8. 머신러닝 모델부터 예측까지 기초 실습 2

KOREATECH 한국기술교육대학교

KOREATECH 능력개발교육원

CONTENTS

01

신경망 (Neural Network)

KOREATECH 한국기술교육대학교

KOREATECH 능력개발교육원

01 신경망 (Neural Network)

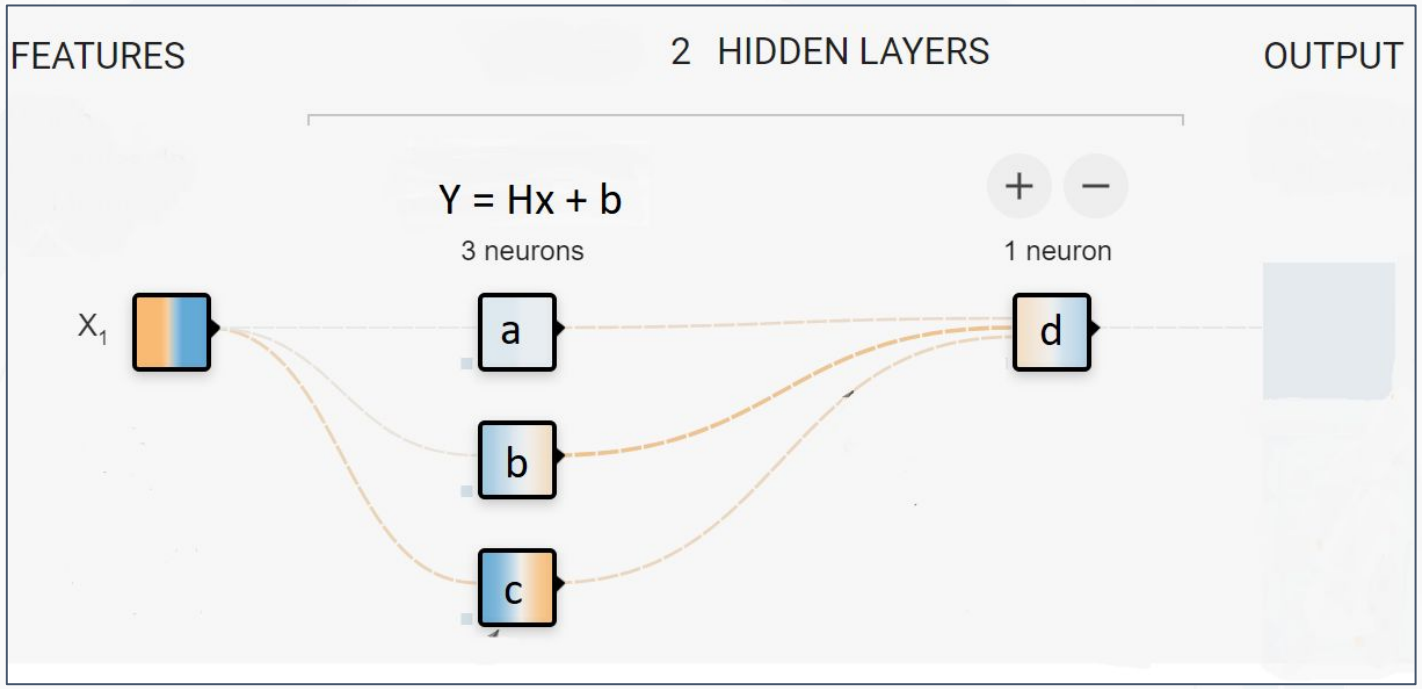
01-1 Dense Layers (화씨 섭씨)
실습

KOREATECH 한국기술교육대학교

KOREATECH 능력개발교육원

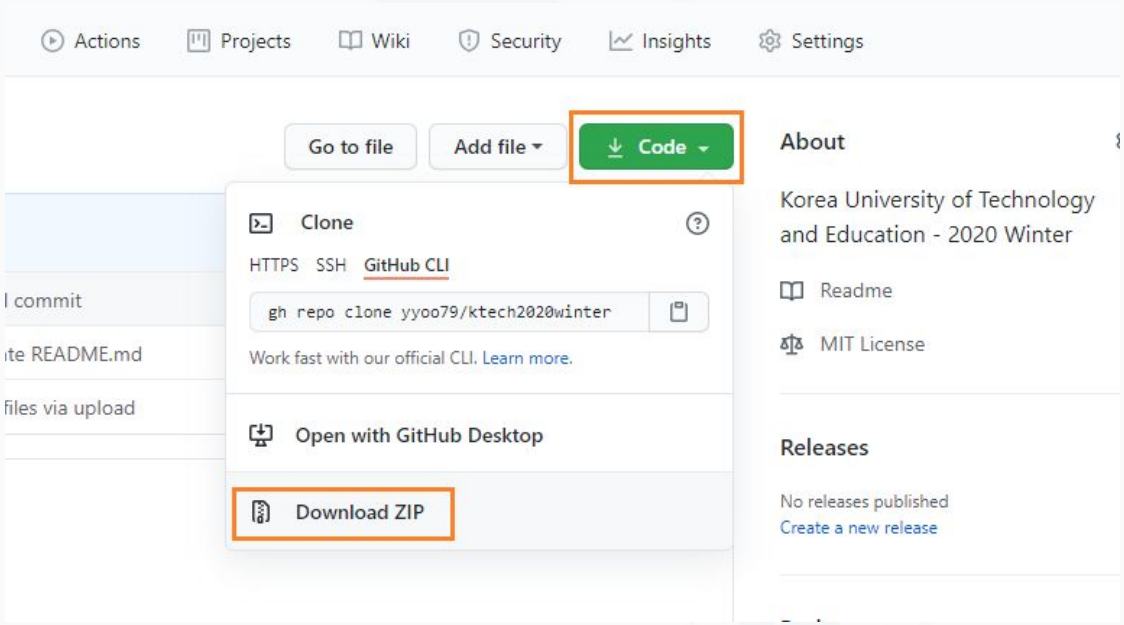
Activity

1. 각 a, b, c, d neuron이 되서 숫자를 추측해 내 보도록 하겠습니다.
2. neuron d는 output 값의 정보를 줄 수 없습니다.
3. neuron d는 한 뉴론에게는 맞다/아니다의 정보를 주어야 합니다.
4. neuron d는 다른 2 뉴론에게는 필요한/유익한 정보를 줄 수 있습니다.



파일 다운로드 받기

1. https://github.com/yyoo79/koeatech-2021-summer 로 이동
2. Code >> Download Zip



파일 코렙에서 열기

1. https://www.google.com/ 에서 구글계정으로 로그인
2. http://colab.research.google.com 로 이동
3. Upload tab을 클릭
4. 파일 - Module TF U2 화씨 섭씨 Celsius to Fahrenheit exercise.ipynb을 선택
5. 파일이름을 바꾸고 자신의 Google Drive로 저장해 놓습니다.

기초: 첫번째 신경망(Neural Network) 모델(model) 훈련

여기서 첫번째 머신러닝(Machine Learning) 신경망(Neural Network) 모델(model)을 학습하게 됩니다! 이번 실습에서는 간단하고 기본적인 개념(basic concept)만을 익혀 보도록 합니다.

우리가 해결할 문제는 섭씨에서 화씨로 변환하는 것입니다. 대략적인 공식은 다음과 같습니다:

$$f = c \times 1.8 + 32$$

앞에서 다루었 듯이 이 공식을 일반적인 파이썬 함수(function)으로 만드는 것은 간단하지만 기계 학습은 아닙니다.

TensorFlow에 몇 가지 샘플 섭씨 값 (0, 8, 15, 22, 38)과 해당 화씨 값 (32, 46, 59, 72, 100)을 만들고, 위의 공식을 훈련(train) 과정을 통해 파악하는 모델(model)을 만들어 볼것 있습니다.

디펜던시(dependencies) 패키지 불러오기(import)

먼저 TensorFlow를 가져옵니다. 여기서는 사용의 편의를 위해 'tf'라고 부릅니다.

TensorFlow 로그는 오류만 표시하도록 알려줍니다.

다음으로, 넘피(Numpy)를 'np'로 가져 오십시오. 넘피(Numpy)는 우리가 데이터를 성능이 높은(highly performance) 목록으로 표현할 수 있도록 도와줍니다.

```
from __future__ import absolute_import, division, print_function, unicode_literals
import tensorflow as tf
tf.logging.set_verbosity(tf.logging.ERROR)

import numpy as np
```

디펜던시(dependencies) 패키지 불러오기(import)

형광으로 표시된 import에 대해서 간단히 알아보도록 하겠습니다.

```
from __future__ import absolute_import, division, print_function, unicode_literals
import tensorflow as tf
tf.logging.set_verbosity(tf.logging.ERROR)

import numpy as np
```

디펜던시(dependencies) 패키지 불러오기(import)

from __future__ import absolute_import

string을 가져 오는 경우 Python은 항상 current_package.string이 아닌 최상위 string 모듈을 찾습니다. 그러나 파이썬이 string 모듈인 파일을 결정하는 데 크게 영향을 미치지 않습니다.

pkg/script.py는 Python에 패키지의 일부가 아닙니다.

```
from __future__ import absolute_import, division, print_function, unicode_literals
import tensorflow as tf
tf.logging.set_verbosity(tf.logging.ERROR)

import numpy as np
```

디펜던시(dependencies) 패키지 불러오기(import)

from __future__ import division

__future__ import division은 / 연산자를 변경하여 모듈 전반에 걸쳐 진정한 나눗셈을 의미합니다. 명령 줄 옵션은 int 또는 long 인수에 적용된 고전적인 구분에 대한 런타임 경고를 활성화합니다.

```
from __future__ import absolute_import, division, print_function, unicode_literals
import tensorflow as tf
tf.logging.set_verbosity(tf.logging.ERROR)

import numpy as np
```

디펜던시(dependencies) 패키지 불러오기(import)

from __future__ import division

예를 들어, a=100/23 (다음라인) print(a)를 하면 4 프린트되지만,

이 명령문이후에는 a=100/23 (다음라인) print(a)를 하면 4.3478… 이 프린트 됩니다.

```
from __future__ import absolute_import, division, print_function, unicode_literals
import tensorflow as tf
tf.logging.set_verbosity(tf.logging.ERROR)

import numpy as np
```

디펜던시(dependencies) 패키지 불러오기(import)

from __future__ import print_function

이렇게 하면 Python 버전 3.x 및 2.x에서의 호환성이 보장됩니다.

2.x에서는 print "x"가 허용되고, 3.x는 print ("x") 만 허용됩니다.

```
from __future__ import absolute_import, division, print_function, unicode_literals
import tensorflow as tf
tf.logging.set_verbosity(tf.logging.ERROR)

import numpy as np
```

디펜던시(dependencies) 패키지 불러오기(import)

from __future__ import unicode_literals

UTF-8로 인코딩 된 byte를 보유하는 byte string을 작성하게됩니다.

string과 함께 유니코드(unicode) string을 사용합니다.

print는이 두 값을 다르게 처리해야 하기 때문에 변경되지 않은 sys.stdout의 byte string을 사용하고, 유니 코드 문자열은 sys.stdout.encoding을 참조합니다. 기본값은 ASCII입니다.

```
from __future__ import absolute_import, division, print_function, unicode_literals
import tensorflow as tf
tf.logging.set_verbosity(tf.logging.ERROR)

import numpy as np
```

학습/트레이닝 데이터 셋업(Set up training data)

전에 보았듯이, 머신러닝의 지도학습은 입력과 출력의 집합이 주어진 알고리즘을 알아내는 것입니다.

여기에서 섭씨로 온도가 주어질 때 화씨로 온도를 줄 수있는 모델을 만듭니다.

```
[ ]  celsius_q    = np.array([-40, -10,  0,  8, 15, 22,  38],  dtype=float)
     fahrenheit_a = np.array([-40,  14, 32, 46, 59, 72, 100],  dtype=float)

     for i,c in enumerate(celsius_q):
       print("{} degrees Celsius = {} degrees Fahrenheit".format(c, fahrenheit_a[i]))
```

학습/트레이닝 데이터 셋업(Set up training data)

실행후 아래와 같이 프린트 됩니다.

```
[2]  celsius_q    = np.array([-40, -10,  0,  8, 15, 22,  38],  dtype=float)
     fahrenheit_a = np.array([-40,  14, 32, 46, 59, 72, 100],  dtype=float)

     for i,c in enumerate(celsius_q):
       print("{} degrees Celsius = {} degrees Fahrenheit".format(c, fahrenheit_a[i]))
```

```
-40.0 degrees Celsius = -40.0 degrees Fahrenheit
-10.0 degrees Celsius = 14.0 degrees Fahrenheit
0.0 degrees Celsius = 32.0 degrees Fahrenheit
8.0 degrees Celsius = 46.0 degrees Fahrenheit
15.0 degrees Celsius = 59.0 degrees Fahrenheit
22.0 degrees Celsius = 72.0 degrees Fahrenheit
38.0 degrees Celsius = 100.0 degrees Fahrenheit
```

머신러닝(Machine Learning) 용어

- 피쳐(Feature) – 모델의 입력입니다. 이 경우 단일 값 - 섭씨(celsius)
- 레이블(Labels) – 모델에서 예측 한 결과입니다. 이 경우 단일 값 - 화씨(fahrenheit)
- 예제(Example) – 교육 중에 사용되는 입력/출력 페어(pair). 여기서는 (22,72)와 같은 특정 인덱스(index)에서 celsius_q 및 fahrenheit_a의 값 pair을 사용합니다.

