COSC 202 Project

November 28, 2024

1 Data Gathering

We will start by getting the data from the 'fashion_dataset' directory.

[ ]:
```
!wget https://github.com/mohamed-12-4/DSAI--COSC-202-/raw/refs/heads/master/
↪fashion_dataset.zip
!unzip fashion_dataset
```

[19]:
```
import os
import glob
import pandas as pd
import cv2
import random
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

We start by defining a function that takes the path of folder and store the image path and label into a pandas dataframe

[20]:
```
def load_data(path: str) -> pd.DataFrame:
    """
    Load the data from the path and return a pandas dataframe with the images␣
↪path and labels.
    param path: str: path to the images directory
    """
    images = []
    labels = []
    for image_path in glob.glob(path + '/*/*.png'):
        images.append(image_path)
        labels.append(image_path.split('/')[-2])

    images_df = pd.DataFrame({'images_path': images, 'labels': labels})
    return images_df
```

[21]:
```
images_df = load_data('./fashion_dataset')
```

2 Data Exploration and Visualization

[22]:
```
images_df
```

[22]:
```
                          images_path labels
0      ./fashion_dataset/3/42968.png      3
1      ./fashion_dataset/3/28724.png      3
2      ./fashion_dataset/3/13600.png      3
3      ./fashion_dataset/3/32667.png      3
4      ./fashion_dataset/3/32859.png      3
…                               …     …
59995  ./fashion_dataset/5/49155.png      5
59996   ./fashion_dataset/5/3531.png      5
59997   ./fashion_dataset/5/6438.png      5
59998  ./fashion_dataset/5/58969.png      5
59999   ./fashion_dataset/5/6929.png      5

[60000 rows x 2 columns]
```

[23]:
```
images_df.labels = images_df.labels.astype(int)
images_df.labels.value_counts()
```

[23]:
```
labels
3    6000
4    6000
8    6000
7    6000
9    6000
0    6000
6    6000
2    6000
1    6000
5    6000
Name: count, dtype: int64
```

As we can see we have exactly 6,000 images per class which implies that the dataset is balanced

[24]:
```
images_df
```

[24]:
```
                          images_path  labels
0      ./fashion_dataset/3/42968.png       3
1      ./fashion_dataset/3/28724.png       3
2      ./fashion_dataset/3/13600.png       3
3      ./fashion_dataset/3/32667.png       3
4      ./fashion_dataset/3/32859.png       3
…                               …      …
59995  ./fashion_dataset/5/49155.png       5
```

```
59996    ./fashion_dataset/5/3531.png       5
59997    ./fashion_dataset/5/6438.png       5
59998   ./fashion_dataset/5/58969.png       5
59999    ./fashion_dataset/5/6929.png       5

[60000 rows x 2 columns]
```

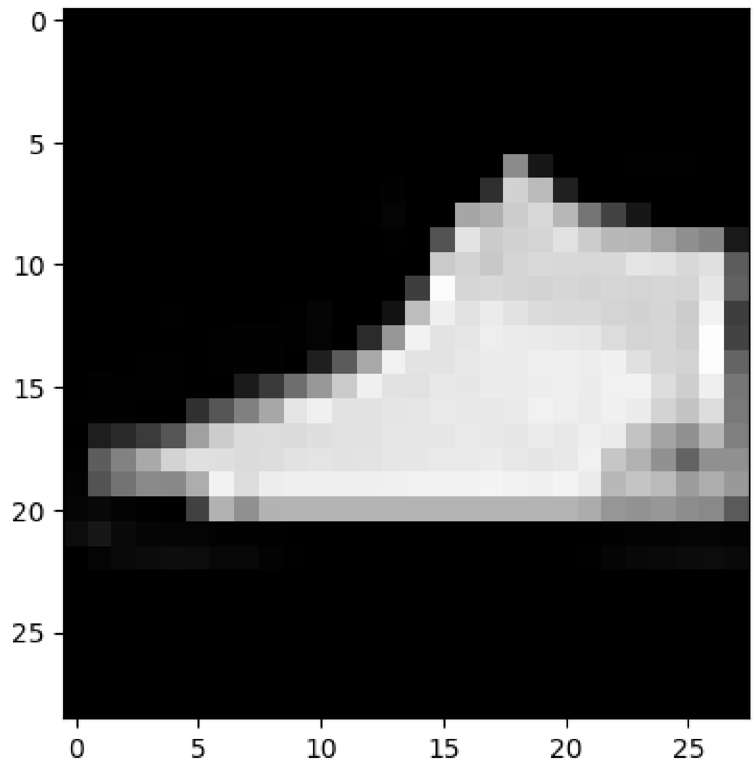
By inspection, we can convert each label to a name class for an easier interpretation of that data

[25]:
```python
labels_map = {
    0: 'T-shirt/top',
    1: 'Trouser',
    2: 'Pullover',
    3: 'Dress',
    4: 'Coat',
    5: 'Sandal',
    6: 'Shirt',
    7: 'Sneaker',
    8: 'Bag',
    9: 'Ankle boot'
}

def class_to_name(label) -> str:
    """
    Convert the label to the name of the class
    param label: int: label of the class
    """

