Programming Exercises

Convolutional Neural Networks

1. We load in the MNIST dataset as shown in the code.
2. We reshaped, scaled, and converted the data to float and the labels to one-hot vectors using the *to_categorical*() function from the Keras library. The images after preprocessing are shown in Fig.1.

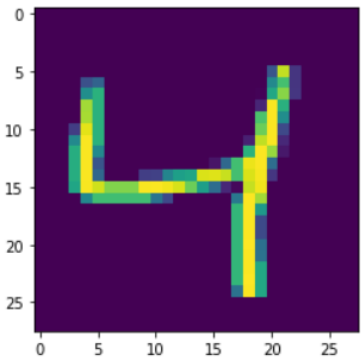


Figure 1: Preprocessed Image

3. We create the CNN as instructed in the homework, and the layers are accessed as shown in Fig.2.

```
[<keras.layers.convolutional.Conv2D at 0x21c2c291a30>,
 <keras.layers.pooling.MaxPooling2D at 0x21c2d532880>,
 <keras.layers.core.Flatten at 0x21c2d532ca0>,
 <keras.layers.core.Dense at 0x21c2d5f4040>,
 <keras.layers.core.Dense at 0x21bfd3bf6a0>]
```

Figure 2: Layers of the CNN

4. On evaluating the CNN once it's done training, the accuracy turns out to be 98.80% with a loss of 0.04.

```
score = cnn.evaluate(testX, testY, verbose=0)
print(score)
```

```
[0.04158346354961395, 0.9880999922752238]
```

Figure 3: Accuracy on test set

5. (a) After training the model for 50 epochs, we plot the training and validation accuracies (Fig.4 and Fig.5). As can be seen in the figures, there is no significant benefit to training for longer. In fact, the validation accuracy starts to dip as the epochs roll on, indicating overfitting of the training set.

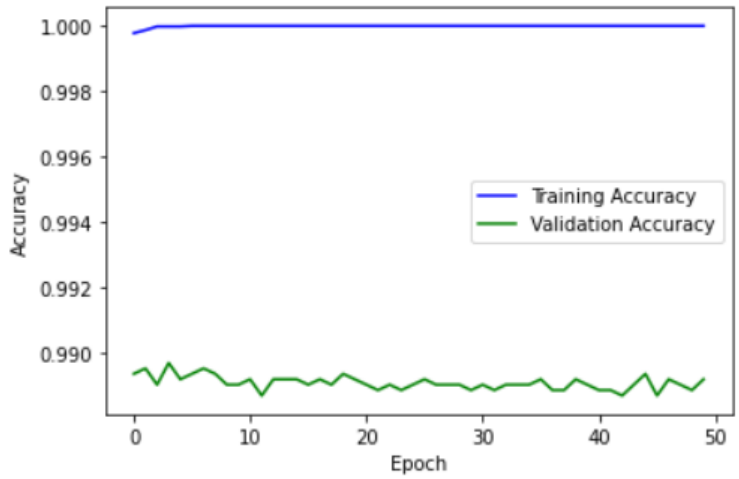


Figure 4: Accuracy over epochs

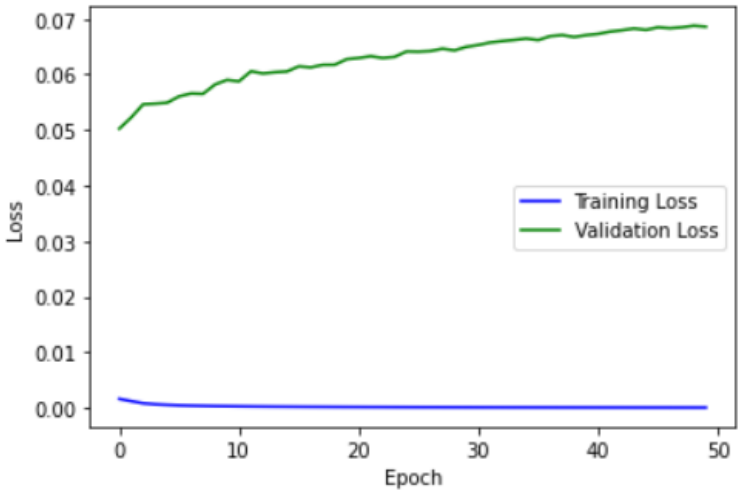


Figure 5: Loss accuracy over epochs

(b) To counter this overfitting, we introduce the dropout layer in the network as instructed in the problem. This does reduce the overfitting problem, as expected (Fig.6 and Fig.7)

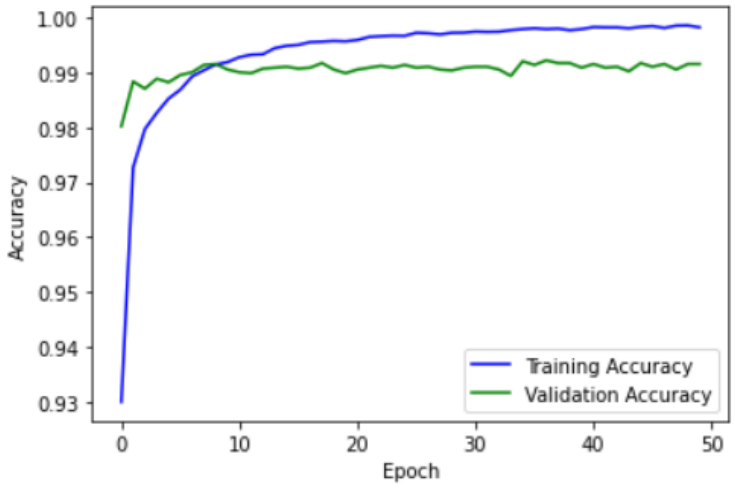


Figure 6: Accuracy over epochs with dropout layer

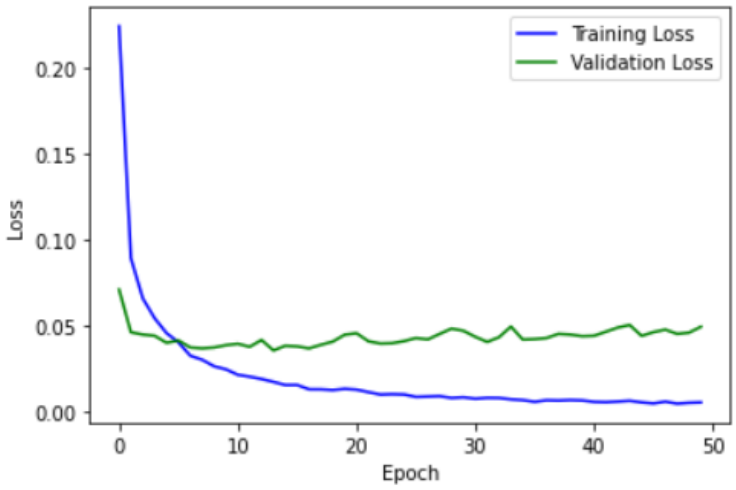


Figure 7: Loss accuracy over epochs with dropout layer

(c) We added in the extra convolution layer as instructed. the accuracy on the test set turned out to be 99.88% with a loss of 0.03.

```
score = cnn_add.evaluate(testX, testY, verbose=0)
print(score)
```

```
[0.027527863159775734, 0.9915000200271606]
```

Figure 8: Accuracy on test set with an extra convolution layer

(d) Upon experimenting with the learning rate, we find that the accuracy drops drastically when the learning rate is set to 0.1

```
score = cnn_low.evaluate(testX, testY, verbose=0)
print(score)
```

```
[0.03898925334215164, 0.9871000051498413]
```

Figure 9: Accuracy on test set with 0.001 learning rate

```
score = cnn_high.evaluate(testX, testY, verbose=0)
print(score)
```

```
[2.31148624420166, 0.11349999904632568]
```

Figure 10: Accuracy on test set with 0.1 learning rate

6. (a) After introducing the dropout layer, the validation loss hovers around the same value. The validation accuracy slowly increases and becomes relatively stable as the epochs happen, These facts imply that the overfitting problem is addressed by adding the dropout layer.

   (b) When the additional convolution layer is added, the accuracy on the test set increases (albeit slightly) and the loss decreases,

   (c) When the learning rate is changed to 0.1, there is a very steep drop in the accuracy and a sharp increase in the loss. Decreasing the learning rate to 0.001 had a similar effect (just way less pronounced).

