4. Multi-variable linear regression(다중 선형 계수)

#기본적으로 알아야 할 TensorFlow 함수의 역할들

```
#행렬 연산 함수
tf.diag      #대각 행렬을 리턴한다
tf.transpose    #전치 행렬을 리턴한다
tf.matmul       #두 텐서의 행렬곱한 결과 텐서를 리턴한다
tf.matrix_determinant    #정방행렬의 행렬식 값을 리턴한다
tf.matrix_inverse     #정장 행렬의 역행렬을 리턴한다.

Hypothesis(가설 함수): H(x) = W*x + b
Cost Function(비용 한수): Cost(W) = tf.reduce_mean(tf.square(W*x - y))
Gradient descent(cost 가 최소가 되는 W 를 찾아주는 알고리즘): W := W - learning_rate(alpha) * tape.gradinet(cost, W)

#Gradient 함수
with tf.GradientTape() as tape:
    W_grad = tape.gradient(cost, W)
    #W_grad 가 cosr 를 W 라는 변수로 미분했을 때의 값을 의미한다.

TensorFlow 에서 Variable 은 변수를 의미한다

#TensorFlow 의 프로그램의 구조는 그래프 생성과 그래프 실행의 두가지 과정으로 이루어져 있기 때문에
연산 등을 실행해도 원하는 값이 안나오기 쉽다
#TensorFlow 1.xx 버전은 그래프의 실행은 Session 안에서 이루어 져야 한다.
#TensorFLow 2.00 버전은 굳이 그렇게 할 필요가 없다.
hello = tf.constant('Hello, TensorFlow!')
a,b,c = tf.constant(10.0), tf.constant(32.0), tf.add(a,b)
print(hello, c)

tf.constant()를 이용한 상수 생성, tf.Variables()를 이용한 변수 생성


#TensorFlow 로 상수를 생성하는 다양한 방법
tf.zeros_like #모든 원소를 0 으로 초기화한 텐서 생성
tf.ones_likt   #모든 원소를 1 로 초기화한 텐서 생성
tf.fill   #주어진 스칼라 값으로 원소를 초기화한 텐서 생성
tf.constant    #함수 인수로 지정된 값을 이용하여 상수 텐서 생성

#TensorFlow 로 텐서를 생성하는 함수
tf.random_normal   #정규 분포를 따르는 난수로 텐서 생성
tf.random_uniform   #균등 분포를 따르는 난수로 텐서 생성
tf.random_shufflw   #첫번째 차원을 기준으로 텐서의 원소를 섞음
tf.set_random_seed   #난수 시드를 설정
```

#Simple Example(2 variables) Hypothesis = W*x+b

```
import tensorflow as tf
import numpy as np
x1_data, x2_data, y_data = [1,0,3,0,5],[0,2,0,4,0],[1,2,3,4,5]
W1, W2, b = tf.Variable(tf.random.uniform((1,),-10.0,10.0)),tf.Variable(tf.random.uniform((1,),-
10.0,10.0)),tf.Variable(tf.random.uniform((1,),-10.0,10.0))
learning_rate = tf.Variable(0.001)
for i in range(1000+1):
    with tf.GradientTape() as tape:
        hypothesis = (W1*x1_data)+(W2*x2_data)+b
        cost = tf.reduce_mean(tf.square(hypothesis-y_data))
    W1_grad, W2_grad, b_grad = tape.gradient(cost, [W1,W2,b])
```

```
    W1.assign_sub(learning_rate*W1_grad)
    W2.assign_sub(learning_rate*W2_grad)
    b.assign_sub(learning_rate*b_grad)
    if i%50 == 0:
        print('{:5}|{:10.6f}|{:10.4f}|{:10.4f}|{:10.6f}'.format(i,cost.numpy(),W1.numpy()[0], W2.numpy()[0],
b.numpy()[0]))
```

#Simple Example(2 variables with matrix) Hypothesis = W*x +b

```
import tensorflow as tf
import numpy as np
x_data = [[1.,0.,3.,0.,5.,],[0.,2.,0.,4.,0.,]]
y_data = [1,2,3,4,5]
W = tf.Variable(tf.random.uniform((1,2),-1.0,1.0))
b = tf.Variable(tf.random.uniform((1,),-1.0,1.0))
learning_rate = tf.Variable(0.001)
for i in range(1000+1):
    with tf.GradientTape() as tape:
        hypothesis = tf.matmul(W,x_data) + b
        cost = tf.reduce_mean(tf.square(hypothesis-y_data))
        W_grad, b_grad = tape.gradient(cost, [W,b])
        W.assign_sub(learning_rate * W_grad)
        b.assign_sub(learning_rate*b_grad)
    if i%50 == 0:
        print('{:5}|{:10.6f}|{:10.4f}|{:10.4f}|{:10.6f}'.format(i, cost.numpy(),
W.numpy()[0][0],W.numpy()[0][1],b.numpy()[0]))
```

#Prediction(예측 using Matrix)

In [51]:
```
import tensorflow as tf
import numpy as np
data = np.array([[73.,80.,75.,152.],[93.,88.,93.,185.],[89.,91.,90.,180.],[96.,98.,100.,196.],[73.,66.,70.,142.]],dtype = np.float
X,y = data[:,:-1],data[:,[-1]]
W,b = tf.Variable(tf.random.normal((3,1))), tf.Variable(tf.random.normal((1,)))
learning_rate = 0.00001
def predict(X):
    return tf.matmul(X,W) + b
print('epoch|cost')
for i in range(10000+1):
    with tf.GradientTape() as tape:
        cost = tf.reduce_mean((tf.square(predict(X)-y)))
    W_grad, b_grad = tape.gradient(cost, [W,b])
    W.assign_sub(learning_rate*W_grad)
    b.assign_sub(learning_rate*b_grad)
    if i%1000 == 0:
        print('{:5}|{:10.4f}'.format(i,cost.numpy()))
```

```
epoch|cost
    0|11433.5791
 1000|    1.0980
 2000|    0.7675
 3000|    0.5727
 4000|    0.4568
 5000|    0.3870
 6000|    0.3442
 7000|    0.3170
 8000|    0.2992
 9000|    0.2869
10000|    0.2778
```

