Application of AI/ML techniques in Atmospheric Science

Summer School, Lahti, 2025

August 2025

Artificial Intelligence Tutorial 1. Handwritten Digit Recognition and Image Classification via a Convolutional Neural Network (CNN)

Module Leader: Zhi-Song Liu

Module Contributor: Zhi-Song Liu

Other Contributors: Ella Litjens, Wenqing Peng

Objectives

The objectives of this laboratory include

(i) to familiarize ourselves with the use of Python Platform for handwritten digit recognition with deep learning (via LeNet, a Convolution Neural Network (CNN))

(ii) to learn input formats and their data structures

(iii) to train a CNN with typical hyper parameters

(iv) to explore the effects of different training hyper parameters

(v) to learn using a trained CNN to acquire results

(vi) to display and store output with the right formats

(vii) to learn re-training using AlexNet

Time: 1hr

Equipment:

Use Google Colab – we will continue to work in the same document as previously. The link can be found here: https://colab.research.google.com/drive/1ECML58h39amzLICZJVoVDJkMmzDBrwY8?usp=sharing

You will also need to install the following modules, but we will take you through this.

python (Version 3.5 or above)
pytorch (Version 1.3.0 or above)

pandas (Version 1.0.3 or above)
matplotlib (Version 3.2.1 or above)

In your working directory, you should have the following files, that you downloaded from the GitHub repository in the previous tutorial:

Files	Description
mnist_train.csv	MNIST training dataset
mnist_test.csv	MNIST testing dataset
data.py	Data preprocessing
data_a.py	Data preprocessing
data_b.py	Data preprocessing
data_c.py	Data preprocessing
data_d.py	Data preprocessing
data_e.py	Data preprocessing
data_f.py	Data preprocessing
network.py	LeNet structure
main.py	Main function for training
eval.py	Main function for evaluation
eval_b.py	Main function for evaluation
eval_c.py	Main function for evaluation
draw_curve.py	Training and testing visualization
/model	Folder of trained parameters
/image	Folder of testing images

You will also need, the following files for the training section of the tutorial:

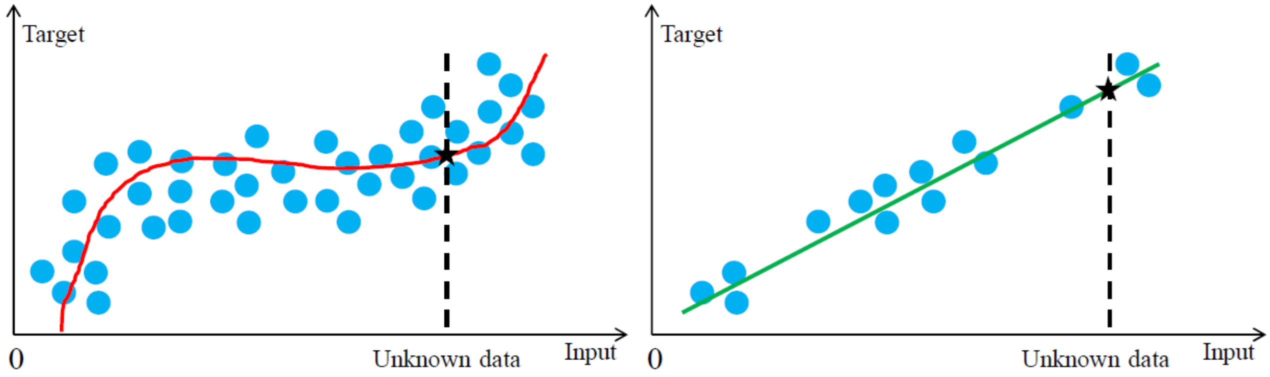
Files	Description
alexnet_main.py	Training of AlexNet
alexnet_eval.py	Testing of the pre-trained AlexNet
lenet_main.py	Training of LeNet
train.py	Define training strategy
lenet.py	Define LeNet5
initialize.py	Initialization of network parameters
data.py	Data preparation
alexnet.py	Define AlexNet
plot_multi_curves.py	Plot curves with different hyper-parameters
class_names_ImageNet.txt	1000 classes names of ImageNet
./datasets/MNIST	Contain train and test sets of MNIST (10 classes hand written digits classification)
./datasets/Caltech15	Contain train and test sets of Caltech15 (15 classes object classification)
./alexnet_images	Testing images for AlexNet

1. Understanding handwritten digit recognition

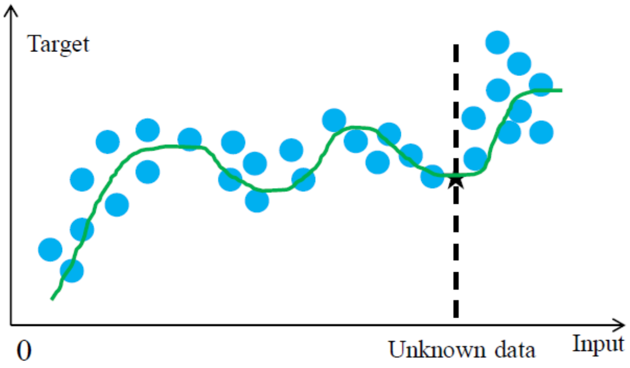
Object classification is a basic computing task. It is a fundamental step for Artificial Intelligence, that is, given an input image to the computer, it can understand the physical meaning of the images. To achieve the task, machine learning is a modern tool used. Conventional machine learning approaches work on small datasets with mediocre performances. Convolutional Neural Network (CNN), on the other hand, has the ability to learn complex feature representation from big datasets. In general, CNN outperforms conventional machine learning approaches. In order to process a huge amounts of data, Graphic Processing Units (GPUs) are usually required for CNN training and testing.

1.1 Principles of Convolutional Neural Networks:

Before introducing Convolutional Neural Network, let’s first understand the motivation of machine learning. Assume that we have a group of observed data X and their corresponding target Y as shown in Figure 1(a), we plot the correlation between X and Y as a 2D diagram. Each blue dot represents one **observation and its target.**



(a) Ground truth model (b) Underfitting model



(c) Overfitting model

Figure 1. Model estimation

Using machine learning approaches, our target is to model all observed data to find the ground truth red curve that can describe the data distribution. Hence, when we have some unknown input, we can recall the target result (classification say for example) using the red curve. In order to find the correct model, there are two issues: 1) gather enough data to represent the real distribution and 2), proper learning approaches for modeling. Figure 1(b) shows the situation when we lack of data for modeling. In this case, we only can obtain a linear model that oversimplifies the data distribution. Figure 1(c) shows another case when we have enough data but choose an over-complex model for modeling. It biases to some outliers that does not follow the general distribution of data. CNN has the advantages of having less of such problems. With the help of GPUs, it can learn the nonlinear relationship from a huge dataset to fit the real distribution.

1.2 Handwritten Digit Recognition:

In this laboratory, one of our goals is to make use of a trained CNN model to recognize a handwritten digit. For the sake of convenience, we make use of a well-known dataset, MNIST. The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training and testing of various image processing systems, including the training and testing machine learning networks. It is a subset from NIST and the digits have been centered in a 28x28 image. It was created by "re-mixing" the samples from NIST's original datasets. A few examples are shown in Figure 2.

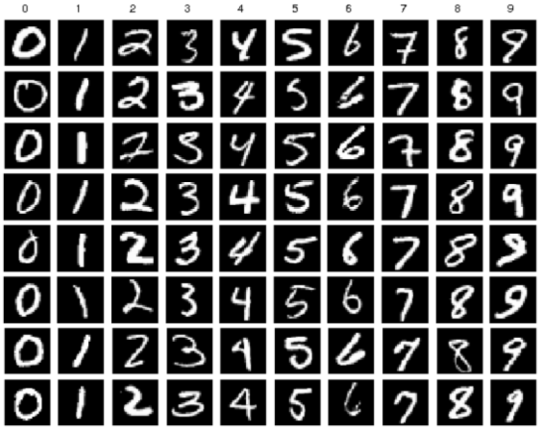


Figure 2. Examples of MNIST digits

The dataset has a training set of 60,000 examples, and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed size grayscale image of dimension 28×28. You can download the dataset from http://yann.lecun.com/exdb/mnist/.

The original dataset is stored in MSBformat, which is a binary format and is not convenient for processing. In our lab, we provide two csv files of MNIST (mnist_train.csv and mnist_test.csv) for easy processing. These can be found at https://pjreddie.com/projects/mnist-in-csv/, and are converted into matrix format. We will introduce it later in the lab.

In the next laboratory exercise, we will learn Training when a set of handwritten images with labels is given. The objective is to train a CNN model to recognize the number (the label). The challenge is that people write numbers very differently. We need the CNN model to be robust enough to recognize the actual number from various handwritten styles. Hence, in order to achieve it, we need a dataset that contains both images and ground truth labels to tell the CNN what to learn. The complete process includes training and testing phases as shown in Figure 3.

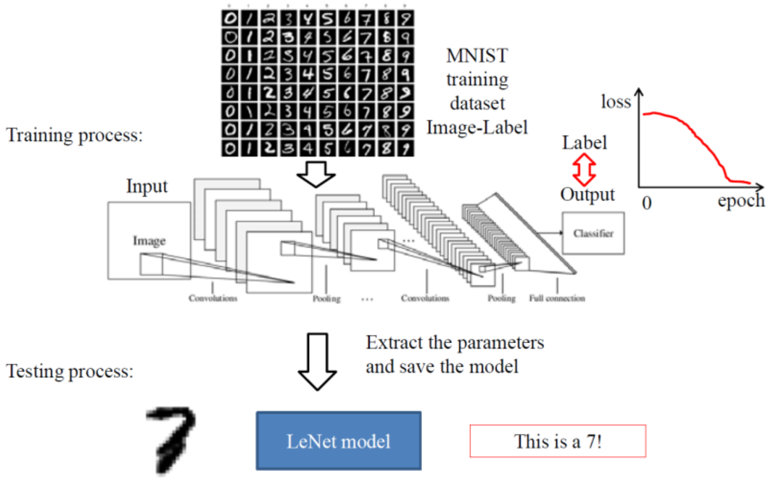
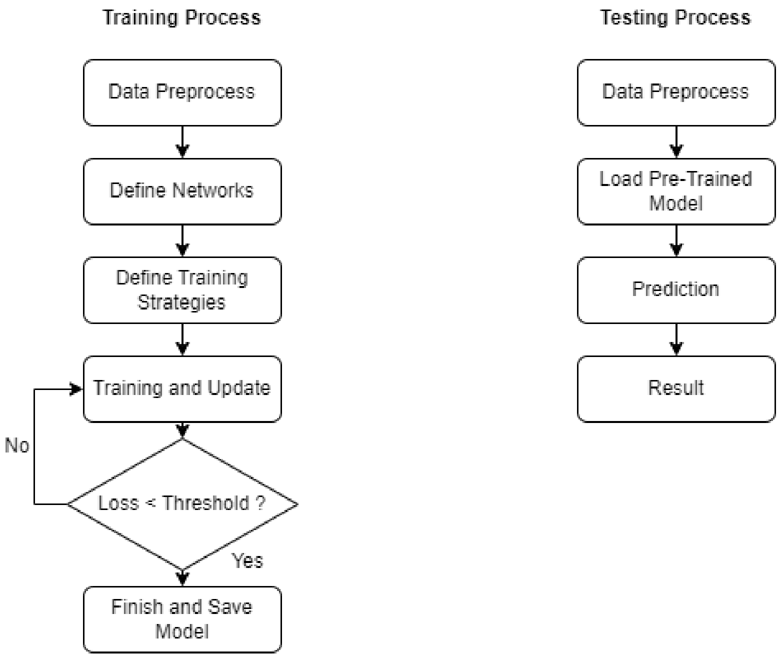


Figure 3. Training and testing processes of MNIST recognition

In the training phase, we need to prepare the training data from MNIST dataset to pair the ground truth labels and input images. We then design the network (in our case, we use the LeNet5 model). Next, we set up the training parameters and start to train the model. The target is to minimize the loss between ground truth label and actual output the model. When the loss is minimized to a threshold, we extract the network parameters and save the model. Note that the definition of epoch is one complete loop of using all the training data.

During the testing phase, we can load a trained model to test real images for the prediction of their classes (0, 1, 2, …, 9 in the laboratory). To summarize, the diagram of training and testing processes are as follows:

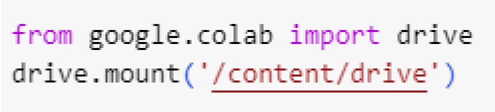


2. Data formatting and Preparation of Data

2.1 Getting Started:

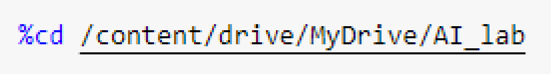
To begin, we should make sure our google drives are still mounted and that we have navigated to the correct directory, where we can find the files we need to work with:

- If you're using Colab, the google drive must be mounted in order to work with the files you want to use. You can use the following command shown last tutorial:



```
from google.colab import drive
drive.mount('/content/drive')
```

- You can navigate through folders using the magic command %cd as shown (make sure you input the correct file path; it won't look the same as this one!):

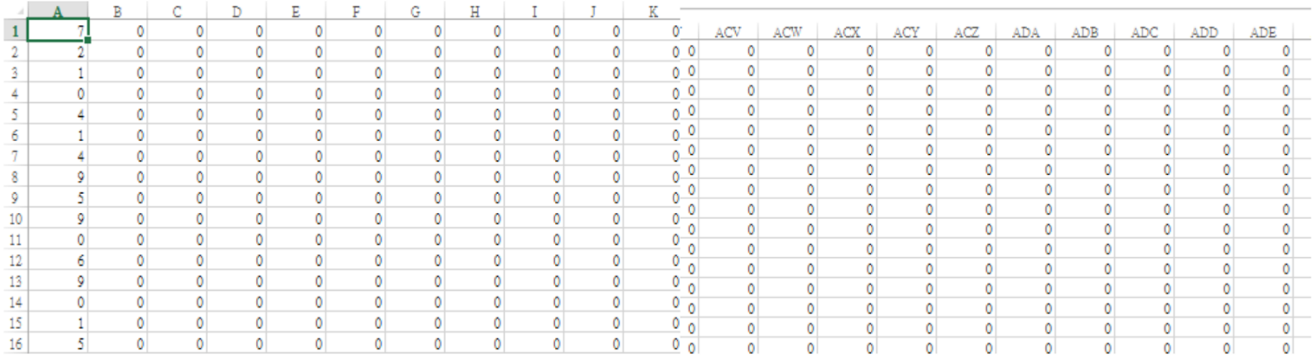


```
%cd /content/drive/MyDrive/AI_lab
```

- To check the files you want to access are located in the folder you're in, you can use the commands !dir or !ls.

2.2 Data Formatting:

In your working directory you should have two data files: "mnist_train.csv" for training and "mnist_test.csv" for testing. Comma-Separated Values (**CSV**) is a common file format used to store data in tabular format. Specifically, "mnist_train.csv" contains all the 60,000 training samples with labels in tabular format while "mnist_test.csv" contains 10,000 testing samples with labels in tabular format. You can read them by executing "data.py" file. The data format is in label-image style. You can open "mnist_train.csv" and/or **"mnist_test.csv" using EXCEL or Notepad to see the structure.** It looks like as follows.



