Deep Learning Assignment 1: MLPs, CNNs and Backpropagation

Paul ten Kaate
10743367
paultenkaate@outlook.com

November 15, 2019

Abstract

In this report, several implementations of a neural network are explored. First, a neural network is built using numpy. Hereafter, pytorch is used to build the same network setup. Hereafter a batch normalization algorithm is written. Lastly, a CNN has been implemented.

1 MLP backprop and NumPy implementation

Question 1.1a

The gradients were computed as follows:

(i) $\frac{\partial L}{\partial x_i^{(N)}} = -\frac{t_i}{x_i^{(N)}}$

(ii)

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

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

$$\frac{\mathbb{1}(i=j)\exp(\tilde{x}_i^{(N)})\sum_k \exp(\tilde{x}_k^{(N)}) - \exp(\tilde{x}_i^{(N)})\exp(\tilde{x}_j^{(N)})}{\big(\sum_k \exp(\tilde{x}_k^{(N)})\big)^2} =$$

$$\frac{\exp(\tilde{x}_i^{(N)})}{\sum_k \exp(\tilde{x}_k^{(N)})}\frac{\mathbb{1}(i=j)\sum_k \exp(\tilde{x}_k^{(N)}) - \exp(\tilde{x}_j^{(N)})}{\sum_k \exp(\tilde{x}_k^{(N)})} =$$

$$\frac{\exp(\tilde{x}_i^{(N)})}{\sum_k \exp(\tilde{x}_k^{(N)})}\Big(\mathbb{1}(i=j) - \frac{\exp(\tilde{x}_j^{(N)})}{\sum_k \exp(\tilde{x}_k^{(N)})}\Big) =$$

$$\sigma(\tilde{x}_i^{(N)})(\mathbb{1}(i=j) - \sigma(\tilde{x}_j^{(N)})) =$$

$$x_i^{(N)}(\mathbb{1}(i=j) - x_j^{(N)})$$

(iii) $\frac{\partial x_i^{(l<N)}}{\partial \tilde{x}_j^{(l<N)}} = \begin{cases} 1 & \text{if j = i and } \tilde{x}_j^{(l<N)} > 0 \\ a & \text{if j = i and } \tilde{x}_j^{(l<N)} < 0 \\ 0 & \text{if j} \neq \text{i} \end{cases}$

(iv) $\dfrac{\partial \tilde{x}_i^{(l)}}{\partial x_j^{(l-1)}} = W_{ij}$

(v) $\dfrac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^{(l)}} = \begin{cases} x_k^{(l-1)} & \text{if j = i} \\ 0 & \text{if j} \neq \text{i} \end{cases}$

(vi) $\dfrac{\partial \tilde{x}_i^{(l)}}{\partial b_j^{(l)}} == \begin{cases} 1 & \text{if j = i} \\ 0 & \text{if j} \neq \text{i} \end{cases}$

Question 1.1b

Using the derivatives from question 1.1a, and using the chain rule as provided in the question, the gradients with respect to the loss are given hereunder:

(i) $\dfrac{\partial L}{\partial \tilde{x}_j^{(n)}} = \sum_i^D \dfrac{\partial L}{\partial x_i^{(N)}} \dfrac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = \sum_i^D \dfrac{\partial L}{\partial x_i^{(N)}} \, x_i^{(N)}(\mathbb{1}(i=j) - x_j^{(N)})$

$$\frac{\partial L}{\partial \tilde{x}^{(n)}} = \frac{\partial L}{\partial x^{(N)}} \left(I \otimes x^{(N)} - x^{(N)} \otimes x^{(N)} \right)$$

(ii) $\dfrac{\partial L}{\partial \tilde{x}_j^{(l<N)}} = \sum_i^D \dfrac{\partial L}{\partial x_i^{(l)}} \dfrac{\partial x_i^{(l)}}{\partial \tilde{x}_j^{(l)}} = \begin{cases} \dfrac{\partial L}{\partial x_j^{(l)}} & \text{if } \tilde{x}_j^{(l<N)} > 0 \\ a \, \dfrac{\partial L}{\partial x_j^{(l)}} & \text{if } \tilde{x}_j^{(l<N)} < 0 \end{cases}$

