

커라스와 텐서플로우

딥러닝 주요 라이브러리

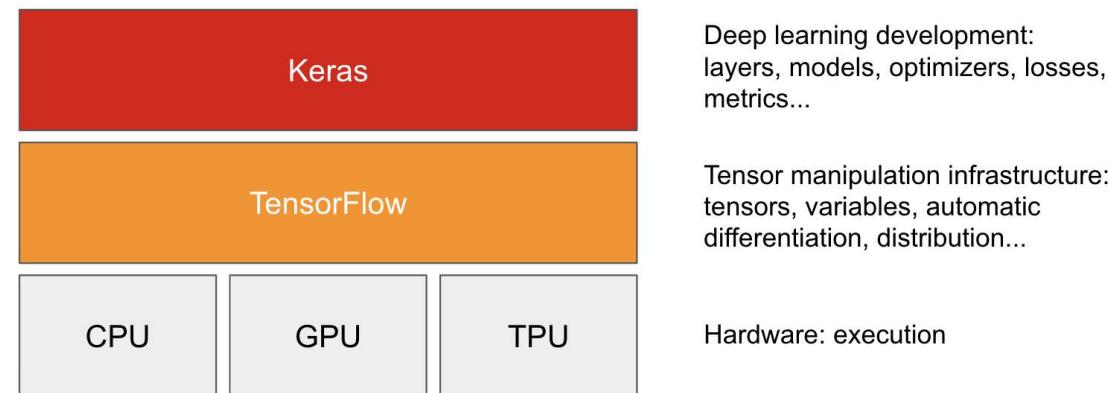
- 텐서플로우
- 케라스
- 파이토치
- JAX

텐서플로우

- 파이썬에 기반한 머신러닝 플랫폼
- 머신러닝 모델의 훈련에 필요한 텐서 연산을 지원
 - 그레이디언트 자동 계산
 - GPU, TPU 등 고성능 병렬 하드웨어 가속기 활용 가능
 - 여러 대의 컴퓨터 또는 클라우드 컴퓨팅 서비스 활용 가능
- C++(게임), 자바스크립트(웹브라우저), TFLite(모바일 장치) 등과 호환 가능
- 단순한 패키지 기능을 넘어서는 머신러닝 플랫폼
 - TF-Agents: 강화학습 연구 지원
 - TFX: 머신러닝 프로젝트 운영 지원
 - TensorFlow-Hub: 사전 훈련된 머신러닝 모델 제공

케라스

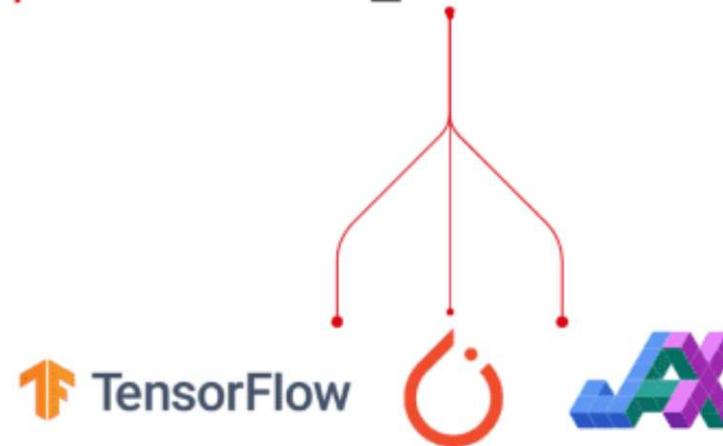
- 딥러닝 모델 구성 및 훈련에 효율적으로 사용될 수 있는 다양한 수준의 API를 제공
- 텐서플로우의 프론트엔드_{front end} 인터페이스 기능 수행



딥러닝 주요 라이브러리 약력

- 2007년: 씨아노Theano 공개. 텐서를 이용한 계산 그래프, 미분 자동화 등을 최초로 지원한 딥러닝 라이브러리.
- 2015년 3월: 케라스 라이브러리 공개. Theano를 백엔드로 사용하는 고수준 패키지.
- 2015년 11월: 텐서플로우 라이브러리 공개.
- 2016년: 텐서플로우가 케라스의 기본 백엔드로 지정됨.
- 2016년 9월: 페이스북이 개발한 파이토치PyTorch 공개.
- 2017년: Theano, 텐서플로우, CNTK(마이크로소프트), MXNet(아마존)이 케라스의 백엔드로 지원됨. 현재 Theano, CNTK 등은 더 이상 개발되지 않으며, MXNet은 아마존에서만 주로 사용됨.
- 2018년 3월: PyTorch와 Caffe2를 합친 PyTorch 출시(페이스북+마이크로소프트)
- 2019년 9월: 텐서플로우 2부터 케라스를 텐서플로우의 최상위 프레임워크로 사용
- 2023년 가을: Keras Core가 케라스 3.0으로 출시 예정. 텐서플로우, PyTorch, JAX의 프론트엔드 기능 지원.

```
import keras_core as keras
```



