기초 인공지능(CSE4185) Assignment #7

2분반 20200183 박정원

1. 3번 문제를 해결하기 위해 본인이 시도한 방법

```python
class CustomMLP(nn.Module):
  def __init__(self, num_layers: int, out_feat_list: List[int], act: str):
    assert len(out_feat_list)==num_layers, 'out_feat_list size should be equal to num_layers'
    super(CustomMLP, self).__init__()
    activation = self.select_act(act)
    layers = list()
    ######################## TODO ########################
    # (linear transformation + activation function / 총 두 개) * num_layers 만큼 리스트에 원소 미리 추가
    for _ in range(num_layers*2):
      layers.append(0)
    # input_data 행렬의 column은 1024이므로, 첫번째 레이어는 직접 추가
    layers[0] = nn.Linear(1024, out_feat_list[0])
    layers[1] = activation
    # for 문을 통해서 반복적으로 linear transformation + activation function 순으로 추가한다.
    for i in range(1, num_layers):
      layers[2*i] = nn.Linear(out_feat_list[i-1], out_feat_list[i])
      layers[2*i+1] = activation
    ######################################################
    self.mlp = nn.Sequential(*layers)
```

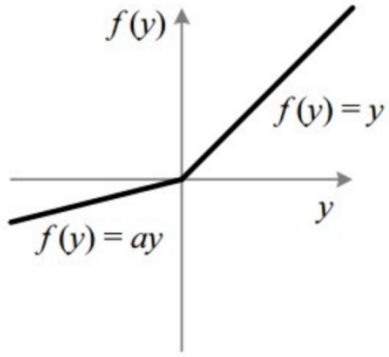
먼저, layers에 레이어들의 개수에 맞게 0이라는 초기화 값을 미리 append 해준다. (인덱싱으로 접근하기 위해서) 이 때, 레이어들의 개수에는 activation function 까지 포함해서 넣어준다. 만약 out_feat_list가 n개면 총 2 * n개의 원소를 갖게 될 것이다. 그 이유는 Linear 뒤에 activation 함수가 따라오기 때문이다. out_feat_list에는 히든 레이어들의 column 개수만 있으므로, 첫번째 층은 직접 초기화해주었다. input_data의 column 개수를 1024로 가정하고 있으므로, 1024와 out_feat_list[0]으로 첫번째 레이어를 layers[0]에 넣어준다. 그 이후 layers[1]은 activation을 넣어준다.

이 후에는 반복문을 통해서 쉽게 MLP의 layers를 구성할 수 있는데, 1 인덱스부터(첫번째 레이어는 이미 구성했으므로) 시작하여, input으로 들어온 num_layers 전까지 반복하는데, 2*i에는 nn.Linear(out_feat_list[i-1], out_feat_list[i])로, 2*i+1에는 활성 함수를 넣어준다. 이렇게 구성하면 성공적으로 MLP의 레이어들을 구성할 수 있다.

```python
def forward(self, x):
  x = self.mlp(x)
  return x

def select_act(self, act: str):
  assert act in ['sigmoid', 'relu', 'tanh', 'leakyrelu'], 'activation should be in [sigmoid, relu, tanh]'
  if act == 'sigmoid':
    return nn.Sigmoid()
  elif act == 'relu':
    return nn.ReLU()
  elif act == 'leakyrelu': # add new activation function 'leaky relu'
    return nn.LeakyReLU()
  else:
    return nn.Tanh()
```

4번 문제에서 성능을 높이기 위해서 여러 가지 활성 함수들을 시도하다가, 수업 시간에 leaky relu라는 활성 함수가 있다는 것을 들었는데, 파이토치에 찾아보니 LeakyReLU로 지원하고 있어서 select_act 함수를 수정하였다.

