Assignment 02
TDT4265 - Datasyn og Dyplæring

Elias Mohammed Elfarri
Christopher Michael Vibe

Februar, 2021

Contents

1 Task 1 - Softmax regression with backpropagation **1**

1.a Backpropagation 1

1.b Vectorize computation 3

2 Task 2 - Softmax Regression with Backpropagation **6**

2.a 6

2.b 6

2.c Plot 6

2.d Model parameter count 6

3 Task 3 - Adding the "Tricks of the Trade" **8**

3.a Improved weights 9

3.b Improved weights & Improved sigmoid 10

3.c Improved Weights & Improved Sigmoid & Momentum 11

4 Task 4 - Experiment with network topology **12**

4.a 32 Neurons - (Code found in file task4ab.py) 12

4.b 128 Neurons - (Code found in file task4ab.py) 13

4.c 14

4.d Approximate parameters - (Code found in file task4d.py) . . . 14

4.e 10 Hidden layers - (Code found in file task4e.py) 16

1 Task 1 - Softmax regression with backpropagation

1.a Backpropagation

First we need to clear up the meaning of the different notations:

Where z_j is the sum of the input nodes and weight going from i to j.

$$z_j = \sum_i^I w_{ji} x_i$$

a_j denotes one of the hidden nodes defined by the sigmoid activation function that squishes the sum of input nodes and weights.

$$a_j = f(z_j) = \frac{1}{1 + e^{-z_j}}$$

z_k that is defined by the sum of weights going from the hidden node j to the output node k mulitplied with its respective hidden node.

$$z_k = \sum_j^J w_{kj} a_j$$

and finally $\hat{y}_k = a_k$ which is the output node squished with the softmax activation function. Note that only the output node uses softmax as its activation function.

$$\hat{y}_k = a_k = f(z_k) = \frac{e^{z_k}}{\sum_{k'}^K e^{z_{k'}}}$$

The multiple classes cross-entropy cost function with its dependencies is defined as:

$$C = -\sum_{k=1}^K y_k \ln[\hat{y}_k(z_k(w_{kj}, a_j(z_j(w_{ji}, x_i))))]$$

Where the notation $\hat{y}_k(z_k(w_{kj}, a_j(z_j(w_{ji}, x_i))))$ means that $\hat{y}_k$ is a function of z_k and z_k is a function of w_{kj}, a_j, etc. Futhermore the expanded sum gives the following result:

$$= -y_1 \ln(\hat{y}_1) - y_2 \ln(\hat{y}_2) - ... - y_K \ln(\hat{y}_K)$$

looking at the dependencies of the expanded sum:

$$= -y_1 \ln[\hat{y}_1(z_1(w_{1j}, a_j(z_j(w_{ji}, x_i)))))] - y_2 \ln[\hat{y}_2(z_2(w_{2j}, a_j(z_j(w_{ji}, x_i))))]$$

$$-... - y_K \ln[\hat{y}_K(z_K(w_{Kj}, a_j(z_j(w_{ji}, x_i))))]$$

Hence to be able to calculate the derivative of the cost function with respect to the weight w_{ji}, all the cost function error nodes/terms need to be included. This can be done by taking the sum from 1 to the k'th chain rule of cost function. This comes from the fact that all output nodes are dependant on the same weight w_{ji}, that is connecting the i'th input node to the j'th hidden node. Such that the expression will look as such:

$$\frac{\partial C}{\partial w_{ji}} = \frac{\partial C}{\partial z_1}\frac{\partial z_1}{\partial a_j}\frac{\partial a_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ji}} + \frac{\partial C}{\partial z_2}\frac{\partial z_2}{\partial a_j}\frac{\partial a_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ji}} + ... + \frac{\partial C}{\partial z_K}\frac{\partial z_K}{\partial a_j}\frac{\partial a_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ji}}$$

Compactly the chain rule expression will look as such:

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

The partial derivative between the cost function and z_k is known from the previous assignment to be:

$$\frac{\partial C}{\partial z_k} = -(y_k - \hat{y}_k) = \delta_k$$

The partial derivative of z_k with respect to a_j is:

$$\frac{\partial z_k}{\partial a_j} = \frac{(\sum_j^J w_{kj} a_j)}{\partial a_j} = w_{kj}$$

The partial derivative of a_j with respect to z_j is:

$$\frac{\partial a_j}{\partial z_j} = \frac{\partial(\frac{1}{1+e^{-z_j}})}{\partial z_j} = \frac{e^{-z_j}}{(1+e^{-z_j})^2} = f'(z_j)$$

The partial derivative of z_j with respect to w_{ji} is:

$$\frac{\partial z_j}{\partial w_{ji}} = \frac{\partial(\sum_i^I w_{ji} x_i)}{\partial w_{ji}} = x_i$$

Resulting in:

$$\frac{\partial C}{\partial w_{ji}} = \sum_k \delta_k w_{kj} f'(z_j) x_i$$

Taking out constant factors outside of the sum:

$$= \underline{\underline{x_i f'(z_j) \sum_k \delta_k w_{kj}}}$$

$$= \delta_j x_i$$

$$\implies \delta_j = \frac{\partial C}{\partial z_j} = \sum_k \frac{\partial C}{\partial z_k}\frac{\partial z_k}{\partial a_j}\frac{\partial a_j}{\partial z_j} = \underline{\underline{f'(z_j)\sum_k \delta_k w_{kj}}}$$

and hence

$$\underline{\underline{w_{ji} = w_{ji} - \alpha\delta_j x_i}}$$

1.b Vectorize computation

We begin first with vectorizing the sum of the weights from hidden node j to output node k multiplied by the error of k'th output.

$$\sum_k^K \delta_k w_{kj} = \delta_1 w_{1j} + \delta_2 w_{2j} + \ldots + \delta_K w_{Kj} = \begin{bmatrix}\delta_1 & \delta_2 & \ldots & \delta_K\end{bmatrix}\begin{bmatrix} w_{1j} \\ w_{2j} \\ \vdots \\ w_{Kj}\end{bmatrix}$$

