CS5785 Applied Machine Learning Homework 4

Author: Yixuan (Rylee) Li - yl2557
Teammate: Ningran Song - ns632

November 2021

1. *CODING EXERCISES*

1.1 Convolutional Neural Networks

I imported library using the following code snippet:

```
# import library
import keras
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np

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

1.1.1 Loading dataset

I downloaded the dataset:

```
# import mnist and download dataset
from keras.datasets import mnist
(train_X, train_Y), (test_X, test_Y) = mnist.load_data()
```

To verify that I had loaded the dataset correctly, I printed out the shape of the train and test dataset matrices. The results were: train_X shape: (60000, 28, 28), test_X shape: (10000, 28, 28).

```
# print out the shape of the train and test dataset matrices
print("train_X shape:", train_X.shape) # train_X shape: (60000, 28, 28)
print("test_X shape:", test_X.shape) # test_X shape: (10000, 28, 28)
```

I picked 10 random images and visualized individual images in this dataset by using imshow() function in pyplot.

```
# Pick 10 random images from the training set
# visualize them by using imshow() function in pyplot
numImage = 10
fig = plt.figure(figsize=(8,3))
fig.suptitle('Visualize 10 random images from the training set', fontsize=16)
for i in range(numImage):
    ax = fig.add_subplot(2, 5, 1 + i, xticks=[], yticks=[])
```

```
    img = train_X[np.random.randint(train_X.shape[0])]
    plt.imshow(img, cmap="gray")
plt.tight_layout()
plt.show()
fig.savefig("visualize-10-random", facecolor='white', edgecolor='none') # save fig
```

Figure 1.1: Visualize 10 random images from the training set

1.1.2 Preprocessing

I reshaped the matrix such that we had a 28 x 28 x 1 sized matrix holding each input data-point. Then I scaled the pixel values such that they lied between 0.0 and 1.0. The print-out returned "train_X shape: (60000, 28, 28, 1), test_X shape: (10000, 28, 28, 1)".

```
# Preprocessing:
# 1. reshape the matrix such that we have a 28 x 28 x 1 sized matrix
# 2. scale the pixel values such that they lie between 0.0 and 1.0
def preprocess(data):
    data = data.astype("float32") / 255
    data = np.expand_dims(data, -1)
    return data

train_X = preprocess(train_X)
print("train_X shape:", train_X.shape) # train_X shape: (60000, 28, 28, 1)
test_X = preprocess(test_X)
print("test_X shape:", test_X.shape) # test_X shape: (10000, 28, 28, 1)
```

I then converted output variable into a one-hot vector by using the function to_categorical.

```
# convert output variable into a one-hot vector by using the function to_categorical
print("train_Y shape before one-hot encoding: ", train_Y.shape)
train_Y = to_categorical(train_Y)
test_Y = to_categorical(test_Y)
print("train_Y shape after one-hot encoding: ", train_Y.shape)
```

I visualized some random images using imshow() function:

```
# visualize some random images using imshow() function
numImage = 10
fig = plt.figure(figsize=(8,3))
fig.suptitle('Preprocessed: visualize some random images', fontsize=16)
for i in range(numImage):
    ax = fig.add_subplot(2, 5, 1 + i, xticks=[], yticks=[])
    img = train_X[np.random.randint(train_X.shape[0])]
```

```
    plt.imshow(img, cmap="gray")
plt.tight_layout()
plt.show()
fig.savefig("visualize-10-random-preprocessed", facecolor='white', edgecolor='none')
```

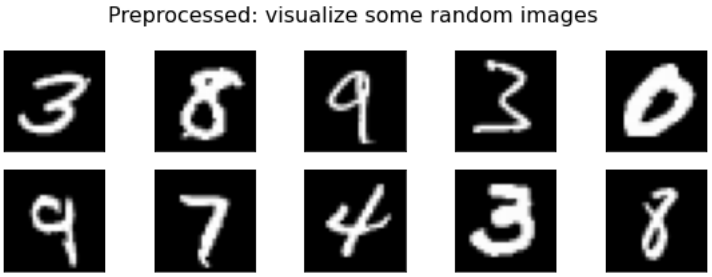


Figure 1.2: Preprocessed: visualize some random images

1.1.3 Implementation

I defined the CNN model as below.

```
# define a CNN model
# reference: homework prompt
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(lr=0.01, momentum=0.9)
    # compile model
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

I ran the CNN model, printed model.layers in the interactive shell. The model was generated as we defined.

```
# print model.layers in the interactive shell
# see that the model is generated as we defined
model = create_cnn()
model.layers
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 flatten (Flatten)           (None, 5408)              0

 dense (Dense)               (None, 100)               540900

 dense_1 (Dense)             (None, 10)                1010

=================================================================
Total params: 542,230
Trainable params: 542,230
Non-trainable params: 0
```

Figure 1.3: Model_CNN: Summary

1.1.4 Training and Evaluating CNN

I called the fit method with a validation split of 0.1 and the evaluate method on the test data-set. This model yielded: Test cross-entropy loss: 0.04300, Test accuracy: 0.98740.

```
# call the fit method with a validation split of 0.1
model.fit(train_X, train_Y, batch_size=32, epochs=10, validation_split=0.1)
# call the evaluate method on the test data-set
score = model.evaluate(test_X, test_Y, verbose=0)
# report the accuracy on test data
print('Test cross-entropy loss: %0.5f' % score[0])
print('Test accuracy: %0.5f' % score[1])
```

```
Epoch 1/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.1777 - accuracy: 0.9449 - val_loss: 0.0703 - val_
accuracy: 0.9802
Epoch 2/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0598 - accuracy: 0.9817 - val_loss: 0.0537 - val_
accuracy: 0.9855
Epoch 3/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0385 - accuracy: 0.9879 - val_loss: 0.0474 - val_
accuracy: 0.9872
Epoch 4/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0274 - accuracy: 0.9913 - val_loss: 0.0518 - val_
accuracy: 0.9852
Epoch 5/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0184 - accuracy: 0.9942 - val_loss: 0.0519 - val_
accuracy: 0.9862
Epoch 6/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0129 - accuracy: 0.9962 - val_loss: 0.0472 - val_
accuracy: 0.9882
Epoch 7/10
1688/1688 [==============================] - 11s 7ms/step - loss: 0.0083 - accuracy: 0.9977 - val_loss: 0.0608 - val_
accuracy: 0.9860
Epoch 8/10
1688/1688 [==============================] - 11s 7ms/step - loss: 0.0052 - accuracy: 0.9990 - val_loss: 0.0514 - val_
accuracy: 0.9882
Epoch 9/10
1688/1688 [==============================] - 11s 7ms/step - loss: 0.0034 - accuracy: 0.9996 - val_loss: 0.0517 - val_
accuracy: 0.9892
Epoch 10/10
1688/1688 [==============================] - 11s 6ms/step - loss: 0.0021 - accuracy: 0.9998 - val_loss: 0.0604 - val_
accuracy: 0.9882
```

Figure 1.4: Model_CNN: Accuracy

1.1.5 Experimentation

Run the above training for 50 epochs. Using pyplot, graph the validation and training accuracy after every 10 epochs. Is there a steady improvement for both training and validation accuracy?

I ran the above training for 50 epochs. For the training set, the accuracy first steadily improves from 0.99 to 1.00 for the first 10 epochs. Then it tends to become steady and maintain a level of 1.0. For the test set, the accuracy first drops for the first 10 epochs. Then it improves steadily in general.

```
# run model
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'])
# store the output of the fit method while training your network
# Using pyplot, graph the validation and training accuracy after every 10 epochs
plt.plot(range(0,51,10), epoch_history.history['accuracy'][::10]+
        [epoch_history.history['accuracy'][-1]])
plt.plot(range(0,51,10), epoch_history.history['val_accuracy'][::10]+
        [epoch_history.history['val_accuracy'][-1]])
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Model Accuracy - CNN')
plt.legend(['train', 'test'], loc='best')
plt.savefig("Model Accuracy - CNN", facecolor='white', edgecolor='none') # save fig
```

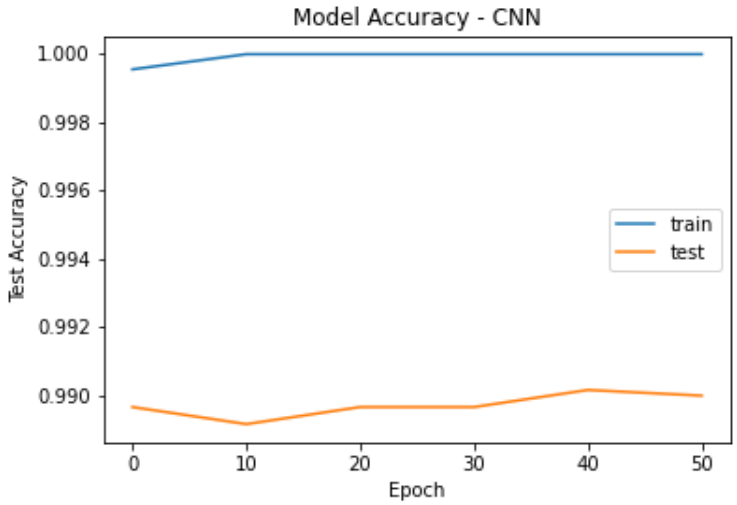


Figure 1.5: Model Accuracy - CNN

To avoid over-fitting in neural networks, we can 'drop out' a certain fraction of units randomly during the training phase. You can add the following layer (before the dense layer with 100 neurons) to your model defined in the function create_cnn. Now, train this CNN for 50 epochs. Graph the validation and train accuracy after every 10 epochs.

