Lab 2: Neural Network

Learning outcome: TensorFlow Neural Networks and STM Cube AI deployment

# Introduction

## Lab overview

## Welcome to the second laboratory of this course. In this lab, we will build the first neural network model with TensorFlow and deploy the model on the embedded board with STM Cube AI. We will implement a letter recognition model which takes accelerometer data from the board and predicts the letter based on the accelerometer data. For example, if you draw the letter [O] with the board, then the model will predict which letter you drew.

## Requirements

A Windows 10 PC set up with the Anaconda environment of lab1and internet connection.

# Getting Set-up

# In this lab, we are going to use TensorFlow to train multi-layer perceptron models and use STM Cube AI to deploy the model on the board. Let's install necessary conda packages for data collection and machine learning. Note that tensorflow 2.3.0 was used in the video.

## Anaconda

1. First open Anaconda Prompt

2. Activate your environment by typing:

conda activate ml\_lab

2. Add conda-forge channel to install packages:

conda config --add channels conda-forge

3. Then install python packages:

conda install jupyter pandas pyserial scikit-learn tensorflow matplotlib

## Prepare the data collection

Next, you need to program the board to acquire accelerometer data. We already implemented the data collection code for the lab, so you can just import the code and program the board on STM Code IDE.

1. Open STM Code IDE. You can download this software from [STMicroelectronics official website](https://www.st.com/en/development-tools/stm32cubeide.html).

2. Click [Import project] and select the MCU Dataset Creation folder inside the lab2 folder.

3. Plug the board into your computer with a USB cable. You may need to reset your board before you program the board. Click the black button on the board for the reset.

4. Right click the project and select [Run As]. Then it compiles and installs the code into your board.

5. Now your board is ready to collect the sensor data.

## Open Jupyter Notebook

Now we are ready to open our notebook.

1. In the same environment, navigate to the lab2 folder:

cd Documents/lab2

2. Open Jupyter notebook by typing:

jupyter notebook

# Application Code

In this lab exercise you will train your first neural network model with TensorFlow and deploy the inference with STM Cube AI. Second, you will implement the activity recognition application using multi-layer perceptron. Third, you will compare the model accuracy with and without feature extraction and then evaluate the performance on an STM32 microcontroller.

First, import necessary packages by executing the first code block.

|  |
| --- |
| **import** serial.tools.list\_ports  **import** sklearn  **import** tensorflow **as** tf  **import** numpy **as** np  **import** pandas **as** pd  **import** matplotlib.pyplot **as** plt  **import** os, random  base\_dir = os.getcwd()  samples\_dir = os.path.join(base\_dir, 'Samples') |

### Connect the board

Next we are going to connect the board for data collection. Make sure that you plugged the board into your computer via the *USB ST-LINK* port, not the *USB OTG* port. Execute this code block and check which port the board is connected to. In our case the board is connected to COM3 port. Enter the port number.

|  |
| --- |
| print('Com ports list:')  comPorts = list(serial.tools.list\_ports.comports())  **for** comPort **in** comPorts:  **print**(comPort)  chooseComPort = **input**('Please insert port number: ')  ser = serial.Serial('COM{}'.format(chooseComPort), 115200) |

Now we have the serial connection with the board. This code block contains several helper functions for computation. Execute the code block to have these functions.

|  |
| --- |
| **def** convert\_to\_list(value):  value = value.replace("b' ", "")  vals = value.split(", ")  **del** vals[-1]  results = list(map(int, vals))  **return** results  **def** convert\_list\_to\_df(lst):  x = lst[0::3]  y = lst[1::3]  z = lst[2::3]  df = pd.DataFrame({'X': x, 'Y': y, 'Z': z})  **return** df |

Acquire the dataset

Now we are ready to acquire the dataset. Execute this code block. This code block will acquire sensor data from the board and save training samples for machine learning.

1. Enter a lower case letter which you want to collect samples for. This is going to be the label of the samples.

2. Then draw the letter with the board and press the blue button on the board.

3. Enter 1 in the box to save the accelerometer data.

4. You may repeat this process until you get enough samples. The more data samples you collect, the better model you will probably get.

