

데이터의 확인과 실행 & 과적합

AI STUDY

과적합

딥러닝 모델 설계에 있어서 가장 중요한 것이 학습시킬 데이터의 개수와 품질이다.

좋은 데이터는 한정되어 있기 때문에 같은 데이터들로만 학습을 시키는 경우가 생길 수 있다. 이 경우 과적합이 발생할 수 있다.

과적합 예시

- 수학 문제집 외워 풀기
- 모델에게만 바지 사이즈 측정

딥러닝을 배우며

지금까지 딥러닝을 공부하면서 하나의 데이터셋을 이용해서 학습을 시키고 또 그 데이터를 이용해서 테스트를 진행했었다.

따라서 다른 데이터셋에서는 정확도가 매우 떨어질 수 있다.

광석과 일반 돌 구분

35년 전에도 일어났던 과적합 문제는 지금도 완벽하게 해결하지 못하는 문제로 남아있다.

광석과 일반 돌 구분

Range Index: 208 entries, 0 to 207			
Data columns (total 61 columns):			
0	208	non-null	float64
1	208	non-null	float64
2	208	non-null	float64
3	208	non-null	float64
4	208	non-null	float64
5	208	non-null	float64
...
58	208	non-null	float64
59	208	non-null	float64
60	208	non-null	object
Dtypes: float64(60), object(1)			
memory usage: 99.2+ KB			

광석과 일반 돌 구분

시드를 설정하고, 데이터를 가져와서 데이터 형 변환이 필요한 곳은 데이터 인코딩한다. 그 다음 딥러닝 모델을 설정하고 컴파일하고 실행하는 것으로 데이터를 불러와 실행할 수 있다.

이 결과로 우리는 정확도가 100%인 딥러닝 모델을 얻었다.

실제로 100%의 정확도일까?

학습 시킨 데이터를 다시 넣어서 실행을 했기 때문에 다른 데이터를 입력시키면 그 정확도가 떨어지게 되는 것이다.

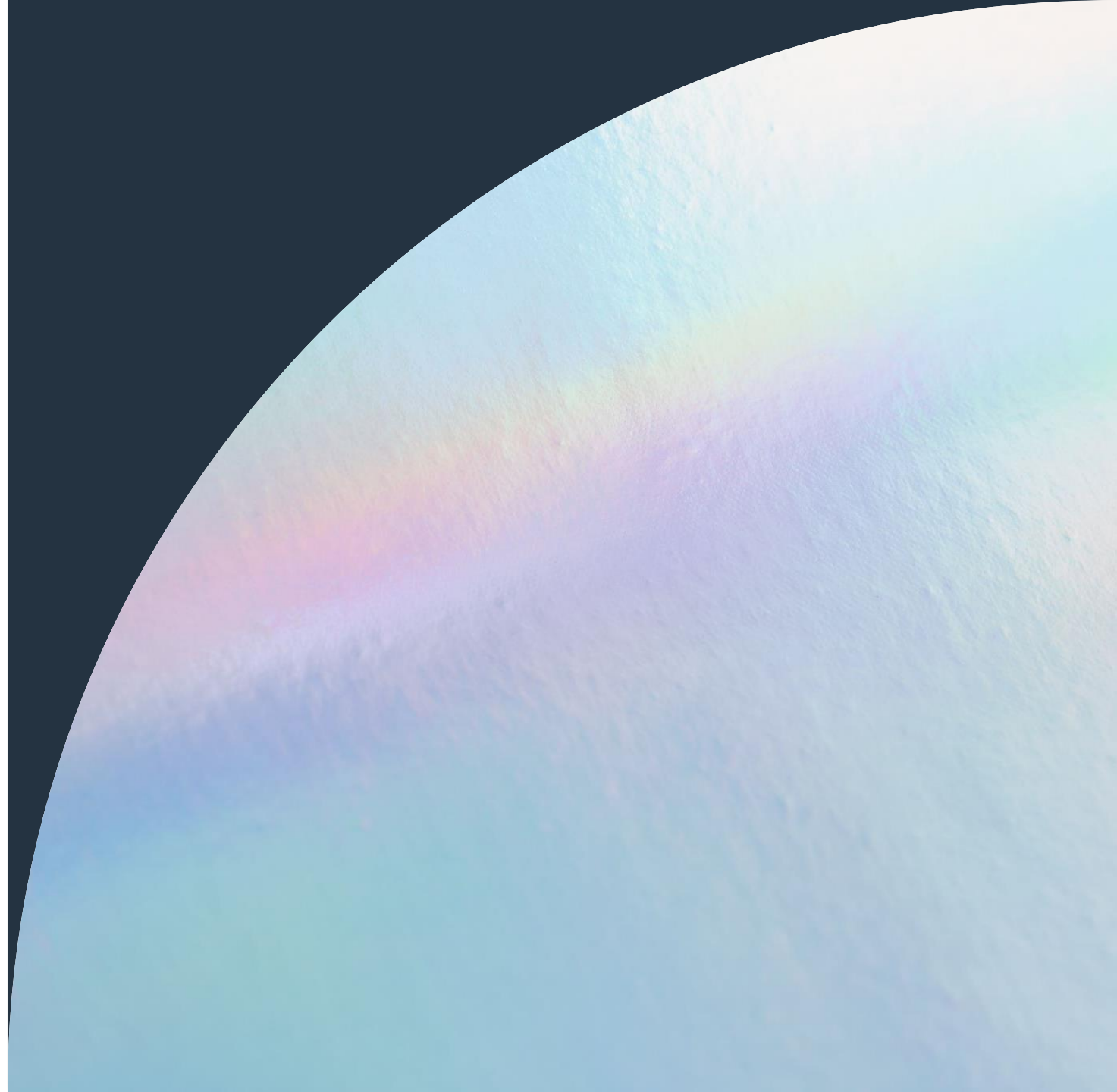
해결 방안

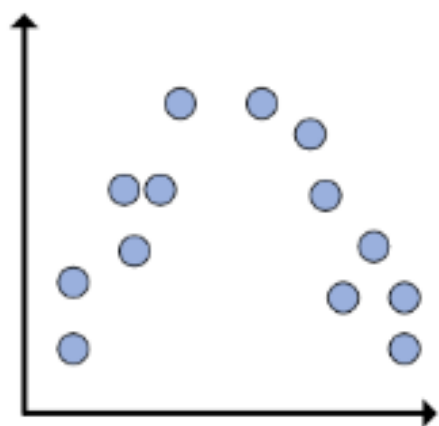
초기 데이터에서 학습셋과 데이터셋을 구분하는 방법이 있다.

