Deep Learning Assignment 1. MLPs, CNNs and Backpropagation

Andrii Skliar
11636785
University of Amsterdam
andrii.skliar@student.uva.nl

1 MLP backprop and NumPy implementation

1.1 Analytical derivation of gradients

Question 1.1 a)

Note, that in the following derivations, following notation was used:

1. $S_i = \frac{\left(\exp \tilde{x}_i^{(N)}\right)}{\sum_{k=1}^{d_N} \exp \tilde{x}_k^{(N)}}$
2. W_{i*} stands for the whole row i in matrix W
3. $diag(x)$ stands for a square diagonal matrix that has elements of x in the main diagonal
4. $\otimes$ stands here for "expanding" Kronecker product, i.e. each new matrix will be put in a new dimension.

Cross-entropy

$$\begin{aligned}\left(\frac{\partial L}{\partial x^{(N)}}\right)_i &= \left(\frac{\partial L}{\partial x_i^{(N)}}\right) \\ &= \frac{\partial(-\sum_i t_i \log x_i^{(N)})}{\partial x_i^{(N)}} \\ &= -\frac{t_i}{x_i^{(N)}}\end{aligned}$$

$$\left(\frac{\partial L}{\partial x^{(N)}}\right) = -\mathbf{t} \circ \mathbf{x}^{(N)^{-1}} = \begin{pmatrix} t_1 \\ \vdots \\ t_N \end{pmatrix} \circ \begin{pmatrix} \frac{1}{x_1^{(N)}} \\ \vdots \\ \frac{1}{x_N^{(N)}} \end{pmatrix}$$

Softmax

$$\begin{aligned}
\left(\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}}\right)_{ij} &= \left(\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}}\right) \\
&= \frac{\partial\left(\frac{\exp \tilde{x}_i^{(N)}}{\sum_{k=1}^{d_N} \exp \tilde{x}_k^{(N)}}\right)}{\partial \tilde{x}_j^{(N)}} \\
&= \begin{cases} \frac{\left(\exp \tilde{x}_i^{(N)}\right)}{\sum_{k=1}^{d_N} \exp \tilde{x}_k^{(N)}} - \frac{\left(\exp \tilde{x}_i^{(N)}\right)^2}{(\sum_{k=1}^{d_N} \exp \tilde{x}_k^{(N)})^2} & \text{if } i = j \\ -\frac{\left(\exp \tilde{x}_i^{(N)}\right)\left(\exp \tilde{x}_j^{(N)}\right)}{(\sum_{k=1}^{d_N} \exp \tilde{x}_k^{(N)})^2} & \text{if } i \neq j \end{cases} \\
&= \begin{cases} S_i(1 - S_j) & \text{if } i = j \\ -S_i S_j & \text{if } i \neq j \end{cases} \\
&= S_i(\delta_{ij} - S_j) \\
&= x_i^{(N)}(\delta_{ij} - x_j^{(N)}) \\
\left(\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}}\right) &= diag(x^{(N)}) - x^{(N)} x^{(N)^T}
\end{aligned}$$

ReLU

$$\begin{aligned}
\left(\frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}}\right) &= \frac{\partial max(0, \tilde{x}^{(l)})}{\partial \tilde{x}^{(l)}} \\
&= diag(\mathbb{1}_{\tilde{x}^{(l)} > 0})
\end{aligned}$$

Linear

$$\begin{aligned}
\left(\frac{\partial \tilde{x}^{(l)}}{\partial x^{(l-1)}}\right) &= \left(\frac{\partial\left(W^{(l)} x^{(l-1)} + b^{(l)}\right)}{\partial x^{(l-1)}}\right) \\
&= W^{(l)} \\
\left(\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}}\right)_{ijk} &= \left(\frac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^{(l)}}\right) \\
&= \left(\frac{\partial\left(W_{i*}^{(l)} x^{(l-1)} + b_i^{(l)}\right)}{\partial W_{jk}^{(l)}}\right) \\
&= \begin{cases} 0, & \text{if } i \neq j \\ x_k^{(l-1)}, & \text{if } i = j \end{cases} \\
&= x^{(l-1)} \otimes \mathbf{I}
\end{aligned}$$