5. Enter 2 when you want to finish the acquisition.

6. You can execute this code block again for other letters. So try collecting the accelerometer data for other letters such as A and U.

|  |
| --- |
| letter = input('Please insert letter to collect data: ')  stride = 30  f = os.path.join(samples\_dir, 'letter\_{}\_stride\_{}.csv'.format(letter, stride))  **if** os.path.exists(f):  print('File exists and data will be appended...')  xyz\_df = pd.read\_csv(f)  **else**:  print('New sample, starting blank...')  xyz\_df = pd.DataFrame(columns=['X', 'Y', 'Z'])  **while** input('1 - acquire sample, 2 - exit: ') == '1':  line = ser.readline()  lineList = convert\_to\_list(str(line))  new\_df = convert\_list\_to\_df(lineList)  print('New data acquired:\n', new\_df.describe())  xyz\_df = pd.concat([xyz\_df, new\_df], ignore\_index=True)  print('Total Data count:', int(xyz\_df.shape[0]/stride))  **print**('Saving data to:', f)  **print**('Total data of sample {}:\n'.format(letter), xyz\_df.describe())  xyz\_df.to\_csv(f, index=False) |

### Load the dataset

Let's check the dataset you collected. Execute this code block to load the dataset. In my dataset, We used the letter o and letter s. The label for letter o is 0 and the label for letter s is 1.

|  |
| --- |
| data\_files = [file **for** file **in** os.listdir(samples\_dir) **if** '.csv' **in** file]  stride = 30  data = []  labels = []  **for** idx, file **in** enumerate(data\_files):  df = pd.read\_csv(os.path.join(samples\_dir, file))  x = df['X'].to\_numpy()  y = df['Y'].to\_numpy()  z = df['Z'].to\_numpy()    **for** i **in** range(int(df.shape[0]/stride)):  base\_idx = i \* stride  batch = np.array([x[base\_idx:base\_idx+stride], y[base\_idx:base\_idx+stride], z[base\_idx:base\_idx+stride]])  batch = batch.reshape((3, stride))  data.append(batch)  labels.append(idx)    **print**('Added {} data to the data list with label: {}'.format(file, idx)) |

This code block will show one training sample. This is the accelerometer data of one training sample.

|  |
| --- |
| **def** plot\_single\_sample(data\_sample, label='Not Specified'):  plt.clf()  scaling = 2\*\*10 # The STM ADC is 10bit so scale to get [g]  fig, axs = plt.subplots(3)  t = np.linspace(0, data\_sample.shape[1] \* 100, data\_sample.shape[1])  # Accelerometer sampled with 100 ms  axs[0].set\_title(label='Single Data Sample of Label {}'.format(label))  axs[0].plot(t, data\_sample[0]/scaling, c='m')  axs[0].set\_ylabel('X [g]')  plt.setp(axs[0].get\_xticklabels(), visible=False)  axs[1].plot(t, data\_sample[1]/scaling, c='m')  axs[1].set\_ylabel('Y [g]')  plt.setp(axs[1].get\_xticklabels(), visible=False)  axs[2].plot(t, data\_sample[2]/scaling, c='m')  axs[2].set\_ylabel('Z [g]')  plt.xlabel('Time [ms]')  plt.show()    idx = random.randint(0, len(data)-1)  plot\_single\_sample(data\_sample=data[idx], label=labels[idx]) |

**Expected Output:**

A picture containing chart

Description automatically generated

Create a model

Now we are going to train a multi-layer perceptron model with the dataset. First, we define a multi-layer perceptron model with 3 dense layers.

|  |
| --- |
| x\_train, y\_train = sklearn.utils.shuffle(np.array(data), np.array(labels))  y\_train = tf.keras.utils.to\_categorical(y\_train, len(np.unique(y\_train)))  model = tf.keras.Sequential(  [  tf.keras.Input(shape=(3, stride)),  tf.keras.layers.Flatten(),  tf.keras.layers.Dense(30, activation="relu"),  tf.keras.layers.Dropout(0.5),  tf.keras.layers.Dense(20, activation="relu"),  tf.keras.layers.Dropout(0.5),  tf.keras.layers.Dense(len(np.unique(y\_train)), activation="softmax")  ]  )  model.summary() |

Compile the model. Then you can see the model has 3392 parameters in total. We train the model for 200 epochs. We will explain why we chose 200 for the number of epochs after the training is finished.

|  |
| --- |
| model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])  history = model.fit(x\_train, y\_train, batch\_size=1, epochs=200, validation\_split=0.4)  model.save('raw\_model.h5') |

Let's plot the training and validation accuracy over epoch. You can see that the training and validation accuracy start to converge after around 150 epochs. This means that the 200 epochs are enough to train the model. If we train the model for too many epochs, then the validation accuracy may drop due to overfitting.

