EE-559 Deep Learning: Project 2

Raphael Uebersax
School of Engineering (STI)
EPFL
Lausanne, Switzerland
raphael.uebersax@epfl.ch

Pedro Bedmar López
School of Computer Science (IC)
EPFL
Lausanne, Switzerland
pedro.bedmarlopez@epfl.ch

Jonas Perolini
School of Engineering (STI)
EPFL
Lausanne, Switzerland
jonas.perolini@epfl.ch

I. Introduction

The progress and growing interest in the field of deep artificial neural networks helped creating many different frameworks specifically conceived for deep learning such as TensorFlow or PyTorch to name a few. Those frameworks allow people to develop more efficient tools along with simplification of programming challenges. This project aims at developing a simple version of such a deep learning framework using python. More precisely, the challenges of implementing a Multi-Layer Perceptron (MLP) model including linear layers, Relu and Tanh activation functions as well as mean square error loss are tackled. The aim is to create an automatic mechanism that performs backpropagation as well a stochastic gradient descent.

The report is organized as follows. First, in section II, the overall structure of the framework as well as mathematical background of the forward and backward pass are described. Second, the usage of the framework is presented in section III. The general methodology for performance assessment is then detailed in section IV before presenting the results and briefly discussed them in section V. Finally, section VI concludes the report.

II. Framework structure

The general idea behind the implemented structure is to automatise the forward as well as the backward pass of a deep network architecture defined by the user. To do this, several modules are implemented and linked together in order to rightfully compute the update rule (gradient step) for each parameters such that the loss can be minimized. Figure 1 provides a useful scheme to recall the mathematical formulas used in training a multi-layer perceptron with non-linear activation functions. One key takeaway from this figure is that for each parameters involved in the forward pass, a corresponding gradient of the loss with respect to it need to be computed in order to train the network. Additionally, some intermediary results of the forward pass are also required in the backward pass which motivates storing them previously. It is also worth mentioning that the framework should be compliant with both training on individual samples as well as on minibatches.

The framework used in this project is structured using a parent class called "Module" from which all other modules inherit. It possesses three methods called "forward", "backward" and "param". The two first need to be implemented in every module. Additionally, a class called "Parameter" is also implemented for which any instance has two attributes: the parameter's value as well as the gradient of the loss with respect to the parameters. The idea is to link the gradient with the parameter to simplify the update step using stochastic gradient descent.

Next, the different implemented modules are presented including details about their mathematical implementation. The notations are aligned with the ones in figure 1

- **Linear:** The Linear module implements a fully connected linear layer. It takes as arguments the number of input units followed by the number of output units. It also proposes two possible weight and bias initialization which follow both a uniform distribution of type $\omega, b \sim U[\frac{-1}{\sqrt{k}}, \frac{1}{\sqrt{k}}]$. The default one is used to control the variance of the activations and has as parameter $k = N_{l-1}$ where N_{l-1} corresponds to the number of input units. The second is Xavier's initialization which additionally tries to control the variance of the gradient with respect to the activations. It has as parameter $k = \frac{N_{l-1}+N_l}{6a^2}$ where N_l corresponds to the number of output units and a to the gain of the nonlinear activation function. To use Xavier's initialization, a third input argument should be given with a boolean value of "True" and a forth to correspond to the gain of the following activation function. Note that by default, the gain is set to $\frac{5}{3}$ which correspond to the tanh activation function.
  - ***forward:*** The forward pass of a linear layer with input x_{l-1} can be expressed by:

$$s_l = w_l \cdot x_{l-1} + b_l \tag{1}$$

    where w_l and b_l correspond to the weight and bias of the layer. It is also expanded to do this operation on tensors to allow for training with minibatches. It is worth mentioning that the input x_{l-1} is stored as an attribute of the layer as this information is needed in the backward pass (see the connections between the forward and backward pass of the linear layers in figure 1).
  - ***backward:*** The backward pass of a linear layer with input $\frac{\partial L}{\partial s_l}$ can be expressed as:

$$\frac{\partial L}{\partial x_{l-1}} = w_l^T \frac{\partial L}{\partial s_l} \tag{2}$$

