Abhishek Bose Follow
Nov 15, 2019 · 12 min read · · Listen

Tracking Deep Learning Experiments using Keras, MlFlow and MongoDb

It is late 2019 and ***Deep Learning*** is not a buzzword anymore. It is significantly used in the technology industry to attain feats of wonders which traditional machine learning and logic based techniques would take a longer time to achieve.

The main ingredient in Deep Learning are ***Neural Networks***, which are computation units called neurons, connected in a specific fashion to perform the task of learning and understanding data. When these networks become extremely deep and sophisticated, they are referred to as Deep Neural Networks and thus Deep Learning is performed.

A human brain learns about an object or concept when it visually experiences it for a longer amount of time. Similar to that, a neural network learns about objects and what they actually represent when it is fed with a large amount of data.

For example, let us consider the ***LeNet architecture*** . It is a small two layered CNN (Convolution Neural Network). Convolution Neural Networks are a special kind of neural network where the mathematical computation being done at every layer are convolution operations.

If enough images of a certain kind are fed to the ***LeNet*** architecture, it starts to understand and classify those images better.

That was a simple introduction to what neural networks are and how they behave.

In this article we will be mostly looking into three main frameworks which can ease out the developer experience of building these neural networks and tracking there performance efficiently.

Nowadays, neural networks are heavily used for classifying objects, predicting data and other similar tasks by many companies out there. When it comes it to training neural networks and keeping track of their performance, the experience is not too subtle.

When building a neural network, a developer would be trying out multiple datasets and experimenting with different hyperparameters. It is essential to keep a track a of these parameters and how they affect the output of the neural networks.

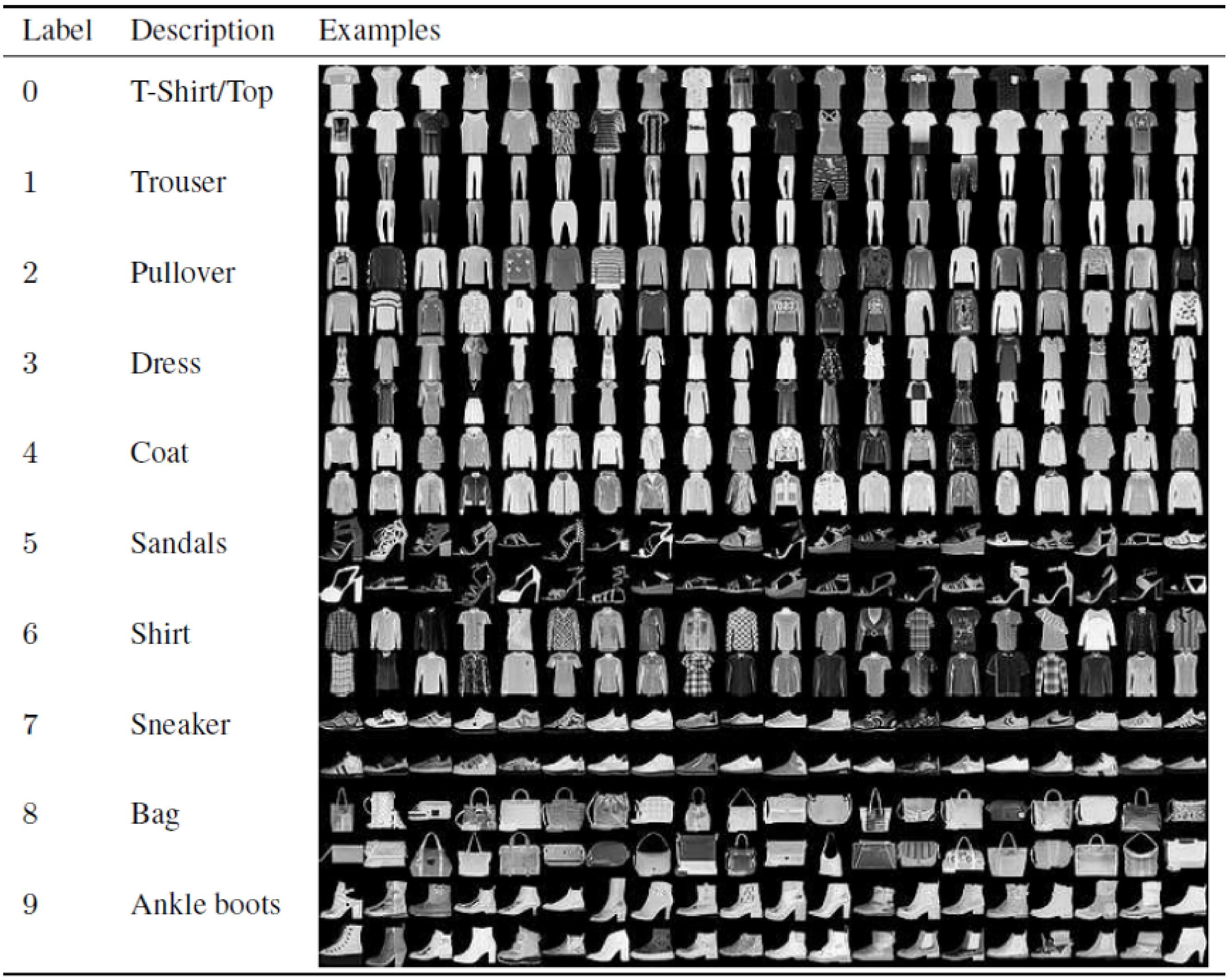
Also debugging neural networks is an extremely cumbersome task. The output performance of different neural networks may vary due to different reasons. Some of the possible causes maybe inadequate data pre-processing, incorrect optimizer, a learning rate which is too low or too high. The number of variables which affect the performance of a neural network are quite a few. Hence it is essential that every parameter is properly tracked and maintained.

Some of the available options present out there include the infamous ***Tensorboard, Bokeh*** to name a few.

outputs of every experiment performed with great precision. We will be looking into **MlFlow's** components with more detail in the subsequent sections.

The framework we would be using for writing our neural networks and training them is **Keras**.

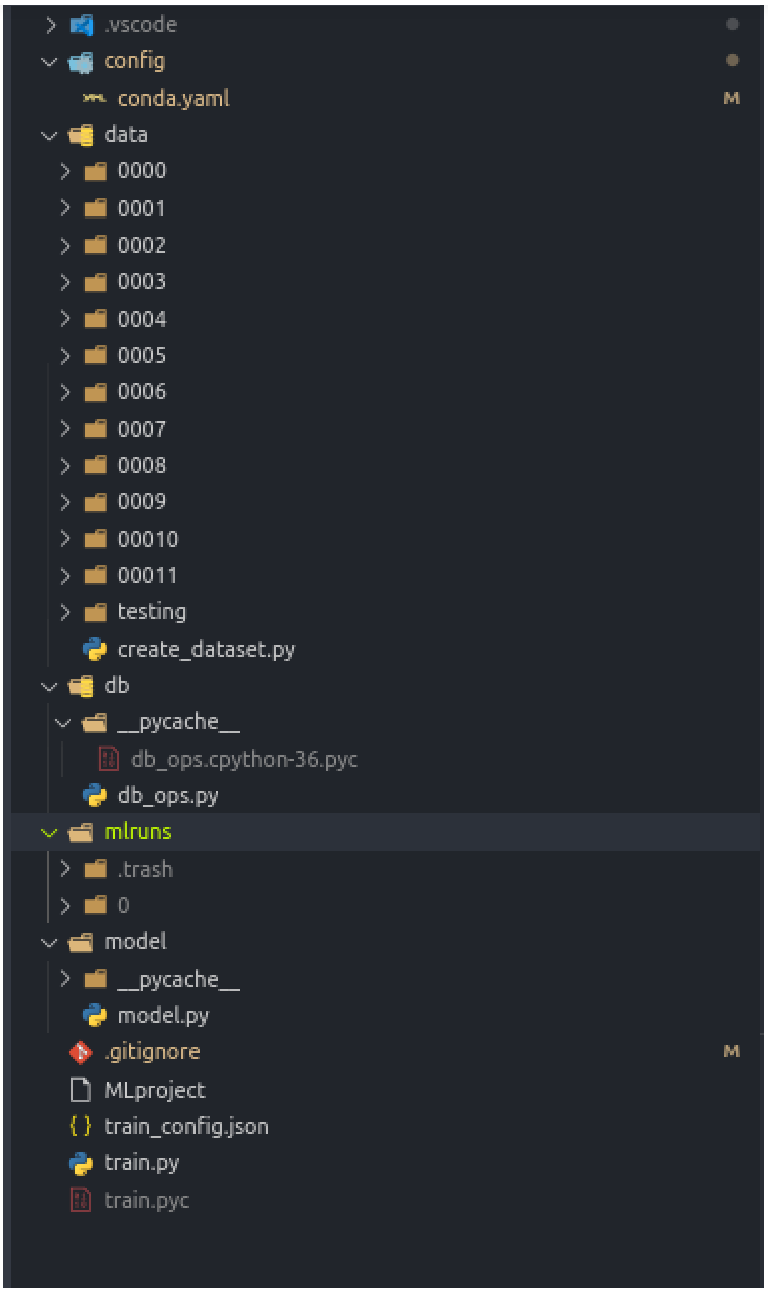
We will be using the **FashionMNIST** dataset. It contains a total of 70000 gray scale images (training:test = 60000:10000) , each scaled at 28x28 associated with one from 10 classes. (Fig 1)



Label	Description	Examples
0	T-Shirt/Top	
1	Trouser	
2	Pullover	
3	Dress	
4	Coat	
5	Sandals	
6	Shirt	
7	Sneaker	
8	Bag	
9	Ankle boots	

Fig 1: Fashion Mnist Dataset

The folder structure of our project looks as shown in Fig 2 below.



```
> .vscode
v config
    conda.yaml                M
v data
  > 0000
  > 0001
  > 0002
  > 0003
  > 0004
  > 0005
  > 0006
  > 0007
  > 0008
  > 0009
  > 00010
  > 00011
  > testing
    create_dataset.py
v db
  v __pycache__
      db_ops.cpython-36.pyc
    db_ops.py
v mlruns
  > .trash
  > 0
v model
  > __pycache__
    model.py
  .gitignore                  M
  MLproject
  train_config.json
  train.py
  train.pyc
```

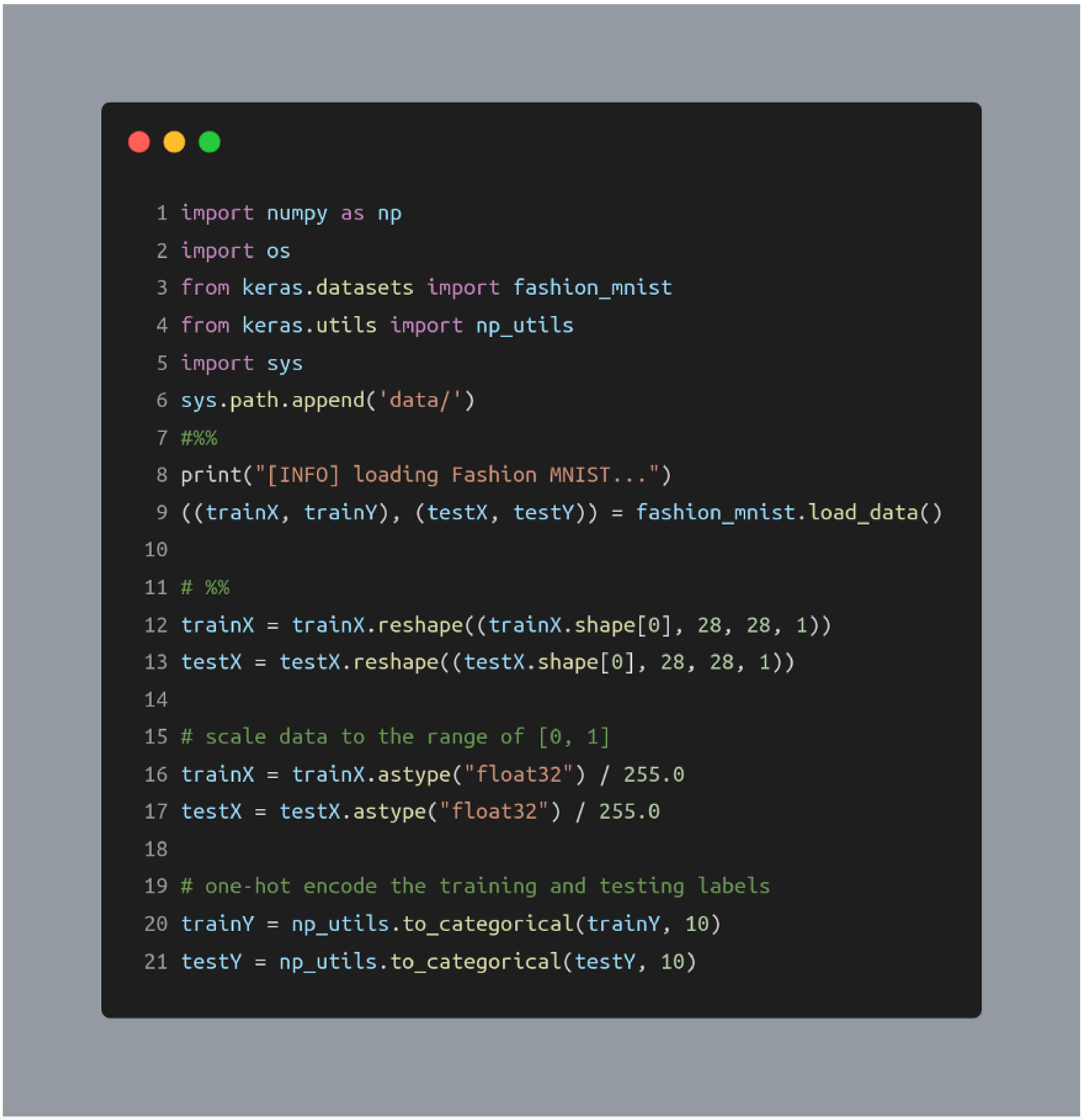
Fig 2: Project folder structure

The data folder contains our ***fashion mnist*** dataset files which will be used for training the model. The db folder contains the python driver code to perform operations on ***MongoDb*** collections. ***MongoDB*** is an extremely easy to use ***NoSql*** database which has been built keeping in developer satisfaction. It is easily integrable with modern day applications and has a large developer community contributing to it's extensions regularly. The model folder contains piece of code with the neural network model definition. The ***mlruns*** folder is created automatically once ***mlflow*** is invoked in the main code.

only one dataset, we will split it into equal parts in order to simulate a multiple dataset scenario.

