Practical 1. MLPs, CNNs and Backpropagation

Xinyi Chen
University of Amsterdam
xinyi.chen@student.uva.nl

1 MLP backprop and NumPy Implementation

1.1 Evaluating the Gradients

1.1.1 a Linear Module

$\frac{\partial L}{\partial W_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \cdot \frac{\partial Y_{mn}}{\partial W_{ij}}$
$= \sum_{m} \frac{\partial L}{\partial Y_{mi}} \cdot \frac{\partial Y_{mi}}{\partial W_{ij}}$
$= \sum_{m} \frac{\partial L}{\partial Y_{mi}} \cdot X_{mj}$
$\frac{\partial L}{\partial W} = (\frac{\partial L}{\partial Y})^T X$

$\frac{\partial L}{\partial X_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \cdot \frac{\partial Y_{mn}}{\partial X_{ij}}$
$= \sum_{n} \frac{\partial L}{\partial Y_{in}} \cdot \frac{\partial Y_{in}}{\partial X_{ij}}$
$= \sum_{n} \frac{\partial L}{\partial Y_{in}} \cdot W_{nj}$
$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \cdot W$

$\frac{\partial L}{\partial B_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \cdot \frac{\partial Y_{mn}}{\partial B_{ij}} = \frac{\partial L}{\partial Y_{ij}} \cdot \frac{\partial Y_{ij}}{B_{ij}}$
$\frac{\partial L}{\partial b} = \vec{1}^T (\frac{\partial L}{\partial Y})$

1.1.2 b Activation Module

$Y = h(X)$
$\frac{\partial L}{\partial X} \Rightarrow \frac{\partial L}{\partial X_{ij}} = \frac{\partial L}{\partial Y_{ij}} \cdot \frac{\partial Y_{ij}}{\partial X_{ij}}$
$= \frac{\partial L}{\partial X_{ij}} \cdot \frac{\partial ELU(X_{ij})}{\partial X_{ij}}$
$= \begin{cases} \frac{\partial L}{\partial Y_{ij}} \cdot 1 & , X_{ij} \geq 0 \\ \frac{\partial L}{\partial e^{X_{ij}}} & , X_{ij} < 0 \end{cases}$

$\frac{\partial L}{\partial X} = \begin{cases} \left[\frac{\partial L}{\partial Y} \circ I\right]_{ij} & , X_{ij} \geq 0 \\ \left[\frac{\partial L}{\partial Y} \circ e^X\right]_{ij} & , X_{ij} < 0 \end{cases}$

1.1.3 c Softmax and Loss Modules

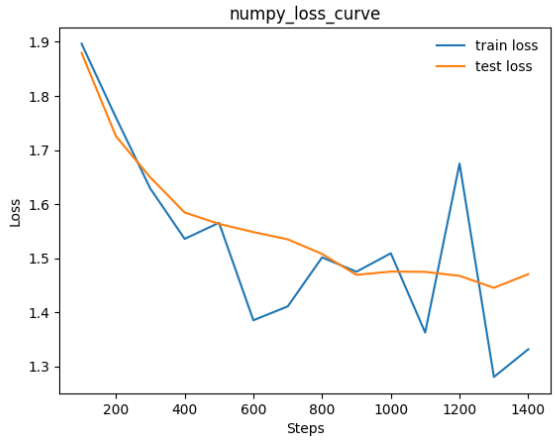
$\frac{\partial L}{\partial X_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \cdot \frac{\partial Y_{mn}}{\partial X_{ij}} = \sum_{n} \frac{\partial L}{\partial Y_{in}} \cdot \frac{\partial Y_{in}}{\partial X_{ij}}$
$= \frac{\partial L}{\partial Y_{ij}} \cdot Y_{ij} - \sum_{n} \frac{\partial L}{\partial Y_{in}} \cdot \frac{e^{X_{in}} e^{X_{ij}}}{(\sum_k e^{X_{ik}})^2}$
$= \frac{\partial L}{\partial Y_{ij}} \cdot Y_{ij} - Y_{ij} \sum_{n} \frac{\partial L}{\partial Y_{in}} \cdot Y_{in}$
$= (\frac{\partial L}{\partial Y_{ij}} - \sum_{n} \frac{\partial L}{\partial Y_{in}} \cdot Y_{in}) Y_{ij}$

