Deep Learning Miniproject 2

Xiao Zhou 294916
xiao.zhou @epfl.ch

***Abstract*—The objective of the project is to design a mini "deep learning framework" without standard math libraries, autograd or the neural network modules in pytorch. To be specific, I will only use the tensor operations in torch to construct a neural network for binary data points classification based on 2D coordinates. Besides the default neural network as required, I also built several additional modules for more flexible combination of network. After performance comparison with standard pytorch modules, my modules prove to be competitive, and can reach the accuracy to around 0.95.**

I. Data Generation

Each of the training set and testing set consist of 1000 data points that are sampled uniformly in $[0, 1]$ in 2 axis in one plane. For each set, the data points that are within the distance of $\frac{1}{\sqrt{2\pi}}$ from the center coordinate $(0.5, 0.5)$ are labeled as 1 while the points outside the circle are labeled as 0. Thus, the task can be interpreted as a binary classification problem, and both positive-labeled and negative-labeled data points have a probability of 0.5 in theory so that the ground truth categories of the classification is unbiased.

II. Basic Module

The basic module is the module that will be inherited by more flexible modules in the next sections. The basic functions of the basic modules include **forward**, **backward** and **param**. The **forward** function will have output tensors from previous layer as input and return the tensors to forward to next module after internal calculation. The **backward** function gets a tensor input that represents the loss gradient accumulated in all of the next layers in the network. It will return the tensor that contains the gradient with respect to the current module's input from **forward** and then sends to previous layer as the accumulation of loss gradient. The **param** contains all the trainable parameters in the module that can be updated by optimizers in the training process.

The following modules will inherit from this basic module, and some will also have their unique functions according to their expected properties when building a neural network.

III. Linear Layer Module

The **Linear** module represents a fully-connected layer in the network, and it contains the majority of trainable parameters in the learning process.

A. Initialization

My **Linear** module offers 3 initialization methods, including zero initialization, normal distribution initialization and Xavier initialization. The defaulted initialization is designed as Xavier's method:

$$w, b \sim N(0, \frac{\sqrt{2}}{\sqrt{(N_{in} + N_{out})}}) \tag{1}$$

B. forward

The forward pass conforms to the theoretical calculation of input tensor by weight and bias parameters:

$$y = wx + b \tag{2}$$

where x is the input from previous layer, y is the output tensor to next layer, w represents the weights in the layer and b the bias. Also, the tensors are also stored after one forward calculation, which helps to track the information during backward pass.

C. backward

In the backward pass, the module will use the parameter tensors stored during forward pass as well as the loss gradient accumulated from next layers. For the trainable parameters of weights and bias, the stored gradient will be calculated as:

$$\frac{dL}{dw} = \frac{dL}{dw} + \frac{dL}{dy} \cdot x \tag{3}$$

$$\frac{dL}{db} = \frac{dL}{db} + \frac{dL}{dy} \cdot 1 \tag{4}$$

Also the backward pass will calculate the accumulated loss gradient wrt forward input before sending to previous layer:

$$\frac{dL}{dx} = w^T \cdot \frac{dL}{dy} \tag{5}$$

D. zero_grad

The function will be set the dw and db to zero for new training batch.

E. param

The information of w, b, dw and db will be stored for updates of trainable parameters according to optimizer.

IV. Regularization Module - Dropout

A dropout layer helps increase independence between units and distributes the representation. In general, it prevents co-adaptation by making the presence of other hidden units unreliable and improves the performance suffering from overfitting. When building the dropout module, I created a Bernoulli matrix by setting probability and input size in **forward**, and store the matrix for corresponding **backward**. Thus the **forward** calculation is the matrix multiplication of Bernoulli matrix and input, while **backward** calculation is multiplication of Bernoulli matrix and accumulated loss gradient. There is no trainable parameters in the layer.

V. Activation Module

The activation model can bring non-linearity to the model. They can be interpreted as non-linear functions during forward and backward pass without trainable parameter.

1) ReLU:

$$y = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{6}$$

$$\frac{dL}{dx} = \frac{dL}{dy} \cdot \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{7}$$

2) Tanh:

$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{8}$$

$$\frac{dL}{dx} = \frac{dL}{dy} \cdot \frac{4}{(e^x + e^{-x})^2} \tag{9}$$

Besides **ReLU** and **Tanh** as required in the project description, I also built **Sigmoid**, **SeLU** and **Softmax** in this section. In theory, **SeLU** can have a good property of keeping mean and prevent from gradient vanishing, and **Softmax** can be a good option in multi-class classification problem.