모델 만들기

이제 모델을 만듭니다. 가능한 가장 단순한 모델인 Dense 네트워크(network)를 사용할 것입니다. 이 화씨섭씨 문제는 간단하기 때문에 이 네트워크에는 단일 뉴런(neuron)이 있는 단일 레이어(layer)만 사용할 것 입니다.

```
l0 = tf.keras.layers.Dense(units=1, input_shape=[1])
```

레이어(layer) 만들기

레이어 l0(엘0) 이란 변수를 만들고 다음과 같은 설정으로tf.keras.layers.Dense를 인스턴스화 합니다:

```
l0 = tf.keras.layers.Dense(units=1, input_shape=[1])
```

레이어(layer) 만들기

input_shape = [1] - 이 레이어에 대한 입력이 단일 값임을 지정합니다. 즉, 모양은 멤버가 하나인 1 차원 배열입니다. 이것이 첫번째 (그리고 유일한) 레이어이기 때문에, 그 입력 모양은 전체 모델의 입력 모양입니다. 단일 값은 섭씨 온도를 나타내는 부동소수점(float point) 숫자입니다.

```
l0 = tf.keras.layers.Dense(units=1, input_shape=[1])
```

레이어(layer) 만들기

units = 1 - 레이어의 뉴런(neuron) 수를 지정합니다. 뉴런 수는 얼마나 많은 내부 변수(internal variables)가 문제를 해결할 방법을 배우고자 하는지를 정의합니다. 또한, 이 경우는 레이어가 하나이기 때문에 모델의 출력 크기(size)이며 화씨 온도를 나타내는 부동소수점(float) 값이기도 합니다. (다중 레이어(multi-layered) 네트워크에서 레이어의 크기와 모양은 다음 레이어의 'input_shape'와 일치해야합니다.)

```
l0 = tf.keras.layers.Dense(units=1, input_shape=[1])
```

레이어를 가지고 모델 만들기

레이어가 정의되면 모델을 레이어를 이용해 조립(assemble)합니다. Sequential 모델 정의는 레이어 리스트(list)을 인수(argument)로 사용하고, 입력에서 출력까지의 계산 순서를 지정합니다.

이 모델은 레이어, l0(엘0) 하나만 갖습니다.

```
model = tf.keras.Sequential([l0])
```

레이어/모델 노트

이 두가지를 동시에 할 수도 있습니다.

모델을 만들때, 레이어를 정의하는 코드를 흔히 볼 수 있을것입니다.

```
model = tf.keras.Sequential([
  tf.keras.layers.Dense(units=1, input_shape=[1])
])
```

손실(loss) 및 최적화(optimizer) 함수(functions) 사용하여 모델 컴파일하기

학습(training) 전에 모델을 컴파일 해야 합니다. 컴파일때 필요한 두가지는 아래와 같습니다:

- **손실 함수(Loss function)** — 예측 결과가 원하는 결과로부터 얼마나 떨어져 있는지 측정하는 방법. (측정 된 차이를 "손실"(loss)이라고합니다.)
- **최적화 함수(Optimizer function)** — 손실을 줄이기 위해 내부 값을 조정하는 방법입니다.

```
model.compile(loss='mean_squared_error',
              optimizer=tf.keras.optimizers.Adam(0.1))
```

손실(loss) 및 최적화(optimizer) 함수(functions) 사용하여 모델 컴파일하기

이 두개는 학습(training)중 사용되며 (다음에 나오는 model.fit()) 각 포인트(each point)에서 손실을 먼저 계산 한 다음 그것을 향상(improve) 시킵니다.

실제로 정확하게 이야기하면, 학습(training)이란 현재 손실을 계산하고 개선(improve)하는 것이라 할수 있습니다.

```
model.compile(loss='mean_squared_error',
              optimizer=tf.keras.optimizers.Adam(0.1))
```

손실(loss) 및 최적화(optimizer) 함수(functions) 사용하여 모델 컴파일하기

Training중, 옵티마이저 함수(optimizer function)는 모델의 내부변수(internal variables)를 조정/계산하는 데 사용됩니다. 여기서 목표는 모델 (정확하게는 그냥 수학공식)이 섭씨를 화씨로 변환하기 위한 실제 공식과 가까워 질때까지 **내부 변수(internal variables)를 조정**하는 것입니다.

```
model.compile(loss='mean_squared_error',
              optimizer=tf.keras.optimizers.Adam(0.1))
```

손실(loss) 및 최적화(optimizer) 함수(functions) 사용하여 모델 컴파일하기

여기에 사용 된 **손실함수(loss function)** - 평균 제곱 오차[**mean squared error**]와 **옵티마이저(optimizer)** - 아담(**Adam**)은 단순한 모델에 주로 사용되고 있지만 많은 다른 모델도 사용할 수 있습니다.

이러한 특정 기능이 어디에 어떻게 작동 하는지는 나중에 보도록 하겠습니다.

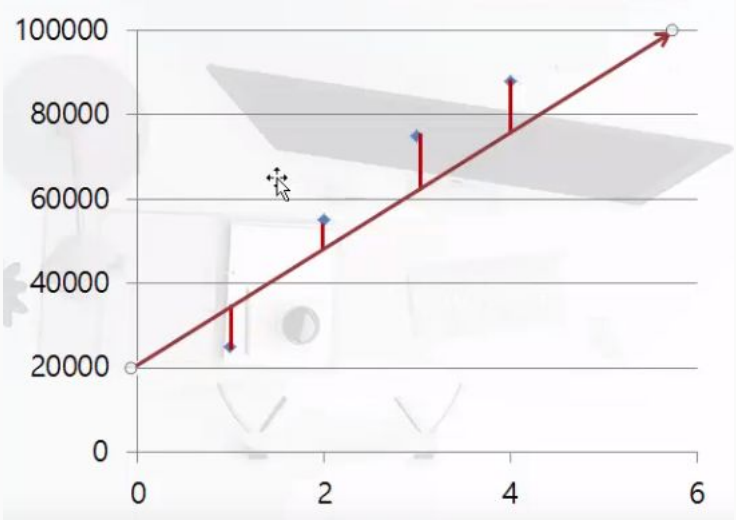
```
model.compile(loss='mean_squared_error',
              optimizer=tf.keras.optimizers.Adam(0.1))
```

손실(loss) 및 최적화(optimizer) 함수(functions) 사용하여 모델 컴파일하기

평균 제곱 오차**[mean squared error]**

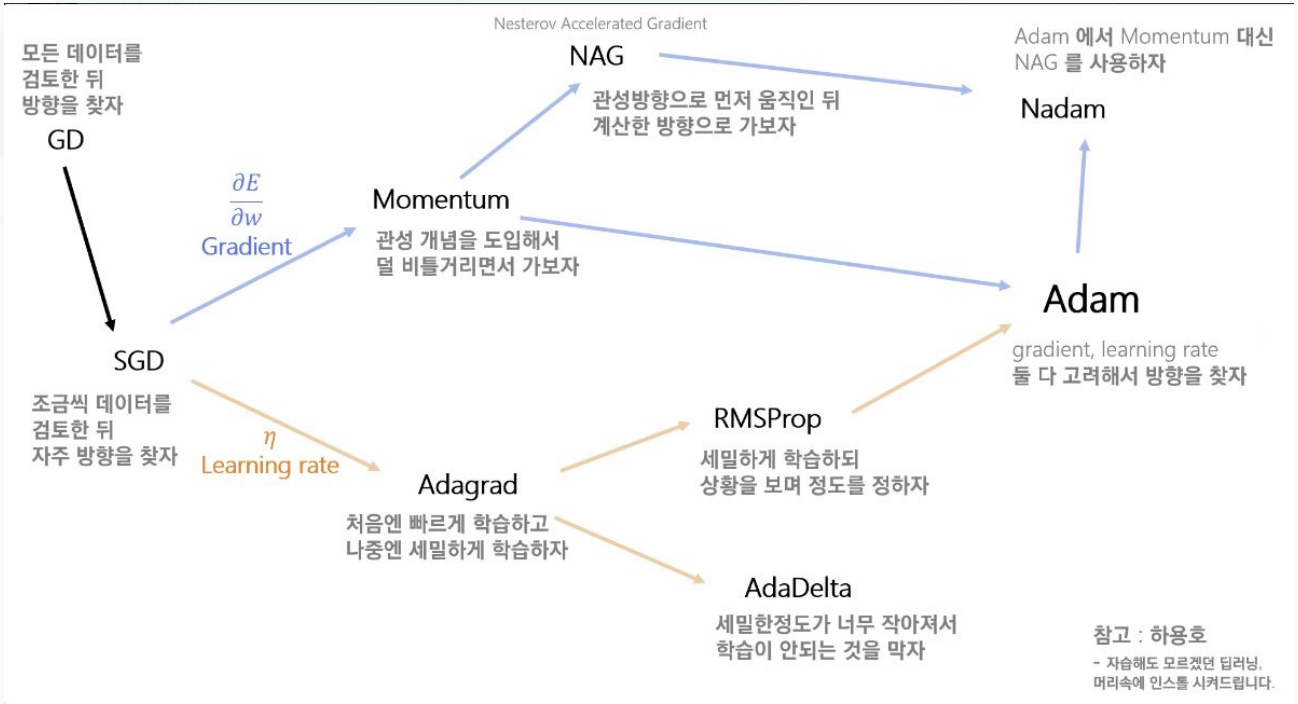
$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

$$cost(W,b) = \frac{1}{m}\sum_{i=1}^{m}(H(x^{(i)}) - y^{(i)})^2$$



손실(loss) 및 최적화(optimizer) 함수(functions) 사용하여 모델 컴파일하기

옵티마이저(optimizer) - 아담(Adam)



resource: https://gomguard.tistory.com/187

손실(loss) 및 최적화(optimizer) 함수(functions) 사용하여 모델 컴파일하기

하나 알아두면 좋은것은, 위의 코드에서 옵티마이저가 사용하는 **0.1**은 **학습 속도**라는 것입니다.

이 값은 모델에서 값을 조정할 때 사용되는 **단계 크기(step size)**입니다.

값이 너무 작으면 모델을 학습하는 데 너무 많은 반복이 필요하고, 너무 크면 정확도가 떨어집니다. 정확한 값을 찾으려면 이런저런 값을 넣어서 계속 돌려봐야 하며, 보통 범위는 0.001(default)에서 0.1사이 입니다.

```
model.compile(loss='mean_squared_error',
              optimizer=tf.keras.optimizers.Adam(0.1))
```

모델 학습(train)하기

`fit` 메소드를 이용해서 모델을 트레이닝 합니다.

```
history = model.fit(celsius_q, fahrenheit_a, epochs=500, verbose=False)
print("Finished training the model")
```

모델 학습(train)하기

training중 모델은 섭씨 값을 받고, 현재 내부변수(internal variable) - **가중치(weight)**를 사용하여 **계산**하고 화씨에 해당하는 값을 출력(output)합니다.

가중치(weight)는 **초기**에 **무작위**로 설정됩니다. 따라서, 출력은 정확한 값에 근접하지 않습니다. 실제 출력(actual weight)과 원하는 출력(desired weight)간의 차이는 손실함수(loss function)를 사용하여 계산되며 옵티마이저함수(optimizer function)은 가중치를 어떻게 조정할지 결정합니다.

```
history = model.fit(celsius_q, fahrenheit_a, epochs=500, verbose=False)
print("Finished training the model")
```

모델 학습(train)하기

계산, 비교, 조정하는 **사이클(cycle)**은 `fit` 메소드가 콘트롤합니다.

첫 번째 인수는 입력(inputs)이고, **두 번째 인수**는 원하는 출력(desired output)입니다.

