

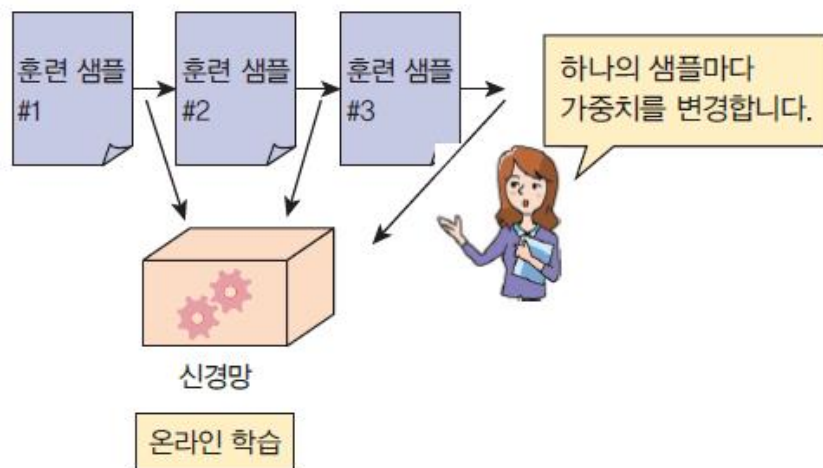
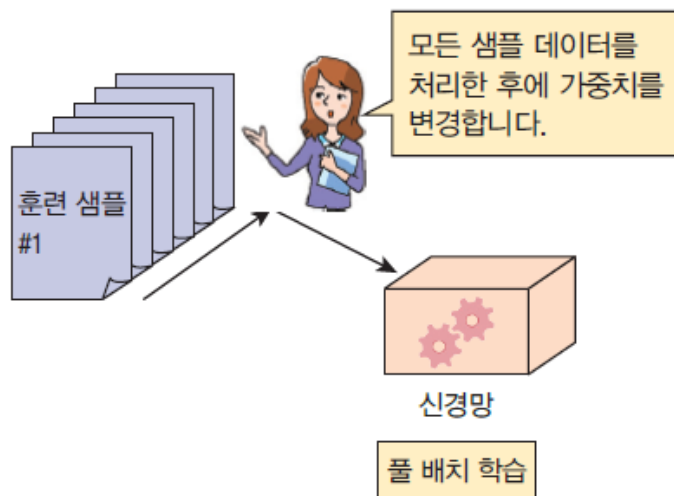
MLP와 케라스 라이브러리



- MLP에서, 몇 개의 샘플을 선택한 후에 가중치를 변경할 것인가?를 결정하는 방법
 - 1. 풀 배치
 - 2. 온라인 학습
 - 3. 미니 배치

풀 배치와 온라인 배치

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





풀 배치와 온라인 배치

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.

확률적 경사 하강법

1. 가중치와 바이어스를 0부터 1 사이의 난수로 초기화한다.
2. 수렴할 때까지 모든 가중치에 대하여 다음을 반복한다.
3. 훈련 샘플 중에서 무작위로 i번째 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

- 온라인 학습과 풀 배치 학습의 중간
- 훈련 샘플들을 작은 배치들로 분리시킨 후에 하나의 배치가 끝날 때마다 학습을 수행하는 방법
- 예. 10000개의 샘플이 있다면 100개 정도의 샘플을 랜덤하게 뽑아서 학습

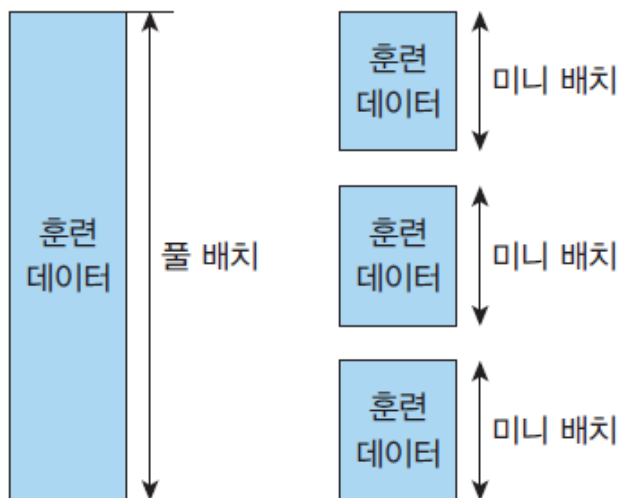
미니 배치 경사 하강법

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. $w \leftarrow w - \eta * \frac{\partial E}{\partial w}$

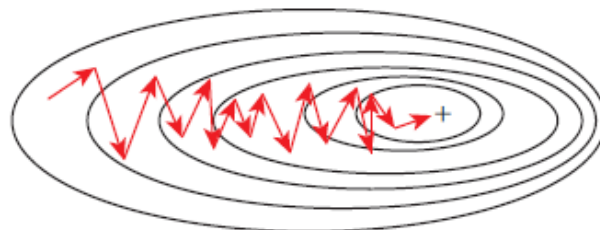


각 방법들의 비교

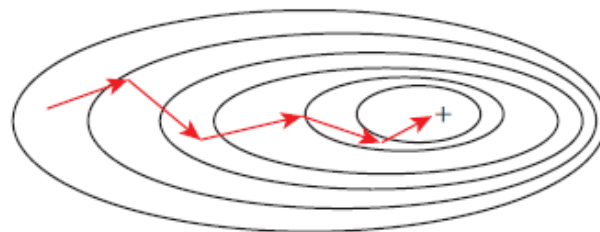
- 풀 배치 학습 : 시간이 오래 걸려. 메모리가 많이 요구
- 온라인 학습 : 하나의 샘플마다 매개변수 업데이트로 잡음에 취약. 특정한 샘플에 의하여 크게 좌우. 즉 하나의 잘못된 샘플때문에 가중치가 엉뚱한 방향으로 갈 수도.
- 미니 배치 : 비교적 안정되게 수렴



확률적 경사 하강법(SGD)

