Deep Learning Assignment 1

Jochem Loedeman
12995282

1 MLP backprop and NumPy Implementation

1.1 Evaluating the Gradients

Question 1.1

(a) Given that

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{W}^T + \boldsymbol{B},$$

we deduce that

$$Y_{mn} = \sum_p X_{mp} W_{np} + B_{mn}.$$

We will now find the required derivatives.

(i)

$$\frac{\partial L}{\partial W_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \frac{\partial Y_{mn}}{\partial W_{ij}}.$$

But

$$\frac{\partial Y_{mn}}{\partial W_{ij}} = \sum_p X_{mp} \frac{\partial W_{np}}{\partial W_{ij}} = \sum_p X_{mp} \delta_{ni} \delta_{pj} = X_{mj} \delta_{ni},$$

and hence

$$\frac{\partial L}{\partial W_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} X_{mj} \delta_{ni} = \sum_m \frac{\partial L}{\partial Y_{mi}} X_{mj}.$$

In matrix-vector notation, this is equivalent to

$$\frac{\partial L}{\partial \boldsymbol{W}} = \left(\frac{\partial L}{\partial \boldsymbol{Y}}\right)^T \boldsymbol{X}$$

(ii)

$$\frac{\partial L}{\partial b_i} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \frac{\partial Y_{mn}}{\partial b_i}$$

But

$$\frac{\partial Y_{mn}}{\partial b_i} = \frac{\partial B_{mn}}{\partial b_i} = \delta_{ni},$$

Since $B_{mn} = b_n$ for all $m = 1, \dots, S$. Therefore,

$$\frac{\partial L}{\partial b_i} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \delta_{ni} = \sum_m \frac{\partial L}{\partial Y_{mi}}.$$

In matrix-vector notation, this is equivalent to

$$\frac{\partial L}{\partial \boldsymbol{b}} = \mathbf{1} \frac{\partial L}{\partial \boldsymbol{Y}}$$

where $\mathbf{1}$ is the $1 \times S$ ones-vector.

(iii)

$$\frac{\partial L}{\partial X_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \frac{\partial Y_{mn}}{\partial X_{ij}}.$$

But

$$\frac{\partial Y_{mn}}{\partial X_{ij}} = \sum_p \frac{\partial X_{mp}}{\partial X_{ij}} W_{np} = \sum_p \delta_{mi}\delta_{pj} W_{np} = \delta_{mi} W_{nj},$$

so

$$\frac{\partial L}{\partial X_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \delta_{mi} W_{nj} = \sum_n \frac{\partial L}{\partial Y_{in}} W_{nj}.$$

In matrix-vector notation, this is equivalent to

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

(b) Like before, we have

$$\frac{\partial L}{\partial X_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \frac{\partial Y_{mn}}{\partial X_{ij}}$$

But

$$\frac{\partial Y_{mn}}{\partial X_{ij}} = \frac{\partial h(X_{mn})}{\partial X_{ij}} = h'(X_{mn})\delta_{mi}\delta_{nj},$$

and therefore,

$$\frac{\partial L}{\partial X_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} h'(X_{mn})\delta_{mi}\delta_{nj} = \frac{\partial L}{\partial Y_{ij}} h'(X_{ij}).$$

Or, in matrix-vector notation:

$$\frac{\partial L}{\partial \boldsymbol{X}} = \frac{\partial L}{\partial \boldsymbol{Y}} \circ h'(\boldsymbol{X})$$

(c) (i)

$$\frac{\partial L}{\partial X_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \frac{\partial Y_{mn}}{\partial X_{ij}}.$$

The derivative of the softmax is

$$\begin{aligned}
\frac{\partial Y_{mn}}{\partial X_{ij}} &= \frac{\partial}{\partial X_{ij}} \frac{e^{X_{mn}}}{\sum_k e^{X_{mk}}} \\
&= \frac{\sum_k e^{X_{mk}} \cdot e^{X_{mn}} \delta_{mi}\delta_{nj} - e^{X_{mn}} \cdot \sum_k e^{X_{mk}} \delta_{mi}\delta_{kj}}{\left(\sum_k e^{X_{mk}}\right)^2} \\
&= \frac{\sum_k e^{X_{mk}} \cdot e^{X_{mn}} \delta_{mi}\delta_{nj} - e^{X_{mn}} \cdot e^{X_{mj}} \delta_{mi}}{\left(\sum_k e^{X_{mk}}\right)^2} \\
&= Y_{mn}\delta_{mi}\delta_{nj} - Y_{mn}Y_{mj}\delta_{mi} \\
&= Y_{mn}\delta_{mi}\left(\delta_{nj} - Y_{mj}\right)
\end{aligned}$$