Random Forest for Image Approximation

1. We download the Mona Lisa and import it into our workspace.
2. We sample 5000 points from the image. One of the main benefits of random forests is that they require little to no preprocessing. therefore, we do not perform and extra pre-processing steps.
3. We decide to regress all the three channels (R, G and B) at once. Besides the functin to perform that, we don't do any other preprocessing since random forests don't require much in the way of that.
4. We implement random forests using sklean's implementation of the same (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html). The output is as shown in Fig.11.

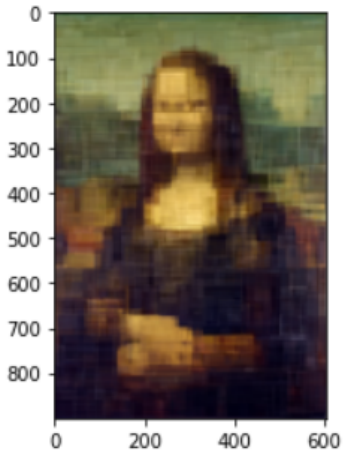


Figure 11: Output of Random Forest

5. (a) Increasing the depth as shown in the code leads to a progressively more detailed image. This can be attributed to the fact that the higher the depth of the tree, the greater its capacity to represent features since the image can be divided into more sections.

   (b) Upon increasing the number of trees while keeping the depth constant, we find that the improvement is relatively much more negligible. The edges in the images (as can be seen in the code) get smoother and smoother as the number of trees goes up. This relative lack of improvement with increase can be attributed to the fact that having more points where there can be a split in the image's representation (a higher depth) has a much more pronounced impact than just having more trees to work with.

(c) Upon running the KNN algorithm (as can be seen in the code), we find that the output image is much more 'smooth'. This is likely an effect of the way KNNs work, in that they use the pixel nearest to the one in question to assign it a value.

(d) For the pruning strategies, we tweaked the *min_samples_split* and *min_samples_leaf* arguments to the regressor. Both of these strategies resulted in an output that was more 'blurred' but had no significant improvement associated with them.

6. (a) The decision tree makes a decision about whether to split the node further depending upon whether the coordinates of the pixel are under or over the threshold.

$$r(x) = \begin{cases} \textbf{true} & \text{if } x_j \leq t \\ \textbf{false} & \text{if } x_j > t \end{cases}$$

Figure 12: Basic Decision tree criteria

(b) The image is composed of boxes whose color and bounds are decided by the decision trees within the random forest. The deeper the decision tree, the more granular and hence detailed the output is the image. The patches are a function of the depth of the tree and can be computed using the formula 2^{depth}.

Written Exercises

Maximum Margin Classifiers

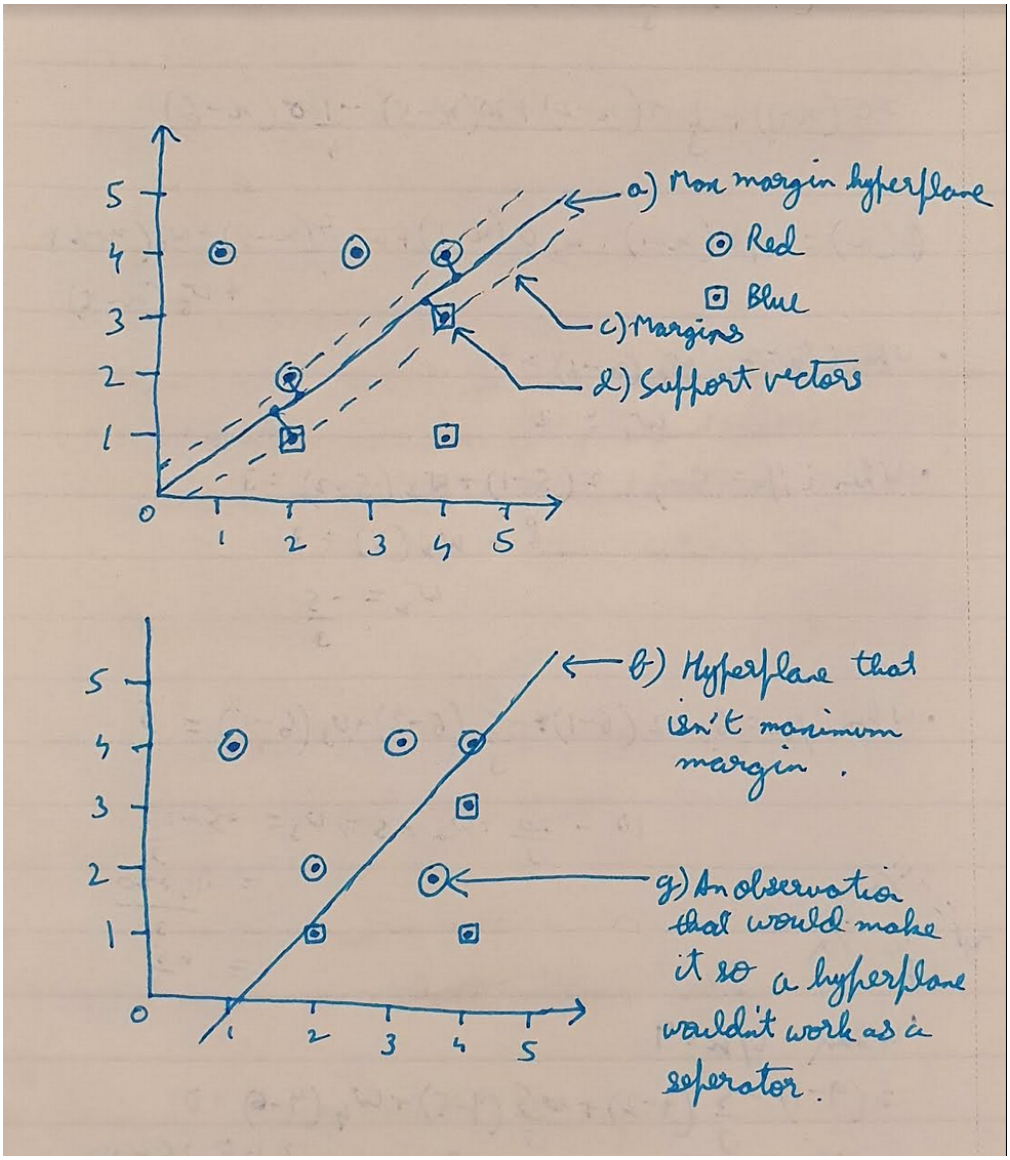


Figure 13: Parts (a), (c), (d), (f), (g)

(b) The equation for the maximum margin hyperplane turns out to be $y - x = -\frac{1}{2}$. Therefore, the classifications rule is: classify to Red if $y - x > -\frac{1}{2}$, classify to Blue if not (where $x = X1$ and $y = X2$).
As for part (e), the last observation is not one of the support vectors, nor is it anywhere near

the decision boundary, which means that any "slight" change will not impact the maximum marginal hyperplane.

Neural Networks as Function Approximators

We are going to represent this function using a neural network that has one hidden layer with 4 neurons. We calculate the biases and the weights as follows:

$$
\begin{aligned}
f(x) = w_1\sigma(x-1) + w_2\sigma(x-2) + w_3\sigma(x-5) + w_4\sigma(x-6) \\
When x = 2, \\
w_1(x-1) = 2 \\
w_1 = 2 \\
When x = 5, \\
2(5-1) + w_2(5-2) = 3 \\
8 + w_2(3) = 3 \\
w_= -\frac{5}{3} \\
When x = 6, \\
2(6-1) - \frac{5}{3}(6-2) + w_3(6-5) = 5 \\
10 - \frac{20}{3} + w_3 = 5 \\
w_3 = -5 + \frac{20}{3} \\
w_3 = \frac{5}{3} \\
When x = 9, \\
2(9-1) - \frac{5}{3}(9-2) + \frac{5}{3}(9-5) + w_4(9-6) = 0 \\
16 - \frac{35}{3} + \frac{20}{3} + 3w_4 = 0 \\
w_4 = -\frac{11}{9}
\end{aligned}
$$

