Deep Learning Report

Dorin Doncenco Zhe Huang Benjamin Maudet *

April 2023

1. Introduction

We detail in this report the details behind the implementation of a pytorch-like, albeit way simpler, deep neural network library, with an implementation of autograd, as a practical exercise for the Deep Learning course. This library is easily extendable by the creation of modules, which can then be encapsulated in one another, and chained, to make custom networks.

2. Building Blocks and Network Architecture

2.1. Tensors and Parameters

In neural networks, computations are usually done with tensors. Mathematically, tensors can be viewed as matrices generalized to dimensions > 2. Programmatically, they can be represented with numpy arrays. In our implementation, they also carry information useful to back-propagation, i.e their gradient, the derivative of the function they were computed with, and the arguments for this derivative.

A special subclass of Tensor, named Parameter, is used to store the weights and biases of the network. A Parameter behaves the same as a Tensor, except that they always specify that their gradient needs to be computed (which is not the case for all tensors, e.g an input to the network), and that you can't back-propagate from them as they are not computed during the forward pass.

In the case of batch learning (which is not implemented), one would have to make sure that they use a gradient accumulation function instead of simply saving the newly computed gradient. For good coding practices, we do the same in our notebook.

*Authors in alphabetical order of last name

2.2. Linear classifier

If d the dimension of the input, $x \in \mathbb{R}^d$ an input vector, k the number of classes, $W \in \mathbb{R}^{(k \times d)}$ a matrix of parameters, and $b \in \mathbb{R}^k$ a vector of biases, then a linear classifier, or a multiclass perceptron, can be defined as

$$\begin{aligned} z &= xW^\top + b \\ \hat{y} &= softmax(z) \end{aligned} \tag{1}$$

Where $z \in \mathbb{R}^k$ is the output from the logit space, and $\hat{y} \in [0,1]^k$ is a probability vector representing the predicted probabilities that x should be classified as each class.

To compute the error, we can use the Negative Log Likelihood loss

$$\begin{aligned} NLL(z,y) &= -\log \frac{e^{z[y]}}{\sum_i^k e^{z[i]}} \\ &= -z[y] + \log \sum_i^k e^{z[i]} \end{aligned} \tag{2}$$

where y is the true class of the sample x.

We are looking to change the parameters W and b in order minimize the error. To do this, we can use gradient descent with learning rate η:

$$\begin{aligned} W^{(t+1)} &\leftarrow W^{(t)} - \eta * \nabla_W NLL \\ b^{(t+1)} &\leftarrow b^{(t)} - \eta * \nabla_b NLL \end{aligned} \tag{3}$$

However, calculating these gradients manually is possible, but annoying. And it will get even more annoying in deep neural networks, where there are multiple layers to go through. This is where the chain rule comes to play. The chain rule states that

$$\frac{\partial NLL}{\partial W} = \frac{\partial NLL}{\partial z} \frac{\partial z}{\partial W} \tag{4}$$

So we can find the gradients of NLL with respect to z and of the linear computation with respect to W and b separately, and multiply them together to get the final gradient. This is the principle behind autograd : we keep for each computation during the forward pass the value of the computation and the formula for its derivative. Then, we can just go backwards in the computation and multiply the gradients together. Let's compute each gradient. For the NLL:

$$\begin{aligned} \nabla_z NLL(z,y) &= \nabla_z \left[-z[y] + \log \sum_i^k e^{z[i]} \right] \\ &= -\mathbf{e}_y + \frac{\nabla_z \sum_i^k e^{z[i]}}{\sum_i^k e^{z[i]}} \\ &= -\mathbf{e}_y + softmax(z) \end{aligned} \tag{5}$$

where $\mathbf{e}_k$ is a vector of all zeros, except for element k which is 1. Then for the linear transformation:

$$\begin{aligned} \nabla_W \left[xW^\top + b\right] &= x \\ \nabla_b \left[xW^\top + b\right] &= 1 \end{aligned} \tag{6}$$

So during training, we do a forward pass:

$$x \xrightarrow{xW^\top + b} z \xrightarrow{NLL} loss \tag{7}$$

Then for backpropagation, we use the following gradients for the update step.

$$\begin{aligned} \nabla_W(NLL) &= \nabla_z(NLL) \times \nabla_W(z) \\ \nabla_b(NLL) &= \nabla_z(NLL) \times \nabla_b(z) \end{aligned} \tag{8}$$

