ML/DL projects REPORT

Benny Cai / Stella Zhang

11.04.2020

ECE196, Fall 2020

Project 1: Indian house price prediction (ML)

INTRODUCTION

This project aims to explore and analyze the data of a set of features/factors relevant to Indian house price, find the relationships between these features and the house price, and build a machine learning model that is able to predict the approximate house prices in India given that set of features.

We found this project idea interesting because, instead of simply classifying the target objects into groups, it expects the target predictions to be continuous. In other words, we shouldn't be doing classification on the target house prices; instead, we are continuously predicting the prices with their distinct values. This continuous nature makes the prediction task much more intriguing and challenging.

PROBLEM_SOLVING APPROACH

To solve this problem, first of all, we were going to do exploratory data analysis (EDA) to understand our data, where we were going to explore the relationships by visualizing the data using a bunch of graphing tools. Then we were going to do feature extraction and engineering based on the first step, to shape the original dataset into a new form with only the features that we've selected after the exploration and processing. Lastly, we were going to select and test out a few machine learning models to fit in our extracted data, compare and evaluate the performances and accuracy scores of different models, and make our predictions on the test data by using the model with the best performance.

SOLUTION:

Exploratory data analysis (EDA) and feature extraction:

At the first glance of our data, there are 11 features given, such as “SQUARE_FT”, “POST_BY”, “RESALE”, and etc. We dropped the column with the feature “ADDRESS” because the address is in text-form. Since a ML model couldn’t fit text-form data, this feature is no longer useful for building our model. Then, we started to visualize the data of each feature one by one against the house price to find their relationships.

An example of exploring and visualizing the data:

We counted the number of houses that are “under construction” and are “ready to move in” by using the “*value_counts()*” method.

```
print(train_data['UNDER_CONSTRUCTION'].value_counts())
```

```
0    24157
1     5294
Name: UNDER_CONSTRUCTION, dtype: int64
```

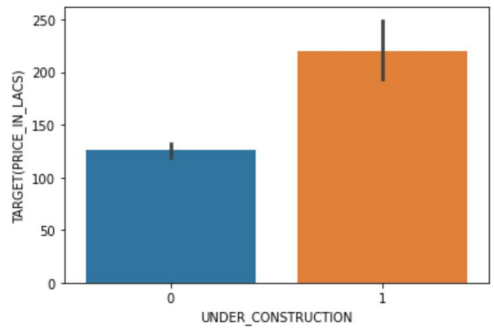
```
print(train_data['READY_TO_MOVE'].value_counts())
```

```
1    24157
0     5294
Name: READY_TO_MOVE, dtype: int64
```

A value of “1” means that a specific house has that specific feature. As expected, these two features are apparently opposite to each other, where in our dataset, there are 5294 houses that are under construction, and 24157 houses that are ready to move in. After finding their connections, we could now visualize them to find their relationships with the house price.

```
sns.barplot(data=train_data, x='UNDER_CONSTRUCTION', y='TARGET(PRICE_IN_LACS)')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8bef15de20>
```



By using the seaborn(sns) package, we visualized the data and found out that houses that are under construction are generally more expensive than ready-to-move-in houses. Therefore, we considered this relationship to be useful for building our model.

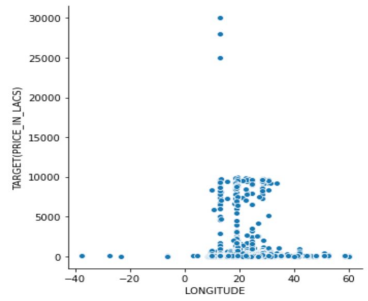
Apart from the “address” column that we dropped in the beginning, there are a few columns that we were unsure whether to drop them or not.

An example of dropping irrelevant data:

We hypothesized that the two columns of “LONGITUDE” and “LATITUDE” are potentially pretty irrelevant because normally people wouldn’t check the coordinates when buying houses.

