Course: Deep Learning **([16-11-2021])**

Submission Assignment #1

Coordinator: Jakub Tomczak *Name:* Marton Fejer, *Netid:* [2694002]

1 Question answers

Question 1 *Work out the local derivatives of both, in scalar terms. Show the derivation. Assume that the target class is given as an integer value.*

The derivative of the softmax function can be formalized as in (1), whereby the derivative of yi can be w.r.t. any o, so o_j.

$$\frac{\partial y_i}{\partial o_j} = \frac{\partial}{\partial o_j} \cdot \frac{e^{o_i}}{\sum_{k=1}^{n} \left(e^{o_k}\right)}$$

Figure 1: equation (1)

Because of the quotient rule for derivatives and because the sum of derivatives (as in the sum equation), we get:

$$\frac{\partial\ y_i}{\partial\ o_j} = \frac{e^{o_i} \cdot \sum_k^n \left(e^{o_k}\right) - e^{o_i} \cdot e^{o_j}}{\left(\sum_k^n \left(e^{o_k}\right)\right)^2}$$

Figure 2: equation (2)

if i = j, this equation can be shortened to:

$$\frac{\partial y_i}{\partial o_i} = \frac{e^{o_i}}{\sum_k^n \left(e^{o_k}\right)} \cdot \frac{\sum_k^n \left(e^{o_k}\right) - e^{o_i}}{\sum_k^n \left(e^{o_k}\right)} = y_i \cdot \left(1 - y_i\right)$$

Figure 3: equation (3)

if i ≠ j, (2) can be shortened to (4), then (5):

$$\frac{\partial\ y_i}{\partial\ o_j} = \frac{0 \cdot \sum_k^n \left(e^{o_k}\right) - e^{o_i} e^{o_j}}{\left(\sum_k^n \left(e^{o_k}\right)\right)^2} = \frac{-e^{o_i} e^{o_j}}{\left(\sum_k^n \left(e^{o_k}\right)\right)^2}$$

Figure 4: equation (4)

$$= -\frac{e^{o_i}}{\sum_k^n \left(e^{o_k}\right)} \cdot \frac{e^{o_j}}{\sum_k^n \left(e^{o_k}\right)} = -y_i \cdot y_j$$

Figure 5: equation (5)

The derivative of the cross-entropy loss is much more straightforward, however only because we are already dealing with the short-hand version, whereby the sum of the output's losses adds up to just the negative log of the network's output for the true class (so yc). As this is the case, the derivative of the loss function is just the derivative of -log(yc):

$$\frac{\partial l}{\partial y_i} = \frac{\partial}{\partial y_i}\Big(-\log\big(y_c\big)\Big) = -\frac{1}{y_c}$$

Figure 6: equation (6)

Question 2 *Work out the derivative of loss wrt to the outputs. Why is this not strictly necessary for a neural network, if we already have the two derivatives we worked out above?*

It is not necessary to work out the derivative because it can be calculated with the multivariate chain rule, whereby the following is true:

$$\frac{\partial l}{\partial o_j} = \frac{\partial l}{\partial y_i} \cdot \frac{\partial y_i}{\partial o_j}$$

Figure 7: equation (7)

Thus, if i = j, we get:

$$\begin{aligned}\frac{\partial\, l}{\partial\, y_i} \cdot \frac{\partial\, y_i}{\partial\, o_i} &= -\left(\frac{1}{y_i}\right) \cdot y_i \cdot \left(1 - y_i\right) \\ &= -1 + y_i\end{aligned}$$

Figure 8: equation (8)

And if i $\neq$ j, we get:

$$\begin{aligned}\frac{\partial\, l}{\partial\, y_i} \cdot \frac{\partial\, y_i}{\partial\, o_j} &= -\left(\frac{1}{y_i}\right) \cdot \left(-y_i\right) \cdot y_j \\ &= y_j\end{aligned}$$

Figure 9: equation (9)

If y_i can be considered a one-hot-encoded vector of the target, than the derivative of the loss writ to the outputs from softmax can be generalized to:

$$\frac{\partial\, l}{\partial\, o_i} = o_i - y_i$$

Figure 10: equation (10)

