Assignment 1: MLPs, CNNs and Backpropagation

Jordan Earle
University of Amsterdam
Amsterdam, 1012 WX Amsterdam
`jordanalexearle@gmail.com`

1 MLP backprop and NumPy implementation

1.1 Analytical derivation of gradients

1.1.a Compute the gradients of the modules

First the analytical derivatives off the gradients need to be found.

Starting with $\frac{\partial L}{\partial x^{(N)}}$:

$$\frac{\partial L}{\partial x^{(N)}} = \frac{\partial}{\partial x^{(N)}} - \sum_{i}^{d_N} t_i \log x_i^{(N)} = \left[\frac{\partial L}{\partial x_1^{(N)}}, \frac{\partial L}{\partial x_2^{(N)}}, \cdots, \frac{\partial L}{\partial x_i^{(N)}}, \cdots, \frac{\partial L}{\partial x_{d_N}^{(N)}}\right] \tag{1}$$

The a derivative of a single element in the vector can be expressed as:

$$\frac{\partial L}{\partial x_i^{(N)}} = \frac{\partial}{\partial x_i^{(N)}} - \sum_{j}^{d_N} t_j \log x_j^{(N)} = -t_i \frac{1}{x_i^{(N)}} \tag{2}$$

Therefor the full vector could be represented as:

$$\frac{\partial L}{\partial x^{(N)}} = \frac{\partial}{\partial x^{(N)}} - \sum_{i}^{d_N} t_i \log x_i^{(N)} = \left[-t_1 \frac{1}{x_1^{(N)}}, -t_2 \frac{1}{x_2^{(N)}}, \cdots, -t_i \frac{1}{x_i^{(N)}}, \cdots, -t_{d_N} \frac{1}{x_{d_N}^{(N)}}\right] \tag{3}$$

which can then be expressed in a matrix form in order to ease computational expense as:

$$\frac{\partial L}{\partial x^{(N)}} = -t^T \operatorname{diag}\left(\frac{1}{x^{(N)}}\right) \tag{4}$$

Next the derivative $\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}}$:

$$\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} = \frac{\partial}{\partial \tilde{x}^{(N)}} \frac{\exp(\tilde{x}^{(N)})}{\sum_j^{d_N} \exp(\tilde{x}_j^{(N)})} = \left[\frac{\partial x^{(N)}}{\partial \tilde{x}_1^{(N)}}, \cdots, \frac{\partial x^{(N)}}{\partial \tilde{x}_i^{(N)}}, \cdots, \frac{\partial x^{(N)}}{\partial \tilde{x}_{d_N}^{(N)}}\right] \tag{5}$$

The derivative of a single element in the vector can be expressed as:

$$\frac{\partial x^{(N)}}{\partial \tilde{x}_i^{(N)}} = \frac{\partial}{\partial \tilde{x}_i^{(N)}} \frac{\exp(\tilde{x}^{(N)})}{\sum_j^{d_N} \exp(\tilde{x}_j^{(N)})} = \begin{bmatrix} \frac{\partial \tilde{x}_1^{(N)}}{\partial \tilde{x}_i^{(N)}} \\ \vdots \\ \frac{\partial \tilde{x}_{d_N}^{(N)}}{\partial \tilde{x}_i^{(N)}} \end{bmatrix} \tag{6}$$

A single element of the vector can be expressed as:

$$\frac{\partial x_k^{(N)}}{\partial \tilde{x}_i^{(N)}} = \frac{\partial}{\partial \tilde{x}_i^{(N)}} \frac{\exp(\tilde{x}_k^{(N)})}{\sum_j^{d_N} \exp(\tilde{x}_j^{(N)})} = \frac{g(\tilde{x}_i^{(N)})}{h(\tilde{x}_i^{(N)})} \tag{7}$$

applying the quotient rule to a signal element of the equation, for the case that $k = i$ it can be seen that:

$$\begin{aligned}
\frac{\partial x_k^{(N)}}{\partial \tilde{x}_i^{(N)}} &= \frac{g'(\tilde{x}_i^{(N)})h(\tilde{x}_i^{(N)}) - h'(\tilde{x}_i^{(N)})g(\tilde{x}_i^{(N)})}{h(\tilde{x}_i^{(N)})^2} && (8)\\
&= \frac{\exp \tilde{x}_k^{(N)} \sum_j^{d_N} \exp(\tilde{x}_j^{(N)}) - \exp \tilde{x}_i^{(N)} \exp \tilde{x}_k^{(N)}}{\left(\sum_j^{d_N} \exp(\tilde{x}_j^{(N)})\right)^2} && (9)\\
&= \frac{\exp \tilde{x}_k^{(N)} \sum_j^{d_N} \exp(\tilde{x}_j^{(N)})}{\left(\sum_j^{d_N} \exp(\tilde{x}_j^{(N)})\right)^2} - \frac{\exp \tilde{x}_i^{(N)} \exp \tilde{x}_k^{(N)}}{\left(\sum_j^{d_N} \exp(\tilde{x}_j^{(N)})\right)^2} && (10)\\
&= \operatorname{softmax}(\tilde{x}_k^{(N)}) - \operatorname{softmax}(\tilde{x}_i^{(N)}) \operatorname{softmax}(\tilde{x}_k^{(N)}) && (11)\\
&= x_k^{(N)} - x_i^{(N)} x_k^{(N)} && (12)
\end{aligned}$$

applying the quotient rule to a signal element of the equation, for the case that $k \neq i$ it can be seen that:

