출처가 명시되지 않은 모든 자료(이미지 등)는 조성현 강사님 블로그 및 강의 자료 기반.

Tensorflow 2.2 ver

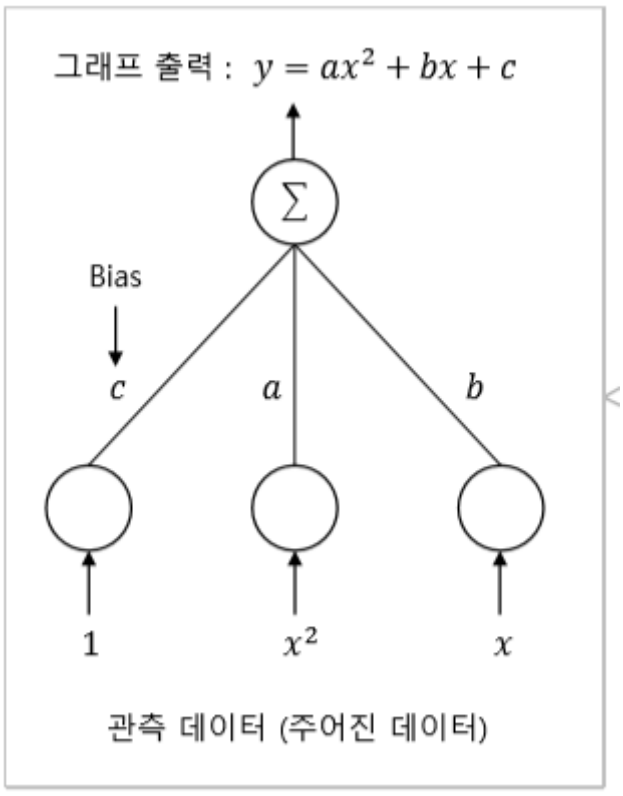
<< 딥러닝 - Keras >>

[Keras + Tensorflow]

1. 개요

Tensorflow를 쉽게 사용할 수 있게 제작된 도구로, `tensorflow.keras`에서 import하여 사용한다. `Keras` 코드를 작성하면 내부적으로 `Tensorflow` 기능이 동작한다.

Keras는 Sequential 방식과 functional API 방식이 있다.



각각의 층을 layer라고 한다. 각 layer들을 이어 붙여서 하나의 큰 모델을 맞추는 형태이다. 특히 맨 밑 층은 입력 데이터를 받는 층이며, 가장 위의 층은 activation 층이다. 내부적으로 최종적으로 가중치 합을 더하는 게 가장 기본.

Sequential Model

그래프를 그린다기보다는, 모델을 만든다는 개념이다.

- 빈 모델을 만들고, 입력 데이터를 받는 층을 쌓는다.
- input data가 1인 것을 `bias`라고 하는데, `Sequential Model`의 경우는 자동으로 설정된다.
- 모델 조건 설정: `model.compile()`
  - loss = 'MSE'.
  - optimizer, learning rate.
- 학습 : epoch도 지정해주면 됨.

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

model = Sequential() # 빈 모델을 만든다.
model.add(Dense(1, input_dim))
model.compile(loss='mse', optimizer=optimizers.Adam(lr=0.05))

