Assignment 1: MLPs, CNNs and Backpropagation

Volodymyr Medentsiy
id: 12179078
volodymyr.medentsiy@student.uva.nl

1 MLP backprop and NumPy implementation

1.1 Analytical derivation of gradients

Compute gradients given $x^l \in R^{d_l}$ and:

$$L(x^{(N)}, t) = -\sum_{i=1}^{d_N} t_i * log(x_i^N) = -log(x^N_{argmax(t)}) \in R$$

$$x^N = softmax(\tilde{x}^N) \in R^{d_N*1}$$

$$x^l = ReLU(\tilde{x}^l) \in R^{d_l*1} \quad \forall l = 1, ..., N-1$$

$$\tilde{x}^l = W^l * x^{l-1} + b^l \in R^{d_l*1} \quad \forall l = 1, ..., N-1$$

Question 1.1.A

$$\textbf{1)}\frac{\partial L}{\partial x^{(N)}} \in R^{1*d_N}, \quad \text{with} \quad (\frac{\partial L}{\partial x^{(N)}})_i = \begin{cases} 0, \text{ if } i \neq argmax(t) \\ \frac{-1}{x^N_{argmax(t)}}, \text{ if } i = argmax(t) \end{cases}$$

$$\textbf{2)}\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} \in R^{d_N*d_N}, \quad \text{with}(\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}})_{i,j} = \begin{cases} \frac{exp(\tilde{x}_i^N)}{\sum k=1^{d_N} exp(\tilde{x}_k^N)} - (\frac{exp(\tilde{x}_i^N)}{\sum k=1^{d_N} exp(\tilde{x}_k^N)})^2 = softmax(\tilde{x}_i^N) - (softmax(\tilde{x}_i^N))^2, \text{ if } i = j \\ -\frac{exp(\tilde{x}_i^N)*exp(\tilde{x}_j^N)}{(\sum k=1^{d_N} exp(\tilde{x}_k^N))^2} = -softmax(\tilde{x}_i^N) * softmax(\tilde{x}_j^N), \text{ if } i \neq j \end{cases}$$

$$\textbf{3)}\frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}} \in R^{d_l*d_l}, \quad \text{with} \quad (\frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}})_{i,j} = \begin{cases} 1[\tilde{x}_i^{(l)} > 0], \text{ if } i = j \\ 0, \text{ if } i \neq j \end{cases}$$

$$\textbf{4)}\frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}} = W^{l+1} \in R^{d_{l+1}*d_l}, \quad \text{thus} \quad (\frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}})_{i,j} = w_{ij}^{l+1}$$

$$\textbf{5)}\frac{\partial \tilde{x}^{(l)}}{\partial W^l} \in R^{d_l*d_l*d_{l-1}}, \quad \text{with} \quad (\frac{\partial \tilde{x}^{(l)}}{\partial W^l})_{i,j,k} = \begin{cases} x_k^{l-1}, \text{ if } i = j \\ 0, \text{ if } i \neq j \end{cases}$$

$$\textbf{6)}\frac{\partial \tilde{x}^{(l)}}{\partial b^l} \in R^{d_l*d_l}, \quad \text{with} \quad (\frac{\partial \tilde{x}^{(l)}}{\partial b^l})_{i,j} = \begin{cases} 1, \text{ if } i = j \\ 0, \text{ if } i \neq j \end{cases}$$

Submitted to the Spring 2019 Amsterdam Deep Learning course

Question 1.1.B In the following equations $\cdot$ - denotes scalar product between two vectors.

$$\mathbf{1)} \frac{\partial L}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} * \frac{\partial x^N}{\partial \tilde{x}^{(N)}} \in R^{1*d_N}, \quad \text{with} \quad (\frac{\partial L}{\partial \tilde{x}^{(N)}})_i = \frac{\partial L}{\partial x^{(N)}} \cdot (\frac{\partial x^N}{\partial \tilde{x}^{(N)}})_{:,i} = \frac{-1}{x^N_{argmax(t)}} * \frac{\partial x^N_{argmax(t)}}{\partial \tilde{x}^{(N)}_i}$$

