Udacity Nanodegree Capstone
Pictionary

Saad Chaouki

March 2021

1 Introduction

In the final project of the capstone, I created a model that can be used to play Pictionary. The final outcome is an application where a user can draw. As the user is drawing, a model that is hosted on AWS makes predictions to try and classify the drawing. The methodology is a combination of a Neural Network trained and deployed on AWS and a drawing application using Tkinter in Python. The model will be accessed through the use of an API, Lambda function, and an Endpoint. The input of the model will be an array representing a 28x28 drawing of the user. Locally, the user is drawing on an application using Tkinter. Whenever the user releases the mouse, the application takes a screenshot of the drawing and converts it to an array of size 784 representing the 28x28 picture. This is then sent to the model to make the prediction that are displayed on a separate window.

1.1 Problem Statement

The problem that I am investigating is whether we can use Machine Learning to classify drawings of users. In addition to that, I will test the ability of making real-time predictions as the user is using the application to draw. The drawing application will also be created as part of the project using Tkinter.

1.2 Dataset

The dataset I am using for this project is the Quick Draw[1] dataset by Google. This contains a large of number of drawings with their classifications and is stored as *npy* files. I will select a total of 30 drawings a starting point and will use a total of 75,000 image of each class. One important thing about the images is that their size is 28x28. Therefore, I will ensure that the screenshots of the drawings of users are also transformed to the same format. The dataset will be directly downloaded from Google Cloud to AWS using the Google Cloud package in Python. Once this is done, the training and testing sets will be created and hosted on an S3 Bucket.

[1] https://github.com/googlecreativelab/quickdraw-dataset

1.3 Metric

The main evaluation metric I will be using is the accuracy of the model on all the classes. I will ensure that there is no imbalance in the dataset in terms of images of each class.

$$\frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

where: TP = True positive; FP = False positive; TN = True negative; FN = False negative

Additional testing will also be completed by drawing using the application.

1.4 Design

The Neural Network will be created using PyTorch and will have an input shape of 784 corresponding to the size of an 28x28 array and the output will correspond to the number of classes (30). The model will be trained and deployed on AWS Sagemaker. To access the predictions from the application, I will create a Lambda function and API.

From the application perspective, I will create a simple drawing application using Tkinter. This will allow the user to start drawing. Whenever the user releases the mouse, a screenshot will be taken and transformed to a 28x28 image. This will then be transformed to a 784 array before it is sent as text to the model to make the prediction. The received prediction will then be displayed on a separate window.

2 Data Analysis & Preparation

The first step of the project is to download the data from Google Cloud and store it locally. This will allow me to start processing it and combining multiple classes into one file. To achieve this, I'm using the Google Cloud package to link to the *quick-draw_dataset* bucket and download the classes that I need only. The files are stored as *npy* files that represent Numpy arrays of the images.

To ensure this process is easy, I'm creating an array with the names of the files that I want to download and then use in the model. These are included in a loop to download. If all the files have previously been downloaded, this does not run.

2.1 Data Processing

Because the data downloaded is stored as a 1D array, I am not able to directly visualise it using MatplotLib. Therefore, the first step to be able to visualise the images is to reshape them from a 784 1D array to a 28*28 2D array. This will allow me to visualise the data but also to have a better understanding of the required format for the drawing application.To achieve this, I'm using the reshape function in *Numpy*.

The data is loaded and then directly added to a dictionary that will have the name of the drawing (class) as the key and an array of drawings as the values.

2.2 Data Visualisation

After transforming the image to 2D arrays, we can start visualising them using imshow in Matplotlib. I start by selecting a single image to visualise.

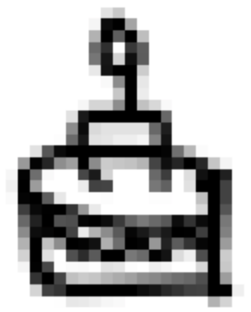


Figure 1: Single Drawing of a Cake

After being able to visualise one drawing, I can start exploring more about the data and draw a grid of images.

