Deep Learning Theory and Applications

Variations on SGD

Yale

Outline

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

Learning vs. pure optimization

Machine learning

- Care about performance measure P wrt test set
  - Sometimes intractable
  - Often optimize P indirectly via a cost function C

Pure optimization

- Minimizing C is the primary goal

Learning vs. pure optimization

Example: classification

- Loss we care about: classification error
  - I.e. the expected 0-1 loss
  - Minimizing this is exponential in the input dimension (typically intractable for even linear classifiers)
- Instead, we optimize a ***surrogate loss function***
  - Acts as a proxy
  - Sometimes results in being able to learn more
    - E.g., test set 0-1 loss can decrease long after the training set 0-1 loss reaches zero as the surrogate loss may push classes further apart
  - Examples: MSE, negative log-likelihood, cross-entropy

Learning vs. pure optimization

Stopping criteria: an important difference

- Pure optimization
  - Stop when we're sufficiently close to a local minimum
  - Typically determined when the gradients are sufficiently small
- Machine learning w/ a surrogate loss
  - Stop based on the true underlying loss (e.g. 0-1 loss) to prevent overfitting
  - May stop while gradients are still large for the surrogate loss

Gradient-based optimization revisited

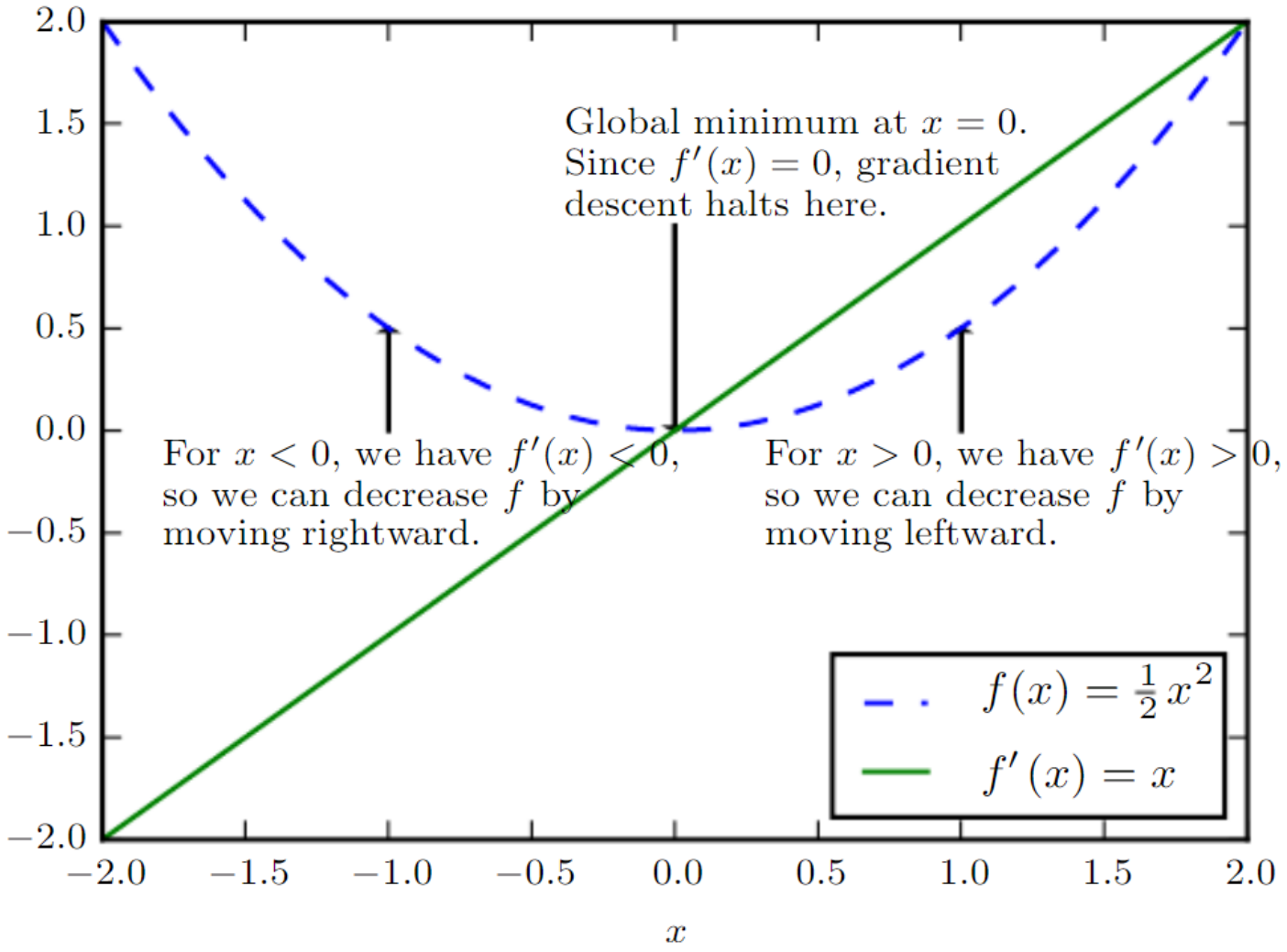
- Deep learning involves optimization of some sort
  - Optimization: the task of either minimizing or maximizing some function $f(x)$ by altering x
  - Typically, we focus on minimizing
    - Maximization can be accomplished by minimizing $-f(x)$
- The function we want to minimize or maximize is the ***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

