딥러닝

* Sequential 함수로 딥러닝의 구조를 한층 씩 쌓아올릴 수 있도록 만들어 줌.
* Model.add를 사용하여 필요한 층을 차례로 추가하게 된다.

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

import numpy as np

import tensorflow as tf

import pandas as pd

(데이터 불러오기)

data = pd.read\_csv('dataset/ThoraricSurgery(1).csv', encoding = 'utf-8')

(feature, target 분리)

X = data\_set[:, 0:17]

y = data\_set[:, 17]

np.random.seed(3)

tf.random.set\_seed(3)

data\_set = np.loadtxt('dataset/ThoraricSurgery.csv', delimiter = ',')

**실제 딥러닝 시행부분**

(딥러닝 구조 결정)

model = Sequential()

model.add(Dense(30, input\_dim = 17, activation = 'relu'))

model.add(Dense(1, activation = 'sigmoid'))

(딥러닝 실행)

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

model.fit(X, y, epochs = 100, batch\_size = 10)

층을 얼마나 쌓을지 결정하는 것은 데이터의 형태에 따라 다르다. 하지만, 캐라스는 model.add로 필요한 만큼의 층을 쉽게 쌓을 수 있다는 장점을 가지고 있다.

* Activation : 다음 층으로 어떻게 값을 넣을지 정하는 것. (relu, sigmoid 함수를 사용)
* Loss : 한번 신경망이 시행될 때마다 오차를 추적하는 함수
* Optimizer : 오차를 줄이는 함수

딥러닝을 시행할 때는, 위의 5줄 형태로만 시행하면 된다. (Sequential, add, compile, fit으로 구현하는 것)

구조 이해

* Input (X : 17개의 feature)
  + Weight : 가중치로 Input에 곱해지는 값이다. 일정한 가중치를 부여하고 입력된 값을 변형하고 다음 layer에 넘긴다. 다음 layer에서도 Input \* weight으로 예측값 Y\*가 나온다.
  + Loss function : 실제 Y와 예측 Y를 비교하여 Loss score를 도출한다. 여기서 Optimizer를 활용하여 가중치를 다시 조정하게 된다.

NL 해보기

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

가상데이터 생성

X = data = np.linspace(1,2,200)

y = X\*4 + np.random.randn(200)\*0.3

model = Sequential()

model.add(Dense(1, input\_dim=1, activation = 'linear'))

# tf.keras.models.Sequential, tf.keras.layer.Dense -> from import로 생략

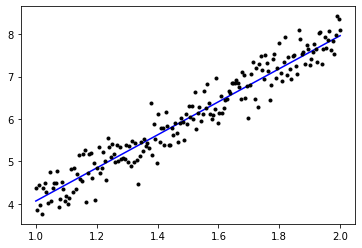
model.compile(loss = 'mse', optimizer = 'sgd', metrics = ['mse'])

model.fit(X, y, batch\_size = 1, epochs = 30)

predict = model.predict(data)

plt.plot(data, predict, 'b', data, y, 'k.')

plt.show()



model.fit(X, y, batch\_size = 1, epochs = 30)

output

Epoch 1/30

200/200 [==============================] - 1s 1ms/step - loss: 1.2804 - mse: 1.2804

Epoch 2/30

200/200 [==============================] - 0s 1ms/step - loss: 0.1637 - mse: 0.1637

…

Epoch 30/30

200/200 [==============================] - 0s 1ms/step - loss: 0.1033 - mse: 0.1033

위에서는 epochs를 30으로 했지만, 만약 100으로 지정한다면 loss값이 감소하다가 향상되는 것을 확인할 수 있음. 즉, Epochs를 늘린다고 해서 loss값이 무조건 줄어드는 것은 아니다. 이런 것을 over-fitting이라고 한다.

Loss : **회귀선(blue line)**과 **실제 값(black dot)**의 차이를 의미한다.

