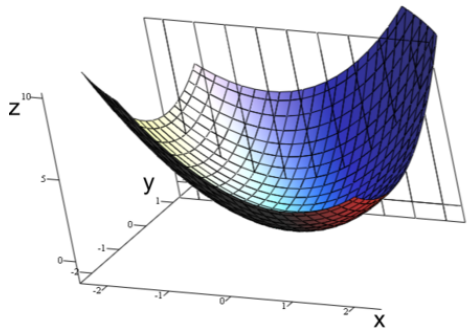
COMS 4776 Lecture 6: Optimization

Richard Zemel

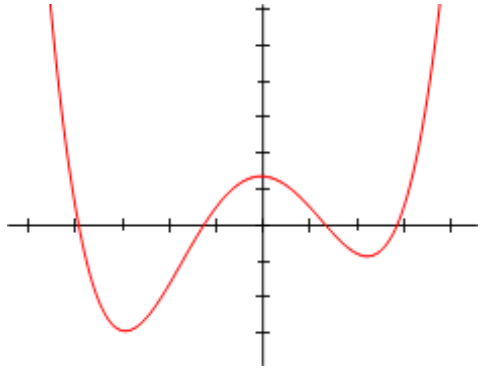
Overview

- We've talked a lot about how to compute gradients. What do we actually do with them?
- Today's lecture: various things that can go wrong in gradient descent, and what to do about them.
- Let's group all the parameters (weights and biases) of our network into a single vector $\boldsymbol{\theta}$.
- This lecture makes heavy use of the spectral decomposition of symmetric matrices, so it would be a good idea to review this.
  - Subsequent lectures will not build on the more mathematical parts of this lecture, so you can take your time to understand it.

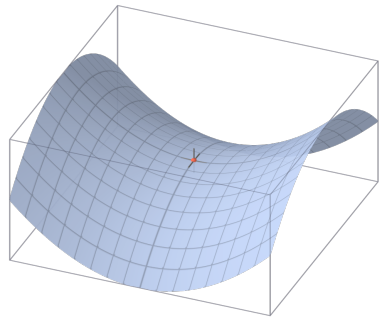
Features of the Optimization Landscape



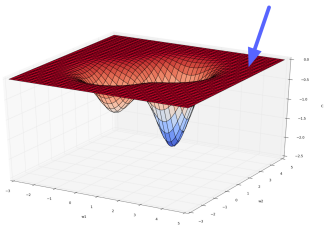
convex functions



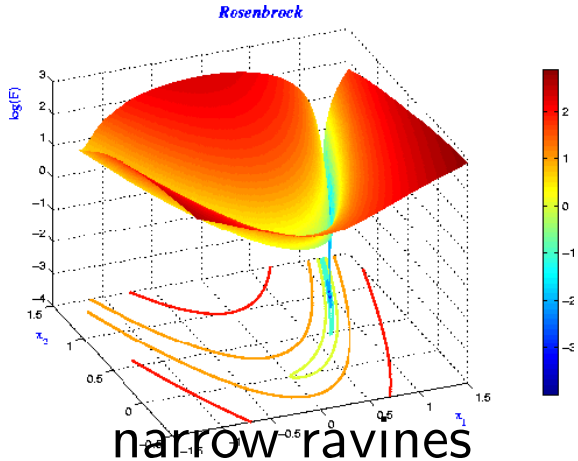
local minima



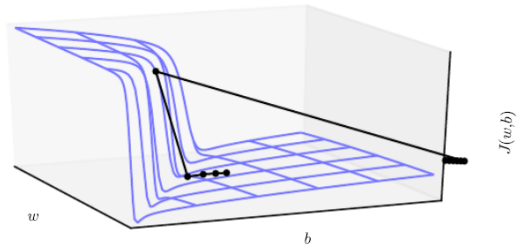
saddle points



plateaux



narrow ravines



cliffs (covered in a later lecture)

Review: Hessian Matrix

- The Hessian matrix, denoted $\mathbf{H}$, or $\nabla^2 \mathcal{J}$ is the matrix of second derivatives:

$$\mathbf{H} = \nabla^2 \mathcal{J} = \begin{pmatrix} \frac{\partial^2 \mathcal{J}}{\partial \theta_1^2} & \frac{\partial^2 \mathcal{J}}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 \mathcal{J}}{\partial \theta_1 \partial \theta_D} \\ \frac{\partial^2 \mathcal{J}}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 \mathcal{J}}{\partial \theta_2^2} & \cdots & \frac{\partial^2 \mathcal{J}}{\partial \theta_2 \partial \theta_D} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 \mathcal{J}}{\partial \theta_D \partial \theta_1} & \frac{\partial^2 \mathcal{J}}{\partial \theta_D \partial \theta_2} & \cdots & \frac{\partial^2 \mathcal{J}}{\partial \theta_D^2} \end{pmatrix}$$

- It's a symmetric matrix because $\frac{\partial^2 \mathcal{J}}{\partial \theta_i \partial \theta_j} = \frac{\partial^2 \mathcal{J}}{\partial \theta_j \partial \theta_i}$.

Review: Hessian Matrix

- Locally, a function can be approximated by its second-order Taylor approximation around a point $\boldsymbol{\theta}_0$:

$$\mathcal{J}(\boldsymbol{\theta}) \approx \mathcal{J}(\boldsymbol{\theta}_0) + \nabla \mathcal{J}(\boldsymbol{\theta}_0)^\top (\boldsymbol{\theta} - \boldsymbol{\theta}_0) + \tfrac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H}(\boldsymbol{\theta}_0)(\boldsymbol{\theta} - \boldsymbol{\theta}_0).$$

- A critical point is a point where the gradient is zero. In that case,

$$\mathcal{J}(\boldsymbol{\theta}) \approx \mathcal{J}(\boldsymbol{\theta}_0) + \tfrac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H}(\boldsymbol{\theta}_0)(\boldsymbol{\theta} - \boldsymbol{\theta}_0).$$

Review: Hessian Matrix

- A lot of important features of the optimization landscape can be characterized by the eigenvalues of the Hessian $\mathbf{H}$.
- Recall that a symmetric matrix (such as $\mathbf{H}$) has only real eigenvalues, and there is an orthogonal basis of eigenvectors.
- This can be expressed in terms of the spectral decomposition:

$$\mathbf{H} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top,$$

where $\mathbf{Q}$ is an orthogonal matrix (whose columns are the eigenvectors) and $\mathbf{\Lambda}$ is a diagonal matrix (whose diagonal entries are the eigenvalues).

Review: Hessian Matrix

- We often refer to $\mathbf{H}$ as the curvature of a function.
- Suppose you move along a line defined by $\boldsymbol{\theta} + t\mathbf{v}$ for some vector $\mathbf{v}$.
- Second-order Taylor approximation:

$$\mathcal{J}(\boldsymbol{\theta} + t\mathbf{v}) \approx \mathcal{J}(\boldsymbol{\theta}) + t\nabla\mathcal{J}(\boldsymbol{\theta})^\top \mathbf{v} + \frac{t^2}{2}\mathbf{v}^\top \mathbf{H}(\boldsymbol{\theta})\mathbf{v}$$

- Hence, in a direction where $\mathbf{v}^\top \mathbf{H}\mathbf{v} > 0$, the cost function curves upwards, i.e. has positive curvature. Where $\mathbf{v}^\top \mathbf{H}\mathbf{v} < 0$, it has negative curvature.

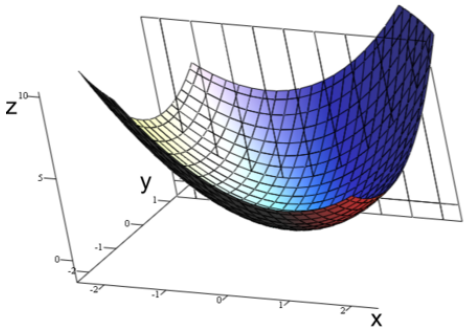
Review: Hessian Matrix

- A matrix $\mathbf{A}$ is positive definite if $\mathbf{v}^\top \mathbf{A}\mathbf{v} > 0$ for all $\mathbf{v} \neq 0$. (I.e., it curves upwards in all directions.)
  - It is positive semidefinite (PSD) if $\mathbf{v}^\top \mathbf{A}\mathbf{v} \geq 0$ for all $\mathbf{v} \neq 0$.
- Equivalently: a matrix is positive definite iff all its eigenvalues are positive. It is PSD iff all its eigenvalues are nonnegative. (Exercise: show this using the Spectral Decomposition.)
- For any critical point $\boldsymbol{\theta}_*$, if $\mathbf{H}(\boldsymbol{\theta}_*)$ exists and is positive definite, then $\boldsymbol{\theta}_*$ is a local minimum (since all directions curve upwards).