감사합니다

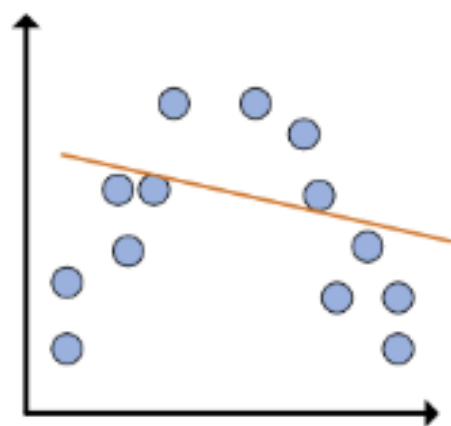
02. 과적합 이해하기

강민서

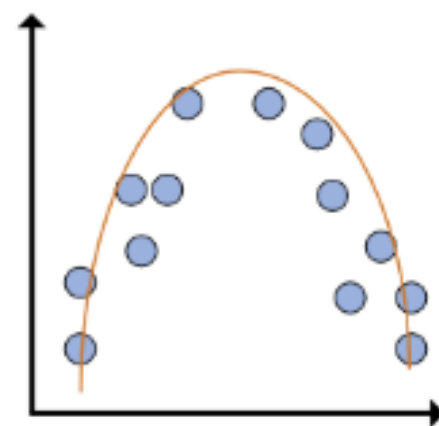




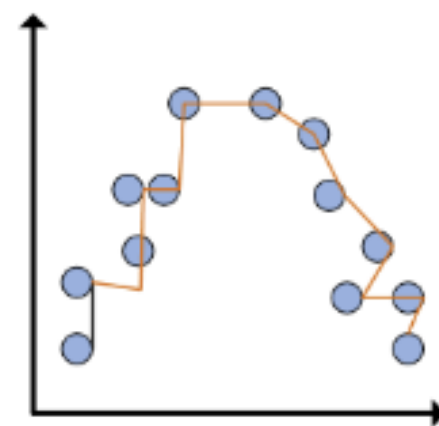
A



B



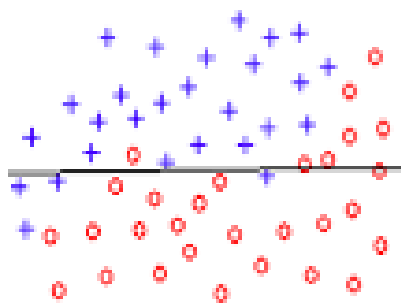
C



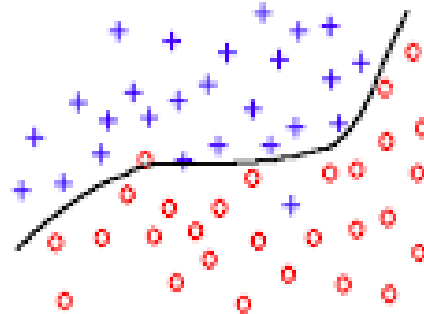
D

과적합(overfitting)이란?

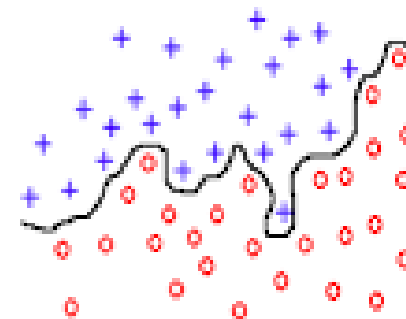
- 모델이 학습 데이터 셋 안에서는 일정 수준 이상의 예측 정확도를 보이지만, 새로운 데이터에 적용하면 잘 맞지 않는 것



underfitting



good fit



overfitting

과적합 발생원인

- 모델의 파라미터 수가 많거나 학습용 데이터 세트의 양이 부족한 경우
- 다항회귀의 차수를 높일수록 발생

과소적합(underfitting)

- 모델이 너무 단순해 데이터의 내재된 구조를 학습하지 못하는 것

과적합을 방지하는 방법들

- 데이터의 양 늘리기
- 모델의 복잡도 줄이기
- 가중치 규제(Regularization) 적용하기
- 드롭아웃

1. 데이터의 양 늘리기

- 데이터의 양이 적을 경우, 데이터의 특정 패턴이나 노이즈까지 암기하므로
- 데이터 양을 늘릴수록 데이터의 일반적인 패턴 학습을 통해 과적합 방지
- 데이터 증식/ 증강(Data Augmentation)
- 텍스트 데이터의 경우에는 역번역(Back Translation)

2. 모델의 복잡도 줄이기

- 인공 신경망의 복잡도는 은닉의 수나 매개변수의 수로 결정
- 모델의 수용력(capacity)

3. 가중치 규제(Regulation) 적용하기

- Lasso(least absolute shrinkage and selection operator) : L1규제
- Ridge : L2규제

$$\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 + \alpha \sum_{j=1}^m |w_j|$$

