Deep Learning 1
Lab 1

David Wessels
11323272
davidwessels15@gmail.com

1 MLP backprop and NumPy implementation

1.1 Analytical derivations of gradients

a For the derivative of $\frac{\partial L}{\partial x^{(n)}}$, which is a derivative of a scalar w.r.t. a vector, we make use of the fact that this derivative can also be calculated per element of that vector. The partial derivative of one of the elements of $\frac{\partial L}{\partial x^{(n)}}$ can then be calculated as:

$$\begin{aligned}
(\frac{\partial L}{\partial x^{(n)}})_i &= \frac{\partial L}{\partial x_i^{(n)}} \\
&= \frac{\partial}{\partial x_i^{(n)}} - \sum_i t_i log(x_i^{(n)}) \\
&= -\frac{t_i}{x_i^{(n)}} \\
\frac{\partial L}{\partial x^{(n)}} &= -\mathbf{t} \circ (\mathbf{x}^{(n)})^{-1}
\end{aligned}$$

The following derivative which is calculated is the one of $\frac{\partial x^{(n)}}{\partial \tilde{x}^{(n)}}$, which is a derivative of vector w.r.t. a vector. Such a derivative results in a matrix and the following formula can be used to calculate the elements of the resulting matrix: $(\frac{\partial x^{(n)}}{\partial \tilde{x}^{(n)}})_{ij} = (\frac{\partial x_i^{(n)}}{\partial \tilde{x}_j^{(n)}})$. However when using this formula we need this derivative can be looked at for two different scenario's, namely where $i = j$ and $i \neq j$. First is the scenario stated for $i = j$:

$$\begin{aligned}
(\frac{\partial x^{(n)}}{\partial \tilde{x}^{(n)}})_{ij} &= (\frac{\partial x_i^{(n)}}{\partial \tilde{x}_j^{(n)}}) \\
&= \frac{\partial}{\partial \tilde{x}_j^{(n)}} \frac{exp(\tilde{x}_i^{(n)})}{\sum_{k=1}^{d_N} exp(\tilde{x}^{(n)})_k} \\
&= \frac{exp(\tilde{x}_i^{(n)}) \sum_k exp(\tilde{x}_k^{(n)}) - exp(\tilde{x}_i^{(n)}) exp(\tilde{x}_j^{(n)})}{(\sum_k exp(\tilde{x}_k^{(n)}))^2} \\
&= \frac{exp(\tilde{x}_i^{(n)}) \sum_k exp(\tilde{x}_k^{(n)})}{(\sum_k exp(\tilde{x}_k^{(n)}))^2} - \frac{exp(\tilde{x}_i^{(n)}) exp(\tilde{x}_j^{(n)})}{(\sum_k exp(\tilde{x}_k^{(n)}))^2} \\
&= \frac{exp(\tilde{x}_i^{(n)})}{\sum_k exp(\tilde{x}_k^{(n)})} - \frac{exp(\tilde{x}_i^{(n)})^2}{(\sum_k exp(\tilde{x}_k^{(n)}))^2} \\
&= \frac{exp(\tilde{x}_i^{(n)})}{\sum_k exp(\tilde{x}_k^{(n)})} - \Big(\frac{exp(\tilde{x}_i^{(n)})}{\sum_k exp(\tilde{x}_k^{(n)})}\Big)^2 \\
&= x_i^{(n)} - (x_i^{(n)})^2
\end{aligned}$$

