Deep Learning - Assignment 1

Albert Harkema - 12854794
Faculty of Natural Sciences, Mathematics and Computer Science
University of Amsterdam
Amsterdam
`albertharkema@gmail.com`

Abstract

In this paper a discussion follows of several deep learning implementations. Firstly, a multi-layer perceptron was constructed using purely NumPy routines. Secondly, this process was repeated with PyTorch routines. Thirdly, a custom batch normalization module was added to PyTorch' functionality. Fourthly, a convolutional Neural Network was implemented using PyTorch routines. Accuracies and loss curves are discussed for each of these implementations.

1 MLP backprop and NumPy implementation

1.1 Analytical derivation of gradients

Question 1.1a)
1:

$$\frac{\partial L}{\partial x^{(N)}} \tag{1}$$
$$= \frac{-t_i}{x_i^{(N)}} \tag{2}$$
$$= \frac{-t}{x^{(N)}} \in \mathbb{R}^{1\times d_N} \tag{3}$$

2:
For the derivative of the softmax it is wise to make a distinction between two cases: one in which i = j and another in which i $\neq$ j. For i $\neq j$:

$$= \frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} \frac{exp(\tilde{x}_i^{(N)})}{\sum_{k=1}^{K} exp(\tilde{x}_k^{(N)})} \tag{4}$$
$$= \frac{\frac{\partial}{\partial \tilde{x}_j} exp(\tilde{x}_i^{(N)}) \cdot \sum_{k=1}^{K} exp(\tilde{x}_k^{(N)}) - exp(\tilde{x}_i^{(N)})(\frac{\partial}{\tilde{x}_j} \sum_{k=1}^{K} exp(\tilde{x}_k^{(N)}))}{\sum_{k=1}^{K} exp(\tilde{x}_k^{(N)}))^2} \tag{5}$$
$$= \frac{0 - exp(\tilde{x}_i^{(N)})(exp(\tilde{x}_j^{(N)}))}{(\sum_{k=1}^{K} exp(\tilde{x}_k^{(N)}))^2} \tag{6}$$
$$= -\frac{exp(\tilde{x}_i^{(N)})}{\sum_{k=1}^{K} exp(\tilde{x}_k^{(N)})} \frac{exp(\tilde{x}_j^{(N)})}{\sum_{k=1}^{K} exp(\tilde{x}_k^{(N)})} \tag{7}$$
$$= -x_i x_j \tag{8}$$

For i = j:

$$= \frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} \frac{exp(\tilde{x}_i^{(N)})}{\sum_{k=1}^{K} exp(\tilde{x}_k^{(N)})} \tag{9}$$

$$= \frac{\frac{\partial}{\partial x_j} exp(\tilde{x}_i^{(N)}) \cdot \sum_{k=1}^{K} exp(\tilde{x}_k^{(N)}) - exp(\tilde{x}_i^{(N)}) \cdot exp(\tilde{x}_j)}{(\sum_{k=1}^{K} exp(\tilde{x}_k^{(N)}))^2} \tag{10}$$

$$= \frac{exp(\tilde{x}_i)(\sum_{k=1}^{K} exp(\tilde{x}_k^{(N)}) - exp(\tilde{x}_j))}{} \tag{11}$$

$$= \frac{exp(\tilde{x}_i)}{\sum_{k=1}^{K} exp(\tilde{x}_k^{(N)})} \frac{\sum_{k=1}^{K} exp(\tilde{x}_k^{(N)}) - exp(\tilde{x}_j)}{\sum_{k=1}^{K} exp(\tilde{x}_k^{(N)})} \tag{12}$$

$$= \frac{exp(\tilde{x}_i)}{\sum_{k=1}^{K} exp(\tilde{x}_k^{(N)})} (1 - \frac{exp(\tilde{x}_j)}{\sum_{k=1}^{K} exp(\tilde{x}_k^{(N)})}) \tag{13}$$

$$= x_i^{(N)}(1 - x_j^{(N)}) \tag{14}$$

And the two cases can be condensed as:

$$x_i^{(N)}(\delta_{ij} - x_j^{(N)}) \in \mathbb{R}^{d_n \times d_n} \tag{15}$$

, where δ is the kronecker delta.