	A	B	C	D	E	F	G	H	I	J	K	ACV	ACW	ACX	ACY	ACZ	ADA	ADB	ADC	ADD	ADE
1	7	0	0	0	0	0	0	0	0	0	0										
2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1	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
5	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

In the figure above, each row represents one training data. Each row has 1+784 columns, where the first column represents the ground truth label(range from 0 to 9, 0 represents this row is a digit 0, 1 represents this row is a digit 1 and so on.) and the rest 784 columns represent the 28x28 (=784) pixels. That is, as shown in Figure 4,

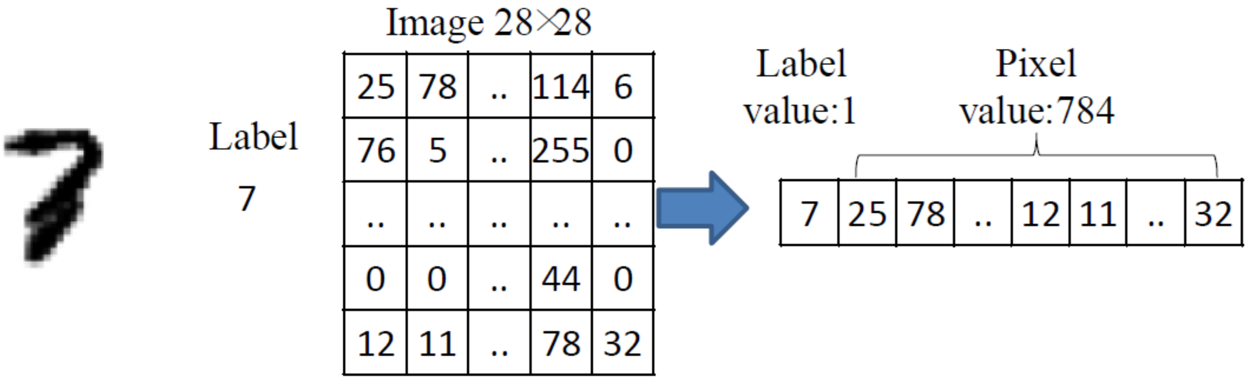


Figure 4. One example of training data

Each row represents a pair of label and image. Knowing the format, we can read all the data from csv files to the memory using the following commands.

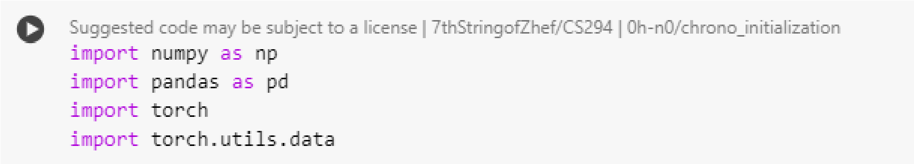
2.3 Import necessary files and libraries:

Next import the maths library: numpy, the data manipulation tool: pandas and torch and torch utilities into your working environment.

```
>>> import numpy as np #import Python maths lib. Numpy & name it np
>>> import pandas as pd #import manipulation tool pan..& name it pd
>>> import torch #import open-source machine learning lib
>>> import torch.utils.data #import torch utilities
```

These commands import the necessary libraries for data preparation.

In Colab your notebook should look like this:



```
import numpy as np
import pandas as pd
import torch
import torch.utils.data
```

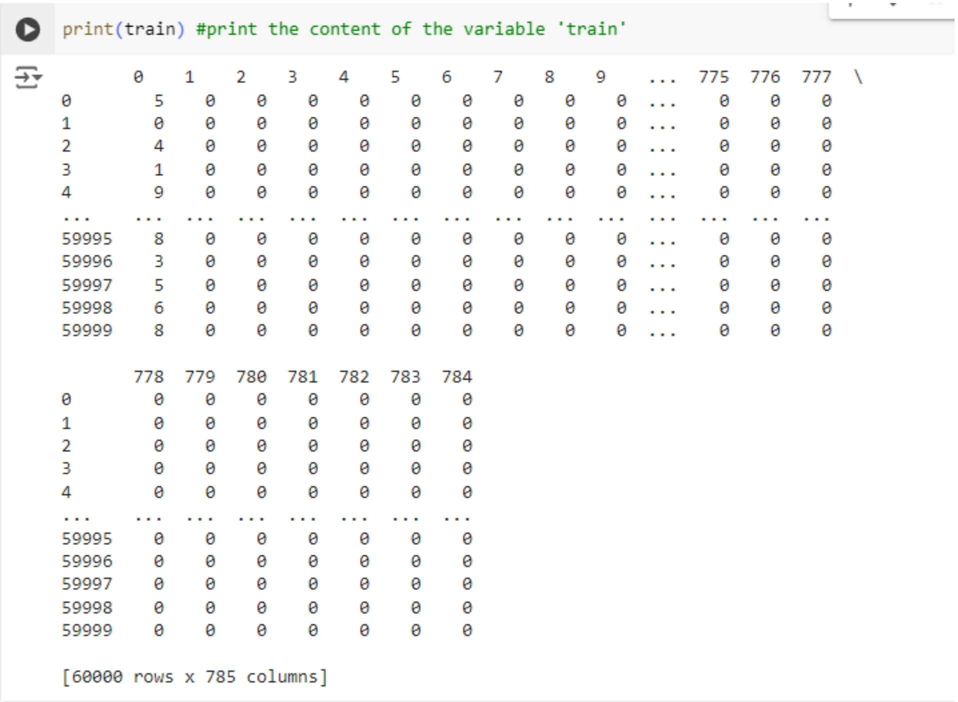
2.4 Converting data into Python tabular form:

In order to process the data in Python, we have to convert the data into Python tabular form. This can be done using the following command:

```
train = pd.read_csv('mnist_train.csv', header=None)
```

This reads a comma-separated values (csv) file called ‘mnist_train.csv’ in your current working directory into DataFrame which is commonly used tabular form in Python. A variable called train is assigned to store the DataFrame. pd.read_csv(file_path, header=None) is a function to perform the task, file_path is a parameter to be entered and it indicates the place where stores your csv file. To read ‘mnist_train.csv’ which is located in your current working directory, file_path should be replaced by ‘mnist_train.csv’. Header is another parameter to be set. header=None means that there is no row used as the column names. In other words, the first row of the file is already the data we need to use.

We can have a look at how the data has been read into a tabular form by using the print function:



```
print(train) #print the content of the variable 'train'
```

```
       0    1    2    3    4    5    6    7    8    9    ...  775  776  777  \
0        5    0    0    0    0    0    0    0    0    0  ...    0    0    0
1        0    0    0    0    0    0    0    0    0    0  ...    0    0    0
2        4    0    0    0    0    0    0    0    0    0  ...    0    0    0
3        1    0    0    0    0    0    0    0    0    0  ...    0    0    0
4        9    0    0    0    0    0    0    0    0    0  ...    0    0    0
...    ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...
59995    8    0    0    0    0    0    0    0    0    0  ...    0    0    0
59996    3    0    0    0    0    0    0    0    0    0  ...    0    0    0
59997    5    0    0    0    0    0    0    0    0    0  ...    0    0    0
59998    6    0    0    0    0    0    0    0    0    0  ...    0    0    0
59999    8    0    0    0    0    0    0    0    0    0  ...    0    0    0

       778  779  780  781  782  783  784
0        0    0    0    0    0    0    0
1        0    0    0    0    0    0    0
2        0    0    0    0    0    0    0
3        0    0    0    0    0    0    0
4        0    0    0    0    0    0    0
...    ...  ...  ...  ...  ...  ...  ...
59995    0    0    0    0    0    0    0
59996    0    0    0    0    0    0    0
59997    0    0    0    0    0    0    0
59998    0    0    0    0    0    0    0
59999    0    0    0    0    0    0    0

[60000 rows x 785 columns]
```

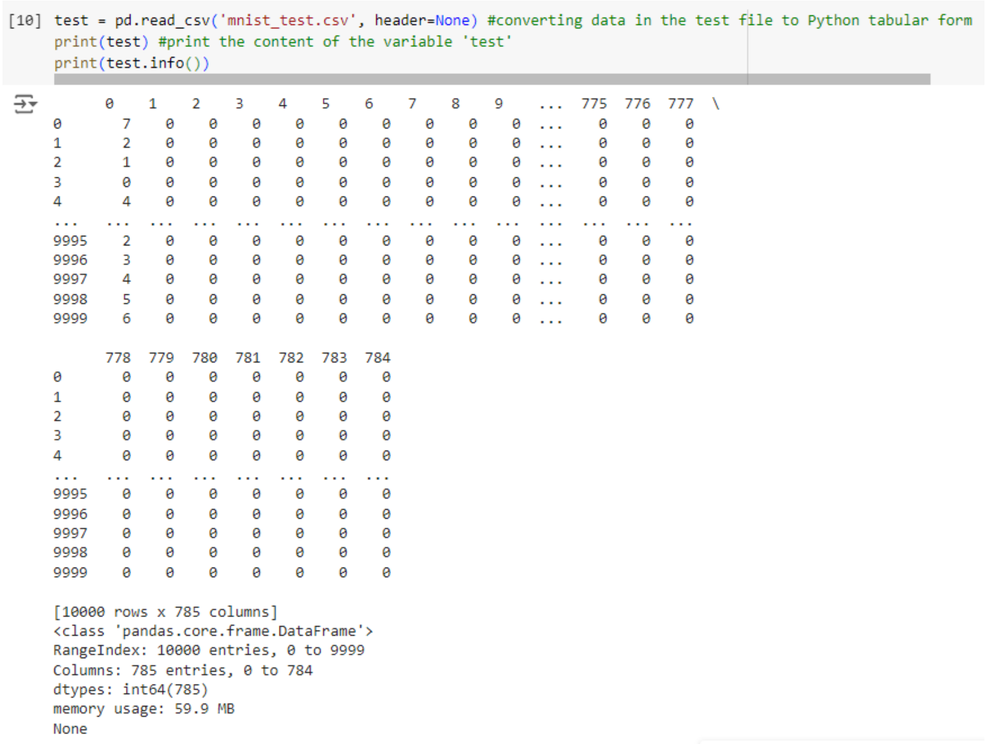
You can also try viewing the data structure with `print(train.info())` to learn about some of the details.

You should see that train is a DataFrame with 60,000 rows and 785 columns which is the same as mentioned previously. You can also see that the data type of this DataFrame is int64 (64-bit integer) and the reading of this csv file occupies 359.3 MB which is the memory usage.

Try now as **Exercise 1** to also convert the testing data into Python tabular form (you can scroll down for the solution).

This time, you should see that test is a Data Frame with 10,000 rows and 785 columns. This csv file occupies 59.9 MB.

Solution:



```
[10] test = pd.read_csv('mnist_test.csv', header=None) #converting data in the test file to Python tabular form
     print(test) #print the content of the variable 'test'
     print(test.info())
```

```
      0    1    2    3    4    5    6    7    8    9    ...  775  776  777  \
0       7    0    0    0    0    0    0    0    0    0  ...    0    0    0   
1       2    0    0    0    0    0    0    0    0    0  ...    0    0    0   
2       1    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   
4       4    0    0    0    0    0    0    0    0    0  ...    0    0    0   
...   ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...   
9995    2    0    0    0    0    0    0    0    0    0  ...    0    0    0   
9996    3    0    0    0    0    0    0    0    0    0  ...    0    0    0   
9997    4    0    0    0    0    0    0    0    0    0  ...    0    0    0   
9998    5    0    0    0    0    0    0    0    0    0  ...    0    0    0   
9999    6    0    0    0    0    0    0    0    0    0  ...    0    0    0   

      778  779  780  781  782  783  784  
0       0    0    0    0    0    0    0  
1       0    0    0    0    0    0    0  
2       0    0    0    0    0    0    0  
3       0    0    0    0    0    0    0  
4       0    0    0    0    0    0    0  
...   ...  ...  ...  ...  ...  ...  ...  
9995    0    0    0    0    0    0    0  
9996    0    0    0    0    0    0    0  
9997    0    0    0    0    0    0    0  
9998    0    0    0    0    0    0    0  
9999    0    0    0    0    0    0    0  

[10000 rows x 785 columns]
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Columns: 785 entries, 0 to 784
dtypes: int64(785)
memory usage: 59.9 MB
None
```

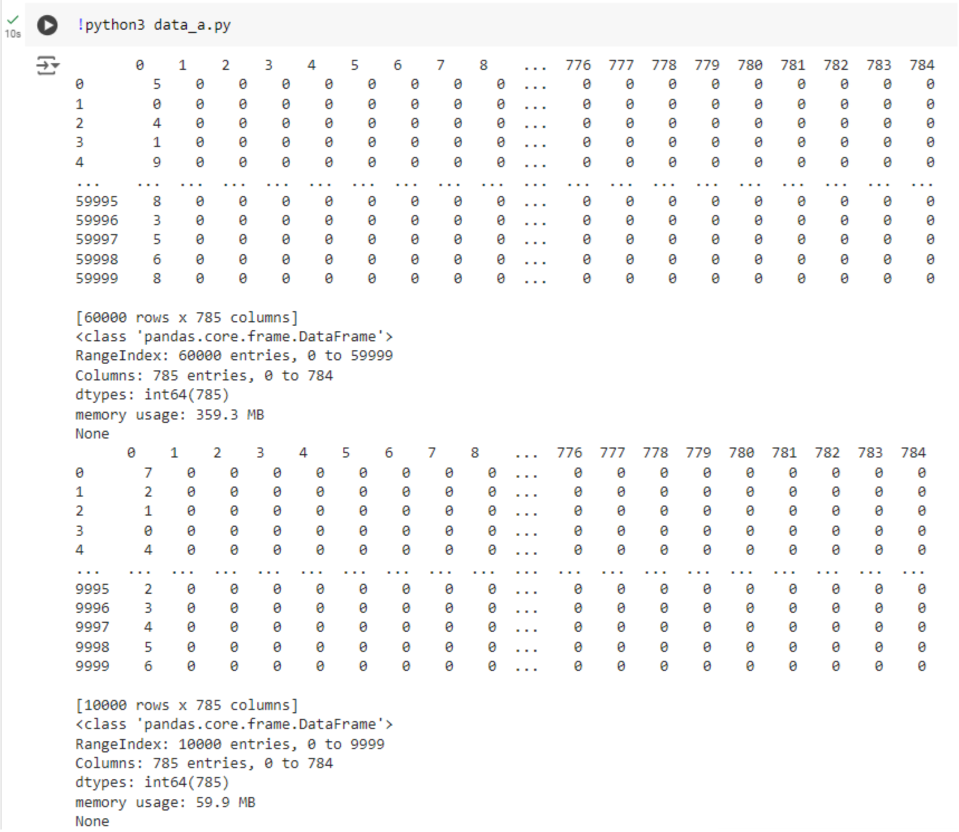
This is how we read our data from csv files to Python variable. In the above, we directly use the python interpreter to type the commands one by one. In practice, we can type all the commands in a python script (a batch file) (with extension .py) beforehand and we can directly execute the script file in the terminal.

2.5 Python Script (batch processing)

We will now try running the same commands from a python script file. This has already been written for you in ‘data_a.py’, which you can check out for yourself by clicking on the file tab on the left and opening the corresponding file.

You can try running the file by typing `!python3 data_a.py`. This command is to execute ‘data_a.py’ using Python. This is a basic command to run a python script or batch file. Later on, you will need to execute other python script files such as ‘eval.py’.In the future when you want to run files, just replace ‘data_a.py’ with whatever the name of the file you are trying to run is called.

You should get a result that looks like this:



```
!python3 data_a.py
```

```
         0    1    2    3    4    5    6    7    8  ...  776  777  778  779  780  781  782  783  784
0        5    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0
1        0    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0
2        4    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0
3        1    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0
4        9    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0
...    ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...
59995    8    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0
59996    3    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0
59997    5    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0
59998    6    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0
59999    8    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0

[60000 rows x 785 columns]
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 60000 entries, 0 to 59999
Columns: 785 entries, 0 to 784
dtypes: int64(785)
memory usage: 359.3 MB
None
        0    1    2    3    4    5    6    7    8  ...  776  777  778  779  780  781  782  783  784
0       7    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0
1       2    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0
2       1    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
4       4    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0
...   ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...
9995    2    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0
9996    3    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0
9997    4    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0
9998    5    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0
9999    6    0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0

[10000 rows x 785 columns]
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Columns: 785 entries, 0 to 784
dtypes: int64(785)
memory usage: 59.9 MB
None
```

2.6 Further formatting(Separating label & Data):

The above format does not exactly fit the NN package, since the format does not differentiate the class label and pixel values. Hence, we have to separate the class labels and the pixel values. We will do this by editing the data_a.py file.