2.3. Deep Neural Networks

A deep neural network, also called Multi Layer Perceptron in its simplest form, is basically multiple linear layers, that we saw in last section, chained together. They are separated by an activation function σ, used to break the linearity of the network. Some popular choices of activation functions are the Rectified Linear Unit $ReLU$ or the hyperbolic tangent $tanh$. We can then represent the deep neural network in the following way:

$$\begin{aligned} z_1 &= xW_1^\top + b_1 \\ z_1^{'} &= \sigma(z_1) \\ z_2 &= z_1^{'}W_2^\top + b_2 \\ z_2^{'} &= \sigma(z_2) \\ &[...] \\ z_n &= z_{n-1}^{'}W_n^\top + b_n \end{aligned} \tag{9}$$

Where n is the number of layers. As with before, the NLL is computed with

$$NLL(z_n, y) = -z_n[y] + \log \sum_j \exp(z_n[j]) \tag{10}$$

Our goal is then to update $W_1 \dots W_n$ and $b_1 \dots b_n$ to minimize the NLL. This is where we're glad to have implemented autograd before. Computing the gradient for each set of parameter manually would be a nightmare. With autograd, everything is done for us:

1. Gradient of NLL with respect to z_n
2. Gradient of z_n with respect to W_n and b_n
3. Multiply 1. and 2. : We can update W_n and b_n !
4. Gradient of $\sigma(z_{n-1})$ with respect to z_{n-1}

5. Gradient of z_{n-1} with respect to W_{n-1} and b_{n-1}
6. Multiply 1., 2., 4. and 5. : We can update W_{n-1} and b_{n-1}!
7. and so on and so on...

The only changes we need to do here compared to the linear part is to compute the derivative of our activation function for use in the chain rule, and compute the derivative of the linear transformation $Wx + b$ with respect to x. We didn't have to do that before because the only input for the linear transformation was the data itself, for which we don't need the gradient. But in this deep network, the input of a linear transformation can be the output of an activation function, so we might need it to continue the backpropagation!

Let's do that:

$$\begin{aligned} ReLU(x) &= max(0, x) \\ \nabla_x ReLU(x) &= \begin{cases} 0, & x < 0 \\ 1, & x > 0 \end{cases} \end{aligned} \tag{11}$$

$$\begin{aligned} tanh(x) &= \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \\ \nabla_x tanh(x) &= \frac{\nabla_x (e^x - e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})\nabla_x (e^x + e^{-x})}{(e^x + e^{-x})^2} \\ &= \frac{(e^x + e^{-x})^2}{(e^x + e^{-x})^2} - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2} \\ &= 1 - tanh^2(x) \end{aligned} \tag{12}$$

$$\nabla_x (Wx + b) = W \tag{13}$$

Now that we have all the gradient formulas that we need, as before, we can go backwards in the computation graph to find all the gradient values we want using the chain rule. For example, if we had the following 2 layer network:

$$\begin{aligned} z_1 &= x W_1^\top + b_1 \\ z_1' &= \sigma(z_1) \\ z_2 &= z_1' W_2^\top + b_2 \end{aligned} \tag{14}$$

The update steps would be:

$$\begin{aligned} W_2 &\leftarrow W_2 - \eta \left[\nabla_{z_2} NLL \times \nabla_{W_2} z_2\right] \\ b_2 &\leftarrow b_2 - \eta \left[\nabla_{z_2} NLL \times \nabla_{b_2} z_2\right] \\ W_1 &\leftarrow W_1 - \eta \left[\nabla_{z_2} NLL \times \nabla_{z_1'} z_2 \times \nabla_{z_1} z_1' \times \nabla_{W_1} z_1\right] \\ b_1 &\leftarrow b_1 - \eta \left[\nabla_{z_2} NLL \times \nabla_{z_1'} z_2 \times \nabla_{z_1} z_1' \times \nabla_{b_1} z_1\right] \end{aligned} \tag{15}$$

All these gradients are easy to compute if we keep track of the values at each step during the forward pass!

3. Implementation

Tensors are objects that contain the data, and if gradient is required, the gradient value (to be computed), along with the backward function to compute the gradient and the backpointers. The backpointers are Tensor values which will be passed to the backward function.