Question 3 *Implement the network drawn in the image below, including the weights. Perform one forward pass, up to the loss on the target value, and one backward pass. Show the relevant code in your report. Report the derivatives on all weights (including biases). Do not use anything more than plain python and the math package.*

The code is included with this report, and will produce the same output as can be seen in figure 11:

```
Run:  main
C:\Users\Marton\Code\PycharmProjects\DL_VU\venv\Scripts\python.exe C:/Users/Marton/Code/PycharmProjects/DL_VU/main.py
Layer 1 weight gradients: [[0.0, 0.0, 0.0], [-0.0, -0.0, -0.0]]
Layer 1 bias gradients  : [0.0, 0.0, 0.0]
Layer 1 weights         : [[1.0, 1.0, 1.0], [-1.0, -1.0, -1.0]]
Layer 1 bias            : [0.0, 0.0, 0.0]

Layer 2 weight gradients: [[-0.44039853898894116, 0.44039853898894116], [-0.44039853898894116, 0.44039853898894116], [-0.44039853898894116, 0.44039853898894116]]
Layer 2 bias gradients  : [-0.5, 0.5]
Layer 2 weights         : [[1.0440398538988942, 0.9559601461011059], [-0.9559601461011059, -1.0440398538988942], [-0.9559601461011059, -1.0440398538988942]]
Layer 2 bias            : [0.05, -0.05]

Process finished with exit code 0
```

Figure 11: Output of one forward pass and one backward pass through the network implemented for q3 + q4

Question 4 *Implement a training loop for your network and show that the training loss drops as training progresses*
The code is included with this report, and will produce a similar output to this:

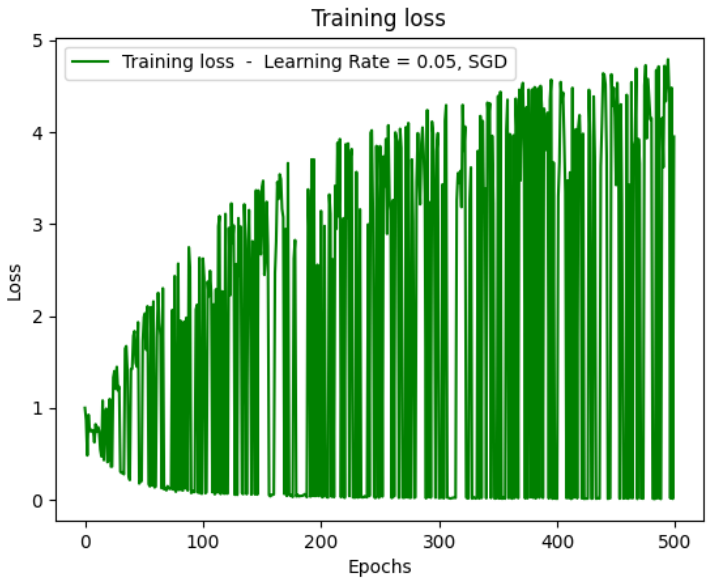

Figure 12: Training Loss after 500 epochs for the network implemented for q3 + q4, with a learning rate of 0.05 and using Stochastic Gradient Descent.

While the loss is not clearly decreasing, it is visible nonetheless that about half of the targets have an extremely low loss value, suggesting that the network is "learning", but it cannot successfully adjust weights to account for all of the possible inputs. What can be said about the data in that case, is that either there is nothing predictable about it, with respect to the target class. Another explanation could be that the network is not sufficiently deep to capture all the information in the inputs.

2 Problem statement

For this assignment, a multilayer perceptron (a fully connected neural network) is implemented which is trained (i.e. the weights of the network are adjusted) by backpropagation of the gradient of cross entropy loss w.r.t. the weights of the network. It is to be evaluated whether this rather simple network can be efficiently used for a multiclass classification problem.
The error, or the cost function which is to be minimized, of the network is given by the cross entropy loss function:

$$l = -\log y_c$$

Apart from the cost function used during training, the accuracy of the network at each stage will also be measured (so the ratio of correct predictions/total predictions).