# 학습
model.fit(X, y, epochs=300)
```

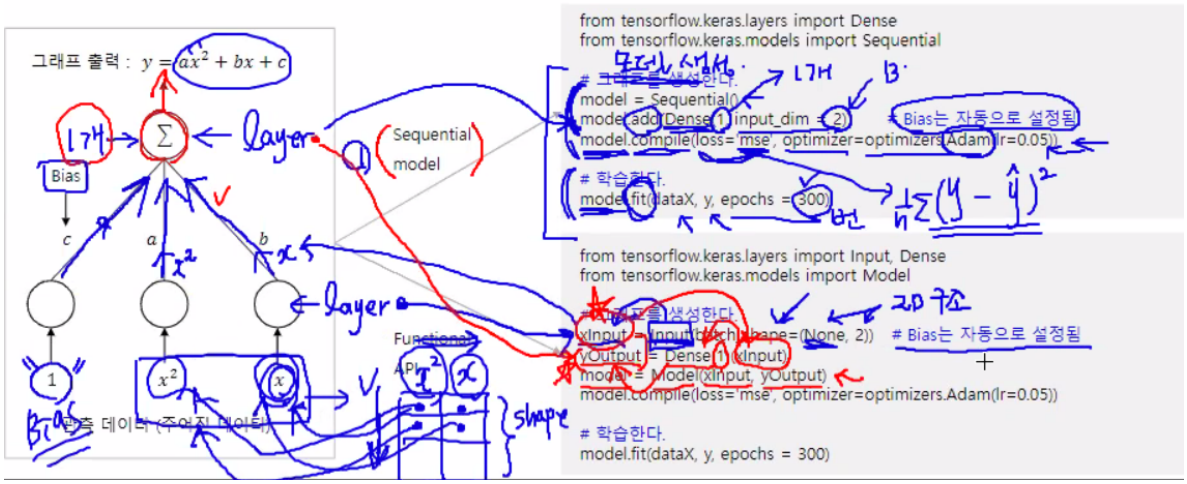
functional API

모델을 먼저 생성하지 않는다.

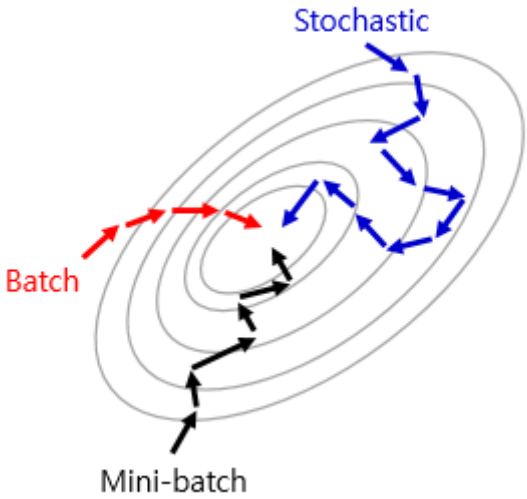
- `xInput` : input layer. 들어간 값이 자동으로 bypass되어 나옴.
  - data shape 맞춰줘야 함. 몇 개인지 모른다는 의미에서 `None` 사용.
  - bias는 자동 설정.
- `yOutput` : 바로 밑의 층이 인자로 들어온다. 즉, 어떤 layer가 입력으로 들어와서 출력되는지를 쓴다.
- `model` : 최종. 결과적으로 model은 `xInput`이 들어가서 `yOutput`이 나오는 구조.

```
from tensorflow.keras.layers import Input, Dense
from
```

두 번째 방식이 더 유연하기 때문에, 복잡한 문제를 해결할 수 있다. 위에 있는 Sequential model은 한 두 번 해보고 말고, 앞으로 functional API 스타일로 계속 코딩한다.



2. Loss Function 계산 방식



- Batch
  - 모든 데이터를 가지고 loss를 한 번에 계산하는 방식.
  - 데이터 전체에 대한 error를 한 번에 계산해서 loss 한 번에 계산하고, parameter를 한 번 업데이트.
  - 데이터 전체에 대해 안정적. 중간에 이상한 데이터가 들어오더라도 평균을 내면 묻혀서 일관되게 목표 지점을 찾아갈 수 있음.
- Stochastic Gradient
  - 데이터 하나씩 읽어 들여서 에러 값을 계산하고 loss 값에 반영.
  - 데이터 하나 당 편차가 클 수 있음.
  - 들어오는 데이터에 따라 들쭉날쭉할 수 있음.
  - 수렴하지 못하고 발산해 버리는 경우도 있을 수 있음.
- Mini-batch
  - 일부 데이터로 에러를 계산하고, loss에 나눠서 반영. 에러 계산 후 loss에 반영할 때마다 parameter에 업데이트.
  - batch와 stochastic gradient 방식의 중간적인 특성.

- 중간적이기 때문에 가장 무난하고, 많이 사용됨.

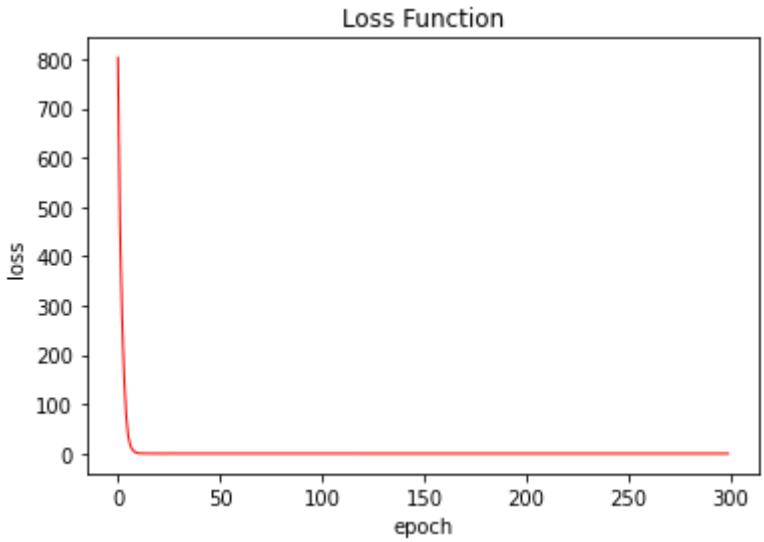
실습 1.

- Mini-batch 방식 : `batch_size` 옵션.
- `RMSprop` optimizer 사용.
- 추정 계수 확인 시 주의.

