Minjung Gim(mjgim.nims.re.kr), National Institute for Mathematical Sciences

특징 추출을 위한 Auto-encoder와 주성분 분석(PCA)

주어진 데이터 X가 고차원이어서(변수의 개수가 많음) 변수(또는 특징)들 끼리의 관계를 한눈에 보기 어렵거나 데이터의 특징이 직관적으로 와닫지 않는 경우가 많다. 데이터의 특징을 잘 표현하면서 처리하기 쉬운 저차원의 벡터로 변환(transform)하는 기법이 특징 추출(feature extraction)이다. 즉, 데이터 변환을 통해서 새로운 변수를 만드는 방법이다(새로 만든 변수를 사용해서 모델 제작). 특징 추출의 방법으로 주로 쓰이는 Auto-encoder와 주성분 분석(Principal component analysis, PCA)에 대해 알아보자.

특징 추출 & 특징 선택

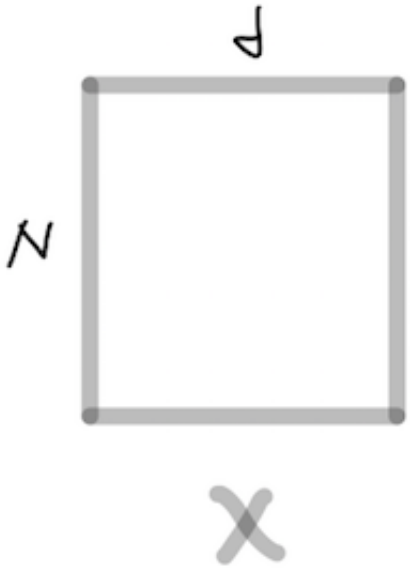
특징 추출과 유사한 차원축소(dimensionality reduction) 기법으로는 특징 선택(feature selection)이 있다. 특징 추출이 모든 독립변수(특징)을 사용하여 변수를 변환해 새로운 변수를 만드는 것이라면 특징 선택은 유의미한 변수들을 남기고 불필요한 특징들을 제거하는 기법이다. 즉, 주어진 독립변수에서 문제를 해결하는데 유의미한 변수(특징)를 찾는 방법이다.

예를 들어서 학생들의 기말고사 성적을 예측하는 모델에서 중간고사, 퀴즈1, 퀴즈2, 학생의 키 4개 독립변수를 사용했다고 가정하자. 4개 변수 모두를 사용해서 기말고사 성적을 예측하는 모델을 만들고 변수를 한개씩 제거해서 만든 모델과 비교해서 유의미하지 않은 변수를 제거한다. 아마 위 경우 학생들의 키는 기말고사 성적과 연관성이 없어 보이기 때문에 제거될 확률이 높다.

특징 선택 방법 중에서 **Variance Threshold**를 소개하면 이 알고리즘은 X의 변수들 중에서 분산이 작은 변수를 제거하는 방식이다. 변수의 분산이 상대적으로 작다고 유의미하지 않다고는 할 수 없다. 하지만 분산이 0에 가까운 변수들은 모든 데이터들이 유사하거나 동일한 값을 갖기 때문에 **Variance Threshold**를 통한 특징 선택은 의미가 있다.

데이터 구조

우리은 다루고자 하는 데이터 X는 d차원 벡터가 N개 있는 $N \times d$ 행렬 (2차원 array)이다(일반적으로 Python으로 데이터를 분석하는 경우 데이터를 이런 모양으로 구성한다). 여기서 데이터의 개수 N은 차원 d보다 많다고 가정한다($N > d$).



주성분 분석

PCA는 특징 추출(feature extraction) 기법 중에서 널리 쓰이는 방법으로 변수들의 분산(variance)을 최대로 하는 R^d 의 기저를 찾아서 선형변환(linear transform)을 통해 새로운 변수를 생성한다. 그리고 생성된 새로운 각 변수들의 분산값들을 고려해서 분산이 작은 변수를 제거하는 기법이다.

$\mathbf{x} \in R^d$ 이고 $\{\mathbf{v}_1, \dots, \mathbf{v}_d\}$ 이 R^d 의 orthonormal basis라고 하자. 즉, $\mathbf{v}_i \in R^d$ 이고

$$\langle \mathbf{v}_i, \mathbf{v}_j \rangle = \delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

을 만족한다. $\mathbf{x}$를 $\{\mathbf{v}_1, \dots, \mathbf{v}_d\}$으로 표현하면 다음과 같다.

$$\underbrace{\langle \mathbf{x}, \mathbf{v}_1 \rangle}_{\text{첫 번째 성분}} \mathbf{v}_1 + \underbrace{\langle \mathbf{x}, \mathbf{v}_2 \rangle}_{\text{두 번째 성분}} \mathbf{v}_2 + \cdots + \langle \mathbf{x}, \mathbf{v}_d \rangle \mathbf{v}_d$$

데이터 X는 $N \times d$행렬이기 때문에 X의 orthonormal basis 열벡터로 구성된 $d \times d$ 행렬 $V = [\mathbf{v}_1, \dots, \mathbf{v}_d]$에 대한 선형변환

$$Z := XV$$

변수(열 벡터)들의 분산이 최대가 되는 V를 찾아야 한다($\{\mathbf{v}_1, \dots, \mathbf{v}_d\}$ 은 R^d 의 orthonormal basis). 여기서 cov(X) 계산 편의를 위해 데이터 X의 각 열의 평균은 0이라고 가정한다. 평균이 0이 아닌 경우에는 아래 코드와 같이 평균을 0으로 맞출 수 있다.

```
X = np.array([[1,2,3],
[2,3,4],
[3,4,5],
[1,2,5]])

X = X - X.mean(axis=0)
```

$$X = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \\ 1 & 2 & 5 \end{pmatrix} \Longrightarrow \begin{pmatrix} -0.75 & -0.75 & -1.25 \\ 0.25 & 0.25 & -0.25 \\ 1.25 & 1.25 & 0.75 \\ -0.75 & -0.75 & 0.75 \end{pmatrix}$$

다시 본론으로 돌아와서 $Z = XV$의 각 열들의 분산을 최대로 만드는 V를 찾는 방법을 살펴보자. Orthonormal basis 중 한 벡터(V의 열 벡터)인 $d \times 1$ 행렬을 $\mathbf{v}$라고 하면 $\mathbf{v}$는 다음을 만족한다.

$$\langle \mathbf{v}, \mathbf{v} \rangle = \mathbf{v}^T \mathbf{v} = 1.$$

위 식을 만족하는 $\mathbf{v}$ 중에서

$$\mathrm{Var}(X\mathbf{v}) = \mathbf{v}^T \mathrm{Cov}(X)\mathbf{v} = \mathbf{v}^T \left(\frac{1}{N-1} X^T X \right) \mathbf{v}$$

값이 최대가 되는 $\mathbf{v}$는 라그랑즈 승수법(method of Lagrange multipliers)을 이용하여 찾을 수 있다. $\mathbf{v} \neq \mathbf{0}$이므로 식(1)과 (2)의 $\mathbf{v}$에 대한 gradient들이 평행(parallel)이 되는 즉, 다음 등식이 성립하는 $\lambda \in R$과 $\mathbf{v}$를 찾는 것이다.

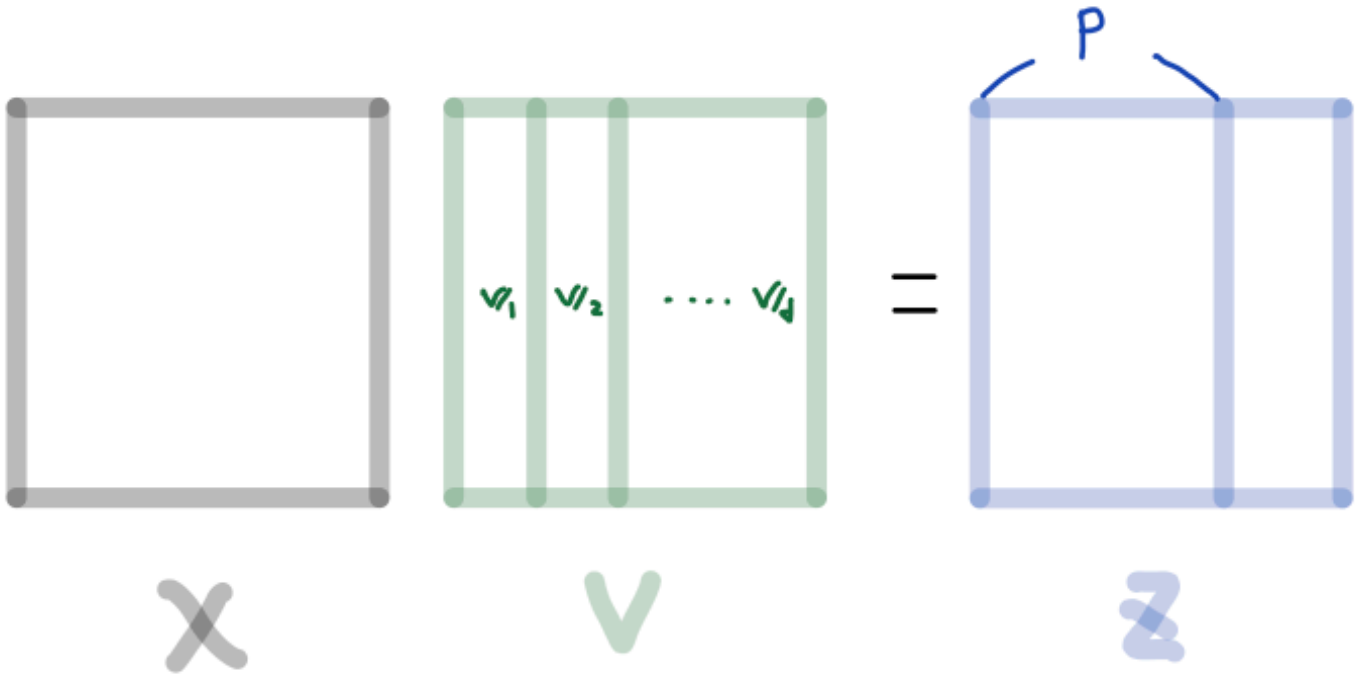
$$\lambda \mathbf{v} = \mathrm{Cov}(X)\mathbf{v}.$$

또한, (1), (2), (3)을 결합하면

$$\mathrm{Var}(X\mathbf{v}) = \mathbf{v}^T \mathrm{Cov}(X)\mathbf{v} = \mathbf{v}^T (\lambda \mathbf{v}) = \lambda$$

을 만족한다. 결론적으로 데이터 X의 선형변환으로 부터 얻은 새로운 데이터 $Z = XV$ 변수들의 분산을 최대로 하는 $V = [\mathbf{v}_1, \ldots, \mathbf{v}_d]$의 열벡터들은 Cov($X$)의 eigen vevtor들이고 N차원 벡터 $X\mathbf{v}$의 분산은 eigen vector $\mathbf{v}$에 대응하는 eigen value λ이다.

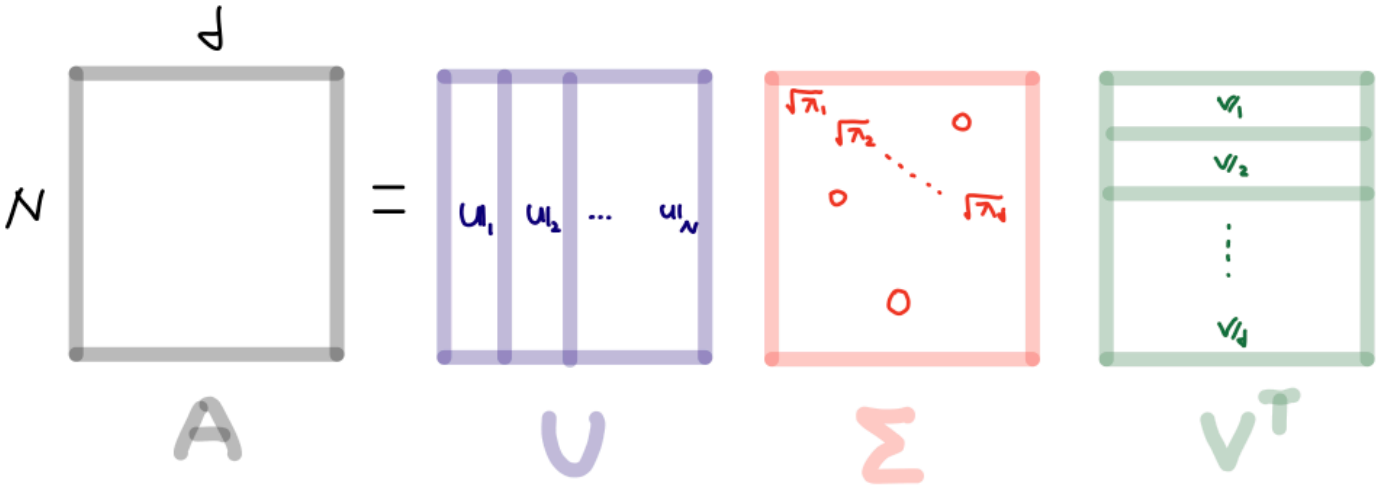
결론적으로 선형변환으로 얻은 $Z = XV$의 d개 변수 중에서 분산값들이 큰 변수들 p개만 특징 선택해서 새로운 변수로 사용하는 것이 주성분 분석 PCA다.



특이값 분해(Singular Value Decomposition, SVD)

Cov$(X) = \left(\frac{1}{N-1}X^TX\right)$의 eigen vector들인 V는 특이값 분해(SVD)를 이용해서 찾을 수 있다. 특이값 분해란 $N \times d$ 행렬 A를 다음과 같이 분해하는 것이다.

$$A = U\Sigma V^T$$



- $U = [\mathbf{u}_1, \ldots, \mathbf{u}_N]$, $\mathbf{u}_i \in R^N$은 $N \times N$ 행렬이고 $V = [\mathbf{v}_1, \ldots, \mathbf{v}_d]$, $\mathbf{v}_i \in R^d$은 $d \times d$ 행렬로 다음을 만족한다.

$$U^T U = I_N, \quad V^T V = I_d$$

- Σ는 $N \times d$ 행렬로 $A^T A$의 d개의 eigen value $\lambda_1 \geq \lambda_2 \cdots \geq \lambda_d \geq 0$들의 제곱근을 대각성분으로 갖는 $d \times d$ 대각행렬(diagonal matrix)과 모든 원소가 0 인 $(N-d) \times d$행렬의 행 결합(row bind)으로 구성 되어있다.

```
X = np.array([[1,2,3],
[2,3,4],
[3,4,5],
[1,2,5]])

def SVD(x):
    x = x - x.mean(axis=0)  #각 열의 평균을 0으로 조정
    U, s, Vt = np.linalg.svd(x) #Numpy에서 제공하는 SVD 함수
    st= np.diag(s)
    ss = np.concatenate((st, np.zeros([x.shape[0]-x.shape[1],x.shape[1]])), axis=0)
    return U, ss, Vt

SVD(X)
```

위 코드를 실행하면 X를 아래와 같이 분해할 수 있다(소숫점 3자리에서 반올림).

$$\underbrace{\begin{pmatrix} -0.75 & -0.75 & -1.25 \\ 0.25 & 0.25 & -0.25 \\ 1.25 & 1.25 & 0.75 \\ -0.75 & -0.75 & 0.75 \end{pmatrix}}_{X}$$

$$= \underbrace{\begin{pmatrix} -0.595 & 0.478 & -0.645 & 0. \\ 0.082 & 0.277 & 0.129 & 0.949 \\ 0.760 & 0.075 & -0.645 & 0. \\ -0.247 & -0.830 & -0.387 & 0.316 \end{pmatrix}}_{U} \underbrace{\begin{pmatrix} 2.523 & -0 & 0 \\ 0 & 1.373 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}}_{\Sigma} \underbrace{\begin{pmatrix} 0.635 & 0.635 & 0.439 \\ 0.311 & 0.311 & -0.898 \\ -0.707 & 0.707 & 0. \end{pmatrix}}_{V^T}$$

Remark of PCA

- PCA는 X의 선형변환 $Z = XV$ 변수들의 분산이 최대가 되도록 하는 V를 찾고 분산이 큰 변수들만 선택해서 차원 축소하는 기법이다. 여기서 V는 $X^T X$의 eigen vector 열 벡터로 구성된 행렬 $V = [\mathbf{v}_1, \ldots, \mathbf{v}_d]$ 이다. 이 벡터들을 주성분이라고 부르고 각 $\mathbf{v}_i$에 대응하는 eigen value λ_i들은 다음과 같이 크기 순으로 배열한다.

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d \geq 0.$$

- $Z = XV$ 분산은 eigen value $(\lambda_1, \lambda_2, \ldots, \lambda_d)$와 비례하기 때문에 주성분의 개수 p를 선택할 때 앞에서 부터 p개를 선택하면 된다.
- Scikit-learn의 PCA 함수를 사용하면 각 열들의 평균을 0으로 조정할 필요없이 X데이터로 부터 PCA의 결과를 얻을 수 있다.

```
from sklearn.decomposition import PCA
```

- 데이터가 선형으로 분리되지 않은 경우 PCA를 이용해서 데이터의 차원을 낮추게 되는 경우 원래 데이터가 갖고 있는 특성을 잃어 버릴 수 있다. 이럴 때는 비선형 kernel PCA를 사용하거나 Auto-encoder를 사용해야 한다.

Auto-encoder

Auto-encoder는 특징 추출을 목적으로 주어진 데이터를 변환하는 비지도 학습(unsupervised) 방식의 딥러닝 알고리즘이다. 이미지 처럼 쉽게 눈에 보이는 형태로 나타낼 수 없어서 쉽게 특징을 찾기 어려울 때 주로 사용하는 방법이다. 즉, 변환된 데이터를 이용해서 데이터에서 찾기 어려웠던 단서를 찾기 위해 사용한다. 또한 학습이 잘 진행되지 않는 분류나 예측 문제에서 parameter들의 초깃값 설정을 위한 fine tuning에도 활용될 수 있다. Auto-encoder 모델은 다음과 같이 encoder와 decoder로 구성되어있다(encoder와 decoder map에 히든 레이어를 원하는 만큼 넣을 수 있다).

$$X \underbrace{\longrightarrow}_{encoder} L \underbrace{\longrightarrow}_{decoder} H$$

- L: 주어진 데이터 X의 변수(특징)들을 변환해서 얻은 새로운 변수
- H: 모델의 예측값으로 독립변수 X와 동일한 값으로 학습

비지도 학습(unsupervised learning)으로 종속변수 Y값은 모델 학습에 필요하지 않고 X값만으로 모델을 제작하는 것이 회귀분석(regression) 예측 모델과의 큰 차이점이다. 모델의 예측값 H가 X값과 가까워지도록 학습을 진행하기 때문에 X에서 encoder와 decoder 합성이 항등함수(identity function)가 되도록 하는 것이 학습의 목표다.

Cost function

Auto-encoder의 cost function은 최종 레이어인 H와 입력값 X와의 오차제곱평균 MSE로 정의한다.

$$\frac{1}{N}\sum_{i=1}^{N}(H_i - X_i)^2$$

데이터의 개수는 N개이고 X_i와 $H_i := H(X_i)$는 i번 째 입력과 예측값이다. 여기에 과적합(overfitting)을 피하기 위해 다음과 같이 regularization term을 추가할 수도 있다.

$$\frac{1}{N}\sum_{i=1}^{N}(H_i - X_i)^2 + \lambda \sum_{weight} W^2$$

Encoder를 통해서 변환된 새로운 변수 L의 노드의 수(차원 수)는 일반적으로 X의 차원보다 작게해서 데이터 차원을 축소한다. 하지만 저차원 벡터로 데이터의 특징을 찾기 어렵거나 관계가 명확하게 드러나지 않는 경우 L의 차원을 X의 차원보다 크게 하는 경우도 있다.

