Assignment 1. MLPs, CNNs and Backpropagation

Martine Toering
11302925
Deep Learning Course Assignment 1
University of Amsterdam
November 15, 2019
martine.toering@student.uva.nl

1 MLP backprop and NumPy implementation

1.1 Analytical derivation of gradients

Question 1.1 a) Compute the gradients of each module.

- $\frac{\partial L}{\partial x_i^{(N)}} = \frac{\partial}{\partial x_i^{(N)}} \left(- \sum_i \ t_i \ log \ x_i^{(N)} \right)$
  $= \frac{\partial}{\partial x_i^{(N)}} \left(- \left(\ t_1 \ log \ x_1^{(N)} + t_2 \ log \ x_2^{(N)} + ... + t_i \ log \ x_i^{(N)} + ... \ \right) \right)$
  $= -t_i \ \frac{1}{x_i^{(N)}} = -\frac{t_i}{x_i^{(N)}}$

- $\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}}$

  $\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = \frac{\partial}{\partial \tilde{x}_j^{(N)}} \left(softmax(\tilde{x}_i^{(N)}) \right)$

  If $i = j$

  $= \frac{\partial}{\partial \tilde{x}_j^{(N)}} \left(\frac{exp(\tilde{x}_i^{(N)})}{\sum_{k=1}^{d_N} exp(\tilde{x}_k^{(N)})} \right)$

  $= \frac{\partial}{\partial \tilde{x}_j^{(N)}} \left(\frac{\frac{\partial}{\partial \tilde{x}_j^{(N)}} \left(exp(\tilde{x}_i^{(N)}) \right) \left(\sum_{k=1}^{d_N} exp(\tilde{x}_k^{(N)}) \right) \ - \ \left(exp(\tilde{x}_i^{(N)}) \right) \frac{\partial}{\partial \tilde{x}_j^{(N)}} \left(\sum_{k=1}^{d_N} exp(\tilde{x}_k^{(N)}) \right)}{\left(\sum_{k=1}^{d_N} exp(\tilde{x}_k^{(N)}) \right)^2} \right)$

$$= \frac{\partial}{\partial \tilde{x}_j^{(N)}} \left(\frac{exp(\tilde{x}_i^{(N)}) \left(\sum_{k=1}^{d_N} exp(\tilde{x}_k^{(N)}) \right) \; - \; exp(\tilde{x}_i^{(N)}) exp(\tilde{x}_j^{(N)})}{\left(\sum_{k=1}^{d_N} exp(\tilde{x}_k^{(N)}) \right)^2} \right)$$

$$= \frac{\partial}{\partial \tilde{x}_j^{(N)}} \left(\frac{exp(\tilde{x}_i^{(N)})}{\sum_{k=1}^{d_N} exp(\tilde{x}_k^{(N)})} \; \cdot \; \left(\frac{\sum_{k=1}^{d_N} exp(\tilde{x}_k^{(N)}) - exp(\tilde{x}_j^{(N)})}{\sum_{k=1}^{d_N} exp(\tilde{x}_k^{(N)})} \right) \right)$$

$$= softmax(\tilde{x}_i^{(N)}) \big(1 - softmax(\tilde{x}_j^{(N)}) \big)$$

$$\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = \frac{\partial S(\tilde{x}_i^{(N)})}{\partial \tilde{x}_j^{(N)}} = S(\tilde{x}_i^{(N)}) \big(1 - S(\tilde{x}_j^{(N)}) \big)$$

If $i \neq j$

$$= \frac{\partial}{\partial \tilde{x}_j^{(N)}} \left(\frac{exp(\tilde{x}_i^{(N)})}{\sum_{k=1}^{d_N} exp(\tilde{x}_k^{(N)})} \right)$$

$$= \frac{\partial}{\partial \tilde{x}_j^{(N)}} \left(\frac{- exp(\tilde{x}_i^{(N)}) exp(\tilde{x}_j^{(N)})}{\left(\sum_{k=1}^{d_N} exp(\tilde{x}_k^{(N)}) \right)^2} \right)$$

$$= \frac{\partial}{\partial \tilde{x}_j^{(N)}} \left(- \frac{exp(\tilde{x}_i^{(N)})}{\sum_{k=1}^{d_N} exp(\tilde{x}_k^{(N)})} \; \cdot \; \left(\frac{exp(\tilde{x}_j^{(N)})}{\sum_{k=1}^{d_N} exp(\tilde{x}_k^{(N)})} \right) \right)$$

$$= - \; softmax(\tilde{x}_i^{(N)}) \; softmax(\tilde{x}_j^{(N)})$$

$$\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = \frac{\partial S(\tilde{x}_i^{(N)})}{\partial \tilde{x}_j^{(N)}} = - \; S(\tilde{x}_i^{(N)}) \; S(\tilde{x}_j^{(N)})$$

