06. Optimizer

AILab
Hanyang Univ.

오늘 실습 내용

1. Tensor의 Gradient 계산방식
2. Optimizers

1. Tensor의 Gradient

Gradient

- Gradient는 어떻게 계산될까?
  - tensor.grad와 loss.backward()
- Example
  - a,b라는 데이터 Q라는 loss

```
import torch

a = torch.tensor([2.], requires_grad=True)
b = torch.tensor([6.], requires_grad=True)
```

requires_grad = False면 gradient 계산안함

```
Q = 3*a**3 - b**2
Q
```

```
tensor([-12.], grad_fn=<SubBackward0>)
```

$$Q = 3a^3 - b^2$$

a = 2, b=6

Q = 3*2^3 - 6^2 = 24 - 36 = -12

Gradient

- Loss backward를 하면 tensor.grad 로 각 텐서의 gradient값이 저장된다.
  → torch.autograd라는 모듈이 해줌

$$\frac{\partial Q}{\partial a} = 9a^2$$

$$\frac{\partial Q}{\partial b} = -2b$$

dQ/da = -9*2^2 = 36

dQ/db = -2*6 = 12

```
[6]  print(a.grad)
     print(b.grad)

     None
     None

[7]  Q.backward()

[8]  print(a.grad)
     print(b.grad)

     tensor([36.])
     tensor([-12.])
```

Gradient

•How does PyTorch optimizer work?

- Autograd
  - torch.autograd
  - PyTorch's automatic differentiation engine
  - https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

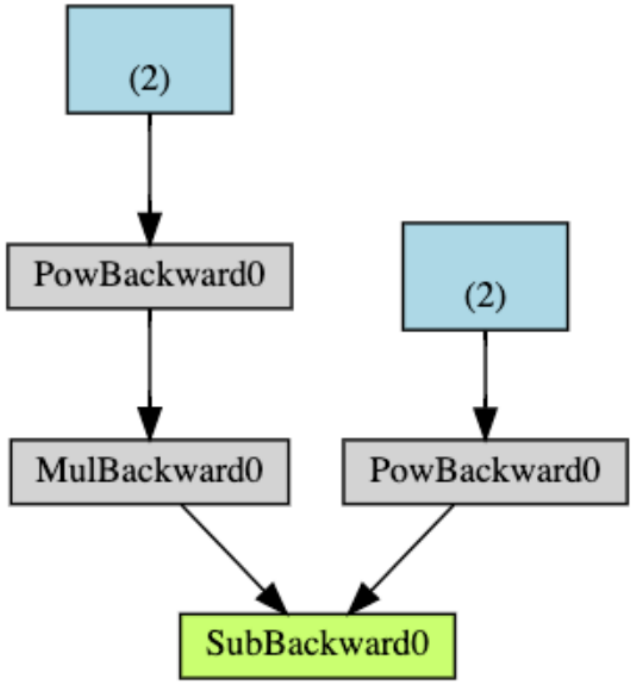
  - `logits = model(input) # Forward`
    - Computational Graph 생성 (DAG), grad_fn 저장

  - `loss.backward() # Backward`
    - Graph의 grad_fn 으로 gradient 계산
    - 각 Tensor.grad에 gradient 저장

  - `optimizer.step()`
    - 각 Tensor.grad에 저장된 gradient와 learning rate로 Tensor값 update

Gradient

- Autograd computing steps
  1. forward 시 autograd가 Graph 생성, grad_fn 저장
  2. backward시 graph root에 시작해 grad_fn으로 gradient 계산
  3. 계산한 gradient가 Tensor.grad에 저장
  4. chain rule을 따라 leaf로 gradient계산해감



$$Q = 3a^3 - b^2$$

```
a**3
tensor([8.], grad_fn=<PowBackward0>)
```

```
[4] b**2
tensor([36.], grad_fn=<PowBackward0>)
```

```
3*a**3
tensor([24.], grad_fn=<MulBackward0>)
```

```
Q = 3*a**3 - b**2
Q
tensor([-12.], grad_fn=<SubBackward0>)
```

Gradient

- Optimizer parameter update
  - optimizer.step()으로 learning rate와 Tensor.grad 사용하여 값을 update
  - https://github.com/pytorch/pytorch/blob/cd9b27231b51633e76e28b6a34002ab83b0660fc/torch/optim/sgd.py#L63

```
for p in group['params']:
    if p.grad is None:
        continue
    d_p = p.grad.data
    if weight_decay != 0:
        d_p.add_(weight_decay, p.data)
    if momentum != 0:
        param_state = self.state[p]
        if 'momentum_buffer' not in param_state:
            buf = param_state['momentum_buffer'] = d_p.clone()
        else:
            buf = param_state['momentum_buffer']
            buf.mul_(momentum).add_(1 - dampening, d_p)
        if nesterov:
            d_p = d_p.add(momentum, buf)
        else:
            d_p = buf

    p.data.add_(-group['lr'], d_p)
```