$L = \frac{1}{S}\sum_i L_i = -\frac{1}{S}\sum_{ik} T_{ik}\log(X_{ik})$
$\frac{\partial L}{\partial X_{ij}} = -\frac{1}{S}\cdot\frac{\sum_{ik} T_{ik}\log(X_{ik})}{\partial X_{ij}}$
$= -\frac{1}{S}T_{ij}\cdot\frac{1}{X_{ij}}$
$= -\frac{1}{S}\cdot\frac{T_{ij}}{X_{ij}}$
$\frac{\partial L}{\partial X} = -\frac{1}{S}[T\circ X^{-1}]_{ij}$

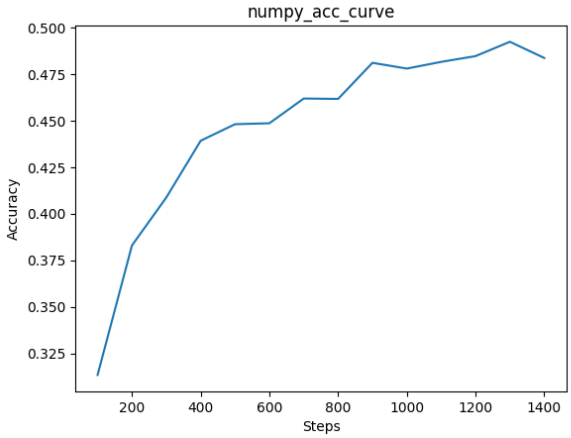
1.1.4 d Bonus

1.2 NumPy implementation

The numpy implementation gets an accuracy of 0.4838 using default paramenters(dnn-hidden-units = 100, learning-rate = 0.001, max-steps = 1400, batch-size=200, eval-freq = 100). The training and testing loss curves can be seen in Figure 1a and the accuracy curve on test set is Figure 1b.



(a) MLP Loss curves with default parameter in numpy implementation

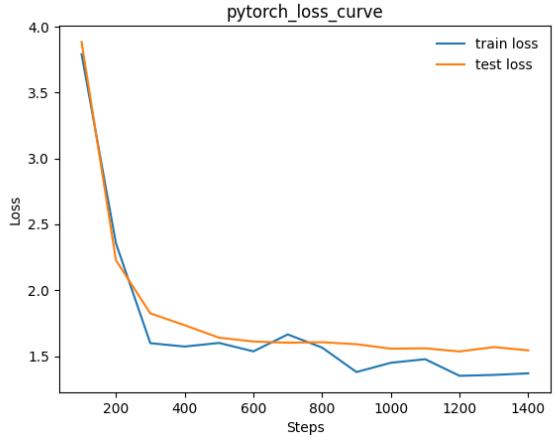


(b) MLP Loss curves with default parameter in numpy implementation

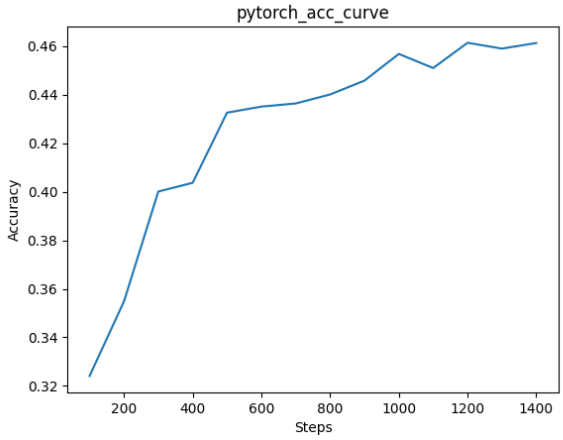
2 PyTorch MLP

2.1 Implementation and Results

Using the default parameters, the Pytorch implement gets an accuracy of 0.4613. And its loss curves are shown in Figure 2a and accuracy curve is Figure 2b.