    return labels_map[label]
```

We also created a funtion that views an image given it's path

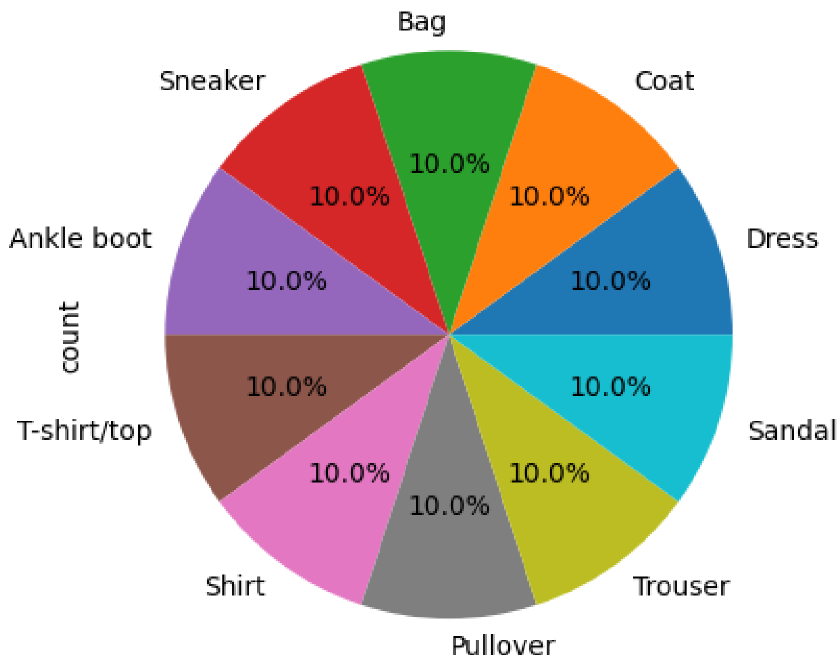
[26]:
```python
def view_image(path: str):
    """
    View the image from the path
    param path: str: path to the image
    """
    import matplotlib.pyplot as plt
    import matplotlib.image as mpimg
    img = cv2.imread(path)
    plt.imshow(img)

index = random.randint(0, len(images_df))
image_path = images_df.images_path[index]
label = images_df.labels[index]
print(class_to_name(label))
view_image(image_path)
```

Sneaker



[27]:
```
# plot a pie plot of the classes
images_df['class_name'] = images_df['labels'].apply(class_to_name)

images_df.class_name.value_counts().plot(kind='pie', autopct='%1.1f%%', )
```

[27]: <Axes: ylabel='count'>



3 Data Preprocessing

We start by utlising tf ImageDataGenerator, then we apply flow_from_dataframe funtion, this will go to each file path in the images dataframe and read it, then it will save it in a dataset, we also have the option to choose the batch size.

[28]:
```
def create_dataset(df: pd.DataFrame, batch_size: int=32, test=False) -> tf.data.
↪Dataset:
    """
    Create the dataset from the dataframe
    param df: pd.DataFrame: dataframe with the images path and labels
    param batch_size: int: batch size of the dataset
    """
    imagegen = tf.keras.preprocessing.image.ImageDataGenerator(rescale = 1/255.)
    if not test:
        dataset = imagegen.flow_from_dataframe(
            df,
            x_col='images_path',
            y_col='labels',
            target_size=(28, 28),
            class_mode='raw',
            batch_size=batch_size,
        )
```

```
    else:
        dataset = imagegen.flow_from_dataframe(
            df,
            x_col='images_path',
            y_col='labels',
            target_size=(28, 28),
            class_mode='raw',
            batch_size=1,
            shuffle=False
        )
    return dataset
```

[29]:
```
from sklearn.model_selection import train_test_split
train_df, test_df = train_test_split(images_df, test_size=0.2, random_state=42)
test_df, val_df = train_test_split(test_df, test_size=0.5, random_state=42)
```

Then we start by applying the create_dataset function to the train, test, validation splits of the dataframe

[30]:
```
dataset = create_dataset(images_df, 64)
train_dataset = create_dataset(train_df, 64)
test_dataset = create_dataset(test_df, 1, test=True)
val_dataset = create_dataset(val_df, 64)
```

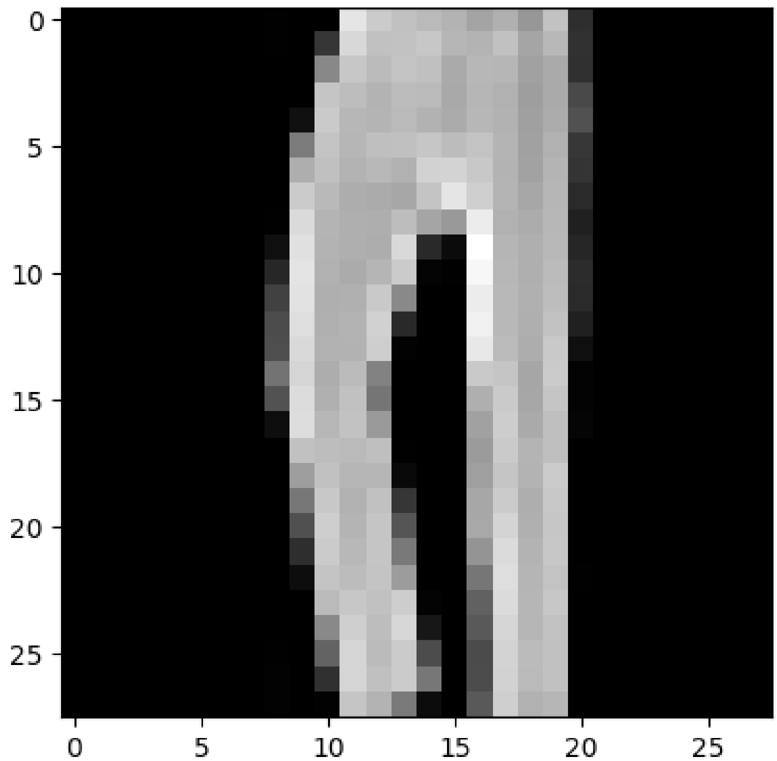
```
Found 60000 validated image filenames.
Found 48000 validated image filenames.
Found 6000 validated image filenames.
Found 6000 validated image filenames.
```

[31]:
```
len(dataset) # 938 batches each batch contains 64 images
```

[31]: 938

[39]:
```
plt.imshow(dataset[0][0][0])
```

[39]: <matplotlib.image.AxesImage at 0x7f024fa19760>



4 Simple NN (Only dense layers)

We start by defining a simple model that only contains Desne layers

[ ]:
```
Dense_model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28, 3)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
Dense_model.summary()
```