leaky relu의 개형 (파이토치의 nn.LeakyReLU()의 경우 디폴트로 음수 구간에 대해 0.01의 기울기를 쓴다.)




```
CustomMLP(
  (mlp): Sequential(
    (0): Linear(in_features=1024, out_features=6, bias=True)
    (1): Sigmoid()
    (2): Linear(in_features=6, out_features=8, bias=True)
    (3): Sigmoid()
    (4): Linear(in_features=8, out_features=3, bias=True)
    (5): Sigmoid()
    (6): Linear(in_features=3, out_features=4, bias=True)
    (7): Sigmoid()
    (8): Linear(in_features=4, out_features=9, bias=True)
    (9): Sigmoid()
    (10): Linear(in_features=9, out_features=10, bias=True)
    (11): Sigmoid()
    (12): Linear(in_features=10, out_features=2, bias=True)
    (13): Sigmoid()
    (14): Linear(in_features=2, out_features=4, bias=True)
    (15): Sigmoid()
  )
)
```

```
CustomMLP(
  (mlp): Sequential(
    (0): Linear(in_features=1024, out_features=5, bias=True)
    (1): ReLU()
    (2): Linear(in_features=5, out_features=8, bias=True)
    (3): ReLU()
    (4): Linear(in_features=8, out_features=13, bias=True)
    (5): ReLU()
    (6): Linear(in_features=13, out_features=6, bias=True)
    (7): ReLU()
    (8): Linear(in_features=6, out_features=4, bias=True)
    (9): ReLU()
    (10): Linear(in_features=4, out_features=9, bias=True)
    (11): ReLU()
    (12): Linear(in_features=9, out_features=9, bias=True)
    (13): ReLU()
    (14): Linear(in_features=9, out_features=2, bias=True)
    (15): ReLU()
    (16): Linear(in_features=2, out_features=6, bias=True)
    (17): ReLU()
    (18): Linear(in_features=6, out_features=3, bias=True)
    (19): ReLU()
    (20): Linear(in_features=3, out_features=9, bias=True)
    (21): ReLU()
    (22): Linear(in_features=9, out_features=10, bias=True)
    (23): ReLU()
  )
)
```

2. 4번 문제를 해결하기 위해 본인이 시도한 방법, 가장 성능이 높았던 방법에 대한 소개,

최종 test 성능

좋은 성능을 뽑아내기 위해서 조정했던 변수는 시드와 활성 함수, 그리고 학습률이다. 시드는 50으로 고정하고 시작하였다.

```
[24] ##################################### TODO #####################################
     SEED = 50 # can be changed
     LEARNING_RATE = 0.001 # can be changed
     torch.random.manual_seed(SEED)
     model = CustomMLP(num_layers=8, out_feat_list=[1024,256,64,16,10, 128, 256, 512], act='relu').to(device) #
     optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE) # can be changed
```

먼저, 같은 레이어 개수와 out_feat_list를 동일한 조건으로 맞춰놓고, 활성 함수를 각각 테스트하였다. 레이어 개수는 기본으로 주어진 개수가 너무 적다고 생각되어, 8개로 늘려주었다. 추가한 column의 개수는, 128, 256, 512 이다. 이렇게 개수를 점차 늘려준 이유는 column의 개수가 너무 줄어들어서 특징들이 압축되었다고 생각했기 때문이다. 또한, 레이어를 좀 더 깊게 만들어준 이유는 층을 깊게 하는 것이 정보를 계층적으로 전달하게 하여 효율적으로 학습을 하는 것에 도움을 주기 때문이다. 즉, 한 층에서 처리해야 하는 이미지의 특징을 배분하여 각 층에서 맡게 되는 정보를 구체적으로 세분화시키는 것이다. 예를 들어, 첫번째 층이 이미지의 테두리를 인식하고, 다음 층은 이미지의 텍스쳐 정보를 인식하게 되면 한 층에서 한번에 처리하는 것보다 더 효율적일 것이다.

학습률 0.001 부터 시작했다.