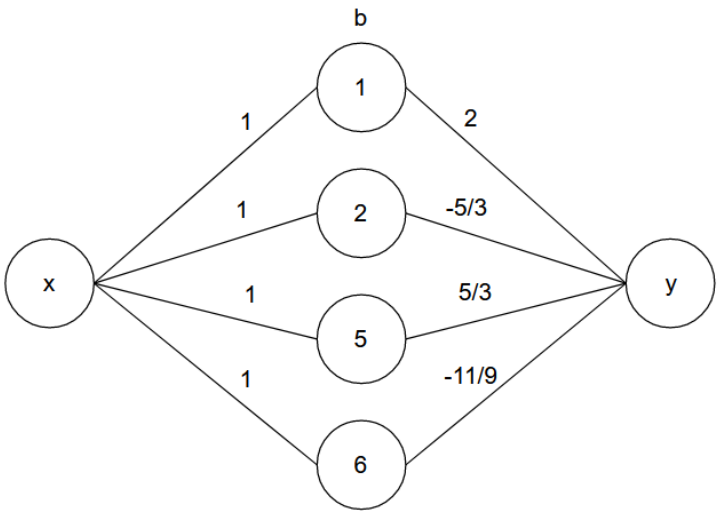
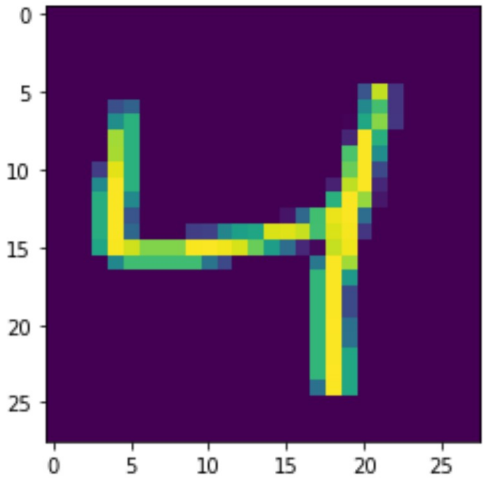


Figure 14: Neural Network Architecture

In [1]:
```
import numpy as np
import keras
from keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
```

In [2]:
```
(trainX, trainY),(testX, testY) = mnist.load_data( )
```

In [3]:
```
x = trainX[2]
plt.imshow(x)
plt.show()
```

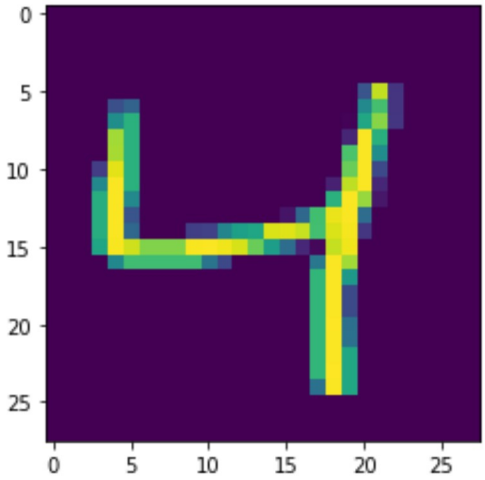


Preprocessing

In [4]:
```
def preprocessing(data, labels):
    data = np.reshape(data, newshape=(data.shape[0], data.shape[1], data.shape
    data = data / 255
    data = data.astype(dtype='float64')

    labels = to_categorical(labels, num_classes=10, dtype='float64')

    return data, labels
```

In [5]:
```
trainX, trainY = preprocessing(trainX, trainY)
testX, testY = preprocessing(testX, testY)
```

In [6]:
```
x = trainX[2]
plt.imshow(x)
plt.show()
```



Implementation

In [7]:
```
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from tensorflow.keras.optimizers import SGD
```

In [8]:
```
def create_cnn(learning_rate=0.01):
    model = Sequential()
    model.add(Conv2D(32,(3,3), activation='relu', kernel_initializer='he_unif
    model.add(MaxPooling2D((2,2)))
    model.add(Flatten())
    model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(10,activation='softmax'))
    opt = SGD(lr=learning_rate, momentum=0.9)

    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['a

    return model
```

In [10]:
```
cnn = create_cnn()
```

```
C:\Users\sidar\AppData\Local\Programs\Python\Python39\lib\site-packages\keras\
optimizer_v2\optimizer_v2.py:355: UserWarning: The `lr` argument is deprecate
d, use `learning_rate` instead.
  warnings.warn(
```

In [11]:
```
cnn.layers
```

Out[11]:
```
[<keras.layers.convolutional.Conv2D at 0x21c2c291a30>,
 <keras.layers.pooling.MaxPooling2D at 0x21c2d532880>,
 <keras.layers.core.Flatten at 0x21c2d532ca0>,
 <keras.layers.core.Dense at 0x21c2d5f4040>,
```

In [12]:
```
cnn.fit(trainX, trainY, batch_size=32, epochs=10, validation_split=0.1)
```

```
Epoch 1/10
1688/1688 [==============================] - 13s 8ms/step - loss: 0.1777 - acc
uracy: 0.9454 - val_loss: 0.0639 - val_accuracy: 0.9817
Epoch 2/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0610 - acc
uracy: 0.9817 - val_loss: 0.0524 - val_accuracy: 0.9860
Epoch 3/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0389 - acc
uracy: 0.9886 - val_loss: 0.0559 - val_accuracy: 0.9852
Epoch 4/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0272 - acc
uracy: 0.9918 - val_loss: 0.0475 - val_accuracy: 0.9880
Epoch 5/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0181 - acc
uracy: 0.9948 - val_loss: 0.0470 - val_accuracy: 0.9878
Epoch 6/10
1688/1688 [==============================] - 13s 7ms/step - loss: 0.0123 - acc
uracy: 0.9964 - val_loss: 0.0488 - val_accuracy: 0.9878
Epoch 7/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0080 - acc
uracy: 0.9981 - val_loss: 0.0512 - val_accuracy: 0.9875
Epoch 8/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0053 - acc
uracy: 0.9989 - val_loss: 0.0511 - val_accuracy: 0.9875
Epoch 9/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0037 - acc
uracy: 0.9994 - val_loss: 0.0525 - val_accuracy: 0.9880
Epoch 10/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0026 - acc
uracy: 0.9997 - val_loss: 0.0529 - val_accuracy: 0.9893
```

Out[12]:
```
<keras.callbacks.History at 0x1ca2f1fb070>
```

In [13]:
```
score = cnn.evaluate(testX, testY, verbose=0)
print(score)
```

```
[0.04158346354961395, 0.988099992275238]
```

