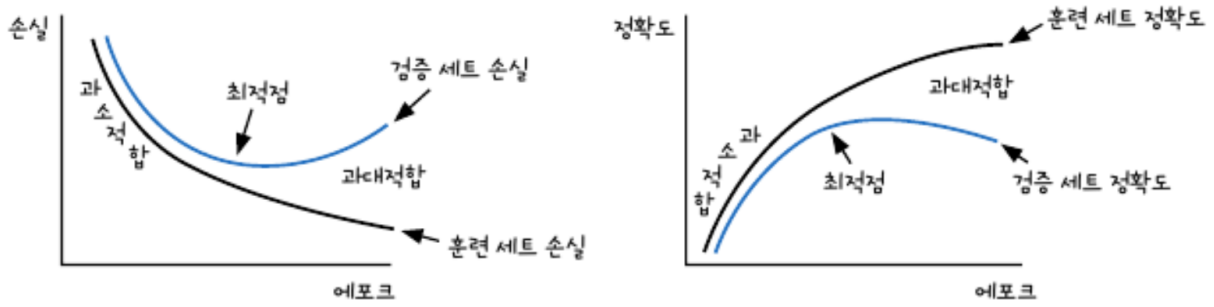
03_신경망 최적화

신경망 최적화

- 최적화

> **신경망 학습의 한계**

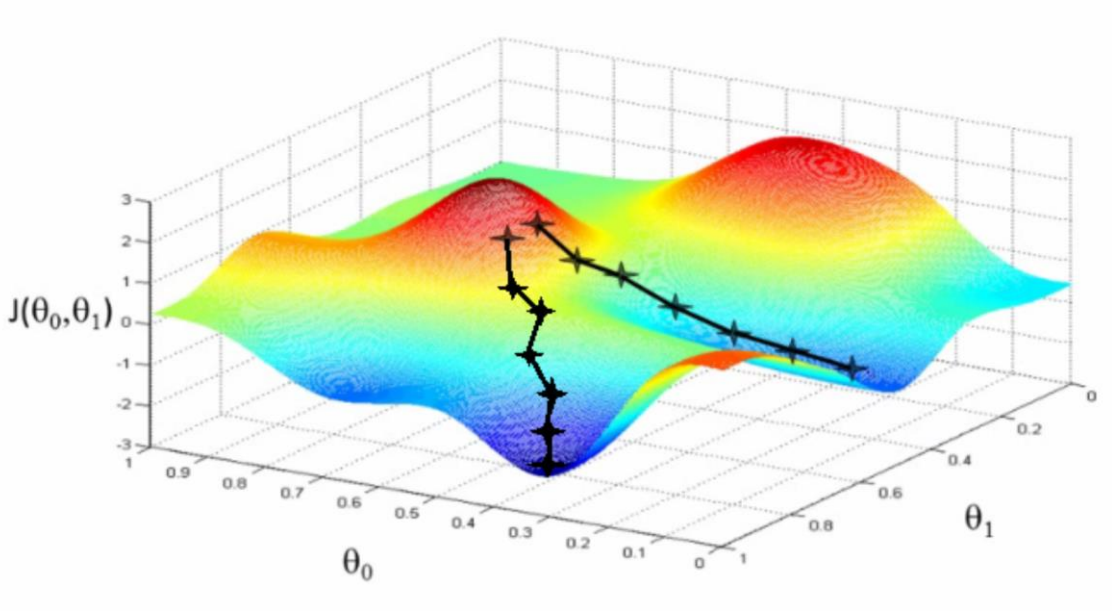
- 신경망 학습은 복잡한 문제를 잘 풀어내지만 쉽게 과적합 되는 구조 때문에 학습이 까다롭다.
- 과적합을 예방하는 근본적인 방법으로는 학습 데이터의 수를 늘리는 방법이 있지만 현실적으로 문제를 해결하기 어렵다.
- 이를 해결하기 위한 다양한 최적화 기법이 존재한다.



• 가중치 초기화

> 가중치 초기화의 중요성

- 신경망의 목적은 손실 함수(Loss)를 최소화하는 과정이다.
- 다음과 같은 손실 함수 그래프가 존재한다고 하면 첫 시작을 어디에서부터 하느냐에 따라 학습의 정도가 달라지게 된다.



신경망 최적화

• 가중치 최적화

> 고정값 최적화
  - 다음과 같이 모든 W를 1로, b를 0으로 고정된 값을 초기화

```
def init_weights(self, n_features):
    self.w1 = np.ones((n_features, self.units))
    self.b1 = np.zeros(self.units)
    self.w2 = np.ones((self.units, 1))
    self.b2 = 0
```

• 가중치 최적화

> 고정값 최적화

```
class DualLayer():
    def __init__(self, units = 10):
        self.units = units
        self.w1 = None
        self.b1 = None
        self.w2 = None
        self.b2 = None
        self.a1 = None
        self.losses = []

    def forpass(self, x):
        z1 = np.dot(x, self.w1) + self.b1
        self.a1 = self.activation(z1)
        z2 = np.dot(self.a1, self.w2) + self.b2
        return z2
```

신경망 최적화

• 가중치 최적화

> 고정값 최적화

```
def backprop(self, x, err):
    m = len(x)
    w2_grad = np.dot(self.a1.T, err) / m
    b2_grad = np.sum(err) / m
    err_to_hidden = np.dot(err, self.w2.T) * self.a1 * (1 - self.a1)
    w1_grad = np.dot(x.T, err_to_hidden) / m
    b1_grad = np.sum(err_to_hidden, axis=0) / m
    return w1_grad, b1_grad, w2_grad, b2_grad

def init_weights(self, n_features):
    self.w1 = np.ones((n_features, self.units))
    self.b1 = np.zeros(self.units)
    self.w2 = np.ones((self.units, 1))
    self.b2 = 0
```

• 가중치 최적화

> 고정값 최적화

```
def training(self, x, y, m):
    z = self.forpass(x)
    a = self.activation(z)
    err = -(y - a)
    w1_grad, b1_grad, w2_grad, b2_grad = self.backprop(x, err)
    self.w1 -= w1_grad
    self.b1 -= b1_grad
    self.w2 -= w2_grad
    self.b2 -= b2_grad
    return a

def activation(self, z):
    z = np.clip(z, -100, None)
    a = 1 / (1 + np.exp(-z))
    return a
```

신경망 최적화

• 가중치 최적화

> 고정값 최적화

```
def fit(self, x, y, epochs = 100):
    y = y.reshape(-1,1)
    m = len(x)
    self.init_weights(x.shape[1])
    for i in range(epochs):
        a = self.training(x, y, m)
        a = np.clip(a, 1e-10, 1-1e-10)
        loss = np.sum(-(y * np.log(a) + (1 - y) * np.log(1 - a)))
        self.losses.append(loss / m)

def predict(self, x):
    z = self.forpass(x)
    return z > 0

def score(self, x, y):
    return np.mean(self.predict(x) == y.reshape(-1, 1))
```

• 가중치 최적화

> 고정값 최적화

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
import numpy as np
from sklearn.preprocessing import StandardScaler

cancer = load_breast_cancer()
X = cancer.data
y = cancer.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

• 가중치 최적화

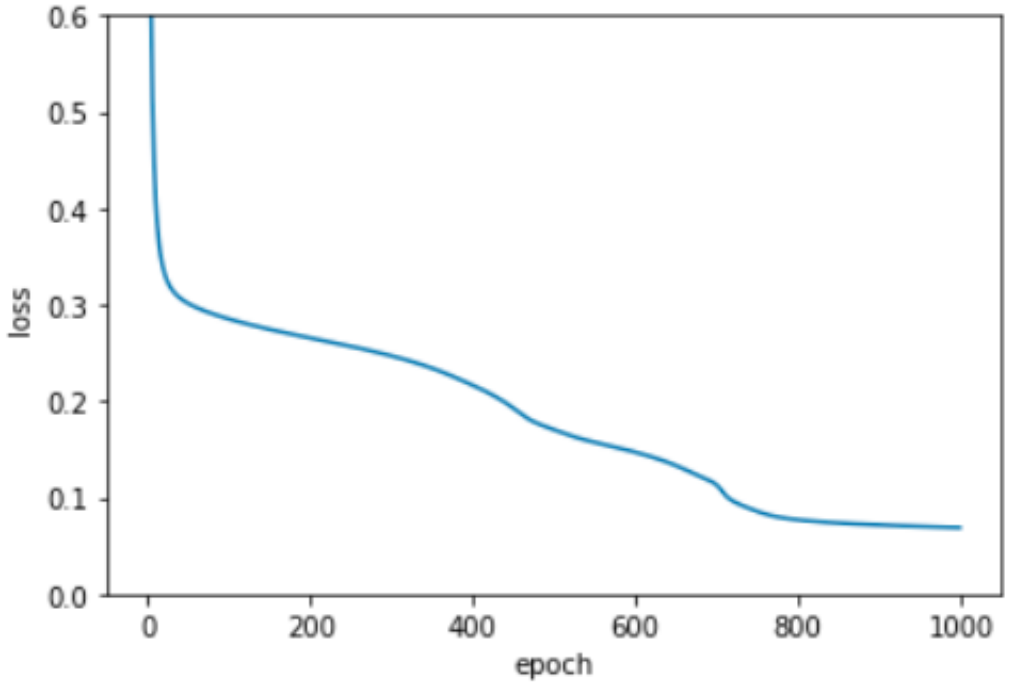
> 고정값 최적화

```
import matplotlib.pyplot as plt

dual_layer = DualLayer()
dual_layer.fit(X_train_scaled, y_train, epochs=1000)
print(dual_layer.score(X_test_scaled, y_test))

plt.plot(dual_layer.losses)
plt.ylim(0,0.6)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

• 가중치 최적화

> 고정값 최적화

Out : 0.956140350877193



가중치가 모두 동일하다면
학습이 부드럽게 진행되지
않는다.
즉, 랜덤한 값으로
가중치를 초기화 해야한다.

• 가중치 최적화

> Random 가중치 초기화
  - 가우시안 정규 분포를 따르는 랜덤한 수로 가중치 초기화

