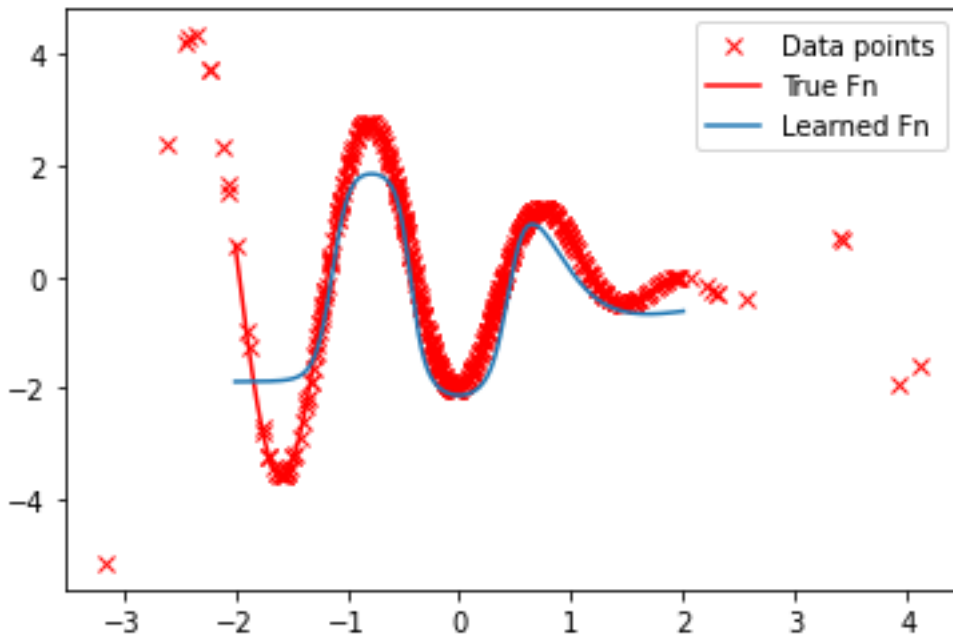


#1.

```
threelayerSGD.py x
1
2 import torch
3 import numpy as np
4 from torch import nn, optim
5 from torch.nn import functional as F
6 from torch.utils.data import TensorDataset, DataLoader
7 import matplotlib.pyplot as plt
8 from sklearn.model_selection import train_test_split
9
10 alpha = 0.1
11 K = 1000
12 B = 128
13 N = 512
14
15 def f_true(x):
16     return (x-2) * np.cos(x*4)
17
18 torch.manual_seed(0)
19 X_train = torch.normal(0.0, 1.0, (N,))
20 y_train = f_true(X_train)
21 X_val = torch.normal(0.0, 1.0, (N//5,))
22 y_val = f_true(X_val)
23
24 train_dataloader = DataLoader(TensorDataset(X_train.unsqueeze(1), y_train.unsqueeze(1)), batch_size=B)
25 test_dataloader = DataLoader(TensorDataset(X_val.unsqueeze(1), y_val.unsqueeze(1)), batch_size=B)
26
27 ...
28 unsqueeze(1) reshapes the data into dimension [N,1],
29 where 1 is the dimension of an data point.
30
31 The batchsize of the test dataloader should not affect the test result
32 so setting batch_size=N may simplify your code.
33 In practice, however, the batchsize for the training dataloader
34 is usually chosen to be as large as possible while not exceeding
35 the memory size of the GPU. In such cases, it is not possible to
36 use a larger batchsize for the test dataloader.
37 ...
38
39 class MLP(nn.Module):
40     def __init__(self):
41         super().__init__()
42         self.linear1 = nn.Linear(1, 64, bias=True)
43         self.linear2 = nn.Linear(64, 64, bias=True)
44         self.linear3 = nn.Linear(64, 1, bias=True)
45
46     def forward(self, x):
47         x = x.float().view(-1, 1)
48         x = nn.functional.sigmoid(self.linear1(x))
49         x = nn.functional.sigmoid(self.linear2(x))
50         x = (self.linear3(x))
51         return x
52
53 model = MLP()
54 loss_function = nn.MSELoss()
55
56 model.linear1.weight.data = torch.normal(0, 1, model.linear1.weight.shape)
57 model.linear1.bias.data = torch.full(model.linear1.bias.shape, 0.03)
58 model.linear2.weight.data = torch.normal(0, 1, model.linear2.weight.shape)
59 model.linear2.bias.data = torch.full(model.linear2.bias.shape, 0.03)
60 model.linear3.weight.data = torch.normal(0, 1, model.linear3.weight.shape)
61 model.linear3.bias.data = torch.full(model.linear3.bias.shape, 0.03)
62
63 optimizer = torch.optim.SGD(model.parameters(), lr = alpha)
