Deep Learning - Assignment 1: MLPs, CNNs and Backpropagation

Ioannis Gatopoulos 12141666

April 19, 2019

Assignment 1 of Deep Learning Course 2019
MSc Artificial Intelligence • University of Amsterdam

Contents

1 MLP backprop and NumPy implementation

1.1 Analytical derivation of gradients

Question 1.1 (a)

i) Matrix-valued derivatives:

$$\frac{\partial \mathcal{L}}{\partial x^{(N)}} = \frac{\partial}{\partial x^{(N)}}\Big(-\log x^{(N)}_{\arg\max(t)}\Big) = \frac{1}{x^{(N)}_{\arg\max(t)}}$$

Scalar gradients:

$$\left(\frac{\partial \mathcal{L}}{\partial x^{(N)}}\right)_i = \left(\frac{\partial \mathcal{L}}{\partial x^{(N)}_i}\right) = \frac{\partial}{\partial x^{(N)}_i}\Big(-\sum_j t_j \log x^{(N)}_j\Big) = -t_i \frac{\partial}{\partial x^{(N)}_i} \log x^{(N)}_i = -\frac{t_i}{x^{(N)}_i}$$

thus

$$\frac{\partial \mathcal{L}}{\partial x^{(N)}} = -\boldsymbol{t} \cdot \frac{\mathbf{1}}{\boldsymbol{x^{(N)}}} = \begin{pmatrix} t_1 \\ \vdots \\ t_N \end{pmatrix} \cdot \begin{pmatrix} x^{(N)}_1 \\ \vdots \\ x^{(N)}_N \end{pmatrix}$$

ii) Scalar gradients:

$$\left(\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}}\right)_{j,i} = \frac{\partial}{\partial \tilde{x}_i^{(N)}} \frac{\exp\left(\tilde{x}_j^{(N)}\right)}{\sum_{k=1}^{dN} \exp\left(\tilde{x}_k^{(N)}\right)}$$

We distinguish two possible cases:

- if $i = j$

$$\begin{aligned}\left(\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}}\right)_{i,i} &= \frac{\exp\left(\tilde{x}_i{}^{(N)}\right) \cdot \sum_{k=1}^{dN} \exp\left(\tilde{x}_k^{(N)}\right) - \exp\left(\tilde{x}_i^{(N)}\right) \cdot \exp\left(\tilde{x}_i^{(N)}\right)}{\left(\sum_{k=1}^{dN} \exp\left(\tilde{x}_k^{(N)}\right)\right)^2} \\ &= \frac{\exp\left(\tilde{x}_i^{(N)}\right)}{\sum_{k=1}^{dN} \exp\left(\tilde{x}_k^{(N)}\right)} - \left(\frac{\exp\left(\tilde{x}_i^{(N)}\right)}{\sum_{i=1}^{dN} \exp\left(\tilde{x}_k^{(N)}\right)}\right)^2 \\ &= x_i^{(N)} - \left(x_i^{(N)}\right)^2 = x_i^{(N)} - \left(x_j^{(N)}\right)^2\end{aligned}$$

- if $i \neq j$

$$\begin{aligned}\left(\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}}\right)_{j,i} &= \frac{0 - \exp\left(\tilde{x}_i^{(N)}\right) \cdot \exp\left(\tilde{x}_j^{(N)}\right)}{\left(\sum_{k=1}^{dN} \exp\left(\tilde{x}_k^{(N)}\right)\right)^2} \\ &= -\frac{\exp\left(\tilde{x}_i^{(N)}\right)}{\sum_{k=1}^{dN} \exp\left(\tilde{x}_k^{(N)}\right)} \cdot \frac{\exp\left(\tilde{x}_j^{(N)}\right)}{\sum_{i=1}^{dN} \exp\left(\tilde{x}_k^{(N)}\right)} \\ &= -x_i^{(N)} \cdot x_j^{(N)}\end{aligned}$$

Grouping them together, we will have:

$$\begin{aligned}\left(\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}}\right) &= \begin{cases} x_i^{(N)} - \left(x_j^{(N)}\right)^2 & \text{if} \quad i = j \\ -x_i^{(N)} \cdot x_j^{(N)} & \text{if} \quad i \neq j \end{cases} \\ &= x_i^{(N)}\left(\delta_{ij} - x_j^{(N)}\right)\end{aligned}$$

where δ_{ij} is the **Kronecker** delta.

