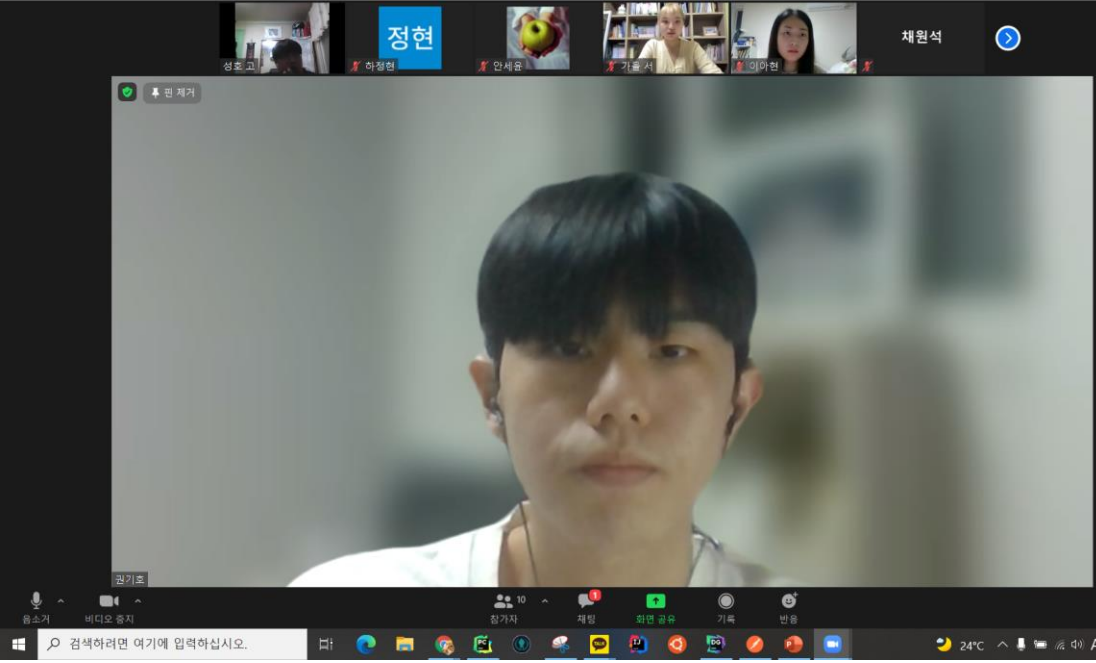
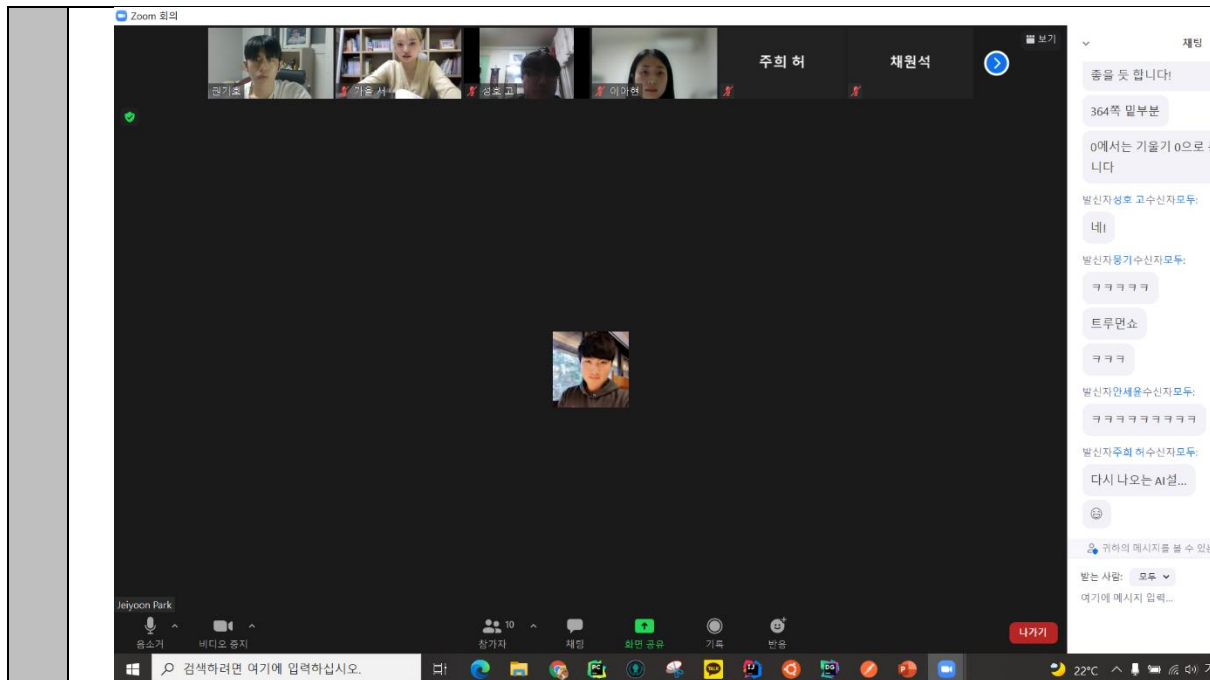


# TAVE 서기

서기 내용			
서 기 일 자	21.09.10	서기	고성호
주 제	케라스를 사용한 인공 신경망 소개		
시 간	20:30~22:30	장소	온라인
스 터 디 인 원	고성호, 권기호, 이아현, 서가을 : 시작		
			