$$\begin{aligned}
\frac{\partial x_k^{(N)}}{\partial \tilde{x}_i^{(N)}} &= \frac{g'(\tilde{x}_i^{(N)})h(\tilde{x}_i^{(N)}) - h'(\tilde{x}_i^{(N)})g(\tilde{x}_i^{(N)})}{h(\tilde{x}_i^{(N)})^2} && (13)\\
&= \frac{0 \sum_j^{d_N} \exp(\tilde{x}_j^{(N)}) - \exp \tilde{x}_i^{(N)} \exp \tilde{x}_k^{(N)}}{\left(\sum_j^{d_N} \exp(\tilde{x}_j^{(N)})\right)^2} && (14)\\
&= -\frac{\exp \tilde{x}_i^{(N)} \exp \tilde{x}_k^{(N)}}{\left(\sum_j^{d_N} \exp(\tilde{x}_j^{(N)})\right)^2} && (15)\\
&= -\operatorname{softmax}(\tilde{x}_i^{(N)}) \operatorname{softmax}(\tilde{x}_k^{(N)}) && (16)\\
&= -x_i^{(N)} x_k^{(N)} && (17)
\end{aligned}$$

Therefor, it can be seen that the elements of the matrix can be defined as $\delta_{ik} x_k^{(N)} - x_i^{(N)} x_k^{(N)}$ where δ_{ik} is the indicator function, where if i=k, it returns 1, otherwise its 0. Therefor the derivative will become:

$$\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} = \operatorname{diag}\left(x^{(N)}\right) - x^{(N)} {x^{(N)}}^T \tag{18}$$

Next the derivative $\frac{\partial x^{(l<N)}}{\partial \tilde{x}^{(l<N)}}$:

$$\frac{\partial x^{(l<N)}}{\partial \tilde{x}^{(l<N)}} = \frac{\partial}{\partial \tilde{x}^{(l)}} \operatorname{LeakyReLU}\left(\tilde{x}^{(l)}\right) = \frac{\partial}{\partial \tilde{x}^{(l)}} \max\left(0, \tilde{x}^{(l)}\right) + a \min\left(0, \tilde{x}^{(l)}\right) \tag{19}$$

$$= \left[\frac{\partial x^{(l)}}{\partial \tilde{x}_1^{(l)}}, \cdots, \frac{\partial x^{(l)}}{\partial \tilde{x}_i^{(l)}}, \cdots, \frac{\partial x^{(l)}}{\partial \tilde{x}_{d_l}^{(l)}}\right] \tag{20}$$

The output from the LeakyReLU is:

$$x_i^{(l)} = \begin{cases} \tilde{x}_i^{(l)} & \text{if } \tilde{x}_i^{(l)} > 0 \\ a\tilde{x}_i^{(l)} & \text{if } \tilde{x}_i^{(l)} < 0 \end{cases} \tag{21}$$

Note that at 0, this is undefined. This will cause issues in the derivative, so in order to deal with this, it is assumed for this derivation, at 0, the derivative of the LeakyReLU will be a. Now, examining an individual element of the derivative:

$$\frac{\partial x^{(l)}}{\partial \tilde{x}_i^{(l)}} = \begin{bmatrix} \frac{\partial \tilde{x}_1^{(N)}}{\partial \tilde{x}_i^{(N)}} \\ \vdots \\ \frac{\partial \tilde{x}_{d_N}^{(N)}}{\partial \tilde{x}_i^{(N)}} \end{bmatrix} \tag{22}$$

From this it can be seen that the derivative will be 0 for every element in that vector except the ith element where it will be defined as:

$$\frac{\partial \tilde{x}_i^{(N)}}{\partial \tilde{x}_i^{(N)}} = \begin{cases} 1 & \text{if } \tilde{x}_i^{(l)} > 0 \\ a & \text{if } \tilde{x}_i^{(l)} < 0 \\ a & \text{if } \tilde{x}_i^{(l)} = 0 \end{cases} \tag{23}$$

where as previously stated, due to the equation being undefended at 0, the derivative will be a in this situation.

Therefor the derivative becomes:

$$\frac{\partial x^{(l<N)}}{\partial \tilde{x}^{(l<N)}} = \operatorname{diag}(\delta_{\tilde{x}^{(l<N)}>0} + a\delta_{\tilde{x}^{(l<N)}\leq 0}) \tag{24}$$

Next the derivative $\frac{\partial \tilde{x}^{(l)}}{\partial x^{(l-1)}}$:

$$\frac{\partial \tilde{x}^{(l)}}{\partial x^{(l-1)}} = \frac{\partial}{\partial x^{(l-1)}} W^{(l)} x^{(l-1)} + b^{(l)} = \left[\frac{\partial \tilde{x}^{(l)}}{\partial x_1^{(l-1)}}, \cdots, \frac{\partial \tilde{x}^{(l)}}{\partial x_{d_{(l-1)}}^{(l-1)}}\right] \tag{25}$$

The derivative of a single element in the vector can be expressed as:

$$\frac{\partial \tilde{x}^{(l)}}{\partial x_i^{(l-1)}} = \frac{\partial}{\partial x_i^{(l-1)}} W^{(l)} x^{(l-1)} + b^{(l)} = \begin{bmatrix} \frac{\partial \tilde{x}^{(l)}}{\partial x_i^{(l-1)}} \\ \vdots \\ \frac{\partial \tilde{x}^{(l)}}{\partial x_i^{(l-1)}} \end{bmatrix} \tag{26}$$

