# UvA Deep Learning Fall 2019 - practical 1

**David Knigge 11327782** `david.knigge@student.uva.nl`

## 1 Introduction

In this report a derivation is given for backpropagation for a simple MLP. The MLP is trained on the CIFAR dataset, which contains images of ten different classes. The MLP will be trained to classify an input image. Evaluation of this MLP is done on a hold-out test set. This MLP is subsequently implemented using Numpy and Pytorch respectively. A derivation is given for the batch normalisation algorithm, which is also implemented as a Pytorch module. Finally, a convolutional neural network based on the VGG-16 architecture is implemented, and its performance is discussed.

## 2 Question 1

### Question 1.1 a

1.

$$\begin{aligned}(\frac{\partial \mathcal{L}}{\partial x^{(N)}})_i &= \frac{\partial \mathcal{L}}{\partial x_i^{(N)}} \\ &= \frac{\partial}{\partial x_i^{(N)}} - \sum_i t_i \log(x_i^{(N)}) \\ &= -\frac{t_i}{x_i^{(N)}}\end{aligned}$$

2. If we denote the softmax function using $\sigma$.
   For $i = j$

$$\begin{aligned}(\frac{\partial x(N)}{\partial \tilde{x}(N)})_{i,j} &= \frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} \\ &= \frac{\partial}{\partial \tilde{x}_j} \frac{e^{\tilde{x}_i^{(N)}}}{\sum_k e^{\tilde{x}_k^{(N)}}} \\ &= \frac{\sum_k e^{\tilde{x}_k^{(N)}} e^{\tilde{x}_i^{(N)}} - e^{\tilde{x}_i^{(N)}} e^{\tilde{x}_j^{(N)}}}{(\sum_k e^{\tilde{x}_k^{(N)}})^2} \\ &= \frac{e^{\tilde{x}_i^{(N)}} (\sum_k e^{\tilde{x}_k^{(N)}} - e^{\tilde{x}_j^{(N)}})}{(\sum_k e^{\tilde{x}_k^{(N)}})^2} \\ &= \frac{e^{\tilde{x}_i^{(N)}}}{\sum_k e^{\tilde{x}_k^{(N)}}} \frac{(\sum_k e^{\tilde{x}_k^{(N)}} - e^{\tilde{x}_j^{(N)}})}{\sum_k e^{\tilde{x}_k^{(N)}}} \\ &= \sigma(\tilde{x}_i^{(N)})(1 - \sigma(\tilde{x}_i^{(N)})) \\ &= x_i^{(N)}(1 - x_i^{(N)})\end{aligned}$$

For $i \neq j$

$$\begin{aligned}\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} &= \frac{\partial}{\partial \tilde{x}_j} \frac{e^{\tilde{x}_i^{(N)}}}{\sum_k e^{\tilde{x}_k^{(N)}}} \\ &= \frac{0 - e^{\tilde{x}_i^{(N)}} e^{\tilde{x}_j^{(N)}}}{(\sum_k e^{\tilde{x}_k^{(N)}})^2} \\ &= \frac{-e^{\tilde{x}_i^{(N)}}}{\sum_k e^{\tilde{x}_k^{(N)}}} \frac{e^{\tilde{x}_j^{(N)}}}{\sum_k e^{\tilde{x}_k^{(N)}}} \\ &= -\sigma(\tilde{x}_i^{(N)})\sigma(\tilde{x}_j^{(N)}) \\ &= -x_i^{(N)} x_j^{(N)}\end{aligned}$$

So:

$$\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = \begin{cases} x_i^{(N)}(1 - x_i^{(N)}) \text{ if } i = j \\ -x_i^{(N)} x_j^{(N)} \text{ if } i \neq j \end{cases} \tag{1}$$

3.