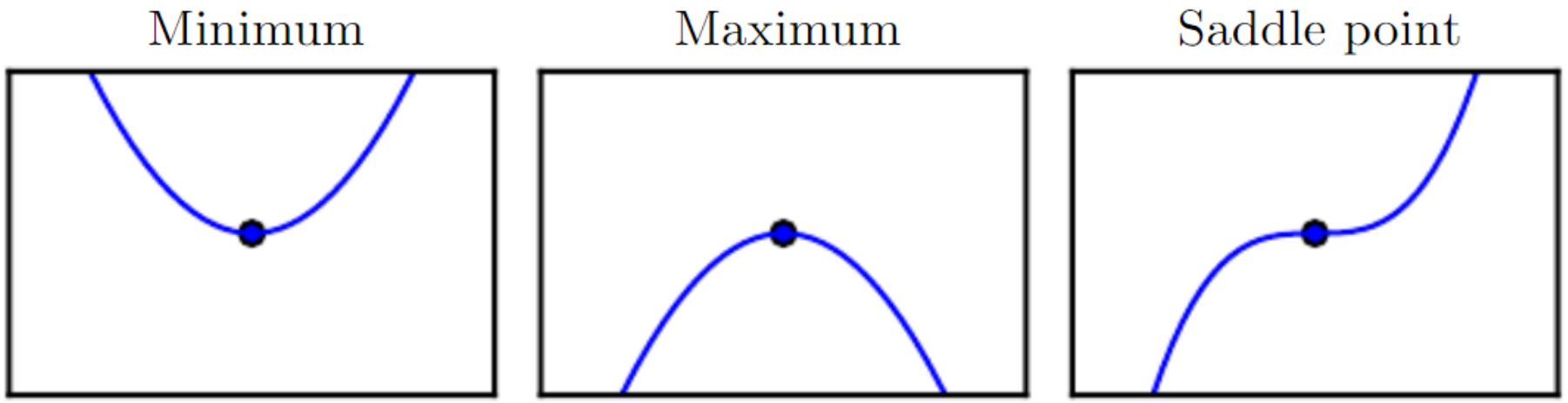
- The derivative $\frac{df}{dx}$ (denoted $f'(x)$) gives the slope of f at the point x
  - I.e., it specifies how to scale a small change in input to obtain corresponding change in the output: $f(x+\epsilon) \approx f(x) + \epsilon f'(x)$
  - Thus the derivative tells us how to change x to make small 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



