12.3 사용자 정의 모델과 훈련 알고리즘

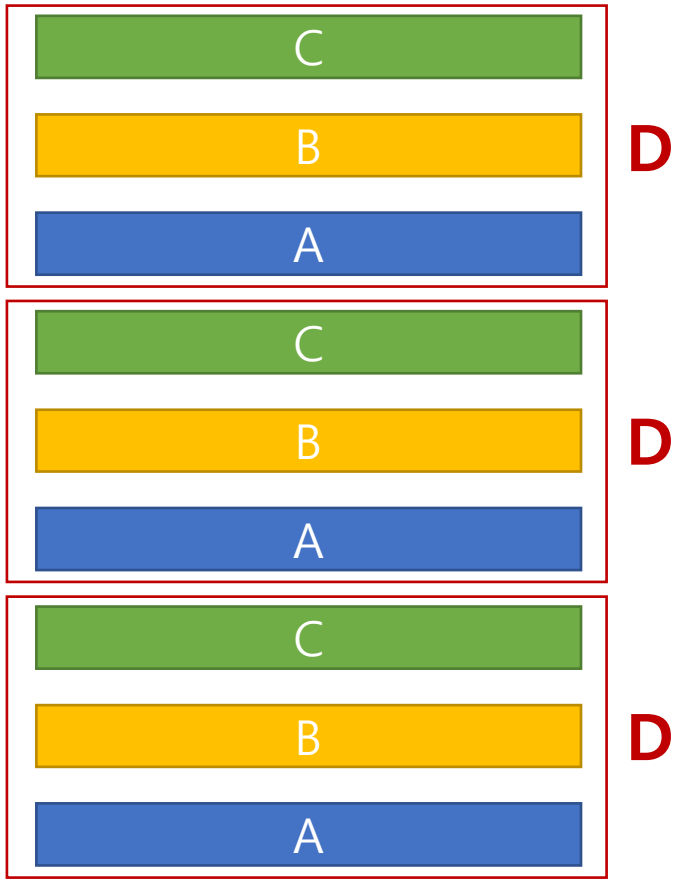
12.3.5 사용자 정의 층
12.3.6 사용자 정의 모델
12.3.7 모델 구성 요소에 기반한 손실과 지표
12.3.8 자동 미분을 사용하여 그레이디언트 계산하기

핸즈온 머신러닝 2판
- 오렐리안 제롱

화학소재솔루션센터
김민근

12.3.5. 사용자 정의 층

텐서플로에는 없는 특이한 층을 가진 네트워크가 필요할 때 사용자 정의 층을 만들어 사용



예) 모델 층이 A,B,C,A,B,C,A,B,C 순서대로 구성되어 있다면 **반복된 A,B,C**를 **사용자 정의 층 D**로 정의해 D,D,D로 구성된 모델을 만들 수 있다.

- **가중치가 필요 없는 경우**, 파이썬 함수를 만든 후 **keras.layers.Lambda** 층으로 감싼다.

ex) 입력에 지수 함수를 적용하는 층:

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

KRICT 한국화학연구원

12.3.5. 사용자 정의 층

상태가 있는(즉, **가중치를 가진 층**)을 만들려면 **keras.layers.Layer** 를 상속해야 함

예) Dense층의 간소화 버전을 구현
생성자는 activation 문자열 매개변수를 받아 적절한 활성화 함수를 설정

```
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)}
```

12.3.5. 사용자 정의 층

상태가 있는(즉, **가중치를 가진 층**)을 만들려면 **keras.layers.Layer** 를 상속해야 함

build() 메서드:
가중치마다 add_weight 함수를 호출하여 층의 변수(커널과 편향)를 만듬

```
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)}
```

12.3.5. 사용자 정의 층

상태가 있는(즉, **가중치를 가진 층**)을 만들려면 **keras.layers.Layer** 를 상속해야 함

call() 메서드:
이 층에 필요한 연산을 수행

```
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)}
```

12.3.5. 사용자 정의 층

상태가 있는(즉, **가중치를 가진 층**)을 만들려면 **keras.layers.Layer** 를 상속해야 함

compute_output_shape() 메서드:
이층의 출력 크기를 반환
여기서는 마지막 차원을 제외하고는 입력과 크기가 같다.

