Applied Machine Learning: Homework #3

Due on DUE DAT0E: Nov 10 AT 11:59PM

CLASS INSTRUCTOR: Volodymyr Kuleshov

Wentao Xu(wx225) Xingyue Dai(xd86)

PROGRAMMING EXERCISES

Problem 1

Convolutional Neural Networks
We will work with the MNIST dataset for this question. This dataset contains 60,000 images of handwritten numbers from 0 to 9 and you need to recognize the handwritten numbers by building a CNN.
You will use the Keras library for this. You might have to install the library if it hasn't been installed already. Please check the installation by printing out the version of Keras inside the python shell as follows.

```
import keras
keras . __version__
```

The above lines will give you the current version of Keras you are using. Once this works, you can proceed with the assignment.

(a) Loading Dataset For using this dataset, you will need to import mnist and use it as follows.

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

To verify that you have loaded the dataset correctly, try printing out the shape of your train and test dataset matrices. Also, try to visualize individual images in this dataset by using imshow() function in pyplot.

(b) Preprocessing The data has images with 28×28 pixel values. Since we use just one grayscale color channel, you need to reshape the matrix such that we have a $28 \times 28 \times 1$ sized matrix holding each input data-point in the training and testing dataset. The output variable can be converted into a one-hot vector by using the function to_categorical (make sure you import to_categorical from keras.utils). For example, if the output label for a given image is the digit 2, then the one-hot representation for this consists of a 10-element vector, where the element at index 2 is set to 1 and all the other elements are zero.

For preprocessing, scale the pixel values such that they lie between 0.0 and 1.0. Make sure that you use the appropriate conversion to float wherever required while scaling. You can include all these steps into a single python function that loads your dataset appropriately. Once you finish this, visualize some images using imshow() function.

(c) Implementation Now, to define a CNN model, we will use the Sequential module in Keras. We are providing you with the code for creating a simple CNN here. We useConv2D (for declaring 2D convolutional networks), MaxPooling2D (for maxpooling layer), Dense (for densely connected neural network layers) and Flatten (for flattening the input for next layer).

```
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 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 ( lr =0.01 , momentum =0.9)
# compile model
model . compile (
optimizer = opt ,
loss =' categorical_crossentropy ',
metrics =[ 'accuracy ']
)
return model
```

Specifically, we have added the following things in this code.

i. A single convolutional layer with 3×3 sized window for computing the convolution, with 32 filters

ii. Maxpooling layer with 2×2 window size.

iii. Flatten resulting features to reshape your output appropriately.

iv. Dense layer on top of this (100 neurons) with ReLU activation

v. Dense layer with 10 neurons for calculating softmax output (Our classification result will output one of the ten possible classes, corresponding to our digits)

After defining this model, we use Stochastic Gradient Descent (SGD) optimizer and cross-entropy loss to compile the model. We are using a learning rate of 0.01 and a momentum of 0.9 here. We have added this to the given code stub already. Please see that this code stub works for you. Try to print model.layers in your interactive shell to see that the model is generated as we defined.

(d) Training and Evaluating CNN Now we will train the network. You can see some examples here. Look at the fit() and evaluate() methods. You will call the fit method with a validation split of 0.1 (i.e. 10validation in every epoch). Please use 10 epochs and a batch size of 32. When you evaluate the trained model, you can call the evaluate method on the test data-set. Please report the accuracy on test data after you have trained it as above. You can refer to the following while you write code for training and evaluating your CNN.

```
model.fit ( train_x , train_y ,batch_size =32 ,epochs =10 ,validation_split =0.1)
score = model.evaluate( test_x , test_y , verbose =0)
```

(e) Experimentation

i. 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? For accessing the required values while plotting, you can store the output of the fit method while training your network. Please refer to the code below.

```
epoch_history = model . fit ( train_x , train_y , batch_size =32 ,
epochs =10 , validation_split =0.1)
# print validation and training accuracy over epochs
print ( epoch_history . history [ 'accuracy '])
print ( epoch_history . history [ ' val_accuracy '])
```

Make sure that your plot has a meaningful legend, title, and labels for X/Y axes.

ii.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

```
model.add(Dropout(0.5))
```

Make sure you import Dropout from keras.layers! Now, train this CNN for 50 epochs. Graph the validation and train accuracy after every 10 epochs. This tutorial might be helpful if you want to see more examples of dropout with Keras.

iii. 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.

iv. 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

(f) Analysis

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

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

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

Solution

(a) I loaded dataset and I printed out the shape of train and test dataset matrices and printed out the version of Keras and TensorFlow.

Below are the result:

2.4.3

2.3.1

(60000, 28, 28)

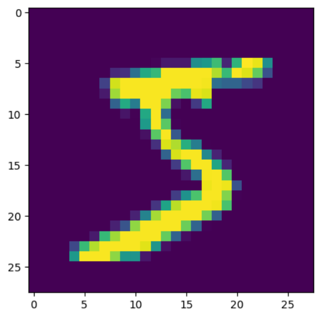
(10000, 28, 28)

```
import keras
import tensorflow as tf
import matplotlib.pyplot as plt
```

```
from keras.datasets import mnist

if __name__ == "__main__":
    (train_X, train_Y), (test_X, test_Y) = mnist.load_data()
    print(train_X.shape)
    print(test_X.shape)
```

And I also visualize individual images in this data using imshow() and I ploted one exmaple of digit 5.



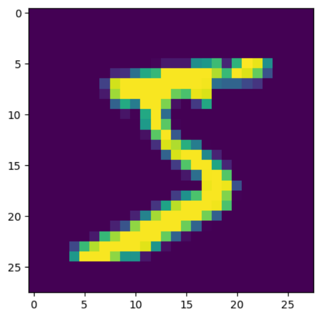
```
def show_example(data):
    plt.imshow(data[0])
    plt.savefig("./example.png")
```

(b) Below are my preprocessing codes, I reshape the matrix and transfer labels into one_hot forms. Also I rescale the pixel values such that they lie between 0.0 and 1.0.

```
from keras.utils import to_categorical
def preprocess(data, width, length):
    outdata = []
    for i in range(len(data)):
        outdata.append(data[i].reshape((width, length, -1))/255.0)
    return np.array(outdata)

def one_hot(data):
    return to_categorical(data)
```

I plotted the same digit 5:



The graph after being scaled looks almost the same.

(c) For this question, I just copied the CNN into my codes and print the layers.

```
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

def print_layers(model):
    for i in model.layers:
        print(i)
```

Below are the layers' information:

tensorflow.python.keras.layers.convolutional.Conv2D object at 0x7f877389fa58

tensorflow.python.keras.layers.pooling.MaxPooling2D object at 0x7f8755c8e2b0

tensorflow.python.keras.layers.core.Flatten object at 0x7f8755c8e4e0

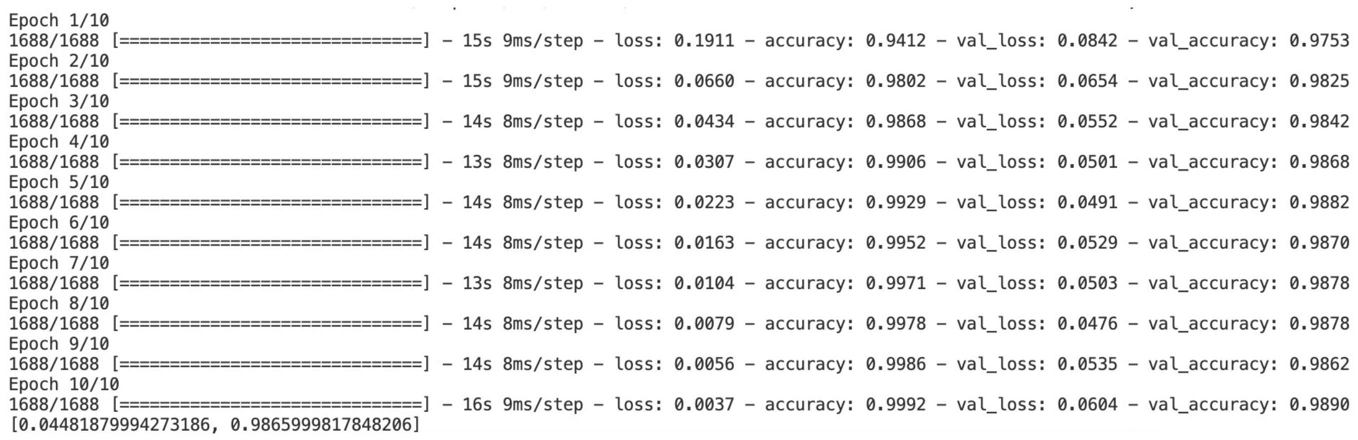
tensorflow.python.keras.layers.core.Dense object at 0x7f8755c8ef28

tensorflow.python.keras.layers.core.Dense object at 0x7f8755cc34a8

(d) Below are training and evaluating codes:

```
def train_and_evaluate(model, train_X, train_Y, test_X, test_Y):
    epoch_history = model.fit(train_X, train_Y, batch_size=32, epochs=10, validation_split=0.1)
    score = model.evaluate(test_X, test_Y, verbose=0)
    # print(epoch_history.history['accuracy'])
    print(score)
    return model
```