I dropped out a certain fraction of units randomly during the training phase:

```
# create the model
def create_cnn_dropped():
```

```
    # 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())
    # Drop out a certain fraction of units randomly during the training phase
    model.add(Dropout(0.5)) # added
    # 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'])
    return model
```

I graphed the validation and train accuracy after every 10 epochs.

```
# run the model
model_dropped = create_cnn_dropped()
epoch_history_dropped = model_dropped.fit(train_X, train_Y, batch_size=32,
            epochs=50, validation_split=0.1)
# print validation and training accuracy over epochs
print(epoch_history_dropped.history['accuracy'])
print(epoch_history_dropped.history['val_accuracy'])
# plot the accuracy
plt.plot(range(0,51,10), epoch_history_dropped.history['accuracy'][::10]+
            [epoch_history_dropped.history['accuracy'][-1]])
plt.plot(range(0,51,10), epoch_history_dropped.history['val_accuracy'][::10]+
            [epoch_history_dropped.history['val_accuracy'][-1]])
plt.title('Model Accuracy - CNN - Dropped Out')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['train', 'test'], loc='best')
plt.savefig("Model Accuracy - CNN - Dropped Out", facecolor='white', edgecolor='none')
```

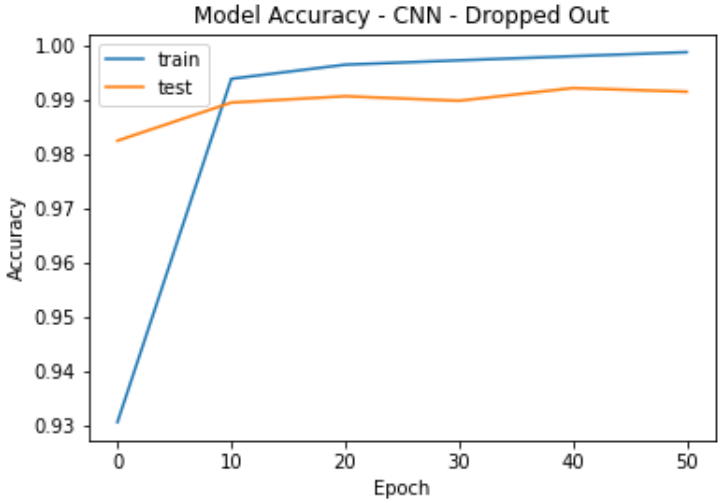


Figure 1.6: Model Accuracy - CNN - Dropped Out

Add another convolution layer and maxpooling layer to the create_cnn function defined above (immediately following the existing maxpooling layer). For the additional convolution layer, use 64 output filters. Train this for 10 epochs and report the test accuracy.

I added another convolution layer and maxpooling layer, trained this for 10 epochs and reported the test accuracy. This model yields - Test cross-entropy loss: 0.02221, Test accuracy: 0.99310.

```
# create model
def create_cnn_add_layer():
    # 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)))
    # Convolution layer
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
                input_shape=(13, 13, 32)))
    # Maxpooling layer
    model.add(MaxPooling2D((2, 2)))
    # Flatten output
    model.add(Flatten())
    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'])
    return model

# run model
```

```
model_add_layer = create_cnn_add_layer()
model_add_layer.fit(train_X, train_Y, batch_size=32, epochs=10, validation_split=0.1)
score_add_layer = model_add_layer.evaluate(test_X, test_Y, verbose=0)
print('Test cross-entropy loss: %0.5f' % score_add_layer[0])
print('Test accuracy: %0.5f' % score_add_layer[1])
```

```
Epoch 1/10
1688/1688 [==============================] - 23s 13ms/step - loss: 0.1963 - accuracy: 0.9371 - val_loss: 0.0463 - val
_accuracy: 0.9863
Epoch 2/10
1688/1688 [==============================] - 23s 14ms/step - loss: 0.0807 - accuracy: 0.9749 - val_loss: 0.0374 - val
_accuracy: 0.9902
Epoch 3/10
1688/1688 [==============================] - 24s 14ms/step - loss: 0.0625 - accuracy: 0.9801 - val_loss: 0.0375 - val
_accuracy: 0.9888
Epoch 4/10
1688/1688 [==============================] - 25s 15ms/step - loss: 0.0502 - accuracy: 0.9843 - val_loss: 0.0355 - val
_accuracy: 0.9905
Epoch 5/10
1688/1688 [==============================] - 34s 20ms/step - loss: 0.0442 - accuracy: 0.9860 - val_loss: 0.0321 - val
_accuracy: 0.9895
Epoch 6/10
1688/1688 [==============================] - 27s 16ms/step - loss: 0.0398 - accuracy: 0.9868 - val_loss: 0.0272 - val
_accuracy: 0.9920
Epoch 7/10
1688/1688 [==============================] - 25s 15ms/step - loss: 0.0359 - accuracy: 0.9884 - val_loss: 0.0302 - val
_accuracy: 0.9925
Epoch 8/10
1688/1688 [==============================] - 22s 13ms/step - loss: 0.0319 - accuracy: 0.9897 - val_loss: 0.0291 - val
_accuracy: 0.9930
Epoch 9/10
1688/1688 [==============================] - 23s 14ms/step - loss: 0.0293 - accuracy: 0.9901 - val_loss: 0.0277 - val
_accuracy: 0.9915
Epoch 10/10
1688/1688 [==============================] - 24s 14ms/step - loss: 0.0276 - accuracy: 0.9914 - val_loss: 0.0280 - val
_accuracy: 0.9925
Test cross-entropy loss: 0.02221
Test accuracy: 0.99310
```

Figure 1.7: Model Accuracy - CNN - Add Layer

```
# inspect layers
model_add_layer.layers
model_add_layer.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)           (None, 26, 26, 32)        320

 max_pooling2d_2 (MaxPooling  (None, 13, 13, 32)       0
 2D)

 conv2d_3 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_3 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 flatten_2 (Flatten)         (None, 1600)              0

 dropout_1 (Dropout)         (None, 1600)              0

 dense_4 (Dense)             (None, 100)               160100

 dense_5 (Dense)             (None, 10)                1010

=================================================================
Total params: 179,926
Trainable params: 179,926
Non-trainable params: 0
_________________________________________________________________
```

Figure 1.8: Model Accuracy - CNN - Add Layer - Summary

We used a learning rate of 0.01 in the given create_cnn function. Using learning rates of 0.001 and 0.1 respectively, train the model and report accuracy on test data-set. To be sure, we are working with 2 convolution layers and training for 10 epochs while doing this experiment

I defined the model as below.

```
# create model
def create_cnn_vary_rate(lr=0.01):
    # 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)))
    # Convolution layer
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
                input_shape=(13, 13, 32)))
    # Maxpooling layer
    model.add(MaxPooling2D((2, 2)))
    # Flatten output
    model.add(Flatten())
    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=lr , momentum=0.9)
    # compile model
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics =['accuracy'])
    return model
```

