Deep Learning 2020 Homework 1

Marcel Velez Vasquez - 11299746
University of Amsterdam
Science Park 904
marcel.velez1997@hotmail.com

1 MLP backprop and NumPy Implementation

For the derivatives I have consulted chapter 2 of the matrix cookbook[1] and the matrix calculus wikipedia page[1].

1.1 Evaluating the Gradients

a Linear Module: with $\mathbf{Y} = \mathbf{X}\mathbf{W}^\top + \mathbf{B}$

- The derivative of $\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial \mathbf{W}} = \frac{\partial L}{\partial Y}\mathbf{X} \otimes \mathbb{1}$
- The derivative of $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial b} = \frac{\partial L}{\partial Y}\mathbb{1}$
- The derivative of $\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial \mathbf{X}} = \frac{\partial L}{\partial Y}\mathbb{1} \otimes \mathbf{W}^\top$

b Activation Module:
For the derivative of an element-wise operator there is a general form, derived with help of a TA:

$$\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial \mathbf{X}} = \left[\frac{\partial L}{\partial Y} \circ \frac{\partial h(X)}{\partial \mathbf{X}}\right]_{ij} \tag{1}$$

c Softmax and Loss Modules:

- Softmax gradient:
The softmax gradient consists of 2 cases, i $\neq$ j and i=j. resulting in a matrix with the $i = j$ along the diagonal and the i $\neq$ j case in the rest of the resulting matrix

$$\frac{\partial \mathcal{L}}{\partial X} = \frac{\partial \mathcal{L}}{\partial Y}\frac{\partial Y}{\partial X} = \frac{\partial \mathcal{L}}{\partial Y}\frac{exp(x_i)}{\sum_{k=1} exp(x_k)} \tag{2}$$

$$= \frac{exp(x_j)}{\sum_{k=1} exp(x_k)} \cdot \frac{\sum_{k=1} exp(x_k) - exp(x_i)}{\sum_{k=1} exp(x_k)} \tag{3}$$

$$= \sigma_j(1 - \sigma_i)(\text{ if i=j}) \tag{4}$$

$$= \frac{0 - exp(x_j)exp(x_i)}{(\sum_{k=1} exp(x_k))^2} = -\sigma_j\sigma_i(\text{ if i} \neq \text{j}) \tag{5}$$

[1] `https://en.wikipedia.org/wiki/Matrix_calculus`

Submitted to 34th Conference on Neural Information Processing Systems (NeurIPS 2020). Do not distribute.

- CrossEntropy gradient:

$$\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial}{\partial \mathbf{X}} - \frac{1}{S}\sum_{ik} T_{ik}\log(X_{ik}) = -\frac{1}{S} * \frac{T}{X} \tag{6}$$

d Bonus Convolution gradient:
In order to keep the gradients clear the matrix multiplications with 0 in b) and c) are omitted. The assumption is also made that the kernel doesnot have to be flipped again, so not a correlation matrix.

(a) Forward propagation for Y_{21}:

$$\begin{aligned} Y &= \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} * \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \\ &= \begin{bmatrix} (X_{11}*K_{11} + X_{12}*K_{12} + X_{21}*K_{21} + X_{22}*K_{22}) & (X_{12}*K_{11} + X_{13}*K_{12} + X_{22}*K_{21} + X_{23}*K_{22}) \\ (X_{21}*K_{11} + X_{22}*K_{12} + X_{31}*K_{21} + X_{32}*K_{22}) & (X_{22}*K_{11} + X_{23}*K_{12} + X_{32}*K_{21} + X_{33}*K_{22}) \end{bmatrix} \end{aligned} \tag{7}$$

thus $Y_{21} = X_{21} * K_{11} + X_{22} * K_{12} + X_{31} * K_{21} + X_{32} * K_{22}$

(b) Gradient of $\frac{\partial \mathcal{L}}{\partial K_{11}}$:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial K_{11}} = \frac{\partial \mathcal{L}}{\partial Y}\frac{\partial}{\partial K_{11}} \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} * \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} = \\ \frac{\partial \mathcal{L}}{\partial Y} \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} * \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = \frac{\partial \mathcal{L}}{\partial Y} \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \end{aligned} \tag{8}$$

(c) Gradient of $\frac{\partial \mathcal{L}}{\partial X_{12}}$:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial X_{12}} &= \frac{\partial \mathcal{L}}{\partial Y}\frac{\partial Y}{\partial X_{12}} \\ &= \frac{\partial \mathcal{L}}{\partial Y}\frac{\partial}{\partial X_{12}} \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} * \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \\ &= \frac{\partial \mathcal{L}}{\partial Y} \begin{bmatrix} 0 & X_{12} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \\ &= \frac{\partial \mathcal{L}}{\partial Y} \begin{bmatrix} K_{12} & K_{11} \\ 0 & 0 \end{bmatrix} \end{aligned} \tag{9}$$