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

4.

$$\begin{aligned}(\frac{\partial \tilde{x}^{(l)}}{\partial x^{(l-1)}})_{i,j} &= \frac{\partial \tilde{x}_i^{(l)}}{\partial x_j^{(l-1)}} \\ &= \frac{\partial}{\partial x_j^{(l-1)}}(W^{(l)} \tilde{x}_i^{(l)} + b^{(l)}) \\ &= W_{i,j}^{(l)}\end{aligned}$$

5.

$$\begin{aligned}(\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}})_{i,j,k} &= \frac{\partial \tilde{x}_i^{(l)}}{\partial W_{j,k}^{(l)}} \\ &= \frac{\partial}{\partial W_{j,k}^{(l)}}(W^{(l)} \tilde{x}_i^{(l)} + b^{(l)}) \\ &= \begin{cases} x_k \text{ if } i = j \\ 0 \text{ if } i \neq j \end{cases}\end{aligned}$$

6.

$$\begin{aligned}(\frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}})_{i,j} &= \frac{\partial \tilde{x}_i^{(l)}}{\partial b_j^{(l)}} \\ &= \frac{\partial}{\partial b_j^{(l)}}(W^{(l)} \tilde{x}_i^{(l)} + b_i^{(l)}) \\ &= \begin{cases} 1 \text{ if } i = j \\ 0 \text{ if } i \neq j \end{cases}\end{aligned}$$

**Question 1.1 b**

1.

$$\frac{\partial L}{\partial x_i^{(N)}}\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = \begin{cases} \sum_i^D \frac{\partial L}{\partial x_i^{(N)}} x_i^{(N)}(1-x_j^{(N)}) \text{ if } i=j \\ \sum_i^D \frac{\partial L}{\partial x_i^{(N)}} - x_i^{(N)}x_j^{(N)} \text{ if } i\neq j \end{cases}$$
$$= \begin{cases} \frac{\partial L}{\partial x_i^{(N)}} x_i^{(N)}(1-x_j^{(N)}) \text{ if } i=j \\ \frac{\partial L}{\partial x_i^{(N)}} - x_i^{(N)}x_j^{(N)} \text{ if } i\neq j \end{cases}$$

So in vector form:

$$\frac{\partial L}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} \cdot (\texttt{diag}(x^{(N)}) - x^{(N)} \otimes x^{(N)^T})$$

2.

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

In matrix form:

$$\frac{\partial L}{\partial \tilde{x}_1^{(l<N)}} = \begin{bmatrix} \begin{cases} \frac{\partial L}{\partial x_1^{(l<N)}} \text{ if } \tilde{x}_1^{(l<N)} > 0 \\ a\frac{\partial L}{\partial x_1^{(l<N)}} \text{ if } \tilde{x}_1^{(l<N)} < 0 \end{cases} & 0 & 0 & ... & 0 \\ 0 & \begin{cases} \frac{\partial L}{\partial x_2^{(l<N)}} \text{ if } \tilde{x}_2^{(l<N)} > 0 \\ a\frac{\partial L}{\partial x_2^{(l<N)}} \text{ if } \tilde{x}_2^{(l<N)} < 0 \end{cases} & 0 & ... & 0 \\ 0 & 0 & \begin{cases} \frac{\partial L}{\partial x_3^{(l<N)}} \text{ if } \tilde{x}_3^{(l<N)} > 0 \\ a\frac{\partial L}{\partial x_3^{(l<N)}} \text{ if } \tilde{x}_3^{(l<N)} < 0 \end{cases} & ... & 0 \\ etc & & & & \end{bmatrix}$$

3.

$$\frac{\partial L}{\partial x_j^{(l<N)}} = \sum_i^D \frac{\partial L}{\partial \tilde{x}_i^{(l+1)}}\frac{\partial \tilde{x}_i^{(l+1)}}{\partial x_j^{(l)}}$$
$$= \sum_i^D \frac{\partial L}{\partial \tilde{x}_i^{(l+1)}} W_{i,j}^{(l+1)}$$

In vector form:

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

4.

$$\frac{\partial L}{\partial W_{j,k}^{(l)}} = \sum_i^D \frac{\partial L}{\partial \tilde{x}_i^{(l)}}\frac{\partial \tilde{x}_i^{(l)}}{\partial W_{j,k}^{(l)}}$$
$$= \begin{cases} \frac{\partial L}{\partial \tilde{x}_i^{(l)}} x_k^{(l-1)} \text{ if } i=j \\ 0 \text{ otherwise} \end{cases}$$