3 Methodology

The dataset that is used for training and evaluation of this network is the classical MNIST dataset of normalized images of hand-drawn digits (0-9) with a size of 28x28 pixels (total=784).

The network was implemented using two linear layers, so a hidden layer with 784 inputs and 300 outputs and an output layer (with 300 inputs) and the weights connecting layers were intialized with a normal random distribution around a mean of 1. On the first layer, a sigmoid activation is included to compress all the values in the range of 0 and 1. A softmax activation function is used on top of the output layer, such that the networks outputs can be described as a probability distribution. Thus, cross entropy is used as a typical measure for measuring the distance between two probability distributions.

The network is trained by means of backpropagation of the error through the network, whereby the weights of the network are adjusted by subtracting the gradient of the error (so the loss) with respect to the weights, multiplied by a learning factor, from the original weights:

$$^{*}W_x = W_x - \mathrm{a}\left(\frac{\partial Error}{\partial W_x}\right)$$

The gradient of the loss w.r.t. the weights can be obtained by first doing a forward pass of the network with training data for which the true class is known. Thus, for each instance of training, the error is obtained. By applying the multivariate chain rule for derivatives, the weights' gradients are calculated by obtaining the gradient of the loss w.r.t to each step of the network. In the network implemented here, error backpropagation with the multivariate chain rule can be described by:

$$\frac{\partial loss}{\partial layer2Weights} = \frac{\partial loss}{\partial Softmax} \cdot \frac{\partial Softmax}{\partial layer2output} \cdot \frac{\partial layer2output}{\partial layer2Weights}$$

Importantly, instead of training the network at each epoch with the entire dataset, small batches are used to run the code within a short timeframe. This is done by calculating the loss and the loss gradients w.r.t the weights for each instance of a batch. Subsequently, the gradients are averaged and directly used to adjust the weights.

4 Experiments

The experiment is conducted by first doing some exploratory analysis of the network and investigating which learning rate is most suited for this simple network to successfully lower its loss in only 5 epochs. For exploration, learning rates of 0.01, 0.03, 0.1, 0.3 alongside a batch size of 50 images were used, whereby each run for every learning rate was performed 6 times.

For the actual experiment, a learning rate of 0.3 was chosen as this produced the largest decrease in loss during both training and validation. The experimental procedure was also done with a batch size of 50 and the number of epochs was 1000.

Training and testing of the network with a test-train split dataset, whereby the network is exclusively trained by one larger training dataset, and its performance is measured with another (comparable) dataset. The goal of this experiment is thus to see if the training results in a generalizable model that can correctly predict the true class of the input images. Alongside the loss of the network with the validation dataset, also the accuracy of the corresponding predictions will be measured in order to verify the network's generalizability.

5 Results and discussion

The exploratory run (see figures 13-16) was successful in the sense that it gave a clear indication of the differences between learning rates. The decrease in loss is most clearly visible with the validation set in all cases, however, it is only with the largest learning rate (=0.3), where a gradual decline of the average loss in the training set

is also observed. Thus, as described in the experimental section, a learning rate of 0.3 was used for the final experimental run.

Figure 13: Training Loss after 5 epochs for the network with a learning rate of 0.01.

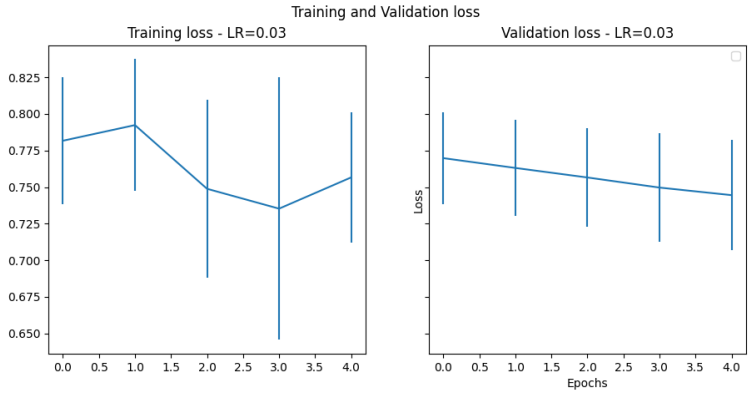

Figure 14: Training Loss after 5 epochs for the network with a learning rate of 0.03.