Open the file and add the following lines:

```
train_label = train.iloc[:,0].values #get the label set from all rows of the 1stcolumn (0th column)
train_img = train.iloc[:,1:] #get the image set from all rows the rest of the columns
print(train_label) #print train_label
print(train_img) #print train_imag
```

Press **Ctrl+S** to save the file and then rerun it.

You can see that there is a line [5 0 4 … 5 6 8] which is the variable train_label. This variable becomes a one dimensional numpy array which stores the class labels of all the training samples. On the other hand, you can also see that train_img is a DataFrame with 60,000 rows and 784 columns. [7 2 1 … 4 5 6] is the test_label and test_img should have the size of 10,000 rows and 784 columns.

If you are having issues just run data_b.py as this contains the original and added code.

Appendix: Further explanation of commands

```
train_label = train.iloc[:,0].values
```

This command returns values of all the rows with only the first column of the variable called train to a variable called train_label.
.iloc[:,0] indicates the region of interest. The first parameter inside the square bracket [,] indicates the location of row and : means all the rows. The second parameter inside square bracket [,] indicates the location of column and 0 means the first column which is the class label.

```
train_img = train.iloc[:,1:]
```

This command returns a DataFrame with the selected region to a variable called train_img. The selected region is the entire DataFrame except the first column.
Similarly, .iloc[:,1:] indicates the region of interest (i.e. selected region). The first parameter inside the square bracket [,] indicates the location of row and : means all the rows. The second parameter inside square bracket [,] indicates the location of column and 1: means all the columns except the first column.

```
print(train_label)
```

Same as before, we print a variable called train_label.

```
print(train_img)
```

Print a variable called train_img.

2.7 Shape of an Array and Reshaping:

The following shows the way to get the shape (dimension) of an array.

Open your data_a.py file and type:

```
print(train_img.shape) #This gives the shape (dimension) of array train_img (return a tuple)
print(test_img.shape) # this gives the shape (dimension) of array test_img (return a tuple)
```

Then save and run it.
If you are struggling data_d.py shows the file you should end up with.

Again, you see can that train_img contains 60,000 rows and 784 columns. On the other hand, test_img contains 10,000 rows and 784 columns.

The training set is used for updating the model and the testing set is used for evaluation and to decide when we stop the training. After loading the data, we need to tell the package that it is a set of 2-D data (not 1-D); hence we have to reshape the row-wised pixel values to 2D matrix, that is, 28×28, as the real input for the CNN model.

Back to the text editor of your ‘data_a.py’, type:

```
train_img = train_img.values.reshape(-1,1,28,28) #Reshape the image set to 28x28
test_img = test_img.values.reshape(-1,1,28,28) #Reshape(batch, channel, x,y), -1   by package
train_img = train_img / 255.0 #Normalization, divide it by 255
test_img = test_img / 255.0 #Normalization, divide it by 255
print(train_img.shape)
print(test_img.shape)
```

If you are struggling data_e.py shows the file you should end up with.

Appendix: Further explanation of commands

These are functions to give a new shape to an array while containing the same data. “shape” is a parameter to be entered. It defines the new shape to be assigned.

```
train_img = train_img.values.reshape(-1,1,28,28)
```

train_img.values.reshape(-1, 1, 28, 28) is to reshape train_img.values into a 4-dimensional array. In image processing using pytorch, we usually formulate our input to B, C, H, W.
B is the batch size (will be explained later, we may regard this as the number of training/testing samples we have at this stage), Parameter -1 means that it is an unknown dimension and we would like to let the function to figure it out based on the other parameters.
C is the number of channels (i.e. 3 for RGB image, 1 for gray-scale image).
H and W are the height and width of each input image respectively. As mentioned before, each training image is gray-scale and with the same size of 28×28. Hence, the last three parameters of the reshape function are 1, 28, 28 respectively.

```
test_img = test_img.values.reshape(-1,1,28,28)
```

This command is the same as the above. The only difference is we deal with test_img instead of train_img this time.
(see line 19-20 in data.py)

```
train_img = train_img / 255.0
```

This command divides train_img by 255.0 to normalize all the values from 0 to 1. Note that an 8-bit gray-scale image has pixel values range from 0 to 255 ($2^8 = 256$).

```
test_img = test_img / 255.0
```

Similar to the above, this command divides test_img by 255.0 to normalize all the values from 0 to 1.
You may check the new shape of train_img and test_img using
print(train_img.shape) and

`print(test_img.shape)` The shapes are (60000, 1, 28, 28) and (10000, 1, 28, 28) respectively.

2.8 Converting to GPU (Tensor) format:

If the system has GPU (fast graphic card), we need to pack the data in Tensor format which is a must for using the GPU accelerated Pytorch. Note that if cuda (GPU) is available for Pytorch, we can put the data in Tensor format on GPU for speeding up process. Otherwise, the data in Tensor format can also be processed by using CPU.
Now, we need to pack the data in Tensor format which is a must for using GPU accelerated Pytorch. Again, switch to the text editor of your ‘data_a.py’ and type:

```
torch_X_train = torch.from_numpy(train_img).float() #convert numpy array ‘train_img’ to tensor ‘torch_X_train’
torch_y_train = torch.from_numpy(train_label) #convert numpy array ‘train_label’ to tensor ‘torch_y_train’
torch_X_test = torch.from_numpy(test_img).float()
torch_y_test = torch.from_numpy(test_label)
print(‘torch_X_traindimension: ’, torch_X_train.shape) #print a string ‘torch_X_train dimension:’ and torch_X_train.shape gives the shape (dimension) of tensor torch_X_train
print(‘torch_y_train dimension: ’, torch_y_train.shape) #print a string ‘torch_y_train dimension:’
```

and torch_y_train.shape gives the shape (dimension) of tensor torch_y_train

```
print(‘torch_X_test dimension: ’, torch_X_test.shape)
print(‘torch_y_test dimension: ’, torch_y_test.shape)
```

The pre-written code can be found in data_f.py if you struggle.

You will see that our training and testing data are converted into torch Tensor format.

Appendix: Further explanation of commands

```
torch.from_numpy(arr)
```

This is a command to convert numpy array (one basic data format in Python) into Tensor format. Note that torch Tensor is a basic data format for Pytorch.
Note that Tensors are similar to numpy n-dimensional arrays, in which Tensors can be used on a GPU to accelerate computing. In other words, Tensor is a replacement for numpy to use the power of GPUs.
arr is a parameter to be entered. It should be a numpy array

`.float()` is to indicate that the data type is floating point number.
Note that the data type of the labels should be integer.

```
torch_X_train = torch.from_numpy(train_img).float()
```

This command converts train_img (a numpy array) into torch_X_train (torch Tensor).

```
torch_y_train = torch.from_numpy(train_label)
```

This command converts train_label (a numpy array) into torch_y_train (torch Tensor).

```
torch_X_test = torch.from_numpy(test_img).float()
```

This command converts test_img (a numpy array) into torch_X_test (torch Tensor).

```
torch_y_test = torch.from_numpy(test_label)
```

This command converts test_label (a numpy array) into torch_y_test (torch Tensor).
(see line 27-30 in data.py)

```
print(‘torch_X_train dimension: ’, torch_X_train.shape)
```

this print(‘torch_X_train dimension:’, torch_X_train.shape) consists of two parameters.
‘torch_X_train dimension: ’ is a string to be printed and torch_X_train.shape is the second piece of information to be printed. It shows the structure of the variable torch_X_train.

```
print(‘torch_y_train dimension:’, torch_y_train.shape)
```

Like the previous. To print a string ‘torch_y_train dimension: ’ and torch_y_train.shape.

```
print(‘torch_X_test dimension: ’, torch_X_test.shape)
```

Like the previous. To print a string ‘torch_X_test dimension: ’ and torch_X_test.shape.

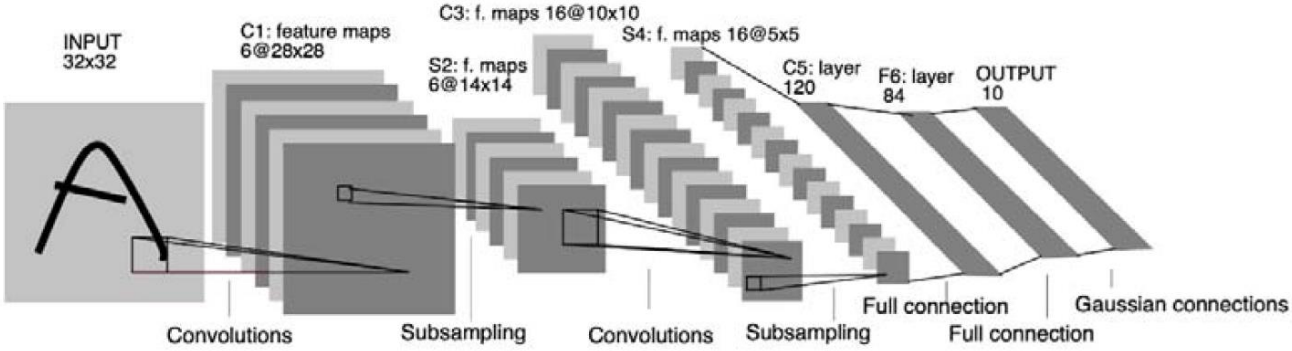
```
print(‘torch_y_test dimension: ’, torch_y_test.shape)
```

Like the previous. To print a string ‘torch_y_test dimension: ’ and torch_y_test.shape.

3. Training

3.1 Reviewing LeNet5, Directory Preparation and Data Preparation

As shown in the figure below, LeNet5 consists of two sets of convolutional and subsampling layers, a flattening convolutional layer then two fully-connected layers followed by a softmax layer. We recall that the first two sets of the convolutional and subsampling layers act as feature extractor in the model to extract invariant features which are useful for the recognition while the remaining layers act as classifier to recognize the input image based on the extracted features. For more details about the convolutional and fully-connected layers, please refer to the following links:
https://pytorch.org/docs/master/generated/torch.nn.Conv2d.html(Convolutional layer)
https://pytorch.org/docs/master/generated/torch.nn.Linear.html#torch.nn.Linear(Fullyconnected layer)
You may also learn more about other types of layers at:https://pytorch.org/docs/master/nn.html



Normally, training and testing data (the images for validation) are stored inside a root directory ‘datasets’, and then grouped according to specific data set, and finally split into test and train folders. For the MNIST data set, the ‘train’ set (for updating network parameters) has 10 folders representing the 10 classes. Each folder contains images corresponding to a particular digit. The ‘test’ set, otherwise known as the ‘val’ set is for monitoring the training (e.g. ensuring the network doesn’t over-fit the data). It also contains a similar 10 class structure.

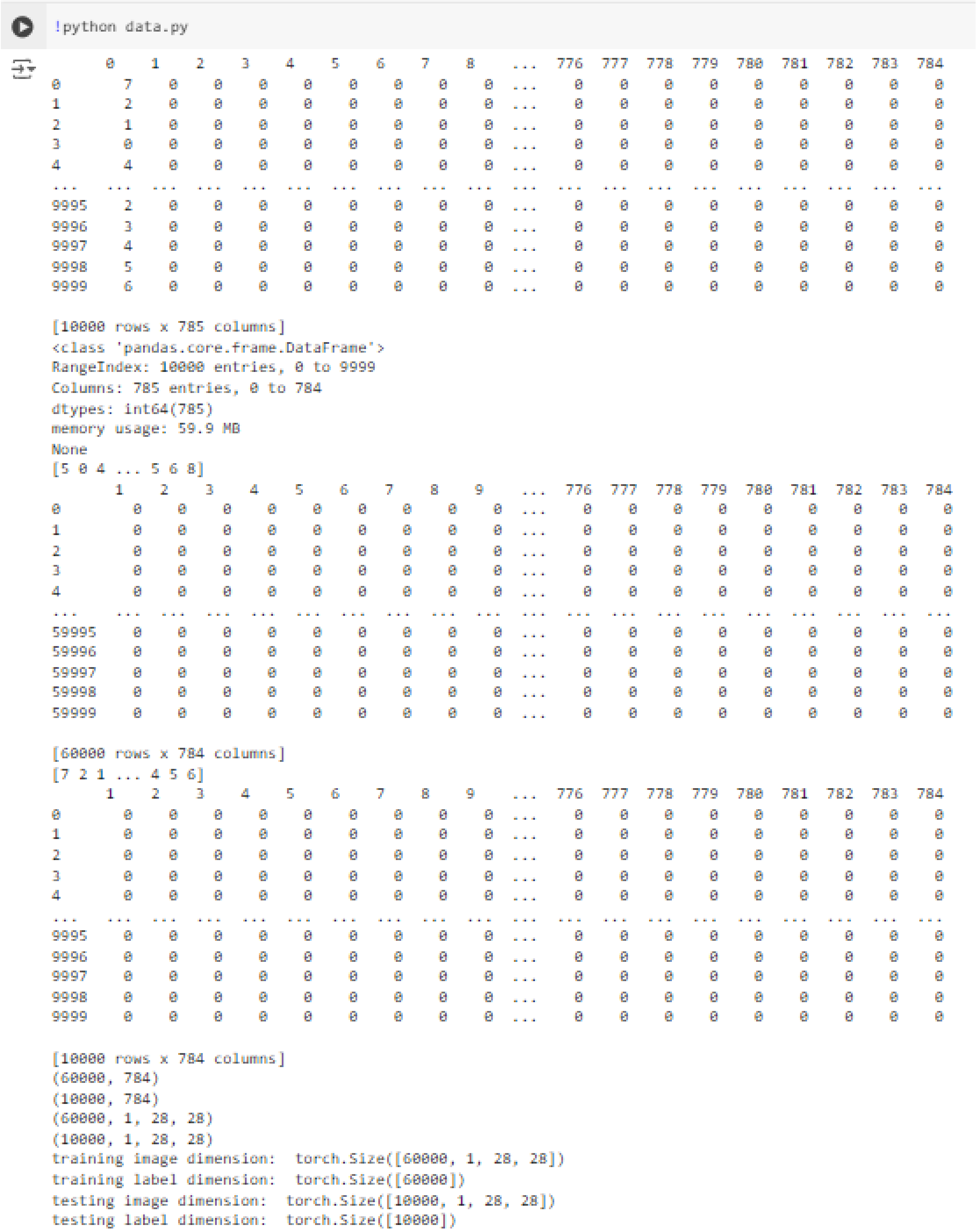
The procedure for loading the data set can be found in the ‘data.py’ file. In order to access the files we need for this section we must ensure the files we need are in our working directory, or navigate to a suitable working directory. This should be
/summer_school_24/summerschool_files/tutorial1/Exp2_4
.