Therefore, in matrix formation the result would be:

$$\left(\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}}\right)_{j,i} = \operatorname{diag}\left(x^{(N)}\right) - x^{(N)} x^{(N)^T}$$

iii)

$$\frac{\partial x^{(l<N)}}{\partial \tilde{x}^{(l<N)}} = \frac{\partial \max\left(0, \tilde{x}^{(l)}\right)}{\partial \tilde{x}^{(l)}} = \operatorname{diag}\left(\mathbf{I}_{\tilde{x}^{(l)}>0}\right)$$

iv)

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

v) Assuming that $\mathbf{W}^{(l)} \in \mathcal{R}^{M\times N}$ and $x^{(l-1)} \in \mathcal{R}^{N\times B}$, we are going to derive the above by implementing a *trick*; rather to derive wrt matrix $\mathbf{W}^{(l)} = \begin{bmatrix}\mathbf{w}_1^{(l)} & \mathbf{w}_2^{(l)} & \dots & \mathbf{w}_N^{(l)}\end{bmatrix}$, where $\mathbf{w}_1^{(l)} = \begin{bmatrix}w_{11}^{(l)} & w_{12}^{(l)} & \dots & w_{1M}^{(l)}\end{bmatrix}^T$, we are going to derive it with $\mathbf{w}_*^{(l)}$ instead. Therefore, we are going to have:

$$\frac{\partial \tilde{x}^{(l)}}{\partial \mathbf{w}_1^{(l)}} = W^{(l)}\frac{\partial}{\partial \mathbf{w}_1^{(l)}}x^{(l-1)} + x^{(l-1)^T}\frac{\partial}{\partial \mathbf{w}_1^{(l)}}W^{(l)} = x^{(l-1)^T}\frac{\partial}{\partial \mathbf{w}_1^{(l)}}W^{(l)}$$

The last term would be

$$\frac{\partial}{\partial \mathbf{w}_1^{(l)}}W^{(l)} = \begin{bmatrix}\frac{\partial W^{(l)}}{\partial \mathbf{w}_{11}^{(l)}} & \frac{\partial W^{(l)}}{\partial \mathbf{w}_{12}^{(l)}} & \dots & \frac{\partial W^{(l)}}{\partial \mathbf{w}_{1M}^{(l)}}\end{bmatrix} = \begin{bmatrix}\begin{bmatrix}1 & 0 & \dots & 0\\ 0 & 0 & \dots & 0\\ \dots & \dots & \dots & \dots\end{bmatrix}\\ \vdots \\ \begin{bmatrix}0 & 0 & \dots & 1\\ 0 & 0 & \dots & 0\\ \dots & \dots & \dots & \dots\end{bmatrix}\end{bmatrix} = \begin{bmatrix}\mathbf{I_{11}}\\ \vdots \\ \mathbf{I_{1N}}\end{bmatrix} \Rightarrow \frac{\partial W^{(l)}}{\partial \mathbf{w}_*^{(l)}} = \begin{bmatrix}\mathbf{I_{*1}}\\ \vdots \\ \mathbf{I_{*N}}\end{bmatrix} \in \mathcal{R}^{N\times(M\times N)}$$

Thus,

$$x^{(l-1)^T}\frac{\partial}{\partial \mathbf{w}_1^{(l)}}W^{(l)} = \begin{bmatrix}x_1^{(l-1)} & x_2^{(l-1)} & \dots & x_N^{(l-1)}\end{bmatrix}\begin{bmatrix}\mathbf{I_{*1}}\\ \vdots \\ \mathbf{I_{*N}}\end{bmatrix} = \begin{bmatrix}0 & 0 & \dots & 0\\ \dots & \dots & \dots & \dots\\ x_1^{(l-1)} & x_2^{(l-1)} & \dots & x_N^{(l-1)}\\ \dots & \dots & \dots & \dots\\ 0 & 0 & \dots & 0\end{bmatrix} = \Phi_*$$