A single element of the vector can be expressed as:

$$\frac{\partial \tilde{x}^{(l)}}{\partial x_i^{(l-1)}} = \frac{\partial}{\partial x_i^{(l-1)}} W^{(l)} x^{(l-1)} + b^{(l)} = w_i^{(l)} \tag{27}$$

where $w_i^{(l)}$ is the ith column of the weights matrix of layer l:

$$w_i^{(l)} = \begin{bmatrix} W_{1i}^{(l)} \\ \vdots \\ W_{d_{(l)}}^{l} i \end{bmatrix} \tag{28}$$

Therefor it can be seen that:

$$\frac{\partial \tilde{x}^{(l)}}{\partial x^{(l-1)}} = \frac{\partial}{\partial x^{(l-1)}} W^{(l)} x^{(l-1)} + b^{(l)} = W^{(l)} \tag{29}$$

Next the derivative $\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}}$

$\tilde{x}^{(l)}$ has a shape of $\mathbb{R}^{d_{(l)} \times 1}$ and $W^{(l)}$ has a shape of $\mathbb{R}^{d_{(l)} \times d_{(l)}}$, therefor the derivative w.r.t. matrix $W^{(l)}$ will have the shape $\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} \in \mathbb{R}^{d_{(l)} \times (d_{(l)} \times d_{(l-1)})}$. Looking at one element off the tensor we find that:

$$\left(\frac{\partial \tilde{x}}{\partial W}\right)_{ijk} = \frac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^{(l)}} = \frac{\partial}{\partial W_{jk}^{(l)}} W_i x^{(l-1)} = \frac{\partial}{\partial W_{jk}^{(l)}} \left[W_{i1}^{(l)} x_1^{(l-1)}, \cdots, W_{i(d_{l-1})}^{(l)} x_{(d_{l-1})}^{(l-1)}\right] \tag{30}$$

$$= \begin{cases} 0 & \text{if } i \neq j \\ x_k^{(l-1)} & \text{if } i = j \end{cases} \tag{31}$$

This would therefor make the full matrix:

$$\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} = \begin{bmatrix} \frac{\partial \tilde{x}_1^{(l)}}{\partial W^{(l)}} \\ \vdots \\ \frac{\partial \tilde{x}_i^{(l)}}{\partial W^{(l)}} \\ \vdots \\ \frac{\partial \tilde{x}_{d_{(l)}}^{(l)}}{\partial W^{(l)}} \end{bmatrix} = \begin{bmatrix} \hat{e_1} x^{(l-1)^T} \\ \vdots \\ \hat{e_i} x^{(l-1)^T} \\ \vdots \\ \hat{e_{d_{(l)}}} x^{(l-1)^T} \end{bmatrix} \tag{32}$$

where $x^{(l-1)^T}$ is at the ith row, and $\hat{e_i}$ represents a unit vector with 1 only in the ith location.

Next the derivative $\frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}}$:

$$\frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} = \frac{\partial}{\partial b^{(l)}} W^{(l)} x^{(l-1)} + b^{(l)} = \left[\frac{\partial \tilde{x}^{(l)}}{\partial b_1^{(l)}}, \cdots, \frac{\partial \tilde{x}^{(l)}}{\partial b_i^{(l)}}, \cdots, \frac{\partial \tilde{x}^{(l)}}{\partial b_{d_l}^{(l)}}\right] \tag{33}$$

Taking a single element of this vector we find:

$$\frac{\partial \tilde{x}^{(l)}}{\partial b_i^{(l)}} = \frac{\partial}{\partial b_i^{(l)}} W^{(l)} x^{(l-1)} + b^{(l)} = \frac{\partial}{\partial b_i^{(l)}} b^{(l)} = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \tag{34}$$

where the 1 corresponds to the ith location in the b vector. Therefor the derivative will be:

$$\frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} = I \tag{35}$$

which is the identity matrix of size $d_l \times d_l$

1.1.b Compute the gradients

Beginning with the gradient $\frac{\partial L}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} \frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}}$:

$$\frac{\partial L}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} \frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} \left(\operatorname{diag}\left(x^{(N)} \right) - x^{(N)} {x^{(N)}}^T \right) \tag{36}$$

assuming that t is a one hot vector

Next the gradient $\frac{\partial L}{\partial \tilde{x}^{(l<N)}}$

$$\frac{\partial L}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial x^{(l)}} \frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}} = \frac{\partial L}{\partial x^{(l)}} \operatorname{diag}(\delta_{\tilde{x}^{(l<N)}>0} + a\delta_{\tilde{x}^{(l<N)}\leq 0}) \tag{37}$$

Next the gradient $\frac{\partial L}{\partial x^{(l<N)}}$:

$$\frac{\partial L}{\partial x^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} \frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} W^{(l+1)} \tag{38}$$

Next the gradient $\frac{\partial L}{\partial W^{(l}}$:

$\frac{\partial L}{\partial W^{(l)}} \in \mathbb{R}^{d_{(l)} \times d_{(l-1)}}$, $\frac{\partial L}{\partial \tilde{x}^{(l)}} \in \mathbb{R}^{(1) \times d_{(l)}}$ and $\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} \in \mathbb{R}^{d_{(l)} \times d_{(l)} \times d_{(l-1)}}$ For a single element in the matrix, we see:

$$\begin{aligned} \left(\frac{\partial L}{\partial W^{(l)}} \right)_{jk} = \sum_i^{d_{(l)}} \left(\frac{\partial L}{\partial \tilde{x}^{(l)}} \right)_i \left(\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} \right)_{ijk} &= \sum_i^{d_{(l)}} \frac{\partial L}{\partial \tilde{x}_i^{(l)}} \frac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^{(l)}} && (39) \\ &= \frac{\partial L}{\partial \tilde{x}_j^{(l)}} x_k^{(l-1)} && (40) \\ & && (41) \end{aligned}$$

