Deep Learning Assignment 1, Fall 2020

Blazej Manczak
University of Amsterdam
`blazej.manczak@student.uva.nl`

Abstract

This assignment treats the implementation of multilayer perceptron with pure numpy as well as PyTorch. I derive the update formulas and discuss the influence of different network characteristics on network performance. Subsequently, I implement the Normalization Layer with automatic differentiation as well as manual implementation with torch.autograd. Then, I implement VGG architecture with skip connections and discuss the results. Finally, I experiment with pretrained ResNet-18 model. All experiments performed with CIFAR10 dataset. Code is avalaiable at:

1 MLP backprop and NumPy Implementation

1.1 Evaluating the gradients

Linear Module gradients

$$\left[\frac{\partial L}{\partial W}\right]_{ij} = \frac{\partial L}{\partial W_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \frac{\partial Y_{mn}}{\partial W_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \frac{\partial}{\partial W_{ij}} \left[X W^T + B \right]_{mn}$$

$$= \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \frac{\partial}{\partial W_{ij}} \sum_p X_{mp} W^T_{pn} = \sum_{m,n} \frac{\partial L}{\partial Y_{m,n}} \sum_p X_{mp} \frac{\partial W_{np}}{\partial W_{ij}}$$

$$= \sum_{m,n} \frac{\partial L}{\partial Y_{m,n}} \sum_p X_{mp} \delta_{ni} \delta_{pj} = \sum_m \frac{\partial L}{\partial Y_{m,i}} X_{mj} = \sum_m \left[\frac{\partial L}{\partial Y}\right]^T_{im} X_{mj}$$

Hence $\frac{\partial L}{\partial W} = \left(\frac{\partial L}{\partial Y}\right)^T X$. Analogically :

$$\left[\frac{\partial L}{\partial X}\right]_{ij} = \frac{\partial L}{\partial X_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \frac{\partial Y_{m,n}}{\partial X_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{m,n}} \frac{\partial}{\partial X_{ij}} \sum_p X_{mp} W^T_{pn}$$

$$= \sum_{m,n,p} \frac{\partial L}{\partial Y_{mn}} \delta_{mi} \delta_{pj} W_{np} = \sum_n \frac{\partial L}{\partial Y_{in}} W_{nj}$$

. Hence $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} W$

$$\left[\frac{\partial L}{\partial b}\right]_i = \sum_{m,n} \frac{\partial L}{\partial Y_{m,n}} \frac{\partial Y_{mn}}{\partial B_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \frac{\partial}{\partial B_i} \left[X W^T + B \right]_{mn} =$$

$$= \sum_{m,n} \frac{\partial L}{\partial Y_{m,n}} \frac{\partial}{\partial b_i} B_{mn} = \sum_{m,n} \frac{\partial L}{\partial Y_{m,n}} \frac{\partial b_n}{\partial b_i} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \delta_{ni} =$$

$$= \sum_m \frac{\partial L}{\partial Y_{mi}}$$

. Hence $\frac{\partial L}{\partial b} = 1^T \frac{\partial L}{\partial Y}, \quad 1 \in \mathbb{R}^{S \times 1}$

Activation Module gradients

Note that $h'(X)$ is just the derivative of a scalar function ($\mathbb{R} \Rightarrow \mathbb{R}$) applied to each element of the matrix X. For ELU $h'(x) = 1$ if $x > 0$ and e^x if $x < 0$.

$$Y = h(X) \Rightarrow Y_{ij} = h(X_{ij})$$

$$\left(\frac{\partial L}{\partial X}\right)_{ij} = \sum_{mn} \frac{\partial L}{\partial Y_{mn}} \frac{\partial Y_{mn}}{\partial X_{ij}} = \sum_{mn} \frac{\partial L}{\partial Y_{mn}} \frac{\partial}{\partial X_{ij}} h(X_{mn})$$

$$= \sum_{mn} \frac{\partial L}{\partial Y_{mn}} h'(X_{mn}) \delta_{mi} \delta_{nj} = \frac{\partial L}{\partial Y_{ij}} h'(X_{ij})$$

$$= \left(\frac{\partial L}{\partial y} \circ h'(X)\right)_{ij}. \text{ Hence } \frac{\partial L}{\partial X} = \frac{\partial L}{\partial y} \circ h'(X)$$