Notice that when $m = i$ (i.e. when the same data point is considered), the derivative reduces to the usual softmax derivative for rank-1 tensors.
We get

$$\begin{aligned}
\frac{\partial L}{\partial X_{ij}} &= \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} Y_{mn}\delta_{mi}\left(\delta_{nj} - Y_{mj}\right) \\
&= \sum_n \frac{\partial L}{\partial Y_{in}} Y_{in}\left(\delta_{nj} - Y_{ij}\right)
\end{aligned}$$

(ii)

$$\begin{aligned}\frac{\partial L}{\partial X_{ij}} &= -\frac{1}{S}\sum_{m,n} T_{mn}\frac{\partial \log X_{mn}}{\partial X_{ij}} \\ &= -\frac{1}{S}\sum_{m,n}\frac{T_{mn}}{X_{mn}}\frac{\partial X_{mn}}{\partial X_{ij}} \\ &= -\frac{1}{S}\sum_{m,n}\frac{T_{mn}}{X_{mn}}\delta_{mi}\delta_{nj} \\ &= -\frac{1}{S}\frac{T_{ij}}{X_{ij}}\end{aligned}$$

The equation can be vectorized by using the elementwise division operator, known as the Hadamard division.

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

1.2 NumPy Implementation

Question 1.2

Training and testing the network for the default parameters, the following loss and accuracy curves as shown in Figure 1 were obtained. Note that the losses are with respect to the training set, whereas the accuracy is obtained from the test set. The classifier reaches an asymptotic accuracy of approximately 0.48.

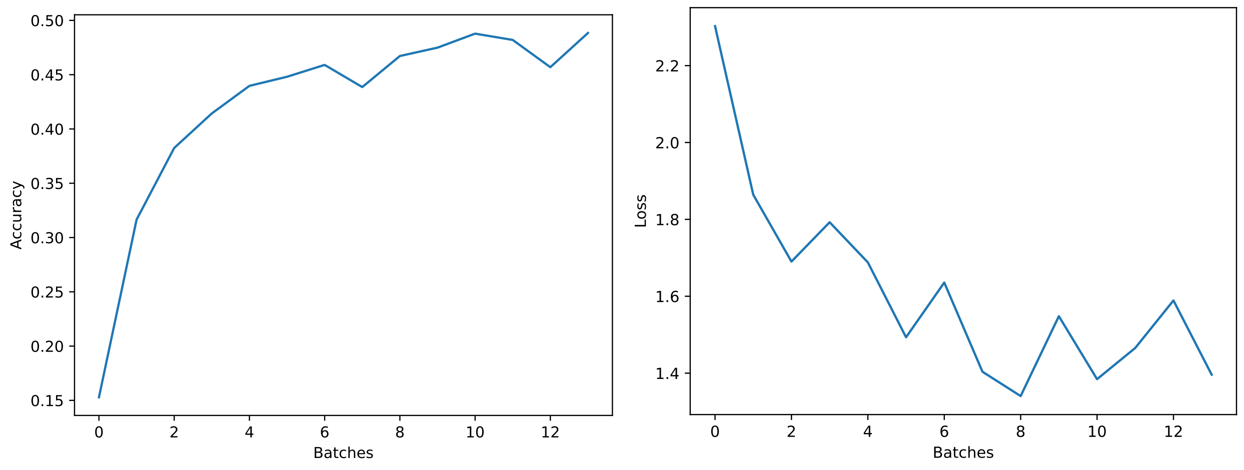

Figure 1: Accuracy and loss for the NumPy implementation of the MLP

2 PyTorch MLP

Question 2.1

Next, the same MLP was implemented using PyTorch. With the default parameters, an accuracy of 0.48 was reached. The following changes were made to the default network to increase the performance:

- A second layer, also consisting of 100 units was added. This modification in itself did not increase the accuracy. However, together with the next modification in this list, it increased accuracy with 0.02.
- The larger network needed more iterations for the accuracy to converge, so the maximum number of steps was increased to 3000. It led to another accuracy gain of 0.02.