Note that there could be more hidden nodes than 1 so if $j \in (1, 2, 3, ..., J)$ which is more realistic then we would get the following:

$$\sum_k^K \delta_k w_{kj} = \begin{bmatrix}\delta_1 & \delta_2 & \ldots & \delta_K\end{bmatrix}\begin{bmatrix} w_{11} & w_{12} & \ldots & w_{1J} \\ w_{21} & w_{22} & \ldots & w_{2J} \\ \vdots & \vdots & \ddots & \vdots \\ w_{K1} & w_{K2} & \ldots & w_{KJ}\end{bmatrix} := \underline{\underline{\mathbf{\Delta}_k^\intercal \mathbf{W}_{kj}}}$$

Where we denote the transposed delta vector and weight matrix as $\mathbf{\Delta}_k^\intercal$ and $\mathbf{W}_{kj}$ respectively. More over, we use this information to vectorize the update rule for the weight matrix from the input layer to the hidden layer by first expanding $\mathbf{\Delta}_k^\intercal \mathbf{W}_{kj}$ and transposing it.

$$(\mathbf{\Delta}_k^\intercal \mathbf{W}_{kj})^\intercal = \begin{bmatrix} \delta_1 w_{11} + \delta_2 w_{21} + \ldots + \delta_K w_{K1} \\ \delta_1 w_{12} + \delta_2 w_{22} + \ldots + \delta_K w_{K2} \\ \vdots \\ \delta_1 w_{1J} + \delta_2 w_{2J} + \ldots + \delta_K w_{KJ} \end{bmatrix} \tag{1.1}$$

From this we can define δ_j in a vectorized format by doing a hadamard product/elementwise matrix multiplication of the transposed vector (1.1) with the derivative of the hidden layer activation function $f'(z_j)$. Note that to seperate the deltas that are part of $\delta_k = -(y_k - \hat{y}_k)$ we superscript a k on the top as such δ_K^k, whilst delta of the hidden layer gets a superscript δ_J^j. Suppose that all equations with deltas without a superscript up until this point is of the type δ_K^k.

$$\mathbf{\Delta}_j := \begin{bmatrix} \delta_1^j \\ \delta_2^j \\ \vdots \\ \delta_J^j \end{bmatrix} = \begin{bmatrix} f'(z_1) \\ f'(z_2) \\ \vdots \\ f'(z_J) \end{bmatrix} \odot \begin{bmatrix} \delta_1^k w_{11} + \delta_2^k w_{21} + \ldots + \delta_K^k w_{K1} \\ \delta_1^k w_{12} + \delta_2^k w_{22} + \ldots + \delta_K^k w_{K2} \\ \vdots \\ \delta_1^k w_{1J} + \delta_2^k w_{2J} + \ldots + \delta_K^k w_{KJ} \end{bmatrix}$$

Note this simple elementwise product representation is assuming we only do this on one picture. Otherwise both the vectors would be matrices with n columns.

$$\mathbf{\Delta}_j = \begin{bmatrix} \delta_1^j \\ \delta_2^j \\ \vdots \\ \delta_J^j \end{bmatrix} = \begin{bmatrix} f'(z_1)(\delta_1^k w_{11} + \delta_2^k w_{21} + \ldots + \delta_K^k w_{K1}) \\ f'(z_2)(\delta_1^k w_{12} + \delta_2^k w_{22} + \ldots + \delta_K^k w_{K2}) \\ \vdots \\ f'(z_J)(\delta_1^k w_{1J} + \delta_2^k w_{2J} + \ldots + \delta_K^k w_{KJ}) \end{bmatrix}$$

Defining also an x input vector as such where we assume there is $i \in (1, 2, ..., I)$ inputs:

$$\mathbf{X} := \begin{bmatrix} x_1 & x_2 & \ldots & x_I \end{bmatrix}$$

Now we can vectorize the update rule for the weight matrix from input i to hidden layer j denoted by matrix notation $\mathbf{W}_{ji}$:

$$\mathbf{W}_{ji} := \begin{bmatrix} w_{11} & w_{12} & \ldots & w_{1I} \\ w_{21} & w_{22} & \ldots & w_{2I} \\ \vdots & \vdots & \ddots & \vdots \\ w_{J1} & w_{J2} & \ldots & w_{JI} \end{bmatrix} - \alpha \begin{bmatrix} \delta_1^j \\ \delta_2^j \\ \vdots \\ \delta_J^j \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \ldots & x_I \end{bmatrix}$$

$$\mathbf{W}_{ji} = \begin{bmatrix} w_{11} & w_{12} & \ldots & w_{1I} \\ w_{21} & w_{22} & \ldots & w_{2I} \\ \vdots & \vdots & \ddots & \vdots \\ w_{J1} & w_{J2} & \ldots & w_{JI} \end{bmatrix} - \alpha \begin{bmatrix} \delta_1^j x_1 & \delta_1^j x_2 & \ldots & \delta_1^j x_I \\ \delta_2^j x_1 & \delta_2^j x_2 & \ldots & \delta_2^j x_I \\ \vdots & \vdots & \ddots & \vdots \\ \delta_J^j x_1 & \delta_J^j x_2 & \ldots & \delta_J^j x_I \end{bmatrix}$$

Hence the final vectorized update rule for the input to the hidden layer is denoted as such:

$$\mathbf{W}_{ji} = \underline{\underline{\mathbf{W}_{ji} - \alpha \mathbf{\Delta}_j \mathbf{X}}}$$

