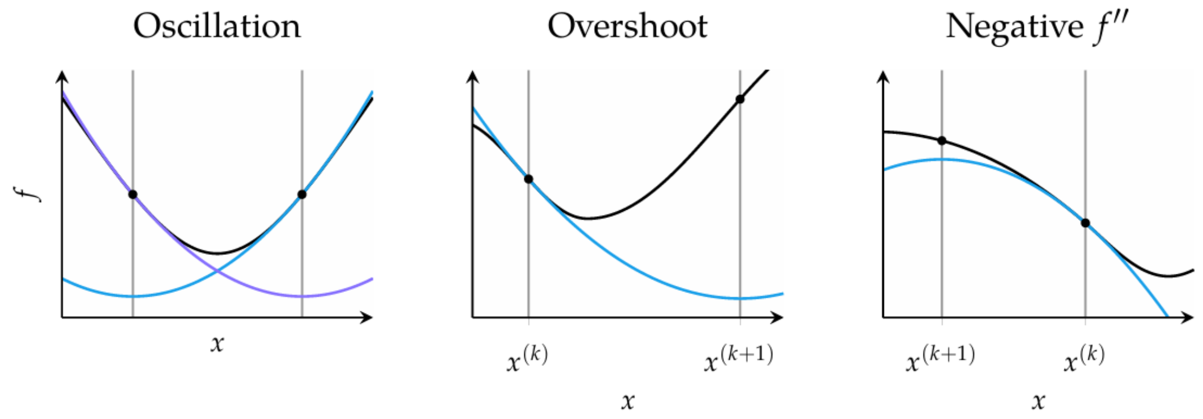


Second Order Methods

Andreas Makris, 2nd year PhD student

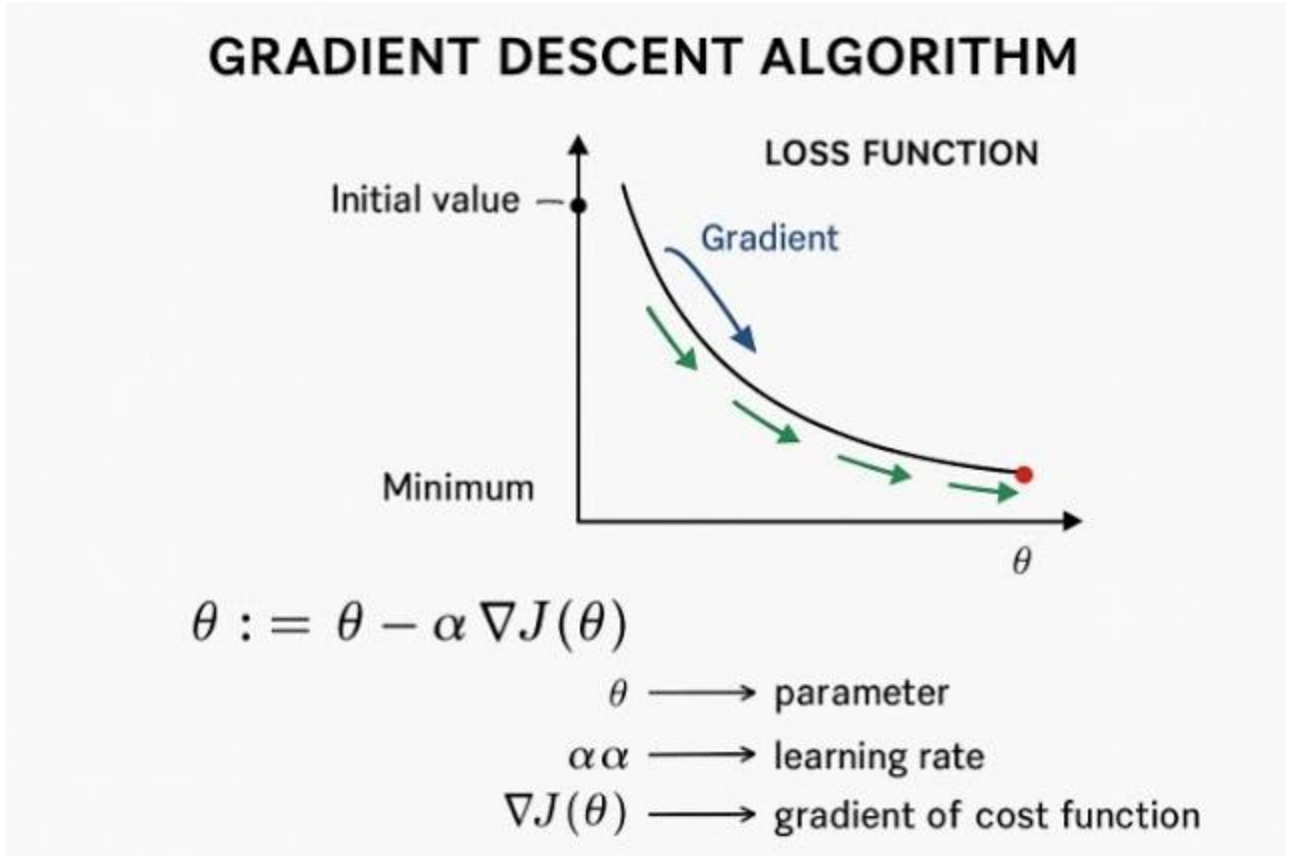
Lancaster University, ProbAI Hub

Based on chapter 6 of the book **Algorithms for Optimization** by Mykel J. Kochenderfer Tim A. Wheeler

Optimization in ML

- The function is the loss function J. The input are the parameters of the model θ.
- We want to find the parameters of the model that minimize the loss function.
- There are a lot of optimization algorithms that use the gradient of the function with respect to the input (e.g. gradient descent, ADAM).
- Today we will focus on how to calculate the gradient of the loss with respect to the parameters of the model.

$$\frac{\partial J}{\partial \theta}$$


GRADIENT DESCENT ALGORITHM
LOSS FUNCTION
Initial value
Gradient
Minimum
θ
$\theta := \theta - \alpha \nabla J(\theta)$
$\theta \longrightarrow$ parameter
$\alpha\alpha \longrightarrow$ learning rate
$\nabla J(\theta) \longrightarrow$ gradient of cost function

First vs Second Order Methods

- First order methods use the first order derivative (or gradient) to perform optimization.
  - Advantage: Cheaper

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k)$$

First vs Second Order Methods

- First order methods use the first order derivative (or gradient) to perform optimization.
  - Advantage: Cheaper
- Second order methods additionally use the second order derivative (or Hessian).
  - Advantage: Faster convergence (sometimes – problem specific).