$$MSE + \alpha \cdot L_1\text{-norm}$$

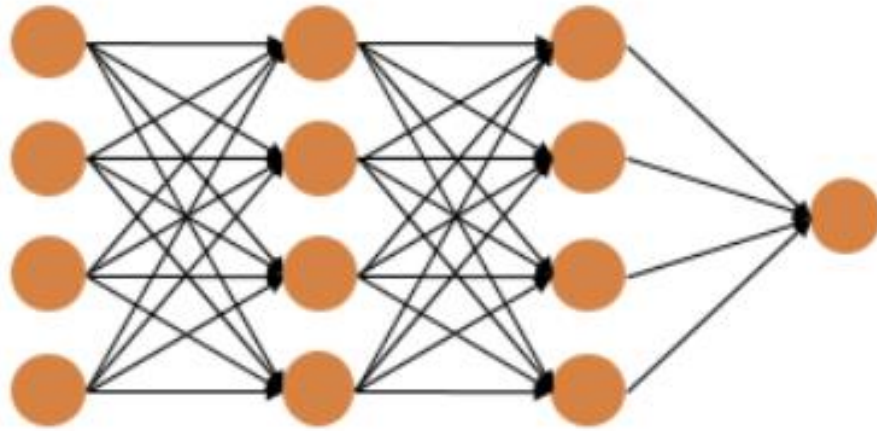
MSE + penalty

$$\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^m w_j^2$$

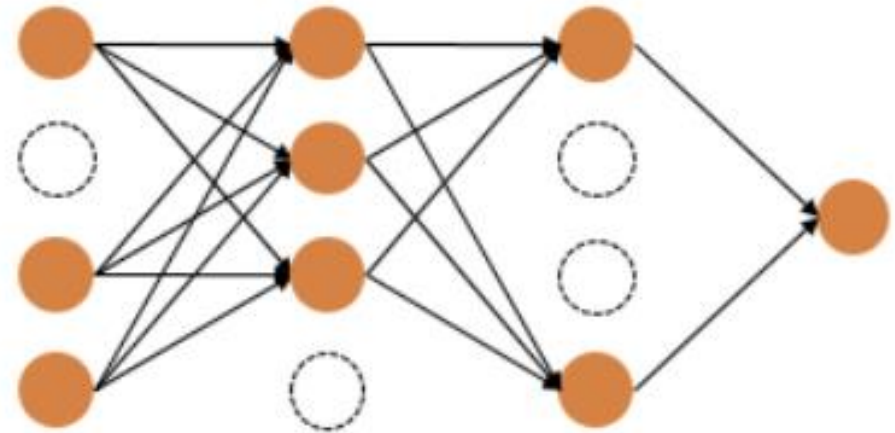
$$MSE + \lambda \sum_{j=1}^m w_j^2$$

Lasso (L1)	Ridge (L2)
가중치를 0으로, 특성 무력화	가중치를 0에 가깝게, 특성들의 영향력 감소
일부 특성이 중요하다면	특성의 중요도가 전체적으로 비슷하다면
sparse model	Non-sparse model
feature selection	

4. 드롭아웃(Dropout)




드롭아웃 적용 전

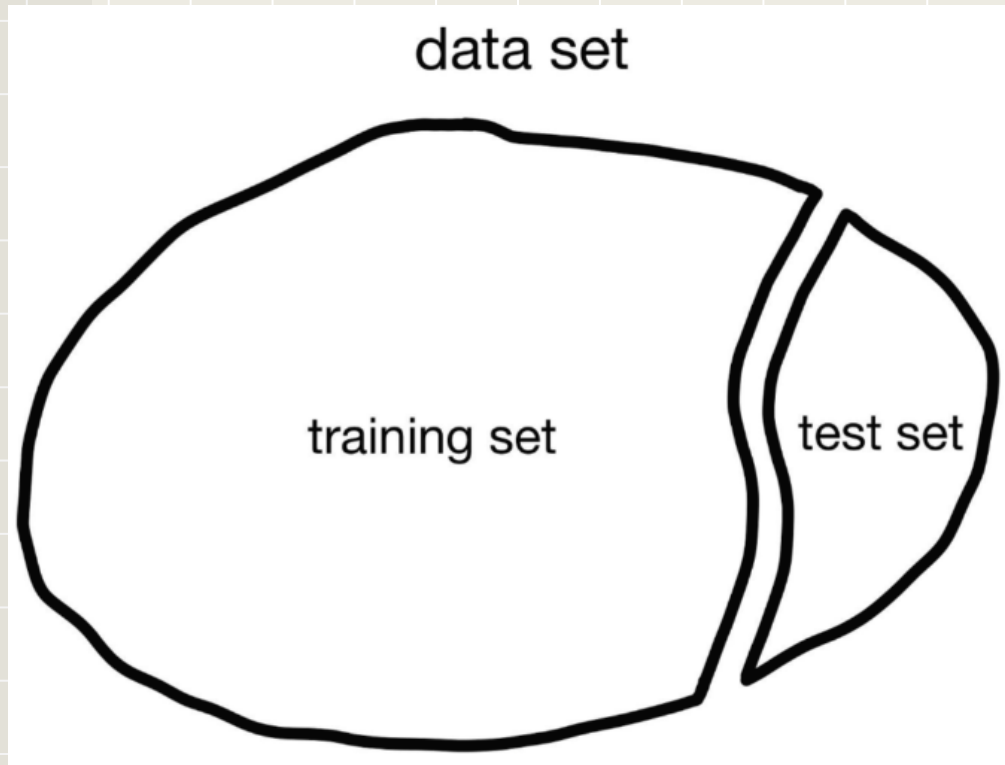


드롭아웃 적용 후

감사합니다



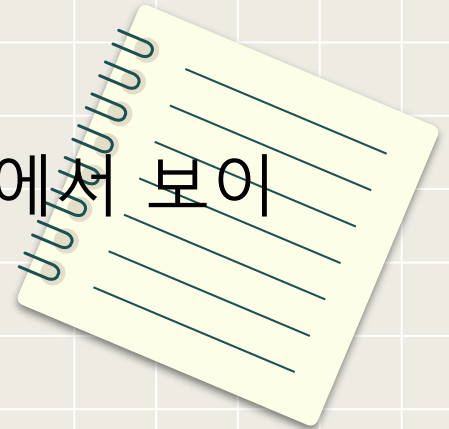
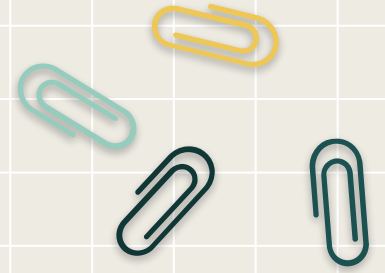
학습셋과 테스트셋



훈련 세트(Training Set) : 모델이 학습할 데이터

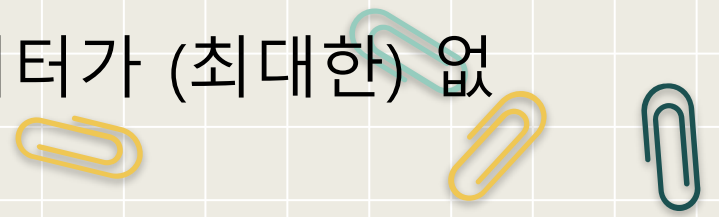
테스트 세트(Test Set) : 모델의 성능을 테스트하기 위해 사용할 데이터