Softmax and Loss Modules

(i) $$Y_{ij} = \frac{e^{X_{ij}}}{\sum_k e^{X_{ik}}}$$

$$\frac{\partial L}{\partial X_{ij}} = \sum_{mn} \frac{\partial L}{\partial Y_{mn}} \frac{\partial Y_{mn}}{\partial X_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \frac{\partial}{\partial X_{ij}} \frac{e^{X_{mn}}}{\sum_k e^{X_{mk}}} =$$

$$= \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \frac{e^{X_{mn}} \delta_{mi} \delta_{nj} \sum_k e^{X_{mk}} - e^{X_{mn}} \sum_k \delta_{mi} \delta_{kj} e^{X_{mk}}}{\left(\sum_k e^{X_{mk}}\right)^2}$$

$$= \frac{\partial L}{\partial Y_{ij}} \frac{e^{X_{ij}}}{\sum_k e^{X_{mk}}} - \sum_n \frac{\partial L}{\partial Y_{in}} \frac{e^{X_{in}} e^{X_{ij}}}{\sum_k e^{X_{ik}} \cdot \sum_k e^{X_{ik}}} =$$

$$= \frac{\partial L}{\partial Y_{ij}} Y_{ij} - \sum_n \frac{\partial L}{\partial Y_{in}} Y_{in} Y_{ij} = \left(\frac{\partial L}{\partial Y} \circ Y\right)_{ij} - \sum_n \left(\frac{\partial L}{\partial Y} \circ Y\right)_{in} Y_{ij}$$

(ii) $$L = \frac{1}{S} \sum_i L_i = -\frac{1}{S} \sum_{ik} T_{ik} \log(X_{ik})$$

$$\left[\frac{\partial L}{\partial X}\right]_{ij} = \frac{\partial}{\partial X_{ij}} - \frac{1}{S} \sum_{mk} T_{mk} \log(X_{mk}) = -\frac{1}{S} \sum_{mk} \frac{T_{mk}}{X_{mk}} \delta_{mi} \delta_{kj}$$

$$= -\frac{1}{S} \cdot \frac{T_{ij}}{X_{ij}}. \text{ Hence } \frac{\partial L}{\partial X} = -\frac{1}{S}\left[T \oslash X\right]$$

