

# HAJ Lecture

Chapter 12. 텐서플로를 사용한 사용자 정의 모델과 훈련  
Chapter 13. 텐서플로에서 데이터 적재와 전처리하기

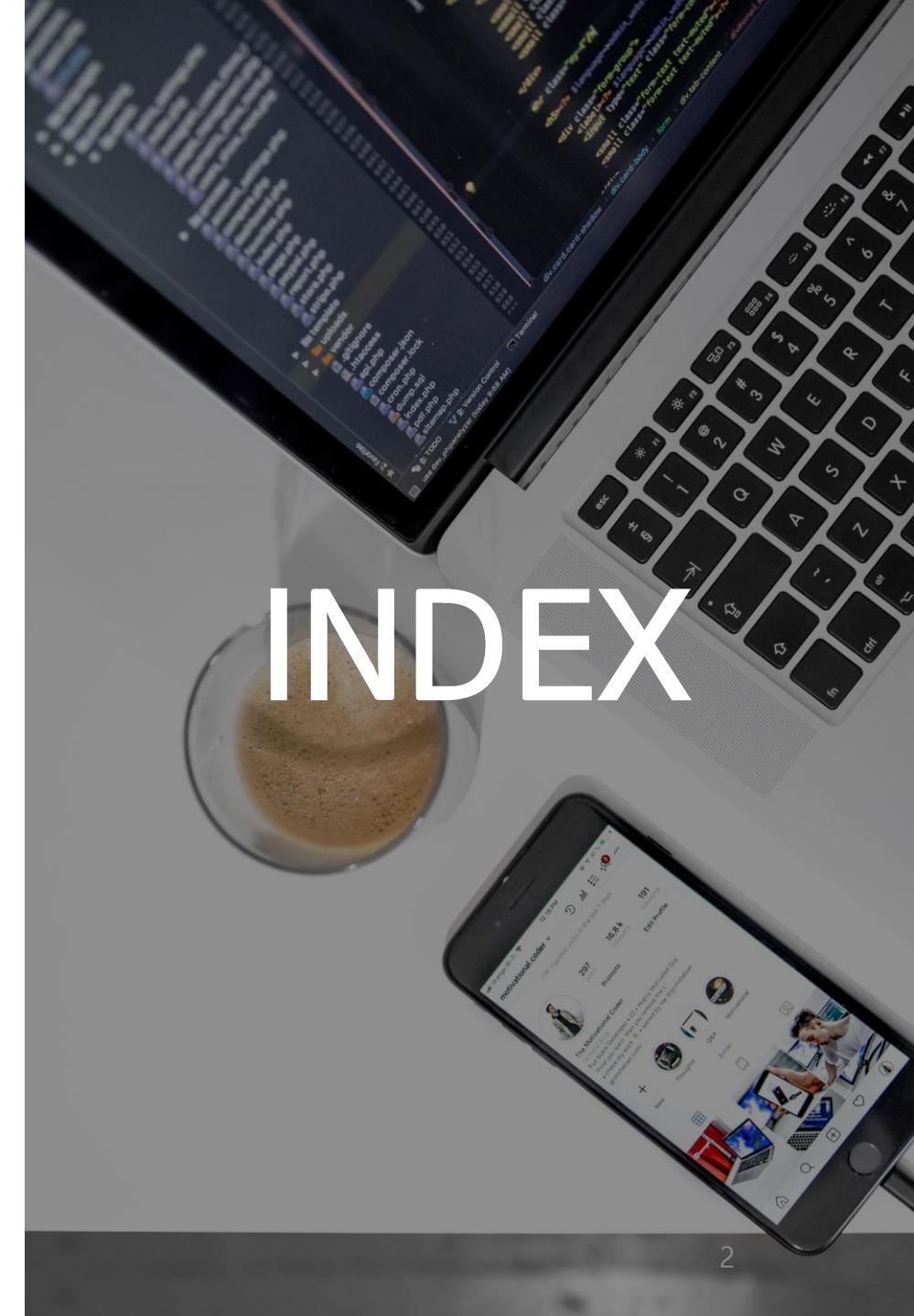
Hands-On Machine Learning  
with Scikit-Learn, Keras & TensorFlow