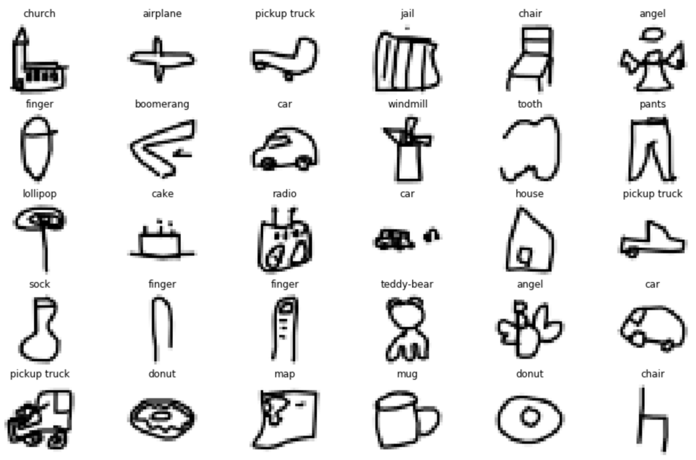


Figure 2: Multiple Drawings Grid

From an input perspective, the values in the array are between 0 and 255 where 0 represents white and 255 represents black. This is important to take into consideration as the screenshots taken from the application will need to match the exact data that was used to train the model. I will present the transformation done to the screenshot in the application section.

2.3 Training and Testing Set Creation

After understanding and visualising the data I am working on, the next step is to create the training and testing sets that will be used to train the model. To achieve this, I start by combining all the arrays of different drawings and then shuffle them. A 70-30 split is then used to create the training and testing set before these are uploaded to S3 and used to create data loaders.

3 Implementation

The implementation part will go through 2 main categories. The first category is the training of the model as well as the functions that are needed to be able to call it using an API. The second stage is linked to the creation of the drawing application using Tkinter.

3.1 Modelling

The modelling part starts by creating the model that will be used using PyTorch. I will then go through the process of training, deploying, and then creating the Lambda function to connect the model to the drawing application.

3.1.1 Model Training

The selected model is a simple mutli-layer neural network with a dropout layer and batch normalisation. The model structure can be found in the appendices.

The test the hyper-parameters and the model structure, I start by training the model using a subset of the data without a training job. Once I am happy with the structure, I create a complete training job that uses the full dataset.

3.1.2 Model Performance

The final performance achieved by the model is 86% on the testing dataset and 94% on the training dataset. This is better than the target I set of 80%.

3.1.3 Model Deployment

To deploy the model, I create an endpoint using custom input and output functions that will allow me to send an array of values representing the drawing and then use them to make the predictions. This category will display the loading, input, output, and lambda functions used as well as the use of API gateway.

Model Loading Function The model loading function allows me to specify how I want the created model to be loaded. The exact function can be found in the appendices.

Input Function The input function is a very simple function that de-serialises the input and transforms it to an array that will be sent to the predict function.

```python
def input_fn(serialized_input_data, content_type):
    print('Deserializing the input data.')
    try:
        decoded = serialized_input_data.decode('utf-8')
        data = np.array(decoded.split(',')).astype(int)
        return data
    except:
        raise Exception('Failed to decode the string. Please send a
        ↪  plain CSV string.')
```

Output Function The output function transform the output of the model to a string before sending it back to the Lambda Function.

```python
def output_fn(prediction_output, accept):
    print('Serializing the generated output.')
    return str(prediction_output)
```

Predict Function The predict function is the function that does most of the work when it comes to calling the model. It starts by transforming the input to a tensor and then using the forward function of the model to make the prediction. The index of the highest value is then returned as the predicted class.

Lambda Function The lambda function takes as an input an array representing the picture. This is then used to call the endpoint that will make the predictions. Once this is completed, a dictionary is returned with the body as the predicted class. The complete Lambda function can be found in the appendices.

