Deep Learning - Assignment 1

Andrei Pauliuc
11596619
andupauliuc@gmail.com

1 MLP backprop and NumPy implementation

1.1 a)

CrossEntropy Module

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

$$\begin{aligned}\Rightarrow \frac{\partial \mathcal{L}}{\partial x^{(N)}} &= -t \odot \frac{1}{x^{(N)}} \\ &= \begin{bmatrix}0 & \dots & 0 & -\frac{1}{x_i^{(N)}} & 0 & \dots & 0\end{bmatrix} \text{ where } t_i = 1\end{aligned}$$

Softmax Module

$$\begin{aligned}\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} &= \frac{\partial}{\partial \tilde{x}_j^{(N)}}\left(\frac{\exp\left\{\tilde{x}_i^{(N)}\right\}}{\sum_k \exp\left\{\tilde{x}_k^{(N)}\right\}}\right) \\ &= \begin{cases} x_i(1-x_i) & \text{if } i = j \\ -x_i x_j & \text{if } i \neq j \end{cases} \\ &= \delta_{ij} \cdot x_i(1-x_i) + (1-\delta_{ij}) \cdot (-x_i x_j)\end{aligned}$$

$$\begin{aligned}\Rightarrow \frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} &= \begin{bmatrix} x_1(1-x_1) & -x_1 x_2 & \dots & -x_1 x_{d_N} \\ -x_2 x_1 & -x_2(1-x_2) & \dots & -x_2 x_{d_N} \\ \dots & \dots & \dots & \dots \\ -x_{d_N} x_1 & -x_{d_N} x_2 & \dots & x_{d_N}(1-x_{d_N}) \end{bmatrix} \\ &= \text{diag}(x^{(N)}) - x^{(N)} \cdot x^{(N)^T} \text{ (for } x^{(N)} \text{ of shape } d_N \times 1)\end{aligned}$$

ReLU Module

$$\begin{aligned}\frac{\partial x_i^{(l<N)}}{\partial \tilde{x}_j^{(l<N)}} &= \frac{\partial}{\partial \tilde{x}_j^{(l)}}\left(\max\left(0, \tilde{x}_i^{(l)}\right)\right) \\ &= \delta_{ij} \cdot \mathbb{1}(\tilde{x}_j^{(l)} > 0)\end{aligned}$$

$$\Rightarrow \frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}} = \begin{bmatrix} \mathbb{1}(\tilde{x}_1^{(l)} > 0) & 0 & \dots & 0 \\ 0 & \mathbb{1}(\tilde{x}_2^{(l)} > 0) & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \mathbb{1}(\tilde{x}_{d_N}^{(l)} > 0) \end{bmatrix}$$

Linear Module

$$\begin{aligned}\frac{\partial \tilde{x}_i^{(l)}}{\partial b_j^{(l)}} &= \frac{\partial}{\partial b_j^{(l)}}\left(\sum_m W_{im} \cdot x_m^{(l-1)} + b_i^{(l)}\right) \\ &= \frac{\partial}{\partial b_j^{(l)}} b_i^{(l)} = \delta_{ij} \\ \Rightarrow \frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} &= \mathbb{I}_{d_l}\end{aligned}$$

$$\begin{aligned}\frac{\partial \tilde{x}_i^{(l)}}{\partial x_j^{(l-1)}} &= \frac{\partial}{\partial x_j^{(l-1)}}\left(\sum_m W_{im}^{(l)} \cdot x_m^{(l-1)} + b_i^{(l)}\right) \\ &= \delta_{jm} W_{im}^{(l)} = W_{ij}^{(l)} \\ \Rightarrow \frac{\partial \tilde{x}^{(l)}}{\partial x^{(l-1)}} &= W^{(l)}\end{aligned}$$

$$\begin{aligned}\frac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^{(l-1)}} &= \frac{\partial}{\partial W_{jk}^{(l-1)}}\left(\sum_m W_{im} \cdot x_m^{(l-1)}\right) \\ &= \begin{cases} x_k^{(l-1)} & \text{if i = j and k = m} \\ 0 & \text{otherwise} \end{cases} \\ &= \delta_{ij}\delta_{km} x_m^{(l-1)} = \delta_{ij} x_k^{(l-1)} \\ \Rightarrow \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l-1)}} &= P^{(l)}, \text{ a 3D Jacobian tensor,} \\ \text{where } (P^{(l)})_{ijk} &= \frac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^{(l-1)}} \\ &= \delta_{ij} x_k^{(l-1)}\end{aligned}$$

It can be observed that $P^{(l)}$ is a sparse matrix, with few elements non-zero. More specifically, one of the diagonals (where $i = j$) in this tensor will contain the vector $x^{(l-1)}$ repeated many times.