I used learning rates of 0.001 and trained the model and report accuracy on test data-set. This yiels - Test cross-entropy loss: 0.03893, Test accuracy: 0.98690.

```
# learning rate = 0.001
model_cnn_vary_rate_1 = create_cnn_vary_rate(lr=0.001)
model_cnn_vary_rate_1.fit(train_X, train_Y, batch_size=32, epochs=10, validation_split=0.1)
score_cnn_vary_rate_1 = model_cnn_vary_rate_1.evaluate(test_X, test_Y, verbose=0)
print('Test cross-entropy loss: %0.5f' % score_cnn_vary_rate_1[0])
print('Test accuracy: %0.5f' % score_cnn_vary_rate_1[1])
```

```
Epoch 1/10
1688/1688 [==============================] - 25s 14ms/step - loss: 0.4250 - accuracy: 0.8652 - val_loss: 0.1009 - val
_accuracy: 0.9742
Epoch 2/10
1688/1688 [==============================] - 24s 14ms/step - loss: 0.1608 - accuracy: 0.9504 - val_loss: 0.0736 - val
_accuracy: 0.9810
Epoch 3/10
1688/1688 [==============================] - 27s 16ms/step - loss: 0.1235 - accuracy: 0.9618 - val_loss: 0.0640 - val
_accuracy: 0.9822
Epoch 4/10
1688/1688 [==============================] - 25s 15ms/step - loss: 0.1037 - accuracy: 0.9683 - val_loss: 0.0577 - val
_accuracy: 0.9830
Epoch 5/10
1688/1688 [==============================] - 26s 15ms/step - loss: 0.0924 - accuracy: 0.9711 - val_loss: 0.0535 - val
_accuracy: 0.9847
Epoch 6/10
1688/1688 [==============================] - 24s 14ms/step - loss: 0.0835 - accuracy: 0.9744 - val_loss: 0.0462 - val
_accuracy: 0.9867
Epoch 7/10
1688/1688 [==============================] - 25s 15ms/step - loss: 0.0777 - accuracy: 0.9756 - val_loss: 0.0451 - val
_accuracy: 0.9870
Epoch 8/10
1688/1688 [==============================] - 24s 14ms/step - loss: 0.0734 - accuracy: 0.9767 - val_loss: 0.0452 - val
_accuracy: 0.9863
Epoch 9/10
1688/1688 [==============================] - 23s 14ms/step - loss: 0.0670 - accuracy: 0.9800 - val_loss: 0.0401 - val
_accuracy: 0.9895
Epoch 10/10
1688/1688 [==============================] - 23s 14ms/step - loss: 0.0655 - accuracy: 0.9796 - val_loss: 0.0433 - val
_accuracy: 0.9872
Test cross-entropy loss: 0.03893
Test accuracy: 0.98690
```

Figure 1.9: Model Accuracy - CNN - 0.001

I used learning rates of 0.1 and trained the model and report accuracy on test data-set. This yiels - Test cross-entropy loss: 2.30874, Test accuracy: 0.10100.

```
# learning rate = 0.1
model_cnn_vary_rate_2 = create_cnn_vary_rate(lr=0.1)
model_cnn_vary_rate_2.fit(train_X, train_Y, batch_size=32, epochs=10, validation_split=0.1)
score_cnn_vary_rate_2 = model_cnn_vary_rate_2.evaluate(test_X, test_Y, verbose=0)
print('Test cross-entropy loss: %0.5f' % score_cnn_vary_rate_2[0])
print('Test accuracy: %0.5f' % score_cnn_vary_rate_2[1])
```

```
Epoch 1/10
1688/1688 [==============================] - 23s 14ms/step - loss: 0.9928 - accuracy: 0.7178 - val_loss: 0.8907 - val
_accuracy: 0.6885
Epoch 2/10
1688/1688 [==============================] - 27s 16ms/step - loss: 1.2484 - accuracy: 0.6214 - val_loss: 1.0980 - val
_accuracy: 0.6158
Epoch 3/10
1688/1688 [==============================] - 25s 15ms/step - loss: 2.0908 - accuracy: 0.2276 - val_loss: 2.3197 - val
_accuracy: 0.1113
Epoch 4/10
1688/1688 [==============================] - 24s 14ms/step - loss: 2.3081 - accuracy: 0.1061 - val_loss: 2.3123 - val
_accuracy: 0.1050
Epoch 5/10
1688/1688 [==============================] - 24s 14ms/step - loss: 2.3079 - accuracy: 0.1066 - val_loss: 2.3106 - val
_accuracy: 0.0960
Epoch 6/10
1688/1688 [==============================] - 24s 14ms/step - loss: 2.3085 - accuracy: 0.1036 - val_loss: 2.3079 - val
_accuracy: 0.0952
Epoch 7/10
1688/1688 [==============================] - 24s 14ms/step - loss: 2.3078 - accuracy: 0.1041 - val_loss: 2.3092 - val
_accuracy: 0.0978
Epoch 8/10
1688/1688 [==============================] - 24s 14ms/step - loss: 2.3083 - accuracy: 0.1040 - val_loss: 2.3075 - val
_accuracy: 0.0978
Epoch 9/10
1688/1688 [==============================] - 23s 14ms/step - loss: 2.3077 - accuracy: 0.1056 - val_loss: 2.3080 - val
_accuracy: 0.1045
Epoch 10/10
1688/1688 [==============================] - 24s 14ms/step - loss: 2.3083 - accuracy: 0.1056 - val_loss: 2.3090 - val
_accuracy: 0.1045
Test cross-entropy loss: 2.30874
Test accuracy: 0.10100
```

Figure 1.10: Model Accuracy - CNN - 0.1

1.1.6 Analysis

Explain how the trends in validation and train accuracy change after using the dropout layer in the experiments.

After using the dropout layer in the experiments, the validation accuracy improves. However, the train accuracy does not become steady at the level of 1.0 (ie. actually reaches 1.0). Instead, it approaches 1.0 steadily. This is because the drop-out avoids over-fitting in neutral networks as we randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much in the following way: 1) During training, dropout samples from an exponential number of different "thinned" networks. 2) At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights.

How does the performance of CNN with two convolution layers differ as compared to CNN with a single convolution layer in your experiments?

To compare the performance as of epochs = 10, I ran the drop-out model with epochs = 10:

```
# run the model - dropped out as of epochs = 10
model_dropped_10 = create_cnn_dropped()
model_dropped_10.fit(train_X, train_Y, batch_size=32, epochs=10, validation_split=0.1)
score_cnn_dropped_10 = model_dropped_10.evaluate(test_X, test_Y, verbose=0)
print('Test cross-entropy loss: %0.5f' % score_cnn_dropped_10[0])
print('Test accuracy: %0.5f' % score_cnn_dropped_10[1])
```

```
Epoch 1/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.2208 - accuracy: 0.9312 - val_loss: 0.0818 - val_
accuracy: 0.9793
Epoch 2/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0897 - accuracy: 0.9729 - val_loss: 0.0658 - val_
accuracy: 0.9813
Epoch 3/10
1688/1688 [==============================] - 11s 7ms/step - loss: 0.0694 - accuracy: 0.9783 - val_loss: 0.0479 - val_
accuracy: 0.9867
Epoch 4/10
1688/1688 [==============================] - 11s 7ms/step - loss: 0.0552 - accuracy: 0.9825 - val_loss: 0.0451 - val_
accuracy: 0.9878
Epoch 5/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0465 - accuracy: 0.9852 - val_loss: 0.0476 - val_
accuracy: 0.9880
Epoch 6/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0403 - accuracy: 0.9871 - val_loss: 0.0424 - val_
accuracy: 0.9888
Epoch 7/10
1688/1688 [==============================] - 11s 7ms/step - loss: 0.0346 - accuracy: 0.9885 - val_loss: 0.0439 - val_
accuracy: 0.9882
Epoch 8/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0315 - accuracy: 0.9895 - val_loss: 0.0434 - val_
accuracy: 0.9893
Epoch 9/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0296 - accuracy: 0.9899 - val_loss: 0.0489 - val_
accuracy: 0.9875
Epoch 10/10
1688/1688 [==============================] - 12s 7ms/step - loss: 0.0259 - accuracy: 0.9911 - val_loss: 0.0436 - val_
accuracy: 0.9895
Test cross-entropy loss: 0.03918
Test accuracy: 0.98710
```

Figure 1.11: Model Accuracy - CNN - Dropped Out - 10

We see an overall improvement in the performance of CNN with two convolution layers compared to CNN with a single convolution layer. For the validation set, the accuracy increases from 0.9895 to 0.9925 as of epoch of 10; the cross-entropy loss decreases from 0.0378 to 0.0280 as of epoch of 10. For the train set, the accuracy decreases a little bit (ie. drop from 0.9924 to 0.9914 as of epoch of 10); the cross-entropy loss decreases from 0.0228 to 0.0276 as of epoch of 10. For the test set, the accuracy increases from 0.98710 to 0.99310 as of epoch of 10; the cross-entropy loss decreases from 0.03918 to 0.02221 as of epoch of 10.

How did changing learning rates change your experimental results in part (iv)?