$$\mathbf{2)} \frac{\partial L}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial x^{(l)}} * \frac{\partial x^l}{\partial \tilde{x}^{(l)}} \in R^{1*d_l}, \quad \text{with} \quad (\frac{\partial L}{\partial \tilde{x}^{(l)}})_i = \frac{\partial L}{\partial x^{(l)}} \cdot (\frac{\partial x^l}{\partial \tilde{x}^{(l)}})_{:,i} = \begin{cases} \frac{\partial L}{\partial x^{(l)}_i}, \text{ if } \tilde{x}^{(l)}_i > 0 \\ 0, \text{ if } \tilde{x}^{(l)}_i \leq 0 \end{cases}$$

$$\mathbf{3)} \frac{\partial L}{\partial x^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} * \frac{\partial \tilde{x}^{l+1}}{\partial \tilde{x}^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} * W^{l+1} \in R^{1*d_l}, \quad \text{thus} \quad (\frac{\partial L}{\partial x^{(l)}})_i = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} \cdot W^{l+1}_{:,i}$$

$$\mathbf{4)} \frac{\partial L}{\partial W^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} * \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} \in R^{d_l * d_{l-1}}, \quad \text{with} \quad (\frac{\partial L}{\partial W^{(l<N)}})_{j,k} = \sum_{i=1}^{d_l} (\frac{\partial L}{\partial \tilde{x}^{(l)}})_i * (\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}})_{i,j,k} = (\frac{\partial L}{\partial \tilde{x}^{(l)}})_j * (\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}})_{j,j,k} = (\frac{\partial L}{\partial \tilde{x}^{(l)}})_j * x^{l-1}_k$$

$$\mathbf{5)} \frac{\partial L}{\partial b^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} * \frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} * \mathbb{I} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \in R^{1*d_l}, \quad \text{thus} \quad (\frac{\partial L}{\partial b^{(l<N)}})_i = \frac{\partial L}{\partial \tilde{x}^{(l)}}_i$$

Question 1.1.C When we compute gradients for mini-batches, we accumulate gradients for each data point in a batch and then average them, thus we obtain the mean estimate of the gradient. So for example:

$$\frac{\partial L_{total}}{\partial \tilde{x}^{(N)}} = \frac{1}{B} \sum_{s=1}^{B} \frac{\partial L_{individual}(x^{N,s}, t^s)}{\partial x^{(N)}} * \frac{\partial x^{N,s}}{\partial \tilde{x}^{(N)}}$$

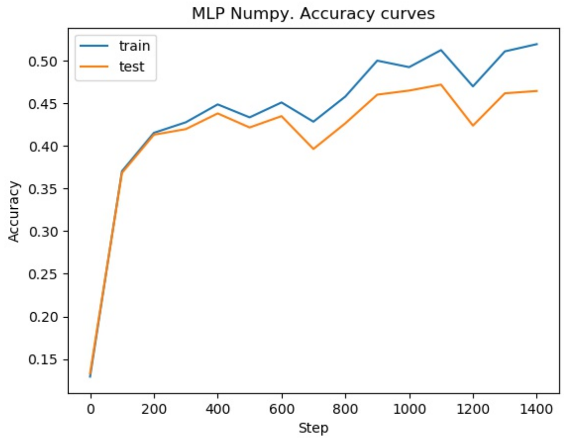
1.2 NumPy Implementation

The implementation of MLP using Numpy library could be found in the mlp_numpy.py and train_mlp_numpy.py file. The accuracy and the loss of the MLP trained with default parameters(1 hidden layer of size 100, batch size = 200, max number of steps = 1500 and SGD optimizer with learning rate = 2e-3) are reported in Figure 1. The accuracy and loss curves have the same behaviour on the train and test data sets, so increase or decrease in accuracy (loss) on the train data set corresponds to increase or decrease on the test data respectively. The maximum achieved accuracy during training on the train data set is 0.52, on the test set is 0.47. The minimum achieved loss during training is 1.5 and 1.4 for the train and test data set respectively.