(iii) $\dfrac{\partial L}{\partial x_j^{(l<N)}} = \sum_i^D \dfrac{\partial L}{\partial \tilde{x}_i^{(l+1)}} \dfrac{\partial \tilde{x}_i^{(l+1)}}{\partial x_j^{(l)}} = \sum_i^D \dfrac{\partial L}{\partial \tilde{x}_i^{(l+1)}} \, W_{ij}^{(l)}$

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

(iv) $$\frac{\partial L}{\partial W^{(l)}_{jk}} = \sum_i^D \frac{\partial L}{\partial \tilde{x}^{(l)}_i} \frac{\partial \tilde{x}^{(l)}_i}{\partial W^{(l)}_{jk}} = \frac{\partial L}{\partial \tilde{x}^{(l)}_j}\ x^{(l-1)}_k$$

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

(v) $$\frac{\partial L}{\partial b^{(l)}_j} = \sum_i^D \frac{\partial L}{\partial \tilde{x}^{(l)}_i} \frac{\partial \tilde{x}^{(l)}_i}{\partial b^{(l)}_j} = \frac{\partial L}{\partial \tilde{x}^{(l)}_j}$$

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

Question 1.1c

If a batchsize different than 1 is used, this will slightly change some formulas. The gradient of $b^{(l)}$ will be calculated by taking the sum over each individual $\frac{\partial L}{\partial \tilde{x}^{(l)}}$. The same process applies for the gradient of $W^{(l)}$. The gradient $\frac{\partial L}{\partial x^{(l<N)}}$ will not change, as the shape of this matrix will grow in row size the same way as $\frac{\partial L}{\partial \hat{x}^{(l+1)}}$ will when the batch size is increased. The gradient of $\tilde{x}^{(l<N)}_j$ does not change, as this is an operation that is performed on each individual element in a matrix. The same applies for the gradient of $\tilde{x}^{(N)}$, as this is an operation that happens row-wise.

Question 1.2

The forward and backward pass functions for each layer can be found in *modules.py.* After running *train_mpl_numpy.py* with the default values of the parameters, a final accuracy of **0.4732** is achieved. Each 100^{th} iteration, the accuracy and loss is evaluated. As can be seen in Figure , the accuracy on the training data keeps increasing even at the 1500^{th} iteration, however the test accuracy already reaches an accuracy of 0.4604 at the 500^{th} iteration. A

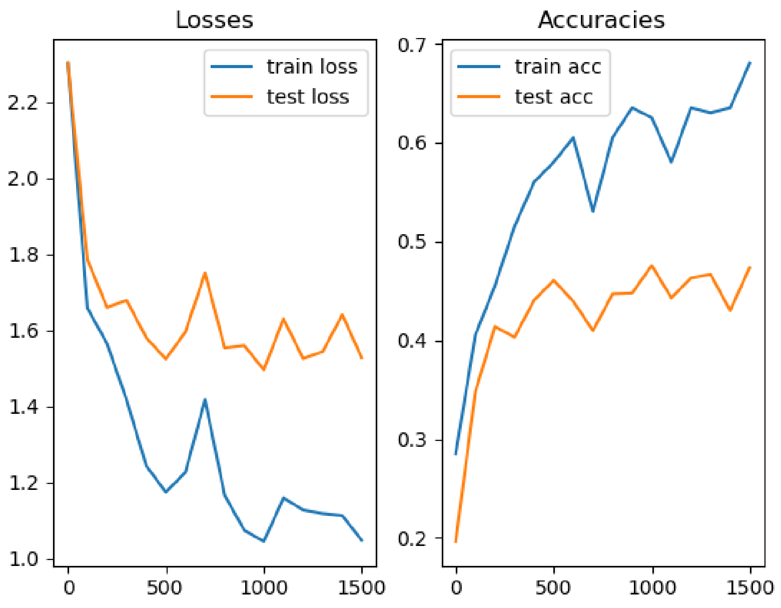

Figure 1: Train and test accuracy with the MLP numpy implementation

threshold that requires the test accuracy to improve each 100^{th} iteration could be used to prevent the network of overfitting.

