

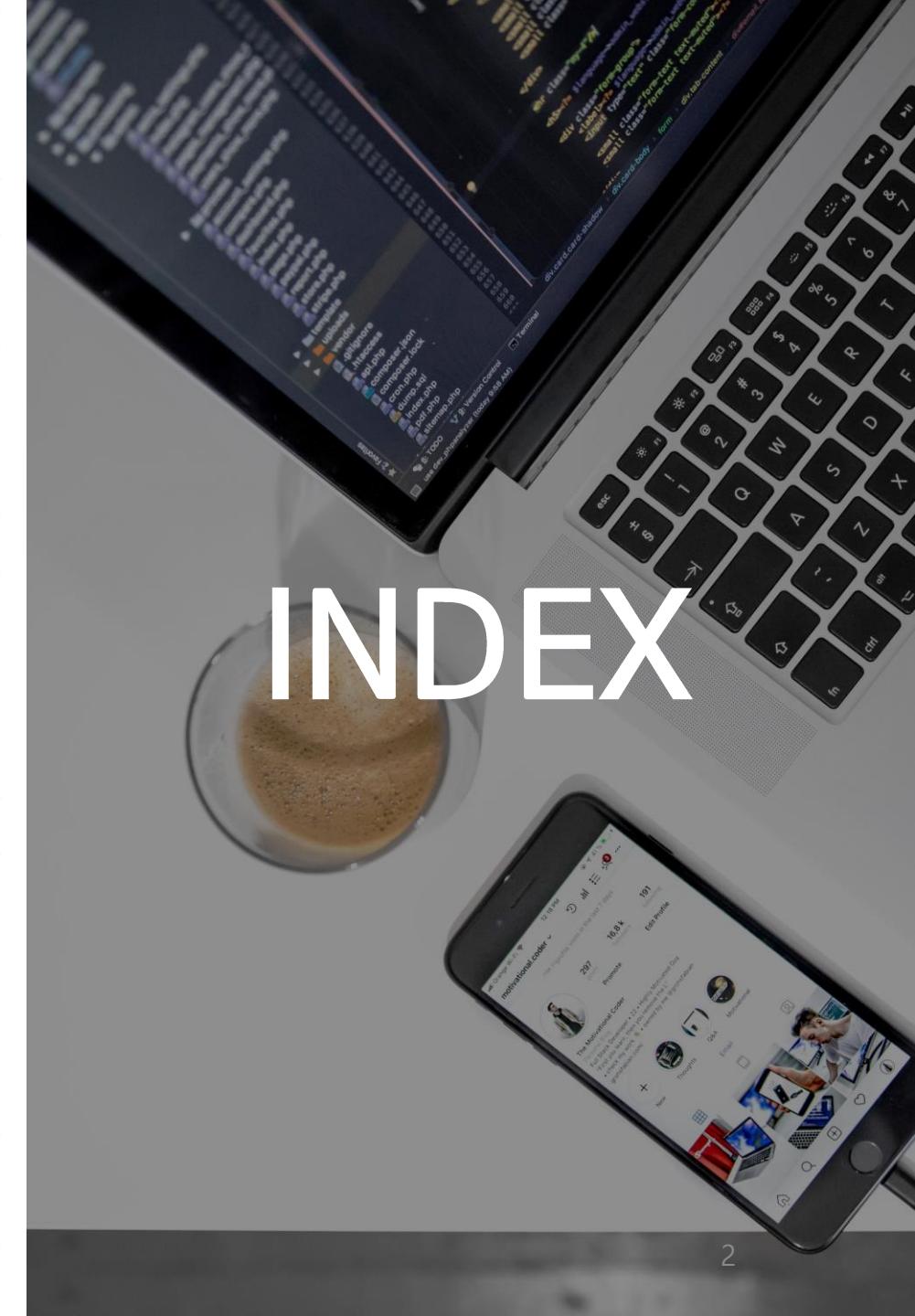


HAJ Lecture

Chapter 10. 케라스를 사용한 인공 신경망 소개
Chapter 11. 심층 신경망 훈련하기

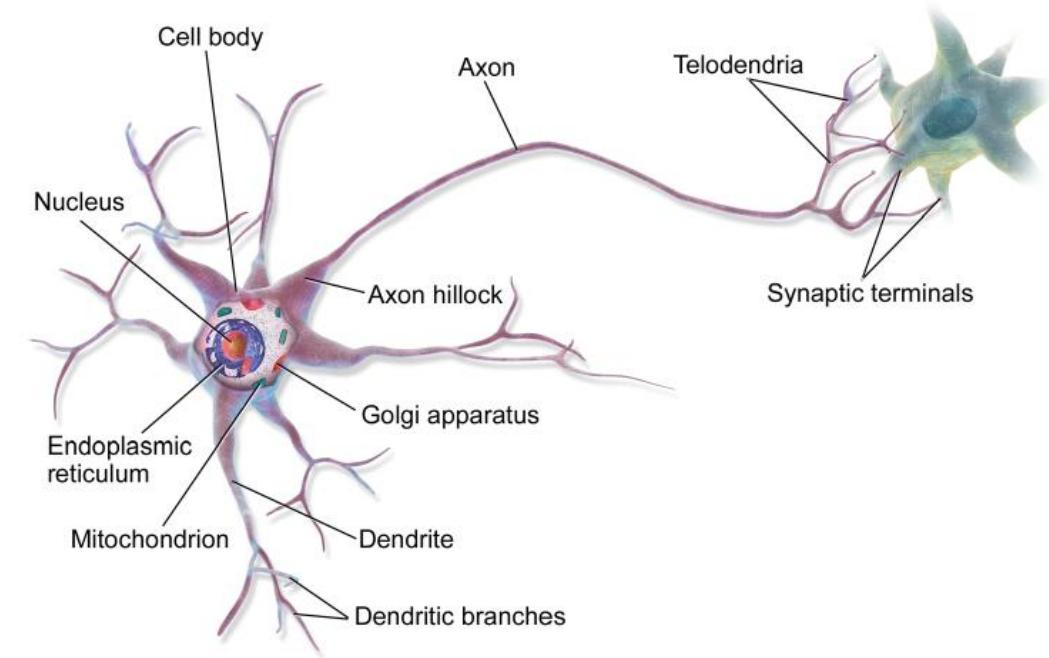
Hands-On Machine Learning
with Scikit-Learn, Keras & TensorFlow

- | 10.1 생물학적 뉴런에서 인공 뉴런까지
- | 10.2 케라스로 다중 퍼셉트론 구현하기
- | 10.3 신경망 하이퍼파라미터 튜닝하기



10.1 생물학적 뉴런에서 인공 뉴런까지

- 수상돌기(dendrite): 나뭇가지 모양 돌기
→ 입력(input)
- 축삭돌기(axon): 아주 긴 돌기 한 개
→ 출력(output)
- 활동전위 또는 신호 등의 전기적 자극을 받으면 축삭돌기를 따라 시냅스가 화학적 신호를 발생하게 함



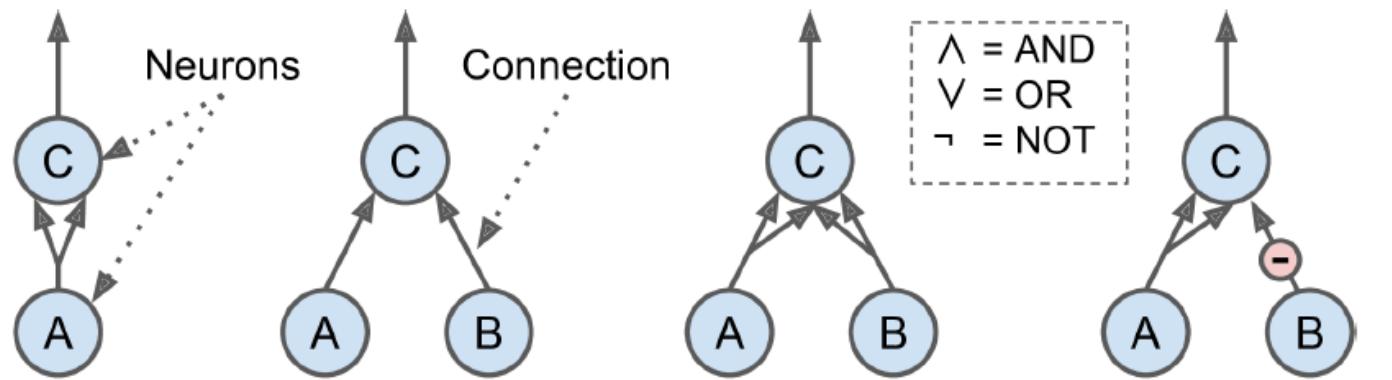
10.1 생물학적 뉴런에서 인공 뉴런까지

인공 뉴런(artificial neuron)

생물학뉴런 개념에 착안하여 만들어진 개념

→ 하나 이상의 이진 입력과 이진 출력 하나를 가짐

- 입력이 일정 개수만큼 활성화되었을 때 출력을 내보낸다.
- 인공 뉴런 네트워크를 만들어 어떤 논리 명제도 계산할 수 있음이 밝혀졌다.