And the result would be

$$\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} = \begin{bmatrix}\Phi_1 & \Phi_2 & \dots & \Phi_M\end{bmatrix}$$

Thus, having a batch of one, will result in:

$$\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} = \left(x^{(l-1)}\right)^T$$

*** Note:** As we saw, with this derivation we ended up with a high-dimensional sparse vector. This is the problem that the backpropagation address; it actually do not numerically compute this derivative but it derives it by back-propagating the error terms δ.

vi)

$$\frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} = \frac{\partial}{\partial b^{(l)}}\left(W^{(l)}x^{(l-1)} + b^{(l)}\right) = \mathbf{I}$$

Question 1.1 (b)

Lets define first the term $\delta_j^{(l)} \equiv \frac{\partial L}{\partial \tilde{x}_j^{(l)}}$ (or $\boldsymbol{\delta}^{(l)} \equiv \frac{\partial L}{\partial \tilde{\boldsymbol{x}}^{(l)}}$ in matrix formation) as **error term**.

i) Assuming that class c would be the ground truth class, we would have

$$\mathcal{L} = -\sum_i t_i \log x_i^{(L)} = -t_c \log x_c^{(L)} = -t_c \log\left(\frac{\exp\left(\tilde{x}_c^{(L)}\right)}{\sum_{i=1}^{d_L}\exp\left(\tilde{x}^{(L)}\right)_i}\right) = -t_c\,\tilde{x}_c^{(L)} + \log\sum_{i=1}^{d_L}\exp\left(\tilde{x}^{(L)}\right)_i$$

Now taking the derivative wrt $\tilde{x}_j^{(L)}$ we would have

$$\frac{\partial \mathcal{L}}{\partial \tilde{x}_j^{(L)}} = -t_c\, \delta_{cj} + \frac{1}{\sum_{i=1}^{d_L} \exp\left(\tilde{x}^{(L)}\right)_i} \cdot \exp\left(\tilde{x}_j^{(L)}\right) = -t_j + x_j^{(L)}$$

Thus, we will end up with

$$\frac{\partial \mathcal{L}}{\partial \tilde{x}^{(L)}} = \frac{\partial \mathcal{L}}{\partial x^{(L)}} \frac{\partial x^{(L)}}{\partial \tilde{x}^{(L)}} = \boldsymbol{x^{(L)} - t} \equiv \boldsymbol{\delta}^{(L)}$$

* Another way to do it is to just multiply the above derivatives.

$$\frac{\partial \mathcal{L}}{\partial \tilde{x}^{(N)}} = \frac{\partial \mathcal{L}}{\partial x^{(N)}} \frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} = \left(-\boldsymbol{t} \cdot \boldsymbol{x^{(N)^{-1}}}\right) \cdot \left(\text{diag}\left(x^{(N)}\right) - x^{(N)} x^{(N)^T}\right) \equiv \boldsymbol{\delta}^{(L)}$$

ii) In scalar notation we would have

$$\frac{\partial \mathcal{L}}{\partial \tilde{x}_j^{(l<N)}} = \sum_k \frac{\partial \mathcal{L}}{\partial \tilde{x}_k^{(l+1)}} \frac{\partial \tilde{x}_k^{(l+1)}}{\partial \tilde{x}_j^{(l)}} = \sum_k \delta_k^{(l+1)} \frac{\partial \tilde{x}_k^{(l+1)}}{\partial \tilde{x}_j^{(l)}} = \sum_k \delta_k^{(l+1)} \sum_i w_{ki}^{(l+1)} \frac{\partial x_i^{(l)}}{\partial \tilde{x}_j^{(l)}} =$$
$$= \sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)} 1_j^{(l)} = 1_j^{(l)} \sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)} \equiv \delta_k^{(l)}$$

