

케라스와 텐서플로우

# 딥러닝 주요 라이브러리

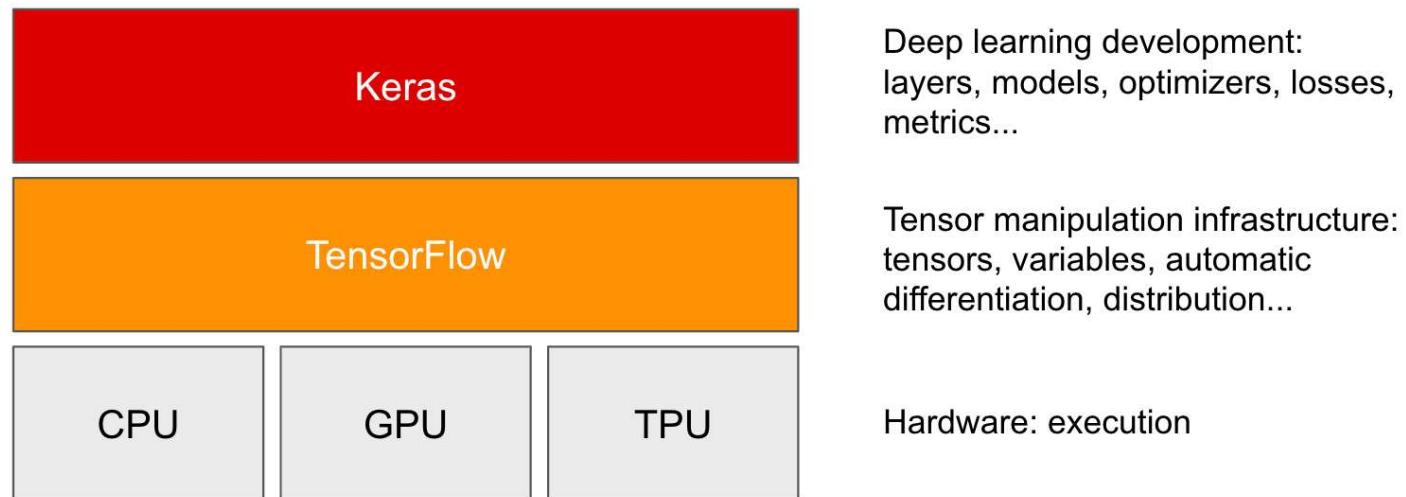
- 텐서플로우
- 케라스 3.0
- 파이토치
- JAX

## 텐서플로우

- 파이썬에 기반한 머신러닝 플랫폼
- 머신러닝 모델의 훈련에 필요한 텐서 연산 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

# 케라스

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



# 케라스 3.0, 텐서플로우, 파이토치, 잭스

- 파이토치 PyTorch: 텐서 연산을 지원하는 딥러닝 라이브러리
- 잭스 Jax: 고성능 딥러닝 프레임워크. LLM 처럼 거대 모델 훈련을 위해 구글에서 만든 라이브러리
- 케라스 3.0: TensorFlow, PyTorch, JAX 의 프론트엔드 기능 지원. 백엔드 선택과 상관 없이 동일한 케라스 코드 활용 가능

The terminal window displays the following code:

```
import keras

model = keras.Sequential([
    keras.layers.Input(shape=(num_features,)),
    keras.layers.Dense(512, activation="relu"),
    keras.layers.Dense(512, activation="relu"),
    keras.layers.Dense(num_classes, activation="softmax"),
])
model.summary()

model.compile(
    optimizer=keras.optimizers.AdamW(learning_rate=1e-3),
    loss=keras.losses.CategoricalCrossentropy(),
    metrics=[
        keras.metrics.CategoricalAccuracy(),
        keras.metrics.AUC(),
    ],
)

history = model.fit(
    x_train, y_train, batch_size=64, epochs=8, validation_split=0.2
)
evaluation_scores = model.evaluate(x_val, y_val, return_dict=True)
predictions = model.predict(x_test)
```

Three separate command-line outputs are shown to the right of the terminal window, each associated with a logo:

- \$ python example.py  
Using TensorFlow backend
- \$ python example.py  
Using PyTorch backend
- \$ python example.py  
Using JAX backend

## 딥러닝 개발환경

- GPU, TPU 등을 활용한 딥러닝 신경망 모델 훈련 추천
- 구글 코랩: 특별한 준비 없이 바로 신경망 모델을 GPU, TPU 등과 함께 훈련시킬 수 있음.
- 윈도우 11 운영체제에서 GPU를 지원하는 개발환경:  
<https://github.com/codingalzi/dlp2/blob/master/INSTALL.md>
- 구글 클라우드 플랫폼 또는 아마존 웹서비스(AWS EC2) 활용 가능

# 케라스 핵심 API

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

## 층 API

- 입력 데이터를 지정된 방식에 따라 다른 모양의 데이터로 변환하는 **순전파** forward pass 담당
- 데이터 변환에 사용되는 가중치 weight와 편향 bias 저장