Iterating over the columns (kth element) it becomes clear that a row of the gradient can be written as:

$$\frac{\partial L}{\partial W^{(l)}}_j = \frac{\partial L}{\partial \tilde{x}_j^{(l)}} {x^{(l-1)}}^T \tag{42}$$

Therefor, the full matrix solution will become:

$$\frac{\partial L}{\partial W^{(l)}} = \left(\frac{\partial L}{\partial \tilde{x}^{(l)}}\right)^T \left(x^{(l-1)}\right)^T \tag{43}$$

Next the gradient $\frac{\partial L}{\partial b^{(l}}$**:**

$$\frac{\partial L}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} I \tag{44}$$

1.1.c

If a batch method (with batch size B) is being used with B vector inputs, instead of the one vector input, the derivatives with respect to inputs (x or $\tilde{x}$), would have an extra dimension.

Additionally, the loss which is propagated backwards will be the average loss rather than the individual. This is because the derivative for the average loss must be first taken w.r.t the individual loss before back propagating the individual. This can be seen in the following:

$$\frac{\partial L_{\text{total}}}{\partial \tilde{x}^{(s,N)}} = \frac{\partial L_{\text{total}}}{L_{\text{individual}}} \frac{\partial L^s_{\text{individal}}}{\partial \tilde{x}^{(s,N)}} = \frac{\partial}{\partial \partial L^s_{\text{individal}}} \frac{1}{B} \sum_{s=1}^{B} L_{\text{individual}} \left(x^{(0),s}, t^s\right) \frac{\partial L^s_{\text{individal}}}{\partial \tilde{x}^{(s,N)}} \tag{45}$$

$$= \frac{1}{B} \frac{\partial L^s_{\text{individal}}}{\partial \tilde{x}^{(s,N)}} \tag{46}$$

1.2 NumPy Implementation

Using the above derivatives, a MLP was created, using the settings provided in the assignment. Model consisted of a linear layer, leaky relu layer, linear layer, and a softmax layer. The loss and accuracy of the model can be seen in figure 1. When running the training, it was confirmed that an accuracy of 0.46 was achieved for the entire test set adn this can be seen in the figure.

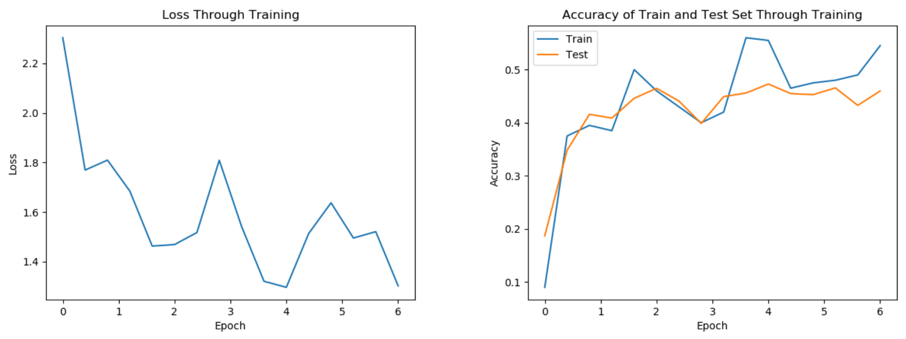

Figure 1: Numpy implementation of the MLP, using SGD, with a batch size of 200, learning rate of 2e-3 and a neg slope for the LeakyRELU of 0.02 over 1500 batches. Left shows the loss of the training set through the simulation, while the right shows the accuracy of the test and training set

The loss figure shows a decrease in loss over the training, which is expected as we approach a local optimum, this is also reflected in the accuracy, as the accuracy increases the local optimum is approached. The test accuracy can also be seen to be increasing which is expected unless the model begins to over fit. At this point in the training, it appears that the system isn't over-fitting yet, as the test is still increasing. Increasing the number of epochs used (adding more steps of batch training) could yield a better solution still. In addition, it can be seen that the plots are very jagged. This could be due to a too high of learning rate. It is possible that implementing a decay factor or decreasing the initial value of the learning rate will help smooth the plot, as the changes in weights being taken will not be as large. It is possible, looking at the graphs, that the MLP is hopping around an optimum it isn't able to reach, due to the learning rate size.

2 PyTorch MLP

When implementing pytorch, first the previous network which was created in numpy was recreated. These can be seen in figure 2. The max accuracy seen in the default settings was 0.44 in this case, which could be due to the seed. From the figure it can be seen that it follows a similar path for the accuracy as the manually implemented modules, but the loss drops significantly faster. This is likely due to implementation change in pytorch that was not taken into consideration.

