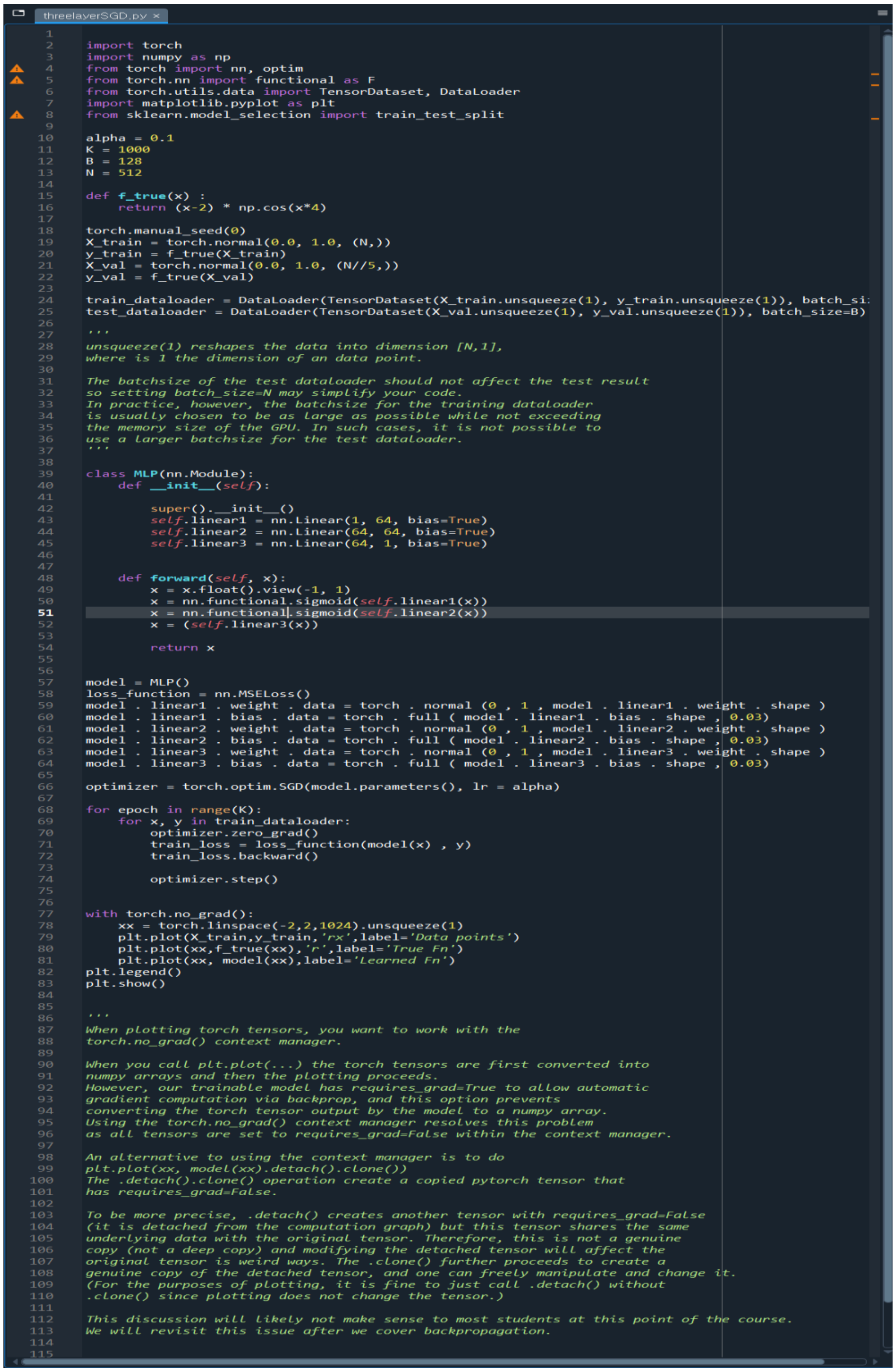