```
class RandomInitialization(DualLayer):
    def init_weights(self, n_features):
        np.random.seed(0)
        self.w1 = np.random.normal(0, 1, (n_features, self.units))
        self.b1 = np.zeros(self.units)
        self.w2 = np.random.normal(0, 1, (self.units, 1))
        self.b2 = 0
```

신경망 최적화

• 가중치 최적화

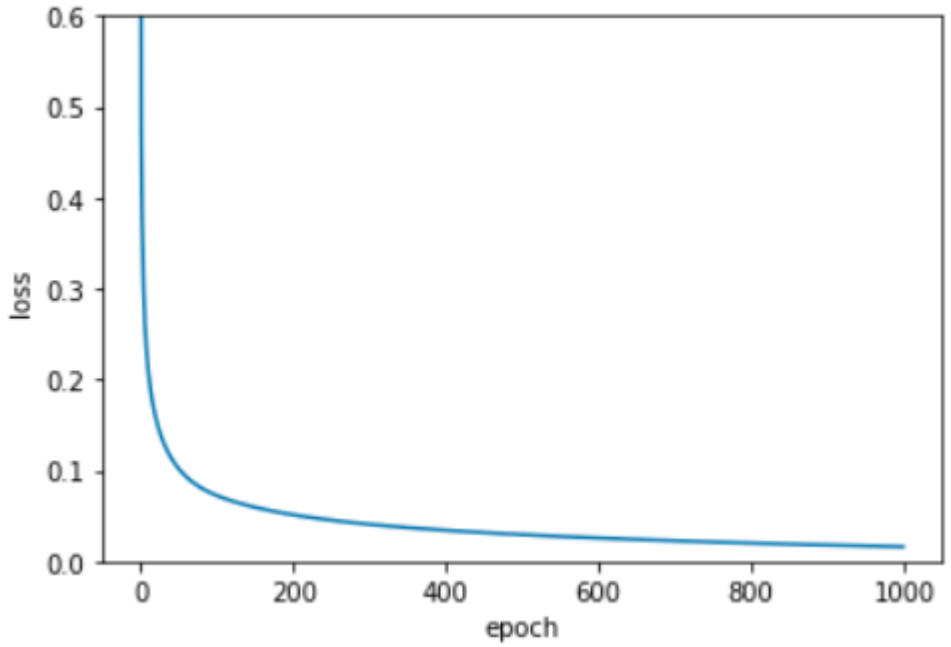
> Random 가중치 초기화
  - 가우시안 정규 분포를 따르는 랜덤한 수로 가중치 초기화

```
random_init = RandomInitialization()
random_init.fit(X_train_scaled, y_train, epochs = 1000)
plt.plot(random_init.losses)
plt.ylim(0,0.6)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

• 가중치 최적화

> Random 가중치 초기화
  - 가우시안 정규 분포를 따르는 랜덤한 수로 가중치 초기화

Out :



고정된 가중치보다 좋은 결과를 확인할 수 있다.
그러나 활성 함수에 따라 이는 좋지않은 결과를 나타낼 수 있다.

• 가중치 최적화

> Xavier 가중치 초기화

- 입력 데이터가 m개, 전달 데이터가 n개일 때, 평균은 0을 가지고 표준편차는 $\frac{2}{\sqrt{m+n}}$로 W를 초기화

```
class XavierInitialization(DualLayer):


    def init_weights(self, n_features):
        self.w1 = np.random.normal(0, 2 / np.sqrt(n_features + self.units),
                                   (n_features, self.units))
        self.b1 = np.zeros(self.units)
        self.w2 = np.random.normal(0, 2 / np.sqrt(n_features + self.units), (self.units, 1))
        self.b2 = 0
```

• 가중치 최적화

> Xavier 가중치 초기화

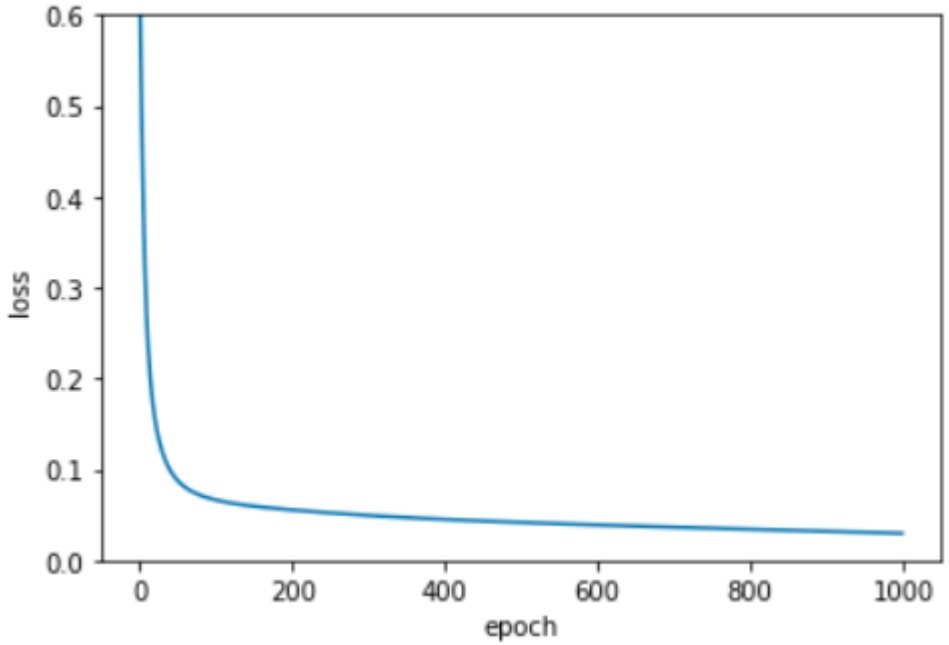
- 입력 데이터가 m개, 전달 데이터가 n개일 때, 평균은 0을 가지고 표준편차는 $\frac{2}{\sqrt{m+n}}$로 W를 초기화

```
xavier_init = XavierInitialization()
xavier_init.fit(X_train_scaled, y_train, epochs = 1000)
plt.plot(xavier_init.losses)
plt.ylim(0,0.6)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

• 가중치 최적화

> Xavier 가중치 초기화
  - 입력 데이터가 m개, 전달 데이터가 n개일 때, 평균은 0을 가지고 표준편차는 $\frac{2}{\sqrt{m+n}}$로 W를 초기화

Out :



• 가중치 최적화

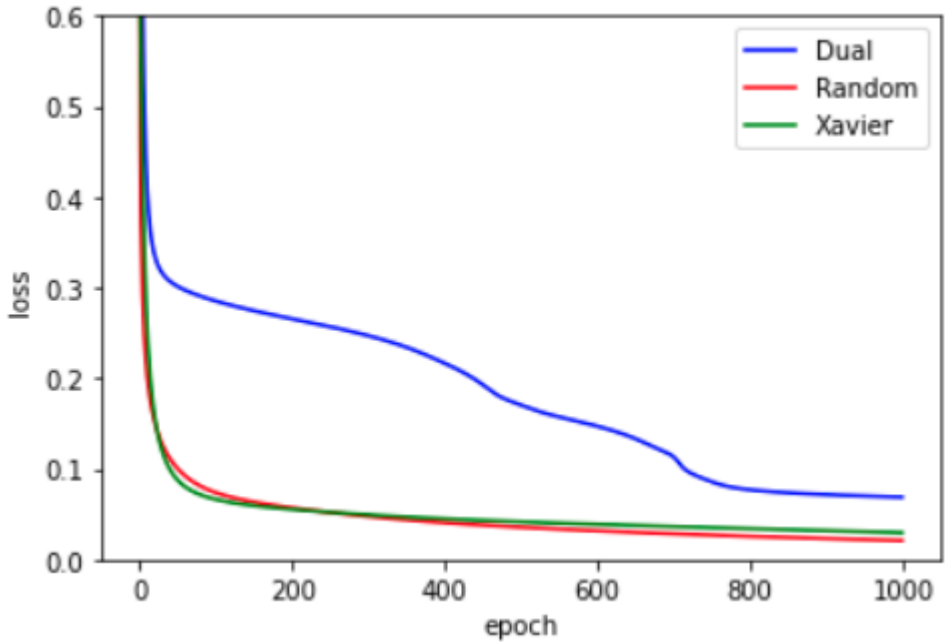
> 가중치 초기화 비교

```
plt.plot(dual_layer.losses, 'b', label = 'Dual')
plt.plot(random_init.losses, 'r', label = 'Random')
plt.plot(xavier_init.losses, 'g', label = 'Xavier')
plt.ylim(0,0.6)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```

• 가중치 최적화

> 가중치 초기화 비교

Out :



확인하면서 작업해야함
loss값이 어떻게 줄어드는지
학습이 잘 되었는지.

> 고정된 가중치보다 랜덤한 가중치를 주었을 때, 좀더 학습이 잘된 것을 확인할 수 있다.

• 규제 적용

> **규제(L1, L2)** L2규제를 더 많이 사용함

- 과적합을 해결하는 대표적인 방법으로 가중치의 값이 커지지 않도록 제한하는 기법
- L1

  : 손실 함수에 가중치의 절대값을 추가하여 제한한다.

