Assignment 1 - Deep Learning

Shannon Doyle
University of Amsterdam
Amsterdam, 1012 WX Amsterdam
Student ID: 11862750
sjdoyle46@gmail.com

Contents

1 MLP backprop and NumPy implementation

1.1 Gradients

First, here is some explanation about the notation used. The dimension of the current layer l is n_l. The activation within one layer is a mapping from a n_l-dim to a n_l-dim space, that is:

$$\begin{aligned} x^{(l)} : \mathbb{R}^{n_l} &\to \mathbb{R}^{n_l} \\ \tilde{x}^{(l)} &\mapsto x^{(l)} \end{aligned}$$

The mapping from layer $(l-1)$ to l is the function:

$$\begin{aligned} \tilde{x}^{(l)} : \mathbb{R}^{n_{l-1}} &\to \mathbb{R}^{n_l} \\ x^{(l-1)} &\mapsto \tilde{x}^{(l)} \end{aligned}$$

This holds for all layers $l = 1, \dots, N$.

Question 1.1a)

I

$$\begin{aligned} \frac{\partial L}{\partial x_i^{(N)}} &= \frac{\partial}{\partial x_i^{(N)}} - \sum_i t_i \log x_i^{(N)} = -\frac{t_i}{x_i^{(N)}} \\ \frac{\partial L}{\partial x^{(N)}} &= \left(-\frac{t_i}{x_i^{(N)}} \right)_{i=1}^{n_N} \in \mathbb{R}^{1 \times n_N} \end{aligned}$$

II

Let us first consider $i = j$. Then

$$\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = \frac{\exp(\tilde{x}_i)\sum_k \exp(\tilde{x}_k) - \exp(\tilde{x}_j)^2}{(\sum_k \exp(\tilde{x}_k))^2} = \frac{\exp(\tilde{x}_i)}{\sum_k \exp(\tilde{x}_k)}\frac{\sum_k \exp(\tilde{x}_k) - \exp(\tilde{x}_j)}{\sum_k \exp(\tilde{x}_k)} = x_i^{(N)}(1 - x_j^{(N)})$$

Now $i \neq j$

$$\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = -\frac{\exp(\tilde{x}_i)\exp(\tilde{x}_j)}{(\sum_k \exp(\tilde{x}_k))^2} = -x_i^{(N)} x_j^{(N)}$$

Then

$$\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} = \Big(x_i^{(N)}(\mathbb{1}\{i=j\} - x_j^{(N)})\Big)_{i,j=1}^{n_N, n_N} = \text{diag}(x_N) - x_N \cdot x_N^T \in \mathbb{R}^{n_N \times n_N}$$

III

Let $l < N$

$$\frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}} = \frac{\partial}{\partial \tilde{x}^{(l)}} \max(0, \tilde{x}^{(l)}) + \alpha \min(0, \tilde{x}^{(l)}) = \begin{bmatrix} \mathbb{1}\{\tilde{x}_1^{(l)} > 0\} + \alpha\mathbb{1}\{\tilde{x}_1^{(l)} \leq 0\} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \mathbb{1}\{\tilde{x}_N^{(l)} > 0\} + \alpha\mathbb{1}\{\tilde{x}_1^{(l)} \leq 0\} \end{bmatrix}$$

$$= \mathbb{1}\{\tilde{x}^{(l)} > 0\} + \alpha\mathbb{1}\{\tilde{x}_1^{(l)} \leq 0\} \circ \boldsymbol{I} \in \mathbb{R}^{n_l \times n_l}$$

where $\boldsymbol{I}$ is the identity matrix, $\mathbb{1}$ the indicator function and $\circ$ is the hadamard product denoting elementwise multiplication between the vector and the matrix.