```
Model: "sequential_3"


 Layer (type)                       Output Shape                    ␣
↪Param #

 flatten_3 (Flatten)                (None, 2352)                      ␣
↪  0
```

```
dense_8 (Dense)                          (None, 128)                       ␣
↪301,184

dense_9 (Dense)                          (None, 64)                         ␣
↪8,256

dense_10 (Dense)                         (None, 10)                          ␣
↪650
```

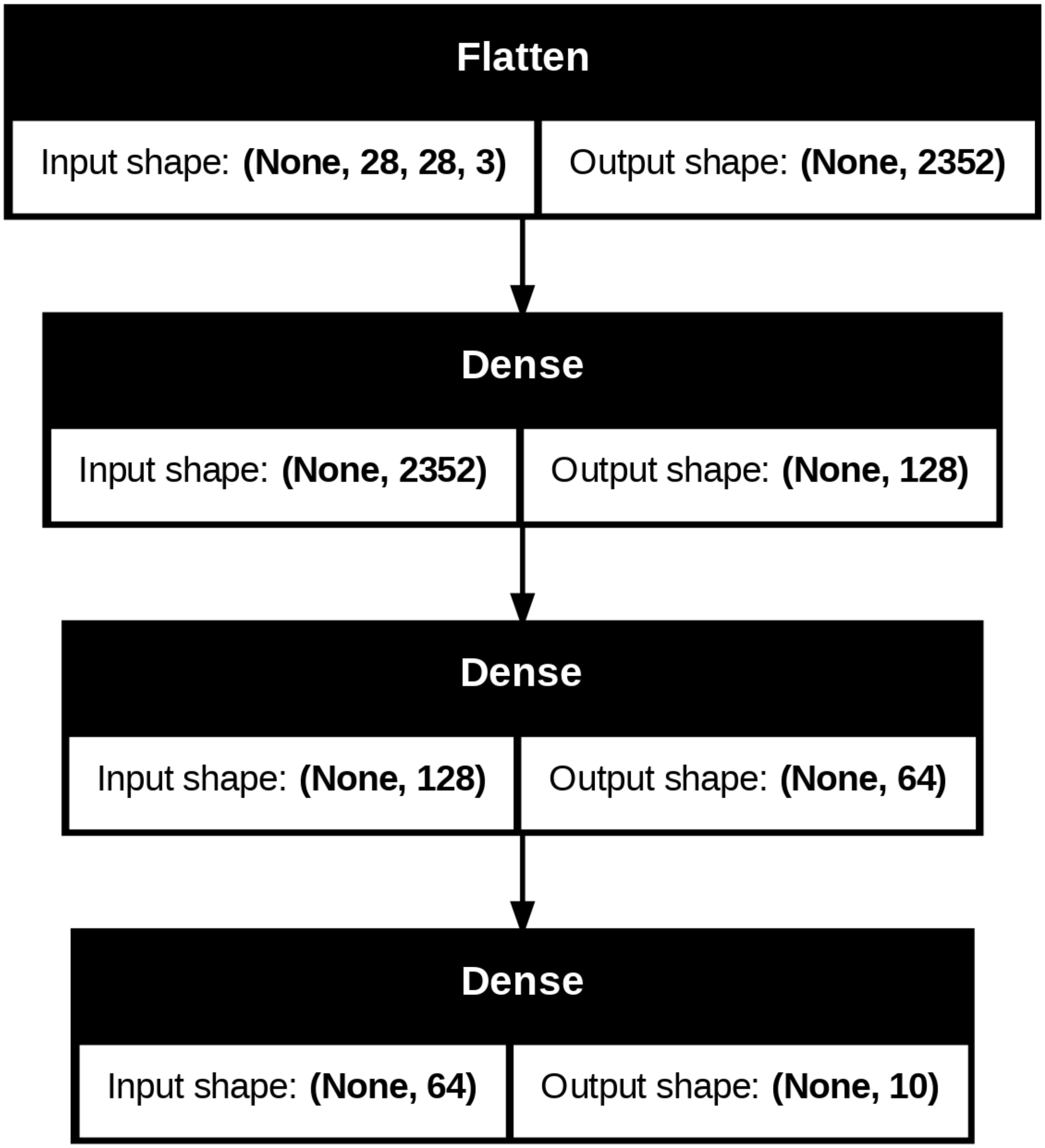
Total params: 310,090 (1.18 MB)

Trainable params: 310,090 (1.18 MB)

Non-trainable params: 0 (0.00 B)

[ ]:
```
tf.keras.utils.plot_model(Dense_model, show_shapes=True)
```

[ ]:



[ ]:
```
# adding a callback to prevent the model from overfitting
callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5,␣
 ↪restore_best_weights=True)
Dense_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
 ↪metrics=['accuracy'])
history = Dense_model.fit(train_dataset, epochs=100, callbacks=[callback],␣
 ↪validation_data=val_dataset)
```

