An MLP exampel : MNIST dataset

Import packages

In [ ]:
```
import numpy as np #numpy는 수치해석용 파이썬 라이브러리로 np라는 축약어로 import한
import pandas as pd # 데이터 처리를 위한 라이브러리이다. pd라는 축약어로 import한다
import tensorflow as tf #Tensorflow 2.0 부터 Keras가 내부 패키지로 포함 되었다. 기존
from tensorflow.keras.layers import Dense, Flatten # tensorflow의 서브패키지인 kera
from tensorflow.keras.models import Sequential #tensorflow의 서브패키지인 keras로 부
import matplotlib.pyplot as plt #matplotlib은 자료를 차트나 그래프로 시각화하는 라이
```

Download dataset : mnist

In [ ]:
```
# 여러 dataet이 저장되어있는 keras의 서브패키지인 datasets로 부터 mnist를 import한다
mnist_dataset = tf.keras.datasets.mnist

# 전체 6만개 데이터 중, 5만개는 학습 데이터용, 1만개는 테스트 데이터용으로 분리하여
(x_train_full, y_train_full), (x_test, y_test) = mnist_dataset.load_data()

# image size는 28x28 pixel의 grayscale 2차원 데이터
print("train dataset shape:", x_train_full.shape)
print("test dataset shape:", x_test.shape)
```

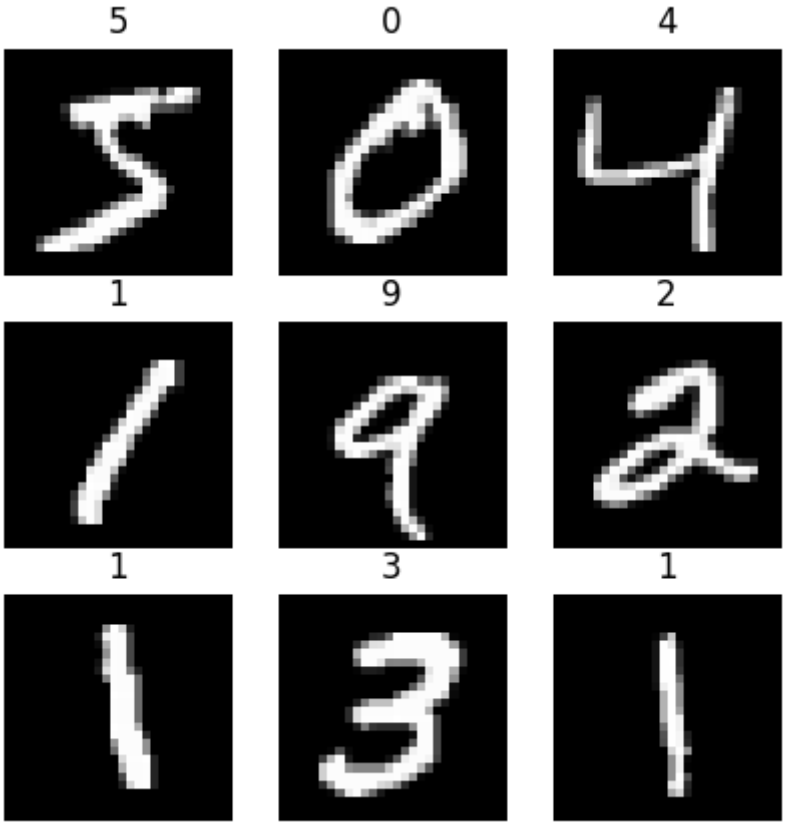
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mn
ist.npz
11490434/11490434 [==============================] - 0s 0us/step
train dataset shape: (60000, 28, 28)
test dataset shape: (10000, 28, 28)
```

Split and visualize dataset

In [ ]:
```
 # 이미지 array의 0~255 사이의 값을 0~1 사이값으로 scale 및 float32 형 변형해주는 함
def preprocessing_data(images):

    images = np.array(images/255.0, dtype=np.float32)
    return images
```

In [ ]:
```
# x_train_full 에서 앞의 5000개(인덱싱은 0~4999로 적용된다) 데이터는 validation set으
x_valid, x_train = preprocessing_data(x_train_full[:5000]), preprocessing_data(x_tra
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
x_test = x_test/255
```

In [ ]:
```
# plot이 그려지는 window의 사이즈를 x축 5, y축 5로 지정한다. (단위는 inch이다)
plt.figure(figsize=(5, 5))
for i in range(9):
    ax = plt.subplot(3, 3, i + 1) #3x3 subplot을 만들고 이미지의 위치를 i가 증가함에
    plt.imshow(x_train_full[i], cmap='gray') # grayscale의 이미지로 시각화해준다.
    plt.title(int(y_train_full[i])) #각 plot의 제목을 지정해준다
    plt.axis("off") # 축에 관한 함수로 "off" 로 지정하면 x축, y축을 표시하지 않는다.
```



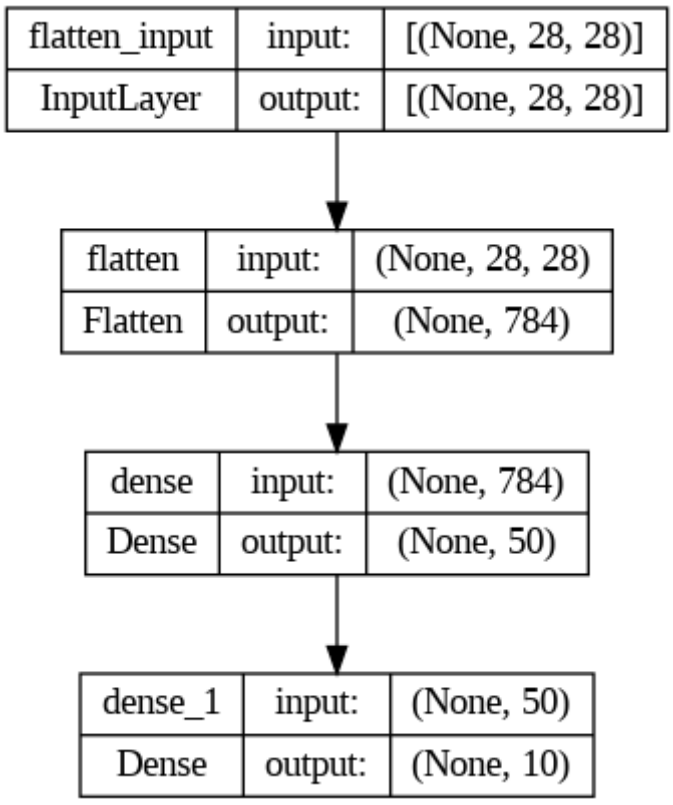
Build a model

In [ ]:
```
'''
    모델을 선언하는 과정이다.
    keras.Sequential은 선언된 각 레이어를 순서대로 실행시켜 주는 함수이며 아래와 같이
'''

INPUT_SIZE = 28

model = Sequential([

    # 가장 먼저 input 차원을 정의한다. input 이미지의 크기가 28*28 이므로 (28,28)을
    # Flatten layer 에서 28*28의 image array를 flatten 시켜서 1*784 로 입력받는다
    Flatten(input_shape=[INPUT_SIZE, INPUT_SIZE]),

    # 50개의 node를 가지는 Dense layer를 정의한다. dense layer이므로 각 뉴런은 이전
    # 총 parameter의 수는 784개의 출력값이 노드 50개와 fully connected 되어있으므로
    Dense(50, activation="relu"),

    # 10개의 node를 가지는 Dense layer를 정의한다. dense layer이므로 각 뉴런은 이전
    # 50개의 node로 부터의 출력값이 이 dense layer의 input 값이 되므로 총 parameter
    Dense(10, activation="softmax")
])

# 모델 구성에 대해서 확인할 수 있게 기본으로 제공해 주는 함수이다.
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 50)                39250

 dense_1 (Dense)             (None, 10)                510

=================================================================
Total params: 39,760
Trainable params: 39,760
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
# 모델 구성에 대해서 확인할 수 있게 기본으로 제공해 주는 함수이다.
tf.keras.utils.plot_model(model, "mnist_model.png", show_shapes=True)
```

Out[ ]:



Training the model

In [ ]:
```
'''
    구성한 모델을 이용하여 실제로 학습을 진행하는 과정이다.
'''

# 한 배치에 들어가는 데이터의 개수를 정의한다. 너무 많은 데이터를 한 배치에 넣으면 메
BATCH_SIZE = 32

# train 데이터를 전부 활용하는 것을 1 epoch라고 한다. 25라고 선언해 줌으로서 train 데
EPOCHS = 25

# 모델을 컴파일 하는 단계이다. 모델을 평가하는 과정에서 사용되는 loss와 metrics를 정
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="adam",
              metrics=["accuracy"])

# 모델이 실제로 학습을 시작하게 하는 함수이다.
```

```
# 학습에 사용되는 x, y 데이터를 입력하고, batch_size와 epochs를 정의 할 수 있다.
# validation은 미리 정의해둔 validation dataset을 사용하면 된다. 또는 비율을 지정해
history = model.fit(x_train, y_train, epochs=EPOCHS, batch_size = BATCH_SIZE,
                    validation_data=(x_valid, y_valid))
```

```
Epoch 1/25
1719/1719 [==============================] - 17s 8ms/step - loss: 0.3387 - accuracy:
0.9037 - val_loss: 0.1979 - val_accuracy: 0.9446
Epoch 2/25
1719/1719 [==============================] - 10s 6ms/step - loss: 0.1770 - accuracy:
0.9487 - val_loss: 0.1457 - val_accuracy: 0.9574
Epoch 3/25
1719/1719 [==============================] - 4s 2ms/step - loss: 0.1316 - accuracy:
0.9612 - val_loss: 0.1325 - val_accuracy: 0.9602
Epoch 4/25
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1047 - accuracy:
0.9686 - val_loss: 0.1067 - val_accuracy: 0.9658
Epoch 5/25
1719/1719 [==============================] - 4s 2ms/step - loss: 0.0869 - accuracy:
0.9735 - val_loss: 0.0938 - val_accuracy: 0.9708
Epoch 6/25
1719/1719 [==============================] - 4s 2ms/step - loss: 0.0737 - accuracy:
0.9773 - val_loss: 0.0879 - val_accuracy: 0.9740
Epoch 7/25
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0613 - accuracy:
0.9812 - val_loss: 0.0977 - val_accuracy: 0.9712
Epoch 8/25
1719/1719 [==============================] - 4s 2ms/step - loss: 0.0541 - accuracy:
0.9831 - val_loss: 0.0877 - val_accuracy: 0.9746
Epoch 9/25
1719/1719 [==============================] - 4s 2ms/step - loss: 0.0467 - accuracy:
0.9855 - val_loss: 0.0880 - val_accuracy: 0.9732
Epoch 10/25
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0414 - accuracy:
0.9873 - val_loss: 0.0877 - val_accuracy: 0.9738
Epoch 11/25
1719/1719 [==============================] - 4s 2ms/step - loss: 0.0366 - accuracy:
0.9889 - val_loss: 0.0888 - val_accuracy: 0.9760
Epoch 12/25
1719/1719 [==============================] - 4s 2ms/step - loss: 0.0323 - accuracy:
0.9899 - val_loss: 0.0834 - val_accuracy: 0.9756
Epoch 13/25
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0287 - accuracy:
0.9915 - val_loss: 0.0867 - val_accuracy: 0.9744
Epoch 14/25
1719/1719 [==============================] - 4s 2ms/step - loss: 0.0262 - accuracy:
0.9916 - val_loss: 0.0956 - val_accuracy: 0.9738
Epoch 15/25
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0234 - accuracy:
0.9930 - val_loss: 0.0914 - val_accuracy: 0.9748
Epoch 16/25
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0208 - accuracy:
0.9939 - val_loss: 0.0950 - val_accuracy: 0.9744
Epoch 17/25
1719/1719 [==============================] - 4s 2ms/step - loss: 0.0184 - accuracy:
0.9945 - val_loss: 0.0979 - val_accuracy: 0.9738
Epoch 18/25
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0169 - accuracy:
0.9952 - val_loss: 0.0989 - val_accuracy: 0.9758
Epoch 19/25
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0160 - accuracy:
0.9953 - val_loss: 0.0984 - val_accuracy: 0.9768
Epoch 20/25
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0136 - accuracy:
0.9961 - val_loss: 0.1070 - val_accuracy: 0.9726
Epoch 21/25
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0120 - accuracy:
0.9967 - val_loss: 0.1093 - val_accuracy: 0.9734
Epoch 22/25
```

```
1719/1719 [==============================] - 4s 3ms/step - loss: 0.0118 - accuracy:
0.9965 - val_loss: 0.1077 - val_accuracy: 0.9740
Epoch 23/25
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0108 - accuracy:
0.9967 - val_loss: 0.1011 - val_accuracy: 0.9738
Epoch 24/25
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0092 - accuracy:
0.9977 - val_loss: 0.1104 - val_accuracy: 0.9740
Epoch 25/25
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0092 - accuracy:
0.9975 - val_loss: 0.1025 - val_accuracy: 0.9764
```

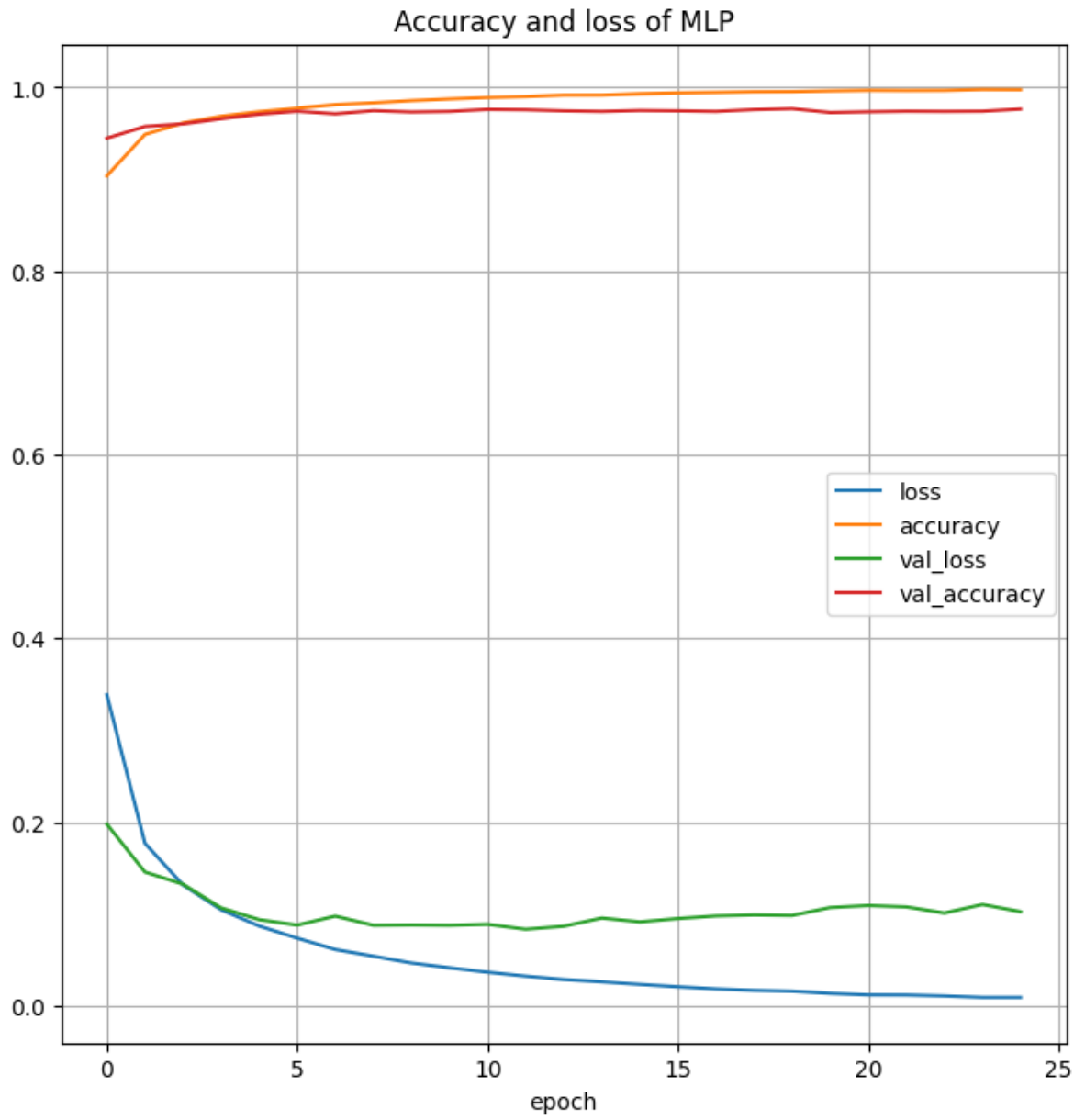
In [ ]:
```
# 학습 후 어떤 key값들이 저장되었는지 확인 할 수 있다.
history.history.keys()
```

Out[ ]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

Plot the result

In [ ]:
```
#plt.plot(history.history['loss'])
#plt.plot(history.history['accuracy'])
#plt.plot(history.history['val_loss'])
#plt.plot(history.history['val_accuracy'])

pd.DataFrame(history.history).plot(figsize=(8, 8)) #plot keys all at once
plt.grid(True) ;plt.title("Accuracy and loss of MLP");plt.xlabel("epoch")
```

Out[ ]:
```
Text(0.5, 0, 'epoch')
```



Testing the model

In [ ]:
```
'''
    학습이 종료된 모델을 평가하는 단계이다.
    데이터를 load 하는 단계부터 미리 분리하여 학습에 전혀 사용되지 않은 test 데이터셋
'''


# test dataset을 사용하여 model을 평가하고 accuracy와 loss를 출력한다
score = model.evaluate(x_test, y_test)

print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.1124 - accuracy: 0.
9757
Test loss: 0.1124197393655777
Test accuracy: 0.9757000207901001
```

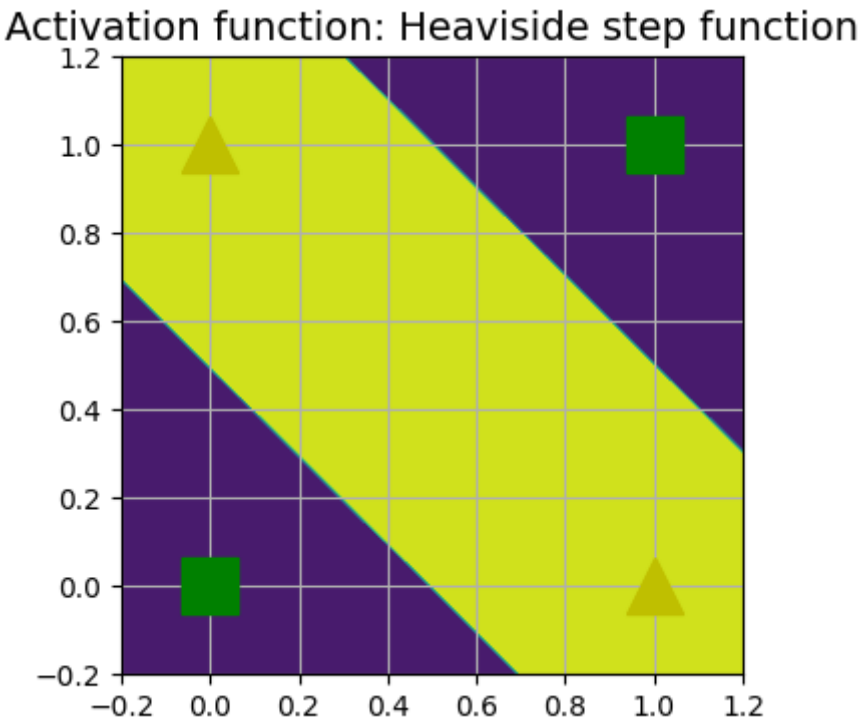
An MLP example : XOR problem

Import packages

In [ ]:
```
#numpy는 수치 연산에 효율적이고, pandas는 데이터 조작 및 분석 등의 작업에 효율적이다
import numpy as np #numpy는 수치해석용 파이썬 라이브러리로 np라는 축약어로 import한
import pandas as pd # 데이터 처리를 위한 라이브러리이다. pd라는 축약어로 import한다
import matplotlib.pyplot as plt #matplotlib은 자료를 차트나 그래프로 시각화하는 라
```

In [ ]:
```
# def를 사용하여 사용자 정의함수를 만들 수 있다. 이 예제에서는 heaviside step functi
# 이 함수의 이름은 step이고, 'z' 입력값을 주었을 때 '(z >= 0).astype(z.dtype)' 값을
def step(z):
    return (z >= 0).astype(z.dtype)

def mlp_xor(x1, x2, activation=step):
    return activation(-activation(x1 + x2 - 1.5) + activation(x1 + x2 - 0.5) - 0.5
```

In [ ]:
```
# numpy 라이브러리에 있는 함수로써 1차원 배열을 만들 수 있고, 그래프를 그릴 때 수평축
# 예를 들어 np.linspace(-0.2, 1.2, 100)은 -0.2부터 1.2까지의 숫자를 길이 100짜리의 배
x1s = np.linspace(-0.2, 1.2, 100)
x2s = np.linspace(-0.2, 1.2, 100)
x1, x2 = np.meshgrid(x1s, x2s) # 그리드를 생성해주는 함수이다


z1 = mlp_xor(x1, x2, activation=step)

# plot이 그려지는 window의 사이즈를 x축 4, y축 4로 지정한다. (단위는 inch이다)
plt.figure(figsize=(4,4))

# contour과 contourf는 색을 칠해준다는 차이점이 있다.
# contour그래프를 그리기 위해서는 1) x,y의 meshgrid를 형성해준다. 2) 해당 mesh의 각각
plt.contourf(x1, x2, z1)
plt.plot([0, 1], [0, 1], "gs", markersize=20)
plt.plot([0, 1], [1, 0], "y^", markersize=20)
plt.title("Activation function: Heaviside step function", fontsize=14) # 그래프의 제

plt.grid(True)
```


Activation function: Heaviside step function
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2

An MLP example : Iris dataset

Import packages

In [12]:
```
import matplotlib.pyplot as plt # matplotlib은 자료를 차트나 그래프로 시각화하는 라
import numpy as np # numpy는 수치해석용 파이썬 라이브러리로 np라는 축약어로 import한
import pandas as pd # 데이터 처리를 위한 라이브러리이다. pd라는 축약어로 import한다
import tensorflow as tf # Tensorflow 2.0 부터 Keras가 내부 패키지로 포함 되었다. 기
from tensorflow.keras.layers import Dense, Flatten, Dropout # tensorflow의 서브패키
from keras.utils import * # keras의 서브패키지인 utils의 모든 함수를 import한다. (*
from sklearn.preprocessing import LabelEncoder # sklearn 라이브러리의 서브패키지인
from sklearn.model_selection import train_test_split # sklearn 라이브러리의 서브패
import tensorflow as tf #Tensorflow 2.0 부터 Keras가 내부 패키지로 포함 되었다. 기
from tensorflow.keras.layers import Dense, Flatten # tensorflow의 서브패키지인 kera
from tensorflow.keras.models import Sequential #tensorflow의 서브패키지인 keras로 부
from sklearn.datasets import load_iris # sklearn 라이브러리의 dataset이 모여있는 서
```

Download dataset : iris(붓꽃)

In [13]:
```
Iris = load_iris() # iris(붓꽃) 데이터 로드
```

In [14]:
```
# feature_names 와 target을 레코드로 갖는 데이터프레임 생성
Iris_Data = pd.DataFrame(data = Iris.data, columns = Iris.feature_names)
Iris_Data['Species'] = Iris.target
```

Split and visualize dataset

In [15]:
```
# 앞의 5개열 데이터를 보여준다. default값은 5이고, ()안에 보고싶은 열의 개수를 쓰면
Iris_Data.head()
```

Out[15]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	Species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

In [16]:
```
# label을 기준으로 인덱싱 해준 후(loc) dataframe 을 numpy 배열로 변환해준다.
X=Iris_Data.loc[:,['sepal length (cm)','sepal width (cm)','petal length (cm)','petal