1.1 b)

Softmax Module

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \tilde{x}^{(N)}} &= \frac{\partial \mathcal{L}}{\partial x^{(N)}} \cdot \frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} \\ &= \begin{bmatrix} 0 \\ \dots \\ 0 \\ -\frac{1}{x_i^{(N)}} \\ 0 \\ \dots \\ 0 \end{bmatrix}^T \cdot \begin{bmatrix} x_1(1-x_1) & -x_1x_2 & \dots & -x_1x_{d_N} \\ -x_2x_1 & -x_2(1-x_2) & \dots & -x_2x_{d_N} \\ \dots & \dots & \dots & \dots \\ -x_{d_N}x_1 & -x_{d_N}x_2 & \dots & x_{d_N}(1-x_{d_N}) \end{bmatrix}\end{aligned}$$

ReLU Module

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \tilde{x}^{(l<N)}} &= \frac{\partial \mathcal{L}}{\partial x^{(l)}} \cdot \frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}} \\ &= \underbrace{\frac{\partial \mathcal{L}}{\partial x^{(l)}}}_{1 \times d_l \text{ vector}} \cdot \begin{bmatrix} \mathbb{1}(\tilde{x}_1^l > 0) & 0 & \dots & 0 \\ 0 & \mathbb{1}(\tilde{x}_2^l > 0) & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \mathbb{1}(\tilde{x}_{d_l}^l > 0) \end{bmatrix}\end{aligned}$$

Linear Module

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial b^{(l)}} &= \frac{\partial \mathcal{L}}{\partial \tilde{x}^{(l)}} \cdot \frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} \\ &= \frac{\partial \mathcal{L}}{\partial \tilde{x}^{(l)}} \cdot \mathbb{I}_{d_l} \\ &= \frac{\partial \mathcal{L}}{\partial \tilde{x}^{(l)}}\end{aligned}$$

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

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W^{(l)}} &= \frac{\partial \mathcal{L}}{\partial \tilde{x}^{(l)}} \cdot \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} \\ &= \frac{\partial \mathcal{L}}{\partial \tilde{x}^{(l)}} \cdot P^{(l)} \quad \text{(with } P^{(l)} \text{ defined above)}\end{aligned}$$

For the last derivation however, it is hard to compute the Jacobian (3D tensors) directly and define multiplication between a vector and itself. However we know that the resulting derivative is a matrix with same shape as $W^{(l)}$; thus we can find another way of finding this derivative, either as dot product of two matrices or the dot product of a column vector and a row vector. This is increasingly important when we use batches, since $P^{(l)}$ will become then a 4D tensor.

Working with a batch size of 1, we can rewrite the derivative as follows:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W_{jk}^{(l)}} &= \sum_{i=1}^{d_l} \frac{\partial \mathcal{L}}{\partial \tilde{x}_i^{(l)}} \cdot \frac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^{(l)}} \\ &= \sum_i \frac{\partial \mathcal{L}}{\partial \tilde{x}_i^{(l)}} \delta_{ij} x_k^{(l-1)} \\ &= \frac{\partial \mathcal{L}}{\partial \tilde{x}_j^{(l)}} \cdot x_k^{(l-1)}\end{aligned}$$

