Deep Learning Hardware Accelerator Design

Lab 1: Multilayer Perceptron (MLP)

Name: 盧允凡

Student ID: 107065507

Titanic Survival Predictions problem

1. **Data preprocessing**

First, 'import pandas' to help us to process data.

```
training = pd.read_csv("titanic/train.csv");
```

In the train.csv file, we can see that there are 891 data, and each of them have 10 features.

Because there are some missing data in some features, to deal with it, I count the missing numbers in each feature. If most of the data are NaNs, I will drop the feature.

```
training.isna().sum()
```

```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

Thus, I drop the feature 'Cabin'.

I also drop 'Name' to make the data preprocessing easier.

I just simply assign the numbers to 'Sex' and 'Embarked'. That is,

```
repCol3 = {"male":0, "female":1}

repCol8 = {"C":0, "Q":1, 'S':2}
```

Next, I combine 'SibSp' and 'Parch' to a new category 'Family', and add another new feature 'IsAlone' according to the number of 'Family'.

```
x_train['Family'] = x_train ['SibSp'] + x_train['Parch']

x_train['IsAlone'] = 1

x_train['IsAlone'].loc[x_train['Family'] > 0] = 0
```

Normalization

To make the MLP engine work successfully, we need to normalize the data.

Otherwise, it could be overflow.