In vector form:

$$\frac{\partial L}{\partial W_{j,k}^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} x^{(l-1)^T}$$

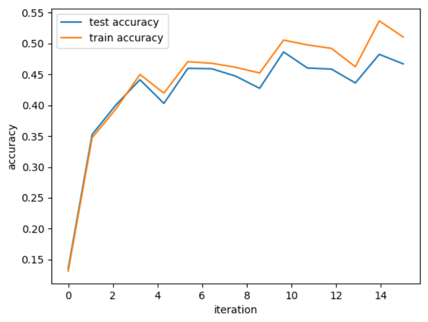

Figure 1: Accuracy on training and test set of MLP numpy implementation. As the number of iterations indcreases the accuracy for test and training sets start to diverge, which is an indication that our model is starting to overfit on the training set.

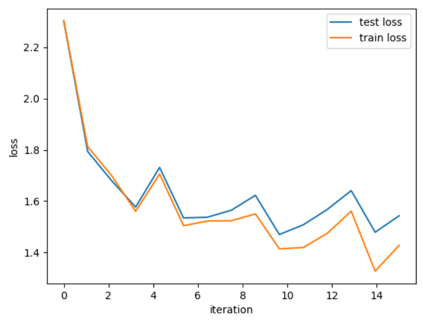

Figure 2: Loss on training and test set of MLP numpy implementation. One iteration is 100 batches. As the number of iterations increases the loss for test and training set start to diverge, which is an indication that our model is beginning to overfit on the training set.

5.

$$\frac{\partial L}{\partial b_j^{(l)}} = \sum_i^D \frac{\partial L}{\partial \tilde{x_i}^{(l)}} \frac{\partial \tilde{x_i}^{(l)}}{\partial b_j^{(l)}}$$
$$= \begin{cases} \frac{\partial L}{\partial \tilde{x_i}^{(l)}} \text{ if } i = j \\ 0 \text{ otherwise} \end{cases}$$

In matrix form:

$$\begin{bmatrix} \frac{\partial L}{\partial \tilde{x_1}^{(l)}} & 0 & \dots & 0 \\ 0 & \frac{\partial L}{\partial \tilde{x_2}^{(l)}} & \dots & 0 \\ 0 & 0 & \frac{\partial L}{\partial \tilde{x_3}^{(l)}} & \dots \\ etc & & & \end{bmatrix}$$

Which results in a vector form:

$$\frac{\partial L}{\partial b_j^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}}$$

### Question 1.1 c

The previously derived formulas are for a batch size of 1. If we now use a larger batch size, the dimensionality of the backpropagation gradients will grow by one dimension, the batch size. Before the weight and bias updates are applied with the derived gradients, the batch dimension will thus need to be averaged over to determine weight and bias updates over the entire batch.

### Question 1.2

In this section performance of the implementation of the previously derived backpropagation is highlighted. The MLP was trained with default settings, training for 1500 batches, with a batch size of 200 samples, and a learning rate of 0.002. The network contained one hidden layer of 100 units. Accuracy and loss for this model is evaluated on the training and test set every 100 batches. See figure 1 for the accuracy and figure 2 for the loss. We can see that the accuracy on the test set hovers around 46, with an average loss on the test set of around 1.6.

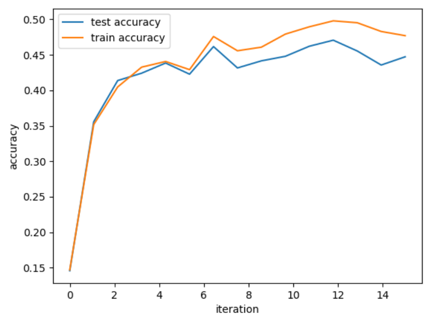

Figure 3: Accuracy on training and test set of MLP pytorch implementation.

