Programming Exercises

0.1 Convolutional Neural Networks

0.1.1 (a) Loading the Dataset

We download the train and the test datasets as shown.

```
import keras
keras.__version__
```

```
'2.7.0'
```

```
from keras.datasets import mnist
import matplotlib.pyplot as plt
import numpy as np

(train_X, train_Y), (test_X, test_Y) = mnist.load_data()
```

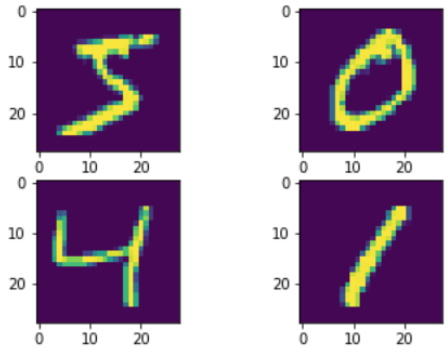
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
```

```
print(train_X.shape, train_Y.shape)
```

```
(60000, 28, 28) (60000,)
```

```
plt.figure()

#subplot(r,c) provide the no. of rows and columns
f, axarr = plt.subplots(2,2)

axarr[0][0].imshow(train_X[0])
axarr[0][1].imshow(train_X[1])
axarr[1][0].imshow(train_X[2])
axarr[1][1].imshow(train_X[3])
```

```
<matplotlib.image.AxesImage at 0x7f6a2a301090>
<Figure size 432x288 with 0 Axes>
```



As we can see, the training set consists of 60000 unique images, each in the shape of a 28 x 28 matrix. The test set consists of 10000 images.

0.1.2 (b) Preprocessing

We first initialise the number of classes to be equal to 10 (for each digit). We then normalise the images by diving them by 255, to get values that lie between 0 and 1. Then we convert the training and test datasets into 28 x 28 x 1 sized matrices. We finally convert class vectors to binary class matrices using $to_categorical which we import from keras.utils$.

```
from tensorflow.keras.utils import to_categorical

# Model / data

num_classes = 10
input_shape = (28, 28, 1)


# Scale images to the [0, 1] range

train_X = train_X.astype("float32") / 255
test_X = test_X.astype("float32") / 255

# Convert image shape to (28, 28, 1)
train_X = np.expand_dims(train_X, -1)
test_X = np.expand_dims(test_X, -1)

print("train_X shape:", train_X.shape)
print("Training samples:", train_X.shape[0])
print("Test samples:", test_Y.shape[0])


# convert class vectors to binary class matrices
train_Y = to_categorical(train_Y, num_classes)
test_Y = to_categorical(test_Y, num_classes)
```

```
train_X shape: (60000, 28, 28, 1)
Training samples: 60000
Test samples: 10000
```

0.1.3 (c) Implementation

We define the CNN as provided in the assignment description. We print model.layers to see if the model is working.

```
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from tensorflow.keras.optimizers import SGD

def create_cnn( ):
  # define using Sequential
  model = Sequential()
  # Convolution layer
  model.add(
      Conv2D(32, (3, 3),
      activation= 'relu',
      kernel_initializer='he_uniform',
      input_shape=(28, 28, 1))
      )
  # Maxpooling layer
  model.add(MaxPooling2D((2, 2)))
  # Flatten output
  model.add(Flatten())
  # Dense layer of 100 neurons
  model.add(
      Dense (100,
      activation= 'relu',
      kernel_initializer='he_uniform')
      )
  model.add(Dense(10, activation='softmax')) # initialize optimizer
  opt = SGD(learning_rate=0.01, momentum=0.9)
  # compile model
  model.compile(
      optimizer=opt,
      loss= 'categorical_crossentropy',
      metrics =['accuracy']
      )
  print(model.layers)

  return model
```

```
model = create_cnn()
```

```
[<keras.layers.convolutional.Conv2D object at 0x7f8892210910>, <keras.layers.pooling.MaxPooling2D object at 0x7f8891964450>, <keras.layers.core.flatten.Flatten object at 0x7f888d8f1190>,
```

0.1.4 (d) Training and Evaluating CNN

We train the CNN for 10 epochs, using batch size 32 and a validation split of 0.1.

After training the CNN, we get the following results:

Training loss: 0.0020 Training accuracy: 0.9998 Validation loss: 0.0522 Validation accuracy: 0.9895

```
model.fit(train_X, train_Y, batch_size=32, epochs=10, validation_split =0.1)
score = model.evaluate(test_X, test_Y, verbose=0)
```

```
Epoch 1/10
1688/1688 [==============================] - 35s 20ms/step - loss: 0.1693 - accuracy: 0.9486 - val_loss: 0.0697 - val_accuracy: 0.9808
Epoch 2/10
1688/1688 [==============================] - 34s 20ms/step - loss: 0.0582 - accuracy: 0.9825 - val_loss: 0.0570 - val_accuracy: 0.9837
Epoch 3/10
1688/1688 [==============================] - 34s 20ms/step - loss: 0.0373 - accuracy: 0.9890 - val_loss: 0.0455 - val_accuracy: 0.9885
Epoch 4/10
1688/1688 [==============================] - 34s 20ms/step - loss: 0.0244 - accuracy: 0.9925 - val_loss: 0.0546 - val_accuracy: 0.9865
Epoch 5/10
1688/1688 [==============================] - 34s 20ms/step - loss: 0.0171 - accuracy: 0.9949 - val_loss: 0.0418 - val_accuracy: 0.9895
Epoch 6/10
1688/1688 [==============================] - 34s 20ms/step - loss: 0.0115 - accuracy: 0.9969 - val_loss: 0.0466 - val_accuracy: 0.9887
Epoch 7/10
1688/1688 [==============================] - 34s 20ms/step - loss: 0.0076 - accuracy: 0.9983 - val_loss: 0.0519 - val_accuracy: 0.9875
Epoch 8/10
1688/1688 [==============================] - 34s 20ms/step - loss: 0.0048 - accuracy: 0.9990 - val_loss: 0.0478 - val_accuracy: 0.9898
Epoch 9/10
1688/1688 [==============================] - 34s 20ms/step - loss: 0.0034 - accuracy: 0.9995 - val_loss: 0.0509 - val_accuracy: 0.9887
Epoch 10/10
1688/1688 [==============================] - 34s 20ms/step - loss: 0.0020 - accuracy: 0.9998 - val_loss: 0.0522 - val_accuracy: 0.9895
```

0.1.5 (e) Experimentation

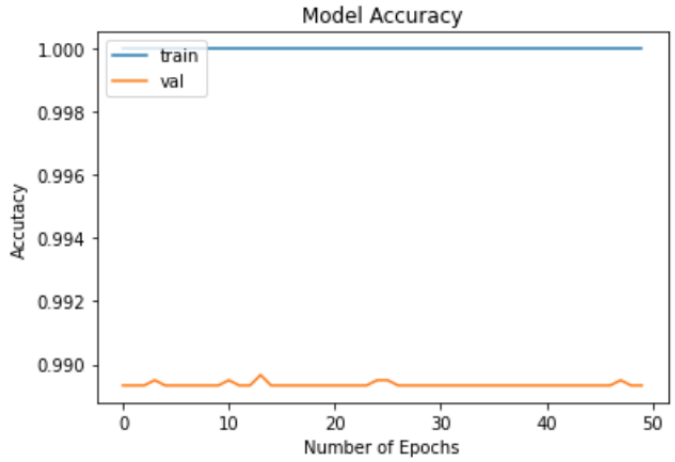
i. We run the CNN for 50 epochs as shown below. The final training accuracy we get on the 50th epoch is 1. The final validation loss accuracy we get on the 50th epoch are 0.0762 and 0.9893 respectively.

