

Challenge 1: Building a model with different classes from QuickDraw.io by Google using Convolutional Neural Network and deploy the model onto a Flask website to demonstrate.

There are several steps I did to complete this challenge:

1. Evaluate the data

The first thing we always need in a machine learning model is data. We might not have the model but the data is always necessary. So the data I used is in the [Quick, Draw! Dataset by googlecreativelab](#). This dataset is a collection of 50 million drawings across 345 categories, contributed by players from the game Quick, Draw! By Google. The drawings were captured as timestamped vectors, tagged with metadata (what the players were supposed to draw and where are they from). The raw data is in the ndjson format separated by categories, has different keys like key_id, category, whether_they_are_recognized, word, timestamp, even countrycode and a drawing_vector represented by a JSON array. Here is how one drawing might look.

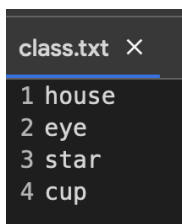
```
{
  "key_id": "5891796615823360",
  "word": "nose",
  "countrycode": "AE",
  "timestamp": "2017-03-01 20:41:36.70725 UTC",
  "recognized": true,
  "drawing": [[[129, 128, 129, 129, 130, 130, 131, 132, 132, 133, 133, 133, ...]]]
}
```

The way they store the drawing vector is very intriguing to me. They do this by storing the coordinates of our drawing strokes and the time since the first point that we draw.

2. Deciding which classes that needed to be trained

[The dataset itself](#), as I mentioned earlier, has 345 different classes and most of them range from very simple to very difficult to draw. I am admittedly not a good drawer(at least on my laptop) so I skimmed through all the classes to find the easiest classes for me to draw.

Our lecturer, Mr. Vũ allows us to choose between 3 and 4 different classes to train so I created a class.txt file and uploaded it to my Google Colab Notebook. I want my model to recognise 4 different objects which are "house", "eye", "cup" and "star". Here is how my class.txt file look on my Google Colab workspace.



```
class.txt X
1 house
2 eye
3 star
4 cup
```

Then I used python to read all the classes in the class.txt files and printed them out to see whether my classes are correctly read.

```
[ ] #reading the classes

f = open("class.txt", "r")

classes = f.readlines()
f.close()

[ ] print(classes)

['house\n', 'eye\n', 'star\n', 'cup\n']

[ ] classes = [c.replace('\n','').replace(' ','_') for c in classes]

[ ] print(classes)

['house', 'eye', 'star', 'cup']
```

After seeing that my classes are correct. I suppose my next step is to download the data. But before doing so, I have to **explore the data**, which is my next step.

3. Explore the data

[According to the documentation of the dataset](#), the raw moderated dataset is in the ndjson format and they provided us with preprocessed dataset on google cloud storage, which I skimmed through to figure which classes I wanted to recognise in step 2. So they provided us with the data in 4 different formats.

- Raw files (ndjson)
- Simplified drawing files (also in ndjson format). This one has way lower size compared to the raw files because it doesn't have the timestamp, and the data is scaled into a 256x256 format.
- Binary files (.bin) for better compression and loading
- Numpy bitmap (.npy). The simplified drawings are transformed into a 28x28 grayscale bitmap in numpy.

Well, I want to use the numpy bitmap format because it is the lightest one. Also it can be used directly by machine learning frameworks like PyTorch and Tensorflow. I chose to use Tensorflow along with Keras for this challenge.

The next step is just downloading the data into a directory called **"data"**, and I used URLLIB to download only the numpy bitmap files for the classes that I defined earlier in step 2.

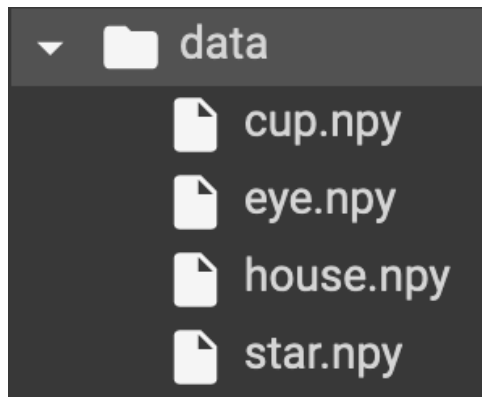
```
[5] #creating a directory to store the dataset
    !mkdir data
```

```
[ ] import urllib.request
    def download():

        base = 'https://storage.googleapis.com/quickdraw_dataset/full/numpy_bitmap/'
        for c in classes:
            cls_url = c.replace('_', '%20')
            path = base+cls_url+'.npy'
            print(path)
            urllib.request.urlretrieve(path, 'data/'+c+'.npy')
```

```
[ ] download()
```

```
https://storage.googleapis.com/quickdraw_dataset/full/numpy_bitmap/house.npy
https://storage.googleapis.com/quickdraw_dataset/full/numpy_bitmap/eye.npy
https://storage.googleapis.com/quickdraw_dataset/full/numpy_bitmap/star.npy
https://storage.googleapis.com/quickdraw_dataset/full/numpy_bitmap/cup.npy
```



4. Load the data

After downloading the data, we have to view the data by loading it. Here in my Colab Notebook, I created a function called `load_data`, which has a 'root' parameter, which takes in the root directory that has the data, in my case it's the **"data"** directory because I downloaded all the .npy files in this folder. I also want to limit the `max_items` trained per class to 4000 to have a smaller amount of data, traded for a much quicker time to train the model and a slightly lower accuracy. The `vfold_ratio` is 0.2, which means we will be using 20% of the data for validation and the other 80% for training.

```
def load_data(root, vfold_ratio=0.2, max_items_per_class= 4000 ):
```

