COMP5434 BIG DATA Computing
Group Project

Group ID：22

Class：Monday

Group Members:

Wang Ziqin 王梓秦 22061726g@connect.polyu.hk

Xia Tianyu 夏天宇 22073597g@connect.polyu.hk

Yang Congrui 杨聪睿 22049609g@connect.polyu.hk

Yang Yaxuan 杨雅萱 22061885g @connect.polyu.hk

1 Group Members’ Duty

Task1: Wang Ziqin
Task2: Yang Congrui
Task3: Xia Tianyu; Yang Congrui; Wang Ziqin
Bonus1: Xia Tianyu; Wang Ziqin
Bonus2: Wang Ziqin; Yang Yaxuan
PPT: Xia Tianyu; Wang Ziqin; Yang Yaxuan
Presentation: Yang Congrui; Wang Ziqin
Report: Yang Yaxuan

2 Introduction

1.1 MapReduce

MapReduce is a programming model for parallel computing on large data sets. It divides data into small chunks, maps these chunks into key-value pairs using the Map function, aggregates the values of the same keys using the Reduce function, and outputs the final result.
The MapReduce programming model consists of two main components: the Map function and the Reduce function. The Map function is used to decompose the input data into several small data blocks and map these small data blocks into key-value pairs, while the Reduce function is responsible for aggregating the values of the same keys to obtain the final result.
MapReduce is suitable for scenarios where large data sets are processed and the data can be broken down into smaller chunks. For this type of data, MapReduce's parallelization feature can make full use of multiple computers in a large cluster to improve data processing efficiency and performance.
Since MapReduce processes data by dividing it into several small data blocks and using Map functions to map these small data blocks into key-value pairs, MapReduce is more suitable for processing structured data, such as log files, sensor data, user transaction records, and other structured data sets.
However, for unstructured data, such as text data and image data, some preprocessing steps are required to make it ready for MapReduce processing. For example, for text data in this problem, we use a word splitting tool to split the text, while for image data, it is usually necessary to convert the image to vector or matrix form first.

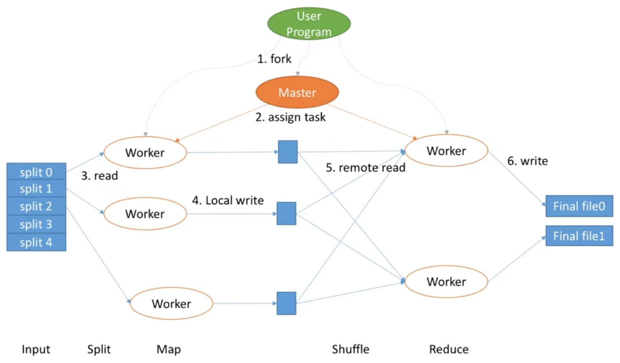


Fig.1 MapReduce architecture

1.2 CNN

CNNs are a commonly used class of neural networks that are widely used for processing tasks in areas such as images, speech and text. Compared with traditional fully connected neural networks, CNNs utilize two special layers, convolutional and pooling layers, to extract spatial features from the input data, thus significantly reducing the computational effort while keeping the number of model parameters constant.

CNN models usually consist of multiple convolutional, pooling and fully connected layers. The convolutional layer is responsible for extracting local features in the image, while the pooling layer is used to downsample the image to reduce the number of parameters. Finally, the fully connected layer takes the features extracted by the convolutional and pooling layers and performs tasks such as classification or regression.

For each layer, CNN models can also use different activation functions, such as ReLU, Sigmoid, and Tanh, to enhance the nonlinear capabilities of the model. In addition, CNN models also often use techniques such as batch normalization and random deactivation to speed up training and prevent overfitting.In essence, this is a multi-classification problem, which can be solved with the softmax model. In order to return the training weight, we use the neural network framework to train the model, but the most important part of the model is only the linear layer.