Experimentation

In [93]:
```
epoch_history = cnn.fit(trainX, trainY, batch_size=32, epochs=50, validation_s
```

```
Epoch 1/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0016 - acc
uracy: 0.9998 - val_loss: 0.0502 - val_accuracy: 0.9893
Epoch 2/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0012 - acc
uracy: 0.9999 - val_loss: 0.0523 - val_accuracy: 0.9895
Epoch 3/50
1688/1688 [==============================] - 13s 8ms/step - loss: 8.2430e-04 -
accuracy: 1.0000 - val_loss: 0.0546 - val_accuracy: 0.9890
Epoch 4/50
```

```
1688/1688 [==============================] - 13s 8ms/step - loss: 6.4579e-04 -
accuracy: 1.0000 - val_loss: 0.0547 - val_accuracy: 0.9897
Epoch 5/50
1688/1688 [==============================] - 13s 8ms/step - loss: 5.4154e-04 -
accuracy: 1.0000 - val_loss: 0.0549 - val_accuracy: 0.9892
Epoch 6/50
1688/1688 [==============================] - 13s 8ms/step - loss: 4.5755e-04 -
accuracy: 1.0000 - val_loss: 0.0560 - val_accuracy: 0.9893
Epoch 7/50
1688/1688 [==============================] - 12s 7ms/step - loss: 4.0050e-04 -
accuracy: 1.0000 - val_loss: 0.0566 - val_accuracy: 0.9895
Epoch 8/50
1688/1688 [==============================] - 12s 7ms/step - loss: 3.6020e-04 -
accuracy: 1.0000 - val_loss: 0.0565 - val_accuracy: 0.9893
Epoch 9/50
1688/1688 [==============================] - 12s 7ms/step - loss: 3.2044e-04 -
accuracy: 1.0000 - val_loss: 0.0582 - val_accuracy: 0.9890
Epoch 10/50
1688/1688 [==============================] - 12s 7ms/step - loss: 2.9658e-04 -
accuracy: 1.0000 - val_loss: 0.0590 - val_accuracy: 0.9890
Epoch 11/50
1688/1688 [==============================] - 12s 7ms/step - loss: 2.7511e-04 -
accuracy: 1.0000 - val_loss: 0.0587 - val_accuracy: 0.9892
Epoch 12/50
1688/1688 [==============================] - 12s 7ms/step - loss: 2.5501e-04 -
accuracy: 1.0000 - val_loss: 0.0606 - val_accuracy: 0.9887
Epoch 13/50
1688/1688 [==============================] - 12s 7ms/step - loss: 2.3791e-04 -
accuracy: 1.0000 - val_loss: 0.0602 - val_accuracy: 0.9892
Epoch 14/50
1688/1688 [==============================] - 12s 7ms/step - loss: 2.2201e-04 -
accuracy: 1.0000 - val_loss: 0.0604 - val_accuracy: 0.9892
Epoch 15/50
1688/1688 [==============================] - 12s 7ms/step - loss: 2.1050e-04 -
accuracy: 1.0000 - val_loss: 0.0605 - val_accuracy: 0.9892
Epoch 16/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.9868e-04 -
accuracy: 1.0000 - val_loss: 0.0614 - val_accuracy: 0.9890
Epoch 17/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.8680e-04 -
accuracy: 1.0000 - val_loss: 0.0613 - val_accuracy: 0.9892
Epoch 18/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.7986e-04 -
accuracy: 1.0000 - val_loss: 0.0617 - val_accuracy: 0.9890
Epoch 19/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.6921e-04 -
accuracy: 1.0000 - val_loss: 0.0617 - val_accuracy: 0.9893
Epoch 20/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.6157e-04 -
accuracy: 1.0000 - val_loss: 0.0627 - val_accuracy: 0.9892
Epoch 21/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.5641e-04 -
accuracy: 1.0000 - val_loss: 0.0629 - val_accuracy: 0.9890
Epoch 22/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.4801e-04 -
accuracy: 1.0000 - val_loss: 0.0633 - val_accuracy: 0.9888
Epoch 23/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.4329e-04 -
accuracy: 1.0000 - val_loss: 0.0629 - val_accuracy: 0.9890
```

```
Epoch 24/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.3731e-04 -
accuracy: 1.0000 - val_loss: 0.0631 - val_accuracy: 0.9888
Epoch 25/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.3169e-04 -
accuracy: 1.0000 - val_loss: 0.0641 - val_accuracy: 0.9890
Epoch 26/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.2641e-04 -
accuracy: 1.0000 - val_loss: 0.0641 - val_accuracy: 0.9892
Epoch 27/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.2302e-04 -
accuracy: 1.0000 - val_loss: 0.0642 - val_accuracy: 0.9890
Epoch 28/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.1828e-04 -
accuracy: 1.0000 - val_loss: 0.0646 - val_accuracy: 0.9890
Epoch 29/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.1379e-04 -
accuracy: 1.0000 - val_loss: 0.0643 - val_accuracy: 0.9890
Epoch 30/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.1079e-04 -
accuracy: 1.0000 - val_loss: 0.0650 - val_accuracy: 0.9888
Epoch 31/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.0625e-04 -
accuracy: 1.0000 - val_loss: 0.0653 - val_accuracy: 0.9890
Epoch 32/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.0343e-04 -
accuracy: 1.0000 - val_loss: 0.0657 - val_accuracy: 0.9888
Epoch 33/50
1688/1688 [==============================] - 12s 7ms/step - loss: 1.0060e-04 -
accuracy: 1.0000 - val_loss: 0.0660 - val_accuracy: 0.9890
Epoch 34/50
1688/1688 [==============================] - 12s 7ms/step - loss: 9.7359e-05 -
accuracy: 1.0000 - val_loss: 0.0662 - val_accuracy: 0.9890
Epoch 35/50
1688/1688 [==============================] - 12s 7ms/step - loss: 9.4678e-05 -
accuracy: 1.0000 - val_loss: 0.0665 - val_accuracy: 0.9890
Epoch 36/50
1688/1688 [==============================] - 12s 7ms/step - loss: 9.2157e-05 -
accuracy: 1.0000 - val_loss: 0.0662 - val_accuracy: 0.9892
Epoch 37/50
1688/1688 [==============================] - 12s 7ms/step - loss: 8.9538e-05 -
accuracy: 1.0000 - val_loss: 0.0669 - val_accuracy: 0.9888
Epoch 38/50
1688/1688 [==============================] - 12s 7ms/step - loss: 8.7372e-05 -
accuracy: 1.0000 - val_loss: 0.0671 - val_accuracy: 0.9888
Epoch 39/50
1688/1688 [==============================] - 12s 7ms/step - loss: 8.4737e-05 -
accuracy: 1.0000 - val_loss: 0.0667 - val_accuracy: 0.9892
Epoch 40/50
1688/1688 [==============================] - 12s 7ms/step - loss: 8.2642e-05 -
accuracy: 1.0000 - val_loss: 0.0671 - val_accuracy: 0.9890
Epoch 41/50
1688/1688 [==============================] - 12s 7ms/step - loss: 8.0612e-05 -
accuracy: 1.0000 - val_loss: 0.0673 - val_accuracy: 0.9888
Epoch 42/50
1688/1688 [==============================] - 12s 7ms/step - loss: 7.8473e-05 -
accuracy: 1.0000 - val_loss: 0.0677 - val_accuracy: 0.9888
Epoch 43/50
1688/1688 [==============================] - 12s 7ms/step - loss: 7.6682e-05 -
```

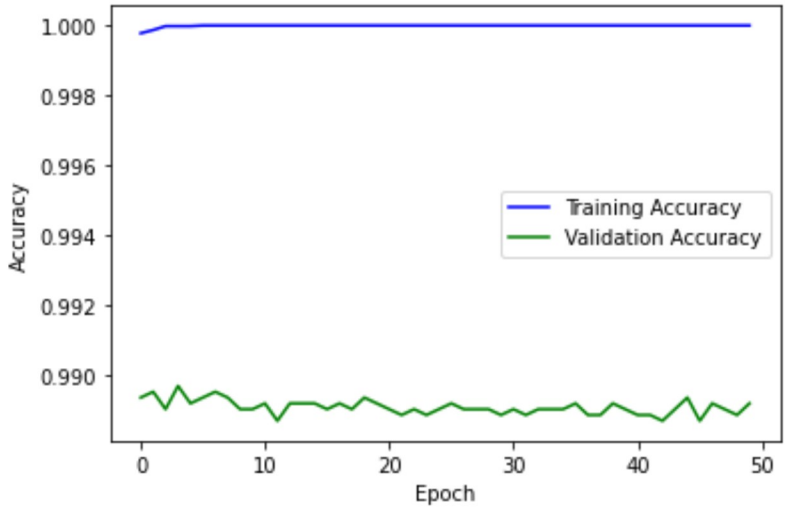
```
accuracy: 1.0000 - val_loss: 0.0679 - val_accuracy: 0.9887
Epoch 44/50
1688/1688 [==============================] - 12s 7ms/step - loss: 7.5278e-05 -
accuracy: 1.0000 - val_loss: 0.0683 - val_accuracy: 0.9890
Epoch 45/50
1688/1688 [==============================] - 12s 7ms/step - loss: 7.3418e-05 -
accuracy: 1.0000 - val_loss: 0.0680 - val_accuracy: 0.9893
Epoch 46/50
1688/1688 [==============================] - 12s 7ms/step - loss: 7.1732e-05 -
accuracy: 1.0000 - val_loss: 0.0685 - val_accuracy: 0.9887
Epoch 47/50
1688/1688 [==============================] - 12s 7ms/step - loss: 7.0046e-05 -
accuracy: 1.0000 - val_loss: 0.0683 - val_accuracy: 0.9892
Epoch 48/50
1688/1688 [==============================] - 12s 7ms/step - loss: 6.8504e-05 -
accuracy: 1.0000 - val_loss: 0.0685 - val_accuracy: 0.9890
Epoch 49/50
1688/1688 [==============================] - 12s 7ms/step - loss: 6.7054e-05 -
accuracy: 1.0000 - val_loss: 0.0688 - val_accuracy: 0.9888
Epoch 50/50
1688/1688 [==============================] - 12s 7ms/step - loss: 6.5691e-05 -
```