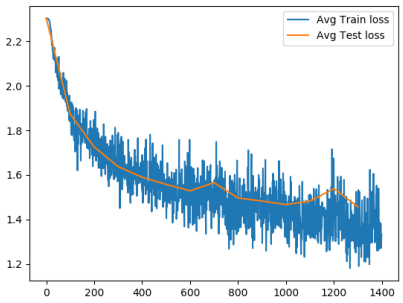
1.2 Numpy MLP

My numpy implementation achieved a whopping 48% accuracy on the test set with no alteration or extra commandline arguments. When looking at the both the loss and accuracy curves, see Figures 1a and 1b, we see that the training and test values do not diverge a lot and thus the model is not (yet) overfitting. We can see that the last 100 or so steps the train and test values start to diverge more and the model will most likely start to overfit if set to train for more steps.

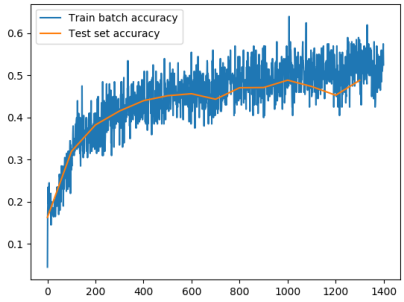
2 PyTorch MLP

2.1 Modifications and why use those modifications

When playing with the hyperparameters something I remembered from last year is that batchnormalization helped increase the performance a lot, the difference with and without batchnorm can be seen in Figures 2b and 3b. An even bigger increase in performance can be achieved when combining batchnorm with a bigger architecture and more training steps, as can be seen in Figure 4b.

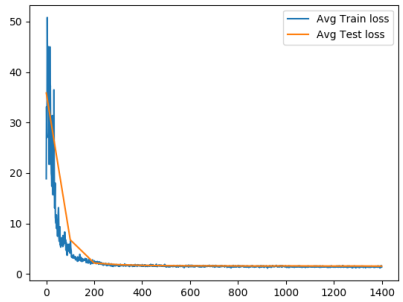


(a) The average loss per batch of the train set and average loss for the entire test set during training.

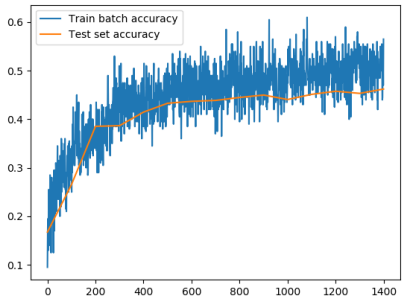


(b) The average accuracy per batch of the train set and average accuracy for the entire test set during training.

Figure 1: Training statistics of my numpy implementation of an MLP

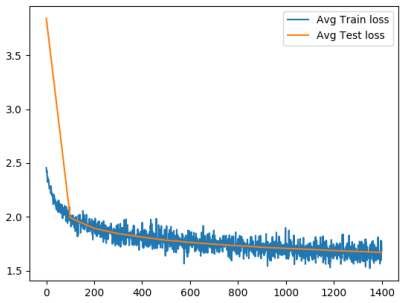


(a) The average loss per batch of the train set and average loss for the entire test set during training.

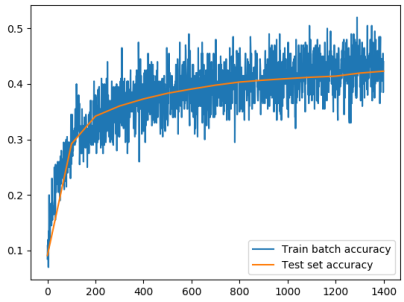


(b) The average accuracy per batch of the train set and average accuracy for the entire test set during training.

Figure 2: Training statistics of the Pytorch MLP model with the standard hyperparameters and no batchnormalization.

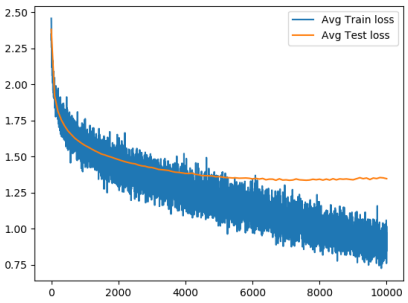


(a) The average loss per batch of the train set and average loss for the entire test set during training.

