Boston Housing

https://www.kaggle.com/c/boston-housing (https://www.kaggle.com/c/boston-housing)

crim: 범죄율이 올라가면 집값은 내려갈 것 zn: 25000 평방미터에 대한 주거지의 비율. 주거지 <-> 상업지? indus: 마을 당 도매사업의 비율. chas: Charles강에 인접한 정도. dummy variable: 1 - 길이 강과 접해있다, 0 - 아니다. 강변일수록 집값은 올라간다. nox: 별로 안좋을 것 같다 rm: 평균 방의 개수 age: 1940년 전에 지어진 집들의 비율 dis: 다섯 개의 Boston 상업지구까지의 거리가 얼마나 가까운지? rad: 고속도로까지의 접근성 tax: 세금. 1만불 당 내는 세금의 비율 ptratio: 학생 대 선생의 비율. 이상적으로는 이 값이 낮은 것이 좋겠지만 높아야 집값이 올라가지 않을까. black: 마을에 있는 흑인의 비율 lstat: 인구의 하층 지위의 비율 medv: 주인이 소유한 집값 - 이게 타겟값이다.

tensorflow는 python을 기반으로 돈다. tensorflow keras dataset을 검색하면 이미 주어져 있는 dataset을 볼 수 있다. boston_housing에는 14개의 column이 존재할 것이다. 하나의 y와 13개의 x값으로 나올 것. cifar10에는 이미지가 여러개 나와 있고, 10가지의 종류로 구분될 수 있다. cifar100에는 똑같이 그림이 나와 있되, 종류가 100가지이다. 그림은 확대하면 깨지는데, 각 픽셀 하나씩에 숫자가 하나씩 들어가있다면: 흑백그림이라면 10000개의 숫자로 표현할 수 있고 - 1층. 흑/백 칼라그림이라면 30000개의 숫자로 표현할 수 있다 - 3층. RGB. fashion_mnist는 흑백사진. RGB 중에서 한 층만 있는 것이다. Green으로만 이루어져 있다. 흑백사진과 같은 개념으로 보면 된다. imdb는 영화 관련된 데이터베이스이다. 세계에서 가장 큰 것. 사람들이 리뷰를 작성했을 텐데, 그게 x값. 좋은지 안좋은지의 평가값이 y값이 될 것. mnist는 숫자 handwriting을 기반으로 실제 숫자를 알아보게 하는 것. reuters는 신문기사. input은 text로 들어가고, output은 topic이 무엇인지.

In [1]:

```
import tensorflow
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

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

In [2]:

```
tensorflow.keras.datasets.boston_housing.load_data(path = 'boston housing.npz')
```

Out[2]:

```
((array([[1.23247e+00, 0.00000e+00, 8.14000e+00, ..., 2.10000e+01,
          3.96900e+02, 1.87200e+01],
         [2.17700e-02, 8.25000e+01, 2.03000e+00, ..., 1.47000e+01,
          3.95380e+02, 3.11000e+00],
         [4.89822e+00, 0.00000e+00, 1.81000e+01, ..., 2.02000e+01,
          3.75520e+02, 3.26000e+00],
         ...,
         [3.46600e-02, 3.50000e+01, 6.06000e+00, ..., 1.69000e+01,
          3.62250e+02, 7.83000e+00],
         [2.14918e+00, 0.00000e+00, 1.95800e+01, ..., 1.47000e+01,
          2.61950e+02, 1.57900e+01],
         [1.43900e-02, 6.00000e+01, 2.93000e+00, ..., 1.56000e+01,
          3.76700e+02, 4.38000e+00]]),
  array([15.2, 42.3, 50. , 21.1, 17.7, 18.5, 11.3, 15.6, 15.6, 14.4, 12.1,
         17.9, 23.1, 19.9, 15.7,  8.8, 50. , 22.5, 24.1, 27.5, 10.9, 30.8,
         32.9, 24. , 18.5, 13.3, 22.9, 34.7, 16.6, 17.5, 22.3, 16.1, 14.9,
         23.1, 34.9, 25. , 13.9, 13.1, 20.4, 20. , 15.2, 24.7, 22.2, 16.7,
         12.7, 15.6, 18.4, 21. , 30.1, 15.1, 18.7,  9.6, 31.5, 24.8, 19.1,
         22. , 14.5, 11. , 32. , 29.4, 20.3, 24.4, 14.6, 19.5, 14.1, 14.3,
         15.6, 10.5,  6.3, 19.3, 19.3, 13.4, 36.4, 17.8, 13.5, 16.5,  8.3,
         14.3, 16. , 13.4, 28.6, 43.5, 20.2, 22. , 23. , 20.7, 12.5, 48.5,
         14.6, 13.4, 23.7, 50. , 21.7, 39.8, 38.7, 22.2, 34.9, 22.5, 31.1,
         28.7, 46. , 41.7, 21. , 26.6, 15. , 24.4, 13.3, 21.2, 11.7, 21.7,
         19.4, 50. , 22.8, 19.7, 24.7, 36.2, 14.2, 18.9, 18.3, 20.6, 24.6,
         18.2,  8.7, 44. , 10.4, 13.2, 21.2, 37. , 30.7, 22.9, 20. , 19.3,
         31.7, 32. , 23.1, 18.8, 10.9, 50. , 19.6,  5. , 14.4, 19.8, 13.8,
         19.6, 23.9, 24.5, 25. , 19.9, 17.2, 24.6, 13.5, 26.6, 21.4, 11.9,
         22.6, 19.6,  8.5, 23.7, 23.1, 22.4, 20.5, 23.6, 18.4, 35.2, 23.1,
         27.9, 20.6, 23.7, 28. , 13.6, 27.1, 23.6, 20.6, 18.2, 21.7, 17.1,
          8.4, 25.3, 13.8, 22.2, 18.4, 20.7, 31.6, 30.5, 20.3,  8.8, 19.2,
         19.4, 23.1, 23. , 14.8, 48.8, 22.6, 33.4, 21.1, 13.6, 32.2, 13.1,
         23.4, 18.9, 23.9, 11.8, 23.3, 22.8, 19.6, 16.7, 13.4, 22.2, 20.4,
         21.8, 26.4, 14.9, 24.1, 23.8, 12.3, 29.1, 21. , 19.5, 23.3, 23.8,
         17.8, 11.5, 21.7, 19.9, 25. , 33.4, 28.5, 21.4, 24.3, 27.5, 33.1,
         16.2, 23.3, 48.3, 22.9, 22.8, 13.1, 12.7, 22.6, 15. , 15.3, 10.5,
         24. , 18.5, 21.7, 19.5, 33.2, 23.2,  5. , 19.1, 12.7, 22.3, 10.2,
         13.9, 16.3, 17. , 20.1, 29.9, 17.2, 37.3, 45.4, 17.8, 23.2, 29. ,
         22. , 18. , 17.4, 34.6, 20.1, 25. , 15.6, 24.8, 28.2, 21.2, 21.4,
         23.8, 31. , 26.2, 17.4, 37.9, 17.5, 20. ,  8.3, 23.9,  8.4, 13.8,
          7.2, 11.7, 17.1, 21.6, 50. , 16.1, 20.4, 20.6, 21.4, 20.6, 36.5,
          8.5, 24.8, 10.8, 21.9, 17.3, 18.9, 36.2, 14.9, 18.2, 33.3, 21.8,
         19.7, 31.6, 24.8, 19.4, 22.8,  7.5, 44.8, 16.8, 18.7, 50. , 50. ,
         19.5, 20.1, 50. , 17.2, 20.8, 19.3, 41.3, 20.4, 20.5, 13.8, 16.5,
         23.9, 20.6, 31.5, 23.3, 16.8, 14. , 33.8, 36.1, 12.8, 18.3, 18.7,
         19.1, 29. , 30.1, 50. , 50. , 22. , 11.9, 37.6, 50. , 22.7, 20.8,
         23.5, 27.9, 50. , 19.3, 23.9, 22.6, 15.2, 21.7, 19.2, 43.8, 20.3,
         33.2, 19.9, 22.5, 32.7, 22. , 17.1, 19. , 15. , 16.1, 25.1, 23.7,
         28.7, 37.2, 22.6, 16.4, 25. , 29.8, 22.1, 17.4, 18.1, 30.3, 17.5,
         24.7, 12.6, 26.5, 28.7, 13.3, 10.4, 24.4, 23. , 20. , 17.8,  7. ,
         11.8, 24.4, 13.8, 19.4, 25.2, 19.4, 19.4, 29.1])),
 (array([[1.80846e+01, 0.00000e+00, 1.81000e+01, ..., 2.02000e+01,
          2.72500e+01, 2.90500e+01],
         [1.23290e-01, 0.00000e+00, 1.00100e+01, ..., 1.78000e+01,
          3.94950e+02, 1.62100e+01],
         [5.49700e-02, 0.00000e+00, 5.19000e+00, ..., 2.02000e+01,
          3.96900e+02, 9.74000e+00],
         ...,
         [1.83377e+00, 0.00000e+00, 1.95800e+01, ..., 1.47000e+01,
          3.89610e+02, 1.92000e+00],
```