Hadamard division

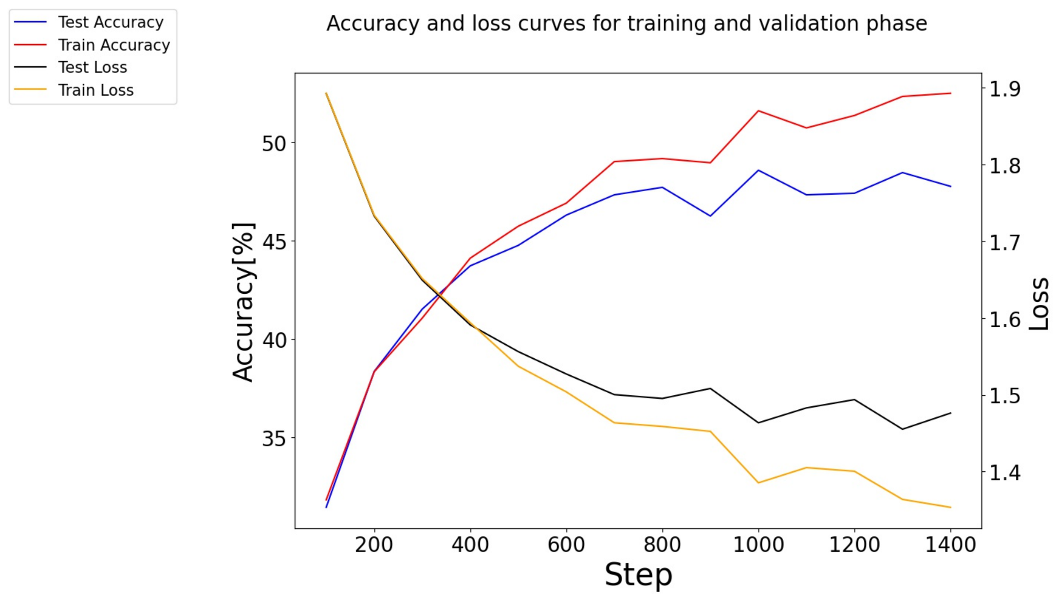

Figure 1: Loss and accuracy curve for MLP run on default settings.

1.2 NumPy implementation

In Figure 1 we see the loss and accuracy curve with the default settings specified in the assignment.

We see that the accuracy on the test set is saturating around 48% whereas the accuracy on the training around 52%. Similar trend where the measure on the test set is slightly worse can be observed for the average loss. We see that the average test loss saturates around 1.5 and train loss around 1.35.

2 PyTorchMLP

To improve the accuracy obtained in the NumPy implementation we modify experiment with a couple of addition. Firstly, I changed the optimizer to Adam with small weight decay, 10^{-4}. Then, I experimented with depth. With increasing the number of layers to 2 with hidden sizes the testset performance slightly increased to around 50%. Adding subsequent layers did not yield significant improvements. Thus, normalization has been considered. Batch normalization has increased the convergence speed, especially in in the initial batches and allowed extra the extra layers to increase performance. We find that a neural network with 3 hidden layers with sizes 256, 128 and 64 respectively performs well, converging to around 53% on the test set, see Figure 2. Other experiments such as Dropout, different activation functions like ReLU, tanh and ELU and intializations (Xavier and Kaiming) were considered but did not offer any significant improvements (Dropout with higher values actually hurt the performance). The results with the three hidden layers (256,128,64), Adam (lr= 0.001, decay = 10^{-4}) as an optimizer, Dropout(p=0.1) before last linear layer, batch size 200 and batch normalization after every hidden layer are portrayed in Figure 2 (3000 steps). The command to replicate this result is in the comment at line 42 of $train\_mlp\_pytorch.py$ script.

However, we see that the problem of overfitting (training accuracy significantly higher than test accuracy) is present and in general performance is not where we would like it to be. This will be improved in Section 4.

To answer the question about Tanh vs. ELU: Elu is simpler and more computationally efficient, as for positive values its derivative is simply 1 and for negative values the derivative does not have to be recomputed (e^x used in the forward pass (which is $e^x - 1$) can be saved to calculate the derivative). A drawback of ELU compared to Tanh is that the gradients can explode if the outputs are big and positive. However, I think that ELU is in principle a better choice as we do not have to deal with the vanishing gradient problem we encounter with Tanh.

