University of Tehran

Electrical and Computer Engineering Department

Neural Networks and Deep Learning

Mini-Project 1

Name	Danial Saeedi
Student No	810198571
Date	Monday, November 22, 2021

Table of contents

1. Question 1 - CNN on MNIST Fashion Dataset

Visualizing the Image Data

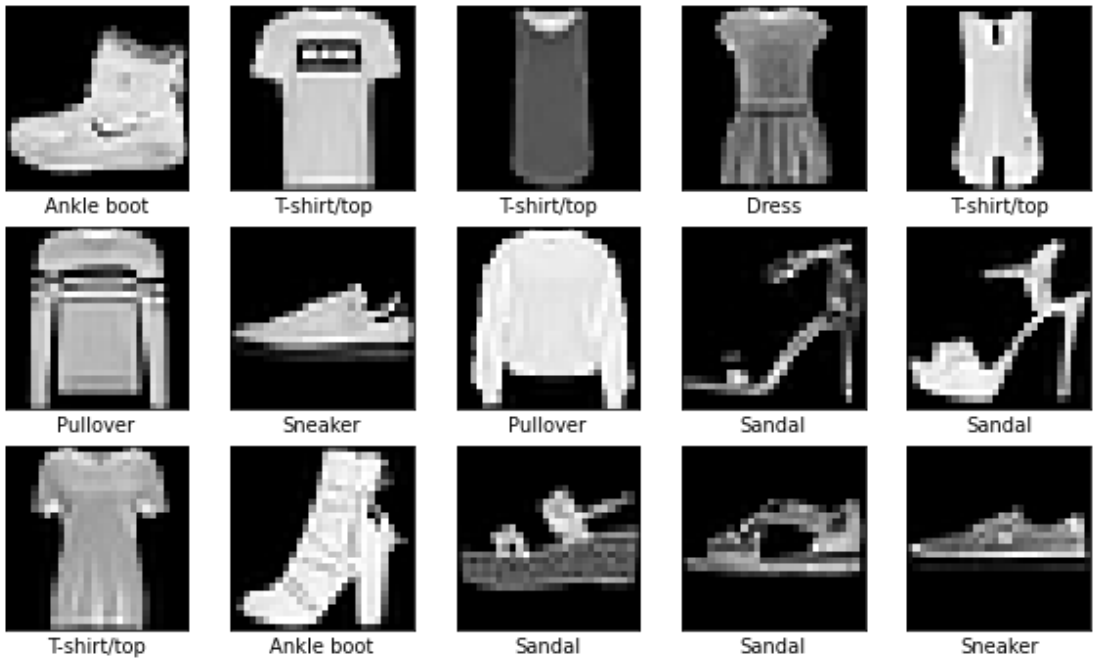


Figure 1 Visualizing the dataset

Pre-processing Data

We first need to make sure the labels will be understandable by our CNN.

Labels

As you can see, labels are categories of numbers. We need to translate this to be one hot encoded so our CNN model can understand it, otherwise it will think this is some sort of regression problem on a continuous axis.

```
In [ ]: # One Hot Encoded
        y_cat_test = to_categorical(y_test,number_of_categories)
        y_cat_train = to_categorical(y_train,number_of_categories)
        y_cat_train
```

```
Out[16]: array([[0., 0., 0., ..., 0., 0., 1.],
                [1., 0., 0., ..., 0., 0., 0.],
                [1., 0., 0., ..., 0., 0., 0.],
                ...,
                [0., 0., 0., ..., 0., 0., 0.],
                [1., 0., 0., ..., 0., 0., 0.],
                [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

Normalizing the images

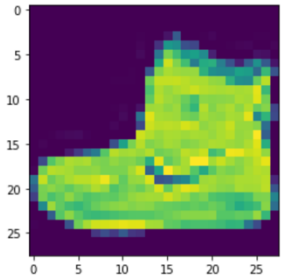
Each pixel value is between 0 and 255. So we can divide each pixel by 255 to normalize it.

```
In [ ]: X_train = X_train/255
        X_test = X_test/255
```

```
In [ ]: scaled_single = X_train[0]
```

```
In [ ]: plt.imshow(scaled_single)
```

```
Out[23]: <matplotlib.image.AxesImage at 0x7f7c19a4b8d0>
```



Flattening the images

We need to flatten the images before we can feed it to the Multi-Layer Perceptron Model.

```
X_train_flattened = X_train.reshape(60000, 784)
X_test_flattened = X_test.reshape(10000, 784)
```

MLP with two hidden layers

```
In [105]: def define_MLP_model(activation_function = 'relu',loss_func = 'categorical_crossentropy') :
            model = Sequential()

            # First Hidden Layer
            model.add(Dense(128, activation=activation_function, input_shape=(784,)))
            model.add(Dropout(0.3))
            model.add(BatchNormalization())

            # Second Hidden Layer
            model.add(Dense(24, activation=activation_function))
            model.add(Dropout(0.3))
            model.add(BatchNormalization())

            # Output Layer (10 Classes)
            model.add(Dense(10, activation='softmax'))

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

Part 1: Trying different batch sizes

Increasing Batch Size resulted in better precision and f1 score:

Batch size effect on Precision and F1-Score

Batch Size	F1-Score	Precision
32	0.86	0.86
64	0.87	0.87
256	0.88	0.88

A) Batch Size = 32

```
In [93]: %%time
model = define_MLP_model()
model.fit(X_train_flattened,y_cat_train,epochs=30, batch_size=32 ,
          validation_data=(X_test_flattened,y_cat_test))
```

```
Epoch 1/30
1875/1875 [==============================] - 10s 5ms/step - loss: 0.8230 - accuracy: 0.7222 -
val_loss: 0.4946 - val_accuracy: 0.8298
Epoch 2/30
1875/1875 [==============================] - 8s 4ms/step - loss: 0.6501 - accuracy: 0.7767 -
val_loss: 0.4584 - val_accuracy: 0.8379
Epoch 3/30
1875/1875 [==============================] - 8s 4ms/step - loss: 0.6150 - accuracy: 0.7879 -
val_loss: 0.4629 - val_accuracy: 0.8356
Epoch 4/30

val_loss: 0.3993 - val_accuracy: 0.8571
CPU times: user 4min 28s, sys: 39.1 s, total: 5min 7s
Wall time: 4min 18s
```

Figure 2 Validation Accuracy and Run Time

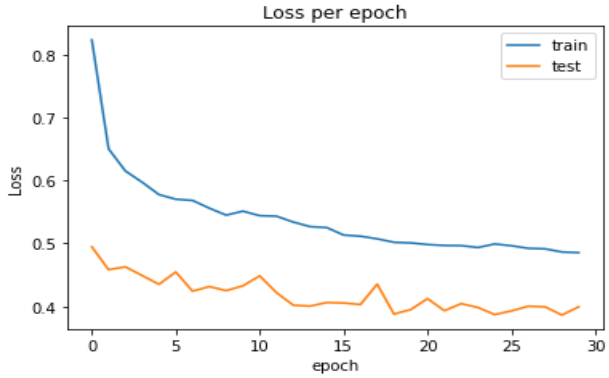


Figure3 Loss per epoch

	precision	recall	f1-score	support
T-shirt/top	0.83	0.82	0.82	1000
Trouser	0.99	0.96	0.97	1000
Pullover	0.81	0.67	0.74	1000
Dress	0.86	0.86	0.86	1000
Coat	0.65	0.88	0.75	1000
Sandal	0.97	0.93	0.95	1000
Shirt	0.68	0.58	0.62	1000
Sneaker	0.93	0.94	0.93	1000
Bag	0.96	0.97	0.96	1000
Ankle boot	0.93	0.96	0.95	1000
accuracy			0.86	10000
macro avg	0.86	0.86	0.86	10000
weighted avg	0.86	0.86	0.86	10000

Figure 4 Classification Report

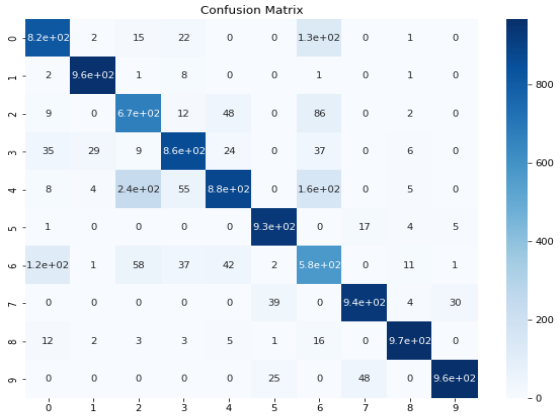


Figure 5 Confusion Matrix

B) Batch Size = 64

```
In [96]: %%time
model = define_MLP_model()
model.fit(X_train_flattened,y_cat_train,epochs=30, batch_size=64 ,
          validation_data=(X_test_flattened,y_cat_test))
```

```
Epoch 1/30
938/938 [==============================] - 7s 6ms/step - loss: 0.8080 - accuracy: 0.7339 - va
l_loss: 0.5154 - val_accuracy: 0.8157
Epoch 2/30
938/938 [==============================] - 6s 6ms/step - loss: 0.5940 - accuracy: 0.7974 - va
l_loss: 0.4373 - val_accuracy: 0.8444
Epoch 3/30
938/938 [==============================] - 5s 6ms/step - loss: 0.5560 - accuracy: 0.8099 - va
l_loss: 0.4377 - val_accuracy: 0.8430
Epoch 4/30
938/938 [==============================] - 6s 6ms/step - loss: 0.5393 - accuracy: 0.8158 - va
l_loss: 0.4405 - val_accuracy: 0.8421

l_loss: 0.3612 - val_accuracy: 0.8691
CPU times: user 3min 12s, sys: 28.3 s, total: 3min 40s
Wall time: 3min 23s
```

Figure 6 Validation Accuracy and Run Time

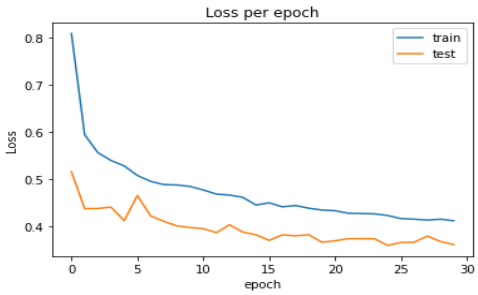


Figure 7 Loss per epoch

	precision	recall	f1-score	support
T-shirt/top	0.78	0.88	0.83	1000
Trouser	0.99	0.96	0.98	1000
Pullover	0.82	0.72	0.77	1000
Dress	0.85	0.90	0.88	1000
Coat	0.70	0.87	0.78	1000
Sandal	0.98	0.94	0.96	1000
Shirt	0.75	0.53	0.62	1000
Sneaker	0.93	0.95	0.94	1000
Bag	0.96	0.97	0.97	1000
Ankle boot	0.94	0.96	0.95	1000
accuracy			0.87	10000
macro avg	0.87	0.87	0.87	10000
weighted avg	0.87	0.87	0.87	10000

Figure 8 Classification Report

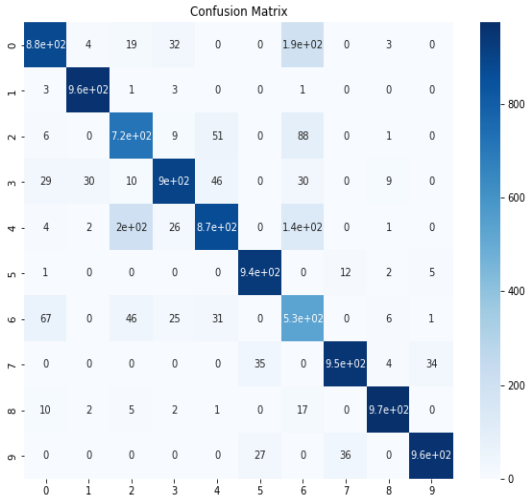


Figure 9 Confusion Matrix

C) Batch Size = 286

```
In [100]: %%time
model = define_MLP_model()
model.fit(X_train_flattened,y_cat_train,epochs=30, batch_size=286
          ,validation_data=(X_test_flattened,y_cat_test))
```

```
Epoch 1/30
210/210 [==============================] - 2s 7ms/step - loss: 0.9126 - accuracy: 0.7116 - va
l_loss: 0.5834 - val_accuracy: 0.8221
Epoch 2/30
210/210 [==============================] - 1s 6ms/step - loss: 0.5869 - accuracy: 0.8094 - va
l_loss: 0.4494 - val_accuracy: 0.8455
Epoch 3/30
210/210 [==============================] - 1s 6ms/step - loss: 0.5184 - accuracy: 0.8280 - va
l_loss: 0.4155 - val_accuracy: 0.8532
Epoch 4/30
210/210 [==============================] - 1s 6ms/step - loss: 0.4858 - accuracy: 0.8370 - va
l_loss: 0.4113 - val_accuracy: 0.8539
                        l_loss: 0.3489 - val_accuracy: 0.8781
                        CPU times: user 47.3 s, sys: 6.75 s, total: 54.1 s
                        Wall time: 42.1 s
```

