# Optimization algorithms in deep learning

Sanzhar Askaruly

Ulsan National Institute of Science and Technology
Ph.D. Candidate in Biomedical Engineering

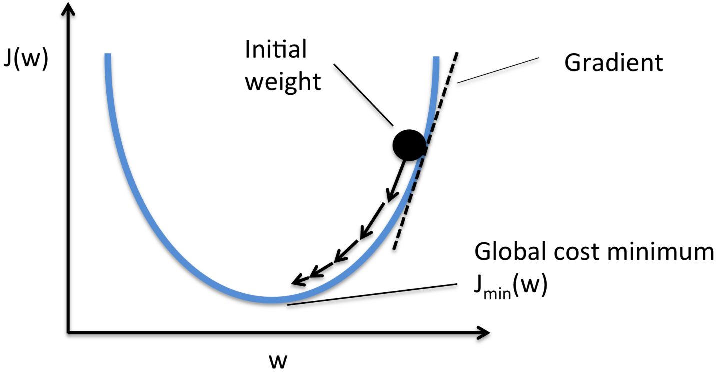
5 November 2022

## Optimization

In *context* of deep learning, goal is to **minimize loss function**

$$w^* = \arg\min_{w} L(w) \tag{1}$$

# What is gradient descent optimization?



# Stochastic Gradient Descent (SGD)

## Algorithm

Update step:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta J(\theta_t) \tag{2}$$

where,

| | |
|---|---|
| $\theta_t$: | current model parameters |
| $\nabla_\theta J(\theta_t)$: | gradient of these model parameters |
| $\eta$: | learning rate (fixed) |

# Stochastic Gradient Descent (SGD)

How we usually call in PyTorch:

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

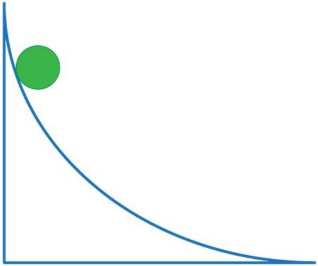
How we can create our "native" class:

```
from torch.optim.optimizer import Optimizer

class CustomSGD(Optimizer):
  def __init__(self, model_params, lr=1e-3):
      self.model_params = list(model_params)
      self.lr = lr

  def zero_grad(self):
      for param in self.model_params:
          param.grad = None

  @torch.no_grad()
  def step(self):
      for param in self.model_params:
          param.sub_(self.lr * param.grad)
```

# SGD with Momentum

General idea:

- Overcome small gradients near flat areas
- Build up from previous "velocity"
- Faster learning



# SGD with Momentum

## Algorithm

Update step [1]:

$$v_{t,i} = \gamma \cdot v_{t-1,i} + \nabla_\theta J(\theta_{t,i}) \tag{3}$$

$$\theta_{t+1} = \theta_t - \eta \cdot v_{t,i} \tag{4}$$

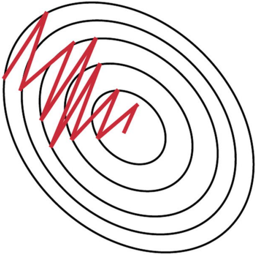
where,

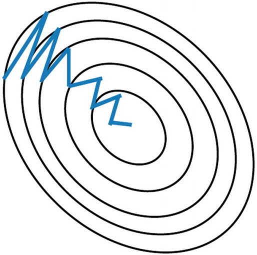
| | |
|---|---|
| $\gamma$: | friction (or momentum, fixed) |
| $v_t$: | velocity |
| $\nabla_\theta J(\theta_t)$: | gradient of these model parameters |
| $\eta$: | learning rate (fixed) |

# SGD with Momentum

```
from torch.optim.optimizer import Optimizer

class CustomSGDMomentum(Optimizer):
  def __init__(self, model_params, lr=1e-3, momentum=0.9):
      self.model_params = list(model_params)
      self.lr = lr
      self.momentum = momentum
      self.v = [torch.zeros_like(p) for p in self.model_params]

  def zero_grad(self):
      for param in self.model_params:
          param.grad = None

  @torch.no_grad()
  def step(self):
      for param, v in zip(self.model_params, self.v):
          v.mul_(self.momentum).add_(param.grad)
          param.sub_(self.lr * v)
```