IV

$$\frac{\partial \tilde{x}_i^{(l)}}{\partial x_j^{(l-1)}} = \frac{\partial}{\partial x_j^{(l-1)}} \sum_j W_{ij}^{(l)} x_j^{(l-1)} + b_i^{(l)} = W_{ij}^{(l)}$$

$$\frac{\partial \tilde{x}^{(l)}}{\partial x^{(l-1)}} = \frac{\partial}{\partial x^{(l-1)}} W^{(l)} x^{(l-1)} + b^{(l)} = W^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$$

V

Here we denote $\tilde{x} \in \mathbb{R}^m$ and $W \in \mathbb{R}^{m \times n}$ for better readability:

$$\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} = \begin{bmatrix} \frac{\partial \tilde{x}_1}{\partial W} \\ \vdots \\ \frac{\partial \tilde{x}_m}{\partial W} \end{bmatrix} \in \mathbb{R}^m$$

$$\frac{\partial \tilde{x}_k^{(l)}}{\partial W^{(l)}} = \begin{bmatrix} \frac{\partial \tilde{x}_k}{\partial W_{11}} & \dots & \frac{\partial \tilde{x}_k}{\partial W_{1n}} \\ \vdots & & \vdots \\ \frac{\partial \tilde{x}_k}{\partial W_{m1}} & \dots & \frac{\partial \tilde{x}_k}{\partial W_{mn}} \end{bmatrix} \in R^{m \times n}$$

$$\frac{\partial \tilde{x}_k^{(l)}}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}} \sum_p W_{kp} x_p^{(l-1)} + b_k^{(l)} = \begin{cases} x_j & \text{if} \quad i = k \\ 0 & \text{else} \end{cases}$$

$$\Rightarrow \frac{\partial \tilde{x}_k^{(l)}}{\partial W^{(l)}} = \begin{bmatrix} & 0 & \\ x_1^{(l-1)} & \dots & x_n^{(l-1)} \\ & 0 & \end{bmatrix} \in R^{m \times n}$$

where the entry is in the k-th row of the matrix. Then we have

$$\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} = \begin{bmatrix} \begin{bmatrix} x_1^{(l-1)} & \dots & x_n^{(l-1)} \\ & \vdots & \\ & 0 & \end{bmatrix} \\ \vdots \\ \begin{bmatrix} & 0 & \\ & \vdots & \\ x_1^{(l-1)} & \dots & x_n^{(l-1)} \end{bmatrix} \end{bmatrix} \in \mathbb{R}^{m \times (m \times n)}$$

VI

$$\frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} = \frac{\partial}{\partial b^{(l)}} W^{(l)} x^{(l-1)} + b^{(l)} = I \in \mathbb{R}^{n_l \times n_l}$$

with I being the identity matrix.

Question 1.1b)

I

$$\begin{aligned} \frac{\partial L}{\partial \tilde{x}^{(N)}} &= \frac{\partial L}{\partial x^{(N)}} \frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} \\ &= \begin{bmatrix} -t_1/x_1 & \dots & -t_{n_N}/x_{n_N} \end{bmatrix} \begin{bmatrix} x_1^{(N)}(1-x_1^{(N)}) & -x_1^{(N)}x_2^{(N)} & \dots & -x_1^{(N)}x_{n_N}^{(N)} \\ -x_2^{(N)}x_1^{(N)} & x_2^{(N)}(1-x_2^{(N)}) & \dots & -x_2^{(N)}x_{n_N}^{(N)} \\ \vdots & & \ddots & \vdots \\ -x_{n_N}^{(N)}x_1^{(N)} & -x_{n_N}^{(N)}x_2^{(N)} & \dots & x_{n_N}^{(N)}(1-x_{n_N}^{(N)}) \end{bmatrix} \in \mathbb{R}^{1 \times n_N} \end{aligned}$$

$$\frac{\partial L}{\partial \tilde{x}^{(N)}} = \left(-x_i^{(N)} \sum_{j=1}^{n_N} \frac{t_j}{x_j} \left(\mathbb{1}\{i=j\} - x_j^{(N)} \right) \right)_{1,i=1}^{1,n_N} \in \mathbb{R}^{1 \times n_N}$$

II

Let $l < N$. Then we can express the gradient of L wrt $\tilde{x}$ in terms of the gradient wrt x by

$$\frac{\partial L}{\partial \tilde{x}^{(l)}} = \frac{\partial L}{\partial x^{(l)}} \frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}} = \frac{\partial L}{\partial x^{(l)}} \mathbb{1}\{\tilde{x}^{(l)} > 0\} + \frac{\partial L}{\partial x^{(l)}} \alpha \mathbb{1}\{\tilde{x}_1^{(l)} \leq 0\} \circ \boldsymbol{I}$$

The dimensions for the matrix multiplication fit because $\frac{\partial L}{\partial x^{(l)}} \in \mathbb{R}^{1 \times n_l}$ and $\mathbb{1}\{\tilde{x}^{(l)} > 0\}$and $\alpha \mathbb{1}\{\tilde{x}_1^{(l)} \leq 0\} \circ \boldsymbol{I} \in \mathbb{R}^{n_l \times n_l}$.

