

케라스 신경망 모델 활 용법

주요 내용

- 다양한 신경망 모델 구성법
- 신경망 모델과 층의 재활용
- 신경망 모델 훈련 옵션: 콜백과 텍서보드

신경망 모델 구성법 1: Sequential 모델 활용

- Sequential 모델은 층으로 스택을 쌓아 만든 모델이며 가장 단순함
- 한 종류의 입력값과 한 종류의 출력값만 사용 가능
- 순전파: 지정된 층의 순서대로 적용

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

Sequential 모델 층 추가

```
model = keras.Sequential()  
model.add(layers.Dense(64, activation="relu"))  
model.add(layers.Dense(10, activation="softmax"))
```

신경망 모델의 가중치 확인

- 모델 선언 후 바로 가중치와 편향 확인 불가

```
>>> model.weights  
...  
ValueError: Weights for model sequential_1 have not yet been created.  
Weights are created when the Model is first called on inputs or  
'build()' is called with an 'input_shape'.
```

모델의 build() 메서드

- 모델의 빌드 메서드는 각 층별 build() 메서드를 차례대로 호출한다.

```
>>> model.build(input_shape=(None, 3))  
...  
>>> len(model.weights)  
4
```

- 층별 build() 메서드

```
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")
```

- 1층의 가중치 행렬(2차원 텐서) 모양: (3, 64)

- 입력값 특성: 3개
 - 출력값 특성: 64개

```
>>> model.weights[0].shape # 가중치 행렬  
TensorShape([3, 64])
```

- 1층의 편향 벡터(1차원 텐서) 모양: (64,)

- 출력값 특성: 64개

```
>>> model.weights[1].shape # 편향 벡터  
TensorShape([64])
```

- 2층의 가중치 행렬(2차원 텐서): (64, 10)

- 입력값 특성: 64개

- 출력값 특성: 10개

```
>>> model.weights[2].shape # 가중치 행렬  
TensorShape([64, 10])
```

- 2층의 편향 벡터(1차원 텐서) 모양: (10,)

- 출력값 특성: 10개

```
>>> model.weights[3].shape # 편향 벡터  
TensorShape([10])
```

summary() 매서드

```
>>> model.summary()
Model: "sequential_1"
-----  
Layer (type)                 Output Shape              Param #
-----  
dense_2 (Dense)              (None, 64)                256  
-----  
dense_3 (Dense)              (None, 10)                650  
-----  
Total params: 906  
Trainable params: 906  
Non-trainable params: 0  
-----
```

Input() 함수

```
model = keras.Sequential()  
model.add(keras.Input(shape=(3,)))  
model.add(layers.Dense(64, activation="relu"))
```

```
>>> model.summary()  
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
<hr/>		
dense_4 (Dense)	(None, 64)	256
<hr/>		
Total params:	256	
Trainable params:	256	
Non-trainable params:	0	
<hr/>		

층을 추가할 때마다 모델의 구조를 확인할 수 있다.

```
>>> model.add(layers.Dense(10, activation="softmax"))
>>> model.summary()
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
<hr/>		
dense_4 (Dense)	(None, 64)	256
<hr/>		
dense_5 (Dense)	(None, 10)	650
<hr/>		
Total params:	906	
Trainable params:	906	
Non-trainable params:	0	
<hr/>		

신경망 모델 구성법 2: 함수형 API

```
inputs = keras.Input(shape=(3,), name="my_input")      # 입력층
features = layers.Dense(64, activation="relu")(inputs)  # 은닉층
outputs = layers.Dense(10, activation="softmax")(features) # 출력층
model = keras.Model(inputs=inputs, outputs=outputs)      # 모델 지정
```

```
>>> model.summary()  
Model: "functional_1"
```

Layer (type)	Output Shape	Param #
<hr/>		
my_input (InputLayer)	[None, 3]	0
dense_6 (Dense)	(None, 64)	256
dense_7 (Dense)	(None, 10)	650
<hr/>		
Total params:	906	
Trainable params:	906	
Non-trainable params:	0	
<hr/>		

다중 입력, 다중 출력 모델

다중 입력과 다중 출력을 지원하는 모델을 구성하는 방법을 예제를 이용하여 설명한다.

