AML_HW3_Write up

Scarlett Huang sh2557 CM

Zihan Zhang zz698 ORIE

Programming 1

1.a Loading Dataset

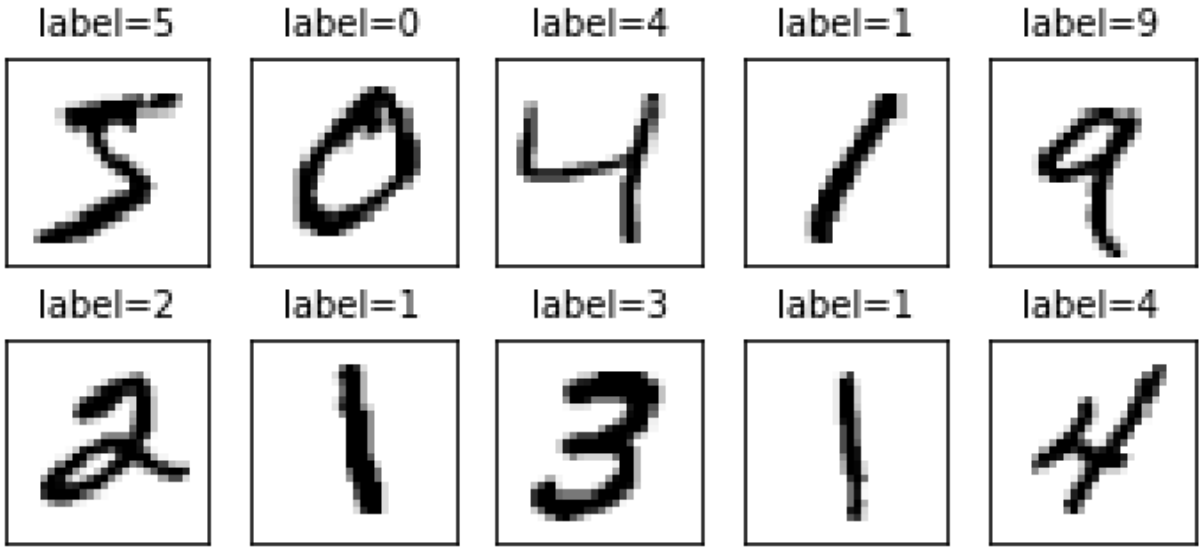
```
#
================================================================================
# 1.a
#
================================================================================
import numpy as np
import matplotlib.pyplot as plt
#loading dataset
from keras.datasets import mnist
(train_X , train_Y), (test_X , test_Y) = mnist.load_data ()
#printing out the shape
print(np.shape(train_X))
print(np.shape(test_X))
#visualize images
fig=plt.gcf()
fig.set_size_inches(6,7)
num=10
idx=0
for i in range(0,num):
    ax=plt.subplot(5,5,i+1)
    ax.imshow(train_X[idx].reshape(28,28),cmap='binary')
    title='label='+str(train_Y[idx])

    ax.set_title(title,fontsize=10)
    ax.set_xticks([]);ax.set_yticks([])
    idx+=1
plt.show()

fig=plt.gcf()
fig.set_size_inches(6,7)
num=10
idx=0
for i in range(0,num):
    ax=plt.subplot(5,5,i+1)
    ax.imshow(test_X[idx].reshape(28,28),cmap='binary')
```

```
    title='label='+str(test_Y[idx])

    ax.set_title(title,fontsize=10)
    ax.set_xticks([]);ax.set_yticks([])
    idx+=1
plt.show()
```

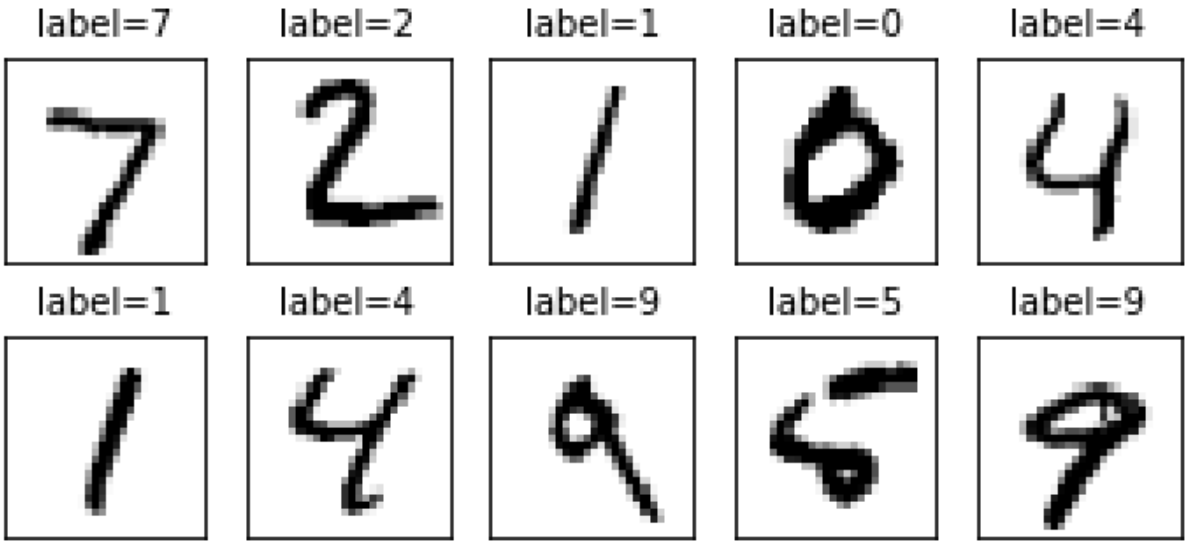
Shape of train and test dataset matrices:

```
(60000, 28, 28)
(10000, 28, 28)
```

Visualize train:



Visualize test:

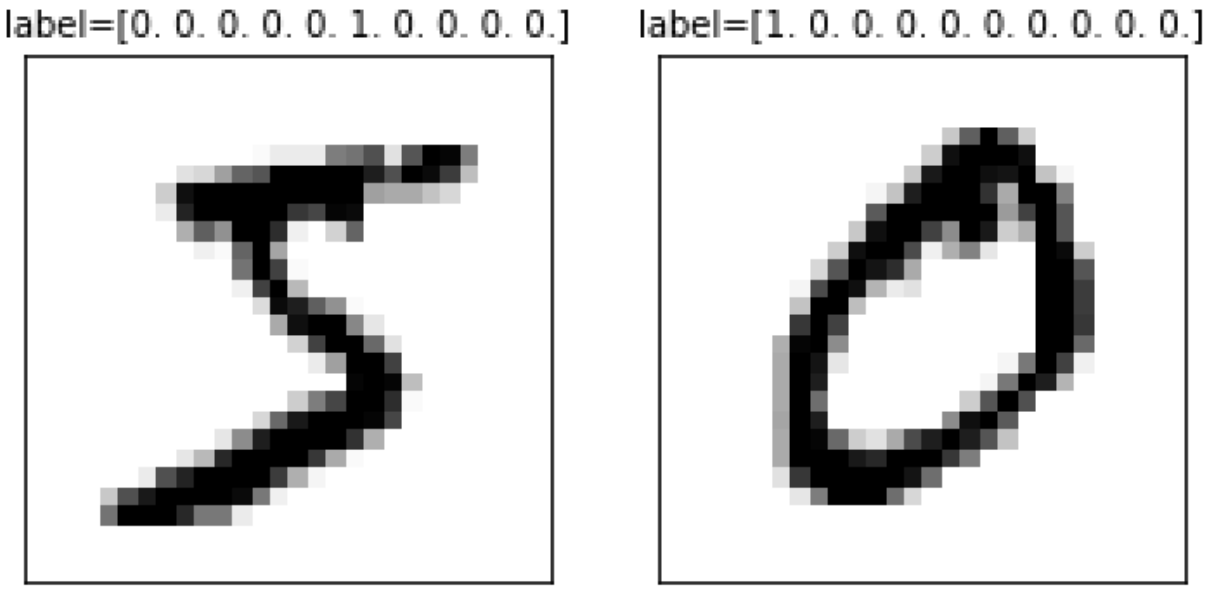


1.b Preprocessing

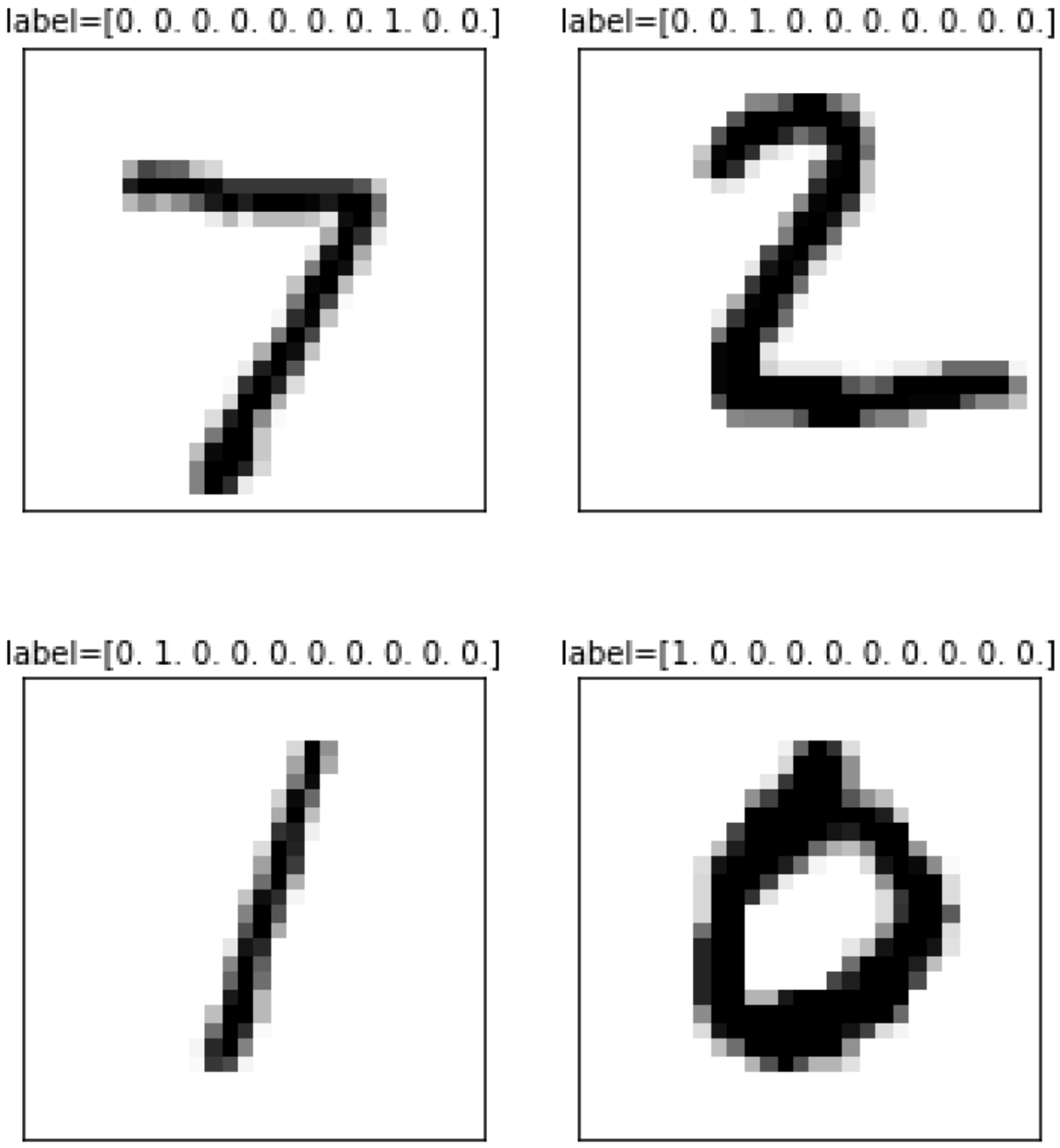
```
#
==============================================================================
# 1.b
#
==============================================================================
from keras.utils import np_utils
#reshape
train_X=train_X.reshape(60000,28,28,1).astype('float32')
test_X=test_X.reshape(10000,28,28,1).astype('float32')
#scale the pixel values
X_train_normalize=train_X/255
X_test_normalize=test_X/255
#one-hot
train_Y_onehot=np_utils.to_categorical(train_Y)
test_Y_onehot=np_utils.to_categorical(test_Y)
#visualize
fig=plt.gcf()
fig.set_size_inches(6,7)
num=4
idx=0
for i in range(0,num):
    ax=plt.subplot(2,2,i+1)
    ax.imshow(X_train_normalize[idx].reshape(28,28),cmap='binary')
    title='label='+str(train_Y_onehot[idx])

    ax.set_title(title,fontsize=10)
    ax.set_xticks([]);ax.set_yticks([])
    idx+=1
plt.show()

fig=plt.gcf()
fig.set_size_inches(6,7)
num=4
idx=0
for i in range(0,num):
    ax=plt.subplot(2,2,i+1)
    ax.imshow(X_test_normalize[idx].reshape(28,28),cmap='binary')
    title='label='+str(test_Y_onehot[idx])

    ax.set_title(title,fontsize=10)
    ax.set_xticks([]);ax.set_yticks([])
    idx+=1
plt.show()
```

Image:

train:


label=[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
label=[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
label=[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
label=[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]

test:



1.c Implementation

```
#
================================================================================
# 1.c
#
================================================================================


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))
            )
    #print(model.layers)
    # Maxpooling layer
    model.add(MaxPooling2D ((2, 2)))
    #print(model.layers)
    # Flatten output
    model.add(Flatten ())
    #print(model.layers)
    # Dense layer of 100 neurons
    model.add(
            Dense (100,
            activation='relu',
            kernel_initializer='he_uniform')
            )
    #print(model.layers)
    model.add(Dense (10, activation='softmax'))
    #print(model.layers)
    # initialize optimizer
    opt = SGD(lr=0.01 , momentum =0.9)
    # compile model
    model.compile(
            optimizer=opt ,
            loss='categorical_crossentropy',
            metrics =['accuracy']
            )
    #print(model.layers)
    return model
#create_cnn()
```

Layers:

```
[<keras.layers.convolutional.Conv2D object at 0x7fa68a4f85c0>,
<keras.layers.pooling.MaxPooling2D object at 0x7fa6962a1ba8>,
<keras.layers.core.Flatten object at 0x7fa6962a1128>,
<keras.layers.core.Dense object at 0x7fa689c43e80>,
<keras.layers.core.Dense object at 0x7fa6722f0d68>]
Out[16]: <keras.engine.sequential.Sequential at 0x7fa6960496d8>
```

1.d Training and Evaluating CNN

```
#
================================================================================
# 1.d
#
================================================================================

model=create_cnn()
model.fit(X_train_normalize,train_Y_onehot, batch_size =32, epochs =10,
validation_split =0.1)
score = model.evaluate(X_test_normalize, test_Y_onehot, verbose =0)
print(score)
```

Accuracy on test data:

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 20s 368us/step - loss: 0.1722 -
accuracy: 0.9477 - val_loss: 0.0643 - val_accuracy: 0.9828
Epoch 2/10
54000/54000 [==============================] - 20s 372us/step - loss: 0.0591 -
accuracy: 0.9824 - val_loss: 0.0567 - val_accuracy: 0.9842
Epoch 3/10
54000/54000 [==============================] - 21s 397us/step - loss: 0.0386 -
accuracy: 0.9881 - val_loss: 0.0465 - val_accuracy: 0.9872
Epoch 4/10
54000/54000 [==============================] - 23s 420us/step - loss: 0.0259 -
accuracy: 0.9922 - val_loss: 0.0466 - val_accuracy: 0.9872
Epoch 5/10
54000/54000 [==============================] - 23s 424us/step - loss: 0.0181 -
accuracy: 0.9945 - val_loss: 0.0462 - val_accuracy: 0.9877
Epoch 6/10
54000/54000 [==============================] - 21s 395us/step - loss: 0.0127 -
accuracy: 0.9964 - val_loss: 0.0455 - val_accuracy: 0.9892
Epoch 7/10
54000/54000 [==============================] - 21s 387us/step - loss: 0.0083 -
accuracy: 0.9978 - val_loss: 0.0468 - val_accuracy: 0.9890
Epoch 8/10
54000/54000 [==============================] - 21s 380us/step - loss: 0.0057 -
accuracy: 0.9986 - val_loss: 0.0483 - val_accuracy: 0.9890
Epoch 9/10
54000/54000 [==============================] - 22s 400us/step - loss: 0.0037 -
accuracy: 0.9993 - val_loss: 0.0500 - val_accuracy: 0.9883
Epoch 10/10
54000/54000 [==============================] - 20s 373us/step - loss: 0.0020 -
accuracy: 0.9998 - val_loss: 0.0464 - val_accuracy: 0.9895
[0.04087784977325482, 0.987500011920929]
```

1.e Experimentation

1.e.i

