Deep Learning 2020, Assignment 1

Spyros Avlonitis
spyrosavl@gmail.com
UvA ID: 12899283

Abstract

In this assignment I implemented and experimented with a multiple layer perceptron and a VGG13 convolutional neural network, in order to classify images from the CIFAR10 dataset.

1 MLP backprop and NumPy Implementation

1.1 Evaluating the gradients

1.1.1 Linear module

Consider a linear module as described above. The input and output features are labeled as X and Y, respectively. Find closed form expressions for

$$\frac{\partial L}{\partial W}, \frac{\partial L}{\partial b}, \frac{\partial L}{\partial X}$$

$$\frac{\partial L}{\partial w_{ij}} = \sum_{nm} \frac{\partial L}{\partial y_{nm}} \frac{\partial (x_{nk} w_{km}^T + b_{nm})}{\partial w_{ij}} = \sum_{nm} \frac{\partial L}{\partial y_{nm}} \delta_{km} \delta_{mi} X_{nk} = \sum_{n} x_{jn}^T \frac{\partial L}{\partial y_{ni}}$$

$$\Rightarrow \frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial Y}$$

$$\frac{\partial L}{\partial b_{ij}} = \sum_{nm} \frac{\partial L}{\partial y_{nm}} \frac{0 + b_{nm}}{\partial b_{ij}} = \sum_{nm} \frac{\partial L}{\partial y_{nm}} \delta_{in} \delta_{jm} = \frac{\partial L}{\partial y_{ij}}$$

$$\Rightarrow \frac{\partial L}{\partial B} = \frac{\partial L}{\partial Y}$$

$$\frac{\partial L}{\partial x_{ij}} = \sum_{nm} \frac{\partial L}{\partial y_{nm}} \frac{\partial (x_{nk} w_{km}^T + b_{nm})}{\partial x_{ij}} = \sum_{nm} \frac{\partial L}{\partial y_{nm}} \frac{\delta_{in} \delta_{jk} w_{km}^T}{\partial x_{ij}} = \sum_{nm} \frac{\partial L}{\partial y_{nm}} w_{mj}$$

$$\Rightarrow \frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} W$$

1.1.2 Activation Module

Consider an element-wise activation function h. The activation module has input and output features labeled by X and Y, respectively. I.e. $Y = h(X) \Rightarrow Y_{ij} = h\left(X_{ij}\right)$. Find a closed form expression for

$$\frac{\partial L}{\partial X}$$

$$\frac{\partial L}{\partial x_{ij}} = \sum_{nm} \frac{\partial L}{\partial y_{nm}} \frac{\partial y_{nm}}{\partial x_{ij}} = \sum_{nm} \frac{\partial L}{\partial y_{nm}} \frac{\partial h(x)}{\partial x_{nm}} \frac{\partial x_{nm}}{\partial x_{ij}} = \sum_{nm} \frac{\partial L}{\partial y_{nm}} \frac{\partial h(x)}{\partial x_{nm}} \delta_{in}\delta_{jm} = \frac{\partial L}{\partial y_{ij}} \frac{\partial h(x)}{\partial x_{ij}}$$

$$\Rightarrow \frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \cdot \frac{\partial h}{\partial x}$$

1.1.3 Softmax and Loss Modules

i. Consider a softmax module such that $Y_{ij} = [softmax(X)]_{ij}$, where X is the input and Y is the output of the module. Find an expression for $\frac{\partial L}{\partial X}$ in terms of $\frac{\partial L}{\partial Y}$. You may keep your result in index notation for this sub-task only.
ii. The gradient that kicks the whole backpropagation algorithm off is the one for the loss module itself. The loss module for the categorical cross entropy takes as input X and returns $L = \frac{1}{S}\sum_i L_i = -\frac{1}{S}\sum_{ik} T_{ik} \log\left(X_{ik}\right)$. Find a closed form expression for $\frac{\partial L}{\partial X}$. Write your answer in terms of matrix operations.

Softmax module

$$\frac{\partial L}{\partial x_{ij}} = \sum_{nm} \frac{\partial L}{\partial y_{nm}} \frac{\partial y_{nm}}{\partial x_{ij}}$$

$$y_{nm} = \frac{g_{nm}}{h_{nm}} = \frac{e^{X_{ij}}}{\sum_k e^{X_{ik}}}$$

$$\Rightarrow \frac{\partial L}{\partial x_{ij}} = \sum_{nm} \frac{\partial L}{\partial y_{nm}} \frac{g_{nm}\delta_{in}\delta_{mj}h_n - g_{nm}\delta_{in}\frac{\partial h_n}{\partial x_{ij}}}{(h_n)^2} = \frac{\partial L}{\partial x_{ij}} = \sum_{m} \frac{\partial L}{\partial y_{im}} y_{im}(\delta_{mj} - e^{x_{ij}}\frac{1}{h_i})$$

$$\Rightarrow \frac{\partial L}{\partial x_{ij}} = y_{ij}(\frac{\partial L}{\partial y_{ij}} - \sum_{m} \frac{\partial L}{\partial y_{im}} y_{im})$$