텐서플로우 텐서

- `tf.Tensor` 자료형
 - 상수 텐서
 - 입출력 데이터 등 변하지 않는 값을 다룰 때 사용.
 - 불변 자료형
- `tf.Variable` 자료형
 - 변수 텐서
 - 모델의 가중치, 편향 등 항목의 업데이트가 필요할 때 사용되는 텐서.
 - 가변 자료형

상수 텐서

- 다양한 방식으로 상수 텐서 생성

```
>>> x = tf.constant([[1., 2.], [3., 4.]])  
>>> print(x)  
tf.Tensor(  
[[1. 2.  
[3. 4.]], shape=(2, 2), dtype=float32)  
  
>>> x = tf.ones(shape=(2, 1))  
>>> print(x)  
tf.Tensor(  
[[1.  
[1.]], shape=(2, 1), dtype=float32)  
  
>>> x = tf.zeros(shape=(2, 1))  
>>> print(x)  
tf.Tensor(  
[[0.  
[0.]], shape=(2, 1), dtype=float32)
```

- `normal()` 함수: 정규 분포 활용

```
>>> x = tf.random.normal(shape=(3, 1), mean=0., stddev=1.)  
>>> print(x)  
tf.Tensor(  
[[ -0.5644841 ]  
[ -0.76016265]  
[  0.30502525]], shape=(3, 1), dtype=float32)
```

- `uniform()` 함수: 균등 분포 활용

```
>>> x = tf.random.uniform(shape=(3, 1), minval=0., maxval=1.)  
>>> print(x)  
tf.Tensor(  
[[ 0.33661604]  
[ 0.09824598]  
[ 0.32487237]], shape=(3, 1), dtype=float32)
```

상수 텐서의 수정 불가능성

```
>>> x[0, 0] = 1.0
TypeError: ..... Traceback (most recent call last)
<ipython-input-7-242a5d4d3c4a> in <module>
----> 1 x[0, 0] = 1.0

TypeError: 'tensorflow.python.framework.ops.EagerTensor' object does not
support item assignment
```

텐서 항목의 자료형

```
>>> type(x[0, 0])
tensorflow.python.framework.ops.EagerTensor
```

변수 텐서

```
>>> v = tf.Variable(initial_value=tf.random.normal(shape=(3, 1)))
>>> print(v)
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[-1.3837979 ],
       [-0.23704937],
       [-0.9790895 ]], dtype=float32)>
```

변수 텐서 교체

```
>>> v.assign(tf.ones((3, 1)))
<tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, numpy=
array([[1.],
       [1.],
       [1.]], dtype=float32)>
```

- 대체하는 텐서의 모양(shape)이 기존 텐서의 모양과 동일해야 함.

```
>>> v.assign(tf.ones((3, 2)))
ValueError                                Traceback (most recent call
last)
<ipython-input-13-e381ab0c94e6> in <module>
     1 v.assign(tf.ones((3, 2)))

~/Anaconda3/lib/site-
packages/tensorflow/python/ops/resource_variable_ops.py in assign(self,
value, use_locking, name, read_value)
    886         else:
    887             tensor_name = " " + str(self.name)
--> 888             raise ValueError(
    889                 ("Cannot assign to variable %s due to variable shape
%s and value %s"
    890                  "shape %s are incompatible") %

ValueError: Cannot assign to variable Variable:0 due to variable shape
(3, 1) and value shape (3, 2) are incompatible
```

변수 텐서 항목 수정

```
>>> v[0, 0].assign(3.)  
<tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, numpy=  
array([[3.],  
       [1.],  
       [1.]], dtype=float32)>
```

assign_add() 와 assign_sub()

- assign_sub() 메서드는 `-=` 연산자
- assign_add() 메서드는 `+=` 연산자

```
>>> v.assign_sub(tf.ones((3, 1)))
<tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, numpy=
array([[2.],
       [0.],
       [0.]], dtype=float32)>
```

