ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

EE-559 Deep Learning

Mini-project 2: Implementing from Scratch a Mini Deep Learning Framework

Authors:
Terkel Bo Olsen
Anmol Porwal
Leandros Stefanou

Hand-in date: 18th of May 2018

1 Introduction

This report describes the implementation of a small framework which can be used for deep learning. It builds only on the basic Python libraries and the tensor operations provided from PyTorch [1] to speed up operations. The current framework implementation is minimal and includes only linear layers (fully connected) and a `Sequential` container for easy definition of simple models including sequentially connected layers. Further, it implements different loss and activations functions and optimizers. The implementation builds on the idea of ease of use and transparency, meaning that the regular Python syntax can be used and that gradients and parameters of each layer are easily available much in the same way as in PyTorch. However, it also implements wrapper functions that can be used to train a network and predict on test samples with a minimum amount of code, if needed.

2 Overview of the framework

The framework provides a simple class `Sequential` acting as a container for holding a type of sequential network. The user will use the class in a similar fashion as to the one implemented in the PyTorch network, by simply writing down the structure of the network. A simple example of usage can be seen below.

```
model = Sequential(Linear(X_train.size(1),25),
                   ReLU(),
                   Linear(25,25),
                   ReLU(),
                   Linear(25,25),
                   ReLU(),
                   Linear(25,2),
                   Tanh())
```

This command will define a sequentially connected model with the given layers of specific input and output sizes and the corresponding activation functions by storing them in an `OrderedDict`. Each layer, activation and loss function was implemented as classes all holding the same generalizable methods such that the forward and backward pass could easily be created. Further, as default, the weight and bias tensors of the layers are initialized using the Xavier initialization. [2]

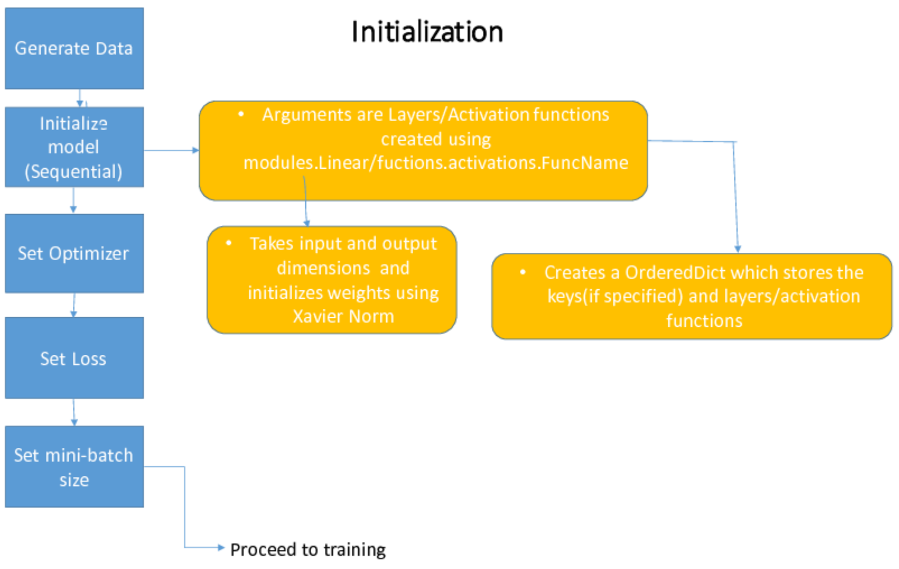
Initialization
Generate Data
Initialize model (Sequential)
Set Optimizer
Set Loss
Set mini-batch size
Proceed to training
• Arguments are Layers/Activation functions created using modules.Linear/fuctions.activations.FuncName
• Takes input and output dimensions and initializes weights using Xavier Norm
• Creates a OrderedDict which stores the keys(if specified) and layers/activation functions

Figure 1: Flowchart of the initialization phase before training commences.