64
65 for epoch in range(K):
66     for x, y in train_dataloader:
67         optimizer.zero_grad()
68         train_loss = loss_function(model(x), y)
69         train_loss.backward()
70         optimizer.step()
71
72 with torch.no_grad():
73     xx = torch.linspace(-2, 2, 1024).unsqueeze(1)
74     plt.plot(X_train, y_train, 'rx', label='Data points')
75     plt.plot(xx, f_true(xx), 'r', label='True Fn')
76     plt.plot(xx, model(xx), label='Learned Fn')
77     plt.legend()
78     plt.show()
79
80 ...
81 When plotting torch tensors, you want to work with the
82 torch.no_grad() context manager.
83
84 When you call plt.plot(...) the torch tensors are first converted into
85 numpy arrays and then the plotting proceeds.
86 However, our trainable model has requires_grad=True to allow automatic
87 gradient computation via backprop, and this option prevents
88 converting the torch tensor output by the model to a numpy array.
89 Using the torch.no_grad() context manager resolves this problem
90 as all tensors are set to requires_grad=False within the context manager.
91
92 An alternative to using the context manager is to do
93 plt.plot(xx, model(xx).detach().clone())
94 The .detach().clone() operation create a copied pytorch tensor that
95 has requires_grad=False.
96
97 To be more precise, .detach() creates another tensor with requires_grad=False
98 (it is detached from the computation graph) but this tensor shares the same
99 underlying data with the original tensor. Therefore, this is not a genuine
100 copy (not a deep copy) and modifying the detached tensor will affect the
101 original tensor in weird ways. The .clone() further proceeds to create a
102 genuine copy of the detached tensor, and one can freely manipulate and change it.
103 (For the purposes of plotting, it is fine to just call .detach() without
104 .clone() since plotting does not change the tensor.)
105
106 This discussion will likely not make sense to most students at this point of the course.
107 We will revisit this issue after we cover backpropagation.
108
109
110
111
112
113
114
115
```



Pytorch로 training을 해보면 위와 같은 결과를 얻을 수 있다.

#2.

먼저 parameter의 개수를 계산해보자. 일단,

$64 \times 1 + 64 \times 64 + 64 \times 1$  개의 parameter가 존재하며, bias를 추가로 계산해보면

$64 + 64 + 1$  개가 추가로 존재함을 알 수 있다.

이는 총 4353개이다. ( $p > N!$ )