Remark of Auto-encoder

- Encoder와 decoder의 자유롭게 구성할 수 있다. X가 이미지 데이터인 경우 encoder에서 convolution 연산을 하고 decoder에서 deconvolution연산을 할 수 있다. 이미지 데이터가 아닌 경우 encoder와 decoder는 MLP로 구성할 수 있다. 일반적으로 encoder와 decoder의 구성은 대칭으로 한다(경험상 대칭인 경우 학습 진행이 잘 됨).
- Encoder에 의해서 X가 L로 변환되고 다시 L은 decoder에 의해서 H로 변환되어 구분하지만 엄밀히 말하면 이 구분은 명확하지 않다. X의 특징을 찾기 위한 새로운 변수 L은 그냥(just) 항등함수를 모델링하는 하나의 히든레이어다. 쉽게 말해서 아래와 같은 레이어를 구성하고 H를 X와 동일한 값이 나오도록 auto-encoder와 같이 학습을 진행한다고 하자.

$$X \longrightarrow H0 \longrightarrow H1 \longrightarrow H2 \longrightarrow H3 \longrightarrow H4 \longrightarrow H$$

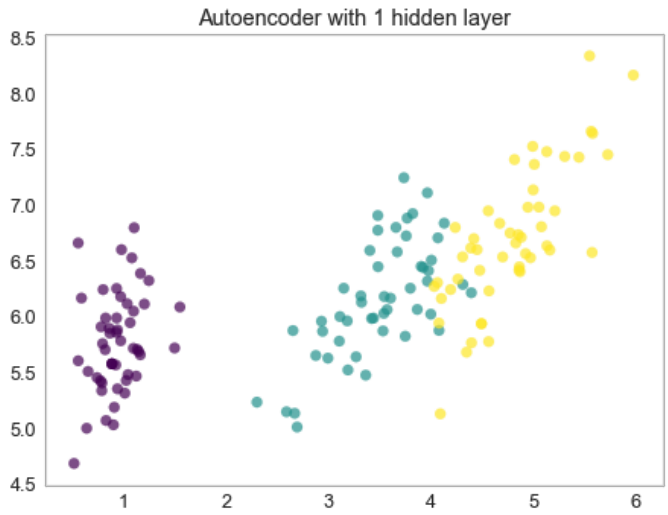
대칭으로 모델을 구성하지 않았다라고 한다면 $H1$이 L이 될 수도 있고 $H2$가 L이 될 수도 있다. 즉, 앞에서 다룬 PCA처럼 L이 수학적으로 의미가 부여되지 않을 수 있다. PCA에서 L은 데이터 X의 분산을 최대로 보존하도록 찾은 직교 기저(basis)로 부터 얻은 성분들이다. - 다음과 같이 encoder와 decoder에 hidden layer가 없는 경우

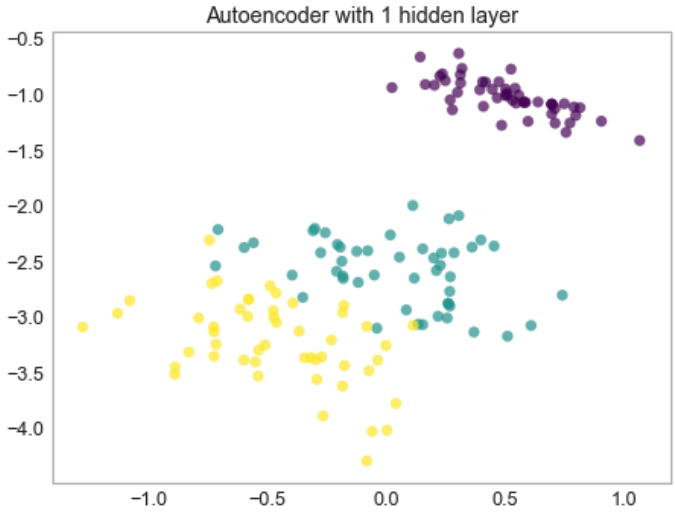
$$X \longrightarrow L \longrightarrow H$$

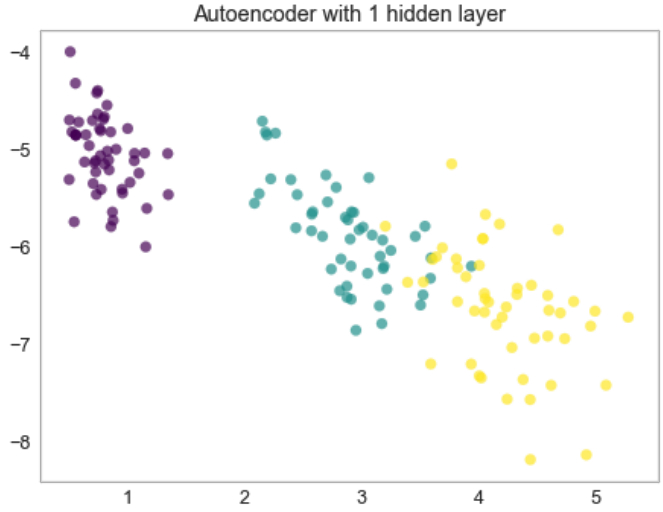
L은 PCA 결과와 유사하게 선형분리의 효과만 얻을 수 있다. 아래 그림은 Iris 데이터 4차원의 X데이터를 PCA와 auto-encoder를 이용하여 2차원으로 차원축소한 결과를 보여준다.

```
from sklearn.datasets import load_iris
iris = load_iris()
x_data = iris.data[:,:]
```

Auto-encoer는 위와 같이 encoder와 decoder에 hidden layer가 없는 형태이고 L의 activation 함수로 항등함수로 주었다. Weight와 bias들의 초기값에 의해서 조금 차이를 보이지만 PCA결과와 유사한 결과를 보여준다.

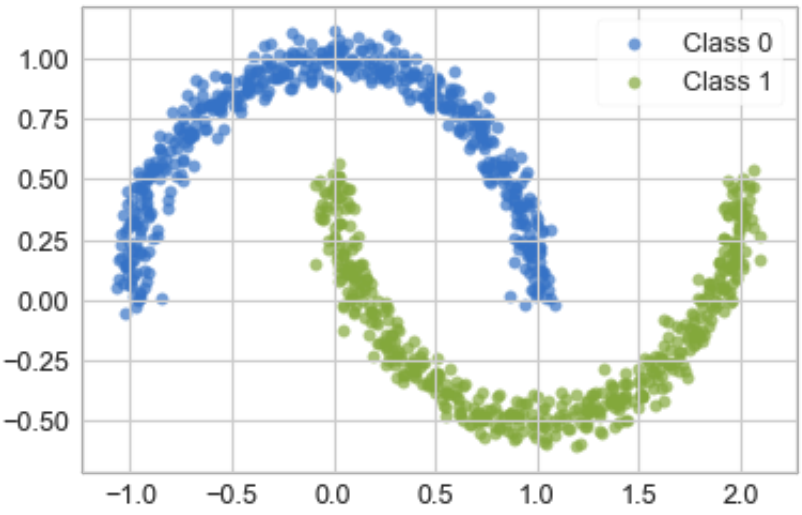




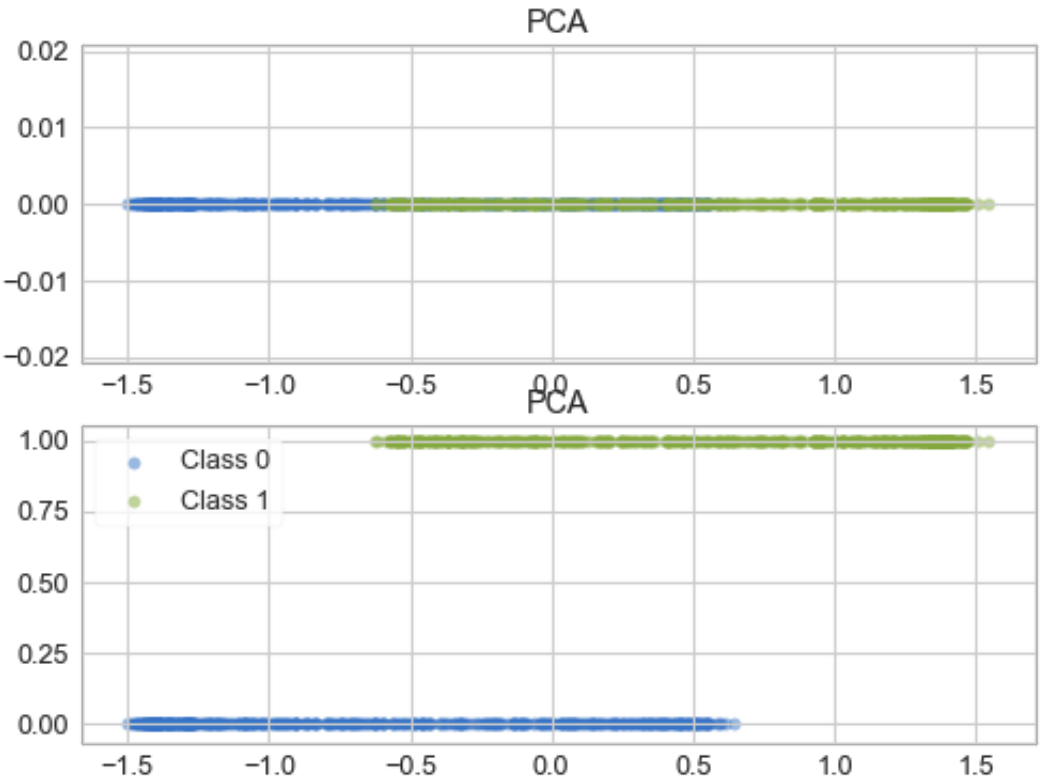


참고: [Auto-association by multilayer perceptrons and singular value decomposition](), [PCA vs Autoencoders]() - 예측값 H가 X와 얼마나 비슷한지 확인해서 학습이 잘 됐는지 여부를 판단한다. 이미지 데이터인 경우 원본이미지와 얼마나 가까운 이미지가 생성됐는지 확인하여 판단하기 때문에 일반적으로 정량적인 지표로 학습 여부를 확인하기 어렵다.

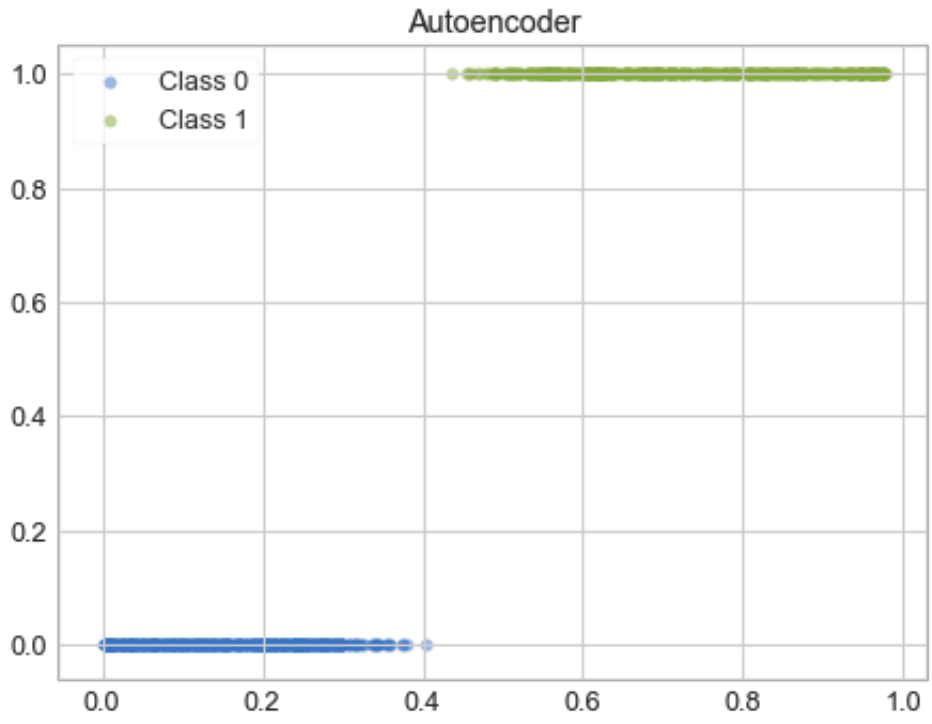
간단한 예를 살펴보자. 아래 X는 2차원 데이터로 산포도를 그리면 다음과 같다. 데이터가 고차원이 아니어서 굳이 PCA를 적용해서 차원축소 할 필요는 없다. 하지만 PCA를 이용해서 특징 추출을 하게 되면 원래 데이터의 특징을 놓칠 수 있다는 것을 보여주는 좋은 예다. 즉, 산포도에서 보는 것 처럼 직선으로 두 집단을 분리하기 어렵다.



PCA를 적용해서 1차원의 데이터로 차원을 축소해서 점을 찍으면 아래 그림의 위 그래프 처럼 표현된다. 파란색 점과 녹색 점이 어느 정도 겹쳐지는지 확인하기 위해서 집단(class)을 기준으로 분리했다. 아래 그림과 같이 선형 분리가 되지 않는 데이터를 PCA를 이용해서 차원축소하게 되면 원래 데이터의 특징을 잃어버리게 된다.



동일한 데이터를 비선형 kernel 방식인 auto-encoder에 적용해서 1차원으로 차원축소하게 되면 다음과 같은 결과를 얻는다. 신기하게 데이터의 라벨값을 지정하지 않았는데도 불구하고 클래스를 고려한 것 처럼 정확하게 데이터가 분리 되었다.



TensorFlow code

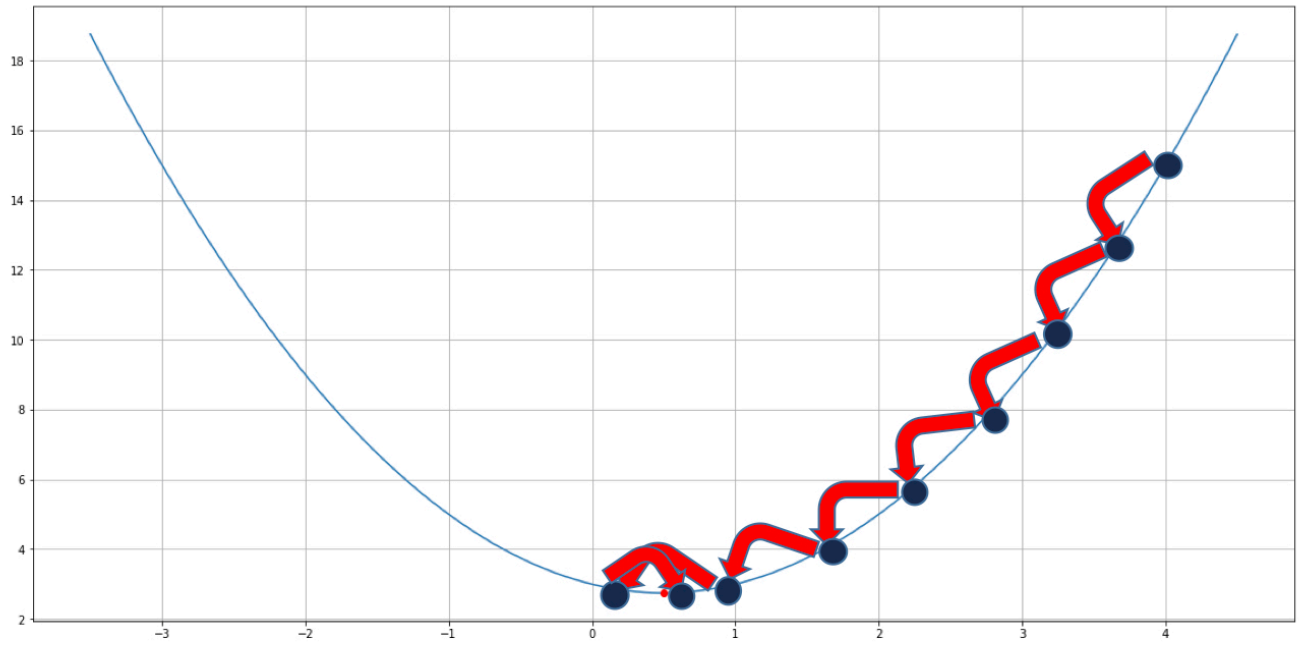
```
import numpy as np
```

```
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.decomposition import PCA
from sklearn.datasets import make_moons

%matplotlib inline

#Moon 데이터 생성
x_moon, y_moon = make_moons(n_samples=1000, shuffle=True, noise=0.05, random_state=1

plt.scatter(x_moon[y_moon == 0, 0], x_moon[y_moon == 0, 1], label="Class 0", alpha=0
plt.scatter(x_moon[y_moon == 1, 0], x_moon[y_moon == 1, 1], label="Class 1", alpha=0
plt.legend()

#PCA를 이용해서 1차원으로 축소
X2 = PCA(1).fit_transform(x_moon)

plt.figure(figsize=(8,6))
plt.subplot(211)
plt.title("PCA")
plt.scatter(X2[y_moon == 0, 0], np.zeros(500), label="Class 0", alpha=0.5)
plt.scatter(X2[y_moon == 1, 0], np.zeros(500), label="Class 1", alpha=0.5)
plt.subplot(212)
plt.title("PCA")
plt.scatter(X2[y_moon == 0, 0], np.zeros(500), label="Class 0", alpha=0.5)
plt.scatter(X2[y_moon == 1, 0], np.ones(500), label="Class 1", alpha=0.5)
plt.legend()

#Auto-encoder 적용

dim = 1
tf.set_random_seed(100)

X = tf.placeholder(shape=[None,2], dtype=tf.float32)


H0 = tf.layers.dense(X, 8, activation=tf.nn.sigmoid)
encoder = tf.layers.dense(H0, dim, activation=tf.nn.sigmoid)
H_1 = tf.layers.dense(encoder, 8, activation=tf.nn.sigmoid)
H = tf.layers.dense(H_1, np.shape(x_moon)[1], activation=tf.identity)


loss = tf.reduce_mean(tf.square(X - H))
optimizer = tf.train.AdamOptimizer(0.01).minimize(loss)

init = tf.global_variables_initializer()
sess = tf.Session()
```

```
sess.run(init)

iteration = 10000

for i in range(iteration):
    _, cost = sess.run([optimizer, loss], feed_dict={X: x_moon})


X2 = PCA(dim).fit_transform(x_moon)
X2_auto = sess.run(encoder, feed_dict={X: x_moon})

plt.figure(figsize=(16,12))
plt.subplot(221)
plt.title("PCA")
plt.scatter(X2[y_moon == 0, 0], np.zeros(500), label="Class 0", alpha=0.5)
plt.scatter(X2[y_moon == 1, 0], np.ones(500), label="Class 1", alpha=0.5)
plt.legend()


plt.subplot(222)
plt.title("Autoencoder")
plt.scatter(X2_auto[y_moon == 0, 0], np.zeros(500), label="Class 0", alpha=0.5)
plt.scatter(X2_auto[y_moon == 1, 0], np.ones(500), label="Class 1", alpha=0.5)

plt.legend()
```

Minjung Gim(mjgim.nims.re.kr), National Institute for Mathematical Sciences

Gradient Descent Optimizer

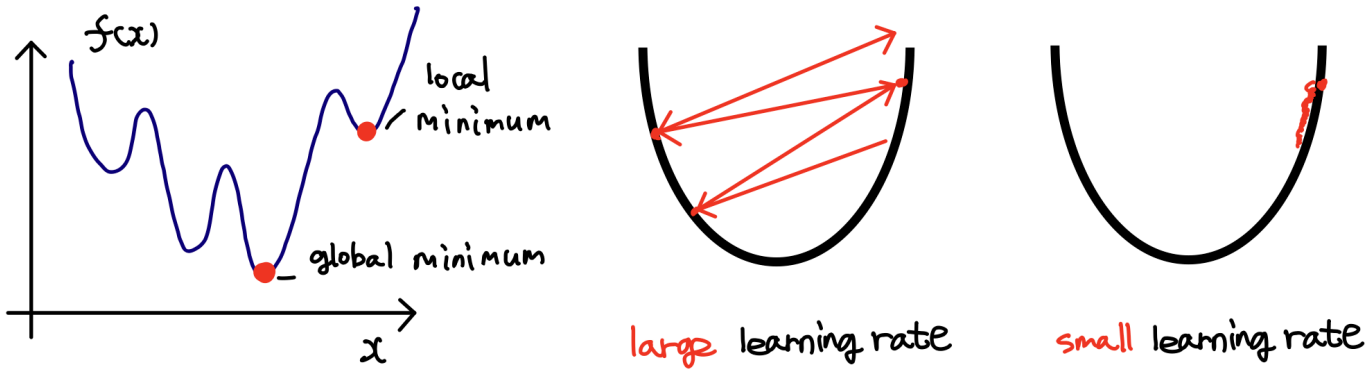


목적함수 f의 최솟값(minimum)을 찾아가는 알고리즘으로 다음과 같은 방식으로 최솟값을 찾아간다.