텐서플로우 활용법 기초

그레이디언트 테이프

$$f(x) = x^2 \implies \nabla f(x) = \frac{df(x)}{dx} = 2x$$

```
>>> input_var = tf.Variable(initial_value=3.)  
  
>>> with tf.GradientTape() as tape:  
...   result = tf.square(input_var)  
  
>>> gradient = tape.gradient(result, input_var)  
  
>>> print(gradient)  
tf.Tensor(6.0, shape=(), dtype=float32)
```

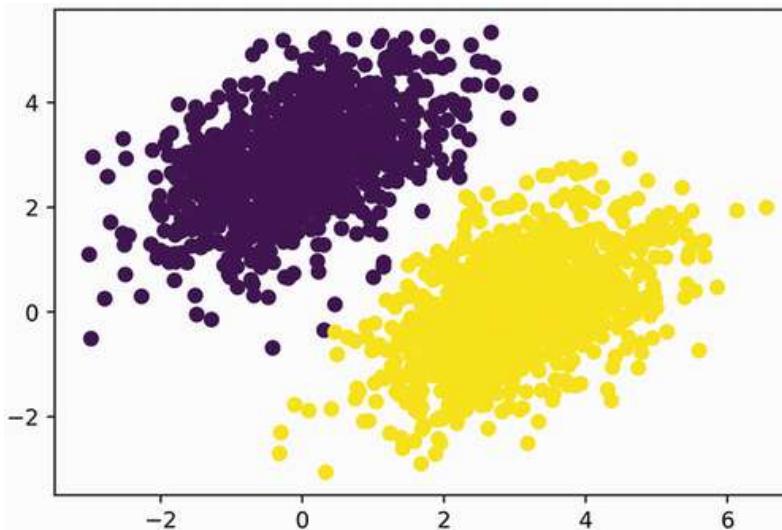
예제: 선형 이진 분류기

1단계: 데이터셋 생성

```
num_samples_per_class = 1000

# 음성 데이터셋
negative_samples = np.random.multivariate_normal(
    mean=[0, 3], cov=[[1, 0.5],[0.5, 1]], size=num_samples_per_class)

# 양성 데이터셋
positive_samples = np.random.multivariate_normal(
    mean=[3, 0], cov=[[1, 0.5],[0.5, 1]], size=num_samples_per_class)
```



음성 데이터셋과 양성 데이터셋을 합쳐서 훈련셋을 생성한다.

- `negative_sample`: (1000, 2) 모양의 텐서
- `positive_sample`: (1000, 2) 모양의 텐서
- `inputs = np.vstack(negative_sample, positive_sample)`: (2000, 2) 모양의 텐서
 - `negative_sample` 데이터셋이 0번부터 999번까지 인덱스.
 - `positive_sample` 데이터셋이 1000번부터 1999번까지 인덱스.

```
inputs = np.vstack((negative_samples, positive_samples)).astype(np.float32)
```

음성 샘플의 레이블은 0, 양성 샘플의 레이블은 1로 지정한다.

- `np.zeros((num_samples_per_class, 1), dtype="float32")`: (1000, 1) 모양의 어레이. 0으로 채워짐. 0번부터 999번 인덱스까지의 모든 음성 데이터의 타깃은 0임.
- `np.ones((num_samples_per_class, 1), dtype="float32")`: (1000, 1) 모양의 어레이. 1로 채워짐. 999번부터 1999번 인덱스까지의 모든 양성 데이터의 타깃은 1임.
- `targets`: (2000, 1) 모양의 어레이.

```
targets = np.vstack((np.zeros((num_samples_per_class, 1), dtype="float32"),
                     np.ones((num_samples_per_class, 1), dtype="float32")))
```

2단계: 선형 회귀 모델 훈련에 필요한 가중치 변수 텐서와 편향 변수 텐서 생성

선형 분류기 모델의 예측값 계산은 다음과 같이 아핀 변환으로 이뤄진다.

```
inputs @ W + b

input_dim = 2      # 입력 샘플의 특성이 2개
output_dim = 1     # 각각의 입력 샘플에 대해 하나의 부동소수점을 예측값으로 계산

# 가중치: (2, 1) 모양의 가중치 행렬을 균등분포를 이용한 무작위 초기화
W = tf.Variable(initial_value=tf.random.uniform(shape=(input_dim,
output_dim)))

# 편향: (1,) 모양의 벡터를 0으로 초기화
b = tf.Variable(initial_value=tf.zeros(shape=(output_dim,)))
```

- inputs: (2000, 2) 모양의 입력 데이터셋 행렬
- W: (2, 1) 모양의 가중치 행렬
- inputs @ W: (2000, 1) 모양의 행렬
- b: (1,) 모양의 편향 벡터
- inputs @ W + b: (2000, 1) 모양의 출력값 행렬. 즉, 2000 개의 입력 데이터 각각에 대해 하나의 값의 계산됨.

