Practical 1: MLPs, CNNs and Backpropagation

Erik Jenner
ID 13237896

1 MLP backprop and NumPy Implementation

1.1 Evaluating the Gradients

1.1.1 Linear Module

Given a linear module $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{W}^T + \boldsymbol{B}$, we want to find the gradients of the loss with respect to $\boldsymbol{X}$, $\boldsymbol{W}$ and $\boldsymbol{b}$.

First, we rewrite the module in index notation, using the Einstein sum convention:

$$Y_{ij} = X_{ik}W_{jk} + b_j$$

This gives us the following partial derivatives:

$$\begin{aligned}\frac{\partial Y_{ij}}{\partial X_{mn}} &= \delta_{im}\delta_{kn}W_{jk} = \delta_{im}W_{jn}\\ \frac{\partial Y_{ij}}{\partial W_{mn}} &= X_{ik}\delta_{jm}\delta_{kn} = \delta_{jm}X_{in}\\ \frac{\partial Y_{ij}}{\partial b_n} &= \delta_{jn}\end{aligned}$$

Using the chain rule, we get

$$\begin{aligned}\left(\frac{\partial L}{\partial \boldsymbol{X}}\right)_{mn} &= \frac{\partial L}{\partial Y_{ij}}\frac{\partial Y_{ij}}{\partial X_{mn}} = \frac{\partial L}{\partial Y_{mj}}W_{jn}\\ \left(\frac{\partial L}{\partial \boldsymbol{W}}\right)_{mn} &= \frac{\partial L}{\partial Y_{ij}}\frac{\partial Y_{ij}}{\partial W_{mn}} = \frac{\partial L}{\partial Y_{im}}X_{in}\\ \left(\frac{\partial L}{\partial \boldsymbol{b}}\right)_{n} &= \frac{\partial L}{\partial Y_{ij}}\frac{\partial Y_{ij}}{\partial b_n} = \sum_i \frac{\partial L}{\partial Y_{in}}\end{aligned}$$

We can rewrite this in matrix form as

$$\begin{aligned}\frac{\partial L}{\partial \boldsymbol{X}} &= \frac{\partial L}{\partial \boldsymbol{Y}}\boldsymbol{W}\\ \frac{\partial L}{\partial \boldsymbol{W}} &= \left(\frac{\partial L}{\partial \boldsymbol{Y}}\right)^T \boldsymbol{X}\\ \frac{\partial L}{\partial \boldsymbol{b}} &= \boldsymbol{1}^T\frac{\partial L}{\partial \boldsymbol{Y}}\end{aligned}$$

where $\boldsymbol{1}$ is the column vector of ones.

Preprint. Work in progress.

1.1.2 Activation Module

An activation module is given by $Y_{ij} = h(X_{ij})$. The derivative of an activation module is given by

$$\begin{aligned}\left(\frac{\partial L}{\partial X}\right)_{mn} &= \frac{\partial L}{\partial Y_{ij}}\frac{\partial Y_{ij}}{\partial X_{mn}} \\ &= \sum_{i,j}\frac{\partial L}{\partial Y_{ij}}h'(X_{ij})\delta_{im}\delta_{jn} \\ &= \frac{\partial L}{\partial Y_{mn}}h'(X_{mn}) \quad \text{(no sum)}\end{aligned}$$

In matrix notation, this is

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

where $\circ$ denotes the Hadamard product and the derivative h' is applied point-wise.

