

# 딥러닝 신경망을 이용한 분류 모델(DNN\_Classification)

- ☐ 모델의 구조(드롭아웃 등) 설명 추가
- ☐ 활성화 함수 설명 추가(ReLU 성능이 좋은 이유)
- ☐ 학습률 설명 추가

- 1. 데이터 전처리
  - (1) 명목형 데이터
  - (2) 숫자형 데이터
- 2. 모델 구성 및 학습, 하이퍼 파라미터 튜닝
  - (1) 모델의 구성
  - (2) 하이퍼 파라미터 튜닝
  - (3) 최적의 하이퍼 파라미터 서치
    - (3)-1. 연속형 컬럼 2개 추가 전 학습
    - (3)-2. 연속형 컬럼 2개 추가 전 학습
- 3. 정리
  - (1) 손실 최고화 하기
  - (2) 하이퍼 파라미터 최적화와 옵티마이저
    - Q1. 그리드 서치 및 랜덤서치 등의 하이퍼 파라미터 튜닝 클래스를 사용하지 않은 이유
    - Q2. 옵티마이저로 Adam만 사용하는 이유 설명
  - (+) 정형 데이터 분류: tabnet

## 1. 데이터 전처리

- 1차원 배열 형태로 입력층에 들어가야하기 때문에 전처리가 필요.

데이터 타입	전처리 방법
문자형 명목 데이터	방법 1) LabelEncode() 사용후 Embedding Layer 방법 2) one-hot 인코딩 사용
숫자형 명목 데이터	Embedding Layer 통과하여 숫자 간 서열 제거
연속형 데이터	스케일링 및 정규화: StandardScaler() 사용

### (1) 명목형 데이터

전체 명목형 데이터	사용할 명목형 데이터
USER_ID_hash	
VIEW_COUPON_ID_hash	
PREF_NAME, Tr_Pref_Name	Tr_Pref_Name
small_area_name, Tr_small_area_name	Tr_small_area_name
capsule_text, Translated_capsule_text	
Translated_genre_name	Translated_genre_name

Q1. USER\_ID\_hash 및 VIEW\_COUPON\_ID\_hash 사용하지 않은 이유

컬럼 이름	class의 개수
USER_ID_hash	15527
VIEW_COUPON_ID_hash	19404

해당 컬럼을 확인해보면 고유한 클래스값이 상당히 많다는 것을 확인할 수 있다. 따라서 주어진 시간 내에 모델에 적용하기 어려울 것이라고 판단하여 해당 모델에서는 제거하고 사용하기로 결정하였다(적용시 과적합이 되는 문제가 있어 제거하는 방식으로 데이터 전처리)

Q2. 출력층의 Target으로 Translated\_genre\_name 사용 이유

Translated\_genre\_name 와 capsule\_text는 동일한 속성의 데이터 분류이다. 다만 capsule\_text의 경우 Translated\_genre\_name 보다 분류가 조금 더 상세하다. 따라서 대분류에 해당하는 Translated\_genre\_name을 사용하였다.

### (2) 숫자형 데이터

- 숫자형식으로 된 명목 데이터 - 1차원 배열로 변환하여 Embedding layer에 넣기
- 연속형 데이터 - StandardScaler() 사용하여 정규분포 하도록 스케일링/정규화

전체 숫자형 명목 데이터	연속형 데이터
PURCHASE_FLG	AGE
disperperiod	view_count

usable_date_mon, usable_date_tue, usable_date_wed, usable_date_thu, usable_date_fri, usable_date_sat, usable_date_sun	PRICE_RATE
usable_date_holiday, usable_date_before_holiday	DISCOUNT_PRICE
Male, Female	
usable_date_sum	

