

Big Data Laboratory

Assignment 5 Deployment of ML model using Fast Api

by

Anik Bhowmick
AE20B102

Instructor: Sudarshan
Teaching Assistant: Jashaswimalya Acharjee



Contents

1	Introduction	1
1.1	The problem statement	1
2	Model pipeline	2
3	Result and conclusion	3
A	Code section	9

1

Introduction

As part of MLOps, this assignment aims to deploy the ML model on a web server using Fast Api. FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.8+ based on standard Python type hints. As a part of this assignment, we had to perform versioning of our project using git hub, to have GitHub access to all the files **click here**.

1.1. The problem statement

We have to set up a web interface to deploy a ML model trained on digit classification task. For this Fast Api has to be used to create the API, and Swagger UI for the webserver.

Task 1 (Creation of FastAPI module)

1. Create a FastAPI module.
2. Take the path of the model as a command line argument.
3. Create a function “def load_model(path: str) -> Sequential,” which will load the model saved at the supply path on the disk and return the `keras.src.engine.sequential.Sequential` model.
4. Create a function “def predict_digit(model: Sequential, data_point: list) -> str” that will take the image serialized as an array of 784 elements and returns the predicted digit as string.
5. Create an API endpoint “@app.post('/predict')” that will read the bytes from the uploaded image to create a serialized array of 784 elements. The array shall be sent to the ‘predict_digit’ function to get the digit. The API endpoint should return “digit:digit” back to the client.
6. Test the API via the Swagger UI (<api endpoint>/docs) or Postman, where you will upload the digit as an image (28x28 size).

Task 2 (Resizing the image to accept any image)

1. Create a new function, “def format_image,” which will resize any uploaded images to a 28x28 greyscale image, followed by creating a serialized array of 784 elements.
2. Modify Task 1 to incorporate “format_image” inside the “/predict” endpoint to preprocess any uploaded content.
3. Now, draw an image of a digit yourself using tools such as “ms-paint” or equivalent using your touch screen or the mouse pointer. Upload your hand-drawn image to your API and find out if your API is able to figure out the digit correctly. Repeat this exercise for 10 such drawings and report the performance of your API/model combo.

2

Model pipeline

As per the instruction, the model chosen for this task was a pre-trained model from the previous mlflow assignment. After proper hyperparameter tuning, the best model was found to have 256 and 128 hidden nodes, with both having sigmoid activation. The model achieved a test accuracy of 0.07 earlier. The model is expected to receive flattened image vectors of dimension 768. So we first have to ensure each image is 28 by 28 and then flatten them into dense 768-dimensional vectors. Below is the small architecture and summary of the number of parameters of the model.

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 256)	200,960
dense_1 (Dense)	(None, 128)	32,896
dense_2 (Dense)	(None, 10)	1,290

Total params: 470,294 (1.79 MB)
Trainable params: 0 (0.00 B)
Non-trainable params: 235,146 (918.54 KB)
Optimizer params: 235,148 (918.55 KB)

Figure 2.1: Model summary

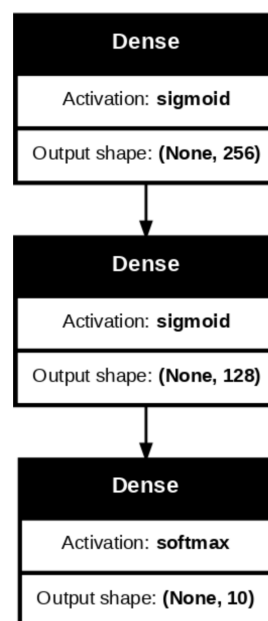


Figure 2.2: Model architecture

3

Result and conclusion

The model performed well on the original test data as shown below



Figure 3.1: Image of 3 (MNIST data)

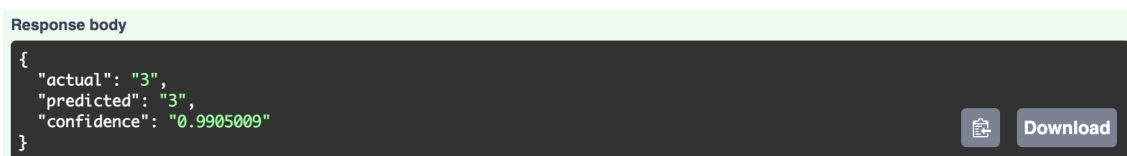


Figure 3.2: Prediction from Swagger UI



Figure 3.3: Image of 9 (MNIST data)

