Optimization Algorithms 1: Momentum & Adagrad

D2L 11.6 - 11.7

Hayden R. Foote
ASTR 502
March 16, 2022

A Quick Recap of Minibatch Gradient Descent (GD)

Weights in each layer are updated via:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta_t \mathbf{g}_t$$

Learning rate at time t, typically decreases with time

The minibatch-averaged gradient at time t based on the weights at time t-1

$$\mathbf{g}_{t,t-1} = \partial_{\mathbf{w}} \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} f(\mathbf{x}_i, \mathbf{w}_{t-1}) = \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} \mathbf{h}_{i,t-1}$$

Gradient of the objective (loss) function for training example i with respect to the weights at time t-1

$$\mathbf{h}_{i,t-1} = \partial_{\mathbf{w}} f(\mathbf{x}_i, \mathbf{w}_{t-1})$$

Momentum

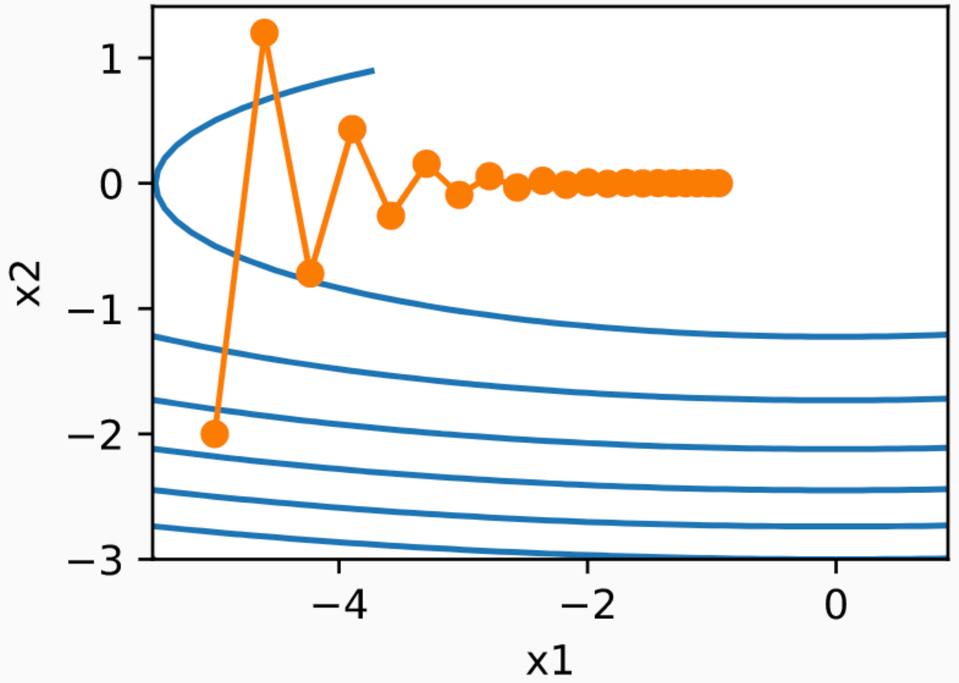
The Problem

- Learning rate η_t must be chosen very carefully - too large and the optimization does not converge, too small and the optimization stalls.
- This is particularly difficult in a “valley” of the objective/loss function, which are quite common near minima. A toy example:

Objective: $f(x) = 0.1x_1^2 + 2x_2^2$

Algorithm: GD with $\eta = 0.4$

Convergence in x1 is very slow



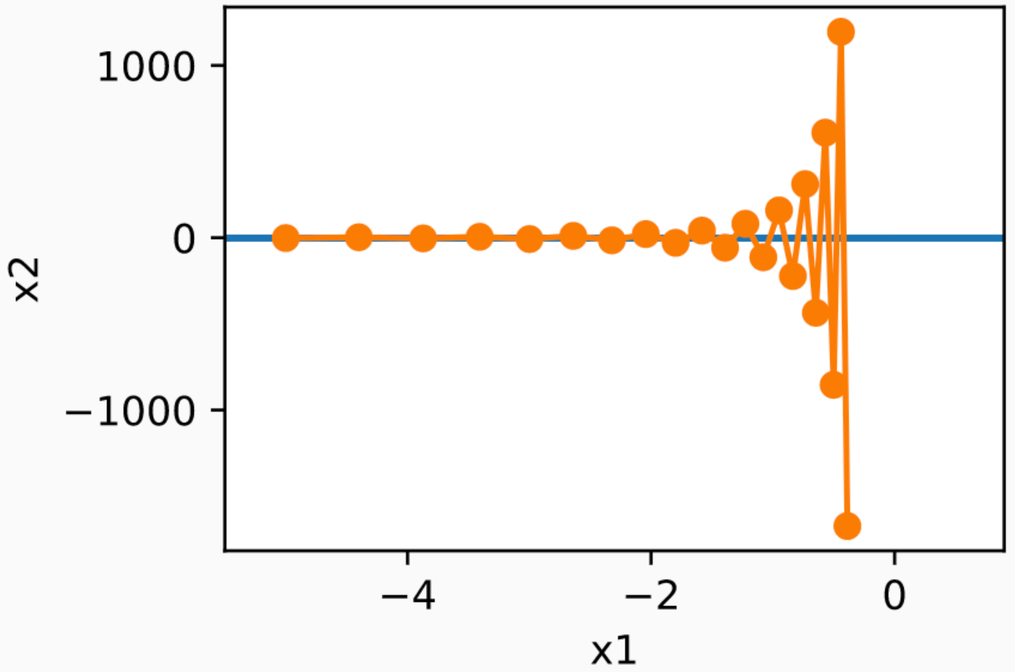
The Problem

- Learning rate η_t must be chosen very carefully - too large and the optimization does not converge, too small and the optimization stalls.
- This is particularly difficult in a “valley” of the objective/loss function, which are quite common near minima. A toy example:

Objective: $f(x) = 0.1x_1^2 + 2x_2^2$

Algorithm: GD with $\eta = 0.6$

Solution diverges in x2



The Solution - Momentum

- Momentum replaces the minibatch gradient with a “leaky” average of the past gradients:

Hyperparameter in (0, 1)

“Momentum” $\mathbf{v}_t \leftarrow \beta \mathbf{v}_{t-1} + \mathbf{g}_{t,t-1},$ ← Gradient

Weights $\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \eta_t \mathbf{v}_t.$

Learning rate at time t

Momentum is recursive:

$$\mathbf{v}_0 = \overrightarrow{0}$$

$$\mathbf{v}_1 = \mathbf{g}_1$$

$$\mathbf{v}_2 = \mathbf{g}_2 + \beta \mathbf{g}_1$$

$$\mathbf{v}_3 = \mathbf{g}_3 + \beta \mathbf{g}_2 + \beta^2 \mathbf{g}_1$$

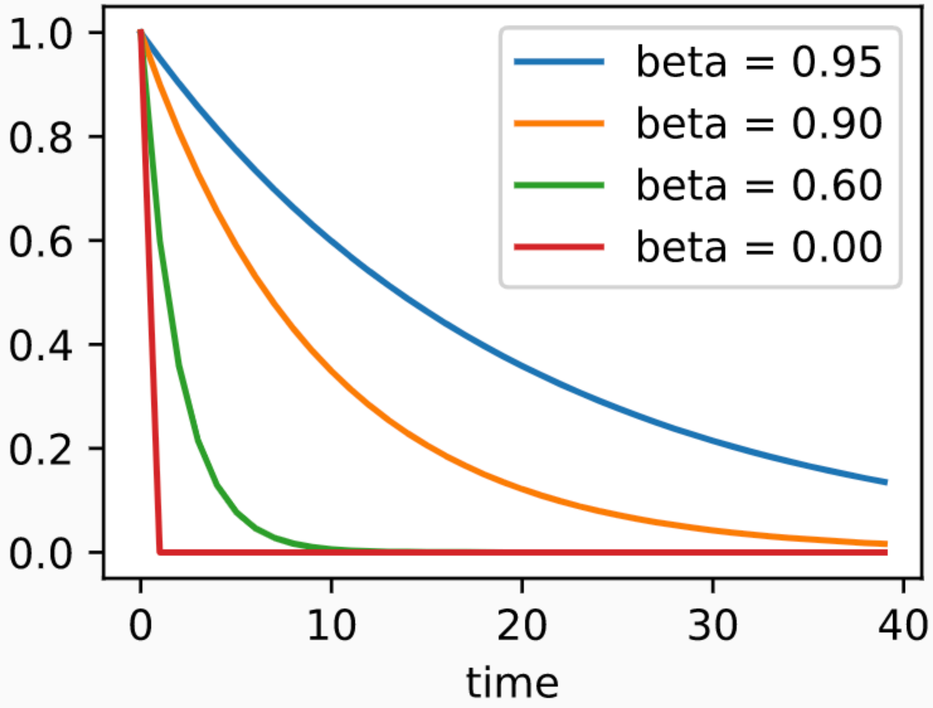
$$\vdots$$

$$\mathbf{v}_t = \sum_{\tau=0}^{t} \beta^\tau \mathbf{g}_{t-\tau}$$

Effective Sample Weight

Momentum $\mathbf{v}_t$ is effectively an average over the past $1/(1-\beta)$ gradients