Below are the results:



```
Epoch 1/10
1688/1688 [==============================] - 15s 9ms/step - loss: 0.1911 - accuracy: 0.9412 - val_loss: 0.0842 - val_accuracy: 0.9753
Epoch 2/10
1688/1688 [==============================] - 15s 9ms/step - loss: 0.0660 - accuracy: 0.9802 - val_loss: 0.0654 - val_accuracy: 0.9825
Epoch 3/10
1688/1688 [==============================] - 14s 8ms/step - loss: 0.0434 - accuracy: 0.9868 - val_loss: 0.0552 - val_accuracy: 0.9842
Epoch 4/10
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0307 - accuracy: 0.9906 - val_loss: 0.0501 - val_accuracy: 0.9868
Epoch 5/10
1688/1688 [==============================] - 14s 8ms/step - loss: 0.0223 - accuracy: 0.9929 - val_loss: 0.0491 - val_accuracy: 0.9882
Epoch 6/10
1688/1688 [==============================] - 14s 8ms/step - loss: 0.0163 - accuracy: 0.9952 - val_loss: 0.0529 - val_accuracy: 0.9870
Epoch 7/10
1688/1688 [==============================] - 13s 8ms/step - loss: 0.0104 - accuracy: 0.9971 - val_loss: 0.0503 - val_accuracy: 0.9878
Epoch 8/10
1688/1688 [==============================] - 14s 8ms/step - loss: 0.0079 - accuracy: 0.9978 - val_loss: 0.0476 - val_accuracy: 0.9878
Epoch 9/10
1688/1688 [==============================] - 14s 8ms/step - loss: 0.0056 - accuracy: 0.9986 - val_loss: 0.0535 - val_accuracy: 0.9862
Epoch 10/10
1688/1688 [==============================] - 16s 9ms/step - loss: 0.0037 - accuracy: 0.9992 - val_loss: 0.0604 - val_accuracy: 0.9890
[0.04481879994273186, 0.9865999817848206]
```

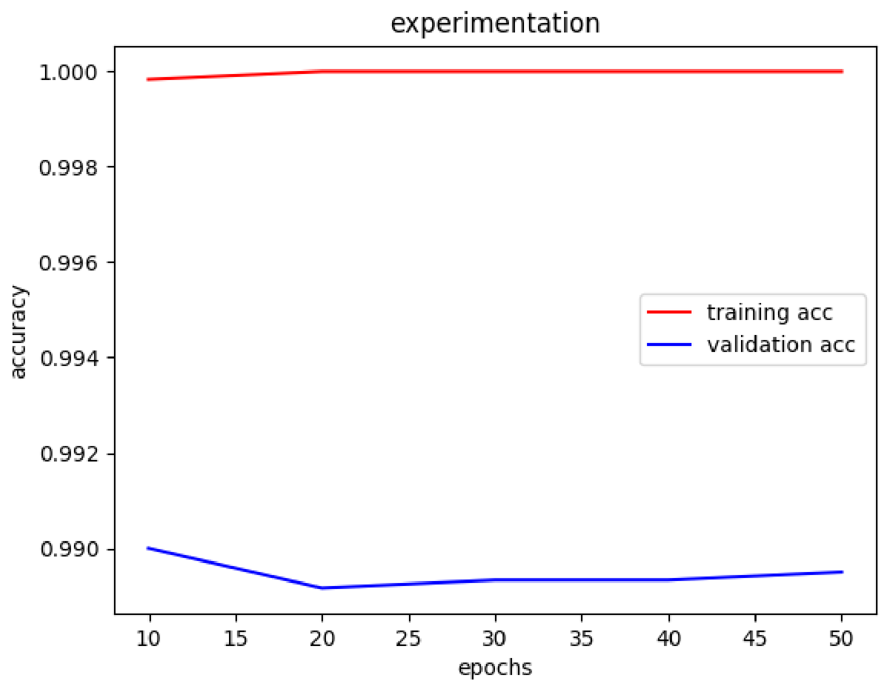
From the above results, we can see that test accuracy on test data is 0.9865.

(e) i.

Below are codes:

```
def experiment(model, train_X, train_Y, test_X, test_Y):
    training_history = []
    validation_history = []
    for i in range(5):
        epoch_history = model.fit(train_X, train_Y, batch_size=32, epochs=10, validation_split=
        score = model.evaluate(test_X, test_Y, verbose=0)
        # recode the last accuracy
        training_history.append(epoch_history.history['accuracy'][-1])
        validation_history.append(epoch_history.history['val_accuracy'][-1])
    X_axies = [10, 20, 30, 40, 50]
    plt.figure()
    l1, = plt.plot(X_axies, training_history, color='red')
    l2, = plt.plot(X_axies, validation_history, color='blue')
    plt.legend(handles=[l1,l2],labels=['training acc','validation acc'],loc='best')
    plt.xlabel('epochs')
    plt.ylabel('accuracy')
    plt.title("experimentation")
    plt.savefig("./experimentation.png")
```

Below is the graph:



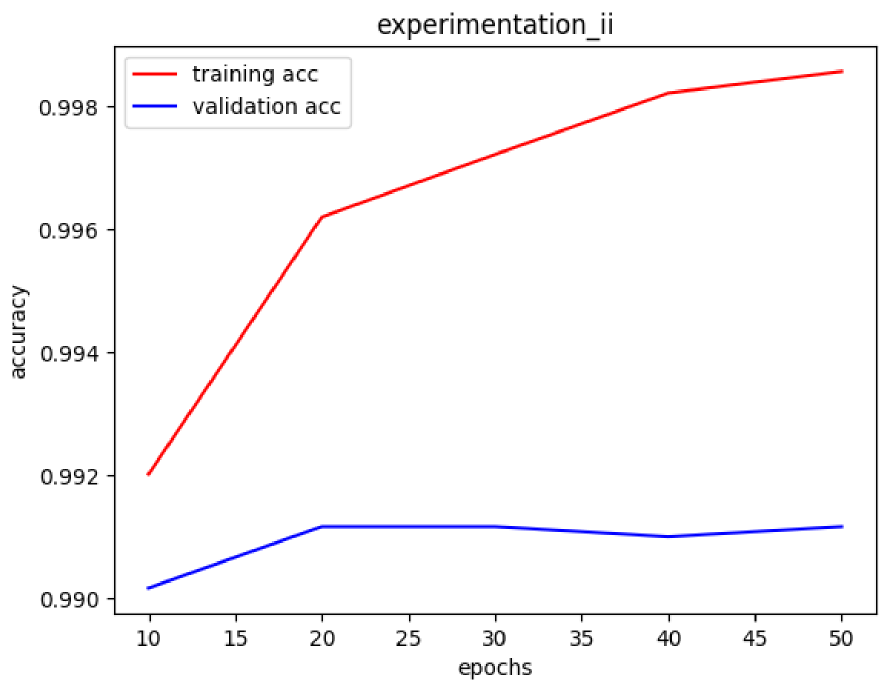
We can see that the training accuracy became 1.0 after 20 epochs and keep being 1.0. The validation accuracy first became smaller and then became larger. So there is no steady improvement for validation, and it seems there is a steady improvement for training accuracy but when the training accuracy became 1, it remains 1.

ii.