- 초기값 x_0와 적당한 Learning rate α 설정
- $n \geq 0$인 정수에 대해서 x_{n+1}은 다음과 같이 정의한다.

$$x_{n+1} := x_n - \alpha \cdot \nabla f(x_n)$$

- 주의사항
  - 함수f의 모양이 convex가 아닌 경우 global minimum이 아닌 local minimum으로 x_n이 수렴할 가능성이 있다.
  - Learning rate α 값이 큰 경우 최솟값으로 x_n이 수렴하는 것이 아니라 발산할 수 있다.
  - Learning rate α 값이 작은 경우 수렴하는 속도가 지나치게 느릴 수 있다.



방정식 $2 \cdot x = 10$ 의 근을 Gradient Descent 알고리즘을 이용해서 찾아보자.

- 목적함수 $f(x) := (2x - 10)^2$ 설정
- 초기값 $x_0 = 0$, Learning rate $\alpha = 0.05$으로 설정

$f'(x) = 4(2x - 10)$이므로 x_1 은 다음과 같이 구할 수 있다.

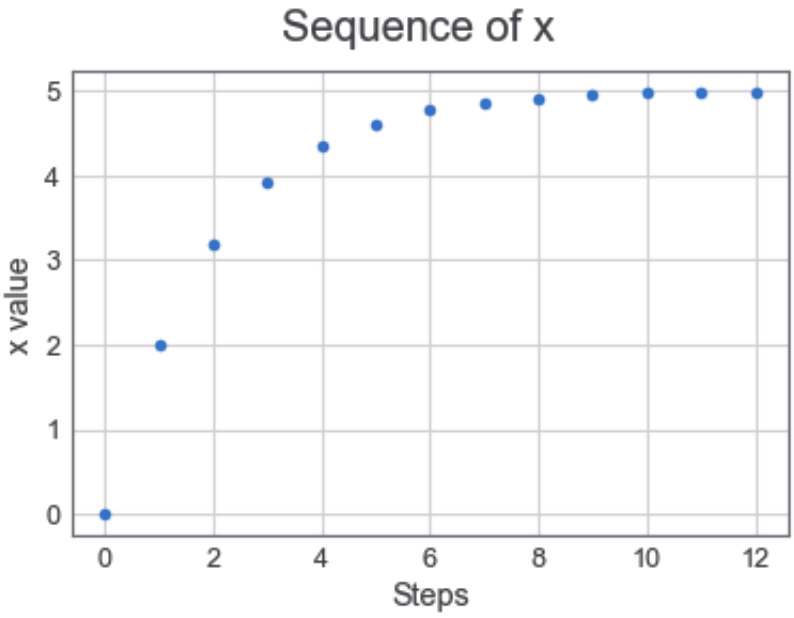
$$x_1 = x_0 - \alpha \cdot f'(x_0) = 0 - 0.05 \cdot (-40) = 2$$

그리고 x_2 는

$$x_2 = x_1 - \alpha \cdot f'(x_1) = 2 - 0.05 \cdot (-24) = 3.2$$

이다. 반복적으로 이 같은 작업을 하면 x_n 의 수열을 다음과 같이 얻을 수 있다.

$$(x_n) = (0, 2, 3.2, 3.92, 4.352, 4.6112, \dots)$$



이를 그래프로 나타내면 다음과 같고 x_n 은 5로 수렴하는 것을 확인 할 수 있다. 이를 `TensorFlow code` 로 작성하면

다음과 같이 작성 할 수 있다.

```
# 방정식 2*x = 10 을 만족하는 x 찾기
# x 초깃값 = 0

import tensorflow as tf
import matplotlib.pyplot as plt

%matplotlib inline
X = tf.Variable(0.)
Y = tf.constant(10.)
H = 2 * X


loss = tf.square(H-Y)
optimize = tf.train.GradientDescentOptimizer(0.05).minimize(loss)


sess = tf.Session()
sess.run(tf.global_variables_initializer())
sequence = []

for i in range(10):
    print("x_%i = %s" %(+i, sess.run(X)))
    sess.run(optimize)
    sequence.append(sess.run(X))

plt.suptitle("Sequence of x", fontsize=20)
plt.ylabel("x value")
plt.xlabel("Steps")
plt.plot(sequence, "o")
```

실행화면

```
x_0 = 0.0
x_1 = 2.0
x_2 = 3.2
x_3 = 3.92
x_4 = 4.352
x_5 = 4.6112
x_6 = 4.76672
x_7 = 4.86003
x_8 = 4.91602
x_9 = 4.94961
```

Learning rate가 0.5인 경우(발산)

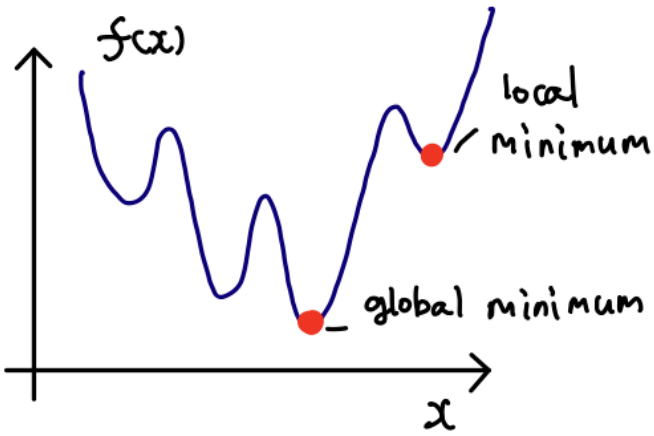
```
x_0 = 0.0
x_1 = 20.0
x_2 = -40.0
x_3 = 140.0
x_4 = -400.0
x_5 = 1220.0
x_6 = -3640.0
x_7 = 10940.0
x_8 = -32800.0
x_9 = 98420.0
```

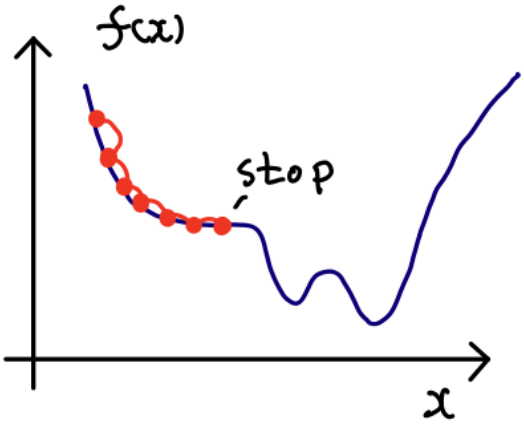
Minjung Gim(mjgim.nims.re.kr), National Institute for Mathematical Sciences

Momentum Optimizer

Gradient Descent Optimizer 보다 개선된 알고리즘으로 이동값에 `관성` 으로 인한 업데이트가 추가된 Optimizer이다. 다음과 같은 Gradient Descent Optimizer의 한계를 개선하는데 효과가 있는 알고리즘이다.

- global minimum이 아닌 local minimum으로 수렴
- 최솟값을 찾아가는 도중에 미분계수가 0인 지점에서 더 이상 이동하지 않음





Gradient Descent Optimize의 이동량은 미분계수에 의해 결정되기 때문에 미분계수가 0이면 더 이상 업데이트가 되지 않는다.

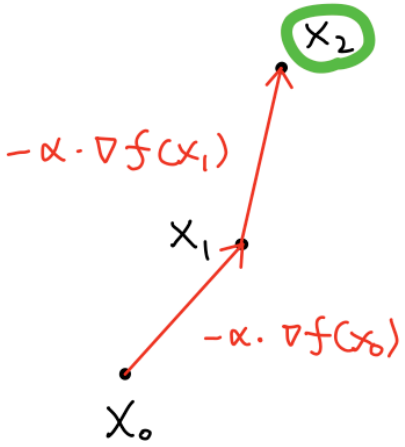
Momentum Optimizer는 다음과 같이 목적함수 f의 최솟값을 찾는다. - 초기값 x_0와 적당한 Learning rate α, momentum β 설정 - $n \geq 0$인 정수에 대해서 a_{n+1} 과 x_{n+1} 은 다음과 같이 정의

$$a_{n+1} := \beta \cdot a_n + \nabla f(x_n), \quad a_0 := 0$$

$$x_{n+1} := x_n - \alpha \cdot a_{n+1}$$

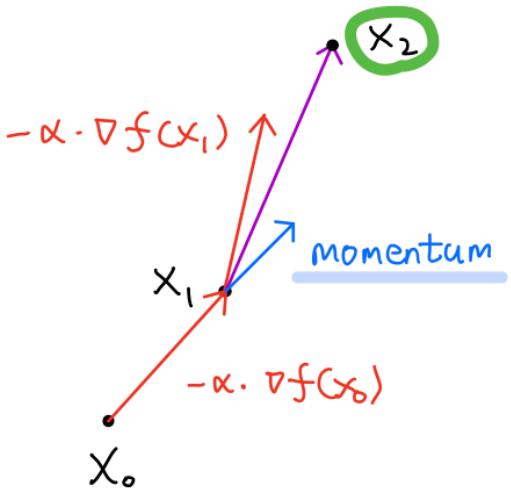
즉, momentum 계수 $\beta = 0$인 경우, Gradient Descent Optimizer와 동일한 알고리즘이다. $\nabla f(x_n) = 0$ 임에도 a_n 에 의한 `관성효과` 로 x_n 은 업데이트된다(다음 그림 참조).

Gradient Descent



$$x_1 = x_0 - \alpha \cdot f(x_0)$$
$$x_2 = x_1 - \alpha \cdot f(x_1)$$

Momentum



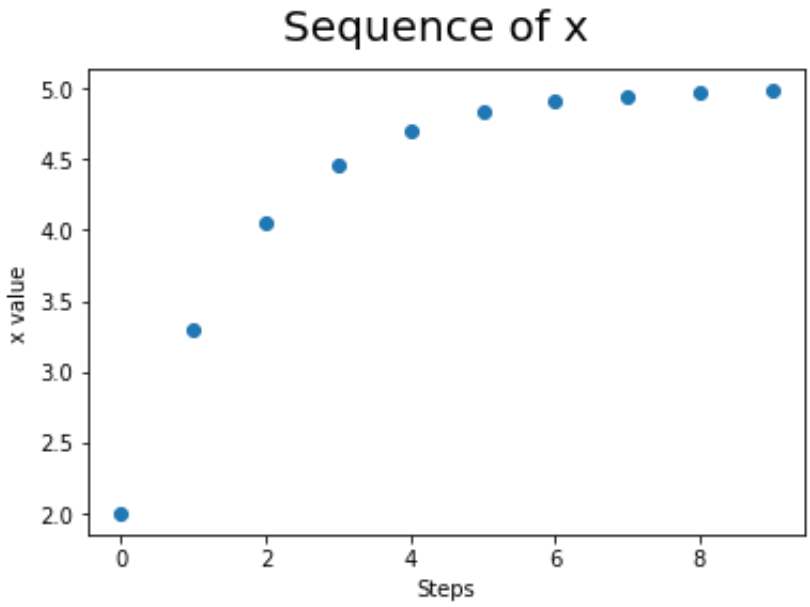
$$x_1 = x_0 - \alpha \cdot f(x_0)$$
$$x_2 = x_1 - \alpha \cdot f(x_1) + \text{momentum}$$

Momentum Optimizer를 이용하여 방정식 $2 \cdot x = 10$ 의 근을 찾는 알고리즘을 `TensorFlow code` 로 작성하면 다음과 같다. Learning rate = 0.05, momentum = 0.03로 설정하였다. 관성효과에 의해서 Gradient Descent 알고리즘보다 빠르게 수렴하는 것을 확인 할 수 있다.

```
# 방정식 2*x = 10 을 만족하는 x 찾기
# x 초깃값 = 0

import tensorflow as tf
import matplotlib.pyplot as plt

%matplotlib inline

X = tf.Variable(0.)
Y = tf.constant(10.)
H = 2 * X


loss = tf.square(H-Y)
optimize = tf.train.MomentumOptimizer(0.05, 0.03).minimize(loss)


sess = tf.Session()
sess.run(tf.global_variables_initializer())
sequence = []

for i in range(10):
    print("x_%i = %s" %(+i, sess.run(X)))
    sess.run(optimize)
    sequence.append(sess.run(X))

plt.suptitle("Sequence of x", fontsize=20)
plt.ylabel("x value")
plt.xlabel("Steps")
plt.plot(sequence, "o")
```

실행화면

```
x_0 = 0.0
x_1 = 2.0
x_2 = 3.26
x_3 = 3.9938
x_4 = 4.41829
x_5 = 4.66371
x_6 = 4.80559
x_7 = 4.88761
x_8 = 4.93503
x_9 = 4.96244
```



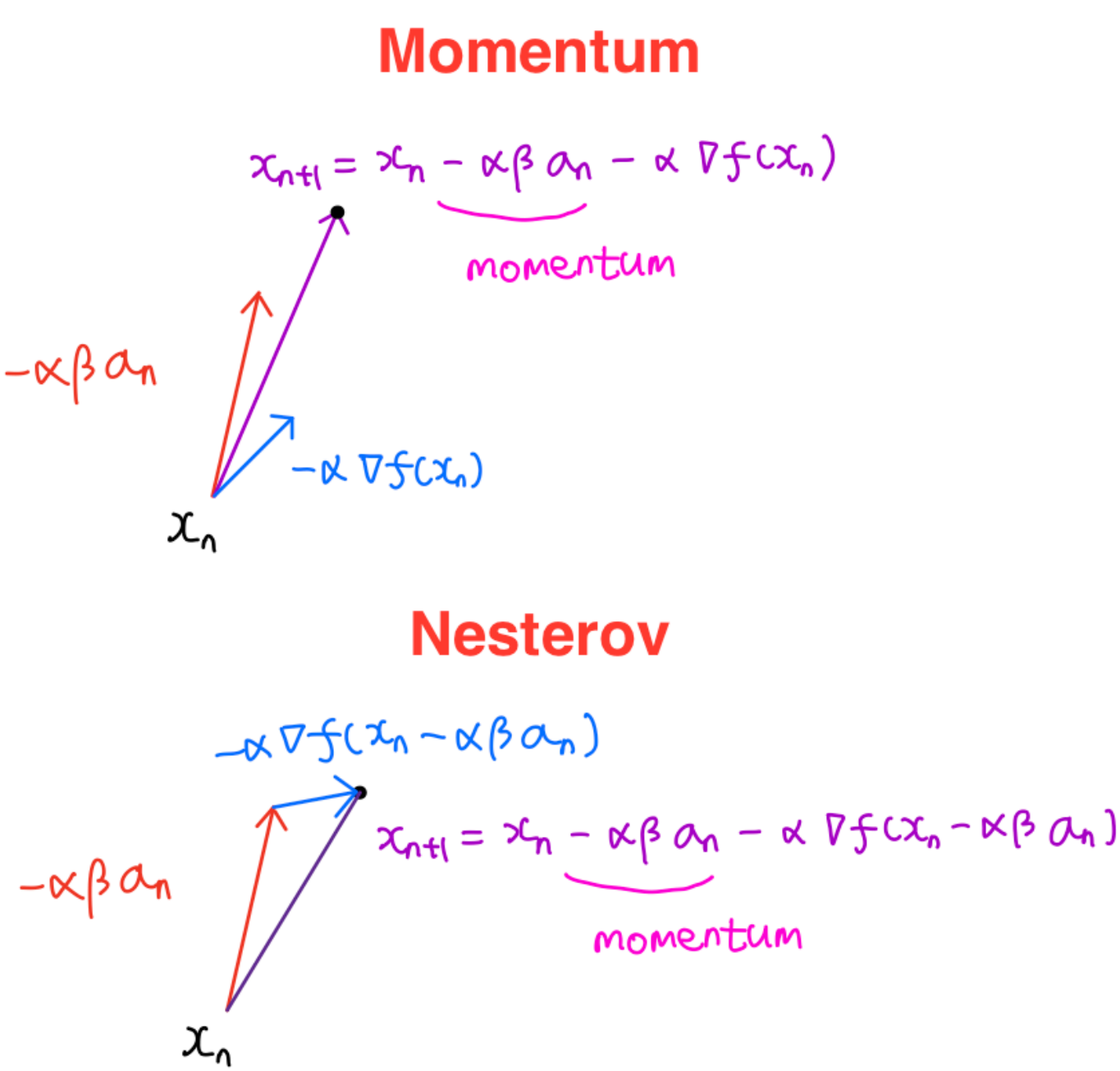
Nesterov Accelerated Gradient(NAG)

`Momentum Optimizer` 의 개선된 알고리즘으로 차이점은 Gradient ∇f 를 x_n 에서 계산하는 것이 아니라 관성(momentum)에 의해서 이동한 $x_n + \beta \cdot a_n$ 에서 계산하는 것이다. **Nesterov Accelerated Gradient** 알고리즘은 다음과 같다.

- 초기값 x_0 와 적당한 Learning rate α, momentum β 설정
- $n \geq 0$인 정수에 대해서 a_{n+1} 과 x_{n+1} 은 다음과 같이 정의한다.

$$a_{n+1} := \beta \cdot a_n + \nabla f(x_n - \alpha \cdot \beta \cdot a_n)$$

$$x_{n+1} := x_n - \alpha \cdot a_{n+1}$$



Momentum Optimizer와 마찬가지로 momentum 계수 $\beta = 0$인 경우, Gradient Descent Optimizer와 동일한 알고리즘이다. NAG의 장점은 학습이 어느 정도 상황에서 momentum optimizer는 관성의 효과로 최적값을 지나칠 수 있는 문제를 방지할 수 있다.