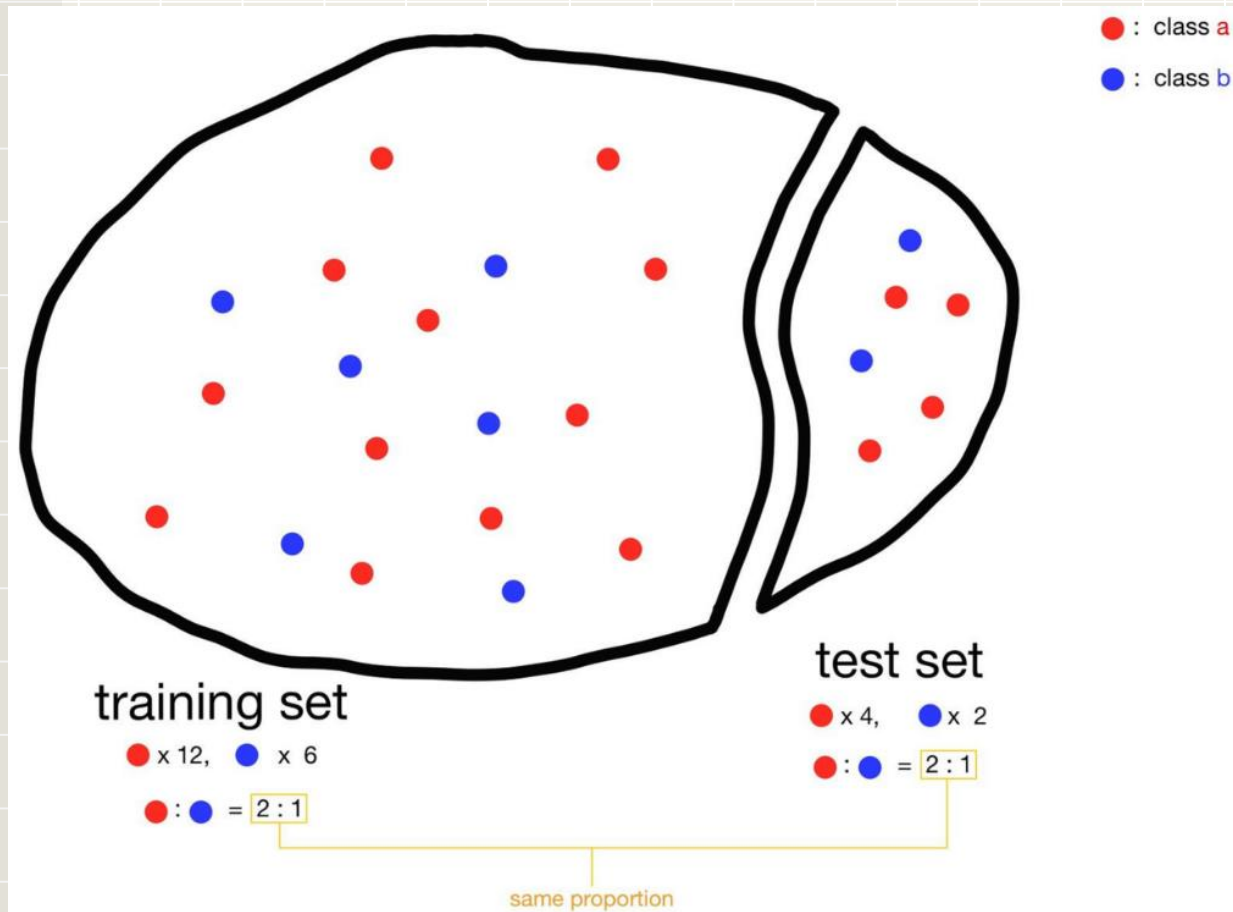
일반화 성능(Generalization Performance) : 모델이 실전에서 보이는 성능



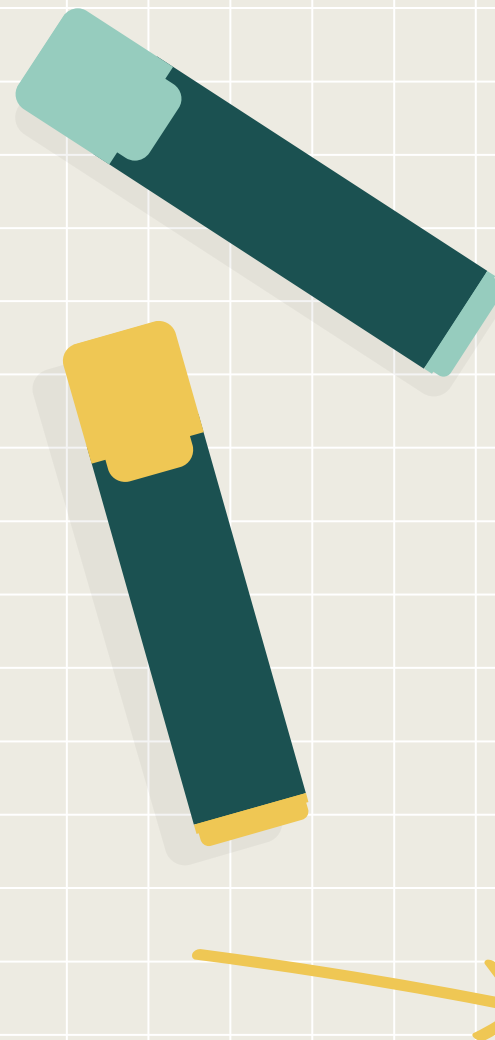


데이터 세트를 훈련 세트와 테스트 세트로 나눌 때의 규칙

1. 훈련 세트의 데이터가 테스트 세트의 데이터보다 많아야 함
 2. 훈련 세트와 테스트 세트가 동일한 비율의 데이터(클래스) 분포를 가지고 있어야 함
 3. 훈련 세트와 테스트 세트에 중복되는 데이터가 (최대한) 없어야 함
- 