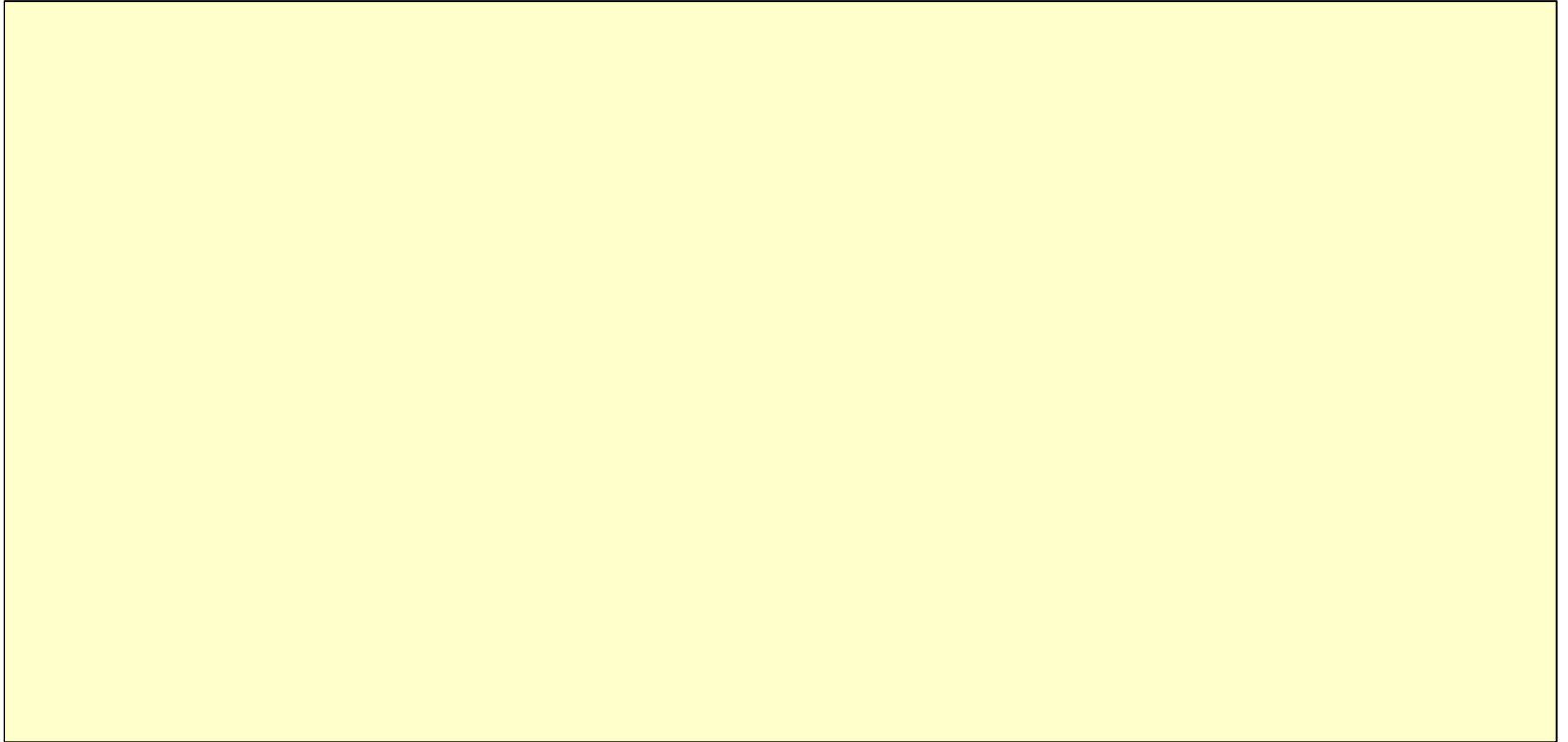


미니 배치 경사 하강법





Lab: 미니 배치 실습 #1

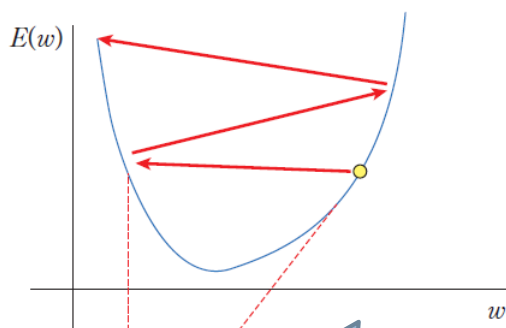




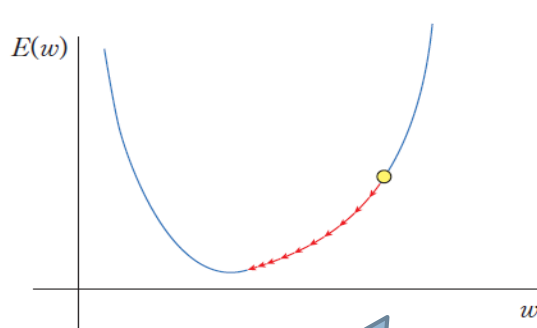
행렬로 미니 배치 구현하기

skip

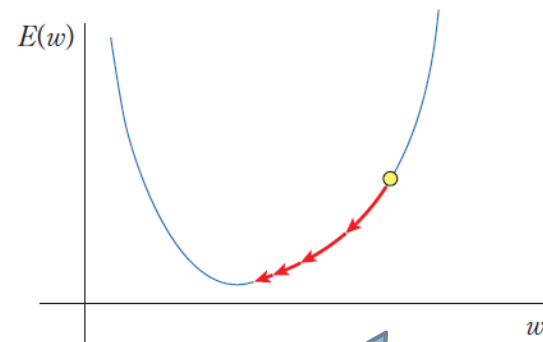
- 학습률(learning rate) : 한 번에 가중치를 얼마나 변경할 것인가
- 모델의 성능에 심대한 영향을 끼치지만 설정하기가 아주 어렵다
 - 학습률이 너무 높으면 오버슈팅. 불안정해지면 발산할 수.
 - 학습률이 너무 낮으면 아주 느리게 학습. 시간도 느려진다.
 - 적당한 학습률은 부드럽게 수렴