```
epoch_history = model.fit(train_X, train_Y, batch_size=32, epochs=50, validation_split =0.1)

# print validation and training accuracy over epochs
print(epoch_history.history['accuracy'])
print(epoch_history.history['val_accuracy'])
```

```
1688/1688 [==============================] - 34s 20ms/step - loss: 5.4706e-05 - accuracy: 1.0000 - val_loss: 0.0734 - val_accuracy: 0.9893
Epoch 22/50
1688/1688 [==============================] - 34s 20ms/step - loss: 5.3810e-05 - accuracy: 1.0000 - val_loss: 0.0736 - val_accuracy: 0.9893
Epoch 23/50
1688/1688 [==============================] - 34s 20ms/step - loss: 5.2918e-05 - accuracy: 1.0000 - val_loss: 0.0738 - val_accuracy: 0.9893
Epoch 24/50
1688/1688 [==============================] - 34s 20ms/step - loss: 5.2007e-05 - accuracy: 1.0000 - val_loss: 0.0735 - val_accuracy: 0.9893
Epoch 25/50
1688/1688 [==============================] - 34s 20ms/step - loss: 5.1206e-05 - accuracy: 1.0000 - val_loss: 0.0738 - val_accuracy: 0.9895
Epoch 26/50
1688/1688 [==============================] - 34s 20ms/step - loss: 5.0237e-05 - accuracy: 1.0000 - val_loss: 0.0738 - val_accuracy: 0.9895
Epoch 27/50
1688/1688 [==============================] - 34s 20ms/step - loss: 4.9446e-05 - accuracy: 1.0000 - val_loss: 0.0742 - val_accuracy: 0.9893
Epoch 28/50
1688/1688 [==============================] - 34s 20ms/step - loss: 4.8693e-05 - accuracy: 1.0000 - val_loss: 0.0742 - val_accuracy: 0.9893
Epoch 29/50
1688/1688 [==============================] - 34s 20ms/step - loss: 4.8042e-05 - accuracy: 1.0000 - val_loss: 0.0744 - val_accuracy: 0.9893
Epoch 30/50
1688/1688 [==============================] - 35s 21ms/step - loss: 4.7228e-05 - accuracy: 1.0000 - val_loss: 0.0746 - val_accuracy: 0.9893
Epoch 31/50
1688/1688 [==============================] - 35s 21ms/step - loss: 4.6562e-05 - accuracy: 1.0000 - val_loss: 0.0745 - val_accuracy: 0.9893
Epoch 32/50
1688/1688 [==============================] - 35s 21ms/step - loss: 4.5807e-05 - accuracy: 1.0000 - val_loss: 0.0750 - val_accuracy: 0.9893
Epoch 33/50
1688/1688 [==============================] - 34s 20ms/step - loss: 4.5195e-05 - accuracy: 1.0000 - val_loss: 0.0747 - val_accuracy: 0.9893
Epoch 34/50
1688/1688 [==============================] - 34s 20ms/step - loss: 4.4556e-05 - accuracy: 1.0000 - val_loss: 0.0747 - val_accuracy: 0.9893
Epoch 35/50
1688/1688 [==============================] - 34s 20ms/step - loss: 4.3961e-05 - accuracy: 1.0000 - val_loss: 0.0747 - val_accuracy: 0.9893
Epoch 36/50
1688/1688 [==============================] - 34s 20ms/step - loss: 4.3281e-05 - accuracy: 1.0000 - val_loss: 0.0755 - val_accuracy: 0.9893
Epoch 37/50
1688/1688 [==============================] - 34s 20ms/step - loss: 4.2668e-05 - accuracy: 1.0000 - val_loss: 0.0752 - val_accuracy: 0.9893
Epoch 38/50
1688/1688 [==============================] - 34s 20ms/step - loss: 4.2090e-05 - accuracy: 1.0000 - val_loss: 0.0752 - val_accuracy: 0.9893
Epoch 39/50
1688/1688 [==============================] - 34s 20ms/step - loss: 4.1457e-05 - accuracy: 1.0000 - val_loss: 0.0754 - val_accuracy: 0.9893
Epoch 40/50
1688/1688 [==============================] - 34s 20ms/step - loss: 4.0890e-05 - accuracy: 1.0000 - val_loss: 0.0751 - val_accuracy: 0.9893
Epoch 41/50
1688/1688 [==============================] - 34s 20ms/step - loss: 4.0352e-05 - accuracy: 1.0000 - val_loss: 0.0755 - val_accuracy: 0.9893
Epoch 42/50
1688/1688 [==============================] - 34s 20ms/step - loss: 3.9910e-05 - accuracy: 1.0000 - val_loss: 0.0757 - val_accuracy: 0.9893
```

We then plot the validation and training accuracies as show. We can see that while there is slight improvement for both sets, it is not very substantial over 50 epochs. We can also note that the training accuracy ends at 1 and there are little peaks of improvement in the validation set accuracy.

```
from matplotlib import pyplot as plt

plt.plot(epoch_history.history['accuracy'])
plt.plot(epoch_history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accutacy')
plt.xlabel('Number of Epochs')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```



ii. We implement a new CNN with an added Dropout layer as shown below. After running it for 50 epochs, the final training loss and accuracy we get are 0.0033 and 0.9989 respectively. The final validation loss and accuracy we get are 0.0517 and 0.9925 respectively.