Figure 10 Validation Accuracy and Run Time

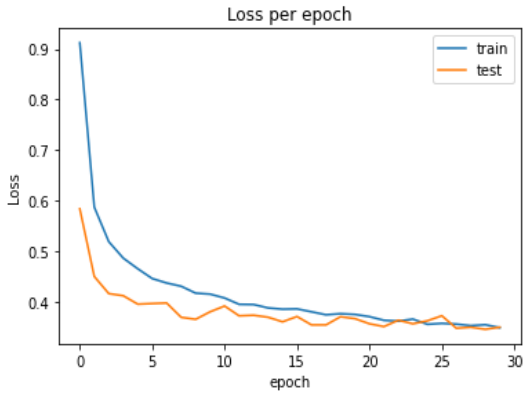


Figure 11 Classification Report

	precision	recall	f1-score	support
T-shirt/top	0.78	0.89	0.83	1000
Trouser	0.98	0.97	0.98	1000
Pullover	0.83	0.77	0.80	1000
Dress	0.87	0.88	0.88	1000
Coat	0.75	0.84	0.79	1000
Sandal	0.98	0.96	0.97	1000
Shirt	0.75	0.58	0.65	1000
Sneaker	0.94	0.95	0.95	1000
Bag	0.95	0.98	0.96	1000
Ankle boot	0.95	0.96	0.96	1000
accuracy			0.88	10000
macro avg	0.88	0.88	0.88	10000
weighted avg	0.88	0.88	0.88	10000

Figure 12 Classification Report

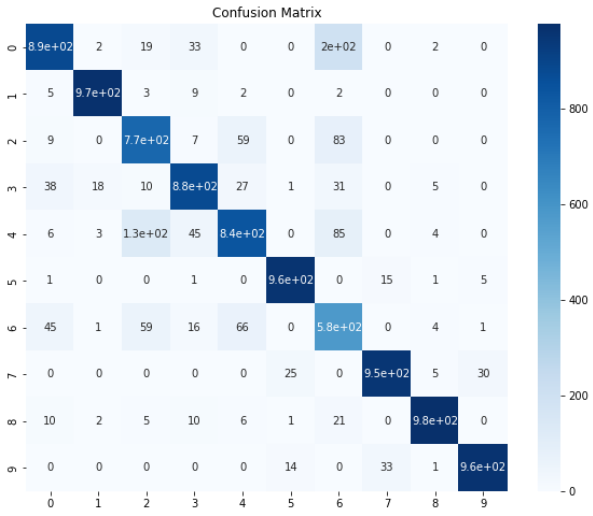


Figure 13 Confusion Matrix

Part 2: Trying different Loss Function and Activation Function

Since this problem is multi-class classification, the loss function should be Cross Entropy not MSE. ReLU and Sigmoid Precision and F1-Score were better than Tanh.

Different Activation Functions on Precision and F1-Score

Activation Function	F1-Score	Precision
Tanh	0.87	0.87
ReLU	0.88	0.88
Sigmoid	0.88	0.88

Loss Functions on Precision and F1-Score

Loss Function	F1-Score	Precision
Cross Entropy	0.88	0.88
Mean Squared Error (MSE)	0.88	0.88

A) Tanh

```
In [106]: %%time
model = define_MLP_model(activation_function = "tanh")
model.fit(X_train_flattened,y_cat_train,epochs=30,
          batch_size=286 ,validation_data=(X_test_flattened,y_cat_test))
```

```
Epoch 1/30
210/210 [==============================] - 2s 7ms/step - loss: 0.7997 - accuracy: 0.7430 - va
l_loss: 0.5130 - val_accuracy: 0.8261
Epoch 2/30
210/210 [==============================] - 1s 6ms/step - loss: 0.5601 - accuracy: 0.8117 - va
l_loss: 0.4530 - val_accuracy: 0.8370
Epoch 3/30
210/210 [==============================] - 1s 6ms/step - loss: 0.5125 - accuracy: 0.8261 - va
l_loss: 0.4532 - val_accuracy: 0.8384
```

```
l_loss: 0.3552 - val_accuracy: 0.8732
CPU times: user 46.3 s, sys: 7.02 s, total: 53.3 s
Wall time: 40 s
```

Figure 14 Validation Accuracy and Run Time

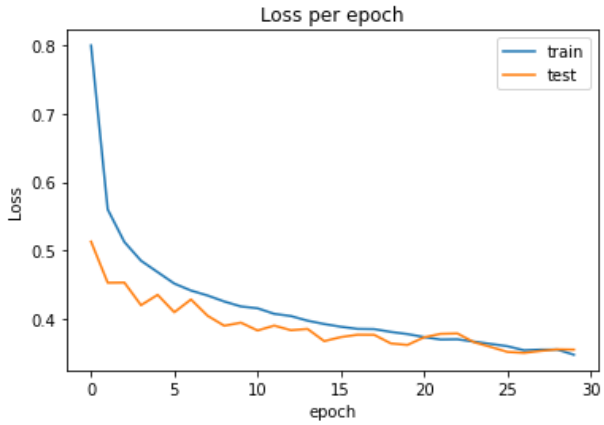


Figure 15 Loss per epoch

	precision	recall	f1-score	support
T-shirt/top	0.83	0.83	0.83	1000
Trouser	0.99	0.96	0.98	1000
Pullover	0.83	0.72	0.77	1000
Dress	0.87	0.88	0.87	1000
Coat	0.73	0.85	0.78	1000
Sandal	0.97	0.94	0.96	1000
Shirt	0.69	0.67	0.68	1000
Sneaker	0.94	0.95	0.94	1000
Bag	0.98	0.98	0.98	1000
Ankle boot	0.95	0.96	0.96	1000
accuracy			0.87	10000
macro avg	0.87	0.87	0.87	10000
weighted avg	0.87	0.87	0.87	10000

Figure 16 Classification Report

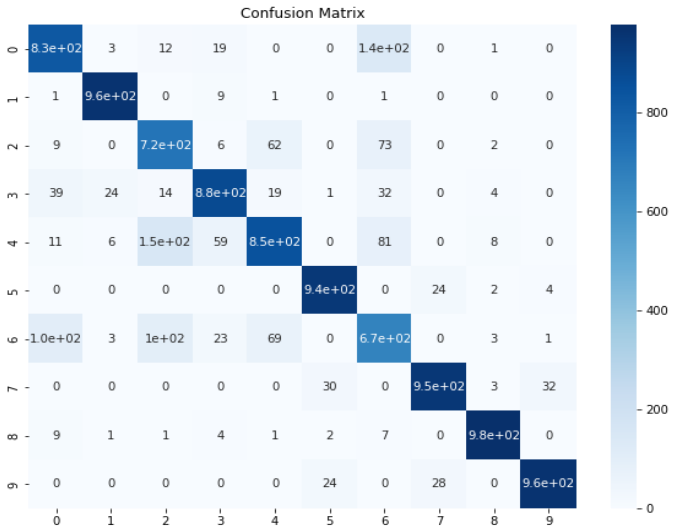


Figure 17 Confusion Matrix

B) ReLU

```
In [109]: %%time
model = define_MLP_model(activation_function = "relu")
model.fit(X_train_flattened,y_cat_train,epochs=30, batch_size=286 ,
          validation_data=(X_test_flattened,y_cat_test))
```

```
Epoch 1/30
210/210 [==============================] - 2s 7ms/step - loss: 0.9321 - accuracy: 0.7141 - va
l_loss: 0.5692 - val_accuracy: 0.8263
Epoch 2/30
210/210 [==============================] - 1s 6ms/step - loss: 0.5924 - accuracy: 0.8087 - va
l_loss: 0.4541 - val_accuracy: 0.8413
Epoch 3/30
210/210 [==============================] - 1s 6ms/step - loss: 0.5228 - accuracy: 0.8248 - va
l_loss: 0.4299 - val_accuracy: 0.8462

l_loss: 0.3418 - val_accuracy: 0.8794
CPU times: user 47.6 s, sys: 6.72 s, total: 54.3 s
Wall time: 41.1 s
```

Figure 18 Validation Accuracy and Run Time

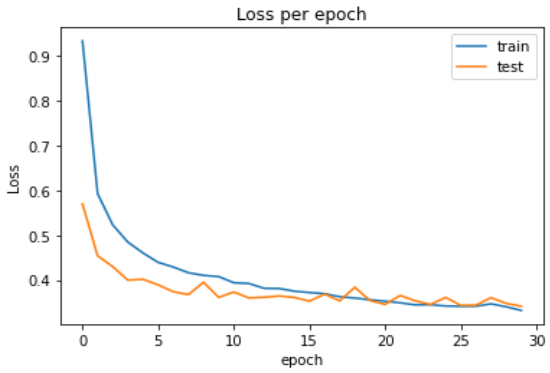


Figure 19 Loss per epoch

	precision	recall	f1-score	support
T-shirt/top	0.79	0.89	0.84	1000
Trouser	0.98	0.97	0.98	1000
Pullover	0.82	0.74	0.78	1000
Dress	0.87	0.90	0.89	1000
Coat	0.72	0.88	0.79	1000
Sandal	0.98	0.96	0.97	1000
Shirt	0.77	0.56	0.65	1000
Sneaker	0.94	0.95	0.95	1000
Bag	0.97	0.97	0.97	1000
Ankle boot	0.95	0.97	0.96	1000
accuracy			0.88	10000
macro avg	0.88	0.88	0.88	10000
weighted avg	0.88	0.88	0.88	10000

Figure20 Classification Report

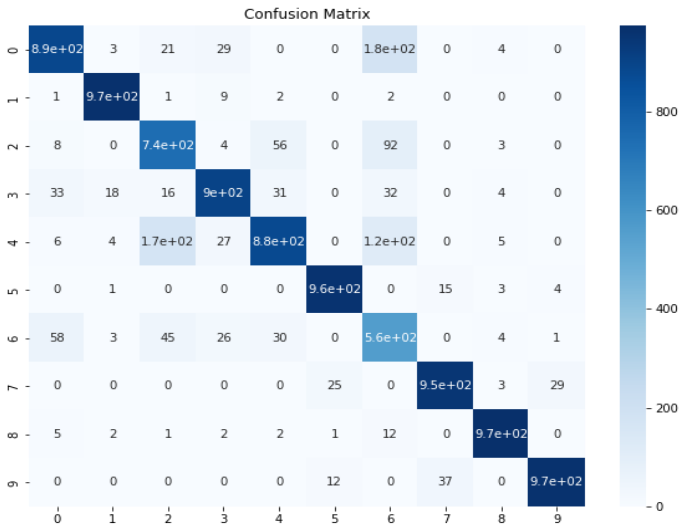


Figure 21 Confusion Matrix

C) Sigmoid

```
In [112]: %%time
          model = define_MLP_model(activation_function = "sigmoid")
          model.fit(X_train_flattened,y_cat_train,epochs=30,
                    batch_size=286 ,validation_data=(X_test_flattened,y_cat_test))
```

```
Epoch 1/30
210/210 [==============================] - 2s 7ms/step - loss: 1.1347 - accuracy: 0.6214 - va
l_loss: 0.7802 - val_accuracy: 0.7913
Epoch 2/30
210/210 [==============================] - 1s 6ms/step - loss: 0.6829 - accuracy: 0.7728 - va
l_loss: 0.5084 - val_accuracy: 0.8263
Epoch 3/30
210/210 [==============================] - 1s 6ms/step - loss: 0.5747 - accuracy: 0.8063 - va
l_loss: 0.4460 - val_accuracy: 0.8389
Epoch 4/30
210/210 [==============================] - 1s 6ms/step - loss: 0.5279 - accuracy: 0.8191 - va
l_loss: 0.4257 - val_accuracy: 0.8453

                l_loss: 0.3379 - val_accuracy: 0.8819
                CPU times: user 47.3 s, sys: 6.64 s, total: 53.9 s
                Wall time: 40.8 s
```