3단계: 모델 선언(포워드 패스)

```
def dense(inputs, W, b, activation=None):
    ... outputs = tf.matmul(inputs, W) + b
    ... if activation != None:
    ...     return activation(outputs)
    ... else:
    ...     return outputs

def model(inputs):
    ... outputs = dense(inputs, W, b)
    ... return outputs
```

4단계: 손실 함수 지정

$$\frac{1}{m_b} \sum (y - \hat{y})^2$$

```
def square_loss(targets, predictions):
    ... per_sample_losses = tf.square(targets - predictions)
    ... return tf.reduce_mean(per_sample_losses)
```

5단계: 훈련 스텝(역전파) 지정

```
def training_step(inputs, targets):
    """
    - inputs: 입력 데이터 배치
    - targets: 타깃 배치
    """

    # 손실 함수의 그레이디언트 계산 준비
    with tf.GradientTape() as tape:
        predictions = model(inputs)
        loss = square_loss(targets, predictions)

    # 가중치와 편향에 대한 손실 함수의 그레이디언트 계산
    grad_loss_wrt_W, grad_loss_wrt_b = tape.gradient(loss, [W, b])

    # 가중치 행렬과 편향 벡터 업데이트
    W.assign_sub(grad_loss_wrt_W * learning_rate) # 가중치 행렬 업데이트
    b.assign_sub(grad_loss_wrt_b * learning_rate) # 편향 업데이트

    return loss
```

6단계: 훈련 루프 지정

- 반복해서 훈련한 내용을 출력
- 설명을 간단하게 하기 위해 전체 데이터셋을 하나의 배치로 사용하는 훈련 구현

```
for step in range(40):
    ...     loss = training_step(inputs, targets)
    ...     print(f"Loss at step {step}: {loss:.4f}")
```

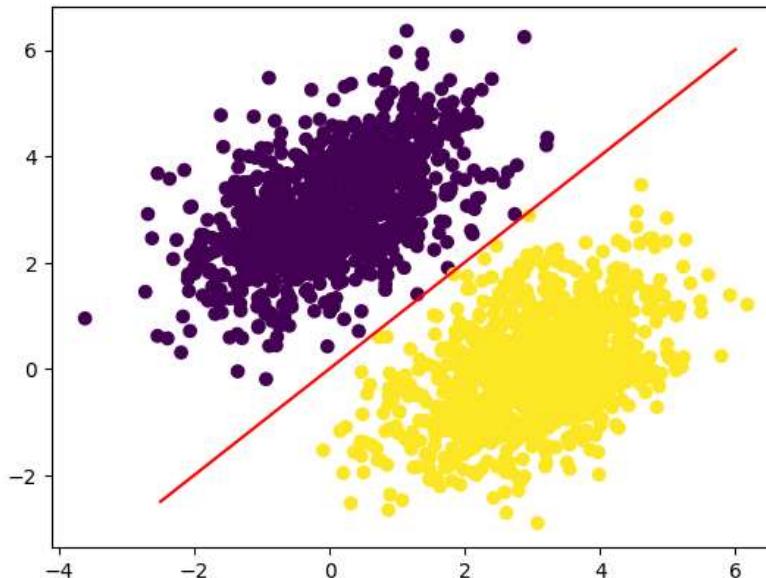
7단계: 결정경계

결정경계를 직선으로 그리려면 아래 일차 함수를 이용한다.

$$y = -W[0] / W[1] * x + (0.5 - b) / W[1]$$

이유는 아래 식으로 계산되는 모델의 예측값이 0.5보다 큰지 여부에 따라 양성/음성이 판단되기 때문이다.

$$W[0]*x + W[1]*y + b$$



케라스 신경망 모델의 핵심 API

- 신경망 모델은 층으로 구성됨
- 모델에 사용되는 층의 종류와 층을 쌓는 방식에 따라 모델이 처리할 수 있는 데이터와 훈련 방식이 달라짐
- 케라스 라이브러리가 층을 구성하고 훈련 방식을 관리하는 다양한 API 제공

층

- 입력 데이터를 지정된 방식에 따라 다른 모양의 데이터로 변환하는 **포워드 패스**
forward pass 담당
- 데이터 변환에 사용되는 가중치_{weight}와 편향_{bias} 저장

층의 종류

층의 종류에 따라 입력 배치 데이터셋 텐서의 모양이 달라진다.