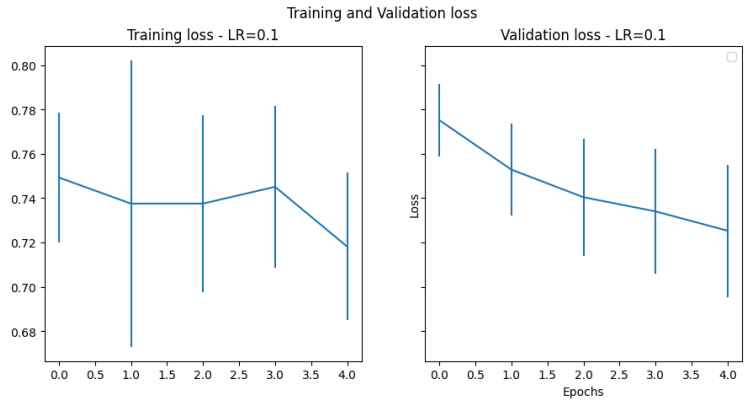

Figure 15: Training Loss after 5 epochs for the network with a learning rate of 0.1.

In the experimental run, it is observed that the network was indeed quite successful in reducing its loss for both training and validation datasets, as can be seen in figure 17. What is remarkable is that the change in training and validation loss are comparable not only in form, but also in values, whereby the loss plateaus at or near a value of 0.1. The generalizability of the network is made clear through the increase in the network's prediction accuracy. While the loss fluctuates considerably, the average validation accuracy is shown to steadily increase (reaching an average of ca. 0.65), whereby it started at nearly chance level (=0.1) and it doesn't yet clearly reach a maximum point.
The findings overall suggest that the learning rate chosen was adequate for training this network and that a

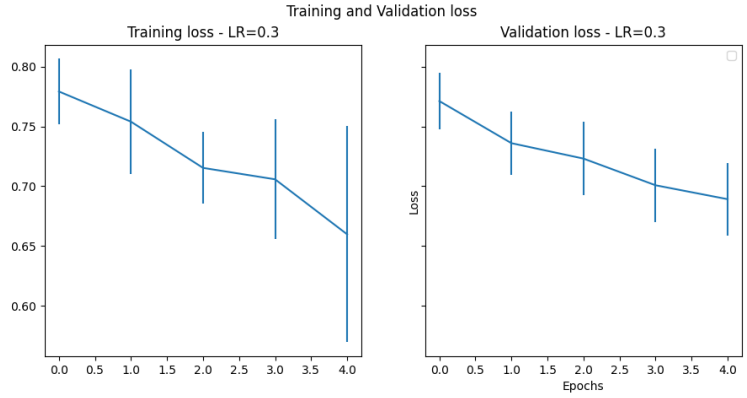

Figure 16: Training Loss after 5 epochs for the network with a learning rate of 0.3.

generalizable network model was created, that despite it's shallowness, is still capable of predicting the correct class of the images at significantly above chance level.