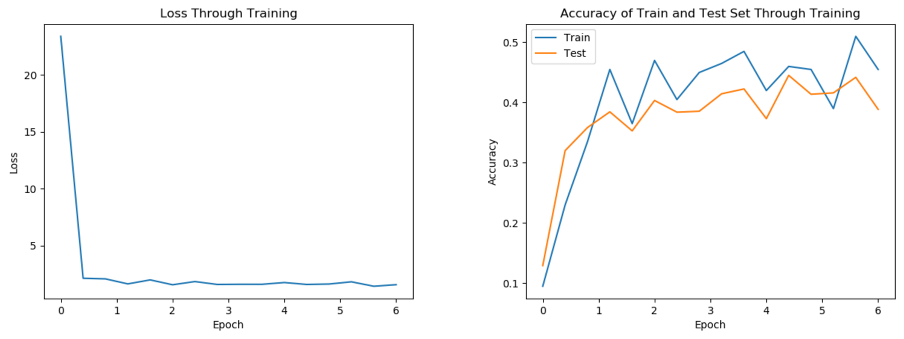

Figure 2: PyTorch implementation of the MLP, using SGD, with a batch size of 200, learning rate of 2e-3 and a neg slope for the LeakyRELU of 0.02 over 1500 batches (steps). Left shows the loss of the training set through the simulation, while the right shows the accuracy of the test and training set

Once it was confirmed that the model performed as expected, modifications were made in an attempt to get the model up to a minimum of 0.52 accuracy. First a simple weight regularization was implemented into the model. This did not increase performance and was not used again as later weight decay was found and implemented instead. Next the optimizer was changed from SGD to Adam, to see if the other optimizer would perform better. It was the case that it did, converging much quicker to a solution, with a slight increase in accuracy (0.003). Since this was not enough, it was decided that the model itself should be changed. In order to do this, the first thing explored was adding more layers, and changing the layers size, making them go from larger to smaller, as we discussed in class. Initially only 2 additional sets of linear and leaky ReLU modules were added. The sizes used for the layers were $[600, 500, 275, 10]$ This gave rise to an increase in performance, from 0.44 to 0.501. Since this worked well, it was decided to add another 2 sets of linear and leaky ReLU layers, with sizes of the layers $[600, 500, 388, 275, 163, 10]$. This change gave a slight increase to an accuracy of 0.508. Next weight decay was added, as from research online it seemed that this was an effective way of regularizing the weights. This was initially implanted with a weight decay of 1e-6 with 4 additional sets of modules. This increased the accuracy again to 0.5113. Since these simple models were close to achieving the desired value of 0.52, and due to time restrictions, a grid search of the parameters of the number of layers, learning weight, and weight decay was undertaken. The values used and the resulting accuracy's can be seen in tables 1.

From the tables it can be seen that MLP size 11 outperformed the size 7, and that the highest accuracy occurred with a learning rate of 1e-4. Change the weight decay slightly changed the accuracy, but they were all within 0.004 of each other, which would indicate that the weight decay could be further searched. Figure 3 shows the loss and accuracy over training of the a layer MLP with a learning rate of 1e-4 and a weight decay of 5e-4 over 16 epochs. From the graph it can be seen that the training has mostly completed by the 3rd epoch. After this point, it can be seen that the model is beginning to over-fit to the data, but it is still increasing the accuracy some so it was allowed to continue. Early stopping could have been implemented in order to prevent the long train time, but it was left running to see just how high the accuracy could become. With these settings an accuracy of 0.55 was for the test set during the training.

MLP Size 7		
Learning Rate	**Weight Decay**	**Accuracy**
1e-2	1e-4	0.5243
	5e-4	0.5162
	1e-5	0.5211
	5e-5	0.5154
1.5e-3	1e-4	0.5111
	5e-4	0.5134
	1e-5	0.5151
	5e-5	0.5233
1.25e-3	1e-4	0.5131
	5e-4	0.5194
	1e-5	0.5133
	5e-5	0.5172
1e-3	1e-4	0.5188
	5e-4	0.5232
	1e-5	0.5234
	5e-5	0.5187
1e-4	1e-4	0.5197
	5e-4	0.5181
	1e-5	0.5135
	5e-5	0.5107

MLP Size 11		
Learning Rate	**Weight Decay**	**Accuracy**
1e-2	1e-4	0.47
	5e-4	0.459
	1e-5	0.3913
	5e-5	0.3138
1.5e-3	1e-4	0.5242
	5e-4	0.5073
	1e-5	0.5261
	5e-5	0.5289
1.25e-3	1e-4	0.5335
	5e-4	0.524
	1e-5	0.5399
	5e-5	0.5372
1e-3	1e-4	0.5389
	5e-4	0.5231
	1e-5	0.5382
	5e-5	0.5354
1e-4	1e-4	0.5434
	5e-4	0.5448
	1e-5	0.5404
	5e-5	0.5416

Table 1: Grid search over MLP size (7,11), Learning Rate (1e-2, 1.5e-3, 1.25e-3, 1e-3 , 1e-4), and Weight Decay (1e-4, 5e-4, 1e-5, 5e-5) with 2500 steps (10 epochs) for PyTorch MLP to achieve minimum 0.52 accuracy.

