# Deep Learning - Assignment 1

**Kiki van Rongen**
10772820

## 1 MLP backprop and NumPy implementation

### 1.1 Analytical derivation of gradients

(a) **Question 1.1a - Individual derivatives**

1. Cross-entropy loss function

$$
\begin{aligned}
L &= -\sum_i t_i \log x_i^{(N)} \\
\frac{\partial L}{\partial x_i^{(N)}} &= -\sum_k t_j \frac{\partial \log x_j^{(N)}}{\partial x_i^{(N)}} \\
&= -\sum_j t_j \frac{\partial \log x_j^{(N)}}{\partial x_j^{(N)}} \frac{\partial x_j^{(N)}}{\partial x_i^{(N)}} \\
&= -\sum_j \frac{t_j}{x_j^{(N)}} \frac{\partial x_j^{(N)}}{\partial x_i^{(N)}} \\
&= -\frac{t_i}{x_i^{(N)}}
\end{aligned}
$$

2. The last layer is a softmax:

$$
x_i^{(N)} = \frac{e^{\tilde{x}_i^{(N)}}}{\sum_{k=1}^{d_N} e^{\tilde{x}_k^{(N)}}} \tag{1}
$$

We distinguish two cases, first $i = j$:

$$\begin{aligned}
\frac{\partial x_{i=j}^{(N)}}{\partial \tilde{x}_j^{(N)}} &= \frac{e^{\tilde{x}_{i=j}^{(N)}} \sum_{k=1}^{d_N} e^{\tilde{x}_k^{(N)}} - e^{\tilde{x}_{i=j}^{(N)}} e^{\tilde{x}_i^{(N)}}}{(\sum_{k=1}^{d_N} e^{\tilde{x}_k^{(N)}})^2} \\
&= \frac{e^{\tilde{x}_{i=j}^{(N)}} (\sum_{k=1}^{d_N} e^{\tilde{x}_k^{(N)}} - e^{\tilde{x}_i^{(N)}})}{(\sum_{k=1}^{d_N} e^{\tilde{x}_k^{(N)}})^2} \\
&= \frac{e^{\tilde{x}_{i=j}^{(N)}}}{\sum_{k=1}^{d_N} e^{\tilde{x}_k^{(N)}}} \frac{\sum_{k=1}^{d_N} e^{\tilde{x}_k^{(N)}} - e^{\tilde{x}_i^{(N)}}}{\sum_{k=1}^{d_N} e^{\tilde{x}_k^{(N)}}} \\
&= x_{i=j}^{(N)} \frac{\sum_{k=1}^{d_N} e^{\tilde{x}_k^{(N)}} - e^{\tilde{x}_i^{(N)}}}{\sum_{k=1}^{d_N} e^{\tilde{x}_k^{(N)}}} \\
&= x_{i=j}^{(N)} (1 - \frac{e^{\tilde{x}_i^{(N)}}}{\sum_{k=1}^{d_N} e^{\tilde{x}_k^{(N)}}}) \\
&= x_{i=j}^{(N)} (1 - x_i^{(N)})
\end{aligned}$$

Next, $i \neq j$:

$$\begin{aligned}
\frac{\partial x_{i\neq j}^{(N)}}{\partial \tilde{x}_j^{(N)}} &= \frac{-e^{\tilde{x}_{i\neq j}^{(N)}} e^{\tilde{x}_j^{(N)}}}{(\sum_{k=1}^{d_N} e^{\tilde{x}_k^{(N)}})^2} \\
&= \frac{-e^{\tilde{x}_{i\neq j}^{(N)}}}{\sum_{k=1}^{d_N} e^{\tilde{x}_k^{(N)}}} \frac{e^{\tilde{x}_j^{(N)}}}{\sum_{k=1}^{d_N} e^{\tilde{x}_k^{(N)}}} \\
&= -x_{i\neq j}^{(N)} x_j^{(N)}
\end{aligned}$$

3. Activation function is a ReLU:

$$x_i^{(l)} = \max(0, \tilde{x}_i^{(l)}) \tag{2}$$

Derivative is split in two parts:

$$\frac{\partial x_i^{(l<N)}}{\partial \tilde{x}_i^{(l<N)}} = \begin{cases} 1, & \text{if } \tilde{x}_i^{(l)} > 0. \\ 0, & \text{else.} \end{cases} \tag{3}$$