For the ELU activation function

$$h(x) := \begin{cases} x, & x \geq 0 \\ e^x - 1, & x < 0\end{cases}$$

this derivative is

$$h'(x) = \begin{cases} 1, & x \geq 0 \\ e^x, & x < 0\end{cases}$$

1.1.3 Softmax and Loss Modules

The softmax module is given by

$$Y_{ij} = \frac{\exp(X_{ij})}{\sum_k \exp(X_{ik})}$$

Using the quotient and chain rule, its derivative is therefore

$$\begin{aligned}\frac{\partial L}{\partial X_{mn}} &= \sum_{i,j}\frac{\partial L}{\partial Y_{ij}}\frac{\partial Y_{ij}}{\partial X_{mn}} \\ &= \sum_{i,j}\frac{\partial L}{\partial Y_{ij}}\frac{\exp(X_{ij})\delta_{im}\delta_{jn}\sum_k \exp(X_{ik}) - \exp(X_{ij})\sum_k \exp(X_{ik})\delta_{im}\delta_{kn}}{\left(\sum_k \exp(X_{ik})\right)^2} \\ &= \sum_{j}\frac{\partial L}{\partial Y_{mj}}\frac{\exp(X_{mn})\delta_{jn}\sum_k \exp(X_{mk}) - \exp(X_{mj})\exp(X_{mn})}{\left(\sum_k \exp(X_{mk})\right)^2} \\ &= \frac{\partial L}{\partial Y_{mn}}\frac{\exp(X_{mn})}{\sum_k \exp(X_{mk})} - \sum_j \frac{\partial L}{\partial Y_{mj}}\frac{\exp(X_{mj})\exp(X_{mn})}{\left(\sum_k \exp(X_{mk})\right)^2} \\ &= \frac{\partial L}{\partial Y_{mn}}Y_{mn} - \sum_j \frac{\partial L}{\partial Y_{mj}}Y_{mj}Y_{mn}\end{aligned}$$

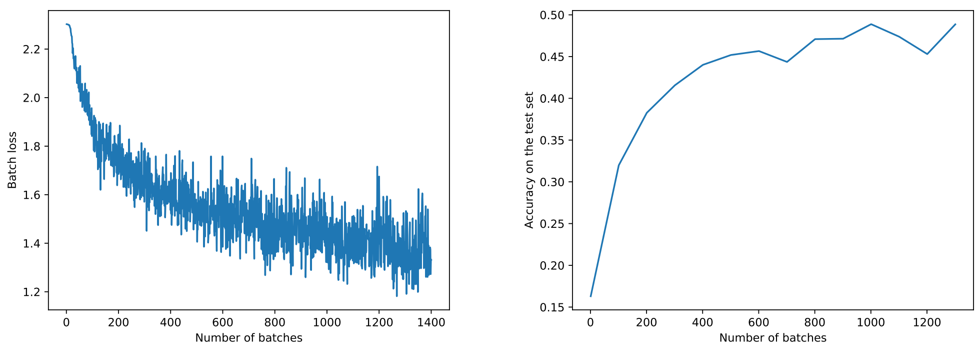
To avoid confusion, we've written out the sums just in this case. We never sum over m and n even where the Einstein convention tells us to.

The categorical cross entropy loss module is given by

$$L = -\frac{1}{S}\sum_{i,k} T_{ik}\log(X_{ik})$$

where $\boldsymbol{T}$ is the matrix of one-hot labels and S is the size of the batch dimension. Since the true labels $\boldsymbol{T}$ are fixed, we only need the derivative with respect to $\boldsymbol{X}$:

$$\begin{aligned}\frac{\partial L}{\partial \boldsymbol{X}_{mn}} &= -\frac{1}{S}\sum_{i,k}T_{ik}\frac{\delta_{im}\delta_{kn}}{X_{ik}} \\ &= -\frac{1}{S}\frac{T_{mn}}{X_{mn}}\end{aligned}$$



(a) Loss (b) Accuracy

Figure 1: Train loss and test accuracy curve with default settings

We could write this as

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

where $\frac{1}{\boldsymbol{X}}$ is the pointwise inverse of $\boldsymbol{X}$ (not $\boldsymbol{X}^{-1}$!).

1.2 NumPy implementation

Figure 1 shows the loss and accuracy curves of the NumPy implementation with default settings. The final test set accuracy after 1400 batches is 48.38%.

2 PyTorch MLP

2.1 Experiments

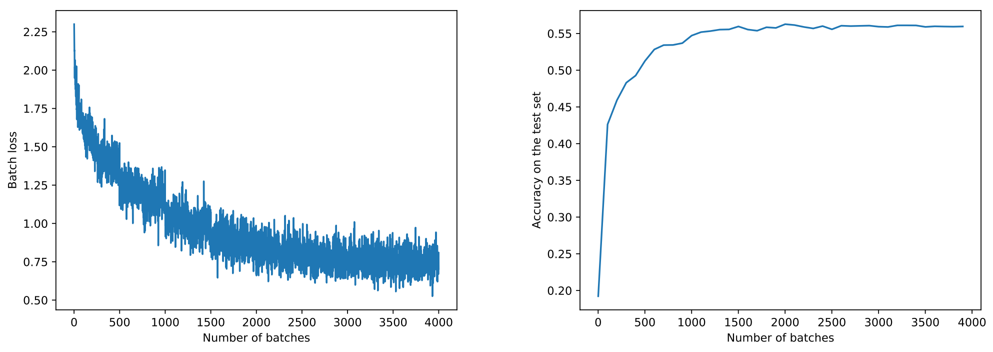
With the default settings from the NumPy implementation, including the same initialization, I get an accuracy of 49.12%. The small difference to the NumPy version presumably comes from different random numbers because the seeding procedures of NumPy and PyTorch are not the same. Just increasing the number of iterations does not help significantly (the accuracy peaks at slightly more than 50% after 2300 steps).