Figure 22 Validation Accuracy and Run Time

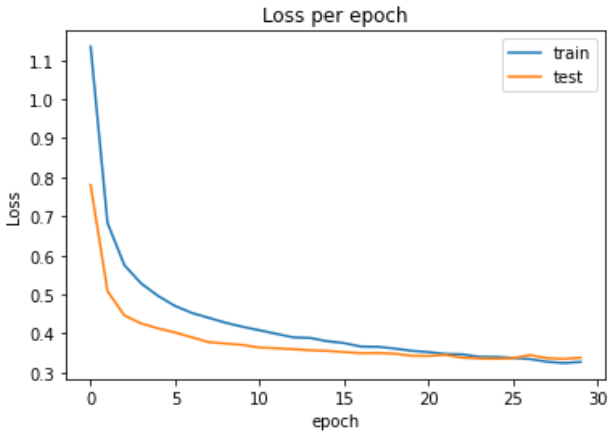


Figure 23 Loss per epoch

	precision	recall	f1-score	support
T-shirt/top	0.84	0.84	0.84	1000
Trouser	0.99	0.97	0.98	1000
Pullover	0.80	0.79	0.79	1000
Dress	0.87	0.90	0.89	1000
Coat	0.80	0.80	0.80	1000
Sandal	0.95	0.97	0.96	1000
Shirt	0.70	0.69	0.69	1000
Sneaker	0.95	0.94	0.94	1000
Bag	0.97	0.97	0.97	1000
Ankle boot	0.96	0.95	0.96	1000
accuracy			0.88	10000
macro avg	0.88	0.88	0.88	10000
weighted avg	0.88	0.88	0.88	10000

Figure 24 Classification Report

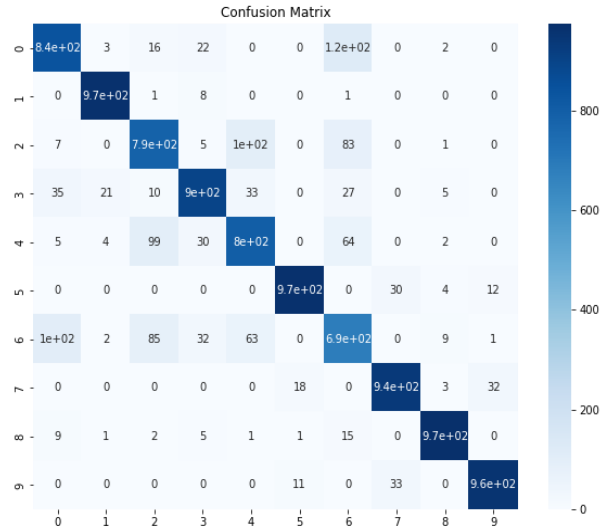


Figure 25 Confusion Matrix

D) Cross Entropy

```
In [115]: %%time
model = define_MLP_model(activation_function = "relu",loss_func='categorical_crossentropy')
model.fit(X_train_flattened,y_cat_train,epochs=30, batch_size=286
          ,validation_data=(X_test_flattened,y_cat_test))
```

```
Epoch 1/30
210/210 [==============================] - 2s 7ms/step - loss: 0.9421 - accuracy: 0.7113 - va
l_loss: 0.6020 - val_accuracy: 0.8110
Epoch 2/30
210/210 [==============================] - 1s 6ms/step - loss: 0.5961 - accuracy: 0.8054 - va
l_loss: 0.4568 - val_accuracy: 0.8392
Epoch 3/30
210/210 [==============================] - 1s 6ms/step - loss: 0.5197 - accuracy: 0.8286 - va
l_loss: 0.4208 - val_accuracy: 0.8512
            l_loss: 0.3397 - val_accuracy: 0.8813
            CPU times: user 47.3 s, sys: 6.66 s, total: 54 s
            Wall time: 42.1 s
```

Figure 26 Validation Accuracy and Run Time

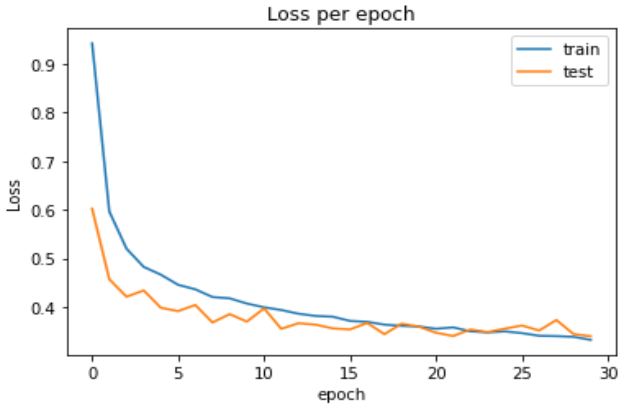


Figure 27 Loss per epoch

	precision	recall	f1-score	support
T-shirt/top	0.79	0.88	0.83	1000
Trouser	0.99	0.96	0.98	1000
Pullover	0.80	0.79	0.80	1000
Dress	0.87	0.91	0.89	1000
Coat	0.80	0.81	0.80	1000
Sandal	0.97	0.96	0.96	1000
Shirt	0.71	0.61	0.66	1000
Sneaker	0.94	0.97	0.95	1000
Bag	0.96	0.97	0.97	1000
Ankle boot	0.97	0.95	0.96	1000
accuracy			0.88	10000
macro avg	0.88	0.88	0.88	10000
weighted avg	0.88	0.88	0.88	10000

Figure 28 Classification Report

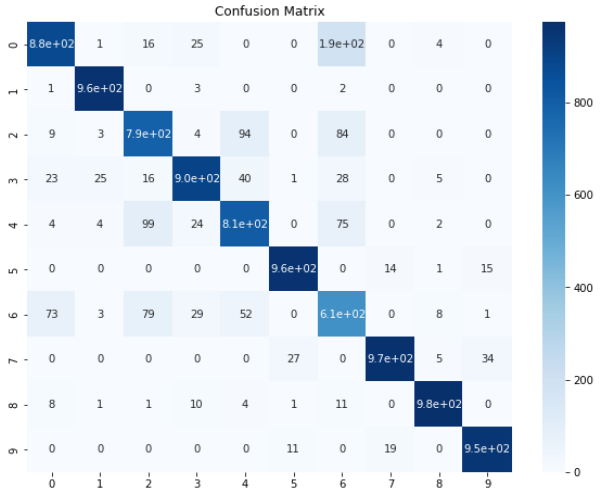


Figure 29 Confusion Matrix

E) Mean Squared Error(MSE)

```
In [118]: %%time
model = define_MLP_model(activation_function = "sigmoid",loss_func='mean_squared_error')
model.fit(X_train_flattened,y_cat_train,epochs=30,
          batch_size=286 ,validation_data=(X_test_flattened,y_cat_test))
```

```
Epoch 1/30
210/210 [==============================] - 2s 7ms/step - loss: 0.0483 - accuracy: 0.6471 - va
l_loss: 0.0343 - val_accuracy: 0.8059
Epoch 2/30
210/210 [==============================] - 1s 6ms/step - loss: 0.0310 - accuracy: 0.7905 - va
l_loss: 0.0253 - val_accuracy: 0.8204
Epoch 3/30
210/210 [==============================] - 1s 6ms/step - loss: 0.0272 - accuracy: 0.8158 - va
l_loss: 0.0232 - val_accuracy: 0.8371

l_loss: 0.0177 - val_accuracy: 0.8822
CPU times: user 48.5 s, sys: 6.11 s, total: 54.6 s
Wall time: 41.1 s
```

Figure 30 Validation Accuracy and Run Time

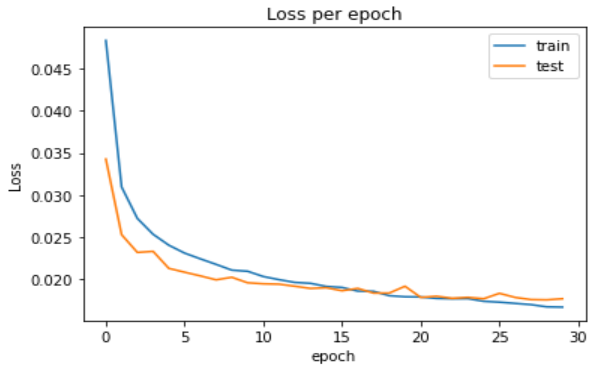


Figure 31 Loss per epoch

	precision	recall	f1-score	support
T-shirt/top	0.86	0.81	0.83	1000
Trouser	0.99	0.97	0.98	1000
Pullover	0.78	0.84	0.81	1000
Dress	0.87	0.89	0.88	1000
Coat	0.81	0.78	0.79	1000
Sandal	0.98	0.95	0.96	1000
Shirt	0.69	0.69	0.69	1000
Sneaker	0.92	0.97	0.94	1000
Bag	0.97	0.97	0.97	1000
Ankle boot	0.97	0.95	0.96	1000
accuracy			0.88	10000
macro avg	0.88	0.88	0.88	10000
weighted avg	0.88	0.88	0.88	10000

Figure 32 Classification Report

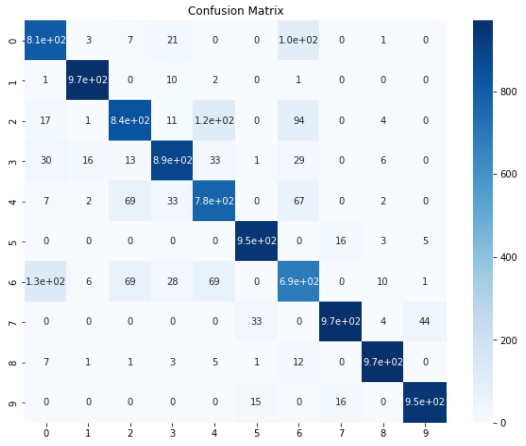


Figure 33 Confusion Matrix

Part 3: Convolutional Neural Network

The accuracy of the model became 3% better with 2d Convolutions.

F1-Score with/without Convolutional Layer.

F1-Score with Convolutional Layer	F1-Score without Convolutional Layer
0.91	0.88

Reshaping the Data

Right now our data is 60,000 images stored in 28 by 28 pixel array formation.

This is correct for a CNN, but we need to add one more dimension to show we're dealing with 1 RGB channel (since technically the images are in black and white, only showing values from 0-255 on a single channel), a color image would have 3 dimensions.