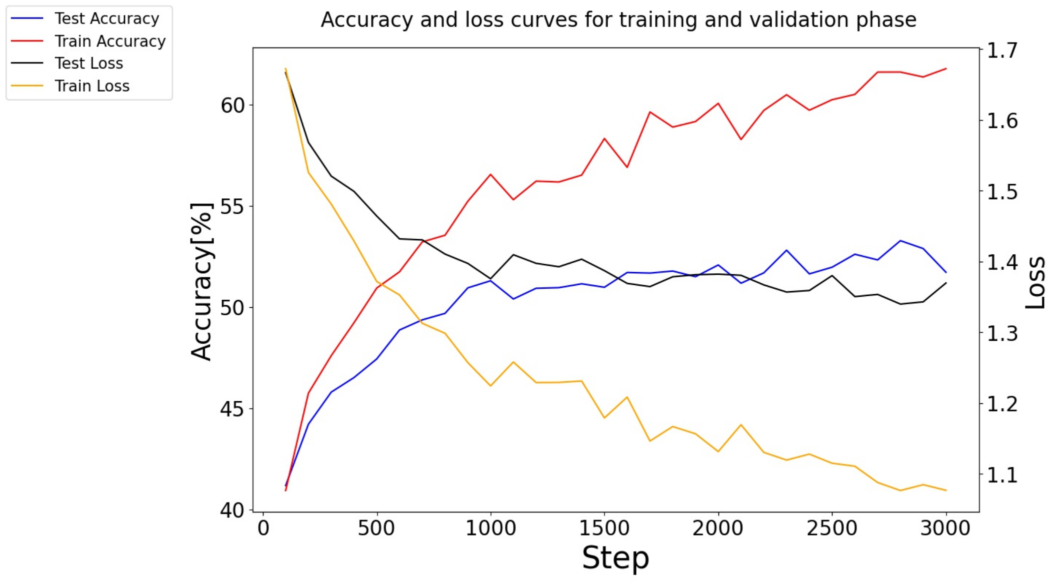

Figure 2: Loss and accuracy curve for MLP run with three hidden layers (256,128,64), Adam (lr= 0.001, decay = 10^{-4}) as an optimizer, Dropout(p=0.1), batch size 200, before last linear layer,and batch normalization after every hidden layer (3000 stpes)

3 Custom Module: Layer Normalization

3.2 Layer vs. Batch norm

In the report I just briefly talk about Layer and Batch normalization. Both of these normalization methods aim to make the job of the optimizer easier and enable more efficient training. The formulas for the two are deceptively similar: they differ only in regards to the axis over which we normalize. Batch normalization computes the mean and variance for each batch, performs the normalization and outputs the normalized input additionally parametrised by scale and shift parameters. Thanks to this operation, the higher order layer interactions are suppressed and training becomes easier.

However,there are situations where batch norm is not suitable. For example, when the batch size is small we can't calculate (if batch size 1) or get noisy estimates for the mean and variance (larger batch size -> better estimates -> better performing batch normalization). Moreover, batch norm is expensive to use for RNN's because one would have to store the statistics for each time step to account for different statistics of recurrent activations at each time-step.

This is where Layer normalization comes in: the mean and variance are calculated independently for each element of the batch by aggregating over the feature dimension (in contrast to batch dimension for batch norm). Hence, the size of the batch does not matter for Layer norm as statistics are calculated independently for each element! This makes Layer normalization more suitable for RNN's or online learning (small batch size, often 1).

3.1 Gradient derivation for layer normalization

$$\frac{\partial \mu_s}{\partial x_{tj}} = \frac{1}{M}\sum_{i=1}^{M}\frac{\partial x_{si}}{\partial x_{tj}} = \frac{1}{M}\sum_{i=1}^{M}\delta_{st}\,\delta_{ij} = \frac{1}{M}\,\delta_{st}$$

$$\frac{\partial \sigma_s^2}{\partial x_{tj}} = \frac{\partial}{\partial x_{tj}}\left[\frac{1}{M}\sum_{i=1}^{M}(x_{si}-\mu_s)^2\right] =$$

$$= \frac{1}{M}\sum_{i=1}^{M}2\,(x_{si}-\mu_s)\left[\delta_{st}\,\delta_{ij} - \frac{\partial \mu_s}{\partial x_{tj}}\right] =$$

$$= \frac{2}{M}\sum_{i=1}^{M} x_{si}\,\delta_{st}\,\delta_{ij} - x_{si}\cdot\frac{1}{M}\delta_{st} - \mu_s\,\delta_{st}\,\delta_{ij} + \mu_s\,\frac{1}{M}\delta_{st} =$$