NOTE: 동적인 층을 제외하고는 tf.keras가 자동으로 출력 크기를 추측할 수 있다면 이 메서드를 생략할 수 있다.

```
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)}
```

12.3.5. 사용자 정의 층

상태가 있는(즉, **가중치를 가진 층**)을 만들려면 **keras.layers.Layer** 를 상속해야 함

get_config() 메서드:
keras.activations.serialize()를 사용하여 활성화 함수의 전체 설정을 저장

```
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)}
```

12.3.5. 사용자 정의 층

상태가 있는(즉, **가중치를 가진 층**)을 만들려면 **keras.layers.Layer** 를 상속해야 함

MyDense 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)}
```

12.3.5. 사용자 정의 층

여러가지 입력/출력을 받는 층

```
class MyMultiLayer(keras.layers.Layer):
    def call(self, X):
        X1, X2 = X
        return [X1 + X2, X1 * X2, X1 / X2]

    def compute_output_shape(self, batch_input_shape):
        b1, b2 = batch_input_shape
        return [b1, b1, b1] # should probably handle broadcasting rules
```

- 여러 가지 입력을 받는 층을 만들려면 call() 메서드에 모든 입력이 포함된 튜플을 매개변수 값으로 전달해야 함
- compute_output_shape() 메서드의 매개변수도 각 입력의 배치 크기를 담은 튜플이어야 함
- 여러 출력을 가진 층을 만들려면 call() 메서드가 출력을 반환해야 함
- 예시의 코드는 두 개의 입력과 세 개의 출력을 만드는 층
- 하나의 입력과 하나의 출력을 가진 층만 사용하는 시퀀셜 API에는 사용할 수 없다.

12.3.5. 사용자 정의 층

훈련과 테스트에서 다르게 동작하는 층

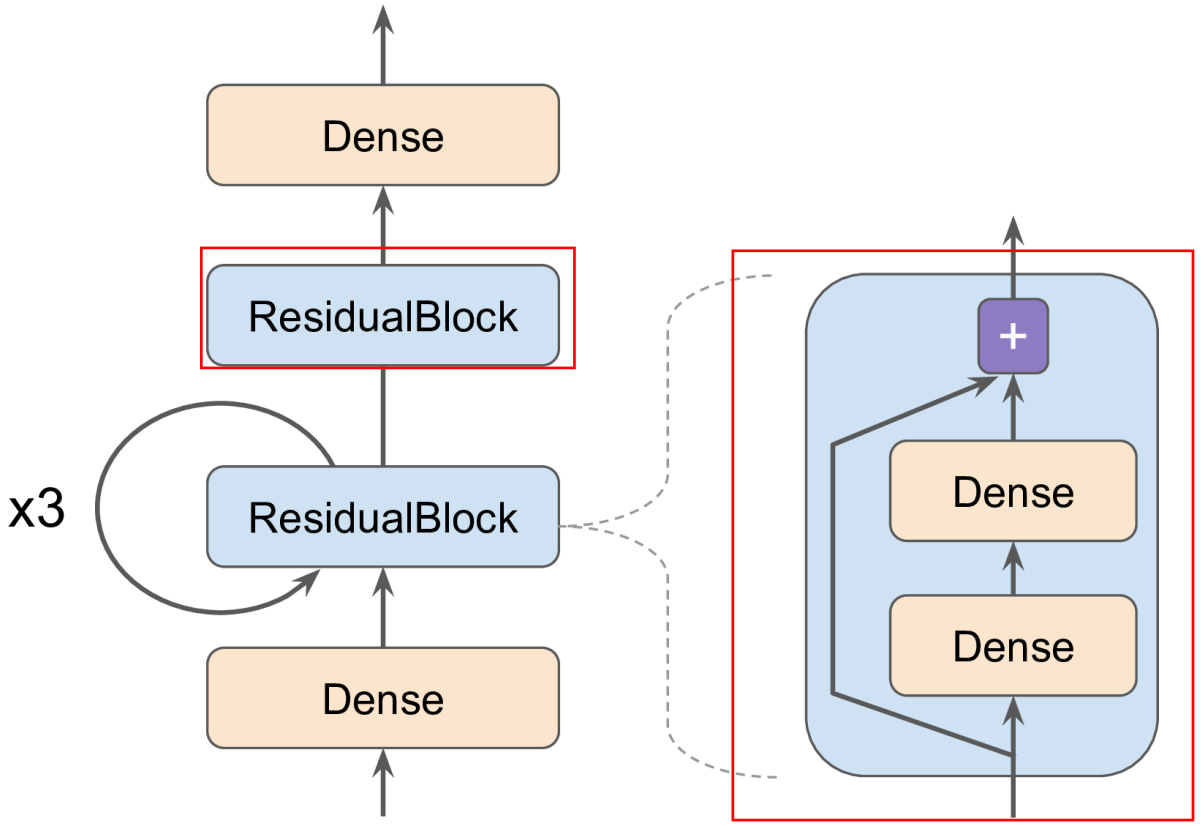
```
class MyGaussianNoise(keras.layers.Layer):
    def __init__(self, stddev, **kwargs):
        super().__init__(**kwargs)
        self.stddev = stddev