모든 파라미터에 대해

tensor gradient를 data로 복사

inplace add 함수로 learning rate와 tensor gradient로 값 업데이트

alpha = d_p
other = learning_rate
https://pytorch.org/docs/stable/generated/torch.Tensor.add_.html#torch.Tensor.add_

torch.add(*input*, *other*, *, *alpha=1*, *out=None*) → Tensor

Adds `other`, scaled by `alpha`, to `input`.

$$\text{out}_i = \text{input}_i + \text{alpha} \times \text{other}_i$$

Gradient

- Optimizer parameter update
  - optimizer.zero_grad()으로 Tensor.grad 값을 0으로 바꿈
  - https://github.com/pytorch/pytorch/blob/cd9b27231b51633e76e28b6a34002ab83b0660fc/torch/optim/optimizer.py#L109

```
def zero_grad(self):
    """Clears the gradients of all optimized :class:`Variable` s."""
    for group in self.param_groups:
        for p in group['params']:
            if p.grad is not None:
                if p.grad.volatile:
                    p.grad.data.zero_()
                else:
                    data = p.grad.data
                    p.grad = Variable(data.new().resize_as_(data).zero_())
```

tensor.grad가 0이 됨

더 자세한 내용은
몰라도 됨

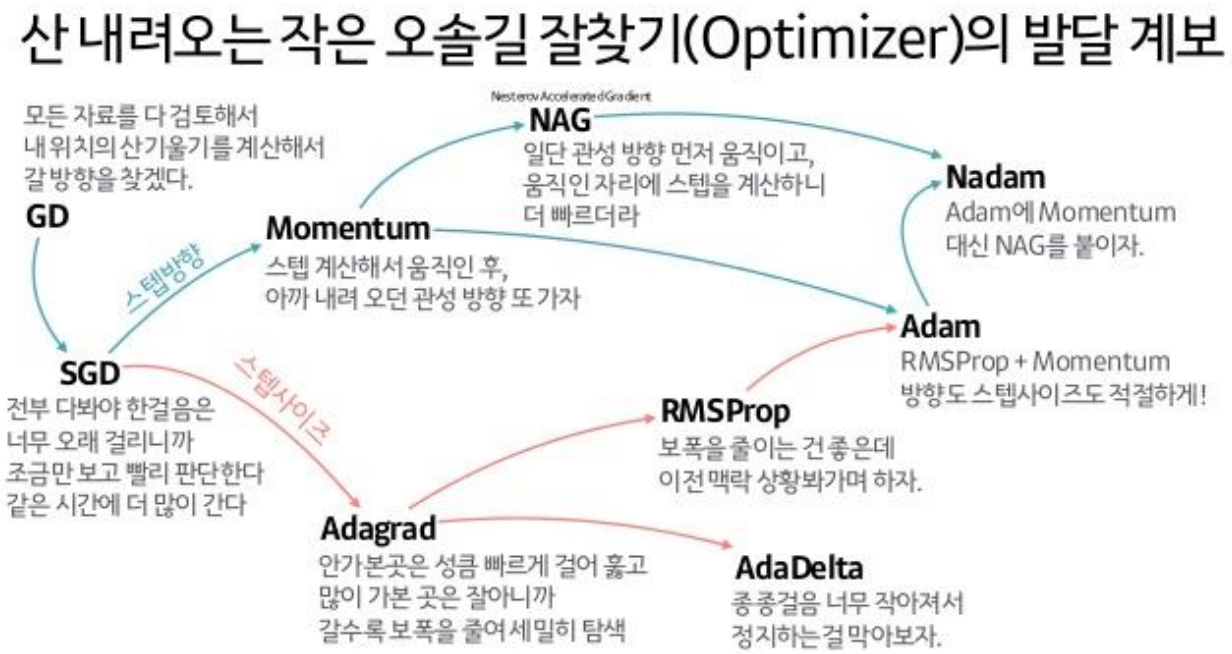
Gradient

- Conclusion
  - Loss backward시 gradient 계산된다.
  - Tensor.grad에 계산된 gradient 가 저장된다.
  - Optimizer.step() 으로 tensor의 값이 learning rate와 Tensor.grad로 update 된다.
  - Optimizer.zero_grad()로 Tensor.grad 값을 0으로 만든다.