To compare the effect of changing learning rates, I also trained the model with a learning rate of 0.01 with two additional layers and the drop-out.

```
# learning rate = 0.01
model_cnn_vary_rate_3 = create_cnn_vary_rate(lr=0.01)
model_cnn_vary_rate_3.fit(train_X, train_Y, batch_size=32, epochs=10, validation_split=0.1)
score_cnn_vary_rate_3 = model_cnn_vary_rate_3.evaluate(test_X, test_Y, verbose=0)
print('Test cross-entropy loss: %0.5f' % score_cnn_vary_rate_3[0])
print('Test accuracy: %0.5f' % score_cnn_vary_rate_3[1])
```

```
Epoch 1/10
1688/1688 [==============================] - 23s 13ms/step - loss: 0.2002 - accuracy: 0.9377 - val_loss: 0.0550 - val
_accuracy: 0.9842
Epoch 2/10
1688/1688 [==============================] - 23s 14ms/step - loss: 0.0815 - accuracy: 0.9740 - val_loss: 0.0440 - val
_accuracy: 0.9888
Epoch 3/10
1688/1688 [==============================] - 22s 13ms/step - loss: 0.0620 - accuracy: 0.9807 - val_loss: 0.0357 - val
_accuracy: 0.9905
Epoch 4/10
1688/1688 [==============================] - 22s 13ms/step - loss: 0.0529 - accuracy: 0.9835 - val_loss: 0.0361 - val
_accuracy: 0.9903
Epoch 5/10
1688/1688 [==============================] - 31s 18ms/step - loss: 0.0429 - accuracy: 0.9861 - val_loss: 0.0311 - val
_accuracy: 0.9918
Epoch 6/10
1688/1688 [==============================] - 23s 14ms/step - loss: 0.0379 - accuracy: 0.9874 - val_loss: 0.0331 - val
_accuracy: 0.9907
Epoch 7/10
1688/1688 [==============================] - 24s 14ms/step - loss: 0.0361 - accuracy: 0.9882 - val_loss: 0.0278 - val
_accuracy: 0.9927
Epoch 8/10
1688/1688 [==============================] - 28s 16ms/step - loss: 0.0318 - accuracy: 0.9898 - val_loss: 0.0263 - val
_accuracy: 0.9937
Epoch 9/10
1688/1688 [==============================] - 27s 16ms/step - loss: 0.0304 - accuracy: 0.9897 - val_loss: 0.0267 - val
_accuracy: 0.9945
Epoch 10/10
1688/1688 [==============================] - 25s 15ms/step - loss: 0.0276 - accuracy: 0.9909 - val_loss: 0.0271 - val
_accuracy: 0.9925
Test cross-entropy loss: 0.02530
Test accuracy: 0.99190
```

Figure 1.12: Model Accuracy - CNN - 0.01

The model with a learning rate of 0.01 outperforms those of 0.1 and 0.001 in terms of accuracy and loss on both the train, the validation, and the test set. Compared to the model with a learning rate of 0.1, that of 0.001 performs better. A detailed comparison among models with different learning rates as of epoch 10 could be found below:

Rate	Train-Accuracy	Train-Loss	Validation-Accuracy	Validation-Loss	Test-Accuracy	Test-Loss
0.1	0.1056	2.3083	0.1045	2.3090	0.10100	2.30874
0.01	0.9909	0.0276	0.9925	0.0271	0.99190	0.02530
0.001	0.9796	0.0655	0.9872	0.0433	0.98690	0.03893

1.2 Random Forests for Image Approximation

1.2.1 Start with an image of the Mona Lisa

I imported the mona lisa image using the following code snippet:

```
# Read the picture from the relative path
img = plt.imread('mona_lisa.jpeg')
plt.imshow(img)
```

```
plt.title("Mona Lisa")
plt.show() # show the image
```

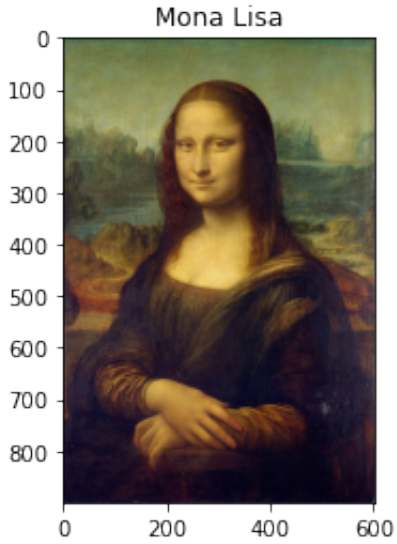


Figure 1.13: Mona Lisa

1.2.2 Preprocessing the input

We preprocessed the input by uniformly sampling 5,000 random (x,y) coordinate locations. And we didn't take any further preprocessing steps such as mean subtraction, standardization, or unit-normalization. These steps were not necessary for random forests inputs since decision trees wouldn't be impacted by the structure of the data.

```
# preprocessing the input
sample = []
sampleNum = 5000
m, n = img.shape[0], img.shape[1]
# uniformly sample 5,000 random (x, y) coordinate locations
for i in range(sampleNum):
    tmp = []
    tmp.append(np.random.randint(m))
    tmp.append(np.random.randint(n))
    sample.append(tmp)
sample = np.array(sample)
print(sample.shape) # (5000, 2)
print(sample) # print out the sample set
```

```
(5000, 2)
[[298 236]
 [377 333]
 [782 438]
 ...
 [134 261]
 [250  84]
 [130  36]]
```

Figure 1.14: Mona Lisa - Sample Shape of (5000, 2) + Print out the Set

1.2.3 Preprocessing the output

We decided to choose the first method of converting the image to gray scale, and rescale the pixel intensities to lie between 0.0 and 1.0.. But we didn't go any further since method of random forests required few preprocessing steps. And the point of this problem was to reconstruct the given image. The preprocessing we took worked fine for the given context.

```
# Rescale the pixel intensities to lie between 0.0 and 1.0.
scaled_img = np.array(img, dtype=float)
for i in range(m):
    for j in range(n):
        scaled_img[i,j] = img[i,j] / 255.0
# Show the img
plt.imshow(scaled_img)
plt.xlabel("Mona Lisa - Scaled")
plt.show()
```

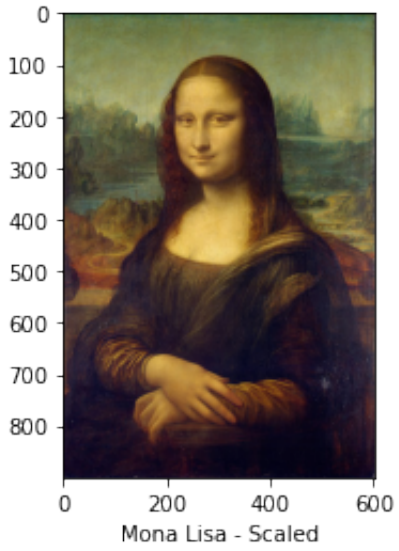


Figure 1.15: Mona Lisa - Scaled

```
# Sample pixel values at each of the given coordinate locations
sample_point = np.zeros([m, n, 3], dtype = np.uint8)
sample_point.fill(255)
for i in range(sampleNum):
    sample_point[sample[i][0]][sample[i][1]] = scaled_img[sample[i][0]][sample[i][1]]
plt.imshow(sample_point)
plt.title("Sample coordinate locations")
plt.show()
```

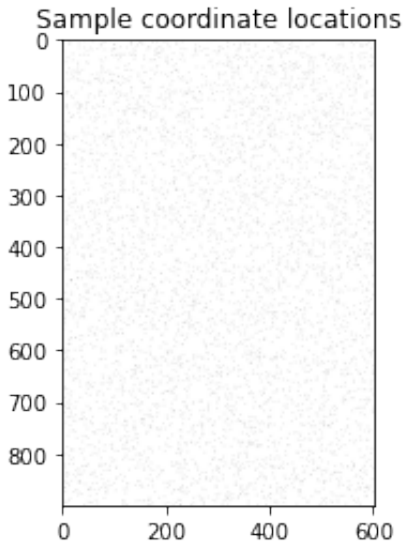


Figure 1.16: Sample coordinate locations

```
# Convert the scaled img to greyscale
# reference: https://stackoverflow.com/questions/12201577/
#           how-can-i-convert-an-rgb-image-into-grayscale-in-python
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.144])
# display the image
scaled_gray_img = rgb2gray(scaled_img)
plt.imshow(scaled_gray_img, cmap = plt.get_cmap('gray'))
plt.title("Mona Lisa - Gray Scale")
plt.show()
```