$$\begin{aligned}\Rightarrow \frac{\partial \mathcal{L}}{\partial W^{(l)}} &= \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \tilde{x}_1^{(l)}} \cdot x_1^{(l-1)} & \frac{\partial \mathcal{L}}{\partial \tilde{x}_1^{(l)}} \cdot x_2^{(l-1)} & \dots & \frac{\partial \mathcal{L}}{\partial \tilde{x}_1^{(l)}} \cdot x_{d_{l-1}}^{(l-1)} \\ \frac{\partial \mathcal{L}}{\partial \tilde{x}_2^{(l)}} \cdot x_1^{(l-1)} & \frac{\partial \mathcal{L}}{\partial \tilde{x}_2^{(l)}} \cdot x_2^{(l-1)} & \dots & \frac{\partial \mathcal{L}}{\partial \tilde{x}_2^{(l)}} \cdot x_{d_{l-1}}^{(l-1)} \\ \dots & \dots & \dots & \dots \\ \frac{\partial \mathcal{L}}{\partial \tilde{x}_{d_l}^{(l)}} \cdot x_1^{(l-1)} & \frac{\partial \mathcal{L}}{\partial \tilde{x}_{d_l}^{(l)}} \cdot x_2^{(l-1)} & \dots & \frac{\partial \mathcal{L}}{\partial \tilde{x}_{d_l}^{(l)}} \cdot x_{d_{l-1}}^{(l-1)} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \tilde{x}_1^{(l)}} \\ \frac{\partial \mathcal{L}}{\partial \tilde{x}_2^{(l)}} \\ \dots \\ \frac{\partial \mathcal{L}}{\partial \tilde{x}_{d_l}^{(l)}} \end{bmatrix} \cdot \begin{bmatrix} x_1^{(l-1)} & x_2^{(l-1)} & \dots & x_{d_{l-1}}^{(l-1)} \end{bmatrix} \\ &= \underbrace{\left(\frac{\partial \mathcal{L}}{\partial \tilde{x}^{(l)}}\right)^T \cdot x^{(l-1)}}_{(d_l \times 1) \cdot (1 \times d_{l-1}) = (d_l \times d_{l-1})}\end{aligned}$$

1.1 c)

For a batch size $B \neq 1$, all the equations presented above get an additional dimension: vectors become matrices, matrices become 3D tensors, 3D tensors become 4D tensors, and so on. Instead of being a vector, input $X^{(0)}$ now is a matrix with size $B \times D$ (batch size $\times$ input dimension). In CrossEntropy module, the gradient with respect to $X^{(N)}$ becomes a matrix where each row has a non-zero entry on the position where each respective t vector is 1.

The tricky part comes in the Softmax module, where for each row $x^{(N)}$ in $X^{(N)}$, a corresponding matrix $\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}}$ needs to be computed. This can be managed by expanding $X^{(N)}$ across one dimension such that the resulting shape is (batch size $\times$ 1 $\times$ d_N) and working with 3D tensors. Thus, the operation (diag$(x^{(N)}) - x^{(N)} \cdot {x^{(N)}}^T$), applied on each row $x^{(N)}$, creates a tensor with shape (batch size $\times$ d_N $\times$ d_N). Taking the dot product of the two 3D tensors, we get a tensor of shape (batch size $\times$ 1 $\times$ d_N), which through reshaping is brought to the final shape of (batch size $\times$ d_N).

Another important part to be mentioned is in the Linear module where the incoming gradient has shape (batch size $\times$ d_l). For the bias, we need to sum over the batch data points (rows) to get the final gradient values. The computation of the gradient with respect to the weights of the module is similar to the one presented above.

$$\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \mathbf{1}_B^T \underbrace{\frac{\partial \mathcal{L}}{\partial \tilde{X}^{(l)}}}_{(B \times d_l)}$$

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \underbrace{\left(\frac{\partial \mathcal{L}}{\partial \tilde{X}^{(l)}}\right)^T \cdot X^{(l-1)}}_{(d_l \times B) \cdot (B \times d_{l-1}) = (d_l \times d_{l-1})}$$

1.2

The default values of parameters result in a simple MLP which reaches a decent accuracy of 48.65% on the test set. The accuracy improves gradually, starting around 36% and exceeding 44% in 600 steps. The training loss is decreasing throughout the whole training process; however, we can see it has high variance. The resulting accuracy and loss curves are presented below in figures 1 and 2.