Secondly the scenario for $i \neq j$ is stated:

$$\begin{aligned}
(\frac{\partial x^{(n)}}{\partial \tilde{x}^{(n)}})_{ij} &= (\frac{\partial x_i^{(n)}}{\partial \tilde{x}_j^{(n)}}) \\
&= \frac{\partial}{\partial \tilde{x}_j^{(n)}} \frac{exp(\tilde{x}_i^{(n)})}{\sum_{k=1}^{d_N} exp(\tilde{x}_k^{(n)})} \\
&= \frac{0 * \sum_k exp(\tilde{x}_k^{(n)}) - exp(\tilde{x}_i^{(n)}) exp(\tilde{x}_j^{(n)})}{(\sum_k exp(\tilde{x}_k^{(n)}))^2} \\
&= -\frac{exp(\tilde{x}_i^{(n)}) exp(\tilde{x}_j^{(n)})}{(\sum_k exp(\tilde{x}_k^{(n)}))^2} \\
&= -\frac{exp(\tilde{x}_i^{(n)})}{(\sum_k exp(\tilde{x}_k^{(n)}))^2} * \frac{exp(\tilde{x}_j^{(n)})}{(\sum_k exp(\tilde{x}_k^{(n)}))^2} \\
&= -x_i^{(n)} * x_j^{(n)}
\end{aligned}$$

So the total derivative of both scenario's can be combined as the following formula:

$$(\frac{\partial x^{(n)}}{\partial \tilde{x}^{(n)}})_{ij} = \begin{cases} x_i^{(n)} - (x_j^{(n)})^2 & \text{if i = j} \\ -x_i^{(n)} * x_j^{(n)} & \text{if i} \neq \text{j} \end{cases}$$

However, for the implementation of a MLP it is not handy to implement this derivative as a decision rule. Both options can be combined using a indicator function, as been stated below:

$$\begin{aligned}
(\frac{\partial x^{(n)}}{\partial \tilde{x}^{(n)}})_{ij} &= x_i^{(n)}(\mathbf{1}_{i=j} - x_j^{(n)}) \\
\frac{\partial x^{(n)}}{\partial \tilde{x}^{(n)}} &= diag(x^{(n)}) - x^{(n)}(x^{(n)})^T
\end{aligned}$$

Next up, we are going to calculate the derivative of $\frac{\partial x^{(l<N)}}{\partial \tilde{x}^{(l<N)}}$. For this derivative we can again use the same rule as the former one. This for the reason we take again a partial derivative of a vector w.r.t. a vector. An element of the resulting matrix of this derivative can be calculated

as following:

$$\begin{aligned}
(\frac{\partial x^{(l<N)}}{\partial \tilde{x}^{(l<N)}})_{ij} &= \frac{\partial x_i^{(l<N)}}{\partial \tilde{x}_j^{(l<N)}} \\
&= \frac{\partial}{\partial \tilde{x}_j^{(l<N)}} max(0, (x)_i^{(l)}) + a * min(0, (x)_i^{(l)}) \\
&= \begin{cases} 1, & \text{if i = j and } \tilde{x}_j > 0 \\ a, & \text{if i = j and } \tilde{x}_j \leq 0 \\ 0, & \text{otherwise} \end{cases}
\end{aligned}$$

The derivative of $\frac{\partial \tilde{x}^{(l)}}{\partial x^{(l-1)}}$ is again a derivative from a vector with respect to another vector, which results in a matrix. The elements of this matrix can again be calculated by the following formula:

$$\begin{aligned}
\frac{\partial \tilde{x}^{(l)}}{\partial x^{(l-1)}} &= \frac{\partial}{\partial x^{(l-1)}} W^{(l)} x^{(l-1)} + b^{(l)} \\
&= W^{(l)}
\end{aligned}$$

$$\begin{aligned}
(\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}})_{ijk} &= \frac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^{(l)}} \\
&= \frac{\partial}{\partial W_{jk}^{(l)}} W_i^{(l)} x^{(l-1)} + b_i^{(l)} \\
&= \mathbf{1}_{i=j} x_k^{(l-1)}
\end{aligned}$$

$$\begin{aligned}
(\frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}})_{ij} &= \frac{\partial \tilde{x}_i^{(l)}}{\partial b_j^{(l)}} \\
&= \frac{\partial}{\partial b_j^{(l)}} W_i^{(l)} x^{(l-1)} + b_i^{(l)} \\
&= \mathbf{1} \\
\frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} &= \frac{\partial}{\partial b^{(l)}} W^{(l)} x^{(l-1)} + b^{(l)} \\
&= \mathbf{I}
\end{aligned}$$

