Deep Learning Theory and Applications

Variations on SGD

Yale

CPSC/CBB/AMTH 663

CPSC 452

Outline

1. Learning vs. pure optimization
2. 2nd Order Methods
3. Momentum methods
4. Approximate 2nd order methods
5. Adaptive learning rates

Gradient-based optimization revisited

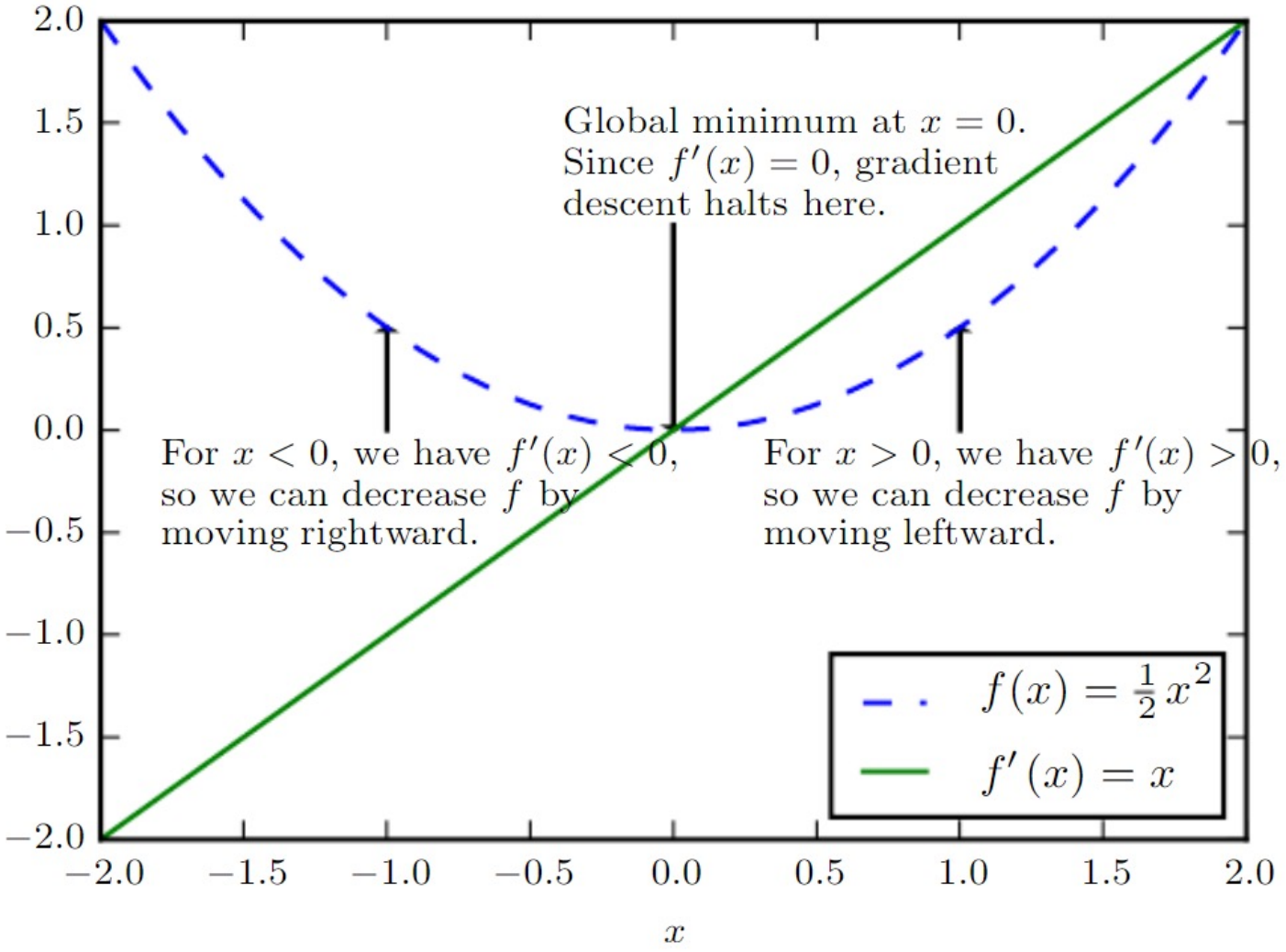
- Deep learning involves optimization of some sort
  - Optimization: the task of either minimizing or maximizing some function $f(x)$ by altering x
  - Typically, we focus on minimizing
    - Maximization can be accomplished by minimizing $-f(x)$
- The function we want to minimize or maximize is the ***objective function***, or ***criterion***
  - When minimizing, may also be referred to as the ***cost function***, ***loss function***, or ***error function***
- The value that minimizes or maximizes a function is often written as

$$x^* = \arg\min f(x)$$

Gradient-based optimization revisited

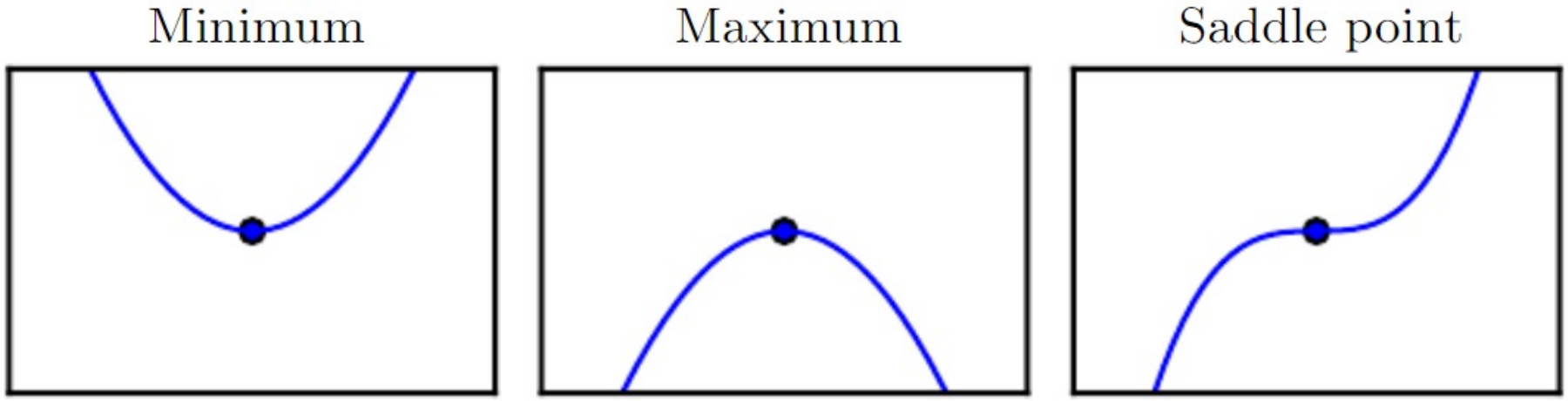
- The derivative $\frac{df}{dx}$ (denoted $f'(x)$) gives the slope of f at the point x
  - I.e., it specifies how to scale a small change in input to obtain corresponding change in the output: $f(x+\epsilon) \approx f(x) + \epsilon f'(x)$
  - Thus the derivative tells us how to change x to make small changes (e.g. decreases if minimizing) to f
- Gradient descent is based on this idea
  - E.g. $f\left(x - \epsilon \text{sign}\left(f'(x)\right)\right) < f(x)$ for small enough ϵ
  - We can reduce $f(x)$ by moving x in small steps with the opposite sign of the derivative