  - Disadvantage: Calculating the Hessian is very expensive in high-dimensional problems.

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k)$$

$$x_{k+1} = x_k - H^{-1} \nabla f(x_k)$$

First vs Second Order Methods

- First order methods use the first order derivative (or gradient) to perform optimization.
    - Advantage: Cheaper
- Second order methods additionally use the second order derivative (or Hessian).
    - Advantage: Faster convergence (sometimes – problem specific).
    - Disadvantage: Calculating the Hessian is very expensive in high-dimensional problems.

- The gradient determines the direction to travel. The Hessian determines how far to move (instead of a step size).

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k)$$

$$x_{k+1} = x_k - H^{-1} \nabla f(x_k)$$

First vs Second Order Methods

- First order methods use the first order derivative (or gradient) to perform optimization.
  - Advantage: Cheaper
- Second order methods additionally use the second order derivative (or Hessian).
  - Advantage: Faster convergence (sometimes – problem specific).
  - Disadvantage: Calculating the Hessian is very expensive in high-dimensional problems.

- The gradient determines the direction to travel. The Hessian determines how far to move (instead of a step size).

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k)$$

$$x_{k+1} = x_k - H^{-1} \nabla f(x_k)$$

Hessian can be calculated by applying autodiff recursively (e.g. Pytorch, usually we will be approximating it).