API Gateway To be able to call the Lambda function that will then call the endpoint, I'm creating an API using the API gateway service on AWS. This API has a single POST method that is linked to the Lambda function.

3.2 Drawing Application

This section will go through the creation of the application and how it is connected to the model.

3.2.1 Application

The drawing application was completely created using the Tkinter package on Python. This is a very simple application that can only be used to draw. An additional window was added to display the predictions. The main action that the application does on the background is to take screenshots and transform the image whenever the user stops drawing.

Drawing Action Whenever the user releases the mouse, a reset function is called that will take a screenshot of the drawing. As soon as the screenshot is taken, it's transformed and sent to the model for predictions.

Screenshot The screenshot function starts by identifying the position of the drawing app. This will allow the user to be able to move the drawing window. Once this is done, the image is processed to match with the input of the model.

Transformation After taking the screenshot, the values of the image are 255 for white and 0 for black. As we analysed the input of the model, we know that this has to be reversed. Therefore, the screenshot is directly inverted so that 0 corresponds to white and 255 to black. After that, 2D matrix is selected from the 3D arrays that represent the RGB colours. As we draw in black, any array will work. An issue identified at this stage is that the values do not always reach 255 that the model expects. Because of this, the images start looking fade. To solve this issue, I start by applying a log transformation on the image. This will ensure that the image is always clear and that drawing additional dark lines will not reduce the colour of the previous lines. Once this is done, is it scaled to values between 0 and 255 before it is transformed to a string of comma separated values.

3.2.2 Model API Access

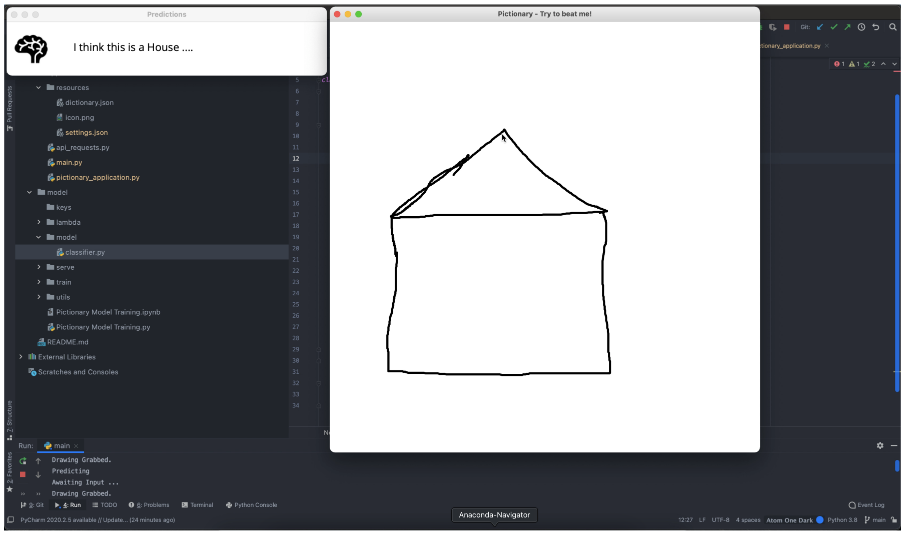
Once the final image is created and transformed, the information is sent to the API. I'm doing this by creating an object RequestsAPI that contains information such as the link of the API. The class is as follow:

```python
import requests
import json

class RequestsAPI():
    def __init__(self):
        settings = json.load(open('resources/settings.json'))
        self.url = settings['url_api']

    def request_prediction(self, body):
        requestBody = {"body": body}
        result = requests.post(self.url, data=requestBody)
        return result.text
```

The class is called from the application by calling the request_prediction function and sending the transformed image as a string.