```
model.layers[0] # Dense 층
>>> <tensorflow.python.keras.layers.core.Dense at 0x7f91ca1a31d0>
model.layers[0].get_weights()
>>> [array([[1.9896517],
        [3.0180054]], dtype=float32), array([4.9938693], dtype=float32)]
```

- 결과

```
==== 추정 결과 ====
     w1 = 2.00
     w1 = 3.00
     b0 = 5.01
```



실습 2. Functional API

- data를 입력할 변수 형태에 맞게 `x^2`, `x` 형의 `np.array`로 만들어 줬다. 이 때 `np.stack`을 사용했음에 유의하자.

```
# 데이터 생성
x = np.array(np.arange(-5, 5, 0.1))
y = 2*x*x + 3*x + 5
X_data = np.stack([x*x, x]).T
```

- 그래프 생성 시 `batch_shape`을 잘 보자.

```
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras import Optimizers

#  그래프 생성
X_input = Input(batch_shape=(None, X_data.shape[1]))
y_output = Dense(1)(X_input) # Dense층 1개
model = Model(X_input, y_output) # input이 들어가서 output이 나오는 구조
model.compile(loss='mse', optimizer=optimizers.Adam(lr=0.05))
```

- `history` : 모델 loss 등에 대한 정보 가지고 있음.

```
h = model.fit(X_data, y, batch_size=10, epochs=300)
plt.plot(h.history['loss']) # loss 그림
```

- 결과
  - layers 확인

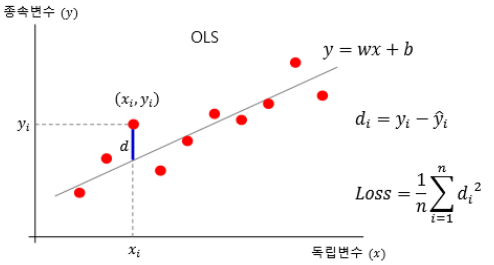
```
[<tensorflow.python.keras.engine.input_layer.InputLayer at 0x7f63968de898>,
 <tensorflow.python.keras.layers.core.Dense at 0x7f634c8f7ef0>]
```

  - parameters 확인 : 마지막 array는 random하게 설정한 bias에서

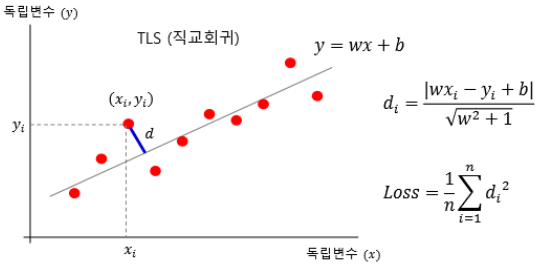
```
[array([[2.000001],
        [3.      ]], dtype=float32), array([4.9999886], dtype=float32)]
```

3. 회귀

OLS와 TLS의 경우 loss function만 달라진다.




