TDT4265 - Computer Vision and Deep Learning Assignment 2

Thomas Aven, Lasse Eggen

February 14, 2019

Task 11

- We were a bit confused about whether we were supposed to prove BP1 or BP4 (or both) from Nielsen's book, so we did both:

BP4 assume squared error (it's possible to prove with a general cost function, but we thought this to be sufficient):

$$\frac{\partial C}{\partial w_{j,i}} = \frac{\frac{1}{2}\sum_k (a_k - t_k)^2}{\partial w_{j,i}} = \sum_k (a_k - t_k)\frac{\partial a_k}{\partial w_{j,i}} \quad \text{(chain rule)}$$

$$= \sum_k (a_k - t_k)\frac{\partial f(z_k)}{\partial w_{j,i}} \Rightarrow z_k = a_j w_{k,j} = \frac{\partial f(z_k)}{\partial w_{j,i}} = f'(z_j) w_{k,j} \frac{\partial z_j}{\partial w_{j,i}}$$

$$= f'(z_j) w_{k,j} \frac{\partial a_i w_{j,i}}{\partial w_{j,i}} = f'(z_j) w_{k,j} a_i$$

put this result back into this for:

$$\frac{\partial C}{\partial w_{j,i}} = \sum_k (a_k - t_k) f'(z_k)(f'(z_j) w_{k,j} a_i)$$

$$= f'(z_j) a_i \sum_k (a_k - t_k) f'(z_k) w_{k,j}$$

$$= f'(z_j) a_i \sum w_{k,j} \delta_k = a_i \delta_j$$

$a_i = x_i$ in this case, as it is the input layer.

(the learning rate was forgotten here, but it's just an extra constant that is irrelevant to the derivation).

Task 1.1

BP1 assuming we know (proven in BP4, and in a specific case in assignment 1).

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

$$= \alpha \frac{\partial C}{\partial z_j} a_i$$

by chain rule:

$$= \alpha a_i \left(\sum_k \frac{\partial C}{\partial z_k} \frac{\partial z_k}{\partial z_j} \right) = \alpha a_i \left(\sum_k \frac{\partial z_k}{\partial z_j} \frac{\partial C}{\partial z_k} \right)$$

$$= \overset{\alpha a_i}{\left(\sum_k \frac{\partial z_k}{\partial z_j} \delta_k \right)} = \overset{\alpha a_i}{\left(\sum \frac{\partial (w_{kj} \cdot a_i)}{\partial z_j} \delta_{ik} \right)}$$

$$\overset{\alpha a_i}{=} \sum_k \frac{\partial (w_{kj} \cdot f(z_j))}{\partial z_j} \delta_k$$

$$\overset{\alpha a_i}{=} f'(z_j) \sum_k w_{kj} \delta_k = \delta_j$$

⇒ which is as specified ∎

($a_i = x_i$ again here, as it's the input).

Task 1.2 (squared paper, yay)

There is really no derivation to arrive at these rules, it's fairly trivial to end up using the hadamard product:

BP1: $\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$

the pointwise multiplication (hadamard), we write this in matrix form as: (for the output layer)

$\delta^L = (\nabla_a) \odot \sigma'(z^L)$

↳ partial derivatives as a vector $\left(\frac{\partial C}{\partial a_j^L}\right)$

example with quadratic cost: $\delta^L = (a^L - y) \odot \sigma'(z^L)$

for a layer in terms of the next we get:

$\delta^L = ((w^{L+1})^T \delta^{L+1}) \odot \sigma'(z_3^L)$,

which can be recognized in our python impl:

```
d = derivative * np.multiply(next_layer.weights.T, d)
```

↳ hadamard product for np.ndarray

As for updating the weights according to gradients, we use elementwise matrix subtraction:

```
layer.weights = layer.weights - update
```

↳ lr * gradients

2 - MNIST Classification

```
model = Model()
model.add_layer(64, Activations.sigmoid, size_of_input_layer)
model.add_layer(10, Activations.softmax)
```

2a) Shown above is the code used to create a model of the given network. This is the general idea behind creating any sort of model within the framework we have created (which is modelled after Keras, which we found to be quite easy to use last semester), as to make it as smooth as possible to add layers (in this case only dense FFNN- and dropout layers). The model above contains an input layer, a hidden layer with a sigmoid activation, and an output layer with softmax, i.e. (784, 64, 10).