 # 레이블 인코딩을 쉽게 해주는 레이블인코더(LabelEncoder)라는 함수를 통해 Irsr_Data의
 # 총 세 종류 이므로 [0, 0, 1] [0, 1, 0] [1, 0, 0] 으로 인코딩 된다.
encoder =  LabelEncoder()
y1 = encoder.fit_transform(Iris_Data['Species'])
Y = pd.get_dummies(y1).values

# train_test_split 함수를 통해 20%의 데이터를 test 데이터로 분류하였다.
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2)
# 위에서 분류한 test 데이터 중 20%를 validation 데이터로 분류하였다.
X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train, test_size=0.
```

```
# 데이터의 차원을 확인하였다. X_train.shape, X_test.shape, y_train.shape, y_test.sha
```

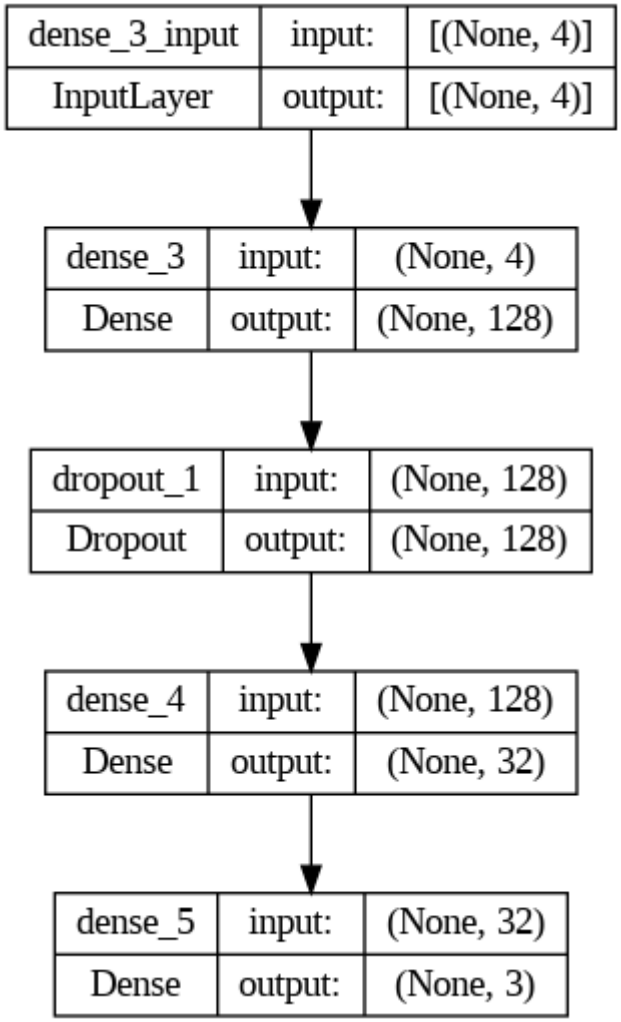
Build a model

In [17]:
```
'''
    모델을 선언하는 과정이다.
    keras.Sequential은 선언된 각 레이어를 순서대로 실행시켜 주는 함수이며 아래와 같이
'''

model = Sequential([

  # 128개의 node를 가지는 Dense layer를 정의한다. dense layer이므로 각 뉴런은 이전 계
  # 4개의 input을 받으므로 총 parameter 수는 640(512(128x4) + 128(bias) = 640)개 이다
  Dense(128, activation='relu',input_shape=(4,)),

  # input 데이터에 10%의 노드들을 무작위로 0으로 만드는 드롭아웃을 적용한다.
  Dropout(rate=0.1),

  # 32개의 node를 가지는 Dense layer를 정의한다. dense layer이므로 각 뉴런은 이전 계
  # 128개의 node로 부터의 출력값이 이 dense layer의 input 값이 되므로 총 parameter 수
  Dense(32, activation='relu'),

  # output 값이 3개 이므로 3개의 node를 가지는 Dense layer를 정의한다. dense layer이
  Dense(3, activation='softmax')
])

# 모델 구성에 대해서 확인할 수 있게 기본으로 제공해 주는 함수이다.
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_3 (Dense)             (None, 128)               640

 dropout_1 (Dropout)         (None, 128)               0

 dense_4 (Dense)             (None, 32)                4128

 dense_5 (Dense)             (None, 3)                 99

=================================================================
Total params: 4,867
Trainable params: 4,867
Non-trainable params: 0
_________________________________________________________________
```

In [18]:
```
# 모델 구성에 대해서 확인할 수 있게 기본으로 제공해 주는 함수이다.
tf.keras.utils.plot_model(model, "mnist_model.png", show_shapes=True)
```

Out[18]:



Training the model

In [19]:

```
'''
    구성한 모델을 이용하여 실제로 학습을 진행하는 과정이다.
'''

# 한 배치에 들어가는 데이터의 개수를 정의한다. 너무 많은 데이터를 한 배치에 넣으면 메
BATCH_SIZE = 10

# train 데이터를 전부 활용하는 것을 1 epoch라고 한다. 25라고 선언해 줌으로서 train 데
EPOCHS = 200

# 모델을 컴파일 하는 단계이다. 모델을 평가하는 과정에서 사용되는 loss와 metrics를 정
# 원 핫 인코딩을 한 경우에는 loss = 'categorical_crossentropy' 를 써준다.
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# 모델이 실제로 학습을 시작하게 하는 함수이다.
# 학습에 사용되는 x, y 데이터를 입력하고, batch_size와 epochs를 정의 할 수 있다.
# validation은 미리 정의해둔 validation dataset을 사용하면 된다. 또는 비율을 지정해
history = model.fit(X_train, y_train, validation_data=(X_valid, y_valid),
                    epochs=EPOCHS, batch_size = BATCH_SIZE)
```

```
Epoch 1/200
10/10 [==============================] - 2s 33ms/step - loss: 1.1400 - accuracy: 0.3
542 - val_loss: 1.0037 - val_accuracy: 0.2917
Epoch 2/200
10/10 [==============================] - 0s 9ms/step - loss: 0.9588 - accuracy: 0.45
83 - val_loss: 0.8665 - val_accuracy: 0.6667
Epoch 3/200
10/10 [==============================] - 0s 9ms/step - loss: 0.8276 - accuracy: 0.72
92 - val_loss: 0.7740 - val_accuracy: 0.7500
Epoch 4/200
10/10 [==============================] - 0s 9ms/step - loss: 0.7349 - accuracy: 0.66
67 - val_loss: 0.6928 - val_accuracy: 0.5833
Epoch 5/200
10/10 [==============================] - 0s 7ms/step - loss: 0.6109 - accuracy: 0.69
79 - val_loss: 0.6086 - val_accuracy: 0.5833
Epoch 6/200
10/10 [==============================] - 0s 9ms/step - loss: 0.5597 - accuracy: 0.70
83 - val_loss: 0.5590 - val_accuracy: 0.7083
Epoch 7/200
10/10 [==============================] - 0s 10ms/step - loss: 0.5083 - accuracy: 0.7
396 - val_loss: 0.5287 - val_accuracy: 0.7083
Epoch 8/200
10/10 [==============================] - 0s 9ms/step - loss: 0.4868 - accuracy: 0.69
79 - val_loss: 0.5187 - val_accuracy: 0.5833
Epoch 9/200
10/10 [==============================] - 0s 9ms/step - loss: 0.4468 - accuracy: 0.73
96 - val_loss: 0.4675 - val_accuracy: 0.9583
Epoch 10/200
10/10 [==============================] - 0s 9ms/step - loss: 0.4354 - accuracy: 0.82
29 - val_loss: 0.4477 - val_accuracy: 0.9583
Epoch 11/200
10/10 [==============================] - 0s 7ms/step - loss: 0.3971 - accuracy: 0.85
42 - val_loss: 0.4400 - val_accuracy: 0.8333
Epoch 12/200
10/10 [==============================] - 0s 9ms/step - loss: 0.3891 - accuracy: 0.76
04 - val_loss: 0.4118 - val_accuracy: 0.9583
Epoch 13/200
10/10 [==============================] - 0s 8ms/step - loss: 0.3801 - accuracy: 0.85
42 - val_loss: 0.3867 - val_accuracy: 0.9583
Epoch 14/200
10/10 [==============================] - 0s 8ms/step - loss: 0.3789 - accuracy: 0.88
54 - val_loss: 0.3636 - val_accuracy: 0.9583
Epoch 15/200
10/10 [==============================] - 0s 9ms/step - loss: 0.3463 - accuracy: 0.92
71 - val_loss: 0.3435 - val_accuracy: 1.0000
Epoch 16/200
10/10 [==============================] - 0s 8ms/step - loss: 0.3270 - accuracy: 0.94
79 - val_loss: 0.3275 - val_accuracy: 0.9583
Epoch 17/200
10/10 [==============================] - 0s 8ms/step - loss: 0.3097 - accuracy: 0.88
54 - val_loss: 0.3072 - val_accuracy: 0.9583
Epoch 18/200
10/10 [==============================] - 0s 8ms/step - loss: 0.3139 - accuracy: 0.89
58 - val_loss: 0.2819 - val_accuracy: 1.0000
Epoch 19/200
10/10 [==============================] - 0s 9ms/step - loss: 0.2852 - accuracy: 0.94
79 - val_loss: 0.2713 - val_accuracy: 0.9583
Epoch 20/200
10/10 [==============================] - 0s 8ms/step - loss: 0.2932 - accuracy: 0.90
62 - val_loss: 0.2439 - val_accuracy: 1.0000
Epoch 21/200
10/10 [==============================] - 0s 7ms/step - loss: 0.2595 - accuracy: 0.95
83 - val_loss: 0.2283 - val_accuracy: 1.0000
Epoch 22/200
```

```
10/10 [==============================] - 0s 8ms/step - loss: 0.2618 - accuracy: 0.91
67 - val_loss: 0.2113 - val_accuracy: 1.0000
Epoch 23/200
10/10 [==============================] - 0s 7ms/step - loss: 0.2523 - accuracy: 0.95
83 - val_loss: 0.1995 - val_accuracy: 1.0000
Epoch 24/200
10/10 [==============================] - 0s 7ms/step - loss: 0.2501 - accuracy: 0.93
75 - val_loss: 0.1833 - val_accuracy: 1.0000
Epoch 25/200
10/10 [==============================] - 0s 8ms/step - loss: 0.2442 - accuracy: 0.93
75 - val_loss: 0.1717 - val_accuracy: 1.0000
Epoch 26/200
10/10 [==============================] - 0s 7ms/step - loss: 0.2002 - accuracy: 0.96
88 - val_loss: 0.1606 - val_accuracy: 1.0000
Epoch 27/200
10/10 [==============================] - 0s 8ms/step - loss: 0.2191 - accuracy: 0.95
83 - val_loss: 0.1473 - val_accuracy: 1.0000
Epoch 28/200
10/10 [==============================] - 0s 7ms/step - loss: 0.2160 - accuracy: 0.92
71 - val_loss: 0.1387 - val_accuracy: 1.0000
Epoch 29/200
10/10 [==============================] - 0s 8ms/step - loss: 0.2151 - accuracy: 0.90
62 - val_loss: 0.1314 - val_accuracy: 1.0000
Epoch 30/200
10/10 [==============================] - 0s 7ms/step - loss: 0.2151 - accuracy: 0.94
79 - val_loss: 0.1317 - val_accuracy: 1.0000
Epoch 31/200
10/10 [==============================] - 0s 8ms/step - loss: 0.1834 - accuracy: 0.94
79 - val_loss: 0.1133 - val_accuracy: 1.0000
Epoch 32/200
10/10 [==============================] - 0s 7ms/step - loss: 0.1852 - accuracy: 0.94
79 - val_loss: 0.1088 - val_accuracy: 1.0000
Epoch 33/200
10/10 [==============================] - 0s 9ms/step - loss: 0.1459 - accuracy: 0.97
92 - val_loss: 0.1016 - val_accuracy: 1.0000
Epoch 34/200
10/10 [==============================] - 0s 7ms/step - loss: 0.1795 - accuracy: 0.92
71 - val_loss: 0.0946 - val_accuracy: 1.0000
Epoch 35/200
10/10 [==============================] - 0s 8ms/step - loss: 0.1462 - accuracy: 0.96
88 - val_loss: 0.0883 - val_accuracy: 1.0000
Epoch 36/200
10/10 [==============================] - 0s 9ms/step - loss: 0.1626 - accuracy: 0.92
71 - val_loss: 0.0837 - val_accuracy: 1.0000
Epoch 37/200
10/10 [==============================] - 0s 12ms/step - loss: 0.1538 - accuracy: 0.9
792 - val_loss: 0.0780 - val_accuracy: 1.0000
Epoch 38/200
10/10 [==============================] - 0s 11ms/step - loss: 0.1298 - accuracy: 0.9
688 - val_loss: 0.0734 - val_accuracy: 1.0000
Epoch 39/200
10/10 [==============================] - 0s 12ms/step - loss: 0.1893 - accuracy: 0.9
375 - val_loss: 0.0710 - val_accuracy: 1.0000
Epoch 40/200
10/10 [==============================] - 0s 14ms/step - loss: 0.1847 - accuracy: 0.9
271 - val_loss: 0.0689 - val_accuracy: 1.0000
Epoch 41/200
10/10 [==============================] - 0s 12ms/step - loss: 0.1297 - accuracy: 0.9
479 - val_loss: 0.0644 - val_accuracy: 1.0000
Epoch 42/200
10/10 [==============================] - 0s 12ms/step - loss: 0.1564 - accuracy: 0.9
479 - val_loss: 0.0643 - val_accuracy: 1.0000
Epoch 43/200
10/10 [==============================] - 0s 9ms/step - loss: 0.1482 - accuracy: 0.91
```

```
67 - val_loss: 0.0603 - val_accuracy: 1.0000
Epoch 44/200
10/10 [==============================] - 0s 10ms/step - loss: 0.1198 - accuracy: 0.9
688 - val_loss: 0.0693 - val_accuracy: 1.0000
Epoch 45/200
10/10 [==============================] - 0s 9ms/step - loss: 0.1538 - accuracy: 0.95
83 - val_loss: 0.0544 - val_accuracy: 1.0000
Epoch 46/200
10/10 [==============================] - 0s 10ms/step - loss: 0.1320 - accuracy: 0.9
583 - val_loss: 0.0527 - val_accuracy: 1.0000
Epoch 47/200
10/10 [==============================] - 0s 10ms/step - loss: 0.1429 - accuracy: 0.9
479 - val_loss: 0.0510 - val_accuracy: 1.0000
Epoch 48/200
10/10 [==============================] - 0s 13ms/step - loss: 0.1684 - accuracy: 0.9
271 - val_loss: 0.0509 - val_accuracy: 1.0000
Epoch 49/200
10/10 [==============================] - 0s 11ms/step - loss: 0.1285 - accuracy: 0.9
688 - val_loss: 0.0467 - val_accuracy: 1.0000
Epoch 50/200
10/10 [==============================] - 0s 11ms/step - loss: 0.1184 - accuracy: 0.9
583 - val_loss: 0.0454 - val_accuracy: 1.0000
Epoch 51/200
10/10 [==============================] - 0s 12ms/step - loss: 0.1361 - accuracy: 0.9
583 - val_loss: 0.0519 - val_accuracy: 1.0000
Epoch 52/200
10/10 [==============================] - 0s 10ms/step - loss: 0.1329 - accuracy: 0.9
271 - val_loss: 0.0435 - val_accuracy: 1.0000
Epoch 53/200
10/10 [==============================] - 0s 11ms/step - loss: 0.1436 - accuracy: 0.9
271 - val_loss: 0.0471 - val_accuracy: 1.0000
Epoch 54/200
10/10 [==============================] - 0s 10ms/step - loss: 0.1297 - accuracy: 0.9
375 - val_loss: 0.0435 - val_accuracy: 1.0000
Epoch 55/200
10/10 [==============================] - 0s 11ms/step - loss: 0.1112 - accuracy: 0.9
792 - val_loss: 0.0408 - val_accuracy: 1.0000
Epoch 56/200
10/10 [==============================] - 0s 10ms/step - loss: 0.0870 - accuracy: 0.9
792 - val_loss: 0.0466 - val_accuracy: 1.0000
Epoch 57/200
10/10 [==============================] - 0s 12ms/step - loss: 0.1303 - accuracy: 0.9
479 - val_loss: 0.0383 - val_accuracy: 1.0000
Epoch 58/200
10/10 [==============================] - 0s 12ms/step - loss: 0.1196 - accuracy: 0.9
479 - val_loss: 0.0385 - val_accuracy: 1.0000
Epoch 59/200
10/10 [==============================] - 0s 13ms/step - loss: 0.1215 - accuracy: 0.9
375 - val_loss: 0.0483 - val_accuracy: 1.0000
Epoch 60/200
10/10 [==============================] - 0s 10ms/step - loss: 0.1341 - accuracy: 0.9
375 - val_loss: 0.0358 - val_accuracy: 1.0000
Epoch 61/200
10/10 [==============================] - 0s 15ms/step - loss: 0.1287 - accuracy: 0.9
271 - val_loss: 0.0428 - val_accuracy: 1.0000
Epoch 62/200
10/10 [==============================] - 0s 29ms/step - loss: 0.1415 - accuracy: 0.9
479 - val_loss: 0.0346 - val_accuracy: 1.0000
Epoch 63/200
10/10 [==============================] - 0s 11ms/step - loss: 0.1229 - accuracy: 0.9
583 - val_loss: 0.0343 - val_accuracy: 1.0000
Epoch 64/200
10/10 [==============================] - 0s 9ms/step - loss: 0.1125 - accuracy: 0.93
75 - val_loss: 0.0334 - val_accuracy: 1.0000
```

```
Epoch 65/200
10/10 [==============================] - 0s 24ms/step - loss: 0.1215 - accuracy: 0.9
479 - val_loss: 0.0356 - val_accuracy: 1.0000
Epoch 66/200
10/10 [==============================] - 0s 21ms/step - loss: 0.1389 - accuracy: 0.9
271 - val_loss: 0.0376 - val_accuracy: 1.0000
Epoch 67/200
10/10 [==============================] - 0s 20ms/step - loss: 0.1056 - accuracy: 0.9
479 - val_loss: 0.0368 - val_accuracy: 1.0000
Epoch 68/200
10/10 [==============================] - 0s 17ms/step - loss: 0.1633 - accuracy: 0.9
479 - val_loss: 0.0355 - val_accuracy: 1.0000
Epoch 69/200
10/10 [==============================] - 0s 20ms/step - loss: 0.1291 - accuracy: 0.9
583 - val_loss: 0.0336 - val_accuracy: 1.0000
Epoch 70/200
10/10 [==============================] - 0s 24ms/step - loss: 0.1136 - accuracy: 0.9
688 - val_loss: 0.0383 - val_accuracy: 1.0000
Epoch 71/200
10/10 [==============================] - 0s 21ms/step - loss: 0.1195 - accuracy: 0.9
583 - val_loss: 0.0310 - val_accuracy: 1.0000
Epoch 72/200
10/10 [==============================] - 0s 12ms/step - loss: 0.1155 - accuracy: 0.9
479 - val_loss: 0.0331 - val_accuracy: 1.0000
Epoch 73/200
10/10 [==============================] - 0s 21ms/step - loss: 0.1381 - accuracy: 0.9
271 - val_loss: 0.0308 - val_accuracy: 1.0000
Epoch 74/200
10/10 [==============================] - 0s 32ms/step - loss: 0.1173 - accuracy: 0.9
583 - val_loss: 0.0390 - val_accuracy: 1.0000
Epoch 75/200
10/10 [==============================] - 0s 18ms/step - loss: 0.1187 - accuracy: 0.9
583 - val_loss: 0.0318 - val_accuracy: 1.0000
Epoch 76/200
10/10 [==============================] - 0s 18ms/step - loss: 0.1165 - accuracy: 0.9
479 - val_loss: 0.0304 - val_accuracy: 1.0000
Epoch 77/200
10/10 [==============================] - 0s 13ms/step - loss: 0.1023 - accuracy: 0.9
583 - val_loss: 0.0291 - val_accuracy: 1.0000
Epoch 78/200
10/10 [==============================] - 0s 13ms/step - loss: 0.0944 - accuracy: 0.9
792 - val_loss: 0.0302 - val_accuracy: 1.0000
Epoch 79/200
10/10 [==============================] - 0s 30ms/step - loss: 0.0934 - accuracy: 0.9
688 - val_loss: 0.0305 - val_accuracy: 1.0000
Epoch 80/200
10/10 [==============================] - 0s 33ms/step - loss: 0.0926 - accuracy: 0.9
688 - val_loss: 0.0279 - val_accuracy: 1.0000
Epoch 81/200
10/10 [==============================] - 0s 26ms/step - loss: 0.1016 - accuracy: 0.9
688 - val_loss: 0.0274 - val_accuracy: 1.0000
Epoch 82/200
10/10 [==============================] - 0s 31ms/step - loss: 0.1279 - accuracy: 0.9
479 - val_loss: 0.0333 - val_accuracy: 1.0000
Epoch 83/200
10/10 [==============================] - 0s 34ms/step - loss: 0.1052 - accuracy: 0.9
479 - val_loss: 0.0291 - val_accuracy: 1.0000
Epoch 84/200
10/10 [==============================] - 0s 23ms/step - loss: 0.1206 - accuracy: 0.9
583 - val_loss: 0.0250 - val_accuracy: 1.0000
Epoch 85/200
10/10 [==============================] - 0s 19ms/step - loss: 0.0978 - accuracy: 0.9
583 - val_loss: 0.0266 - val_accuracy: 1.0000
Epoch 86/200
```