Mse : Mean Squared Error로 총 오차 값을 의미한다. (해당 출력에서는 loss = mse)

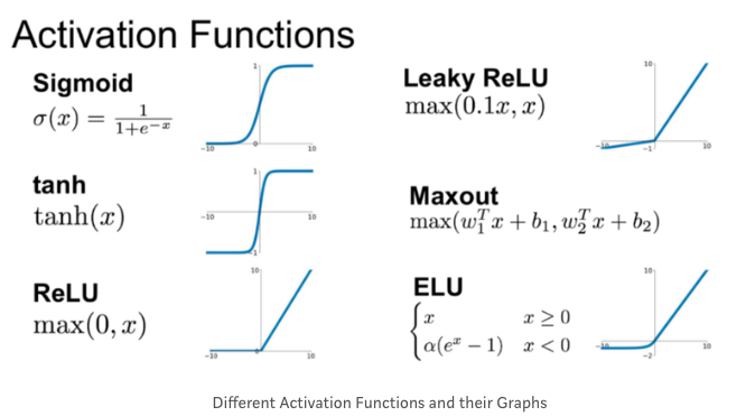
또한, 그래프를 보면, 첫번째 그래프가 파란 선 (선형 회귀)으로 도출된 것을 볼 수 있다. 즉, NL이 선형 회귀와 관련성이 있다고 볼 수 있는 것이다. (검은 점은 실제 y값임.) 여기서 선형 그래프는 y = ax+b라고 할수 있다.

* ‘a’는 weight, 가중치라고 한다.
* ‘b’는 bias, 편향이라고 한다.

기울기(a), 절편(b)를 조정하여 오차를 최소화하는 것이 필요함. 그런데, loss는 2차 함수로 표현되는 것을 알 수 있음. (mse=loss이므로) 그런데 2차 함수의 형태는 볼록한 형태임. 2차함수의 최솟값(오차의 최솟값)을 도출하는 과정은 미분을 통해서 할 수 있음. Y = ax2의 최솟값은 미분한다면, y’=2ax로 도출할 수 있고 x=0일 때, y’=0이 된다. 이런 형태로 loss 값을 구할 수 있는 것이다. (참고로 미분 값은 접선의 기울기이기도 함)

그런데, 변수가 여러 개 있는 경우라면 편미분의 개념을 도입해야 한다.

* F(x,y) = x2 +yx + a
* Y에 대한 편미분 : x
* X에 대한 편미분 : 2x+y



* 시그모이드 : 로지즈틱 회귀와 같은 분류 문제를 확률적으로 표현하는데 사용함.
  + 선형 함수의 결과를 0~1 사이의 비선형 형태로 변형해 줌.
  + 하지만, hidden layer의 깊이가 커지면 기울기 소멸 문제가 발생함. (미분을 계속 시행하다 보면, 값이 작아지고 결국 기울기가 0으로 만들어지는 문제임)
  + 시그모이드를 미분한다면, 0에서 가장 높은 값을 가지고 멀어질수록 0에 수렴함
* 하이퍼볼릭 탄젠트 (tanh) : -1, 1 사이의 비선형으로 바꿔 줌.
  + 0이 아닌 양수로 만들어 줬지만, 기울기 소멸 문제 잔존
* ReLu
  + 기울기 소멸의 문제가 발생하지 않음. 입력이 음수라면 0, 양수이면 x를 출력함.
  + Tanh보다 6배 정도 높은 성능을 자랑함
  + 하지만 음수일 때 0을 출력하기에 Leaky ReLu를 도입하여 한계를 보완함.
* 소프트맥스
  + 입력 값을 0 ~ 1 로 출력되도록 정규화 함. 출력의 총 합은 항상 1이 되도록 형성
  + 전체 경우의 수 중에서 특정 사건이 발생할 확률로 표현된다.

sigmoid 함수는 0, 1으로 결국 분리할 수 있음. X가 무한히 작거나, 크면 sigmoid에서는 0, 1으로 표현되게 된다.

선형 회귀

<https://github.com/gilbutITbook/080228> (git clone + Link) 후 ‘code .’ or ‘jupyter notebook’ 실행

# -\*- coding: utf-8 -\*-

import numpy as np

# x 값과 y값

x=[2, 4, 6, 8]

y=[81, 93, 91, 97]

# x와 y의 평균값

mx = np.mean(x)

my = np.mean(y)

print("x의 평균값:", mx)

print("y의 평균값:", my)

# 기울기 공식의 분모

divisor = sum([(mx - i)\*\*2 for i in x])

# 기울기 공식의 분자

def top(x, mx, y, my):

d = 0

for i in range(len(x)):

d += (x[i] - mx) \* (y[i] - my)

return d

dividend = top(x, mx, y, my)

print("분모:", divisor)

print("분자:", dividend)