Then as explained in the flowchart in Figure 1, the network has to be initialized using some of the provided functions before training commences. This includes, setting and defining the optimizer to be used, setting the loss function with the `setLoss` method of the `Sequential` model and then defining the mini-batch size. As the loss is not defined when creating the model using `Sequential`, it is evident that one sets this before training commences, if not done the framework will raise a `ValueError`. Moving on to training, the framework provides two convenient wrapper methods for the `Sequential` class to facilitate training. These are the `train` and `predict` methods, which allows to perform training and predictions in a single line of code. Then a simple example of usage of the framework can be constructed as presented below.

```
optimizer = SGD(eta = 1e-3)
model.setLoss(MSE())
mini_batch_size = 100

model.train(X_train,
            y_train,
            epochs = 1000,
            mini_batch_size,
            optimizer,
            verbose_every_nth=10)

predictions = model.predict(X_test)
```

From above, it is clear that after defining the model, one simply defines the optimizer, uses the `setLoss` method to set the loss function for training and defines the mini batch size. Then by using the `train` and `predict` methods one is able to easily specify a training routine using a minimum of code. It is off course not mandatory to use these wrapper functions and one could easily define a training procedure from scratch by using the generic `forward` and `backward` procedures as further specified in the flowchart in Figure 2.

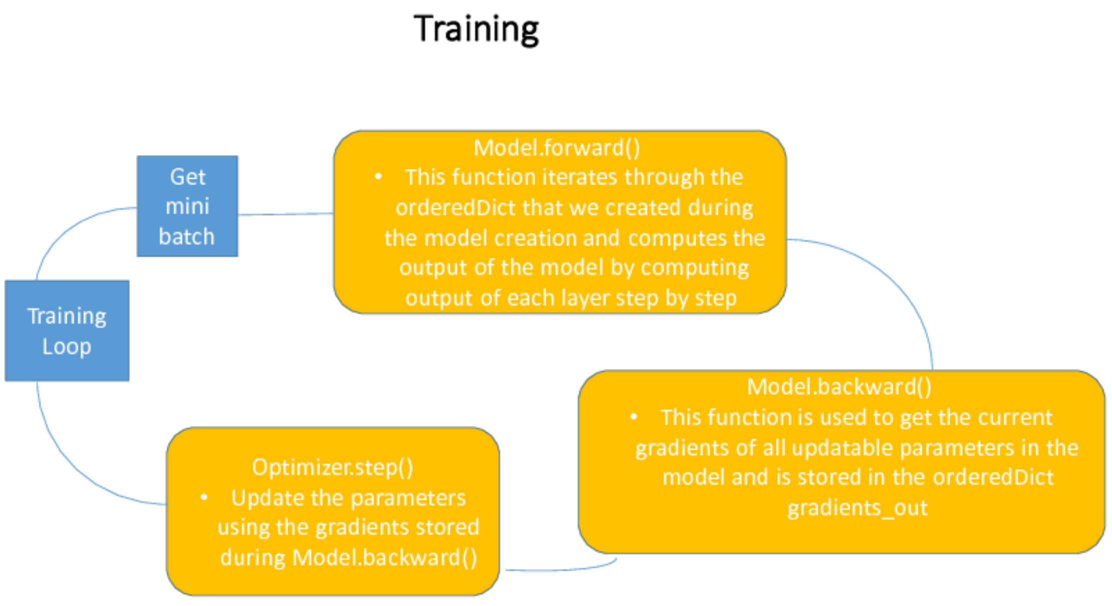
Training
Get mini batch
Model.forward()
• This function iterates through the orderedDict that we created during the model creation and computes the output of the model by computing output of each layer step by step
Training Loop
Model.backward()
• This function is used to get the current gradients of all updatable parameters in the model and is stored in the orderedDict gradients_out
Optimizer.step()
• Update the parameters using the gradients stored during Model.backward()

