Implementing From Scratch a mini deep Learning Framework

Miniproject 2 - A project in Deep learning Course @EPFL

Saleh Gholam Zadeh
saleh.gholamzadeh@epfl.ch
EPFL

Neeraj Yadav
neeraj.yadav@epfl.ch
IIT Kanpur

Selmane Kechkar
selamne.kechkar@epfl.ch
EPFL

Olagoke Lukman O.
lukman.olagoke@epfl.ch
EPFL

***Abstract*—Research in deep learning has gain a lot of momentum in recent times. Accordingly, a number of notable libraries or software API's have been churned out that offers robust implementations of the standard deep learning algorithm. However, a very effective and genuine way to learn about neural models is to build one simple model from the scratch. This will ensure a good grasp of the inner mechanism of the 'black box' deep learning models. Effectively, The objective of this project is to design in particular a mini 'deep learning framework' using only pytorch's tensor operations and the standard math library.**

I. Introduction

A. Deep Learning

A deep neural network (DNN) is an artificial neural network (ANN) with multiple hidden layers between the input and output layers. DNNs can model complex non-linear relationships. DNN architectures generate compositional models where the object is expressed as a layered composition of primitives. The extra layers enable composition of features from lower layers, potentially modeling complex data [?]. Generally, a network defines a mapping

$$f(x;\boldsymbol{\theta})$$

and learns the value of the parameters $\boldsymbol{\theta}$ that result in the best function approximation. A feedforward network is derived when information flow through the function evaluated from x, along intermediate computations used to define f and finally to output y. When feed forward network are endowed with feedback connections then we have a recurrent neural network. This project focuses on feed forward network model in particular.

B. Model Implementation Structure

The implementation structure follows the same pattern as described in the project description. The implementation structure consist of 4 class modules. The class modules are itemized below:

- class linear
- class Relu
- class Tanh
- class Sigmoid
- class identity
- class Sequential

In addition 2 functional module were defined

- function generate_disc_set
- function train_model

One internal functions were also declared :

- function __structurize__

The plot below shows the core module of our implementation.

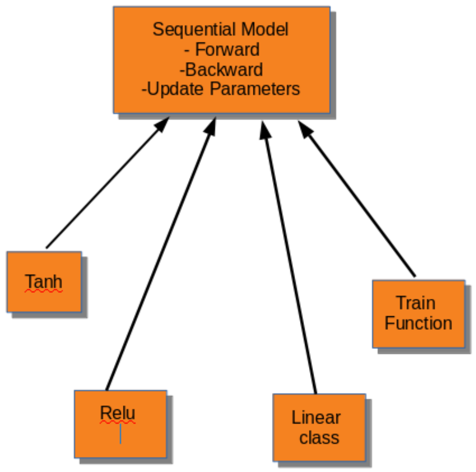


Fig. 1: Core Modules of The Implementation

The class modules will be explained in great details later. The internal function modules are defined with 2 underscore and are used for internal computations within the model structure. They can not be called by users. The __structurize__ function helps to separate the linear operations from the 'activation or nonlinear' operation in the list of operations parsed during function call. Essentially, it adds identity function from identity class whenever a linear operation is followed by activation function in model structure and collect this in a list of operations. It also collect the linear operations , non linear operations in a separate list. This will be important while trying to compute back propagation and while doing forward pass. The class Identity is used to handle where we have consecutive linear operations. Though this might not make sense in actual implementation structure, because combination of linear operation are replaceable by one linear operation. In such case, we embed an identity function in between the consecutive linear. Using this approach makes the entire program structure is independent of the number or order of linear units used before a nonlinear activation function is called. This gives flexibility to parse any architecture into the model - provided the tensor size input and target are consistent.

The function generate_dic_set generates data set for training and testing the model . The function train is used for training the model.

C. Model Mathematical Analysis

The most important mathematical formulation used during the model implementation will be discussed here specially for forward pass, backward pass and weight update. The figure below would be important for understanding the mathematical equations we used.