In this function, we have to get all the npy files path from the root path, so I used the `glob`(global) library to load in the npy files from the 'data' directory.

```
#get the list of npy files in the path directory
all_files = glob.glob(os.path.join(root, '*.npy'))
```

I mentioned the drawing is in the 28x28 GRAYSCALE bitmap in numpy so we need an array that can store from 0 to 784, because 28x28 equals 784. So the next step is creating 2 variables for input features(x) and labels(y).

```
#initiate variables for input features(x) and labels(y)
x = np.empty([0, 784]) #store a flattened 28x28 grayscale image
y = np.empty([0]) #store corresponded labels
```

Also we need an array to store my four classes.

```
#creating an array for 4 of our classes
class_names = []
```

After that, I create a loop through iterate through all the numpy files in the directory, and do these steps:

```
#load each data file
for idx, file in enumerate(all_files):
```

- Load the files by using np.load()

```
#just load the data using np.load()
data = np.load(file)
```

- Limit the number of maximum items for class to the defined parameter(which is 4000)

```
#here we limit the number of maximum items for class to the defined parameter(which is 5000)
data = data[0: max_items_per_class, :]
```

- Create the labels for the loaded data(which fundamentally is just assigning a unique index for each class)

```
#create labels for the loaded data(assigning a unique index for each class)
labels = np.full(data.shape[0], idx)
```

- Concatenate the existing data and labels into x and y

```
x = np.concatenate((x, data), axis=0) #Concatenate data along rows
y = np.append(y, labels) #keep appending labels to y array
```

- Extract the class name and extension from path name and add them to the list

```
class_name, ext = os.path.splitext(os.path.basename(file))
class_names.append(class_name)
```

Then we create 2 variables data and labels, then randomize the dataset to have a better generalization of the dataset, ensure that the training and testing dataset are representative of the overall dataset, avoid order bias and last but not least, to avoid overfitting when we train the model.

```
#randomize the dataset
permutation = np.random.permutation(y.shape[0])
x = x[permutation, :]
y = y[permutation]
```

After doing so, we have to split the dataset into the training and testing based on the vfold-ratio which is 0.2 that we defined in the parameter of the function and return them.

```
#separate into training and testing based on vfold ratio
vfold_size = int(x.shape[0]/100*(vfold_ratio*100))

x_test = x[0:vfold_size, :]
y_test = y[0:vfold_size]

x_train = x[vfold_size:x.shape[0], :]
y_train = y[vfold_size:y.shape[0]]
return x_train, y_train, x_test, y_test, class_names
```

Then I just used the function and printed out the length of the train size to see if it's correct

```
#split into train
x_train, y_train, x_test, y_test, class_names = load_data('data')
num_classes = len(class_names)
image_size = 28

print(len(x_train))

12800
```

I have 4 different classes, which is $4 \times 4000 = 16000 \times 80\% = 12800$ for the length of the `x_train`.

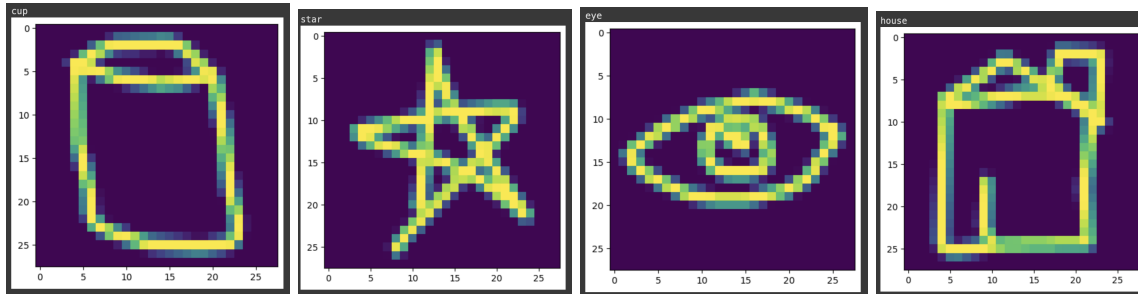
Then I viewed some images of different classes of the dataset by using random and `matplotlib.pyplot`.

```
import matplotlib.pyplot as plt
from random import randint

%matplotlib inline

idx = randint(0, len(x_train))
plt.imshow(x_train[idx].reshape(28,28))
print(class_names[int(y_train[idx].item())])
```

Here are four different images of different classes that I got by running this function.



After seeing that the dataset is indeed loaded correctly, and has the 28x28 size, I decided to move on to the next step, which is preprocessing data.

5. Preprocess data

As I just mentioned before, the image_size after being loaded is 28x28. Despite its small size, we still have to **reshape the data into the correct image size and color scale**.

Here the `x_train.shape[0]` means the number of samples in the training dataset, then `image_size`, `image_size` means that it's a square image with the size 28x28, we also defined the `image_size` as 28 in step 4. The number "1" indicates that the images are grayscale. If we want the images to be RGB, we would write 3 instead of 1. But remember that the dataset is in a 28x28 grayscale image so we want to reshape them into grayscale. `astype('float32')` simply means that we want to work with the image in a floating point number. 32 bit floating point number is just enough to be precise for computation and still saves us memory.

```
#reshape and normalise
#remember we defined image_size = 28
x_train = x_train.reshape(x_train.shape[0], image_size, image_size, 1).astype('float32')
x_test = x_test.reshape(x_test.shape[0], image_size, image_size, 1).astype('float32')
```

After reshaping the data, the next step is normalisation. By dividing all the pixels by 255, which is the maximum pixel value for a 8 bit colour image, it makes the input to be between 0 and 1.

```
x_train /= 255  
x_test /= 255
```

So why do we normalise?