Convex Functions

- Recall: a set $\mathcal{S}$ is convex if for any $\mathbf{x}_0, \mathbf{x}_1 \in \mathcal{S}$,

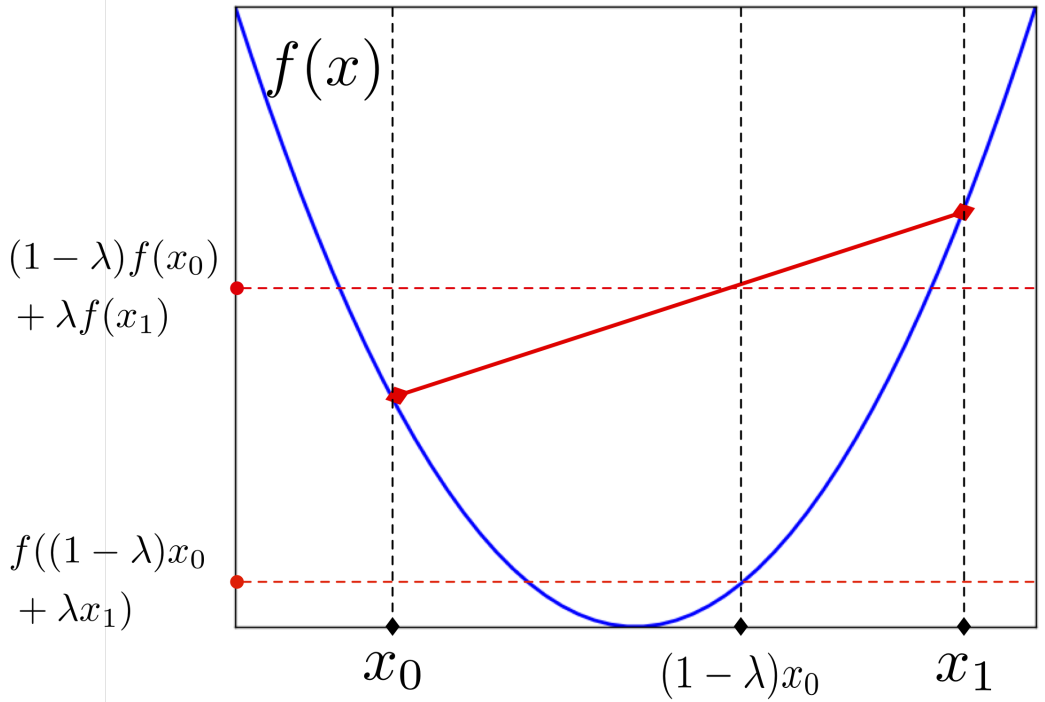
$$(1-\lambda)\mathbf{x}_0 + \lambda\mathbf{x}_1 \in \mathcal{S} \quad \text{for } 0 \leq \lambda \leq 1.$$



- A function f is convex if for any $\mathbf{x}_0, \mathbf{x}_1$,

$$f((1-\lambda)\mathbf{x}_0 + \lambda\mathbf{x}_1) \leq (1-\lambda)f(\mathbf{x}_0) + \lambda f(\mathbf{x}_1)$$

- Equivalently, the set of points lying above the graph of f is convex.
- Intuitively: the function is bowl-shaped.

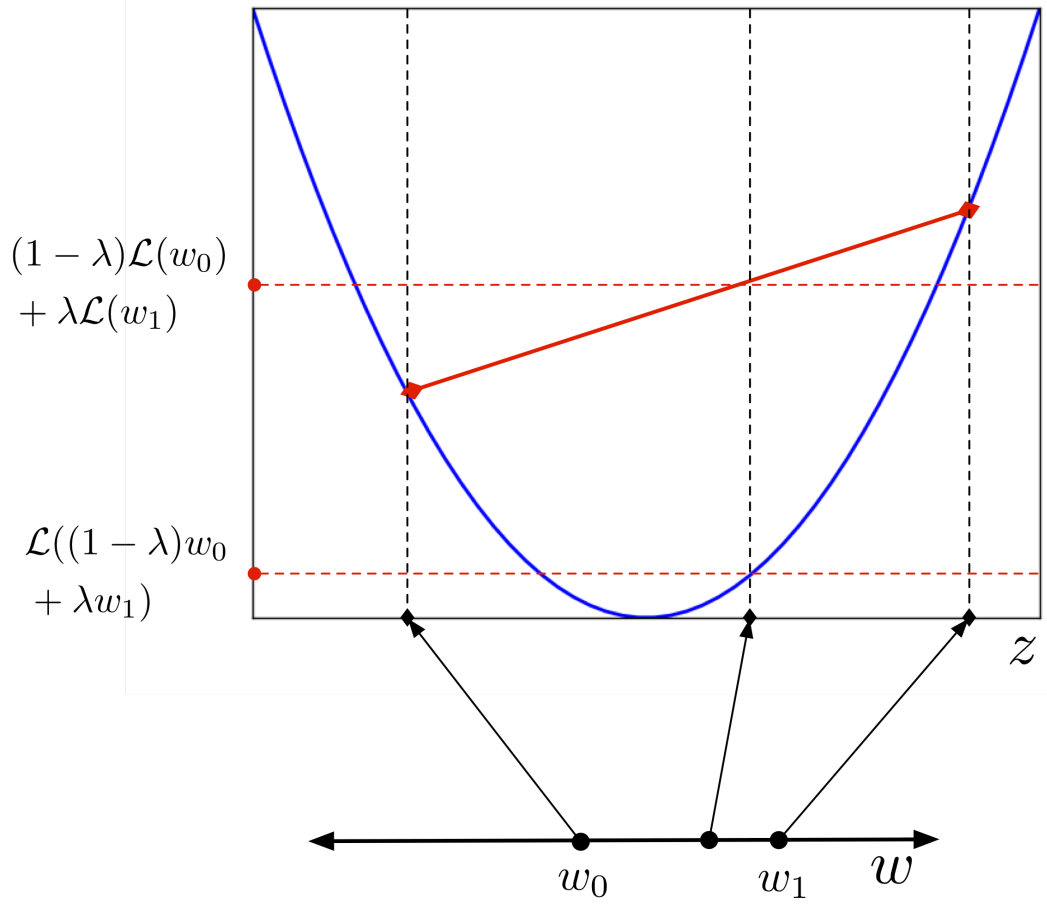


Convex Functions

- If $\mathcal{J}$ is smooth (more precisely, twice differentiable), there's an equivalent characterization in terms of $\mathbf{H}$:
  - A smooth function is convex iff its Hessian is positive semidefinite everywhere.
  - **Special case:** a univariate function is convex iff its second derivative is nonnegative everywhere.
- **Exercise:** show that squared error, logistic-cross-entropy, and softmax-cross-entropy losses are convex (as a function of the network outputs) by taking second derivatives.

Convex Functions

- For a linear model, $z = \mathbf{w}^\top \mathbf{x} + b$ is a linear function of $\mathbf{w}$ and b. If the loss function is convex as a function of z, then it is convex as a function of $\mathbf{w}$ and b.
- Hence, linear regression, logistic regression, and softmax regression are convex.