For the vectorized update from the hidden layer to the output layer:

$$\mathbf{W}_{kj} := \begin{bmatrix} w_{11} & w_{12} & \ldots & w_{1J} \\ w_{21} & w_{22} & \ldots & w_{2J} \\ \vdots & \vdots & \ddots & \vdots \\ w_{K1} & w_{K2} & \ldots & w_{KJ} \end{bmatrix} - \alpha \begin{bmatrix} \delta_1^k \\ \delta_2^k \\ \vdots \\ \delta_K^k \end{bmatrix} \begin{bmatrix} a_1 & a_2 & \ldots & a_J \end{bmatrix}$$

with nodes from the hidden layer are defined as the vector:

$$\mathbf{A} := \begin{bmatrix} a_1 & a_2 & \dots & a_J \end{bmatrix}$$

and the expanded form of this update rule is:

$$\mathbf{W}_{kj} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1J} \\ w_{21} & w_{22} & \dots & w_{2J} \\ \vdots & \vdots & \ddots & \vdots \\ w_{K1} & w_{K2} & \dots & w_{KJ} \end{bmatrix} - \alpha \begin{bmatrix} \delta_1^k a_1 & \delta_1^k a_2 & \dots & \delta_1^k a_J \\ \delta_2^k a_1 & \delta_2^k a_2 & \dots & \delta_2^k a_J \\ \vdots & \vdots & \ddots & \vdots \\ \delta_K^k a_1 & \delta_K^k a_2 & \dots & \delta_K^k a_J \end{bmatrix}$$

And hence the compact format is:

$$\mathbf{W}_{kj} = \underline{\underline{\mathbf{W}_{kj} - \alpha \mathbf{\Delta}_k \mathbf{A}}}$$

Note that the vectors and matrices might require to be transposed in the coding assignment to fit how the dimensions are defined in the template classes of the task. The parameters inside the vectors/matrices remain though the same.

2 Task 2 - Softmax Regression with Backpropagation

2.a

Coding assignment.

2.b

Coding assignment.

2.c Plot

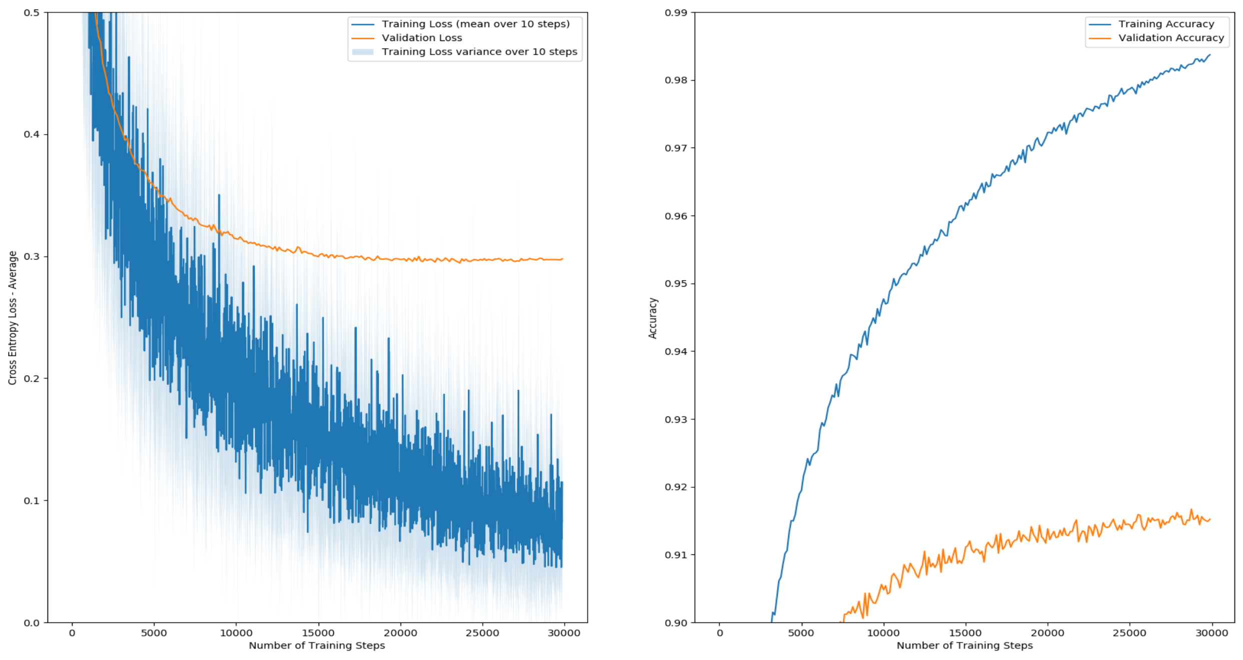


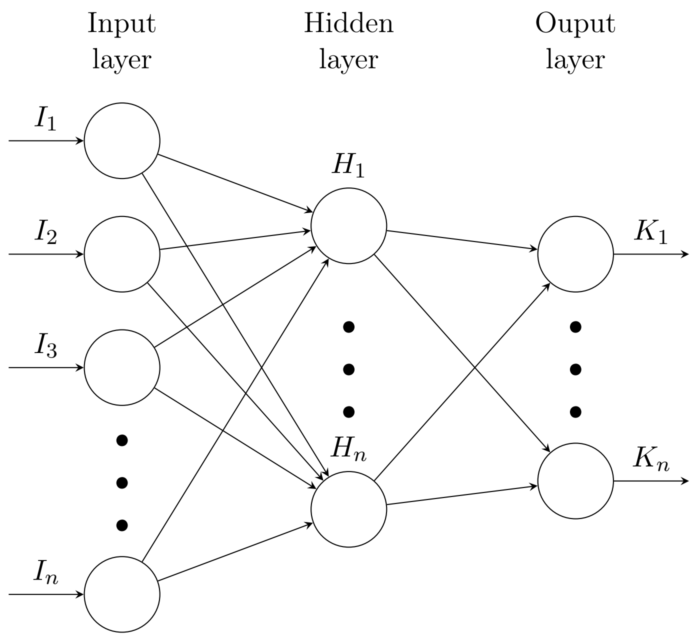
Figure 1: Naive Implementation - Training loss and accuracy.