4.

$$\begin{aligned}
\tilde{x}^{(l)} &= W^{(l)} x^{(l-1)} + b^{(l)} \\
\frac{\partial \tilde{x}_i^{(l)}}{\partial x_j^{(l-1)}} &= \frac{\partial}{\partial x_j^{(l-1)}} \sum_k \left(W_{ik}^{(l)} x_k^{(l-1)}\right) + b_i^{(l)} \\
&= W_{ij}
\end{aligned}$$

5. Two cases, first $i = j$:

$$\begin{aligned}
\frac{\partial \tilde{x}_{i=j}^{(l)}}{\partial W_{jk}^{(l)}} &= \frac{\partial}{\partial W_{jk}^{(l)}} \sum_m \left(W_{i=j,m}^{(l)} x_m^{(l-1)}\right) + b_{i=j}^{(l)} \\
&= x_k^{(l-1)}
\end{aligned}$$

Next, $i \neq j$:

$$\begin{aligned}
\frac{\partial \tilde{x}_{i=j}^{(l)}}{\partial W_{jk}^{(l)}} &= \frac{\partial}{\partial W_{jk}^{(l)}} \sum_m \left(W_{i\neq j,m}^{(l)} x_m^{(l-1)}\right) + b_{i\neq j}^{(l)} \\
&= 0
\end{aligned}$$

Only those elements where $i = j$ contribute, otherwise derivative is 0.

6. Two cases, first $i = j$:

$$\begin{aligned}\frac{\partial \tilde{x}^{(l)}_{i=j}}{\partial b_j} &= \frac{\partial}{\partial b_j} \sum_k \left(W^{(l)}_{i=j,k} x^{(l-1)}_k\right) + b^{(l)}_{i=j} \\ &= 1\end{aligned}$$

Next, $i \neq j$:

$$\begin{aligned}\frac{\partial \tilde{x}^{(l)}_{i\neq j}}{\partial b_j} &= \frac{\partial}{\partial b_j} \sum_k \left(W^{(l)}_{i\neq j,k} x^{(l-1)}_k\right) + b^{(l)}_{i\neq j} \\ &= 0\end{aligned}$$

(b) **Question 1.1b - Chain rule products**

1. We use index notation, so

$$\begin{aligned}\frac{\partial L}{\partial \tilde{x}^{(N)}_i} &= \sum_j \frac{\partial L}{\partial x^{(N)}_j} \frac{\partial x^{(N)}_j}{\partial \tilde{x}^{(N)}_i} \\ &= -\frac{t_i}{x^{(N)}_i} x^{(N)}_{i=j}(1 - x^{(N)}_i) + \sum_{j\neq i} -\frac{t_j}{x^{(N)}_j}(-x^{(N)}_j x^{(N)}_i) \\ &= -t_i + t_i x^{(N)}_i + \sum_{j\neq i} t_j x^{(N)}_i \\ &= x^{(N)}_i (t_i + \sum_{j\neq i} t_j) - t_i \\ &= x^{(N)}_i - t_i\end{aligned}$$

$$\frac{\partial L}{\partial \tilde{x}^{(N)}} = x^{(N)} - t$$

2.

$$\begin{aligned}\frac{\partial L}{\partial \tilde{x}^{(l<N)}_i} &= \sum_j \frac{\partial L}{\partial \tilde{x}^{(l)}_j} \frac{\partial x^{(l)}_j}{\partial \tilde{x}^{(l)}_i} \\ &= \sum_j \frac{\partial L}{\partial \tilde{x}^{(N)}_j} \frac{\partial x^{(l)}_j}{\partial \tilde{x}^{(l)}_i}\end{aligned}$$