- $$\frac{\partial x_i^{(l<N)}}{\partial \tilde{x}_j^{(l<N)}} = \frac{\partial}{\partial \tilde{x}_j^{(l<N)}} \left(\; LeakyRELU(\tilde{x}_i^{(l<N)}) \; \right)$$
  $$= \frac{\partial}{\partial \tilde{x}_j^{(l<N)}} \left(\; max\big(0, \tilde{x}_i^{(l<N)}\big) + a \cdot min\big(0, \tilde{x}_i^{(l<N)}\big) \; \right)$$

If i = j

$$= \begin{cases} 1 & \text{if } \tilde{x}_j^{(l<N)} \geq 0 \\ a & \text{if } \tilde{x}_j^{(l<N)} < 0 \end{cases}$$

$$\frac{\partial x_i^{(l<N)}}{\partial \tilde{x}_j^{(l<N)}} = \frac{\partial \left(\, LeakyRELU(\tilde{x}_i^{(l<N)}) \, \right)}{\partial \tilde{x}_j^{(l<N)}} = \begin{cases} 1 & \text{if } \tilde{x}_j^{(l<N)} \geq 0 \\ a & \text{if } \tilde{x}_j^{(l<N)} < 0 \end{cases}$$

If $i \neq j$ 0

- $\frac{\partial \tilde{x}_i^{(l)}}{\partial x_j^{(l-1)}} = \frac{\partial}{\partial x_j^{(l-1)}} \left(\sum_m \, W_{mi}^{(l)} \, x_i^{(l-1)} + b_m^{(l)} \right) = \sum_m \, W_{mj}^{(l)}$

- $\frac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^{(l)}} = \frac{\partial}{\partial W_{jk}^{(l)}} \left(\sum_m \, W_{mi}^{(l)} \, x_i^{(l-1)} + b_m^{(l)} \right) = x_k^{(l-1)}$

- $\frac{\partial \tilde{x}_i^{(l)}}{\partial b_j^{(l)}} = \frac{\partial}{\partial b_j^{(l)}} \left(\sum_m \, W_{mi}^{(l)} \, x_i^{(l-1)} + b_m^{(l)} \right) = 1$

Question 1.1 b) Calculate the gradients by backpropagation.

- $\frac{\partial L}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} \frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}}$

  $\frac{\partial L}{\partial \tilde{x_i}^{(N)}} = \sum_j \, \left(\frac{\partial L}{\partial x_j^{(N)}} \right) \left(\frac{\partial x_j^{(N)}}{\partial \tilde{x_i}^{(N)}} \right)$

  $= \sum_j \, \left(\frac{\partial L}{\partial x_j^{(N)}} \right) \left(S(\tilde{x}_j^{(N)}) (1 - S(\tilde{x}_i^{(N)})) \text{ if i=j else } - S(\tilde{x}_j^{(N)}) \, S(\tilde{x}_i^{(N)}) \right)$

$$\nabla_{\tilde{x}^{(N)}}(L) = \left(\frac{\partial \mathbf{x}^{(N)}}{\partial \tilde{\mathbf{x}}^{(N)}} \right)^T \cdot \nabla_{x^{(N)}}(L)$$

$$\nabla_{\tilde{x}^{(N)}}(L) = \begin{bmatrix} \frac{\partial x_1^{(N)}}{\partial \tilde{x_1}^{(N)}} & \frac{\partial x_1^{(N)}}{\partial \tilde{x_2}^{(N)}} & \cdots \\ \frac{\partial x_2^{(N)}}{\partial \tilde{x_1}^{(N)}} & \frac{\partial x_2^{(N)}}{\partial \tilde{x_2}^{(N)}} & \cdots \\ \cdots & \cdots & \cdots \end{bmatrix}^T \cdot \nabla_{x^{(N)}}(L)$$

$$\nabla_{\tilde{x}^{(N)}}(L) = \begin{bmatrix} S(\tilde{x}_1^{(N)})\big(1 - S(\tilde{x}_1^{(N)})\big) & -S(\tilde{x}_1^{(N)})\, S(\tilde{x}_2^{(N)}) & ... \\ -S(\tilde{x}_2^{(N)})\, S(\tilde{x}_1^{(N)})\big) & S(\tilde{x}_2^{(N)})\big(1 - S(\tilde{x}_2^{(N)})\big) & ... \\ ... & ... & ... \end{bmatrix}^T \cdot \nabla_{x^{(N)}}(L)$$

- $\dfrac{\partial L}{\partial \tilde{x}^{(l<N)}} = \dfrac{\partial L}{\partial x^{(l)}} \dfrac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}}$