```
# Loss function을 정의한다. (MSE : Mean Square Error)
predY = tf.add(tf.multiply(W, x), b)
loss = tf.reduce_mean(tf.square(predY - y))
```




```
# 점 (x, y)와 회귀직선 사이의 수직 거리
bunja = tf.abs(tf.subtract(tf.add(tf.multiply(W, x), b), y))  # 분자
bunmo = tf.sqrt(tf.add(tf.square(W), tf.constant(1.0)))       # 분모

# Loss function을 정의한다. (MSE : Mean Square Error)
loss = tf.reduce_mean(tf.square(tf.divide(bunja, bunmo)))
```

$$Loss = \frac{1}{n}\Sigma_{i=1}^{n} {d_i}^2$$

인데, 앞의 `1/n`과 `Sigma`는 `tf.reduce_mean(tf.sqrt())`로 해결된다.

안에 들어가는 거리 제곱이 어떻게 정의되는지만 잘 보자.

실습 3. OLS, Tensorflow

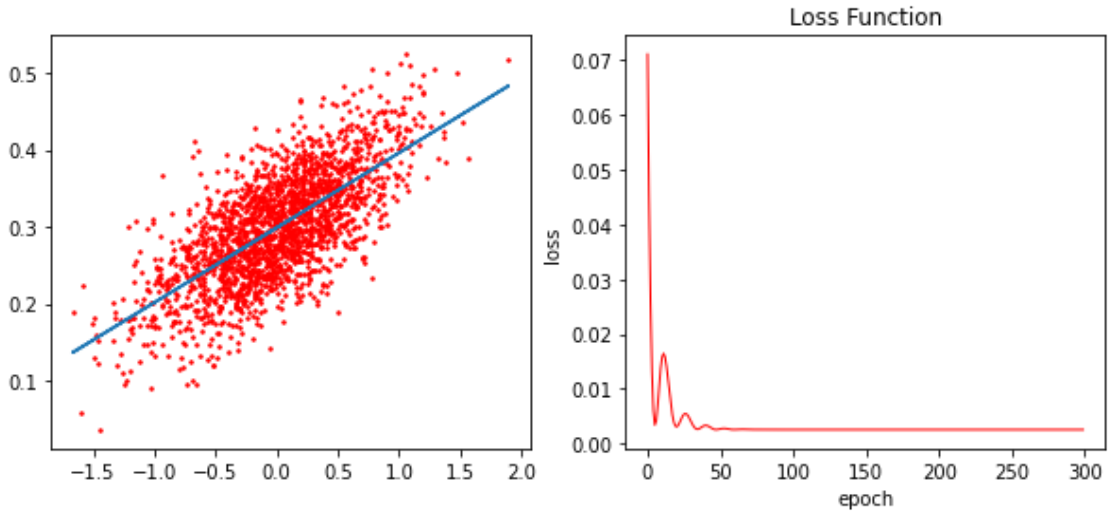
- weight : random값 하나 뽑아서 시작하도록. `W`의 경우 random하게 값을 하나 줘야 한다. 0에서 초기값 시작하면 안 된다. 데이터가 가중치 곱해져서 올라와야 하는데 0이면 통과시키지 말라는 뜻이나 마찬가지.
- bias : 초기값 0으로 시작.
- 내부 그래프 그려지는 게 달라지니까 산술식 말고, `tensorflow` 내부 함수를 사용하는 게 좋음.
- loss function 정의 및 cost minimize

```
def loss(x):
    y_pred = tf.add(tf.multiply(W, x), b) # y_pred = H*x + b
    return tf.reduce_mean(tf.square(y_pred - y))

# 학습
train_loss = []
for i in range(300):
    opt.minimize(lambda: loss(x), var_list=[W, b])
    train_loss.append(loss(x))
```

  - 일반 함수에서 loss라고 하면 그 값을 의미하고,
  - `loss(x)`라고 하면 그 함수의 return값이 된다. 숫자, 배열, 행렬 등. 이 경우에는 Tensor.
  - optimizer는 cost 자체를 최소화하니까 넘겨 주는 방식이 다르다.
  - `train_loss[-1]` 그대로 찍으면 `Tensor`가 출력된다.
- 결과 확인

