CSC 561: Neural Networks and Deep Learning

Optimization (part 2)

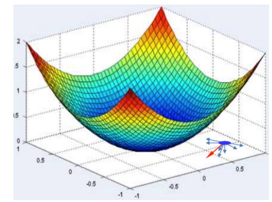
Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2024

Momentum methods

- Problems with GD
  - slow convergence — due to noisy gradients (mini-batch) or a flat landscape
  - oscillations
  - local minima



- Momentum
  - accumulate a history of gradients from previous iterations
  - combine past steps (velocity) with the current gradient
  - updates are larger in directions with smooth convergence and smaller in directions with oscillations

Momentum update

SGD

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} J\left(\mathbf{w}^{(t)}\right)$$

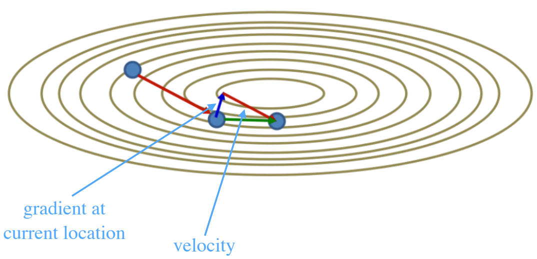
```
for i in range(n_iter):
    dw = gradient(w)
    w = w - lr * dw
```

SGD with momentum

$$\mathbf{v}^{(t+1)} = \beta \mathbf{v}^{(t)} - \eta \nabla_{\mathbf{w}} J\left(\mathbf{w}^{(t)}\right)$$

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \mathbf{v}^{(t+1)}$$

```
v = 0
for i in range(n_iter):
    dw = gradient(w)
    v = beta * v - dw
    w = w + lr * v
```



Momentum update

SGD with momentum

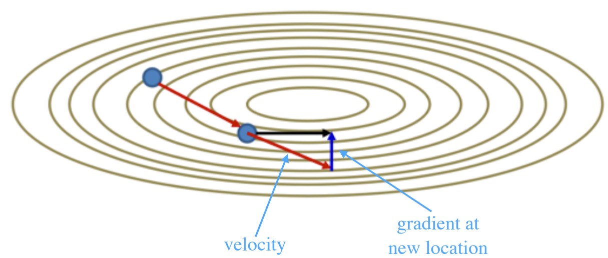
$$\mathbf{v}^{(t+1)} = \beta \mathbf{v}^{(t)} - \eta \nabla_{\mathbf{w}} J\left(\mathbf{w}^{(t)}\right)$$

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \mathbf{v}^{(t+1)}$$

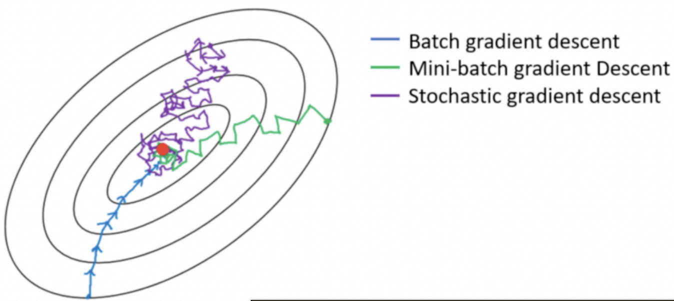
Nesterov momentum

$$\mathbf{v}^{(t+1)} = \beta \mathbf{v}^{(t)} - \eta \nabla_{\mathbf{w}} J\left(\mathbf{w}^{(t)} + \beta \mathbf{v}^{(t)}\right)$$

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \mathbf{v}^{(t+1)}$$



Momentum update



momentum provides faster convergence with sgd and mini-bath gradient descent

SGD

SGD
+ SGD

Pytorch's SGD

CLASS torch.optim.SGD(*params, lr=0.001, momentum=0, dampening=0, weight_decay=0, nesterov=False, *, maximize=False, foreach=None, differentiable=False*) [SOURCE]

Implements stochastic gradient descent (optionally with momentum).