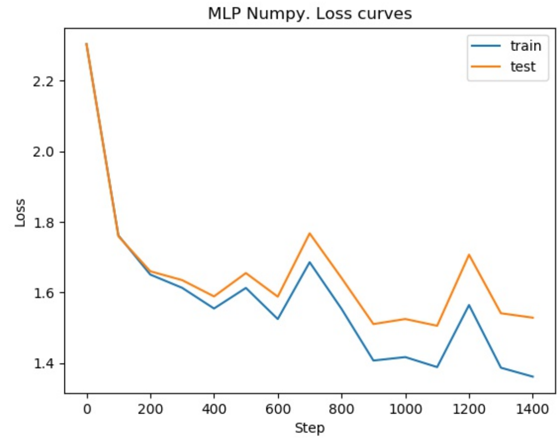
2 PyTorch MLP

The implementation of MLP in Pytorch could be found in the mlp_pytorch.py and train_mlp_pytorch.py file. The accuracy and the loss of the MLP trained with default parameters(batch size = 200, max number of steps = 1500 and SGD optimizer with learning rate = 2e-3) are reported in Figure 2. As in the previous case accuracy and loss have the same behaviour on the train and test data. Interestingly, loss curves start with very high values. This could be caused by initialization strategies, which differs from numpy implementation. The maximum accuracy on the training set is 0.48, on the test set is 0.44. The minimum achieved loss is 1.5 and 1.6 for the train and test data set respectively.

To increase the performance of the MLP, first I changed the optimizer to Adam, which gave 0.02 increase in accuracy on the test data set. Secondly, I made network deeper (4 layers of size 300, 200, 100 and 50) and changed the batch size to 264, which helped to increase accuracy up to 0.516 on the test data. After this, I lowered the learning rate to 1e-4, set weight decay to 0.1 and increased number of iteration to 3000. Unfortunately, this hyperparameters setting did not improve the accuracy, but introducing weight decay had regularizing effect. Such learning rate could have been too low, so I

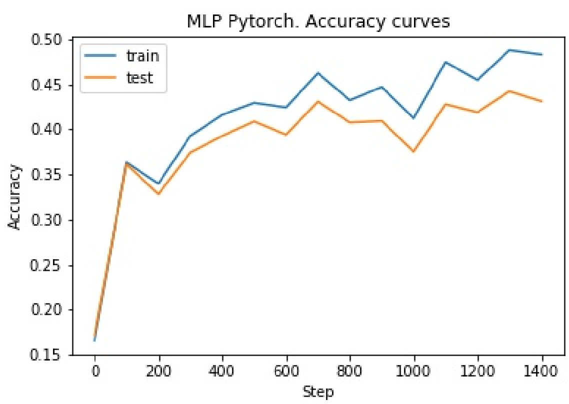


(a) Accuracy

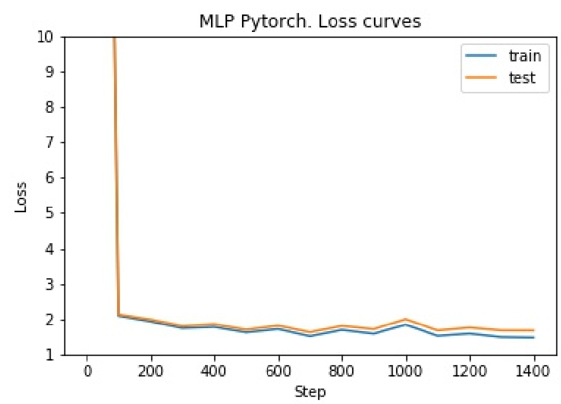


(b) Loss

Figure 1: Performance regarding iterations of the MLP (numpy implementation) with default parameters