Chart, line chart

Description automatically generated

### Learning rate

For the training, we used the default learning rate for the model. But how will the model change with different learning rates? If the learning rate is too high, the model is more likely to overshoot the minima, meaning that the model cannot converge to the minima. On the other hand, if the learning rate is too low, the model reaches the minima too slowly, requiring more training time.

Let's first try a high learning rate. Here, I set the learning rate of the optimizer as 1000. Execute the code block. This graph shows the training and validation loss values over epoch. You can see that the loss values fluctuate a lot so the model has difficulty in reaching the minima.

|  |
| --- |
| # High learning rate (lr = 1000)  model\_hlr = tf.keras.models.clone\_model(model)  optimizer = tf.optimizers.Adam(learning\_rate = 1000)  model\_hlr.compile(loss='categorical\_crossentropy', optimizer=optimizer, metrics=['accuracy'])  history = model\_hlr.fit(x\_train, y\_train, batch\_size=32, epochs=100, validation\_split=0.4)  plt.plot(history.history['loss'], label='loss')  plt.plot(history.history['val\_loss'], label = 'val loss')  plt.xlabel('Epoch')  plt.ylabel('Loss')  plt.ylim([0, 1000])  plt.legend(loc='lower right') |

**Expected output:**

Chart, histogram

Description automatically generated

Then let's try a lower learning rate, which is 0.0001. Execute the code block. The graph shows the training and validation loss values decrease much more slowly. So, it is important to use a proper learning rate in training.

|  |
| --- |
| # Low learning rate (lr = 0.0001)  model\_llr = tf.keras.models.clone\_model(model)  optimizer = tf.optimizers.Adam(learning\_rate = 0.0001)  model\_llr.compile(loss='categorical\_crossentropy', optimizer=optimizer, metrics=['accuracy'])  history = model\_llr.fit(x\_train, y\_train, batch\_size=32, epochs=100, validation\_split=0.4)  plt.plot(history.history['loss'], label='loss')  plt.plot(history.history['val\_loss'], label = 'val loss')  plt.xlabel('Epoch')  plt.ylabel('Loss')  plt.ylim([0, 1000])  plt.legend(loc='lower right') |

**Expected output:**

Graphical user interface, chart, line chart

Description automatically generated

### Test the model

Finally, we got the trained model and it's time to test it. Execute this code block.

1. Press enter when you are ready to draw a letter.

2. Draw a letter and press the blue button on the board.

3. Check whether the prediction is correct or not.

|  |
| --- |
| input('Press Enter once MCU is ready')  line = ser.readline()  lineList = convert\_to\_list(str(line))  new\_df = convert\_list\_to\_df(lineList)  **print**('New data acquired:\n', new\_df.describe())  x = new\_df['X'].to\_numpy()  y = new\_df['Y'].to\_numpy()  z = new\_df['Z'].to\_numpy()  inf\_data = np.array([x, y, z])  plot\_single\_sample(data\_sample=inf\_data.reshape((3, stride)))  # For inference we have to explicitly tell that the data has a batch size of 1  inf\_data = inf\_data.reshape((1, 3, stride))  pred = model.predict(inf\_data)  **print**('Model Prediction: ', np.argmax(pred)) |

### Extract features

Until now, we trained and tested the model with the raw accelerometer data. Now, we are going to extract features from the data and train a model which makes prediction based on the extracted features. Here, we are going to use the mean and standard deviation of each axis as features.