- 총 사용하는 컬럼 - 명목형 16개, 연속형 4개
- 출력층 - 13개의 클래스를 가진 genre\_name

## 2. 모델 구성 및 학습, 하이퍼 파라미터 튜닝 [↗](#)

### (1) 모델의 구성 [↗](#)

DNN: 심층 신경망 모델

- 모델의 구조

**input - embedding layer - dense layer\_hidden1 - densel layer\_hidden2 - output**

- Dense 밀집층 사용 - 층의 개수는 2개로 고정한다.
- 드롭아웃: 각 은닉층을 거칠 때마다 0.5의 드롭아웃층을 거치도록 구성하여 과대적합을 피함. test용으로 돌린 트라이얼 모델에서 드롭아웃을 0.3으로 설정하였을때 과적합이 발생하였으므로 해당 모델에서는 고정적으로 0.5를 사용하기로 하였다.
- 콜백 - 조기종료 사용. patience = 5로 고정 사용.
- 활성화 함수 - 성능을 위해 모든 모델의 은닉층에 렐루 함수를 사용하였다.
- 출력층 활성화 함수 - 다중분류에 해당하는 모델이기 때문에 softmax 함수를 사용하였다.

### (2) 하이퍼 파라미터 튜닝 [↗](#)

하이퍼 파라미터 튜닝

neuron_counts	batch_size	learning_rate
13	64	0.1
50	256	0.01
100	512	0.001

수동으로 조절해줄 파라미터 = 층의 뉴런의 개수(neuron\_counts), batch\_size, learning\_rate

그밖에도 epochs와 optimizer 등을 조절하여 튜닝이 가능하지만 수동으로 조절하는 것에 한계가 있으므로 다음의 세 가지 파라미터를 수정하여 성능 및 손실의 변화를 살펴보고 최적의 하이퍼 파라미터 값을 찾아보았다.

#### 1) neuron\_counts = [13,50,100]

해당 모델은 두 개의 은닉층으로 구성되어 있으며 해당 층에 각각 뉴런의 개수를 다르게 두는 것이 가능하다. 해당 분석에서는 각 층의 뉴런 개수를 동일하게 조절하는 방법을 선택하였다. 먼저 뉴런의 개수는 출력층의 타겟수보다 적으면 정보가 부족하게 전달되므로 최소 뉴런의 개수를 출력층의 뉴런인 13개로 잡았다.

다음으로는 50개(model2로 구현), 100개(model3로 구현)로 설정하여 성능 및 손실의 추이를 살펴다.

#### 2) batch\_size = [64,256,512]

배치 사이즈란 파라미터를 초기화하는 데이터의 단위를 설정하는 파라미터이다. GPU의 물리적인 구조로 인해 항상 2의 제곱으로 설정하는 것이 일반적이므로 64,256,512 이 세 가지 batch\_size를 설정하여 튜닝을 진행하였다.

#### 3) learning\_rate = [0.1,0.01, 0.001]

(학습률에 대한 설명 추가)

### (3) 최적의 하이퍼 파라미터 서치 [↗](#)

수동으로 튜닝하는 것의 한계는 경우의 수는 많은데 시간과 자원은 한정되어 있다는 점이다. 때문에 먼저 뉴런개수 13개로 파라미터를 고정한 mode1을 기준으로 학습률과 배치 사이즈를 튜닝한 뒤, 해당 9 개의 모델에서 가장 효율이 좋은(과대적합 없이 성능이 상대적으로 우수하며 손실이 적은) 배치사이즈와 학습률을 가지고 뉴런의 개수를 조절하여 튜닝을 진행하였다.