b

$$\begin{aligned}
\frac{\partial L}{\partial \tilde{x}_j} &= \sum_i \frac{\partial L}{\partial x_i^{(n)}} \frac{\partial x_i^{(n)}}{\partial \tilde{x}_j^{(n)}} \\
&= \sum_i \frac{\partial L}{\partial x_i^{(n)}} * x_i^{(n)} (\mathbf{1}_{i=j} - x_j^{(n)}) \\
\frac{\partial L}{\partial \tilde{x}} &= \frac{\partial L}{\partial x^{(n)}} (diag(x^{(n)}) - x^{(n)} (x^{(n)})^T)
\end{aligned}$$

$$(\frac{\partial L}{\partial \tilde{x}^{(l<N)}})_j = \sum_i \frac{\partial L}{\partial x_i^{(l)}} \frac{\partial x_i^{(l)}}{\partial \tilde{x}_j^{(l)}}$$
$$= \sum_i \begin{cases} \frac{\partial L}{\partial x_i^{(l)}} * 1, & \text{if i = j and } \tilde{x}_j > 0 \\ \frac{\partial L}{\partial x_i^{(l)}} * a, & \text{if i = j and } \tilde{x}_j \leq 0 \\ 0, & \text{otherwise} \end{cases}$$

For implementation purposes is this decision rule implemented in a diagonal matrix in which the diagonal elements, so if i == j, are a if $\tilde{x}_j > 0$ and is 1 if $\tilde{x}_j \leq 0$. This matrix can than be muliplicated with the gradients of the former module, with the formula stated below:

$$(\frac{\partial L}{\partial \tilde{x}^{(l<N)}})_j = \begin{bmatrix} \frac{\partial L}{\partial x_1^{(l)}} \\ \frac{\partial L}{\partial x_2^{(l)}} \\ \vdots \\ \frac{\partial L}{\partial x_n^{(l)}} \end{bmatrix} \begin{bmatrix} d_1 & 0 & \dots & 0 \\ 0 & d_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & d_n \end{bmatrix} = \begin{bmatrix} \frac{\partial L}{\partial x_1^{(l)}} * d_1 \\ \frac{\partial L}{\partial x_2^{(l)}} * d_2 \\ \vdots \\ \frac{\partial L}{\partial x_l^{(l)}} * d_n \end{bmatrix}$$

$$(\frac{\partial L}{\partial x^{(l<N)}})_i = \sum_j (\frac{\partial L}{\partial \tilde{x}^{(l+1)}})_j (\frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}})_{ji}$$

$$(\frac{\partial L}{\partial W^{(l)}})_{ik} = \sum_j \frac{\partial L}{\partial \tilde{x}_j^{(l)}} \frac{\partial \tilde{x}_j^{(l)}}{\partial W_{ik}^{(l)}}$$
$$= \sum_j \frac{\partial L}{\partial \tilde{x}_j^{(l)}} \mathbf{1}_{i=j} x_k^{(l-1)}$$

$$(\frac{\partial L}{\partial b^{(l)}})_i = \sum_j \frac{\partial L}{\partial \tilde{x}_j^{(l)}} \frac{\partial \tilde{x}_j^{(l)}}{\partial b_i^{(l)}}$$
$$= \sum_j \frac{\partial L}{\partial \tilde{x}_j^{(l)}} * \mathbf{I}$$

c The first thing which has to be changed when $B \neq 1$, is the way the cross entropy is calculated. When the $B > 1$ the output of the crossentropy still needs to be a scalar. However, because of the extra dimension, corresponding to the batch size, we get also multiple losses (for every entry in a batch). To get a single loss, we summ all losses.

Another reason ....