```
#
================================================================================
```

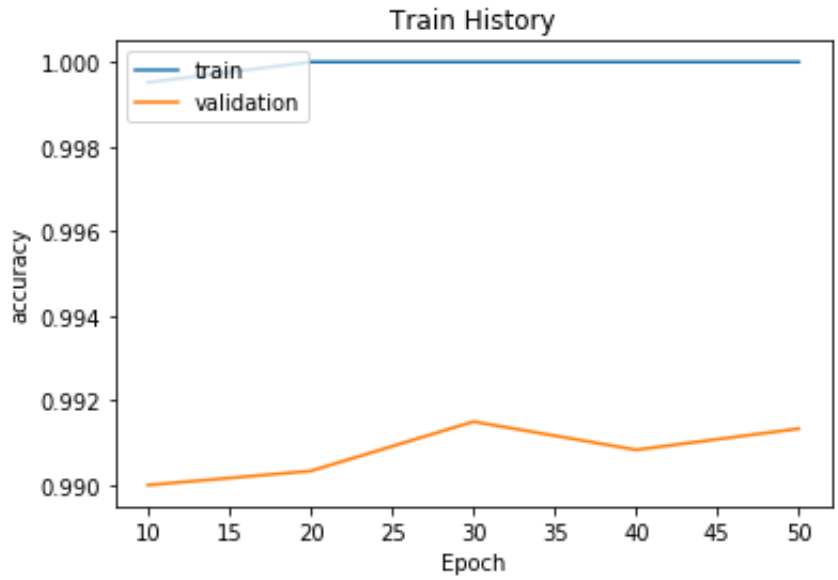
```
# 1.e.1
#
================================================================================


from keras.datasets import mnist
(train_X , train_Y), (test_X , test_Y) = mnist.load_data ()
#reshape
train_X=train_X.reshape(60000,28,28,1).astype('float32')
test_X=test_X.reshape(10000,28,28,1).astype('float32')
#scale the pixel values
X_train_normalize=train_X/255
X_test_normalize=test_X/255
#one-hot
train_Y_onehot=np_utils.to_categorical(train_Y)
test_Y_onehot=np_utils.to_categorical(test_Y)
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))
            )
    #print(model.layers)
    # Maxpooling layer
    model.add(MaxPooling2D ((2, 2)))
    #print(model.layers)
    # Flatten output
    model.add(Flatten ())
    #print(model.layers)
    # Dense layer of 100 neurons
    model.add(
            Dense (100,
            activation='relu',
            kernel_initializer='he_uniform')
            )
    #print(model.layers)
    model.add(Dense (10, activation='softmax'))
    #print(model.layers)
    # initialize optimizer
    opt = SGD(lr=0.01 , momentum =0.9)
    # compile model
    model.compile(
            optimizer=opt ,
            loss='categorical_crossentropy',
            metrics =['accuracy']
```

```
            )
    #print(model.layers)
    return model
model_1=create_cnn()
epoch_history_1 = model_1.fit(X_train_normalize , train_Y_onehot, batch_size
=32, epochs =50, validation_split =0.1)
print(epoch_history_1.history['accuracy'])
print(epoch_history_1.history['val_accuracy'])
a_1=epoch_history_1.history['accuracy']
b_1=epoch_history_1.history['val_accuracy']
acc_1=[]
val_acc_1=[]
for i in range(50):
    if (i+1)%10==0:
        acc_1.append(a_1[i])
        val_acc_1.append(b_1[i])
x=[10,20,30,40,50]
def show_epoch_history(epoch_history,train,validation):

    plt.plot(x,acc_1)
    plt.plot(x,val_acc_1)
    plt.title('Train History')
    plt.ylabel(train)
    plt.xlabel('Epoch')
    plt.legend(['train','validation'],loc='upper left')
    plt.show()

show_epoch_history(epoch_history_1,'accuracy','val_accuracy')

def show_epoch_history(epoch_history,train,validation):

    plt.plot(a_1)
    plt.plot(b_1)
    plt.title('Train History')
    plt.ylabel(train)
    plt.xlabel('Epoch')
    plt.legend(['train','validation'],loc='upper left')
    plt.show()

show_epoch_history(epoch_history,'accuracy','val_accuracy')

# print validation and training accuracy over epochs
```

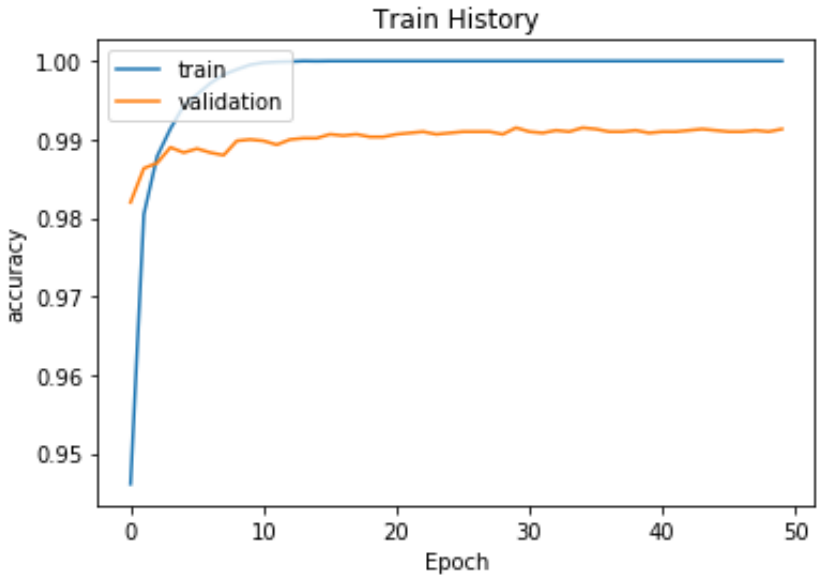
Result:

```
[0.9461111, 0.98046297, 0.98785186, 0.9913704, 0.9942963, 0.9957407,
0.9971667, 0.9982778, 0.9989259, 0.9995185, 0.9997963, 0.9998889,
0.9999259, 1.0, 0.99998146, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
[0.9819999933242798, 0.9863333106040955, 0.9869999885559082,
0.9890000224113464, 0.9883333444595337, 0.9888333082199097,
0.9883333444595337, 0.9879999756813049, 0.9898333549499512,
0.9900000095367432, 0.9898333549499512, 0.9893333315849304,
0.9900000095367432, 0.9901666641235352, 0.9901666641235352,
0.9906666874885559, 0.9904999732971191, 0.9906666874885559,
0.9903333187103271, 0.9903333187103271, 0.9906666874885559,
0.9908333420753479, 0.9909999966621399, 0.9906666874885559,
0.9908333420753479, 0.9909999966621399, 0.9909999966621399,
0.9909999966621399, 0.9906666874885559, 0.9915000200271606,
0.9909999966621399, 0.9908333420753479, 0.9911666512489319,
0.9909999966621399, 0.9915000200271606, 0.9913333058357239,
0.9909999966621399, 0.9909999966621399, 0.9911666512489319,
0.9908333420753479, 0.9909999966621399, 0.9909999966621399,
0.9911666512489319, 0.9913333058357239, 0.9911666512489319,
0.9909999966621399, 0.9909999966621399, 0.9911666512489319,
0.9909999966621399, 0.9913333058357239]
```

Plot:



There is an improvement for both training and validation accuracy.



By plotting the accuracy in every epoch, we can see the steady improvement more clearly.

1.e.ii

```
#
================================================================================
# 1.e.2 dropout
#
================================================================================

from keras.layers import Dropout
from keras.datasets import mnist
(train_X , train_Y), (test_X , test_Y) = mnist.load_data ()
#reshape
train_X=train_X.reshape(60000,28,28,1).astype('float32')
test_X=test_X.reshape(10000,28,28,1).astype('float32')
#scale the pixel values
X_train_normalize=train_X/255
X_test_normalize=test_X/255
#one-hot
train_Y_onehot=np_utils.to_categorical(train_Y)
test_Y_onehot=np_utils.to_categorical(test_Y)
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))
            )
    #print(model.layers)
    # Maxpooling layer
```

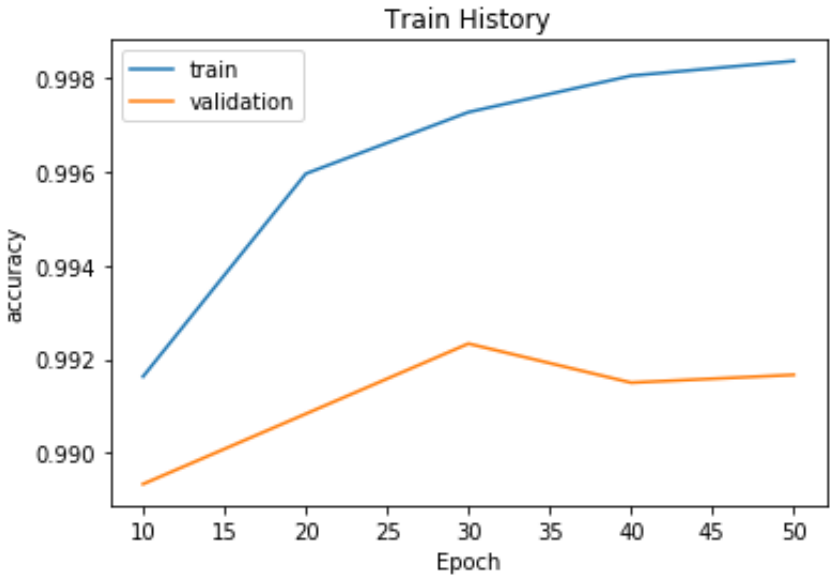
```
    model.add(MaxPooling2D ((2, 2)))
    #print(model.layers)
    # Flatten output
    model.add(Flatten ())
    #print(model.layers)
    model.add(Dropout (0.5))
    # Dense layer of 100 neurons
    model.add(
            Dense (100,
            activation='relu',
            kernel_initializer='he_uniform')
            )
    #print(model.layers)
    model.add(Dense (10, activation='softmax'))
    #print(model.layers)
    # initialize optimizer
    opt = SGD(lr=0.01 , momentum =0.9)
    # compile model
    model.compile(
            optimizer=opt ,
            loss='categorical_crossentropy',
            metrics =['accuracy']
            )
    #print(model.layers)
    return model
model_2=create_cnn()
epoch_history_2 = model_2.fit(X_train_normalize , train_Y_onehot, batch_size
=32, epochs =50, validation_split =0.1)
print(epoch_history_2.history['accuracy'])
print(epoch_history_2.history['val_accuracy'])
a_2=epoch_history_2.history['accuracy']
b_2=epoch_history_2.history['val_accuracy']
acc_2=[]
val_acc_2=[]
for i in range(50):
    if (i+1)%10==0:
        acc_2.append(a_2[i])
        val_acc_2.append(b_2[i])
x=[10,20,30,40,50]
def show_epoch_history(epoch_history,train,validation):

    plt.plot(x,acc_2)
    plt.plot(x,val_acc_2)
    plt.title('Train History')
    plt.ylabel(train)
    plt.xlabel('Epoch')
    plt.legend(['train','validation'],loc='upper left')
    plt.show()
```

```
show_epoch_history(epoch_history_2,'accuracy','val_accuracy')

def show_epoch_history_2(epoch_history,train,validation):

    plt.plot(epoch_history_2.history['accuracy'])
    plt.plot(epoch_history_2.history['val_accuracy'])
    plt.title('Train History')
    plt.ylabel(train)
    plt.xlabel('Epoch')
    plt.legend(['train','validation'],loc='upper left')
    plt.show()

show_epoch_history_2(epoch_history_2,'accuracy','val_accuracy')
```

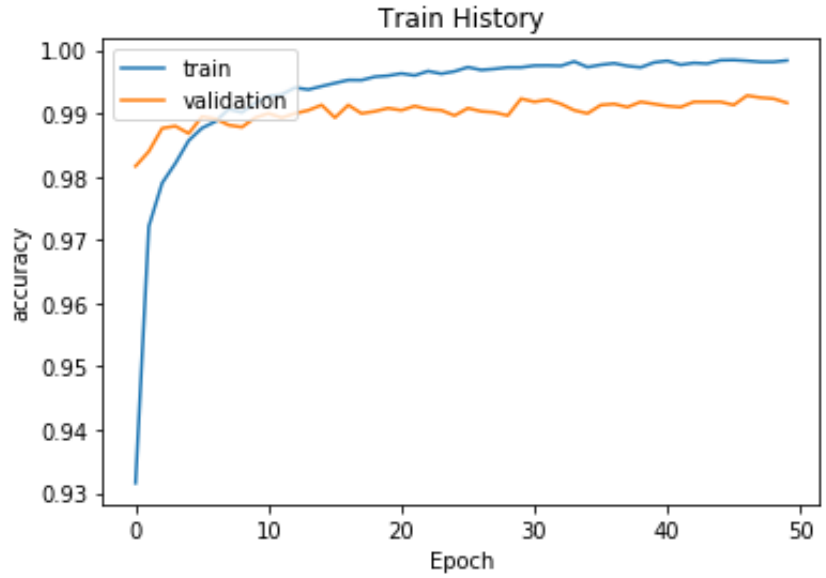
Result:

```
[0.9315926, 0.97216666, 0.979, 0.98212963, 0.98572224, 0.9876852,
0.9887222, 0.9906667, 0.99016666, 0.9916296, 0.99274075, 0.993,
0.99407405, 0.99377775, 0.9942963, 0.9947963, 0.99527776, 0.9952593,
0.9957963, 0.995963, 0.9963148, 0.996, 0.9966852, 0.99627775, 0.99664813,
0.9973148, 0.9968704, 0.99703705, 0.9972778, 0.9972778, 0.99755555,
0.9975741, 0.99751854, 0.9982037, 0.9973148, 0.9976852, 0.99792594,
0.9975, 0.9972778, 0.9980556, 0.9983148, 0.9976852, 0.9979815, 0.9978704,
0.9984074, 0.99844444, 0.9983148, 0.99814814, 0.99814814, 0.99837035]
[0.9816666841506958, 0.984000027179718, 0.987666666507721,
0.9879999756813049, 0.9868333339691162, 0.9894999861717224,
0.9891666769981384, 0.9881666898727417, 0.9878333210945129,
0.9893333315849304, 0.9900000095367432, 0.9893333315849304,
0.9900000095367432, 0.9904999732971191, 0.9913333058357239,
0.9893333315849304, 0.9913333058357239, 0.9900000095367432,
0.9903333187103271, 0.9908333420753479, 0.9904999732971191,
0.9911666512489319, 0.9906666874885559, 0.9904999732971191,
0.9896666407585144, 0.9908333420753479, 0.9903333187103271,
0.9901666641235352, 0.9896666407585144, 0.9923333525657654,
0.9918333292007446, 0.9921666383743286, 0.9915000200271606,
0.9904999732971191, 0.9900000095367432, 0.9913333058357239,
0.9915000200271606, 0.9909999966621399, 0.9918333292007446,
0.9915000200271606, 0.9911666512489319, 0.9909999966621399,
0.9918333292007446, 0.9918333292007446, 0.9918333292007446,
0.9913333058357239, 0.9928333163261414, 0.9925000071525574,
0.9923333525657654, 0.9916666746139526]
```

Plot:



There is a steady improvement for both training and validation accuracy.



By plotting the accuracy in each epoch, the improvement is fluctuated.

1.e.iii

```
#
================================================================================
# 1.e.3 Add another convolution layer and maxpooling layer
#
================================================================================

from keras.datasets import mnist
(train_X , train_Y), (test_X , test_Y) = mnist.load_data ()
#reshape
train_X=train_X.reshape(60000,28,28,1).astype('float32')
test_X=test_X.reshape(10000,28,28,1).astype('float32')
#scale the pixel values
X_train_normalize=train_X/255
```

```
X_test_normalize=test_X/255
#one-hot
train_Y_onehot=np_utils.to_categorical(train_Y)
test_Y_onehot=np_utils.to_categorical(test_Y)
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)))
    # Add another convolution layer
    model.add(
            Conv2D(64,(3,3),
            activation='relu',
            kernel_initializer='he_uniform',
            input_shape =(28, 28, 1))
            )
    model.add(MaxPooling2D((2,2)))
    # Flatten output
    model.add(Flatten ())
    #print(model.layers)
    model.add(Dropout (0.5))
    # Dense layer of 100 neurons
    model.add(
            Dense (100,
            activation='relu',
            kernel_initializer='he_uniform')
            )
    #print(model.layers)
    model.add(Dense (10, activation='softmax'))
    #print(model.layers)
    # initialize optimizer
    opt = SGD(lr=0.01 , momentum =0.9)
    # compile model
    model.compile(
            optimizer=opt ,
            loss='categorical_crossentropy',
            metrics =['accuracy']
            )
    #print(model.layers)
    return model
model=create_cnn()
```

```
model.fit(X_train_normalize,train_Y_onehot, batch_size =32, epochs =10,
validation_split =0.1)
score = model.evaluate(X_test_normalize, test_Y_onehot, verbose =0)
print(score)
```

Test accuracy:

```
In [39]: score = model.evaluate(X_test_normalize, test_Y_onehot, verbose =0)
    ...: print(score)
[0.023546459459901234, 0.9915000200271606]
```

Accuracy for each epoch:

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 36s 671us/step - loss: 0.1984 -
accuracy: 0.9366 - val_loss: 0.0464 - val_accuracy: 0.9870
Epoch 2/10
54000/54000 [==============================] - 35s 639us/step - loss: 0.0794 -
accuracy: 0.9753 - val_loss: 0.0388 - val_accuracy: 0.9880
Epoch 3/10
54000/54000 [==============================] - 35s 645us/step - loss: 0.0623 -
accuracy: 0.9808 - val_loss: 0.0342 - val_accuracy: 0.9910
Epoch 4/10
54000/54000 [==============================] - 34s 637us/step - loss: 0.0505 -
accuracy: 0.9836 - val_loss: 0.0384 - val_accuracy: 0.9885
Epoch 5/10
54000/54000 [==============================] - 34s 632us/step - loss: 0.0429 -
accuracy: 0.9863 - val_loss: 0.0296 - val_accuracy: 0.9920
Epoch 6/10
54000/54000 [==============================] - 34s 632us/step - loss: 0.0380 -
accuracy: 0.9879 - val_loss: 0.0286 - val_accuracy: 0.9913
Epoch 7/10
54000/54000 [==============================] - 34s 637us/step - loss: 0.0361 -
accuracy: 0.9882 - val_loss: 0.0263 - val_accuracy: 0.9925
Epoch 8/10
54000/54000 [==============================] - 34s 635us/step - loss: 0.0328 -
accuracy: 0.9892 - val_loss: 0.0296 - val_accuracy: 0.9918
Epoch 9/10
54000/54000 [==============================] - 34s 635us/step - loss: 0.0307 -
accuracy: 0.9900 - val_loss: 0.0278 - val_accuracy: 0.9920
Epoch 10/10
54000/54000 [==============================] - 35s 639us/step - loss: 0.0263 -
accuracy: 0.9910 - val_loss: 0.0268 - val_accuracy: 0.9937
```

1.e.iv

```
#
================================================================================
# 1.e.4 learning rate
#
================================================================================

#0.001 learning rate

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

```
train_X=train_X.reshape(60000,28,28,1).astype('float32')
test_X=test_X.reshape(10000,28,28,1).astype('float32')
#scale the pixel values
X_train_normalize=train_X/255
X_test_normalize=test_X/255
#one-hot
train_Y_onehot=np_utils.to_categorical(train_Y)
test_Y_onehot=np_utils.to_categorical(test_Y)
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)))
    # Add another convolution layer
    model.add(
            Conv2D(64,(3,3),
            activation='relu',
            kernel_initializer='he_uniform',
            input_shape =(28, 28, 1))
            )
    model.add(MaxPooling2D((2,2)))
    # Flatten output
    model.add(Flatten ())
    #print(model.layers)
    model.add(Dropout (0.5))
    # Dense layer of 100 neurons
    model.add(
            Dense (100,
            activation='relu',
            kernel_initializer='he_uniform')
            )
    #print(model.layers)
    model.add(Dense (10, activation='softmax'))
    #print(model.layers)
    # initialize optimizer
    opt = SGD(lr=0.001 , momentum =0.9)
    # compile model
    model.compile(
            optimizer=opt ,
            loss='categorical_crossentropy',
            metrics =['accuracy']
            )
```

```
    #print(model.layers)
    return model
model=create_cnn()
model.fit(X_train_normalize,train_Y_onehot, batch_size =32, epochs =10,
validation_split =0.1)
score = model.evaluate(X_test_normalize, test_Y_onehot, verbose =0)
print(score)

#0.1 learning rate

from keras.datasets import mnist
(train_X , train_Y), (test_X , test_Y) = mnist.load_data ()
#reshape
train_X=train_X.reshape(60000,28,28,1).astype('float32')
test_X=test_X.reshape(10000,28,28,1).astype('float32')
#scale the pixel values
X_train_normalize=train_X/255
X_test_normalize=test_X/255
#one-hot
train_Y_onehot=np_utils.to_categorical(train_Y)
test_Y_onehot=np_utils.to_categorical(test_Y)
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)))
    # Add another convolution layer
    model.add(
            Conv2D(64,(3,3),
            activation='relu',
            kernel_initializer='he_uniform',
            input_shape =(28, 28, 1))
            )
    model.add(MaxPooling2D((2,2)))
    # Flatten output
    model.add(Flatten ())
    #print(model.layers)
    model.add(Dropout (0.5))
    # Dense layer of 100 neurons
    model.add(
            Dense (100,
            activation='relu',
```

```
            kernel_initializer='he_uniform')
            )
    #print(model.layers)
    model.add(Dense (10, activation='softmax'))
    #print(model.layers)
    # initialize optimizer
    opt = SGD(lr=0.1 , momentum =0.9)
    # compile model
    model.compile(
            optimizer=opt ,
            loss='categorical_crossentropy',
            metrics =['accuracy']
            )
    #print(model.layers)
    return model
model=create_cnn()
model.fit(X_train_normalize,train_Y_onehot, batch_size =32, epochs =10,
validation_split =0.1)
score = model.evaluate(X_test_normalize, test_Y_onehot, verbose =0)
print(score)
```

Test accuracy with learning rate=0.001:

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 41s 761us/step - loss: 0.4341 -
accuracy: 0.8627 - val_loss: 0.0998 - val_accuracy: 0.9732
Epoch 2/10
54000/54000 [==============================] - 35s 650us/step - loss: 0.1632 -
accuracy: 0.9505 - val_loss: 0.0781 - val_accuracy: 0.9758
Epoch 3/10
54000/54000 [==============================] - 41s 755us/step - loss: 0.1237 -
accuracy: 0.9612 - val_loss: 0.0604 - val_accuracy: 0.9835
Epoch 4/10
54000/54000 [==============================] - 39s 722us/step - loss: 0.1047 -
accuracy: 0.9683 - val_loss: 0.0552 - val_accuracy: 0.9857
Epoch 5/10
54000/54000 [==============================] - 35s 645us/step - loss: 0.0929 -
accuracy: 0.9711 - val_loss: 0.0497 - val_accuracy: 0.9852
Epoch 6/10
54000/54000 [==============================] - 35s 645us/step - loss: 0.0831 -
accuracy: 0.9745 - val_loss: 0.0504 - val_accuracy: 0.9852
Epoch 7/10
54000/54000 [==============================] - 34s 634us/step - loss: 0.0784 -
accuracy: 0.9757 - val_loss: 0.0435 - val_accuracy: 0.9870
Epoch 8/10
54000/54000 [==============================] - 34s 636us/step - loss: 0.0725 -
accuracy: 0.9778 - val_loss: 0.0415 - val_accuracy: 0.9882
Epoch 9/10
54000/54000 [==============================] - 34s 636us/step - loss: 0.0670 -
accuracy: 0.9787 - val_loss: 0.0400 - val_accuracy: 0.9885
Epoch 10/10
54000/54000 [==============================] - 34s 632us/step - loss: 0.0639 -
accuracy: 0.9799 - val_loss: 0.0380 - val_accuracy: 0.9895
[0.03657758141306695, 0.9876999855041504]
```

Test acc = 0.9876999855041504.

Test accuracy with learning rate=0.1:

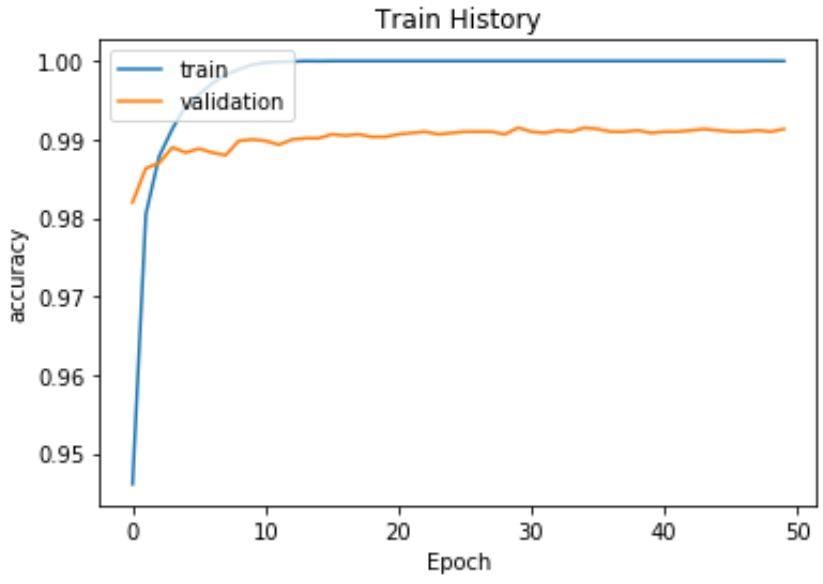
```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 35s 649us/step - loss: 1.0162 -
accuracy: 0.7235 - val_loss: 0.7907 - val_accuracy: 0.7328
Epoch 2/10
54000/54000 [==============================] - 35s 646us/step - loss: 2.2747 -
accuracy: 0.1252 - val_loss: 2.3057 - val_accuracy: 0.1045
Epoch 3/10
54000/54000 [==============================] - 36s 658us/step - loss: 2.3079 -
accuracy: 0.1062 - val_loss: 2.3059 - val_accuracy: 0.1050
Epoch 4/10
54000/54000 [==============================] - 35s 649us/step - loss: 2.3095 -
accuracy: 0.1058 - val_loss: 2.3062 - val_accuracy: 0.1050
Epoch 5/10
54000/54000 [==============================] - 35s 650us/step - loss: 2.3082 -
accuracy: 0.1041 - val_loss: 2.3076 - val_accuracy: 0.1050
Epoch 6/10
54000/54000 [==============================] - 35s 648us/step - loss: 2.3079 -
accuracy: 0.1044 - val_loss: 2.3056 - val_accuracy: 0.1050
Epoch 7/10
54000/54000 [==============================] - 35s 651us/step - loss: 2.3086 -
accuracy: 0.1063 - val_loss: 2.3068 - val_accuracy: 0.0960
Epoch 8/10
54000/54000 [==============================] - 35s 647us/step - loss: 2.3082 -
accuracy: 0.1041 - val_loss: 2.3147 - val_accuracy: 0.1050
Epoch 9/10
54000/54000 [==============================] - 35s 651us/step - loss: 2.3076 -
accuracy: 0.1045 - val_loss: 2.3067 - val_accuracy: 0.1050
Epoch 10/10
54000/54000 [==============================] - 35s 649us/step - loss: 2.3077 -
accuracy: 0.1056 - val_loss: 2.3117 - val_accuracy: 0.1050
[2.307198419189453, 0.11349999904632568]
```

Test acc = 0.11349999904632568

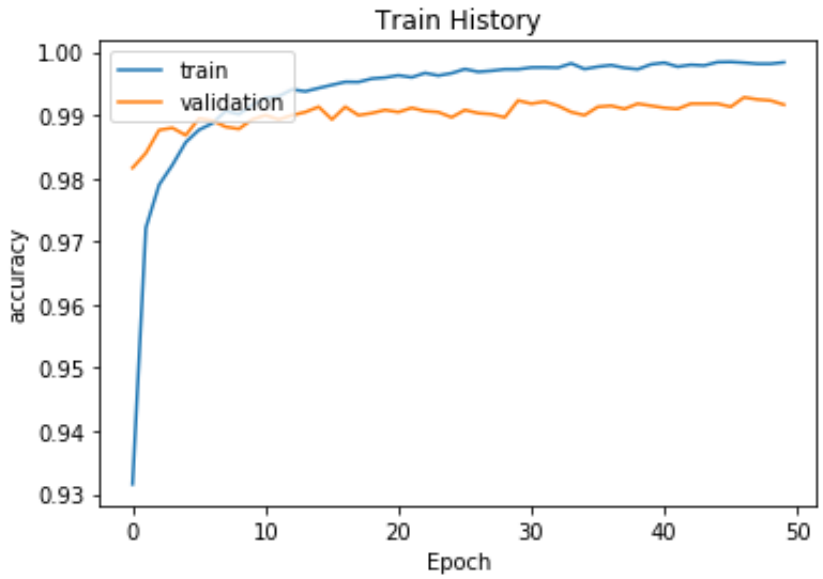
1.f Analysis

1.f.1

The trend in validation and train accuracy before using the dropout layer:



The trend in validation and train accuracy after using the dropout layer:



The improvement of train accuracy before using the dropout layer is more steady than that after using the dropout layer.

The performance of validation is better after using the dropout layer than before.

Since the dropout layer reduces the probability of overfitting, the performance of the training data would be a little bit worse after using it, which is reasonable.

1.f.2

The performance of CNN with one convolution layer:

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 20s 368us/step - loss: 0.1722 -
accuracy: 0.9477 - val_loss: 0.0643 - val_accuracy: 0.9828
Epoch 2/10
54000/54000 [==============================] - 20s 372us/step - loss: 0.0591 -
accuracy: 0.9824 - val_loss: 0.0567 - val_accuracy: 0.9842
Epoch 3/10
54000/54000 [==============================] - 21s 397us/step - loss: 0.0386 -
accuracy: 0.9881 - val_loss: 0.0465 - val_accuracy: 0.9872
Epoch 4/10
54000/54000 [==============================] - 23s 420us/step - loss: 0.0259 -
accuracy: 0.9922 - val_loss: 0.0466 - val_accuracy: 0.9872
Epoch 5/10
54000/54000 [==============================] - 23s 424us/step - loss: 0.0181 -
accuracy: 0.9945 - val_loss: 0.0462 - val_accuracy: 0.9877
Epoch 6/10
54000/54000 [==============================] - 21s 395us/step - loss: 0.0127 -
accuracy: 0.9964 - val_loss: 0.0455 - val_accuracy: 0.9892
Epoch 7/10
54000/54000 [==============================] - 21s 387us/step - loss: 0.0083 -
accuracy: 0.9978 - val_loss: 0.0468 - val_accuracy: 0.9890
Epoch 8/10
54000/54000 [==============================] - 21s 380us/step - loss: 0.0057 -
accuracy: 0.9986 - val_loss: 0.0483 - val_accuracy: 0.9890
Epoch 9/10
54000/54000 [==============================] - 22s 400us/step - loss: 0.0037 -
accuracy: 0.9993 - val_loss: 0.0500 - val_accuracy: 0.9883
Epoch 10/10
54000/54000 [==============================] - 20s 373us/step - loss: 0.0020 -
accuracy: 0.9998 - val_loss: 0.0464 - val_accuracy: 0.9895
[0.04087784977325482, 0.987500011920929]
```

The performance of CNN with two convolution layers:

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 36s 671us/step - loss: 0.1984 -
accuracy: 0.9366 - val_loss: 0.0464 - val_accuracy: 0.9870
Epoch 2/10
54000/54000 [==============================] - 35s 639us/step - loss: 0.0794 -
accuracy: 0.9753 - val_loss: 0.0388 - val_accuracy: 0.9880
Epoch 3/10
54000/54000 [==============================] - 35s 645us/step - loss: 0.0623 -
accuracy: 0.9808 - val_loss: 0.0342 - val_accuracy: 0.9910
Epoch 4/10
54000/54000 [==============================] - 34s 637us/step - loss: 0.0505 -
accuracy: 0.9836 - val_loss: 0.0384 - val_accuracy: 0.9885
Epoch 5/10
54000/54000 [==============================] - 34s 632us/step - loss: 0.0429 -
accuracy: 0.9863 - val_loss: 0.0296 - val_accuracy: 0.9920
Epoch 6/10
54000/54000 [==============================] - 34s 632us/step - loss: 0.0380 -
accuracy: 0.9879 - val_loss: 0.0286 - val_accuracy: 0.9913
Epoch 7/10
54000/54000 [==============================] - 34s 637us/step - loss: 0.0361 -
accuracy: 0.9882 - val_loss: 0.0263 - val_accuracy: 0.9925
Epoch 8/10
54000/54000 [==============================] - 34s 635us/step - loss: 0.0328 -
accuracy: 0.9892 - val_loss: 0.0296 - val_accuracy: 0.9918
Epoch 9/10
54000/54000 [==============================] - 34s 635us/step - loss: 0.0307 -
accuracy: 0.9900 - val_loss: 0.0278 - val_accuracy: 0.9920
Epoch 10/10
54000/54000 [==============================] - 35s 639us/step - loss: 0.0263 -
accuracy: 0.9910 - val_loss: 0.0268 - val_accuracy: 0.9937
```

```
In [39]: score = model.evaluate(X_test_normalize, test_Y_onehot, verbose =0)
    ...: print(score)
[0.023546459459901234, 0.9915000200271606]
```

The performance of CNN with a single layer is better than two layers on training data, has a highere accuracy on train and the improvement is steady.

However, the performance of CNN with two layers is better than a single layer on validation and test data. It has a higher accuracy and a lower loss on test data.