- 입력층: 세 개
- 은닉층: 두 개
- 출력층: 두 개

```
vocabulary_size = 10000    # 사용빈도 1만등 인내 단어 사용
num_tags = 100             # 태그 수
num_departments = 4        # 부서 수

# 입력층: 세 개
title = keras.Input(shape=(vocabulary_size,), name="title")
text_body = keras.Input(shape=(vocabulary_size,), name="text_body")
tags = keras.Input(shape=(num_tags,), name="tags")

# 은닉층
features = layers.concatenate([title, text_body, tags]) # shape=(None,
10000+10000+100)
features = layers.Dense(64, activation="relu")(features)

# 출력층: 두 개
priority = layers.Dense(1, activation="sigmoid", name="priority")(features)
department = layers.Dense(
    num_departments, activation="softmax", name="department")(features)

# 모델 빌드: 입력값으로 구성된 입력값 리스트와 출력값으로 구성된 출력값 리스트
# 사용
model = keras.Model(inputs=[title, text_body, tags], outputs=[priority,
department])
```

모델 컴파일

모델 훈련

```
model.fit([title_data, text_body_data, tags_data],  
         [priority_data, department_data],  
         epochs=10)
```

모델 평가

```
model.evaluate([title_data, text_body_data, tags_data],  
              [priority_data, department_data])
```

모델 활용

- 우선 순위 예측값: 0과 1사이의 확률값

```
>>> priority_preds  
array([[0.],  
       [0.],  
       [0.],  
       ...,  
       [0.],  
       [0.],  
       [0.]], dtype=float32)
```

- 처리 부서 예측값: 각 부서별 적정도를 가리키는 확률값

```
>>> department_preds  
array([[1.267548e-05, 3.678832e-11, 2.387379e-03, 9.975999e-01],  
... [5.863077e-03, 4.932786e-09, 6.003909e-01, 3.937459e-01],  
... [1.562561e-03, 3.384366e-07, 2.208202e-02, 9.763551e-01],  
...,  
[2.978364e-03, 6.375713e-07, 4.778040e-03, 9.922429e-01],  
[2.411681e-05, 3.638920e-10, 3.098509e-01, 6.901249e-01],  
[9.115771e-05, 7.135761e-10, 7.342336e-02, 9.264854e-01]],  
dtype=float32)
```

각각의 요구사항을 처리해야 하는 부서는 `argmax()` 메서드로 확인된다.

```
>>> department_preds.argmax()  
array([3, 2, 3, ..., 3, 3, 3])
```

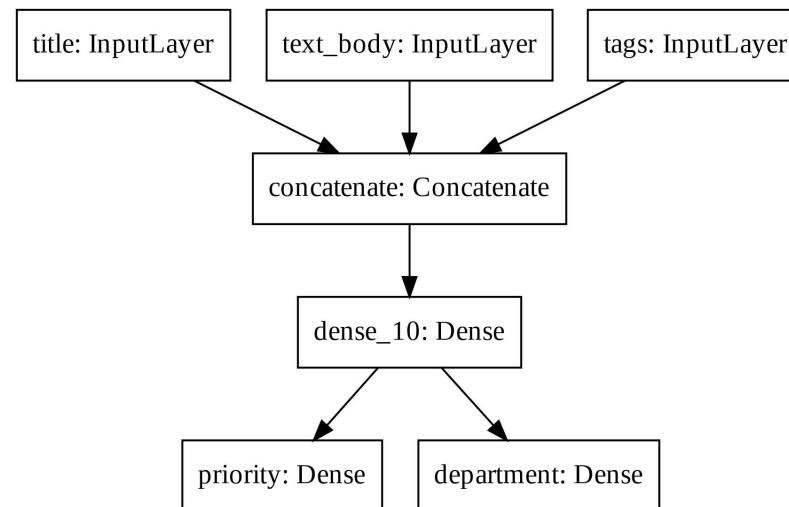
사전 객체 활용

```
model.compile(optimizer="adam",
              loss={"priority": "mean_squared_error",
                    "department": "categorical_crossentropy"},
              metrics={"priority": ["mean_absolute_error",
                                    "mean_squared_error"],
                        "department": ["accuracy", "AUC", "Precision"]})

model.fit({"title": title_data,
           "text_body": text_body_data,
           "tags": tags_data},
           {"priority": priority_data,
            "department": department_data},
           epochs=1)
```