To get significantly better results, I wanted to increase the number of layers. But with the default initialization scheme (constant variance of 0.0001), two hidden layers lead to no learning at all, so I instead normalized the input to unit variance per channel and used PyTorch's default initialization for the linear layers.

Even so, training didn't work well and my network with two hidden layers underperformed the baseline with default parameters. Switching the optimizer from SGD to Adam (with its default PyTorch settings) lead to very large improvements.

With hidden layers of sizes [200, 200, 150, 100, 100] and longer training, I then reached the 52% accuracy. However, performance plateaued relatively early. To profit from long training times, I introduced a learning rate scheduler that decreased the learning rate by a factor of 0.5 every 500 iterations. With these settings, I reached an accuracy of about 56% after 2000 iterations, after which the performance plateaus. The loss and accuracy curves for these settings are shown in fig. 2. The noise in the loss curve comes from the fact that each loss is based on only one batch, it does not reflect actual noise in the performance on the entire train set (though that also exists with SGD). To reproduce these results, run

```
$ python train_mlp_pytorch --dnn_hidden_units \
        "200,200,150,100,100" --better-init --max_steps 4000
```



(a) Loss (b) Accuracy

Figure 2: Train loss and test accuracy curve with the best found settings

I also tried different learning rates than the default of 0.001, networks with more than 5 layers, SGD with momentum, batchnorm, and weight decay, all without significant improvements of even with slightly worse performance. It seems likely that some larger networks could achieve better performance if all the other parameters were adjusted in the right way but they did not with the settings I tried.

I also tried using ReLU and tanh activation functions instead of ELU but both were worse, ReLU slightly and tanh by a lot.

To summarize, the key changes for better performance were:

- More than one hidden layer (five)
- Adam instead of SGD
- Learning rate scheduler

These changes did not help or were detrimental:

- Even more than five layers
- Batchnorm
- Weight decay
- Different initial learning rates
- Other activation functions (ReLU, tanh)

2.2 Tanh vs ELU

The main advantage of the ELU activation function over Tanh is that it doesn't saturate for large activations. This means that it has large gradients for all positive activations, whereas the Tanh gradients approaches zero quite quickly for large inputs.

A potential advantage of Tanh is that it is symmetric. If the distribution over input activations is symmetric around 0 (e.g. Gaussian), then so is the output distribution. In particular, the expected value of 0 is preserved. In contrast, the ELU activation distorts symmetric distributions and it's output may have non-zero expected value even if the input is centered around zero. To avoid a shift in the mean activations, the bias may need to be adjusted correctly – or in other words: if the bias is initialized to zero, the network may initially exhibit a shift in the mean activations over different layers.

3 Layer normalization

In this entire section, we do **not** use the Einstein sum convention!

The layer normalization module is given by

$$Y_{si} = \gamma_i \hat{X}_{si} + \beta_i$$

where

$$\hat{X}_{si} = \frac{X_{si} - \mu_s}{\sqrt{\sigma_s^2 + \epsilon}}$$

with μ_s the mean and σ_s^2 the (biased) variance of $X_{s\cdot}$.

3.1 Implementation using autograd

See code

3.2 Manual implementation of backward pass

We first calculate the derivatives of Y_{si}:

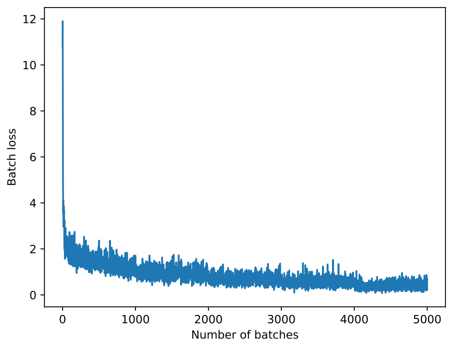
$$\begin{aligned}
\frac{\partial Y_{si}}{\partial \gamma_j} &= \hat{X}_{si}\delta_{ij} \\
\frac{\partial Y_{si}}{\partial \beta_j} &= \delta_{ij} \\
\frac{\partial Y_{si}}{\partial \hat{X}_{lj}} &= \gamma_i \delta_{ij}\delta_{sl}
\end{aligned}$$