`TensorFlow code`

다음과 같이 Momentum Optimizer 함수의 argument를 설정하면 된다.

```
tf.train.MomentumOptimizer(0.05, 0.03, use_nesterov=True)
```

Minjung Gim(mjgim.nims.re.kr), National Institute for Mathematical Sciences

Adaptive Gradient Optimizer

`Adagrad(Adaptive Gradient) Optimizer` 는 목적함수 f의 최솟값을 찾는 알고리즘으로 기본 철학 및 아이디어는 다음과 같다.

- **변수들이 update하는 방향과 크기를 목적함수 f의 gradient와 이전 단계에서의 변화량을 고려하려 결정한다. 이전 단계에서 변화량이 큰 변수들은 update를 작게하고 변화량이 작은 변수들은 update를 크게 한다. 즉, 변화를 크게 했던 변수들은 최솟값 근처에 도달했을 것이라고 생각하고 update 크기를 작게하고 변화를 작게했던 변수들에 대해서는 최솟값에 도달하기 위해서 더 큰 값으로 update 한다**(~~소득 재분배 같은 느낌~~).

Adagrad Optimizer는 다음과 같이 목적함수 f의 최솟값을 찾는다. - 초기값 x_0와 적당한 Learning rate α, ε 설정 - $n \geq 0$인 정수에 대해서 a_{n+1}과 x_{n+1}은 다음과 같이 정의

$$a_{n+1} := a_n + \nabla f(x_n) \odot \nabla f(x_n), \quad a_0 := 0$$

$$x_{n+1} := x_n - \frac{\alpha}{\sqrt{a_{n+1} + \varepsilon}} \odot \nabla f(x_n)$$

ε은 분모에 0이 되는 것을 방지하는 수로 보통 0.01 정도의 작은 값을 대입한다. 여기서, 벡터 a_{n+1}과 상수 ε의 합은 broad casting해서 연산한 $a_{n+1} + \varepsilon \cdot (1, 1, \ldots, 1)$을 의미하고 $\odot$은 element wise 곱을 의미한다. 예를 들어서 $(1, 2, 3) \odot (1, 2, 3) = (1, 4, 9)$ 이다. 즉, Adagrad Optimizer에서 update 방향은 $(-\nabla f)$ 벡터 방향이 아니라 ($-\nabla f$)와 Adaptive 벡터와의 element wise 곱 방향이다.

Adagrad의 단점은 학습 초기에 변수들이 update하는 양이 learning rate α와 비슷한 값이기 때문에 초기 update 속도가 느리다. 또한 학습을 진행하면서 a_n 들이 계속 증가해 update하는 양이 지나치게 작아져서 제대로 update가 되지 않는 문제가 있다.

방정식 $2 \cdot x = 10$ 의 근을 Adagrad 알고리즘을 이용해서 찾아보자. - 목적함수 $f(x) := (2x - 10)^2$ 정의 - 초기값 $x_0 = 0$, Learning rate $\alpha = 0.05$, $\varepsilon = 0.01$로 설정

$f'(x) = 4(2x - 10)$이므로 a_1과 x_1은 다음과 같이 구할 수 있다.

$$a_1 = a_0 + (f'(x_0))^2 = 0 + 1600 = 1600$$

그리고

$$\begin{aligned} x_1 &= x_0 - \frac{\alpha}{\sqrt{a_1 + \varepsilon}} \cdot f'(x_0) \\ &= 0 - \frac{0.05}{\sqrt{1600 + 0.01}} \cdot (-40) = 0.0499998. \end{aligned}$$

같은 방법으로 a_2 과 x_2 은 다음과 같이 구할 수 있다.

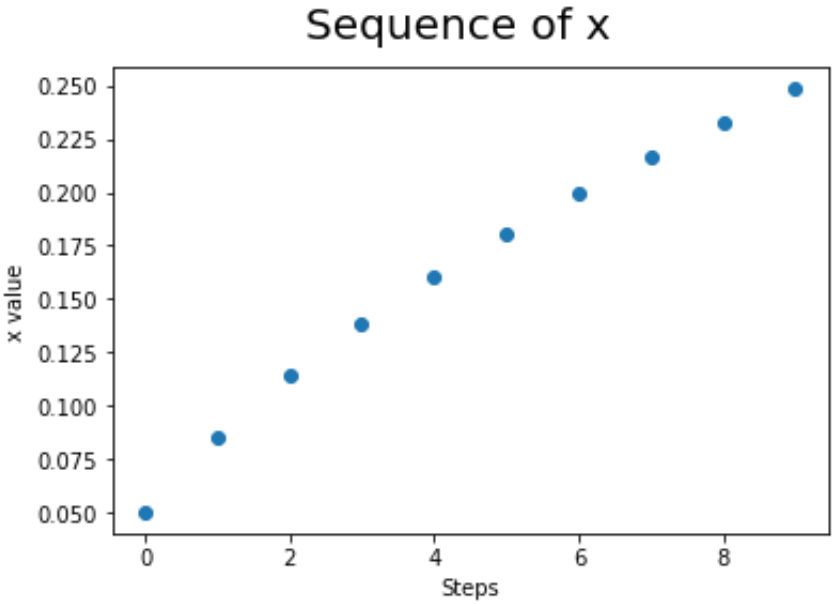
$$a_2 = a_1 + (f'(x_1))^2 = 1600 + 1568.1601267 = 3168.1601267,$$

그리고

$$\begin{aligned} x_2 &= x_1 - \frac{\alpha}{\sqrt{a_2 + \varepsilon}} \cdot f'(x_1) \\ &= 0.0499998 - \frac{0.05}{\sqrt{3168.1601267 + 0.01}} \cdot (-39.6000016) = 0.085177. \end{aligned}$$

반복적으로 이 같은 작업을 수행하면 x_n 의 수열을 다음과 같이 얻을 수 있다.

$$(x_n) = (0, 0.0499998, 0.085177, 0.11381, 0.138548, 0.16063, \dots)$$



x_n 이 5로 수렴해야 하는데 초기에 업데이트 되는 속도가 `Gradient Descent` 나 `Momentum` 알고리즘과 비교해서 수렴 속도가 느린 것을 확인 할 수 있다(초기 변화량은 learning rate와 비슷하다).

`TensorFlow code`

```
# 방정식 2*x = 10 을 만족하는 x 찾기
# x 초깃값 = 0

import tensorflow as tf
import matplotlib.pyplot as plt

%matplotlib inline

X = tf.Variable(0.)
Y = tf.constant(10.)
H = 2 * X

# learning rate = 0.05, epsilon = 0.01
loss = tf.square(H-Y)
optimize = tf.train.AdagradOptimizer(0.05, initial_accumulator_value=0.01).minimize(


sess = tf.Session()
sess.run(tf.global_variables_initializer())
sequence = []

for i in range(10):
    print("x_%i = %s" %(+i, sess.run(X)))
    sess.run(optimize)
    sequence.append(sess.run(X))

plt.suptitle("Sequence of x", fontsize=20)
plt.ylabel("x value")
plt.xlabel("Steps")
plt.plot(sequence, "o")
```

RMSProp Optimizer

`RMSProp Optimizer`는 Adagra Optimizer가 학습을 계속 진행할 수록 update하는 지나치게 작아져서 update가 제대로 이루어지지 않는 문제를 해결하기 위해서 제안된 알고리즘으로 a_{n+1}을 a_n과 $\nabla f(x_n) \odot \nabla f(x_n)$의 합으로 정의하는 것이 아닌 다음과 같이 지수이동평균(Exponential Moving Average), decay rate(γ)으로 정의한다.

$$a_{n+1} := \gamma \cdot a_n + (1-\gamma)\nabla f(x_n) \odot \nabla f(x_n), \quad a_0 := 0$$

$$x_{n+1} := x_n - \frac{\alpha}{\sqrt{a_{n+1} + \varepsilon}} \odot \nabla f(x_n)$$

Decay rate γ는 보통 0.9나 0.99로 설정한다.

방정식 $2 \cdot x = 10$ 의 근을 RMSProp 알고리즘으로 찾아보자. - 초기값 $x_0 = 0$, Learning rate $\alpha = 0.05$, decay rate $\gamma = 0.9$, $\varepsilon = 0.01$로 설정

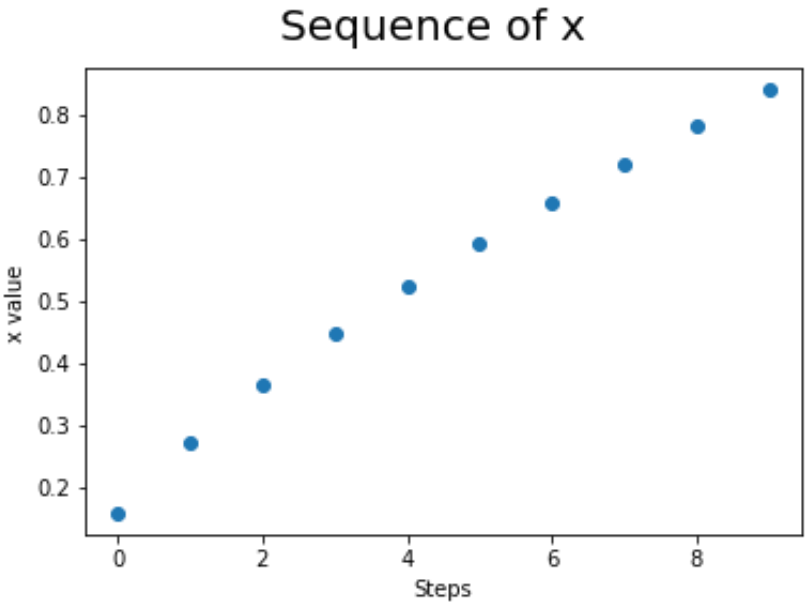
$$a_1 = \gamma \cdot a_0 + (1-\gamma) \cdot (f'(x_0))^2 = 0.9 \cdot 0 + 0.1 \cdot 1600 = 160,$$

$$x_1 = x_0 - \frac{\alpha}{\sqrt{a_1 + \varepsilon}} \cdot f'(x_0) = 0 - \frac{0.05}{\sqrt{160 + 0.01}} \cdot (-40) = 0.1580645.$$

같은 방법으로 a_2 와 x_2 는 다음과 같이 구할 수 있다.

$$a_2 = \gamma \cdot a_1 + (1-\gamma) \cdot (f'(x_1))^2 = 0.9 \cdot 160 + 0.1 \cdot 1500.4377207 = 294.0437721,$$

$$x_2 = x_1 - \frac{\alpha}{\sqrt{a_2 + \varepsilon}} \cdot f'(x_1) = 0.1580645 - \frac{0.05}{\sqrt{294.0437721 + 0.01}} \cdot (-38.735484) = 0.2710091.$$



Adagrad 알고리즘 보다 초반에 빠르게 최적값으로 수렴하는 것을 확인 할 수 있다.

`TensorFlow code`

x_n의 계산값과 TensorFlow의 출력값과 약간의 오차가 존재한다.

```
# 방정식 2*x = 10 을 만족하는 x 찾기
# x 초깃값 = 0

import tensorflow as tf
import matplotlib.pyplot as plt

%matplotlib inline

X = tf.Variable(0.)
Y = tf.constant(10.)
H = 2 * X

# learning rate = 0.05, decay = 0.9, epsilon = 0.01
loss = tf.square(H-Y)
optimize = tf.train.RMSPropOptimizer(0.05, decay=0.9, epsilon=0.01).minimize(loss)


sess = tf.Session()
sess.run(tf.global_variables_initializer())
sequence = []

for i in range(10):
    print("x_%i = %s" %(+i, sess.run(X)))
    sess.run(optimize)
    sequence.append(sess.run(X))

plt.suptitle("Sequence of x", fontsize=20)
plt.ylabel("x value")
plt.xlabel("Steps")
plt.plot(sequence, "o")
```

Minjung Gim(mjgim.nims.re.kr), National Institute for Mathematical Sciences

Adam Optimizer

Adam Optimizer(Adaptive Moment estimation) (ref. 2015 ICL "Adam: A Method for Stochastic Optimization)는 목적함수 f의 최솟값을 찾는 알고리즘으로 Momentum과 RMSProp의 기본 아이디어를 합친 알고리즘이다. Momentum term에 RMSProp에서 등장했던 지수이동평균을 적용하고 RMSProp과 같은 방식으로 변수들이 update하는 방향과 크기는 Adaptive 벡터에 의해서 결정된다(소득 재분배). 알고리즘의 형태는 조금 복잡해보이지만 두 optimizer를 적절하게 조화시킨 형태다(~~optimizer의 완결판~~).

Adam Optimizer는 다음과 같이 목적함수 f의 최솟값을 찾는다. - 초기값 x_0와 적당한 Learning rate $\alpha, \beta_1, \beta_2, \varepsilon$ 설정 - β_1 : momentum decay rate `(default : 0.9)` - β_2 : adaptive term decay rate `(default : 0.999)`

- $n \geq 0$인 정수에 대해서 momentun term a_{n+1} 과 adaptive term b_{n+1} 을다음과 같이 정의한다(⊙는 element wise 곱, 참고).

$$a_{n+1} := \beta_1 \cdot a_n + (1-\beta_1) \cdot \nabla f(x_n), \quad a_0 := 0$$

$$b_{n+1} := \beta_2 \cdot b_n + (1-\beta_2) \cdot \nabla f(x_n) \odot \nabla f(x_n), \quad b_0 := 0$$

여기서 a_{n+1} 을 정의할 때 기존의 Momentum 알고리즘과 달리 지수이동평균을 사용하기 때문에 ∇f의 계수$(1-\beta_1)$가 작은 값을 갖는다. 그래서 Adam Optimizer는 초기 update 속도를 보정(올리기)하기 위해서 상수 $\widehat{\alpha}_{n+1}, n \geq 0$를 다음과 같이 정의하여 변수를 update하는데 사용한다.

$$\widehat{\alpha}_{n+1} := \alpha \cdot \frac{\sqrt{1-(\beta_2)^{n+1}}}{1-(\beta_1)^{n+1}}$$

$n >> 1$인 경우 $\widehat{\alpha}_n$ 은 α로 수렴한다. 마지막으로 변수 x_{n+1}는 다음과 같이 정의한다.

$$x_{n+1} := x_n - \frac{\widehat{\alpha}_{n+1}}{\sqrt{b_{n+1}} + \varepsilon} \odot a_{n+1}$$

ε은 분모에 0이 되는 것을 방지하는 수로 보통 (1e-8) 정도의 작은 값을 대입한다. **Adam Optimizer**의 정의로 부터 Momentum Optimizer와 RMSProp Optimizer의 장점들을 모두 기대할 수 있다.

방정식 $2 \cdot x = 10$ 의 근을 Adam Optimizer 알고리즘을 이용해서 찾아보자.

- 목적함수 $f(x) := (2x-10)^2$ 정의
- 초기값 $x_0 = 0$, Learning rate $\alpha = 0.5, \beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 0.01$로 설정

$f'(x) = 4(2x - 10)$이므로

$$a_1 = \beta_1 \cdot a_0 + (1 - \beta_1) \cdot f'(x_0) = -4,$$

$$b_1 = \beta_2 \cdot b_0 + (1 - \beta_2) \cdot (f'(x_0))^2 = 1.6,$$

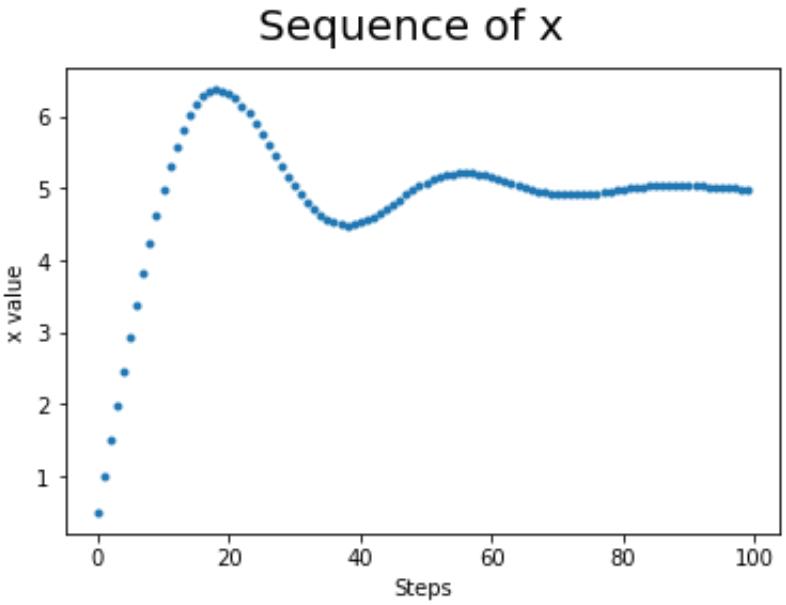
$$\widehat{\alpha}_1 = \alpha \cdot \frac{\sqrt{1 - \beta_2}}{1 - \beta_1} = 0.0158113$$

이고 x_1 은 다음과 같이 얻을 수 있다.

$$x_1 = x_0 - \frac{\widehat{\alpha}_1}{\sqrt{b_1} + \varepsilon} \cdot a_1 = 0.5.$$

같은 방법으로 작업을 반복적으로 수행하면 x_n 의 수열을 다음과 같이 얻을 수 있다.

$$(x_n) = (0, 0.5, 0.997937, 1.49206, 1.9803, 2.46017, \dots)$$



x_n 이 5로 수렴하는 것을 확인 할 수 있고 초기에 업데이트 되는 속도가 `Gradient Descent` 에 비해 느리지만 주목할 부분이 **`Gradient Descent` 에서 learning rate=0.5 인 경우 발산했지만 Adam Optimizer는 발산하지 않고 수렴한다**.

부연 설명 : 위 그림에서 보면 시행 횟수가 16회 정도 되었을 때 x_n 이 우리가 원하는 정답값 5를 지나친다. 만약에 Gradient Descent Optimizer를 사용했다면 x_n 값이 5보다 작다가 5를 넘어가게 되면 미분계수의 부호가 음수에서 양수로 변경이 되어서 바로 돌아와야 하는데 관성(Momentum) 효과, 즉 이동하고 있던 이동량을 기억하고 있기 때문에 돌아오지 않고 정답 5에서 멀어지고 있는 것이다. 이 같은 현상이 35번 째에서도 발생한다.