```
        [3.58090e-01, 0.00000e+00, 6.20000e+00, ..., 1.74000e+01,
         3.91700e+02, 9.71000e+00],
        [2.92400e+00, 0.00000e+00, 1.95800e+01, ..., 1.47000e+01,
         2.40160e+02, 9.81000e+00]]),
 array([ 7.2, 18.8, 19. , 27. , 22.2, 24.5, 31.2, 22.9, 20.5, 23.2, 18.6,
        14.5, 17.8, 50. , 20.8, 24.3, 24.2, 19.8, 19.1, 22.7, 12. , 10.2,
        20. , 18.5, 20.9, 23. , 27.5, 30.1,  9.5, 22. , 21.2, 14.1, 33.1,
        23.4, 20.1,  7.4, 15.4, 23.8, 20.1, 24.5, 33. , 28.4, 14.1, 46.7,
        32.5, 29.6, 28.4, 19.8, 20.2, 25. , 35.4, 20.3,  9.7, 14.5, 34.9,
        26.6,  7.2, 50. , 32.4, 21.6, 29.8, 13.1, 27.5, 21.2, 23.1, 21.9,
        13. , 23.2,  8.1,  5.6, 21.7, 29.6, 19.6,  7. , 26.4, 18.9, 20.9,
        28.1, 35.4, 10.2, 24.3, 43.1, 17.6, 15.4, 16.2, 27.1, 21.4, 21.5,
        22.4, 25. , 16.6, 18.6, 22. , 42.8, 35.1, 21.5, 36. , 21.9, 24.1,
        50. , 26.7, 25. ]))
```

In [3]:

```
(x_train, y_train), (x_test, y_test) = tensorflow.keras.datasets.boston_housing.load_data(path =
'boston housing.npz')
```

In [4]:

```
model = Sequential() #모델 선언
model.add(Dense(1, input_shape=[13]))
#input layer의 노드는 x값이 13개 있어야 하고, 1인 노드가 있기 때문에 노드는 총 14개.
model.compile(optimizer='rmsprop', loss='mse')
model.summary()
```

```
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.
WARNING:tensorflow:From C:₩Users₩Hyunseo Choi₩Anaconda3₩lib₩site-packages₩tensorfl
ow₩python₩keras₩utils₩losses_utils.py:170: to_float (from tensorflow.python.ops.ma
th_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.cast instead.
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 1)                 14
=================================================================
Total params: 14
Trainable params: 14
Non-trainable params: 0
_________________________________________________________________
```

In [5]:

```
model.fit(x_train, y_train, epochs = 10)
#fit은 훈련이라는 뜻이 있다.
#epochs = 10이라면 데이터를 한 번만 쭉 돌리는 것. 1000을 넣으면 데이터를 1000번 돌리는 것이 된다.
#반복해서 돌릴수록 loss값이 점점 줄어드는 것을 볼 수 있다.
```