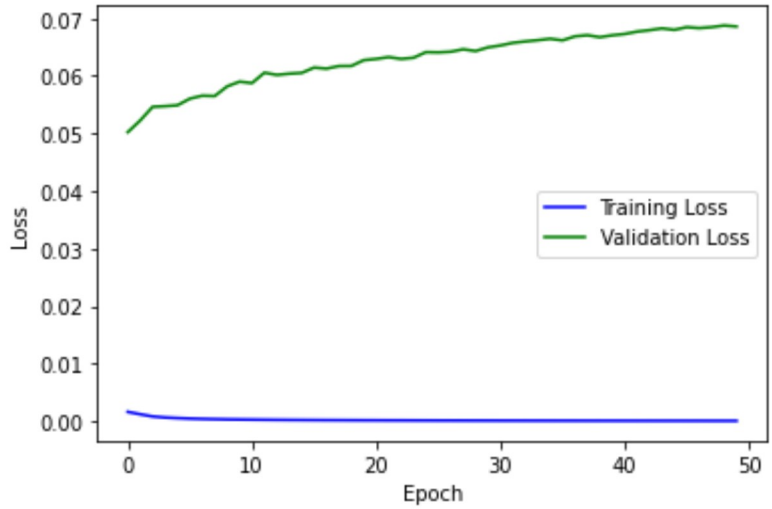
In [97]:

```
def display(history):
    train_accuracy = history.history['accuracy']
    train_loss = history.history['loss']
    val_accuracy = history.history['val_accuracy']
    val_loss = history.history['val_loss']
    x_values = list(range(0,len(train_accuracy)))

    plt.plot(x_values, train_accuracy, 'b-', label='Training Accuracy')
    plt.plot(x_values, val_accuracy, 'g-', label='Validation Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()

    plt.plot(x_values, train_loss, 'b-', label='Training Loss')
    plt.plot(x_values, val_loss,'g-', label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss' )
    plt.legend()
    plt.show()
```

In [98]:

```
display(epoch_history)
```





In [103…

```
def create_cnn_dropout(learning_rate=0.01):
    model = Sequential()
    model.add(Conv2D(32,(3,3), activation='relu', kernel_initializer='he_unif
    model.add(MaxPooling2D((2,2)))
    model.add(Flatten())

    model.add(Dropout(0.5))

    model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(10,activation='softmax'))
    opt = SGD(lr=learning_rate, momentum=0.9)

    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['a

    return model
```

In [104…

```
cnn_dropout = create_cnn_dropout()
```

In [105…

```
epoch_history_dropout = cnn_dropout.fit(trainX, trainY, batch_size=32, epochs=
```

```
Epoch 1/50
1688/1688 [==============================] - 14s 8ms/step - loss: 0.2241 - acc
```

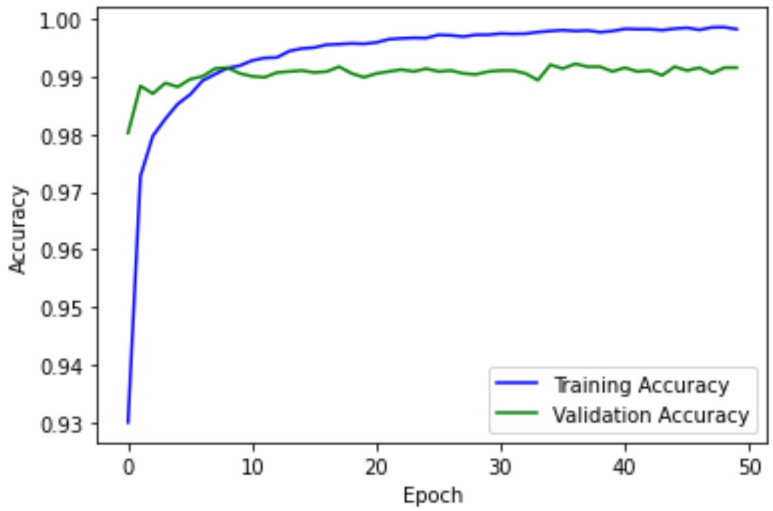
```
uracy: 0.9300 - val_loss: 0.0708 - val_accuracy: 0.9802
Epoch 2/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0890 - acc
uracy: 0.9727 - val_loss: 0.0459 - val_accuracy: 0.9883
Epoch 3/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0656 - acc
uracy: 0.9797 - val_loss: 0.0446 - val_accuracy: 0.9870
Epoch 4/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0541 - acc
uracy: 0.9826 - val_loss: 0.0439 - val_accuracy: 0.9888
Epoch 5/50
1688/1688 [==============================] - 14s 8ms/step - loss: 0.0454 - acc
uracy: 0.9852 - val_loss: 0.0398 - val_accuracy: 0.9882
Epoch 6/50
1688/1688 [==============================] - 14s 9ms/step - loss: 0.0401 - acc
uracy: 0.9868 - val_loss: 0.0412 - val_accuracy: 0.9895
Epoch 7/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0323 - acc
uracy: 0.9892 - val_loss: 0.0371 - val_accuracy: 0.9900
Epoch 8/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0299 - acc
uracy: 0.9904 - val_loss: 0.0366 - val_accuracy: 0.9913
Epoch 9/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0262 - acc
uracy: 0.9914 - val_loss: 0.0371 - val_accuracy: 0.9915
Epoch 10/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0244 - acc
uracy: 0.9919 - val_loss: 0.0384 - val_accuracy: 0.9905
Epoch 11/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0212 - acc
uracy: 0.9927 - val_loss: 0.0392 - val_accuracy: 0.9900
Epoch 12/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0201 - acc
uracy: 0.9932 - val_loss: 0.0375 - val_accuracy: 0.9898
Epoch 13/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0187 - acc
uracy: 0.9933 - val_loss: 0.0415 - val_accuracy: 0.9907
Epoch 14/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0171 - acc
uracy: 0.9944 - val_loss: 0.0354 - val_accuracy: 0.9908
Epoch 15/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0152 - acc
uracy: 0.9948 - val_loss: 0.0381 - val_accuracy: 0.9910
Epoch 16/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0153 - acc
uracy: 0.9950 - val_loss: 0.0377 - val_accuracy: 0.9907
Epoch 17/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0127 - acc
uracy: 0.9955 - val_loss: 0.0366 - val_accuracy: 0.9908
Epoch 18/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0127 - acc
uracy: 0.9956 - val_loss: 0.0386 - val_accuracy: 0.9917
Epoch 19/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0123 - acc
uracy: 0.9957 - val_loss: 0.0405 - val_accuracy: 0.9905
Epoch 20/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0130 - acc
uracy: 0.9956 - val_loss: 0.0445 - val_accuracy: 0.9898
Epoch 21/50
```

```
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0126 - acc
uracy: 0.9959 - val_loss: 0.0454 - val_accuracy: 0.9905
Epoch 22/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0111 - acc
uracy: 0.9964 - val_loss: 0.0407 - val_accuracy: 0.9908
Epoch 23/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0097 - acc
uracy: 0.9966 - val_loss: 0.0394 - val_accuracy: 0.9912
Epoch 24/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0099 - acc
uracy: 0.9967 - val_loss: 0.0396 - val_accuracy: 0.9908
Epoch 25/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0097 - acc
uracy: 0.9966 - val_loss: 0.0408 - val_accuracy: 0.9913
Epoch 26/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0083 - acc
uracy: 0.9972 - val_loss: 0.0426 - val_accuracy: 0.9908
Epoch 27/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0085 - acc
uracy: 0.9971 - val_loss: 0.0418 - val_accuracy: 0.9910
Epoch 28/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0087 - acc
uracy: 0.9969 - val_loss: 0.0450 - val_accuracy: 0.9905
Epoch 29/50
1688/1688 [==============================] - 14s 8ms/step - loss: 0.0077 - acc
uracy: 0.9972 - val_loss: 0.0481 - val_accuracy: 0.9903
Epoch 30/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0081 - acc
uracy: 0.9972 - val_loss: 0.0468 - val_accuracy: 0.9908
Epoch 31/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0073 - acc
uracy: 0.9974 - val_loss: 0.0433 - val_accuracy: 0.9910
Epoch 32/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0078 - acc
uracy: 0.9974 - val_loss: 0.0403 - val_accuracy: 0.9910
Epoch 33/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0077 - acc
uracy: 0.9974 - val_loss: 0.0431 - val_accuracy: 0.9905
Epoch 34/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0069 - acc
uracy: 0.9977 - val_loss: 0.0493 - val_accuracy: 0.9893
Epoch 35/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0065 - acc
uracy: 0.9979 - val_loss: 0.0417 - val_accuracy: 0.9920
Epoch 36/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0055 - acc
uracy: 0.9980 - val_loss: 0.0420 - val_accuracy: 0.9913
Epoch 37/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0064 - acc
uracy: 0.9979 - val_loss: 0.0426 - val_accuracy: 0.9922
Epoch 38/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0062 - acc
uracy: 0.9979 - val_loss: 0.0450 - val_accuracy: 0.9917
Epoch 39/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0065 - acc
uracy: 0.9976 - val_loss: 0.0446 - val_accuracy: 0.9917
Epoch 40/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0063 - acc
uracy: 0.9979 - val_loss: 0.0437 - val_accuracy: 0.9908
```

```
Epoch 41/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0054 - acc
uracy: 0.9982 - val_loss: 0.0439 - val_accuracy: 0.9915
Epoch 42/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0053 - acc
uracy: 0.9982 - val_loss: 0.0463 - val_accuracy: 0.9908
Epoch 43/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0056 - acc
uracy: 0.9982 - val_loss: 0.0487 - val_accuracy: 0.9910
Epoch 44/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0061 - acc
uracy: 0.9980 - val_loss: 0.0502 - val_accuracy: 0.9902
Epoch 45/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0052 - acc
uracy: 0.9983 - val_loss: 0.0438 - val_accuracy: 0.9917
Epoch 46/50
1688/1688 [==============================] - 14s 8ms/step - loss: 0.0045 - acc
uracy: 0.9984 - val_loss: 0.0460 - val_accuracy: 0.9910
Epoch 47/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0056 - acc
uracy: 0.9981 - val_loss: 0.0475 - val_accuracy: 0.9915
Epoch 48/50
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0044 - acc
uracy: 0.9985 - val_loss: 0.0450 - val_accuracy: 0.9905
Epoch 49/50
1688/1688 [==============================] - 14s 8ms/step - loss: 0.0049 - acc
uracy: 0.9985 - val_loss: 0.0457 - val_accuracy: 0.9915
Epoch 50/50
1688/1688 [==============================] - 14s 8ms/step - loss: 0.0051 - acc
uracy: 0.9982 - val_loss: 0.0491 - val_accuracy: 0.9915
```