In [49]:
```
predict([[90.,95.,92.,],[84.,92.,85.,]]).numpy()
```

Out[49]:
```
array([[183.7254],
       [170.7684]], dtype=float32)
```

In [50]:
```
predict(X).numpy()
```

Out[50]:
```
array([[150.2095 ],
       [185.27461],
       [179.824  ],
       [198.15025],
       [140.7955 ]], dtype=float32)
```

위와 같은 방법으로 아래 표에서의 X값들을 이용하여 Y의 값을 도출해 내는데, predict(X).numpy()는 가설함수 tf.matmul(X,W) + b로 계산한 값을 구하는 것이고, predict([[90.,95.,92.],[84.,92,85.]]).numpy()는 새로운 X의 값을 matrix의 형태로 입력하고, 그에 해당하는 Y값의 예측을 출력해준다. 새로 입력해준 데이터가 2행, 3열이기 때문에 행의 개수와 같은 2개의 Y값을 출력해 준다. 또한, 위의 학습시키는 과정에서 보면 cost를 최소한으로 줄이는 것이 목표인 만큼 학습 횟수를 증가시킬수록 cost또한 작아지는 형태를 띈다.

X1	X2	X3	Y1(predict)
90	95	92	183.7254
84	92	85	170.7684

X1	X2	X3	Y1
73	80	75	152
93	88	93	185
89	91	90	180
96	98	100	196
73	66	70	142

5.Logistic Regression(로지스틱 회귀)

#Logistic Regression

앞서 공부했던 linear regression과 달리 이 경우애는 0과 1, 두개의 값만으로 출력이 가능하도록 한다. 앞서서 구했던 가설함수는 X*W인데, 이를 구하면 연속적인 값이 출력이 된다. 또한, 0과 1사이의 y값만 출력 되는 것이 아니기 때문에 출력값의 범위가 0과 1 사이로 한정 될 수 있도록 sigmoid function이라는 것을 사용하도록 한다. 그 함수는 tensorflow로 작성하였을 때 hypothesis는 tf.sigmoid(z)로, cost함수는, 즉 오차는 tf.reduce_mean(tf.reduce_sum(y*(tf.log(hypothesis) – (1-y)*(tf.log(1=hypothesis))) 라는 코드를 이용하여 구현한다.

그리고 나중에 cost함수의 최솟값을 구하기 위해서는 z, 혹은 theta에 대해 미분을 하여 최솟값을 구하도록 한다. Tf.gradient(cost, [z]) 이런 식으로 앞서 구했던 방식과 비슷하게 구현하면 될 것이다.

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
#반복해서 시행할 수 있도록 하기 위해서
tf.random.set_seed(1800)
data = np.array([[1.,2,0.],[2.,3.,0.],[3.,1.,0.],[4.,3.,1.],[5.,3.,1.],[6.,2.,1.]],dtype = np.float32)
x_train, y_train = data[:, :-1], data[:, [-1]]
#logistic classification의 모델 W와 b를 설정
W = tf.Variable(tf.random.normal([2,1]))
b = tf.Variable(tf.random.normal([1]))
x_test, y_test = [[5.,2.]], [[1.]]
x1,x2 = [x[0] for x in x_train], [x[1] for x in x_train]
colors = [int(y[0]%3) for y in y_train]
plt.scatter(x1, x2, c = colors,marker='^')
plt.scatter(x_test[0][0], x_test[0][1],c = 'red')
plt.xlabel('x1')
plt.ylabel('x2')
```

```
plt.show()


# logistic regression(가설 설정)
# sigmoid function = 1/(1+e^(XW))
def logistic_regression(x_train):
    hypothesis = tf.divide(1., 1 + tf.exp(tf.matmul(x_train, W) + b))
    return hypothesis


# sigmoid 예측값을 0 또는 1 로 반환
def accuracy_fn(hypothesis, labels):
    predicted = tf.cast(hypothesis > 0.5, dtype=tf.float32)
    accuracy = tf.reduce_mean(tf.cast(tf.equal(predicted, labels), dtype=tf.int32))
    return accuracy


epoches = 1001
learning_rate = 0.01
for i in range(epoches):
    with tf.GradientTape() as tape:
        cost = -tf.reduce_mean(y_train * tf.math.log(logistic_regression(x_train)) + (
                (1 - y_train) * (tf.math.log(1 - logistic_regression(x_train)))))
    W_grad, b_grad = tape.gradient(cost, [W, b])
    W.assign_sub(learning_rate * W_grad)
    b.assign_sub(learning_rate * b_grad)
    if i % 100 == 0:
        print('Iter:{},loss:{:.4f}'.format(i, cost.numpy()))
test_acc = accuracy_fn(logistic_regression(x_test), y_test)
print('Testset Accuracy: {:.4f}'.format(test_acc))
```

여기서 중요한 것은, 그리고 앞과 다른 점은 가설 함수를 정의 하는 방법이 sigmoid 함수식을 이용한다는 점 뿐만 아니라 가설 값을, 즉 logistic_regression의 값과 0.5와의 대소 여부에 따라 0과 1 로 값을 이분화하여 출력하는 accuracy_fn함수가 새로 생겼다는 점에서 차이를 보인다.

6. Softmax Classification

앞서 공부 했던 로지스틱 회귀와 다중 선형 계수의 개념이 모두 필요한 부분이다.

Logistic 회귀의 linear 회귀와의 제일 큰 차이점은 이분법적으로 0 또는 1의 출력값을 만들기 위해 sigmoid함수라는 것을 이용한다는 것이다. 이 함수를 g(z)라고 하며, 여기서의 z는 tf.matmul(X, W) + b이고 sigmoid 함수식은 $\frac{1}{1+e^{-z}}$이다.

그리고 logistic회귀에서의 cost함수는 $\frac{\sum_0^m (y*\log(z)+(1-y)*\log(1-z)}{-m}$ 이다.

여기에 오늘 배운 내용은 multinominal classification이라는 내용이다. 분류 기준이 많은 경우에 사용하는데, 주어지는 입력값에 대한 예측의 결과값을 3개 이상으로 나누는 것이다. 보통 A,B,C 3개의 class가 있다고 가정 할 때 A or not A, B or not B의 식으로 구분을 하곤 한다. $\begin{matrix} W_{a1} & W_{a2} & W_{a3} \\ W_{b1} & W_{b2} & W_{b3} \\ W_{c1} & W_{c2} & W_{c3} \end{matrix} * \begin{matrix} X_1 \\ X_2 \\ X_3 \end{matrix}$ = tf.matmul(X, W)이다. 이렇게 해서 나오는 행렬이 y의 예측값들이고, 그렇게 계산한 값을 sigmoid함수로 계산하였을 때에 나오는 확률값들이 0과 1 사이의 우리가 구하고자 하는 값들이다. 그리고 제일 큰 확률만 1로 하고 나머지는 0으로 바꾸어 주기 위해서 one-hot encoding이라는 것을 사용한다.

$D(S,L) = -\sum_i L_i * \log(S_i)$ 이고 이 방법을 이용해서 도출되는 cost함수의 식은 $\frac{\sum_i D(S(W_i+b),L_i)}{m}$ 이다. 그리고 여기서 S()는 sigmoid 함수의 약자이다.

이를 간단한 input data 를 이용해서 계산하는 것을 코드를 짜보자면