```
Epoch 1/100
750/750              19s 23ms/step -
```

```
accuracy: 0.7602 - loss: 0.6916 - val_accuracy: 0.8510 - val_loss: 0.4319
Epoch 2/100
750/750            22s 26ms/step -
accuracy: 0.8542 - loss: 0.4024 - val_accuracy: 0.8562 - val_loss: 0.3952
Epoch 3/100
750/750            16s 22ms/step -
accuracy: 0.8692 - loss: 0.3575 - val_accuracy: 0.8658 - val_loss: 0.3747
Epoch 4/100
750/750            21s 22ms/step -
accuracy: 0.8779 - loss: 0.3340 - val_accuracy: 0.8715 - val_loss: 0.3449
Epoch 5/100
750/750            17s 23ms/step -
accuracy: 0.8791 - loss: 0.3202 - val_accuracy: 0.8700 - val_loss: 0.3567
Epoch 6/100
750/750            17s 23ms/step -
accuracy: 0.8851 - loss: 0.3037 - val_accuracy: 0.8738 - val_loss: 0.3505
Epoch 7/100
750/750            20s 23ms/step -
accuracy: 0.8902 - loss: 0.2937 - val_accuracy: 0.8788 - val_loss: 0.3398
Epoch 8/100
750/750            20s 22ms/step -
accuracy: 0.8912 - loss: 0.2859 - val_accuracy: 0.8720 - val_loss: 0.3629
Epoch 9/100
750/750            20s 26ms/step -
accuracy: 0.8980 - loss: 0.2708 - val_accuracy: 0.8672 - val_loss: 0.3667
Epoch 10/100
750/750            25s 33ms/step -
accuracy: 0.8973 - loss: 0.2714 - val_accuracy: 0.8837 - val_loss: 0.3300
Epoch 11/100
750/750            25s 33ms/step -
accuracy: 0.9024 - loss: 0.2560 - val_accuracy: 0.8812 - val_loss: 0.3328
Epoch 12/100
750/750            19s 25ms/step -
accuracy: 0.9077 - loss: 0.2435 - val_accuracy: 0.8847 - val_loss: 0.3293
Epoch 13/100
750/750            21s 28ms/step -
accuracy: 0.9089 - loss: 0.2406 - val_accuracy: 0.8823 - val_loss: 0.3397
Epoch 14/100
750/750            39s 25ms/step -
accuracy: 0.9131 - loss: 0.2321 - val_accuracy: 0.8845 - val_loss: 0.3406
Epoch 15/100
750/750            18s 23ms/step -
accuracy: 0.9113 - loss: 0.2294 - val_accuracy: 0.8840 - val_loss: 0.3404
Epoch 16/100
750/750            16s 21ms/step -
accuracy: 0.9128 - loss: 0.2245 - val_accuracy: 0.8815 - val_loss: 0.3610
Epoch 17/100
750/750            17s 23ms/step -
```

```
accuracy: 0.9175 - loss: 0.2198 - val_accuracy: 0.8798 - val_loss: 0.3585
```

As we can see the mode has a validation accuracy of around 88%

5 Convolutional NN

6 Convolutional NN w/MaxPooling layers

Now, we try Convolutional Neural Networks on the data set, we start by a simple one of only Conv and MaxPooling Layers

[ ]:
```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation='leaky_relu',␣
 ↪input_shape=(28, 28, 3)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='leaky_relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='leaky_relu'),
    tf.keras.layers.Conv2D(64, (3, 3), activation='leaky_relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation='leaky_relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
model.summary()
```

```
/usr/local/lib/python3.10/dist-
packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Model: "sequential_1"


 Layer (type)                    Output Shape                           ␣
↪Param #

 conv2d (Conv2D)                 (None, 26, 26, 32)                       ␣
↪896

 conv2d_1 (Conv2D)               (None, 24, 24, 64)                     ␣
↪18,496

 max_pooling2d (MaxPooling2D)    (None, 12, 12, 64)                       ␣
↪  0
```

```
conv2d_2 (Conv2D)                    (None, 10, 10, 64)              ␣
↪36,928

conv2d_3 (Conv2D)                    (None, 8, 8, 64)                ␣
↪36,928

max_pooling2d_1 (MaxPooling2D)       (None, 4, 4, 64)                   ␣
↪  0

flatten_1 (Flatten)                  (None, 1024)                       ␣
↪  0

dense_3 (Dense)                      (None, 64)                      ␣
↪65,600

dense_4 (Dense)                      (None, 10)                         ␣
↪650
```

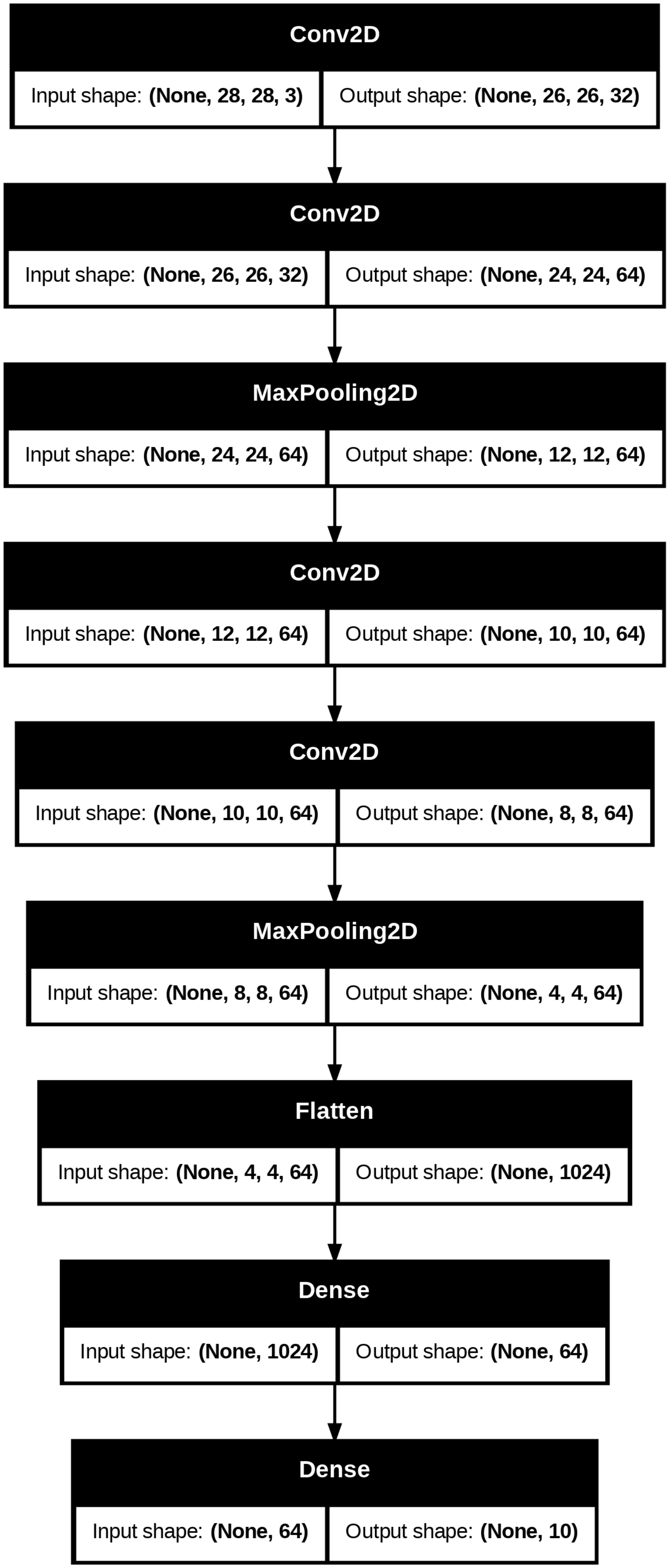
Total params: 159,498 (623.04 KB)

Trainable params: 159,498 (623.04 KB)

Non-trainable params: 0 (0.00 B)

[ ]:
```
tf.keras.utils.plot_model(model, show_shapes=True)
```

[ ]:



Conv2D
Input shape: (None, 28, 28, 3)
Output shape: (None, 26, 26, 32)
Conv2D
Input shape: (None, 26, 26, 32)
Output shape: (None, 24, 24, 64)
MaxPooling2D
Input shape: (None, 24, 24, 64)
Output shape: (None, 12, 12, 64)
Conv2D
Input shape: (None, 12, 12, 64)
Output shape: (None, 10, 10, 64)
Conv2D
Input shape: (None, 10, 10, 64)
Output shape: (None, 8, 8, 64)
MaxPooling2D
Input shape: (None, 8, 8, 64)
Output shape: (None, 4, 4, 64)
Flatten
Input shape: (None, 4, 4, 64)
Output shape: (None, 1024)
Dense
Input shape: (None, 1024)
Output shape: (None, 64)
Dense
Input shape: (None, 64)
Output shape: (None, 10)

[ ]:
```
callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5,␣
 ↪restore_best_weights=True)
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
 ↪metrics=['accuracy'])
history = model.fit(train_dataset, epochs=100, callbacks=[callback],␣
 ↪validation_data=val_dataset)
```

