Computational Thinking & Artificial Intelligence

(10th Lecture, Deep Learning)

Eunsom Jeon
Dept. of Computer Science & Engineering
ejeon6@seoultech.ac.kr

Lecture

- Understanding Python Programming
- Understanding Neural Network in Programming
- Understanding Deep Learning

Lecture program

- 1st Lecture: Introduction
- 2nd Lecture: Matrix and Vector
- 3rd Lecture: Perceptron
- 4th Lecture: Understand Learning
- 5th Lecture: Neural Network
- 6th Lecture: Learning Techniques
- 7th Lecture: Deep Learning

8th Practice: Deep Learning Library

9th Practice: Python Programming

10th Practice: Deep Learning Programming 1

11th Practice: Deep Learning Programming 2

12th Lecture: Generative AI

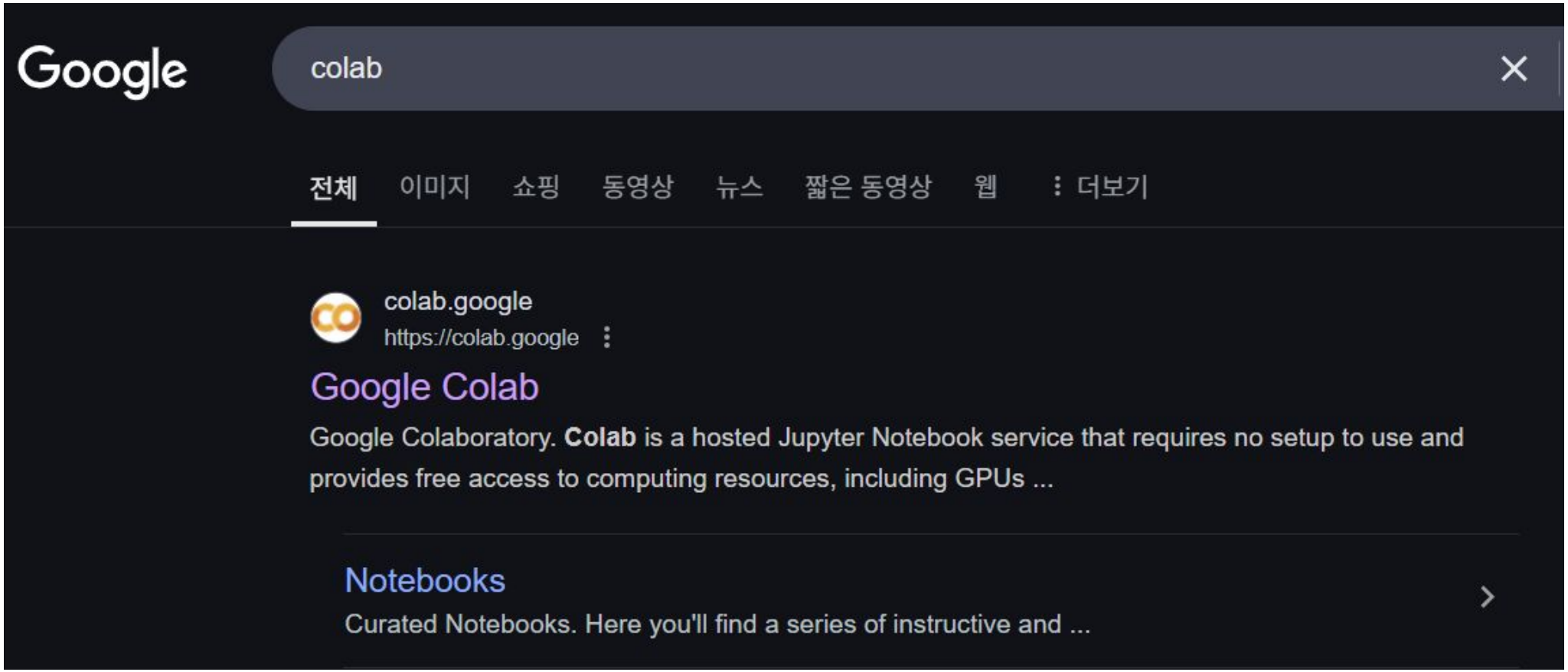
13th Practice: AI and Ethics

14th Lecture: Final Term Exam

15th Lecture: Make-up Lecture

Google Colab Usage

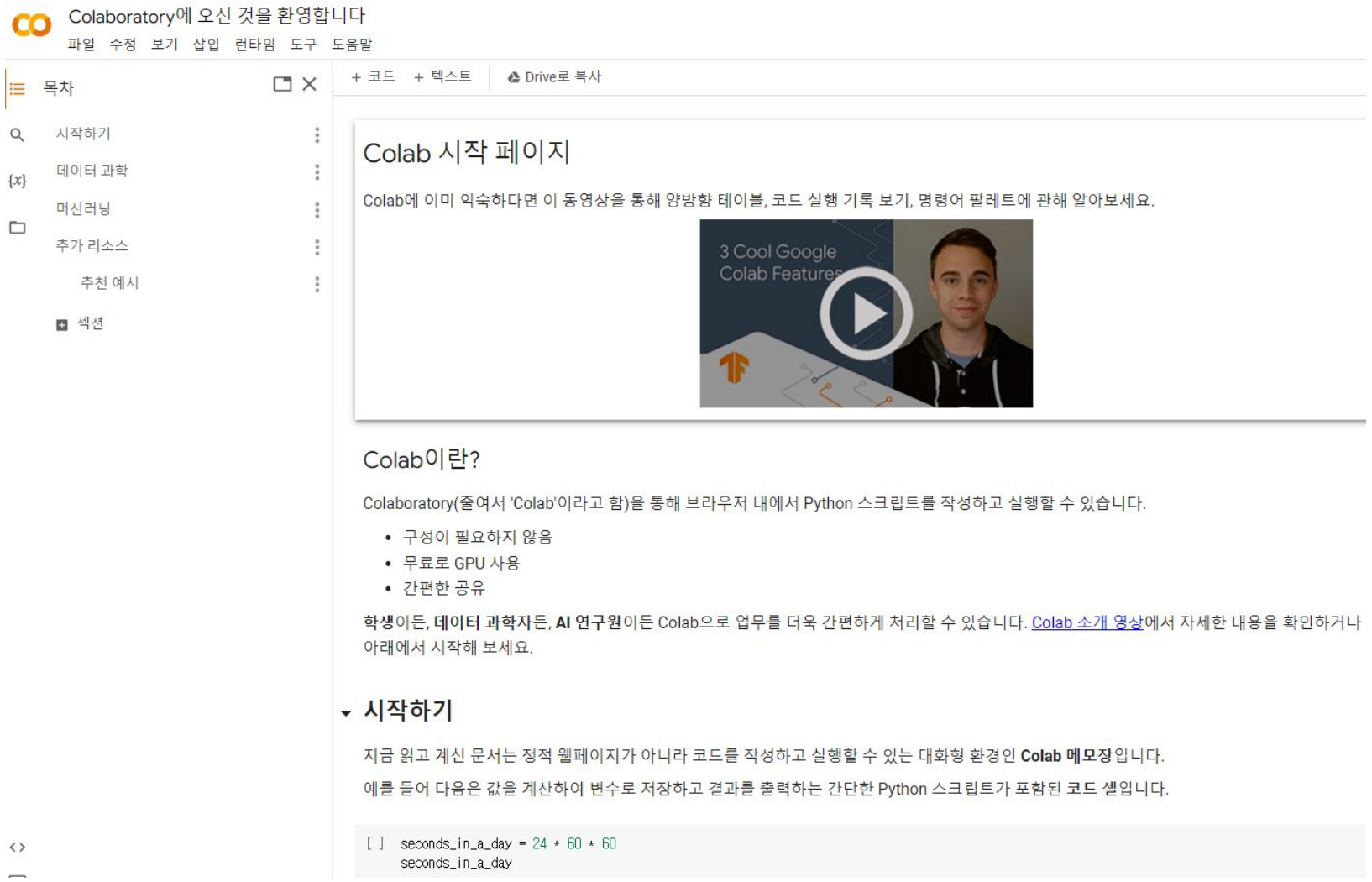
Google Colab Usage



코랩을 쓰는 가장 큰 이유는 프로그램을 따로 설치할 필요 없고,
구글 계정만 있으면 **GPU**까지 무료로 쓸 수 있다는 장점이 있습니다.