In [121]:
```
X_train = X_train.reshape(60000, 28, 28, 1)
X_test = X_test.reshape(10000,28,28,1)
X_train.shape
```

Out[121]:
```
(60000, 28, 28, 1)
```

Defining the CNN model

In [133]:
```
model = Sequential()

# Convolutional Layer
model.add(Conv2D(filters=32, kernel_size=(4,4),input_shape=(28, 28, 1), activation='relu',))

# Flatten images from 28 by 28 to 764
model.add(Flatten())

#Hidden Layers
model.add(Dense(128, activation='relu'))
model.add(Dense(24, activation='relu'))

#Output Layer
model.add(Dense(10, activation='softmax'))

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

Figure34 Model Summary

```
Model: "sequential_18"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)           (None, 25, 25, 32)        544

 flatten_2 (Flatten)         (None, 20000)             0

 dense_54 (Dense)            (None, 128)               2560128

 dense_55 (Dense)            (None, 24)                3096

 dense_56 (Dense)            (None, 10)                250

=================================================================
Total params: 2,564,018
Trainable params: 2,564,018
Non-trainable params: 0
_________________________________________________________________
```

Training the model

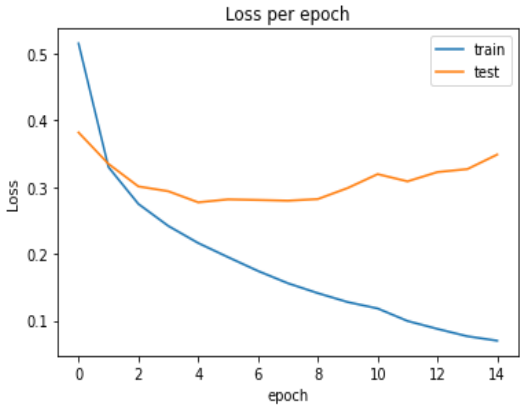


Figure 35 Loss per epoch

	precision	recall	f1-score	support
T-shirt/top	0.84	0.88	0.86	1000
Trouser	0.99	0.98	0.98	1000
Pullover	0.82	0.90	0.86	1000
Dress	0.92	0.91	0.91	1000
Coat	0.83	0.87	0.85	1000
Sandal	0.98	0.97	0.98	1000
Shirt	0.83	0.66	0.73	1000
Sneaker	0.96	0.96	0.96	1000
Bag	0.97	0.98	0.98	1000
Ankle boot	0.96	0.97	0.96	1000
accuracy			0.91	10000
macro avg	0.91	0.91	0.91	10000
weighted avg	0.91	0.91	0.91	10000

Figure36 Classification Report

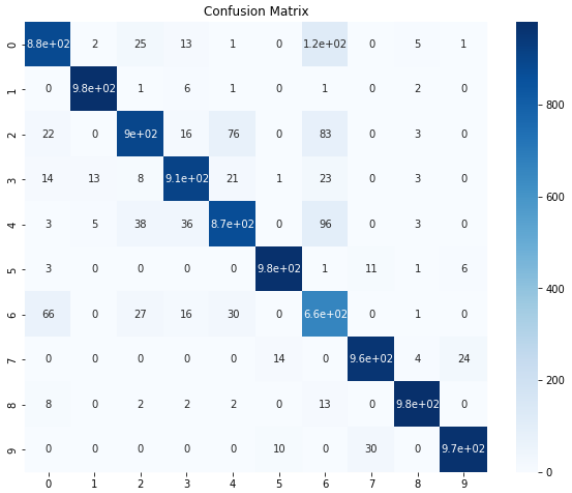


Figure 37 Confusion Matrix

Part 4: Batch Normalization & Pooling Layer

The accuracy of the model became 3% better with Batch Normalization and Pooling Layer.

F1-Score with/without Batch Normalization and Pooling Layer.

Current F1-Score	Previous F1-Score
0.91	0.88

In [144]:
```
model = Sequential()

# Convolutional Layer
model.add(Conv2D(filters=32, kernel_size=(4,4),input_shape=(28, 28, 1), activation='relu',))

# Pooling Layer
model.add(MaxPool2D(pool_size=(2, 2)))

# Flatten images from 28 by 28 to 764
model.add(Flatten())

#Hidden Layers
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())

model.add(Dense(24, activation='relu'))
model.add(BatchNormalization())

#Output Layer
model.add(Dense(10, activation='softmax'))

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

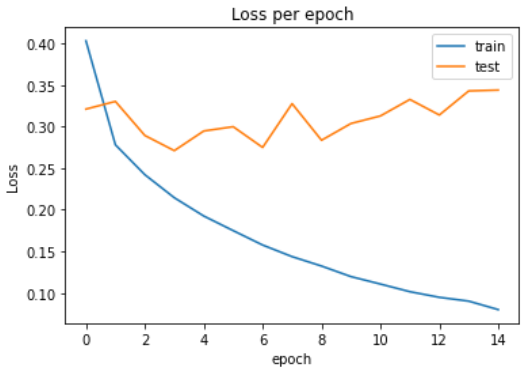


Figure 38 Loss per epoch

	precision	recall	f1-score	support
T-shirt/top	0.87	0.83	0.85	1000
Trouser	0.99	0.98	0.99	1000
Pullover	0.82	0.88	0.85	1000
Dress	0.93	0.89	0.91	1000
Coat	0.85	0.85	0.85	1000
Sandal	0.98	0.96	0.97	1000
Shirt	0.74	0.74	0.74	1000
Sneaker	0.93	0.98	0.95	1000
Bag	0.98	0.98	0.98	1000
Ankle boot	0.97	0.95	0.96	1000
accuracy			0.91	10000
macro avg	0.91	0.91	0.91	10000
weighted avg	0.91	0.91	0.91	10000

Figure 39 Classification Report

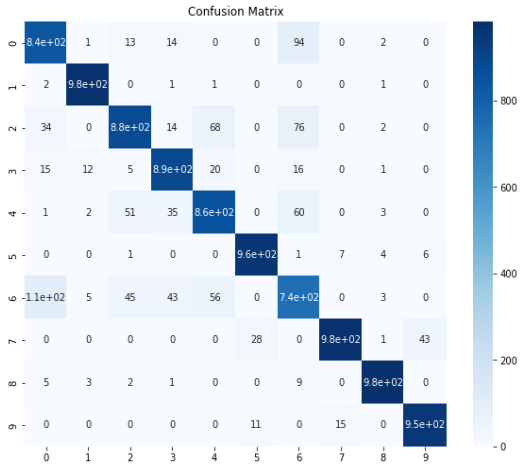


Figure 40 Confusion Matrix

Part 5: Adding Dropout

What is Dropout Layer?

Dropout Layer is one of the most popular regularization techniques to reduce overfitting in the deep learning models. Overfitting in the model occurs when it shows more accuracy on the training data but less accuracy on the test data or unseen data. In the dropout technique, some of the neurons in hidden or visible layers are dropped randomly.

The accuracy of the model became 5% better with Dropout layer. This experiment shows that the dropout technique regularizes the neural network model to produce a robust model which does not overfit.

F1-Score with/without Dropout.

F1-Score with Dropout	F1-Score without Dropout
0.93	0.88

In [156]:
```
model = Sequential()

# Convolutional Layer
model.add(Conv2D(filters=32, kernel_size=(4,4),input_shape=(28, 28, 1), activation='relu',))

# Pooling Layer
model.add(MaxPool2D(pool_size=(2, 2)))

# Flatten images from 28 by 28 to 764
model.add(Flatten())

#Hidden Layers
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.3))
model.add(BatchNormalization())

model.add(Dense(24, activation='relu'))
model.add(Dropout(0.3))
model.add(BatchNormalization())

#Output Layer
model.add(Dense(10, activation='softmax'))

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

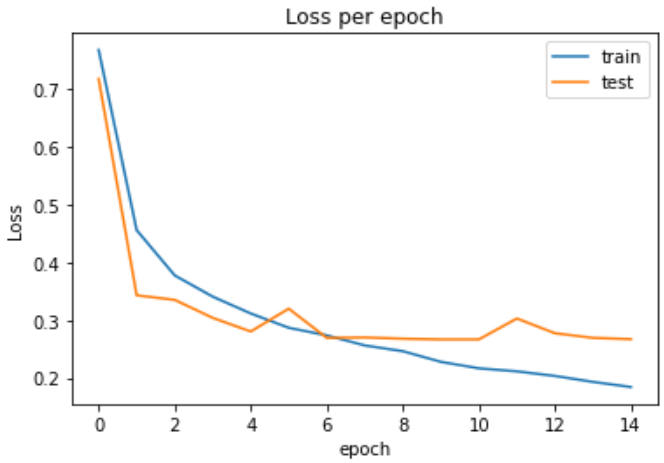


Figure 41 Loss per epoch

	precision	recall	f1-score	support
T-shirt/top	0.84	0.89	0.86	1000
Trouser	0.99	0.98	0.99	1000
Pullover	0.84	0.87	0.86	1000
Dress	0.92	0.92	0.92	1000
Coat	0.83	0.89	0.86	1000
Sandal	0.97	0.99	0.98	1000
Shirt	0.81	0.67	0.73	1000
Sneaker	0.96	0.97	0.96	1000
Bag	0.98	0.99	0.98	1000
Ankle boot	0.97	0.96	0.97	1000
accuracy			0.91	10000
macro avg	0.93	0.93	0.93	10000
weighted avg	0.93	0.93	0.93	10000

Figure42 Classification Report

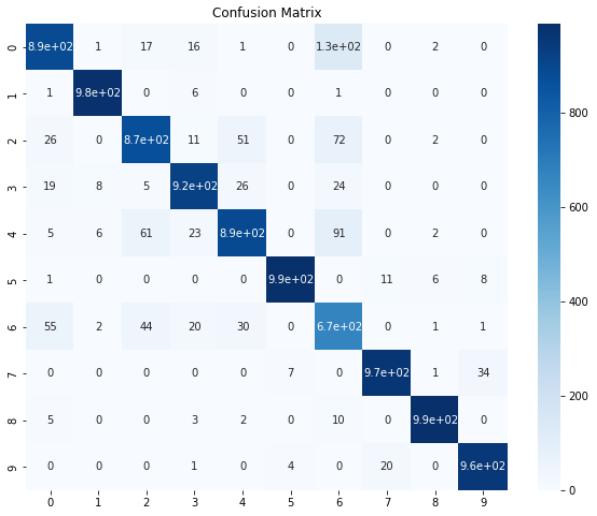


Figure 43 Confusion Matrix

2. Question 2 - Transfer Learning

Part 1: VGG-16

VGG-16 is a convolutional neural network that is 16 layers deep. The pre-trained version of the network was trained on more than a million images from the **ImageNet** database. The pretrained network can classify images into **1000** object categories.

VGG-16 Architecture

- **Input:** The VGGNet takes in an image input size of 224×224.
- **Convolutional Layers**: VGG's convolutional layers leverage a minimal receptive field, for example 3×3, the smallest possible size that still captures up and down and left and right. And also, there are also 1×1 convolution filters acting as a linear transformation of the input. This is followed by a ReLU unit. The convolution stride is fixed at 1 pixel to keep the spatial resolution preserved after convolution.
- **Hidden Layers:** All the hidden layers in the VGG network use ReLU as activation function. There are 2 Fully Connected Layers with 4096 neurons as hidden layers.
- **Output:** The output layer has 1000 neurons for each class. And also softmax is used as activation function in this layer.

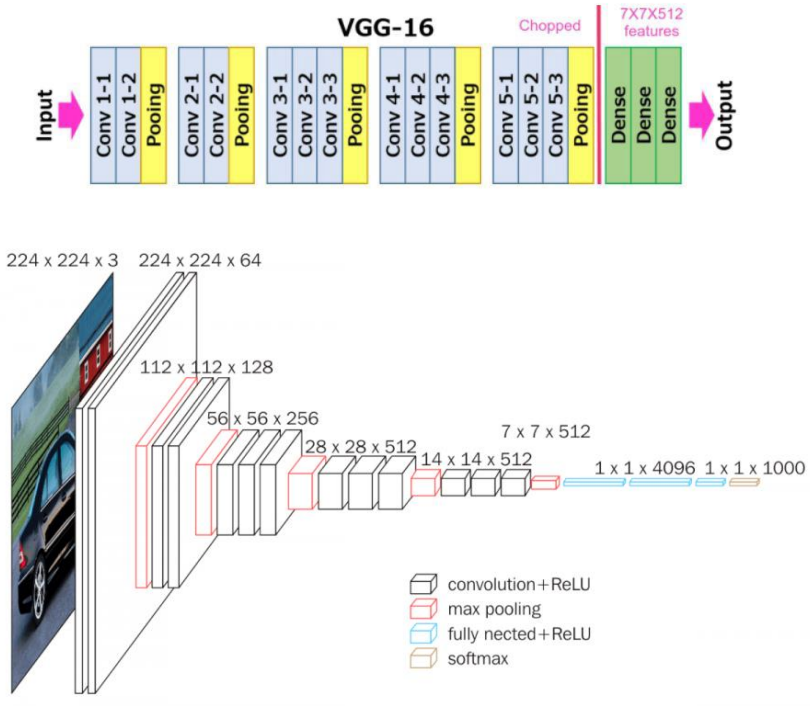


Figure 44 VGG-16 Architecture

VGG-16 Advantages and Disadvantages

The **advantages** are:

- VGG-16 significantly outperforms the previous generation of models in the ILSVRC-2012 and ILSVRC-2013 competitions.
- VGG-16 architecture achieves the best result (7.0% test error), outperforming a single GoogLeNet by 0.9%.
- VGG-16 model achieved 92.7% top-5 test accuracy in ImageNet.

The **disadvantages** are:

- It is painfully slow to train.
- The network architecture weights themselves are quite large

Loading and pre-processing an image

We can load the image using any library such as OpenCV, PIL, skimage etc. We perform the following steps on an input image:

1. Load the image. This is done using the load_img() function. Keras uses the PIL format for loading images. Thus, the image is in width x height x channels format.
2. Convert the image from PIL format to Numpy format using img_to_array() function.
3. The networks accept a 4-dimensional Tensor as an input of the form (batchsize, height, width, channels). This is done using the expand_dims() function in Numpy.
4. Preprocess the input by subtracting the mean value from each channel of the images in the batch. Mean is an array of three elements obtained by the average of R, G, B pixels of all images obtained from ImageNet. The values for ImageNet are : [103.939, 116.779, 123.68]. This is done using the preprocess_input() function in Keras.
5. Get the classification result, which is a Tensor of dimension (batch size x 1000). This is done by model.predict() function.
6. Convert the result to human-readable labels. This is done by decode_predictions.