And in vector

$$\frac{\partial \mathcal{L}}{\partial \tilde{x}^{(l<N)}} = \frac{\partial \mathcal{L}}{\partial \tilde{x}^{(l+1)}} \frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}} \frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}} = \left(W^{(l+1)}\right)^T \times \boldsymbol{\delta}^{(l+1)} \cdot \text{diag}\left(\mathbf{I}_{\tilde{x}^{(l)}>0}\right) \equiv \boldsymbol{\delta}^{(l)}$$

iii)

$$\frac{\partial \mathcal{L}}{\partial x^{(l<N)}} = \frac{\partial \mathcal{L}}{\partial \tilde{x}^{(l+1)}} \frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}} = \left(W^{(l+1)}\right)^T \frac{\partial \mathcal{L}}{\partial \tilde{x}^{(l+1)}} = \left(W^{(l+1)}\right)^T \boldsymbol{\delta}^{(l+1)}$$

iv)

$$\frac{\partial \mathcal{L}}{\partial W(l)} = \frac{\partial \mathcal{L}}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} = \frac{\partial \mathcal{L}}{\partial \tilde{x}^{(l)}} \left(x^{(l-1)}\right)^T = \boldsymbol{\delta}^{(l)} \left(x^{(l-1)}\right)^T$$

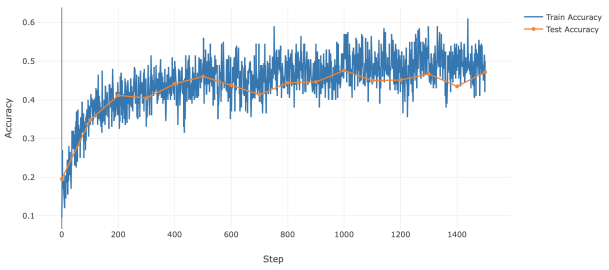
v)

$$\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \frac{\partial \mathcal{L}}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} = \frac{\partial \mathcal{L}}{\partial \tilde{x}^{(l)}} \mathbf{I} = \frac{\partial \mathcal{L}}{\partial \tilde{x}^{(l)}} = \boldsymbol{\delta}^{(l)}$$

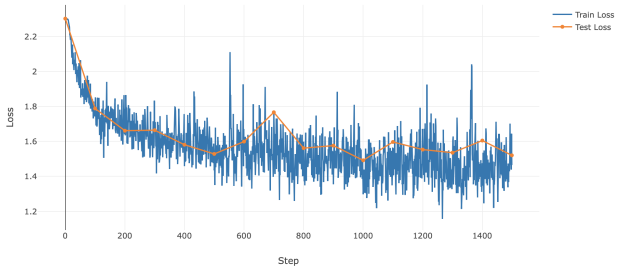
Question 1.1 (c)

If a batch size bigger than one is used (mini-batch), cross-entropy loss will be averaged over the batch and return a scalar. The backpropagation matrix will expand by an additional dimension, which corresponds to the batch size. Thus, the dimentionality of the error terms would be $B \times d_{(l)}$.

Due to the fact that every neuron is independent by all the others that belongs to the same layer, they will have separate gradients for every batch. So the derivatives wrt $\tilde{x}^{(l)}$ and $x^{(l)}$ will have again an extra dimension of batch size. However, for the weights gradients, we will sum over the batch dimension, taking into account the influence of each neuron of the previous layer that ends up to it and thus, the dimensions will not change. This is because now the neurons of layer (l) decent on all those of of layer $(l-1)$



(a) Accuracy performance of `mlp_numpy` model



(b) Loss performance of `mlp_numpy` model

Figure 1: Performance of NumPy MLP implementation. The evaluation on the Test set took place every 100 steps.

1.2 NumPy implementation

We implemented a multi-layer perceptron implimentation using purely NumPy routines. We call it `mlp_numpy`. The architecture and hyperparameters that was used can be seen at table 1.