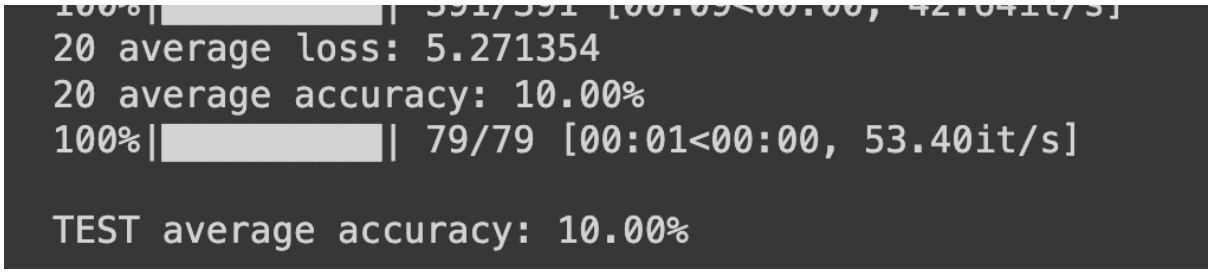
먼저, relu 부터 테스트해보았다.

```
19 average accuracy: 46.82%
100%|██████████| 391/391 [00:09<00:00, 41.19it/s]
20 average loss: 1.480501
20 average accuracy: 47.14%
100%|██████████| 79/79 [00:01<00:00, 52.48it/s]

TEST average accuracy: 42.32%
```

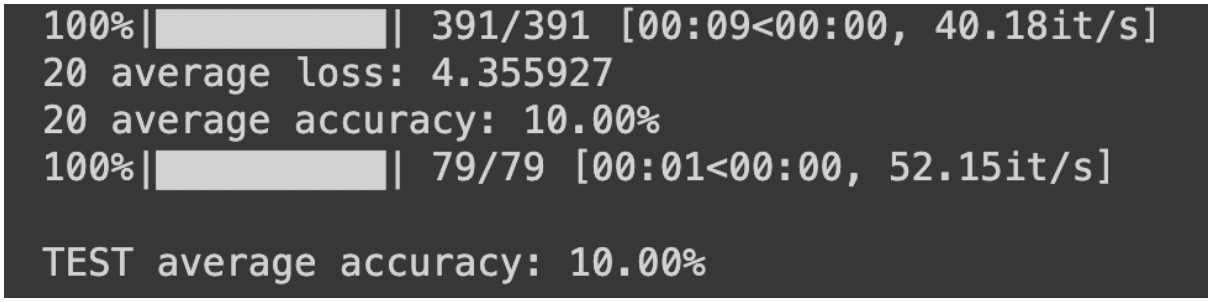
다음은, leaky relu이다.

```
19 average accuracy: 46.95%
100%|██████████| 391/391 [00:09<00:00, 41.01it/s]
20 average loss: 1.464397
20 average accuracy: 47.76%
100%|██████████| 79/79 [00:01<00:00, 49.54it/s]

TEST average accuracy: 42.14%
```

다음은 sigmoid 이다.



다음은 tanh 이다.



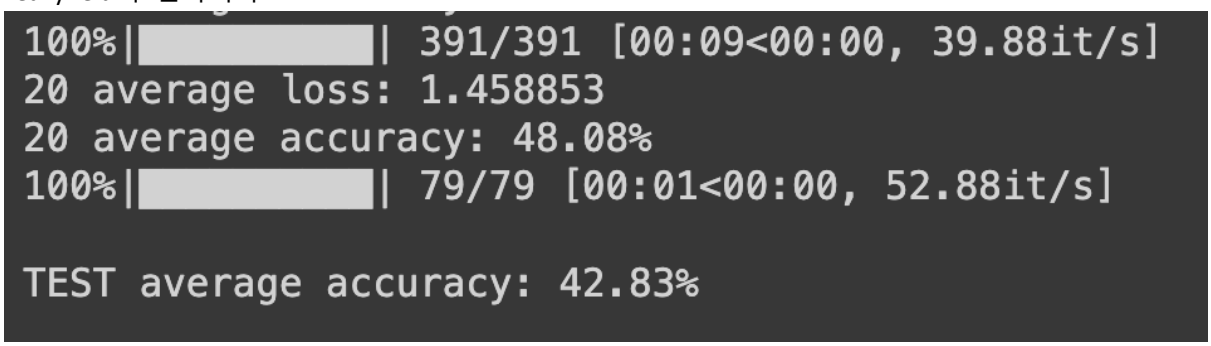
Sigmoid와 tanh는 테스트 데이터에 대한 학습률이 둘다 10프로로 동일했다. Leaky relu와 relu에 비해 너무 낮은 수치여서, 다음 테스트에서 활성 함수는 leaky relu와 relu 둘 중 하나로 사용하기로 했다.