- Dense 클래스
 - 밀집층 생성
 - (배치 크기, 특성 수) 모양의 2D 텐서로 입력된 데이터셋 처리.
- LSTM 또는 Conv1D 클래스
 - 순차 데이터와 시계열 데이터 분석에 사용되는 순환층 생성
 - (배치 크기, 타임스텝 수, 특성 수) 모양의 3D 텐서로 입력된 순차 데이터셋 처리.
- Conv2D 클래스
 - 합성곱 신경망(CNN) 구성에 사용되는 합성곱층 생성
 - (배치 크기, 가로, 세로, 채널 수) 모양의 4D 텐서로 제공된 이미지 데이터셋 처리.

tf.keras.layers.Layer 클래스

- 케라스의 모든 층 클래스는 `tf.keras.layers.Layer` 클래스를 상속한다.
- 상속되는 `__call__()` 메서드:
 - 가중치와 편향 텐서의 초기화. 가중치와 편향이 이미 생성되어 있다면 새로 생성하지 않고 그대로 사용.
 - 입력 데이터셋을 출력 데이터셋으로 변환하는 포워드 패스를 수행
- `__call__()` 메서드가 하는 일

```
def __call__(self, inputs):
    if not self.built:
        self.build(inputs.shape)
        self.built = True
    return self.call(inputs)
```

Dense 클래스 직접 구현하기

```
from tensorflow import keras

class SimpleDense(keras.layers.Layer):
    def __init__(self, units, activation=None):
        super().__init__()
        self.units = units           # 유닛 개수 지정
        self.activation = activation # 활성화 함수 지정

    # 가중치와 편향 초기화
    def build(self, input_shape):
        input_dim = input_shape[-1] # 입력 샘플의 특성 수
        self.W = self.add_weight(shape=(input_dim, self.units),
                               initializer="random_normal")
        self.b = self.add_weight(shape=(self.units,), initializer="zeros")

    # 데이터 변환(포워드 패스)
    def call(self, inputs):
        y = tf.matmul(inputs, self.W) + self.b
        if self.activation is not None:
            y = self.activation(y)
        return y
```

SimpleDense 층의 데이터 변환

- 유닛 수: 512개

- 활성화 함수: `relu`

```
>>> my_dense = SimpleDense(units=512, activation=tf.nn.relu)
```

- 128: 배치 크기

- 784: MNIST 데이터셋의 손글씨 이미지 한 장의 특성 수($28 * 28 = 128$)

```
>>> input_tensor = tf.ones(shape=(128, 784))
```

- 층의 데이터 변환 결과

```
>>> output_tensor = my_dense(input_tensor)
>>> print(output_tensor.shape)
(128, 512)
```

내부적으로는 `__call__()` 메서드가 호출됨:

- 가중치 텐서와 와 편향 텐서가 생성되지 않은 경우
 - (784, 512) 모양의 가중치 텐서 `W` 생성 및 무작위 초기화. 782는 입력 샘플의 특성 수, 512는 층의 유닛 수.
 - (512,) 모양의 편향 텐서 `b` 생성 및 0으로 초기화. 512는 층의 유닛 수.
 - 포워드 패스: 생성된 가중치와 편향을 이용하여 출력값 계산.
- 가중치 텐서와 와 편향 텐서가 생성되어 있는 경우. 즉 훈련이 반복되는 경우.
 - 포워드 패스: 역전파로 업데이트된 가중치와 편향을 이용하여 출력값 계산.

모델

- 케라스의 모든 모델 클래스는 `tf.keras.Model` 클래스를 상속
- 아래 모델 직접 구현

```
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

```
class MyMNistModel(keras.Model):
    def __init__(self):
        super().__init__()
        self.layer_1 = SimpleDense(units=512, activation=tf.nn.relu)
        self.layer_2 = SimpleDense(units=10, activation=tf.nn.softmax)

    def call(self, inputs):
        features = self.layer_1(inputs)
        outputs = self.layer_2(features)
        return outputs

model = MyMNistModel()
```

`keras.layers.Dense` 층을 이용한다면 다음과 같이 활성화 함수를 문자열로 지정할 수 있다.