In [106…

```
display(epoch_history_dropout)
```



In [110…

```
def create_cnn_add(learning_rate=0.01):
    model = Sequential()
    model.add(Conv2D(32,(3,3), activation='relu', kernel_initializer='he_unif
    model.add(MaxPooling2D((2,2)))
    model.add(Conv2D(64,(3,3), activation='relu', kernel_initializer='he_unif
    model.add(MaxPooling2D((2,2)))
    model.add(Flatten())
    model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(10,activation='softmax'))
    opt = SGD(lr=learning_rate, momentum=0.9)

    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['a

    return model
```

In [111…

```
cnn_add = create_cnn_add()
```

In [112…

```
cnn_add.fit(trainX, trainY, batch_size=32, epochs=10, validation_split=0.1)
```

```
Epoch 1/10
1688/1688 [==============================] - 16s 9ms/step - loss: 0.1418 - acc
uracy: 0.9554 - val_loss: 0.0546 - val_accuracy: 0.9845
Epoch 2/10
1688/1688 [==============================] - 15s 9ms/step - loss: 0.0474 - acc
uracy: 0.9853 - val_loss: 0.0431 - val_accuracy: 0.9873
Epoch 3/10
1688/1688 [==============================] - 15s 9ms/step - loss: 0.0314 - acc
uracy: 0.9898 - val_loss: 0.0392 - val_accuracy: 0.9882
Epoch 4/10
1688/1688 [==============================] - 14s 9ms/step - loss: 0.0235 - acc
uracy: 0.9925 - val_loss: 0.0329 - val_accuracy: 0.9913
Epoch 5/10
1688/1688 [==============================] - 16s 10ms/step - loss: 0.0175 - ac
curacy: 0.9943 - val_loss: 0.0323 - val_accuracy: 0.9918
Epoch 6/10
1688/1688 [==============================] - 16s 9ms/step - loss: 0.0126 - acc
uracy: 0.9956 - val_loss: 0.0321 - val_accuracy: 0.9920
Epoch 7/10
1688/1688 [==============================] - 16s 9ms/step - loss: 0.0095 - acc
uracy: 0.9969 - val_loss: 0.0349 - val_accuracy: 0.9908
Epoch 8/10
1688/1688 [==============================] - 16s 9ms/step - loss: 0.0068 - acc
uracy: 0.9978 - val_loss: 0.0533 - val_accuracy: 0.9890
Epoch 9/10
1688/1688 [==============================] - 16s 9ms/step - loss: 0.0042 - acc
uracy: 0.9987 - val_loss: 0.0346 - val_accuracy: 0.9928
Epoch 10/10
1688/1688 [==============================] - 16s 9ms/step - loss: 0.0038 - acc
```

```
uracy: 0.9988 - val loss: 0.0346 - val accuracy: 0.9928
```

In [113…

```
score = cnn_add.evaluate(testX, testY, verbose=0)
print(score)
```

```
[0.027527863159775734, 0.9915000200271606]
```

In [114…

```
cnn_low = create_cnn_add(learning_rate=0.001)
```

In [115…

```
cnn_low.fit(trainX, trainY, batch_size=32, epochs=10, validation_split=0.1)
```

```
Epoch 1/10
1688/1688 [==============================] - 16s 9ms/step - loss: 0.2822 - acc
uracy: 0.9162 - val_loss: 0.1076 - val_accuracy: 0.9730
Epoch 2/10
1688/1688 [==============================] - 15s 9ms/step - loss: 0.1045 - acc
uracy: 0.9690 - val_loss: 0.0691 - val_accuracy: 0.9822
Epoch 3/10
1688/1688 [==============================] - 15s 9ms/step - loss: 0.0744 - acc
uracy: 0.9771 - val_loss: 0.0564 - val_accuracy: 0.9847
Epoch 4/10
1688/1688 [==============================] - 16s 9ms/step - loss: 0.0613 - acc
uracy: 0.9812 - val_loss: 0.0529 - val_accuracy: 0.9863
Epoch 5/10
1688/1688 [==============================] - 15s 9ms/step - loss: 0.0522 - acc
uracy: 0.9846 - val_loss: 0.0483 - val_accuracy: 0.9872
Epoch 6/10
1688/1688 [==============================] - 15s 9ms/step - loss: 0.0459 - acc
uracy: 0.9862 - val_loss: 0.0535 - val_accuracy: 0.9848
Epoch 7/10
1688/1688 [==============================] - 15s 9ms/step - loss: 0.0410 - acc
uracy: 0.9876 - val_loss: 0.0470 - val_accuracy: 0.9875
Epoch 8/10
1688/1688 [==============================] - 15s 9ms/step - loss: 0.0374 - acc
uracy: 0.9887 - val_loss: 0.0506 - val_accuracy: 0.9865
Epoch 9/10
1688/1688 [==============================] - 15s 9ms/step - loss: 0.0340 - acc
uracy: 0.9896 - val_loss: 0.0419 - val_accuracy: 0.9877
Epoch 10/10
1688/1688 [==============================] - 15s 9ms/step - loss: 0.0310 - acc
uracy: 0.9904 - val_loss: 0.0450 - val_accuracy: 0.9883
```

Out[115…

```
<keras.callbacks.History at 0x13ab1291250>
```

In [116…

```
score = cnn_low.evaluate(testX, testY, verbose=0)
print(score)
```

```
[0.03898925334215164, 0.9871000051498413]
```

In [117…

```
cnn_high = create_cnn_add(learning_rate=0.1)
```

In [118…

```
cnn_high.fit(trainX, trainY, batch_size=32, epochs=10, validation_split=0.1)
```

```
Epoch 1/10
```

```
1688/1688 [==============================] - 17s 10ms/step - loss: 0.3498 - ac
curacy: 0.9074 - val_loss: 0.3094 - val_accuracy: 0.9237
Epoch 2/10
1688/1688 [==============================] - 15s 9ms/step - loss: 1.4909 - acc
uracy: 0.4775 - val_loss: 2.3095 - val_accuracy: 0.1000
Epoch 3/10
1688/1688 [==============================] - 15s 9ms/step - loss: 2.3087 - acc
uracy: 0.1057 - val_loss: 2.3098 - val_accuracy: 0.0915
Epoch 4/10
1688/1688 [==============================] - 15s 9ms/step - loss: 2.3078 - acc
uracy: 0.1077 - val_loss: 2.3054 - val_accuracy: 0.1050
Epoch 5/10
1688/1688 [==============================] - 15s 9ms/step - loss: 2.3079 - acc
uracy: 0.1038 - val_loss: 2.3069 - val_accuracy: 0.1050
Epoch 6/10
1688/1688 [==============================] - 15s 9ms/step - loss: 2.3081 - acc
uracy: 0.1053 - val_loss: 2.3055 - val_accuracy: 0.0978
Epoch 7/10
1688/1688 [==============================] - 15s 9ms/step - loss: 2.3084 - acc
uracy: 0.1046 - val_loss: 2.3153 - val_accuracy: 0.1050
Epoch 8/10
1688/1688 [==============================] - 15s 9ms/step - loss: 2.3080 - acc
uracy: 0.1027 - val_loss: 2.3044 - val_accuracy: 0.1045
Epoch 9/10
1688/1688 [==============================] - 15s 9ms/step - loss: 2.3087 - acc
uracy: 0.1044 - val_loss: 2.3080 - val_accuracy: 0.1113
Epoch 10/10
1688/1688 [==============================] - 15s 9ms/step - loss: 2.3086 - acc
uracy: 0.1035 - val_loss: 2.3160 - val_accuracy: 0.1050
```