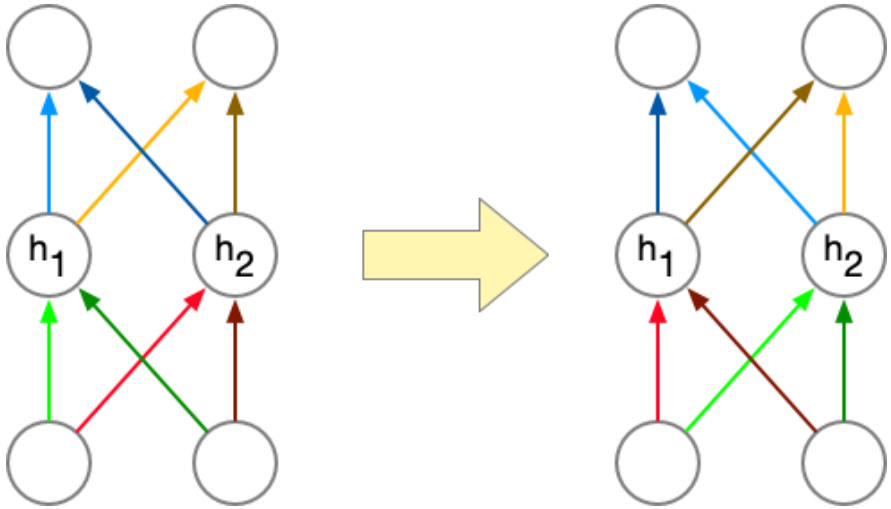


Local Minima

- If a function is convex, it has no spurious local minima, i.e. any local minimum is also a global minimum.
- This is very convenient for optimization since if we keep going downhill, we'll eventually reach a global minimum.

Local Minima

- If a function is convex, it has no spurious local minima, i.e. any local minimum is also a global minimum.
- This is very convenient for optimization since if we keep going downhill, we'll eventually reach a global minimum.
- Unfortunately, training a network with hidden units cannot be convex because of permutation symmetries.
  - I.e., we can re-order the hidden units in a way that preserves the function computed by the network.



Local Minima

- By definition, if a function $\mathcal{J}$ is convex, then for any set of points $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_N$ in its domain,

$$\mathcal{J}(\lambda_1 \boldsymbol{\theta}_1 + \cdots + \lambda_N \boldsymbol{\theta}_N) \leq \lambda_1 \mathcal{J}(\boldsymbol{\theta}_1) + \cdots + \lambda_N \mathcal{J}(\boldsymbol{\theta}_N) \quad \text{for } \lambda_i \geq 0, \sum_i \lambda_i = 1.$$

- Because of permutation symmetry, there are $K!$ permutations of the hidden units in a given layer which all compute the same function.
- Suppose we average the parameters for all $K!$ permutations. Then we get a degenerate network where all the hidden units are identical.
- If the cost function were convex, this solution would have to be better than the original one, which is ridiculous!
- Hence, training multilayer neural nets is non-convex.

Local Minima (optional, informal)

- Generally, local minima aren't something we worry much about when we train most neural nets.
  - They're normally only a problem if there are local minima "in function space". E.g., CycleGANs (covered later in this course) have a bad local minimum where they learn the wrong color mapping between domains.

Local Minima (optional, informal)

- Generally, local minima aren't something we worry much about when we train most neural nets.
  - They're normally only a problem if there are local minima "in function space". E.g., CycleGANs (covered later in this course) have a bad local minimum where they learn the wrong color mapping between domains.
- It's possible to construct arbitrarily bad local minima even for ordinary classification MLPs. It's poorly understood why these don't arise in practice.

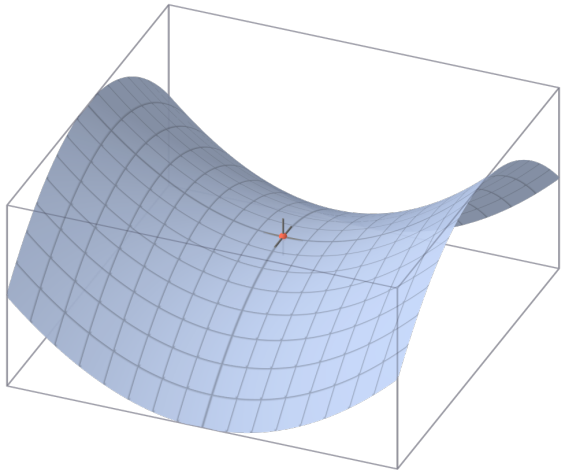
Local Minima (optional, informal)

- Generally, local minima aren't something we worry much about when we train most neural nets.
  - They're normally only a problem if there are local minima "in function space". E.g., CycleGANs (covered later in this course) have a bad local minimum where they learn the wrong color mapping between domains.
- It's possible to construct arbitrarily bad local minima even for ordinary classification MLPs. It's poorly understood why these don't arise in practice.
- Intuition pump: if you have enough randomly sampled hidden units, you can approximate any function just by adjusting the output layer.
  - Then it's essentially a regression problem, which is convex.
  - Hence, local optima can probably be fixed by adding more hidden units.
  - Note: this argument hasn't been made rigorous.

Local Minima (optional, informal)

- Generally, local minima aren't something we worry much about when we train most neural nets.
  - They're normally only a problem if there are local minima "in function space". E.g., CycleGANs (covered later in this course) have a bad local minimum where they learn the wrong color mapping between domains.
- It's possible to construct arbitrarily bad local minima even for ordinary classification MLPs. It's poorly understood why these don't arise in practice.
- Intuition pump: if you have enough randomly sampled hidden units, you can approximate any function just by adjusting the output layer.
  - Then it's essentially a regression problem, which is convex.
  - Hence, local optima can probably be fixed by adding more hidden units.
  - Note: this argument hasn't been made rigorous.
- Over the past 5 years or so, CS theorists have made lots of progress proving gradient descent converges to global minima for some non-convex problems, including some specific neural net architectures.

Saddle points

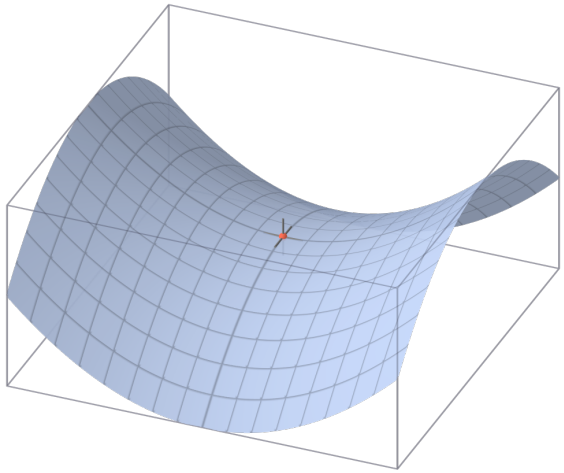


A saddle point is a point where:

- $\nabla \mathcal{J}(\boldsymbol{\theta}) = \mathbf{0}$
- $\mathbf{H}(\boldsymbol{\theta})$ has some positive and some negative eigenvalues, i.e. some directions with positive curvature and some with negative curvature.

When would saddle points be a problem?

Saddle points



A saddle point is a point where:

- $\nabla \mathcal{J}(\boldsymbol{\theta}) = \mathbf{0}$
- $\mathbf{H}(\boldsymbol{\theta})$ has some positive and some negative eigenvalues, i.e. some directions with positive curvature and some with negative curvature.

When would saddle points be a problem?