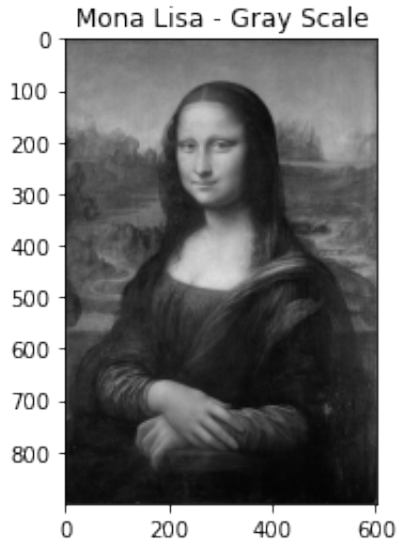


Figure 1.17: Mona Lisa - Gray Scale

1.2.4 Prepare to build the final image

We used random forest regressor from sklearn library. Reference: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

```
# get the label from sample
label = []
for point in sample:
    label.append(scaled_gray_img[point[0], point[1]])

# run random forest:
# reference: https://scikit-learn.org/stable/modules/
#      generated/sklearn.ensemble.RandomForestRegressor.html
from sklearn.ensemble import RandomForestRegressor
def random_forest(data, label, img_name, depth = None, n_tree = 1, samples_split = 2):
    rf = RandomForestRegressor(max_depth = depth, n_estimators = n_tree,
        min_samples_split = samples_split)
    rf.fit(data, label)
    pred = np.zeros([m, n, 3])
    for i in range(m):
        for j in range(n):
            point = [i,j]
            point = np.array(point)
            pred[i,j] = rf.predict(point.reshape(1,-1)) / 255
    # use imshow to view the result
    plt.title(img_name)
    plt.imshow(pred * 255.0)
    plt.show()
    return pred, rf

# for each pixel of the output, feed the pixel coordinate through the random forest
# color the resulting pixel with the output prediction
random_forest(sample, label, "Run Random Forest on Mona Lisa")
```

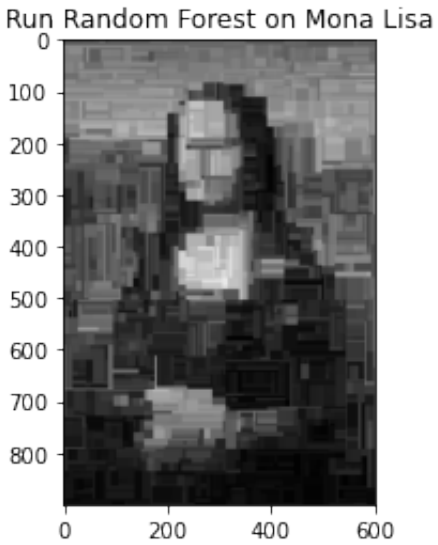


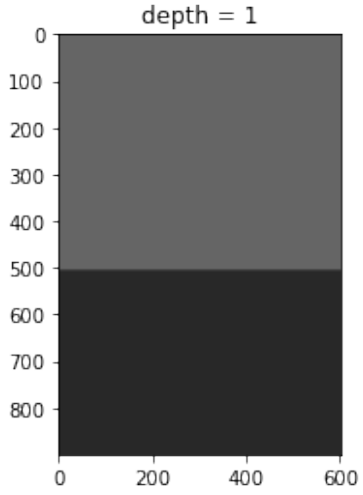
Figure 1.18: Run Random Forest on Mona Lisa

1.2.5 Experimentation

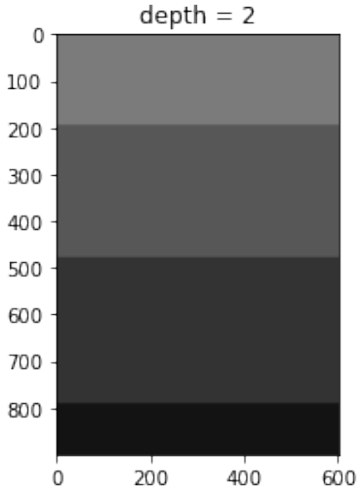
Repeat the experiment for a random forest containing a single decision tree, but with depths 1, 2, 3, 5, 10, and 15. How does depth impact the result? Describe in detail why

From the images below, we can see the image gets clearer as the depth increases (ie. get closer to the original mona lisa image). This is because the deeper the tree, the more splits it has and it captures more information about the data. When the depth gets larger, more splits a tree can make before coming to a prediction. For example, for depth = 1, there are 2 possible categories that each pixel can fall into; for depth = 2, there are 4 divisions. While we increase the depth, each pixel will be further categorized for deeper layers. Hence, we can construct a clearer and detailed image with increasing depths.

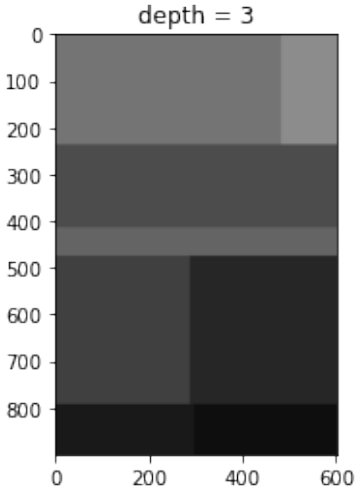
```
# depths 1, 2, 3, 5, 10, and 15.
depth = [1, 2, 3, 5, 10, 15]
for i in depth:
    img_name = "depth = " + str(i)
    random_forest(sample, label, img_name, depth=i)
```



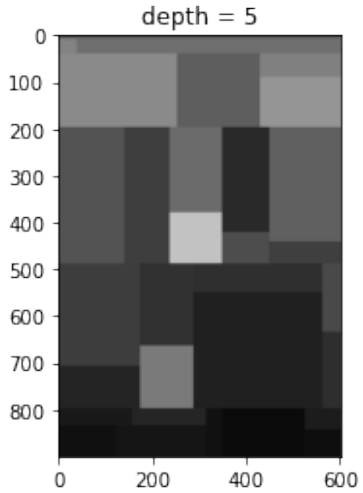
(a) depth = 1



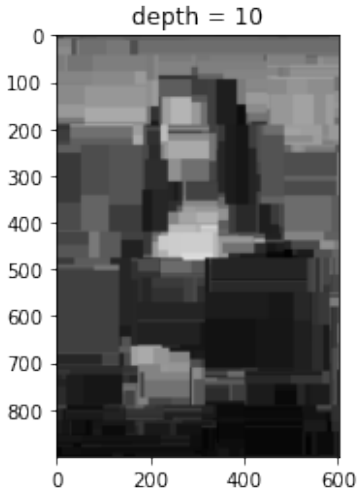
(b) depth = 2



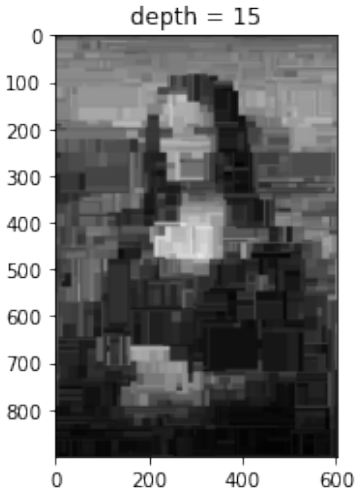
(c) depth = 3



(d) depth = 4



(e) depth = 10

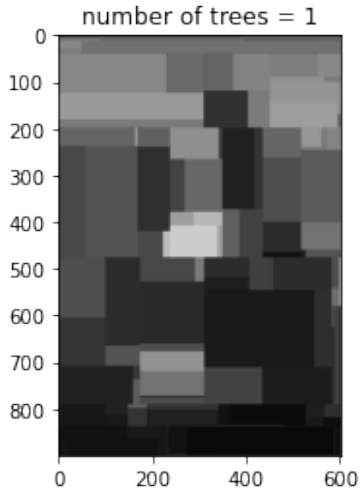


(f) depth = 15

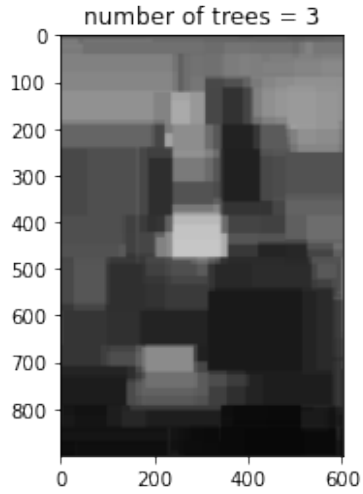
Repeat the experiment for a random forest of depth 7, but with number of trees equal to 1, 3, 5, 10, and 100. How does the number of trees impact the result? Describe in detail why.

From the images below, we can see the image gets clearer and smoother. As we have more trees in the forest, we generate the prediction by averaging the different predictions that each tree yields. Hence, the model will yield higher prediction accuracy with better tolerance for abnormal value and noise. Meanwhile, it results in reduced sharpness and increased blurriness since split points may be different for each prediction.

