In [1]:

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

<간단한 모델 만들기>

Sequential 모델 : Keras에서는 층(layer)을 조합하여 모델을 만든다. 모델은 층의 그래프이다. 가장 흔한 모델 구조는 층을 차례대로 쌓은 tf.keras.Sequential 모델이다.

완전 연결(fully connected) 네트워크(= 다층 퍼셉트론(multi layer perceptron)) 만들기

In [15]:

```
from tensorflow.keras import layers

model = tf.keras.Sequential()
# 64개의 유닛을 가진 완전 연결 층을 모델에 추가합니다:
model.add(layers.Dense(64, activation='relu'))
# 또 하나를 추가합니다:
model.add(layers.Dense(64, activation='relu'))
# 10개의 출력 유닛을 가진 소프트맥스 층을 추가합니다:
model.add(layers.Dense(10, activation='softmax'))
```

층 설정(tf.keras.layers) 아래의 클래스들은 일부 생성자 매개변수를 공통으로 가짐.

-activation : 층의 활성화 함수를 설정. 이 매개변수에서는 기본으로 제공되는 함수의 이름을 쓰거나 호출 가능한 객체를 지정할 수 있다. 기본값은 활성화 함수를 적용하지 않은 것이다.

-kernel_initializer & bias_initializer : 층의 가중치(weight) 커널(kernel)과 절편(bias)를 초기화하는 방법이다. 내장 함수나 호출 가능한 객체를 지정한다. 기본값은 "glorot_uniform" 초기화이다.

-kernel_regularizer & bias_regularizer : L1 또는 L2 규제(regularization)와 같이 층의 가중치(커널과 절편)에 적용할 규제 방법을 지정한다. 기본값은 규제를 적용하지 않은 것이다.

In [16]:

```
# 시그모이드 활성화 층을 만듭니다:
layers.Dense(64, activation='sigmoid')
# 또는 다음도 가능합니다:
# layers.Dense(64, activation=tf.keras.activations.sigmoid)

# 커널 행렬에 L1 규제가 적용된 선형 활성화 층. 하이퍼파라미터 0.01은 규제의 양을 조절합니다:
layers.Dense(64, kernel_regularizer=tf.keras.regularizers.l1(0.01))

# 절편 벡터에 L2 규제가 적용된 선형 활성화 층. 하이퍼파라미터 0.01은 규제의 양을 조절합니다:
layers.Dense(64, bias_regularizer=tf.keras.regularizers.l2(0.01))

# 커널을 랜덤한 직교 행렬로 초기화한 선형 활성화 층:
layers.Dense(64, kernel_initializer='orthogonal')

# 절편 벡터를 상수 2.0으로 설정한 선형 활성화 층:
layers.Dense(64, bias_initializer=tf.keras.initializers.Constant(2.0))
```

Out[16]:

```
<tensorflow.python.keras.layers.core.Dense at 0x160441e3688>
```

훈련과 평가

훈련 준비 : 모델을 구성한 후 compile 메서드를 호출하여 학습 과정을 설정한다.

In [17]:

```
model = tf.keras.Sequential([
```

```
model = tf.keras.Sequential([
# 64개의 유닛을 가진 완전 연결 층을 모델에 추가합니다:
layers.Dense(64, activation='relu', input_shape=(32,)),
# 또 하나를 추가합니다:
layers.Dense(64, activation='relu'),
# 10개의 출력 유닛을 가진 소프트맥스 층을 추가합니다:
layers.Dense(10, activation='softmax')])

model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

tf.keras.Model.compile에는 3개의 중요한 매개변수가 있다.

-optimizer : 훈련 과정을 설정한다. tf.keras.optimizers.Adam이나 tf.keras.optimizers.SGD와 같은 tf.keras.optimizers 아래의 옵티마이저 객체를 전달한다. 기본 매개변수를 사용할 경우 'adam'이나 'sgd'와 같이 문자열로 지정할 수도 있다.

-loss : 최적화 과정에서 최소화될 손실 함수(loss function)를 설정한다. 평균 제곱 오차(mse)와 categorical_crossentropy, binary_crossentropy 등이 자주 사용된다. 손실 함수의 이름을 지정하거나 tf.keras.losses 모듈 아래의 호출 가능한 객체를 전달할 수 있다.

-metrics : 훈련을 모니터링하기 위해 사용된다. 이름이나 tf.keras.metrics 모듈 아래의 호출 가능한 객체이다.

추가적으로 모델의 훈련과 평가를 즉시 실행하려면 run_eagerly=True 매개변수를 전달할 수 있다.

In [18]:

```
# 평균 제곱 오차로 회귀 모델을 설정합니다.
model.compile(optimizer=tf.keras.optimizers.Adam(0.01),
              loss='mse',       # 평균 제곱 오차
              metrics=['mae'])  # 평균 절댓값 오차