1.f.3

Learing rate=0.1:

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 35s 649us/step - loss: 1.0162 -
accuracy: 0.7235 - val_loss: 0.7907 - val_accuracy: 0.7328
Epoch 2/10
54000/54000 [==============================] - 35s 646us/step - loss: 2.2747 -
accuracy: 0.1252 - val_loss: 2.3057 - val_accuracy: 0.1045
Epoch 3/10
54000/54000 [==============================] - 36s 658us/step - loss: 2.3079 -
accuracy: 0.1062 - val_loss: 2.3059 - val_accuracy: 0.1050
Epoch 4/10
54000/54000 [==============================] - 35s 649us/step - loss: 2.3095 -
accuracy: 0.1058 - val_loss: 2.3062 - val_accuracy: 0.1050
Epoch 5/10
54000/54000 [==============================] - 35s 650us/step - loss: 2.3082 -
accuracy: 0.1041 - val_loss: 2.3076 - val_accuracy: 0.1050
Epoch 6/10
54000/54000 [==============================] - 35s 648us/step - loss: 2.3079 -
accuracy: 0.1044 - val_loss: 2.3056 - val_accuracy: 0.1050
Epoch 7/10
54000/54000 [==============================] - 35s 651us/step - loss: 2.3086 -
accuracy: 0.1063 - val_loss: 2.3068 - val_accuracy: 0.0960
Epoch 8/10
54000/54000 [==============================] - 35s 647us/step - loss: 2.3082 -
accuracy: 0.1041 - val_loss: 2.3147 - val_accuracy: 0.1050
Epoch 9/10
54000/54000 [==============================] - 35s 651us/step - loss: 2.3076 -
accuracy: 0.1045 - val_loss: 2.3067 - val_accuracy: 0.1050
Epoch 10/10
54000/54000 [==============================] - 35s 649us/step - loss: 2.3077 -
accuracy: 0.1056 - val_loss: 2.3117 - val_accuracy: 0.1050
[2.307198419189453, 0.11349999904632568]
```

Learning rate=0.01:

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 20s 368us/step - loss: 0.1722 -
accuracy: 0.9477 - val_loss: 0.0643 - val_accuracy: 0.9828
Epoch 2/10
54000/54000 [==============================] - 20s 372us/step - loss: 0.0591 -
accuracy: 0.9824 - val_loss: 0.0567 - val_accuracy: 0.9842
Epoch 3/10
54000/54000 [==============================] - 21s 397us/step - loss: 0.0386 -
accuracy: 0.9881 - val_loss: 0.0465 - val_accuracy: 0.9872
Epoch 4/10
54000/54000 [==============================] - 23s 420us/step - loss: 0.0259 -
accuracy: 0.9922 - val_loss: 0.0466 - val_accuracy: 0.9872
Epoch 5/10
54000/54000 [==============================] - 23s 424us/step - loss: 0.0181 -
accuracy: 0.9945 - val_loss: 0.0462 - val_accuracy: 0.9877
Epoch 6/10
54000/54000 [==============================] - 21s 395us/step - loss: 0.0127 -
accuracy: 0.9964 - val_loss: 0.0455 - val_accuracy: 0.9892
Epoch 7/10
54000/54000 [==============================] - 21s 387us/step - loss: 0.0083 -
accuracy: 0.9978 - val_loss: 0.0468 - val_accuracy: 0.9890
Epoch 8/10
54000/54000 [==============================] - 21s 380us/step - loss: 0.0057 -
accuracy: 0.9986 - val_loss: 0.0483 - val_accuracy: 0.9890
Epoch 9/10
54000/54000 [==============================] - 22s 400us/step - loss: 0.0037 -
accuracy: 0.9993 - val_loss: 0.0500 - val_accuracy: 0.9883
Epoch 10/10
54000/54000 [==============================] - 20s 373us/step - loss: 0.0020 -
accuracy: 0.9998 - val_loss: 0.0464 - val_accuracy: 0.9895
[0.04087784977325482, 0.987500011920929]
```

Learning rate=0.001:

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 41s 761us/step - loss: 0.4341 -
accuracy: 0.8627 - val_loss: 0.0998 - val_accuracy: 0.9732
Epoch 2/10
54000/54000 [==============================] - 35s 650us/step - loss: 0.1632 -
accuracy: 0.9505 - val_loss: 0.0781 - val_accuracy: 0.9758
Epoch 3/10
54000/54000 [==============================] - 41s 755us/step - loss: 0.1237 -
accuracy: 0.9612 - val_loss: 0.0604 - val_accuracy: 0.9835
Epoch 4/10
54000/54000 [==============================] - 39s 722us/step - loss: 0.1047 -
accuracy: 0.9683 - val_loss: 0.0552 - val_accuracy: 0.9857
Epoch 5/10
54000/54000 [==============================] - 35s 645us/step - loss: 0.0929 -
accuracy: 0.9711 - val_loss: 0.0497 - val_accuracy: 0.9852
Epoch 6/10
54000/54000 [==============================] - 35s 645us/step - loss: 0.0831 -
accuracy: 0.9745 - val_loss: 0.0504 - val_accuracy: 0.9852
Epoch 7/10
54000/54000 [==============================] - 34s 634us/step - loss: 0.0784 -
accuracy: 0.9757 - val_loss: 0.0435 - val_accuracy: 0.9870
Epoch 8/10
54000/54000 [==============================] - 34s 636us/step - loss: 0.0725 -
accuracy: 0.9778 - val_loss: 0.0415 - val_accuracy: 0.9882
Epoch 9/10
54000/54000 [==============================] - 34s 636us/step - loss: 0.0670 -
accuracy: 0.9787 - val_loss: 0.0400 - val_accuracy: 0.9885
Epoch 10/10
54000/54000 [==============================] - 34s 632us/step - loss: 0.0639 -
accuracy: 0.9799 - val_loss: 0.0380 - val_accuracy: 0.9895
[0.03657758141306695, 0.9876999855041504]
```

When the learning rate is set too large, the gradient may oscillate back and forth in the vicinity and fail to converge.

When the learning rate is set too small, the convergence process will become slow.

A proper learning rate can make the objective function converge to a local minimum in a proper time.

Therefore, choosing an appropriate learning rate is crucial for the training of the model.

Programming 2

2.b Preprocessing the input

1.**Unit-Normalization**: To avoid overfitting, I used Unit-Normalization, that is, mapping all coordinate locations to the [0,1] range.

2.**Standardization**: I didn't use the standardization method. Because standardization is not required for tree-based models (e.g. random forest, Bagging and boosting) which are not sensitive to variable size.

3.**Mean subtraction**: I didn't use mean subtraction, because it's not necessary to do this in this task.

2.c Preprocessing the output

1.Rescale the pixel intensities to lie between 0.0 and 1.0.

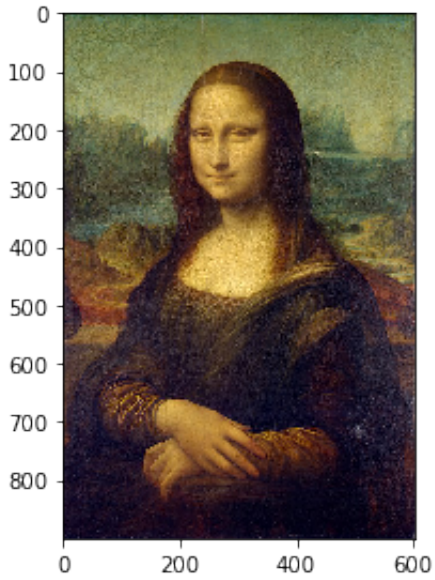
2.Convert the image to grayscale.

```
# import modules

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import cv2
import os
from skimage import
data,filters,io,transform,feature,segmentation,restoration,util,color
```

```
# use color image

img = io.imread('data/monalisa_ori.jpg') # read image
print (img.shape)
print (img.ndim)
io.imshow(img)
print(img[:2])

# gray: default 0-1
# RGB: default 0-255
img = np.array(img, dtype=np.float32)/255.0
img
```

```
(900, 604, 3)
3
[[[103 121  97]
  [ 79  87  66]
  [112 113  97]
  ...
  [ 85  91  79]
  [116 117 111]
  [ 97  93  90]]

 [[110 128 102]
  [107 115  94]
  [100 101  85]
  ...
  [ 71  77  63]
  [ 84  85  77]
  [ 60  59  54]]]
```

```
array([[[0.40392157, 0.4745098 , 0.38039216],
        [0.30980393, 0.34117648, 0.25882354],
        [0.4392157 , 0.44313726, 0.38039216],
        ...,
        [0.33333334, 0.35686275, 0.30980393],
        [0.45490196, 0.45882353, 0.43529412],
        [0.38039216, 0.3647059 , 0.3529412 ]],

       [[0.43137255, 0.5019608 , 0.4       ],
        [0.41960785, 0.4509804 , 0.36862746],
        [0.39215687, 0.39607844, 0.33333334],
        ...,
        [0.2784314 , 0.3019608 , 0.24705882],
        [0.32941177, 0.33333334, 0.3019608 ],
        [0.23529412, 0.23137255, 0.21176471]],

       [[0.31764707, 0.3882353 , 0.28627452],
        [0.3372549 , 0.36862746, 0.2784314 ],
        [0.42352942, 0.42745098, 0.35686275],
        ...,
        [0.29411766, 0.32156864, 0.25882354],
        [0.27058825, 0.28627452, 0.23137255],
        [0.16862746, 0.16862746, 0.13725491]],

       ...,

       [[0.44313726, 0.43137255, 0.46666667],
        [0.24313726, 0.23137255, 0.27450982],
        [0.12156863, 0.10588235, 0.16470589],
        ...,
        [0.1254902 , 0.10588235, 0.18431373],
        [0.52156866, 0.49411765, 0.5647059 ],
        [0.1882353 , 0.16078432, 0.23137255]],

       [[0.47843137, 0.46666667, 0.5019608 ],
        [0.27058825, 0.25882354, 0.3019608 ],
        [0.30980393, 0.29411766, 0.3529412 ],
        ...,
        [0.09411765, 0.07450981, 0.15294118],
        [0.30588236, 0.2784314 , 0.34901962],
        [0.1254902 , 0.09803922, 0.16862746]],

       [[0.43529412, 0.42352942, 0.45882353],
        [0.18431373, 0.17254902, 0.21568628],
        [0.26666668, 0.2509804 , 0.30980393],
        ...,
        [0.4117647 , 0.39215687, 0.47058824],
        [0.16078432, 0.13333334, 0.20392157],
        [0.39215687, 0.3647059 , 0.43529412]]], dtype=float32)
```




```
# meshgrid, get coordinate locations

x = np.arange(0, img.shape[1])
y = np.arange(0, img.shape[0])
locations = np.meshgrid(y,x)
locations = np.stack(locations, axis=-1).reshape(-1, 2)

locations.shape
```

```
(543600, 2)
```

```
# sample points: uniformly sample 5,000 random (x,y) coordinate locations, to
build a training set

num_sample_points = 5000
np.random.shuffle(locations)
sample_points = locations[:num_sample_points]
test_points = locations[num_sample_points:]

print (sample_points[:10])
```

```
[[772 241]
 [377 485]
 [447 163]
 [448  11]
 [ 56 539]
 [161 163]
 [403 437]
 [835 314]
 [581 364]
 [193 460]]
```

```
# pixels match

sample_point_pixels = np.array([img[x[0], x[1]] for x in sample_points],
dtype='float32')
test_point_pixels = np.array([img[x[0], x[1]] for x in test_points],
dtype='float32')
```

```
# normalize the coordinates

normalized_sample_points = sample_points.copy().astype('float32')
normalized_test_points = test_points.copy().astype('float32')

normalized_sample_points[:, 0] /= img.shape[0]
normalized_sample_points[:, 1] /= img.shape[1]
normalized_test_points[:, 0] /= img.shape[0]
normalized_test_points[:, 1] /= img.shape[1]
```

```
print (sample_point_pixels[:5])
print (test_point_pixels[:5])
```

```
[[0.5411765  0.34117648 0.17254902]
 [0.3137255  0.31764707 0.19607843]
 [0.21568628 0.15294118 0.19607843]
 [0.45882353 0.27450982 0.20392157]
 [0.42745098 0.49411765 0.32941177]]
[[0.5058824  0.30588236 0.19215687]
 [0.13333334 0.04313726 0.11372549]
 [0.08627451 0.03921569 0.09411765]
 [0.6313726  0.627451   0.41568628]
 [0.49411765 0.54509807 0.3254902 ]]
```

```
print (normalized_sample_points[:5])
print (normalized_test_points[:5])
```

```
[[0.8577778  0.39900663]
 [0.4188889  0.8029801 ]
 [0.49666667 0.26986754]
 [0.4977778  0.01821192]
 [0.06222222 0.8923841 ]]
[[0.86777776 0.35927153]
 [0.9522222  0.39072847]
 [0.57       0.8642384 ]
 [0.21111111 0.13907285]
 [0.08555555 0.294702  ]]
```

```
x_train = normalized_sample_points
y_train = sample_point_pixels
x_test = normalized_test_points
y_test = test_point_pixels
```

2.d Implement Random Forest Regressor, Build the final image

```
%run 1.Q2_b+c_ColorImage.py
```

```
# implement Random Forest Regressor

from sklearn.ensemble import RandomForestRegressor

rfr = RandomForestRegressor(n_estimators=50, max_depth=8, criterion='mse')
rfr.fit(x_train, y_train)
y_pred = rfr.predict(x_test)

print ("y_train:\n", y_train[:5])
print ("y_pred:\n", y_pred[:5])
print ("y_pred shape:", y_pred.shape)
```

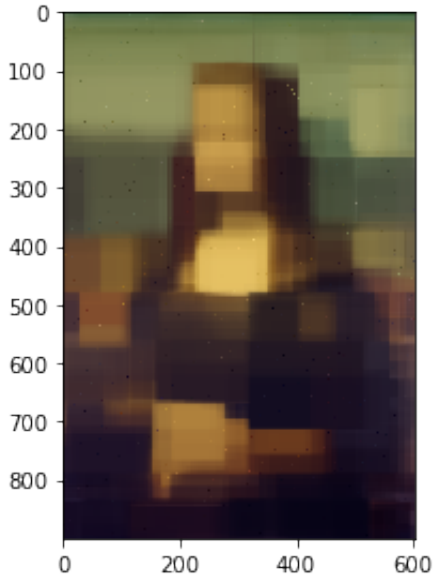
```
y_train:
 [[0.47843137 0.5294118  0.31764707]
 [0.06666667 0.05490196 0.12941177]
 [0.42352942 0.43137255 0.2784314 ]
 [0.3764706  0.41568628 0.28235295]
 [0.23921569 0.20392157 0.21568628]]
y_pred:
 [[0.31693894 0.35834485 0.26902258]
 [0.50575719 0.37444329 0.18090416]
 [0.06159043 0.03571006 0.11538387]
 [0.38446765 0.38338337 0.26658679]
 [0.21594101 0.14575999 0.13848056]]
y_pred shape: (538600, 3)
```

```python
# combine points and sample points

image = np.zeros_like(img)
print(image.shape)

for point, pixel in zip(sample_points, sample_point_pixels):
    image[point[0], point[1]] = pixel
for point, pixel in zip(test_points, y_pred):
    image[point[0], point[1]] = pixel
```

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

```python
# show the final image

image = (image * 255).astype('uint8')
io.imshow(image)
```

```
<matplotlib.image.AxesImage at 0x7faf884094a8>
```




```
# model evaluation

from sklearn import metrics

MSE = metrics.mean_squared_error(y_test, y_pred)
print('Mean Squared Error(MSE)=', MSE)
```

```
Mean Squared Error(MSE)= 0.0054287955890030375
```

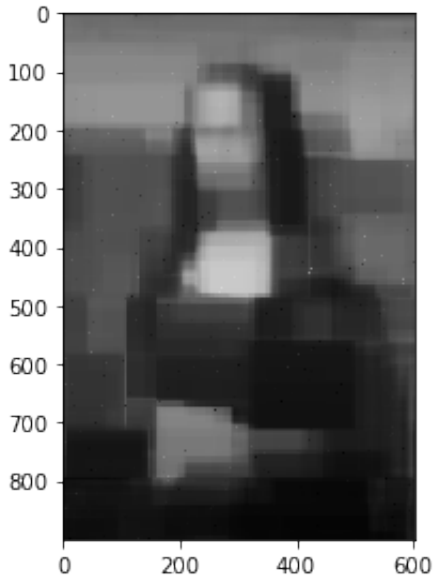
2.b+c+d Working on a gray image

```
# import modules

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import cv2
import os
from skimage import
data,filters,io,transform,feature,segmentation,restoration,util,color
```

```
# use gray image

img = io.imread('data/monalisa_ori.jpg', as_gray=True) #grayscale
img = np.array(img, dtype=np.float32)

# meshgrid, get coordinate locations
x = np.arange(0, img.shape[1])
```

```
y = np.arange(0, img.shape[0])
locations = np.meshgrid(y,x)
locations = np.stack(locations, axis=-1).reshape(-1, 2)