I added ’drop out’ before the dense layer with 100 neurons. Below are codes:

```
model.add(MaxPooling2D((2 , 2)))
# Flatten output
model.add(Flatten())
# Dense layer of 100 neurons
model.add(Dropout(0.5))
model.add(
        Dense(100 ,
        activation = 'relu',
        kernel_initializer = 'he_uniform')
        )
```

Below is the graph:



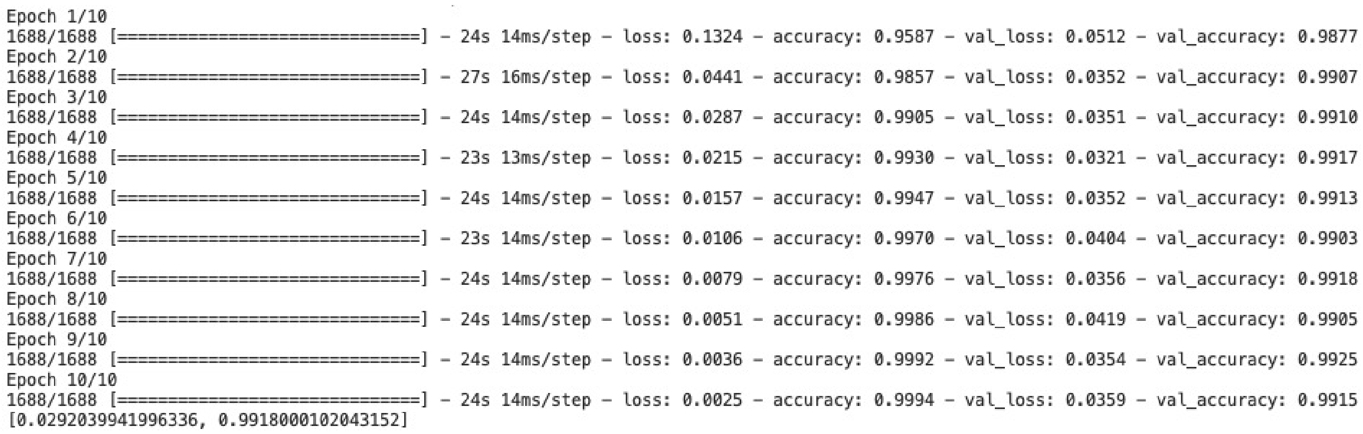
iii:

Below are models with 2 CNN layers and maxpooling layers.

```
def modified_cnn():
    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)))
    model.add(
            Conv2D(64, (3, 3),
            activation = 'relu',
            kernel_initializer = 'he_uniform',
            input_shape = (13, 13, 32))
            )
    model.add(MaxPooling2D((2 , 2)))
    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 set the num of filters to be 64 here. And below is the train results.



```
Epoch 1/10
1688/1688 [==============================] - 24s 14ms/step - loss: 0.1324 - accuracy: 0.9587 - val_loss: 0.0512 - val_accuracy: 0.9877
Epoch 2/10
1688/1688 [==============================] - 27s 16ms/step - loss: 0.0441 - accuracy: 0.9857 - val_loss: 0.0352 - val_accuracy: 0.9907
Epoch 3/10
1688/1688 [==============================] - 24s 14ms/step - loss: 0.0287 - accuracy: 0.9905 - val_loss: 0.0351 - val_accuracy: 0.9910
Epoch 4/10
1688/1688 [==============================] - 23s 13ms/step - loss: 0.0215 - accuracy: 0.9930 - val_loss: 0.0321 - val_accuracy: 0.9917
Epoch 5/10
1688/1688 [==============================] - 24s 14ms/step - loss: 0.0157 - accuracy: 0.9947 - val_loss: 0.0352 - val_accuracy: 0.9913
Epoch 6/10
1688/1688 [==============================] - 23s 14ms/step - loss: 0.0106 - accuracy: 0.9970 - val_loss: 0.0404 - val_accuracy: 0.9903
Epoch 7/10
1688/1688 [==============================] - 24s 14ms/step - loss: 0.0079 - accuracy: 0.9976 - val_loss: 0.0356 - val_accuracy: 0.9918
Epoch 8/10
1688/1688 [==============================] - 24s 14ms/step - loss: 0.0051 - accuracy: 0.9986 - val_loss: 0.0419 - val_accuracy: 0.9905
Epoch 9/10
1688/1688 [==============================] - 24s 14ms/step - loss: 0.0036 - accuracy: 0.9992 - val_loss: 0.0354 - val_accuracy: 0.9925
Epoch 10/10
1688/1688 [==============================] - 24s 14ms/step - loss: 0.0025 - accuracy: 0.9994 - val_loss: 0.0359 - val_accuracy: 0.9915
[0.0292039941996336, 0.9918000102043152]
```

We can see that the test accuracy is 0.9918.

iv.

The test accuracy of training result of learning rate of 0.001 is 0.9886:

[0.03320832550525665, 0.9886999726295471]

The test accuracy of training result of learning rate of 0.1 is 0.1135:

[2.304381847381592, 0.11349999904632568]

(f) i.

Validation accuracy changes to being larger after adding dropout, while training accuracy changes to become not as large as 1.0 after we add dropout. The reason for this is that "dropout" layer can prevent the model from being overfitting, the dropout layer will randomly set input units to 0 with a frequency of rate at each step during training time, Inputs not set to 0 are scaled up by 1/(1 - rate) such that the sum over all inputs is unchanged, which prevent the model from being overfitting, that's why training accuracy became not as large as 1.0 while validation accuarcy became larger.

ii.

The performance has been improved from the result. The test accuracy was 0.9865 for the single layer CNN model, while the test accuracy was 0.9918 for the 2 CNN layers model. And the loss value has also been decreased to 0.029 from 0.0448.

iii.

When the learning rate is 0.01, the performance of the model is the better than the conditions with learning rate 0.001 and 0.1. And the performance of learning rate with 0.001 is better the learning rate with 0.1. The learning rate of 0.1 seems to make the model become random-guessing model, since the test accuracy is 0.11, while the training speed for the model of learning rate of 0.001 is slow.

Problem 2

Random forests for image approximation In this question, you will use random forest regression to approximate an image by learning a function, $f : R^2 -> R$, that takes image (x, y) coordinates as input and outputs pixel brightness. This way, the function learns to approximate areas of the image that it has not seen before.

(a) Start with an image of the Mona Lisa. If you don't like the Mona Lisa, pick another interesting image of your choice.

(b) Preprocessing the input. To build your "training set," uniformly sample 5,000 random (x, y) coordinate locations.

• What other preprocessing steps are necessary for random forests inputs? Describe them, implement them, and justify your decisions. In particular, do you need to perform mean subtraction, standardization, or unit-normalization?

(c) Preprocessing the output. Sample pixel values at each of the given coordinate locations. Each pixel contains red, green, and blue intensity values, so decide how you want to handle this. There are several options available to you: • Convert the image to grayscale • Regress all three values at once, so your function maps (x, y) coordinates to (r, g ,b) values: $f : R^2 -> R^3$ • Learn a different function for each channel, $f_{Red} : R^2 -> R$, and likewise for f_{Green}, f_{Blue} . Note that you may need to rescale the pixel intensities to lie between 0.0 and 1.0. (The default for pixel values may be between 0 and 255, but your image library may have different defaults.) What other preprocessing steps are necessary for random regression forest outputs? Describe them, implement them, and justify your decisions.

(d) To build the final image, for each pixel of the output, feed the pixel coordinate through the random forest and color the resulting pixel with the output prediction. You can then use imshow to view the result. (If you are using grayscale, try imshow(Y, cmap='gray') to avoid fake-coloring). You may use any implementation of random forests, but you should understand the implementation and you must cite your sources.

(e) Experimentation

i. 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.

ii. 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.

iii. 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?

iv. Experiment with different pruning strategies of your choice.

(f) Analysis.

i. 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 of the trained decision trees inside the forest. Feel free to define any variables you need.

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

iii. Straightforward: How many patches of color may be in the resulting image if the forest contains a single decision tree? Define any variables you need. iv. Tricky: How many patches of color might be in the resulting image if the forest contains n decision trees? Define any variables you need.

Solution

(a) I chose Mona Lisa picture.

(b) Here I randomly choose 5000 coordinate locations,and I didn't scale the coordinate locations, but I scale their R,G,B to 0.0 to 1.0, which will be helpful for models to process, and I didn't do other data preprocessing, since random forest doesn't need much data preprocessing.

Below are codes:

```
def preprocess(data):
    i = 0
    chosen = set()
    total = data.shape[0] * data.shape[1]
    sample = []
    while(i < 5000):
        index = int(random.random() * total)
        while (index in chosen):
            index = int(random.random() * total)
        chosen.add(index)
        i += 1
    sample_y = []
    for i in chosen:
        w = i//data.shape[1]
        h = i%data.shape[1]
        sample.append([w, h])
        sample_y.append(data[w][h]/255.0)
    sample_data = np.array(sample)
    sample_y = np.array(sample_y)
    return sample_data, sample_y
```

(c) I chose the third way, which is learn a different function for each channel, $f_{Red} : R^2 -> R$, and likewise for f_{Green}, f_{Blue} . I also rescale the pixel intensities to lie between 0.0 and 1.0. (From the slides, not much preprocessing steps are required for random regression forest outputs, which is a good point of random regression forest.) The only thing I do to process the output of random forest regressor is to multiply every density of R,G,B with 255 and then plot the result(since there is no much processing required to process the output of the random forest).