| 12.1 TensorFlow 훑어보기

| 12.2 Numpy처럼 TensorFlow 사용하기

| 12.3 사용자 정의 모델과 훈련 알고리즘

| 12.4 TensorFlow 함수와 그래프



## | 12.1 TensorFlow 훑어보기

### 텐서플로(TensorFlow)

수치 계산용 라이브러리, 대규모 머신러닝에 잘 맞도록 튜닝되어 있음

→ 이미지 분류, 자연어 처리, 추천 시스템, 시계열 예측 등의 작업 수행에 쓰임

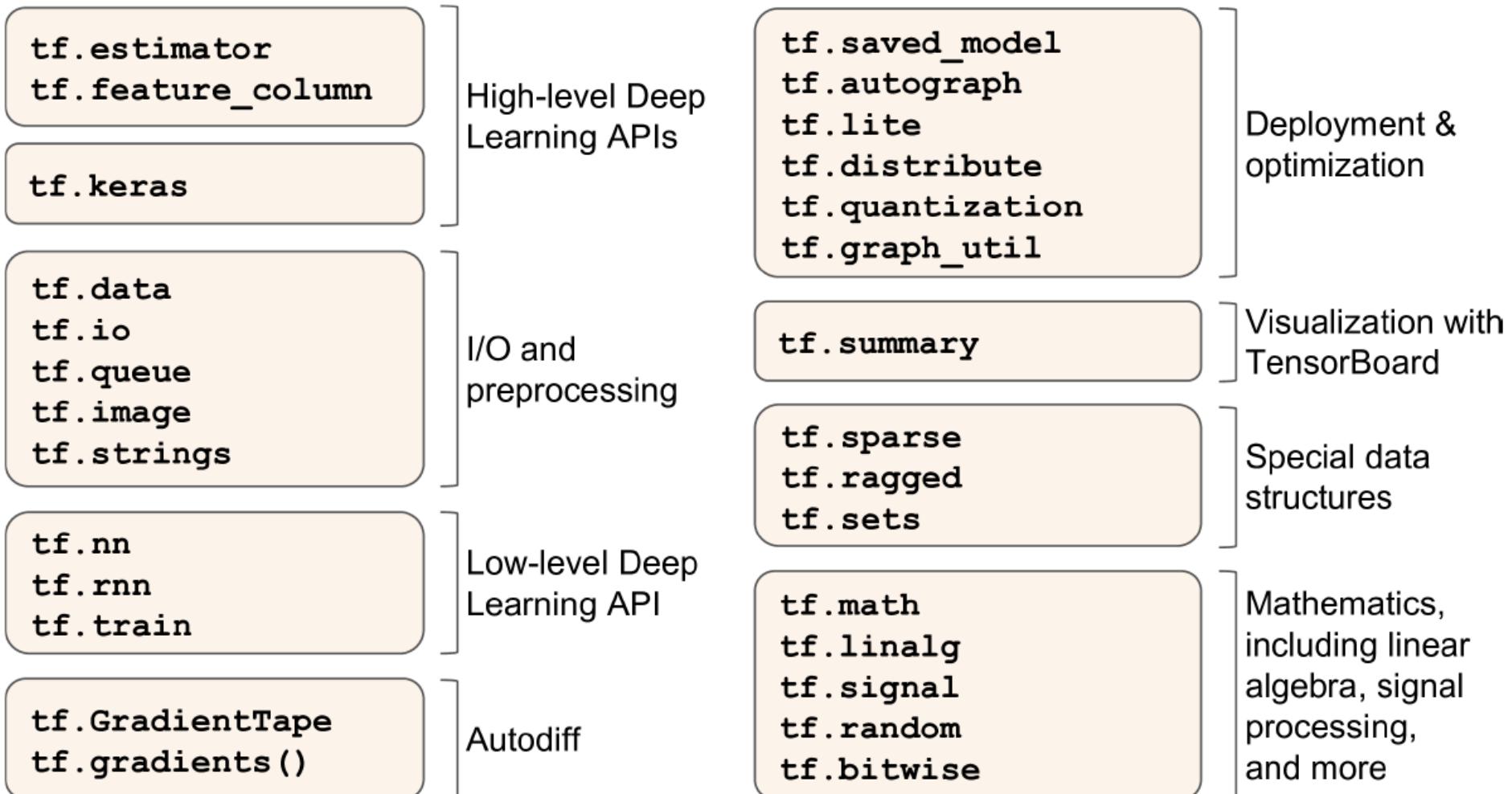
- numpy와 비슷한 구조를 띠나, GPU를 지원한다.
- JIT 컴파일러; 계산 그래프 추출 등을 동반하는 최적화를 진행한다.
- 플랫폼에 중립적인 포맷으로 내보내기가 가능하다.
- 자동 미분(autodiff) 기능과 고성능 옵티마이저를 제공  
→ 모든 종류의 손실 함수를 쉽게 최소화하는데 용이하다.



**TensorFlow** 3

# 12.1 TensorFlow 훑어보기

## 텐서플로(TensorFlow)



## 12.2 Numpy처럼 TensorFlow 사용하기

### 텐서와 연산

`tf.constant()`을 이용하여 텐서를 만들 수 있다.

e.g. 2개의 row와 3개의 column을 가진 float 행렬

```
>>> tf.constant([[1., 2., 3.], [4., 5., 6.]]) # matrix  
<tf.Tensor: id=0, shape=(2, 3), dtype=float32, numpy=  
array([[1., 2., 3.],  
       [4., 5., 6.]], dtype=float32)>  
>>> tf.constant(42) # scalar  
<tf.Tensor: id=1, shape=(), dtype=int32, numpy=42>
```

#### 1) 인덱싱

```
>>> t[:, 1:]  
<tf.Tensor: id=5, shape=(2, 2), dtype=float32, numpy=  
array([[2., 3.],  
       [5., 6.]], dtype=float32)>  
>>> t[..., 1, tf.newaxis]  
<tf.Tensor: id=15, shape=(2, 1), dtype=float32, numpy=  
array([[2.],  
       [5.]], dtype=float32)>
```

#### 2) 텐서 연산

```
>>> t + 10  
<tf.Tensor: id=18, shape=(2, 3), dtype=float32, numpy=  
array([[11., 12., 13.],  
       [14., 15., 16.]], dtype=float32)>  
>>> tf.square(t)  
<tf.Tensor: id=20, shape=(2, 3), dtype=float32, numpy=  
array([[ 1.,  4.,  9.],  
       [16., 25., 36.]], dtype=float32)>  
>>> t @ tf.transpose(t)  
<tf.Tensor: id=24, shape=(2, 2), dtype=float32, numpy=  
array([[14., 32.],  
       [32., 77.]], dtype=float32)>
```

## | 12.2 Numpy처럼 TensorFlow 사용하기

### 텐서와 넘파이

numpy 배열과 Tensor를 자유롭게 넘나들며 사용이 가능하며, 연산도 마찬가지

```
>>> a = np.array([2., 4., 5.])
>>> tf.constant(a)
<tf.Tensor: id=111, shape=(3,), dtype=float64, numpy=array([2., 4., 5.])>
>>> t.numpy() # or np.array(t)
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
>>> tf.square(a)
<tf.Tensor: id=116, shape=(3,), dtype=float64, numpy=array([4., 16., 25.])>
>>> np.square(t)
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)
```

### 타입 변환

타입 변환은 성능을 크게 감소시킬 우려가 있어, 어떤 타입변환도 자동으로 이뤄지지 X

→ 필요한 경우에는 `tf.cast()`를 이용한다.

## | 12.2 Numpy처럼 TensorFlow 사용하기

### 변수

tf.Tensor는 변경이 불가능한 객체이다.(immutable)

- 이것으로는 역전파로 변경되어야 하는 신경망의 가중치를 나타낼 수 X
- tf.Variable을 사용한다. assign( ), assign\_add( ) 등으로 수정이 가능

```
>>> v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])  
>>> v  
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=  
array([[1., 2., 3.],  
       [4., 5., 6.]], dtype=float32)>  
v.assign(2 * v)          # => [[2., 4., 6.], [8., 10., 12.]]  
v[0, 1].assign(42)      # => [[2., 42., 6.], [8., 10., 12.]]  
v[:, 2].assign([0., 1.]) # => [[2., 42., 0.], [8., 10., 1.]]  
v.scatter_nd_update(indices=[[0, 0], [1, 2]], updates=[100., 200.])  
                           # => [[100., 42., 0.], [8., 10., 200.]]
```

## | 12.2 Numpy처럼 TensorFlow 사용하기

### 다른 데이터 구조

#### 1] 희소 텐서(sparse tensor)      **tf.SparseTensor**

대부분 0으로 채워진 텐서를 효율적으로 나타낼 수 있다.