- Ensures numerical stability during computation, especially when we work with gradient-based optimisation methods, such as some deep learning algorithms
- Prevents large and complex values that can rise if the pixels weren't scaled or reshaped down. If the input is small enough, it can help with the optimization algorithm converge faster and more reliably during training

After normalising all the pixels in the image, we have to convert all the labels into one-hot encoded vectors by using `keras.utils.to_categorical()`. Here we convert all the y-test and y-train labels into one hot encoded vector.

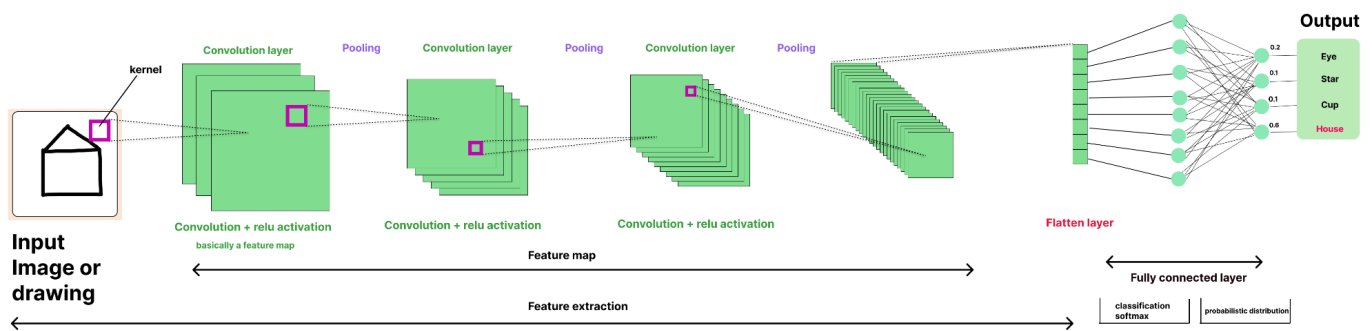
```
# convert class labels into one-hot encoded vector  
# for example, if there are 10 classes, a class label 3 would be converted to [0, 0, 0, 1, 0, 0, 0, 0, 0, 0].  
# 'num_classes' represents the total number of classes in the classification problem.  
  
y_train = keras.utils.to_categorical(y_train, num_classes)  
y_test = keras.utils.to_categorical(y_test, num_classes)
```

After processing the data, we move on to the next step, which is creating the model.

6. Making the model

To make the model, I define a basic Convolutional Neural Network using the Keras library, which is a high level neural networks API that can run on top of Tensorflow.

Here is my own illustration of how the Convolutional Neural Network would work in my model, which has 4 different classes("house", "eye", "cup" and "star").



The first step is to create a Sequential model. This would initialise a sequential model. To make it easier to imagine, it's basically a shelf that has a stack of layers, which we will be adding later.

```
#create a sequential cnn model
model = keras.Sequential()
```

Then we add three convolution layers with 16, 32, and 64 layers respectively. Each layer is followed by a max pooling layer with pool size of (2, 2)(you see the layers get smaller and smaller after pooling). ReLu is used as an activation function for these layers.

```
#adding layers
model.add(layers.Convolution2D(16, (3, 3),
                               padding='same',
                               input_shape=x_train.shape[1:], activation='relu'))

model.add(layers.MaxPooling2D(pool_size=(2, 2)))

model.add(layers.Convolution2D(32, (3, 3),
                               padding='same', activation='relu'))

model.add(layers.MaxPooling2D(pool_size=(2, 2)))

model.add(layers.Convolution2D(64, (3, 3),
                               padding='same', activation='relu'))

model.add(layers.MaxPooling2D(pool_size=(2, 2)))

model.add(layers.Flatten())

model.add(layers.Dense(128, activation='tanh'))
```

I want to go a bit deeper about max pooling. In max pooling, for each region in the input, the max value is taken then gets outputted. For example, in a max pooling (2, 2) region, the max value of that 2x2 region is selected then passed onto the output. For example:

[[1, 3],
[4, 2]] after going through max_pooling(2,2) would give us the max value, which is [[4]].

I also want to dive a bit into the convolution layer. Basically, they are fundamental building blocks in CNN that are used for **feature extraction**. They slide in a small kernel(you see the purple squares in the illustration above) to detect local patterns.

And about the numbers like 16, 32, and 64 that we parse into the function. They are the numbers of filters. They are basically just choices made by the developers depending on the complexity of the problem.

- The first convolution layer with 16 filters captures basic low level features like edges, corners and simple features
- The second layer with 32 filters operates on the feature map generated by the first layer. This layer can gather more complex features and with some of the low-level features
- The third layer with 64 filters works in a bit more abstract features from the previous layers. This layer would learn high-level features and complex features that are composed of lower-level features.

More filters in deeper layers would allow the network to learn more diverse and complicated features which can help with tasks like classification and segmentation.