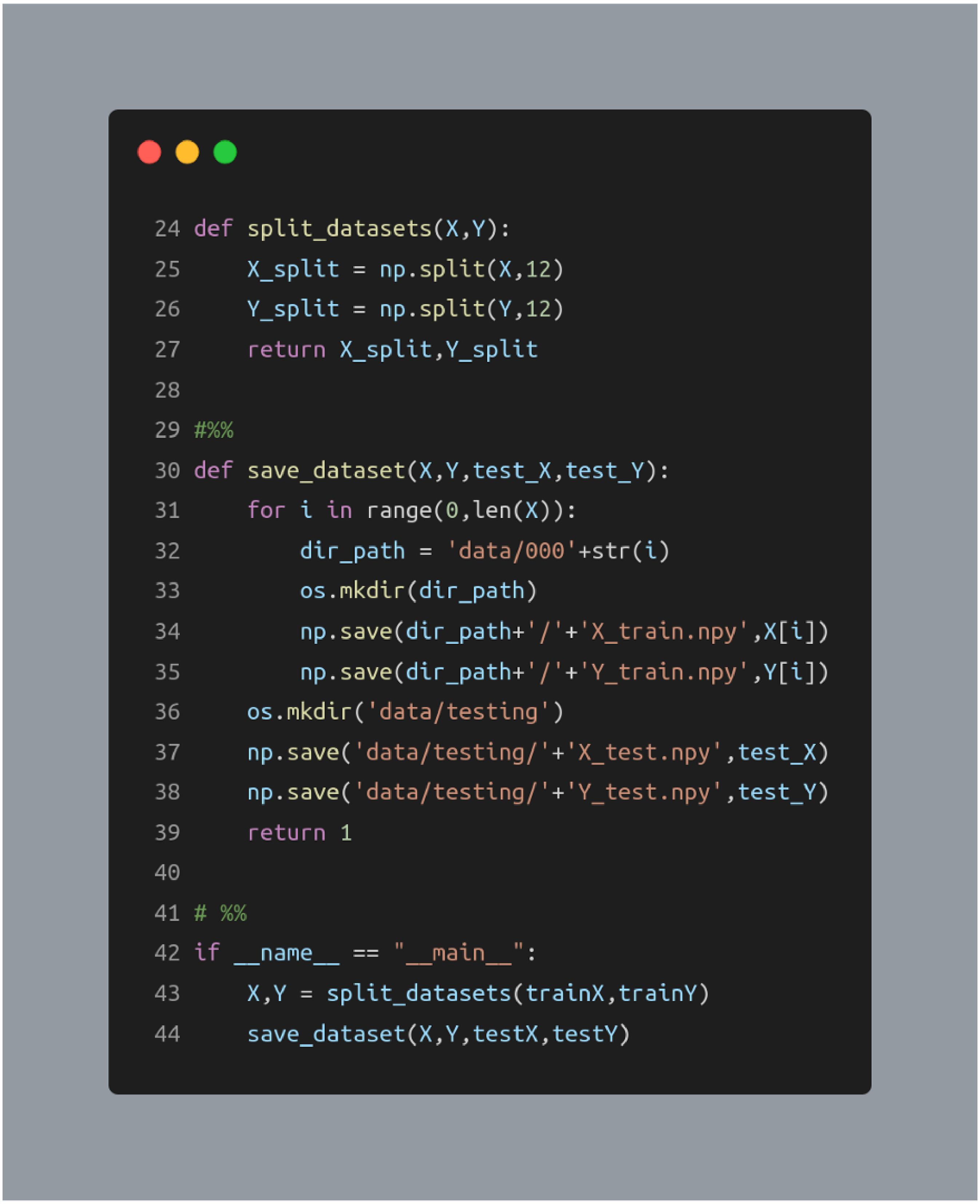
Let’s start off with the ***create_dataset*** script, which is used to split the fashion mnist into equal parts and store them inside the data folder with proper serial number.

In Fig 3 shown below, we import fashion mnist from ***keras.datasets*** and perform the necessary normalization step



```
import numpy as np
import os
from keras.datasets import fashion_mnist
from keras.utils import np_utils
import sys
sys.path.append('data/')
#%%
print("[INFO] loading Fashion MNIST...")
((trainX, trainY), (testX, testY)) = fashion_mnist.load_data()

# %%
trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
testX = testX.reshape((testX.shape[0], 28, 28, 1))

# scale data to the range of [0, 1]
trainX = trainX.astype("float32") / 255.0
testX = testX.astype("float32") / 255.0

# one-hot encode the training and testing labels
trainY = np_utils.to_categorical(trainY, 10)
testY = np_utils.to_categorical(testY, 10)
```

directory (Fig 4)



```python
def split_datasets(X,Y):
    X_split = np.split(X,12)
    Y_split = np.split(Y,12)
    return X_split,Y_split

#%%
def save_dataset(X,Y,test_X,test_Y):
    for i in range(0,len(X)):
        dir_path = 'data/000'+str(i)
        os.mkdir(dir_path)
        np.save(dir_path+'/'+'X_train.npy',X[i])
        np.save(dir_path+'/'+'Y_train.npy',Y[i])
    os.mkdir('data/testing')
    np.save('data/testing/'+'X_test.npy',test_X)
    np.save('data/testing/'+'Y_test.npy',test_Y)
    return 1

# %%
if __name__ == "__main__":
    X,Y = split_datasets(trainX,trainY)
    save_dataset(X,Y,testX,testY)
```

Fig 4: The large dataset is equally split into 12 equal parts for training

managing data with flexibility. Given ***NoSql***'s schema-less nature, managing collections and documents is a breeze.

The ease with which any document can be edited in ***MongoDB*** is superb. The best part is, whenever a collection is queried , the result returned is a ***json*** making it extremely easy to be parsed using any programming language. Aggregation queries in mongo are also very simple and allows users to cross reference collections in a swift manner.

In order to connect our python scripts with MongoDb we will be using **pymongo** which can be easily installed using the ***pip install pymongo***. To install MongoDb, follow this tutorial ***https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/***

Once MongoDb is installed and tested to be running properly on your system, create a new database called fashion_mnist. Inside the database create a new collection named dataset as shown in Fig 5 below.

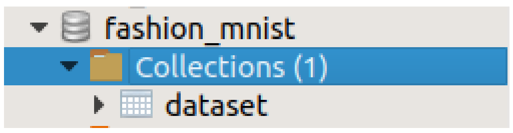
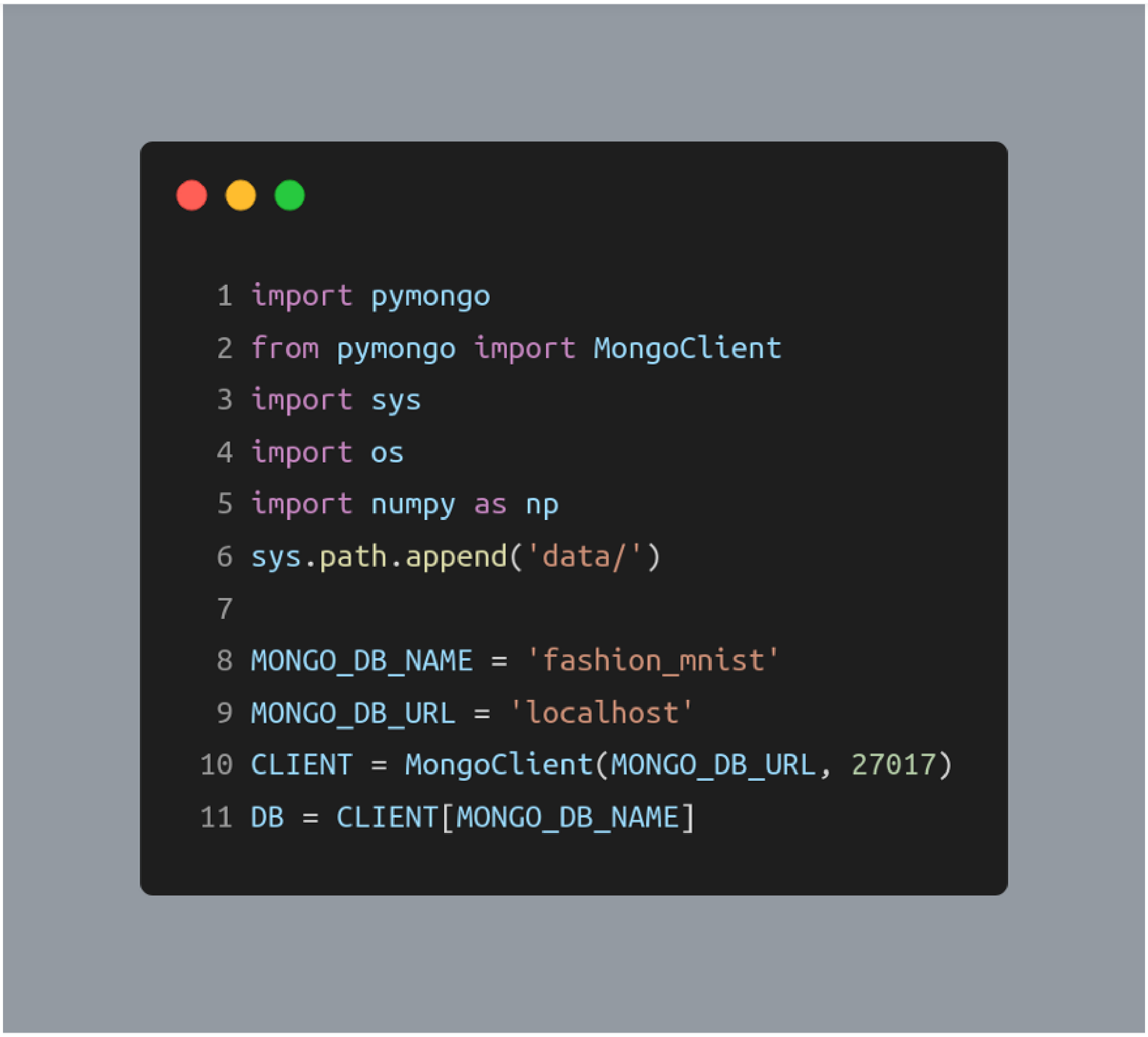


Fig 5: Collection named as Dataset created in MongoDb

A great GUI to interact with ***MongoDb*** is ***robo3t***. It's free and easy to use. It can be downloaded from the following link ***https://robomongo.org/download.***Since our DB is setup and datasets are created, we can progress with the task of inserting necessary information into the dataset collection



