1. 속도 향상을 위한 테크닉

1 조기 종료

속도 향상을 위한 테크닉이 필요한 이유와 확장성

1.
조기 종료

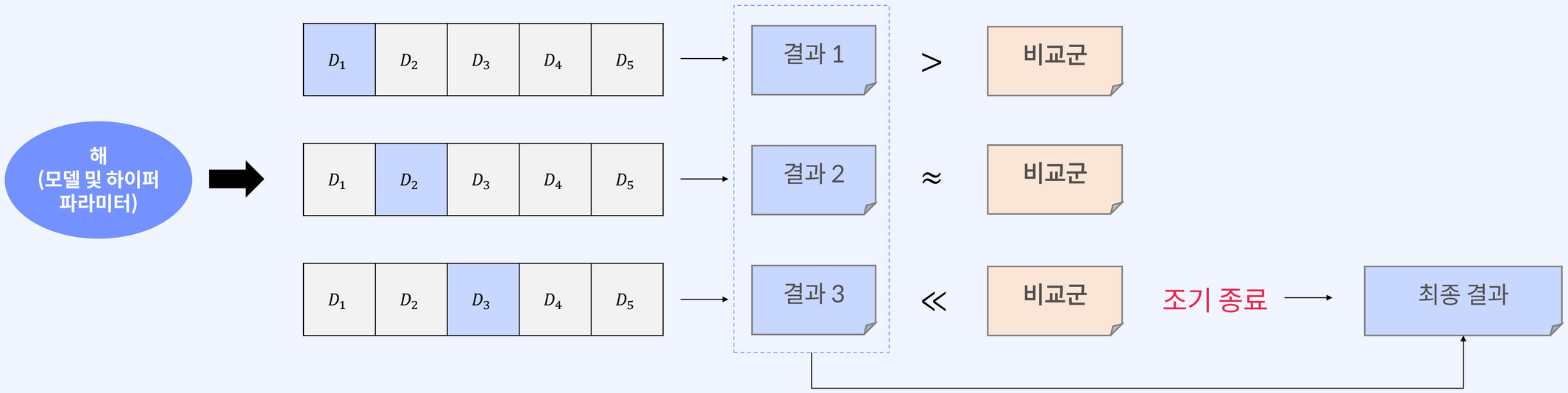
하이퍼파라미터 튜닝을 비롯한 머신러닝 자동화에서 다루는 최적화 문제는 하나의 해를 평가하는 데 시간이 오래 걸림

게다가 여러 개의 해를 탐색해서 비교해야 한다는 점을 고려하면 더더욱 오랜 시간이 걸림

글로벌 대기업은 수많은 컴퓨팅 자원을 사용해 머신러닝 파이프라인 평가 작업을 나누어 할당하고 여러 작업을 병렬적으로 수행함으로써 탐색 시간을 줄이는데, 이를 확장성이라 함

조기 종료란?

조기 종료는 폴드를 평가하는 중에 다른 해보다 현저히 낮은 성능이 나오면 그대로 평가를 중단시키는 방법입니다.



조기 종료 실습

그리드 서치를 이용해 신경망의 하이퍼파라미터를 튜닝하는 과정에서 조기 종료를 적용해보겠습니다.

하이퍼 파라미터 튜닝 범위

하이퍼 파라미터	범위
은닉층	{(10,), (10, 10, 10), (20, 20), (30, 30, 30)}
조기 종료 여부[1)]	{True, False}
최대 이터레이션 횟수	{100, 500, 1000}

[1)] 여기서 말하는 조기 종료는 신경망의 하이퍼 파라미터입니다.

데이터 준비 및 환경 설정

실습에 사용할 데이터를 불러오고 특징과 라벨로 분리하겠습니다.

데이터 불러오기

```
1  import pandas as pd
2  df = pd.read_csv("../../data/classification/winequality-red.csv")
3  X = df.drop('y', axis = 1)
4  y = df['y']
```

경고 무시하기

```
1  import warnings
2  warnings.filterwarnings("ignore")
```

하이퍼 파라미터 그리드 정의

하이퍼파라미터 튜닝에 필요한 모듈을 불러오고 하이퍼 파라미터 그리드를 정의하겠습니다.