  몇몇 중요한 가중치들만 남게 됨.

L1 Regularization

$$\text{Cost} = \sum_{i=0}^{N}(y_i - \sum_{j=0}^{M} x_{ij}W_j)^2 + \lambda \sum_{j=0}^{M} |W_j|$$

- L2

  : 손실 함수에 가중치의 제곱값을 추가하여 제한한다.

  전체적으로 가중치를 작아지게 한다.

L2 Regularization

$$\text{Cost} = \underbrace{\sum_{i=0}^{N}(y_i - \sum_{j=0}^{M} x_{ij}W_j)^2}_{\text{Loss function}} + \underbrace{\lambda \sum_{j=0}^{M} W_j^2}_{\text{Regularization Term}}$$

• 규제 적용

> **L1 규제 적용** : 엘라스틱 규제

- 이진분류인 모델에서 손실 함수인 로지스틱 손실 함수에 L1 규제를 추가하면 다음과 같다.

$$L=-(ylog(a)+(1-y)log(1-a))+\alpha\sum_{i=1}^{n}|w_i|$$

- 이때, 규제 앞 α는 규제의 양을 조절한다.

> **L1 규제의 미분**

- sign이라는 뜻은 절대값을 미분하면 부호만 남기에 이렇게 표현했다.

$$\frac{\partial}{\partial w}L=-(y-a)x+\alpha\times sign(w)$$

• 규제 적용

> 모델에 L1 규제 적용

```
class L1_regular(DualLayer):
    def __init__(self, units = 10, l1 = 0):
        self.units = units
        self.w1 = None
        self.b1 = None
        self.w2 = None
        self.b2 = None
        self.a1 = None
        self.l1 = l1
        self.losses = []
```

• 규제 적용

> 모델에 L1 규제 적용

```
def training(self, x, y, m):
    z = self.forpass(x)
    a = self.activation(z)
    err = -(y - a)
    w1_grad, b1_grad, w2_grad, b2_grad = self.backprop(x, err)

    self.w1 -= w1_grad + self.l1 * np.sign(self.w1)
    self.b1 -= b1_grad
    self.w2 -= w2_grad + self.l1 * np.sign(self.w2)
    self.b2 -= b2_grad
    return a
```

• 규제 적용

> 모델에 L1 규제 적용

```
dual_layer = DualLayer()
dual_layer.fit(X_train_scaled, y_train, epochs = 1000)
print(dual_layer.score(X_test_scaled, y_test))

l1 = L1_regular(l1 = 0.01)
l1.fit(X_train_scaled, y_train, epochs = 1000)
print(l1.score(X_test_scaled, y_test))
```

Out : 0.956140350877193

0.9473684210526315

• 규제 적용

> L2 규제 적용

- 이진분류인 모델에서 손실 함수인 로지스틱 손실 함수에 L2 규제를 추가하면 다음과 같다.

$$L=-(ylog(a)+(1-y)log(1-a))+\frac{1}{2}a\sum_{i=1}^{n}|w_i|^2$$

- ½ 값은 미분 결과를 보기 좋게 하기 위해서 추가했다.

> L2 규제의 미분

- L2를 미분하면 w만 남는다.

$$\frac{\partial}{\partial w}L=-(y-a)x+a\times w$$

규제를 손실함수에 걸어야
/ 계층마다 규제를 걸기도 함

• 규제 적용

> 모델에 L2 규제 적용

```
class L2_regular(DualLayer):
    def __init__(self, units = 10, l2 = 0):
        self.units = units
        self.w1 = None
        self.b1 = None
        self.w2 = None
        self.b2 = None
        self.a1 = None
        self.l2 = l2
        self.losses = []
```

• 규제 적용

> 모델에 L2 규제 적용

```
def training(self, x, y, m):
    z = self.forpass(x)
    a = self.activation(z)
    err = -(y - a)
    w1_grad, b1_grad, w2_grad, b2_grad = self.backprop(x, err)

    self.w1 -= w1_grad + self.l2 * self.w1
    self.b1 -= b1_grad
    self.w2 -= w2_grad + self.l2 * self.w2
    self.b2 -= b2_grad
    return a
```

• 규제 적용

> 모델에 L2 규제 적용

```
dual_layer = DualLayer()
dual_layer.fit(X_train_scaled, y_train, epochs = 1000)
print(dual_layer.score(X_test_scaled, y_test))

l2 = L2_regular(l2 = 0.01)
l2.fit(X_train_scaled, y_train, epochs = 1000)
print(l2.score(X_test_scaled, y_test))
```

Out : 0.956140350877193

0.9649122807017544

• 규제 적용

> 모델에 L1, L2 규제 적용 : 엘라스틱 규제

```
class Regularization(DualLayer):
    def __init__(self, units = 10, l1 = 0, l2 = 0):
        self.units = units
        self.w1 = None
        self.b1 = None
        self.w2 = None
        self.b2 = None
        self.a1 = None
        self.l1 = l1
        self.l2 = l2
        self.losses = []
```

• 규제 적용

> 모델에 L1, L2 규제 적용

```
def training(self, x, y, m):
    z = self.forpass(x)
    a = self.activation(z)
    err = -(y - a)
    w1_grad, b1_grad, w2_grad, b2_grad = self.backprop(x, err)

    self.w1 -= w1_grad + self.l1 * np.sign(self.w1) + self.l2 * self.w1
    self.b1 -= b1_grad
    self.w2 -= w2_grad + self.l1 * np.sign(self.w2) + self.l2 * self.w2
    self.b2 -= b2_grad

    return a
```

• 규제 적용

> 모델에 L1, L2 규제 적용

```
dual_layer = DualLayer()
dual_layer.fit(X_train_scaled, y_train, epochs = 1000)
print(dual_layer.score(X_test_scaled, y_test))


regular = Regularization(l1 = 0.01, l2 = 0.01)
regular.fit(X_train_scaled, y_train, epochs = 1000)
print(regular.score(X_test_scaled, y_test))
```

Out : 0.956140350877193

0.947368421052631 5

• 규제 적용

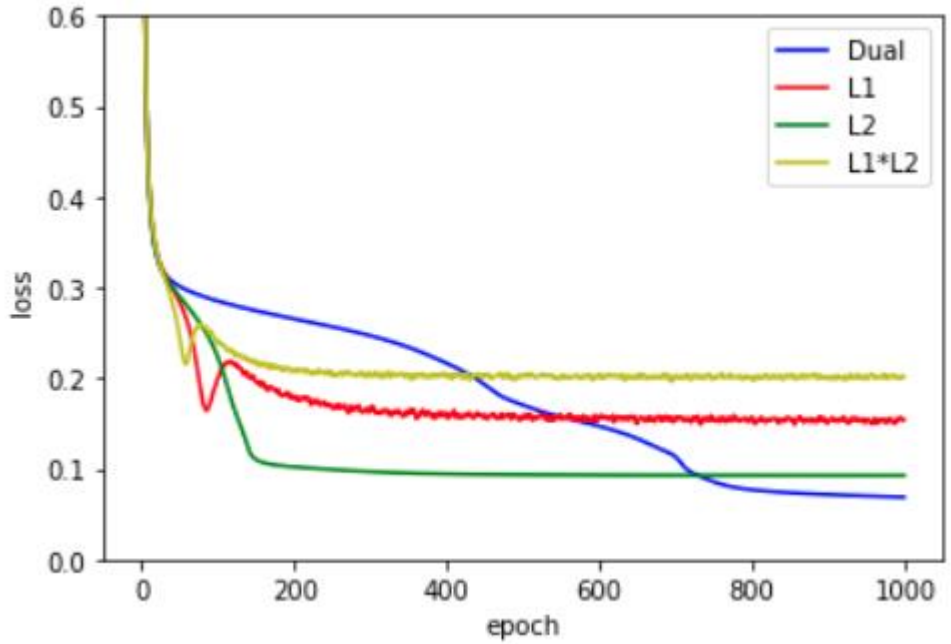
> 모델에 L1, L2 규제 적용

```
plt.plot(dual_layer.losses, 'b', label = 'Dual')
plt.plot(l1.losses, 'r', label = 'L1')
plt.plot(l2.losses, 'g', label = 'L2')
plt.plot(regular.losses, 'y', label = 'L1*L2')
plt.ylim(0,0.6)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```

• 규제 적용

> 모델에 L1, L2 규제 적용

Out :



• Learning rate

> 가중치 업데이트에 학습률을 넣어 학습효과 상승

```
class Learning(DualLayer):
    def __init__(self, units = 10, learning_rate = 0.1):
        self.units = units
        self.w1 = None
        self.b1 = None
        self.w2 = None
        self.b2 = None
        self.a1 = None
        self.lr = learning_rate
        self.w_history = []
        self.losses = []
```

• Learning rate

> 가중치 업데이트에 학습률을 넣어 학습효과 상승

```
def training(self, x, y, m):
    z = self.forpass(x)
    a = self.activation(z)
    err = -(y - a)
    w1_grad, b1_grad, w2_grad, b2_grad = self.backprop(x, err)

    self.w1 -= self.lr * w1_grad
    self.b1 -= b1_grad
    self.w2 -= self.lr * w2_grad
    self.b2 -= b2_grad

    return a
```

가중치를 학습률만큼 수정함

신경망 최적화

- Learning rate

> 가중치 업데이트에 학습률을 넣어 학습효과 상승