```
import tensorflow as tf
import numpy as np
x_data = [[1, 2, 1, 1],
          [2, 1, 3, 2],
          [3, 1, 3, 4],
          [4, 1, 5, 5],
          [1, 7, 5, 5],
          [1, 2, 5, 6],
          [1, 6, 6, 6],
          [1, 7, 7, 7]]
y_data = [[0, 0, 1],
          [0, 0, 1],
          [0, 0, 1],
          [0, 1, 0],
          [0, 1, 0],
          [0, 1, 0],
          [1, 0, 0],
          [1, 0, 0]]
#계산하기 쉽도록 자료형을 바꿔줌
x_data = np.asarray(x_data, dtype = np.float32)
y_data = np.asarray(y_data, dtype = np.float32)
nb_classes = 3
W = tf.Variable(tf.random.normal((4,nb_classes)))
b = tf.Variable(tf.random.normal((nb_classes,)))
#가설 함수
def hypothesis(X):
    return tf.nn.softmax(tf.matmul(X,W) + b)
#오차함수
def cost_fn(X,Y):
    cost = -tf.reduce_sum(Y*tf.math.log(hypothesis(X)),axis = 1)
    #가설값과 실제값의 차이인 cost 는 계속 감소하도록 하는 것이 ! 목표
    cost_mean = tf.reduce_mean(cost)
    return cost_mean

epochs = 2000
learning_rate = 0.1
for i in range(epochs):
    with tf.GradientTape() as tape:
        loss = cost_fn(x_data,y_data)
    W_grad, b_grad = tape.gradient(loss, [W,b])
    W.assign_sub(learning_rate*W_grad)
    b.assign_sub(learning_rate*b_grad)
    if i %100 == 0:
        print('Loss at epoch %d : %f' %(i+1, cost_fn(x_data,y_data).numpy()))
```

이렇게 된다.

다음은 동물의 특징으로 구분을 하는 좀더 실생활에 응용이 가능한 분류기준을 가지고 코드를 작성해 보자.

우선 이 경우에는 jupyter notebook에서 csv파일을 불러오는데 좀 문제가 있었으나 그냥 jupyternotebook의 home에 파일을 업로드 하면 되는 거였다.

우선 코딩을 한번 해보고 강의를 들었는데 강의를 들은 뒤에 알게된 사실들이 몇 개 존재한다.

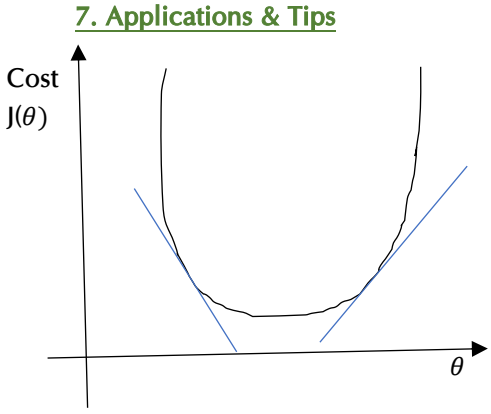
1. prediction(X,Y)라는 함수를 정의한다. 이는 accuracy를 판단하기 위함인데, 정확도를 측정하기 위해서는 cost_fn(X,Y)함수를 통해 구해지는 cost와 실제 예측값의 차이를 합해 평균을 내어서 구한다. 결과적으로는 accuracy가 1에 가까워 질수록 좋기 때문에 실제로 그렇게 결과가 나와야 제대로 구현을 했다고 할 수 있다.

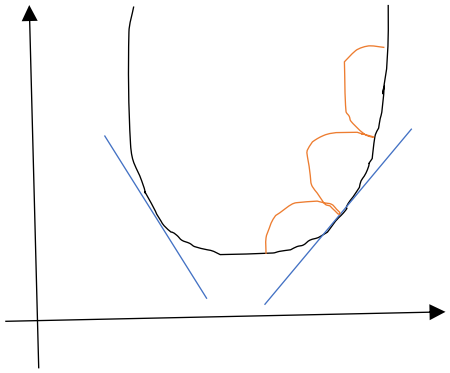
2. tf.nn.softmax(tf.matmul(X,W)+b))에서 softmax는 곧 sigmoid 함수(tf.divide(1., 1+tf.exp(tf.matmul(x_train, W) + b)))를 간단하게 tensoflow함수를 이용한 것이다.

