Deep Learning Practical 1

Longxiang Wang
Master of Artificial Intelligence
University of Amsterdam
wlongxiang1119@gmail.com

1 MLP backprop and NumPy implementation

1.1 Analytical derivation of gradients

1.1 a)

cross entropy loss

For the cross entropy loss module:

$$L(x^{(N)}, t) = -\sum_i t_i log x_i^{(N)} \longrightarrow \frac{\partial L}{\partial x_i^{(N)}} = \begin{cases} -\frac{1}{x_i^{(N)}}, & \text{if } i = argmax(t) \\ 0, & \text{if } i \neq argmax(t) \end{cases} \tag{1}$$

In other words, the categorical cross entropy loss derivative w.r.t, to the final output is zero for all positions when one-hot encoded values are zero.

In matrix or vector form:

$$\frac{\partial L}{\partial x^{(N)}} = -\frac{1}{x^{(N)}} \odot t \tag{2}$$

where $\odot$ denotes element-wise multiplication.

softmax module

For the softmax module, we have:

$$x_i^{(N)} = \frac{exp(\tilde{x}_i^{(N)})}{\sum_{k=1}^{d_N} exp(\tilde{x}_k^{(N)})} \tag{3}$$

Therefore, if $i \neq j$:

$$\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = -\frac{exp(\tilde{x}_i^{(N)}) exp(\tilde{x}_j^{(N)})}{\Big(\sum_{k=1}^{d_N} exp(\tilde{x}_k^{(N)})\Big)^2} = -x_i^{(N)} x_j^{(N)} \tag{4}$$

If $i = j$:

$$\begin{aligned} \frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = \frac{exp(\tilde{x}_i^{(N)}) \sum_{k=1}^{d_N} exp(\tilde{x}_k^{(N)}) - exp(\tilde{x}_j^{(N)}) exp(\tilde{x}_i^{(N)})}{\Big(\sum_{k=1}^{d_N} exp(\tilde{x}_k^{(N)})\Big)^2} &= x_i^{(N)} - x_i^{(N)} x_j^{(N)} \\ &= x_i^{(N)}\big(1 - x_i^{(N)}\big) \end{aligned} \tag{5}$$

In matrix or vector from we can rewrite the result as:

$$\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} = diag(x^{(N)}) - x^{(N)} x^{(N)^T} \tag{6}$$

Leaky ReLu Activation

For the hidden layer activation function, a.k.a. leaky ReLu function, each function output is independent from each other, i.e., unlike the softmax module, here the Jacobian is a diagonal matrix.

If $i \neq j$:

$$\frac{\partial x_i^{(l<N)}}{\partial \tilde{x}_j^{(l<N)}} = 0 \tag{7}$$

If $i = j$:

$$\frac{\partial x_i^{(l<N)}}{\partial \tilde{x}_j^{(l<N)}} = \begin{cases} 1, & \text{if } \tilde{x}_j^{(l<N)} \geq 0 \\ a, & \text{if } \tilde{x}_j^{(l<N)} < 0 \end{cases} \tag{8}$$

In matrix form, we can rewrite it into:

$$\frac{\partial x^{(l<N)}}{\partial \tilde{x}^{(l<N)}} = diag(a^{[\tilde{x}_j^{(l<N)}<0]}) \tag{9}$$

where $[\tilde{x}_j^{(l<N)} < 0]$ denote a vector whose element is 1 is condition $\tilde{x}_j^{(l<N)} < 0$ is met, 0 otherwise.

Linear Weights Module

For the linear module, we have:

$$\tilde{x}^{(l)} = W^{(l)} x^{(l-1)} + b^{(l)} \tag{10}$$

Now we can solve the derivative w.r.t. the output of previous layer:

$$\frac{\partial \tilde{x}_i^{(l)}}{\partial x_j^{(l-1)}} = w_{ij}^{(l)} \tag{11}$$

where w_{ij} denotes the weight of incoming connection from j-th neuron in previous layer $(l-1)$ to i-th neuron in current layer (l).

In matrix form:

$$\frac{\partial \tilde{x}^{(l)}}{\partial x^{(l-1)}} = W^{(l)} \tag{12}$$

where $W^{(l)}$ denotes the matrix of weights sending connections from previous layer.

The derivative w.r.t. the weights:

$$\frac{\partial \tilde{x}_i^{(l)}}{\partial w_{jk}^{(l)}} = \begin{cases} x_k^{(l-1)}, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases} \tag{13}$$

Therefore, in its matrix form, $\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}}$ is a three-dimensional tensor where the dialognal elements are vectors of input from previous layer.

