Deep Learning EE-559

Project 2: Mini Deep Learning Framework

Clémentine Aguet, Yamin Sepehri, Mahdi Nobar

May 16, 2019

Abstract

Deep learning has recently become popular in various research domains or industrial applications, notably thanks to the improvement in frameworks like *PyTorch* or *Tensorflow*. Although these frameworks are easily used as a *blackbox*, being aware of the logic behind them helps to achieve good results and guides the optimization process. In this project, a mini framework is implemented without utilization of available neural network libraries to acquire the knowledge of the general process.

Keywords: Deep Learning, Framework, Neural Network, PyTorch

Introduction

With the rise of big data and computer power, the interest for deep learning keeps growing. This data-driven approach has recently achieved huge successes in various domains. In order to facilitate the implementation of deep learning models, multiple frameworks are at our disposal. Among others *Tensorflow*, *PyTorch*, *MXNet* or *Caffe* are widely used. Such toolkits offer a clear and rapid way to build, train and validate deep learning networks by simplifying some programming challenges for the developer. However writing the code from scratch can be of great help to understand more precisely the methodology behind. The purpose of this project is to understand the underlying concept of deep learning framework by designing one with no helps of *autograd* or existing neural network modules.

First of all, a rapid overview of the mathematical background behind the implementation of deep learning networks is presented. Starting from a simplified representation, neural networks can be seen as a black box including a learning process and a prediction process. During learning, the model takes inputs and expected outputs to appropriately update its internal state. The model then makes predictions based on given inputs and the learning internal states.

The architecture of a neural network is organized in layers. First, an input layer brings the initial data to the system. One or several hidden layers are following. They output weighted inputs of the previous layer with affine transformation. Finally, an output layer makes the prediction related to the task.

The first part of the learning process consists of a forward propagation. In this pass, the inputs x are linearly transformed by multiplying a weighted matrix w and adding bias b. The results are then passed through an activation function before being given as inputs to the next layer. This process can be written as follows, $\forall l = 1, 2, ..., L$:

$$x^{(0)} = x \tag{1}$$

$$s^{(l)} = w^{(l)}x^{(l-1)} + b^{(l)} \tag{2}$$

$$x^{(l)} = \sigma(s^{(l)}) \tag{3}$$

The output of the network is defined as:

$$x^{(L)} = f(x; w, b) \tag{4}$$

The learning process consists in finding a set of parameters (w and b) minimizing an empirical loss. A typical loss function is the *Mean-Squared Error* (MSE):

$$\begin{aligned}\mathcal{L}(w, b) &= \sum_n l(f(x_n; w, b), y_n) \\ &= \frac{1}{N}\sum_{n=1}^{N}(f(x_n; w, b) - y_n)^2\end{aligned}$$

The minimization algorithm generally used is the gradient descent. It exploits local linear information in order to iteratively progress and reach local minimum.

Backpropagation is a fast way of computing the gradient of the loss function. This method is based on the chain rule. It first computes the derivative from the definition of the loss: $\left[\frac{\partial \ell}{\partial x^{(L)}}\right]$. The next step is to recursively back-propagate the derivative of the loss with respect to the activations.

$$\left[\frac{\partial \ell}{\partial x^{(l)}}\right] = \left(w^{(l+1)}\right)^T \left[\frac{\partial \ell}{\partial s^{(l+1)}}\right] \tag{5}$$

$$\left[\frac{\partial \ell}{\partial s^{(l)}}\right] = \left[\frac{\partial \ell}{\partial x^{(l)}}\right] \odot \sigma'(s^{(l)}) \tag{6}$$

It finally computes the derivatives with respect to the parameters:

$$\left[\frac{\partial \ell}{\partial w^{(l)}}\right] = \left[\frac{\partial \ell}{\partial s^{(l)}}\right] \left(x^{(l-1)}\right)^T \tag{7}$$

$$\left[\frac{\partial \ell}{\partial b^{(l)}}\right] = \left[\frac{\partial \ell}{\partial s^{(l)}}\right] \tag{8}$$

The iterative rule (Equation 9 and 10) updates the parameters and operates following the steepest descent. It will eventually and hopefully end up in a local minimum. However the process is highly dependent on w and b initialization and the learning rate η. Some parameter tuning is thus required.

