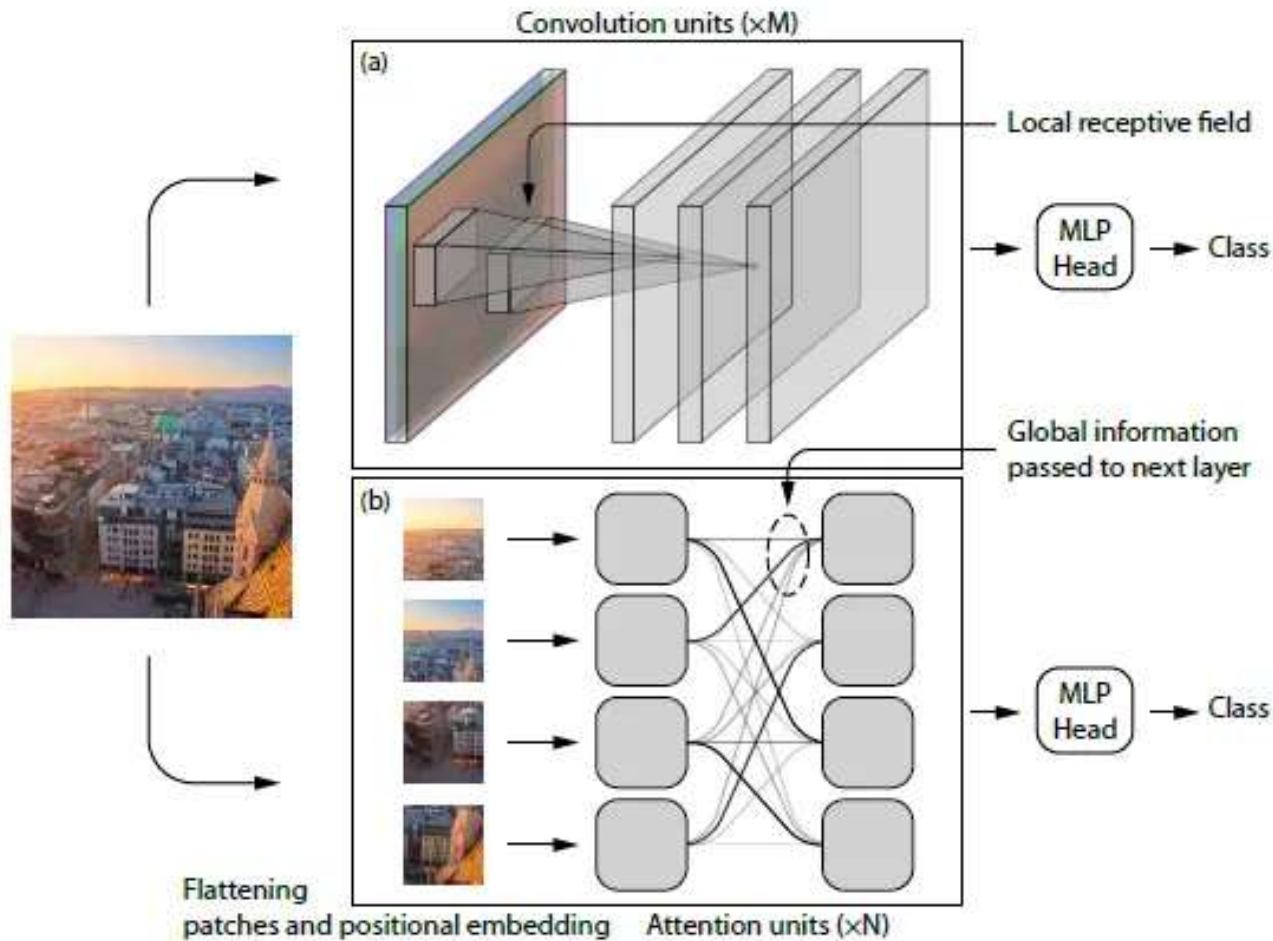


MLP와 케라스 라이브러리



Role of MLP → Top of all models





학습 목표

- 미니 배치의 개념을 이해한다.
- 학습률의 개념을 이해한다.
- 케라스 라이브러리로 MLP를 구현해본다.
- 케라스 라이브러리를 살펴본다.
- 하이퍼 매개변수에 대하여 살펴본다





몇 개의 샘플을 처리한 후에 가중치를 변경할 것인가?

- 일반적으로는 훈련 샘플의 개수는 아주 많다.

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

60000개의 훈련 샘플과
10000개의 테스트 샘플이
저장되어 있습니다.





가중치를 변경하는 2가지의 방법

- 풀 배치 학습(full batch learning):
- 온라인 학습(online learning) 또는 확률적 경사 하강법(Stochastic Gradient Descent: **SGD**) :

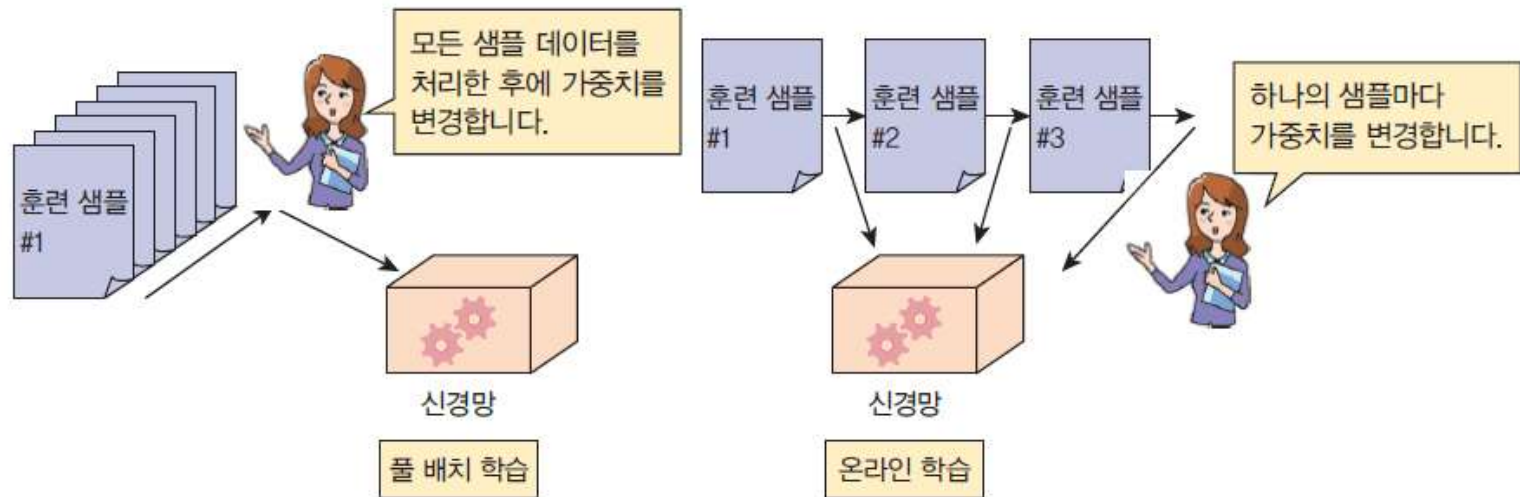


그림 7-1 배치 학습과 온라인 학습



풀 배치 학습

풀 배치 학습

1. 가중치와 바이어스를 0부터 1 사이의 난수로 초기화한다.
2. 수렴할 때까지 모든 가중치에 대하여 다음을 반복한다.
3. 모든 훈련 샘플을 처리하여 평균 그래디언트 $\frac{\partial E}{\partial w} = \frac{1}{N} \sum_{k=1}^N \frac{\partial E_k}{\partial w}$ 을 계산한다.
4. $w \leftarrow w - \eta * \frac{\partial E}{\partial w}$

계산 시간이 많이 걸리고
늦게 수렴할 수 있다.



온라인 학습 (SGD: 확률적 경사 하강법)

확률적 경사 하강법

1. 가중치와 바이어스를 0부터 1 사이의 난수로 초기화한다.
2. 수렴할 때까지 모든 가중치에 대하여 다음을 반복한다.
3. 훈련 샘플 중에서 무작위로 i 번째 샘플을 선택한다.
4. 그래디언트 $\frac{\partial E}{\partial w}$ 을 계산한다.
5. $w \leftarrow w - \eta * \frac{\partial E}{\partial w}$

계산하기 쉽지만 샘플에 따라서 우왕좌왕하기 쉽다.



미니 배치 학습 (mini-batch)

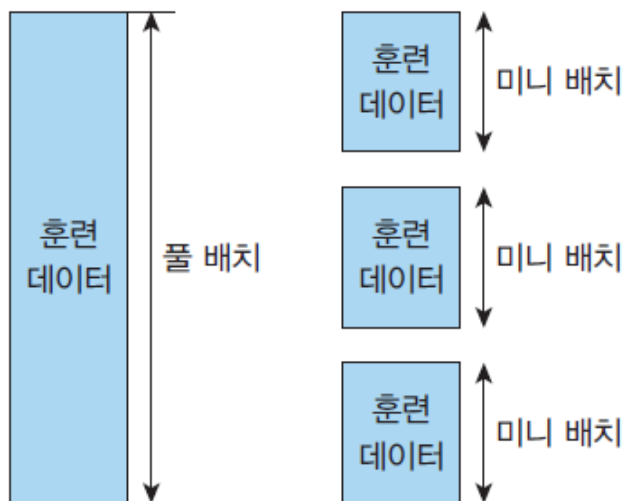
- 온라인 학습과 풀 배치 학습의 중간에 있는 방법

미니 배치 경사 하강법

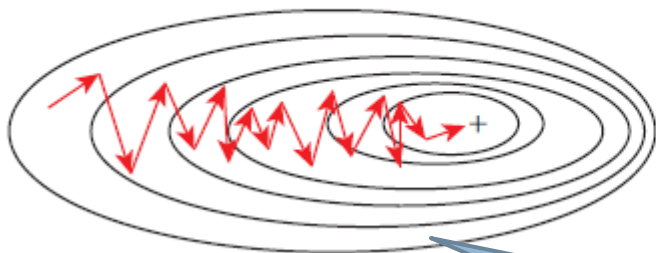
1. 가중치와 바이어스를 0부터 1 사이의 난수로 초기화한다.
2. 수렴할 때까지 모든 가중치에 대하여 다음을 반복한다.
3. 훈련 샘플 중에서 무작위로 B개의 샘플을 선택한다.
4. 그래디언트 $\frac{\partial E}{\partial w} = \frac{1}{B} \sum_{k=1}^B \frac{\partial E_k}{\partial w}$ 을 계산한다.
5. $w \leftarrow w - \eta * \frac{\partial E}{\partial w}$



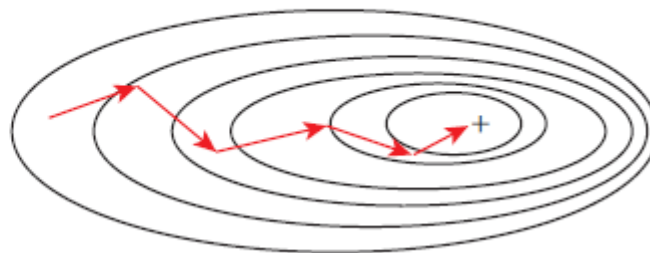
각 방법들의 비교



확률적 경사 하강법(SGD)



미니 배치 경사 하강법



노이즈 샘플에 따라서 많이 흔들린다.