$$C = A$$

항등함수

$$C = A \wedge B$$

논리곱 연산

$$C = A \vee B$$

논리합 연산

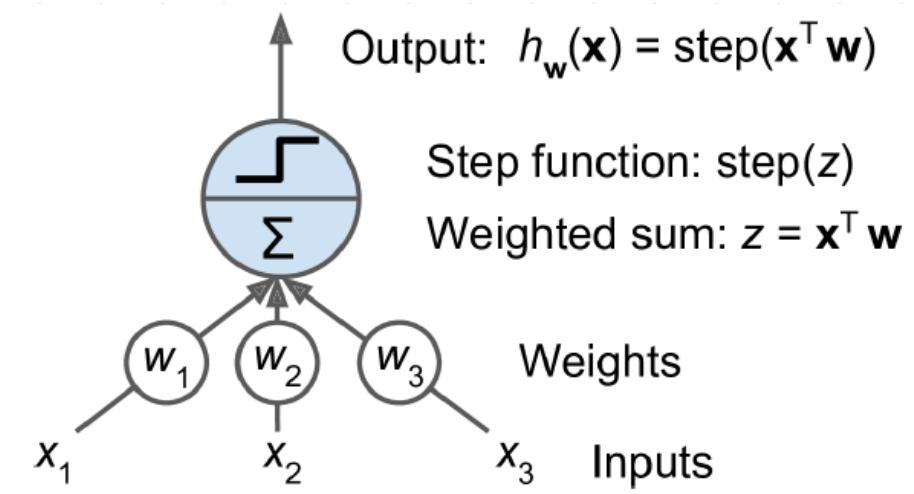
$$C = A \wedge \neg B$$

복잡한 논리명제

10.1 생물학적 뉴런에서 인공 뉴런까지

퍼셉트론(perceptron)

- 가장 간단한 인공 신경망의 형태
- TLU 또는 LTU라고 불리는 다른 형태의 인공 뉴런을 기반으로 함
- 입력과 출력이 어떤 숫자이고(non-binary), 각각의 입력 연결은 가중치와 연관되어 있다.
- 입력의 가중치 합을 계산 후 그것을 계단함수에 넣어 결과를 출력한다.

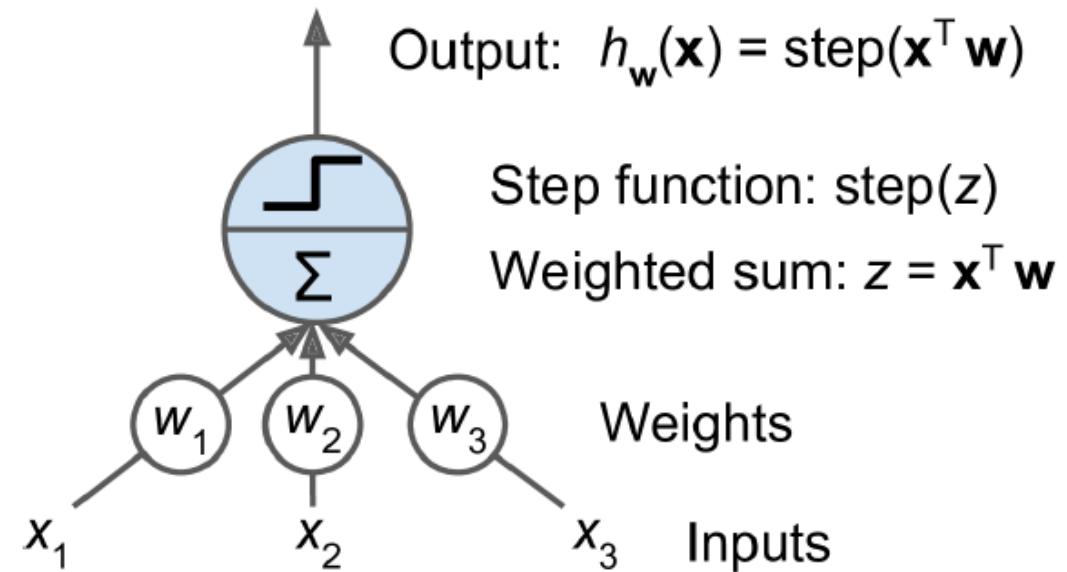


10.1 생물학적 뉴런에서 인공 뉴런까지

퍼셉트론(perceptron)

$$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n = \mathbf{x}^T \mathbf{w}$$

$$h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z), z = \mathbf{x}^T \mathbf{w}$$



계단함수(step function)

1) Heaviside step function

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

2) Sign function

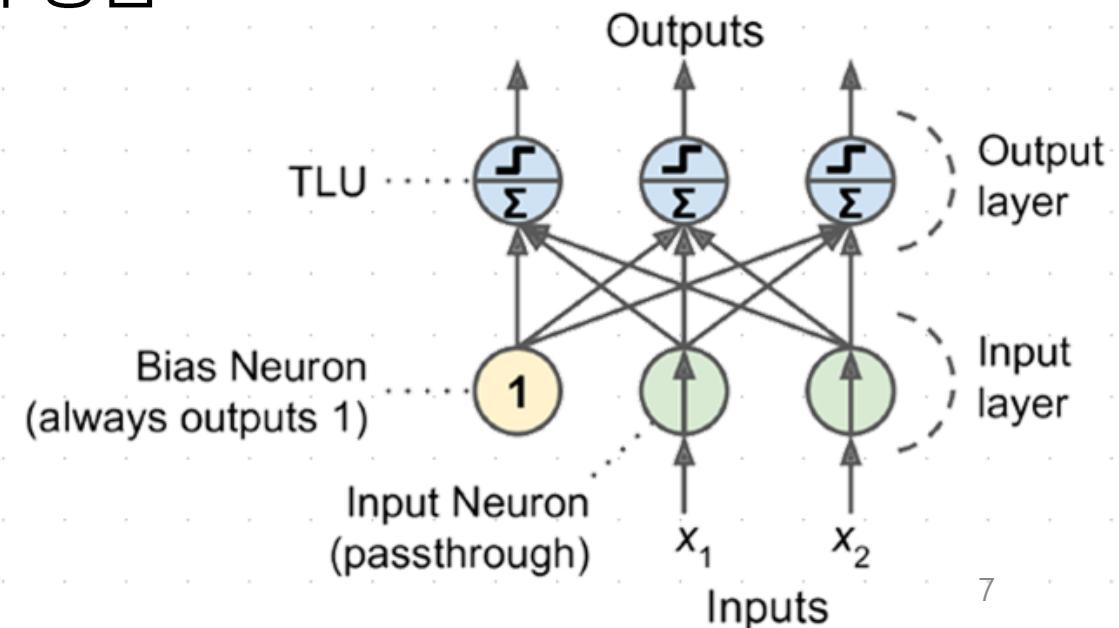
$$\text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

10.1 생물학적 뉴런에서 인공 뉴런까지

퍼셉트론(perceptron)

- 층이 하나뿐인 TLU로 구성된다.
- 입력: 입력뉴런이라고 불리는 특별한 통과 뉴런에 주입됨
→ 이 뉴런은 어떤 입력이 주입되든 그냥 출력으로 통과시킴
- 입력층(input layer): 모두 입력뉴런으로 구성됨

e.g. 입력 2개, 출력 3개 구성 퍼셉트론



| 10.1 생물학적 뉴런에서 인공 뉴런까지

Q. 퍼셉트론은 어떻게 훈련될까?

- Inspired by 헤브의 규칙

: “생물학적 뉴런이 다른 뉴런을 활성화시킬 때 둘 사이의 연결이 더 강해짐”

→ 두 뉴런이 동시에 활성화 될 때 연결 가중치가 증가

- 퍼셉트론 학습 규칙(가중치 업데이트)

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

10.1 생물학적 뉴런에서 인공 뉴런까지

In Scikit-Learn,

하나의 TLU 네트워크를 구현한 **Perceptron** 객체를 제공한다.

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)] # petal length, petal width
y = (iris.target == 0).astype(np.int) # Iris Setosa?

per_clf = Perceptron()
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

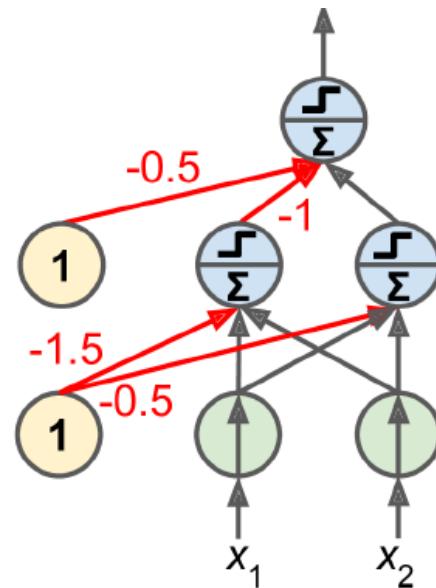
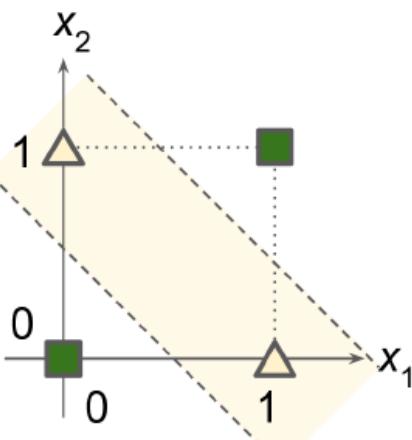
확률적 경사 하강법과 비슷해보임.