Implementation	Hidden units	Optimizer	Learning rate	Batch size
`mlp_numpy`	100	SGD	2e-3	200

Table 1: `mlp_numpy` architecture and hyperparameters.

Figure 1 illustrates the accuracy and the loss of `mlp_numpy` and the table 2 its best performance on the test and train dataset. As we can see, the model performs very poorly and it fails to generalize (its accuracy is equally a with random pick of classes for every image, just like a random model). Reasons behind its performance is the number of hidden units and the depth of the neural network (NN). Both are very small and the model fails to detect meaningful and complex patents of the data. Another reason is that the model was not tuned identically, in order to perform the best with this architecture.

Accuracy		**Loss**	
Train	**Test**	**Train**	**Test**
0.61	0.48	1.16	1.50

Table 2: `mlp_numpy`'s best performance on the test and train dataset.

2 PyTorch MLP

The goal of this part is to make use of the PyTorch framework and by changing the architecture and hyperparameters of the MLP to achieve better results.

Take into account the previous observations, our first action was to make the network deeper. Because, in general, image data have complex data patterns, a deeper NN can find patterns within the patterns and hopefully can find keypoints that distinguishes one class from the others. We followed the convention of layers in power of two, and decreasing it as we are going deeper. Therefore, the "1024, 512, 128, 64, 32" was choosen as default.

Due to the complexity of the data, '**Adam**' seemed the optimal choice, since it outperforms all other types of optimizers in a complex, non-convex search space. However, we experiment also with SGD, with momentum 0.9 and nesterov option enabled, which also performed very close to Adam. The choice of the learning rate as well as this of weight decay, was chosen by performing grid search on first on the spases (1e-02 - 1e-07) and then (1e-05, 9e-05) for the former, and on the space (1e-09 - 1e-11).

Hidden units	Optimizer	Learning rate	Weight decay	Batch size	Accuracy		Loss	
					Train	Test	Train	Test
100	SGD	5e-03	1e-5	200	0.62	0.53	1.09	1.35
100	Adam	5e-05	1e-10	200	0.65	0.54	1.28	1.31
1024, 512, 128, 64, 32	SGD	5e-03	1e-5	200	0.66	0.56	0.98	1.25
512, 128, 64	SGD	5e-03	1e-5	200	0.64	0.57	1.02	1.22
1000, 800, 200	Adam	5e-05	1e-10	200	0.97	0.58	0.11	1.19
4000, 1000, 4000*	Adam	5e-05	1e-10	200	1.0	0.586	0.02	1.22
4000, 1000, 800, 200*	Adam	4e-05	1e-10	256	1.0	0.587	0.02	1.21
1024, 512, 128, 64, 32	Adam	5e-05	2e-10	128	0.86	**0.59**	0.51	1.23

Table 4: `mlp_pytorch`'s performance with different hyperparameters. All of them also include drop-out regularization to every layer with probability 0.2 and every SGD optimizer had momentum of 0.9 and nesterov movement enabled. The last one is the best one. Those with * are inspired by [2].

	3e-05	4e-05	5e-5	6e-05
32	0.537	0.545	0.551	0.551
64	0.567	0.569	0.575	0.578
128	0.582	0.583	**0.590**	0.589
256	0.577	0.588	0.586	0.585

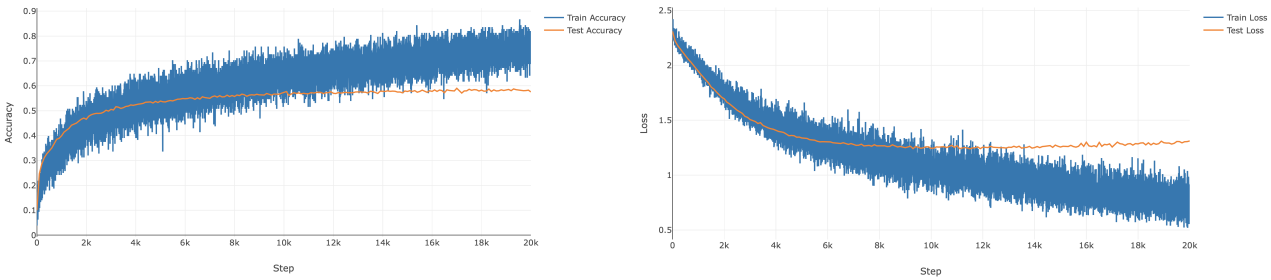
Table 3: `mlp_pytorch`'s Test accuracies with various hyperparameters – batch size (left) and learning rate (top). Though grid search the optimal combination is learning rate $= 5e-5$ and batch size $= 128$. The training process took 20k steps for each.