에포크**`epochs` 인수**는 사이클를 몇번 실행해야하는지 지정하고 **verbose 인수**는 메소드가 생성하는 출력의 양을 정합니다.

```
history = model.fit(celsius_q, fahrenheit_a, epochs=500, verbose=False)
print("Finished training the model")
```

모델 학습(train)하기

다음 모듈에서 더 자세한 내용과 Dense 레이어가 실제로 내부적으로 어떻게 작동하는지에 대해 알아 보기로 합니다 .

print()는 training이 끝났음을 확인해 주기도 합니다. (처음에 tf log verbose를 에러만 표시하기로 정했지 때문입니다.)

```
history = model.fit(celsius_q, fahrenheit_a, epochs=500, verbose=False)
print("Finished training the model")
```

training 통계 보기

`fit` 메소드는 **history** 객체(**object**)를 반환(**return**)합니다. 이 object를 사용하여 각 **training epoch** 후에 모델의 손실이 어떻게 감소하는지 플랏(plot) 할 수 있습니다. 손실이 많다는 것은 모델이 예측하는 화씨 온도가 `fahrenheit_a`의 해당 값과 멀리 떨어져 있음을 의미합니다.

```
import matplotlib.pyplot as plt
plt.xlabel('Epoch Number')
plt.ylabel("Loss Magnitude")
plt.plot(history.history['loss'])
```

training 통계 보기

`fit` 메소드는 **history** 객체(**object**)를 반환(**return**)합니다. 이 object를 사용하여 각 **training epoch** 후에 모델의 손실이 어떻게 감소하는지 플랏(plot) 할 수 있습니다. 손실이 많다는 것은 모델이 예측하는 화씨 온도가 `fahrenheit_a`의 해당 값과 멀리 떨어져 있음을 의미합니다.

```
import matplotlib.pyplot as plt
plt.xlabel('Epoch Number')
plt.ylabel("Loss Magnitude")
plt.plot(history.history['loss'])
```

training 통계 보기

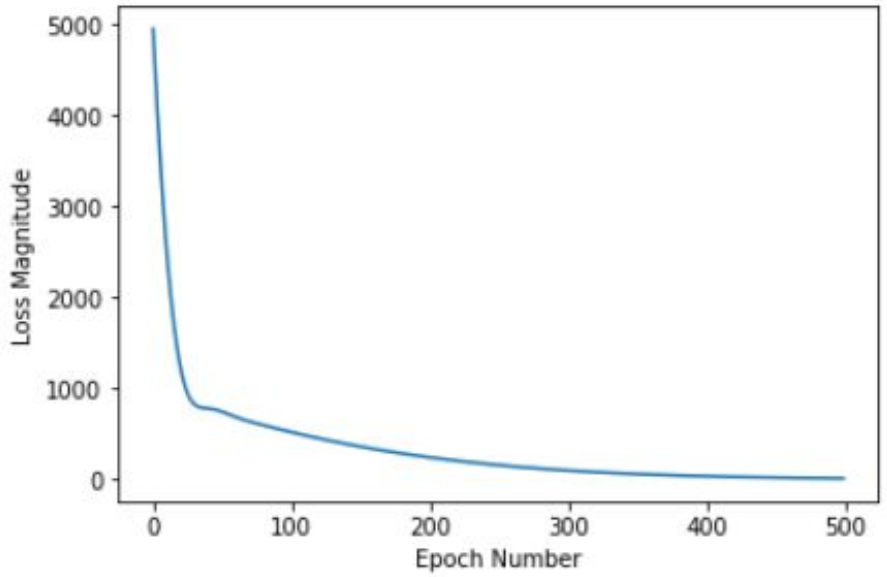
여기서는 **Matplotlib**을 사용하여 시각화 합니다 (다른 도구를 사용할 수도 있음). 보시다시피, 이 모델은 처음에는 매우 빨리 향상되고, 끝부분에는 "완벽(perfect)"해 질때까지 꾸준이(steady) 천천히 값을 맞춰나갑니다.

```
import matplotlib.pyplot as plt
plt.xlabel('Epoch Number')
plt.ylabel("Loss Magnitude")
plt.plot(history.history['loss'])
```

training 통계 보기

```
import matplotlib.pyplot as plt
plt.xlabel('Epoch Number')
plt.ylabel("Loss Magnitude")
plt.plot(history.history['loss'])
```

```
[<matplotlib.lines.Line2D at 0x7f0b8cfde9e8>]
```



모델을 사용하여 값 예측하기

이제`celsius_q`와`fahrenheit_a` 사이의 관계를 배우도록 훈련 된 모델이 있습니다. `predict()` 메소드를 사용하여 예측하면 이전에 몰랐던 섭씨도에 대한 화씨도를 계산할 수 있습니다.

예를 들어 섭씨 값이 100 인 경우 화씨 결과는 어떻게 될까요? 이 코드를 실행하기 전에 추측해 보십시오.

모델을 사용하여 값 예측하기

정답은 100×1.8+32=212 입니다. 우리가 만들 모델이 잘 작동하는걸 확인할수 있습니다.

```
print(model.predict([100.0]))
```

```
[[211.30298]]
```

검토(Review)

- 우리는 Dense 레이어가 있는 모델을 만들었습니다.
- 우리는 3500가지의 예(7 pairs, 500번 넘게 반복으로 돌림)로 학습 했습니다.

이 모델은 섭씨 값에 대해 정확한 화씨 값을 출력 할 수있을 때까지 dense layer의 변수(가중치)를 조정(tuning)했습니다. (섭씨 100은 우리의 training 데이터의 일부가 아니었습니다.)

```
print(model.predict([100.0]))
```

```
[[211.30298]]
```

레이어(layer) 가중치(weight) 살펴보기

마지막으로 Dense layer의 내부변수(internal variable)를 print해 봅시다.

```
print("These are the layer variables: {}".format(l0.get_weights()))
```

레이어(layer) 가중치(weight) 살펴보기

첫 번째 변수는 ~1.8에 가까우며 두 번째 변수는 ~32에 가까움. 이 값 (1.8 및 32)은 실제 변환 공식의 실제(actual) 변수(variable)입니다.

```
print("These are the layer variables: {}".format(l0.get_weights()))
```

```
These are the layer variables: [array([[1.8252511]], dtype=float32), array([28.777864], dtype=float32)]
```

레이어(layer) 가중치(weight) 살펴보기

이 값은 변환 공식의 값에 아주 가깝습니다. 다음 모듈에서 Dense layer가 어떻게 작동하는지 설명하겠지만, 하나의 입력과 하나의 출력을 가진 단 하나의 뉴런의 경우, 내부(internal) 수학공식은 y=mx+b 에 대한 방정식과 동일하게 보입니다.

마치 변환 공식 f=1.8c+32 와 같다는걸 알수 있습니다.

```
print("These are the layer variables: {}".format(l0.get_weights()))
```

```
These are the layer variables: [array([[1.8252511]], dtype=float32), array([28.777864], dtype=float32)]
```

레이어(layer) 가중치(weight) 살펴보기

형식(form)이 같기 때문에 변수는 1.8과 32의 표준(standard)값을 커버할수(coverage) 있습니다.
이것이 안에서 정확히 무슨 일이 일어 났는지 알수 있는 증거가 됩니다.

추가 뉴런, 추가 입력 및 추가 출력이 생기면 수식은 훨씬 더 복잡해 지지만 아이디어는 같습니다.

```
print("These are the layer variables: {}".format(l0.get_weights()))
```

```
These are the layer variables: [array([[1.8252511]], dtype=float32), array([28.777864], dtype=float32)]
```

작은 실험 - 레이어 더하기

단위(unit)가 다른 여러개의 Dense layer를 만들어 더하면 변수(variable)는 어떻게 달라지고 계산은 어떻게 달라지는 실험해 보도록 해보겠습니다.

hint: 변수는 더 많아 집니다.

```
l0 = tf.keras.layers.Dense(units=4, input_shape=[1])
l1 = tf.keras.layers.Dense(units=4)
l2 = tf.keras.layers.Dense(units=1)
model = tf.keras.Sequential([l0, l1, l2])
model.compile(loss='mean_squared_error', optimizer=tf.keras.optimizers.Adam(0.1))
model.fit(celsius_q, fahrenheit_a, epochs=500, verbose=False)
print("Finished training the model")
print(model.predict([100.0]))
print("Model predicts that 100 degrees Celsius is: {} degrees Fahrenheit".format(model.predict([100.0])))
print("These are the l0 variables: {}".format(l0.get_weights()))
print("These are the l1 variables: {}".format(l1.get_weights()))
print("These are the l2 variables: {}".format(l2.get_weights()))
```

작은 실험 - 레이어 더하기

code snippet: (아래 코드를 복사/붙여넣기해서 돌려봅니다)

```
l0 = tf.keras.layers.Dense(units=4, input_shape=[1])
l1 = tf.keras.layers.Dense(units=4)
l2 = tf.keras.layers.Dense(units=1)
model = tf.keras.Sequential([l0, l1, l2])
model.compile(loss='mean_squared_error', optimizer=tf.keras.optimizers.Adam(0.1))
model.fit(celsius_q, fahrenheit_a, epochs=500, verbose=False)
print("Finished training the model")
print(model.predict([100.0]))
print("Model predicts that 100 degrees Celsius is: {} degrees
Fahrenheit".format(model.predict([100.0])))
print("These are the l0 variables: {}".format(l0.get_weights()))
print("These are the l1 variables: {}".format(l1.get_weights()))
print("These are the l2 variables: {}".format(l2.get_weights()))
```

작은 실험 - 레이어 더하기

Code snippet의 output을 봤을때, 모델은 해당 화씨 값을 실제로 잘 예측할 수 있다는 것을 알 수 있습니다. 그러나 l0 과 l0 레이어의 변수 (가중치)를 보면 ~ 1.8과 32에 가까운 것조차 없습니다. 추가 된 layer들은 변환 방정식을 복잡하게 만들면서 "단순한" 형태로 알아볼수 없게 됩니다.

```
Finished training the model
[[211.74742]]
Model predicts that 100 degrees Celsius is: [[211.74742]] degrees Fahrenheit
These are the l0 variables: [array([[ 0.1752512 , -0.99963695, -0.24394271, -0.2333549 ]],
      dtype=float32), array([-1.7598908, -3.6002858,  2.5424023, -3.1726222], dtype=float32)]
These are the l1 variables: [array([[ 0.56879264, -0.2757305 , -0.55458635, -0.01638995],
       [ 0.2742892 ,  0.04785438, -0.06205853, -1.2118344 ],
       [ 0.03723637, -0.25656947,  0.12342595,  1.1168426 ],
       [ 1.2436881 ,  0.21936058, -0.757944  ,  0.0558406 ]],
      dtype=float32), array([-3.5458465, -2.2659526,  3.4215868,  3.4653406], dtype=float32)]
These are the l2 variables: [array([[-0.9190079 ],
       [-0.19857241],
       [ 0.64798874],
       [ 1.379522  ]], dtype=float32), array([3.4039733], dtype=float32)]
```

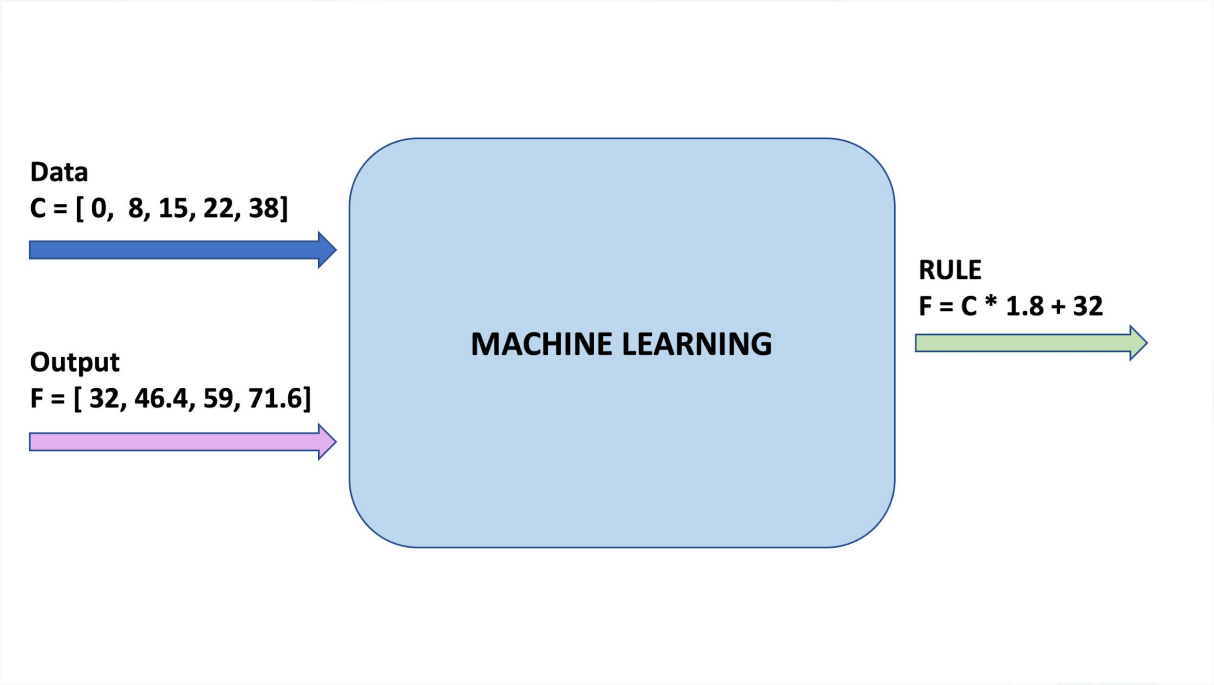
01 신경망 (Neural Network)

01-2 요약 (Recap) 및 정리

KOREATECH 한국기술교육대학교

KOREATECH HRDI 능력개발교육원

신경망 머신러닝 모델 만드는 방법을 배워 봤습니다. 입력 데이터와 해당 출력을 사용하여 모델을 교육하고 입력을 1.8 배로 늘린 다음 더 정확한 결과를 얻기 위해 32를 더하는 방법을 배웠습니다.