Last expression means that derivative will be a 3d tensor with diagonal matrix $x_i\mathbf{I}$ in first two dimensions and having $i = 1 \dots k$ in the third dimension.

$$\begin{aligned}
\left(\frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}}\right) &= \left(\frac{\partial\left(W^{(l)} x^{(l-1)} + b^{(l)}\right)}{\partial b^{(l)}}\right) \\
&= \mathbf{I}
\end{aligned}$$

Question 1.1 b)

$$
\begin{aligned}
\left(\frac{\partial L}{\partial \tilde{x}^{(N)}}\right) &= \frac{\partial L}{\partial x^{(N)}} \frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} \\
&= \frac{\partial L}{\partial x^{(N)}} \left(diag(x^{(N)}) - x^{(N)} x^{(N)^T} \right) \\
\left(\frac{\partial L}{\partial \tilde{x}^{(l<N)}}\right) &= \frac{\partial L}{\partial x^{(l)}} \frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}} \\
&= \frac{\partial L}{\partial x^{(l)}} diag(\mathbb{1}_{\tilde{x}^{(l)}>0}) \\
\left(\frac{\partial L}{\partial \tilde{x}^{(l<N)}}\right) &= \frac{\partial L}{\partial \tilde{x}^{(l+1)}} \frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}} \\
&= \frac{\partial L}{\partial \tilde{x}^{(l+1)}} W^{(l+1)} \\
\left(\frac{\partial L}{\partial W^{(l)}}\right) &= \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} \\
&= \sum_i \frac{\partial L}{\partial \tilde{x}_i^{(l)}} \frac{\partial \tilde{x}_i^{(l)}}{\partial W^{(l)}} \\
&= \frac{\partial L}{\partial \tilde{x}^{(l)}} x^{(l-1)^T} \\
\left(\frac{\partial L}{\partial b^{(l)}}\right) &= \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} \\
&= \frac{\partial L}{\partial \tilde{x}^{(l)}} \mathbf{I} \\
&= \frac{\partial L}{\partial \tilde{x}^{(l)}}
\end{aligned}
$$

Question 1.1 c)

If a batch size $B > 1$ is used, cross-entropy loss will be averaged over the batches to get a single scalar as a result for the whole batch. However, due to an additional dimension, corresponding to the batch size, backpropagation would also have to account for this, expanding its dimensionality by 1 and resulting in each layer l backpropagating matrix of dimensionality Bxd_{l-1}.

As far as each feature is independent of all the other features in the same vector in a batch, gradients can be computed for each feature vector separately using previously derived equations. However, for weights of the linear layer, we would also need to sum over the batch dimension to take into account the influence of each feature vector on the final value. This is due to the fact, that in the case of linear layer, features in a vector (each feature corresponds to a single neuron) are actually not independent and influence output of each neuron in the current layer.

Question 1.2

2 PyTorch MLP

Question 2

To improve the accuracy of the model, multiple things were tried:

1. **Changing the optimizer**: multiple papers [5] [4] have suggested that SGD optimizer can find better optimum compared to other optimizers. However, saying that, we have to notice that SGD optimizer might not be able to converge to an optimal solution and Adam optimizer yields better results. Some papers were trying to improve performance of SGD introducing momentum. However, the main issue still remains in that SGD needs better momentum and learning rate tuning or introduction of adaptive methods [6] for doing that compared to other