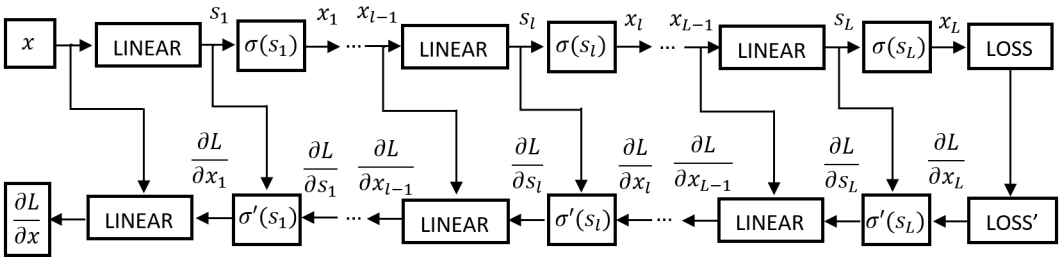

Fig. 1: Learning rate tuning for each model

Moreover, two additional terms need to be calculated which are the gradient of the loss with respect to the layer's parameters. Those will then be stored as an attribute of the parameter itself to be able to perform the gradient step later on. The gradients are given by:

$$\frac{\partial L}{\partial w_l} = \frac{\partial L}{\partial s_l} x_l^T \quad \text{(3a)} \qquad \frac{\partial L}{\partial b_l} = \frac{\partial L}{\partial s_l} \quad \text{(3b)}$$

It is worth noting that when the network is trained with minibatches, the mean gradient over the mini-batch rather than the sum of all gradients is stored. This is done in order to keep a similar order of magnitude independently of the mini-batch size which generally facilitates the learning rate tuning.

- ***param:*** The param method returns a list of all internal parameters of the linear module. The list has two elements $[\omega, b]$ where both have as attribute the parameter's value and the gradient of the loss with respect to the parameter.

- **Tanh:** The Tanh module implements the hyperbolic tangent activation function. It does not take any argument and does not need any parameter initialization.
  - ***forward:*** The forward method starts by storing the input tensor s_l as an attribute as it is needed for the backward pass (see the connections from the forward towards backward pass of the derivative of the activation function in figure 1). Then, it computes the element-wise hyperbolic tangent:

$$\sigma_T(s_l) = tanh(s_l) \tag{4}$$

  - ***backward:*** The backward pass takes as input $\frac{\partial L}{\partial x_l}$ and computes the derivative of the loss with respect to the activation function which can be expressed as:

$$\frac{\partial L}{\partial s_l} = \frac{\partial L}{\partial x_l} \odot \sigma_T'(s_l) \tag{5}$$

    where the $\odot$ is the Hadamard component-wise product.

- **ReLU:** The ReLU module also implements an activation function which is called Rectified Linear Unit. Similarly to the Tanh module, it does not take any argument and does not need to initialize any parameters.
  - ***forward:*** Similarly to the Tanh module, the forward method starts a by storing the input tensor s_l as an attribute and then computes the output of the ReLU activation which is given by:

$$\sigma_R(s_l) = \begin{cases} 0, & \text{if } s_l < 0 \\ s_l, & \text{otherwise} \end{cases} \tag{6}$$

  - ***backward:*** The backward pass again takes as input $\frac{\partial L}{\partial x_l}$ and computes the derivative of the loss with respect to the activation function which can be expressed as:

$$\frac{\partial L}{\partial s_l} = \frac{\partial L}{\partial x_l} \odot \sigma_R'(s_l) \tag{7}$$

- **MSELoss:** The MSELoss module implements the mean square error loss function. It does not require any input argument as initialization.
  - ***forward:*** The forward method takes as arguments the network's output x_L and the target output tensor t. The MSE loss is then computed as follows:

$$L = (x_L - t)^2 \tag{8}$$

  - ***backward:*** The backward method takes the same input as the forward and computes the derivative of the loss with respect to the output of the last layer.

$$L = 2 \cdot (x_L - t) \tag{9}$$

- **Sequential:** The sequential module links all other modules together. It takes as input an ordered list of all the layers and activations required to build a desired network as in figure 1.
  - ***forward:*** Its forward method accesses the forward methods of all the modules defined in the initialization in order to go through the whole network.
  - ***backward:*** Its backward method those the same but in the reversed order. It propagates the gradient through all layers.
  - ***param:*** Finally, its param method returns a list of all parameters of the network.

Overall, the Network is trained using gradient decent in order to minimize the MSE loss. The parameters returned by the sequential allow to update the weights and biases of each linear layers with the gradient step:

$$w_l = w_l - \eta \frac{\partial L}{\partial w_l} \quad \text{(10a)} \qquad b_l = b_l - \eta \frac{\partial L}{\partial b_l} \quad \text{(10b)}$$

where η is the learning rate.