```
10/10 [==============================] - 0s 11ms/step - loss: 0.0907 - accuracy: 0.9
792 - val_loss: 0.0253 - val_accuracy: 1.0000
Epoch 87/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0976 - accuracy: 0.9
688 - val_loss: 0.0242 - val_accuracy: 1.0000
Epoch 88/200
10/10 [==============================] - 0s 15ms/step - loss: 0.1014 - accuracy: 0.9
688 - val_loss: 0.0238 - val_accuracy: 1.0000
Epoch 89/200
10/10 [==============================] - 0s 16ms/step - loss: 0.0925 - accuracy: 0.9
688 - val_loss: 0.0232 - val_accuracy: 1.0000
Epoch 90/200
10/10 [==============================] - 0s 18ms/step - loss: 0.0985 - accuracy: 0.9
583 - val_loss: 0.0219 - val_accuracy: 1.0000
Epoch 91/200
10/10 [==============================] - 0s 29ms/step - loss: 0.1024 - accuracy: 0.9
583 - val_loss: 0.0222 - val_accuracy: 1.0000
Epoch 92/200
10/10 [==============================] - 0s 13ms/step - loss: 0.1028 - accuracy: 0.9
583 - val_loss: 0.0226 - val_accuracy: 1.0000
Epoch 93/200
10/10 [==============================] - 0s 10ms/step - loss: 0.0828 - accuracy: 0.9
688 - val_loss: 0.0259 - val_accuracy: 1.0000
Epoch 94/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0781 - accuracy: 0.9
792 - val_loss: 0.0214 - val_accuracy: 1.0000
Epoch 95/200
10/10 [==============================] - 0s 15ms/step - loss: 0.1064 - accuracy: 0.9
583 - val_loss: 0.0271 - val_accuracy: 1.0000
Epoch 96/200
10/10 [==============================] - 0s 13ms/step - loss: 0.0827 - accuracy: 0.9
792 - val_loss: 0.0239 - val_accuracy: 1.0000
Epoch 97/200
10/10 [==============================] - 0s 9ms/step - loss: 0.1078 - accuracy: 0.94
79 - val_loss: 0.0216 - val_accuracy: 1.0000
Epoch 98/200
10/10 [==============================] - 0s 14ms/step - loss: 0.1041 - accuracy: 0.9
688 - val_loss: 0.0286 - val_accuracy: 1.0000
Epoch 99/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0864 - accuracy: 0.9
792 - val_loss: 0.0212 - val_accuracy: 1.0000
Epoch 100/200
10/10 [==============================] - 0s 10ms/step - loss: 0.1032 - accuracy: 0.9
479 - val_loss: 0.0206 - val_accuracy: 1.0000
Epoch 101/200
10/10 [==============================] - 0s 13ms/step - loss: 0.0845 - accuracy: 0.9
792 - val_loss: 0.0195 - val_accuracy: 1.0000
Epoch 102/200
10/10 [==============================] - 0s 15ms/step - loss: 0.1054 - accuracy: 0.9
688 - val_loss: 0.0182 - val_accuracy: 1.0000
Epoch 103/200
10/10 [==============================] - 0s 13ms/step - loss: 0.1394 - accuracy: 0.9
271 - val_loss: 0.0180 - val_accuracy: 1.0000
Epoch 104/200
10/10 [==============================] - 0s 14ms/step - loss: 0.1085 - accuracy: 0.9
583 - val_loss: 0.0323 - val_accuracy: 1.0000
Epoch 105/200
10/10 [==============================] - 0s 12ms/step - loss: 0.1205 - accuracy: 0.9
479 - val_loss: 0.0207 - val_accuracy: 1.0000
Epoch 106/200
10/10 [==============================] - 0s 10ms/step - loss: 0.0947 - accuracy: 0.9
792 - val_loss: 0.0249 - val_accuracy: 1.0000
Epoch 107/200
10/10 [==============================] - 0s 8ms/step - loss: 0.1007 - accuracy: 0.96
```

```
88 - val_loss: 0.0225 - val_accuracy: 1.0000
Epoch 108/200
10/10 [==============================] - 0s 8ms/step - loss: 0.1461 - accuracy: 0.92
71 - val_loss: 0.0226 - val_accuracy: 1.0000
Epoch 109/200
10/10 [==============================] - 0s 7ms/step - loss: 0.1025 - accuracy: 0.96
88 - val_loss: 0.0264 - val_accuracy: 1.0000
Epoch 110/200
10/10 [==============================] - 0s 7ms/step - loss: 0.0823 - accuracy: 0.96
88 - val_loss: 0.0217 - val_accuracy: 1.0000
Epoch 111/200
10/10 [==============================] - 0s 9ms/step - loss: 0.0906 - accuracy: 0.96
88 - val_loss: 0.0207 - val_accuracy: 1.0000
Epoch 112/200
10/10 [==============================] - 0s 9ms/step - loss: 0.0864 - accuracy: 0.96
88 - val_loss: 0.0261 - val_accuracy: 1.0000
Epoch 113/200
10/10 [==============================] - 0s 9ms/step - loss: 0.1038 - accuracy: 0.95
83 - val_loss: 0.0206 - val_accuracy: 1.0000
Epoch 114/200
10/10 [==============================] - 0s 7ms/step - loss: 0.0891 - accuracy: 0.97
92 - val_loss: 0.0192 - val_accuracy: 1.0000
Epoch 115/200
10/10 [==============================] - 0s 7ms/step - loss: 0.1029 - accuracy: 0.95
83 - val_loss: 0.0193 - val_accuracy: 1.0000
Epoch 116/200
10/10 [==============================] - 0s 7ms/step - loss: 0.0928 - accuracy: 0.96
88 - val_loss: 0.0188 - val_accuracy: 1.0000
Epoch 117/200
10/10 [==============================] - 0s 6ms/step - loss: 0.0903 - accuracy: 0.95
83 - val_loss: 0.0245 - val_accuracy: 1.0000
Epoch 118/200
10/10 [==============================] - 0s 8ms/step - loss: 0.0868 - accuracy: 0.95
83 - val_loss: 0.0171 - val_accuracy: 1.0000
Epoch 119/200
10/10 [==============================] - 0s 9ms/step - loss: 0.1066 - accuracy: 0.96
88 - val_loss: 0.0188 - val_accuracy: 1.0000
Epoch 120/200
10/10 [==============================] - 0s 6ms/step - loss: 0.0832 - accuracy: 0.96
88 - val_loss: 0.0204 - val_accuracy: 1.0000
Epoch 121/200
10/10 [==============================] - 0s 7ms/step - loss: 0.0862 - accuracy: 0.95
83 - val_loss: 0.0167 - val_accuracy: 1.0000
Epoch 122/200
10/10 [==============================] - 0s 10ms/step - loss: 0.0860 - accuracy: 0.9
688 - val_loss: 0.0177 - val_accuracy: 1.0000
Epoch 123/200
10/10 [==============================] - 0s 7ms/step - loss: 0.1102 - accuracy: 0.96
88 - val_loss: 0.0166 - val_accuracy: 1.0000
Epoch 124/200
10/10 [==============================] - 0s 8ms/step - loss: 0.1012 - accuracy: 0.97
92 - val_loss: 0.0170 - val_accuracy: 1.0000
Epoch 125/200
10/10 [==============================] - 0s 6ms/step - loss: 0.1039 - accuracy: 0.96
88 - val_loss: 0.0196 - val_accuracy: 1.0000
Epoch 126/200
10/10 [==============================] - 0s 8ms/step - loss: 0.1019 - accuracy: 0.96
88 - val_loss: 0.0162 - val_accuracy: 1.0000
Epoch 127/200
10/10 [==============================] - 0s 8ms/step - loss: 0.0782 - accuracy: 0.96
88 - val_loss: 0.0173 - val_accuracy: 1.0000
Epoch 128/200
10/10 [==============================] - 0s 9ms/step - loss: 0.0936 - accuracy: 0.97
92 - val_loss: 0.0199 - val_accuracy: 1.0000
```

```
Epoch 129/200
10/10 [==============================] - 0s 9ms/step - loss: 0.0899 - accuracy: 0.95
83 - val_loss: 0.0147 - val_accuracy: 1.0000
Epoch 130/200
10/10 [==============================] - 0s 8ms/step - loss: 0.1139 - accuracy: 0.94
79 - val_loss: 0.0173 - val_accuracy: 1.0000
Epoch 131/200
10/10 [==============================] - 0s 7ms/step - loss: 0.0841 - accuracy: 0.96
88 - val_loss: 0.0170 - val_accuracy: 1.0000
Epoch 132/200
10/10 [==============================] - 0s 8ms/step - loss: 0.1042 - accuracy: 0.95
83 - val_loss: 0.0190 - val_accuracy: 1.0000
Epoch 133/200
10/10 [==============================] - 0s 9ms/step - loss: 0.0858 - accuracy: 0.96
88 - val_loss: 0.0166 - val_accuracy: 1.0000
Epoch 134/200
10/10 [==============================] - 0s 9ms/step - loss: 0.0808 - accuracy: 0.95
83 - val_loss: 0.0176 - val_accuracy: 1.0000
Epoch 135/200
10/10 [==============================] - 0s 8ms/step - loss: 0.0832 - accuracy: 0.97
92 - val_loss: 0.0168 - val_accuracy: 1.0000
Epoch 136/200
10/10 [==============================] - 0s 7ms/step - loss: 0.0825 - accuracy: 0.98
96 - val_loss: 0.0159 - val_accuracy: 1.0000
Epoch 137/200
10/10 [==============================] - 0s 8ms/step - loss: 0.0917 - accuracy: 0.97
92 - val_loss: 0.0169 - val_accuracy: 1.0000
Epoch 138/200
10/10 [==============================] - 0s 9ms/step - loss: 0.0934 - accuracy: 0.97
92 - val_loss: 0.0159 - val_accuracy: 1.0000
Epoch 139/200
10/10 [==============================] - 0s 8ms/step - loss: 0.0849 - accuracy: 0.97
92 - val_loss: 0.0175 - val_accuracy: 1.0000
Epoch 140/200
10/10 [==============================] - 0s 7ms/step - loss: 0.0824 - accuracy: 0.96
88 - val_loss: 0.0163 - val_accuracy: 1.0000
Epoch 141/200
10/10 [==============================] - 0s 9ms/step - loss: 0.0813 - accuracy: 0.95
83 - val_loss: 0.0162 - val_accuracy: 1.0000
Epoch 142/200
10/10 [==============================] - 0s 9ms/step - loss: 0.0963 - accuracy: 0.95
83 - val_loss: 0.0182 - val_accuracy: 1.0000
Epoch 143/200
10/10 [==============================] - 0s 7ms/step - loss: 0.0901 - accuracy: 0.97
92 - val_loss: 0.0156 - val_accuracy: 1.0000
Epoch 144/200
10/10 [==============================] - 0s 7ms/step - loss: 0.0924 - accuracy: 0.96
88 - val_loss: 0.0160 - val_accuracy: 1.0000
Epoch 145/200
10/10 [==============================] - 0s 8ms/step - loss: 0.0958 - accuracy: 0.96
88 - val_loss: 0.0162 - val_accuracy: 1.0000
Epoch 146/200
10/10 [==============================] - 0s 9ms/step - loss: 0.0706 - accuracy: 0.97
92 - val_loss: 0.0141 - val_accuracy: 1.0000
Epoch 147/200
10/10 [==============================] - 0s 8ms/step - loss: 0.0868 - accuracy: 0.97
92 - val_loss: 0.0164 - val_accuracy: 1.0000
Epoch 148/200
10/10 [==============================] - 0s 9ms/step - loss: 0.0699 - accuracy: 0.97
92 - val_loss: 0.0182 - val_accuracy: 1.0000
Epoch 149/200
10/10 [==============================] - 0s 9ms/step - loss: 0.0916 - accuracy: 0.97
92 - val_loss: 0.0158 - val_accuracy: 1.0000
Epoch 150/200
```

```
10/10 [==============================] - 0s 11ms/step - loss: 0.0963 - accuracy: 0.9
792 - val_loss: 0.0157 - val_accuracy: 1.0000
Epoch 151/200
10/10 [==============================] - 0s 13ms/step - loss: 0.0792 - accuracy: 0.9
688 - val_loss: 0.0153 - val_accuracy: 1.0000
Epoch 152/200
10/10 [==============================] - 0s 10ms/step - loss: 0.0872 - accuracy: 0.9
688 - val_loss: 0.0144 - val_accuracy: 1.0000
Epoch 153/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0887 - accuracy: 0.9
479 - val_loss: 0.0148 - val_accuracy: 1.0000
Epoch 154/200
10/10 [==============================] - 0s 11ms/step - loss: 0.1114 - accuracy: 0.9
583 - val_loss: 0.0165 - val_accuracy: 1.0000
Epoch 155/200
10/10 [==============================] - 0s 12ms/step - loss: 0.1196 - accuracy: 0.9
479 - val_loss: 0.0152 - val_accuracy: 1.0000
Epoch 156/200
10/10 [==============================] - 0s 10ms/step - loss: 0.0996 - accuracy: 0.9
792 - val_loss: 0.0215 - val_accuracy: 1.0000
Epoch 157/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0834 - accuracy: 0.9
792 - val_loss: 0.0139 - val_accuracy: 1.0000
Epoch 158/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0825 - accuracy: 0.9
792 - val_loss: 0.0146 - val_accuracy: 1.0000
Epoch 159/200
10/10 [==============================] - 0s 12ms/step - loss: 0.0954 - accuracy: 0.9
688 - val_loss: 0.0140 - val_accuracy: 1.0000
Epoch 160/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0761 - accuracy: 0.9
688 - val_loss: 0.0131 - val_accuracy: 1.0000
Epoch 161/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0765 - accuracy: 0.9
688 - val_loss: 0.0128 - val_accuracy: 1.0000
Epoch 162/200
10/10 [==============================] - 0s 12ms/step - loss: 0.0849 - accuracy: 0.9
792 - val_loss: 0.0152 - val_accuracy: 1.0000
Epoch 163/200
10/10 [==============================] - 0s 12ms/step - loss: 0.1007 - accuracy: 0.9
688 - val_loss: 0.0129 - val_accuracy: 1.0000
Epoch 164/200
10/10 [==============================] - 0s 14ms/step - loss: 0.0922 - accuracy: 0.9
583 - val_loss: 0.0131 - val_accuracy: 1.0000
Epoch 165/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0976 - accuracy: 0.9
583 - val_loss: 0.0163 - val_accuracy: 1.0000
Epoch 166/200
10/10 [==============================] - 0s 9ms/step - loss: 0.0821 - accuracy: 0.97
92 - val_loss: 0.0137 - val_accuracy: 1.0000
Epoch 167/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0829 - accuracy: 0.9
792 - val_loss: 0.0130 - val_accuracy: 1.0000
Epoch 168/200
10/10 [==============================] - 0s 10ms/step - loss: 0.0812 - accuracy: 0.9
792 - val_loss: 0.0126 - val_accuracy: 1.0000
Epoch 169/200
10/10 [==============================] - 0s 12ms/step - loss: 0.0921 - accuracy: 0.9
688 - val_loss: 0.0136 - val_accuracy: 1.0000
Epoch 170/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0842 - accuracy: 0.9
688 - val_loss: 0.0118 - val_accuracy: 1.0000
Epoch 171/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0878 - accuracy: 0.9
```

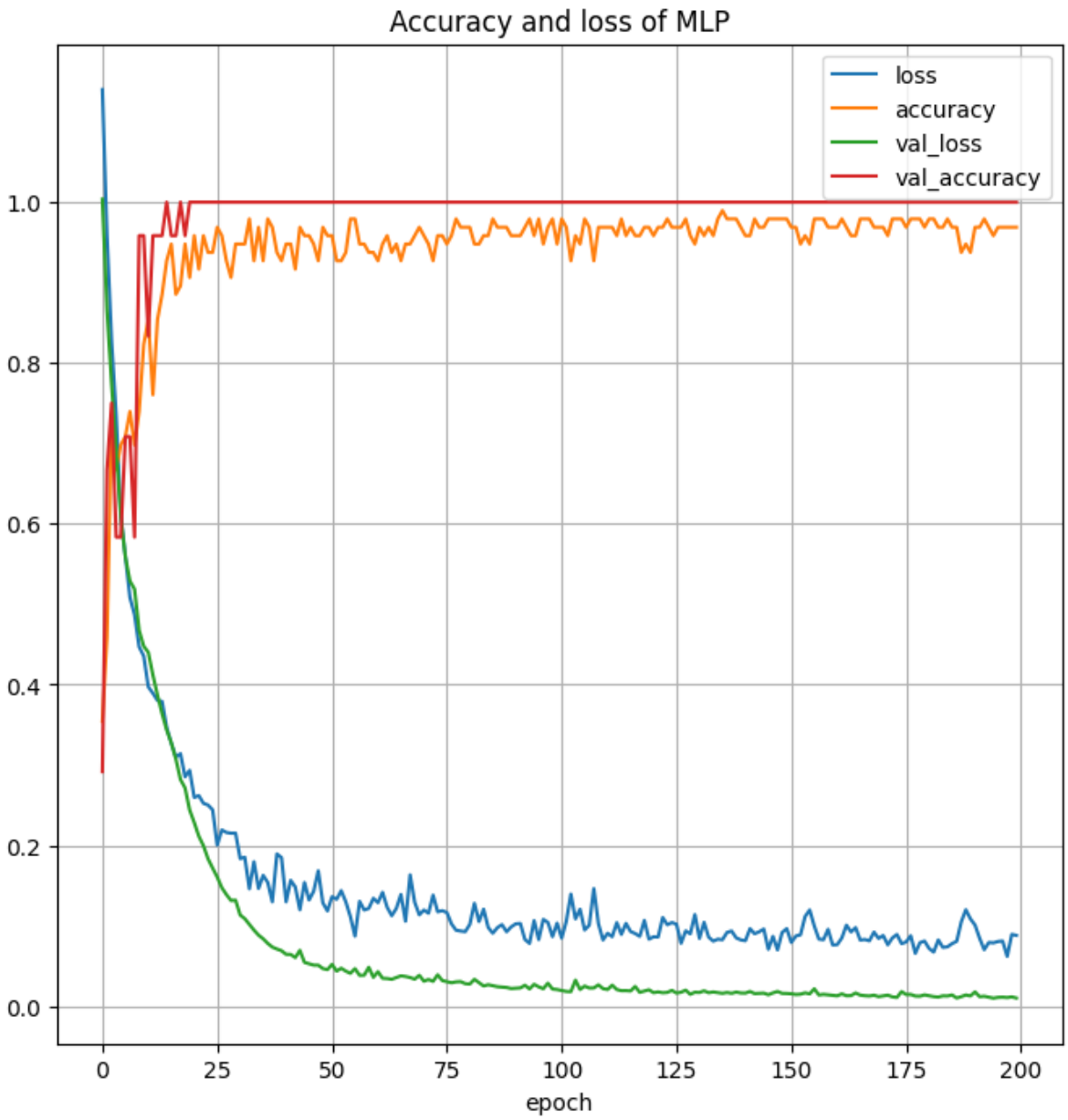
```
688 - val_loss: 0.0126 - val_accuracy: 1.0000
Epoch 172/200
10/10 [==============================] - 0s 12ms/step - loss: 0.0763 - accuracy: 0.9
583 - val_loss: 0.0136 - val_accuracy: 1.0000
Epoch 173/200
10/10 [==============================] - 0s 12ms/step - loss: 0.0848 - accuracy: 0.9
792 - val_loss: 0.0115 - val_accuracy: 1.0000
Epoch 174/200
10/10 [==============================] - 0s 14ms/step - loss: 0.0891 - accuracy: 0.9
792 - val_loss: 0.0108 - val_accuracy: 1.0000
Epoch 175/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0777 - accuracy: 0.9
792 - val_loss: 0.0181 - val_accuracy: 1.0000
Epoch 176/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0798 - accuracy: 0.9
688 - val_loss: 0.0145 - val_accuracy: 1.0000
Epoch 177/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0875 - accuracy: 0.9
792 - val_loss: 0.0145 - val_accuracy: 1.0000
Epoch 178/200
10/10 [==============================] - 0s 12ms/step - loss: 0.0656 - accuracy: 0.9
792 - val_loss: 0.0125 - val_accuracy: 1.0000
Epoch 179/200
10/10 [==============================] - 0s 12ms/step - loss: 0.0789 - accuracy: 0.9
792 - val_loss: 0.0123 - val_accuracy: 1.0000
Epoch 180/200
10/10 [==============================] - 0s 12ms/step - loss: 0.0809 - accuracy: 0.9
688 - val_loss: 0.0140 - val_accuracy: 1.0000
Epoch 181/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0716 - accuracy: 0.9
792 - val_loss: 0.0128 - val_accuracy: 1.0000
Epoch 182/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0672 - accuracy: 0.9
792 - val_loss: 0.0118 - val_accuracy: 1.0000
Epoch 183/200
10/10 [==============================] - 0s 10ms/step - loss: 0.0823 - accuracy: 0.9
688 - val_loss: 0.0113 - val_accuracy: 1.0000
Epoch 184/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0730 - accuracy: 0.9
688 - val_loss: 0.0127 - val_accuracy: 1.0000
Epoch 185/200
10/10 [==============================] - 0s 11ms/step - loss: 0.0737 - accuracy: 0.9
792 - val_loss: 0.0126 - val_accuracy: 1.0000
Epoch 186/200
10/10 [==============================] - 0s 13ms/step - loss: 0.0778 - accuracy: 0.9
688 - val_loss: 0.0139 - val_accuracy: 1.0000
Epoch 187/200
10/10 [==============================] - 0s 13ms/step - loss: 0.0806 - accuracy: 0.9
688 - val_loss: 0.0101 - val_accuracy: 1.0000
Epoch 188/200
10/10 [==============================] - 0s 9ms/step - loss: 0.1048 - accuracy: 0.93
75 - val_loss: 0.0118 - val_accuracy: 1.0000
Epoch 189/200
10/10 [==============================] - 0s 9ms/step - loss: 0.1199 - accuracy: 0.94
79 - val_loss: 0.0139 - val_accuracy: 1.0000
Epoch 190/200
10/10 [==============================] - 0s 7ms/step - loss: 0.1085 - accuracy: 0.93
75 - val_loss: 0.0127 - val_accuracy: 1.0000
Epoch 191/200
10/10 [==============================] - 0s 9ms/step - loss: 0.1008 - accuracy: 0.96
88 - val_loss: 0.0177 - val_accuracy: 1.0000
Epoch 192/200
10/10 [==============================] - 0s 7ms/step - loss: 0.0844 - accuracy: 0.96
88 - val_loss: 0.0117 - val_accuracy: 1.0000
```

```
Epoch 193/200
10/10 [==============================] - 0s 7ms/step - loss: 0.0702 - accuracy: 0.97
92 - val_loss: 0.0123 - val_accuracy: 1.0000
Epoch 194/200
10/10 [==============================] - 0s 8ms/step - loss: 0.0791 - accuracy: 0.96
88 - val_loss: 0.0111 - val_accuracy: 1.0000
Epoch 195/200
10/10 [==============================] - 0s 9ms/step - loss: 0.0787 - accuracy: 0.95
83 - val_loss: 0.0099 - val_accuracy: 1.0000
Epoch 196/200
10/10 [==============================] - 0s 7ms/step - loss: 0.0804 - accuracy: 0.96
88 - val_loss: 0.0109 - val_accuracy: 1.0000
Epoch 197/200
10/10 [==============================] - 0s 8ms/step - loss: 0.0810 - accuracy: 0.96
88 - val_loss: 0.0111 - val_accuracy: 1.0000
Epoch 198/200
10/10 [==============================] - 0s 10ms/step - loss: 0.0618 - accuracy: 0.9
688 - val_loss: 0.0106 - val_accuracy: 1.0000
Epoch 199/200
10/10 [==============================] - 0s 9ms/step - loss: 0.0892 - accuracy: 0.96
88 - val_loss: 0.0115 - val_accuracy: 1.0000
Epoch 200/200
10/10 [==============================] - 0s 8ms/step - loss: 0.0880 - accuracy: 0.96
88 - val_loss: 0.0100 - val_accuracy: 1.0000
```

Plot the result

In [20]:
```
pd.DataFrame(history.history).plot(figsize=(8, 8)) #plot keys all at once
plt.grid(True) ;plt.title("Accuracy and loss of MLP");plt.xlabel("epoch")
```

Out[20]:
```
Text(0.5, 0, 'epoch')
```



Testing the model

In [21]:
```
'''
    학습이 종료된 모델을 평가하는 단계이다.
    데이터를 load 하는 단계부터 미리 분리하여 학습에 전혀 사용되지 않은 test 데이터셋
'''

score = model.evaluate(X_test, y_test)

print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

```
1/1 [==============================] - 0s 53ms/step - loss: 0.0193 - accuracy: 1.000
0
Test loss: 0.019313892349600792
Test accuracy: 1.0
```

