Deep Learning Lab Exercise Report

Julien Rolland
Alexandre Combeau
Gaëtan Serré
`{first name.surname}@universite-paris-saclay.fr`

Contents

1 Introduction

This tutorial aims at implementing a set of Python classes to create, train and infer a neural network. We will use Numpy[4] to facilitate and accelerate the matrix calculations. In order train and test our model, we will use the MNIST[2] dataset, which consists in recognizing handwritten numbers.
A neural network can be seen as a sequence of n functions that our data will traverse: $y = f_n(f_{n-1}(...f_1(x)...))$ with x a data and y the predictions of the network. Certain of these functions use some parameters P that will have to be modified during the training phase so that the network predictions are as close as possible to our gold labels. To quantify this distance, we will use a cost function $\mathcal{L}$ which grows accordingly to the amount of errors in our network.
The goal is to modify the parameters P in order to minimize $\mathcal{L}$. We will use the gradient descent algorithm (and one of its variants):

$$P \leftarrow P - \epsilon \nabla_P \mathcal{L},\ \epsilon \in \mathbb{R} \tag{1}$$

To do this, we need the gradient of $\mathcal{L}$ with respect to each parameter P - i.e. we want to know how an infinitesimal change of P impacts $\mathcal{L}$.
The difficulty of this exercise is to implement a data structure allowing to, for each parameter P, calculate $\nabla_P \mathcal{L}$ and update it automatically.

The overwhelming majority of the calculations and ideas come from the Deep Learning course taught by Caio Filippo Corro[1].

2 Chain rule

As said above, we need to compute $\nabla_P \mathcal{L}$ to update P. To compute this gradient, we will use the chain rule:

Let $f:\ \mathbb{R}^m \times \mathbb{R}^{k \times d} \rightarrow \mathbb{R}^n$, $x \in \mathbb{R}^m$ and $P \in \mathbb{R}^{k \times d}$.

$$\begin{gathered} y = f(x, P) \\ \frac{\partial \mathcal{L}}{\partial P_{i,l}} = \sum_{j=1}^{n} \frac{\partial \mathcal{L}}{\partial y_j} \frac{\partial y_j}{\partial P_{i,l}} \iff \nabla_P \mathcal{L} = \mathbf{J}_p y^\top \,.\, \nabla_y \mathcal{L} \end{gathered} \tag{2}$$

This equation shows that, in order to calculate $\nabla_P \mathcal{L}$, we need to know $\nabla_y \mathcal{L}$. We therefore need to implement a data structure in which we can transmit the gradients where they are required. We will organize our operations in the form of a graph which we will call a computational graph.

3 Computational graph

A computational graph is a graph where each node represents a parameter or the result of a function (which we will call tensor). The edges represent the links between the tensors and the variable used to calculate its value. We can therefore use the edges to transfer the gradients from the child nodes to the parent nodes as illustrated in Figure 1.

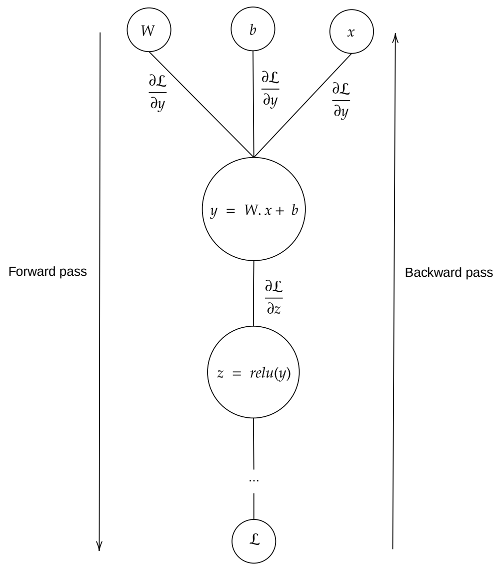

W
b
x
$\frac{\partial \mathcal{L}}{\partial y}$
$\frac{\partial \mathcal{L}}{\partial y}$
$\frac{\partial \mathcal{L}}{\partial y}$
$y = W.x + b$
Forward pass
Backward pass
$\frac{\partial \mathcal{L}}{\partial z}$
$z = relu(y)$
...
$\mathcal{L}$

Figure 1: Example of a computational graph

The computational graph will be built when computing the prediction for a data x. We will call this phase the forward pass. Then, we have to start from from the last node (i.e. the computation of the cost function), calculate its gradient transmits it to the parent nodes and repeat these steps until there are no more no gradient to compute. We will call this phase the backward pass.