	고성호, 권기호, 이아현, 서가을 : 종료		



## 내용

### [목차]

## Chapter 10. 케라스를 사용한 인공 신경망 소개

### 10.1 생물학적 뉴런에서 인공 뉴런까지

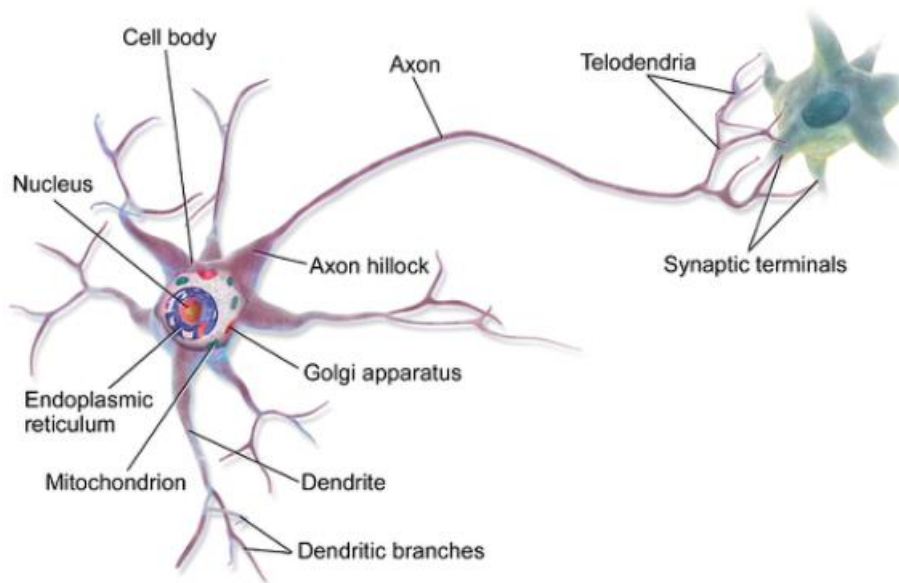
### 10.2 케라스로 다층 퍼셉트론 구현하기

### 10.3 신경망 하이퍼파라미터 튜닝하기

### 10.4 연습문제

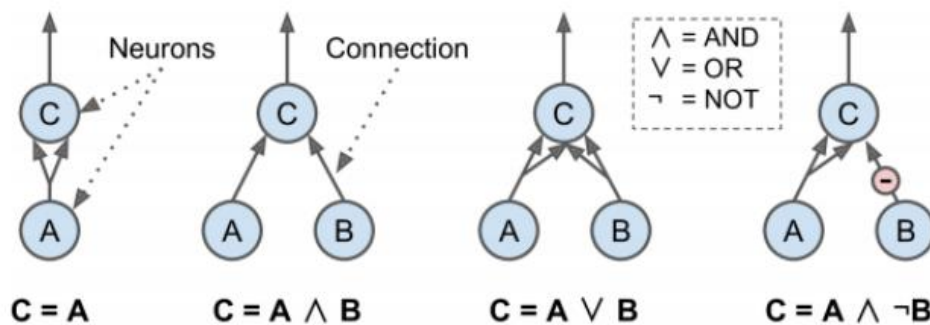
## 배운 내용

### 10.1.1 생물학적 뉴런



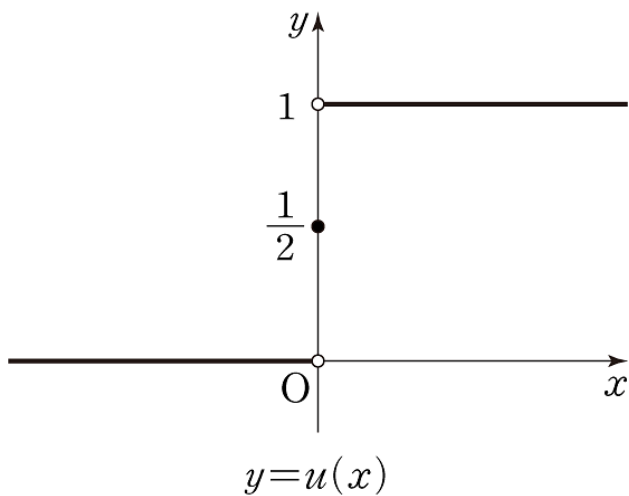
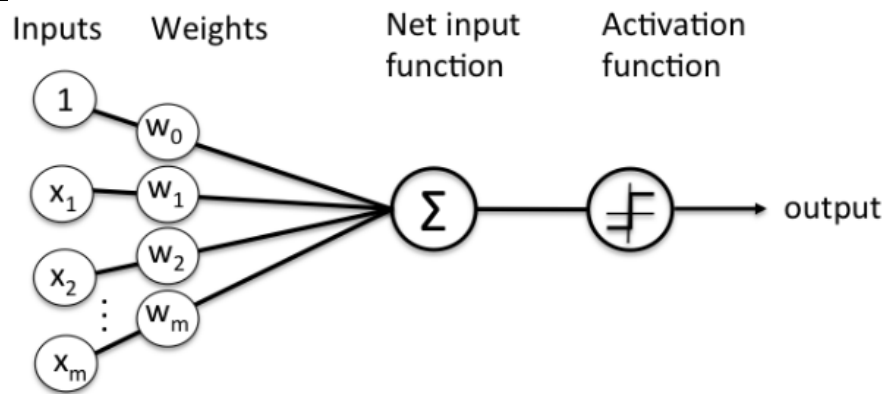
- 시냅스 말단이라는 구조를 통해 정보를 전달받음
- 이 정보를 신경전달물질이라는 화학적 신호로 바꾸어 다른 뉴런의 수상돌기나 세포체에 연결됨.
- 인공 신경망의 인공 뉴런 또한 비슷한 원리로 작동함.
- 여러 개의 input을 전달받으면 activation function을 거쳐 특정 output으로 도출해냄.
- 여러 개의 데이터를 받아 특정 결과를 도출해낸다는 점에서 생물학적 뉴런 구조와 인공 뉴런과 상통함.

## 10.2 뉴런을 사용한 논리연산



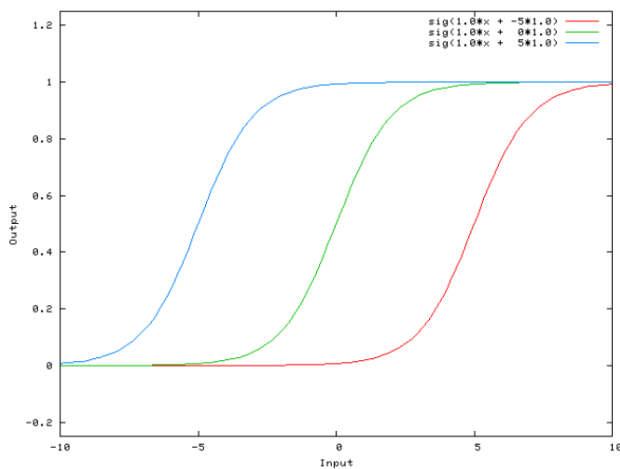
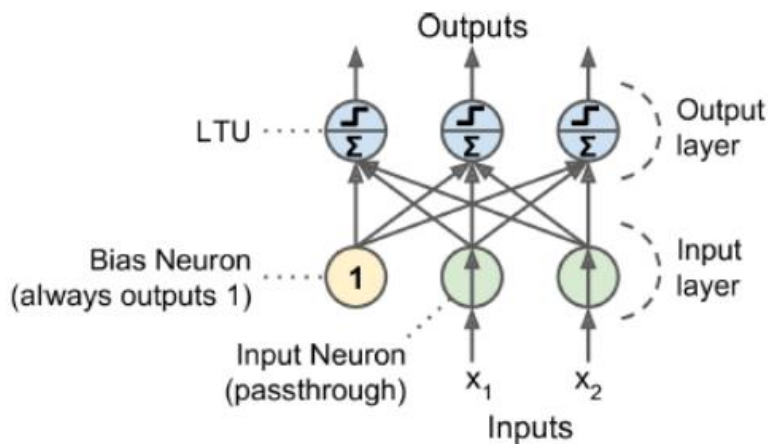
- 생물학적 뉴런에서 착안한 매우 단순한 신경망 모델
- A, B는 input, C는 네트워크이다.
- 첫 번째 네트워크는 항등함수이다.
- 두 번째 네트워크는 논리곱 연산을 수행한다. A and B 모두 켜져있을 때 C 함수가 활성화 된다.
- 세 번째 네트워크는 논리합 연산을 수행한다. A or B가 켜져있을 때 작동한다.
- 네 번째 네트워크는 B가 켜져있지 않고 A가 켜질 때 활성화 된다.

### 10.1.3 퍼셉트론



- 퍼셉트론은 TLU(Threshold Logic Unit), LTI(Linear threshold unit)
- Input : binary value가 아닌 연속적인 value
- Input function 에서는 input 벡터들을 받아 선형결합을 한 값을 출력한다.
- Activation Function에서는 heaviside step function 또는 signal function을 사용한다.
- TLU를 훈련한다는 것은 최적의  $w_0, w_1, w_2, \dots, w_n$ 을 찾는다는 의미.
- 이러한 단일 퍼셉트론은 이진 분류 문제에 활용 가능하다.

층이 하나인 TLU(Threshold Logic Unit)



- 완전 연결층(Fully connected layer), 밀집 층(Dense Layer) : 한 층에 있는 모든 뉴런이 이전 층에 있는 모든 뉴런과 연결되어 있음
- 입력 뉴런 : input을 받는 뉴런. 받은 input에 어떠한 처리 없이 그대로 output으로 return한다.
- 입력 층(input layer) : 입력 뉴런으로 구성된 층
- 편향 뉴런(Bias neuron) : bias를 더해주는 뉴런

완전 연결층의 출력 계산

$$h_{w,b}(x) = \Phi(XW + b)$$

- $X$ 는 input을 뜻한다.
- $W$ 는 bias를 제외한 모든 연결 가중치를 뜻한다.
- 모든 인공 뉴런마다 하나의 편향 값이 존재한다.
- $\Phi$ 는 activation function을 뜻한다. 인공 뉴런이 TLU일 경우 이

함수는 step function이 된다.

퍼셉트론의 훈련

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

- Donald Hebb : 생물학적 뉴런 A와 B가 있다고 했을 때 A가 B를 활성화 시킬 때마다, A와 B의 연결은 더욱 강해진다.
- Donald Hebb의 아이디어에서 영감을 받은 프랑크 로젠블라트는 위와 같은 뉴런 학습 아이디어를 제시함
- 왼쪽 항에 있는  $w_{i,j}^{(next\ step)}$  은 update할 가중치이다. 즉 update할 가중치를 구하는 수식이다.
- 오른쪽 항에 있는  $w_{i,j}$  는 현재 step의 가중치이다.  
·  $i$  번째 뉴런과  $j$  번째 뉴런을 연결하는 가중치이다.
- $\eta$  는 학습률(Learning Rate)을 뜻한다.
- $\hat{y}$  은 출력 뉴런의 출력값이다.
- $Y$  는 훈련 샘플의 True 값, 혹은 Target값이다.