신경망 모델 구조 그래프

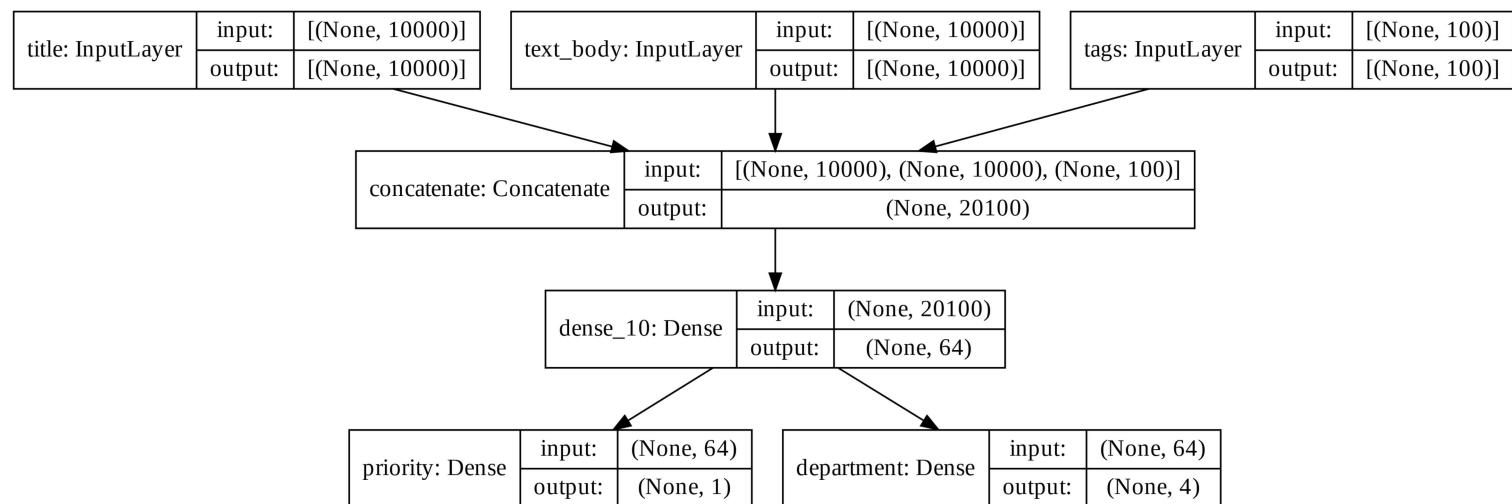
```
>>> keras.utils.plot_model(model, "ticket_classifier.png")
```



plot_model() 함수 사용 준비 사항

- pydot 모듈 설치: pip install pydot
- graphviz 프로그램 설치: <https://graphviz.gitlab.io/download/>
- 구글 코랩에서는 기본으로 지원됨.

```
>>> keras.utils.plot_model(model, "ticket_classifier_with_shape_info.png",  
show_shapes=True)
```



신경망 모델 재활용

```
>>> model.layers  
[<keras.src.engine.input_layer.InputLayer at 0x7fc3a1313fd0>,  
<keras.src.engine.input_layer.InputLayer at 0x7fc3a13ce450>,  
<keras.src.engine.input_layer.InputLayer at 0x7fc3a13c5990>,  
<keras.src.layers.merging.concatenate.Concatenate at 0x7fc3a13e0d50>,  
<keras.src.layers.core.dense.Dense at 0x7fc3a13a6310>,  
<keras.src.layers.core.dense.Dense at 0x7fc3a12f6850>,  
<keras.src.layers.core.dense.Dense at 0x7fc3a13e2f90>]
```