```
Epoch 1/100
```

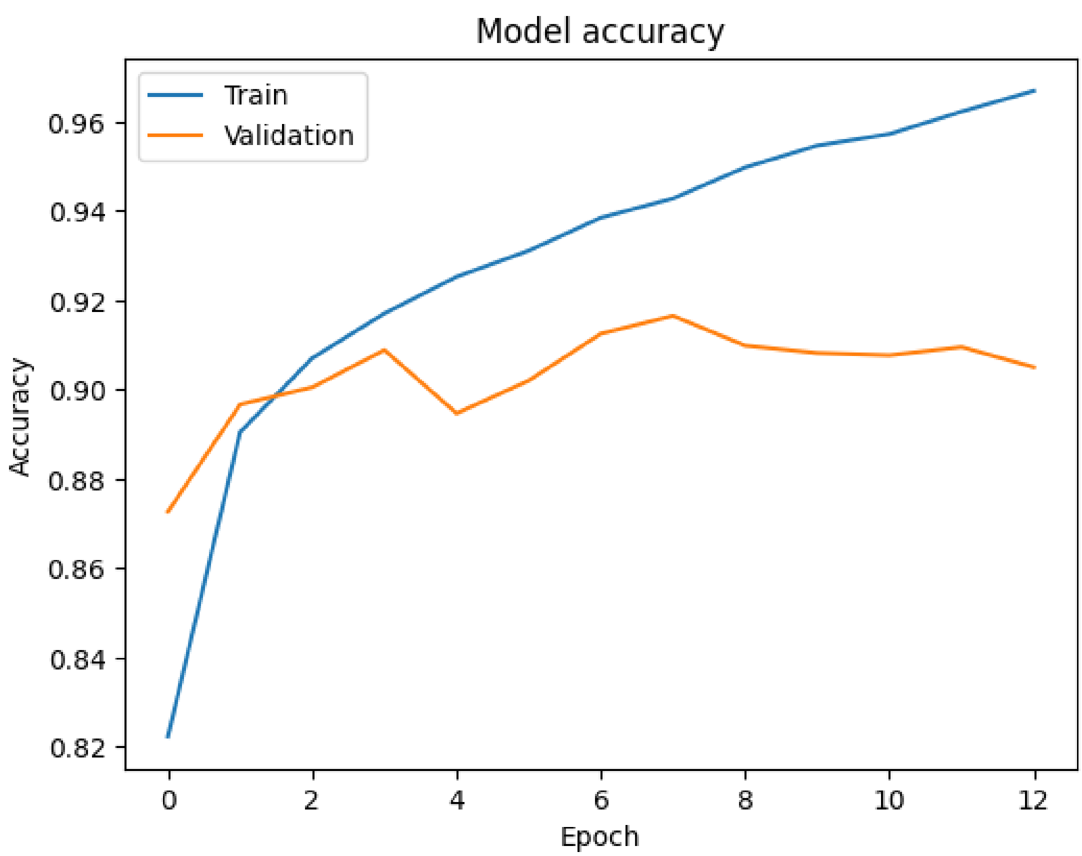
```
/usr/local/lib/python3.10/dist-
packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:122:
UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in
its constructor. `**kwargs` can include `workers`, `use_multiprocessing`,
`max_queue_size`. Do not pass these arguments to `fit()`, as they will be
ignored.
  self._warn_if_super_not_called()
```