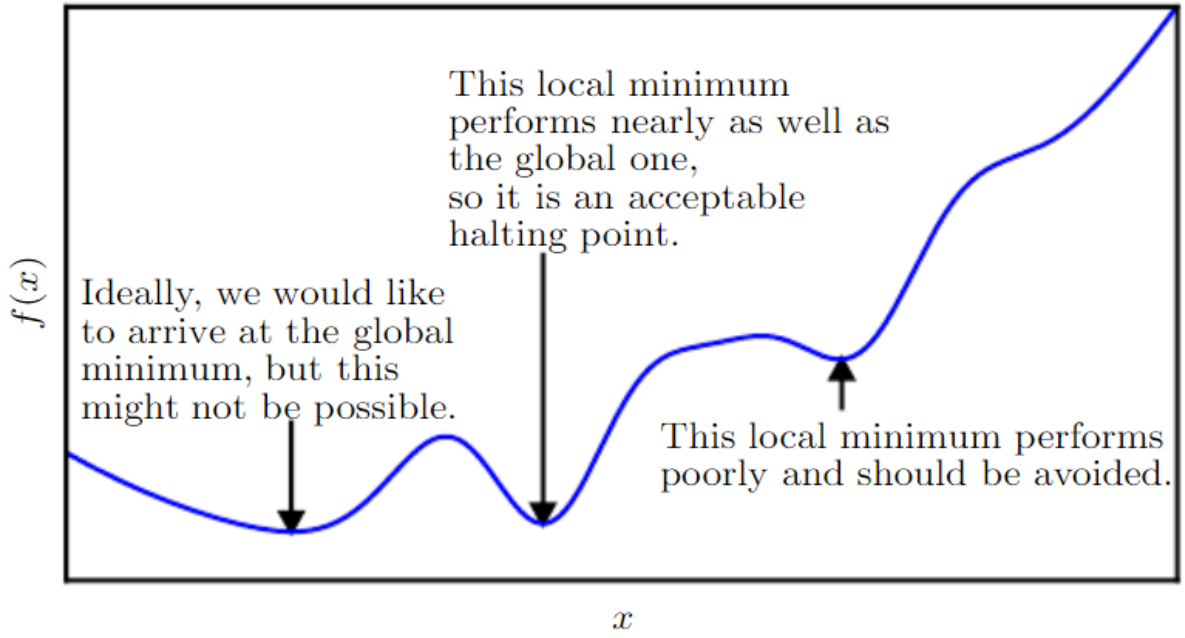
Gradient-based optimization revisited

- 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

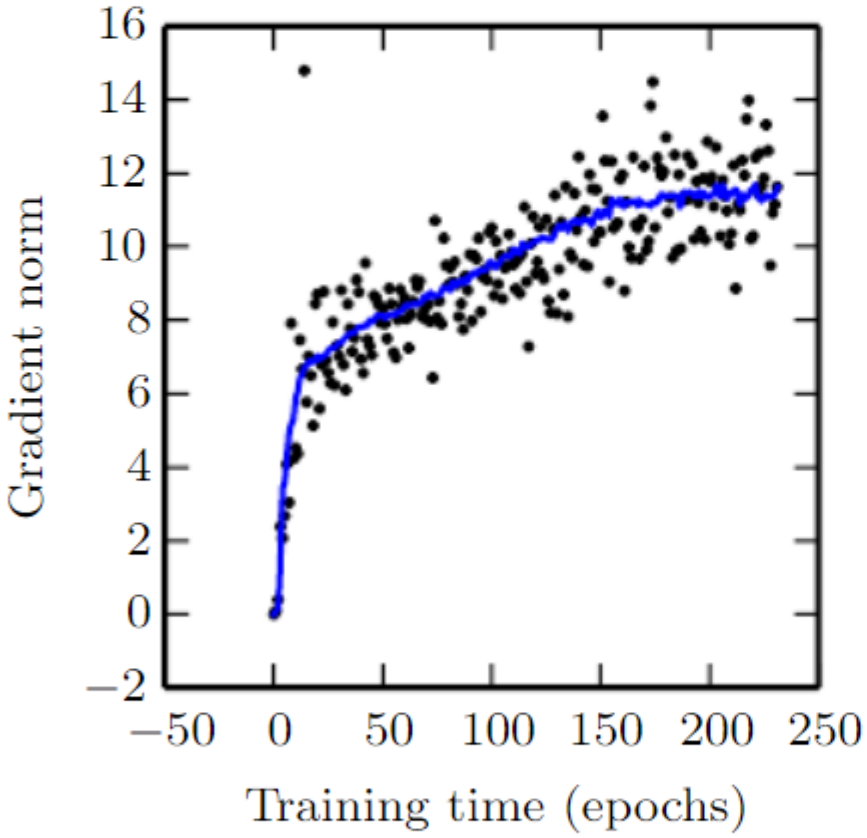


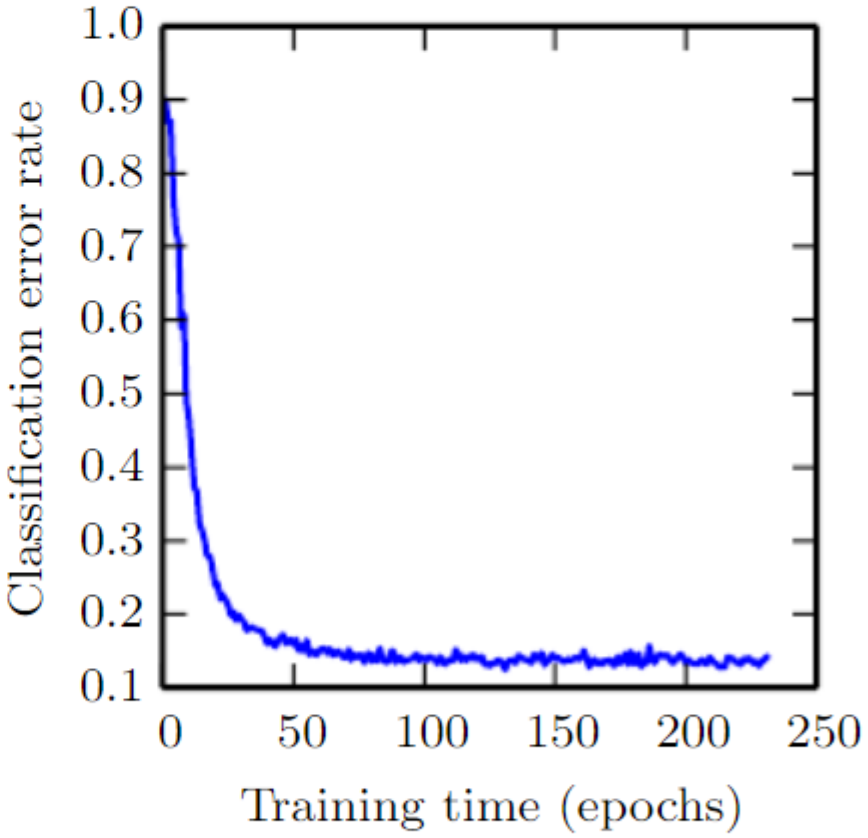
Are local minima a problem?

