CSC413 Neural Networks and Deep Learning

Lecture 5: Optimization

February 6/8, 2024

Table of Contents

Lecture Plan

- Closer look at the optimization problem
- Different landscapes of an optimization problem
- Methods to mitigate the challenges of a difficult optimization landscape

Section 1

The Optimization Problem

The Optimization Problem

- So far, we've talked a lot about computing gradients and different neural models.
- How do we actually train those models using gradients?
- There are various things that can go wrong in gradient descent, we will learn what to do about them, e.g.
  - How to tune the learning rates.

Optimization Problem

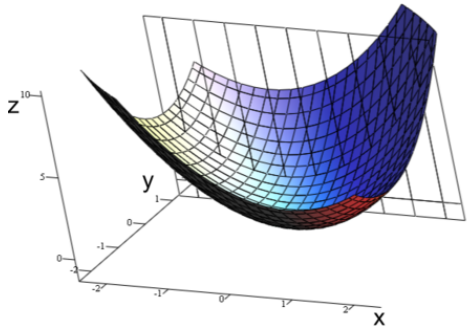
Let's group all the parameters (weights and biases) of a network into a single vector θ

We wish to find the minima of a function $f(\theta) : \mathbb{R}^D \to \mathbb{R}$.

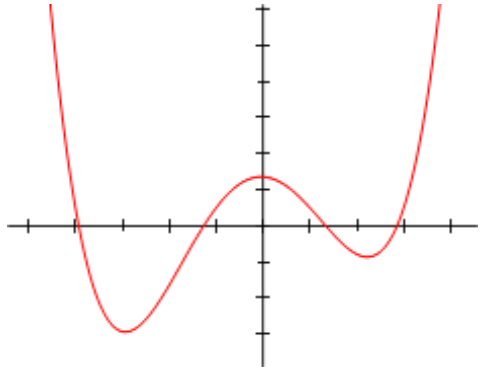
- What property does f need to have for gradient descent to work well?
- Are there techniques that can work better than (vanilla) gradient descent?
- Are there cases where gradient descent (and related) optimization methods fail?
- How can deep learning practitioners diagnose and fix optimization issues?

Optimization Landscape

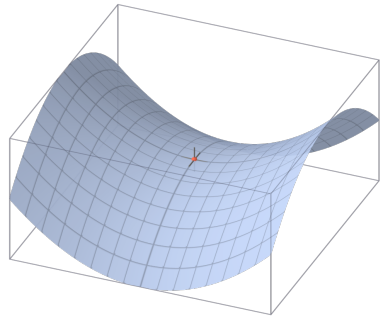
- The optimization landscape refers to how the function $f(\theta)$ changes as we vary θ.
- The landscape affects the behaviour of the optimization algorithm.
  - Not all landscapes are equal!
- We want to become more familiar with some common ones.