TLU 네트워크를 구현한 Perceptron 클래스 사용

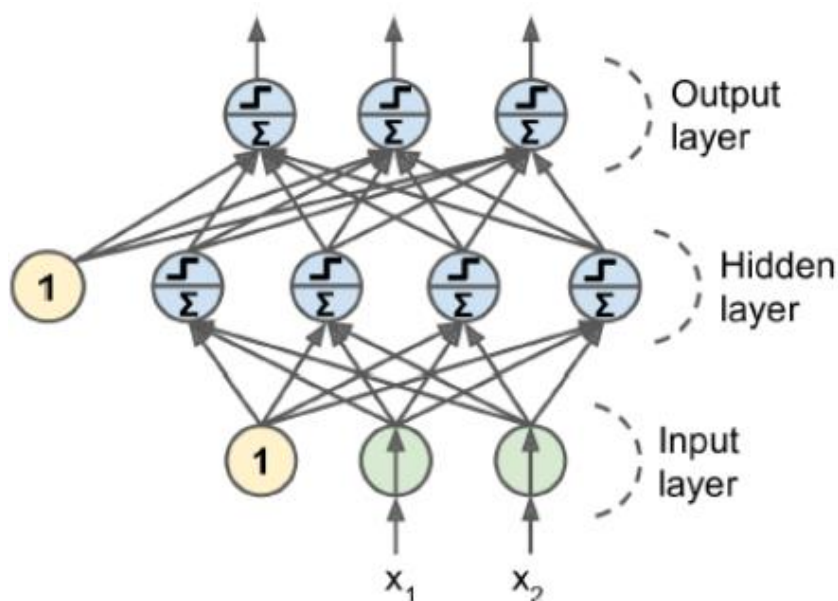
```
1 import numpy as np
2 from sklearn.datasets import load_iris
3 from sklearn.linear_model import Perceptron
4
5 iris=load_iris()
6 X=iris.data[:,(2,3)] #2,3번째 column은 iris의 length, width
7 y=(iris.target==0).astype(np.int) #target 0은 Iris Setosa
8
9 per_clf=Perceptron()
10 per_clf.fit(X,y)
11
12 y_pred=per_clf.predict([[2, 0.5]])
13 y_pred
```

array([0])

- Scikit learn library는 Perceptron이라는 클래스를 제공하고 있다.
- 위와 같이 Perceptron 을 활용해 객체를 생성하여 이진분류가 가능하다.
- Perceptron 클래스는 확률적 경사 하강법과 비슷한 원리로 가중치를 학습함.

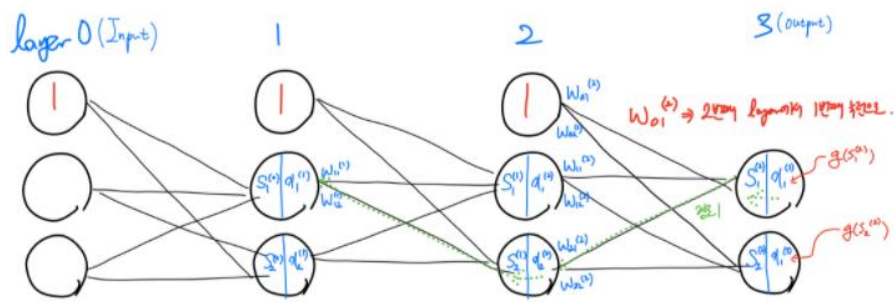
#### 10.1.4 다층 퍼셉트론과 역전파

다층 퍼셉트론



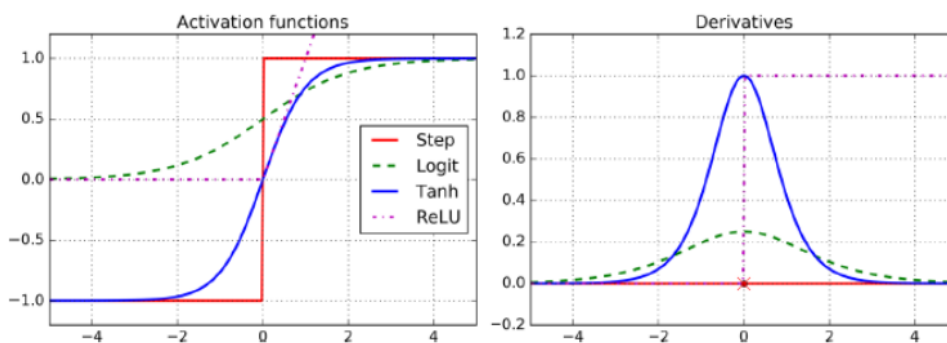
- 입력층, 1개 이상의 은닉층, 출력층으로 구분되어 있다.
- 출력층을 제외하고 모든 층은 bias neuron을 포함하고 다음 층과 완전히 연결되어 있다.
- 은닉층을 여러 개 쌓아 올린 신경망 모델을 DNN(Deep Neural Network)라고 한다.

다층 퍼셉트론 훈련 방법 – Backpropagation



- 역전파는 오차를 감소시키는 방향으로 가중치를 update하는 방법이다.
- Epoch : 가중치를 update하는 한 주기의 Loop를 뜻한다.
- Forward pass : 가중치를 통해 target value에 대한 예측값을 구하는 단계
- Backward pass : Forward Pass를 통해 구한 예측값으로 가중치를 update하는 단계
- Chain rule : 역전파 과정에서 가중치 update시 사용되는 미분 기법

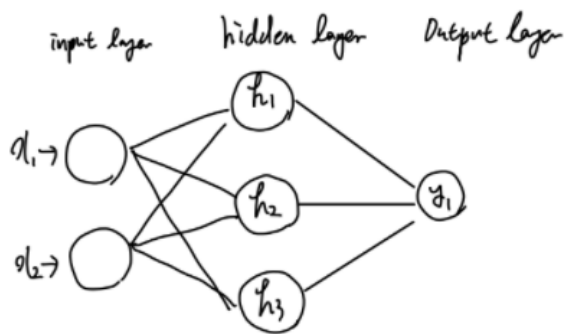
## 다층 퍼셉트론 Activation Function



- 다층 퍼셉트론의 가중치 update를 위해서는 backpropagation과정을 거쳐야 하는데, 이 때 미분을 통해 기울기 계산을 해야 한다.
- 단층 퍼셉트론에서 봤던 계단함수는 수평선 밖에 없으므로 사용할 기울기가 존재하지 않는다.
- 때문에 ReLU나 Hyperbolic Tangent 함수를 Activation으로 사용한다.
- 왼쪽은 원래 함수의 그래프이고 오른쪽은 도함수 그래프이다.

Activation Function으로 선형 함수를 사용하면 안 되는 이유





$$h_1 = w_{11}x_1 + w_{12}x_2$$

$$h_2 = w_{21}x_1 + w_{22}x_2$$

$$h_3 = w_{31}x_1 + w_{32}x_2$$

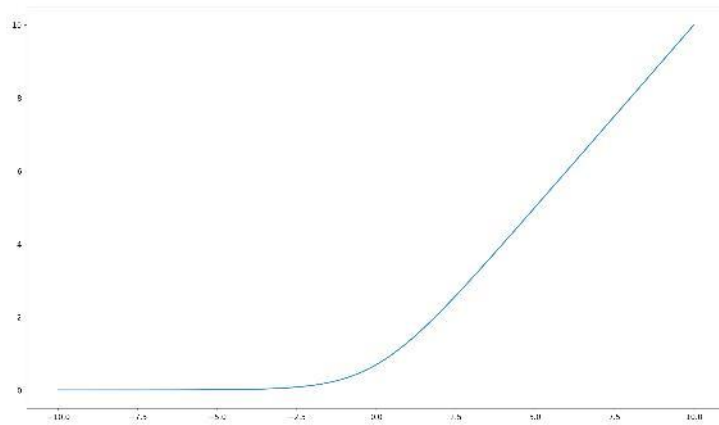
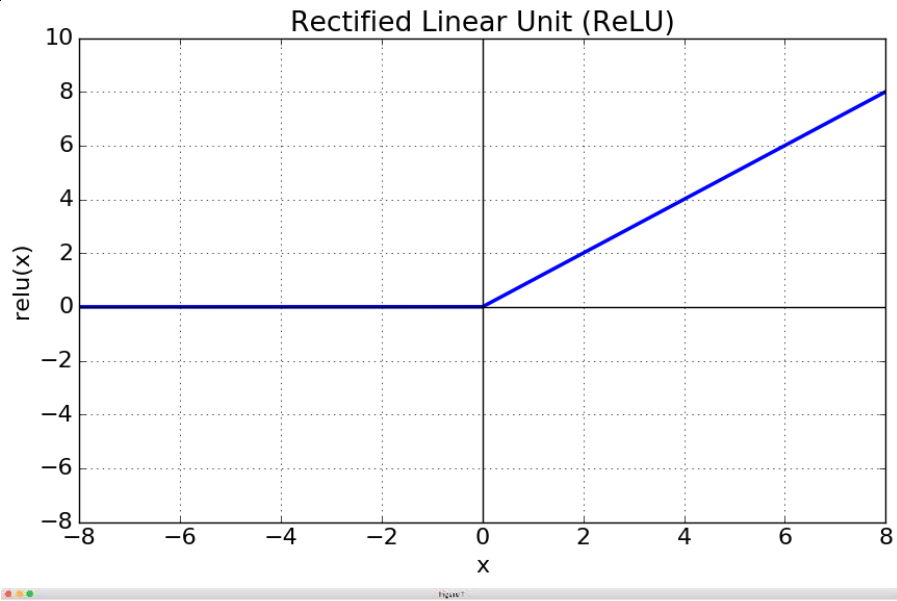
$$y = w_{21}h_1 + w_{22}h_2 + w_{23}h_3$$

$$y = w_{21}(w_{11}x_1 + w_{12}x_2) + w_{22}(w_{21}x_1 + w_{22}x_2) + w_{23}(w_{31}x_1 + w_{32}x_2)$$