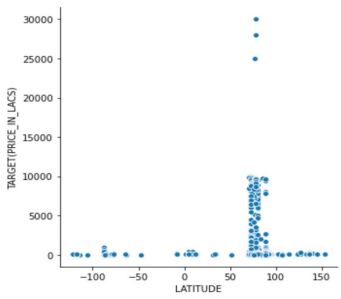
```
sns.relplot(data=train_data, x='LONGITUDE', y='TARGET(PRICE_IN_LACS)')
```

```
<seaborn.axisgrid.FacetGrid at 0x7fa3fb835f10>
```




```
sns.relplot(data=train_data, x='LATITUDE', y='TARGET(PRICE_IN_LACS)')
```

```
<seaborn.axisgrid.FacetGrid at 0x7fa3f83320a0>
```



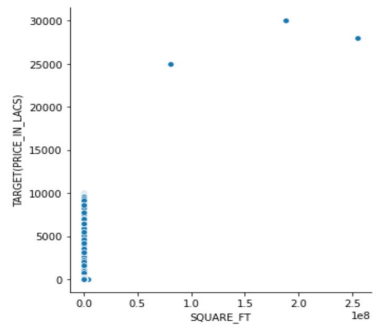
After visualizing them, we got two similar graphs where the relationship between x and y value seems to be really complex and incomprehensible. Therefore, we decided to drop these two columns in case that they affected the accuracy of our model.

An example of processing and tuning the data:

The most complicated feature in our data turned out to be “square feet” — the area of the house. This was expected because its data is of thousands of random large numbers instead of simply being 1 or 0 like other deterministic features such as “under construction or not”.

```
sns.relplot(data=train_data, x='SQUARE_FT', y='TARGET(PRICE_IN_LACS)')
```

```
<seaborn.axisgrid.FacetGrid at 0x7f8bf06126a0>
```



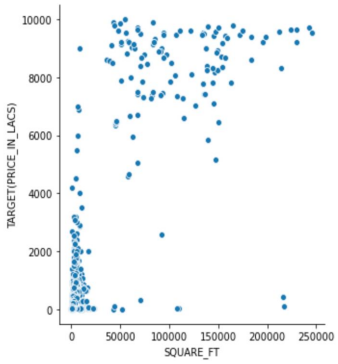
After visualizing it, we found out that there are a few outliers, or extreme cases, at the top right that we need to get rid of in order to display the general relationship. Hence, we wrote a simple for-loop to drop the houses with an area over 300000 square feet and visualized the data again:

```
for i in range(29451):
    if train_data['SQUARE_FT'][i] > 300000:
        print("i = " + str(i))
        print(train_data['SQUARE_FT'][i])
        train_data = train_data.drop([i])
```

```
i = 373
433333.3333
i = 10745
254545454.5
i = 11238
187500000.0
i = 15598
792857.1429
i = 15724
80645161.29
i = 22155
340000.0
i = 25241
4300000.0
i = 28591
875000.0
```

```
sns.relplot(data=train_data, x='SQUARE_FT', y='TARGET(PRICE_IN_LACS)')
```

```
<seaborn.axisgrid.FacetGrid at 0x7f8bf022d820>
```



Though it's not obvious, probably due to the large amount of data we had, we noticed that, from the top of the graph, a bunch of houses with relatively larger areas tend to be more expensive. Therefore, we decided to integrate this feature into our model.

Remark: In fact, we did try to build the model without removing these outliers, and it turned out that the models we built had a much lower accuracy score. When we built with LinearRegression, the model gave an accuracy of around 35%; DecisionTreeRegressor gave a higher accuracy of 74%, which was acceptable but not high enough in our opinion.