하이퍼 파라미터 그리드 정의

```
1  from sklearn.neural_network import MLPClassifier as MLP
2  from sklearn.model_selection import ParameterGrid, KFold
3  from sklearn.metrics import accuracy_score
4
5  grid = ParameterGrid(
6          {"hidden_layer_sizes": [(10, ), (10, 10, 10), (20, 20), (30, 30, 30)],
7           "early_stopping": [True, False],
8           "max_iter": [100, 500, 1000],
9           "random_state":[2022]})
```

- **라인 9:** random_state는 고정하기 위해 그리드에 추가했습니다.

그리드 서치 시간 측정

조기 종료를 사용하지 않고 그리드 서치를 적용했을 때의 시간을 측정해보겠습니다.

그리드 서치 시간 측정

```
1  import time
2  kf = KFold(n_splits = 5, shuffle = True, random_state = 2022)
3  start_time = time.time()
4  best_score = -1
5  for parameter in grid:
6      score = 0
7      for train_index, test_index in kf.split(X):
8          X_train, X_test = X.loc[train_index], X.loc[test_index]
9          y_train, y_test = y.loc[train_index], y.loc[test_index]
10         model = MLP(**parameter).fit(X_train, y_train)
11         y_pred = model.predict(X_test)
12         score += accuracy_score(y_test, y_pred) / 5
13     if score > best_score:
14         best_score = score
15         best_parameter = parameter
16
17 end_time = time.time()
18 print(best_parameter, best_score)
19 print(int(end_time - start_time))
```

- **라인 3:** 반복문에 들어가기 전의 시간을 측정하여 start_time에 저장합니다.
- **라인 7~12:** 파라미터가 parameter일 때 신경망 모델을 5-겹 교차 검증 방식으로 평가한 정확도를 score에 저장합니다.
- **라인 13~15:** 현재까지 찾은 최고 점수인 best_score보다 현재 찾은 점수인 score가 더 크다면, best_score와 best_parameter를 각각 score와 parameter로 업데이트합니다.
- **라인 17:** 반복문이 종료됐을 때의 시간을 측정하여 end_time에 저장합니다.

```
{'early_stopping': False, 'hidden_layer_sizes': (10, 10, 10), 'max_iter': 1000, 'random_state': 2022}
0.5885050940438872

89
```

조기 종료 시 그리드 서치 시간 측정 (누적 평균)

```
1  start_time = time.time()
2  best_score = -1
3  for parameter in grid:
4      score = 0
5      k = 0
6      for train_index, test_index in kf.split(X):
7          X_train, X_test = X.loc[train_index], X.loc[test_index]
8          y_train, y_test = y.loc[train_index], y.loc[test_index]
9          model = MLP(**parameter).fit(X_train, y_train)
10         y_pred = model.predict(X_test)
11         score += accuracy_score(y_test, y_pred) / 5
12         k += 1
13         if score * (5/k) < best_score * 0.95:
14             break
15
16     if score > best_score:
17         best_score = score
18         best_parameter = parameter
19
20 end_time = time.time()
21 print(best_parameter, best_score)
22 print(int(end_time - start_time))
```

- **라인 5:** 현재 평가하는 폴드 인덱스 k를 0으로 초기화합니다.
- **라인 7~12:** k번째 폴드에 대해 하이퍼파라미터 평가를 진행합니다.
- **라인 13~14:** k번째 폴드까지의 평균 정확도인 score * (5/k)가 best_score에 0.95를 곱한 것보다 작으면 break를 사용해 for 문을 빠져나옵니다. 즉, 조기 종료를 합니다. 여기서 score에 (5/k)를 곱한 이유는 score는 (정확도/5)를 이터레이션마다 더한 것이기에 누적 평균으로 변환하기 위해서입니다.

```
{'early_stopping': False, 'hidden_layer_sizes': (10, 10, 10), 'max_iter': 1000, 'random_state': 2022}
0.5885050940438872

81
```

- 모든 파라미터에 대해 5-겹 교차 검증을 했을 때보다 약간의 시간 감소 (약 8.9%)가 있음
- 극적으로 시간이 줄지 않은 이유는 라인 13의 조기 종료 조건을 만족하는 파라미터가 거의 없었기 때문으로 보임
- 5-겹 교차 검증을 하는 상황에서 조기 종료했을 때 얻을 수 있는 최대 시간 감소율은 약 80%지만, 최악의 상황 에서는 오히려 시간이 증가할 수도 있습니다.

