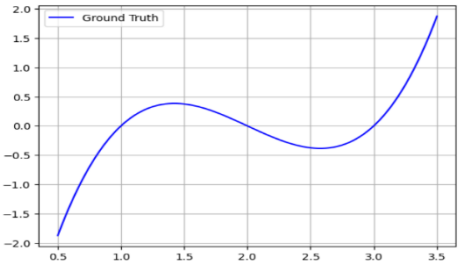
Summary Report [Week 4]. PyTorch

20190106 KimByungjun (김병준)

● Exercises

```
[ ]  x = np.linspace(0.5, 3.5, 100)
     y = x**3 - 6*x**2 + 11*x - 6

     plt.plot(x, y, color='blue', label='Ground Truth')
     plt.grid()
     plt.legend()
     plt.show()
```



[Fig 1] Function for Approximation

이번 Exercise 의 목표는 다항함수 $y = x^3 - 6x^2 + 11x - 6$ 을 approximating 하는 linear regression model 을 Pytorch 를 이용해서 구현하는 것이며, 이 때 linear model 은 $\hat{y} = w_1x_1 + w_2x_2 + w_3x_3 + b\ (W = [w_1, w_2, w_3]: weight, b: bias)$ 로 설정할 것이다. 이 경우, x_1, x_2, x_3 에는 input data 로 받은 단일 x 값을 $[x^3, x^2, x]$ 의 size (3,1)인 tensor 로 가공하고 $[x_1, x_2, x_3]$ 에 넣으면 되고, bias b가 상수항의 역할을 하여, linear regression model 로 표현할 수 있다.

그렇게 얻은 model 을 epoch = 25000 으로 학습(train)시켜 $y = x^3 - 6x^2 + 11x - 6$에 유사한 trained 3 차 함수 model 을 얻는 게 최종 목표이다. 다시 말해, 적합한 weight 와 bias 를 model training 을 통해 찾는 것이다.

torch, numpy 라이브러리와 matplotlib.pyplot 모듈을 import 하고, 우리가 근사시키고 싶은 함수 $y = x^3 - 6x^2 + 11x - 6$ 를 plotting 하면 **[Fig 1]**과 같다. Code 의 맨 윗 줄을 보면 [0.5, 3.5] 구간 내 동일 간격의 100 개 점을 ndarry type 변수 x 에 저장했다. 이에 대한 가공은 **2)** 에서 설명한다.

```
0초  class LinearModel(nn.Module):
         def __init__(self, input_dim, output_dim):
             super(LinearModel, self).__init__()
             self.linear = nn.Linear(input_dim, output_dim)

         def forward(self, x):
             out = self.linear(x)
             return out
```

[Fig 2] LinearModel class Implementation

앞서 말한 linear model 구현을 위해 **[Fig 2]**와 같이 LinearModel 이란 이름의 두 parameters(input_dim: 입력 텐서 크기, output_dim: 출력 텐서 크기)를 인자로 받는 class 를 정의하고 linear 라는 속성으로 torch 내 nn.Linear 함수를 쓰게 implement 하였다. Model 에 input data 를 넣는 부분이 forward 함수이며, 해당 함수는 만들어진 model 에 input tensor 를 넣어서 나오는 output tensor 를 반환한다.

```
0초  [55]  model = LinearModel(3,1).linear
```

[Fig 3] Code for Exercise 1

1) Construct an appropriate model for this exercise.

< Explanation >

앞서 정의한 LinearModel class 를 이용해 model 인스턴스를 구현할 것이며, nn.Linear 함수 활용을 위해, LinearModel 내 linear 속성을 사용하였다. 이 때, input 으로 들어갈 data 들은 $[x^3, x^2, x]$ 형태의 size 가 3 인 tensor, output 의 경우 단일 y 값 (size = 1) 하나가 나오므로 parameter 로는 **[Fig 3]**과 같이 (input_dim = 3, output_dim = 1)로 지정해야 한다. (해당 code 의 output 은 따로 없다)

2) How should you create input data for your linear model?