- 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)
  - Can result in an uncountably infinite number of local minima
  - However, they all have the same cost (therefore not problematic)
- 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
  - Thus local minima typically aren't a problem

Are local minima a problem?

- Can test for local minima by checking gradient sizes

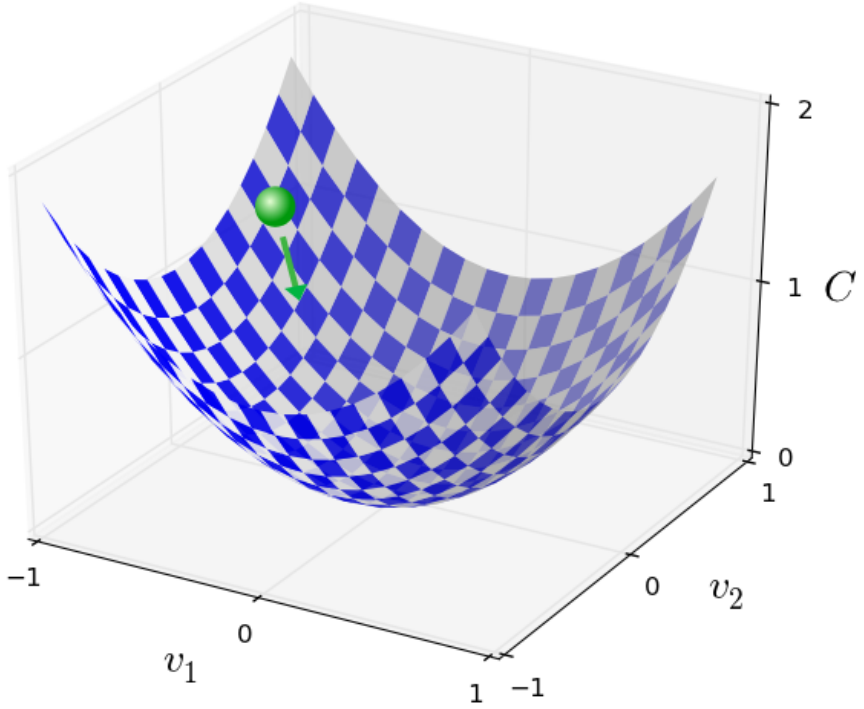




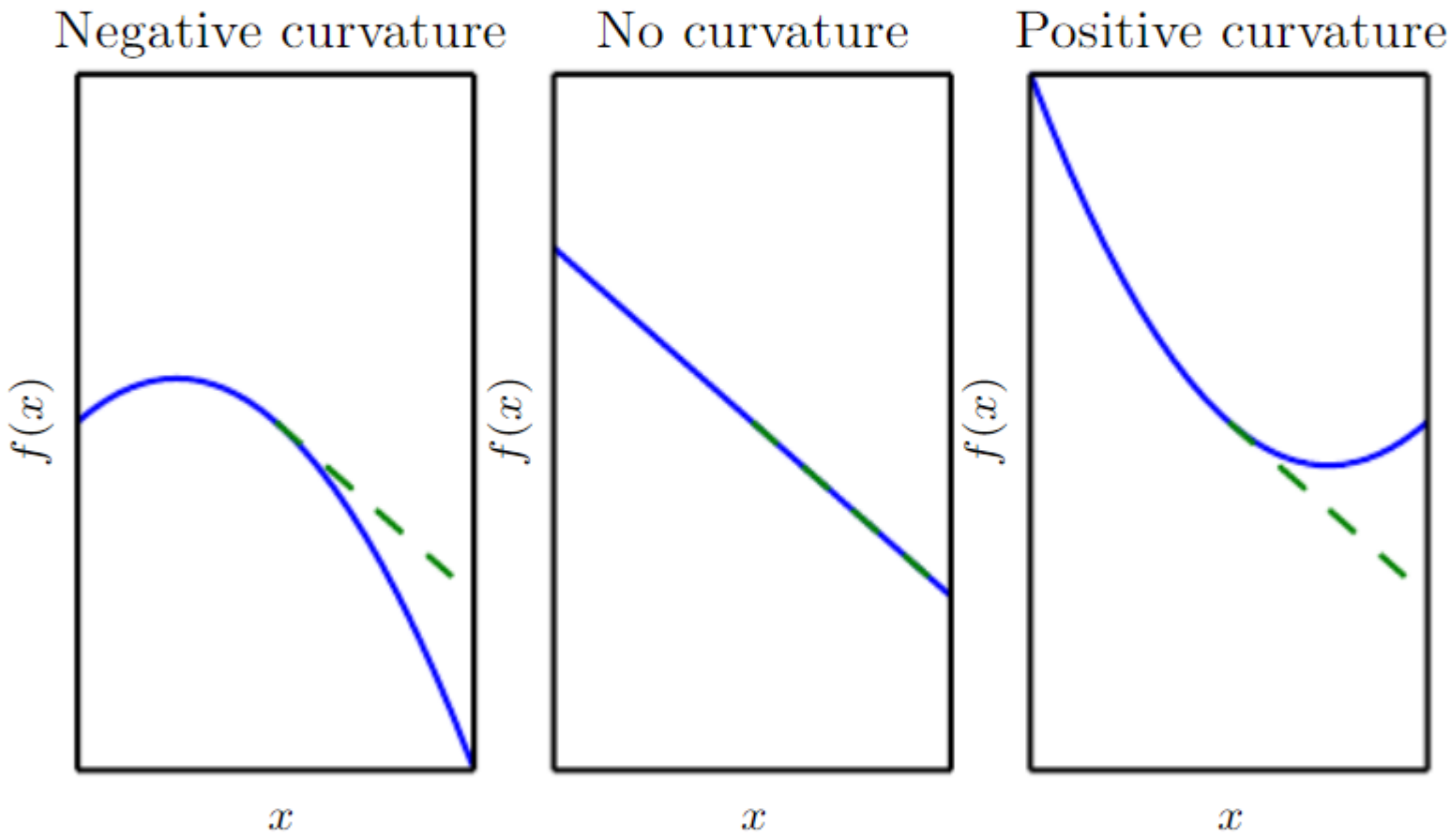
Gradient-based optimization revisited

- We often minimize functions with multiple inputs: $f: \mathbb{R}^d \to \mathbb{R}$
  - Does it make sense to have multidimensional output?
- The gradient generalizes the notion of derivative to a vector
- Neural network update step:

$$w' \leftarrow w - \eta \nabla_w C$$



Curvature