```
xy = np.loadtxt('data-04-zoo.csv', delimiter=',', dtype=np.float32)
x_data = xy[:, 0:-1]
y_data = xy[:, -1]

nb_classes = 7  # 0 ~ 6

# Make Y data as onehot shape
Y_one_hot = tf.one_hot(y_data.astype(np.int32), nb_classes)

W = tf.Variable(tf.random.normal((16, nb_classes)), name='weight')
b = tf.Variable(tf.random.normal((nb_classes,)), name='bias')
variables = [W, b]


# tf.nn.softmax computes softmax activations
# softmax = exp(logits) / reduce_sum(exp(logits), dim)
def logit_fn(X):
    return tf.matmul(X, W) + b
def hypothesis(X):
    return tf.nn.softmax(logit_fn(X))
def cost_fn(X, Y):
    logits = logit_fn(X)
    cost_i = tf.keras.losses.categorical_crossentropy(y_true=Y, y_pred=logits,from_logits=True)
    cost = tf.reduce_mean(cost_i)
    return cost
def grad_fn(X, Y):
    with tf.GradientTape() as tape:
        loss = cost_fn(X, Y)
        grads = tape.gradient(loss, variables)
        return grads
def prediction(X, Y):
    pred = tf.argmax(hypothesis(X), 1)
    correct_prediction = tf.equal(pred, tf.argmax(Y, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    return accuracy
def fit(X, Y, epochs=2000, verbose=200):
    optimizer = tf.keras.optimizers.SGD(learning_rate=0.1)

    for i in range(epochs):
        grads = grad_fn(X, Y)
        optimizer.apply_gradients(zip(grads, variables))
        if (i == 0) | ((i + 1) % verbose == 0):
            #             print('Loss at epoch %d: %f' %(i+1, cost_fn(X, Y).numpy()))
            acc = prediction(X, Y).numpy()
            loss = cost_fn(X, Y).numpy()
            print('Steps: {} Loss: {}, Acc: {}'.format(i + 1, loss, acc))


fit(x_data, Y_one_hot)
```

결국에 이렇게 구현을 할 수 있고, 중요한 것은 현재에도 기울기 최소 최적 알고리즘을 이용하고 있다는 것이다. W와 b라는 두개의 변수를 이용해 잔차 함수를 구하고, 본래는 3차원 공간에 그려져야 하지만 2차원 공간에, 각각의 점을 z축의 세워진 직선이라고 생각하면 된다. 그리고 GradientTape()를 이용해서 잔차 함수를 W,b두 매개 변수로 편미분을 해 그 기울기가 0이 되는 시점을 찾으면 된다. 따라서 learning_rate의 적절한 설정이 중요한데, 자칫 잘못하면 최솟값을 넘어가 /지나쳐 버리는 일이 발생 할 수도 있기 때문이다.

7. Applications & Tips





위와 같은 함수가 있을 때 기울기가 양의 방향, 음의 방향이 모두 있기 때문에 왔다 갔다 하면서, 즉 기울기가 양수일때와 음수일때를 오가는 것이 아닌 오른쪽 그림의 빨간 선처럼 최종적으로 cost함수의 극솟값에 도달하도록 learning rate를 조절해 주어야 한다.

```
W_grad, b_grad = tape.gardient(cost, [W,b])
W.assign_sub(learning_rate*W_grad)
b.assigh_sub(learning_rate*b_grad)
```

이렇게 코드를 짜는 것이 바로 위의 그래프의 기울기를 이용하는 것과 동일하다. 그런데 이 learning_rate를 계산 할 때에 계속해서 변화를 주는 방법도 존재한다. Annealing the learning_rate인데

```
def exponential_decay(epoch):
    starter_rate = 0.01
    k = 0.96
    exp_rate = starter_rate * exp(-k*t)
    return exp_rate
```

이런 방법으로 코드를 짜서 learning_rate에 변화를 준다. 이렇게 하면 더 정확하게 오차의 최솟값에 도달하도록 교육이 가능해진다.

데이터를 전처리하는 과정에서 표준화하는 방법과 정교화 하는 방법에 대해 알아보면 '밀집된' 값들을 알아보기 위해 $X_{new} = \frac{x-\mu}{\sigma}$

이 방법을 사용한다. 이는 해당 x값에서 평균을 뺀 뒤에 표준편차로 나누는 과정이고, 코드를 짜면

```
standardization = ((data - np.mean(data))/sqrt(np.sum(data-np.mean(data))^2))/np.count(data))
```

정교화 과정은 0에서 1 사이의 값으로 데이터를 변형시키는 과정이다. $X_{new} = \frac{x-x_{min}}{x_{max}-x_{min}}$ 이렇게 최대 최소의 차로 현재 데이터에서 최솟값을 뺀 값을 나눈다.

```
normalization = (data - np.min(data, theta)) / (np.max(data, theta) - np.min(data, theta))
```

이런 식으로 코드가 짜여짐을 확인 할 수 있다.

```
# 학습시킬 값들을 batch_size 에 맞춰서 담는다
dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train)).batch(len(x_train))
# W,b 는 학습을 통해 생성되는 모델에 쓰이는 Weigth 와 bias 이다.
W = tf.Variable(tf.random.normal((3, 3)))  # (가로로 요소의 개수, 세로로 요소의 개수)
b = tf.Variable(tf.random.normal((3,)))


# softmax 함수로 가설을 선언하는데, 0~1 사이의 값으로 구함
def soft_max(features):
    hypothesis = tf.nn.softmax(tf.matmul(features, W) + b)
    return hypothesis


# loss, 혹은 cost 를 출력하는 함수(가설을 검증하기 위해 쓰임)
# cross entropy loss 를 사용
def loss_fn(hypothesis, features, labels):
    cost = tf.reduce_mean(-tf.reduce_mean(labels * tf.math.log(hypothesis), axis=1))
    return cost


