In [1]:

```
#TensorFlow and tf.keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
#각각의 함수를 직접 불러오기 위해서 각각의 함수를 import한다.

import tensorflow as tf
#혹시 모르니까 tensorflow도 그냥 불러와두자.

#Helper libraries
import numpy as np
import matplotlib.pyplot as plt

#위에처럼 해도 되고, 이것처럼 해도 된다.
#이렇게 import 하는 경우, 밑에서 함수를 쓸 때
#model = tf.keras.models.Sequential()가 아닌, model = Sequential()을 쓰기만 해도 된다.
#이게 더 선호되는 방법.
```

In [2]:

```
(x_train, y_train),(x_test, y_test) = tf.keras.datasets.mnist.load_data()
#만약 tf.keras.datasets.mnist.load_data()가 아닌, mnist.load_data()만 쓰고 싶다면?
#from tensorflow.keras.datasets import mnist 를 써주면 된다.
#import tensorflow조차도 되어 있지 않다면 이 함수조차 불러올 수 없게 된다.

x_train, x_test = x_train / 255.0, x_test / 255.0
```

In [3]:

```
#x_train, y_train 등의 dimension을 알고 싶다면? shape를 쓰면 된다.
x_train.shape
#총 그림의 개수는 60000개, 한 그림의 dimension은 28*28. 총 vector는 784
```

Out[3]:

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

In [4]:

```
y_train.shape
#y_train에 들어 있는 정보는 총 60000개이다.
```

Out[4]:

```
(60000,)
```

In [5]:

```
(1,2,3) #tuple
[1,2,3] #list
(1) #tuple을 만들고자 했지만 tuple이 아니다.
```

Out[5]:

```
1
```

In [6]:

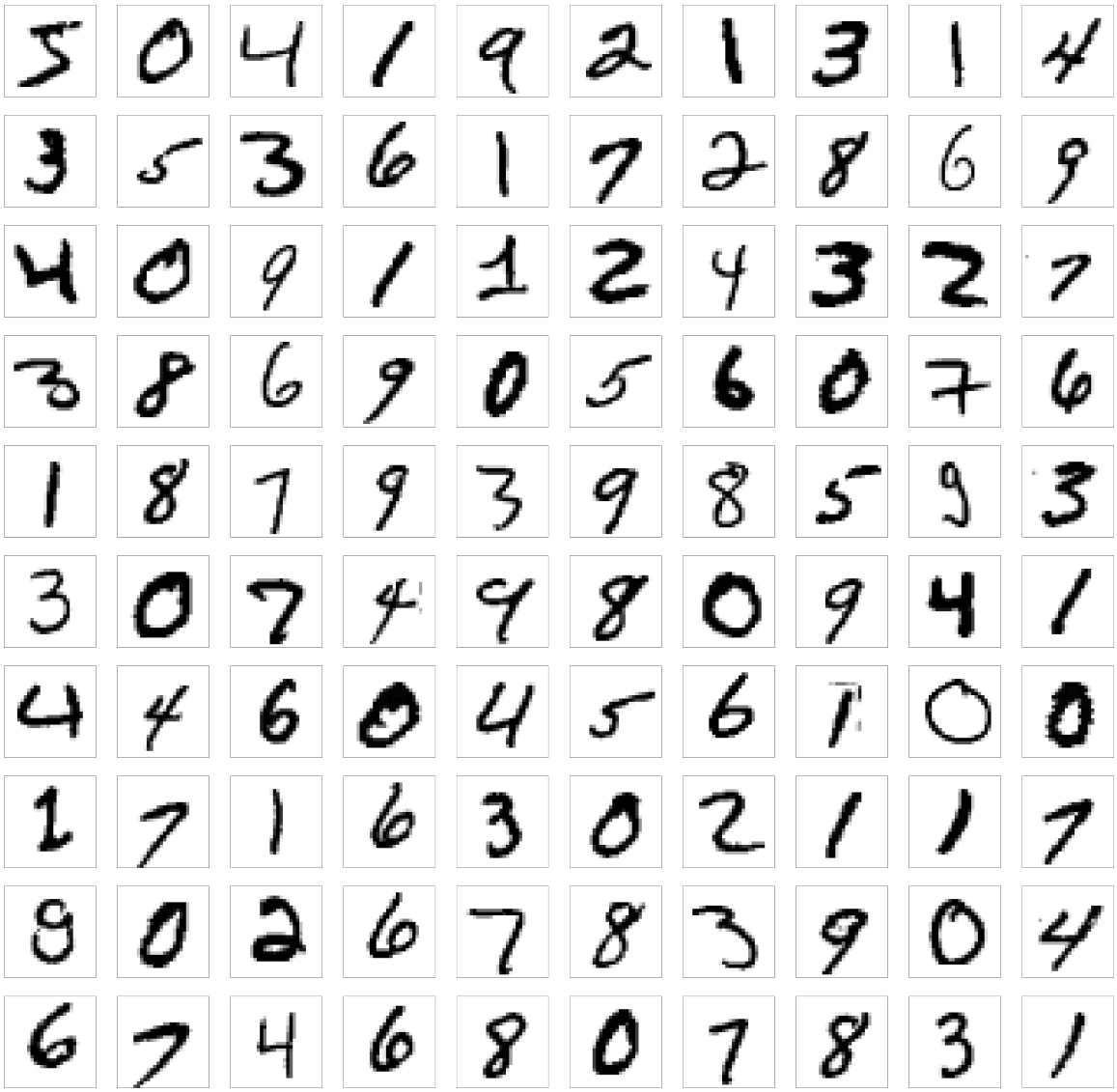
```
(1,) #하나의 element가 들어가는 tuple을 만들고 싶다면 이렇게 쓴다.
```