    def call(self, X, training=None):
        if training:
            noise = tf.random.normal(tf.shape(X), stddev=self.stddev)
            return X + noise
        else:
            return X

    def compute_output_shape(self, batch_input_shape):
        return batch_input_shape
```

- call() 메서드에서 training 매개변수를 추가하여 훈련인지 테스트인지를 결정
- 예) 훈련하는 동안 (규제 목적으로) 가우스 잡음을 추가하고 테스트 시에는 아무것도 하지 않는 층>
  > 케라스에서는 이와 동일한 작업을 하는 층이 있다 keras.layers.GaussianNoise

12.3.6. 사용자 정의 모델



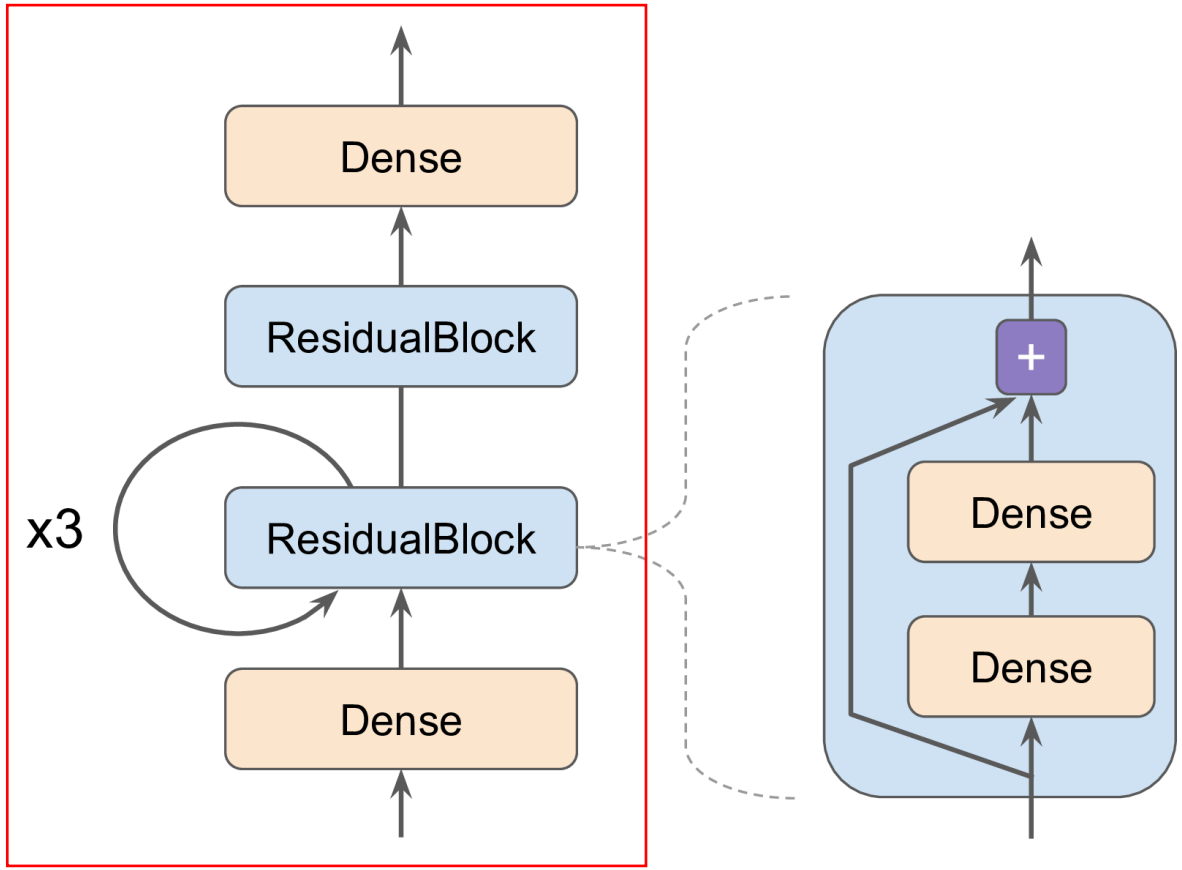