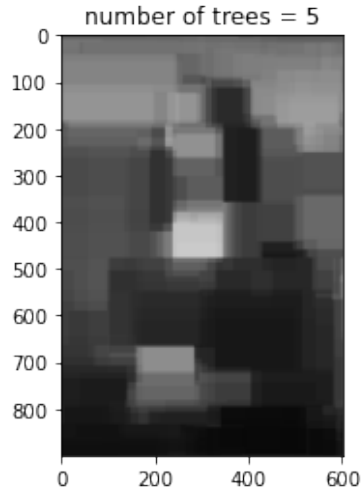
```
# different number of trees
numTree = [1, 3, 5, 10, 100]
for num in numTree:
    img_name = "number of trees = " + str(num)
    random_forest(sample, label, img_name, n_tree=num, depth=7)
```



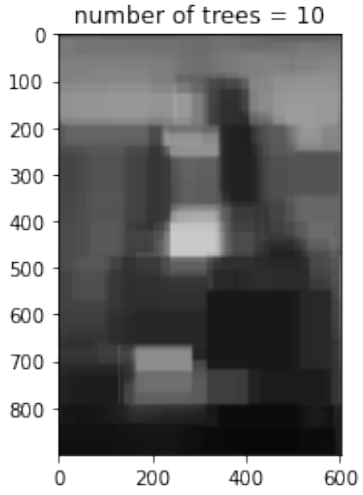
(a) number of trees = 1



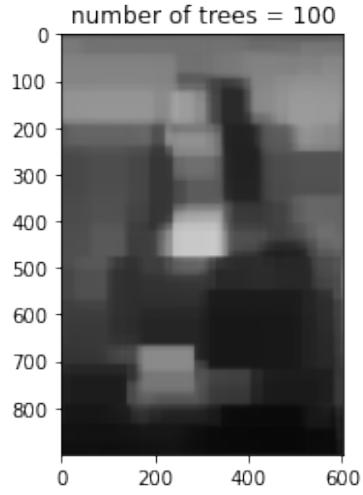
(b) number of trees = 3



(c) number of trees = 5



(d) number of trees = 10



(e) number of trees = 100

As a simple baseline, repeat the experiment using a k-NN regressor, for k = 1. This means that every pixel in the output will equal the nearest pixel from the "training set." Compare and contrast the outlook: why does this look the way it does?

- From the image below, it seems that the image has been divided into different patches of color pixels, where each group shares the same pixel intensity value. This is because k-nearest neighbors algorithm predicts the class labels for each pixel based on the class of the k nearest neighbors.
- Compared to random forest which generates patches in regular rectangular shape of different size (in the case of our image, mona lisa), the knn model generates patches in irregular shape. Random forest model generates divisions in rectangular shape because of the algorithm of decision tree. KNN yields irregular shape as each pixel looks for the nearest class.
- We can see some sort of grains in the image generated with knn model. This is because we generate our sample points from uniform distribution and all regions have the same type of grainy structure.

```
# Reference: https://scikit-learn.org/stable/modules
#                   /generated/sklearn.neighbors.KNeighborsClassifier.html
def kNN(title = "k-NN"):
    knn = KNeighborsClassifier(n_neighbors=1)
    knn.fit(sample, [int(l*255) for l in label])
    predictions = np.zeros([m, n, 3])
    for i in range(m):
        for j in range(n):
            point = [i,j]
            point = np.array(point)
            predictions[i,j] = knn.predict(point.reshape(1,-1)) / 255
    plt.xlabel(title)
    plt.imshow(predictions)
    plt.show()
    return predictions
# run the model
pred_knn = kNN()
```

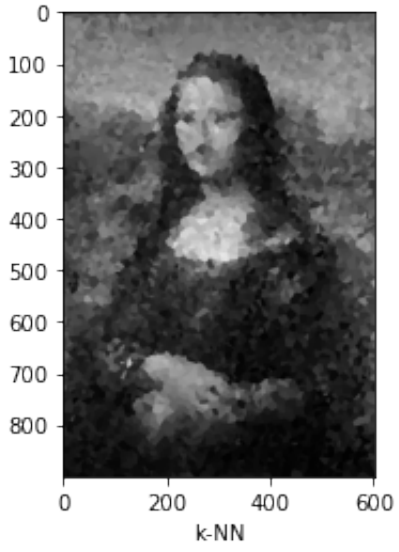
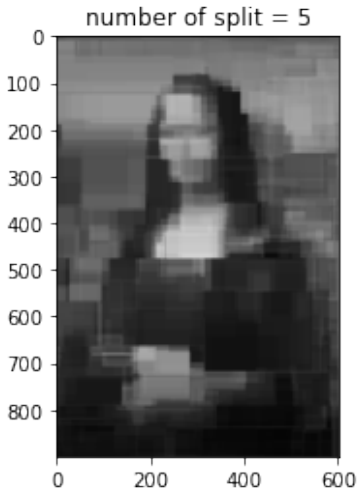


Figure 1.21: Mona Lisa - KNN

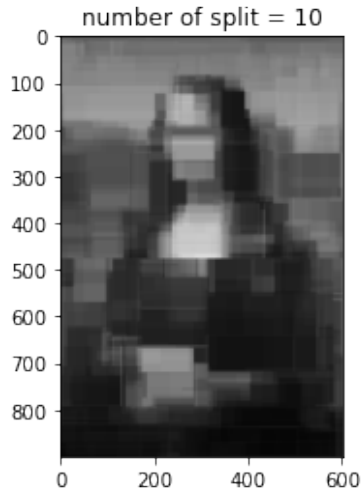
Experiment with different pruning strategies of your choice.

From the above experiments we run on the depth and the number of trees, we can see that model with depth of 10 and number of tree of 5 yields good performance and acceptable training time. We decide to pursue this setting and explore pruning methods on the parameter min_sample_split (ie. minimum number of samples required to split an internal node). We adjust the min_sample_split parameter for 4 different values (ie. 5, 10, 15, 20). By limiting the lower bounds of the samples on a node that can split, we found that the image gets blurrier as the lower bound increases. The default value of min_sample_split set by sklearn (ie. min_sample_split = 2) returns the best performance score.

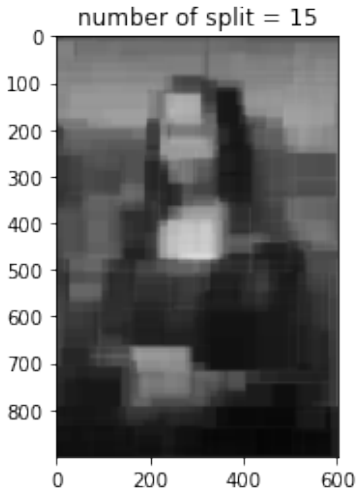
```
# min_samples_split: [5, 10, 15, 20]
numSplit = [5, 10, 15, 20]
for num in numSplit:
    img_name = "number of split = " + str(num)
    random_forest(sample, label, img_name, n_tree=5, depth=10, samples_split=num)
```



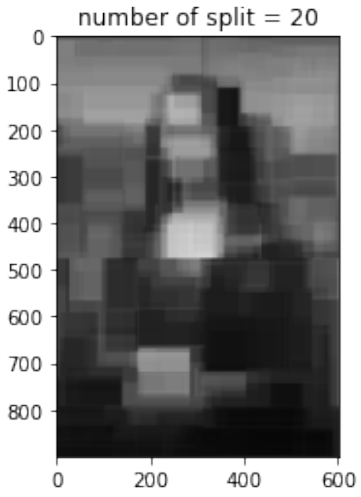
(a) min number of split = 5



(b) min number of split = 10



(c) min number of split = 15



(d) min number of split = 20

1.2.6 Analysis

What is the decision rule at each split point? Write down the 1-line formula for the split point at the root node for one the trained decision trees inside the forest. Feel free to define any variables you need.

Here, we have the coordinate (x_0, x_1) and output a node on the next level of the tree. When it reaches the lead node, it outputs the color value. The decision rule $r : \mathcal{X} \to \{\text{true}, \text{false}\}$ is a partition of the feature space into two disjoint regions.

$$r(x) = \begin{cases} true & \text{if } x_i \geq threshold \\ false & \text{if } x_i < threshold \end{cases} \tag{1.1}$$

Why does the resulting image look like the way it does? What shape are the patches of color, and how are they arranged?

- Random Forest: random forest generates patches in regular rectangular shape. And all these rectangles are arranged side-by-side; they are stacked vertically or horizontally. This is because the mechanism of decision tree follows the rule of using hyper-planes which are parallel or vertical to the axis and other hyper-planes, thereby generating rectangular shape.
- K-NN: K-NN model generates patches in irregular shape. They do not follow the arrangement of forming vertical or horizontal stacks. And the image seems grainy. This is because k-nearest neighbors algorithm predicts the class labels for each pixel based on the class of the k nearest neighbors. And we generate our sample points from uniform distribution and all regions have the same type of grainy structure.

