Hannah Lim
10588973
hannah_lim@outlook.com

Assignment 1
Deep Learning, 2019

November 15, 2019

1 MLP backprop and NumPy implementation

1.1 Analytical derivation of gradients

1.1a)

Cross Entropy Module:

$$\frac{\partial L}{\partial x_i^{(N)}} = \frac{\partial - \sum_i t_i log(x_i^{(N)})}{\partial x_i^{(N)}} = \frac{-t_i}{x_i^{(N)}}, \text{ for all i = 0,...,N}$$
$$\frac{\partial L}{\partial x^{(N)}} = \frac{-t}{x^{(N)}}$$

Softmax Module:

$$\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = \frac{\partial \frac{exp(\tilde{x}_i^{(N)})}{\sum_k^{d_N} exp(\tilde{x}_k^{(N)})}}{\partial \tilde{x}_j^{(N)}} = \begin{cases} = \frac{\sum_k^{d_N} exp(\tilde{x}_k^{(N)}) exp(\tilde{x}_i) - exp(\tilde{x}_i)^2}{\sum_k^{d_N} exp(\tilde{x}_k^{(N)})^2} & \text{if } i = j \\ = \frac{-exp(\tilde{x}_i) exp(\tilde{x}_j)}{\sum_k^{d_N} exp(\tilde{x}_k^{(N)})^2} & \text{if } i \neq j \end{cases}$$
$$\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} = diag(x^{(N)}) - x^{(N)} {x^{(N)}}^T$$

Leaky ReLU Module:

$$\frac{\partial x^{(l<N)}}{\partial \tilde{x}^{(l<N)}} = \frac{\partial max(0, \tilde{x}^{(l<N)}) + a \cdot min(0, \tilde{x}^{(l<N)})}{\partial \tilde{x}^{(l<N)}} = \begin{cases} 1 & \text{if } \tilde{x}^{(l<N)} > 0 \\ a & \text{if } \tilde{x}^{(l<N)} \leq 0 \end{cases}$$
$$= \mathbb{1}(\tilde{x_i}^{(l)} \leq 0)(a) + \mathbb{1}(\tilde{x_i}^{(l)} > 0)(\tilde{x_i}^{(l)})$$

Linear Module:

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

$$\frac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^{(l)}} = \frac{\partial W_i^{(l)} x^{(l-1)} + b_i^{(l)}}{\partial W_{jk}^{(l)}} = \mathbb{1}_{i=j} x_k^{(l-1)}$$

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

1.1b)

Softmax Module backward:

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

Leaky Relu Module backward:

$$\frac{\partial L}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial x^{(l)}} \frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}} = \frac{\partial L}{\partial x^{(l)}} (\mathbb{1}(\tilde{x_i}^{(l)} \leq 0)(a) + \mathbb{1}(\tilde{x_i}^{(l)} > 0)(\tilde{x_i}^{(l)}))$$

Linear Module backward:

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

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

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

1.1c)

If a batchsize of B $\neq$ 1 is used, the cross entropy loss is computed by taking the mean of all the individual losses of the batch. All the gradients computed with back propagation will be of a matrix with for each row the gradient of a sample of the batch. So the samples of the batch will run parallel trough the model. The gradients of the bias in the linear module will be summed, so it fits the bias parameters. For the gradients of the parameters, instead of a vector multiplication it is now a matrix multiplication with the the previous gradient and the input vector of the model.

1.2 NumPy implementation

The Numpy MLP model is trained with the default parameters. In the top graph of figure 1 the accuracy on the training and the test data during training is shown and in the bottom graph the loss of the training data is shown. The highest accuracy for the test data with default parameters is 47.66 % after approximately 900 steps. The target line (46% accuracy) is shown with a dashed green line. The test data stays around this line from the 900 steps. The loss of the training data starts a little above 2.2 and quickly (around 200-300 steps) drops down and will stay between the range 1.4-1.8.