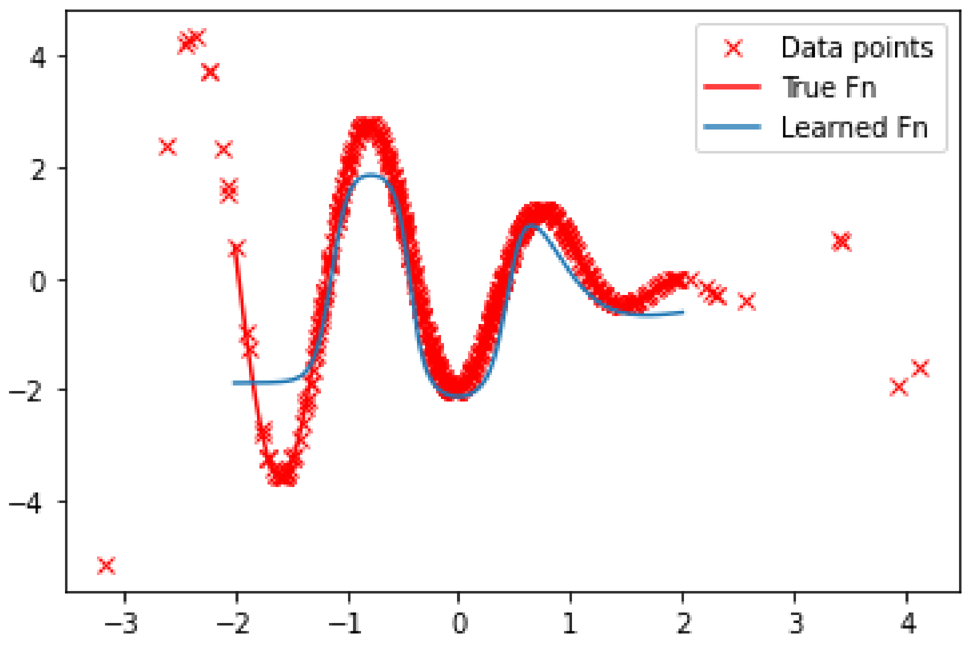
#1.