Below we explain the lines of code for getting the train and test images into the format of the PyTorch Data Loader:

```
import os # import os library for file path operations
import numpy as np # import numpy library for array operations, name it as np
import torch # import torch library for using neural network (nn)
import torchvision.datasets as datasets # import torchvision datasets library, name it as datasets
import torchvision.transforms as transforms # import torchvision transforms for input pre-process, name it
transforms
def get_train_and_test_data(data_path, data_transforms, batch_size): # define a function called
get_train_and_test_data (), it takes three input params, namely data_path (where is the data), data_transforms
(pre-process), and
batch_size
        print("\nStart to load train and test data") # print a string, ‘start to load the train and test data’
        images_datasets = {x: datasets.ImageFolder(os.path.join(data_path, x), data_transforms[x]) for x in
        ['train', 'te
        st']}
```

```
        # create images_datasests, which has two lists, ‘train’ and ‘test’, contains all the image paths inside the
        ‘train’ and ‘test’ folders respectively
        train_data_loader = torch.utils.data.DataLoader(images_datasets['train'], batch_size=batch_size,
        shuffle=True, num_workers=4) # create train data loader, randomly (shuffle=True) separate the train
        images into a number of batches
        test_data_loader = torch.utils.data.DataLoader(images_datasets['test'], batch_size=batch_size,
        shuffle=False,
        num_workers=4) # create test data loader, accordingly (shuffle=False) separate the test images into a
        number of batches
        dataset_sizes = {x: len(images_datasets[x]) for x in ['train', 'test']}# get dataset sizes, how many train and
        test images
        class_names = images_datasets['train'].classes # obtain class names of the dataset (based on names of
        subfolders)
        num_classes = len(class_names) # get how many classes are defined, store it to ‘num_classes’
        print('The total number of training and testing images: {}'.format(dataset_sizes))# print a string, show
        no. of images
        print('The total number of classes: {}'.format(num_classes)) # print a string, show the no. of classes
        print('Class names: {}'.format(class_names)) # print a string, show the class names

        return train_data_loader, test_data_loader, dataset_sizes, class_names, num_classes # return the above
        defined
        # variables for further usage in the main program
# the following is for testing the get_train_and_test_data() function

if __name__ == "__main__": # define the main() entry point
        data_transforms = { # define data_transforms for ‘train’ and ‘test’ images
        'train': transforms.Compose([transforms.Grayscale(), # convert to grayscale
        transforms.ToTensor() # put the data to tensor format
        ]),'test': transforms.Compose([transforms.Grayscale(), transforms.ToTensor()])}
        # try to call get_train_and_test_data() to read the datasets/MNIST, batch_size is 1000.
        get_train_and_test_data('datasets/MNIST', data_transforms, batch_size=1000)
```

If you run data.py in your Colab, you should get the following results:



```
!python data.py
```

```
      0    1    2    3    4    5    6    7    8   ...  776  777  778  779  780  781  782  783  784
0     7    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0
1     2    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0
2     1    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
4     4    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0
...  ..  ...  ...  ...  ...  ...  ...  ...  ...   ...  ...  ...  ...  ...  ...  ...  ...  ...  ...
9995  2    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0
9996  3    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0
9997  4    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0
9998  5    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0
9999  6    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0

[10000 rows x 785 columns]
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Columns: 785 entries, 0 to 784
dtypes: int64(785)
memory usage: 59.9 MB
None
[5 0 4 ... 5 6 8]
       1    2    3    4    5    6    7    8    9   ...  776  777  778  779  780  781  782  783  784
0      0    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0
1      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
3      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
...   ..  ...  ...  ...  ...  ...  ...  ...  ...   ...  ...  ...  ...  ...  ...  ...  ...  ...  ...
59995  0    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0
59996  0    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0
59997  0    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0
59998  0    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0
59999  0    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0

[60000 rows x 784 columns]
[7 2 1 ... 4 5 6]
      1    2    3    4    5    6    7    8    9   ...  776  777  778  779  780  781  782  783  784
0     0    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0
1     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
3     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
...  ..  ...  ...  ...  ...  ...  ...  ...  ...   ...  ...  ...  ...  ...  ...  ...  ...  ...  ...
9995  0    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0
9996  0    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0
9997  0    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0
9998  0    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0
9999  0    0    0    0    0    0    0    0    0   ...    0    0    0    0    0    0    0    0    0

[10000 rows x 784 columns]
(60000, 784)
(10000, 784)
(60000, 1, 28, 28)
(10000, 1, 28, 28)
training image dimension:  torch.Size([60000, 1, 28, 28])
training label dimension:  torch.Size([60000])
testing image dimension:  torch.Size([10000, 1, 28, 28])
testing label dimension:  torch.Size([10000])
```

As you can see at the bottom of the screenshot, in this MNIST dataset, we have 60,000 train images and 10,000 test images respectively. We have 10 classes in total and the corresponding class names are ‘0’ to ‘9’ as shown in the above

3.2 Defining Networks

In this laboratory, we are using LeNet5 as the CNN network for integer recognition. The basic structure includes 2 convolution layers and 3 fully connected layers. After knowing how we can organize our train and test images, we should define our network structure. The codes for defining the network are in ‘lenet.py’ and explained below:

Input NN and Naming network:

```
import torch # to load pytorch library
import torch.nn as nn # import pytorchmaths library torch.nn and name it as nn
from initialize import weights_init_kaiming_normal# from initialize.py import weights_init_kaiming_normal function
```

Define Structure:

```
class LeNet5(nn.Module): # Create a class from nn.Module and call it LetNet5

    # define LetNet5
    def __init__(self, num_classes): # __init__() function to assign values to names
        super(LeNet5, self).__init__() # super class for inheritance, derives attributes and behaviors from the parent class, nn.Module
        self.conv1 = nn.Conv2d(in_channels=1,out_channels=6,kernel_size=5,stride=1,padding=2)
        #21 define self.conv1 as nn.Conv2d with parameters: no. of in channels=1, no. of out channels=6, filter size=5x5
        # stride=1 and padding=2
        self.acti1 = nn.ReLU() # define activation function acti1 as ReLU() from nn.
        self.maxpool1 = nn.MaxPool2d(kernel_size=2,stride=2,padding=0)
        # define self.maxpool1 as nn.MaxPool2d with params: filter size=2x2, stride=2, and no padding
        self.conv2 = nn.Conv2d(in_channels=6,out_channels=16,kernel_size=5,stride=1,padding=0)
        self.acti2 = nn.ReLU() # define activation function acti2 as ReLU() from nn
        self.maxpool2 = nn.MaxPool2d(kernel_size=2,stride=2,padding=0)
        # define self.maxpool2, same as self.maxpool1

        # Define the fully-connected layers
        self.fc3 = nn.Linear(in_features=400,out_features=120)
        # define self.fc3 as nn.Linear (input 5x5x16=400, output, output to 400 neurons (fully-connected layer))
        self.acti3 = nn.ReLU() # define self.acti3 as nn.ReLU()
        self.fc4 = nn.Linear(in_features=120,out_features=84) # fully-connected layer, covert 120 to 84 neurons
        self.acti4 = nn.ReLU() # define self.acti4 as nn.ReLU(),
        self.fc5 = nn.Linear(in_features=84,out_features=num_classes) # fc layer, convert 84 input to 10 classes
        weights_init_kaiming_normal(self) # same initialization codes but modularized.
```

Main Program:

```
    def forward(self, x):
            # Convolutional Layer, Activation and MaxPooling Layer
        # the first convolutional, activation and maxpooling layer
        x = self.conv1(x)#convolution execution: execute self.conf1() with data x
        x = self.acti1(x)#non-linear function execution: execute self.act1() with data x
        x = self.maxpool1(x)#max-pooling execution: execute self maxpool1() with data x
        # the second convolutional, activation and maxpooling layer
        x = self.conv2(x)#execute self.conv2() with data x
        x = self.acti2(x)#execute …
        x = self.maxpool2(x)#execute …
        # stack the activation maps into 1d vector
```

```
        x = x.view(-1, 400)#reshape the 5x5x16 matrix to 1D of size 400
        # third fully-connected (fc) layer and activation layer
        x = self.fc3(x)#execute fully connected layer self.fc1() with data x
        x = self.acti3(x)#execute ….
        # fourth fully-connected layer and activation layer
        x = self.fc4(x)#execute ….
        x = self.acti4(x)#execute ….
        # last fc layer
        y = self.fc5(x)#execute fully connected layer3 with self.fc3 and output it as output

        return y
```

Let’s explain the structure of the LeNet5. A set of convolutional and subsampling layers consists of a convolutional layer, an activation function layer, and a max pooling layer. The syntax of these three operations are explained as follows:

```
self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5,
stride=1, padding=0)
```

This defines a convolutional layer in which in_channels, out_channels, kernel_size, stride, and padding are the parameters to be defined. in_channels defines the number of input channels (e.g. 3 for RGB image and 1 for grayscale image); out_channels defines the number of output channels; kernel_size means the filter size which controls the receptive field at the corresponding layer; stride controls the step size to slide the filter over the entire image; padding controls the number of zero paddings for each dimension to obtain a desired output spatial size. The output size can be computed by the equations:

$$H_{out} = \frac{H_{in} + 2 \times \text{padding} - (\text{kernel\_size} - 1)}{\text{stride}} + 1$$

$$W_{out} = \frac{W_{in} + 2 \times \text{padding} - (\text{kernel\_size} - 1)}{\text{stride}} + 1$$

For the MNIST dataset, the size of input image is 28×28 and we can substitute the parameters defined at `self.conv1`to calculate the output size after the convolutional layer (the output size is also 28×28 in this case). See https://pytorch.org/docs/master/generated/torch.nn.Conv2d.htmlfor details.

```
self.maxpool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
```

We also define a maxpooling layer to reduce the spatial size when the network goes deeper and deeper. The maxpooling operation is straightforward in which we select the element with the largest activation value among all the elements covered by the kernel. Kernel_size, stride, and padding are the parameters to be defined. kernel_size controls the size of the sampling area; stride controls the step size to slide the filter over the entire image; padding controls the number of zero paddings for each dimension to obtain a desired output spatial size. We usually use maxpooling layer to reduce the input spatial size by half using the above setting. For example, if the input size is 28×28, the output size would be 14×14. For readers who are interested, see https://pytorch.org/docs/master/generated/torch.nn.MaxPool2d.html#torch.nn.MaxPool2dfor details.

To conclude the changes in the sizes among the three layers, namely `self.conv1`, `self.acti1` and `self.maxpool1`, assume that the input is a 28×28 grayscale image and we use the above mentioned parameter setting, to get six 28×28 feature maps after the `self.conv1`. We then get six 28×28 activation maps (filter out those elements with negative values) after the `self.acti1`. Finally, we get six 14×14 activation maps after the `self.maxpool1` as it reduces the input spatial size by half.

```
self.fc3 = nn.Linear(in_features=400, out_features=120)
```

Apart from the three layers explained in above, we also define fully-connected layer in which in_features, and out_features are the parameters to be defined. in_features defines the input size while out_features indicates the output size. For example, at this layer, we define the input size as 400 and the output size as 120. Please see for https://pytorch.org/docs/master/generated/torch.nn.Linear.html torch.nn.Linear details.

3.3 Program Details and Training Strategy

We have to load this program for execution and define the training strategy. The main process for training is described in "train.py". It includes three components:

Import Necessary Libraries:

```
importnumpyas np # to handle matrix and data operation
import pandas as pd # to read csv and handle dataframe
importos # operation system library
import torch # to load pytorch library
importmatplotlib.pyplotasplt # to load matplotlib.pyplot, name it plt
import time # to load time library
import math # to load math library
```

Define Useful Functions:

```
def plot_train_curve(train_info, model_name, learning_rate, batch_size):# define plotting function
        if learning_rate == 0.1: # check learning_rate, then define lr
                lr = '01'
        elif learning_rate == 0.01:
                lr = '001'
        elif learning_rate == 0.001:
                lr = '0001'
        else:
                lr = 'Selfdefined'

        train_info = np.array(train_info) # convert train_info to numpy array.
        train_loss = train_info[:,0] #split train_info into subgroups according to the column that the information
        train_acc = train_info[:,1] #is in
        test_loss = train_info[:,2]
        test_acc = train_info[:,3]
        num_epochs = len(train_loss) #define number of epochs as length of train_loss list
        plt.plot(np.arange(1, num_epochs+1), train_loss) # plot the training loss curve (save as Lab1, 2)
        plt.plot(np.arange(1, num_epochs+1), test_loss)
        plt.ylabel('Loss') #add labels to y axis
        plt.xlabel('Epoch no.') #add labels to x axis
        plt.grid(linestyle='—') #add a grid to the plot
        plt.legend(['Train Loss', 'Test Loss']) #add legend to the plot
        fig_filename = '{}_lr{}_batchsize{}_loss_curve.png'.format(model_name, lr, batch_size)
```

```
        #define a file name for each figure based on its specific information
        plt.savefig(fig_filename, bbox_inches='tight') #save figure under specified file name
        plt.clf() #clear current figure
        plt.plot(np.arange(1, num_epochs+1), train_acc) # plot the training acc curve (save as Lab1, 2)
        plt.plot(np.arange(1, num_epochs+1), test_acc) #next commands are similar to above, essentially
        formatting the graphs
        plt.ylabel('Accuracy')
        plt.xlabel('Epoch no.')
        plt.ylim(0.0, 1.0)
        plt.grid(linestyle='--')
        plt.legend(['Train Acc', 'Test Acc'])
        fig_filename = '{}_lr{}_batchsize{}_acc_curve.png'.format(model_name, lr, batch_size)
        plt.savefig(fig_filename, bbox_inches='tight')
        plt.clf()

        return True

def write_train_info(train_info, model_name, learning_rate, batch_size):# define writing function
        if learning_rate == 0.1: # check learning_rate, then define lr
                lr = '01'
        elif learning_rate == 0.01:
                lr = '001'
        elif learning_rate == 0.001:
                lr = '0001'
        else:
                lr = 'Selfdefined'

        filename = '{}_lr{}_batchsize{}.csv'.format(model_name, lr, batch_size)# define file name for training
        results csv
        df = pd.DataFrame(train_info) #create a data frame for training results
        df.to_csv(filename, header=False, index=False) #save as csv
        return True

def save_checkpoints(model, save_path, model_name, epoch): # define saving checkpoint function

        filename = '{}_epoch_{}.pth'.format(model_name, epoch) #define file name
        torch.save(model.state_dict(), os.path.join(save_path, filename)) #save checkpoint
        return True
```

Define Training and Testing Functions:

```
def train_and_test(opt, model, criterion, optimizer, train_data_loader, test_data_loader, dataset_sizes, device):
        # define train_and_test function
        since = time.time() # record the start training time, save it to variable since
        num_epochs = opt.epochs # define num_epochs which depends on opt.epochs by users
        num_train_iter = math.ceil(dataset_sizes['train'] / opt.batch_size) # compute num_train_iter based on
        # the dataset sizes and the defined batch size
        num_test_iter = math.ceil(dataset_sizes['test'] / opt.batch_size) # compute num_test_iter
        train_info = [] # create train_info, an empty list
        for epoch in range(num_epochs):# create a for-loop, loop over num_epochs
                model.train() # put the model to train mode
                train_loss = 0.0 # create variable train_loss, set it to 0
```

```
    train_acc = 0.0 # create variable train_acc, set it to 0
    test_loss = 0.0 # create variable test_loss, set it to 0
    test_acc = 0.0 # create variable test_acc, set it to 0

    for batch_idx, (inputs, labels) in enumerate(train_data_loader):# create a for-loop, loop over all
    batches in train_data_loader, batch_idx is the index of batches, (inputs,lables) is the pair of each
train image
        inputs = inputs.to(device)# put inputs (train images) to device, if GPU is available,
        device=cuda, else, cpu
        labels = labels.to(device)# put labels (train labels) to device,
        model.zero_grad() # clear up the gradients of the model first
        optimizer.zero_grad() # clear up the gradients of the optimizer first

        # forward pass
        y = model(inputs) # pass inputs (train images) to the model and get the output, y
        _, preds = torch.max(y.data, 1) # use torch.max to get the prediction with the highest
        probability,
        loss = criterion(y, labels) # create the loss, using the predicted y and the ground truth
        labels

        # backward pass
        loss.backward() # backward pass the loss,
        optimizer.step() # update the model parameters
        train_loss += loss.item() * inputs.size(0) # accumulate the train loss of each batch of
        train images
        running_corrects = preds.eq(labels.data.view_as(preds)) # check how many correct
        predictions for the batch

        acc = torch.mean(running_corrects.type(torch.FloatTensor))# calculate the accuracy of
        the prediction
        train_acc += acc.item() * inputs.size(0)# accumulate the train accuracy of each batch of
        train images
```

The above codes show the typical procedure for updating the network parameters through forward pass the training images to the network and backward pass the loss.

Simply speaking, the flow is as follows.

1. Put the model into training mode
2. Clear the accumulated gradients first
3. Forward pass a batch of training images to the network to get the prediction
4. Based on the prediction, we calculate the loss
5. Backward pass the loss and update the parameters

If you would like to watch a basic introduction to the concept, in order to get better intuition for the processes of digit recognition and back propagation, we recommend watching the linked videos from 3Blue1Brown, which have some really nice animations. :)

https://www.youtube.com/watch?v=Ilg3gGewQ5U&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=3

These videos are part of a larger series that would be good to watch, if you are further interested.