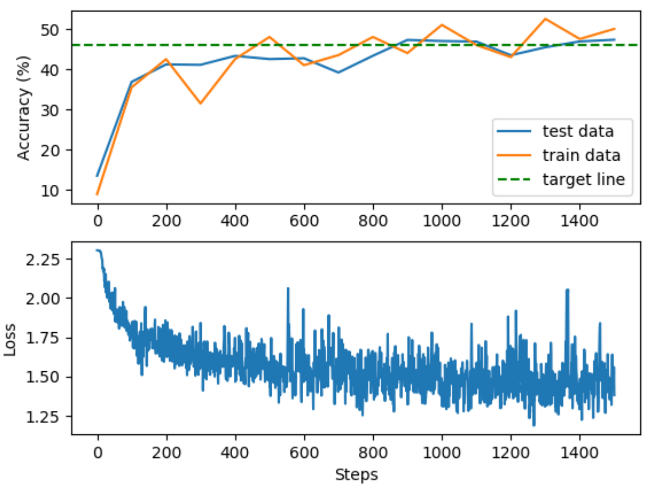

Figure 1: NumPy implemenation; top graph: accuracy of the train and test data, bottom graph: loss of the training data (with default settings)

In the 2 we can see that the after approximately 1800 steps the accuracy on the test data stays around the 46%, while the accuracy on training data keeps increasing. So the model is overfitting from the 1800 steps and therefore the get the best model, we should stop iterating before overfitting. On the default settings with 1500 steps, the model is not yet overfitting (only around the 5% difference in accuracy).

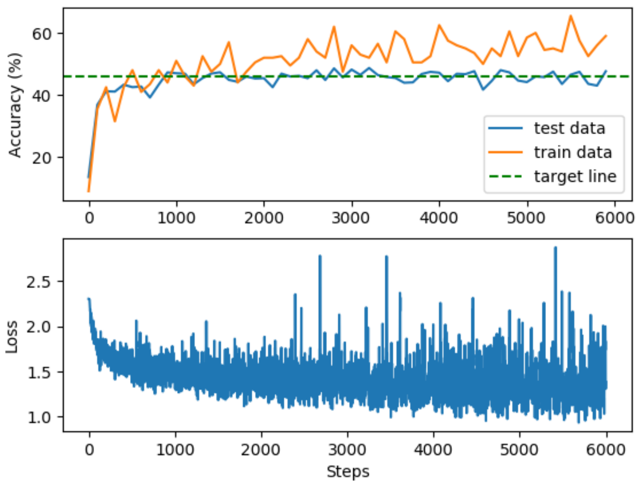

Figure 2: NumPy implemenation; top graph: accuracy of the train and test data, bottom graph: loss of the training data (with default settings left unchanged except an increasement of the number of steps to 6000 steps)

2 PyTorch MLP

The Pytorch MLP model is trained with default settings. This results in a accuracy 42.9% for the test set.
To find the highest accuracy and the lowest loss, I experimented with the following parameters for the PyTorch MLP: number of steps, changing amount of layers, changing amount of units per layer, different optimizers (SGD and Adam with or without Amsgrad), learning rate, negative slope and a regularization term.

In top of figure 3 the accuracy on the training and test data during training are shown and in the bottom graph the loss of the training data is shown with the following parameters:

1. number of hidden layers: 5, with number of units: [1000, 500, 200, 100, 100]
2. negative slope: 0.002
3. optimizer: Adam optimizer with Amsgrad set to True
4. learning rate: 1e-4
5. weight decay: 1e-5

With these parameters the highest accuracy (54.1%) for the test data was found after 6000 steps. But in the graph we can see that from around the 2000 steps the accuracy remains pretty much the same. In the graph we can see that the accuracy increases very fast up till 800 steps. From 800 steps the slope of the accuracy of the test set becomes very small.
Also after 2000 steps the loss will remain bellow 0.5.

We can see that from around the 3000 steps the accuracy of the test set barely increases while the accuracy of the train data still increases, which results in a divergence graph and can be concluded that the model is overfitting. In addition the loss of the train data is very low (0.5). Plotting the loss of the train and test data would probably result in a divergence graph.