다음은 학습률을 조정해보았다.
이 때 테스트한 활성 함수는 leaky relu, relu이다.

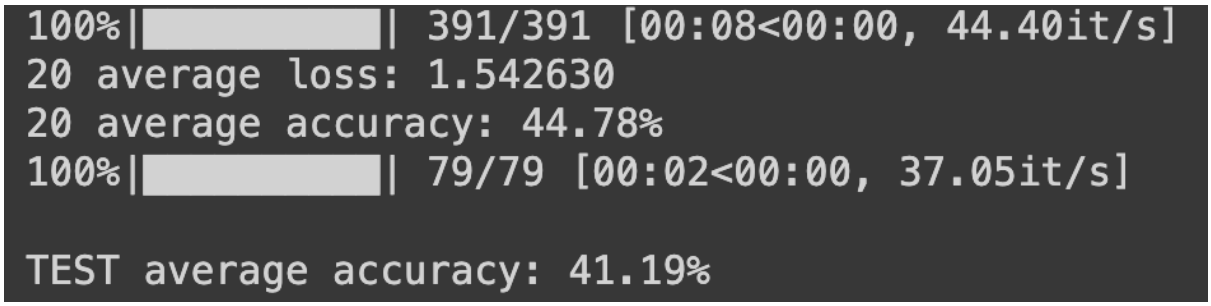
학습률을 처음에는 0.0001과 0.01, 즉 각각 1/10 배와 10배를 테스트해보았는데, 너무 학습률이 낮게 나와 조정할 때 조금만 수정하여, 0.0009와 0.0011 학습률에 대해 테스트해보았다.

먼저, 0.0011이다.

Leaky relu의 결과이다.

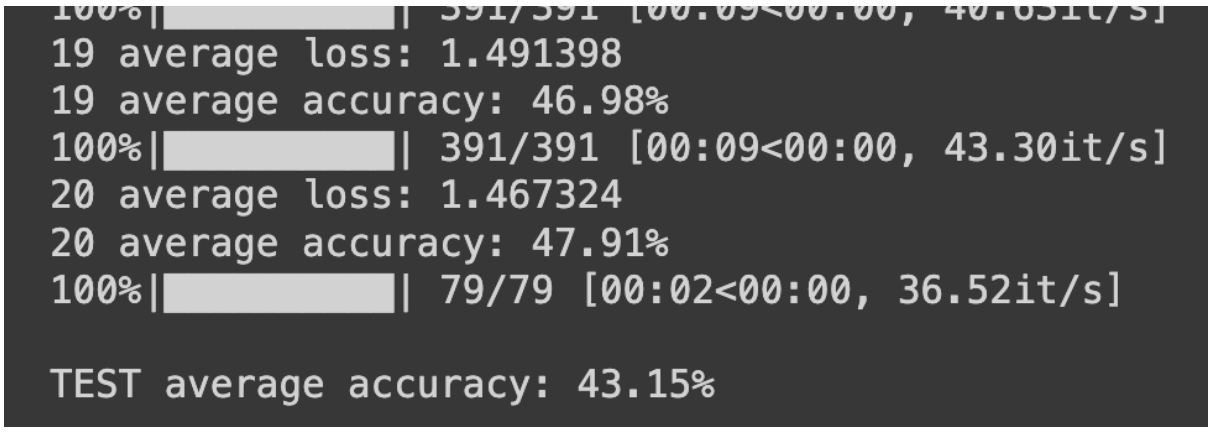


Relu 의 결과이다.