To find $\frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{X}}$, we first need

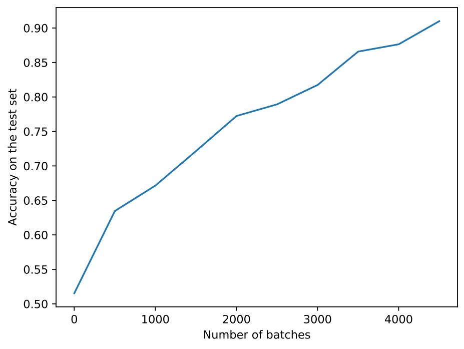
$$\begin{aligned}
\frac{\partial \mu_s}{\partial X_{lj}} &= \frac{1}{M}\delta_{sl} \\
\frac{\partial \sigma_s^2}{\partial X_{lj}} &= \frac{1}{M}\sum_i 2(X_{si} - \mu_s)(\delta_{sl}\delta_{ij} - \frac{\partial \mu_s}{\partial X_{lj}}) \\
&= \frac{2}{M}\delta_{sl}(X_{sj} - \mu_s) \\
\frac{\partial \hat{X}_{si}}{\partial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

Now we get

$$\begin{aligned}
\frac{\partial Y_{si}}{\partial X_{lj}} &= \sum_{m,n}\frac{\partial Y_{si}}{\partial \hat{X}_{mn}}\frac{\partial \hat{X}_{mn}}{\partial X_{lj}} \\
&= \sum_{m,n}\gamma_i \delta_{in}\delta_{sm}\frac{\partial \hat{X}_{mn}}{\partial X_{lj}} \\
&= \gamma_i \frac{\partial \hat{X}_{si}}{\partial X_{lj}} \\
&= \frac{\gamma_i \delta_{sl}}{M\sqrt{\sigma_s^2 + \epsilon}}\left(M\delta_{ij} - 1 - \hat{X}_{si}\hat{X}_{sj}\right)
\end{aligned}$$



(a) Loss



(b) Accuracy

Figure 3: Train loss and test accuracy curve with the default settings

The derivatives of the loss are then given as follows:

$$\begin{aligned}
\frac{\partial L}{\partial \gamma_j} &= \sum_{s,i} \frac{\partial L}{\partial Y_{si}} \frac{\partial Y_{si}}{\partial \gamma_j} \\
&= \sum_{s} \frac{\partial L}{\partial Y_{sj}} \hat{X}_{sj} \\
\frac{\partial L}{\partial \beta_j} &= \sum_{s,i} \frac{\partial L}{\partial Y_{si}} \frac{\partial Y_{si}}{\partial \beta_j} \\
&= \sum_{s} \frac{\partial L}{\partial Y_{sj}} \\
\frac{\partial L}{\partial X_{lj}} &= \sum_{s,i} \frac{\partial L}{\partial Y_{si}} \frac{\partial Y_{si}}{\partial X_{lj}} \\
&= \sum_{i} \frac{\partial L}{\partial Y_{li}} \frac{\gamma_i}{M\sqrt{\sigma_l^2 + \epsilon}} \left(M\delta_{ij} - 1 - \hat{X}_{li}\hat{X}_{lj} \right)
\end{aligned}$$

3.3 Layer normalization vs Batch normalization

Batch normalization normalizes each feature separately over an entire batch, with the desired mean and variance as learnable parameters. This means that changing the weights in linear layers does not change the mean/variance of activations, which is instead governed independently by different parameters. This allows the network to keep the variances in a desirable range even in very deep networks, and it allows us to use a higher learning rate because weight changes in linear layers don't disturb the distribution of activations as much.

However, if the batch size is very small, computing the mean and variance based on mini-batches becomes very noisy. Layer normalization works perfectly fine for small batch sizes because it normalizes across the different features (of which there are usually enough for statistics that are not too noisy). In contrast to batch normalization, this does change the expressiveness of the network (the activations for all features are forced to have the same mean/variance).

4 Pytorch CNN

4.1 CNN implementation

Figure 3 shows loss and accuracy curves for the CNN with default settings. The test set accuracy after 5000 batches is 91%, better than expected from the task statement. To reproduce, simply run `$ python train_convnet_pytorch.py`.