#### 2] 텐서 배열(tensor array)      **tf.TensorArray**

텐서의 리스트로서, 고정된 길이를 기본적으로 가정하지만, 동적으로 바꿀 수 있다. 한 텐서 배열 내의 모든 텐서는 데이터 타입이 동일해야 한다.

#### 3] 래그드 텐서(ragged tensor)      **tf.RaggedTensor**

리스트의 리스트를 나타낸다. 텐서에 포함된 값은 동일한 데이터 타입은 가져야 하지만 리스트의 길이는 다를 수 있다.

## | 12.3 사용자 정의 모델과 훈련 알고리즘

### 사용자 정의 손실 함수

훈련세트에 잡음이 섞여있는 경우 평균 제곱 오차보다는 후버(Huber) 손실을 비용함수로 채택하는 편이 더 적절함 (큰 오차에 다소 과한 벌칙을 행사하기 때문)

후버 손실은 케라스 API에서 공식지원X → 직접 구현해보자.

```
def huber_fn(y_true, y_pred):
    error = y_true - y_pred
    is_small_error = tf.abs(error) < 1
    squared_loss = tf.square(error) / 2
    linear_loss = tf.abs(error) - 0.5
    return tf.where(is_small_error, squared_loss, linear_loss)
```

```
model.compile(loss=huber_fn, optimizer="adam")
model.fit(X_train, y_train, [...])
```

## | 12.3 사용자 정의 모델과 훈련 알고리즘

사용자 정의 요소를 가진 모델을 저장하고 로드하기

모델을 load할 때는 함수이름과 실제 함수를 mapping한 dictionary를 전달해야 한다. → 이름 + 객체

```
model = keras.models.load_model("my_model_with_a_custom_loss.h5",
                                custom_objects={"huber_fn": huber_fn})
```

다른 기준이 필요할 때는 매개변수를 받을 수 있는 함수를 만들 수도 있다.

```
def create_huber(threshold=1.0):
    def huber_fn(y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < threshold
        squared_loss = tf.square(error) / 2
        linear_loss = threshold * tf.abs(error) - threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    return huber_fn

model.compile(loss=create_huber(2.0), optimizer="nadam")
```

## | 12.3 사용자 정의 모델과 훈련 알고리즘

사용자 정의 요소를 가진 모델을 저장하고 로드하기

모델을 저장 시 threshold 값은 저장되지 않는다.

→ 모델을 load할 때 이를 지정, keras.losses.Loss 객체를 상속하고  
get\_config() 메서드를 구현하여 해결할 수 있다.

```
class HuberLoss(keras.losses.Loss):
    def __init__(self, threshold=1.0, **kwargs):
        self.threshold = threshold
        super().__init__(**kwargs)
    def call(self, y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < self.threshold
        squared_loss = tf.square(error) / 2
        linear_loss = self.threshold * tf.abs(error) - self.threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}
```

label과 예측을 받고 모든 샘플의  
손실을 계산하여 반환

하이퍼파라미터 이름과 같이  
매핑된 dictionary를 반환

## | 12.3 사용자 정의 모델과 훈련 알고리즘

활성화 함수, 초기화, 규제, 제한을 커스터마이징하기

손실, 규제, 제한, 초기화, 지표, 활성화 함수, 층, 모델과 같은 대부분의 Keras 기능은 유사한 방법으로 커스터마이징할 수 있다.

```
def my_softplus(z): # return value is just tf.nn.softplus(z)
    return tf.math.log(tf.exp(z) + 1.0)

def my_glorot_initializer(shape, dtype=tf.float32):
    stddev = tf.sqrt(2. / (shape[0] + shape[1]))
    return tf.random.normal(shape, stddev=stddev, dtype=dtype)

def my_l1_regularizer(weights):
    return tf.reduce_sum(tf.abs(0.01 * weights))

def my_positive_weights(weights): # return value is just tf.nn.relu(weights)
    return tf.where(weights < 0., tf.zeros_like(weights), weights)

layer = keras.layers.Dense(30, activation=my_softplus,
                           kernel_initializer=my_glorot_initializer,
                           kernel_regularizer=my_l1_regularizer,
                           kernel_constraint=my_positive_weights)
```

직접 설계한 4개의 함수를 Dense 객체의 생성자에 파라미터로 넣는 방식으로 커스터마이징에 성공하였다.

## | 12.3 사용자 정의 모델과 훈련 알고리즘

### 사용자 정의 지표

- 손실: 모델을 훈련하기 위해 경사 하강법에서 사용하므로 미분 가능해야 하고, gradient가 모든 곳에서 0이 아니어야 함 e.g. cross entropy
- 지표: 모델을 평가할 때 사용하며, 미분이 가능하지 않거나 gradient가 특정 지점에서 0이 되어도 무관함 e.g. accuracy(정확도)

```
class HuberMetric(keras.metrics.Metric):  
    def __init__(self, threshold=1.0, **kwargs):  
        super().__init__(**kwargs) # handles base args (e.g., dtype)  
        self.threshold = threshold  
        self.huber_fn = create_huber(threshold)  
        self.total = self.add_weight("total", initializer="zeros")  
        self.count = self.add_weight("count", initializer="zeros")  
    def update_state(self, y_true, y_pred, sample_weight=None):  
        metric = self.huber_fn(y_true, y_pred)  
        self.total.assign_add(tf.reduce_sum(metric))  
        self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))  
    def result(self):  
        return self.total / self.count  
    def get_config(self):  
        base_config = super().get_config()  
        return {**base_config, "threshold": self.threshold}
```

스트리밍 지표(배치마다 점진적으로 업데이트 되는 것)을 만들기 위해서는 keras.metrics.Metric 객체를 상속해야 한다.

## | 12.3 사용자 정의 모델과 훈련 알고리즘

### 사용자 정의 층

- 가중치가 필요 없는 사용자 정의 층

파이썬 함수를 만든 후 keras.layers.Lambda 층으로 감싼다.

e.g. 입력에 지수함수를 적용하는 layer

```
exponential_layer = keras.layers.Lambda(lambda x: tf.exp(x))
```

- 상태가 있는(가중치가 있는) 사용자 정의 층

keras.layers.Layer를 상속해야 한다.