3: Derivative of Leaky ReLU w.r.t its input.

$$= \frac{\partial}{\partial \tilde{x}^{l<N}} LeakyReLU(\tilde{x}^{(l)}) \tag{16}$$

$$= \frac{\partial}{\partial \tilde{x}^{l<N}} max\big(0, \tilde{x}^{(l)}\big) + a \cdot min\big(0, \tilde{x}^{(l)}\big) \tag{17}$$

$$= \begin{cases} 1, & \text{if } \boldsymbol{x}^{l<N} \geq 0 \\ a, & \text{otherwise} \end{cases} \tag{18}$$

, which will be a diagonal matrix with the values sitting on the diagonal, so:

$$diag\left(\begin{cases} 1, & \text{if } \boldsymbol{x}^{l<N} \geq 0 \\ a, & \text{otherwise} \end{cases} \right) \in \mathbb{R}^{d_l \times d_l} \tag{19}$$

4:

$$= \frac{\partial \tilde{x}^{(l)}}{\partial \tilde{x}^{(l-1)}} \tag{20}$$

$$= \frac{\partial}{\tilde{x}^{(l-1)}} W^{(l)} x^{(l-1)} + b^{(l)} = W^{(l)} \tag{21}$$

, which we know is a matrix $\in \mathbb{R}^{d_l \times d_{l-1}}$

5:
Note: this derivation was inspired by example 5.12 of the mathematics for ML book. First off, we have $\partial \tilde{x}_t^{(l)} \in \mathbb{R}^{1 \times d_l}$ and W $\in \mathbb{R}^{d_l \times d_{l-1}}$

$$\frac{\partial \tilde{x}_t^{(l)}}{\partial W_{ij}^{(l)}} = \frac{\partial}{\partial W_{ij}^{(l)}} \sum_q W_{tq}^{(l)} x_q^{(l-1)} + b_t^{(l)}. \tag{22}$$

This will only be x_j if i = k, otherwise this will be zero. So the matrix result of $\frac{\partial \tilde{x}_t^{(l)}}{\partial W^{(l)}}$ will be a matrix $\in \mathbb{R}^{d_l \times d_{l-1}}$ in which the t'th row will be $\tilde{x}^T$:

$$\begin{pmatrix} \mathbf{0}^T \\ \vdots \\ \mathbf{0}^T \\ \tilde{x}^T \\ \mathbf{0}^T \\ \vdots \\ \mathbf{0}^T \end{pmatrix}$$

And we can stack this matrix d_l times to get a final matrix $\in \mathbb{R}^{d_l \times d_l \times d_{l-1}}$

6:

$$\frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} = \frac{\partial}{b^{(l)}} W^{(l)} x^{(l-1)} + b^{(l)} = \mathbf{1} \tag{23}$$

, which will be a matrix of 1's $\in \mathbb{R}^{d_l \times d_l}$

Question 1.1b)
1:

$$\frac{\partial L}{\partial x^{(N)}} \frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} \tag{24}$$

$$= \begin{pmatrix} -\frac{t_i}{x_i} & , \dots , -\frac{t_n}{x_n} \end{pmatrix} (x_i^{(N)} (\delta_{ij} - x_j^{(N)}) \tag{25}$$

This is the product of a left matrix $\in \mathbb{R}^{1 \times d_N}$ and a right matrix $\in \mathbb{R}^{d_n \times d_n}$ resulting in a final matrix $\in \mathbb{R}^{1 \times d_n}$

2:

$$\frac{\partial L}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial x^{(l)}} \frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}} = \frac{\partial L}{\partial x^{(l)}} diag \left(\begin{cases} 1, & \text{if } \boldsymbol{x}^{l<N} \geq 0 \\ a, & \text{otherwise} \end{cases} \right) \tag{26}$$

3:

$$\frac{\partial L}{\partial x^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} \frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} W^{(l+1)} \tag{27}$$

Which is a product of a left matrix $\in \mathbb{R}^{1 \times d_{l+1}}$ and a right matrix $\in \mathbb{R}^{d_{l+1} \times d_l}$, resulting in a matrix $\in \mathbb{R}^{1 \times d_l}$

4:

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} \tag{28}$$

Which is a product of a left matrix $\in \mathbb{R}^{1 \times d_l}$ and a right matrix $\in \mathbb{R}^{d_l \times d_l \times d_{l-1}}$, resulting in a matrix $\in \mathbb{R}^{1 \times d_l \times d_{l-1}}$