요약 Intro

레이어, 모델, 컴파일, 훈련 및 **예측**등 이 모든것을 몇줄 안되는 코드로 만들수 있었습니다.

정리하면 아래와 같습니다.

```
l0 = tf.keras.layers.Dense(units=1, input_shape=[1])
model = tf.keras.Sequential([l0])
model.compile(loss='mean_squared_error', optimizer=tf.keras.optimizers.Adam(0.1
history = model.fit(celsius_q, fahrenheit_a, epochs=500, verbose=False)
model.predict([100.0])
```

요약 Intro

이 예제는 모든 머신러닝 프로그램의 **일반적인 플랜(general plan)**이라고 볼수 있습니다. **동일한 구조(structure)**를 사용하여 신경망을 만들고 학습하고 예측을 하기 위해 사용합니다.

```
l0 = tf.keras.layers.Dense(units=1, input_shape=[1])
model = tf.keras.Sequential([l0])
model.compile(loss='mean_squared_error', optimizer=tf.keras.optimizers.Adam(0.1
history = model.fit(celsius_q, fahrenheit_a, epochs=500, verbose=False)
model.predict([100.0])
```

Training Process (학습과정)

model.fit(...) - 학습과정은 실제로 네트워크의 내부 변수를 가능한 최상의 값으로 **조정(tuning)**하고 입력을 출력에 **매핑(mapping)**하는 것입니다.

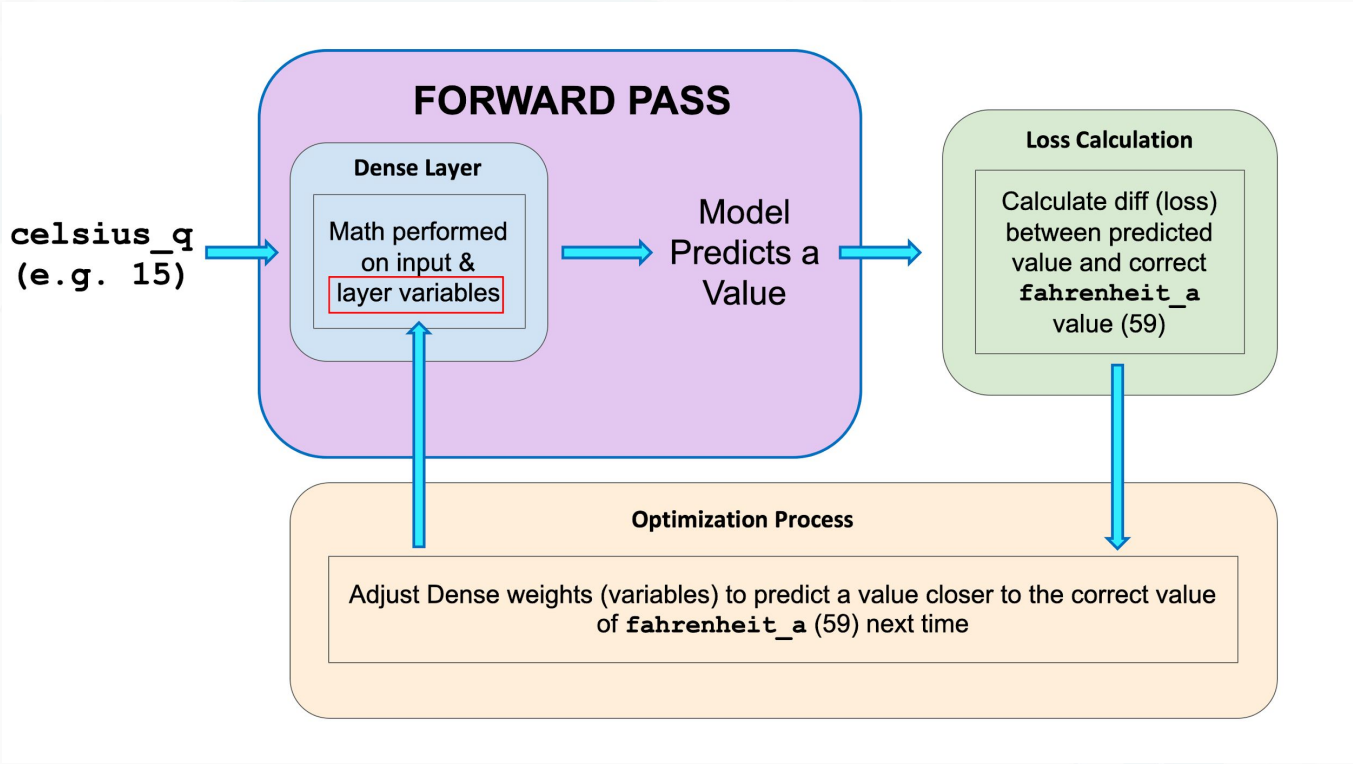
- Gradient Descent라는 최적화 프로세스(optimization process)를 통해 이루어지며,
- Numeric Analysis를 사용하여 모델의 내부 변수에 대해 가능한 최상의 값을 찾습니다.

Training Process (학습과정)

머신러닝 측면에서 볼때 이 부분은 아주 자세하게 이해할 필요가 없지만, **그라디언트 디센트(gradient descent, 혹은 구배 강하)**에 대해 간단히 배워 보도록 합니다.

- 그라디언트 디센트는 매개 변수를 반복적으로 조정하여 “최상의 가치”(best values)에 도달 할 때까지 한 번에 조금씩 **올바른 방향**으로 **전환**
- 하지만, 모델이 "최상의 가치"(best values)에 **도달한 후 더 방향을 틀면** 모델의 **성능**이 **저하** 될 수 있습니다. 반복되는 동안 모델이 얼마나 좋은지 또는 나쁜지를 측정하는 기능을 "손실 기능" 이라고 하며 **매번 기울기의 목표**는 "**손실 기능 최소화**"입니다.

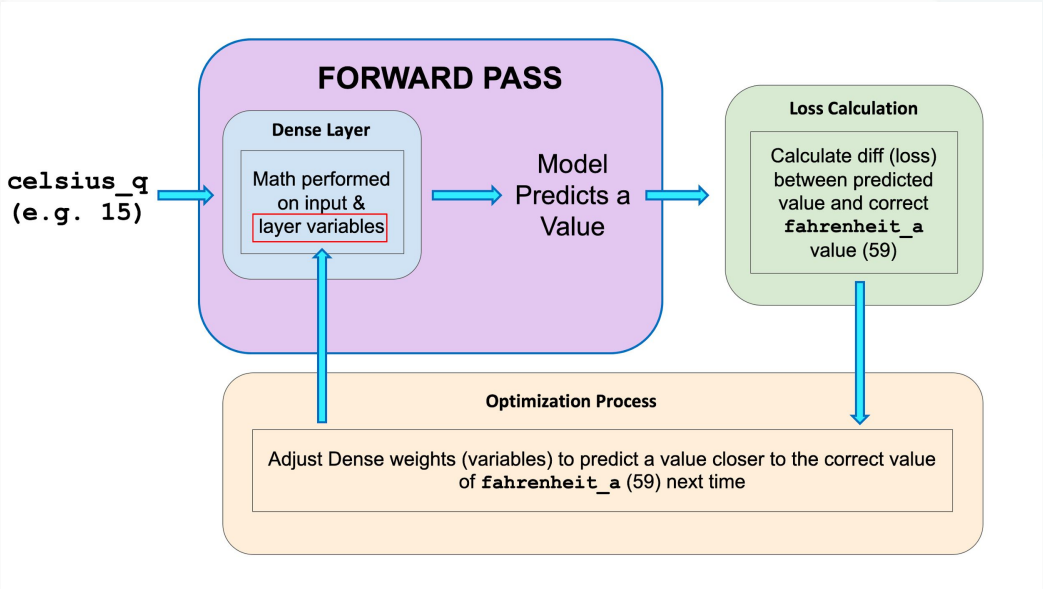
포워드 패스(Forward Pass)



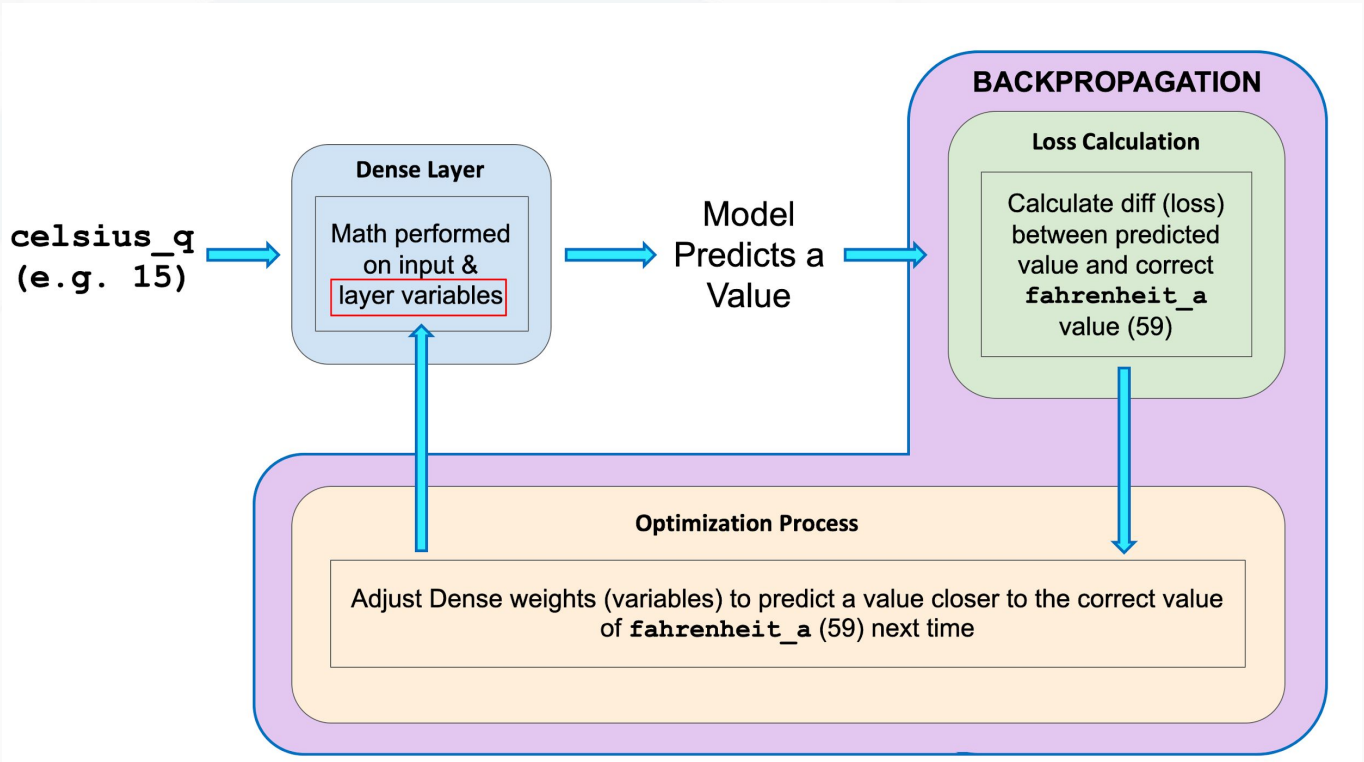
포워드 패스(Forward Pass)

훈련과정(training process)은 입력 데이터가 신경망에 공급되는 순방향 통과(forward pass)로 시작됩니다. 그런 다음 모델은 내부 수학을 입력 및 내부 변수에 적용하여 해답을 예측합니다. (그림의 "모델은 값을 예측합니다(Model Predicts a Value)")

이 예에서 입력 값은 섭씨로 표시되었으며, 모델은 해당 화씨를 화씨로 예측 했습니다.