학습률이 너무 큰 경우



학습률이 너무 작은 경우

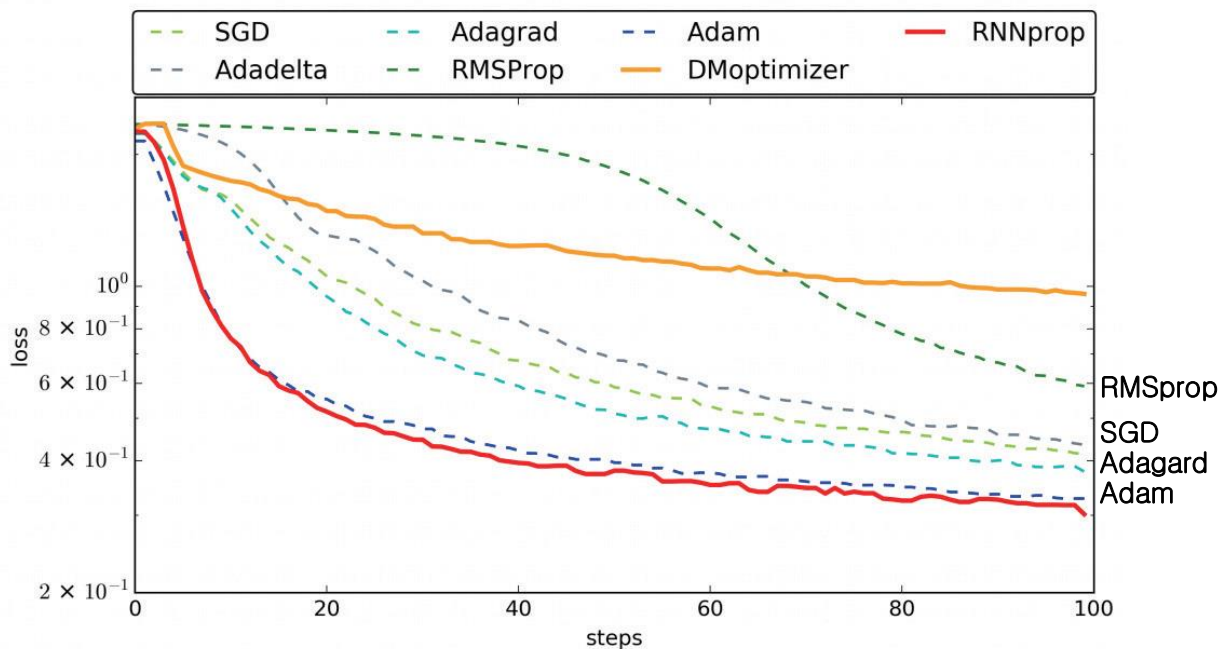


학습률이 적당한 경우



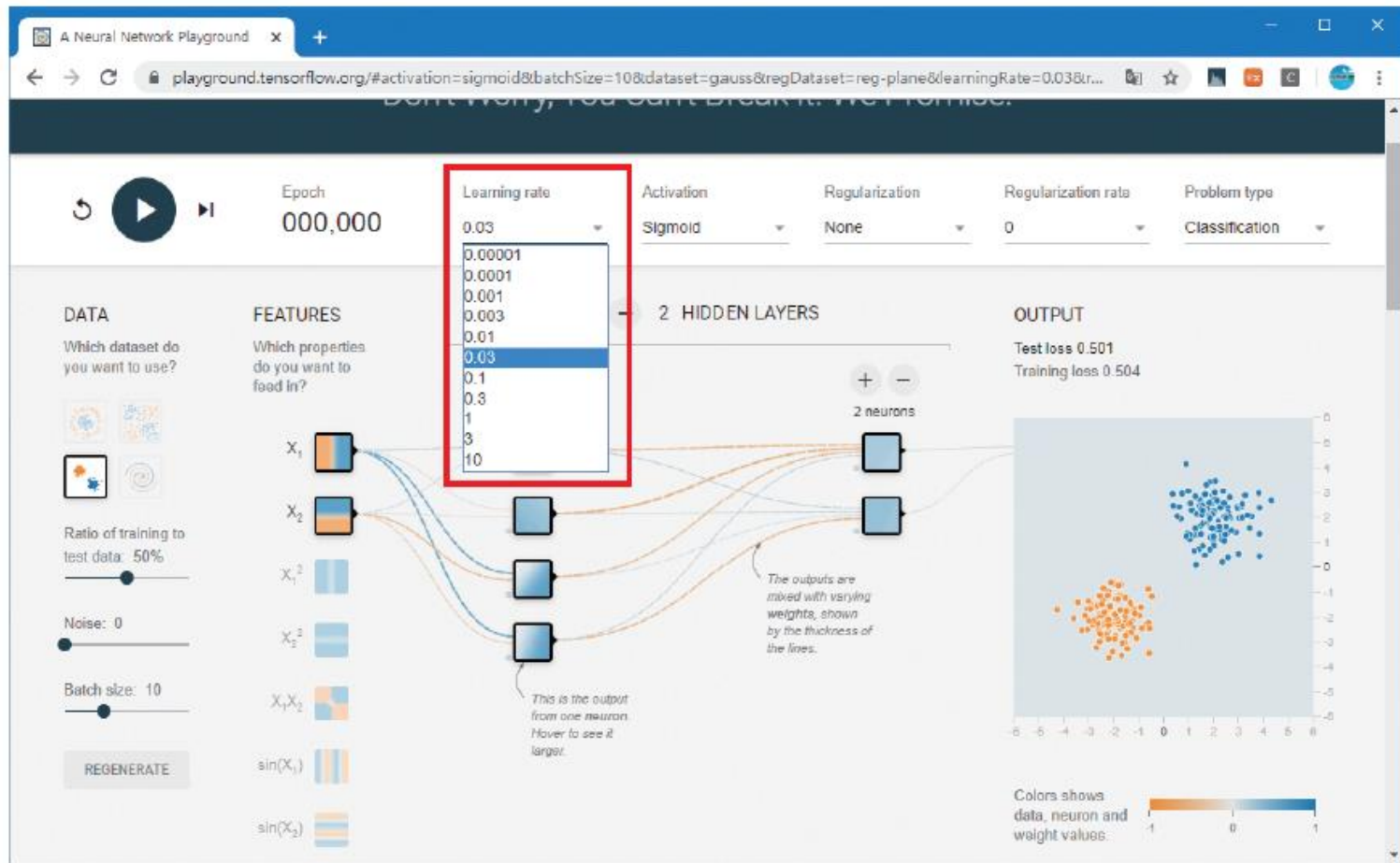
적절한 학습률 설정하기

- 학습률을 고정시키지 말고 현재 그래디언트 값이나 가중치의 크기, 학습의 정도 등을 고려하여 적응적 학습률로 설정
 - Adagrad : SGD 방법을 개량한 최적화 방법
 - RMSprop : 심층신경망을 위한 효과적인 최적화 알고리즘으로 증명
 - Adam(Adaptive Moment Estimation): 가장 인기있는 최적화 알고리즘