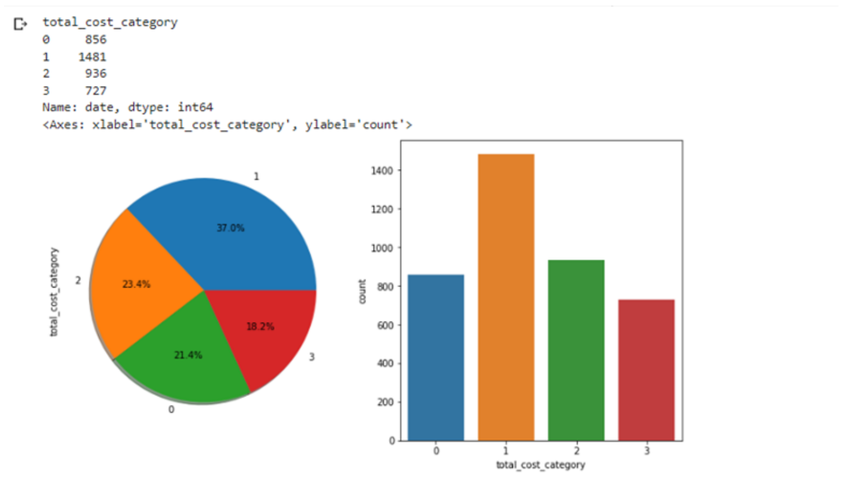
1.3 Federated Learning

Federated Learning is an emerging machine learning framework that aims to jointly train machine learning models while protecting data privacy by merging data and computing resources scattered across multiple devices or data centers.

In federated learning, different devices or data centers have local data, and the training process of machine learning models does not need to transfer these data to centralized servers. Instead, each device or data center performs certain computing tasks locally, and transmits local model parameter updates to the central server for aggregation, thereby gradually improving the global model

3 Data preprocessing & analytics

3.1 Visual analysis of the four spending price ranges

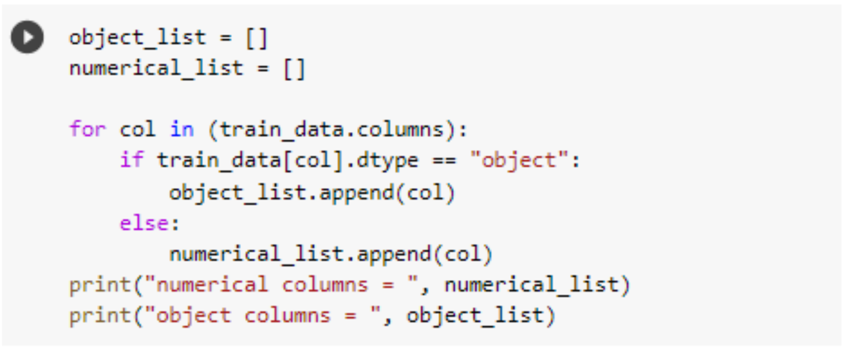


```
total_cost_category
0     856
1    1481
2     936
3     727
Name: date, dtype: int64
<Axes: xlabel='total_cost_category', ylabel='count'>
```

Fig.2 Visualization of the number of each consumption interval

According to Fig.2 we can visually get the number of the four price ranges and the proportion of the number of each price range. The highest percentage of ‘The total cost is greater than or equal to 300000HKD and less than 500000HKD’ is 37%; ‘The total cost is greater than or equal to 700000HKD’ has the lowest percentage, accounting for only 18.2%.

3.2 Data pre-processing -- conversion of numerical forms



```
object_list = []
numerical_list = []

for col in (train_data.columns):
    if train_data[col].dtype == "object":
        object_list.append(col)
    else:
        numerical_list.append(col)
print("numerical columns = ", numerical_list)
print("object columns = ", object_list)
```

Fig.3 Conversion of numerical forms

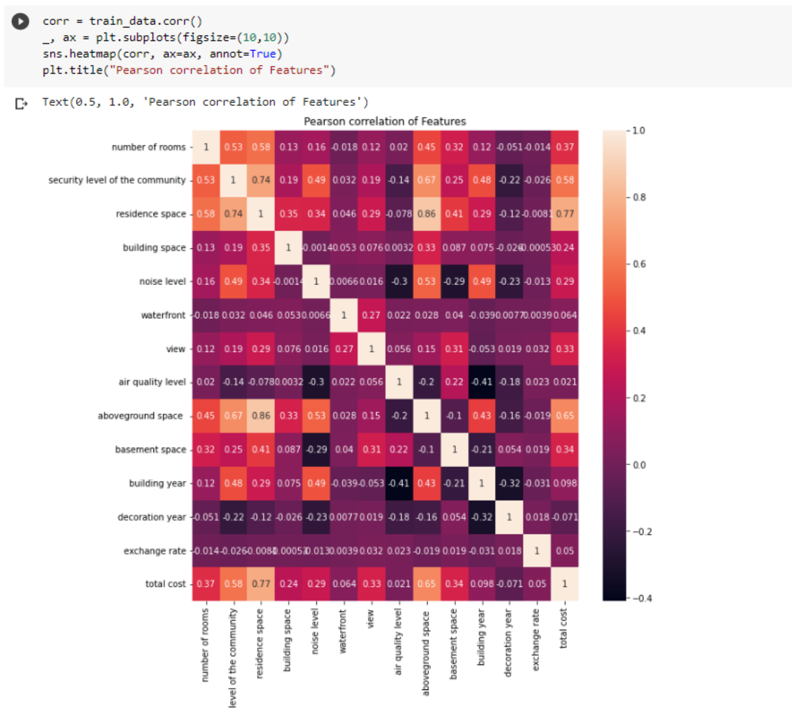
According to the meaning of the problem, the data is divided into four parts. The teacher's problem requires a division into 1 to 4, but since we use the Pytorch framework to complete the construction of the neural network and the subscripts of Pytorch are indexed from 0, here we divide the data into numeric data and category data so that we can treat the two types of data differently.