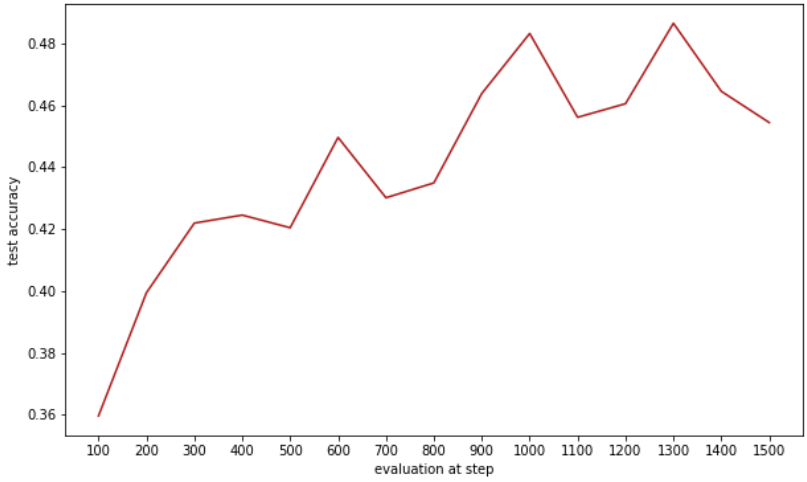

Figure 1: MLP NumPy - accuracy curve

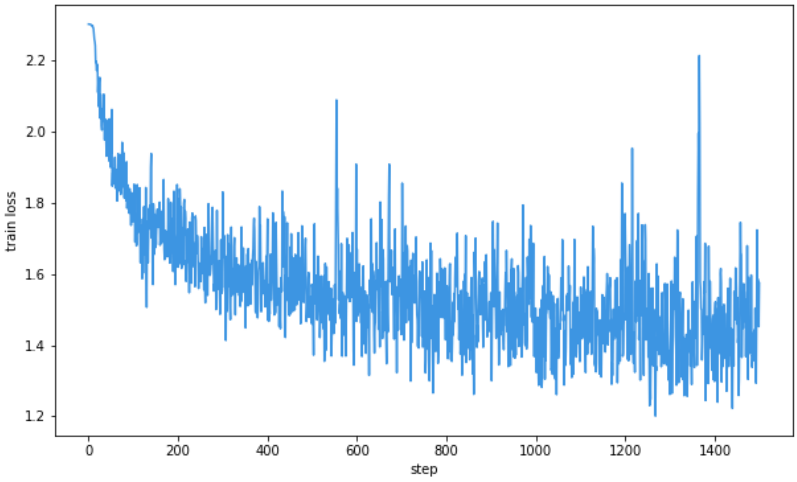

Figure 2: MLP NumPy - training loss curve

2 PyTorch MLP

My implementation of MLPs in PyTorch using the default parameters results in accuracy values lower than the NumPy implementation. Figures 3 and 4 provide an overview of the accuracy score and the training loss. Since the maximum accuracy value on test set achieved by this network is 42.3%, this suggests that the initialisation of the weights and the choice of optimiser influences a lot how the network performs. It can also be observed that the training loss has a sharp peak in the first 100 steps.