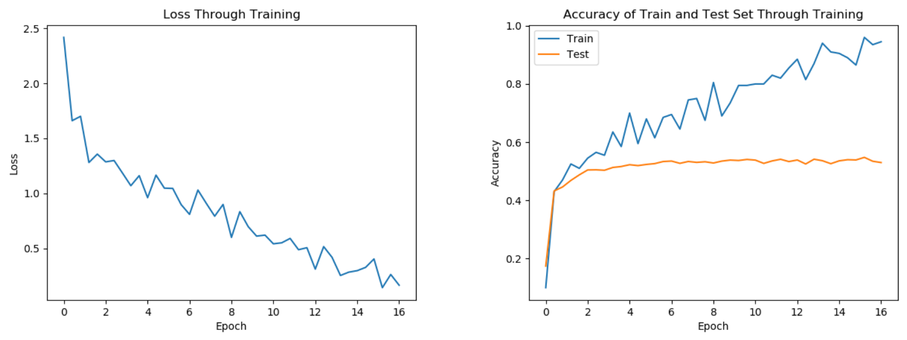

Figure 3: PyTorch implementation of the MLP, using ADAM optimizer, with a batch size of 200, learning rate of 1e-4, weight decay of 5e-4 and a neg slope for the LeakyRELU of 0.02 over 16 epochs. Left shows the loss of the training set through the simulation, while the right shows the accuracy of the test and training set

If there was more time, it would be nice to search in even lower learning rates, as the accuracy is still increasing as the learning rate is lowered. Finally, dropout was discussed in class and it would be of interest to see if they added to the accuracy if added to the model, but due to time constraints this was not feasible at this time.

3 Custom Module: Batch Normalization

3.1 Automatic differentiation

To see how effective the custom batch norm module which takes advantage of autograd was, it was added into the 11 layer model which used the basic parameters from 2. This added an additional 5 layers to the model added before the hidden linear layers. The results can be seen in figure 4. The addition of this layer increased the accuracy from 0.511 to 0.55, with no additional parameter tuning.

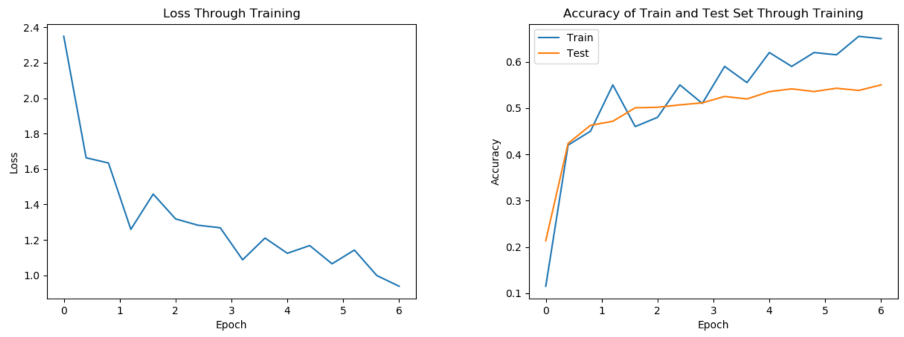

Figure 4: Pytorch implementation of the MLP with autograd custom batch norm layers added before the hidden linear layers, using ADAM, with a batch size of 200, learning rate of 2e-3 and a neg slope for the LeakyRELU of 0.02 over 1500 batches. Left shows the loss of the training set through the simulation, while the right shows the accuracy of the test and training set

3.2 Manual implementation of backwards pass

First before we manually implement the backwards pass, we must first take the derivatives. Starting with $\frac{\partial L}{\partial \gamma}$:

$$\left(\frac{\partial L}{\partial \gamma}\right)_j = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial \gamma_j} = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial}{\partial \gamma_j} \gamma_i \hat{x}_i^s + \beta_i = \sum_s \frac{\partial L}{\partial y_j^s} \hat{x}_j^s \tag{47}$$

Then we find $\frac{\partial L}{\partial \beta}$:

$$\left(\frac{\partial L}{\partial \beta}\right)_j = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial \beta_i} = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial}{\partial \beta_i} \gamma_i \hat{x}_i^s + \beta_i = \sum_s \frac{\partial L}{\partial y_j^s} 1 = \sum_s \frac{\partial L}{\partial y_j^s} \tag{48}$$

Finally we examine $\frac{\partial L}{\partial x}$. The derivative w.r.t. the jth component of x for the rth sample can be expressed as:

$$\left(\frac{\partial L}{\partial x}\right)_i^r = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial x_j^r} \tag{49}$$

$$= \sum_s \sum_i \frac{\partial L}{\partial \hat{x}_i^s} \frac{\partial \hat{x}_i^s}{\partial x_j^r} + \frac{\partial L}{\partial \mu_i} \frac{\partial \mu_i}{\partial x_j^r} + \frac{\partial L}{\partial {\sigma_i}^2} \frac{\partial {\sigma_i}^2}{\partial x_j^r} \tag{50}$$

Looking at these equations it can be seen that if i is not equal to j, the derivative will be 0. Therefor we will be examining the cases where $i = j$. In order to get this, first we must take the derivative w.r.t $\hat{x}$:

$$\left(\frac{\partial L}{\partial \hat{x}}\right)_j^r = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial \hat{x}_j^r} = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial}{\partial \hat{x}_j^r} \gamma_i \hat{x}_i^s + \beta_i = \delta_{i=j} \delta_{r=s} \frac{\partial L}{\partial y_j^r} \gamma_j \tag{51}$$