2.d Model parameter count

The neural network figure below shows the bulk of parameters representing the model with a node edge network. Depending on who you ask the number of parameters is an ambigous question, but here the focus will be to quantify the number of weights in between each layer, ignoring parameters like the learning rate or other hyperparameters.

Our model has three node layers, input, hidden and output at 785 (including bias-node), 64, and 10. Each node is fully connected ie. each node in the ith position is connected to all nodes in the previous layer. This leads to the following calculation:
Connections between input and hidden layer: $= I \cdot H$
Connections between hidden and output layer: $= H \cdot K$

$$Total = H \cdot (I + K) = 64 \cdot (785 + 10) = \underline{\underline{50880}}$$



Note since the bias trick is used, and there is only one bias node on the input layer. It is counted as one of the 785 input parameters, and hence it would be the same as finding $784 \cdot 64 + 1 \cdot 64$ parameters between the input and hidden layer. There is therefore 64 bias terms and 50816 weight terms, and the sum of these gives 50880.

3 Task 3 - Adding the "Tricks of the Trade"

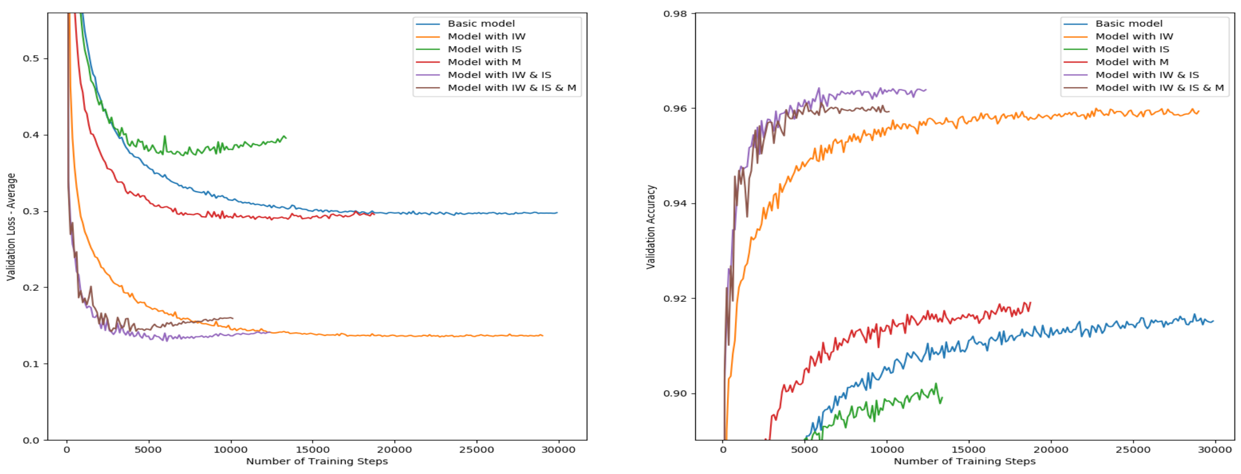


Figure 2: Validation graphs - where the abbreviations IW, IS and M mean Improved Weights, Improved Sigmoid and Momentum respectively.

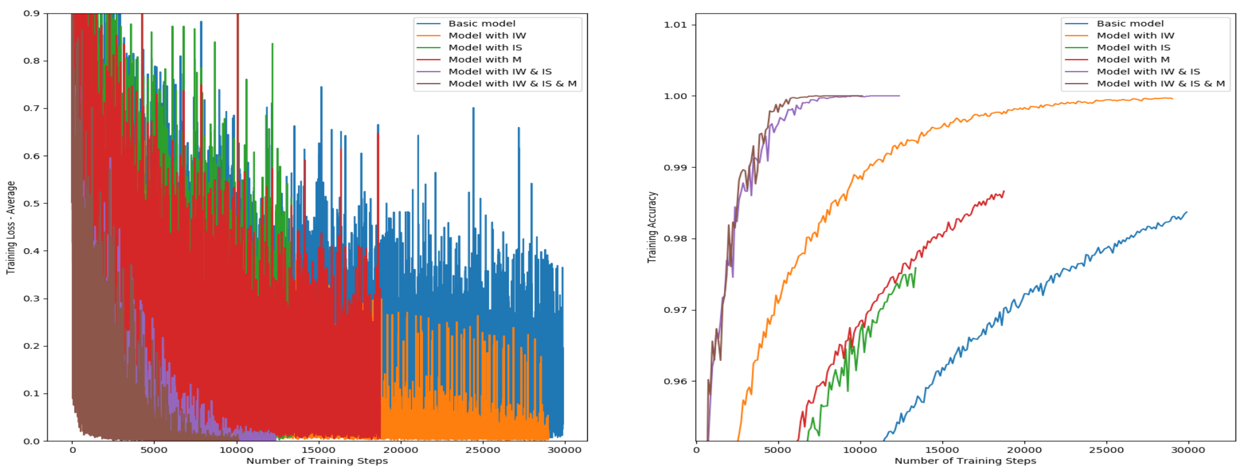


Figure 3: Training graphs, - where the abbreviations IW, IS and M mean Improved Weights, Improved Sigmoid and Momentum respectively.

Tricks of the trade - Comparative statistics					
model	stopping epoch	training loss	training accuracy	validation Loss	validation accuracy
basic	47	.0765	.9837	.2978	.9152
IW	46	.01307	.9996	.1366	.9594
IS	21	.0991	.9759	.3954	.8991
momentum	30	.0620	.9866	.2954	.9191
IS & IW	19	.0034	1.0	.1411	.9639
all	16	.0017	1.0	.1592	.9593

3.a Improved weights

The model with improved weight initialization has a slightly better convergence on the validation set than the basic model (decrease from 47 to 46 epochs). However, it shines most in the dramatic increase in validation accuracy (increase from 91.52% to 95.94%), which also indicates that the model is better at generalizing. A large gap in validation and training accuracy indicates over-fitting, in this case the training accuracy is 99.96%, so some regularization or could be useful.