```
def fit(self, x, y, epochs = 100):
    y = y.reshape(-1,1)
    m = len(x)
    self.init_weights(x.shape[1])
    self.w_history.append([self.w1.copy(), self.w2.copy()])

    for i in range(epochs):
        a = self.training(x, y, m)
        a = np.clip(a, 1e-10, 1-1e-10)
        loss = np.sum(-(y * np.log(a) + (1 - y) * np.log(1 - a)))
        self.losses.append(loss / m)
        self.w_history.append([self.w1.copy(), self.w2.copy()])
```

• Learning rate

정확도 그래프

> 가중치 업데이트에 학습률을 넣어 학습효과 상승

```
learning = Learning(learning_rate=0.1)
learning.fit(X_train_scaled, y_train, epochs = 1000)
print(learning.score(X_test_scaled, y_test))

w1 = []
for w in learning.w_history:
    w1.append(w[0])
for i,w in enumerate(w1):
    plt.plot(i,w[0,0],'o')
```

10000으로 바꾸면 정확도 매우 높아짐

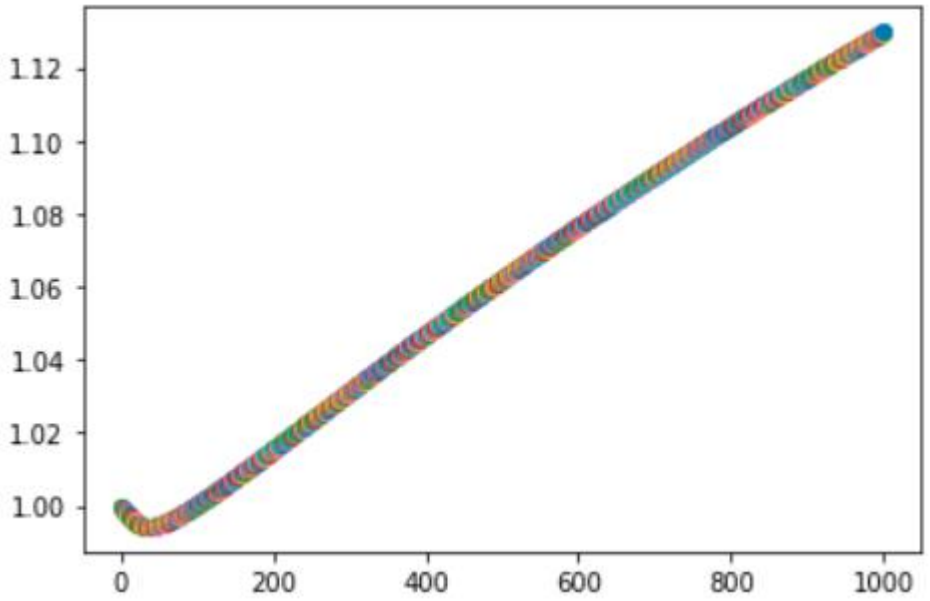
같은 에폭스 조건이면 오히려 정확도 떨어짐
너무 적게 움직이므로 최소값까지 가지 못함
그러므로 에폭스 횟수 높여줘야 함

Out : 0.8771929824561403

- Learning rate

> 가중치 업데이트에 학습률을 넣어 학습효과 상승

Out :



• Learning rate

> 가중치 업데이트에 학습률을 넣어 학습효과 상승

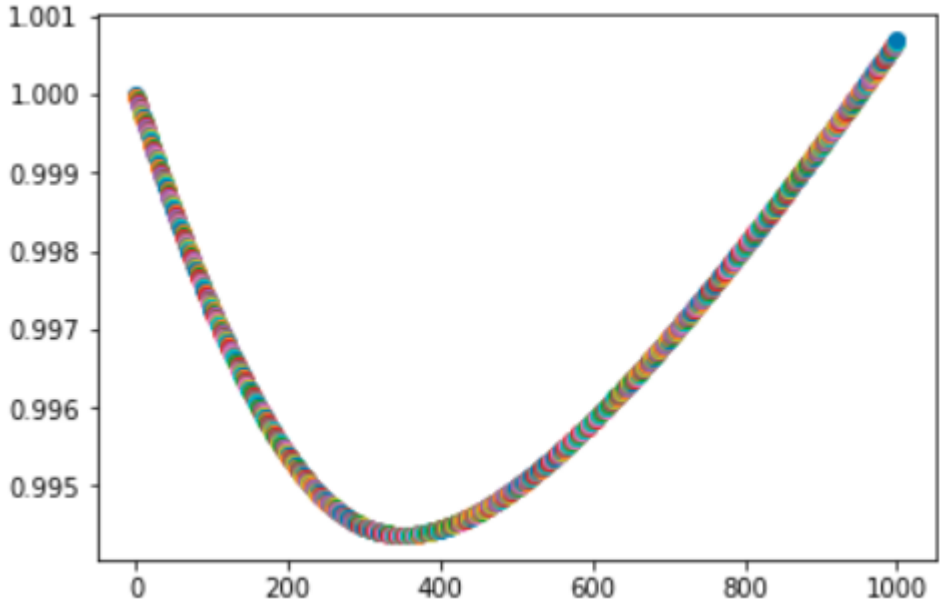
```
learning = Learning(learning_rate=0.01)
learning.fit(X_train_scaled, y_train, epochs = 1000)
print(learning.score(X_test_scaled, y_test))


w1 = []
for w in learning.w_history:        # 모든 가중치 w값 저장됨
    w1.append(w[0])
for i,w in enumerate(w1):
    plt.plot(i,w[0,0],'o')                ##첫번째 w값의 첫번째 값...첫.
```

Out : 0.8859649122807017

- Learning rate

> 가중치 업데이트에 학습률을 넣어 학습효과 상승

Out :



신경망 최적화

• 검증 데이터를 통한 학습 진행

> 검증 데이터를 입력받고 Loss를 기록 : 검증데이터는 학습하지 않은 데이터
train데이터는 학습

```
class Validation(DualLayer):
    def __init__(self, units = 10):
        self.units = units
        self.w1 = None
        self.b1 = None
        self.w2 = None
        self.b2 = None
        self.a1 = None
        self.losses = []
        self.val_losses = []
```

원래는 데이터 하나하나씩 for로 계산했으나
묶음으로 계산함,

• 검증 데이터를 통한 학습 진행

> 검증 데이터를 입력받고 Loss를 기록

```
def fit(self, x, y, epochs = 100, x_val = None, y_val = None):
    y = y.reshape(-1,1)
    y_val = y_val.reshape(-1, 1)
    m = len(x)
    self.init_weights(x.shape[1])

    for i in range(epochs):
        a = self.training(x, y, m)
        a = np.clip(a, 1e-10, 1-1e-10)
        loss = np.sum(-(y * np.log(a) + (1 - y) * np.log(1 - a)))
        self.losses.append(loss / m)
        self.update_val_loss(x_val, y_val)
```

• 검증 데이터를 통한 학습 진행

> 검증 데이터를 입력받고 Loss를 기록

```
def update_val_loss(self, x_val, y_val):
    if x_val is None:
        return
    val_loss = 0
    z = self.forpass(x_val)
    a = self.activation(z)
    a = np.clip(a, 1e-10, 1-1e-10)
    val_loss = np.sum(-(y_val * np.log(a) + (1 - y_val) * np.log(1 - a)))
    self.val_losses.append(val_loss / len(x_val))
```

• 검증 데이터를 통한 학습 진행

> 검증 데이터를 입력받고 Loss를 기록

```
from sklearn.model_selection import train_test_split

X_tr, X_val, y_tr, y_val = train_test_split(X_train_scaled, y_train,
                                            test_size = 0.2, random_state = 0)

validation = Validation()
validation.fit(X_tr, y_tr, epochs = 1000, x_val=X_val, y_val=y_val)
print(validation.score(X_test_scaled, y_test))
```

Out : 0.956140350877193

• 검증 데이터를 통한 학습 진행

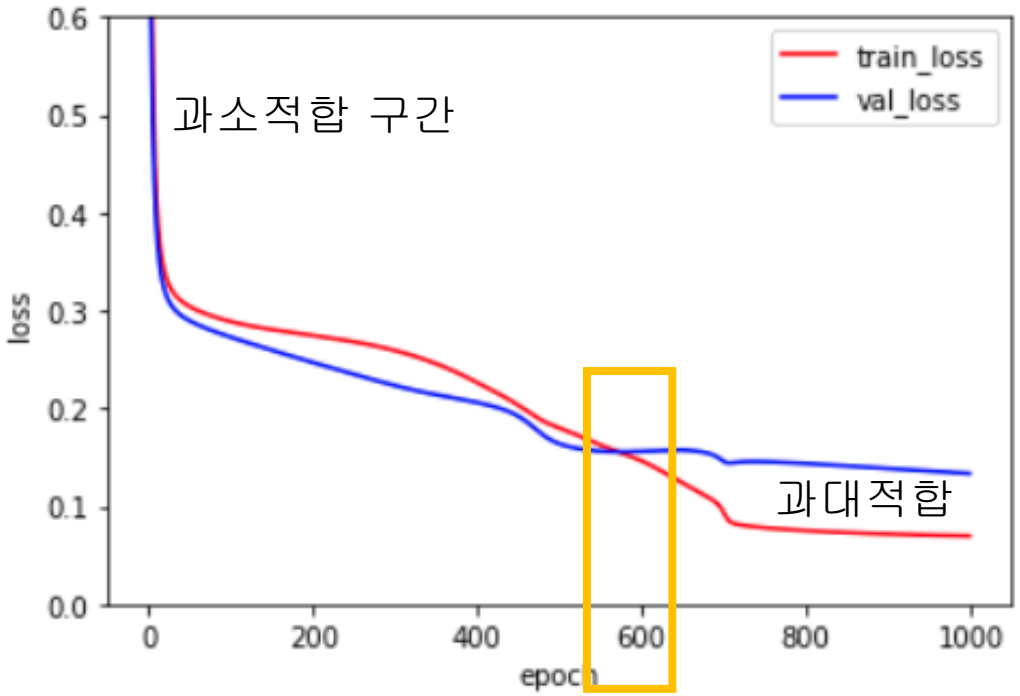
> 검증 데이터를 입력받고 Loss를 기록