2. *WRITTEN EXERCISES*

2.1 Maximum Margin Classifiers

2.1.1 Sketch the observations and the maximum-margin separating hyperplane.

```
# input the observations and their labels
X = np.array([[3,4],[2,2],[4,4],[1,4],[2,1],[4,3],[4,1]], dtype=np.int32)
y= np.array([1,1,1,1,0,0,0])

# fit the model, don't regularize for illustration purposes
clf = svm.SVC(kernel="linear", C=1000)
clf.fit(X, y)

plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Paired)

# plot the decision function
ax = plt.gca()
plt.xlim([0, 6])
plt.ylim([0, 6])
xlim = ax.get_xlim()
ylim = ax.get_ylim()

# create grid to evaluate model
xx = np.linspace(xlim[0], xlim[1], 30)
yy = np.linspace(ylim[0], ylim[1], 30)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T
Z = clf.decision_function(xy).reshape(XX.shape)

# plot decision boundary and margins
ax.contour(
    XX, YY, Z, colors="k", levels=[-1, 0, 1], alpha=0.5, linestyles=["--", "-", "--"]
)
# plot support vectors
ax.scatter(
    clf.support_vectors_[:, 0],
    clf.support_vectors_[:, 1],
    s=100,
    linewidth=1,
    facecolors="none",
    edgecolors="k",
)
```

```
# plot margin vectors
theta = clf.coef_[0]
theta0 = clf.intercept_
for idx in clf.support_[:4]:
    x0 = X[idx]
    y0 = y[idx]
    margin_x0 = (theta.dot(x0) + theta0)[0] / np.linalg.norm(theta)
    w = theta / np.linalg.norm(theta)
    plt.plot([x0[0], x0[0]-w[0]*margin_x0], [x0[1], x0[1]-w[1]*margin_x0], color='blue')
    plt.scatter([x0[0]-w[0]*margin_x0], [x0[1]-w[1]*margin_x0], color='blue')

# save the plot
plt.xlabel("x1")
plt.ylabel("x2")
plt.title("Observations and maximum-margin separating hyperplane")
plt.savefig("Observations and maximum-margin separating hyperplane")
plt.show()
```

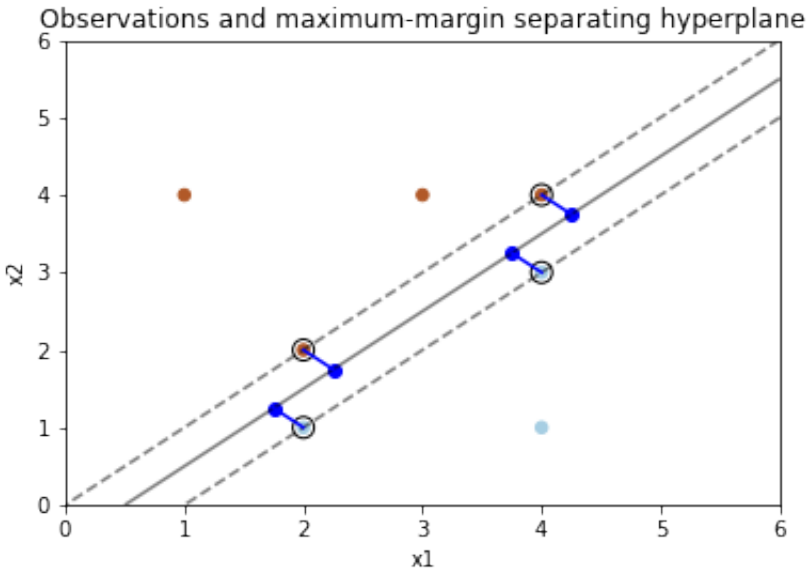


Figure 2.1: Observations and maximum-margin separating hyperplane

2.1.2 Describe the classification rule for the maximal margin classifier.

The following code yields the answer:

```
# get the separating hyperplane
w = clf.coef_[0]
a = -w[0] / w[1]
xx = np.linspace(-5, 5)
yy = a * xx - (clf.intercept_[0]) / w[1]
```

Classify to Red if $\beta_0 + \beta_1 X_1 + \beta_2 X_2 > 0$, and classify to Blue otherwise, where $\beta_0 = 0.5, \beta_1 = -1, \beta_2 = 1$.

$$r(x) = \begin{cases} Red & \text{if } 0.5 - X_1 + X_2 > 0 \\ Blue & \text{otherwise} \end{cases} \tag{2.1}$$

2.1.3 On your sketch, indicate the margin for the maximal margin hyperplane.

The margin is the distance between the solid line and the dotted line.

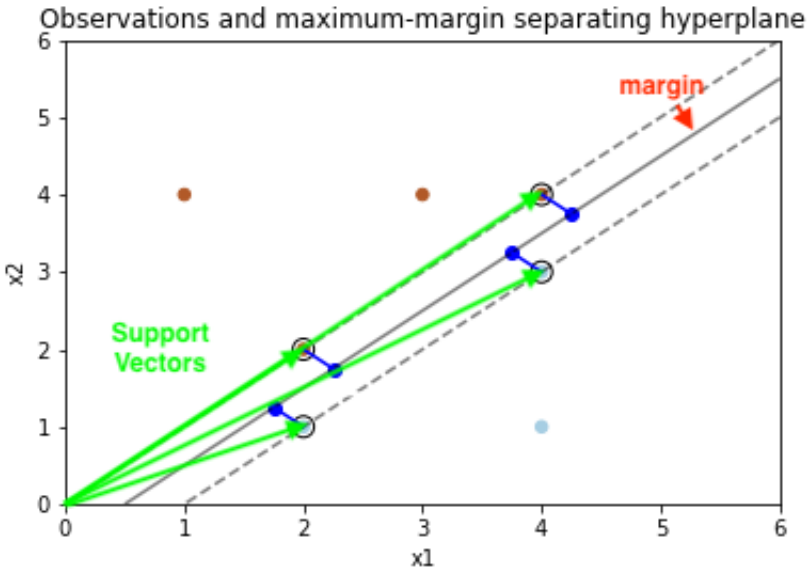


Figure 2.2: Observations and maximum-margin separating hyperplane - margin

2.1.4 Indicate the support vectors for the maximal margin classifier.

The following code snippet returns the anwser: [[2., 1.], [4., 3.], [2., 2.], [4., 4.]]

```
# get the support vectors
clf.support_vectors_
```

2.1.5 Argue that a slight movement of the seventh observation would not affect the maximal margin hyperplane.

The seventh observation (4, 1) does not contribute to the construction of support vectors. It's not close to the margin compared to the fifth and sixth observation (ie. (2, 1) and (4, 3)).

2.1.6 Sketch a hyperplane that separates the data, but is not the maximum-margin separating hyperplane. Provide the equation for this hyperplane.

```
# input the observations and their labels
X = np.array([[3,4],[2,2],[4,4],[1,4],[2,1],[4,3],[4,1]], dtype=np.int32)
y= np.array([1,1,1,1,0,0,0])
# plot the points
plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Paired)

# plot the hyperplane that separates the data
plt.xlim([0, 6])
plt.ylim([0, 6])
x = np.linspace(0, 10, 1000)
plt.plot(x, 1.2*x - 1, linestyle='-')  # solid
```

```
# save the plot
plt.xlabel("x1")
plt.ylabel("x2")
plt.title("Not maximum-margin separating hyperplane")
plt.savefig("Not maximum-margin separating hyperplane")
plt.show()
```

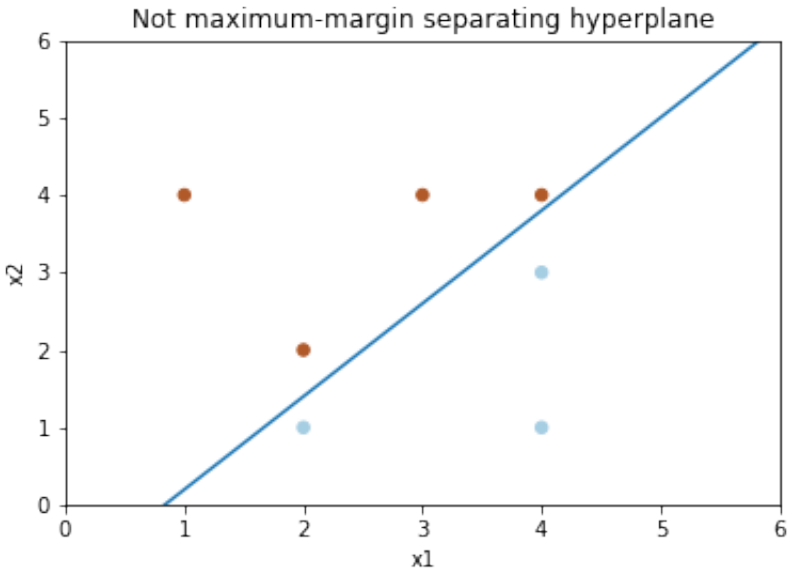


Figure 2.3: Not maximum-margin separating hyperplane

Classify to Red if $\beta_0 + \beta_1 X_1 + \beta_2 X_2 > 0$, and classify to Blue otherwise, where $\beta_0 = 1, \beta_1 = -1.2, \beta_2 = 1$.