Newton’s Method

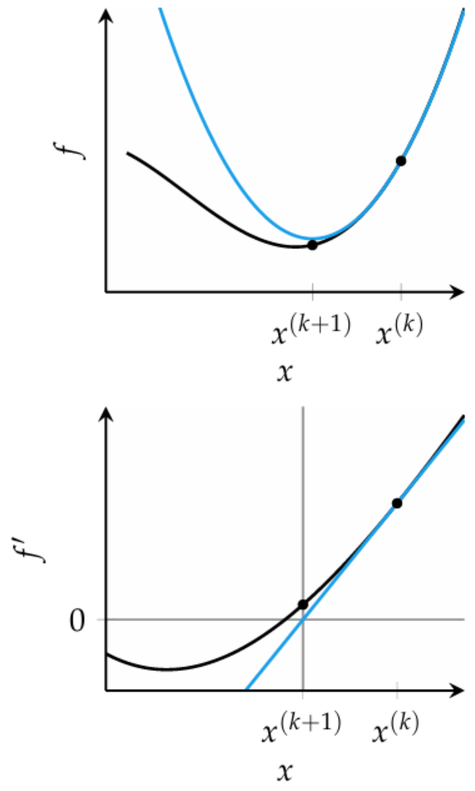
- Consider the 2nd order Taylor expansion of a function.

$$q(x) = f(x^{(k)}) + (x - x^{(k)})f'(x^{(k)}) + \frac{(x - x^{(k)})^2}{2} f''(x^{(k)})$$

- Calculate the derivatives of each side, set them to zero and find the root.

$$\frac{\partial}{\partial x} q(x) = f'(x^{(k)}) + (x - x^{(k)})f''(x^{(k)}) = 0$$