# sample points: uniformly sample 5,000 random (x,y) coordinate locations, to
build a training set

num_sample_points = 5000
np.random.shuffle(locations)
sample_points = locations[:num_sample_points]
test_points = locations[num_sample_points:]

# pixels match
sample_point_pixels = np.array([img[x[0], x[1]] for x in sample_points],
dtype='float32')
test_point_pixels = np.array([img[x[0], x[1]] for x in test_points],
dtype='float32')

# normalize the coordinates
normalized_sample_points = sample_points.copy().astype('float32')
normalized_test_points = test_points.copy().astype('float32')
normalized_sample_points[:, 0] /= img.shape[0]
normalized_sample_points[:, 1] /= img.shape[1]
normalized_test_points[:, 0] /= img.shape[0]
normalized_test_points[:, 1] /= img.shape[1]

# use Random Forest Regressor
from sklearn.ensemble import RandomForestRegressor
x_train = normalized_sample_points
y_train = sample_point_pixels
x_test = normalized_test_points
y_test = test_point_pixels

rfr = RandomForestRegressor(n_estimators=50, max_depth=8, criterion='mse')
rfr.fit(x_train, y_train)
y_pred = rfr.predict(x_test)

# combine points and sample points
image = np.zeros_like(img)
for point, pixel in zip(sample_points, sample_point_pixels):
    image[point[0], point[1]] = pixel
for point, pixel in zip(test_points, y_pred):
    image[point[0], point[1]] = pixel

# show the final image
image = (image * 255).astype('uint8')
io.imshow(image)
```

```
<matplotlib.image.AxesImage at 0x7f81404af5f8>
```




```
# model evaluation

from sklearn import metrics

MSE = metrics.mean_squared_error(y_test, y_pred)
print('Mean Squared Error(MSE)=', MSE)
```

```
Mean Squared Error(MSE)= 0.0057140428558273984
```

2.e Experimentation

2.e.i Random Forest Experiment 1

The experiment's codes and results are shown below.

According to the experiment results (MSE scores and final images), we found that:

- At first, with the depth increases (when max_depth=[1,2,3,5,10]), the performance (i.e. MSE) of the random forest model is also improved correspondingly and the final image becomes increasingly closer to the original image.
- But, when max_depth = 15, the performance of the random forest model decreases comparing to that of the max_depth=10.

Reason:

- The deeper the tree structure is, the better its ability to fit the data, but it may also lead to overfitting.
- Without restriction, a decision tree will grow until no more features are available. Such

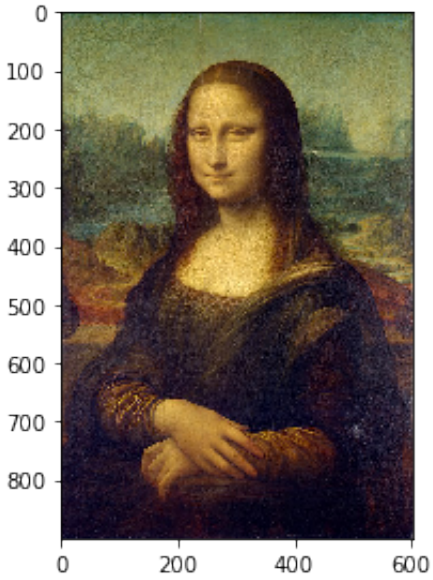
decision trees tend to overfit.

- max_depth is used to limit the maximum depth of the tree, branches above the set depth will be cutted. Limiting tree depth can effectively limit overfitting.

```
%run 1.Q2_b+c_ColorImage.py
```

```
(900, 604, 3)
3
[[[103 121  97]
  [ 79  87  66]
  [112 113  97]
  ...
  [ 85  91  79]
  [116 117 111]
  [ 97  93  90]]

 [[110 128 102]
  [107 115  94]
  [100 101  85]
  ...
  [ 71  77  63]
  [ 84  85  77]
  [ 60  59  54]]]
[[459   3]
 [892 353]
 [ 64 411]
 [650 515]
 [740  61]
 [ 38 422]
 [699 454]
 [887 516]
 [749 208]
 [574 121]]
[[0.41568628 0.30980393 0.3372549 ]
 [0.03921569 0.01960784 0.10980392]
 [0.49019608 0.5254902  0.31764707]
 [0.15294118 0.14117648 0.22352941]
 [0.03529412 0.02352941 0.13725491]]
[[0.07450981 0.06666667 0.16078432]
 [0.27450982 0.33333334 0.21960784]
 [0.2509804  0.1882353  0.23137255]
 [0.20392157 0.29411766 0.22352941]
 [0.10588235 0.05882353 0.14509805]]
[[0.51       0.00496689]
 [0.9911111  0.5844371 ]
 [0.07111111 0.68046355]
 [0.7222222  0.85264903]
 [0.82222223 0.10099338]]
```

```
[[0.8388889  0.8509934 ]
 [0.36666667 0.7731788 ]
 [0.63555557 0.24006623]
 [0.35333332 0.7682119 ]
 [0.99       0.43543047]]
```

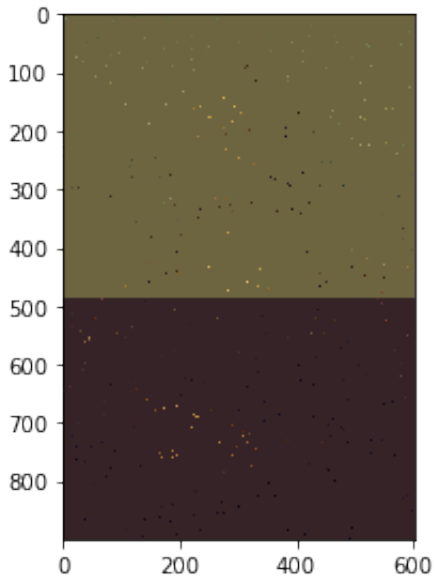



```
# Experiment: set n_estimators=1, max_depth=[1,2,3,5,10,15]

from sklearn.ensemble import RandomForestRegressor
from sklearn import metrics

class random_forest_experimentation:

    def __init__(self):
        self.a=1

    def rfr_model(self,n,d):
        rfr = RandomForestRegressor(n_estimators=n, max_depth=d,
criterion='mse')
        rfr.fit(x_train, y_train)
        y_pred = rfr.predict(x_test)
        return y_pred

    def calculate_mse(self,y_pred):
        MSE = metrics.mean_squared_error(y_test, y_pred)
        return MSE

    def final_image(self,y_pred):
        # combine points and sample points
        image = np.zeros_like(img)
        for point, pixel in zip(sample_points, sample_point_pixels):
            image[point[0], point[1]] = pixel
        for point, pixel in zip(test_points, y_pred):
```

```
            image[point[0], point[1]] = pixel
        # show the final image
        image = (image * 255).astype('uint8')
        io.imshow(image)
```

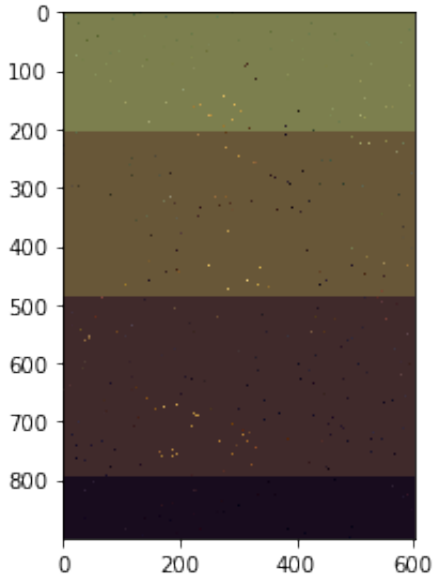
```
# set n_estimators=1, max_depth=1

rfr_ex = random_forest_experimentation()
y_pred = rfr_ex.rfr_model(1,1)
MSE1 = rfr_ex.calculate_mse(y_pred)
print ("MSE=", MSE1)
rfr_ex.final_image(y_pred)
```

```
MSE= 0.023464145858369253
```



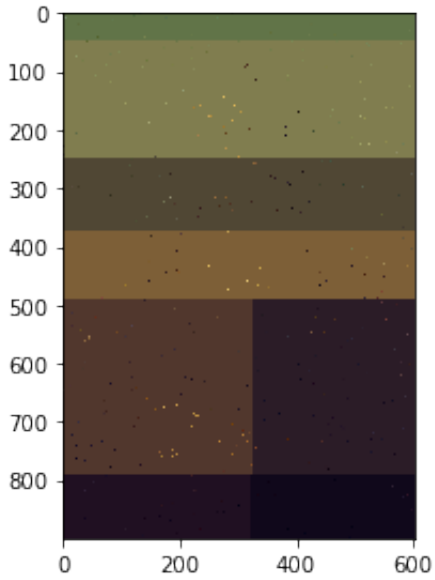

```
# set n_estimators=1, max_depth=2

rfr_ex = random_forest_experimentation()
y_pred = rfr_ex.rfr_model(1,2)
MSE2 = rfr_ex.calculate_mse(y_pred)
print ("MSE=", MSE2)
rfr_ex.final_image(y_pred)
```

```
MSE= 0.02035360798607523
```



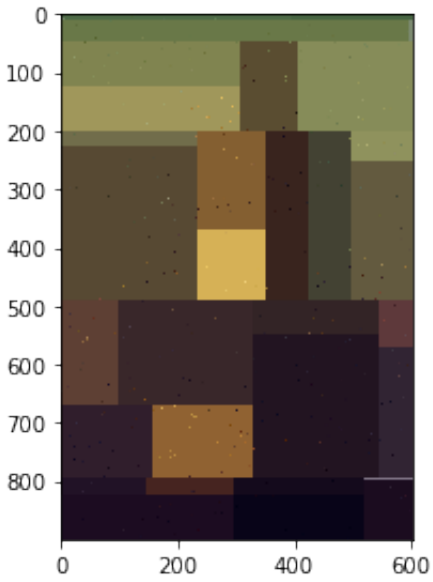

```
# set n_estimators=1, max_depth=3

rfr_ex = random_forest_experimentation()
y_pred = rfr_ex.rfr_model(1,3)
MSE3 = rfr_ex.calculate_mse(y_pred)
print ("MSE=", MSE3)
rfr_ex.final_image(y_pred)
```

```
MSE= 0.01874219369715135
```



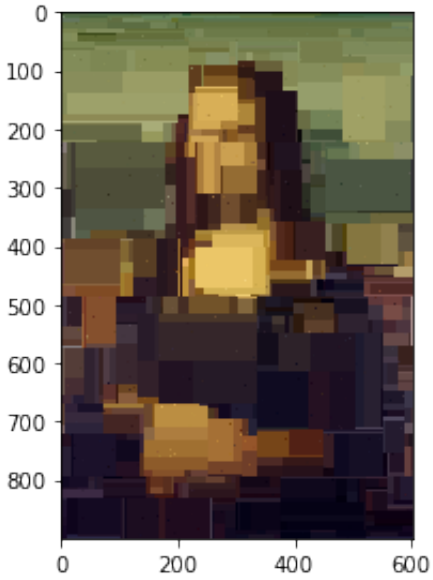
```
# set n_estimators=1, max_depth=5

rfr_ex = random_forest_experimentation()
y_pred = rfr_ex.rfr_model(1,5)
MSE4 = rfr_ex.calculate_mse(y_pred)
print ("MSE=", MSE4)
rfr_ex.final_image(y_pred)
```

```
MSE= 0.011496722490643761
```



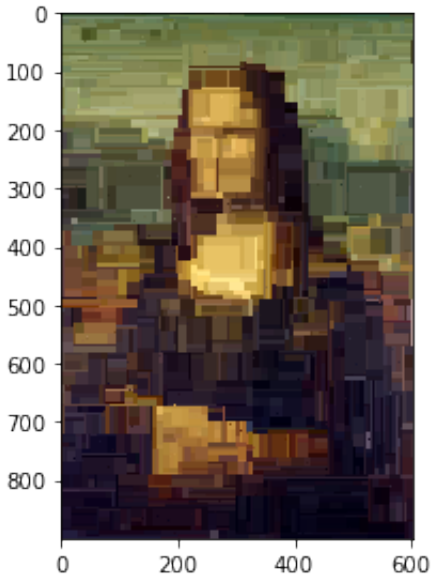

```
# set n_estimators=1, max_depth=10

rfr_ex = random_forest_experimentation()
y_pred = rfr_ex.rfr_model(1,10)
MSE5 = rfr_ex.calculate_mse(y_pred)
print ("MSE=", MSE5)
rfr_ex.final_image(y_pred)
```

```
MSE= 0.007258782597167338
```

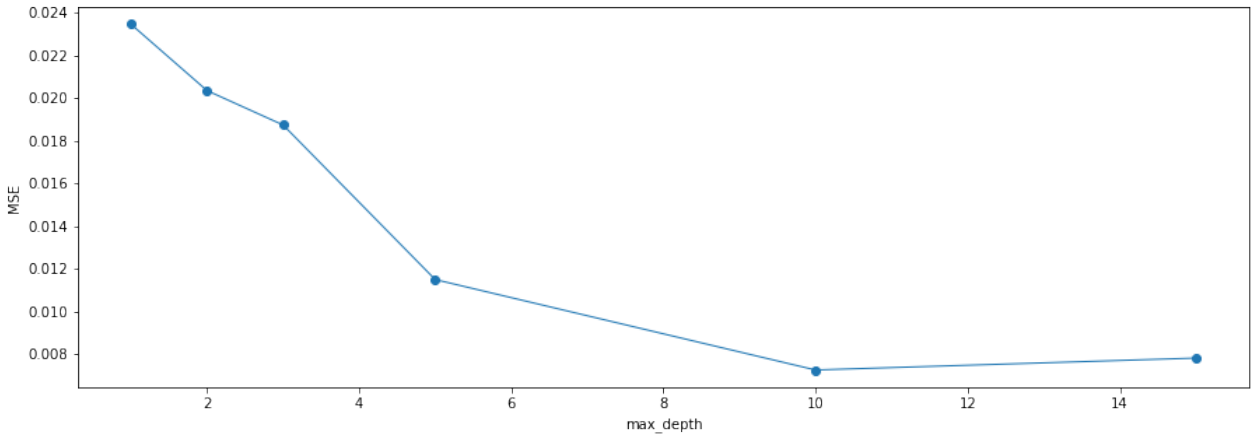



```
# set n_estimators=1, max_depth=15

rfr_ex = random_forest_experimentation()
y_pred = rfr_ex.rfr_model(1,15)
MSE6 = rfr_ex.calculate_mse(y_pred)
print ("MSE=", MSE6)
rfr_ex.final_image(y_pred)
```

```
MSE= 0.007816664587058429
```



```
# plot mse trend
# In this experiment: when n_estimators=1, best max_depth=10

a = [1,2,3,5,10,15]
b = [MSE1,MSE2,MSE3,MSE4,MSE5,MSE6]
plt.figure(figsize=(15,5))
plt.plot(a,b,'o-',linewidth=1)
plt.xlabel("max_depth")
plt.ylabel("MSE")
plt.show()
```



2.e.ii Random Forest Experiment 2

The experiment's codes and results are shown below.

According to the experiment results (MSE scores and final images), we found that:

- With the tree number increases(when n_estimators=[1,3,5,10]), the performance (i.e. MSE) of the random forest model is also significantly improved and the final image becomes increasingly closer to the original image.
- But the model performance with n_estimators=100 doesn't significantly better than the model performance with n_estimators=10.

Reason:

- Generally, the more trees there are, the better the model's performance is.
- However, when the number of trees exceeds a certain value, the error rate of the model tends to converge. At this point, if we continue to increase the number of trees, it will not improve the model performance, instead, it will reduce the training speed.
- Therefore, the number of decision trees doesn't need to be too large, choosing a comparatively large number is enough.

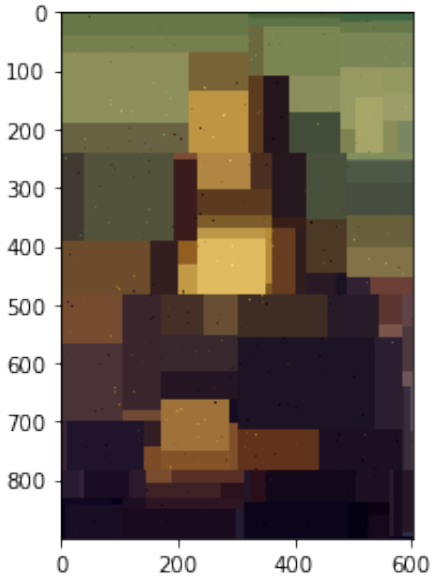
```
%run 1.Q2_b+c_ColorImage.py
```

```
from sklearn.ensemble import RandomForestRegressor
from sklearn import metrics
```