```
In [24]: def preprocess_image_for_vgg16(image) :
           # Convert the image pixels to a numpy array
           image = img_to_array(image)

           # Reshape data for the model
           image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))

           # Prepare the image for the VGG model
           image = preprocess_input(image)
           return image
```

Part 2: VGG-16 Implementation

First, we download weights & biases of VGG-16 and then load it to our model.

In [ ]:
```
model = Sequential()
model.add(Convolution2D(input_shape=(224,224,3),filters=64,kernel_size=(3,3),padding="same",
                        activation="relu"))
model.add(Convolution2D(filters=64,kernel_size=(3,3),padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))
model.add(Convolution2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Convolution2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))
model.add(Convolution2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Convolution2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Convolution2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))
model.add(Convolution2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Convolution2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Convolution2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))
model.add(Convolution2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Convolution2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Convolution2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2),name='vgg16'))
model.add(Flatten(name='flatten'))
model.add(Dense(4096, activation='relu', name='fc1'))
model.add(Dense(4096, activation='relu', name='fc2'))
model.add(Dense(1000, activation='sigmoid', name='output'))
```

In [ ]:
```
model.load_weights('VGG16_Weights_Biases.h5')
```

Predicting labels from 3 sample images

The VGG-16 model predicted all of these images correctly.

Image 1: Coffee Mug

Figure45 Coffee Mug

In [ ]:
```
sample_mug = preprocess_image_for_vgg16(sample_mug)
```

In [ ]:
```
# Predict the probability across all output classes
yhat = model.predict(sample_mug)
# Convert the probabilities to class labels
label = decode_predictions(yhat)
# Retrieve the most likely result, e.g. highest probability
label = label[0][0]
# Print the classification
print('%s (%.2f%%)' % (label[1], label[2]*100))
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/imagenet_cl
ass_index.json
40960/35363 [==================================] - 0s 0us/step
49152/35363 [==========================================] - 0s 0us/step
coffee_mug (100.00%)
```

Image 2: Golden Retriever

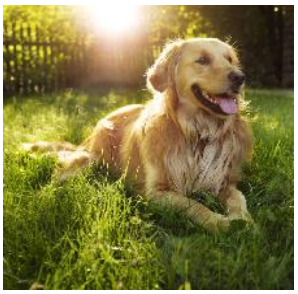


Figure 46 Golden Retriever

```
In [ ]: sample_dog = preprocess_image_for_vgg16(sample_dog)
```

```
In [ ]: # Predict the probability across all output classes
        yhat = model.predict(sample_dog)
        # Convert the probabilities to class labels
        label = decode_predictions(yhat)
        # Retrieve the most likely result, e.g. highest probability
        label = label[0][0]
        # Print the classification
        print('%s (%.2f%%)' % (label[1], label[2]*100))
```

```
golden_retriever (100.00%)
```

Image 3: Rooster

Figure 47 Rooster

```
In [ ]: sample_rooster = preprocess_image_for_vgg16(sample_rooster)
```

```
In [ ]: # Predict the probability across all output classes
        yhat = model.predict(sample_rooster)
        # Convert the probabilities to class labels
        label = decode_predictions(yhat)
        # Retrieve the most likely result, e.g. highest probability
        label = label[0][0]
        # Print the classification
        print('%s (%.2f%%)' % (label[1], label[2]*100))
```

```
cock (100.00%)
```

Part 3

What types of image does VGG-16 classify?

The VGG16 model achieved 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to **1000 classes**. For example the model can predict these classes: (You can find the labels that VGG-16 can predict from here)

- goldfish, Carassius auratus
- hen
- kite
- and 997 more class...

What should we do if the classes we want classify are not in the VGG-16 model?

The answer is **Transfer Learning**.

Transfer learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task. It is a popular approach in deep learning where pre-trained models are used as the starting point on computer vision and natural language processing tasks given the vast compute and time resources required to develop neural network models on these problems and from the huge jumps in skill that they provide on related problems.

Now suppose we want to classify different types of flowers. The VGG-16 model has seen many images of flowers but not specific types. We can modify the model architecture to solve this problem.

We know that the ImageNet dataset contains images of different flowers. We can import a model that has been pre-trained on the ImageNet dataset and use its **pre-trained layers** for **feature extraction**.

Now we can't use the entirety of the pre-trained model's architecture. The Fully-Connected layer generates 1,000 different output labels, whereas our Target Dataset has only 10 classes for prediction. So we'll import a pre-trained model like VGG16, but remove the Fully-Connected layer of the model.

Transfer Learning Approaches

1. Feature Extraction

We use the pre-trained model's architecture to create a new dataset from our input images in this approach. We'll import the Convolutional and Pooling layers but leave out the Fully-Connected layer of the model(Top Layer).

Recall that our example model, VGG16, has been trained on millions of images - including vehicle images. Its convolutional layers and trained weights can detect generic features such as edges, colors, wheels, windshields, etc.

2. Fine-Tuning Extraction (The selected method)

The goal of fine-tuning is to allow a portion of the pre-trained layers to retrain. Steps of Fine-Tuning we will follow in this problem:

- Changing the fully connected and output layer.
- Freezing pre-trained convolutional layers
- Un-freezing the last few pre-trained layers training.

Fine-Tuning VGG-16 For Flowers Classification

After preprocessing the images:

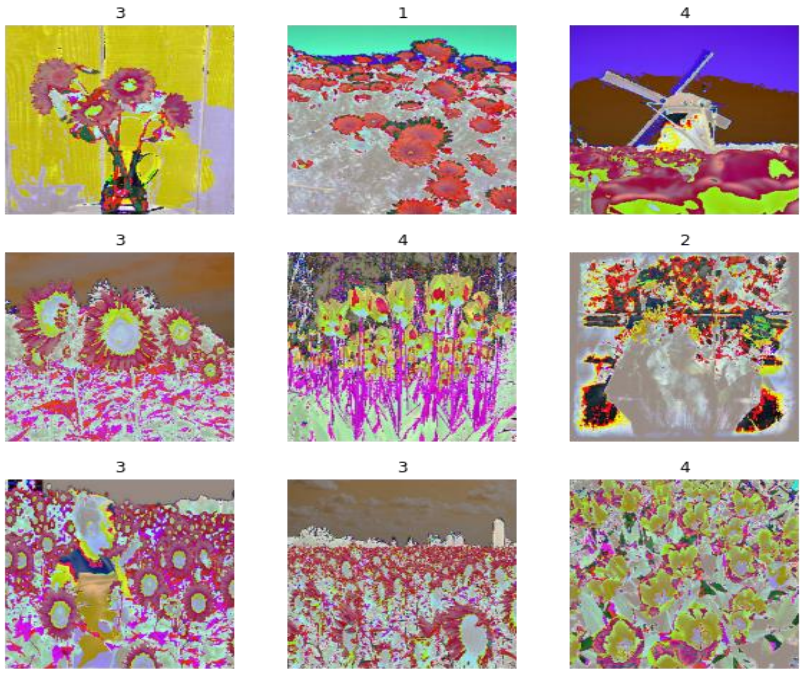


Figure 48 Visualizing the pre-processed images

Load the pre-trained model

Here, we'll load pre-trained weights & biases from VGG-16 model without the top layer which consists of Fully Connected Layers.

```
In [73]: from tensorflow.keras.applications import vgg16
         vgg_conv = vgg16.VGG16(weights='imagenet', include_top = False,
                                input_shape=(image_size, image_size, 3))
```

Freeze the required layers

In Keras, each layer has a parameter called “trainable”. For freezing the weights of a particular layer, we should set this parameter to False, indicating that this layer should not be trained.

In [74]:
```
# Freeze all the layers
for layer in vgg_conv.layers[:]:
  layer.trainable = False

# Check the trainable status of the individual layers
for layer in vgg_conv.layers:
  print(layer, layer.trainable)
```

```
<keras.engine.input_layer.InputLayer object at 0x7fcbb078ea90> False
<keras.layers.convolutional.Conv2D object at 0x7fcc46836c10> False
<keras.layers.convolutional.Conv2D object at 0x7fcbca04fb10> False
<keras.layers.pooling.MaxPooling2D object at 0x7fcbb045e690> False
<keras.layers.convolutional.Conv2D object at 0x7fcbb1ea0850> False
<keras.layers.convolutional.Conv2D object at 0x7fcbb1ea7310> False
<keras.layers.pooling.MaxPooling2D object at 0x7fcbb1ea7050> False
<keras.layers.convolutional.Conv2D object at 0x7fcbb1f835d0> False
<keras.layers.convolutional.Conv2D object at 0x7fcbca024f90> False
<keras.layers.convolutional.Conv2D object at 0x7fcbb1eb0190> False
<keras.layers.pooling.MaxPooling2D object at 0x7fcbb1ead210> False
<keras.layers.convolutional.Conv2D object at 0x7fcbb1eb6090> False
<keras.layers.convolutional.Conv2D object at 0x7fcc46836050> False
<keras.layers.convolutional.Conv2D object at 0x7fcbb1e3f750> False
<keras.layers.pooling.MaxPooling2D object at 0x7fcbb1e94050> False
<keras.layers.convolutional.Conv2D object at 0x7fcbb1e40790> False
<keras.layers.convolutional.Conv2D object at 0x7fcbb1eb4e10> False
<keras.layers.convolutional.Conv2D object at 0x7fcbb1e43110> False
<keras.layers.pooling.MaxPooling2D object at 0x7fcbb1e4ed10> False
```

Create a new model

Now that we have set the trainable parameters of our base network, we would like to add a classifier on top of the convolutional base. We will simply add a fully connected layer followed by a softmax layer with 5 outputs (Since there are 5 classes of flowers).

In [105]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPool2D, Flatten, Dropout,
BatchNormalization

# Create the model
model = Sequential()

# Add the vgg convolutional base model
model.add(vgg_conv)

# Add new layers
model.add(Flatten())
model.add(Dense(1024, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(5, activation='softmax'))

# Show a summary of the model. Check the number of trainable parameters
model.summary()
```

```
Model: "sequential_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 7, 7, 512)         14714688

 flatten_9 (Flatten)         (None, 25088)             0

 dense_18 (Dense)            (None, 1024)              25691136

 dropout_9 (Dropout)         (None, 1024)              0

 dense_19 (Dense)            (None, 5)                 5125

=================================================================
Total params: 40,410,949
Trainable params: 25,696,261
Non-trainable params: 14,714,688
_________________________________________________________________
```

Figure 49 The new model summary

Training the model

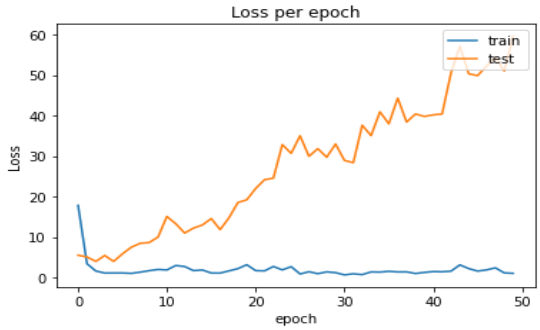


Figure 50 Loss per epoch

	precision	recall	f1-score	support
0.0	0.91	0.88	0.89	129
1.0	0.87	0.95	0.91	176
2.0	0.79	0.87	0.83	120
3.0	0.95	0.82	0.88	152
4.0	0.86	0.87	0.86	157
accuracy			0.88	734
macro avg	0.88	0.87	0.87	734
weighted avg	0.88	0.88	0.88	734

Figure 51 Classification Report

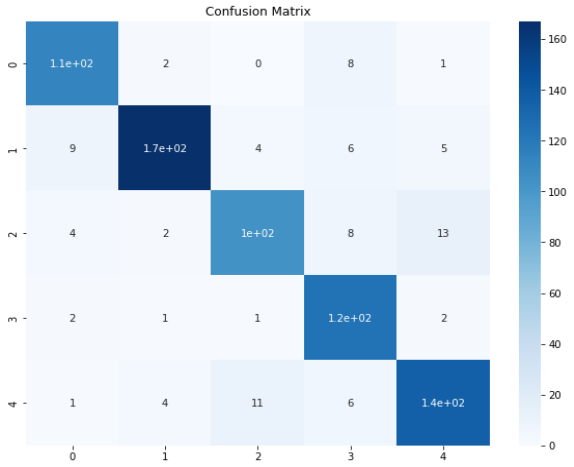


Figure 52 Confusion Matrix

3. Question 3 - Semantic Segmentation

Part 1:

Application of Semantic Segmentation

1) Handwriting Recognition

Semantic segmentation is being used to extract words and lines from handwritten documents.