A CNN example with fashion MNIST

Setup

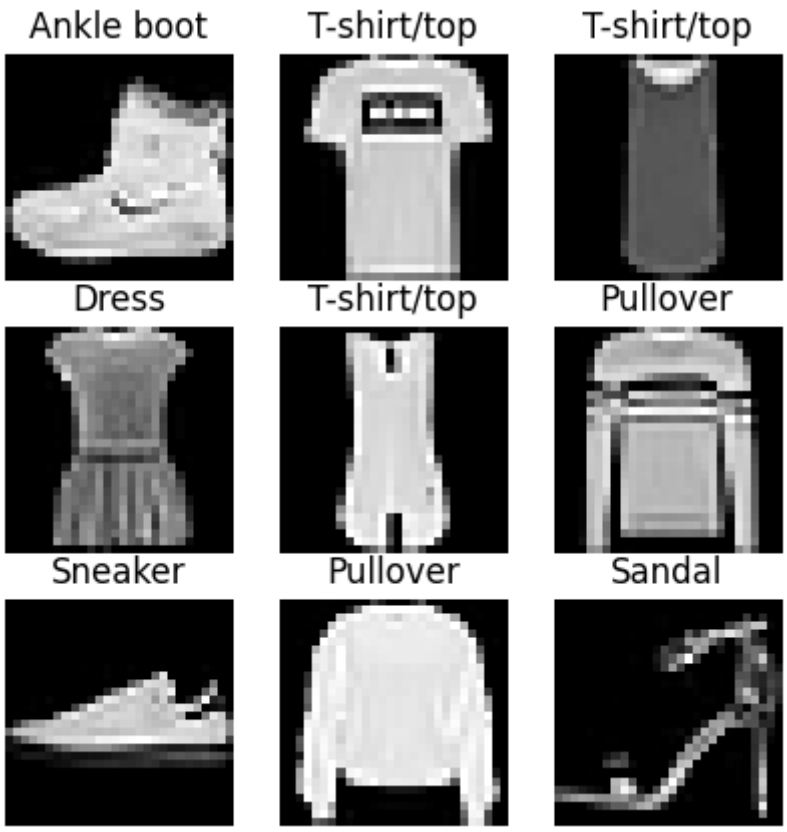
In [ ]:
```
import numpy as np  # numpy는 수치해석용 파이썬 라이브러리로 np라는 축약어로 import
import pandas as pd # 데이터 처리를 위한 라이브러리이다. pd라는 축약어로 import한다
import matplotlib.pyplot as plt #matplotlib은 자료를 차트나 그래프로 시각화하는 라

from tensorflow import keras    # Tensorflow로 부터 keras를 import한다.
from tensorflow.keras import layers # tensorflow의 서브패키지인 keras로 부터 모델에
```

Prepare the data

In [ ]:
```
'''
    데이터 셋을 준비하는 과정이다.
    데이터를 불러오고 가공하며 train 과 test셋으로 분리한다.
    x 는 학습에 사용되는 입력 데이터를 의미하며 y는 입력된 x를 통해 예상하여야 하는
'''
# 최종 학습에서 예측에 활용되는 클래스의 갯수이다. 즉, 현재 학습에서는 최종적으로 입
num_classes = 10
# 학습에 사용되는 데이터의 차원을 미리 선언해주는 단계이다. 이번 학습에서는 fasion m
input_shape = (28, 28, 1)

# keras가 제공해주는 datasets 중 fasion_mist를 로드하는 과정이다.
(x_train, y_train_class), (x_test, y_test_class) = keras.datasets.fashion_mnist.loa

# 0 부터 255가지 표현되어 있는 x를 0 부터 1까지로 스케일링 해주는 과정이다. astype("
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
# fashon mnist데이터는 가로 X 세로, 28 X 28 데이터 이기 때문에 이를 (28, 28, 1)로 확
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
# 최종적으로 전처리가 완료된 데이터의 차원을 직접 눈으로 확인하기 위해 print를 사용하
print("x_train shape:", x_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")


# 0 부터 9까지 단순한 숫자로 이루어져 있는 y를 카테고리 형식의 바이너리 matrix로 변환
# 예를 들어 y가 0 부터 9 까지 클래스 중 3 이라면 카테고리 형식의 바이너리 matrix로 변
y_train = keras.utils.to_categorical(y_train_class, num_classes)
y_test = keras.utils.to_categorical(y_test_class, num_classes)
```

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

In [ ]:
```
labels = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt',
# plot이 그려지는 window의 사이즈를 x축 5, y축 5로 지정한다. (단위는 inch이다)
plt.figure(figsize=(5, 5))
for i in range(9):
    ax = plt.subplot(3, 3, i + 1) #3x3 subplot을 만들고 이미지의 위치를 i가 증가함에
    plt.imshow(x_train[i], cmap='gray') # grayscale의 이미지로 시각화해준다.
    plt.title(labels[y_train_class[i]]) #각 plot의 제목을 지정해준다
    plt.axis("off") # 축에 관한 함수로 "off" 로 지정하면 x축, y축을 표시하지 않는다.
```



Build the model

In [ ]:
```
'''
    모델을 선언하는 과정이다.
    keras.Sequential은 선언된 각 레이어를 순서대로 실행시켜 주는 함수이며 아래와 같이
'''

model = keras.Sequential(
    [
        # 가장 먼저 input 차원을 정의한다. input_shape는 위에 데이터 처리 과정에서 (
        keras.Input(shape=input_shape),

        # 2D Convolution layer를 선언한다. 64는 필터의 갯수로 현재 (28, 28, 1)의 데이
        # 커널사이즈는 Convolution 계산을 위한 커널의 사이즈를 정의한 것이며, activa
        # 즉, (28, 28, 1)의 데이터가 (28, 28, 64)의 크기로 변환되는 Convolution laye
        layers.Conv2D(64, kernel_size=(7, 7), activation="relu", padding="same"),

        # Max pooling layer를 선언한다. pool_size는 pooling에 사용되는 커널의 크기이
        layers.MaxPooling2D(pool_size=(2, 2)),

        # 2D Convolution layer를 선언한다. (14, 14, 64)의 입력 데이터는 이후 (14, 14
        layers.Conv2D(128, kernel_size=(3, 3), activation="relu", padding="same"),

        # 2D Convolution layer를 선언한다. (14, 14, 128)의 입력 데이터는 이후 (14, 1
        layers.Conv2D(128, kernel_size=(3, 3), activation="relu", padding="same"),

        # Max pooling layer를 선언한다. (14, 14, 128)의 데이터가 (7, 7, 128)로 변환된
        layers.MaxPooling2D(pool_size=(2, 2)),

        # 2D Convolution layer를 선언한다. (7, 7, 128)의 입력 데이터는 이후 (7, 7, 2
        layers.Conv2D(256, kernel_size=(3, 3), activation="relu", padding="same"),

        # 2D Convolution layer를 선언한다. (7, 7, 256)의 입력 데이터는 이후 (7, 7, 2
        layers.Conv2D(256, kernel_size=(3, 3), activation="relu", padding="same"),
```

```
        # Max pooling layer를 선언한다. (7, 7, 256)의 데이터가 (3, 3, 256)로 변환된다
        layers.MaxPooling2D(pool_size=(2, 2)),

        # Fully connected layer에 적용하기 전 (3, 3, 256)의 3차원 데이터를 1차원 데이
        # 즉, 데이터 크기는 3X3X256 인 2304가 된다.
        layers.Flatten(),

        # Fully connected layer를 선언해 준다. 2304의 input 데이터를 받아 총 128의 다
        layers.Dense(128, activation="relu"),

        # 특정 비율로 랜덤하게 노드를 비활성화 하는 Dropout레이어 이다. 현재 128개의
        # Dropout을 하는 이유로는 특정 설명변수 feature에만 과도하게 집중하여 발생하
        layers.Dropout(0.5),

        # Fully connected layer를 선언해 준다. 128의 input 데이터를 받아 총 64의 데이
        layers.Dense(64, activation="relu"),

        # 특정 비율로 랜덤하게 노드를 비활성화 하는 Dropout레이어 이다. 현재 64개의
        layers.Dropout(0.5),

        # Fully connected layer를 선언해 준다. 최종 layer이기 때문에 출력하는 데이터
        # softmax를 사용함으로서 출력되는 값을 정규화 한다.
        layers.Dense(num_classes, activation="softmax"),
    ]
)

# 자신이 만든 모델과 데이터의 차원 변화를 눈으로 확인할 수있게 도와주는 함수이다.
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 64)        3200

 max_pooling2d (MaxPooling2D  (None, 14, 14, 64)       0
 )

 conv2d_1 (Conv2D)           (None, 14, 14, 128)       73856

 conv2d_2 (Conv2D)           (None, 14, 14, 128)       147584

 max_pooling2d_1 (MaxPooling  (None, 7, 7, 128)        0
 2D)

 conv2d_3 (Conv2D)           (None, 7, 7, 256)         295168

 conv2d_4 (Conv2D)           (None, 7, 7, 256)         590080

 max_pooling2d_2 (MaxPooling  (None, 3, 3, 256)        0
 2D)

 flatten (Flatten)           (None, 2304)              0

 dense (Dense)               (None, 128)               295040

 dropout (Dropout)           (None, 128)               0

 dense_1 (Dense)             (None, 64)                8256

 dropout_1 (Dropout)         (None, 64)                0

 dense_2 (Dense)             (None, 10)                650

=================================================================
Total params: 1,413,834
Trainable params: 1,413,834
Non-trainable params: 0
_________________________________________________________________
```

Train the model

In [ ]:

```
'''
    구성한 모델을 이용하여 실제로 학습을 진행하는 과정이다.
'''

# 한 배치에 들어가는 데이터의 갯수를 정의한다. 너무 많은 데이터를 한 배치에 넣으면 메
batch_size = 4096
# train 데이터를 전부 활용하는 것을 1 epoch라고 한다. 25라고 선언해 줌으로서 train 데
epochs = 25

# 모델을 컴파일 하는 단계이다. 모델을 평가하는 과정에서 사용되는 loss와 metrics를 정
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"

# 모델이 실제로 학습을 시작하게 하는 함수이다.
# 학습에 사용되는 x, y 데이터를 입력하고, batch_size와 epochs를 정의 할 수 있다.
# validation_split을 0.1로 정의하였기 때문에 train 데이터중 10퍼센트를 로스를 계산하
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validati
```

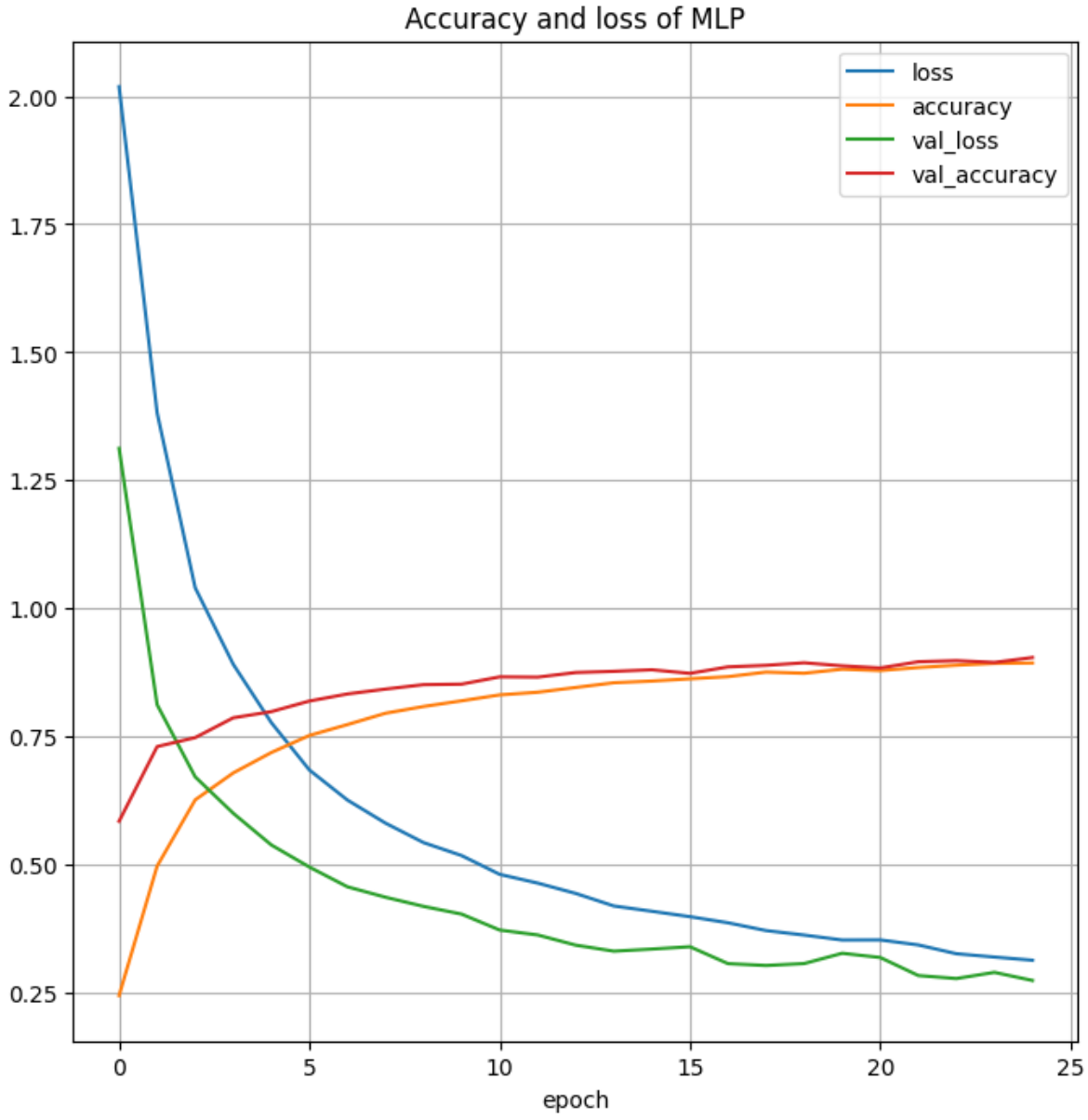
```
Epoch 1/25
14/14 [==============================] - 34s 754ms/step - loss: 2.0183 - accuracy:
0.2451 - val_loss: 1.3127 - val_accuracy: 0.5852
Epoch 2/25
14/14 [==============================] - 5s 381ms/step - loss: 1.3818 - accuracy: 0.
4977 - val_loss: 0.8125 - val_accuracy: 0.7308
Epoch 3/25
14/14 [==============================] - 5s 387ms/step - loss: 1.0407 - accuracy: 0.
6269 - val_loss: 0.6717 - val_accuracy: 0.7483
Epoch 4/25
14/14 [==============================] - 5s 388ms/step - loss: 0.8913 - accuracy: 0.
6794 - val_loss: 0.6007 - val_accuracy: 0.7870
Epoch 5/25
14/14 [==============================] - 5s 390ms/step - loss: 0.7776 - accuracy: 0.
7192 - val_loss: 0.5389 - val_accuracy: 0.7990
Epoch 6/25
14/14 [==============================] - 6s 396ms/step - loss: 0.6852 - accuracy: 0.
7525 - val_loss: 0.4958 - val_accuracy: 0.8197
Epoch 7/25
14/14 [==============================] - 6s 406ms/step - loss: 0.6265 - accuracy: 0.
7737 - val_loss: 0.4573 - val_accuracy: 0.8335
Epoch 8/25
14/14 [==============================] - 6s 410ms/step - loss: 0.5815 - accuracy: 0.
7959 - val_loss: 0.4369 - val_accuracy: 0.8430
Epoch 9/25
14/14 [==============================] - 6s 399ms/step - loss: 0.5435 - accuracy: 0.
8090 - val_loss: 0.4190 - val_accuracy: 0.8517
Epoch 10/25
14/14 [==============================] - 6s 416ms/step - loss: 0.5182 - accuracy: 0.
8201 - val_loss: 0.4040 - val_accuracy: 0.8527
Epoch 11/25
14/14 [==============================] - 6s 405ms/step - loss: 0.4815 - accuracy: 0.
8317 - val_loss: 0.3729 - val_accuracy: 0.8668
Epoch 12/25
14/14 [==============================] - 6s 422ms/step - loss: 0.4645 - accuracy: 0.
8369 - val_loss: 0.3633 - val_accuracy: 0.8663
Epoch 13/25
14/14 [==============================] - 6s 414ms/step - loss: 0.4443 - accuracy: 0.
8462 - val_loss: 0.3434 - val_accuracy: 0.8753
Epoch 14/25
14/14 [==============================] - 6s 426ms/step - loss: 0.4198 - accuracy: 0.
8554 - val_loss: 0.3316 - val_accuracy: 0.8777
Epoch 15/25
14/14 [==============================] - 6s 421ms/step - loss: 0.4095 - accuracy: 0.
8587 - val_loss: 0.3358 - val_accuracy: 0.8807
Epoch 16/25
14/14 [==============================] - 6s 434ms/step - loss: 0.3990 - accuracy: 0.
8631 - val_loss: 0.3403 - val_accuracy: 0.8737
Epoch 17/25
14/14 [==============================] - 6s 424ms/step - loss: 0.3870 - accuracy: 0.
8673 - val_loss: 0.3073 - val_accuracy: 0.8863
Epoch 18/25
14/14 [==============================] - 6s 435ms/step - loss: 0.3719 - accuracy: 0.
8762 - val_loss: 0.3038 - val_accuracy: 0.8893
Epoch 19/25
14/14 [==============================] - 6s 429ms/step - loss: 0.3631 - accuracy: 0.
8739 - val_loss: 0.3075 - val_accuracy: 0.8943
Epoch 20/25
14/14 [==============================] - 6s 428ms/step - loss: 0.3534 - accuracy: 0.
8817 - val_loss: 0.3275 - val_accuracy: 0.8883
Epoch 21/25
14/14 [==============================] - 6s 415ms/step - loss: 0.3536 - accuracy: 0.
8791 - val_loss: 0.3193 - val_accuracy: 0.8842
Epoch 22/25
```

```
14/14 [==============================] - 6s 417ms/step - loss: 0.3438 - accuracy: 0.
8851 - val_loss: 0.2841 - val_accuracy: 0.8963
Epoch 23/25
14/14 [==============================] - 6s 425ms/step - loss: 0.3267 - accuracy: 0.
8896 - val_loss: 0.2783 - val_accuracy: 0.8985
Epoch 24/25
14/14 [==============================] - 6s 416ms/step - loss: 0.3202 - accuracy: 0.
8932 - val_loss: 0.2903 - val_accuracy: 0.8950
Epoch 25/25
14/14 [==============================] - 6s 425ms/step - loss: 0.3140 - accuracy: 0.
8940 - val_loss: 0.2745 - val_accuracy: 0.9047
```

Evaluate the trained model

In [ ]:
```
import matplotlib.pyplot as plt #matplotlib은 자료를 차트나 그래프로 시각화하는 라C

pd.DataFrame(history.history).plot(figsize=(8, 8)) #plot keys all at once
plt.grid(True) ;plt.title("Accuracy and loss of MLP");plt.xlabel("epoch")
```

Out[ ]:
```
Text(0.5, 0, 'epoch')
```



In [ ]:
```
'''
    학습이 종료된 모델을 평가하는 단계이다.
    데이터를 load 하는 단계부터 미리 분리하여 학습에 전혀 사용되지 않은 test 데이터셋
```

```
'''


# test 데이터 셋을 활용하여 모델을 평가, 그 후 loss와 accuracy를 출력한다.
# verbose를 0으로 설정하였기 때문에 따로 진행바가 표시 되지 않는다.
score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

```
Test loss: 0.2858674228191376
Test accuracy: 0.8991000056266785
```

A CNN example with AlexNet

Setup

In [ ]:
```
import numpy as np #numpy는 수치해석용 파이썬 라이브러리로 np라는 축약어로 import한
from tensorflow import keras #Tensorflow 2.0 부터 Keras가 내부 패키지로 포함 되었다
from tensorflow.keras import layers #tensorflow의 서브패키지인 keras로 부터 모델에
```

Prepare the data

In [ ]:
```
'''
    데이터 셋을 준비하는 과정이다.
    데이터를 불러오고 가공하며 train 과 test셋으로 분리한다.
    x 는 학습에 사용되는 입력 데이터를 의미하며 y는 입력된 x를 통해 예상하여야 하는
'''
# 최종 학습에서 예측에 활용되는 클래스의 갯수이다. 즉, 현재 학습에서는 최종적으로 입
num_classes = 10
# 학습에 사용되는 데이터의 차원을 미리 선언해주는 단계이다. 이번 학습에서는 fasion m
input_shape = (28, 28, 1)

# keras가 제공해주는 datasets 중 mist를 로드하는 과정이다.
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# 0 부터 255가지 표현되어 있는 x를 0 부터 1까지로 스케일링 해주는 과정이다. astype("
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
# mnist데이터는 가로 X 세로, 28 X 28 데이터 이기 때문에 이를 (28, 28, 1)로 확장해 주
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
# 최종적으로 전처리가 완료된 데이터의 차원을 직접 눈으로 확인하기 위해 print를 사용하
print("x_train shape:", x_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")

# 0 부터 9까지 단순한 숫자로 이루어져 있는 y를 카테고리 형식의 바이너리 matrix로 변환
# 예를 들어 y가 0 부터 9 까지 클래스 중 3 이라면 카테고리 형식의 바이너리 matrix로 변
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mn
ist.npz
11490434/11490434 [==============================] - 0s 0us/step
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

Build the model

In [ ]:
```
'''
    모델을 선언하는 과정이다. 이번에는 구현하는 모델은 AlexNet이라는 모델이며
    AlexNet은 2012년 ImageNet ILSVRC challenge에서 가장 뛰어난 성능을 보여주었던 모델
    keras.Sequential은 선언된 각 레이어를 순서대로 실행시켜 주는 함수이며 아래와 같이
'''


model = keras.Sequential(
```

```
    [
        # 가장 먼저 input 차원을 정의한다. input_shape는 위에 데이터 처리 과정에서 (
        keras.Input(shape=input_shape),
        # 2D Convolution layer를 선언한다. 96는 필터의 갯수로 현재 (28, 28, 1)의 데이
        # 커널사이즈는 Convolution 계산을 위한 커널의 사이즈를 정의한 것이며, activa
        # 즉, (28, 28, 1)의 데이터가 (5, 5, 96)의 크기로 변환되는 Convolution layer이
        layers.Conv2D(96, kernel_size=(11, 11), strides=(4, 4), activation="relu", p
        # Max pooling layer를 선언한다. pool_size는 pooling에 사용되는 커널의 크기이
        layers.MaxPooling2D(pool_size=(3, 3), strides=(2, 2), padding="valid"),
        # 2D Convolution layer를 선언한다. (2, 2, 96)의 입력 데이터는 이후 (2, 2, 25
        layers.Conv2D(256, kernel_size=(5, 5), strides=(1, 1), activation="relu", pa
        # Max pooling layer를 선언한다. pool_size는 pooling에 사용되는 커널의 크기이
        layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding="valid"),
        # 2D Convolution layer를 선언한다. (1, 1, 256)의 입력 데이터는 이후 (1, 1, 3
        layers.Conv2D(384, kernel_size=(3, 3), activation="relu", padding="same"),
        # 2D Convolution layer를 선언한다. (1, 1, 256)의 입력 데이터는 이후 (1, 1, 3
        layers.Conv2D(384, kernel_size=(3, 3), activation="relu", padding="same"),
        # 2D Convolution layer를 선언한다. (1, 1, 256)의 입력 데이터는 이후 (1, 1, 2
        layers.Conv2D(256, kernel_size=(3, 3), activation="relu", padding="same"),
        # Fully connected layer에 적용하기 전 (1, 1, 256)의 3차원 데이터를 1차원 데이
        # 즉, 데이터 크기는 1X1X256 인 256가 된다.
        layers.Flatten(),
        # Fully connected layer를 선언해 준다. 256의 input 데이터를 받아 총 4096의 다
        layers.Dense(4096, activation="relu"),
        # 특정 비율로 랜덤하게 노드를 비활성화 하는 Dropout레이어 이다. 현재 4096개의
        # Dropout을 하는 이유로는 특정 설명변수 feature에만 과도하게 집중하여 발생하
        layers.Dropout(0.5),
        # Fully connected layer를 선언해 준다. 4096의 input 데이터를 받아 총 4096의
        layers.Dense(4096, activation="relu"),
        # Fully connected layer를 선언해 준다. 최종 layer이기 때문에 출력하는 데이터
        # softmax를 사용함으로서 출력되는 값을 정규화 한다.
        layers.Dense(num_classes, activation="softmax"),
    ]
)

# 자신이 만든 모델과 데이터의 차원 변화를 눈으로 확인할 수있게 도와주는 함수이다.
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 5, 5, 96)          11712

 max_pooling2d (MaxPooling2D  (None, 2, 2, 96)         0
 )

 conv2d_1 (Conv2D)           (None, 2, 2, 256)         614656

 max_pooling2d_1 (MaxPooling  (None, 1, 1, 256)        0
 2D)

 conv2d_2 (Conv2D)           (None, 1, 1, 384)         885120

 conv2d_3 (Conv2D)           (None, 1, 1, 384)         1327488

 conv2d_4 (Conv2D)           (None, 1, 1, 256)         884992

 flatten (Flatten)           (None, 256)               0

 dense (Dense)               (None, 4096)              1052672

 dropout (Dropout)           (None, 4096)              0

 dense_1 (Dense)             (None, 4096)              16781312

 dense_2 (Dense)             (None, 10)                40970

=================================================================
Total params: 21,598,922
Trainable params: 21,598,922
Non-trainable params: 0
_________________________________________________________________
```