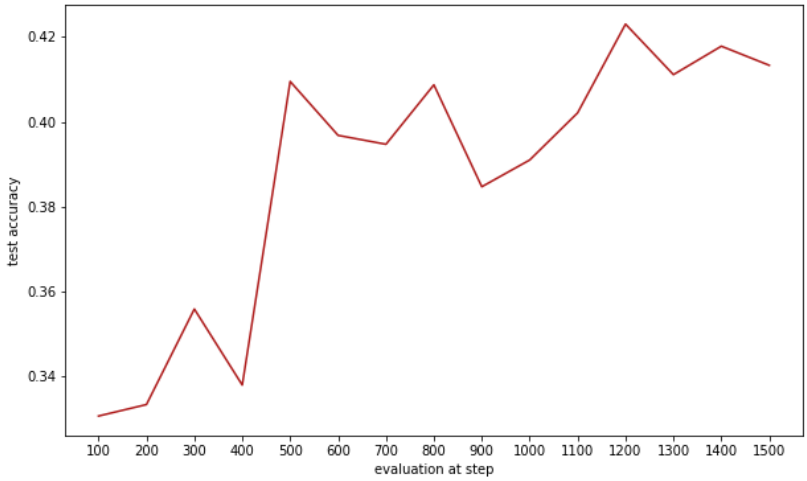

Figure 3: PyTorch default parameters - accuracy curve

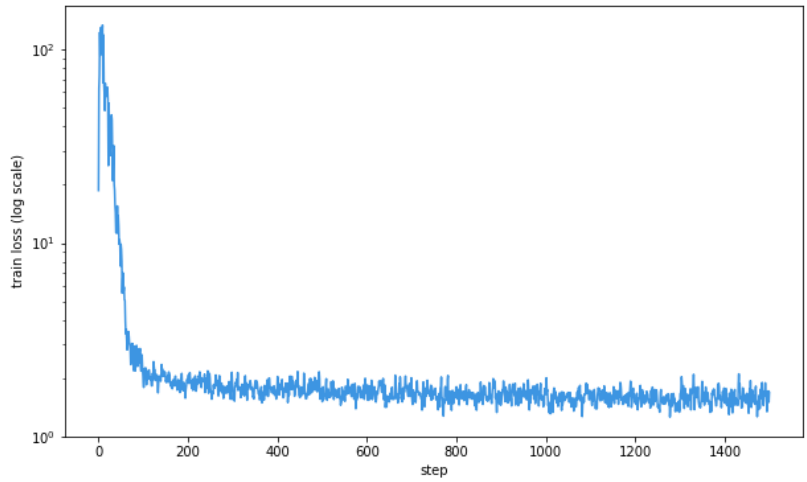

Figure 4: PyTorch default parameters - training loss curve

Improving MLP PyTorch

In an effort to improve the performance of the MLP, I have tried and trained a various number of models, which differ in network architecture and hyperparameter setup. Here I cover only the ones which present the most interesting findings. All models were trained with batch size of 200, making 250 training steps equivalent to one epoch (one complete pass through the train set). Being limited by the performance of my laptop, deeper models could not have been trained in due time, although this could have increased the overall performance.

Starting at an accuracy of 42.3% for the MLP with one layer of size 100 and trained on 1500 steps, my first try was to increase the size of the network, train it on more steps and use a different optimiser. Thus, the model composed of 4 layers with sizes $[1000, 500, 200, 200]$ was trained on 3000 steps; for the optimiser, I have used Adam because of its benefits posed by using different adaptive learning rates for parameters and also taking into consideration past gradients. With learning rate of $2\text{e}{-3}$, the best evaluated accuracy score for this model is 52%. A slight increase in performance is achieved by increasing the size of the network to $[1024, 512, 256, 128]$ and also training it on more steps (4500), resulting in an accuracy of 53.8%.

Without managing to pass this rate only by changing the hyper-parameters, it was pointed out to me by one of my friends that weight normalisation and dropout layers could bump up the performance to accuracy over 56%. It's interesting how the idea of disassociating the magnitude of the weight tensors from their direction can actually improve the network. Additionally, I have introduced dropout layers to avoid overfitting of the large neural networks.

Implementing these two concepts, one of the models has 5 layers of size $[1000, 500, 250, 200, 100]$, and by training it on 5000 steps with Adam optimiser and learning rate $1\mathrm{e}{-3}$, the best accuracy achieved was 56.63%. However, the best model proved to be the one composed of 7 layers with sizes $[2048, 1024, 1024, 512, 512, 256, 256]$ and trained on 7500 steps with Adam optimiser and learning rate $1\mathrm{e}{-3}$. This model achieved 57.42% accuracy, the best performance of all tested MLPs. Figures 5 and 6 below present the curves for accuracy tests and training loss for this model.