First, we extract the features from the collected dataset and save the features for training. You can check the extracted features with this code block. These are the extracted features from one data sample.

|  |
| --- |
| data\_files = [file **for** file **in** os.listdir(samples\_dir) **if** '.csv' **in** file]  stride = 30  slidingWindowExt = 6  feature\_data = []  feature\_labels = []  **for** idx, file **in** enumerate(data\_files):  df = pd.read\_csv(os.path.join(samples\_dir, file))  x = df['X'].to\_numpy()  y = df['Y'].to\_numpy()  z = df['Z'].to\_numpy()    **for** i **in** range(int(df.shape[0]/stride)):  base\_idx = i \* stride  # Mean feature  x\_mean\_ext = np.array([np.mean(x[i:i + slidingWindowExt]) **for** i **in** range(base\_idx, base\_idx + stride, slidingWindowExt)])  y\_mean\_ext = np.array([np.mean(y[i:i + slidingWindowExt]) **for** i **in** range(base\_idx, base\_idx + stride, slidingWindowExt)])  z\_mean\_ext = np.array([np.mean(z[i:i + slidingWindowExt]) **for** i **in** range(base\_idx, base\_idx + stride, slidingWindowExt)])  # STD feature  x\_std\_ext = np.array([np.std(x[i:i + slidingWindowExt]) **for** i **in** range(base\_idx, base\_idx + stride, slidingWindowExt)])  y\_std\_ext = np.array([np.std(y[i:i + slidingWindowExt]) **for** i **in** range(base\_idx, base\_idx + stride, slidingWindowExt)])  z\_std\_ext = np.array([np.std(z[i:i + slidingWindowExt]) **for** i **in** range(base\_idx, base\_idx + stride, slidingWindowExt)])    batch = np.array([x\_mean\_ext, y\_mean\_ext, z\_mean\_ext, x\_std\_ext, y\_std\_ext, z\_std\_ext])  feature\_data.append(batch)  feature\_labels.append(idx)    **print**('Added {} data to the feature data list with label: {}'.format(file, idx)) |

**Expected output:**

Chart

Description automatically generated

Then, we create a new multi-layer perceptron model for the features. The new model has 1592 parameters because it uses a smaller input than the previous model. Train the model and check the accuracy.

|  |
| --- |
| x\_train, y\_train = sklearn.utils.shuffle(np.array(feature\_data), np.array(feature\_labels))  y\_train = tf.keras.utils.to\_categorical(y\_train, len(np.unique(y\_train)))  data\_shape = x\_train[0].shape  model = tf.keras.Sequential(  [  tf.keras.Input(shape=data\_shape),  tf.keras.layers.Flatten(),  tf.keras.layers.Dense(data\_shape[0] \* data\_shape[1], activation="relu"),  tf.keras.layers.Dropout(0.5),  tf.keras.layers.Dense(20, activation="relu"),  tf.keras.layers.Dense(len(np.unique(y\_train)), activation="softmax")  ]  )  model.summary()  model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])  model.fit(x\_train, y\_train, batch\_size=32, epochs=500, validation\_split=0.4) |

You can see the accuracy improved with the feature extraction. You can test the new model with this code block to check if the model generates better results.

|  |
| --- |
| input('Press Enter once MCU is ready')  line = ser.readline()  lineList = convert\_to\_list(str(line))  new\_df = convert\_list\_to\_df(lineList)  print('New data acquired:\n', new\_df.describe())  x = new\_df['X'].to\_numpy()  y = new\_df['Y'].to\_numpy()  z = new\_df['Z'].to\_numpy()  # Mean feature  x\_mean\_ext = np.array([np.mean(x[i:i + slidingWindowExt]) **for** i **in** range(0, stride, slidingWindowExt)])  y\_mean\_ext = np.array([np.mean(y[i:i + slidingWindowExt]) **for** i **in** range(0, stride, slidingWindowExt)])  z\_mean\_ext = np.array([np.mean(z[i:i + slidingWindowExt]) **for** i **in** range(0, stride, slidingWindowExt)])  # STD feature  x\_std\_ext = np.array([np.std(x[i:i + slidingWindowExt]) **for** i **in** range(0, stride, slidingWindowExt)])  y\_std\_ext = np.array([np.std(y[i:i + slidingWindowExt]) **for** i **in** range(0, stride, slidingWindowExt)])  z\_std\_ext = np.array([np.std(z[i:i + slidingWindowExt]) **for** i **in** range(0, stride, slidingWindowExt)])  inf\_data = np.array([x\_mean\_ext, y\_mean\_ext, z\_mean\_ext, x\_std\_ext, y\_std\_ext, z\_std\_ext])  # For inference we have to explicitly tell that the data has a batch size of 1  plot\_single\_feature\_sample(data\_sample=inf\_data)  inf\_data = inf\_data.reshape((1, data\_shape[0], data\_shape[1]))  pred = model.predict(inf\_data)  **print**('Model Prediction: ', np.argmax(pred)) |