$$= \frac{2}{M}\left[x_{sj}\,\delta_{st} - \mu_s\cancel{\delta_{st}} + \mu_s\cancel{\delta_{st}} - \frac{1}{M}\delta_{st}\sum_{i=1}^{M}x_{si}\right] = \frac{2}{M}\,x_{sj}\,\delta_{st} - \frac{2}{M^2}\,\delta_{st}\sum_{i=1}^{M}x$$

$$= \frac{2}{M}\delta_{st}\left(x_{sj} - \underbrace{\frac{1}{M}\sum_{i=1}^{M}x_{si}}_{\overset{\shortparallel}{\mu_s}}\right)$$

$$\frac{\partial \hat{x}_{si}}{\partial x_{tj}} = \frac{\partial}{\partial x_{tj}}\;\frac{x_{si}-\mu_s}{\sqrt{\sigma_s^2+\varepsilon}} =$$

$$\frac{\left(\frac{\partial x_{si}}{\partial x_{tj}} - \frac{\partial \mu_s}{\partial x_{tj}}\right)\sqrt{\sigma_s^2+\varepsilon} - (x_{si}-\mu_s)\,\frac{1}{2\sqrt{\sigma_s^2+\varepsilon}}\cdot\frac{\partial \sigma_s^2}{\partial x_{tj}}}{\sigma_s^2+\varepsilon} =$$

$$= \frac{\delta_{st}\,\delta_{ij} - \frac{1}{M}\delta_{st}}{\sqrt{\sigma_s^2+\varepsilon}} - \frac{(x_{si}-\mu_s)\cdot\frac{2}{M}\,\delta_{st}\,(x_{sj}-\mu_s)}{2\,(\sigma_s^2+\varepsilon)^{\frac{3}{2}}}$$

$$\frac{\partial Y_{si}}{\partial X_{tj}} = \frac{\partial}{\partial X_{tj}}\left[\gamma_i \hat{X}_{si} + \beta_i\right] = \gamma_i \frac{\partial \hat{X}_{si}}{\partial X_{tj}}$$

$$\frac{\partial L}{\partial X_{tj}} = \sum_{s,i} \frac{\partial L}{\partial Y_{si}} \frac{\partial Y_{si}}{\partial X_{tj}} = \sum_{s,i}\frac{\partial L}{\partial Y_{si}}\gamma_i \frac{\partial \hat{X}_{si}}{\partial X_{tj}}$$

$$= \sum_{s,i} \frac{\partial L}{\partial Y_{si}} \gamma_i \left[\frac{\delta_{st}\delta_{ij} - \frac{1}{M}\delta_{st}}{\sqrt{\sigma_s^2+\varepsilon}} - \frac{(X_{si}-\mu_s)\cdot\delta_{st}(X_{sj}-\mu_s)}{M(\sigma_s^2+\varepsilon)^{\frac{3}{2}}}\right] =$$

$$= \frac{\frac{\partial L}{\partial Y_{tj}}\gamma_j}{\sqrt{\sigma_s^2+\varepsilon}} - \frac{\frac{1}{M} \sum_i \frac{\partial L}{\partial Y_{ti}}\gamma_i}{\sqrt{\sigma_s^2+\varepsilon}} - \sum_i \frac{\partial L}{\partial Y_{ti}}\gamma_i \frac{(X_{ti}-\mu_s)(X_{tj}-\mu_s)}{M(\sigma_s^2+\varepsilon)^{\frac{3}{2}}} =$$

$$= \frac{1}{M\sqrt{\sigma_s^2+\varepsilon}} \cdot \left[M\cdot \frac{\partial L}{\partial Y_{tj}}\gamma_j - \sum_{i=1}^{M}\frac{\partial L}{\partial Y_{ti}}\gamma_i - \hat{X}_{tj} \sum_i \frac{\partial L}{\partial Y_{ti}}\gamma_i \hat{X}_{ti}\right]$$

dY in code

4 PyTorch CNN

In this part we significantly improve upon the results from Sections 1 and 2 by employing a VGG architecture with skip connections and a ImageNet pretrained ResNet18 architecture.