# learning rate 값을 조정하기 위한 learning decay 설정
# starter_learning_rate, global_step = 최초 학습 시 learning rate, 현재 학습 회수
is_decay, starter_learning_rate = True, 0.1
if (is_decay):
    learning_rate = tf.keras.optimizers.schedules.ExponentialDecay(initial_learning_rate=starter_learning_rate,
                                                                   decay_steps=1000,
                                                                   decay_rate=0.96,
                                                                   staircase=True)
    optimizer = tf.keras.optimizers.SGD(learning_rate)
else:
    optimizer = tf.keras.optimizers.SGD(learning_rate=starter_learning_rate)


def grad(hypothesis, features, labels):
    with tf.GradientTape() as tape:
        hypothesis = soft_max(features)
        loss_value = loss_fn(hypothesis, features, labels)
    return tape.gradient(loss_value, [W, b])
```

```
# 가설을 통해 실제 값과 비교한 정확도를 측정한다.
def accuracy_fn(hypothesis, labels):
    prediction = tf.argmax(hypothesis, 1)
    is_correct = tf.equal(prediction, tf.argmax(labels, 1))
    accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))
    return accuracy


# 위의 data를 cost 함수를 통해 학습시킨 후 모델을 생성한다.
epochs = 1001
for i in range(epochs):
    for features, labels in iter(dataset):
        features = tf.cast(features, tf.float32)
        labels = tf.cast(labels, tf.float32)
        grads = grad(soft_max(features), features, labels)
        optimizer.apply_gradients(grads_and_vars=zip(grads, [W, b]))
        if i % 100 == 0:
            print('iter: {}, Loss: {:.4f}'.format(i, loss_fn(soft_max(features), features, labels)))
x_test = tf.cast(x_test, tf.float32)
y_test = tf.cast(y_test, tf.float32)
test_acc = accuracy_fn(soft_max(x_test), y_test)
print('Test Accuracy: {:.4f}'.format(test_acc))
```

다음으로 공부한 내용은 linear regression에 Eager(normalization, decay, l2_loss)를 적용한 것이다.

L2_loss라는 함수를 만든 이유는 loss값이 너무 작거나(under fitting), 혹은 너무 큰 경우(over fitting)일 경우를 위해 learning rate값을 조절하는 것을 멈추고 지속하는 것을 결정하기 위함이다. 너무 작다면 오차를 고려하지 못하여 여러 test case를 맞추지 못할 것이다. Learning rate를 좀더 정형화된 값으로 유지하기 위해서 실행 했었던 'learning rate annealing'에 오차가 발생할 수도 있다는 의미이다. 따라서 이 상황에서는 learning rate에 곱해주던 decay_rate를 멈추거나 실행하는 등을 유동적으로 가능하도록 learning decay 과정을 진행한다.

Learning rate -> gradient, good and bad learning rate, annealing the learning rate(decay)

Data preprocessing -> standardization/normanalization, noisy data

linear regression with regularization:

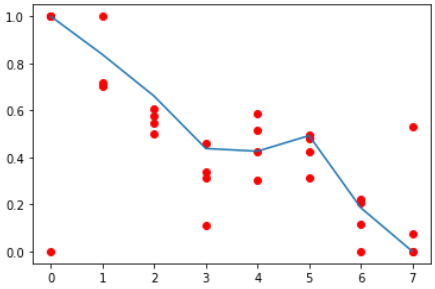
model: $$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

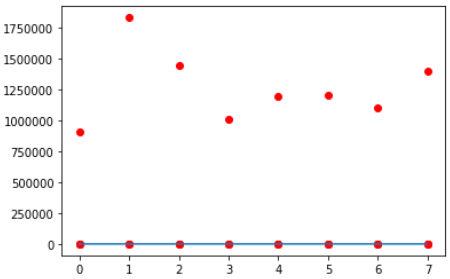
$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}((h_\theta(x^{(i)}) - y^{(i)}))^2 + \frac{\tau}{2m}\sum_{j=1}^{m}\theta_j^2$$

Overfitting -> set a features, regularization

```
# normalizaion, 즉 특징 정교화의 과정을 거쳐 불필요한 과정 제외 0~1사이의 값으로 반환
def normalization(data):
    numerator = data - np.min(data, 0)
    denominator = np.max(data, 0) - np.min(data, 0)
    return numerator / denominator

# data를 기준으로 linear regression
dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train)).batch(len(x_train))
W = tf.Variable(tf.random.normal((4, 1)), dtype=tf.float32)
b = tf.Variable(tf.random.normal((1,)), dtype=tf.float32)


def hypo(features):
    hypothesis = tf.matmul(features, W) + b
    return hypothesis
```

```python
# weight의 수가 많아지면 수만큼 더한다
# tf.nn.l2_loss(W2)+tf.nn.l2_loss(W3)
def l2_loss(loss, beta=0.01):
    W_reg = tf.nn.l2_loss(W)
    loss = tf.reduce_mean(loss + W_reg * beta)
    return loss


# 가설을 검증할 cost함수 정의
def loss_fn(hypothesis, features, labels, flag=False):
    cost = tf.reduce_mean(tf.square(hypothesis - labels))
    if flag:
        cost = l2_loss(cost)
    return cost


# learning rate 조정
is_decay, starter_learning_rate = True, 0.1
if (is_decay):
    learning_rate = tf.keras.optimizers.schedules.ExponentialDecay(initial_learning_rate=starter_learning_rate,
                                                                   decay_steps=50,
                                                                   decay_rate=0.96,
                                                                   staircase=True)
    optimizer = tf.keras.optimizers.SGD(learning_rate)
else:
    optimizer = tf.keras.optimizers.SGD(learning_rate=starter_learning_rate)


def grad(hypothesis, features, labels, l2_flag):
    with tf.GradientTape() as tape:
        loss_value = loss_fn(hypo(features), features, labels, l2_flag)
    return tape.gradient(loss_value, [W, b]), loss_value


# 학습 진행
epochs = 101
for i in range(epochs):
    for features, labels in dataset:
        features = tf.cast(features, tf.float32)
        labels = tf.cast(labels, tf.float32)
        grads, loss_value = grad(hypo(features), features, labels, False)
        optimizer.apply_gradients(grads_and_vars=zip(grads, [W, b]))
    if i % 10 == 0:
        print('lter:{},loss: {:.4f}'.format(i, loss_value))
```