III

Similar to above we can express the gradient of L wrt to x in terms of the gradient wrt $\hat{x}$ by

$$\frac{\partial L}{\partial x^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} \frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} W^{(l+1)}$$

Therefore we have everything to compute the gradient. The dimensions for the multiplication fit again because $\frac{\partial L}{\partial \tilde{x}^{(l+1)}} \in \mathbb{R}^{1 \times n_{l+1}}$ and $W^{(l+1)} \in \mathbb{R}^{n_{l+1} \times n_l}$.

IV

We know that

$$\frac{\partial L}{\partial \tilde{x}^{(l)}} \in \mathbb{R}^{1\times n_l} \quad \text{and} \quad \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} \in \mathbb{R}^{n_l \times n_l \times n_{l-1}}$$

So we know that

$$\frac{\partial L}{\partial W^{(l)}} \in \mathbb{R}^{n_l \times n_{l-1}}$$

Each of the parts is known. The left part is given by 1.1b) II and the right part by 1.1a) V. But due to the special structure of $\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}}$ a simplification can be made that avoids the use of a tensor product. Let us denote

$$\frac{\partial L}{\partial \tilde{x}^{(l)}} = [a_1, \dots, a_{n_l}]$$

Then we have

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} = [a_1, \dots, a_{n_l}] \begin{bmatrix} \begin{bmatrix} x_1^{(l-1)} & \dots & x_{n_{l-1}}^{(l-1)} \\ & \vdots & \\ & 0 & \end{bmatrix} \\ \vdots \\ \begin{bmatrix} & 0 & \\ & \vdots & \\ x_1^{(l-1)} & \dots & x_{n_{l-1}}^{(l-1)} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} a_1 x_1^{(l-1)} & a_1 x_2^{(l-1)} & \dots & a_1 x_{n_{l-1}}^{(l-1)} \\ \vdots & & \ddots & \vdots \\ a_{n_l} x_1^{(l-1)} & & \dots & a_{n_l} x_{nl-1}^{(l-1)} \end{bmatrix}$$

$$= \left(\begin{bmatrix} x_1^{(l-1)} \\ \vdots \\ x_{n_{l-1}}^{(l-1)} \end{bmatrix} [a_1, \dots, a_{n_l}] \right)^T = \left(x^{(l-1)} \frac{\partial L}{\partial \tilde{x}^{(l)}} \right)^T \in \mathbb{R}^{n_l \times n_{l-1}}$$

V

We know from task 1.1a) that $\partial \tilde{x} / \partial b = \boldsymbol{I}$. Therefore

$$\frac{\partial L}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \in \mathbb{R}^{1\times n_l}$$

Question 1.1c)

The dimension of the derivative of a function f of dimension m wrt to some x of dimension $B \times n$ is

$$\frac{\partial f}{\partial x} \in \mathbb{R}^{m \times (B \times n)}$$

Therefore all derivatives wrt to variables that have a batch (which are (x, $\tilde{x}$ or b) get an additional dimension at the appropriate place. That means that we for example have

$$\dim\left(\frac{\partial L}{\partial \tilde{x}^{(l)}}\right) = \dim\left(\frac{\partial L}{\partial x^{(l)}}\right) = \dim\left(\frac{\partial L}{\partial b^{(l)}}\right) = 1 \times B \times n_l$$

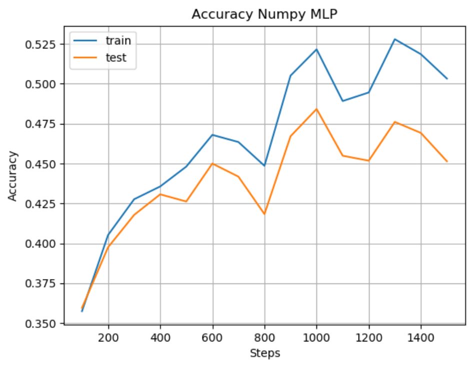
and

$$\dim\left(\frac{\partial \tilde{x}^{(l)}}{\partial \tilde{x}^{(l-1)}}\right) = (B \times n_l) \times (B \times n_{l-1}) \quad \text{and} \quad \dim\left(\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}}\right) = (B \times n_l) \times (n_l \times n_{l-1})$$