## 12.3 사용자 정의 모델과 훈련 알고리즘

### 사용자 정의 층

```
class MyDense(keras.layers.Layer):
    def __init__(self, units, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = keras.activations.get(activation)

    def build(self, batch_input_shape):
        self.kernel = self.add_weight(
            name="kernel", shape=[batch_input_shape[-1], self.units],
            initializer="glorot_normal")
        self.bias = self.add_weight(
            name="bias", shape=[self.units], initializer="zeros")
        super().build(batch_input_shape) # must be at the end

    def call(self, X):
        return self.activation(X @ self.kernel + self.bias)

    def compute_output_shape(self, batch_input_shape):
        return tf.TensorShape(batch_input_shape.as_list()[:-1] + [self.units])

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "units": self.units,
                "activation": keras.activations.serialize(self.activation)}
```

생성자는 하이퍼파라미터를 매개 변수로 받는다.

build( ) 메서드는 가중치마다 add\_weight( ) 메서드를 호출하여 층의 변수를 만든다.  
이 시점에는 Keras가 층의 입력크기를 알고 있어 이를 전달함

해당 층에서 필요한 연산을 수행

이 층의 출력크기를 반환

활성화 함수의 전체 설정을 저장

## | 12.3 사용자 정의 모델과 훈련 알고리즘

### 사용자 정의 모델

keras.Model 객체를 상속하여 생성자에서 층과 변수를 만들고 모델이 해야 할 작업을 call( ) 메서드에 구현한다.

```
class ResidualBlock(keras.layers.Layer):
    def __init__(self, n_layers, n_neurons, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(n_neurons, activation="elu",
                                         kernel_initializer="he_normal")
                      for _ in range(n_layers)]

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        return inputs + Z
```

```
class ResidualRegressor(keras.models.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = keras.layers.Dense(30, activation="elu",
                                         kernel_initializer="he_normal")
        self.block1 = ResidualBlock(2, 30)
        self.block2 = ResidualBlock(2, 30)
        self.out = keras.layers.Dense(output_dim)

    def call(self, inputs):
        Z = self.hidden1(inputs)
        for _ in range(1 + 3):
            Z = self.block1(Z)
        Z = self.block2(Z)
        return self.out(Z)
```

## | 12.3 사용자 정의 모델과 훈련 알고리즘

### 자동 미분을 사용하여 gradient 계산하기

변수를 정의하고, `tf.GradientTape` 블록을 만들어 이 변수와 관련된 모든 연산을 자동으로 기록한다. 마지막으로 이 테이프에 변수에 대한  $z$ 의 gradient를 요청하면 계산이 수행된다.

- `gradient()` 메서드가 호출된 후에는 자동으로 테이프가 지워진다.

```
w1, w2 = tf.Variable(5.), tf.Variable(3.)
with tf.GradientTape() as tape:
    z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])

>>> gradients
[<tf.Tensor: id=828234, shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: id=828229, shape=(), dtype=float32, numpy=10.0>]
```

## | 12.3 사용자 정의 모델과 훈련 알고리즘

### 자동 미분을 사용하여 gradient 계산하기

- `gradient()` 메서드를 2번 이상 호출해야 한다면 지속 가능한 테이프를 만들고 사용이 끝난 후 이를 삭제해야 한다.

```
with tf.GradientTape(persistent=True) as tape:  
    z = f(w1, w2)  
  
    dz_dw1 = tape.gradient(z, w1) # => tensor 36.0  
    dz_dw2 = tape.gradient(z, w2) # => tensor 10.0, works fine now!  
    del tape
```

- 테이프는 기본적으로 변수가 포함된 연산만을 기록한다. 변수가 아닌 다른 객체에 대한 `z`의 `gradient` 계산을 시도한 경우 `None`이 반환된다.

```
c1, c2 = tf.constant(5.), tf.constant(3.)  
with tf.GradientTape() as tape:  
    z = f(c1, c2)  
  
gradients = tape.gradient(z, [c1, c2]) # returns [None, None]
```

## | 12.3 사용자 정의 모델과 훈련 알고리즘

### 자동 미분을 사용하여 gradient 계산하기

- 어떤 경우에는 신경망의 일부분에 gradient가 역전파되지 않도록 막아야 하는 경우도 존재한다. 이때는 `tf.stop_gradient()` 함수를 사용해야 한다.  
이는 정방향 계산을 할 때 입력을 반환하고,  
역전파 시에는 gradient를 전파하지 않고, 상수처럼 동작시킨다.

```
def f(w1, w2):  
    return 3 * w1 ** 2 + tf.stop_gradient(2 * w1 * w2)  
  
with tf.GradientTape() as tape:  
    z = f(w1, w2) # same result as without stop_gradient()  
  
gradients = tape.gradient(z, [w1, w2]) # => returns [tensor 30., None]
```

## | 12.3 사용자 정의 모델과 훈련 알고리즘

### 사용자 정의 훈련 반복

- 간단한 모델 만들기

```
l2_reg = keras.regularizers.l2(0.05)
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="elu", kernel_initializer="he_normal",
                       kernel_regularizer=l2_reg),
    keras.layers.Dense(1, kernel_regularizer=l2_reg)
])
```

- 훈련세트에서 샘플 batch를 랜덤하게 추출하는 함수 정의

```
def random_batch(X, y, batch_size=32):
    idx = np.random.randint(len(X), size=batch_size)
    return X[idx], y[idx]
```

- 지표를 포함한 훈련상태를 출력하는 함수

```
def print_status_bar(iteration, total, loss, metrics=None):
    metrics = " - ".join(["{}: {:.4f}"].format(m.name, m.result())
                         for m in [loss] + (metrics or [])))
    end = "" if iteration < total else "\n"
    print("\r{} / {} - ".format(iteration, total) + metrics,
          end=end)
```

## | 12.3 사용자 정의 모델과 훈련 알고리즘

### 사용자 정의 훈련 반복

- 몇 개의 하이퍼파라미터를 정의하고,  
옵티마이저, 손실함수, 지표를 선택
- 사용자 정의 훈련 반복

```
n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
optimizer = keras.optimizers.Nadam(lr=0.01)
loss_fn = keras.losses.mean_squared_error
mean_loss = keras.metrics.Mean()
metrics = [keras.metrics.MeanAbsoluteError()]
```

```
for epoch in range(1, n_epochs + 1):
    print("Epoch {} / {}".format(epoch, n_epochs))
    for step in range(1, n_steps + 1):
        X_batch, y_batch = random_batch(X_train_scaled, y_train)
        with tf.GradientTape() as tape:
            y_pred = model(X_batch, training=True)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
        mean_loss(loss)
        for metric in metrics:
            metric(y_batch, y_pred)
        print_status_bar(step * batch_size, len(y_train), mean_loss, metrics)
    print_status_bar(len(y_train), len(y_train), mean_loss, metrics)
    for metric in [mean_loss] + metrics:
        metric.reset_states()
```