그러나,
퍼셉트론은 클래스 확률을 제공하지
않고 고정된 임곗값을 기준으로
예측을 만든다.
→ 퍼셉트론보다 로지스틱 회귀가
선호된다.

10.1 생물학적 뉴런에서 인공 뉴런까지

다중 퍼셉트론(MLP; multi layered perceptron)

- 퍼셉트론을 여러 개 쌓아올리면 일부 제약을 줄일 수 있음이 밝혀짐
- XOR 문제를 풀 수 있다.(대표적 사례)

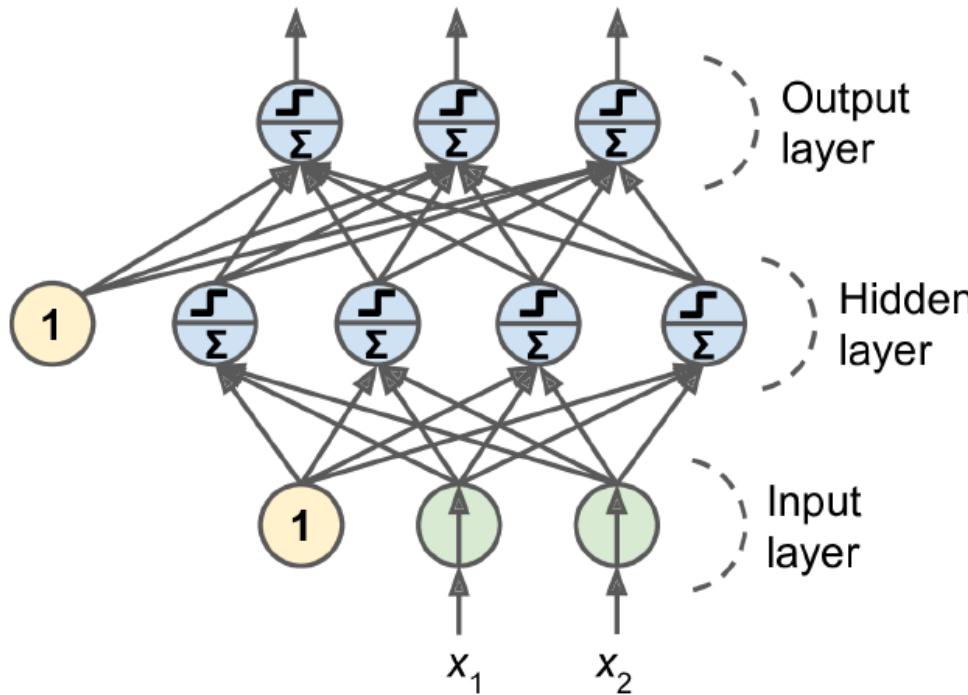


- 입력이 $(0, 0)$ 이나 $(1, 1)$ 일 때
→ 네트워크는 0을 출력한다.
- 입력이 $(0, 1)$ 이나 $(1, 0)$ 일 때
→ 네트워크는 1을 출력한다.
- 빨간색 선을 제외한 다른 모든 연결의 가중치는 10이다.

10.1 생물학적 뉴런에서 인공 뉴런까지

다중 퍼셉트론(MLP; multi layered perceptron)

- 입력층, 하나 이상의 TLU층인 은닉층, 그리고 출력층



출력층

심층 신경망(DNN; deep neural network)

은닉층

은닉층을 여러 개 쌓아 올린
인공 신경망

입력층

→ 딥러닝: 심층 신경망을 연구하는 분야

| 10.1 생물학적 뉴런에서 인공 뉴런까지

역전파(Back Propagation)

- 효율적인 기법으로 gradient를 자동으로 계산하는 경사 하강법
- 네트워크를 총 2번 통과하는 것만으로 모든 모델 파라미터에 대한 네트워크 오차의 gradient를 계산할 수 있다.(정방향 1회+역방향 1회)
- 자동 미분(automatic differentiation)
자동으로 gradient를 계산하는 것
역전파에서는 후진모드 자동 미분을 사용
(→ 미분할 함수가 변수가 많고 출력이 적은 경우 유리하다.)

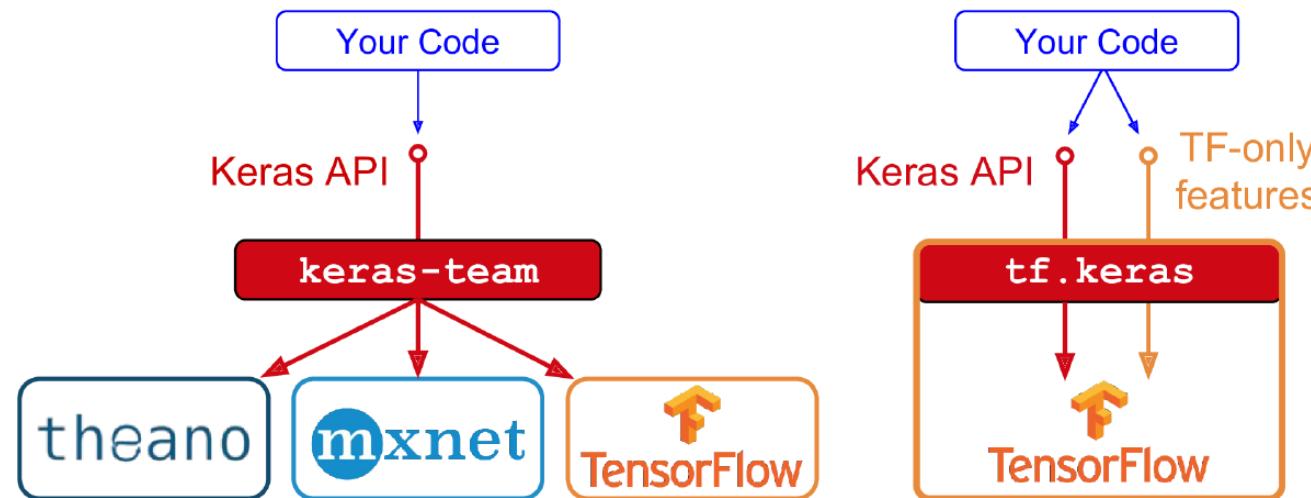
| 10.1 생물학적 뉴런에서 인공 뉴런까지

- 에포크(epoch): 미니배치를 기본단위로 하나씩 진행시켜 전체 훈련세트를 처리하는데 각각의 반복을 일컫는 말
- 정방향 계산(forward pass)
각각의 미니배치가 입력층 → 은닉층 → … → 출력층에서의 계산까지 진행하는 것
- 연쇄법칙(chain rule) in Calculus

| 10.2 케라스로 다층 퍼셉트론 구현하기

케라스(Keras)

모든 종류의 신경망을 손쉽게 만들고 훈련/평가/실행할 수 있는
고수준 딥러닝 API



```
$ python3 -m pip install --upgrade tensorflow
>>> import tensorflow as tf
>>> from tensorflow import keras
```

10.2 케라스로 다층 퍼셉트론 구현하기

시퀀셜 API를 사용하여 이미지 분류기 만들기

1] 캐라스를 사용하여 데이터셋 적재하기

```
fashion_mnist = keras.datasets.fashion_mnist  
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

입력 특성 scale 조정하기 → 픽셀 강도 [0,255] → [0,1]

```
X_valid, X_train = X_train_full[:5000] / 255.0, X_train_full[5000:] / 255.0  
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

label에 해당하는 아이템을 나타내기 위한 클래스 이름 리스트

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```



```
>>> class_names[y_train[0]]  
'Coat'
```

10.2 케라스로 다중 퍼셉트론 구현하기

시퀀셜 API를 사용하여 이미지 분류기 만들기

2] 시퀀셜 API를 사용하여 모델 만들기

```
model = keras.models.Sequential()  
model.add(keras.layers.Flatten(input_shape=[28, 28]))  
model.add(keras.layers.Dense(300, activation="relu"))  
model.add(keras.layers.Dense(100, activation="relu"))  
model.add(keras.layers.Dense(10, activation="softmax"))  
  
>>> model.summary()
```

Layer (type)	Output Shape	Param #
=====		
flatten_1 (Flatten)	(None, 784)	0
dense_3 (Dense)	(None, 300)	235500
dense_4 (Dense)	(None, 100)	30100
dense_5 (Dense)	(None, 10)	1010
=====		
Total params: 266,610		
Trainable params: 266,610		
Non-trainable params: 0		

