2022년 2학기 기초인공지능(CSE4185) 과제 4 보고서

학번: 20171665 | 이름: 이선호

문제 3

3번 문제를 해결하기 위해 본인이 시도한 방법

생성자 `__init__`의 파라미터로 들어오는 `out_feat` list에 MLP의 output 차원이 들어오므로, 이 list의 원소를 하나씩 보면서 `layers` list에 차례대로 넣어주는 작업이 요구된다. 그래서 `for` loop으로 `out_feat_list`에서 원소에 하나씩 접근하여 `layers`의 `append` 메소드로 layer를 넣어준다. 대신에 output 뿐만 아니라 layer에 들어오는 input 차원이 필요하므로 `in_feat` 변수를 선언했다.

첫 번째 layer에서는 문제에서 주어진 $1,024$ 차원을 input으로 받아야 하므로 변수의 값을 $1,024$로 초기화한다.

```
1   in_feat = 1024
```

그 외의 layer에서는 직전 layer의 output 차원을 input 차원으로 받을 수 있도록 매 iteration 마다 output 차원을 `in_feat`에 할당한다. 사용 가능한 layer는 `nn.Linear`이므로 아래처럼 layer를 생성하여 `layers`에 원소로 추가한다.

```
1   layers.append(nn.Linear(in_feat, out_feat))
```

파라미터로 들어오는 `act` 문자열을 통해 어떠한 activation function을 사용할지를 알 수 있으므로 조건문을 사용하여 각각의 layer 바로 뒤에 적합한 activation function이 오도록 `layers`에 원소로 삽입한다. Sigmoid 함수일 때는 `nn.Sigmoid()`, ReLU 함수는 `nn.ReLU()`, Tanh 함수일 경우 `nn.Tanh()`를 `layers`의 `append` 메소드로 `layers`에 삽입한다. 이를 구현하면 다음과 같이 쓸 수 있다.

```
if act == "sigmoid":
    layers.append(nn.Sigmoid())
elif act == "relu":
    layers.append(nn.ReLU())
elif act == "tanh":
    layers.append(nn.Tanh())
```

다음 layer의 input 차원은 현재 layer의 output 차원과 같으므로 다음 layer 생성 시 `in_feat`을 input 차원으로 사용할 수 있도록 `in_feat`에 현재 `out_feat`을 할당한다.

```
in_feat = out_feat
```

위의 내용을 종합하여 `for` loop으로 반복하여 `CustomMLP`의 생성자를 구현했다.

문제 4

> 4번 문제를 해결하기 위해 본인이 시도한 방법. 가장 성능이 높았던 방법에 대해서 소개. 최종적인 test 성능

사실 가장 성능이 높았던 방법은 기존의 코드와 크게 다른 부분이 있지 않다. `CustomMLP`를 모델로 생성할 때 `out_feat_list` 파라미터로 기존의 길이 5의 list가 아니라 32를 output으로 내보내도록 네 번째 위치에 레이어를 추가한 것밖에 없다. 다양한 hyperparameter tuning을 시도했으나 기존에 주어진 코드의 generalized performance가 좋은 편이어서 큰 수정을 거치지 않고도 40%의 test accuracy를 통과했다.

Activation function을 ReLU에서 다른 종류의 함수로 변경하면 오히려 성능이 더 하락했다. 이는 ReLU가 Sigmoid나 Tanh보다 backpropagation에서 발생할 수 있는 gradient vanishing 문제에서 상대적으로 더 자유롭기 때문이라고 해석했다. ReLU와 유사한 Swish 함수 사용을 시도해보았으나 test accuracy에 유의미한 차이가 발생하지 않았다.

Learning rate을 0.005로 늘려보았으나 오히려 test accuracy가 10%로 나왔고, 0.0005로 줄여보았으나 마찬가지로 test 결과가 좋지 않았다. Learning rate을 너무 크게 잡으면 local optima를 빠져나올 가능성은 높아지지만 global optima에 다가가는 이동 폭이 커서 잘 수렴이 되지 않을 수 있고, 반면에 learning rate을 너무 작게 잡으면 수렴은 잘 될 수 있으나 local optima에 빠졌을 때 이를 빠져 나오기가 쉽지 않을 수 있고 local optima에 수렴하는 속도가 너무 느릴 수 있다고 추정했다.

Layer의 수를 조절하는 방법을 시도했는데, 6개의 layer보다 더 많아지면 과적합이 발생해서인지 train 결과는 50%를 상회했으나 test 결과에서 30% 후반대로 accuracy가 나왔다. 반대로 layer의 수를 줄이면 train 결과에서부터 성능이 좋지 않게 나와서 파라미터 수가 충분치 않아 underfitting이 발생한 것으로 해석했다.

Seed를 100에서 125로도 바꿔봤으나 오히려 성능이 하락하여 seed를 변경하는 방법은 black box적인 면이 없지 않아 있어서 성능 향상이 쉽지 않을 것으로 판단하여 100에서 더 이상 seed를 바꾸는 시도를 해보지 않았다.

Optimizer도 SGD(Stochastic Gradient Descent)와 momentum을 사용해 보았지만 성능 향상을 보이지 못했다. Adam optimizer가 momentum과 RMSProp의 장점을 동시에 가져가면서 일반적으로 자주 사용되고, 성능도 좋다고 알려져 있어서 optimizer를 수정하는 방법으로는 성능 향상의 결과를 보이지 못했다.

결론적으로 test accuracy를 40% 상회하는 가장 좋은 결과는 `out_feat_list`를 `[1024, 256, 64, 16, 10]`에서 `[1024, 256, 64, 32, 16, 10]`으로 변경하는 방법이었으며, 최종적인 test 성능은 42.5%로 나왔다.