< Explanation >

1)에서 만든 model 에 input 으로 넣을 input tensor 인 x_for_model 을 만들어야 한다. 이는 앞서 언급한 np.linspace 로 생성된 ndarry 타입 변수 x 내 원소 100 개 각각에 대해 $[x^3, x^2, x]$ 의 size 3 의 새 array 로 가공해야 한다. 이에 대한 code 가 [Fig 4]의 빨간색 box 부분에 해당한다. for 문을 이용해 모든 원소 100 개에 대한 size 3 array($[x_i^3, x_i^2, x_i]$)를 만들었다.

```
x_for_model = np.array([[x_i**3, x_i**2, x_i] for x_i in x]) # Enter your code
print(x_for_model.shape)

class CustomDataset(Dataset):
    def __init__(self, x, y):
        self.x_data = torch.FloatTensor(x)
        self.y_data = torch.FloatTensor(y.reshape(-1, 1))

    def __len__(self):
        return len(self.x_data)

    def __getitem__(self, idx):
        return self.x_data[idx], self.y_data[idx]

train_data = CustomDataset(x_for_model, y)
train_loader = DataLoader(train_data, batch_size=10, shuffle=True) # Total data size 100, mini batch size 10
```

```
(100, 3)
```

[Fig 4] Code for Exercise 2

Result. x_for_model.shape 를 출력하면 올바르게 (100,3)이 출력되는 걸 알 수 있다.

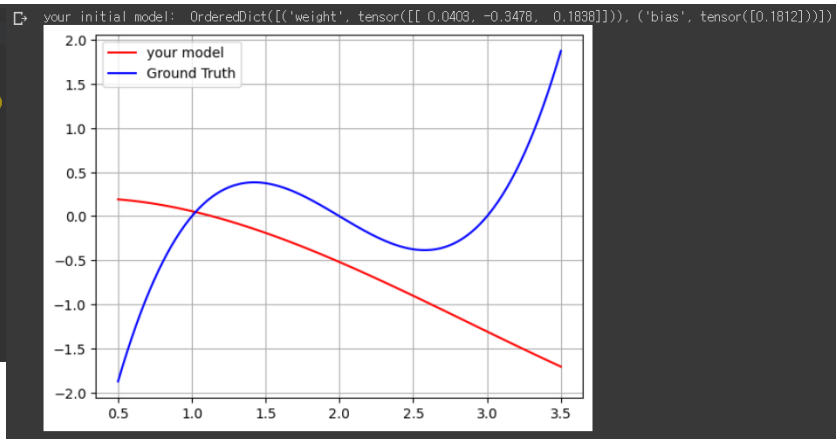
Note. train_data 는 CustomDataset 의 객체로 앞에 가공한 x_for_model 과 **[Fig 1]**에서 정의되었던 y array($y = x^3 - 6x^2 + 11x - 6$) 를 인자로 가지며, torch.FloatTensor 형으로 바뀐 x_data, y_data 를 속성으로 지닌다. train_loader 는 train_data 를 불러오는 DataLoader 객체로, batch_size=10 (iteration=$\frac{100(\text{data size})}{10(batch\ size)} = 10$), shuffle=True (data 섞기) parameter 을 custom 하게 설정하였다.

3) Complete the plot_model to plot your current model.

```
def plot_model(model):
    with torch.no_grad():
        ##############################
        y_hat = model.forward(torch.FloatTensor(x_for_model))
        ##############################

    plt.plot(x, y_hat, color='red', label='your model')
    plt.plot(x, y, color='blue', label='Ground Truth')
    plt.grid()
    plt.legend()
    plt.show()

print("your initial model: ", model.state_dict())
plot_model(model)
```

[Fig 5-1] Code for Exercise 3 (Left)

```
your initial model:  OrderedDict([('weight', tensor([[ 0.0403, -0.3478,  0.1838]])), ('bias', tensor([0.1812]))])
```



[Fig 5-2] Result for Exercise 3 (Right)