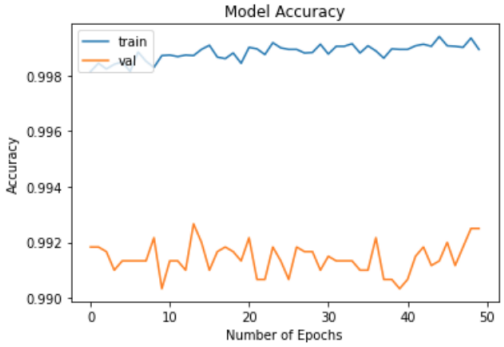
```
from keras.layers import Dropout

def create_cnn_with_dropout( ):
  # define using Sequential
  model = Sequential()
  # Convolution layer
  model.add(
      Conv2D(32, (3, 3),
      activation= 'relu',
      kernel_initializer='he_uniform',
      input_shape=(28, 28, 1))
      )
  # Maxpooling layer
  model.add(MaxPooling2D((2, 2)))
  # Flatten output
  model.add(Flatten())
  # Dropout Layer
  model.add(Dropout(0.5))
  # Dense layer of 100 neurons
  model.add(
      Dense (100,
      activation= 'relu',
      kernel_initializer='he_uniform')
      )
  model.add(Dense(10, activation='softmax')) # initialize optimizer
  opt = SGD(lr=0.01, momentum=0.9)
  # compile model
  model.compile(
      optimizer=opt,
      loss= 'categorical_crossentropy',
      metrics =['accuracy']
      )

  #print(model.layers)

  return model
```

```
new_model = create_cnn_with_dropout()

new_model.fit(train_X, train_Y, batch_size=32, epochs=50, validation_split =0.1)
score = new_model.evaluate(test_X, test_Y, verbose=0)
dropout_history = new_model.fit(train_X, train_Y, batch_size=32, epochs=50, validation_split =0.1)
```

```
Epoch 20/50
1688/1688 [==============================] - 38s 22ms/step - loss: 0.0036 - accuracy: 0.9986 - val_loss: 0.0457 - val_accuracy: 0.9907
Epoch 21/50
1688/1688 [==============================] - 38s 23ms/step - loss: 0.0030 - accuracy: 0.9992 - val_loss: 0.0459 - val_accuracy: 0.9910
Epoch 22/50
1688/1688 [==============================] - 38s 23ms/step - loss: 0.0039 - accuracy: 0.9987 - val_loss: 0.0444 - val_accuracy: 0.9920
Epoch 23/50
1688/1688 [==============================] - 38s 22ms/step - loss: 0.0037 - accuracy: 0.9987 - val_loss: 0.0480 - val_accuracy: 0.9920
Epoch 24/50
1688/1688 [==============================] - 38s 23ms/step - loss: 0.0028 - accuracy: 0.9990 - val_loss: 0.0522 - val_accuracy: 0.9922
Epoch 25/50
1688/1688 [==============================] - 39s 23ms/step - loss: 0.0032 - accuracy: 0.9989 - val_loss: 0.0489 - val_accuracy: 0.9920
Epoch 26/50
1688/1688 [==============================] - 39s 23ms/step - loss: 0.0030 - accuracy: 0.9990 - val_loss: 0.0494 - val_accuracy: 0.9925
Epoch 27/50
1688/1688 [==============================] - 39s 23ms/step - loss: 0.0030 - accuracy: 0.9991 - val_loss: 0.0507 - val_accuracy: 0.9912
Epoch 28/50
1688/1688 [==============================] - 40s 24ms/step - loss: 0.0026 - accuracy: 0.9990 - val_loss: 0.0456 - val_accuracy: 0.9923
Epoch 29/50
1688/1688 [==============================] - 40s 24ms/step - loss: 0.0031 - accuracy: 0.9990 - val_loss: 0.0482 - val_accuracy: 0.9922
Epoch 30/50
1688/1688 [==============================] - 39s 23ms/step - loss: 0.0032 - accuracy: 0.9991 - val_loss: 0.0495 - val_accuracy: 0.9913
Epoch 31/50
1688/1688 [==============================] - 38s 23ms/step - loss: 0.0031 - accuracy: 0.9988 - val_loss: 0.0483 - val_accuracy: 0.9917
Epoch 32/50
1688/1688 [==============================] - 39s 23ms/step - loss: 0.0028 - accuracy: 0.9990 - val_loss: 0.0481 - val_accuracy: 0.9922
Epoch 33/50
1688/1688 [==============================] - 39s 23ms/step - loss: 0.0024 - accuracy: 0.9992 - val_loss: 0.0460 - val_accuracy: 0.9925
Epoch 34/50
1688/1688 [==============================] - 39s 23ms/step - loss: 0.0035 - accuracy: 0.9989 - val_loss: 0.0476 - val_accuracy: 0.9920
Epoch 35/50
1688/1688 [==============================] - 39s 23ms/step - loss: 0.0027 - accuracy: 0.9991 - val_loss: 0.0471 - val_accuracy: 0.9927
Epoch 36/50
1688/1688 [==============================] - 38s 23ms/step - loss: 0.0028 - accuracy: 0.9991 - val_loss: 0.0479 - val_accuracy: 0.9920
```

Plotting the history, we can see that the variance for both the training and validation accuracies are higher. While there are lots of peaks and pits, the validation accuracy tends to rise in the last 10 epochs significantly. Furthermore, the training accuracy is less than 1. This shows that the dropout layer successfully helped the model to reduce overfitting.

```
from matplotlib import pyplot as plt

plt.plot(dropout_history.history['accuracy'])
plt.plot(dropout_history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Number of Epochs')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```



iii. We add another convolution layer to the CNN, using 64 input filters. After training it for 10 epochs, we get a training loss of 0.0455 and a training accuracy of 0.9855. Our validation loss is 0.0267 and the validation accuracy is 0.9915.

```
def create_cnn_with_layers( ):
  # define using Sequential
  model = Sequential()
  # Convolution layer 1
  model.add(
      Conv2D(64, (3, 3),
      activation= 'relu',
      kernel_initializer='he_uniform',
      input_shape=(28, 28, 1))
      )
  # Maxpooling layer 1
  model.add(MaxPooling2D((2, 2)))
  # Convolution Layer 2
  model.add(
      Conv2D(32, (3, 3),
      activation= 'relu',
      kernel_initializer='he_uniform',
      input_shape=(28, 28, 1))
      )
  # Maxpooling layer 2
  model.add(MaxPooling2D((2, 2)))
  # Flatten output
  model.add(Flatten())
  # Dropout Layer
  model.add(Dropout(0.5))
  # Dense layer of 100 neurons
  model.add(
      Dense (100,
      activation= 'relu',
      kernel_initializer='he_uniform')
      )
  model.add(Dense(10, activation='softmax')) # initialize optimizer
  opt = SGD(lr=0.01, momentum=0.9)
  # compile model
  model.compile(
      optimizer=opt,
      loss= 'categorical_crossentropy',
      metrics =['accuracy']
      )

  #print(model.layers)

  return model
```

```
# Training the new CNN

mult_layered_model = create_cnn_with_layers()

new_model.fit(train_X, train_Y, batch_size=32, epochs=10, validation_split =0.1)
score = mult_layered_model.evaluate(test_X, test_Y, verbose=0)
dropout_history = mult_layered_model.fit(train_X, train_Y, batch_size=32, epochs=10, validation_split =0.1)
```