After adding the convolutional and max pooling layers, the 3D output is flattened into a 1D vector.

```
model.add(layers.Flatten())
```

Then we would add two dense layers. The first dense layer contains 128 neurons and uses the 'tanh' activation function. The final dense layer has 4 neurons which use the 'softmax' activation function which we use for multiclass classification problems. Keep in mind that we have to change the number of neurons according to the number of classes in our model.

```
model.add(layers.Dense(128, activation='tanh'))

# change 4 to the number of classes that you have for the sake of classification
model.add(layers.Dense(4, activation='softmax'))
```

Then we just have to compile the model, here we use the categorical_entropy function, adam optimizer and we evaluate accuracy as our evaluation metric.

```
adam = tf.keras.optimizers.Adam()

model.compile(loss='categorical_crossentropy',
              optimizer=adam,
              metrics=['accuracy'])
```

Adam optimizer (Adaptive Moment Estimation) is an extension of Stochastic Descent(SGD) and it's widely used in deep learning. In the arguments of the compile() function, the loss='categorical_crossentropy' means that the loss function uses categorical cross_entropy, which is commonly used for multiclass classification problems. And more importantly, for a

problem where target variables are one hot encoded vector(see step 5), categorical cross_entropy is suitable.

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 28, 28, 16)	160
max_pooling2d_3 (MaxPooling2D)	(None, 14, 14, 16)	0
conv2d_4 (Conv2D)	(None, 14, 14, 32)	4640
max_pooling2d_4 (MaxPooling2D)	(None, 7, 7, 32)	0
conv2d_5 (Conv2D)	(None, 7, 7, 64)	18496
max_pooling2d_5 (MaxPooling2D)	(None, 3, 3, 64)	0
flatten_1 (Flatten)	(None, 576)	0
dense_2 (Dense)	(None, 128)	73856
dense_3 (Dense)	(None, 3)	387

```
Total params: 97539 (381.01 KB)
Trainable params: 97539 (381.01 KB)
Non-trainable params: 0 (0.00 Byte)
None
```

Here is the output of that function.

7. Training and testing the model

After creating the model, we have to train it... That is the purpose of the model.

```
#training
model.fit(x = x_train, y = y_train, validation_split=0.1, batch_size = 256, verbose=2, epochs=5)
```

I am going to explain the arguments in the function

- `x=x_train` is the input data for the network
- `y=y_train` is the target labels for training data
- `Validation split = 0.1` is the amount of data used for validation checks. In this case we use 10% of the training data for validation *during training*.
- `batch_size=256` is the number of samples will be used for each iteration when training. Using smaller batches can help with lowering training time.
- `epochs=5` is the number of times the entire dataset will be passed forward and backward through the network.

- `verbose=2` means that the training progress will be displayed in a detailed way, showing progress bars,...

I also plot the training and validation loss values and their accuracy values.

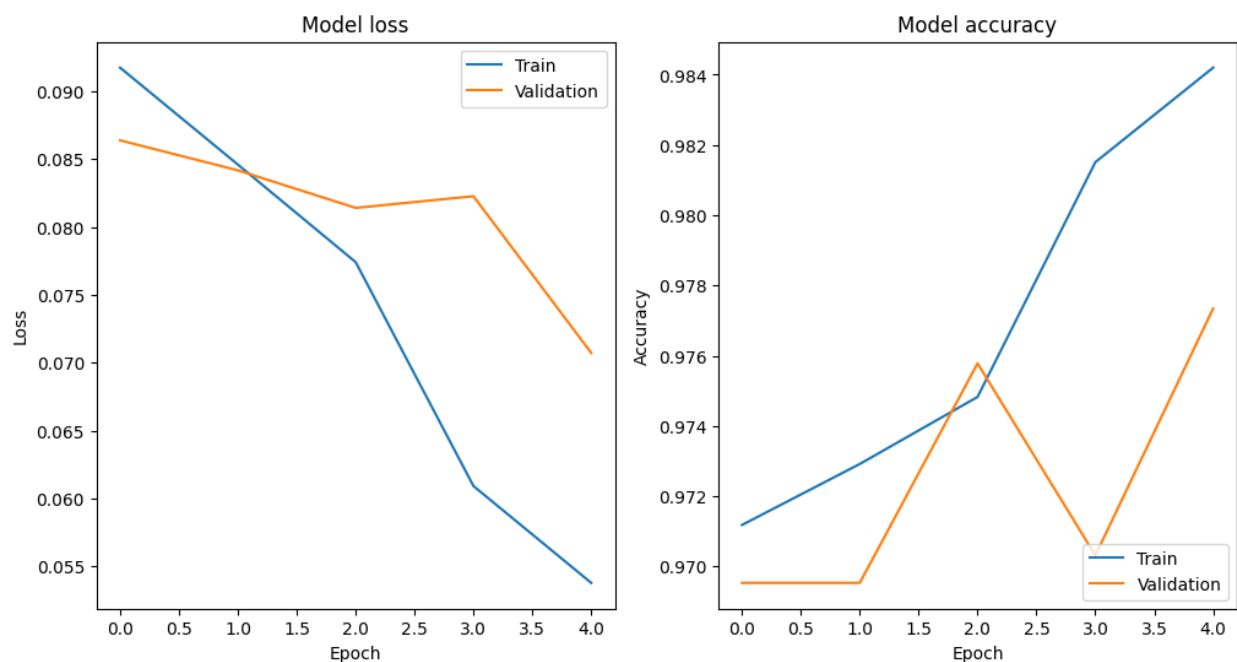
```
# Plot training & validation loss values
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'], loc='upper right')

# Plot training & validation accuracy values
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Train', 'Validation'], loc='lower right')

plt.show()
```