조기 종료 시 그리드 서치 시간 측정 (폴드 평가값 기준)


```
1  import numpy as np
2  start_time = time.time()
3  best_score = -1
4  for parameter in grid:
5      score_list = []
6      for train_index, test_index in kf.split(X):
7          X_train, X_test = X.loc[train_index], X.loc[test_index]
8          y_train, y_test = y.loc[train_index], y.loc[test_index]
9          model = MLP(**parameter).fit(X_train, y_train)
10         y_pred = model.predict(X_test)
11         score = accuracy_score(y_test, y_pred)
12         score_list.append(score)
13         if score < best_score * 0.95:
14             break
15
16     mean_score = np.mean(score_list)
17     if mean_score > best_score:
18         best_score = mean_score
19         best_parameter = parameter
20
21 end_time = time.time()
22 print(best_parameter, best_score)
23 print(int(end_time - start_time))
```

- **라인 1:** 평균을 계산하기 위해 numpy 모듈을 불러옵니다.
- **라인 11~12:** 폴드를 평가한 점수를 score에 저장하고 score_list에 추가합니다.
- **라인 13~14:** 현재 폴드를 평가한 점수가 best_score의 95%보다 작으면 조기 종료를 합니다.

```
{'early_stopping': False, 'hidden_layer_sizes': (10, 10, 10), 'max_iter': 1000, 'random_state': 2022}
0.5885050940438872
```

76

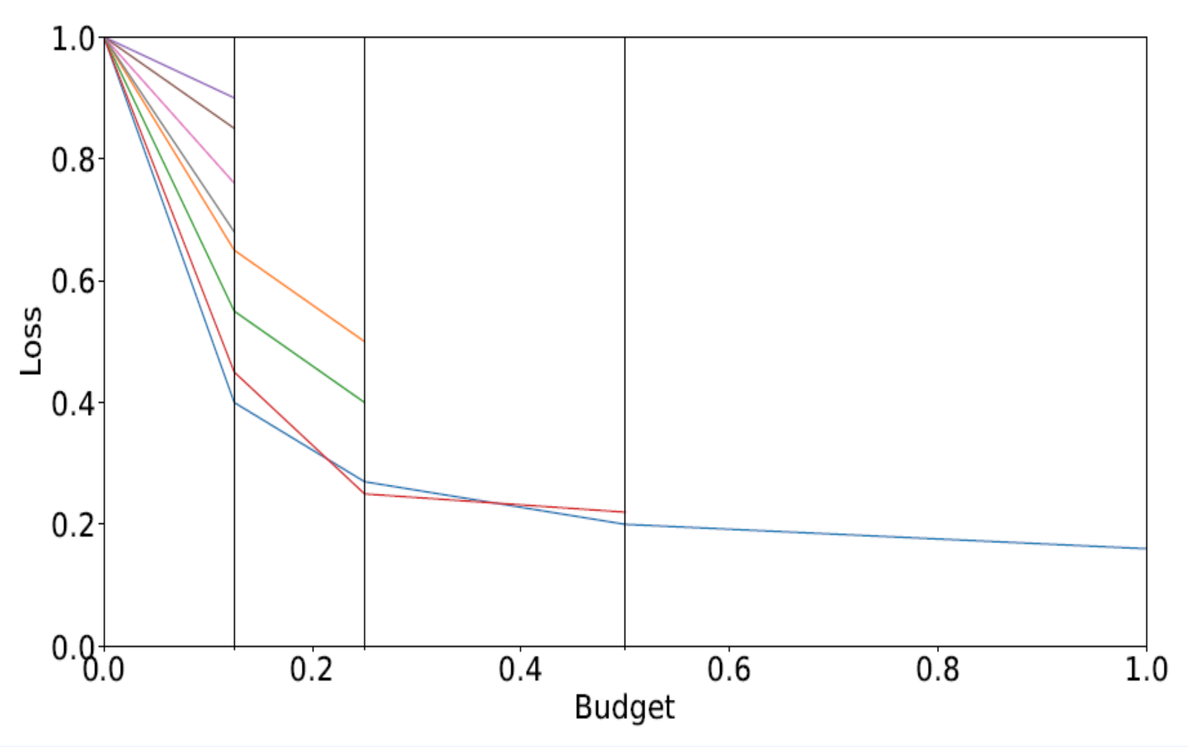
- 누적 평균을 사용했을 때보다 k번째 폴드의 정확도를 사용했을 때 더 빠르게 종료됨
- 그 이유는 누적 평균 정확도는 값이 안정적이지만, 각 폴드의 정확도는 변동이 클 수 있어 조기 종료 조건을 만족할 가능성이 크기 때문
- 그러나 특정 폴드에서만 나쁜 해를 놓칠 위험이 존재함

1. 속도 향상을 위한 테크닉

2 다중 충실도

다중 충실도란?