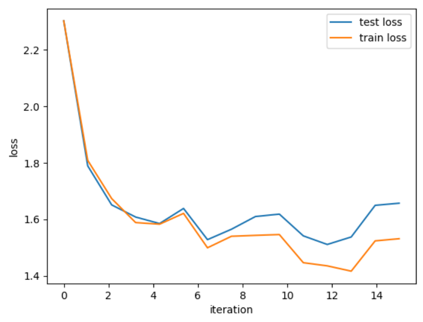

Figure 4: Loss on training and test set of MLP pytorch implementation. One iteration is 100 batches.

## Question 2

In this section the previously implemented MLP was reimplemented using pytorch. Performance on the default settings is given first, after which a number of parameters and their influence on accuracy and loss highlighted.

### Default parameters

The performance of this implementation on the default parameters is given. In figure 3 the accuracy of the model on the training and test set is given, in figure 4 the loss on the training and test set is given. We can see that the loss and accuracy follow a pattern very similar to that of the previously mentioned numpy implementation.

### Obtaining higher accuracy

In order to obtain a higher accuracy on the test set, I experimented with a number of parameters of the model. In the following sections each of the parameters and its influence on the accuracy is explained.

### Optimiser / Learning rate

I started off using normal Stochastic Gradient Descent (SGD) as optimisation algorithm, as this is what was used in the numpy implementation. SGD is a somewhat simple algorithm as it keeps the learning rate constant over the entire training period, regardless of the magnitude of the gradient. First I tried experimenting with the learning rate to see what difference this would make. However, increasing the learning rate lead to the gradients exploding quite quickly, whereas decreasing the learning rate seemed to make it so that the model barely learned anything and the test accuracy barely increased. I expected an adaptive learning rate to work better, as this would allow the model to converge faster and find a more accurate local minimum. As pytorch allows us to easily swap out SGD for a different optimiser, I attempted the Adam optimiser. Adam combines AdaGrad (adaptive gradient descent, an adaptive learning rate) with RMSProp (Root mean square error propagation, learning rates that are adapted based on the average of recent magnitudes of gradient). Using Adam as optimiser but keeping all other parameters at their defaults, the accuracy and loss became a lot noisier. Most notably, the accuracy dips at numerous iterations, and the loss greatly increases seemingly randomly. Results of changing only the optimiser on the accuracy and loss are shown in figures 5 and 6 respectively.

### Architecture

Secondly, I experimented with the architecture of the network to obtain better results. I experimented with a number of different setups, adding one hidden layer at a time and running it for a high number

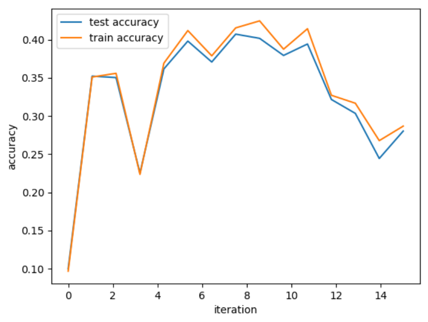

Figure 5: Accuracy on training and test set of MLP pytorch implementation using the Adam optimiser. One iteration is 100 batches.

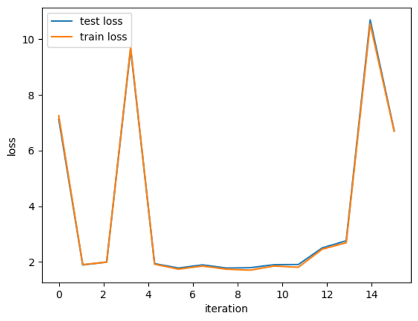

Figure 6: Loss on training and test set of MLP pytorch implementation using the Adam optimiser. One iteration is 100 batches.

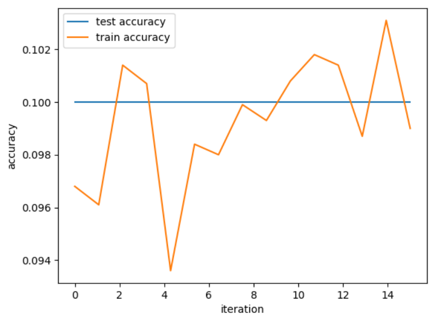

Figure 7: Accuracy on training and test set of MLP pytorch implementation using five hidden layers. One iteration is 100 batches.