$$x^{(k+1)} = x^{(k)} - \frac{f'(x^{(k)})}{f''(x^{(k)})}$$



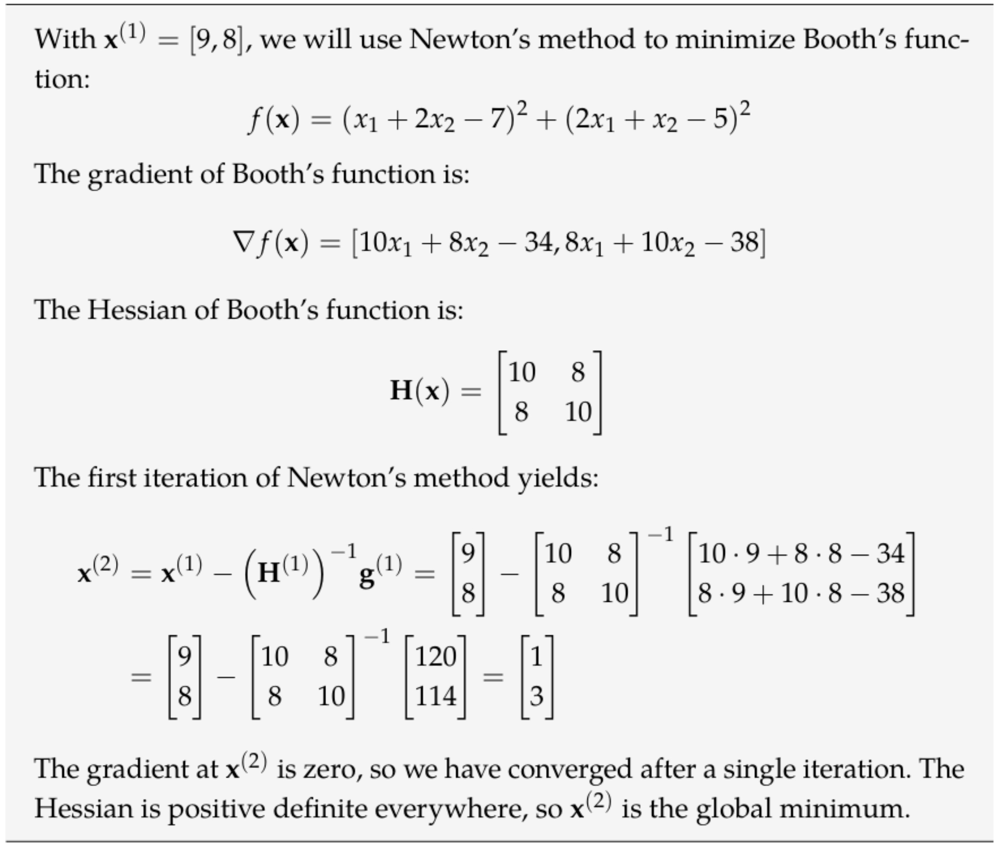
Newton's Method

- Univariate Newton's method:

$$x^{(k+1)} = x^{(k)} - \frac{f'(x^{(k)})}{f''(x^{(k)})}$$

$$x_{k+1} = x_k - H^{-1}\nabla f(x_k)$$

- Intuitively, the step size is set to one over the second derivative. If that is large (high curvature), we take a smaller step. If it is small (flatter function), we take a larger step.
- Instability issues when the second derivative is close to 0. If it is 0, the update is undefined. If it is negative, we are moving to the opposite (from desired) direction.



With $\mathbf{x}^{(1)} = [9, 8]$, we will use Newton's method to minimize Booth's function:

$$f(\mathbf{x}) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$$

The gradient of Booth's function is:

$$\nabla f(\mathbf{x}) = [10x_1 + 8x_2 - 34, 8x_1 + 10x_2 - 38]$$

The Hessian of Booth's function is:

$$\mathbf{H}(\mathbf{x}) = \begin{bmatrix} 10 & 8 \\ 8 & 10 \end{bmatrix}$$

The first iteration of Newton's method yields:

$$\mathbf{x}^{(2)} = \mathbf{x}^{(1)} - \left(\mathbf{H}^{(1)}\right)^{-1}\mathbf{g}^{(1)} = \begin{bmatrix} 9 \\ 8 \end{bmatrix} - \begin{bmatrix} 10 & 8 \\ 8 & 10 \end{bmatrix}^{-1} \begin{bmatrix} 10 \cdot 9 + 8 \cdot 8 - 34 \\ 8 \cdot 9 + 10 \cdot 8 - 38 \end{bmatrix}$$

$$= \begin{bmatrix} 9 \\ 8 \end{bmatrix} - \begin{bmatrix} 10 & 8 \\ 8 & 10 \end{bmatrix}^{-1} \begin{bmatrix} 120 \\ 114 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

The gradient at $\mathbf{x}^{(2)}$ is zero, so we have converged after a single iteration. The Hessian is positive definite everywhere, so $\mathbf{x}^{(2)}$ is the global minimum.

Secant Method

- Approximate the second derivative using first derivatives.
- Cheaper than Newton's method but typically has slower convergence, as it approximates the second derivative.

$$f''(x^{(k)}) \approx \frac{f'(x^{(k)}) - f'(x^{(k-1)})}{x^{(k)} - x^{(k-1)}}$$

$$x^{(k+1)} \leftarrow x^{(k)} - \frac{x^{(k)} - x^{(k-1)}}{f'(x^{(k)}) - f'(x^{(k-1)})} f'(x^{(k)})$$

When does Newton’s Method work/fail?