# 크로스엔트로피 손실 함수로 분류 모델을 설정합니다.
model.compile(optimizer=tf.keras.optimizers.RMSprop(0.01),
              loss=tf.keras.losses.CategoricalCrossentropy(),
              metrics=[tf.keras.metrics.CategoricalAccuracy()])
```

넘파이 데이터를 사용한 훈련

데이터셋이 작은 경우 넘파이(NumPy) 배열을 메모리에 적재하여 모델을 훈련하고 평가합니다. 모델은 fit 메서드를 통해서 훈련 데이터를 학습한다.

In [19]:

```
import numpy as np

data = np.random.random((1000, 32))
labels = np.random.random((1000, 10))

model.fit(data, labels, epochs=10, batch_size=32)
```

```
Train on 1000 samples
Epoch 1/10
1000/1000 [==============================] - 0s 383us/sample - loss: 275.0390 -
categorical_accuracy: 0.0820
Epoch 2/10
1000/1000 [==============================] - 0s 48us/sample - loss: 1129.4560 -
categorical_accuracy: 0.0910
Epoch 3/10
1000/1000 [==============================] - 0s 48us/sample - loss: 2380.0999 -
categorical_accuracy: 0.0930
Epoch 4/10
1000/1000 [==============================] - 0s 48us/sample - loss: 3797.7487 -
categorical_accuracy: 0.0980
Epoch 5/10
1000/1000 [==============================] - 0s 48us/sample - loss: 5581.8812 -
categorical_accuracy: 0.1000
Epoch 6/10
1000/1000 [==============================] - 0s 48us/sample - loss: 7845.6561 -
categorical_accuracy: 0.0840
Epoch 7/10
```

```
1000/1000 [==============================] - 0s 52us/sample - loss: 10098.8897 -
categorical_accuracy: 0.1140
Epoch 8/10
1000/1000 [==============================] - 0s 48us/sample - loss: 12846.1974 -
categorical_accuracy: 0.0830
Epoch 9/10
1000/1000 [==============================] - 0s 52us/sample - loss: 16169.0040 -
categorical_accuracy: 0.0980
Epoch 10/10
1000/1000 [==============================] - 0s 48us/sample - loss: 18984.0344 -
categorical_accuracy: 0.0790
```

Out[19]:

```
<tensorflow.python.keras.callbacks.History at 0x160442e4ec8>
```

tf.keras.Model.fit에는 세 개의 중요한 매개변수가 있습니다:

- epochs: 훈련은 에포크(epoch)로 구성됩니다. 한 에포크는 전체 입력 데이터를 한번 순회하는 것입니다(작은 배치로 나누어 수행됩니다).
- batch_size: 넘파이 데이터를 전달하면 모델은 데이터를 작은 배치로 나누고 훈련 과정에서 이 배치를 순회합니다. 이 정수 값은 배치의 크기를 지정합니다. 전체 샘플 개수가 배치 크기로 나누어 떨어지지 않으면 마지막 배치의 크기는 더 작을 수 있습니다.
- validation_data: 모델의 프로토타입(prototype)을 만들 때는 검증 데이터(validation data)에서 간편하게 성능을 모니터링해야 합니다. 입력과 레이블(label)의 튜플을 이 매개변수로 전달하면 에포크가 끝날 때마다 추론 모드(inference mode)에서 전달된 데이터의 손실과 측정 지표를 출력합니다.

다음이 validation_data를 사용하는 예입니다:

In [20]:

```
import numpy as np

data = np.random.random((1000, 32))
labels = np.random.random((1000, 10))

val_data = np.random.random((100, 32))
val_labels = np.random.random((100, 10))

model.fit(data, labels, epochs=10, batch_size=32,
          validation_data=(val_data, val_labels))
```