```
lter:0,loss: 1.7346
lter:10,loss: 0.0745
lter:20,loss: 0.0438
lter:30,loss: 0.0273
lter:40,loss: 0.0181
lter:50,loss: 0.0128
lter:60,loss: 0.0099
lter:70,loss: 0.0080
lter:80,loss: 0.0068
lter:90,loss: 0.0060
lter:100,loss: 0.0054
```

Lab 7-5 Fashion MNIST introduction

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow import keras
tf.random.set_seed(777)
#keras 를 활용한 fashion MNIST 분류 모델 생성

fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
#fashion MNIST Data 확인
plt.figure()
plt.imshow(train_images[3])
plt.colorbar()
plt.grid(False)

#위의 data 를 이용하여 분류 모델 만들기
#0~1 사이의 값으로 정규화 및 data 출력
train_images = train_images/255.0
test_images = test_images / 255.0
plt.figure(figsize=(10,10))  #사진 크기
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i],cmap = plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])

#tensorflow keras API 를 이용해 모델에 대한 정의
model = keras.Sequential([
    keras.layers.Flatten(input_shape = (28,28)),
    keras.layers.Dense(128, activation = tf.nn.relu),
    keras.layers.Dense(10, activation = tf.nn.softmax)
])
#Adam optimizer 과 Cross Entropy Loss 선언
#5Epoch 로 학습할 Data 로 학습 수행
model.compile(optimizer = 'adam',
              loss = 'sparse_categorical_crossentropy',
              metrice = ['accuracy'])
model.fit(train_images, train_labels, epochs= 5)
test_acc= model.evaluate(test_images, test_labels)
```

mnist라는 이미지 데이터가 저장되어 있는 데이터 베이스에서의 데이터를 꺼내어 사용하는 과정을 구현해 보았다.

이미지를 행렬로 나타내어 0에 가까울수록 흰색, 1에 가까울수록 검은색을 띄도록 구현을 한 것이다. Mnist.train.labels는 TensorShape([Dimension(55000), Dimension(10)]의 구조를 갖는 텐서이다.

Lab 7-6 IMDB-introduction

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
tf.random.set_seed(777)
#학습에 쓰이는 data
imdb = keras.datasets.imdb
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words = 10000)
#A dictionary mapping words to an integer index
word_index = imdb.get_word_index()
#The first indicies are reserved
#data 전처리 과정
word_index = {k:(v+3) for k,v in word_index.items()}
word_index["<PAD>"] = 0
word_index["<START>"] = 1
word_index["<UNK>"] = 2
word_index["<UNUSED>"] = 3
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
def decode_review(text):
    return ' '.join([reverse_word_index.get(i,'?') for i in text])

#data 를 기준으로 분류 모델 만들기
train_data = keras.preprocessing.sequence.pad_sequences(train_data, value = word_index["<PAD>"],padding='post',maxlen =
256)
test_data = keras.preprocessing.sequence.pad_sequences(test_data, value = word_index['<PAD>'],padding = 'post',
maxlen=256)
#tensorflow keras API 를 통해 모델에 대한 정의
#입력 size 와 학습시킬 layer 의 크기와 activation function 정의
vocab_size = 10000
model = keras.Sequential()
model.add(keras.layers.Embedding(vocab_size, 16))
model.add(keras.layers.GlobalAveragePooling1D())
model.add(keras.layers.Dense(16, activation=tf.nn.relu))
model.add(keras.layers.Dense(1, activation=tf.nn.sigmoid))
model.summary()

#Adam Optimizer 과 Cross Entropy 생성
model.compile(optimizer = 'adam', loss = 'binary_crossentropy',metrics = ['accuracy'])
#모델을 평가할 test data 에 대한 정의(10000 을 기준으로 학습과 평가 데이터 생성)
x_val = train_data[:10000]
partial_x_train = train_data[10000:]
y_val = train_labels[:10000]
partial_y_train = train_labels[10000:]

history = model.fit(partial_x_train, partial_y_train, epochs = 40, batch_size = 512,validation_data = (x_val,
y_val),verbose=1)
results = model.evaluate(test_data, test_labels)
print(results)
```

iMDB는 자연어 처리를 하기 위한 database이고, 여기에서의 data를 이용해 자연어 처리에 필요한 전처리 과정과 학습, 예측까지의 과정을 짠 코드이다.

MACHINE LEARNING BEGINNER

<선형 관계>

Y는 예측하려는 값, m은 기울기, x는 입력 특성 값, b는 y의 절편이다. $y = mx + b$

Y'은 얻고자 하는 출력(예측된 라벨), b는 편향(y절편, 또는 w_0), w_i는 i번째 가중치, x_i는 특성(알려진 입력값) 이다.

$$y' = b = w_1x_1 + w_2x_2 + w_3x_3$$

모델을 학습 시킨다는 것은 라벨이 있는 데이터로부터 올바른 가중치와 편향값을 학습하는 것이다. 손실(loss, cost)는 한 가지 예에서 모델의 예측이 얼마나 잘못되었는지를 나타내는 수이다. 모젤의 예측은 따라서 손실이 작을수록 완벽하고 학습의 목표는 모든 예에서 평균적으로 작은 손실을 갖는 가중치와 편향의 집합을 찾는 것이다.

<손실함수의 예>

1. L2함수(제곱 손실)