III. Framework usage

This section presents how to use the framework. First, the network model is defined. To do so, the network's linear layers and activation functions are passed as an ordered list to the sequential module. As previously mentioned, the user can choose between the Tanh and the ReLU activation function. Additionally, the criterion must be defined as MSELoss. This can be done as follows:

$$model = Sequential([Linear(in, out), Tanh(), ...])$$
$$criterion = MSELoss()$$

Training can be performed by iterating through the training set over several epochs. First, the forward method of the model must be called with an input either of 1-dimensional (individual sample) or of 2-dimensional (mini-batch). Second, the gradient of the loss with respect to the network output can be computed by calling the backward method of the criterion. the backward method of the model then computes each gradient of the loss with respect to the parameters. Lastly, to update the network, the gradient step can be performed using the following code:

$$output = model.forward(train\_input)$$
$$grad\_loss = criterion.backward(output, train\_target)$$
$$model.backward(grad\_loss)$$
$$for\ p\ in\ model.param() : \quad p.p = p.p - \eta \cdot p.gradient$$

IV. Methodology

This section details the methodology implemented to test the framework on a simple binary classification task. The classification is performed on a training set and a testing set. Each set is composed of 1000 two dimensional points each sampled uniformly in $[0;\ 1]^2$, with label 0 or 1 if the point lays outside or inside the disk of radius $\frac{1}{\sqrt{2\pi}}$ centered at $(0.5;\ 0.5)$ respectively. In order to assess the performances of the framework, a simple network is implemented through the sequential module. The network's architecture uses two input units, two output units and three hidden layers of 25 units. It is trained with MSELoss over 250 epochs using a mini-batch size of 100 samples and a learning rate of 0.15.

The network is then tested on two experiments. For each experiment, statistics (i.e mean and standard deviation) on the training and testing classification errors over 10-iterations (each time shuffling the training set) are reported.

The first experiment is deigned to ensure that the framework works properly. It was thus decided to test all possible combinations. The network is trained and tested for both types of activation functions, namely hyperbolic tangent and ReLU as well as both types of weights and bias initialization, namely the default one and Xavier's. It's important to note that the ReLU activation function should not be used after the last linear layer of the Network. Additionally, the loss over the training at every epoch is saved in order to analyze the training process of the network.

The second experiment is a sanity check designed to evaluate how the performance results of the proposed framework compare to PyTorch. To ensure a fair comparison, a random seed is planted (so both networks are trained and tested on the same sets) and the same network architectures were implemented in PyTorch and the proposed framework. In both cases, the default bias and weight initialization is used with hyperbolic tangent as activation function.

V. Results and Discussion

In this section, the results obtained from the previously mentioned experiments are shown and briefly discussed.

TABLE I: Error percentage on the simple classification task using the new framework with different activation functions and initializations.

		Default init.		Xavier initi.	
	Error	Mean (%)	Std (%)	Mean (%)	Std (%)
Tanh	Training	3.50	0.41	3.41	0.43
	Testing	4.09	0.77	2.47	0.42
ReLU	Training	2.98	1.01	3.28	1.18
	Testing	2.98	0.87	5.03	1.41

TABLE II: Error percentage on the simple classification task using the new framework as well as PyTorch with fixed seeds.

		Mean Error (%)	Std Deviation (%)
Framework	Training	3.32	0.46
	Testing	3.49	0.62
PyTorch	Training	3.32	0.46
	Testing	3.49	0.62

Table I displays the performances of the Network implemented with the newly proposed framework on the binary classification task. The small error percentages indicate that all the different modules implemented in the framework work properly. Furthermore, results in table II show that the networks implemented in PyTorch and the newly proposed framework are consistent. Indeed, the errors at both testing and training are identical.

VI. Conclusion

The recent interest in deep neural network motivates the design of well-functioning and efficient deep learning frameworks. This project looked at the creation of a simple framework allowing to easily implement a neural Network composed of any combination of fully connected layers with hyperbolic tangent or ReLu as activation functions. The designed framework achieved its main goal to automate the forward pass as well as the backpropagation of simple neural networks. Results showed that the framework it is working properly with performances identical to PyTorch in terms of accuracy. Further work could implement more activation functions as well as more advanced modules such as convolutional layers. Additionally, other losses and optimizer could also be added to the framework in order to cover a wider range of classification and regression tasks.