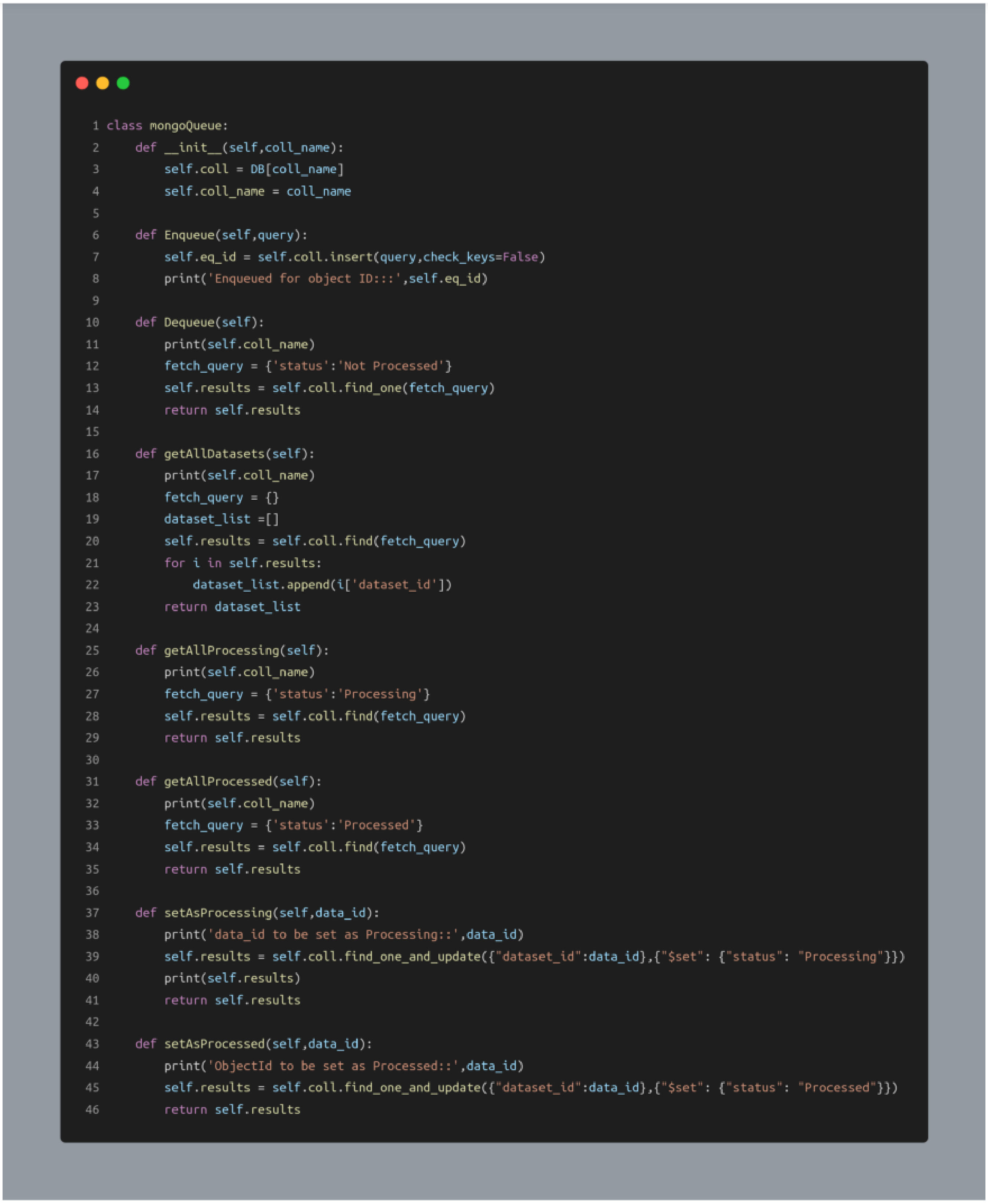
```python
import pymongo
from pymongo import MongoClient
import sys
import os
import numpy as np
sys.path.append('data/')

MONGO_DB_NAME = 'fashion_mnist'
MONGO_DB_URL = 'localhost'
CLIENT = MongoClient(MONGO_DB_URL, 27017)
DB = CLIENT[MONGO_DB_NAME]
```

Fig 6: MongoClient is configured in db_ops python script

In Fig 6 shown above, we are importing ***MongoClient*** from the ***pymongo*** library which will essentially be used to connect to our ***mongoDB*** database.

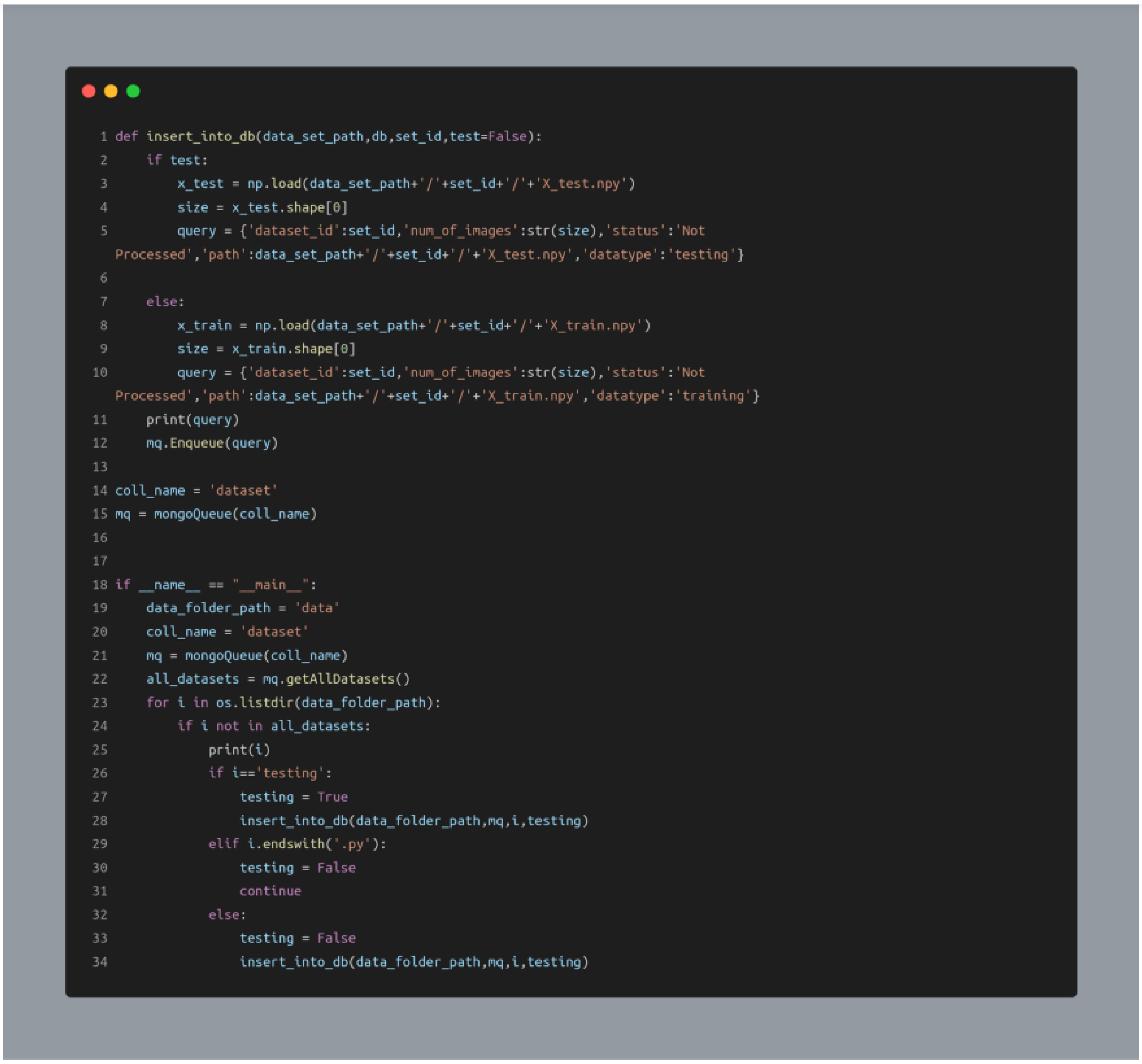
Fig 7 below describes ***mongoQueue*** class which has been written in order to interact with our dataset collection. In line 18 and 19, the collection name is initialized, which is used in all the member functions. The **Enqueue** method in line 6 is used to insert dataset information into the dataset collection. The ***Dequeue*** method in line 10 fetches the first dataset which has a *status* field of '***Not* Processed**'. The **setAsProcessing** *and* **setAsProcessed** methods are used to set the status field of respective dataset documents in the collection.

Open in app Get started



```python
class mongoQueue:
    def __init__(self,coll_name):
        self.coll = DB[coll_name]
        self.coll_name = coll_name

    def Enqueue(self,query):
        self.eq_id = self.coll.insert(query,check_keys=False)
        print('Enqueued for object ID:::',self.eq_id)

    def Dequeue(self):
        print(self.coll_name)
        fetch_query = {'status':'Not Processed'}
        self.results = self.coll.find_one(fetch_query)
        return self.results

    def getAllDatasets(self):
        print(self.coll_name)
        fetch_query = {}
        dataset_list =[]
        self.results = self.coll.find(fetch_query)
        for i in self.results:
            dataset_list.append(i['dataset_id'])
        return dataset_list

    def getAllProcessing(self):
        print(self.coll_name)
        fetch_query = {'status':'Processing'}
        self.results = self.coll.find(fetch_query)
        return self.results

    def getAllProcessed(self):
        print(self.coll_name)
        fetch_query = {'status':'Processed'}
        self.results = self.coll.find(fetch_query)
        return self.results

    def setAsProcessing(self,data_id):
        print('data_id to be set as Processing::',data_id)
        self.results = self.coll.find_one_and_update({"dataset_id":data_id},{"$set": {"status": "Processing"}})
        print(self.results)
        return self.results

    def setAsProcessed(self,data_id):
        print('ObjectId to be set as Processed::',data_id)
        self.results = self.coll.find_one_and_update({"dataset_id":data_id},{"$set": {"status": "Processed"}})
        return self.results
```

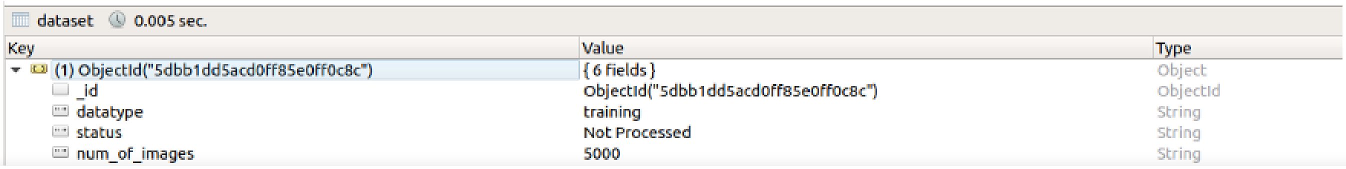
Fig 7: MongoQueue class for handling operations on the database



```
 1 def insert_into_db(data_set_path,db,set_id,test=False):
 2     if test:
 3         x_test = np.load(data_set_path+'/'+set_id+'/'+'X_test.npy')
 4         size = x_test.shape[0]
 5         query = {'dataset_id':set_id,'num_of_images':str(size),'status':'Not
   Processed','path':data_set_path+'/'+set_id+'/'+'X_test.npy','datatype':'testing'}
 6
 7     else:
 8         x_train = np.load(data_set_path+'/'+set_id+'/'+'X_train.npy')
 9         size = x_train.shape[0]
10         query = {'dataset_id':set_id,'num_of_images':str(size),'status':'Not
   Processed','path':data_set_path+'/'+set_id+'/'+'X_train.npy','datatype':'training'}
11     print(query)
12     mq.Enqueue(query)
13
14 coll_name = 'dataset'
15 mq = mongoQueue(coll_name)
16
17
18 if __name__ == "__main__":
19     data_folder_path = 'data'
20     coll_name = 'dataset'
21     mq = mongoQueue(coll_name)
22     all_datasets = mq.getAllDatasets()
23     for i in os.listdir(data_folder_path):
24         if i not in all_datasets:
25             print(i)
26             if i=='testing':
27                 testing = True
28                 insert_into_db(data_folder_path,mq,i,testing)
29             elif i.endswith('.py'):
30                 testing = False
31                 continue
32             else:
33                 testing = False
34                 insert_into_db(data_folder_path,mq,i,testing)
```

Fig 8: Methods to insert data into MongoDB

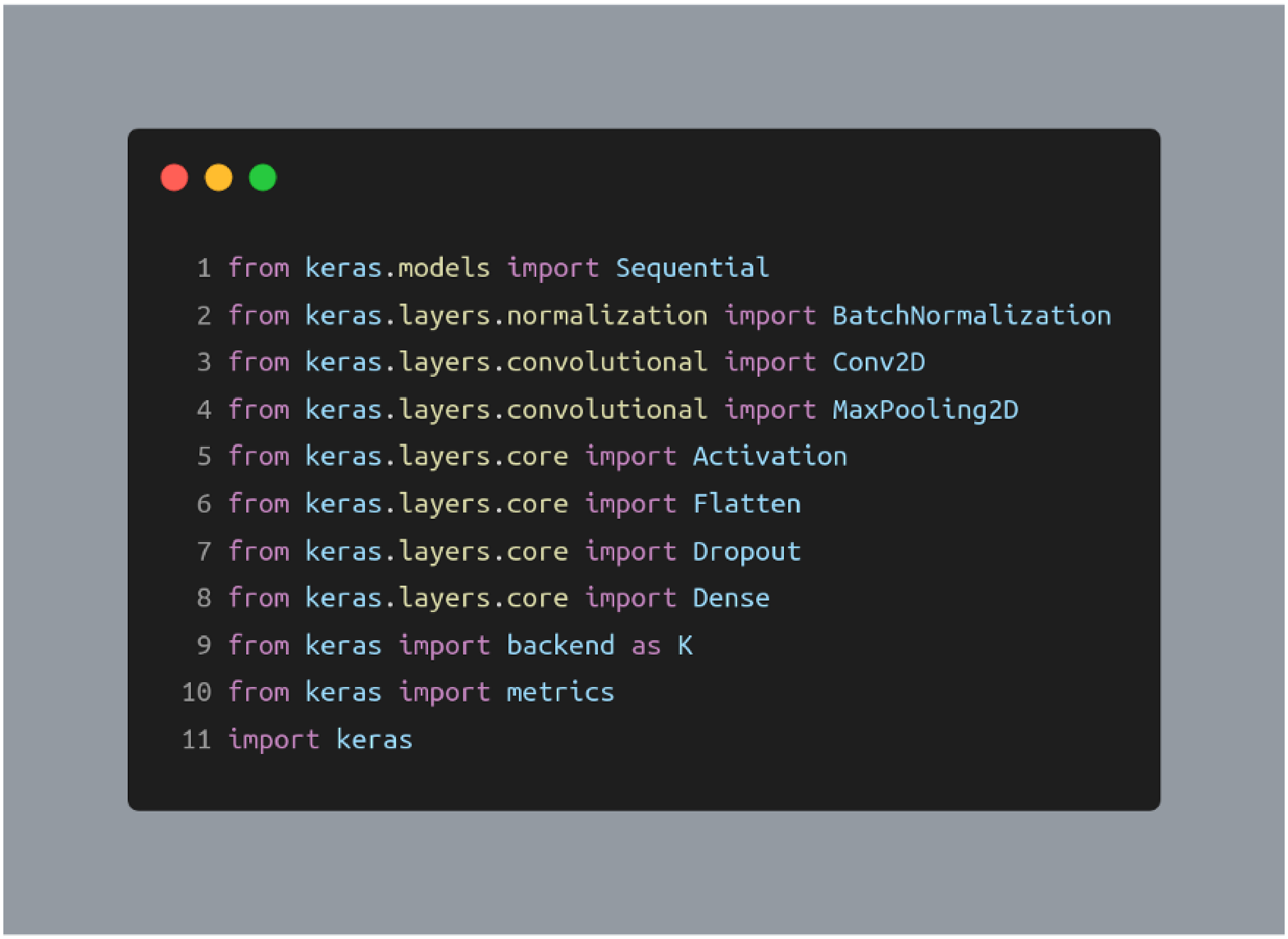
We use the ***insert_into_db*** method shown in Fig 8, line 1, to insert information about our newly created datasets into our **mongoDb** dataset collection. In line 23 of the *main* function, we iterate over the dataset folder and call ***insert_into_db*** to insert the necessary information for that dataset into the collection. Once every dataset is successfully inserted into the collection, the fields appear as shown in Fig 9 below.



dataset 0.005 sec.