- The optimizer was changed to Adam. Without this modification, the others seem to have much less effect. Changing from SGD to Adam with the other modifications in place, the accuracy increase is 0.07. Adam improves on regular SGD by including an adaptive learning rate, momentum and bias-corrected parameters [3].
- A batch normalization module was added after each ELU unit. It increased the accuracy with 0.06 with respect to the situation were the other modifications are applied, but no batch normalization is used. Batch normalization

Additionally, different learning rates and batch sizes were tried, but no significant increase in the accuracy was observed. In Figure 2, the accuracy and loss curves for the optimal settings as described above are plotted. An accuracy of 0.54 was attained. The shape of the curves are no surprise, looking similar to the ones for the NumPy implementation. An interesting (and expected) feature is that the accuracy curve converges around 15 batches, whereas the loss keeps decreasing. This can be regarded as a first sign of overfitting, since the generalization power of the model does not increase, but performance on the training set does.

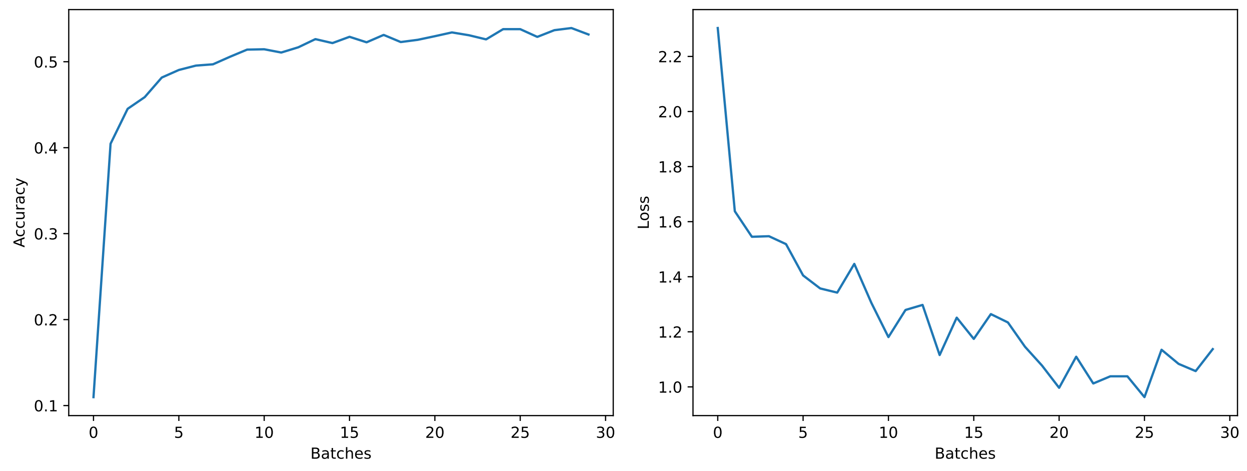

Figure 2: Accuracy and loss for the PyTorch implementation of the MLP

Question 2.2

Using Tanh instead of ELU mainly has drawbacks. The most important one is the problem of vanishing gradients. Since the derivative of Tanh always lies between 0 and 1 (except at identically 0), the gradients attenuate during backpropagation. As a consequence, the shallow layers learn much slower compared to the deeper layers. A possible advantage of Tanh over ELU is that it is zero-centered, meaning that its average output is zero. Zero-centered inputs lead to better training, so a zero-centered activation function is desired. This problem can however be solved by adding batch normalization between layers.

3 Custom Module: Layer Normalization

3.1 Automatic differentiation

Question 3.1

See the accompanying code

3.2 Manual implementation of backward pass

Question 3.2

(a) Given that

$$Y_{sj} = \gamma_i \hat{X}_{sj} + \beta_j,$$

we will find the required derivatives.