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay), μ (momentum), τ (dampening), *nesterov*, *maximize*

for $t = 1$ **to** $\ldots$ **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  if $\mu \neq 0$
    if $t > 1$
      $\mathbf{b}_t \leftarrow \mu\mathbf{b}_{t-1} + (1-\tau)g_t$
    else
      $\mathbf{b}_t \leftarrow g_t$
    if *nesterov*
      $g_t \leftarrow g_t + \mu\mathbf{b}_t$
    else
      $g_t \leftarrow \mathbf{b}_t$
  if *maximize*
    $\theta_t \leftarrow \theta_{t-1} + \gamma g_t$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t$

return θ_t

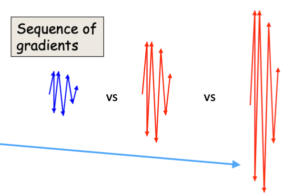
Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-3)
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)
- **maximize** (*bool, optional*) – maximize the objective with respect to the params, instead of minimizing (default: False)
- **foreach** (*bool, optional*) – whether foreach implementation of optimizer is used. If unspecified by the user (so foreach is None), we will try to use foreach over the for-loop implementation on CUDA, since it is usually significantly more performant. Note that the foreach implementation uses ~ sizeof(params) more peak memory than the for-loop version due to the intermediates being a tensorlist vs just one tensor. If memory is prohibitive, batch fewer parameters through the optimizer at a time or switch this flag to False (default: None)
- **differentiable** (*bool, optional*) – whether autograd should occur through the optimizer step in training. Otherwise, the step() function runs in a torch.no_grad() context. Setting to True can impair performance, so leave it False if you don't intend to run autograd through this instance (default: False)

Adagrad

- Idea: per-parameter learning rates
  - steep directions
    - large "total movement"
    - want lower learning rates
  - flat directions
    - want larger learning rates
  - keep a historical sum of squares in each dimension
  - squares capture total motion on both sides of oscillation




```
dw_squared = 0
for i in range(n_iter):
    dw = gradient(w)
    dw_squared += dw * dw
    w = w − lr * dw / sqrt(dw_squared + eps)
```

step size decays to zero over time (denominator grows)

From Adagrad to RMSProp

Adagrad

```
    dw_squared = 0
    for i in range(n_iter):
        dw = gradient(w)
        dw_squared += dw * dw
        w = w − lr * dw / sqrt(dw_squared + eps)
```

step size decays to zero over time

RMSProp

```
dw_squared = 0
for i in range(n_iter):
    dw = gradient(w)
    dw_squared = dr * dw_squared + (1−dr) * dw * dw
    w = w − lr * dw / sqrt(dw_squared + eps)
```

decay rate

Adam

- Previous methods
  - RMSProp adapts the learning rate per-parameter
  - Momentum smooths the gradient
- Adam (adaptive moments) combines both
  - moving averages for both the gradients (momentum) and the squared gradients (RMSProp)

```
first_moment = 0
second_moment = 0
for i in range(n_iter):
    dw = gradient(w)
    first_moment = beta1 * first_moment + (1-beta1) * dw
    second_moment = beta2 * second_moment + (1-beta2) * dw * dw
    w = w - lr * first_moment / (sqrt(second_moment) + eps)
```


Adam

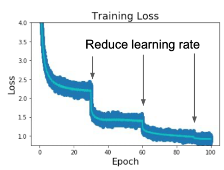
- Removing bias
  - as moments are initialized to zero, first update may be very large (small second_moment)

good starting points $\beta_1 = 0.9$, $\beta_2 = 0.999$

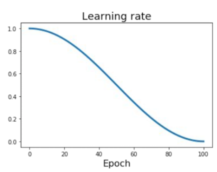
```
first_moment = 0
second_moment = 0
for i in range(n_iter):
    dw = gradient(w)
    first_moment = beta1 * first_moment + (1-beta1) * dw
    second_moment = beta2 * second_moment + (1-beta2) * dw * dw

    first = first_moment / (1 - beta1**(i+1))
    second = second_moment / (1 - beta2**(i+1))

    w = w - lr * first / (sqrt(second) + eps)
```

denominators approach 1 as i grows

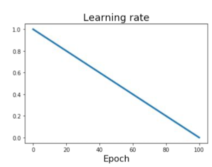
LR schedules (decaying over time)



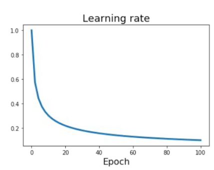
reduce LR at fixed intervals



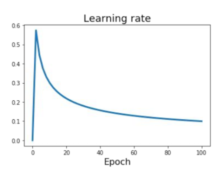
use cosine annealing



use linear decay



use inverse square root decay



use linear warmup