```
def preprocess(data):
    i = 0
    chosen = set()
    total = data.shape[0] * data.shape[1]
    sample = []
    while(i < 5000):
        index = int(random.random() * total)
        while (index in chosen):
            index = int(random.random() * total)
        chosen.add(index)
        i += 1
    sample_y = []
    for i in chosen:
        w = i//data.shape[1]
```

```
            h = i%data.shape[1]
            sample.append([w, h])
            sample_y.append(data[w][h]/255.0)
        sample_data = np.array(sample)
        sample_y = np.array(sample_y)
        return sample_data, sample_y

    def preprocess_label(data):
        r = []
        g = []
        b = []
        for i in range(len(data)):
            r.append(data[i][0])
            g.append(data[i][1])
            b.append(data[i][2])
        return np.array(r), np.array(g), np.array(b)

    # construct the model
    train_X, train_Y = preprocess(img)
    # print(train_X.shape)
    r, g, b = preprocess_label(train_Y)
    f_red = create_RF()
    f_green = create_RF()
    f_blue = create_RF()
    f_red.fit(train_X, r)
    f_green.fit(train_X, g)
    f_blue.fit(train_X, b)
    models = [f_red, f_green, f_blue]

    # build_image(models, img, train_X, train_Y)

    # process the result
    pic[w][h]=[int(r[i]*255), int(g[i]*255), int(b[i]*255)]
```

(d) I used randomforestRegressor from sklearn.

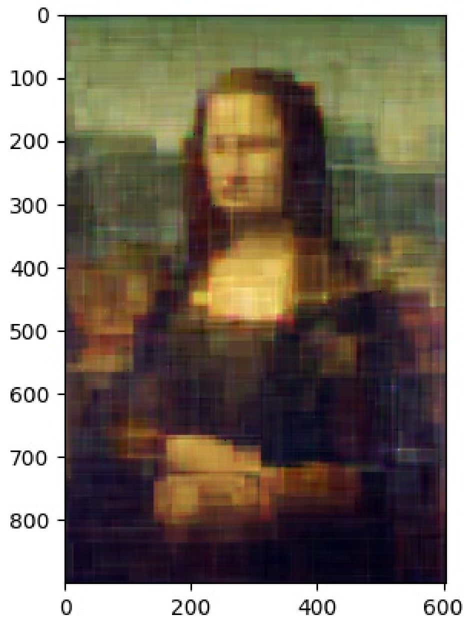
Below are codes of building the final image:

```
    def create_RF(n = 10):
        clf = RandomForestRegressor(n_estimators = n)
        return clf