```
class MyMNistModel(keras.Model):
    def __init__(self):
        super().__init__()
        self.layer_1 = keras.layers.Dense(units=512, activation="relu")
        self.layer_2 = keras.layers.Dense(units=10, activation="softmax")

    def call(self, inputs):
        features = self.layer_1(inputs)
        outputs = self.layer_2(features)
        return outputs
```

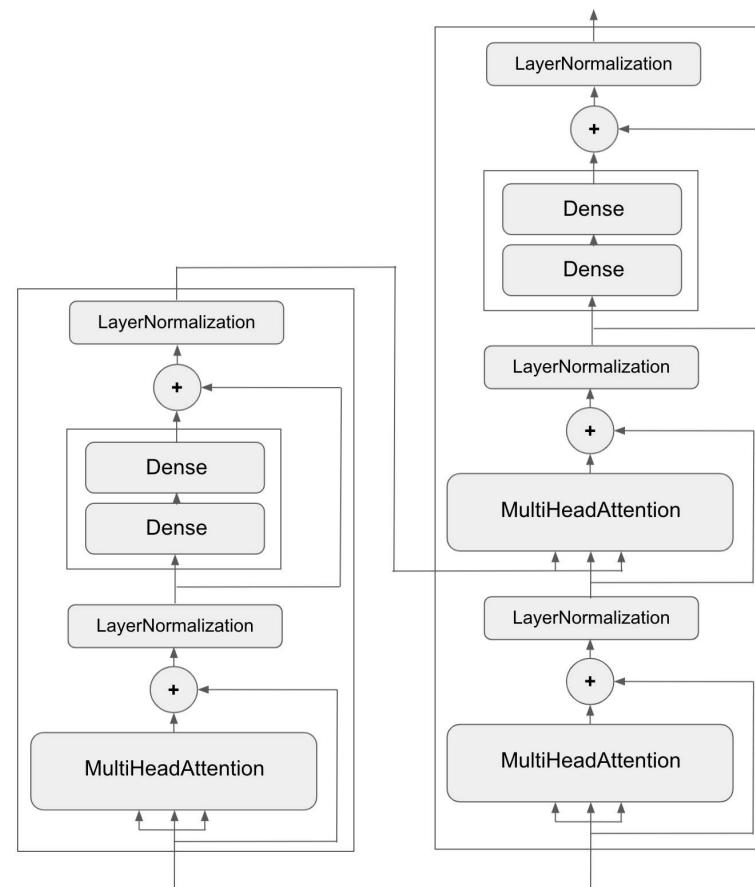
모델을 하나의 층으로 활용하기

- 기존에 정의된 모델을 다른 모델을 구성할 때 하나의 층으로 활용할 수도 있다.
- 이런 이유로 `tf.keras.Model` 클래스는 `tf.keras.layers.Layer` 클래스를 상속하도록 설계되어 있다.

모델의 학습과정과 층의 구성

- 모델의 학습과정은 전적으로 층의 구성방식에 의존한다.
- 층의 구성 방식은 주어진 데이터셋과 모델이 해결해야 하는 문제에 따라 달라진다.
- 층을 구성할 때 특별히 정해진 규칙은 없다.
- 문제 유형에 따른 권장 모델이 다양하게 개발되어 있다.

예제: 자연어 처리에 사용되는 트랜스포머 Transformer 모델



모델 컴파일

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

문자열	파이썬 객체
"rmsprop"	keras.optimizers.RMSprop()
"mean_squared_error"	keras.losses.sparse_categorical_crossentropy()
"accuracy"	keras.metrics.Accuracy()

지정된 문자열을 사용하는 대신 파이썬 객체를 직접 지정해도 된다.

