



베스트 모델 만들기

4팀 - 정유정, 김향임, 박시현, 천승원



목차

01

데이터의 확인과 실행

02

모델 업데이트하기

03

그래프로 확인하기

04

학습의 자동 중단





01

데이터의 확인과 실행



O1 데이터의 확인과 실행

sample() 함수란?

데이터를 랜덤 추출할 때 사용되는 함수

사용법

```
df.sample(n = None, frac = None, replace = False, weights = None,  
random_state = None, axis = None, ignore_index = False)
```

- **n** : 추출할 갯수, **replace**가 FALSE면 **n**의 최댓값은 레이블의 갯수
- **frac** : 추출할 비율, 1보다 작은 값 설정, **n**과 동시 사용X
- **replace** : 중복추출의 허용 여부
- **weight** : 가중치, 합계가 1이 아닐 경우 자동으로 1로 연산
- **random_state** : 랜덤 추출한 값 시드 설정
- **axis** : 추출할 레이블 (0 : index, 1 : columns)
- **ignore_index** : index의 무시 여부 (True면 index 무시하고 숫자 출력)

O1 데이터의 확인과 실행

Q4. sample() 함수 옵션에 대한 설명으로 옳지 않은 것을 고르시오. *

```
1 df.sample(n = None, frac = None, replace = False, weights = None,  
2 | | random_state = None, axis = None, ignore_index = False)
```

- ☐ n: 추출할 갯수
- ☐ frac: 추출할 비율
- ☒ replace: 대체 값 설정
- ☐ weight: 가중치
- ☐ axis: 추출할 레이블

O1 데이터의 확인과 실행





02

모델 업데이트하기





모델 저장 & 재사용

```
from keras.models import load_model  
  
model.save('my_model.h5')  
  
model = load_model('my_model.h5')
```

모델 저장 후 모델 재사용

```
import os  
  
MODEL_DIR = './model/'  
  
if not os.path.exists(MODEL_DIR):  
    os.mkdir(MODEL_DIR)  
  
modelpath = "./model/{epoch:02d}-{val_loss:.4f}.hdf5"
```

모델 저장 시 에포크와 결과 오차 함께 저장





모델 저장 & 재사용

```
import os
```

```
MODEL_DIR = './model/'
```

```
if not os.path.exists(MODEL_DIR) :  
    os.mkdir(MODEL_DIR)
```

```
modelpath = "./model/{epoch:02d}-{val_loss:.4f}.hdf5"
```

os 모듈 불러오기

모델을 저장할 폴더 지정

모델을 저장하는 폴더가 없다면 새로 생성

모델 이름 지정





ModelCheckpoint() 함수

```
from keras.callbacks import ModelCheckpoint
```

모델을 저장할 때 사용하는 함수

```
tf.keras.callbacks.ModelCheckpoint(  
    filepath, monitor='val_loss', verbose=0, save_best_only=False,  
    save_weights_only=False, mode='auto', save_freq='epoch', options=None, **kwargs  
)
```

Filepath : 모델을 저장할 경로

Monitor : 모델을 저장할 기준이 되는 값

Verbose : 모델이 저장되었을 때, 메시지를 표시할지 선택 (0,1)

save_best_only : 모델이 최상의 상태일때만 저장할지 선택 (True, False)





Callbacks 함수

콜백 함수 : 특정 조건에서 자동으로 실행되는 함수

ModelCheckpoint, EarlyStopping, ReduceLROnPlateau 등이 있음

EarlyStopping : 모델을 더 이상 학습을 못할 경우(loss, metric등의 개선이 없을 경우), 학습 도중 미리 학습을 종료시키는 콜백함수

ModelCheckpoint : 모델을 저장할 때 사용되는 콜백함수

ReduceLROnPlateau : 모델의 개선이 없을 경우, Learning Rate를 조절해 모델의 개선을 유도하는 콜백함수





모델 저장

```
Checkpoint = ModelCheckpoint(filepath = modelpath, monitor = 'val_loss', verbose = 1)  
model.fit(x,y, validation_split = 0.2, epochs = 200, batch_size = 200, verbose = 0, callbacks = [checkpointer])
```