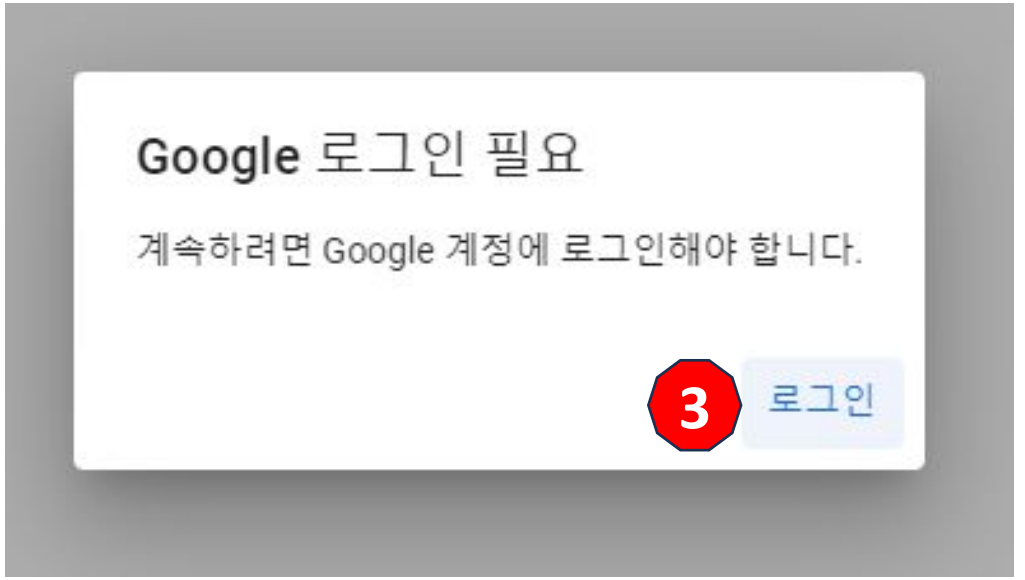
코랩을 실행하는 방법은
(1) 구글에 **colab** 혹은 코랩을 검색합니다.

Google Colab Usage



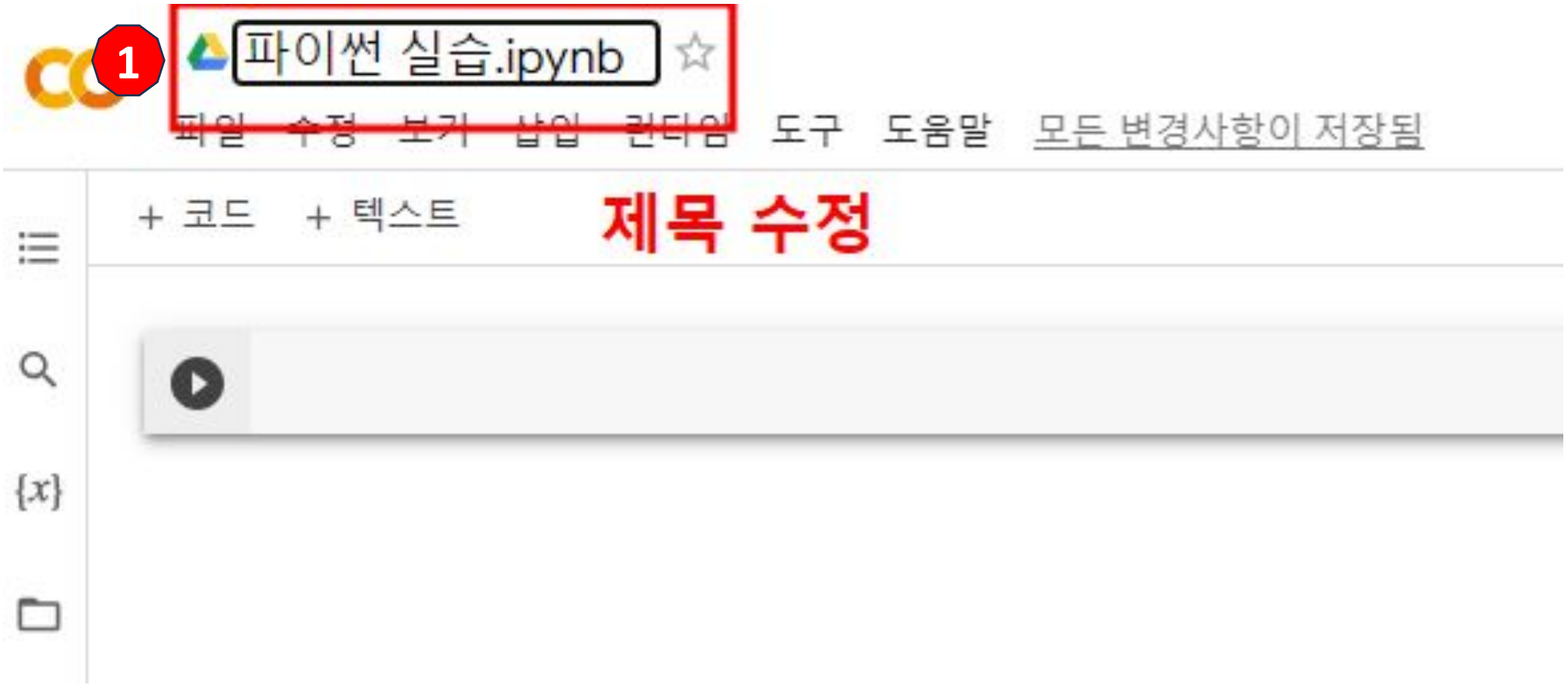
(2) 들어가면 이런 시작 화면이 뜹니다.

Google Colab Usage



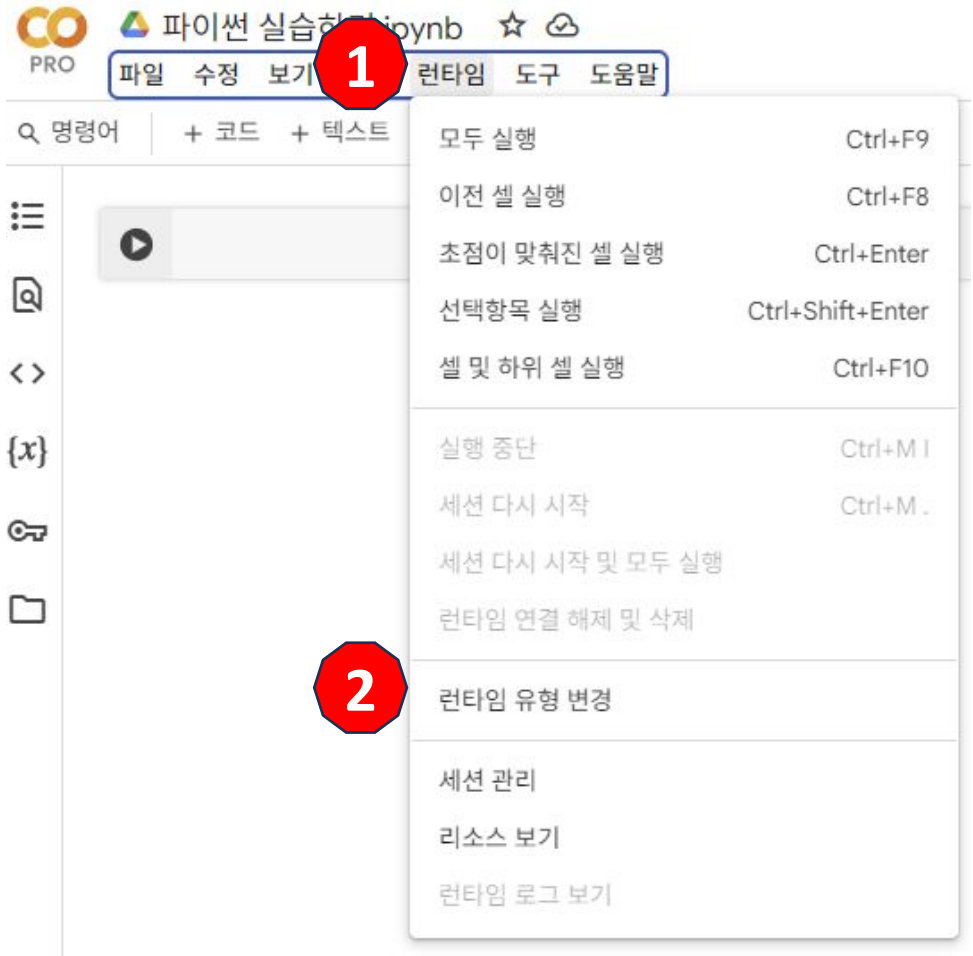
(3) 새로운 노트를 만들어야 하는데 만들려면 구글 로그인이 필요합니다.
본인 구글 계정으로 로그인을 하면 됩니다.