When we work with a dataset, there are usually different types of feature columns. These feature columns can be numeric or textual data, or other types of data such as dates, categories, etc. And when performing machine learning tasks, these feature columns need to be converted into a numerical form that the machine learning algorithm can understand.

In this code, we first create two empty lists, object_list and numerical_list, and then iterate through each column of the training data set train_data, adding it to object_list if the current column is of type "object", and adding it to numerical_list otherwise. otherwise it will be added to numerical_list. In this process, we use the dtype property provided by the pandas library to get the data type of each column.
Therefore, this code is one of the important steps to perform data preprocessing, and it helps us to easily classify different types of feature columns for subsequent processing.

3.3 Pearson correlation analysis between features

In this code, we calculate the Pearson correlation coefficient between features in the train_data dataset, which quantifies the linear relationship between features as a value ranging from -1 to 1, where 0 indicates no correlation between two features, a positive value indicates a positive correlation, and a negative value indicates a negative correlation.
Next, we use the Heatmap function of the Seaborn library to create a heat map that takes as input a matrix of correlation coefficients and indicates the magnitude of the correlation coefficients by the shade of color. By labeling the correlation coefficient values of each cell in the map with the annot parameter, we can more intuitively understand the correlation between each feature in the data.



```
corr = train_data.corr()
_, ax = plt.subplots(figsize=(10,10))
sns.heatmap(corr, ax=ax, annot=True)
plt.title("Pearson correlation of Features")
```

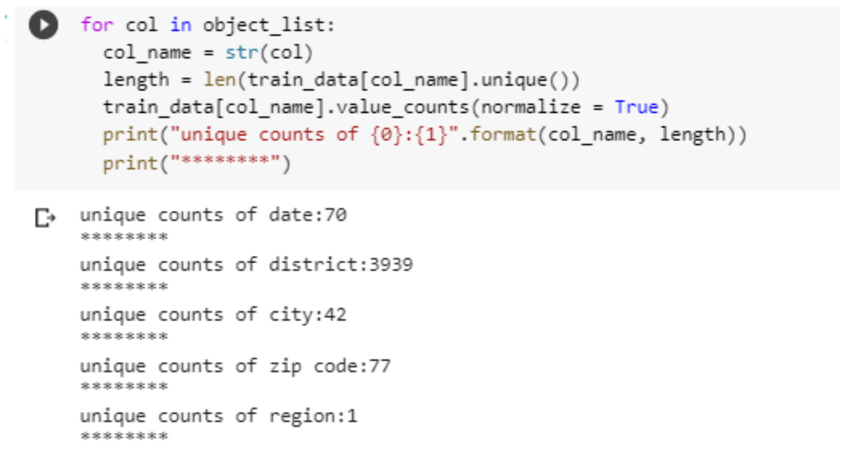
```
Text(0.5, 1.0, 'Pearson correlation of Features')
```

Fig.4 Feature Correlation Heat Map

3.4 Feature Engineering

The most important thing in feature engineering is feature encoding. Encoding can convert data into acomputer-recognizable format and put it in the model for training. One-hot is the most famousencoding, but one-hot encoding will increase thedimension of features,so we need to look at theunique value of the categorical data, which can tellus which features cannot be encoded with one-hot to avoid gradient explosion.

Next is feature conversion. Although some featuresare numerical features, their data 0, 1, 2, and 3represent a certain degree of comparison of thefeature, so we should convert it into a categoryfeature, and this feature The degree they representshould be preserved using label encoding.
In order to avoid gradient explosion, we encodethe district features with count encoding, and the remaining features with one- hot encoding.