층별 입력값/출력값 정보

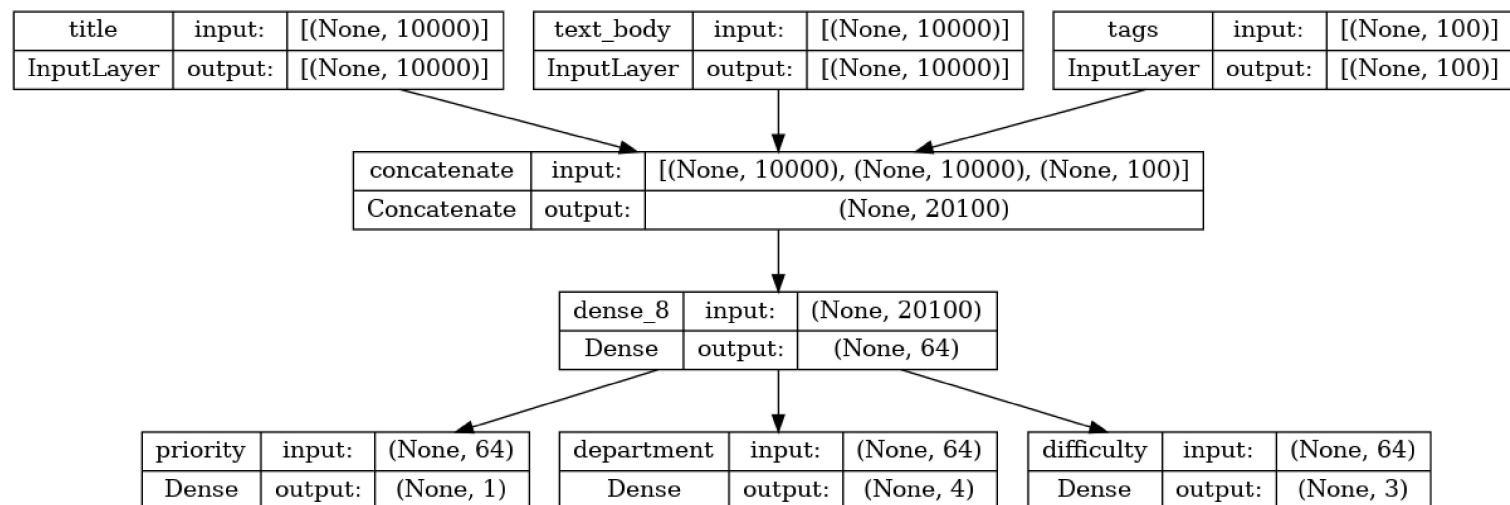
```
>>> model.layers[3].input  
<KerasTensor: shape=(None, 10000) dtype=float32 (created by layer 'title'),  
 <KerasTensor: shape=(None, 10000) dtype=float32 (created by layer  
 'text_body'),  
 <KerasTensor: shape=(None, 100) dtype=float32 (created by layer 'tags')>  
  
>>> model.layers[3].output  
<KerasTensor: shape=(None, 20100) dtype=float32 (created by layer  
 'concatenate')>
```

출력층을 제외한 층 재활용

```
features = model.layers[4].output
difficulty = layers.Dense(3, activation="softmax", name="difficulty")(features)

new_model = keras.Model(
    inputs=[title, text_body, tags],
    outputs=[priority, department, difficulty])
```

```
>>> keras.utils.plot_model(new_model, "updated_ticket_classifier.png",  
show_shapes=True)
```



신경망 모델 구성법 3: 서브클래싱

- `keras.Model` 클래스를 상속하는 모델 클래스를 직접 선언
- `__init__()` 메서드(생성자): 은닉층과 출력층으로 사용될 층 객체 지정
- `call()` 메서드: 층을 연결하는 과정 지정. 즉, 입력값으로부터 출력값을 만들어내는 순전파 과정 묘사.