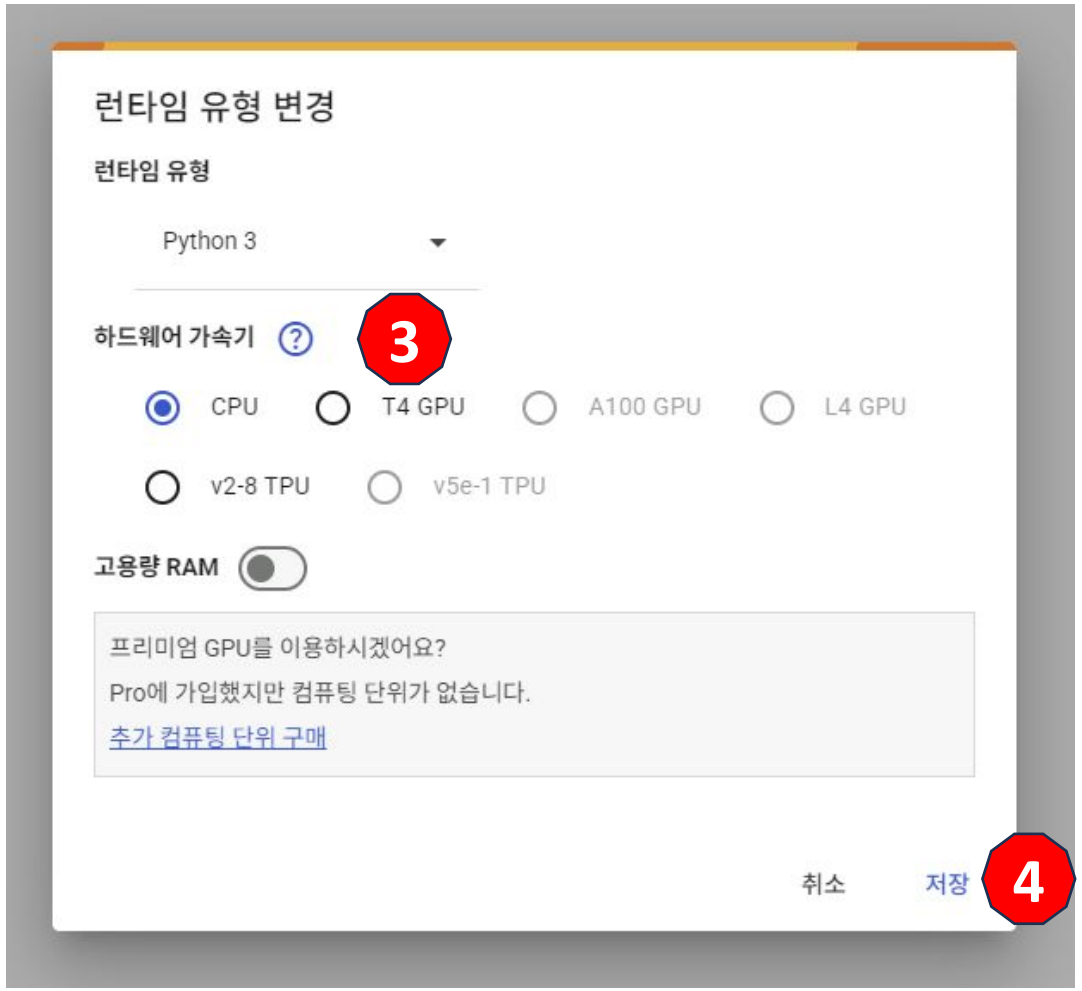
Google Colab Usage



(3) 로그인 후 다시 새 노트를 만들면, 위에 사진처럼 뜨면 된 겁니다.
이때 다시 파일을 구분하기 위해 제목을 수정합니다.

Google Colab Usage



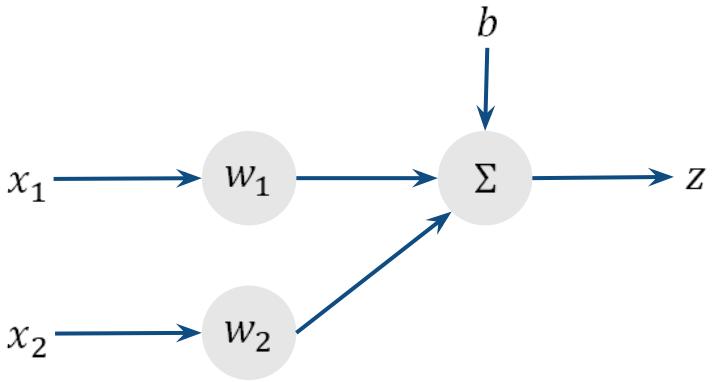


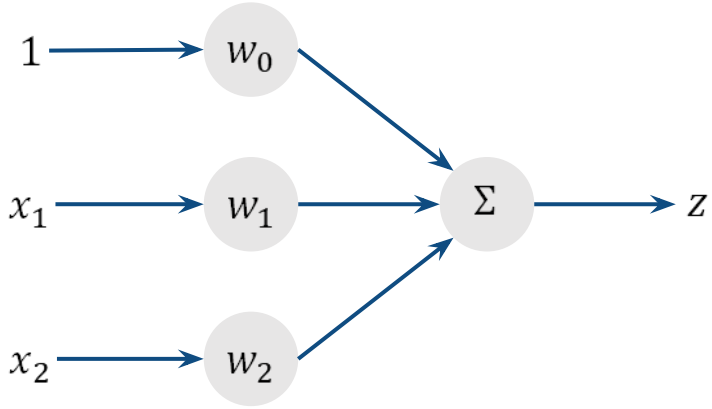
(3) GPU는 딥러닝을 돌릴 때 필요한 자원으로 코랩에서는 무료로 지원해주고 있습니다.
설정은 위와 같이 (1) 런타임을 누르고 (2) 런타임 유형 변경을 누릅니다. 그러면 오른쪽처럼 뜨게 되는데,
(3) 이때 T4 GPU나 v2-8 GPU를 클릭하고 (4) 저장하면 GPU를 간단하게 설정할 수 있습니다.

Python Programming with NN

Perceptron - simple perceptron

- Single output perceptron using a weight vector $\boldsymbol{w} = [w_1, \cdots, w_n]^T$ and a bias $\boldsymbol{b}$
  - Let $\boldsymbol{x} = [x_1, \cdots, x_n]^T$ be a vector than we define a homogeneous vector $\tilde{\boldsymbol{x}} = [1, x_1, \cdots, x_n]^T$
  - Integrate a bias to a weight vector using a homogeneous vector $\tilde{\boldsymbol{x}}$





$$z = \boldsymbol{w}^T\boldsymbol{x} + b$$

$$z = w_1x_1 + w_2x_2 + b$$

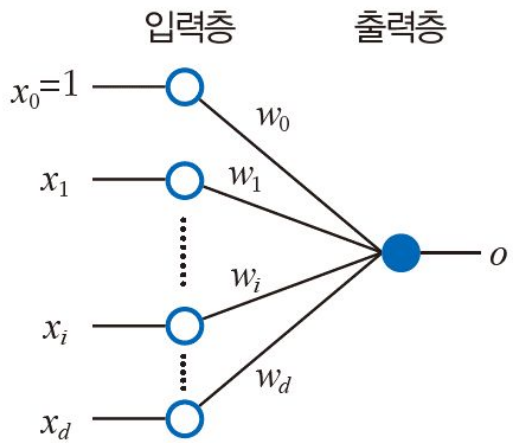
$$z = \boldsymbol{w}^T\tilde{\boldsymbol{x}}$$

$$z = w_1x_1 + w_2x_2 + 1 \cdot w_0$$

where, w = [w1, w2]
w1, w2는 우리가 원하는 정답을 잘 표현하도록 계속해서 값이 변함 -> 해당 과정을 모델이 “학습” 한다고 표현.

Perceptron

- Single output perceptron using a weight vector $\boldsymbol{w} = [w_1, \cdots, w_n]^T$ and a bias $\boldsymbol{b}$
  - Let $\boldsymbol{x} = [x_1, \cdots, x_n]^T$ be a vector than we define a homogeneous vector $\tilde{\boldsymbol{x}} = [1, x_1, \cdots, x_n]^T$
  - Integrate a bias to a weight vector using a homogeneous vector $\tilde{\boldsymbol{x}}$




```
import numpy as np                                          # numpy 라이브러리를 불러옴
x=np.array([[1,0,0],[1,0,1],[1,1,0],[1,1,1]])    # [그림 4-3(a)]의 4개 샘플
w=np.array([-0.5,1.0,1.0])    # [그림 4-3(b)]의 퍼셉트론 가중치 벡터
s=np.sum(x*w,axis=1)          # 샘플 각각에 대해 요소별로 곱한 후 요소를 합산
```

해당 임의의 Input은 3개의 특징을 가진 데이터 샘플 4개로 이루어짐
ex) (한 사람 당 키, 몸무게, 혈액형 정보) x 4명의 사람

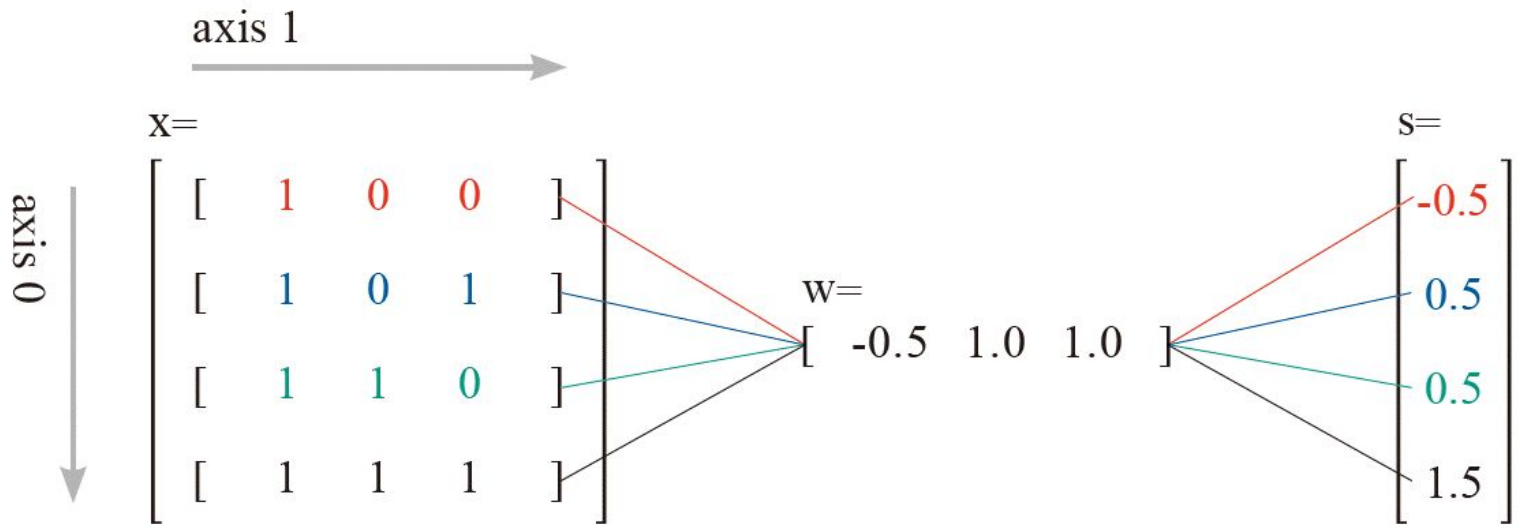
이를 행렬로 표현할 시,

	키	몸무게	혈액형 <- 특징
사람 1:	1	0	0
사람 2:	1	0	1
사람 3:	1	0	0
사람 4:	1	1	1

키, 몸무게, 혈액형에 각각 어느정도의 가중치를 줘서 한 사람의 MBTI를 맞출 것인가! -> 컴퓨터가 많은 키, 몸무게, 혈액형의 정보를 가진 많은 사람들의 데이터를 보고, MBTI를 맞추는데 최적의 키, 몸무게, 혈액형 정보의 비율을 결정함! (weight update)