Out[6]:

```
(1,)
```

In [7]:

```
plt.figure(figsize=(100,100)) # size of figure
for i in range(100):
    plt.subplot(10,10,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(x_train[i], cmap=plt.cm.binary)
plt.show()
```



In [8]:

```
img_rows = 28
img_cols = 28

model = Sequential()
model.add(Flatten(input_shape=(img_rows, img_cols)))
#28x28의 2차원을 flatten 해줌으로써 1차원으로 만들어줄 수 있다.

model.add(Dense(512, activation='relu'))
model.add(Dense(10, activation ='softmax'))

#Boston Housing에서처럼 코드를 수정해 보았다.
#model을 한 단계씩 만들어가는 느낌이라 이게 더 효율적일지도 모르겠다.
```

```
WARNING:tensorflow:From C:₩Users₩Hyunseo Choi₩Anaconda3₩lib₩site-packages₩tensorfl
ow₩python₩ops₩resource_variable_ops.py:435: colocate_with (from tensorflow.python.
framework.ops) is deprecated and will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.
```

In [9]:

```
# Take a look at the model summary
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 512)               401920
_________________________________________________________________
dense_1 (Dense)              (None, 10)                5130
=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_________________________________________________________________
```

In [12]:

```
Adam = tf.keras.optimizers.Adam
model.compile(optimizer = Adam(lr = 0.001), loss='sparse_categorical_crossentropy', metrics=['ac
curacy'])

#optimizer로 Adam을 좀 더 세부적으로 사용하고 싶다면
#optimizer = 'adam'이 아닌, optimizer = Adam(lr = 0.01)로 쓴다.
#lr = learning rate를 조절해주게 된다. 이 외에도 여러가지 옵션이 존재하긴 한다.
#Adam을 따로 지정해주는 것의 장점? learning rate를 조절해줌으로써 속도를 조절해줄 수 있다.
#learning rate가 빠르면 속도는 빠르지만 정교함이 떨어질 수 있다. 반면, lr가 느리다면 속도는 느리
지만 정교함이 올라갈 수 있다.
```

In [13]:

```
print(model.input_shape)
print(model.output_shape)
#가장 처음과 끝이 여기서 정의되어 있다.
#input은 왜 2차원이며, output은 왜 1차원일까? 생각해보자.

#one hot coding을 받아들인다. 불일치를 어떻게 해결해야 하는가? 원래 데이터를 one hot coding의 형태로 바꿔주면 된다.
#윗 줄에서 sparse를 써줌으로써 그 기능을 할 수 있게 된다.
#sparse를 뺀다면 1은 [0100000000] 등으로 바꿔줘야 한다.
```

```
(None, 28, 28)
(None, 10)
```