Then the derivative $\frac{\partial \hat{x}}{\partial x}$:

$$\left(\frac{\partial \hat{x}}{\partial x}\right)_j^r = \frac{\partial}{\partial x_i^r} \frac{x_i^s - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} = \delta_{i=j} \delta_{r=s} \frac{1}{\sqrt{\sigma_j^2 + \epsilon}} \tag{52}$$

next we examine $\frac{\partial L}{\partial \mu}$

$$\frac{\partial L}{\partial \mu_i} = \frac{\partial L}{\partial \hat{x}} \frac{\partial \hat{x}}{\partial \mu_i} + \frac{\partial L}{\partial \sigma_i^2} \frac{\partial \sigma_i^2}{\partial \mu_i} \tag{53}$$

now $\frac{\partial \hat{x}}{\partial \mu}$:

$$\frac{\partial \hat{x}_i}{\partial \mu_i} = \frac{\partial}{\partial \mu_i} \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} = \frac{-1}{\sqrt{\sigma_i^2 + \epsilon}} \tag{54}$$

next $\frac{\partial \sigma^2}{\partial \mu}$:

$$\frac{\partial \sigma^2}{\partial \mu} = \frac{\partial}{\partial \sigma^2} \frac{1}{B} \sum_{s=1}^{B} \left(x_i^s - \mu_i\right)^2 = \frac{1}{B} \sum_{i=1}^{B} -2\left(x_i - \mu_i\right) \tag{55}$$

next $\frac{\partial L}{\partial \sigma^2}$:

$$\frac{\partial L}{\partial \sigma_i^2} = \frac{\partial L}{\partial \hat{x}_i} \frac{\partial \hat{x}_i}{\partial \sigma_i^2} \tag{56}$$

where:

$$\frac{\partial \hat{x}_i}{\partial \sigma_i^2} = \sum_{i=1}^{B} \left(x_i - \mu_i\right) \cdot (-0.5) \cdot \left(\sigma_i^2 + \epsilon\right)^{-1.5} \tag{57}$$

Combining the derivatives seen and after some manipulation we find that:

$$\frac{\partial L}{\partial \mu_i} = \sum_{s=1}^{B} \frac{\partial L}{\partial \hat{x}_i^s} \cdot \frac{-1}{\sqrt{\sigma_i^2 + \epsilon}} \tag{58}$$

There are 2 derivative missing now, $\frac{\partial \mu_i}{\partial x_j^r}$ and $\frac{\partial \sigma_i^2}{\partial x_j^r}$:

$$\frac{\partial \mu_i}{\partial x_j^r} = \frac{\partial}{\partial x_j^r} \frac{1}{B} \sum_{s=1}^{B} x_i^s = \delta_{i=j} \frac{1}{B} \tag{59}$$

and

$$\frac{\partial \sigma_i^2}{\partial x_j^r} = \frac{\partial}{\partial x_j^r} \frac{1}{B} \sum_{s=1}^{B} \left(x_i^s - \mu_i\right)^2 = \delta_{i=j} \frac{2\left(x_i - \mu\right)}{B} \tag{60}$$

Combining all of the derivatives and we find:

$$\left(\frac{\partial L}{\partial x}\right)_i^r = \sum_s \sum_i \frac{\partial L}{\partial \hat{x}_i^s} \frac{\partial \hat{x}_i^s}{\partial x_j^r} + \frac{\partial L}{\partial \mu_i} \frac{\partial \mu_i}{\partial x_j^r} + \frac{\partial L}{\partial {\sigma_i}^2} \frac{\partial {\sigma_i}^2}{\partial x_j^r} \tag{61}$$

$$= \frac{\partial L}{\partial \hat{x}_i^s} \frac{1}{\sqrt{\sigma_j^2 + \epsilon}} + \sum_{s=1}^{B} \frac{\partial L}{\partial \hat{x}_i^s} \cdot \frac{-1}{\sqrt{\sigma_i^2 + \epsilon}} \left(\delta_{i=j} \frac{1}{B}\right) + \frac{\partial L}{\partial \hat{x}_i^s} \sum_{i=1}^{B} \left(x_i - \mu_i\right) \cdot (-0.5) \cdot \left(\sigma_i^2 + \epsilon\right)^{-1.5} \delta_{i=j} \frac{2\left(x_i - \mu\right)}{B} \tag{62}$$

This then simplifies to:

$$\left(\frac{\partial L}{\partial x}\right)_i^r = \frac{\left(\sigma_i^2 + \epsilon\right)^{-0.5}}{B}\left(B\frac{\partial L}{\partial \hat{x}_i^r} - \sum_{j=1}^{B}\frac{\partial L}{\partial \hat{x}_j^r} - \hat{x}_i^r \sum_{j=1}^{B}\frac{\partial L}{\partial \hat{x}_j^r}\hat{x}_j^r\right) \tag{63}$$

These derivatives were then implemented in the specified locations in the custom_batchnorm.py file.

3.3 Manual implementation of backwards pass

To see how effective the custom batch norm module with the manual forward and backward propagation was, it was added into the 11 layer model which used the basic parameters from 2. This added an additional 5 layers to the model. The results can be seen in figure 5. The addition of this layer increased the accuracy from 0.511 to 0.544, with no additional parameter tuning. This custom module appears to perform comparably with the one using pytorch's autograd. Since it performed slightly worse, it was decided that it should be run again, to check if it was a product of the starting location. When running both again, they appeared to both have values within 0.54-0.56. Therefor, the slight change in performance is likely due to the starting location.