Similarly, we can have the derivative w.r.t. to the bias vector:

$$\frac{\partial \tilde{x}_i^{(l)}}{\partial b_j^{(l)}} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases} \tag{14}$$

In matrix form:

$$\frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} = \mathcal{I} \tag{15}$$

where $\mathcal{I}$ denotes identity matrix.

1.1 b)

Here in this question, we calculate the backprop required gradients.

softmax module

Using chain rule, we can plug in the solutions derivated in previous section to get the matrix form of the backward gradients:

$$\frac{\partial L}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} \frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} = \Big(\frac{1}{x^{(N)}} \odot t\Big)\Big(diag(x^{(N)}) - x^{(N)} {x^{(N)}}^T\Big) \tag{16}$$

Note that in practice activations like $x^{(N)}$ is cached during forward pass, thus we can easily calculate the backward derivative.

Leaky ReLu module

$$\frac{\partial L}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial x^{(l)}} \frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}} = \frac{\partial L}{\partial x^{(l)}} diag(a^{[\tilde{x}_j^{(l<N)}<0]}) \tag{17}$$

Note that we can also rewrite the dot prododuct with a diagonal matrix into an element-wise multiplication (this will make vectorized computation in numpy easier):

$$\frac{\partial L}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial x^{(l)}} diag(a^{[\tilde{x}_j^{(l<N)}<0]}) = \frac{\partial L}{\partial x^{(l)}} \odot a^{[\tilde{x}_j^{(l<N)}<0]} \tag{18}$$

linear weights module

The derivative of loss function w.r.t. the hidden layer output is:

$$\frac{\partial L}{\partial x^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} \frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} W^{(l+1)} \tag{19}$$

The derivative of loss function w.r.t. the weights of hidden layer connections is:

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} \tag{20}$$

Due to the fact that $\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}}$ is a tensor, it is easier to write this derivative in index notation:

$$(\frac{\partial L}{\partial W^{(l)}})_{ij} = \frac{\partial L}{\partial \tilde{x}_i^{(l)}} \frac{\partial \tilde{x}_i^{(l)}}{\partial {W_{ij}}^{(l)}} = \frac{\partial L}{\partial \tilde{x}_i^{(l)}} x_j^{(l-1)} \tag{21}$$

Therefore, we can write it in a matrix form as following:

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \Big(x^{(l-1)}\Big)^T \tag{22}$$

As we seen before, the derivative of bias is an identity matrix, so the derivative of loss function w.r.t. the biases of hidden layer connections is:

$$\frac{\partial L}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \tag{23}$$

1.1 c)

In batch propagation, when $B \neq 1$, compared to single sample stochastic case, the main difference is that we need to do the forward pass and backprop for all B samples.

Now the toal loss is defined as the mean of losses of single samples:

$$L_{total}(W, b) = \frac{1}{B} \sum_{i=1}^{B} L_i(W, b) \tag{24}$$

When calculating the derivative w.r.t. to weights and bias, we need to calculate the derivate for each sample, then get the average gradient for all B samples. That is to say, we update the weights using following when doing batch SGD:

$$W^{\tau+1} = W^{\tau} - \eta \frac{1}{B} \sum_{i=1}^{B} \frac{\partial L_i(W, b)}{\partial W} \tag{25}$$

1.2 Numpy Implementation

The numpy implementation can be see in the code. The loss and accuracy of this implemtation is show in Figure 1. The final test accuracy after 1.5k iterations is around 0.466 with the default settings.