5:

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} \tag{29}$$

$$\tag{30}$$

Here we can leverage the fact that we have zeros in most rows except for 1 per depth matrix. This means that we don't have to actually compute the product with the 3d-tensor because we'll just end up with the outer product of our upstream gradient $\frac{\partial L}{\partial \tilde{x}^{(l)}}$ with the gradient w.r.t the loss, which can be written as a column vector times a row vector, ultimately resulting in:

$$\frac{\partial L}{\partial \tilde{x}^{(l)}}^T \boldsymbol{x}^{l-1} \tag{31}$$

. This is the product of a vector $\in \mathbb{R}^{d_l \times 1}$ and a vector $\in \mathbb{R}^{1 \times d_{l-1}}$ resulting in a matrix $\in \mathbb{R}^{d_l \times d_{l-1}}$ which is the same shape as the weights matrix which is convenient for the gradient update. See also the implementation in modules.py

6:

$$\frac{\partial L}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \cdot \mathbf{1} \tag{32}$$

Which is the product of a left matrix $\in \mathbb{R}^{1 \times d_l}$ and a right matrix $\in \mathbb{R}^{d_l \times d_l}$, which ends up being a matrix $\in \mathbb{R}^{1 \times d_l}$

Question 1.1c) The backpropagation equations will change slightly in the sense that we will often have to deal with an extra dimension for the batch size. This dimension comes into play for all the derivatives with respect to x, $\tilde{x}$ and the bias. This is because the dimensions of these derivatives are a direct result of the input size, which now has an extra dimension (instead of dummy dimension '1' for a single input). This means that we have to be extra careful with tensor products now, because things change slightly in the new case, where old diagonal matrices now become 3d cubes with a diagonal slice filled with the values per input of the batch. The best example of this is $\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} \in \mathbb{R}^{B \times d_N \times d_N}$. In the file modules.py we will see all of these dimensions in action.

1.2 NumPy implementation

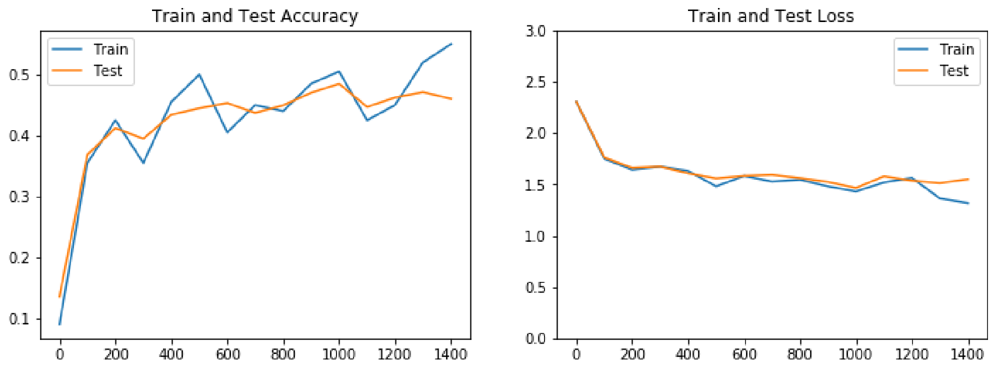

Figure 1: Test and Train accuracies and loss for NumPy MLP

In Figure 1 the train and test set accuracies and respective losses are plotted. In this plot it becomes clear that the NumPy MLP implementation achieves the mentioned 46% accuracy on the test set with the default parameters.

2 PyTorch MLP

Question 2:

In this part of the assignment, a number of different parameter settings were tested on a PyTorch implementation of a multi-layer perceptron. Our goal is to reach at least 52% accuracy on the test set and in the following we will investigate several changes to the parameters that may contribute to achieving this accuracy.

In Figure 2 the train and test accuracy have been plotted alongside the train and test loss curves with the default parameter settings: one hidden layer with 100 neurons, max step size of 1500, learning rate of 0.002 and no dropout. It becomes clear that the MLP performs pretty well, but converges at around 46% accuracy. Apparently, these settings are not expressive enough to obtain the desired 52% accuracy. What could be an improvement?