After running the function we would get the training progress and the graphs.

```
Epoch 1/5
34/34 - 11s - loss: 0.5678 - accuracy: 0.7796 - val_loss: 0.3346 - val_accuracy: 0.8521 - 11s/epoch - 321ms/step
Epoch 2/5
34/34 - 0s - loss: 0.2987 - accuracy: 0.8882 - val_loss: 0.2149 - val_accuracy: 0.9187 - 288ms/epoch - 8ms/step
Epoch 3/5
34/34 - 0s - loss: 0.2089 - accuracy: 0.9265 - val_loss: 0.2074 - val_accuracy: 0.9135 - 243ms/epoch - 7ms/step
Epoch 4/5
34/34 - 0s - loss: 0.1686 - accuracy: 0.9391 - val_loss: 0.1370 - val_accuracy: 0.9542 - 256ms/epoch - 8ms/step
Epoch 5/5
34/34 - 0s - loss: 0.1295 - accuracy: 0.9571 - val_loss: 0.1234 - val_accuracy: 0.9563 - 264ms/epoch - 8ms/step
<keras.src.callbacks.History at 0x7a80fc4c38e0>
```



Here is the output of the function

After training the model, the next step is to evaluate it. Here our accuracy is high enough so we don't have to bother with preprocessing the data again or adjusting the arguments of the training function.

```
[ ] score = model.evaluate(x_test, y_test, verbose=0)
    print('Test accuracy: {:.2f}%'.format(score[1] * 100))
```

```
Test accuracy: 95.91%
```

After evaluating it, we have to test the model. Here I use the random library to randomly get an image from the x_test and display them by using matplotlib.pyplot

```
import matplotlib.pyplot as plt
from random import randint
%matplotlib inline
idx = randint(0, len(x_test))
img = x_test[idx]
plt.imshow(img.squeeze())
```

Then I perform inference on the selected image using the model we just trained.

`np.expand_dims` is used to match the expected input shape for the model. We use predict to get the model's prediction for the image, which is the probabilities for different classes.

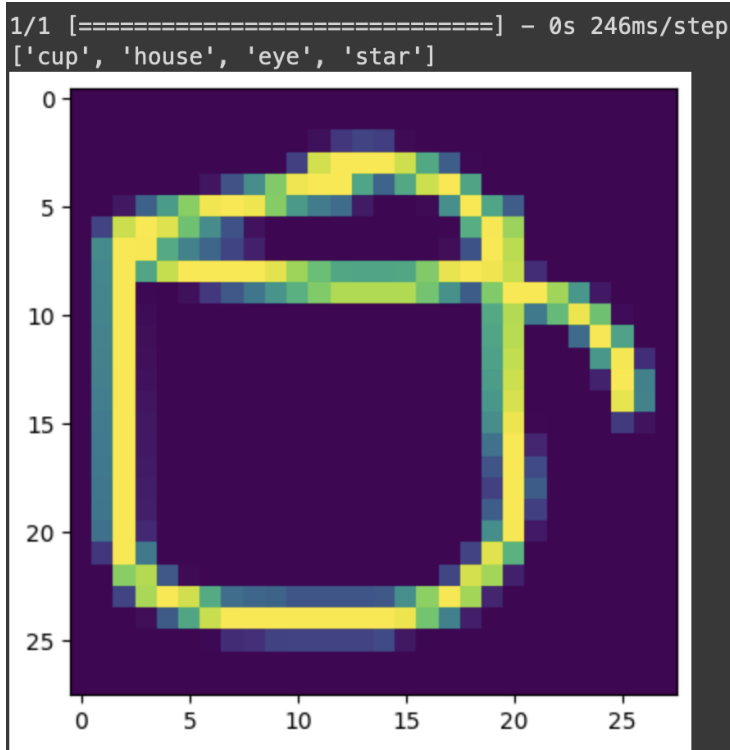
```
pred = model.predict(np.expand_dims(img, axis=0))[0]
```

After that, I sort the probability in descending order and only get the top 5 predictions.

```
ind = (-pred).argsort()[:5]
```

Then I map the top 5 indices to the class_names that we defined earlier using the class.txt file. These indices should match with the classes with the highest predicted probability

```
latex = [class_names[x] for x in ind]
print(latex)
```



Here is the output of the function, you can see an image that looks like a cup and a sorted array that shows the cup has the highest probability, followed by house, eye and star.

8. Downloading the model

Then we have to store the classes and write them into a `class_names.txt` file.

```
#storing classes
with open('class_names.txt', 'w') as file_handler:
    for item in class_names:
        file_handler.write("{}\n".format(item))
```

After that we saved the model.

```
#saving the model and download it into laptop
model.save('keras.h5')

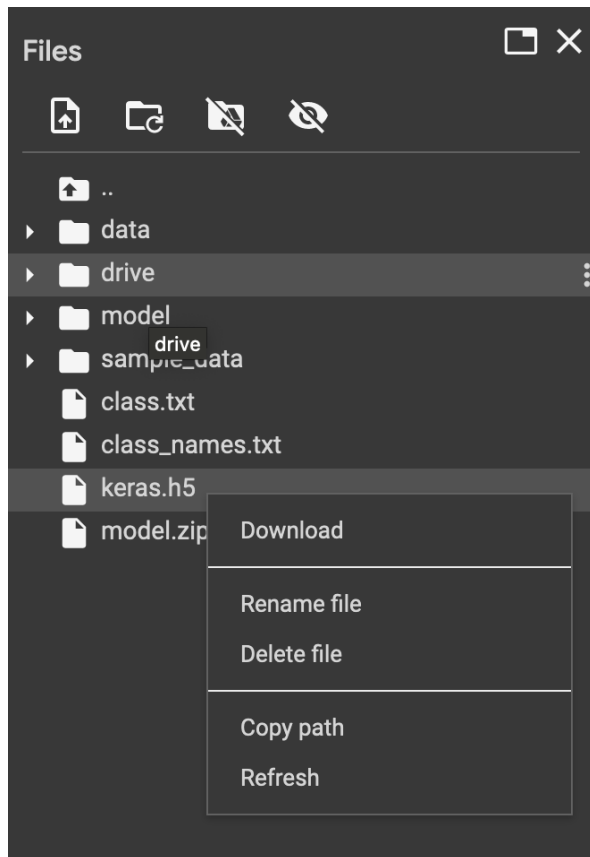
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3079: UserWarning: You are saving your model as an HD
saving_api.save_model()
```