데이터셋을 얼마의 비율로 나눠서
트레이닝 데이터셋과 테스트 데이터셋으로
사용할 지 결정



```
Checkpoint = ModelCheckpoint(filepath = modelpath, monitor = 'val_loss', verbose = 1, save_best_only = True)
```





03

그래프로 확인하기





학습 과정 시각화

모델을 업데이트하기 위해서는 에포크를 얼마나 지정할지 결정해야 함.

- 에포크가 많으면 **과적합**의 문제가 발생.
 - **과적합** : 모델이 훈련 세트에 과하게 적합한 상태가 되어 일반성이 떨어지는 상황
- 에포크가 적어도 **과소적합**의 문제가 발생.
 - **과소 적합**: 모델이 훈련 세트의 규칙을 제대로 찾지 못해 모델의 성능이 낮게 나오는 현상

=>모델 학습 시간에 따른 **정확도와 테스트 결과를 그래프로 확인해보자!**





그래프로 확인하기

```
history=model.fit(X,Y,validation_split=0.33, epochs=3500,  
                  batch_size=500,verbose=0)  
  
hist_df = pd.DataFrame(history.history)  
hist_df  
  
# y_vloss에 테스트셋의 오차를 저장합니다.  
y_vloss = hist_df['val_loss']  
  
# y_loss에 학습셋의 오차를 저장합니다.  
y_loss = hist_df['loss']  
  
# x 값을 지정하고 테스트셋의 오차를 빨간색으로, 학습셋의 오차를 파란색으로 표시합니다.  
x_len = np.arange(len(y_loss))  
plt.plot(x_len, y_vloss, "o", c="red", markersize=2, label='Testset_loss')  
plt.plot(x_len, y_loss, "o", c="blue", markersize=2, label='Trainset_loss')  
  
plt.legend(loc='upper right')  
plt.xlabel('epoch')  
plt.ylabel('loss')  
plt.show()
```





Keras의 history란?

```
history=model.fit(X,Y,validation_split=0.33, epochs=3500,  
                  batch_size=130)
```



```
Epoch 61/3500  
6/6 [=====] - 0s 9ms/step - loss: 0.1888 - accuracy: 0.9326 - val_loss: 0.1720 - val_accuracy: 0.9503  
Epoch 62/3500  
6/6 [=====] - 0s 9ms/step - loss: 0.1774 - accuracy: 0.9464 - val_loss: 0.1821 - val_accuracy: 0.9472  
Epoch 63/3500  
6/6 [=====] - 0s 13ms/step - loss: 0.1752 - accuracy: 0.9403 - val_loss: 0.1702 - val_accuracy: 0.9472  
Epoch 64/3500  
6/6 [=====] - 0s 9ms/step - loss: 0.1740 - accuracy: 0.9403 - val_loss: 0.1713 - val_accuracy: 0.9472  
Epoch 65/3500
```

모델 학습을 위해 사용한 fit 함수.

epoch마다의 학습 이력을 history 객체에 저장함.





Keras의 history란?

```
import matplotlib.pyplot as plt

y_vloss=history.history['val_loss']
y_acc=history.history['accuracy']
```

[출력 예시]

```
[0.7519985437393188,
0.5888906121253967,
0.4848150908946991,
0.41306009888648987,
0.40018966794013977,
0.4547359049320221,
0.4725721478462219,
0.4430282413959503,
0.38579079508781433,
0.34649938344955444,
0.332129567861557,
0.3291972577571869,
0.32635411620140076,
0.3212141990661621,
0.31415385007858276,
0.3074224889278412,
0.303087055683136,
```

epoch 마다 오차와 정확도에 대한 정보들이 저장
History 함수를 이용해서 **학습 이력의 정보를**
리턴할 수 있음.