Train the model

In [ ]:

```
'''
    구성한 모델을 이용하여 실제로 학습을 진행하는 과정이다.
'''

# 한 배치에 들어가는 데이터의 갯수를 정의한다. 너무 많은 데이터를 한 배치에 넣으면 메
batch_size = 4096
# train 데이터를 전부 활용하는 것을 1 epoch라고 한다. 25라고 선언해 줌으로서 train 데
epochs = 25

# 모델을 컴파일 하는 단계이다. 모델을 평가하는 과정에서 사용되는 loss와 metrics를 정
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"

# 모델이 실제로 학습을 시작하게 하는 함수이다.
# 학습에 사용되는 x, y 데이터를 입력하고, batch_size와 epochs를 정의 할 수 있다.
# validation_split을 0.1로 정의하였기 때문에 train 데이터중 10퍼센트를 로스를 계산하
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validati
```

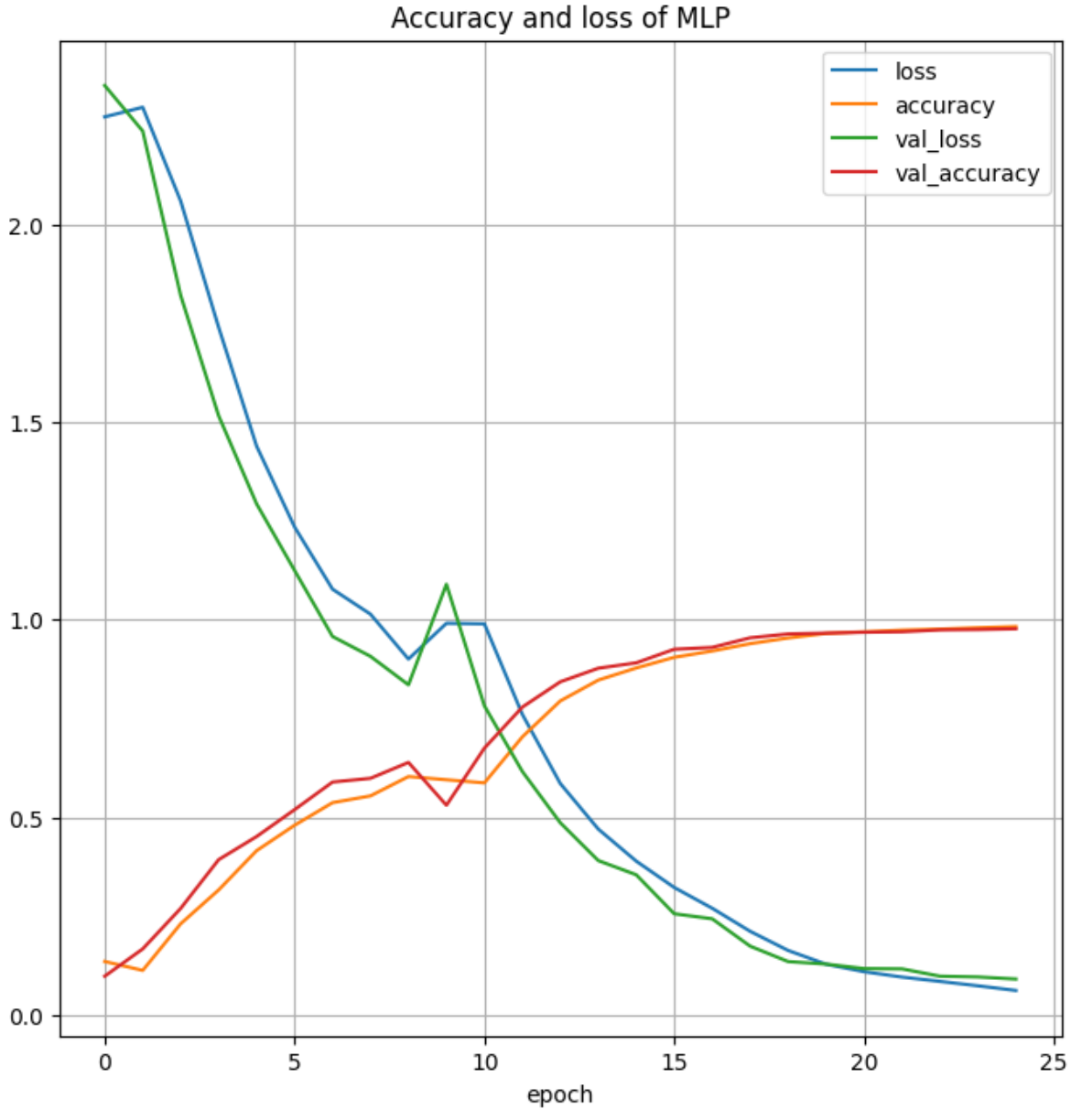
```
Epoch 1/25
14/14 [==============================] - 30s 479ms/step - loss: 2.2721 - accuracy:
0.1351 - val_loss: 2.3513 - val_accuracy: 0.0978
Epoch 2/25
14/14 [==============================] - 3s 190ms/step - loss: 2.2963 - accuracy: 0.
1125 - val_loss: 2.2362 - val_accuracy: 0.1668
Epoch 3/25
14/14 [==============================] - 3s 194ms/step - loss: 2.0591 - accuracy: 0.
2305 - val_loss: 1.8203 - val_accuracy: 0.2693
Epoch 4/25
14/14 [==============================] - 3s 203ms/step - loss: 1.7396 - accuracy: 0.
3160 - val_loss: 1.5158 - val_accuracy: 0.3927
Epoch 5/25
14/14 [==============================] - 3s 194ms/step - loss: 1.4390 - accuracy: 0.
4154 - val_loss: 1.2929 - val_accuracy: 0.4512
Epoch 6/25
14/14 [==============================] - 3s 201ms/step - loss: 1.2346 - accuracy: 0.
4797 - val_loss: 1.1245 - val_accuracy: 0.5192
Epoch 7/25
14/14 [==============================] - 3s 198ms/step - loss: 1.0770 - accuracy: 0.
5368 - val_loss: 0.9574 - val_accuracy: 0.5890
Epoch 8/25
14/14 [==============================] - 3s 204ms/step - loss: 1.0138 - accuracy: 0.
5540 - val_loss: 0.9071 - val_accuracy: 0.5985
Epoch 9/25
14/14 [==============================] - 3s 200ms/step - loss: 0.9002 - accuracy: 0.
6029 - val_loss: 0.8350 - val_accuracy: 0.6387
Epoch 10/25
14/14 [==============================] - 3s 198ms/step - loss: 0.9906 - accuracy: 0.
5949 - val_loss: 1.0895 - val_accuracy: 0.5303
Epoch 11/25
14/14 [==============================] - 3s 199ms/step - loss: 0.9893 - accuracy: 0.
5869 - val_loss: 0.7819 - val_accuracy: 0.6748
Epoch 12/25
14/14 [==============================] - 3s 199ms/step - loss: 0.7602 - accuracy: 0.
7033 - val_loss: 0.6157 - val_accuracy: 0.7787
Epoch 13/25
14/14 [==============================] - 3s 207ms/step - loss: 0.5852 - accuracy: 0.
7943 - val_loss: 0.4860 - val_accuracy: 0.8427
Epoch 14/25
14/14 [==============================] - 3s 198ms/step - loss: 0.4700 - accuracy: 0.
8470 - val_loss: 0.3903 - val_accuracy: 0.8770
Epoch 15/25
14/14 [==============================] - 3s 198ms/step - loss: 0.3889 - accuracy: 0.
8776 - val_loss: 0.3544 - val_accuracy: 0.8907
Epoch 16/25
14/14 [==============================] - 3s 204ms/step - loss: 0.3223 - accuracy: 0.
9046 - val_loss: 0.2558 - val_accuracy: 0.9250
Epoch 17/25
14/14 [==============================] - 3s 203ms/step - loss: 0.2698 - accuracy: 0.
9205 - val_loss: 0.2430 - val_accuracy: 0.9297
Epoch 18/25
14/14 [==============================] - 3s 198ms/step - loss: 0.2114 - accuracy: 0.
9391 - val_loss: 0.1738 - val_accuracy: 0.9540
Epoch 19/25
14/14 [==============================] - 3s 199ms/step - loss: 0.1629 - accuracy: 0.
9534 - val_loss: 0.1348 - val_accuracy: 0.9637
Epoch 20/25
14/14 [==============================] - 3s 197ms/step - loss: 0.1281 - accuracy: 0.
9651 - val_loss: 0.1284 - val_accuracy: 0.9658
Epoch 21/25
14/14 [==============================] - 3s 205ms/step - loss: 0.1092 - accuracy: 0.
9692 - val_loss: 0.1170 - val_accuracy: 0.9680
Epoch 22/25
```

```
14/14 [==============================] - 3s 199ms/step - loss: 0.0956 - accuracy: 0.
9735 - val_loss: 0.1163 - val_accuracy: 0.9692
Epoch 23/25
14/14 [==============================] - 3s 198ms/step - loss: 0.0846 - accuracy: 0.
9759 - val_loss: 0.0978 - val_accuracy: 0.9740
Epoch 24/25
14/14 [==============================] - 3s 207ms/step - loss: 0.0734 - accuracy: 0.
9795 - val_loss: 0.0959 - val_accuracy: 0.9748
Epoch 25/25
14/14 [==============================] - 3s 198ms/step - loss: 0.0617 - accuracy: 0.
9823 - val_loss: 0.0906 - val_accuracy: 0.9768
```

Evaluate the trained model

In [ ]:
```
import matplotlib.pyplot as plt #matplotlib은 자료를 차트나 그래프로 시각화하는 라C
import pandas as pd

pd.DataFrame(history.history).plot(figsize=(8, 8)) #plot keys all at once
plt.grid(True) ;plt.title("Accuracy and loss of MLP");plt.xlabel("epoch")
```

Out[ ]:
```
Text(0.5, 0, 'epoch')
```



In [ ]:
```
'''
    학습이 종료된 모델을 평가하는 단계이다.
```

```
    데이터를 load 하는 단계부터 미리 분리하여 학습에 전혀 사용되지 않은 test 데이터셋
'''

# test 데이터 셋을 활용하여 모델을 평가, 그 후 loss와 accuracy를 출력한다.
# verbose를 0으로 설정하였기 때문에 따로 진행바가 표시 되지 않는다.
score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

```
Test loss: 0.10491205006837845
Test accuracy: 0.9718999862670898
```

A CNN example with ResNet

Introduction

This example shows how to do image classification from scratch, starting from JPEG image files on disk, without leveraging pre-trained weights or a pre-made Keras Application model. We demonstrate the workflow on the Kaggle Cats vs Dogs binary classification dataset.

We use the `image_dataset_from_directory` utility to generate the datasets, and we use Keras image preprocessing layers for image standardization and data augmentation.

Setup

In [ ]:
```
import numpy as np  # numpy는 수치해석용 파이썬 라이브러리로 np라는 축약어로 import
import pandas as pd # 데이터 처리를 위한 라이브러리이다. pd라는 축약어로 import한다
import matplotlib.pyplot as plt #matplotlib은 자료를 차트나 그래프로 시각화하는 라이

import tensorflow as tf #Tensorflow 2.0 부터 Keras가 내부 패키지로 포함 되었다. 기존
from tensorflow import keras #Tensorflow로 부터 keras를 import한다.
from tensorflow.keras import layers #tensorflow의 서브패키지인 keras로 부터 모델에
```

Load the data: the Cats vs Dogs dataset

Raw data download

First, let's download the 786M ZIP archive of the raw data:

In [ ]:
```
# !는 Collab에서 리눅스 쉘 명령어를 수행하기 위한 명령어이다.
# 아래 링크에서 파일을 다운로드 받아오는 명령어 이다.
!curl -O https://download.microsoft.com/download/3/E/1/3E1C3F21-ECDB-4869-8368-6DEBA
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  786M  100  786M    0     0   114M      0  0:00:06  0:00:06 --:--:--  120M
```

In [ ]:
```
# 원본 데이터가 zip으로 압축되어 있기 때문에 압축을 해제 하는 명령어이다.
!unzip -q kagglecatsanddogs_5340.zip
# 현재 경로에 파일 리스트를 출력하는 명령어이다. 압축이 해제 되었기 때문에 압축 해제
!ls
```

```
CDLA-Permissive-2.0.pdf      PetImages        sample_data
kagglecatsanddogs_5340.zip  'readme[1].txt'
```

Now we have a `PetImages` folder which contain two subfolders, `Cat` and `Dog`. Each subfolder contains image files for each category.

In [ ]:
```
# PetImages, 즉 우리가 학습에 사용되어야 하는 데이터인 사진이 들어있는 경로이다.
# 목록을 출력하면 하위 폴더로 Cat, Dog가 출력되며 이는 Cat 사진이 들어있는 폴더, Dog
!ls PetImages
```

```
Cat  Dog
```

Filter out corrupted images

When working with lots of real-world image data, corrupted images are a common occurence. Let's filter out badly-encoded images that do not feature the string "JFIF" in their header.

In [ ]:
```
'''
    다운로드 받은 파일 중 붕괴된 파일을 걸러내는 작업이다.
    실제 이미지를 학습에 활용하는 경우 이와 같이 깨진 파일을 걸러내는 작업이 필요하다
'''
import os   # 코드를 통해 리눅스 쉘 명령어를 사용하기 위한 라이브러리이다.

# 깨진 파일의 갯수를 확인하기 위한 변수이다.
num_skipped = 0

# folder_name 을 순서대로 "Cat" 과 "Dog"로 할당하여 반복문을 실행한다.
for folder_name in ("Cat", "Dog"):
    # 경로를 지정한다. os.path.join은 첫번째 인자와 두번째 인자에 ₩를 넣어 결합하여
    # 즉, folder_name 이 "Cat" 인 경우 folder_path 는 "PetImages₩Cat"이 된다.
    folder_path = os.path.join("PetImages", folder_name)
    # os.listdir은 인자로 넣어준 경로 상의 모든 파일을 리스팅 하는 함수이다. 즉 fnam
    for fname in os.listdir(folder_path):
        # 현재 경로와 파일 이름을 결합하여 완전한 파일 경로를 생성한다.
        fpath = os.path.join(folder_path, fname)
        # try 부분을 먼저 실행하고 명령의 성공 실패 여부와 상관없이 finally를 실행한
        try:
            # 이미지 파일을 열어준다. rb는 바이너리 형식으로 파일을 열어준다는 의미이
            fobj = open(fpath, "rb")
            # 이미지 파일 안에 존재하는 상위 10개의 바이너리 데이터를 jfif 형식으로
            # is_jfif는 이미지가 손상된 경우 [0,0,0,0,0,.....,0]으로 표현되기 때문에
            is_jfif = tf.compat.as_bytes("JFIF") in fobj.peek(10)
        finally:
            # 이미지 파일을 닫아준다.
            fobj.close()

        # is_jfif가 0인 경우, 즉 이미지가 손상된 경우 num_skipped의 갯수를 1 올려주고
        if not is_jfif:
            num_skipped += 1
            # Delete corrupted image
            os.remove(fpath)

# 최종적으로 몇개의 이미지가 손상되었고 지워주었는지 확인하기 위한 print문 이다.
print("Deleted %d images" % num_skipped)
```

```
Deleted 1590 images
```

Generate a `Dataset`

In [ ]:
```
'''
    데이터 셋을 train 데이터와 validation으로 나누어 주는 부분이다.
'''
# 모델을 선언 할 때 input_size를 지정해 주기 위해 우선 image의 크기를 선언한다. 해당
image_size = (180, 180)
# 한 배치에 들어가는 데이터의 수를 지정해 준다.
batch_size = 128

# 경로상의 데이터를 활용하여 train 데이터 셋과 validation 데이터 셋을 분리한다.
# PetImages 경로상의 데이터를 활용하며, 20퍼센트의 데이터를 validation으로 사용한다.
# subset은 반환하는 데이터의 subset을 정의하는 인자이다. train, validatoin, both 중
```

```
# seed는 랜덤하게 validation 데이터를 선정하기 때문에 그에 따른 랜덤 시드를 지정 할
train_ds, val_ds = tf.keras.utils.image_dataset_from_directory(
    "PetImages",
    validation_split=0.2,
    subset="both",
    seed=1337,
    image_size=image_size,
    batch_size=batch_size,
)
```

```
Found 23410 files belonging to 2 classes.
Using 18728 files for training.
Using 4682 files for validation.
```

Visualize the data

Here are the first 9 images in the training dataset. As you can see, label 1 is "dog" and label 0 is "cat".

In [ ]:

```
'''
    다운로드 한 데이터를 시각화 하여 눈으로 확인하는 과정이다.
    학습에 영향을 미치는 부분은 아니나 직접 데이터를 눈으로 확인하고 학습의 목표를 ㅈ
'''
label_names = ['cat', 'dog']

# 총 9개의 이미지를 subplot으로 출력할 예정이며 각 이미지의 사이즈를 (10, 10)으로 선
plt.figure(figsize=(10, 10))
# python의 take함수를 이용해 배치 1개의 images와 그 labels를 추출한다. batch_size를
for images, labels in train_ds.take(1):
    for i in range(9):
        # 사용되는 subplot의 크기와 위치를 정의하는 부분이다.
        ax = plt.subplot(3, 3, i + 1)
        # 이미지를 표시하는 부분이다.
        plt.imshow(images[i].numpy().astype("uint8"))
        # subplot의 title을 해당 이미지의 label로 표시하는 부분이다.
        plt.title(label_names[int(labels[i])])
        # plot의 축을 지움으로서 좀 더 깔금하게 이미지를 확인 할 수 있다.
        plt.axis("off")
```

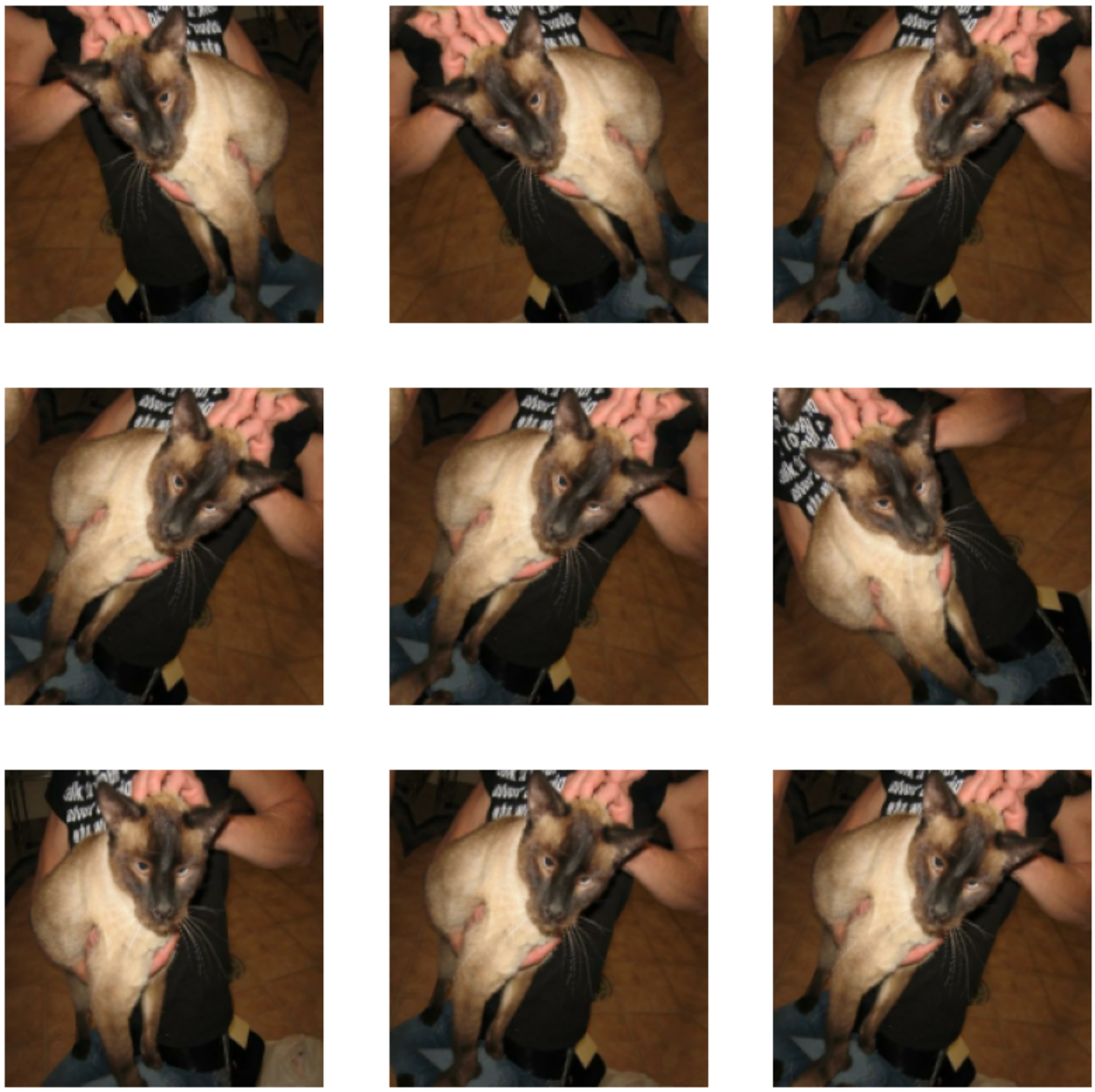
Using image data augmentation

When you don't have a large image dataset, it's a good practice to artificially introduce sample diversity by applying random yet realistic transformations to the training images, such as random horizontal flipping or small random rotations. This helps expose the model to different aspects of the training data while slowing down overfitting.

In [ ]:
```
'''
    데이터가 부족한 경우 랜덤하게 데이터를 조작해 인위적으로 생성된 데이터를 사용하여
    이 단계는 랜덤하게 이미지를 뒤집고 회전하는 모델을 생성하는 단계이다.
'''
data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
    ]
)
```

Let's visualize what the augmented samples look like, by applying `data_augmentation` repeatedly to the first image in the dataset:

In [ ]:
```
# 위에서 선언한 data_augmentation의 결과를 눈으로 확인하는 부분이다.
plt.figure(figsize=(10, 10))
for images, _ in train_ds.take(1):
    for i in range(9):
        augmented_images = data_augmentation(images)
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(augmented_images[0].numpy().astype("uint8"))
        plt.axis("off")
```



Standardizing the data

Our image are already in a standard size (180x180), as they are being yielded as contiguous `float32` batches by our dataset. However, their RGB channel values are in the `[0, 255]` range. This is not ideal for a neural network; in general you should seek to make your input values small. Here, we will standardize values to be in the `[0, 1]` by using a `Rescaling` layer at the start of our model.

Configure the dataset for performance

Let's apply data augmentation to our training dataset, and let's make sure to use buffered prefetching so we can yield data from disk without having I/O becoming blocking:

In [ ]:
```
'''
    data_augmentation을 train 데이터 셋에 적용시키는 부분이다.
'''
# tensorflow 객체의 map함수는 리스트 혹은 튜플의 요소를 지정된 함수로 처리해주는 함수
# 아래 함수는 train_ds에 포함된 모든 이미지를 data_augmentation을 적용시켜 (img, lab
# num_parallel_calls를 지정해 줌으로서 여러개의 스레드를 사용하여 처리속도를 높여줄
# tf.data.AUTOTUNE는 데이터를 로드하고 다음 데이터를 미리 준비하여 데이터를 추출하는
train_ds = train_ds.map(
    lambda img, label: (data_augmentation(img), label),
    num_parallel_calls=tf.data.AUTOTUNE,
)
# Prefetching 을 이용하여 train, validatoin 데이터 셋에도 AUTOTUNE옵션을 적용함으로서
train_ds = train_ds.prefetch(tf.data.AUTOTUNE)
val_ds = val_ds.prefetch(tf.data.AUTOTUNE)
```

Build a model

We'll build a small version of the Xception network. We haven't particularly tried to optimize the architecture; if you want to do a systematic search for the best model configuration, consider using KerasTuner.

Note that:

- We start the model with the `data_augmentation` preprocessor, followed by a `Rescaling` layer.
- We include a `Dropout` layer before the final classification layer.

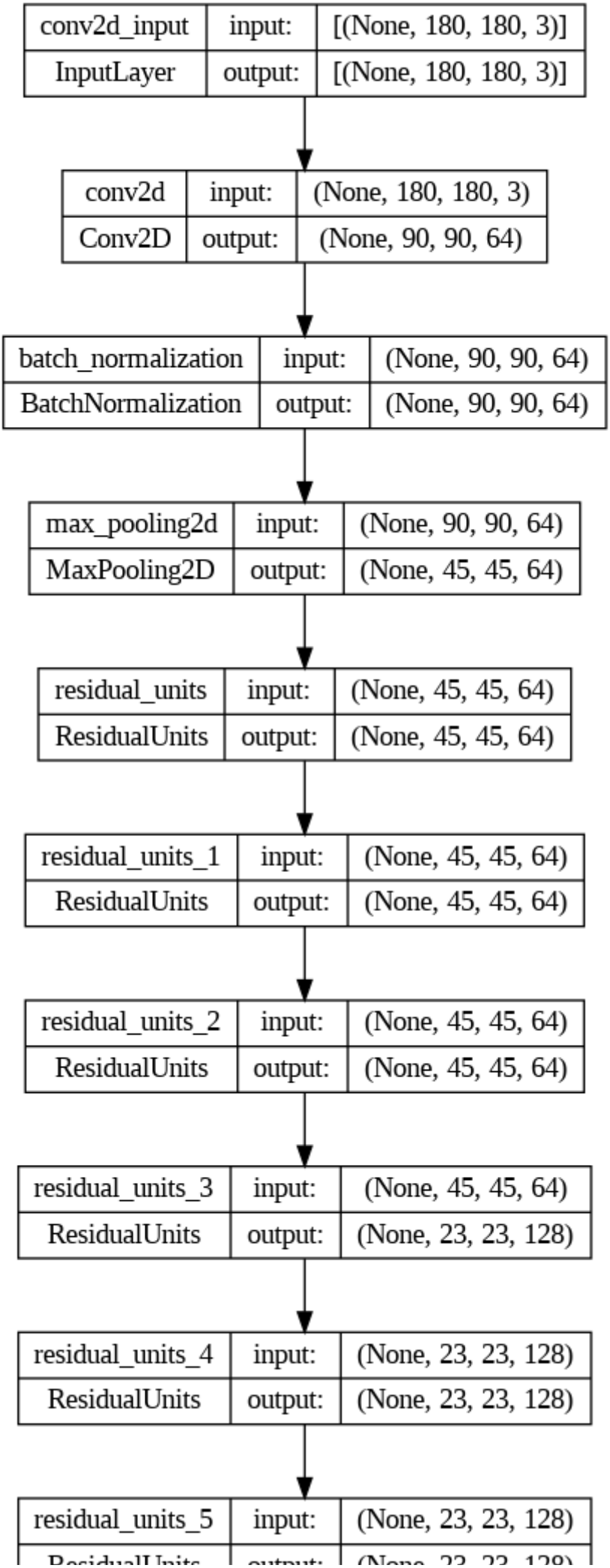
In [ ]:
```
'''
    ResNet-34 모델에서 사용되는 Residual Unit을 구현하기 위한 객체이다.
    주 된 역할은 main_layer와 skip_layer를 구성하는 것이며 조건에 따라 skip_layer를
    객체를 선언하고 호출할 때 조건에 맞는 skip_layer가 구성되며 main_layer의 결과물과
'''
class ResidualUnits(keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        # keras.activations은 keras 패키지 내에 저장되어 있는 activation을 불러온다.
        self.activation = keras.activations.get(activation)
        # Residual unit에 사용될 main layer를 정의한다.
        self.main_layers = [
            # 2D Convolution layer를 선언한다. ResidualUnits 객체를 선언하며 입력한
            keras.layers.Conv2D(filters, 3, strides=strides, padding="same", use_bia