## 12.4 TensorFlow 함수와 그래프

step1. 입력의 세제곱을 계산하는 함수

```
def cube(x):
    return x ** 3
```

step2. 파이썬 상수(정수, 실수 …), 텐서를 사용하여 함수 호출

```
>>> cube(2)
8
>>> cube(tf.constant(2.0))
<tf.Tensor: id=18634148, shape=(), dtype=float32, numpy=8.0>
```

step3. tf.function( )을 사용하여 파이썬 함수→TensorFlow 함수 전환하기

```
>>> tf_cube = tf.function(cube)
>>> tf_cube
<tensorflow.python.eager.def_function.Function at 0x1546fc080>
```

tf.function 데코레이터 활용

```
@tf.function
def tf_cube(x):
    return x ** 3
```

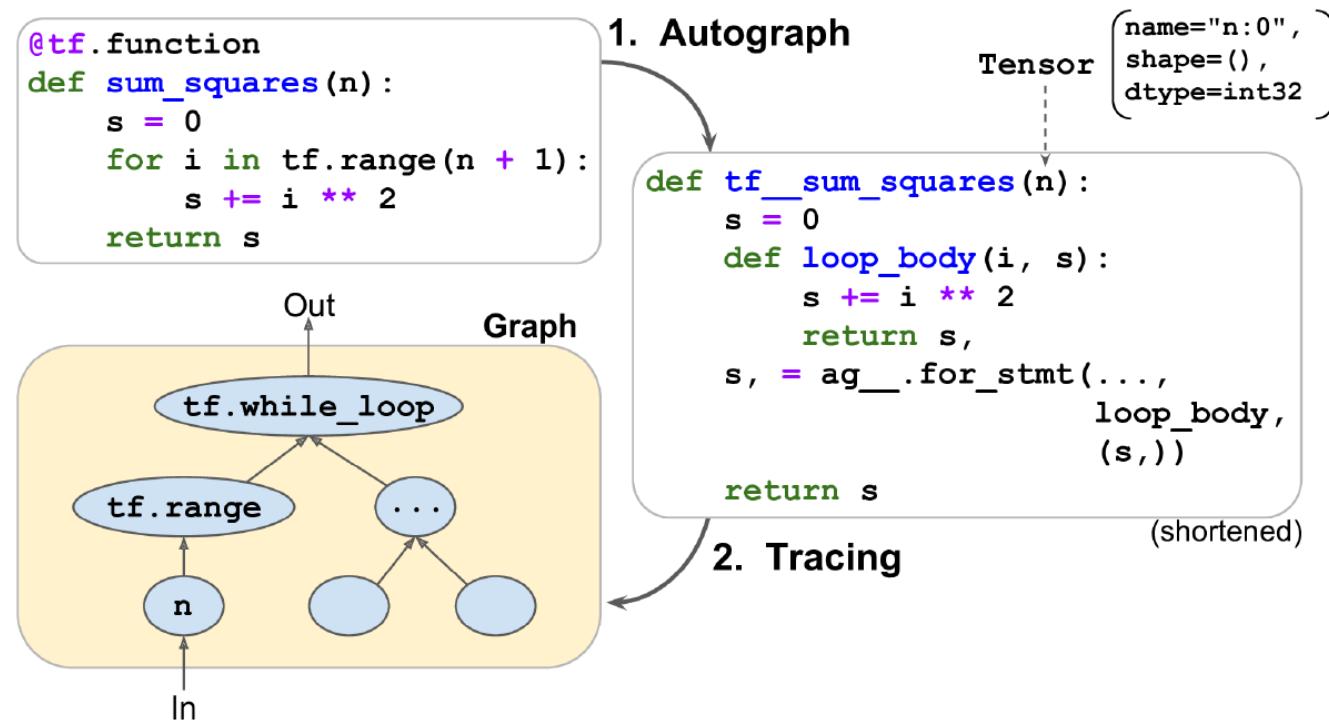
## 12.4 TensorFlow 함수와 그래프

### 오토그래프(autograph)

함수의 소스코드를 분석하여 for, while, if 등의 제어문을 찾아내는 과정

→ 코드분석 후, 함수의 모든 제어문을 TensorFlow 연산으로 바꾼 버전을 생성

→ 이 바꾼 버전의 함수를 호출, 매개변수 값 대신 symbolic tensor를 전달



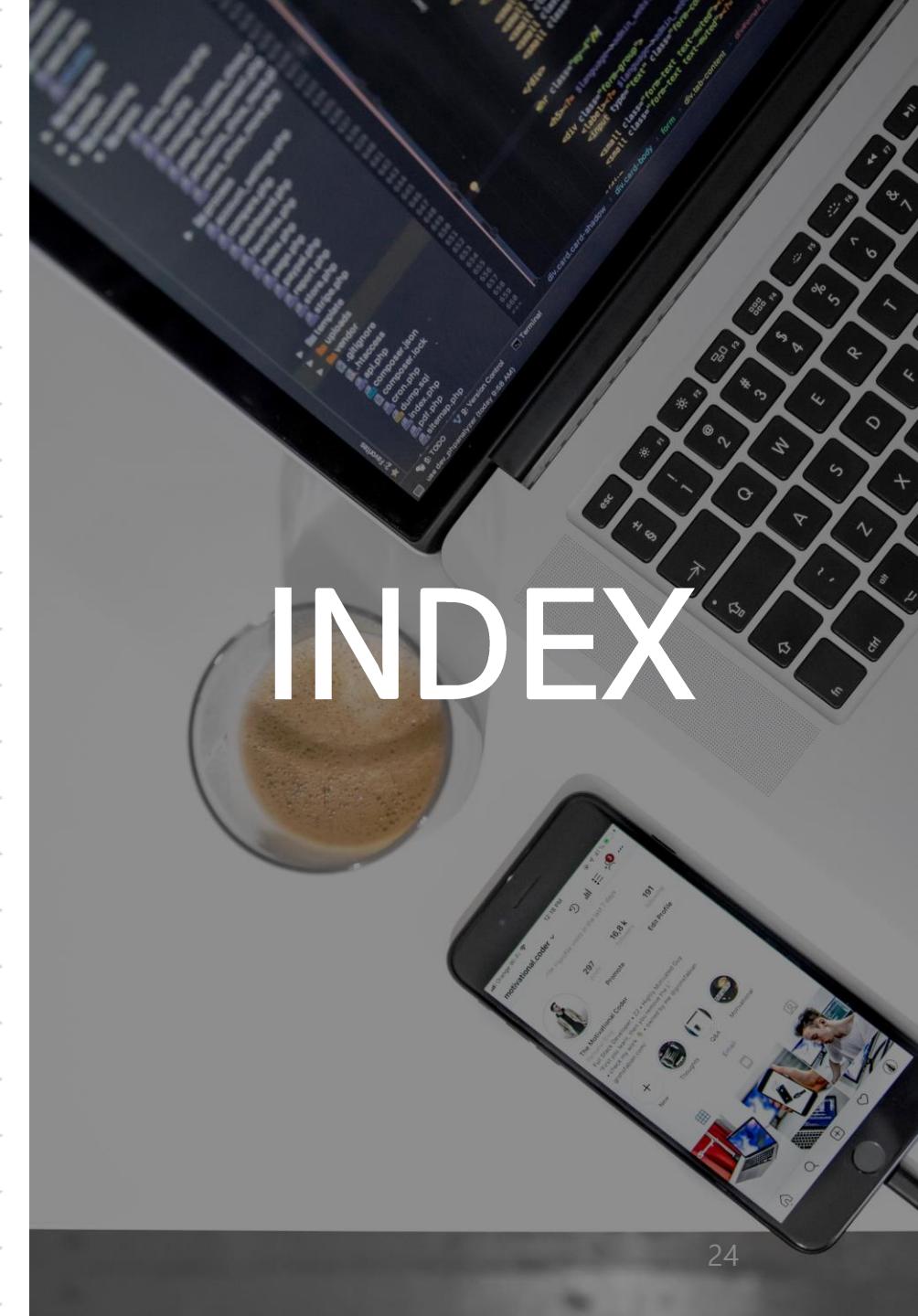
| 13.1 데이터 API

| 13.2 TFRecord 포맷

| 13.3 입력 특성 전처리

| 13.4 TF 변환

| 13.5 TensorFlow 데이터셋(TFDS) 프로젝트



INDEX

## 13.1 데이터 API

### 데이터셋

연속된 데이터 샘플의 집합

→ 디스크에서 데이터를 점진적으로 읽는 데이터셋을 주로 사용한다.

```
>>> X = tf.range(10) # any data tensor  
>>> dataset = tf.data.Dataset.from_tensor_slices(X)  
>>> dataset  
<TensorSliceDataset shapes: (), types: tf.int32>
```

tf.data.Dataset.from\_tensor\_slices()

메모리에서 전체 데이터셋 생성

from\_tensor\_slices()

Tensor를 받아 X의 각 원소가 item으로 표현되는 tf.data.Dataset을 생성

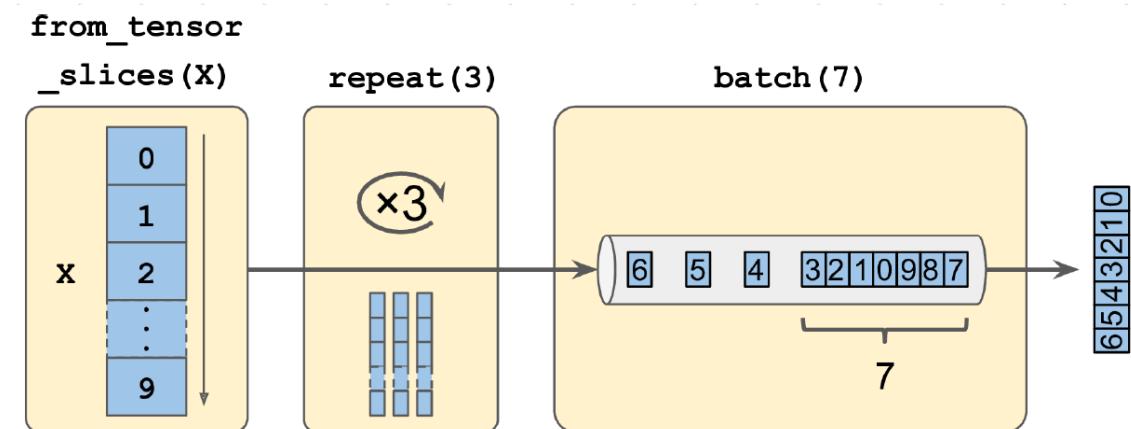
```
>>> for item in dataset:  
...     print(item)  
...  
tf.Tensor(0, shape=(), dtype=int32)  
tf.Tensor(1, shape=(), dtype=int32)  
tf.Tensor(2, shape=(), dtype=int32)  
[...]  
tf.Tensor(9, shape=(), dtype=int32)
```

# 13.1 데이터 API

## 연쇄 변환

변환 메소드를 호출하여 여러 종류의 변환이 가능한데, 이를 연결시킬 수도 있다.

```
>>> dataset = dataset.repeat(3).batch(7)  