    def build_image(models, data, train_X, train_Y):
        p_data = []
        t_tain = train_X.tolist()
        for i in range(data.shape[0]):
            for j in range(data.shape[1]):
                temp = [i,j]
                if temp in t_tain:
                    continue
                p_data.append([i, j])
```

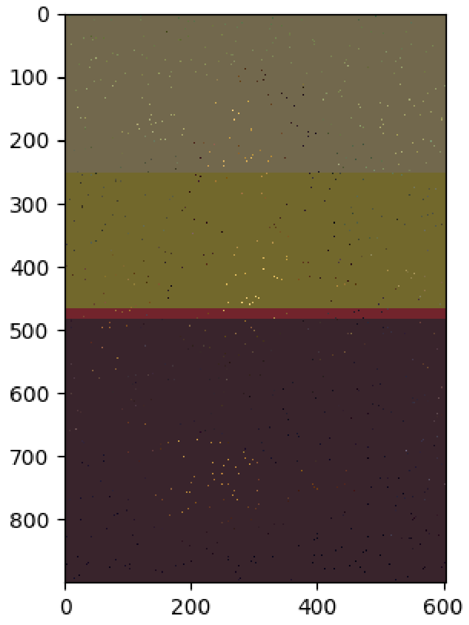
```
p_data = np.array(p_data)
print(p_data.shape)
r = models[0].predict(p_data)
g = models[1].predict(p_data)
b = models[2].predict(p_data)
pic = []
t_p_data = p_data.tolist()
print(type(train_Y))
train_Y = train_Y.tolist()
for i in range(data.shape[0]):
    # print(i)
    pic.append([0] * data.shape[1])
for i in range(len(t_p_data)):
    # print(i)
    w = t_p_data[i][0]
    h = t_p_data[i][1]
    pic[w][h]=[int(r[i]*255), int(g[i]*255), int(b[i]*255)]
for i in range(len(train_X)):
    # print(i)
    w = train_X[i][0]
    h = train_X[i][1]
    pic[w][h] = [int(train_Y[i][0]*255), int(train_Y[i][1]*255), int(train_Y[i][2]*
pic = np.array(pic)
print(pic.shape)
pic = np.array(pic)
fig = plt.figure()
plt.imshow(pic)
plt.show()
fig.savefig("./mo.jpg")
```

And below is the final image I build:

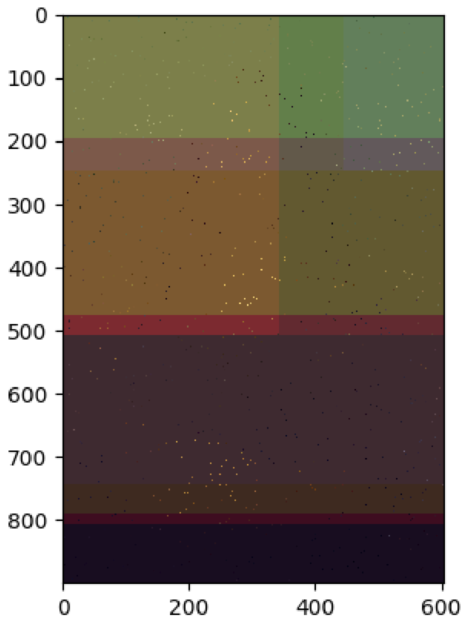


(e) i.

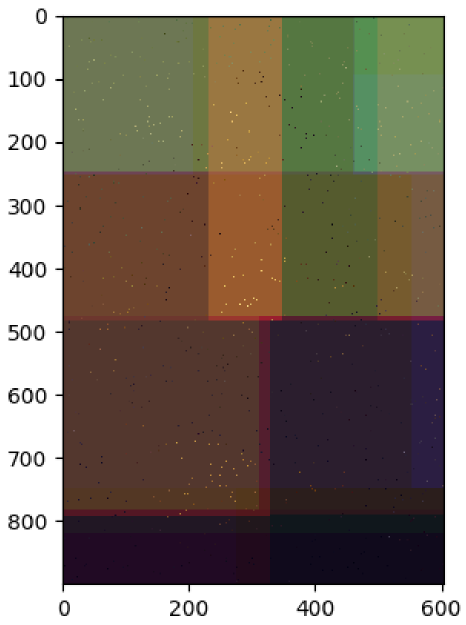
The image of depth 1:



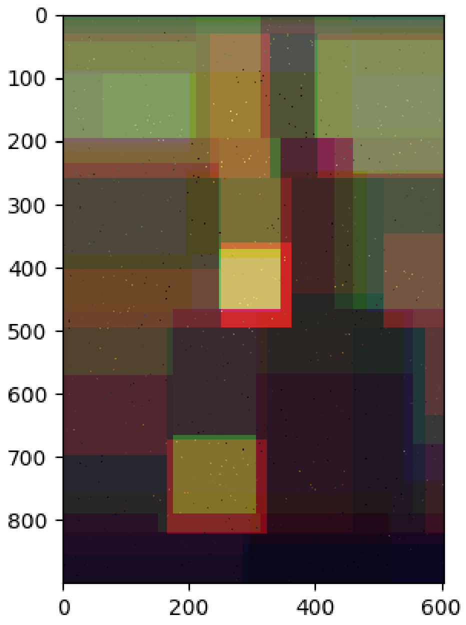
The image of depth 2:



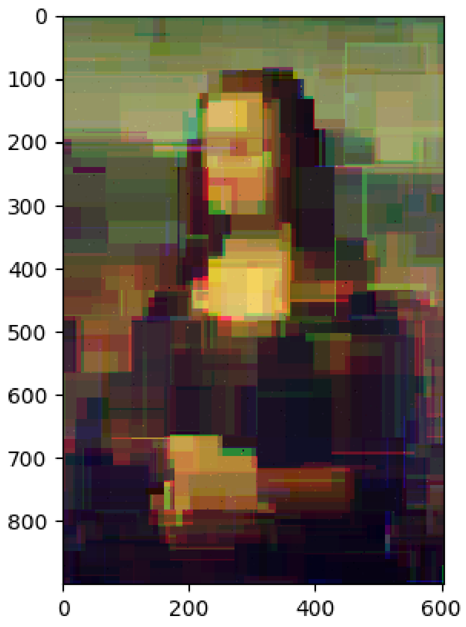
The image of depth 3:



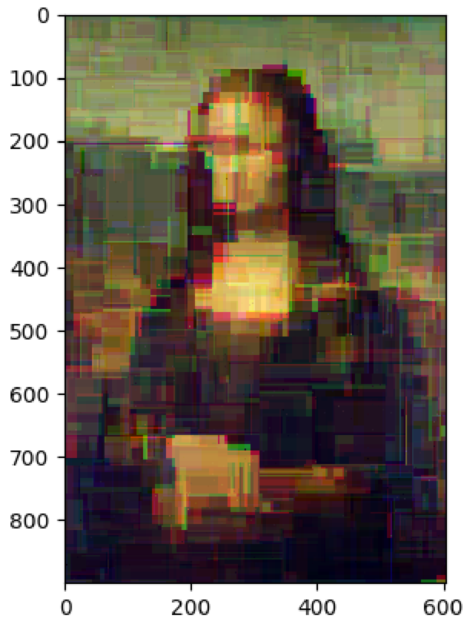
The image of depth 5:



The image of depth 10:



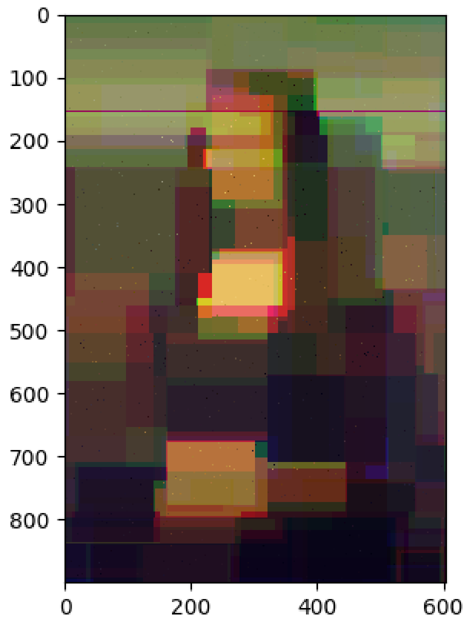
The image of depth 15:



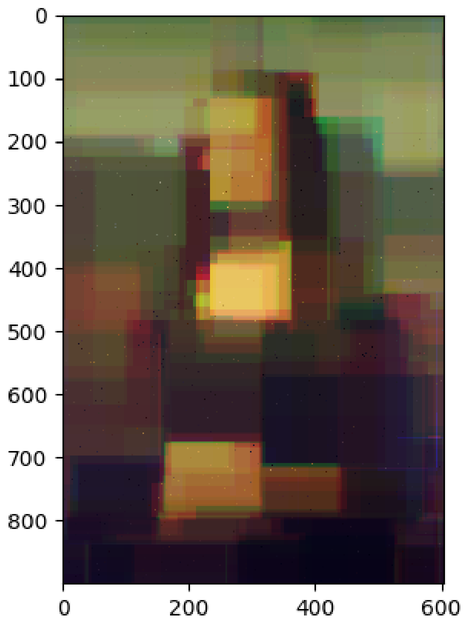
From the above result, we can see that the with the increment of depths, the image became clearer and clearer, and more details of the image have been generated. The larger the depths become, the more divisions(partitions of the image, which corresponds to the patches of colors) can be generated, which generates more details of the image.

ii.

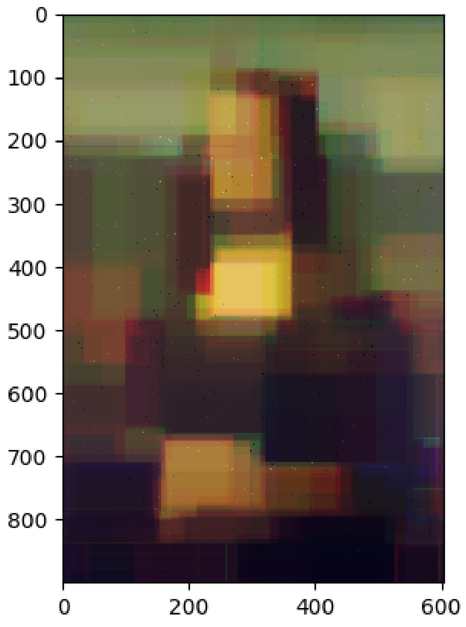
The image of num 1:



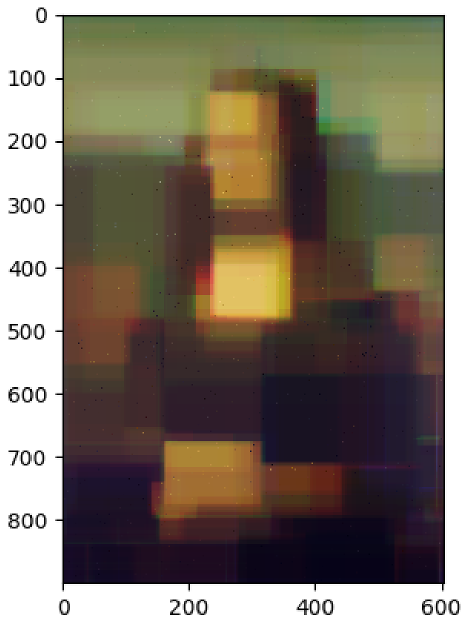
The image of num 3:



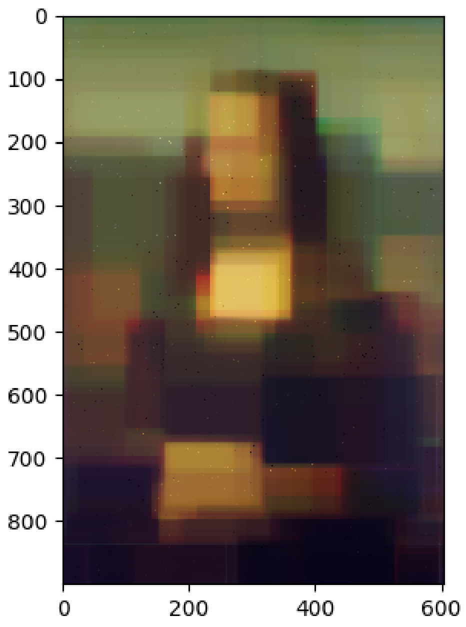
The image of num 5:



The image of num 10:



The image of num 100:



From the above result, we can see that with the increment of trees, the image just became slightly clearer, the result is not as good as the result generated by deeper randomforestRegressor. Not as many details generated by the model with deeper depth, the model with more trees just reach the better result of the depth of 7 here, so even the performance of the model with more trees but with the same depth has been improved when the number of trees is being increased, still there will not be many divisions(partitions of the image, which corresponds to the patches of colors) generated by the model, so there are not so many details in the image even with 100 trees.

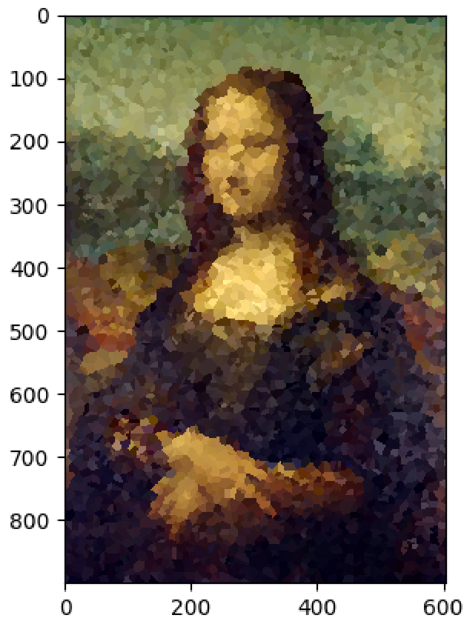
iii.

The codes:

```
def KNN_model():
    clf = KNeighborsRegressor(n_neighbors=1)
    return clf

def experimentation_iii(data, train_X, train_Y, r, g, b):
    f_red = KNN_model()
    f_green = KNN_model()
    f_blue = KNN_model()
    f_red.fit(train_X, r)
    f_green.fit(train_X, g)
    f_blue.fit(train_X, b)
    models = [f_red, f_green, f_blue]
    pic_name = "./KNN.png"
    exp_image(models, data, train_X, train_Y, pic_name)
```

The graph:



We can see from the result that the image is different from the image generated from decision tree in their outlooks. Since the divisions of KNN are generated by K nearest neighbor methods, the shape of every division(partitions of the image, which corresponds to the patches of colors) is different and in irregular shape. While the divisions(partitions of the image, which corresponds to the patches of colors) in image generated by randomForestRegressor are in the rectangle shape, which is due to the mechanic of decision tree. The decision tree use hyper-planes which are parallel or vertical to the axis and each other(hyper-plane), so the divisions(partitions of the image, which corresponds to the patches of colors) in the image generated by randomForestRegressor are in the rectangle shape.

iv.

Here I found that there are different pruning strategies in the slides, such as ID3, C4.5, C5.0. While it seems not appropriate to apply them on the random forest Regressor, so here I just change some parameters which made the performance of the model be better(which includes the parameter ccp_alpha, which is used to do the pruning work as described in the official guide).

So here, my pruning strategy is that we can first fix the number of estimators, and we can change the depth, when it generates a very good performance, then we fix this depth, change the number of estimators.

And here I used "score" function of the random Forest to decide whether the model is good or not.

Below is the code:

```
p_data = []
t_tain = train_X.tolist()
for i in range(data.shape[0]):
    for j in range(data.shape[1]):
        temp = [i,j]
        if temp in t_tain:
            continue
        p_data.append([i, j])
p_data = np.array(p_data)
n_estimators = 1
depth = [1,3, 5,10,15, 20, 25]
for i in depth:
    f_red = RandomForestRegressor(n_estimators = n_estimators, max_depth=i)
    f_green = RandomForestRegressor(n_estimators = n_estimators, max_depth=i)
    f_blue = RandomForestRegressor(n_estimators = n_estimators, max_depth=i)
    f_red.fit(train_X, r)
    f_green.fit(train_X, g)
    f_blue.fit(train_X, b)
    print("score:")
    print(f_red.score(train_X, r))
    print(f_green.score(train_X, g))
    print(f_blue.score(train_X, b))
```

From the result, we can see that

score:

0.2840453184453462

0.39136939473100885

0.29616217937868705

score:

0.45011550381665966

0.5425688037307513

0.4310845085767486

score:

0.6538688215613997

0.6964134311782813

0.5317384385947259

score:

0.8941584816064597

0.8902707552349705

0.7419484245148276

score:

0.9091414670200164

0.9196397345171541

0.8034971608504872

score:

0.9206428700784153

0.9258759880858126

0.8167613234980047

score:

0.9085802742403891

0.9251875185029771

0.8217174163982514

From the above result, we can see that the depth with 20 is relatively better, so we choose depth with 20. And then we change the number of estimators.

Below are codes:

```
n_estimators = [1, 5, 10, 20, 30, 100, 120, 150]
for i in n_estimators:
    f_red = RandomForestRegressor(n_estimators = i, max_depth=20)
    f_green = RandomForestRegressor(n_estimators = i, max_depth=20)
    f_blue = RandomForestRegressor(n_estimators = i, max_depth=20)
    f_red.fit(train_X, r)
    f_green.fit(train_X, g)
    f_blue.fit(train_X, b)
    print("score:")
    print(f_red.score(train_X, r))
    print(f_green.score(train_X, g))
    print(f_blue.score(train_X, b))
```

From the result, I find that when the number of estimators is approximately equal to 100, the performance is good, so I choose depth with 20 and the number of estimators with 100.

Here I fix depth with 20 and estimators with 100, and then I modified the parameter ”ccp_alpha”, which is used to do the pruning as described in scikit-learn.org. Here I set it from 0.0 to 0.1.

Below are codes:

```
ccp = 0.0
while (ccp < 0.1):
    f_red = RandomForestRegressor(n_estimators = 100, max_depth=20, ccp_alpha=ccp)
    f_green = RandomForestRegressor(n_estimators = 100, max_depth=20, ccp_alpha=ccp)
    f_blue = RandomForestRegressor(n_estimators = 100, max_depth=20, ccp_alpha=ccp)
    f_red.fit(train_X, r)
    f_green.fit(train_X, g)
    f_blue.fit(train_X, b)
    print("score:")
    print(f_red.score(train_X, r))
    print(f_green.score(train_X, g))
    print(f_blue.score(train_X, b))
    ccp += 0.02
```

And from the result, I found that 0.0 is the best, so finally the parameter of ccp_alpha is 0.0. So above are the strategies I took to do the pruning.

(f) i.

Here x is the coordinate (x_0, x_1), and the output of the model is the color value.

We have a decision tree regressor, and the decision rule r: $X -> true, false$ is a partition of the feature space into two disjoint regions:

$$r(x) = \begin{cases} true\ if\ x_i \geq C \\ false\ if\ x_i < C \end{cases}$$

Here C is generated by learning the decision rules, for continuous condition, we have the following optimization problem:

$$min_{r \in u}(L((x,y) \in D | r(x) = T) + L((x,y) \in D | r(x) = F))$$

where L is a loss function over a subset of the data flagged by the rule and u is the set of possible rule, we need to minimize it and get the rule, and then we can do the partition.

The above formula and process also applies to the split point at the root node for one of the trained decision trees inside the forest. We split the nodes into two nodes in each layer, and then we reach the max_depth and we get the leaves.

ii.

The shapes of patches of color generated by KNN regressor are in irregular shape, the shapes of patches of color generated by randomforestRegressor are in rectangle shapes.

We can see from the result that the image of KNN is different from the image generated from decision tree in their outlooks. Since the divisions of KNN are generated by K nearest neighbor methods, the shape of every division(partitions of the image, which corresponds to the patches of colors) is different and in irregular shape. While the divisions(partitions of the image, which corresponds to the patches of colors) in image generated by randomForestRegressor are in the rectangle shape, which is due to the mechanic of decision tree. The decision tree use hyper-plane which are parallel or vertical to the axis and each other(hyper-plane), so the divisions(partitions of the image, which corresponds to the patches of colors) in the image generated by randomForestRegressor are in the rectangle shape.

Also, the number of divisions(partitions of the image, which corresponds to the patches of colors) of the image generated by random forest is related to the depth of random forest.

iii.

We first define the number of depth to be N_{depth}, and we here add an assumption, the color is decided by one value, not three-channel values. So the number of patches of color may be at most $2^{N_{depth}}$, since the random forest with one tree will generate $2^{N_{depth}}$ leaves, which corresponds to $2^{N_{depth}}$ divisions of the image, which corresponds to at most $2^{N_{depth}}$ patches of colors.

iv.

We first define the number of depth to be N_{depth}, and we here add an assumption, the color is decided by one value, not three-channel values. Here we have n trees, and every tree has the depth of N_{depth}, every tree has $2^{N_{depth}}$ leaves.

Since for the final result, the random forest need to get the average prediction of n decision trees, so we can have $2^{N_{depth}} * 2^{N_{depth}} * 2^{N_{depth}} * ...$, so there are $(2^{N_{depth}})^n$ choices of a prediction, which means at most there will be $(2^{N_{depth}})^n$ divisions(partitions), which corresponds $(2^{N_{depth}})^n$ patches of colors.

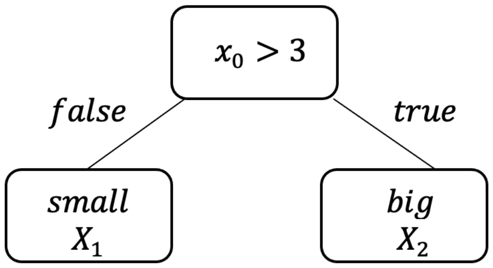
WRITTEN EXERCISES

Problem 3

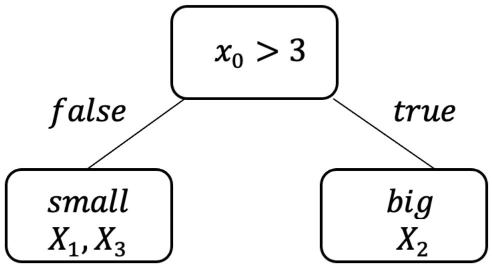
Solution

(a) i. A.

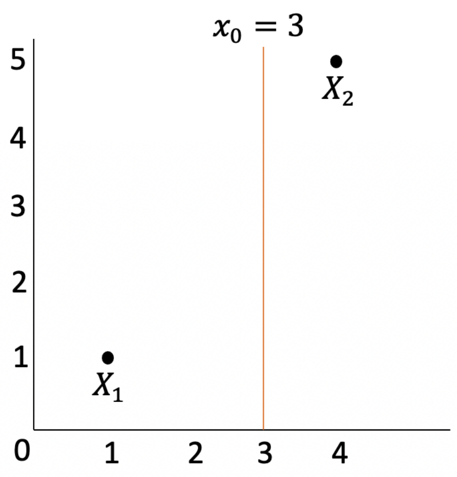
No, $X^1$and X^2 are separable as long as there exists a decision rule as with a decision rule, there will always be a partition that separates these two points into two disjoint regions. For example, let $X^1 = (1, 1)$ and $X^2 = (4, 5)$ and let the decision rule be if $X_0 > 3$ for point X^i then the decision stump classifies the point X^i as big, otherwise it's small as shown below



But if adding the third point, then it's possible that these three different points are not separable by a decision stump. Because with only one decision rule we will only be able to create two disjoint regions, which means at least two of theses three different points will be in the same category. So they are not separable by a decision stump.Adding a new point $X^3 = (2, 1)$ to the same example above, then we can illustrate this case with the following graph.

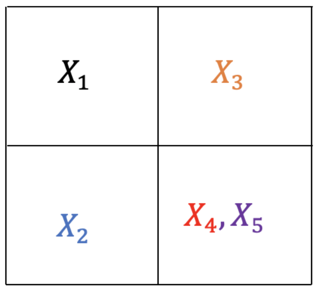


A linear classifier can separate two different points but not three. Below is the illustration of the aforementioned example with two points.



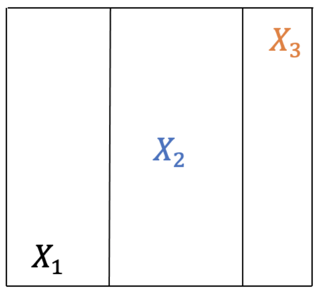
i. B.

Yes,it's possible that adding more points to the first example such they won't be separable by a two-level decision tree,because each decision rule creates only two disjoint regions, a two-level decision tree can only create four disjoint regions such that if there are points with more than four classes, they won't be separable by such a decision tree. Take five points with different classes as an example, then we have



i. C.

yes, consider three different points with the construction as shown below



then apparently these three points are separable by a two-level decision tree but not a linear classifier

ii. Using Entropy (information gain) to determine the preferred attribute, let E be the entropy function,I be the function of information gained then we have

E(Preference) $= -\frac{5}{11}log_2(\frac{5}{11}) - \frac{6}{11}log_2(\frac{6}{11}) \approx 0.994$

for attribute Location

E(Location=Rural) $= -\frac{1}{2}log_2(\frac{1}{2}) - \frac{1}{2}log_2(\frac{1}{2}) = 1$

E(Location=Urban) $= -\frac{1}{2}log_2(\frac{1}{2}) - \frac{1}{2}log_2(\frac{1}{2}) = 1$

E(Location=semi-rural) $= -\frac{2}{3}log_2(\frac{2}{3}) - \frac{1}{3}log_2(\frac{1}{3}) \approx 0.918$

Then the weighted average for location is

E(Location) $= \frac{4}{11} + \frac{4}{11} + \frac{3}{11} \times 0.918 \approx 0.978$

Information gained

I(Preference, Location) = E(preference)-E(Location)=0.994-0.978=0.016

for attribute Pollution

E(Pollution=low) $= -\frac{2}{3}log_2(\frac{2}{3}) - \frac{1}{3}log_2(\frac{1}{3}) \approx 0.918$

E(Pollution=med) $= -\frac{3}{4}log_2(\frac{3}{4}) - \frac{1}{4}log_2(\frac{1}{4}) \approx 0.811$

E(Pollution=high) $= -\frac{3}{4}log_2(\frac{3}{4}) - \frac{1}{4}log_2(\frac{1}{4}) \approx 0.811$

Then the weighted average for Pollution is

E(Pollution) $= \frac{3}{11} \times 0.918 + \frac{8}{11} \times 0.811 \approx 0.84$

Information gained

I(Preference, Pollution) = E(preference)-E(Pollution)=0.994-0.84=0.154

for attribute Area

E(Area=large) $= -\frac{3}{5}log_2(\frac{3}{5}) - \frac{2}{5}log_2(\frac{2}{5}) \approx 0.971$

E(Area=small) $= -\frac{1}{2}log_2(\frac{1}{2}) - \frac{1}{2}log_2(\frac{1}{2}) = 1$

Then the weighted average for Area is

E(Area) $= \frac{5}{11} \times 0.971 + \frac{6}{11} \times 1 \approx 0.987$