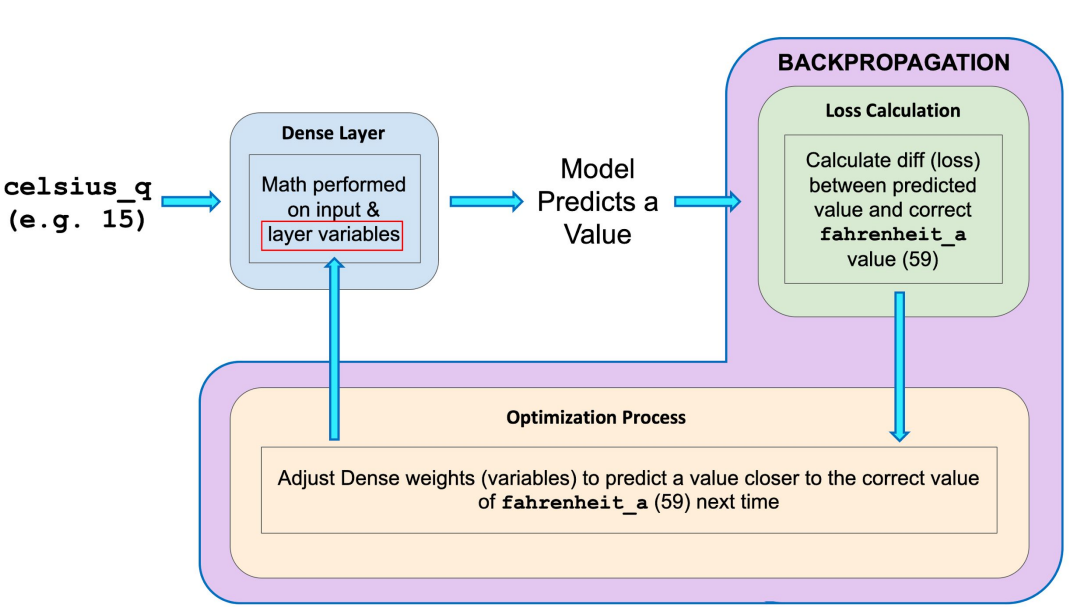


역전파(Backpropagation)



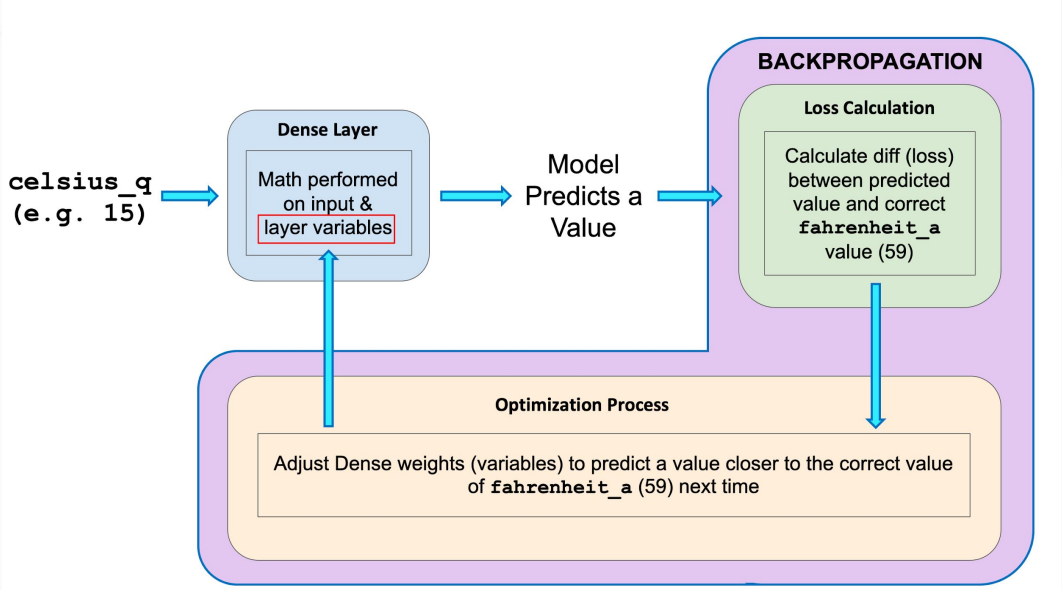
역전파(Backpropagation)

1. 값이 예측되면 해당 예측 값과 올바른 값 간의 차이가 계산
2. 이 차이가 손실(loss)이며 모델이 매핑 작업을 얼마나 잘 되었는지 측정
3. 손실 값은 `model.compile()`을 call할 때 손실 매개 변수(loss parameter)로 지정한 손실 함수(loss function)를 사용하여 계산



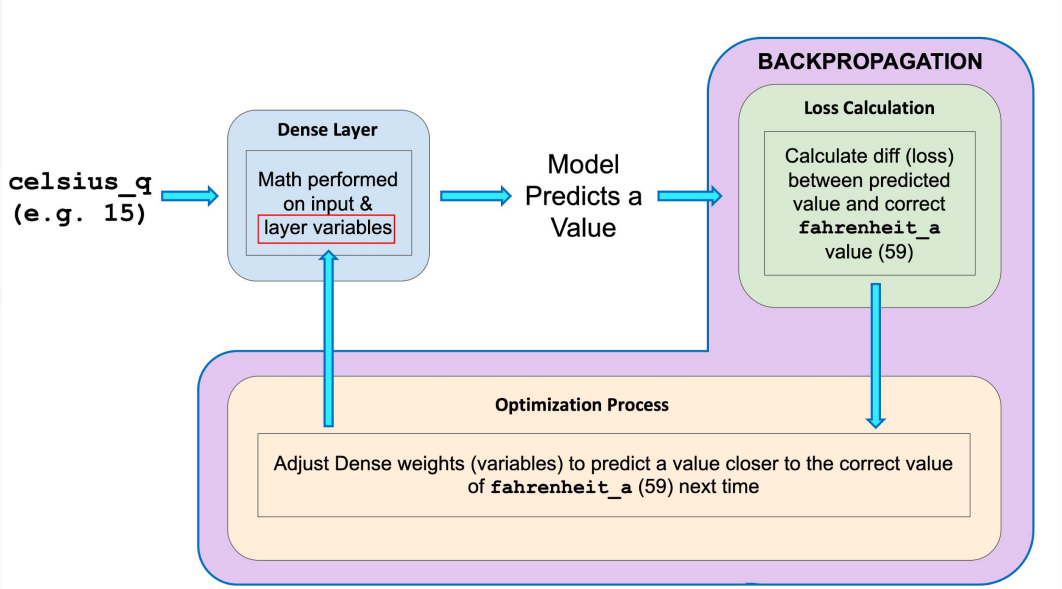
역전파(Backpropagation)

4. 손실이 계산 된 후, 신경망의 모든 계층(layer)의 내부 변수 (**가중치[weights]**와 **바이어스[biases]**)가 조정되어이 손실을 최소화
5. 즉 출력 값을 올바른 값에 더 가깝게



역전파(Backpropagation)

- 이 최적화 과정(optimization process) = **Gradient Descent**
- 각 내부 변수의 새로운 값을 계산하는 데 사용되는 특정 알고리즘은 **model.compile** (...)을 call 할 때 optimizer 매개 변수에 의해 지정
- 이 예제에서는 **Adam** 최적화 도구를 사용



노트 - reducing loss에 관한 구글 머신러닝 크래쉬 코스(영어) - [lesson on reducing loss in Google's machine learning crash course]

용어 정리

- **특징(Feature)**: 모델에 들어가는 입력(input)
- **예(Examples)**: training에 사용되는 입력(input) / 출력(output) 페어(pair)
- **레이블(Label)**: 모델의 출력
- **레이어(Layer)**: 신경망(neural network)내에서 서로 연결된 노드 collection
- **모델(Model)**: 신경망의 레프리젠데이션(representation)
- **Dense와 완전 연결(Fully Connected, FC)**: 한 계층(layer)의 각 노드(node)는 이전 계층의 각 노드와 연결
- **가중치(Weight)과 편향(bias)**: 모델의 내부 변수

용어 정리

- **손실(Loss)**: 원하는 출력과 실제 출력 간의 차이
- **MSE(Mean Squared Error)**: 평균 제곱 오차. 많은 수의 작은 오차보다 작은 불일치를 계산하는 손실 함수 유형
- **경사하강법(Gradient Descent)**: 손실 함수를 점차적으로 줄이기 위해 내부 변수를 한 번에 조금씩 변경하는 알고리즘
- **최적화(Optimizer)**: 그라디언트 디센트 알고리즘을 구현하는 것 중 하나입니다. (이것을 구현하는 방법에는 많은 알고리즘이 있으며, 이 과정에서는 ADAptive(어댑티브)와 Momentum(모멘텀)를 나타내는 "Adam"Optimizer 만 사용합니다. 이 알고리즘은 가장 많이 사용되는 최적화 도구입니다.)

용어 정리

- **학습률(Learning rate)**: 기울기 강하 중 손실 개선을위한 "단계 크기"
- **배치(Batch)**: 신경 회로망을 훈련하는 동안 사용 된 일련의 예제
- **이팍(Epoch)**: 전체 교육 데이터 세트에 대한 전체 전달
- **포워드 패스(Forward pass)**: 입력을 사용해서 나온 출력 계산값
- **백워드 패스 - Backward pass (backpropagation)**: 출력 layer 에서 시작하여 각 계층을 통해 입력(input)으로 되돌아가는 최적화(Optimizer) 알고리즘에 따라 내부 변수(internal variable) 조정을 계산

01 신경망 (Neural Network)

01-3 덴스 레이어 (Dense Layer)

KOREATECH 한국기술교육대학교

KOREATECH HRDI 능력개발교육원

신경망 - 실습 (Neural Network)

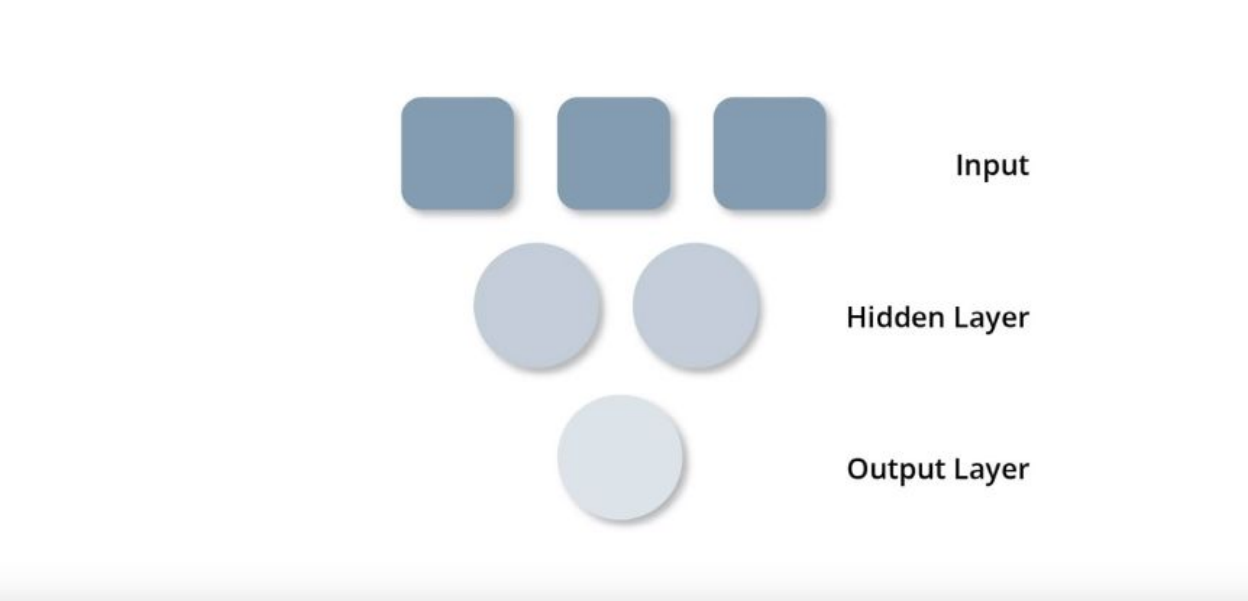
이전에 우리는 하나의 **Dense layer**로 이루어진 **신경망**을 만들어서 섭씨를 화씨로 변환시키는 **모델**을 만들었습니다. 이제 Dense layer에 대해 좀더 알아 보도록 합니다.

```
[ ]  l0 = tf.keras.layers.Dense(units=1, input_shape=[1])
```

```
[ ]  model = tf.keras.Sequential([l0])
```