Finally, save the model by executing this code block.

|  |
| --- |
| **with** open('test.npy', 'wb') **as** f:  np.save(f, x\_train)  **with** open('test\_out.npy', 'wb') **as** f:  np.save(f, y\_train)    model.save('feature\_mlp.h5') |

# MCU deployment

Let's deploy the model on the board using STM Cube AI and STM Cube IDE. We are going to first show how to validate the model on the board. STM Cube AI is an extension software of STM Cube MX. It is not installed by default, so we need to first install Cube AI on STM Cube MX. As this step includes a lot of GUI steps, please consolidate the provided lab video.

### Install STM Cube AI

1. Open STM Cube MX.

2. Click [Access to Board Selector]. Find your board and click [Start Project].

3. First we need to configure the project. Go to [Project Manager]. Write the project name and select the project location where the project will be saved.

4. This figure shows the configuration of pinouts. We don't need to use all of them. So, we are going to clear the pinouts. On [Pinout] menu, click [Clear pinouts].

5. Now we are going to install Cube AI and enable it for your project.

6. On [Software Packs] menu, click [Select Components].

7. Click [Install] button. Expand the menu and enable Cube AI. For device application, choose [Validation].

8. You can see that Cube AI is now enabled.

### Validate model on the board

1. On this Cube AI menu, click [Add network]. Here, you can load your model, validation input, and validation output.

2. Click [Analyze], then it will generate a detailed report on the model. You can check the number of parameters, the size of the weights, and the amount of memory to be used.

3. Click [Show graph], then it visualizes your model.

4. Click [Validate on Desktop], then it validates the model with the provided input and output on your desktop.

5. To validate the model on the target, we need to first generate code and install the code on the target.

6. Click [Generate Code], then it automatically generates the code for validating the model on the board.

7. Open STM Cube IDE. On this menu, choose [STM32 Project from an Existing STM32CubeMX Configuration File]. Then navigate to the project folder and select the ioc file. Install the code with [Run As]. Now it is installing the Cube AI application to the board.

8. Go back to STM Cube MX and press [Validate on target].

### Generate Cube AI Application

Now the model is validated, let's implement an application with the model.

1. Open the list of Software Packages. Change the device application as [Application].

2. Click [Generate Code] again, then the project is updated.

3. Open STM Cube IDE. Since we re-generated the code again, we need to delete the existing project and import the project again.

4. Open X-CUBE-AI/App/app\_x-cube-ai.c. This is the auto-generated code by STM Cube MX.

5. Between USER\_CODE\_BEGIN and USER\_CODE\_END, you can write code for your application.

6. There are two main functions that you should fill. The first function is [acquire\_and\_process\_data]. This function is for getting data from sensors and processing the data, for example, extracting features. The second function is [postprocess] for postprocessing the output.

### Demo with the sample application

1. For demo, we already implemented a sample application, so you can test the application with your board.

2. Open any terminal application to see the output from the board. I will use [Tera Term] in this lab. Set up the serial connection. The speed is 115200.

3. Import our application. Click the File tab and click [Import]. Select [Existing project into workspace]. Go to [lab2/Misc] and select [MCU\_Activity\_Recognition].

4. Open X-CUBE-AI/App/app\_x-cube-ai.c. We already filled the functions for data acquisition from the accelerometer sensor and feature extraction. Install the code with [Run As].

5. Now you can see the output on the terminal.

6. Press the blue button on the board, draw a letter, then press the button again.