```
Train on 1000 samples, validate on 100 samples
Epoch 1/10
1000/1000 [==============================] - 0s 160us/sample - loss: 22850.9523 -
categorical_accuracy: 0.1070 - val_loss: 29911.3282 - val_categorical_accuracy: 0.1400
Epoch 2/10
1000/1000 [==============================] - 0s 56us/sample - loss: 27704.0324 -
categorical_accuracy: 0.0960 - val_loss: 25121.8881 - val_categorical_accuracy: 0.0500
Epoch 3/10
1000/1000 [==============================] - 0s 72us/sample - loss: 31393.4821 -
categorical_accuracy: 0.1070 - val_loss: 25921.7847 - val_categorical_accuracy: 0.1000
Epoch 4/10
1000/1000 [==============================] - 0s 68us/sample - loss: 34711.0805 -
categorical_accuracy: 0.1030 - val_loss: 46008.9078 - val_categorical_accuracy: 0.1000
Epoch 5/10
1000/1000 [==============================] - 0s 72us/sample - loss: 40819.9130 -
categorical_accuracy: 0.0860 - val_loss: 59189.6200 - val_categorical_accuracy: 0.0500
Epoch 6/10
1000/1000 [==============================] - 0s 56us/sample - loss: 45233.7463 -
categorical_accuracy: 0.1190 - val_loss: 58472.7950 - val_categorical_accuracy: 0.1100
Epoch 7/10
1000/1000 [==============================] - 0s 56us/sample - loss: 53175.3352 -
categorical_accuracy: 0.0900 - val_loss: 60300.5644 - val_categorical_accuracy: 0.0700
Epoch 8/10
1000/1000 [==============================] - 0s 60us/sample - loss: 56646.0995 -
categorical_accuracy: 0.1070 - val_loss: 54633.3570 - val_categorical_accuracy: 0.1000
Epoch 9/10
1000/1000 [==============================] - 0s 56us/sample - loss: 64232.5261 -
categorical_accuracy: 0.1030 - val_loss: 72994.1200 - val_categorical_accuracy: 0.1100
Epoch 10/10
1000/1000 [==============================] - 0s 48us/sample - loss: 70362.6090 -
categorical_accuracy: 0.0940 - val_loss: 62957.1834 - val_categorical_accuracy: 0.1000
```

Out[20]:

```
<tensorflow.python.keras.callbacks.History at 0x160457b2888>
```

tf.data 데이터셋을 사용한 훈련

데이터셋 API를 사용하여 대규모 데이터셋이나 복수의 장치로 확장시킬 수 있습니다. fit 메서드에 tf.data.Dataset 객체를 전달합니다:

In [21]:

```
# 예제 `Dataset` 객체를 만듭니다:
dataset = tf.data.Dataset.from_tensor_slices((data, labels))
dataset = dataset.batch(32)

# Dataset에서 `fit` 메서드를 호출할 때 `steps_per_epoch` 설정을 잊지 마세요.
model.fit(dataset, epochs=10, steps_per_epoch=30)
```

```
Train for 30 steps
Epoch 1/10
30/30 [==============================] - 0s 11ms/step - loss: 74126.9893 - categorical_accuracy: 0
.1052
Epoch 2/10
 1/30 [>.............................] - ETA: 0s - loss: 114534.8906 - categorical_accuracy: 0.187
5WARNING:tensorflow:Your input ran out of data; interrupting training. Make sure that your dataset
or generator can generate at least `steps_per_epoch * epochs` batches (in this case, 300 batches).
You may need to use the repeat() function when building your dataset.
```

Out[21]:

```
<tensorflow.python.keras.callbacks.History at 0x160442eacc8>
```

여기에서 fit 메서드는 steps_per_epoch 매개변수를 사용합니다. 다음 에포크로 넘어가기 전에 모델이 수행할 훈련 단계 횟수입니다. Dataset이 배치 데이터를 생성하기 때문에 batch_size가 필요하지 않습니다.

Dataset은 검증 데이터에도 사용할 수 있습니다:

In [22]:

```
dataset = tf.data.Dataset.from_tensor_slices((data, labels))
dataset = dataset.batch(32)

val_dataset = tf.data.Dataset.from_tensor_slices((val_data, val_labels))
val_dataset = val_dataset.batch(32)

model.fit(dataset, epochs=10,
          validation_data=val_dataset)
```

```
Train for 32 steps, validate for 4 steps
Epoch 1/10
32/32 [==============================] - 0s 4ms/step - loss: 83140.9224 - categorical_accuracy: 0.
0890 - val_loss: 68785.7344 - val_categorical_accuracy: 0.1200
Epoch 2/10
32/32 [==============================] - 0s 2ms/step - loss: 94032.5439 - categorical_accuracy: 0.
1000 - val_loss: 90046.4570 - val_categorical_accuracy: 0.1200
Epoch 3/10
32/32 [==============================] - 0s 2ms/step - loss: 99205.5904 - categorical_accuracy: 0.
0970 - val_loss: 99886.6309 - val_categorical_accuracy: 0.0700
Epoch 4/10
32/32 [==============================] - 0s 2ms/step - loss: 107401.4375 - categorical_accuracy: 0
.0990 - val_loss: 122230.1641 - val_categorical_accuracy: 0.1100
Epoch 5/10
32/32 [==============================] - 0s 2ms/step - loss: 116973.7113 - categorical_accuracy: 0
.1170 - val_loss: 167639.2461 - val_categorical_accuracy: 0.1700
Epoch 6/10
32/32 [==============================] - 0s 2ms/step - loss: 126541.9594 - categorical_accuracy: 0
.1050 - val_loss: 91253.3164 - val_categorical_accuracy: 0.1400
Epoch 7/10
32/32 [==============================] - 0s 2ms/step - loss: 127753.4159 - categorical_accuracy: 0
.1080 - val_loss: 129382.1973 - val_categorical_accuracy: 0.1200
Epoch 8/10
```