Since our model is a fully-connected NN and the dimension of each sample (image) is large, it is more likely to overfit. That is way is necessary to apply realization techniques to our model. Besides ***L2***, we also apply **Dropout** with probability 0.2 to every layer. For numerical stability, the **Log Softmax** output of the NN and the **Non-Negative Log Likelihood** as criterion was preferred.

Moreover, in order to get the maximum out of the weight updates, we also normalize the inputs of every layer. Another benefit with this is that this **normalization** smoothness the affect of the outliers. Finally, the **data palatalization** option of our model was choose, which gave an extra boost to its performance.

On Table 4 we provide with the most interested architectures and hyperparameters that were tried. Firstly, from the first NN, we can see that even that we use the same architecture as the `mlp_numpy` model, we achiever a better performance, due to the tuning and the regulations of the model. We focused more on the "bottleneck" approach, (first layers are on larger dimensionality and gradually project it onto the lower-dimensional space) and as we can see, after one point stacking more layers or make them from large to small and then again to large (5th NN) they all fall into the same performance on the test set. We will analyze the last one, since it represent a more balanced version of the previous neural networks, very good test accuracy and loss and is less ; good accuracy and loss on the train test without over-fitting quickly.

In order to pick the optimal hyperparameters for the model, we performed a **grid** search on a range of values. To narrow down the search space, we perform some initial runs for small step size. Table 3 provides the test accuracies of the models. The combination of learning rate $= 5e-5$ and batch size $= 128$ seemed to be the



(a) Accuracy performance of `mlp_pytorch` model

(b) Loss performance of `mlp_pytorch` model

Figure 2: Performance of best model MLP with PyTorch implementation. The evaluation on the Test set took place every 100 steps.

best one.

The performance of the best model is visualized on figure 2. The model run for 20k steps, with the evaluation on the test set to take place every 100 steps. It is very surprising that the model from the beginning until the 1.000th step. A reasons behind it is that the dropout, in constraint during test, during training it slices off some random collection of these classifiers, and thus, the training suffers, while in testing, we use all the nodes of the NN. This is true also for the L2 regularization. This also indicates that the choice of the hyperparameters were actually very good. From the loss plot, it is clear that the model starts to overfit at around step 10k, as the test loss starts to slowly increasing. After that point, the accuracy seems to remain relatively the same. This happens because even though the classifier classifies the images like before, it does not have the same confident. For example, at step 9700, the MLP classified an image of an airplane as airplane with probability, lets say, 0.8. But after that point, it still classify it as an airplane, but now with probability 0.7. Thus, even though the loss is increasing, the accuracy remains the same.

Never the less, we have manage to improve the model by a margin of 10%, but still the performance is far from satisfactory.

3 Custom Module: Batch Normalization

3.1 Automatic differentiation

3.1.1 Manual implementation of backward pass

The Batch Normalization operation implemented on `custom_batchnorm.py` file on the attached code.