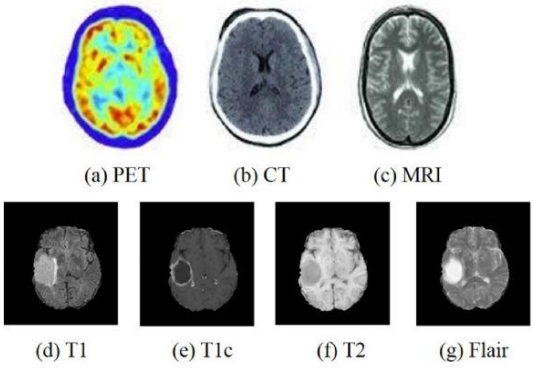
2) Autonomous Driving

Self-driving cars need a complete understanding of their surroundings to a pixel perfect level. So image segmentation is used to identify lanes and other necessary information.

After segmentation

3) Medical Image Segmentation

Semantic Segmentation is used to identify salient elements in medical scans. It is especially useful to identify abnormalities such as tumors.



Segmentation of medical scans

DeepLab

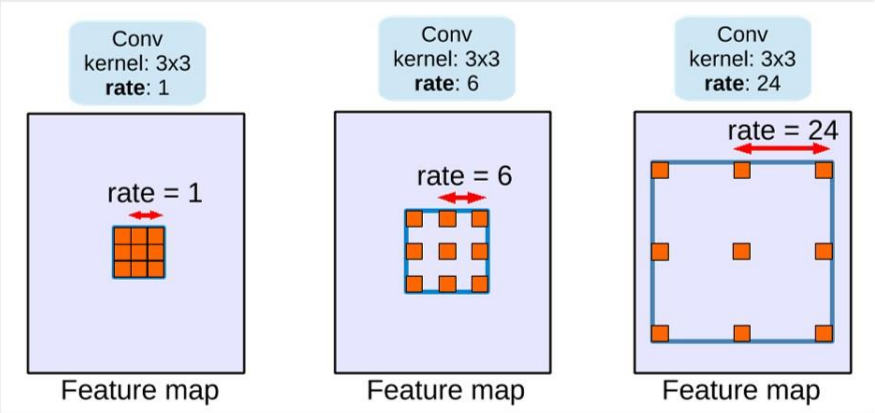
DeepLab have proposed some new techniques to improve the existing results and get better results at lower computational costs. The main suggestion for improvement from DeepLab group from Google are:

1. Atrous convolutions
2. Atrous Spatial Pyramidal Pooling
3. Conditional Random Fields

1) Atrous Convolutions

One of the major problems with Fully Convolutional Network is the excessive downsizing due to Pooling Layers and can result in information loss. This paper suggests Atrous Convolution Layer.

Dilated convolutions (Atrous Convolutions) introduce another parameter to convolutional layers called the dilation rate. This defines a spacing between the values in a kernel. Note that When the rate is equal to 1 it is nothing but the normal convolution.



Example of Atrous Convolution Layer

In Deeplab last pooling layers are replaced to have stride 1 instead of 2 thereby keeping the down sampling rate to only 8x.

2) Atrous Spatial Pyramidal Pooling (ASPP)

Spatial Pyramidal Pooling (SPP) layers do not just apply one pooling operation, it applies a couple of different output sized pooling operations and combines the results before sending them to the next layer.

In the paper, the author suggests ASPP for multi-scale information. ASPP takes the concept of combining information from different scales and applies it to Atrous convolutions.

In this picture, you can see that the input is convolved with 3x3 filters of dilation rates 6, 12, 18 and 24 and the outputs are concatenated together since they are of same size.

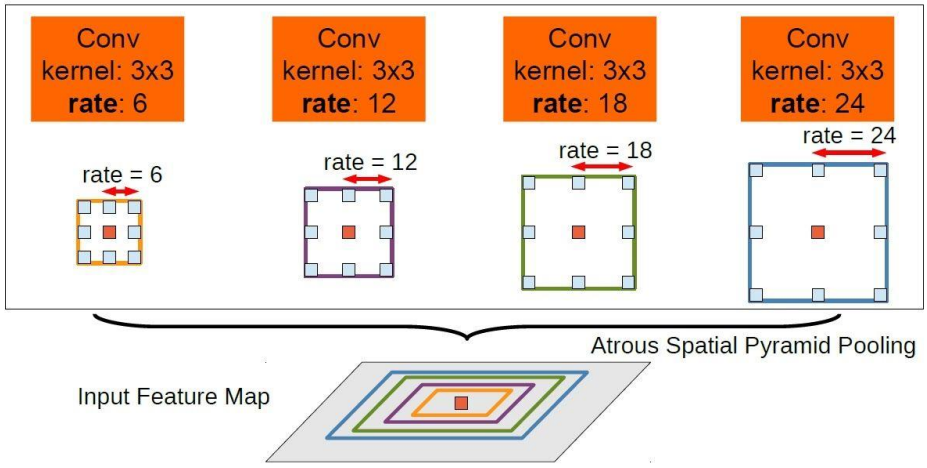


Figure 53 Atrous Spatial Pyramidal Pooling (ASPP)

3) Conditional Random Fields (CRF)

Conditional Random Fields are a type of **discriminative classifier**. Discriminative classifiers try to find boundaries that separate classes.

DeepLab-v3+ Architecture.

This model consists of Encoder and Decoder parts. Deeplab-v3+ suggested to have a decoder instead of plain bilinear up sampling 16x. The encoder output is up sampled 4x using bilinear up sampling and concatenated with the features from encoder which is again up sampled 4x after performing a 3x3 convolution. Deeplab-v3 introduced batch normalization and suggested dilation rate should be multiplied by 1,2 and 4 inside each layer.

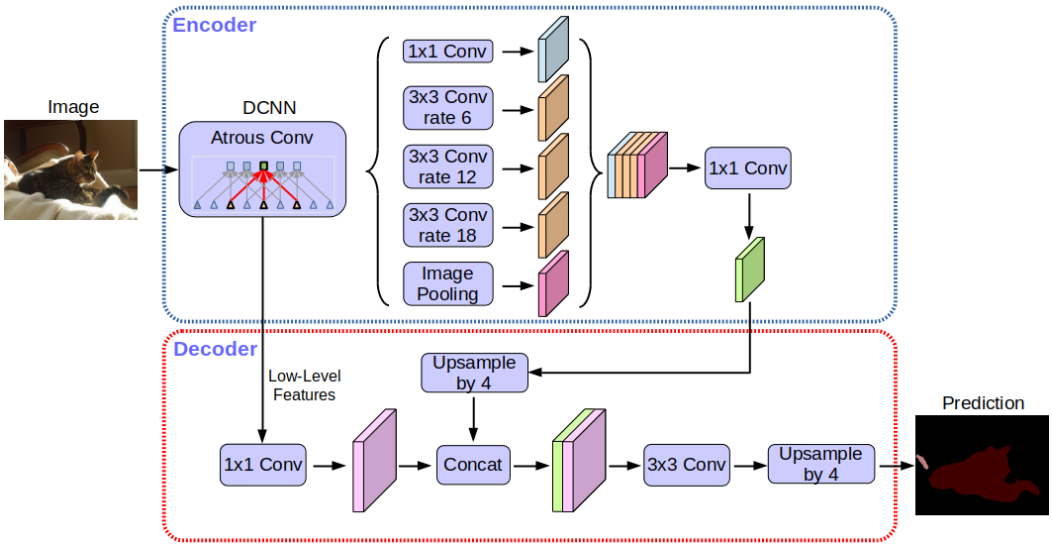


Figure 54 Encoder-Decoder architecture of DeepLab v3+

Part 2:

Pre-processing input

Pre-processing steps:

- Resizing the image to 512x512.
- Add Padding to the image to match training images
- Apply Normalization (Divide by 255).

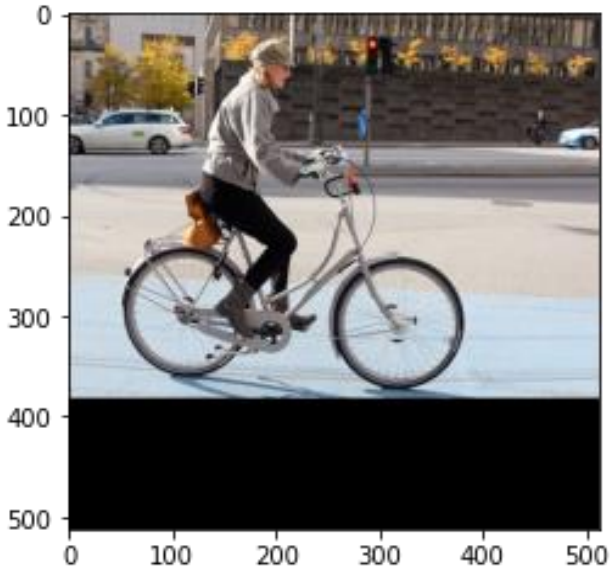


Figure 55 After Pre-processing

Testing the model with a sample picture

Semantic segmentation is the process of classifying each pixel belonging to a particular label. It doesn't differentiate across different instances of the same object. For instance, there are two cars in this picture and the model gives **same label(color)** to all the pixels of both cars. The person and his bicycle are labeled almost perfectly with green and dark purple respectively.

Figure 56 The result

4. Question 4 - Object Detection

Part 1: Object Detection Categories

There are two types of Object Detection:

- **Two Stage Detector:** These networks separate the tasks of determining the location of objects and their classification and they also achieve the highest detection accuracy but are typically slower.

  Famous Networks:

  - Faster R-CNN
  - Mask R-CNN
  - G-RCNN
  - Granulated RCNN (G-RCNN)

- **One Stage Detector:** These networks can predict bounding boxes and class scores at once. These networks are faster but their accuracy is lower compared to two stage detectors.

  Famous Networks:

  - YOLO
  - SSD
  - RetinaNet

Part 2: Overlap Problem in YOLOv1

YOLOv1 had a problem with overlapping objects. YOLOv1 predicted X, Y coordinates and width and height directly by regressing at the end so it couldn't classify overlapping objects correctly.

R-CNN and SSD take the concept of anchor boxes with 3 different scales and 3 different aspect ratios and compute the offsets for these pre-given anchor boxes to predict boxes for objects. This makes it easy for the algorithm to learn just the offsets and select the size of the box than to learn this info on its own.

YOLOv2 adopted this idea of **anchor boxes** but instead of predefining anchor boxes, it looks at bounding boxes of training data and run k-means clustering on those boxes and came up with basically a set of dimension clusters which are more grounded in reality.