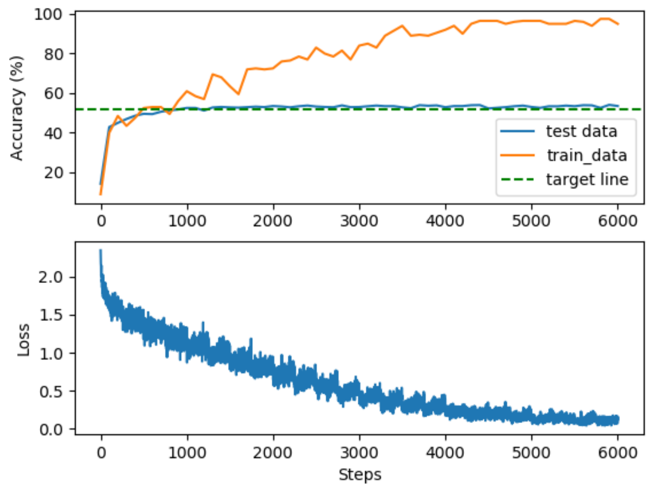

Figure 3: PyTorch MLP; top graph: accuracy of the test data, bottom graph: loss of the training data (with parameters; number of hidden layers: 5, with number of units: [1000, 500, 200, 100, 100], negative slope: 0.002, optimizer: Adam optimizer with Amsgrad set to True, learning rate: 1e-4, weight decay: 1e-5)

3 Custom Module: Batch Normalization

3.1 Automatic differentiation

See code.

3.2 Manual implementation of backward pass

3.2a)

$$(\tfrac{\partial L}{\partial \gamma})_j = \textstyle\sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial \gamma_j} = \sum_s \sum_i \frac{L}{\partial \gamma_i^s} \mathbb{1}\{i = j\} \hat{x}_j^s = \sum_s \frac{\partial L}{\partial y_j^s} \hat{x}_j^s$$

$$(\tfrac{\partial L}{\partial \beta})_j = \textstyle\sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial \beta_j} = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \mathbb{1}\{i = j\} = \sum_s \frac{\partial L}{\partial y_j^s}$$

$$(\tfrac{\partial L}{\partial x})_j^r = \textstyle\sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial x_j^r}$$
$$\frac{\partial y_i^s}{\partial x_j^r} = \frac{\partial \gamma_i \hat{x}_i^s + \beta_i}{\partial x_j^r} = \gamma_i \frac{\partial \hat{x}_i^s}{\partial x_j^r}$$

$$\frac{\partial \hat{x}_i^s}{\partial x_j^r} = \frac{\partial}{\partial x_j^r} \frac{x_i^s - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} = \frac{\sqrt{\sigma_i^2+\epsilon}\frac{\partial}{\partial x_j^r}(x_i^s-\mu_i) - (x_i^s-\mu_i)\frac{\partial}{\partial x_j^r}\sqrt{\sigma_i^2+\epsilon}}{\sigma_i^2+\epsilon}$$

$$\frac{\partial}{\partial x_j^r}(x_i^s - \mu_i) = \mathbb{1}(i=j)\mathbb{1}(s=r) - \frac{\mathbb{1}(i=j)}{B}$$

$$\frac{\partial}{\partial x_j^r}\sqrt{\sigma_i^2+\epsilon} = \frac{\frac{\partial}{\partial x_j^r}\sigma_i^2+\epsilon}{2\sqrt{s_i^2+\epsilon}} = \frac{\frac{2\mathbb{1}\{i=j\}}{B}(x_i^r-\mu_i)}{2\sqrt{s_i^2+\epsilon}} = \frac{x_i^r-\mu_i}{B\sqrt{\sigma_i^2+\epsilon}}$$

filling in:

$$\frac{\partial \hat{x}_i^s}{\partial x_j^r} = \frac{\partial}{\partial x_j^r} \frac{x_i^s - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} = \frac{\sqrt{\sigma_i^2+\epsilon}\frac{\partial}{\partial x_j^r}(x_i^s-\mu_i) - (x_i^s-\mu_i)\frac{\partial}{\partial x_j^r}\sqrt{\sigma_i^2+\epsilon}}{\sigma_i^2+\epsilon} =$$