  $\dfrac{\partial L}{\partial \tilde{x_i}^{(l<N)}} = \sum_j \left(\dfrac{\partial L}{\partial x_j^{(l)}}\right)\left(\dfrac{\partial x_j^{(l)}}{\partial \tilde{x_i}^{(l)}}\right)$

  $\dfrac{\partial L}{\partial \tilde{x_i}^{(l<N)}} = \sum_j \left(\dfrac{\partial L}{\partial x_j^{(l)}}\right)\Big(\Big(1 \text{ if } \tilde{x}_j^{(l<N)} \geq 0 \text{ else } a\Big) \text{ if i=j else } 0\Big)$

  $\nabla_{\tilde{x}^{(l<N)}}(L) = \Big(\dfrac{\partial \mathbf{x}^{(l)}}{\partial \tilde{\mathbf{x}}^{(l)}}\Big)^T \cdot \nabla_{x^{(l)}}(L)$

  $$\nabla_{\tilde{x}^{(l<N)}}(L) = \begin{bmatrix} \dfrac{\partial x_1^{(l)}}{\partial \tilde{x_1}^{(l)}} & \dfrac{\partial x_1^{(l)}}{\partial \tilde{x_2}^{(l)}} & ... \\ \dfrac{\partial x_2^{(l)}}{\partial \tilde{x_1}^{(l)}} & \dfrac{\partial x_2^{(l)}}{\partial \tilde{x_2}^{(l)}} & ... \\ ... & ... & ... \end{bmatrix}^T \cdot \nabla_{x^{(l)}}(L)$$

  $$\nabla_{\tilde{x}^{(l<N)}}(L) = \begin{bmatrix} \Big(1 \text{ if } \tilde{x}_1^{(l<N)} \geq 0 \text{ else } a\Big) & 0 & ... \\ 0 & \Big(1 \text{ if } \tilde{x}_2^{(l<N)} \geq 0 \text{ else } a\Big) & ... \\ ... & ... & ... \end{bmatrix}^T \cdot \nabla_{x^{(l)}}(L)$$

  $$\nabla_{\tilde{x}^{(l<N)}}(L) = \begin{bmatrix} \Big(1 \text{ if } \tilde{x}_1^{(l<N)} \geq 0 \text{ else } a\Big) \\ \Big(1 \text{ if } \tilde{x}_2^{(l<N)} \geq 0 \text{ else } a\Big) \\ ... \end{bmatrix}^T \cdot \nabla_{x^{(l)}}(L)$$

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

  $\dfrac{\partial L}{\partial x_i^{(l<N)}} = \sum_j \left(\dfrac{\partial L}{\partial \tilde{x_j}^{(l+1)}}\right)\left(\dfrac{\partial \tilde{x_j}^{(l+1)}}{\partial x_i^{(l)}}\right)$

$$= \sum_j \left(\frac{\partial L}{\partial \tilde{x_j}^{(l+1)}} \right) \left(\sum_m W_{mj}^{(l)} \right)$$

$$\nabla_{x^{(l<N)}}(L) = (\mathbf{W}^{(l)})^T \cdot \nabla_{\tilde{x}^{(l+1)}}(L)$$

- $$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}}$$
  $$\frac{\partial L}{\partial W_{ij}^{(l)}} = \sum_j \left(\frac{\partial L}{\partial \tilde{x_j}^{(l)}} \right) \left(\frac{\partial \tilde{x_j}^{(l)}}{\partial W_{ij}^{(l)}} \right)$$
  $$= \sum_j \left(\frac{\partial L}{\partial \tilde{x_j}^{(l)}} \right) \left(x_j^{(l-1)} \right)$$
  $$\nabla_{W^{(l)}}(L) = (\mathbf{x}^{(l-1)})^T \cdot \nabla_{\tilde{x}^{(l)}}(L)$$

- $$\frac{\partial L}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}}$$
  $$\frac{\partial L}{\partial b_i^{(l)}} = \sum_j \left(\frac{\partial L}{\partial \tilde{x_j}^{(l)}} \right) \left(\frac{\partial \tilde{x_j}^{(l)}}{\partial b_i^{(l)}} \right)$$
  $$\nabla_{b^{(l)}}(L) = \nabla_{\tilde{x}^{(l)}}(L)$$

Question 1.1 c) Argue how the backpropagation equations derived above change if a batchsize B $\neq$ 1 is used.