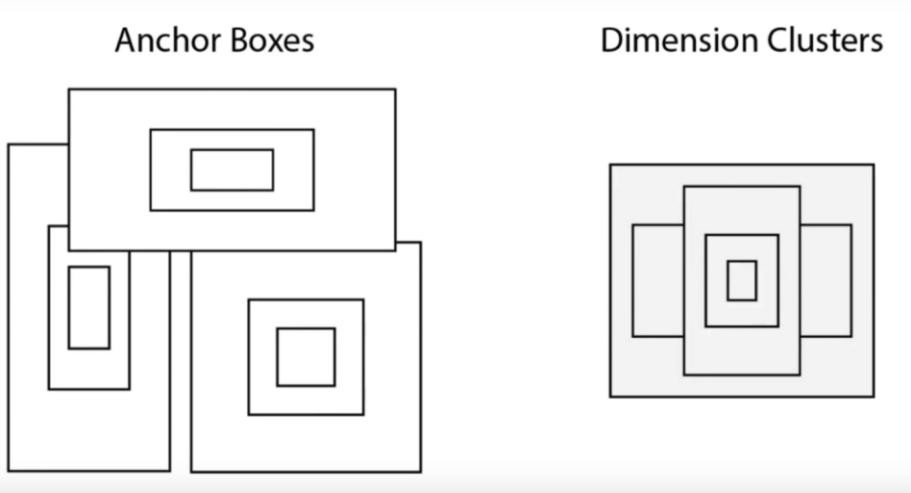


Figure 57 Anchor Boxes

Part 3: YOLOv5 vs YOLOv4

YOLOv4

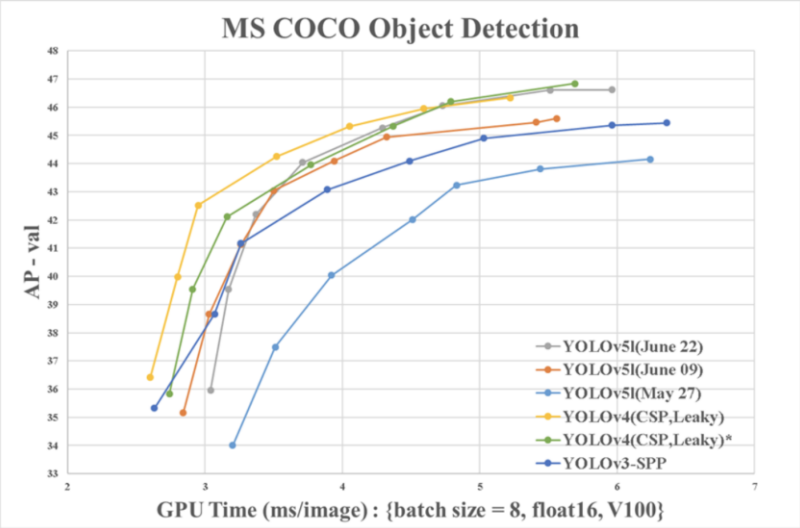
YOLOv4 introduced the concept of the **bag of freebies** and the **bag of specials**. In addition, AP (Average Precision) is 12% compared to YOLOv3.

bag of freebies: Techniques that bring about an enhancement in model performance without increasing the inference cost.

bag of specials: Techniques that increase accuracy while increasing the computation cost.

YOLOv5

YOLOv5 achieves **the same** if not better accuracy(mAP of 55.6) than the other YOLO models while taking **less computation power**.



Source: https://blog.roboflow.com/yolov5-improvements-and-evaluation/

Part 4: Hyperparameter Tuning in YOLOv3

The hyperparameter tuning focused on three hyperparameters:

- **The input size of an image:** Input training images are first resized to size width x height before training. Default values are 416×416. We can improve results if we increase it to 608×608, but it would take longer to train too.
- **The learning rate:** The parameter learning rate controls how aggressively we should learn based on the current batch of data. Usually this is a number between 0.01 and 0.0001.
  At the beginning of the training process, the learning rate needs to be high. But as the neural network sees a lot of data, the weights need to change less aggressively. In other words, the learning rate needs to be decreased over time. In the configuration file, this decrease in learning rate is accomplished by first specifying that our learning rate decreasing policy is steps.
- **The number of epochs used for training:** This is important because we want to avoid overfitting. We can use early stopping method for choosing correct amount of epochs.
- **Batch Size:** The batch parameter indicates the batch size used during training. Batch Size is chosen according to the available memory.
- **Subdivisions configuration:** lets you process a fraction of the batch size at one time on your GPU.
  You can start the training with subdivisions=1(There's no division), and if you get an Out of memory error, increase the subdivisions parameter by multiples of 2 till the training proceeds successfully.

We also can choose hyperparameters with wide **range, random search** approach:

- Image size = [224,320,416,608,800,1024]
- Learning Rate = [5e-3,1e-5,5e-4,1e-4,1e-5,5e-5]
- Epochs = [5,10,20,40,60,100]

Part 5:

	mAP@.5	mAP@.5:.95:	Precision	Recall	Training Time	Inference Time
YOLOv5x	0.879	0.572	0.905	0.84	64.02 min	25ms
YOLOv5s	0.881	0.569	0.92	0.819	63.18 min	20ms

Figure 60 The result of experiment

The largest YOLOv5 is YOLOv5x(roughly 370 MB). The training time and inference time in YOLOv5s is faster than YOLOv5x because YOLOv5s is smaller than YOLOv5x.

The precision and mAP 0.5 of YOLOv5s is better than YOLOv5x. But recall and mAP 0.5:0.95 of YOLOv5x is better than YOLOv5s.

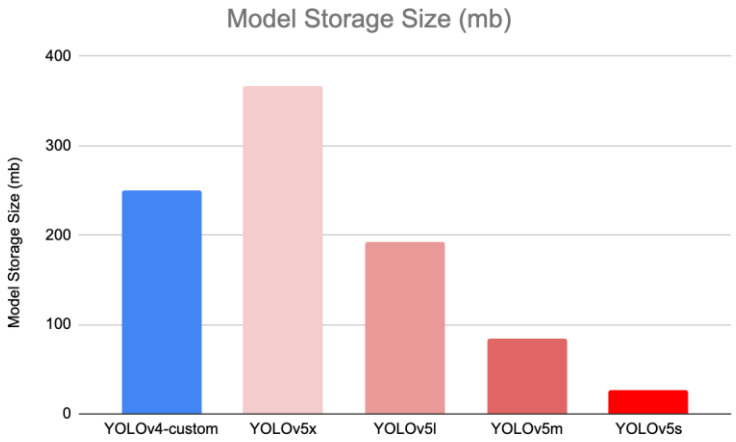


Figure 58 Model Storage Size

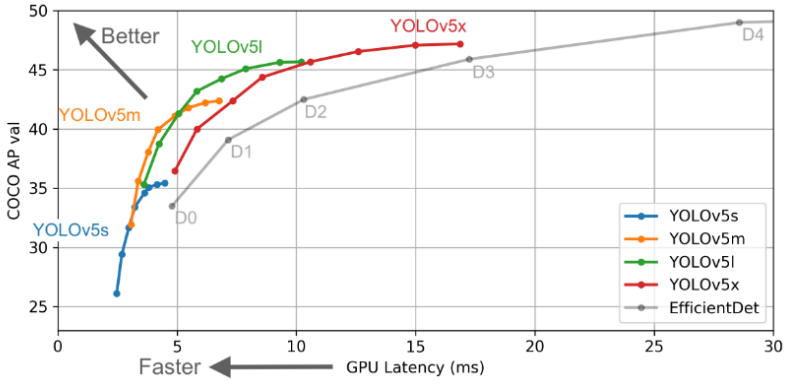


Figure 59 YOLO v5 stats

Advantages of bigger model(YOLOv5x)

The precision and mAP 0.5 of YOLOv5s is better than YOLOv5x. But recall and mAP 0.5:0.95 of YOLOv5x is better than YOLOv5s.

- Better Precision
- Better Recall
- Better mAP

Disadvantages of bigger model(YOLOv5x)

- Slower training time
- Slower inference time

Install Dependencies

In [3]:
```
# clone YOLOv5 repository
!git clone https://github.com/ultralytics/yolov5  # clone repo
%cd yolov5
!git reset --hard 886f1c03d839575afecb059accf74296fad395b6
```

In [4]:
```
# install dependencies as necessary
!pip install -qr requirements.txt  # install dependencies (ignore errors)
import torch

from IPython.display import Image, clear_output  # to display images
from utils.google_utils import gdrive_download  # to download models/datasets

# clear_output()
print('Setup complete. Using torch %s %s' % (torch.__version__, torch.cuda.get_device_properties(0) if torch.cuda.is_av
```

```
     |████████████████████████████████| 596 kB 4.1 MB/s eta 0:00:01
Setup complete. Using torch 1.10.0+cu111 _CudaDeviceProperties(name='Tesla P100-PCIE-16GB', major=6, minor=0, total_m
emory=16280MB, multi_processor_count=56)
```

Loading the dataset

I uploaded the dataset to Roboflow website. And then with generated API Key I loaded the dataset.

In [5]:
```
!pip install roboflow

from roboflow import Roboflow
rf = Roboflow(api_key="iuyWvFXwxBrdK01CPXty")
project = rf.workspace().project("bocce-ball")
dataset = project.version(1).download("yolov5")
```

YOLOv5s

Define Model Configuration and Architecture

We will write a yaml script that defines the parameters for our model like the number of classes, anchors, and each layer.

In [7]:
```
# define number of classes based on YAML
import yaml
with open(dataset.location + "/data.yaml", 'r') as stream:
    num_classes = str(yaml.safe_load(stream)['nc'])
```

In [8]:
```
#this is the model configuration we will use for our tutorial
%cat /content/yolov5/models/yolov5s.yaml
```

Train YOLOv5s

Here, we are able to pass a number of arguments:

1. **img:** define input image size
2. **batch:** determine batch size
3. **epochs:** define the number of training epochs. (Note: often, 3000+ are common here!)
4. **data:** set the path to our yaml file
5. **cfg:** specify our model configuration
6. **weights:** specify a custom path to weights.
7. **name:** result names
8. **nosave:** only save the final checkpoint
9. **cache:** cache images for faster training

```
In [12]: # train yolov5s on custom data for 100 epochs
         # time its performance
         %%time
         %cd /content/yolov5/
         !python train.py --img 416 --batch 16 --epochs 100 --data {dataset.location}/data.yaml --cfg
         ./models/custom_yolov5s.yaml --weights '' --name yolov5s_results  --cache
```

After 100 epochs of training (Wall time: 1h 3min 49s) the results for YOLOv5s are:

Class	Images	Targets	P	R	mAP@.5	mAP@.5:.95
all	330	5.61e+03	0.92	0.819	0.881	0.569
blue	330	654	0.983	0.983	0.993	0.635
green	330	1.63e+03	0.938	0.917	0.974	0.579
red	330	1.63e+03	0.924	0.981	0.989	0.674
vline	330	725	0.974	0.669	0.911	0.687
white	330	324	0.791	0.398	0.431	0.145
yellow	330	648	0.912	0.965	0.986	0.697