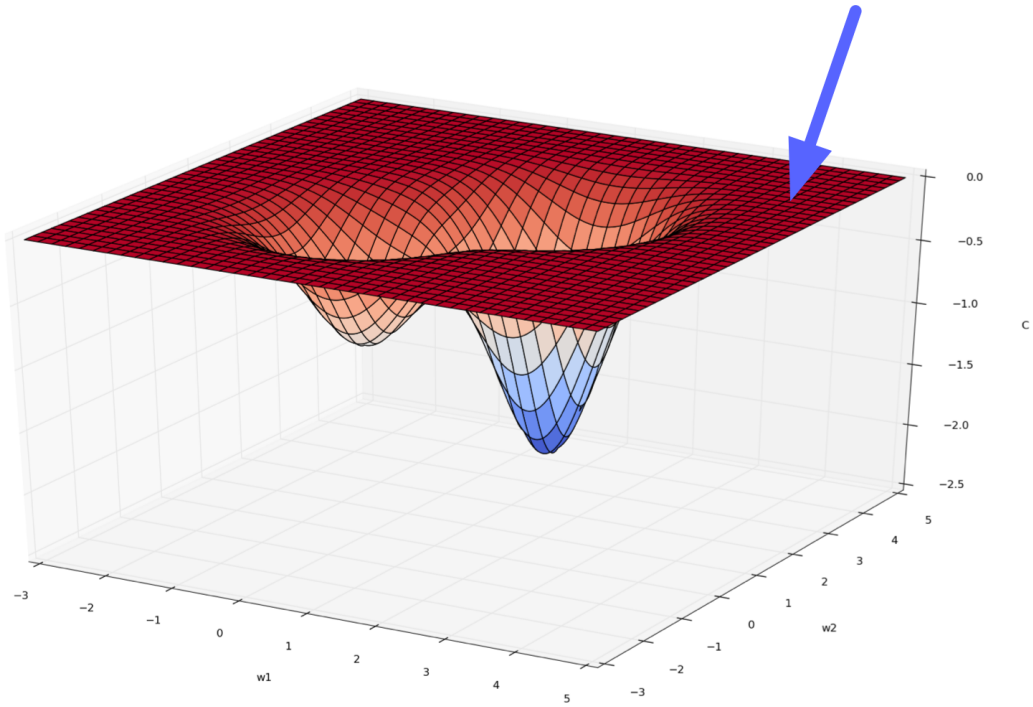
- If we're exactly on the saddle point, then we're stuck.
- If we're slightly to the side, then we can get unstuck.

Saddle points

- Suppose you have two hidden units with identical incoming and outgoing weights.
- After a gradient descent update, they will still have identical weights. By induction, they'll always remain identical.
- But if you perturbed them slightly, they can start to move apart.
- Important special case: don't initialize all your weights to zero!
  - Instead, break the symmetry by using small random values.

Plateaux

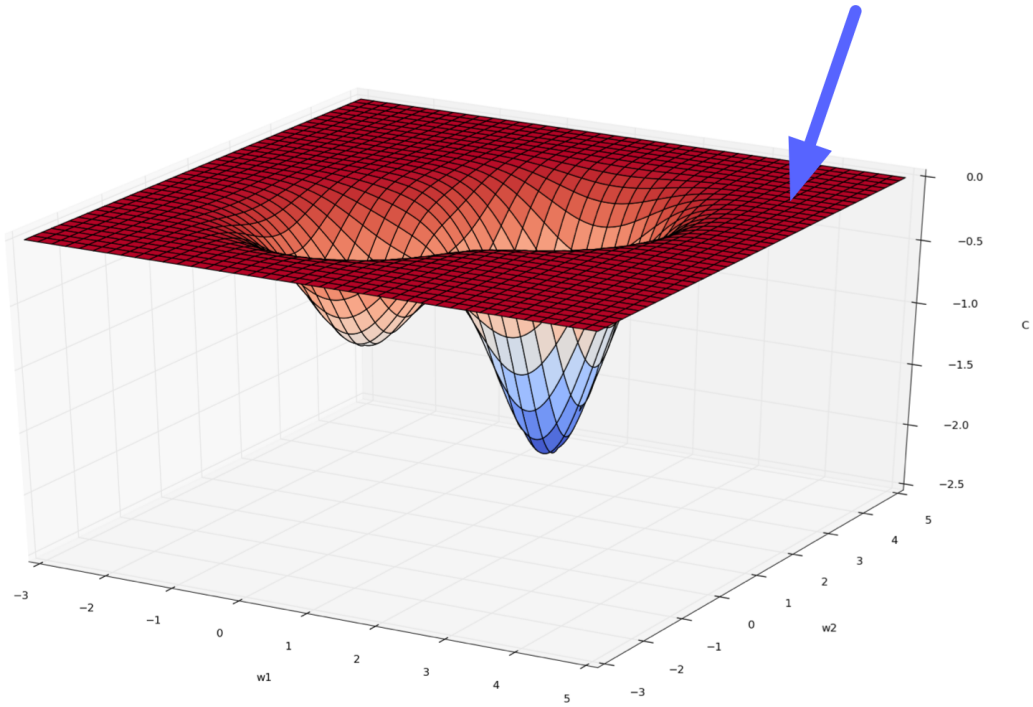
A flat region is called a plateau. (Plural: plateaux)



Can you think of examples?

Plateaux

A flat region is called a plateau. (Plural: plateaux)



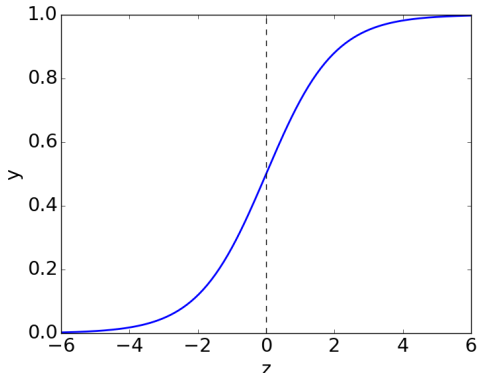
Can you think of examples?

- 0–1 loss
- hard threshold activations
- logistic activations & least squares

Plateaux

- An important example of a plateau is a saturated unit. This is when it is in the flat region of its activation function. Recall the backprop equation for the weight derivative:

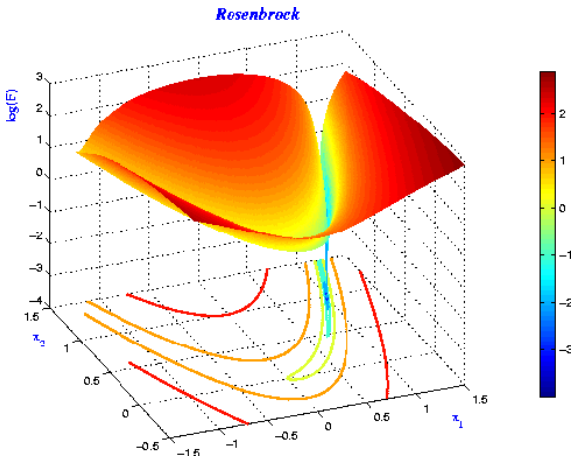
$$\overline{z_i} = \overline{h_i}\,\phi'(z)$$
$$\overline{w_{ij}} = \overline{z_i}\,x_j$$

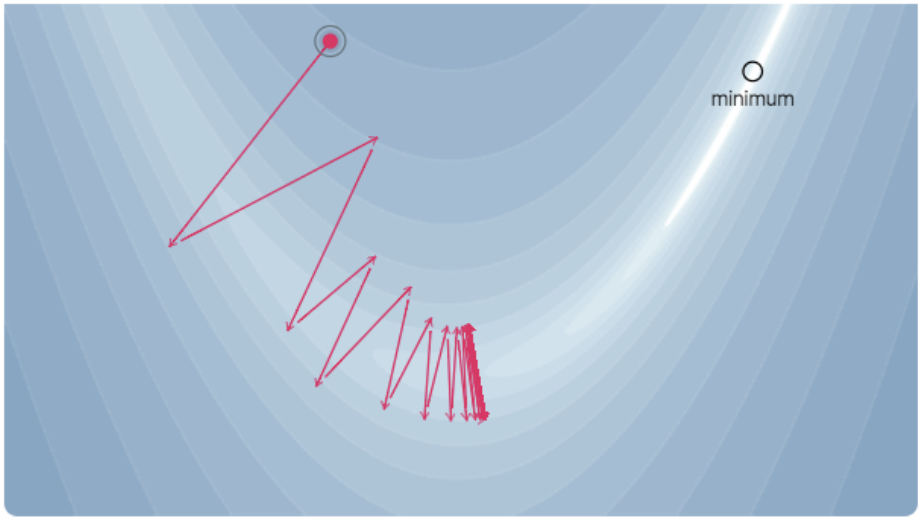


- If $\phi'(z_i)$ is always close to zero, then the weights will get stuck.
- If there is a ReLU unit whose input z_i is always negative, the weight derivatives will be *exactly* 0. We call this a dead unit.