Figure 2: Mechanism of the framework responsible for the training procedure

As seen in Figure 2, the training procedure is very easily defined. First, the training loop is initialized and run for as many epochs as needed. Then another inner loop is initialized looping over each of the mini batches in the data set. If mini batch training is not used then the optimization problem will reduce to a non stochastic one, such that using the stochastic gradient descent procedure would actually amount to simply doing gradient descent.
For every mini batch processed, first the `forward` procedure is used to create predictions on the training set using the current model weights. This procedure internally works by looping over the `OrderedDict` created on initialization of the model, and sequentially calls the `compute` procedure on every element in the dictionary. The function further stores the output of each element in the model in another `OrderedDict` which is used during calculating of the gradients. Then the `backward` function is called with respect to the predicted output and the target labels. This function basically uses the chain rule to backpropagate the gradient through the network.[3], [4]
This amounts to first calculating the gradient with respect to the loss function and then sequentially looping back through the reversed `OrderedDict` as defined upon initialization. Our implementation of this procedure builds on the fact that every class for either layers or activation functions in the framework have a generic `propGrad` method, which internally propagates the gradient using the appropriate operator, i.e. matrix-multiplication for layers and point-wise multiplication for activation functions. For more details about the `backward` method, the reader will find the implementation in the `module.py` file.
Then, after having obtained the gradients from using the `backward` method, the gradients should be passed to the defined optimizer, using the implemented `step` method. It should be noted that this procedure works as an inplace operation inspired by the procedure used in the PyTorch framework.

3 Example of usage

Now, having described the flow of the framework during training and prediction, a simple example as seen below can be constructed to train on any given data set.

```python
# Assuming that a Sequential model has been created
# and stored in the variable "model":
for step in range(epochs):
    for b in range(0, X_train.size(0), mini_batch_size):
        X_train_batch = X_train.narrow(0,b,mini_batch_size)
        y_train_batch = y_train.narrow(0,b,mini_batch_size)
        output = model.forward(X_train_batch)
        gradients = model.backward(y_train_batch)
        optimizer.step(model, gradients)
    if np.mod(step,verbose_every_nth) == 0:
        print('Loss at iteration %i was %.2f' \
        % (step, model.loss.compute(output, y_train_batch)) )
```

As described in the previous section it is also possible to use the wrapper function `train` to perform the same procedure as above, if the user of the framework wants to use a minimum of time writing trivial code and use a maximum of time on experimenting with different configurations of networks. If the user wants to do so they should write code as was presented in the previous Python listing.

4 Discussion on further improvements and conclusion

In this report, a mini deep learning framework - building on the tensor operations provided by PyTorch - was presented. The basic idea of the framework is to provide a convenient set of methods to define simple sequential models using any type of layers, activations or loss functions. The current state of the framework only implements the `Linear` layer, `Tanh` and `ReLU` activation functions as well as the `MSE` loss function, but the framework was created with a mindset such that it would be easily extendable to include more attributes like a convolutional layer. Such an implementation could be easily made if the user defines the convolution operator and the corresponding gradient and then defines a new class making use of the standard generic function that should be included in every base class. These include an `__init__` function initializing the weight tensors, a `compute` function that takes an input and compute the output of the layer. Further, for the backward pass to work, one should include a `gradient` function, i.e. for convolutional layers this would be the corresponding transposed convolution. Finally, a function to backpropagate the gradient `propGrad` and two functions for wrapping and returning gradients and parameters of the layer; `returnParamGrads` and `returnLayerParams`.

Overall, the framework was designed such that extensions would be easily implemented while still providing a simple user interface allowing for easy experimenting with different network architectures using a minimum of coding. It draws inspiration from the PyTorch framework and uses the underlying Tensor operations to speed up computation time and hence the performance of the framework is good.

References

[1] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch”, 2017.

[2] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks”, in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning internal representations by error propagation”, California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[4] Y. LeCun, D. Touresky, G. Hinton, and T. Sejnowski, “A theoretical framework for back-propagation”, in *Proceedings of the 1988 connectionist models summer school*, CMU, Pittsburgh, Pa: Morgan Kaufmann, 1988, pp. 21–28.