```
class random_forest_experimentation:

    def __init__(self):
        self.a=1

    def rfr_model(self,n):
        rfr = RandomForestRegressor(n_estimators=n, max_depth=7,
criterion='mse')
        rfr.fit(x_train, y_train)
        y_pred = rfr.predict(x_test)
        return y_pred

    def calculate_mse(self,y_pred):
        MSE = metrics.mean_squared_error(y_test, y_pred)
        return MSE

    def final_image(self,y_pred):
        # combine points and sample points
        image = np.zeros_like(img)
        for point, pixel in zip(sample_points, sample_point_pixels):
            image[point[0], point[1]] = pixel
        for point, pixel in zip(test_points, y_pred):
            image[point[0], point[1]] = pixel
        # show the final image
        image = (image * 255).astype('uint8')
        io.imshow(image)
```

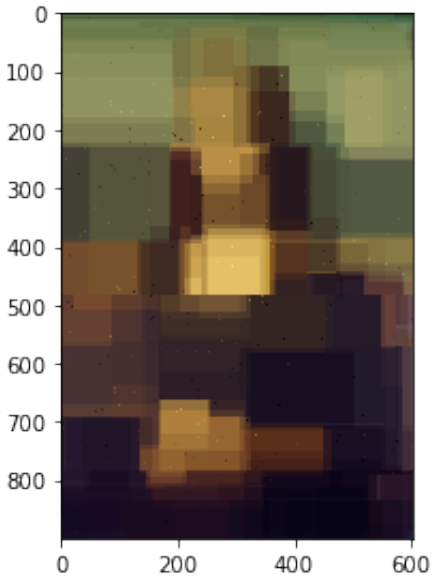
```
# set n_estimators=1, max_depth=7

rfr_ex = random_forest_experimentation()
y_pred = rfr_ex.rfr_model(1)
MSE1 = rfr_ex.calculate_mse(y_pred)
print ("MSE=", MSE1)
rfr_ex.final_image(y_pred)
```

```
MSE= 0.007938261974141865
```



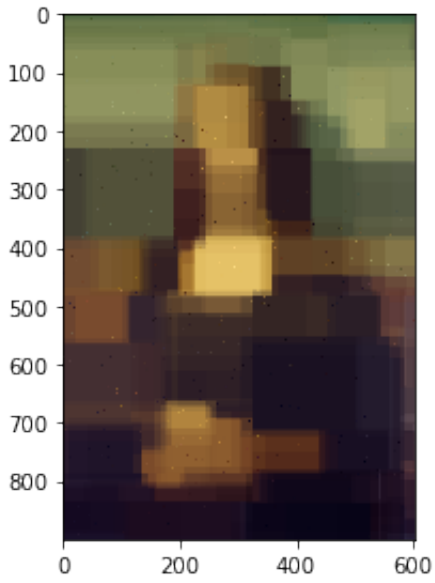

```
# set n_estimators=3, max_depth=7

rfr_ex = random_forest_experimentation()
y_pred = rfr_ex.rfr_model(3)
MSE2 = rfr_ex.calculate_mse(y_pred)
print ("MSE=", MSE2)
rfr_ex.final_image(y_pred)
```

```
MSE= 0.007180770156637865
```



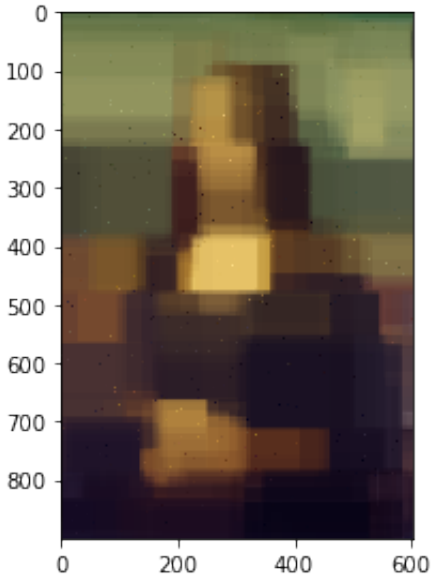
```
# set n_estimators=5, max_depth=7

rfr_ex = random_forest_experimentation()
y_pred = rfr_ex.rfr_model(5)
MSE3 = rfr_ex.calculate_mse(y_pred)
print ("MSE=", MSE3)
rfr_ex.final_image(y_pred)
```

```
MSE= 0.0068234340318858375
```



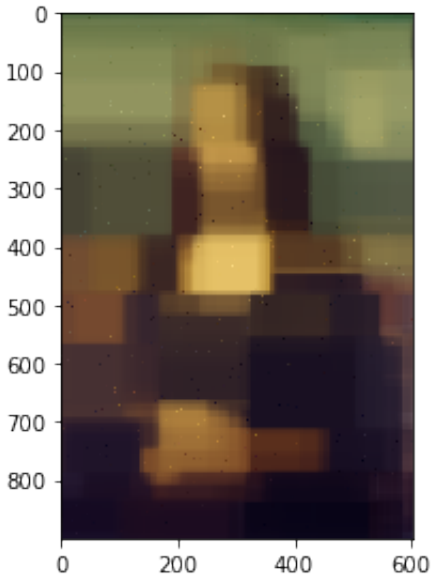

```
# set n_estimators=10, max_depth=7

rfr_ex = random_forest_experimentation()
y_pred = rfr_ex.rfr_model(10)
MSE4 = rfr_ex.calculate_mse(y_pred)
print ("MSE=", MSE4)
rfr_ex.final_image(y_pred)
```

```
MSE= 0.006556347669496025
```

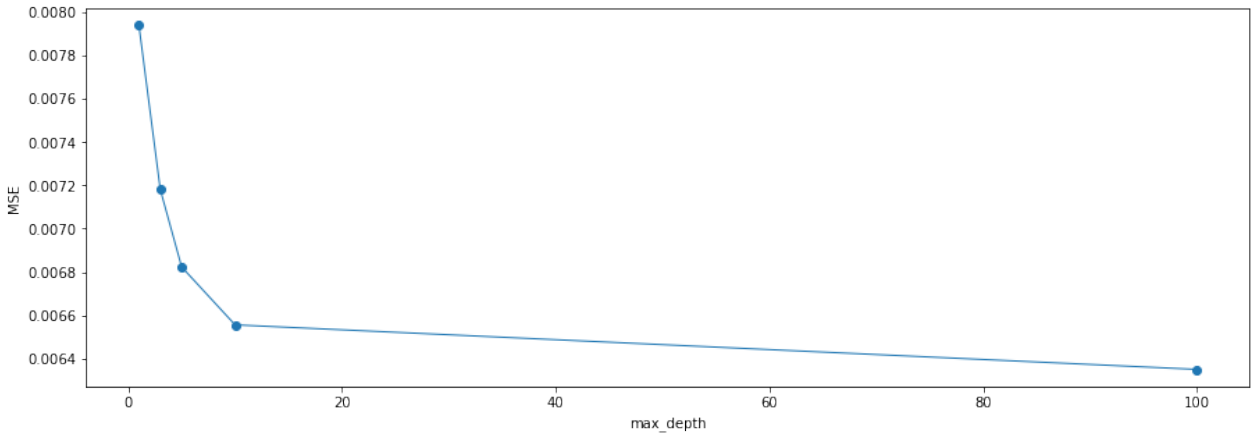



```
# set n_estimators=100, max_depth=7

rfr_ex = random_forest_experimentation()
y_pred = rfr_ex.rfr_model(100)
MSE5 = rfr_ex.calculate_mse(y_pred)
print ("MSE=", MSE5)
rfr_ex.final_image(y_pred)
```

```
MSE= 0.006350843014001221
```



```
# plot mse trend
# In this experiment: when max_depth=7, best n_estimators=100

a = [1,3,5,10,100]
b = [MSE1,MSE2,MSE3,MSE4,MSE5]
plt.figure(figsize=(15,5))
plt.plot(a,b,'o-',linewidth=1)
plt.xlabel("max_depth")
plt.ylabel("MSE")
plt.show()
```



2.e.iii KNN Regressor

The experiment's codes and results are shown below.

Compare and contrast the outlook: why does this look the way it does?

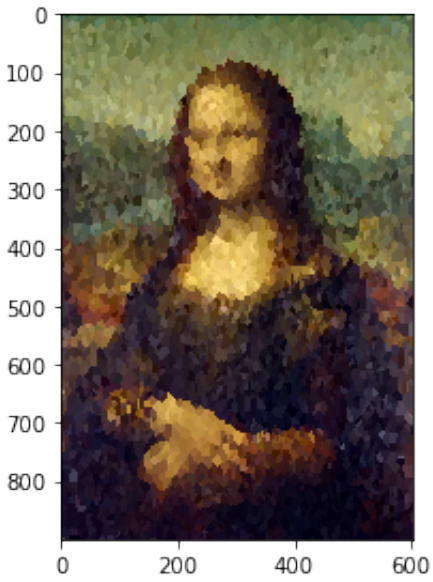
- The resulting image of Random Forest Regressor has many **obvious dividing lines**, which is very consistent with the characteristics of decision trees. Because each partition built by the decision tree is a dividing line that divides data points into different nodes based on eigenvalues.
- The resulting image of KNN Regressor don't have obvious dividing lines, instead, the boundaries among its categories are quite **blurred**. This is also very consistent with the characteristics of KNN model, because the principle of KNN is to take the average of the closest points as the value of this point.

```
%run 1.Q2_b+c_ColorImage.py
```

```
# use KNeighbors Regressor, set k=1
from sklearn.neighbors import KNeighborsRegressor

knr = KNeighborsRegressor(n_neighbors=1)
knr.fit(x_train,y_train)
y_pred = knr.predict(x_test)
```

```
# combine points and sample points
image = np.zeros_like(img)
for point, pixel in zip(sample_points, sample_point_pixels):
    image[point[0], point[1]] = pixel
for point, pixel in zip(test_points, y_pred):
    image[point[0], point[1]] = pixel

# show the final image
image = (image * 255).astype('uint8')
io.imshow(image)
```

```
<matplotlib.image.AxesImage at 0x7fbefae6aef0>
```




```
# model evaluation

from sklearn import metrics

MSE = metrics.mean_squared_error(y_test, y_pred)
print('Mean Squared Error(MSE)=', MSE)
```

```
Mean Squared Error(MSE)= 0.006142866
```

2.e.iv Experiment with different pruning strategies

The experiment's codes and results are shown below.

In order to increase the generalization of decision trees, I used the following pruning strategies.

1.**min_samples_leaf**:

- min_samples_leaf limits the minimum number of samples of leaf nodes. If the number of leaf nodes is less than the sample number, the leaves will be pruned together with their siblings. It can guarantee the minimum size of each leaf and avoid the occurrence of low-variance and overfitting leaf nodes in regression problems.
- My experiment result shows that the best min_samples_leaf is 1.

2.**min_samples_split**:

- min_samples_split limits the conditions under which subtrees can continue to divide. When the number of samples on a node is less than the value, the tree stops growing. So, setting a larger S also serves the function of pruning.
- My experiment result shows that the best min_samples_split is 7.

```
%run 1.Q2_b+c_ColorImage.py
```

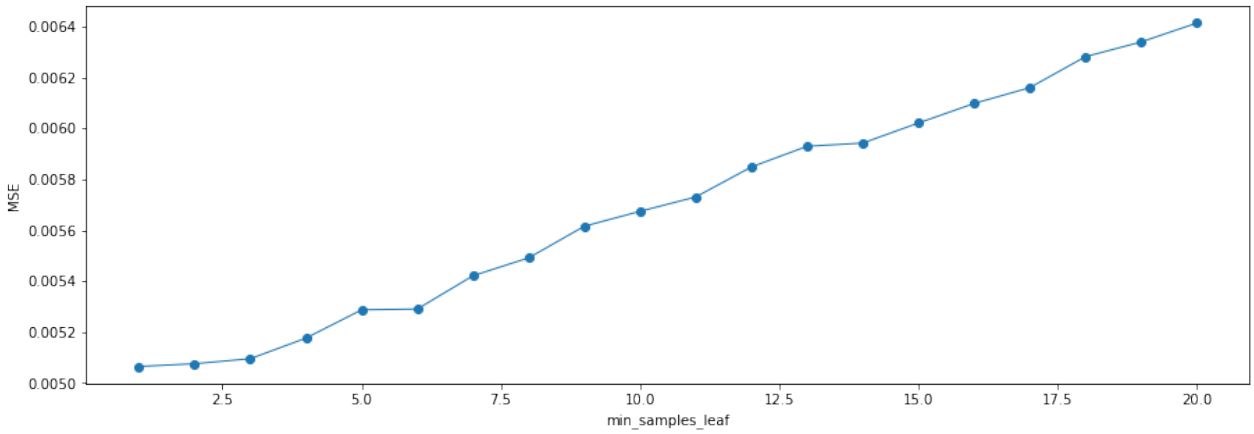
```
# Pruning Experiment 1 - Find the best min_samples_leaf

from sklearn.ensemble import RandomForestRegressor
from sklearn import metrics

class random_forest_experimentation_1:

    def __init__(self):
        self.a=1

    def cal_pred_list(self):
        pred_list = []
        param_list = list(range(1,21))

        for p in param_list:
            #use the best n_estimators and max_depth from previous experiments
in (e)
            rfr = RandomForestRegressor(min_samples_leaf=p,
                                        n_estimators=50,
                                        max_depth=8,
                                        criterion='mse')
            rfr.fit(x_train, y_train)
            y_pred = rfr.predict(x_test)
            pred_list.append(y_pred)
        return pred_list,param_list

    def cal_mse_list(self,pred_list):
        mse_list = []

        for i in pred_list:
            mse = metrics.mean_squared_error(y_test, i)
            mse_list.append(mse)
```

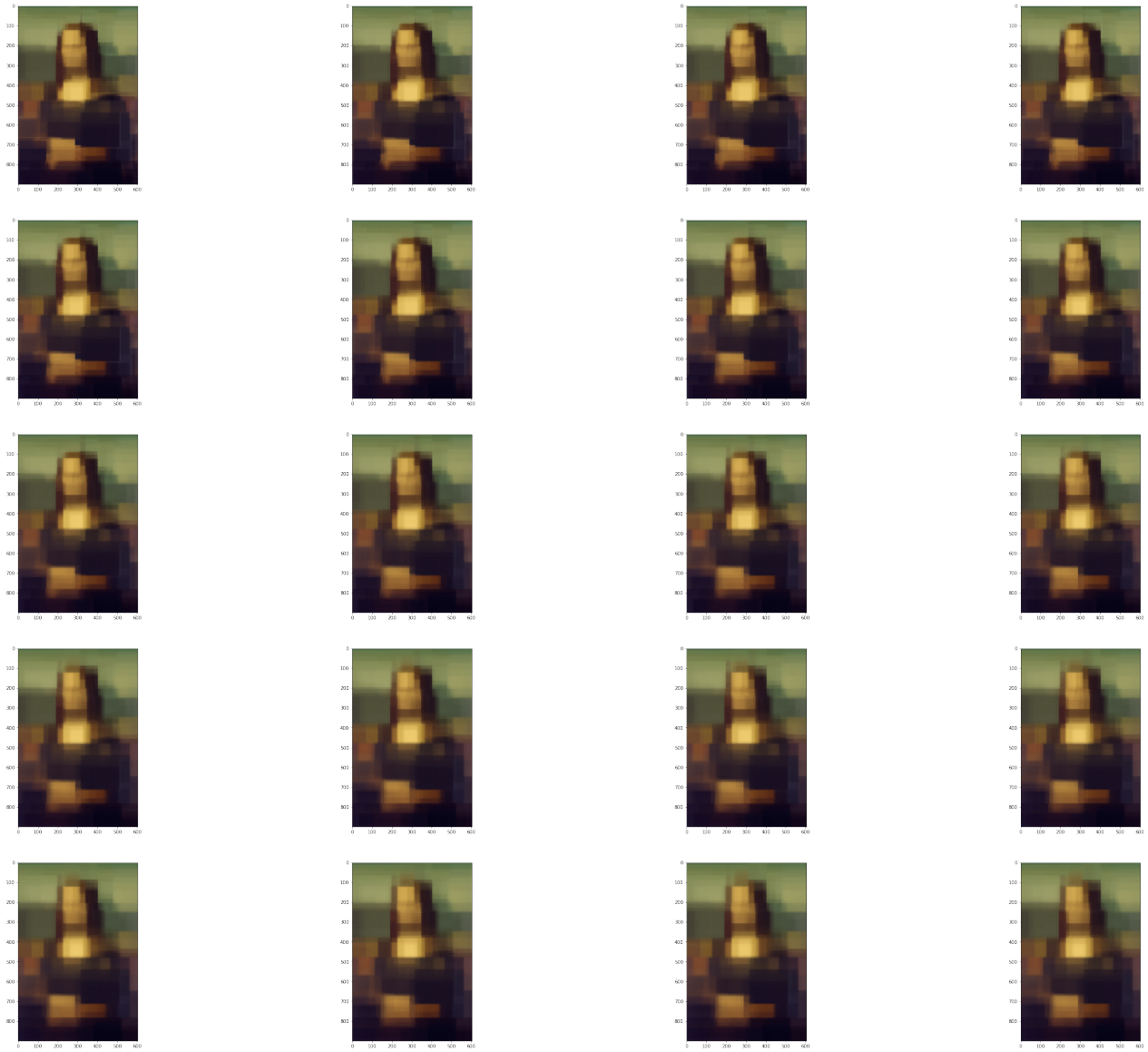
```
        return mse_list

    def cal_min_mse(self,mse_list,param_list):
        min_mse = min(mse_list)
        index = mse_list.index(min_mse)
        param = param_list[index]
        return min_mse,param

    def cal_img_list(self,pred_list):
        img_list = []

        for j in pred_list:
            # combine points and sample points
            image = np.zeros_like(img)
            for point, pixel in zip(sample_points, sample_point_pixels):
                image[point[0], point[1]] = pixel
            for point, pixel in zip(test_points, j):
                image[point[0], point[1]] = pixel
            # show the final image
            image = (image * 255).astype('uint8')
            img_list.append(image)
            #io.imshow(img_list[0])
        return img_list

    def show_img_collections(self,img_list):
        plt.figure(figsize=(50, 50))
        length = len(img_list)
        for r in range(0,length):
            plt.subplot(int(length/4+1),4,r+1)
            plt.imshow(img_list[r])
        plt.show()

    def mse_plot(self,param_list,mse_list):
        a = param_list
        b = mse_list
        plt.figure(figsize=(15,5))
        plt.plot(a,b,'o-',linewidth=1)
        plt.xlabel("min_samples_leaf")
        plt.ylabel("MSE")
        plt.show()
```