# 기울기와 y 절편 구하기

a = dividend / divisor

b = my - (mx\*a)

# 출력으로 확인

print("기울기 a =", a)

print("y 절편 b =", b)

output

x의 평균값: 5.0

y의 평균값: 90.5

분모: 20.0

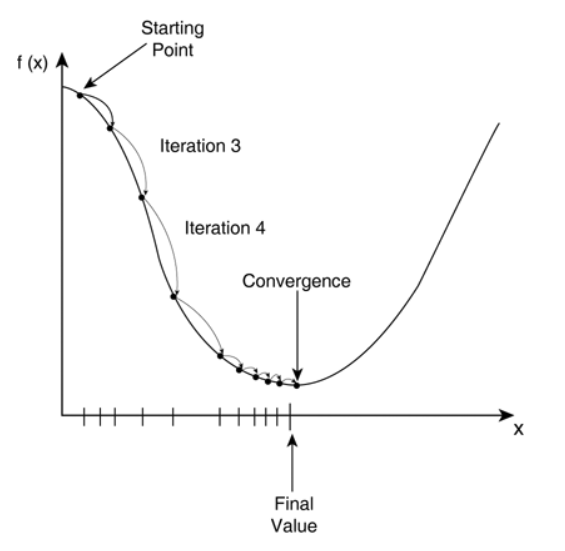
분자: 46.0

기울기 a = 2.3

y 절편 b = 79.0

여기서 평균제곱오차(MSE)는 로 y-hat = y-pred라고. 생각하면 됨.

경사 하강법 (gradient descent)

* Loss 의 최솟값을 찾기 위해서 임의의 a(가중치)를 대입하고 loss가 가장 작은 지점을 구해나가는 방식을 의미함.
* 애초에 loss함수는 이므로 2차함수의 형태임. 여기서 임의의 a를 넣고 미분을 한다면, 2차함수의 기울기를 알 수 있음. 기울기가 0이 될 때, 오차가 최솟값이 되기 때문에 일일이 대입해보고 기울기를 0으로 만드는 지점을 찾는 것이 경사 하강법임.
* 그래프를 보면 알 수 있음.
* 오차가 최소가 되는 Final Value를 찾아가는 과정에서 학습률 개념이 도입된다.
  + 학습률은 얼마나 이동할 것인지 이동거리를 결정하는 법
  + 여기서 a 값을 조정하는 법이 optimizer라고 한다. (adam 등의 여러 방식이 있음)
* 물론 b를 구하는 방식에도 경사하강법이 사용된다.

y-hat = ax + b이므로 에 대입하고 편미분을 진행하면 된다. 그리고 경사하강법을 도입하여 최적의 a, b를 찾게 된다. (기울기를 낮추기 위해 a간격을 조정하는 법이 optimizer, 이동거리가 학습률, learning rate)

y-hat = a \* x + b, Error = y – y-hat

a\_diff : MSE를 a로 편미분한 값

a = a – lr(학습률) \* a\_diff(편미분 계수)

(실습)

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

data = [[2,81], [4,93], [6,91], [8,97]]

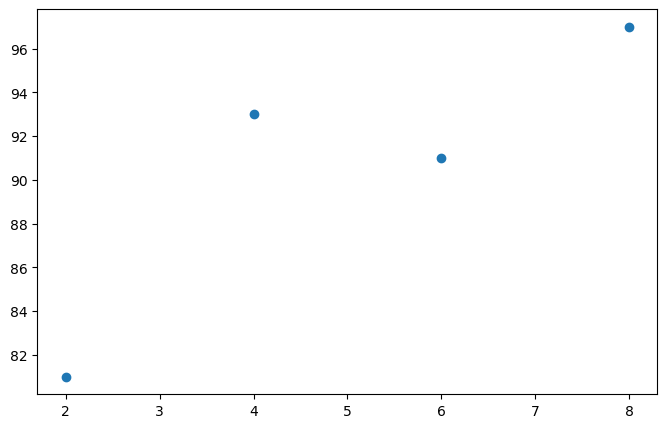
X = [i[0] for i in data]

y = [i[1] for i in data]

plt.figure(figsize=(8,5))

plt.scatter(X,y)

plt.show()