- Univariate Newton’s method:

$$x^{(k+1)} = x^{(k)} - \frac{f'(x^{(k)})}{f''(x^{(k)})}$$

$$x_{k+1} = x_k - H^{-1}\nabla f(x_k)$$

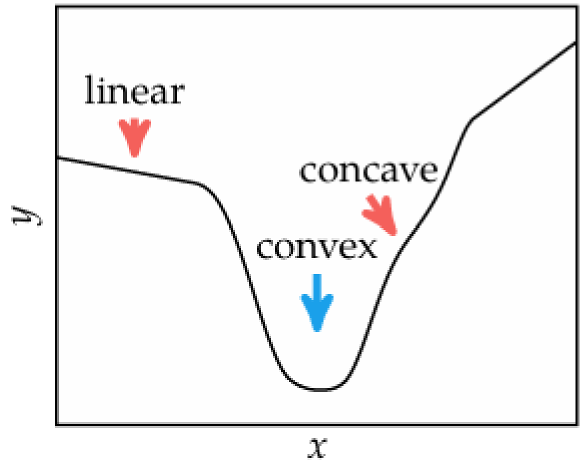
- Newton’s method assumes that locally the function behaves like a quadratic. Therefore, it performs well when a quadratic approximation is a good fit, and poorly otherwise.
- Smooth functions can (usually) be approximated well by a quadratic near the minimum (as they are typically convex).
- In more linear or concave regions it is better to use gradient descent.

Levenberg-Marquardt Algorithm

- Interpolation between gradient descent and approximate Newton updates.
- After each calculation, check if the update decreases the value of f. If it does, decrease δ. Otherwise, increase δ and discard the new value.
- The idea is that if we are far from a minimum we want to use gradient descent, whereas if we are close to a minimum we want to incorporate second order information.

$$\mathbf{x}' = \mathbf{x} - (\mathbf{H} + \delta\mathbf{I})^{-1}\mathbf{g}$$

$$\mathbf{x}' = \mathbf{x} - (\mathbf{H} + \delta\mathrm{diag}(\mathrm{diag}(\mathbf{H})))^{-1}\mathbf{g}$$



Quasi-Newton Methods

- The secant method is only for the univariate case. Quasi-Newton methods approximate the inverse Hessian (inverting the Hessian is expensive).
- Usually set Q^1 to the identity matrix.
- There are many Quasi-Newton methods, we will talk about the Davidon-Fletcher-Powell (DFP) method and the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method.
- For ease of notation, we define

$$\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} - \alpha^{(k)}\mathbf{Q}^{(k)}\mathbf{g}^{(k)}$$

Quasi-Newton Methods

- The secant method is only for the univariate case. Quasi-Newton methods approximate the inverse Hessian (inverting the Hessian is expensive).
- Usually set Q^1 to the identity matrix.
- There are many Quasi-Newton methods, we will talk about the Davidon-Fletcher-Powell (DFP) method and the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method.
- For ease of notation, we define

$$\boldsymbol{\gamma}^{(k+1)} \equiv \mathbf{g}^{(k+1)} - \mathbf{g}^{(k)}$$
$$\boldsymbol{\delta}^{(k+1)} \equiv \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$$

$$\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} - \alpha^{(k)}\mathbf{Q}^{(k)}\mathbf{g}^{(k)}$$

We now have a step size!

DFP Method

- The DFP method uses:

$$\mathbf{Q} \leftarrow \mathbf{Q} - \frac{\mathbf{Q}\boldsymbol{\gamma}\boldsymbol{\gamma}^\top\mathbf{Q}}{\boldsymbol{\gamma}^\top\mathbf{Q}\boldsymbol{\gamma}} + \frac{\boldsymbol{\delta}\boldsymbol{\delta}^\top}{\boldsymbol{\delta}^\top\boldsymbol{\gamma}}$$

- The update rule keeps Q symmetric and positive definite.

$$\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} - \alpha^{(k)}\mathbf{Q}^{(k)}\mathbf{g}^{(k)}$$

$$\boldsymbol{\gamma}^{(k+1)} \equiv \mathbf{g}^{(k+1)} - \mathbf{g}^{(k)}$$
$$\boldsymbol{\delta}^{(k+1)} \equiv \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$$