Now that we have our data structure, all we have to do is list the mathematical functions that we will use in our neural network and compute their gradient expression using the chain rule before moving on to the implementation in Python.

4 Useful Maths

Our neural network will be relatively simple and use only layers that apply an affine transformation on the incoming data. We will also be able to use two activation functions: $relu$ and $tanh$. Finally, we need to define a cost function $\mathcal{L}$ and as our problem is a multi-class classification problem we will use the negative log-likelihood. We will now detail the expression of these different functions as well as the gradient of their variable(s) with respect to $\mathcal{L}$.

4.1 Affine transform

This layer represents the core of our neural network. It will transform the input data $x \in \mathbb{R}^n$ to a vector of size m using two parameters: a matrix $W \in \mathbb{R}^{m \times n}$ and a vector $b \in \mathbb{R}^m$ (often called bias). The list of W and b (one pair (W, b) per layer) will be the only trainable parameters of our network.

Expression:

$$\begin{aligned} f : \mathbb{R}^{m\times n} \times \mathbb{R}^n \times R^m &\to \mathbb{R}^m \\ (W, x, b) &\mapsto W.x + b \end{aligned} \tag{3}$$

Gradients:

Let $y = f(W, x, b)$

- $\nabla_W \mathcal{L}$:

$$\begin{aligned} &\frac{\partial \mathcal{L}}{\partial W_{i,l}} = \sum_{j=1}^{m} \frac{\partial \mathcal{L}}{\partial y_j} \frac{\partial y_j}{\partial W_{i,l}} \\ &\frac{\partial y_j}{\partial W_{i,l}} = x_l \\ &\nabla_W \mathcal{L} = (\nabla_y \mathcal{L}) \ . \ (x^\top) \end{aligned} \tag{4}$$

- $\nabla_b \mathcal{L}$:

$$\begin{aligned} &\frac{\partial \mathcal{L}}{\partial b_i} = \sum_{j=1}^{m} \frac{\partial \mathcal{L}}{\partial y_j} \frac{\partial y_j}{\partial b_i} \\ &\frac{\partial y_j}{\partial b_i} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases} \\ &\nabla_b \mathcal{L} = \nabla_y \mathcal{L} \end{aligned} \tag{5}$$

- $\nabla_x \mathcal{L}$:

$$\begin{aligned} &\frac{\partial \mathcal{L}}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial \mathcal{L}}{\partial y_j} \frac{\partial y_j}{\partial x_i} \\ &\frac{\partial y_j}{\partial x_i} = W_{j,i} \\ &\nabla_x \mathcal{L} = (W^\top) \ . \ (\nabla_y \mathcal{L}) \end{aligned} \tag{6}$$

4.2 Activation functions

4.2.1 Hyperbolic tangent

This function can be used to project the result of affine layers into the interval $]-1, 1[$.
The advantage of *tanh* is that it returns small numbers centered on 0. However, it is subject to the gradient vanishing problem that we will detail later.

Expression:

$$\begin{aligned} tanh : \mathbb{R}^m &\to \mathbb{R}^m \\ x &\mapsto \frac{1 - e^{-2x}}{1 + e^{-2x}} \end{aligned} \tag{7}$$

Gradient:

Let $y = tanh(x)$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial x_i} &= \sum_{j=1}^{m} \frac{\partial \mathcal{L}}{\partial y_j}\frac{\partial y_j}{\partial x_i} \\ \frac{\partial y_j}{\partial x_i} &= \begin{cases} 1 - tanh(x_i)^2, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases} \\ \nabla_x \mathcal{L} &= (\nabla_y \mathcal{L}) \left[1 - tanh(x)^2\right]\end{aligned} \tag{8}$$

4.2.2 Rectified Linear Unit

This function is very popular in deep learning s very simple to compute and avoids the gradient vanishing problem. However, ReLU is subject to the gradient exploding problem and the gradient of a negative input is zero which will stop its learning.

Expression:

$$\begin{aligned} relu : \mathbb{R}^m &\to \mathbb{R}^m \\ x &\mapsto max(0, x)\end{aligned} \tag{9}$$

Gradient:
We use the subgradient of $relu$.
Let $y = relu(x)$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial x_i} &= \sum_{j=1}^{m} \frac{\partial \mathcal{L}}{\partial y_j}\frac{\partial y_j}{\partial x_i} \\ \frac{\partial y_j}{\partial x_i} &= \begin{cases} 1, & \text{if } i = j \text{ and } 0 \leq x_i \\ 0, & \text{if } i \neq j \text{ or } 0 > x_i \end{cases} \\ \nabla_x \mathcal{L} &= (\nabla_y \mathcal{L})\ v\end{aligned} \tag{10}$$

Where $v \in \mathbb{R}^n$ such that, $\forall\ 1 < i < n \in \mathbb{N}$, if $x_i > 0$ then $v_i = 1$ else $v_i = 0$.