(a) Accuracy

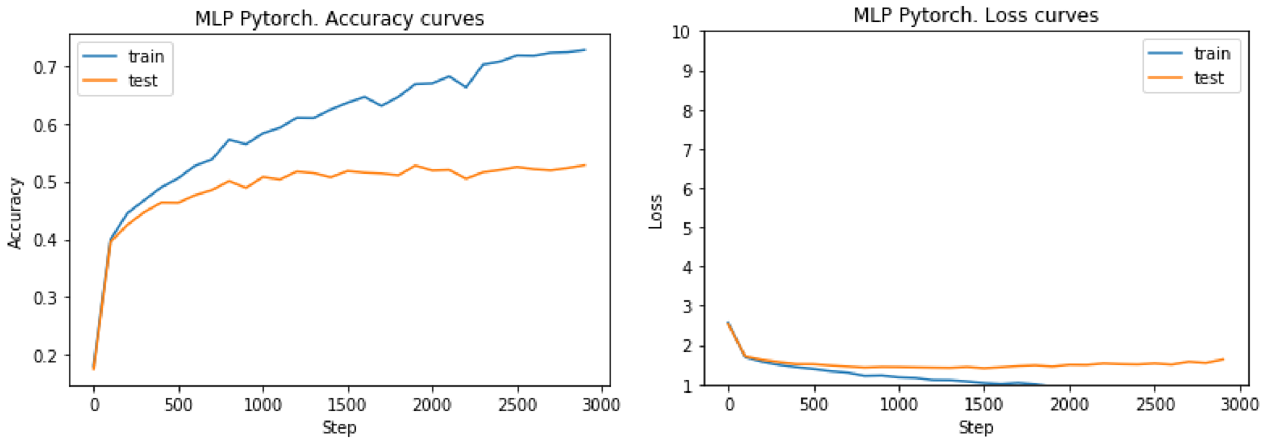


(b) Loss

Figure 2: Performance regarding iterations of the MLP (pytorch implementation) with default parameters

increased it to 1e-3, and because the goal was to achieve 0.52 accuracy on the test data irrespective to the performance on the training data set and possible overfitting, I neglected weight decay. MLP with the final hyperparameter settings has 0.527 accuracy on the test data and the 0.7 on the training data set, which is indicative of the overfitting. The loss and accuracy curves for the final hyperparameter setting are represented on Figure 3. The best accuracy achieved during training on the train and test data sets for different hyperparameters are described in the Table .

Hidden layers	Optimizer	Learning rate	Weight decay	Number of iterations	Batch size	Test accuracy	Train accuracy
100	SGD	2e-3	-	1500	200	0.44	0.48
100	Adam	2e-3	-	1500	200	0.46	0.52
300, 200, 100, 50	Adam	2e-3	-	1500	264	0.516	0.6
300, 200, 100, 50	Adam	1e-4	0.01	3000	264	0.509	0.54
300, 200, 100, 50	Adam	1e-3	-	3000	264	0.527	0.726



(a) Accuracy (b) Loss

Figure 3: Performance regarding iterations of the MLP (pytorch implementation) with the best parameters

3 Custom Module: Batch Normalization

3.1 Automatic differentiation

The implementation of Batch normalization as nn.Module could be found in the custom_batchnorm.py file.

3.2 Manual implementation of backwards pass

Question 3.2.A In the following equations $\delta_{ij} = 1 \quad \text{if i = j} \quad$, and 0 otherwise.

$$\mathbf{1)} \frac{\partial L}{\partial \gamma} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \gamma}, \quad \text{with} \quad (\frac{\partial L}{\partial \gamma})_j = \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \gamma_j} = \left[\frac{\partial y_i^s}{\partial \gamma_j} = \tilde{x}_i^s * 1[i == j]\right] = \sum_s \frac{\partial L}{\partial y_j^s} * \tilde{x}_j^s$$

$$\mathbf{2)} \frac{\partial L}{\partial \beta} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \beta}, \quad \text{with} \quad (\frac{\partial L}{\partial \beta})_j = \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \beta_j} = \left[\frac{\partial y_i^s}{\partial \beta_j} = 1[i == j]\right] = \sum_s \frac{\partial L}{\partial y_j^s}$$

$$\mathbf{3)} \frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}, \quad \text{with} \quad (\frac{\partial L}{\partial x})_j^r = \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial x_j^r} = \left[\frac{\partial y_i^s}{\partial x_j^r} = 0 \quad \text{if} \quad j \neq i\right] = \sum_s \frac{\partial L}{\partial y_j^s}\frac{\partial y_j^s}{\partial x_j^r} = \sum_s \frac{\partial L}{\partial y_j^s}\frac{\partial(\gamma_j \frac{x_j^s - \mu_j}{\sqrt{\sigma_j^2+\epsilon}} + \beta_j)}{\partial x_j^r} =$$