3. Now, there are two possibilities for $x^{(l)}_i$, because of the ReLU. First we set $\tilde{x}^{(l)}_i > 0$, so $x^{(l)}_i > 0$. We get the following derivatives:

$$\frac{\partial L}{\partial \tilde{x}^{(l<N)}_i} = x^{(N)}_i - t_i$$

$$\begin{aligned}\frac{\partial L}{\partial x^{(l<N)}_i} &= \sum_j \frac{\partial L}{\partial \tilde{x}^{(l+1)}_j} \frac{\partial \tilde{x}^{(l+1)}_j}{\partial x^{(l)}_i} \\ &= \sum_j (x^{(N)}_j - t_j) W_{ji}\end{aligned}$$

$$\begin{aligned}\frac{\partial L}{\partial W^{(l)}_{jk}} &= \sum_i \frac{\partial L}{\partial \tilde{x}^{(l)}_j} \frac{\partial \tilde{x}^{(l)}_j}{\partial W^{(l)}_{jk}} \\ &= \sum_i (x^{(N)}_i - t_i) x^{(l-1)}_k\end{aligned}$$

$$\begin{aligned}\frac{\partial L}{\partial b_i^{(l)}} &= \sum_i \frac{\partial L}{\partial \tilde{x}_j^{(l)}} \frac{\partial \tilde{x}_j^{(l)}}{\partial b_i^{(l)}} \\ &= \sum_j (x_j^{(N)} - t_j)\end{aligned}$$

The other case where $\tilde{x}_i^{(l)} <= 0$ leads to all zero derivatives.

(c) **Question 1.1c**
We have to extend the equations above to a matrix, where every column contains the calculated derivatives for one sample in the batch. In order to multiply the forward pass, we need to use these matrices instead of the vector representations we had earlier. The loss function is simply an average over all the values.

## 1.2 NumPy implementation

We first plot the accuracy and loss of the MLP network with default parameters in Figure 1. The loss function declines, as expected, and the accuracy is approximately $39.5\%$. This is not very high of course, so we try to adapt the settings to get a more sufficient result. The batch size is not altered. It has been tested whether this variable significantly improves performance. Although increasing the parameter leads to a higher accuracy, the training period of the network takes much longer. Since the difference is also not substantial, we leave the batch size at its original value.

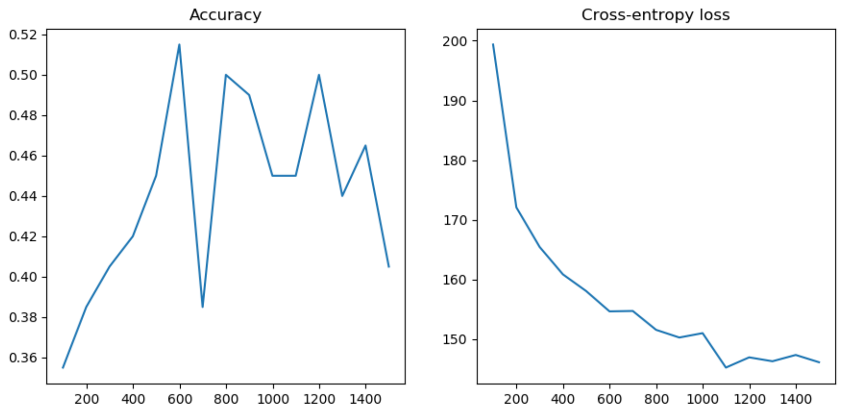

Figure 1: Accuracy and loss with default parameter settings

Next, we modify the learning rate. After running multiple experiments, it can be concluded that the best result is obtained by setting the learning rate equal to $0.001$. The corresponding accuracy is then $48.5\%$, significantly higher than before.

Lastly, we look at the effect of deepening the network. From Figure 2 it can be concluded that more units leads to more accurate results. But, if the network is too deep it becomes too complex for our dataset. We end up with an accuracy of $50\%$.

| **Hidden units** | **Accuracy** |
|---|---|
| 100 | 0.485 |
| 500 | 0.5 |
| 800 | 0.48 |

Figure 2: Accuracy for different number of units in the hidden layer

## 2 PyTorch MLP

The default configuration leads to an accuracy of $43.8\%$. We immediately note that the network structure under this configuration only has one hidden unit with 100 neurons. This is very small, so we expect that adding more layers as well as deepening the units increases performance significantly. However, we do not have the computing power for such large networks, so there is a trade-off. Moreover, large networks could require more steps for training, which also increases runtime. We outline a table for the various settings that are tested. From the table we can derive that our intuition was correct. Adding more hidden units that are