Gradient-based optimization revisited



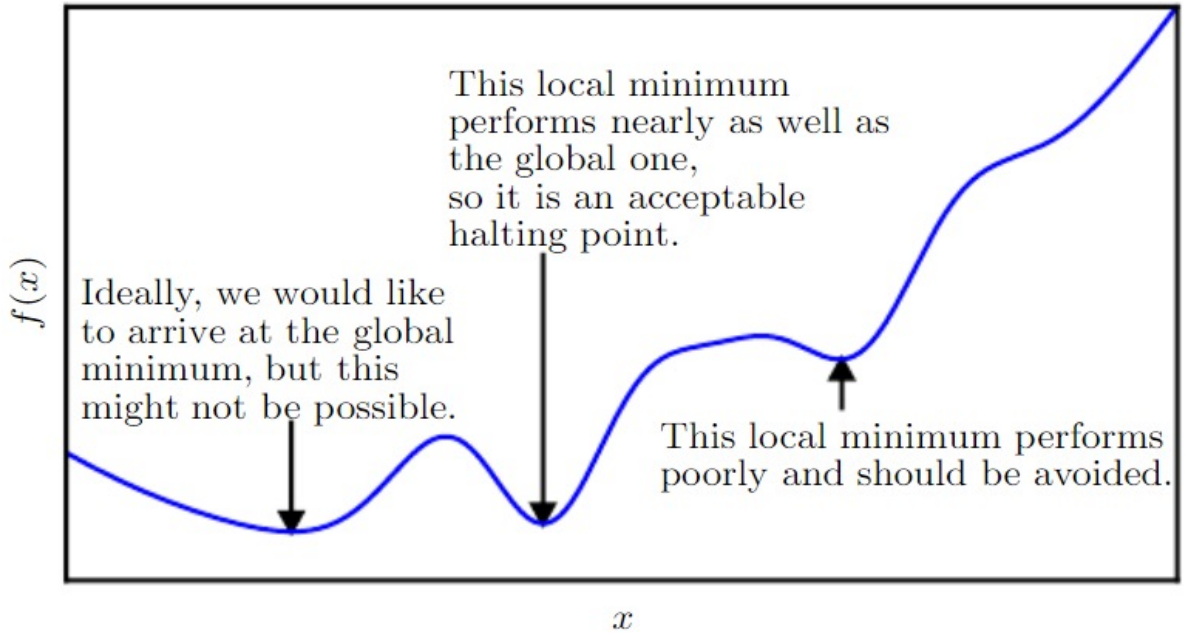
Gradient-based optimization revisited

- What does $f'(x) = 0$ mean?
  - Corresponding points are ***critical points*** or ***stationary points***
  - ***Local minimum*** is a point where $f(x)$ is lower than all neighboring points
  - ***Local maximum*** is a point where $f(x)$ is higher than all neighboring points
  - ***Saddle points*** are neither



Gradient-based optimization revisited

- ***Global minimum***: a point that obtains the absolute lowest value of $f(x)$
  - May be multiple global minima
- Local minima may not be globally optimal
- In deep learning, we often optimize functions with many nonoptimal local minima and many saddle points



Are local minima a problem?

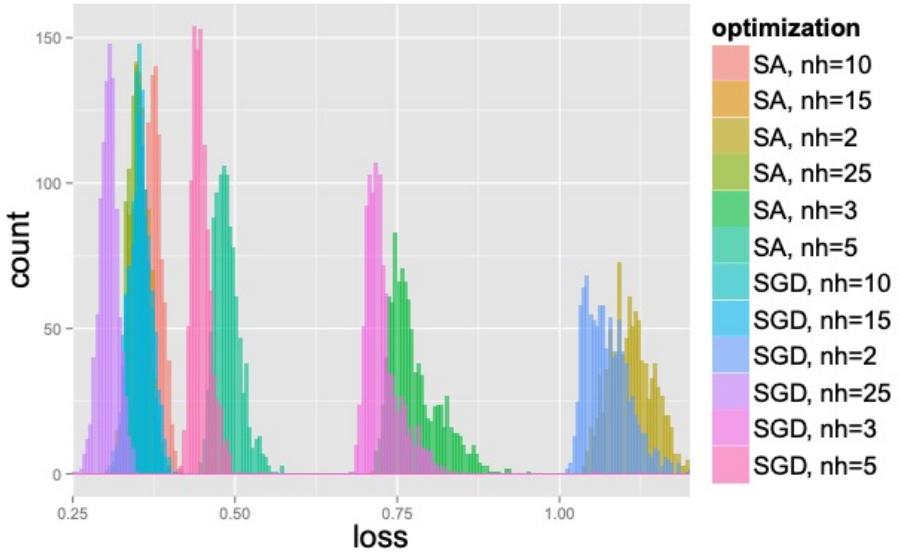


Figure 6: Test loss distributions for SGD and SA for different numbers of hidden units (nh).