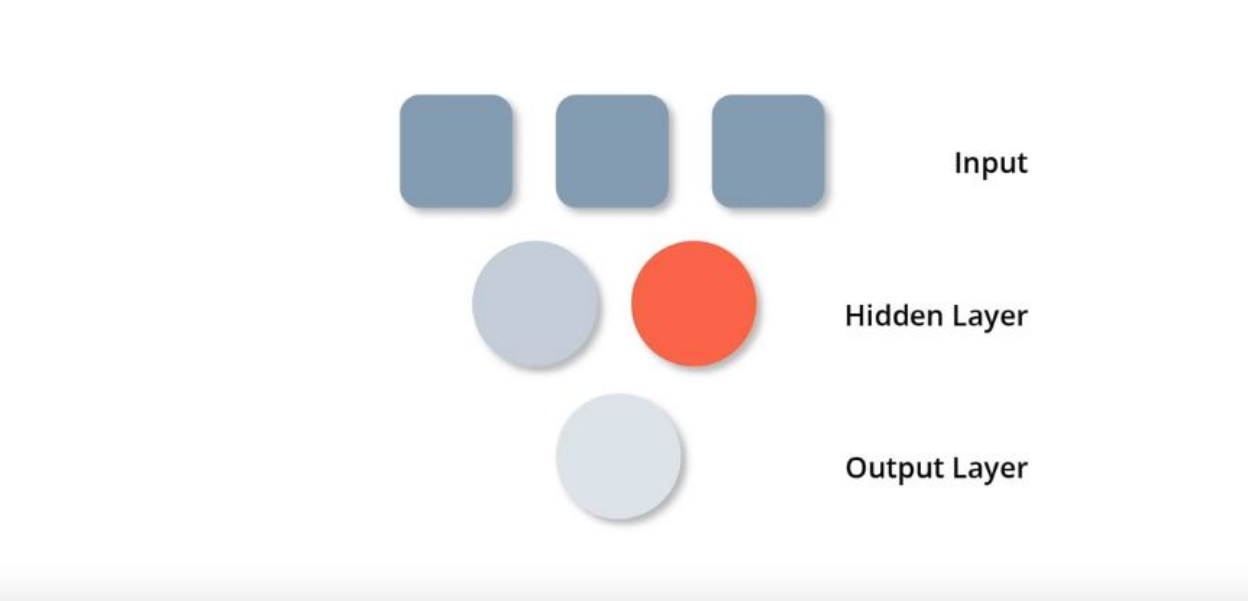
신경망 - 실습 (Neural Network)

Dense Layer를 이해하기 위해 좀더 **복잡한 신경망**을 만들어 봅니다. 이 모델은 **3개의 input**과 **1개의 hidden layer**와 **출력 layer**로 돼 있으며 출력은 한개의 유닛으로 되있습니다. 각 layer는 유닛으로 이루어져 있습니다.



이 유닛를 **뉴런**이라고 합니다.

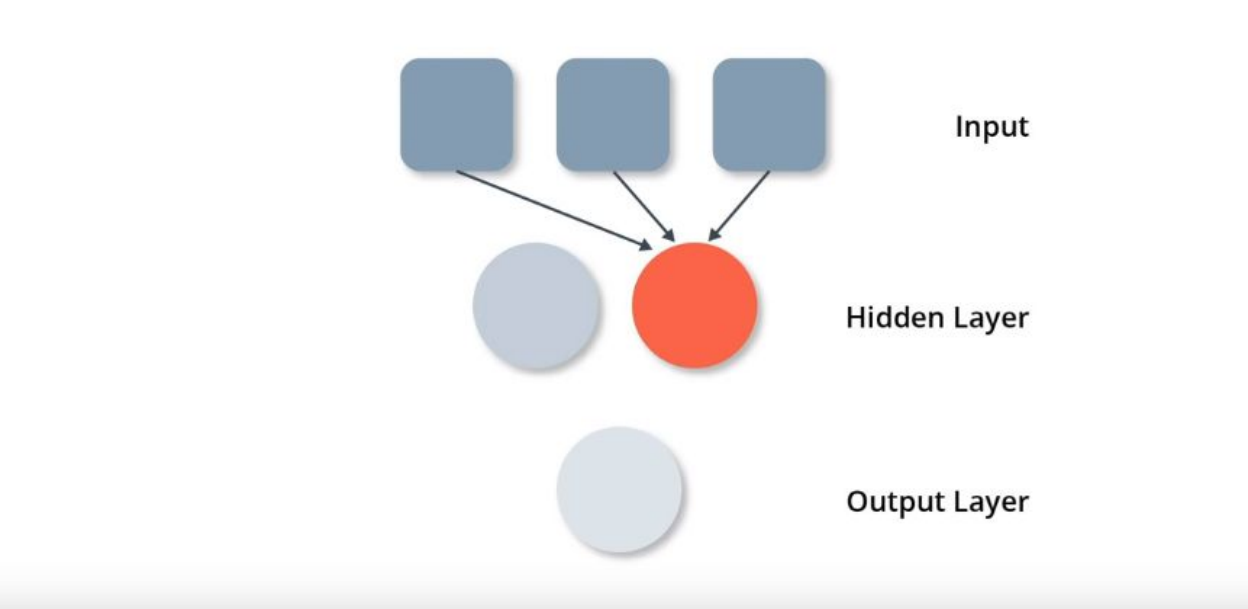
각 계층의 뉴런은 다음 계층의 뉴런에 **연결**될 수 있습니다.



신경망 - 실습 (Neural Network)

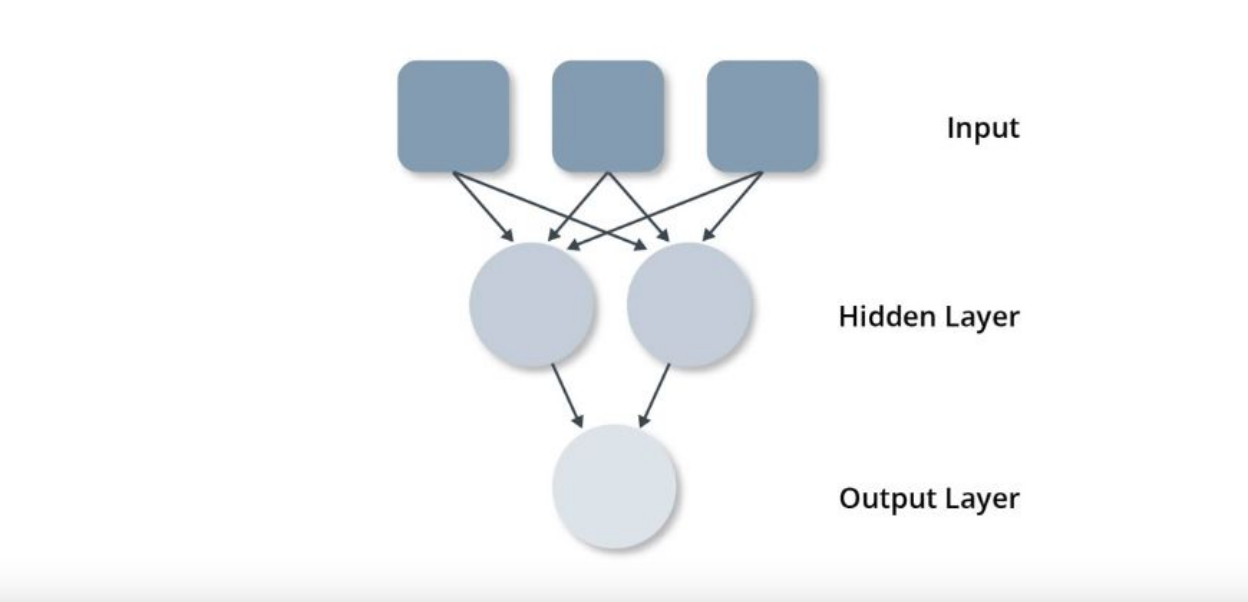
숨겨진 레이어의 뉴런이 모든 입력에서 데이터를 받습니다.

다른 뉴런들도 같은 방식으로 입력을 받게 됩니다.



완전히 연결된(Fully-Connected) / Dense Layer

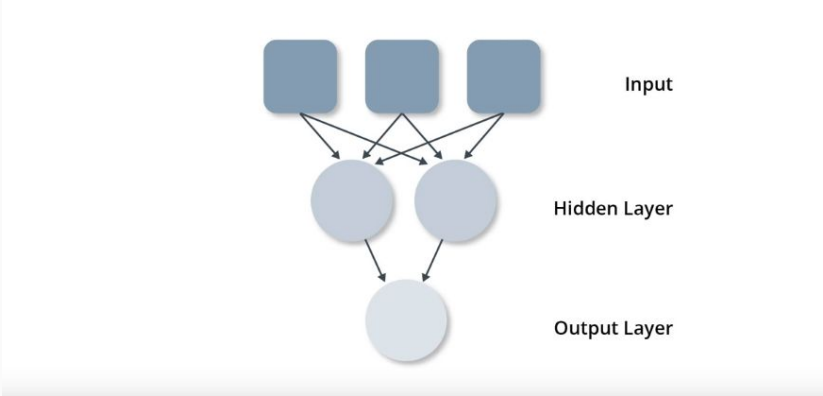
각 층의 모든 뉴런이 이전 층의 모든 뉴런과 연결이 되어 있는 경우



keras에서 dense layer를 사용해서 모델을 만들면, 각 레이어들이 완전히 연결됩니다.

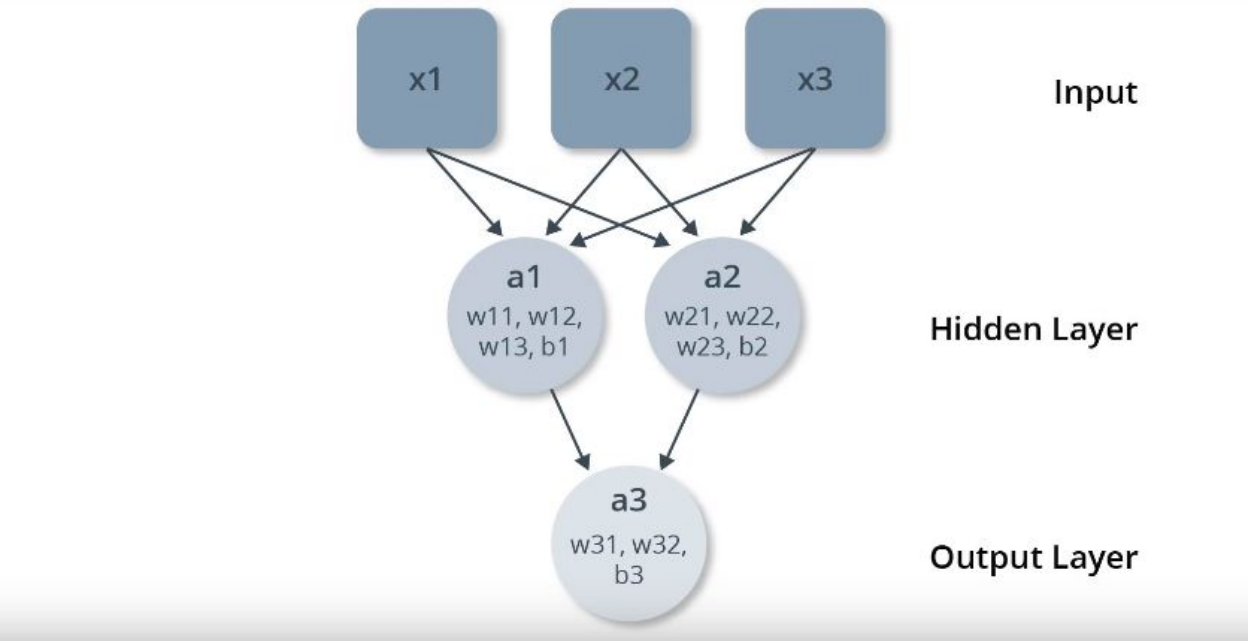
코드는 아래와 같습니다.

```
hidden = keras.layers.Dense(units=2, input_shape=[3])
output = keras.layers.Dense(units=1)
model = tf.keras.Sequential( [ hidden, output ] )
```