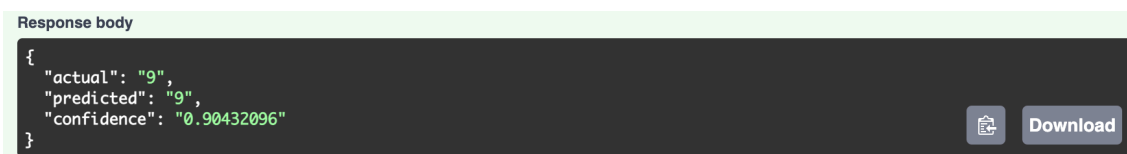


Figure 3.4: Prediction from Swagger UI

For custom handwritten digits:



Figure 3.5: Image of 1

Response body

```
{
  "actual": "1",
  "predicted": "1",
  "confidence": "0.48598436"
}
```


 [Download](#)

Figure 3.6: Prediction from Swagger UI



Figure 3.7: Image of 2

Response body

```
{
  "actual": "2",
  "predicted": "2",
  "confidence": "0.5355249"
}
```


 [Download](#)

Figure 3.8: Prediction from Swagger UI



Figure 3.9: Image of 3

Response body

```
{
  "actual": "3",
  "predicted": "3",
  "confidence": "0.98432577"
}
```



Download

Figure 3.10: Prediction from Swagger UI

Figure 3.11: Image of 4

Response body

```
{
  "actual": "4",
  "predicted": "4",
  "confidence": "0.9767869"
}
```



Download

Figure 3.12: Prediction from Swagger UI

Figure 3.13: Image of 5

Response body

```
{
  "actual": "5",
  "predicted": "5",
  "confidence": "0.9789844"
}
```



Download

Figure 3.14: Prediction from Swagger UI

Figure 3.15: Image of 6

Response body

```
{  
  "actual": "6",  
  "predicted": "6",  
  "confidence": "0.91772485"  
}
```



Download

Figure 3.16: Prediction from Swagger UI



Figure 3.17: Image of 7

Response body

```
{  
  "actual": "7",  
  "predicted": "8",  
  "confidence": "0.5583249"  
}
```



Download

Figure 3.18: Prediction from Swagger UI



Figure 3.19: Image of 8

Response body

```
{  
  "actual": "8",  
  "predicted": "8",  
  "confidence": "0.5882316"  
}
```



Download

Figure 3.20: Prediction from Swagger UI



Figure 3.21: Image of 9

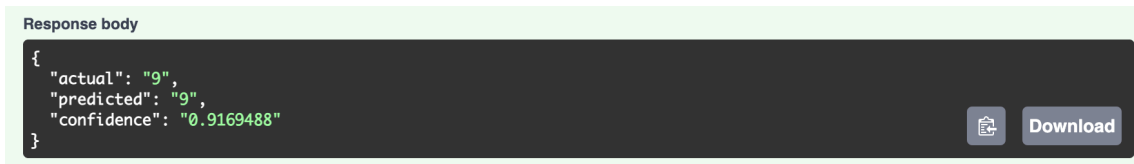


Figure 3.22: Prediction from Swagger UI

We performed some extra tests on slightly different-looking digits (making them thinner)



Figure 3.23: Image of 1 (thinned)

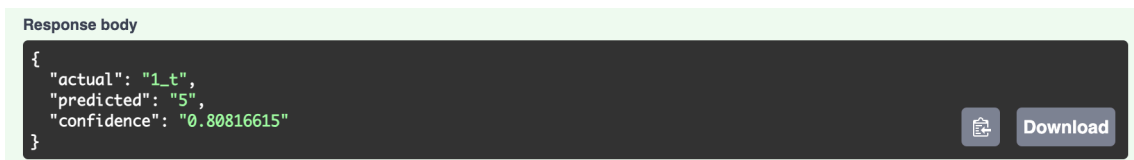


Figure 3.24: Prediction from Swagger UI



Figure 3.25: Image of 2 (thinned)



Figure 3.26: Prediction from Swagger UI



Figure 3.27: Image of 3 (thinned)

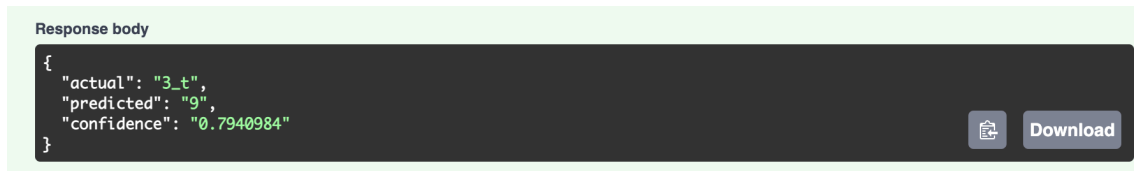
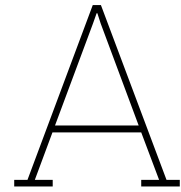


Figure 3.28: Prediction from Swagger UI

So, for all thinner-looking digits, the model performed poorly. Whereas for thick digits except 7, all predictions were correct. The reason for this is that the model is trained completely on the raw data without augmentation. So, the model performance is not that good for even slightly modified data.