Lab: 미니 배치 실습 #1

```
import numpy as np
import tensorflow as tf
```

mini_batch.py

```
# 데이터를 학습 데이터와 테스트 데이터로 나눈다.
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

```
data_size = x_train.shape[0]
batch_size = 12    # 배치 크기
```

```
selected = np.random.choice(data_size, batch_size)
print(selected)
x_batch = x_train[selected]
y_batch = y_train[selected]
```

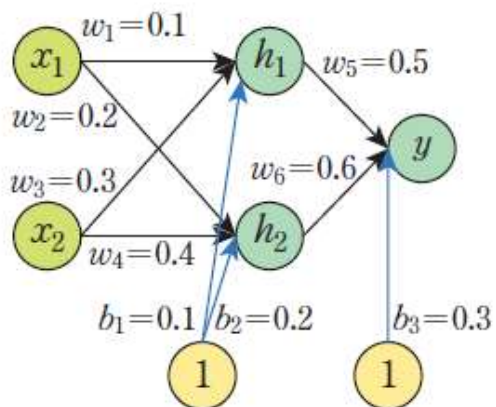
배치 크기만큼 랜덤하게 선택

```
[58298 3085 27743 33570 35343 47286 18267 25804 4632 10890 44164 18822]
```



행렬로 미니 배치 구현하기

0.0
0.0



0.0

$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix} + [b_1 \ b_2]$$

여기에 다른 예제를 붙이면 어떨까?

$$Z_1 = X \times W + B = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ x_1^{(3)} & x_2^{(3)} \\ x_1^{(4)} & x_2^{(4)} \end{bmatrix} \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix} + [b_1 \ b_2]$$



각 샘플별로 출력이 계산된다. → 행렬 연산

$$A_1 = f(Z_1) = \begin{bmatrix} f(z_1^{(1)}) & f(z_2^{(1)}) \\ f(z_1^{(2)}) & f(z_2^{(2)}) \\ f(z_1^{(3)}) & f(z_2^{(3)}) \\ f(z_1^{(4)}) & f(z_2^{(4)}) \end{bmatrix}$$

$$Z_2 = A_1 \times W_2 + B_2 = \begin{bmatrix} h_1^{(1)} & h_2^{(1)} \\ h_1^{(2)} & h_2^{(2)} \\ h_1^{(3)} & h_2^{(3)} \\ h_1^{(4)} & h_2^{(4)} \end{bmatrix} \times \begin{bmatrix} w_5 \\ w_6 \end{bmatrix} + [b_3] = \begin{bmatrix} z_y^{(1)} \\ z_y^{(2)} \\ z_y^{(3)} \\ z_y^{(4)} \end{bmatrix}$$

$$Y = f(Z_2) = \begin{bmatrix} f(z_y^{(1)}) \\ f(z_y^{(2)}) \\ f(z_y^{(3)}) \\ f(z_y^{(4)}) \end{bmatrix} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ y^{(3)} \\ y^{(4)} \end{bmatrix}$$



미니 배치의 구현: XOR (행렬 연산)

```
import numpy as np
```

mlp_mini_XOR.py

```
# 시그모이드 함수
```

```
def actf(x):  
    return 1/(1+np.exp(-x))
```

```
# 시그모이드 함수의 미분치
```

```
def actf_deriv(x):  
    return x*(1-x)
```

```
# 입력유닛의 개수, 은닉유닛의 개수, 출력유닛의 개수
```

```
inputs, hiddens, outputs = 2, 2, 1  
learning_rate = 0.5
```

```
# 훈련 입력과 출력
```

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  
T = np.array([[0], [1], [1], [0]])
```

```
# 가중치를 -1.0에서 1.0 사이의 난수로 초기화한다.
```

```
W1 = 2*np.random.random((inputs, hiddens))-1  
W2 = 2*np.random.random((hiddens, outputs))-1  
B1 = np.zeros(hiddens)  
B2 = np.zeros(outputs)
```



미니 배치의 구현

순방향 전파 계산

def predict(x):

layer0 = x

Z1 = np.dot(layer0, W1)+B1

layer1 = actf(Z1)

Z2 = np.dot(layer1, W2)+B2

layer2 = actf(Z2)

return layer0, layer1, layer2

입력을 layer0에 대입한다.

행렬의 곱을 계산한다.

활성화 함수를 적용한다.

행렬의 곱을 계산한다.

활성화 함수를 적용한다.

역방향 전파 계산

def fit():

global W1, W2, B1, B2

for i in range(60000):

layer0, layer1, layer2 = predict(X)

layer2_error = layer2-T

layer2_delta = layer2_error*actf_deriv(layer2)

layer1_error = np.dot(layer2_delta, W2.T)

layer1_delta = layer1_error*actf_deriv(layer1)

가중치 갱신 (Batch의 평균 기울기로 갱신)

W2 += -learning_rate*np.dot(layer1.T, layer2_delta)/4.0

W1 += -learning_rate*np.dot(layer0.T, layer1_delta)/4.0

B2 += -learning_rate*np.sum(layer2_delta, axis=0)/4.0

B1 += -learning_rate*np.sum(layer1_delta, axis=0)/4.0

바이어스
입력이 1
이므로 단순
히 합만
계산한다.



미니 배치의 구현

```
def test():  
    for x, y in zip(X, T):  
        x = np.reshape(x, (1, -1))  
        layer0, layer1, layer2 = predict(x)  
        print(x, y, layer2)  
  
fit()  
test()
```

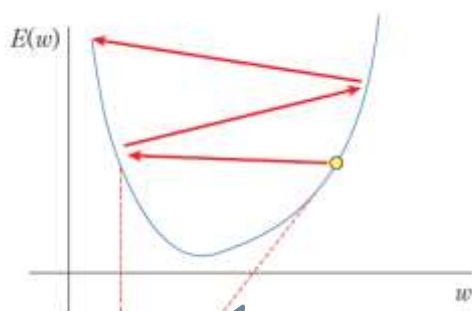
하나여도 2차원 형태이어야 한다.

```
[[0 0]] [0] [[0.0124954]]  
[[0 1]] [1] [[0.98683933]]  
[[1 0]] [1] [[0.9869228]]  
[[1 1]] [0] [[0.01616628]]
```

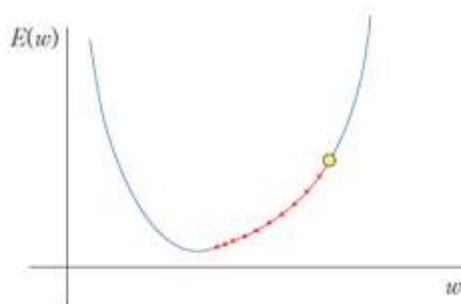


학습률 (lr: learning rate) → 모델 최적화

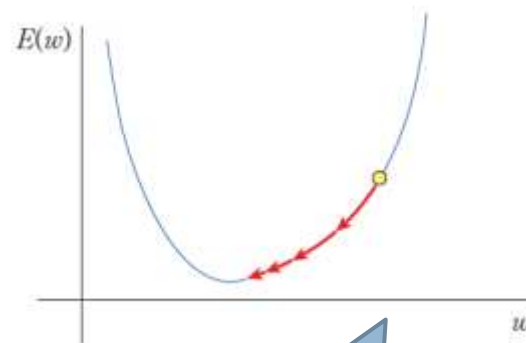
- 학습률이란 한 번에 가중치를 얼마나 변경할 것인가를 나타낸다.
- 학습률이 모델의 성능에 심대한 영향을 끼치지만 설정하기가 아주 어렵다는 것을 발견하였다.



학습률이 너무 큰 경우



학습률이 너무 작은 경우

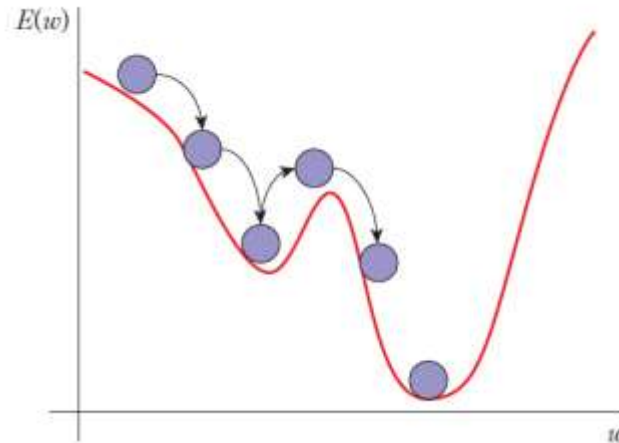


학습률이 적당한 경우



모멘텀 (momentum)

- **모멘텀(momentum)**은 운동량으로 학습 속도를 가속시킬 목적으로 사용한다.(지역 최소에서 벗어나게 **W**를 변경)



$$W_{t+1} = W_t - \eta \frac{\partial E}{\partial W} + \text{momentum} * W_t$$



학습률을 설정하는 방법 - Adagrad

- Adagrad: Adagrad는 **가변 학습률**을 **사용**하는 방법으로 SGD 방법을 개량한 최적화 방법이다. 주된 방법은 **학습률 감소**(learning rate decay)이다. Adagrad는 학습률을 이전 단계의 **기울기들을 누적인 값에 반비례**하여서 설정한다

$$W_{t+1} = W_t - \eta \frac{1}{\sqrt{G_t + \epsilon}} \frac{\partial E}{\partial W}$$

학습률이 점점 줄어든다.



학습 알고리즘 설정하는 방법 - RMSprop

- **RMSprop** : RMSprop은 Adagrad에 대한 수정판이다. Geoffy Hinton이 Coursea 강의 과정에서 제안했다. Adadelata와 유사하지만 한 가지 차이점이 있다. 그래디언트 누적 대신에 **지수 가중 이동 평균**을 사용한다

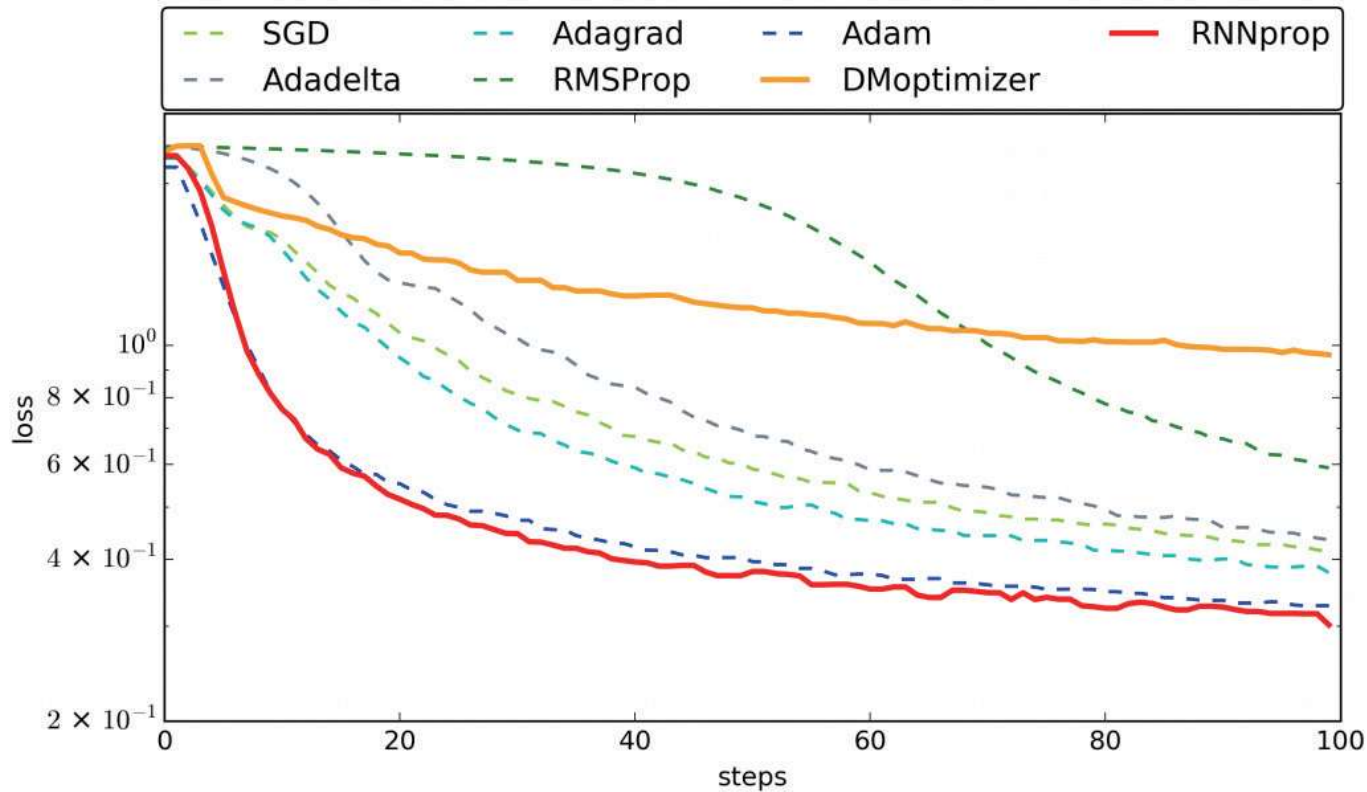
$$v(t) = \rho v(t-1) + (1-\rho) * \left[\frac{\partial E}{\partial W} \right]^2$$

$$W_{t+1} = W_t - \eta \frac{1}{\sqrt{v_t + \epsilon}} \frac{\partial E}{\partial W}$$