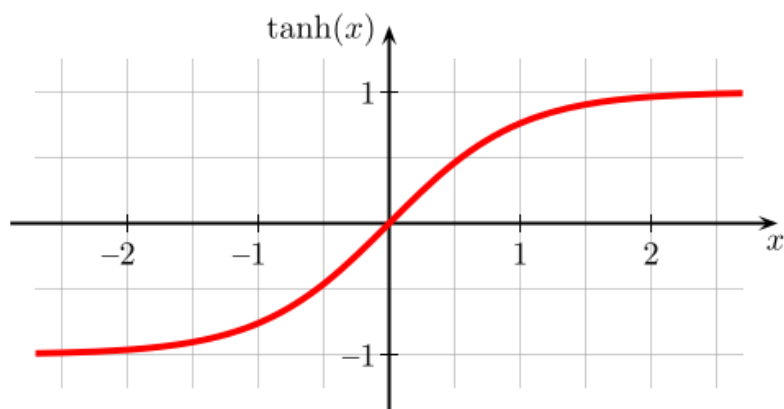
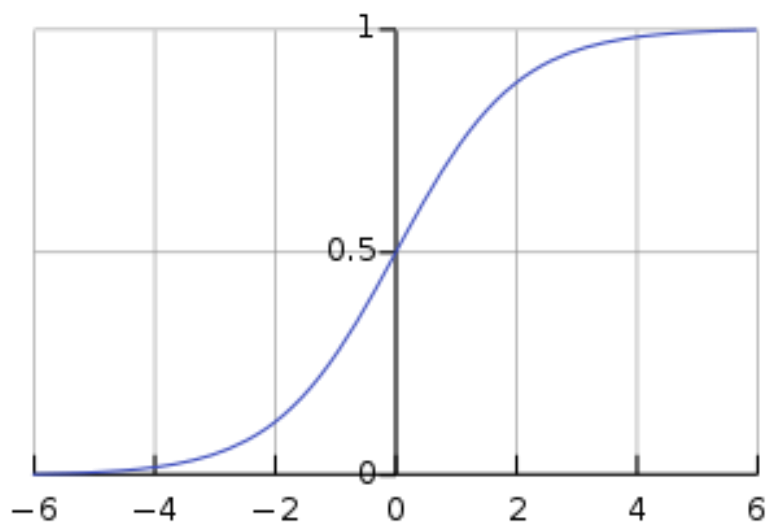
$$= \underline{w_{11}w_{21}x_1 + \dots}$$

Linear

- $h_1, h_2, h_3$ 는 각각 같은 층에 속해있는 neuron이고 선형 함수를 activation function으로 가지고 있다.
- $W$ 는 가중치이다.
- $x_1, x_2$ 는 input이다.
- $y$ 는 output layer의 neuro이다.
- Output 을 보면 선형 함수 식으로 나타나는 것을 볼 수 있다.
- XOR문제에서 보았듯이 선형적으로 구분되지 않는 문제를 풀기 위해 DNN이 도출되었다. 그런데 Activation Function을 선형 함수로 설정 하면, hidden layer를 추가해 비선형적인 구분을 가능케 한다는 DNN의 목적이 전혀 달성되지 않는다는 것을 알 수 있다.
- 그렇기 때문에 activation function으로는 선형 함수를 사용할 수 없다.

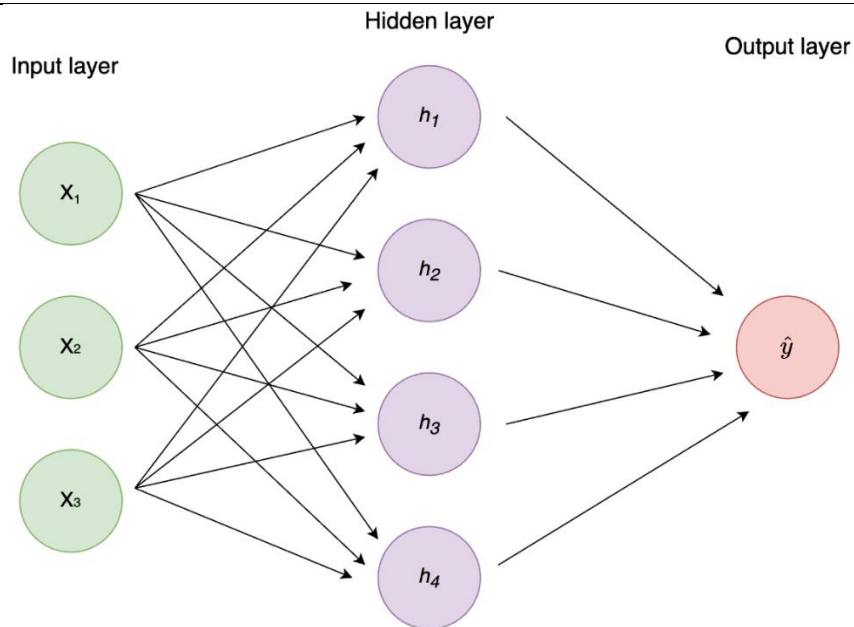


- 회귀용 다층 퍼셉트론을 만들 때 출력 뉴런에 activation function을 사용하지 않고, 어떤 범위의 값도 출력되도록 함. 다시 말해 이전 층의 값을 가중합 한 결과를 그대로 출력한다.
- 만약, 회귀분석 결과 값을 0 이상으로 제한해야 한다면, ReLU, softplus 함수를 사용한다.

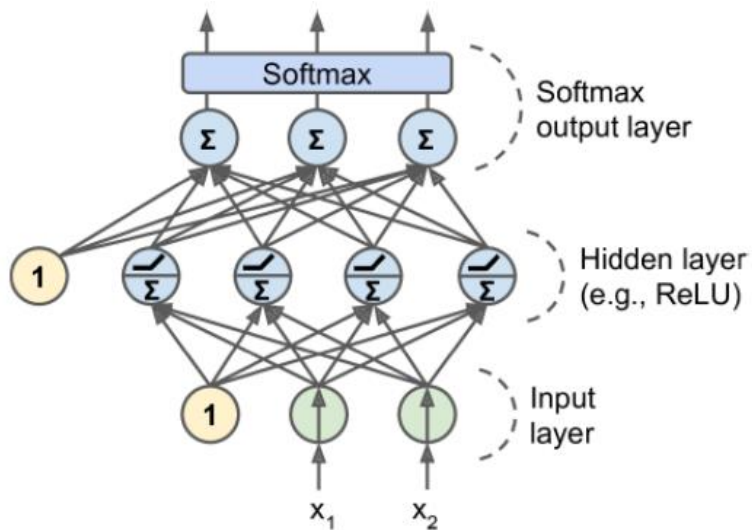


- 만약 회귀분석 결과를 0~1사이의 값으로 제한하고 싶으면 Sigmoid 함수를 사용한다.
- 만약 회귀분석 결과를 -1~1사이의 값으로 제한하고 싶으면 hyperbolic tangent 함수를 사용한다.

이진 분류 문제



- 0~1사이의 값을 출력해야 하는 이진 분류 문제의 경우 output layer에 1개의 neuron만 필요하다.



- 다중 레이블 이진 분류의 경우 softmax함수를 사용하여 분류한다.