모델이 데이터의 규칙을 제대로 찾아내지 못하고, 테스트를 함에 있어서 모델의 정확한 성능을 추정하지 못할 수 있음!!




```
# one-hot-encoding
e = LabelEncoder()
e.fit(Y_obj)
Y = e.transform(Y_obj)

# 학습 셋과 테스트 셋의 구분
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, random_state=seed)

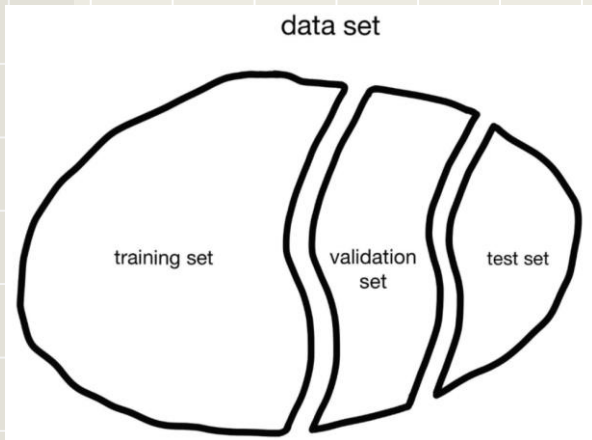
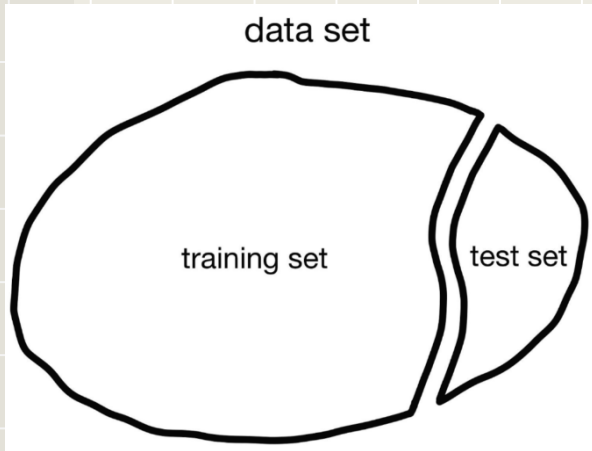
# 딥러닝 구조를 결정(모델을 설정하고 실행)
model = Sequential()
model.add(Dense(24, input_dim=60, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# 딥러닝 실행
model.compile(loss='mean_squared_error',
              optimizer='adam',
              metrics=['accuracy'])
model.fit(X_train, Y_train, epochs=100, batch_size=5)

# 테스트셋에 모델 적용
print("\n Test Accuracy: %.4f" % (model.evaluate(X_test, Y_test)[1]))
```

```
Epoch 95/100
29/29 [=====] - 0s 2ms/step - loss: 0.0209 - accuracy: 0.9931
Epoch 96/100
29/29 [=====] - 0s 2ms/step - loss: 0.0185 - accuracy: 0.9931
Epoch 97/100
29/29 [=====] - 0s 2ms/step - loss: 0.0174 - accuracy: 1.0000
Epoch 98/100
29/29 [=====] - 0s 2ms/step - loss: 0.0182 - accuracy: 0.9931
Epoch 99/100
29/29 [=====] - 0s 2ms/step - loss: 0.0159 - accuracy: 1.0000
Epoch 100/100
29/29 [=====] - 0s 2ms/step - loss: 0.0154 - accuracy: 1.0000
2/2 [=====] - 0s 8ms/step - loss: 0.1371 - accuracy: 0.8413

Test Accuracy: 0.8413
```



모델은 과연 실전에서도 좋은 성능을 보여줄 수 있는가?

테스트 세트에 대해서는 좋은 성능을 보
이겠지만, 실전에서도 동일한 성능을 보
이기는 힘들

훈련 데이터를 다시 **훈련시킬 데이터
세트와 성능을 평가할 데이터 세트(검
증 세트)**로 나눠주면 됨!!

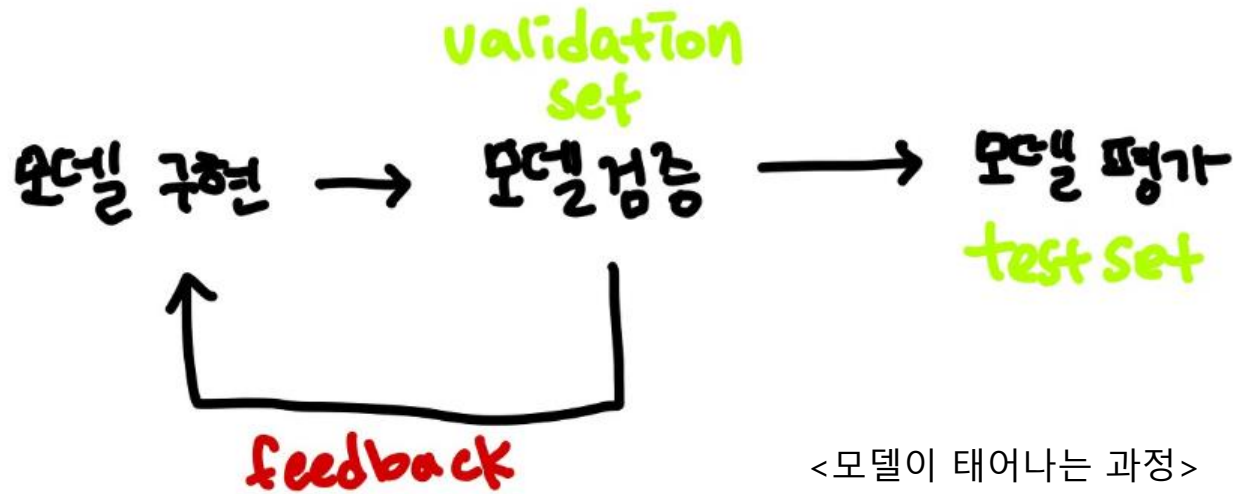




보통 검증세트를 분할할 때에는 전체 데이터 세트를 8 : 2로 나누어 훈련 세트와 테스트 세트를 만들고, 다시 훈련세트를 8 : 2로 나누어 훈련세트와 검증세트를 만든다



그냥 Test Set으로 성능 평가하면 되는 거 아니가요?