```
import matplotlib.pyplot as plt


plt.ylim(0, 0.6)
plt.plot(validation.losses, 'r', label='train_loss')
plt.plot(validation.val_losses, 'b', label='val_loss')
plt.ylabel('loss')        검증 데이터의 손실률
plt.xlabel('epoch')
plt.legend()
plt.show()
```

• 검증 데이터를 통한 학습 진행

> 검증 데이터를 입력받고 Loss를 기록

Out :



< 학습이 잘 되고 있는지 확인하는 방법>
* train과 검증데이터가 교차하는 구간이 과적합이 의심됨
* 검증 데이터는 학습을 하면 할수록 증가함 --> 과적합이라고 추측

학습이 잘 되고 있는지 확인할 수 있는 하나의 기준이 될 수 있음

• 검증 데이터를 통한 학습 진행

> 검증 데이터를 입력받고 Loss를 기록

```
validation = Validation()
validation.fit(X_tr, y_tr, epochs = 500, x_val=X_val, y_val=y_val)
print(validation.score(X_test_scaled, y_test))

validation = Validation()
validation.fit(X_tr, y_tr, epochs = 600, x_val=X_val, y_val=y_val)
print(validation.score(X_test_scaled, y_test))

validation = Validation()
validation.fit(X_tr, y_tr, epochs = 700, x_val=X_val, y_val=y_val)
print(validation.score(X_test_scaled, y_test))
```

Out :

```
0.9298245614035088
0.9473684210526315
0.9298245614035088
```

신경망 최적화

- 미니 배치 경사 하강법

> 데이터를 나누어 학습하고 W, b를 업데이트

```
class Minibatch(DualLayer):
    def __init__(self, units = 10, batch_size = 32):
        super().__init__(units)
        self.batch_size = batch_size
```

보통 2의 제곱수

super 모 class의 기존 init 전체 그대로 가져옴
units 만 설정해야하므로 가져옴

원래는 데이터 하나하나씩 for로 계산하거나
한꺼번에 계산했으나
미니 배치경사하강법은 전체를 묶음으로 나눠서 묶음별로 계산함,

• 미니 배치 경사 하강법

> 데이터를 나누어 학습하고 W, b를 업데이트

```
    def fit(self, x, y, epochs = 100):
        self.init_weights(x.shape[1])

        for i in range(epochs):
            loss = 0
            for x_batch, y_batch in self.gen_batch(x, y):
                y_batch = y_batch.reshape(-1, 1)
                m = len(x_batch)
                a = self.training(x_batch, y_batch, m)
                a = np.clip(a, 1e-10, 1-1e-10)
                loss += np.sum(-(y_batch * np.log(a) +
                                 (1 - y_batch) * np.log(1 - a)))
            self.losses.append(loss / len(x))
```

• 미니 배치 경사 하강법

> 데이터를 나누어 학습하고 W, b를 업데이트

```
def gen_batch(self, x, y):  나누는 함수
    length = len(x)
    bins = length // self.batch_size
    if length % self.batch_size:
        bins += 1
    indexes = np.random.permutation(np.arange(len(x)))
    x = x[indexes]          랜덤하게 섞는 함수
    y = y[indexes]
    for i in range(bins):
        start = self.batch_size * i
        end = self.batch_size * (i + 1)
        yield x[start:end], y[start:end]
```

[0,10]

값들을 묶어서 리턴하는 역할 : 나머지값도 가져옴

• 미니 배치 경사 하강법

> 데이터를 나누어 학습하고 W, b를 업데이트

```
dual_layer = DualLayer()
dual_layer.fit(X_train_scaled, y_train, epochs = 1000)
print(dual_layer.score(X_test_scaled, y_test))

minibatch = Minibatch()      Minibatch(batch_size=5)
minibatch.fit(X_train_scaled, y_train, epochs = 1000)
print(minibatch.score(X_test_scaled, y_test))
```

Out : 0.956140350877193
0.956140350877193

배치사이즈가 적을수록/ 에포크가 클 수록, 속도 늦고 과적합
: 정확도이 더 좋아지지만 속도는

• 미니 배치 경사 하강법

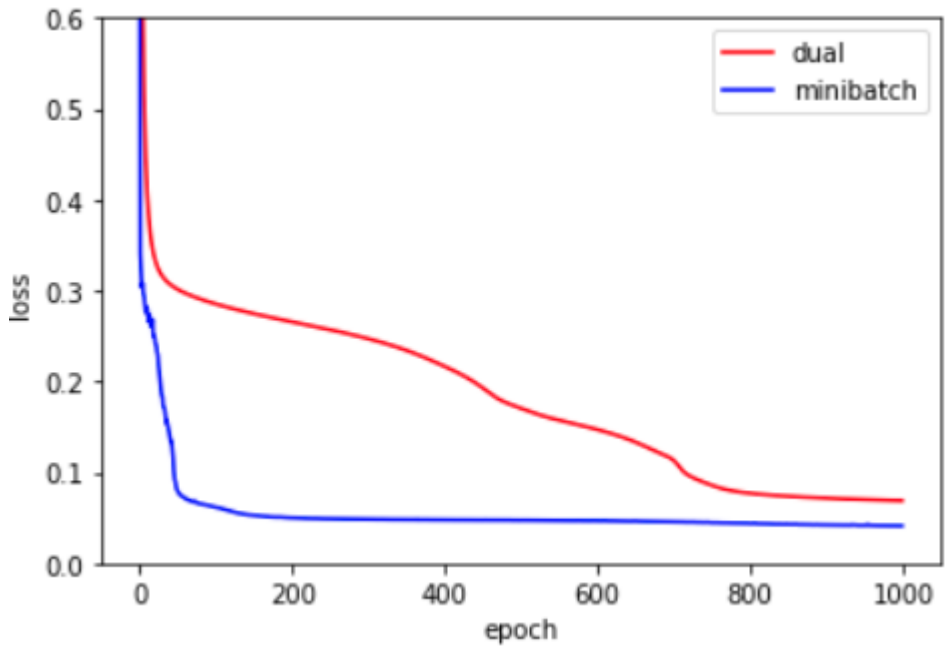
> 데이터를 나누어 학습하고 W, b를 업데이트

```
plt.plot(dual_layer.losses, 'r', label = 'dual')
plt.plot(minibatch.losses, 'b', label = 'minibatch')
plt.ylim(0,0.6)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```

• 미니 배치 경사 하강법

> 데이터를 나누어 학습하고 W, b를 업데이트

Out :

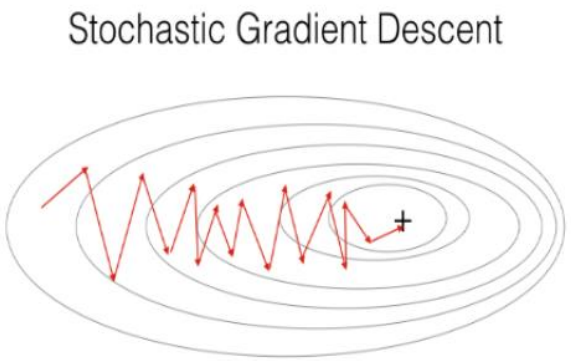


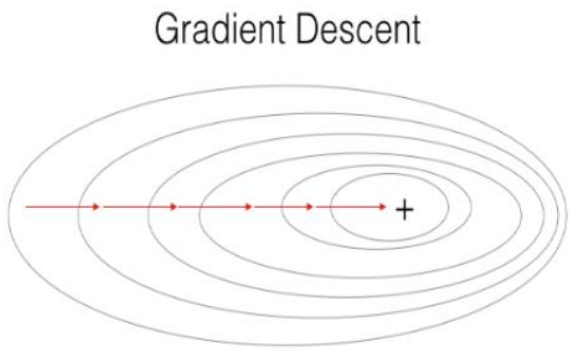
-배치사이즈가 적을수록 정확도 높은 편
-값이 왔다갔다하는 오차가 종종 튄다.
-정확도보다는 속도를 줄이기위해 사용

• 미니 배치 경사 하강법

> 배치 경사 하강법
  - 모든 데이터를 가지고 경사 하강법 진행
  - 데이터를 전부 사용하여 오차를 줄이기에 학습 한번에 오랜 시간이 걸림

> 확률적 경사 하강법
  - 랜덤한 하나의 데이터만을 가지고 경사 하강법 진행
  - 하나의 데이터로 오차를 줄이기에 속도가 빠름
  - 단, 불안정하고 학습 횟수가 증가할 수 있다.



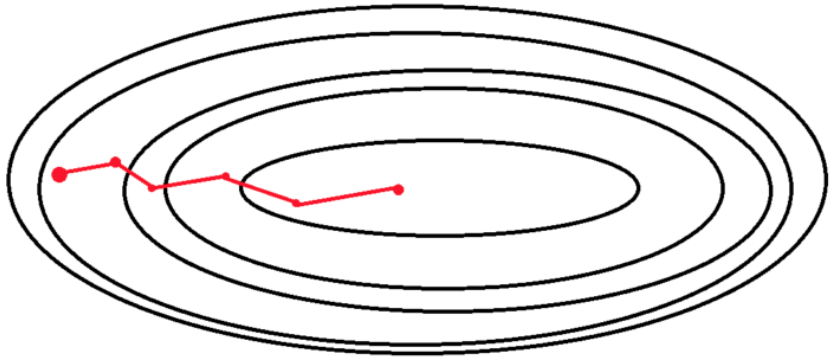


배치 경사하강법: 중심부에서 왔다갔다하면서
오차 최소값으로 가지못하는 경우도 있음

• 미니 배치 경사 하강법

> 미니 배치 경사 하강법

- 배치 경사 하강법과 확률적 경사 하강법을 모두 보완하는 경사 하강법
- 일정 개수의 데이터를 가지고 가중치를 줄임
- 적은 데이터를 사용하여 오차를 줄이기에 학습 속도가 빠름
- epoch 1회에 모든 데이터를 가지고 학습을 진행하여 불안정한 학습을 보완할 수 있다.