Perceptron

- Computation in program



Perceptron

- Gate

1. AND 게이트

두 입력(A, B)가 있을 때, A와 B 모두 1일 때에만(active) 출력 Q가 1(active)

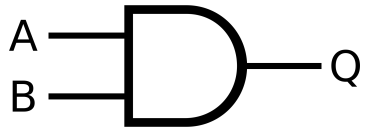
2. OR Gate

두 입력(A, B)가 있을 때, 두 입력 중, 하나만 1(active)이라도 출력 Q가 1(active)
즉, 두 입력이 모두 0일 때에만 출력 Q가 0임

Perceptron의 output: $z = \boldsymbol{w}^T \boldsymbol{x} + b$

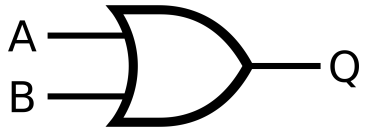
이때, $\boldsymbol{w}^T \boldsymbol{x} + b = 0$ 이 되는 곳이 결정경계 이며, (2차원에서는 직선)
결정경계의 한 쪽은 +1(논리게이트 output 상 1) ~ $\boldsymbol{w}^T \boldsymbol{x} + b > 0$
나머지 한 쪽은 -1(논리게이트 output 상 0)로 분류 ~ $\boldsymbol{w}^T \boldsymbol{x} + b < 0$

AND gate

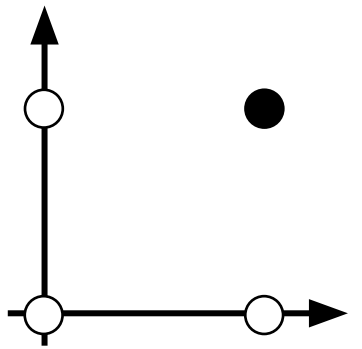


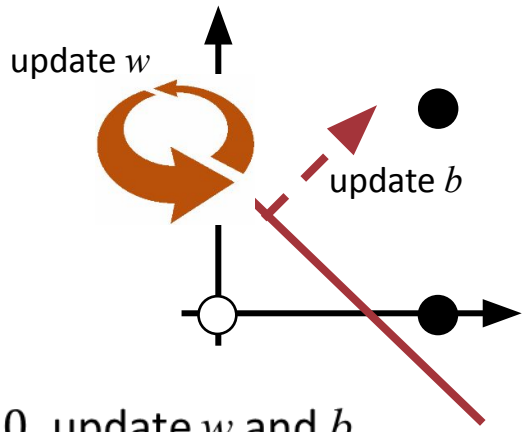
Input		Output
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

OR gate



Input		Output
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1





$3x_1 + 3x_2 - 3 > 0$, update w and b

Perceptron

- OR data

  - Binary classification (-1, 1)

```python
from sklearn.linear_model import Perceptron

# 훈련 집합 구축
X=[[0,0],[0,1],[1,0],[1,1]]
y=[-1,1,1,1]

# fit 함수로 Perceptron 학습
p=Perceptron()
p.fit(X,y)

print("학습된 퍼셉트론의 매개변수: ",p.coef_,p.intercept_)
print("훈련집합에 대한 예측: ",p.predict(X))
print("정확률 측정: ",p.score(X,y)*100,"%")
```

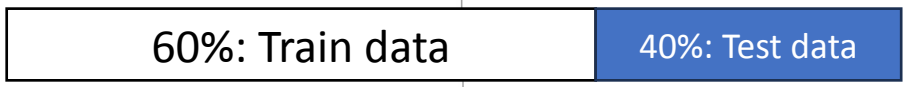
```
학습된 퍼셉트론의 매개변수:  [[2. 2.]] [-1.]
훈련집합에 대한 예측:  [-1  1  1  1]
정확률 측정:  100.0%
```

Perceptron

- Perceptron by using sklearn

  - Handwritten digit data classification (10 class)

```
01  from sklearn import datasets
02  from sklearn.linear_model import Perceptron
03  from sklearn.model_selection import train_test_split
04  import numpy as np
05
06  # 데이터셋을 읽고 훈련 집합과 테스트 집합으로 분할
07  digit=datasets.load_digits()
08  x_train,x_test,y_train,y_test=train_test_split
    (digit.data,digit.target,train_size=0.6)
09
10  # fit 함수로 Perceptron 학습
11  p=Perceptron(max_iter=100,eta0=0.001,verbose=0)
12  p.fit(x_train,y_train) # digit 데이터로 모델링
13
14  res=p.predict(x_test)  # 테스트 집합으로 예측
15
```



데이터 읽기

모델 객체 생성

모델 학습

학습된 모델로 예측

Perceptron

- Accuracy metric

  - Confusion matrix

```
17  conf=np.zeros((10,10))
18  for i in range(len(res)):
19      conf[res[i]][y_test[i]]+=1
20  print(conf)
21
22  # 정확률 계산
23  no_correct=0
24  for i in range(10):
25      no_correct+=conf[i][i]
26  accuracy=no_correct/len(res)
27  print("테스트 집합에 대한 정확률은 ", accuracy*100, "%입니다.")
```

성능 측정

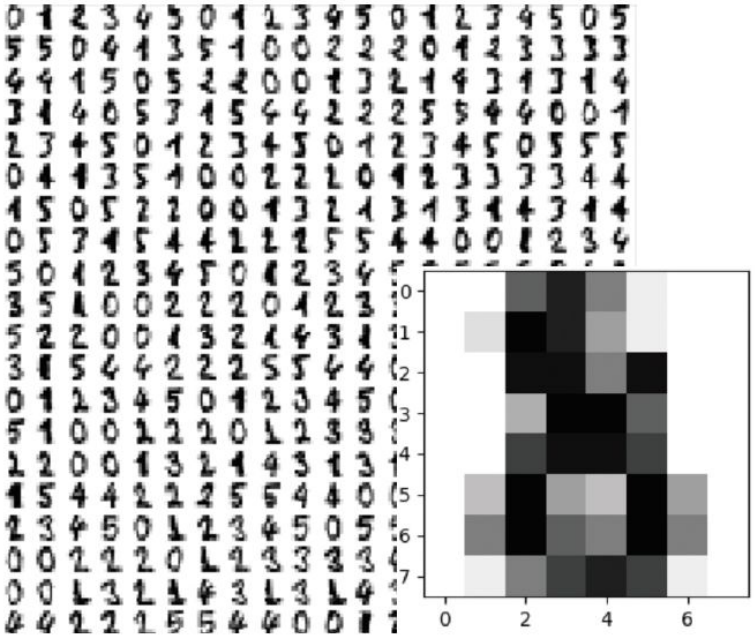
Ground truth

Predicted Class

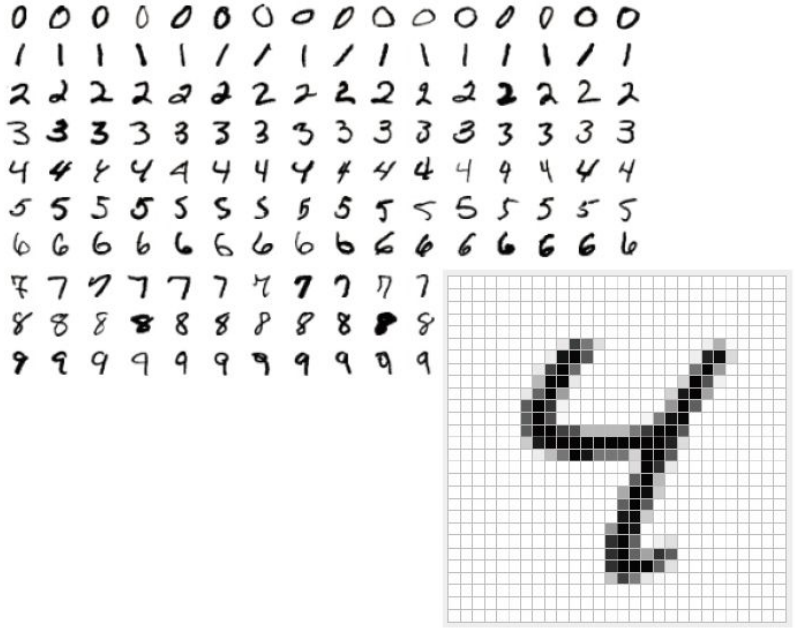
```
[[66.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0. 65.  0.  0.  0.  0.  1.  0.  6.  2.]
 [ 0.  2. 58.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0. 62.  0.  0.  0.  0.  1.  0.]
 [ 0.  1.  0.  0. 60.  0.  0.  2.  1.  0.]
 [ 0.  0.  0.  0.  0. 71.  0.  0.  1.  1.]
 [ 0.  0.  0.  0.  0.  0. 80.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0. 75.  0.  0.]
 [ 0.  3.  0.  2.  0.  0.  1.  1. 66.  0.]
 [ 0.  1.  0.  1.  0.  9.  0.  3.  2. 72.]]
테스트 집합에 대한 정확률은 93.88038942976355%입니다.
```