1.2 Numpy implementation

Below are the results stated for my numpy implementation of a multilayer peceptron. These results are obtained with the given default values of the parameters. In 1 there are two plots presented, the left plot shows the accuracies for the training and test set, the right plot shows the losses for the training and test set. These results are obtained after 1500 steps of training, with batches of size 200, with one hidden layer of 100 neurons.

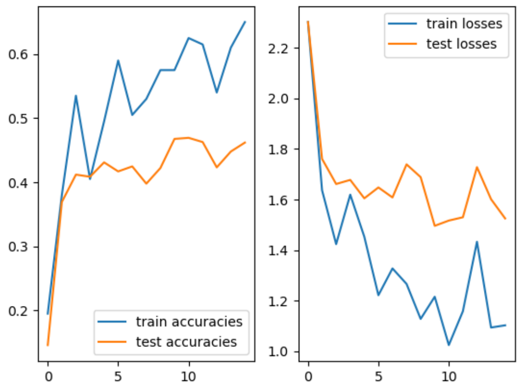

Figure 1: Results of my numpy version of a MLP with the default settings.

2 PyTorch MLP

In this section the multilayer perceptron is again discussed, however this time implemented in the PyTorch library. For this exercise I experimented with the different parameters and the architecture to improve the performance of the model, which will be discussed in this section.

Firstly, I tried to improve the performance is to change the optimizer from standard SGD to the ADAM optimizer. I chose for the ADAM optimizer, while it gave on almost every test higher results then the SGD optimizer. On the final architecture this difference in performance was the most significant. With the SGD optimiser I got an accuracy of 29.3 % and with the adam optimizer an accuracy of 58.1%, with the same learning rate for both optimizers. However it has to be said that the SGD optimizer did performed better with another network architecture of two hidden layers with 512 neurons as been said in the former paragraph.

Secondly, I tried to improve the performance by tweaking and changing the network architecture of my implemented multilayer perceptron. The first layer structure which I tried was adding an extra hidden layer to the default one, and set the number of neurons for both layers to 512. This gave already an increase in performance from 50.5% to 53.4 %. Besides the increase of performance, it also increased in over fitting on the training set, while in the created plots could be seen that after 1500 steps the loss on the test set increased again. However the accuracy flattend on the test set.

Another layer structure I tried is a funnel like structure. Such a structure could intuitively explained, while it is been said that deeper layer contain higher order features. So, with a funnel like architecture it could be argued that by lowering the number of neurons in the next layers, multiple lower layer features can be combined to a more specific feature in the next layer. After running the experiments I found out that it initially did not work. However after adding batch normalization into the architecture it the performance did improve. Batch normalization helps with overfitting, by normalizing the output of former layers to a zero mean. By adding these batch normalization layers, after every linear layer and before the activation function, the accuracy improved from 53.4 % to 54.9%, with the funnel like structure. I tried multiple funnel like architectures but the one from which the results are presented, and which performed the best, is with a hidden layer structure of ('800', '400', '200', '100,). I also removed the softmax activation function after the last layer, which made also an improvement.

The last thing I added to the architecture are Dropout layers. At first I added a dropout layer with a probability of 0.2 after every hidden layer. This ended up in a increase of the accuracy to 57.6%. However, I thought about the applications of these layers and argued that a dropout layer in the last layer could be nog very sufficient. This because the last layer contains the most expressive features, and dropping some of these neurons could lead to losing important information. So I tried another experiment I only removed the dropout layer after the last linear layer. To prevent losing important features in this last linear layer. This resulted agian in an increase of accuracy, up to 58.1 %.

Besides all mensioned optimizations, I also expirimented with the learning rate and the batch size. The in- or decrease in performance changed with the optimizer which is used, however also with the changes in the network architecture. The optimal learning rate and batch size I found is a learning rate of $1e^{-4}$ and a batch size of 512. I also initialised the weights of all linear layers with a zero mean and a standard deviation of 0.0001, the same as given for the numpy implementation.