Testing:

```
# testing
with torch.no_grad():# use torch.no_grad() to indicate no gradient will be computed for the following lines
    model.eval() # put the model to evaluation mode, no gradient will be stored and no update

    #exactly the same as how we deal with the train images, except that no update will be performed
    for batch_idx, (inputs, labels) in enumerate(test_data_loader):
        inputs = inputs.to(device)
        labels = labels.to(device)

        # forward pass
        y = model(inputs) # make the prediction
        _, preds = torch.max(y.data, 1)
        loss = criterion(y, labels)
        test_loss += loss.item() * inputs.size(0) # get the test loss
        running_corrects = preds.eq(labels.data.view_as(preds))
        acc = torch.mean(running_corrects.type(torch.FloatTensor))
        test_acc += acc.item() * inputs.size(0) # get the test accuracy

        # complete one epoch
        epoch_train_loss = train_loss / dataset_sizes['train']# compute epoch_train_loss by average the train loss
        epoch_train_acc = train_acc / dataset_sizes['train']# compute epoch_train_acc by average the train acc
        epoch_test_loss = test_loss / dataset_sizes['test'] # compute epoch_test_loss by average the test loss
        epoch_test_acc = test_acc / dataset_sizes['test'] # compute epoch_test_acc by average the test acc
        print('Epoch {}/{}, Train loss: {:.4f}, Train acc: {:.4f}'.format(epoch+1, num_epochs, epoch_train_loss, epoch_train_acc))

        # print the training results
        print('Epoch {}/{}, Test loss: {:.4f}, Test acc: {:.4f}'.format(epoch+1, num_epochs, epoch_test_loss, epoch_test_acc))

        # append the results of the current epoch to train_info
        train_info.append([epoch_train_loss, epoch_train_acc, epoch_test_loss, epoch_test_acc])
        # save the checkpoints
        save_checkpoints(model, opt.save_path, opt.model_name, epoch+1)
        time_elapsed = time.time() - since # compute the time to complete one epoch
        # print the required time to complete one epoch
        print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed // 60, time_elapsed % 60))
        # call write function to write the results of the current epoch
        write_train_info(train_info, opt.model_name, opt.lr, opt.batch_size)
        # call the plot function to plot the train results
        plot_train_curve(train_info, opt.model_name, opt.lr, opt.batch_size)

        return model, train_info # return the trained model and the train information.
```

The above codes show the typical procedure for validating the network performance on the validation set. Usually, after one training epoch (i.e. all the training images are fed into the network once), we will test our network on a separate set of images to see the performance. One of the main purposes is to avoid overfitting. The validation loop is exactly the same as the training loop shown above, except that this time we set the model to evaluation mode such that no gradient will be calculated.

Main train file:

The main process for training of LeNet is described in “lenet_main.py”

```
import argparse # import argparse library, for user input
import numpy as np # import numpy library, name it as np
import os # import os library
import torch # import torch library for using nn
import torch.nn as nn # import torch.nn library, name it nn
import torch.optim as optim # import torch.optim, name it optim
import torchvision.transforms as transforms # import torchvision.transforms as transforms
from data import get_train_and_test_data # import the previous defined data load function
from lenet import LeNet5 # import the LeNet5 model
from train import train_and_test # import the previous defined train test functions
"""
input commands
"""
parser = argparse.ArgumentParser()# define parser for getting input arguments when running a batch
# file in command line window
parser.add_argument("--batch_size", type=int, default=1000, help="Batch size")# define batch_size
# default is 1000
parser.add_argument("--epochs", type=int, default=50, help="Number of training epochs")# define epochs
# default is 50
parser.add_argument("--lr", type=float, default=0.01, help="Learning rate")# define lr, default 0.01
parser.add_argument("--data_path", type=str, default='datasets/MNIST', help="Path to dataset")
# define the data_path, i.e. path to the dataset, default is MNIST
parser.add_argument("--save_path", type=str, default='checkpoints', help="Path to save your checkpoints")
# define the save_path, path to save the checkpoints
parser.add_argument("--model_name", type=str, default='LeNet5', help="Model name. LeNet5 or AlexNet")
# define the model_name, LeNet5 or AlexNet
parser.add_argument("--momentum", type=float, default=0.9, help="SGD momentum (default: 0.9)")
# define momentum for optimizer, default is 0.9
parser.add_argument("--weight_decay", type=float, default=5e-4, help="SGD weight decay (default: 5e-4)")
# define weight_decay for optimizer, default is 5e-4
opt = parser.parse_args()# define a variable opt, to store all the input arguments by users.
```

Define the main function:

```
if __name__ == "__main__": # define the main entry point.
        if not os.path.exists(opt.save_path): # check whether the path for saving checkpoints exists or not
        os.mkdir(opt.save_path) # if not, create the directory
        data_transforms = { # define the data_transforms, same as above
                'train':transforms.Compose([transforms.Grayscale(),transforms.ToTensor()]),
                'test':transforms.Compose([ transforms.Grayscale(), transforms.ToTensor()])}
        train_data_loader, test_data_loader, dataset_sizes, class_names, num_classes =
        get_train_and_test_data(opt.data_path, data_transforms, opt.batch_size) # call the get data function, load
        the train and test data
        print("\nDefine model and loss function ...") # print a string, define model and loss function
        model = LeNet5(num_classes) # define LeNet5 class object, name it as model
```

```
criterion = nn.CrossEntropyLoss() # define criterion, CrossEntropyLoss (typical classification loss)
print("Completed")
print("\nMoving to GPU if possible ...")
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")# check GPU is available, if so, define device
model = model.to(device) # put the model to device (cuda for GPU, cpu for CPU)
criterion = criterion.to(device) # also put the loss function to device
print(device)
print("Completed")
print("\nSetting optimizer ...")
optimizer = optim.SGD(model.parameters(), lr=opt.lr, momentum=opt.momentum, weight_decay=opt.weight_decay) # define the optimizer to be used for updating parameters, we use typical Stochastic Gradient Descent in this lab
print("Setting optimizer completed")
print("\nStart training ...")
model = train_and_test(opt, model, criterion, optimizer, train_data_loader, test_data_loader, dataset_sizes, device) # run the train_and_test function for training and validation.
print("Training completed")
```

The above codes show how the whole training process is defined. After loading the train and test images, we have to define our model and the loss function to be used to train the model.
In this example, we use LeNet5 and the loss function is a typical Cross Entropy Loss function for classification task. Then, we move everything to the GPU is possible. If so, the whole training process can be sped up. After that, we have to define the optimizer to be used for updating the parameters. In this lab, we introduce the most typical optimizer called “SGD”, Stochastic Gradient Descent, to train our model. As you can see, there are three hyper-parameters to be defined for this optimizer, namely learning rate (lr), momentum, and weight decay. Apart from these three hyper-parameters, the batch size and the number of training epochs are also some typical hyper-parameters needed to be defined for training. In this lab, we fix all the hyper-parameters for simplicity and we will modify only two hyperparameters, namely learning rate and batch size for observing their effects on the training process.

The key components include:

1. batch_size: it defines the size of one data group. A smaller batch can save computation memory for deeper and more complex networks. A larger batch can make use of available memory to speed up the training process and learn a more general representation of the training data. The disadvantage of using a small batch is lack of representative of the training data, it can affect the general ability of the model. Using a large batch number can also be problematic because the model may learn an average distribution that does not explore the complexity of the data.

2. epochs: it defines how many iterations we want to train the model. Generally, the more epochs we have, the better prediction results we have.

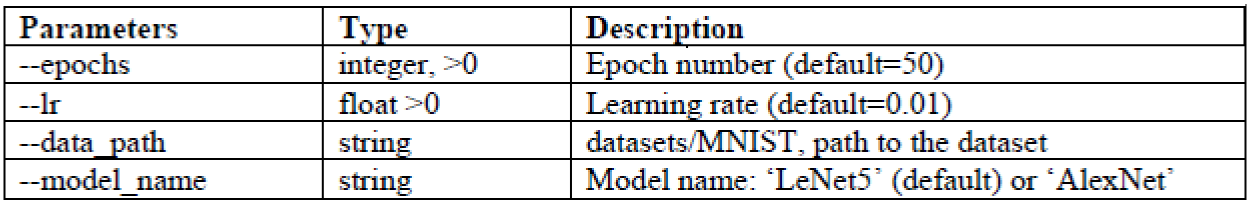
3. learning rate: it defines the step of updating parameters. You need to choose a proper learning rate to control the network training. A large learning rate can skip the minimal point and a small learning rate can be too slow to reach the optima.

4. Optimization approach: The standard approach for deep learning training is Stochastic Gradient Descent (SGD). There are many other approaches that speed up the training process and achieve better results.

5. Fine-tuning: The operation of fine-tuning is similar to initialization. The goal is to initialize the network parameters with some prior information. Different from initialization, fine-tuning means we harvest the model trained from other dataset. For example, we want to use ImageNet to train a model for classification. However, ImageNet dataset is too big to be processed using our own computers. With available pre-trained model provided by others, we can use it to initialize the network parameters to fine-tune. There are many other issues for training a CNN model. We only list the most important points. During the training, we provide you with a compact command that you can use to modify the parameters by yourself as follows:

python lenet_main.py

We provide parameters for you to change, including:



Parameters	Type	Description
--epochs	integer, >0	Epoch number (default=50)
--lr	float >0	Learning rate (default=0.01)
--data_path	string	datasets/MNIST, path to the dataset
--model_name	string	Model name: ‘LeNet5’ (default) or ‘AlexNet’

For example, you can try the default setting as

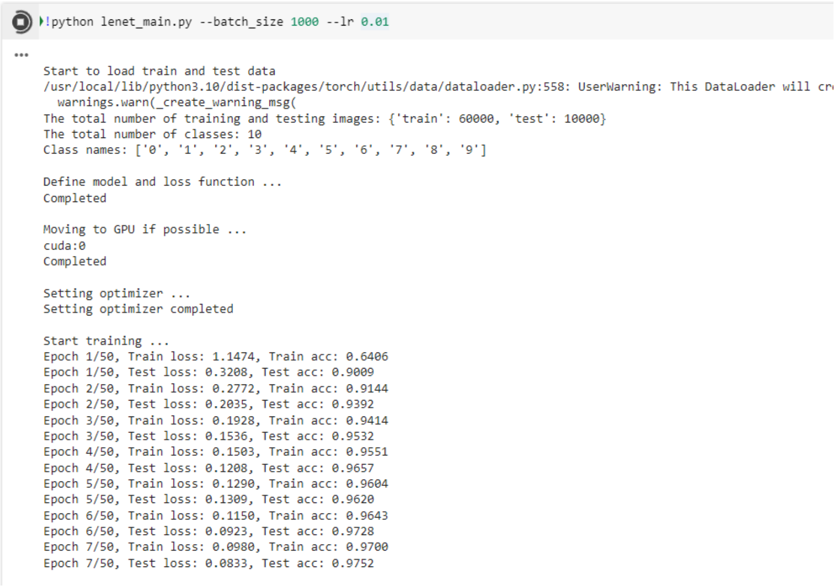
```
python lenet_main.py --batch_size 1000 --lr 0.1
```

3.4 Starting Training and Getting Training Results

This part is about the training of the model.
In this lab, we fix the epoch number to 50, optimizer to ‘SGD’, initialization to ‘Kaiming’. However, you may change batch size and learning rate and then observe the effects on the training.
In your terminal, go to your working directory and type:

```
!python lenet_main.py --batch_size 1000 --lr 0.01
```

in your command line window. You should see:

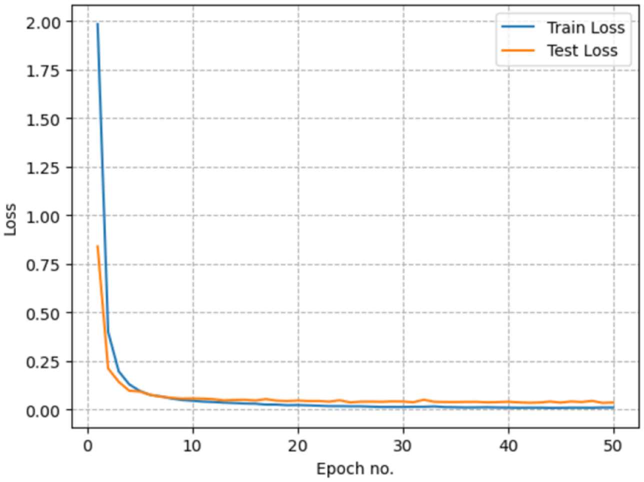


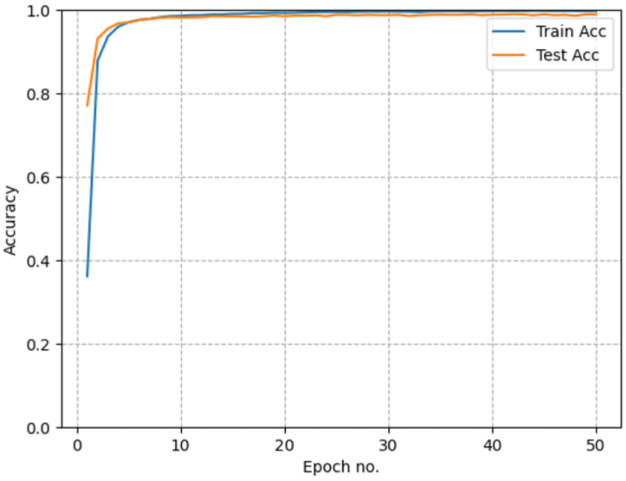
```
!python lenet_main.py --batch_size 1000 --lr 0.01
```

```
Start to load train and test data
/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:558: UserWarning: This DataLoader will cr
  warnings.warn(_create_warning_msg(
The total number of training and testing images: {'train': 60000, 'test': 10000}
The total number of classes: 10
Class names: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

Define model and loss function ...
Completed

Moving to GPU if possible ...
cuda:0
Completed

Setting optimizer ...
Setting optimizer completed

Start training ...
Epoch 1/50, Train loss: 1.1474, Train acc: 0.6406
Epoch 1/50, Test loss: 0.3208, Test acc: 0.9009
Epoch 2/50, Train loss: 0.2772, Train acc: 0.9144
Epoch 2/50, Test loss: 0.2035, Test acc: 0.9392
Epoch 3/50, Train loss: 0.1928, Train acc: 0.9414
Epoch 3/50, Test loss: 0.1536, Test acc: 0.9532
Epoch 4/50, Train loss: 0.1503, Train acc: 0.9551
Epoch 4/50, Test loss: 0.1208, Test acc: 0.9657
Epoch 5/50, Train loss: 0.1290, Train acc: 0.9604
Epoch 5/50, Test loss: 0.1309, Test acc: 0.9620
Epoch 6/50, Train loss: 0.1150, Train acc: 0.9643
Epoch 6/50, Test loss: 0.0923, Test acc: 0.9728
Epoch 7/50, Train loss: 0.0980, Train acc: 0.9700
Epoch 7/50, Test loss: 0.0833, Test acc: 0.9752
```

The training process is running, and it takes several minutes. When the training process is completed. The trained model is saved inside the 'checkpoints' folder. There should be a csv file for storing the training statistics and two figures show the loss and accuracy curves as follows.

The left shows the train and test loss while the right shows the train and test accuracy.





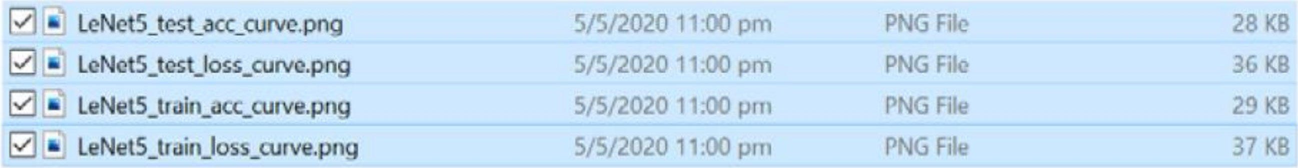
3.5 Trying Different Hyper-Parameters

You should try different learning rates and batch sizes by changing the numbers in bold red in the above command. Try (0.1, 0.01, 0.001) for the learning rates and (10, 100, 1000) for the batch sizes, such that you get 9 sets of training results (3x3 combinations).