The detailed calculations in forward and backward pass in these additional activation functions are available in **APPENDIX**.

VI. Sequential Module

The sequential module helps to combine the previous built modules in a sequential structure to construct a neural network. The **forward** utilizes a loop to pass and calculate the output from original input to final network output before sending to loss functions. Similarly, **backward** passes loss gradient in accumulation from final to first layer and accumulates gradients for trainable parameters in internal modules. The **param** returns all the trainable parameters and corresponding gradients in the whole network, waiting to be updated by optimizers.

VII. Loss Functions

The loss function receives the output from sequential network and calculate the final classification loss, and sends the computed gradient back to the network for updating parameters. In the project, I mainly built mean square error (MSE) and binary cross entropy as the the criterion loss function.

A. MSE

$$L = (y - \hat{y})^2 \tag{10}$$

$$\frac{dL}{dy} = 2 \cdot (y - \hat{y}) \tag{11}$$

where y is predicted value from sequential network and $\hat{y}$ is the ground truth label.

B. Cross entropy

$$L = -\hat{y_1} \cdot log(y_1 + \epsilon) - \hat{y_2} \cdot log(y_2 + \epsilon) \tag{12}$$

$$\frac{dL}{dy_1} = -\frac{\hat{y_1}}{y_1 + \epsilon} + \frac{\hat{y_2}}{1 - y_1 + \epsilon} \tag{13}$$

$$\frac{dL}{dy_2} = -\frac{\hat{y_2}}{y_2 + \epsilon} + \frac{\hat{y_1}}{1 - y_2 + \epsilon} \tag{14}$$

It should be noted that I add a small ϵ (default as 1e-2) in the above equations in forward and backward pass to alleviate gradient explosion issues.

VIII. Optimizers

The optimizers determine the way to update trainable parameters in each step, based on the accumulated gradients in the backward pass. Two optimizers are constructed, including stochastic gradient descent (SGD) and Adam optimizer.

A. SGD

My SGD optimizer is based on mini-batch pattern, and it'll update the parameters after finishing the forward and backward pass of a batch of data. Before the learning of each batch, it'll also set the parameter gradients in the model to zero. If the batch size is set as 1, it will perform as a standard SGD. If the size equals the length of whole data set, it can be interpreted as GD. The update in each step is shown as follows:

$$p = p - \gamma \cdot \frac{dL}{dp} \tag{15}$$

where p represents the trainable parameter in sequential model (the weights and bias in linear modules), and γ is the learning rate.

B. Adam

Adam optimizer is also constructed to update parameters in a batch fashion. It uses the squared gradients to scale the learning rate and takes advantage of momentum by using moving average of the gradient. It can be interpreted that it computes individual learning rates for different parameters. Adam is an effective optimization algorithm that can help achieve fast convergence in some cases. The detailed calculation for updating parameters in Adam optimizer in each step will be given in **APPENDIX**.

IX. Testing my module

A. Building models

So far I have finished constructing the modules that can be used for building a neural network for the binary classification mission.

Model 1 is designed as demanded in project description. The model contains 3 hidden fully-connected layers with hidden output dimension of 25, and the output dimension of final fully-connected layer is 2. ReLU is selected as the activation module after each hidden FC layer, while Tanh after the final FC layer. The criterion loss function is MSE and the optimizer is SGD.

Model 2 is combined with some additional modules to improve the learning performance. The FC layers are the same while a Dropout layer is added after the hidden FC layer. Also, the activation modules after hidden layers are combined with ReLU and SeLU, and the final activation is Softmax. Binary cross entropy is chosen as the criterion loss function and Adam optimizer is used for updating trainable parameters.

```
my_model_design_1=[Linear(2,25), ReLU(), Linear(25,25), Dropout(p=0.5), ReLU(),
                   Linear(25,25), ReLU(),Linear(25,2),Tanh()]
my_model_1=Sequential(my_model_design_1)
optimizer_1=SGD(my_model_1,lr=1e-3)
criterion_1=LossMSE()


my_model_design_2=[Linear(2,25), ReLU(), Linear(25,25), Dropout(p=0.5), SeLU(),
                   Linear(25,25),Dropout(p=0.5), ReLU(),Linear(25,2),
                   Softmax()]
my_model_2=Sequential(my_model_design_2)
optimizer_2=Adam(my_model_2,lr=1e-3)
criterion_2=CrossEntropy()
```

Figure 1: Building 2 models by my modules

In Figure 1, it can be observed that sequential network model, criterion loss function and optimizers can be called in an easy and straightforward way according to my modules. Afterwards, the 2 models will be used to train and test on the data sets.