2개의 은닉층으로 구성된
분류용 다중 퍼셉트론

Sequential
순서대로 연결된 층을 일렬로
쌓아서 구성함

summary() 메소드를 통한
모델의 구조 파악하기

| 10.2 케라스로 다층 퍼셉트론 구현하기

시퀀셜 API를 사용하여 이미지 분류기 만들기

3] 모델 컴파일

compile() 메서드를 손실함수, Optimizer 등을 지정하여 호출

```
model.compile(loss="sparse_categorical_crossentropy",
               optimizer="sgd",
               metrics=["accuracy"])
```

4] 모델 훈련과 평가

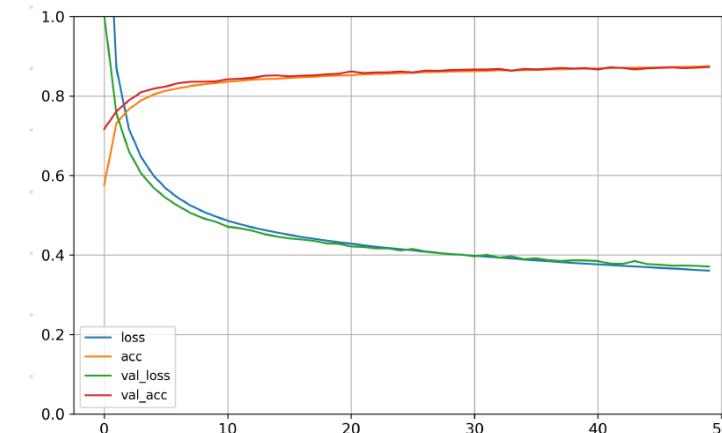
fit() 메서드를 호출하여 모델 훈련을 진행

아래의 코드는 학습 곡선을 그리는 코드임

```
import pandas as pd

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1) # set the vertical range to [0-1]
plt.show()
```

```
>>> history = model.fit(X_train, y_train, epochs=30,
                        validation_data=(X_valid, y_valid))
```



| 10.2 케라스로 다층 퍼셉트론 구현하기

시퀀셜 API를 사용하여 이미지 분류기 만들기

5] 모델을 사용하여 예측하기

`predict()` 메서드를 사용해 새로운 샘플에 대해 예측을 만들 수 있다.

```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0.   , 0.   , 0.   , 0.   , 0.09, 0.   , 0.12, 0.   , 0.79],
       [0.   , 0.   , 0.94, 0.   , 0.02, 0.   , 0.04, 0.   , 0.   ],
       [0.   , 1.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ]],
      dtype=float32)
```

전체적인 확률을 알고 싶을 때

```
-----  
>>> y_pred = model.predict_classes(X_new)
>>> y_pred
array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
>>> y_new = y_test[:3]
>>> y_new
array([9, 2, 1])
```

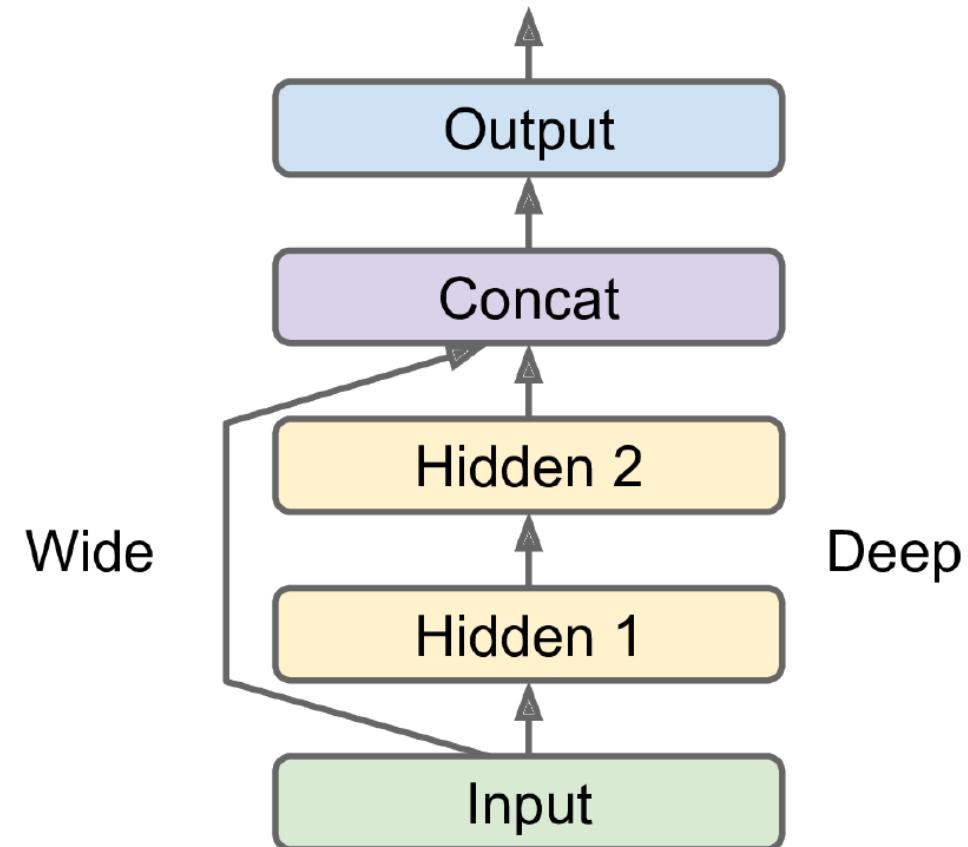
가장 높은 확률을 가진
클래스만 알고 싶을 때
→ `predict_classes()`

| 10.2 케라스로 다층 퍼셉트론 구현하기

시퀀셜 API를 사용하여 회귀용 다층 퍼셉트론 만들기

```
input = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.models.Model(inputs=[input], outputs=[output])
```

TIP. 코드와 다이어그램 비교해보기

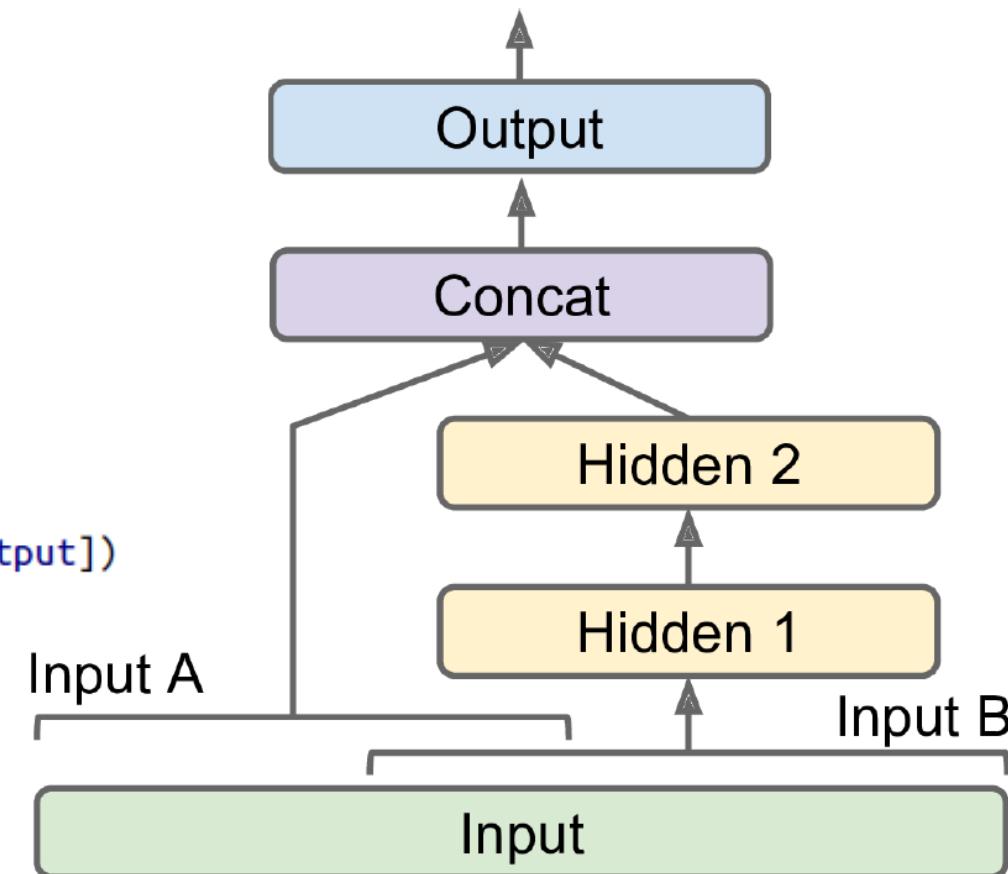


10.2 케라스로 다층 퍼셉트론 구현하기

시퀀셜 API를 사용하여 회귀용 다층 퍼셉트론 만들기

```
input_A = keras.layers.Input(shape=[5])
input_B = keras.layers.Input(shape=[6])
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.models.Model(inputs=[input_A, input_B], outputs=[output])
```