• 드롭 아웃

> 드롭 아웃
  - 뉴런의 연결을 랜덤하게 삭제하여 과적합을 막는 기법 :오직 과적합을 막기위해 사용됨

```
class Dropout(DualLayer):
    def __init__(self, units = 10, dropout = 0.5):
        super().__init__(units)      out 계층들에 있는 가중치들을 일부 랜덤으로 지움
        self.dropout = dropout

    def drop(self, z):
        if self.dropout == 1:
            return np.zeros_like(z)
        mask = np.random.uniform(0, 1, z.shape) > self.dropout    핵심 코드
        return (mask * z) / (1.0 - self.dropout)
```

• 드롭 아웃

> 드롭 아웃
  - 뉴런의 연결을 랜덤하게 삭제하여 과적합을 막는 기법

```
def training(self, x, y, m):
    z = self.forpass(x)
    z = self.drop(z)  # 드롭 아웃 적용   # 입력 > 은닉 > z
                                          # z > 활성화 >  a
    a = self.activation(z)                #  a > 은닉 > z
    err = -(y - a)                        #  z>마지막 활성화 >z
    w1_grad, b1_grad, w2_grad, b2_grad = self.backprop(x, err)
    self.w1 -= w1_grad
    self.b1 -= b1_grad
    self.w2 -= w2_grad
    self.b2 -= b2_grad


    return a
```

• 드롭 아웃

드롭아웃 코드는 아무데나 넣어도 됨
z값 지우고 끼워 넣음

> 드롭 아웃
  - 뉴런의 연결을 랜덤하게 삭제하여 과적합을 막는 기법

```
dual_layer = DualLayer()
dual_layer.fit(X_train_scaled, y_train, epochs = 1000)
print(dual_layer.score(X_test_scaled, y_test))


dropout = Dropout()
dropout.fit(X_train_scaled, y_train, epochs = 1000)
print(dropout.score(X_test_scaled, y_test))
```

Out : 0.956140350877193

0.9473684210526315

• 드롭 아웃

> 드롭 아웃
  - 뉴런의 연결을 랜덤하게 삭제하여 과적합을 막는 기법

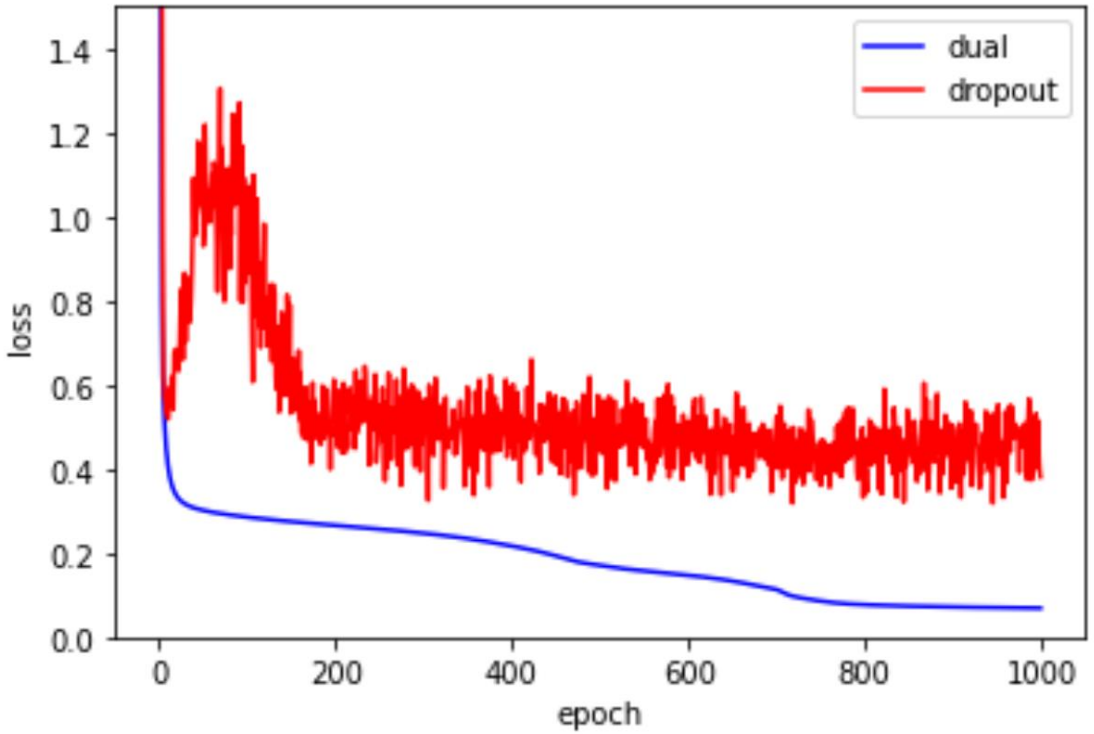
```
plt.plot(dual_layer.losses, 'b', label = 'dual')
plt.plot(dropout.losses, 'r', label = 'dropout')
plt.ylim(0,1.5)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```

• 드롭 아웃

드롭 아웃과 규제는 과적합 방지 코드
크면클수록 정확도 떨어짐

> 드롭 아웃
  - 뉴런의 연결을 랜덤하게 삭제하여 과적합을 막는 기법

Out :



신경망 최적화

• 조기종료(earlystop)

> 조기종료

- 검증 데이터를 사용하여 학습되지 않은 데이터의 loss가 더 이상 줄어들지 않으면 학습을 종료시키는 기법 즉, 과적합이 일어날때

```
class Early_stop(Validation):
    def __init__(self, units = 10, earlystop = 10):
        super().__init__(units)
        self.minloss = None
        self.earlystop = earlystop
```

최소 구간

최솟값(검정데이터)

• 조기종료(earlystop)

> 조기종료

```
def fit(self, x, y, epochs = 100, x_val = None, y_val = None):
    y = y.reshape(-1,1)
    y_val = y_val.reshape(-1, 1)
    m = len(x)
    self.init_weights(x.shape[1])
    for i in range(epochs):
        a = self.training(x, y, m)
        a = np.clip(a, 1e-10, 1-1e-10)
        loss = np.sum(-(y * np.log(a) + (1 - y) * np.log(1 - a)))
```

• 조기종료(earlystop)

> 조기종료

```
            self.losses.append(loss / m)
            self.update_val_loss(x_val, y_val)
            if i == (self.earlystop - 1):
                self.minloss = min(self.val_losses)
            if i >= self.earlystop:
                if self.minloss < min(self.val_losses[-self.earlystop:]):    최솟값이 갱신되었다면
                    print(f'EarlyStop : Epochs {i}')
                    break
                else:
                    self.minloss = min(self.val_losses)
```

최소값

• 조기종료(earlystop)

> 조기종료

```
early_stop =  Early_stop(earlystop = 10)
early_stop.fit(X_tr, y_tr, epochs = 1000, x_val=X_val, y_val=y_val)
print(early_stop.score(X_test_scaled, y_test))
```

Out : EarlyStop : Epochs 585

0.947368421052631 5

• 조기종료(earlystop)

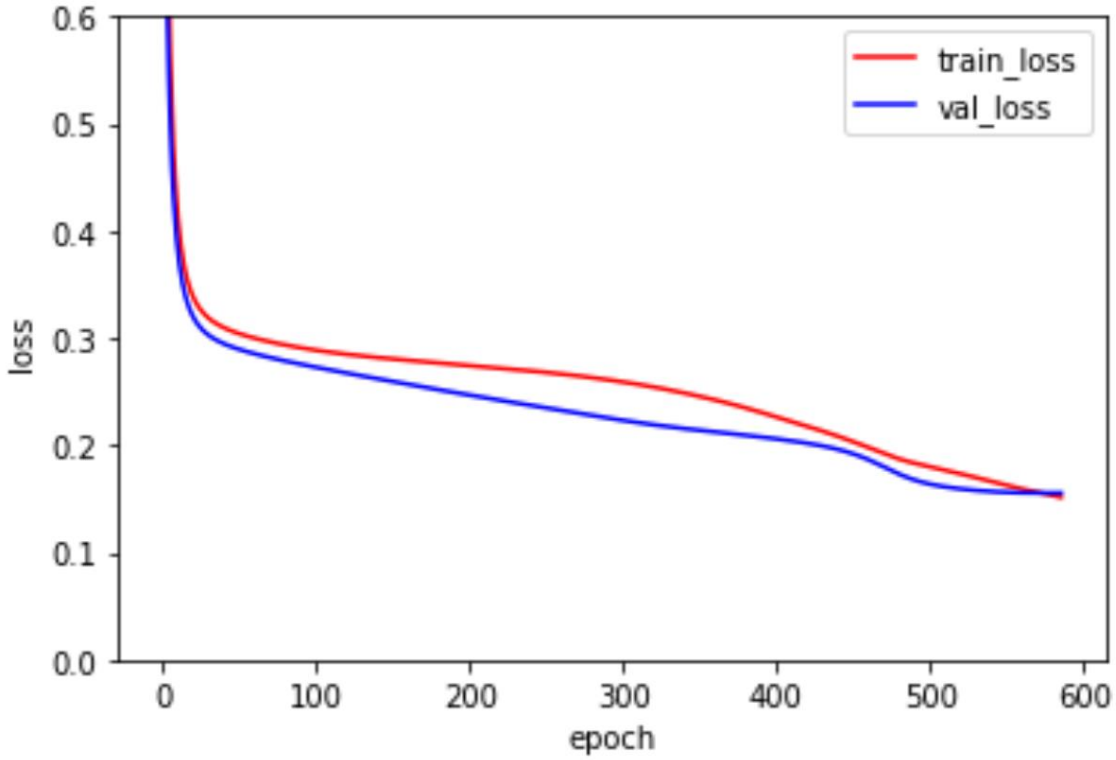
> 조기종료