`TensorFlow code`

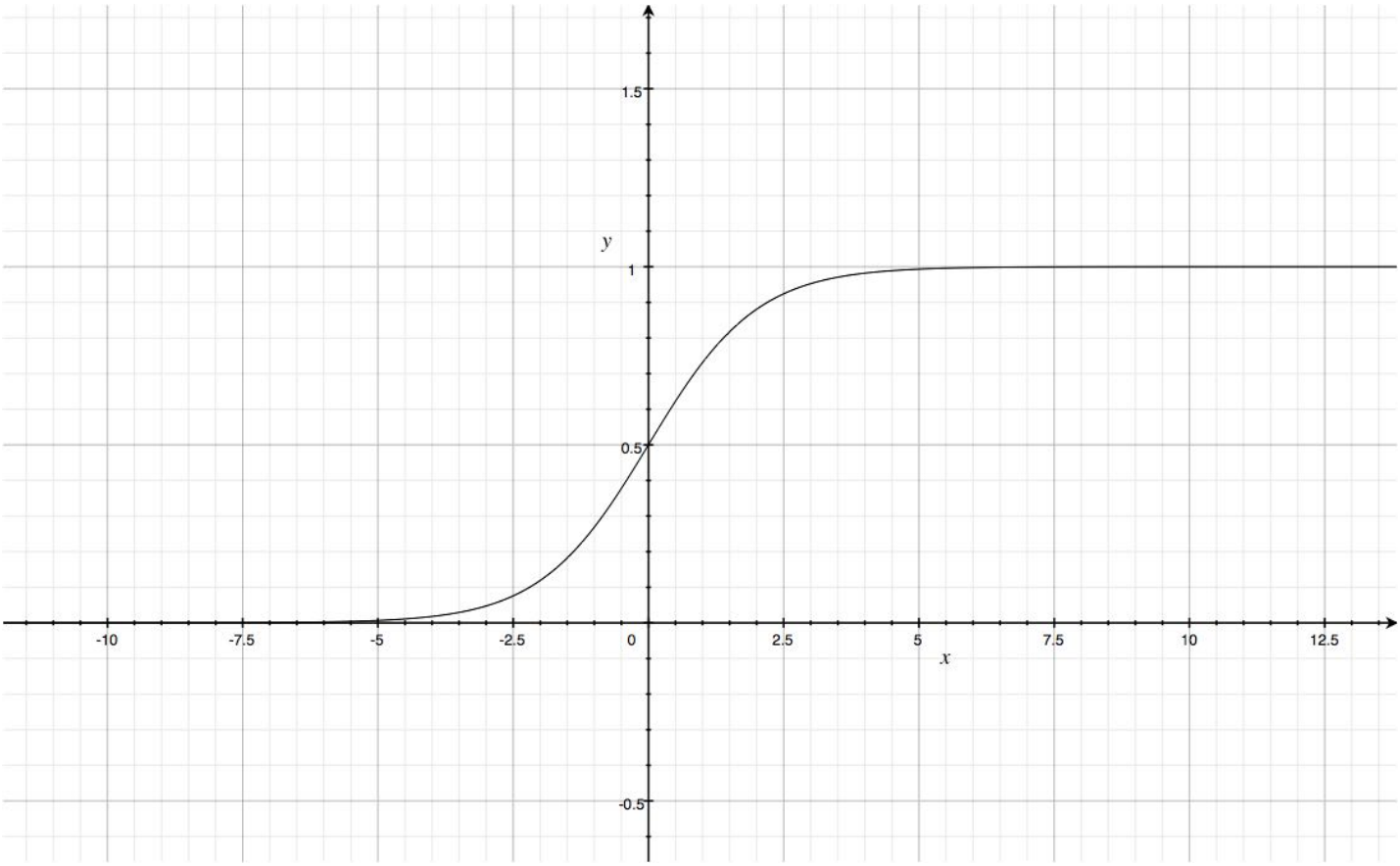
β_1, β_2, ε을 따로 지정하지 않으면 default값으로 사용

```
# 방정식 2*x = 10 을 만족하는 x 찾기
# x 초깃값 = 0

import tensorflow as tf
import matplotlib.pyplot as plt

%matplotlib inline


X = tf.Variable(0.)
Y = tf.constant(10.)
H = 2 * X

# learning rate = 0.5
loss = tf.square(H-Y)
optimize = tf.train.AdamOptimizer(0.5).minimize(loss)


sess = tf.Session()
sess.run(tf.global_variables_initializer())
sequence = []

for i in range(100):
    #print("x_%i = %s" %(+i, sess.run(X)))
    sess.run(optimize)
    sequence.append(sess.run(X))

plt.suptitle("Sequence of x", fontsize=20)
plt.ylabel("x value")
plt.xlabel("Steps")
plt.plot(sequence, ".")
```

Minjung Gim(mjgim.nims.re.kr), National Institute for Mathematical Sciences

활성함수(Activation) 시그모이드(Sigmoid)함수 정의

로지스틱 회귀분석 또는 Neural network의 Binary classification(이진 분류) 마지막 레이어의 활성함수로 사용하는 시그모이드 $s(z) = \frac{1}{1+e^{-z}}$ 에 대해 살펴보겠다. 다음 그림은 시그모이드 함수의 그래프다.



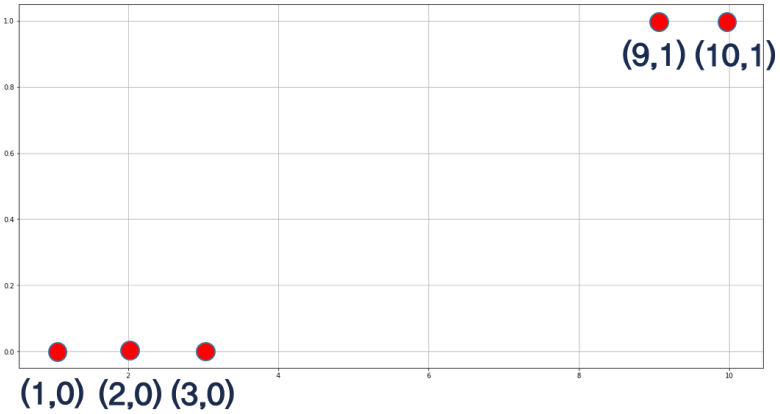
데이터를 두 개의 그룹으로 분류하는 문제에서 가장 기본적인 방법은 로지스틱 회귀분석이다. 회귀분석과의 차이는 회귀분석에서는 우리가 원하는 것이 예측값(실수)이기 때문에 종속변수의 범위가 실수이지만 로지스틱 회귀분석에서는 종속변수 y값이 0 또는 1을 갖는다. 그래서 우리는 주어진 데이터를 분류할 때 0인지 1인지 예측하는 모델을 만들어야 한다.

`여기서 0을 실패, 1을 성공 이라고 하겠다.`

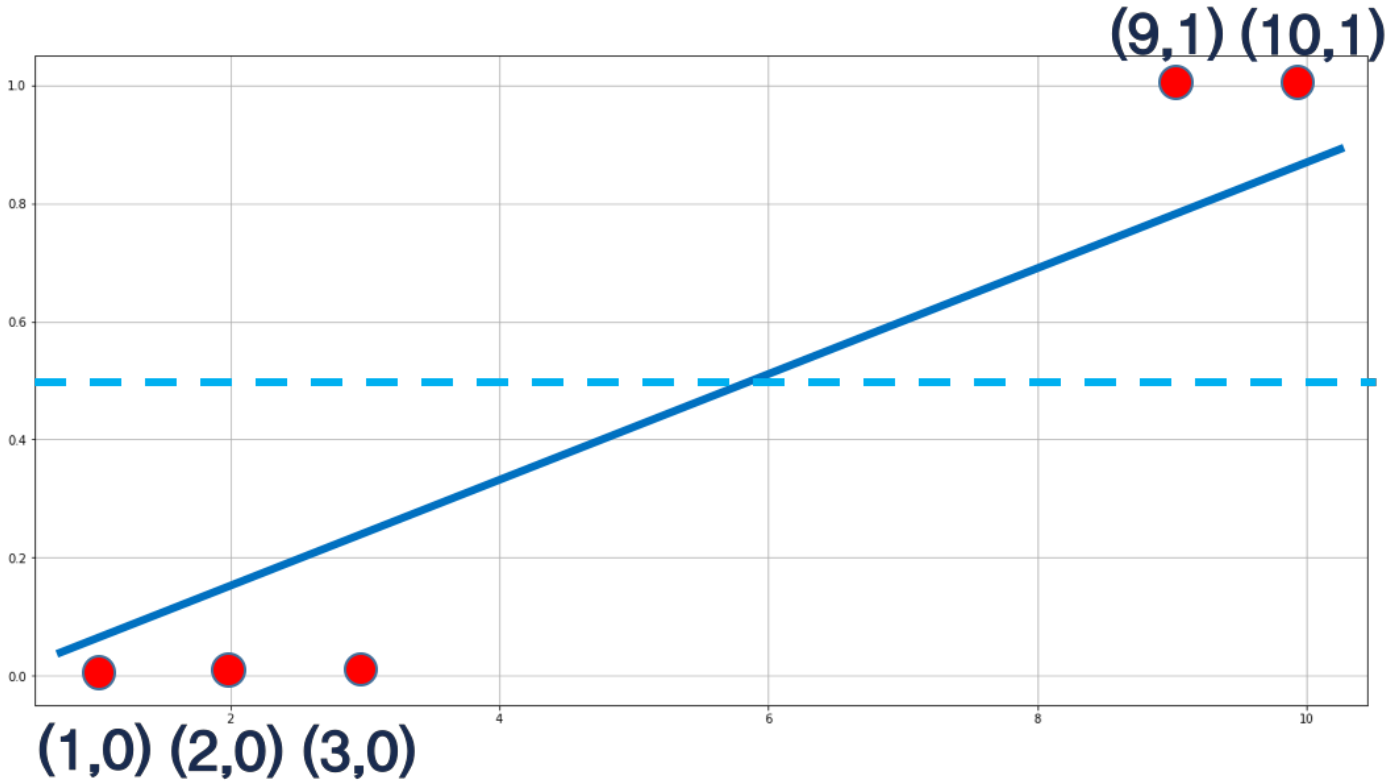
로지스틱 회귀분석에서 데이터를 두 개의 그룹으로 분리하는데 선형함수(직선)을 사용하면 안되는 이유를 먼저 살펴보겠다. 예를 들어서 다음과 같이 데이터가 주어졌다고 가정해보자.

로지스틱회귀분석

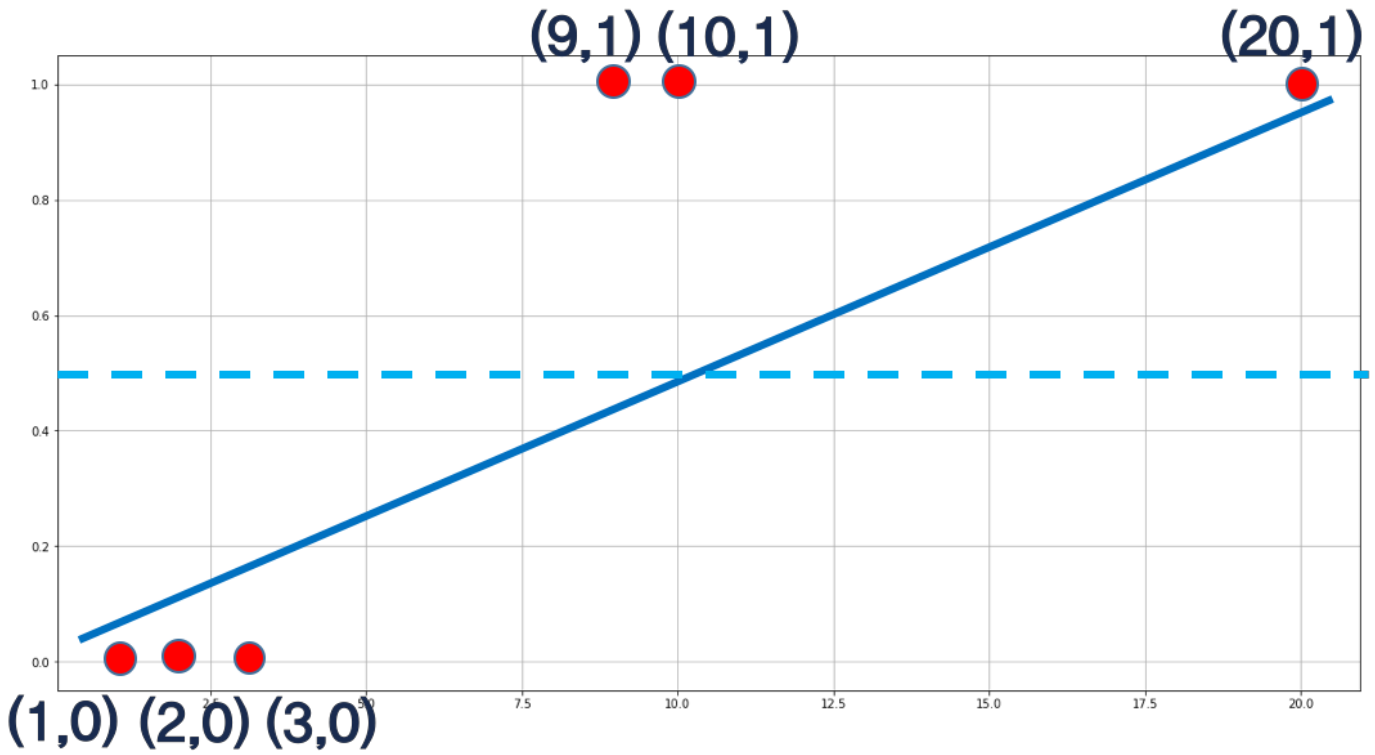
X 데이터	Y 데이터
10	Pass (1)
9	Pass (1)
3	Fail (0)
2	Fail (0)
1	Fail (0)
5	???



먼저 데이터를 다음과 같이 선형함수로 분류를 했다고 하자.

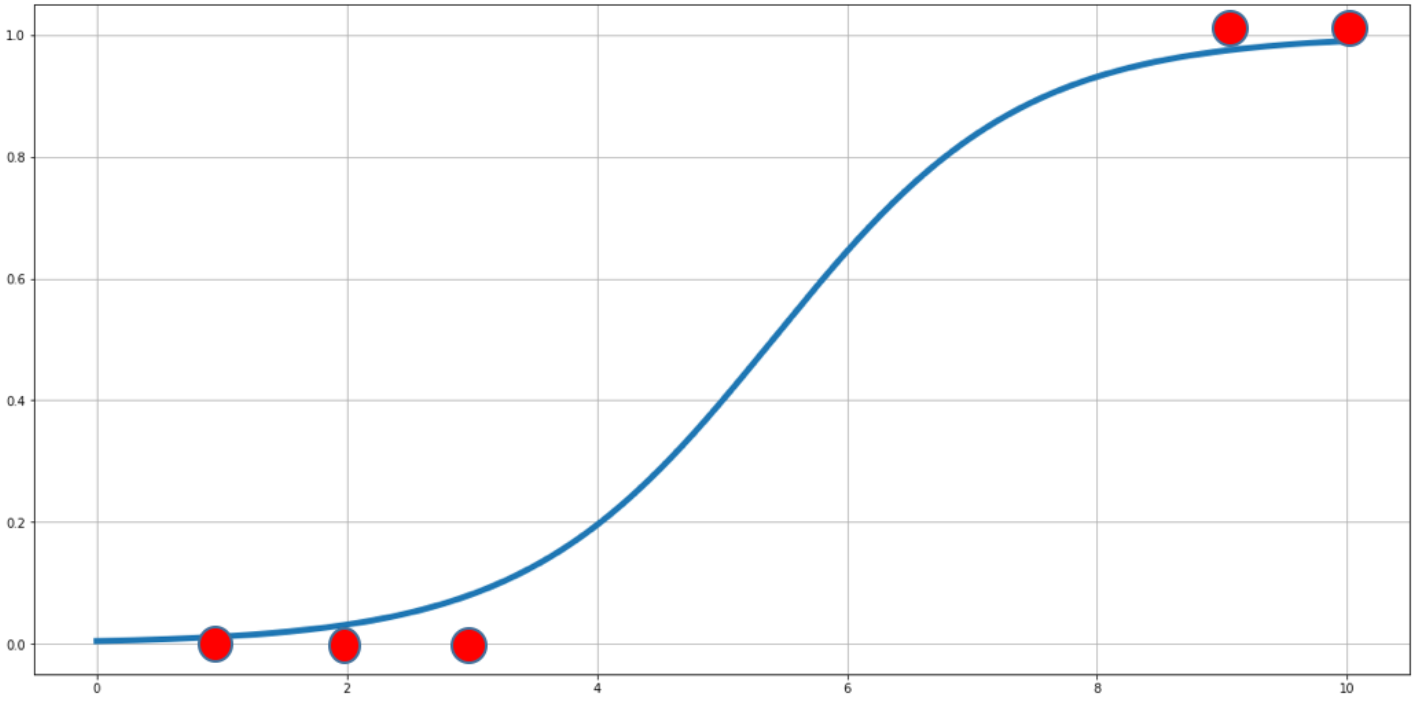


즉, 함수값이 $\frac{1}{2}$이 나오는 x를 기준으로 성공과 실패가 구분된다고 보면 된다. 우선 현재의 데이터와 결과를 봤을 때 잘 분류한 것 처럼 보인다. 하지만 $x = 20$이라는 새로운 데이터가 성공인지 실패인지 알아보기 위해 선형함수에 적용해보면 함숫값은 1보다 큰 값이 되고 이 함숫값에 적절한 의미를 부여하기 어렵다. 또 다른 이유로, 학습데이터에 새로운 값(20,1)이 추가된다면($x = 20$을 갖고 성공한 경우) 두 그룹(성공과 실패)으로 분류하는 직선은 다음과 같이 변경될 것이다.



다른 x값에 비해 큰 $x = 20$으로 인해서 선형함수의 기울기가 더 작아지고 새로운 데이터의 추가로 인해서 기존에 잘 분류되었던 (9,1)과 (10,1)을 분류하는데 실패하게 된다. 즉, 새로운 데이터의 추가가 기존의 분류 모델에 큰 영향을 미치게 된다.

그래서 로지스틱 회귀분석에서는 다음과 같은 형태의 함수를 활성함수로 사용하여 데이터를 성공과 실패로 분류한다.



함수의 특징을 살펴보면 다음과 같다.

- 성공과 실패를 구분하는 부분은 경사가 급하고 나머지 부분에서는 경사가 완만하다.

- $y = 1, y = 0$ 두 평행선이 점근선이고 치역은 (0,1)이다. 즉 위와 같은 활성함수의 함숫값은 성공확률이라는 의미로 해석할 수 있다(1을 실패라고 정의했다면 실패 확률로 해석하면 된다).

이러한 특징을 만족하는 함수로 시그모이드 $s(z) = \frac{1}{1+e^{-z}}$ 함수를 활성함수로 사용한다. 여기서 활성함수를 시그모이드를 사용하는 것이 맞는지에 대한 질문에 대답을 해야한다. 혹자는 시그모이드가 계단함수(Step function)의 미분가능한 형태이기 때문이라고 하고 또는 작은 자극에는 감각을 거의 느끼지 못하다가 어떤 임계값을 넘어가야 감각을 느끼는 우리의 신경망 세포와 비슷하기 때문이라고 말하지만 이것은 수학적이지 못하다.

그래서 왜 시그모이드를 사용할까?

단순선형회귀분석(독립변수의 개수가 1개)은 앞에서 언급한 것처럼 목표가 실수값 예측이기 때문에 선형함수 $y = wx + b$를 이용하여 예측한다(예측 변수의 수가 하나인 경우). 하지만 로지스틱 회귀분석에서는 종속변수가 0 또는 1이기 때문에 $y = wx + b$을 이용해서 예측하는 것은 의미가 없다고 앞에서 살펴보았다. 그래서 $Odds$를 이용하는데 $Odds$는 다음과 같이 정의 된다. 확률 p가 주어져 있을 때

$$Odds(p) := \frac{p}{1-p}$$

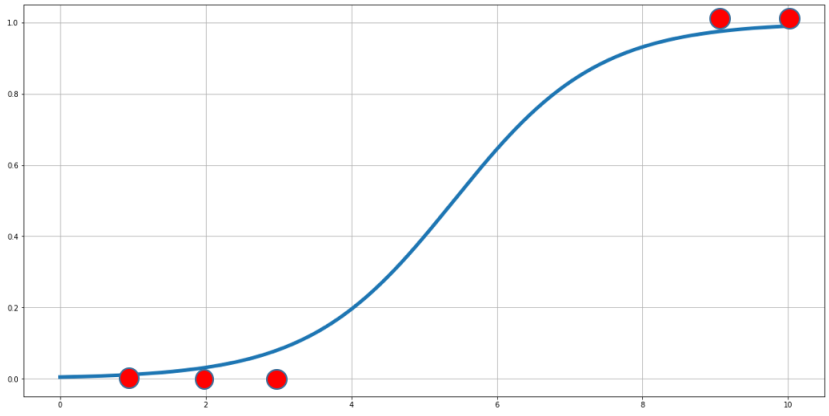
로 정의한다. 확률 p의 범위가 (0,1)이라면 $Odds(p)$의 범위는 $(0,\infty)$이 된다. $Odds$에 로그함수를 취한 $\log(Odds(p))$은 범위가 $(-\infty,\infty)$이 된다. 즉, 범위가 실수 전체이다. $\log(Odds(p))$의 범위가 실수이므로 이 값에 대한 선형회귀분석을 하는 것은 의미가 있다.

$$\log(Odds(p)) = wx + b$$

으로 선형회귀분석을 실시해서 w와 b를 얻을 수 있다. 위 식을 p로 정리하면 다음과 같은 식을 얻을 수 있는데 이 식이 시그모이드이다.

$$p(x) = \frac{1}{1 + e^{-(wx+b)}}$$

x 데이터가 주어졌을 때 성공확률을 예측하는 로지스틱 회귀분석은 학습데이터를 잘 설명하는 시그모이드 함수의 w와 b를 찾는 문제다.