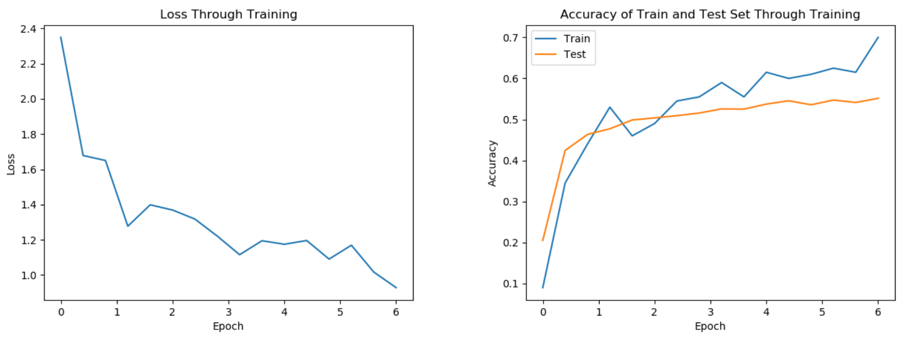

Figure 5: Pytorch implementation of the MLP with manual custom batch forward and backward norm layers added before the hidden linear layers, using ADAM, with a batch size of 200, learning rate of 2e-3 and a neg slope for the LeakyRELU of 0.02 over 1500 batches. Left shows the loss of the training set through the simulation, while the right shows the accuracy of the test and training set

4 PyTorch CNN

In this part of the assignment a CNN was created as described in figure 6 which was taken from the assignment, using an ADAM optimizer. This was done as in image classification and processing, spatial information is important and can increase the predictive power of the model. In this model, all of the conv blocks consist of a 2D-convolutional layer, followed by a batch normalization and ReLU layer.

The loss and accuracy through the training can be seen in figure7. From the figure it can be seen that using the CNN, by half an epoch the network has already gotten to the accuracy of the previous MLP models. The loss is trending down as usual, and interestingly, the accuracy of both the test and train are rising at approximately the same rate. This could be due to the features being found being more universal/useful for images and applying better to the test set. All of these results indicated that the assumption that spatial information is important in image classification is a valid one.

Once final addition to the experiment was made, and that was adding another 2 layers before the last linear layer, a batch normalization and a ReLu layer. This was done as previously there was a Leaky ReLu layer before the last linear layer so the idea was that it would increase the performance, and since in this model, a batch normalization as added before the ReLu, that was also added. The results can be seen in figure 8. It can be seen that there was no notable rise in predictive power. This is likely due to the previous layer being a max pool, which would make sense since the model itself doesn't add them after the maxpools.

Name	Kernel	Stride	Padding	Channels In/Out
conv1	3×3	1	1	3/64
maxpool1	3×3	2	1	64/64
conv2	3×3	1	1	64/128
maxpool2	3×3	2	1	128/128
conv3_a	3×3	1	1	128/256
conv3_b	3×3	1	1	256/256
maxpool3	3×3	2	1	256/256
conv4_a	3×3	1	1	256/512
conv4_b	3×3	1	1	512/512
maxpool4	3×3	2	1	512/512
conv5_a	3×3	1	1	512/512
conv5_b	3×3	1	1	512/512
maxpool5	3×3	2	1	512/512
linear	–	-	-	512/10

Figure 6: CNN specifications as given in the assignment. All of the conv blocks consist of a 2D-convolutional layer, followed by a batch normalization and ReLU layer.

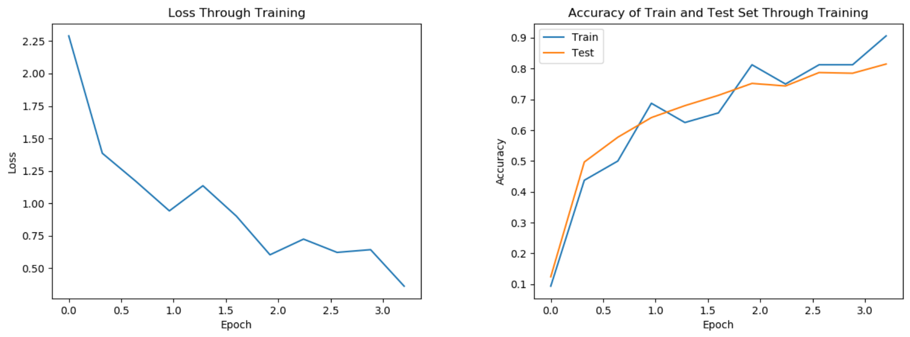

Figure 7: PyTorch implementation of the CNN as described in the assignment. Left shows the loss of the training set through the simulation, while the right shows the accuracy of the test and training set

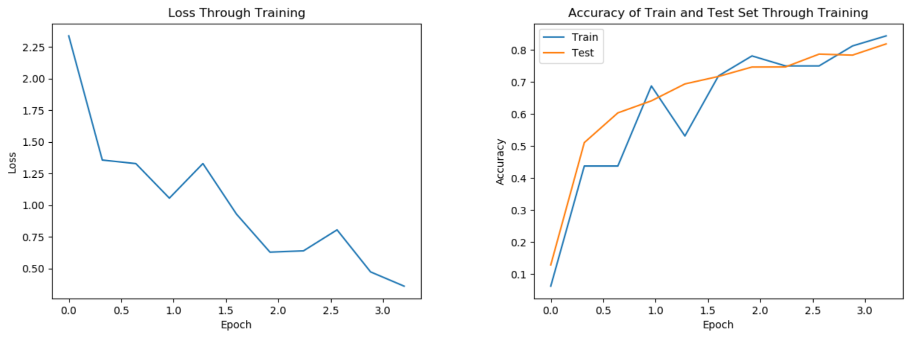

Figure 8: PyTorch implementation of the CNN as described in the assignment. Left shows the loss of the training set through the simulation, while the right shows the accuracy of the test and training set