Information gained

I(Preference, Area) = E(preference)-E(Area)=0.994-0.987=0.007

for attribute Windows

E(Windows=large) $= -\frac{4}{5}log_2(\frac{4}{5}) - \frac{1}{5}log_2(\frac{1}{5}) \approx 0.722$

E(Windows=small) $= -\frac{2}{3}log_2(\frac{2}{3}) - \frac{1}{3}log_2(\frac{1}{3}) = 0.918$

Then the weighted average for Windows is

E(Windows) $= \frac{5}{11} \times 0.722 + \frac{6}{11} \times 0.918 \approx 0.829$

Information gained

I(Preference, Windows) = E(preference)-E(Windows)=0.994-0.829=0.165

Because we will gain most information choosing attribute Windows, we should choose Windows at the root of this decision tree.

Using Gini Index to determine the preferred attribute, let G(x) be the Gini index of x, then we have for attribute Location

P(Location=Rural)=$\frac{4}{11}$

P(Location=Rural,Preference=yes)=$\frac{1}{2}$

P(Location=Rural,Preference=no)=$\frac{1}{2}$

G(Location=Rural)=$1 - (\frac{1}{2}^2 + \frac{1}{2}^2) = \frac{1}{2}$

P(Location=Urban)=$\frac{4}{11}$

P(Location=Urban,Preference=yes)=$\frac{1}{2}$

P(Location=Urban,Preference=no)=$\frac{1}{2}$

G(Location=Urban)=$1 - (\frac{1}{2}^2 + \frac{1}{2}^2) = \frac{1}{2}$

P(Location=semi-rural)=$\frac{3}{11}$

P(Location=semi-rural,Preference=yes)=$\frac{2}{3}$

P(Location=semi-rural,Preference=no)=$\frac{1}{3}$

G(Location=semi-rural)=$1 - (\frac{2}{3}^2 + \frac{1}{3}^2) = \frac{4}{9}$

the weighted sum of the Gini indices for location is

G(Location)=$\frac{4}{11} \times \frac{1}{2} + \frac{4}{11} \times \frac{1}{2} + \frac{3}{11} \times \frac{4}{9} \approx 0.485$

for attribute Pollution

P(Pollution=low)=$\frac{3}{11}$

P(Pollution=low,Preference=yes)=$\frac{2}{3}$

P(Pollution=low,Preference=no)=$\frac{1}{3}$

G(Pollution=low)=$1 - (\frac{2}{3}^2 + \frac{1}{3}^2) = \frac{4}{9}$

P(Pollution=med)=$\frac{4}{11}$

P(Pollution=med,Preference=yes)=$\frac{3}{4}$

P(Pollution=med,Preference=no)=$\frac{1}{4}$

G(Pollution=med)=$1 - (\frac{3}{4}^2 + \frac{1}{4}^2) = \frac{3}{8}$

P(Pollution=high)=$\frac{4}{11}$

P(Pollution=high,Preference=yes)=$\frac{1}{4}$

P(Pollution=high,Preference=no)=$\frac{3}{4}$

G(Pollution=high)=$1 - (\frac{1}{4}^2 + \frac{3}{4}^2) = \frac{3}{8}$

the weighted sum of the Gini indices for Pollution is

G(Pollution)=$\frac{3}{11} \times \frac{4}{9} + \frac{4}{11} \times \frac{3}{8} + \frac{4}{11} \times \frac{3}{8} \approx 0.394$

for attribute Area

P(Area=large)=$\frac{5}{11}$

P(Area=large,Preference=yes)=$\frac{3}{5}$

P(Area=large,Preference=no)=$\frac{2}{5}$

G(Area=large)=$1 - (\frac{3}{5}^2 + \frac{2}{5}^2) = \frac{12}{25}$

P(Area=small)=$\frac{6}{11}$

P(Area=small,Preference=yes)=$\frac{1}{2}$

P(Area=small,Preference=no)=$\frac{1}{2}$

G(Area=small)=$1-(\frac{1}{2}^2+\frac{1}{2}^2)=\frac{1}{2}$

the weighted sum of the Gini indices for Area is

G(Area)=$\frac{5}{11}\times\frac{12}{25}+\frac{6}{11}\times\frac{1}{2}\approx 0.491$

for attribute Windows

P(Windows=large)=$\frac{5}{11}$

P(Windows=large,Preference=yes)=$\frac{1}{5}$

P(Windows=large,Preference=no)=$\frac{4}{5}$

G(Windows=large)=$1-(\frac{1}{5}^2+\frac{4}{5}^2)=\frac{8}{25}$

P(Windows=small)=$\frac{6}{11}$

P(Windows=small,Preference=yes)=$\frac{2}{3}$

P(Windows=small,Preference=no)=$\frac{1}{3}$

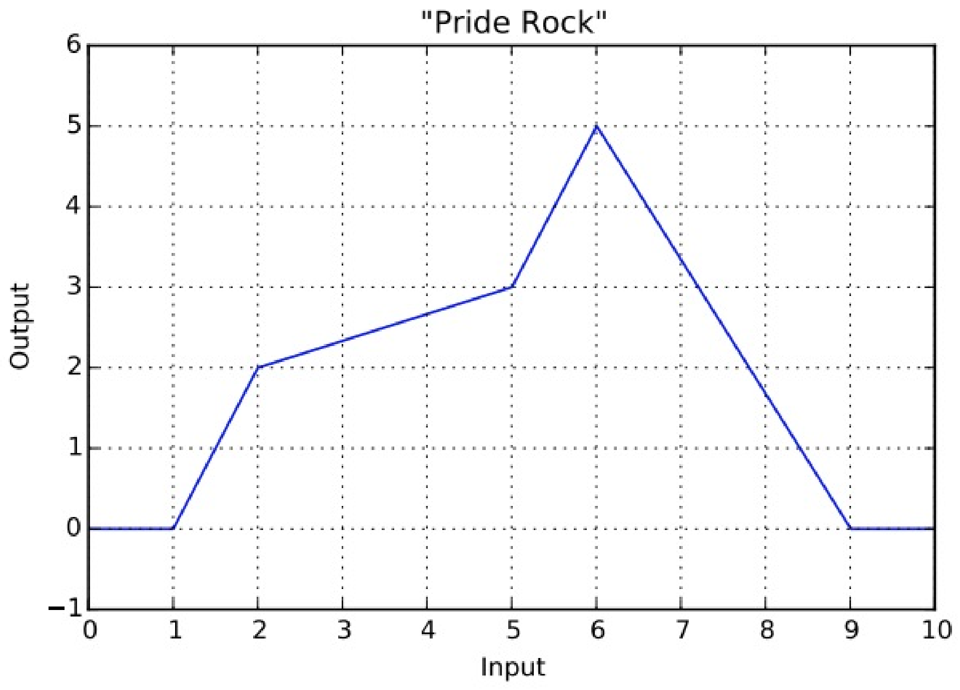
G(Windows=small)=$1-(\frac{2}{3}^2+\frac{1}{3}^2)=\frac{4}{9}$

the weighted sum of the Gini indices for Windows is

G(Windows)=$\frac{5}{11}\times\frac{8}{25}+\frac{6}{11}\times\frac{4}{9}\approx 0.388$

Since attribute Windows has the smallest Gini index, we should choose Windows at the root of this decision tree.

(b) Here I refer to the link:https://medium.com/@maximlopin/why-is-relu-non-linear-aa46d2bad518 and I first calculated the function of the poly-line. Our poly-line is like:



And based on the link above, I first calculate the function f(input) of poly-line with Relu activation function, which is in the form of $f(input) = b + \sum_{i=1}^{4} a_i.Relu(input - k_i)$. Here I use the form of Relu: $a_i.Relu(x - k_i)$. So every turning-point of the poly-line is generated by add a Relu term: $a_i.Relu(x - k_i)$.

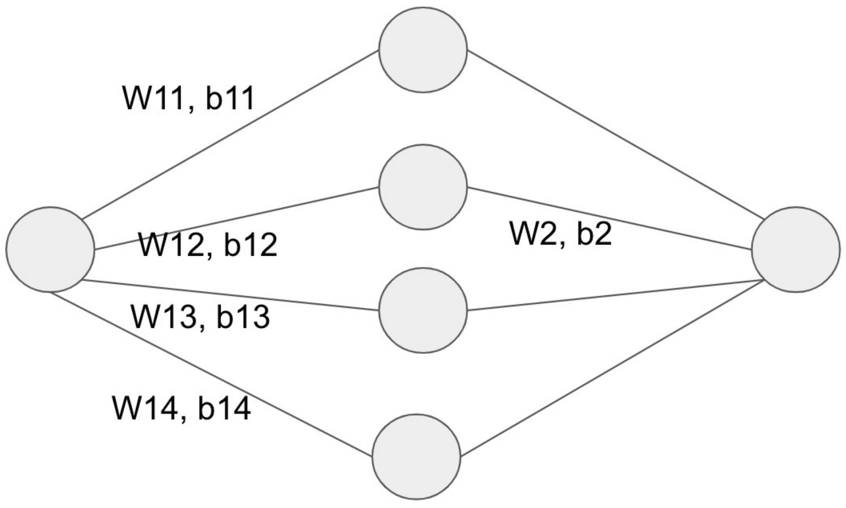
So here I calculated 4 Relu forms:

$$a_1 = 2, k_1 = 1$$
$$a_2 = \frac{-5}{3}, k_2 = 2$$
$$a_3 = \frac{5}{3}, k_3 = 5$$
$$a_4 = \frac{-11}{3}, k_4 = 6$$

And the bias of it, I set it to be 0. So the function is like this:

$$f(input) = 0 + \sum_{i=1}^{4} a_i.Relu(input - k_i) = Relu(0 + \sum_{i=1}^{4} a_i.Relu(input - k_i)$$

But we need to get a neural network, so I need to do some modification on it. So here I constructed a neural network like this:



So the input is Y_0, the value in hidden layer is Y_1, the output is Y_2. Here I can construct the W_1, B_1 and W_2, B_2 which can match the result of the poly-line function.

Here I set all the W_{1i} to be 1, and $b_{11} = -1$,$b_{12} = -2$,$b_{13} = -5$ and $b_{14} = -6$

So $Y_1 = [\sigma(W_{11}.Y_0 - 1), \sigma(W_{12}.Y_0 - 2, \sigma(W_{13}.Y_0 - 5), \sigma(W_{14}.Y_0 - 6)]^T$, which is

$Y_1 = \sigma(W_1.Y_0^T + B_1), here\ W_{1i} = 1, i = 1, 2, 3, 4\ and\ B_{11} = -1, B_{12} = -2, B_{13} = -5, B_{14} = -6$

And then, I set $W_{21} = 2, W_{22} = \frac{-5}{3}, W_{23} = \frac{5}{3}\ and\ W_{24} = \frac{-11}{3}$

And I set $B_2 = b_2 = 0$,

So $Y_2 = \sigma(W_{21}.Y_{11} + W_{22}.Y_{12} + W_{23}.Y_{13} + W_{24}.Y_{14} + b2)$

which is

$Y_2 = \sigma(2.Y_{11} + \frac{-5}{3}.Y_{12} + \frac{5}{3}.Y_{13} + \frac{-11}{3}.Y_{14} + 0) = Relu(0 + \sum_{i=1}^{4} a_i.Relu(input - k_i)$

which is

$Relu(0 + \sum_{i=1}^{4} a_i.Relu(input - k_i) = Y_2 = \sigma(W_2.Y_1^T + B_2) = \sigma(W_2.(\sigma(W_1.Y_0^T + B_1))^T + B_2)$

So, here I constructed such a neural network:

$W_{1i} = 1, here\ i = 1, 2, 3, 4$

$B_{11} = -1, B_{12} = -2, B_{13} = -5, B_{14} = -6$

$W_{21} = 2, W_{22} = \frac{-5}{3}, W_{23} = \frac{5}{3}, W_{24} = \frac{-11}{3}$

$B_2 = 0$

And the neural network I constructed has 1 hidden layer, and the number of units in the hidden layer is 4, the weights and the bias are shown above, the connections are shown in the graph above.

(c) There are N examples in the original training set which means that for each example in the original set, the probability of a specific example being selected on the first draw is $\frac{1}{N}$, i.e. the probability of that example not being selected on the first draw is $1 - \frac{1}{N}$. Since there should be N examples in a replicate set, there are N independent draws in total. So, the probability of a specific example not being selected by any draw (excluded from the bootstrap replicate set) is $(1 - \frac{1}{N})^N$

As n goes towards infinity, we have

$$\lim_{N \to \infty} (1 - \frac{1}{N})^N = \frac{1}{e}$$

which is the probability of a specific example not being selected at all. Since there are N independent Bernoulli trials, the fraction of the training set that does not appear at all in the bootstrap replicate follows a binomial distribution. Thus, the expected value of the training set that does not appear at all in the bootstrap replicate is $\frac{N}{e}$, so the expected fraction of the training set that does not appear at all in the bootstrap replicate is $\frac{N}{e}/N = \frac{1}{e}$.

So the expected fraction of the training set that does not appear at all in the bootstrap replicate is $\frac{1}{e}$