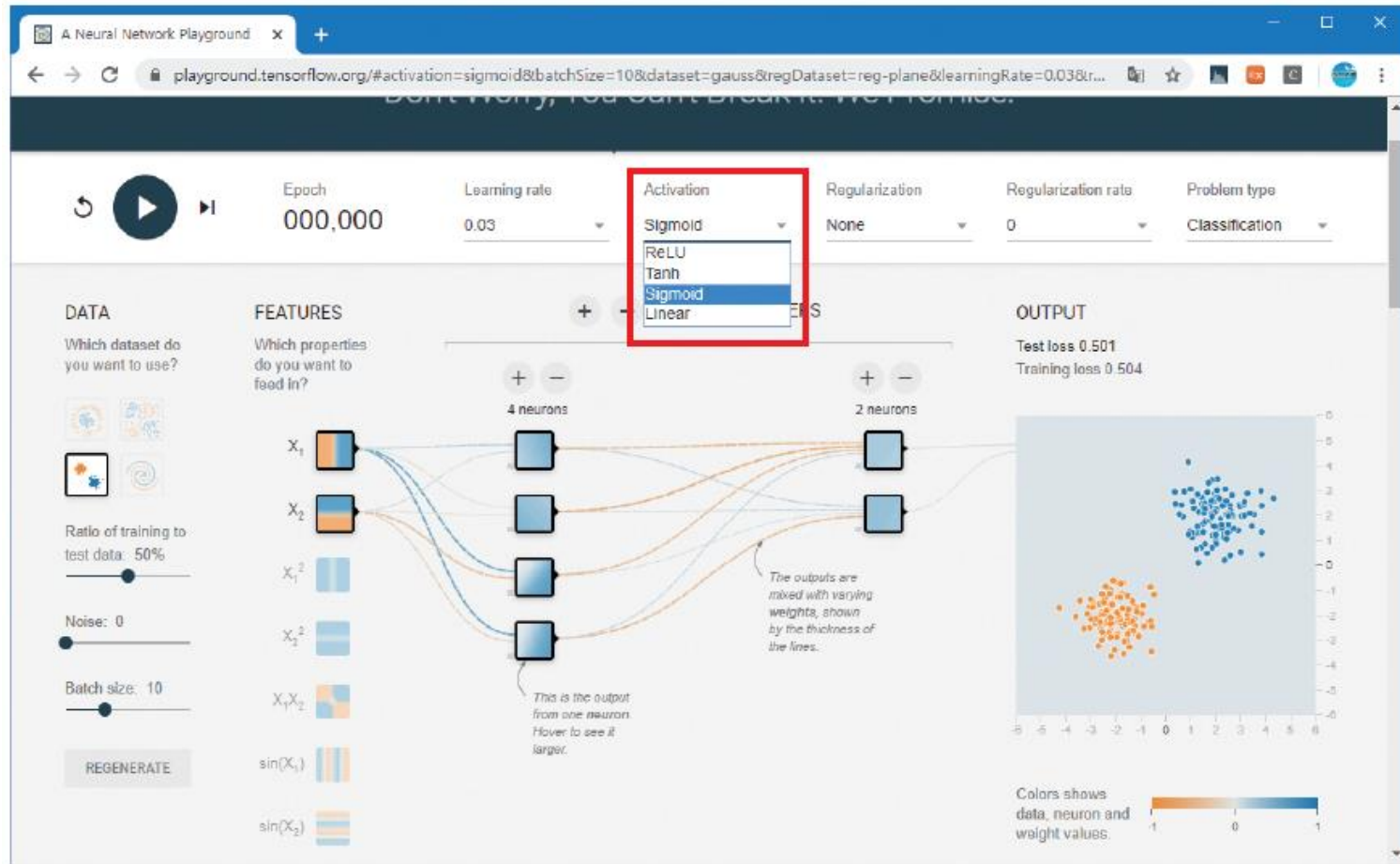


Lab: 학습과 배치 크기 실험



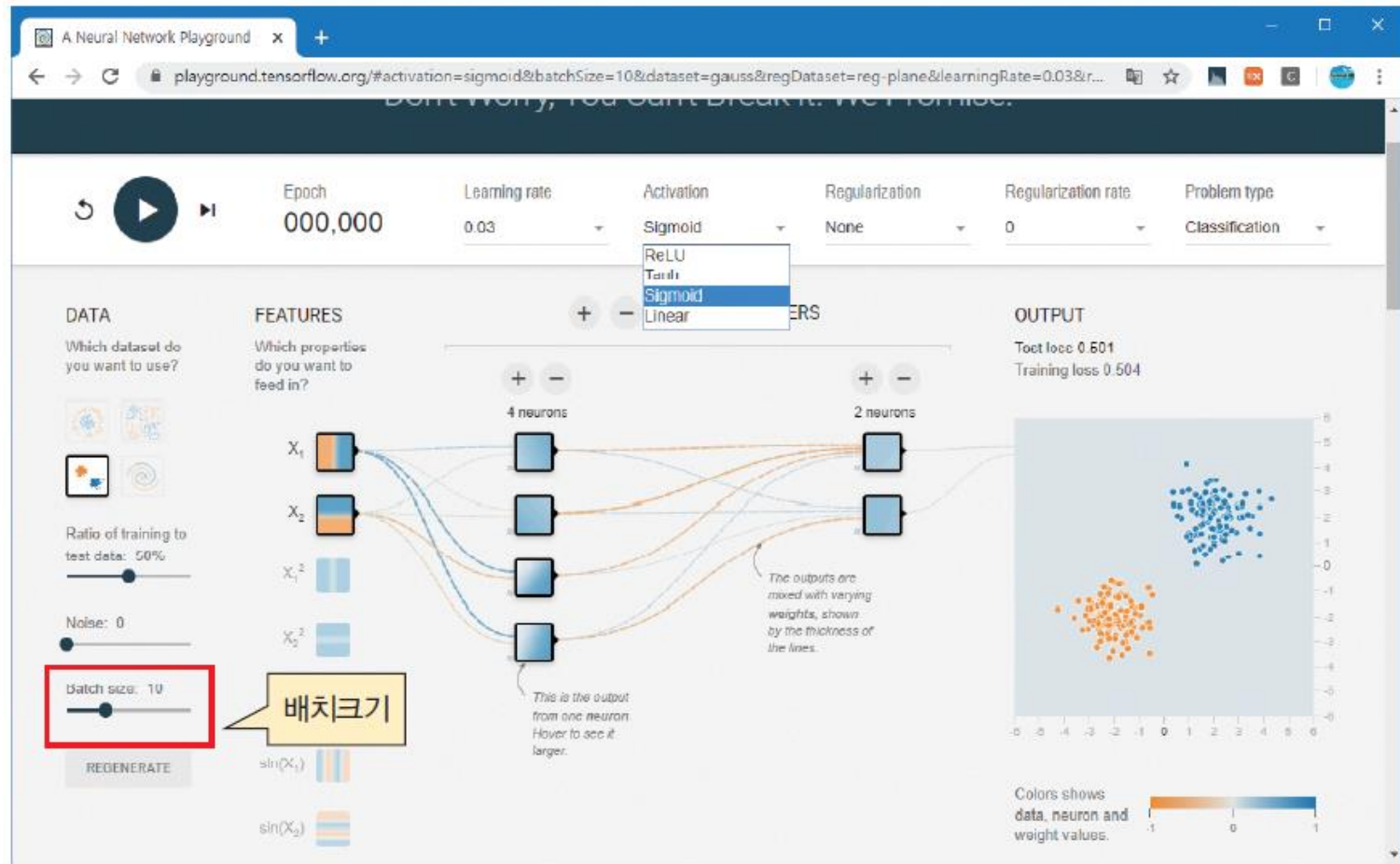


Lab: 학습과 배치 크기 실험





Lab: 학습과 배치크기 실험



텐서플로우와 케라스

- 텐서플로우(TensorFlow) : 딥러닝 프레임워크
- 텐서 = 다차원 배열, 플로우 = 데이터 흐름
- 연산을 나타내는 유닛 사이로 텐서들이 흘러다니는 개념

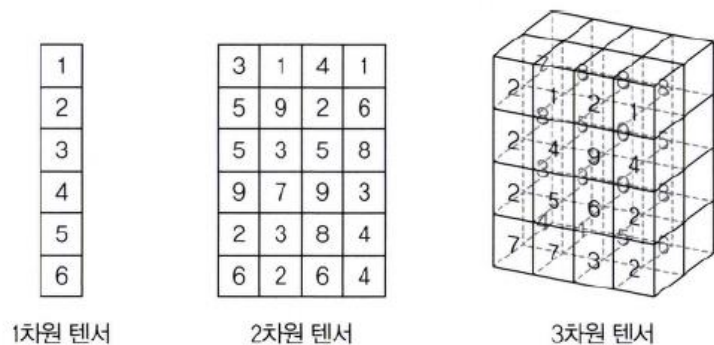


그림 7-3 텐서의 정의

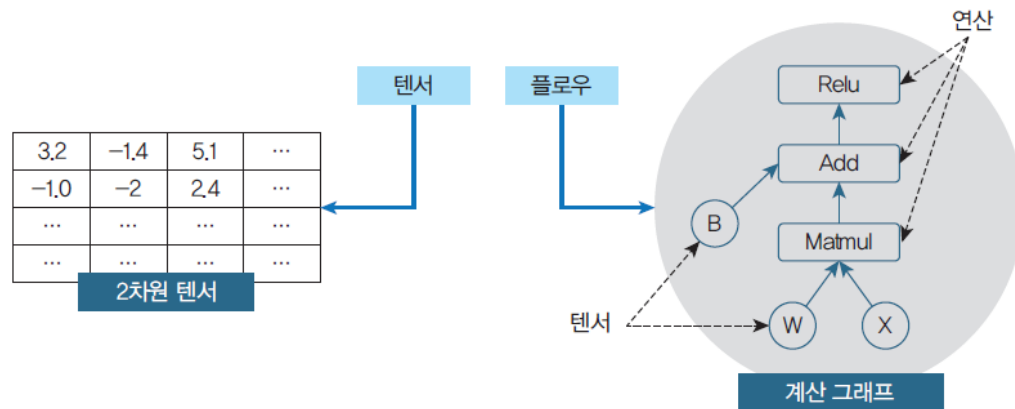
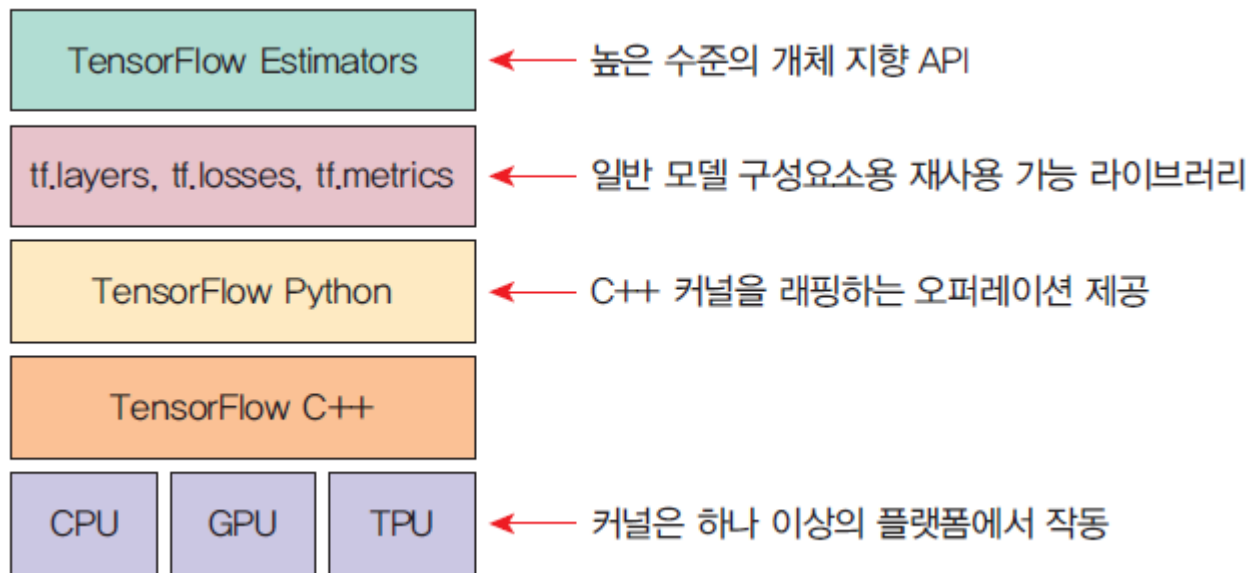


그림 7-4 텐서플로우의 개념

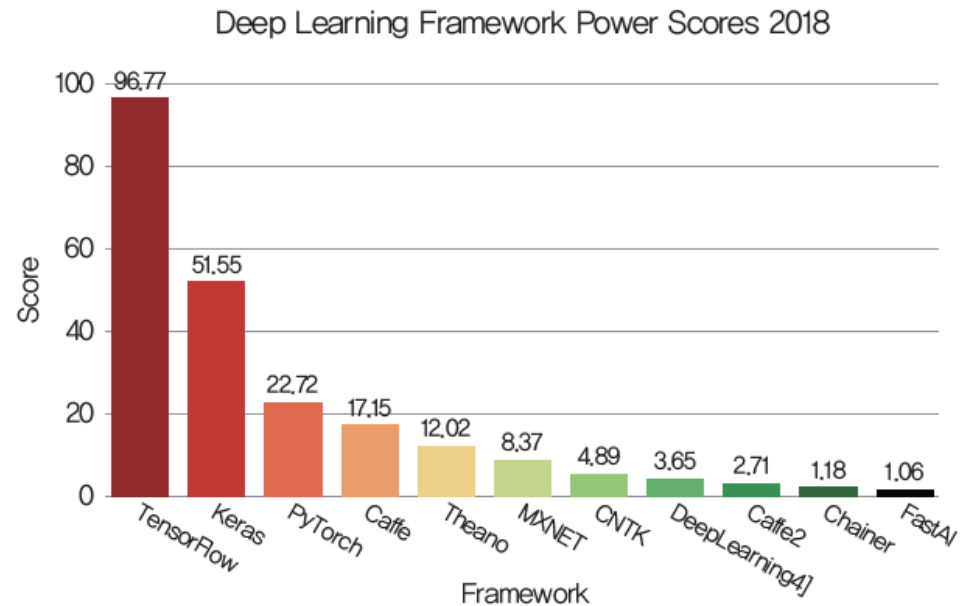
텐서플로우의 구조

- 텐서플로우는 기본적으로 저수준 **API** 이기 때문에 상당히 코드가 복잡하기 때문에 고수준 **API**를 제공하는 케라스 라이브러리를 사용



케라스(keras)