2. Optimizer

Optimizer

Loss function 을 통해 구한 차이를 사용해 기울기를 구하고
Network의 parameter(W, b) 의 학습에 어떻게 반영할 것인지를 결정하는 방법



Optimizer

- Gradient Descent(GD)
- Stochastic Gradient Descent (SGD)
- Momentum
- Nesterov Accelerated Gradient (NAG)
- Adagrad
- RMSProp
- AdaDelta
- Adaptive Moment Estimation(Adam)

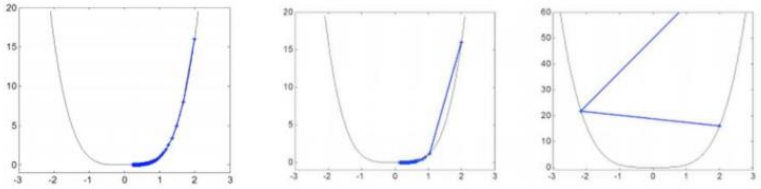
Gradient Descent(GD)

1회 step시 현재 모델의 **모든 data**에 대해서 예측 값에 대한 loss 미분을 learning rate 만큼 보정해서 반영하는 방법.

https://pytorch.org/docs/stable/generated/torch.optim.SGD.html

· 함수

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta)$$



$\eta = 0.01$ $\eta = 0.03$ $\eta = 0.13$

· 코드

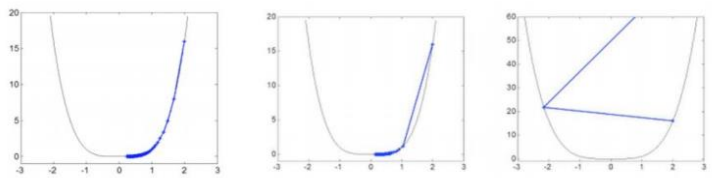
```
optimizer = torch.optim.SGD(params=model.parameters(), lr=1e-3)
```

Stochastic Gradient Descent (SGD)

- 한번 step을 내딛을 때 전체 데이터에 대한 Loss Function을 계산하면 매우 느림
- 이를 방지하기 위해 일부의 data sample이 전체 data set의 gradient와 유사할 것이라는 가정하에 **일부**에 대해서만 loss function을 계산

- 함수

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$



$\eta = 0.01$ $\eta = 0.03$ $\eta = 0.13$

- 코드

```
optimizer = torch.optim.SGD(params=model.parameters(), lr=1e-3)
```

Momentum

- 이전 step의 방향(=관성)과 현재 상태의 gradient를 더해 현재 학습할 방향과 크기를 정함
- Local minima를 빠져 나올 수 있다

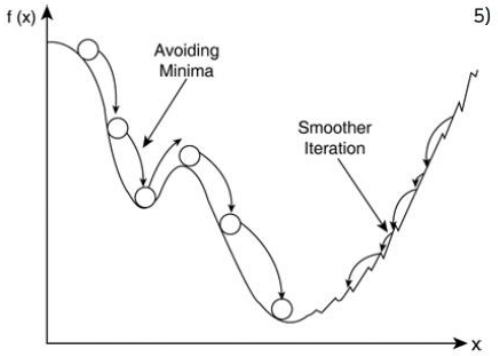
SGD

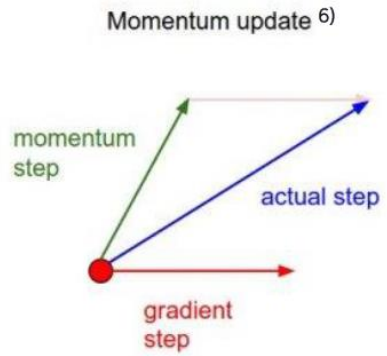
Momentum

· 함수

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$





· 코드

```
optimizer = torch.optim.SGD(params=model.parameters(), lr=1e-3, momentum=0.9)
```

γ : momentum

Adagrad (Adaptive Gradient)

- Parameter 별로 gradient 를 다르게 주는 방식, G : 이전 gradient 제곱의 합

- 많이 변화한 변수들은 G에 저장된 값이 커지기 때문에 step size가 작은 상태로
- 적게 변화한 변수들은 상대적으로 step size가 큰 상태로 학습에 반영
  - 단점: 학습이 오래 진행되는 경우 G값이 너무 커져서 학습이 제대로 되지 않는다

https://pytorch.org/docs/stable/generated/torch.optim.Adagrad.html#torch.optim.Adagrad

- 함수

$$G_t = G_{t-1} + (\nabla_\theta J(\theta_t))^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{(G_t + \epsilon)}} \cdot \nabla_\theta J(\theta_t)$$