```
======= 회귀직선의 방정식(OLS) =======
y = 0.0973 * x + 0.2988
```

결과 Plot



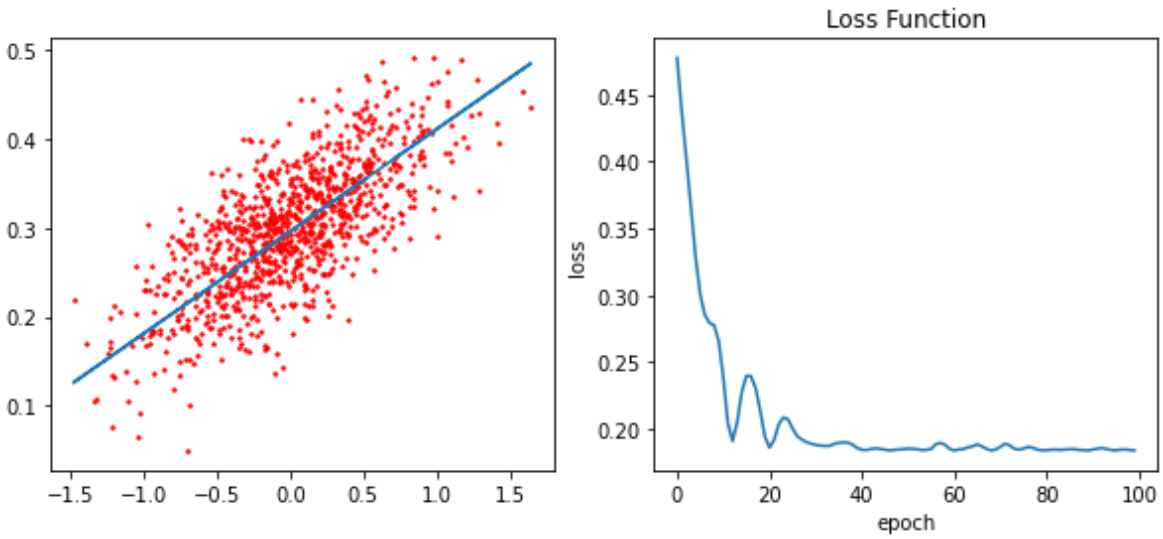
실습 4. TLS, Tensorflow

- loss function 정의하는 것만 다르다.

```
def loss(x):
    bunja = tf.abs(tf.subtract(tf.add(tf.multiply(W, x), b), y))
    bunmo = tf.sqrt(tf.add(tf.square(W), tf.constant(1.0)))

    return tf.reduce_mean(tf.square(tf.divide(bunja, bunmo)))
```

- 결과

```
학습 횟수를 입력하세요: 100
0번째 epoch: loss 0.4777
10번째 epoch: loss 0.2387
20번째 epoch: loss 0.1860
30번째 epoch: loss 0.1880
40번째 epoch: loss 0.1843
50번째 epoch: loss 0.1851
60번째 epoch: loss 0.1838
70번째 epoch: loss 0.1866
80번째 epoch: loss 0.1840
90번째 epoch: loss 0.1845

추정치: 0.3556296264573446, 실제: 0.37073875975089426
추정치: 0.25005374957249904, 실제: 0.22554158424110354
추정치: 0.3474873096231994, 실제: 0.3910314183939273
추정치: 0.2084900487327573, 실제: 0.2272126151481109
추정치: 0.3596666410614376, 실제: 0.4185682496508286
추정치: 0.3852047168773716, 실제: 0.40274390179473374
추정치: 0.29637810868137826, 실제: 0.28003919160933044
추정치: 0.3543123950957454, 실제: 0.36826782573180333
추정치: 0.1821549508294648, 실제: 0.16768729920111763
추정치: 0.2552032795142538, 실제: 0.309847777112495

========= 회귀 직선의 방정식(TLS) =========
y = 0.1149*x + 0.2963
```



더 공부해야 할 것

- optimizer ver1 -> ver2 바뀐 것.
  - optimizer에서 `lambda` 사용.
  - `var_list` 필요.
- 함수 vs. 함수 pointer