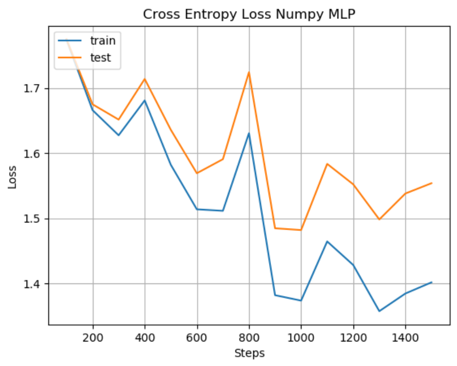
The remaining derivatives change in a similar way. One important point to mention is that we take the average over the first component of the bias gradient when updating the bias in SGD.

	Test set	Train set
Accuracy	54.57%	83.4%
Loss	1.366	0.567

Table 1: Accuracy and cross entropy loss of numpy MLP after the last training step



(a) Accuracy



(b) Loss

Figure 1: Accuracy and cross entropy loss of numpy MLP implementation

1.2 NumPy Implementation

The learning cuvres for accuracy and cross entropy loss of the numpy MLP implementation can be seen in figure 1 and are additionally stored in numpy array .npy files in the folder `results_np`. For optimization the mini-batch stochastic gradient descent algorithm was implemented. After every 100 steps, a batch of size 200 was evaluated on the test set. The accuracy and loss at the evaluation steps are shown which leads to some oscillations.

Finally, the values of the two metrics after the final evaluation step are shown in table 1. The results were obtained with the following standard settings

- 1 hidden layer of 100 neurons
- Learning rate $\eta = 2 \cdot 10^{-3}$
- Number of evaluation steps = 1500
- Batch size = 200
- Evaluation at each 100 steps
- Mini-batch Stochastic Gradient Descent

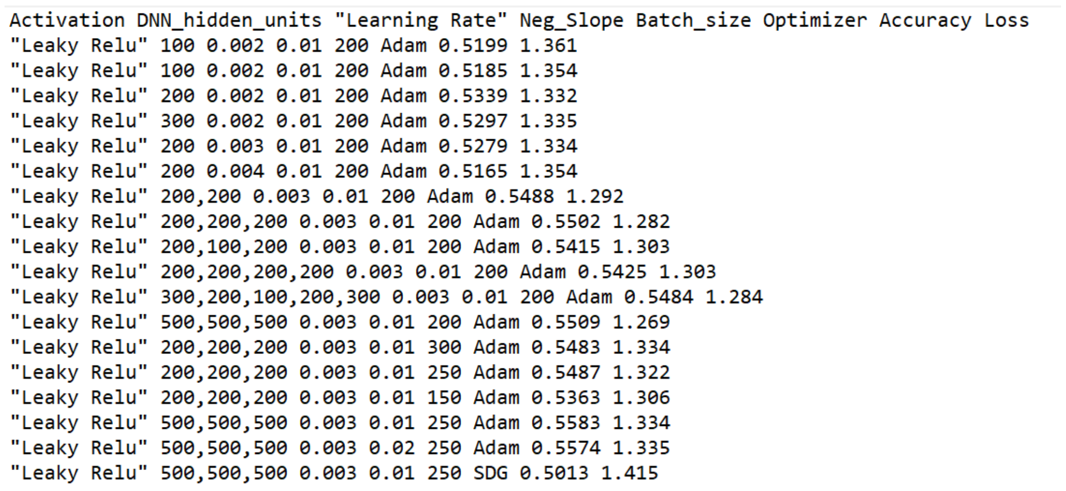
The trend of the learning curve on the test set is still slightly increasing and therefore we can assume that there is no over-fitting.

2 PyTorch MLP

I tried changing the activation function (standard ReLU or LeakyReLU), hidden layer architecture (number of hidden units in each layer and number of layers), batch size and optimizer (Adam, SDG).