Out[118…

```
<keras.callbacks.History at 0x13ab078ad30>
```

In [120…

```
score = cnn_high.evaluate(testX, testY, verbose=0)
print(score)
```

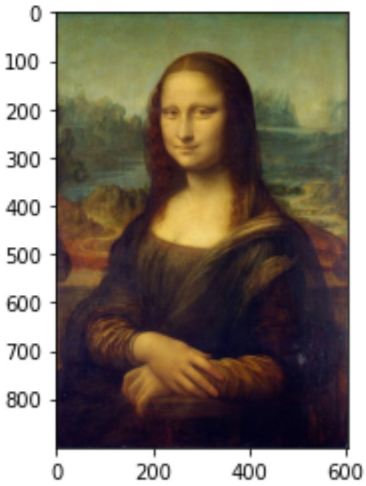
```
[2.31148624420166, 0.11349999904632568]
```

In [2]:
```
import numpy as np
from PIL import Image
import pandas as pd
import requests
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestRegressor
```

In [3]:
```
img = Image.open("C:\\Users\\sidar\\Desktop\\MLISA.jpg")
```

In [4]:
```
plt.imshow(img)
```

Out[4]:
```
<matplotlib.image.AxesImage at 0x16eae8b8310>
```



In [5]:
```
mLisa_array = np.array(img)
```

In [6]:
```
def sample(height, width, num=5000):
    coordinates = []
    for i in range(num):
        x = np.random.randint(low=0, high=height, size=(1,))
        y = np.random.randint(low=0, high=width, size=(1,))
        coordinates.append([x[0], y[0]])
    return coordinates
```

In [7]:
```
coordinates = sample(height=900, width=604)
```

In [8]:
```
mLisa_array.shape
```

Out[8]:
```
(900, 604, 3)
```

In [9]:
```
def regress(coordinates):
    pixels = []
    for coord in coordinates:
        rgb = mLisa_array[coord[0], coord[1], :]
        pixels.append(rgb)
    return pixels
```

In [10]:
```
pixels = regress(coordinates)
```

In [11]:
```
pixels = np.array(pixels)
```

In [12]:
```
rfr = RandomForestRegressor(n_estimators=10)
rfr.fit(X=coordinates, y=pixels)
```

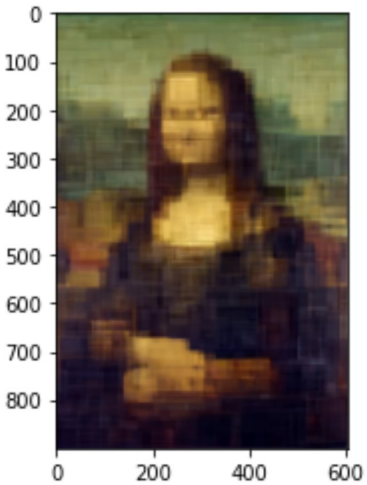
Out[12]:
```
RandomForestRegressor(n_estimators=10)
```

In [13]:
```
def approximate(classifier, image_array):
    predicted = []
    for i in range(0, image_array.shape[0], 1):
        row_predictions = []
        for j in range(0, image_array.shape[1], 1):
            prediction_coordinate = [[i, j]]
            rgb_prediction = classifier.predict(prediction_coordinate)
            row_predictions.append(rgb_prediction[0])
        predicted.append(row_predictions)
    predicted = np.asarray(predicted, dtype=int)
    return predicted
```

In [67]:
```
predicted_array = approximate(rfr, mLisa_array)
```

In [68]:
```
plt.imshow(predicted_array)
```

Out[68]:
```
<matplotlib.image.AxesImage at 0x18b33d80940>
```



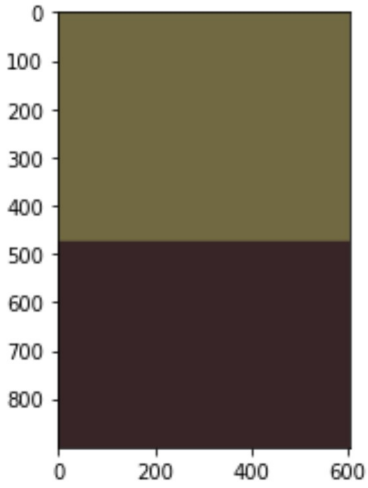
Varying Depth of one Tree

In [94]:
```
rfr_d1 = RandomForestRegressor(n_estimators=1, max_depth=1)
rfr_d1.fit(coordinates, pixels)
```

Out[94]:
```
RandomForestRegressor(max_depth=1, n_estimators=1)
```

In [95]:
```
predicted_array = approximate(rfr_d1, mLisa_array)
plt.imshow(predicted_array)
```

Out[95]:
```
<matplotlib.image.AxesImage at 0x18b29792a30>
```



In [96]:
```
rfr_d2 = RandomForestRegressor(n_estimators=1, max_depth=2)
rfr_d2.fit(coordinates, pixels)
```

Out[96]:
```
RandomForestRegressor(max_depth=2, n_estimators=1)
```

In [97]:
```
predicted_array = approximate(rfr_d2, mLisa_array)
plt.imshow(predicted_array)
```

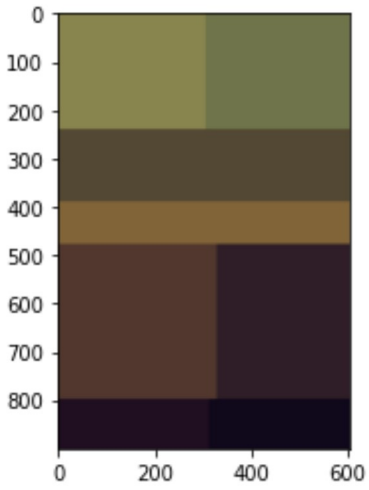
Out[97]:
```
<matplotlib.image.AxesImage at 0x18b2dc807c0>
```

In [98]:
```
rfr_d3 = RandomForestRegressor(n_estimators=1, max_depth=3)
rfr_d3.fit(coordinates, pixels)
```

Out[98]:
```
RandomForestRegressor(max_depth=3, n_estimators=1)
```

In [99]:
```
predicted_array = approximate(rfr_d3, mLisa_array)
plt.imshow(predicted_array)
```

Out[99]:
```
<matplotlib.image.AxesImage at 0x18b372145b0>
```

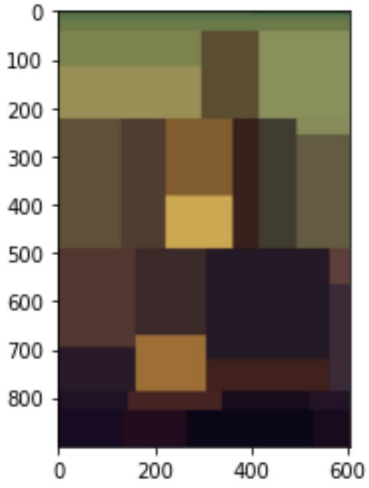


In [100…
```
rfr_d5 = RandomForestRegressor(n_estimators=1, max_depth=5)
rfr_d5.fit(coordinates, pixels)
```

Out[100…
```
RandomForestRegressor(max_depth=5, n_estimators=1)
```

In [101…
```
predicted_array = approximate(rfr_d5, mLisa_array)
plt.imshow(predicted_array)
```

Out[101…
```
<matplotlib.image.AxesImage at 0x18b33693a60>
```

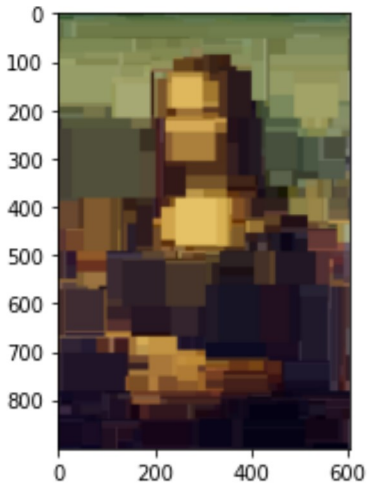


In [102…
```
rfr_d10 = RandomForestRegressor(n_estimators=1, max_depth=10)
rfr_d10.fit(coordinates, pixels)
```

Out[102…
```
RandomForestRegressor(max_depth=10, n_estimators=1)
```

In [103…
```
predicted_array = approximate(rfr_d10, mLisa_array)
plt.imshow(predicted_array)
```