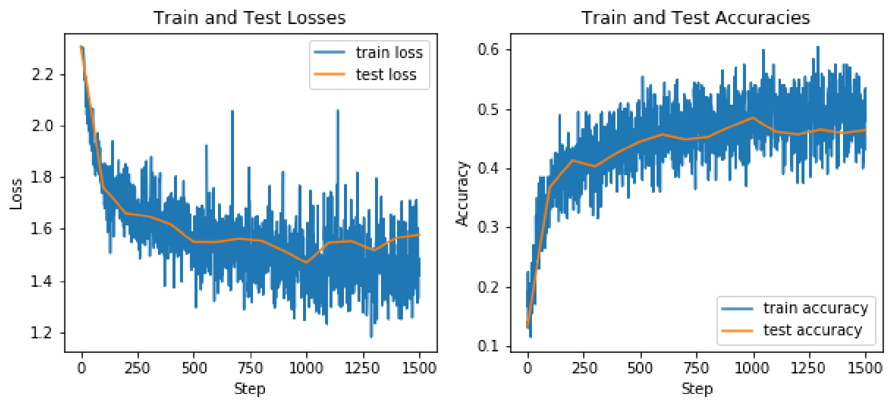

Figure 1: NumPy MLP implementation loss and accuracy curves

optimization methods. In our case, we have found out Adam to work much better even after trying to find optimal learning rate for SGD and using momentum of 0.9. Also, we have tried using AdaGrad, but it wasn't giving any increase in accuracy compared to Adam.

2. **Changing the architecture**: First, to improve the performance of the model, we have tried stacking up to 4 layers with "bottleneck" approach where we would start with layers of larger dimensionality and gradually project it onto the lower-dimensional space i.e. $(2048, 1024, 512, 100)$. Though this approach was giving good results, it was not generalizing well, outputting really high accuracy of up to 90% on the training set and only 58% on the test set even after introducing regularization. Also, this architecture would take much longer to train. However, to tackle aforementioned problem, we have decided to try models with less layers, namely, the best results were achieved with model having following architecture: $(512, 512)$. This architecture is more trainable and also proved to generalize much better, so though it has a bit lower accuracy of 57.7%

3. **Changing learning rate and batch size**: Often learning rate should be changed together with batch size which is why we were changing learning rate based not only on the optimizer, but also batch size. As for the batch size, we have tried sizes of $(200, 500, 1000, 2000, 4000)$ together with learning rates of $(0.1, 0.01, 0.001, 0.0001, 0.00001)$ and found out that higher learning rate with larger batch size in our case would still yield worse performance, which contradicts existing papers [3] [1]. However, this might be due to the fact that we need to set up learning rate scheduler in order to use approach described in the aforementioned papers, which we considered to be excessive for this homework. In the end, the best parameters turned out to be $batch_size = 200$ and $lr = 0.0001$. Also, the reason for choosing smaller batch size was that with larger batch size, model was overfitting.

4. **Changing activation function**: We have tried using SELU as suggested in [2] in order to reduce overfitting and generally increase accuracy of our model. However, we didn't get any improvement in terms of accuracy. What was suprising is that while using weight initialization that was suggested in the paper, our model was performing worse than when using Kaiming uniform initialization, which is default for linear layers in PyTorch. Also, weirdly, SELU didn't reduce overfitting in our case so it was decided not to use in the end. Due to all the mentioned issues, in the end we decided to use $ReLU$ as it was giving slightly better results.

5. **Adding regularization**: We have used multiple regularization methods, namely dropout and weight decay. While dropout of 0.1 didn't improve the performance, dropout of 0.2 itself has increased accuracy by 2%, however, it didn't help with overfitting and MLP was still not generalizing well enough. Weight decay was used as L2 regularization with values of $(0.01, 0.001, 0.0001, 0.0005, 0.00001)$ tried. This also didn't help to reduce overfitting, however, slightly improving the results. The best value for weight decay was found to be 0.0001.

6. **Adding batch normalzation**: To reduce overfitting, before each activation function, batch normalization was added. It has led to slightly better performance, however, its main

contribution is that it has significantly decreased overfitting, as you can see in fig. 2. Before that, training set accuracy would reach 90%.

So, the best parameters are:

$$\begin{aligned}
\text{batch size} &= 200 \\
\text{learning rate} &= 0.0001 \\
\text{optimizer} &= Adam \\
\text{weight decay} &= 0.0001 \\
\text{dropouts} &= (0.2, 0.2) \\
\text{hidden layers} &= (512, 512)
\end{aligned}$$

With these, model has reached an accuracy of 57.67%.