```
Epoch 1/10
   3/1688 [..............................] - ETA: 48s - loss: 2.7154e-04 - accuracy: 1.0000/usr/local/lib/python3.7/dist-packages/keras/optimizer
  super(SGD, self).__init__(name, **kwargs)
1688/1688 [==============================] - 38s 22ms/step - loss: 0.0019 - accuracy: 0.9994 - val_loss: 0.0539 - val_accuracy: 0.9918
Epoch 2/10
1688/1688 [==============================] - 37s 22ms/step - loss: 0.0018 - accuracy: 0.9995 - val_loss: 0.0534 - val_accuracy: 0.9915
Epoch 3/10
1688/1688 [==============================] - 37s 22ms/step - loss: 0.0025 - accuracy: 0.9992 - val_loss: 0.0578 - val_accuracy: 0.9908
Epoch 4/10
1688/1688 [==============================] - 37s 22ms/step - loss: 0.0026 - accuracy: 0.9992 - val_loss: 0.0510 - val_accuracy: 0.9917
Epoch 5/10
1688/1688 [==============================] - 38s 22ms/step - loss: 0.0024 - accuracy: 0.9991 - val_loss: 0.0501 - val_accuracy: 0.9920
Epoch 6/10
1688/1688 [==============================] - 38s 22ms/step - loss: 0.0027 - accuracy: 0.9990 - val_loss: 0.0488 - val_accuracy: 0.9917
Epoch 7/10
1688/1688 [==============================] - 37s 22ms/step - loss: 0.0023 - accuracy: 0.9994 - val_loss: 0.0488 - val_accuracy: 0.9922
Epoch 8/10
1688/1688 [==============================] - 37s 22ms/step - loss: 0.0020 - accuracy: 0.9993 - val_loss: 0.0522 - val_accuracy: 0.9923
Epoch 9/10
1688/1688 [==============================] - 38s 22ms/step - loss: 0.0024 - accuracy: 0.9991 - val_loss: 0.0495 - val_accuracy: 0.9922
Epoch 10/10
1688/1688 [==============================] - 37s 22ms/step - loss: 0.0017 - accuracy: 0.9995 - val_loss: 0.0528 - val_accuracy: 0.9922
Epoch 1/10
1688/1688 [==============================] - 68s 40ms/step - loss: 0.2731 - accuracy: 0.9137 - val_loss: 0.0553 - val_accuracy: 0.9835
Epoch 2/10
1688/1688 [==============================] - 66s 39ms/step - loss: 0.1217 - accuracy: 0.9614 - val_loss: 0.0472 - val_accuracy: 0.9855
Epoch 3/10
1688/1688 [==============================] - 66s 39ms/step - loss: 0.0947 - accuracy: 0.9701 - val_loss: 0.0359 - val_accuracy: 0.9895
Epoch 4/10
1688/1688 [==============================] - 67s 40ms/step - loss: 0.0817 - accuracy: 0.9742 - val_loss: 0.0381 - val_accuracy: 0.9882
Epoch 5/10
1688/1688 [==============================] - 67s 40ms/step - loss: 0.0692 - accuracy: 0.9779 - val_loss: 0.0323 - val_accuracy: 0.9903
Epoch 6/10
1688/1688 [==============================] - 67s 40ms/step - loss: 0.0642 - accuracy: 0.9791 - val_loss: 0.0316 - val_accuracy: 0.9903
Epoch 7/10
1688/1688 [==============================] - 68s 40ms/step - loss: 0.0579 - accuracy: 0.9818 - val_loss: 0.0329 - val_accuracy: 0.9905
Epoch 8/10
1688/1688 [==============================] - 69s 41ms/step - loss: 0.0527 - accuracy: 0.9828 - val_loss: 0.0294 - val_accuracy: 0.9908
Epoch 9/10
1688/1688 [==============================] - 68s 41ms/step - loss: 0.0501 - accuracy: 0.9839 - val_loss: 0.0273 - val_accuracy: 0.9918
Epoch 10/10
1688/1688 [==============================] - 68s 40ms/step - loss: 0.0455 - accuracy: 0.9855 - val_loss: 0.0267 - val_accuracy: 0.9915
```

iv. We then train the model with the 2 convolution and maxpooling layers, and a dropout layer and experiment with a learning rate of 0.001. After training the model for 10 epochs, we get a training loss of 0.0568 and a training accuracy of 0.9823. Our validation loss is 0.0329 and the validation accuracy is 0.9910.

```
def final_create_cnn( ):
  # define using Sequential
  model = Sequential()
  # Convolution layer 1
  model.add(
      Conv2D(64, (3, 3),
      activation= 'relu',
      kernel_initializer='he_uniform',
      input_shape=(28, 28, 1))
      )
  # Maxpooling layer 1
  model.add(MaxPooling2D((2, 2)))
  # Convolution Layer 2
  model.add(
      Conv2D(32, (3, 3),
      activation= 'relu',
      kernel_initializer='he_uniform',
      input_shape=(28, 28, 1))
      )
  # Maxpooling layer 2
  model.add(MaxPooling2D((2, 2)))
  # Flatten output
  model.add(Flatten())
  # Dropout Layer
  model.add(Dropout(0.5))
  # Dense layer of 100 neurons
  model.add(
      Dense (100,
      activation= 'relu',
      kernel_initializer='he_uniform')
      )
  model.add(Dense(10, activation='softmax')) # initialize optimizer
  opt = SGD(learning_rate=0.001, momentum=0.9)
  # compile model
  model.compile(
      optimizer=opt,
      loss= 'categorical_crossentropy',
      metrics =['accuracy']
      )


  return model
```

```
# Training the new CNN

model1 = final_create_cnn()

model1.fit(train_X, train_Y, batch_size=32, epochs=10, validation_split =0.1)
score = model1.evaluate(test_X, test_Y, verbose=0)
history1 = model1.fit(train_X, train_Y, batch_size=32, epochs=10, validation_split =0.1)
```

```
Epoch 1/10
1688/1688 [==============================] - 68s 40ms/step - loss: 0.4692 - accuracy: 0.8499 - val_loss: 0.1064 - val_accuracy: 0.9692
Epoch 2/10
1688/1688 [==============================] - 68s 40ms/step - loss: 0.1875 - accuracy: 0.9418 - val_loss: 0.0761 - val_accuracy: 0.9780
Epoch 3/10
1688/1688 [==============================] - 68s 40ms/step - loss: 0.1479 - accuracy: 0.9549 - val_loss: 0.0634 - val_accuracy: 0.9812
Epoch 4/10
1688/1688 [==============================] - 68s 40ms/step - loss: 0.1269 - accuracy: 0.9609 - val_loss: 0.0577 - val_accuracy: 0.9847
Epoch 5/10
1688/1688 [==============================] - 67s 40ms/step - loss: 0.1140 - accuracy: 0.9646 - val_loss: 0.0531 - val_accuracy: 0.9860
Epoch 6/10
1688/1688 [==============================] - 69s 41ms/step - loss: 0.1036 - accuracy: 0.9672 - val_loss: 0.0477 - val_accuracy: 0.9883
Epoch 7/10
1688/1688 [==============================] - 69s 41ms/step - loss: 0.0949 - accuracy: 0.9713 - val_loss: 0.0477 - val_accuracy: 0.9872
Epoch 8/10
1688/1688 [==============================] - 69s 41ms/step - loss: 0.0889 - accuracy: 0.9723 - val_loss: 0.0423 - val_accuracy: 0.9883
Epoch 9/10
1688/1688 [==============================] - 67s 40ms/step - loss: 0.0848 - accuracy: 0.9734 - val_loss: 0.0397 - val_accuracy: 0.9892
Epoch 10/10
1688/1688 [==============================] - 67s 40ms/step - loss: 0.0795 - accuracy: 0.9751 - val_loss: 0.0409 - val_accuracy: 0.9887
Epoch 1/10
1688/1688 [==============================] - 67s 40ms/step - loss: 0.0768 - accuracy: 0.9759 - val_loss: 0.0371 - val_accuracy: 0.9903
Epoch 2/10
1688/1688 [==============================] - 67s 40ms/step - loss: 0.0741 - accuracy: 0.9774 - val_loss: 0.0385 - val_accuracy: 0.9893
Epoch 3/10
1688/1688 [==============================] - 67s 40ms/step - loss: 0.0706 - accuracy: 0.9772 - val_loss: 0.0380 - val_accuracy: 0.9895
Epoch 4/10
1688/1688 [==============================] - 67s 40ms/step - loss: 0.0688 - accuracy: 0.9782 - val_loss: 0.0371 - val_accuracy: 0.9885
Epoch 5/10
1688/1688 [==============================] - 67s 40ms/step - loss: 0.0669 - accuracy: 0.9791 - val_loss: 0.0356 - val_accuracy: 0.9898
Epoch 6/10
1688/1688 [==============================] - 67s 40ms/step - loss: 0.0641 - accuracy: 0.9797 - val_loss: 0.0350 - val_accuracy: 0.9897
Epoch 7/10
1688/1688 [==============================] - 67s 40ms/step - loss: 0.0605 - accuracy: 0.9808 - val_loss: 0.0347 - val_accuracy: 0.9907
Epoch 8/10
1688/1688 [==============================] - 67s 40ms/step - loss: 0.0598 - accuracy: 0.9809 - val_loss: 0.0390 - val_accuracy: 0.9892
Epoch 9/10
1688/1688 [==============================] - 67s 40ms/step - loss: 0.0585 - accuracy: 0.9819 - val_loss: 0.0357 - val_accuracy: 0.9900
Epoch 10/10
1688/1688 [==============================] - 67s 40ms/step - loss: 0.0568 - accuracy: 0.9823 - val_loss: 0.0329 - val_accuracy: 0.9910
```

Training the model with a learning rate of 0.1 for 10 epochs, we get a training loss of 2.3085 and an accuracy of 0.1044. Our validation loss is 2.3070 and the validation accuracy is 0.1113. We can clearly see that the losses for this learning rate are really high, whereas the accuracies are really low. This shows that for image recognition, 0.1 is a very high learning rate which does not perform well. This is due to the step size being too large, therefore the model does not have enough space to adjust. It results in an unstable training with very low accuracy.