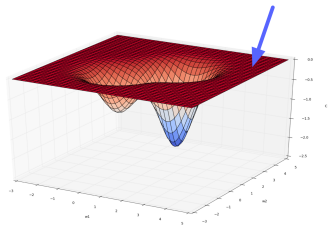
convex functions



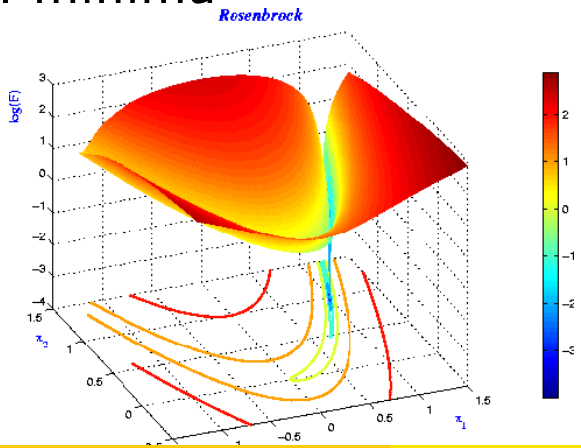
local minima



saddle points



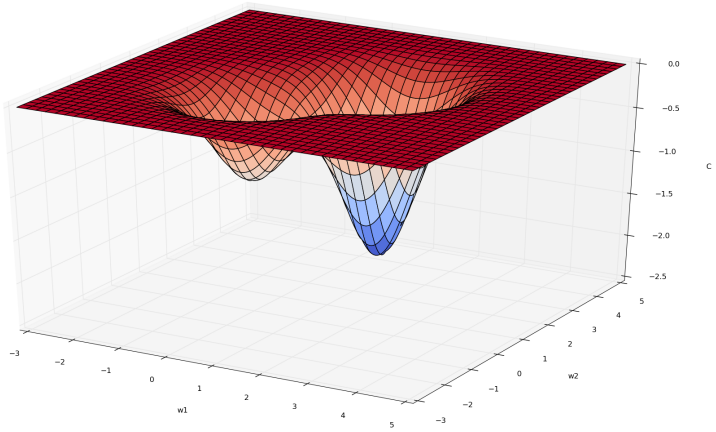
plateaux



Visualizing Optimization Problems

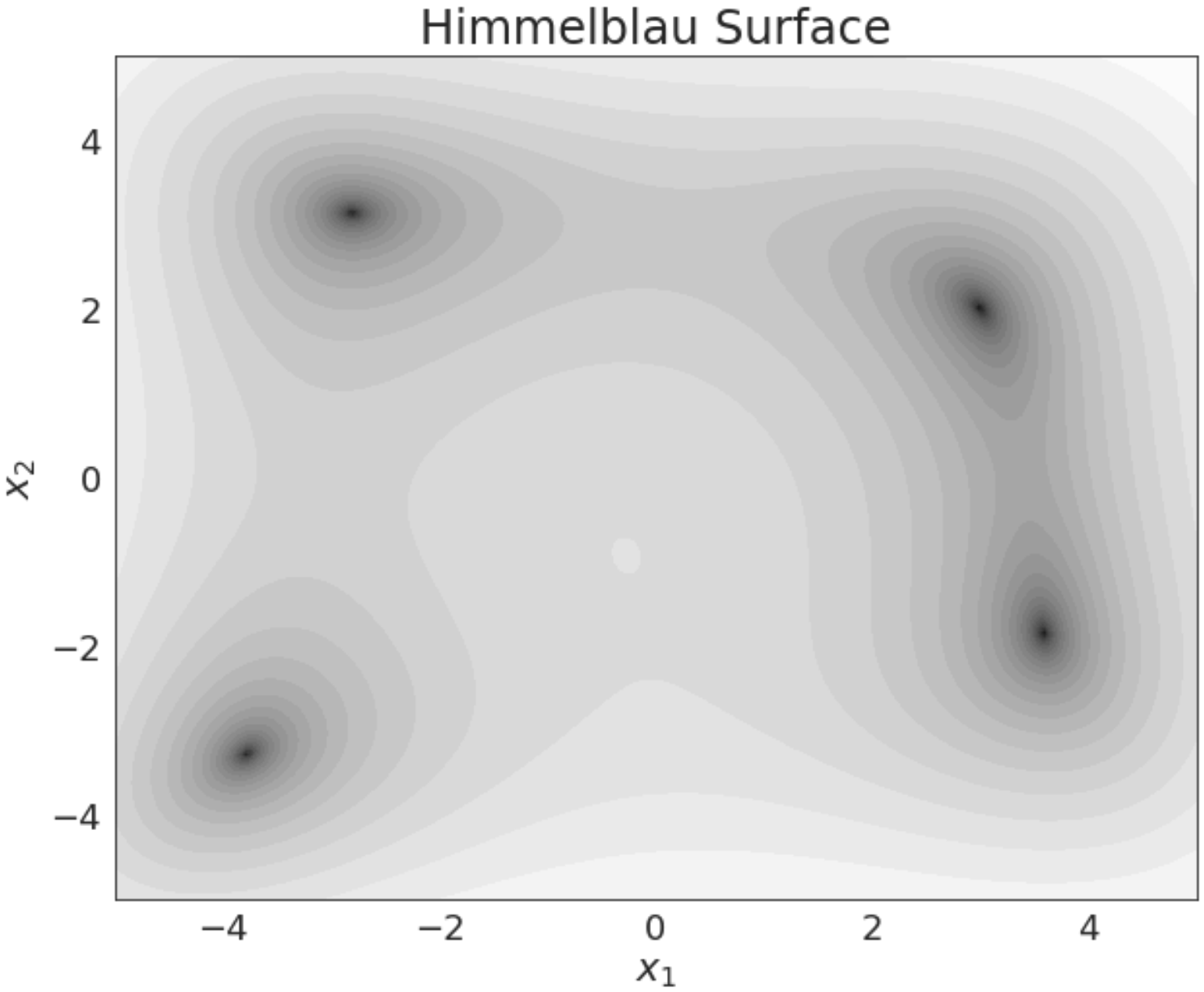
Visualizing optimization problems in high dimensions is challenging. Intuitions that we get from 1D and 2D optimization problems can be helpful.