(a) MLP Loss curves with default parameter in pytorch implementation



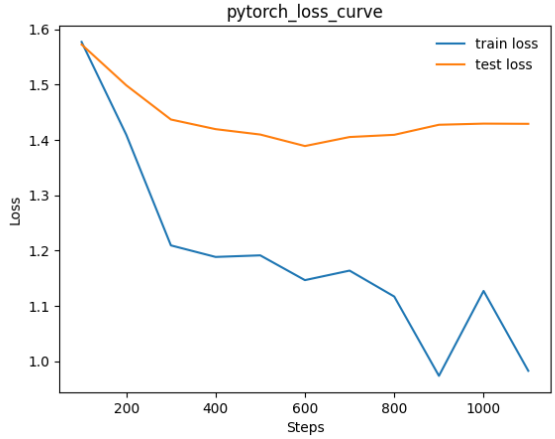
(b) MLP Loss curves with default parameter in pytorch implementation

To improve the accuracy of the MLP model, I change the number of hidden layers, max steps for training, batch size as well as use Adam optimizer and add weight decay to the optimizer. The best result is 0.5231 with the following parameters: dnn-hidden-units : 100, 100, 100, 100, 100 learning-rate : 0.001 max-steps : 1100 batch-size : 512 eval-freq : 100 weight-decay : 0.001 optimizer : Adam.

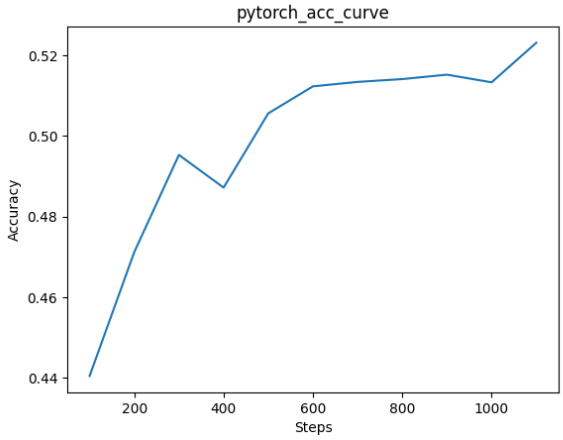
The Table 1 shows the attempts I tried to improve the performance of the network and their results.Some intuitions I used before trying out the parameters:1.Use a middle size layers, not too deep or to shallow. So I start with the number 5. 2. Replace the SGD optimizer with Adam which usually have better performance. And as Adam optimizer can automatically change the learning rate, I just keep the default learning rate. 3.As the layers go deeper, it's better to use some regularization to prevent overfitting. One way to do it is to use decay weights. 4.Choose the max steps according to the loss and accuracy curves. When the testing loss and the accuracy tends to go down, stop training. 5. Use a larger batch size.

2.2 Tanh and ELU

The benefits using Tanh is that the output is in range (-1,1) instead of (0, inf) for $x > 0$ using ELU function. So this way it won't blow up the activations. The shortcomings of using TanH is that it has vanishing gradient problems.



(a) MLP Loss curves with best parameters in pytorch implementation



(b) MLP Loss curves with best parameter in pytorch implementation

Table 1: Sample table title

number of hidden layers	learning rate	max steps	batch size	optimizer	weigh decay	accuracy
1	0.001	1400	200	SGD	/	0.4631
5	0.001	1400	200	Adam	0.001	0.4927
5	0.001	1400	200	Adam	0.01	0.4837
5	0.001	1400	512	Adam	0.001	0.5205
5	0.001	1100	512	Adam	0.001	0.5231

3 Custom Module: Layer Normalization

3.1 Automatic differentiation

The function has been implemented in the file custom_layernorm.py.