When using batches of input samples, the input **x** becomes a matrix of batch size $\times$ number of input features instead of a vector. The partial gradients therefore also change. The softmax derivative $\left(\frac{\partial \mathbf{x}^{(N)}}{\partial \tilde{\mathbf{x}}^{(N)}}\right)^T$ becomes a Jacobian for each sample in the batch size; thus can for instance be implemented as a tensor of batch size $\times$ number of input features $\times$ number of input features. However, this Jacobian multiplied by the loss with respect to $\tilde{x^N}$ (pre-softmax activations), the loss gradient with respect to the softmax output $\nabla_{x^{(N)}}(L)$, , ends up being two dimensional again with dimensions batch size $\times$ output features. This makes sense, as all gradients with respect to the loss need to be the same size as the functions. The LeakyReLU derivative vector $\left(\frac{\partial \mathbf{x}^{(l)}}{\partial \tilde{\mathbf{x}}^{(l)}}\right)^T$ which was a diagonal Jacobian before (and thus a vector) becomes a full matrix Jacobian. The weight matrix $\boldsymbol{W}$ and the bias $\boldsymbol{b}$ are *independent* of the batch size. The gradient of the bias becomes the sum of the loss with respect to the input over the batch size dimension.

1.2 1.2 NumPy implementation

Question 1.2) Implement a multi-layer perceptron using purely NumPy routines. Provide accuracy and loss curves in your report for the default values of parameters.

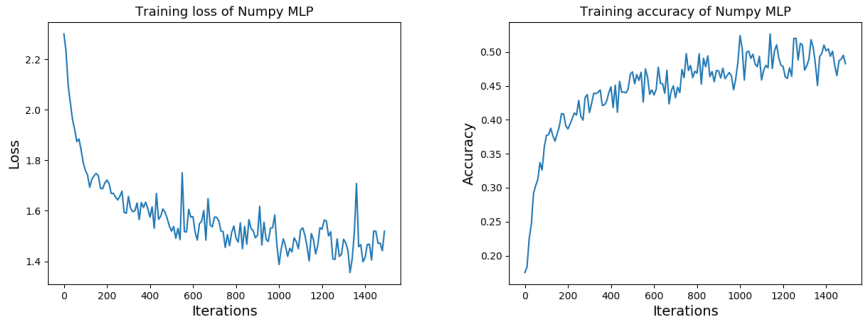

Figure 1: The NumPy MLP training loss and accuracy curves for default parameters

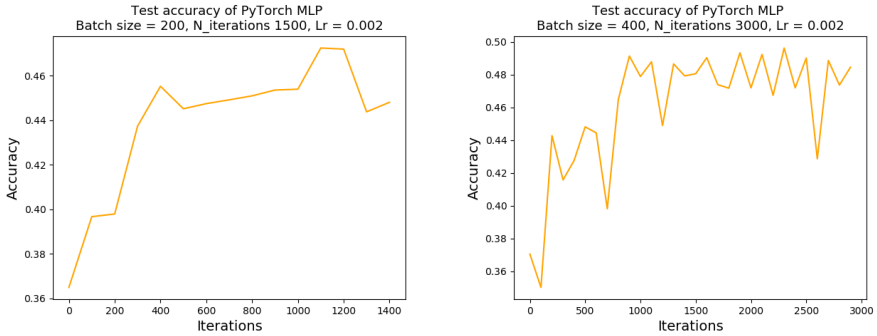

Figure 2: Test accuracy evaluated each *eval_frequency = 100* iterations plotted against the number of iterations of PyTorch MLP with 1-hidden layer with 100 neurons. On the left, default parameters were used and on the right both the batch size as well as the number of iterations were doubled.

Testing the NumPy MLP implementation with one hidden layer of 100 units and the default parameters after training (on iteration 1500) results in an accuracy of **45.66**%. Figure 1 shows the loss curve on the left and the training accuracy on the right. The curves are smoothed by taking the average value over each 10 iterations in order to obtain a better visualization of the curves as the training process is not very stable. The loss curve shows that the loss decreases rapidly at the start and seems to continue to decrease at the end of the training process. The parameter *eval_frequency* has been interpreted as the number of training iterations after which the performance on the test set needs to be evaluated. Thus, the model has been tested *max_steps / eval_frequency* times, for the default parameters this was $1500/100 = 15$ times. The best test accuracy obtained was 48.43% (on iteration 1400).

2 Pytorch MLP

Question 2 Implement the MLP in mlp_pytorch.py file.

Using the same parameters as in the NumPy implementation of Question 1.2, the accuracy obtained by evaluating the test set after training resulted in an accuracy of **44.81%**. Using the default value of *eval_frequency = 100*, the best accuracy obtained is 47.25% (on iteration 1300). Figure 3 shows the loss curve for the training and the training accuracy plotted against the number of steps or iterations taken. The training accuracy and loss were calculated each iteration but averaged over every 10 iterations. The test accuracy calculated each *eval_frequency=100* iterations is shown in figure 2.