(i)

$$\frac{\partial L}{\partial \gamma_i} = \sum_{s,j} \frac{\partial L}{\partial Y_{sj}} \frac{\partial Y_{sj}}{\partial \gamma_i}.$$

But

$$\frac{\partial Y_{sj}}{\partial \gamma_i} = \hat{X}_{sj}\delta_{ij},$$

and therefore

$$\frac{\partial L}{\partial \gamma_i} = \sum_{s,j} \frac{\partial L}{\partial Y_{sj}} \hat{X}_{sj}\delta_{ij} = \sum_{s} \frac{\partial L}{\partial Y_{si}} \hat{X}_{si}.$$

In vectorized form, this is equivalent to

$$\frac{\partial L}{\partial \boldsymbol{\gamma}} = \left(\frac{\partial L}{\partial \boldsymbol{Y}}\right)^T \hat{\boldsymbol{X}}$$

(ii)

$$\frac{\partial L}{\partial \beta_i} = \sum_{s,j} \frac{\partial L}{\partial Y_{sj}} \frac{\partial Y_{sj}}{\partial \beta_i}.$$

But

$$\frac{\partial Y_{sj}}{\partial \beta_i} = \delta_{ij},$$

and therefore

$$\frac{\partial L}{\partial \beta_i} = \sum_{s,j} \frac{\partial L}{\partial Y_{sj}} \delta_{ij} = \sum_{s} \frac{\partial L}{\partial Y_{si}}.$$

In vectorized form, this is equivalent to

$$\frac{\partial L}{\partial \boldsymbol{\gamma}} = \mathbf{1}\frac{\partial L}{\partial \boldsymbol{Y}}$$

where $\mathbf{1}$ is the $1 \times S$ ones-vector.

(iii)

$$\frac{\partial L}{\partial X_{ri}} = \sum_{s,j} \frac{\partial L}{\partial Y_{sj}} \frac{\partial Y_{sj}}{\partial X_{ri}} = \sum_{s,j} \frac{\partial L}{\partial Y_{sj}} \frac{\partial Y_{sj}}{\partial \hat{X}_{sj}} \frac{\partial \hat{X}_{sj}}{\partial X_{ri}},$$

where we used the fact that Y_{sj} depends only explicitly on $\hat{\boldsymbol{X}}$ via $\hat{X}_{sj}$, such that we do not need to write another sum over the elements of $\hat{\boldsymbol{X}}$. Notice that $\hat{X}_{sj}$ depends both implicitly and explicitly on X_{ri}. Therefore we calculate the total derivative of $\hat{X}_{sj} = f(X_{sj}, \mu_s, \sigma_s^2)$ with respect to X_{ri}.

$$\begin{aligned}
\frac{\partial f}{\partial X_{ri}} &= \frac{\partial \hat{X}_{sj}}{\partial X_{ri}} + \frac{\partial \hat{X}_{sj}}{\partial \mu_s}\frac{\partial \mu_s}{\partial X_{ri}} + \frac{\partial \hat{X}_{sj}}{\partial \sigma_s^2}\frac{\partial \sigma_s^2}{\partial X_{ri}} \\
&= \delta_{rs}\delta_{ij} - \frac{1}{\sqrt{\sigma_s^2 + \epsilon}} \cdot \frac{1}{M}\sum_l \delta_{rs}\delta_{li} - \frac{1}{2}\left(X_{sj} - \mu_s\right)\left(\sigma_s^2 + \epsilon\right)^{-3/2} \cdot \frac{2}{M}\sum_l \left(X_{sl} - \mu_s\right)\delta_{rs}\delta_{li} \\
&= \delta_{rs}\delta_{ij} - \frac{\delta_{rs}}{M\sqrt{\sigma_s^2 + \epsilon}}\sum_l \delta_{li} - \frac{\left(\sigma_s^2 + \epsilon\right)^{-3/2}}{M}\left(X_{sj} - \mu_s\right)\left(X_{si} - \mu_s\right)\delta_{rs}
\end{aligned}$$

The remaining expression to find is $\frac{\partial Y_{sj}}{\partial \hat{X}_{sj}}$:

$$\frac{\partial Y_{sj}}{\partial \hat{X}_{sj}} = \gamma_j$$

We can now combine the found expressions to calculate the required derivative.

