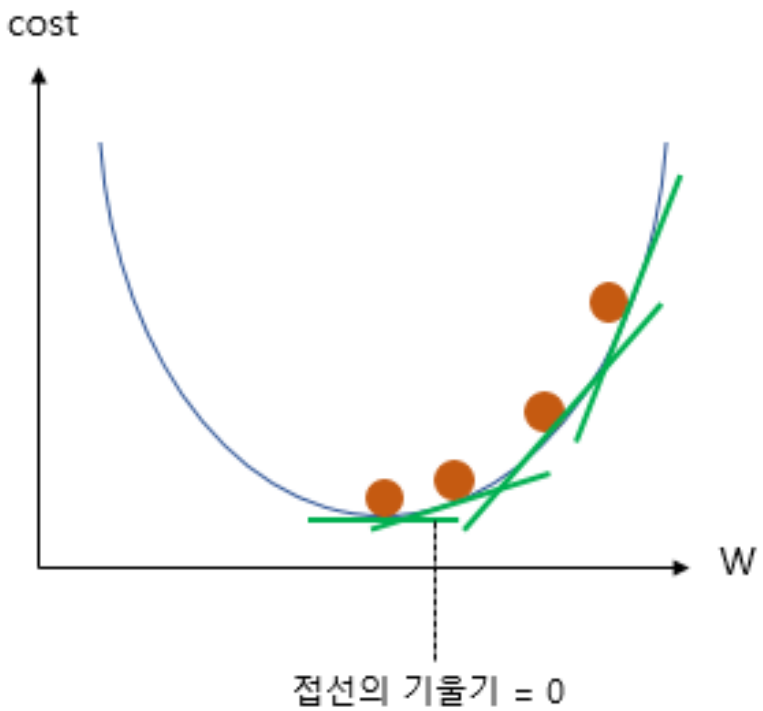
Pytorch 선형회귀 예제

선형회귀



https://wikidocs.net/60754

초기선언

📌 변수 선언

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

x_train = torch.FloatTensor([[1], [2], [3], [4], [5], [6]])
y_train = torch.FloatTensor([[2], [4], [6], [7], [11], [14]])
```

📌 가중치, 편향 초기화

```
#%% 가중치, 편향 초기화
W = torch.zeros(1, requires_grad=True)
b = torch.zeros(1, requires_grad=True)
```

선형회귀 식을 만들기 위해서 기울기와 절편에 대한 가중치를 초기화한다. requires_grad=True는 해당 변수에 gradient 값을 저장되고, 학습을 통해서 값이 계속 업데이트 된다는 것을 의미한다.

경사하강법 사용 학습

```
1  #경사 하강법 구현
2  optimizer = optim.SGD([W, b], lr=0.01)
3
4  epochs = 20000
5
6  for epoch in range(epochs + 1):
7
8      H = x_train * W + b  #가설 설정
9      cost = torch.mean((H - y_train ) ** 2) #비용함수 선언
10
11     optimizer.zero_grad()  #경사 0으로 초기화
12     cost.backward()  #비용함수 미분해서 gradient 계산
13     optimizer.step()  #W,b 업데이트
14
15     if epoch % 100 == 0:
16            print('Epoch {:4d}/{} W: {:.3f}, b: {:.3f} Cost: {:.6f}'.format(
17                epoch, epochs, W.item(), b.item(), cost.item()))
```

for 반복문을 통해서 에폭마다 학습을 진행하면서 경사하강을 진행한다.

라인11 : 기울기를 0으로 초기화한다. 각 에폭을 반복할 때 마다 기울기를 초기화해야 새로운 가중치화 편향에 대한 기울기를 구할 수 있다.

라인12 : 새로운 W, b에 대한 gradient(기울기)를 계산한다.

라인13 : 설정한 옵티마이저에 step함수를 이용해서 새로 계산한 기울기에 learning rate를 곱해서 새로운 값으로 업데이트 한다.

자동 미분

자동미분(Autograd)

```
1  w = torch.tensor(2.0, requires_grad=True)
2
3  y = w**2
4  z = 2*y + 5
5
6  z.backward()
7
8  w.grad  #미분값 출력
```

```
In [41]: w.grad
Out[41]: tensor(8.)
```

파이토치에서는 경사하강법 수행 과정에서 자동 미분을 실시하는 autograd 기능을 지원한다.

$2w^2+5$ 라는 식에 대해서 backward()로 미분을 실시해서 gradient를 계산하면, w에 대한 미분값(4w)이 자동으로 w.grad에 8로 저장되어 있다.

행렬 연산으로 구하기

$$H(X) = w_1x_1 + w_2x_2 + w_3x_3$$

위 식은 아래와 같이 두 벡터의 내적으로 표현할 수 있습니다.

$$\begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} x_1w_1 + x_2w_2 + x_3w_3 \end{pmatrix}$$

https://wikidocs.net/54841

H(x)로 표현된 다중회귀식을 아래와 같이 행렬 연산으로 간단하게 표현할 수 있다.

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \\ x_{41} & x_{42} & x_{43} \\ x_{51} & x_{52} & x_{53} \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} + \begin{pmatrix} b \\ b \\ b \\ b \\ b \end{pmatrix} = \begin{pmatrix} x_{11}w_1 + x_{12}w_2 + x_{13}w_3 + b \\ x_{21}w_1 + x_{22}w_2 + x_{23}w_3 + b \\ x_{31}w_1 + x_{32}w_2 + x_{33}w_3 + b \\ x_{41}w_1 + x_{42}w_2 + x_{43}w_3 + b \\ x_{51}w_1 + x_{52}w_2 + x_{53}w_3 + b \end{pmatrix}$$

Zero Grad사용 이유

zero_grad() 사용 이유