Question 3.2 (a)

i)

$$\left(\frac{\partial L}{\partial \gamma}\right)_j = \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \gamma_j} = \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial}{\partial \gamma_j}\left(\gamma_i \hat{x}_i^s + \beta_i\right) = \sum_s \frac{\partial L}{\partial y_j^s}\hat{x}_j^s$$

ii)

$$\left(\frac{\partial L}{\partial \beta}\right)_j = \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \beta_j} = \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial}{\partial \beta_j}\left(\gamma_i \hat{x}_i^s + \beta_i\right) = \sum_s \frac{\partial L}{\partial y_j^s}$$

iii)

$$\left(\frac{\partial L}{\partial x}\right)_j^r = \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial x_j^r}$$

We are going to calculate the latter step by step:

- $\frac{\partial y_i^s}{\partial x_j^r} = \frac{\partial}{\partial x_j^r}\left(\gamma_i \hat{x}_i^s + \beta_i\right) = \gamma_i \frac{\partial \hat{x}_i^s}{\partial x_j^r}$
- $\frac{\partial \hat{x}_i^s}{\partial x_j^r} = \frac{\partial}{\partial x_j^r}\left(\frac{x_i^s - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}\right)$
$= \frac{1}{\sigma_i^2 + \epsilon}\left(\frac{\partial}{\partial x_j^r}\left(x_i^s - \mu_i\right)\sqrt{\sigma_i^2 + \epsilon} - \left(x_i^s - \mu_i\right)\frac{\partial}{\partial x_j^r}\left(\sqrt{\sigma_i^2 + \epsilon}\right)\right)$
- $\frac{\partial}{\partial x_j^r}\left(x_i^s - \mu_i\right) = \delta_{ij}\delta_{rs} - \frac{1}{B}\delta_{ij} = \delta_{ij}\left(\delta_{rs} - \frac{1}{B}\right) \Rightarrow \frac{\partial}{\partial x_i^r}\left(x_i^s - \mu_i\right) = \delta_{rs} - \frac{1}{B}$
- $\frac{\partial}{\partial x_j^r}\left(\sqrt{\sigma_i^2 + \epsilon}\right) = \frac{1}{2\sqrt{\sigma_i^2 + \epsilon}}\frac{\partial}{\partial x_j^r}\left(\sigma_i^2\right)$
- $\frac{\partial \sigma_i^2}{\partial x_j^r} = \frac{\partial}{\partial x_j^r}\left(\frac{1}{B}\sum_{s=1}^{B}\left(x_j^s - \mu_i\right)^2\right)$
$= \frac{1}{B}\sum_s 2\left(x_j^s - \mu_j\right)\cdot\left(\delta_{rs} - \frac{1}{B}\right)$
$= \frac{2\left(x_j^r - \mu_j\right)}{B} \quad \textit{because} \sum_s x_j^s - \mu_j = 0$

Putting them all together we result in the wanted equation.

$$\begin{aligned}
\frac{\partial y_j^s}{\partial x_j^r} &= \frac{\gamma_j}{\sigma_j^2 + \epsilon}\left(\left(\delta_{rs} - \frac{1}{B}\right)\sqrt{\sigma_j^2 + \epsilon} - \left(x_j^s - \mu_j\right)\frac{1}{2\sqrt{\sigma_j^2 + \epsilon}}\frac{2\left(x_j^r - \mu_j\right)}{B}\right) \\
&= \frac{\gamma_j}{B(\sigma_j^2 + \epsilon)}\left(\left(B\delta_{rs} - 1\right)\sqrt{\sigma_j^2 + \epsilon} - \frac{1}{\sqrt{\sigma_j^2 + \epsilon}}\left(\left(x_j^s - \mu_j\right)\cdot\left(x_j^r - \mu_j\right)\right)\right) \\
&= \frac{\gamma_j}{B(\sigma_j^2 + \epsilon)}\left(\left(B\delta_{rs} - 1\right)\sqrt{\sigma_j^2 + \epsilon} - \frac{1}{\sqrt{\sigma_j^2 + \epsilon}}\hat{x}_j^s \cdot \hat{x}_j^r \cdot \left(\sigma_j^2 + \epsilon\right)\right) \\
&= \frac{\gamma_j}{B\sqrt{\sigma_j^2 + \epsilon}}\left(B\delta_{rs} - 1 - \hat{x}_j^s\hat{x}_j^r\right)
\end{aligned}$$