2 PyTorch MLP

Question 2

In section 2, the same network setup was used, only now pytorch was used to created the layers and activations. With the same default values of the parameters, the final prediction accuracy on the test set is 0.4079. This is lower than the accuracy with the numpy implementation, however we can see in Figure that the accuracy is still increasing. Indeed, when increasing the amount of iterations to 2500, the final test accuracy rises to 0.4615, as can be seen in Figure . Furthermore, an increase in accuracy can be found when increasing the amount

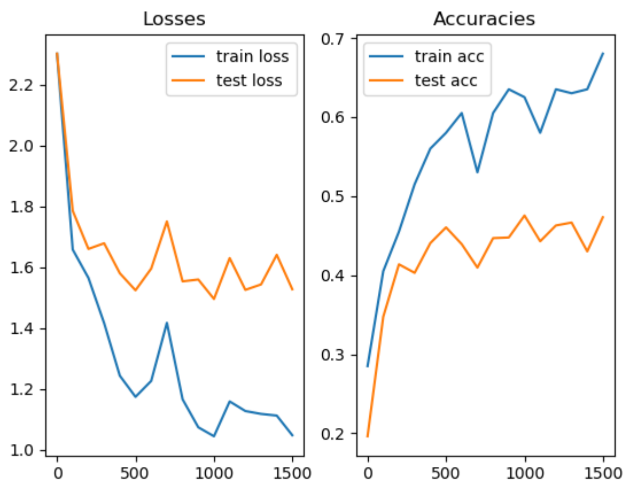

Figure 2: Train and test accuracy with the MLP pytorch implementation with hidden layers = [100] and 1500 iterations

of hidden layers. Different setups were tried, with a different amount of layers, where each following layer is either the same shape or of a smaller shape than the previous layer. Furthermore an Adam optimizer was used, as this optimizer is quicker than the SGD optimizer A setup with the hidden layers *[300, 200, 100, 100, 50, 50]*, with 2500 iterations, resulted in the best final accuracy of 0.5226 as can be seen in Figure .

Custom Module: Batch Normalization

Question 3.1

β and γ where initialized with *nn.parameters()*. Running the *unittests.py* file gives an ok.

Question 3.2a

The gradients of β and γ are as follows:

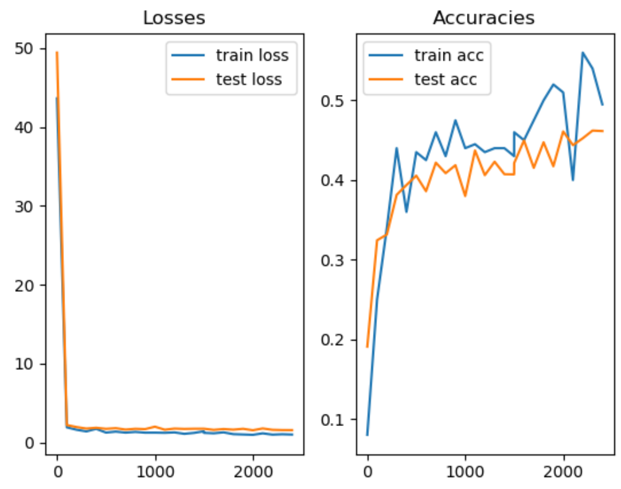

Figure 3: Train and test accuracy with the MLP pytorch implementation with hidden layers = [100] and 2500 iterations

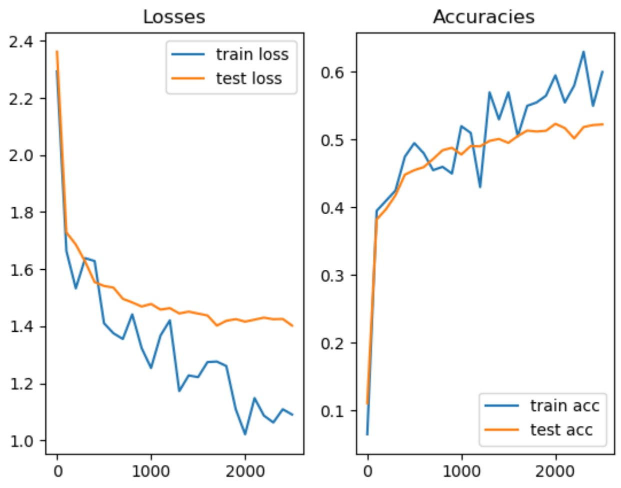

Figure 4: Train and test accuracy with the MLP pytorch implementation with hidden layers = [300, 200, 100, 100, 50, 50] and 2500 iterations