모델검증과 모델평가는 분리해서 생각해야 함

- 모델 검증은 모델의 성능을 평가하고, 그 결과를 토대로 모델을 튜닝하는 작업을 진행함

- 반면, 모델 평가는 최종적으로 '이 모델이 실제 데이터에서 얼마나 성능을 낼 것이다!'라는 것을 확인하는 단계

if) 모델이 검증에 쓰였던 데이터를 모델 평가한다면 당연히 높은 점수를 낼 것!!

-> 이런 현상을 'Test Set의 정보가 모델에 새어 나갔다' or '모델의 일반화 성능이 왜곡되었다'

```
# 학습 셋과 테스트 셋의 구분
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, random_state=seed)

# 학습 셋과 검증 셋의 구분
sub_input, val_input, sub_target, val_target = train_test_split(X_train, Y_train, test_size=0.3, random_state=seed)

# 딥러닝 구조를 결정(모델을 설정하고 실행)
model = Sequential()
model.add(Dense(24, input_dim=60, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# 딥러닝 실행
model.compile(loss='mean_squared_error',
              optimizer='adam',
              metrics=['accuracy'])

model.fit(X_train, Y_train, epochs=100, batch_size=5)
model.fit(sub_input, sub_target, epochs=100, batch_size=5)

# 테스트셋에 모델 적용
print("\n Test Accuracy: %.4f" % (model.evaluate(X_test, Y_test)[1]))
print("\n Test Accuracy: %.4f" % (model.evaluate(val_input, val_target)[1]))
```

```
Epoch 95/100
21/21 [=====] - 0s 2ms/step - loss: 5.1206e-04 - accuracy: 1.0000
Epoch 96/100
21/21 [=====] - 0s 2ms/step - loss: 5.3471e-04 - accuracy: 1.0000
Epoch 97/100
21/21 [=====] - 0s 2ms/step - loss: 5.4076e-04 - accuracy: 1.0000
Epoch 98/100
21/21 [=====] - 0s 2ms/step - loss: 5.0498e-04 - accuracy: 1.0000
Epoch 99/100
21/21 [=====] - 0s 2ms/step - loss: 4.7440e-04 - accuracy: 1.0000
Epoch 100/100
21/21 [=====] - 0s 2ms/step - loss: 4.7000e-04 - accuracy: 1.0000
2/2 [=====] - 0s 8ms/step - loss: 0.1151 - accuracy: 0.8571

Test Accuracy: 0.8571
2/2 [=====] - 0s 10ms/step - loss: 0.0103 - accuracy: 0.9773

Test Accuracy: 0.9773
```

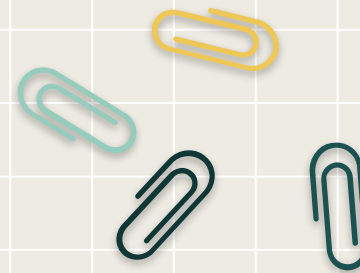


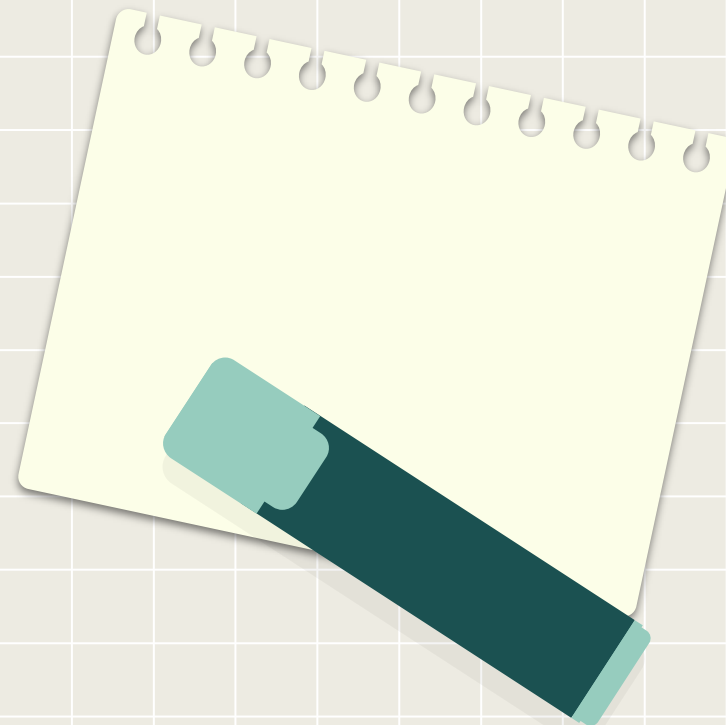
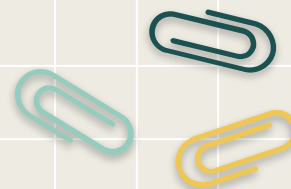
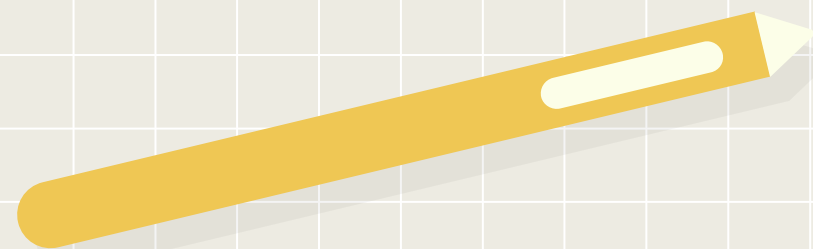

Warning!!!

전체 데이터 양이 너무 적은 경우에는 검증 세트의 비율이나 매개변수의 값이 조금만 바뀌어도 성능 평가 점수가 크게 변함



데이터 양이 너무 적은 경우에는 검증 세트를 나누지 않고 교차검증(cross validation)을 사용!!!





Thanks!!