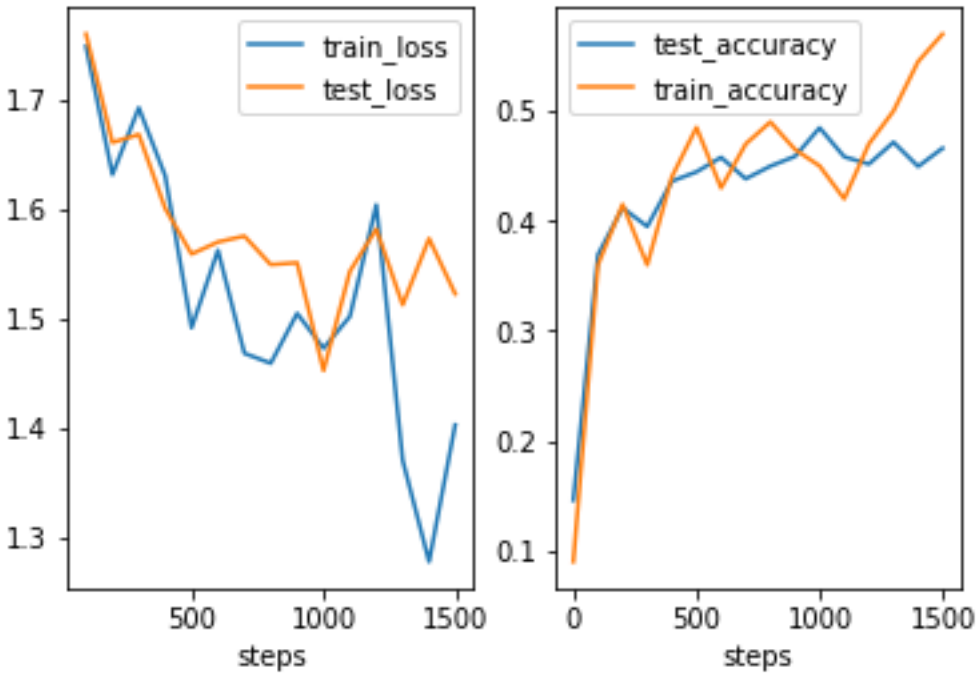

Figure 1: Test accuracy of 0.466. Loss and accuracy of default Pytorch implementation of MLP with parameters: $lr = 0.002, steps = 1500, batch\_size = 200, leaky\_relu\_slope = 0.02, hidden\_units =$ "100"

2 PyTorch MLP

The Pytorch implementation is in the code.

With the default parameter settings as in numpy implementation, the achieved test accuracy is around 0.438. The result is shown in 2. The performance is slightly lower than what is achieved in numpy implementation. One possible explanation is that Pytorch uses a different weight initialization methods.

A grid search using different combinations of parameters is used. It is observed that deeper and wider network is the most effective. Higher batch size helps to converge faster but also requires more computing resources.

The best test accuracy 0.525 is achieved by a grid search hyper parameter tuning as shown in Figure 3.

3 Batch Normalization

3.1 Automatic Differentiation

Using PyTorch nn.Module enables us to do backprop automatically. We just need to register parameters with nn.Parameters API, then the backward method (implicitly inherited from nn.Module) can recognize where to implement auto differentiation ; and implement the forward method for the function of interest.

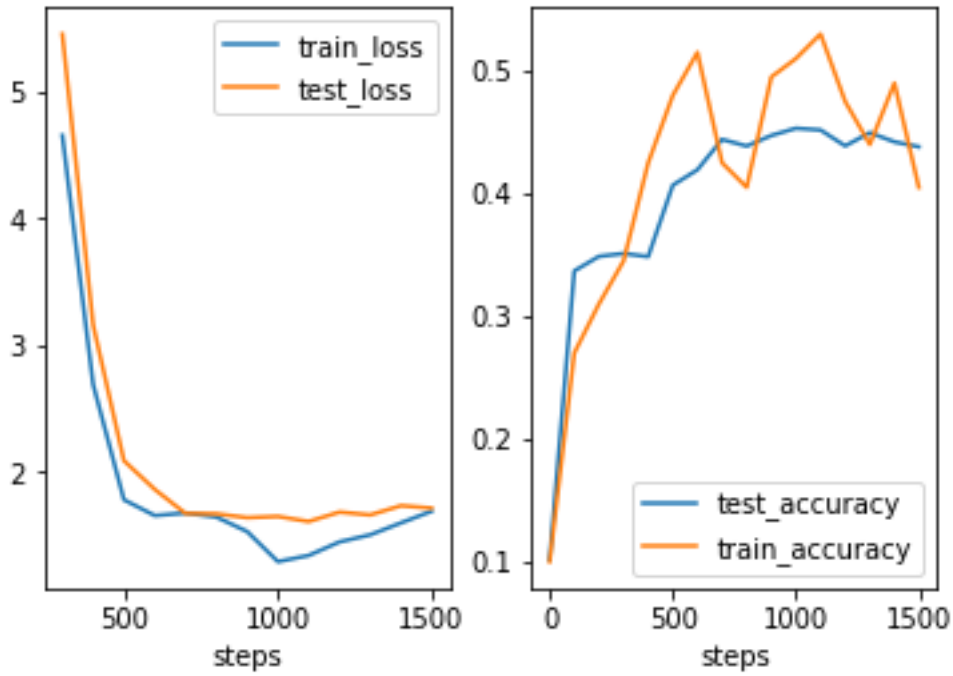

Figure 2: Test accuracy of 0.438. Loss and accuracy of default Pytorch implementation of MLP with parameters: $lr = 0.002, steps = 1500, batch\_size = 200, leaky\_relu\_slope = 0.02, hidden\_units =$ "100"