In 1D and 2D, we can visualize f by drawing plots, e.g. surface plots and contour plots



Q: Sketch a contour plot that represents the same function as the figure above.

Review: Contour Plots



Section 2

Optimization of Univariate Functions

Optimization of Univariate Functions

Suppose we have a function $f(\theta) : \mathbb{R}^1 \to \mathbb{R}$ that we wish to minimize. How do we go about doing this?

$\theta_{\text{new}} \leftarrow \theta_{\text{old}} - \alpha f'(\theta_{\text{old}})$

- Why is GD a good idea?

Optimization of Univariate Functions

The **Taylor series expansion** of the function can give us some intuition about why GD is a good idea.

Recall that the Taylor series expansion of a (sufficiently differentiable) function $f(\theta)$ around θ_0 is

$$f(\theta) \approx f(\theta_0) + f'(\theta_0)(\theta - \theta_0) + \frac{1}{2}f''(\theta_0)(\theta - \theta_0)^2 + \frac{1}{3!}f'''(\theta_0)(\theta - \theta_0)^3 + \cdots$$

Before talking about GD, let us see what the Taylor series expansion can tell us about the function.

Behaviour of a Function around its Critical Point

How does a function look like around its critical point θ^*?

Recall that the critical point is where its derivative is zero, so $f'(\theta^*) = 0$. * A critical point might be minimum or maximum or a saddle point.

We use the Taylor series expansion:

$$\begin{aligned} f(\theta) &\approx f(\theta^*) + f'(\theta^*)(\theta - \theta^*) + \frac{1}{2}(\theta - \theta^*)^2 f''(\theta^*) \\ &= f(\theta^*) + \frac{1}{2}(\theta - \theta^*)^2 f''(\theta^*). \end{aligned}$$

This is a quadratic function!

- If $f''(\theta^*) > 0$, it increases as we get farther from θ^*.
  - θ^* is a minimum!
- If $f''(\theta^*) < 0$, it decreases as we get farther from θ^*.
  - θ^* is a maximum!

Optimization in 1D – First-Order Method

Consider the first-order Taylor series expansion of $f(\theta)$ around point θ_0:

$$f(\theta) \approx f(\theta_0) + f'(\theta_0)(\theta - \theta_0)$$

Instead of optimizing $f(\theta)$, we can try to optimize its approximation $f(\theta_0) + f'(\theta_0)(\theta - \theta_0)$.

So we solve

$$\theta \leftarrow \arg\min_{\theta} f(\theta_0) + f'(\theta_0)^\top(\theta - \theta_0)$$

- This is a linear function of θ, so the minimizer of it pushes θ to either $+\infty$ or $-\infty$ (unless $f'(\theta_0)$ is exactly zero).
- This is not a good solution because this approximation is only valid in a neighbourhood of θ_0.

Optimization in 1D – First-Order Method

Let us penalize solutions that get far from θ_0 by adding a $\frac{\lambda}{2}|\theta - \theta_0|^2$ term. We solve

$$\theta \leftarrow \arg\min_{\theta} \left\{ f(\theta_0) + f'(\theta_0)(\theta - \theta_0) + \frac{\lambda}{2}|\theta - \theta_0|^2 \right\}.$$

Larger λ indicates that we do not want to deviate far from θ_0.

If we solve this, we get that

$$\theta \leftarrow \theta_0 - \frac{1}{\lambda} f'(\theta_0).$$

This is GD with the learning rate $\alpha = \frac{1}{\lambda}$.

Optimization in 1D – Second-Order Method

What if we started from the second-order Taylor series approximation of $f(\theta)$ instead?

$$f(\theta) \approx f(\theta_0) + f'(\theta_0)(\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)^2 f''(\theta_0)$$

and then solved

$$\theta \leftarrow \arg\min_{\theta} \left\{ f(\theta_0) + f'(\theta_0)(\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)^2 f''(\theta_0) \right\}.$$

Let us take the derivative w.r.t. θ and make it equal to zero:

$$f'(\theta_0) + (\theta - \theta_0) f''(\theta_0) = 0 \Rightarrow \theta = \theta_0 - \frac{1}{f''(\theta_0)} f'(\theta_0).$$

This is the **Newton** method! It uses both first and second derivatives of a function. It is an example of a **second-order** optimization method.