모델 저장과 재사용

전체 모델 저장 및 로딩

학습이 끝난 후 만든 모델을 저장하면 언제든지 이를 불러와 다시 사용할 수 있습니다.
학습 결과를 저장하려면 `model.save()` 함수를 이용해 모델 이름을 적어 저장합니다.

```
# 모델 이름과 저장할 위치를 함께 지정합니다.  
model.save('./data/model/my_model.hdf5')
```

저장된 파일에는 다음의 정보가 담겨 있다

- 나중에 모델을 재구성하기 위한 모델의 구성정보
- 모델을 구성하는 각 뉴런들의 가중치
- 손실함수, 최적하기 등의 학습 설정
- 재학습을 할 수 있도록 마지막 학습 상태

TensorFlow SavedModel & Keras h5

- SavedModel 형식 (텐서플로 기본값)
 - model.save()를 사용할 때의 기본값
 - 모델 이름의 폴더를 만들고 그 안에 모델 구조 및 훈련구성(옵티마이저, 손실 등)랑 가중치 모두 저장
 - model.save('my_model')
- Keras h5 형식 (케라스 기본값)
 - Keras는 모델의 구조, 가중치 값, compile() 정보를 포함하는 단일 H5 파일 저장을 지원
 - SavedModel의 경량 대안

모델 저장 및 불러오기

(저장)

```
model = create_model()

model.fit(train_images, train_labels, epochs=5)

# 전체 모델을 HDF5 파일로 저장합니다
model.save('my_model.h5')
```

(불러오기)

```
# 가중치와 옵티마이저를 포함하여 정확히 동일한 모델을 다시 생성합니다
new_model = keras.models.load_model('my_model.h5')
new_model.summary()

loss, acc = new_model.evaluate(test_images, test_labels, verbose=2)
print("복원된 모델의 정확도: {:.2f}%".format(100*acc))
# 복원된 모델의 정확도: 84.20%
```

모델의 가중치 값만 저장 및 로딩

매일 전체 모델을 저장하고 로딩하다보면 학습에 시간이 몇일이나 걸리는 매우 복잡한 모델을 훈련시킬 때 굉장히 비효율적일수 있다.

다음과 같은 경우에 유용할 수 있습니다.

- 추론을 위한 모델만 필요합니다. 이 경우 훈련을 다시 시작할 필요가 없으므로 컴파일 정보나 옵티마이저 상태가 필요하지 않습니다.
- 전이 학습을 수행하고 있습니다. 이 경우 이전 모델의 상태를 재사용하는 새 모델을 훈련하므로 이전 모델의 컴파일 정보가 필요하지 않습니다.

TF Checkpoint 형식 (체크포인트 콜백함수)

- 이는 진행하다가 도중에 시작할 수 있는 저장점으로, 훈련시에도 이러한 체크포인트를 둘 수 있다
- 콜백함수를 통해 우리는 학습되는 과정 사이에 학습률을 변화시키거나 val_loss가 개선되지 않으면 학습을 멈추게 하는 등의 작업을 할 수 있다

TF Checkpoint

```
checkpoint_path = "training_1/cp.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)
```

- 체크포인트 정보가 저장될 경로를 적어주고, 그 경로에 디렉토리를 만들어준다
- 이때 쓰인 파일 확장자가 .ckpt인데 이는 체크포인트 파일입니다 보통 가중치만 저장할 때 이를 사용
- os.path.dirname() 함수를 통해 경로 중 디렉토리 명만 얻는다 이는 입력 경로의 폴더경로까지 꺼내준다

체크포인트 콜백 만들기

```
cp_callback = tf.keras.callbacks.ModelCheckpoint(checkpoint_path,
                                                  save_weights_only=True,
                                                  verbose=1)
```

```
model = create_model()
```

```
model.fit(train_images, train_labels, epochs = 10,
          validation_data = (test_images, test_labels),
          callbacks = [cp_callback]) # 훈련 단계에 콜백을 전달합니다
```


Tensorflow, Keras 콜백함수 ModelCheckpoint

```
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  
)
```

인자	설명
filepath	모델을 저장할 경로를 입력합니다. 추가 설명으로 만약 monitor가 val_loss일 때, 모델 경로를 '{epoch:02d}-{val_loss:.5f}.h5' 라고 입력하면, 에폭-해당에폭에서의 val_loss.h5로 모델이 저장됩니다. 예: 01-0.39121.h5 로 저장됩니다.
monitor	모델을 저장할 때, 기준이 되는 값을 지정합니다. 예를 들어, validation set의 loss가 가장 작을 때 저장하고 싶으면 'val_loss'를 입력하고 만약 train set의 loss가 가장 작을 때 모델을 저장하고 싶으면 'loss'를 입력합니다. 이 외에도 다양한 값들을 기준으로 삼을 수 있습니다.
verbose	0, 1 1일 경우 모델이 저장 될 때, '저장되었습니다' 라고 화면에 표시되고, 0일 경우 화면에 표시되는 것 없이 그냥 바로 모델이 저장됩니다.
save_best_only	True, False True 인 경우, monitor 되고 있는 값을 기준으로 가장 좋은 값으로 모델이 저장됩니다. False인 경우, 매 에폭마다 모델이 filepath(epoch)으로 저장됩니다. (model0, model1, model2,...)

save_weights_only	True, False True인 경우, 모델의 weights만 저장됩니다. False인 경우, 모델 레이어 및 weights 모두 저장됩니다.
mode	'auto', 'min', 'max' val_acc 인 경우, 정확도이기 때문에 클수록 좋습니다. 따라서 이때는 max를 입력해줘야합니다. 만약 val_loss 인 경우, loss 값이기 때문에 값이 작을수록 좋습니다. 따라서 이때는 min을 입력해줘야합니다. auto로 할 경우, 모델이 알아서 min, max를 판단하여 모델을 저장합니다.
save_freq	'epoch' 또는 integer(정수형 숫자) 'epoch'을 사용할 경우, 매 에폭마다 모델이 저장됩니다. integer을 사용할 경우, 숫자만큼의 배치를 진행되면 모델이 저장됩니다. 예를 들어 숫자 8을 입력하면, 8번째 배치가 train 된 이후, 16번째 배치가 train 된 이후 모델이 저장됩니다.
options	tf.train.CheckpointOptions를 옵션으로 줄 수 있습니다. 분산환경에서 다른 디렉토리에 모델을 저장하고 싶을 경우 사용합니다. 자세한 내용은 아래 링크를 참조해주세요. www.tensorflow.org/api_docs/python/tf/train/CheckpointOptions