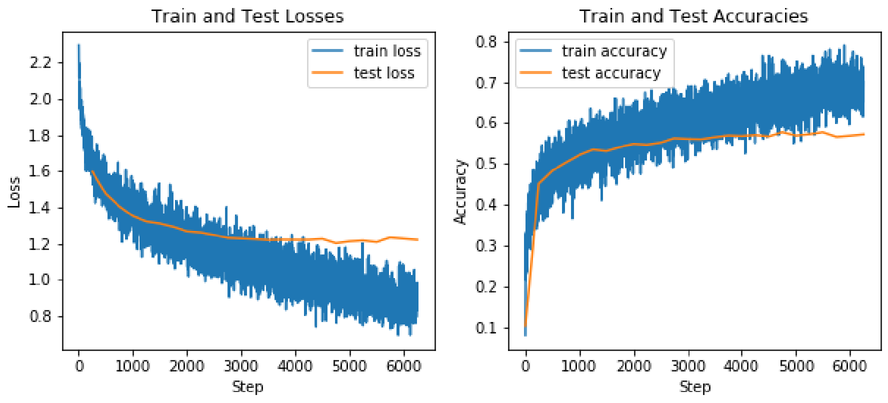

Figure 2: Pytorch MLP implementation loss and accuracy curves

2.1 Custom Module: Batch Normalization

3.2 Manual implementation of backward pass

Question 3.2 a)

$$\begin{aligned}
\left(\frac{\partial L}{\partial \gamma}\right)_j &= \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial \gamma_j} \\
&= \sum_s \frac{\partial L}{\partial y_j^s} \circ \hat{x}_j \\
\left(\frac{\partial L}{\partial \beta}\right)_j &= \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial \beta_j} \\
&= \sum_s \frac{\partial L}{\partial y_j^s} \cdot 1 \\
&= \sum_s \frac{\partial L}{\partial y_j^s}
\end{aligned}$$

$$
\begin{aligned}
\frac{\partial L}{\partial x_i^r} &= \sum_s \frac{\partial L}{\partial \hat{x}_i^s}\frac{\partial \hat{x}_i^s}{\partial \mu_i}\frac{\partial \mu_i}{\partial x_i^r} + \frac{\partial L}{\partial \hat{x}_i^r}\frac{\partial \hat{x}_i^r}{\partial x_i^r} + \sum_s \frac{\partial L}{\partial \hat{x}_i^s}\frac{\partial \hat{x}_i^s}{\partial \sigma_i^2}\left[\frac{\partial \sigma_i^2}{\partial x_i^r} + \frac{\partial \sigma_i^2}{\partial \mu_i}\frac{\partial \mu_i}{\partial x_i^r}\right] \\
\frac{\partial L}{\partial \hat{x}_i^s} &= \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \hat{x}_i^s} \\
&= \frac{\partial L}{\partial y_i^s}\gamma \\
\frac{\partial \hat{x}_i^s}{\partial x_i^s} &= \frac{1}{\sqrt{\sigma^2+\epsilon}} \\
\frac{\partial \hat{x}_i^s}{\partial \mu_i} &= -\frac{1}{\sqrt{\sigma^2+\epsilon}} \\
\frac{\partial \mu_i}{\partial x_i^s} &= \frac{1}{B} \\
\frac{\partial \sigma_i^2}{\partial \mu_i} &= -\frac{2}{B}\sum_s (x_i^s-\mu_i) \\
&= -2\left(\frac{1}{B}\sum_s x_i^s - \frac{1}{B}\cdot B\cdot \mu_i\right) \\
&= -2\left(\mu_i-\mu_i)\right) \\
&= 0 \\
\frac{\partial \sigma_i^2}{\partial x_i^s} &= \frac{2}{B}(x_i^s-\mu_i) \\
\frac{\partial \hat{x}_i^s}{\partial \sigma_i^2} &= -\frac{1}{2}(x_i^s-\mu_i)(\sigma_i^2+\epsilon)^{-\frac{3}{2}} \\
\frac{\partial L}{\partial x_i^r} &= \left[\sum_s\left(-\frac{1}{\sqrt{\sigma_i^2+\epsilon}}\frac{\partial L}{\partial \hat{x}_i^s}\frac{1}{B}\right) + \frac{\partial L}{\partial \hat{x}_i^r}\frac{1}{\sqrt{\sigma_i^2+\epsilon}} + \sum_s\left(\frac{\partial L}{\partial \hat{x}^i}\left(-\frac{1}{2}(x_i^s-\mu_i)(\sigma_i^2+\epsilon)^{-\frac{3}{2}}\right)\left(\frac{2}{B}(x_i^r-\mu_i)\right)\right)\right] \\
&= \left\{\text{Using that}\,\frac{x_i^s-\mu_i}{\sqrt{\sigma_i^2+\epsilon}} = \hat{x}_i^s \text{ and } (\sigma_i^2+\epsilon)^{-\frac{3}{2}} = \left(\frac{1}{\sqrt{\sigma_i^2+\epsilon}}\right)^3\right\} \\
&= \frac{1}{B\sqrt{\sigma_i^2+\epsilon}}\left[-\sum_s \frac{\partial L}{\partial \hat{x}_i^s} + B\frac{\partial L}{\partial \hat{x}_i^r} - \hat{x}_i^r\sum_s \frac{\partial L}{\partial \hat{x}_i^s}\hat{x}_i^s\right]
\end{aligned}
$$