Loss module

$$\frac{\partial L}{\partial x_{nm}} = \frac{-1}{S} \sum_{ik} \frac{\partial (T_{ik}\log(x_{ik}))}{\partial x_{nm}} = \frac{-1}{S} \frac{T_{nm}}{x_{nm}}$$

$$\Rightarrow \frac{\partial L}{\partial X} = -\frac{1}{S}\frac{T}{X}$$

1.2 NumPy implementation

The implementation of NumPy MLP can be found in "mlp_numpy.py" and "train_mpl_numpy.py". The loss and accuracy curves are presented in figure 1. By using the defaults parameters we can reach a test accuracy of 0.481300.

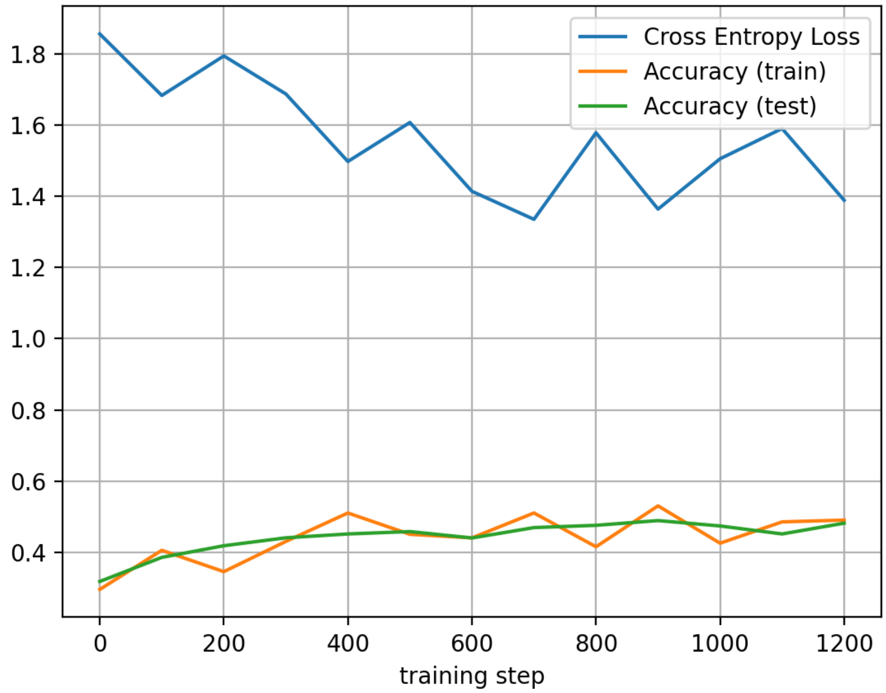

Figure 1: Loss Accuracy curves for NumPy MLP with default parameters.

Table 1: PyTorch MLP Parameters

Configuration	Hidden Layers	Optimizer	Learning Rate	Max Steps	Test Accuracy
1	100	SGD	1e-3	1400	0.421
2	100	Adam	1e-3	1400	0.519
3	100,100	Adam	1e-3	1400	0.521
4	100,100	Adam	1e-3	3000	0.529
5	500,300	Adam	1e-3	3000	0.546
6	500,300	Adam	1e-2	3000	0.532
7	500,300,100	Adam	1e-3	3000	0.555

2 PyTorch MLP

2.1 Implementation

The implementation of the MLP using PyTorch can be found "mlp_pytorch.py" and "train_mpl_pytorch.py". The modification that I did, in order to achieve a better accuracy can be found on the table 1 . The loss and accuracy curve of the best model can be found on figure 2.

2.2 Tanh vs ELU

ELU has stronger derivatives compared to Tanh. In addition, Tanh has the vanishing gradient issue. On the other hand, ELU can blow up the activations for x > 0.