TIP. 코드와 다이어그램 비교해보기



10.2 케라스로 다층 퍼셉트론 구현하기

TIP. 코드와 다이어그램 비교해보기

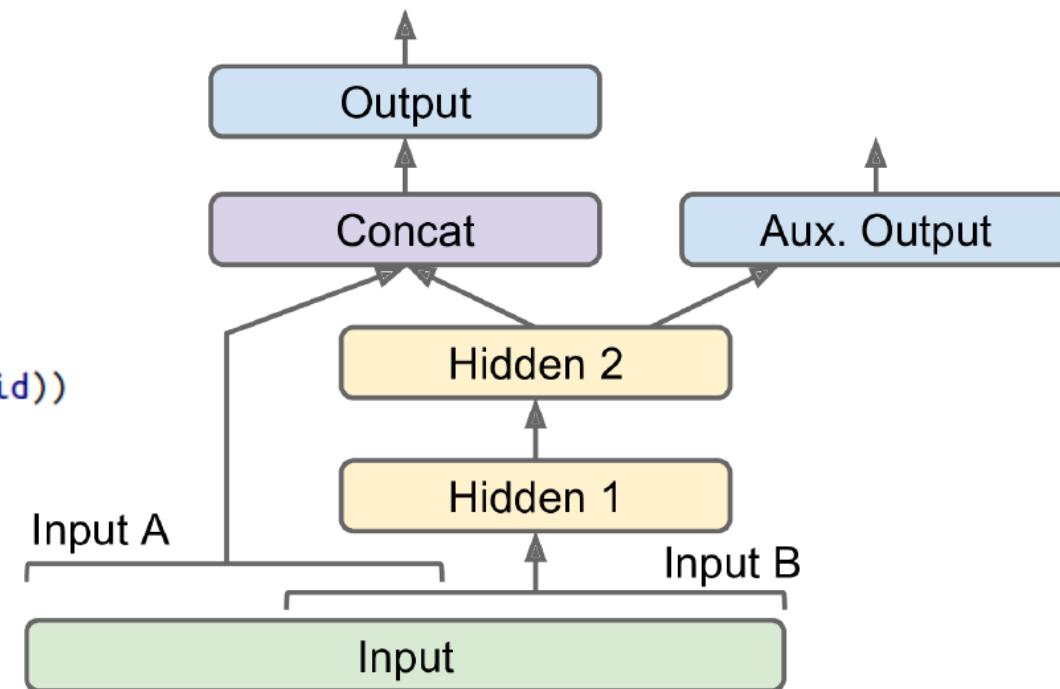
시퀀셜 API를 사용하여 회귀용 다층 퍼셉트론 만들기

```
model.compile(loss="mse", optimizer="sgd")

X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:]
X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:]
X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:]
X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]

history = model.fit((X_train_A, X_train_B), y_train, epochs=20,
                     validation_data=((X_valid_A, X_valid_B), y_valid))
mse_test = model.evaluate((X_test_A, X_test_B), y_test)
y_pred = model.predict((X_new_A, X_new_B))

output = keras.layers.Dense(1)(concat)
aux_output = keras.layers.Dense(1)(hidden2)
model = keras.models.Model(inputs=[input_A, input_B],
                           outputs=[output, aux_output])
```



| 10.2 케라스로 다층 퍼셉트론 구현하기

모델 저장과 복원

케라스는 HDF5 포맷을 사용하여 모든 층의 하이퍼파라미터를 포함한 모델 구조와 층의 모든 모델 파라미터(연결 가중치/편향)를 저장한다.

```
model.save("my_keras_model.h5")
```

모델 저장하기

```
model = keras.models.load_model("my_keras_model.h5")
```

모델 불러오기

| 10.3 신경망 하이퍼파라미터 튜닝하기

신경망의 유연성 → 조정할 하이퍼파라미터가 많아진다. (→ 단점이 되기도 함)

- 바꿀 수 있는 것?
 - 다중 퍼셉트론에서의 층의 개수
 - 한 층 내의 뉴런의 개수
 - 각 층에서 사용하는 활성화 함수(activation function)
 - 가중치 초기화 전략 ...
- Q. 어떤 하이퍼파라미터 조합이 주어진 문제에서 **최적의 해**가 될까?
(일종의 가이드라인을 학습하는 과정)

| 10.3 신경망 하이퍼파라미터 튜닝하기

은닉층 개수

복잡한 문제에서는 심층 신경망이 얇은 신경망에 비해 파라미터 효율성 (parameter efficiency)가 훨씬 좋은 편이다.

→ WHY? 심층 신경망은 복잡한 함수를 모델링하는 데 얇은 신경망보다 훨씬 적은 수의 뉴런을 사용하기 때문이다.

- 계층구조(hierarchical structure)

대부분의 저수준 구조를 학습할 필요가 없게 된다. → focus on 고수준 구조

(→ 전이학습: 비슷한 작업에서 가장 뛰어난 성능을 낸 미리 훈련된 네트워크의 일부를 재사용
→ 훈련속도 UP, 필요한 데이터 DOWN)

| 10.3 신경망 하이퍼파라미터 튜닝하기

은닉층의 뉴런 개수

- 입력층/출력층의 뉴런 개수는 입력과 출력의 형태에 의해 결정됨
- 일반적으로 각 층의 뉴런을 점점 줄여 깔때기 구성을 띠는 것이 일반적이었음
(저수준의 많은 특성이 고수준의 적은 특성으로 합쳐질 수 있기 때문)
- 요즘에는 모든 은닉층에 같은 크기를 사용해도 동일하거나 더 나은 성능을 보임
- (실전) 필요한 것보다 많은 층과 뉴런을 가진 모델을 설계
→ 과대적합을 막기 위해 조기종료/규제 기법 등을 사용 (stretch pants 기법)

| 10.3 신경망 하이퍼파라미터 튜닝하기

학습률, 배치 크기 그리고 기타 하이퍼파라미터

- 학습률: 최적의 학습률은 대략 최대 학습률의 절반 정도에 해당한다.
 매우 낮은 학습률에서 점진적으로 큰 학습률까지 수백 번 반복하는 것
- 옵티마이저: 고전적 경사 하강법보다 더 좋은 옵티마이저를 선택
 (다양한 고급 옵티마이저가 존재한다.)
- 배치 크기: 큰 배치 크기를 사용하면 GPU 등의 하드웨어 가속기를 효율적으로
 사용할 수 있다. → 초당 처리 샘플의 증가 → 훈련 초기는 불안정
 작은 배치 크기일수록 속도는 낮아짐
 → 학습률 예열을 사용해 큰 배치 크기를 점진적으로 시도
 (불안정하거나 결과가 만족스럽지 않으면 조금씩 낮춤)

| 10.3 신경망 하이퍼파라미터 튜닝하기

학습률, 배치 크기 그리고 기타 하이퍼파라미터

- 활성화 함수

일반적으로 ReLU 활성화 함수가 모든 은닉층에 좋은 기본값에 해당한다.

출력층의 활성화 함수는 수행하는 작업에 따라 달라진다.

- 옵티마이저

반복 횟수는 튜닝할 필요가 없고, 대신 조기 종료를 사용한다.

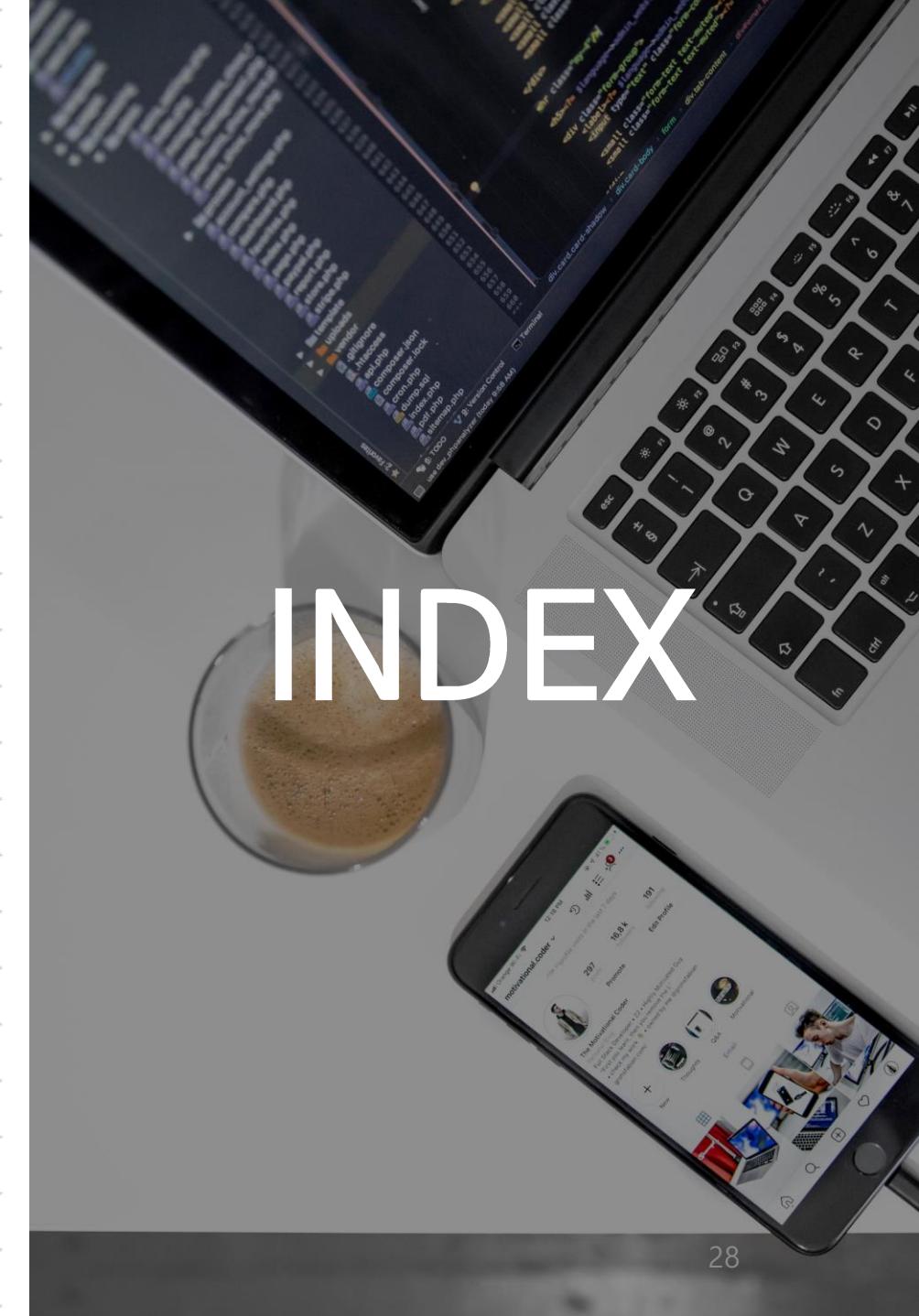
- 최적의 학습률은 다른 하이퍼파라미터에 의존적임.
- 배치 크기에 특히 영향을 많이 받으므로 다른 하이퍼파라미터를 수정하면 학습률도 반드시 튜닝해야 한다.