MLP Implementation

MNIST Dataset



(a) sklearn에서 제공하는 데이터셋

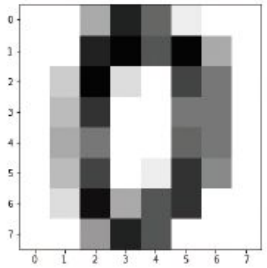


(b) MNIST 데이터셋

MNIST Dataset

```
01  from sklearn import datasets
02  import matplotlib.pyplot as plt
03
04  digit=datasets.load_digits()
05
06  plt.figure(figsize=(5,5))
07  plt.imshow(digit.images[0],cmap=plt.cm.gray_r,interpolation='nearest')
                                                    # 0번 샘플을 그림
08  plt.show()
09  print(digit.data[0])                            # 0번 샘플의 화솟값을 출력
10  print("이 숫자는 ",digit.target[0],"입니다.")
```

MNIST Dataset



어떤 특징을 사용해야 높은 성능을 얻을 수 있을까?

```
[ 0.  0.  5. 13.  9.  1.  0.  0.  0.  0. 13. 15. 10. 15.  5.  0.  0.  3.
 15.  2.  0. 11.  8.  0.  0.  4. 12.  0.  0.  8.  8.  0.  0.  5.  8.  0.
  0.  9.  8.  0.  0.  4. 11.  0.  1. 12.  7.  0.  0.  2. 14.  5. 10. 12.
  0.  0.  0.  0.  6. 13. 10.  0.  0.  0.]

이 숫자는 0입니다.
```

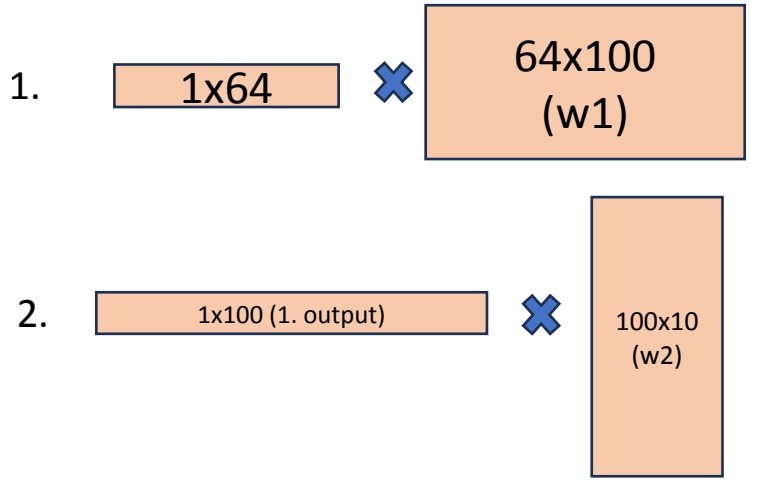
NOTE matplotlib을 이용한 시각화

파이썬에서 matplotlib 라이브러리는 시각화에 가장 널리 쓰인다. 인공지능은 학습 과정이나 예측 결과를 시각화하는 데 matplotlib을 자주 사용한다. matplotlib 사용이 처음이라면 부록 B를 공부해 기초를 먼저 다진다. matplotlib의 공식 사이트에서 제공하는 튜토리얼 문서를 공부하는 것도 효과적인 방법이다. [표 2-1]에서 제시한 https://matplotlib.org/users에 접속해 [Tutorials] 메뉴를 선택한다. 튜토리얼은 Introductory, Intermediate, Advanced로 나뉘어 있으니 최소한 Introductory 코스를 숙지하고 넘어간다.

MLP Implementation

- **MLP**

```
01  from sklearn import datasets
02  from sklearn.neural_network import MLPClassifier
03  from sklearn.model_selection import train_test_split
04  import numpy as np
05
06  # 데이터셋을 읽고 훈련 집합과 테스트 집합으로 분할
07  digit=datasets.load_digits()
08  x_train,x_test,y_train,y_test=train_test_split(digit.data,digit.target,train_
    size=0.6)
09
10  # MLP 분류기 모델을 학습
11  mlp=MLPClassifier(hidden_layer_sizes=(100),learning_rate_init=0.001,batch_
    size=32,max_iter=300,solver='sgd',verbose=True)
12  mlp.fit(x_train,y_train)
13
14  res=mlp.predict(x_test) # 테스트 집합으로 예측
15
16  # 혼동 행렬
17  conf=np.zeros((10,10))
18  for i in range(len(res)):
19      conf[res[i]][y_test[i]]+=1
20  print(conf)
21
22  # 정확률 계산
23  no_correct=0
24  for i in range(10):
25      no_correct+=conf[i][i]
26  accuracy=no_correct/len(res)
27  print("테스트 집합에 대한 정확률은 ", accuracy*100, "%입니다.")
```



- Input은 64개의 특징을 가진 데이터 1개로 가정.
  (행렬 표현 시, 1x64 행렬)
- 첫번째 weight 행렬과 서로 행렬곱
  -> output 행렬은 1x100행렬
- 두 번째 weight 행렬과 서로 행렬곱
  -> 최종 output 행렬은 1x10
  (마지막 weight 행렬의 열은 class 개수와 같음)
- 이 중, 가장 값이 높은 인덱스를 최종 반환!
- 총 파리미터(weight) 수 = 64x100x100x10 = 6,400,000

```
# MLP 분류기 모델을 학습
mlp=MLPClassifier(hidden_layer_sizes=(100),learning_rate_init=0.001,batch_
size=32,max_iter=300,solver='sgd',verbose=True)
mlp.fit(x_train,y_train)

res=mlp.predict(x_test) # 테스트 집합으로 예측
```

MLP Implementation

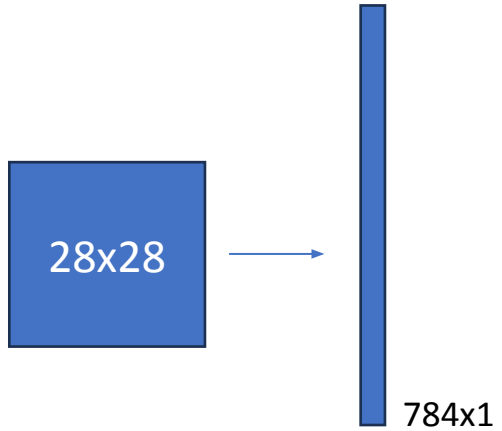
- **MLP**
  - Performance (Accuracy)
    - MLP: **97.22%**

```
Iteration 1, loss = 1.93427517
Iteration 2, loss = 0.32451464
Iteration 3, loss = 0.20142691
Iteration 4, loss = 0.14313824
...
Iteration 40, loss = 0.01163957
Iteration 41, loss = 0.01127886
...
Iteration 80, loss = 0.00538125
Iteration 81, loss = 0.00538663
...
Iteration 107, loss = 0.00405861
Iteration 108, loss = 0.00402524
Iteration 109, loss = 0.00397349
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.

[[73.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0. 69.  1.  0.  1.  0.  1.  0.  1.  0.]
 [ 0.  0. 70.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0. 67.  0.  0.  0.  0.  1.  3.]
 [ 1.  0.  0.  0. 77.  0.  1.  0.  0.  0.]
 [ 1.  0.  0.  0.  0. 64.  1.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0. 67.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  1.  0. 83.  0.  0.]
 [ 0.  2.  0.  0.  0.  0.  0.  1. 58.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.  0. 71.]]

테스트 집합에 대한 정확률은 97.2183588317107%입니다.
```

Deep Learning Implementation

Deep Learning Implementation

- **MNIST recognition example**
  - Step 1: Prepare your data




```
01  import numpy as np
02  import tensorflow as tf
03  from tensorflow.keras.datasets import mnist
04
05  from tensorflow.keras.models import Sequential
06  from tensorflow.keras.layers import Dense
07  from tensorflow.keras.optimizers import Adam
08
09  # MNIST 읽어 와서 신경망에 입력할 형태로 변환
10  (x_train, y_train), (x_test, y_test) = mnist.load_data()
11  x_train = x_train.reshape(60000,784)                    #  텐서 모양 변환
12  x_test = x_test.reshape(10000,784)
13  x_train=x_train.astype(np.float32)/255.0                # ndarray로 변환
14  x_test=x_test.astype(np.float32)/255.0
15  y_train=tf.keras.utils.to_categorical(y_train,10)  # 원핫 코드로 변환
16  y_test=tf.keras.utils.to_categorical(y_test,10)
17
```