Further engineering of our train_data: one hot encoding and standard scalar

After analyzing all the features, we extracted out the relevant ones and created a new dataframe for them:

	POSTED_BY	UNDER_CONSTRUCTION	RERA	BHK_NO.	BHK_OR_RK	SQUARE_FT	READY_TO_MOVE	RESALE
0	Owner	0	0	2	BHK	1300.236407	1	1
1	Dealer	0	0	2	BHK	1275.000000	1	1
2	Owner	0	0	2	BHK	933.159722	1	1
3	Owner	0	1	2	BHK	929.921143	1	1
4	Dealer	1	0	2	BHK	999.009247	0	1

As mentioned earlier, we are not able to fit text-form data into the model, so we implemented one-hot encoding on the "POST_BY" and "BHK_OR_RK" features to turn them into deterministic numbers (i.e. 0 or 1) by using the "*pd.get_dummies*" method. (BHK means

bedroom, hall and kitchen; RK means room and kitchen; this feature indicates the type of property). Furthermore, we noticed that the data in the “SQUARE_FT” column is way much bigger than the other columns, so we standardized it to make the computation easier by applying the “*StandardScaler()*” object and the “*sc.fit_transform*” method.

	UNDER_CONSTRUCTION	RERA	BHK_NO.	SQUARE_FT	READY_TO_MOVE	RESALE	BHK	RK	Builder	Dealer	Owner
0	0	0	2	-0.065022	1	1	1	0	0	0	1
1	0	0	2	-0.068143	1	1	1	0	0	1	0
2	0	0	2	-0.110422	1	1	1	0	0	0	1
3	0	1	2	-0.110822	1	1	1	0	0	0	1
4	1	0	2	-0.102278	0	1	1	0	0	1	0

This is what our final train_data looks like.

Model selection and performance:

As mentioned in the introduction, our model should be able to predict continuous results, so we were using regression models (Regressor) instead of classification models (Classifier). We selected and tested out three different regression models:

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor()
y_train = np.ravel(y_train)
rf.fit(X_train, y_train)
rf.score(X_test, y_test)
```

```
0.8992559312129621
```

```
from sklearn.linear_model import LinearRegression

lr = LinearRegression()
lr.fit(X_train, y_train)
lr.score(X_test, y_test)
```

```
0.7532827883677469
```

```
from sklearn.tree import DecisionTreeRegressor

dc = DecisionTreeRegressor()
dc.fit(X_train, y_train)
dc.score(X_test, y_test)
```