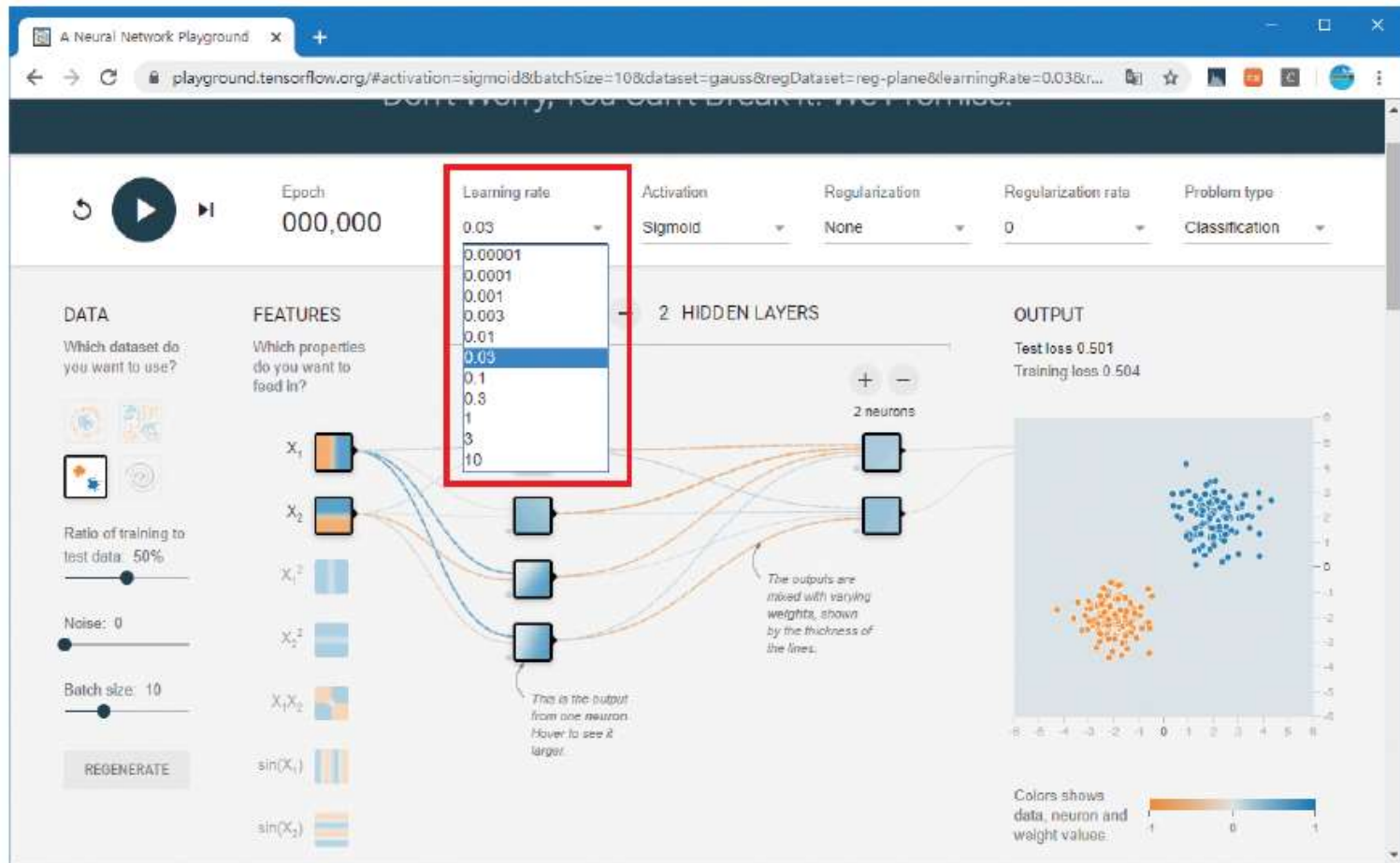
학습을 최적화하는 방법 - Adam

- **Adam**: Adam은 Adaptive Moment Estimation의 약자이다. **Adam**은 기본적으로 (**RMSprop+ 모멘텀**)이다.



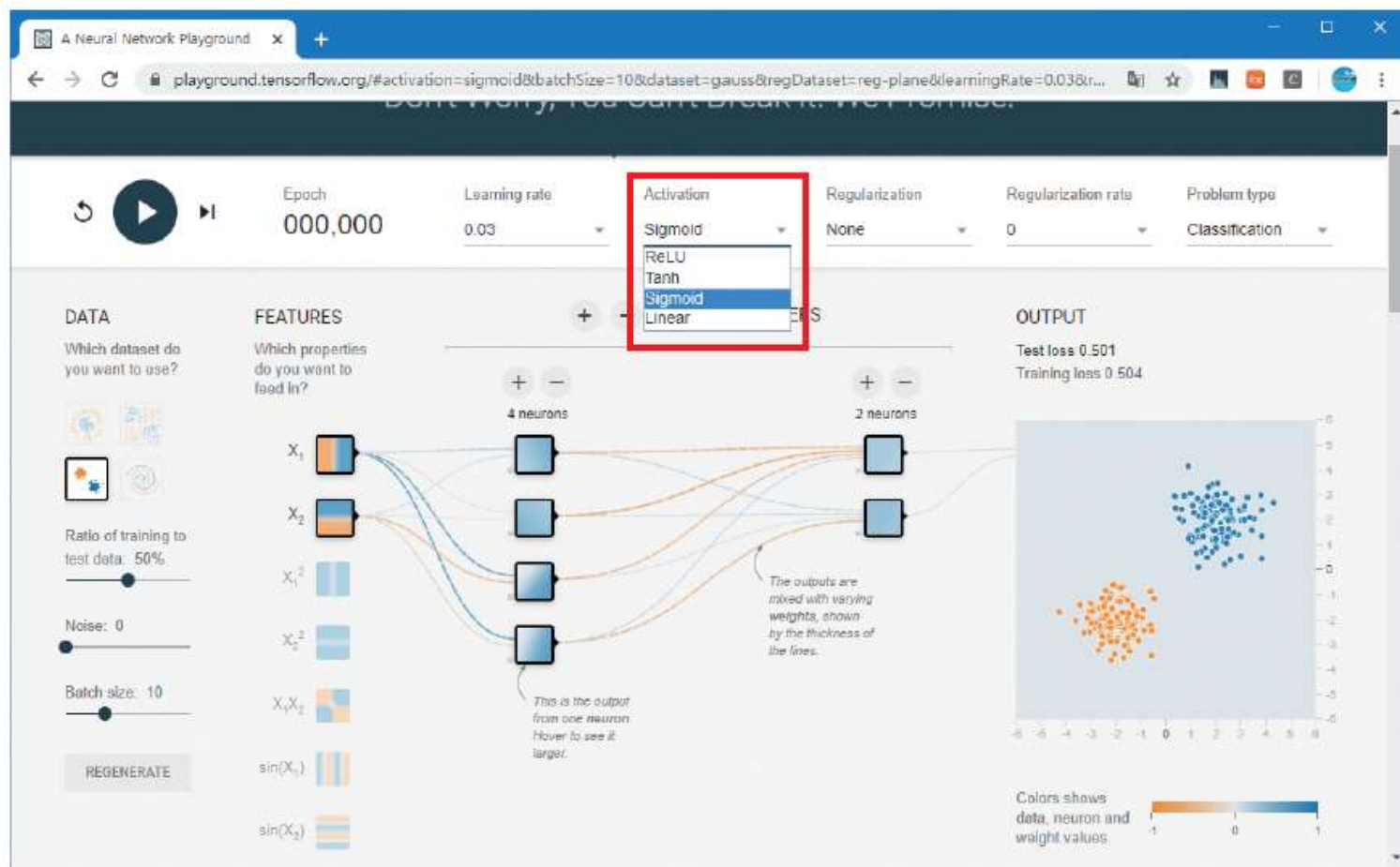


Lab: 학습률과 배치크기 실험



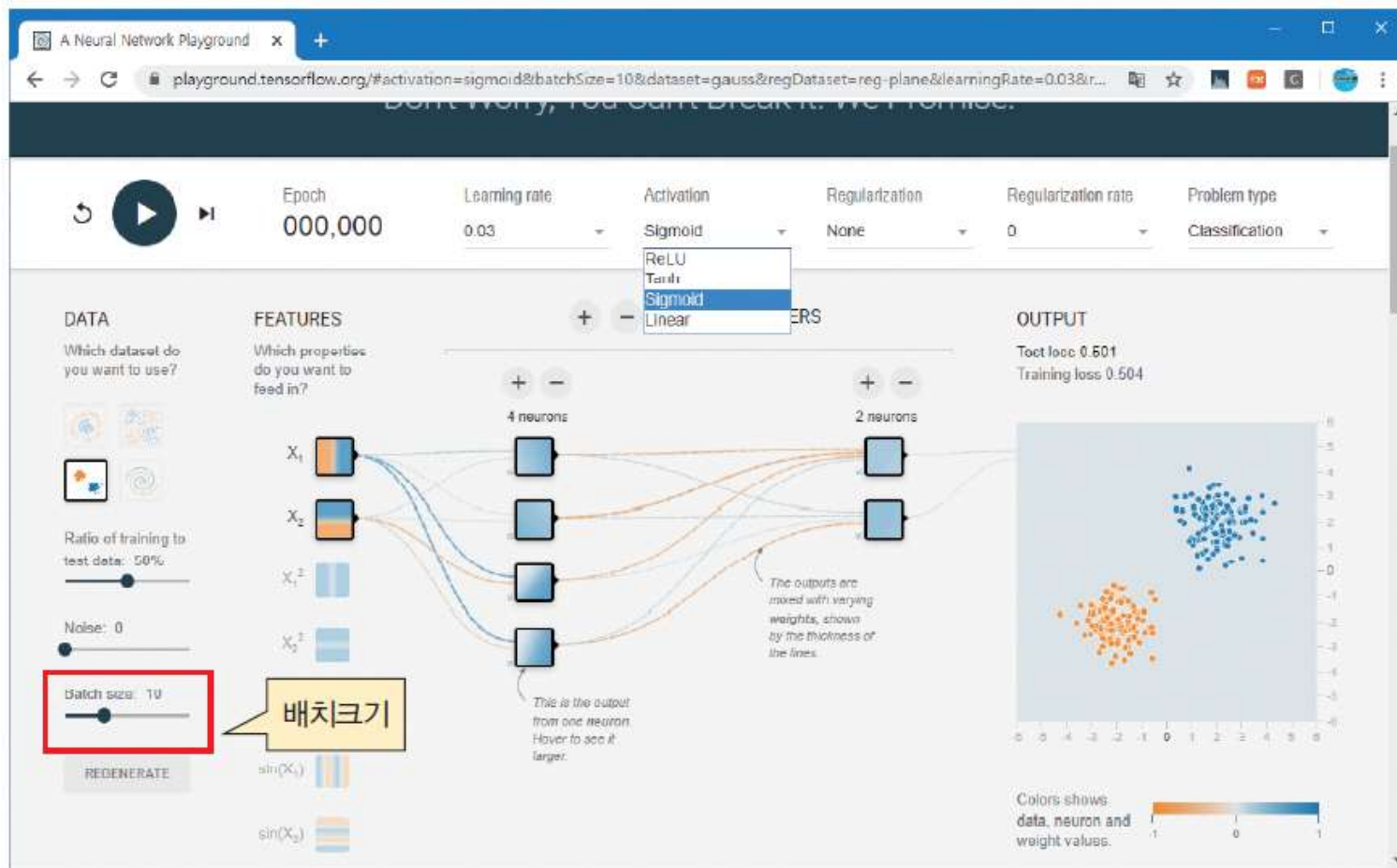


Lab: 학습과 배치 크기 실험





Lab: 학습과 배치크기 실험





텐서플로우와 케라스

- **텐서플로우(TensorFlow)**는 딥러닝 프레임워크의 일종이다. 텐서플로우는 내부적으로 **C/C++**로 구현되어 있고 파이썬을 비롯하여 여러 가지 언어에서 접근할 수 있도록 **데이터 구성과 계산 인터페이스를 제공**한다.