I started with the default parameter from the assignment (except that I used a negative slope value for the LeakyReLU of 0.01 instead of 0.02, as this is the default value given in torch).

So the initial parameters were:



```
Activation DNN_hidden_units "Learning Rate" Neg_Slope Batch_size Optimizer Accuracy Loss
"Leaky Relu" 100 0.002 0.01 200 Adam 0.5199 1.361
"Leaky Relu" 100 0.002 0.01 200 Adam 0.5185 1.354
"Leaky Relu" 200 0.002 0.01 200 Adam 0.5339 1.332
"Leaky Relu" 300 0.002 0.01 200 Adam 0.5297 1.335
"Leaky Relu" 200 0.003 0.01 200 Adam 0.5279 1.334
"Leaky Relu" 200 0.004 0.01 200 Adam 0.5165 1.354
"Leaky Relu" 200,200 0.003 0.01 200 Adam 0.5488 1.292
"Leaky Relu" 200,200,200 0.003 0.01 200 Adam 0.5502 1.282
"Leaky Relu" 200,100,200 0.003 0.01 200 Adam 0.5415 1.303
"Leaky Relu" 200,200,200,200 0.003 0.01 200 Adam 0.5425 1.303
"Leaky Relu" 300,200,100,200,300 0.003 0.01 200 Adam 0.5484 1.284
"Leaky Relu" 500,500,500 0.003 0.01 200 Adam 0.5509 1.269
"Leaky Relu" 200,200,200 0.003 0.01 300 Adam 0.5483 1.334
"Leaky Relu" 200,200,200 0.003 0.01 250 Adam 0.5487 1.322
"Leaky Relu" 200,200,200 0.003 0.01 150 Adam 0.5363 1.306
"Leaky Relu" 500,500,500 0.003 0.01 250 Adam 0.5583 1.334
"Leaky Relu" 500,500,500 0.003 0.02 250 Adam 0.5574 1.335
"Leaky Relu" 500,500,500 0.003 0.01 250 SDG 0.5013 1.415
```

Figure 2: Optimization steps with PyTorch MLP implementation with 1500 iterations

- 1 hidden layer of 100 neurons
- Learning rate $\eta = 2 \cdot 10^{-3}$
- Number of evaluation steps = 1500
- Batch size =200
- Evaluation at each 100 steps
- Adam Optimizer with default learning rate

The most relevant steps I took for experimenting with the parameters can be seen in Figure 2 :

At first, I checked whether the LeakyReLU activation function improves the accuracy over the standard ReLU function. This was the case, so I kept this activation function.

Then I changed the number of hidden neurons. I found that results were best with 200 or 500 hidden neurons per layer and three hidden layers worked well. I also experimented with some network structures, like decreasing and increasing number of neurons per hidden layer.

I changed the learning rate and found a learning rate of 0.003 to produce best results.

I experimented with the batch size and found that for 3 hidden layers of 200 neurons a batch size of 200 worked well, and for 3 hidden layers of 500 neurons a batch size of 250 worked better. I continued evaluating the network with 500 neurons per hidden layer, as this is where I encountered the best results.

I changed the negative slope parameter of the LeakyReLU function to 0.02, which decreased performance of the model again.

I finally compared the Adam Optimizer to the stochastic gradient descent optimizer. I found the performance of Adam to be much better.

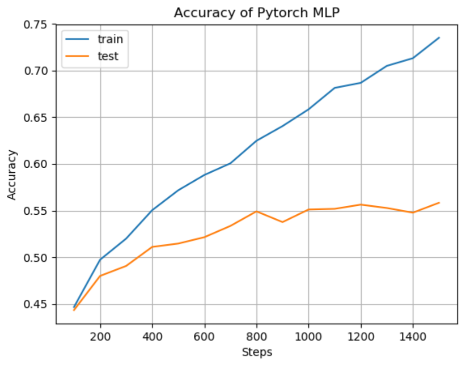
Finally, the parameters I found to produce the best results are the following:

- 3 hidden layers with of 500 neurons each
- Learning rate$\eta = 3 \cdot 10^{-3}$
- Number of evaluation steps = 1500
- Batch size = 250
- Evaluation at each 100 steps
- Adam Optimizer with default learning rate