```python
w = torch.tensor(2.0, requires_grad=True)

nb_epochs = 20
for epoch in range(nb_epochs + 1):
    z = 2*w
    z.backward()
    print(w.grad)
```

```
tensor(2.)
tensor(4.)
tensor(6.)
tensor(8.)
tensor(10.)
tensor(12.)
tensor(14.)
tensor(16.)
tensor(18.)
tensor(20.)
tensor(22.)
tensor(24.)
tensor(26.)
tensor(28.)
tensor(30.)
tensor(32.)
tensor(34.)
tensor(36.)
tensor(38.)
tensor(40.)
tensor(42.)
```

모델 정의 및 학습

```
1  x_train  =  torch.FloatTensor([[73,  80,  75],
2                                 [93,  88,  93],
3                                 [89,  91,  80],
4                                 [96,  98,  100],
5                                 [73,  66,  70]])
6
7  y_train  =  torch.FloatTensor([[152],  [185],  [180],  [196],  [142]])
8
9  print(x_train.shape)
10 print(y_train.shape)
11
12 #가중치, 편향 정의
13 W = torch.zeros((3, 1), requires_grad=True)  #변수가 3개이므로 가중치도 3개
14 b = torch.zeros(1, requires_grad=True)
15
16
17 #경사하강법
18 optimizer = optim.SGD([W, b], lr=1e-5)
19
20 epochs = 50
21
22 for epoch in range(epochs +1):
23     H = x_train.matmul(W) + b  #행렬 연산으로 간단하게 표현
24     cost = torch.mean((H-y_train) ** 2)
25
26     optimizer.zero_grad()
27     cost.backward()
28     optimizer.step()
29
30     print('Epoch {:4d}/{} hypothesis: {} Cost: {:.6f}'.format(
31         epoch, epochs, H.squeeze().detach(), cost.item()
32     ))
```

예제 다중회귀식에 대해서 경사하강법을 실시하는 과정을 나타내면 위와 같다.

라인13: 변수가 3개이므로 이에 해당하는 가중치도 3개로 설정한다.
라인23: 다중회귀식을 matmul 함수로 간단하게 한 줄로 표현할 수 있다.

모델 정의 및 학습

파이토치에는 여러 알고리즘에 대한 함수가 구현되어 있어서 이를 불러와서 간단하게 사용할 수 있다.
다중선형회귀의 경우 torch.nn에서 nn.Linear()를 사용해서 구현할 수 있다.

```
1  import torch
2  import torch.nn as nn
3  import torch.nn.functional as F
4
5  # 데이터 생성
6  x_train = torch.FloatTensor([[1], [2], [3]])
7  y_train = torch.FloatTensor([[2], [4], [6]])
8
9  # 모델을 선언 및 초기화
10 model = nn.Linear(1,1)
11
12 #모델 파라미터 출력
13 print(list(model.parameters()))
14
15 #옵티마이저 설정
16 optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
17
18 #학습 : 경사하강법
19 epochs = 3000
20 for epoch in range(epochs+1):
21     prediction = model(x_train)  #H(x)계산, 클래스에 그냥 입력하면 됨
22
23     cost = F.mse_loss(prediction, y_train) #cost 계산
24
25     #cost로 H(x) 개선
26     optimizer.zero_grad()  #gradient=0 초기
27     cost.backward()  # 비용함수 미분해서 gradient 계산
28     optimizer.step()  # 파라미터 업데이트
29
30     if epoch % 100 == 0:
31     # 100번마다 로그 출력
32       print('Epoch {:4d}/{} Cost: {:.6f}'.format(
```

```
In [65]: print(list(model.parameters()))
[Parameter containing:
tensor([[-0.5435,  0.3462, -0.1188]], requires_grad=True), Parameter contain
tensor([0.2937], requires_grad=True)]
```

라인10 : 이 예제는 단순 선형 회귀이므로 input_dim=3, output_dim=1 으로 설정한다.
라인13 : model.parameters() 를 통해서

첫 번째가 기울기, 두 번째가 편향이다. 경사하강으로 업데이트 되는 변수이므로 requires_grad=True로 표시된다.

라인21 : prediction 함수로 행렬 연산에서 matmul이나 직

Class 사용

파이토치에서는 대부분 클래스를 이용해서 모델을 구현한다.

```
class LinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(1,1)

    def forward(self, x):
        return self.linear(x)

model = LinearRegressionModel()
```

모델 학습

```
x_train = torch.FloatTensor([[1], [2], [3]])
y_train = torch.FloatTensor([[2], [4], [6]])

optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

epochs = 20000
for epoch in range(epochs+1):

    prediction = model(x_train)
    cost = F.mse_loss(prediction, y_train)

    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    if epoch % 100 == 0:
        print('epoch {:4d}/{} Cost :{:.6f}' .format(epoch, epochs, cost.item()))
```