```
def final_create_cnn( ):
  # define using Sequential
  model = Sequential()
  # Convolution layer 1
  model.add(
      Conv2D(64, (3, 3),
      activation= 'relu',
      kernel_initializer='he_uniform',
      input_shape=(28, 28, 1))
      )
  # Maxpooling layer 1
  model.add(MaxPooling2D((2, 2)))
  # Convolution Layer 2
  model.add(
      Conv2D(32, (3, 3),
      activation= 'relu',
      kernel_initializer='he_uniform',
      input_shape=(28, 28, 1))
      )
  # Maxpooling layer 2
  model.add(MaxPooling2D((2, 2)))
  # Flatten output
  model.add(Flatten())
  # Dropout Layer
  model.add(Dropout(0.5))
  # Dense layer of 100 neurons
  model.add(
      Dense (100,
      activation= 'relu',
      kernel_initializer='he_uniform')
      )
  model.add(Dense(10, activation='softmax')) # initialize optimizer
  opt = SGD(learning_rate=0.1, momentum=0.9)
  # compile model
  model.compile(
      optimizer=opt,
      loss= 'categorical_crossentropy',
      metrics =['accuracy']
      )


  return model
```

```
# Training the new CNN

model2 = final_create_cnn()

model2.fit(train_X, train_Y, batch_size=32, epochs=10, validation_split =0.1)
score = model2.evaluate(test_X, test_Y, verbose=0)
history2 = model2.fit(train_X, train_Y, batch_size=32, epochs=10, validation_split =0.1)
```

```
Epoch 1/10
1688/1688 [==============================] - 66s 39ms/step - loss: 1.0745 - accuracy: 0.6936 - val_loss: 1.1743 - val_accuracy: 0.6415
Epoch 2/10
1688/1688 [==============================] - 66s 39ms/step - loss: 1.3197 - accuracy: 0.6063 - val_loss: 0.9009 - val_accuracy: 0.7183
Epoch 3/10
1688/1688 [==============================] - 65s 39ms/step - loss: 1.4983 - accuracy: 0.5166 - val_loss: 1.0165 - val_accuracy: 0.6652
Epoch 4/10
1688/1688 [==============================] - 66s 39ms/step - loss: 1.5875 - accuracy: 0.4621 - val_loss: 1.2858 - val_accuracy: 0.5953
Epoch 5/10
1688/1688 [==============================] - 66s 39ms/step - loss: 2.0825 - accuracy: 0.2509 - val_loss: 2.3082 - val_accuracy: 0.0992
Epoch 6/10
1688/1688 [==============================] - 66s 39ms/step - loss: 2.3095 - accuracy: 0.1035 - val_loss: 2.3107 - val_accuracy: 0.0952
Epoch 7/10
1688/1688 [==============================] - 66s 39ms/step - loss: 2.3080 - accuracy: 0.1044 - val_loss: 2.3036 - val_accuracy: 0.0960
Epoch 8/10
1688/1688 [==============================] - 66s 39ms/step - loss: 2.3085 - accuracy: 0.1043 - val_loss: 2.3062 - val_accuracy: 0.1113
Epoch 9/10
1688/1688 [==============================] - 66s 39ms/step - loss: 2.3089 - accuracy: 0.1049 - val_loss: 2.3081 - val_accuracy: 0.1045
Epoch 10/10
1688/1688 [==============================] - 67s 40ms/step - loss: 2.3088 - accuracy: 0.1044 - val_loss: 2.3101 - val_accuracy: 0.1050
Epoch 1/10
1688/1688 [==============================] - 67s 40ms/step - loss: 2.3083 - accuracy: 0.1059 - val_loss: 2.3061 - val_accuracy: 0.1113
Epoch 2/10
1688/1688 [==============================] - 68s 40ms/step - loss: 2.3091 - accuracy: 0.1036 - val_loss: 2.3088 - val_accuracy: 0.0960
Epoch 3/10
1688/1688 [==============================] - 67s 40ms/step - loss: 2.3085 - accuracy: 0.1028 - val_loss: 2.3106 - val_accuracy: 0.0960
Epoch 4/10
1688/1688 [==============================] - 67s 40ms/step - loss: 2.3079 - accuracy: 0.1044 - val_loss: 2.3066 - val_accuracy: 0.0960
Epoch 5/10
1688/1688 [==============================] - 67s 40ms/step - loss: 2.3081 - accuracy: 0.1040 - val_loss: 2.3068 - val_accuracy: 0.1050
Epoch 6/10
1688/1688 [==============================] - 68s 40ms/step - loss: 2.3086 - accuracy: 0.1046 - val_loss: 2.3078 - val_accuracy: 0.1113
Epoch 7/10
1688/1688 [==============================] - 67s 40ms/step - loss: 2.3081 - accuracy: 0.1043 - val_loss: 2.3112 - val_accuracy: 0.0952
Epoch 8/10
1688/1688 [==============================] - 67s 40ms/step - loss: 2.3084 - accuracy: 0.1042 - val_loss: 2.3089 - val_accuracy: 0.0960
Epoch 9/10
1688/1688 [==============================] - 68s 40ms/step - loss: 2.3085 - accuracy: 0.1032 - val_loss: 2.3090 - val_accuracy: 0.1113
Epoch 10/10
1688/1688 [==============================] - 67s 40ms/step - loss: 2.3085 - accuracy: 0.1044 - val_loss: 2.3070 - val_accuracy: 0.1113
```

0.1.6 (f) Analysis

i. A dropout layer adds regularisation to a CNN, as it randomly removes nodes from the neural network temporarily, to be able to reduce overfitting. Randomly sampling nodes allows the model to reduce thinning while training. We can see in our experimentation that the training accuracy was lower than 1 compared to other models. This shows that the model did not memorise the training set as it did without the dropout layer. We can also see that over the training epochs, both training and validation accuracies have big changes that can be seen from the zig zagged plot. This is due to some nodes performing better than others. At the end of the training, the model reduces overfitting as the validation accuracy peaks and has an upward trend in the last 10 epochs.

ii. Adding more pooling layers allows a model to learn more in-depth and abstract structures like images. Comparing the results from question e-ii (CNN with 1 layer and dropout) to e-iii (CNN with 2 layers and dropout), the extra layer helped reduce overfitting further, and

resulted in a slightly lower accuracy. The performance of both models are really high, but the 2-layered network's is more realistic.

iii. In order to find the best fit for a CNN, it is a good practice to experiment with different learning rates. For complex tasks such as classifying images, a very low learning rate will result in an overly complex model with increased training time, whereas a very large learning rate will result in unstable and non-efficient training. Comparing the 0.001 learning rate to our baseline (0.1), we can see that the validation accuracy is lower than 0.9915 (0.9910). The runtime for this model took much longer, and still resulted in a slightly lower accuracy. For the high learning rate 0.1, the validation accuracy was drastically lower, 0.1113. This was due to the model not being able to learn to recognise digits with such a big step size.

0.2 Random Forests for Image Approximation

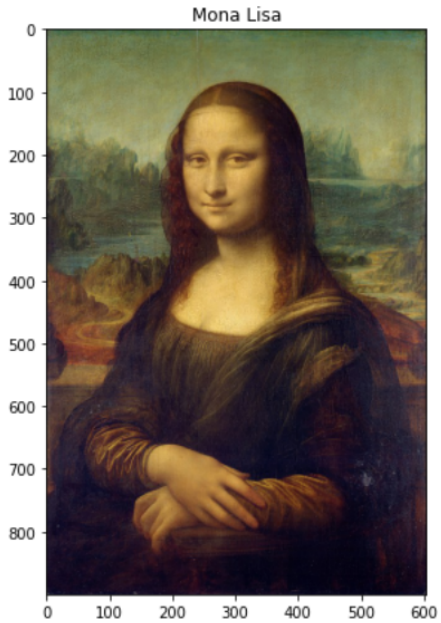
0.2.1 (a)

We start by downloading and printing the photo of the Mona Lisa painting.

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsRegressor

from google.colab import drive

drive.mount('/content/drive/')
%cd /content/drive/MyDrive/MonaLisa/
```

```
Mounted at /content/drive/
/content/drive/MyDrive/MonaLisa
```

```
img = plt.imread('monaLisa2.jpg')
plt.figure(figsize=[7,7])
plt.imshow(img);
plt.title("Mona Lisa");
```