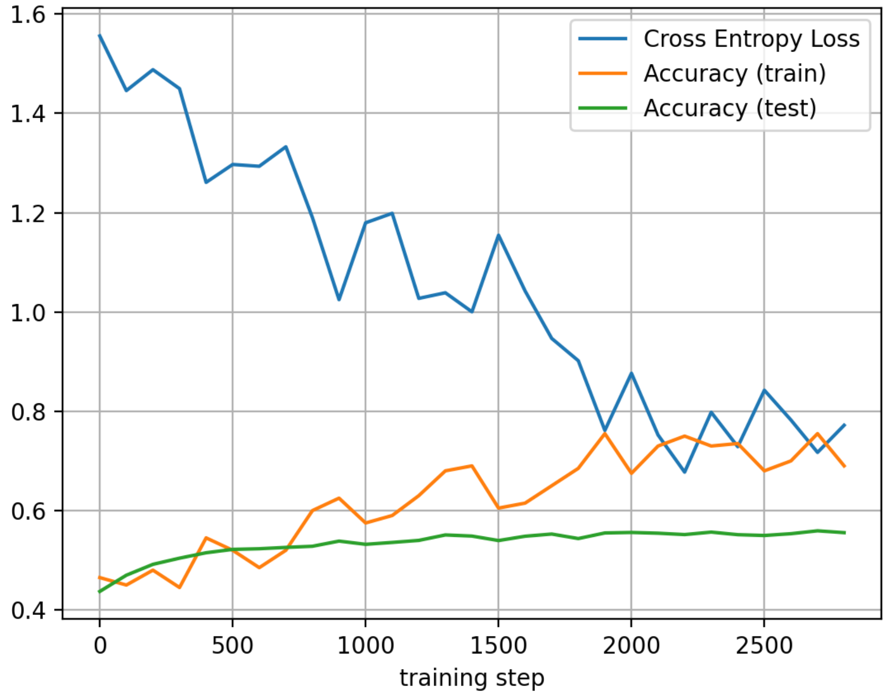

Figure 2: Loss Accuracy curves for PyTorch MLP with Configuration No 7 from table 1.

3 Custom Module: Layer Normalization

3.1 Automatic differentiation

The automatic differentiation code has be implemented in "custom_layernorm.py".

3.2 Manual implementation of backward pass

3.2.1

Compute the backpropagation equations for the layer normalization operation, that is, compute

$$\frac{\partial L}{\partial \gamma}, \quad \frac{\partial L}{\partial \beta} \quad and \quad \frac{\partial L}{\partial X}$$

$$\frac{\partial L}{\partial \gamma_i} = \sum_{sj} \frac{\partial L}{\partial y_{sj}} \frac{\partial y_{sj}}{\partial \gamma_i} = \sum_{sj} \frac{\partial L}{\partial y_{sj}} \delta_{ji} \hat{x}_{sj} = \sum_{s} \frac{\partial L}{\partial y_{si}} \hat{x}_{si}$$

$$\frac{\partial L}{\partial \beta_i} = \sum_{sj} \frac{\partial L}{\partial y_{sj}} \frac{\partial y_{sj}}{\partial \beta_i} = \sum_{sj} \frac{\partial L}{\partial y_{sj}} \delta_{ji} = \sum_{s} \frac{\partial L}{\partial y_{si}}$$

3.2.2

Implementation can be found in "custom_layernorm.py" but grad_input is not implemented. I was not able to find the derivative.