3.2 Manual implementation of backward pass

3.2.1 a

$\frac{\partial L}{\partial \gamma_i} = \sum_{s,j} \frac{\partial L}{\partial Y_{s,j}} \frac{\partial Y_{s,j}}{\partial \gamma_i}$
$= \sum_s \frac{\partial L}{\partial Y_{s,i}} \frac{\partial Y_{s,i}}{\partial \gamma_i}$

$= \sum_s \frac{\partial L}{\partial Y_{s,i}} \hat{X}_{si}$

$\frac{\partial L}{\partial \beta_i} = \sum_s \frac{\partial L}{\partial Y_{si}}$

$\frac{\partial L}{\partial X_s i} = \sum_m \sum_n \frac{\partial L}{\partial Y_{mn}} \frac{\partial Y_{mn}}{\partial X_{si}}$
$= \frac{\partial L}{\partial \hat{X}_{si}} \frac{\partial \hat{X}_{si}}{\partial X_{si}} + \sum_n \frac{\partial L}{\partial \hat{X}_{sn}} \frac{\partial \hat{X}_{sn}}{\partial \mu_s} \frac{\partial \mu_s}{\partial X_{si}} + \sum_n \frac{\partial L}{\partial \hat{X}_{sn}} \frac{\partial \hat{X}_{sn}}{\partial \sigma_s^2} (\frac{\partial \sigma_s^2}{\partial X_{si}} + \frac{\partial \sigma_s^2}{\partial \mu_s} \frac{\partial \mu_s}{\partial X_{si}})$
$= \frac{\partial L}{\partial Y_{si}} \gamma_i (\sigma_s^2 + \epsilon)^{-0.5} + \sum_n \frac{\partial L}{\partial Y_{sn}} \gamma_n (-(\sigma_s^2 + \epsilon)^{-0.5}) \frac{1}{M}$
$+ \sum_n \frac{\partial L}{\partial Y_{sn}} \gamma_n [-0.5(X_{sn} - \mu_s)(\sigma_s^2 + \epsilon)^{-1.5}][\frac{2}{M}(X_{si} - \mu_s) + 0]$
$= \frac{1}{M(\sigma_s^2 + \epsilon)^{-0.5}} (M \frac{\partial L}{\partial Y_{si}} \gamma_i - \sum_n \frac{\partial L}{\partial Y_{sn}} \gamma_n - \hat{X}_{si} \sum_n \frac{\partial L}{\partial Y_{sn}} \gamma_n \hat{X}_{sn})$

3.2.2 b

The function has been implemented in the file custom_layernorm.py.

3.3 c

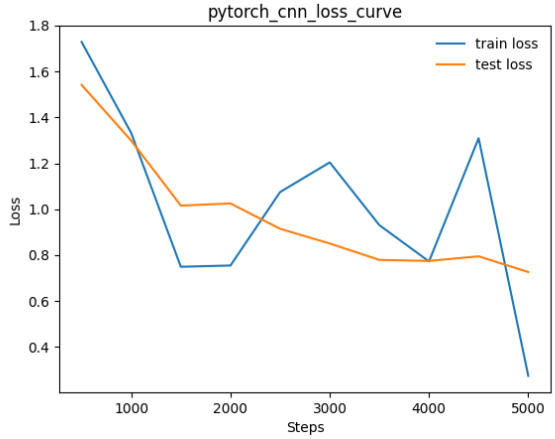
The function has been implemented in the file custom_layernorm.py.

3.4 d

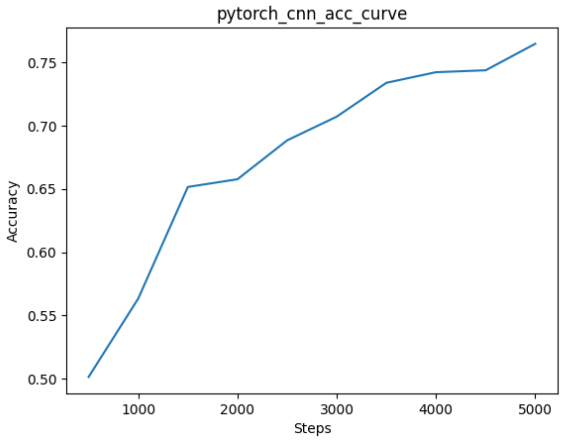
Layer Normalization and Batch Normalization

4 PyTorch CNN

The Pytorch CNN function has an accuracy of 0.7983 using default parameters. The training and testing loss curves can be seen in Figure 4a and the accuracy curve on test set is Figure 4b.



(a) CNN Loss curves with default parameter in pytorch implementation



(b) CNN Loss curves with default parameter in pytorch implementation