As can be seen on Figure 3, VGG with skip connections trained from scratch achieves around 80% accuracy on the test data.

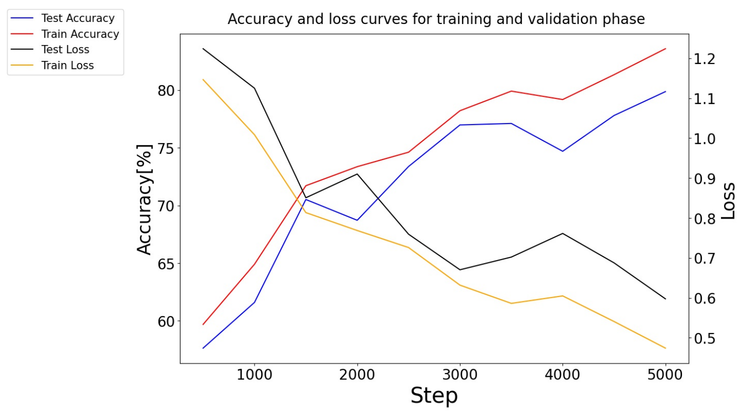

Figure 3: Loss and accuracy curve for VGG with skip connections run on default settings.

To further improve the performance I experimented with the ResNet-18 model pretrained on ImageNet, see script $transfer\_learning.py$. To improve generalisability I am using random horizontal flips

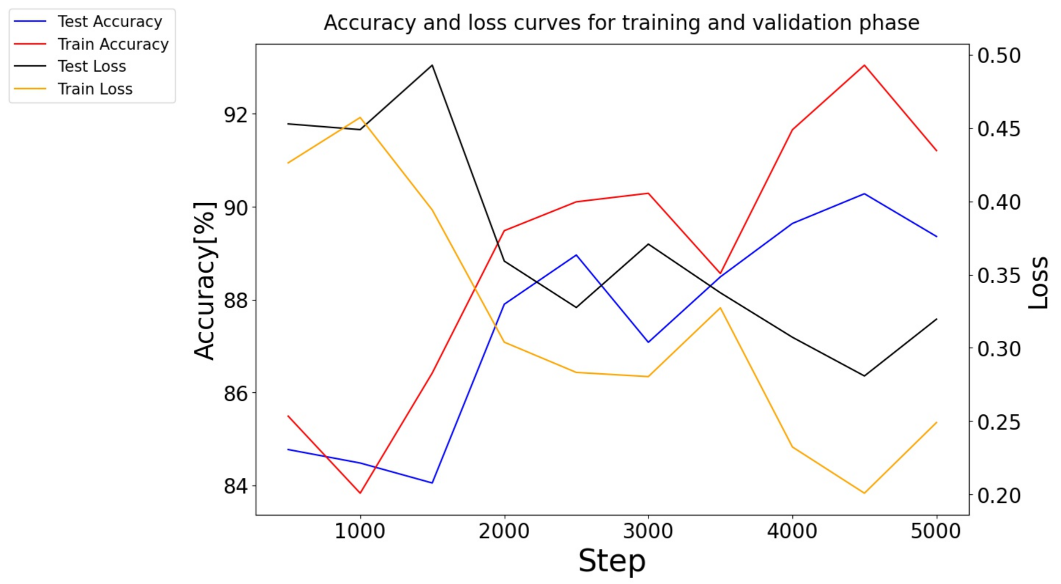

Figure 4: Loss and accuracy curve for pretrained ResNet run on default settings for VGG with skip connections.

and random crops in this case. To adapt the architecture I re-scaled the 32x32 CIFAR-10 images to 224x224. Furthermore, to adapt the weights to the new domain I unfroze the 4-th block and fully-connected layers of the ResNet-18 model. Additionally, I added an extra fully-connected layer at the end. This way, I managed to achieve approx. 90% on the CIFAR-10 test set, see Figure 4 for loss and accuracy curves.