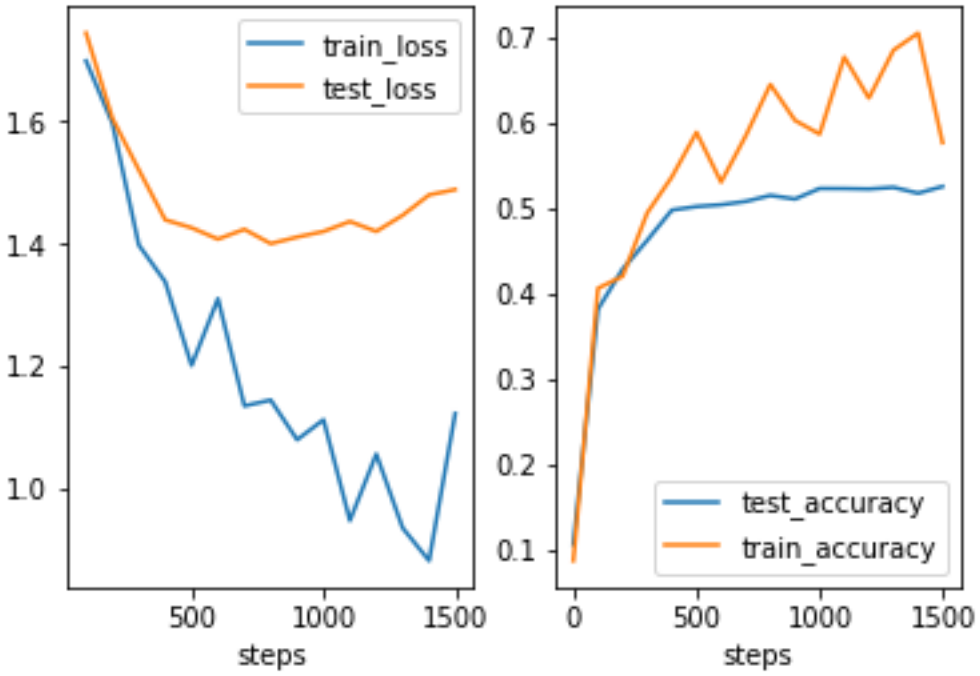

Figure 3: Test accuracy of 0.53. Loss and accuracy of best Pytorch implementation of MLP with parameters: $lr = 0.002, steps = 1500, batch\_size = 500, leaky\_relu\_slope = 0.02, , hidden\_units =$ "500, 500, 500"

The implementation is in the code.

3.2 a) Backprop Equation for Batch Normalization

We are interested in the gradient of loss w.r.t. γ, β and input x. In the backprop equation, the gradient of loss w.r.t. the output y is obtained by backpropgating from the latter layer.

$$\Big(\frac{\partial L}{\partial \gamma}\Big)_j = \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \gamma_j} \tag{26}$$

where the superscript s denotes batch count, i denotes the index of outputs.

As we know by definition of batch normalization $y_i^s = \gamma_i \hat{x}_i^s + \beta_i$. Therefore,

$$\frac{\partial y_i^s}{\partial \gamma_j} = 0, \text{if } i \neq j \tag{27}$$

Hence the second summation disappears:

$$\Big(\frac{\partial L}{\partial \gamma}\Big)_j = \sum_s \frac{\partial L}{\partial y_j^s}\frac{\partial y_j^s}{\partial \gamma_j} = \sum_s \frac{\partial L}{\partial y_j^s}\hat{x}_j \tag{28}$$

By following similar derivation process, we have:

$$\Big(\frac{\partial L}{\partial \beta}\Big)_j = \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \beta_j} = \sum_s \frac{\partial L}{\partial y_j^s}\frac{\partial y_j^s}{\partial \beta_j} = \sum_s \frac{\partial L}{\partial y_j^s} \tag{29}$$

The derivative w.r.t. input x is more complex due to the in dependencies across batches.