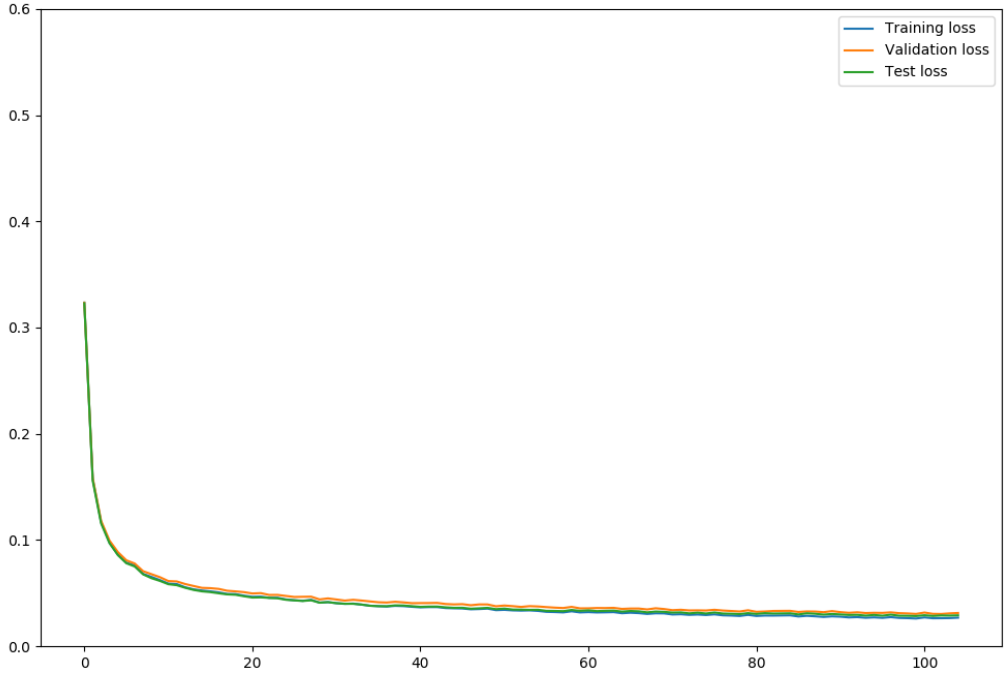
Before training, we split MNIST into a training and validation set. 10% of the dataset is selected as validation set, and thus removed from the training set. Log-cross entropy is used in all cases. No regularization is implemented. Early stopping was not used, as we simply observed the training behaviour over several training sessions and decided that 15 epochs was sufficient. Evaluation is done 20 times per epoch to create a smooth graph of the training development.

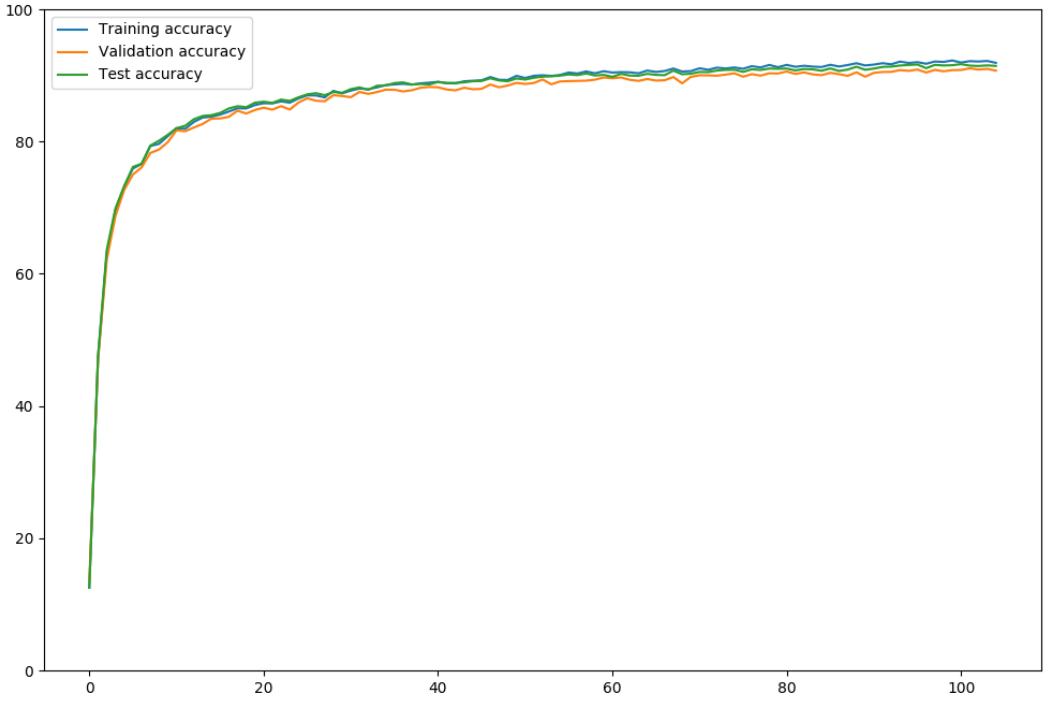
The hyperparameters are given to the model when kicking off training. They are:

```
model.train(mnist, epochs=15, batch_size=128, lr=0.5, evaluate=True)
```

2b) For this task we implemented a `check_gradients` method in the `Model` class. This will do the numerical gradient check over every weight in the network, layer by layer. We let this run for only about 50 input images, as it took quite a while to compute. The maximum absolute difference between the gradient we got from the backpropagation algorithm and the numerical computation was consistently in the range of 10^{-7} to 10^{-8} when using an ϵ of 0.01, making it well within the required big-O (for both the hidden layer and the output layer).