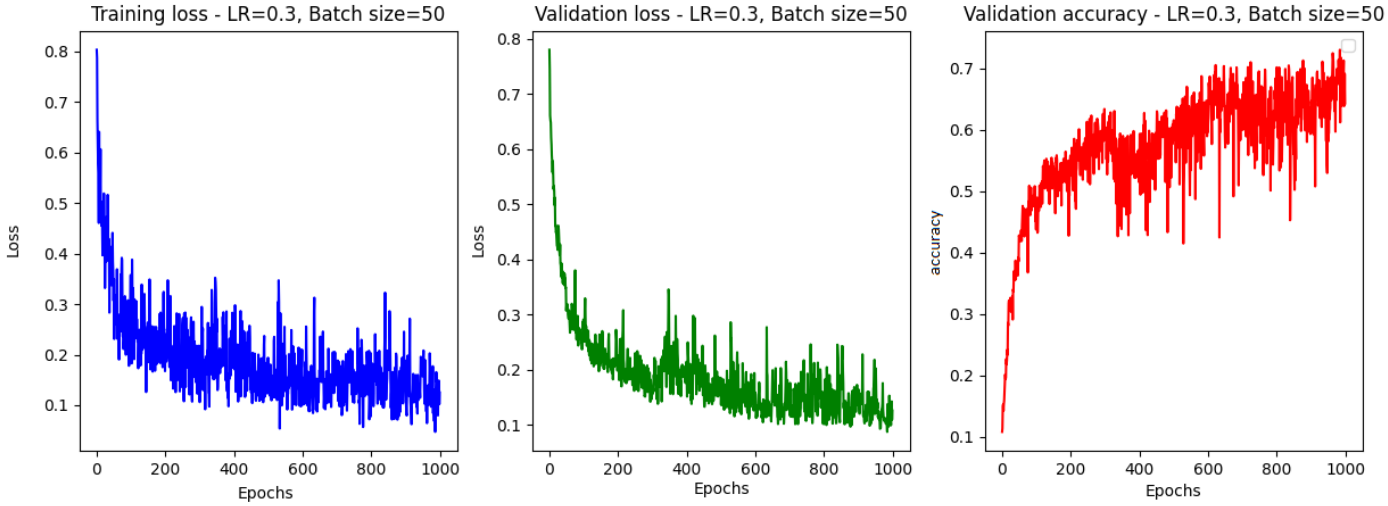

Figure 17: Training Loss, Validation Loss and Validation Accuracy after 1000 epochs for the network with a learning rate of 0.3.

While the results are very promising, there is clearly also room for improvement. As already mentioned, the network is quite shallow and it would be inappropriate to call this "true" deep learning. Creating a more complex network might be computationally more expensive, however, such a network might also have more "representational" capabilities, i.e. it can extract more features from the input data. This can most simply be achieved by adding additional hidden layers. Another improvement might be to increase the density of the network, i.e. by increasing the connectivity of the network (e.g. one can add additional weights by skipping layers).

6 A code snippet

```
# connect layers --> feedforward
layer1_in = np.matmul(input_data[x], layer1_weights) + layer1_bias
layer1_out = stable_sigmoid(layer1_in)

layer2_in = np.matmul(layer1_out, layer2_weights) + layer2_bias
layer2_out = softmax(layer2_in)  # This is the prediction
layer2_out = np.clip(layer2_out, 0.0001, 0.9999)  # this is to make the algorithm more numerically stable

cross_entropy_loss = (1/n_classes) * -np.log(layer2_out[input_targets[x]])

# one-hot-encoding of target, we use this for calculating accuracy, but also for calculating gradient wrt loss and softmax
target_v = one_hot(n_classes, input_targets[x])
if np.argmax(layer2_out) == np.argmax(target_v):
    acc += 1.0

sum_batch_loss += cross_entropy_loss

# GRADIENT CALCULATIONS (backpropagation of error)
# first step is gradient descent of Error wrt both the cross entropy loss and the softmax activation
# this is simplified to the prediction_vector minus the target_vector (i.e. the output layer outputs)
dL_dSM = layer2_out - target_v
dL_db2 = dL_dSM.copy()  # gradient bias

# next is the gradient wrt the weights, i.e. nabla-y dot h-transposed
dL_dW2 = np.outer(dL_dSM, layer1_out)
dL_dh2 = (dL_dSM[None,:] * layer2_weights).sum(axis=1)

# next is gradient of Sigmoid (which is also equal to the gradient wrt bias)
dL_dSG = dL_dh2 * layer1_out * (1 - layer1_out)
dL_db1 = dL_dSG.copy()

# now the gradient wrt first layer weights
dL_dW1 = np.outer(dL_dSG, input_data[x])
```

Figure 18: Code snippet showing the forward pass of the algorithm, resulting in the calculated cross entropy loss, as well as the backpropagation of the calculated error.