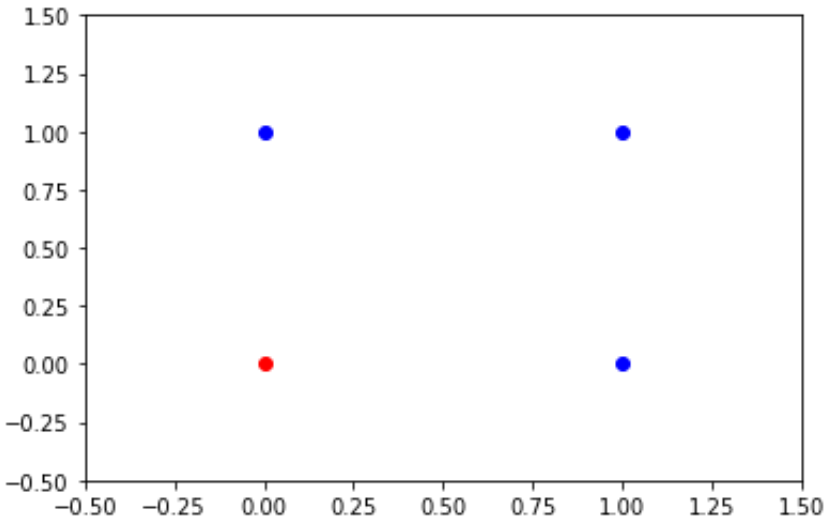


- $y = H(x) = \frac{1}{1+e^{-(wx+b)}}$
- $w = 1.0072499$
- $b = -5.4476051$

Minjung Gim(mjgim.nims.re.kr), National Institute for Mathematical Sciences

Binary classification with Cross entropy

종속변수가 `categorial variable` 인 **분류(classificaion)** 문제를 살펴보자. 주어진 데이터를 두 개의 그룹(0 또는 1)으로 분류하는 이진 분류(binary classification)문제를 Hidden layer가 없는 가장 단순한 형태인 로지스틱회귀분석(logistic regression)으로 해결할 것이다. 회귀분석과 달리 분류문제는 최종 레이어 H를 정의할 때 activation 함수로 `Sigmoid` 를 사용한다. 간단한 예로 다음과 같은 **"OR"** 문제를 살펴보자. X는 2차원 데이터로 두 변수 중에서 1이 하나라도 있으면 Y값은 1이고 그 외의 경우는 0이다.

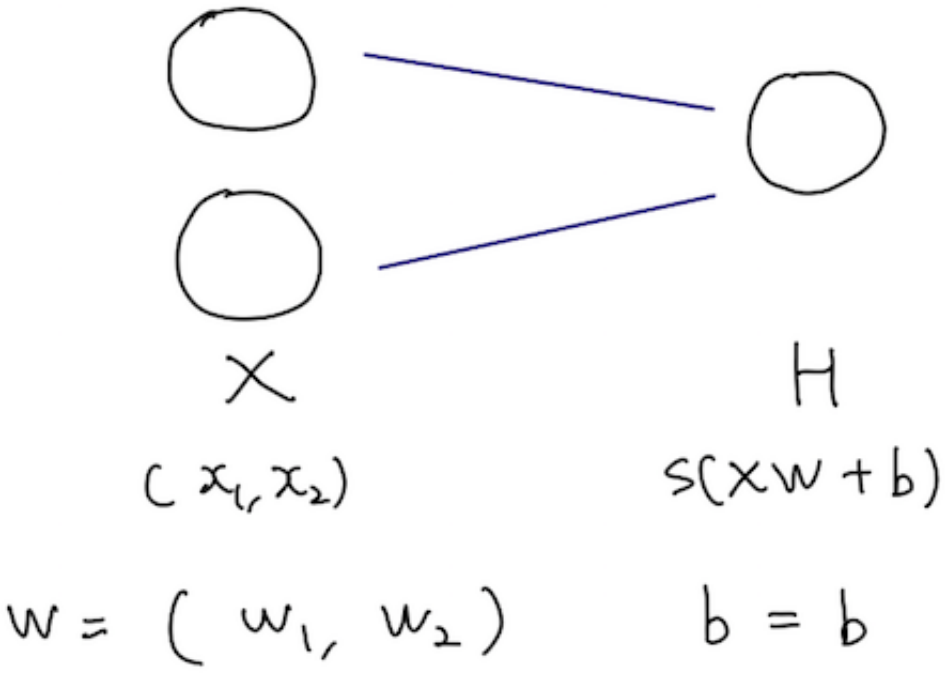


X	Y
(0,0)	0
(0,1)	1
(1,0)	1
(1,1)	1

로지스틱회귀분석이기 때문에 목표는 주어진 데이터 X와 Y를 잘 설명하는

$$H(X) = s(XW + b) = s(w_1 \cdot x_1 + w_2 \cdot x_2 + b)$$

의 W와 b를 찾는 것이다. 여기서 $s(z) := \frac{1}{1+e^{-z}}$ 이고 $X = (x_1, x_2)$, $W = (w_1, w_2)$이다.



분류(classificaion)의 cost 함수

cost 함수란 실제값 Y와 모델이 예측한 값 H의 차이를 측정(measure)하는 함수로 예측값 H가 실제값 Y를 잘 예측하기 위해 cost함수를 최소화하는 방향으로 W와 b를 업데이트해야 한다. 회귀분석(regression)에서는 cost 함수를 MSE(Mean square error)로 정의하였는데 분류문제에서 cost 함수를 MSE로 정의하면 지나치게 작은 Sigmoid 함수의 미분계수로 인해서 weight와 bias가 제대로 업데이트 되지 않는다. 이를 확인하기 위해 MSE로 정의한 cost 함수와 w_1 에 대한 편미분을 확인해보자.

$$
\begin{aligned}
MSE(W,b) &= \frac{1}{4}\sum_{i=1}^{4}\Big(H(X_i)-y_i\Big)^2 = \frac{1}{4}\sum_{i=1}^{4}\Big(s(w_1\cdot x_{i1}+w_2\cdot x_{i2}+b)-y_i\Big)^2 \\
&= \frac{1}{4}\Big\{\Big(s(b)\Big)^2+\Big(s(w_2+b)-1\Big)^2+\Big(s(w_1+b)-1\Big)^2+\Big(s(w_1+w_2+b)-1\Big)^2\Big\}
\end{aligned}
$$

이고

$$
\begin{aligned}
\frac{\partial MSE(W,b)}{\partial w_1} &= \frac{1}{4}\Big\{2\Big(s(w_1+b)-1\Big)\cdot\frac{\partial s(w_1+b)}{\partial w_1}+2\Big(s(w_1+w_2+b)-1\Big)\cdot\frac{\partial s(w_1+w_2+b)}{\partial w_1}\Big\}. \\
&= \frac{1}{4}\Big\{2\Big(s(w_1+b)-1\Big)\cdot s(w_1+b)\cdot\Big(1-s(w_1+b)\Big) \\
&\quad +2\Big(s(w_1+w_2+b)-1\Big)\cdot s(w_1+w_2+b)\cdot\Big(1-s(w_1+w_2+b)\Big)\Big\}.
\end{aligned}
$$

Weight와 bias의 초깃값과 Sigmoid함수의 성질에 의해 편미분계수가 지나치게 작은 값이 나오기 때문에 weight와 bias가 제대로 업데이트가 되지 않는다. 그래서 이진 분류문제의 cost 함수는 MSE가 아닌 Cross Entropy로 정의한다.

$$cost(W,b) = -\frac{1}{4}\sum_{i=1}^{4}\Big(y_i \cdot \log(H(X_i)) + (1-y_i)\cdot\log(1-H(X_i))\Big)$$
$$= -\frac{1}{4}\Big(\log(1-s(b)) + \log s(w_2+b) + \log s(w_1+b) + \log s(w_1+w_2+b)\Big).$$

MSE와 비교를 위해서 Cross Entropy의 w_1 에 대한 편미분을 구하면 다음과 같다.

$$\frac{\partial cost(W,b)}{\partial w_1} = -\frac{1}{4}\Big(\big(1-s(w_1+b)\big) + \big(1-s(w_1+w_2+b)\big)\Big)$$

Cross Entropy cost 함수를 이용하여 위 **OR** 문제를 로지스틱회귀분석을 실시하면 다음과 같은 결과를 얻을 수 있다(학습 횟수 =100, learning rate = 0.5).

```
Step : 100, Cost : 0.161213  Accuracy : 1.0
```

TensorFlow code

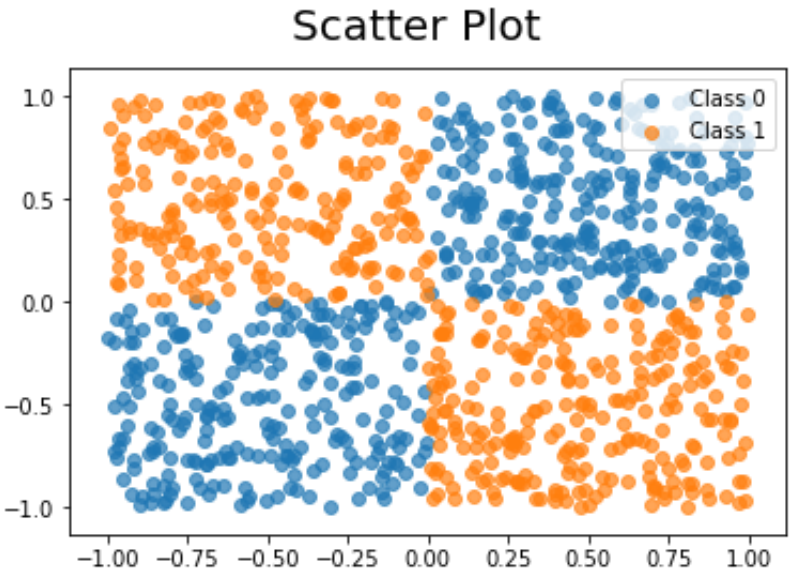
```
import tensorflow as tf
import matplotlib.pyplot as plt

%matplotlib inline

# 데이터 구성
x_data = [[0,0],[0,1],[1,0],[1,1]]

y_data = [[0],[1],[1],[1]]

# 모델 설계(로지스틱회귀분석)
X = tf.placeholder(tf.float32, shape=[None,2])
Y = tf.placeholder(tf.float32, shape=[None,1])

W = tf.Variable(tf.random_normal([2,1]), dtype=tf.float32)
b = tf.Variable(tf.random_normal([1]), dtype=tf.float32)

H = tf.sigmoid(tf.matmul(X,W)+b)

# cost 함수 정의(cross entropy), learning rate = 0.5
loss = - tf.reduce_mean(Y * tf.log(H) + (1 - Y) * tf.log(1 - H))
train = tf.train.GradientDescentOptimizer(learning_rate=0.5).minimize(loss)

# Accuracy computation
# True if hypothesis>0.5 else False
predicted = tf.cast(H > 0.5, dtype=tf.float32)
accuracy = tf.reduce_mean(tf.cast(tf.equal(predicted, Y), dtype=tf.float32))
```

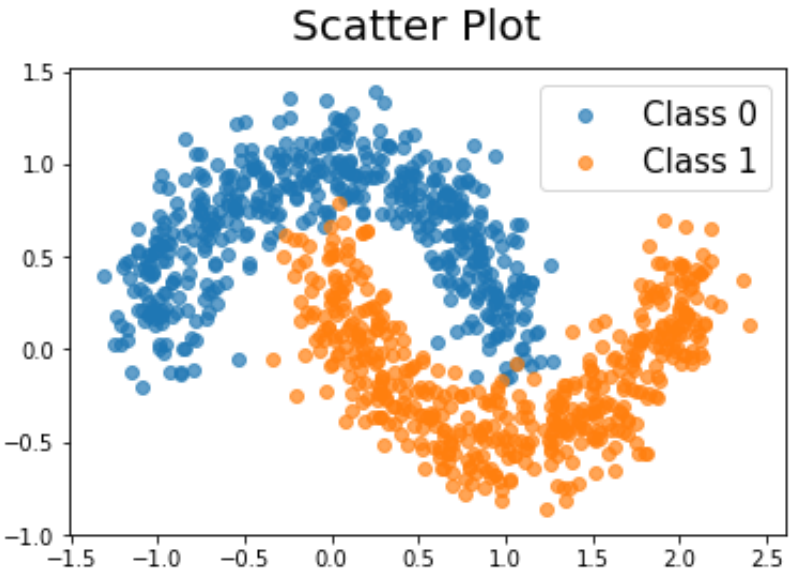
```
sess = tf.Session()
sess.run(tf.global_variables_initializer())

cost_list = []
acc_list = []

iteration = 100
for step in range(iteration):
    acc, cost, _, = sess.run([accuracy, loss,  train], feed_dict={X: x_data, Y: y_da
    cost_list.append(cost)
    acc_list.append(acc)

print("Step : %i, Cost : %s  Accuracy : %s" %(step+1, cost, acc))

plt.figure(figsize=(12,9))
plt.subplot(221)
plt.xlabel("Steps")
_ = plt.plot(cost_list, "c")
plt.subplot(222)
plt.xlabel("Steps")
_ = plt.plot(acc_list, "k--")
```

Minjung Gim(mjgim.nims.re.kr), National Institute for Mathematical Sciences

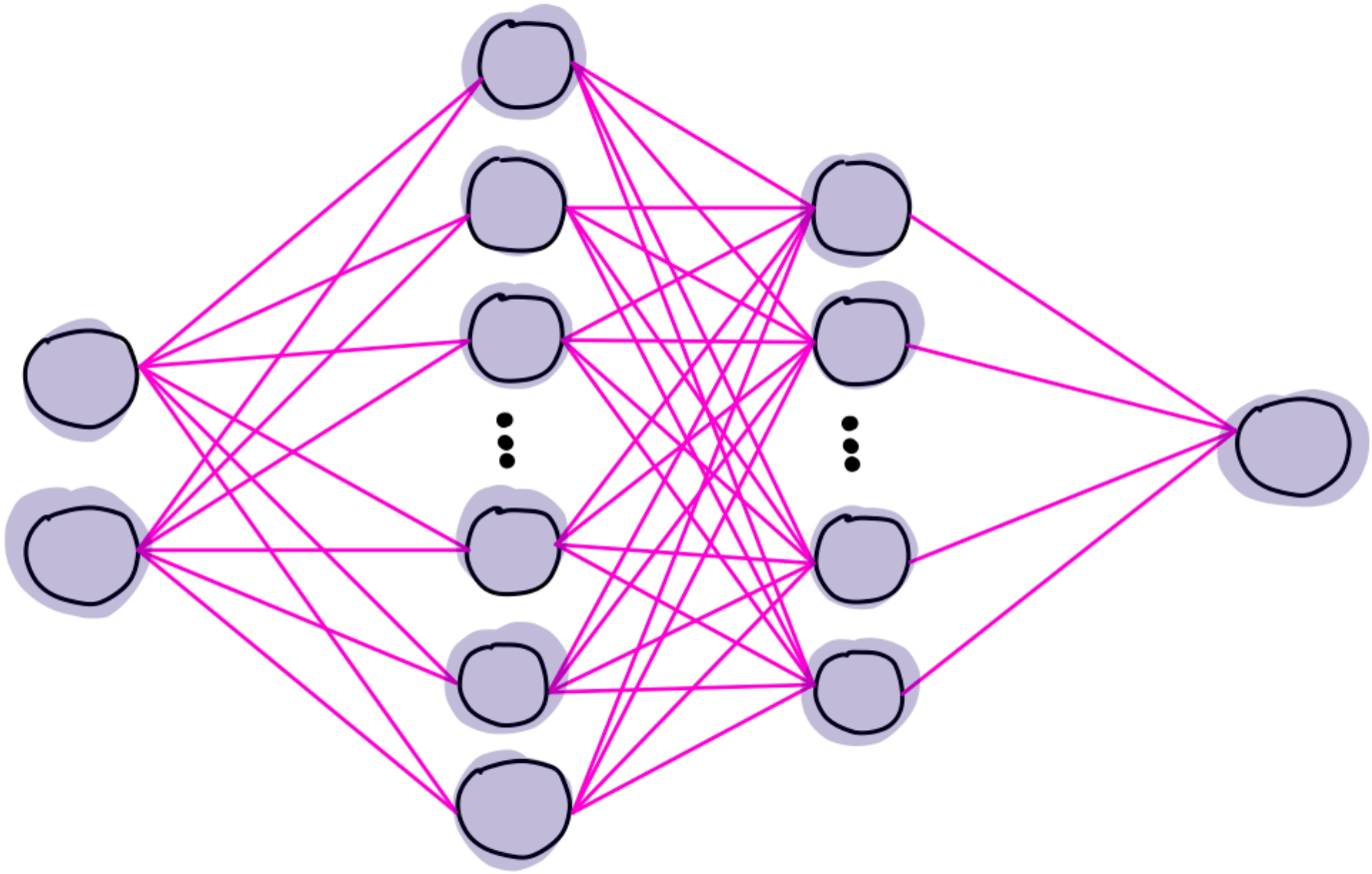
Binary classification using Multilayer Perceptron

종속변수 y가 categorical variable이고 라벨이 0과 1 두 개인 이진분류(binary classification)문제를 Multilayer perceptron(MLP)를 이용해서 해결해보자. 이진분류 문제이기 때문에 final layer의 activation 함수는 sigmoid를 사용한다(cost 함수는 cross entropy를 사용). 다음과 같이 주어진 데이터를 분류해보자.





X는 2차원이고 y는 0 또는 1을 갖는 데이터로 같은 색은 같은 클래스를 나타낸다. 두 데이터셋 모두 로지스틱회귀분석으로 해결하기 어려운 구조이다*(로지스틱회귀분석으로 분류를 하는 것은 주어진 데이터를 직선으로 분류(linear separation)하는 것이라고 생각하면 된다)*. 그래서 다음과 같이 Hidden layer가 2개 있는 MLP(Miltilayer Perceptron)로 분류 모델을 만들 것이다.



첫 번째 layer의 노드 수는 20, 두 번째 layer의 노드 수는 10으로 설정하였고 activation으로 모두 sigmoid를 사용하였다. - 입력 $X = (x_1, x_2)$, Weight $W^0 = \begin{pmatrix} w^0_{1\,1} & \cdots & w^0_{1\,20} \\ w^0_{2\,1} & \cdots & w^0_{2\,20} \end{pmatrix}$, bias $b^0 = (b^0_1 \cdots b^0_{20})$으로 첫 번째 hidden layer H^0을 다음과 같이 구성하였다.

$$H^0 = H^0(X) = f(XW^0 + b^0)$$

여기서 f는 sigmoid이다. 즉, 첫 번째 hidden layer $H^0 = (h^0_1, \cdots, h^0_{20})$은 20차원의 벡터다.