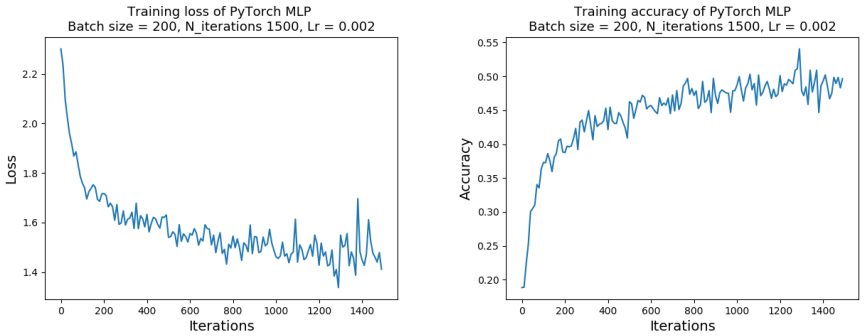

Figure 3: The loss and accuracy curves of PyTorch MLP with 1-hidden layer with 100 neurons with default parameters

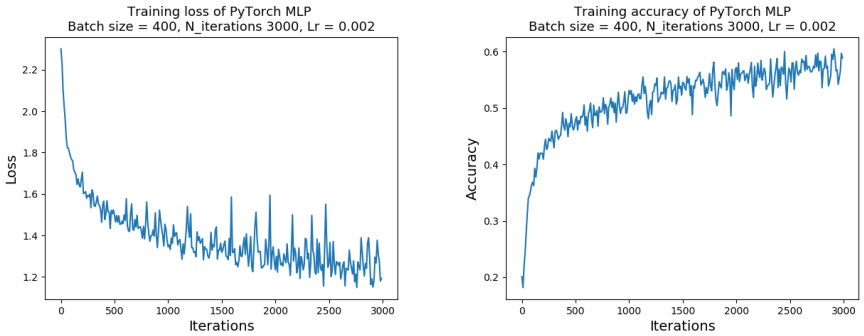

Figure 4: The loss and accuracy curves of PyTorch MLP with 1-hidden layer with 100 neurons for batch size 400, number of iterations equal to 3000

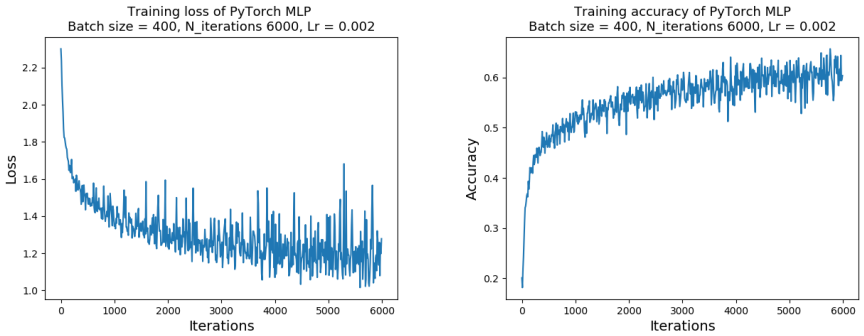

Figure 5: The loss and accuracy curves of PyTorch MLP with 1-hidden layer with 100 neurons for batch size 400, number of iterations equal to 6000

Pytorch MLP Study

The goal is to improve the accuracy obtained on the test set, along with getting good accuracy on the training set. Good performance on both the training set and the test set indicates that our model generalizes well.

First we tried to improve the model without adding layers to get a feeling whether this simple one hidden layer network could be improved. We ran the model for a larger number of iterations and a larger batch size from here on; a larger number of iterations was chosen because looking at the training loss, the loss did not completely reached a plateau yet. The loss curve indicates that there might be a bit of improvement possible when running the model for a larger amount of iterations. The number of iterations were set to double the size (3000 instead of 1500) and the batch size was set to 400. Often it is best practice to set the batch size as large as possible. For this model, this should probably be (considerably) lower as we have a relatively small model. However, mini-batch still has advantages over using full batches, as mini-batch has noisy gradients that can help with generalization and getting out local minima.

Simply running the model for 3000 iterations with batch size set to 400 resulted in an accuracy on the test set of **48.46**%; the highest accuracy on the test set of the 30 times that the model was evaluated on the test set is 49.63% at step 2400. Looking at the loss graph in figure 4 it seemed possible that there could be *some* improvement left, running the model for 6000 iterations was tried. This resulted in a really unstable loss at a large number of iterations; this is probably due to the model having a too high learning rate. It could also be solved by using a learning rate schedule. Here, the learning rate stays the same throughout the training, also when we hit a loss plateau which results in (presumably) oscillating around minima (see figure 5). The training accuracy has increased quite a bit to around 60.0% for larger batch sizes and larger number of iterations in comparison to the default parameters as can be seen in the figures. Changing the batch size to a value of 800 and using 3000 steps did not result in an improvement.