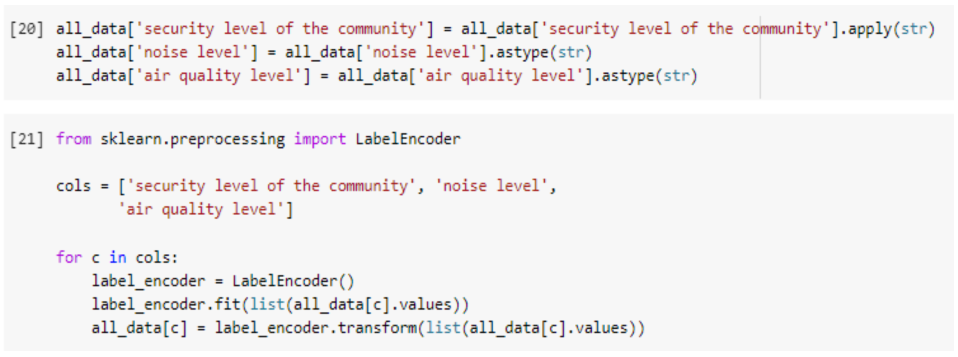


```
for col in object_list:
  col_name = str(col)
  length = len(train_data[col_name].unique())
  train_data[col_name].value_counts(normalize = True)
  print("unique counts of {0}:{1}".format(col_name, length))
  print("********")
```

```
unique counts of date:70
********
unique counts of district:3939
********
unique counts of city:42
********
unique counts of zip code:77
********
unique counts of region:1
********
```

Fig.5 Number of unique values for each classification feature

The results of this segment (Fig.5) show the number of unique values for each categorical feature in the dataset. The results of this segment show the number of unique values for each categorical feature in the dataset. Specifically, for the date classification feature, there are 70 different dates in the dataset; for the district classification feature, there are 3939 different community areas; for the city classification feature, there are 42 different cities; for the zip code classification feature, there are 77 different zip codes; and for the region classification feature, only one unique value appears which is the USA region.

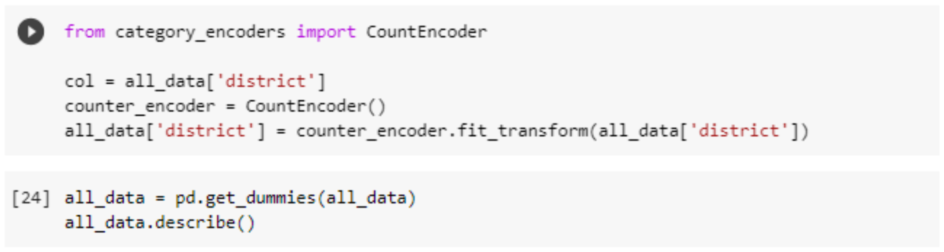


```
[20] all_data['security level of the community'] = all_data['security level of the community'].apply(str)
     all_data['noise level'] = all_data['noise level'].astype(str)
     all_data['air quality level'] = all_data['air quality level'].astype(str)
```

```
[21] from sklearn.preprocessing import LabelEncoder

     cols = ['security level of the community', 'noise level',
            'air quality level']

     for c in cols:
         label_encoder = LabelEncoder()
         label_encoder.fit(list(all_data[c].values))
         all_data[c] = label_encoder.transform(list(all_data[c].values))
```

Fig.6 Conversion of numeric types

Next, the code in Fig.6 converts the data types of the classification features to strings using the apply and astype functions, which is because the LabelEncoder needs to accept string types as input.
We also uses the LabelEncoder class in the Sklearn library to encode the string labels as integers for subsequent modeling and training.



```
from category_encoders import CountEncoder

col = all_data['district']
counter_encoder = CountEncoder()
all_data['district'] = counter_encoder.fit_transform(all_data['district'])
```

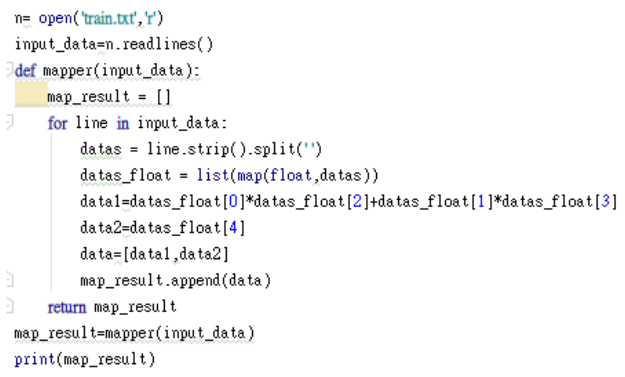
```
[24] all_data = pd.get_dummies(all_data)
     all_data.describe()
```

Fig.7 Encoding of classification features

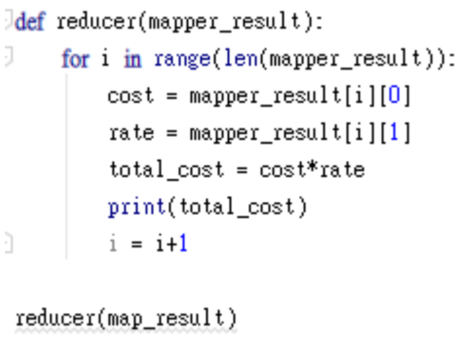
The code in Fig.7 uses the CountEncoder class from the category_encoders library to encode the district categorical features as counts, encoding each occurrence as an integer, which effectively reduces the effect of noise in the categorical features.
Finally, this code uses pd.get_dummies to convert all categorical features into 0/1 encoded dummy variables.