Code section

Fast Api code

```
1 import os
2 from typing import List
3 from fastapi import FastAPI, UploadFile, File, HTTPException
4 import numpy as np
5 from tensorflow.keras.models import load_model
6 import tensorflow as tf
7 from PIL import Image, ImageOps
8 from io import BytesIO
9 import sys
10 import base64
11
12 # Create the FastAPI app
13 app = FastAPI()
14
15 # Load the model from the specified path
16 def get_model(path: str):
17     return load_model(path)
18
19 # Load the MNIST model
20
21 model_path = "/Users/anikbhowmick/Python/Big_Data_Assignment/A06/MNIST_model.keras"
22 #load the pretrained model
23 model = get_model(model_path)
24 # set the model in inference mode
25 model.trainable=False
26
27 # Function to preprocess image and make prediction
28
29 def predict_digit(model, data_point):
30
31     # get the prediction containg the score
32     pred = model.predict(data_point)
33     # get the class label
34     prediction = tf.argmax(pred,axis=-1)
35     c_score = np.max(pred)# store the confidence score
36     return str(prediction[0].numpy()),str(c_score)
37
38 # API endpoint to accept image upload and return prediction
39 def format_image(image):
40     """
41     get a pillow image
42     """
43     # resize the image in 28X28 format
44     return image.resize((28,28))
45
46
47 @app.post('/predict')
48 async def predict(file: UploadFile = File(...)):
49     # load the image in the byte format
50     content = await file.read()
51     accepted_formats = ['.jpeg', '.jpg', '.png']
52     file_format = os.path.splitext(file.filename)[1].lower()
53     # check for the image file is valid or not
54     if file_format not in accepted_formats:
```

```

55     # raise the error message if the file is wrong in format
56     raise HTTPException(status_code=400, detail="Bad file format. Accepted formats are .
        jpeg, .jpg, .png")
57     file_name = os.path.splitext(file.filename)[0]
58     image = Image.open(BytesIO(content))
59     #convert the image to gray scale first
60     image = image.convert('L')
61     if image.size!=(28,28):
62         # if the image is not 28 by 28 resize it
63         image = format_image(image)
64     flat = np.array(image,dtype='float32').reshape(-1)/255.0# flatten the image and normalize
        in 0 to 1 scale
65     flat = flat[None,:]
66     output, c_score = predict_digit(model, flat)
67     return {
68         "actual":file_name,
69         "predicted": output,
70         "confidence":c_score}

```

Model training code

```

1  import tensorflow as tf
2  from keras.datasets import mnist
3  from keras.layers import Dense, Input, RandomRotation
4  from keras.models import Sequential
5  from keras.models import Sequential
6  from keras.optimizers import Adam
7  from keras.losses import SparseCategoricalCrossentropy
8  import numpy as np
9  # for training the model with the best hyperparameters of previous assignment
10 def get_architecture():
11     # sequentail definition of the model
12     model = Sequential(
13     [
14         Input(shape=(784,)),
15         Dense(units = 256, activation = 'sigmoid'),
16         Dense(units = 128, activation = 'sigmoid'),
17         Dense(units = 10, activation='softmax')
18     ])
19     return model
20
21 def train_model():
22     model = get_architecture()
23     model.compile(loss=SparseCategoricalCrossentropy(),metrics=['acc'])
24     (x_train, y_train), (x_test, y_test) = mnist.load_data()
25     # make sure the images are in white backgrround instead of black background it will make
        the inference easier for hand written digits for testing
26     x_train=255-x_train
27     x_test=255-x_test
28     # flatten the datapoints to 784 dimensional vector as the model is simple ANN and expects
        linear data points
29     x_train = np.array(x_train,dtype='float32').reshape(60000,-1)/255.0
30     x_test = np.array(x_test,dtype='float32').reshape(10000,-1)/255.0
31
32     model.fit(x_train,y_train, validation_split = 0.2,epochs =10)
33     model.evaluate(x_test,y_test)
34     # save the model for use in FAST API later
35     model.save('MNIST_model.keras')
36
37 if __name__=="__main__":
38     train_model()
39     # start the code from here

```