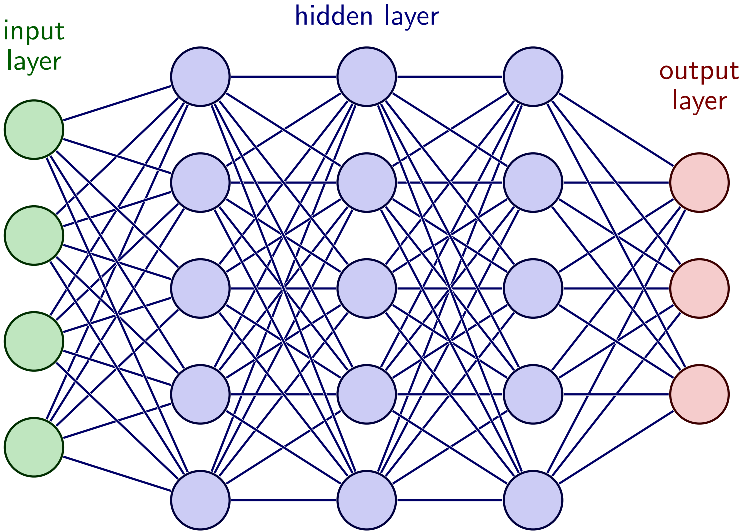
# SGD with Momentum [1]



# Experiment

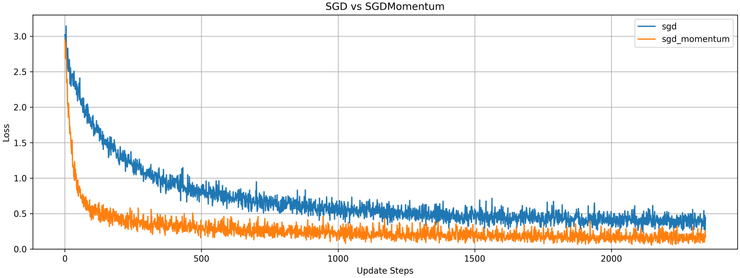
A vanilla MLP (Multilayer Perceptron)



# Experiment

MNIST dataset

# Experiment


SGD vs SGDMomentum
sgd
sgd_momentum
Loss
Update Steps
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
500
1000
1500
2000

# Adagrad

## Algorithm

Update step [**duchi2011adaptive**]:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,i}+\epsilon}} \cdot \nabla_\theta J(\theta_{t,i}) \tag{5}$$

where,

$$G_{t,i} = G_{t-1,i} + (\nabla_\theta J(\theta_{t,i}))^2 \tag{6}$$

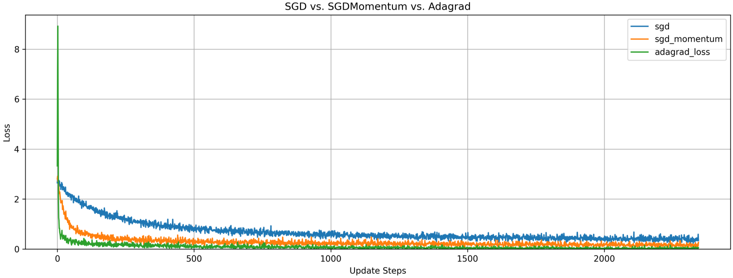
and,

| | |
|---|---|
| $G_{t,i}$: | the sum of the squared gradients |
| $\epsilon$: | a small number, to avoid division by zero |
| $\theta_t$: | current model parameters |
| $\nabla_\theta J(\theta_t)$: | gradient of these model parameters |
| $\eta$: | learning rate (fixed) |

# Adagrad

```
from torch.optim.optimizer import Optimizer

class CustomAdagrad(Optimizer):
  def __init__(self, model_params, lr=1e-2, init_acc_sqr_grad=0, eps=1e-10)
      self.model_params = list(model_params)
      self.lr = lr
      self.acc_sqr_grads = [torch.full_like(p, init_acc_sqr_grad) for p in 
      self.eps = eps

  def zero_grad(self):
      for param in self.model_params:
          param.grad = None

  @torch.no_grad()
  def step(self):
      for param, acc_sqr_grad in zip(self.model_params, self.acc_sqr_grads)
          acc_sqr_grad.add_(param.grad * param.grad)
          std = acc_sqr_grad.sqrt().add(self.eps)
          param.sub_((self.lr / std) * param.grad)
```

# Experiment



SGD vs. SGDMomentum vs. Adagrad
sgd
sgd_momentum
adagrad_loss
Loss
0
2
4
6
8
0
500
1000
1500
2000
Update Steps

Ning Qian. "On the momentum term in gradient descent learning algorithms". In: *Neural networks* 12.1 (1999), pp. 145–151.