## Dense 층 상세

MNIST 데이터셋을 이용한 다중 클래스 분류에 사용된 모델을 구성하는 Dense 층을 직접 구현하기

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

# SimpleDense 클래스 선언

tf.keras.layers.Layer 클래스 상속 필수

```
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

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

- 유닛 수: 512개
- 활성화 함수: relu

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

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

```
output_tensor = my_dense(input_tensor)
```

```
print(output_tensor.shape)
```

```
(128, 512)
```

## Layer 클래스의 `__call__()` 매직 메서드

- `my_dense(input_tensor)`: `my_dense` 를 마치 함수호출하듯이 활용
- 내부적으로는 `Layer` 클래스의 `__call__()` 메서드가 호출됨:  
`my_dense.__call__(input_tensor)`

## tf.keras.layers.Layer 클래스

`__call__()` 메서드 상속:

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

## my\_dense.\_\_call\_\_(input\_tensor)

```
y = input_tensor @ W + b # 아핀 변환  
y = tf.nn.relu(y) # 활성화 함수 적용
```

- `build()` 메서드 호출. 가중치 텐서와 편향 텐서가 존재하지 않은 경우 아래 내용 실행.
  - (784, 512) 모양의 가중치 텐서 `W` 생성 및 무작위 초기화.
  - (512, ) 모양의 편향 벡터 `b` 생성 및 0으로 초기화.
- `call()` 메서드 호출. 저장되어 있는 가중치 텐서와 편향 벡터를 이용하여 입력 데 이터셋을 아래 방식으로 변환.

## 모델 API

예제: 아래 모델 직접 구현하기

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

## MySequential 클래스 직접 구현

케라스의 모든 모델 클래스는 tf.keras.Model 클래스를 상속

```
class MySequential(keras.Model):
    def __init__(self, list_layers): # 층들의 리스트 지정
        super().__init__()
        self.list_layers = list_layers

    # 포워드 패스: 층과 층을 연결하는 방식으로 구현
    def call(self, inputs):
        outputs = inputs
        for layer in self.list_layers:
            outputs = layer(outputs)
        return outputs
```

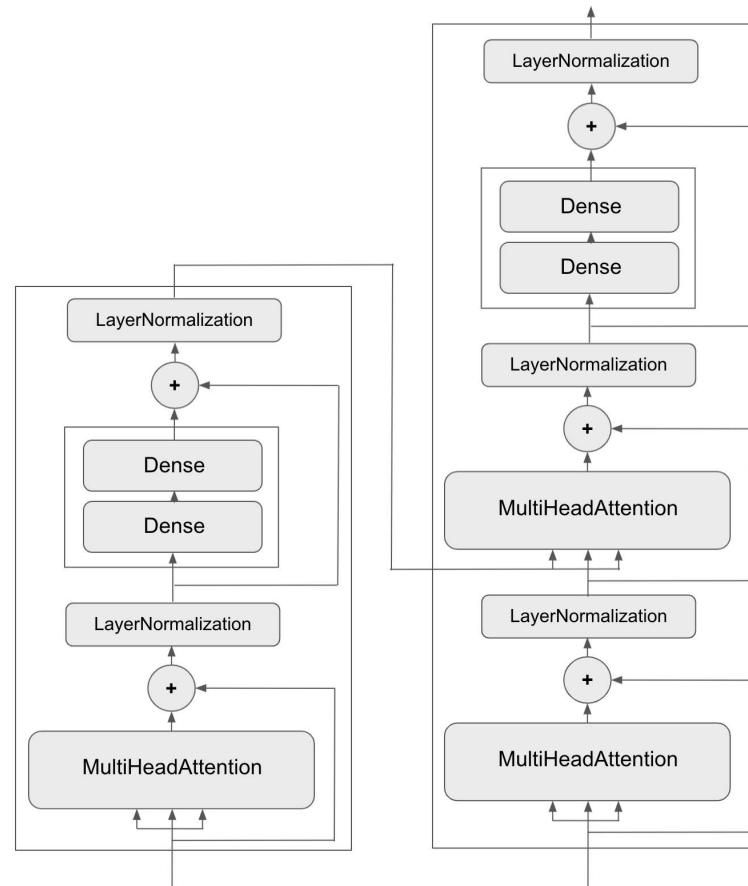
## MNIST 모델 구현