To understand the significant improvement of the improved weight initialization, we have to first understand what it means to use uniform distribution (basic) versus a normal distribution (improved). In a uniform distribution, the probability of picking a high number, such as -1 or 1, is much higher in the basic weight implementation than in the improved weights implementation. This is because each number between -1 and 1 has the same probability of being picked out, as the probability density plot is just a line. In the improved weights model, the probability density plot in a bell shape with mean 0 and standard deviation of $1/\sqrt{fan-in}$. This means the probability of picking a small number near zero is more likely as the bell curve tends towards its center. Using the same argument, picking a large negative or positive number will be rare. Note that the higher the number of inputs into a neuron, the closer to zero the weights are initialized.

Why is it important that the weights are not initialized close to -1 or 1? There are a few keywords that are important to cover here; activation function, backpropagation and saturation. If one takes a look at the sigmoid activation function in fig. 4a, it is largely flat close to 0 and 1. This makes the gradient less sensitive to which direction it is ideal to move in given a small change dx. In this situation one would say that the neuron is saturated. This phenomenon can make it difficult for the gradient in backpropagation to decide which direction or weights to update to better performance. Note that an overly small gradient caused by saturation will also make a minuscule change to the weight update rule. One may begin to think that a zero weight initialization would be a good idea to completely avoid saturation, but this also causes problems during backpropogation as it may struggle to identify which weight contributed to a correct or incorrect classification. The neural net is forming an opinion, it needs to stand by its core beliefs (some large weights), but it also need to have an open mind that it might be wrong (some small weights).

Hence we can conclude that since the improved weight initialization is better as it accounts for the number of input neurons, and ends up lowering the risk of activation function saturation significantly. This gives an overall improvement in performance in terms of validation accuracy/loss which is also an indication of a more generalized model. It also makes for better convergence, as the weight parameters can learn more smoothly, with less saturation and big gradients when justified. In return, this allows for a higher learning rate, as per lecun et. al. - section 4.6. This theory is backed up by the validation loss/accuracy plot found in fig. 2

3.b Improved weights & Improved sigmoid

The training time was reduced from epoch 46 to 19, and the validation accuracy was increased from 95.94% to 96.39% when using both improved weights and improved sigmoid, giving the best performance of all the models. It is however, suprising to see that IS & IW give a final training accuracy of 100%. Without early stopping, we could be risking over-fitting the model, and some regularization or other might be useful.

By switching to an improved activation function activations tend to stay within its "good zone" (non-zero gradient), where we avoid saturation of the sigmoid function. The next two points are arguments from lecun et. al.: As illustrated in the fig. 4, the mean is not centered around zero with the logistic function, as it is for the hyperbolic function. Centering around zero, prevents the gradients from becoming disproportionately large. Another point raised is that the slopes around the extremities of the function are almost, but not equal to zero. This leaves the gradients little room to be sensitive to differences in activation levels and worsen saturation.

It is also worth noting that convergence/learning rate usually increases if the input variables over the training set are normalized. The TanH activation function contributes because of how the hyperbolic function maps input values into output values. TanH is symmetric around the origin, just like the input layer values are after normalization. In other words, this streamlined mapping of the TanH activation function allows for normalized inputs to stay closer to zero in the output, reducing the summed up Z-value - the input in the next layer of neurons. The combination of using TanH and input normalization allow for activations to remain closer to zero. The sigmoid function activations are also purely positive and may be a driver for saturation, while the TanH produces activations that are both negative and positive, which can cancel each other out when summed.

In conclusion, normally distributed weights, normalized inputs, and using activation function TanH, encourages output mappings with similar characteristics to the input values and give a significant improvement compared to the base model. However, the most important contribution from the improved sigmoid, is the fact that it converges faster than the sigmoid. This is backed up both by our implementation in fig. 2 as well as the theory discussed above. It is also important to point out that the activation function TanH on its own, seems to reduce generalization on the training set in fig. 3 to the validation set in fig. 2. In fact, it performs worse than all the other models ending up with a validation accuracy of 89.91%. This leads us to conclude that uniformly distributed weights in combination with the improved sigmoid is over-fitting on the training data and requires tuning as the training accuracy is 97.59%.