>>> for item in dataset:  
...     print(item)  
...  
tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int32)  
tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int32)  
tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int32)  
tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int32)  
tf.Tensor([8 9], shape=(2,), dtype=int32)
```



- `repeat( int n )`: 원본 데이터셋의 아이템을 n 차례 반복하는 새로운 데이터셋을 반환
- `batch( int m )`: 이전 데이터셋의 아이템을 m개씩 그룹으로 묶는다.

## 13.1 데이터 API

### 연쇄 변환

- `map()` : 각 아이템에 변환을 적용

e.g. 모든 아이템에 2를 곱하여 새로운 데이터셋을 생성

```
>>> dataset = dataset.map(lambda x: x * 2) # Items: [0,2,4,6,8,10,12]
```

- `apply()` : 데이터셋 전체에 변환을 적용

e.g. 7개의 정수로 이루어진 batch → 하나의 정수 tensor

```
>>> dataset = dataset.apply(tf.data.experimental.unbatch()) # Items: 0,2,4,...
```

- `filter()` : 데이터셋 필터링 (+ 조건 필요)

e.g.

```
>>> dataset = dataset.filter(lambda x: x < 10) # Items: 0 2 4 6 8 0 2 4 6...
```

## 13.1 데이터 API

### 데이터 셔플링

- 경사 하강법은 훈련세트의 샘플이 독립적이고 동일한 분포일 때 최고의 성능을 발휘한다. → shuffle()을 이용하여 샘플을 섞는다.
- 원본 데이터셋의 처음 item을 buffer\_size 만큼 추출하여 버퍼에 채움
- 버퍼에서 랜덤하게 하나를 꺼내 반환하고, 새로운 아이템을 추출하여 비워진 버퍼를 채우는 방식
- 따라서, 버퍼 크기를 충분히 크게 해야 shuffling의 효과가 극대화된다.

```
>>> dataset = tf.data.Dataset.range(10).repeat(3) # 0 to 9, three times
>>> dataset = dataset.shuffle(buffer_size=5, seed=42).batch(7)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([0 2 3 6 7 9 4], shape=(7,), dtype=int64)
tf.Tensor([5 0 1 1 8 6 5], shape=(7,), dtype=int64)
tf.Tensor([4 8 7 1 2 3 0], shape=(7,), dtype=int64)
tf.Tensor([5 4 2 7 8 9 9], shape=(7,), dtype=int64)
tf.Tensor([3 6], shape=(2,), dtype=int64)
```