```
32/32 [==============================] - 0s 2ms/step - loss: 140288.9684 - categorical_accuracy: 0
.0850 - val_loss: 144269.6133 - val_categorical_accuracy: 0.1200
Epoch 9/10
32/32 [==============================] - 0s 2ms/step - loss: 156637.8523 - categorical_accuracy: 0
.0690 - val_loss: 169352.1992 - val_categorical_accuracy: 0.1200
Epoch 10/10
32/32 [==============================] - 0s 2ms/step - loss: 160274.1964 - categorical_accuracy: 0
.1020 - val_loss: 217883.3242 - val_categorical_accuracy: 0.0700
```

Out[22]:

```
<tensorflow.python.keras.callbacks.History at 0x16046baf9c8>
```

평가와 예측

tf.keras.Model.evaluate와 tf.keras.Model.predict 메서드에는 넘파이 배열이나 tf.data.Dataset을 사용할 수 있습니다.

주어진 데이터로 추론 모드의 손실이나 지표를 평가합니다:

In [23]:

```
data = np.random.random((1000, 32))
labels = np.random.random((1000, 10))

model.evaluate(data, labels, batch_size=32)

model.evaluate(dataset, steps=30)
```

```
1000/1000 [==============================] - 0s 32us/sample - loss: 223190.4403 -
categorical_accuracy: 0.0960
30/30 [==============================] - 0s 1ms/step - loss: 225024.1474 - categorical_accuracy: 0
.0896
```

Out[23]:

```
[225024.14739583334, 0.08958333]
```

주어진 데이터로 추론 모드에서 마지막 층의 출력을 예측하여 넘파이 배열로 반환합니다:

In [24]:

```
result = model.predict(data, batch_size=32)
print(result.shape)
```

```
(1000, 10)
```

고급 모델 만들기

함수형 API tf.keras.Sequential 모델은 단순히 층을 쌓은 것으로 임의의 구조를 표현할 수 없습니다. 케라스 함수형 API를 사용하면 다음과 같은 복잡한 모델 구조를 만들 수 있습니다:

- 다중 입력 모델
- 다중 출력 모델
- 층을 공유하는 모델 (동일한 층을 여러번 호출합니다),
- 데이터 흐름이 차례대로 진행되지 않는 모델 (예를 들면 잔차 연결(residual connections)).

함수형 API로 모델을 만드는 방식은 다음과 같습니다:

1. 하나의 층 객체는 호출 가능하고 텐서를 반환합니다.
2. tf.keras.Model 객체를 정의하기 위해 입력 텐서와 출력 텐서를 사용합니다.
3. 이 모델은 Sequential 모델과 동일한 방식으로 훈련됩니다.

다음 코드는 함수형 API를 사용하여 간단한 완전 연결 네트워크를 만드는 예입니다:

In [25]:

```
inputs = tf.keras.Input(shape=(32,))  # 입력 플레이스홀더를 반환합니다.

# 층 객체는 텐서를 사용하여 호출되고 텐서를 반환합니다.
x = layers.Dense(64, activation='relu')(inputs)
x = layers.Dense(64, activation='relu')(x)
predictions = layers.Dense(10, activation='softmax')(x)
```

입력과 출력을 사용해 모델의 객체를 만듭니다.

In [26]:

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

# 컴파일 단계는 훈련 과정을 설정합니다.
model.compile(optimizer=tf.keras.optimizers.RMSprop(0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# 5번의 에포크 동안 훈련합니다.
model.fit(data, labels, batch_size=32, epochs=5)
```

```
Train on 1000 samples
Epoch 1/5
1000/1000 [==============================] - 0s 368us/sample - loss: 13.5369 - accuracy: 0.1000
Epoch 2/5
1000/1000 [==============================] - 0s 44us/sample - loss: 22.2957 - accuracy: 0.1010
Epoch 3/5
1000/1000 [==============================] - 0s 48us/sample - loss: 36.9584 - accuracy: 0.0980
Epoch 4/5
1000/1000 [==============================] - 0s 48us/sample - loss: 55.8587 - accuracy: 0.0940
Epoch 5/5
1000/1000 [==============================] - 0s 44us/sample - loss: 79.4541 - accuracy: 0.0950
```

Out[26]:

```
<tensorflow.python.keras.callbacks.History at 0x16046dabec8>
```

모델 클래스 상속

tf.keras.Model 클래스를 상속하고 자신만의 정방향 패스(forward pass)을 정의하여 완전히 커스터마이징된 모델을 만들 수 있습니다. **init** 메서드에서 층을 만들어 클래스 객체의 속성으로 지정합니다. 정방향 패스는 call 메서드에 정의합니다.