예제: 고객 요구사항 처리 모델

```
class CustomerTicketModel(keras.Model):
    def __init__(self, num_departments):
        super().__init__()
        self.concat_layer = layers.concatenate()
        self.mixing_layer = layers.Dense(64, activation="relu")
        self.priority_scoring = layers.Dense(1, activation="sigmoid")
        self.department_classifier = layers.Dense(
            num_departments, activation="softmax")

    def call(self, inputs):
        title = inputs["title"]
        text_body = inputs["text_body"]
        tags = inputs["tags"]

        features = self.concat_layer([title, text_body, tags])
        features = self.mixing_layer(features)
        priority = self.priority_scoring(features)
        department = self.department_classifier(features)
        return priority, department

>>> model = CustomerTicketModel(num_departments=4)
```

서브클래싱 기법의 장단점

- 장점
 - `call()` 함수를 이용하여 층을 임의로 구성할 수 있다.
 - `for` 반복문 등 파이썬 프로그래밍 모든 기법을 적용할 수 있다.
- 단점
 - 모델 구성을 전적으로 책임져야 한다.
 - 모델 구성 정보가 `call()` 함수 외부로 노출되지 않아서 앞서 보았던 그래프 표현을 사용할 수 없다.

모델은 층의 자식 클래스

- keras.Model 이 keras.layers.Layer 의 자식 클래스
- 모델 클래스: fit(), evaluate(), predict() 메서드를 함께 지원

혼합 신경망 모델 구성법

예제: 서브클래싱 모델을 함수형 모델에 활용하기

```
class Classifier(keras.Model):

    def __init__(self, num_classes=2):
        super().__init__()
        if num_classes == 2:
            num_units = 1
            activation = "sigmoid"
        else:
            num_units = num_classes
            activation = "softmax"
        self.dense = layers.Dense(num_units, activation=activation)

    def call(self, inputs):
        return self.dense(inputs)

inputs = keras.Input(shape=(3,)) # 입력 층
features = layers.Dense(64, activation="relu")(inputs) # 은닉 층
outputs = Classifier(num_classes=10)(features) # 출력 층

model = keras.Model(inputs=inputs, outputs=outputs)
```

예제: 함수형 모델을 서브클래싱 모델에 활용하기

```
inputs = keras.Input(shape=(64,))
outputs = layers.Dense(1, activation="sigmoid")(inputs)
binary_classifier = keras.Model(inputs=inputs, outputs=outputs)

class MyModel(keras.Model):

    def __init__(self, num_classes=2):
        super().__init__()
        self.dense = layers.Dense(64, activation="relu")
        self.classifier = binary_classifier

    def call(self, inputs):
        features = self.dense(inputs)
        return self.classifier(features)
```

훈련 평가 방식 지정

훈련 과정 동안 관찰할 수 있는 내용은 일반적으로 다음과 같다.

- 에포크별 손실값
- 에포크별 평가지표

사용자 정의 평가지표

- Metric 클래스 상속
- 아래 세 개의 메서드를 재정의 overriding
 - update_state()
 - result()
 - reset_state()

예제: RootMeanSquaredError 클래스

```
class RootMeanSquaredError(keras.metrics.Metric):

    def __init__(self, name="rmse", **kwargs):
        super().__init__(name=name, **kwargs)
        self.mse_sum = self.add_weight(name="mse_sum", initializer="zeros")
        self.total_samples = self.add_weight(
            name="total_samples", initializer="zeros", dtype="int32")

    def update_state(self, y_true, y_pred, sample_weight=None):
        y_true = tf.one_hot(y_true, depth=tf.shape(y_pred)[1])
        mse = tf.reduce_sum(tf.square(y_true - y_pred))
        self.mse_sum.assign_add(mse)
        num_samples = tf.shape(y_pred)[0]
        self.total_samples.assign_add(num_samples)

    def result(self):
        return tf.sqrt(self.mse_sum / tf.cast(self.total_samples, tf.float32))

    def reset_state(self):
        self.mse_sum.assign(0.)
        self.total_samples.assign(0)
```