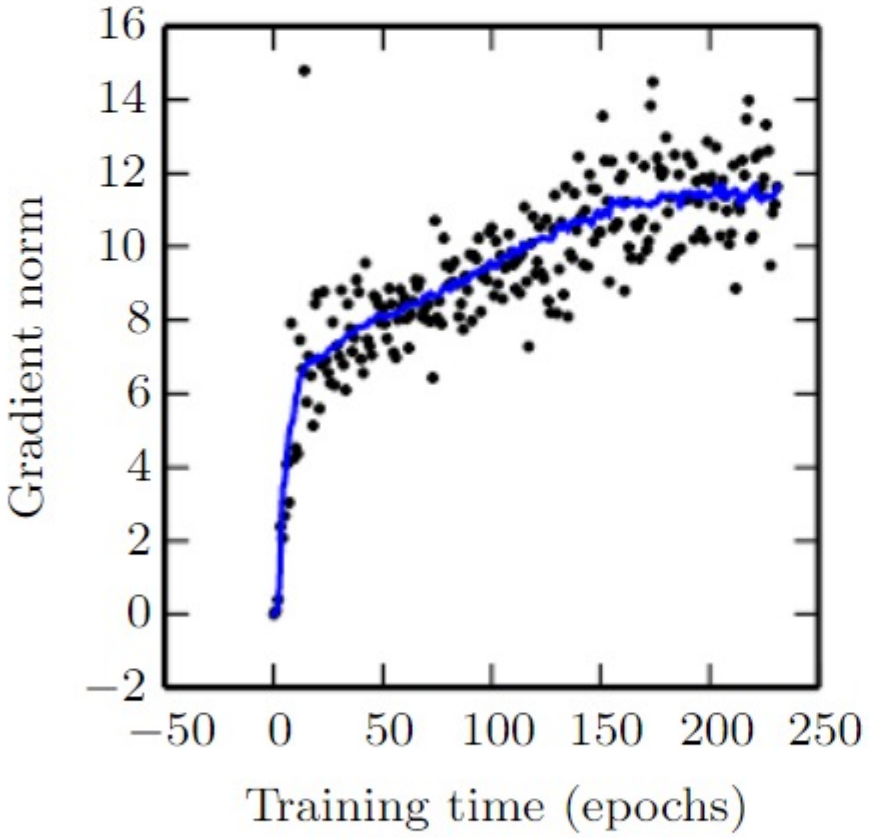
- Results show that in large neural networks most local minima are close to the global minima! [Choromanska et al. 2014]
- Why could this be?

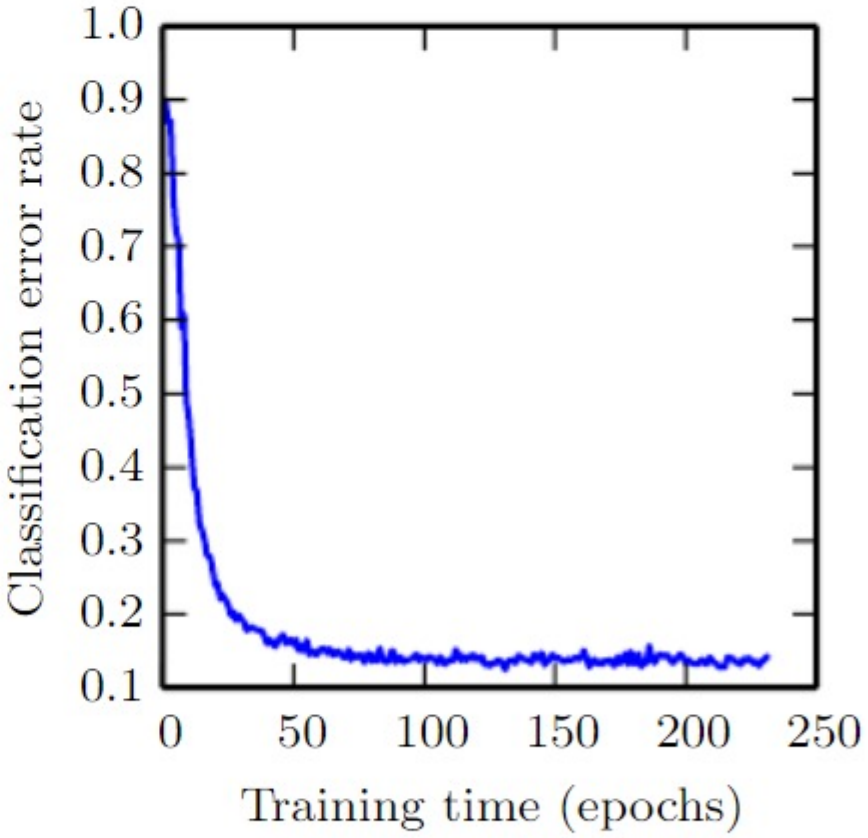
Are local minima a problem?

- Nearly any deep model is guaranteed to have a large number of local minima
  - This is due to weight space symmetry (we could modify the neural network by swapping order of hidden nodes)
  - Additionally, invariance to scale can result in an uncountably infinite number of local minima (increase all weights by x 10)
  - However, they all have the same cost
- It's possible to construct a small network with local minima higher than the global minimum
- But current research suggests that in large networks, local minima typically have low cost function values
  - The probability of finding a non-optimal local minima diminishes with the size of the network.