BFGS Method

- The BFGS method uses:

$$\mathbf{Q} \leftarrow \mathbf{Q} - \left(\frac{\boldsymbol{\delta}\boldsymbol{\gamma}^\top \mathbf{Q} + \mathbf{Q}\boldsymbol{\gamma}\boldsymbol{\delta}^\top}{\boldsymbol{\delta}^\top \boldsymbol{\gamma}}\right) + \left(1 + \frac{\boldsymbol{\gamma}^\top \mathbf{Q}\boldsymbol{\gamma}}{\boldsymbol{\delta}^\top \boldsymbol{\gamma}}\right)\frac{\boldsymbol{\delta}\boldsymbol{\delta}^\top}{\boldsymbol{\delta}^\top \boldsymbol{\gamma}}$$

$$\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} - \alpha^{(k)}\mathbf{Q}^{(k)}\mathbf{g}^{(k)}$$

$$\boldsymbol{\gamma}^{(k+1)} \equiv \mathbf{g}^{(k+1)} - \mathbf{g}^{(k)}$$
$$\boldsymbol{\delta}^{(k+1)} \equiv \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$$

- Numerically more stable, widely used.

- Very expensive when x has more than 1000 dimension (in Deep Learning parameters of neural networks).

Limited-Memory BFGS Method

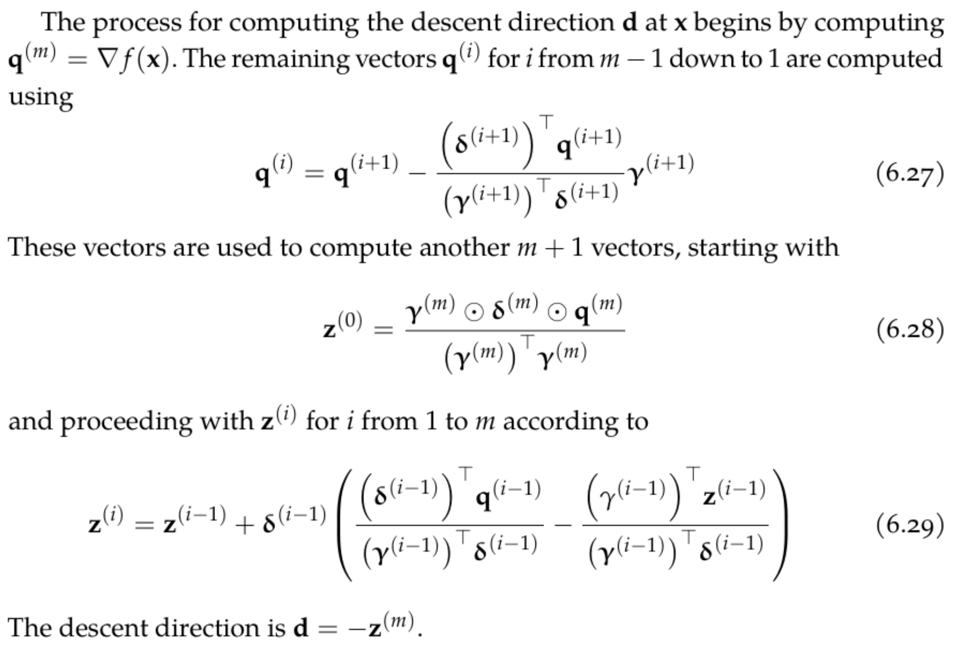
- Storing and updating the inverse Hessian is very expensive in high-dimensional problems.
- L-BFGS approximates BFGS by storing the last m (e.g. 5-20) values of γ and δ. From these pairs we can estimate $\mathbf{Q}^{(k)}\mathbf{g}^{(k)}$

L-BFGS is used in Deep Learning!

$$\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} - \alpha^{(k)}\mathbf{Q}^{(k)}\mathbf{g}^{(k)}$$

$$\boldsymbol{\gamma}^{(k+1)} \equiv \mathbf{g}^{(k+1)} - \mathbf{g}^{(k)}$$

$$\boldsymbol{\delta}^{(k+1)} \equiv \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$$