4.3 Negative log-likelihood

This function, also called categorical cross-entropy, is a cost function adapted for classification problems. classification problems. The more a data is misclassified (i.e. the probability given by the network that the data belong to another class), the higher the result of the negative log-likelihood will be higher.

Expression:

$$\begin{aligned} nll : \mathbb{R}^m \times \mathbb{N} &\to \mathbb{R} \\ (x, gold) &\mapsto ln\ \frac{\exp(x_{gold})}{\sum_j \exp(x_j)}\end{aligned} \tag{11}$$

Gradient:

Let $y = nll(x)$

In our case, nll is our cost function, it is the entry point of the computational graph and so we do not need the chain rule anymore. However, for the sake of generalization, we will express the gradient of an arbitrary cost function $\mathcal{L}$ with respect to x. In our implementation, the incoming gradient $(\nabla_y \mathcal{L})$ will be equal to 1.

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial x_i} &= \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial x_i} \\ \frac{\partial y}{\partial x_i} &= \frac{e^{x_i}}{\sum_j e^{x_j}} - \mathbb{1}_{[i=gold]} \\ \nabla_x \mathcal{L} &= (\frac{\partial \mathcal{L}}{\partial y})\,(-v + softmax(x)) \end{aligned} \tag{12}$$

Where $v \in \mathbb{R}^n$ such that, $\forall\ 1 < i < n \in \mathbb{N}$, if $i = gold$ then $v_i = 1$ else $v_i = 0$.

We now have all the mathematical formulas we need. Before moving on to the implementation, we will quickly talk about the initialization of the parameters of a neural network.

5 Parameter initialization

Training a neural network is just an optimization problem, and like any optimization problem a good initialization of the parameters allows to converge faster to the optimal solution as illustrated in the Figure 2.

In deep learning, we have two constraints during initialization: the parameters must have the same magnitude so that there is not a subset of layers doing all the work and they must all be close to 0. The latter avoids two problems: gradient exploding and gradient vanishing.

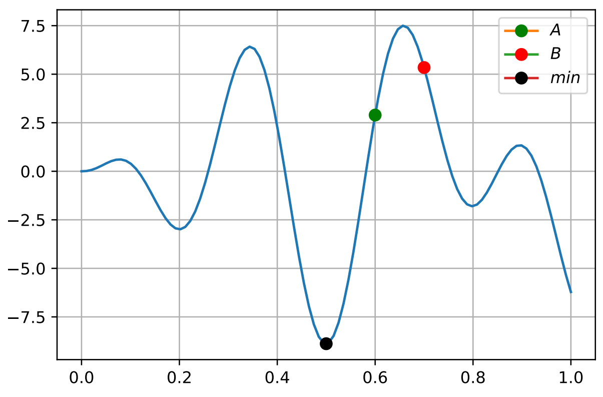


Figure 2: min is much more easy to reach if we start from A than from B

5.1 Gradient exploding

During the backward pass, we multiply gradients between them. If the parameters have been initialized with numbers that are too large, we risk having gradients that explode and after a while, our computer will no longer be able to handle such large numbers. It is to avoid this problem that we use functions such as *tanh* that project the output of the layers to small numbers.

5.2 Gradient vanishing

Some activation functions such as *tanh* or *sigmoid* have a particular derivative: the larger the input, the closer the derivative is to 0 (Figure 3).
Consequently, if we have very large parameters, during the backward pass some 0 will appear in the calculation of the gradients and therefore some will be zero and will vanish.

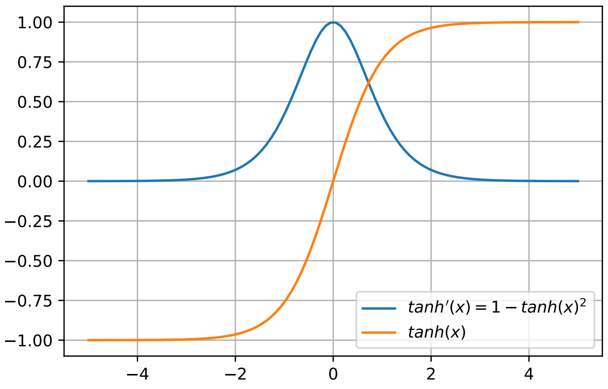


Figure 3: $\lim_{x\to\infty} tanh'(x) = 0$

Thus, to meet these constraints, in our implementation we will use the Glorot[3] : $W \sim \mathcal{U}[\pm\frac{\sqrt{6}}{\sqrt{m+n}}]$, initialization when the activation function is *tanh* and the Kaiming[5] : $W \sim \mathcal{U}[\pm\frac{\sqrt{6}}{\sqrt{n}}]$, initialization when the activation function is *ReLU*.