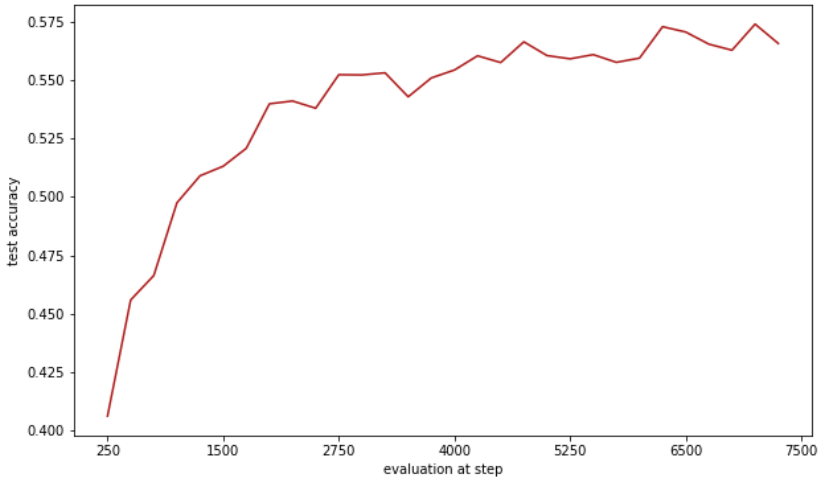

Figure 5: PyTorch best - accuracy curve

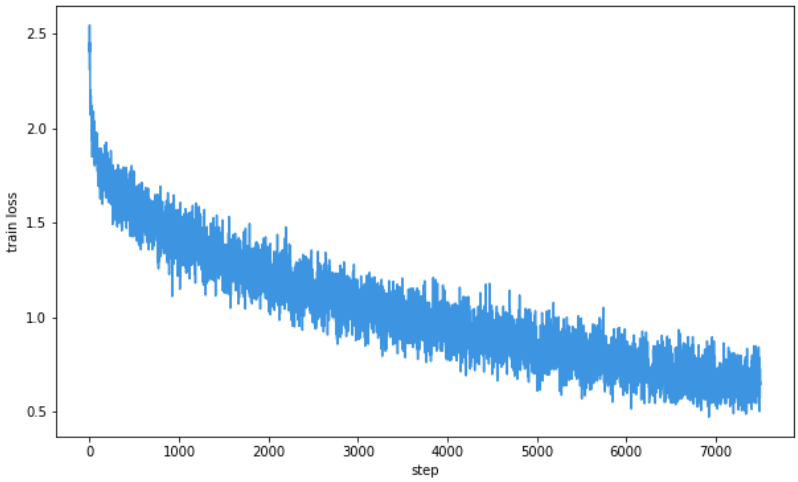

Figure 6: PyTorch best - training loss curve

3 Custom Module: Batch Normalization

Derivative for γ

$$\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{\gamma}}\right)_j = \sum_s \sum_i \frac{\partial \mathcal{L}}{\partial y_i^s} \frac{\partial y_i^s}{\partial \gamma_j} = \sum_s \sum_i \frac{\partial \mathcal{L}}{\partial y_i^s} \delta_{ij} \hat{x}_i^s = \sum_s \frac{\partial \mathcal{L}}{\partial y_i^s} \hat{x}_i^s$$

$$\Rightarrow \frac{\partial \mathcal{L}}{\partial \boldsymbol{\gamma}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{Y}} \cdot \frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{\gamma}}$$

$$= \sum_{s=1}^{B} \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}^s} \odot \hat{\boldsymbol{x}}^s$$

Derivative for β

$$\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{\beta}}\right)_j = \sum_s \sum_i \frac{\partial \mathcal{L}}{\partial y_i^s} \frac{\partial y_i^s}{\partial \beta_j} = \sum_s \sum_i \frac{\partial \mathcal{L}}{\partial y_i^s} \delta_{ij} \cdot 1 = \sum_s \frac{\partial \mathcal{L}}{\partial y_i^s}$$

$$\Rightarrow \frac{\partial \mathcal{L}}{\partial \boldsymbol{\beta}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{Y}} \cdot \frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{\beta}}$$

$$= \sum_{s=1}^{B} \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}^s}$$

Derivative for X

The element-wise derivation of this part is cumbersome given the multitude of indices involved in summations. Thus, for simplicity and ease of understanding, I work directly with derivations on vectors/matrices. For one $\boldsymbol{x}^s$ (one row in the $\boldsymbol{X}$ matrix), we have the following equation given by the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}^s} = \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}^s} \cdot \frac{\partial \hat{\boldsymbol{x}}^s}{\partial \boldsymbol{x}^s} + \frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}} \cdot \frac{\partial \boldsymbol{\mu}}{\partial \boldsymbol{x}^s} + \frac{\partial \mathcal{L}}{\partial \boldsymbol{\sigma}^2} \cdot \frac{\partial \boldsymbol{\sigma}^2}{\partial \boldsymbol{x}^s}$$

Breaking it down for each element:

$$\frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}^s} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}^s} \cdot \frac{\partial \boldsymbol{y}^s}{\partial \hat{\boldsymbol{x}}^s} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}^s} \odot \boldsymbol{\gamma}$$