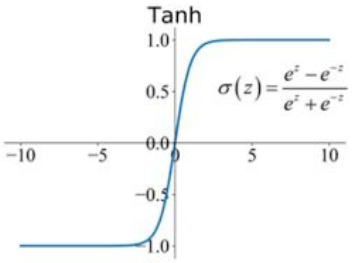
11~12행: reshape 함수로 2차원 구조의 텐서를 1차원 구조로 변환
[60000, 28, 28] -> [60000, 784]

13~14행: float32 데이터형으로 변환하고 [0,255] 범위를 [0,1] 범위로 정규화

15~16행: 레이블을 원핫 코드로 변환

Deep Learning Implementation

- **MNIST recognition example**
  - Step 2: Design the neural network structure
    - Lines 18-20: *Setting the number of nodes* in the input layer, hidden layer, and output layer
    - Line 22: *Create a Sequential model* and store it in the mlp object
    - Line 23: Add a hidden layer (input_shape is set to the input layer, units is set to the currently stacked hidden layer)
    - Line 24: Add an output layer (input_shape can be omitted, units are set to the output layer currently being built)

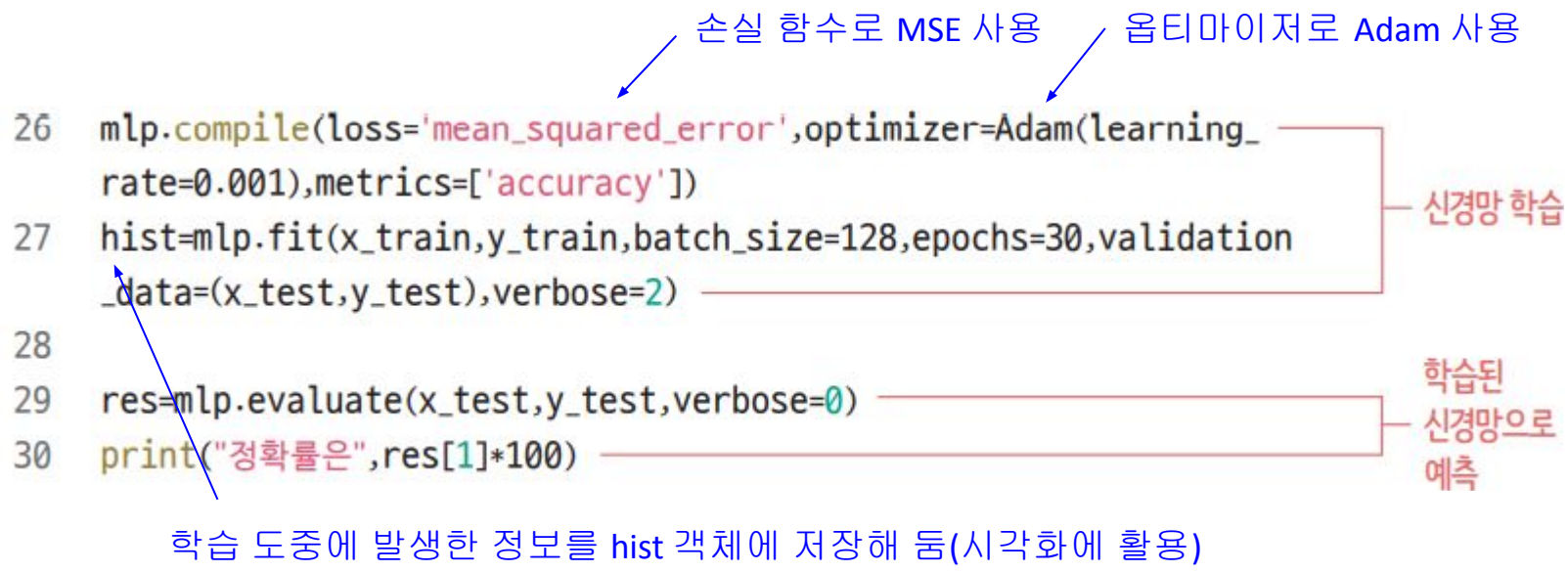



```
18  n_input=784
19  n_hidden=1024
20  n_output=10
21
22  mlp=Sequential()
23  mlp.add(Dense(units=n_hidden,activation='tanh',input_shape=(n_input,),
    kernel_initializer='random_uniform',bias_initializer='zeros'))
24  mlp.add(Dense(units=n_output,activation='tanh',kernel_
    initializer='random_uniform',bias_initializer='zeros'))
25
```

신경망 구조 설계

Deep Learning Implementation

- **MNIST recognition example**
  - Step 3: Train the neural network
    - Line 26: Prepare training with the compile function (loss parameter is the loss function, optimizers is the optimizer settings)
    - Line 27: The fit function performs the actual training (batch_size, epochs set, validation_data sets the validation set to be used during training)
  - Step 4: Predict
    - Line 29: Measure accuracy with the evaluate function



Deep Learning Implementation

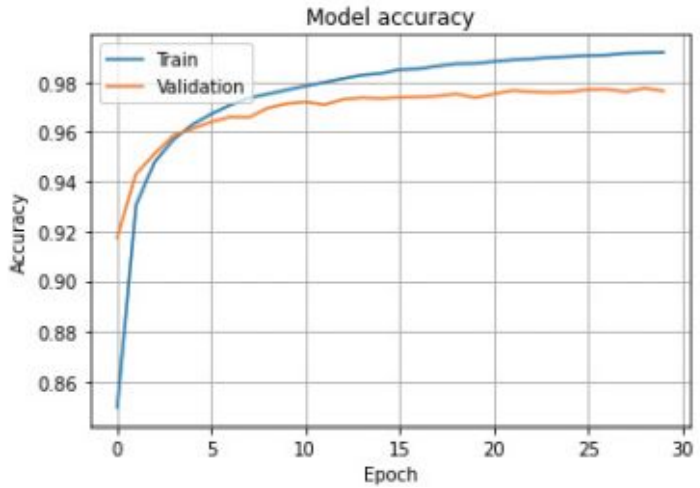
- **MNIST recognition example**
  - Execution result
    - 97.65% accuracy on test set

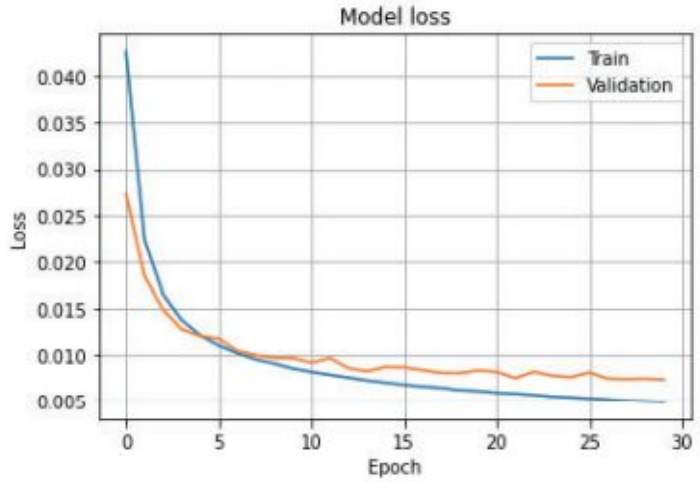
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 - 2s - loss: 0.0427 - accuracy: 0.8492 - val_loss: 0.0272 - val_accuracy: 0.9173
Epoch 2/30
60000/60000 - 2s - loss: 0.0223 - accuracy: 0.9305 - val_loss: 0.0184 - val_accuracy: 0.9432
...
Epoch 30/30
60000/60000 - 2s - loss: 0.0049 - accuracy: 0.9919 - val_loss: 0.0074 - val_accuracy: 0.9765
정확률은 97.64999747276306
```

Deep Learning Implementation

- **MNIST recognition example**
  - Learning curve visualization
    - Visualize using the information contained in the *hist* object