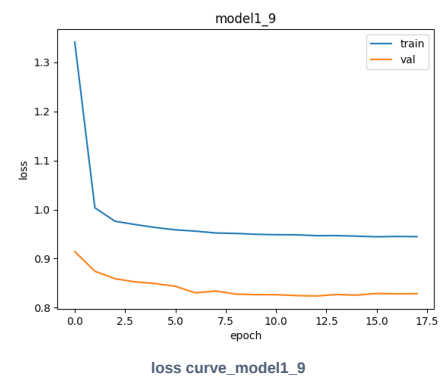
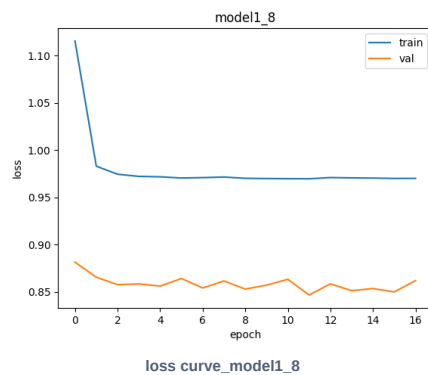
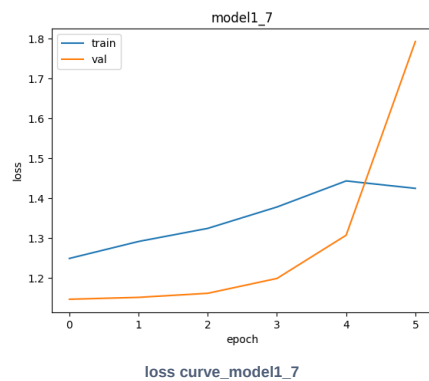
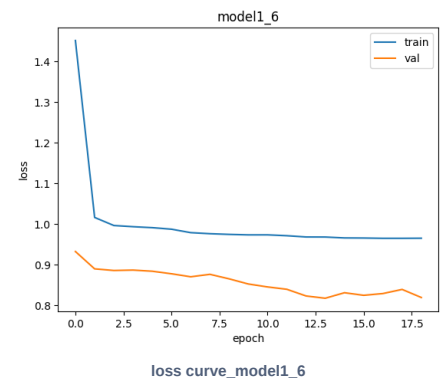
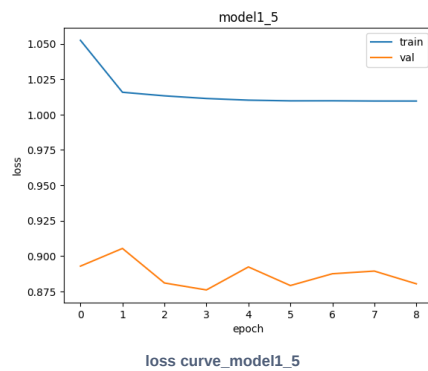
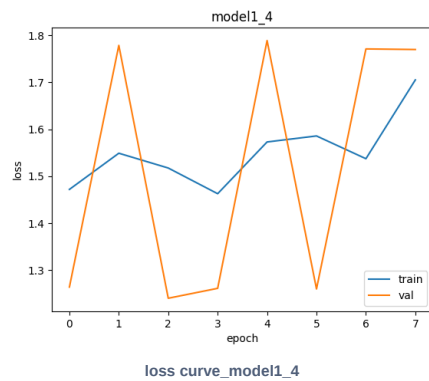
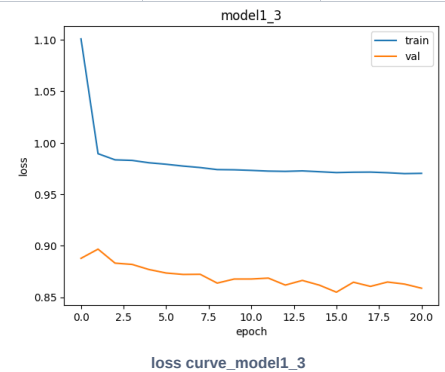
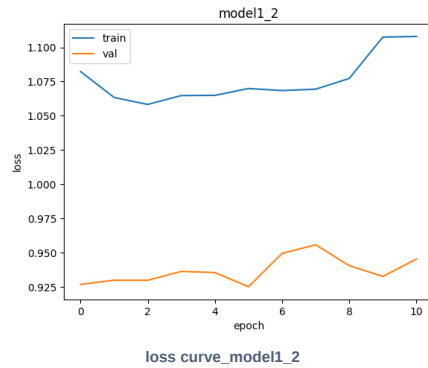
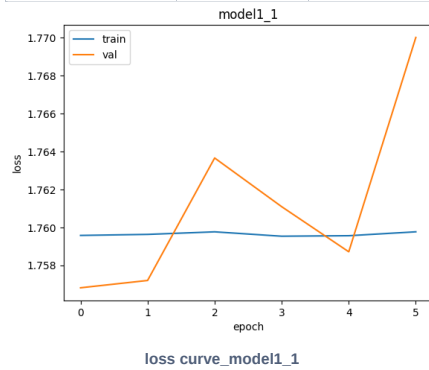
#### (3)-1. 연속형 컬럼 2개 추가 전 학습 [↗](#)