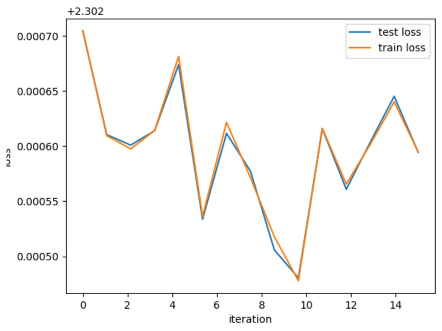

Figure 8: Loss on training and test set of MLP pytorch implementation using five hidden layers. One iteration is 100 batches.

of iterations to allow it to learn the different classes. I ultimately decided on a setup of multiple hidden layers, as this seemed to obtain the best test accuracy. Specifically, I added five hidden layers to the network, decreasing in dimensionality. I expected this to obtain better results, as adding more layers to the network increases its complexity and thus its expressivity. Keeping the other settings at their default values, I used the following number of nodes in the hidden layers: 900, 500, 350, 150, 50. The accuracy obtained with this architecture is shown in figures 7, 8. With the default number of training steps and batch size, this architecture did not manage to learn much. With the default learning rate and number of steps the weight updates probably aren't large enough to make a real difference to the model. Accuracy is as good as random, and the loss barely decreases.

### Maximum iterations

I furthermore experimented with the maximum number of iterations I let the model train for and keeping other settings at their default. This did not increase the accuracy, which was to be expected, as the test accuracy already seems to flatten out in the original number of iterations quite quickly (this can be seen in figure 3. Accuracy and loss on a larger number of maximum interations is shown in figures 9 and 10 respectively.

### Combining

Lastly I combined the aforementioned experiments. The architecture was extended to increase the expressivity of the model, the optimisation algorithm was changed to allow faster convergence on

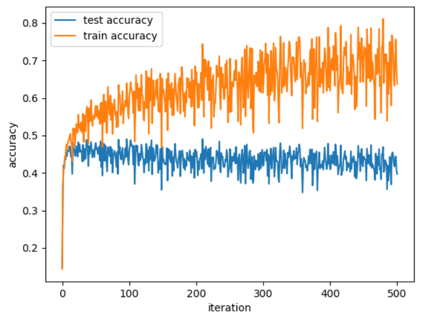

Figure 9: Accuracy on training and test set of MLP pytorch implementation using 50000 training batches. One iteration is 100 batches.

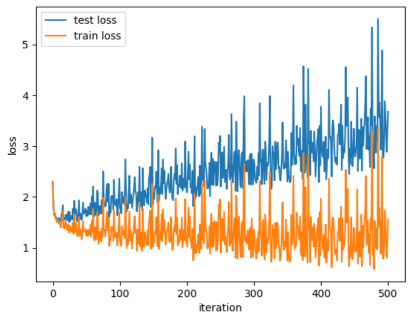

Figure 10: Loss on training and test set of MLP pytorch implementation using 50000 training batches. One iteration is 100 batches.

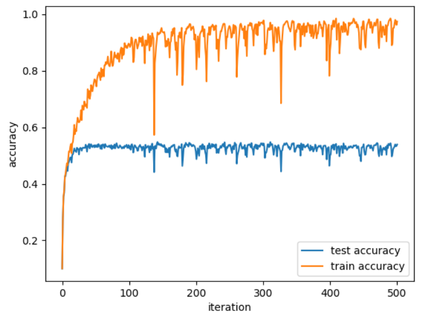

Figure 11: Accuracy of improved model on test and training set.