```python
class ResidualBlock(keras.layers.Layer):
    def __init__(self, n_layers, n_neurons, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(n_neurons, activation="elu",
                                          kernel_initializer="he_normal")
              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

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

- Keras.Model 클래스를 상속하여 생성자에서 층과 변수를 만들고 모델이 해야할 작업을 call() 메서드에서 구현
- 잔차 블록(residual block: 14장에서 다시 한번 다룬다): 출력에 입력을 더함
- n_layers의 만큼의 Dense층을 가지는 ResidualBlock층

12.3.6. 사용자 정의 모델

서브클래싱 API를 사용해 이 모델 정의




```python
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)
```

- 입력이 첫 번째 완전 연결 층을 통과하여 두 개의 완전 연결 층과 스킵 연결로 구성된 잔차 블록(residual block)으로 전달된다.
- 그 다음 동일한 잔차 블록에 세 번 더 통과시킨다.
- 그 다음 두 번째 잔차 블록을 지나 마지막 출력이 완전 연결된 출력 층에 전달

12.3.7. 모델 구성 요소에 기반한 손실과 지표

- 앞서 정의한 사용자 손실과 지표는 모두 레이블과 예측을 기반으로 함
- 은닉층의 가중치나 활성화 함수 등과 같이 모델 구성 요소에 기반한 손실을 정의해야 할 때가 있음
- 이런 손실은 모델의 내부 모니터링 상황을 모니터링 할 때 유용함
- 모델 구성 요소에 기반한 손실을 정의>계산>add_loss()에 전달
- 예) 다섯 개의 은닉층과 출력층으로 구성된 MLP 모델 > 맨 위의 은닉층에 보조 출력을 가짐 >> 보조출력에 연결된 손실-재구성 손실 (17장)이라 정의

```
class ReconstructingRegressor(keras.models.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(30, activation="selu",
                                          kernel_initializer="lecun_normal")

                       for _ in range(5)]
        self.out = keras.layers.Dense(output_dim)

    def build(self, batch_input_shape):
        n_inputs = batch_input_shape[-1]
        self.reconstruct = keras.layers.Dense(n_inputs)
        super().build(batch_input_shape)

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        reconstruction = self.reconstruct(Z)
        recon_loss = tf.reduce_mean(tf.square(reconstruction - inputs))
        self.add_loss(0.05 * recon_loss)
        return self.out(Z)
```


12.3.7. 모델 구성 요소에 기반한 손실과 지표

- 생성자가 다섯개의 은닉층과 하나의 출력층으로 구성된 심층 신경망
- build() 메서드에서 덴스 레이어를 하나 더 추가하여 모델의 입력을 재구성하는데 사용, 완전 연결층의 유닛 개수는 입력 개수와 같아야 함.
- call() 메서드에서 입력이 다섯개의 은닉층에 모두 통과시킴, 결과값을 재구성층에 넘겨 재구성 값을 만듬

```
class ReconstructingRegressor(keras.models.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(30, activation="selu",
                                          kernel_initializer="lecun_normal")
                       for _ in range(5)]
        self.out = keras.layers.Dense(output_dim)

    def build(self, batch_input_shape):
        n_inputs = batch_input_shape[-1]
        self.reconstruct = keras.layers.Dense(n_inputs)
        super().build(batch_input_shape)

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        reconstruction = self.reconstruct(Z)
        recon_loss = tf.reduce_mean(tf.square(reconstruction - inputs))
        self.add_loss(0.05 * recon_loss)
        return self.out(Z)
```


12.3.7. 모델 구성 요소에 기반한 손실과 지표

- call() 메서드에서 재구성 손실을 계산하고 add_loss() 메서드를 사용해 모델의 손실 리스트에 추가. 재구성 손실이 주 손실을 압도하지 않도록 0.05를 곱해 크기를 줄임
- call() 메서드 마지막에서 출력층에 결과를 반환함

```
class ReconstructingRegressor(keras.models.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(30, activation="selu",
                                          kernel_initializer="lecun_normal")
                       for _ in range(5)]
        self.out = keras.layers.Dense(output_dim)

    def build(self, batch_input_shape):
        n_inputs = batch_input_shape[-1]
        self.reconstruct = keras.layers.Dense(n_inputs)
        super().build(batch_input_shape)

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        reconstruction = self.reconstruct(Z)
        recon_loss = tf.reduce_mean(tf.square(reconstruction - inputs))
        self.add_loss(0.05 * recon_loss)
        return self.out(Z)
```

참조: TF 2.2에 있는 이슈(#46858) 때문에 build() 메서드와 add_loss()를 함께 사용할 수 없습니다.