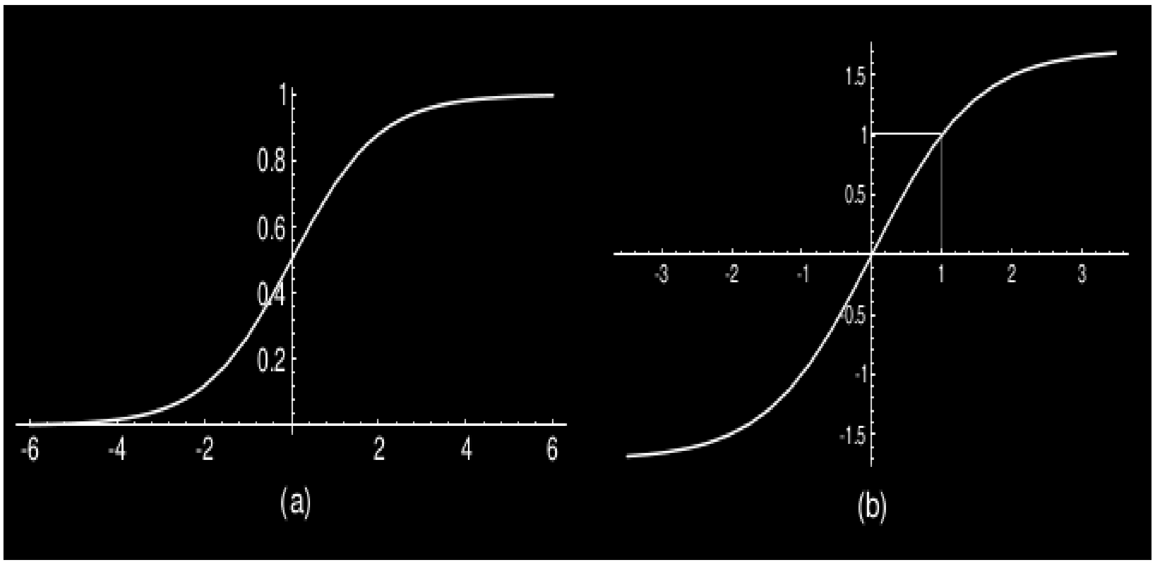


Figure 4: Two activation functions, figure from lecun et. al. Sigmoid (a) and improved sigmoid - TanH (b)

3.c Improved Weights & Improved Sigmoid & Momentum

The training time was reduced from 19 to 16 epochs, and the accuracy was decreased from 96.39% to 95.93% for the model with all improvements. Although performance might have gotten slightly worse, it is theoretically justified that momentum accelerates training, smooths oscillations and gives a significant improvement in convergence. Which are observable in the figure over training set (fig. 3) and the validation set (fig. 2). Signs of over-fitting are also observed in this model just like the IS & IW model, as it reaches 100% training accuracy and may be starting to memorize it, leaving the validation stats lagging behind. One can therefore expect even worse performance on the validation set if the training continued without early stopping.

Furthermore momentum provides a state-of-the-art optimization for stochastic gradient descent. The gradient descent models without momentum average the gradients of the batch and greedily navigate in the complex solution space landscape based on only instantaneous slope information that may translate to changing its mind on direction at every step. Momentum provides "inertia" or "short term memory" to the gradient, such that it can escape local minimums and move towards more optimal solutions. Because of this property of momentum, it smooths the oscillating behavior of the gradient and convergences faster as the optimal solutions are found significantly quicker.

It is how ever important to note that momentum is great for improving the effective learning rate, but is not the main reason for better validation accuracy/loss. In fact in our case, it has been observed that adding momentum on top of IS & IW, reduces performance over the validation dataset. Using a metaphor for a n-dimensional solution space, momentum naturally takes smaller steps as it nears its final local/global minima (flatter gradient), it may have prevented the solution from settling quietly, like a ball rolling overshooting the bottom of a bowl. This analogy is observed in the performance of the validation accuracy of the model with all vs the validation accuracy of the model with only IS & IW. Where the validation accuracy is slightly reduced with momentum.

4 Task 4 - Experiment with network topology

4.a 32 Neurons - (Code found in file task4ab.py)

The accuracy decreased for both the training and validation datasets when we set the number of neurons to 32 compared to the base model with 64 neurons (see fig. 5). This could be an indication of insufficient neurons to train the model adequately. Hence using too few neurons will run the risk of under-fitting the model and therefore also perform worse on the validation set. In short, if the number of neurons is made even smaller than 32 then the validation accuracy/loss will show even worse performance further justifying the argument about the under-fitted model.

See section 4.b, where both neuron configurations are discussed together (32 and 128 neurons).

4.b 128 Neurons - (Code found in file task4ab.py)

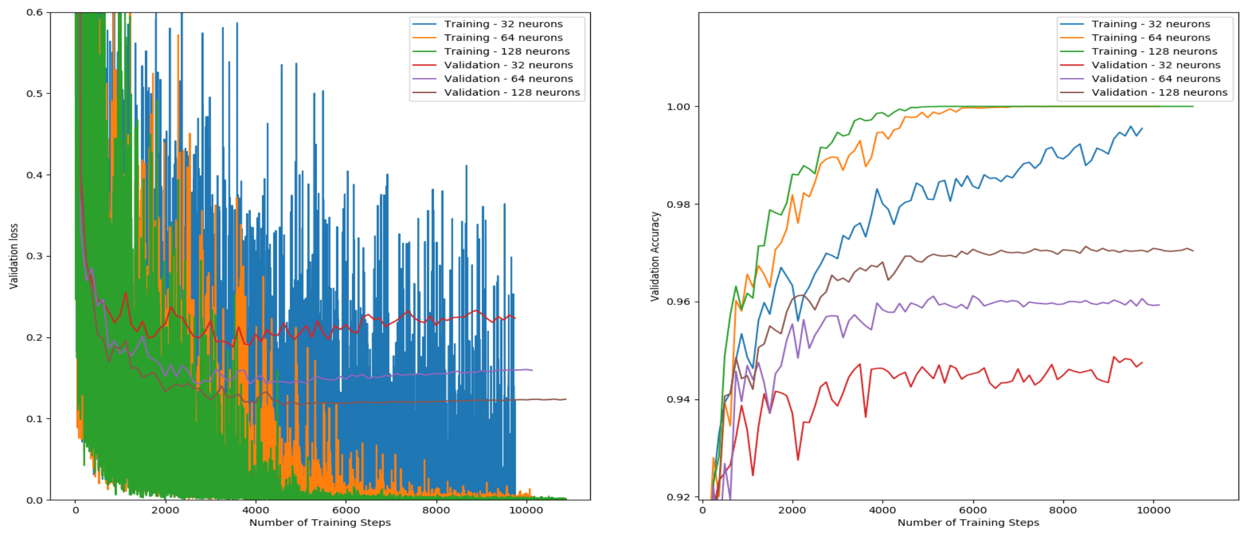


Figure 5: Comparison between models of 32, 64 and 128 neurons.