An attempt has been made at finding the best learning rate. The default parameter for the learning rate is expected to be too high because of the fluctuating loss and accuracy. A learning rate to high can result in an unstable training. A learning rate too small can result in the training being very slow and the model possibly not converging. A learning rate of 2e-2 did not result in an improvement (as expected), reducing the learning rate of 2e-4 gave a much more stable learning process and allowed for a higher number of iterations. A test accuracy of **52.03%** on iteration 10,000 (with batch size set to 400) was obtained. Batch size set to 800 did not result in a considerable improvement. A last attempt with the current architecture was adding weight decay to the SGD optimizer; this is a form of regularization. Adding a weight decay of 1e-3, 1e-2 and 1e-1 resulted in similar results; **52.2%** on the test set for 1e-1. Adding momentum of 0.9 to the SGD optimizer did not improve results.

Next, we tried some ideas that would possibly give larger improvements. As the architecture of the network plays a large role in the performance, we first attempt to extend the architecture for the current settings. With batch size set to 400, number of iterations to 10,000 and learning rate to 2e-4, we tried adding a hidden layer with 100 neurons before the hidden layer with 100 neurons. The results of this architecture were poor; after having done several experiments it seemed that adding one hidden layer with more neurons so that the network consists of two hidden layers almost always results in a (large) decline of performance. It turned out that for *some* weight initialization methods like using the one from question 2.1 and for instance Xavier weight initialization it results in this bad performance. The test accuracy consistently stays 10%. Using He or Kaiming initialization however resulted for the before mentioned settings in 36.99% on the test set. The training process seemed to indicate it was learning extremely slow or not improving at all after a certain amount of steps. Changing the learning rate back to 2e-3 for this Kaiming initialization improves to 45% test accuracy. Using the settings of batch size 400, number of iterations to 10,000 and learning rate to 2e-3, but with hidden layers of 1000 and 100 neurons, the test accuracy obtained was 49.51%. Adding momentum of 0.9 and/or weight decay of 0.1 again did not gave an improvement. Using Adam optimizer with all the same settings (without momentum) also did not improve results. However, changing the architecture to 1000, 1000 did improve results considerably. The highest accuracy of **55.07**% was obtained with the parameter settings from before; batch size of 400, 10,000 steps, SGD optimizer with learning rate of 2e-3 and weight decay of 0.1. The training accuracy reaches 85% at the end of training. The loss curve and the training accuracy can be seen in figure 6. For these settings, the

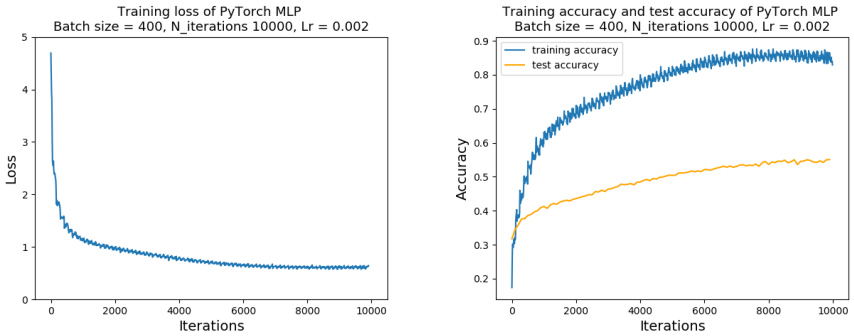

Figure 6: The loss and accuracy curves of PyTorch MLP with two hidden layers both with 1000 neurons for batch size 400, number of iterations equal to 10000, learning rate of 2e-3, SGD Optimizer with 0.1 weight decay and He initialization

training and test results have a significant gap in accuracy which indicates that our model overfits on the training data. Even though we have improved the first results a bit, the multi-layer perceptron is probably not the most suitable for images.

3 Custom Module: Batch Normalization

Question 3.1 Implement the Batch Normalization operation as a nn.Module at the designated position in the file custom_batchnorm.py.

Question 3.2 a) Compute the backpropagation equations for the batch normalization operation.