4 Model design and implementation

4.1 MapReduce



```
n= open('train.txt','r')
input_data=n.readlines()
def mapper(input_data):
    map_result = []
    for line in input_data:
        datas = line.strip().split('')
        datas_float = list(map(float,datas))
        data1=datas_float[0]*datas_float[2]+datas_float[1]*datas_float[3]
        data2=datas_float[4]
        data=[data1,data2]
        map_result.append(data)
    return map_result
map_result=mapper(input_data)
print(map_result)
```



```
def reducer(mapper_result):
    for i in range(len(mapper_result)):
        cost = mapper_result[i][0]
        rate = mapper_result[i][1]
        total_cost = cost*rate
        print(total_cost)
        i = i+1

reducer(map_result)
```

Fig.8 MapReduce

The first question we use MapReduce to calculate the total cost, the main idea is: decompose the training data set into 5 parts (price of residence space, residence space, unit price of building space, building space andexchange rate) for the training of Map function, these small tasks can be computed in parallel without any dependency between them. Then the results of the Map stage are globally aggregated using Reduce, i.e., using the formula ‘(The unit price of residence space * Residence space + Unit price of building space * Building space) * Exchange rate = Total cost’ is used to calculate the final cost result.
The MapReduce steps are as follows:
1 In the previous section, we extracted a few lines of data from the training data required for task 1, de-indexed them, and wrote them to train.txt. Use the readlines() method to read the contents of all lines in the file and store the results in the input_data list.
2 Mapper function:
The Mapper function accepts the file content input_data as an argument and processes each line of data, maps it to two values, cost and rate, and returns a list map_result containing cost and rate. the

specific process is as follows:

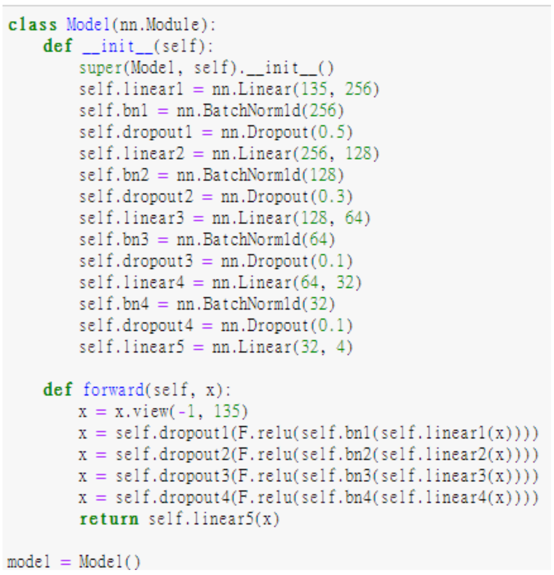
- First create an empty form.
- Cut each row of data by removing the first and last spaces and using " as a separator to generate a list datas consisting of string elements, and convert the string elements to floating point type to generate a list datas_float consisting of floating point elements.
- Calculate the cost cost of each row of data, Data1 is the sum of the product of unit area and price; data2 is the exchange rate rate, save them in the array data. Add the array data to the map_result list and return the map_result list after processing all the data.

3 The Mapper function is called and the result is saved in the map_result variable.

4 The Reducer function accepts the map_result output from the Mapper function as an argument and does the following:

- Use a for loop to iterate through each element of the map_result list in turn.
- For each element map_result[i], take out from it and calculate the product of cost and rate to get the total cost total_cost.
- Print out the total_cost.