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

2^{nd} Order Methods

The Hessian

- 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

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

  - H is the ***Hessian*** matrix with jkth entry as $\frac{\partial^2 C}{\partial w_j \partial w_k}$

The Hessian

- The second derivatives (i.e. the Hessian) give a measure of the curvature of the cost function
- Incorporating the second derivatives should give better results
- 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

Second-order optimization

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

- Referred to as Newton's Method
- Due to the approximation, better to take small steps:

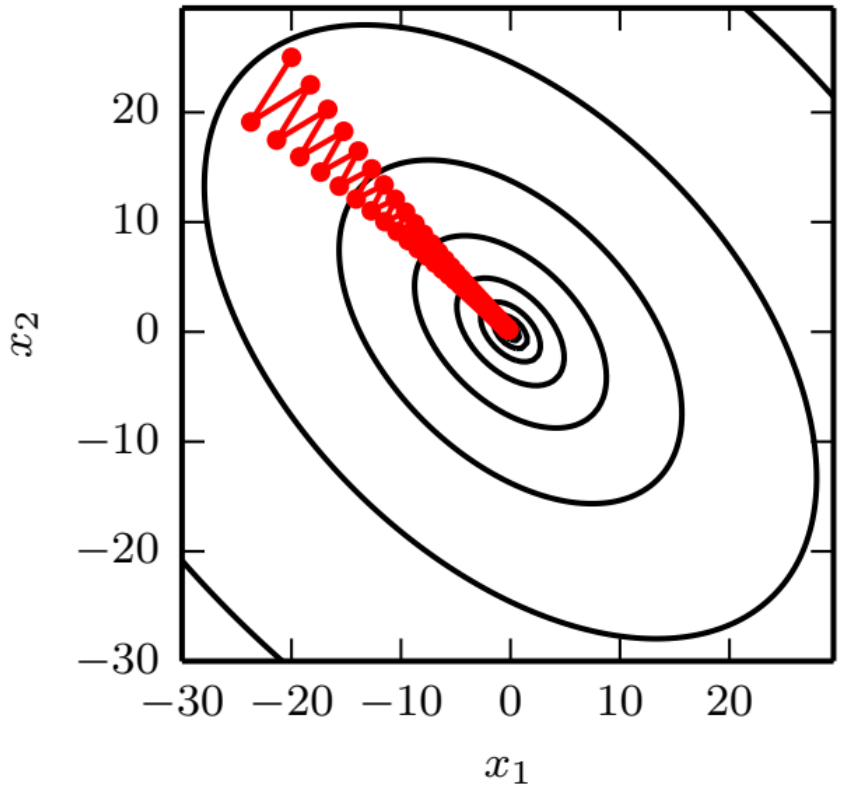
$$w' \leftarrow w - \eta H^{-1}\nabla C$$