```
= the square of the difference between the label and the prediction
  = (observation - prediction(x))^2
  = (y - y')^2
```

평균 제곱 오차(MSE)는 예시당 평균 제곱 손실이다. 이를 계산하려면 개별 예의 모든 제곱 손실을 합한 다음 예의 개수로 나눈다.

$$MSE = \frac{1}{N}\sum_{(x,y)\in D}(y - prediction(x))^2$$

(x,y)는 예이고 prediction(x)는 특성 집합 x와 결합된 가중치 및 편향의 함수이고 D는 (x,y)쌍과 같이 여러 라벨이 있는 예가 포함된 데이터 세트, n은 D에 포함된 예의 개수이다.

머신러닝 모델이 반복을 통해 손실을 줄이는 방법에 대해 알아보자. 머신러닝 시스템은 손실 함수의 값을 검토하여 b와 w_1의 새로운 값을 생성한다. 그리고 손실 값이 가장 낮은 모델 매개변수를 발견할 때까지 반복 학습을 한다.

경사 하강법 알고리즘은 기울기에 학습률/보폭 이라고 불리는 스칼라를 곱하여 다음 지점을 결정한다. 학습률은 너무 작으면 최솟값을 지나버리게 되거나 너무 크면 시간이 오래 걸릴 수 있다. 따라서 적정한 learning_rate를 지정해 주는 것이 중요하다.

배치, 즉 데이터의 개수가 너무 커지면 단일 밤복으로 계산하는데 너무 오랜 시간이 걸릴 수 있으며 중복된 데이터를 계산하고 있을 가능성이 높아진다. 만약에 훨씬 적은 계산으로 적절한 기울기를 얻을 수 있다면 훨씬 적은 데이터 세트로 중요한 평균값을 추청 할 수 있을 것이다. 이 아이디어를 확장한 것이 확률적 경사 하강법(SGD)이고, 반복당 하나의 예만을 사용한다. 소규모 배치, 또는 예가 하나뿐인 배치(SGD)는 경사 하강법을 수행하는데에 있어서 훨씬 간단한다. 알고리즘은 반복마다 무작위로 예를 확보하곤 한다.

<머신러닝에서 사용되는 초매개변수>

Steps: 총 학습 반복 횟수

Batch size: 하나의 단계와 관련된 예시의 수(임의로 선택됨) 예를 들면 SGD의 batch size는 1이다.

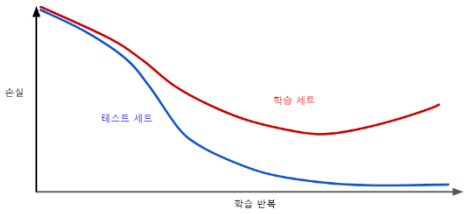
<학습 및 평가 세트: 데이터 분할>

우리는 머신 러닝 전체 과정에서 사용될 데이터 세트 하나를 학습 세트와 평가 세트로 분할하여야 한다. 뿐만 아니라 과적합(overfitting)현상을 막기 위해서는 데이터 세트 하나를 셋, 즉 학습, 검증, 테스트 세츠로 분할 하는 것을 권장한다. 학습 세트로 모델을 학습 시킨 뒤에 검증세트로 평가한 결과에서 가장 우수한 결과를 보이는 모델을 선택 한 뒤에 테스트 세트의 결과를 확인하는 것이다.

One-hot-incoding은 예시에 적용되는 값의 경우 관련 벡터 요소 1개만을 1로 설정하고 다른 요소는 모두 0으로 설정하는 것이고, 여러 값이 1일때는 multi-hot-incoding이라고 한다. 특성 값 조정이란 부동 소수점 특성 값을 100~900 등의 자연 범위에서 0~1 또는 -1~+1 등의 표준 범위로 변환하는 작업이다. 특성 세트가 단일 특성으로만 구성된 경우 조정에 따르는 실질적인 이점은 거의 없습니다. 그러나 특성 세트가 여러 특성으로 구성되었다면 특성 조정으로 다음과 같은 이점을 누릴 수 있으며 경사하강법이 더 빠르게 수렴

하고 'NaN 트랩'이 방지된다. NaN 트랩이란 모델의 숫자 중 하나가 NaN(예: 학습 중에 값이 부동 소수점 정밀도 한도를 초과하는 경우)이 된 후 수학 연산 과정에서 모델의 다른 모든 숫자가 결국 NaN이 되는 상황을 의미한다.

<정규화: 단순성>



왼쪽에 제시된 일반화 곡선은 학습 반복 횟수에 대해 학습 세트와 검증 세트의 손실을 보여준다.

모델 복잡도가 가중치에 대한 함수인 경우 높은 절대값을 사용하는 특성 가중치는 낮은 절댓값을 사용하는 특성 가중치보다 더 복잡하다. 모든 특성 가중치를 제곱한 값의 합계로 정규화 항을 정의하는 L2정규화 공식을 사용하여 복잡도의 수치화가 가능하다. 이상점 가중치는 큰 영향을 미치는데, 따라서 데이터의 가중치가 [0.5, 0.2, 5, 1, 0.25, 0.3]과 같이 하나만 유독 클 경우에 사용한다.

모델 개발자는 람다(τ)라는 스카라(정규화율)을 정규화 항의 값에 곱하여 정규화 항의 전반적인 영향을 조정한다.

$$최소화(손실(데이터|모델) + \tau 복잡도(모델))$$

이를 수행하면 가중치 값을 0으로 유도하고 정규 분포를 사용하여 가중치 평균을 0으로 유도한다.

<로지스틱 회귀: 확률 계산>