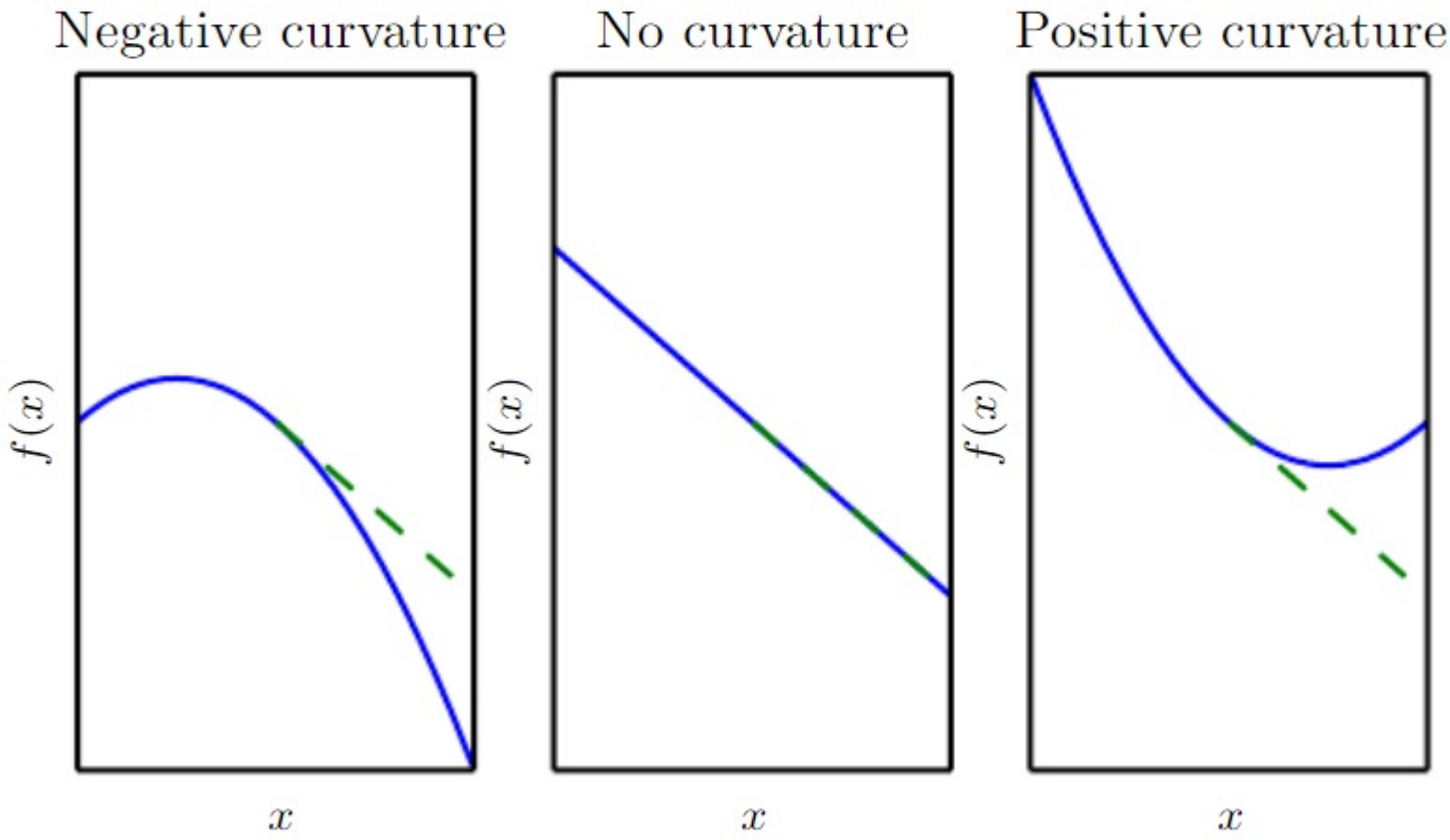
Can we *actually find* local minima?

- Gradient is increasing and not decreasing during training, but cost is not improving.





Curvature



- Can we better approximate the actual change in the cost based on curvature?

2^{nd} Order Methods

Jacobian

- In normal gradient descent we compute a matrix of partial derivatives called the Jacobian

$$\mathbf{J} = \frac{d\mathbf{f}}{d\mathbf{x}} = \left[\frac{\partial \mathbf{f}}{\partial x_1} \cdots \frac{\partial \mathbf{f}}{\partial x_n} \right] = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Hessian

- The Hessian is a generalization of this to 2nd order

$$\mathbf{H_F} = \nabla \mathbf{J} = \nabla \mathbf{F} \nabla^T = \frac{d^2 \mathbf{F}}{d\mathbf{x}^2} = \begin{bmatrix} \frac{\partial^2 f_1}{\partial {x_1}^2} & \cdots & \frac{\partial^2 f_1}{\partial x_n^2} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f_m}{\partial {x_1}^2} & \cdots & \frac{\partial^2 f_m}{\partial {x_n}^2} \end{bmatrix}$$

Hessian and Multidimensional Curvature

- The Hessian matrix tells us about curvature in every direction
- Eigendecomposition of the Hessian results in eigenvalues
- If they are ALL positive then the curvature is positive everywhere (local maxima)
- If they are mixed then it is a saddle point

Condition Number

- The condition number of the Hessian measures how much the second derivatives differ from each other
- Condition number is ratio of the largest eigenvalue to the smallest
- When the Hessian has high condition number, gradient descent performs poorly
  - The derivative increases rapidly, while in another direction, it increases slowly
  - Poor condition number also makes choosing a good step size difficult
  - The step size must be small enough to avoid overshooting
- Solution: Hessian matrix needs to be used for minima search
  - The simplest method for doing so is known as Newton's method

Taylor expansion

- Consider a cost function $C(w)$ of many variables $w = w_1, w_2, \ldots$
- Taylor series expansion:

$$C(w + \Delta w) = C(w) + \sum_j \frac{\partial C}{\partial w_j} \Delta w_j + \frac{1}{2} \sum_{jk} \Delta w_j \frac{\partial^2 C}{\partial w_j \partial w_k} \Delta w_k + \cdots$$

- Rewrite as

$$C(w + \Delta w) = C(w) + \nabla C \cdot \Delta w + \frac{1}{2} \Delta w^T H \Delta w + \cdots$$

Taylor expansion

- The second derivatives (i.e. the Hessian) give a measure of the curvature of the cost function
- Approximate the Taylor series expansion:

$$C(w + \Delta w) \approx C(w) + \nabla C \cdot \Delta w + \frac{1}{2} \Delta w^T H \Delta w$$

- Minimize this (decrease C as much as possible):

$$\Delta w = -H^{-1} \nabla C$$

- If the approximation is good, we expect a large decrease in the cost function

Newton's Method

Possible algorithm for minimizing cost

1. Choose starting point w
2. Update $w' \leftarrow w - H^{-1}\nabla C$
   - Hessian and gradient are computed at w
3. Repeat step 2 until convergence

