Deep Learning
Mini Project 2

Mammad Hajili **294808**, Reza Hosseini **281368**, Alexandru Mocanu **295172**

May 2019

1. Introduction

In this project, we have tried to design our mini deep learning framework from the scratch. As it has been instructed, we only used basic torch.Tensor object without any Autograd functions or Neural Network modules. For this purpose we put turned off the Autograd machinery using torch.set_grad_enabled(False).
The logic behind the codes was inspired from PyTorch library, and then we modified it using our ideas to avoid defining unnecessary classes and methods which will not be useful for the purpose of our project.

2. Code's Structure

The main structure can be broken down into three main parts:

1. **Module:** This is the base class for Neural Network modules. It contains linear layers, activation layers, losses, and sequential module classes. All of the codes for this class can be found in nn.py source file.
2. **Parameter and Variable:** Parameter is a torch.Tensor subclass that we implemented for keeping track of the parameters that should be trained during the training phase. We implemented this architecture in a way that Parameters can be automatically added to the Module sub-classes after its declaration. On the other hand, Variable is a torch.Tensor sub-class that we use to store previous leaves of the computation graph, including the previous module and inputs for using, which will be used in the backward pass.
3. **Functional:** This is a set of functions that we wrote that will compute the forward pass and gradients of various functions that are needed in the Modules.
4. **Optimizer:** This class has been used as a base class for all optimizers.

2.1 Module

We designed Module class in a way that it automatically sees and adds the parameters to the class. We also added the ability for a module to contains sub-modules. Sub-modules again can be seen and added automatically to a module. This enabled us to not only implement Sequential, but also to able to implement more complex architectures like ResNet that requires more complex modules. The process of defining a non-sequential module is exactly the same as definition of a custom module in PyTorch. The only difference is that for the definition, one may only use the already defined modules (e.g. Linear, ReLU, and etc.) and not directly the functions inside Functional. However, any user-defined modules can be used for this purpose. The following modules has been implemented in our project:

- **Linear Layers**
- **Activation Layers:** relu, tanh, softmax.
- **Losses:** MSE loss, cross entropy loss.
- **Sequential**

2.2 Parameter and Variable

- **Parameter:** The attribute grad_f indicates whether this parameter should be updated or not, and the attribute grad_v keeps the gradient values of the parameter. grad_f is useful in some more complex situations, such as *transfer learning* where we don't want to update the trained parameters of the previously trained network. The updating behavior is controlled in the Optimizer class.
- **Variable:** Since we didn't need to define something as complex as *Autograd* machinery which has the native ability of computing the gradients for its functions, in our design we combined torch.Autograd.Function with torch.nn.Module class with the help of custom-defined Variable class. Any gradients is then stored through grad_v attribute in the Variable and *computation graph* is stored using previous_inputs and previous_module attributes. Thus, for being able to compute the gradients correctly in the backward pass, one should only use the modules that we have already defined, or uses Module and Variable class together to define a new user-defined module, which should contain both the *forward* and *backward* method, the same as defining a new function using torch.Autograd.Function class.

2.3 Functional

Functional contains all functions that are needed to for the defining the modules and their gradients. We chose this structure because it made implementing the modules easier. The implemented functions are *linear, mse_loss, relu, tanh, sigmoid, softmax, cross_entropy* and their derivatives respectively,

2.4 Optimizer

Optimizer class is a base class for all the optimizers. It updates the *parameters* that needs to be optimized in each step. Optimizers control the ability to whether update a parameter or not using grad_f field. It gets parameters list that needs to be optimized and also any other relevant *hyper parameters*, which is specific to the instance using others attribute. We have implemented the following optimizers:

- *SGD*
- *Adam*

3. Source Files

We organized the structure of our project's codes using the following source files:

- ***functional.py*** This source file contains all the functions that we explained in the 2.3 Functional
- ***nn.py*** It contains source code for Module, Variable, and Parameter classes.

- ***optim.py*** Contains Optimizer classes, including the base class and SGD and Adam implementations.
- ***helper.py*** This contains helper functions that are necessary for our project, including *generate_dataset*, *normalize*, *one_hot_encoding*, *train_model*, *compute_nb_errors*
- ***test.py*** This is the main executable file which runs the whole project and reports the test and train set accuracy.

4. Model Architecture

We design the two following models for predicting the target labels. The modules has been created to showcase the implementation of our architecture.

- **Sequential:** In this model, we combined two fully connected and ReLU activation sequentially and have used Cross Entropy loss for training the model. All of them has been done using nn.Sequential module.
- **Custom:** This model has the same architecture of the model above. The goal of creating this model was to show the flexibility of our designed architecture. This allows one to implement more complex custom-defined modules as depicted in the source code.

5. Conclusion

In conclusion, we tried to design the structure of our mini deep learning architecture as flexible as possible, given the objective of this project. We tried to implement the Autograd Machinery by ourselves in the level of *modules*, instead of the native Autograd compliance functions in PyTorch. Then, we showcased how to define a *sequential* and *user-defined* modules using our designed architecture. We also optimized the necessary hyper parameters to make it easy to run for the evaluation by TAs. We must mention that we did not use all of the defined modules and functions in these showcases, however it is easy to see how this simple architecture can be extended upon the request.