# Deep Learning - Assignment 1

**Ragger Jonkers**
10542604
MSc Artificial Intelligence
Universiteit van Amsterdam
Science Park 904
`ragger.jonkers@student.uva.nl`

# 1 MLP backprop and NumPy implementation

## 1.1 Analytical derivation of gradient

### 1.1.1 Gradient modules

Let us denote $argmax(t) = y$. When calculating the softmax derivative there is two options: one when $y = j$ and one when $y \neq j$.

$$\frac{\partial L}{\partial x^{(N)}} = \frac{\partial[-\log x_y^{(N)}]}{\partial x^{(N)}} = -\frac{1}{x^{(N)}} \quad \text{if} \quad x^{(N)} = y \quad \text{else} \quad 0$$

$$\frac{\partial x_y^{(N)}}{\partial \widetilde{x}_j^{(N)}} = \frac{\frac{\exp \widetilde{x}_y^{(N)}}{\sum_{k=1}^{d_N} \exp \widetilde{x}_k^{(N)}}}{\partial \widetilde{x}_j^{(N)}} = \frac{\exp \widetilde{x}_y^{(N)} \sum_{k=1}^{d_N} \exp \widetilde{x}_k^{(N)} - \exp \widetilde{x}_j^{(N)} \exp \widetilde{x}_y^{(N)}}{(\sum_{k=1}^{d_N} \exp \widetilde{x}_k^{(N)})^2}$$
$$= \frac{\exp \widetilde{x}_y^{(N)}}{\sum_{k=1}^{d_N} \exp \widetilde{x}_k^{(N)}} \frac{\sum_{k=1}^{d_N} \exp \widetilde{x}_k^{(N)} - \exp \widetilde{x}_j^{(N)}}{\sum_{k=1}^{d_N} \exp \widetilde{x}_k^{(N)}}$$
$$= \widetilde{x}_y^{(N)}(1 - \widetilde{x}_j^{(N)}) \quad \text{for } y = j$$

$$\frac{\partial x_y^{(N)}}{\partial \widetilde{x}_j^{(N)}} = \frac{\frac{\exp \widetilde{x}_y^{(N)}}{\sum_{k=1}^{d_N} \exp \widetilde{x}_k^{(N)}}}{\partial \widetilde{x}_j^{(N)}} = \frac{-\exp \widetilde{x}_j^{(N)} \exp \widetilde{x}_y^{(N)}}{(\sum_{k=1}^{d_N} \exp \widetilde{x}_k^{(N)})^2}$$
$$= \frac{-\exp \widetilde{x}_j^{(N)}}{\sum_{k=1}^{d_N} \exp \widetilde{x}_k^{(N)}} \frac{\exp \widetilde{x}_y^{(N)}}{\sum_{k=1}^{d_N} \exp \widetilde{x}_k^{(N)}}$$
$$= -\widetilde{x}_j^{(N)} \widetilde{x}_y^{(N)} \quad \text{for } y \neq j$$

$$\frac{\partial x^{(l<N)}}{\partial \widetilde{x}^{(l<N)}} = \frac{\partial[max(0, \widetilde{x}^{(l<N)})]}{\partial \widetilde{x}^{(l<N)}} = 0 \quad \text{if} \quad \widetilde{x}^{(l<N)} \leq 0, \quad 1 \quad \text{if} \quad \widetilde{x}^{(l<N)} > 0$$

$$\frac{\partial \widetilde{x}^{(l)}}{\partial x^{(l-1)}} = \frac{\partial W^{(l)} x^{(l-1)} + b^{(l)}}{\partial x^{(l-1)}} = W^{(l)}$$

$$\frac{\partial \widetilde{x}^{(l)}}{\partial W^{(l)}} = \frac{\partial W^{(l)} x^{(l-1)} + b^{(l)}}{\partial W^{(l)}} = x^{(l-1)}$$

$$\frac{\partial \widetilde{x}^{(l)}}{\partial b^{(l)}} = \frac{\partial W^{(l)} x^{(l-1)} + b^{(l)}}{\partial b^{(l)}} = 1$$