The Dynamics of Optimizers

How can we compare GD vs. Newton method?

Let us look at the sequence of (approximate) solutions they find. To simplify our exposition, suppose that the function we want to minimize is $f(\theta) = \frac{c}{2}\theta^2$ for some curvature parameter c. Its minimum is at $\theta^* = 0$. Our initial point is θ_0.

For GD with learning rate α, as $f'(\theta) = c\theta$, we have

$$\theta_{k+1} \leftarrow \theta_k - \alpha f'(\theta_k) = \theta_k - \alpha c\theta_k = (1 - c\alpha)\theta_k.$$

By induction, we get that $\theta_k = (1 - c\alpha)^k \theta_0$.

The Dynamics of Optimizers

$\theta_k = (1 - c\alpha)^k \theta_0$.

- When does this sequence converge to $\theta^* = 0$?
  - If $-1 < 1 - c\alpha < 1$, at every step, θ_k becomes smaller (closer to $\theta^* = 0$). This is equivalent of having

$$0 < \alpha < \frac{2}{c}.$$

So if the curvature parameter c is large, we need to choose a smaller step size to ensure convergence.

If we knew the curvature, we could choose our step size to be $\alpha = \frac{1}{c}$ and solve the problem in one step.

The Dynamics of Optimizers

For the Newton method, notice that $f''(\theta) = c$ (for any θ), so we have

$$\theta_1 = \theta_0 - \frac{1}{f''(\theta_0)} f'(\theta_0) = \theta_0 - \frac{1}{c} c\theta_0 = \theta_0 - \theta_0 = 0.$$

This means that the Newton method can find the solution in one step, without any need to tune the learning rate.

The situation is a bit more complicated for multivariate functions. Let's look at it!

Section 3

Optimization of Multivariate Functions

Taylor Series Expansion of Multivariate Functions

Suppose we have a function $f(\theta) : \mathbb{R}^D \to \mathbb{R}$ that we wish to minimize.

Again, we can explore approximating f with its **Taylor series expansion**:

$$f(\theta) \approx f(\theta_0) + \nabla f(\theta_0)^\top (\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)^T H(\theta_0)(\theta - \theta_0) + \cdots$$

Recap: Gradient

The **gradient** of a function $f(\theta) : \mathbb{R}^D \to \mathbb{R}$ is the vector of partial derivatives:

$$\nabla_\theta f = \frac{\partial f}{\partial \theta} = \begin{bmatrix} \frac{\partial f}{\partial \theta_1} \\ \frac{\partial f}{\partial \theta_2} \\ \vdots \\ \frac{\partial f}{\partial \theta_D} \end{bmatrix}$$

Recap: Hessian

The **Hessian Matrix**, denoted **H** or $\nabla^2 f$ is the matrix of second derivatives

$$H = \nabla^2 f = \begin{bmatrix} \frac{\partial^2 f}{\partial \theta_1^2} & \frac{\partial^2 f}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 f}{\partial \theta_1 \partial \theta_D} \\ \frac{\partial^2 f}{\partial \theta_1 \partial \theta_2} & \frac{\partial^2 f}{\partial \theta_2^2} & \cdots & \frac{\partial^2 f}{\partial \theta_2 \partial \theta_D} \\ \vdots \frac{\partial^2 f}{\partial \theta_n \partial \theta_2} & \frac{\partial^2 f}{\partial \theta_n \partial \theta_2} & \cdots & \frac{\partial^2 f}{\partial \theta_D^2} \end{bmatrix}$$

The Hessian is symmetric because $\frac{\partial^2 f}{\partial \theta_i \partial \theta_j} = \frac{\partial^2 f}{\partial \theta_j \partial \theta_i}$

Multivariate Taylor Series

Recall the second-order **Taylor series expansion** of f:

$$f(\theta) \approx f(\theta_0) + \nabla f(\theta_0)^T(\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)^T H(\theta_0)(\theta - \theta_0)$$

A **critical point** of f is a point where the gradient is zero, so that

$$f(\theta) \approx f(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T H(\theta_0)(\theta - \theta_0)$$

How do we know if the critical point is a maximum, minimum, or something else?