Ill-conditioned curvature

Long, narrow ravines:





- Suppose **H** has some large positive eigenvalues (i.e. high-curvature directions) and some eigenvalues close to 0 (i.e. low-curvature directions).
- Gradient descent bounces back and forth in high curvature directions and makes slow progress in low curvature directions.
  - To interpret this visually: the gradient is perpendicular to the contours.
- This is known as ill-conditioned curvature. It's very common in neural net training.

Ill-conditioned curvature: gradient descent dynamics

- To understand why ill-conditioned curvature is a problem, consider a convex quadratic objective

$$\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{\theta}^\top \mathbf{A}\boldsymbol{\theta},$$

  where $\mathbf{A}$ is PSD.

- Gradient descent update:

$$\begin{aligned}\boldsymbol{\theta}_{k+1} &\leftarrow \boldsymbol{\theta}_k - \alpha \nabla \mathcal{J}(\boldsymbol{\theta}_k) \\ &= \boldsymbol{\theta}_k - \alpha \mathbf{A}\boldsymbol{\theta}_k \\ &= (\mathbf{I} - \alpha \mathbf{A})\boldsymbol{\theta}_k\end{aligned}$$

- Solving the recurrence,

$$\boldsymbol{\theta}_k = (\mathbf{I} - \alpha \mathbf{A})^k \boldsymbol{\theta}_0$$

Ill-conditioned curvature: gradient descent dynamics

- We can analyze matrix powers such as $(\mathbf{I} - \alpha\mathbf{A})^k\boldsymbol{\theta}_0$ using the spectral decomposition.
- Let $\mathbf{A} = \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^\top$ be the spectral decomposition of $\mathbf{A}$.

$$\begin{aligned}(\mathbf{I} - \alpha\mathbf{A})^k\boldsymbol{\theta}_0 &= (\mathbf{I} - \alpha\mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^\top)^k\boldsymbol{\theta}_0 \\ &= [\mathbf{Q}(\mathbf{I} - \alpha\boldsymbol{\Lambda})\mathbf{Q}^\top]^k\boldsymbol{\theta}_0 \\ &= \mathbf{Q}(\mathbf{I} - \alpha\boldsymbol{\Lambda})^k\mathbf{Q}^\top\boldsymbol{\theta}_0\end{aligned}$$

- Hence, in the $\mathbf{Q}$ basis, each coordinate gets multiplied by $(1 - \alpha\lambda_i)^k$, where the λ_i are the eigenvalues of $\mathbf{A}$.
- Cases:
  - $0 < \alpha\lambda_i \leq 1$: decays to 0 at a rate that depends on $\alpha\lambda_i$
  - $1 < \alpha\lambda_i \leq 2$: oscillates
  - $\alpha\lambda_i > 2$: unstable (diverges)

Ill-conditioned curvature: gradient descent dynamics

- Just showed
  - $0 < \alpha\lambda_i \leq 1$: decays to 0 at a rate that depends on $\alpha\lambda_i$
  - $1 < \alpha\lambda_i \leq 2$: oscillates
  - $\alpha\lambda_i > 2$: unstable (diverges)
- Hence, we need to set the learning rate $\alpha < 2/\lambda_{\max}$ to prevent instability, where $\lambda_{\max}$ is the largest eigenvalue, i.e. maximum curvature.
- This bounds the rate of progress in another direction:

$$\alpha\lambda_i < \frac{2\lambda_i}{\lambda_{\max}}.$$

- The quantity $\lambda_{\max}/\lambda_{\min}$ is known as the condition number of $\mathbf{A}$. Larger condition numbers imply slower convergence of gradient descent.

Ill-conditioned curvature: gradient descent dynamics

- The analysis we just did was for a quadratic toy problem

$$\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{\theta}^\top \mathbf{A}\boldsymbol{\theta}.$$

- It can be easily generalized to a quadratic not centered at zero, since the gradient descent dynamics are invariant to translation.

$$\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{\theta}^\top \mathbf{A}\boldsymbol{\theta} + \mathbf{b}^\top \boldsymbol{\theta} + c$$

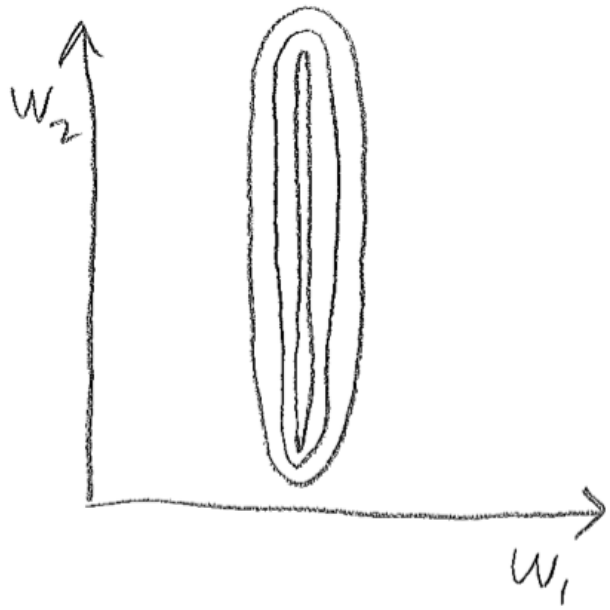
- Since a smooth cost function is well approximated by a convex quadratic (i.e. second-order Taylor approximation) in the vicinity of a (local) optimum, this analysis is a good description of the behavior of gradient descent near a (local) optimum.
- If the Hessian is ill-conditioned, then gradient descent makes slow progress towards the optimum.

Ill-conditioned curvature: normalization

- Suppose we have the following dataset for linear regression.

x_1	x_2	t
114.8	0.00323	5.1
338.1	0.00183	3.2
98.8	0.00279	4.1
⋮	⋮	⋮

$$\overline{w_i} = \overline{y}\, x_i$$

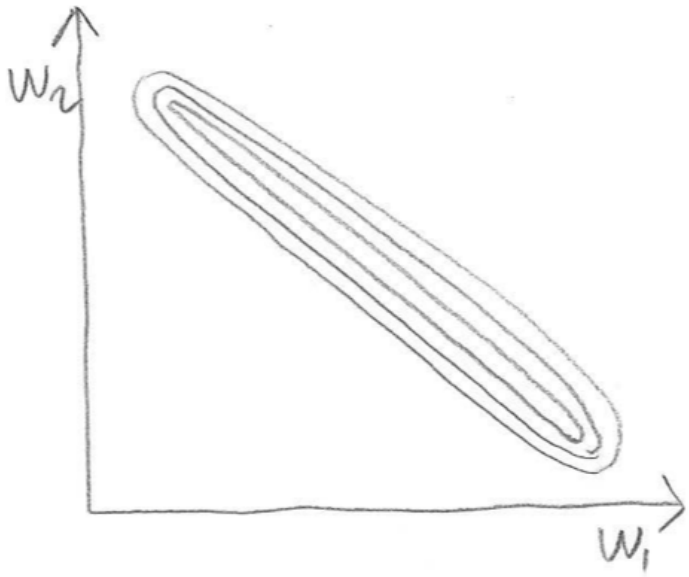


- Which weight, w_1 or w_2, will receive a larger gradient descent update?
- Which one do you want to receive a larger update?
- Note: the figure vastly *understates* the narrowness of the ravine!

Ill-conditioned curvature: normalization

- Or consider the following dataset:

x_1	x_2	t
1003.2	1005.1	3.3
1001.1	1008.2	4.8
998.3	1003.4	2.9
⋮	⋮	⋮



Ill-conditioned curvature: normalization

- To avoid these problems, it's a good idea to center your inputs to zero mean and unit variance, especially when they're in arbitrary units (feet, seconds, etc.).

$$\tilde{x}_j = \frac{x_j - \mu_j}{\sigma_j}$$

- Hidden units may have non-centered activations, and this is harder to deal with.
  - One trick: replace logistic units (which range from 0 to 1) with tanh units (which range from -1 to 1)
  - A method called batch normalization explicitly centers each hidden activation. It often speeds up training by 1.5-2x, and it's available in all the major neural net frameworks.