## 10.2 케라스로 다층 퍼셉트론 구현하기

### 10.2.1 텐서플로 설치 및 버전확인

```
import tensorflow as tf
from tensorflow import keras
```

```
tf.__version__
```

```
'2.6.0'
```

```
keras.__version__
```

```
'2.6.0'
```

## 10.2.1 시퀀셜 API를 사용하여 이미지 분류기 만들기

### 케라스를 이용하여 데이터셋 적재하기

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

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
32768/29515 [=====] - 0s 0us/step
40960/29515 [=====] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26427392/26421880 [=====] - 0s 0us/step
26435584/26421880 [=====] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
16384/5148 [=====] - 0s 0us/step
=====] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [=====] - 0s 0us/step
4431872/4422102 [=====] - 0s 0us/step
```

```
# 훈련 세트 크기
X_train_full.shape
```

```
(60000, 28, 28)
```

```
# 훈련 세트 데이터 타입
X_train_full.dtype
```

```
dtype('uint8')
```

```
# 검증 세트 생성
X_valid, X_train = X_train_full[:5000] / 255.0, X_train_full[5000:] / 255.0 # 입력 특성의 스케일 조정 (0 ~ 1)
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

```
X_test = X_test / 255.0 # 입력 특성의 스케일 조정
# 훈련 데이터의 통계 속성을 사용하여 스케일을 조정한 경우, 새로운 데이터도 반드시 훈련 데이터에서 얻은 통계값을 이용하여 스케일 조정
```

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

```
class_names[y_train[0]]
```

```
'Coat'
```

- Keras에서 기본적으로 제공하는 fashion mnist dataset을 load하였다.
- y\_train\_full과 y\_train의 경우 0,1,2,3....과 같이 숫자로 labeling 되어있기 때문에 알아보기 쉽도록 label의 순서와 동일한 영문 label명에 해당하는 list를 생성하였다.

## 시퀀셜 API를 사용하여 모델 만들기

### 1) 객체에 add 메소드를 사용하여 layer 쌓기

```
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"))
```

- Sequential() 객체를 생성하여, add 메소드를 이용해 layer를 하나씩 쌓는다.
- Flatten layer는 fashion mnist의 이미지가 28 x 28로 2차원이기 때문에 1차원으로 DNN에 집어넣기 위하여 1차원으로 flatten해주는 작업을 한다.
- hidden layer의 activation function은 relu로, output layer에 있는 activation function은 softmax로 사용하였다.

### 2) 객체 생성시 layer의 정보가 포함된 list를 한 번에 넘기기

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

## Summary() 메소드

```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 300)	235500
dense_1 (Dense)	(None, 100)	30100
dense_2 (Dense)	(None, 10)	1010

Total params: 266,610

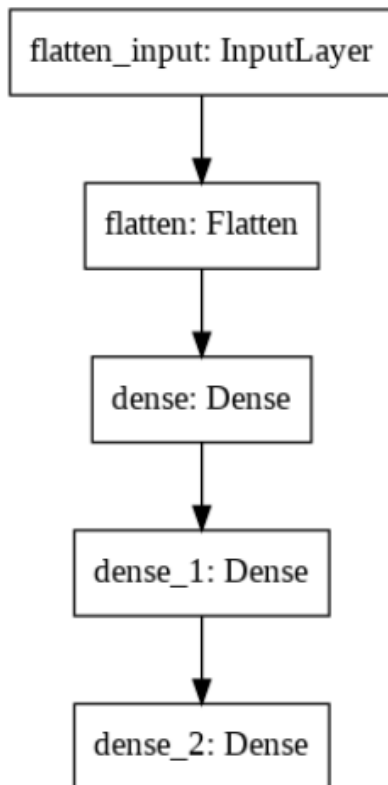
Trainable params: 266,610

Non-trainable params: 0

- summary() 메소드를 사용하면 우리가 생성한 model의 구조, 각 층별 이름, 층의 종류, shape, parameter 개수까지 한 번에 확인할 수 있다.

모델 구조를 이미지로 출력

```
keras.utils.plot_model(model) # 모델 구조를 이미지로 출력
```



- plot\_model을 사용하면 model summary()와는 다르게 도표로써 모델의 구조가 확인 가능하다.

```
[<keras.layers.core.Flatten at 0x7f87d48b6910>,  
<keras.layers.core.Dense at 0x7f87d4e89a90>,  
<keras.layers.core.Dense at 0x7f87d4d3ffd0>,  
<keras.layers.core.Dense at 0x7f87d4d3f3d0>]
```

```
hidden1 = model.layers[1]      # 인덱스로 층 선택
hidden1.name
'dense'
```

```
model.get_layer('dense') is hidden1 # 이름으로 층 선택
True
```

- `get_weights()`, `set_weights()`

```
weights, biases = hidden1.get_weights() # Dense 층의 경우 연결 가중치, 편향 모두 포함
```

weights

```
array([[ 0.00606353,  0.01166028, -0.03036456, ..., -0.06538334,
         0.01233295,  0.0401781 ],
       [-0.0636062,  0.06282045,  0.01129972, ..., -0.00776301,
         0.05862258, -0.03868764],
       [-0.02304003,  0.02108692,  0.02172109, ...,  0.07169919,
        -0.06120358, -0.04145927],
       ...,
       [-0.04925218,  0.04150926,  0.03803544, ..., -0.0438657 ,
        -0.01363878,  0.0219386 ],
       [-0.01981895, -0.03224923, -0.04766261, ...,  0.06424057,
        -0.04749575,  0.04019612],
       [-0.06746492, -0.03933114, -0.03702291, ..., -0.02811691,
        -0.04753896, -0.00322841]], dtype=float32)
```

```
weights.shape
```

(784, 300)

biases

[illegible]

biases.shape

(300, )

- 위와 같이 weights와 biases의 shape, 모델에 있는 층의 리스트 등을 확인 가능하다.

## 모델 컴파일



```
model.compile(loss="sparse_categorical_crossentropy", # loss=keras.losses.sparse_categorical_crossentropy
              optimizer="sgd", # optimizer=keras.optimizers.SGD
              metrics=["accuracy"]) # metrics=[keras.metrics.sparse_categorical_accuracy]
```

- compile시에는 cost function, optimizer를 지정할 수 있다.
- cost function에는 sparse\_categorical\_crossentropy(레이블이 정수 1개로 이뤄졌을 경우), categorical\_crossentropy(샘플마다 클래스별 타깃 확률을 갖는 경우), binary\_crossentropy(이진 분류, 다중 레이블 이진 분류) 등을 사용할 수 있다.
- optimizer는 여러가지가 있지만 책의 예시에서는 sgd를 사용하였다. sgd는 기본 확률적 경사 하강법을 사용하여 모델을 훈련시키는 알고리즘이다. sgd 사용시 학습을 또한 지정할 수 있는데 default는 0.01로 되어있다.
- metrics parameter를 통해 평가 지표를 무엇으로 할 것인지에 대해 정할 수 있다. 회귀문제가 아닌 분류 문제이므로 accuracy를 사용하였다.

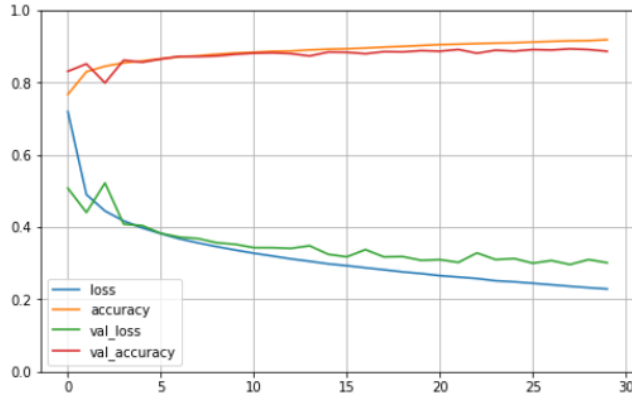
## 모델 훈련과 평가

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

- fit() 메소드를 train data를 활용해 학습을 진행할 수 있다.
- 학습 진행시 epoch, validation data등을 옵션으로 지정할 수 있다.
- history 객체에 훈련 결과를 담아서 추후에 시각화 등에 활용할 수 있다.

```
import pandas as pd
import matplotlib.pyplot as plt

pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
plt.gca().set_ylim(0, 1) # 수직축의 범위를 [0-1] 사이로 설정
plt.show()
```



- history 객체를 사용해 위와 같은 학습에 대한 loss, accuracy에 대한 시각화를 진행하였다.
- legend에 쓰여 있는 loss, accuracy는 train set에 해당하고, val\_loss, val\_accuracy는 validation set에 해당한다.

## 모델 평가

```
model.evaluate(X_test, y_test)

313/313 [=====] - 1s 2ms/step - loss: 0.3350 - accuracy: 0.8794
[0.33498960733413696, 0.8794000148773193]
```

- 모델 평가는 evaluate메소드를 통해 가능하다.
- evaluation시 test set을 통해 평가를 진행하게 된다.

## 예측

```
X_new = X_test[:3]
y_proba = model.predict(X_new) # 각 클래스에 해당될 확률 추정
y_proba.round(3)

array([[0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.002, 0.    ,
        0.998],
       [0.    , 0.    , 1.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    ,
        0.    ],
       [0.    , 1.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    ,
        0.    ]], dtype=float32)
```

- 일부 데이터를 가지고 predict()메서드를 사용해 예측을 해 보았다.

## 10.2.3 시퀀셜 API를 사용하여 회귀용 다중 퍼셉트론 만들기

```

from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()

# 훈련, 테스트 세트
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target
)

# 검증 세트
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full
)

# 스케일 조정
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)

```

- fetch api를 가지고 California 주택가격 데이터를 load 해서 train, validation, test set으로 나누었다.
- StandardScaler()를 통해 각 데이터셋을 scaling까지 해주었다.