$$\begin{aligned}
\frac{\partial L}{\partial X_{ri}} &= \sum_{s,j} \frac{\partial L}{\partial Y_{sj}} \gamma_j \left[\delta_{rs}\delta_{ij} - \frac{\delta_{rs}}{M\sqrt{\sigma_s^2+\epsilon}} \sum_l \delta_{li} - \frac{\left(\sigma_s^2+\epsilon\right)^{-3/2}}{M} \left(X_{sj}-\mu_s\right)\left(X_{si}-\mu_s\right)\delta_{rs} \right] \\
&= \frac{\partial L}{\partial Y_{ri}} \gamma_i - \sum_{l,j} \frac{\partial L}{\partial Y_{rj}} \frac{\gamma_j \delta_{li}}{M\sqrt{\sigma_r^2+\epsilon}} - \sum_j \frac{\partial L}{\partial Y_{rj}} \frac{\gamma_j \left(\sigma_r^2+\epsilon\right)^{-3/2}}{M} \left(X_{rj}-\mu_r\right)\left(X_{ri}-\mu_r\right) \\
&= \frac{\partial L}{\partial Y_{ri}} \gamma_i - \frac{1}{M\sqrt{\sigma_r^2+\epsilon}} \sum_{l,j} \frac{\partial L}{\partial Y_{rj}} \gamma_j \delta_{li} - \frac{\left(\sigma_r^2+\epsilon\right)^{-3/2}}{M} \hat{X}_{ri} \sum_j \frac{\partial L}{\partial Y_{rj}} \gamma_j \hat{X}_{rj}
\end{aligned}$$

(b) See the accompanying code.

(c) Likewise, the implementation can be found in the attached python files.

(d) Batch normalization speeds up training by normalizing neuron inputs with the batch statistics [2]. This is useful when the batch size is significant, because the accuracy of the mean and variance estimates increases with the batch size. This approach can not be applied when training must be done with single-sample batches, or when training an RNN. For that reason, layer normalization was introduced [1]. It has the same effect of speeding up training, and has proved very effective for RNNs. However, it does not performs as well on image classification tasks, as discussed in the lecture.

Now we will discuss the performance of the two methods for different batch sizes. For small sizes, batch normalization will result in less accurate statistics, and therefore will not be as effective. For large batch sizes, batch normalization will yield very accurate statistics and therefore the method will be very effective in generating normally distributed activations. Like discussed before, in layer normalization statistics are computed over the feature dimension. This gives a set of statistics for each sample in the batch, and the performance of layer normalization is therefore independent of batch size.

4 PyTorch CNN

Question 4

(a) The CNN was implemented according to the instructions. The corresponding loss and accuracy curves are found in 3. The best accuracy was found to be 0.80.

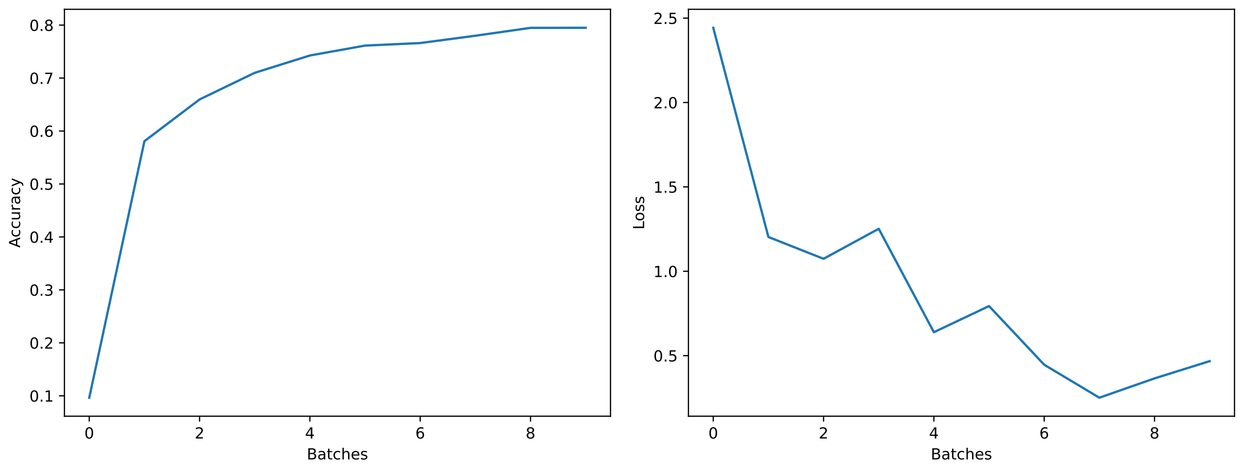

Figure 3: Accuracy and loss for the PyTorch implementation of the CNN

References

[1] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[2] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[3] D. P. Kingma and J. A. Ba. A method for stochastic optimization. arxiv 2014. *arXiv preprint arXiv:1412.6980*, 434, 2019.