| 11.1 Gradient 소실과 폭주 문제

| 11.2 사전훈련된 층 재사용하기

| 11.3 고속 옵티마이저

| 11.4 규제를 사용해 과대적합 피하기



INDEX

| 11.1 Gradient 소실과 폭주 문제

역전파 알고리즘

출력층 → 입력층으로 오차 gradient를 전파하면서 진행된다.

신경망의 모든 파라미터에 대한 오차 함수의 gradient를 계산하면
경사 하강법 단계에서 이 gradient를 사용하여 각 파라미터를 수정한다.

| 11.1 Gradient 소실과 폭주 문제

Gradient 소실(vanishing gradient)

알고리즘이 하위층으로 진행될수록 gradient가 점점 작아지는 현상

→ 경사 하강법이 하위층의 연결 가중치를 변경되지 않은 채로 둔다면
좋은 솔루션으로 가지 X

Gradient 폭주(exploding gradient)

gradient가 점점 커져서 여러 층이 비정상적으로 큰 가중치로 갱신될 경우

알고리즘은 발산(diverge)한다.

11.1 Gradient 소실과 폭주 문제

불안정한 gradient 문제를 완화하는 방법은?

예측 시에는 정방향, gradient를 역전파할 때는 역방향으로 양방향 신호가 적절히 흘러야 하는데, 이를 만족하려면

각 층의 출력에 대한 분산 = 입력에 대한 분산을 만족해야 한다.

- 팬-인(fan-in): 층의 입력 연결 개수
 - 팬-아웃(fan-out): 층의 출력 연결 개수

이 두 개의 값이 같지 않으면
위를 보장할 수 X

→ 해결책: 각 층의 연결 가중치를 무작위로 초기화(세이비어 초기화)

$$fan_{avg} = \frac{fan_{in} + fan_{out}}{2}$$

| 11.1 Gradient 소실과 폭주 문제

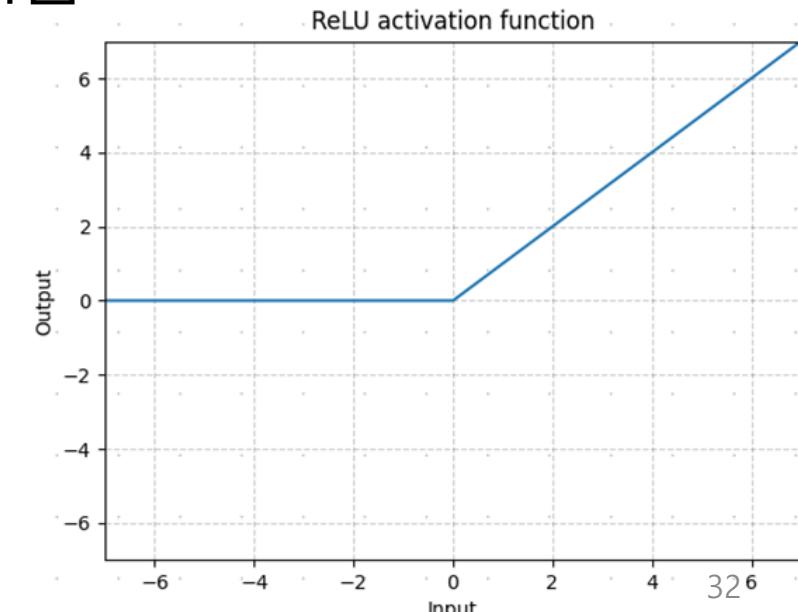
수렴하지 않는 활성화 함수

활성화 함수를 잘못 선택하면 gradient의 소실이나 폭주로 이어질 수 있다.

→ 시그모이드 함수가 최선일 것이라 생각했으나, 다른 활성화 함수가 더 잘 동작하는 것이 관찰되었다.

→ **ReLU 함수**: 특정 양수값에 수렴하지 않고 계산이 빠름

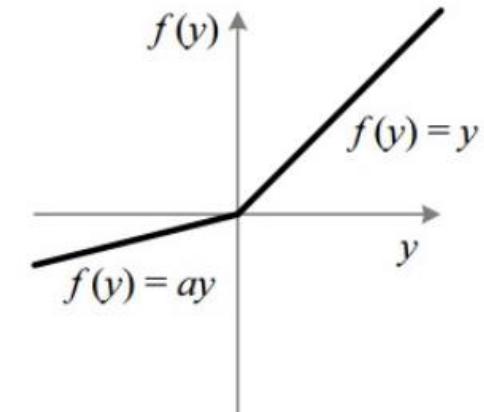
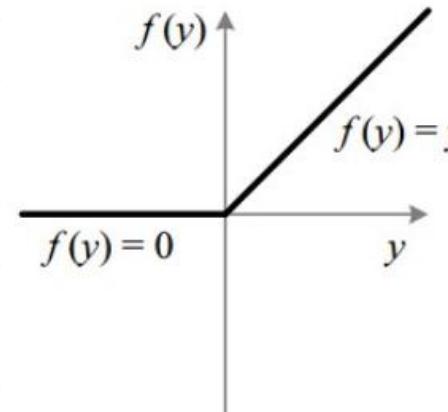
- 죽은 ReLU(dying ReLU)
훈련하는 동안 일부 뉴런이 0 이외의 값을 출력하지 않게 되는 현상



11.1 Gradient 소실과 폭주 문제

수렴하지 않는 활성화 함수

LeakyReLU 함수: ReLU 함수의 변종



$$\text{LeakyReLU}_a(z) = \max(az, z)$$

- a : 이 함수가 ‘새는(leaky)’ 정도를 나타낸다.

$z < 0$ 일 때 이 함수의 기울기, 일반적으로는 0.01

→ 이 작은 기울기가 dying ReLU 문제점을 해결해준다.

→ dying 상태에 갈 수는 있으나, 다시 깨어날 가능성을 부여

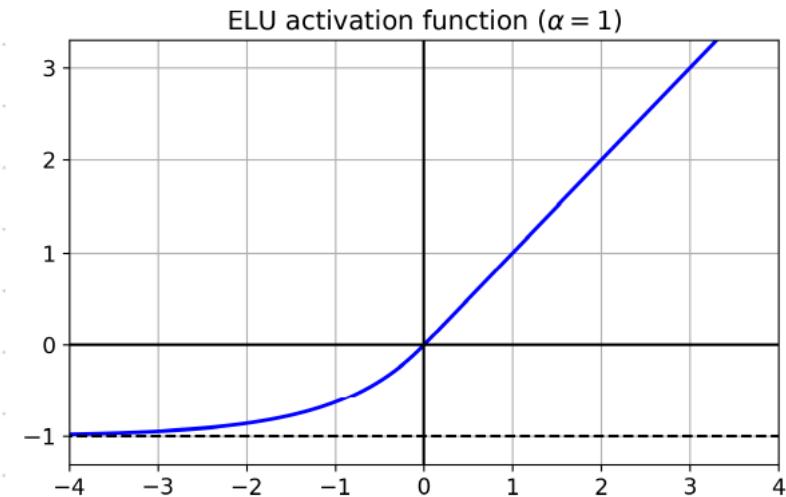
→ $a=0.2$ (많이 통과)일 때가 $a=0.01$ (조금 통과)일 때보다 더 나은 성능을 보임

```
leaky_relu = keras.layers.LeakyReLU(alpha=0.2)
layer = keras.layers.Dense(10, activation=leaky_relu,
                         kernel_initializer="he_normal")
```

| 11.1 Gradient 소실과 폭주 문제

수렴하지 않는 활성화 함수

ELU 함수: exponential linear unit



$$\text{ELU}_a(z) = \begin{cases} a(e^z - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

- $z < 0$ 일 때 음수값이 들어오므로 활성화 함수의 평균출력이 0에 더 가까워진다.
- $z < 0$ 이어도 gradient가 0이 아니므로 dying ReLU를 발생시키지 않는다.
- $a = 10$ 이면 이 함수는 $z = 0$ 에서 급격하게 변동하지 않게 되므로 전 구간에서 매끄러워 경사 하강법의 속도를 높여준다.