4.2 CNN Module



```
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.linear1 = nn.Linear(135, 256)
        self.bn1 = nn.BatchNorm1d(256)
        self.dropout1 = nn.Dropout(0.5)
        self.linear2 = nn.Linear(256, 128)
        self.bn2 = nn.BatchNorm1d(128)
        self.dropout2 = nn.Dropout(0.3)
        self.linear3 = nn.Linear(128, 64)
        self.bn3 = nn.BatchNorm1d(64)
        self.dropout3 = nn.Dropout(0.1)
        self.linear4 = nn.Linear(64, 32)
        self.bn4 = nn.BatchNorm1d(32)
        self.dropout4 = nn.Dropout(0.1)
        self.linear5 = nn.Linear(32, 4)

    def forward(self, x):
        x = x.view(-1, 135)
        x = self.dropout1(F.relu(self.bn1(self.linear1(x))))
        x = self.dropout2(F.relu(self.bn2(self.linear2(x))))
        x = self.dropout3(F.relu(self.bn3(self.linear3(x))))
        x = self.dropout4(F.relu(self.bn4(self.linear4(x))))
        return self.linear5(x)

model = Model()
```

Fig.9 CNN

The model uses nn.Module as the base class and contains 5 linear layers, also called fully connected layers, each with Batch Normalization and Dropout operations. The model has an input data dimension of 135 and an output data dimension of 4.

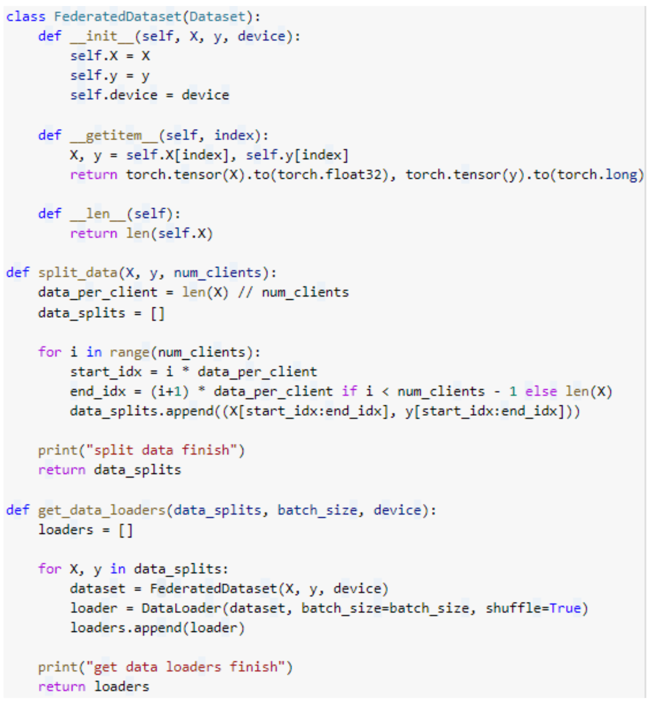
In the constructor, the model defines the structure of each layer, including the linear layer (nn.Linear), the batch normalization layer (nn.BatchNormld), and the random deactivation layer (nn.Dropout). Among them, the first linear layer maps the input dimensionality from 135 to 256, while the later layers gradually downscale and finally map the features to the output data dimensionality. We use the dropout layer to prevent overfitting.

In the forward propagation method forward(), the model first reshapes the input data into a two-dimensional tensor and performs forward computation through the respective layers.

Specifically, the model uses the F.relu() function as the activation function, and the output of the linear layer is processed through the batch normalization layer and the random deactivation layer, and fed again to the next layer.

5 Framework of Federated Learning

First we define a class named "FederatedDataset", which is a subclass of the Dataset class in PyTorch. This class is used to represent datasets in federated learning, where the data is scattered across multiple local data sources.
A function named split_ data() is defined to split the given input data X and the target variable y into a specified number of clients. This is a data partitioning method designed for joint learning.
Defines a function named get_data_ loaders() that creates a list of PyTorch data loaders, each of which is used to load data for one client of the joint learning.
A loop is used to iterate through all the batches of data in the train_ loader data loader. For each batch, the gradient of the optimizer is set to zero so that when calculating subsequent losses, the subsequent losses are not affected by the previous gradient. The data is then fed into the model for forward computation and the losses are computed. The losses are passed back to the model along the back propagation path and the parameters of the model are updated.
The relevant codes are as follows(Fig.10)：