Momentum

- Unfortunately, even with these normalization tricks, ill-conditioned curvature is a fact of life. We need algorithms that are able to deal with it.
- Momentum is a simple and highly effective method. Imagine a hockey puck on a frictionless surface (representing the cost function). It will accumulate momentum in the downhill direction:

$$\mathbf{p} \leftarrow \mu\mathbf{p} - \alpha\frac{\partial\mathcal{J}}{\partial\boldsymbol{\theta}}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{p}$$

- α is the learning rate, just like in gradient descent.
- μ is a damping parameter. It should be slightly less than 1 (e.g. 0.9 or 0.99). Why not exactly 1?

Momentum

- Unfortunately, even with these normalization tricks, ill-conditioned curvature is a fact of life. We need algorithms that are able to deal with it.
- Momentum is a simple and highly effective method. Imagine a hockey puck on a frictionless surface (representing the cost function). It will accumulate momentum in the downhill direction:

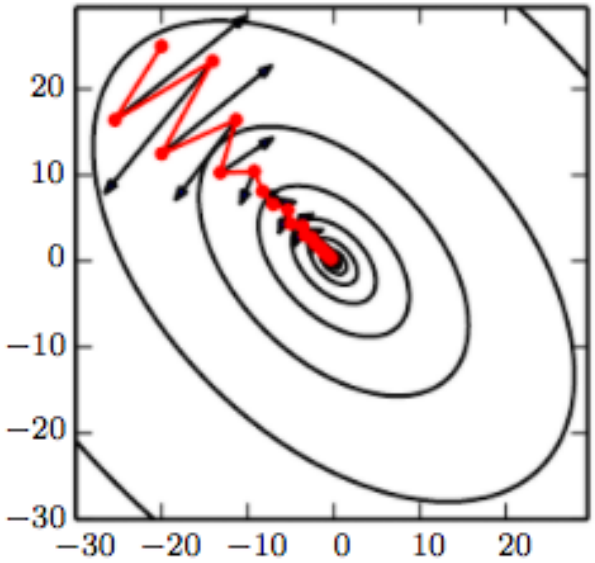
$$\mathbf{p} \leftarrow \mu\mathbf{p} - \alpha\frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{p}$$

- α is the learning rate, just like in gradient descent.
- μ is a damping parameter. It should be slightly less than 1 (e.g. 0.9 or 0.99). Why not exactly 1?
  - If $\mu = 1$, conservation of energy implies it will never settle down.

Momentum

- In the high curvature directions, the gradients cancel each other out, so momentum dampens the oscillations.
- In the low curvature directions, the gradients point in the same direction, allowing the parameters to pick up speed.



- If the gradient is constant (i.e. the cost surface is a plane), the parameters will reach a terminal velocity of

$$-\frac{\alpha}{1-\mu} \cdot \frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}}$$

  This suggests if you increase μ, you should lower α to compensate.

- Momentum sometimes helps a lot, and almost never hurts.

Ravines

- Even with momentum and normalization tricks, narrow ravines are still one of the biggest obstacles in optimizing neural networks.
- Empirically, the curvature can be many orders of magnitude larger in some directions than others!
- An area of research known as second-order optimization develops algorithms which explicitly use curvature information (second derivatives), but these are complicated and difficult to scale to large neural nets and large datasets.
- There is an optimization procedure called Adam which uses just a little bit of curvature information and often works much better than gradient descent. It's available in all the major neural net frameworks.

RMSprop and Adam

- Recall: SGD takes large steps in directions of high curvature and small steps in directions of low curvature.
- RMSprop is a variant of SGD which rescales each coordinate of the gradient to have norm 1 on average. It does this by keeping an exponential moving average s_j of the squared gradients.
- The following update is applied to each coordinate j independently:

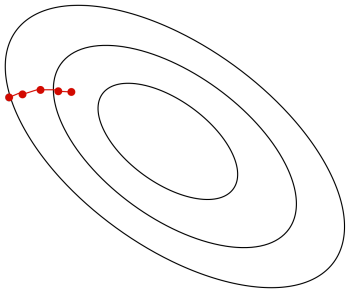
$$s_j \leftarrow (1-\gamma)s_j + \gamma[\frac{\partial \mathcal{J}}{\partial \theta_j}]^2$$

$$\theta_j \leftarrow \theta_j - \frac{\alpha}{\sqrt{s_j + \epsilon}} \frac{\partial \mathcal{J}}{\partial \theta_j}$$

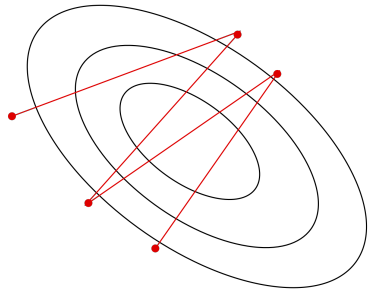
- If the eigenvectors of the Hessian are axis-aligned (dubious assumption), then RMSprop can correct for the curvature. In practice, it typically works slightly better than SGD.
- Adam = RMSprop + momentum
- Both optimizers are included in TensorFlow, Pytorch, etc.

Learning Rate

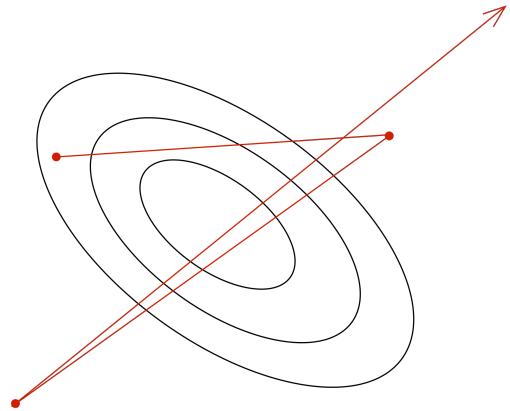
- The learning rate α is a hyperparameter we need to tune. Here are the things that can go wrong in batch mode:



α too small:
slow progress



α too large:
oscillations

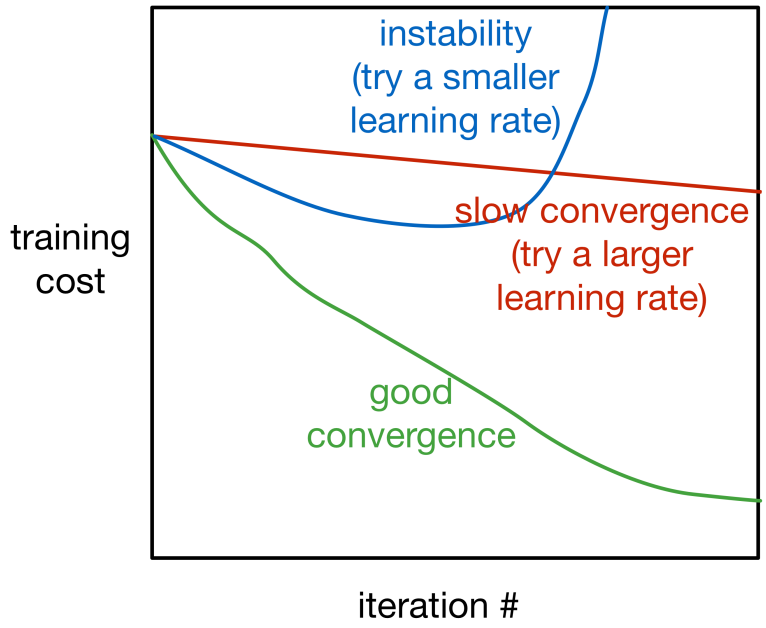


α much too large:
instability

- Good values are typically between 0.001 and 0.1. You should do a grid search if you want good performance (i.e. try $0.1, 0.03, 0.01, \ldots$).

Training Curves

- To diagnose optimization problems, it's useful to look at training curves: plot the training cost as a function of iteration.
- **Gotcha:** use a fixed subset of the training data to monitor the training error. Evaluating on a different batch (e.g. the current one) in each iteration adds a *lot* of noise to the curve!
- **Gotcha:** it's very hard to tell from the training curves whether an optimizer has converged. They can reveal major problems, but they can't guarantee convergence.