B. Learning performance

The learning curves of 2 models by my modules, as well as the 2 models by standard Pytorch modules are shown as follows:

According to Figure 2, Model 2 has a better final accuracy and faster convergence in both modules compared with Model 1. The final accuracy of my module and torch module in one model is close. One difference is that Model 1

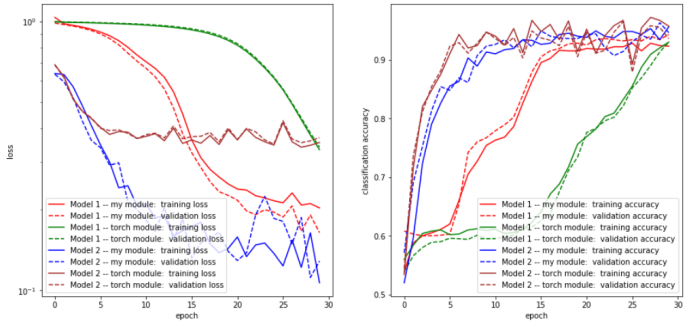

Figure 2: Learning curves of 2 models by my module and torch module **(validation = testing)**

with torch module seems to converge more slowly than my module.

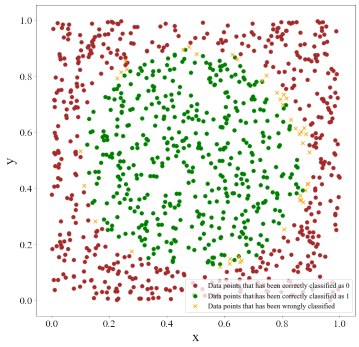

Figure 3: Classification performance of Model 2 by my module on testing set

From Figure 3, most data points in testing set can be classified correctly, and the wrongly classified points are near the boundary of the decision circle. Since the model has no priori knowledge of the circle information, the performance is acceptable.

Besides, to have a statistical view of model performance. I run each model by different modules 10 times and obtain the statistics as follows:

Model	Testing Accuracy	Standard Deviation	Time (s)
Model 1 (mine)	0.925	0.036	23.98
Model 1 (torch)	0.927	0.067	18.32
Model 1 (mine)	0.952	0.028	37.11
Model 2 (torch)	0.949	0.015	29.86

Table I: Comparison table of models by my module and torch module after 10 runs of 30 epochs

It can be observed from Table 1 that the accuracy performance is very close to standard torch modules. However, my module seems to take more time, especially in Model 2, and this remains to be optimized in future work.

X. Conclusion

In this project, I build some modules to create neural network models for a mini deep learning framework. These

include linear layer, dropout layer, activation functions, sequential structure, loss functions and optimizers. The accuracy performance of my modules is close to standard torch library and can achieve good classification results.

XI. Appendix

A. Calculation in additional activation modules

1) Sigmoid:

$$y = \frac{1}{1+e^{-x}} \tag{16}$$

$$\frac{dL}{dx} = \frac{dL}{dy} \cdot \frac{e^{-x}}{(1+e^{-x})^2} \tag{17}$$

2) SeLU:

$$y = \begin{cases} \lambda \cdot x & x \geq 0 \\ \alpha \cdot (e^x - 1) & x < 0 \end{cases} \tag{18}$$

$$\frac{dL}{dx} = \frac{dL}{dy} \cdot \begin{cases} \lambda & x \geq 0 \\ \alpha \cdot e^x & x < 0 \end{cases} \tag{19}$$

where λ and α are default as 1.

3) Softmax:

$$y = \frac{e^x}{\sum_{i=1}^{N} e^i} \tag{20}$$

$$\frac{dL}{dx} = \frac{dL}{dy} \cdot \frac{\sum_{i=1}^{N} e^i - e^x}{(\sum_{i=1}^{N} e^i)^2} \tag{21}$$

B. Step Calculation in Adam Optimizer

$$m_{t+1} = \beta_1 \cdot m_t + (1-\beta_1) \cdot \frac{dL}{dp} \tag{22}$$

$$v_{t+1} = \beta_2 \cdot v_t + (1-\beta_2) \cdot (\frac{dL}{dp})^2 \tag{23}$$

$$\hat{m} = m_{t+1}/(1-(\beta_1)^t) \tag{24}$$

$$\hat{v} = v_{t+1}/(1-(\beta_2)^t) \tag{25}$$

$$p = p - \gamma \cdot \hat{m}/(\hat{v}^{0.5} + \epsilon) \tag{26}$$