생성자

- Metric 클래스 상속, 평가지표 이름 지정, 에포크 별로 계산될 평가지표 계산에 필요한 속성(변수) 초기화
- name="rmse" : 평가지표 이름. 훈련중에 평가지표 구분에 사용됨.
- mse_sum : (에포크 별) 누적 제곱 오차. 0으로 초기화.
- total_samples : (에포크 별) 훈련에 사용된 총 데이터 샘플 수. 0으로 초기화.

```
def __init__(self, name="rmse", **kwargs):  
    super().__init__(name=name, **kwargs)  
    self.mse_sum = self.add_weight(name="mse_sum", initializer="zeros")  
    self.total_samples = self.add_weight(  
        name="total_samples", initializer="zeros", dtype="int32")
```

에포크 상태 업데이트

- `update_state`: 에포크 내에서 스텝 단위로 지정된 속성 업데이트
- `mse`: 입력 배치 단위로 계산된 모든 샘플들에 대한 예측 오차의 제곱의 합.
- `mse_sum` 업데이트: 새롭게 계산된 `mse`를 기준 `mse_sum`에 더함.
- `num_samples`: 배치 크기
- `total_samples` 업데이트: 새로 훈련된 배치 크기를 기준 `total_samples`에 더함

```
def update_state(self, y_true, y_pred, sample_weight=None):  
    ... y_true = tf.one_hot(y_true, depth=tf.shape(y_pred)[1])  
    ... mse = tf.reduce_sum(tf.square(y_true - y_pred))  
    ... self.mse_sum.assign_add(mse)  
    ... num_samples = tf.shape(y_pred)[0]  
    ... self.total_samples.assign_add(num_samples)
```

에포크 단위 평가지표 계산

- `result` : 에포크 별로 평가지표 계산
- 여기서는 에포크 별로 평균 제곱근 오차 계산.
- `mse_sum` 을 `total_samples` 으로 나눈 값의 제곱근 계산

```
def result(self):  
    ... return tf.sqrt(self.mse_sum / tf.cast(self.total_samples, tf.float32))
```

에포크 상태 초기화

- `reset_state`: 새로운 에포크 훈련이 시작될 때 모든 인스턴스 속성(변수) 초기화
- 여기서는 `mse_sum`과 `total_samples` 모두 0으로 초기화

```
def reset_state(self):  
    self.mse_sum.assign(0.)  
    self.total_samples.assign(0)
```

사용자 정의 평가지표 활용

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

```
>>> model.fit(train_images, train_labels,  
...             epochs=3,  
...             validation_data=(val_images, val_labels))
```

```
Epoch 1/3  
1563/1563 [=====] - 12s 7ms/step - loss: 0.2929 - accuracy: 0.9136 - rmse: 7.1776  
- val_loss: 0.1463 - val_accuracy: 0.9576 - val_rmse: 7.3533  
Epoch 2/3  
1563/1563 [=====] - 11s 7ms/step - loss: 0.1585 - accuracy: 0.9546 - rmse: 7.3526  
- val_loss: 0.1215 - val_accuracy: 0.9650 - val_rmse: 7.3954  
Epoch 3/3  
1563/1563 [=====] - 11s 7ms/step - loss: 0.1293 - accuracy: 0.9636 - rmse: 7.3837  
- val_loss: 0.1058 - val_accuracy: 0.9711 - val_rmse: 7.4182  
313/313 [=====] - 2s 5ms/step - loss: 0.1003 - accuracy: 0.9731 - rmse: 7.4307
```

콜백

컴퓨터 프로그래밍에서 **콜백** callback은 하나의 프로그램이 실행되는 도중에 추가적으로 다른 API를 호출하는 기능 또는 해당 API를 가리킨다. 호출된 콜백은 자신을 호출한 프로그램과 독립적으로 실행된다.

가장 많이 활용되는 콜백은 다음과 같다.