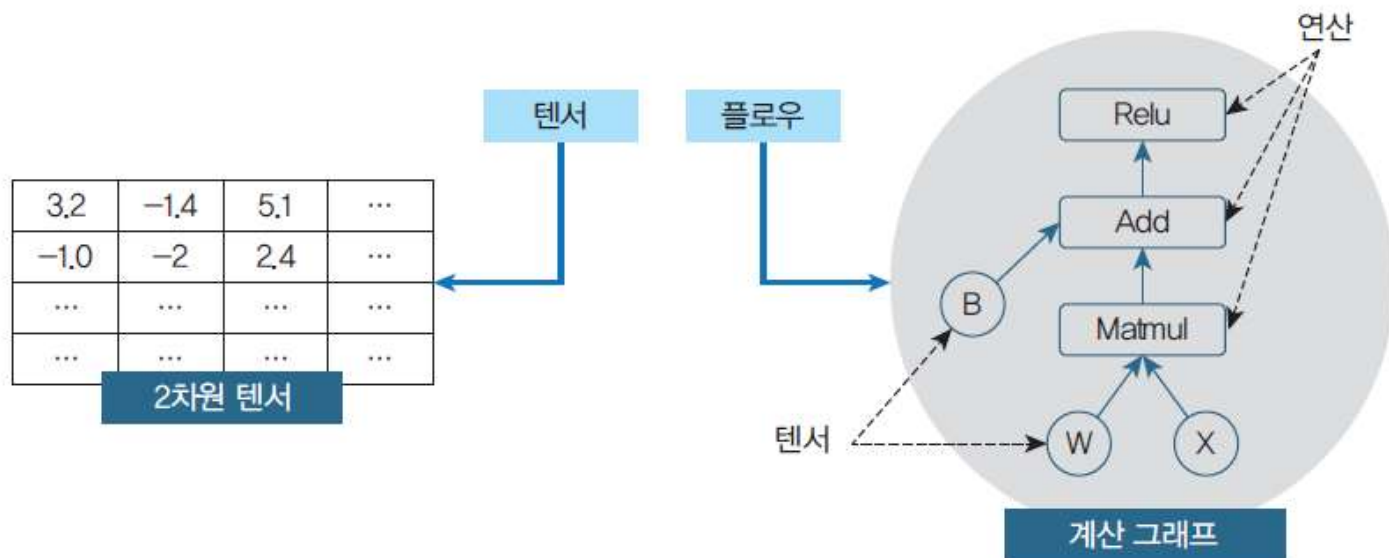
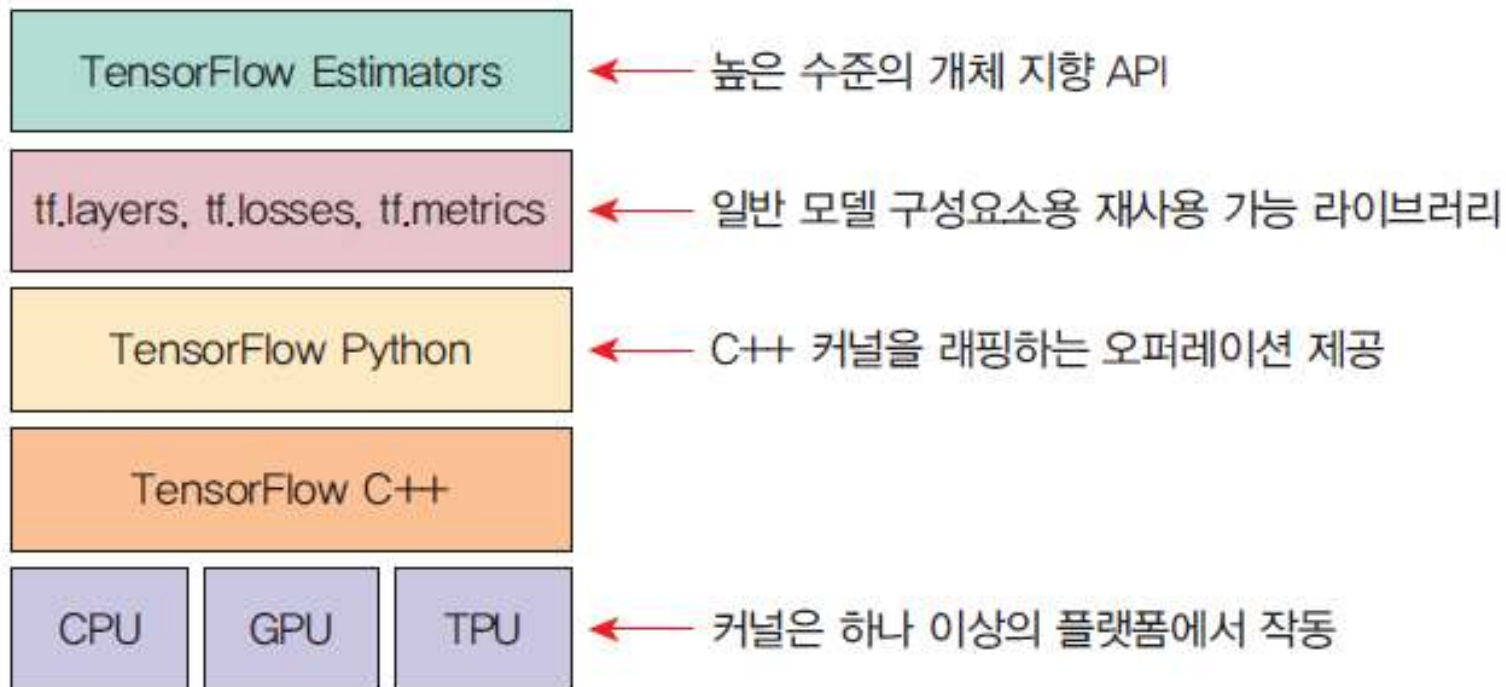


그림 7-4 텐서플로우의 개념



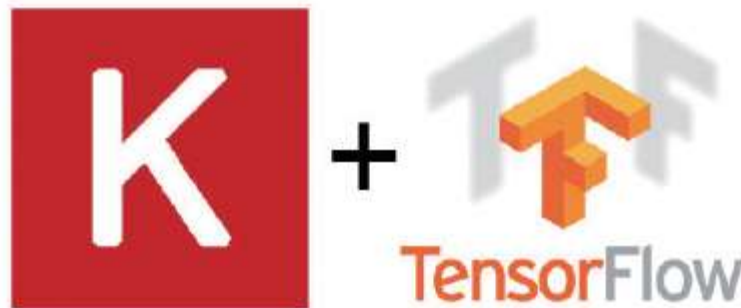
텐서플로우의 구조





케라스 (Keras)

- 케라스는 파이썬으로 작성되었으며, 고수준 딥러닝 **API**이다. 케라스에서는 여러 가지 백엔드를 선택할 수 있지만, 아무래도 가장 많이 선택되는 백엔드는 텐서플로우이다. (**Keras는 TF 2.x에 내장**)
- 쉽고 빠른 프로토타이핑이 가능하다.
- 피드포워드 신경망, 컨볼루션 신경망과 순환 신경망은 물론, 여러 가지의 조합도 지원한다.
- **CPU** 및 **GPU**에서 원활하게 실행된다.

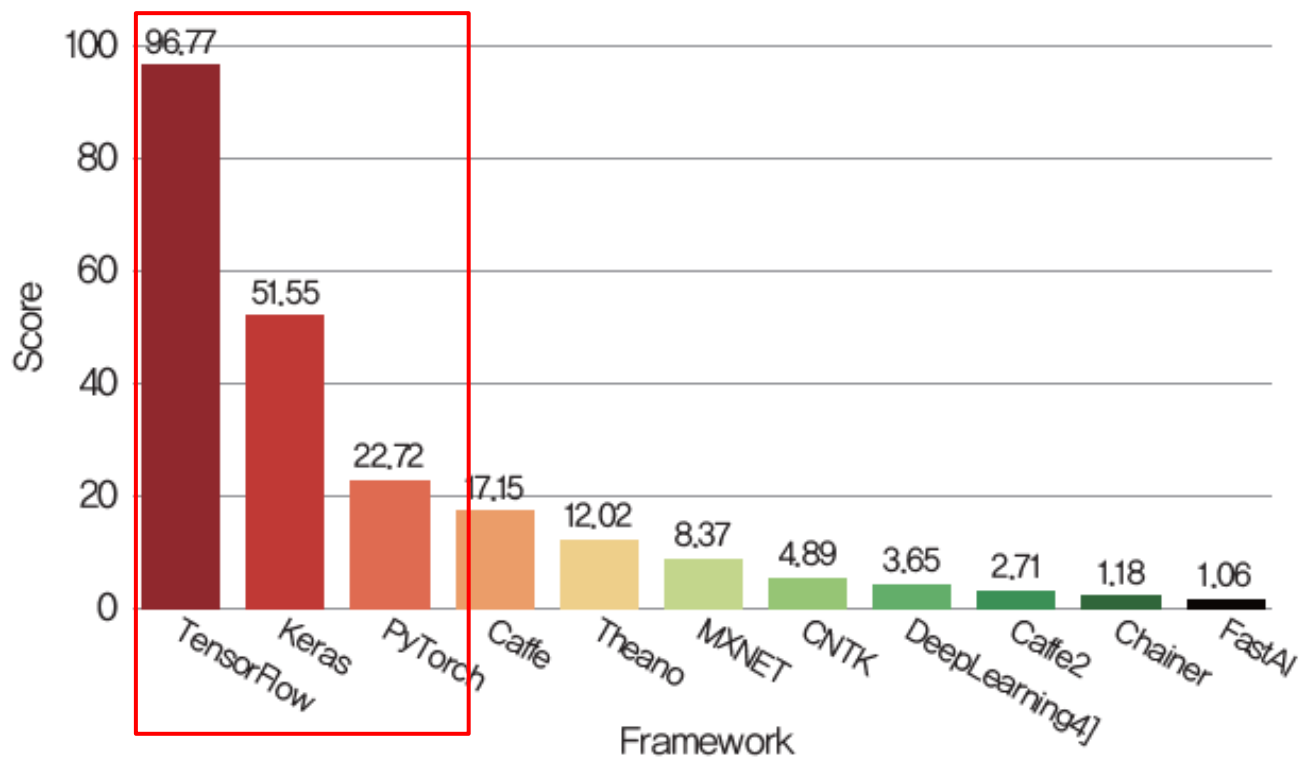


pip install tensorflow



딥러닝 프레임워크

Deep Learning Framework Power Scores 2018





케라스로 신경망을 작성하는 절차

- 케라스의 핵심 데이터 구조는 모델(model)이며 이것은 레이어를 구성하는 방법을 나타낸다.
- 가장 간단한 모델 유형은 **Sequential** 선형 스택 모델이다. **Sequential** 모델은 레이어를 선형으로 쌓을 수 있는 신경망 모델이다

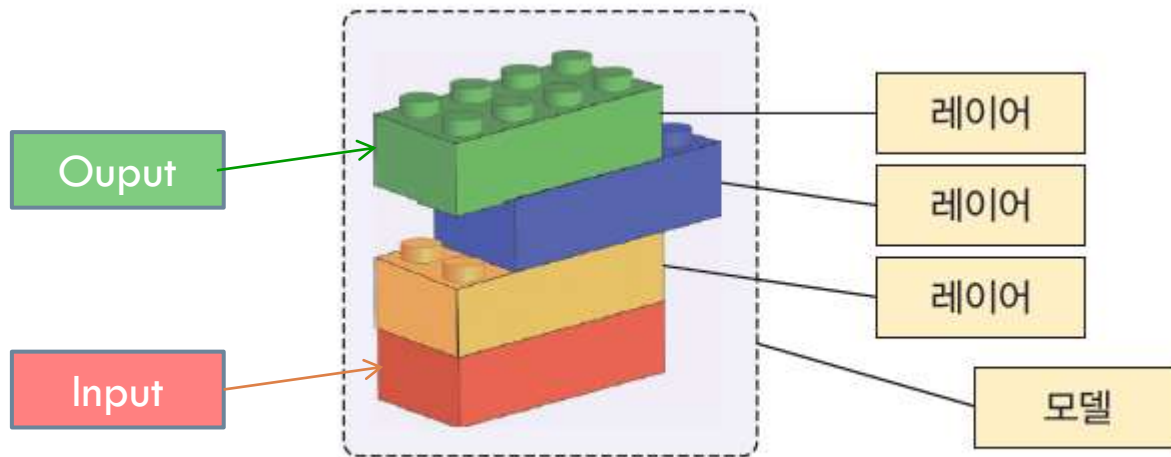
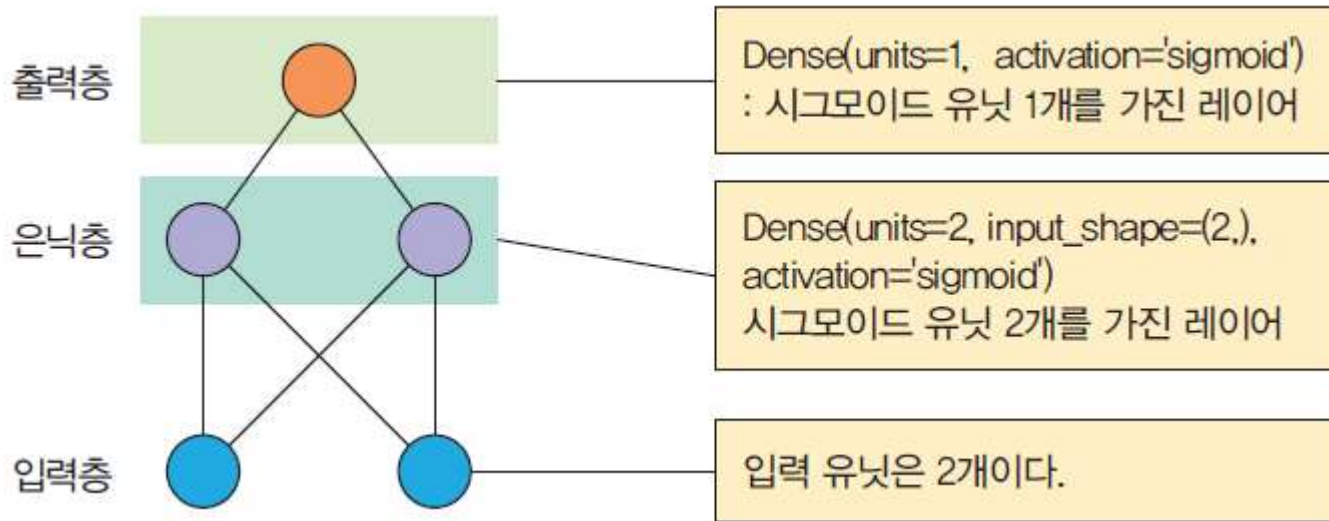