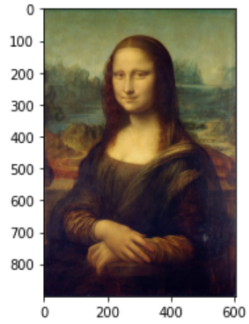


```
height, width = img.shape[0], img.shape[1]
print(height, width)
```

```
900 604
```

```
img_array = np.asarray(img)
img_array = np.reshape(img_array, (height, width, 3)) / 255
plt.imshow(img_array)
print(img_array.shape)
```

```
(900, 604, 3)
```



0.2.2 (b) Preprocessing the Input

We then randomly pick 5000 pixels for the input. The random coordinates are picked using the np.random function, within the ranges width and height of the image.

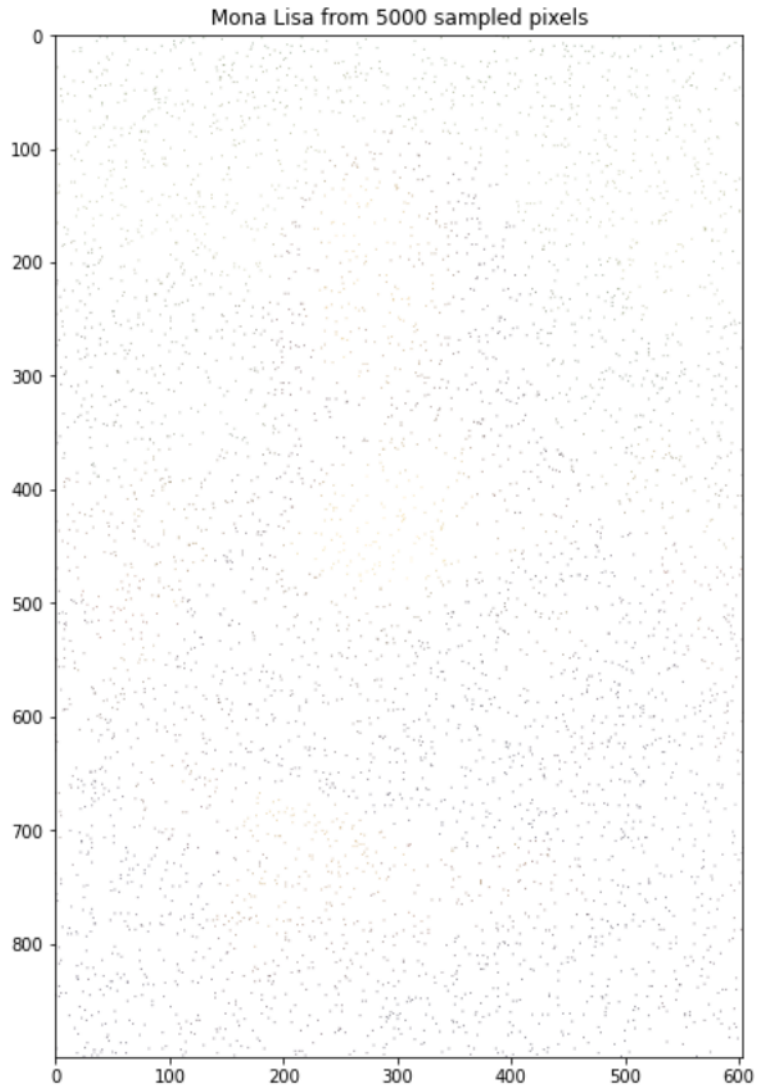
```
y_5000 = np.random.choice(height, size=5000, replace=True)
x_5000 = np.random.choice(width, size=5000, replace=True)
```

```
coordinates = list(zip(x_5000, y_5000))
print(len(coordinates))
```

```
5000
```

0.2.3 (c) Preprocessing the Output

```
img_5000px = np.zeros([height, width, 3]) + 255
pixels = []
for x, y in coordinates:
  pixels.append(img[y,x])
  img_5000px[y,x] = img[y,x]

# Dividing by 255 to rescaling the pixel intensities
img_5000px = img_5000px / 255

plt.figure(figsize=[11,11])
plt.imshow(img_5000px);
plt.title("Mona Lisa from 5000 sampled pixels");
```



We decided to use a function that maps (x,y) coordinates to RGB values. We selected the RGB values from the image array of the randomly picked pixels, and created a new image array with the selected pixels. We wanted to keep the colour as it helps visualise the effects of the different steps taken better when implementing random forests.