From fig. 5, one can observe an expected improvement in training accuracy in the model with 128 neurons, where it can reproduce answers with 100% accuracy after approximately 5000 training steps. On the other hand, we see the opposite relationship for the validation accuracy, it is not able to generalize as well. This is a classical dilemma presented by over-fitting, and this is why we have a validation set in the first place. The model seems to be doing great in the training set, but it is in fact just "memorizing" the answers, therefore not generalizing as we increase model complexity. To counteract over-fitting in the more complex model, it may be useful to introduce regularization or another technique promoting generalization.

It is also interesting to see that the less neurons there are on the hidden layer, the faster the early stopping kicks in. Such that the model with 128 neurons has the most training steps and the model with 32 neurons uses the least time. Therefore it gives a general sense of how the number of neurons also affects training time.

In conclusion too few neurons and the training will be fast, but there is not enough neurons such that the model is under-fitted. How ever using too many neurons, the opposite problem happens where there is an abundance of neurons compared to the limited data causing over-fitting with slower training speed. Obviously some sort of middle ground between the two neuron configurations must be reached. This goes to show that network topology plays as big a role in performance as tuning any other parameter in a neural network. Hence one should be observant of the training data size, and experiment with neuron sizes of the hidden layer such that it is adequate for the task that is supposed to be solved.

4.c

Coding assignment.

4.d Approximate parameters - (Code found in file task4d.py)

In task 3 we do not change the network and so we have the same network parameters as the one calculated in task 2d. With layers [785 64 10] there are totally 50880 parameters. Now with the requirement of two hidden layers of equal size, having approximately the same number of parameters as the network from task 3. We calculate the following:

$$[785\ x\ x\ 10] \implies \underbrace{784 \cdot x + x \cdot x + x \cdot 10}_{\text{weight parameters}} + \underbrace{1 \cdot x}_{\text{bias parameters}} = 50880$$

Solving this second degree polynomial gives us:

$$x^2 + 795x - 50880 = 0$$

$$x = 59.5 \lor x = -854.5$$

We decide therefore to have two hidden layers with 60 nodes per hidden layer, such that the format is [785 60 60 10]. This gives a total parameter of:

$$785 \cdot 60 + 60 \cdot 60 + 60 \cdot 10 = \underline{\underline{51300}}$$

The total parameter difference between the network of task 3 and the following network is 420 parameters.

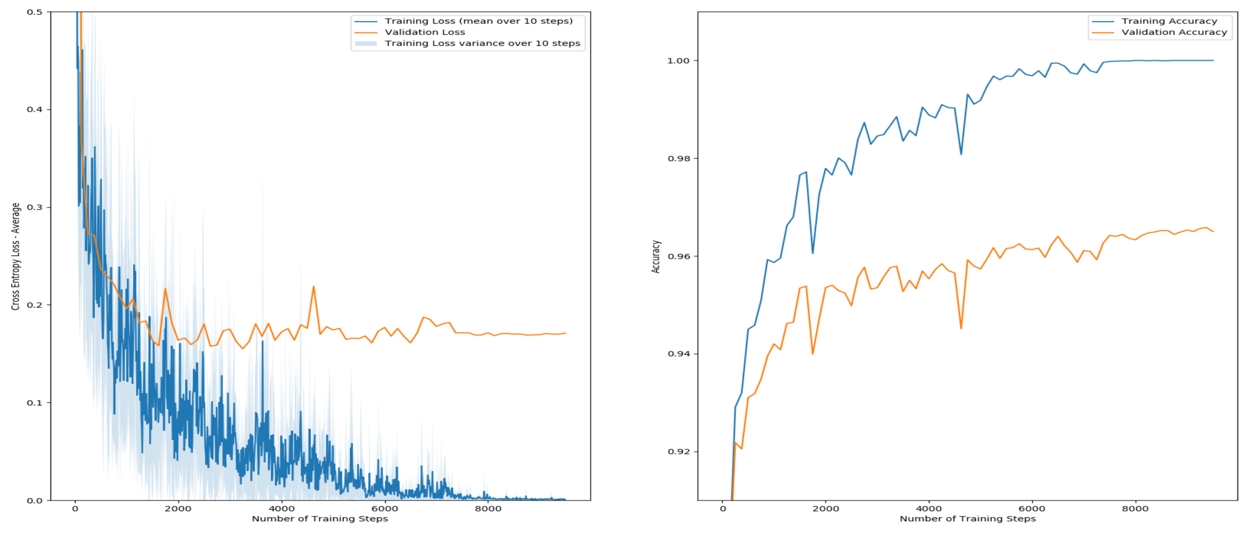


Figure 6: Training and validation loss/accuracy for model with 2 hidden layers of size 60 each and an approximately same parameter amount as the model fram task 3.