```
0.8746622076735466
```

Apparently, after processing our raw data, the performance of our models had been significantly improved. The best model out of these was the RandomForestRegressor, which gives an accuracy score of nearly 90% on my testing data.

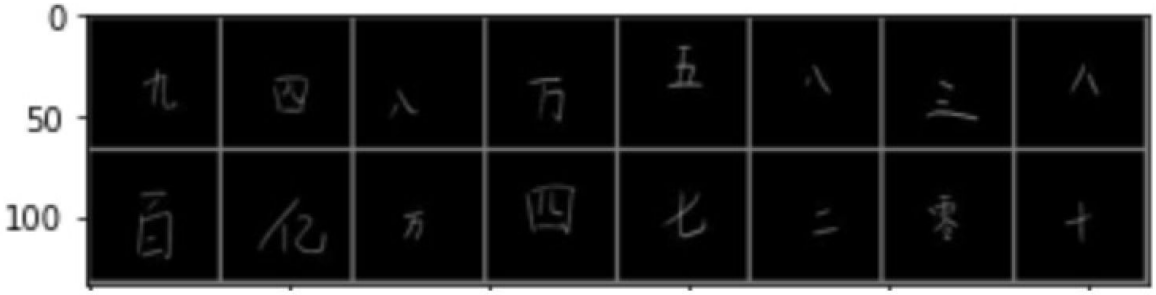
SUMMARY

In summary, by implementing data preprocessing such as EDA, we have significantly improved the relevance between the extracted data and the target prediction. Our model achieves a really strong performance with its 90% accuracy, considering it's a regression problem instead of a classification problem. One way to further improve the performance of our model, I believe, could be applying hyperparameters tuning, where choosing some optimal hyperparameters could have better control over the model's learning process and hence its overall behaviours.

Project 2: Chinese MNIST Classification (ML/DL)

INTRODUCTION

This project idea was originated from the classic MNIST example we saw in class, where we've gone through a few ML/DL models that are able to recognize the images of handwritten digits from 0 to 9. In our project, we were going to build both ML and DL models that are able to recognize the images of fifteen Chinese number characters, from 0 to 9 plus the characters for ten, hundred, thousand, 10 thousand and 100 million, and accurately predict the numerical values of these images. In the meantime, we were also going to compare and reflect on the performance of the ML and DL models that we built. We found this idea interesting because Chinese characters have much more complicated structures than number digits, so we would like to challenge ourselves to solve this problem for Chinese people. Below are a few examples of the characters' images that we are dealing with.



Machine learning model:

PROBLEM_SOLVING APPROACH

First of all, unlike the built-in classic MNIST dataset, where images have already been converted to 28*28 pixels data and flattened to a 784-dimension array, we need to find a way to manually read in the 64*64 pixels of our images and flatten them into a 4096-dimension array in order to process the data of our images. Then, we were going to use PCA to reduce the dimension of our array, because 4096 is quite a large number which may slow down the computation. Lastly, we were going to apply the KNN classifier on our dataset.

SOLUTION:

Data extraction and processing:

As we loaded in the original dataset, we had the dataframe on the left. Take the first row as an example, “suite_id”, “sample_id”, and “code” columns basically form the file names of our images, as we can see from the right image, where its file name is constructed as “1_1_10”. We can think of this sequence of three numbers as a unique ID for each image. The “value” column indicates the numerical value of this character.

	suite_id	sample_id	code	value	character
0	1	1	10	9	九
1	1	10	10	9	九
2	1	2	10	9	九
3	1	3	10	9	九
4	1	4	10	9	九

One way to read in our images is to apply the “*plt.imread()*” method on the image name. In order to apply this method, we had to manually construct the unique ID for each image.

```
def create_file_name(x):

    file_name = f"input_{x[0]}_{x[1]}_{x[2]}.jpg"
    return file_name
```

```
train_data["file_name"] = train_data.apply(create_file_name, axis=1)
```

Above is a helper function that we created to construct the file name for each image. After applying this helper function to our train_data, we had a new column named as "file_name" that specifies the unique ID for each image.

	suite_id	sample_id	code	value	character	file_name
0	1	1	10	9	九	input_1_1_10.jpg
1	1	10	10	9	九	input_1_10_10.jpg
2	1	2	10	9	九	input_1_2_10.jpg
3	1	3	10	9	九	input_1_3_10.jpg
4	1	4	10	9	九	input_1_4_10.jpg

Flatten the 64*64 pixels:

Now we could apply the "*plt.imread()*" to our images, the next step is to flatten these 64*64 arrays into a 4096-dimensional array. To achieve this, we could simply apply the "*flatten()*" method on each 64*64 array.

Below is a simple for-loop we wrote that creates a new numpy array of shape (15000, 4096) —— we had 15000 images and each image has dimension of 64*64 = 4096.

```
numpy_data = np.zeros(shape=(15000,4096))
for i in range(15000):
    string = "data/" + train_data["file_name"][i]
    image = plt.imread(string).flatten()
    numpy_data[i] = image
```

```
numpy_data.shape
```

```
(15000, 4096)
```

Now we could use this new numpy array to construct a new panda dataframe, which is the final dataframe we were going to use for building the model. We can see that all the images have been flattened into a 4096-dimensional array.

```
new_train = pd.DataFrame(data=numpy_data, index=[i for i in range(1, 15001)],
                         columns=[i for i in range(1, 4097)])
new_train.head()
```

	1	2	3	4	5	6	7	8	9	10	...	4087	4088	4089	4090	4091	4092	4093	4094	4095	4096
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Dimensionality reduction and model application:

As mentioned earlier, a dimension of 4096 is way too large and so our computational speed might be badly affected. Therefore, we've decided to reduce our dimension by applying a PCA object on our dataset.

Below is the code snippet. We firstly initialized a PCA object with n_componenets=15, and then we fitted and transformed our original train data with the PCA object.