[학습 이력 정보]

- **loss** : 학습셋의 오차
- **accuracy** : 학습셋의 정확도
- **val_loss** : 검증셋의 오차
- **val_accuracy** : 검증셋의 정확도

=> **history.history**는 **dictionary**
type으로 출력됨.






Keras의 history란?

```
#데이터를 파악하기 쉽도록 DataFrame으로  
histor_DF=pd.DataFrame(history.history)  
histor_DF
```

[출력 예시]



	loss	accuracy	val_loss	val_accuracy
0	1.094046	0.738132	0.751999	0.773292
1	0.876708	0.741194	0.588891	0.776398
2	0.687868	0.754977	0.484815	0.782609
3	0.552582	0.767228	0.413060	0.788820
4	0.468500	0.774885	0.400190	0.798137
...
3495	0.002753	1.000000	0.351473	0.984472
3496	0.002699	1.000000	0.352755	0.984472
3497	0.002703	1.000000	0.354217	0.984472
3498	0.002596	1.000000	0.353983	0.984472
3499	0.002541	1.000000	0.353701	0.984472

3500 rows x 4 columns

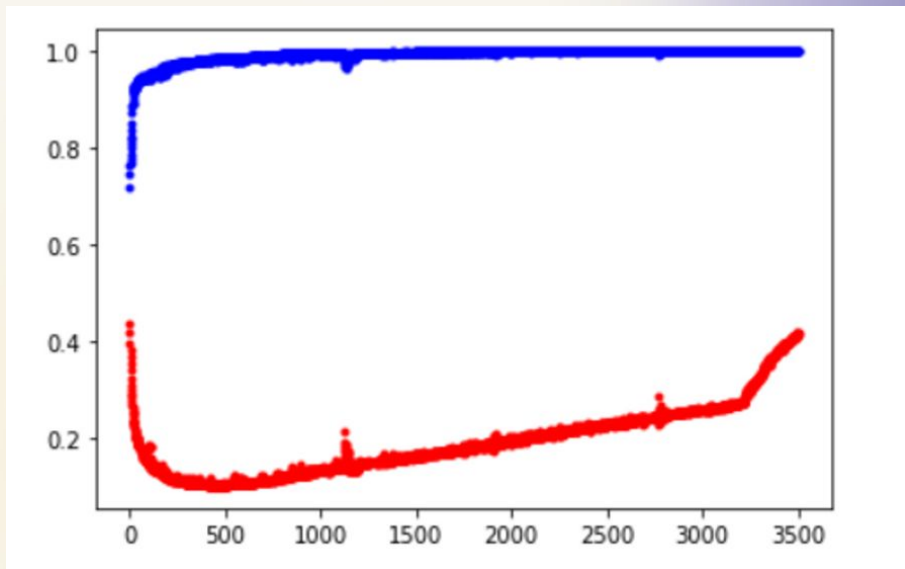
History 함수의 반환값은 dictionary type이므로 데이터를 더 파악하기 쉽게 하려면 DataFrame으로 변경.

적합한 epoch를 알려면 한 눈에 보기 위해서는 데이터를 시각화해야 함.