6 Optimizer

The training of a neural network consists in finding the parameters P that minimize the loss $\mathcal{L}$. It is thus a problem of optimization that we will try to solve with the gradient descent.

6.1 Stochastic gradient descent

The simplest variant. Compute the gradient $\nabla_P\mathcal{L}$ and modify the parameters in the opposite direction to this one. With $\epsilon \in \mathbb{R}$ to modulate the step size.

$$P^{(t+1)} = P^{(t)} - \epsilon\,\nabla_P\mathcal{L},\ \epsilon \in \mathbb{R} \tag{13}$$

6.2 Momentum stochastic gradient descent

A more advanced variant. We introduce a new value γ symbolizing the "velocity" or "momentum", which is more or less modified by the gradient $\nabla_P\mathcal{L}$ at each step, according to a new hyper parameter $\mu \in \mathbb{R}$. The P parameters are then modified as before according to the value of γ.

Intuitively, the idea is to "keep in memory" the previous gradients, which allows to always advance in the global direction of the descent, taking less account of the last calculated gradient. This is similar to the Newtonian mechanics of a ball subjected to the force of gravity which would roll down the slope of the loss. Gravity (here $\nabla_P \mathcal{L}$) affects the momentum (here γ) which then affects the position (here P).

$$\begin{aligned} \gamma^{(t+1)} &= \mu\gamma^{(t)} + \nabla_P \mathcal{L}, \quad \mu \in \mathbb{R} \\ P^{(t+1)} &= P^{(t)} - \epsilon\gamma^{(t+1)}, \quad \epsilon \in \mathbb{R} \end{aligned} \tag{14}$$

7 Implementation details

We must now implement the computational graph described in 3 in Python. For that we will define two classes: *Tensor* and *Parameter*.

7.1 *Tensor*

The class *Tensor* contains several attributes:

- *data*: A matrix that corresponds to the mathematical value of this *Tensor*;
- *require_grad*: A boolean indicating whether the gradient of this *Tensor* should be calculated;
- *gradient*: The value of the gradient of $\mathcal{L}$ with respect to this *Tensor*;
- *backptr*: The list of variables useful to calculate *gradient*;
- *d*: A function that will compute and accumulate the gradient of each variable v located in *backptr* (if v is a *Tensor* and its gradient must be computed). See Figure 4b.

It also contains a main method: *backward* which will call the function d and then call the *backward* methods of each of the *Tensor* of the list *backptr* (if their gradient is necessary).

7.2 *Parameter*

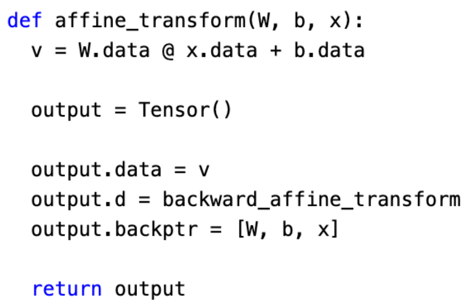
The *Parameter* class inherits from *Tensor*. The only differences are that its attribute *require_grad* is always true because we want to update the parameters during training, so we need their gradient and also, it has no *backward* method because the parameters are leaves of the computational graph, so they don't have any parents to transmit their gradient to.

The objects of type *Tensor* and *Parameter* represent the nodes of our computational graph and their attribute *backptr* represents the edges of this node.

7.3 Forward/Backward pass

We can now easily create and train a neural network.

1. We initialize the parameters of the affine layers according to their shape: we have created the leaves of the computational graph.
2. Then, we define a method *forward* that will pass our data in a set of functions of the form described in Figure 4a.
   Each of these functions creates a *Tensor* - i.e. a new node and its edges, in the computational graph.
3. Call the *forward* method using a training data and store its output - i.e. the predictions of the network in a variable $\widehat{y}$
4. Now, to do the backward pass, we just have to compute a tensor with the value of $\mathcal{L}$ using $\widehat{y}$ and call its method *backward*.
   When this method has finished its execution, the attribute *gradient* of each parameter p of the neural network will contain the value $\nabla_p\mathcal{L}$.
5. We can now update our parameters using our optimizer.
6. Redo steps 3, 4 and 5 as many times as we have epochs.