Figure 60 Training Results

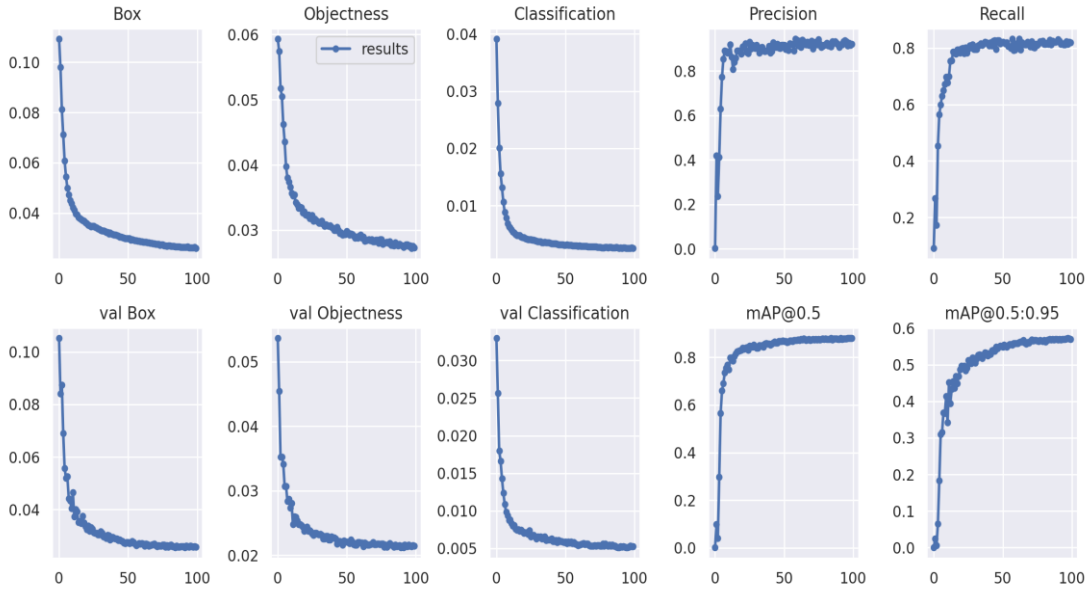


Figure 61 Plotting different metrics per epoch

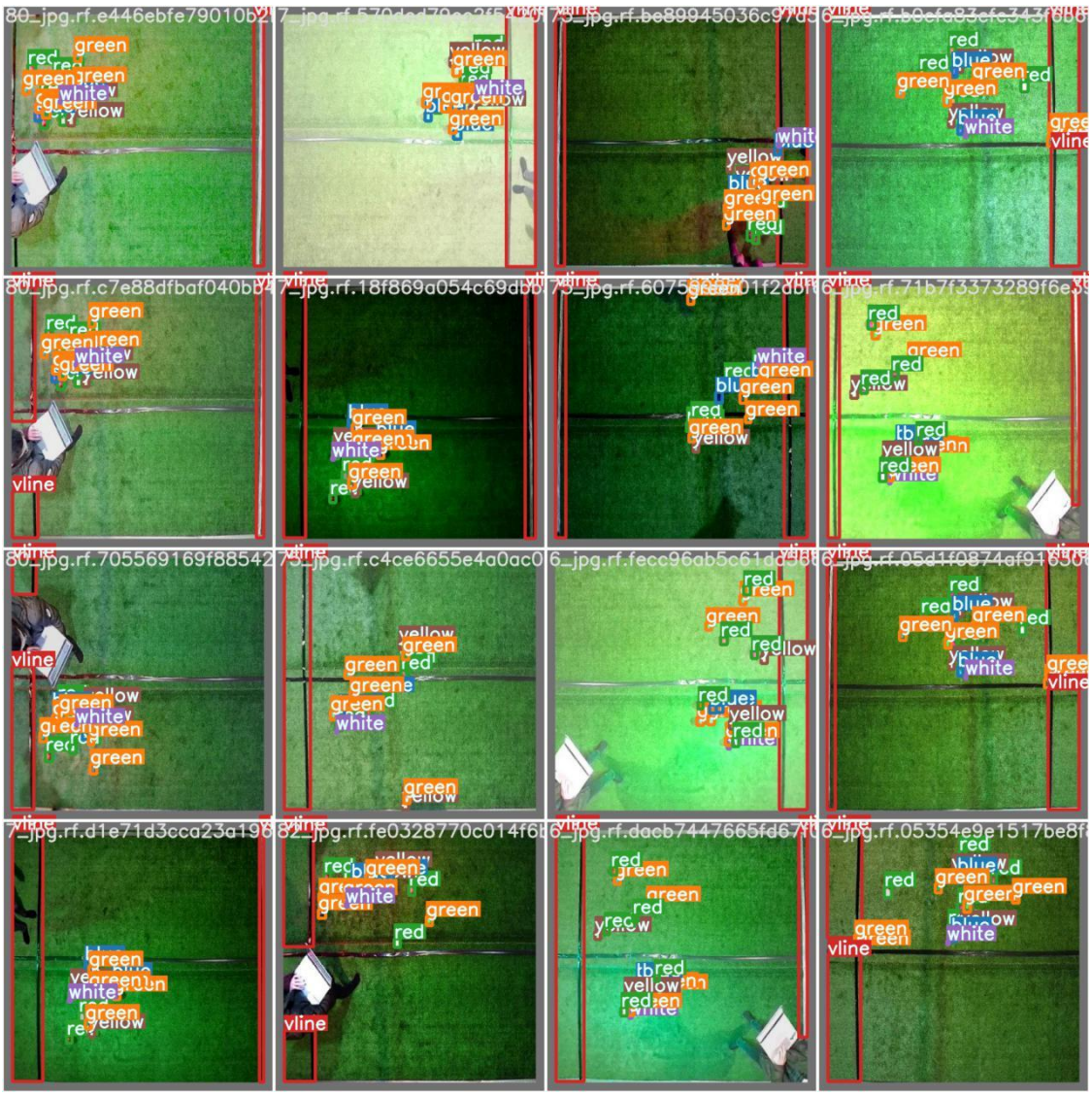


Figure 62 The balls detected by the model with original dataset

YOLOv5x

Define Model Configuration and Architecture

We will write a yaml script that defines the parameters for our model like the number of classes, anchors, and each layer.

```
In [20]: # define number of classes based on YAML
         import yaml
         with open(dataset.location + "/data.yaml", 'r') as stream:
             num_classes = str(yaml.safe_load(stream)['nc'])
```

```
In [21]: #this is the model configuration we will use for our tutorial
         %cat /content/yolov5/models/yolov5x.yaml
```

Train YOLOv5x

```
In [25]: # train yolov5s on custom data for 100 epochs
         # time its performance
         %%time
         %cd /content/yolov5/
         !python train.py --img 416 --batch 16 --epochs 100
         --data {dataset.location}/data.yaml --cfg ./models/custom_yolov5x.yaml --weights '' --name yolov5x_results  --cache
```

After 100 epochs of training (Wall time: 1h 4min 33s) the results for YOLOv5x are:

Class	Images	Targets	P	R	mAP@.5	mAP@.5:.95:
all	330	5.61e+03	0.905	0.84	0.879	0.572
blue	330	654	0.979	0.983	0.995	0.636
green	330	1.63e+03	0.94	0.942	0.98	0.591
red	330	1.63e+03	0.911	0.982	0.987	0.678
vline	330	725	0.968	0.746	0.911	0.69
white	330	324	0.73	0.401	0.415	0.145
yellow	330	648	0.901	0.986	0.987	0.693

Figure 63 Training Results

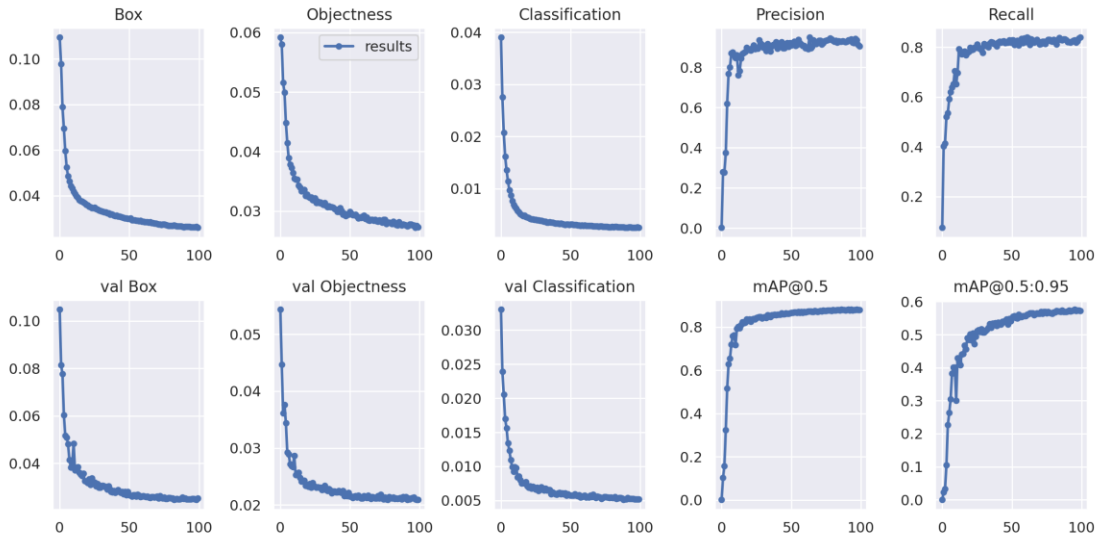


Figure 64 Plotting different metrics per epoch

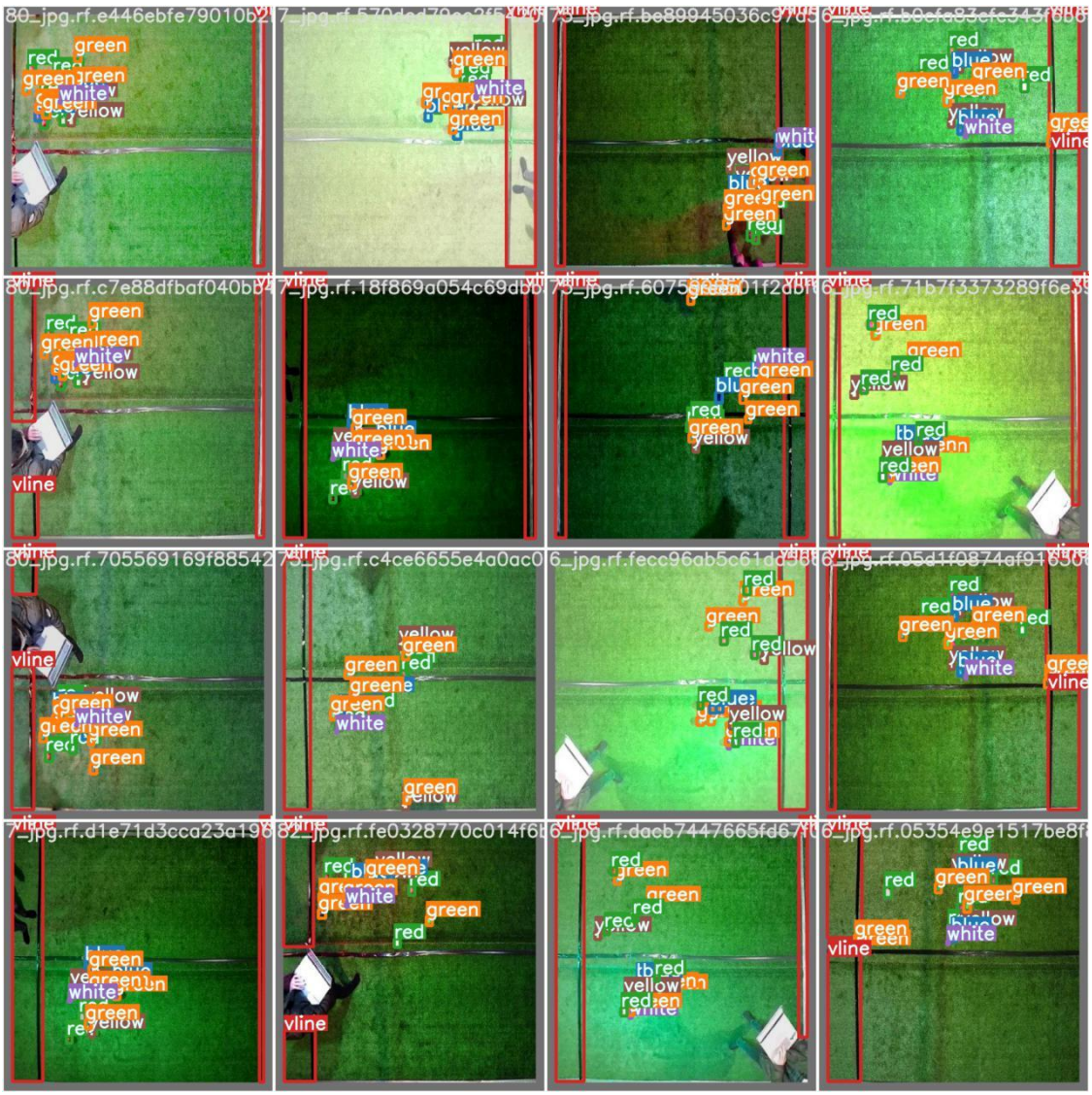


Figure 65 The balls detected by the model with original dataset