그래프로 확인하기



테스트셋의 정확도는 점점 100%에 수렴하지만,

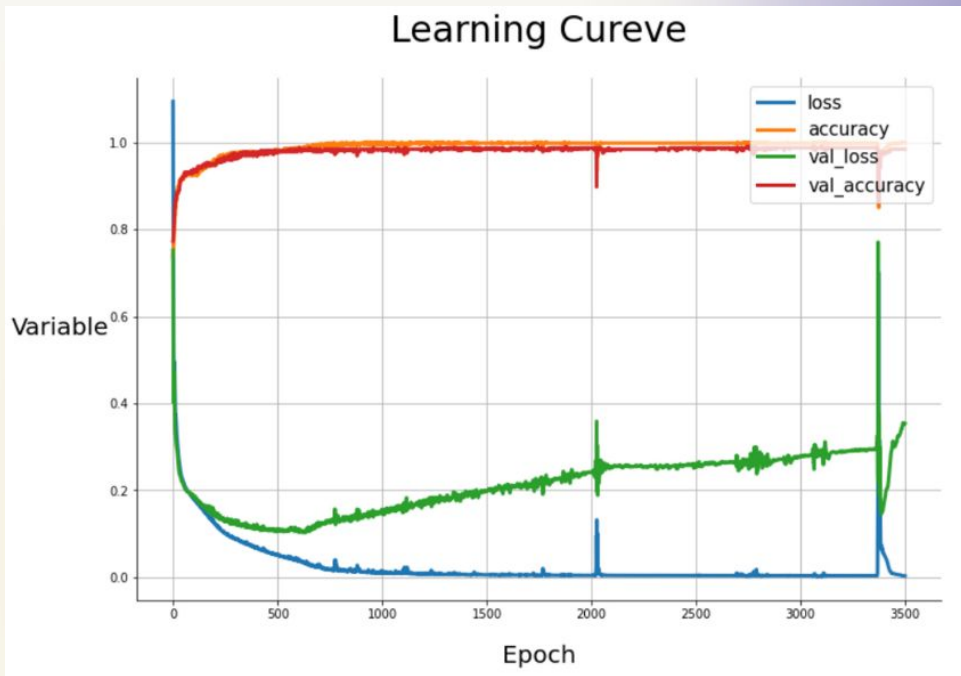
검증셋의 오차는 epoch가 500 정도일 때를 기준으로 증가하는 것을 알 수 있음.

=> 테스트셋에 과도하게 적응한 과적합 문제가 발생.





epoch가 3500번일 때



테스트셋의 정확도는 점점 100%에 수렴하지만,

검증셋의 오차는 epoch가 600 정도일 때를 기준으로 증가하는 것을 알 수 있음.

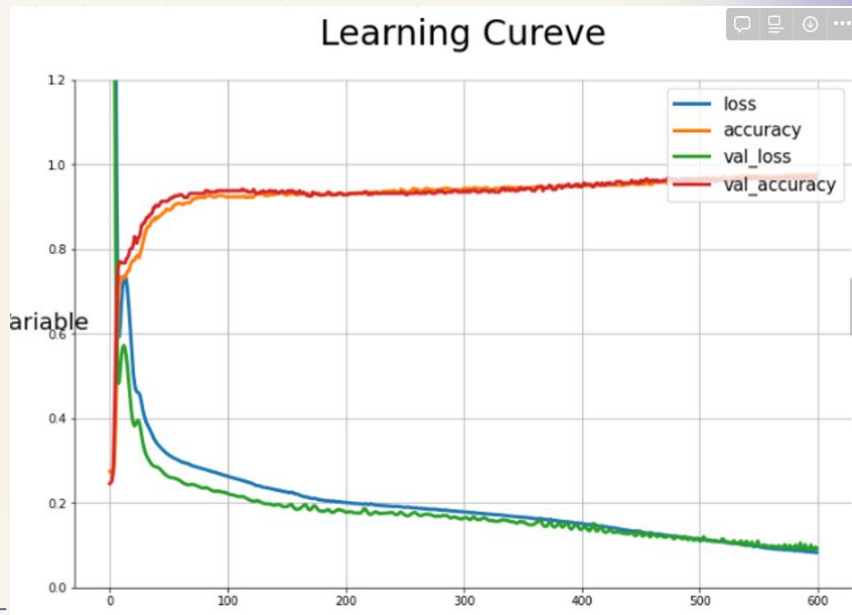
=> 테스트셋에 과도하게 적응한 과적합 문제가 발생. epoch가 600일 때의 그래프는?





epoch가 600번일 때

<https://colab.research.google.com/drive/1OCxgKLsyfXg48JNXh4M7yIRGGpeB8Obf?usp=sharing>



**테스트셋과 검증셋가 비슷한 경향을 보이고
있으므로 과적합이 일어나고 있지 않음.**

=>그래프를 통해 적절한 epoch를 찾아냄





Tensorboard란?