$$w^{(l)} \longleftarrow w^{(l)} - \eta \left[\frac{\partial \ell}{\partial w^{(l)}}\right] \tag{9}$$

$$b^{(l)} \longleftarrow b^{(l)} - \eta \left[\frac{\partial \ell}{\partial b^{(l)}}\right] \tag{10}$$

Framework implementation

The whole implementation of the framework is based on a class named Module. The different modules of this framework inherit from this class, in such manner that they have the same structure:

- forward method gets input, performs forward pass and returns a tensor as output.
- backward method gets as input a tensor containing the gradient of the loss with respect to the module's output. It accumulates the gradient with respect to the parameters, and returns a tensor containing the gradient of the loss with respect to the module's input.
- param method (optional) returns a lists of pairs, composed of parameter tensor and gradient tensor.

The framework is organized in seven main files. Each, in most cases, inherits from the class Module and contains other classes or methods described below:

1. activation_functions file contains two classes, each for a different activation function, which are described as follows:
   - Relu class calculates rectified linear unit activation function (ReLU) for forward and backward computations. This function is linear for all positive values, and zero for all negative values. It is especially used in *Convolutional Neural Networks* (CNN).

     $$f(x) = \begin{cases} 0 & for \quad x < 0 \\ x & for \quad x \geq 0 \end{cases} \tag{11}$$

   - Tanh class defines the hyperbolic tangent activation function and computes the forward and backward passes. It is a scaled *sigmoid* function with steeper derivatives and bounded in [-1, 1].

     $$f(x) = \frac{2}{1+e^{-2x}} - 1 \tag{12}$$

2. data file includes three methods employed in the data pre-processing to prepare the data for the model:
   - generate_data method synthesizes data points sampled uniformly in $[0,1]^2$, each with a label 0 if outside a disk of radius$1/\sqrt{2\pi}$ and 1 inside.
   - normalization method is used to normalize data by subtracting the mean and dividing the result by the standard deviation.
   - convert_to_one_hot_labels method generates a one-hot tensor corresponding to each target labels.
3. initialization file includes a method required to perform parameter initialization based on the *Xavier* method and calculates the gain with respect to activation function type provided by the user as input to this method. This aims to keep the values in a range through many layers.
4. losses file contains the definition of multiple loss functions:
   - MSELoss class provides forward and backward passes for *mean square error* loss function. It computes the average magnitude errors of predictions. MSE might be more appropriate for regression tasks.

     $$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{13}$$

   - CrossEntropyLoss class provides *cross entropy loss* function. It defines the performance of a task with probability outputs. In contrast to MSE, cross entropy might be more suitable for classification tasks.

     $$-\left[y_i \cdot log(\hat{y}_i) + (1-y_i) \cdot log(1-\hat{y}_i)\right] \tag{14}$$

5. neural_network file consists of two sub-classes:
   - Linear class describes the forward and backward propagation through a fully connected layer. In such layer, neurons of adjacent layers share all connections, but neurons within a single layer are not connected.
   - Sequential class combines layers used to build the model. The forward and the backward passes are applied through a for loop over all the stacked modules.
6. optimization file includes a class typically used to optimized the model parameters.
   - SGD class applies *Vanilla mini-batch stochastic gradient descent* to the weight calculation. It also benefits from *momentum* to add inertia in the step direction choice.
7. main_new_framework file is an executable that imports the implemented framework and verifies it using a synthesized dataset and a simple neural network. It contains three main methods:
   - training method aims to improve the model performance. This phase is done on batches of a training set containing inputs with their corresponding outputs. At each epoch, the outputs of the model are compared to the expected targets through the loss function. This criterion is then used for optimization and to adjust the model parameters. The process is described more in details in the introduction.
   - compute_error method purpose is to evaluate the model performance. It quantifies the difference between the model predictions and the target values of a given dataset.
   - main method merges all steps together. It first initializes the model and training parameters. A training and test datasets are generated and normalized. After building the model in a sequential

manner, it learns its parameters based on the training set. And its performance is evaluated through the error rate on the test set.