2c) Below are the plots obtained from evaluating the model every 1/20th epoch. The result is very much as expected, reaching an accuracy of about 92% without any tricks.


Training loss
Validation loss
Test loss
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
20
40
60
80
100


Training accuracy
Validation accuracy
Test accuracy
100
80
60
40
20
0
0
20
40
60
80
100

3 - Adding the "Tricks of the Trade"

The labels in the table below represent [B]ase, [S]huffle, [I]mproved Sigmoid, [N]ormal distribution, [M]omentum ($\mu = 0.9$).

Table 0.1: (train_acc, val_acc, test_acc) in epoch [1, 5, 10, 15]

	1	5	10	15
B	(85.9, 85.3, 86.0)	(92.0, 91.6, 91.6)	(93.8, 93.0, 92.8)	(94.8, 93.4, 93.5)
S	(85.6, 85.0, 86.1)	(92.4, 91.0, 91.7)	(94.2, 92.4, 93.0)	(95.3, 93.6, 93.7)
I	(81.9, 81.3, 82.0)	(91.8, 91.0, 91.3)	(93.5, 92.1, 92.9)	(94.6, 93.5, 93.7)
N	(94.5, 94.3, 94.1)	(97.3, 96.1, 96.5)	(98.1, 96.4, 96.9)	(98.8, 96.7, 97.1)
M	(93.8, 93.2, 93.6)	(96.9, 95.4, 96.0)	(98.4, 96.4, 97.0)	(98.8, 96.4, 97.0)

Note that we did not opt to use plots for this (except a select few below), as we found it hard to capture the nuances in a good manner within a single plot (they only differ slightly, so we found sampling the accuracy every 5th epoch to be a sufficient metric). This was done without normalizing the gradients like the loss, which here gave faster convergence.

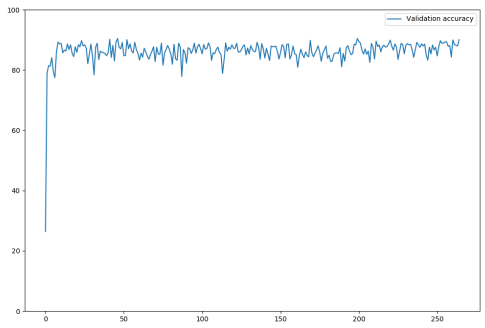
a) Shuffling examples applies mostly to stochastic learning, and perhaps to batch learning, given that the mini batches are small enough that the risk of creating ones that are not representative of the overall dataset is present. Shuffling may in those cases reduce variance, but our results show no noticeable difference from shuffling. This is likely due to the fact that a mini batch size of 128 is big enough to represent the dataset (of 10 different classes) well in general.

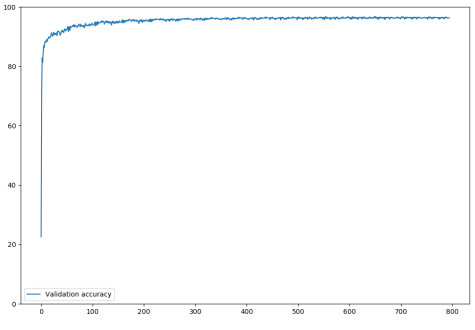
b) The usage of *tanh* did not manifest itself in the results shown above. A common theme when switching to hyperbolic tangent is faster convergence, attributed to its *stronger gradients* that are centered around 0. We saw similar results for the usage of simple FFNNs on MNIST with Keras, where trying tanh over the standard logistic function often made little difference. Another difference between the two is that the logistic function is not zero-centered, as opposed to tanh. Non zero-centered activations make the network prone to reaching all positive or negative weights during training, but the zig-zagging that may be introduced due to this is often mitigated by adding up batches of data.