- This approach is referred to as 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

- This is due to ill-conditioning of the Hessian
- A gradient descent step of $-\eta\nabla C$ adds to the cost: $\frac{1}{2}\eta^2(\nabla C)^T H\nabla C - \eta(\nabla C)^T\nabla C$
- Problematic if $\frac{1}{2}\eta^2(\nabla C)^T H\nabla C > \eta(\nabla C)^T\nabla C$



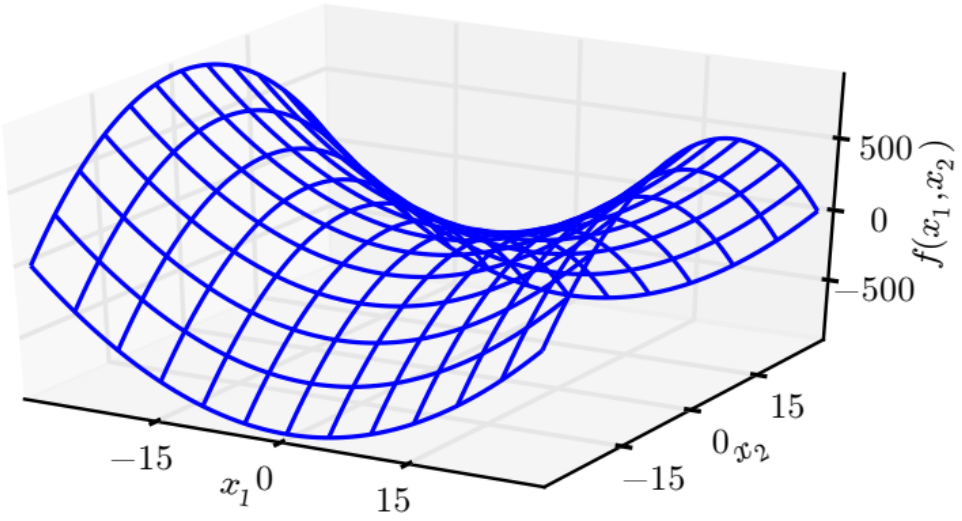
Advantages of Hessian methods

- Hessian methods correct for ill-conditioning and avoid many pathologies in GD
- Backpropagation can be modified to compute the Hessian

- So should we use Hessian methods to train neural networks?
  - There are some issues with using Hessian and other 2^{nd}-order methods with neural networks

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
- 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?