Key	Value	Type
(1) ObjectId("5dbb1dd5acd0ff85e0ff0c8c")	{ 6 fields }	Object
_id	ObjectId("5dbb1dd5acd0ff85e0ff0c8c")	ObjectId
datatype	training	String
status	Not Processed	String
num_of_images	5000	String

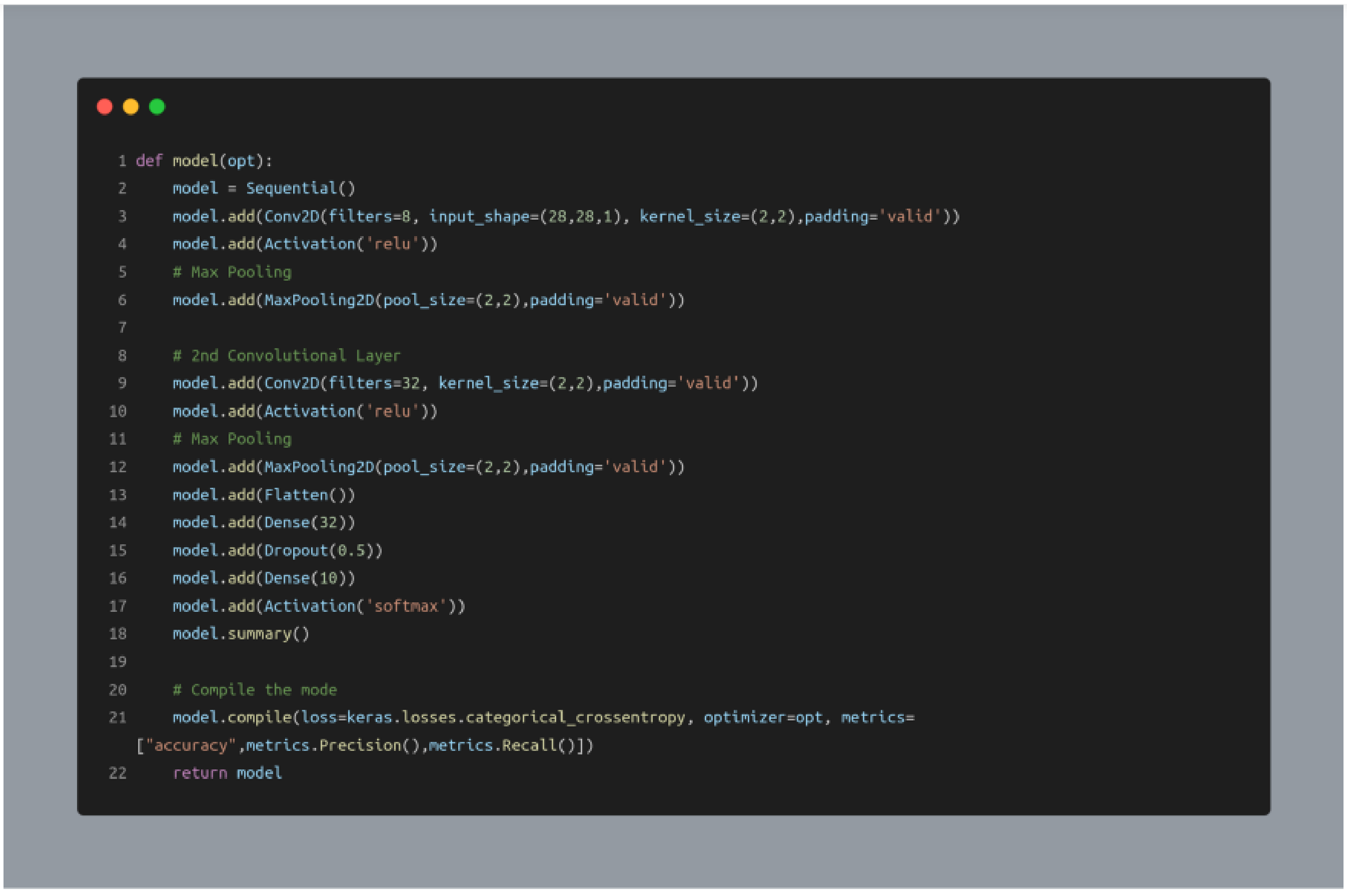
We can now define our model for training our deep learning network. Inside ***model/model.py*** we import all necessary ***keras*** packages to build our CNN network (shown in Fig 10a)



```python
from keras.models import Sequential
from keras.layers.normalization import BatchNormalization
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dropout
from keras.layers.core import Dense
from keras import backend as K
from keras import metrics
import keras
```

Fig 10a: Importing all necessary keras packages

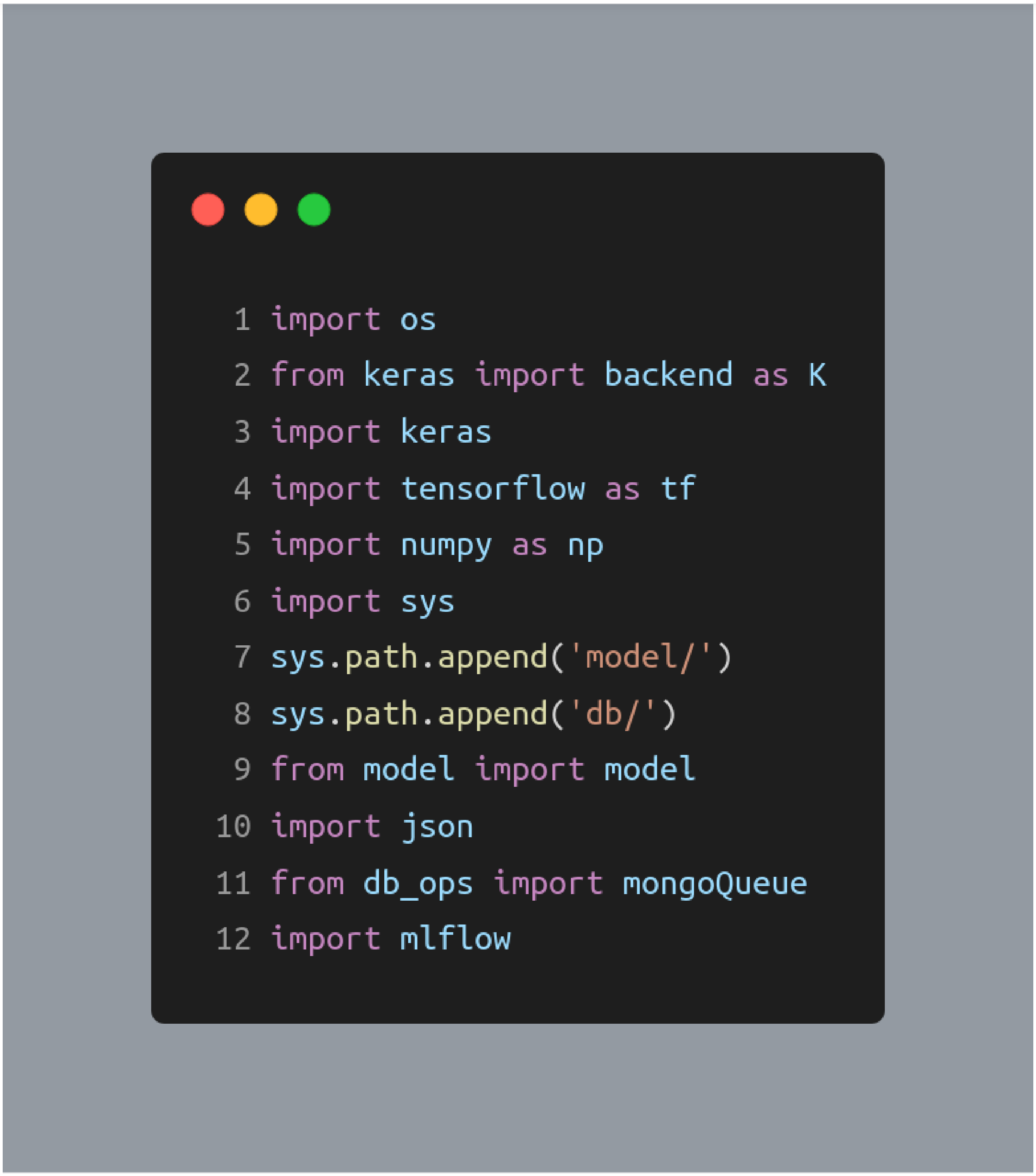
Fig 10b shows the model architecture. It is a simple two layer CNN, with two **MaxPool** layers and **RelU** activation in between. Two **Dense** layers are also added with 32 and 10 neurons respectively. I have also added a **Dropout** of 0.5 before the last Dense layer.



```python
def model(opt):
    model = Sequential()
    model.add(Conv2D(filters=8, input_shape=(28,28,1), kernel_size=(2,2),padding='valid'))
    model.add(Activation('relu'))
    # Max Pooling
    model.add(MaxPooling2D(pool_size=(2,2),padding='valid'))

    # 2nd Convolutional Layer
    model.add(Conv2D(filters=32, kernel_size=(2,2),padding='valid'))
    model.add(Activation('relu'))
    # Max Pooling
    model.add(MaxPooling2D(pool_size=(2,2),padding='valid'))
    model.add(Flatten())
    model.add(Dense(32))
    model.add(Dropout(0.5))
    model.add(Dense(10))
    model.add(Activation('softmax'))
    model.summary()

    # Compile the mode
    model.compile(loss=keras.losses.categorical_crossentropy, optimizer=opt, metrics=
["accuracy",metrics.Precision(),metrics.Recall()])
    return model
```

Fig 10b: CNN model definintion

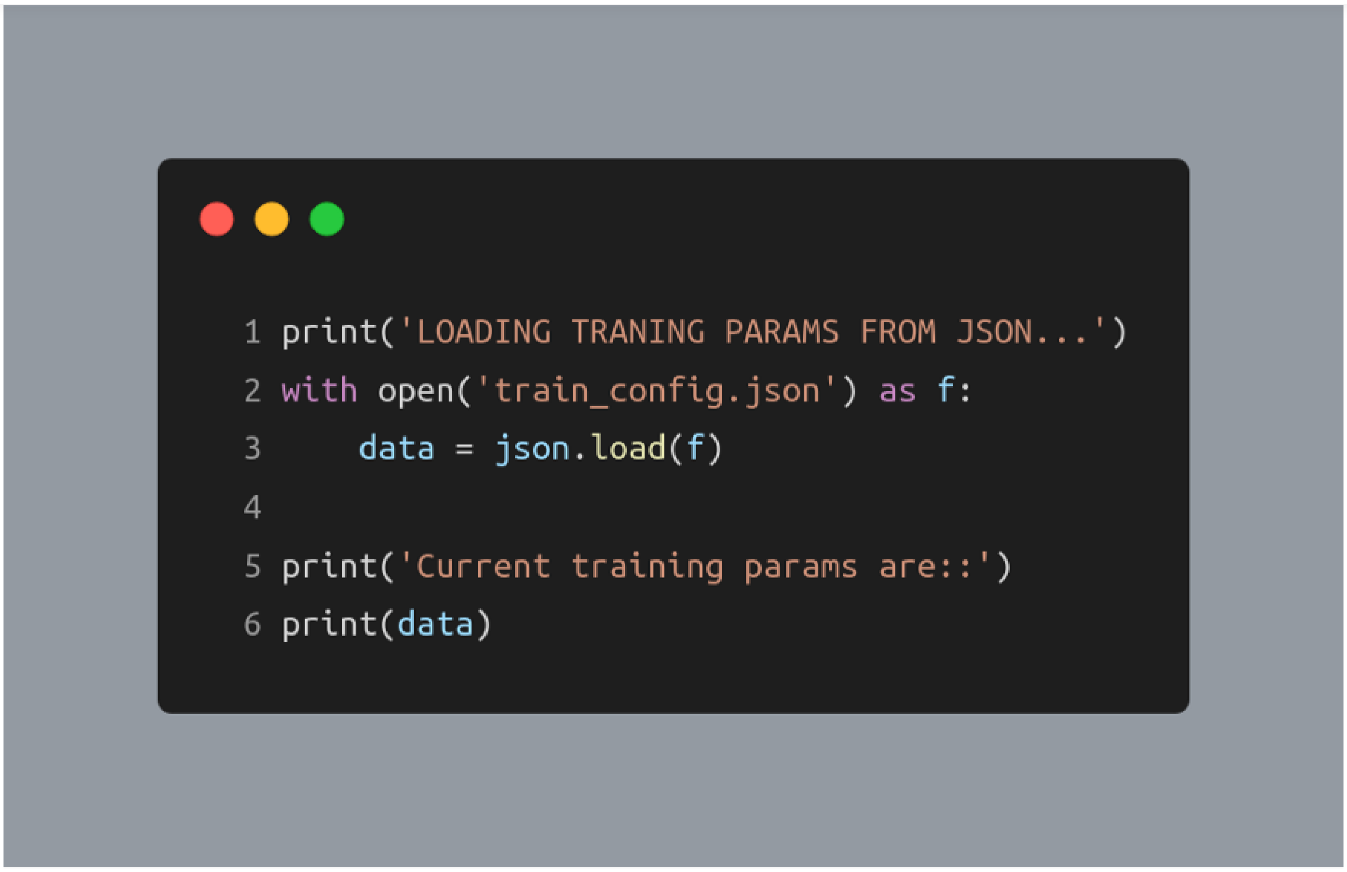
Now, in our ***train.py*** script we import all the necessary modules needed from the ***keras*** library to get on with our training. Along with all the ***keras*** libraries we import ***mlflow*** as well (Fig 11)



```python
import os
from keras import backend as K
import keras
import tensorflow as tf
import numpy as np
import sys
sys.path.append('model/')
sys.path.append('db/')
from model import model
import json
from db_ops import mongoQueue
import mlflow
```