```
layer_1 = SimpleDense(units=512, activation=tf.nn.relu) # 첫째 밀집층  
layer_2 = SimpleDense(units=10, activation=tf.nn.softmax) # 둘째 밀집층  
  
model = MySequential([layer_1, layer_2])
```

## 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 모델



## 모델 컴파일 API

모델 컴파일은 선언된 모델을 진행하기 위해 필요한 다음 세 가지 설정을 추가로 지정하는 과정을 가리킨다.

- 손실 함수
  - 훈련 중 모델의 성능이 얼마나 나쁜지 측정.
  - 가중치와 편향 의존.
  - 가중치와 편향에 대해 미분 가능해야 함.
  - 옵티마이저가 경사하강법을 적용할 때 사용되는 함수.
- 옵티마이저
  - 가중치와 편향을 업데이트하는 역전파 반복 실행
- 평가지표
  - 훈련과 테스트 과정을 모니터링 할 때 사용되는 모델 평가 지표.
  - 손실 함수와는 달리 훈련에 직접 사용되지는 않음.

## 케라스와 문자열

케라스가 지원하는 많은 API는 문자열로 지정 가능.

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

## 문자열을 사용하지 못하는 경우

다음 두 가지의 경우엔 문자열 대신 해당 객체를 지정해야 한다.

- 예를 들어, 기본값과 다른 학습률(learning\_rate)을 사용하는 옵티마이저를 지정하는 경우
- 사용자가 직접 정의한 객체를 사용하는 경우

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

# 옵티마이저 API

옵티마이저	문자열
keras.optimizers.SGD	"SGD"
keras.optimizers.RMSprop	"rmsprop"
keras.optimizers.Adam	"adam"
keras.optimizers.AdamW	"adamw"
keras.optimizers.Adagrad	"adagrad"
keras.optimizers.Nadam	"nadam"

# 손실 함수 API

손실 함수	문자열	용도
keras.losses.CategoricalCrossentropy	"categorical_crossentropy"	다중 클래스 분류. 원-핫 형식 타깃
keras.losses.SparseCategoricalCrossentropy	"sparse_categorical_crossentropy"	다중 클래스 분류. 정수 타깃
keras.losses.BinaryCrossentropy	"binary_crossentropy"	이진 분류
keras.losses.MeanSquaredError	"mean_squared_error"	회귀
keras.losses.MeanAbsoluteError	"mean_absolute_error"	회귀
keras.losses.CosineSimilarity	"cosine_similarity"	문장 번역, 물건 추천, 이미지 분류 등

# 평가지표 API

평가지표	문자열	용도
keras.metrics.CategoricalAccuracy	"categorical_accuracy"	다중클래스 분류 정확도 측정. 원-핫 형식 타깃
keras.metrics.SparseCategoricalAccuracy	"sparse_categorical_accuracy"	다중클래스 분류 정확도 측정. 정수 타깃
keras.metrics.BinaryAccuracy	"binary_accuracy"	이진 분류 정확도 측정
keras.metrics.AUC	없음	다중 클래스 분류 AUC 측정
keras.metrics.Precision	없음	다중 클래스 분류 정밀도 측정
keras.metrics.Recall	없음	다중 클래스 분류 재현율 측정

## 모델 훈련 API

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

## 지도학습 모델 훈련

```
training_history = model.fit(  
    ...,  
    inputs,  
    ...,  
    targets,  
    ...,  
    epochs=5,  
    ...,  
    batch_size=128  
)
```

## History 객체: 훈련 결과

- 모델의 훈련 결과를 담은 History 객체 반환
- History 객체의 history 속성: 에포크별로 계산된 손실값과 평가지표값 저장

```
training_history.history
```

```
{'loss': [0.2729695439338684,
          0.11179507523775101,
          0.07302209734916687,
          0.0526457279920578,
          0.04022042825818062],
  'accuracy': [0.9212833046913147,
               0.9672333598136902,
               0.9783666729927063,
               0.9844833612442017,
               0.988099992275238]}
```

## 검증셋 활용

- 머신러닝 모델 훈련의 목표:
  - 훈련셋에 대한 높은 성능이 아니라 훈련에서 보지 못한 새로운 데이터에 대한 높은 성능
- 새로운 데이터에 대한 모델의 성능 예측: 증용 데이터셋을 훈련중에 활용
- 검증셋: 주어진 훈련셋의 20~30 % 정도를 검증셋으로 지정
  - 훈련셋의 크기에 따라 검증셋의 비율을 적절하게 조정
  - 훈련셋 자체가 매우 작은 경우: K-겹 교차 검증 사용 추천
- 훈련셋과 검증셋이 서로 겹치지 않도록 해야 함.

## 검증셋 활용 모델 훈련

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

## 검증셋에 대한 에포크별 손실값 평가지표

```
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,  
0.9893333315849304,  
0.9904444217681885]}
```

## 모델 평가 API

- 훈련이 종료된 모델을 실전 성능 평가를 `evaluate()` 메서드를 테스트셋과 함께 호출
- 모델의 실전 성능 평가 또한 지정된 배치 단위로 실행됨.
- 반환값: 테스트셋에 대한 손실값과 평가지표값을 담은 리스트

```
loss_and_metrics = model.evaluate(val_inputs, val_targets, batch_size=128)  
print(loss_and_metrics)  
[0.29411643743515015, 0.5333333611488342]
```

## 모델 활용 API

- 실전에 배치된 모델은 새로이 입력된 데이터에 대한 예측을 실행
- `predict()` 메서드 호출. 배치 사용

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