- 파이썬으로 작성. 고수준 딥러닝 API
- 텐서플로를 설치하면 케라스가 자동 설치
- 업계와 학계 모두에서 폭넓게 사용
- 파이토치(Pytorch)는 주로 연구자들이 선호





케라스로 신경망을 작성하는 절차

- 모델(model) : 케라스의 핵심 데이터 구조. 레이어를 구성하는 방법.
- Sequential 선형 스택 모델 : 가장 간단. 레이어를 선형으로 쌓을 수 있는 신경망 모델

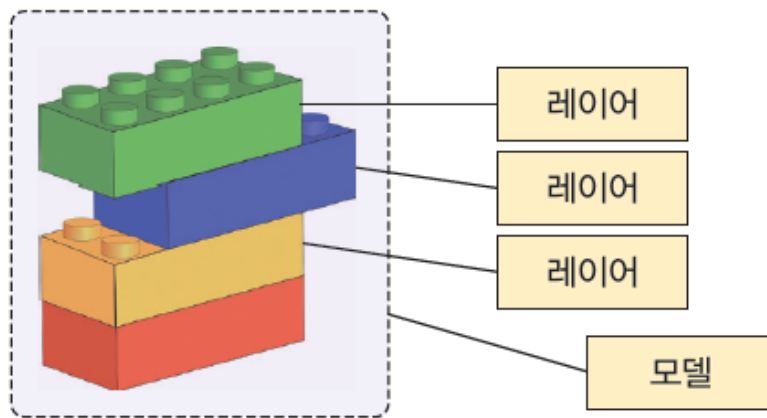


그림 7-5 케라스의 기본 개념



케라스로 신경망을 작성하는 절차

- ① 입력데이터와 정답 레이블을 준비한다. 넘파이 배열이나 파이썬 리스트 형식으로.

- ② Sequential 모델을 생성한다.

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

- ③ add() 함수를 이용하여 필요한 레이어를 추가한다.

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

- ④ compile() 함수를 호출하여 Sequential 모델을 컴파일한다.

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

- ⑤ fit()를 호출하여 학습을 수행한다

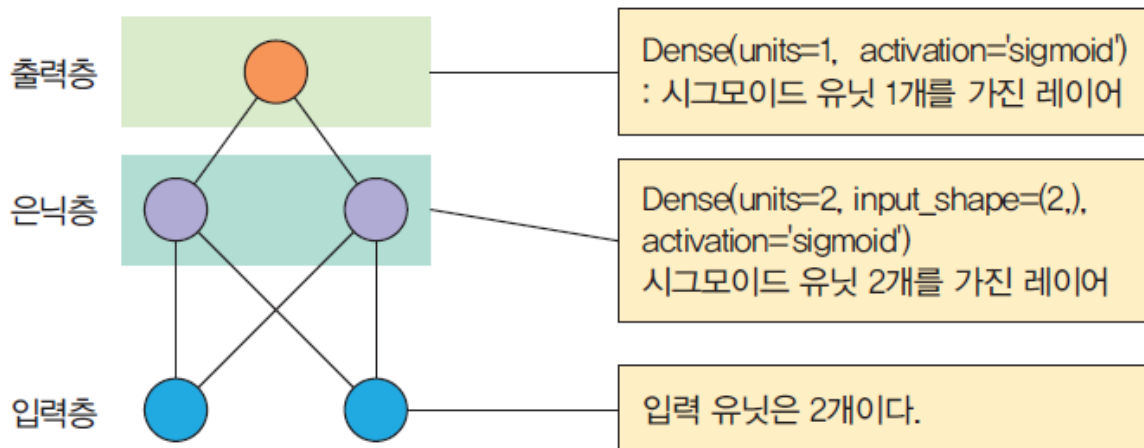
```
model.fit(X, y, batch_size=1, epochs=10000)
```

- ⑥ predict()를 호출하여 모델을 테스트한다.


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

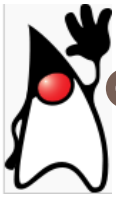


예제: XOR를 학습하는 MLP를 작성

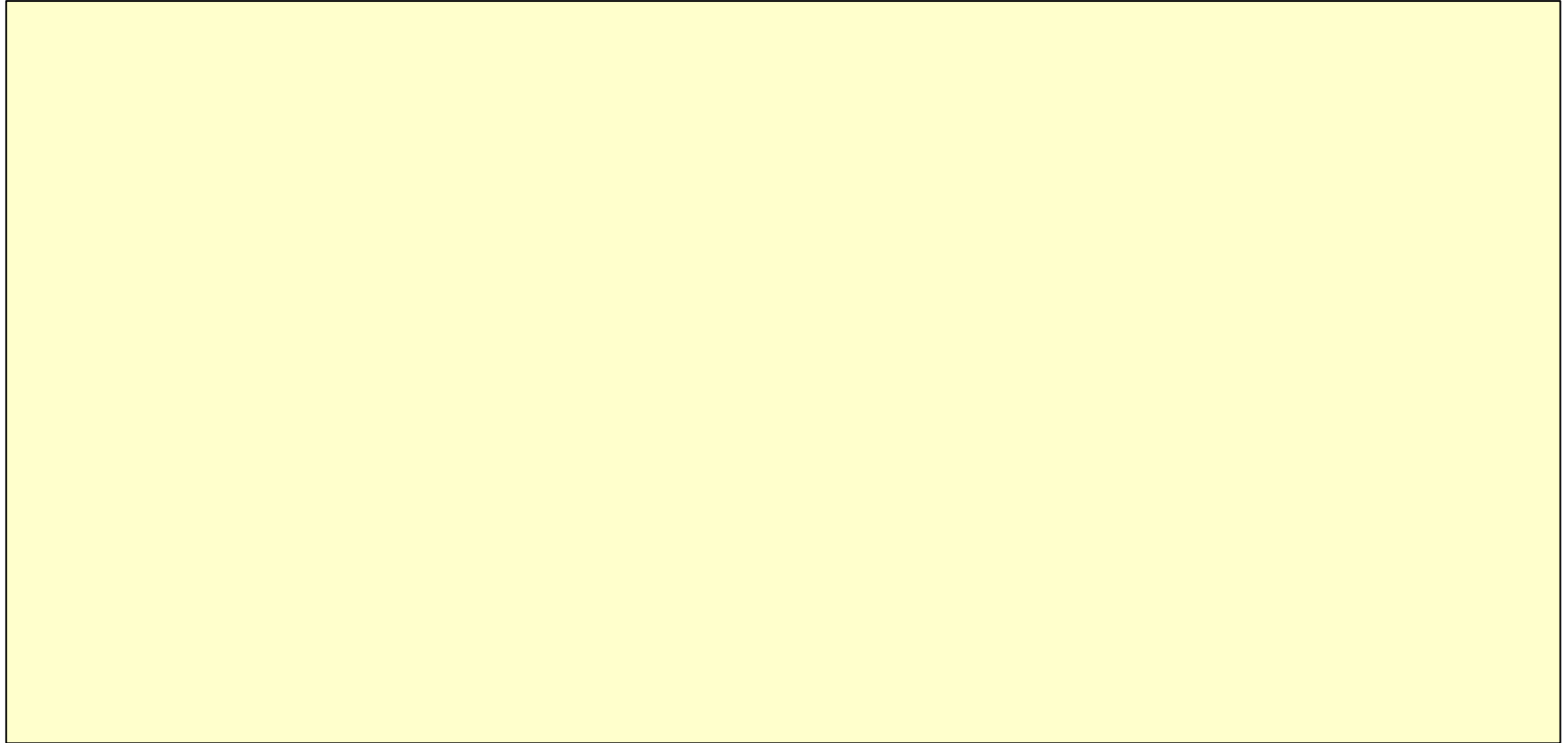


	x1	x2	y
샘플 #1	0	0	0
샘플 #2	0	1	1
샘플 #3	1	0	1
샘플 #4	1	1	0





예제: XOR를 학습하는 MLP를 작성



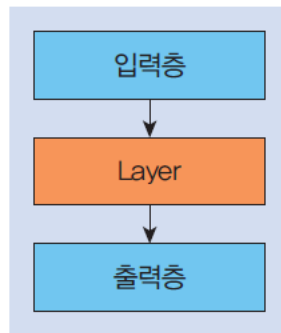


케라스를 사용하는 3가지 방법

(1) Sequential 모델 – add 사용

```
model = Sequential()

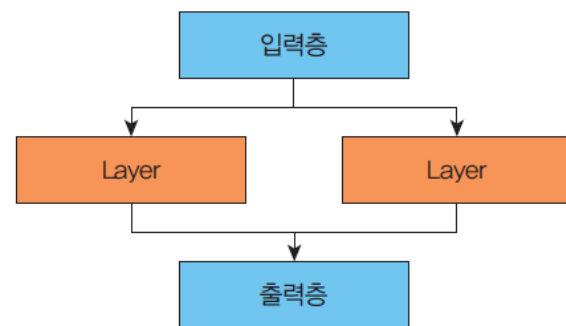
model.add(Dense(units=2, input_shape=(2,), activation='sigmoid'))
model.add(Dense(units=1, activation='sigmoid'))
```