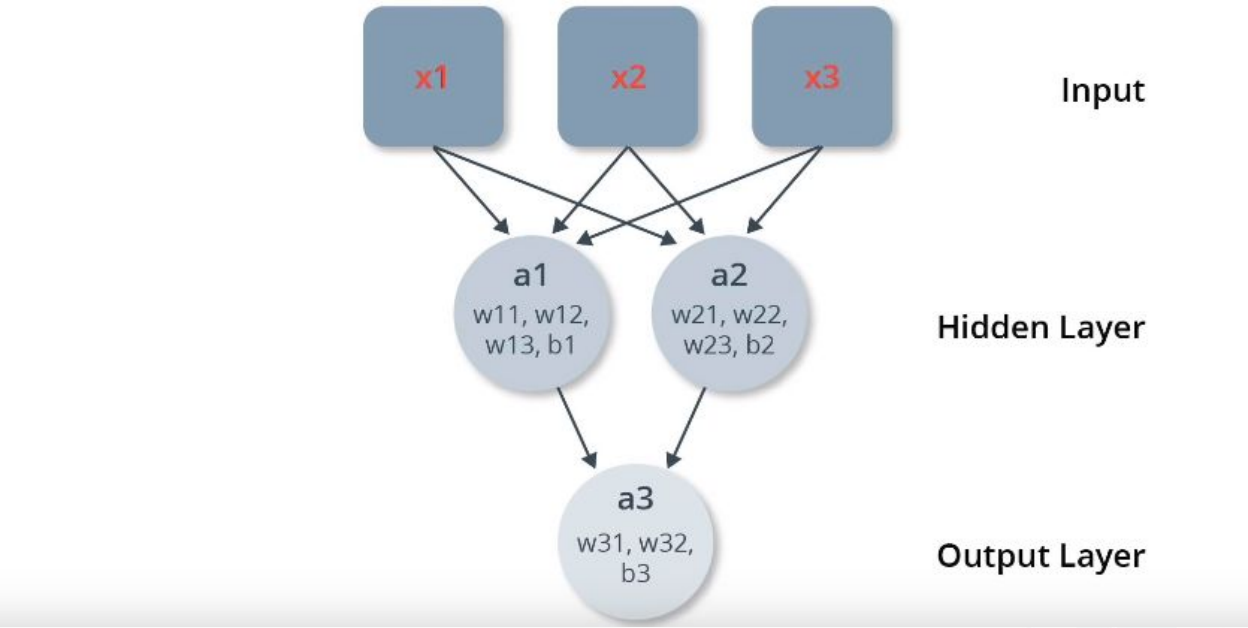
Dense Layer 계산

맵(mapping)되어 있는 내부변수(internal variables)



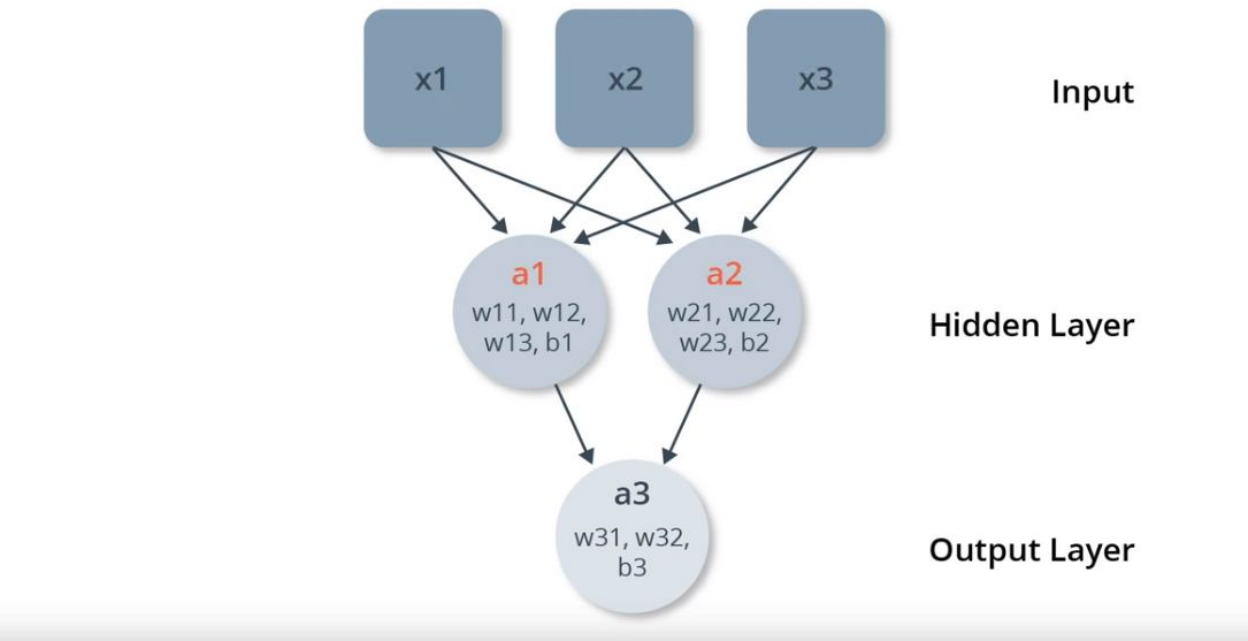
내부변수(Internal Variables)

학습과정중 입력값과 출력값을 근접하게 조정됨.

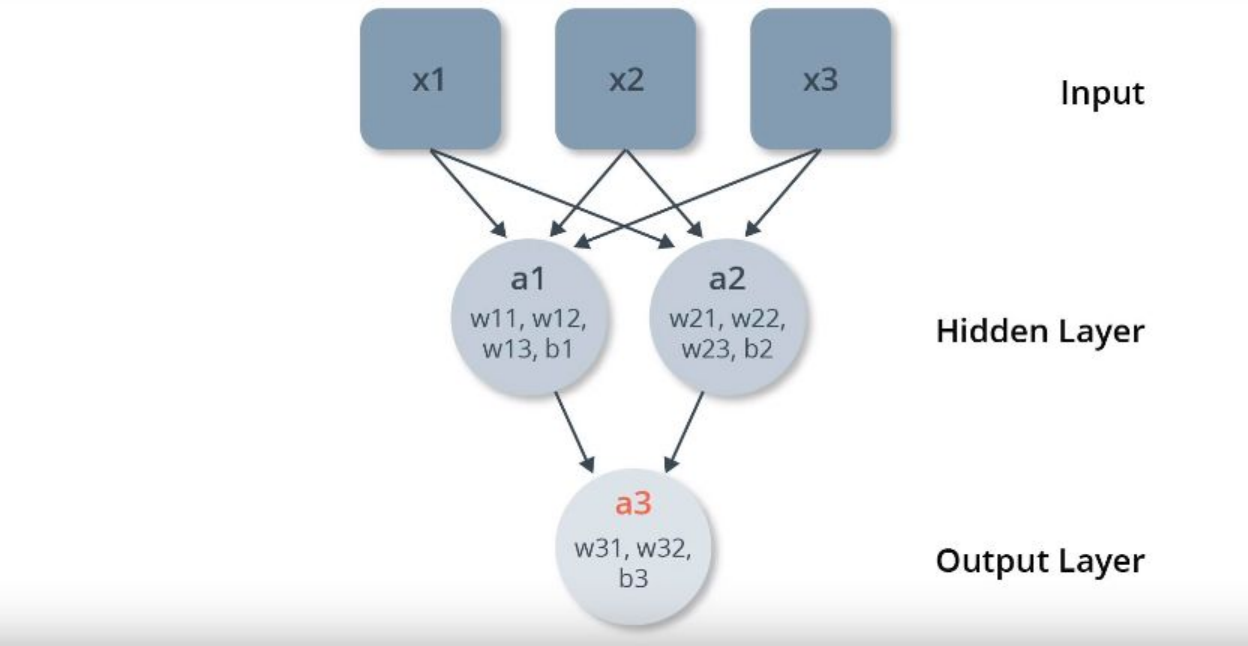


w = 가중치(weight)

b = biases

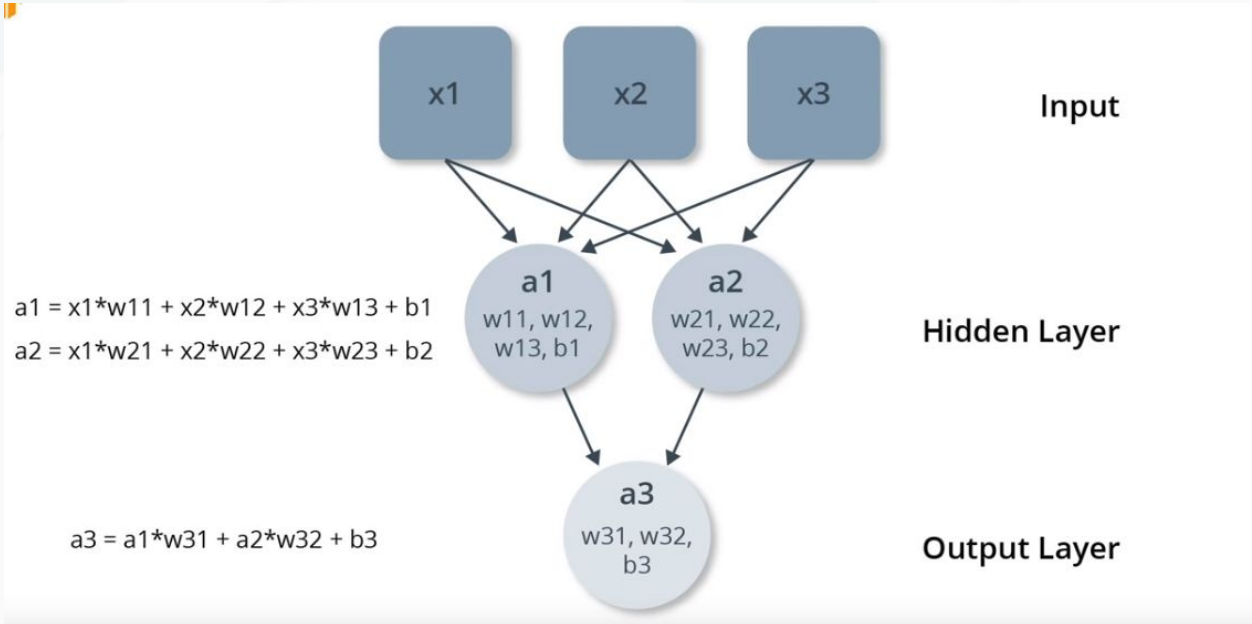


a3가 계산된후 출력.


x1
x2
x3
Input
a1
w11, w12,
w13, b1
a2
w21, w22,
w23, b2
Hidden Layer
a3
w31, w32,
b3
Output Layer

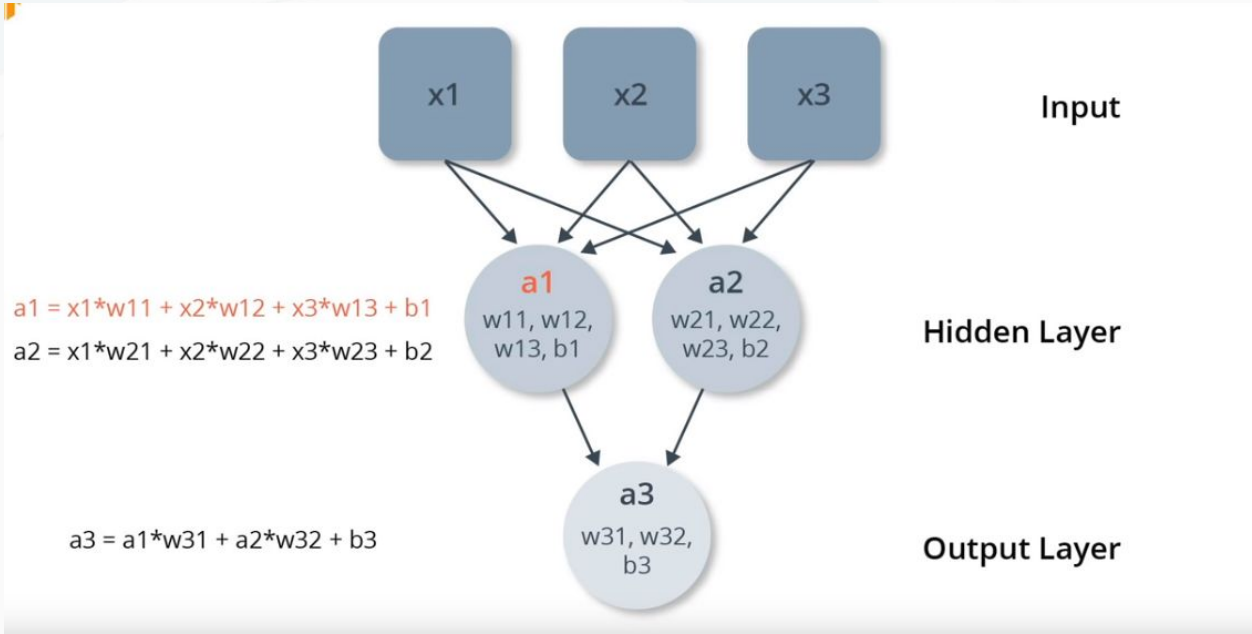
뉴런간의 계산

뉴런 a1의 출력 값은 입력 x1에 가중치 w11을 곱한 다음 입력 x2에 가중치 w12를 곱한 다음 입력 x3에 w13을 곱한 다음 신호 가중치 b1을 가산하여 계산됨.