The process for computing the descent direction $\mathbf{d}$ at $\mathbf{x}$ begins by computing $\mathbf{q}^{(m)} = \nabla f(\mathbf{x})$. The remaining vectors $\mathbf{q}^{(i)}$ for i from $m-1$ down to 1 are computed using

$$\mathbf{q}^{(i)} = \mathbf{q}^{(i+1)} - \frac{\left(\boldsymbol{\delta}^{(i+1)}\right)^\top \mathbf{q}^{(i+1)}}{\left(\boldsymbol{\gamma}^{(i+1)}\right)^\top \boldsymbol{\delta}^{(i+1)}}\boldsymbol{\gamma}^{(i+1)} \tag{6.27}$$

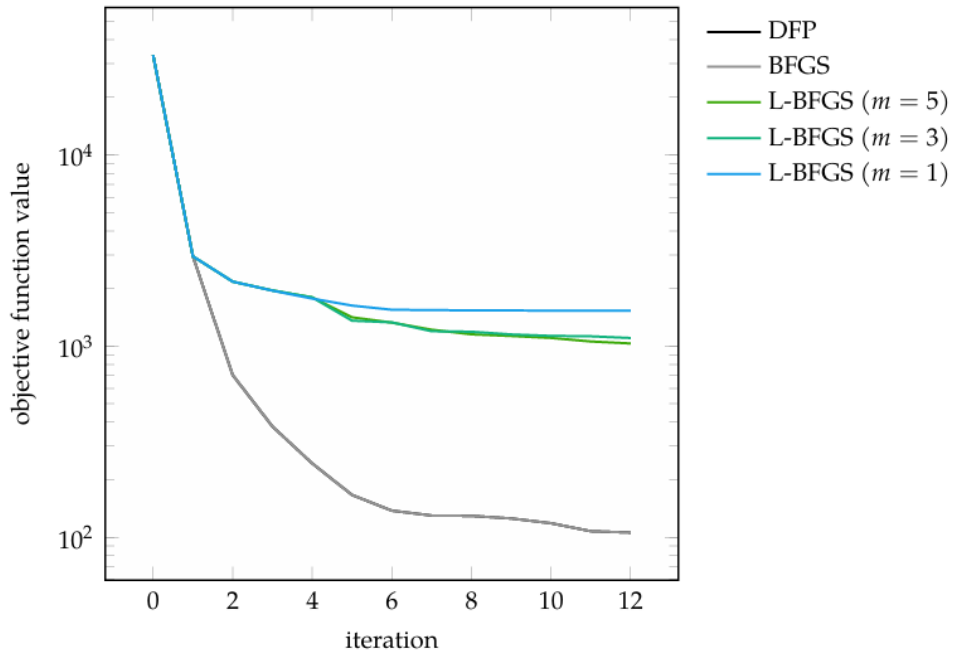
These vectors are used to compute another $m+1$ vectors, starting with

$$\mathbf{z}^{(0)} = \frac{\boldsymbol{\gamma}^{(m)} \odot \boldsymbol{\delta}^{(m)} \odot \mathbf{q}^{(m)}}{\left(\boldsymbol{\gamma}^{(m)}\right)^\top \boldsymbol{\gamma}^{(m)}} \tag{6.28}$$

and proceeding with $\mathbf{z}^{(i)}$ for i from 1 to m according to

$$\mathbf{z}^{(i)} = \mathbf{z}^{(i-1)} + \boldsymbol{\delta}^{(i-1)}\left(\frac{\left(\boldsymbol{\delta}^{(i-1)}\right)^\top \mathbf{q}^{(i-1)}}{\left(\boldsymbol{\gamma}^{(i-1)}\right)^\top \boldsymbol{\delta}^{(i-1)}} - \frac{\left(\gamma^{(i-1)}\right)^\top \mathbf{z}^{(i-1)}}{\left(\boldsymbol{\gamma}^{(i-1)}\right)^\top \boldsymbol{\delta}^{(i-1)}}\right) \tag{6.29}$$

The descent direction is $\mathbf{d} = -\mathbf{z}^{(m)}$.