$$r(x) = \begin{cases} Red & \text{if } 1 - 1.2X_1 + X_2 > 0 \\ Blue & \text{otherwise} \end{cases} \tag{2.2}$$

2.1.7 Draw an additional observation on the plot so that the two classes are no longer separable by a hyperplane.

We add an observation $X_1 = 2, X_2 = 3, Y = Blue$. This makes the two classes no longer separable by a hyperplane.

```
# input the observations and their labels
X = np.array([[3,4],[2,2],[4,4],[1,4],[2,1],[4,3],[4,1],[2, 3]], dtype=np.int32)
y= np.array([1,1,1,1,0,0,0,0])
# plot the points
plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Paired)

# save the plot
plt.xlabel("x1")
plt.ylabel("x2")
plt.title("No longer separable by a hyperplane")
plt.savefig("No longer separable by a hyperplane")
plt.show()
```

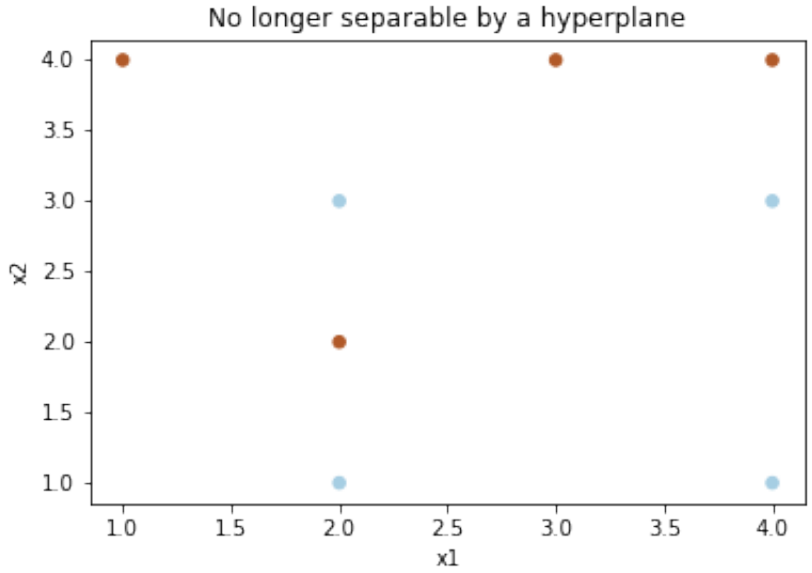


Figure 2.4: No longer separable by a hyperplane

2.2 Neural Networks as Function Approximators

First, we translate this graph into function. Then we have

$$y = \begin{cases} 0 & 0 \leq x < 1 \\ 2x - 2 & 1 \leq x < 2 \\ \frac{1}{3}x + \frac{4}{3} & 2 \leq x < 5 \\ 2x - 7 & 5 \leq x < 6 \\ -\frac{5}{3}x + 15 & 6 \leq x < 9 \\ 0 & 9 \leq x \leq 10 \end{cases} \tag{2.3}$$

We deduct the formula for the output of the ith layer.

$$y_1 = \sigma(2x - 2 - 0) = \sigma(2x - 2) \tag{2.4}$$

$$y_2 = \sigma(\frac{1}{3}x + \frac{4}{3} - 2x - 2) = \sigma(-\frac{5}{3}x + \frac{10}{3}) \tag{2.5}$$

$$y_3 = \sigma(2x - 7 - \frac{1}{3}x - \frac{4}{3}) = \sigma(\frac{5}{3}x - \frac{25}{3}) \tag{2.6}$$

$$y_4 = \sigma(-\frac{5}{3}x + 15 - 2x + 7) = \sigma(-\frac{11}{3}x + 22) \tag{2.7}$$

$$y_5 = \sigma(0 + \frac{5}{3}x - 15) = \sigma(\frac{5}{3}x - 15) \tag{2.8}$$

Then, we try to express the input function as a sum of weighted and scaled ReLU units.

$$Y = \sigma(2x-2) + \sigma(-\frac{5}{3}x + \frac{10}{3}) + \sigma(\frac{5}{3}x - \frac{25}{3}) + \sigma(-\frac{11}{3}x + 22) + \sigma(\frac{5}{3}x - 15) \tag{2.9}$$

$$= 2 \cdot \sigma(\frac{1}{2}(2x-2)) - \frac{10}{3} \cdot \sigma(-\frac{3}{10}(-\frac{5}{3}x + \frac{10}{3})) + \frac{25}{3} \cdot \sigma(\frac{3}{25}(\frac{5}{3}x - \frac{25}{3})) \tag{2.10}$$

$$-22 \cdot \sigma(-\frac{1}{22}(-\frac{11}{3}x + 22)) + 15 \cdot \sigma(\frac{1}{15}(\frac{5}{3}x - 15)) \tag{2.11}$$

$$= \sigma(2y_1) + \sigma(-\frac{10}{3}y_2) + \sigma(\frac{25}{3}y_3) + \sigma(-22y_4) + \sigma(12y_5) \tag{2.12}$$

Then we have $W_2 = \begin{bmatrix} 2 \\ -\frac{10}{3} \\ \frac{25}{3} \\ -22 \\ 15 \end{bmatrix}$, $\beta_2 = 0$.

Accordingly, we backtrack the functions for units in the hidden layers using the input function Y, weight W_2 and bias β_2. We get:

$$y_1 = \sigma(x-1) \tag{2.13}$$

$$y_2 = \sigma(\frac{1}{2}x - 1) \tag{2.14}$$

$$y_3 = \sigma(\frac{1}{5}x - 1) \tag{2.15}$$

$$y_4 = \sigma(\frac{1}{6}x - 1) \tag{2.16}$$

$$y_5 = \sigma(\frac{1}{9}x - 1) \tag{2.17}$$

Thus, we have $W_1 = \begin{bmatrix} 1 \\ \frac{1}{2} \\ \frac{1}{5} \\ \frac{1}{6} \\ \frac{1}{9} \end{bmatrix}$, $\beta_1 = -1$.

From the above deduction, we can conclude that our feed-forward neural network used to approximate the given 1-dimensional function has 1 input layer, 1 hidden layer which has 5 units within, and 1 output layer. The hidden layers, units, connections, weights, and biases are detailed as below.

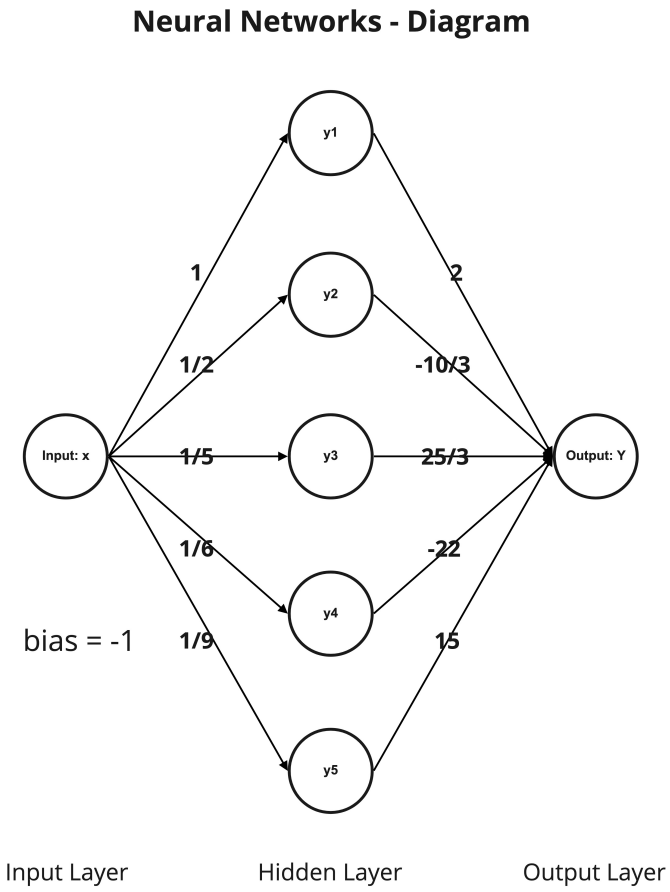


Figure 2.5: Neural Networks Diagram