 (산점도 출력 결과)

#리스트 -> np array로 변환 (인덱스 부여로 하나씩 계산이 가능)

X\_data = np.array(X)

y\_data = np.array(y)

# 기울기 a와 절편 b의 값을 초기화 합니다.

a = 0

b = 0

#학습률을 정합니다.

lr = 0.03

#몇 번 반복될지를 설정합니다.

epochs = 2001

#경사 하강법 for 문

for i in range(epochs): # epoch 수 만큼 반복

y\_hat = a \* X\_data + b # y추정치를 구하는 식

error = y\_data - y\_hat # 실제 y와 추정한 값의 차이

a\_diff = -(2/len(X\_data)) \* sum(X\_data \* (error)) # a로 미분한 error

b\_diff = -(2/len(X\_data)) \* sum(error) # b로 미분한 error

a = a - lr \* a\_diff # 학습률을 곱해 기존의 a값을 업데이트합니다.

b = b - lr \* b\_diff # 학습률을 곱해 기존의 b값을 업데이트합니다.

if i % 100 == 0: # 100번 반복될 때마다 현재의 a값, b값을 출력합니다.

print("epoch=%5.f, 기울기(a, 가중치)=%.04f, 절편(b, bias)=%.04f, \

MSE=%.04f" % (i, a, b, sum(error)/4))

Output

epoch= 0, 기울기(a, 가중치)=27.8400, 절편(b, bias)=5.4300, MSE=90.5000

epoch= 100, 기울기(a, 가중치)=7.0739, 절편(b, bias)=50.5117, MSE=4.6644

epoch= 200, 기울기(a, 가중치)=4.0960, 절편(b, bias)=68.2822, MSE=1.7548

epoch= 300, 기울기(a, 가중치)=2.9757, 절편(b, bias)=74.9678, MSE=0.6602

epoch= 400, 기울기(a, 가중치)=2.5542, 절편(b, bias)=77.4830, MSE=0.2484

epoch= 500, 기울기(a, 가중치)=2.3956, 절편(b, bias)=78.4293, MSE=0.0934

epoch= 600, 기울기(a, 가중치)=2.3360, 절편(b, bias)=78.7853, MSE=0.0352

epoch= 700, 기울기(a, 가중치)=2.3135, 절편(b, bias)=78.9192, MSE=0.0132

epoch= 800, 기울기(a, 가중치)=2.3051, 절편(b, bias)=78.9696, MSE=0.0050

epoch= 900, 기울기(a, 가중치)=2.3019, 절편(b, bias)=78.9886, MSE=0.0019

epoch= 1000, 기울기(a, 가중치)=2.3007, 절편(b, bias)=78.9957, MSE=0.0007

epoch= 1100, 기울기(a, 가중치)=2.3003, 절편(b, bias)=78.9984, MSE=0.0003

epoch= 1200, 기울기(a, 가중치)=2.3001, 절편(b, bias)=78.9994, MSE=0.0001

epoch= 1300, 기울기(a, 가중치)=2.3000, 절편(b, bias)=78.9998, MSE=0.0000

epoch= 1400, 기울기(a, 가중치)=2.3000, 절편(b, bias)=78.9999, MSE=0.0000 (나머지는 생략)

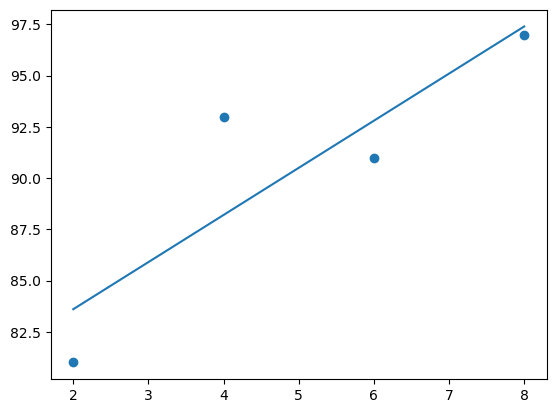
그래프 그려보기

y\_pred = a \* X\_data + b

plt.scatter(X, y)

plt.plot([min(X\_data), max(X\_data)], [min(y\_pred), max(y\_pred)])

plt.show()



로지스틱 회귀 분석

* True, False 값으로 도출될 수 있음. (합격 or 불합격의 형태)
  + 하지만 0, 1 값으로 도출되는 그래프를 그리려면 sigmoid함수를 사용하는 것이 가장 바람직하다. ( 으로 표현할 수 있는 함수를 사용)
  + a 값은 시그모이드의 경사도를 결정하며, b는 그래프 좌우 이동을 결정한다.
  + 따라서 a가 크다면, 오차는 0에 근접하게 될 것이며, b는 적절한 값으로 설정해야 오차를 감소시킬 수 있을 것이다.