We rescaled the pixel intensities to lie between 0 and 1. No other preprocessing steps were needed for random forests, as for this regression task, random forests average the results of multiple decision trees.

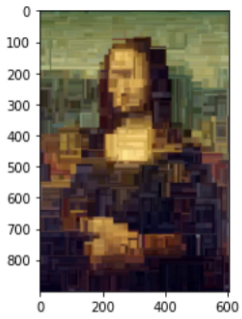
0.2.4 (d) Building the Random Forest

```
trainset = np.array([y_5000, x_5000]).T
```

```
def build_random_forest(n_estimators=10, max_depth=None, min_samples_split=2, min_samples_leaf=1):
    rf = RandomForestRegressor(n_estimators=n_estimators,
                               max_depth=max_depth,
                               min_samples_split=min_samples_split,
                               min_samples_leaf=min_samples_leaf)
    rf.fit(trainset, pixels)

    prediction = np.zeros([height, width, 3])
    for i in range(height):
        for j in range(width):
            prediction[i,j] = rf.predict(np.array([i,j]).reshape(1,-1))
    return prediction / 255
```

```
def experiment_depths(depths):
  for max_depth in depths:
    rf_image = build_random_forest(n_estimators=1, max_depth=max_depth)
    plt.imshow(rf_image)
    title = " Depth=" + str(max_depth)
    plt.title(title);
    plt.show()
```

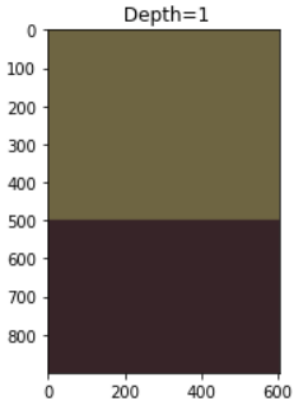
```
img_rf = build_random_forest(1)
plt.imshow(img_rf)
plt.show()
```

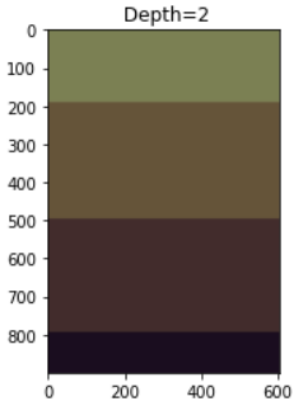


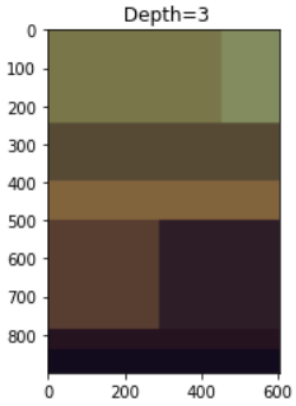
0.2.5 (e) Experimentation

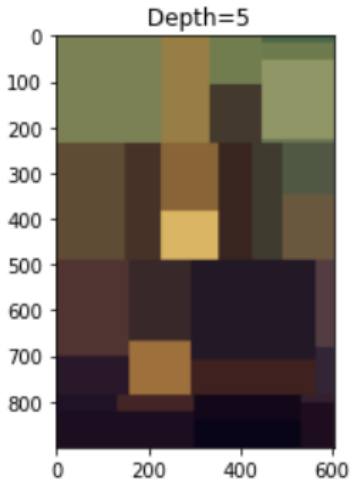
i. When we increase tree depth, we increase the number of possible values the pixel intensity can take. For a tree with depth 2, 2^2 pixel intensity values are generated. As the depth of the tree, n, increases, the pixel intensity values increase in the function of 2^n

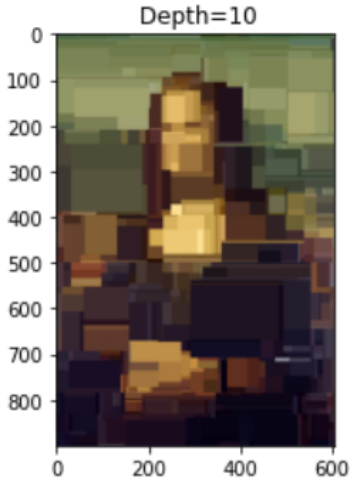
```
depths = [1,2,3,5,10,15]

experiment_depths(depths)
```

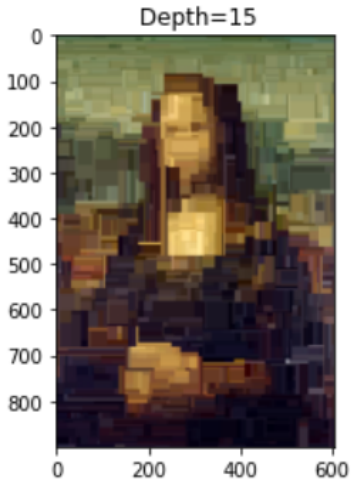










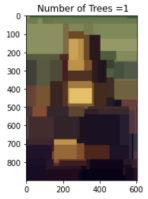


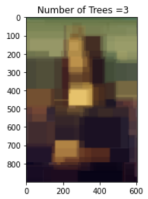
ii. If we call the number of trees we pick n, we then take the average of n different predictions generated by those trees. This reduces the noise in regression by making the regressor more

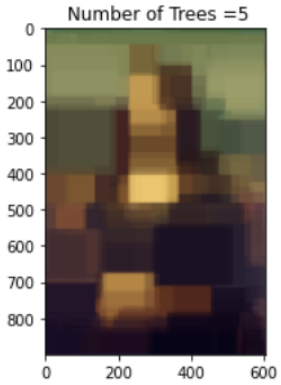
tolerant to noisy prediction as the more trees we use, the better results we get. Every tree in the prediction phase might use different features, each tree will result with different split points. As we average these predictions, the pixel intensity values will yield images that have less sharpness and are more connected, and will be more blurry.

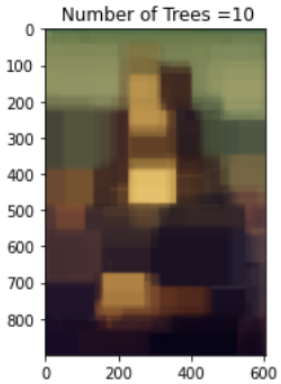
```
def experiment_numTrees(numTrees):
  for num in numTrees:
    rf_image = build_random_forest(n_estimators=num, max_depth=7)
    plt.imshow(rf_image)
    title = " Number of Trees =" + str(num)
    plt.title(title);
    plt.show()
```

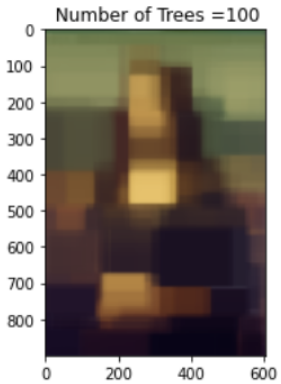
```
numTrees = [1,3,5,10,100]

experiment_numTrees(numTrees)
```





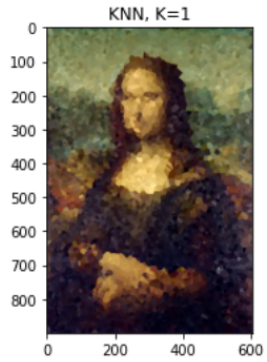






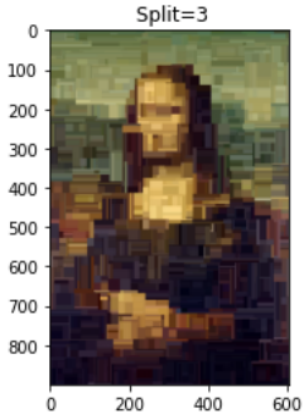
iii. The K-NN regressor looks for the pixel in training set with coordinates closest to each point in question. Since the training pixels were randomly picked, all regions have the same resolution. This is due to the fact that, the training test contains equal number of "ground truth" pixels per unit area anywhere in the image. This results in all regions ahving the same type and level of granularity. The grain boundaries in the image would correspond to K-NN boundaries.