```

model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]), # 은닉층
    keras.layers.Dense(1) # 출력층
])
model.compile(loss="mean_squared_error", optimizer="sgd") # 손실함수
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
mse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3] # 새로운 샘플
y_pred = model.predict(X_new)

```

```

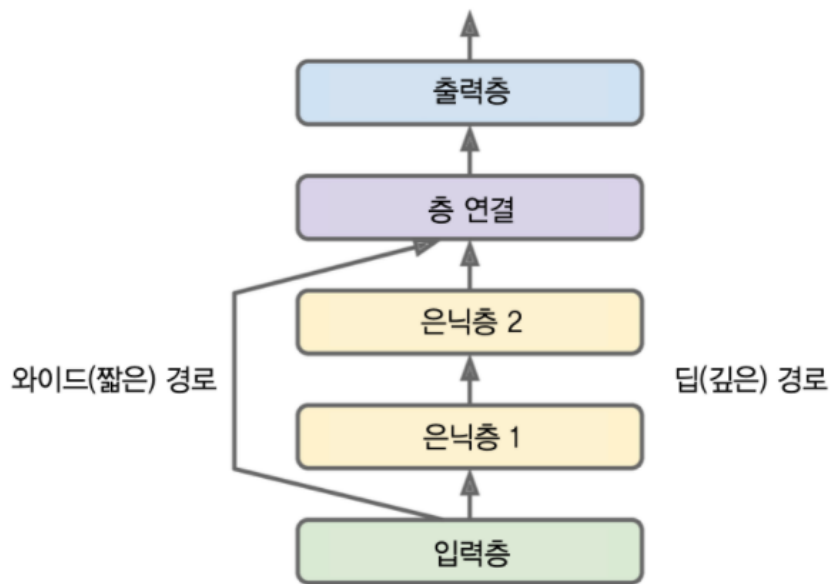
Epoch 1/20
363/363 [=====] - 1s 2ms/step - loss: 0.7506 - val_loss: 3.4619
Epoch 2/20
363/363 [=====] - 1s 1ms/step - loss: 0.5390 - val_loss: 5.3912
Epoch 3/20
363/363 [=====] - 1s 1ms/step - loss: 0.6334 - val_loss: 0.4467
Epoch 4/20
363/363 [=====] - 0s 1ms/step - loss: 0.4268 - val_loss: 0.4482

```

- 앞서 배운 내용을 통해 keras api를 사용한 model을 만들어 훈련을 진행하고, 평가, 예측까지 한 번에 진행하였다.

## 10.2.4 함수형 API를 사용해 복잡한 모델 만들기

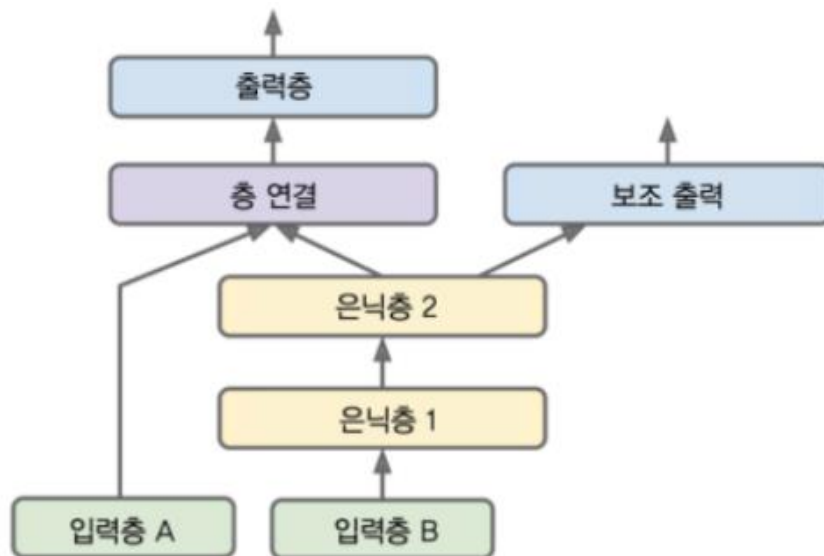
와이드 & 딥 신경망



- 짧은 경로와 깊은 경로 두 가지로 나누어 깊게 쌓은 층에는 복잡한 패턴을 학습시키고, 짧은 경로는 간단한 규칙을 학습 시키거나, 수동으로 찾은 특성을 제공하기 위해 사용한다.
- 위 도표를 함수형 API를 사용해 구현하면 아래와 같다.

```
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.Model(inputs=[input_], outputs=[output])
```

## 여러 출력 사용



- 여러 개의 출력이 필요한 경우, 가령 주요 물체 분류, 물체 중심의

좌표, 너비, 높이 예측 동일한 데이터에서 독립적인 여러 작업을 수행하는 등의 경우 위와 같이 출력을 출력층과, 보조 출력층으로 나누어 실시한다.

- 위 도표를 코드로 나타내면 아래와 같다.

```
input_A = keras.layers.Input(shape=[5], name="wide_input")
input_B = keras.layers.Input(shape=[6], name="deep_input")
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, name="main_output")(concat)
aux_output = keras.layers.Dense(1, name="aux_output")(hidden2)
model = keras.Model(inputs=[input_A, input_B], outputs=[output, aux_output])
```

### 10.2.5 서브클래싱 API로 동적 모델 만들기

```
class WideAndDeepModel(keras.Model): # Model 클래스를 상속한 다음 생성자 안에서 필요
한 층 생성
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs) # 표준 매개변수 처리 (ex. name)
        self.hidden1 = keras.layers.Dense(units, activation=activation)
        self.hidden2 = keras.layers.Dense(units, activation=activation)
        self.main_output = keras.layers.Dense(1)
        self.aux_output = keras.layers.Dense(1)

    def call(self, inputs): # call() 메서드 안에 수행하려는 연산 기술
        input_A, input_B = inputs
        hidden1 = self.hidden1(input_B)
        hidden2 = self.hidden2(hidden1)
        concat = keras.layers.concatenate([input_A, hidden2])
        main_output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return main_output, aux_output

model = WideAndDeepModel()
```

- 위 코드는 와이드 앤 딥 모델을 서브클래싱 API로 구현한 것이다.
- 위와 같이 서브클래싱 API로 구현을 하면 시퀀셜, 함수형 API에서는 활용할 수 없는 동적인 구조를 구성할 수 있다는 장점이 있다.
- call 메서드 내에서는 for문, if문, 텐서플로우 저수준 연산 등이 가능하기 때문에 모델 구성 자유도가 높다.
- 하지만 모델을 저장, 복사 하는 것이 불가능하고, summary() 메서드 사용시 층 간의 연결 정보에 대해 출력하지 않는 등의 단점이 있기 때문에 높은 유연성이 필요하지 않은 경우라면, 시퀀셜 API, 함수형 API를 사용하는 것이 좋을 것이다.

### 10.2.6 모델 저장과 복원

모델 저장

```
import tensorflow as tf
from tensorflow import keras

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

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:]

X_test = X_test / 255.0

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28,28]),
    keras.layers.Dense(300, activation="relu"), # activation=keras.activations.relu
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])

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

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

Epoch 1/30
1719/1719 [=====] - 6s 4ms/step - loss: 0.7196 - accuracy: 0.7593
- val_loss: 0.5258 - val_accuracy: 0.8208
Epoch 2/30
1719/1719 [=====] - 6s 3ms/step - loss: 0.4925 - accuracy: 0.8282
- val_loss: 0.4503 - val_accuracy: 0.8476
Epoch 3/30
1719/1719 [=====] - 6s 3ms/step - loss: 0.4507 - accuracy: 0.8417
- val_loss: 0.4301 - val_accuracy: 0.8528
Epoch 4/30
1719/1719 [=====] - 6s 3ms/step - loss: 0.4239 - accuracy: 0.8509
- val_loss: 0.4398 - val_accuracy: 0.8424
Epoch 5/30
1719/1719 [=====] - 6s 3ms/step - loss: 0.4043 - accuracy: 0.8573
- val_loss: 0.4120 - val_accuracy: 0.8566
```

- 앞서 시행했던 fashion mnist data 분류 모델이다.

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

- 모델의 저장은 save() 메소드를 사용해 가능하다. 또한 모델을 불러 오는 것은 load\_model() 메소드를 통해 아래와 같이 가능하다.

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

## 10.2.7 콜백 사용하기

### ModelCheckpoint