Each layer in the neural network implements a forward and backward pass. Forward passes take as an input Tensors (one or [optionally more]) and return the function output of the layer as a Tensor, along with the back pointers and its paired backward function. For example, for a function f(x):

```
def f_forward_pass(x, [W,b]):
    v = f(x.data, [W.data, b.data])
    output = Tensor(v, require_grad=x.require_grad)
    output.d = backward_pass
    output.backptr = [x, [W,b]]
    return output
```

The backward pass associated with it computes the gradient for the current layer via the derivative function and multiplies it with the gradient g from the next layer (or $g = 1$ if this is the loss (last) layer), and adds it to the current existing gradient for this layer.

```
def f_backward_pass(backptr, g):
    x, [W,b] = backptr
    for ptr in backptr:
        if ptr.require_grad:
            #d_f is different based on the parameter
            ptr.accumulate_gradient(g * d_f(ptr.data))
```

where d_f() is the derivative of f() with respect to ptr.

When backpropagation is initiated from a tensor (e.g the NLL), its backwards pass function is called to compute its gradient with respect to its parameters, then the back-propagation is continued on each parameter if needed:

```
# simplified code for Tensor.backward() (no error handling)
def backward(self, g=None):
    if g is not None:
        g=self.gradient
    self.d(self.backptr, g) # e.g f_backward_pass
    for ptr in self.backptr:
        ptr.backward()
```

For the optimizer, we have implemented the momentum feature in addition to the typical step update. The momentum allows to maintain the gradient direction from previous steps.

The momentum is updated after every step, and is initialized as $m_{W_t} = 0$ at the first time step $t = 0$:

$$m_{W_t} \leftarrow \beta \times m_{W_{t-1}} + (1 - \beta) \times \nabla_{W_{t-1}} \tag{16}$$

$$W_t \leftarrow W_{t-1} - \eta \times m_{W_t} \tag{17}$$

where β is the decay rate of the momentum, W is a model parameter being optimized (the optimizer keeps track of a separate momentum for every parameter), and t is the time step.

We have implemented the linear layer and activation functions as individual classes extending Module, which enables us to build sequential deep network instead of storing a list of weights for every layer:

```
# layer initialization for DeepNetwork
self.layers = ModuleList()
indim, outdim = dim_input, hidden_dim
for i in range(n_layers):
    if i == n_layers-1:
        outdim = dim_output
    if i > 0:
        indim = hidden_dim
    self.layers.extend([
        LinearNetwork(indim, outdim),
        ReluLayer() if not tanh else TanhLayer()
    ])
# for last layer, we don't want an activation layer
self.layers.pop()
```

We initialize the weights using the Glorot initialization formula, which is known to work well. This keeps the weights close to zero the bigger a layer size is, preventing bigger layers from taking control over the architecture in the context of an architecture of differing sizes. This prevents gradient explosions in context of ReLU, and gradient vanishing for the tanh (which flattens outside of the middle values). This leads to quicker convergence during training.

$$W_{k,d,} = U[-\frac{\sqrt{6}}{\sqrt{k+d}}, \frac{\sqrt{6}}{\sqrt{k+d}}] \tag{18}$$

After training the network for a few epochs, it tends to reach a point where it seemnily stops getting better. This could be caused by a step size that is too high, leading to the gradient jumping around a local minimum without improving. When we notice this happening (by noting that the training loss increases), we decrease the step size i.e. learning rate decay.

```
#i : current epoch, decay_rate : a scalar between ]0,1[
if loss[i] > loss[i-1]:
    learning_rate = learning_rate * decay_rate
```

4. Experiments and Results

To study the interplay between the hyper-parameters and find the best ones for the task, we did a grid search on possible values. Due to time constraints and computation cost, we only tested a small subset of parameters and possible values :

$$\begin{aligned} \text{number of layers} &\in \{2, 3, 5\} \\ \text{momentum} &\in \{Enabled, Disabled\} \\ \text{learning rate decay} &\in \{Enabled, Disabled\} \\ \text{activation} &\in \{ReLU, tanh\} \end{aligned} \tag{19}$$

unfortunately parameters like initial learning rate, momentum rate β, learning rate decay factor, number of hidden neurons... were not investigated. All runs were ran for 10 epochs. The entire search took 7 hours and 5 minutes to complete on Kaggle. The results are detailed on figure 2, in the appendix.

We can see that all configurations yield almost the same highest development accuracy. They are all included in the range $[0.9725, 0.979]$, so not a significant difference between the lowest and highest performing one. What we can see though is that some achieve their highest accuracy faster than others.

The average epoch to achieve higher accuracy is 7.16 with momentum, and 8.16 without, which shows that on average, momentum allows the model to converge a bit faster. However, for the accuracy, the results are so close that we can't really make claims about the impact of each parameter. Nevertheless, we can try picking the parameters that yielded the highest accuracy (3 layers, weight decay, no momentum, ReLU), and train it for longer than 10 epochs to see how good it can get.