- 텐서보드는 **머신러닝 실험**에 필요한 **시각화** 및 도구를 제공한다.
- 학습이 모두 종료된 이후 시각화를 할 수 있는 **matplotlib**과는 달리, 훈련하는 도중에도 **실시간 시각화**가 가능하다.
- **텐서플로우** 뿐만 아니라 **Pytorch**에서도 활용 가능하다
- 주요 기능
 - **loss** 및 **accuracy**와 같은 **측정 항목 추적** 및 **시각화**
 - 시간의 경과에 따라 달라지는 **weight, bias** 등의 **히스토그램 확인**
 - **Tensorflow** 프로그램 **프로파일링** 등





Tensorboard를 활용하여 학습 과정 시각화 하기

<https://colab.research.google.com/drive/1Yx2AYoJf1WQEeB6u4I-ZNnSFYbuvWZSa?usp=sharing>





04

학습의 자동 중단





Preclass quiz 정답.

Q6. 다음 중 과적합을 방지하기 위해 사용할 수 있는 방법으로 알맞지 않은것은?



객관식 질문

☐ EarlyStopping



☐ Weight Decay



☐ Weight Constraint



☐ Dropout

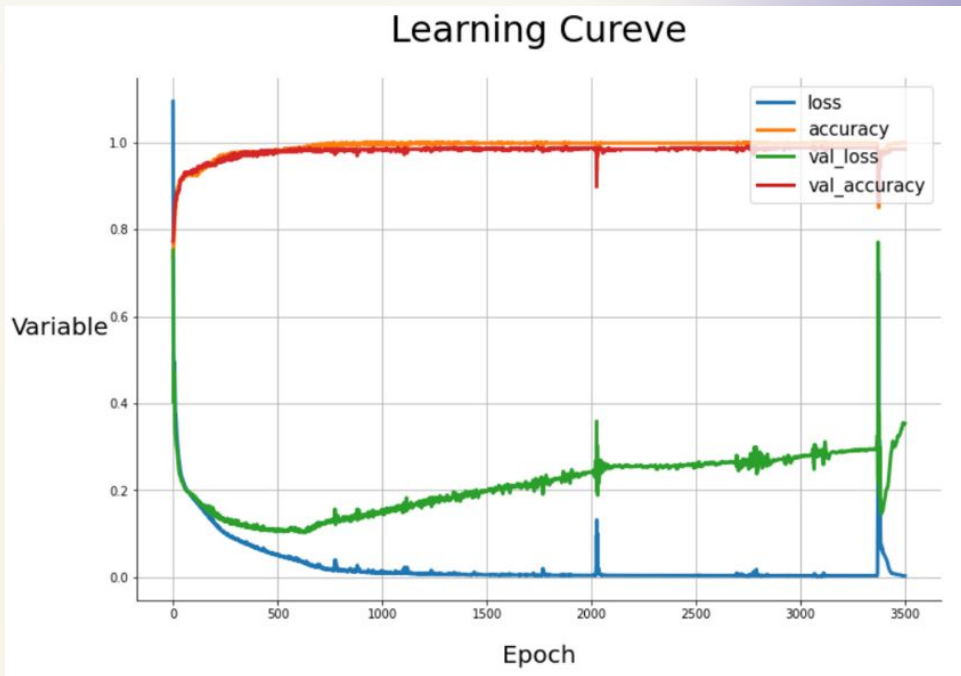


☒ Drop Constraint





EarlyStopping



학습이 진행될 수록 테스트셋의 정확도는 점점 100%에 수렴하지만,

검증셋의 오차는 과적합 문제가 발생하여 실험결과가 점점 나빠지게 된다는 점을 역이용.

=> 학습이 진행되어도 테스트셋의 오차가 줄지 않으면 학습을 멈추게 함.