```
from sklearn.decomposition import PCA

pca = PCA(n_components=15)
pca.fit(X_train)
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)
```

Finally, we applied our newly-shaped train and test data to a KNN classifier, and obtained an accuracy of 73.7%.

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier()  # Intialize KNeighborsClassifier with a value of k <= 7
knn.fit(X_train_pca, y_train)   # Fit knn classifier to X_train_pca
y_pred = knn.predict(X_test_pca)   # Generate predictions based on X_test_pca
sum(y_pred == y_test) / len(y_pred)  # Outputs the accuracy score for your model
```

```
0.737
```

Remark: We noticed that the n_components attribute of the PCA object could also make a huge difference to the performance of our model. Initially, we specified this attribute to be 30, and the KNN model only output an accuracy score of around 64%. After several modifications to this attribute, we found that 15 is the best choice.

Deep learning model:

PROBLEM_SOLVING APPROACH

Clearly, an accuracy score of 73.7% doesn't seem to be good. Therefore, we were going to apply DL concepts to see if we can improve the accuracy. In this project, we were going to build DL models using two different approaches with different imported packages —— one with tensorflow.keras, the other with pytorch, and compare their performance.

For both approaches, we were going to construct and implement a neural network, and apply the network to train our data to build the model. Ideally, the model we built with pytorch should perform better because we've applied a stronger neural network called "convolutional neural network (CNN)" algorithm to that model, which technically has a more complex and well-formed network structure.

SOLUTION:

MLP network with tensorflow.keras:

Data processing:

We basically did the same processing as we did for the ML model, where we extracted the file name of each image, and created a new numpy array to store the pixels of the 15000 images. But this time we left the image to be a 64*64 array without flattening it.

One minor change we did was that we changed all the big values (hundred, thousand, 10 thousand, 100 million) into continuous values of 11 ~ 14. This is because the last layer of our network would have 15 neurons/classes, and making these 15 classes to be continuous, instead of jumping from 10 to 100, would make the training process easier.

```
# Change the big numbers into continuous ones
for i in range(15000):
    if train_data['value'][i] == 100:
        train_data['value'][i] = 11                # 100 is 11
    elif train_data['value'][i] == 1000:
        train_data['value'][i] = 12                # 1000 is 12
    elif train_data['value'][i] == 10000:
        train_data['value'][i] = 13                # 10000 is 13
    elif train_data['value'][i] == 100000000:
        train_data['value'][i] = 14                # 100000000 is 14
```

Then we normalized our input values and resized them for use in the MLP network.

```
# Normalize input values
X_train = X_train/255.
X_test = X_test/255.

# Resize images for use in MLP
mlp_x_train = X_train.reshape((X_train.shape[0], 4096))
mlp_x_test = X_test.reshape((X_test.shape[0], 4096))
```

Implement MLP network and training:

There are basically four layers in our MLP network. The first layer had an input dimension of 4096 because that is the dimension of our input images. We made the dimension of the two subsequent dense layers to be 2048 because we considered this to be a reasonable dimensionality reduction, being right in the middle of 4096 where it's neither too large nor too small. The last classification dense layer would have a dimension of 15 because, as explained earlier, we have fifteen Chinese number characters to learn and hence we would have 15 neurons/classes at the end.

```
# Implements the network using TensorFlow.Keras
def MLPNetwork(inputDim):
    # input layer (note that batches are already taken care of for you)
    x = keras.Input(shape=(inputDim,))

    # subsequent dense layers
    a1 = keras.layers.Dense(2048, activation='relu')(x)
    a2 = keras.layers.Dense(2048, activation='relu')(a1)

    # classification dense layer
    logits = keras.layers.Dense(15, activation='softmax')(a2)

    # Creates the model given the above structure
    model = keras.Model(inputs=x, outputs=logits)
    return model

MLPmod = MLPNetwork(4096)
MLPmod.summary()
```

```
# Sets up optimizer
opt = keras.optimizers.Adam(learning_rate=0.001)
MLPmod.compile(optimizer=opt, loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# The callback forces the network to train until it reaches 100% accuracy
hist = MLPmod.fit(mlp_x_train, y_train, epochs=10, shuffle=True)
```

```
Epoch 1/10
375/375 [==============================] - 38s 101ms/step - loss: 1.1273 - accuracy: 0.6320
Epoch 2/10
375/375 [==============================] - 39s 105ms/step - loss: 0.3194 - accuracy: 0.8898
Epoch 3/10
375/375 [==============================] - 44s 117ms/step - loss: 0.1471 - accuracy: 0.9498
Epoch 4/10
375/375 [==============================] - 40s 108ms/step - loss: 0.0798 - accuracy: 0.9728
Epoch 5/10
375/375 [==============================] - 41s 110ms/step - loss: 0.0602 - accuracy: 0.9809
Epoch 6/10
375/375 [==============================] - 43s 116ms/step - loss: 0.1056 - accuracy: 0.9647
Epoch 7/10
375/375 [==============================] - 43s 115ms/step - loss: 0.0678 - accuracy: 0.9790
Epoch 8/10
375/375 [==============================] - 36s 95ms/step - loss: 0.0569 - accuracy: 0.9838
Epoch 9/10
375/375 [==============================] - 37s 99ms/step - loss: 0.0364 - accuracy: 0.9892
Epoch 10/10
375/375 [==============================] - 42s 111ms/step - loss: 0.0296 - accuracy: 0.9924
```

```
# Evaluate the trained network on the testing dataset
testLoss, testAcc = MLPmod.evaluate(mlp_x_test, y_test)
print("Test accuracy for this model is {}".format(testAcc))
```

```
94/94 [==============================] - 2s 20ms/step - loss: 0.5853 - accuracy: 0.8750
Test accuracy for this model is 0.875
```

Finally, we started to train our data. We could see that the accuracy of our model has been constantly improving as we moved along the epochs. After evaluating the network on the testing dataset, we obtained an accuracy score of 87.5%, which was an huge improvement from the previous 73.7% accuracy score of the ML model. So now, the question is, can we do even better?

Convolutional neural network with pytorch:

Creating a customized class of my own Dataset:

The data processing for this approach is basically the same as the last approach with tensorflow, so we don't need to go through that again.

In order to apply the pytorch Dataloader, we created a customized class called "ChineseMnistDatatset" that inherits the built-in Dataset class from pytorch. We initialized this class by passing in the csv_file (in our case, an updated datafile with the "file_name" column), the root directory of those images, and a transform to be applied on my data. Then we override the inherited "*len*" and "*getitem*" methods, where "*len*" returns the length of our dataset, and "*getitem*" returns a tuple of a 64*64 image's pixels and the image's corresponding value. We did so because we were going to apply an iterator on our DataLoader in the training procedure, where the iterator should be able to extract out these two elements, images and their values, separately.