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

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

  $\frac{\partial L}{\partial \gamma_j} = \sum_s \sum_i \left(\frac{\partial L}{\partial y_i^s}\right)\left(\hat{x}_i^s\right)$

  $\nabla_\gamma(L) = \sum_s \left(\frac{\partial \mathbf{y^s}}{\partial \boldsymbol{\gamma}}\right)^T \nabla_{y^s}(L)$

  $$\nabla_\gamma(L) = \sum_s \begin{bmatrix} \frac{\partial y_1^s}{\partial \gamma_1} & \frac{\partial y_1^s}{\partial \gamma_2} & \dots \\ \frac{\partial y_2^s}{\partial \gamma_1} & \frac{\partial y_2^s}{\partial \gamma_2} & \dots \\ \dots & \dots & \dots \end{bmatrix}^T \nabla_{y^s}(L)$$

  $$\nabla_\gamma(L) = \begin{bmatrix} x_1^1 & x_2^2 & \dots \\ x_1^2 & x_2^2 & \dots \\ \dots & \dots & \dots \end{bmatrix}^T \sum_s \nabla_{y^s}(L)$$

  $\nabla_\gamma(L) = \left(\hat{\mathbf{x}}\right)^T \sum_s \nabla_{y^s}(L)$

- $\frac{\partial L}{\partial \beta} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \beta}$

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

$$\frac{\partial L}{\partial \beta_j} = \sum_s \sum_i \Big(\frac{\partial L}{\partial y_i^s}\Big)\Big(1\Big)$$
$$\nabla_\beta(L) = \sum_s \nabla_{y^s}(L)$$

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

$$\frac{\partial L}{\partial \sigma^2} = \frac{\partial L}{\partial \hat{x}}\frac{\partial \hat{x}}{\partial \sigma^2}$$
$$\frac{\partial L}{\partial \sigma_j^2} = \sum_s \sum_i \frac{\partial L}{\partial \hat{x}_i^s}\frac{\partial \hat{x}_i^s}{\partial \sigma_j^2}$$

$$\frac{\partial L}{\partial \sigma_j^2} = \sum_s \sum_i \frac{\partial L}{\partial \hat{x}_i^s}\Big(-\tfrac{1}{2}\sum_{s=1}^{B}(x_j^s - \mu_j)(\sigma_j^2+\epsilon)^{-3/2}\Big)$$

$$\frac{\partial L}{\partial \mu} = \frac{\partial L}{\partial \hat{x}}\frac{\partial \hat{x}}{\partial \mu} + \frac{\partial L}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial \mu}$$
$$\frac{\partial L}{\partial \mu_j} = \sum_s \sum_i \frac{\partial L}{\partial \hat{x}_i^s}\frac{\partial \hat{x}_i^s}{\partial \mu_j} + \frac{\partial L}{\partial \sigma_j^2}\frac{\partial \sigma_j^2}{\partial \mu_j}$$
$$\frac{\partial L}{\partial \mu_j} = \sum_s \sum_i \frac{\partial L}{\partial \hat{x}_i^s}\Big(-\frac{1}{\sqrt{\sigma_j^2+\epsilon}}\Big) + \frac{\partial L}{\partial \sigma_j^2}\Big(\tfrac{1}{B}\sum_{s=1}^{B} -2(x_j^s - \mu_j)\Big)$$
$$\frac{\partial L}{\partial \mu_j} = \sum_s \sum_i \frac{\partial L}{\partial \hat{x}_i^s}\Big(-\frac{1}{\sqrt{\sigma_j^2+\epsilon}}\Big) + \frac{\partial L}{\partial \sigma_j^2}\Big(-2(\mu_j - \mu_j)\Big)$$
$$\frac{\partial L}{\partial \mu_j} = \sum_s \sum_i \frac{\partial L}{\partial \hat{x}_i^s}\Big(-\frac{1}{\sqrt{\sigma_j^2+\epsilon}}\Big)$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial \hat{x}}\frac{\partial \hat{x}}{\partial x} + \frac{\partial L}{\partial \mu}\frac{\partial \mu}{\partial x} + \frac{\partial L}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial x}$$
$$\frac{\partial L}{\partial x_j^r} = \sum_s \sum_i \frac{\partial L}{\partial \hat{x}_i^s}\frac{\partial \hat{x}_i^s}{\partial x_j^r} + \frac{\partial L}{\partial \mu_i}\frac{\partial \mu_i}{\partial x_j^r} + \frac{\partial L}{\partial \sigma_i^2}\frac{\partial \sigma_i^2}{\partial x_j^r}$$
$$\frac{\partial L}{\partial x_j^r} = \sum_s \sum_i \Big(\frac{\partial L}{\partial \hat{x}_i^s}\Big)\Big(\frac{1}{\sqrt{\sigma_j^s+\epsilon}}\Big) + \Big(\frac{\partial L}{\partial \mu_i}\Big)\Big(\frac{1}{B}\Big) + \Big(\frac{\partial L}{\partial \sigma_i^2}\Big)\Big(\frac{1}{B}\ 2(x_j^s - \mu_j)\Big)$$
$$\frac{\partial L}{\partial x_j^r} = \sum_s \sum_i \Big(\frac{\partial L}{\partial \hat{x}_i^s}\Big)\Big(\frac{1}{\sqrt{\sigma_j^s+\epsilon}}\Big) - \Big(\frac{\partial L}{\partial \hat{x}_i^s}\frac{1}{\sqrt{\sigma_i^2+\epsilon}}\Big)\Big(\frac{1}{B}\Big) - \Big(\frac{\partial L}{\partial \hat{x}_i^s}\Big(\tfrac{1}{2}\sum_{s=1}^{B}(x_i^s - \mu_i)(\sigma_i^2 + \epsilon)^{-3/2}\Big)\Big(\tfrac{1}{B}\ 2(x_j^s - \mu_j)\Big)$$
$$\frac{\partial L}{\partial x_j^r} = \sum_s \sum_i \Big(\frac{\partial L}{\partial \hat{x}_i^s}\Big)\Big(\frac{1}{\sqrt{\sigma_j^2+\epsilon}}\Big) - \Big(\frac{\partial L}{\partial \hat{x}_i^s}\frac{1}{B}\frac{1}{\sqrt{\sigma_i^2+\epsilon}}\Big) - \Big(\frac{\partial L}{\partial \hat{x}_i^s}\frac{1}{B}\frac{1}{\sqrt{\sigma_j^2+\epsilon}}\frac{x_i^s - \mu_i}{\sqrt{\sigma_i^2+\epsilon}}\frac{x_j^s - \mu_j}{\sqrt{\sigma_i^2+\epsilon}}\Big)$$
$$\frac{\partial L}{\partial x_j^r} = \frac{1}{B}\frac{1}{\sqrt{\sigma_j^2+\epsilon}}\Big(\sum_s \sum_i \Big(\frac{\partial L}{\partial \hat{x}_i^s}B\Big) - \Big(\frac{\partial L}{\partial \hat{x}_i^s}\Big) - \Big(\frac{\partial L}{\partial \hat{x}_i^s}\hat{x}_i^s\ \hat{x}_j^s\Big)\Big)$$