```
rfr_ex_1 = random_forest_experimentation_1()
pred_list,param_list = rfr_ex_1.cal_pred_list()
mse_list = rfr_ex_1.cal_mse_list(pred_list)
print ("min_samples_leaf list=", param_list)
print ("MSE list=", mse_list)

min_mse,param = rfr_ex_1.cal_min_mse(mse_list,param_list)
print ("Minimal MSE=", min_mse)
```

```
print ("the min_samples_leaf value that leads to the Minimal MSE=", param)

rfr_ex_1.mse_plot(param_list,mse_list)

img_list = rfr_ex_1.cal_img_list(pred_list)
rfr_ex_1.show_img_collections(img_list)
```

```
min_samples_leaf list= [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20]
MSE list= [0.005065197845615978, 0.005075971517852472, 0.005095737005531484,
0.005176551759823216, 0.0052877266462888275, 0.005290745042943867,
0.005421524214701545, 0.005491734779493454, 0.005615895419724165,
0.005674535964108541, 0.005730764026031047, 0.005848699628924443,
0.005929948547243868, 0.005942554241871655, 0.006021332810625519,
0.006098064496280067, 0.006160194478391557, 0.0062818007425528,
0.006339647387255588, 0.006413966489991174]
Minimal MSE= 0.005065197845615978
the min_samples_leaf value that leads to the Minimal MSE= 1
```





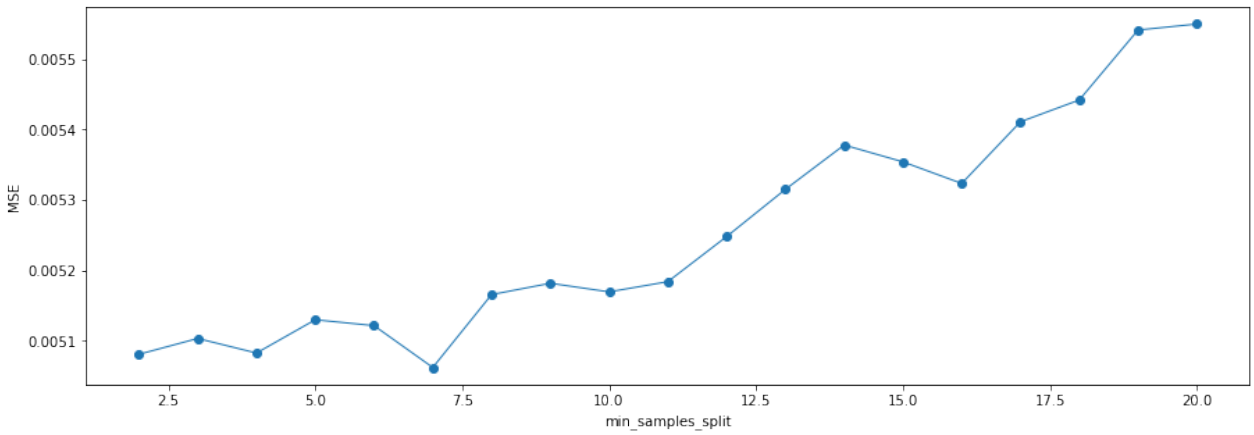
```
# Pruning Experiment 2 - Find the best min_samples_split

from sklearn.ensemble import RandomForestRegressor
from sklearn import metrics

class random_forest_experimentation_2:

    def __init__(self):
        self.a=1

    def cal_pred_list(self):
        pred_list = []
        param_list = list(range(2,21))

        for p in param_list:
            rfr = RandomForestRegressor(min_samples_split=p,
                                        min_samples_leaf=1, #use the best
min_samples_leaf value from pruning experiment 1
```

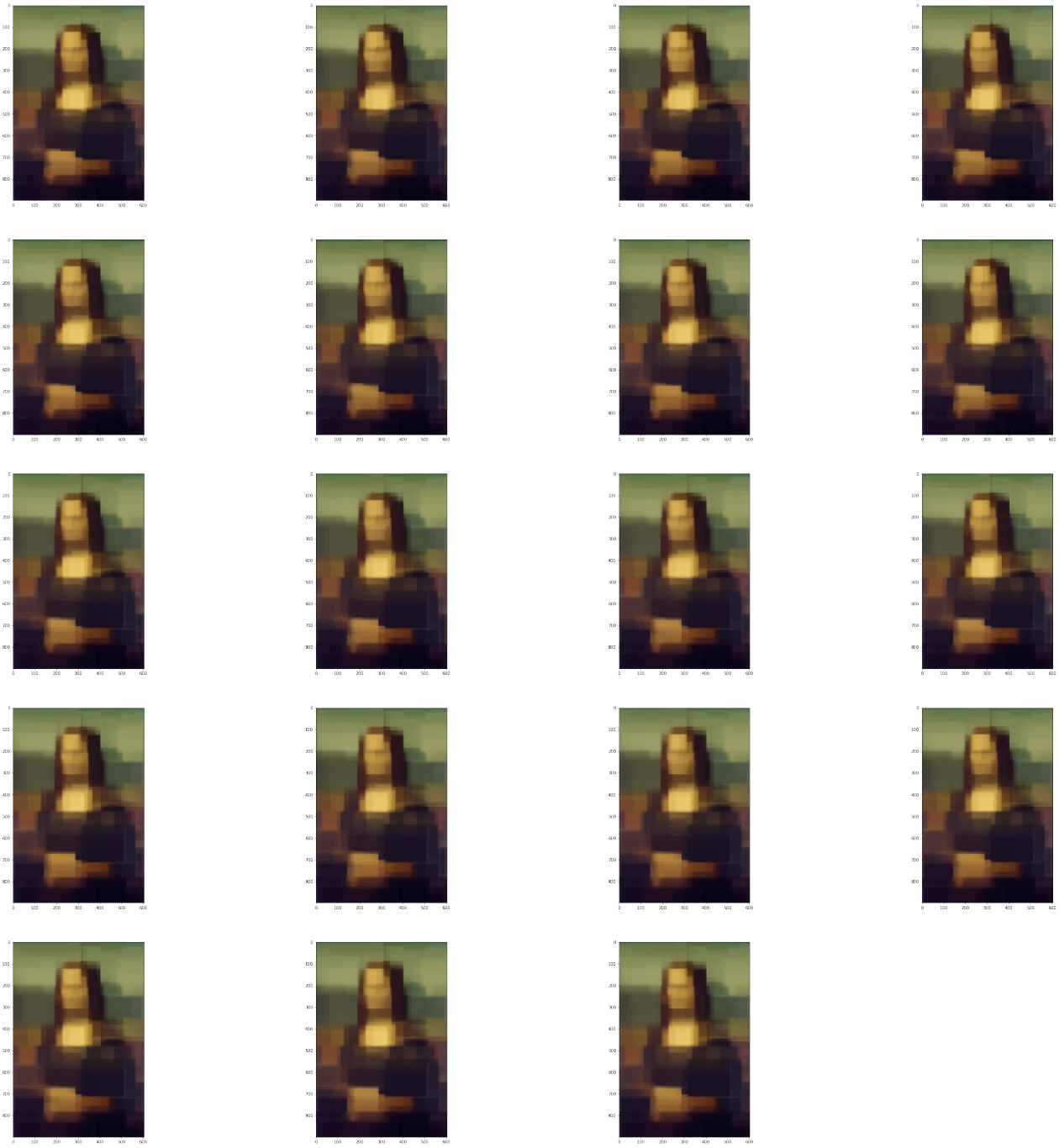
```
                                        n_estimators=50,
                                        max_depth=8,
                                        criterion='mse')
            rfr.fit(x_train, y_train)
            y_pred = rfr.predict(x_test)
            pred_list.append(y_pred)
        return pred_list,param_list

    def cal_mse_list(self,pred_list):
        mse_list = []

        for i in pred_list:
            mse = metrics.mean_squared_error(y_test, i)
            mse_list.append(mse)
        return mse_list

    def cal_min_mse(self,mse_list,param_list):
        min_mse = min(mse_list)
        index = mse_list.index(min_mse)
        param = param_list[index]
        return min_mse,param

    def cal_img_list(self,pred_list):
        img_list = []

        for j in pred_list:
            # combine points and sample points
            image = np.zeros_like(img)
            for point, pixel in zip(sample_points, sample_point_pixels):
                image[point[0], point[1]] = pixel
            for point, pixel in zip(test_points, j):
                image[point[0], point[1]] = pixel
            # show the final image
            image = (image * 255).astype('uint8')
            img_list.append(image)
            #io.imshow(img_list[0])
        return img_list

    def show_img_collections(self,img_list):
        plt.figure(figsize=(50, 50))
        length = len(img_list)
        for r in range(0,length):
            plt.subplot(int(length/4+1),4,r+1)
            plt.imshow(img_list[r])
        plt.show()

    def mse_plot(self,param_list,mse_list):
        a = param_list
        b = mse_list
```

```
        plt.figure(figsize=(15,5))
        plt.plot(a,b,'o-',linewidth=1)
        plt.xlabel("min_samples_split")
        plt.ylabel("MSE")
        plt.show()
```

```
rfr_ex_2 = random_forest_experimentation_2()
pred_list,param_list = rfr_ex_2.cal_pred_list()
mse_list = rfr_ex_2.cal_mse_list(pred_list)
print ("min_samples_split list=", param_list)
print ("MSE list=", mse_list)

min_mse,param = rfr_ex_2.cal_min_mse(mse_list,param_list)
print ("Minimal MSE=", min_mse)
print ("the min_samples_split value that leads to the Minimal MSE=", param)

rfr_ex_2.mse_plot(param_list,mse_list)

img_list = rfr_ex_2.cal_img_list(pred_list)
rfr_ex_2.show_img_collections(img_list)
```