(2) 함수형 API 사용 – 레이어와 레이어를 변수로 연결

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

model = Model(inputs=inputs, outputs=prediction)
```





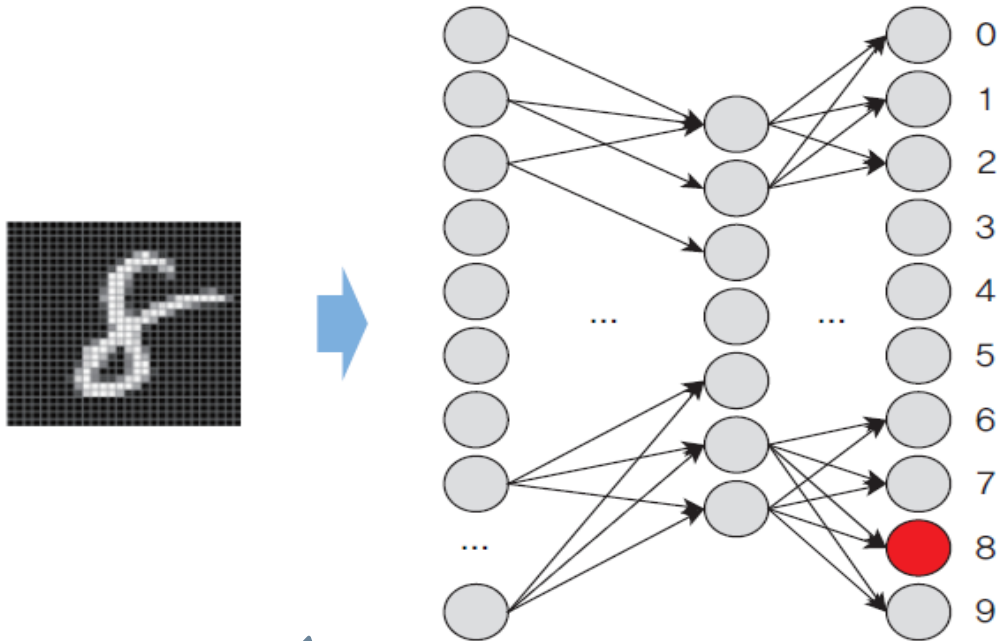
케라스를 사용하는 3가지 방법

(3) Model 클래스 사용

```
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 세트
- 6만개의 훈련 이미지와 1만개의 테스트 이미지



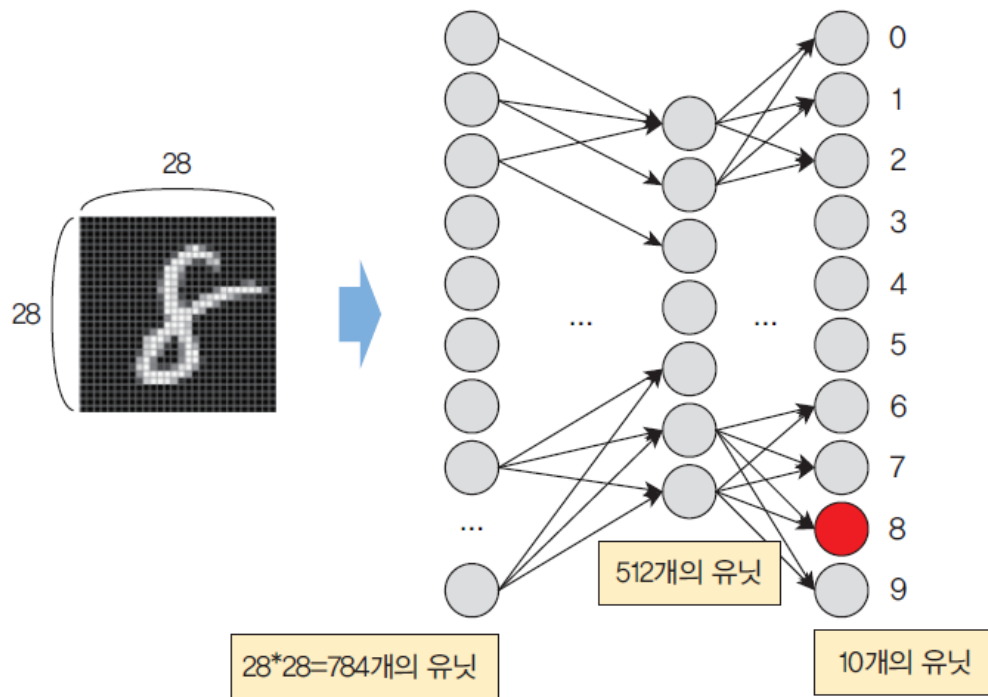
그림 7-7 MNIST 데이터

신경망 모델 구축하기

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

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

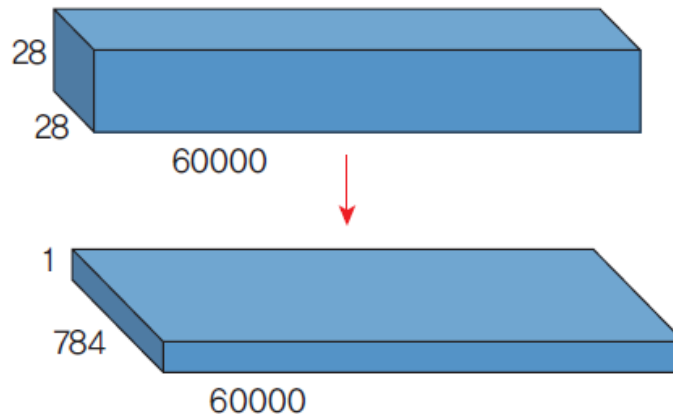
```
model.add(tf.keras.layers.Dense(10, activation='sigmoid'))
```





데이터 처리

```
train_images = train_images.reshape((60000, 784))  
train_images = train_images.astype('float32') / 255.0  
  
test_images = test_images.reshape((10000, 784))  
test_images = test_images.astype('float32') / 255.0
```





```
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]



케라스의 입력 데이터

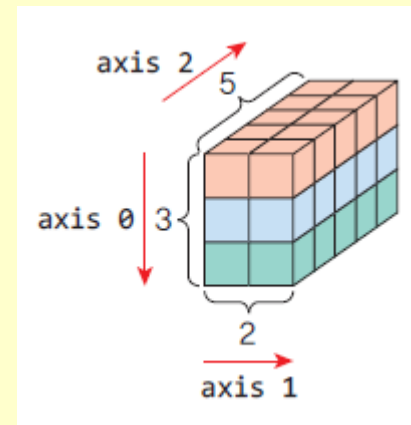
- 입력 데이터 유형

- ① 넘파이 배열 : 데이터가 메모리에 적재될 수 있다면 좋은 선택이다.
- ② **TensorFlow Dataset** 객체: 크기가 커서, 메모리에 한 번에 적재될 수 없는 경우에 디스크 또는 분산 파일 시스템에서 스트리밍될 수 있다.
- ③ 파이썬 제너레이터: **keras.utils.Sequence** 클래스는 하드 디스크에 위치한 파일을 읽어서 순차적으로 케라스 모델로 공급할 수 있다.

텐서(tensor)

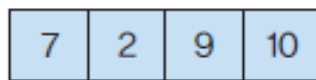
- 다차원 넘파이 배열
- 케라스를 비롯한 대부분 머신 러닝 시스템에서 기본 데이터 구조로 사용
- 데이터(실수)를 저장하는 컨테이너
- 예. 3차원 텐서