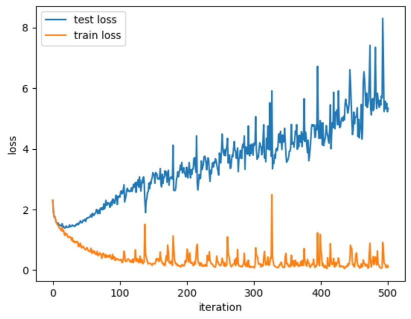

Figure 12: Loss of improved model on test and training set.

the larger gradient search space, and the number of maximum iterations was increased to allow the model time to converge. The following settings where thus used:

- Hidden Layers: 900, 500, 350, 150, 50
- Batch Size: 200
- Learning Rate: 0.002
- Optimiser: Adam
- Max Steps: 50000
- Negative Slope: 0.02

This ultimately somewhat increased the accuracy of the model, accuracy and loss of which can be seen in figures 11 and 12 respectively. Accuracy of this model on the test set peaked at 0.54! As can be seen from the figure, after a relatively small number of iterations the accuracy on training and test set start to diverge heavily. At 50000 iterations, training accuracy reaches over 0.95, whereas test accuracy remains at around 0.53. Although this indicates heavy overfitting on the training dataset, the accuracy obtained on the test set is still satisfactory.

## Question 3

### Question 3.1

See `custom_batchnorm.py`.

**Question 3.2**

a

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

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

For the last derivative, w.r.t. the input of the batch norm layer, we apply the chain rule to obtain:

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

Which becomes:

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

We can easily determine:

$$\frac{\partial y_i^s}{\partial \hat{x}_j^r} = \gamma_j \text{if } i = j \text{ and } s = t, \text{ 0 otherwise}$$

So:

$$(\frac{\partial L}{\partial x})_j^r = \gamma_j \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial \hat{x}_i^s}{\partial x_j^r}$$

Now we determine:

$$\begin{aligned}\frac{\partial \hat{x}_i^s}{\partial x_j^r} &= \frac{\partial}{\partial x_j^r} \frac{x_i^s - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} \\ &= \frac{\partial}{\partial x_j^r} (x_i^s - \mu_i)(\sigma_i^2 + \epsilon)^{-\frac{1}{2}} \\ &= (\delta_{i,j}\delta_{r,s} - \frac{\delta_{i,j}}{B})(\sigma_i^2 + \epsilon)^{-\frac{1}{2}} - \frac{1}{2}(x_i^s - \mu_i)(\sigma_i^2 + \epsilon)^{-\frac{3}{2}} \frac{\partial \sigma_i^2}{\partial x_j^r}\end{aligned}$$

We still need to determine:

$$\begin{aligned}
\frac{\partial \sigma_i^2}{\partial x_j^r} &= \frac{\partial}{\partial x_j^r}(\frac{1}{B}\sum_{s=1}^{B}(x_i^s - \mu_i)^2) \\
&= \frac{2}{B}\sum_{s=1}^{B}(x_i^s - \mu_i)\frac{\partial(x_i^s - \mu_i)}{\partial x_j^r} \\
&= \frac{2}{B}\sum_{s=1}^{B}(x_i^s - \mu_i)(\delta_{r,s}\delta_{i,j} - \frac{\delta_{i,j}}{B}) \\
&= -\frac{2}{B}\sum_{s=1}^{B}\frac{\delta_{i,j}}{B}(x_i^s - \mu_i) + \frac{2}{B}\sum_{s=1}^{B}(\delta_{r,s}\delta_{i,j})(x_i^s - \mu_i) \\
&= -\frac{2}{B}\sum_{s=1}^{B}\frac{\delta_{i,j}}{B}(x_i^s - \mu_i) + \frac{2}{B}\delta_{i,j}(x_i^r - \mu_i) \\
&= -\frac{2\delta_{i,j}}{B}(\sum_{s=1}^{B}\frac{1}{B}x_i^s) - \mu_i) + \frac{2}{B}\delta_{i,j}(x_i^r - \mu_i) \\
&= \frac{2}{B}\delta_{i,j}(x_i^r - \mu_i)
\end{aligned}$$