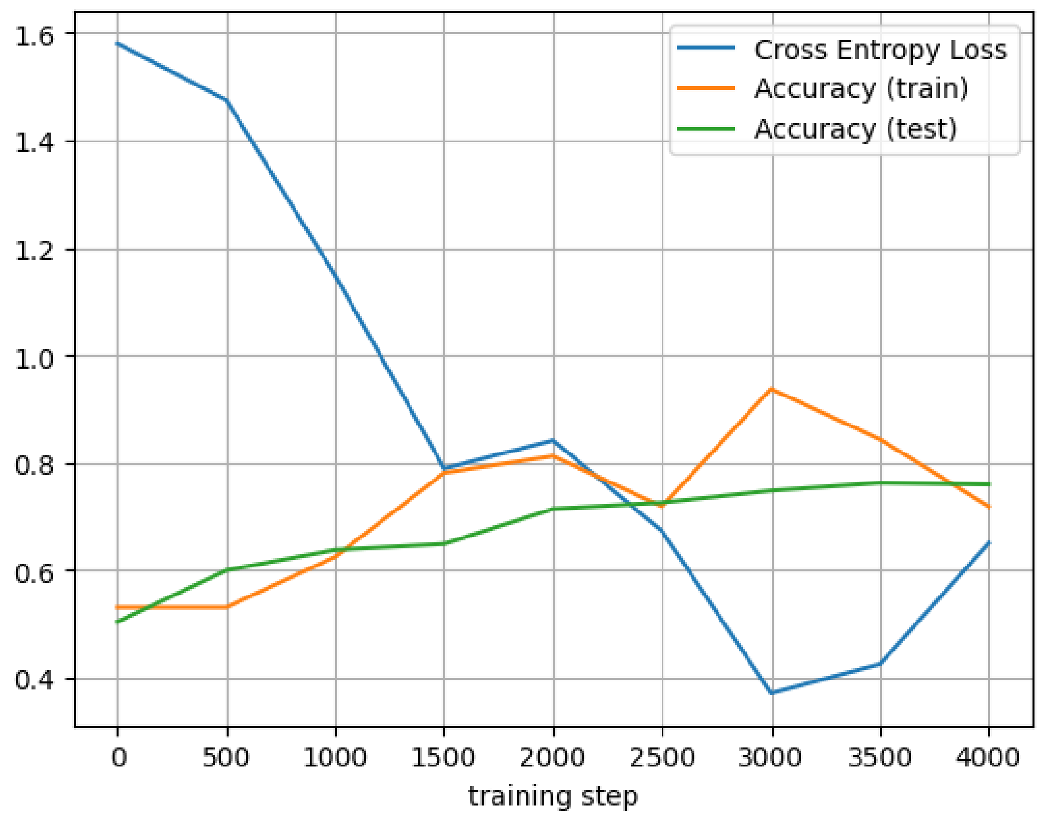

Figure 3: Loss Accuracy curves for PyTorch ConvNet with default parameters.

3.2.3

Implementation can be found in "custom_layernorm.py".

3.2.4 Layer Normalization vs Batch Normalization

Batch normalization normalizes the input features across the batch dimension. On the other hand, layer normalization normalizes the inputs across the features. This means that in layer normalization we have an independence between inputs and in bath normalization between features. Experimental results show that layer normalization performs well for recurrent neural networks.
In addition, batch normalization adds lower limit on the size of the batch. It can't work well with very small batches as it can't gather enough statistics for normalization. In layer normalization we don't have this issue.

4 PyTorch CNN

4.1 Implementation

The VGG13 network implemented with PyTorch is available in "convnet_pytorch.py" and "train_convnet_pytorch.py". The accuracy achieved by using the default parameters is 0.759706. The loss and accuracy curves can be found on figure 3.

4.2 Transfer Learning

By using a VGG13 pretrained network, offered by PyTorch, and fine tuning it on CIFAR10 we can achieve a higher accuracy of 0.82. You can run the pretrained network by setting the

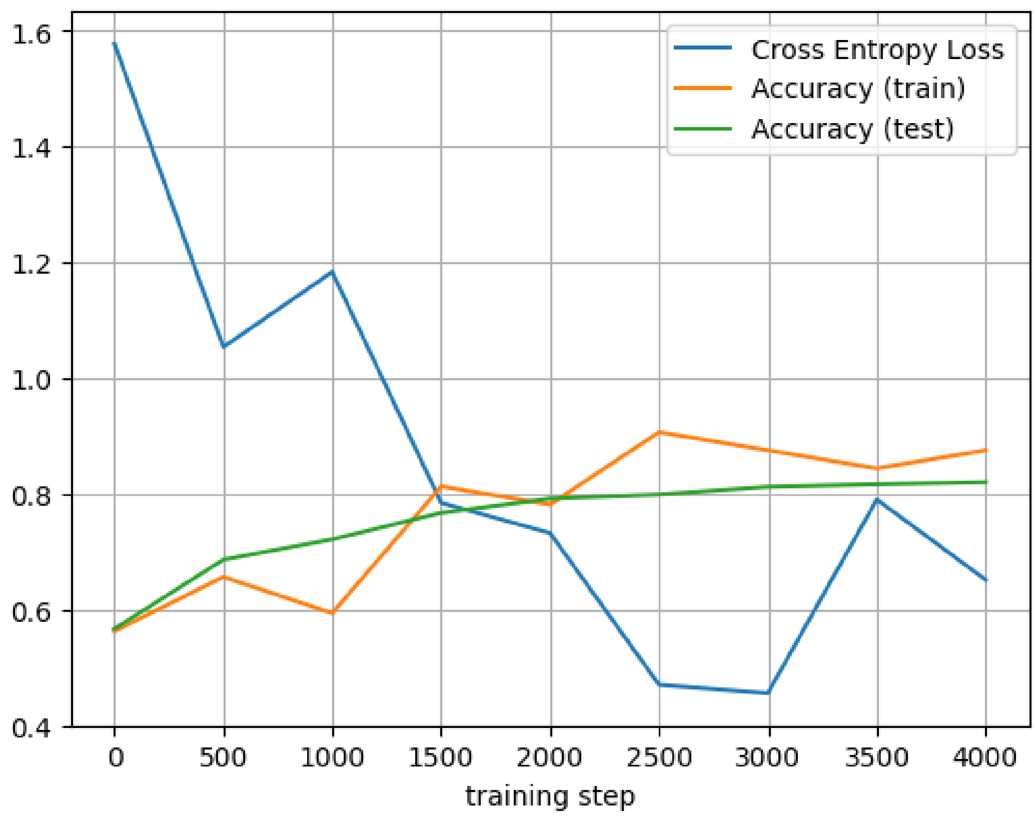

Figure 4: Loss Accuracy curves for PyTorch MLP by using a VGG13 pretrained network.

"USE_PRETRAINED_VGG" variable to True. The loss and accuracy curves can be found on figure 4.