- Due to the approximation, better to take small steps:

$$w' \leftarrow w - \eta H^{-1}\nabla C$$

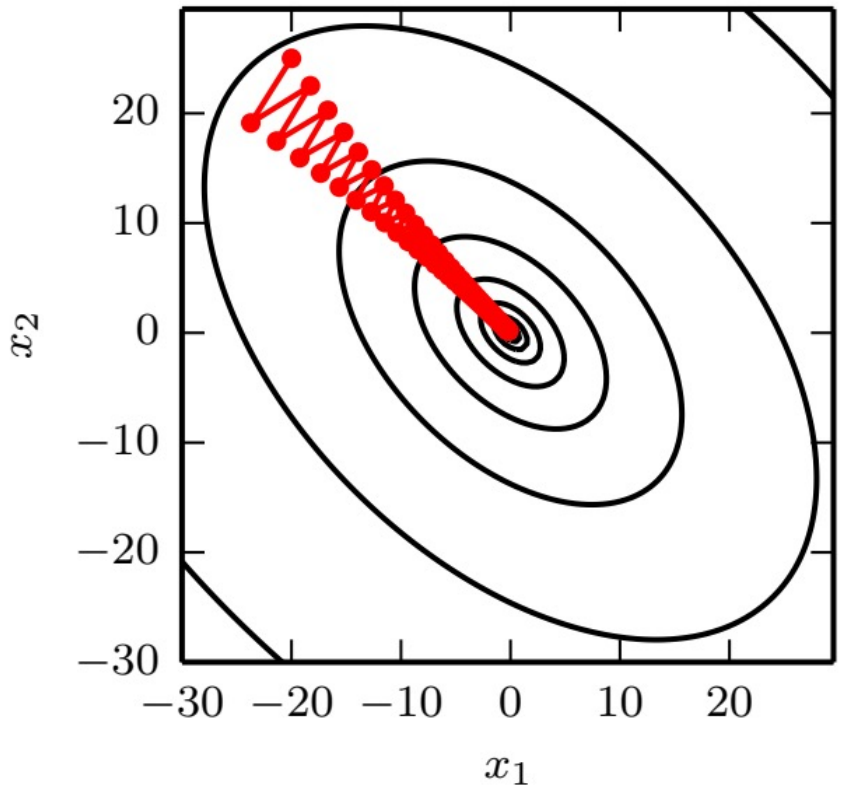
- This approach also called Hessian optimization
  - It is a ***second-order*** optimization algorithm
  - Gradient descent is a ***first-order*** optimization algorithm

Advantages of Hessian methods

Theoretical and empirical results show Hessian methods converge in fewer steps than standard GD

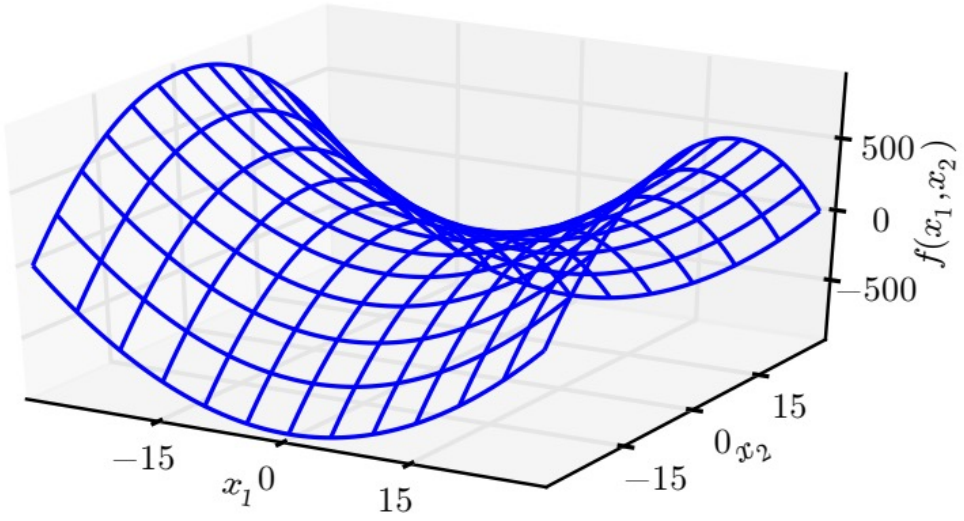
- A gradient descent step of $-\eta\nabla C$ adds to the cost: $\frac{1}{2}\eta^2(\nabla C)^T H\nabla C - \eta(\nabla C)^T\nabla C$

- Backpropagation can be modified to compute the Hessian



Disadvantages of Hessian methods

- Difficult to apply in practice
  - Suppose we have 10^7 weights and biases
  - H would have 10^{14} entries
  - Computing $H^{-1}\nabla C$ in practice would be very difficult
  - There are variations which get around this
- Newton's method converges quickly to critical points
  - A problem when near a saddle point
  - Using a learning rate can potentially help



Are saddle points a problem?

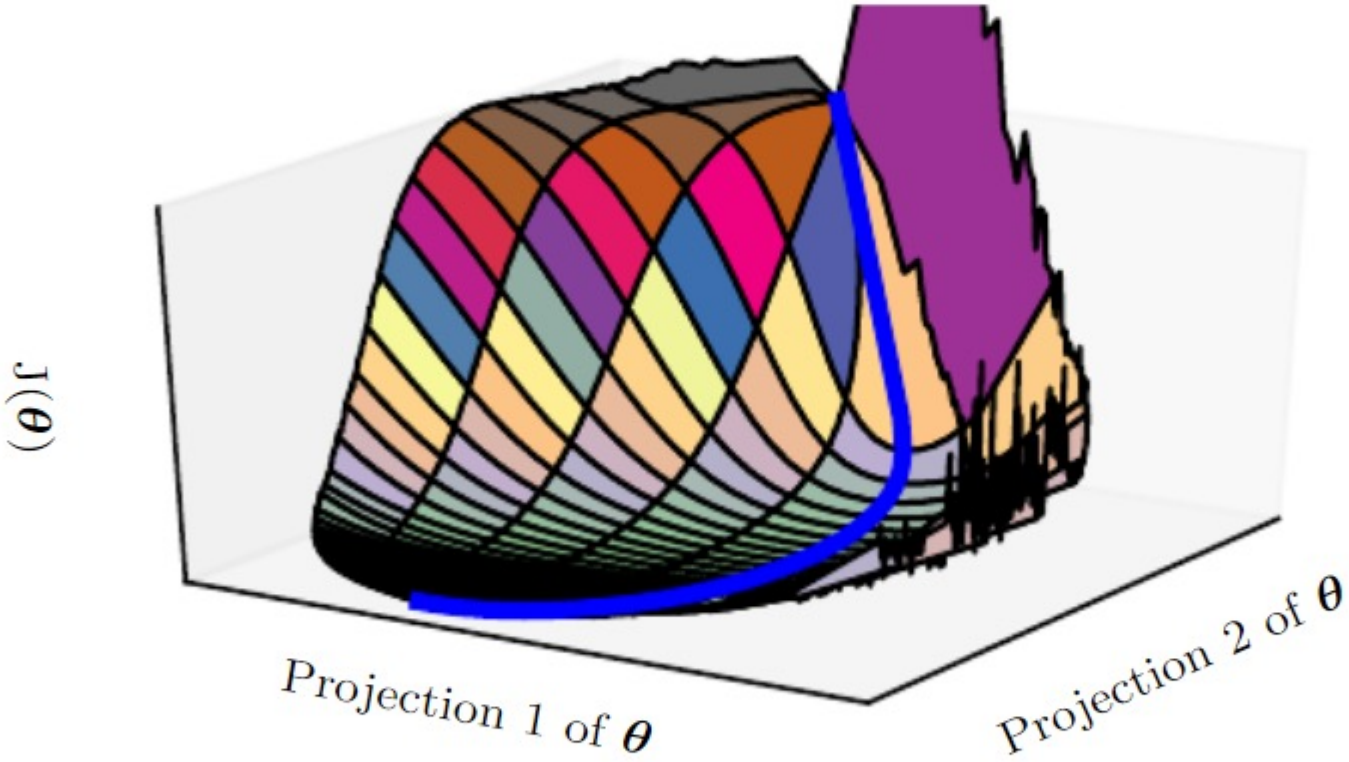
- In high-dimensional nonconvex functions, local minima are rare compared to saddle points
  - For a random function $f: \mathbb{R}^d \to \mathbb{R}$, the expected ratio of the number of saddle points to local minima grows exponentially with d
- Intuition: consider the Hessian matrix
  - At a local minima, the Hessian has only positive eigenvalues
  - At a saddle point, the Hessian has both positive and negative eigenvalues
  - Suppose the sign of each eigenvalue is determined by a coinflip
    - Heads ⇒ positive, tails ⇒ negative
  - It is exponentially unlikely that d coin tosses will all be heads