Then create a directory called 'model' and save the model file in a HDF5 format(.h5 extension) and output into the model directory.

```
!mkdir model
!tensorflowjs_converter --input_format keras keras.h5 model/

2023-11-07 05:35:35.998056: E tensorflow/compiler/xla/stream_executor/cuda/cuda_dnn.cc:9342] Unable to register cuDNN fact
2023-11-07 05:35:35.998115: E tensorflow/compiler/xla/stream_executor/cuda/cuda_fft.cc:609] Unable to register cuFFT facto
2023-11-07 05:35:35.998154: E tensorflow/compiler/xla/stream_executor/cuda/cuda_blas.cc:1518] Unable to register cuBLAS fa
2023-11-07 05:35:36.984530: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not find TensorR
```

You can download the keras.h5 directly after this step and stop the notebook because at the end you only need the keras.h5 model.



Then copy the class_names.txt into the model directory and zip them up by using the zip library.

```
#zip and download
!cp class_names.txt model/class_names.txt
```

```
!zip -r model.zip model
```

```
adding: model/ (stored 0%)
adding: model/model.json (deflated 82%)
adding: model/class_names.txt (stored 0%)
adding: model/group1-shard1of1.bin (deflated 8%)
```

The last step is just downloading the model by using files in colab.

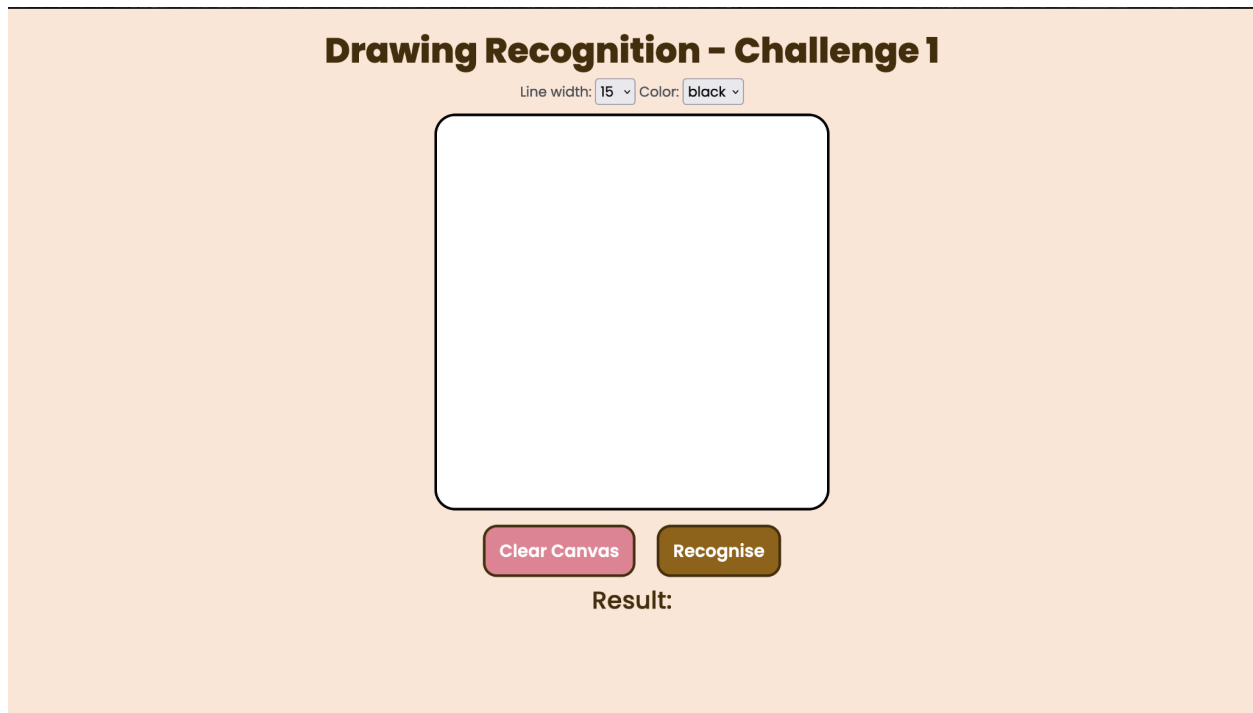
```
from google.colab import files
files.download('model.zip')
```

9. Deploy the model

After downloading the model file into the working directory, the next step is to deploy the model onto a web server. I decided to use Flask as it's simple and fast to program.

9.1. Creating a templates folder for the UI of the app and handling image transfer to Flask server.

I created a simple site with basic UI, which has a canvas that allows users to draw on, a button to clean the canvas, a button to predict the drawing and two options dropdown that allow users to choose the thickness and colour of the stroke.