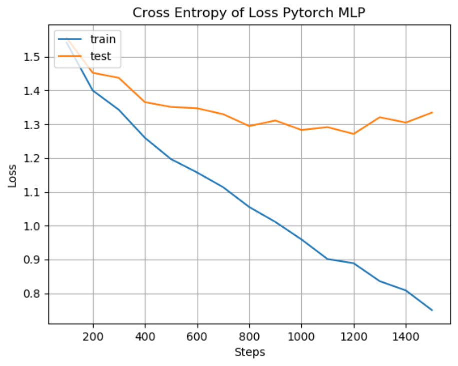
The results of these setting can be seen in Figure 3.

	Test set	Train set
Accuracy	55.83%	73.51%
Loss	1.334	0.751

Table 2: Accuracy and cross entropy loss of PyTorch MLP after the last training step



(a) Accuracy



(b) Loss

Figure 3: Accuracy and cross entropy loss of PyTorch MLP implementation with 1500 iterations

The plots show that after about 800 steps there is no improvement of the accuracy and the loss in the test set anymore. Therefore, training could have been stopped at this point, for example by using early stopping. The parameters could also be further investigated by implementing a grid search. The Pytorch implementation produces better results as it is further optimized than the Numpy version in the previous exercise. The accuracy improved most with increasing the number of hidden layers, as the model can use the information from the previous layers (up to a point).

3 Custom Module: Batch Normalization

This section is about a custom batch normalization implementation. The code for the implementation of the tasks 3.1, 3.2b) and 3.2c) can be found in the `custom_batchnorm.py` file. In the following we present the gradients for the backward pass of the custom batch autograd function in question 3.2b).

3.2 a)

I

At first we want to compute

$$\frac{\partial L}{\partial \gamma} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \gamma}$$

Let us denote the input and output dimension of the current layer by n and the batch size by B. Then it is first helpful to remind the dimensions

$$y \in \mathbb{R}^{B\times n} \quad \text{and} \quad \gamma \in \mathbb{R}^n$$

Then we have for the dimensions of the derivatives

$$\frac{\partial L}{\partial \gamma} \in \mathbb{R}^{1\times n} \quad \text{and} \quad \frac{\partial L}{\partial y} \in \mathbb{R}^{B\times n} \quad \text{and} \quad \frac{\partial y}{\partial \gamma} \in \mathbb{R}^{(B\times n)\times n}$$

In order to arrive at the desired dimension we interpret the derivative in the middle as of $1 \times (B \times n)$ multiply it with the derivative on the right over the $B \times n$. Now we derive the derivatives

$$\frac{\partial L}{\partial y} \quad \text{is known and calculated by next layer in backward pass}$$

$$\frac{\partial y_i^s}{\partial \gamma_j} = \begin{cases} \hat{x}_i^s & \text{if} \quad i = j \\ 0 & \text{else} \end{cases}$$

So the latter derivative has a diagonal structure. To execute the matrix multiplication between the two derivatives we have to sum over the s elements in the batch and the feature components j. But since y_i is independent of γ_j for $i \neq j$, the sum over i vanishes.

$$\frac{\partial L}{\partial \gamma_j} = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial \gamma_j} = \sum_s \frac{\partial L}{\partial y_j^s} \hat{x}_j^s$$

II

For the second derivative we see that

$$\frac{\partial y_i^s}{\partial \beta_j} = \begin{cases} 1 & \text{if} \quad i = j \\ 0 & \text{else} \end{cases}$$

and get

$$\frac{\partial L}{\partial \beta_j} = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial \beta_j} = \sum_s \frac{\partial L}{\partial y_j^s}$$

III

The task is to calculate

$$\frac{dL}{dx} = \frac{\partial L}{\partial y} \frac{dy}{dx}$$

which can be written in components as

$$\frac{dL}{dx_i^r} = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{dy_i^s}{dx_j^r}$$

Note here that we switched the notation from partial derivative to total derivative. Often there are treated as interchangeable but here a differentiation is necessary.