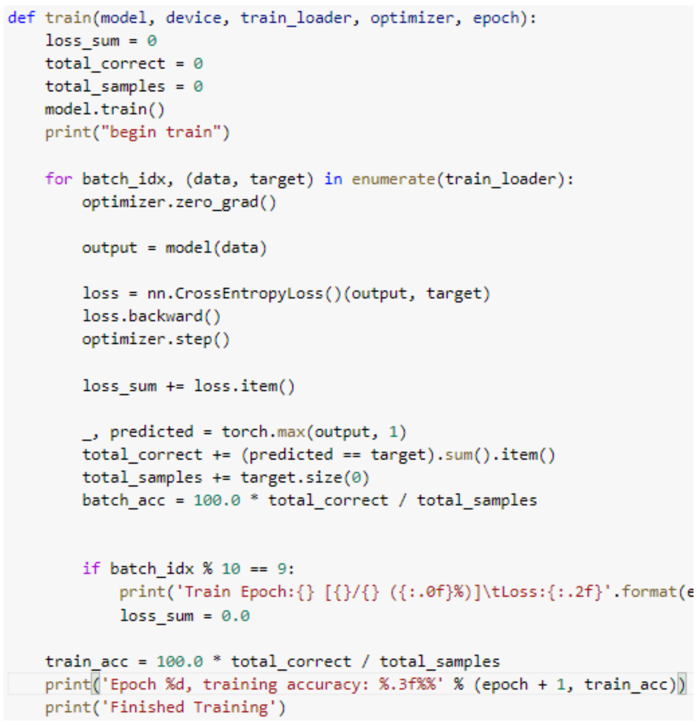
```python
class FederatedDataset(Dataset):
    def __init__(self, X, y, device):
        self.X = X
        self.y = y
        self.device = device

    def __getitem__(self, index):
        X, y = self.X[index], self.y[index]
        return torch.tensor(X).to(torch.float32), torch.tensor(y).to(torch.long)

    def __len__(self):
        return len(self.X)

def split_data(X, y, num_clients):
    data_per_client = len(X) // num_clients
    data_splits = []

    for i in range(num_clients):
        start_idx = i * data_per_client
        end_idx = (i+1) * data_per_client if i < num_clients - 1 else len(X)
        data_splits.append((X[start_idx:end_idx], y[start_idx:end_idx]))

    print("split data finish")
    return data_splits

def get_data_loaders(data_splits, batch_size, device):
    loaders = []

    for X, y in data_splits:
        dataset = FederatedDataset(X, y, device)
        loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
        loaders.append(loader)

    print("get data loaders finish")
    return loaders
```

Fig.10 FederatedDataset class creation

The function that builds the training neural network has parameters including the model, device, train_loader, optimizer, and epoch (Fig. 11).
At the beginning of the function, the model is first set to the training state, then the training dataset is traversed using enumerate, and the gradient of the optimizer is cleared to zero. The input data is then passed into the neural network model and the output is obtained. The loss is then calculated

using the cross-entropy loss function and the gradient is back-propagated to solve the gradient problem. Finally, the weights are updated using an optimizer.

During the training process, the loss and correctness of each batch are accumulated, and every 10 iterations the current round, the amount of data that has been trained, the average loss of the current batch, and the accuracy of the current batch are printed. When a round of training is completed, the current number of rounds and training accuracy are printed out and "End of Training" is output.

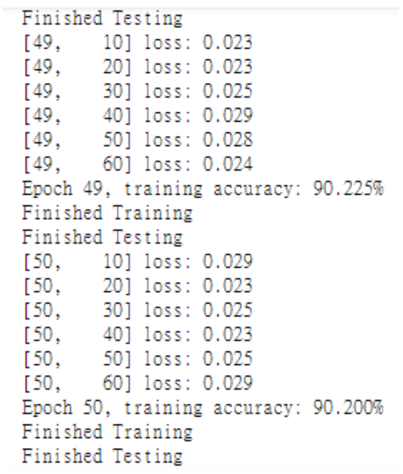


```
def train(model, device, train_loader, optimizer, epoch):
    loss_sum = 0
    total_correct = 0
    total_samples = 0
    model.train()
    print("begin train")

    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()

        output = model(data)

        loss = nn.CrossEntropyLoss()(output, target)
        loss.backward()
        optimizer.step()

        loss_sum += loss.item()

        _, predicted = torch.max(output, 1)
        total_correct += (predicted == target).sum().item()
        total_samples += target.size(0)
        batch_acc = 100.0 * total_correct / total_samples


        if batch_idx % 10 == 9:
            print('Train Epoch:{} [{}/{} ({:.0f}%)]\tLoss:{:.2f}'.format(e
            loss_sum = 0.0

    train_acc = 100.0 * total_correct / total_samples
    print('Epoch %d, training accuracy: %.3f%%' % (epoch + 1, train_acc))
    print('Finished Training')
```