8. main_pytorch script is also provided. Structured in comparable manner as main_new_framework, it describes the implementation of a model similar to the one described above but built using *PyTorch* and trained with exactly the same parameters. It is used as a comparitve and validation approach to ensure the correct development of our framework.

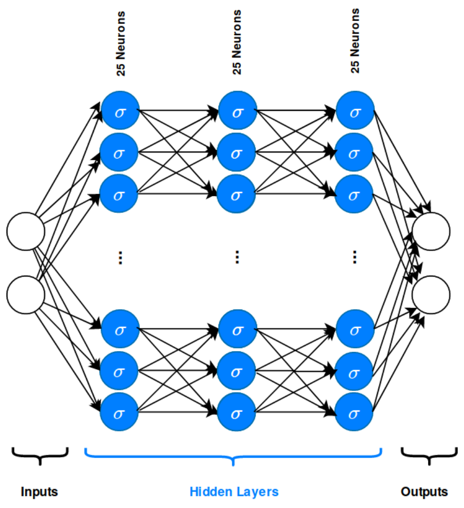

Figure 1: *NN with 3 hidden layers, 25 neurons each*

Verification & Comparison

In order to validate the executability of the mini deep learning framework implemented in this project, a simple model was trained on synthesized data. For the purpose of comparison and correct implementation confirmation of our approach, a similar model with same parameters was built using *PyTorch* framework.

The synthesized data consists of points uniformly sampled in $[0,1]^2$. They are marked such that points inside the disk of radius $1/\sqrt{2\pi}$ have a label of 1 and 0 otherwise. Two datasets are generated, a training set and a test set each containing 1000 samples.

Layer	Type	Input size	Output size
1	Linear	2	25
	Tanh		
2	Linear	25	25
	Tanh		
3	Linear	25	25
	Tanh		
4	Linear	25	2

Table 1: *Model architecture*

The model architecture itself has two inputs and two output units. It consists of three hidden layer of 25 units, (see table 1). The model is then trained on the training set with forward and backward passes using *MSE* loss function and *SGD* optimizer. The test set serves for the evaluation process. The performance are estimated through 100 iterations and reported as error rate. After some parameter tuning, a number of epochs of 200, a batch size of 100 and learning rate of 0.01 were selected as most appropriate.

The results obtained with those hyperparameters after running 100 iterations of the two implementations are displayed in table 2. It points out that the new framework obtains very similar outcomes than the classical *PyTorch* framework, whether for the training or the test error rates.

	Train		Test	
	Mean	Std	Mean	Std
New framework	0.0129	0.39	0.0333	1.31
PyTorch framework	0.0128	0.36	0.0344	1.17

Table 2: *Train and test error rates. Trained over 200 epochs with batch size of 100 and learning rate of 0.001*

Regarding the computational time, there is a slight non-significant improvement for our implementation (1.25s vs 1.4s for a training iteration). This difference probably comes from the fact that this framework is much less complex and contains very few functions compared to *PyTorch* framework.

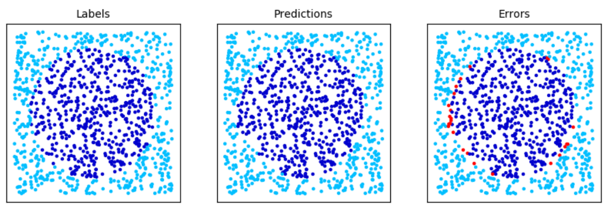

Figure 2: *Predictions and errors visualization*

Figure 2 shows a good visualization of this verification task. The first column represents labeled data, with in light blue points outside the disk and dark blue inside. The second column illustrates the model predictions. And the last one shows a really small error rate, with only few miss-classified red points. The implemented framework thus allowed to correctly train a model and converge to a good solution.

Conclusion

By developing a small deep learning framework from scratch one can gain a strong intuition on the general process of neural networks. Using already implemented framework hides a lot of complex computations. Going from theory to executable requires to achieve mathematical reformulations in order to be able to obtain great performance with powerful and fast computations.

References

[1] Aaron Courville Ian Goodfellow, Yoshua Bengio. *Deep Learning*. MIT Press, 2016.

[2] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer Science, 2006.