The first we can do is note that y_i^s is independent of x_j^r, which means that

$$\frac{\partial y_i^s}{\partial x_j^r} = 0 \quad \text{for} \quad i \neq j$$

Then the component gradient can be simplified to

$$\frac{dL}{dx^r} = \sum_s \frac{\partial L}{\partial y^s} \frac{dy^s}{dx^r}$$

and we suppress the indexation with i. Now we note that $\hat{x}$ is actually a function not only of x but also of μ and σ^2, which are itself functions of x (and μ in the case of σ).

$$\hat{x} = \hat{x}(x, \mu(x), \sigma^2(x, \mu(x))$$

By now applying the chain rule for total derivatives we get

$$\frac{dy}{dx} = \frac{\partial y}{\partial \hat{x}} \frac{d\hat{x}}{dx}$$

and

$$\frac{d\hat{x}}{dx} = \frac{\partial \hat{x}}{\partial x} + \frac{d\hat{x}}{d\mu} \frac{\partial \mu}{\partial x} + \frac{\partial \hat{x}}{\partial \sigma^2} \frac{d\sigma^2}{dx} \tag{1}$$

$$= \frac{\partial \hat{x}}{\partial x} + \left[\frac{\partial \hat{x}}{\partial \mu} + \frac{\partial \hat{x}}{\partial \sigma^2} \frac{\partial \sigma^2}{\partial \mu} \right] \frac{\partial \mu}{\partial x} + \frac{\partial \hat{x}}{\partial \sigma^2} \left[\frac{\partial \sigma^2}{\partial x} + \frac{\partial \sigma^2}{\partial \mu} \frac{\partial \mu}{\partial x} \right] \tag{2}$$

Now we can calculate all the derivatives that we need. We reintroduce indexing and consider only the pairs with $i = j$ because the derivatives for $i \neq j$ are zero.

$$\frac{\partial y_i^s}{\partial \hat{x}_i^r} = \gamma_i \quad \text{for} \quad s = r \quad \text{and} \quad 0 \quad \text{else}$$

$$\frac{\partial \hat{x}_i^s}{\partial x_i^r} = \frac{1}{\sqrt{\sigma_i^2 + \epsilon}} \quad \text{for} \quad s = r \quad \text{and} \quad 0 \quad \text{else}$$

$$\frac{\partial \hat{x}_i^s}{\partial \mu_i} = -\frac{1}{\sqrt{\sigma_i^2 + \epsilon}}$$

$$\frac{\partial \mu_i}{\partial x_i^s} = \frac{1}{B}$$

$$\frac{\partial \hat{x}_i^s}{\partial \sigma_i^2} = -\frac{1}{2}\frac{x_i^s - \mu_i}{(\sigma_i^2 + \epsilon)^{3/2}}$$

$$\frac{\partial \sigma_i^2}{\partial x_i^r} = \frac{2}{B}(x_i^r - \mu_i)$$

$$\frac{\partial \sigma_i^2}{\partial \mu_i} = -\frac{2}{B}\sum_s^B (x_i^s - \mu_i) = -2\left(\mu - \frac{B}{B}\mu\right) = 0$$

$$\frac{\partial \mu_i}{\partial x_i^s} = \frac{1}{B}$$

Since we have $\partial\sigma^2/\partial\mu = 0$, the formula in equation 2 simplifies to

$$\frac{d\hat{x}}{dx} = \frac{\partial \hat{x}}{\partial x} + \frac{\partial \hat{x}}{\partial \mu}\frac{\partial \mu}{\partial x} + \frac{\partial \hat{x}}{\partial \sigma^2}$$

Then we can plug in the derivatives and work ourselves from the inside piece-wise out

$$\begin{aligned}
\frac{d\hat{x}^s}{dx^r} &= \frac{\partial \hat{x}^s}{\partial x^r} + \frac{\partial \hat{x}^s}{\partial \mu}\frac{\partial \mu}{\partial x^r} + \frac{\partial \hat{x}^s}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial x^r} \\
&= \frac{1}{\sqrt{\sigma^2 + \epsilon}}\mathbb{1}\{s = r\} - \frac{1}{\sqrt{\sigma^2 + \epsilon}}\frac{1}{B} - \frac{1}{2}\frac{x_i^s - \mu_i}{(\sigma_i^2 + \epsilon)^{3/2}}\frac{2}{B}(x_i^r - \mu_i) \\
&= \frac{1}{B\sqrt{\sigma^2 + \epsilon}}\left(B\mathbb{1}\{s = r\} - 1 - \hat{x}^r\hat{x}^s\right)
\end{aligned}$$