**step1.** model1(뉴런개수 13개) - 최적의 학습률과 배치 사이즈를 찾기

모델 파라미터의 개수: **856개**

model	neuron	batch	lr	train_acc	test_acc	loss	종료	시간
model1_1	13	64	0.1	27.82%	28.86%	1.759	epoch=6	6m

model1_2	13	64	0.01	64.72%	71.16%	1.108	epoch=11	9m
model1_3	13	64	0.001	68.40%	71.36%	0.970	epoch=21	19m
model1_4	13	256	0.1	35.30%	28.86%	1.705	epoch=8	2m
model1_5	13	256	0.01	66.72%	71.09%	1.009	epoch=9	2m
model1_6	13	256	0.001	68.51%	72.27%	0.964	epoch=19	4m
model1_7	13	512	0.1	47.97%	53.59%	1.424	epoch=6	1m
model1_8	13	512	0.01	68.49%	72.05%	0.97	epoch=17	1m
model1_9	13	512	0.001	69.7%	72.18%	0.944	epoch=18	2m



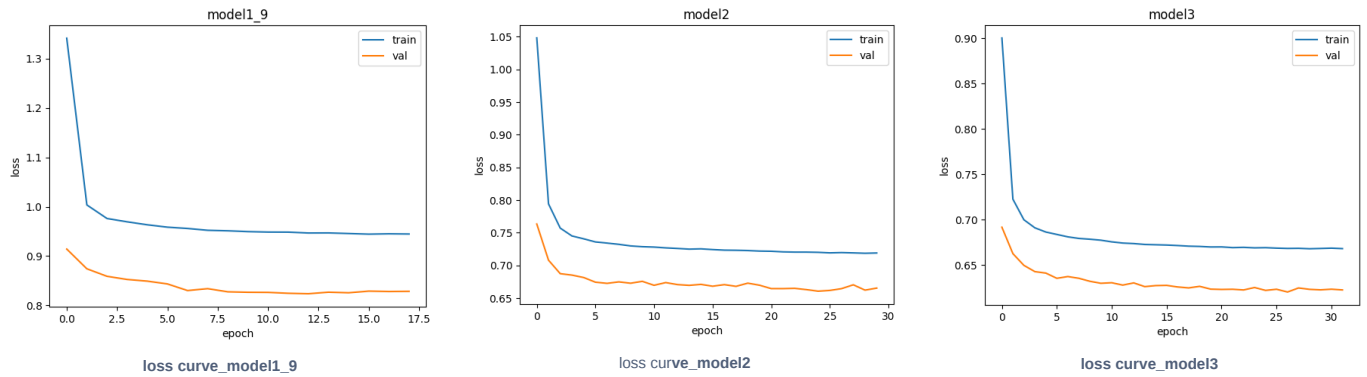
loss curve의 모양이나 성능 및 손실값을 미루어 보았을때 **model1\_6, model1\_8, model1\_9**의 파라미터가 효율적으로 판단된다. 훈련 시간 대비 성능 면에서 볼 때 model1\_9가 가장 우수하다고 판단하였다.

학습률이 높을수록 loss는 줄어들었지만 모델이 돌아가는 시간이 길어졌으며 배치 사이즈가 커질수록 모델 학습 시간이 줄어드는 것을 확인할 수 있었다.