```
WARNING:tensorflow:From C:₩Users₩Hyunseo Choi₩Anaconda3₩lib₩site-packages₩tensorfl
ow₩python₩ops₩math_ops.py:3066: to_int32 (from tensorflow.python.ops.math_ops) is
deprecated and will be removed in a future version.
Instructions for updating:
Use tf.cast instead.
Epoch 1/10
404/404 [==============================] - 0s 1ms/sample - loss: 1909.4909
Epoch 2/10
404/404 [==============================] - 0s 91us/sample - loss: 1654.8757
Epoch 3/10
404/404 [==============================] - 0s 86us/sample - loss: 1470.4109
Epoch 4/10
404/404 [==============================] - 0s 84us/sample - loss: 1310.2056
Epoch 5/10
404/404 [==============================] - 0s 77us/sample - loss: 1154.9123
Epoch 6/10
404/404 [==============================] - 0s 99us/sample - loss: 1015.4184
Epoch 7/10
404/404 [==============================] - 0s 80us/sample - loss: 887.2080
Epoch 8/10
404/404 [==============================] - 0s 74us/sample - loss: 771.7756
Epoch 9/10
404/404 [==============================] - 0s 84us/sample - loss: 667.3640
Epoch 10/10
404/404 [==============================] - 0s 79us/sample - loss: 569.7036
```

Out[5]:

```
<tensorflow.python.keras.callbacks.History at 0x29ce99fc400>
```

In [6]:

```
model.evaluate(x_test, y_test)
```

```
102/102 [==============================] - 0s 1ms/sample - loss: 364.7826
```

Out[6]:

```
364.7826310700061
```

In [7]:

```
print(model.input_shape)
print(model.output_shape)
```

```
(None, 13)
(None, 1)
```

In [8]:

```
# INPUT::
# 1. crim: per capita crime rate by town.
# 2. zn: proportion of residential land zoned for lots over 25,000 sq.ft.
# 3. indus: proportion of non-retail business acres per town.
# 4. chas: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise).
# 5. nox: nitrogen oxides concentration (parts per 10 million).
# 6. rm: average number of rooms per dwelling.
# 7. age: proportion of owner-occupied units built prior to 1940.
# 8. dis: weighted mean of distances to five Boston employment centres.
# 9. rad: index of accessibility to radial highways.
# 10. tax: full-value property-tax rate per $10,000.
# 11. ptratio: pupil-teacher ratio by town.
# 12. black: 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town.
# 13. lstat: lower status of the population (percent).

# TARGET::
# medv: median value of owner-occupied homes in $1000s.

model.predict(np.array([0, 1, 50, 1, 0, 5, .5, 10, 1000, 100, 10, 100, 1]).reshape(1,13))
#위에서 지정되있는 순서에 따라서 값을 조금씩 변경시켜 보면
```

Out[8]:

```
array([[245.09528]], dtype=float32)
```

In [9]:

```
model = Sequential() #모델 선언
model.add(Dense(10, input_shape = [13], activation = 'relu'))
#input의 x의 개수는 13개, 하지만 1인 노드를 포함하면 14개.
#hidden layer는 하나지만 그 layer의 노드는 10개. 따라서 곱해진 개수는 모두 140개.
#hidden layer에서도 마찬가지로 1인 노드가 있다. 그러나 앞쪽에서 1인 노드로는 화살표가 그려지지
 않는다.
#단 다음 layer로 넘어갈 때는 1인 노드에서도 화살표가 나가게 된다. 총 11개.

#activation은 각 hidden layer에 첨가해서 성능을 올릴 수 있다. tanh, relu, sigmoid 등.
#regression의 경우에는 아무런 activation도 주지 않아도 된다.
#classification의 경우, 두 가지로 나뉜다. binary일 때는 sigmoid, 여러개 중 고를 때는 softmax

model.add(Dense(1))
#여기는 input_shape가 없다. 앞의 값이 input으로 들어올 것을 이미 알고 있기 때문이다.
#output의 개수 1개를 이야기한다.
#한 layer마다의 묶음의 input-output 세트를 Dense라고 생각하면 된다.

model.compile(optimizer='rmsprop', loss='mse')
#compile할 때: regression의 경우, classification의 경우.
#regression의 경우, optimizer = 'rmsprop', loss = 'mse'
#classification의 경우, optimizer = 'adam', loss = 'enthrophy', 앞으로 나올 것.