Are saddle points a problem?

- This is true for random functions; what about neural networks?
- It's been shown theoretically that shallow, linear autoencoders have global minima and saddle points, but no local minima with higher cost than the global minimum
- It's been shown experimentally that real neural networks have loss functions with many high-cost saddle points
  - See Goodfellow et al., section 8.2.3 for references

Are saddle points a problem?

- How do these saddle points affect training?
- Gradient descent seems to escape saddle points despite low gradients



Are saddle points a problem?

- Why does gradient descent escape?
- Gradient descent simply tries to move downhill
  - Not designed to explicitly find a critical point
- In contrast, Newton's method is designed to solve for a point where the gradient is zero
  - Thus without modification it can jump to a saddle point
- A saddle-free Newton method has been developed that improves on this (Dauphin et al., 2014)
  - Scalability is still an issue

Approximate 2nd Order Methods

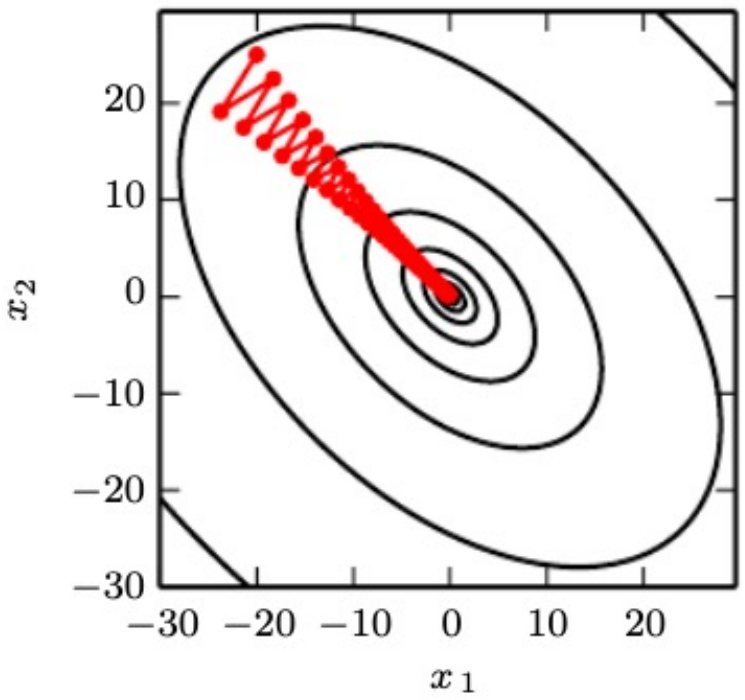
Newton's method revisited

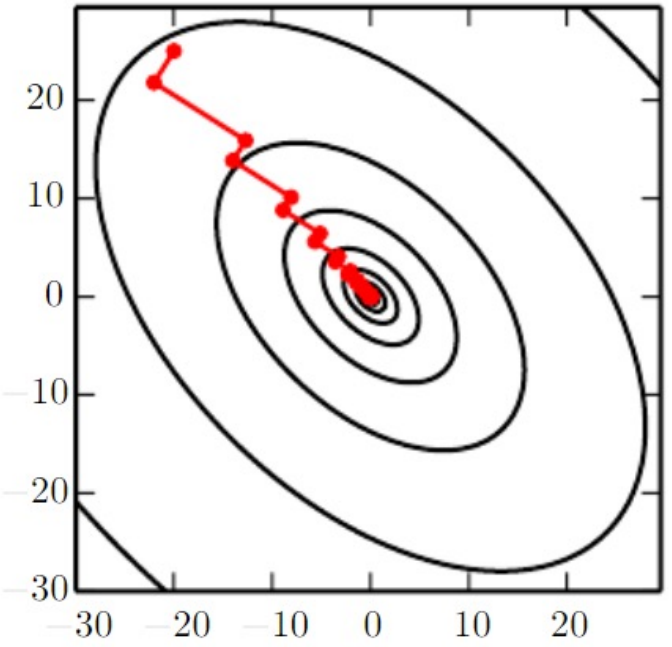
- Newton's method is appropriate only when the Hessian is positive definite (all eigenvalues > 0)
- In deep learning, the objective function is nonconvex and may contain many saddle points
  - If eigenvalues of the Hessian are not all positive near a saddle point, Newton's method can jump to the saddle point
- We can avoid this by regularizing the Hessian

$$w' = w - (H + \alpha I)^{-1} \nabla C$$

  - I is the identity matrix and α is a positive constant
  - Works well as long as negative eigenvalues are close to zero
  - As α increases, the αI term dominates, which may result in very small step sizes
  - Still have to invert a large matrix at each training iteration

Line Search vs Conjugate Gradients





The gradient doesn't point in the right direction when there is high curvature

Need to search directions conjugate to the gradient g, pick u s.t. uHg^t = 0

Broyden-Fletcher-Goldfarb-Shanno algorithm

The BFGS algorithm

- Recall Newton's method update step:

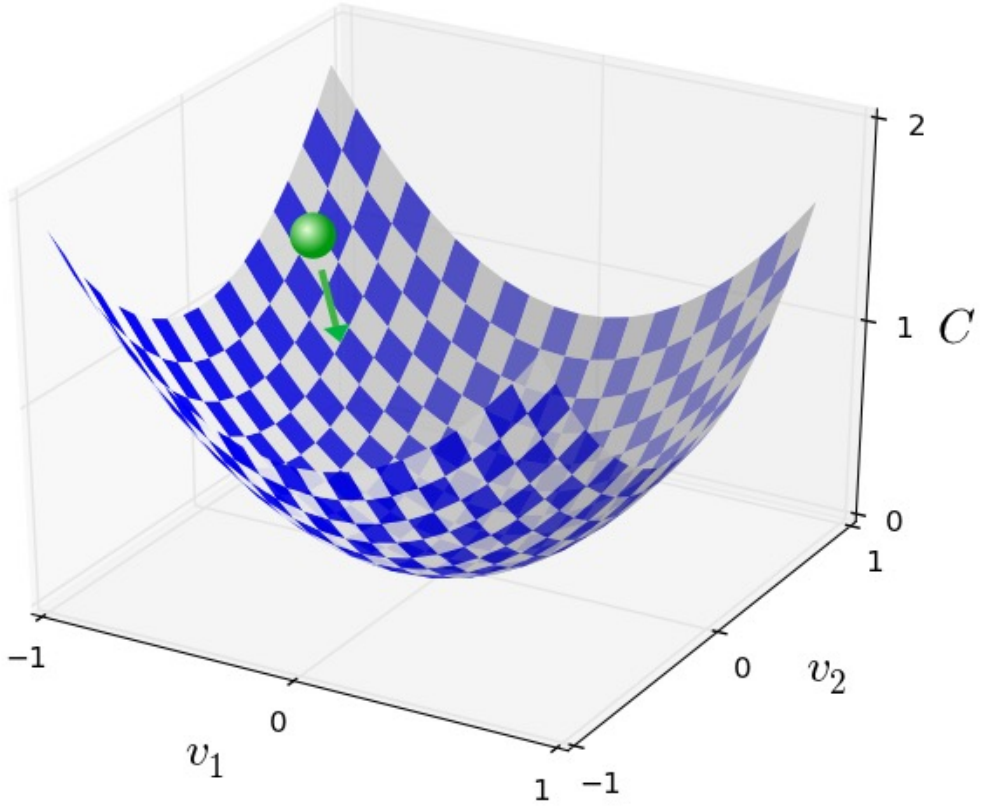
$$w' = w - H^{-1}\nabla C$$

- BFGS attempts to approximate H^{-1} with a matrix M that is iteratively refined by low-rank updates
- Line search is performed to select the learning rate/step size η
  - More robust than conjugate gradients
- However, M must still be stored in memory which can be huge
  - Some lower memory adaptations exist

Momentum Methods

Momentum-based gradient descent

- Momentum-based GD incorporates information about how the gradient is changing w/o requiring large matrices of 2nd derivatives
- Momentum technique modifies GD to make it more similar to the physics



Momentum-based gradient descent

Two modifications to GD:

1. Change the “velocity” instead of the “position”
2. Introduce a friction term which gradually reduces velocity

Momentum-based gradient descent

- Introduce velocity variables $v = v_1, v_2, \ldots$ for each corresponding w_j variable
- New update rule:

$$v \rightarrow v' = \mu v - \eta \nabla C$$
$$w \rightarrow w' = w + v'$$

  - μ is a hyper-parameter that controls the damping/friction of the system
- Consider $\mu = 1$ (no friction)
  - ∇C modifies the velocity v which controls rate of change of w
  - I.e. we're building up velocity by adding gradient terms
  - If the gradient is in roughly the same direction through several rounds, we move more quickly

Momentum-based gradient descent

$$v \rightarrow v' = \mu v - \eta \nabla C$$
$$w \rightarrow w' = w + v'$$

- However, if we reach the bottom or the gradient changes, we could move in the wrong direction
  - μ controls this
  - $\mu = 0 \Rightarrow$ lot of friction and velocity can't build up (standard GD)
  - In practice, choose $0 < \mu < 1$ via validation
- μ is referred to as the ***momentum coefficient***

Momentum-based gradient descent

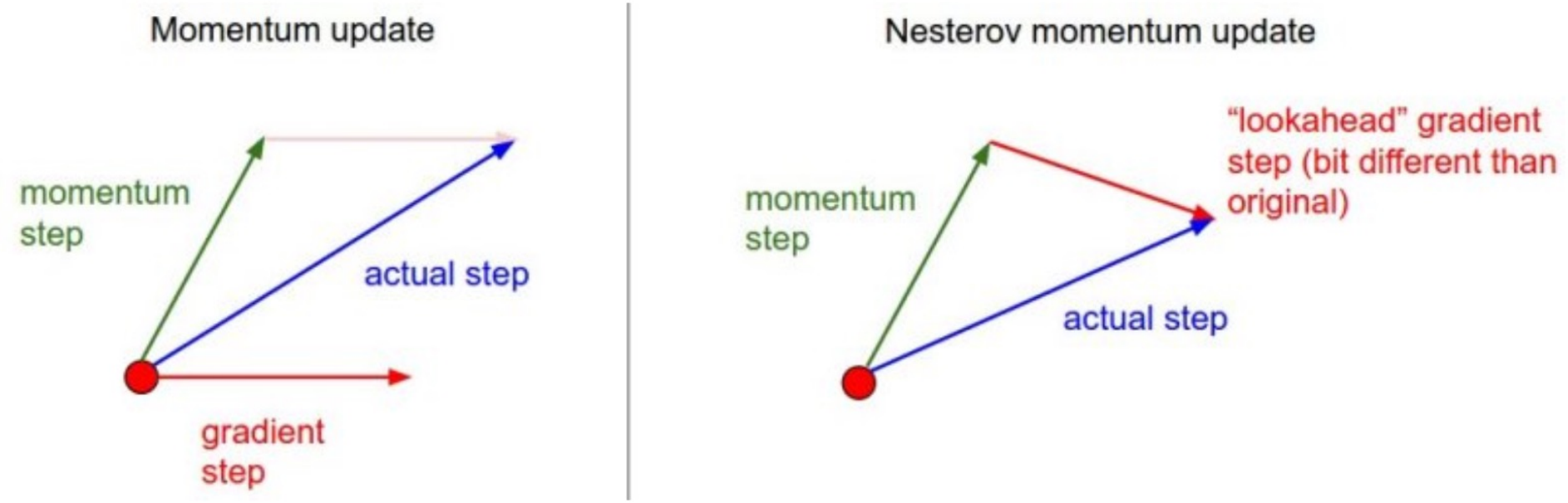
- If ∇C is the same always, the terminal velocity is
$$\frac{\eta \|\nabla C\|}{1-\mu}$$
- It can be helpful to think of the momentum coefficient in this context
  - Typical values include 0.5, 0.9, and 0.99