```
plt.plot(early_stop.losses, 'r', label='train_loss')
plt.plot(early_stop.val_losses, 'b', label='val_loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.ylim(0, 0.6)
plt.legend()
plt.show()
```

• 조기종료(earlystop)

> 조기종료 : 케라스 등에서는 조기종료시
기중치 등 정보 자동저장해놓음

Out :



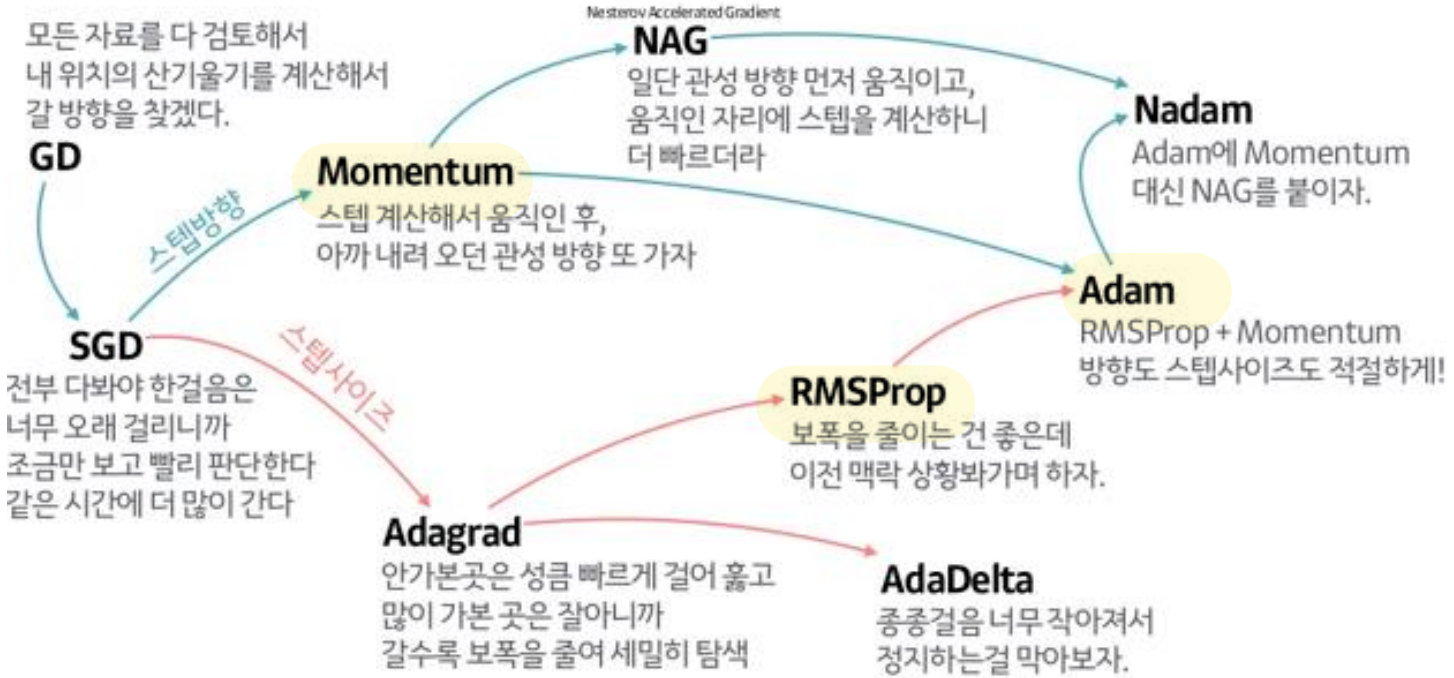
• 옵티마이저(Optimizer)

> 옵티마이저란?

- 경사 하강법처럼 실제 값과 예측 값의 차이만큼 가중치를 주어 오차를 줄이는 방법들을 말한다.
- 딥러닝에는 다양한 옵티마이저 방법들이 존재한다.
- 크게는 SGD(확률적 경사 하강법)에서 변형된 종류가 있다.

• 옵티마이저(Optimizer)

> 옵티마이저 종류



• Momentum

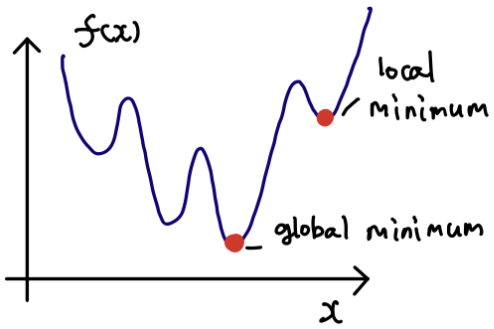
> Momuntum이란

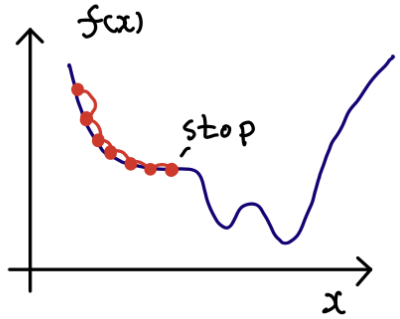
- 물리학 용어로 동력을 의미하며, 추진력, 여세, 타성 등 물체가 한 방향으로 지속적으로 변화하려는 경향을 의미한다.
- 수식은 다음과 같다.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$

- 다음과 같은 손실함수가 존재한다면 SGD의 단점을 보안한다.





• Momentum

> Momentum 구현

```
class Momentum(DualLayer):
    def __init__(self, units = 10, learning_rate = 0.01, momentum = 0.9):
        super().__init__(units)
        self.lr = learning_rate
        self.momentum = momentum
        self.v1 = None
        self.v2 = None
```

드롭아웃 코드는 아무데나 넣어도 됨
z값 지우고 끼워 넣음

• Momentum

> Momentum 구현

```
def training(self, x, y, m):
    z = self.forpass(x)
    a = self.activation(z)
    err = -(y - a)
    w1_grad, b1_grad, w2_grad, b2_grad = self.backprop(x, err)
    if self.v1 is None:
        self.v1 = np.zeros_like(self.w1)
        self.v2 = np.zeros_like(self.w2)
```

• Momentum

> Momentum 구현

```
        self.v1 = - self.lr * w1_grad + self.momentum * self.v1
        self.v2 = - self.lr * w2_grad + self.momentum * self.v2
        self.w1 -= self.lr * w1_grad - self.momentum * self.v1
        self.b1 -= b1_grad
        self.w2 -= self.lr * w2_grad - self.momentum * self.v2
        self.b2 -= b2_grad
        return a
```

• Momentum

> Momentum 구현

```
dual_layer = DualLayer()
dual_layer.fit(X_train_scaled, y_train, epochs = 10000)
print(dual_layer.score(X_test_scaled, y_test))

momentum = Momentum()
momentum.fit(X_train_scaled, y_train, epochs = 10000)
print(momentum.score(X_test_scaled, y_test))
```

Out : 0.956140350877193

0.9649122807017544

• Momentum

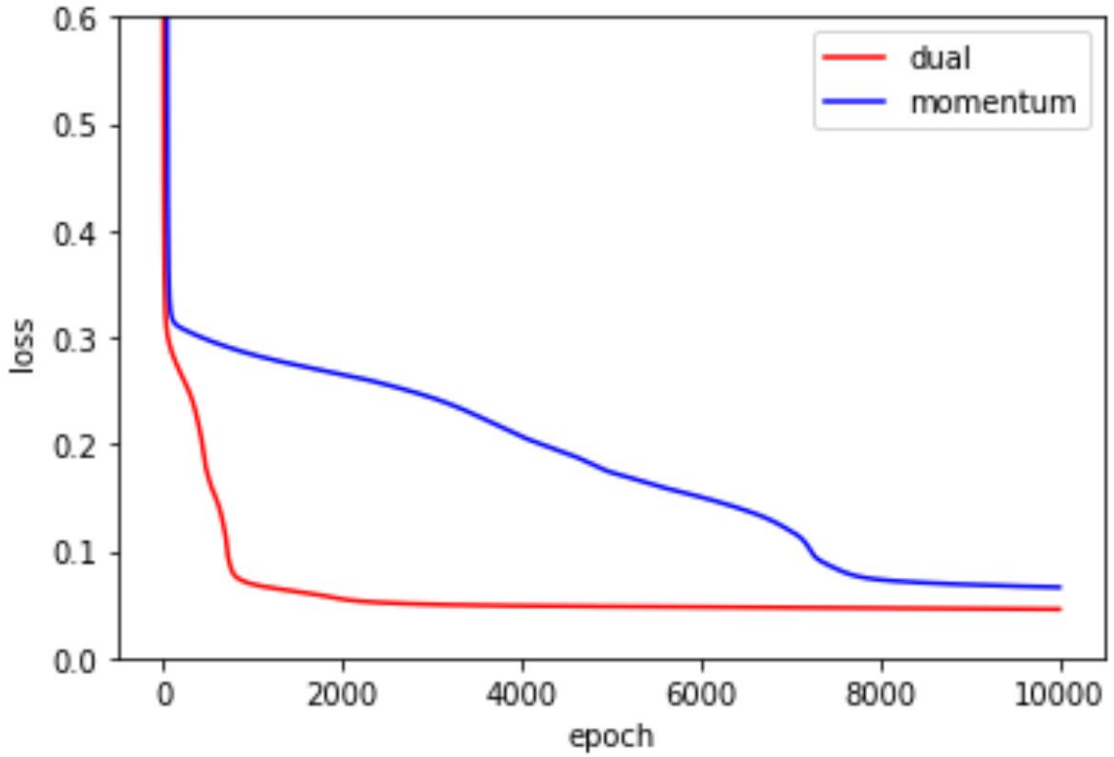
> Momentum 구현