- 훈련 기록 작성
 - 훈련 에포크마다 보여지는 손실값, 평가지표 등 관리
 - `keras.callbacks.CSVLogger` 클래스 활용.
- 훈련중인 모델의 상태 저장
 - 훈련 중 가장 좋은 성능의 모델(의 상태) 저장
 - `keras.callbacks.ModelCheckpoint` 클래스 활용
- 훈련 조기 종료
 - 검증셋에 대한 손실이 더 이상 개선되지 않는 경우 훈련을 종료 시키기
 - `keras.callbacks.EarlyStopping` 클래스 활용
- 하이퍼 파라미터 조정
 - 학습률 동적 변경 지원
 - `keras.callbacks.LearningRateScheduler` 또는
`keras.callbacks.ReduceLROnPlateau` 클래스 활용

예제

콜백은 `fit()` 함수의 `callbacks`라는 옵션 인자를 이용하여 지정한다. 예를 들어 아래 코드는 두 종류의 콜백을 지정한다.

- `EarlyStopping`: 검증셋에 대한 정확도가 2 에포크 연속 개선되지 않을 때 훈련 종료
- `ModelCheckpoint`: 매 에포크마다 훈련된 모델 상태 저장
 - `save_best_only=True`: 검증셋에 대한 손실값이 가장 낮은 모델만 저장

```
callbacks_list = [
    keras.callbacks.EarlyStopping(
        monitor="val_accuracy",
        patience=2,
    ),
    keras.callbacks.ModelCheckpoint(
        filepath="checkpoint_path",
        monitor="val_loss",
        save_best_only=True,
    )
]

model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels,
          epochs=10,
          callbacks=callbacks_list,
          validation_data=(val_images, val_labels))
```

사용자 정의 콜백

케라스와 호환되는 콜백 클래스를 정의하려면 `keras.callbacks.Callback` 클래스를 상속하면 된다. 매 에포크와 매 배치 훈련 단계의 시작과 종료 지점에서 수행해야 할 기능을 정의해야 하며 아래 메서드를 재정의하는 방식으로 이루어진다.

```
on_epoch_begin(epoch, logs)
on_epoch_end(epoch, logs)
on_batch_begin(batch, logs)
on_batch_end(batch, logs)
on_train_begin(logs)
on_train_end(logs)
```

각 메서드에 사용되는 인자는 훈련 과정 중에 자동으로 생성된 객체로부터 값을 받아온다.

- `logs` 인자: 이전 배치와 에포크의 훈련셋과 검증셋에 대한 손실값, 평가지표 등을 포함한 사전 객체.
- `batch`와 `epoch`: 배치와 에포크 정보