A possible first improvement could be to increase the maximum number of steps that we take before we stop training. However, the fact that the test accuracy in Figure 2 already seemed to be declining indicated that we may have already been overfitting at this point. Increasing the maximum number of steps wouldn't be of much use in that case, but it could still prove to be an interesting case. In Figure

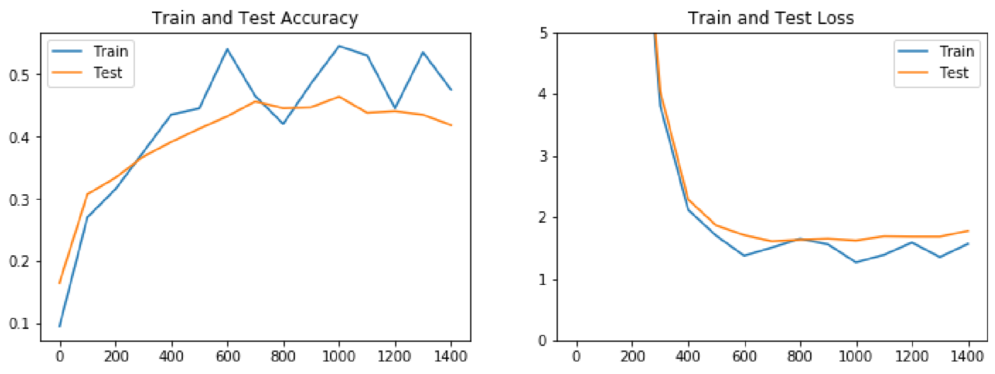

Figure 2: Default Parameters for PyTorch MLP

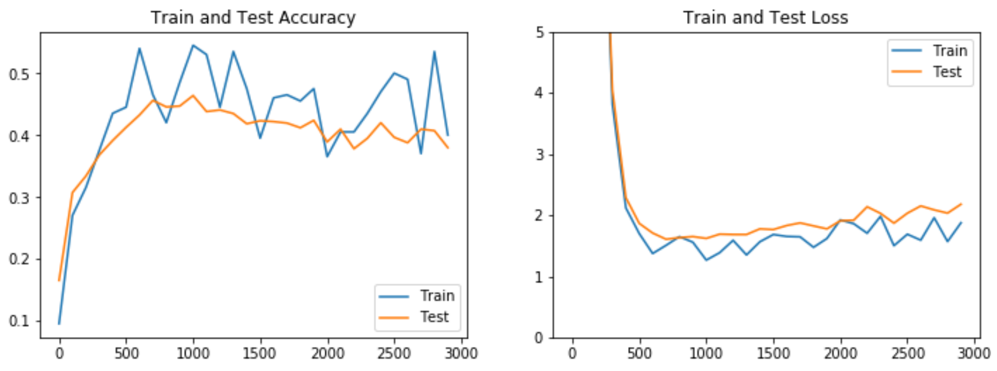

Figure 3: Max Steps changed to 3000

3 the train and test accuracy curves have been plotted alongside their corresponding loss curves. Indeed, this is a clear case of overfitting with a network that is too small to capture more interesting underlying features. This is illustrated by the even further decrease in test set accuracy down to about 40%. Clearly, with a network this small, increasing the maximum number of steps won't be of any help.