= " EarlyStopping "





EarlyStopping

```
[4] # 먼저 EarlyStopping을 위해 그 함수를 import함.  
from keras.callbacks import EarlyStopping  
  
# EarlyStopping 함수의 주요 파라미터  
# monitor = 모니터할 값 => val_loss  
# patience = 상태가 좋아지지 않아도 몇번 기다릴건지 => 100번  
# min_delta = 개선으로 간주되는 최소 변경 크기. 이 값만큼 개선이 없으면 EarlyStopping대상이 됨. => 0.001  
# restore_best_weights = Early Stopping시 이전에 찾은 최적의 가중치값으로 복원함 => True  
  
# EarlyStopping 함수를 통해 종료할 조건을 설정함.  
early_stopping_callback = EarlyStopping(monitor='val_loss', patience = 100, restore_best_weights=True, min_delta=0.001)  
  
# EarlyStopping을 통해 설정한 조건에 도달할 시 종료하도록 함.  
model.fit(x,y, validation_split = 0.33, epochs = 2000, batch_size = 500, callbacks=[early_stopping_callback])  
  
25/25 [=====] - 0s 8ms/step - loss: 0.0389 - accuracy: 0.9874 - val_loss: 0.0421 - val_accuracy: 0.9860  
Epoch 393/2000  
25/25 [=====] - 0s 6ms/step - loss: 0.0414 - accuracy: 0.9865 - val_loss: 0.0405 - val_accuracy: 0.9867  
Epoch 394/2000  
25/25 [=====] - 0s 8ms/step - loss: 0.0375 - accuracy: 0.9874 - val_loss: 0.0378 - val_accuracy: 0.9863  
Epoch 395/2000
```





Weight Decay(L2 regularization)

$$\begin{aligned} w &\leftarrow w - \eta \left(\frac{\partial \text{DataLoss}}{\partial w} + \lambda w \right) \\ &= w(1 - \eta\lambda) - \eta \frac{\partial \text{DataLoss}}{\partial w} \end{aligned}$$

모델에 **weight**의 제곱합을 패널티 텀으로 주어(=제약을 걸어) **loss**를 최소화함.

=> 가중치를 감소시키는 방법.

= " **Weight Decay** "

L1, L2 norm의 개념을 적용하여 **L2 regularization** 이라고도 함. (시간상 설명이 힘들어 생략하겠습니다.)





Weight Decay의 원리

$$\begin{aligned} w &\leftarrow w - \eta \left(\frac{\partial DataLoss}{\partial w} + \lambda w \right) \\ &= w(1 - \eta\lambda) - \eta \frac{\partial DataLoss}{\partial w} \end{aligned}$$

원래의 **data loss** 값에 제곱합을 더하여 **loss**값을 조정한다. (첫번째 식)

이를 미분하고 (두번째 식) 그 값을 기존의 가중치 업데이트 식 (세번째 식) 에서의 오차의 미분값에 대입하면 다음과 같은 식이 나오게 된다. (네번째 식)

미분을 했을 때, 기본 **dataloss**에 **w**의 **lambda**배 만큼 더하게 되므로 가중치 값이 그만큼 보정된다.





Weight Decay의 원리

$$\begin{aligned} w &\leftarrow w - \eta \left(\frac{\partial DataLoss}{\partial w} + \lambda w \right) \\ &= w(1 - \eta\lambda) - \eta \frac{\partial DataLoss}{\partial w} \end{aligned}$$

결과적으로 원래의 업데이트 식에서의 w 가 $w(1 - \eta\lambda)$ 가 되기 때문에 **weight**가 아주 작은 **factor**에 비례해 감소하게 되며, 그래서 **Weight Decay**라는 이름이 붙게 됨.

=> 웨이트의 값이 증가하는 것을 막아 특정 가중치가 비정상적으로 커지는 것을 막을 수 있고, 이로 인해 과적합이 발생하는 것도 막을 수 있게 된다.