We will plot the curves together to explore the effects of different hyper-parameters on the training process. The code for plotting the cures is found inside plot_multi_curves.py. To use it, type:

```
!python plot_multi_curves.py --model_name LeNet5
```

You should observe that there are 4 new figures in your current working directory as follows.



☑ LeNet5_test_acc_curve.png	5/5/2020 11:00 pm	PNG File	28 KB
☑ LeNet5_test_loss_curve.png	5/5/2020 11:00 pm	PNG File	36 KB
☑ LeNet5_train_acc_curve.png	5/5/2020 11:00 pm	PNG File	29 KB
☑ LeNet5_train_loss_curve.png	5/5/2020 11:00 pm	PNG File	37 KB

Exercise 3:

1. Try plotting with the different parameters and compare the differences. You can modify ‘draw_curve_single.py’ to draw training losses for different setups.

2. See if you can train a model which achieves 99% testing accuracy.

3. How do batch size and learning rate affect training results?

4. Using a Modern Pre-Trained (AlexNet) Model for Classification

In this section we will use a state-of-the-art CCN, called AlexNet, for image classification. This CNN has been pre-trained already, so we will just be using it for classification.

Before beginning navigate to …/tutorial1/Exp2_4 and make sure that the following materials are in your current working directory by using the !ls command:

1. ‘alexnet_images’ folder
2. ‘alexnet_eval.py’
3. ‘class_names_ImageNet.txt’

Opening ‘alexnet_eval.py’ you should see the following:

```
#importing the packages we will need to use
import os
import torch
import torch.nn
import torchvision.models as models
import torchvision.transforms as transforms
import torch.nn.functional as F
import torchvision.utils as utils
import cv2
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import argparse

"""
```

```
input commands
"""
paser = argparse.ArgumentParser() #define a variable paser, an argument parser, for input params
paser.add_argument("--test_img", type=str, default='whippet.jpg', help="testing image") #add input arg. Called test_img, default values is whippet.jpg to give the path of the testing image
opt = paser.parse_args() #create a variable opt, for referring to the input params, 'test_img'

# function for visualizing the feature maps
def visualize_activation_maps(input, model):
***out of the scope of this lab***
```

Formatting the input data:

```
# main
if __name__ == "__main__":  #define an entry point the main function
    """
    data transforms, for pre-processing the input testing image before feeding into the net
    """
    data_transforms = transforms.Compose([ #define list of data transforms for input processing
        transforms.Resize(256),            # resize the input to 256x256
        transforms.CenterCrop(224),        # center crop the input to 224x224
        transforms.ToTensor(),             # put the input to tensor format
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])  # normalize the input
        # the normalization is based on images from ImageNet
    ])
```

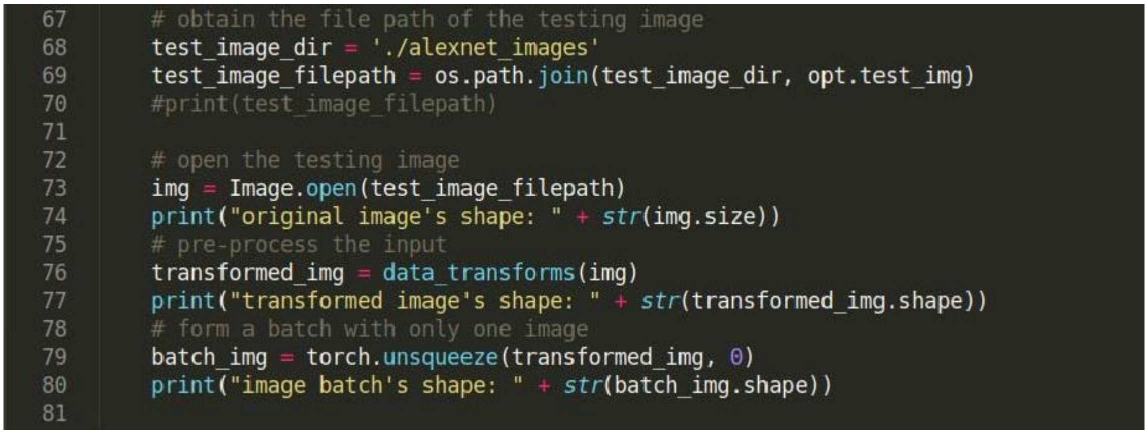
`data_transforms` is for pre-processing the input testing image beforefeeding it into the pre-trained AlexNet. It involves four main transforms:

1. `transforms.Resize(256)`: the input is resize to 256×256
2. `transforms.CenterCrop(224)`: using the centre point of the input, crop theinput to 224×224
3. `transforms.ToTensor()`: same as before, a common practice, put the input toTensor format
4. `transforms.Normalize([mean],[standard d viation])`: normalize the input with respect to the [mean] and [standard deviation]. As our inputis RGB image, there are 3 values for [mean] and [standard deviation] for R, G, B channels. Note
5. that these values are calculate based on the ImageNet dataset which consists of more than a million images.

After defining the format using transforms for pre-processing the input, we read and transform the input testing image as follows.

```
# obtain the file path of the testing image
test_image_dir = './alexnet_images' # create a variable test_image_dir, a string './alexnet_images'
test_image_filepath = os.path.join(test_image_dir, opt.test_img) # get the path of the input testing image
```

```
# open the testing image
img = Image.open(test_image_filepath) # open test image and use variable img to store the test image
print("original image's shape: " + str(img.size)) # print function to show the size of the testing image
# pre-process the input
transformed_img = data_transforms(img) # run the list of pre-processing defined previously
print("transformed image's shape: " + str(transformed_img.shape)) #print to show size of pre-processed image
# form a batch with only one image
batch_img = torch.unsqueeze(transformed_img, 0) # form a batch with 1 image, fit the input size of the model
print("image batch's shape: " + str(batch_img.shape)) # print the shape (dimension) of the batch of a single img
```



```
67      # obtain the file path of the testing image
68      test_image_dir = './alexnet_images'
69      test_image_filepath = os.path.join(test_image_dir, opt.test_img)
70      #print(test_image_filepath)
71
72      # open the testing image
73      img = Image.open(test_image_filepath)
74      print("original image's shape: " + str(img.size))
75      # pre-process the input
76      transformed_img = data_transforms(img)
77      print("transformed image's shape: " + str(transformed_img.shape))
78      # form a batch with only one image
79      batch_img = torch.unsqueeze(transformed_img, 0)
80      print("image batch's shape: " + str(batch_img.shape))
81
```

Lines 68-69 define the file name of the testing image. opt.test_img is the input parameter to be used to indicate the file name of the testing image. Note that the testing image should be stored inside the ‘alexnet_images’ folder.

Lines 73-74 are to read the testing image and print the size of the testing image. Here, we use an existing library package Image to read the testing image with the file path indicated by the variable test_image_filepath. The read image will be pointed by the variable: img.

Lines 76-77 are to pre-process the testing image and print the size of the transformed testing image. Here, we use our defined `data_transforms` to pre-process the testing image pointed by img and the transformed testing image is pointed by transformed_img.

Lines 79-80 are to adjust the dimension of the input in order to feed it into the pre-trained AlexNet. Usually, a network receives input with size N×C×H×W, for which N is the batch size, C is the number of input channels, H and W are the height and width of the input respectively. In our case, we input a RGB image to the network. Hence, N = 1, C = 3, H=W=224 (pre-defined by the first layer of AlexNet).

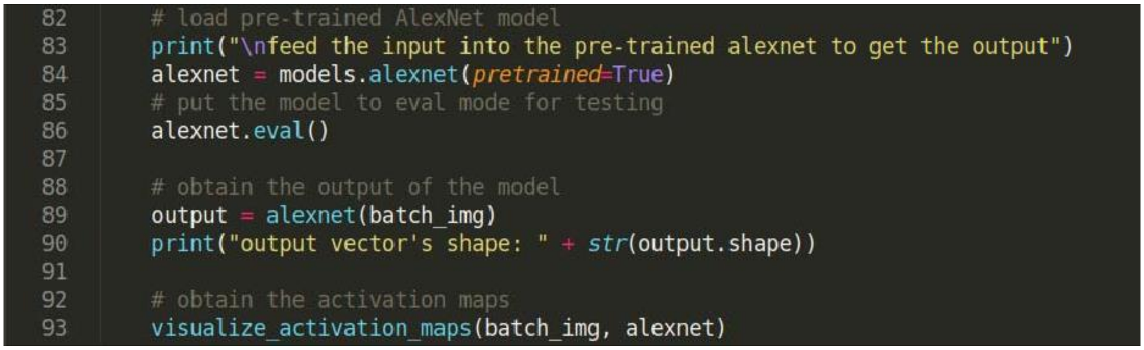
The command: `torch.unsqueeze(transformed_img, 0)` is to add one dimension with value = 1 before the original dimension (dim=0).

With the line 80 which prints the size of the variable batch_img, you can check the changes in the size of the input later on when you are executing the script file.

Load & Test the Pre-trained AlexNet:

Now we can load the pre-trained model and feed our testing image into the model to get the prediction using by the following code.

```
# load pre-trained AlexNet model
print("\nfeed the input into the pre-trained alexnet to get the output") # print a string for our action
alexnet = models.alexnet(pretrained=True) # get pre-trained alexnet from torchvision.models

# put the model to eval mode for testing
alexnet.eval()

# obtain the output of the model
output = alexnet(batch_img) # feed the testing image to the pre-trained alexnet, get output
print("output vector's shape: " + str(output.shape))
# obtain the activation maps
visualize_activation_maps(batch_img, alexnet)
```



```
82      # load pre-trained AlexNet model
83      print("\nfeed the input into the pre-trained alexnet to get the output")
84      alexnet = models.alexnet(pretrained=True)
85      # put the model to eval mode for testing
86      alexnet.eval()
87
88      # obtain the output of the model
89      output = alexnet(batch_img)
90      print("output vector's shape: " + str(output.shape))
91
92      # obtain the activation maps
93      visualize_activation_maps(batch_img, alexnet)
```

From the figure above, line 84 is the only line to load the pre-trained AlexNet and we indicate the loaded model by the variable alexnet. By using this line to get the pre-trained AlexNet, we do not need to define the structure and the main program of AlexNet as what we did for LeNet5.

Similar as before, line 86 is to put the model to evaluation mode in which no gradient will be calculated in order to save the memory usage.

Line 89 is to obtain the prediction of the model by feeding our pre-processed testing image batch_img into alexnet. Line 90 prints the size of the output of the model. Line 93 generates 5 activation maps in your current working directory, Lab2, in which the 5 maps give ideas about what features are being highly activated during the classification. Note that there are 5 convolution layers in a standard AlexNet, hence there are 5 activation maps for observations, from basic edge features to abstract complex features.

Result interpretation and plotting:

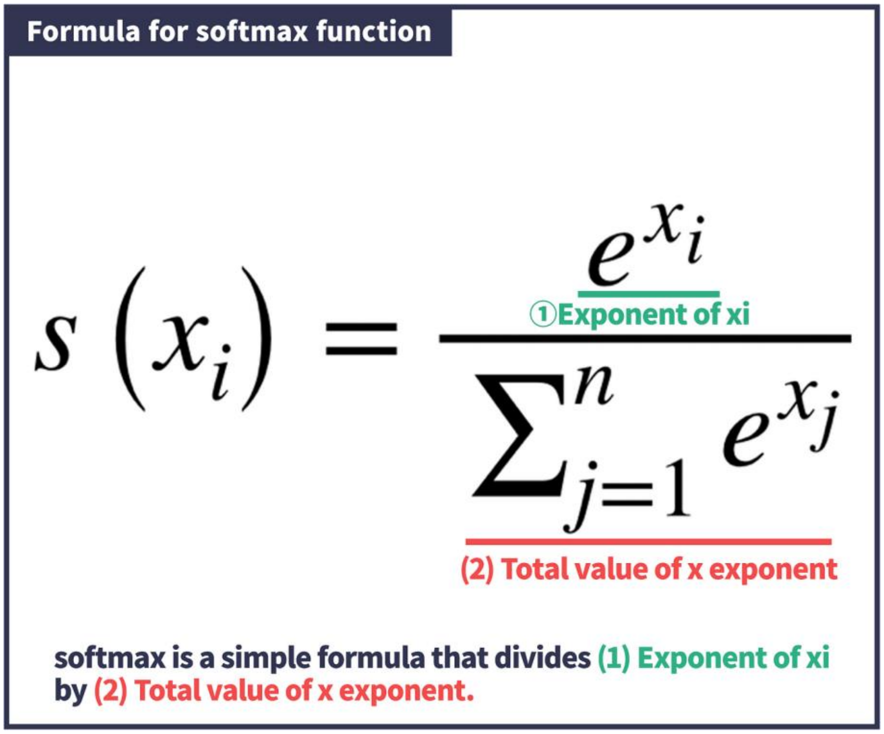
After getting the prediction results, we have to interpret them and display them in a readable format as follows.

```
# map the class no. to the corresponding label
with open('class_names_ImageNet.txt') as labels: # open text file to read the class label
    classes = [i.strip() for i in labels.readlines()] # each line of the text file is one class label
```

```
# print the first 5 classes to see the labels
print("\nprint the first 5 classes to see the lables")
for i in range(5): # create a loop, i from 0 to 4 (5 times)
    print("class " + str(i) + ": " + str(classes[i]))

# sort the probability vector in descending order
sorted, indices = torch.sort(output, descending=True) # sort the probability vector in descending order
percentage = F.softmax(output, dim=1)[0] * 100.0  # convert output to percentage using softmax
```

Note: A brief introduction to the *softmax* function

In the line above we utilize a softmax function in order to convert our output vector from raw scores into probabilities, that sum up to 1. It works by raising each input to the power of *e* and then normalising by dividing all exponentiated values by the sum of ALL the exponential values. Its formula looks like this:



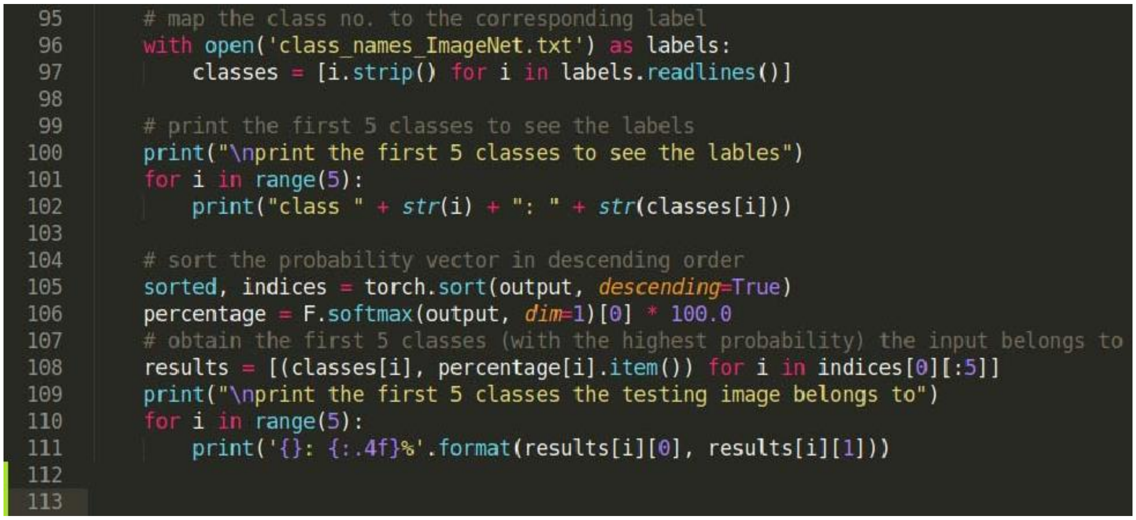
$$s\left(x_i\right) = \frac{\underbrace{e^{x_i}}_{①}}{\underbrace{\sum_{j=1}^{n} e^{x_j}}_{(2)}}$$

softmax is a simple formula that divides (1) Exponent of xi by (2) Total value of x exponent.