Fig 11: Importing packages in the main script

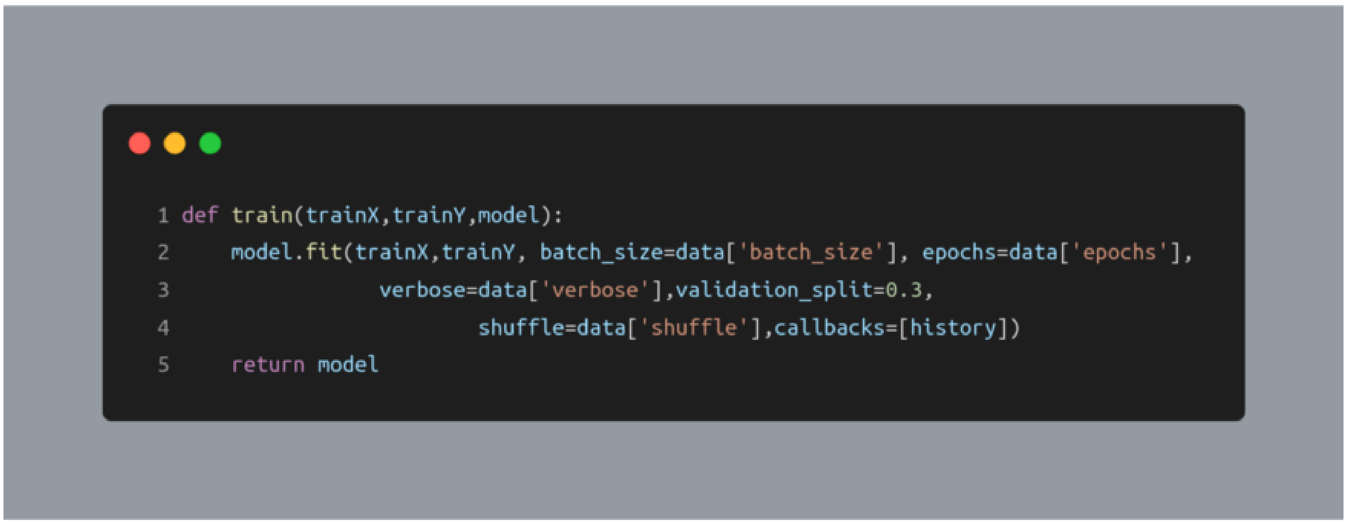
All the hyperparameters which will be used for training is stored in a config file named as ***train_config.json***. This file is read (Fig 12a) and used for defining training parameters



```python
print('LOADING TRANING PARAMS FROM JSON...')
with open('train_config.json') as f:
    data = json.load(f)

print('Current training params are::')
print(data)
```

Fig 12a: parameters required for training are loaded

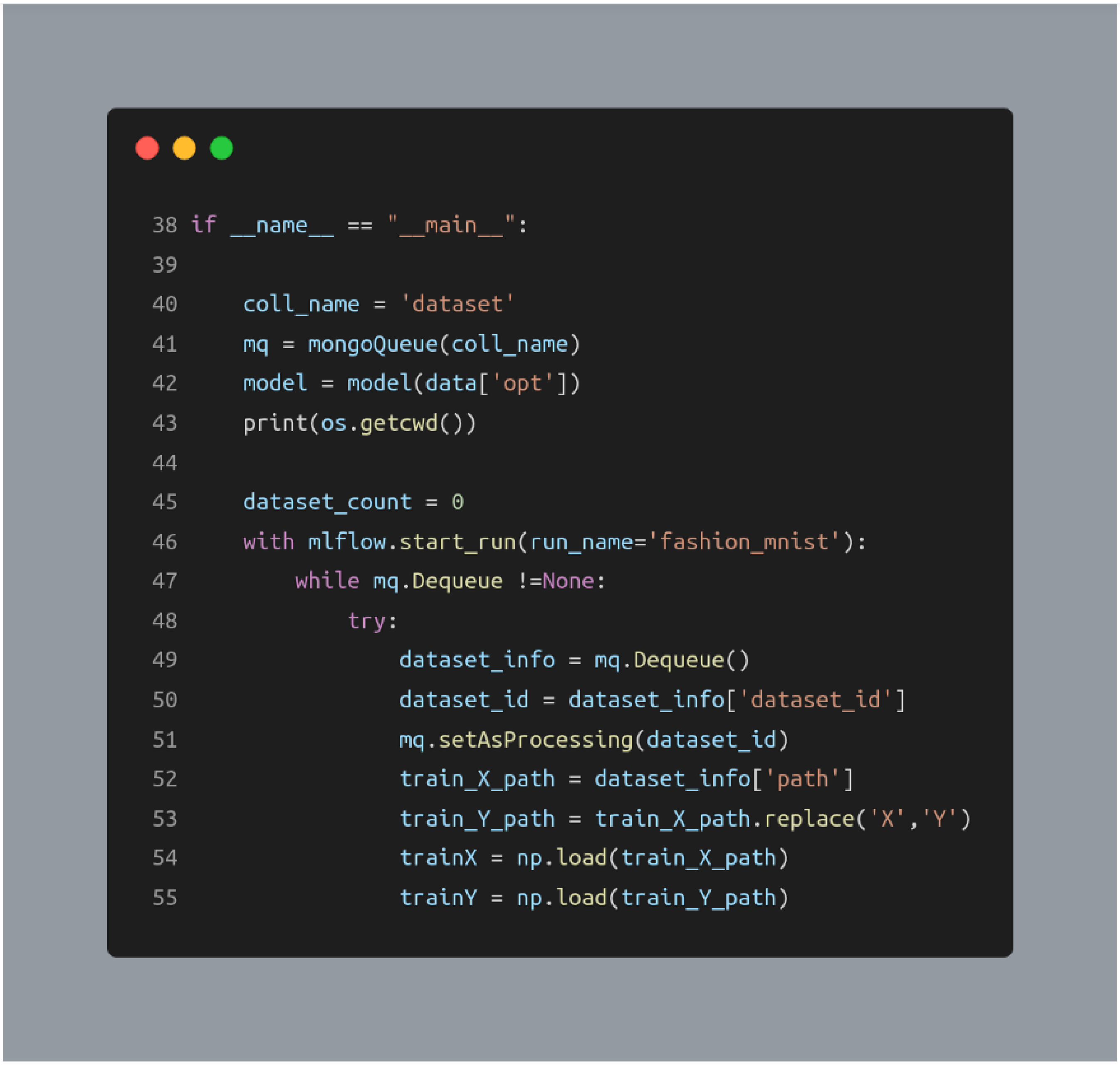
In Fig 12b, we have defined out training function , which takes arguments ***trainX** (*our training set) ***,trainY** (*training set labels*)* and the ***model*** (CNN model)



```python
def train(trainX,trainY,model):
    model.fit(trainX,trainY, batch_size=data['batch_size'], epochs=data['epochs'],
                verbose=data['verbose'],validation_split=0.3,
                        shuffle=data['shuffle'],callbacks=[history])
    return model
```

Fig 12b: Function to start training

From line 38 (In Fig 13), the **main** function starts where we define our ***MongoQueue***



```
38 if __name__ == "__main__":
39
40     coll_name = 'dataset'
41     mq = mongoQueue(coll_name)
42     model = model(data['opt'])
43     print(os.getcwd())
44
45     dataset_count = 0
46     with mlflow.start_run(run_name='fashion_mnist'):
47         while mq.Dequeue !=None:
48             try:
49                 dataset_info = mq.Dequeue()
50                 dataset_id = dataset_info['dataset_id']
51                 mq.setAsProcessing(dataset_id)
52                 train_X_path = dataset_info['path']
53                 train_Y_path = train_X_path.replace('X','Y')
54                 trainX = np.load(train_X_path)
55                 trainY = np.load(train_Y_path)
```

Fig 13: Main function which trains and also logs data using MlFlow

In line 42 (Fig 13) , a new CNN model is created by calling the ***model*** function which accepts the optimizer type as input. In this experiment we would be using the '**SGD**' (**Stochastic Gradient Descent**) to train the network .

Everytime we invoke ***mlflow*** in our training code for logging, it is known as an ***mlflow*** run. ***MlFlow*** provides us with an API for starting and managing ***MlFlow*** runs. For example, Fig 14a and 14b

```
import mlflow
```

Fig 14a: Mlflow example (importing and logging params)

```
with mlflow.start_run() as run:
    ...
```

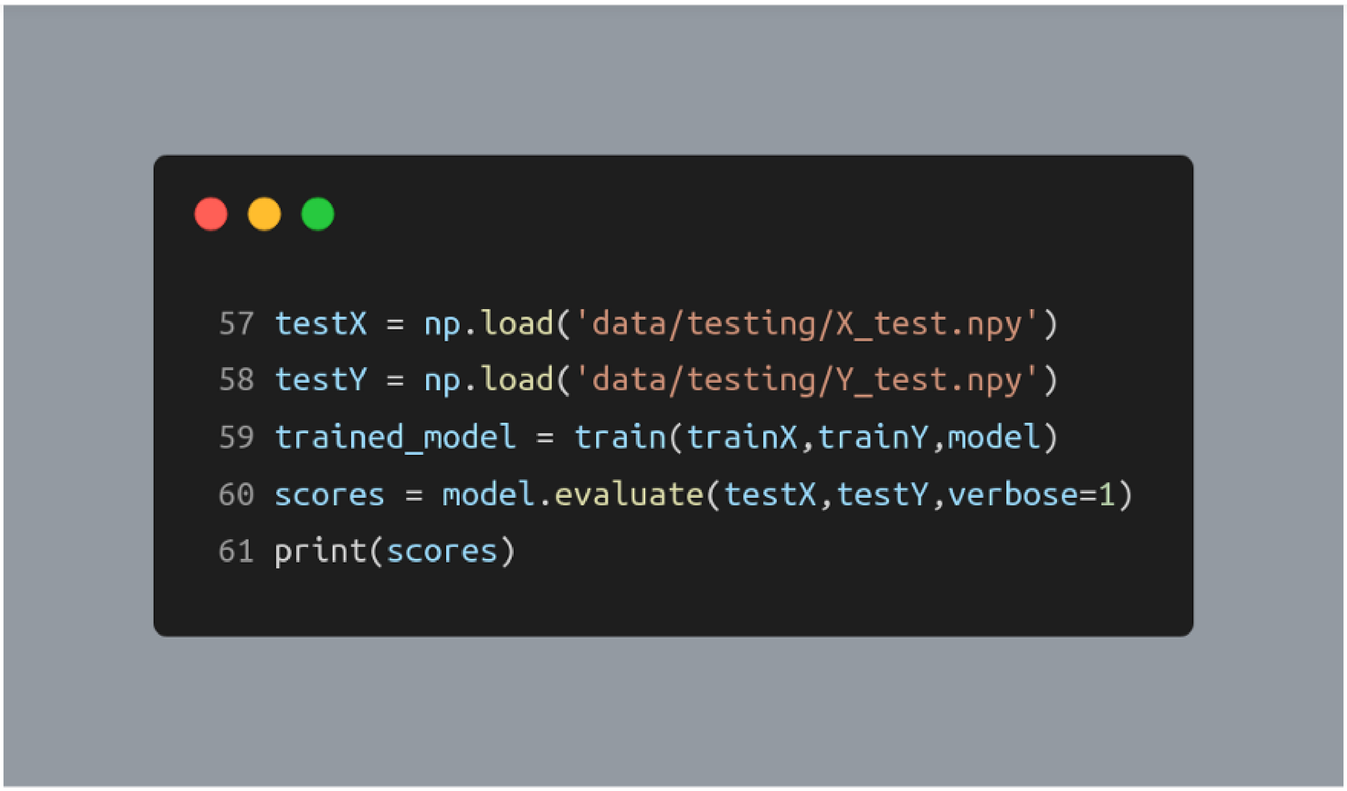
Fig 14b: Context managers can be used to declare and Mlflow run

In our code we start the ***mlflow*** run using the python ***context manager*** as shown in Fig 14b.

At line 46 in Fig 13, we define our our ***mlflow*** run with the run name as '***fashion mnist***'. All data and metrics will be logged under this run name on the ***mlflow*** dashboard.