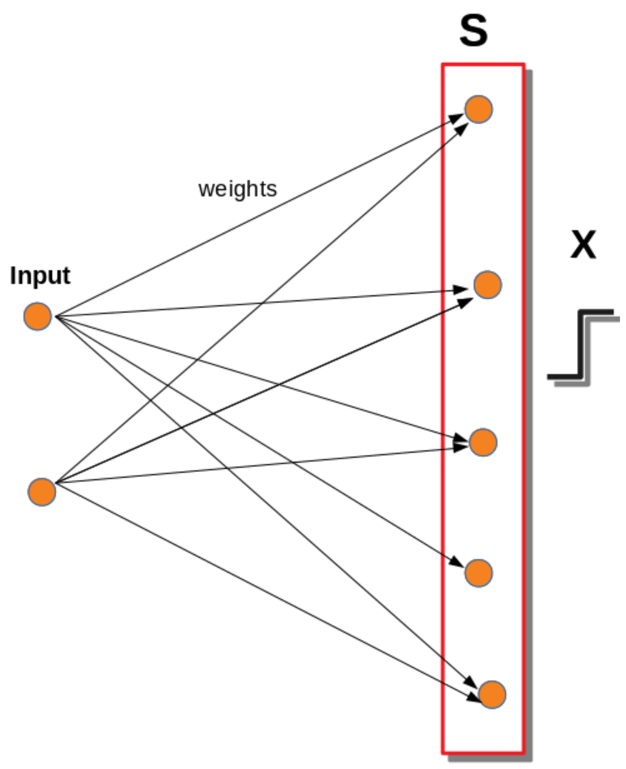


Fig. 2: Model Description.

We assume that we give input to the neural network. Then **S** represent the sum of the computations after application of weights and biases in other words $sy = wx + b$ -only weight is shown in the figure. Furthermore, **X** would represent the model output after the application of activation function.

1) Forward Pass: The Figure 3 below shows how the forward pass was computed in the model. Again, each s represent the sum of the linear units. Consequently, x is the output after application of activation. This separation of linear units before application of activation function results in formulation efficiency especially for the computation of back propagation.

Forward pass

Compute the activations.

$$x^{(0)} = x, \quad \forall l = 1, \ldots, L, \begin{cases} s^{(l)} = w^{(l)} x^{(l-1)} + b^{(l)} \\ x^{(l)} = \sigma\left(s^{(l)}\right) \end{cases}$$

Fig. 3: Equation for forward pass

2) Backward Pass: For the backward pass we compute the derivative of the loss to weight and bias in two stages:

$$\begin{cases} \left[\frac{\partial \ell}{\partial x^{(L)}}\right] = \nabla_1 \ell\left(x^{(L)}\right) \\ \text{if } l < L, \left[\frac{\partial \ell}{\partial x^{(l)}}\right] = \left(w^{(l+1)}\right)^T \left[\frac{\partial \ell}{\partial s^{(l+1)}}\right] \end{cases} \qquad \left[\frac{\partial \ell}{\partial s^{(l)}}\right] = \left[\frac{\partial \ell}{\partial x^{(l)}}\right] \odot \sigma'\left(s^{(l)}\right)$$

Fig. 4: First stage for Computation of weights

Big **L** represent the last layer. So that for the last layer we compute the derivative of l wrt x as in first case above. While for all layers less than **L** , $l <$ **L**, we use the second casey. After the computation of this derivative of layer wrt x,we do point-wise multiplication of the results with the derivative of activation function of s for the layer.

We also compute the loss with respect to parameters as below:

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

Fig. 5: Second stage for Computation of derivative wrt weight and bias

The derivative above makes use of the derivative computed in the first stage.

3) Gradient Update: For the gradient update, we proceeded as below:

$$w^{(l)} \leftarrow w^{(l)} - \eta \left[\left[\frac{\partial \ell}{\partial w^{(l)}}\right]\right] \qquad b^{(l)} \leftarrow b^{(l)} - \eta \left[\frac{\partial \ell}{\partial b^{(l)}}\right]$$

Fig. 6: Equation for gradient update

The equation above represent how we update the model parameters. η represents the learning color

II. Model Setup And Implementation

A. Implementation Deliverables

The following deep learning modules are available in the implementation:

- Sequential: combines several modules in basic sequential structure
- Forward : gets input, and returns, a tensor or a tuple of tensors.
- Backward: gets as input a tensor or tuple of tensors containing the gradient of the loss with respect to the module's output, accumulate the gradient wrt the parameters, and return a tensor or a tuple of tensors containing the gradient of the loss wrt the module's input.
- Parameters : returns a list of pairs, each composed of a parameter tensor, and a gradient tensor of same size. This list is empty for parameterless modules (e.g. ReLU).

The modules above abre all-inclusive: this means with these modules we can build networks - which combines fully connected layers, Tanh, Relu and Sigmoid - run forward and backward passes, while at the same time optimizing parameters with SGD for MSE.

In particular, Forward and Backward are both implemented for the Relu,Tanh,Sigmoid and Linear classes. This gives on flexibility to compute and collect forward and backward computation for those modules.

1) Sequential Module: The sequential module takes in a sequence of operations. It allow combination of operations sequentially. The allowed operations are linear and activation - from the class linear , Tanh, Relu and Sigmoid. The parameters is implemented as a function in the sequential module where it stores the model parameters

The Sequential class has the function parameters that returns the list of parameters for the model. If an operation has no optimizable parameter then empty list is simply returned. To return the parameters we check that the module 'backward' has been called.