Weight Decay

```
# regularizers import
from keras import regularizers

model = Sequential()
model.add(Dense(30, input_dim = 10, activation = 'relu', kernel_regularizer=regularizers.l2(0.01), activity_regularizer=regularizers.l1(0.01))) # L2, L1 regularization
# regularizers.l2(0.001) : 가중치 행렬의 모든 원소를 제공하고 0.001을 곱하여 네트워크의 전체 손실에 더해진다는 의미, 이 규제(패널티)는 훈련할 때만 추가됨
model.add(Dense(12, activation='relu', kernel_regularizer=regularizers.l2(0.01), activity_regularizer=regularizers.l1(0.01)))
model.add(Dense(8, activation = 'relu', kernel_regularizer=regularizers.l2(0.01), activity_regularizer=regularizers.l1(0.01)))
model.add(Dense(1, activation = 'sigmoid', kernel_regularizer=regularizers.l2(0.01), activity_regularizer=regularizers.l1(0.01)))

model.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics = ['accuracy'])

model.fit(x,y, epochs = 100, batch_size = 100)

print("\n Accuracy : %.4f" %(model.evaluate(x,y)[1]))
```

182/182 [=====] - 1s 8ms/step - loss: 0.7152 - accuracy: 0.5491
Epoch 74/100
182/182 [=====] - 1s 5ms/step - loss: 0.7133 - accuracy: 0.5491
Epoch 75/100
182/182 [=====] - 1s 4ms/step - loss: 0.7115 - accuracy: 0.5491
Epoch 76/100





Weight Constraint

```
[6] # constraints import
    from tensorflow.keras.constraints import MaxNorm

model = Sequential()
model.add(Dense(30, input_dim = 10, activation = 'relu',
                kernel_regularizer=regularizers.l2(0.01), activity_regularizer=regularizers.l1(0.01),
                kernel_constraint=MaxNorm(2.))) # L2, L1 regularization
# regularizers.l2(0.001) : 가중치 행렬의 모든 원소를 제곱하고 0.001을 곱하여 네트워크의 전체 손실에 더해진다는 의미, 이 규제(패널티)는 훈련할 때만 추가됨
model.add(Dense(12, activation='relu',
                kernel_regularizer=regularizers.l2(0.01), activity_regularizer=regularizers.l1(0.01),
                kernel_constraint=MaxNorm(2.)))
model.add(Dense(8, activation = 'relu',
                kernel_regularizer=regularizers.l2(0.01), activity_regularizer=regularizers.l1(0.01),
                kernel_constraint=MaxNorm(2.)))
model.add(Dense(1, activation = 'sigmoid'))

model.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics = ['accuracy'])

model.fit(x,y, epochs = 100, batch_size = 100)

print("\n Accuracy : %.4f" % (model.evaluate(x,y)[1]))
```

Weight Decay는 가중치의 학습에 영향을 주는 방법이라면, **Constraint**는 직접적으로 가중치의 크기를 규제하는 방법.

특정한 기준값을 초과하는 가중치의 값을 그 기준에 미치지 못하는 값으로 함수등을 통해 변경하는 방법으로 확실하게 가중치를 원하는 값 이하로 떨어뜨릴 수 있다는 장점이 있음.

=> " **Weight Constraint** "