```
750/750               25s 26ms/step -
accuracy: 0.7476 - loss: 0.6945 - val_accuracy: 0.8727 - val_loss: 0.3612
Epoch 2/100
750/750               36s 25ms/step -
accuracy: 0.8865 - loss: 0.3103 - val_accuracy: 0.8967 - val_loss: 0.2875
Epoch 3/100
750/750               19s 25ms/step -
accuracy: 0.9057 - loss: 0.2585 - val_accuracy: 0.9005 - val_loss: 0.2771
Epoch 4/100
750/750               19s 23ms/step -
accuracy: 0.9186 - loss: 0.2271 - val_accuracy: 0.9088 - val_loss: 0.2610
Epoch 5/100
750/750               22s 24ms/step -
accuracy: 0.9253 - loss: 0.1995 - val_accuracy: 0.8947 - val_loss: 0.2823
Epoch 6/100
750/750               17s 23ms/step -
accuracy: 0.9333 - loss: 0.1808 - val_accuracy: 0.9020 - val_loss: 0.2874
Epoch 7/100
750/750               21s 24ms/step -
accuracy: 0.9399 - loss: 0.1653 - val_accuracy: 0.9125 - val_loss: 0.2562
Epoch 8/100
750/750               17s 23ms/step -
accuracy: 0.9434 - loss: 0.1496 - val_accuracy: 0.9165 - val_loss: 0.2507
Epoch 9/100
750/750               21s 24ms/step -
accuracy: 0.9531 - loss: 0.1284 - val_accuracy: 0.9098 - val_loss: 0.2877
Epoch 10/100
750/750               17s 23ms/step -
accuracy: 0.9567 - loss: 0.1166 - val_accuracy: 0.9082 - val_loss: 0.2751
```

```
Epoch 11/100
750/750              18s 24ms/step -
accuracy: 0.9600 - loss: 0.1045 - val_accuracy: 0.9077 - val_loss: 0.3118
Epoch 12/100
750/750              18s 24ms/step -
accuracy: 0.9652 - loss: 0.0935 - val_accuracy: 0.9095 - val_loss: 0.3285
Epoch 13/100
750/750              20s 23ms/step -
accuracy: 0.9697 - loss: 0.0831 - val_accuracy: 0.9050 - val_loss: 0.3681
```

We noticed that this model perform slightly better on the data acheving around 90% in validation accuracy

[ ]:
```python
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



7 Convolutional NN + Regulization / Normalization layers

Now we Try a better one that have an extra Regualization (Dropout layers) and Normalization Layers. These layers will prevent the model from overfitting.

[ ]:
```
CNNR_model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation='leaky_relu',␣
 ↪input_shape=(28, 28, 3)),
    tf.keras.layers.Conv2D(32, (3, 3), activation='leaky_relu'),
    tf.keras.layers.Normalization(),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='leaky_relu'),
    tf.keras.layers.Conv2D(64, (3, 3), activation='leaky_relu'),
    tf.keras.layers.Normalization(),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation='leaky_relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
CNNR_model.summary()
```

```
/usr/local/lib/python3.10/dist-
packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Model: "sequential_7"


 Layer (type)                         Output Shape                        ␣
↪Param #

 conv2d_43 (Conv2D)                   (None, 26, 26, 32)                      ␣
↪896

 conv2d_44 (Conv2D)                   (None, 24, 24, 32)                    ␣
↪9,248

 normalization (Normalization)        (None, 24, 24, 32)                      ␣
↪ 65

 dropout (Dropout)                    (None, 24, 24, 32)                      ␣
↪  0
```

```
max_pooling2d_17 (MaxPooling2D)        (None, 12, 12, 32)                  ␣
↪  0

conv2d_45 (Conv2D)                     (None, 10, 10, 64)                ␣
↪18,496

conv2d_46 (Conv2D)                     (None, 8, 8, 64)                  ␣
↪36,928

normalization_1 (Normalization)        (None, 8, 8, 64)                    ␣
↪129

dropout_1 (Dropout)                    (None, 8, 8, 64)                    ␣
↪  0

max_pooling2d_18 (MaxPooling2D)        (None, 4, 4, 64)                    ␣
↪  0

flatten_7 (Flatten)                    (None, 1024)                        ␣
↪  0

dense_20 (Dense)                       (None, 64)                        ␣
↪65,600

dense_21 (Dense)                       (None, 10)                          ␣
↪650
```

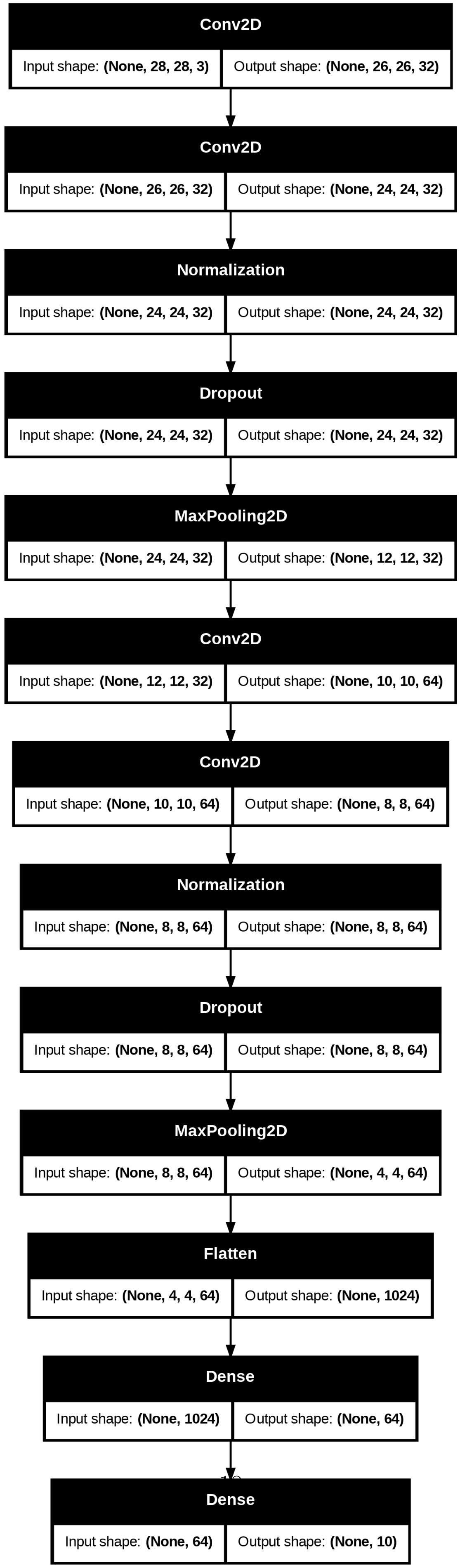
Total params: 132,012 (515.68 KB)