```
# 모델 구성, 컴파일 이후
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5")
history = model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint_cb])
```

- 훈련 중 예기치 못한 상황으로 훈련이 중단되어 이전 훈련 데이터가 유실되는 것을 막기 위해 ModelCheckpoint()함수를 통해 일정한 간격으로 모델 체크포인트를 저장할 수 있다.

save\_best\_only=True

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5",
                                                save_best_only=True)
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid),
                    callbacks=checkpoint_cb)
model = keras.models.load_model("my_keras_model.h5") # 최상의 모델로 복원
```

- validation set을 설정해 놓았을 경우 validation set을 기준으로 최상의 검증 세트 점수에서 모델을 저장한다.
- epoch를 너무 크게 잡으면 모델이 과적합 될 수 있는데, 과적합과 상관없이 최적의 모델을 저장할 수 있도록 도와줌

## EarlyStopping

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10, # 에보프 10번동안 검증
                                                  # 세트에 대한 점수가 향상되지 않으면 훈련 종료
                                                  restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, early_stopping_cb]) # 동시 사용 가능
```

- 일정 epochs동안 validation set에 대한 score가 향상되지 않으면 훈련을 종료한다.

## 사용자 정의 콜백

```
class PrintValTrainRatioCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        print("\nval/train: {:.2f}".format(logs["val_loss"]/logs["loss"]))
```

- 위와 같이 콜백 함수를 사용자가 직접 만들어 사용할 수도 있다.
- 위 함수는 훈련하는 동안 검증 손실, 훈련 손실의 비율을 출력하는 함수이다.

## 10.2.8 텐서보드를 사용해 시각화하기

### 텐서보드

- 인터랙티브 시각화 도구이다. 기능은 다음과 같다.
- 실시간 학습 곡선 시각화
- 계산 그래프 시각화
- 훈련 통계 분석
- 모델이 생성한 이미지 확인
- 3D에 투영된 복잡한 다차원 데이터 시각화
- 자동 클러스터링

```
import os
root_logdir = os.path.join(os.getcwd(), "my_logs")

def get_run_logdir():
    import time
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir()
```

```
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid),
                    callbacks=[tensorboard_cb])
```

```
Epoch 1/30
1719/1719 [=====] - 6s 4ms/step - loss: 0.2111 - accuracy: 0.9235
- val_loss: 0.3093 - val_accuracy: 0.8922
Epoch 2/30
1719/1719 [=====] - 6s 3ms/step - loss: 0.2084 - accuracy: 0.9251
- val_loss: 0.3102 - val_accuracy: 0.8896
Epoch 3/30
1719/1719 [=====] - 6s 3ms/step - loss: 0.2043 - accuracy: 0.9261
- val_loss: 0.2900 - val_accuracy: 0.8978
Epoch 4/30
```

```
%load_ext tensorboard
```

```
The tensorboard extension is already loaded. To reload it, use:
%reload_ext tensorboard
```

```
%tensorboard --logdir {"my_logs"}
```

```
<IPython.core.display.Javascript object>
```

## 10.3 신경망 하이퍼 파라미터 튜닝하기

```
1 def build_model(n_hidden=1, n_neurons=30, learning_rate=3e-3, input_shape=[8]):
2     model=keras.models.Sequential()
3     model.add(keras.layers.InputLayer(input_shape=input_shape))
4     for layer in range(n_hidden):
5         model.add(keras.layers.Dense(n_neurons, activation='relu'))
6     model.add(keras.layers.Dense(1))
7     optimizer=keras.optimizers.SGD(lr=learning_rate)
8     model.compile(loss='mse', optimizer=optimizer)
9     return model
10
```