즉시 실행이 활성화되어 있을 때 정방향 패스를 명령형 프로그래밍 방식으로 작성할 수 있기 때문에 모델 클래스 상속이 매우 유용합니다.

노트: 정방향 패스를 항상 명령형 프로그래밍 방식으로 실행하려면 super 객체의 생성자를 호출할 때 dynamic=True를 지정하세요.

중요 포인트: 작업에 맞는 API를 사용하세요. 모델 클래스 상속은 유연성을 제공하지만 복잡도가 증가하고 사용자 오류가 발생할 가능성이 높아집니다. 가능한한 함수형 API를 사용하세요.

다음 코드는 tf.keras.Model의 클래스를 상속하여 명령형 프로그래밍 방식으로 실행할 필요가 없는 정방향 패스를 구현한 예입니다:

In [27]:

```
class MyModel(tf.keras.Model):

  def __init__(self, num_classes=10):
    super(MyModel, self).__init__(name='my_model')
    self.num_classes = num_classes
    # 층을 정의합니다.
    self.dense_1 = layers.Dense(32, activation='relu')
    self.dense_2 = layers.Dense(num_classes, activation='sigmoid')

  def call(self, inputs):
    # 정방향 패스를 정의합니다.
    # `__init__` 메서드에서 정의한 층을 사용합니다.
    x = self.dense_1(inputs)
    return self.dense_2(x)
```

새 모델 클래스의 객체를 만듭니다:

In [28]:

```
model = MyModel(num_classes=10)

# 컴파일 단계는 훈련 과정을 설정합니다.
model.compile(optimizer=tf.keras.optimizers.RMSprop(0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# 5번의 에포크 동안 훈련합니다.
model.fit(data, labels, batch_size=32, epochs=5)
```

```
Train on 1000 samples
Epoch 1/5
1000/1000 [==============================] - 0s 328us/sample - loss: 11.5239 - accuracy: 0.0930
Epoch 2/5
1000/1000 [==============================] - 0s 48us/sample - loss: 11.4994 - accuracy: 0.1000
Epoch 3/5
1000/1000 [==============================] - 0s 48us/sample - loss: 11.4934 - accuracy: 0.0970
Epoch 4/5
1000/1000 [==============================] - 0s 48us/sample - loss: 11.4897 - accuracy: 0.1020
Epoch 5/5
1000/1000 [==============================] - 0s 48us/sample - loss: 11.4867 - accuracy: 0.0970
```

Out[28]:

```
<tensorflow.python.keras.callbacks.History at 0x160471e9e88>
```

맞춤형 층

맞춤형 층(custom layer)을 만들려면 tf.keras.layers.Layer 클래스를 상속하고 다음 메서드를 구현합니다:

- **init**: 이 층에서 사용되는 하위 층을 정의할 수 있습니다.
- build: 층의 가중치를 만듭니다. add_weight 메서드를 사용해 가중치를 추가합니다.
- call: 정방향 패스를 구현합니다.

다음 코드는 입력과 커널 행렬의 matmul 계산을 구현한 맞춤형 층의 예입니다:

In [29]:

```
class MyLayer(layers.Layer):

  def __init__(self, output_dim, **kwargs):
    self.output_dim = output_dim
    super(MyLayer, self).__init__(**kwargs)

  def build(self, input_shape):
    # 이 층에서 훈련할 가중치 변수를 만듭니다.
    self.kernel = self.add_weight(name='kernel',
                                  shape=(input_shape[1], self.output_dim),
                                  initializer='uniform',
                                  trainable=True)

  def call(self, inputs):
    return tf.matmul(inputs, self.kernel)

  def get_config(self):
    base_config = super(MyLayer, self).get_config()
    base_config['output_dim'] = self.output_dim
    return base_config

  @classmethod
  def from_config(cls, config):
    return cls(**config)
```

맞춤형 층을 사용하여 모델을 만듭니다:

In [44]:

```
model = tf.keras.Sequential([
    MyLayer(10),
    layers.Activation('softmax')])

# 컴파일 단계는 훈련 과정을 설정합니다.
model.compile(optimizer=tf.keras.optimizers.RMSprop(0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# 5번의 에포크 동안 훈련합니다.
model.fit(data, labels, batch_size=32, epochs=5)
```

```
Train on 1000 samples
Epoch 1/5
1000/1000 [==============================] - 0s 224us/sample - loss: 11.5055 - accuracy: 0.0950
Epoch 2/5
1000/1000 [==============================] - 0s 36us/sample - loss: 11.5054 - accuracy: 0.1010
Epoch 3/5
1000/1000 [==============================] - 0s 44us/sample - loss: 11.5052 - accuracy: 0.0970
Epoch 4/5
1000/1000 [==============================] - 0s 44us/sample - loss: 11.5049 - accuracy: 0.1000
Epoch 5/5
1000/1000 [==============================] - 0s 36us/sample - loss: 11.5046 - accuracy: 0.0940
```