Trainable params: 131,818 (514.91 KB)

Non-trainable params: 194 (784.00 B)

[ ]:
```
tf.keras.utils.plot_model(CNNR_model, show_shapes=True)
```

[ ]:


Conv2D
Input shape: (None, 28, 28, 3)
Output shape: (None, 26, 26, 32)
Conv2D
Input shape: (None, 26, 26, 32)
Output shape: (None, 24, 24, 32)
Normalization
Input shape: (None, 24, 24, 32)
Output shape: (None, 24, 24, 32)
Dropout
Input shape: (None, 24, 24, 32)
Output shape: (None, 24, 24, 32)
MaxPooling2D
Input shape: (None, 24, 24, 32)
Output shape: (None, 12, 12, 32)
Conv2D
Input shape: (None, 12, 12, 32)
Output shape: (None, 10, 10, 64)
Conv2D
Input shape: (None, 10, 10, 64)
Output shape: (None, 8, 8, 64)
Normalization
Input shape: (None, 8, 8, 64)
Output shape: (None, 8, 8, 64)
Dropout
Input shape: (None, 8, 8, 64)
Output shape: (None, 8, 8, 64)
MaxPooling2D
Input shape: (None, 8, 8, 64)
Output shape: (None, 4, 4, 64)
Flatten
Input shape: (None, 4, 4, 64)
Output shape: (None, 1024)
Dense
Input shape: (None, 1024)
Output shape: (None, 64)
Dense
Input shape: (None, 64)
Output shape: (None, 10)

[ ]:
```
callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5,␣
 ↪restore_best_weights=True)
CNNR_model.compile(optimizer='nadam', loss='sparse_categorical_crossentropy',␣
 ↪metrics=['accuracy'])
history = CNNR_model.fit(train_dataset, epochs=100, callbacks=[callback],␣
 ↪validation_data=val_dataset)
```

```
Epoch 1/100
750/750              24s 26ms/step -
accuracy: 0.7501 - loss: 0.6978 - val_accuracy: 0.8693 - val_loss: 0.3765
Epoch 2/100
750/750              37s 25ms/step -
accuracy: 0.8802 - loss: 0.3223 - val_accuracy: 0.8880 - val_loss: 0.3129
Epoch 3/100
750/750              20s 25ms/step -
accuracy: 0.8980 - loss: 0.2743 - val_accuracy: 0.9002 - val_loss: 0.2799
Epoch 4/100
750/750              19s 25ms/step -
accuracy: 0.9084 - loss: 0.2476 - val_accuracy: 0.9002 - val_loss: 0.2788
Epoch 5/100
750/750              18s 23ms/step -
accuracy: 0.9145 - loss: 0.2320 - val_accuracy: 0.9097 - val_loss: 0.2520
Epoch 6/100
750/750              18s 23ms/step -
accuracy: 0.9229 - loss: 0.2164 - val_accuracy: 0.9075 - val_loss: 0.2609
Epoch 7/100
750/750              21s 23ms/step -
accuracy: 0.9238 - loss: 0.2027 - val_accuracy: 0.9122 - val_loss: 0.2487
Epoch 8/100
750/750              22s 25ms/step -
accuracy: 0.9296 - loss: 0.1903 - val_accuracy: 0.9125 - val_loss: 0.2508
Epoch 9/100
750/750              17s 23ms/step -
accuracy: 0.9353 - loss: 0.1751 - val_accuracy: 0.9147 - val_loss: 0.2432
Epoch 10/100
750/750              21s 24ms/step -
accuracy: 0.9365 - loss: 0.1675 - val_accuracy: 0.9092 - val_loss: 0.2655
Epoch 11/100
750/750              18s 24ms/step -
accuracy: 0.9415 - loss: 0.1561 - val_accuracy: 0.9158 - val_loss: 0.2395
Epoch 12/100
750/750              17s 23ms/step -
accuracy: 0.9435 - loss: 0.1505 - val_accuracy: 0.9153 - val_loss: 0.2566
Epoch 13/100
750/750              18s 24ms/step -
```

```
accuracy: 0.9481 - loss: 0.1405 - val_accuracy: 0.9137 - val_loss: 0.2683
Epoch 14/100
750/750              17s 23ms/step -
accuracy: 0.9491 - loss: 0.1362 - val_accuracy: 0.9092 - val_loss: 0.2682
Epoch 15/100
750/750              17s 23ms/step -
accuracy: 0.9518 - loss: 0.1281 - val_accuracy: 0.9138 - val_loss: 0.2662
Epoch 16/100
750/750              18s 24ms/step -
accuracy: 0.9503 - loss: 0.1323 - val_accuracy: 0.9155 - val_loss: 0.2730
```

We can see that this model is better than the previous two with 91.5% calidation accuracy

[ ]:
```
CNNR_model.evaluate(test_dataset)
```

```
94/94            2s 18ms/step -
accuracy: 0.9234 - loss: 0.2260
```

[ ]:
```
[0.23158198595046997, 0.9211666584014893]
```

We will choose the last model as our best one

Lastly we save the model to be tested on real-life data on another notebook file

[ ]:
```
import pickle
with open('model.pkl', 'wb') as f:
    pickle.dump(CNNR_model, f)
```