**step2.** model1\_9의 학습률과 배치사이즈를 가지고 최적의 뉴런 개수 찾기

batch\_size = 512, learning\_rate = 0.001로 고정한 뒤 각 층의 뉴런의 개수를 다르게 조절한다. model2의 경우 각 밀집층의 뉴런의 개수가 50개이며, model3는 100개이다.

model	neuron	Total params	train_acc	test_acc	loss	종료	시간
model1_9	13	856	69.7%	72.18%	0.944	epoch=18	2m
model2	50	4,149	75.48	76.94%	0.7188	epoch=18	6m
model3	100	12,949	77.11%	78.29%	0.6682	epoch=32	10m



손실 함수의 모양은 비슷하였으나 가장 손실이 적고 성능이 높았던 것은 **model3** 즉, 뉴런의 개수가 100개인 모델이 가장 우수하다고 판단하였다.

### (3)-2. 연속형 컬럼 2개 추가 전 학습

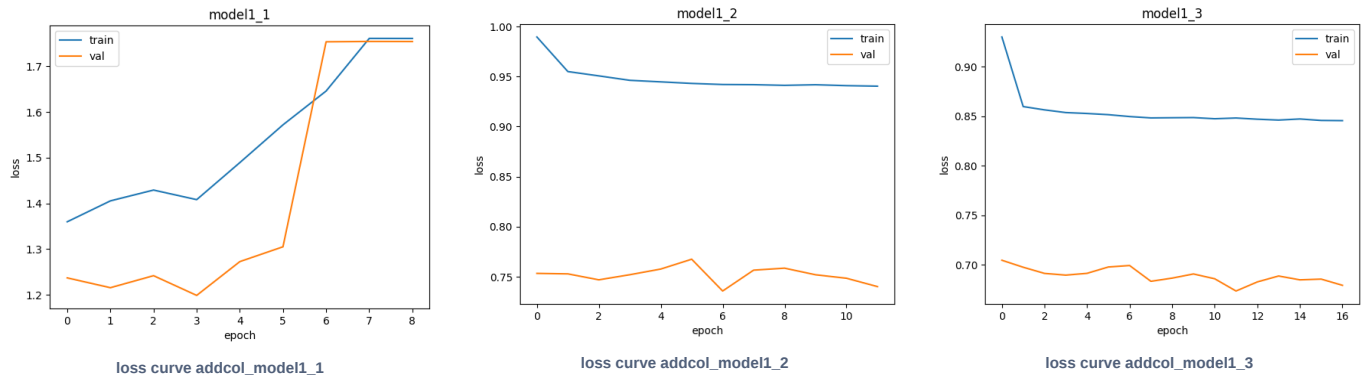
이번에는 연속형 컬럼인 'PRICE\_RATE', 'DISCOUNT\_PRICE'가 추가된 데이터를 모델에 학습해보았다.

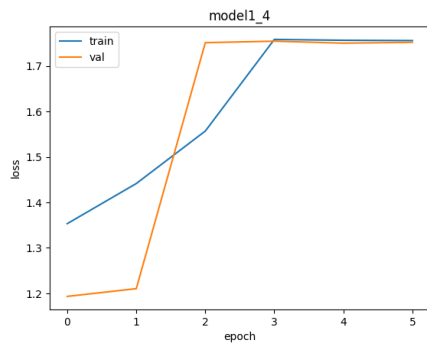
**step1.** model1(뉴런개수 13개) - 최적의 학습률과 배치 사이즈를 찾기

컬럼 추가 전과 비교했을때 약간의 성능 상승과 손실 감소를 확인할 수 있다.