## 13.1 데이터 API

### 데이터 전처리

- 입력 특성에서 평균을 빼고 표준편차로 나누어 스케일을 조정(표준화)

```
X_mean, X_std = [...] # mean and scale of each feature in the training set  
n_inputs = 8
```

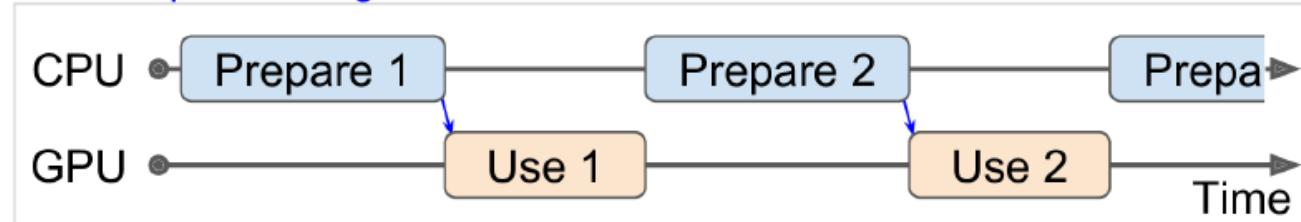
```
def preprocess(line):  
    defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]  
    fields = tf.io.decode_csv(line, record_defaults=defs)  
    x = tf.stack(fields[:-1])  
    y = tf.stack(fields[-1:])  
    return (x - X_mean) / X_std, y  
  
>>> preprocess(b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782')  
(<tf.Tensor: id=6227, shape=(8,), dtype=float32, numpy=  
array([ 0.16579159,  1.216324   , -0.05204564, -0.39215982, -0.5277444 ,  
       -0.2633488 ,  0.8543046 , -1.3072058 ], dtype=float32)>,  
<tf.Tensor: [...]>, numpy=array([2.782], dtype=float32))
```

## 13.1 데이터 API

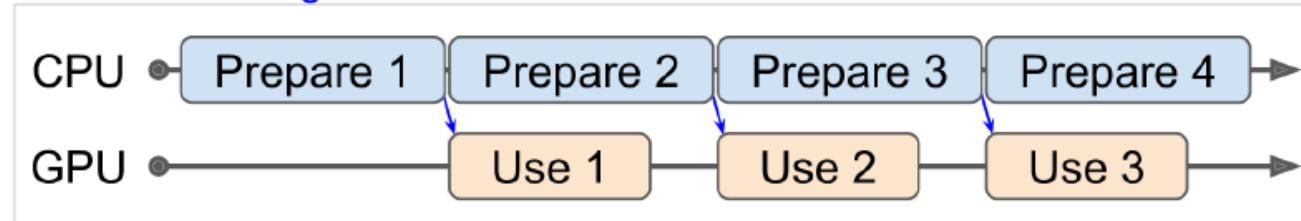
### 프리패치

- 데이터셋이 항상 한 배치가 미리 준비되도록 하는 것  
→ 훈련 알고리즘이 한 배치로 작업을 하는 동안 이 데이터셋이 동시에 다음 배치를 준비한다.(e.g. 데이터를 읽고 미리 전처리 수행)
- 매 순간마다 CPU, GPU에서는 데이터를 처리하게 된다. (쉬지X)

Without prefetching



With Prefetching



## 13.2 TFRecord 포맷

### TFRecord

크기가 다른 연속된 이진 레코드를 저장하는 단순한 이진 포맷  
(레코드 길이, 길이확인 CRC 체크섬, 실제 데이터, 데이터확인 CRC 체크섬)

### tf.io.TFRecordWriter 객체

```
with tf.io.TFRecordWriter("my_data.tfrecord") as f:  
    f.write(b"This is the first record")  
    f.write(b"And this is the second record")  
  
filepaths = ["my_data.tfrecord"]  
dataset = tf.data.TFRecordDataset(filepaths)  
for item in dataset:  
    print(item)  
  
tf.Tensor(b'This is the first record', shape=(), dtype=string)  
tf.Tensor(b'And this is the second record', shape=(), dtype=string)
```

## 13.2 TFRecord 포맷

TFRecord 파일 압축(특히 네트워크 환경일 때 유용)

```
options = tf.io.TFRecordOptions(compression_type="GZIP")
with tf.io.TFRecordWriter("my_compressed.tfrecord", options) as f:
    [...]
dataset = tf.data.TFRecordDataset(["my_compressed.tfrecord"],
                                  compression_type="GZIP")
```

압축파일 생성

압축파일 읽기

일반적으로 직렬화된 프로토콜 버퍼를 담고 있다.

```
syntax = "proto3";
message Person {
    string name = 1;
    int32 id = 2;
    repeated string email = 3;
}
```

의미: 프로토콜 버퍼 포맷 버전 3 사용  
Person 객체는 string타입 name, int32타입 id,  
string타입 email을 가지고, 각각의 숫자는 필드 식  
별자로 레코드의 이진 표현에 사용됨

## 13.2 TFRecord 포맷

```
>>> from person_pb2 import Person # import the generated access class
>>> person = Person(name="Al", id=123, email=["a@b.com"]) # create a Person
>>> print(person) # display the Person
name: "Al"
id: 123
email: "a@b.com"
>>> person.name # read a field
"Al"
>>> person.name = "Alice" # modify a field
>>> person.email[0] # repeated fields can be accessed like arrays
"a@b.com"
>>> person.email.append("c@d.com") # add an email address
>>> s = person.SerializeToString() # serialize the object to a byte string
>>> s
b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com'
>>> person2 = Person() # create a new Person
>>> person2.ParseFromString(s) # parse the byte string (27 bytes long)
27
>>> person == person2 # now they are equal
True
```

## 13.2 TFRecord 포맷

### TensorFlow 프로토콜 버퍼

```
syntax = "proto3";
message BytesList { repeated bytes value = 1; }
message FloatList { repeated float value = 1 [packed = true]; }
message Int64List { repeated int64 value = 1 [packed = true]; }
message Feature {
    oneof kind {
        BytesList bytes_list = 1;
        FloatList float_list = 2;
        Int64List int64_list = 3;
    }
};
message Features { map<string, Feature> feature = 1; };
message Example { Features features = 1; };
```