            # 한 배치단위 내에서 정규화를 수행한다.
            keras.layers.BatchNormalization(),

            # activation layer를 정의해준다. 기본 activation function은 relu 이다.
            self.activation,

            keras.layers.Conv2D(filters, 3, strides=1, padding="same", use_bias=Fal
            keras.layers.BatchNormalization()]

        self.skip_layers = []
        # strides라는 조건을 두어 조건을 만족하는 특정상황에서는 skip_layers가 Convo
        if strides > 1:
            self.skip_layers = [
```

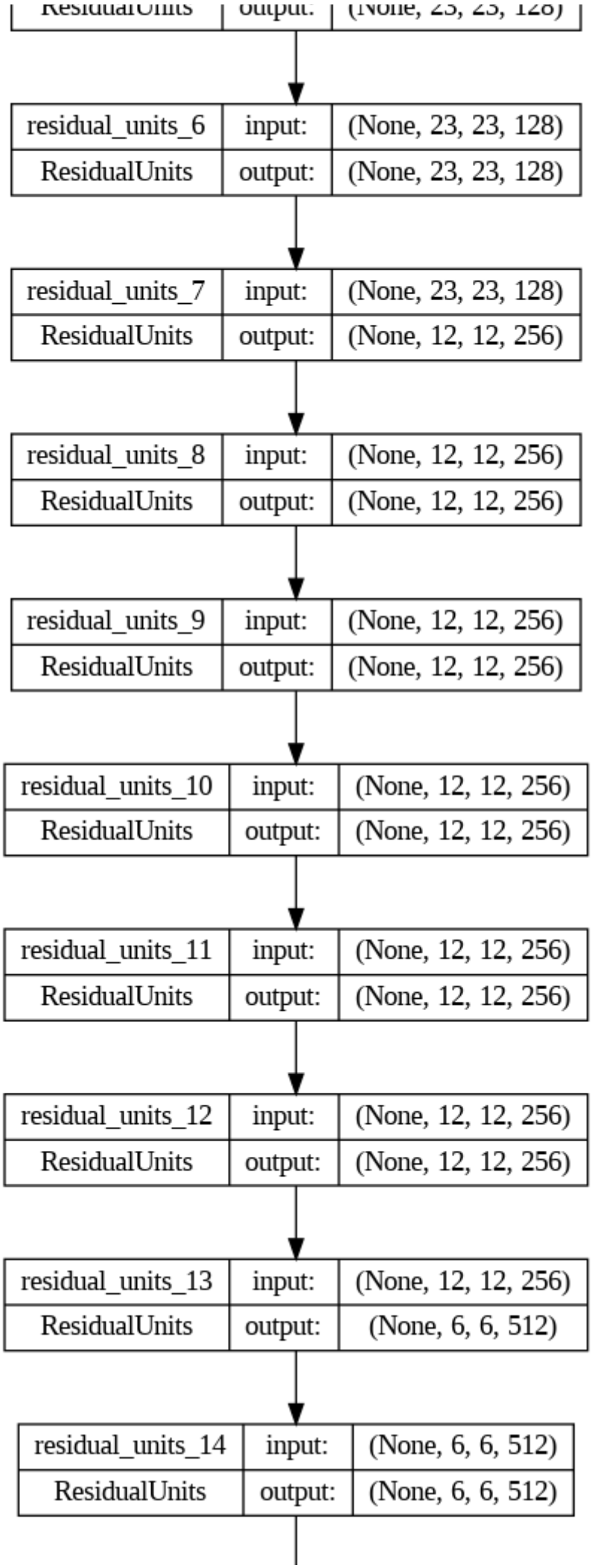
```
                keras.layers.Conv2D(filters, 1, strides=strides, padding="same", use
                keras.layers.BatchNormalization()]

    # call 메소드는 ResidualUnits 객체를 호출 할 때 실행되는 함수이다.
    def call(self, inputs):
        # 호출 할 때 인자로 입력해준 inputs 값을 z에 할당한다.
        z = inputs
        # main_layers의 layer들을 하나씩 꺼내어 z를 통과시킨다.
        for layer in self.main_layers:
            z = layer(z)
        # skip_z를 생성하여 inputs을 할당한다.
        skip_z = inputs
        # skip_layers의 layer들을 하나씩 꺼내어 skip_z를 통과시킨다.
        for layer in self.skip_layers:
            skip_z = layer(skip_z)
        # 최종적으로 z와 skip_z의 요소를 합하여 activation layer를 통과시킨 후 반환한
        return self.activation(z + skip_z)
```

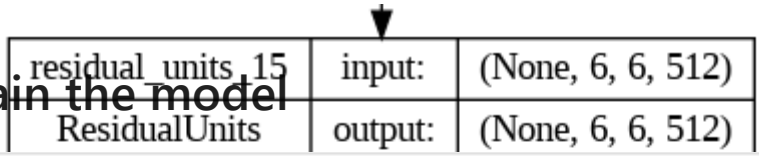
In [ ]:

```
'''
    모델을 선언하기 위한 단계이다. 이번에 구현하는 모델은 ResNet-34이라는 모델이다
    이전에 언급한 바와 같이 우선 keras.models.Sequential()를 model로 정의해 준 후, a
'''

model = keras.models.Sequential()

# 2D Convolution layer를 선언한다. input_shape는 (180, 180) 으로 정의한 image_size 0
model.add(keras.layers.Conv2D(64, 7, strides=2, input_shape=image_size + (3,), padd

# Normalization layer를 선언한다
model.add(keras.layers.BatchNormalization())

# Max pooling layer를 선언한다.
model.add(keras.layers.MaxPool2D(pool_size=3, strides=2, padding="same"))

prev_filters = 64
# Residual Units을 반복하여 모델에 추가한다.
# 64를 3번, 128을 4번, 256을 6번, 512를 3번 반복한다.
for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:

    # 이전 filters와 현재 filters를 비교하여 같으면 strides 를 1, 다르다면 2를 할당한
    strides = 1 if filters == prev_filters else 2

    model.add(ResidualUnits(filters, strides=strides))
    prev_filters = filters

# Average pooling layer를 추가한다.
model.add(keras.layers.GlobalAveragePooling2D())

# Fully Connected Layer에 추가하기 이전 1차원 데이터로 flatten하는 layer를 추가한다.
model.add(keras.layers.Flatten())

# Fully Connected Layer를 추가한다. 현재 학습은 이미지가 0 인지 1 인지, 즉 개인지 고
model.add(keras.layers.Dense(1, activation="sigmoid"))

# 모델의 구조를 그림으로 출력하여 직접 눈으로 확인 할 수 있다.
keras.utils.plot_model(model, show_shapes=True)
```

Out[ ]:


conv2d_input | input: | [(None, 180, 180, 3)]
InputLayer | output: | [(None, 180, 180, 3)]
conv2d | input: | (None, 180, 180, 3)
Conv2D | output: | (None, 90, 90, 64)
batch_normalization | input: | (None, 90, 90, 64)
BatchNormalization | output: | (None, 90, 90, 64)
max_pooling2d | input: | (None, 90, 90, 64)
MaxPooling2D | output: | (None, 45, 45, 64)
residual_units | input: | (None, 45, 45, 64)
ResidualUnits | output: | (None, 45, 45, 64)
residual_units_1 | input: | (None, 45, 45, 64)
ResidualUnits | output: | (None, 45, 45, 64)
residual_units_2 | input: | (None, 45, 45, 64)
ResidualUnits | output: | (None, 45, 45, 64)
residual_units_3 | input: | (None, 45, 45, 64)
ResidualUnits | output: | (None, 23, 23, 128)
residual_units_4 | input: | (None, 23, 23, 128)
ResidualUnits | output: | (None, 23, 23, 128)
residual_units_5 | input: | (None, 23, 23, 128)


ResidualUnits output: (None, 23, 23, 128)
residual_units_6 input: (None, 23, 23, 128)
ResidualUnits output: (None, 23, 23, 128)
residual_units_7 input: (None, 23, 23, 128)
ResidualUnits output: (None, 12, 12, 256)
residual_units_8 input: (None, 12, 12, 256)
ResidualUnits output: (None, 12, 12, 256)
residual_units_9 input: (None, 12, 12, 256)
ResidualUnits output: (None, 12, 12, 256)
residual_units_10 input: (None, 12, 12, 256)
ResidualUnits output: (None, 12, 12, 256)
residual_units_11 input: (None, 12, 12, 256)
ResidualUnits output: (None, 12, 12, 256)
residual_units_12 input: (None, 12, 12, 256)
ResidualUnits output: (None, 12, 12, 256)
residual_units_13 input: (None, 12, 12, 256)
ResidualUnits output: (None, 6, 6, 512)
residual_units_14 input: (None, 6, 6, 512)
ResidualUnits output: (None, 6, 6, 512)



Train the model

In [ ]:
```
'''
    구성한 모델을 이용하여 실제로 학습을 진행하는 과정이다.
'''

# train 데이터를 전부 활용하는 것을 1 epoch라고 한다. 5라고 선언해 줌으로서 train 데
# 이번 학습에 5 epochs가 충분하지 않으나 시간과 연산 성능에 한계가 있으므로 5 epochs
epochs = 5

# 모델을 컴파일 하는 단계이다. 모델을 평가하는 과정에서 사용되는 loss와 metrics를 정
# 현재 optimizer는 adam을 사용하며 learning rate로는 1e-3을 사용한다.
model.compile(
    optimizer=keras.optimizers.Adam(1e-3),
    loss="binary_crossentropy",
    metrics=["accuracy"],
)

# 모델이 실제로 학습을 시작하게 하는 함수이다.
# 이전 학습과는 다르게 x, y 로 구성된 학습 데이터 셋과 validation_split으로 특정 비율
# train_ds 와 val_ds 라는 tensorflow의 _BatchDataset 객체를 그대로 학습에 이용한다.
model.fit(
    train_ds,
    epochs=epochs,
    validation_data=val_ds,
)
```

```
Epoch 1/5
147/147 [==============================] - 142s 730ms/step - loss: 0.7046 - accurac
y: 0.6501 - val_loss: 4.5464 - val_accuracy: 0.5162
Epoch 2/5
147/147 [==============================] - 107s 718ms/step - loss: 0.5552 - accurac
y: 0.7152 - val_loss: 1.3103 - val_accuracy: 0.6047
Epoch 3/5
147/147 [==============================] - 103s 692ms/step - loss: 0.5035 - accurac
y: 0.7531 - val_loss: 1.3584 - val_accuracy: 0.5942
Epoch 4/5
147/147 [==============================] - 106s 708ms/step - loss: 0.4592 - accurac
y: 0.7836 - val_loss: 0.5733 - val_accuracy: 0.7078
Epoch 5/5
147/147 [==============================] - 104s 698ms/step - loss: 0.4025 - accurac
y: 0.8173 - val_loss: 1.6005 - val_accuracy: 0.5167
```

Out[ ]:
```
<keras.callbacks.History at 0x7a7d2382f880>
```

We get to >90% validation accuracy after training for 25 epochs on the full dataset (in practice, you can train for 50+ epochs before validation performance starts degrading).

Run inference on new data

Note that data augmentation and dropout are inactive at inference time.

In [ ]:
```
'''
    학습이 종료되고 모델을 평가하는 부분이다.
    시간과 연산성능이 충분하지 않기 때문에 test 데이터 셋을 만들어 검증하는 것이 아닌
    특정 사진 한장을 시각화 해서 확인 한 후 모델이 어떻게 평가하는지를 눈으로 확인 할
'''

# 학습에 사용한 데이터 중 특정 사진을 불러온다.
```

```
img = keras.utils.load_img(
    "PetImages/Cat/6779.jpg", target_size=image_size
)

# 불러온 이미지를 표시하여 눈으로 확인한다.
plt.imshow(img)
plt.axis("off")
img_array = keras.utils.img_to_array(img)
img_array = tf.expand_dims(img_array, 0)

# 모델을 통해 해당 이미지의 class를 예측한다.
predictions = model.predict(img_array)

# 모델이 예측한 해당 이미지의 class를 score로 표현하고 이를 표시한다.
score = float(predictions[0])
print(f"This image is {100 * (1 - score):.2f}% cat and {100 * score:.2f}% dog.")
```

```
1/1 [==============================] - 1s 927ms/step
This image is 99.53% cat and 0.47% dog.
```

An RNN example — forecasting a time series

In [ ]:
```
import numpy as np #numpy는 수치해석용 파이썬 라이브러리로 np라는 축약어로 import한
import tensorflow as tf #Tensorflow 2.0 부터 Keras가 내부 패키지로 포함 되었다. 기존
from tensorflow import keras #Tensorflow로 부터 keras를 import한다.
import matplotlib.pyplot as plt #matplotlib은 자료를 차트나 그래프로 시각화하는 라이
```

Generate the Dataset

In [ ]:
```
'''
    def를 사용하여 사용자 정의함수를 만들 수 있다. 이 예제에서는 sin값으로 time seri
    이 함수의 이름은 generate_time_series이고, 입력값으로는 batch_size 와 n_steps를
    이 부분은 함수를 정의하는 부분이고, 사용할 때 입력값을 정의해줘야 된다.
'''


def generate_time_series(batch_size, n_steps):

    # 각 네개의 변수에 차례대로 np.random.rand 함수의 값을 받는다.
    # numpy의 random 서브패키지는 난수를 생성하는 여러 함수를 포함하고있다. 그 중 ra
    # 이 예제의 경우 (10, 1) array를 총 4개 생성해준다.
    freq1, freq2, offsets1, offsets2 = np.random.rand(4, batch_size, 1)

    # numpy 라이브러리에 있는 함수로써 1차원 배열을 만들 수 있고, 그래프를 그릴 때 수
    # 예를 들어 np.linspace(0, 1, n_steps)은 0부터 1까지의 숫자를 길이 n_steps짜리의
    time = np.linspace(0, 1, n_steps)

    # 두 개의 sine wave를 더한 후 noise를 추가해준다
    series = 0.5 * np.sin((time - offsets1) * (freq1 * 10 + 10)) # wave 1
    series += 0.2 * np.sin((time - offsets2) * (freq2 * 20 + 20)) # + wave 2
    series += 0.1 * (np.random.rand(batch_size, n_steps) - 0.5) # + noise

    return series[..., np.newaxis].astype(np.float32)
```

In [ ]:
```
# 특정 시작 숫자를 정해주면 일정한 알고리즘에 의해 마치 난수처럼 보이는 수들을 쏟아니
# 이러한 시작 숫자를 seed라고 하며 이 숫자를 고정해주면 난수를 제어할 수 있다f
np.random.seed(42)

n_steps = 50

# generate_time_series 함수를 이용하여 (10000, 51, 1)의 time series data를 만들었다.
series = generate_time_series(10000, n_steps + 1)

# train, validation 그리고 test dataset을 쪼개어 나누었다.

X_train, y_train = series[:7000, :n_steps], series[:7000, -1]
X_valid, y_valid = series[7000:, :n_steps], series[7000:, -1]
```

Computing Some Baselines

In [ ]:
```
'''
    모델을 선언하는 과정이다.
    keras.Sequential은 선언된 각 레이어를 순서대로 실행시켜 주는 함수이며 아래와 같이
'''
model = keras.models.Sequential([
```

```
    # 은닉상태의 크기가 20인 simpleRNN 모델을 정의한다. 이 모델은 이전 timestep의 출
    # return_sequences = False 인 경우에는 마지막 시점의 은닉 상태만 출력한다. True일
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),

    # 은닉상태의 크기가 20인 simpleRNN 모델을 정의한다.
    keras.layers.SimpleRNN(20),

    # time step마다 출력값이 한 개 이므로 노드 한 개의 dense layer를 정의한다.
    keras.layers.Dense(1)
])


# 모델 구성에 대해서 확인할 수 있게 기본으로 제공해 주는 함수이다.
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn (SimpleRNN)      (None, None, 20)          440

 simple_rnn_1 (SimpleRNN)    (None, 20)                820

 dense (Dense)               (None, 1)                 21

=================================================================
Total params: 1,281
Trainable params: 1,281
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:

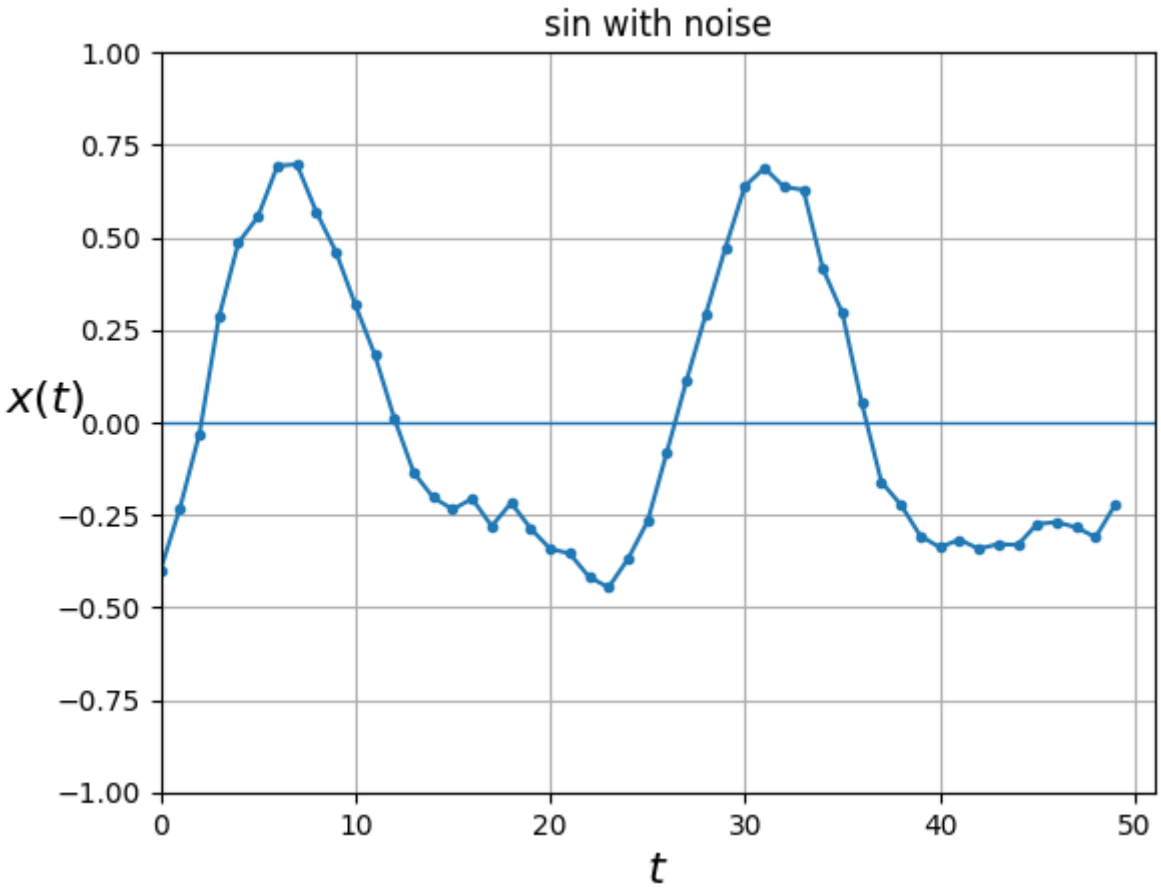
```
'''
 모델을 컴파일 하는 단계이다. 모델을 평가하는 과정에서 사용되는 loss와 metrics를 정의
'''

model.compile(loss="mse", optimizer="adam")

# 모델이 실제로 학습을 시작하게 하는 함수이다.
# 학습에 사용되는 x, y 데이터를 입력하고, batch_size와 epochs를 정의 할 수 있다.
# validation은 미리 정의해둔 validation dataset을 사용하면 된다. 또는 비율을 지정해
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
```

```
Epoch 1/20
219/219 [==============================] - 8s 27ms/step - loss: 0.0294 - val_loss:
0.0063
Epoch 2/20
219/219 [==============================] - 6s 27ms/step - loss: 0.0047 - val_loss:
0.0044
Epoch 3/20
219/219 [==============================] - 5s 23ms/step - loss: 0.0037 - val_loss:
0.0035
Epoch 4/20
219/219 [==============================] - 6s 29ms/step - loss: 0.0035 - val_loss:
0.0033
Epoch 5/20
219/219 [==============================] - 5s 24ms/step - loss: 0.0033 - val_loss:
0.0034
Epoch 6/20
219/219 [==============================] - 7s 31ms/step - loss: 0.0032 - val_loss:
0.0033
Epoch 7/20
219/219 [==============================] - 5s 23ms/step - loss: 0.0033 - val_loss:
0.0031
Epoch 8/20
219/219 [==============================] - 6s 26ms/step - loss: 0.0032 - val_loss:
0.0033
Epoch 9/20
219/219 [==============================] - 5s 25ms/step - loss: 0.0031 - val_loss:
0.0031
Epoch 10/20
219/219 [==============================] - 5s 22ms/step - loss: 0.0030 - val_loss:
0.0029
Epoch 11/20
219/219 [==============================] - 6s 29ms/step - loss: 0.0030 - val_loss:
0.0031
Epoch 12/20
219/219 [==============================] - 5s 22ms/step - loss: 0.0030 - val_loss:
0.0030
Epoch 13/20
219/219 [==============================] - 6s 29ms/step - loss: 0.0030 - val_loss:
0.0030
Epoch 14/20
219/219 [==============================] - 5s 25ms/step - loss: 0.0030 - val_loss:
0.0035
Epoch 15/20
219/219 [==============================] - 5s 25ms/step - loss: 0.0031 - val_loss:
0.0029
Epoch 16/20
219/219 [==============================] - 6s 27ms/step - loss: 0.0029 - val_loss:
0.0028
Epoch 17/20
219/219 [==============================] - 5s 22ms/step - loss: 0.0029 - val_loss:
0.0028
Epoch 18/20
219/219 [==============================] - 6s 29ms/step - loss: 0.0030 - val_loss:
0.0030
Epoch 19/20
219/219 [==============================] - 5s 22ms/step - loss: 0.0029 - val_loss:
0.0028
Epoch 20/20
219/219 [==============================] - 7s 31ms/step - loss: 0.0029 - val_loss:
0.0026
```

Forecasting Several Steps Ahead

In [ ]:
```
n_steps = 50 # 총 time step 수 이다.
pred_steps = 10 # 총 예측 할 step 수 이다
series = generate_time_series(1, n_steps + pred_steps) # 위에서 정의한 generate_time
X, Y = series[:, :n_steps], series[:, n_steps:] # training에 쓰일 data와 test에 쓰일
```

In [ ]:
```
# 생성한 time series dataset을 plot으로 시각화해주는 함수이다.
# [batchsize, time steps, 1]을 return해주는 함수이다
def plot_series(series, y=None, y_pred=None, x_label="$t$", y_label="$x(t)$", lege
    plt.plot(series, ".-")
    if y is not None:
        plt.plot(n_steps, y, "bo", label="Target")  # bo: 파란색의 원형 마커 (blue +
    if y_pred is not None:
        plt.plot(n_steps, y_pred, "rx", markersize=10, label="Prediction") # rx : 빨
    plt.grid(True) # grid를 추가해준다
    if x_label:
        plt.xlabel(x_label, fontsize=16) # fontsize 인자를 통해 글자의 크기를 지정해
    if y_label:
        plt.ylabel(y_label, fontsize=16, rotation=0) # rotation 인자를 통해 label의
    plt.hlines(0, 0, 100, linewidth=1) # (y, xmin, xmax)이므로 xmin~xmax까지 y값의 =
    plt.axis([0, n_steps + 1, -1, 1])
    if legend and (y or y_pred):
        plt.legend(fontsize=14, loc="upper left")

# 위에서 만든 series data를 시각화 한 결과이다
plt.title("sin with noise")
plot_series(X[0, :, 0])
```

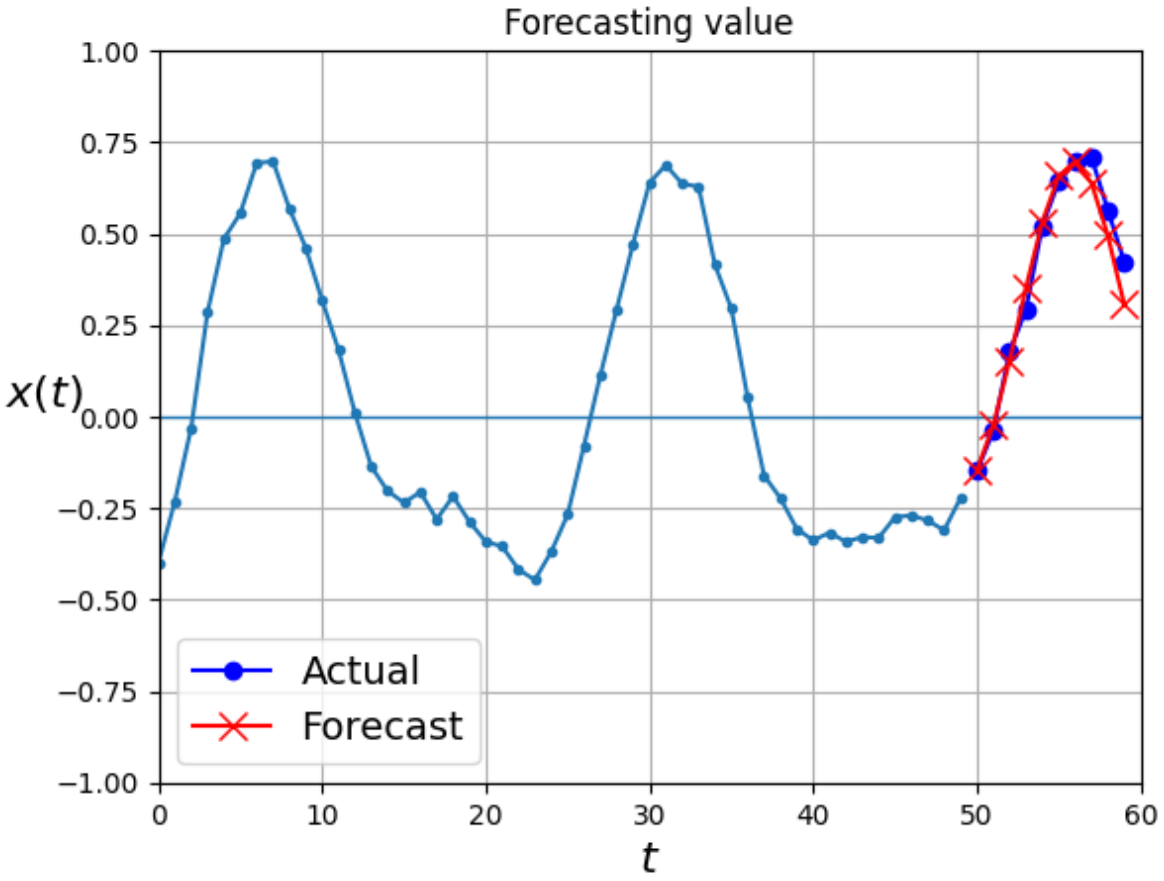


In [ ]:
```
for step_ahead in range(pred_steps):
    y_pred_one = model.predict(X[:, step_ahead:])[:, np.newaxis, :]
    X = np.concatenate([X, y_pred_one], axis=1)

Y_pred = X[:, n_steps:]
```

```
1/1 [==============================] - 0s 323ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 33ms/step
```

In [ ]:

```
# 예측 결과와 함께 시각화하기 위해 만든 함수이다.
def plot_multiple_forecasts(X, Y, Y_pred):
    n_steps = X.shape[1]
    ahead = Y.shape[1]
    plot_series(X[0, :, 0])
    plt.plot(np.arange(n_steps, n_steps + ahead), Y[0, :, 0], "bo-", label="Actual")
    plt.plot(np.arange(n_steps, n_steps + ahead), Y_pred[0, :, 0], "rx-", label="For
    plt.axis([0, n_steps + ahead, -1, 1])
    plt.legend(fontsize=14)

plt.title("Forecasting value")
plot_multiple_forecasts(X[:, :n_steps], Y, Y_pred)
plt.show()
```



An LSTM example

In [19]:
```
#!는 Collab에서 리눅스 쉘 명령어를 수행하기 위한 명령어이다.
#finance-datareader를 설치하는 명령어 이다.
! pip install -U finance-datareader
```

```
Requirement already satisfied: finance-datareader in c:₩users₩user₩miniconda3₩lib₩si
te-packages (0.9.50)
Requirement already satisfied: requests>=2.3.0 in c:₩users₩user₩miniconda3₩lib₩site-
packages (from finance-datareader) (2.28.1)
Requirement already satisfied: requests-file in c:₩users₩user₩miniconda3₩lib₩site-pa
ckages (from finance-datareader) (1.5.1)
Requirement already satisfied: lxml in c:₩users₩user₩miniconda3₩lib₩site-packages (f
rom finance-datareader) (4.9.3)
Requirement already satisfied: pandas>=0.19.2 in c:₩users₩user₩miniconda3₩lib₩site-p
ackages (from finance-datareader) (2.0.3)
Requirement already satisfied: tqdm in c:₩users₩user₩miniconda3₩lib₩site-packages (f
rom finance-datareader) (4.65.0)
Requirement already satisfied: tzdata>=2022.1 in c:₩users₩user₩miniconda3₩lib₩site-p
ackages (from pandas>=0.19.2->finance-datareader) (2023.3)
Requirement already satisfied: python-dateutil>=2.8.2 in c:₩users₩user₩miniconda3₩li
b₩site-packages (from pandas>=0.19.2->finance-datareader) (2.8.2)
Requirement already satisfied: numpy>=1.21.0 in c:₩users₩user₩miniconda3₩lib₩site-pa
ckages (from pandas>=0.19.2->finance-datareader) (1.25.2)
Requirement already satisfied: pytz>=2020.1 in c:₩users₩user₩miniconda3₩lib₩site-pac
kages (from pandas>=0.19.2->finance-datareader) (2023.3)
Requirement already satisfied: charset-normalizer<3,>=2 in c:₩users₩user₩miniconda3
₩lib₩site-packages (from requests>=2.3.0->finance-datareader) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in c:₩users₩user₩miniconda3₩lib₩site-pac
kages (from requests>=2.3.0->finance-datareader) (3.4)
Requirement already satisfied: certifi>=2017.4.17 in c:₩users₩user₩miniconda3₩lib₩si
te-packages (from requests>=2.3.0->finance-datareader) (2022.12.7)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in c:₩users₩user₩miniconda3₩lib
₩site-packages (from requests>=2.3.0->finance-datareader) (1.26.15)
Requirement already satisfied: six in c:₩users₩user₩miniconda3₩lib₩site-packages (fr
om requests-file->finance-datareader) (1.16.0)
Requirement already satisfied: colorama in c:₩users₩user₩miniconda3₩lib₩site-package
s (from tqdm->finance-datareader) (0.4.6)
```

In [11]:
```
import FinanceDataReader as fdr # python의 한국 주식 가격, 미국 주식 가격, 지수, 환

#numpy는 수치 연산에 효율적이고, pandas는 데이터 조작 및 분석 등의 작업에 효율적이다
import pandas as pd # 데이터 처리를 위한 라이브러리이다. pd라는 축약어로 import한다
import numpy as np #numpy는 수치해석용 파이썬 라이브러리로 np라는 축약어로 import한
import matplotlib.pyplot as plt  #matplotlib은 자료를 차트나 그래프로 시각화하는 라

import tensorflow as tf #Tensorflow 2.0 부터 Keras가 내부 패키지로 포함 되었다. 기존
from tensorflow import keras #Tensorflow로 부터 keras를 import한다.
from sklearn.preprocessing import StandardScaler #scikit-learn 라이브러리의 데이터
```

Generate the Dataset

In [12]:
```
'''
    def를 사용하여 사용자 정의함수를 만들 수 있다. 이 예제에서는 주식 데이터를 읽는
    이 함수의 이름은 generate_stock_data이고, 입력값으로는 batch_size, n_steps 그리고
    이 부분은 함수를 정의하는 부분이고, 사용할 때 입력값을 정의해줘야 된다.
'''
```

```
def generate_stock_data(batch_size, n_steps, kospi=False):
    ss = StandardScaler()
    if kospi:
        df = fdr.DataReader('KS11').reset_index() # fdr 의 DataReader 에 코스피 지수

        # drop은 지정한 변수를 삭제해준다. axis = 1 은 열방향으로 동작하라는 의미이다
        # inplace=True는 drop을 실행한 dataframe이 바로 적용되어 저장된다는 의미이다
        # 이번 예제에서는 Close 데이터만 사용하기 위해 나머지 열을 제거해 주었다.
        df.drop(['Open', 'High', 'Low', 'Volume', 'Adj Close'], axis=1, inplace=Tru

        # groupby 함수는 주어진 데이터를 그룹 별로 구분하여 데이터를 보기 위해 사용되
        # groupby 함수와 mean 함수를 적용 해 줌으로서 월별 평균 데이터를 얻을 수 있다
        df = df.groupby([df['Date'].dt.year, df['Date'].dt.month]).mean()

        # reset_index는 인덱스를 새로 reset 해주는 함수이다. drop=True로 설정하면 기
        df.reset_index(drop=True, inplace=True)

        # 상장된지 얼마 지나지 않은 주식의 경우 최근 데이터만 가지고 있기 때문에 원하
        # 때문에 원하는 길이의 n_steps 부터 상장 개시일 까지의 주가를 0으로 맞추어 학
        if len(df) < n_steps:
            pad_len = n_steps - len(df)
            pad = np.zeros((pad_len, 1))
            df = np.concatenate((pad, df), axis=0)
        series = np.zeros((1, n_steps, 1))
        # 주식 데이터의 경우 종목마다 스케일이 다르기 때문에 정규화를 시켜준다.
        series[0, :, :] = ss.fit_transform(df[:n_steps])
        return series

    # fdr 라이브러리에서 제공하는 한국 증권 시장 정보를 불러온다.
    df_krx = fdr.StockListing('KRX')
    # 한국 증권시장 정보 중, 종목코드를 batch_size만큼 랜덤하게 불러와 random_code에
    random_code = df_krx['Code'].sample(n=batch_size)
    random_code.reset_index(drop=True, inplace=True)
    series = np.zeros((batch_size, n_steps, 1))

    # 랜덤하게 불러온 종목 코드의 월별 평균 종가를 반복문을 이용하여 불러온다.
    for idx, code in enumerate(random_code):
        df = fdr.DataReader(code).reset_index()
        df.drop(['Open', 'High', 'Low', 'Volume', 'Change'], axis=1, inplace=True)
        df = df.groupby([df['Date'].dt.year, df['Date'].dt.month]).mean()
        df.reset_index(drop=True, inplace=True)
        # 상장된지 얼마 지나지 않은 주식의 경우 최근 데이터만 가지고 있기 때문에 원하
        # 때문에 원하는 길이의 n_steps 부터 상장 개시일 까지의 주가를 0으로 맞추어 학
        if len(df) < n_steps:
            pad_len = n_steps - len(df)
            pad = np.zeros((pad_len, 1))
            df = np.concatenate((pad, df), axis=0)
        # 주식 데이터의 경우 종목마다 스케일이 다르기 때문에 정규화를 시켜준다.
        series[idx, :, :] = ss.fit_transform(df[:n_steps])
    return series
```

In [13]:

```
'''
    학습을 위한 데이터 셋을 마련하는 부분이다.
'''

# 특정 시작 숫자를 정해주면 일정한 알고리즘에 의해 마치 난수처럼 보이는 수들을 쏟아내
# 이러한 시작 숫자를 seed라고 하며 이 숫자를 고정해주면 난수를 제어할 수 있다
np.random.seed(42)

n_steps = 50
# generate_stock_data 함수를 이용하여 (100, 50 + 1, 1)의 time series data를 만들었다
series = generate_stock_data(100, n_steps + 1)
X_train, y_train = series[:70, :n_steps], series[:70, -1]
```

```
X_valid, y_valid = series[70:90, :n_steps], series[70:90, -1]
X_test, y_test = series[90:, :n_steps], series[90:, -1]
```

Computing Some Baselines

In [14]:
```
'''
    모델을 선언하고 학습을 진행하는 과정이다.
    keras.Sequential은 선언된 각 레이어를 순서대로 실행시켜 주는 함수이며 아래와 같이
    50 + 1의 steps을 가지는 데이터 셋을 만들었기 때문에 50개 steps의 데이터를 통해 다
'''

model = keras.models.Sequential([
    # 은닉상태의 크기가 20인 simpleRNN 모델을 정의한다. 이 모델은 이전 timestep의 출
    # return_sequences = False 인 경우에는 마지막 시점의 은닉 상태만 출력한다. True일
    keras.layers.LSTM(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.LSTM(20),
    # time step마다 출력값이 한 개 이므로 노드 한 개의 dense layer를 정의한다.
    keras.layers.Dense(1)
])

# 모델을 컴파일 하는 단계이다. 모델을 평가하는 과정에서 사용되는 loss와 metrics를 정
model.compile(loss="mse", optimizer="adam")

# 모델이 실제로 학습을 시작하게 하는 함수이다.
# 학습에 사용되는 x, y 데이터를 입력하고, batch_size와 epochs를 정의 할 수 있다.
# validation은 미리 정의해둔 validation dataset을 사용하면 된다. 또는 비율을 지정해
history = model.fit(X_train, y_train, epochs=50,
                    validation_data=(X_valid, y_valid))
```

```
Epoch 1/50
3/3 [==============================] - 4s 399ms/step - loss: 1.8726 - val_loss: 1.24
25
Epoch 2/50
3/3 [==============================] - 0s 34ms/step - loss: 1.7177 - val_loss: 1.144
5
Epoch 3/50
3/3 [==============================] - 0s 37ms/step - loss: 1.5665 - val_loss: 1.055
4
Epoch 4/50
3/3 [==============================] - 0s 35ms/step - loss: 1.4339 - val_loss: 0.973
0
Epoch 5/50
3/3 [==============================] - 0s 37ms/step - loss: 1.3084 - val_loss: 0.906
6
Epoch 6/50
3/3 [==============================] - 0s 34ms/step - loss: 1.1850 - val_loss: 0.847
6
Epoch 7/50
3/3 [==============================] - 0s 34ms/step - loss: 1.0800 - val_loss: 0.792
1
Epoch 8/50
3/3 [==============================] - 0s 34ms/step - loss: 0.9842 - val_loss: 0.746
3
Epoch 9/50
3/3 [==============================] - 0s 34ms/step - loss: 0.8905 - val_loss: 0.706
5
Epoch 10/50
3/3 [==============================] - 0s 34ms/step - loss: 0.8185 - val_loss: 0.669
5
Epoch 11/50
3/3 [==============================] - 0s 34ms/step - loss: 0.7440 - val_loss: 0.631
3
Epoch 12/50
3/3 [==============================] - 0s 33ms/step - loss: 0.6879 - val_loss: 0.587
8
Epoch 13/50
3/3 [==============================] - 0s 35ms/step - loss: 0.6260 - val_loss: 0.547
1
Epoch 14/50
3/3 [==============================] - 0s 35ms/step - loss: 0.5708 - val_loss: 0.505
8
Epoch 15/50
3/3 [==============================] - 0s 34ms/step - loss: 0.5312 - val_loss: 0.471
0
Epoch 16/50
3/3 [==============================] - 0s 35ms/step - loss: 0.4936 - val_loss: 0.445
2
Epoch 17/50
3/3 [==============================] - 0s 34ms/step - loss: 0.4665 - val_loss: 0.434
9
Epoch 18/50
3/3 [==============================] - 0s 34ms/step - loss: 0.4395 - val_loss: 0.438
0
Epoch 19/50
3/3 [==============================] - 0s 34ms/step - loss: 0.4174 - val_loss: 0.446
7
Epoch 20/50
3/3 [==============================] - 0s 34ms/step - loss: 0.4046 - val_loss: 0.449
0
Epoch 21/50
3/3 [==============================] - 0s 34ms/step - loss: 0.3896 - val_loss: 0.415
0
Epoch 22/50
```