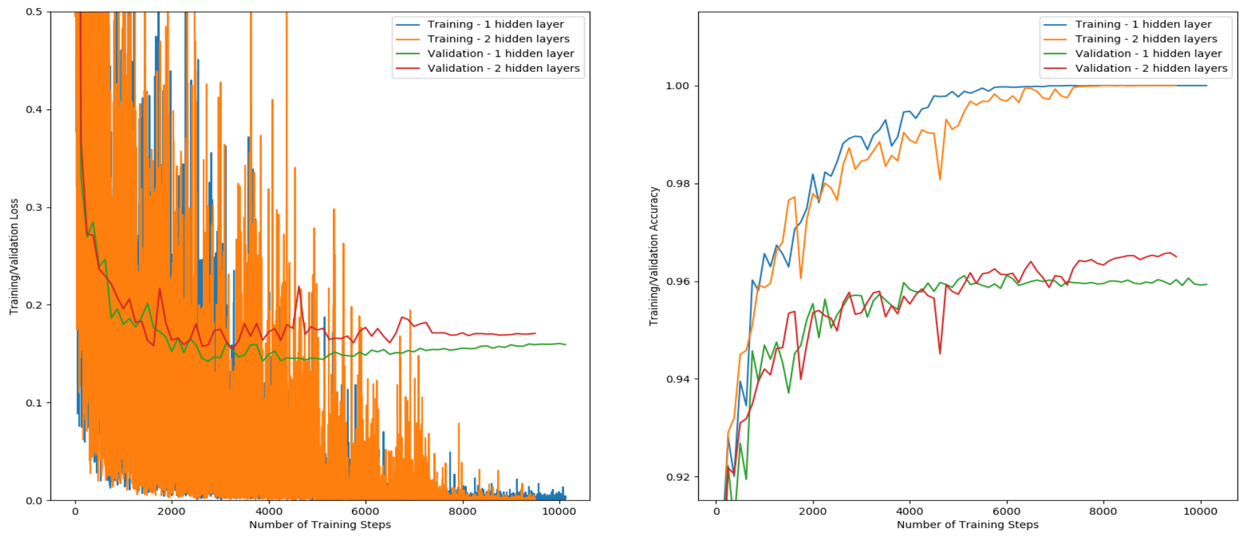


Figure 7: Comparison of model from task 3 (1 hidden layer of size 64) and task 4d (2 hidden layers of size 60).

From observation of fig. 7 it seems like the model with 1 and 2 hidden layers have similar performance both in training and validation statistics. This intuitively makes sense as the number of weights and biases are preserved in the two vs three layers. It may seem like the number of neurons and layers are irrelevant, as long as the parameter number is the same in the two different network configurations. However this is not the case, as different layered networks with approximately the same parameters, will have different back-propagation and access to different neuron activation combinations that represent the difference between a shallow vs deep network. More abstractly the activation functions are what gives a model its ability to make non-linear function mappings. An activated node in a neural network is often compared to a single "soft" universal logic gate. Changing the number of neurons will surely affect the type of logical gate combinations that the network can express. The number of neurons are like the width of a logic gate graph, and the number of layers is like the length of such a graph.

Another observation is that the number of training epochs has gone from 16 epochs with the 1-hidden layer model to 15 epochs with the 2-hidden layer model. The training accuracy for the 1-layer model reaches its convergence at 96% in around 4000 training steps, whilst the the 2-layer model uses much longer time. This may indicate that back-propogation is affected by the number of layers, and is discussed in more detail in the next section, when comparing with a 10-layer network.

4.e 10 Hidden layers - (Code found in file task4e.py)

The deep neural network (10-hidden layered network) is performing much worse than the network with 1 and 2-hidden layers, in terms of the validation set accuracy and loss. One can also see that the learning speed of the deep neural network is much slower than the other two. Since all training parameters and hyper parameters remain the same for all the networks, one could conclude that the reason is related to back-propagation. This is the vanishing gradient problem. The back-propagation is meant to spread from layer K, layer by layer, until it reaches the first hidden layer. As the neural nets depth increases the effective weight updates become lost on the way (like a game of telephone). The learning speed therefore becomes slower the more hidden layers are added and it explains why the deep neural network trains for longer. The other problem is that a deep network ends up under-fitting on the training set with the given parameters, and for that reason performs a lot worse on the validation set. This is because it is struggling with convergence.

Another observation is that the number of training epochs has gone from 16 epochs with the 1-hidden layer model to 29 epochs with the 10-hidden layer model. Both plots in figure 8 show that that convergence for deeper networks tend to take more time than shallow networks. As an example for the shallow models, the curves explode upwards in the accuracy plot and converges quickly, and vice versa for the deep models.

In conclusion when implementing neural networks for training a simple model it is sufficient to use a low complexity neural network with 0 or 1 hidden layers. Such that anything that resembles a deep neural network should be reserved for complex datasets.

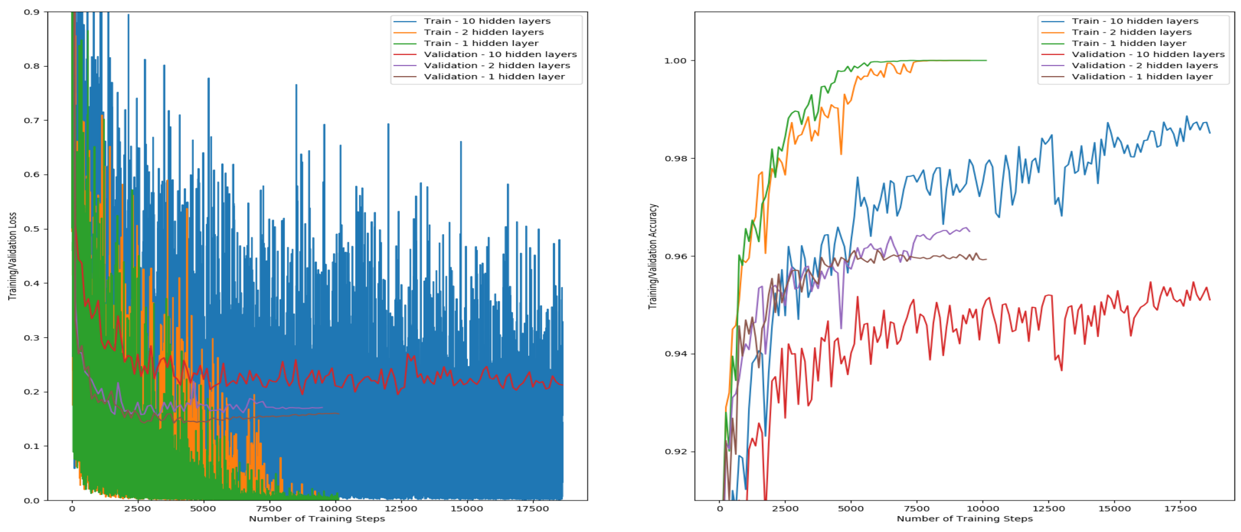


Figure 8: Comparison of models with 1,2 and 10 hidden layers.