Here I want to go deeper into the drawing function of the canvas and how the image is sent for processing as it is the main point. Here we create a simple canvas in HTML.

```
<canvas
  id="myCanvas"
  width="450"
  height="450"
  style="border: 3px solid black; margin-top: 10px"
></canvas>
```

Then I created some functions for the canvas. Here I just retrieved the canvas by using `getElementById()`, then I got the 2D rendering context by using `canvas.getContext()`, filling the entire canvas with white colour to create an empty 'white board'. After that I defined some event handling for the canvas: `mousedown`, `mousemove`, `mouseup`, `mouseleave` based on some parameters I defined outside of the `init()` function.

```
var mousePressed = false;
var lastX, lastY;

var ctx;
```

```
function init() {
  canvas = document.getElementById("myCanvas");
  ctx = canvas.getContext("2d");
  ctx.fillStyle = "white";
  ctx.fillRect(0, 0, canvas.width, canvas.height);

  $("#myCanvas").mousedown(function (e) {
    mousePressed = true;
    draw(
      e.pageX - $(this).offset().left,
      e.pageY - $(this).offset().top,
      false
    );
  });
  $("#myCanvas").mousemove(function (e) {
    if (mousePressed) {
      draw(
        e.pageX - $(this).offset().left,
        e.pageY - $(this).offset().top,
        true
      );
    }
  });
  $("#myCanvas").mouseup(function (e) {
    mousePressed = false;
  });
  $("#myCanvas").mouseleave(function (e) {
    mousePressed = false;
  });
}
```

When users draw, the canvas would check if the mouse click is down. If the user is holding their click button down, the stroke color and width would be taken from the option dropdowns, then draw them onto the canvas by setting the coordinate x and y constantly for each stroke. When the user's click button is up, the old x and y coordinate would be marked to end a stroke.


```
function draw(x, y, isDown) {
  if (isDown) {
    ctx.beginPath();

    ctx.strokeStyle = $("#selColor").val();
    ctx.lineWidth = $("#selWidth").val();

    ctx.lineJoin = "round";

    ctx.moveTo(lastX, lastY);
    ctx.lineTo(x, y);
    ctx.closePath();
    ctx.stroke();
  }

  lastX = x;
  lastY = y;
}
```

Recognise

When users submit the drawing by pressing the recognise button. The image would first be removed from the data prefix (`data:image/png;base64`, or `data:image/jpg;base64`), from the data URL, leaving only the base64 decoded image data. Then the image is sent to the flask server using AJAX by using the POST method. If the image is successfully sent, it would expect a prediction value from the server, and update the `<div id="result">` innerHTML to the prediction.

```
function postImage() {
  let canvas = document.getElementById("myCanvas");
  let image = canvas.toDataURL("image/png");

  image = image.replace(/^data:image\/(png|jpg);base64/, "");

  $.ajax({
    type: "POST",
    url: "/recognize",
    data: JSON.stringify({ image: image }),
    contentType: "application/json; charset=UTF-8",
    dataType: "json",
    success: function (msg, status, jqXHR) {
      var data = JSON.parse(jqXHR.responseText);

      var prediction = data.prediction;

      document.getElementById("result").innerHTML = prediction;
    },
  });
}
```

9.2. Using the UI and model in the Flask Server

We create a file called 'app.py' and start working to create a flask server.

The first step is to render to UI, we create a Flask app and render the template we just created above.

```
app = Flask(__name__)  
  
@app.route('/')  
def index():  
    return render_template('/index.html')
```

Before handling the recognize() function, we have to load our model. After downloading the keras.h5 model (see step 6), I renamed it to drawingModelChallenge1.h5 and loaded it into my app.py and used the make_predict_function().

```
model = tf.keras.models.load_model('drawingModelChallenge1.h5')  
model.make_predict_function()
```

Then we loaded the class_names.txt which contains the classes for my model, which are "eye", "house", "star", "cup" into a class_labels array, which we will be using later in the predict function.

```
app.py 5, M    class_names.txt X  
  
model > class_names.txt  
You, 4 days ago | 1 author (You)  
1 eye  
2 house  
3 star  
4 cup
```

```
with open('./model/class_names.txt', 'r') as file:  
    class_labels = file.read().splitlines()
```

After that we work with the `recognize()` function, which will be triggered when we receive a 'POST' request.

```
@app.route('/recognize', methods=['POST'])
def recognize():
```

When we receive a POST request, we will get the image data from the request. The image is in the imagebase64 format so we have to decode it by using the base64 library.

```
#here we receive the image from the web browser
if request.method == "POST":
    print("Receive image and is predicting it")
    #we receive the request and get the image under imagebase64 format and decode it
    data = request.get_json()
    imageBase64 = data['image']
    imgBytes = base64.b64decode(imageBase64)
```

Then we write the image into a temp.jpg image in our working directory.

```
with open("temp.jpg", "wb") as temp:
    temp.write(imgBytes)
image = cv2.imread('temp.jpg')
```

After doing that, we have to remember our input data in the CNN. The image must have the 28x28 size, in GRAYSCALE. So the first thing we have to do is to resize the image to 28x28 to make sure it has the correct size to be put into the predict function of the model later on. `interpolation=cv2.INTER_AREA`` simply means the interpolation method used for resizing.

```
image = cv2.resize(image, (28,28), interpolation = cv2.INTER_AREA)
```

After resizing the image we have to convert them from RGB to GRAYSCALE

```
image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

Then we have to reshape the grey image to have a specific shape of (28, 28, 1), now it's reshaped into a 3D array with dimensions (28, 28, 1). By doing so it can be fitted into the predict function later.