Stochastic Gradient Descent

- So far, the cost function $\mathcal{J}$ has been the average loss over the training examples:

$$\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{J}^{(i)}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(y(\mathbf{x}^{(i)}, \boldsymbol{\theta}), t^{(i)}).$$

- By linearity,

$$\nabla\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\nabla\mathcal{J}^{(i)}(\boldsymbol{\theta}).$$

- Computing the gradient requires summing over *all* of the training examples. This is known as batch training.
- Batch training is impractical if you have a large dataset (e.g. millions of training examples)!

Stochastic Gradient Descent

- Stochastic gradient descent (SGD): update the parameters based on the gradient for a single training example:

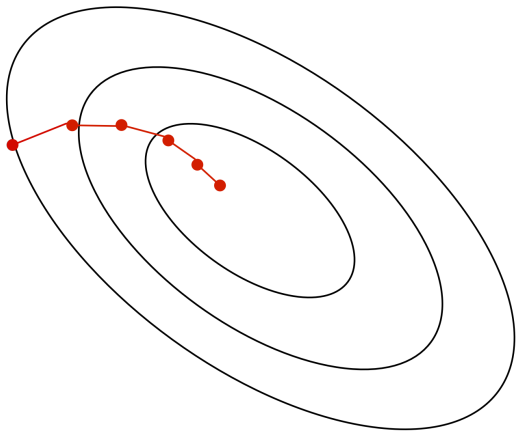
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla \mathcal{J}^{(i)}(\boldsymbol{\theta})$$

- SGD can make significant progress before it has even looked at all the data!
- Mathematical justification: if you sample a training example at random, the stochastic gradient is an unbiased estimate of the batch gradient:

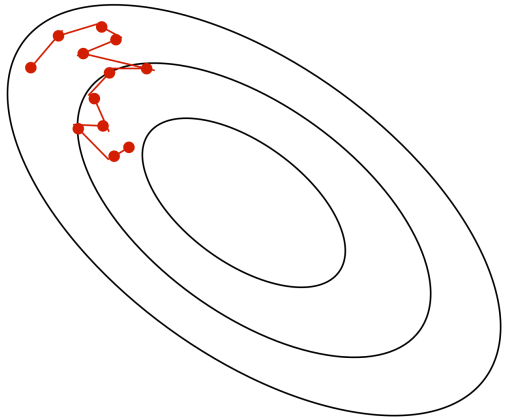
$$\mathbb{E}_i \left[\nabla \mathcal{J}^{(i)}(\boldsymbol{\theta}) \right] = \frac{1}{N} \sum_{i=1}^{N} \nabla \mathcal{J}^{(i)}(\boldsymbol{\theta}) = \nabla \mathcal{J}(\boldsymbol{\theta}).$$

Stochastic Gradient Descent

- Batch gradient descent moves directly downhill. SGD takes steps in a noisy direction, but moves downhill on average.



batch gradient descent



stochastic gradient descent

Stochastic Gradient Descent

- **Problem:** if we only look at one training example at a time, we can't exploit efficient vectorized operations.
- **Compromise approach:** compute the gradients on a medium-sized set of training examples, called a mini-batch.
- Each entire pass over the dataset is called an epoch.
- Stochastic gradients computed on larger mini-batches have smaller variance:

$$\mathrm{Var}\left[\frac{1}{S}\sum_{i=1}^{S}\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}\right] = \frac{1}{S^2}\mathrm{Var}\left[\sum_{i=1}^{S}\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}\right] = \frac{1}{S}\mathrm{Var}\left[\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}\right]$$

- The mini-batch size S is a hyperparameter. Typical values are 10 or 100.

Stochastic Gradient Descent: Batch Size

- The mini-batch size S is a hyperparameter that needs to be set.
  - **Large batches:** converge in fewer weight updates because each stochastic gradient is less noisy.
  - **Small batches:** perform more weight updates per second because each one requires less computation.

Stochastic Gradient Descent: Batch Size

- The mini-batch size S is a hyperparameter that needs to be set.
  - **Large batches:** converge in fewer weight updates because each stochastic gradient is less noisy.
  - **Small batches:** perform more weight updates per second because each one requires less computation.
- **Claim:** If the wall-clock time were proportional to the number of FLOPs, then $S = 1$ would be optimal.
  - 100 updates with $S = 1$ requires the same FLOP count as 1 update with $S = 100$.
  - Rewrite minibatch gradient descent as a for-loop:

S = 1

For $k = 1, \ldots, 100$:

$$\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} - \alpha \nabla \mathcal{J}^{(k)}(\boldsymbol{\theta}_{k-1})$$

S = 100

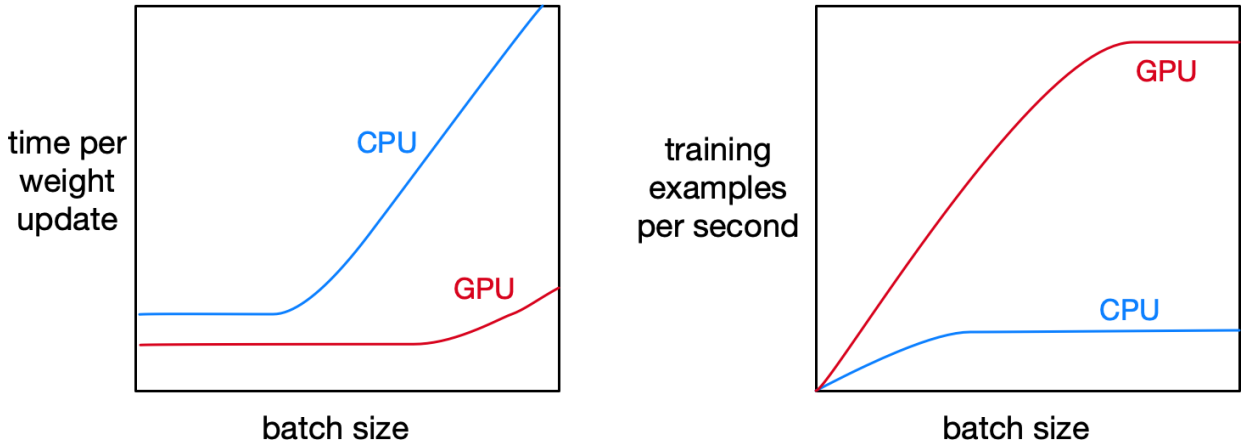
For $k = 1, \ldots, 100$:

$$\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} - \tfrac{\alpha}{100} \nabla \mathcal{J}^{(k)}(\boldsymbol{\theta}_0)$$

  - All else being equal, you'd prefer to compute the gradient at a fresher value of $\boldsymbol{\theta}$. So $S = 1$ is better.

Stochastic Gradient Descent: Batch Size

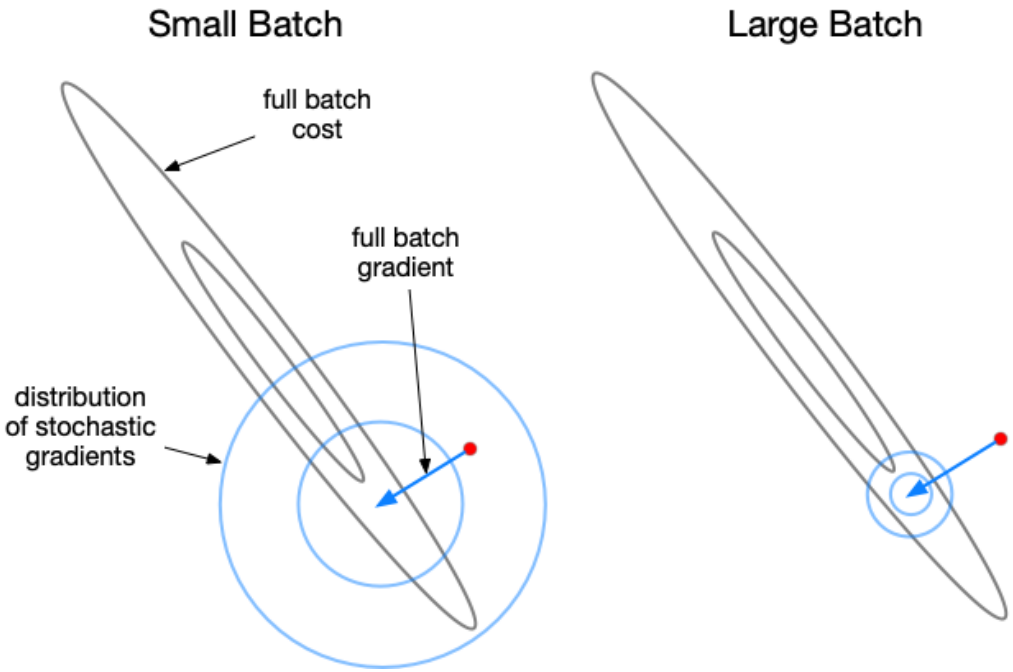
- The reason we don't use $S = 1$ is that larger batches can take advantage of fast matrix operations and parallelism.
- **Small batches:** An update with $S = 10$ isn't much more expensive than an update with $S = 1$.
- **Large batches:** Once S is large enough to saturate the hardware efficiencies, the cost becomes linear in S.
- Cartoon figure, not drawn to scale:

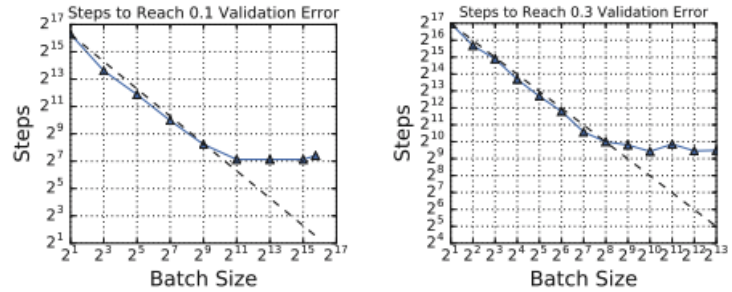


- Since GPUs afford more parallelism, they saturate at a larger batch size. Hence, GPUs tend to favor larger batch sizes.

Stochastic Gradient Descent: Batch Size

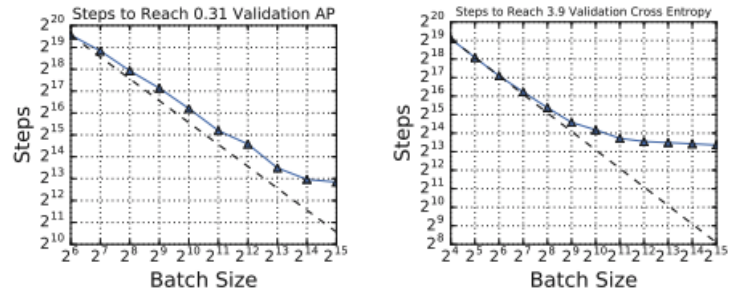
- The convergence benefits of larger batches also see diminishing returns.
- **Small batches:** large gradient noise, so large benefit from increased batch size
- **Large batches:** SGD approximates the batch gradient descent update, so no further benefit from variance reduction.





(b) Simple CNN on Fashion MNIST

(c) ResNet-8 on CIFAR-10



(e) ResNet-50 on Open Images

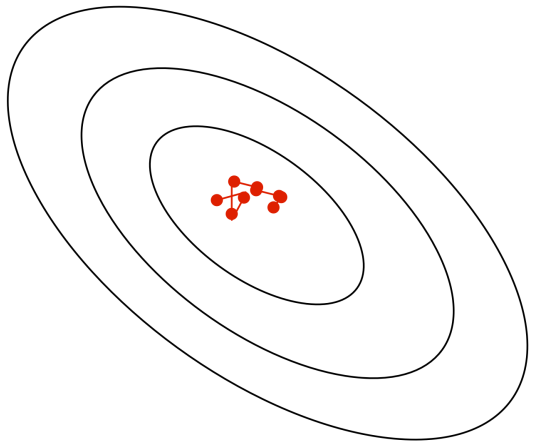
(f) Transformer on LM1B

- **Right:** # iterations to reach target validation error as a function of batch size. (Shallue et al., 2018)

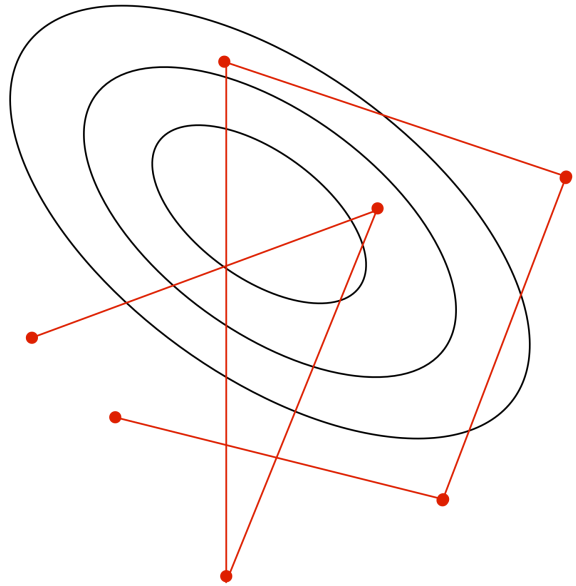
SGD Learning Rate

- In stochastic training, the learning rate also influences the fluctuations due to the stochasticity of the gradients.

small learning rate



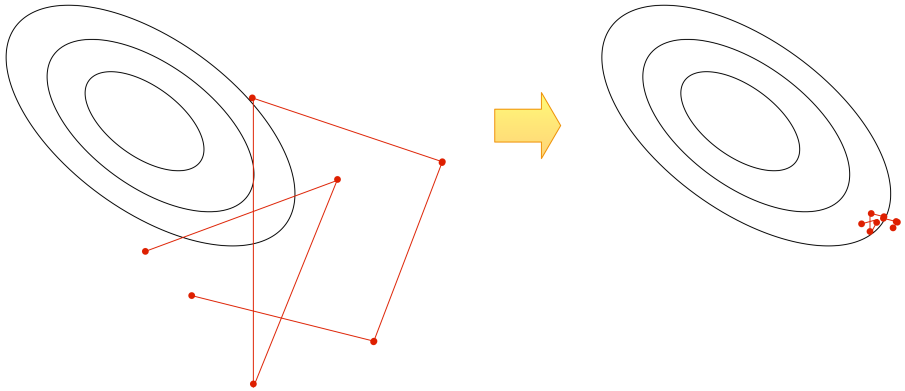
large learning rate

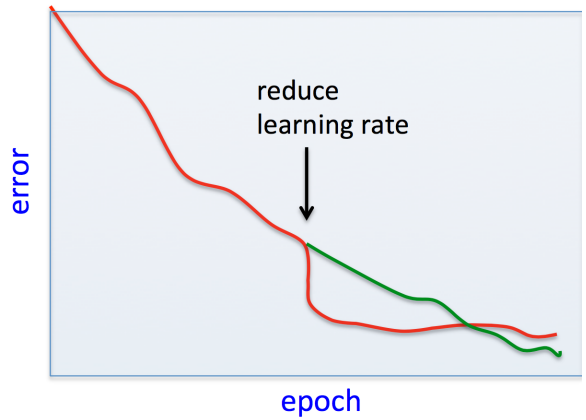


- Typical strategy:
  - Use a large learning rate early in training so you can get close to the optimum
  - Gradually decay the learning rate to reduce the fluctuations

SGD Learning Rate

- Warning: by reducing the learning rate, you reduce the fluctuations, which can appear to make the loss drop suddenly. But this can come at the expense of long-run performance.





Recap

Problem	Diagnostics	Workarounds
incorrect gradients	finite differences	fix them, or use autodiff
local optima	(hard)	random restarts
symmetries	visualize $\mathbf{W}$	initialize $\mathbf{W}$ randomly
slow progress	slow, linear training curve	increase α; momentum
instability	cost increases	decrease α
oscillations	fluctuations in training curve	decrease α; momentum
fluctuations	fluctuations in training curve	decay α; iterate averaging
dead/saturated units	activation histograms	initial scale of $\mathbf{W}$; ReLU
ill-conditioning	(hard)	normalization; momentum; Adam; second-order opt.