Ref: https://medium.com/@sue_nlp/what-is-the-softmax-function-used-in-deep-learning-illustrated-in-an-easy-to-understand-way-8b937fe13d49

It’s often used in the final layer of neural networks for multi-class classification problems.

```
# obtain the first 5 classes (with the highest probability) the input belongs to
results = [(classes[i], percentage[i].item()) for i in indices[0][:5]] # get the 5 predictions with highest prob.
print("\nprint the first 5 classes the testing image belongs to")
for i in range(5): # create a loop, i from 0 to 4 (5 times)
print('{}: {:.4f}%'.format(results[i][0], results[i][1])) # print the top-5 predicted class labels
```



```
95      # map the class no. to the corresponding label
96      with open('class_names_ImageNet.txt') as labels:
97          classes = [i.strip() for i in labels.readlines()]
98
99      # print the first 5 classes to see the labels
100     print("\nprint the first 5 classes to see the lables")
101     for i in range(5):
102         print("class " + str(i) + ": " + str(classes[i]))
103
104     # sort the probability vector in descending order
105     sorted, indices = torch.sort(output, descending=True)
106     percentage = F.softmax(output, dim=1)[0] * 100.0
107     # obtain the first 5 classes (with the highest probability) the input belongs to
108     results = [(classes[i], percentage[i].item()) for i in indices[0][:5]]
109     print("\nprint the first 5 classes the testing image belongs to")
110     for i in range(5):
111         print('{}: {:.4f}%'.format(results[i][0], results[i][1]))
112
113
```

Lines 96-97 map the class numbers back to the corresponding class names. We have provided the class name file called ‘class_names_ImageNet.txt’. The class names are pointed by the variable classes. Lines 100-102 print the first 5 classes to see the labels. This is to verify that the class name file is successfully read.

Line 105 sorts the outputs of the model based on the values in a descending order. The one with the highest value will be the top 1 of predicted labels for your evaluation.

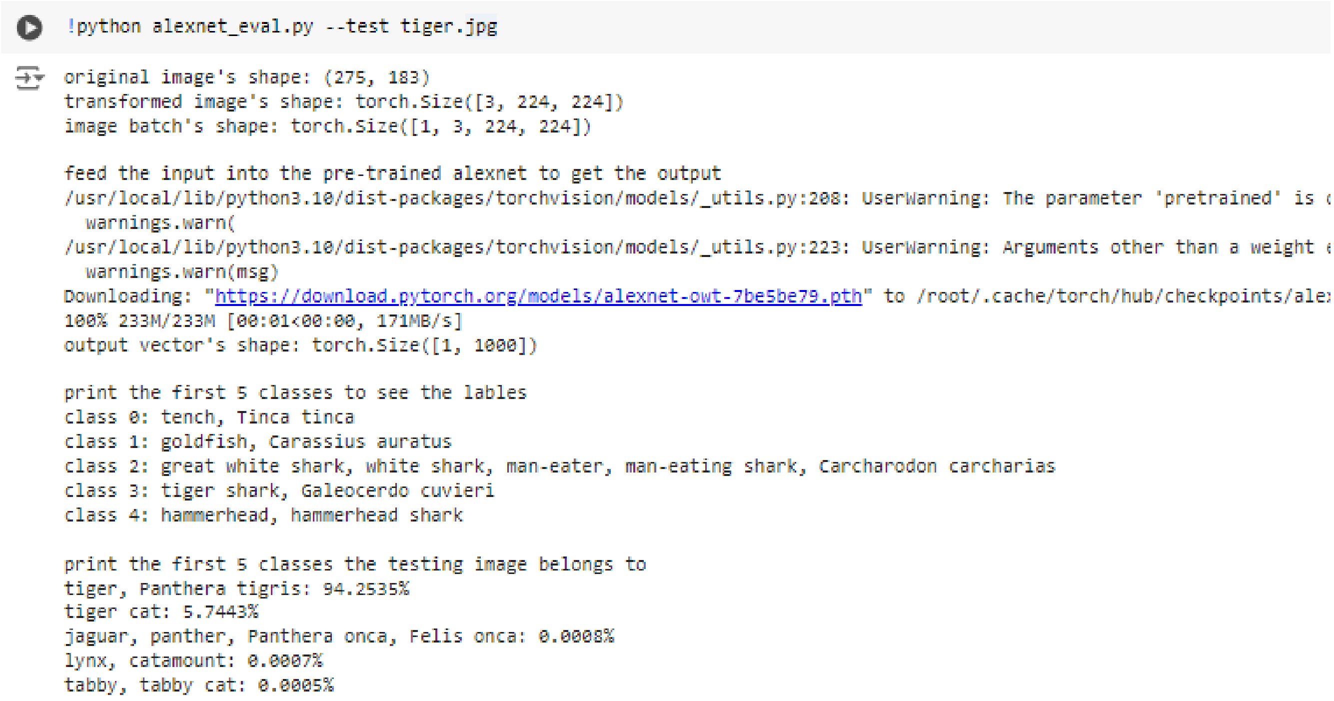
Line 106 performs a softmax function to normalize the output from 0 to 1 and convert the output into a form of percentage. After the softmax operation, the sum of all the output values should equal 1 and actually the output vector becomes a probability vector. Each element in aprobability vector is the probability that the input belongs to the corresponding class.

Lines 108-111 show the top 5 predicted labels of the input testing image for evaluation.

Try running the following command:

```
!python alexnet_eval.py --test tiger.jpg
```

You should see the following:



```
!python alexnet_eval.py --test tiger.jpg
```

```
original image's shape: (275, 183)
transformed image's shape: torch.Size([3, 224, 224])
image batch's shape: torch.Size([1, 3, 224, 224])

feed the input into the pre-trained alexnet to get the output
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is 
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight 
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/alexnet-owt-7be5be79.pth" to /root/.cache/torch/hub/checkpoints/ale
100% 233M/233M [00:01<00:00, 171MB/s]
output vector's shape: torch.Size([1, 1000])

print the first 5 classes to see the lables
class 0: tench, Tinca tinca
class 1: goldfish, Carassius auratus
class 2: great white shark, white shark, man-eater, man-eating shark, Carcharodon carcharias
class 3: tiger shark, Galeocerdo cuvieri
class 4: hammerhead, hammerhead shark

print the first 5 classes the testing image belongs to
tiger, Panthera tigris: 94.2535%
tiger cat: 5.7443%
jaguar, panther, Panthera onca, Felis onca: 0.0008%
lynx, catamount: 0.0007%
tabby, tabby cat: 0.0005%
```

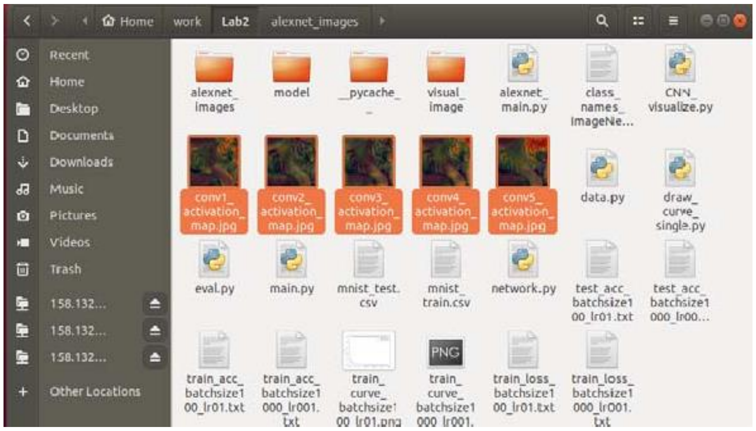
You can see that the size of the original testing image is 275×183 and the size of the input to the pre-trained model is 1×3×224×224. The output size of the model is 1×1000 which is a 1000-class image classification task.

From the prediction results of the pre-trained AlexNet, we are 94.2535% confident that the input belongs to the class, tiger, Panthera tigris. The input testing image is shown in below.

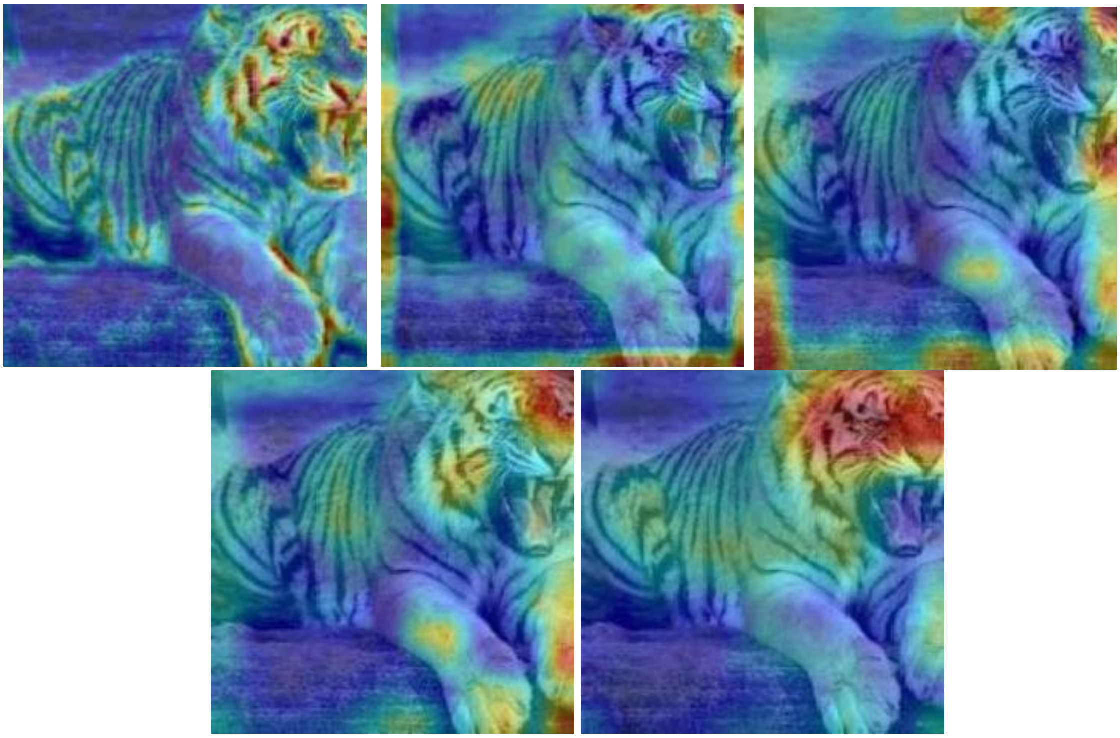
As you can see the network has given us a correct answer!

As mentioned before, apart from the predicted labels of the input testing image, we also save 5 activation maps to visualize the information passed among the model.

Please check your current working directory, you should see there are 5 newly generated images like this.



Open these 5 images, you should see the following:



In order these are convolution activation maps 1-5. We can see that simple edge features are being highly activated at convolutional layer 1. At convolutional layer 5, the head of the tiger is highly activated. This means that this part of information in this image is important to the classification. You may also observe that the extracted features are more and more abstract for the later convolutional layers.

Exercise 4:

Please try other images provided or you can download an image from the internet to test the pre-trained AlexNet by modifying the input parameter --test_img. Please write down you observation(s).

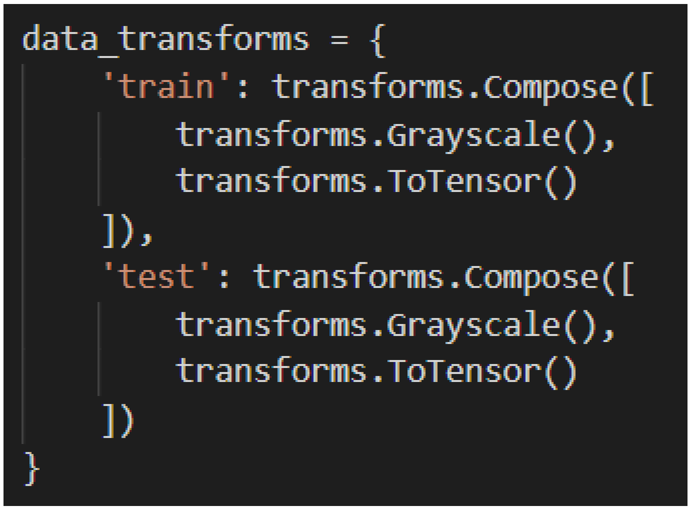
5. Training and testing of AlexNet for 15-Class Object Classification

After directly testing a pre-trained AlexNet, we will now try to train it in the same way we trained LeNet previously. The only difference is the data preprocessing, since LeNet accepts gray-scale 28x28 images as input while AlexNet accepts RGB 224x224 images as input.

We will be working with the Caltech15 dataset. This is a 15-class object classification task dataset. The folder structure is the same as the MNIST we introduced before.

- datasets
  - Caltech15
    - test
    - train
      - bear
      - cake
      - camel

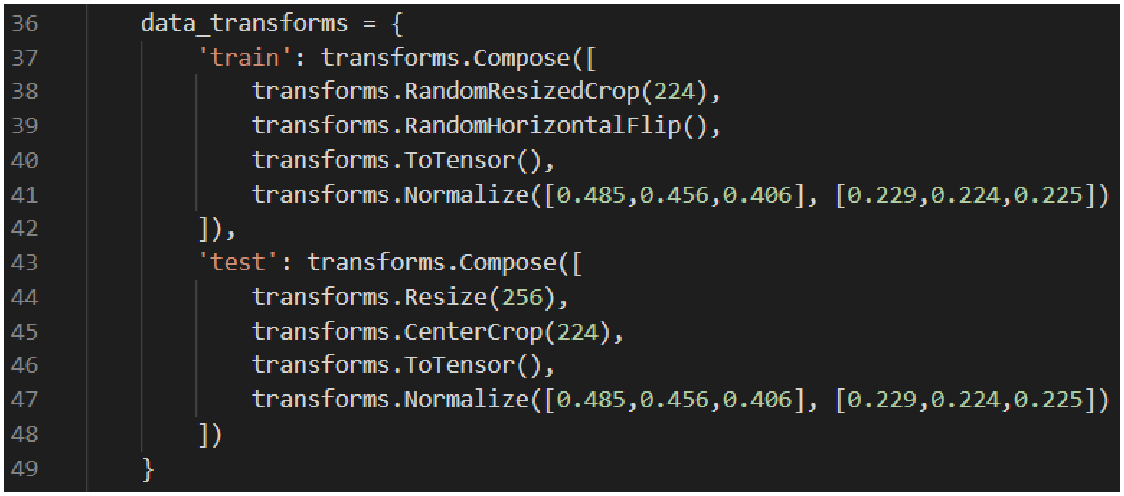
This is the data_transforms for LeNet5 (data pre-processing for LeNet5), students can refer to lenet_main.py for details.



```
data_transforms = {
    'train': transforms.Compose([
        transforms.Grayscale(),
        transforms.ToTensor()
    ]),
    'test': transforms.Compose([
        transforms.Grayscale(),
        transforms.ToTensor()
    ])
}
```

```
data_transforms = {
    'train': transforms.Compose([ # define pre-processing for the train images (training)
      for LeNet5transforms.Grayscale(),  # convert the image to grey-scale first

      transforms.ToTensor()              # put the train data to tensor format

    ]),

    'test': transforms.Compose([        # define pre-processing for the test images (testing)
      for LeNet5transforms.Grayscale(),  # convert the image to grey-scale first
      transforms.ToTensor()              # put the test data to tensor format

    ])
  }
```

This is the data_transforms for AlexNet (for data pre-processing for AlexNet, refer to alexnet_main.py). You can see that the size of the images is different from that of LeNet5 and we perform some data augmentation to increase the number of training images.



```
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485,0.456,0.406], [0.229,0.224,0.225])
    ]),
    'test': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485,0.456,0.406], [0.229,0.224,0.225])
    ])
}
```