그림 7-5 케라스의 기본 개념



예제: XOR를 학습하는 MLP를 작성





훈련 데이터 (XOR)

	x1	x2		y
샘플 #1	0	0	➡	0
샘플 #2	0	1		1
샘플 #3	1	0		1
샘플 #4	1	1		0



model → training → testing

keras_xor.py

```
model = tf.keras.models.Sequential()
```

Sequential 모델을 생성

```
model.add(tf.keras.layers.Dense(units=2, input_shape=(2,), activation='sigmoid')) #①  
model.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
```

Sequential 모델에 add() 함수를
여 필요한 레이어를 추가

```
model.summary()
```

```
model.compile(loss='mean_squared_error', optimizer=keras.optimizers.SGD(lr=0.3))
```

```
model.fit(X, y, batch_size=1, epochs=10000)
```

compile() 함수를 호출하여서
Sequential 모델을 컴파일한다

```
print( model.predict(X) )
```

fit()를 호출하여서 학습을 수행한다.

predict()를 호출하여서 모델을 테스트한다



model.summary()

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
dense (Dense)	(None, 2)	6
=====		
dense_1 (Dense)	(None, 1)	3
=====		
Total params: 9		
Trainable params: 9		
Non-trainable params: 0		
=====		



실행 결과 - 1 (1-Hidden layers MLP)

...

Epoch 10000/10000

4/4 [=====] - 0s 748us/sample - loss: 9.7796e-04

[[0.02736092]

[0.9704443]

[0.9701309]

[0.03734875]]



실행 결과 - 2 (2-Hidden layers MLP)

```
model.add(tf.keras.layers.Dense(units=2, input_shape=(2,), activation='sigmoid')) #①
model.add(tf.keras.layers.Dense(units=4, activation='sigmoid')) #
model.add(tf.keras.layers.Dense(units=1, activation='sigmoid')) # output layer
model.compile(loss='mean_squared_error', optimizer=tf.keras.optimizers.SGD(lr=0.3))
```

```
model.fit(X, y, batch_size=1, epochs=10000)
```

```
print(model.predict(X))
```

```
# [[0.00956685]
# [0.9900732 ]
# [0.9919224 ]
# [0.00822851]]
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 2)	6
dense_1 (Dense)	(None, 4)	12
dense_2 (Dense)	(None, 1)	5
Total params: 23		
Trainable params: 23		
Non-trainable params: 0		



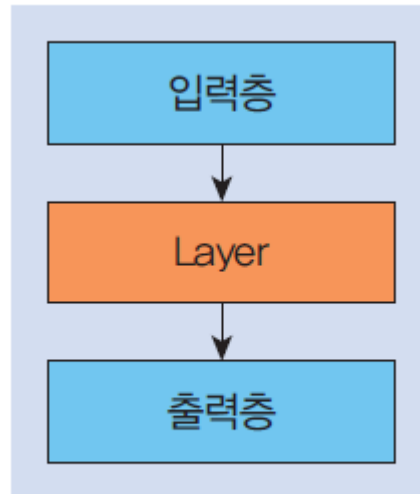
케라스를 사용하는 3가지 방법

(1) **Sequential** 모델을 만들고 모델에 필요한 레이어를 추가하는 방법이다.

```
model = Sequential()
```

```
model.add(Dense(units=2, input_shape=(2,), activation='sigmoid'))
```

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



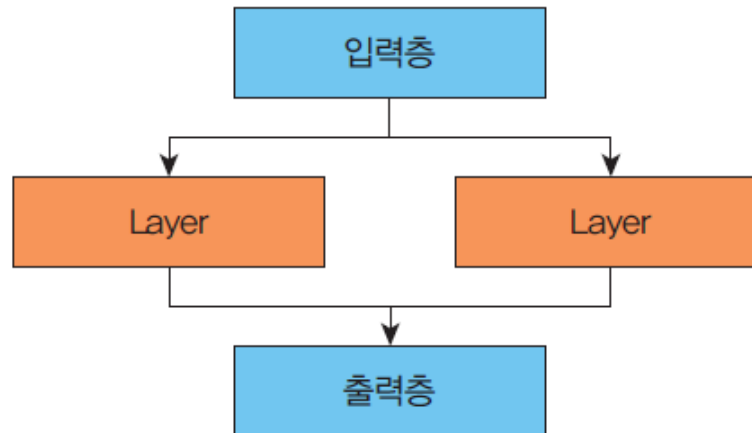
Sequential



케라스를 사용하는 3가지 방법

(2) 함수형 API를 사용하는 방법

```
inputs = Input(shape=(2,))          # 입력층  
x = Dense(2, activation="sigmoid")(inputs)  # 은닉층 ①  
prediction = Dense(1, activation="sigmoid")(x)  # 출력층  
  
model = Model(inputs=inputs, outputs=prediction)
```





케라스를 사용하는 3가지 방법

(3) Model 클래스를 상속받아서 우리 나름대로의 **클래스를 정의**하는 방법

```
class SimpleMLP(Model):

    def __init__(self, num_classes):    # 생성자 작성
        super(SimpleMLP, self).__init__(name='mlp')
        self.num_classes = num_classes

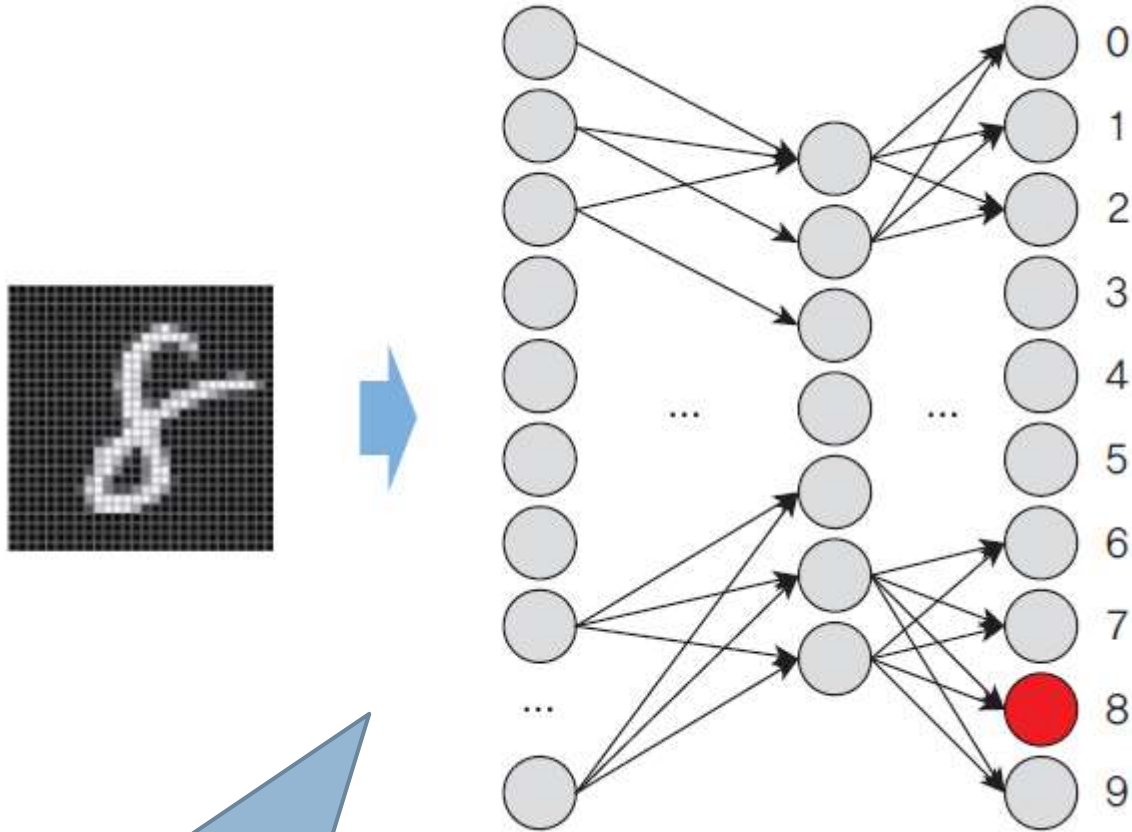
        self.dense1 = Dense(32, activation='sigmoid')
        self.dense2 = Dense(num_classes, activation='sigmoid')

    def call(self, inputs):              # 순방향 호출을 구현한다.
        x = self.dense1(inputs)
        return self.dense2(x)

model = SimpleMLP()
model.compile(...)
model.fit(...)
```



케라스를 이용한 MNIST 숫자 인식



딥러닝의 “Hello World” 예제



MNIST 필기체 숫자 데이터 세트

- 1980년대에 미국의 국립표준 연구소(NIST)에서 수집한 데이터 세트로 6만 개의 훈련 이미지와 1만개의 테스트 이미지로 이루어져 있다

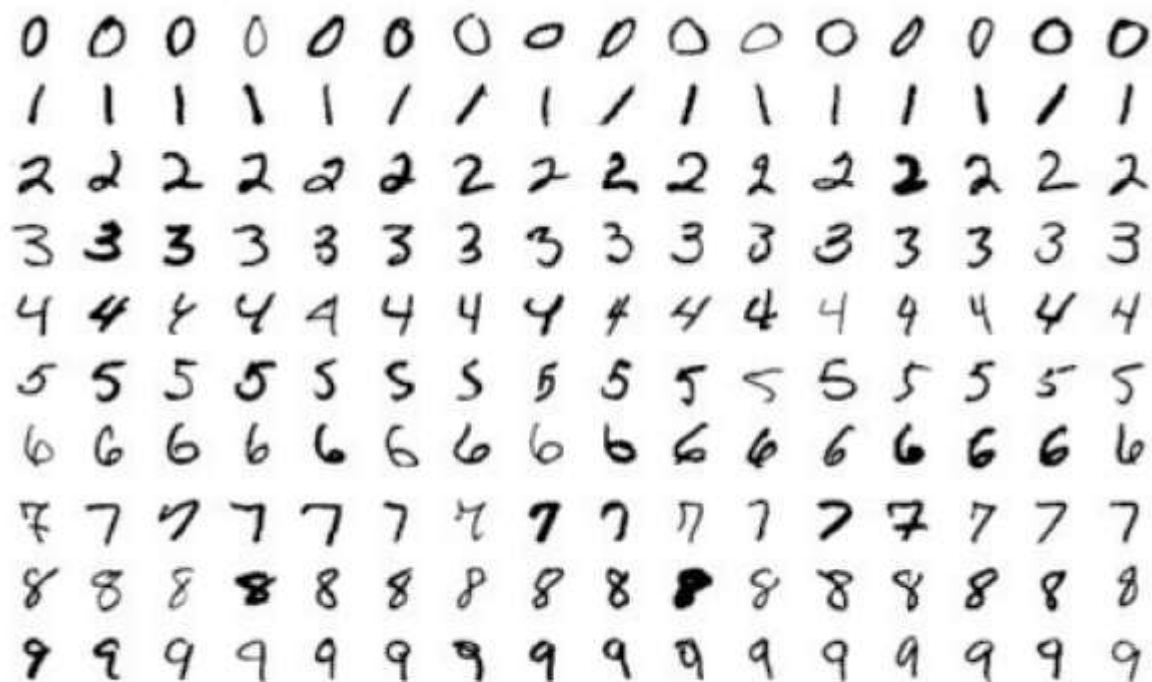


그림 7-7 MNIST 데이터



숫자 데이터 가져오기

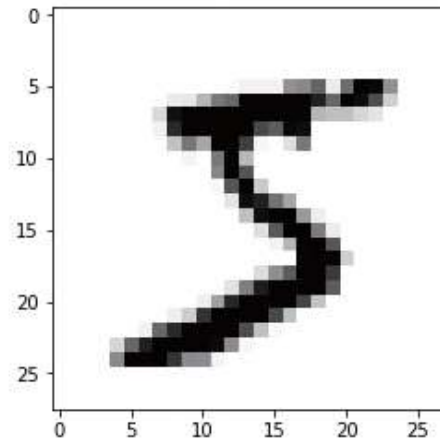
```
import matplotlib.pyplot as plt
import tensorflow as tf