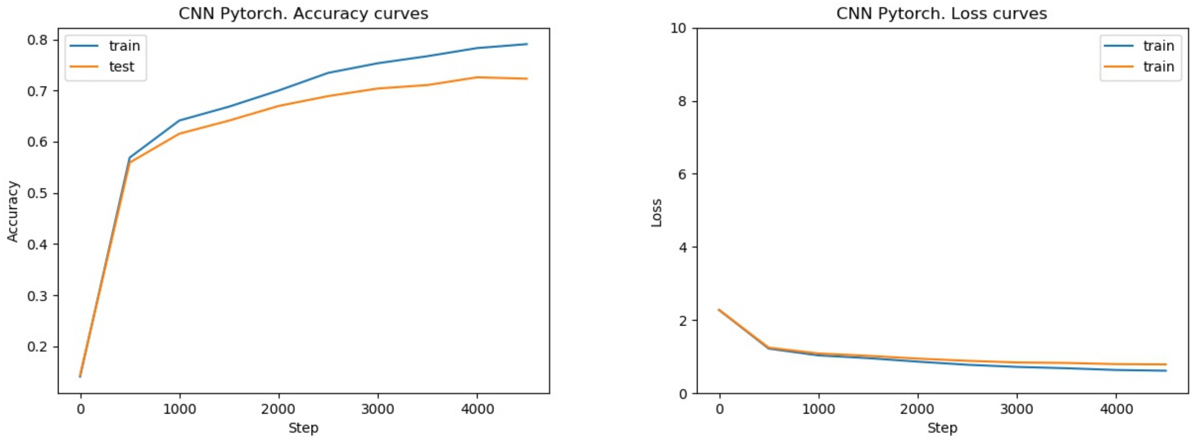
$$= \sum_s \frac{\partial L}{\partial y_j^s} * \left(\frac{\partial(\gamma_j(x_j^s - \mu_j))}{\partial x_j^r} * \frac{1}{\sqrt{\sigma_j^2+\epsilon}} + \gamma_j(x_j^s - \mu_j)\frac{\partial \frac{1}{\sqrt{\sigma_j^2+\epsilon}}}{\partial x_j^r}\right) = \sum_s \frac{\partial L}{\partial y_j^s} * \left(\gamma_j(\delta_{rs} - \frac{1}{B}) * \frac{1}{\sqrt{\sigma_j^2+\epsilon}} + \gamma_j(x_j^s - \mu_j) * (\frac{\partial(\sigma_j^2+\epsilon)^{\frac{-1}{2}}}{\partial x_j^r})\right),$$

$$\text{with} \quad \frac{\partial(\sigma_j^2+\epsilon)^{\frac{-1}{2}}}{\partial x_j^r} = \frac{-1}{2}\left(\frac{1}{B}\sum_k((x_j^k - \mu_j)^2 + \epsilon)\right)^{\frac{-3}{2}} * \frac{1}{B}\sum_k(2(x_j^k - \mu_j)(\delta rk - \frac{1}{B}) =$$

$$= \frac{-1}{B(\sigma_j^2+\epsilon)^{\frac{3}{2}}} * \left((x_j^r - \mu_j) - \frac{1}{B}\sum_k x_j^k + \mu_j\right) = -\frac{1}{B(\sigma_j^2+\epsilon)^{\frac{3}{2}}} * (x_j^r - \mu_j).$$

So finally :

$$(\frac{\partial L}{\partial x})_j^r = \sum_s \frac{\partial L}{\partial y_j^s} * \gamma_j\left(\frac{\delta_{rs} - \frac{1}{B}}{\sqrt{\sigma_j^2+\epsilon}} - \frac{(x_j^s - \mu_j)(x_j^r - \mu_j)}{B(\sigma_j^2+\epsilon)^{\frac{3}{2}}}\right)$$



(a) Accuracy

(b) Loss

Figure 4: Performance regarding iterations of the CNN with default parameters

Question 3.2.B The class CustomBatchNormManualFunction() could be found in the custom_batchnorm.py file. To compute the gradient $\frac{\partial L}{\partial x}$ I have used formulas in Ioffe and Szegedy [2015], which decomposes the gradient into sum of $\frac{\partial L}{\partial \sigma^2}$, $\frac{\partial L}{\partial \mu}$ and $\frac{\partial L}{\partial \hat{x}}$.
Question 3.2.C The class CustomBatchNormManualModule() could be found in the custom_batchnorm.py file.

4 PyTorch CNN

The implementation of CNN could be found in the convnet_pytorch.py and train_convnet_pytorch.py file. Every convolutional layer is followed by batch normalization layer and then by ReLU layer. The accuracy and loss are reported in Figure 4. The highest accuracy achieved on the training data is 0.79, on the test data - 0.73. As expected CNN substantially outperforms MLP.

References

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 2015. URL `https://arxiv.org/abs/1502.03167`.