- 코드

```
optimizer = torch.optim.Adagradd(params=model.parameters(), lr=1e-3)
```

RMSProp

- 학습이 오래 진행되면 step size가 너무 작아지는 Adagrad의 단점을 보완
- 각 변수에 대한 gradient의 제곱을 계속 더하는 것이 아니라, 지수평균으로 바꾸어 G값이 무한정 커지지 않도록 방지하면서 변화량의 상대적인 크기 차이를 유지

https://pytorch.org/docs/stable/generated/torch.optim.RMSprop.html#torch.optim.RMSprop

$$G = \gamma G + (1-\gamma)(\nabla_\theta J(\theta_t))^2$$

- 함수

$$\theta = \theta - \frac{\eta}{\sqrt{G+\epsilon}} \cdot \nabla_\theta J(\theta_t)$$

- 코드

```
optimizer = torch.optim.RMSprop(params=model.parameters(), lr=1e-3, alpha=0.99)
```

$alpha : \gamma$

Adaptive Moment Estimation(Adam)

- Momentum 방식과 유사하게 지금까지 계산해온 기울기의 지수평균을 저장
- RMSProp 와 유사하게 지금까지 계산해온 기울기의 제곱값의 지수평균을 저장
- 학습에 초반부에 m과 v가 0에 가깝게 bias되어 있을 것이라고 판단해 unbiased 작업을 거친 후에 계산.
  https://pytorch.org/docs/stable/generated/torch.optim.Adam.html#torch.optim.Adam

- 함수

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta J(\theta))^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta = \theta - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}}\hat{m}_t$$

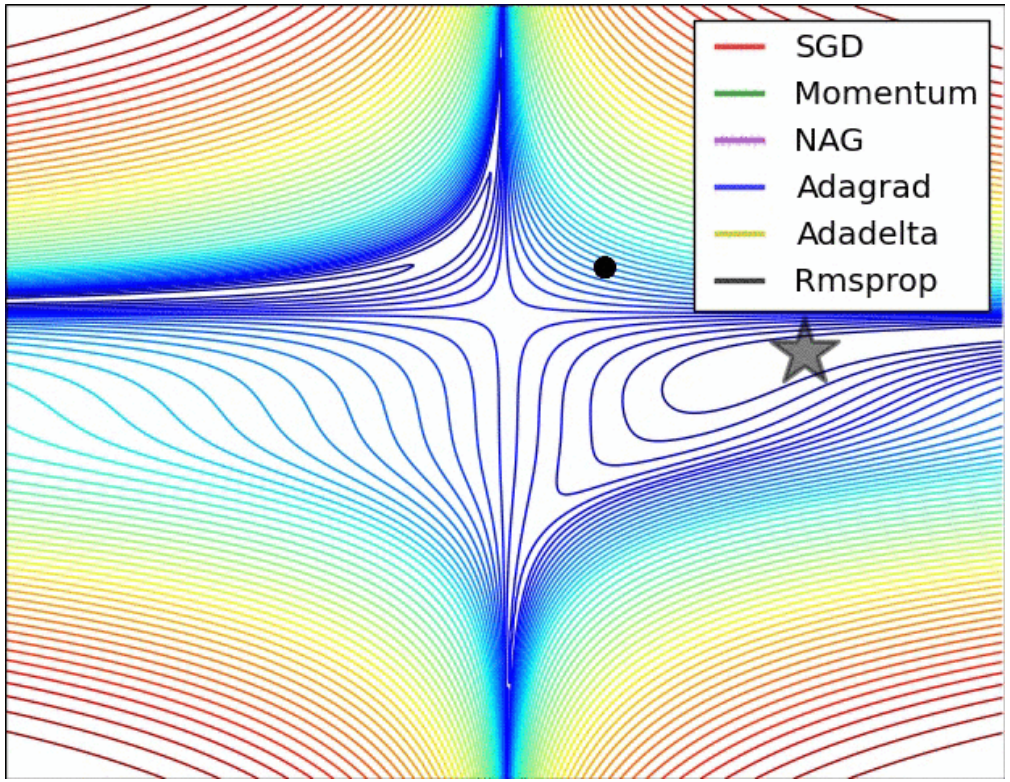
- 코드

```
optimizer = torch.optim.Adam(params=model.parameters(), lr=1e-3, betas=(0.9, 0.999))
```

Optimizer Comparisons

Optimizer 중 SGD의 수렴속도가 제일 느리다.
Optimizer간 수렴속도나 이동방향이 다름

별 : global optima



오늘 실습 내용

다양한 Optimizer를 이전 실습 코드에 적용하여 차이 확인해보기

step/epoch마다 loss, accuracy 변화 정도 등