Y -train 에 noise 를 추가하고 동일한 실험을 반복하였다. 코드와 결과는 다음과 같았다. 실험 결과, 그래프의 개형이 비슷하게 나타났고, outlier 점에 의한 overfitting 현상도 오히려 감소한 것과 같은 모습이 보여졌다.

```

3 import numpy as np
4 from torch import nn, optim
5 from torch.nn import functional as F
6 from torch.utils.data import TensorDataset, DataLoader
7 import matplotlib.pyplot as plt
8 from sklearn.model_selection import train_test_split
9
10 alpha = 0.1
11 K = 1000
12 B = 128
13 N = 512
14
15 def f_true(x):
16     return (x-2) * np.cos(x**4)
17
18 torch.manual_seed(0)
19 X_train = torch.normal(0.0, 1.0, (N,))
20 y_train = f_true(X_train)
21 y_train = f_true(X_train) + torch.normal(0, 0.5, X_train.shape)
22 X_val = torch.normal(0.0, 1.0, (N/5,))
23 y_val = f_true(X_val)
24
25 train_dataloader = DataLoader(TensorDataset(X_train.unsqueeze(1), y_train.unsqueeze(1)), batch_size=B)
26 test_dataloader = DataLoader(TensorDataset(X_val.unsqueeze(1), y_val.unsqueeze(1)), batch_size=B)
27
28 '''
29 unsqueeze(1) reshapes the data into dimension [N,1],
30 where 1 is the dimension of an data point.
31
32 The batchsize of the test dataloader should not affect the test result
33 so setting batch_size=N may simplify your code.
34 In practice, however, the batchsize for the training dataloader
35 is usually chosen to be as large as possible while not exceeding
36 the memory size of the GPU. In such cases, it is not possible to
37 use a larger batchsize for the test dataloader.
38 '''
39
40 class MLP(nn.Module):
41     def __init__(self):
42         super().__init__()
43         self.linear1 = nn.Linear(1, 64, bias=True)
44         self.linear2 = nn.Linear(64, 64, bias=True)
45         self.linear3 = nn.Linear(64, 1, bias=True)
46
47     def forward(self, x):
48         x = x.float().view(-1, 1)
49         x = nn.functional.sigmoid(self.linear1(x))
50         x = nn.functional.sigmoid(self.linear2(x))
51         x = (self.linear3(x))
52
53         return x
54
55 model = MLP()
56 loss_function = nn.MSELoss()
57
58 model.linear1.weight.data = torch.normal(0, 1, model.linear1.weight.shape)
59 model.linear1.bias.data = torch.full(model.linear1.bias.shape, 0.03)
60 model.linear2.weight.data = torch.normal(0, 1, model.linear2.weight.shape)
61 model.linear2.bias.data = torch.full(model.linear2.bias.shape, 0.03)
62 model.linear3.weight.data = torch.normal(0, 1, model.linear3.weight.shape)
63 model.linear3.bias.data = torch.full(model.linear3.bias.shape, 0.03)
64
65 optimizer = torch.optim.SGD(model.parameters(), lr = alpha)
66
67 for epoch in range(K):
68     for x, y in train_dataloader:
69         optimizer.zero_grad()
70         train_loss = loss_function(model(x), y)
71         train_loss.backward()
72         optimizer.step()
73
74 with torch.no_grad():
75     xx = torch.linspace(-2,2,1024).unsqueeze(1)
76     plt.plot(X_train, y_train, 'rx', label='Data points')
77     plt.plot(xx, f_true(xx), 'r', label='True Fn')
78     plt.plot(xx, model(xx), label='Learned Fn')
79     plt.legend()
80     plt.show()
81
82 '''
83 When plotting torch tensors, you want to work with the
84 torch.no_grad() context manager.
85
86 When you call plt.plot(...) the torch tensors are first converted into
87 numpy arrays and then the plotting proceeds.
88 However, our trainable model has requires_grad=True to allow automatic
89 gradient computation via backprop, and this option prevents
90 converting the torch tensor output by the model to a numpy array.
91 Using the torch.no_grad() context manager resolves this problem
92 as all tensors are set to requires_grad=False within the context manager.
93
94 An alternative to using the context manager is to do
95 plt.plot(xx, model(xx).detach().clone())
96 The .detach().clone() operation create a copied pytorch tensor that
97 has requires_grad=False.
98
99 To be more precise, .detach() creates another tensor with requires_grad=False
100 (it is detached from the computation graph) but this tensor shares the same
101 underlying data with the original tensor. Therefore, this is not a genuine
102 copy (not a deep copy) and modifying the detached tensor will affect the
103 original tensor in weird ways. The .clone() further proceeds to create a
104 genuine copy of the detached tensor, and one can freely manipulate and change it.
105 (For the purposes of plotting, it is fine to just call .detach() without
106 .clone() since plotting does not change the tensor.)
107
108 This discussion will likely not make sense to most students at this point of the course.
109 We will revisit this issue after we cover backpropagation.
110
111
112
113
114
115
116

```