Question 4

After training mini-VGG on the Cifar-10 and plotting it in fig. 3, we can see, that, compared to MLP model, VGG has much higher accuracy (81.1% against 57.7% for MLP), lower loss (0.55 against 1.22 for MLP) and also doesn't overfit as much. This might be due to the fact, that for images, convolutional neural networks are proved to generalize much better that fully connected ones. Also, as you can see, network performance was still increasing even after running it for 50 epochs, while performance of MLP was stabilizing even after 10 epochs and would not improve any further.

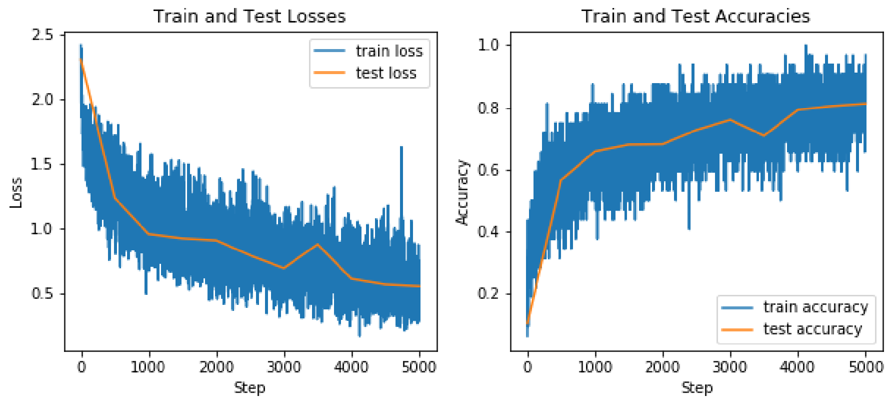

Figure 3: Convnet loss and accuracy curves

References

[1] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[2] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. *CoRR*, abs/1706.02515, 2017.

[3] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. *CoRR*, abs/1711.00489, 2017.

[4] Samuel L. Smith and Quoc V. Le. A bayesian perspective on generalization and stochastic gradient descent. *CoRR*, abs/1710.06451, 2017.

[5] Chen Xing, Devansh Arpit, Christos Tsirigotis, and Yoshua Bengio. A walk with sgd. *CoRR*, abs/1802.08770, 2018.

[6] Jian Zhang, Ioannis Mitliagkas, and Christopher Ré. Yellowfin and the art of momentum tuning. *arXiv preprint arXiv:1706.03471*, 2017.