threelayerSGD.py ×

```python

import torch
import numpy as np
from torch import nn, optim
from torch.nn import functional as F
from torch.utils.data import TensorDataset, DataLoader
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

alpha = 0.1
K = 1000
B = 128
N = 512

def f_true(x) :
    return (x-2) * np.cos(x*4)

torch.manual_seed(0)
X_train = torch.normal(0.0, 1.0, (N,))
y_train = f_true(X_train)
X_val = torch.normal(0.0, 1.0, (N//5,))
y_val = f_true(X_val)

train_dataloader = DataLoader(TensorDataset(X_train.unsqueeze(1), y_train.unsqueeze(1)), batch_si
test_dataloader = DataLoader(TensorDataset(X_val.unsqueeze(1), y_val.unsqueeze(1)), batch_size=B)

'''
unsqueeze(1) reshapes the data into dimension [N,1],
where is 1 the dimension of an data point.

The batchsize of the test dataloader should not affect the test result
so setting batch_size=N may simplify your code.
In practice, however, the batchsize for the training dataloader
is usually chosen to be as large as possible while not exceeding
the memory size of the GPU. In such cases, it is not possible to
use a larger batchsize for the test dataloader.
'''

class MLP(nn.Module):
    def __init__(self):

        super().__init__()
        self.linear1 = nn.Linear(1, 64, bias=True)
        self.linear2 = nn.Linear(64, 64, bias=True)
        self.linear3 = nn.Linear(64, 1, bias=True)


    def forward(self, x):
        x = x.float().view(-1, 1)
        x = nn.functional.sigmoid(self.linear1(x))
        x = nn.functional.sigmoid(self.linear2(x))
        x = (self.linear3(x))

        return x


model = MLP()
loss_function = nn.MSELoss()
model . linear1 . weight . data = torch . normal (0 , 1 , model . linear1 . weight . shape )
model . linear1 . bias . data = torch . full ( model . linear1 . bias . shape , 0.03)
model . linear2 . weight . data = torch . normal (0 , 1 , model . linear2 . weight . shape )
model . linear2 . bias . data = torch . full ( model . linear2 . bias . shape , 0.03)
model . linear3 . weight . data = torch . normal (0 , 1 , model . linear3 . weight . shape )
model . linear3 . bias . data = torch . full ( model . linear3 . bias . shape , 0.03)

optimizer = torch.optim.SGD(model.parameters(), lr = alpha)

for epoch in range(K):
    for x, y in train_dataloader:
        optimizer.zero_grad()
        train_loss = loss_function(model(x) , y)
        train_loss.backward()

        optimizer.step()


with torch.no_grad():
    xx = torch.linspace(-2,2,1024).unsqueeze(1)
    plt.plot(X_train,y_train,'rx',label='Data points')
    plt.plot(xx,f_true(xx),'r',label='True Fn')
    plt.plot(xx, model(xx),label='Learned Fn')
plt.legend()
plt.show()


'''
When plotting torch tensors, you want to work with the
torch.no_grad() context manager.

When you call plt.plot(...) the torch tensors are first converted into
numpy arrays and then the plotting proceeds.
However, our trainable model has requires_grad=True to allow automatic
gradient computation via backprop, and this option prevents
converting the torch tensor output by the model to a numpy array.
Using the torch.no_grad() context manager resolves this problem
as all tensors are set to requires_grad=False within the context manager.

An alternative to using the context manager is to do
plt.plot(xx, model(xx).detach().clone())
The .detach().clone() operation create a copied pytorch tensor that
has requires_grad=False.

To be more precise, .detach() creates another tensor with requires_grad=False
(it is detached from the computation graph) but this tensor shares the same
underlying data with the original tensor. Therefore, this is not a genuine
copy (not a deep copy) and modifying the detached tensor will affect the
original tensor is weird ways. The .clone() further proceeds to create a
genuine copy of the detached tensor, and one can freely manipulate and change it.
(For the purposes of plotting, it is fine to just call .detach() without
.clone() since plotting does not change the tensor.)

This discussion will likely not make sense to most students at this point of the course.
We will revisit this issue after we cover backpropagation.