Below is the final architecture formly stated, and in figure 2 are the results stated:

$$
\begin{aligned}
&(0) : Linear(in_features = 3072, out_features = 800, bias = True)\\
&(1) : BatchNorm1d(800, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)\\
&(2) : LeakyReLU(negative_slope = 0.02)\\
&(3) : Dropout(p = 0.2, inplace = False)\\
&(4) : Linear(in_features = 800, out_features = 400, bias = True)\\
&(5) : BatchNorm1d(400, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)\\
&(6) : LeakyReLU(negative_slope = 0.02)\\
&(7) : Dropout(p = 0.2, inplace = False)\\
&(8) : Linear(in_features = 400, out_features = 200, bias = True)\\
&(9) : BatchNorm1d(200, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)\\
&(10) : LeakyReLU(negative_slope = 0.02)\\
&(11) : Dropout(p = 0.2, inplace = False)\\
&(12) : Linear(in_features = 200, out_features = 100, bias = True)\\
&(13) : BatchNorm1d(100, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)\\
&(14) : LeakyReLU(negative_slope = 0.02)\\
&(15) : Linear(in_features = 100, out_features = 10, bias = True)
\end{aligned}
$$

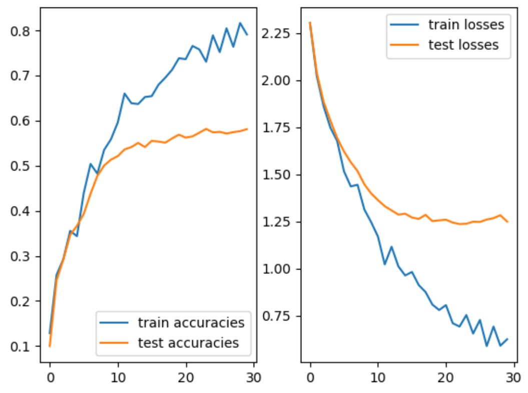

Figure 2: Results of my pytorch version of a MLP with the default settings.

3 Costum Module: Batch normalization

3.1 Automatic differentiation

3.2 Manual implementation backward pass

$$\begin{aligned}
\frac{\partial L}{\partial \gamma} &= \frac{\partial L}{\partial y}\frac{\partial y}{\partial \gamma} \\
(\frac{\partial L}{\partial \gamma})_j &= \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \gamma_j} \\
&= \sum_s \frac{\partial L}{\partial y_j^s} * \hat{x}_j^s
\end{aligned}$$

$$\begin{aligned}
\frac{\partial L}{\partial \beta} &= \frac{\partial L}{\partial y}\frac{\partial y}{\partial \beta} \\
(\frac{\partial L}{\partial \beta})_j &= \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \beta_j} \\
&= \sum_s \frac{\partial L}{\partial y_j^s} * 1
\end{aligned}$$

4 PyTorch CNN

In this section, I discuss my implementation of the simplified VGG network. The implemented architecture was trained on the CIFAR-10 dataset, and resulted in a much higher accuracy then the previous trained implementations. This result is as expected, while it is said that convolutional layers generalize better for images than fully connected layers. In fig 3 are the results stated of the simplified version of this VGG network. It can be seen that the accuracy on the test set, after 5000 steps of training, is not flattened. This means that with longer training it might increase even more.

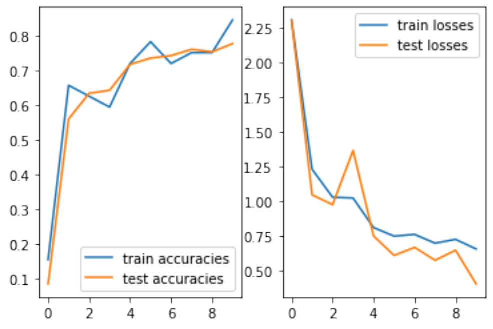

Figure 3: Results the simplified VGG netwrok with the default settings.