- 두 번째 hidden layer H^1은 H^0을 입력으로하고 Weight $W^1 = \begin{pmatrix} w^1_{1\,1} & \cdots & w^1_{1\,10} \\ & \cdots & \\ w^1_{20\,10} & \cdots & w^1_{20\,10} \end{pmatrix}$, bias $b^1 = (b^1_1, \cdots, b^1_{10})$으로 구성되고 식은 다음과 같다.

$$H^1 = H^1(X) = f(H^0W^1 + b^1)$$

위와 같은 이유로 $H^1 = (h^1_1, \cdots, h^1_{10})$은 10차원 벡터가 되고 마지막 H는 다음과 같다.

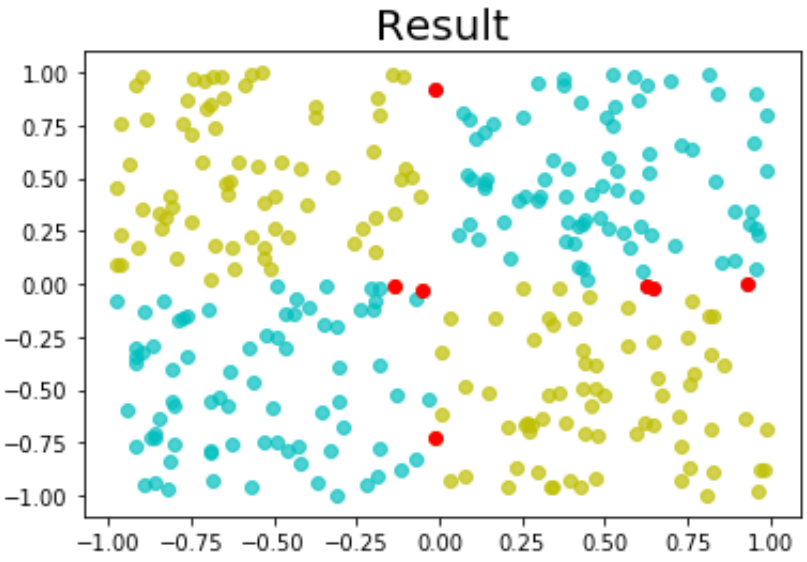
$$\begin{aligned} H = H(X) &= f(H^1W^2 + b^2) \\ &= f\left(\sum_{i=1}^{10} h^1_i \cdot w^2_i + b^2\right) \end{aligned}$$

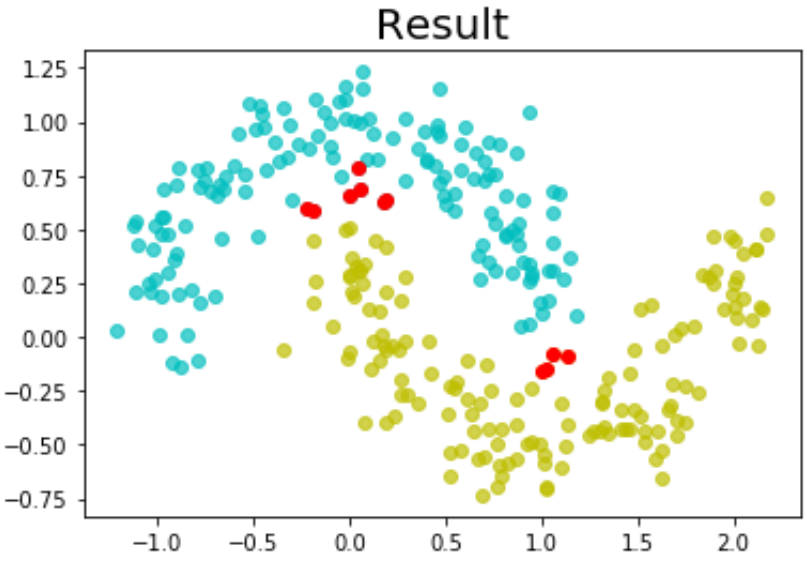
여기서 $W^2 = \begin{pmatrix} w_1^2 \\ \vdots \\ w_{10}^2 \end{pmatrix}$, bias $b^2 = b^2$.

학습데이터와 테스트데이터의 비율은 7:3으로 설정하여 학습을 진행하였다.

결과

다음은 학습에 사용하지 않은 테스트데이터의 산포도를 나타내며 민트색은 파란색(Class 0) 그룹이라고 판단한 점이고 녹색은 오렌지색(Class 1) 그룹이라고 판단한 점들이다. 빨간색 점은 실제 라벨 y를 잘못 예측한 점들을 나타낸다.





`TensorFlow code`

XOR data

```
# 라이브러리 불러오기
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split

%matplotlib inline

# XOR 데이터셋 구성
np.random.seed(20180324)

x_data = np.random.uniform(-1,1, [1000,2])
y_data = np.array([0 if x_data[i][0]*x_data[i][1] >= 0.
else 1 for i in range(len(x_data))])

plt.suptitle("Scatter Plot", fontsize=20)
plt.scatter(x_data[y_data == 0, 0], x_data[y_data == 0, 1], label="Class 0", alpha=0
plt.scatter(x_data[y_data == 1, 0], x_data[y_data == 1, 1], label="Class 1", alpha=0
plt.legend()

# 학습데이터와 테스트데이터 구분(비율 7:3)
x_train, x_test, y_train, y_test = \
train_test_split(x_data, y_data, test_size=0.3, random_state=777)

# MLP 모델 구성
X = tf.placeholder(tf.float32, shape=[None,2])
Y = tf.placeholder(tf.float32, shape=[None,1])

# Variables
W0 = tf.Variable(tf.random_normal([2,20]), dtype=tf.float32)
b0 = tf.Variable(tf.random_normal([20]), dtype=tf.float32)

W1 = tf.Variable(tf.random_normal([20,10]), dtype=tf.float32)
b1 = tf.Variable(tf.random_normal([10]), dtype=tf.float32)

W2 = tf.Variable(tf.random_normal([10,1]), dtype=tf.float32)
b2 = tf.Variable(tf.random_normal([1]), dtype=tf.float32)


def model(inputs):
    H0 = tf.sigmoid(tf.matmul(inputs,W0)+b0)
    H1 = tf.sigmoid(tf.matmul(H0,W1)+b1)
    H = tf.sigmoid(tf.matmul(H1,W2)+b2)

    return H
```

```
H = model(X)

loss = - tf.reduce_mean(Y * tf.log(H) + (1 - Y) * tf.log(1 - H))
train = tf.train.GradientDescentOptimizer(learning_rate=0.05).minimize(loss)

predicted = tf.cast(H > 0.5, dtype=tf.float32)
accuracy = tf.reduce_mean(tf.cast(tf.equal(predicted, Y), dtype=tf.float32))

sess = tf.Session()
sess.run(tf.global_variables_initializer())

cost_list = []
acc_list = []

iteration = 10000
for step in range(iteration):
    acc, cost, _, = sess.run([accuracy, loss,  train], feed_dict={X: x_train, Y: np.
    cost_list.append(cost)
    acc_list.append(acc)
    if (step+1) % (iteration//10) ==0:
        print("Step : %i, Cost : %s  Accuracy : %s" %(step+1, cost, acc))

plt.figure(figsize=(12,10))
plt.subplot(221)
plt.xlabel("Steps")
plt.title("Cost", fontsize=20)
_ = plt.plot(cost_list, "c")

plt.subplot(222)
plt.xlabel("Steps")
plt.title("Accuracy", fontsize=20)
_ = plt.plot(acc_list, "k--")

# 학습한 모델을 테스트 데이터에 적용
Hypothesis = sess.run(H, feed_dict={X: x_test})

plt.subplot(223)
plt.title("Result", fontsize=20)
for i, j in enumerate(Hypothesis):
    if j > 0.5 :
        _ = plt.plot(x_test[i][0], x_test[i][1], "yo", alpha=0.7)
    else :
        _ = plt.plot(x_test[i][0], x_test[i][1], "co", alpha=0.7)
for i in range(len(Hypothesis)):
    if np.around(Hypothesis)[i] != y_test[i]:
        _ = plt.plot(x_test[i][0], x_test[i][1], "ro")
```

Moon data

`sklearn` 을 이용하여 Moon 데이터셋을 만들 수 있고 동일한 MLP 모델에 적용 할 수 있다.

```
# Moon data
x_moon, y_moon = make_moons(n_samples=1000, shuffle=True, noise=0.15, random_state=1

plt.suptitle("Scatter Plot", fontsize=20)
plt.scatter(x_moon[y_moon == 0, 0], x_moon[y_moon == 0, 1], label="Class 0", alpha=0
plt.scatter(x_moon[y_moon == 1, 0], x_moon[y_moon == 1, 1], label="Class 1", alpha=0
plt.legend(fontsize=15)

x_train, x_test, y_train, y_test = \
train_test_split(x_moon, y_moon, test_size=0.3, random_state=777)
```

Minjung Gim(mjgim.nims.re.kr), National Institute for Mathematical Sciences

Avoid Overfitting

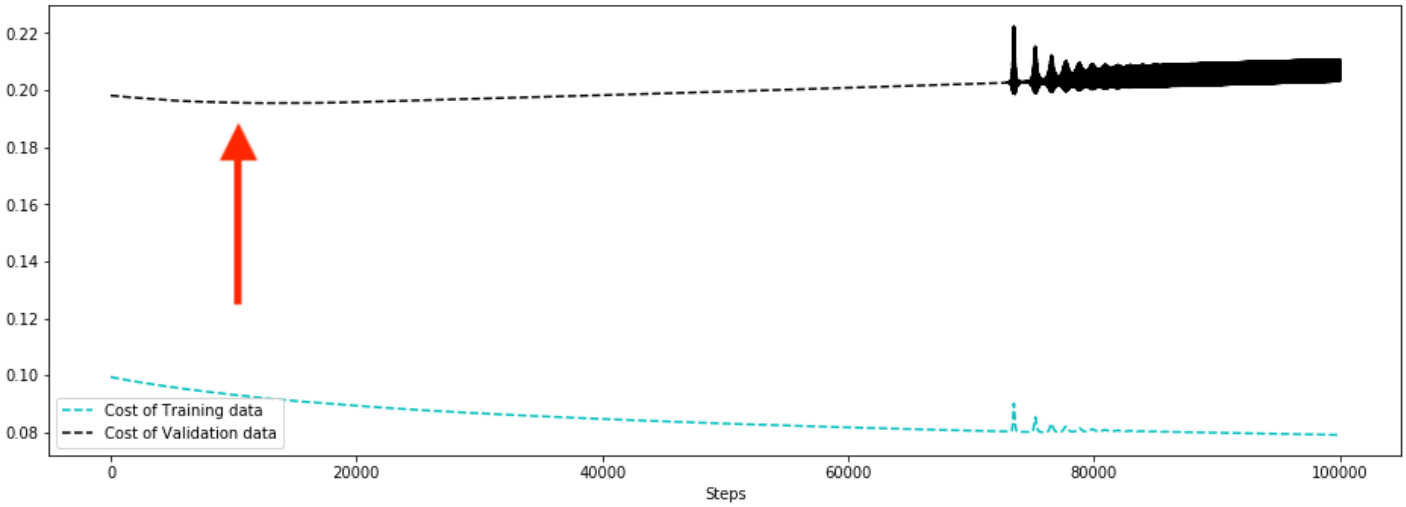
Overfitting(과적합) 이란 데이터의 학습결과가 학습데이터에만 최적화되어 모집단에 대한 일반화(Generalization)에 실패한 경우를 의미한다. 일반적으로 딥러닝 알고리즘에서는 Underfitting 보다 Overfitting이 자주 발생하기 때문에 딥러닝 모델을 구성할 때 이를 피하는 방법을 항상 고려해야 한다.

Overfitting을 피하기 위해 주로 사용하는 방법은 다음과 같다.

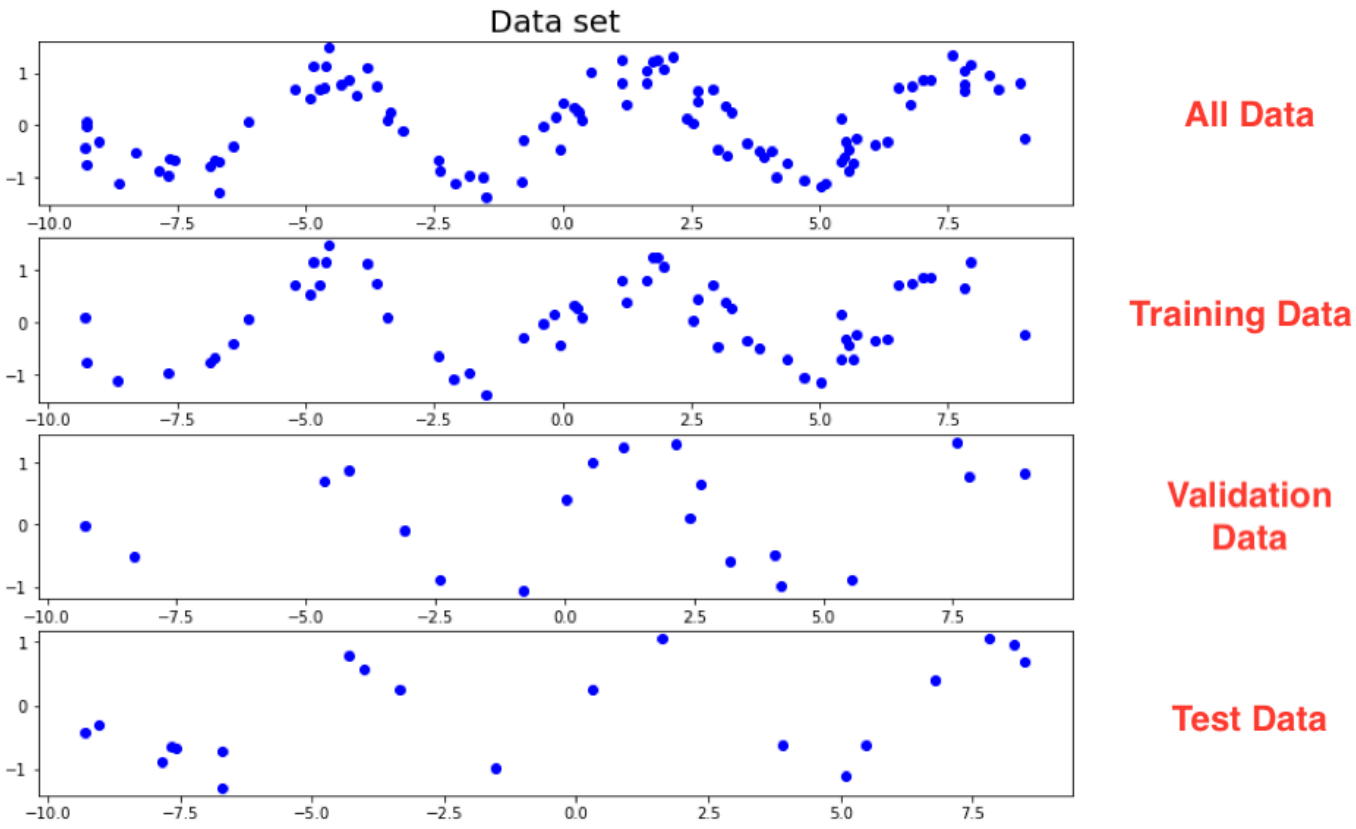
1. **Validation data set**
2. **Regularization**
3. **Dropout**

1. Validation data set

Validation data set을 이용하는 방법은 주어진 데이터를 학습데이터와 테스트데이터 `Validation data` 라고 하는 `확인데이터` (또는 `검증데이터`)로 분할하는 것이다(학습데이터와 테스트데이터 두 개로만 구분하는 것이 아닌). 여기서 Validation data란 데이터 학습에는 쓰이지는 않지만 cost를 계산해서 학습을 중단할 시점을 결정하는데 쓰이는 데이터이다. 딥러닝 알고리즘에서 반복적으로 학습데이터로 학습을 진행하면 학습데이터의 cost함수는 계속 감소하게된다. 이 과정에서 모델이 학습데이터에만 최적화되는 Overfitting 현상이 발생하는데 학습에 사용되지 않는 Validation data의 cost값이 증가하는 시점에서 학습을 중단시켜 Overfitting되는 현상을 막는다(그림 참조). 즉, validation data를 활용하여 학습 횟수를 결정한다.



- Training data : 모집단의 분포를 예측하기 위한 학습에 쓰이는 데이터이다.
- Validation data : 학습에는 쓰이지 않고 학습데이터에 모델이 Overfitting되는 현상을 막는데 쓰인다.
- Test data : 학습이 제대로 이루어 졌는지 즉, 일반화가 잘 됐는지 확인하는데 쓰인다.



2. Regularization

두 번째 방법으로, Regularization term (또는 weight decay term라고도 함)을 활용하는 방법이 있다. 이후 진행되는 코드에서는 Overfitting이 발생하는 경우를 확인하기 위한 것으로 테스트데이터와 Validation 데이터를 따로 설정하지 않았

다. **Regularization term**은 예측값과 실제값의 오차를 계산하는 cost function에 Weight에 대한 강도를 감소시키기 위한 cost function에 추가하는 항으로 보통 weight의 L^2 -norm 이나 L^1 -norm을 사용한다. 우리는 목적함수 cost를 최소화하는 쪽으로 학습을 진행하기 때문에 weight의 값들이 과도하게 커지는 것을 막을 수 있다.