SELU 함수: scaled ELU; 스케일이 조정된 ELU 활성화 함수의 변종

| 11.2 사전훈련된 층 재사용하기

전이 학습(Transfer Learning)

해결하려는 것과 비슷한 유형의 문제를 처리한 신경망을 찾아본 후
그 신경망의 하위층을 재사용하는 기법

- 모델 A를 load하고, 이 모델의 층을 기반으로 model_B_on_A를 만든다.(출력층 제외 모두 재사용)

```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

- 위의 둘은 일부를 공유하기에, 서로 영향을 받는다. 이를 원치 않으면 clone할 수 있다.

→ 모델의 구조를 복제한 후 가중치를 복사 (cf. `clone_model()`은 가중치 복제 X)

```
model_A_clone = keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

11.2 사전훈련된 층 재사용하기

- 작업 B를 위해 model_B_on_A를 훈련
새로운 출력층이 랜덤하게 초기화되어 이는 큰 오차를 초래할 가능성이 존재
- 초기 몇 번의 epoch 동안은 재사용된 층을 동결하고 새로운 층에게 적절한 가중치 학습의 시간을 부여하는 것 → 모든 층의 trainable 속성을 False로 지정

```
for layer in model_B_on_A.layers[:-1]:  
    layer.trainable = False
```

```
model_B_on_A.compile(loss="binary_crossentropy", optimizer="sgd",  
                      metrics=["accuracy"])
```

- 얼마 지난 후, 동결을 해제한다.

```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,  
                           validation_data=(X_valid_B, y_valid_B))
```

```
for layer in model_B_on_A.layers[:-1]:  
    layer.trainable = True
```

11.2 사전훈련된 층 재사용하기

- 작업 B에 맞게 재사용된 층을 세밀하게 튜닝하기 위해 훈련을 계속한다.
이때 재사용된 층의 동결을 해제한 후에는 학습률을 낮추는 것이 권장됨
(재사용된 가중치가 망가지는 것을 완화하기 위함)

```
optimizer = keras.optimizers.SGD(lr=1e-4) # the default lr is 1e-3
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                      metrics=[ "accuracy"])
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                            validation_data=(X_valid_B, y_valid_B))
```

- 결과: 전이 학습이 오차율을 2.8%에서 0.7%까지 낮춤

```
>>> model_B_on_A.evaluate(X_test_B, y_test_B)
[0.06887910133600235, 0.9925]
```

| 11.3 고속 옵티마이저

훈련 속도를 높이는 방법

- 연결 가중치에 좋은 초기화 전략 적용하기
- 좋은 활성화 함수 사용하기
- 배치 정규화 사용하기
- (보조작업 또는 비지도 학습을 사용하여 만들 수 있는)

사전훈련된 네트워크의 일부 재사용하기

- 표준적인 경사 하강법 옵티마이저 대신 더 빠른 옵티마이저 사용하기

e.g. 모멘텀 최적화, 네스테로프 가속 경사, AdaGrad, RMSProp, Adam, ...

11.3 고속 옵티마이저

모멘텀 최적화(momentum optimization)

- 이전 gradient가 얼마였는지에 무게를 둠
- 매 반복에서 현재 gradient를 학습률에 곱한 후 모멘텀 벡터 m 에 더하고 이 값을 빼는 방식으로 가중치를 갱신
- gradient를 속도가 아니라 가속도로 사용한다.
- 모멘텀이라는 하이퍼파라미터 β 가 추가됨 $\rightarrow 0$ (높은 마찰저항) ~ 1 (마찰저항X)

$$1. \quad m \leftarrow \beta m - \eta \nabla_{\theta} J(\theta)$$

$$2. \quad \theta \leftarrow \theta + m$$

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

11.3 고속 옵티마이저

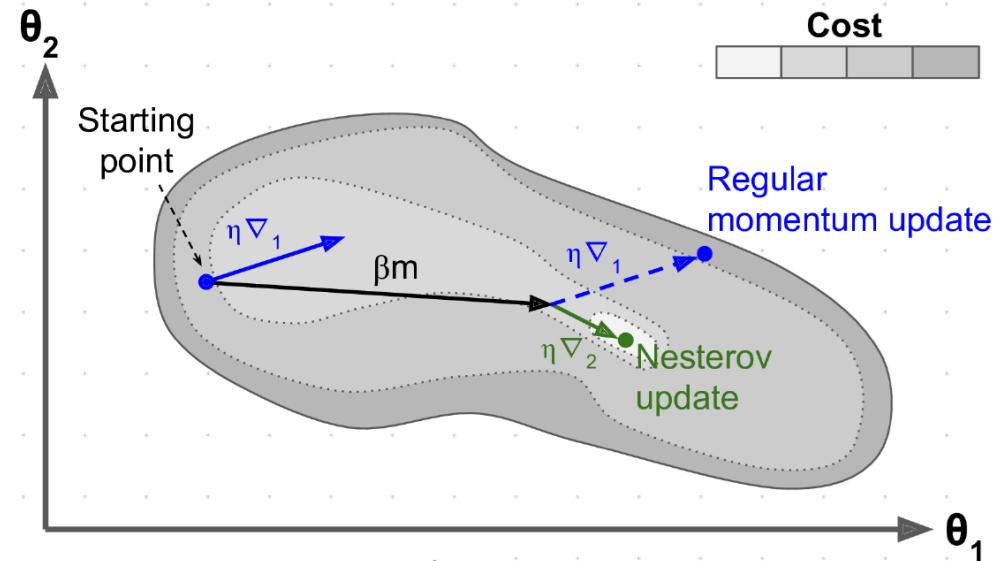
네스테로프 가속 경사(NAG; Nesterov accelerated gradient)

- 현재 위치 θ 가 아니라 모멘텀의 방향으로 조금 앞선 $\theta + \beta m$ 에서 비용함수의 gradient를 계산
- 일반적으로 기본 모멘텀 최적화보다 훈련속도가 빠르다.
- `use_nesterov = True`

$$1. \quad \mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\theta + \beta \mathbf{m})$$

$$2. \quad \theta \leftarrow \theta + \mathbf{m}$$

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```



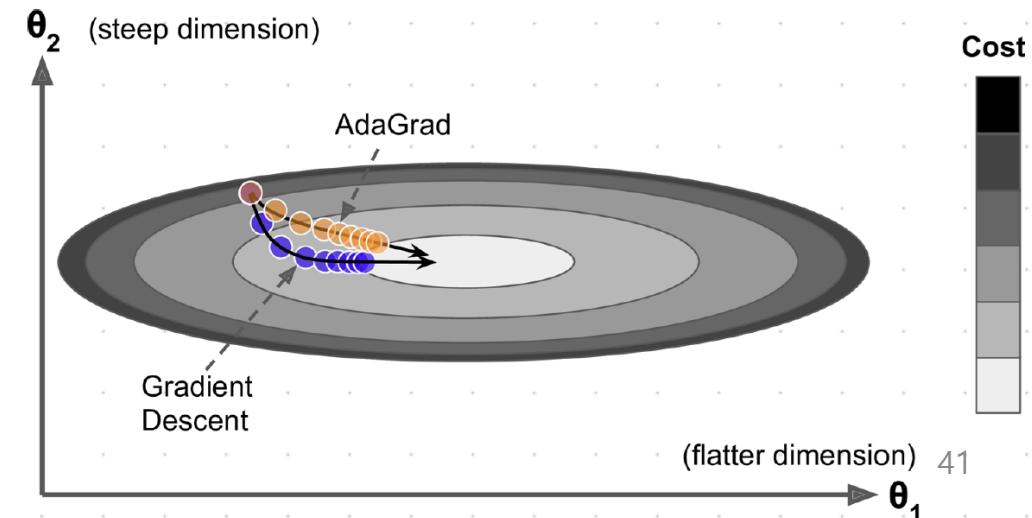
11.3 고속 옵티마이저

AdaGrad

- “경사 하강법에서 전역 최적점으로 곧 향하지 않고 가장 가파른 경사를 미리 감지하고 전역 최적점 쪽으로 좀 더 정확한 방향을 잡을 수 있지 않았을까?”
- 가장 가파른 차원을 따라 gradient vector의 스케일을 감소시킨다.
- 학습률의 감소가 커져 전역 최적점 도달 전에 알고리즘이 완전히 멈춤
→ 선형 회귀 등을 제외하고는, **심층 신경망에서는 사용하지 않아야 함**

$$1. \quad s \leftarrow s + \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$$

$$2. \quad \theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \epsilon}$$



| 11.3 고속 옵티마이저

RMSProp

- AdaGrad는 너무 빨리 느려져 전역 최적점에 수렴X

→ 가장 최근 반복에서 비롯된 gradient만 누적함으로써 이 문제를 해결

1. $s \leftarrow \beta s + (1 - \beta) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$ 감쇠율(β)은 보통 0.9로 설정함
2. $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \epsilon}$