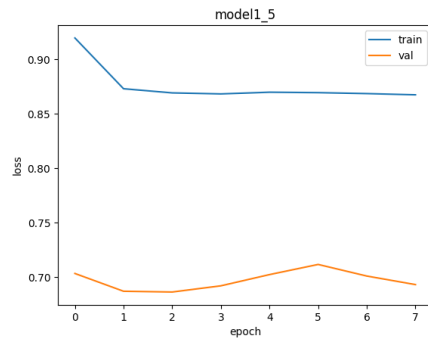
모델 파라미터의 개수: **882**(데이터 추가 전 모델보다 파라미터 수가 약간 늘어남)

model	neuron	batch	lr	train_acc	test_acc	loss	종료	시간
model1_1	13	64	0.1	27.79	53.559%	1.7600	epoch=9	10m
model1_2	13	64	0.01	72.19	77.224%	0.9405	epoch=12	15m
model1_3	13	64	0.001	0.7334	79.570%	0.845	epoch=17	14m
model1_4	13	256	0.1	0.2818	45.357%	1.7559	epoch=6	1m
model1_5	13	256	0.01	0.7271	78.332%	0.8674	epoch=8	2m
model1_6	13	256	0.001	0.7471	78.292%	0.8569	epoch=7	1m
model1_7	13	512	0.1	0.4941	76.739%	1.3448	epoch=6	1m
model1_8	13	512	0.01	0.7348	78.197%	0.8614	epoch=9	1m
model1_9	13	512	0.001	0.7499	79.505%	0.8496	epoch=18	2m