많은 경우 로지스틱 회귀 출력을 이진 분류 문제의 해결 방법으로 매핑한다. Logistic regression의 모델의 출력이 항상 0과 1 사이의 값인 이유는 sigmoid 함수를 이용하기 때문이다.

Z가 logistic regression을 사용하여 학습된 모델의 선형 레이어의 출력을 나타내는 경우 sigmoid(z)는 $y' = \frac{1}{1+e^{-z}}$ 이다.

Z는 사실상 sigmoid 함수의 역수이므로 $z = \log\left(\frac{y}{1-y}\right)$ 이다.

<로지스틱 회귀의 손실 함수>

선형 회귀의 손실 함수는 제곱 손실이다. 로지스틱 회귀의 손실 함수는 로그 손실로 아래와 같이 정의된다.

$$\sum_{(x,y)\in D} -ylog(y') - (1-y)\log(1-y')$$

Y는 라벨이 있는 예의 라벨이고 로지스틱 회귀이므로 y의 값은 모두 0 또는 1이다. Y'은 x의 특성 세트에 대한 예측값 0~1 사이이다.

정규화는 로지스틱 회귀 모델링에서 매우 중요하다. 정규화하지 않으면 로지스틱 회귀의 점근 특성이 고차원에서 계속 손실을 0으로 만들려고 시도한다. 결과적으로 대부분의 로지스틱 회귀 모델에서 모델 복잡성을 줄이기 위해 L2 정규화 또는 조기 중단, 즉 학습 단계 수 또는 학습률을 제한하는 방법을 이용한다.

<분류: 예측 편향>

로지스틱 회귀 예측은 편향되어서는 안된다. 따라서 예측 편향, 즉 두 평균이 서로 얼마나 멀리 떨어져 있는지 측정하는 수량은 관찰 평균과 수렴해야 한다.

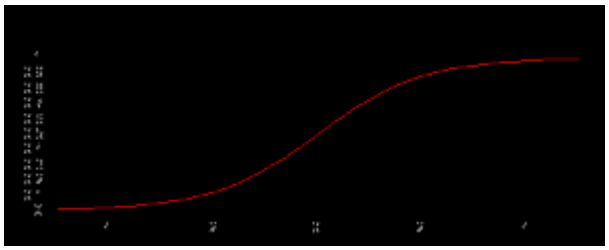
이 때문에 L1정규화를 이용한다. 이를 사용해 모델에서 유용하지 않은 많은 계수를 정확히 0이 되도록 유도하여 추론 단계에서 RAM을 절약할 수 있다. L1 정규화와 L2 정규화는 각기 다른 방법으로 가중치에 페널티를 주는데, 결과적으로 L1과 L2는 서로 다르게 미분이 된다. L2의 미분계수는 2*가중치, L1의 미분계수는 k(가중치와 무관한 값을 갖는 상수)이다.

L2의 미분계수는 매번 가중치의 x%만큼 제거하는 것이고, L1의 미분계수는 매번 가중치에서 일정 상수를 빼는 것으로 생각하면 된다.

<활성화 함수와 신경망 학습>

우리는 선형 모델로 충분히 학습이 되지 않는 모델에 대해 모델링을 위해 비선형성을 직접 도입할 수 있다.

아래 그림의 그래프로 나타낸 모델에서 히든 레이어 1의 각 노드 값이 비선형 함수로 변환된 후에 다음 레이어의 가중합으로 전달된다. 이 비선형 함수를 활성화 함수라고 한다. 대표적인 활성화 함수로 sigmoid함수가 존재한고 이는 가중 합을 0과 1 사이의 값으로 변환.

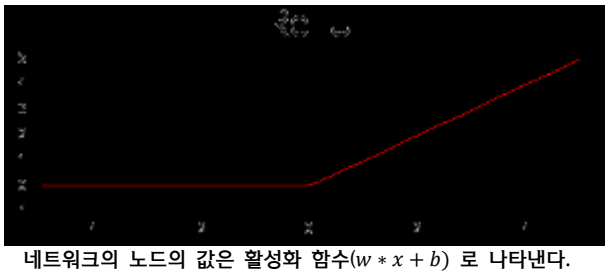


$F(x) = \frac{1}{1+e^{-x}}$ 이런 형태로 생긴 함수가 sigmoid이다.

정류 선형 유닛(ReLU)활성화 함수는 시그모이드와 같은 매끄러운 함수보다 효과적이면서 계산이 쉽다.

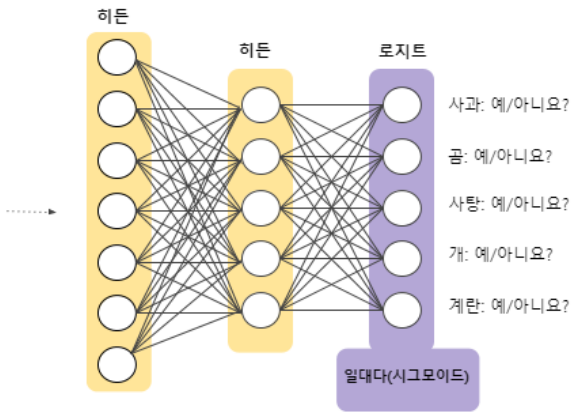
$$F(x) = \max(0, x)$$

위와 같은 식을 가지는 함수이고 위와 달리 시그모이드의 반응성은 양측에서 상대적으로 빨리 떨어진다.



네트워크의 노드의 값은 활성화 함수($w * x + b$) 로 나타낸다.

<다중 클래스 신경망>



출력 노드가 다른 클래스를 나태내는 심층 신경망을 이용하여 훨씬 더 효율적인 일대다 모델을 만들 수 있다. 왼쪽 그림은 해당 과정을 보여준다.

로지스틱 회귀는 0과 1.0사이의 소수를 생성한다.

소프트맥스(softmax)는 이 아이디어를 다중 클래스에 적용한다. 즉 소프트맥스가 다중 클래스 문제의 각 클래스에 probabiity를 할당한다. 이 소수 확률의 합은 무조건1이다.

$$p(y = j|x) = \frac{e^{(w_j^T x + b_j)}}{\sum_{k \in K} e^{(w_k^T x + b_k)}}$$

위의 수식이 소프트맥스 방정식이고 이 수식은 기본적으로 로지스틱 회귀 수식을 다중 클래스로 확장한다.

단, 소프트 맥스 방정식을 사용하는 데에도 한계가 있는데, 각 예가 정확히 한 클래스의 멤버여야만 한다. 그렇지 않은 경우, 예를 들면 모든 종류의 클래스가 포함된 예가 존재하는 경우 logistic regression을 여러 번 사용해야 한다.

<임베딩>

벡터 간의 의미있는 관계가 부족할 때에는 임베딩이라는 솔루션이 존재한다. 이는 크기가 큰 희소 벡터를 의미론적 관계를 보존하는

저차원 공간으로 변환하는 것이다. 임베딩은 하나의 행렬이고, 행렬의 각 열은 어휘 항목 하나에 대응한다. 단일 어휘 항목에 대한 밀집 벡터를 얻으려면 해당 항목에 대응하는 열을 검색한다.