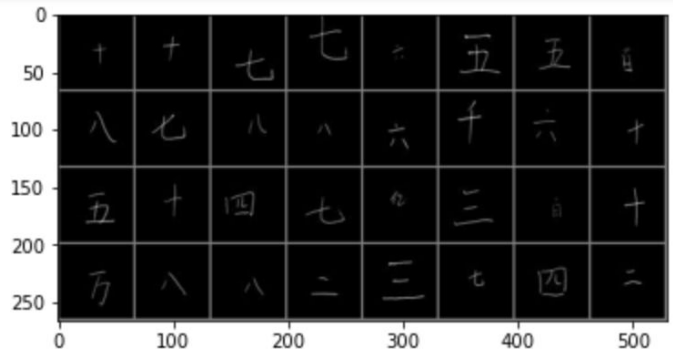
```
class ChineseMnistDataset(torch.utils.data.Dataset):
    """Dataset wrapping images and target labels for Chinese_mnist

    Arguments:
        A CSV file path
        Path to image folder
        Optional transform to be applied on a sample.
    """
    def __init__(self, csv_file, root_dir, transform=None):

        self.label_frame = pd.read_csv(csv_file)
        self.root_dir = root_dir
        self.transform = transform

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

    def __getitem__(self, index):
        if torch.is_tensor(index):
            index = index.tolist()

        img_name = os.path.join(self.root_dir, self.label_frame['file_name'].loc[index])
        image = io.imread(img_name)
        label = self.label_frame['value'].loc[index]

        sample = (image, label)
        if self.transform:
            sample = (self.transform(image), torch.tensor(label))

        return sample
```

After creating our customized class, we could now apply the pytorch Dataloader to load in our data for training, and we named it “trainloader”.

```
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5, ), (0.5, ))])
trainset = ChineseMnistDataset(csv_file='new_train_file.csv',
                               root_dir='../input/chinese-mnist/data/data/', transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=32, shuffle=True, num_workers=2)
```

Then, we did a testing round to test out if the iterator on our trainloader could work properly. We applied an iterator to our trainloader, and called the “*imshow*” method, which is a helper function we defined by ourselves to display an image, on the image that the iterator is pointing to. Here we could see that 32 images with their corresponding values have been shown together, because we specified our batch size to be 32. Clearly, the iterator worked really well.

```
# The function to show an image.
def imshow(img):
    img = img / 2 + 0.5     # Unnormalize.
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

# Get some random training images.
dataiter = iter(trainloader)
for i in range(2):
    images, labels = next(dataiter)

    # Show images.
    imshow(torchvision.utils.make_grid(images))
    # Print labels.
    print(labels)
```




```
tensor([10, 10,  7,  7,  6,  5,  5, 11,  8,  7,  8,  8,  6, 12,  6, 10,  5, 10,
         4,  7, 14,  3, 11, 10, 13,  8,  8,  2,  3,  7,  4,  2])
```

Implement CNN network and training:

There are basically four layers in our CNN algorithm —– two convolutional layers and two linear layers. For the first convolutional layer, the input channel is 1 and the output channel is 64 because we've checked that the size of one of our images is (1, 64, 64). We also specified the kernel size to be 3*3 and the padding to be 1 for both convolutional layers. For the second linear layer, we specified the out features to be 15 because we expected the output to be placed into 15 classes, as previously explained.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 64, 3, padding=1)
        self.conv2 = nn.Conv2d(64, 20, 3, padding=1)
        self.fc1 = nn.Linear(5120, 100)
        self.fc2 = nn.Linear(100, 15)

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), 2)
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()     # Create the network instance.
net.to(device)  # Move the network parameters to the specified device.
```

```
Net(
  (conv1): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(64, 20, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=5120, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=15, bias=True)
)
```

After defining the loss function and optimizer, we could now start our training.

```
# We use cross-entropy as loss function.
loss_func = nn.CrossEntropyLoss()
# We use stochastic gradient descent (SGD) as optimizer.
opt = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)
```

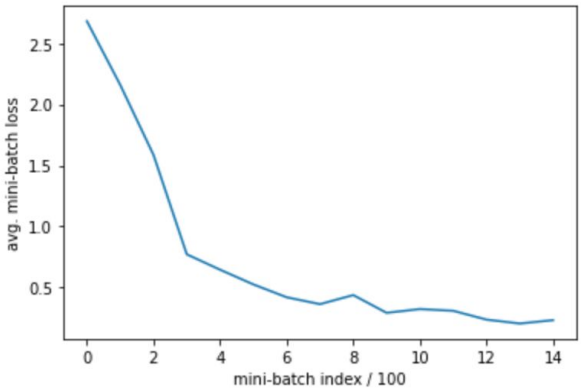
In our training procedure, we specify the epochs to be 5 and the print frequency to be 100. As mentioned earlier, we used the "*enumerate(trainloader, 0)*" method as an iterator to iterate through our dataset. Inside the for-loop, we firstly extracted out the input image data and its corresponding label, and then we applied the CNN network for forward propagation, followed by the backward propagation where we applied the loss function to tweak the weights of our model.