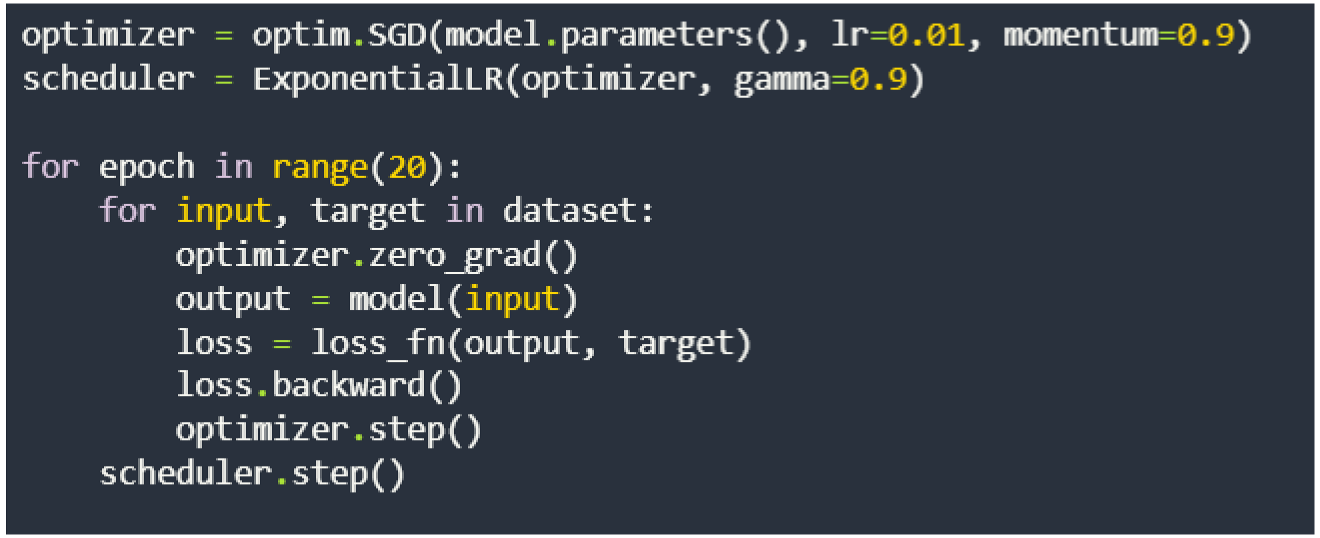
Aside – Tricks to improve training in DL (1)

- Raw parameter updates can be noisy (especially in diffusion models).
- Exponential Moving Average (EMA); Update the weights according to a moving average. Typical values of β are range between 0.99 and 0.9999.
- During the first few updates (or epochs) EMA is not applied! This is called the warm up phase.

$$w = \beta \cdot w_{old} + (1 - \beta) \cdot w_{new}$$

Aside – Tricks to improve training in DL (2)

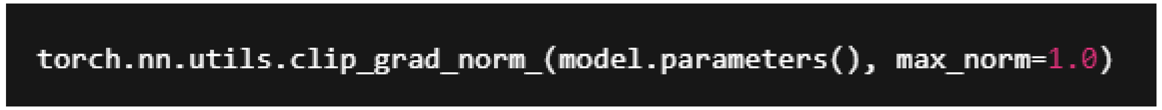
- Learning rate schedulers: every few epochs change the learning rate. Typically, we want a higher learning rate initially.



```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
scheduler = ExponentialLR(optimizer, gamma=0.9)

for epoch in range(20):
    for input, target in dataset:
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
    scheduler.step()
```

- Gradient clipping: Clip the values of the gradients between two values so that the updates are smoother.



```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
```

Thank you for listening! Questions?

Go with the flow
Achievement unlocked: Diffusion models can spell now!
they left me here to rust and rot away like all the others
Astronaut & Piggy: Buddies for Life

Stable Diffusion 3 (uses Flow model)