```
min_samples_split list= [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20]
MSE list= [0.005080948720796094, 0.005103460822964675, 0.005082757316573325,
0.005130005115810771, 0.005121683566921728, 0.005062248368407982,
0.005165740087461685, 0.005181846710163913, 0.005169605168117714,
0.005184116948579885, 0.0052481850682677515, 0.005315210539290803,
0.005378149474865756, 0.005354265125755791, 0.0053235771420690305,
0.005411184187239778, 0.00544212548361325, 0.005541448427207736,
0.005549983067078493 5]
Minimal MSE= 0.005062248368407982
the min_samples_split value that leads to the Minimal MSE= 7
```



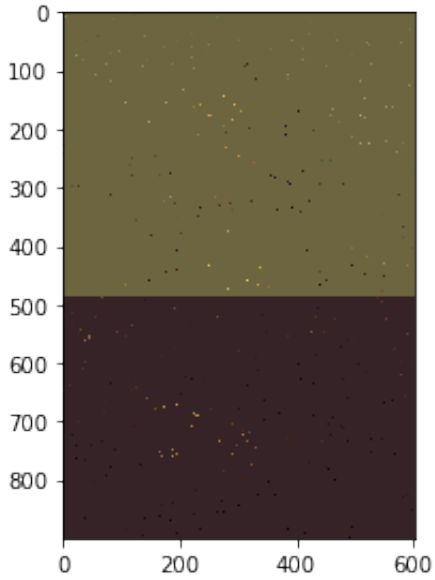


2.f Analysis

2-f-i

Chosen decision tree: single decision tree, tree depth=1.

The image below is the resulting image of this decision tree.



From the image, we can see that there is one obvious dividing line which splits two colors.

Assuming that:

- coordinate locations = (x,y);
- the value of the color of a coordinate location is C;
- the values of the two colors in the image are C1 and C2, respectively;
- the y coordinate of the dividing line is K.

The formula:

$$C = \begin{cases} C_1 & 0 \leq y < K \\ C_2 & K \leq y \leq 900 \end{cases}$$

2-f-ii

1.Why does the resulting image look the way it does:

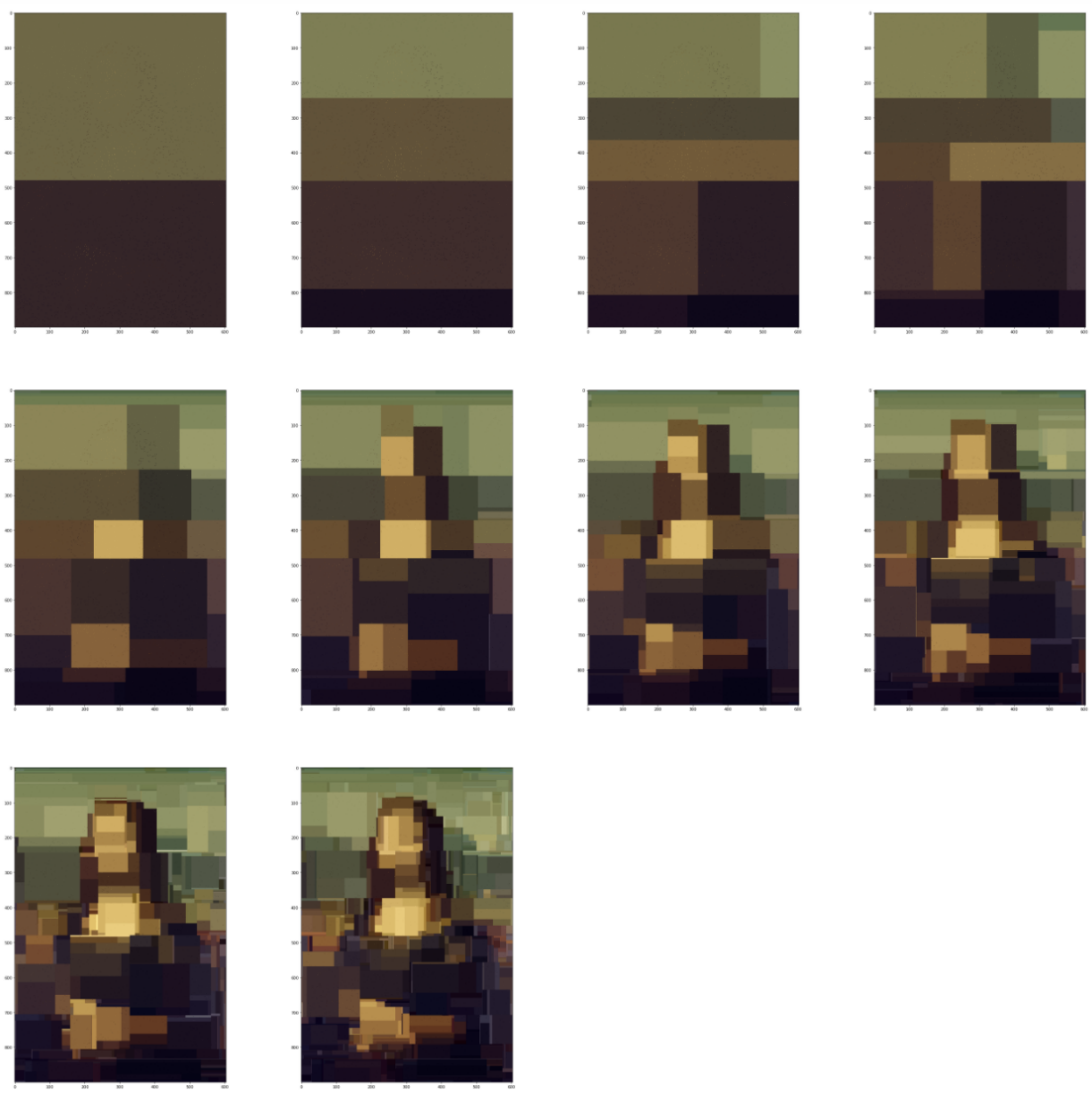
The resulting image of Random Forest Regressor has many obvious dividing lines, which is very consistent with the characteristics of decision trees. Because each partition built by the decision tree is a dividing line that divides data points into different nodes based on eigenvalues.

2.What shape are the patches of color, and how are they arranged:

- The shape of the patches of color is rectangular.
- They are arranged horizontally or vertically according to the color distribution of the original image.
- Under the condition of the same number of decision trees, the deeper the tree structures are , the more rectangles there are in the image, and the more close the image outlook is to the original.

2-f-iii

Figure: the resulting images of Random Forest Regressor contains a single decision tree but with different depths range from 1 to 10.



Assuming that:

- Y: the number of patches of color in the resulting image of the forest that contains a single decision tree;
- d: tree depth.

Then we can get the equation:

$$Y = 2^d$$

2-f-iv

Assuming that:

- Y: the number of patches of color in the resulting image of the forest;
- n: the number of decision trees;
- d: tree depth.

Then we get the equation:

$$Y = 2^{(n*d)}$$

Explanation:

- Because random forest regresion is the average of N trees, the maximal possible Y of each tree is 2^d, so the average maximal possible Y of N trees should be $2^{(n*d)}$.
- The Y here is the upper bound, namely the highest possibility. But the actual Y in real situations can't usually achieve this value, because there's no guarantee that the decision tree will be full or that every tree will have the same number of split nodes.

Written Problem 2

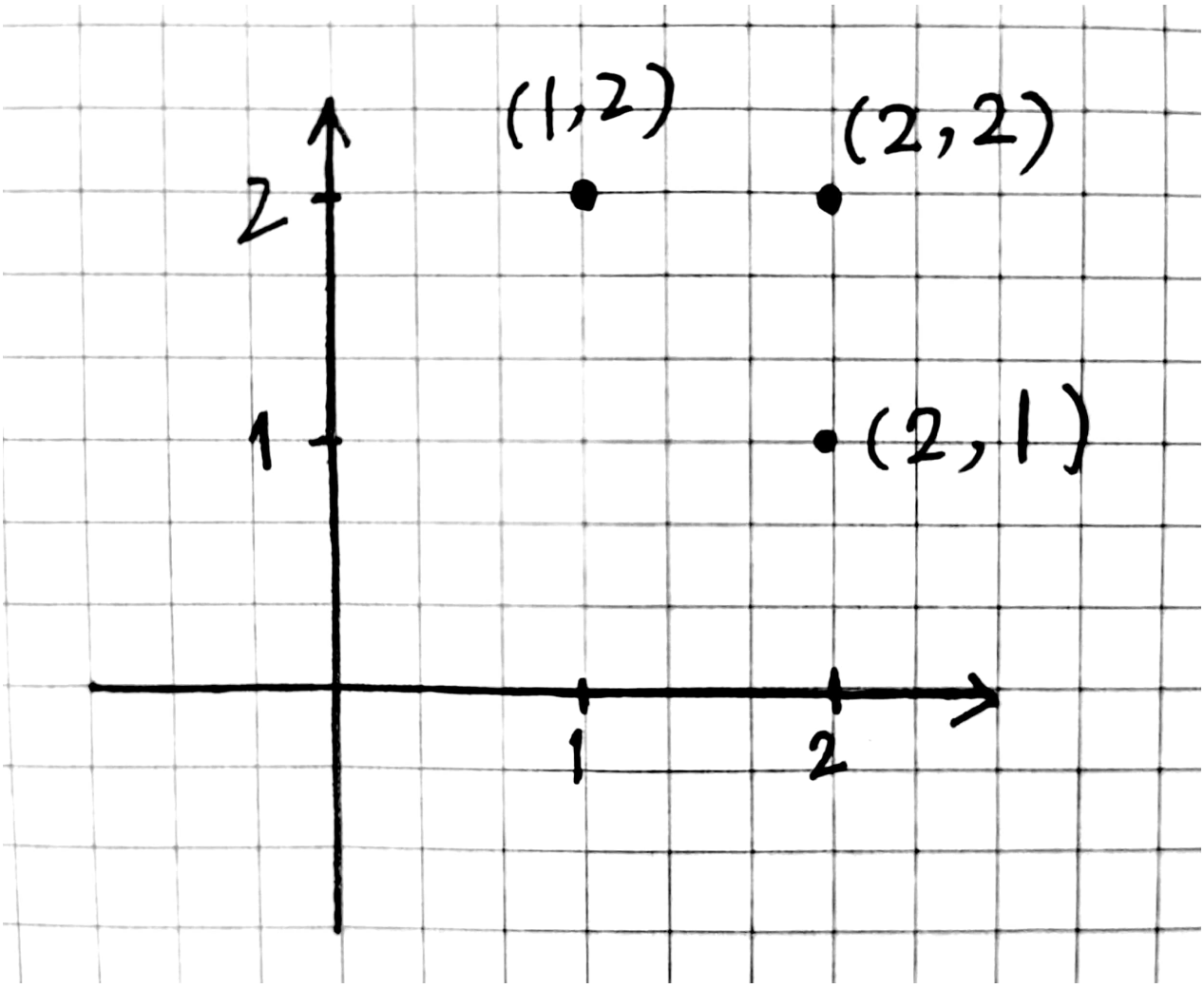
Question i

A.

1.No, it's impossible to create an example such that X^1 and X^2 are not separable by decision stump.

2.Yes, adding another data-point can make it possible to create an example such that X^1 and X^2 are not separable by decision stump.

For example: See the diagram below. Two points and three points can both be separated with Linear Classifier.



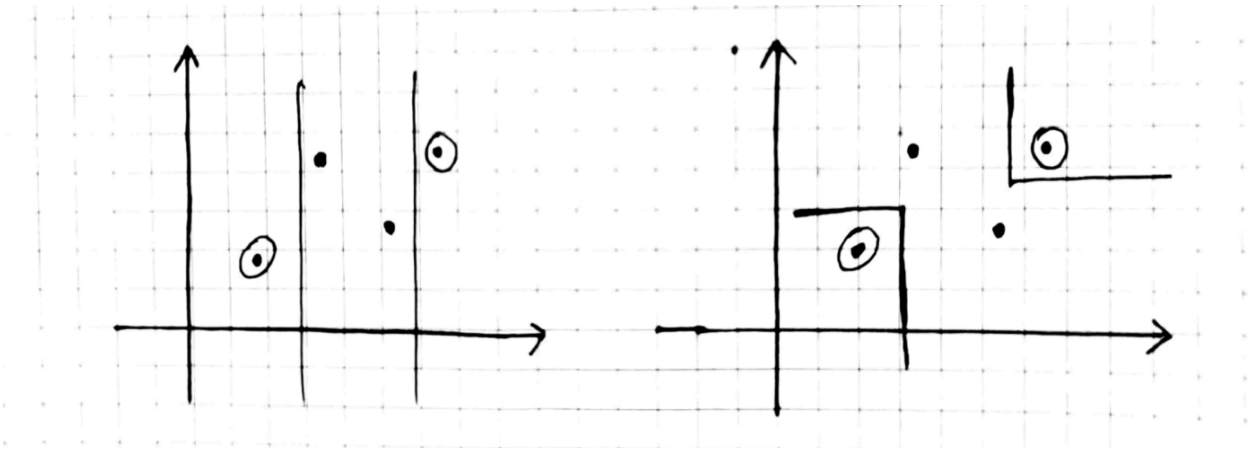
B.

No, It's impossible to add more data-points to the first example such they are not separable by a two-level decision tree.

C.

Yes, it's possible to construct an example such that the data-points are separable by a 2-level decision tree but not by a linear classifier.

For example, see the diagrams below.



Question ii

Considering the preference as binary output variable that we want to predict, which attribute should be at the root of this decision tree? Answer using cross-entropy and Gini index.

Method 1: Using Gini Index

Formulas:

$$Gini(D) = \sum_{k=1}^{|y|} \sum_{k' \neq k} p_k p_{k'} = 1 - \sum_{k=1}^{|y|} p_k^2$$

$$Gini\_index(D, a) = \sum_{v=1}^{V} \frac{|D^v|}{|D|} Gini(D^v)$$

$Gini(Location = Rural) = 1 - (\frac{2}{4})^2 - (\frac{2}{4})^2 = 0.5$

$Gini(Location = Urban) = 1 - (\frac{2}{4})^2 - (\frac{2}{4})^2 = 0.5$

$Gini(Location = Semi - rural) = 1 - (\frac{2}{3})^2 - (\frac{1}{3})^2 = 0.4444444444$

$Gini\_index(Location) = \frac{4}{11} * 0.5 + \frac{4}{11} * 0.5 + \frac{3}{11} * 0.4444444444 = 0.4848484848$

...

$Gini\_index(Pollution) = 0.3939393939$

$Gini\_index(Area) = 0.4909090909$

$Gini\_index(Windows) = 0.3878787879$

Calculation process and results:

Attribute	Value	Yes	No	Number of instances	Gini(Attribute =Value)	Gini_index(Attribute)	Ent(D)	Ent(Attribute=Value)	Ent(Attribute)	Gain(Attribute)
Location	Rural	2	2	4	0.5	0.4848484848	0.9940302115	1	0.9777170457	0.01631316583
	Urban	2	2	4	0.5			1		
	Semi-rural	2	1	3	0.4444444444			0.9182958341		
Pollution	low	2	1	3	0.4444444444	0.3939393939		0.9182958341	0.8404647725	0.1535654389
	med	3	1	4	0.375			0.8112781245		
	high	1	3	4	0.375			0.8112781245		
Area	small	3	3	6	0.5	0.4909090909		1	0.9867957248	**0.007234486725**
	large	3	2	5	0.48			0.9709505945		
Windows	small	2	4	6	0.4444444444	**0.3878787879**		0.9182958341	0.8290377708	0.1649924407
	large	4	1	5	0.32			0.7219280949		

Link: https://docs.google.com/spreadsheets/d/1HmGG7J4gXY1wkCIcYnJDQC_hm_EeCnt6eV_KSx6Z3IA/edit?usp=sharing

> The detailed calculation process of the Info Gain of these attributes is shown in a Google sheet. Please see the sheet screenshot above or open the Google sheet link. Thanks!

Method 2: Using Cross-entropy

Formulas:

$$Ent(D) = -\sum_{k=1}^{|y|} p_k \, log_2 p_k$$

$$Gain(D, a) = Ent(D) - \sum_{v=1}^{V} \frac{|D^v|}{|D|} Ent(D^v)$$

$Ent(D) = -(\frac{6}{11} * log_2(\frac{6}{11}) + \frac{5}{11} * log_2(\frac{5}{11})) = 0.9940302115$

$Ent(Location = Rural) = -(\frac{2}{4} * log_2(\frac{2}{4}) + \frac{2}{4} * log_2(\frac{2}{4})) = 1$

$Ent(Location = Urban) = -(\frac{2}{4} * log_2(\frac{2}{4}) + \frac{2}{4} * log_2(\frac{2}{4})) = 1$

$Ent(Location = Semi - rural) = -(\frac{2}{3} * log_2(\frac{2}{3}) + \frac{1}{3} * log_2(\frac{1}{3})) = 0.9182958341$

$Ent(Location) = \frac{4}{11} * 1 + \frac{4}{11} * 1 + \frac{3}{11} * 0.9182958341 = 0.9777170457$

$Gain(Location) = Ent(D) - Ent(Location) = 0.9940302115 - 0.9777170457 = 0.01631316583$

...

$Gain(Pollution) = 0.1535654389$

$Gain(Area) = 0.007234486725$

$Gain(Windows) = 0.1649924407$

Calculation process and results:

Attribute	Value	Yes	No	Number of instances	Gini(Attribute =Value)	Gini_index(Attribute)	Ent(D)	Ent(Attribute=Value)	Ent(Attribute)	Gain(Attribute)
Location	Rural	2	2	4	0.5	0.4848484848	0.9940302115	1	0.9777170457	0.01631316583
	Urban	2	2	4	0.5			1		
	Semi-rural	2	1	3	0.4444444444			0.9182958341		
Pollution	low	2	1	3	0.4444444444	0.3939393939		0.9182958341	0.8404647725	0.1535654389
	med	3	1	4	0.375			0.8112781245		
	high	1	3	4	0.375			0.8112781245		
Area	small	3	3	6	0.5	0.4909090909		1	0.9867957248	**0.007234486725**
	large	3	2	5	0.48			0.9709505945		
Windows	small	2	4	6	0.4444444444	**0.3878787879**		0.9182958341	0.8290377708	0.1649924407
	large	4	1	5	0.32			0.7219280949		

Link: https://docs.google.com/spreadsheets/d/1HmGG7J4gXY1wkCIcYnJDQC_hm_EeCnt6eV_KSx6Z3IA/edit?usp=sharing

> The detailed calculation process of the Info Gain of these attributes is shown in a Google sheet. Please see the sheet screenshot above or open the Google sheet link above. Thanks!

Final Answer:

According to my calculation results, **the attribute "Windows" has both the minimal Gini Index and the maximal Info Gain among all four attributes. Therefore, the "Windows" attribute should be at the root of this decision tree.**

Written Problem 2

For input $x \in [0, 2)$, output:

$$f(x) = ReLU(2x - 2)$$

For input $x \in [2, 5)$, output

$$f(x) = \frac{1}{3}x + \frac{4}{3}$$

For input $x \in [5, 6)$, output

$$f(x) = 2x - 7$$

For input $x \in [6, 10]$, output

$$f(x) = ReLU(-\frac{5}{3}x + 15)$$

The function is

$$f(x) = 2ReLU(x-1) + (-2+\frac{1}{3})ReLU(x-2) + (-\frac{1}{3}+2)ReLU(x-5) +$$
$$(-2-\frac{5}{3})ReLU(x-6) + (\frac{5}{3})(x-9)$$
$$= 2ReLU(x-1) + (-\frac{5}{3})ReLU(x-2) + \frac{5}{3}ReLU(x-5) +$$
$$(-\frac{11}{3})ReLU(x-6) + \frac{5}{3}(x-9)$$

Thus, we have 1 hidden layer and 5 units.

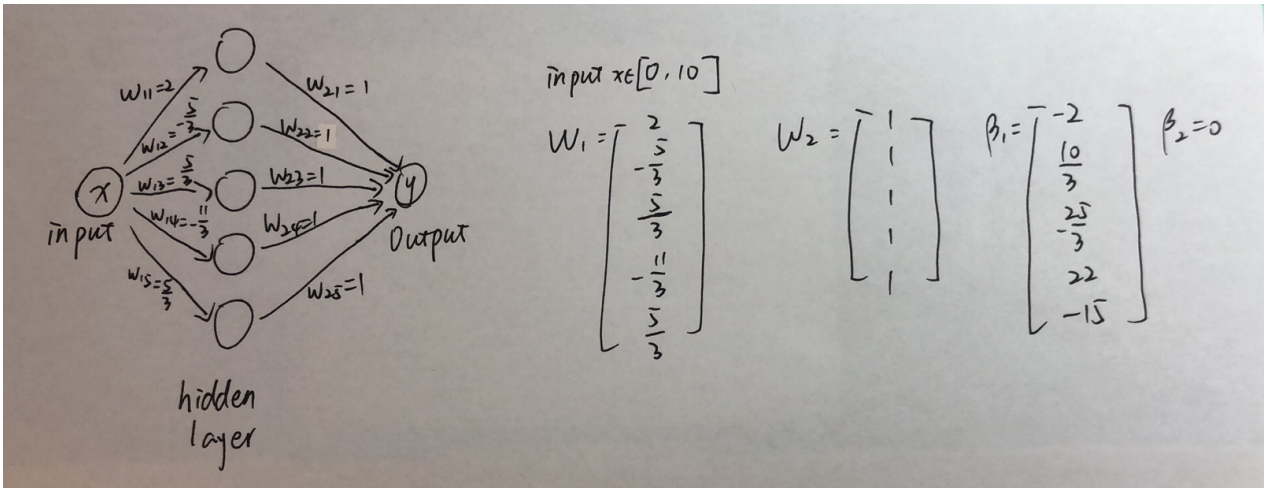
Since

$$Y_1 = \sigma(W_1 Y_0^T + \beta_1)$$
$$Y_i = \sigma(W_i(\sigma(W_{i-1}Y_{i-2}^T + \beta_{i-1}))^T + \beta_i)$$

The weights are $W_1 = [2,\ -\frac{5}{3},\ \frac{5}{3},\ -\frac{11}{3},\ \frac{5}{3}]$ and $W_2 = [1,1,1,1,1]$

The biases are $\beta_1 = [-2, \frac{10}{3}, -\frac{25}{3}, 22, -15]$ and β_2=0

The neutal network is



Written Problem 3

P: probability that the training set does not appear at all in the bootstrap replicate.

Expected number of the training set that does not appear at all in the bootstrap replicate:

$$Num = E[N \times P]$$
$$= N \times (1 - \frac{1}{N})^N$$

Expected fraction of the training set that does not appear at all in the bootstrap replicate:

$$Fraction = \frac{Num}{N} = (1 - \frac{1}{N})^N$$

The limit of this expectation as N → ∞:

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