```



Pytorch로 training을 해보면 위와 같은 결과를 얻을 수 있다.

#2.

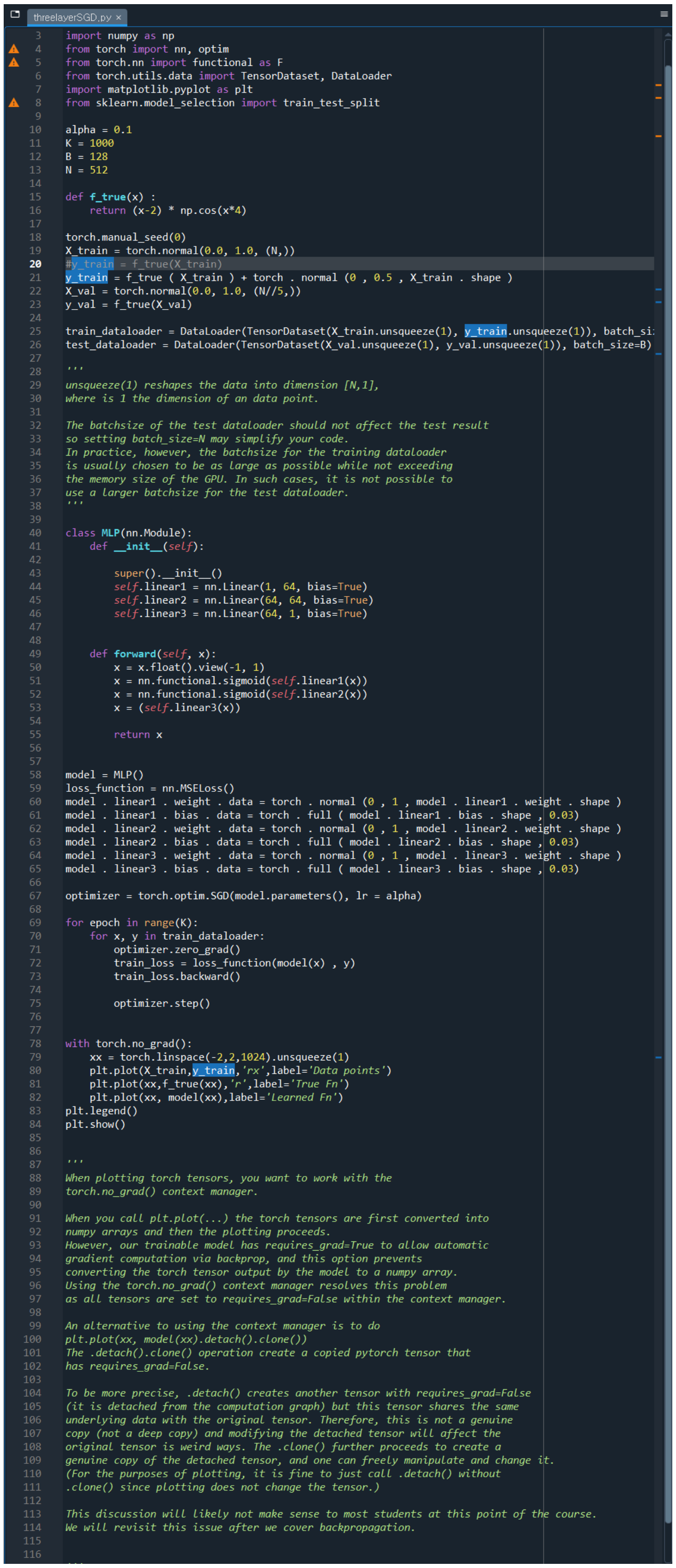
먼저 parameter의 개수를 계산해보자. 일단,

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

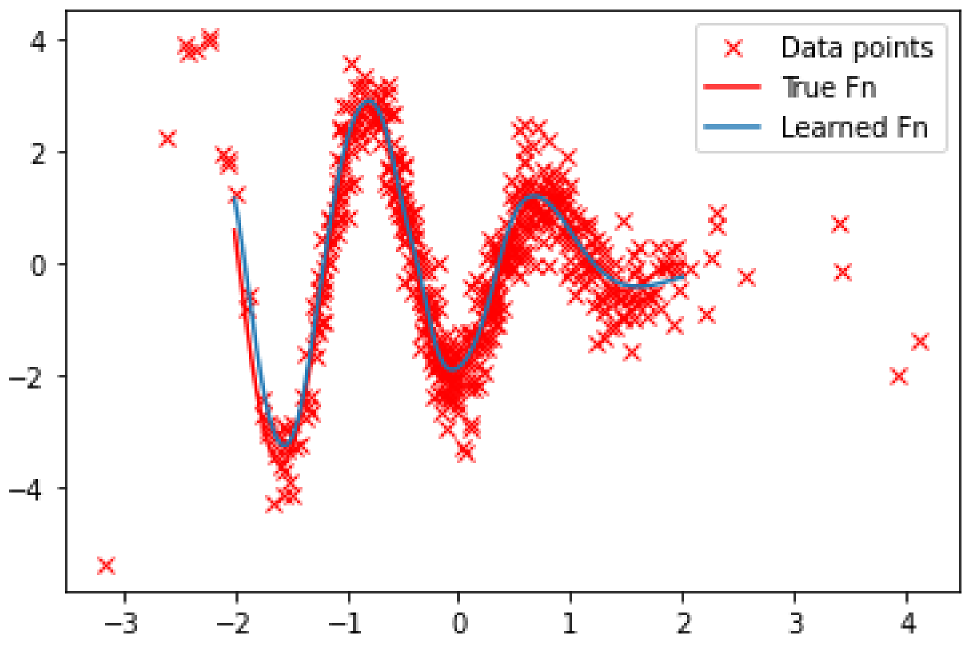
$64 + 64 + 1$ 개가 추가로 존재함을 알 수 있다.