| **Hidden units** | **Learning rate** | **Max steps** | **Batch size** | **Accuracy** |
|---|---|---|---|---|
| 100 | 2e-3 | 1500 | 200 | 0.438 |
| 100 | 1e-3 | 1500 | 200 | 0.479 |
| 512 | 2e-3 | 1500 | 200 | 0.500 |
| 512 | 1e-3 | 1500 | 200 | 0.452 |
| 100,100,100 | 2e-3 | 1500 | 200 | 0.478 |
| 100,100,100 | 1e-3 | 1500 | 200 | 0.475 |
| 512, 512, 512 | 2e-3 | 1500 | 200 | 0.469 |
| 512,512,512 | 1e-3 | 1500 | 200 | 0.489 |
| 512,512,512 | 1e-3 | 2500 | 200 | 0.512 |
| 512,512,1024 | 1e-3 | 2500 | 200 | 0.519 |
| 512,512,1024 | 1e-3 | 2500 | 350 | 0.524 |

Figure 3: Accuracy of the PyTorch MLP with various network structures

deeper increases performance significantly. The 3x512 structure is close to the $52\%$ accuracy we are looking for. From that point onward we try manipulating a single layer, as well as the maximum number of steps and batch size. The last row of the table results in a satisfactory accuracy.

We have also outlined the effect of the learning rate, since this also had an effect on the numpy MLP. It can be concluded that a smaller learning rate performs better for most networks, especially large and deep structured networks. Hence, the last tested networks have a learning rate of 0.001. Figure 4 displays the accuracy and loss function of the network with highest performance (3x512) per time step. Evaluation takes place every 100 iterations.

## 3 Custom Module: Batch Normalization

1. Derivative of the loss function to gamma:

$$\frac{\partial L}{\partial \gamma} = \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial \gamma} = \sum_{i=1}^{s} \frac{\partial f}{\partial y_i}\hat{x}_i$$

2. Derivative of the loss function to beta:

$$\frac{\partial L}{\partial \beta} = \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial \beta} = \sum_{i=1}^{s} \frac{\partial f}{\partial y_i}$$

3. For the derivative of the loss function to x, we first specify the derivative to $\hat{x}$:

$$\frac{\partial L}{\partial \hat{x}} = \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial \hat{x}} = \frac{\partial L}{\partial y_i}\gamma$$

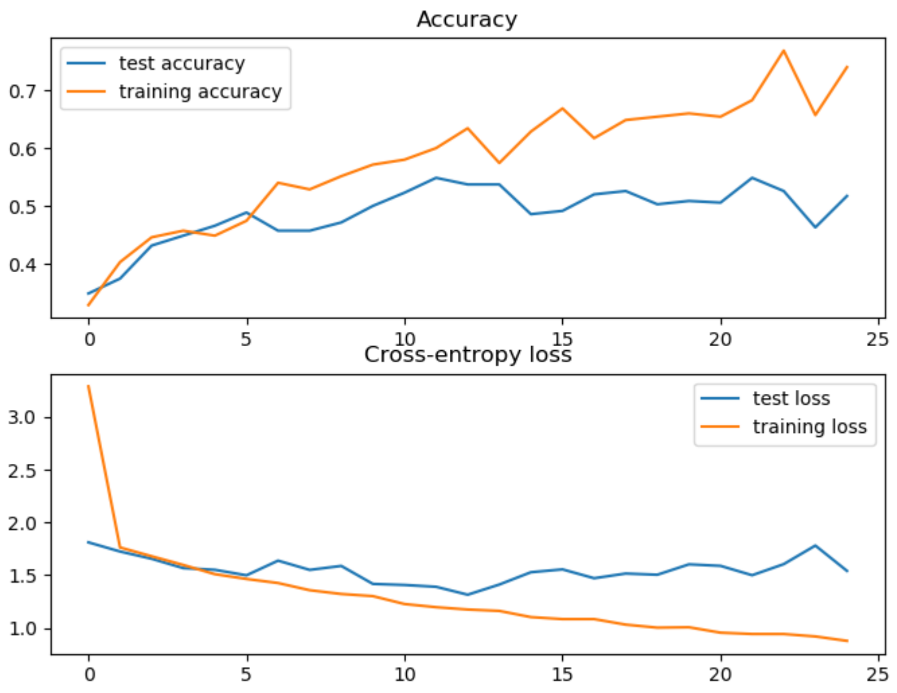

Figure 4: Accuracy and loss function of the 3x512 network

In order to calculate the derivative w.r.t. $x$, we have to take into account $\mu$ and $\sigma$ (they are also depending on $x$).