In addition sequential module has the functions 'backward' and 'forward' too which when called checks the type of operation - in our case linear , Tanh, Sigmoid and Relu - and consequently apply the implemented forward or backward function for that operation. In this way we keep track of the operation and the derivatives.

The table in the next column shows the general outline for using the module sequential. We generate model units - linear , relu. We feed these into the sequential class and create a model. Thereafter, we compute the forward and backward propagation respectively. For the forward propagation we can get the result of the operation either for the linear or activation module (relu) by calling $model.linear\_result$ or $model.activation\_result$ respectively. In addition both results are included in the output after calling forward. Similar operation hold for the backward operation but in this case we get gradient of weight and bias.

Algorithm 1 NeuralModel: Using Sequential Module

```
1: Instantiate the model objects
       $linear1 = linear(2, 5)$
       $linear2 = linear(5, 4)$
       $relu = Relu()$
       $linear3 = linear(4, 3)$
       $input\_variable = torch.zeros(2, 1)$
       $target\_variable = torch.zeros(3, 1)$
2: Parse model objects into Sequential
       $model = sequential(linear1, linear2, relu, linear3)$
3: Compute Forward
       $forward = model.forward(input\_variable)$
       $model.linear\_result \quad //view \quad result$
       $model.activation\_result \quad //view \quad result$
4: Compute Backward
       $backward = model.backward\_pass(target\_variable)$
       $model.dl\_dw \quad //view \quad weight \quad gradient$
       $model.dl\_db \quad //view \quad bias \quad gradient$
```

B. Training a Model

The training of a model is done using the function train_model: As depicted below, training takes in the model as

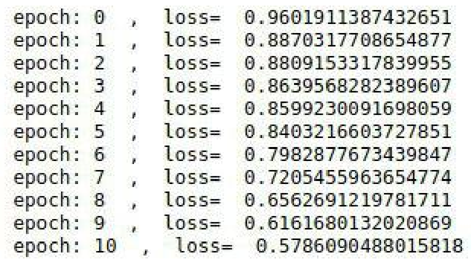
Algorithm 2 NeuralModel: Training a Model

```
1: Training a Model
       $train\_model(model, train\_input, train\_target,$
       $mini\_batch\_size = 1, epoch = 25, learning\_rate = 0.1)$
```

input, together with 4 other parameters. These parameters can be tweaked while training the model. This training function is basically implemented as for loop over the batch size. At each iteration we compute the forward and backward pass. The update function of sequential updates the parameter per iteration. In parallel the loss function of the sequential computes the MSE loss per iteration. But for each epoch we compute the normalized loss. Finally we used the zero_grad function of the sequential module to zerorise the accumulated derivatives at the end of each loop in the iteration.

C. Testing our model

For testing our model we generated a training and a test set of 1000 points sampled uniformly in $[0, 1]^2$, each with a label 0 if outside the disk of radius $\frac{1}{\sqrt{2\pi}}$ centered at (0.5,0.5) and 1 if it is inside, then we built a network with two input units two output units and three hidden layers of 25 units then we trained the network by considering MSE loss. We print the train below:



```
epoch: 0  ,  loss=  0.9601911387432651
epoch: 1  ,  loss=  0.8870317708654877
epoch: 2  ,  loss=  0.8809153317839955
epoch: 3  ,  loss=  0.8639568282389607
epoch: 4  ,  loss=  0.8599230091698059
epoch: 5  ,  loss=  0.8403216603727851
epoch: 6  ,  loss=  0.7982877673439847
epoch: 7  ,  loss=  0.7205455963654774
epoch: 8  ,  loss=  0.6562691219781711
epoch: 9  ,  loss=  0.6161680132020869
epoch: 10  ,  loss=  0.5786090488015818
```

Fig. 7: Loss Values Generated during training. Only 10 epoch is shown

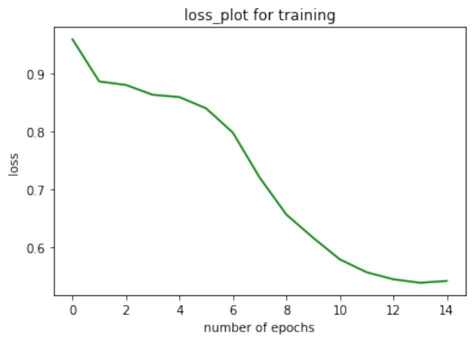


Fig. 8: Loss plot for training

From the plot above, we observe that the loss values decays gradually. This corresponds to the expected behaviour. Since we are training in batches per epoch, we expect not to get a smooth decay - thats why we have some points where the loss actually tries to increase.

III. Conclusion

With this project we gained a deep intuition about how neural Nets work and the computational challenges related to it. Also we observed the need for mathematical reformulation of concept when going from theory to efficient implementation - because we need an efficient algorithm. Finally we see every thing from scratch and we understand how other neural nets frameworks work