### 1.1.2 Gradients

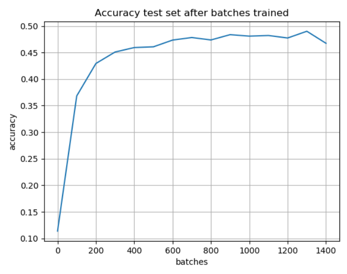
$$
\begin{aligned}
\frac{\partial L}{\partial \widetilde{x}_j^{(N)}} &= \frac{\partial L}{\partial x_y^{(N)}} \frac{\partial x_y^{(N)}}{\partial \widetilde{x}_j^{(N)}} = \frac{\partial L}{\partial x_y^{(N)}} (x_y^{(N)}(1 - x_j^{(N)})) \quad \text{for } y = j \\
\frac{\partial L}{\partial \widetilde{x}_j^{(N)}} &= \frac{\partial L}{\partial x_y^{(N)}} \frac{\partial x_y^{(N)}}{\partial \widetilde{x}^{(N)}} = \frac{\partial L}{\partial x_y^{(N)}} (-x_j^{(N)} x_y^{(N)}) \quad \text{for } y \neq j \\
\frac{\partial L}{\partial \widetilde{x}^{(l<N)}} &= \frac{\partial L}{\partial x^{(l)}} \frac{\partial x^{(l)}}{\partial \widetilde{x}^{(l)}} = \frac{\partial L}{\partial x^{(l)}} \quad \text{where} \quad \widetilde{x}^{(l)} > 0, \quad \text{elsewhere} \quad 0 \\
\frac{\partial L}{\partial x^{(l<N)}} &= \frac{\partial L}{\partial \widetilde{x}^{(l+1)}} \frac{\partial \widetilde{x}^{(l+1)}}{\partial x^{(l)}} = \frac{\partial L}{\partial \widetilde{x}^{(l+1)}} W^{T^{(l)}} \\
\frac{\partial L}{\partial W^{(l)}} &= \frac{\partial L}{\partial \widetilde{x}^{(l)}} \frac{\partial \widetilde{x}^{(l)}}{\partial W^{(l)}} = \frac{\partial L}{\partial \widetilde{x}^{(l)}} x^{(l-1)} \\
\frac{\partial L}{b^{(l)}} &= \frac{\partial L}{\partial \widetilde{x}^{(l)}} \frac{\partial \widetilde{x}^{(l)}}{\partial b^{(l)}} = \frac{\partial L}{\partial \widetilde{x}^{(l)}}
\end{aligned}
$$

### 1.1.3 Batches

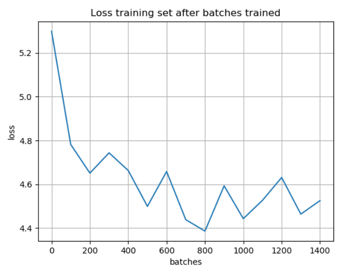
The backpropagation formulas do not change when using batches, it is just that the gradient is averaged over multiple individual points, but formulas of these individual points do not change. However, because we want to avoid iterations, we cannot simply solve the problem by looping over the amount of samples in the batch and then averaging. A vectorized solution gets more complicated for the softmax function, which we defined in two cases per batch sample.

## 1.2 Numpy implementation

In figure 1 you can see accuracy of the test set (size 10000) after each 100 batches of images which were used for optimizing the model parameters. The learning rate was 0.002 and the batch size was 200. Only a hidden layer of 100 neurons was used.