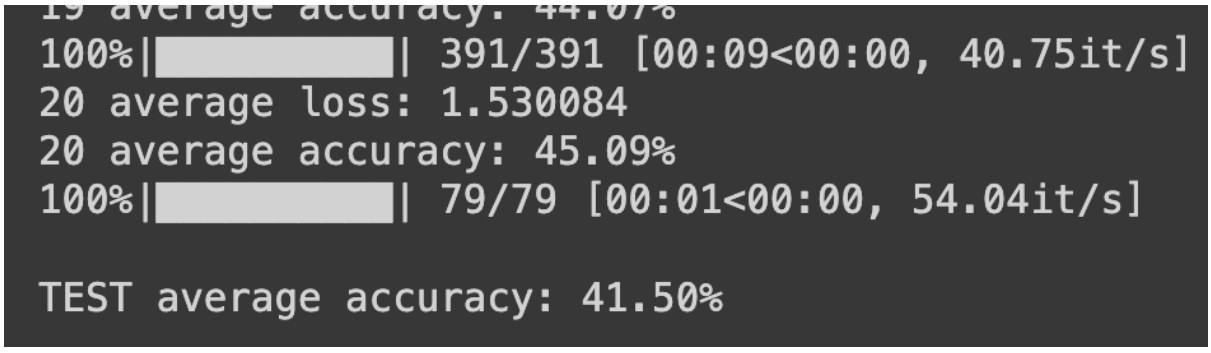


그 다음에는 0.0009이다.

먼저, leaky relu 결과이다.



다음은, relu이다.

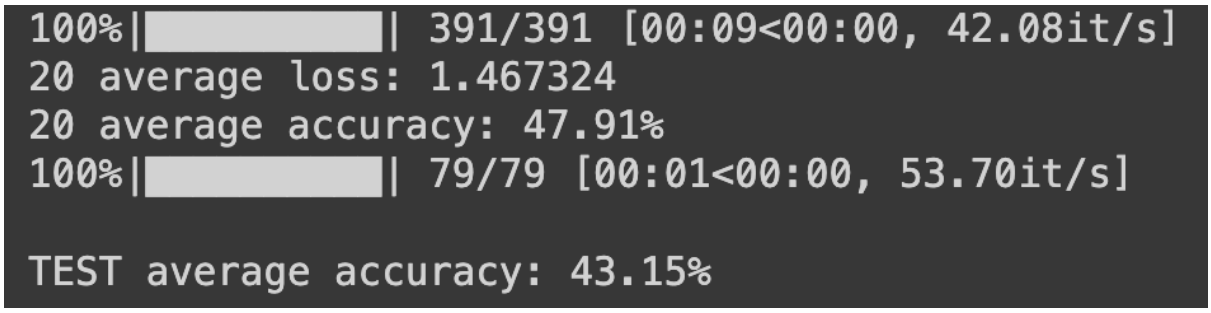


학습률 0.001일 때는 relu가 leaky relu보다 좋은 성능을 보였으나 학습률 0.0009와 0.0011에 대해서는 둘 다 leaky relu가 근소하지만 좋은 성능을 보였다.

결과적으로는 학습률 0.0009일 때, leaky relu가 테스트 데이터에 대해서 43.15퍼센트의 가장 높은 정확도를 보여서 최종적으로 이 파라미터 세팅을 사용하게 되었다.

```
################################### TODO ###################################
SEED = 50 # can be changed
LEARNING_RATE = 0.0009 # can be changed
torch.random.manual_seed(SEED)
model = CustomMLP(num_layers=8, out_feat_list=[1024,256,64,16,10, 128, 256, 512], act='leakyrelu').to(device) # can be changed (but meet conditions above)
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE) # can be changed
```

최종 test 성능


100%| | 391/391 [00:09<00:00, 42.08it/s]
20 average loss: 1.467324
20 average accuracy: 47.91%
100%| | 79/79 [00:01<00:00, 53.70it/s]
TEST average accuracy: 43.15%