```
1 keras_reg=keras.wrappers.scikit_learn.KerasRegressor(build_model)
```

- 하이퍼 파라미터는 GridSearchCV나 RandomizedSearch CV를 사용하여 튜닝이 가능하다.
- 하지만, 그리드 탐색이나 랜덤 탐색을 사용하려면 모델을 scikit learn 추정기 처럼 보이도록 바꿔야 한다.
- 딥러닝 하이퍼 파라미터 튜닝시에는 그리드 탐색 보다는 랜덤 탐색이 유용하다.
- 하이퍼 파라미터를 수동으로 튜닝할 때는 범위를 크게 해서 빠르게 첫 번째 탐색을 수행하고 첫 번째 탐색에서 찾은 최상의 파라미터 값으로 더 좁은 범위를 탐색하는 것이 적절하다.



- 하지만 이러한 과정을 수동으로 진행하는 것은 많은 시간이 소요되기 때문에 자동으로 탐색 지역이 좋다고 판명될 때 그 영역을 중심으로 더 좁은 영역을 탐색하는 파이썬 라이브러리들을 사용하는 것이 좋다.
- 이러한 파이썬 라이브러리의 종류는 Hyperopt, Hyperas, kopt, Talos, 케라스 튜너 등등이 있다.

### 10.3.1 은닉층 개수

- 이론적으로 hidden layer가 1개여도 뉴런의 개수가 충분하면, 아주 복잡한 함수 또한 모델링이 가능하다.
- 하지만 복잡한 문제에서는 hidden layer를 여러 개를 쌓는 것이 더 파라미터 효율성이 좋다.
- 계층 구조로 나누어 아래쪽에는 저수준의 구조를, 위쪽에는 고수준의 구조를 모델링 한다면, transfer learning에도 효율적으로 사용할 수 있다.
- 다른 목적을 가진 모델이라도, 방향성이 비슷하다면, 저수준의 구조를 가져와서 학습을 진행할 수 있다. 저수준 구조의 가중치를 random으로 설정하는 것 보다 타 모델의 저수준 구조를 가져와 가중치를 사용하게 되면 시간적, 비용적인 효율성을 달성할 수 있게 된다.

### 10.3.2 은닉층의 뉴런 개수

- 은닉층의 뉴런 개수는 보통 모든 층을 동일하게 구성해도 괜찮다.
- 원래는 층의 개수와 뉴런을 과대적합이 시작되기 전까지 점진적으로 늘려가는 것이 맞지만, 실전에서는 필요한 것보다 더 많은 층과 뉴런을 가진 모델을 선택하고 과대적합을 억제하는 early stopping이나 여러가지 규제 기법을 사용하는 것이 효율적이다.

### 10.3.3 학습률, 배치 크기 그리고 다른 하이퍼 파라미터

#### 학습률

- 최적 학습률을 찾는 기본적인 방법은 다음과 같다.

- 매우 낮은 학습률 (ex - 0.00005) 부터 점진적으로 매우 큰 학습률 (ex - 10) 까지 수백번 반복하여 모델을 훈련한다. 반복마다 일정한 값(ex -  $\exp(\log(10^{-6}/500))$ )을 학습률에 곱한다.(ex-500번)

#### 배치 크기

- 배치 크기는 GPU와 RAM의 크기를 고려하여 설정한다.
- 보통은 32와 같은 작은 단위의 배치를 이용하지만, 8192와 같이 매우 큰 배치를 사용하는 경우도 있다. 큰 배치를 사용하면 일반화 성능에 영향을 미치지 않는다는 장점이 있다.

#### 활성화 함수

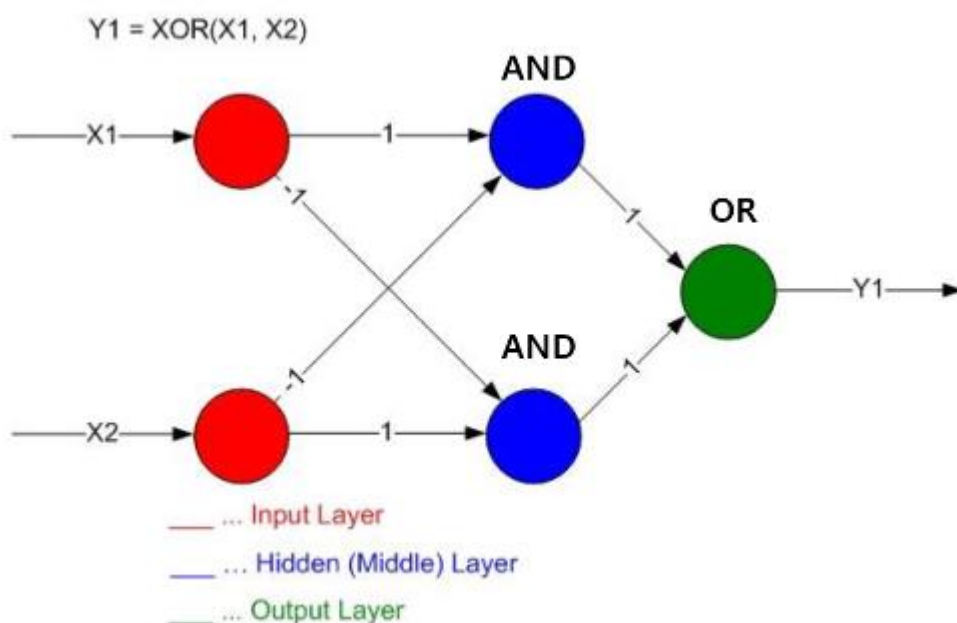
- ReLU가 가장 많이 쓰인다.

#### 반복 횟수

- 반복 횟수는 튜닝을 할 필요가 없이 early stopping을 사용하면 된다.

## 10.4 연습문제

2번.

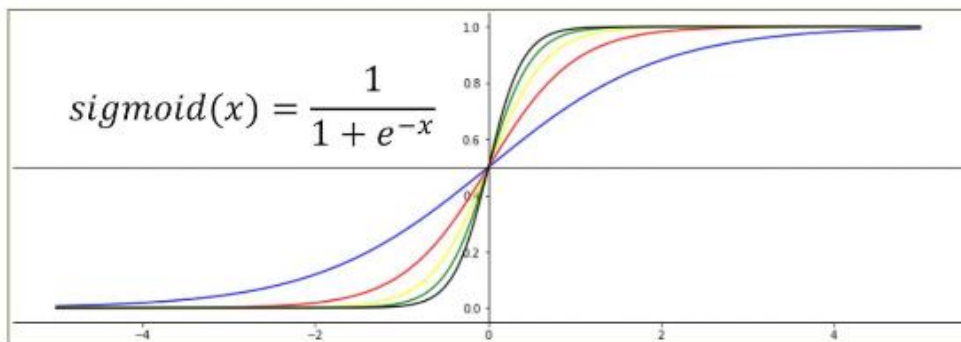


- XOR문제는 AND, OR 퍼셉트론들을 활용해 위와 같이 구현하여 문제를 해결할 수 있다.

### 3번. 퍼셉트론과 로지스틱

- 고전적인 퍼셉트론은 클래스 확률을 제공하지 않고 threshold를 넘어가는지 안 넘어가는지를 기준으로 예측을 만든다.
- 때문에 로지스틱 회귀 분류기가 일반적으로 더 선호된다.
- 퍼셉트론을 로지스틱 회귀 분류기와 동등하게 만들고 싶다면, activation function을 step function에서 logistic 활성화 함수로 변환하면 된다.

### 4번. 왜 초창기 MLP는 로지스틱이 핵심이였는가?



- 계단함수는 수평선 밖에 없으므로 activation function을 미분한 기울기를 활용하는 Backpropagation에 활용할 수 없다.
- 하지만, logistic함수는 어디서든지 0이 아닌 기울기를 가지기 때문에 backpropagation에 적합하여, 초창기엔 로지스틱 함수가 핵심적인 역할을 하였다.

### 5번. 인기 많은 활성화 함수 세 가지

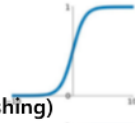
## Activation Functions

2순위 Relu에서 가중합 0이하 비활성화가 아쉬울 때

### Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

안씀(Gradient Vanishing)



### tanh

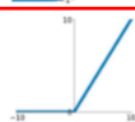
$$\tanh(x)$$

안씀(Gradient Vanishing)



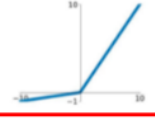
### ReLU

$$\max(0, x)$$



### Leaky ReLU

$$\max(0.1x, x)$$

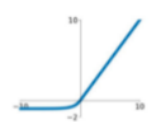


### Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

### ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



1순위

Different Activation Functions and their Graphs

- 책이 발간될 당시의 환경과 지금의 환경은 조금 차이가 있다. 현재 sigmoid와 hyperbolic tangent함수는 거의 사용하지 않는다.
- 그 대신 ReLU함수를 제일 많이 사용한다. ReLU함수는 0이하의 값은 모두 0으로 처리하는 함수이다.
- 만약 음수의 값이 0으로 일괄 처리 되는 것이 아쉽다면 Leaky ReLU 함수를 사용하면 된다.

6번. 통과 뉴런 10개로 구성된 입력층, 뉴런 50개로 구성된 은닉층, 뉴런 3개로 구성된 출력층의 다층 퍼셉트론의 구조의 계산 과정을 작성해 보아라.

$$X_{m \times n} \times W_{n \times j} + B_{m \times j} = Y_{m \times j}$$

입력층 크기 → m (배치 크기) × n (입력 뉴런)

은닉층 가중치  $W_h \rightarrow n$  (입력뉴런) × j (은닉 뉴런)

은닉층 편향 벡터  $b_h \rightarrow j$  (은닉 뉴런) ※ 편향 행렬로 원했다면  $m \times n$

$$X_{m \times n} \times W_{n \times j} + B_{m \times j} = Y_{m \times j}$$

은닉층 크기 → m (배치 크기) × j (은닉 뉴런)

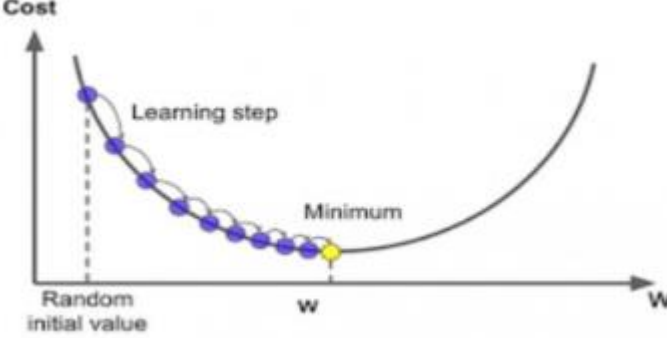
출력층 가중치  $W_o \rightarrow j$  (은닉 뉴런) × k (출력 뉴런)

출력층 편향 벡터  $b_h \rightarrow k$  (출력 뉴런) ※ 편향 행렬로 원했다면  $m \times k$

$$\rightarrow Y = \text{Relu}(\text{Relu}(X \times W_h + b_h)W_o + b_o)$$

7번. 스팸 메일을 분류하기 위해서는 몇 개의 뉴런이 필요하고 activation function으로는 무엇을 사용해야 하는가? Mnist문제에서 출력층에 어떤 activation function을 사용해야 하고 뉴런은 몇 개가 필요한가

- 스팸메일 필터를 만들 때 출력층의 뉴런 개수는 1개가 필요하다. 또

	<p>한 활성화 함수로는 ReLU나 로지스틱 함수를 사용하면 된다.</p> <ul style="list-style-type: none"> <li>MNIST의 경우 출력층의 뉴런 개수는 label의 개수대로 설정해 10개가 될 것이다. 또한 활성화 함수는 Softmax함수를 사용하게 된다.</li> </ul> <p>8번 역전파와 후진모드 자동미분</p>  <ul style="list-style-type: none"> <li>역전파는 기울기를 자동으로 계산하는 경사하강법 과정을 뜻한다.</li> <li>후진모드 자동 미분은 역전파에서 사용되는 미분 방식으로 연쇄법칙을 활용한다.</li> </ul> <p>9번. MLP에서 조정 가능한 하이퍼파라미터는? 과적합을 피하기 위해서는 어떻게 해야하는가?</p> <ul style="list-style-type: none"> <li>은닉층 개수, 뉴런 개수, 학습률, 옵티마이저, 배치 크기, 활성화 함수, 반복 횟수 등이 있다.</li> <li>과적합시에는 은닉층, 뉴런의 개수를 조정하거나 앞서 배웠던 early stopping을 사용하면 된다.</li> </ul>
특이사항	<p>스터디 발표 및 질의응답이 끝난 후 이후, 스터디 방법에 대한 논의과정에서 밑바닥부터 시작하는 딥러닝 1권에 대해 추가적으로 공부해보자는 의견이 제시되었다. 단, 밑바닥부터 시작하는 딥러닝 1권은 정식 스터디 교재로 사용하지 않고 부교재로서 사용하며, 밑바닥부터 시작하는 딥러닝에 대한 질의응답에 대해서는 정식 스터디 교재에 대한 공부가 끝난 이후에 남은 스터디 시간에 나누기로 하였다. 모든 스터디원이 이를 동의하였고 다음주 스터디부터 반영하게 되었다.</p>
비고	<p>없음</p>