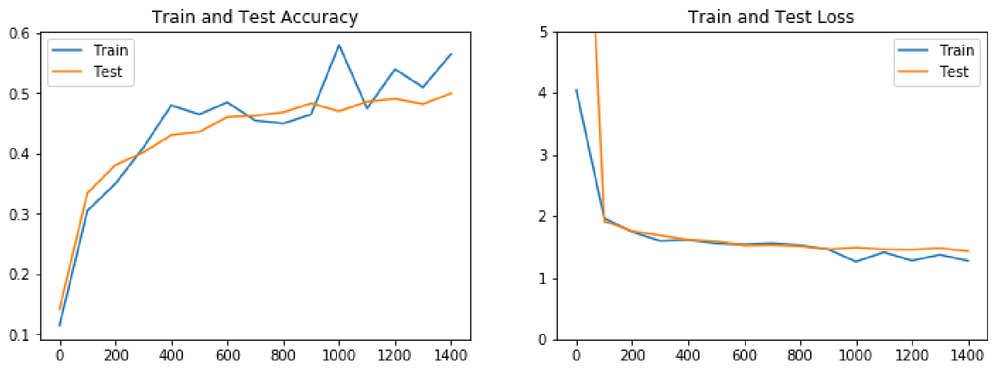

Figure 4: 3 hidden layers of 200, 100, 50 neurons respectively

Seemingly, a wise thing to do next would be to increase the depth of the network. Instead of just one hidden layer of 100 neurons, this can be extended to three hidden layers with 200, 100 and 50 neurons respectively, the results of which have been depicted in Figure 4. The number of max steps was reduced to the default setting of 1500. Perhaps unsurprisingly, the test set accuracy has gone up to almost 50%! Evidently, increasing the depth of the network has helped tremendously in increasing the test set accuracy because the network is now able to fit more underlying features of the dataset. Moreover, it seems like overfitting hasn't set in which would potentially allow for increasing the number of max steps. However, firstly an investigation follows of what a further deepening of the network can bring about.

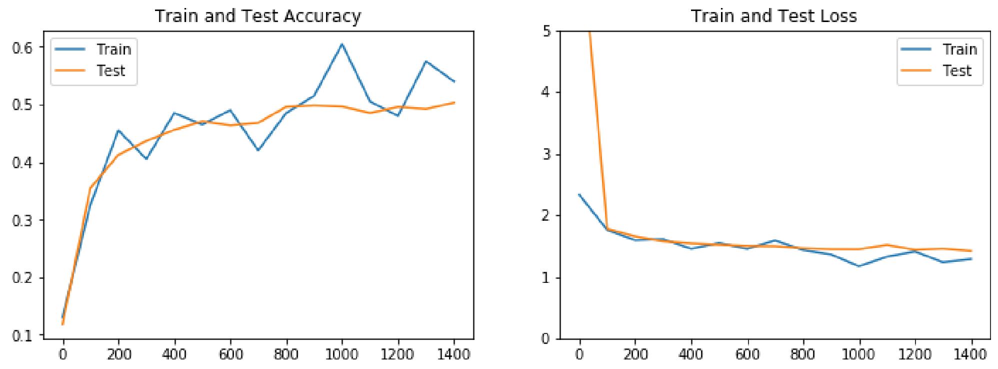

Figure 5: 5 hidden layers of 800, 400, 200, 100, 50 neurons respectively

Consequently, in Figure 5 the amount of hidden layers has been changed from 3 to 5, with 800, 400, 200, 100, 50 neurons respectively. As depicted, the test set accuracy does not increase, pointing to the fact that the further increase in depth does not bring about a significant change in its ability to capture the underlying features inherent to the dataset. Subsequently, it seems futile to merely keep increasing the depth of the network. Rather, a different approach must be found to attain the goal of 52% test set accuracy. A potential solution could be to increase the learning rate, which is investigated next.

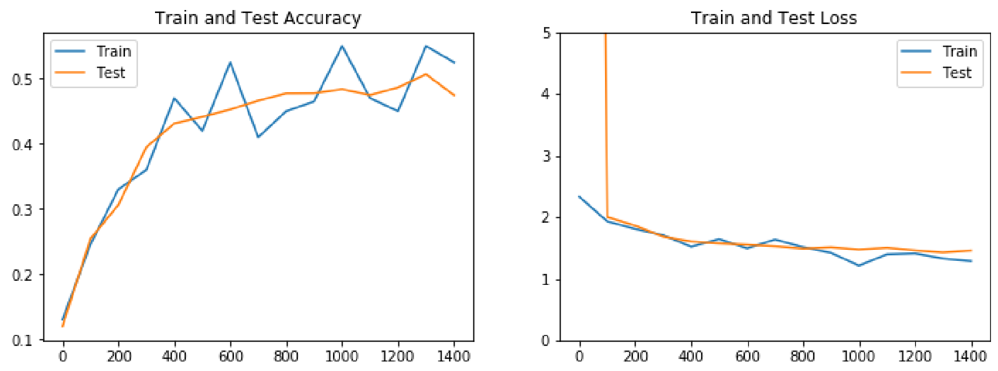

Figure 6: Increased learning rate from 0.002 to 0.005

In Figure 6 the accuracy and loss curves have been plotted for the same network as in Figure 5 with the learning rate increased from 0.002 to 0.005. Unsurprisingly, the train accuracy becomes slightly more squiggly and stochastic due to increase turbulence caused by a higher learning rate. However, the test set accuracy has also gotten worse, corresponding to a flattened loss curve. Apparently, a one-sided increase in learning rate can not explain the network's incapacity to grow to at least a 52% test set accuracy. Then what's left in the arsenal of increasing the test set accuracy might be considered a slight brute-force: increasing the number of units per hidden layer, increasing the number of hidden layers, increasing the maximum number of steps to 3000 to allow for better training of the bigger network, and resetting the learning rate to 0.002.