```
>>> 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)
```



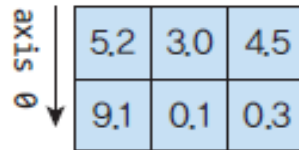
텐서의 속성

- 차원 : 축의 개수 = rank. ndim 속성
- 형상 : **shape**. 각 축으로 얼마나 데이터가 있는지
- 데이터 타입: **dtype**. 자료형



axis 0 →

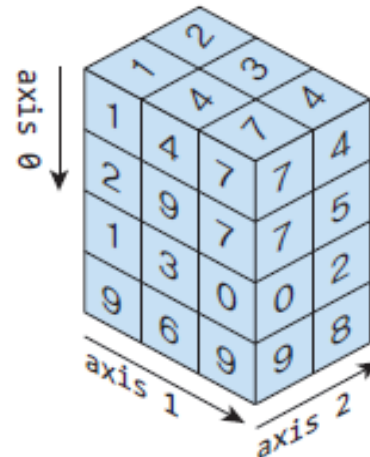
shape: (4,)



axis 0 ↓

axis 1 →

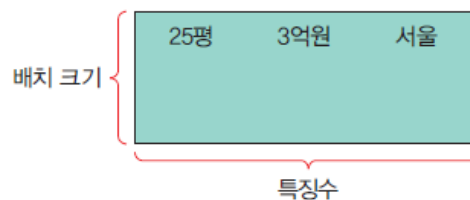
shape: (2, 3)



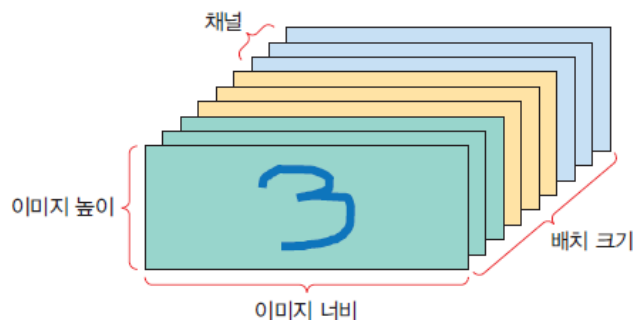
shape: (4, 3, 2)

호러 데이터의 형상

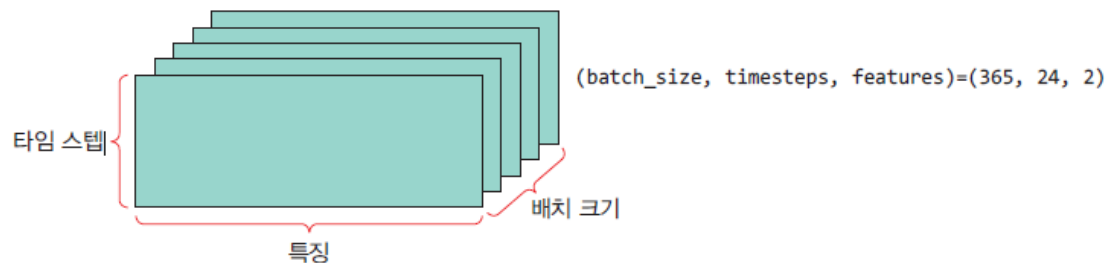
- 벡터 데이터: (배치 크기, 특징수)



- 이미지 데이터: (배치 크기, 이미지 높이, 이미지 너비, 채널수)



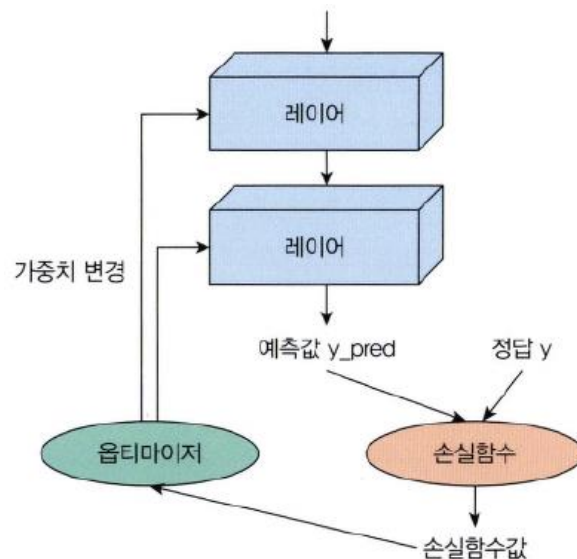
- 시계열 데이터: (배치 크기, 타임 스텝, 특징수)





케라스의 클래스들

- 모델: 하나의 신경망
- 레이어: 신경망에서 하나의 층
- 입력 데이터: 텐서플로우 텐서 형식
- 손실 함수: 신경망의 출력과 정답 레이블 간의 차이를 측정하는 함수
- 옵티마이저: 학습을 수행하는 최적화 알고리즘. 학습률과 모멘텀을 동적으로 변경.





Sequential 모델

- `compile(optimizer, loss=None, metrics=None)`: 훈련을 위해서 모델을 구성하는 메소드
 - `optimizer` : 옵티마이저 이름
 - `loss` : 손실함수 이름
 - `metrics` : 훈련 또는 테스트 과정에서 평가할 성능 측정 항목. 보통은 `metrics=['accuracy']`를 사용
- `fit(x=None, y=None, batch_size=None, epochs=1, verbose=1)`: 훈련(학습) 메소드
 - `x` : 훈련 샘플을 저장하는 2차원 넘파이 배열
 - `y` : 정답 레이블을 저장하는 1차원 넘파이 배열
 - `batch_size` : 가중치를 업데이트할 때 처리하는 샘플의 수
 - `epochs` : `x`와 `y` 데이터 세트를 반복하는 횟수
- `evaluate(x=None, y=None)`: 평가. 테스트 모드에서 모델의 손실 함수 값과 측정 항목 값을 반환
- `predict(x, batch_size=None)`: 입력 샘플에 대한 예측값을 생성
- `add(layer)`: 레이어를 모델에 추가



- **Input(shape, batch_size, name):** 입력을 받아서 케라스 텐서를 생성하는 객체
 - **shape** : 입력의 형상을 나타내는 정수 튜플. 예를 들어 **shape=(32,)**는 입력이 32차원 벡터라는 의미
 - **batch_size** : 배치 크기
- **Dense(units, activation=None, use_bias=True, input_shape):** 유닛들이 전부 연결된 레이어
 - **units** : 유닛의 개수
 - **activation** : 유닛의 활성화 함수
 - **use_bias** : 유닛에서 바이어스 사용 유무
 - **input_shape** : 입력의 형태
- **Embedding(input_dim, output_dim):** 자연어 처리의 첫 단계에서 사용되는 레이어

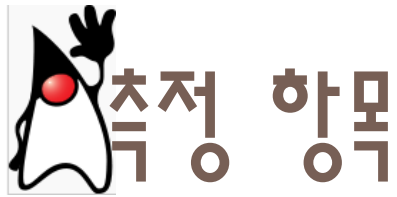


- 손실함수 : 회귀문제에서는 **MSE**. 분류문제에서는 **entropy** 확률분포 사용
- **MeanSquaredError**: 정답 레이블과 예측값 사이의 평균 제곱 오차를 계산
- **BinaryCrossentropy**: 정답 레이블과 예측 레이블 간의 교차 엔트로피 손실을 계산. **레이블 분류가 두 개인 경우**(예를 들어서 강아지, 강아지 아님)에 사용
- **CategoricalCrossentropy**: 정답 레이블과 예측 레이블 간의 교차 엔트로피 손실을 계산. **여러 개의 레이블 부류**(예를 들어서 강아지, 고양이, 호랑이)가 있을 때 사용. **정답 레이블은 원핫 인코딩으로 제공되어야 한다.**