Contribution to Average Gradient / Momentum



Iterations/Timesteps Behind

Momentum Performance

- Advantages: momentum takes effectively larger steps in a more efficient direction compared to GD. Including momentum usually works better than GD.
- Drawbacks: $\eta(t)$ and β still need to be chosen manually, bad choices may still not converge.
- A really nice visualization and article: https://distill.pub/2017/momentum/

A Generic Training Loop

```
def train_concise_ch11(trainer_fn, hyperparams, data_iter, num_epochs=4):
    # Initialization
    net = nn.Sequential(nn.Linear(5, 1))
    def init_weights(m):
        if type(m) == nn.Linear:
            torch.nn.init.normal_(m.weight, std=0.01)
    net.apply(init_weights)

    optimizer = trainer_fn(net.parameters(), **hyperparams)
    loss = nn.MSELoss(reduction='none')
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            optimizer.zero_grad()
            out = net(X)
            y = y.reshape(out.shape)
            l = loss(out, y)
            l.mean().backward()
            optimizer.step()
            n += X.shape[0]
            if n % 200 == 0:
                timer.stop()
                # `MSELoss` computes squared error without the 1/2 factor
                animator.add(n/X.shape[0]/len(data_iter),
                             (d2l.evaluate_loss(net, data_iter, loss) / 2,))
                timer.start()
    print(f'loss: {animator.Y[0][-1]:.3f}, {timer.avg():.3f} sec/epoch')
```

Network is a single dense layer with 5 inputs and one output

Parameter initialization from Gaussian

Set up optimizer

Set up MSE loss function

Loop over epochs

Loop over iterations/batches

Minibatch GD

Using Momentum

- Momentum is already implemented in `pytorch`'s SGD optimizer:

```
trainer = torch.optim.SGD
d2l.train_concise_ch11(trainer, {'lr': 0.005, 'momentum': 0.9}, data_iter)
```

Passing `momentum=beta` to `torch.optim.SGD` is all you need to do!

Adagrad

The Problem

- Learning rate η_t must be chosen very carefully - too large and the optimization does not converge, too small and the optimization stalls.
- But what happens when some features are very sparse compared to others?
  - The weights/parameters are only updated in response to features when they appear - convergence for sparse features will be very slow!
  - Parameters updated by sparse features tend to have small gradients, while parameters updated by frequent features tend to have large gradients.
  - A time-decreasing learning rate tends to optimize for frequent features, while sparse features may not come close to converging.

The Solution - Adagrad (Adaptive Gradient)

- We'd like an algorithm that uses large learning rates for parameters with low gradients, and small learning rates for parameters with high gradients.
- Recall that the Hessian matrix can be used to adjust the learning rate on a per-parameter basis based on the curvature of the objective function.
- However, Hessians are in practice very difficult or impossible to calculate for large numbers of parameters.
- Fortunately, the magnitude of the gradient itself is a useful proxy for the diagonal of the Hessian.
- Adagrad adjusts the learning rate on a per-parameter basis, based on the value of the gradient: higher gradients reduce the learning rate more.

Adagrad Details

Gradient of loss function $\mathbf{g}_t = \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})),$

Scaling for learning rate $\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2,$

Parameter update $\mathbf{w}_t = \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t.$

In Adagrad, all operations are **elementwise!**

Small positive number to avoid division by zero, typically 1e-8

Scaling is also recursive:

$$\mathbf{s}_0 = \overrightarrow{0}$$

$$\mathbf{s}_1 = \mathbf{g}_1^2$$

$$\mathbf{s}_2 = \mathbf{g}_2^2 + \mathbf{g}_1^2$$

$$\vdots$$

$$\mathbf{s}_t = \sum_{\tau=0}^{t} \mathbf{g}_\tau^2$$

$\mathbf{s}_t$ grows roughly linearly, leading to a learning rate that decreases as $\mathcal{O}(t^{-1/2})$.