출처: https://zenoahn.tistory.com/112 [제노 엔진:티스토리]

12.3.8. 자동 미분을 사용하여 그레이디언트 계산하기

```
def f(w1, w2):
    return 3 * w1 ** 2 + 2 * w1 * w2
```

$$f(w_1, w_2) = 3w_1^2 + 2w_1w_2$$

$$f'_{w_1}(w_1, w_2) = 6w_1 + 2w_2$$

$$f'_{w_2}(w_1, w_2) = 2w_1$$

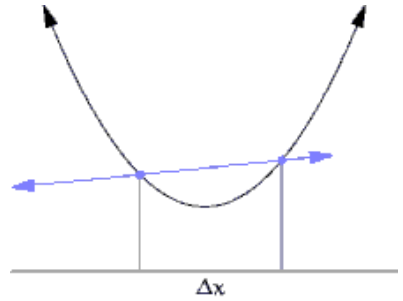
$$f'_{w_1}(5,3) = \mathbf{36}$$

$$f'_{w_2}(5,3) = \mathbf{10}$$

```
>>> w1, w2 = 5, 3
>>> eps = 1e-6
>>> (f(w1 + eps, w2) - f(w1, w2)) / eps
36.000003007075065
>>> (f(w1, w2 + eps) - f(w1, w2)) / eps
10.000000003174137
```

평균 변화율:

$$\frac{\Delta y}{\Delta x} = \frac{f(x + \Delta x) - f(x)}{\Delta x}$$



- 잘 작동 & 쉽게 구현
- 근사값 &
  파라미터 마다 f() 적어도 한번은 호출
  > **대규모 신경망에 적용 어려움**

12.3.8. 자동 미분을 사용하여 그레이디언트 계산하기

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

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

- 자동 미분 사용
- 변수 정의하고 tf.GradientTape 블록 생성
  > 관련된 모든 연산 자동 기록
- 이 tape에 변수에 대한 z의 그레이디언트 요청

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

12.3.8. 자동 미분을 사용하여 그레이디언트 계산하기

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

dz_dw1 = tape.gradient(z, w1) # => tensor 36.0
dz_dw2 = tape.gradient(z, w2) # RuntimeError!
```

- gradient() 메서드가 호출된 후에는 자동으로 테이프가 즉시 지워짐
  >> 두 번 호출하면 예외가 발생

- gradient() 메서드를 한번 이상 호출하려면 지속가능한 테이프를 만들어 주면 된다.
- 사용이 끝난 후 테이프를 삭제하여 리소스를 해야 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
```


12.3.8. 자동 미분을 사용하여 그레이디언트 계산하기

```
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]
```

- 테이프는 변수가 포함된 연산만을 기록
- 변수가 아닌 객체에 대한 z의 그레디언트를 계산하려면 None이 반환됨

- tape.watch() > 관련된 모든 연산을 기록하도록 강제
- 변수처럼 이런 텐서에 대해 그레이디언트를 계산할 수 있다.

```
with tf.GradientTape() as tape:
    tape.watch(c1)
    tape.watch(c2)
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2]) # returns [tensor 36., tensor 10.]
```


12.3.8. 자동 미분을 사용하여 그레이디언트 계산하기

```
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]
```

- 신경망의 일부분에 그레이디언트 역전파가 되지 않도록 막고싶다면, tf.stop_graident() 함수 사용

```
>>> x = tf.Variable([100.])
>>> with tf.GradientTape() as tape:
...     z = my_softplus(x)
...
>>> tape.gradient(z, [x])
<tf.Tensor: [...] numpy=array([nan], dtype=float32)>
```

- 가끔 그레이디언트 계산시 부동소수점 정밀도 오류로 인해 자동 미분이 무한 나누기 계산을 하게되어 NaN이 반환됨

- 다행이 수치적으로 안전한 소프트플러스의 도함수를 해석적으로 구할수 있음.@tf.custom_gradient 데코레이터를 사용하여 일반 출력과 도함수를 계산하는 함수를 반환하여 텐서플로가 my_softplus 함수의 그레이디언트를 계산할때 안전한 함수를 사용하도록 만듬

```
@tf.custom_gradient
def my_better_softplus(z):
    exp = tf.exp(z)
    def my_softplus_gradients(grad):
        return grad / (1 + 1 / exp)
    return tf.math.log(exp + 1), my_softplus_gradients
```