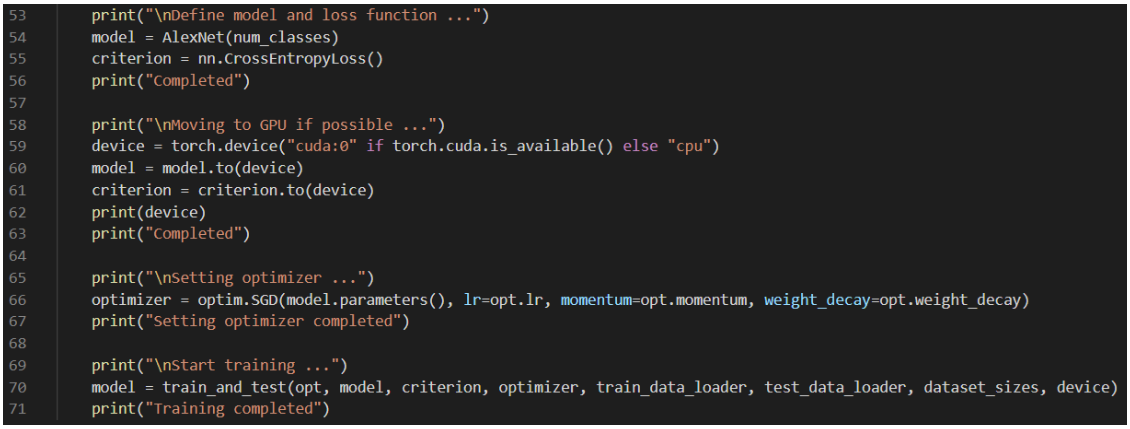
```
data_transforms = {
  'train': transforms.Compose([               # define pre-processing for the train images (training)
    for AlexNettransforms.RandomResizedCrop(224),# resize and crop the input size of train
    images to 224x224, transforms.RandomHorizontalFlip(), # data augmentation, randomly
    flip the input images transforms.ToTensor(),          # put the data to tensor format
    transforms.Normalize([0.485,0.456,0.406], [0.229,0.224,0.225])# normalize the input data

  ]),

  'test': transforms.Compose([      # define pre-processing for the test images (testing)
    for AlexNettransforms.Resize(256),# resize the input size of test images
    transforms.CenterCrop(224),# crop the resized images to size of 224x224
    transforms.ToTensor(),          # put the data to tensor format
    transforms.Normalize([0.485,0.456,0.406], [0.229,0.224,0.225])# normalize the
    input data

  ])
 }
```

Below is a function to get the AlexNet model for this part (please refer to alexnet.py). Originally, AlexNet is trained for 1000-class object classification. Here, we modify the AlexNet to perform a 15-class object classification by modifing the last fully-connected layer.

```
import torch                          # import torch library for nn
import torch.nn as nn                 # import torch.nn library, name it as nn

import torchvision                    # import torchvision

import torchvision.models as models # import torchvision.models, name it models, for pretrained
model
```

```
def AlexNet(num_classes):              # define a function called AlexNet, to obtain a pre-trained
AlexNet from torchvision.models library
  model = models.alexnet(pretrained=True)# get the pre-trained model from torchvision.models,
  name it model

  for param in model.features.parameters():# create a loop, loop the layers in the AlexNet
    param.requires_grad = False # turn off the parameter update attribute of the convolution
    layers of
AlexNet. we do a fine-tuning in this lab, just train the last fully-connected layers

  in_feats = model.classifier[-1].in_features # create a pointer (in_feats) which point to the
in_channels of the last layer
  model.classifier[-1] = nn.Linear(in_feats, num_classes)   # modify the out_channles to
        num_classes,# we do a 15-class classification in this lab.

  return model   # return the modified model for further usage in the main program
```

The definition of the training of AlexNet in this part is written inside alexnet_main.py:



```
    print("\nDefine model and loss function ...")
    model = AlexNet(num_classes)
    criterion = nn.CrossEntropyLoss()
    print("Completed")

    print("\nMoving to GPU if possible ...")
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    model = model.to(device)
    criterion = criterion.to(device)
    print(device)
    print("Completed")

    print("\nSetting optimizer ...")
    optimizer = optim.SGD(model.parameters(), lr=opt.lr, momentum=opt.momentum, weight_decay=opt.weight_decay)
    print("Setting optimizer completed")

    print("\nStart training ...")
    model = train_and_test(opt, model, criterion, optimizer, train_data_loader, test_data_loader, dataset_sizes, device)
    print("Training completed")
```

As you can see, it is the **same as the LeNet** one.

```
print("\nDefine model and loss function ...")        # print a string, define model and loss function
model = AlexNet(num_classes)                 # call AlexNet function to get the model, name it as
modelcriterion = nn.CrossEntropyLoss()       # define criterion, CrossEntropyLoss (typical
classification loss)
print("Completed")

print("\nMoving to GPU if possible ...")
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")# check GPU is available, if
so, define device
model = model.to(device)                              # put the model to device (cuda for GPU, cpu
for CPU)
```

```
criterion = criterion.to(device)                    # also put the loss function to device
print(device)
print("Completed")

print("\nSetting optimizer ...")
optimizer = optim.SGD(model.parameters(), lr=opt.lr, momentum=opt.momentum,
weight_decay=opt.weight_decay)
        # define the optimizer to be used for updating parameters, we use typical Stochastic
Gradient Descentin this lab
print("Setting optimizer completed")

print("\nStart training ...")
model = train_and_test(opt, model, criterion, optimizer, train_data_loader, test_data_loader,
dataset_sizes, device)

        # run the train_and_test function for training and validation.
print("Training completed")
```

To run the file, type:

```
!python alexnet_main.py --lr 0.01 --batch_size 64
```

You should see the following:

```
[10] !python alexnet_main.py --lr 0.01 --batch_size 64

Start to load train and test data
/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:558: UserWarning: This DataLoader will create 4 worker processes in total. Our suggested max number of worker in current system is 2, which is smaller than what this Da
  warnings.warn(_create_warning_msg(
The total number of training and testing images: {'train': 1200, 'test': 300}
The total number of classes: 15
Class names: ['bear', 'cake', 'camel', 'chimp', 'cormorant', 'elephant', 'frog', 'goat', 'gorilla', 'horse', 'owl', 'penguin', 'people', 'raccoon', 'snake']

Define model and loss function ...
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivale
  warnings.warn(msg)
Completed

Moving to GPU if possible ...
cuda:0
Completed

Setting optimizer ...
Setting optimizer completed

Start training ...
Epoch 1/30, Train loss: 1.4895, Train acc: 0.5508
Epoch 1/30, Test loss: 1.2818, Test acc: 0.7000
Epoch 2/30, Train loss: 0.8439, Train acc: 0.7225
Epoch 2/30, Test loss: 1.0411, Test acc: 0.7300
Epoch 3/30, Train loss: 0.6199, Train acc: 0.8042
Epoch 3/30, Test loss: 0.9540, Test acc: 0.7067
Epoch 4/30, Train loss: 0.5409, Train acc: 0.8283
Epoch 4/30, Test loss: 1.0119, Test acc: 0.7167
Epoch 5/30, Train loss: 0.4378, Train acc: 0.8608
Epoch 5/30, Test loss: 1.0789, Test acc: 0.6967
Epoch 6/30, Train loss: 0.4452, Train acc: 0.8500
Epoch 6/30, Test loss: 0.9305, Test acc: 0.7100
Epoch 7/30, Train loss: 0.4174, Train acc: 0.8600
Epoch 7/30, Test loss: 0.9506, Test acc: 0.7300
Epoch 8/30, Train loss: 0.3779, Train acc: 0.8808
Epoch 8/30, Test loss: 0.9989, Test acc: 0.7267
Epoch 9/30, Train loss: 0.3578, Train acc: 0.8808
Epoch 9/30, Test loss: 1.0176, Test acc: 0.7300
Epoch 10/30, Train loss: 0.3794, Train acc: 0.8792
```

As you can see, this time we have 15 classes, namely ‘bear’, ‘cake’, …, ‘snake’.

The training process is running, and it takes several minutes. When the training process is completed. The trained model is saved inside the ‘checkpoints’ folder. There should be a csvfile for storing the training statistics and two figures show the loss and accuracy curves as follows.

Exercise 5:

Try different hyper-parameters.

Please try different 2 different learning rates (0.01, 0.001) and 3 different batch sizes (4, 32, 64) by typing:

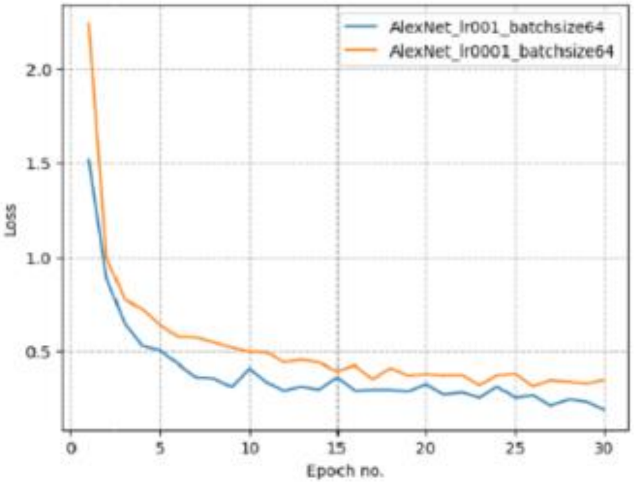
```
!python alexnet_main.py --lr 0.001 --batch_size 64
```

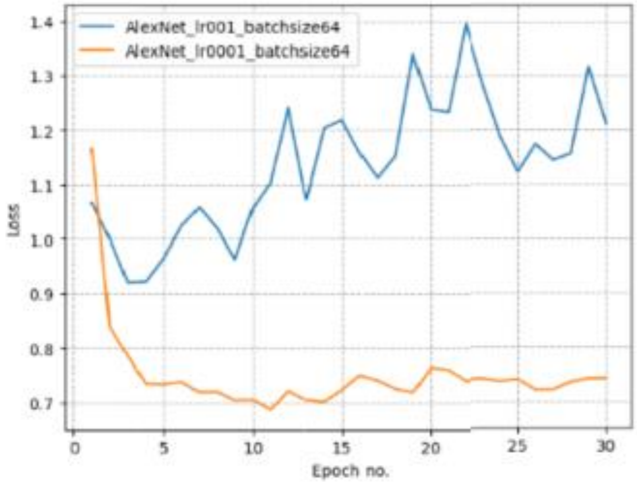
Note that the parameters bold in red should be changed accordingly. You should have 6 sets of training results (2-by-3 combinations). Now, we will plot the curves together to see the effects of different hyper-parameters on the training process. The codes are inside the plot_multi_curves.py. (same as previous part)

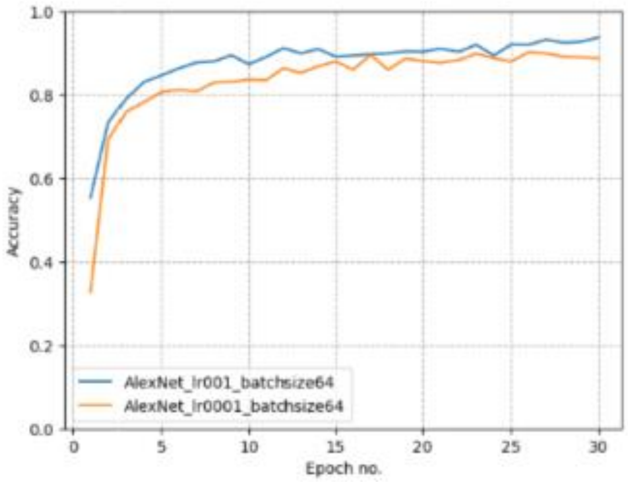
Please type:

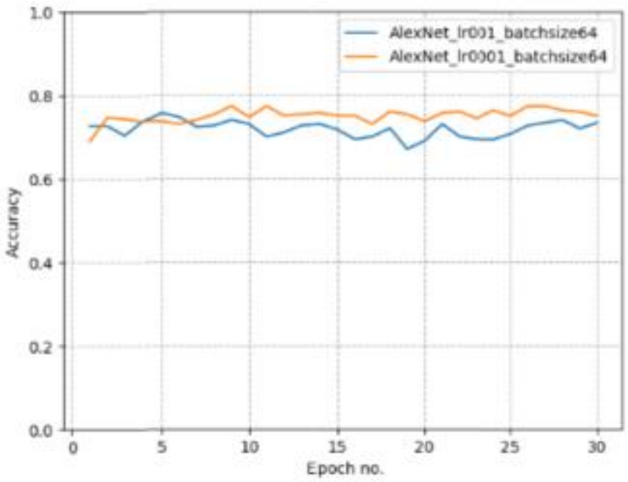
```
!python plot_multi_curves.py - model_name AlexNet
```

You should observe that there are 4 new figures in your current working directory as follows.









Left-top: train loss;

Right-top: test loss;

Left bottom: train accuracy

Right-bottom: test accuracy

Students can try to observe the effects of different learning rates and batch sizes on thetraining process.

(this is just an example, there should be 6 lines for each graph as we let students try 6combinations)

Exercise 5:

Now finally, we will try testing our newly trained model.

In order to do this we need to make some modifications to the alexnet_eval.py file before running it again in the same way we did previously with the pretrained model.

The things we need to address include:

1.Instantiate the model, but this time we need to set ‘pretrained = False’ since we are using our own training. We change the line below:

```
alexnet = models.alexnet(pretrained=True)
```

to this:

```
alexnet = models.alexnet(pretrained=False)
```

2. When we trained the model, what we did was fine tune the final layer. We also made the change that the model has an output size of only 15, compared to the 1000 that the pretrained model has. In order to use our new training we need to make sure the classification layer matches these new dimensions. To do this we add the following line of code:

```
# Modify the classifier to match the one used during training
alexnet.classifier[6] = torch.nn.Linear(alexnet.classifier[6].in_features, 15)
```

3. Additionally, we need to load the model weights from the saved checkpoint of the training process. We will use the last epoch (30) but you can experiment with others. During training we saved the trained models state dictionary, so we can load this using the following line of code:

```
print("\nfeed the input into the updated trained model to get the output")
alexnet.load_state_dict(torch.load('/YOUR/PATH/TO/checkpoints/AlexNet_epoch_30.pth'))
```

4. Finally, we need to consider the mapping of the classes to their labels. In the following section of the code in alexnet_eval.py:

```
    # map the class no. to the corresponding label
    with open('class_names_ImageNet.txt') as labels:
    classes = [i.strip() for i in labels.readlines()]
```

We see that the classes are mapped using the original class names list which contains 1000 classes. Since we trained our network ourselves with 15 classes, we need to modify this class mapping, or we will get nonsensical results.

One easy way to go about this is to create a new class_names_15.txt file for the new mapping, and call this one instead.

Your task is to make the outlined changes to your alexnet_eval.py file and then run the execution line:

```
!python alexnet_eval.py --test_img frog.png
```

to test our newly trained model.

- How does it perform?
  - Can you find ways to tweak the model and improve the performance?

IF you're stuck, you can find the updated code in the file named alexnet_eval_trained.py, and try running that instead. We have also created a file named class_names_15.txt with the correct mapping, but try first to get it right on your own.

Exercise 7:

One additional feature of this updated alexnet_eval_trained.py code is that we added a short section for handling images with an extra RGB channel:

```
# Convert image to RGB if it has an alpha channel
 if img.mode == 'RGBA':
     img = img.convert('RGB')
 elif img.mode != 'RGB':
     raise ValueError("Image is not in RGB or RGBA format")
```

This code uses img.convert to make sure that our input image follows the models expected input data format.

Now try and test the newly trained network on your own image from the internet.

To do this you just need to upload an image in png or jpg format to your google drive folder in the following working directory: .../summerschool_files/tutorial1/Exp2_4/alexnet_images, and then edit the command:

```
!python alexnet_eval_trained.py --test_img YOUR_IMAGE.png
```

to call whatever you have saved your image as.