Out[44]:

```
<tensorflow.python.keras.callbacks.History at 0x1604a39c288>
```

콜백

콜백(callback)은 훈련하는 동안 모델의 동작을 변경하고 확장하기 위해 전달하는 객체입니다. 자신만의 콜백을 작성하거나 다음과 같은 내장 tf.keras.callbacks을 사용할 수 있습니다:

- tf.keras.callbacks.ModelCheckpoint: 일정 간격으로 모델의 체크포인트를 저장합니다.
- tf.keras.callbacks.LearningRateScheduler: 학습률(learning rate)을 동적으로 변경합니다.
- tf.keras.callbacks.EarlyStopping: 검증 성능이 향상되지 않으면 훈련을 중지합니다.
- tf.keras.callbacks.TensorBoard: 텐서보드를 사용하여 모델을 모니터링합니다.

tf.keras.callbacks.Callback을 사용하려면 모델의 fit 메서드에 전달합니다:

In [49]:

```
callbacks = [
  # `val_loss`가 2번의 에포크에 걸쳐 향상되지 않으면 훈련을 멈춥니다.
  tf.keras.callbacks.EarlyStopping(patience=2, monitor='val_loss'),
  # `./logs` 디렉토리에 텐서보드 로그를 기록니다.
  tf.keras.callbacks.TensorBoard(log_dir='./logs')
]

model.fit(data, labels, batch_size=32, epochs=5,
          #callbacks=callbacks
          validation_data=(val_data, val_labels))
```

```
Train on 1000 samples, validate on 100 samples
Epoch 1/5
1000/1000 [==============================] - 0s 128us/sample - loss: 11.5050 - accuracy: 0.0960 -
val_loss: 11.3210 - val_accuracy: 0.1000
Epoch 2/5
1000/1000 [==============================] - 0s 48us/sample - loss: 11.5044 - accuracy: 0.0960 - v
al_loss: 11.3212 - val_accuracy: 0.1000
Epoch 3/5
1000/1000 [==============================] - 0s 56us/sample - loss: 11.5044 - accuracy: 0.0950 - v
al_loss: 11.3204 - val_accuracy: 0.1000
Epoch 4/5
1000/1000 [==============================] - 0s 44us/sample - loss: 11.5045 - accuracy: 0.0960 - v
al_loss: 11.3198 - val_accuracy: 0.1000
Epoch 5/5
1000/1000 [==============================] - 0s 48us/sample - loss: 11.5039 - accuracy: 0.0930 - v
al_loss: 11.3210 - val_accuracy: 0.1000
```

Out[49]:

```
<tensorflow.python.keras.callbacks.History at 0x16048fb5d88>
```

저장과 복원

가중치 tf.keras.Model.save_weights를 사용하여 모델의 가중치를 저장하고 복원합니다.

In [32]:

```
model = tf.keras.Sequential([
layers.Dense(64, activation='relu', input_shape=(32,)),
layers.Dense(10, activation='softmax')])

model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

In [33]:

```
# 가중치를 텐서플로의 체크포인트 파일로 저장합니다.
model.save_weights('./weights/my_model')

# 모델의 상태를 복원합니다.
# 모델의 구조가 동일해야 합니다.
model.load_weights('./weights/my_model')
```

Out[33]:

```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x160488c69c8>
```

기본적으로 모델의 가중치는 텐서플로 체크포인트 파일 포맷으로 저장됩니다. 케라스의 HDF5 포맷으로 가중치를 저장할 수도 있습니다(다양한 백엔드를 지원하는 케라스 구현에서는 HDF5가 기본 설정입니다):

In [34]:

```
# 가중치를 HDF5 파일로 저장합니다.
model.save_weights('my_model.h5', save_format='h5')

# 모델의 상태를 복원합니다.
model.load_weights('my_model.h5')
```

설정

모델 설정을 저장하면 가중치는 제외하고 모델의 구조를 직렬화합니다. 원본 모델을 정의한 코드가 없어도 저장된 설정을 사용하여 동일한 구조를 만들고 초기화할 수 있습니다. 케라스는 JSON과 YAML 직렬화 포맷을 지원합니다:

In [35]:

```
# 모델을 JSON 포맷으로 직렬화합니다.
json_string = model.to_json()
json_string
```

Out[35]:

```
'{"class_name": "Sequential", "config": {"name": "sequential_6", "layers": [{"class_name":
"Dense", "config": {"name": "dense_29", "trainable": true, "batch_input_shape": [null, 32],
"dtype": "float32", "units": 64, "activation": "relu", "use_bias": true, "kernel_initializer": {"c
lass_name": "GlorotUniform", "config": {"seed": null}}, "bias_initializer": {"class_name":
"Zeros", "config": {}}, "kernel_regularizer": null, "bias_regularizer": null,
"activity_regularizer": null, "kernel_constraint": null, "bias_constraint": null}}, {"class_name":
"Dense", "config": {"name": "dense_30", "trainable": true, "dtype": "float32", "units": 10, "activ
ation": "softmax", "use_bias": true, "kernel_initializer": {"class_name": "GlorotUniform",
"config": {"seed": null}}, "bias_initializer": {"class_name": "Zeros", "config": {}},
"kernel_regularizer": null, "bias_regularizer": null, "activity_regularizer": null,
"kernel_constraint": null, "bias_constraint": null}}]}, "keras_version": "2.2.4-tf", "backend": "t
```

'kernel_constraint': null, 'bias_constraint': null}}], 'keras_version': '2.2.4-tf', 'backend': 't
ensorflow"}'

In [36]:

```
import json
import pprint
pprint.pprint(json.loads(json_string))
```

```
{'backend': 'tensorflow',
 'class_name': 'Sequential',
 'config': {'layers': [{'class_name': 'Dense',
                        'config': {'activation': 'relu',
                                   'activity_regularizer': None,
                                   'batch_input_shape': [None, 32],
                                   'bias_constraint': None,
                                   'bias_initializer': {'class_name': 'Zeros',
                                                        'config': {}},
                                   'bias_regularizer': None,
                                   'dtype': 'float32',
                                   'kernel_constraint': None,
                                   'kernel_initializer': {'class_name': 'GlorotUniform',
                                                          'config': {'seed': None}},
                                   'kernel_regularizer': None,
                                   'name': 'dense_29',
                                   'trainable': True,
                                   'units': 64,
                                   'use_bias': True}},
                       {'class_name': 'Dense',
                        'config': {'activation': 'softmax',
                                   'activity_regularizer': None,
                                   'bias_constraint': None,
                                   'bias_initializer': {'class_name': 'Zeros',
                                                        'config': {}},
                                   'bias_regularizer': None,
                                   'dtype': 'float32',
                                   'kernel_constraint': None,
                                   'kernel_initializer': {'class_name': 'GlorotUniform',
                                                          'config': {'seed': None}},
                                   'kernel_regularizer': None,
                                   'name': 'dense_30',
                                   'trainable': True,
                                   'units': 10,
                                   'use_bias': True}}],
            'name': 'sequential_6'},
 'keras_version': '2.2.4-tf'}
```

JSON 파일로부터 (완전히 새로 초기화된) 모델을 만듭니다.

In [37]:

```
fresh_model = tf.keras.models.model_from_json(json_string)
```

YAML 포맷으로 직렬화하려면 텐서플로를 임포트하기 전에 pyyaml을 설치해야 합니다:

In [38]:

```
yaml_string = model.to_yaml()
print(yaml_string)
```

```
backend: tensorflow
class_name: Sequential
config:
  layers:
  - class_name: Dense
    config:
      activation: relu
      activity_regularizer: null
      batch_input_shape: !!python/tuple
      - null
      - 32
      bias_constraint: null
```

```
      bias_initializer:
        class_name: Zeros
        config: {}
      bias_regularizer: null
      dtype: float32
      kernel_constraint: null
      kernel_initializer:
        class_name: GlorotUniform
        config:
          seed: null
      kernel_regularizer: null
      name: dense_29
      trainable: true
      units: 64
      use_bias: true
  - class_name: Dense
    config:
      activation: softmax
      activity_regularizer: null
      bias_constraint: null
      bias_initializer:
        class_name: Zeros
        config: {}
      bias_regularizer: null
      dtype: float32
      kernel_constraint: null
      kernel_initializer:
        class_name: GlorotUniform
        config:
          seed: null
      kernel_regularizer: null
      name: dense_30
      trainable: true
      units: 10
      use_bias: true
  name: sequential_6
keras_version: 2.2.4-tf
```

YAML 파일로부터 모델을 다시 만듭니다.

In [39]:

```
fresh_model = tf.keras.models.model_from_yaml(yaml_string)
```

```
C:\Users\rmfos\anaconda3\envs\opencv_workspace\lib\site-
packages\tensorflow_core\python\keras\saving\model_config.py:76: YAMLLoadWarning: calling
yaml.load() without Loader=... is deprecated, as the default Loader is unsafe. Please read
https://msg.pyyaml.org/load for full details.
  config = yaml.load(yaml_string)
```

주의: Model 클래스를 상속하여 만든 모델은 call 메서드의 본문에 파이썬 코드로 구조가 정의되어 있기 때문에 직렬화되지 않습니다.

전체 모델

가중치와 모델 설정, 심지어 옵티마이저 설정까지 포함된 전체 모델을 파일에 저장할 수 있습니다. 모델의 중간 상태를 저장하고 나중에 원본 코드가 없어도 정확히 동일한 상태에서 훈련을 재개할 수 있습니다.

In [40]:

```
# 간단한 모델을 만듭니다.
model = tf.keras.Sequential([
  layers.Dense(10, activation='softmax', input_shape=(32,)),
  layers.Dense(10, activation='softmax')
])
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels, batch_size=32, epochs=5)
```

```
# 전체 모델을 HDF5 파일로 저장합니다.
model.save('my_model.h5')

# 가중치와 옵티마이저를 포함하여 정확히 같은 모델을 다시 만듭니다.
model = tf.keras.models.load_model('my_model.h5')
```

```
Train on 1000 samples
Epoch 1/5
1000/1000 [==============================] - 0s 336us/sample - loss: 11.5277 - accuracy: 0.1020
Epoch 2/5
1000/1000 [==============================] - 0s 44us/sample - loss: 11.5327 - accuracy: 0.0910
Epoch 3/5
1000/1000 [==============================] - 0s 44us/sample - loss: 11.5419 - accuracy: 0.0970
Epoch 4/5
1000/1000 [==============================] - 0s 44us/sample - loss: 11.5395 - accuracy: 0.0980
Epoch 5/5
1000/1000 [==============================] - 0s 52us/sample - loss: 11.5321 - accuracy: 0.0830
```

즉시 실행

즉시 실행은 연산을 즉각 평가하는 명령형 프로그래밍(imperative programming) 환경입니다. 케라스에서는 즉시 실행이 필수가 아니지만 tf.keras는 이를 지원합니다. 이 기능은 프로그램을 검사하고 디버깅하는데 유용합니다.

모든 tf.keras 모델링 API는 즉시 실행과 호환됩니다. Sequential이나 함수형 API와 사용할 수 있지만 즉시 실행은 특히 모델 상속과 맞춤형 층을 만들 때 장점이 나타납니다. 이런 API는 (기존의 층을 조합하여 모델을 만드는 대신) 직접 정방향 패스의 코드를 작성하기 때문입니다.

분산 처리

다중 GPU tf.keras 모델은 tf.distribute.Strategy를 사용하여 다중 GPU에서 실행할 수 있습니다. 이 API는 기존 코드를 거의 수정하지 않고 다중 GPU에서 훈련을 분산시킬 수 있습니다.

현재는 tf.distribute.MirroredStrategy가 유일하게 지원되는 분산 전략입니다. MirroredStrategy는 한 대의 장치에서 계산 결과를 모두 수집하는 방식인 그래프 내 복제(in-graph replication)를 수행합니다. distribute.Strategy를 사용하려면 Strategy의 .scope() 안에 옵티마이저 객체 생성, 모델 구성, 컴파일 단계를 포함시킨 다음 모델을 훈련합니다.

다음 코드는 한 대의 컴퓨터에서 다중 GPU를 사용해 tf.keras.Model을 분산 처리하는 예입니다.

먼저, MirroredStrategy의 scope() 안에서 모델을 정의합니다:

In [41]:

```
strategy = tf.distribute.MirroredStrategy()

with strategy.scope():
  model = tf.keras.Sequential()
  model.add(layers.Dense(16, activation='relu', input_shape=(10,)))
  model.add(layers.Dense(1, activation='sigmoid'))

  optimizer = tf.keras.optimizers.SGD(0.2)

  model.compile(loss='binary_crossentropy', optimizer=optimizer)

model.summary()
```

```
WARNING:tensorflow:There are non-GPU devices in `tf.distribute.Strategy`, not using nccl
allreduce.
INFO:tensorflow:Using MirroredStrategy with devices
('/job:localhost/replica:0/task:0/device:CPU:0',)
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_33 (Dense)             (None, 16)                176
_________________________________________________________________
dense_34 (Dense)             (None, 1)                 17
=================================================================
Total params: 193
```

```
Total params: 193
Trainable params: 193
Non-trainable params: 0
_________________________________________________________________
```

In [42]:

```
x = np.random.random((1024, 10))
y = np.random.randint(2, size=(1024, 1))
x = tf.cast(x, tf.float32)
dataset = tf.data.Dataset.from_tensor_slices((x, y))
dataset = dataset.shuffle(buffer_size=1024).batch(32)

model.fit(dataset, epochs=1)
```

```
Train for 32 steps
32/32 [==============================] - 2s 74ms/step - loss: 0.7006
```

Out[42]:

```
<tensorflow.python.keras.callbacks.History at 0x160485dab88>
```

In [ ]:

```
```