4 Results and Conclusion

The final result of this project is a drawing application that is connected to a model making predictions on these drawings. A screenshot of the final application is displayed below:
A recording of the application and the model making predictions can also be found on my Github.



I think this is a House ....

```
Drawing Grabbed.
Predicting
Awaiting Input ...
Drawing Grabbed.
```

PyCharm 2020.2.5 available // Update... (24 minutes ago)

Figure 3: Screenshot of the Final Application

This has been extremely useful not to just learn about the way models are created and deployed on AWS, but also to see how a model starts making live predictions. This gives the possibility to understand how a model works and differentiate between different drawings. An example that is interesting is the difference between a house and a church. While these 2 drawings can be very similar, the difference between a house and a church for a model is a cross that is on top of the house.

4.1 Improvement & Next Steps

I will continue improving the model to make it work even better and on more drawings. The main improvement is to use a pre-trained model and build a custom classifier on top. I am expecting this to work really well. The second improvement is on increasing the number of images that the model can recognise. Finally, I want to train the model using the time of the drawing as well. Because of the way the model was created, it will always assume that the drawing it received is the final one. I want to improve this by including the time of that drawing. For example, if the user draws wheels first, the model will understand that this will be a car.

Appendices

Neural Network

```python
class NeuralNetClassifier(nn.Module):
    def __init__(self, inputSize, hiddenSize, outputSize, dropout = 0.0):
        super(NeuralNetClassifier, self).__init__()

        self.classifier = nn.Sequential(OrderedDict([

                      ('linear1', nn.Linear(inputSize, hiddenSize)),
                      ('relu1', nn.ReLU()),

                      ('linear2', nn.Linear(hiddenSize, hiddenSize)),
                      ('batchnorm1', nn.BatchNorm1d(num_features=hiddenSize)),
                      ('relu2', nn.ReLU()),

                      ('dropout1', nn.Dropout(dropout)),

                      ('linear3', nn.Linear(hiddenSize, hiddenSize)),
                      ('relu3', nn.ReLU()),

                      ('linear4', nn.Linear(hiddenSize, hiddenSize)),
                      ('batchnorm2', nn.BatchNorm1d(num_features=hiddenSize)),
                      ('relu4', nn.ReLU()),

                      ('linear5', nn.Linear(hiddenSize, outputSize))
                ]))

    def forward(self, x):
        x = self.classifier(x)
        return x
```

Lambda Function

```python
import boto3
from urllib.parse import unquote

def lambda_handler(event, context):

    runtime = boto3.Session().client('sagemaker-runtime')
    response = runtime.invoke_endpoint(EndpointName =
    ↪  'sagemaker-pytorch-2021-03-07-16-24-54-682',
                                       ContentType = 'text/plain',
                                       Body =
                                       ↪  unquote(event['body'][5:]))
    result = response['Body'].read().decode('utf-8')

    return {
        'statusCode' : 200,
        'headers' : { 'Content-Type' : 'text/plain',
        ↪  'Access-Control-Allow-Origin' : '*' },
        'body' : result
    }
```

Model Loading Function

```python
def model_fn(model_dir):
    """Load the PyTorch model from the `model_dir` directory."""
    print("Loading model.")

    # First, load the parameters used to create the model.
    model_info = {}
    model_info_path = os.path.join(model_dir, 'model_info.pth')
    with open(model_info_path, 'rb') as f:
        model_info = torch.load(f)

    print("model_info: {}".format(model_info))

    # Determine the device and construct the model.
    device = torch.device("cuda" if torch.cuda.is_available() else
    ↪  "cpu")
    model = NeuralNetClassifier(model_info['input_size'],
                                model_info['hidden_size'],
                                model_info['output_size'],
                                model_info['dropout'])

    # Load the stored model parameters.
    model_path = os.path.join(model_dir, 'model.pth')
    with open(model_path, 'rb') as f:
        model.load_state_dict(torch.load(f))

    model.to(device).eval()

    print("Done loading model.")
    return model
```