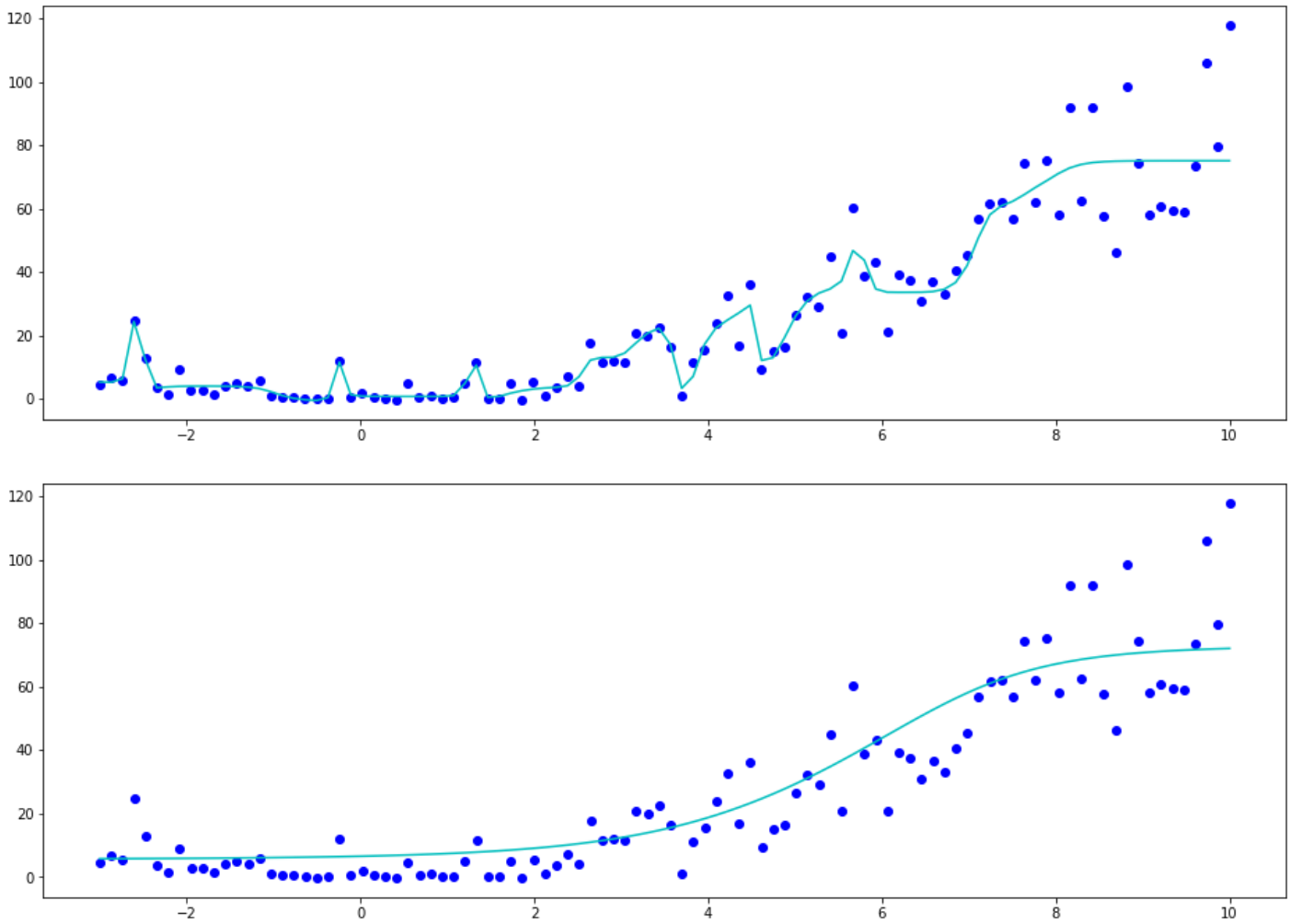
즉, cost function을 다음과 같이 정의한다.

$$cost += \lambda \frac{1}{2} \sum_{weight\ W} W^2 \qquad (L^2 - loss)$$

또는

$$cost += \lambda \sum_{weight\ W} |W| \qquad (L^1 - loss).$$

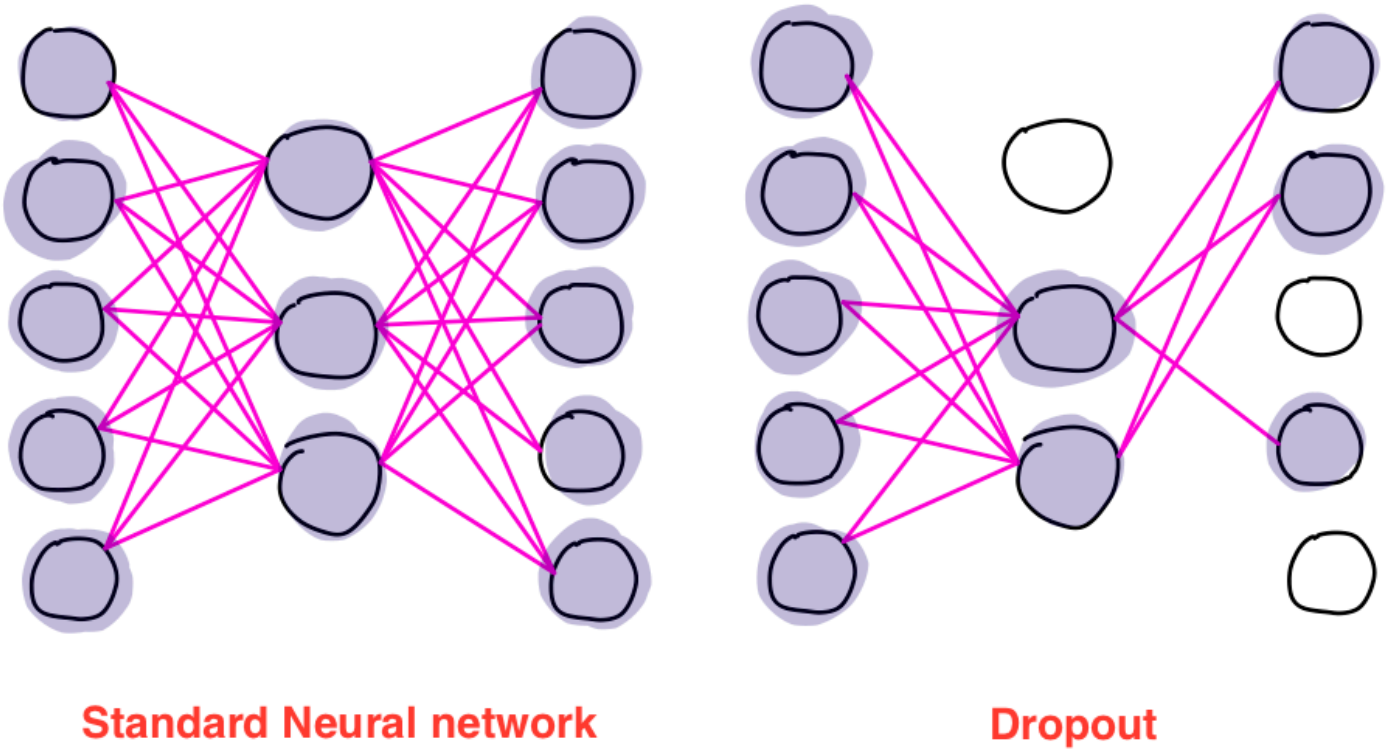
λ는 regularization constant로 Weight의 강도를 감소시키는 것의 중요도를 의미한다(λ=0인 경우 기존의 cost function과 같다). 아래 그림은 동일한 데이터를 동일 횟수로 학습한 결과를 보여준다.



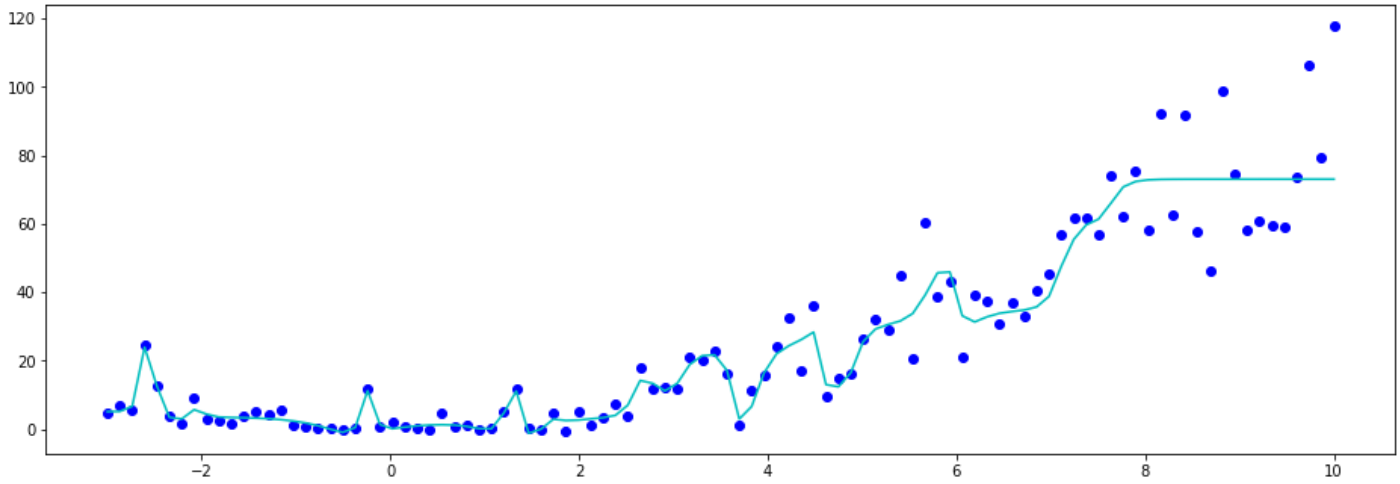
파란색 점은 학습데이터를 나타내고 민트색선은 예측값을 나타낸다. 위의 그림은 cost로 MSE를 사용하였고 아래 그림은 MSE에 **Regularization term**을 추가 한 것이다. 그림에서 보는 것처럼 **Regularization term** 추가로 overfitting을 막는 효과를 확인할 수 있다.

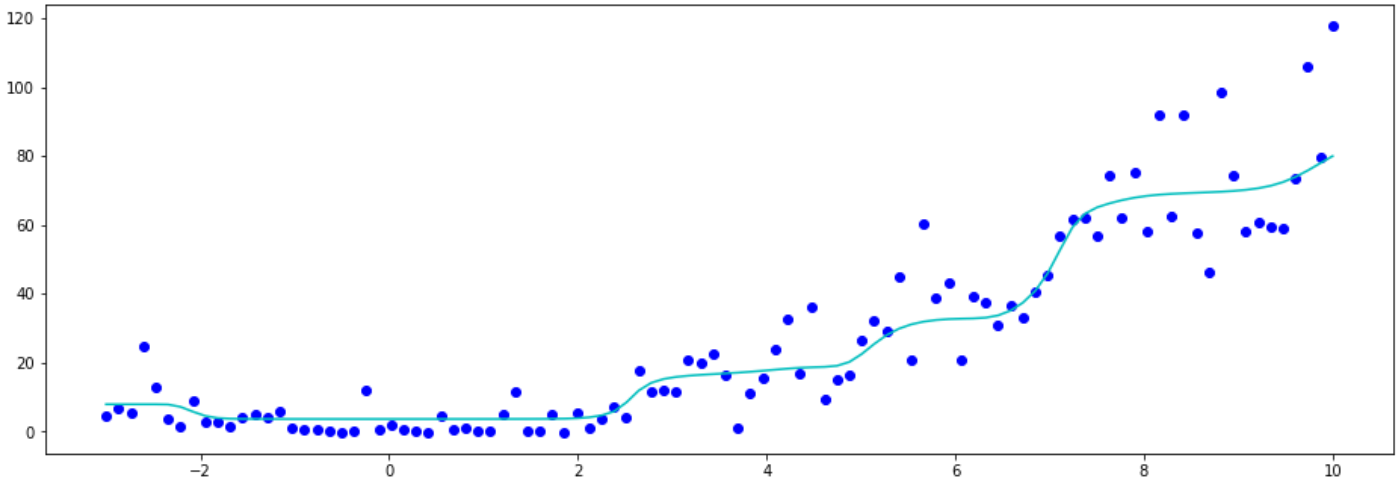
3. Dropout

마지막으로 **Dropout**이란 딥러닝 네트워크를 구성할 때 모든 노드를 complete graph로 연결하는 것이 아니라 확률적으로 노드를 막아서 데이터 학습을 진행하는 방법을 말한다. 즉, weight와 bias가 학습할 때 확률적으로 선택된 노드에서만 진행한다고 생각하면 된다. 즉 노드를 남길 확률인 keep*prob=1 인 경우는 Dropout을 사용하지 않은 것이랑 같은 알고리즘다. 최종적으로 예측값을 구할 때는 노드를 막지 않고(keep*prob=1) complete graph 상태에서 예측값을 구한다.



아래 그림은 위와 같이 동일한 데이터를 동일 횟수로 학습했고 두 경우 모두 cost function은 MSE를 사용하였다.





위의 그림은 Dropout을 사용하지 않고 학습을 진행해서 예측한 결과를 보여주는 그래프이고 아래 그림은 Dropout (keep_prob=0.5)을 사용하여 학습한 결과다. 위 그림에서 보는 것 처럼 Dropout을 사용한 예측이 Overfitting 없이 적절하게 모집단의 분포를 파악하고 있는 것을 확인할 수 있다.

TensorFlow Code

1. Validation data set

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

# 데이터 구성 (100개 데이터)
np.random.seed(20)
tf.set_random_seed(20)
```

```
x_data = np.random.uniform(-10, 10, size=[100,1])

y_data = np.sin(x_data + 0.2 * np.random.normal(size=[100,1]))- 0.3 * np.random.norm

# 딥러닝 네트워크 구성
X = tf.placeholder(tf.float32, shape=[None,1])
Y = tf.placeholder(tf.float32, shape=[None,1])

W0 = tf.Variable(tf.random_normal([1,50]), dtype=tf.float32)
b0 = tf.Variable(tf.random_normal([50]), dtype=tf.float32)

H0 = tf.sigmoid(tf.matmul(X,W0)+b0)

W1 = tf.Variable(tf.random_normal([50,50]), dtype=tf.float32)
b1 = tf.Variable(tf.random_normal([50]), dtype=tf.float32)

H1 = tf.sigmoid(tf.matmul(H0,W1)+b1)

W2 = tf.Variable(tf.random_normal([50,1]), dtype=tf.float32)
b2 = tf.Variable(tf.random_normal([1]), dtype=tf.float32)

H = tf.matmul(H1,W2)+b2

loss = tf.reduce_mean(tf.square(H-Y))

train = tf.train.GradientDescentOptimizer(learning_rate = 0.01).minimize(loss)

sess = tf.Session()
sess.run(tf.global_variables_initializer())

# 학습 및 결과 시각화
list_tr_cost = []
list_va_cost = []
list_te_cost = []

# training 60개, validation 20개, test 20개

iteration = 150000
for step in range(iteration):
    _, cost0 = sess.run([train, loss], feed_dict={X: x_data[:60], Y: y_data[:60]})
    cost1 = sess.run(loss, feed_dict={X: x_data[60:80], Y: y_data[60:80]})
    list_tr_cost.append(cost0)
    list_va_cost.append(cost1)
    if (step+1) % (iteration//10) ==0 :
        print("Step : %i, Cost : %s" %((step+1), cost0))

prediction = sess.run(H, feed_dict={X: x_data[80:]})
```

```
plt.figure(figsize=(16,12))
plt.subplot(211)
plt.xlabel("Steps")
_ = plt.plot(list_tr_cost[50000:], "c--", label='Cost of Training data')
_ = plt.plot(list_va_cost[50000:], "k--", label='Cost of Validation data')

plt.legend(loc='lower left')


plt.subplot(212)
_ = plt.plot(x_data[80:], y_data[80:], "bo")
_ = plt.plot(x_data[80:], prediction, "co")
```

실행화면

```
Step : 15000, Cost : 0.139464
Step : 30000, Cost : 0.118637
Step : 45000, Cost : 0.103913
Step : 60000, Cost : 0.0931755
Step : 75000, Cost : 0.08788
Step : 90000, Cost : 0.0847073
Step : 105000, Cost : 0.0824003
Step : 120000, Cost : 0.0806131
Step : 135000, Cost : 0.0803184
Step : 150000, Cost : 0.0791041
```

2. Regularization

loss0은 MSE cost function이고 loss1은 $\lambda = 1$인 regularization term이 추가된 cost function이다.

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

# 데이터 구성
np.random.seed(20)
tf.set_random_seed(20)
x_data = np.reshape(np.linspace(-3, 10, 100), [-1,1])

y_data = (x_data + np.random.normal(size=[100,1]))**2- 0.3 * np.random.normal(size=[
```

```
# 딥러닝 네트워크 구성
X = tf.placeholder(tf.float32, shape=[None,1])
Y = tf.placeholder(tf.float32, shape=[None,1])

W0 = tf.Variable(tf.random_normal([1,50]), dtype=tf.float32)
b0 = tf.Variable(tf.random_normal([50]), dtype=tf.float32)

H0 = tf.sigmoid(tf.matmul(X,W0)+b0)

W1 = tf.Variable(tf.random_normal([50,50]), dtype=tf.float32)
b1 = tf.Variable(tf.random_normal([50]), dtype=tf.float32)

H1 = tf.sigmoid(tf.matmul(H0,W1)+b1)

W2 = tf.Variable(tf.random_normal([50,1]), dtype=tf.float32)
b2 = tf.Variable(tf.random_normal([1]), dtype=tf.float32)

H = tf.matmul(H1,W2)+b2

# 두 가지 loss(cost) function
# loss0 : MSE loss, loss1 : regularization 추가

loss0 = tf.reduce_mean(tf.square(H-Y))

Ws = [W0, W1, W2]

loss1 = loss0
for weight in Ws:
    loss1 += tf.nn.l2_loss(weight)

loss = loss1

train = tf.train.GradientDescentOptimizer(learning_rate = 0.01).minimize(loss)

sess = tf.Session()
sess.run(tf.global_variables_initializer())

list_tr_cost = []

iteration = 100000
for step in range(iteration):
    _, cost0 = sess.run([train, loss], feed_dict={X: x_data, Y: y_data})
    list_tr_cost.append(cost0)
    if (step+1) % (iteration//10) ==0 :
        print("Step : %i, Cost : %s" %((step+1), cost0))

prediction = sess.run(H, feed_dict={X: x_data})
```

```
plt.figure(figsize=(16,12))
plt.subplot(211)
plt.xlabel("Steps")
_ = plt.plot(list_tr_cost, "c--", label='Cost of Training data')

plt.legend(loc='lower left')


plt.subplot(212)
_ = plt.plot(x_data, y_data, "bo")
_ = plt.plot(x_data, prediction, "c")
```

3. Dropout

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

# 데이터 구성
np.random.seed(20)
tf.set_random_seed(20)
x_data = np.reshape(np.linspace(-3, 10, 100), [-1,1])

y_data = (x_data + np.random.normal(size=[100,1]))**2- 0.3 * np.random.normal(size=[

# 딥러닝 네트워크 구성
# 노드를 확률적으로 막음 (keep_prob)
X = tf.placeholder(tf.float32, shape=[None,1])
Y = tf.placeholder(tf.float32, shape=[None,1])

prob = tf.placeholder(tf.float32)

W0 = tf.Variable(tf.random_normal([1,50]), dtype=tf.float32)
b0 = tf.Variable(tf.random_normal([50]), dtype=tf.float32)

H0 = tf.nn.dropout(tf.sigmoid(tf.matmul(X,W0)+b0), keep_prob=prob)

W1 = tf.Variable(tf.random_normal([50,50]), dtype=tf.float32)
b1 = tf.Variable(tf.random_normal([50]), dtype=tf.float32)

H1 = tf.nn.dropout(tf.sigmoid(tf.matmul(H0,W1)+b1), keep_prob=prob)

W2 = tf.Variable(tf.random_normal([50,1]), dtype=tf.float32)
b2 = tf.Variable(tf.random_normal([1]), dtype=tf.float32)
```

```
H = tf.matmul(H1,W2)+b2

loss = tf.reduce_mean(tf.square(H-Y))


train = tf.train.GradientDescentOptimizer(learning_rate = 0.01).minimize(loss)

sess = tf.Session()
sess.run(tf.global_variables_initializer())

list_tr_cost = []


# 학습을 진행할 때는 keep_prob = 0.5, 예측할 때는 keep_prob = 1
iteration = 100000
for step in range(iteration):
    _, cost0 = sess.run([train, loss], feed_dict={X: x_data, Y: y_data, prob : 0.5})
    list_tr_cost.append(cost0)
    if (step+1) % (iteration//10) ==0 :
        print("Step : %i, Cost : %s" %((step+1), cost0))

prediction = sess.run(H, feed_dict={X: x_data, prob : 1})

plt.figure(figsize=(16,12))
plt.subplot(211)
plt.xlabel("Steps")
_ = plt.plot(list_tr_cost, "c--", label='Cost of Training data')

plt.legend(loc='lower left')


plt.subplot(212)
_ = plt.plot(x_data, y_data, "bo")
_ = plt.plot(x_data, prediction, "c")
```