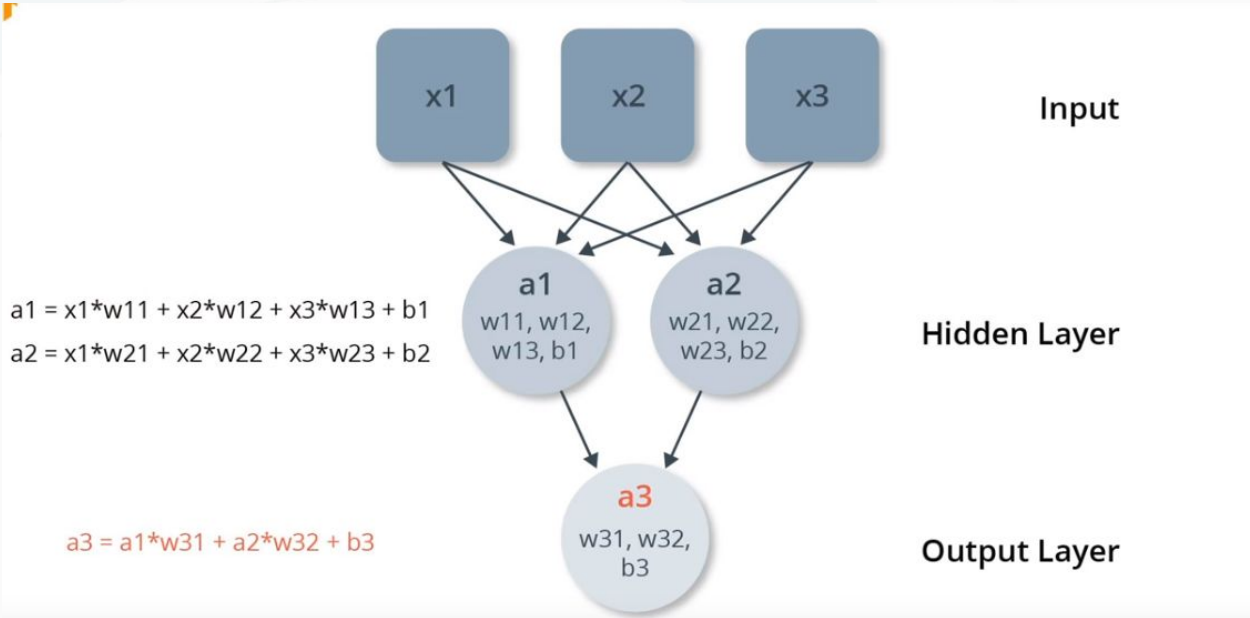
뉴런간의 계산

a1 계산


x1
x2
x3
Input
a1
w11, w12,
w13, b1
a2
w21, w22,
w23, b2
Hidden Layer
a1 = x1*w11 + x2*w12 + x3*w13 + b1
a2 = x1*w21 + x2*w22 + x3*w23 + b2
a3
w31, w32,
b3
Output Layer
a3 = a1*w31 + a2*w32 + b3

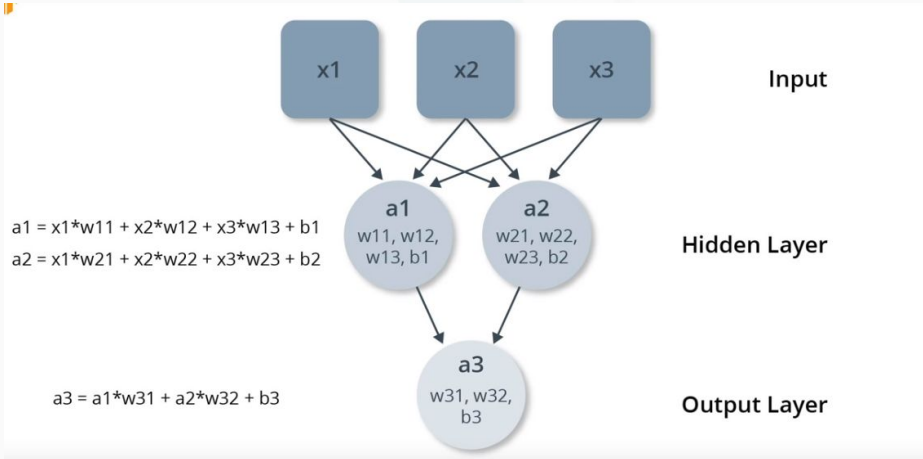
뉴런간의 계산

a3 계산


x1
x2
x3
Input
a1
w11, w12,
w13, b1
a2
w21, w22,
w23, b2
Hidden Layer
a1 = x1*w11 + x2*w12 + x3*w13 + b1
a2 = x1*w21 + x2*w22 + x3*w23 + b2
a3
w31, w32,
b3
Output Layer
a3 = a1*w31 + a2*w32 + b3

뉴런간의 계산

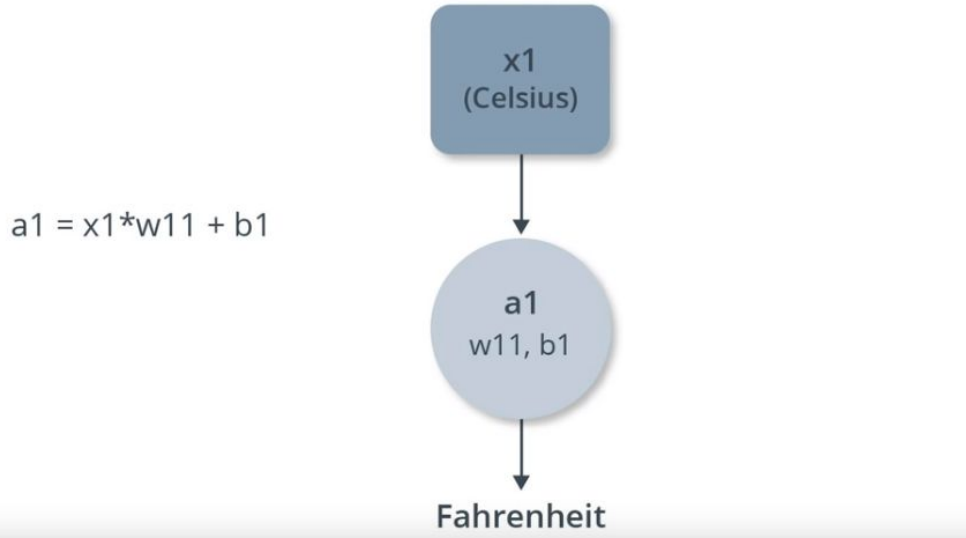
- 출력 값 a3을 계산 - a1의 결과에 w31을 곱한 다음 a2에 w32를 곱한 결과를 더한 다음 단일 가중치 b3을 더함.
- 최상의 가중치와 편향을 조정, 수학적인 의미의 공식은 변하지 않음.
- 다르게 말하면, training 과정은 오직 w와 b 변수가 입력을 출력과 일치시킬 수 있도록 변경
- 개념적으로 머신러닝이 어떻게 작동하는지 설명



화씨섭씨 모델

- 1개의 뉴론. 1개 weight = w11, 1개 bias = b1

```
[ ] l0 = tf.keras.layers.Dense(units=1, input_shape=[1])
```

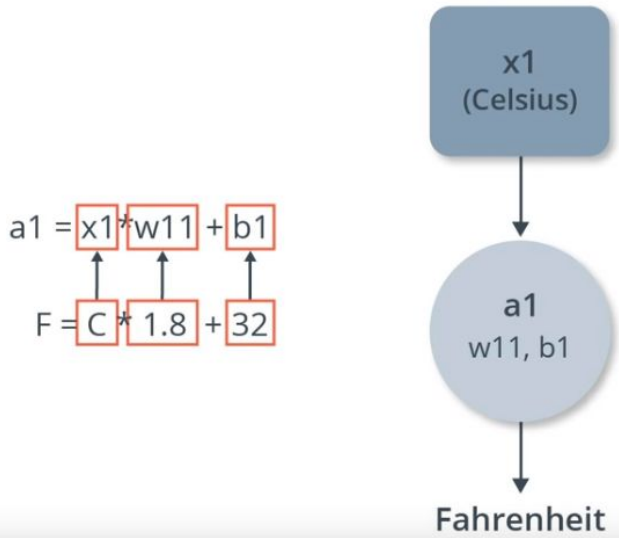


화씨섭씨 모델

섭씨를 화씨로 변환하는 공식은 선형 방정식 F (C = 1.8, 32)입니다.

w11 = 1.8, b1=32, x1 = C(섭씨)

```
[ ] l0 = tf.keras.layers.Dense(units=1, input_shape=[1])
```



화씨섭씨 모델

- 가중치(weight) 는 1.8에 가깝게 조정되고 편향(bias)은 32
- 맵핑(mapping)를 보여주기 위한 예제
- 실제 머신러닝에서는 변수를 이렇게 정확히 맞추는것은 거의 불가능
- 알고리즘(혹은 공식)을 모르는 상태에서 레이어, 가중치를 잘 만들어서 튜닝(tuning)시킴
- 계속 여러 종류의 레이어와 모델 형식을 바꿔가면서 학습(training)을 해 가면서, 가장 좋은 형태를 찾는것이 중요

화씨섭씨 모델

weight와 bias가 실제 공식과 맞아 떨어지는 경우의 예

```
[ ] print("These are the layer variables: {}".format(l0.get_weights()))
```

```
These are the layer variables: [array([[1.8215632]], dtype=float32), array([29.174837], dtype=float32)]
```

화씨섭씨 모델

더 많은 레이어를 사용 (뉴런도 더 많음)

```
[20] l0 = tf.keras.layers.Dense(units=4, input_shape=[1])
     l1 = tf.keras.layers.Dense(units=4)
     l2 = tf.keras.layers.Dense(units=1)
     model = tf.keras.Sequential([l0,l1,l2])
     model.compile(loss='mean_squared_error',optimizer=tf.keras.optimizer.Adam(0.1))
     model.fit(celsius_q, fahrenheit_a, epochs=500, verbos=False)
     print("Finished training the model")
     print(model.predict([100.0]))
     print("Model predicts that 100 degrees Celsius is: {} degrees Fahrenheit".format(model.predict([100.00])))
     print("These are the l0 variables: {}".format(l0.get_weights()))
     print("These are the l1 variables: {}".format(l1.get_weights()))
     print("These are the l2 variables: {}".format(l2.get_weights()))
```

화씨섭씨 모델

1.8이나 32 같은 (이전에 weight와 bias) 값을 찾을수 없음.

```
Finished training the model
[[211.74742]]
Model predicts that 100 degrees Celsius is: [[211.74742]] degrees Fahrenheit
These are the 10 variables: [array([[-0.18800685, -0.19248432, -0.5858944, -0.67664355]],
      dtype=float32), array([-3.5765584, 1.7353412, -3.6145658, -3.606096 ], dtype=float32)]
These are the l1 variables: [array([[ 1.0775942, 0.3412887, -0.20648848, 1.0967993 ],
      [ 0.38938883, -0.01514232, 0.62254876, -1.0326661 ],
      [ 0.34104902, -0.23185489, -0.68317974, -0.02780363],
      [ 0.5441887, -0.2788269, -0.5909913, 0.32856533]],
      dtype=float32), array([-3.5302405, 1.1981959, 3.5821548, -3.5761232], dtype=float32)]
These are the l2 variables: [array([[-1.0786525],
      [ 0.20307049],
      [ 0.92228246],
      [-0.7923224]], dtype=float32), array([3.4605324], dtype=float32)]
```