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

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}} = \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{X}}} \cdot \frac{\partial \hat{\boldsymbol{X}}}{\partial \boldsymbol{\mu}} + \frac{\partial \mathcal{L}}{\partial \boldsymbol{\sigma}^2} \cdot \frac{\partial \boldsymbol{\sigma}^2}{\partial \boldsymbol{\mu}} \text{ (full derivation below)}$$

$$= \sum_{t=1}^{B} \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}^t} \cdot \frac{-1}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}}$$

$$\frac{\partial \boldsymbol{\mu}}{\partial \boldsymbol{x}^s} = \frac{\partial}{\partial \boldsymbol{x}^s} \frac{1}{B} \sum_{i=1}^{B} \boldsymbol{x}^i = \frac{1}{B}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\sigma}^2} = \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{X}}} \cdot \frac{\partial \hat{\boldsymbol{X}}}{\partial \boldsymbol{\sigma}^2}$$

$$= -\frac{1}{2} \sum_{t=1}^{B} \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}^t} \cdot (\boldsymbol{x}^t - \boldsymbol{\mu})(\boldsymbol{\sigma}^2 + \epsilon)^{-1.5}$$

$$\frac{\partial \boldsymbol{\sigma}^2}{\partial \boldsymbol{x}^s} = \frac{\partial}{\partial \boldsymbol{x}^s} \frac{1}{B} \sum_{i=1}^{B} (\boldsymbol{x}^i - \boldsymbol{\mu})^2$$

$$= \frac{2}{B} (\boldsymbol{x}^s - \boldsymbol{\mu})$$

Full derivation with respect to $\boldsymbol{\mu}$:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}} &= \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{X}}} \cdot \frac{\partial \hat{\boldsymbol{X}}}{\partial \boldsymbol{\mu}} + \frac{\partial \mathcal{L}}{\partial \boldsymbol{\sigma}^2} \cdot \frac{\partial \boldsymbol{\sigma}^2}{\partial \boldsymbol{\mu}} \\
&= \left(\sum_{t=1}^{B} \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}^t} \cdot \frac{-1}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}} \right) + \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{\sigma}^2} \cdot \frac{-2}{B} \sum_{t=1}^{B} (\boldsymbol{x}^t - \boldsymbol{\mu}) \right) \\
&= \left(\sum_{t=1}^{B} \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}^t} \cdot \frac{-1}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}} \right) + \left[\frac{\partial \mathcal{L}}{\partial \boldsymbol{\sigma}^2} \cdot (-2) \underbrace{\left(\underbrace{\frac{1}{B} \sum_{t=1}^{B} \boldsymbol{x}^t}_{=\boldsymbol{\mu}} - \underbrace{\frac{1}{B} \sum_{t=1}^{B} \boldsymbol{\mu}}_{=\boldsymbol{\mu}} \right)}_{=0} \right] \\
&= \sum_{t=1}^{B} \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}^t} \cdot \frac{-1}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}}
\end{aligned}$$

Putting it all together, we get:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}^s} &= \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}^s} \cdot \frac{1}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}} + \frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}} \cdot \frac{1}{B} + \frac{\partial \mathcal{L}}{\partial \boldsymbol{\sigma}^2} \cdot \frac{2(\boldsymbol{x}^s - \boldsymbol{\mu})}{B} \\
&= \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}^s} \cdot \frac{1}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}} + \frac{1}{B} \sum_{t=1}^{B} \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}^t} \cdot \frac{-1}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}} - \frac{\boldsymbol{x}^s - \boldsymbol{\mu}}{B} \sum_{t=1}^{B} \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}^t} \cdot (\boldsymbol{x}^t - \boldsymbol{\mu})(\boldsymbol{\sigma}^2 + \epsilon)^{-1.5} \\
&= (\boldsymbol{\sigma}^2 + \epsilon)^{-0.5} \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}^s} - \frac{(\boldsymbol{\sigma}^2 + \epsilon)^{-0.5}}{B} \sum_{t=1}^{B} \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}^t} - \frac{(\boldsymbol{\sigma}^2 + \epsilon)^{-0.5}}{B} \underbrace{\frac{\boldsymbol{x}^s - \boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}}}_{\hat{\boldsymbol{x}}^s} \sum_{t=1}^{B} \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}^t} \cdot \underbrace{\frac{\boldsymbol{x}^t - \boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}}}_{\hat{x}^t} \\
&= \frac{(\boldsymbol{\sigma}^2 + \epsilon)^{-0.5}}{B} \left[B \cdot \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}^s} - \sum_{t=1}^{B} \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}^t} - \hat{\boldsymbol{x}}^s \sum_{t=1}^{B} \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}^t} \cdot \hat{\boldsymbol{x}}^t \right]
\end{aligned}$$