Considering this, we plotted the trajectories of the accuracies sigmoid and tanh for only the first epoch, but this did not show any definite differences either, so we decided not to include the plots. We believe the reasoning to be as described above, which is similar to what happened in a). As an aside, we did more tests later, after doing normalization differently, which showed tanh to be converging faster than sigmoid. This faster convergence should be what you are expected to see in this task.

c) Changing the initialization of weights to use fan-in definitely helped, as shown in the table above. This is as expected, as the training set has been normalized, and the improved sigmoid is used, which is what is described in section 4.6 of *Efficient Backprop*.

d) For momentum we implemented classic momentum and used a weight $\mu = 0.9$ for the most recent gradient. This is done primarily to have the gradient descent algorithm get out of local minima, often thought of as a rolling marble escaping a basin. With our original learning rate, we saw huge oscillations in the accuracy and loss (after just 5 epochs), perhaps signaling that the marble has too much momentum and overshoots. Lowering the learning rate from 0.5 to 0.05 gave much smaller oscillations, and a better result, as shown in the images below. Momentum did not increase the accuracy on the test and validation from task **c)**, but we are certain that this is more likely to be because of the fact that the network is starting to struggle with overfitting, as we will elaborate on in task 5.





4 - Experiment with network topology

For this task we used the same hyperparameters as in task **3**, as well as the tricks of the trade, except momentum. This task was particularly easy to do with the way we had created our networks, as changing the units and adding layers is as simple as it is with Keras.

Table 0.2: (train_acc, val_acc, test_acc) in epoch [1, 5, 10, 15]

Neurons	1	5	10	15
5	(83.2, 83.0, 83.0)	(85.0, 84.0, 84.8)	(85.9, 85.1, 85.1)	(85.6, 84.1, 84.7)
10	(89.5, 89.3, 89.4)	(91.1, 90.3, 90.8)	(91.5, 90.7, 90.9)	(92.4, 91.6, 91.5)
20	(91.1, 91.3, 91.3)	(93.3, 93.1, 93.3)	(95.1, 94.6, 93.9)	(95.7, 95.0, 94.7)
32	(92.7, 92.3, 92.6)	(94.9, 93.9, 94.4)	(97.1, 95.3, 95.8)	(97.4, 95.2, 95.9)
64	(94.5, 94.3, 94.1)	(97.3, 96.1, 96.5)	(98.1, 96.4, 96.9)	(98.8, 96.7, 97.1)
128	(93.7, 93.3, 93.6)	(97.6, 96.4, 96.6)	(98.2, 96.8, 96.9)	(99.5, 97.3, 97.5)
60, 60	(94.4, 93.8, 94.1)	(97.7, 96.9, 96.5)	(98.5, 96.9, 96.8)	(98.8, 97.0, 96.8)

The table above shows the results for this task, again opting for a table instead of multiple big plots. Each entry in the table is for an amount of neurons in the hidden layer of the network, except for the last one, which is a network with two hidden layers, both with 60 hidden units.

a) We see that halving the number of hidden units inhibits the learning of the network quite drastically. By lowering the amount of hidden units even more, we notice that the effect is that we are trying to force too much information through a network that has too little complexity. The layer with a low amount of units becomes an information bottleneck.

b) Doubling the amount of units increases the validation and test accuracy slightly, but more importantly, it drives the training accuracy through the roof; the network is overfitting quite hard to the training set. By seeing a 99.5% accuracy on the training set with 128 hidden units, we noticed that using a number of hidden units that is too large is extremely prone to overfitting.

c) With one hidden layer the architecture has $764 * 64 + 10 * 64 = 49536$ weights. To approximate this number of weights, we find that 60 neurons in each of the hidden layers gives about the same total number of weights in the network, $764*60+60*60+60*10 = 50040$. This gave a performance that was about the same as the network with a single hidden layer of 64 units, telling us that extra layers are not necessarily better. The effect of adding layers may help with detecting more features within a dataset, but is also prone to cause overfitting. Thus we conclude that we have a reached a point where the main issue is overfitting, and changing the layout of the network is unlikely to help.

5 - Bonus

We have implemented dropout layers for this task, and added the ReLU activation function, as we have found it to work quite well in previous endeavours. We also added Xavier initialization of weights, which originally was created to solve the issue of vanishing gradients, but still works quite well with ReLU (even though vanishing gradients, well, vanish with ReLU). Decay of learning rate is also added to the training method.

We found out that using dropout for training increases the training time, both by increasing the amount of matrix multiplications, the addition of remasking, as well as increasing the convergence time of the network.

We saw much of the same results from using the code we have implemented in this assignment in the mini-project using Keras in TDT4195 last semester. It is hard to push the accuracy much further without data augmentation, even by using dropout.

Dropout shows a slight increase in performance (we reached as high as 98% accuracy on the validation/testing sets during experimentation), especially showing less overfitting, but still struggles to get close to the 99% accuracy a CNN will perform at quite easily. The only way we pushed an FFNN to 99% accuracy with Keras was by augmenting the MNIST data set with `ImageDataGenerator`. We could steal some code online to augment MNIST, but figured that we would rather spend time learning about other interesting networks, as micro optimizing FFNNs for MNIST is fairly tedious and unrewarding.