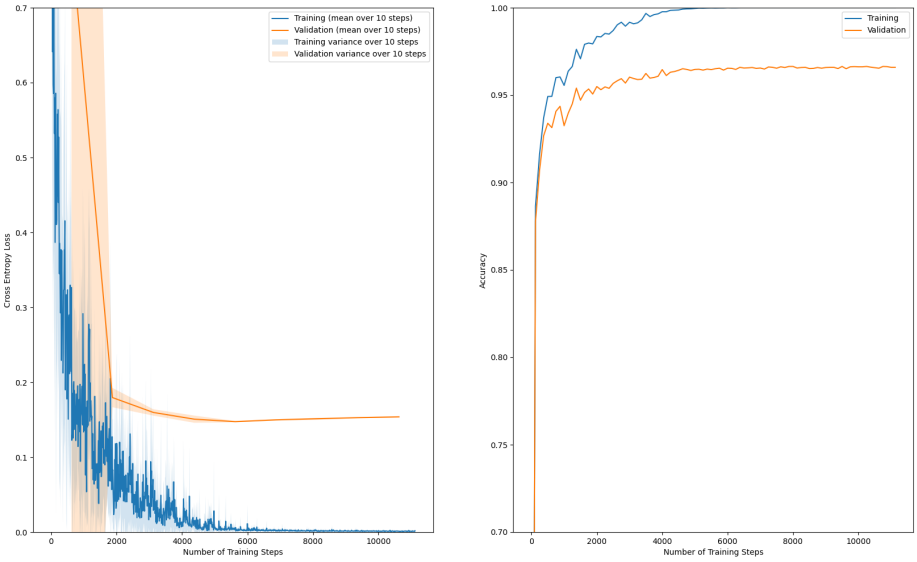

Figure 6: 4b) Network with 128 hidden units

4.3 4d) Multiple hidden layers

The number of parameters used in task 3 is the same as in task 2 (50880). Hence, in order to find the number of units needed to produce a two-hidden-layer network with approximately the same number of parameters, a quadratic equation must be solved. Indeed,

$$\begin{aligned} 785 * n + n * n + n * 10 &= 50880 \\ \Leftrightarrow n^2 + 795n - 50880 &= 0 \\ \Rightarrow n &\approx 60 \end{aligned}$$

produces $785 * 60 + 60 * 60 + 60 * 10 = 51300$ parameters.

The performance of the resulting network is shown in Figure 7. Compared to the network of task 3, we see a slight increase in terms of training accuracy. One can picture each hidden layer to represent one layer of abstraction, in terms of shape (or other attributes for that sake). It is thus expected that a deeper topology, with an equal amount of parameters, would preform better on more complex problems. The validation accuracy, on the other hand, is slightly lower. This might indicate slight overfitting, and is to be expected increasing the complexity of the topology. In terms of convergence rate, the two-hidden-layer topology is slightly slower. This is also expected from a more complex topology.

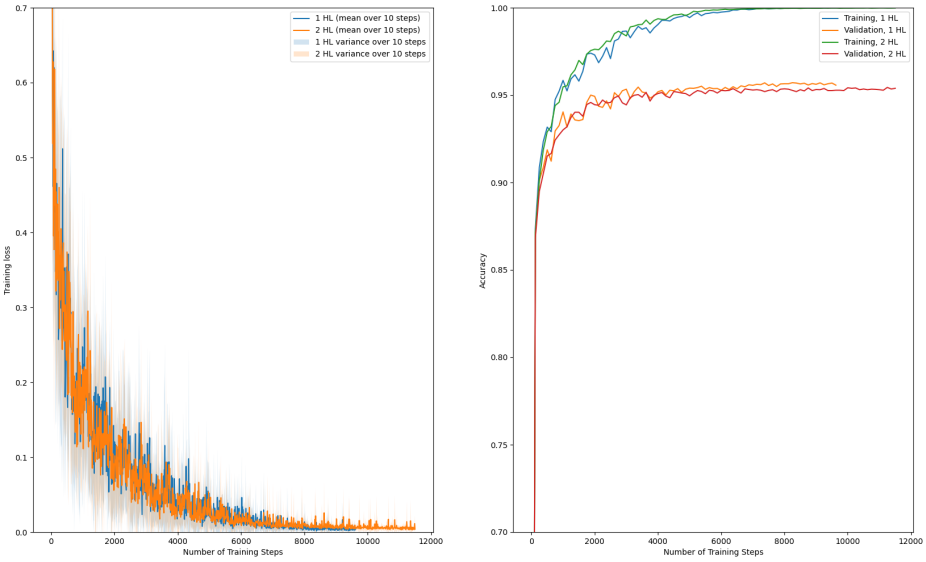

Figure 7: 4d) Network with two hidden layers with 60 units

4.4 4e) Model training with ten hidden layers and 64 hidden nodes

The performance is shown in Figure 8, and there are only minor differences compared to the network of task 3. It performs slightly worse in terms of training accuracy, indicating that the topology is more complex than needed for the given problem. The training loss is also slightly higher, indicating that the complexity has introduced some degree of overfitting. In terms of convergence speed, the complex network spends more than twice as many training steps before converging. This is to be expected when introducing a major change in the topology complexity. In conclusion, it is clear that the complexity introduced by 10 hidden layers is redundant for the given problem.

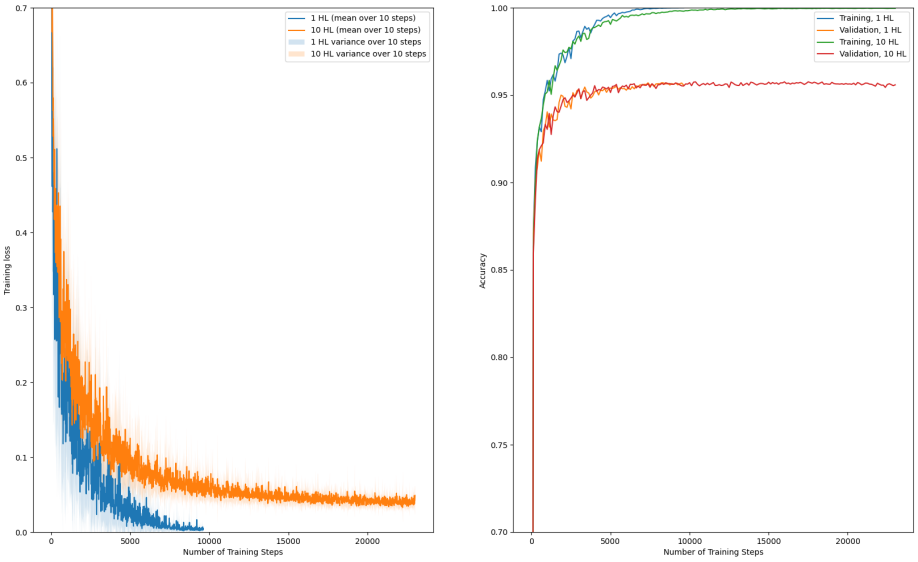

Figure 8: 4e) 10 hidden-layer network compared to 1 hidden-layer network