Generalising to whole $\boldsymbol{X}$:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{X}} = \frac{(\boldsymbol{\sigma}^2 + \epsilon)^{-0.5}}{B} \left(B \cdot \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{X}}} - \mathbf{1}_B^T \cdot \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{X}}} - \hat{\boldsymbol{X}} \odot \left(\underbrace{\mathbf{1}_B \cdot \left(\underbrace{\mathbf{1}_B^T \cdot \left(\frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{X}}} \odot \hat{\boldsymbol{X}} \right)}_{\text{summation over batch}} \right)}_{\text{create matrix by repeating the row vector}} \right) \right)$$

4 PyTorch CNN

Implementing a convolutional neural network with PyTorch, we can see how powerful they are at image classification compared to the usual MLPs. The smaller version of VGG reaches the highest accuracy of 76.92% in only $4,000$ steps. Both accuracy and loss curves can be visualised in figures 7 and 8.

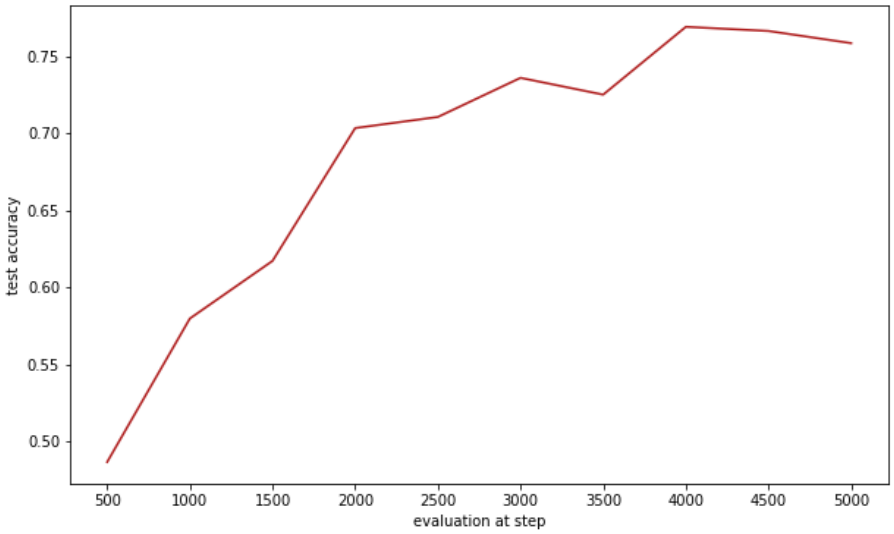

Figure 7: PyTorch ConvNet - accuracy curve

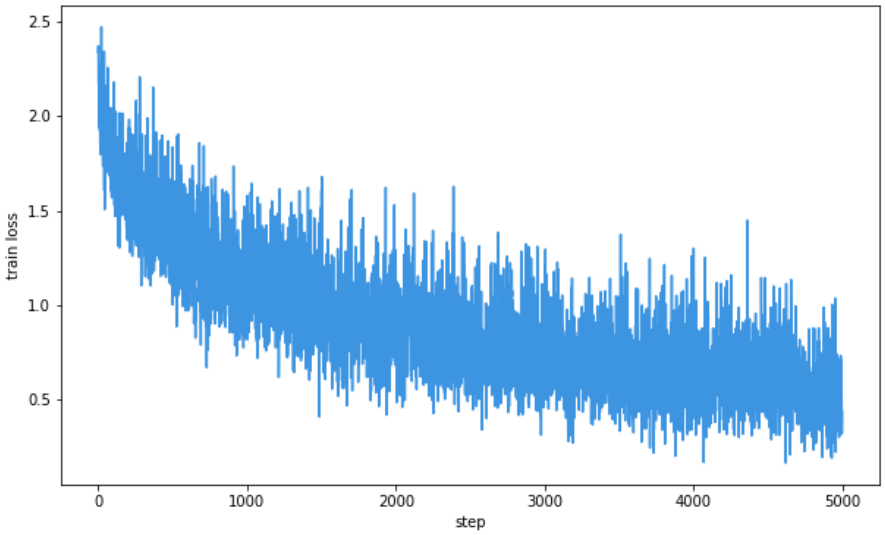

Figure 8: PyTorch ConvNet - training loss curve