Using the extended version of chain rule, we can write out the derivative into 3 components of partial derivatives:

$$\begin{aligned}\Big(\frac{\partial L}{\partial x}\Big)_j^r &= \frac{\partial L}{\partial x_j^r} \\ &= \frac{\partial L}{\partial \hat{x}_j^r}\frac{\partial \hat{x}_j^r}{\partial x_j^r} + \frac{\partial L}{\partial \mu_j}\frac{\partial \mu_j}{\partial x_j^r} + \frac{\partial L}{\partial \sigma_j^2}\frac{\partial \sigma_j^2}{\partial x_j^r}\end{aligned}$$

Now we just need to solve different components in the above equation.

The 1st Term:

$$\boxed{\frac{\partial L}{\partial \hat{x}_j^r} = \frac{\partial L}{\partial y_j^r}\frac{\partial y_j^r}{\partial \hat{x}_j^r} = \frac{\partial L}{\partial y_j^r}\gamma_j}$$

Where $\frac{\partial L}{\partial y_j^r}$ is normally available as d_{out} backpropgated from deeper layer.

The 2nd Term:

$$\boxed{\frac{\partial \hat{x}_j^r}{\partial x_j^r} = \frac{1}{\sqrt{\sigma_j^2 + \epsilon}}}$$

The 3rd Term:

$$\frac{\partial L}{\partial \mu_j} = \sum_{s=1}^{B}\frac{\partial L}{\partial \hat{x}_j^s}\frac{\partial \hat{x}_j^s}{\partial \mu_j} + \frac{\partial L}{\partial \sigma_j^2}\frac{\partial \sigma_j^2}{\partial \mu_j} = \sum_{s=1}^{B}\frac{\partial L}{\partial \hat{x}_j^s}\Big(-\frac{1}{\sqrt{\sigma_j^2 + \epsilon}}\Big) + \frac{\partial L}{\partial \sigma_j^2}\Big(-\frac{2}{B}\sum_{s=1}^{B}(x_j^s - \mu_j)\Big)$$

As we plug in back, we realize the second part in this case is actually zero

$$\frac{1}{B}\sum_{s=1}^{B}(x_j^s - \mu_j) = \frac{1}{B}\sum_{s=1}^{B}x_j^s - \mu_j = 0$$

Therefore the final form of third term is:

$$\boxed{\frac{\partial L}{\partial \mu_j} = \sum_{s=1}^{B}\frac{\partial L}{\partial \hat{x}_j^s}(-\frac{1}{\sqrt{\sigma_j^2+\epsilon}})}$$

The 4th Term:

$$\boxed{\frac{\partial \mu_j}{\partial x_j^r} = \frac{1}{B}}$$

The 5th Term:

We need to calculate the derivative w.r.t. variance:

$$\boxed{\frac{\partial L}{\partial \sigma_j^2} = \sum_{s=1}^{B}\frac{\partial L}{\partial \hat{x}_j^s}\frac{\partial \hat{x}_j^s}{\partial \sigma_j^2} = \sum_{s=1}^{B}\frac{\partial L}{\partial \hat{x}_j^s}\frac{\partial\Big((x_j^s-\mu_j)(\sigma_j^2+\epsilon)^{-0.5}\Big)}{\partial \sigma_j^2} = -0.5\sum_{s=1}^{B}\frac{\partial L}{\partial \hat{x}_j^s}\Big((x_j^s-\mu_j)(\sigma_j^2+\epsilon)^{-1.5}\Big)}$$

The 6th Term:

$$\boxed{\frac{\partial \sigma_j^2}{\partial x_j^r} = \frac{\partial}{\partial x_j^r}\Big(\frac{1}{B}\sum_{s=1}^{B}(x_j^s-\mu_j)^2\Big) = \frac{2(x_j^r-\mu_j)}{B}}$$

Now we can put it all together:

$$\begin{aligned}
\Big(\frac{\partial L}{\partial x}\Big)_j^r &= \frac{\partial L}{\partial x_j^r} \\
&= \frac{\partial L}{\partial \hat{x}_j^r}\frac{\partial \hat{x}_j^r}{\partial x_j^r} + \frac{\partial L}{\partial \mu_j}\frac{\partial \mu_j}{\partial x_j^r} + \frac{\partial L}{\partial \sigma_j^2}\frac{\partial \sigma_j^2}{\partial x_j^r} \\
&= \frac{\partial L}{\partial \hat{x}_j^r}\frac{1}{\sqrt{\sigma_j^2+\epsilon}} + \frac{1}{B}\sum_{s=1}^{B}\frac{\partial L}{\partial \hat{x}_j^s}(-\frac{1}{\sqrt{\sigma_j^2+\epsilon}}) + \frac{\mu_j - x_j^r}{B}\sum_{s=1}^{B}\frac{\partial L}{\partial \hat{x}_j^s}\Big((x_j^s-\mu_j)(\sigma_j^2+\epsilon)^{-1.5}\Big) \\
&= \frac{\partial L}{\partial \hat{x}_j^r}\frac{1}{\sqrt{\sigma_j^2+\epsilon}} + \frac{1}{B}\sum_{s=1}^{B}\frac{\partial L}{\partial \hat{x}_j^s}(-\frac{1}{\sqrt{\sigma_j^2+\epsilon}}) + \frac{\mu_j - x_j^r}{B\sqrt{\sigma_j^2+\epsilon}}\sum_{s=1}^{B}\frac{\partial L}{\partial \hat{x}_j^s}\frac{(x_j^s-\mu_j)}{\sqrt{\sigma_j^2+\epsilon}}\frac{1}{\sqrt{\sigma_j^2+\epsilon}} \\
&= \frac{1}{\sqrt{\sigma_j^2+\epsilon}}\Big(\frac{\partial L}{\partial \hat{x}_j^r} - \frac{1}{B}\sum_{s=1}^{B}\frac{\partial L}{\partial \hat{x}_j^s} - \frac{\hat{x}_j^s}{B}\sum_{s=1}^{B}\frac{\partial L}{\partial \hat{x}_j^s}\hat{x}_j^s\Big)
\end{aligned}$$

Optionally, we can also plug in the first term, so that we can write out the equation to express it with the derivative of the output of the whole layer.

3.2 b) & c)

The implementation can be seen in the code.

4 PyTorch CNN

In this part, a CNN network with VGG architecture was implemented with pytorch. A test accuracy of around 0.79 was achieved. The training process and performance evaluation is shown in Figure 4. Overfitting does not occur even after around 5000 iterations (the margin between test and training accuracy is small). This experiment shows that a specialized network architecture like CNN works really well in its domain of image recognition. Some attributes that make it successful are trainable filters, shift invariance, etc. Also the presence of Batch Norm layer in the VGG network boosts the test accuracy and helps generization.

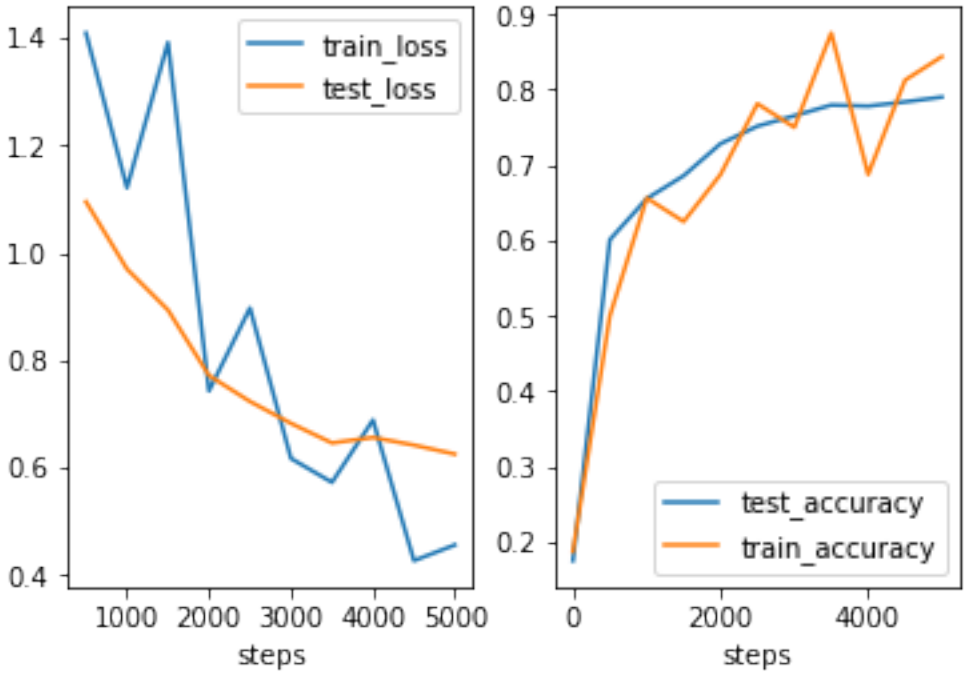

Figure 4: Test accuracy of 0.79. Loss and accuracy of VGG network with parameters: $lr = 0.0001, steps = 5000, batch\_size = 32, leaky\_relu\_slope = 0.02$