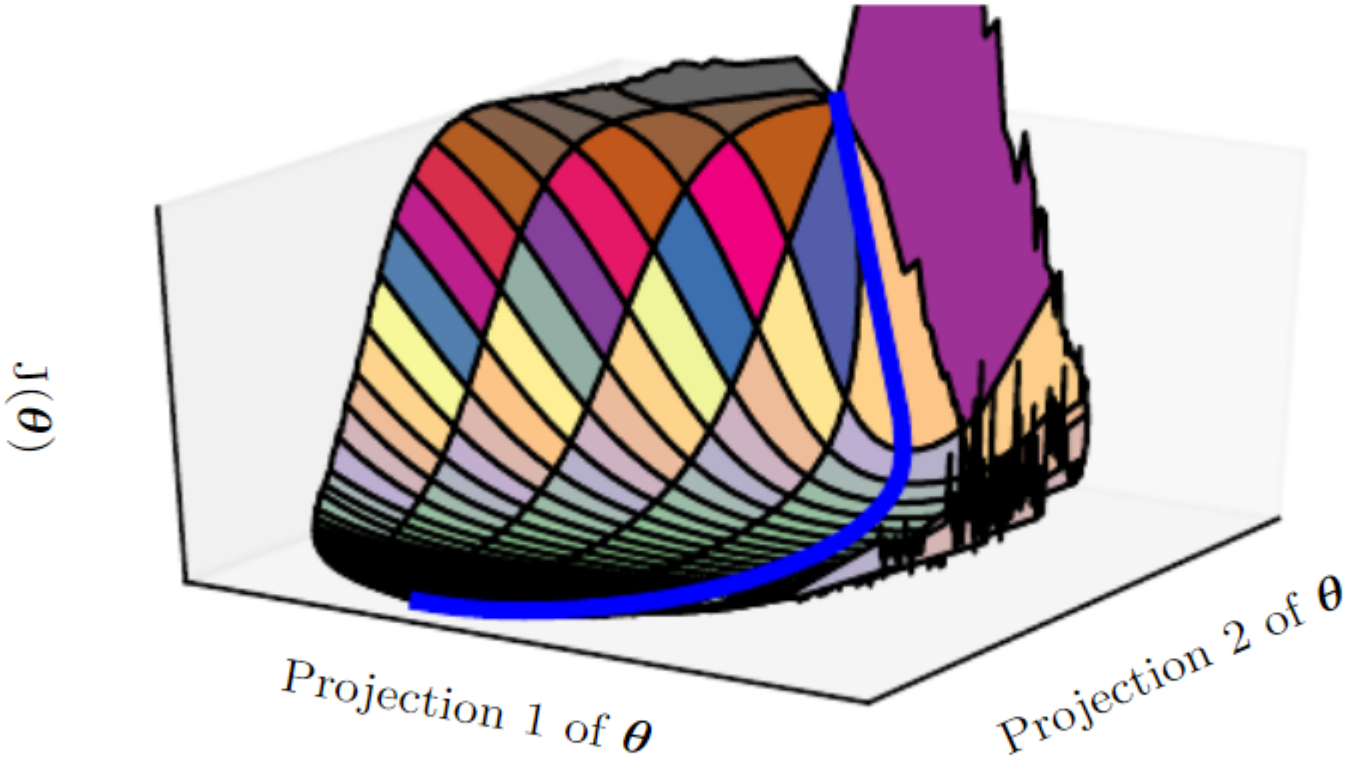
- In high-dimensional nonconvex functions, local minima are rare compared to saddle points
  - For a random function $f: \mathbb{R}^d \to \mathbb{R}$, the expected ratio of the number of saddle points to local minima grows exponentially with d
- Intuition: consider the Hessian matrix
  - At a local minima, 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



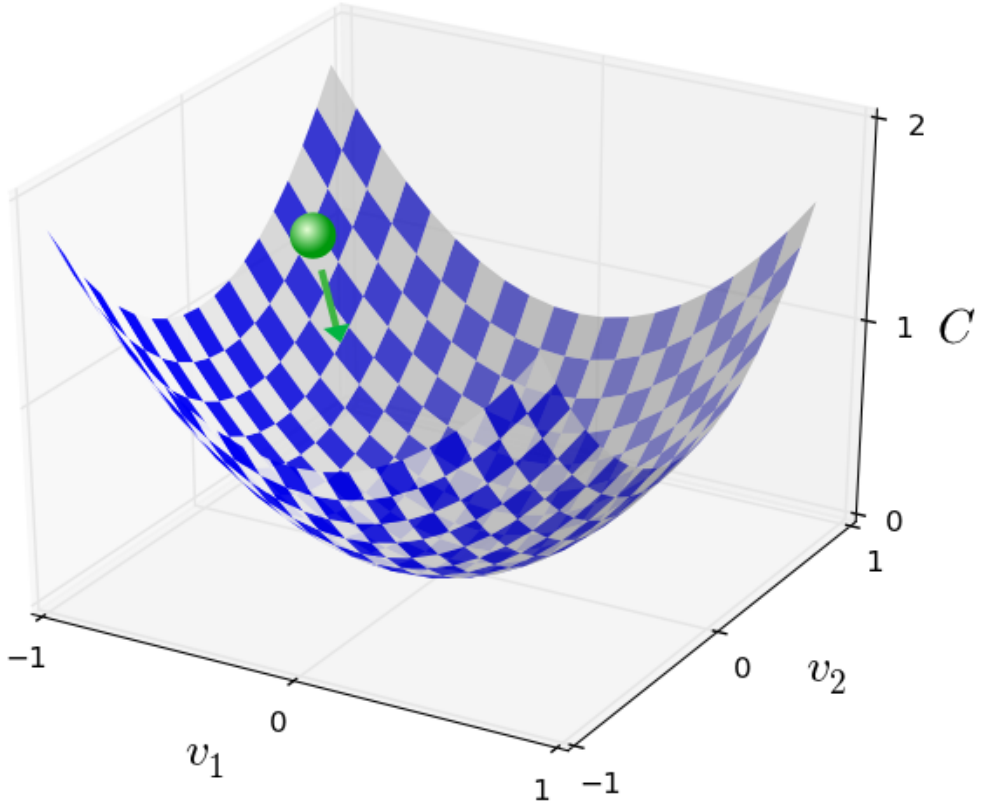
Are saddle points a problem?