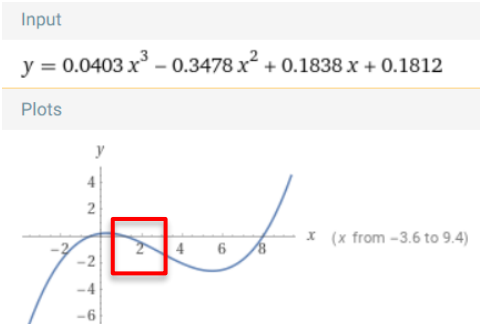
< Explanation >

[Fig 3]에서 정의한 model 의 초기 weight tensor $[w_1, w_2, w_3]$ 와 bias b 는 randomly 하게 설정되어 있다. 즉, $\hat{y} = w_1x_1 + w_2x_2 + w_3x_3 + b$ 는 아직 training 을 하지 않은 무작위한 coefficient w_1, w_2, w_3와 b를 가지고 있으며, 그 상태에서 x_for_model 을 [Fig 5-1]에 보이듯이 model 에 input 으로 넣었다. 그 결과인 size=100 의 tensor 를 y_hat 변수에 저장하였는데, 이 때 x_for_model 이 **[Fig 4]**에서 ndarray 자료형으로 선언되어 있으므로, model 적용을 위해 torch.FloatTensor 로 변환시켰다(앞선 CustomDataset class 와 consistent 한 type 으로 FloatTensor 를 사용한다).

Pyplot 모듈에 내장된 함수(plot,grid,legend,show)로 x = np.linspace(0.5,3.5,100) 에 따른 ground-truth(기대하는 answer) $y = x^3 - 6x^2 + 11x - 6$를 파란색, training 전 model $\hat{y} = w_1x^3 + w_2x^2 + w_3x + b$ (randomized weight, bias)를 빨간색으로 plotting 한 결과가 **[Fig 5-2]**이다. 더 자세한 정보를 위해 model.state_dict() 함수를 통해 weight 와 bias tensor 를 추가로 출력하였다.

Result. **[Fig 5-2]**를 보면 알 수 있듯이 training 전 model 은 ground truth 와 전혀 다른 개형을 보이고 있음을 알 수 있다. 이는 앞서 말했듯이, weight 와 bias 가 무작위로 초기값을 가지고 있으며 이는

출력된 weight : tensor([0.0403, -0.3478, 0.1838]), bias : tensor([0.1812])를 보면 알 수 있다. Training 전 model 은 $\hat{y} = 0.0403x^3 - 0.3478x^2 + 0.1838x + 0.1812$ 이고, **[Fig 5-2]**는 **[Fig 6]**의 표시된 modeling function(training 전)의 일부분에 해당한다(Wolframalpha 를 사용하였다).



[Fig 6] Regression Model before training

● Discussion

그렇다면 training 을 거친 이후 model 은 ground truth 와 유사한 지 확인해보고자 한다. Mean square error(MSE) loss 를 해당 model 의 training 을 위한 loss function 으로 설정하며(**[Fig 7]**), loss 를 기반으로 model 의 weight 와 bias 를 수정하는 것이 regression 의 핵심이다.

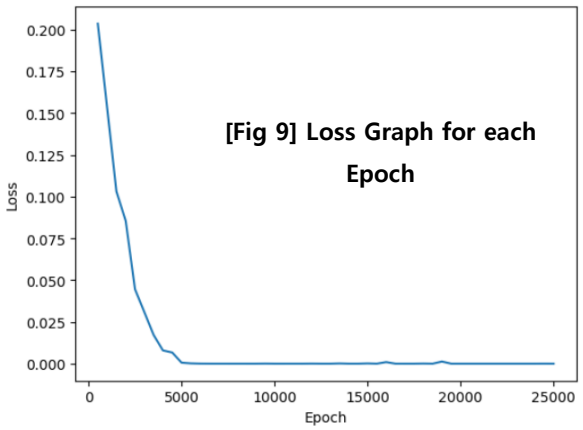
```
[ ]  loss_function = nn.MSELoss()

     optimizer = optim.Adam(params=model.parameters(), lr=0.01)
```

[Fig 7] loss function and optimization function

정해진 함수들을 기반으로 **[Fig 8]**과 같이 model 을 training 시킨다. Epoch 는 25000 으로 설정하며, 500 epoch 마다 loss 와 weight, bias tensor 를 출력하게 설정하였다. **[Fig 9]**는 epoch 에 대한 loss 의 변화를 plotting 한 graph 이고, **[Fig 10]** 은 loss 와 weight, bias tensor 값을 처음 2000 epoch 까지와 마지막에서 3000 epoch 인 경우에 대해 출력한 결과를 나타낸 것이다. 우선, ① **[Fig 9]**에서 대략 5000 epoch 부터는

```
[ ]  train_epochs = 25000
     loss_list = []
     epoch_list = []

     for epoch in range(train_epochs):
         average_loss = []
         steps = 0

         for i, (x_train, y_train) in enumerate(train_loader):
             pred = model(x_train)
             loss = loss_function(pred, y_train)
             average_loss.append(loss.detach().numpy())

             optimizer.zero_grad()
             loss.backward()
             optimizer.step()

         if (epoch+1) % 500 == 0:
             print('Epoch: ',  epoch+1, ' / Loss: ', np.mean(average_loss), ' /model parameter: ', model.state_dict())
             loss_list.append(np.mean(average_loss))
             epoch_list.append(epoch+1)

     plt.plot(epoch_list, loss_list, label='train_loss')
     plt.xlabel("Epoch")
     plt.ylabel("Loss")
     plt.show()
```