Advantages of momentum technique

- Relatively simple to modify GD to incorporate momentum
  - Backpropagation and mini-batch sampling is the same
  - Much more computationally friendly than the Hessian technique
- We obtain some of the advantages of the Hessian technique

- Because of this, the momentum technique is commonly used to speed up learning

Nesterov Momentum



Nesterov momentum

New procedure

1. Apply interim update $w' \leftarrow w + \mu v$
2. Compute gradient ∇C at w'
3. Compute velocity update $v \leftarrow \mu v - \eta \nabla_{w'} C$
4. Apply update $w \leftarrow w + v$

- This helps because while the gradient term always points in the right direction
- If the momentum term points in the wrong direction or overshoots, the gradient can still "go back" and correct it in the same update step.

Adaptive Learning Rates

AdaGrad, RMSProp, Adam

Adaptive learning rates

- The learning rate is one of the hardest hyper-parameter to tune
- Cost functions are often highly sensitive to some directions in parameter space and insensitive to others
  - In this case, it makes sense to have separate learning rates for each parameter
- Not more parameters!
- Set these parameters adaptively
- Early heuristic: delta-bar-delta algorithm (Jacobs, 1988)
  - If the partial derivative of the loss wrt to a parameter remains the same sign, then increase the learning rate
  - If the sign changes, decrease the learning rate
  - Only applies to batch optimization

AdaGrad

- Adapts the learning rates by scaling them inversely proportional to the square root of the sum of all the historical squared values of the gradient

1. Compute gradient ∇C from mini-batch
2. Accumulate squared gradient: $r \leftarrow r + \nabla C \odot \nabla C$
3. Compute update: $w \leftarrow w - \frac{\eta}{\delta + \sqrt{r}} \odot \nabla C$
   - Division and square root are applied element-wise to r
   - η is the global learning rate and δ is a small constant (e.g. 10^{-7}) for numerical stability

AdaGrad

- Adapts the learning rates by scaling them inversely proportional to the square root of the sum of all the historical squared values of the gradient
- Parameters with largest partial derivative have rapid decrease in their learning rate
- Parameters with small partial derivatives have relatively small decrease in their learning rate
  - Net effect: greater progress in the more gently sloped directions
- In convex optimization, AdaGrad has nice theoretical properties
- But for neural networks, AdaGrad can result in premature and excessive decrease in the effective learning rate
  - Performs well for some but not all deep learning models

RMSProp

- Modifies AdaGrad to perform better in nonconvex settings
- Changes the gradient accumulation into an exponentially weighted moving average

1. Compute gradient ∇C from mini-batch
2. Accumulate squared gradient:
$$r \leftarrow \rho r + (1-\rho)\nabla C \odot \nabla C$$
3. Compute update: $w \leftarrow w - \frac{\eta}{\delta + \sqrt{r}} \odot \nabla C$
   - Division and square root are applied element-wise to r
   - η is the global learning rate and δ is a small constant (e.g. 10^{-7}) for numerical stability

RMSProp

- AdaGrad is designed to converge rapidly for a convex function
- When learning in a neural network, the learning trajectory may pass through many different structures and eventually arrive at a locally convex region
- AdaGrad shrinks the learning rate according to the entire history of the trajectory
  - Learning rate may be too small before arriving at the convex structure
- RMSProp use an exponentially decaying average to throw out history from the extreme past
  - Thus it converges quickly after finding a convex bowl
    - Similar to initializing AdaGrad within that bowl

RMSProp with Nesterov Momentum

1. Compute interim update $w' \leftarrow w + \mu v$
2. Compute gradient $\nabla C(w')$ from mini-batch at w'
3. Accumulate squared gradient:
$$r \leftarrow \rho r + (1-\rho)\nabla C(w') \odot \nabla C(w')$$
4. Update velocity: $v \leftarrow \mu v - \frac{\eta}{\sqrt{r}} \odot \nabla C(\mathrm{w}')$
   - Division and square root are applied element-wise to r
   - η is the global learning rate
5. Compute update: $w \leftarrow w + v$

- RMSProp works very well empirically

Adam

- Name derived from “adaptive moments”
  - Different moments for each parameter
- A combination of RMSProp and momentum
- RMSProp adjust learning rate based on average gradients
- Momentum uses moving average of gradients to take a step
- Adam uses first and 2nd moments
- What is a moment?

$$m_n = E[X^n]$$

Moments of gradients

- Gradients of cost function can be considered a random variable
- Moving averages of gradient and squared gradient

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

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

$$E[m_t] = E[g_t]$$

$$E[v_t] = E[g_t^2]$$

Correcting Bias in moments

- It turns out that estimating moments from sampled data incurs bias
- In statistics there is a way to correct for this bias with bias corrected estimators

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

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

- Adam then applies these to obtain weights

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Adam

- Adam is fairly robust to hyper-parameters
  - Learning rate needs to be changed sometimes from suggested default
- See Goodfellow et al., Section 8.5.3 for suggested initial parameters

Which method should I choose?

- Good question…
- Schaul et al. (2014) presented a large comparison
  - Algorithms with adaptive learning rates generally performed best and were robust
  - But no clear winner emerged
- Popular choices are SGD, SGD with momentum, RMSProp, RMSProp with momentum, and Adam
- In other words, you can choose one and become familiar with it and be relatively confident that there isn't a universally better approach out there
  - Especially once you become familiar with how to tune hyper-parameters with your choice

Further reading

- Goodfellow et al., Sections 4.3, 8.1-8.3, 8.5-8.6
- Nielsen book, chapter 3
- Choromanska et al, “The Loss Surfaces of Multilayer Neural Networks,” 2014.
- https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c