Out[103…
```
<matplotlib.image.AxesImage at 0x18b33669d00>
```

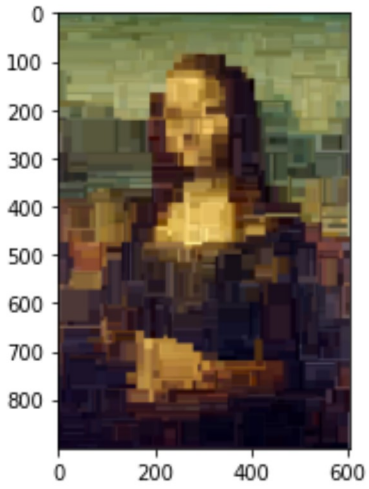


In [104…
```
rfr_d15 = RandomForestRegressor(n_estimators=1, max_depth=15)
rfr_d15.fit(coordinates, pixels)
```

Out[104…
```
RandomForestRegressor(max_depth=15, n_estimators=1)
```

In [105…
```
predicted_array = approximate(rfr_d15, mLisa_array)
plt.imshow(predicted_array)
```

Out[105…
```
<matplotlib.image.AxesImage at 0x18b2c1e9520>
```



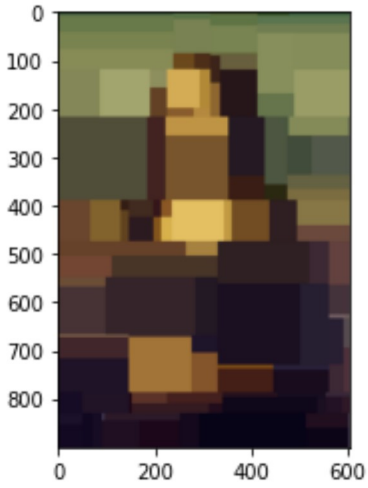
Varying Number of Trees

In [81]:
```
rfr_n1 = RandomForestRegressor(n_estimators=1, max_depth=7)
rfr_n1.fit(coordinates, pixels)
```

Out[81]:
```
RandomForestRegressor(max_depth=7, n_estimators=1)
```

In [82]:
```
predicted_array = approximate(rfr_n1, mLisa_array)
plt.imshow(predicted_array)
```

Out[82]:
```
<matplotlib.image.AxesImage at 0x18b34ec33a0>
```



In [83]:
```
rfr_n3 = RandomForestRegressor(n_estimators=3, max_depth=7)
rfr_n3.fit(coordinates, pixels)
```

Out[83]:
```
RandomForestRegressor(max_depth=7, n_estimators=3)
```

In [84]:
```
predicted_array = approximate(rfr_n3, mLisa_array)
plt.imshow(predicted_array)
```

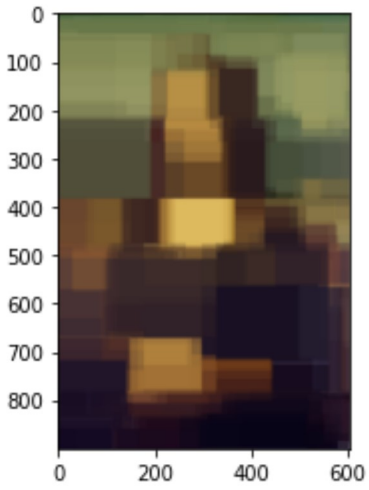
Out[84]:
```
<matplotlib.image.AxesImage at 0x18b2d724910>
```

In [85]:
```
rfr_n5 = RandomForestRegressor(n_estimators=5, max_depth=7)
rfr_n5.fit(coordinates, pixels)
```

Out[85]:
```
RandomForestRegressor(max_depth=7, n_estimators=5)
```

In [86]:
```
predicted_array = approximate(rfr_n5, mLisa_array)
plt.imshow(predicted_array)
```

Out[86]:
```
<matplotlib.image.AxesImage at 0x18b2f54dd90>
```

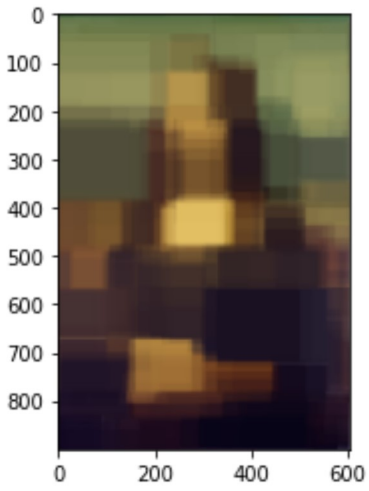


In [87]:
```
rfr_n10 = RandomForestRegressor(n_estimators=10, max_depth=7)
rfr_n10.fit(coordinates, pixels)
```

Out[87]:
```
RandomForestRegressor(max_depth=7, n_estimators=10)
```

In [88]:
```
predicted_array = approximate(rfr_n10, mLisa_array)
plt.imshow(predicted_array)
```

Out[88]:
```
<matplotlib.image.AxesImage at 0x18b35e761c0>
```

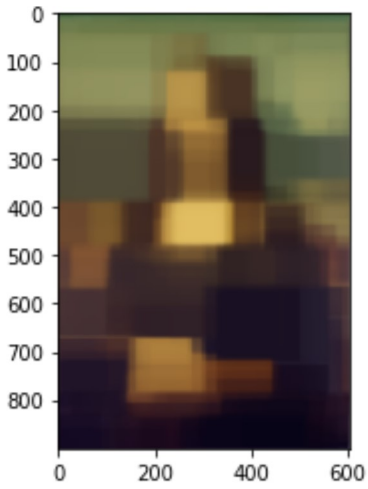


In [89]:
```
rfr_n100 = RandomForestRegressor(n_estimators=100, max_depth=7)
rfr_n100.fit(coordinates, pixels)
```

Out[89]:
```
RandomForestRegressor(max_depth=7)
```

In [90]:
```
predicted_array = approximate(rfr_n100, mLisa_array)
plt.imshow(predicted_array)
```

Out[90]:
```
<matplotlib.image.AxesImage at 0x18b2d589c10>
```



k-NN

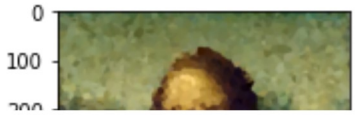
In [91]:
```
from sklearn.neighbors import KNeighborsRegressor
```

In [92]:
```
knn = KNeighborsRegressor(n_neighbors=1)
knn.fit(coordinates, pixels)
```

Out[92]:
```
KNeighborsRegressor(n_neighbors=1)
```

In [93]:
```
predicted_array = approximate(knn, mLisa_array)
plt.imshow(predicted_array)
```

Out[93]:
```
<matplotlib.image.AxesImage at 0x18b3121b520>
```



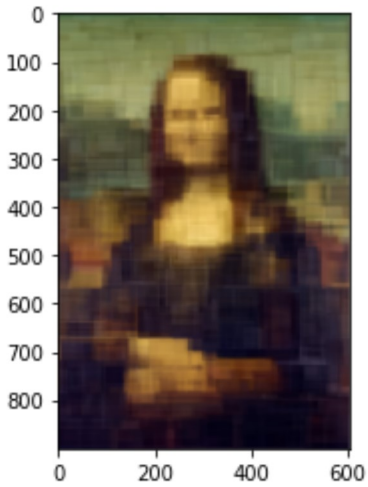
Pruning

In [108…
```
rfr_split = RandomForestRegressor(n_estimators=10, min_samples_split=5)
rfr_split.fit(coordinates, pixels)
```

Out[108…
```
RandomForestRegressor(min_samples_split=5, n_estimators=10)
```

In [109…
```
predicted_array = approximate(rfr_split, mLisa_array)
plt.imshow(predicted_array)
```

Out[109…
```
<matplotlib.image.AxesImage at 0x18b2fc874f0>
```

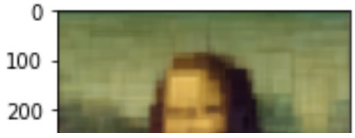


In [110…
```
rfr_leaf = RandomForestRegressor(n_estimators=10, min_samples_leaf=2)
rfr_leaf.fit(X=coordinates, y=pixels)
```

Out[110…
```
RandomForestRegressor(min_samples_leaf=2, n_estimators=10)
```

In [111…
```
predicted_array = approximate(rfr_leaf, mLisa_array)
plt.imshow(predicted_array)
```

Out[111…
```
<matplotlib.image.AxesImage at 0x18b2bef7d90>
```


0
100
200