Fig. 11 Training function of neural network

6 Performance Evaluation

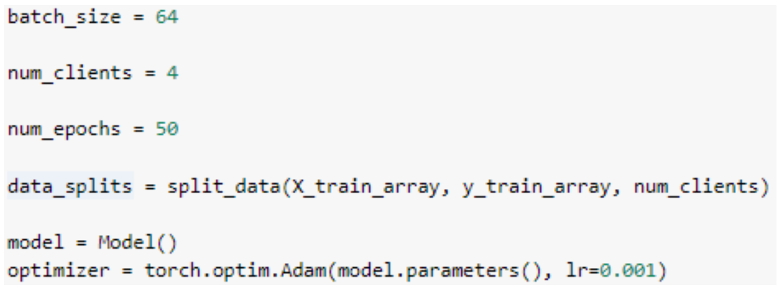
6.1 Task2 Model Accuracy



```
Finished Testing
[49,    10] loss: 0.023
[49,    20] loss: 0.023
[49,    30] loss: 0.025
[49,    40] loss: 0.029
[49,    50] loss: 0.028
[49,    60] loss: 0.024
Epoch 49, training accuracy: 90.225%
Finished Training
Finished Testing
[50,    10] loss: 0.029
[50,    20] loss: 0.023
[50,    30] loss: 0.025
[50,    40] loss: 0.023
[50,    50] loss: 0.025
[50,    60] loss: 0.029
Epoch 50, training accuracy: 90.200%
Finished Training
Finished Testing
```

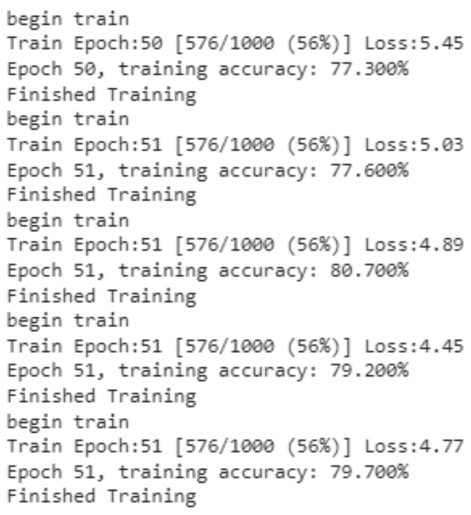
Fig.12 Loss and model accuracy for task 2

6.2 Task3 Model Accuracy



```
batch_size = 64

num_clients = 4

num_epochs = 50

data_splits = split_data(X_train_array, y_train_array, num_clients)

model = Model()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Fig.13 Hyperparameter code



```
begin train
Train Epoch:50 [576/1000 (56%)] Loss:5.45
Epoch 50, training accuracy: 77.300%
Finished Training
begin train
Train Epoch:51 [576/1000 (56%)] Loss:5.03
Epoch 51, training accuracy: 77.600%
Finished Training
begin train
Train Epoch:51 [576/1000 (56%)] Loss:4.89
Epoch 51, training accuracy: 80.700%
Finished Training
begin train
Train Epoch:51 [576/1000 (56%)] Loss:4.45
Epoch 51, training accuracy: 79.200%
Finished Training
begin train
Train Epoch:51 [576/1000 (56%)] Loss:4.77
Epoch 51, training accuracy: 79.700%
Finished Training
```

Fig.14 Loss and model accuracy for task 3

By comparing the above data(Fig.12 and Fig.14), we conclude that with the same hyperparameters, including learning rate, number of iterations, optimizer, and model framework, the accuracy of the convolutional neural network model training is about 90%, while the federal learning model is only about 80%.

7 Conclusion and Future Work

7.1 Conclusion

In federated learning, because the data is scattered across multiple devices, each device can only access its local data for training, so the training efficiency and accuracy of the entire model may be affected. In addition, because the data is trained locally, the distribution and quantity of data on each device may be different, which will also have a certain impact on the accuracy of federated learning.
Therefore, it is normal for federated learning to have lower accuracy than convolutional neural networks compared to training with the full dataset in a centralized environment.

7.2 Future Work

We can try to improve the accuracy of federated learning through some methods, such as increasing the number of communications between devices, changing the model architecture, increasing the number of training rounds, etc. At the same time, the architecture of the model can also be modified to try to get a model with better generalization performance.

8 Reference

1. Kairouz, P., McMahan, H. B., Avent, B., Bellet, A., Bennis, M., Bhagoji, A. N., … Zhang, L. (2019). Advances and Open Problems in Federated Learning.
arXiv:1912.04977. https://arxiv.org/abs/1912.04977

2. McMahan, H. B., Moore, E., Ramage, D., Hampson, S., & y Arcas, B. (2017). Communication-Efficient Learning of Deep Networks from Decentralized Data. In Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS 2017) (Vol. 54, pp. 1273–1282).
http://proceedings.mlr.press/v54/mcmahan17a/mcmahan17a.pdf

3. Li, T., Sahu, A. K., Talwalkar, A., & Smith, V. (2020). FedScale: Benchmarking Model Scaling for Federated Learning. arXiv:2007.09030.
https://arxiv.org/abs/2007.090