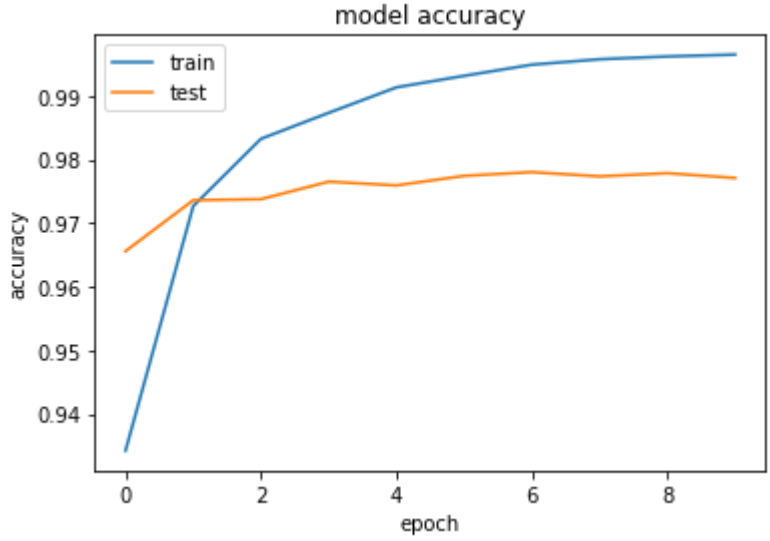
In [14]:

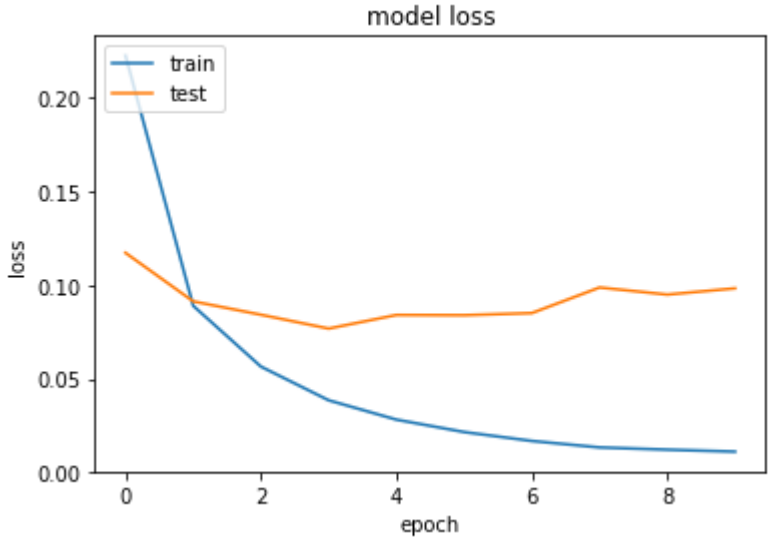
```
history = model.fit(x_train, y_train, epochs = 10, batch_size = 32, validation_split = 0.2)
#train하면서 나왔던 모든 자료들이 history라는 변수에 모두 저장이 될 것이다.

#batch_size는 위에서 본 None과 관련이 있다. 이걸 vector로 넣는 것이 아니라 여러 개를 쌓아서 넣을 때 유용하다.
#32가 평균. 32개씩 돌아가기 때문에 시간이 좀 걸린다.

#validation_split: model을 만들고 난 후에 test set을 이용해서 확인해 보는데, train을 하면서도 일종의 test과정을 거칠 수 있다. 이것이 validation.
#validation_split = 0.2: 지금 현재 train set 중에서 20퍼센트에 해당하는 부분을 validation으로 써라.

#결과를 보면 loss, acc와 더불어서 validation loss, validation accuracy가 따로 나온다.
#80퍼센트에 대해서는 그냥 train으로 돌리고(loss, acc), 20퍼센트에 대해서는 test해보는 것(val_loss, val_acc)
#train set에서의 accuracy는 가면 갈수록 올라가는 모습을 보인다. 그러나 validation_accuracy는 조금 올라가다가 떨어지기도 하는 모습을 보인다.
```