Filling this in we get:

$$\begin{aligned}
\frac{\partial \hat{x}_i^s}{\partial x_j^r} &= (\delta_{i,j}\delta_{r,s} - \frac{\delta_{i,j}}{B})(\sigma_i^2 + \epsilon)^{-\frac{1}{2}} - \frac{1}{2}(x_i^s - \mu_i)(\sigma_i^2 + \epsilon)^{-\frac{3}{2}}\frac{2}{B}\delta_{i,j}(x_i^r - \mu_i) \\
&= (\delta_{i,j}\delta_{r,s} - \frac{\delta_{i,j}}{B})(\sigma_i^2 + \epsilon)^{-\frac{1}{2}} - \frac{\delta_{i,j}}{B}(x_i^s - \mu_i)(\sigma_i^2 + \epsilon)^{-\frac{3}{2}}(x_i^r - \mu_i) \\
&= \frac{\delta_{i,j}\delta_{r,s} - \frac{\delta_{i,j}}{B}}{\sqrt{\sigma_i^2 + \epsilon}} - \frac{\delta_{i,j}}{B}\frac{1}{\sqrt{\sigma_i^2 + \epsilon}}\hat{x}_i^s\hat{x}_i^r \\
&= \frac{\delta_{i,j}\delta_{r,s} - \frac{\delta_{i,j}}{B} - \frac{\delta_{i,j}}{B}\hat{x}_i^s\hat{x}_i^r}{\sqrt{\sigma_i^2 + \epsilon}} \\
&= \frac{B\delta_{i,j}\delta_{r,s} - \delta_{i,j} - \delta_{i,j}\hat{x}_i^s\hat{x}_i^r}{B\sqrt{\sigma_i^2 + \epsilon}}
\end{aligned}$$

Filling this in we get:

$$\begin{aligned}
(\frac{\partial L}{\partial x})_j^r &= \gamma_j \sum_s \sum_i \frac{\partial L}{\partial y_i^s}(\frac{B\delta_{i,j}\delta_{r,s} - \delta_{i,j} - \delta_{i,j}\hat{x}_i^s\hat{x}_i^r}{B\sqrt{\sigma_i^2 + \epsilon}}) \\
&= \frac{\gamma_j}{B\sqrt{\sigma_i^2 + \epsilon}}(B\frac{\partial L}{\partial y_i^s} - \sum_s \frac{\partial L}{\partial y_i^s} - \sum_s \frac{\partial L}{\partial y_i^s}\hat{x}_j^s\hat{x}_j^r)
\end{aligned}$$

## Question 4

In this section performance of a convolutional neural network on the CIFAR dataset is briefly highlighted. Using the default parameters, the plots of the accuracy and loss on training and test sets are shown in figures 13 and 14 respectively. As we can see, the accuracy of the convolutional neural network on the test set is considerably higher than that of the simple MLP. This is attributable to the fact that using convolution kernels, the model is able to better learn feature representations to optimally distinguish the different classes based on visual information. We end up with a test accuracy of around 0.76, and a loss of 0.75 on the test set.

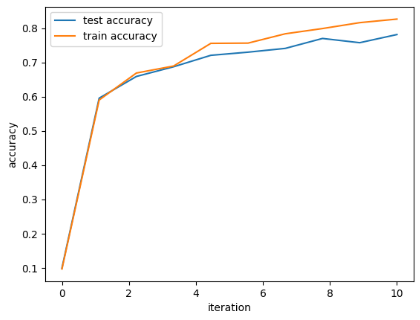

Figure 13: Accuracy of pytorch CNN implementation on test and training set. One iteration is 500 batches.

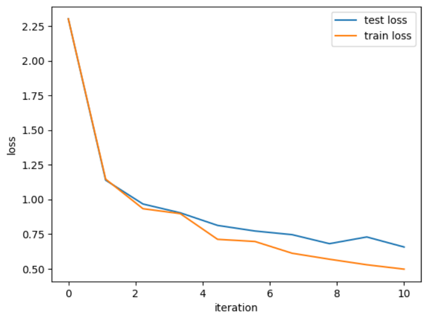

Figure 14: Loss of pytorch CNN implementation on test and training set. One iteration is 500 batches.

## References

[1] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.