And thus

$$\begin{aligned}
\left(\frac{\partial L}{\partial x}\right)_j^r &= \sum_s \frac{\partial L}{\partial y_j^s}\frac{\gamma_i}{B\sqrt{\sigma_j^2 + \epsilon}}\left(B\delta_{rs} - 1 - \hat{x}_j^s\hat{x}_j^r\right) \\
&= \frac{\gamma_j}{B\sqrt{\sigma_j^2 + \epsilon}}\cdot\left(B\frac{\partial L}{\partial y_j^r} - \sum_s \frac{\partial L}{\partial y_j^s} - \hat{x}_j^r\sum_s \frac{\partial L}{\partial y_j^s}\hat{x}_j^s\right)
\end{aligned}$$

Question 3.2 (b)

Implemented on `custom_batchnorm.py` file on the attached code.

Question 3.2 (c)

Implemented on `custom_batchnorm.py` file on the attached code.

4 PyTorch CNN

Name	Kernel	Stride	Padding	Channels In/Out
conv1	3×3	1	1	3/64
maxpool1	3×3	2	1	64/64
conv2	3×3	1	1	64/128
maxpool2	3×3	2	1	128/128
conv3_a	3×3	1	1	128/256
conv3_b	3×3	1	1	256/256
maxpool3	3×3	2	1	256/256
conv4_a	3×3	1	1	256/512
conv4_b	3×3	1	1	512/512
maxpool4	3×3	2	1	512/512
conv5_a	3×3	1	1	512/512
conv5_b	3×3	1	1	512/512
maxpool5	3×3	2	1	512/512
avgpool	1×1	1	0	512/512
linear	–	-	-	512/10

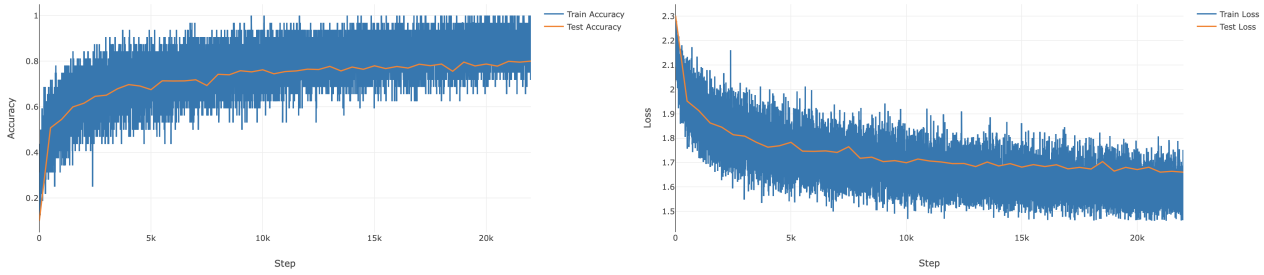
Figure 3: Specification of ConvNet architecture. All conv blocks consist of 2D-convolutional layer, followed by Batch Normalization layer and ReLU layer.

In 2012, with AlexNet it became clear that CNN were the identical Neural Networks to use for images. In this section we implement a small version of the popular VGGNet [3]. The architecture can be seen on Figure 3.

As we can clearly see, both on the performance table (5) and the performance plots (4), the CNN model outperforms the MLP by a margin of 21%! It is also impressive that the model did not overfit, not even until the 22kth step, while the MLP model over-fitted at step around 10k. The reason is because CNN uses local-sparce connectivity between its layer.

Accuracy		**Loss**	
Train	**Test**	**Train**	**Test**
1.00	0.80	1.46	1.66

Table 5: `covnet_pytorch`'s best performance on the test and train dataset.



(a) Accuracy performance of `covnet_pytorch` model

(b) Loss performance of `covnet_pytorch` model

Figure 4: Performance of the CNN implementation. The evaluation on the test set took place every 500 steps.

Bibliography

[1] A note on matrix differentiation , Pawe Kowal (link).

[2] How far can we go without convolution: improving fully-connected networks, Zhouhan Lin Roland Memisevic (link).

[3] Very Deep Convolutional Networks for Large-Scale Image Recognition, Karen Simonyan Andrew Zisserman (link).