(a) Test accuracy



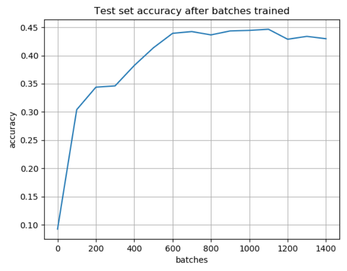
(b) Training loss

Figure 1: Training loss versus test set accuracy - Numpy MLP default settings

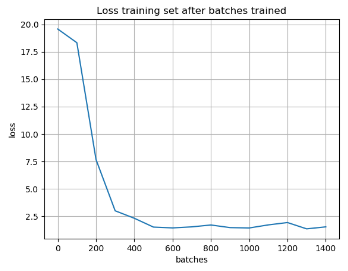
## 2 MLP Pytorch

The implementation of a MLP in Pytorch was run on default parameters and then compared to custom parameters. The default parameters are the same as the default parameters in section 1.2. The default results can be seen in figure 2. The default model only has a hidden layer of 100.

I wanted to add more depth to the network by adding the following sequence of hidden layers: 512-512-256-128-64. Since the input layer is of size 1024, it feels natural to let the model gradually divide amount of neurons for the image in halves. I set the learning rate a magnitude smaller than the default one for a more precise finishing at the best parameters. I

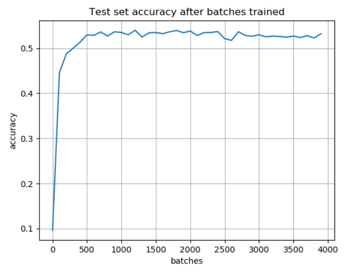


(a) test accuracy

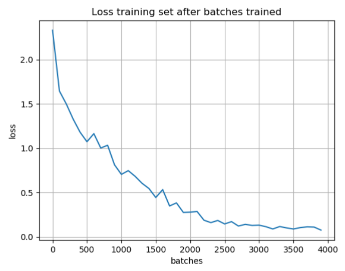


(b) training loss

Figure 2: Training loss versus test set accuracy - Torch MLP default settings



(a) test accuracy



(b) training loss

Figure 3: Training loss versus test set accuracy - Torch MLP custom settings

tried the SGD optimizer, but it didn't come close in terms of convergence to the default Adam optimizer which was my final optimizer. I doubled the batch size, since I also decreased the learning rate. That will decrease noise. The amount of steps (or batches to train on) have been increased to 4000 to make sure to arrive at the optimal parameters at the end of the run. These results can be seen in figure 3

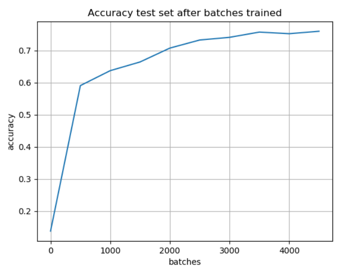
The default settings only get the model to just under 50% accuracy, whereas the custom settings achieved up to 54% accuracy on a test set. This is not too bad of a bad score, considering a baseline model which should score around 10% accuracy predicting randomly, given the 10 different targets (assuming the test set is more or less balanced). For both runs, it can be seen that the model is already near optimal around 500 batches. After that the train loss is decreasing still, but no significant performance is gained judging by the test set accuracy.

## 3 Custom module: Batch normalization

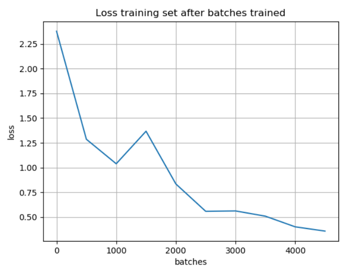
The batch norm forward pass was implemented succesfully, but by prioritization of other tasks in this assignments the manual backward pass of the batch normalization module has been omitted.

## 4 Pytorch CNN

Figure 4 shows the results of an implementation of the VGG network, using default parameters. It performs better than the MLP's we have seen before. For computational reasons the training loss is measured for a single batch at time step `max_steps%FLAGS.eval_freq==0`, hence there is a peak at 1500 steps visible in the training loss. This could possibly be fixed by averaging over the last `FLAGS.eval_freq` batches.



(a) test accuracy



(b) training loss

Figure 4: Training loss versus test set accuracy - Torch CNN default settings