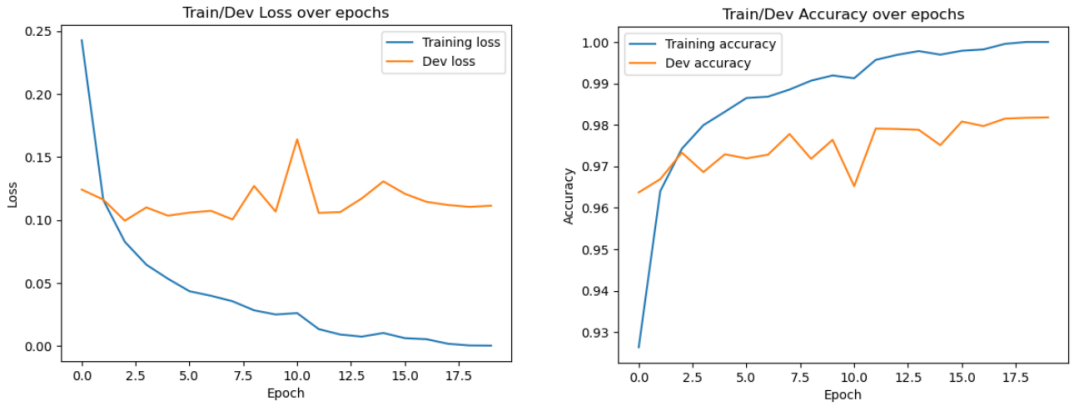

Figure 1: Plots of evolution of train/dev loss and accuracy for the re-training of the best model

After training for 20 epochs, the model got a development accuracy of 0.9818. This result is okay, but we still need one final test to verify that we did "fit" our model to the development set, through over-adapting the hyper-parameters to achieve higher results

on the development set. We can make this final test on the test set, which hasn't been touched at any point until now.

After running the test, we get an accuracy of 0.9816 on the test set. This is very close to our accuracy on the development set, which means our model is able to generalize well enough to unseen data (or at least, unseen data from the same data-set).

5. Conclusion

In this report we have described our implementation of a deep neural network. The implementation mimics the functioning of the PyTorch library, enabling the backpropagation algorithm to train neural networks, along features such as learning decay rate, optimizer momentum, weight initialization and a PyTorchian network builder. We have ran a grid-search on various hyper-parameters and the dev set, and finally achieve an accuracy of 0.9818 on the test set (used only once).

6. Future work

Next steps to probably achieve higher performances would be to implement batch learning, and ultimately implement convolution layers, which are the gold standard for image recognition tasks (the best accuracy achieved on MNIST is 99.91% with a simple CNN). Convolution layers would probably not be hard to implement in our current system thanks to the modules and the autograd algorithm. It would also be beneficial to speed up the training by running matrix multiplication operations on GPU, process multiple data points with a single matrix multiplication by making x a $(batchsize, nfeatures)$ matrix instead of a vector, and implement the linear transformation layer with a single matrix multiplication Wx instead of $Wx + b$ by including the biases in the W matrix and augmenting the data matrix with ones.

Appendix

A. Grid search results

	Layers	LR decay	Momentum	Tanh	Max dev accuracy	At epoch
0	2	True	True	True	0.9771	7
1	2	True	True	False	0.9782	8
2	2	True	False	True	0.9759	9
3	2	True	False	False	0.9762	9
4	2	False	True	True	0.9759	7
5	2	False	True	False	0.9762	8
6	2	False	False	True	0.9761	7
7	2	False	False	False	0.9781	9
8	3	True	True	True	0.9778	7
9	3	True	True	False	0.9738	5
10	3	True	False	True	0.9761	9
11	3	True	False	False	0.9790	9
12	3	False	True	True	0.9767	8
13	3	False	True	False	0.9771	7
14	3	False	False	True	0.9770	9
15	3	False	False	False	0.9789	8
16	5	True	True	True	0.9752	8
17	5	True	True	False	0.9743	5
18	5	True	False	True	0.9725	7
19	5	True	False	False	0.9770	9
20	5	False	True	True	0.9732	9
21	5	False	True	False	0.9746	7
22	5	False	False	True	0.9725	7
23	5	False	False	False	0.9760	6

Figure 2: Results of the grid search, with the value of each investigated hyper-parameter, the maximum development accuracy reached, and the epoch (out of 10) where the maximum development accuracy was reached. Highlighted is the test with the highest development accuracy.