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/10
48000/48000 [==============================] - 22s 451us/sample - loss: 0.2227 - a
cc: 0.9342 - val_loss: 0.1173 - val_acc: 0.9656
Epoch 2/10
48000/48000 [==============================] - 19s 395us/sample - loss: 0.0890 - a
cc: 0.9727 - val_loss: 0.0914 - val_acc: 0.9736
Epoch 3/10
48000/48000 [==============================] - 21s 430us/sample - loss: 0.0565 - a
cc: 0.9832 - val_loss: 0.0842 - val_acc: 0.9737
Epoch 4/10
48000/48000 [==============================] - 19s 404us/sample - loss: 0.0385 - a
cc: 0.9873 - val_loss: 0.0769 - val_acc: 0.9765
Epoch 5/10
48000/48000 [==============================] - 18s 381us/sample - loss: 0.0282 - a
cc: 0.9913 - val_loss: 0.0841 - val_acc: 0.9759
Epoch 6/10
48000/48000 [==============================] - 19s 401us/sample - loss: 0.0216 - a
cc: 0.9931 - val_loss: 0.0840 - val_acc: 0.9774
Epoch 7/10
48000/48000 [==============================] - 19s 403us/sample - loss: 0.0168 - a
cc: 0.9949 - val_loss: 0.0851 - val_acc: 0.9780
Epoch 8/10
48000/48000 [==============================] - 21s 431us/sample - loss: 0.0134 - a
cc: 0.9957 - val_loss: 0.0988 - val_acc: 0.9773
Epoch 9/10
48000/48000 [==============================] - 21s 432us/sample - loss: 0.0122 - a
cc: 0.9962 - val_loss: 0.0950 - val_acc: 0.9778
Epoch 10/10
48000/48000 [==============================] - 19s 403us/sample - loss: 0.0111 - a
cc: 0.9965 - val_loss: 0.0983 - val_acc: 0.9771
```

In [15]:

```
# summarize history for accuracy
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
#위에서 저장했던 train의 기록인 history를 plot해본다.

# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```





In [ ]:

```
#위의 그래프는 accuracy를 기반으로 해서 plotting한 것이고, 아래 그래프는 loss를 기반으로 해서 plotting 한 것이다.
#accuracy:
#train set은 epoch가 늘어날수록 계속 오른다.
#test set은 비슷한 값으로 계속 가다가 어느 순간 떨어지기도 하고 오르기도 한다.
#epoch = 1인 경우, val_acc의 변화가 줄어든다. under fit.
#epoch = 2인 경우, test set의 상승이 별로 없다. 그러나 train set은 계속 상승하고 있다.
#fit한다:
#추가로 이 모델이 개선될 여지가 있다면 under fit. epoch가 0일 때의 train
#지금 가지고 있는 데이터 만으로는 잘 돌아가지만, 사전에 보유하지 않은 데이터로는 예상한대로 잘 나오지 않는다. 일반화가 되지는 않았으나 모델은 성능이 좋아지는 경우라면 over fit. epoch가 6일 때의 train.

#loss:
#loss는 많으면 좋지 않다. 작을 수록 좋다.
#train set은 점점 값이 떨어진다. 점점 fit해져가는 것.
#test set은 epoch가 2인 시점에서 loss가 떨어지지 않고 올라가는 양상을 보인다. 이후의 epoch에서도 over fit함을 볼 수 있다.
#이걸 봄으로써 epoch를 얼마나 돌려야 하는지 알 수 있다. 이 경우에는 2번 epoch 돌았을 때가 under fit도 아니고, over fit도 아닌 경우이므로 가장 이상적인 모델임을 알 수 있다.

#epoch = 2일 때의 model이 가장 좋은 모델인데, 부분부분 저장하지 않았다면 이걸 다시 불러올 수가 없다.
#이걸 하나하나 보고 저장하고, 나중에 다시 불러오는 기능을 fashion_mnist에서 다룰 것이다.

#그래프의 각 점 사이가 learning rate하고 관계가 있다. epoch의 시점과 관련이 있다.
#만약 learning rate가 너무너무 낮으면 epoch가 100이라도 under fit이 될 수 있는가? Y

#-------------------------------------------------------------------------------------------

#epoch를 너무 많이 하면 train 시간은 느리지만 결과는 너무 많이 나와서 over fit. 반대는 under fit.
#train data는 완벽하게 설명하지만 새로운 데이터가 들어간 경우 설명을 못할 때 over fit이라고 한다.
#train data든 뭐든 더 좋아질 가능성이 있다면 under fit.
```