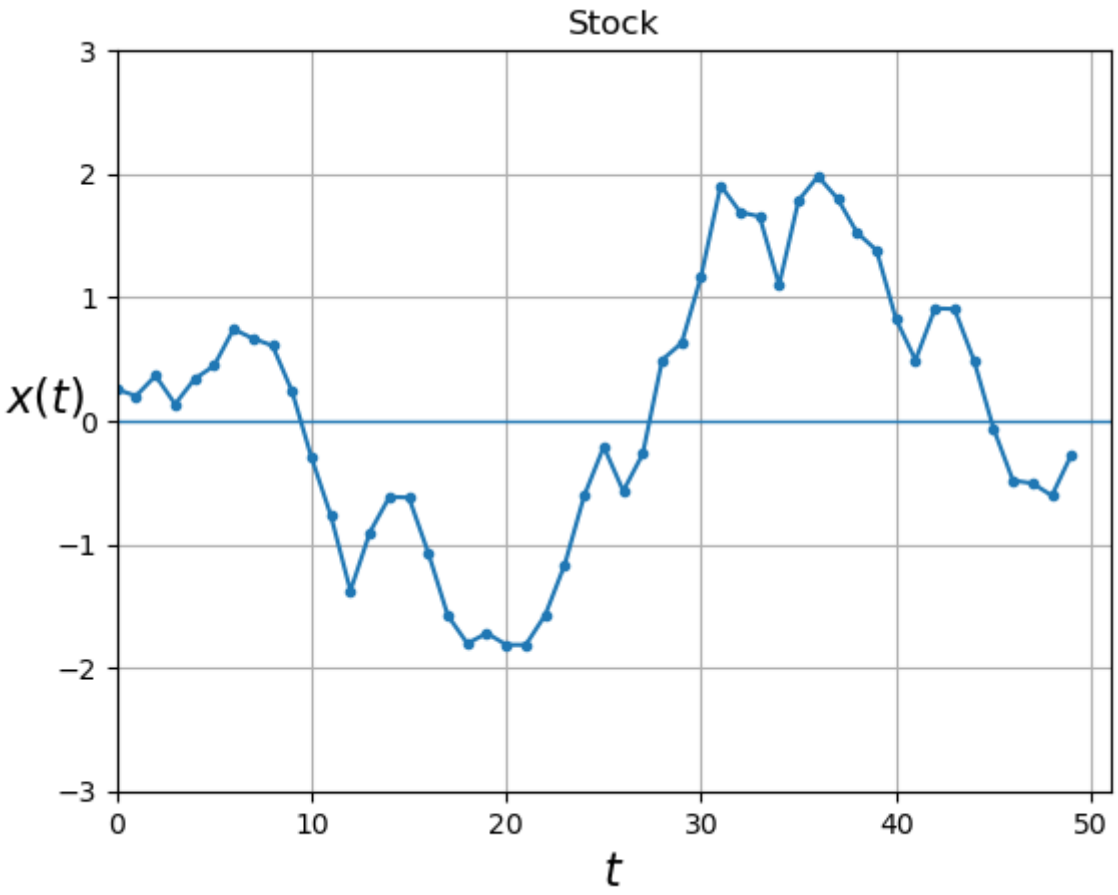
```
3/3 [==============================] - 0s 34ms/step - loss: 0.3746 - val_loss: 0.377
7
Epoch 23/50
3/3 [==============================] - 0s 34ms/step - loss: 0.3591 - val_loss: 0.352
6
Epoch 24/50
3/3 [==============================] - 0s 34ms/step - loss: 0.3488 - val_loss: 0.333
9
Epoch 25/50
3/3 [==============================] - 0s 34ms/step - loss: 0.3411 - val_loss: 0.325
7
Epoch 26/50
3/3 [==============================] - 0s 36ms/step - loss: 0.3304 - val_loss: 0.320
5
Epoch 27/50
3/3 [==============================] - 0s 35ms/step - loss: 0.3220 - val_loss: 0.318
2
Epoch 28/50
3/3 [==============================] - 0s 34ms/step - loss: 0.3142 - val_loss: 0.309
4
Epoch 29/50
3/3 [==============================] - 0s 33ms/step - loss: 0.3068 - val_loss: 0.308
2
Epoch 30/50
3/3 [==============================] - 0s 34ms/step - loss: 0.2989 - val_loss: 0.312
1
Epoch 31/50
3/3 [==============================] - 0s 38ms/step - loss: 0.2934 - val_loss: 0.302
8
Epoch 32/50
3/3 [==============================] - 0s 36ms/step - loss: 0.2877 - val_loss: 0.276
5
Epoch 33/50
3/3 [==============================] - 0s 34ms/step - loss: 0.2807 - val_loss: 0.259
1
Epoch 34/50
3/3 [==============================] - 0s 35ms/step - loss: 0.2737 - val_loss: 0.248
8
Epoch 35/50
3/3 [==============================] - 0s 34ms/step - loss: 0.2710 - val_loss: 0.243
1
Epoch 36/50
3/3 [==============================] - 0s 34ms/step - loss: 0.2660 - val_loss: 0.249
6
Epoch 37/50
3/3 [==============================] - 0s 36ms/step - loss: 0.2606 - val_loss: 0.255
1
Epoch 38/50
3/3 [==============================] - 0s 35ms/step - loss: 0.2561 - val_loss: 0.256
2
Epoch 39/50
3/3 [==============================] - 0s 34ms/step - loss: 0.2525 - val_loss: 0.244
7
Epoch 40/50
3/3 [==============================] - 0s 33ms/step - loss: 0.2445 - val_loss: 0.228
7
Epoch 41/50
3/3 [==============================] - 0s 34ms/step - loss: 0.2370 - val_loss: 0.225
4
Epoch 42/50
3/3 [==============================] - 0s 33ms/step - loss: 0.2365 - val_loss: 0.221
2
Epoch 43/50
3/3 [==============================] - 0s 36ms/step - loss: 0.2360 - val_loss: 0.222
```

```
5
Epoch 44/50
3/3 [==============================] - 0s 36ms/step - loss: 0.2227 - val_loss: 0.225
8
Epoch 45/50
3/3 [==============================] - 0s 33ms/step - loss: 0.2229 - val_loss: 0.222
3
Epoch 46/50
3/3 [==============================] - 0s 37ms/step - loss: 0.2181 - val_loss: 0.201
8
Epoch 47/50
3/3 [==============================] - 0s 34ms/step - loss: 0.2106 - val_loss: 0.189
0
Epoch 48/50
3/3 [==============================] - 0s 34ms/step - loss: 0.2127 - val_loss: 0.189
3
Epoch 49/50
3/3 [==============================] - 0s 34ms/step - loss: 0.2053 - val_loss: 0.201
0
Epoch 50/50
3/3 [==============================] - 0s 34ms/step - loss: 0.1997 - val_loss: 0.202
1
```

Forecasting Several Steps Ahead

In [15]:
```
'''
    위에서 학습된 모델을 이용하여 50개 steps을 가지고 추가적으로 10개 steps의 데이터

'''
n_steps = 50
pred_steps = 10
# test에 사용되는 종목은 코스피이다.
series = generate_stock_data(1, n_steps + pred_steps, kospi=True)
X, Y = series[:, :n_steps], series[:, n_steps:]
```

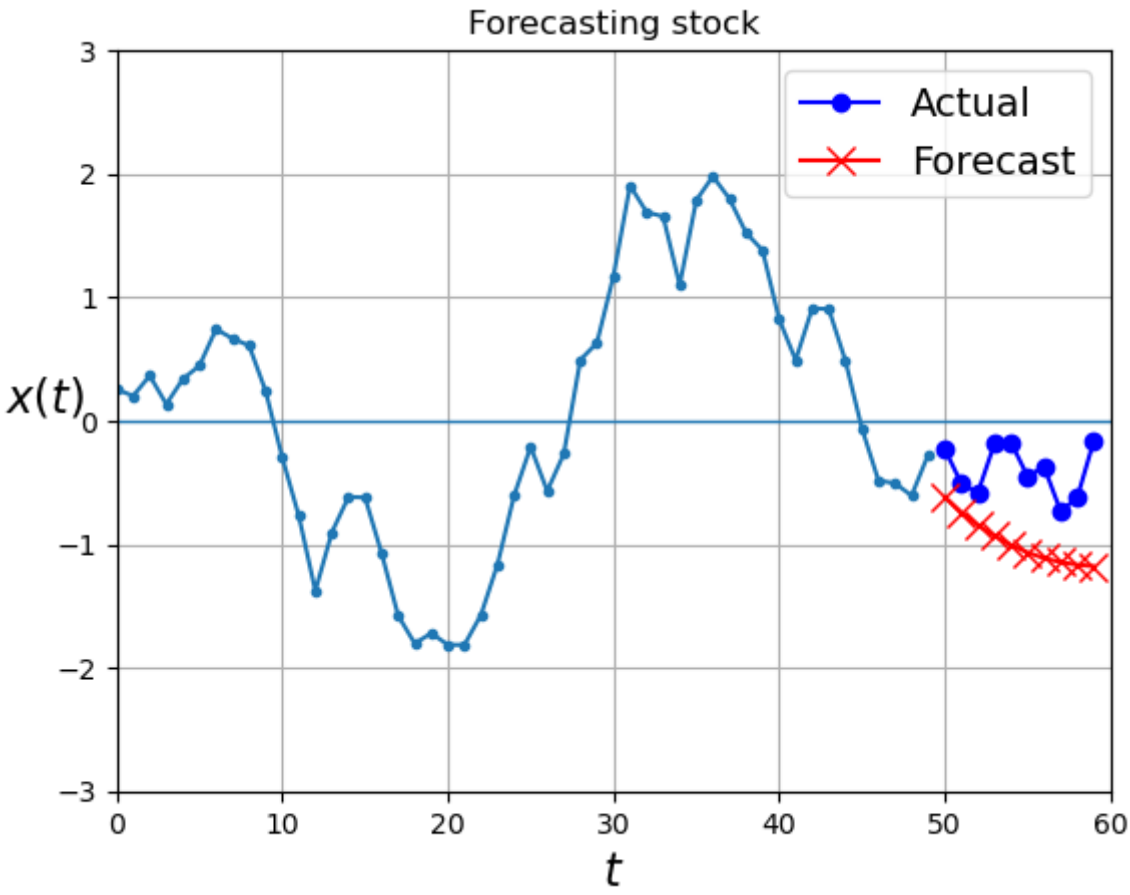
In [16]:
```
# 데이터를 시각화 하여 보여줄 수 있는 함수이다.
# 모델을 통해 예측을 하기 전, 데이터를 직접 눈으로 확인 할 수 있다.
# 또한 예측 후, 실제 데이터와 예측된 데이터의 차이까지 눈으로 확인 할 수 있는 함수이
def plot_series(series, y=None, y_pred=None, x_label="$t$", y_label="$x(t)$", lege
    plt.plot(series, ".-")
    if y is not None:
        plt.plot(n_steps, y, "bo", label="Target")
    if y_pred is not None:
        plt.plot(n_steps, y_pred, "rx", markersize=10, label="Prediction")
    plt.grid(True)
    if x_label:
        plt.xlabel(x_label, fontsize=16)
    if y_label:
        plt.ylabel(y_label, fontsize=16, rotation=0)
    plt.hlines(0, 0, 100, linewidth=1)
    plt.axis([0, n_steps + 1, -3, 3])
    if legend and (y or y_pred):
        plt.legend(fontsize=14, loc="upper left")

plt.title("Stock")
plot_series(X[0, :, 0])
```



In [17]:
```
# 학습된 모델을 통하여 추후 10개의 steps을 예측한다.
for step_ahead in range(pred_steps):
    y_pred_one = model.predict(X[:, step_ahead:])[:, np.newaxis, :]
    X = np.concatenate([X, y_pred_one], axis=1)

Y_pred = X[:, n_steps:]
```

```
1/1 [==============================] - 1s 611ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 16ms/step
```

In [18]:
```
# 위에서 작성한 plot_series를 활용하여 시각화 하는 함수이다.
def plot_multiple_forecasts(X, Y, Y_pred):
    n_steps = X.shape[1]
    ahead = Y.shape[1]
    plot_series(X[0, :, 0])
    plt.plot(np.arange(n_steps, n_steps + ahead), Y[0, :, 0], "bo-", label="Actual")
    plt.plot(np.arange(n_steps, n_steps + ahead), Y_pred[0, :, 0], "rx-", label="For
    plt.axis([0, n_steps + ahead, -3, 3])
    plt.legend(fontsize=14)

plt.title("Forecasting stock")
plot_multiple_forecasts(X[:, :n_steps], Y, Y_pred)
plt.show()
```


Forecasting stock
Actual
Forecast
$x(t)_0$
t
3
2
1
−1
−2
−3
0
10
20
30
40
50
60

Autoencoder example for fashion MNIST

In [ ]:
```
import numpy as np  # numpy는 수치해석용 파이썬 라이브러리로 np라는 축약어로 import
import tensorflow as tf #Tensorflow 2.0 부터 Keras가 내부 패키지로 포함 되었다. 기존
from tensorflow import keras    # Tensorflow로 부터 keras를 import한다.

import matplotlib as mpl    #matplotlib은 자료를 차트나 그래프로 시각화하는 라이브러
import matplotlib.pyplot as plt # pyplot이라는 서브패키지를 plt라는 축약어로 import
```

In [ ]:
```
'''
    데이터 셋을 준비하는 과정이다.
    이번에 사용하는 데이터는 fashion_mnist이며
    train, validation, test 데이터 셋을 분리한다.
'''


(X_train_full, y_train_full), (X_test, y_test) = keras.datasets.fashion_mnist.load_d
X_train_full = X_train_full.astype(np.float32) / 255
X_test = X_test.astype(np.float32) / 255
X_train, X_valid = X_train_full[:-5000], X_train_full[-5000:]
y_train, y_valid = y_train_full[:-5000], y_train_full[-5000:]
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/tr
ain-labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/tr
ain-images-idx3-ubyte.gz
26421880/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t1
0k-labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t1
0k-images-idx3-ubyte.gz
4422102/4422102 [==============================] - 0s 0us/step
```

In [ ]:
```
# y의 값을 반올림하여 accuracy를 구하는 함수이다.
# 바로 다음 모델을 학습하는데 사용 할 예정이다.
def rounded_accuracy(y_true, y_pred):
    return keras.metrics.binary_accuracy(tf.round(y_true), tf.round(y_pred))
```

In [ ]:
```
# 특정 시작 숫자를 정해주면 일정한 알고리즘에 의해 마치 난수처럼 보이는 수들을 쏟아내
# 이러한 시작 숫자를 seed라고 하며 이 숫자를 고정해주면 난수를 제어할 수 있다
tf.random.set_seed(42)
np.random.seed(42)

# autoencoder의 encoder 부분의 모델을 정의한다.
# (28, 28)의 2차원으로 이루어진 사진 데이터를 1차원 데이터로 flatten 하는 레이어를 추
# 100개의 output을 가진 FC layer를 추가하며 activation function은 "selu"를 사용한다.
# 30개의 output을 가진 FC layer를 추가하며 activation function은 "selu"를 사용한다.
stacked_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu"),
])
# autoencoder의 decoder 부분의 모델을 정의한다.
# 100개의 output을 가진 FC layer를 추가하며 activation function은 "selu"를 사용한다.
# 28*28개의 output을 가진 FC layer를 추가한다. activation function은 "sigmoid"를 사용
# (28*28)의 1차원으로 이루어진 데이터를 (28, 28)의 2차원 사진 데이터로 변형하기 위해
stacked_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
```

```
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

# 위에서 정의한 encoder와 decoder를 순차적으로 작동하는 autoencoder모델을 정의한다.
stacked_ae = keras.models.Sequential([stacked_encoder, stacked_decoder])
# autoencoder모델을 컴파일 한다.
# loss는 "binary_crossentropy"를 사용하며 optimizer로는 SGD, learning_rate는 1.5로 한
# 평가 metrics는 위에서 정의한 rounded_accuracy를 사용한다.
stacked_ae.compile(loss="binary_crossentropy",
                   optimizer=keras.optimizers.SGD(learning_rate=1.5), metrics=[round
# 모델을 학습한다.
# autoencoder에 넣어준 사진을 decoder가 그대로 복원하는 것을 목표로 하고 있기 때문에
history = stacked_ae.fit(X_train, X_train, epochs=20,
                         validation_data=(X_valid, X_valid))
```

```
Epoch 1/20
1719/1719 [==============================] - 9s 5ms/step - loss: 0.3366 - rounded_ac
curacy: 0.8884 - val_loss: 0.3130 - val_rounded_accuracy: 0.9083
Epoch 2/20
1719/1719 [==============================] - 13s 8ms/step - loss: 0.3053 - rounded_a
ccuracy: 0.9156 - val_loss: 0.3019 - val_rounded_accuracy: 0.9200
Epoch 3/20
1719/1719 [==============================] - 18s 10ms/step - loss: 0.2980 - rounded_
accuracy: 0.9221 - val_loss: 0.2968 - val_rounded_accuracy: 0.9238
Epoch 4/20
1719/1719 [==============================] - 13s 8ms/step - loss: 0.2940 - rounded_a
ccuracy: 0.9256 - val_loss: 0.2940 - val_rounded_accuracy: 0.9277
Epoch 5/20
1719/1719 [==============================] - 14s 8ms/step - loss: 0.2913 - rounded_a
ccuracy: 0.9282 - val_loss: 0.2915 - val_rounded_accuracy: 0.9284
Epoch 6/20
1719/1719 [==============================] - 9s 5ms/step - loss: 0.2893 - rounded_ac
curacy: 0.9301 - val_loss: 0.2894 - val_rounded_accuracy: 0.9319
Epoch 7/20
1719/1719 [==============================] - 9s 5ms/step - loss: 0.2880 - rounded_ac
curacy: 0.9310 - val_loss: 0.2883 - val_rounded_accuracy: 0.9322
Epoch 8/20
1719/1719 [==============================] - 9s 5ms/step - loss: 0.2867 - rounded_ac
curacy: 0.9322 - val_loss: 0.2976 - val_rounded_accuracy: 0.9266
Epoch 9/20
1719/1719 [==============================] - 8s 4ms/step - loss: 0.2859 - rounded_ac
curacy: 0.9328 - val_loss: 0.2874 - val_rounded_accuracy: 0.9327
Epoch 10/20
1719/1719 [==============================] - 9s 5ms/step - loss: 0.2850 - rounded_ac
curacy: 0.9336 - val_loss: 0.2867 - val_rounded_accuracy: 0.9343
Epoch 11/20
1719/1719 [==============================] - 9s 5ms/step - loss: 0.2844 - rounded_ac
curacy: 0.9342 - val_loss: 0.2851 - val_rounded_accuracy: 0.9356
Epoch 12/20
1719/1719 [==============================] - 8s 5ms/step - loss: 0.2838 - rounded_ac
curacy: 0.9346 - val_loss: 0.2847 - val_rounded_accuracy: 0.9357
Epoch 13/20
1719/1719 [==============================] - 9s 5ms/step - loss: 0.2834 - rounded_ac
curacy: 0.9349 - val_loss: 0.2849 - val_rounded_accuracy: 0.9357
Epoch 14/20
1719/1719 [==============================] - 10s 6ms/step - loss: 0.2829 - rounded_a
ccuracy: 0.9353 - val_loss: 0.2841 - val_rounded_accuracy: 0.9368
Epoch 15/20
1719/1719 [==============================] - 8s 5ms/step - loss: 0.2825 - rounded_ac
curacy: 0.9357 - val_loss: 0.2848 - val_rounded_accuracy: 0.9324
Epoch 16/20
1719/1719 [==============================] - 9s 5ms/step - loss: 0.2821 - rounded_ac
curacy: 0.9360 - val_loss: 0.2840 - val_rounded_accuracy: 0.9359
Epoch 17/20
1719/1719 [==============================] - 9s 5ms/step - loss: 0.2817 - rounded_ac
curacy: 0.9363 - val_loss: 0.2832 - val_rounded_accuracy: 0.9354
Epoch 18/20
1719/1719 [==============================] - 10s 6ms/step - loss: 0.2814 - rounded_a
ccuracy: 0.9365 - val_loss: 0.2827 - val_rounded_accuracy: 0.9360
Epoch 19/20
1719/1719 [==============================] - 9s 5ms/step - loss: 0.2811 - rounded_ac
curacy: 0.9368 - val_loss: 0.2823 - val_rounded_accuracy: 0.9369
Epoch 20/20
1719/1719 [==============================] - 9s 5ms/step - loss: 0.2808 - rounded_ac
curacy: 0.9369 - val_loss: 0.2826 - val_rounded_accuracy: 0.9370
```

In [ ]:
```
# 이미지를 보여주는 기능의 코드를 반복적으로 사용하기 때문에 함수화 하였다.
def plot_image(image):
```

```
    plt.imshow(image, cmap="binary")
    plt.axis("off")
```

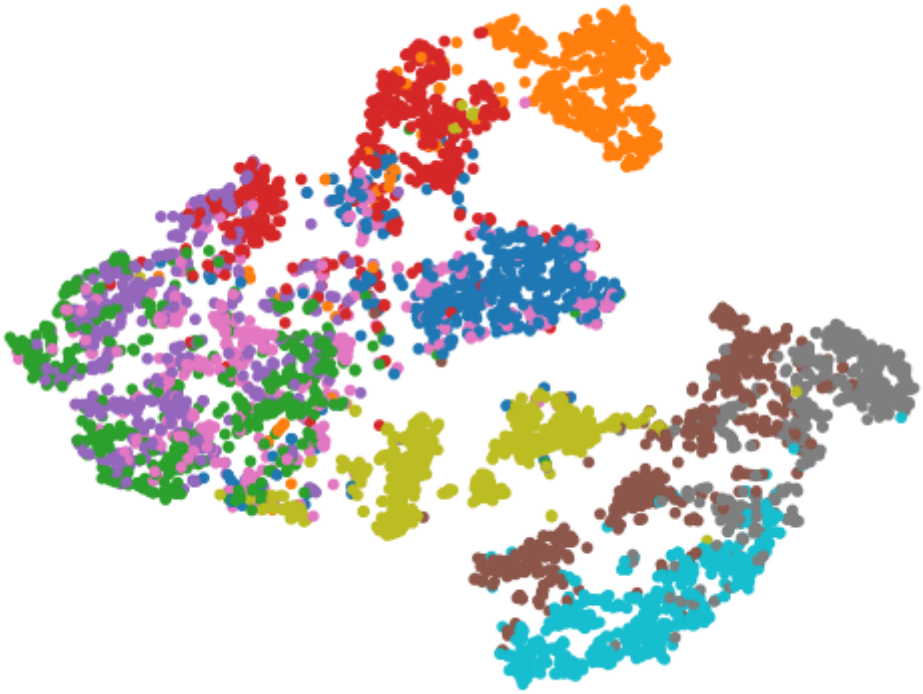
In [ ]:
```
# decoder가 재 생성한 이미지를 비교가 가능하도록 시각화 하는 함수이다.
def show_reconstructions(model, images=X_valid, n_images=5):
    reconstructions = model.predict(images[:n_images])
    fig = plt.figure(figsize=(n_images * 1.5, 3))
    for image_index in range(n_images):
        plt.subplot(2, n_images, 1 + image_index)
        plot_image(images[image_index])
        plt.subplot(2, n_images, 1 + n_images + image_index)
        plot_image(reconstructions[image_index])

show_reconstructions(stacked_ae)
```

```
1/1 [==============================] - 0s 36ms/step
```

In [ ]:
```
np.random.seed(42)

# tsne는 높은 차원의 데이터를 2차원으로 축소시키는 차원 축소 기법이다.
# 현재 우리의 autoencoder 모델의 encoder는 최종적으로 30개의 output을 출력하기 때문0
from sklearn.manifold import TSNE

# validation 데이터를 autoencoder의 encoder모델에 넣고 결과 값을 저장한다.
X_valid_compressed = stacked_encoder.predict(X_valid)
tsne = TSNE()

# 30차원으로 표현된 input 데이터를 TSNE를 통해 2차원으로 축소 한다.
X_valid_2D = tsne.fit_transform(X_valid_compressed)
# 스케일링을 한다.
X_valid_2D = (X_valid_2D - X_valid_2D.min()) / (X_valid_2D.max() - X_valid_2D.min()
```

```
157/157 [==============================] - 0s 1ms/step
```

In [ ]:
```
# autoencoder의 encoder에 의해 생성된 2차원 features를 scatter plot으로 그린다.
plt.scatter(X_valid_2D[:, 0], X_valid_2D[:, 1], c=y_valid, s=10, cmap="tab10")
plt.axis("off")
plt.show()
```



In [ ]:
```
# 위 scatter plot이 표현하고 있는 input 데이터를 시각적으로 확인하기 위한 코드이다.
plt.figure(figsize=(10, 8))
cmap = plt.cm.tab10
plt.scatter(X_valid_2D[:, 0], X_valid_2D[:, 1], c=y_valid, s=10, cmap=cmap)
image_positions = np.array([[1., 1.]])
for index, position in enumerate(X_valid_2D):
    dist = np.sum((position - image_positions) ** 2, axis=1)
    if np.min(dist) > 0.02: # if far enough from other images
        image_positions = np.r_[image_positions, [position]]
        imagebox = mpl.offsetbox.AnnotationBbox(
            mpl.offsetbox.OffsetImage(X_valid[index], cmap="binary"),
            position, bboxprops={"edgecolor": cmap(y_valid[index]), "lw": 2})
        plt.gca().add_artist(imagebox)
plt.axis("off")
plt.show()
```