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 10)                140
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 11
=================================================================
Total params: 151
Trainable params: 151
Non-trainable params: 0
_________________________________________________________________
```

In [10]:

```
# parameter # = 화살표의 개수.
# 140은 (input layer 13 + 1) * 10 = 140
# 11은 (hidden layer 10 + 1) * 1 = 11
# 각 Dense를 구성하는 화살표의 개수를 parameter number로 나타낸다.
```

In [11]:

```
print(model.input_shape)
print(model.output_shape)

#왜 none이라고 붙을까?
#처음에 1x13의 vector가 들어가서 hidden layer를 만나 13x10을 거쳐 10x1을 통해 output을 나타낸다.
#가장 처음에 1x130이 아니라 100x130이어도 곱셈이 성립한다.
#따라서 input_shape와 output_shape의 none부분에는 어떤 것이 들어가든 상관이 없는 것이다.
#이 neural net에 하나의 샘플을 집어넣을 경우를 앞에서 살펴본 것이지만, 100개의 샘플을 넣어도 무
방하다.
#아무거나 들어가도 상관없다는 의미의 None일 것. 대신 input과 output에서 같은 값을 가지게 될 것.
```

```
(None, 13)
(None, 1)
```

None = batch_size?

정해지지 않은 값이라는 의미에서 None을 사용하였는데, 항상 지정되지 않은 값이어야만 하는 것은 아니다. 한번에 처리할 값을 batch_size라고 한다. 1개씩 처리할 것이라면 batch_size = 1, 100개씩 처리할 것이라면 batch_size = 100

In [12]:

```
x_train.shape
#1x13의 데이터 샘플이 404개가 있다.
```

Out[12]:

```
(404, 13)
```

In [14]:

```
#model.fit(x_train, y_train, epochs = 1000, batch_size = 1)
#1x13짜리 vector를 batch_size = 1에 따라 1개씩 쓰는 것. 이걸 실행시킨다면 화살표가 하나씩 올라가는 것을 볼 수 있다.

#model.fit(x_train, y_train, epochs = 1000, batch_size = 404)
#batch_size = 404라고 쓰면 404개의 데이터를 한 번에 집어넣는 것을 의미한다.

model.fit(x_train, y_train, epochs = 10)
#batch_size를 설정하지 않는다면 default값이 batch_size = 32가 될 것이다.

#1ms/sample은 샘플 하나를 처리하는 데 걸리는 시간을 이야기한다.
#batch_size가 얼마가 좋은지는 실행을 거듭하면서 찾아보는 것이 좋다.
#batch_size가 높다면 정보 처리 속도는 빠르지만 loss는 많이 떨어지지 않을 것이다.
#batch_size가 낮으면 정보 처리 속도는 느릴지라도 loss는 상당히 떨어지는 것을 볼 수 있다.
```

```
Epoch 1/10
404/404 [==============================] - 0s 132us/sample - loss: 31.1056
Epoch 2/10
404/404 [==============================] - 0s 80us/sample - loss: 30.1377
Epoch 3/10
404/404 [==============================] - 0s 89us/sample - loss: 27.0584
Epoch 4/10
404/404 [==============================] - 0s 79us/sample - loss: 29.2193
Epoch 5/10
404/404 [==============================] - 0s 86us/sample - loss: 28.7015
Epoch 6/10
404/404 [==============================] - 0s 91us/sample - loss: 31.2391
Epoch 7/10
404/404 [==============================] - 0s 82us/sample - loss: 30.3260
Epoch 8/10
404/404 [==============================] - 0s 91us/sample - loss: 28.7189
Epoch 9/10
404/404 [==============================] - 0s 81us/sample - loss: 32.3590
Epoch 10/10
404/404 [==============================] - 0s 89us/sample - loss: 31.1066
```

Out[14]:

```
<tensorflow.python.keras.callbacks.History at 0x29ce4556978>
```

In [15]:

```
model.evaluate(x_test, y_test)
```

```
102/102 [==============================] - 0s 2ms/sample - loss: 25.0390
```

Out[15]:

```
25.039006850298712
```

In [16]:

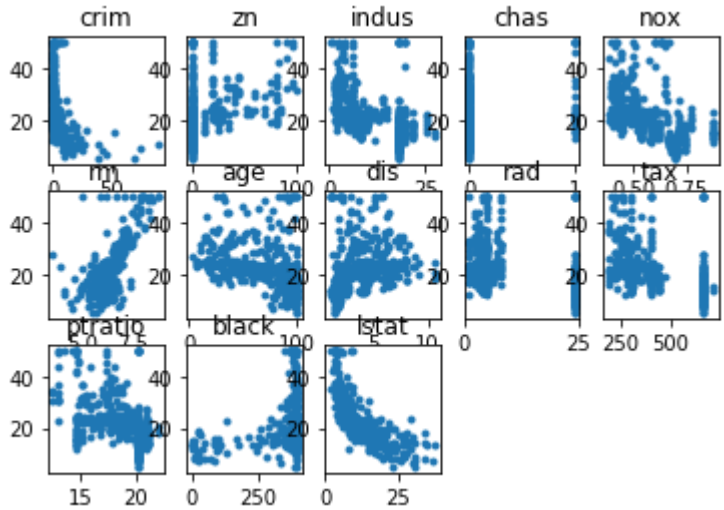
```
# INPUT::
# 1. crim: per capita crime rate by town.
# 2. zn: proportion of residential land zoned for lots over 25,000 sq.ft.
# 3. indus: proportion of non-retail business acres per town.
# 4. chas: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise).
# 5. nox: nitrogen oxides concentration (parts per 10 million).
# 6. rm: average number of rooms per dwelling.
# 7. age: proportion of owner-occupied units built prior to 1940.
# 8. dis: weighted mean of distances to five Boston employment centres.
# 9. rad: index of accessibility to radial highways.
# 10. tax: full-value property-tax rate per $10,000.
# 11. ptratio: pupil-teacher ratio by town.
# 12. black: 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town.
# 13. lstat: lower status of the population (percent).

# TARGET::
# medv: median value of owner-occupied homes in $1000s.

boston_housing = ['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age',
                  'dis', 'rad', 'tax', 'ptratio', 'black', 'lstat']

for i in range(13):
    plt.subplot(3,5,i+1)
    plt.plot(x_train[:,i], y_train, '.')
    #x_train과 y_train을 .으로 plot해보자.

    plt.title(boston_housing[i])
plt.show()
```



가상으로 factor를 만들어서 가장 집값을 높이는 방법과 낮추는 방법을 찾아보자.

make the highest price

In [17]:

```
model.predict(np.array([0, 90, 0, 1, 0,
                        100, 0, 10, 0, 250,
                        13, 300, 0]).reshape(1,13))
#model.fit / model.evaluate / model.predict
#model.predict: 입력만 넣으면 출력이 나오는 것. 10개 항목이 list로 이루어져 있다.
#list를 array로 바꿔주고 나서 reshape를 한다. 왜 reshape을 해야 하는가?
#이 모델의 input_shape가 (None, 13)이었다. 이 형태를 맞춰주기 위해서 reshape을 해줘야 한다.
```

Out[17]:

```
array([[296.6921]], dtype=float32)
```

make the lowest price

In [18]:

```
model.predict(np.array([100, 10, 25, 0, 1,
                        50, 100, 0, 25, 700,
                        23, 0, 30]).reshape(1,13))
```

Out[18]:

```
array([[191.89088]], dtype=float32)
```

In [19]:

```
#여기까지가 regression이다. 이게 가장 중요하다.
```

In [ ]:

```

```