From line 47 we start a while loop, which continuously invokes the ***dequeue*** function from the **MongoQeueue** class. What this does is fetches every row corresponding to a particular dataset from the dataset collection which has a **status field** = ***Not Processed*** (Fig 9). As soon as this dataset is fetched, **setAsProcessing** function is called in line 51 which sets the status of that dataset to *Processing* in **MongoDb**. This enables us to understand which dataset is currently being trained by our system. This is particularly helpful is large systems where there are multiple datasets and many training instances running in parallel.

In lines 54 and 55, the datasets are loaded from the ***data*** folder corresponding to the ***dataset_id*** fetched from the db.

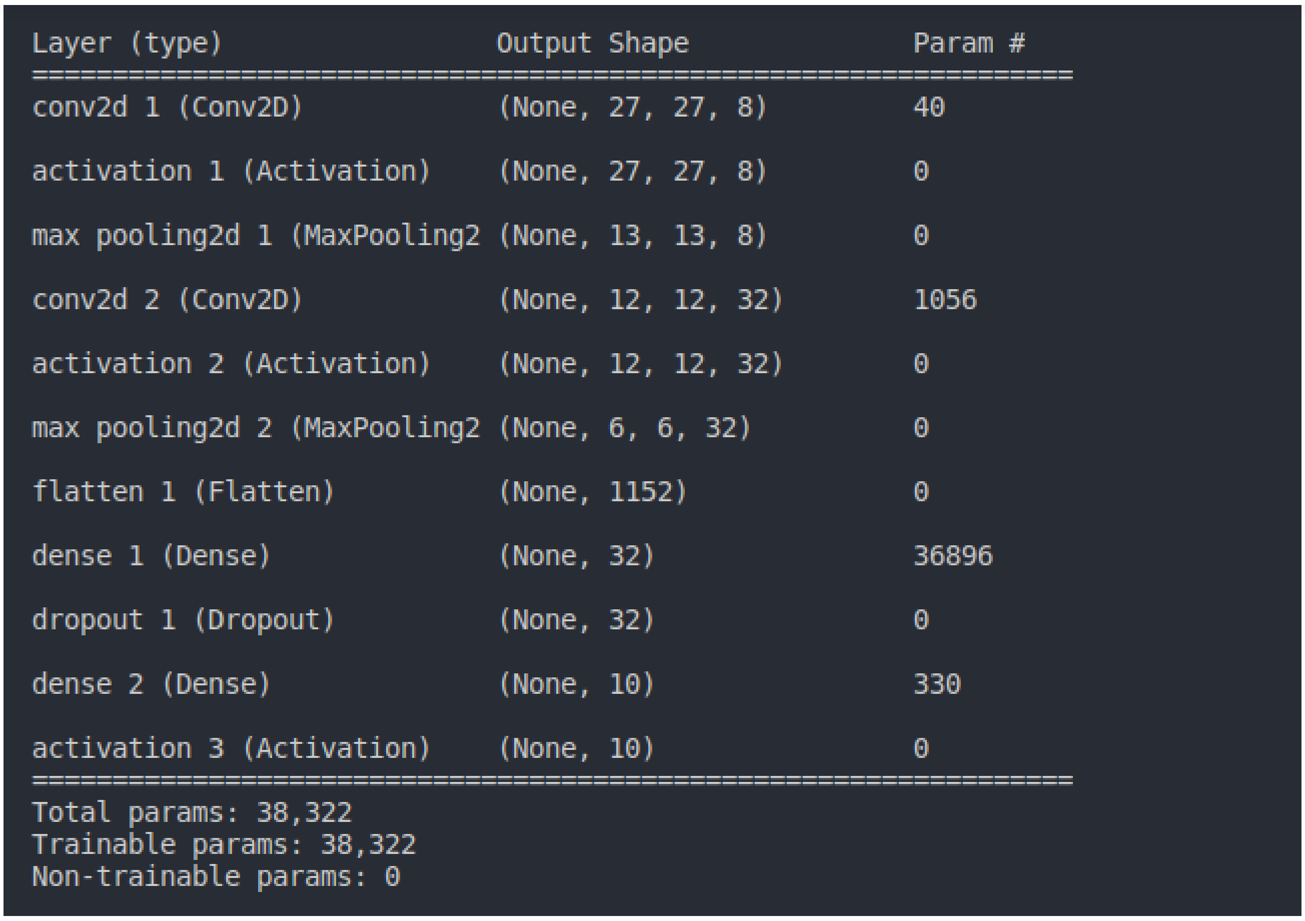


```
57 testX = np.load('data/testing/X_test.npy')
58 testY = np.load('data/testing/Y_test.npy')
59 trained_model = train(trainX,trainY,model)
60 scores = model.evaluate(testX,testY,verbose=1)
61 print(scores)
```

Fig 15: Loading Test data and evaluating on it

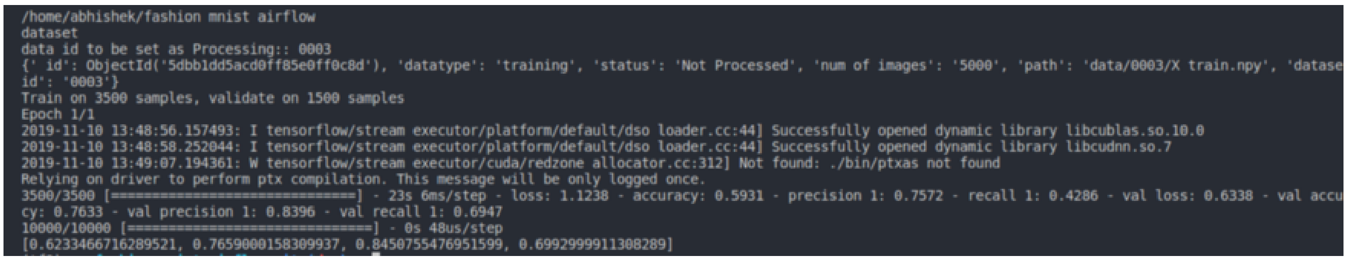
Lines 57 and 58 loads the test sets and the training is started at line 59 by calling the *train* function. We then use the trained model to predict our scores as shown in line 60 in Fig 15.

The training output looks as shown below (Fig 16a and Fig 16b)



```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d 1 (Conv2D)            (None, 27, 27, 8)         40

activation 1 (Activation)    (None, 27, 27, 8)         0

max pooling2d 1 (MaxPooling2 (None, 13, 13, 8)         0

conv2d 2 (Conv2D)            (None, 12, 12, 32)        1056

activation 2 (Activation)    (None, 12, 12, 32)        0

max pooling2d 2 (MaxPooling2 (None, 6, 6, 32)          0

flatten 1 (Flatten)          (None, 1152)              0

dense 1 (Dense)              (None, 32)                36896

dropout 1 (Dropout)          (None, 32)                0

dense 2 (Dense)              (None, 10)                330

activation 3 (Activation)    (None, 10)                0
=================================================================
Total params: 38,322
Trainable params: 38,322
Non-trainable params: 0
```

Fig 16a: Model summary when starting to train

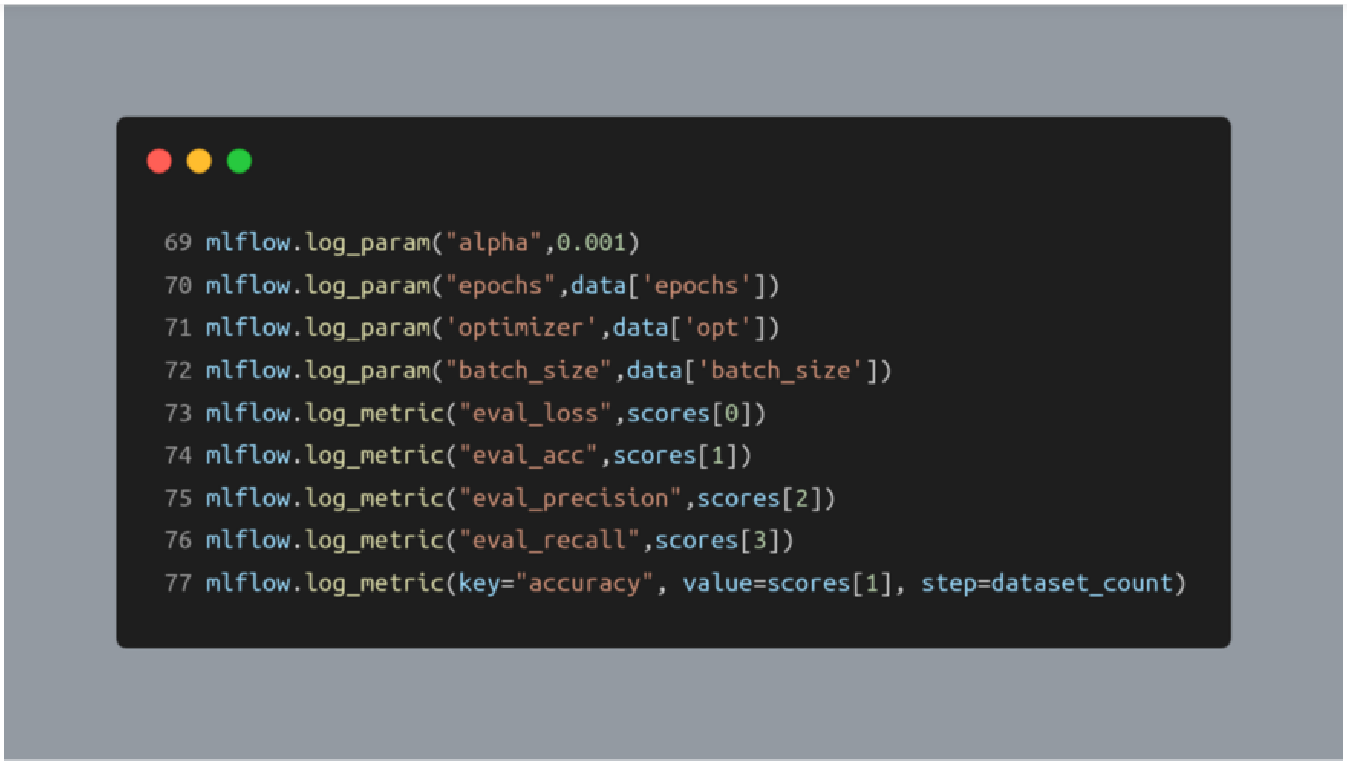


```
/home/abhishek/fashion mnist airflow
dataset
data id to be set as Processing:: 0003
{' id': ObjectId('5dbb1dd5acd0ff85e0ff0c8d'), 'datatype': 'training', 'status': 'Not Processed', 'num of images': '5000', 'path': 'data/0003/X train.npy', 'datase
id': '0003'}
Train on 3500 samples, validate on 1500 samples
Epoch 1/1
2019-11-10 13:48:56.157493: I tensorflow/stream executor/platform/default/dso loader.cc:44] Successfully opened dynamic library libcublas.so.10.0
2019-11-10 13:48:58.252044: I tensorflow/stream executor/platform/default/dso loader.cc:44] Successfully opened dynamic library libcudnn.so.7
2019-11-10 13:49:07.194361: W tensorflow/stream executor/cuda/redzone allocator.cc:312] Not found: ./bin/ptxas not found
Relying on driver to perform ptx compilation. This message will be only logged once.
3500/3500 [==============================] - 23s 6ms/step - loss: 1.1238 - accuracy: 0.5931 - precision 1: 0.7572 - recall 1: 0.4286 - val loss: 0.6338 - val accu
cy: 0.7633 - val precision 1: 0.8396 - val recall 1: 0.6947
10000/10000 [==============================] - 0s 48us/step
[0.6233466716289521, 0.7659000158309937, 0.8450755476951599, 0.6992999911308289]
```

Fig 16b : Outputs after first epoch

As shown in Fig 16b, the training happens for an epoch and the evaluation metrics for the test dataset gets logged.