다중 충실도는 원 데이터를 샘플링한 데이터로 해를 평가하고 좋은 해에 대해서만 추가로 샘플링한 데이터로 평가하는 것을 반복합니다.



- 각 선은 해를 나타냄
- 초기에 손실이 컸던 해는 다음 이터레이션에서 평가에 제외되고 그 만큼 사용하는 데이터(budget)이 커짐

그림 출처:Zöller, M. A., & Huber, M. F. (2021). Benchmark and survey of automated machine learning frameworks. *Journal of artificial intelligence research*, 70, 409-472.

코드 실습

```
1  from scipy.stats import rankdata
2  start_time = time.time()
3
4  s_param_idx_list = np.arange(len(grid)).astype(int)
5  for i in range(5):
6      score_list = []
7      for s_param_idx in s_param_idx_list:
8          s_parameter = grid[s_param_idx]
9          s_idx = np.random.choice(X.index,
10                                  int(len(X) / len(s_param_idx_list)),
11                                  replace = False)
12
13         s_X = X.loc[s_idx]
14         s_y = y.loc[s_idx]
15
16         score = 0
17         for train_index, test_index in kf.split(s_X):
18             s_X_train, s_X_test = s_X.iloc[train_index], s_X.iloc[test_index]
19             s_y_train, s_y_test = s_y.iloc[train_index], s_y.iloc[test_index]
20             model = MLP(**s_parameter).fit(s_X_train, s_y_train)
21             s_y_pred = model.predict(s_X_test)
22             score += accuracy_score(s_y_test, s_y_pred) / 5
23         score_list.append(score)
24     score_list = np.array(score_list)
25     s_param_idx_list = rankdata(-score_list)[:int(len(s_param_idx_list) * 0.8)] - 1
26     s_param_idx_list = s_param_idx_list.astype(int)
27
28 end_time = time.time()
29 print(best_parameter, best_score)
30 print(int(end_time - start_time))
```

- **라인 4:** grid에서 추가로 탐색할 파라미터의 인덱스를 s_param_idx_list에 저장합니다. 초기에는 모든 파라미터가 탐색 대상입니다.
- **라인 5:** 총 5번의 이터레이션을 반복함으로써 각 이터레이션마다 하위 20%까지의 파라미터를 추가로 탐색하지 않습니다.
- **라인 8:** s_param_idx를 이용해 현재 이터레이션에서 평가할 파라미터를 s_parameter에 저장합니다.
- **라인 9~11:** 현재 이터레이션에서 사용할 데이터를 샘플링하기 위한 인덱스 s_idx를 생성합니다. s_idx는 데이터의 크기를 평가할 파라미터 개수로 나눈 것과 같은 크기의 인덱스 배열입니다.
- **라인 13~14:** 라인 11~13에서 샘플링한 인덱스로 X와 y를 각각 필터링하여 s_X와 s_y에 저장합니다.
- **라인 23:** 각 파라미터에 대한 평가 점수인 score를 score_list에 추가합니다.
- **라인 25:** rankdata 함수를 사용해 상위 80%까지의 파라미터 인덱스를 s_param_idx_list에 다시 저장합니다. 이때 rankdata는 값이 가장 작으면 1을, 그다음으로 작으면 2를 반환하므로 (-score_list)의 순위를 구하여 값이 클수록 높은 순위가 되게 했으며 인덱스로 사용할 수 있게 1을 뺐습니다.

{'early_stopping': False, 'hidden_layer_sizes': (10, 10, 10), 'max_iter': 1000, 'random_state': 2022} 0.5885050940438872

- 이전과 같은 파라미터를 찾았으나 탐색 시간을 2배 가까이 줄였음
- 다른 폴드에서의 평가 점수는 우수하지만, 유독 한 폴드의 평가 점수만 나쁜 파라미터가 버려질 수 있음에 주의해야 함