$$\begin{aligned}
\frac{\partial L}{\partial \mu} &= \frac{\partial L}{\partial \hat{x}}\frac{\partial \hat{x}}{\partial \mu} + \frac{\partial L}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial \mu} \\
\frac{\partial \hat{x}}{\partial \mu} &= \frac{-1}{\sqrt{\sigma^2+\epsilon}} \\
\frac{\partial \sigma^2}{\partial \mu} &= \frac{1}{s}\sum_{i=1}^{s} -2(x_i-\mu) \\
\frac{\partial L}{\partial \sigma^2} &= \frac{\partial L}{\partial \hat{x}}\frac{\partial \hat{x}}{\partial \sigma^2} \\
&= \sum_{i=1}^{s}(x_i-\mu)\cdot(-0.5)\cdot(\sigma^2+\epsilon)^{-1.5} \\
\frac{\partial L}{\partial \mu} &= \sum_{i=1}^{s}\frac{\partial L}{\partial \hat{x}}\cdot\frac{-1}{\sqrt{\sigma^2+\epsilon}}
\end{aligned}$$

We have almost everything to compute the derivative w.r.t. $x$:

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial \hat{x}}\frac{\partial \hat{x}}{\partial x_i} + \frac{\partial L}{\partial \mu}\frac{\partial \mu}{\partial x_i} + \frac{\partial L}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial x_i}$$

The remaining derivatives are rather easy:

$$\frac{\partial \hat{x}}{\partial x_i} = \frac{1}{\sqrt{\sigma^2 + \epsilon}}$$
$$\frac{\partial \mu}{\partial x_i} = \frac{1}{s}$$
$$\frac{\partial \sigma^2}{\partial x_i} = \frac{2(x_i - \mu)}{s}$$

We plug in all the partial derivatives and simplify, in order to get the following expression:

$$\begin{aligned}\frac{\partial L}{\partial x_i} &= (\frac{\partial L}{\partial \hat{x}} \cdot \frac{1}{\sqrt{\sigma^2 + \epsilon}}) + (\frac{\partial L}{\partial \mu} \cdot \frac{1}{s}) + (\frac{\partial L}{\partial \sigma^2} \cdot \frac{2(x_i - \mu)}{s}) \\ &= \frac{(\sigma^2 + \epsilon)^{-0.5}}{s}(s\frac{\partial L}{\partial \hat{x}} - \sum_{j=1}^{s} \frac{\partial L}{\partial \hat{x}_j} - \hat{x}_i \sum_{j=1}^{s} \frac{\partial L}{\partial \hat{x}_j}\hat{x}_j)\end{aligned}$$

## 4 CNN

We have implemented a convolutional neural network, which led to an accuracy of $93.75\%$. Figure 5 plots the loss and accuracy curves.

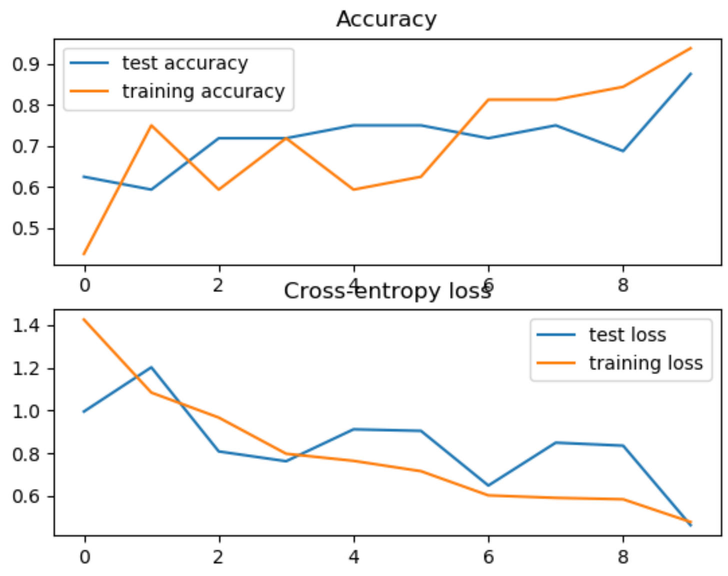

Figure 5: Loss and accuracy curves for CNN