```
avg_losses = []   # Avg. losses.
epochs = 5        # Total epochs.
print_freq = 100  # Print frequency.

for epoch in range(epochs):  # Loop over the dataset multiple times.

    running_loss = 0.0       # Initialize running loss.
    for i, data in enumerate(trainloader, 0):
        # Get the inputs.
        inputs, labels = data

        # Move the inputs to the specified device.
        inputs, labels = inputs.to(device), labels.to(device)

        # Zero the parameter gradients.
        opt.zero_grad()

        # Forward step.
        outputs = net(inputs)
        loss = loss_func(outputs, labels)

        # Backward step.
        loss.backward()

        # Optimization step (update the parameters).
        opt.step()

        # Print statistics.
        running_loss += loss.item()
        if i % print_freq == print_freq - 1: # Print every several mini-batches.
            avg_loss = running_loss / print_freq
            print('[epoch: {}, i: {:5d}] avg mini-batch loss: {:.3f}'.format(
                epoch, i, avg_loss))
            avg_losses.append(avg_loss)
            running_loss = 0.0
print('Finished Training.')
```

```
[epoch: 0, i:    99] avg mini-batch loss: 2.682
[epoch: 0, i:   199] avg mini-batch loss: 2.160
[epoch: 0, i:   299] avg mini-batch loss: 1.586
[epoch: 1, i:    99] avg mini-batch loss: 0.771
[epoch: 1, i:   199] avg mini-batch loss: 0.644
[epoch: 1, i:   299] avg mini-batch loss: 0.524
[epoch: 2, i:    99] avg mini-batch loss: 0.419
[epoch: 2, i:   199] avg mini-batch loss: 0.362
[epoch: 2, i:   299] avg mini-batch loss: 0.437
[epoch: 3, i:    99] avg mini-batch loss: 0.291
[epoch: 3, i:   199] avg mini-batch loss: 0.322
[epoch: 3, i:   299] avg mini-batch loss: 0.308
[epoch: 4, i:    99] avg mini-batch loss: 0.235
[epoch: 4, i:   199] avg mini-batch loss: 0.204
[epoch: 4, i:   299] avg mini-batch loss: 0.231
Finished Training.
```

After the training, we plotted the training loss curve as shown below, which displays a nice downhill curve indicating the fact that the loss/error of our model had been constantly reducing.

```
plt.plot(avg_losses)
plt.xlabel('mini-batch index / {}'.format(print_freq))
plt.ylabel('avg. mini-batch loss')
plt.show()
```



The last step is evaluating the accuracy of our model. As shown below, we obtained an accuracy of 90%! This is so far the highest record among the three models we had built.

```
# Get test accuracy.
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 90 %
```

SUMMARY

In summary, the two deep learning models turned out to perform much better than the machine learning model, where the DL models gave an accuracy score of 87.5% and 90% respectively whereas the ML model only gave 73.7%. Between the two DL models, the model that implemented the convolutional neural network had a slightly higher accuracy. The difference might not be that significant, but it does confirm our hypothesis that a CNN algorithm performs better than a normal neural network.

Although an accuracy of 90% seems to be a really nice result, considering the complexity of Chinese number characters, there might still be some other ways to improve the performance. For example, we didn't try implementing a LeNet network using tensorflow.keras that could also involve a CNN algorithm structure. There are certainly other networks that we could try implementing that may potentially further improve the performance as well as the accuracy score.