```
model.compile(optimizer=keras.optimizers.RMSprop(),
              loss=keras.losses.sparse_categorical_crossentropy(),
              metrics=[keras.metrics.Accuracy()])
```

문자열을 사용하지 못하는 경우

아래와 같은 경우에도 문자열 대신 해당 객체를 지정해야 한다.

- 기본값과 다른 학습률(learning_rate)을 사용하는 옵티마이저를 지정하는 경우
- 사용자가 직접 구현한 옵티마이저, 손실 함수, 평가지표 등을 사용하는 경우

```
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=1e-4),  
              loss=사용자정의손실함수객체,  
              metrics=[사용자정의평가지표_1, 사용자정의평가지표_2])
```

옵티마이저 종류

다양한 옵티마이저의 장단점에 대해서는 [Hands-on Machine Learning 3판](#)의 11장에 정리되어 있다.

- SGD (with or without momentum)
- RMSprop
- Adam
- Adagrad

등등.

손실 함수 종류

일반적으로 모델의 종류에 따라 손실 함수를 선택한다.

- CategoricalCrossentropy
- SparseCategoricalCrossentropy
- BinaryCrossentropy
- MeanSquaredError
- KLDivergence
- CosineSimilarity

등등.

평가 지표 종류

일반적으로 모델 종류와 목적에 따라 평가 지표를 선택한다.

- CategoricalAccuracy
- SparseCategoricalAccuracy
- BinaryAccuracy
- AUC
- Precision
- Recall

등등.

훈련 루프

- `fit()` 메서드를 호출
- 스텝 단위로 반복
- 지정된 에포크 만큼 또는 학습이 충분히 이루어졌다는 평가가 내려질 때까지 훈련 반복

지도학습 모델 훈련

```
# MNIST 데이터셋 준비
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = 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

# 모델 훈련
training_history = model.fit(train_images, train_labels, epochs=5,
batch_size=128)
```

History 객체: 훈련 결과

- 모델의 훈련 결과 History 객체
- History 객체의 history 속성: 에포크별로 계산된 손실값과 평가지표값 저장

```
>>> training_history.history
... {'loss': [0.2772371768951416,
... 0.11135932803153992,
... 0.07323136925697327,
... 0.05248244106769562,
... 0.03955799713730812],
... 'accuracy': [0.9214833378791809,
... 0.9670000076293945,
... 0.9781500101089478,
... 0.9845499992370605,
... 0.988349974155426]}
```

훈련 중 모델 검증

- 검증셋 활용 훈련
- 아래 코드는 무작위적으로 훈련셋을 7:3으로 훈련용과 검증용으로 나눔

```
# 훈련 데이터셋 인덱스 무작위 섞기
indices_permutation = np.random.permutation(len(train_images))

# 훈련 데이터셋 무작위 섞기
shuffled_inputs = train_images[indices_permutation]
shuffled_targets = train_labels[indices_permutation]

# 인덱스의 30%를 활용하여 훈련용과 검증용을 7대 3으로 분리
num_validation_samples = int(0.3 * len(train_images))

# 검증용
val_inputs = shuffled_inputs[:num_validation_samples]
val_targets = shuffled_targets[:num_validation_samples]

# 훈련용
training_inputs = shuffled_inputs[num_validation_samples:]
training_targets = shuffled_targets[num_validation_samples:]
```

```
model.fit(  
    ... training_inputs,  
    ... training_targets,  
    ... epochs=5,  
    ... batch_size=128,  
    ... validation_data=(val_inputs, val_targets)  
)
```

```
>>> training_history.history
{'loss': [0.030107110738754272,
 0.021242942661046982,
 0.015943283215165138,
 0.011588473804295063,
 0.00903001707047224],
'accuracy': [0.9907857179641724,
 0.9943333268165588,
 0.9962618947029114,
 0.9973333477973938,
 0.9980000257492065],
'val_loss': [0.031793683767318726,
 0.034385889768600464,
 0.033837877213954926,
 0.029722534120082855,
 0.029139608144760132],
'val_accuracy': [0.9900000095367432,
 0.9891111254692078,
 0.9888333082199097,
 0.989333315849304,
 0.9904444217681885]}
```

훈련 후 모델 검증

```
>>> loss_and_metrics = model.evaluate(test_images, test_labels,  
batch_size=128)  
79/79 [=====] - 0s 2ms/step - loss: 0.0619 -  
accuracy: 0.9819  
  
>>> print(loss_and_metrics)  
[0.06193564459681511, 0.9818999767303467]
```

예측

- 방식 1: 모델 적용
- 모델을 마치 함수처럼 이용
 - predictions = model(test_images)
- 내부적으로 앞서 설명한 `__call__()` 메서드가 실행됨
- 배치 사용하지 않음.

- 방식 2: predict() 메서드 호출

```
predictions = model.predict(test_images, batch_size=128)
```