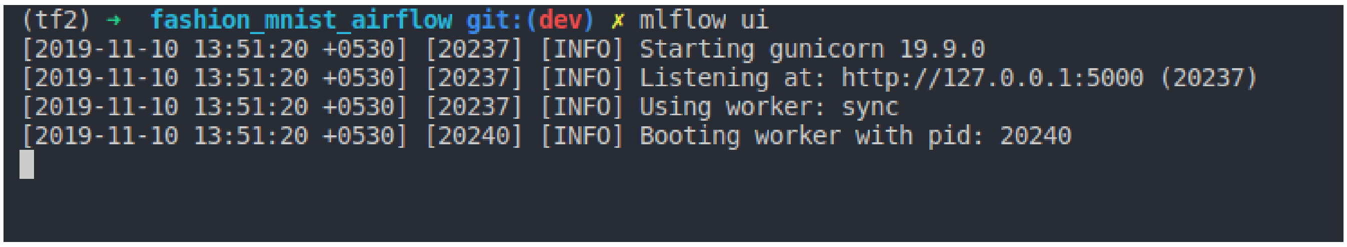
All outputs of the evaluation done using our trained model can be logged using the ***MlFlow*** tracking api (as shown in Fig 17). The ***tracking API*** comes with functions such as ***log_param*** and ***log_metric*** which enables us to log every hyperparameter and output values into ***mlflow***.



```
mlflow.log_param("alpha",0.001)
mlflow.log_param("epochs",data['epochs'])
mlflow.log_param('optimizer',data['opt'])
mlflow.log_param("batch_size",data['batch_size'])
mlflow.log_metric("eval_loss",scores[0])
mlflow.log_metric("eval_acc",scores[1])
mlflow.log_metric("eval_precision",scores[2])
mlflow.log_metric("eval_recall",scores[3])
mlflow.log_metric(key="accuracy", value=scores[1], step=dataset_count)
```

Fig 17: Using Mlflow's tracking API to log metrics and params

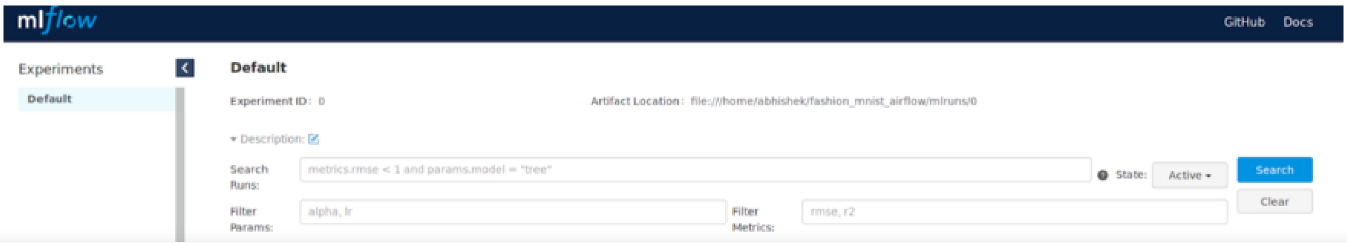
The best feature about ***mlflow*** is the ***dashboard*** it provides. It has a very intuitive UI and can be used efficiently for tracking our experiments. The ***dashboard*** can be started easily by simply hitting ***mlflow ui*** in your terminal as shown in Fig 18 below.



```
(tf2) ➜  fashion_mnist_airflow git:(dev) ✗ mlflow ui
[2019-11-10 13:51:20 +0530] [20237] [INFO] Starting gunicorn 19.9.0
[2019-11-10 13:51:20 +0530] [20237] [INFO] Listening at: http://127.0.0.1:5000 (20237)
[2019-11-10 13:51:20 +0530] [20237] [INFO] Using worker: sync
[2019-11-10 13:51:20 +0530] [20240] [INFO] Booting worker with pid: 20240
```

Fig 18 : Starting the mlflow ui from the command line

To access the dashboard, just type ***http://localhost:5000*** in your browser and hit enter.



Default

Experiment ID : 0

Artifact Location : file:///home/abhishek/fashion_mnist_airflow/mlruns/0

Description:

Search Runs: metrics.rmse < 1 and params.model = "tree"

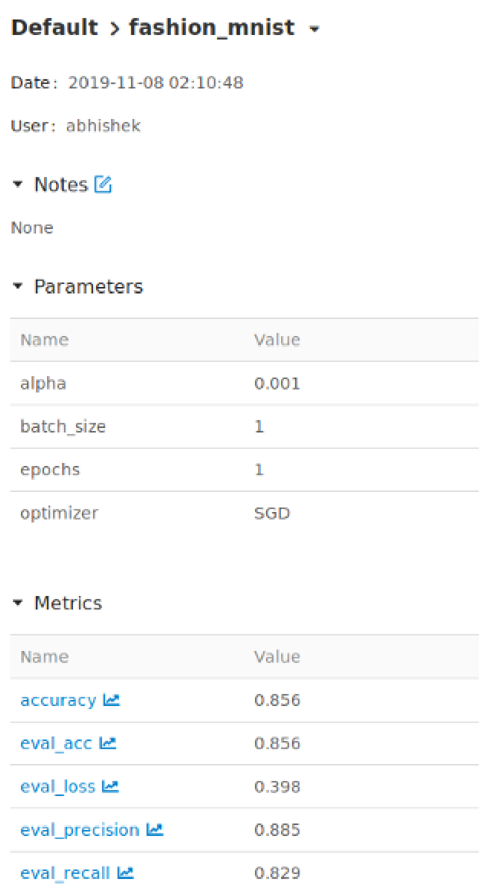
State: Active

Filter Params: alpha, lr

Filter Metrics: rmse, r2

Fig 19: Mlflow dashboard view.

Fig 19 shows how the dashboard looks like. Each ***MlFlow*** run is logged using a ***run ID*** and a ***run name***. The Parameters and the Metrics column log display the parameters and the metrics which were logged while we were training our model



Default › fashion_mnist

Date: 2019-11-08 02:10:48 Source: train.py

User: abhishek Duration: 2.3min

Notes

None

Parameters

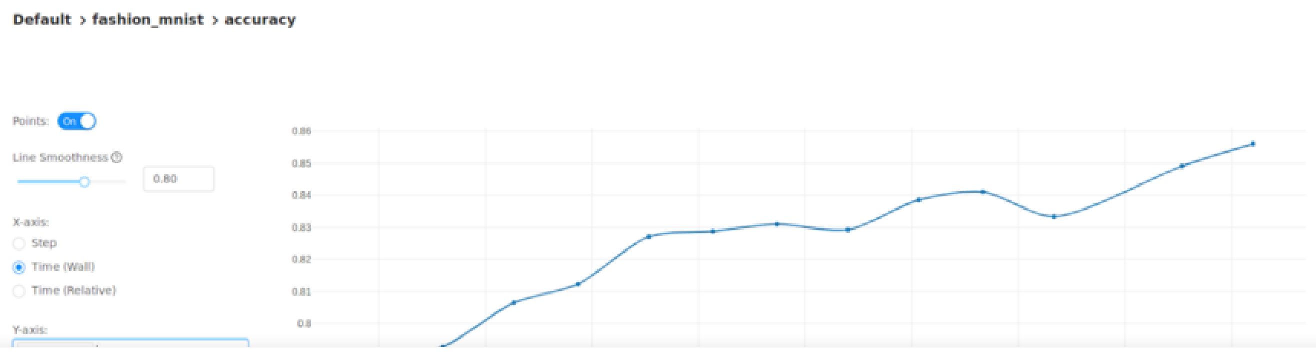
Name	Value
alpha	0.001
batch_size	1
epochs	1
optimizer	SGD

Metrics

Name	Value
accuracy	0.856
eval_acc	0.856
eval_loss	0.398
eval_precision	0.885
eval_recall	0.829

Fig 20: Individual run information

Further clicking on a particular run, takes us to another page where we can display all information about our run (Fig 20)



Default › fashion_mnist › accuracy

Points: On

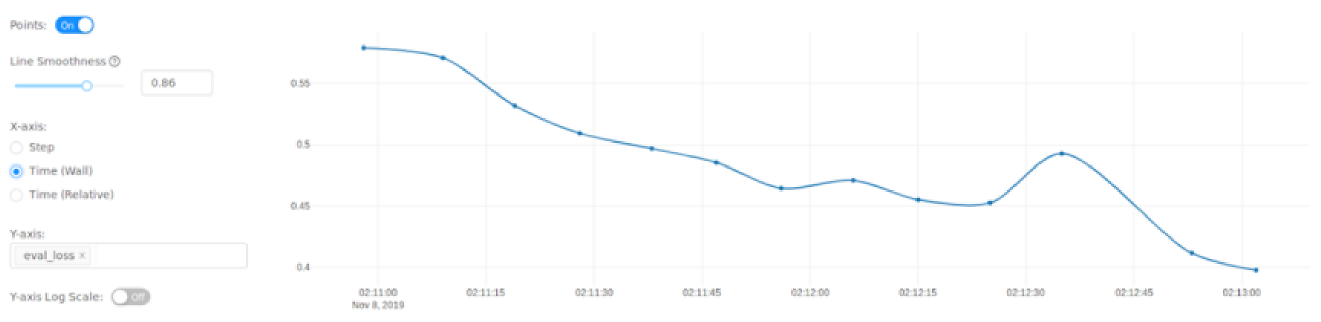
Line Smoothness 0.80

X-axis: Step [✓ Time (Wall)] Time (Relative)

Y-axis:

Fig 21a: Visualizing the test accuracy plot in mlflow

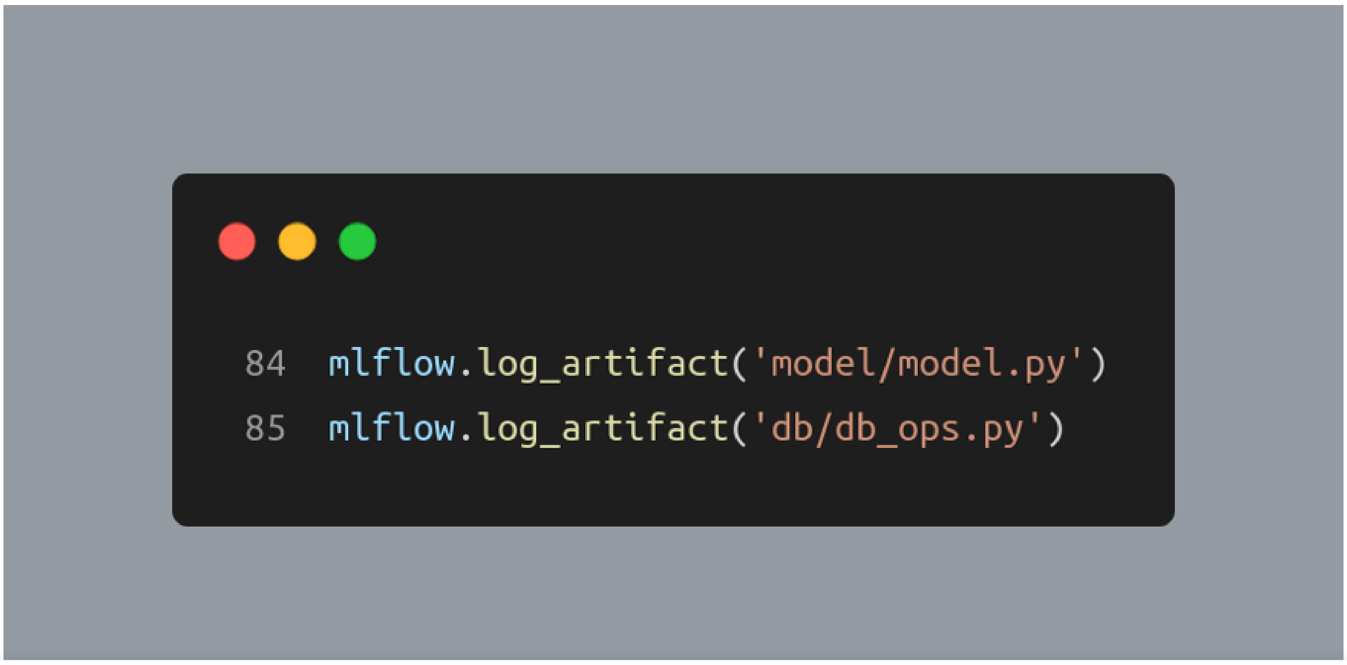
MlFlow provides us with this amazing feature to generate plots for our results. As you can see in Fig 21a, the test accuracy change can be visualized across different training datasets and time. We can also choose to display other metrics such as the ***eval*** loss, as shown in Fig 21b. The smoothness of the curve can also be controlled using the slider.



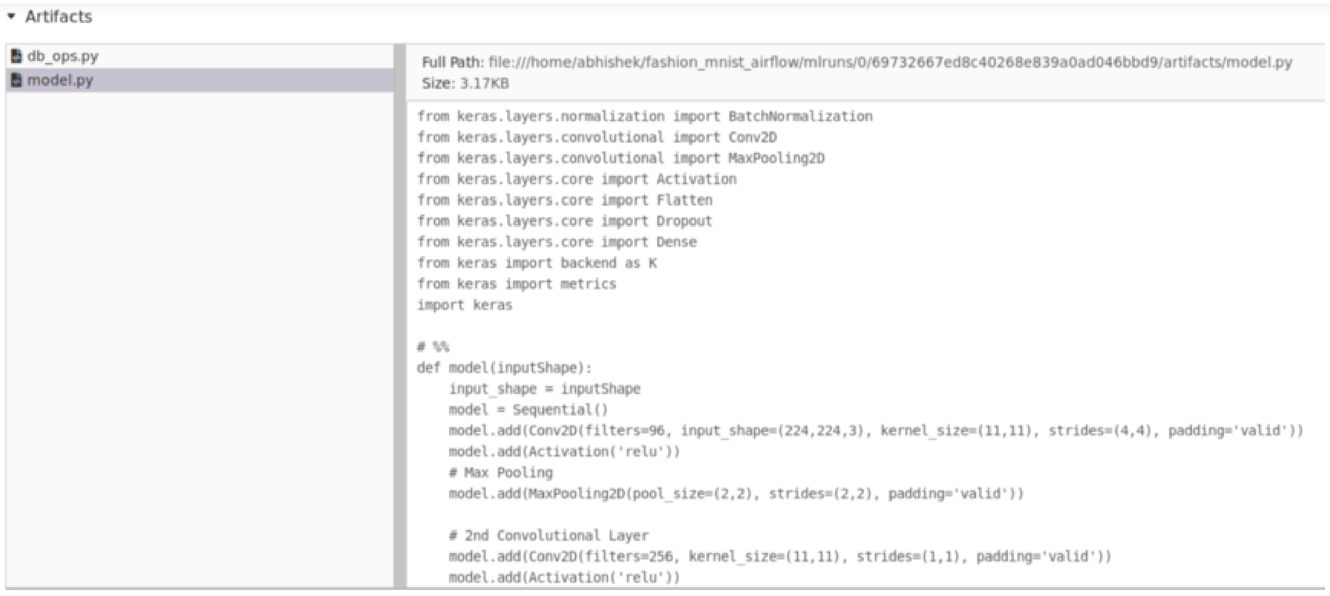
Points: On
Line Smoothness: 0.86
X-axis: ( ) Step [✓ Time (Wall)] ( ) Time (Relative)
Y-axis: eval_loss
Y-axis Log Scale: Off