(train_images, train_labels), (test_images, test_labels)
    = tf.keras.datasets.mnist.load_data()
```

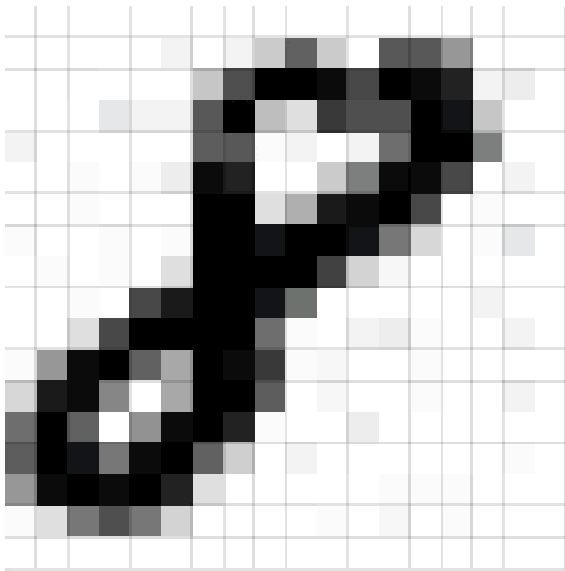



숫자 데이터 표시하기

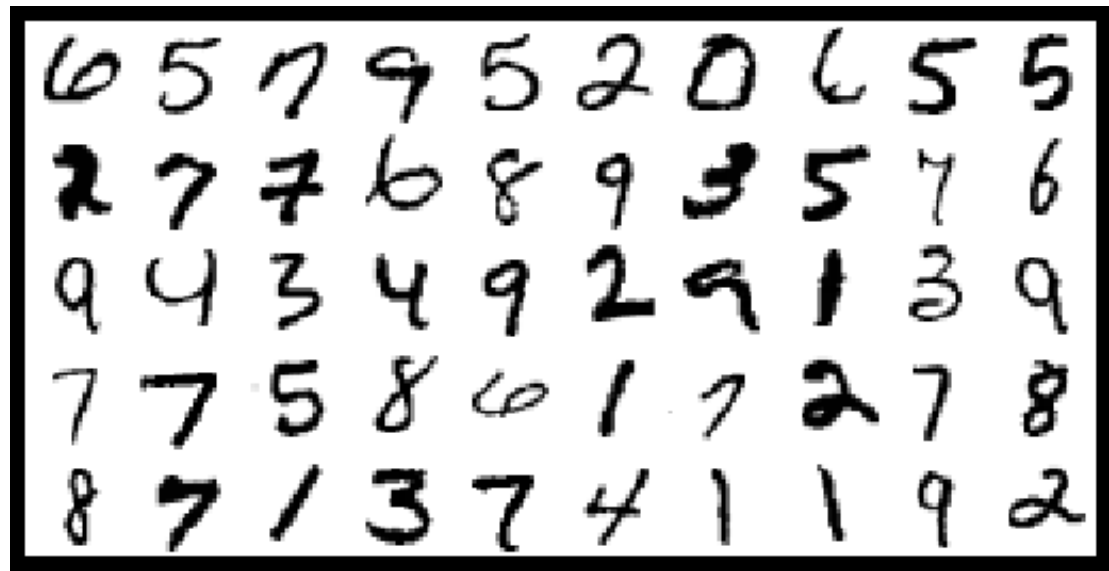
```
>>> train_images.shape  
(60000, 28, 28)  
  
>>> train_label  
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)  
  
>>> test_images.shape  
(10000, 28, 28)  
  
>>> plt.imshow(train_images[0], cmap="Greys")
```



MNIST

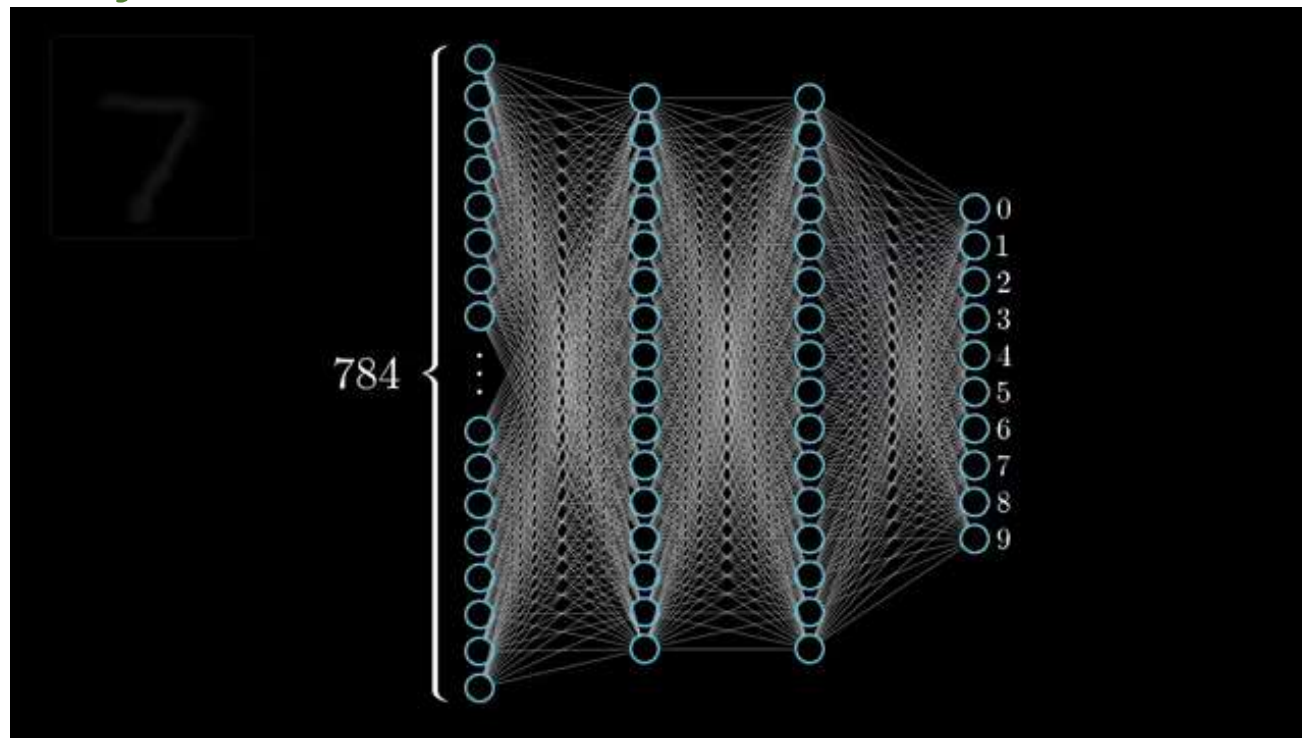
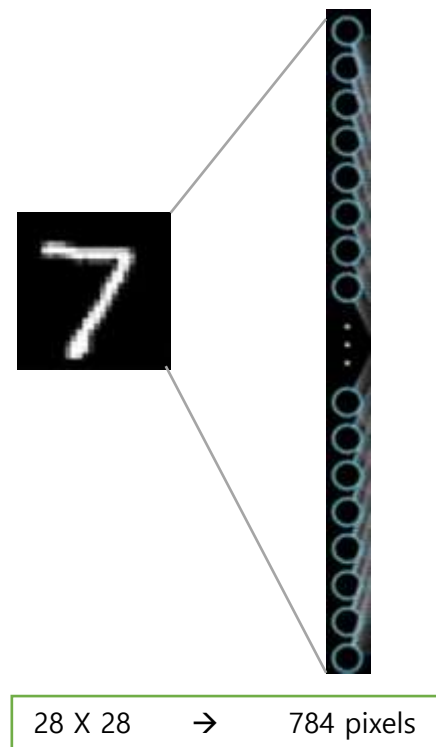


28 X 28 : 784 pixels
Gray scale [0 ~ 255]



MLP: Multi-Layer Perceptron (90%)

Fully Connected Network (FCN)



Source: <https://gfycat.com/ko/deadlydeafeningatlanticblackgoby-three-blue-one-brown-machines-learning>

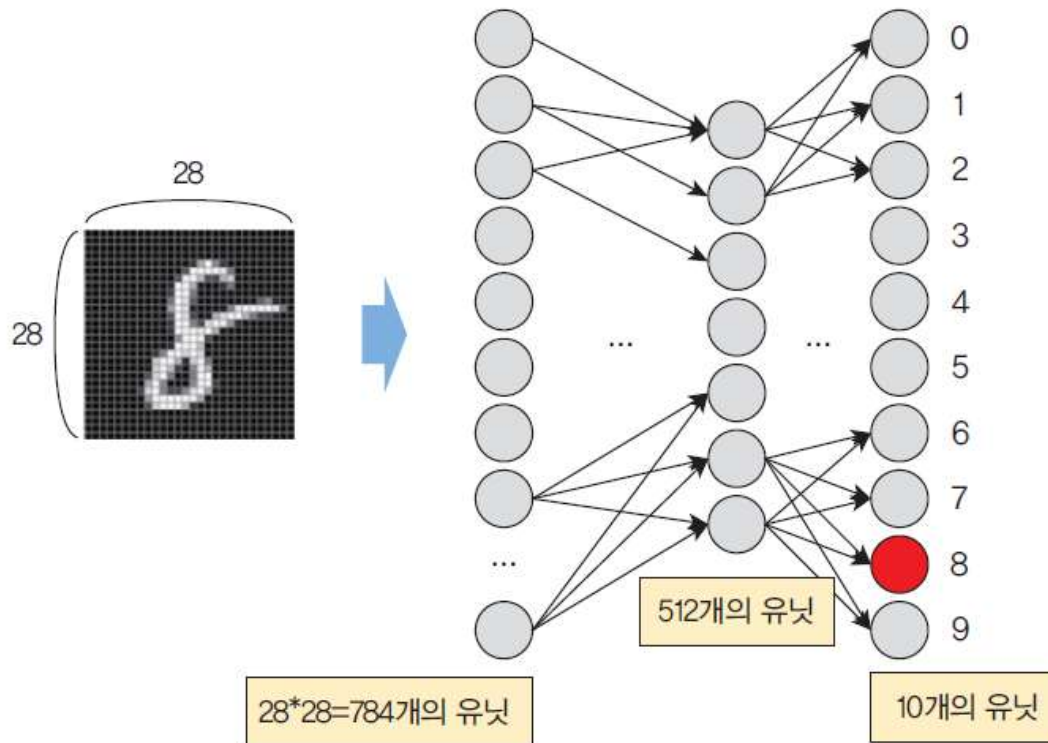


신경망 모델 구축하기

```
model = tf.keras.models.Sequential()
```

```
model.add(tf.keras.layers.Dense(512, activation='relu', input_shape=(784,)))
```

```
model.add(tf.keras.layers.Dense(10, activation='sigmoid'))
```





옵티마이저와 손실함수, 지표 등을 정의하는 컴파일 단계

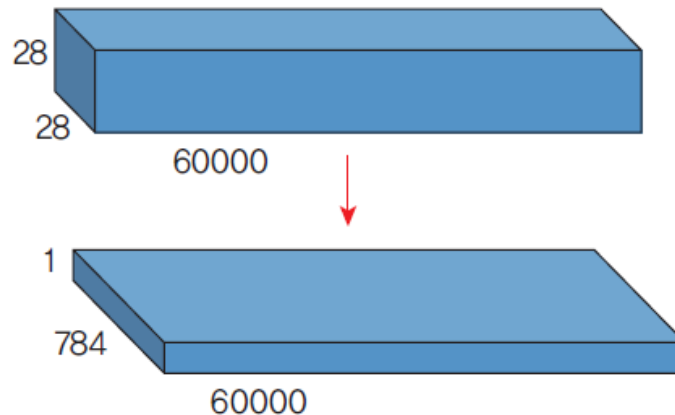
```
model.compile(optimizer='rmsprop',  
              loss='mse',  
              metrics=['accuracy'])
```

- 손실함수(**loss function**): 신경망의 출력과 정답 간의 오차를 계산하는 함수
- 옵티마이저(**optimizer**): 손실 함수를 기반으로 신경망의 파라미터를 최적화하는 알고리즘
- 지표(**metric**): 훈련과 테스트 과정에서 사용되는 척도



데이터 전처리: flattening & normalization