Next we take the component formulation of the next outward derivative and note

$$\frac{dy^s}{dx^r} = \sum_p \frac{\partial y^s}{\partial \hat{x}^p}\frac{d\hat{x}^p}{dx^r} = \frac{\partial y^s}{\partial \hat{x}^s}\frac{d\hat{x}^s}{dx^r} = \gamma\frac{d\hat{x}^s}{dx^r}$$

And now the last outward derivative

$$\begin{aligned}
\frac{dL}{dx^r} &= \sum_s \frac{\partial L}{\partial y^s}\gamma\frac{d\hat{x}^s}{d\hat{x}^r} \\
&= \sum_s \frac{\partial L}{\partial y^s}\gamma\left[\frac{1}{B\sqrt{\sigma^2 + \epsilon}}\left(B\mathbb{1}\{s = r\} - 1 - \hat{x}^r\hat{x}^s\right)\right] \\
&= \frac{\gamma}{B\sqrt{\sigma^2 + \epsilon}}\sum_s \frac{\partial L}{\partial y^s}\left(B\mathbb{1}\{s = r\} - 1 - \hat{x}^r\hat{x}^s\right) \\
&= \frac{\gamma}{B\sqrt{\sigma^2 + \epsilon}}\left(B\frac{\partial L}{\partial y^r} - \sum_s \frac{\partial L}{\partial y^s} - \hat{x}^r\sum_s \hat{x}^s\frac{\partial L}{\partial y^s}\right)
\end{aligned}$$

where the indexing is always with i. We can now use the derivatives of L wrt to γ and β and simplify:

$$\frac{dL}{dx^r} = \frac{\gamma}{B\sqrt{\sigma^2+\epsilon}}\left(B\frac{\partial L}{\partial y^r} - \frac{\partial L}{\partial \beta} - \hat{x}^r\frac{\partial L}{\partial \gamma}\right)$$

This is also the way the gradient is going to be implemented.

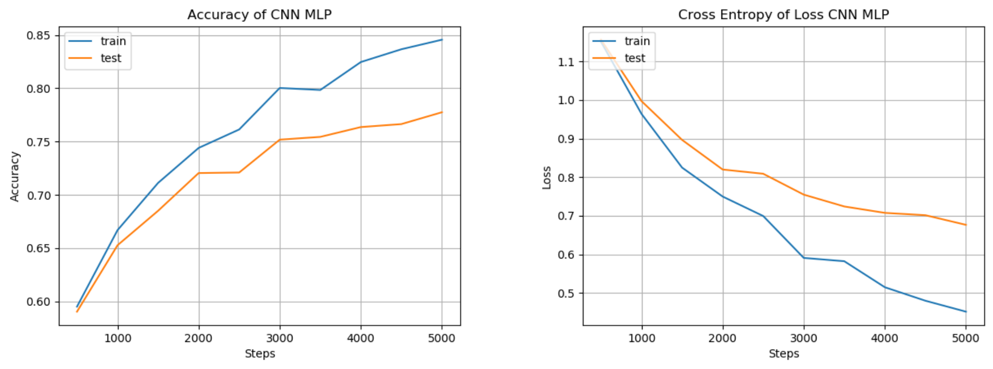
4 PyTorch CNN

Using a convolutional neural network proved to work best for classifying our images, see figures 4 and the table 3. At each evaluation step, the loss and accuracy were calculated for the entire test and train set, but in small batches of size 32. The settings used were:

- Learning rate $\eta = 10^{-4}$
- Number of evaluation steps = 5000
- Batch size = 32
- Evaluation at each 500 steps of the whole training and test set
- Adam Optimizer with standard Momentum and second order estimate parameters of $\beta_1 = 0.9$ and $\beta_2 = 0.999$

	Test set	Train set
Accuracy	77.75%	84.56%
Loss	0.676	0.452

Table 3: Accuracy and cross entropy loss of PyTorch MLP after the last training step



(a) Accuracy (b) Loss

Figure 4: Accuracy and cross entropy loss of PyTorch CNN implementation with 5000 iterations