Finally, the 52% test set accuracy mark has been breached with a total test set accuracy of 52.57%, as depicted in figure 7! Naturally, a deeper network also requires a larger number of maximum steps for it to be able to learn the increased number of parameters in order to capture more of the features underlying the data. Apparently, this combination of an even deeper network with an increased number of maximum steps allowed for a better fit for the dataset. However, the test set accuracy in Figure 5 and Figure 6 seemed to potentially indicate a slight overfitting, given the test set accuracy curve's slight decline near the 3000 step mark. The natural counter to this would be to include a dropout layer as a means of regularization. Will adding a dropout layer further increase the test set performance?

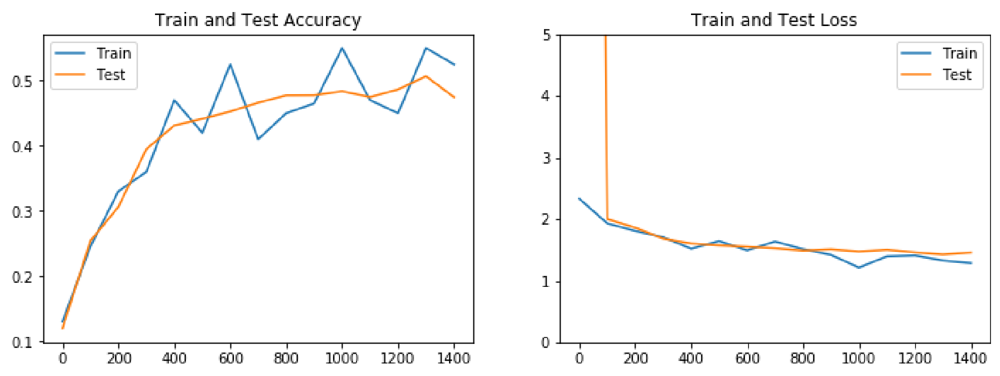

Figure 7: 7 layers of 1600, 800, 400, 200, 100, 50, 25 neurons, max steps 3000 and learning rate 0.002

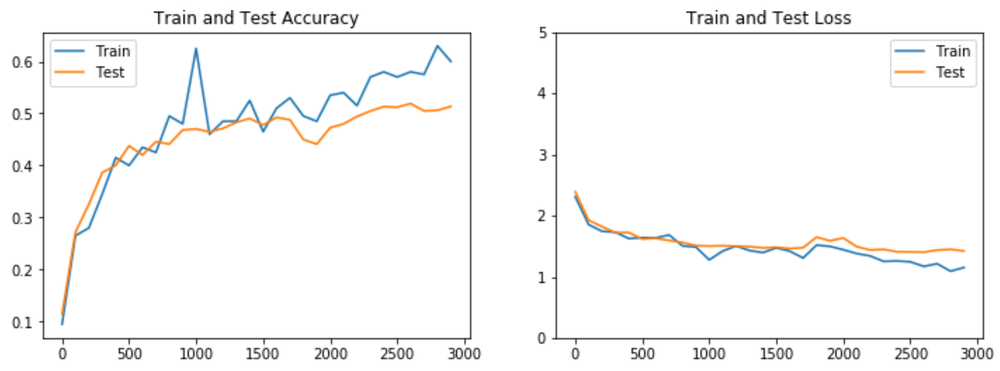

Figure 8: Addition of a dropout layer

Alas, the test set performance got slightly worse after adding a single dropout layer of 20%, as depicted in Figure 8. However, the test accuracy curve shows the effect of adding the dropout layer: there is less sign of overfitting as the train set accuracy and test set accuracy both increase consistently, albeit slightly more gradually. This relatively gradual incline as opposed to the rise in 7 indicates that dropout is doing its work, and may potentially cause this network to surpass that of its predecessor given a larger number of maximum step. This can be grounds for further tinkering.