- **SparseCategoricalCrossentropy**: **CategoricalCrossentropy**와 목적은 같다. **정답 레이블은 정수로 제공되어야 한다.**



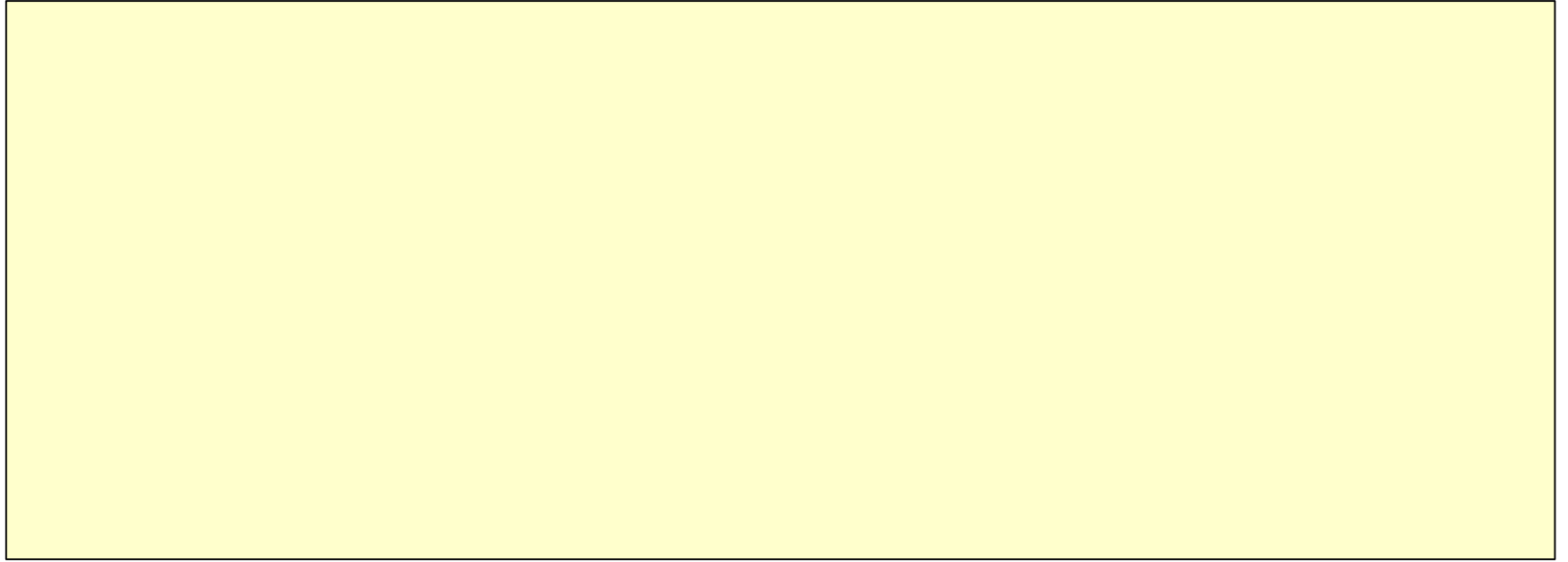
	메서드 표현	문자열 표현
회귀	MeanSquaredError	mean_squared_error
분류(두개)	BinaryCrossentropy	binary_crossentropy
분류(여러개)	CategoricalCrossentropy	categorical_crossentropy
분류(여러개)	SparseCategoricalCrossentropy	Sparse_categorical_crossentropy

참고. 예제 코드



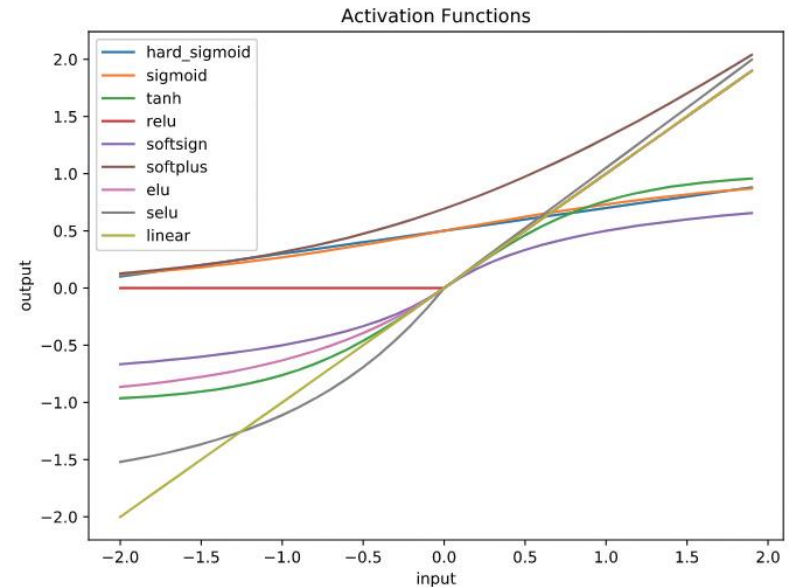
- 모델의 성능 평가
- Accuracy: 정확도. 예측값이 정답 레이블과 같은 횟수를 계산.
- categorical_accuracy: 범주형 정확도. 예측값이 원-핫 레이블과 일치하는 빈도를 계산.

- 손실함수를 미분하여 가중치를 변경하는 객체.
- 그래디언트의 크기에 따라 학습률을 다르게 하는 방법을 많이 사용
- SGD: 확률적 경사 하강법(Stochastic Gradient Descent). 고전적인 최적화 알고리즘. 얇은 신경망에 적합
- Adagrad: 가변 학습률을 사용하는 방법
- Adadelat: 감소하는 학습률 문제를 처리하는 Adagrad의 변형
- RMSprop: Adagrad에 대한 수정판.
- Adam: RMSprop + 모멘텀. 가장 인기있는 최적화 알고리즘
- Adam -> RMSprop -> SGD 순으로 많이 사용
- 성능비교 예. p290. 문제 10번



- sigmoid
- relu(Rectified Linear Unit)
- softmax
- tanh
- selu(Scaled Exponential Linear Unit)
- softplus

← p.191





하이퍼 매개변수(hyper parameter)

- 이미 있는 매개변수(parameter : 가중치, 바이어스)와 구분하기 위해 하이퍼라 부름
- 은닉층의 개수 : 입력층과 출력층 사이의 레이어. 간단한 법칙-테스트 세트의 오차가 더 이상 개선되지 않을 때까지 계속 레이어를 추가
- 유닛 개수 : 많은 것이 좋다. 유닛의 개수에 따라서 정확도가 증가
- 학습률 : 적응적 학습률이 좋다.
- 모멘텀 : 학습 도중에 진동을 방지하는데 도움. 일반적으로 0.5에서 0.9사이
- 미니 배치 크기 : 기본값은 32정도. 64, 128, 256 등을 사용
- 에포크 수 : 전체 데이터 세트를 반복하는 횟수

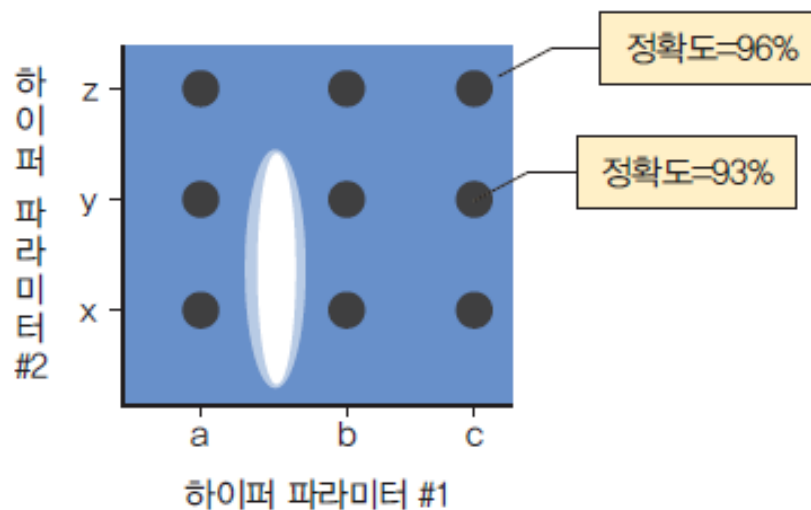


하이퍼 매개 변수를 찾는 방법

- 기본값 사용: 라이브러리 개발자가 설정한 기본값을 그대로 사용
- 수동 검색: 사용자가 하이퍼 매개변수를 지정
- **그리드 검색**: 격자 형태로 하이퍼 매개변수를 변경하면서 성능을 측정하는 방법
- 랜덤 검색: 랜덤으로 검색

그리드 검색

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





- 알고리즘
 - n 차원 그리드를 정의한다. 각 차원은 하나의 하이퍼 매개변수를 의미한다. 예를 들어 $n=(\text{learning_rate}, \text{dropout_rate}, \text{batch_size})$ 와 같이 정의할 수 있다.
 - 각 차원에 대하여 가능한 범위를 정의한다. 예를 들어 $\text{batch_size}=[8, 16, 32, 64, 128, 256, 512]$ 라고 지정할 수 있다.
 - 모든 가능한 조합에 대하여 신경망을 돌려본다. 그 중에서 가장 좋은 결과를 내는 조합을 선택한다. $C1=(0.2, 0.1, 32) \rightarrow \text{정확도} = 80\%$, $C2=(0.3, 0.2, 64) \rightarrow \text{정확도} = 93\%$ 등
- 단점 : 하이퍼 매개변수 후보가 많으면 시간이 매우 오래 걸린다.



그리드 검색 예제