- 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

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:

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

  - 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

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

- 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
- 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

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 $w \leftarrow w + v$

- Equivalent to $w \leftarrow w + \mu^2 v - (1 + \mu)\eta \nabla_w C$
- This gives more weight to the gradient
- For convex batch gradient methods, Nesterov momentum improves the convergence of excess error from $O\left(\frac{1}{k}\right)$ to $O\left(\frac{1}{k^2}\right)$, where k is the number of steps
  - Unfortunately, Nesterov momentum does not improve SGD convergence

Adaptive Learning Rates

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”
- A combination of RMSProp and momentum
  - Corrects for bias in moments due to initialization

1. Compute gradient ∇C from mini-batch
2. $t \leftarrow t + 1$
   - $t = 0$ initially
3. Update biased first moment: $s \leftarrow \rho_1 s + (1 - \rho_1)\nabla \mathrm{C}$
4. Update biased second moment:
   $$r \leftarrow \rho_2 r + (1 - \rho_2)\nabla C \odot \nabla C$$
5. Correct bias in first moment: $s \leftarrow \frac{s}{1-\rho_1^t}$
6. Correct bias in second moment: $r \leftarrow \frac{r}{1-\rho_2^t}$
7. Compute update: $w \leftarrow w - \eta \frac{s}{\sqrt{r}+\delta}$ (element-wise)

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

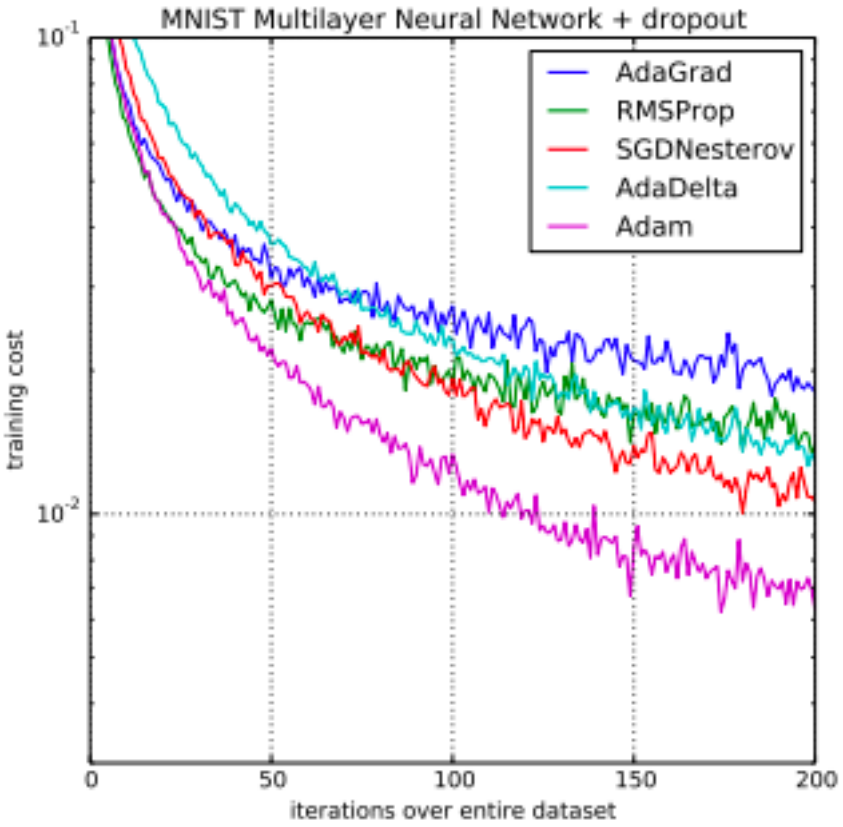
Adam

- From the original paper:
  - Straightforward to implement.
  - Computationally efficient.
  - Little memory requirements.
  - Invariant to diagonal rescale of the gradients.
  - Well suited for problems that are large in terms of data and/or parameters.
  - Appropriate for non-stationary objectives.
  - Appropriate for problems with very noisy/or sparse gradients.
  - Hyper-parameters have intuitive interpretation and typically require little tuning.

Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980*(2014).

Adam

- Adam often does much better than alternatives
- Though relative performance between models is very context-specific
  - Dataset, model dependent



https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

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