```
for col in x_train.columns[1:]:
    x_train[col] = x_train[col] / x_train[col].max()
```

Therefore, we have all the data between 0 and 1.

Finally, we have total 7 features and all of them are normalized.

2. MLP architecture & hyper-parameter

By using the Multi-Layer Perceptron architecture, we input some training data in our MLP engine to get a neural network (NN). After, we can input the testing data to show how the testing accuracy works.

The following are the basic concepts of MLP,

The layer 0 is called the "input layer". The original data will input to it.

The middle 2 to i-1 layers are "hidden layer". We can choose to set how many hidden layers. The last layer is "output layer".

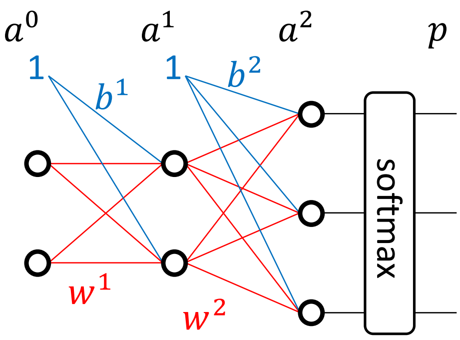

Figure 1. A neural network model inputs a^0, outputs p

The w_{ij}^l is the **connecting weight** from input node j to output node i,

and the b_k^l is the **bias** to output node k.

In my implementation, I initialize the weight and bias by random.

The MLP contains Forward and Backward parts.

Forward

Linear equation

$$z^l = w^l a^{l-1} + b^l, \forall l \neq 0$$

there is a relationship between *z* and *a* called activation function.

Activation functions

$$\begin{cases} \text{ReLU:} & a_i^l = \max(z_i^l, 0) \\ \text{sigmoid:} & a_i^l = \dfrac{1}{1+e^{-z_i^l}} \end{cases} \quad , \forall i, 0 < l < L$$

in my implementation, I use sigmoid function.

Finally, the output layer uses the softmax function to get the predicted answers.

Backward

Given a data set (a^0, y), find w^l and b^l, such that loss function J is minimized.

Loss function:

$$J = -\sum_{i=0}^{d} (y_i \ln p_i + (1 - y_i) \ln(1 - p_i))$$

in my implementation, the loss function reduces to

$$J = -\sum_{i=0}^{d} y_i \ln p_i$$

we have to derive $\frac{\partial J}{\partial w^l}$ and $\frac{\partial J}{\partial b^l}$ with $J, p, y, a^l, z^l, w^l, b^l$, for all l.

Ignoring the details of derivation.

We have the result,

$$\frac{\partial J}{\partial w^l} = \begin{bmatrix} \frac{\partial J}{\partial w_{00}^l} & \frac{\partial J}{\partial w_{10}^l} & \cdots \\ \frac{\partial J}{\partial w_{01}^l} & \frac{\partial J}{\partial w_{11}^l} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} = \begin{bmatrix} \frac{\partial J}{\partial z_0^l}\frac{\partial z_0^l}{\partial w_{00}^l} & \frac{\partial J}{\partial z_1^l}\frac{\partial z_1^l}{\partial w_{10}^l} & \cdots \\ \frac{\partial J}{\partial z_0^l}\frac{\partial z_0^l}{\partial w_{01}^l} & \frac{\partial J}{\partial z_1^l}\frac{\partial z_1^l}{\partial w_{11}^l} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

$$= \begin{bmatrix} \frac{\partial J}{\partial z_0^l} a_0^{l-1} & \frac{\partial J}{\partial z_1^l} a_0^{l-1} & \cdots \\ \frac{\partial J}{\partial z_0^l} a_1^{l-1} & \frac{\partial J}{\partial z_1^l} a_1^{l-1} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} = \begin{bmatrix} a_0^{l-1} \\ a_1^{l-1} \\ \vdots \end{bmatrix} \begin{bmatrix} \frac{\partial J}{\partial z_0^l} & \frac{\partial J}{\partial z_1^l} & \cdots \end{bmatrix} = a^{l-1} \cdot \frac{\partial J}{\partial z^l}$$

$$\frac{\partial J}{\partial b^l} = \begin{bmatrix} \frac{\partial J}{\partial b_0^l} & \frac{\partial J}{\partial b_1^l} & \cdots \end{bmatrix} = \begin{bmatrix} \frac{\partial J}{\partial z_0^l}\frac{\partial z_0^l}{\partial b_0^l} & \frac{\partial J}{\partial z_1^l}\frac{\partial z_1^l}{\partial b_1^l} & \cdots \end{bmatrix} = \begin{bmatrix} \frac{\partial J}{\partial z_0^l} & \frac{\partial J}{\partial z_1^l} & \cdots \end{bmatrix} = \frac{\partial J}{\partial z^l}$$

Update

simply, by the Optimization algorithm (*Gradient Descent*), we have,

$$w^{l'} = w^l - \epsilon \cdot \left(\frac{\partial J}{\partial w^l}\right)^{\mathrm{T}}$$

$$b^{l'} = b^l - \epsilon \cdot \left(\frac{\partial J}{\partial b^l}\right)^{\mathrm{T}}$$

where ϵ is the learning rate.

Training

Giving a number of Epoch, and runs the **Forward**, **Backward** and **Update** of total Epoch times.

Finally, we have trained a neural network (NN).

3. Experiment results

Titanic Problem from Kaggle:

Setting an neural network (NN) by the following parameters.

```
nnTitanic = MLP([9, 23, 2], activationFunction = "sigmoid")
nnTitanic.train(input_x_train, input_y_train, numEpoch=7000, lr=0.25, bs=3)
```

and the **Training Accuracy** = 0.8595505617977528

Following, we input the testing data to our neural network (NN) we just trained.

We got the result of the **Testing Accuracy** = 0.8277945619335347

MNIST hand-written digit recognition:

For training data of 60k.

Setting an neural network (NN) by the following parameters.

```
nnMNIST60k = MLP([784, 80, 10], activationFunction = "sigmoid")
nnMNIST60k.train(train_x_60k, train_y_60k, numEpoch=1000, lr=0.2, bs=500)
```

and the **Training Accuracy** = 0.9419333333333333

Following, we input the testing data of 10k to our neural network (NN) we just trained.

We got the result of the **Testing Accuracy** = 0.9375

Reference

[1] https://www.kaggle.com/moghazy/simple-mlp-with-feature-engineering-and-eda

[2] Terence Parr, Jeremy Howard, “The Matrix Calculus You Need For Deep Learning”, CoRR abs/1802.01528 (2018)