$$\frac{\sqrt{\sigma_i^2+\epsilon}\mathbb{1}(i=j)\mathbb{1}(s=r) - \frac{\mathbb{1}(i=j)}{B} - (x_i^s-\mu_i)\frac{x_i^r-\mu_i}{B\sqrt{\sigma_i^2+\epsilon}}}{\sigma_i^2+\epsilon}$$

$$\frac{\partial y_i^s}{\partial x_j^r} = \frac{\partial \gamma_i \hat{x}_i^s + \beta_i}{\partial x_j^r} = \gamma_i \frac{\partial \hat{x}_i^s}{\partial x_j^r} = \gamma_i \frac{\sqrt{\sigma_i^2+\epsilon}(\mathbb{1}(i=j)\mathbb{1}(s=r) - \frac{\mathbb{1}(i=j)}{B}) - (x_i^s-\mu_i)\frac{x_i^r-\mu_i}{B\sqrt{\sigma_i^2+\epsilon}}}{\sigma_i^2+\epsilon}$$

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

$$\sum_s \sum_i \frac{\partial L}{\partial y_i^s}\gamma_i \frac{\sqrt{\sigma_i^2+\epsilon}(\mathbb{1}(i=j)\mathbb{1}(s=r) - \frac{\mathbb{1}(i=j)}{B}) - (x_i^s-\mu_i)\frac{x_i^r-\mu_i}{B\sqrt{\sigma_i^2+\epsilon}}}{\sigma_i^2+\epsilon}$$

$$= \frac{\gamma_j}{B\sqrt{\sigma_j^2+\epsilon}}(B\frac{\partial L}{\partial y_j^r} - \sum_s \frac{\partial L}{\partial y_j^s} - \hat{x_j^r}\sum_s \frac{\partial L}{\partial y_j^s}\hat{x}_j^s))$$

3.2b)

See code.

3.2c)

See code.

4 PyTorch CNN

The PyTorch CNN is trained with default settings. This results in an accuracy of 78.81% for the test data. In top of figure 4 the accuracy on the training and

test data during training are shown and in the bottom graph the loss of the training data is shown. Until 800 steps the accuracy of both the train and the test set increases fast and thereafter it keeps increasing but slowly. The loss begins at around the 2.5 but quickly decreases and will stay a little below 1.0 after 2000 iterations.
As we can see that the accuracy of the test set training set are similar we can conclude that the model is not overfitting.

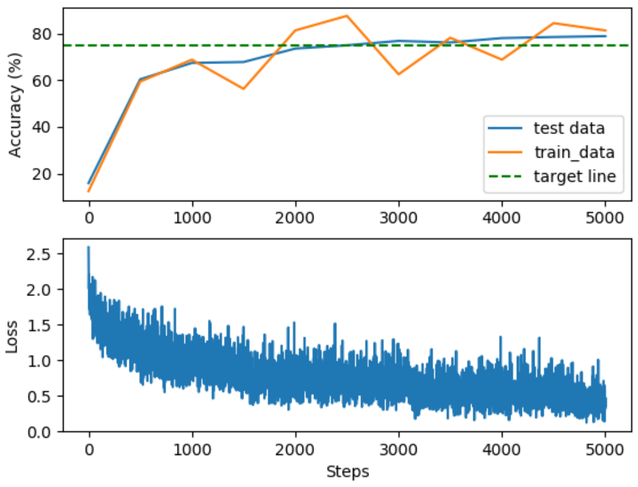

Figure 4: PyTorch CNN; top graph: accuracy of the test data, bottom graph: loss of the training data (with default settings)

5 Conclusion

It can be concluded that the most appropriate model for classifying images (based on the CIFAR-10 dataset) is the PyTorch CNN since it results in an accuracy of 78.81% on the testdata and is not overfitting.
A MLP implemented with NumPy and PyTorch give a similar accuracy and changing parameters will improve this. Especially increasing the layers of the MLP will improve the accuracy. In addition, adding more units to the first layers and less units in the last layers will again improve the accuracy.
The highest accuracy received in this assignment with the PyTorch MLP implementation was 54.1% which is considerably lower than the accuracy reached with the CNN. The reason the accuracy of the CNN is much higher is because CNNs are able to preserve the spatial information of an image, while the MLPs lose the spatial information.