Concluding, it has become apparent that increasing the depth of the neural network combined with an increase of layer size and an increase in the maximum number of steps taken, allows the network to better capture the intrinsically high-dimensional feature space of pictures, causing it to surpass the desired test set accuracy of 52% due to its increased expressive nature.

3 Custom Module: Batch Normalization

3.1 Automatic Differentiation

See the file custom_batchnorm.py for the implementation.

3.2 Manual implementation of backward pass

Question 3.2a)
1:

$$\left(\frac{\partial L}{\partial \gamma}\right)_j = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \gamma} \tag{33}$$

$$= \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \gamma_j} \tag{34}$$

$$= \sum_s \frac{\partial L}{\partial y_i^s} x_i^s \tag{35}$$

Because $\frac{\partial y_i^s}{\partial \gamma_j}$ is just x_i^s following from the derivative of the batchnorm formula.

2:
Next, we take the derivative w.r.t beta:

$$\left(\frac{\partial L}{\partial \beta}\right) = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \beta} \tag{36}$$

$$= \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \beta_j} \tag{37}$$

$$= \sum_s \frac{\partial L}{\partial y_i^s} \tag{38}$$

Because the derivative w.r.t β will just be an identity matrix.

4 PyTorch CNN

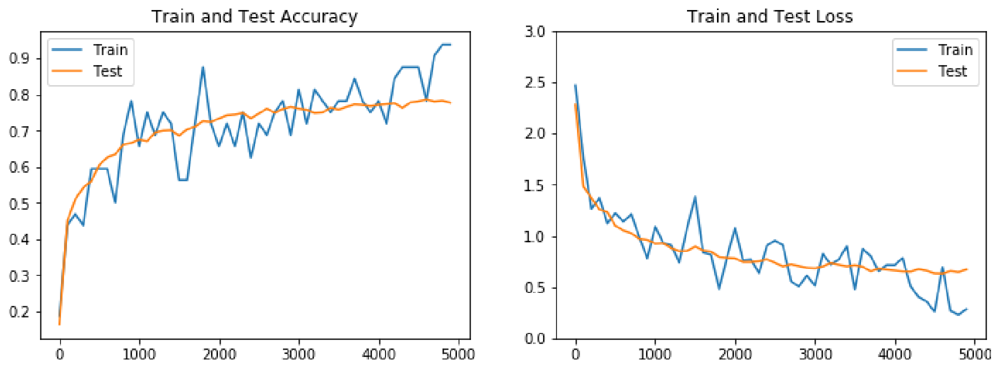

Figure 9: Train and Test Accuracies and Loss for PyTorch CNN

In Figure 9 the train and test set accuracies and the train and test loss for the PyTorch CNN implementation have been plotted. The curves for the train set appear to be quite squiggly, which is explained mostly by the fact that the batch size was only 32. This allows for a lot of variation between how well the model is performing due to the more stochastic nature of dealing with small mini batches. However, it is more important to note the general trend which shows that the train and test set accuracy increase gradually and consistently over the steps.

Moreover, the achieved test set accuracy is far superior to the test set accuracy we were able to attain with the MLP without convolutional layers. The increase was from a mediocre 52% to 78%. This increase is quite impressive, and can be attributed in a large part to the convolutional layers which leaves most of the spatial context intact. In addition, it appears as if the network has not yet come to a point where it is overfitting on the train set, as the test set is still gradually, albeit slowly, increasing, just like the train set. However, it seems to be on the brink of convergence and therefore training it for far longer than the 5000 steps used seems to be redundant.

Additionally, the loss curve shows the same trend as the curves for the train and set accuracies. The train set curve is quite squiggly due to the use of relatively small mini-batches but goes down consistently over the course of the steps. The same trend applies to the test loss, which is less hindered by stochastic processes as it is evaluated on the full test set at each time step. Again it appears as though the network hasn't yet started to overfit, but is definitely very close to convergence given the rapid diminishing returns of the decrease in the test loss.

All in all, it has become clear that Convolutional Neural Networks drastically improve on multi-layer perceptrons for classifying images. A large portion of this improvement can be accredited to leaving the spatial information intact by using convolutions. These assumptions have been corroborated by the experiments which show a large improvement in test set performance.