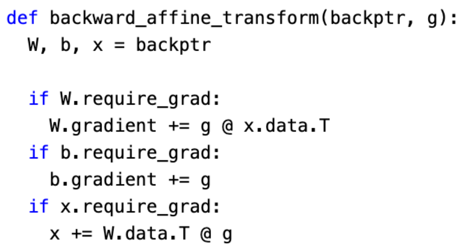
```
def affine_transform(W, b, x):
  v = W.data @ x.data + b.data

  output = Tensor()

  output.data = v
  output.d = backward_affine_transform
  output.backptr = [W, b, x]

  return output
```

(a) Simplified example of a function computing a *Tensor*



```
def backward_affine_transform(backptr, g):
  W, b, x = backptr

  if W.require_grad:
    W.gradient += g @ x.data.T
  if b.require_grad:
    b.gradient += g
  if x.require_grad:
    x += W.data.T @ g
```

(b) Simplified example of a *d* function

Figure 4: We used the mathematical expressions defined in 4

8 Results

For each of the neural network configurations tried, the score on the training, validation and test set is roughly the same. We also observe the expected curves: the cost function decreasing and the train accuracy increasing with respect to the number of epochs as illustrated in Figure 5. For a network without hidden layer, we obtain a score of about 0.92. With two hidden layers of 100 neurons, a score of 0.97 with *ReLU* or *tanh* as activation function. We also implemented an SGD momentum optimizer but the score remained the same. However, we did not try to optimize the hyperparameters.

9 Conclusion

The implementation of a deep learning library is a big challenge in both mathematics and in computer science. Indeed, it is first mandatory to define the necessary mathematical tools then find and implement a data structure that allows to train the parameters efficiently. We were able to put our knowledge in these two domains during this exercise and this allowed us to better understand how a neural network works both in theory and in practice. However, our implementation has many limitations, the most obvious being, in our opinion, the slowness of the calculations.

References

[1] Caio Filippo Corro. Deep learning course. 2022.

[2] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[3] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[4] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.

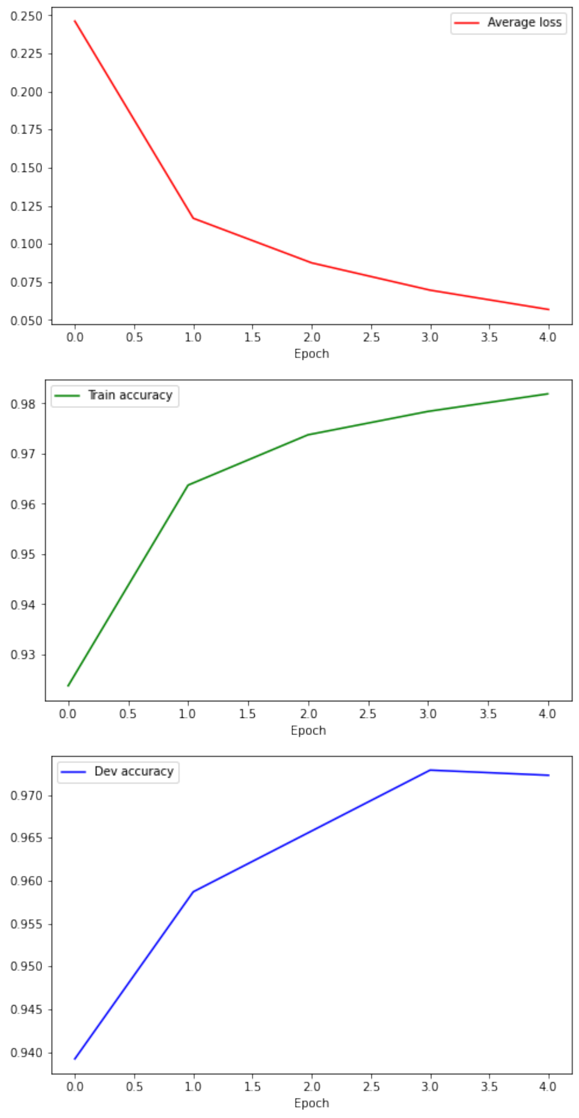


Figure 5: Results of a neural network with 2 hidden layers of 100 neurons using *ReLU*