(i) $\dfrac{\partial y_i}{\partial \gamma_j} = \begin{cases} \hat{x} & \text{if j = i} \\ 0 & \text{if j != i} \end{cases}$

$\dfrac{\partial L}{\partial y}\dfrac{\partial y}{\partial \gamma} = \sum_{s=1}^{B} \dfrac{\partial L}{\partial y^s}\hat{x^s}$

(ii) $\dfrac{\partial y_i}{\partial \beta_j} = \begin{cases} 1 & \text{if j = i} \\ 0 & \text{if j != i} \end{cases}$

$\dfrac{\partial L}{\partial y}\dfrac{\partial y}{\partial \beta} = \sum_{s=1}^{B} \dfrac{\partial L}{\partial y^s}$

For the last derivation, I was not able to do this by hand. For the implementation in python, I used the following derivation, found at *https://kevinzakka.github.io/2016/09/14/batch_ normalization/*

$$\frac{\partial f}{\partial x_i} = \frac{B\dfrac{\partial f}{\partial \hat{x}_i} - \sum_{s=1}^{B}\dfrac{\partial f}{\partial \hat{x}^s} - \hat{x}_i \sum_{s=1}^{B}\dfrac{\partial f}{\partial \hat{x}^s} \cdot \hat{x}^s}{B\sqrt{\sigma^2 + \epsilon}}$$

Question 3.2b and 3.2c

The functions can be found in the *custom_ batchnorm.py* file.

3 PyTorch CNN

Question 4

A CNN was implemented with an architecture provided in Table 1 of the assignment, and with the default parameters. The loss score, due to computational difficulties, has been calculated by taking a new batch of 1000 images with the *.next_ batch()* function that was provided, instead of using the full test set every evaluation moment. The final accuracy after 5000 iterations is 0.726, the losses and accuracies can be found in Figure 5. The final accuracy is lower than the given expected of 0.75, however if we look at the graph, the test accuracy lowers and the loss rises at iteration 5000, while the train loss

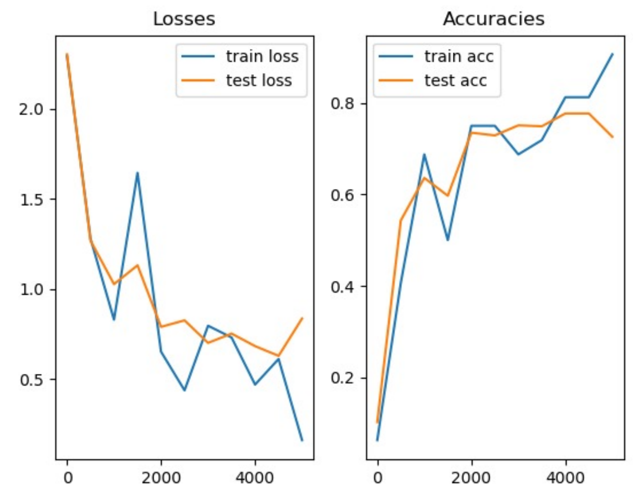

Figure 5: Accuracy and loss over the train and test set with the CNN.

converges to zero. This means that the network is already overfitting on the data.

Looking at the final accuracy of 0.726 with the CNN and comparing this with accuracies of 0.4732 and 0.5226 of respectively the numpy and pytorch implementation of an MLP network, the conclusion can be drawn that for this dataset, the CNN solution works considerably better than the MLP network. This is consistent with the general view on image recognition, that better results can be achieved with a CNN, than with an MLP.