Tensorflow, Keras 콜백함수 ModelCheckpoint

save_weights_only	<p>True, False</p> <p>True인 경우, 모델의 weights만 저장됩니다. False인 경우, 모델 레이어 및 weights 모두 저장됩니다.</p>
verbose	<p>0, 1</p> <p>1일 경우 모델이 저장 될 때, '저장되었습니다' 라고 화면에 표시되고, 0일 경우 화면에 표시되는 것 없이 그냥 바로 모델이 저장됩니다.</p>

TF Checkpoint

체크포인트 콜백 만들기

```
cp_callback = tf.keras.callbacks.ModelCheckpoint(checkpoint_path,  
                                                  save_weights_only=True,  
                                                  verbose=1)
```

```
model = create_model()
```

```
model.fit(train_images, train_labels, epochs = 10,  
          validation_data = (test_images, test_labels),  
          callbacks = [cp_callback]) # 훈련 단계에 콜백을 전달합니다
```

- tf.keras에 정의된 ModelCheckpoint 함수를 만든다
- 모델의 weights만 저장한다
- 모델이 저장될 때 화면에 표시되게 한다
- 훈련단계에 콜백을 전달한다

736/1000 [=====>.....] - ETA: 0s - loss: 0.0657 - accuracy: 0.9918

Epoch 00008: saving model to training_1/cp.ckpt

- 실제 학습을 진행하면, 에포크마다 학습 진행정도가 파일로 저장되어 백업된다

TF Checkpoint

```
model = create_model()

loss, acc = model.evaluate(test_images, test_labels, verbose=2)
print("훈련되지 않은 모델의 정확도: {:.2f}%".format(100*acc))
```

1000/1 - 0s - loss: 2.4276 - accuracy: 0.0830

훈련되지 않은 모델의 정확도: 8.30%

- 훈련된 데이터의 파라미터를 가져오기 위하여, 새롭게 학습되지 않은 모델을 만들어준다
- 현재 상태의 정확도를 보면 알 수 있듯이, 아예 학습이 진행되지 않은 모델임을 확인할 수 있다

TF Checkpoint

```
model.load_weights(checkpoint_path)
loss, acc = model.evaluate(test_images, test_labels, verbose=2)
print("복원된 모델의 정확도: {:.2f}%".format(100*acc))
```

1000/1 - 0s - loss: 0.3528 - accuracy: 0.8680

복원된 모델의 정확도: 86.80%

- load_weights를 사용하여, 체크포인트의 path를 인자값으로 넣어주면, 딱히 학습을 하지 않고도, 이전에 학습된 모델의 파라미터를 복사해 가져온다
- verbose : 함수 수행시 발생하는 상세한 정보들을 표준 출력으로 자세히 내보낼 것인가를 나타낸다

수동으로 가중치 저장

```
# 가중치를 저장합니다
```

```
model.save_weights('./checkpoints/my_checkpoint')
```

```
# 가중치를 복원합니다
```

```
model = create_model()
```

```
model.load_weights('./checkpoints/my_checkpoint')
```

```
loss, acc = model.evaluate(test_images, test_labels, verbose=2)
```

```
print("복원된 모델의 정확도: {:.2f}%".format(100*acc))
```

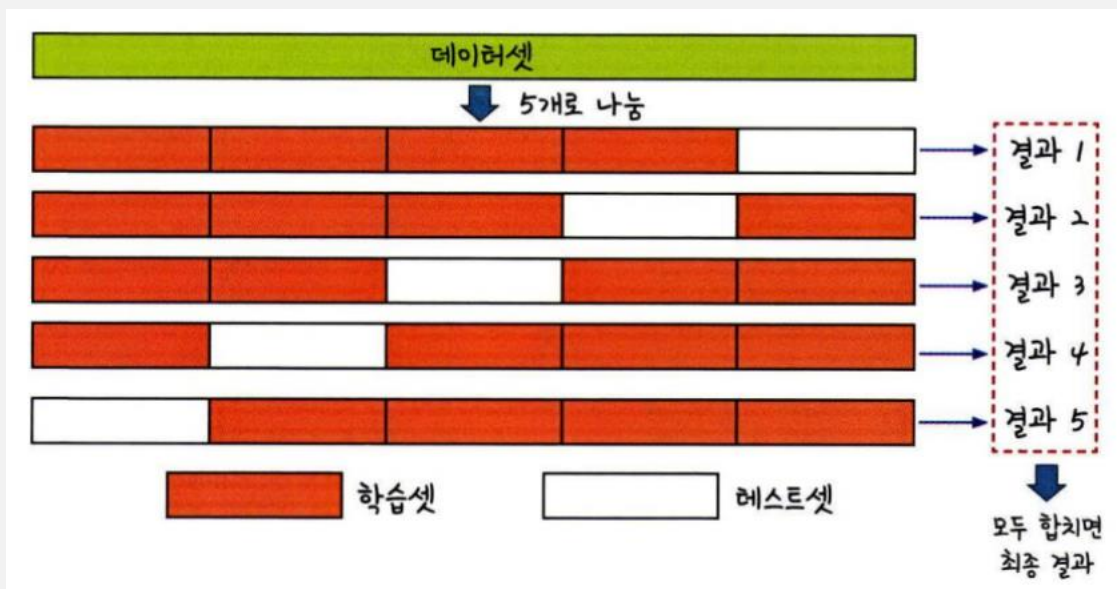
- 위와 같이 콜백 함수가 아닌 수동으로도 save를 할 수 있다

C H 1 3 과 적 합 피 하 기

k접교차검증

k겹 교차 검증이란?

데이터셋을 여러 개로 나누어
하나씩 테스트셋으로 사용하고
나머지를 모두 합해서 학습셋으로 사용하는 방법



StratifiedKFold

(n_splits, shuffle, random_state)

n_splits : fold의 개수 k값

(기본값 5, 정수형)

shuffle : 전체 데이터셋 분할 전 섞을지의 여부

(기본값 False)

random_state : 난수 설정

k겹 교차 검증의 장단점

01

과적합 방지

학습셋과 테스트셋을 나누어 작업

02

데이터셋의 활용 (일반화, 정확도 향상)

가지고 있는 데이터의 100%를 테스트셋으로 사용

03

연산 비용의 증가

모델을 k개 만들어야 하므로 속도 저하