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

11.3 고속 옵티마이저

Adam(adaptive moment estimation; 적응적 모멘트 추정)

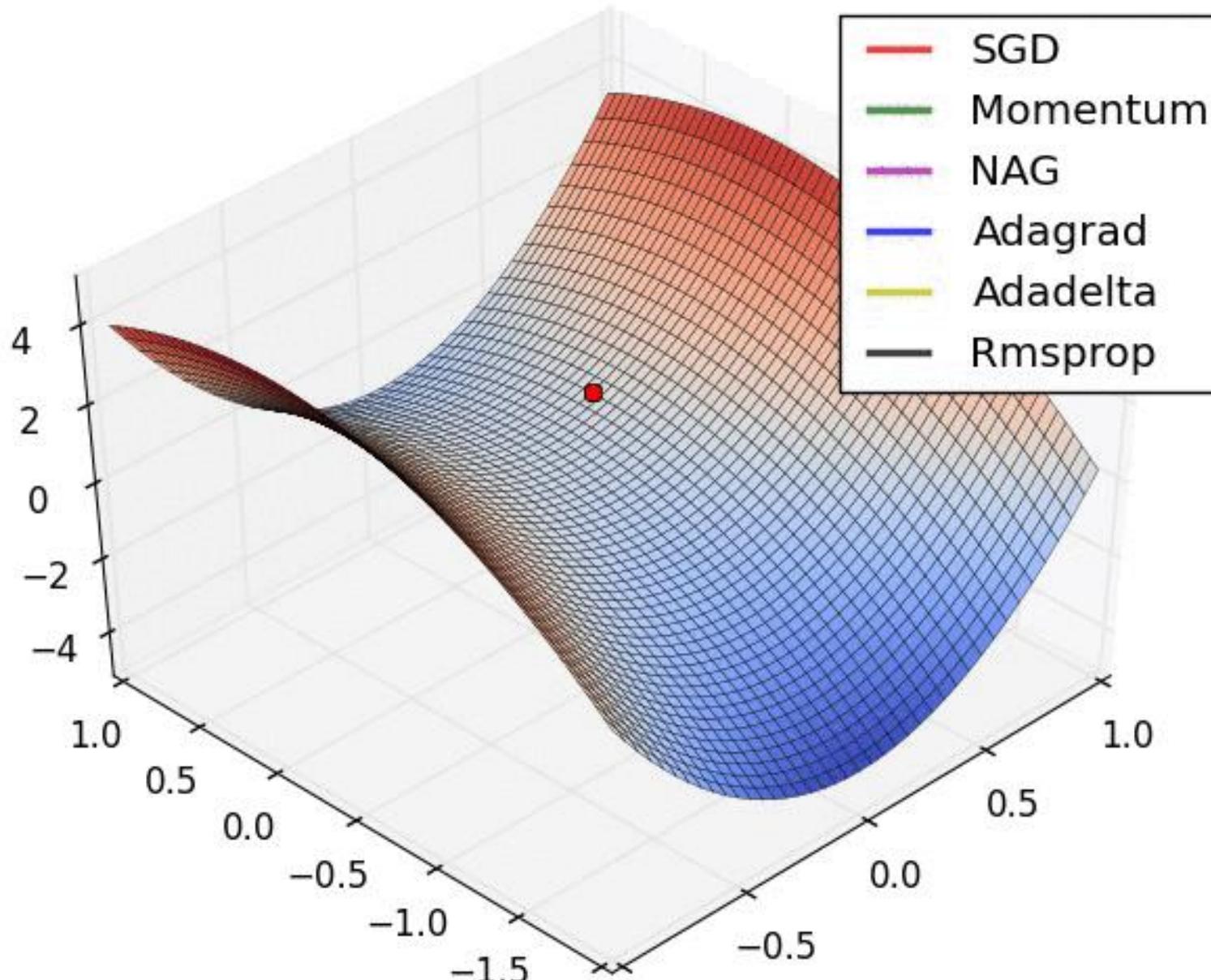
- 모멘텀 최적화 – 지난 gradient의 지수 감소 평균을 따름

RMSProp – 지난 gradient 제곱의 지수 감소된 평균을 따름

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\theta} J(\theta)$ 일반적으로 $\beta_1 = 0.9, \beta_2 = 0.999$ 로 초기화
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$ 또한, $\epsilon = 10^{-7}$
3. $\widehat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^t}$ keras.backend.epsilon()
4. $\widehat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^t}$
5. $\theta \leftarrow \theta + \eta \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}} + \epsilon}$

11.3 고속 옵티마이저



| 11.4 규제를 사용해 과대적합 피하기

| I1과 I2 규제

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
```

I2() 함수는 훈련하는 동안 규제 손실을 계산하기 위해 각 스텝에서 호출되는 규제 객체를 반환한다.
→ 이 손실은 최종 손실에 합산됨

- I1 규제: keras.regularizers.l1()
- I1 + I2 규제: keras.regularizers.l1_l2()
- 또 다른 방법: functions.partial() 함수를 사용하여 기본 매개변수 값을 사용하여
함수 호출을 감싸는 것

| 11.4 규제를 사용해 과대적합 피하기

| I1과 I2 규제

```
from functools import partial

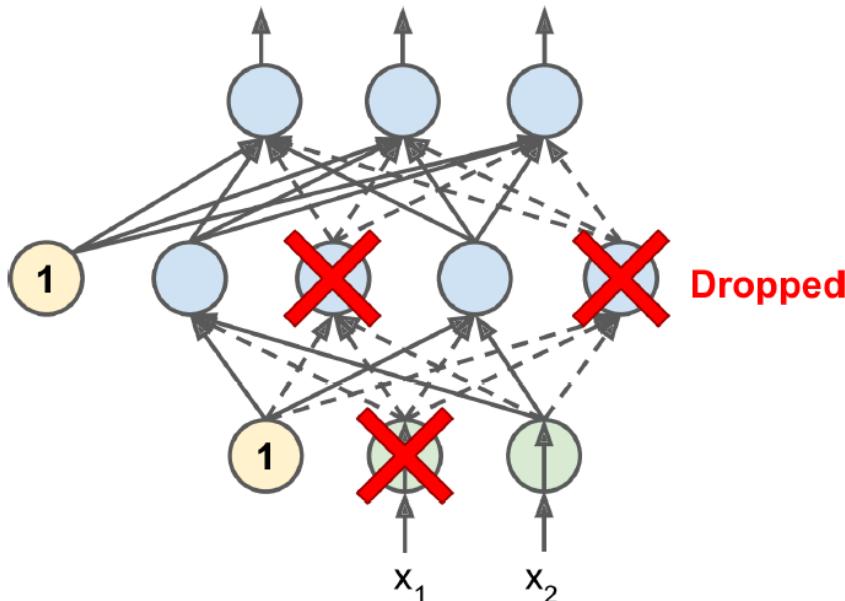
RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax",
                     kernel_initializer="glorot_uniform")
])
```

| 11.4 규제를 사용해 과대적합 피하기

드롭아웃(dropout)

- 매 훈련스텝에서 각 뉴런(출력뉴런은 제외)은 임시적으로 드롭아웃될 확률 p
- 이번 훈련스텝에는 완전히 무시되나, 다음 스텝에서 활성화될 가능성 존재
- 드롭아웃 비율(dropout rate): 하이퍼파라미터 p
(보통 10% ~ 50% 사이 → 순환 신경망 2~30%, 합성곱 신경망 4~50%)



드롭아웃으로 훈련된 뉴런은 이웃한 뉴런에 맞추어 적응될 수 없다. → 자기 자신이 유용해져야 됨

모든 입력 뉴런에 주의를 기울여야 됨
→ 입력값의 작은 변화에 덜 민감해짐
→ 더 안정적인 네트워크가 되어 일반화 성능 증가

| 11.4 규제를 사용해 과대적합 피하기

드롭아웃(dropout)

- in Keras, keras.layers.Dropout
- e.g. 드롭아웃 비율 0.2를 사용한 드롭아웃 규제를 모든 Dense 층 이전에 적용한 코드

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

모델이 과대적합되었다면? → 드롭아웃 비율을 늘릴 수 있다. (+층이 클 때)

모델이 과소적합되었다면? → 드롭아웃 비율을 낮추어야 한다. (+층이 작을 때)



Thank You