In [16]:

```
from IPython.display import HTML
```

In [17]:

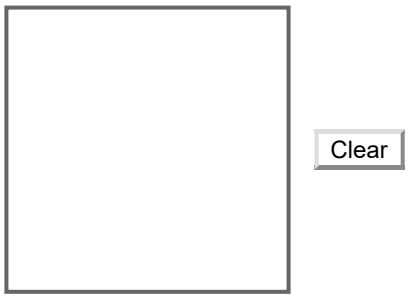
```
input_form = """
<table>
<td style="border-style: none;">
<div style="border: solid 2px #666; width: 143px; height: 144px;">
<canvas width="140" height="140"></canvas>
</div></td>
<td style="border-style: none;">
<button onclick="clear_value()">Clear</button>
</td>
</table>
"""

javascript = """
<script type="text/Javascript">
    var pixels = [];
    for (var i = 0; i < 28*28; i++) pixels[i] = 0
    var click = 0;

    var canvas = document.querySelector("canvas");
    canvas.addEventListener("mousemove", function(e){
        if (e.buttons == 1) {
            click = 1;
            canvas.getContext("2d").fillStyle = "rgb(0,0,0)";
            canvas.getContext("2d").fillRect(e.offsetX, e.offsetY, 8, 8);
            x = Math.floor(e.offsetY * 0.2)
            y = Math.floor(e.offsetX * 0.2) + 1
            for (var dy = 0; dy < 2; dy++){
                for (var dx = 0; dx < 2; dx++){
                    if ((x + dx < 28) && (y + dy < 28)){
                        pixels[(y+dy)+(x+dx)*28] = 1
```

```
                    }
                }
            }
        } else {
            if (click == 1) set_value()
            click = 0;
        }
    });

    function set_value(){
        var result = ""
        for (var i = 0; i < 28*28; i++) result += pixels[i] + ","
        var kernel = IPython.notebook.kernel;
        kernel.execute("image = [" + result + "]");
    }

    function clear_value(){
        canvas.getContext("2d").fillStyle = "rgb(255,255,255)";
        canvas.getContext("2d").fillRect(0, 0, 140, 140);
        for (var i = 0; i < 28*28; i++) pixels[i] = 0
    }
</script>
"""
```

In [19]:

```
HTML(input_form + javascript)
#여기에 handwriting으로 숫자를 쓴다.
```

Out[19]:



In [20]:

```
len(image)
#여기서 image의 길이는 784이지만 우리가 필요한 것은 28*28.
```

Out[20]:

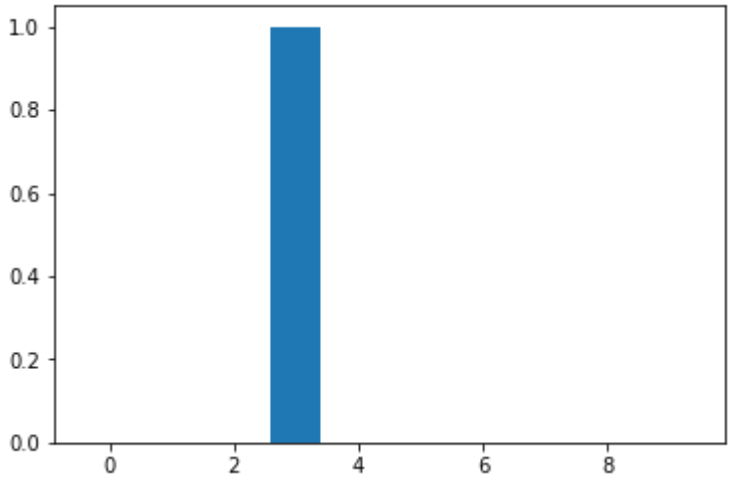
784

In [22]:

```
image_3darray = np.array(image).reshape([1,img_rows,img_cols])
#그래서 여기서 reshape를 해주게 된다.
#None 부분에 한 장임을 나타내기 위해서 1이 들어가야 한다.

result = model.predict(image_3darray)
plt.bar(list(range(10)), list(result.reshape([10])))

#어떤 숫자에 가장 가까운지 0-1사이의 비율로 나타내 준다.
```

Out[22]:

```
<BarContainer object of 10 artists>
```



In [ ]:

```

```