8 Test

[10]:
```
import pickle
model = pickle.load(open('model.pkl', 'rb'))
```

```
W0000 00:00:1732646876.483687   37084 gpu_device.cc:2344] Cannot dlopen some GPU
libraries. Please make sure the missing libraries mentioned above are installed
properly if you would like to use GPU. Follow the guide at
https://www.tensorflow.org/install/gpu for how to download and setup the
required libraries for your platform.
Skipping registering GPU devices…
```

[2]:
```
model.summary()
```

```
Model: "sequential_7"


 Layer (type)                      Output Shape                  Param #

 conv2d_43 (Conv2D)                (None, 26, 26, 32)                896
```

conv2d_44 (Conv2D)	(None, 24, 24, 32)	9,248
normalization (Normalization)	(None, 24, 24, 32)	65
dropout (Dropout)	(None, 24, 24, 32)	0
max_pooling2d_17 (MaxPooling2D)	(None, 12, 12, 32)	0
conv2d_45 (Conv2D)	(None, 10, 10, 64)	18,496
conv2d_46 (Conv2D)	(None, 8, 8, 64)	36,928
normalization_1 (Normalization)	(None, 8, 8, 64)	129
dropout_1 (Dropout)	(None, 8, 8, 64)	0
max_pooling2d_18 (MaxPooling2D)	(None, 4, 4, 64)	0
flatten_7 (Flatten)	(None, 1024)	0
dense_20 (Dense)	(None, 64)	65,600
dense_21 (Dense)	(None, 10)	650

Total params: 395,651 (1.51 MB)

Trainable params: 131,818 (514.91 KB)

Non-trainable params: 194 (784.00 B)

Optimizer params: 263,639 (1.01 MB)

[ ]:
```
import numpy as np
def model_predict(model, data):
    """
    Predict the data using the model
    param model: the model to predict the data
    param data: the data to be used for prediction
    """
    predictions = model.predict(data)
    return np.argmax(predictions, axis=1)
```

```python
def compare_predictions(predictions, labels):
    """
    Compare the predictions with the labels and print the accuracy
    param predictions: the predictions of the model
    param labels: the labels of the data
    """
    for i in predictions:
        print('Prediction:', class_to_name(i), end=' ')
        print('Label:', class_to_name(labels[i]))

    print('Accuracy:', np.mean(predictions == labels))
```

[42]:
```python
model.evaluate(test_dataset)
```

```
94/94            2s 17ms/step -
accuracy: 0.9471 - loss: 0.1382
```

[42]: [0.1509178876876831, 0.9453333616256714]

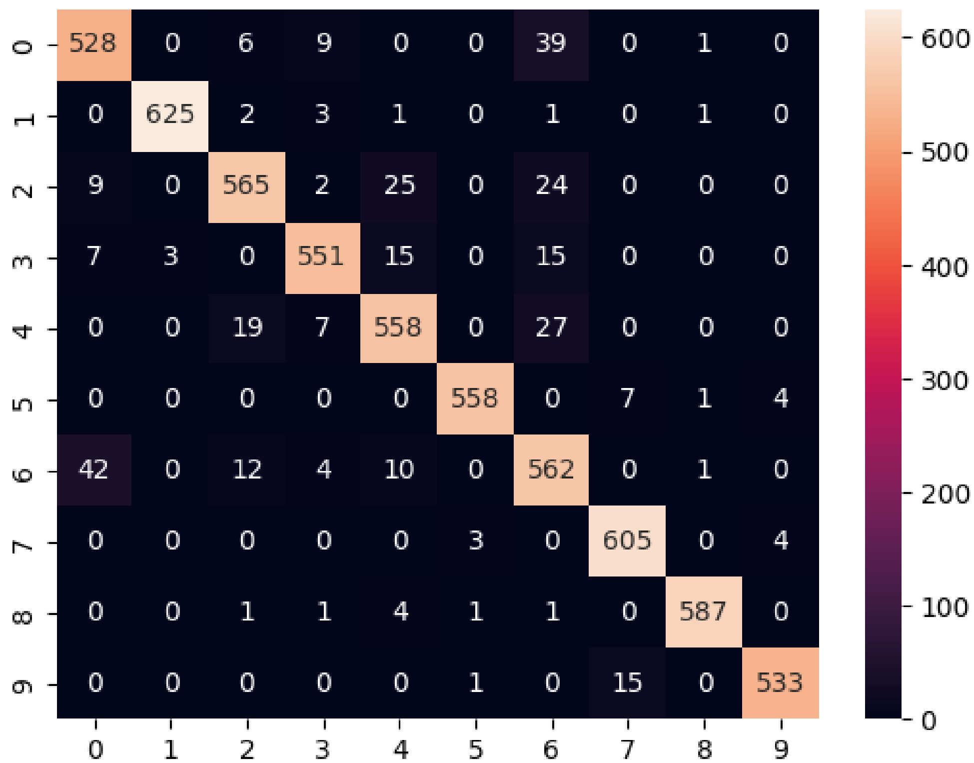
Now Applying the model on test data we get 94% accuracy

8.1 Confusion matrix

[ ]:
```python
# create confusion matrix
from sklearn.metrics import confusion_matrix
import seaborn as sns


y_pred = model_predict(model, test_dataset)
y_true = test_df.labels.values
#
sns.heatmap(confusion_matrix(y_true, y_pred), annot=True, fmt='d')
```

```
6000/6000              12s 2ms/step
```

[ ]: <Axes: >



0
1
2
3
4
5
6
7
8
9
0 100 200 300 400 500 600
528 0 6 9 0 0 39 0 1 0
0 625 2 3 1 0 1 0 1 0
9 0 565 2 25 0 24 0 0 0
7 3 0 551 15 0 15 0 0 0
0 0 19 7 558 0 27 0 0 0
0 0 0 0 0 558 0 7 1 4
42 0 12 4 10 0 562 0 1 0
0 0 0 0 0 3 0 605 0 4
0 0 1 1 4 1 1 0 587 0
0 0 0 0 0 1 0 15 0 533