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.


threelayerSGD.py

```python
import numpy as np
from torch import nn, optim
from torch.nn import functional as F
from torch.utils.data import TensorDataset, DataLoader
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

alpha = 0.1
K = 1000
B = 128
N = 512

def f_true(x) :
    return (x-2) * np.cos(x*4)

torch.manual_seed(0)
X_train = torch.normal(0.0, 1.0, (N,))
#y_train = f_true(X_train)
y_train = f_true ( X_train ) + torch . normal (0 , 0.5 , X_train . shape )
X_val = torch.normal(0.0, 1.0, (N//5,))
y_val = f_true(X_val)

train_dataloader = DataLoader(TensorDataset(X_train.unsqueeze(1), y_train.unsqueeze(1)), batch_si
test_dataloader = DataLoader(TensorDataset(X_val.unsqueeze(1), y_val.unsqueeze(1)), batch_size=B)

'''
unsqueeze(1) reshapes the data into dimension [N,1],
where is 1 the dimension of an data point.

The batchsize of the test dataloader should not affect the test result
so setting batch_size=N may simplify your code.
In practice, however, the batchsize for the training dataloader
is usually chosen to be as large as possible while not exceeding
the memory size of the GPU. In such cases, it is not possible to
use a larger batchsize for the test dataloader.
'''

class MLP(nn.Module):
    def __init__(self):

        super().__init__()
        self.linear1 = nn.Linear(1, 64, bias=True)
        self.linear2 = nn.Linear(64, 64, bias=True)
        self.linear3 = nn.Linear(64, 1, bias=True)


    def forward(self, x):
        x = x.float().view(-1, 1)
        x = nn.functional.sigmoid(self.linear1(x))
        x = nn.functional.sigmoid(self.linear2(x))
        x = (self.linear3(x))

        return x


model = MLP()
loss_function = nn.MSELoss()
model . linear1 . weight . data = torch . normal (0 , 1 , model . linear1 . weight . shape )
model . linear1 . bias . data = torch . full ( model . linear1 . bias . shape , 0.03)
model . linear2 . weight . data = torch . normal (0 , 1 , model . linear2 . weight . shape )
model . linear2 . bias . data = torch . full ( model . linear2 . bias . shape , 0.03)
model . linear3 . weight . data = torch . normal (0 , 1 , model . linear3 . weight . shape )
model . linear3 . bias . data = torch . full ( model . linear3 . bias . shape , 0.03)

optimizer = torch.optim.SGD(model.parameters(), lr = alpha)

for epoch in range(K):
    for x, y in train_dataloader:
        optimizer.zero_grad()
        train_loss = loss_function(model(x) , y)
        train_loss.backward()

        optimizer.step()


with torch.no_grad():
    xx = torch.linspace(-2,2,1024).unsqueeze(1)
    plt.plot(X_train,y_train,'rx',label='Data points')
    plt.plot(xx,f_true(xx),'r',label='True Fn')
    plt.plot(xx, model(xx),label='Learned Fn')
plt.legend()
plt.show()


'''
When plotting torch tensors, you want to work with the
torch.no_grad() context manager.

When you call plt.plot(...) the torch tensors are first converted into
numpy arrays and then the plotting proceeds.
However, our trainable model has requires_grad=True to allow automatic
gradient computation via backprop, and this option prevents
converting the torch tensor output by the model to a numpy array.
Using the torch.no_grad() context manager resolves this problem
as all tensors are set to requires_grad=False within the context manager.

An alternative to using the context manager is to do
plt.plot(xx, model(xx).detach().clone())
The .detach().clone() operation create a copied pytorch tensor that
has requires_grad=False.

To be more precise, .detach() creates another tensor with requires_grad=False
(it is detached from the computation graph) but this tensor shares the same
underlying data with the original tensor. Therefore, this is not a genuine
copy (not a deep copy) and modifying the detached tensor will affect the
original tensor is weird ways. The .clone() further proceeds to create a
genuine copy of the detached tensor, and one can freely manipulate and change it.
(For the purposes of plotting, it is fine to just call .detach() without
.clone() since plotting does not change the tensor.)

This discussion will likely not make sense to most students at this point of the course.
We will revisit this issue after we cover backpropagation.

'''
```



Data points
True Fn
Learned Fn
4
2
0
−2
−4
−3
−2
−1
0
1
2
3
4