```
train_images = train_images.reshape((60000, 784)) # 28 x 28 → 784  
train_images = train_images.astype('float32') / 255.0 # Normalization  
  
test_images = test_images.reshape((10000, 784))  
test_images = test_images.astype('float32') / 255.0
```





정답 레이블 형태 변경 (원핫 인코딩)

```
train_labels = tf.keras.utils.to_categorical(train_labels)  
test_labels = tf.keras.utils.to_categorical(test_labels)
```

0	→	[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
1	→	[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
2	→	[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
3	→	[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]



하
삼
기

```
model.fit(train_images, train_labels, epochs=5, batch_size=128)
```

Epoch 1/5

469/469 [=====] - 2s 3ms/step - loss: 0.0158 - accuracy:
0.9168

...

Epoch 5/5

469/469 [=====] - 2s 3ms/step - loss: 0.0027 - accuracy:
0.9867



테스트

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print('테스트 정확도:', test_acc)
```

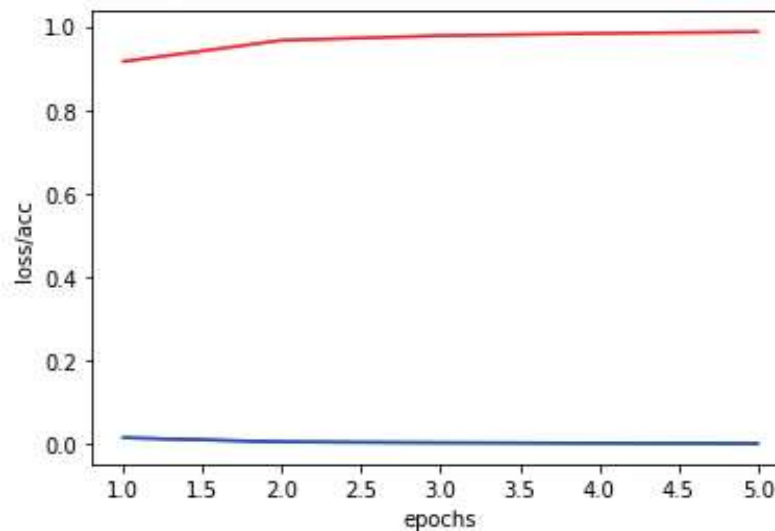
```
313/313 [=====] - 0s 892us/step - loss: 0.0039 -
accuracy: 0.9788
테스트 정확도: 0.9787999987602234
```



그래프 그리기 — loss & accuracy

```
history = model.fit(train_images, train_labels, epochs=5, batch_size=128)
loss = history.history['loss']
acc = history.history['accuracy']
epochs = range(1, len(loss)+1)
```

```
plt.plot(epochs, loss, 'b', label='Training Loss')
plt.plot(epochs, acc, 'r', label='Accuracy')
plt.xlabel('epochs')
plt.ylabel('loss/acc')
plt.show()
```





실제 이미지로 테스트하기

```
import cv2 as cv
```

```
image = cv.imread('test.png', cv.IMREAD_GRAYSCALE)
```

```
image = cv.resize(image, (28, 28))
```

```
image = image.astype('float32')
```

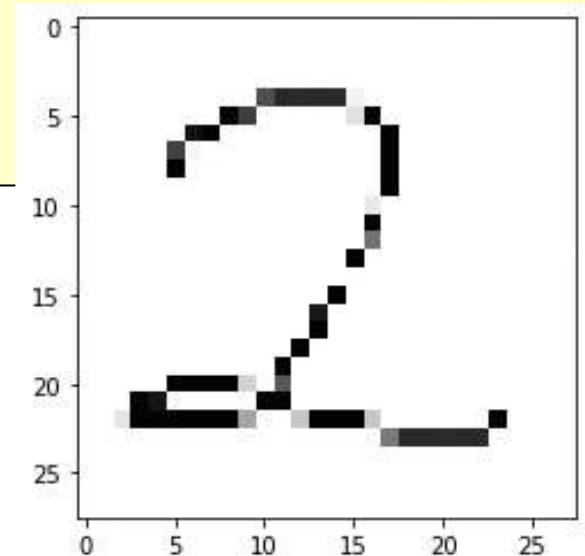
```
image = image.reshape(1, 784) # 1 batch
```

```
image = 255-image # BW inversion
```

```
image /= 255.0
```

```
plt.imshow(image.reshape(28, 28), cmap='Greys')
```

```
plt.show()
```





테스트

```
pred = model.predict(image.reshape(1, 784), batch_size=1)  
print("추정된 숫자=", pred.argmax())
```

추정된 숫자= 2

도전문제

- (1) 은닉층 유닛의 개수는 성능에 어떻게 영향을 끼치는가? 은닉층 유닛의 개수를 변경하면서 정확도가 어떻게 변하는지를 관찰해보자.
- (2) 배치 크기를 변경하면서 학습의 정확도가 어떻게 변하는지를 관찰해보자.
- (3) 은닉층의 활성화 함수를 relu에서 시그모이드 함수로 변경해보자. 학습의 정확도가 어떻게 변하는지를 관찰해보자.



케라스의 입력 데이터

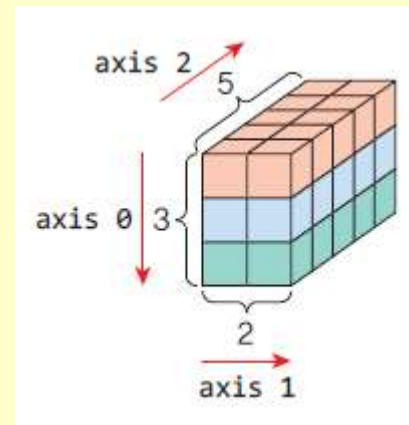
- 넘파이 배열: `numpy Array`
- **TensorFlow Dataset** 객체: 크기가 커서, 메모리에 한 번에 적재될 수 없는 경우에 디스크 또는 분산 파일 시스템에서 스트리밍될 수 있다.
- 파이썬 제너레이터: 예를 들어서 `keras.utils.Sequence` 클래스는 하드 디스크에 위치한 파일을 읽어서 순차적으로 케라스 모델로 공급할 수 있다.



텐서

- 텐서는 다차원 넘파이 배열이다.
- 텐서는 데이터(실수)를 저장하는 컨테이너라고 생각하면 된다. 텐서에서는 배열의 차원을 축(**axis**)이라고 부른다.
- 예를 들어서 3차원 텐서는 다음과 같이 생성할 수 있다.

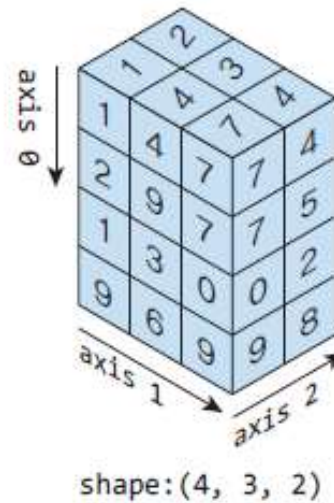
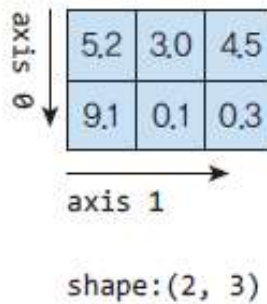
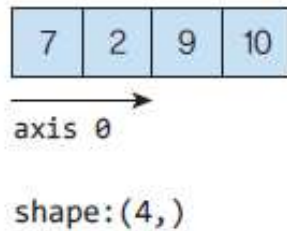
```
>>> import numpy as np
x = np.array(
    [[0, 1, 2, 3, 4],
     [5, 6, 7, 8, 9]],
    [[10, 11, 12, 13, 14],
     [15, 16, 17, 18, 19]],
    [[20, 21, 22, 23, 24],
     [25, 26, 27, 28, 29]],])
>>> x.ndim
3
>>> x.shape
(3, 2, 5)
```





텐서의 속성

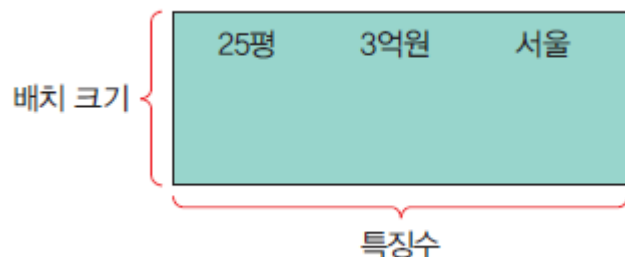
- 텐서의 차원(축의 개수): 텐서에 존재하는 축의 개수이다. 3차원 텐서에는 3개의 축이 있다. **ndim** 속성
- 형상(**shape**): 텐서의 각 축으로 얼마나 데이터가 있는지를 파이썬의 튜플로 나타낸 것이다.
- 데이터 타입(**data type**): 텐서 요소의 자료형.



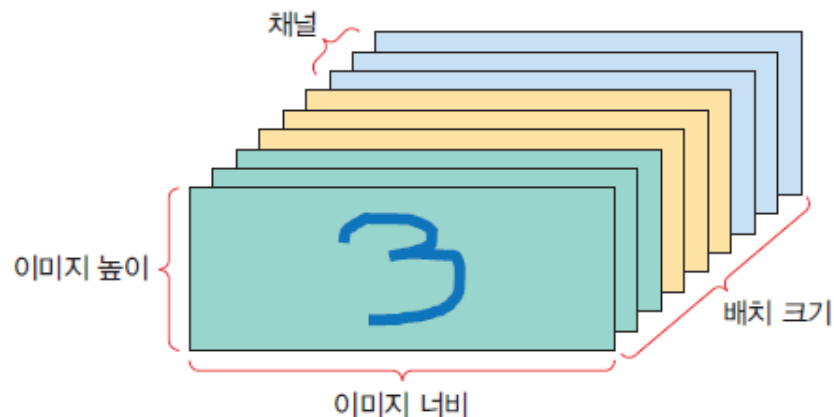


훈련 데이터의 형상 - 벡터, 이미지

- 벡터 데이터: (배치 크기, 특징수)의 형상을 가진다. (2차원 ndarray)



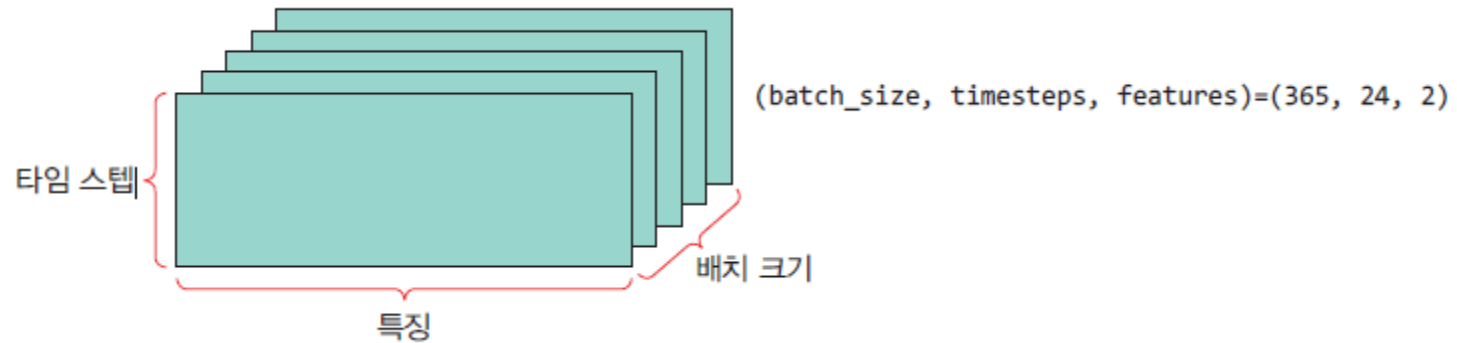
- 이미지 데이터: (배치 크기, 이미지 높이, 이미지 너비, 채널수) 형상의 4차원 넘파이 텐서에 저장된다





훈련 데이터의 형상 - 시계열

- 시계열 데이터: (배치 크기, 타임 스텝, 특징수) 형상의 3차원 넘파이 텐서에 저장된다.