## 13.2 TFRecord 포맷

### TensorFlow 프로토콜 버퍼

```
from tensorflow.train import BytesList, FloatList, Int64List
from tensorflow.train import Feature, Features, Example

person_example = Example(
    features=Features(
        feature={
            "name": Feature(bytes_list=BytesList(value=[b"Alice"])),
            "id": Feature(int64_list=Int64List(value=[123])),
            "emails": Feature(bytes_list=BytesList(value=[b"a@b.com",
                                              b"c@d.com"]))
        })
)

with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:
    f.write(person_example.SerializeToString())
```

Example 프로토콜 버퍼를 만들고  
SerializeToString( ) 메서드를 호출하여 직렬화하고  
결과 데이터를 TFRecord 파일에 저장

### | 13.3 입력 특성 전처리

신경망을 위해 데이터를 준비하려면 모든 특성을 **수치 특성**으로 변환하고 정규화해야 한다.

만일 데이터에 범주형 특성이나 텍스트 특성이 있다면 이를 숫자로 바꾸어야 한다. (→ 인코딩)

- 원-핫 벡터(one-hot vector)
- 임베딩(embedding)

## | 13.3 입력 특성 전처리

원-핫 벡터를 사용해 범주형 특성 인코딩하기

범주 개수가 작은 경우에 적합하며, lookup table을 사용해 각 범주를 인덱스 맵핑

```
vocab = ["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]  
indices = tf.range(len(vocab), dtype = tf.int64)
```

```
table_init = tf.lookup.KeyValueTensorInitializer(vocab, indices)
```

```
num_oov_buckets = 2
```

```
table = tf.lookup.StaticVocabularyTable(table_init, num_oov_buckets)
```

```
categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
```

```
cat_indices = table.lookup(categories)
```

```
cat_indices
```

```
<tf.Tensor: id=514, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>
```

```
cat_one_hot = tf.one_hot(cat_indices, depth=len(vocab) + num_oov_buckets)
```

```
cat_one_hot
```

```
<tf.Tensor: id=524, shape=(4, 7), dtype=float32, numpy=
```

```
array([[0., 0., 0., 1., 0., 0., 0.],
```

```
      [0., 0., 0., 0., 1., 0., 0.],
```

```
      [0., 1., 0., 0., 0., 0., 0.],
```

```
      [0., 1., 0., 0., 0., 0., 0.]]), dtype=float32>
```

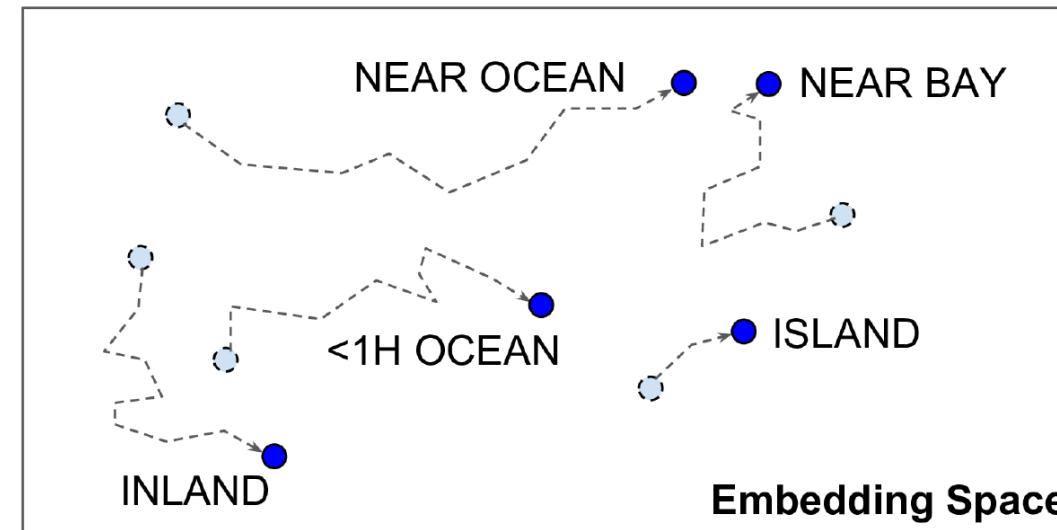
## | 13.3 입력 특성 전처리

임베딩을 사용해 범주형 특성 인코딩하기

어휘 사전이 커질 경우에는 임베딩이 적합

범주를 표현하는 훈련 가능한 밀집 벡터

**표현학습**: 표현이 좋을수록 신경망이 정확한 예측을 만들기 쉬워지고,  
범주가 유용하게 표현되도록 임베딩이 훈련되는 경향을 띠는 것



## | 13.3 입력 특성 전처리

### 임베딩을 사용해 범주형 특성 인코딩하기

각 범주의 임베딩을 담은 임베딩 행렬

→ 변수에 저장된다.(훈련 과정에서 경사 하강법으로 학습할 수 있다.)

`tf.nn.embedding_lookup()`

임베딩 행렬에서 주어진 인덱스에 해당하는 행을 찾는다.

`keras.layers.Embedding`

이 층이 생성될 때 임베딩 행렬을 랜덤하게 초기화하고 어떤 범주 인덱스로 호출될 때  
임베딩 행렬에 있는 그 인덱스의 행을 반환

## 13.4 TF 변환

- 전처리는 계산비용이 크다. → 사전에 처리하면 속도를 향상시킬 수 있다.
- 데이터 크기가 작은 경우 → cache() 사용
- 데이터 크기가 큰 경우 → Apache Beam이나 Spark 사용
- 만약 전처리 과정이 바뀌어야 하는 상황 등이 온다면? → 유지보수 힘들어짐  
→ 훈련/서빙 왜곡(training/serving skew)
- 전처리 연산을 딱 한 번만 정의하는 것이 필요 → “TF 변환”의 탄생

```
import tensorflow_transform as tft

def preprocess(inputs): # inputs is a batch of input features
    median_age = inputs["housing_median_age"]
    ocean_proximity = inputs["ocean_proximity"]
    standardized_age = tft.scale_to_z_score(median_age - tft.mean(median_age))
    ocean_proximity_id = tft.compute_and_apply_vocabulary(ocean_proximity)
    return {
        "standardized_median_age": standardized_age,
        "ocean_proximity_id": ocean_proximity_id
    }
```



Thank You