순전파, 역전파

* Input 값이 은닉층을 투과하여 최종 y-hat을 도출하는 과정이 순전파
* y-hat과 실제 y의 차이를 비교하여 오차를 도출하고 다시 돌아가서 가중치를 조정하는 과정을 반복하여 오차를 줄여나가는 것이 역전파라고 보면 된다. (이때 미분 개념이 들어가게 됨)

그런데, 은닉층이 많을 때, sigmoid를 사용한다면, 역전파를 진행할 때, 미분을 계속 진행한다면 기울기가 소실되는 문제가 발생한다. 그런 문제를 해결하기 위해서 relu, softmax를 사용하게 된다. 따라서 은닉층에 사용하는 함수는 대체로 relu를 사용하며, 마지막 y-hat을 도출하는 때에 sigmoid를 사용하게 된다.

오차 역전파 (back propagation)

가중치를 설정하는 법은 경사하강법을 그대로 적용하게 됨.

(구조)

X1 -w1 n1 + b1 ->w5

-w2 1

Y-hat + b3 = 0.95 -> w1,w2,…,w6 수정 후 y-hat -y = error

-w3 0

X2 -w4 n2 + b2 ->w6

1-0.95 = 0.05라는 오차가 발생한다. 여기서 w1,w2…w6을 수정하고 다시 오차를 도출하고 가중치도 수정하게 된다. 이런 과정을 반복한다면? => 오차 역전파(back propagation)

X1 \* w1 + X2 \* w3 = > n1 + b1

X1 \* w2 + X2 \* w4 = > n2 + b2를 도출한다. 이를 input으로 설정

W5 \* (N1 + b1) + w6 \* (N2 + b2) => y-hat + b3 = 0.95

DNN (신경망의 확장)

* MLP(다층 퍼셉트론)에서 hidden layer의 수를 증가시킨 것으로 deep이라는 용어는 hidden layer가 많아서 층이 깊다는 것을 의미한다.
* 오차 역전파 과정을 진행하다보면, 은닉층이 많아질 수록 gradient가 소멸하는 문제가 발생한다. (gradient vanishing)
  + 오차를 수정하려고 hidden layer를 거슬러 올라가다보면, 가중치가 0이되는 기울기 소멸 문제가 발생하는 것임.
  + 미분을 통해서 가중치를 수정해야하는데, 층이 늘어나다보면, 기울기 값이 점점 작아져서 처음 층까지 전달되지 않는 다는 점이다. Sigmoid함수를 1차 미분을 하는 경우, 최댓값이 0.3으로 된다. 이를 해결하기 위해 다른 활성화 함수를 도입한 것이다.
* 과적합 문제 (over fitting)
  + 분류를 시행하면서 훈련을 너무 많이 한 나머지, Train의 분류는 잘하지만, test에서 정확도가 낮아지는 문제임.

딥러닝 구현을 위해 탠서플로우가 있음.

* Sequential 초기 작업.
* Model.add(dense()) activation, input
* Model.add(dense())
* Model.compile 모듈 생성 (loss, optimizer, metrics측정)
* Model.fit 훈련 (X, y, 훈련횟수)
* Predict 예측
* Evaluate 평가 (X\_test, y\_test)

데이터 다루기

* 데이터의 절대적 양보다는 필요한 데이터가 많아야 한다.
  + Bias가 없고 불필요한 정보가 없으며, 왜곡이 없는 데이터가 좋은 데이터이다.
  + 데이터의 양을 많이 모았다면, 머신러닝과 딥러닝에 활용할 수 있도록 데이터의 가공을 잘 해야 한다. **Preprocessing**
    - **노이즈, 이상치, 결측치 (noise, outlier, Nan)**

Dataset : pima\_indian\_diabetes.csv