케라스의 클래스들

- **모델**: 하나의 신경망을 나타낸다.
- **레이어**: 신경망에서 하나의 층이다.
- **입력 데이터**: 텐서플로우 텐서 형식이다.
- **손실 함수**: 신경망의 출력과 정답 레이블 간의 차이를 측정하는 함수이다.
- **옵티마이저**: 학습을 수행하는 최적화 알고리즘이다. 학습률과 모멘텀을 동적으로 변경한다



Sequential 모델

- **add(layer):** 레이어를 모델에 추가한다.
- **compile(optimizer, loss=None, metrics=None):** 훈련을 위해서 모델을 구성하는 메소드
- **fit(x=None, y=None, batch_size=None, epochs=1, verbose=1):** 훈련 메소드
- **evaluate(x=None, y=None):** 테스트 모드에서 모델의 손실 함수 값과 측정 항목 값을 반환
- **predict(x, batch_size=None):** 입력 샘플에 대한 예측값을 생성



레이어 클래스들 (layers.*)

- `Input(shape, batch_size, name)`: 입력을 받아서 케라스 텐서를 생성하는 객체
- `Dense(units, activation=None, use_bias=True, input_shape)`: 유닛들이 전부 연결된 레이어
- `Embedding(input_dim, output_dim, input_length)`:
- `Flatten()`
- `Conv2D`
- `MaxPooling2D`



손실 함수

- **MeanSquaredError**: 정답 레이블과 예측값 사이의 평균 제곱 오차를 계산한다.
- **BinaryCrossentropy**: 정답 레이블과 예측 레이블 간의 교차 엔트로피 손실을 계산한다(예를 들어서 강아지, 강아지 아님).
- **CategoricalCrossentropy**: 정답 레이블과 예측 레이블 간의 교차 엔트로피 손실을 계산한다(예를 들어서 강아지, 고양이, 호랑이).
→ 정답 레이블은 원핫 인코딩으로 제공되어야 한다.
- **SparseCategoricalCrossentropy**: 정답 레이블과 예측 레이블 간의 교차 엔트로피 손실을 계산한다 (예를 들어서 강아지, 고양이, 호랑이).
→ 정답 레이블은 정수로 제공되어야 한다.



측정 항목

- **Accuracy:** 정확도이다. 예측값이 정답 레이블과 같은 횟수를 계산한다.
- **categorical_accuracy:** 범주형 정확도이다. 신경망의 예측값이 원-핫 레이블과 일치하는 빈도를 계산한다.

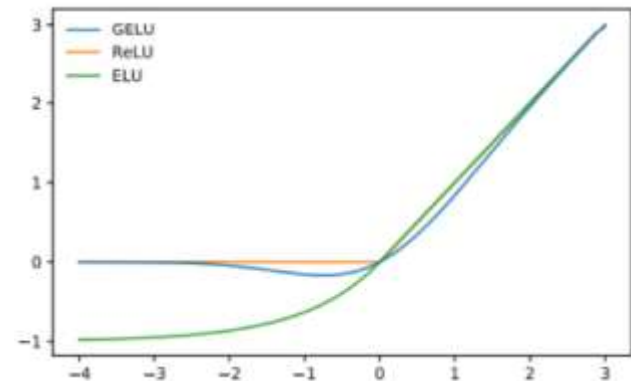
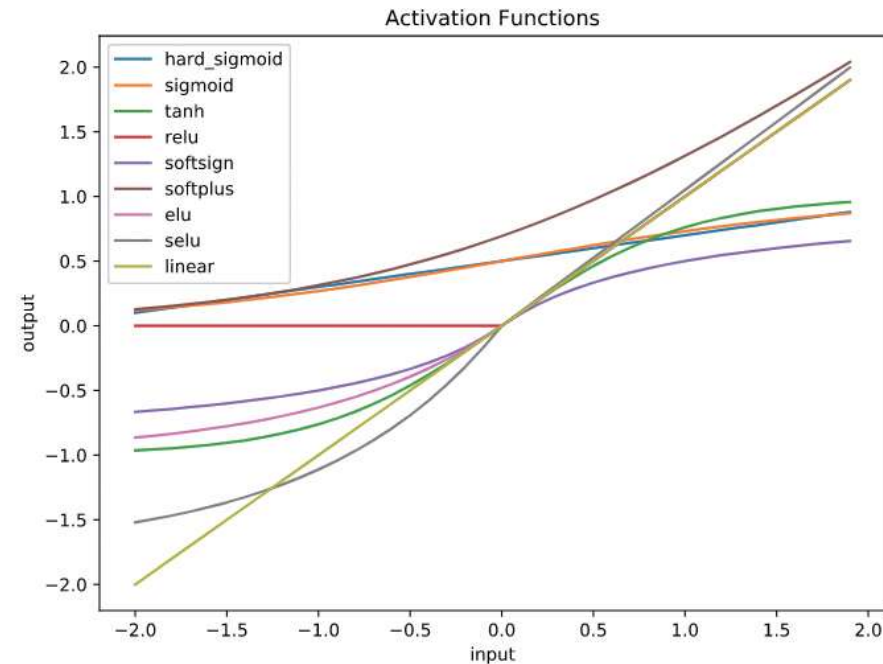


- **SGD:** 확률적 경사 하강법(Stochastic Gradient Descent, SGD) . Nesterov 모멘텀을 지원한다.
- **Adagrad:** Adagrad는 가변 학습률을 사용하는 방법
- **Adadelat:** Adadelat는 모멘텀을 이용하여 감소하는 학습률 문제를 처리하는 Adagrad의 변형이다.
- **RMSprop:** RMSprop는 Adagrad에 대한 수정판이다.
- **Adam:** Adam은 기본적으로 (**RMSprop + 모멘텀**)이다.



활성화 함수 (activation function)

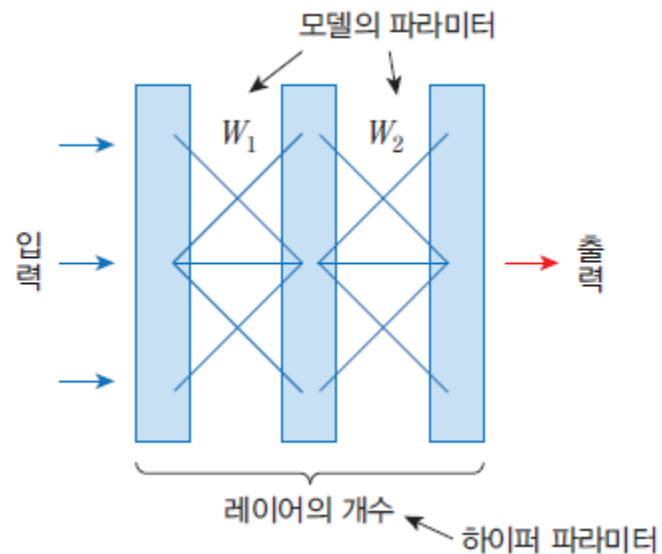
- **sigmoid**
- **relu**(Rectified Linear Unit)
- **softmax**
- **tanh**
- **selu**(Scaled Exponential Linear Unit)
- **Softplus**
- **gelu** (Gaussian Error Linear Unit)





하이퍼 매개변수 (hyper-parameters)

- 학습률이나 은닉층을 몇 개로 할 것이며, 은닉층의 개수나 유닛의 개수는 누가 정하는 것일까? -> **하이퍼 매개변수**
- 즉 신경망의 학습률이나 모멘텀의 가중치, 은닉층의 개수, 유닛의 개수, 미니 배치의 크기 등이 하이퍼 매개변수이다





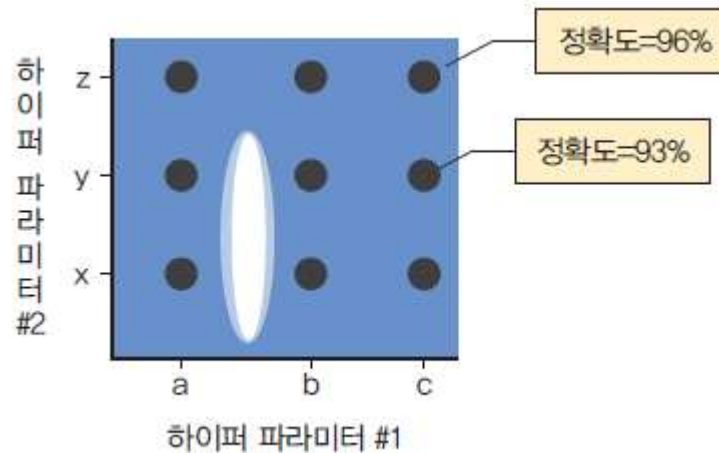
하이퍼 매개 변수를 찾는 방법

- 기본값 사용: 라이브러리 개발자가 설정한 기본값을 그대로 사용한다.
- 수동 검색: 사용자가 하이퍼 매개변수를 지정한다.
- 그리드 검색: 격자 형태로 하이퍼 매개변수를 변경하면서 성능을 측정하는 방법이다.
- 랜덤 검색: 랜덤으로 검색한다.
- **AutoML**
- **AutoAI**



그리드 검색

- 각 하이퍼 매개변수에 대하여 몇 개의 값을 지정하면 이 중에서 가장 좋은 조합을 찾아주는 알고리즘이다.
- **sklearn** 패키지에서 제공해주고 있기 때문에 손쉽게 사용할 수 있다





예제 : Grid search - 1

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.model_selection import GridSearchCV
# from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
from scikeras.wrappers import KerasClassifier

# 데이터 세트 준비
(train_images, train_labels), (test_images, test_labels) =
tf.keras.datasets.mnist.load_data()

train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255

train_labels = tf.keras.utils.to_categorical(train_labels)
test_labels = tf.keras.utils.to_categorical(test_labels)
```



예제 : Grid search - 2

Sequential model을 함수로 정의

신경망 모델 구축

```
def build_model():
```

```
    network = tf.keras.models.Sequential()
```

```
    network.add(tf.keras.layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
```

```
    network.add(tf.keras.layers.Dense(10, activation='sigmoid'))
```

```
    network.compile(optimizer='rmsprop',  
                    loss='categorical_crossentropy',  
                    metrics=['accuracy'])
```

```
    return network
```

하이퍼 매개변수 딕셔너리

```
param_grid = {
```

```
    'epochs':[1, 2, 3], # 에포크 수: 1, 2, 3
```

```
    'batch_size':[32, 64]      # 배치 크기: 32, 64
```

```
}
```



예제 : Grid search - 3

```
# 케라스 모델을 KerasClassifier를 사용해서 포장한다.  
model = KerasClassifier(build_fn = build_model, verbose=1)  
  
# 그리드 검색  
gs = GridSearchCV(  
    estimator=model,  
    param_grid=param_grid,  
    cv=3,  
    n_jobs=-1  
)  
  
# 그리드 검색 결과 출력  
grid_result = gs.fit(train_images, train_labels)  
print(grid_result.best_score_)  
print(grid_result.best_params_)
```



실험결과 : Grid search - 4

...

Epoch 3/3

938/938 [=====] - 3s 4ms/step - loss: 0.0664 - accuracy: 0.9799

157/157 [=====] - 0s 939us/step

0.968666672706604

{'batch_size': 64, 'epochs': 3}



Summary

- **풀배치**는 훈련 데이터 세트를 모두 처리한 후에 가중치를 변경하는 것이다. 안정적이지만 학습속도는 느다. **SGD**는 훈련 데이터 세트 중에서 랜덤하게 하나를 뽑아서 처리하고 가중치를 변경하는 방법이다. 속도는 빠르지만 불안정하게 수렴할 수 있다. 중간 방법이 미니 배치이다. **미니 배치**에서는 일정한 수의 샘플을 뽑아서 처리한 후에 가중치를 변경한다.
- 학습률은 중요한 하이퍼 매개변수이다. 학습률은 적응적 학습 방법이 많이 사용된다. 즉 현재 그래디언트 값이나 가중치의 값을 고려하여 **학습률이 동적으로 결정**된다. 많이 사용되는 알고리즘으로 **RMSprop**나 **Adam**이 있다.
- 케라스를 사용하면 쉽게 신경망 모델을 구축할 수 있다. 가장 간단한 방법은 **Sequential** 모델을 생성하고 여기에 필요한 레이어들을 추가하는 방법이다. 많이 사용되는 레이어에는 **Dense** 레이어가 있다. **Dense** 레이어는 레이어 안의 유닛들이 다른 레이어의 뉴론들과 전부 연결된 형태의 레이어이다.
- **하이퍼 매개변수**란 신경망 모델의 가중치나 바이어스와는 다르게, 개발자가 모델에 대하여 임의로 결정하는 값이다. 은닉층의 수나 유닛의 수, 학습률 등이 여기에 속한다. 그리드 검색을 사용하여 최적의 값을 검색할 수도 있다



Q & A