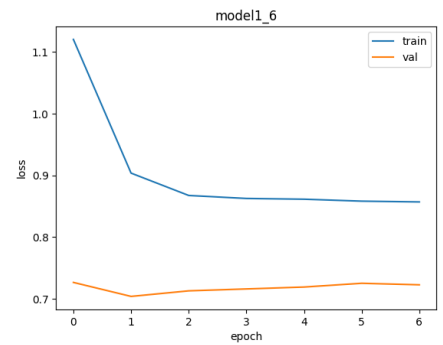




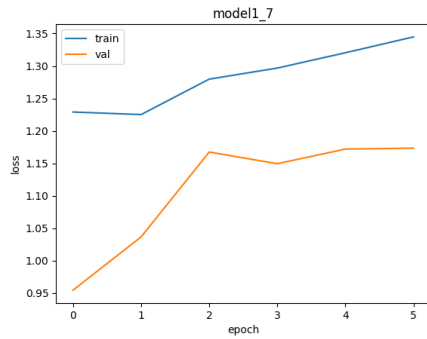
loss curve addcol\_model1\_4



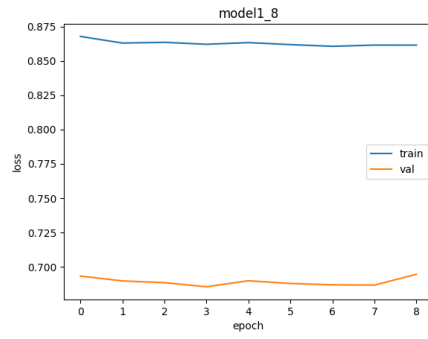
loss curve addcol\_model1\_5



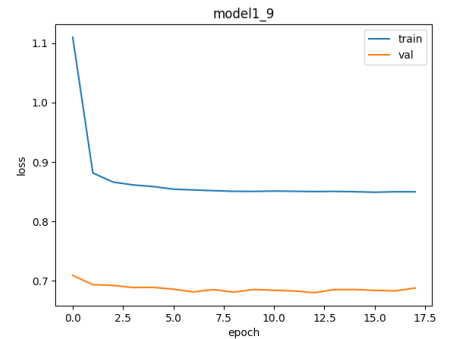
loss curve addcol\_model1\_6



loss curve addcol\_model1\_1



loss curve addcol\_model1\_8



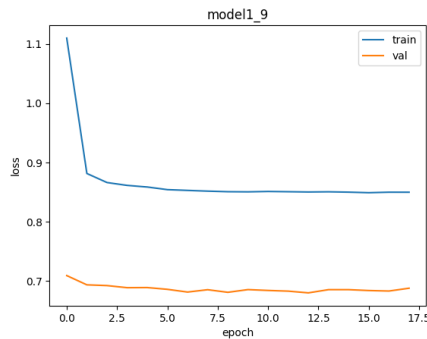
loss curve addcol\_model1\_9

step2. model1\_9의 학습률과 배치사이즈를 가지고 최적의 뉴런 개수 찾기

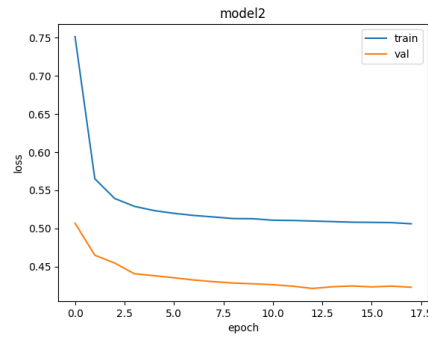
각 층의 뉴런의 개수를 다르게 조절 - model2, model3

model	neuron	params	train_acc	test_acc	loss	종료	시간
model1_9	13	856	0.7499	79.505%	0.8496	epoch=18	2m
model2	50	4,249	0.8416	86.645%	0.5060	epoch=18	3m
model3	100	13,149	0.8697	88.760%	0.4123	epoch=34	10m

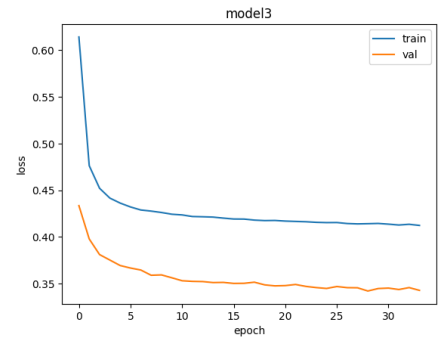
model3를 사용할 경우 loss는 0.4에 근접하게 수렴하며 성능은 88.76%까지 올라가는 것을 확인할 수 있다.



loss curve addcol\_model1\_9



loss curve addcol\_model2



loss curve addcol\_model3

### 3. 정리


#### (1) 손실 최소화 하기

하이퍼 파라미터에 따른 손실과 성능의 관계 확인

## (2) 하이퍼 파라미터 최적화와 옵티마이저 [↗](#)

### Q1. 그리드 서치 및 랜덱서치 등의 하이퍼 파라미터 튜닝 클래스를 사용하지 않은 이유 [↗](#)

keras에서는 자체적으로 하이퍼 파라미터 튜닝을 위한 KerasTuner 클래스를 제공하고 있다.

 [Keras documentation: KerasTuner](#)

```
1 import keras_tuner
2 from tensorflow import keras
```

하지만 주피터노트북에서 keras에서 제공하는 예제코드 실행시 FailedPreconditionError가 발생하는 것을 확인하였고 해결을 위해 구글링해 본 결과 이는 현재 주피터에서 사용하는 tensorflow의 최신 버전이 해당 keras\_tuner에 호환되지 않았다.

참고:  [tensorflow 오류 해결하기](#) | FailedPreconditionError: Error while reading resource variable \_AnonymousVar443 from Container: localhost.

튜너 사용을 위해 다운그레이드하여 튜너 사용을 시도했지만 현재 다운그레이드할 수 있는 버전보다 더 낮은 버전을 사용해야 keras\_tuner를 사용할 수 있는 것으로 판단되어 직접 하이퍼 파라미터 튜닝을 시도하게 되었다.

### Q2. 옵티마이저로 Adam만 사용하는 이유 설명 [↗](#)

(설명 추가 예정)

- 연구 결과에 따르면, Adam 알고리즘에서 학습률은 0.001 이 가장 좋은 설정 값이라고 알려져 있습니다.

test: 뉴런의 개수가 작은 모델로 최적의 parameter 값을 찾고 이를 기반으로 성능이 가장 안정적이었던 뉴런 50개의 모델을 가지고 학습하였다.

## (+) 정형 데이터 분류: tabnet [↗](#)