[Fig 8] Training Model



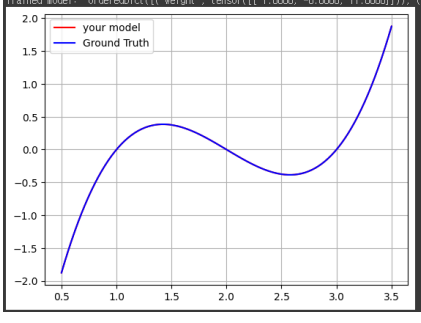
[Fig 9] Loss Graph for each Epoch

```
Epoch:  500  / Loss:  0.20356789  /model parameter:  OrderedDict([('weight', tensor([[ 0.1807, -0.9425,  1.7169]])), ('bias', tensor([-1.1889]))])
Epoch:  1000  / Loss:  0.15332231  /model parameter:  OrderedDict([('weight', tensor([[ 0.3037, -1.6808,  3.0721]])), ('bias', tensor([-1.9006]))])
Epoch:  1500  / Loss:  0.1081328  /model parameter:  OrderedDict([('weight', tensor([[ 0.4138, -2.3837,  4.3526]])), ('bias', tensor([-2.5659]))])
Epoch:  2000  / Loss:  0.0854823  /model parameter:  OrderedDict([('weight', tensor([[ 0.5214, -3.0315,  5.5435]])), ('bias', tensor([-3.1788]))])
Epoch:  23000  / Loss:  1.564028e-09  /model parameter:  OrderedDict([('weight', tensor([[ 1.0000, -6.0002, 11.0003]])), ('bias', tensor([-6.0002]))])
Epoch:  23500  / Loss:  6.874946e-08  /model parameter:  OrderedDict([('weight', tensor([[ 1.0000, -6.0001, 11.0004]])), ('bias', tensor([-6.0001]))])
Epoch:  24000  / Loss:  3.2730045e-09  /model parameter:  OrderedDict([('weight', tensor([[ 0.9999, -5.9995, 10.9992]])), ('bias', tensor([-5.9997]))])
Epoch:  24500  / Loss:  1.9088486e-05  /model parameter:  OrderedDict([('weight', tensor([[ 0.9998, -5.9993, 10.9983]])), ('bias', tensor([-5.9989]))])
Epoch:  25000  / Loss:  1.5143413e-10  /model parameter:  OrderedDict([('weight', tensor([[ 1.0000, -6.0000, 11.0000]])), ('bias', tensor([-6.0000]))])
```

[Fig 10] Loss / weight tensor / bias tensor (Left)

loss 가 거의 0 에 수렴한다는 것을 알 수 있으며, ② **[Fig 10]**의 초기(0~2000 epoch)에는 상대적으로 높은 loss 와 ground truth 에 해당하는 [1,-6,11] 과는 다른 weight, [-6]과는 다른 bias 를 보이지만 23000~25000 epoch 에선 거의 유사한 weight 와 bias tensor 를 가짐을 쉽게 알 수 있다. Regression 이 Linear Model 을 통해 잘되었음을 알 수 있으며, ③ **[Fig 11]**의 output 상단에 보이는 최종 weight, bias tensor 그리고 그 아래 그려진 ground truth 와 (거의) 완전히 일치하는 model graph 를 통해서도 training 이 잘되었음을 확인할 수 있다.

```
print("Trained model: ", model.state_dict())
plot_model(model)

Trained model:  OrderedDict([('weight', tensor([[ 1.0000, -6.0000, 11.0000]])), ('bias', tensor([-6.0000]))])
```



[Fig 11] Regression model (Trained)