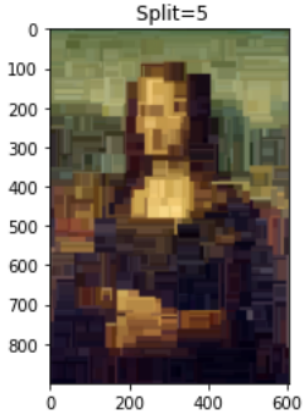
```
knn = KNeighborsRegressor(n_neighbors=1)
knn.fit(trainset, pixels)

prediction = np.zeros([height, width, 3])
for i in range(height):
    for j in range(width):
        prediction[i,j] = knn.predict(np.array([i,j]).reshape(1,-1))

plt.imshow(prediction / 255)
plt.title('KNN, K=1');
plt.show()
```



iv. We experimented with minimum samples split. We generated images for 3, 5 and 20 minimum samples split. The min_samples_split argument allows the random forest to create arbitrary small leaves, if there are enough samples at the internal node. We can see that since we average more samples as the number grows, the images get more blurry.

```
splits = [3,5,20]

experiment_splits(splits)
```





0.2.6 (f) Analysis

i. The decision rule at split point of each tree takes an input at a position (x, y) and outputs a prediction (x, y) for the next tree split. At the leaf node, the output is the RGB values.

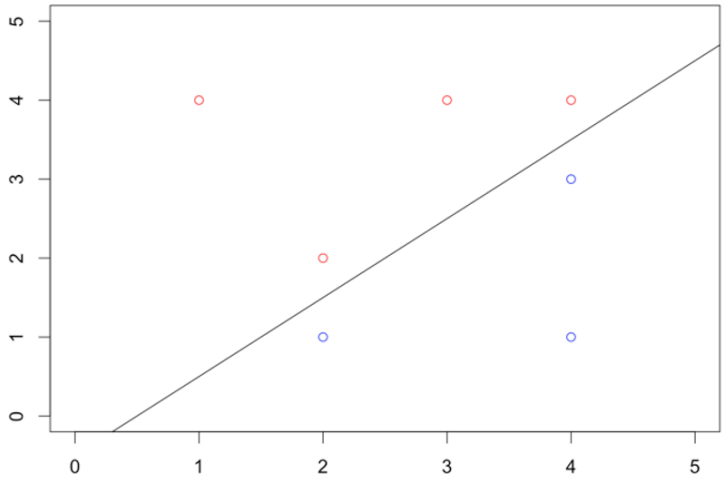
Split Rule = (left branch of the tree — $x >= threshold$) (right branch of the tree — $x < threshold$)

ii. For the root node with only one split, we get two images that are squared and stacked vertically on top of each other. This is due to the painting having a darker average colour on the bottom half of the image and a lighter at the top half. Since most of the "high-level" variation is found along the middle axis, the impurity can be reduced the most by splitting the two half planes to separate the picture vertically, at the root node.

Written Exercises

0.2.7 1. Maximum Margin Classifiers

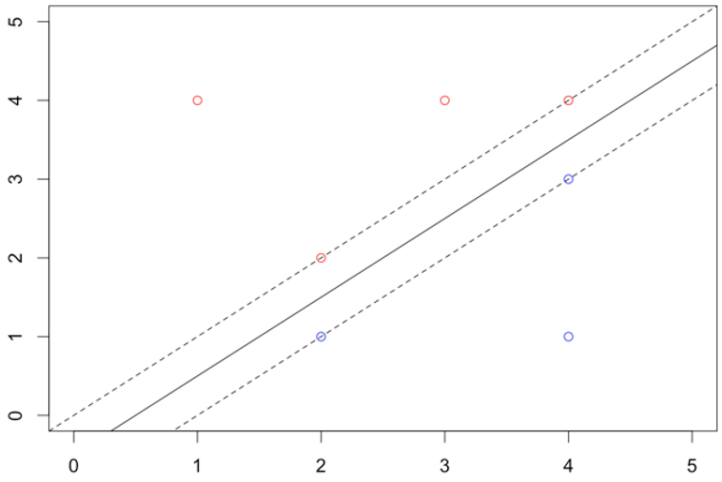
a. We plot the points as shown. The hyperplane has the equation $X2 = X1 - 0.2$.



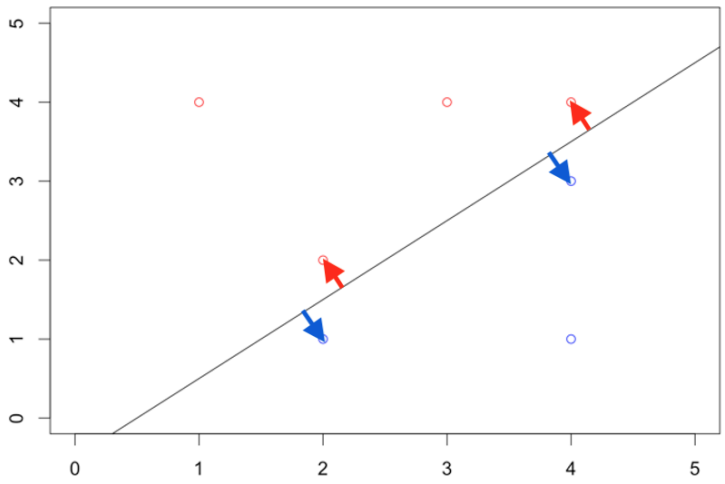
b. Classify Red if $X1–X2 - 0.5 < 0$, Blue otherwise.

$\beta_1 = 1; \beta_2 = -1; \beta_0 = -0.5$

c. The maximal margin zone is between blue(2, 1), red(2, 2), and blue(4,3), red(4, 4).



d. We take the dot product of the unit vector and support points. The support vectors are blue(2, 1), red(2, 2), and blue(4,3), red(4, 4).



e. The seventh observation is not a support vector, since it is far away from the margin region. Therefore, a slight movement would not move the maximum margin classifier.

f. The hyperplane has the equation $X2 = X1 - 0.2$. The hyperplane isn't a maximum-margin separating hyperplane, although it separates different classes of data correctly. Points have higher chance of been misclassified as blue.

g. Adding a blue point to (2,4) would make the data point inseparable by a hyperplane.