```
plt.plot(dual_layer.losses, 'r', label = 'dual')
plt.plot(momentum.losses, 'b', label = 'momentum')
plt.ylim(0,0.6)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```

• Momentum

> Momentum 구현

Out :



• RMSProp

> RMSProp이란

- Adagrad를 개선한 방법이다.
- 학습률이 계속해서 작아지는 것을 방지하기 위해 지수이동평균을 적용하여 학습의 최소 Step은 유지할 수 있다.
- 수식은 다음과 같다.

$$G_t = \gamma G_{t-1} + (1-\gamma)(\nabla_\omega J(\omega_t))^2$$

$$\omega_{t+1} = \omega_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot \nabla_\omega J(\omega_t)$$

• RMSProp

> RMSProp 구현

```
class RMSprop(DualLayer):
    def __init__(self, units = 10, learning_rate = 0.1, decay_rate = 0.99):
        super().__init__(units)
        self.lr = learning_rate
        self.decay_rate = decay_rate
        self.h1 = None
        self.h2 = None
        self.epsilon = 1e-6
```

• RMSProp

> RMSProp 구현

```
def training(self, x, y, m):
    z = self.forpass(x)
    a = self.activation(z)
    err = -(y - a)
    w1_grad, b1_grad, w2_grad, b2_grad = self.backprop(x, err)
    if self.h1 is None:
        self.h1 = np.zeros_like(self.w1)
        self.h2 = np.zeros_like(self.w2)
```

• RMSProp

> RMSProp 구현

```
        self.h1 *= self.decay_rate
        self.h2 *= self.decay_rate
        self.h1 += w1_grad * w1_grad * (1-self.decay_rate)
        self.h2 += w2_grad * w2_grad * (1-self.decay_rate)
        self.w1 -= self.lr * w1_grad / (np.sqrt(self.h1 + self.epsilon))
        self.b1 -= b1_grad
        self.w2 -= self.lr * w2_grad / (np.sqrt(self.h2 + self.epsilon))
        self.b2 -= b2_grad

        return a
```

• RMSProp

> RMSProp 구현

```
dual_layer = DualLayer()
dual_layer.fit(X_train_scaled, y_train, epochs = 15000)
print(dual_layer.score(X_test_scaled, y_test))

adagrad = Adagrad()
adagrad.fit(X_train_scaled, y_train, epochs = 15000)
print(adagrad.score(X_test_scaled, y_test))
```

Out : 0.956140350877193

0.956140350877193

• RMSProp

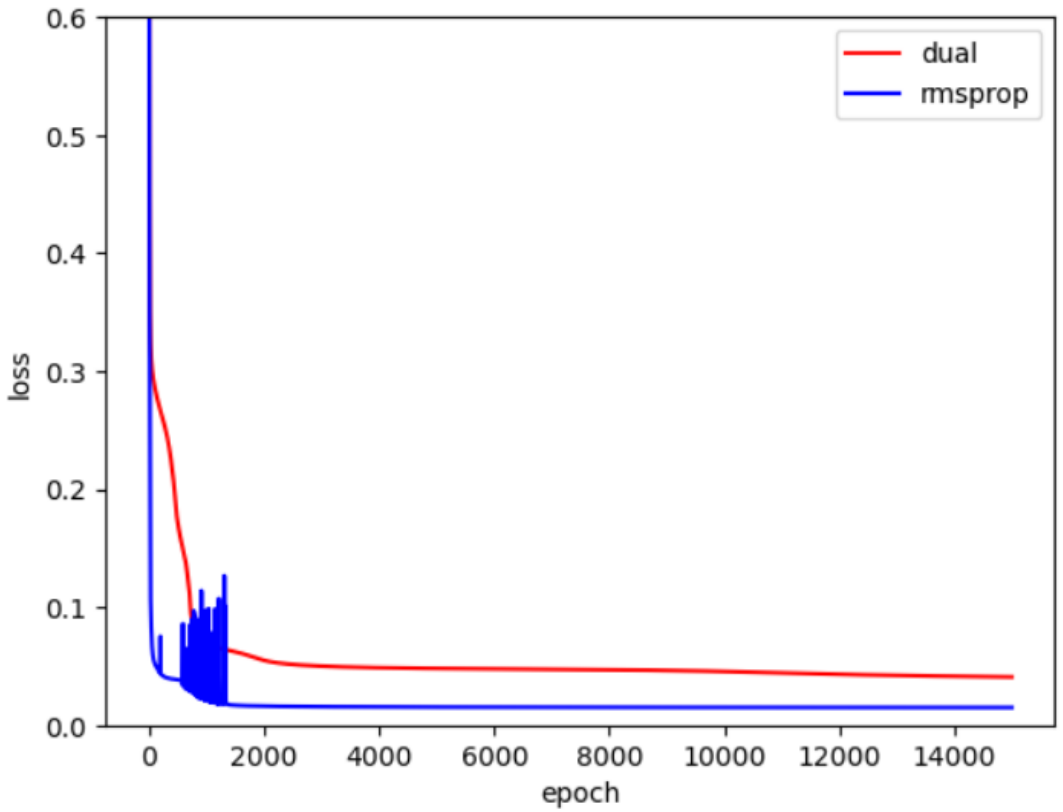
> RMSProp 구현

```
plt.plot(dual_layer.losses, 'r', label = 'dual')
plt.plot(rmsprop.losses, 'b', label = 'rmsprop')
plt.ylim(0,0.6)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```

• RMSProp

> RMSProp 구현

Out :



• Adam(Adaptive Moment Estimation)

> Adam이란

- RMSProp과 Momentum 기법을 합쳐 관성계수 m과 계산된 v로 지수이동평균을 활용하여 Step을 조절하는 방식이다.
- 수식은 다음과 같다.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_\omega J(\omega_t)$$

$$v_t = \beta_2 m_{t-1} + (1 - \beta_2) (\nabla_\omega J(\omega_t))^2$$

$$\omega_{t+1} = \omega_t - m_t \frac{\eta}{\sqrt{v_t + \epsilon}}$$

• Adam(Adaptive Moment Estimation)

> Adam 구현

```
class Adam(DualLayer):
    def __init__(self, units = 10, learning_rate = 0.01, beta1 = 0.9, beta2 = 0.999):
        super().__init__(units)
        self.lr = learning_rate
        self.beta1 = beta1
        self.beta2 = beta2
        self.m1 = None
        self.m2 = None
        self.v1 = None
        self.v2 = None
        self.iter = 0
        self.epsilon = 1e-6
```

• Adam(Adaptive Moment Estimation)

> Adam 구현

```
def training(self, x, y, m):
    z = self.forpass(x)
    a = self.activation(z)
    err = -(y - a)
    w1_grad, b1_grad, w2_grad, b2_grad = self.backprop(x, err)
    if self.m1 is None:
        self.m1 = np.zeros_like(self.w1)
        self.m2 = np.zeros_like(self.w2)
        self.v1 = np.zeros_like(self.w1)
        self.v2 = np.zeros_like(self.w2)
    self.m1 *= self.beta1
    self.v1 *= self.beta2
```

• Adam(Adaptive Moment Estimation)

> Adam 구현

```
        self.m1 += ((1 - self.beta1) * w1_grad) / (1 - self.beta1)
        self.v1 += ((1 - self.beta2) * w1_grad * w1_grad) / (1 - self.beta2)
        self.m2 *= self.beta1
        self.v2 *= self.beta2
        self.m2 += ((1 - self.beta1) * w2_grad) / (1 - self.beta1)
        self.v2 += ((1 - self.beta2) * w2_grad * w2_grad) / (1 - self.beta2)
        self.w1 -= self.lr * self.m1 / (np.sqrt(self.v1) + self.epsilon)
        self.b1 -= b1_grad
        self.w2 -= self.lr * self.m2 / (np.sqrt(self.v2) + self.epsilon)
        self.b2 -= b2_grad


        return a
```

• Adam(Adaptive Moment Estimation)

> Adam 구현

```
dual_layer = DualLayer()
dual_layer.fit(X_train_scaled, y_train, epochs = 10000)
print(dual_layer.score(X_test_scaled, y_test))

adam = Adam()
adam.fit(X_train_scaled, y_train, epochs = 10000)
print(adam.score(X_test_scaled, y_test))
```

Out : 0.956140350877193

0.9649122807017544

• Adam(Adaptive Moment Estimation)

> Adam 구현

```
plt.plot(dual_layer.losses, 'r', label = 'dual')
plt.plot(adam.losses, 'b', label = 'Adam')
plt.ylim(0,0.6)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```

• Adam(Adaptive Moment Estimation)

> Adam 구현

Out :