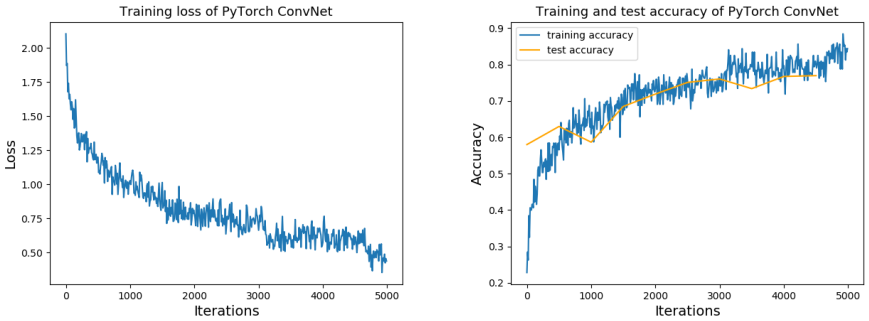

Figure 7: PyTorch ConvNet training loss on the left and training accuracy and test accuracy on the right (evaluated each *eval_frequency = 500* iterations) for the default parameters

$$\frac{\partial L}{\partial x_j^r} = \frac{1}{B}\frac{1}{\sqrt{\sigma_j^2+\epsilon}}\left(\sum_s \sum_i \left(\frac{\partial L}{\partial y_i^s}\left(\gamma_j\right)B\right) - \left(\frac{\partial L}{\partial y_i^s}\left(\gamma_j\right)\right) - \left(\frac{\partial L}{\partial y_i^s}\left(\gamma_j\right)\hat{x}_i^s\ \hat{x}_j^s\right)\right)$$

Question 3.2 b) Implement the Batch Norm operation as a torch.autograd.Function.

Question 3.2 c) Create a nn.Module with γ and β as nn.Parameters as before. In the forward pass call the autograd function.

4 Pytorch CNN

Question 4 Implement the ConvNet specified in Table 1 inside *convnet_pytorch.py* file by following the instructions inside the file.

Using the default settings and Adam optimizer the accuracy on the test set is **76.95**%. Figure 7 shows the training accuracy, training loss and test accuracy for the model. Although the model is a relatively small ConvNet, the training accuracy and test accuracy start both as high as 55%. The training accuracy reaches around 85% at the end of training. The training accuracy and test accuracy are both visible in the same plot. If we consider the test set a validation set we can see that both training and validation continue to improve and there is not a large difference between training and validation. This indicates that we are likely not overfitting. This ConvNet based on VGGNet thus performs as expected better on CIFAR-10 than multi-layer perceptrons based on NumPy or PyTorch.