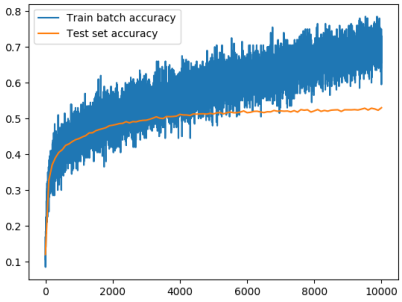


(b) The average accuracy per batch of the train set and average accuracy for the entire test set during training.

Figure 3: Training statistics of the Pytorch MLP model with batchnormalization, and more neurons [512,256,128]



(a) The average loss per batch of the train set and average loss for the entire test set during training.



(b) The average accuracy per batch of the train set and average accuracy for the entire test set during training.

Figure 4: Training statistics of the Pytorch MLP model with batchnormalization, more train steps, and more neurons [512,256,128]

2.2 Tanh vs. ELu

One of the benefits of Tanh is it maps everything between fixed bounds whereas ELu is [-1,> when dealing with non-normalized features. A drawback of tanh is that it suffers from vanishing gradients as opposed to ELU that does not suffer from vanishing gradients.

3 Custom Module: Layer Normalization

3.1 Automatic differentiation

For the implementation of the layer normalization See File "custom_layernorm.py".

3.2 Manual implementation of backward pass

a)Compute the backpropagation equations:

- $$\frac{\partial L}{\partial \gamma} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial \gamma} = \frac{\partial L}{\partial Y}\hat{X} \tag{10}$$
- $$\frac{\partial L}{\partial \beta} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial \beta} = \frac{\partial L}{\partial Y}\mathbb{1} \tag{11}$$
- X

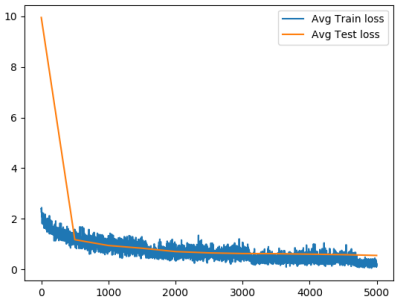
b) not implemented
c) not implemented
d) a drawback of batchnormalization is the fact that every batch will be normalized differently and the bigger the batch size the more general the normalization, thus the normalization factor is dependent on each of the entries of the batch whereas with layernormalization the normalization factor will stay constant per entry as the other axis is normalized, this may become weird when different valuerange features are used, e.g. one feature ranging from 0-400 and the rest being 0-1 will result in an odd normalization.

4 PyTorch CNN

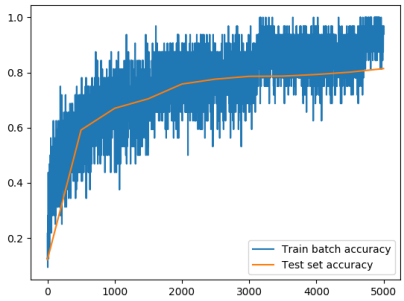
4.1 Convnet

When looking at figure 6a and 6b we can see that the CNN quickly diminishes the loss and and within very few iterations already outperforms the numpy and pytorch implementation of an MLP, this might

be partly because the CNN takes the image into account with the structural information as it is while the MLP requires the 3d data to be squished into a vector. It is an unfair comparison though, because the difference in amount of parameters is immense.



(a) The average loss per batch of the train set and average loss for the entire test set during training.

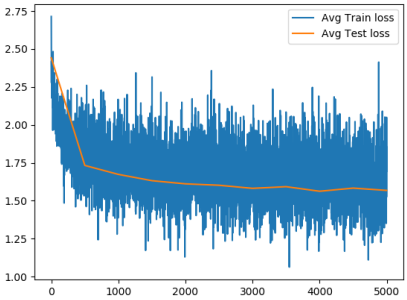


(b) The average accuracy per batch of the train set and average accuracy for the entire test set during training.

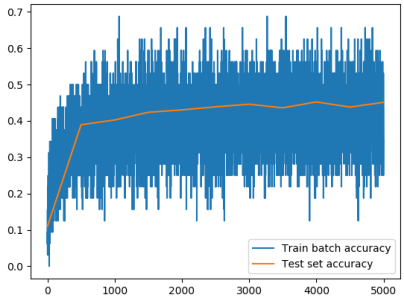
Figure 5: Training statistics of the Convnet model with standard hyperparameters

4.2 Transferlearning

We can see that the model starts of very fast but stagnates around 40% accuracy. This might be caused by having a too high learning rate. But this is very difficult to debug.



(a) The average loss per batch of the train set and average loss for the entire test set during training.



(b) The average accuracy per batch of the train set and average accuracy for the entire test set during training.

Figure 6: Training statistics of the transfer learning model, alexnet

References

References

[1] KB Petersen, MS Pedersen, et al. The matrix cookbook, vol. 7. *Technical University of Denmark*, 15, 2008.