```
image_prediction = np.reshape(image_gray, (28,28,1))
```

After reshaping the image, we have to normalise it. I already explained the reason for normalising above(check step 5) but I will repeat it again. It is simply to ensure the numerical value during computation and to prevent sophisticated values that can make the training process longer.

```
#we also have to normalise the image before parsing into the cnn predict model
image_prediction = (255 - image_prediction.astype('float')) / 255
```

Then we perform prediction by using `model.predict()`, the prediction is an array containing probabilities of all classes and we get the highest class index value by using ``argmax(axis=-1)``.

```
#predicting
prediction = np.argmax(model.predict(np.array([image_prediction])), axis=-1)
```

After getting the index with highest probability, we simply match that index with the `class_labels` array we defined earlier. Then we return the `predicted_class` into the JSON response for the UI to update the `<div id="result">` class

```
#since we loaded class_labels from a text file, class with the highest prediction
#percentage would be returned
#and sent back into the web

predicted_class = class_labels[prediction[0]]
#running predict here
return jsonify({
    'prediction': str(predicted_class),
    'status': True
})
```

When received the response it would update the result div.

```
$.ajax({
  type: "POST",
  url: "/recognize",
  data: JSON.stringify({ image: image }),
  contentType: "application/json; charset=UTF-8",
  dataType: "json",
  success: function (msg, status, jqXHR) {
    var data = JSON.parse(jqXHR.responseText);

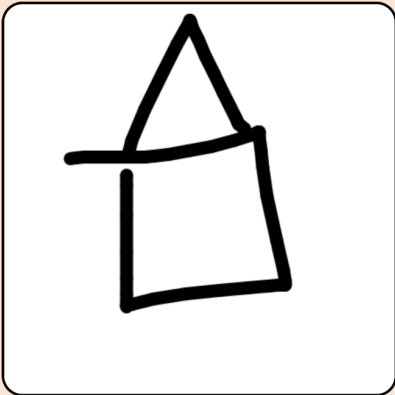
    var prediction = data.prediction;

    document.getElementById("result").innerHTML = prediction;
  },
});
```

That is all of my steps that are needed to complete the challenge.
Here are some of my images of the project.

Drawing Recognition - Challenge 1

Line width: 15 Color: black

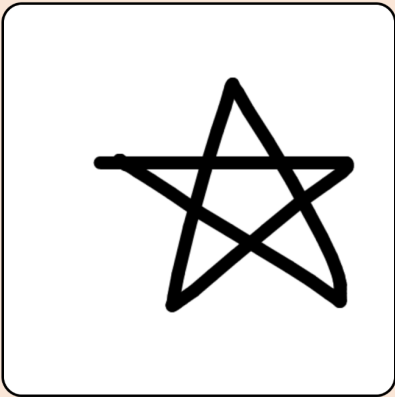


Clear Canvas Recognise

Result: house

Drawing Recognition - Challenge 1

Line width: 15 Color: black

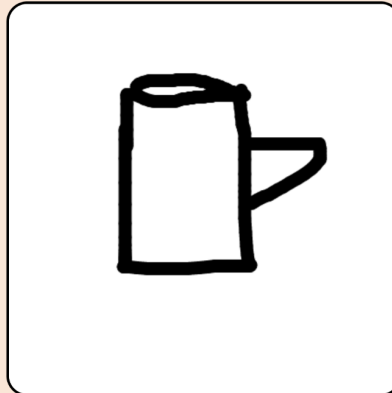


Clear Canvas Recognise

Result: star

Drawing Recognition - Challenge 1

Line width: 15 Color: black



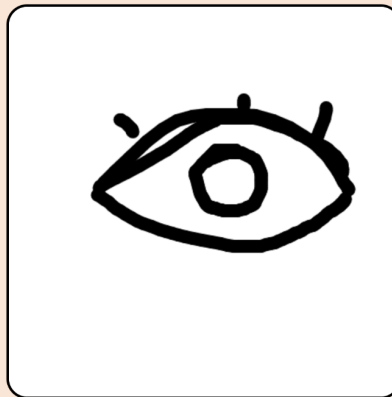
Clear Canvas

Recognise

Result: cup

Drawing Recognition - Challenge 1

Line width: 15 Color: black



Clear Canvas

Recognise

Result: eye

Here is the github repository of the project:

<https://github.com/ngoinhaoto/drawingRecognitionChallenge1>.

The part for my model training is the ipynb file called Challenge_1_Artificial_Intelligence_2023.ipynb and the rest are for deploying model on the Flask server.

To run it, just clone the repository and open it in Visual Studio Code, **install all the libraries**, type `app.py` in the terminal and control click the <http://127.0.0.1:5000> or type `localhost:5000` on your browser.

```
○ (base) ritherthemuncher@Rithers-MacBook-Air drawingRecognitionChallenge1 % python app.py
WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.Adam` runs slowly on M1/M2 Macs, please use the legacy Keras optimizer instead, located at `tf.keras.optimizers.legacy.Adam`.
WARNING:absl:There is a known slowdown when using v2.11+ Keras optimizers on M1/M2 Macs. Falling back to the legacy Keras optimizer, i.e., `tf.keras.optimizers.legacy.Adam`.
WARNING:tensorflow:Error in loading the saved optimizer state. As a result, your model is starting with a freshly initialized optimizer.
WARNING:tensorflow:Error in loading the saved optimizer state. As a result, your model is starting with a freshly initialized optimizer.
['eye', 'house', 'star', 'cup']
* Serving Flask app 'app'
* Debug mode: on
INFO:werkzeug:WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
```