다음 `LossHistory` 콜백 클래스는 배치 훈련이 끝날 때마다 손실값을 저장하고 에포크가 끝날 때마다 배치별 손실값을 그래프로 저장하여 훈련이 종료된 후 시각화하여 보여주도록 한다.

```
from matplotlib import pyplot as plt

class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs):
        self.per_batch_losses = []

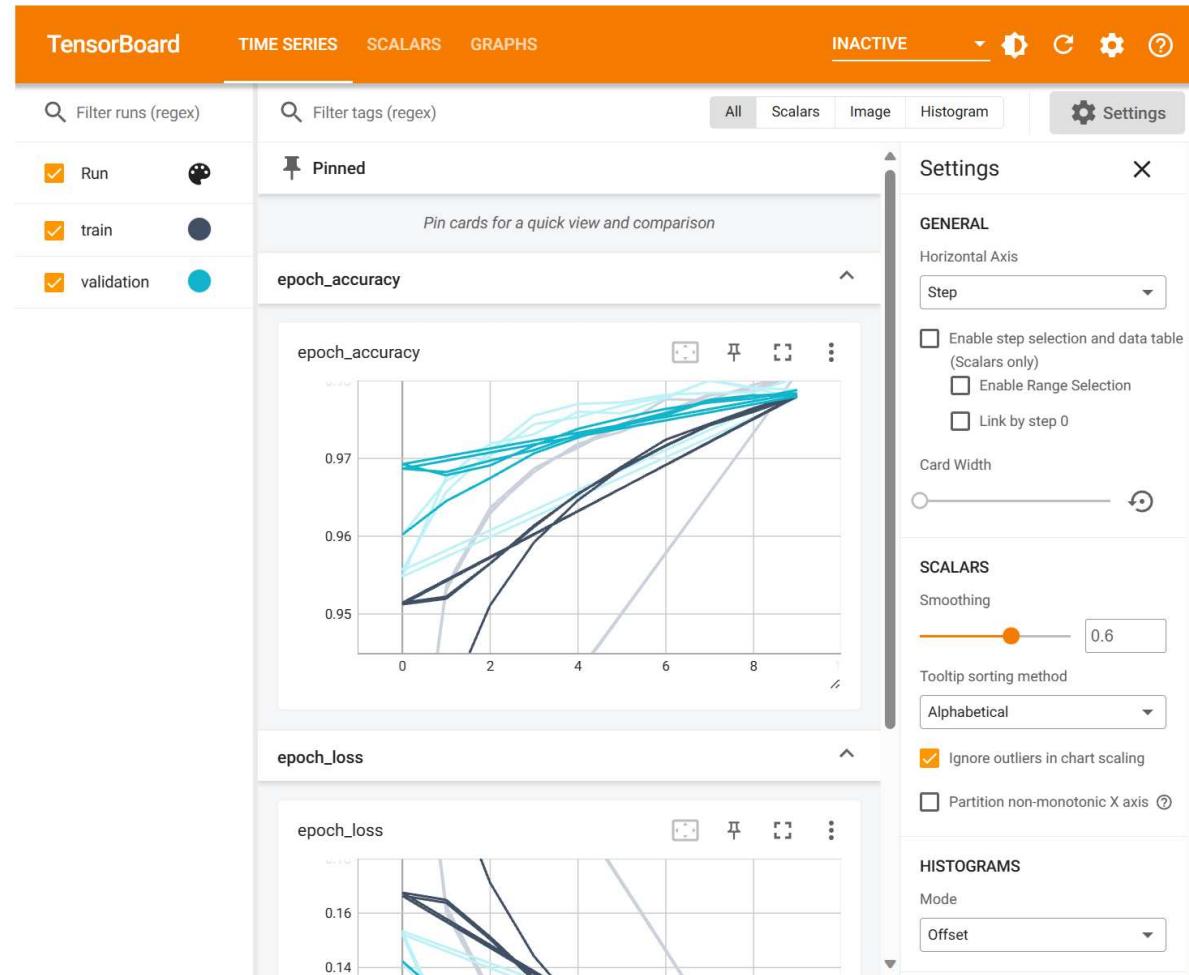
    def on_batch_end(self, batch, logs):
        self.per_batch_losses.append(logs.get("loss"))

    def on_epoch_end(self, epoch, logs):
        plt.clf()
        plt.plot(range(len(self.per_batch_losses)), self.per_batch_losses,
                  label="Training loss for each batch")
        plt.xlabel(f"Batch (epoch {epoch})")
        plt.ylabel("Loss")
        plt.legend()
        plt.savefig(f"plot_at_epoch_{epoch}")
        self.per_batch_losses = []
```

텐서보드

텐서보드TensorBoard는 모델 훈련 과정을 시각적으로 실시간 관찰할 수 있는 도구를 제공하며 텐서플로우와 함께 설치된다.

- 신경망 모델 구조 시각화
- 손실값, 정확도 등의 변화 시각화
- 가중치, 편향 텐서 등의 변화 히스토그램
- 이미지, 텍스트, 오디오 데이터 시각화
- 기타 다양한 기능 제공



텐서보드는 `TensorBoard` 콜백 클래스를 활용한다.

- `log_dir`: 텐서보드 서버 실행에 필요한 데이터 저장소 지정

```
tensorboard = keras.callbacks.TensorBoard(  
    ... log_dir='./tensorboard_log_dir',  
    ...)  
  
model.fit(train_images, train_labels,  
    ... epochs=10,  
    ... validation_data=(val_images, val_labels),  
    ... callbacks=[tensorboard])
```

- 주피터 노트북에서

```
%load_ext tensorboard  
%tensorboard --logdir ./tensorboard_log_dir
```

- 터미널에서

```
tensorboard --logdir ./tensorboard_log_dir
```