Adagrad Performance

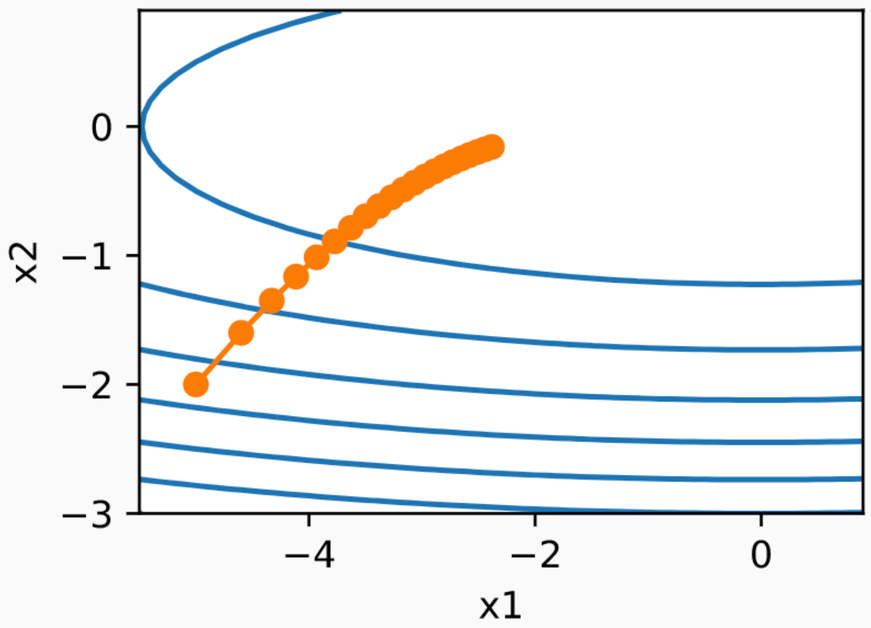
- Advantages: Significantly outperforms GD in applications that have sparse features, like language recognition, automated advertising, or classifying galaxies.

- Drawbacks: Since the learning rate decreases monotonically with time, it can very quickly become so small that the optimization stalls, especially if the problem is non-convex.

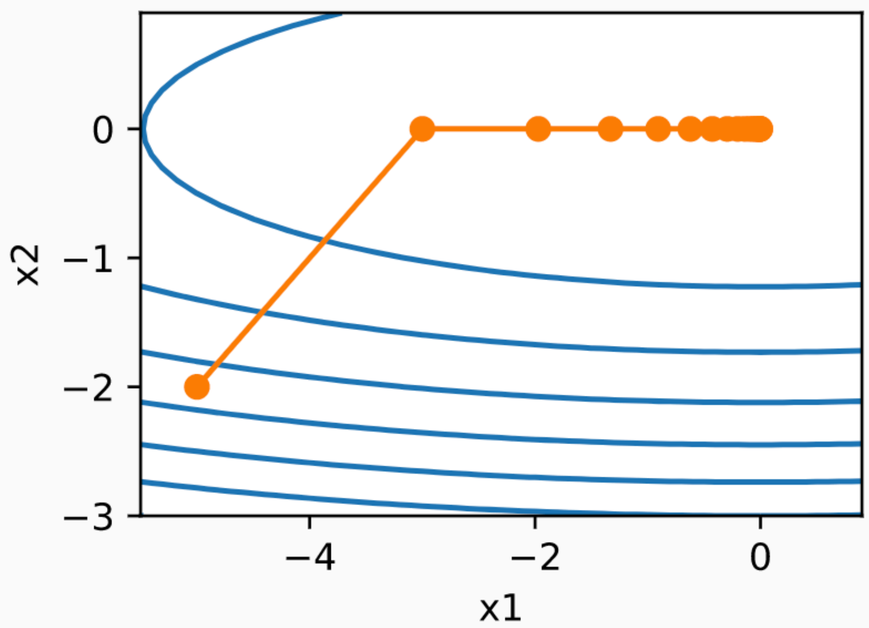
Adagrad Performance

Objective: $f(x) = 0.1x_1^2 + 2x_2^2$

$\eta = 0.4$



$\eta = 2$



Using Adagrad

- Adagrad is a separate optimizer in `pytorch`, but otherwise its use is identical to GD:

```
trainer = torch.optim.Adagrad
d2l.train_concise_ch11(trainer, {'lr': 0.1}, data_iter)
```

Summary

- Choosing a learning rate for Minibatch GD is difficult to do. If the loss function contains valleys near minima or the model is being trained to recognize sparse features, GD will often diverge or fail to converge.
- Momentum uses a leaky average of past gradients to update parameters, which damps oscillations in the optimization, leading to good performance in valleys.
- Adagrad takes larger steps in directions of low gradient, while taking smaller steps in directions of high gradient, leading to good performance when training for sparse features.
- Both algorithms are built-in to `pytorch` and are very easy to use.

References

- https://d2l.ai/index.html Chapter 11
- https://ruder.io/optimizing-gradient-descent/
- https://distill.pub/2017/momentum/