Fig 21b: Visualizing the eval loss plot in mlflow

We can also log important files or scripts in our project to MlFlow using the ***mlflow.log_artifact*** command . Fig 22a shows how to use it in your training script and Fig 22b shows how it is displayed on the mlflow dashboard.



```
mlflow.log_artifact('model/model.py')
mlflow.log_artifact('db/db_ops.py')
```

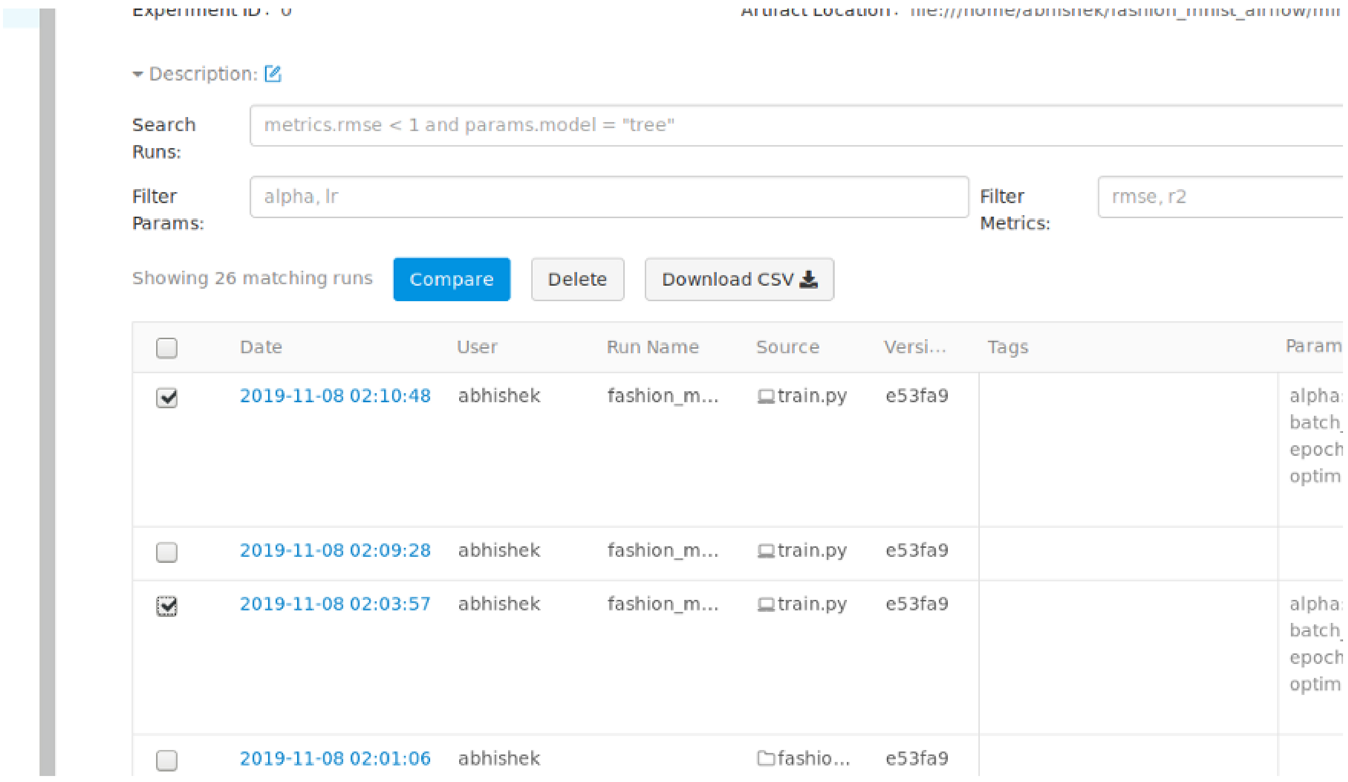


Full Path: file:///home/abhishek/fashion_mnist_airflow/mlruns/0/69732667ed8c40268e839a0ad046bbd9/artifacts/model.py
Size: 3.17KB

```
from keras.layers.normalization import BatchNormalization
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dropout
from keras.layers.core import Dense
from keras import backend as K
from keras import metrics
import keras

# %%
def model(inputShape):
    input_shape = inputShape
    model = Sequential()
    model.add(Conv2D(filters=96, input_shape=(224,224,3), kernel_size=(11,11), strides=(4,4), padding='valid'))
    model.add(Activation('relu'))
    # Max Pooling
    model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='valid'))

    # 2nd Convolutional Layer
    model.add(Conv2D(filters=256, kernel_size=(11,11), strides=(1,1), padding='valid'))
    model.add(Activation('relu'))
```

Fig 22b: model file and db_ops file logged as artifact on mlflow

MlFlow also allows users to compare two runs simultaneously and generate plots for it. Just tick the check-boxes against the runs you want to compare and press on the blue ***Compare*** button (Fig 23)


Experiment ID: 0
Artifact Location: file:///home/abhishek/fashion_mnist_airflow/mlr
Description:
Compare
Delete
Download CSV

Search Runs: metrics.rmse < 1 and params.model = "tree"

Filter Params: alpha, lr

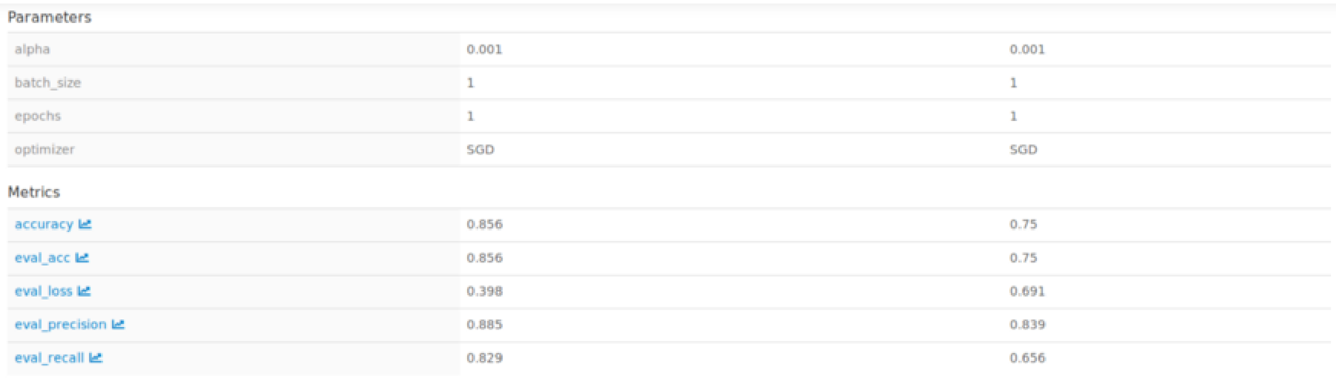
Filter Metrics: rmse, r2

Showing 26 matching runs

☐	Date	User	Run Name	Source	Versi...	Tags	Param
✓	2019-11-08 02:10:48	abhishek	fashion_m...	train.py	e53fa9		alpha batch epoch optim
☐	2019-11-08 02:09:28	abhishek	fashion_m...	train.py	e53fa9		
✓	2019-11-08 02:03:57	abhishek	fashion_m...	train.py	e53fa9		alpha batch epoch optim
☐	2019-11-08 02:01:06	abhishek		fashio...	e53fa9		

Fig 23: Mlflow dashboard which lists all the mlflow runs sequentially

Once you click on compare, another page pops up where all metrics and parameters of



Parameters		
alpha	0.001	0.001
batch_size	1	1
epochs	1	1
optimizer	SGD	SGD
Metrics		
accuracy	0.856	0.75
eval_acc	0.856	0.75
eval_loss	0.398	0.691
eval_precision	0.885	0.839
eval_recall	0.829	0.656

Fig 24a: Comparing multiple mlflow runs in parallel

The user can also choose to display metrics such as accuracy and loss in parallel charts as shown in Fig 24b.

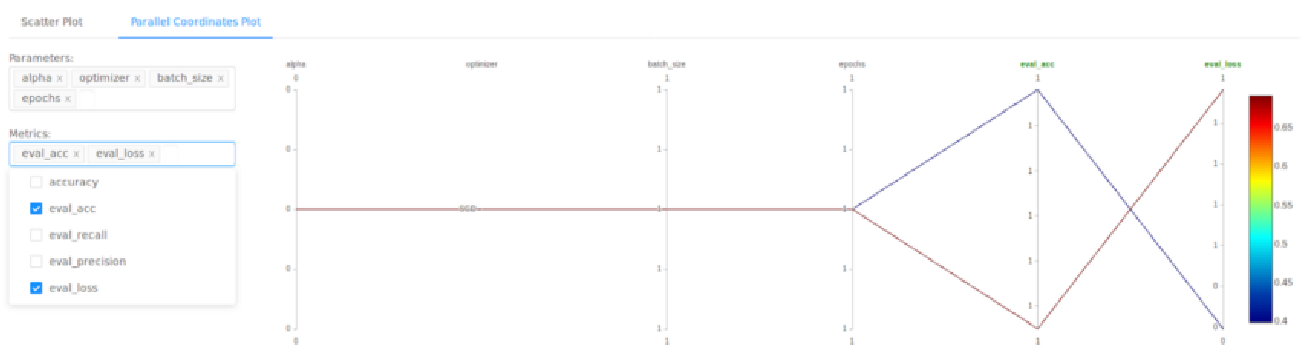
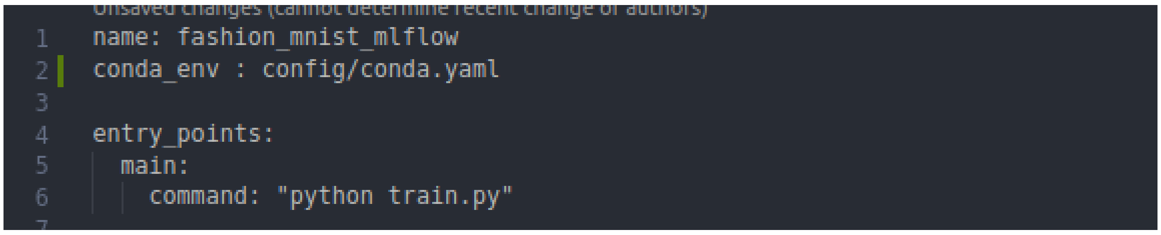


FIg 24b: Comparing multiple mlflow runs using visual plots based on metrics

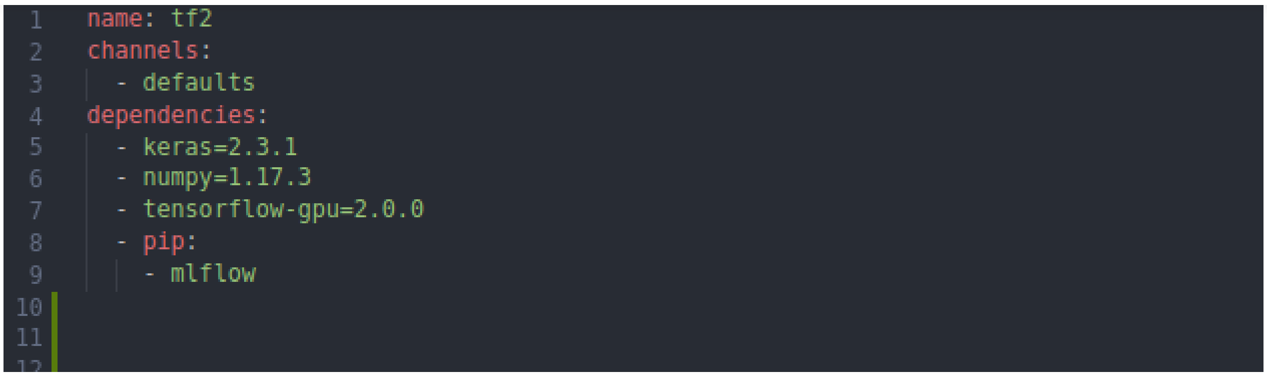
Users can add an MlFlow Project file (a text file in ***YAML*** syntax) to their MlFlow project allowing them to package their code better and run and recreate results from any machine. The MlFlow Project file for our ***fashion_mnsit*** project looks as shown below in Fig 25a



```
Unsaved changes (cannot determine recent change or authors)
name: fashion_mnist_mlflow
conda_env : config/conda.yaml

entry_points:
  main:
    command: "python train.py"
```

Fig 25a: MLfow project file containing entry points and also conda path

We can also specify a ***conda*** environment for our ***MlFlow*** project and specify a



```
name: tf2
channels:
  - defaults
dependencies:
  - keras=2.3.1
  - numpy=1.17.3
  - tensorflow-gpu=2.0.0
  - pip:
    - mlflow
```

Fig 25b: Conda file specifying packages and dependencies in the specific environment

Hence, with this we conclude our project. Developers face several on-demand requirements for monitoring metrics during training a neural network. These metrics can be extremely critical for predicting the output of their neural networks and are also critical in understanding how to modify a neural network for better performance. Traditionally when starting off with deep learning experiments, many developers are unaware of proper tools to help them. I hope this was piece of writing was helpful in understanding how deep learning experiments can be conducted in a proper manner during production when large numbers of datasets are needed to be managed and multiple training and evaluation instances are required to be monitored.

MlFlow also has multiple other features which is beyond the scope of this tutorial and can be covered later. For any information on how to use ***MlFlow*** one can head to the ***https://mlflow.org/***