```
31  import matplotlib.pyplot as plt
32
33  # 정확률 곡선
34  plt.plot(hist.history['accuracy'])
35  plt.plot(hist.history['val_accuracy'])
36  plt.title('Model accuracy')
37  plt.ylabel('Accuracy')
38  plt.xlabel('Epoch')
39  plt.legend(['Train','Validation'], loc='upper left')
40  plt.grid()
41  plt.show()
42
43  # 손실 함수 곡선
44  plt.plot(hist.history['loss'])
45  plt.plot(hist.history['val_loss'])
46  plt.title('Model loss')
47  plt.ylabel('Loss')
48  plt.xlabel('Epoch')
49  plt.legend(['Train','Validation'], loc='upper right')
50  plt.grid()
51  plt.show()
```





Deep Learning Implementation

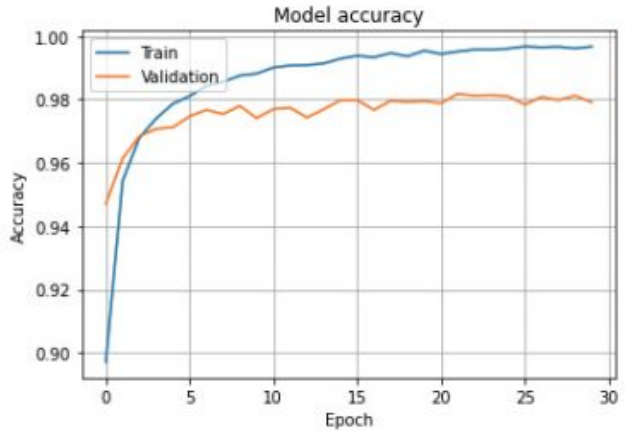
- **Extending the MNIST recognition example from MLP to Deep MLP**
  - 1 hidden layer □ expanded to 4
  - The rest of the code is the same as the previous MLP, only the shaded part is added.

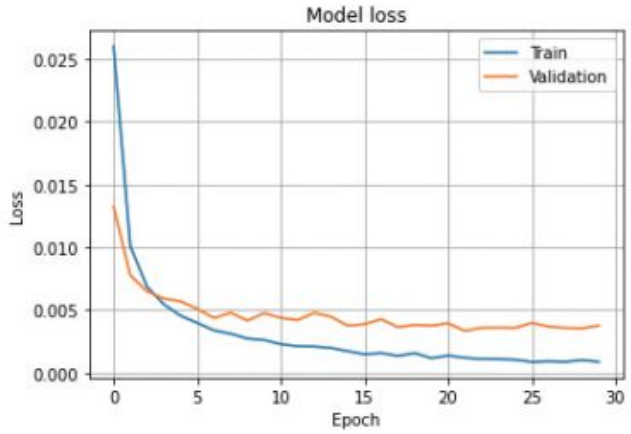
```
17  # 신경망 구조 설정
18  n_input=784
19  n_hidden1=1024
20  n_hidden2=512
21  n_hidden3=512
22  n_hidden4=512
23  n_output=10
24
25  # 신경망 구조 설계
26  mlp=Sequential()
27  mlp.add(Dense(units=n_hidden1,activation='tanh',input_shape=(n_input,),kernel_
    initializer='random_uniform',bias_initializer='zeros'))
28  mlp.add(Dense(units=n_hidden2,activation='tanh',kernel_initializer='random_
    uniform',bias_initializer='zeros'))
29  mlp.add(Dense(units=n_hidden3,activation='tanh',kernel_initializer='random_
    uniform',bias_initializer='zeros'))
30  mlp.add(Dense(units=n_hidden4,activation='tanh',kernel_initializer='random_
    uniform',bias_initializer='zeros'))
31  mlp.add(Dense(units=n_output,activation='tanh',kernel_initializer='random_
    uniform',bias_initializer='zeros'))
```

Deep Learning Implementation

- **Extending the MNIST recognition example from MLP to Deep MLP**
  - Performance results
    - Accuracy rate **97.91%** □ ***0.26% improvement*** compared to MLP's 97.65%

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 - 5s - loss: 0.0260 - accuracy: 0.8971 - val_loss: 0.0132 - val_accuracy: 0.9471
Epoch 2/30
60000/60000 - 5s - loss: 0.0101 - accuracy: 0.9543 - val_loss: 0.0078 - val_accuracy: 0.9614
Epoch 29/30
60000/60000 - 5s - loss: 0.0010 - accuracy: 0.9961 - val_loss: 0.0035 - val_accuracy: 0.9812
Epoch 30/30
60000/60000 - 5s - loss: 8.8359e-04 - accuracy: 0.9967 - val_loss: 0.0038 - val_accuracy: 0.9791
정확률은 97.9099988937378
```

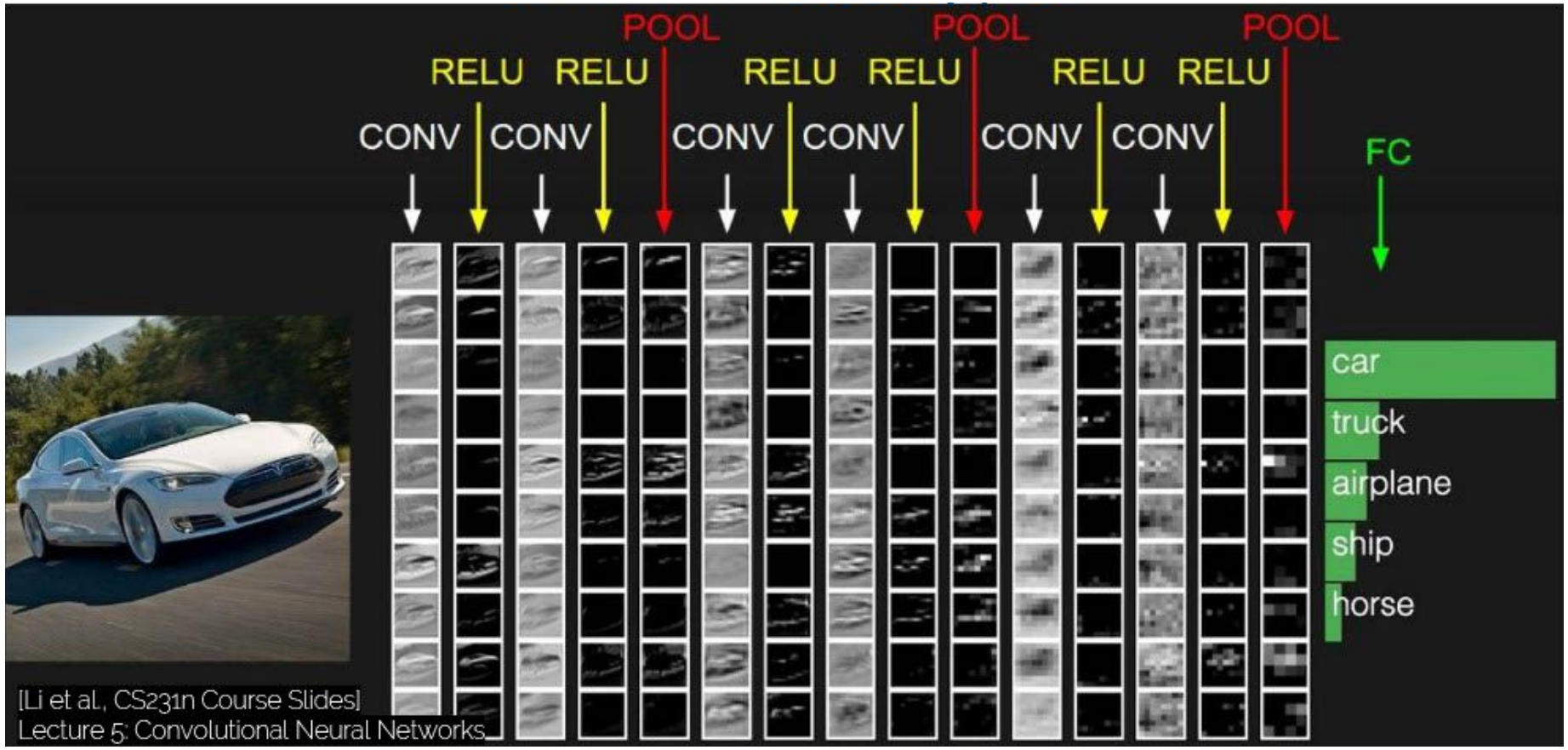




CNN Architecture

CNN Architecture

- CNN Prototype



CNN Architecture

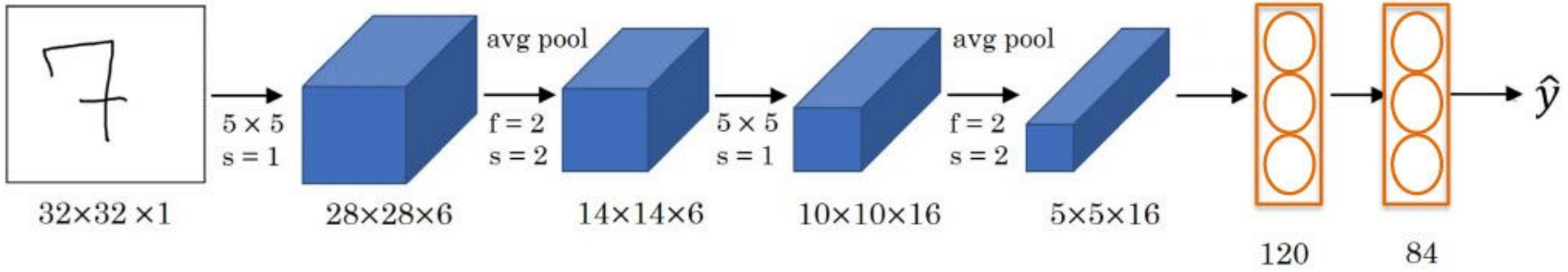
- LeNet

•Digit recognition: 10 classes

•Use of tanh/sigmoid activations □ not common now!