Weight Constraint

```
[6] # constraints import
    from tensorflow.keras.constraints import MaxNorm

    model = Sequential()
    model.add(Dense(30, input_dim = 10, activation = 'relu',
                    kernel_regularizer=regularizers.l2(0.01), activity_regularizer=regularizers.l1(0.01),
                    kernel_constraint=MaxNorm(2.))) # L2, L1 regularization
    # regularizers.l2(0.001) : 가중치 행렬의 모든 원소를 제공하고 0.001을 곱하여 네트워크의 전체 손실에 더해진다는 의미, 이 규제(패널티)는 훈련할 때만 추가됨
    model.add(Dense(12, activation='relu',
                    kernel_regularizer=regularizers.l2(0.01), activity_regularizer=regularizers.l1(0.01),
                    kernel_constraint=MaxNorm(2.)))
    model.add(Dense(8 , activation = 'relu',
                    kernel_regularizer=regularizers.l2(0.01), activity_regularizer=regularizers.l1(0.01),
                    kernel_constraint=MaxNorm(2.)))
    model.add(Dense(1 , activation = 'sigmoid'))

    model.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics = ['accuracy'])

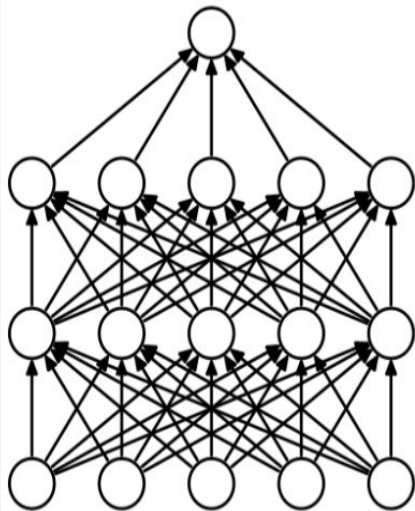
    model.fit(x,y, epochs = 100, batch_size = 100)

    print("\n Accuracy : %.4f" %(model.evaluate(x,y)[1]))
```

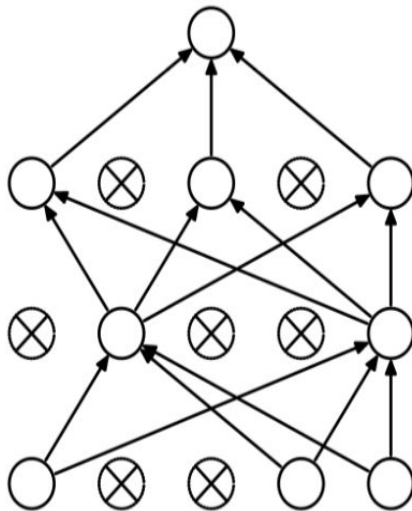




Dropout



(a) Standard Neural Net



(b) After applying dropout.

학습과정에서 신경망 일부를 사용하지 않는 방법. ex) 드롭아웃의 비율을 0.5로 한다면 학습과정마다 랜덤으로 절반의 뉴런을 사용하지 않고, 나머지 절반만을 사용함.

=> 학습시에 인공신경망이 특정 뉴런, 조합에 너무 의존하는 것을 방지, 매번 랜덤선택으로 서로 다른 신경망을 사용하는 효과를 내어 과적합을 방지함.

= " Dropout "





Dropout



```
from tensorflow.keras.layers import Dropout

model = Sequential()
model.add(Dense(30, input_dim = 10, activation = 'relu',
                kernel_regularizer=regularizers.l2(0.01), activity_regularizer=regularizers.l1(0.01),
                kernel_constraint=MaxNorm(2.)))
model.add(Dropout(0.5)) # 드롭아웃 추가. 비율은 50%
model.add(Dense(12, activation='relu',
                kernel_regularizer=regularizers.l2(0.01), activity_regularizer=regularizers.l1(0.01),
                kernel_constraint=MaxNorm(2.)))
model.add(Dropout(0.5)) # 드롭아웃 추가. 비율은 50%
model.add(Dense(8 , activation = 'relu',
                kernel_regularizer=regularizers.l2(0.01), activity_regularizer=regularizers.l1(0.01),
                kernel_constraint=MaxNorm(2.)))
model.add(Dropout(0.5)) # 드롭아웃 추가. 비율은 50%
model.add(Dense(1 , activation = 'sigmoid'))

model.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics = ['accuracy'])
```





실습 문제 설명

<https://colab.research.google.com/drive/1A4ThX0afdJqS3jQe9Ear18pBaKXQc1A7?usp=sharing>





Thanks!

Do you have any questions?

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon** and infographics & images by **Freepik**