•Conv □ Pool □ Conv □ Pool □ Conv □ FC

- As w



60k parameters

CNN Architecture

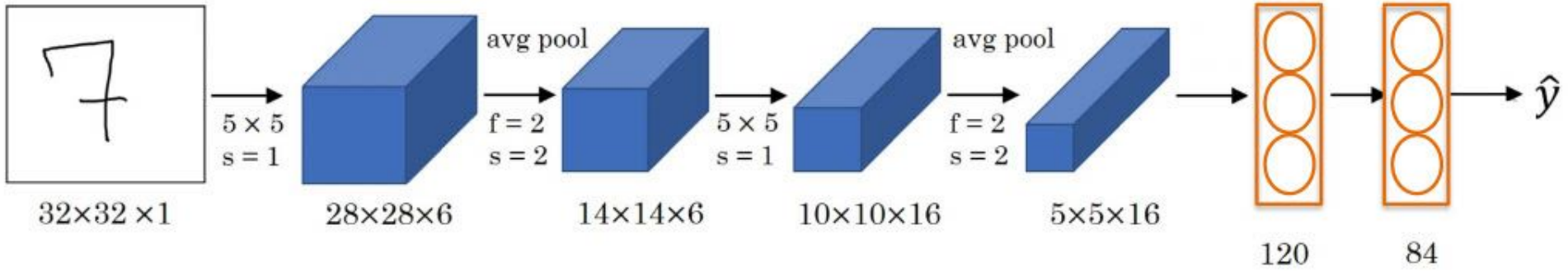
- LeNet

•Digit recognition: 10 classes

•Use of tanh/sigmoid activations □ not common now!

•Conv □ Pool □ Conv □ Pool □ Conv □ FC

- As w



60k parameters

CNN Implementation

CNN Implementation

- LeNet Implementation

•MNIST dataset recognition using LeNet-5
•C-P-C-P-C-FC-FC structure
•The code itself is similar to the existing DMLP
•Since the 2D structure must be maintained reshape(60000,28,28,1)

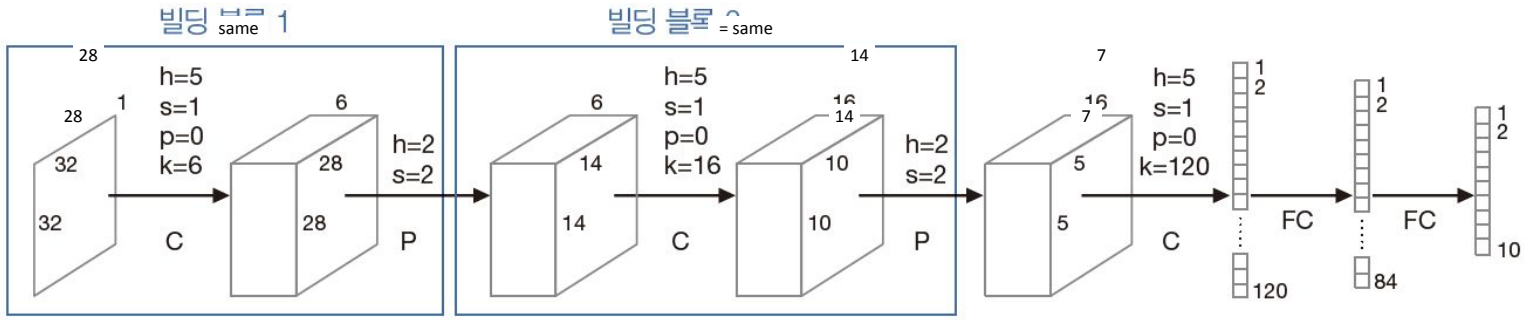
```
01  import numpy as np
02  import tensorflow as tf
03  from tensorflow.keras.datasets import mnist
04  from tensorflow.keras.models import Sequential
05  from tensorflow.keras.layers import Conv2D,MaxPooling2D,Flatten,Dense
06  from tensorflow.keras.optimizers import Adam
07
08  # MNIST 데이터셋을 읽고 신경망에 입력할 형태로 변환
09  (x_train,y_train),(x_test,y_test)=mnist.load_data()
10  x_train=x_train.reshape(60000,28,28,1)
11  x_test=x_test.reshape(10000,28,28,1)
12  x_train=x_train.astype(np.float32)/255.0
13  x_test=x_test.astype(np.float32)/255.0
14  y_train=tf.keras.utils.to_categorical(y_train,10)
15  y_test=tf.keras.utils.to_categorical(y_test,10)
16
```

CNN Implementation

- LeNet Implementation

```
17  # LeNet-5 신경망 모델 설계
18  cnn=Sequential()
19  cnn.add(Conv2D(6,(5,5),padding='same',activation='relu',input_shape=(28,28,1)))
20  cnn.add(MaxPooling2D(pool_size=(2,2)))
21  cnn.add(Conv2D(16,(5,5),padding='same',activation='relu'))
22  cnn.add(MaxPooling2D(pool_size=(2,2)))
23  cnn.add(Conv2D(120,(5,5),padding='same',activation='relu'))
24  cnn.add(Flatten())
25  cnn.add(Dense(84,activation='relu'))
26  cnn.add(Dense(10,activation='softmax'))
```

C-P-C-P-C-FC-FC 구조의 컨볼루션 신경망 설계



CNN Implementation

- LeNet Implementation

```
17  # LeNet-5 신경망 모델 설계
18  cnn=Sequential()
19  cnn.add(Conv2D(6,(5,5),padding='same',activation='relu',input_shape=(28,28,1)))
20  cnn.add(MaxPooling2D(pool_size=(2,2)))
21  cnn.add(Conv2D(16,(5,5),padding='same',activation='relu'))
22  cnn.add(MaxPooling2D(pool_size=(2,2)))
23  cnn.add(Conv2D(120,(5,5),padding='same',activation='relu'))
24  cnn.add(Flatten())
25  cnn.add(Dense(84,activation='relu'))
26  cnn.add(Dense(10,activation='softmax'))
27
28  # 신경망 모델 학습
29  cnn.compile(loss='categorical_crossentropy',optimizer=Adam(),metrics=['accuracy'])
30  hist=cnn.fit(x_train,y_train,batch_size=128,epochs=30,validation_data=(x_test,
    y_test),verbose=2)
31
32  # 신경망 모델 정확률 평가
33  res=cnn.evaluate(x_test,y_test,verbose=0)
34  print("정확률은",res[1]*100)
35
```

CNN Implementation

- LeNet Implementation

```
36  import matplotlib.pyplot as plt
37
38  # 정확률 그래프
39  plt.plot(hist.history['accuracy'])
40  plt.plot(hist.history['val_accuracy'])
41  plt.title('Model accuracy')
42  plt.ylabel('Accuracy')
43  plt.xlabel('Epoch')
44  plt.legend(['Train','Validation'],loc='best')
45  plt.grid()
46  plt.show()
47
48  # 손실 함수 그래프
49  plt.plot(hist.history['loss'])
50  plt.plot(hist.history['val_loss'])
51  plt.title('Model loss')
52  plt.ylabel('Loss')
53  plt.xlabel('Epoch')
54  plt.legend(['Train','Validation'],loc='best')
55  plt.grid()
56  plt.show()
```

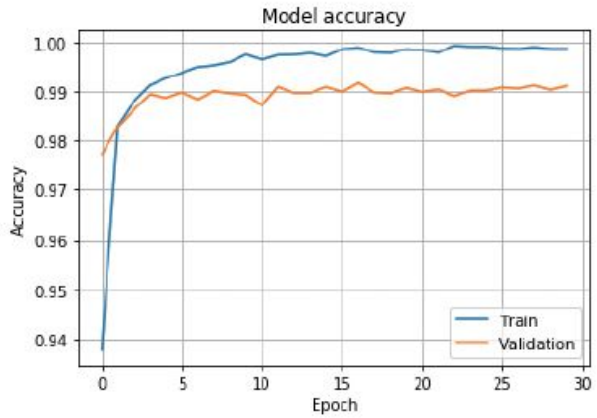
CNN Implementation

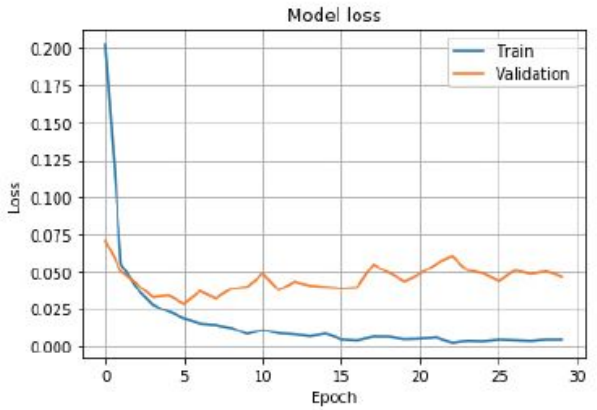
- LeNet Implementation

•Result

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 - 29s - loss: 0.2021 - accuracy: 0.9378 - val_loss: 0.0708 - val_accuracy: 0.9772
Epoch 2/30
60000/60000 - 31s - loss: 0.0552 - accuracy: 0.9830 - val_loss: 0.0501 - val_accuracy: 0.9828
Epoch 3/30
60000/60000 - 49s - loss: 0.0385 - accuracy: 0.9879 - val_loss: 0.0417 - val_accuracy: 0.9862
...
Epoch 29/30
60000/60000 - 30s - loss: 0.0043 - accuracy: 0.9987 - val_loss: 0.0500 - val_accuracy: 0.9903
Epoch 30/30
60000/60000 - 32s - loss: 0.0044 - accuracy: 0.9987 - val_loss: 0.0465 - val_accuracy: 0.9911
```

정확률은 99.1100013256073





Computational Thinking & Artificial Intelligence

Thanks for your attention.

Eunsom Jeon
Dept. of Computer Science & Engineering
ejeon6@seoultech.ac.kr