- **Minimum**: The Hessian is positive definite.
- **Maximum**: The Hessian is negative definite.

Spectral Decomposition of H

$$f(\theta) \approx f(\theta_0) + f'(\theta_0)^T(\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)^2 f''(\theta_0) + \dots$$

We won't go into details in this course, but...

- A lot of important features of the optimization landscape can be characterized by the eigenvalues of the Hessian H.
- Recall that a symmetric matrix (such as H) has only real eigenvalues, and there is an orthogonal basis of eigenvectors.
- This can be expressed in terms of the **spectral decomposition**: $H = Q\Lambda Q^T$, where Q is an orthogonal matrix (whose columns are the eigenvectors) and Λ is a diagonal matrix (whose diagonal entries are the eigenvalues).

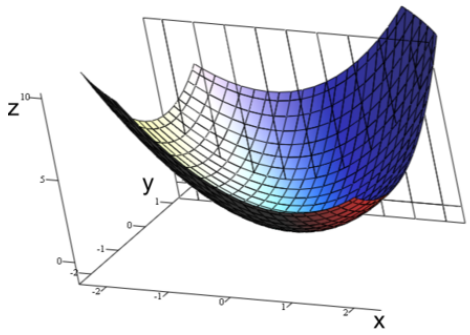
First-Order vs Second-Order Information and Optimizer

- First-order methods use information about the gradient.
  - Gradient Descent: $\theta_{k+1} \leftarrow \theta_k - \alpha \nabla f(\theta_k)$.
- The second-order methods use information about the **Hessian**.
  - Newton: $\theta_{k+1} \leftarrow \theta_k - H(\theta_k)^{-1} \nabla f(\theta_k)$.
  - These methods require the information about Hessian and need to (approximately) compute its inverse. This may not be scalable for large neural networks.

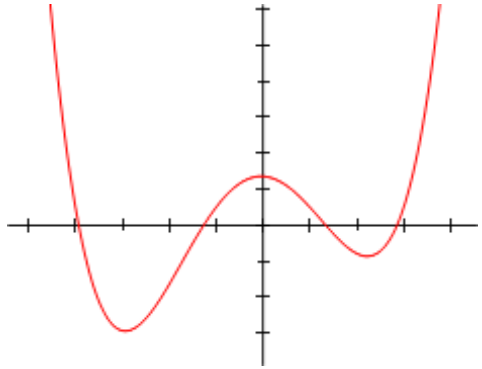
Section 4

Features of the Optimization Landscape

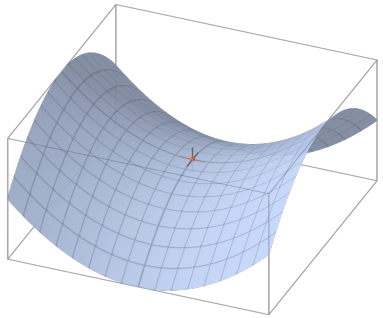
Features of the Optimization Landscape



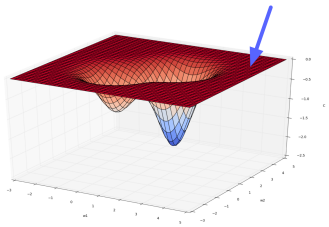
convex functions



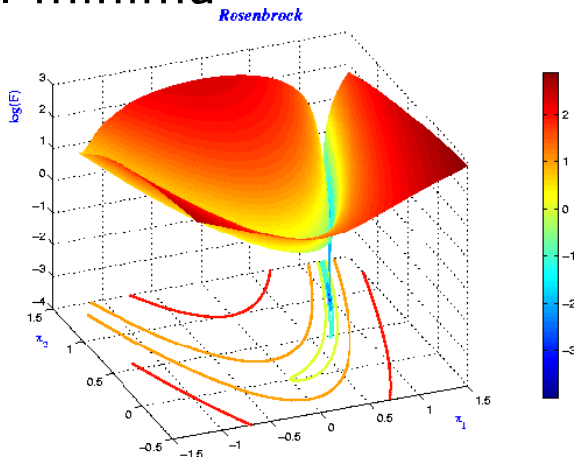
local minima



saddle points



plateaux



narrow ravines

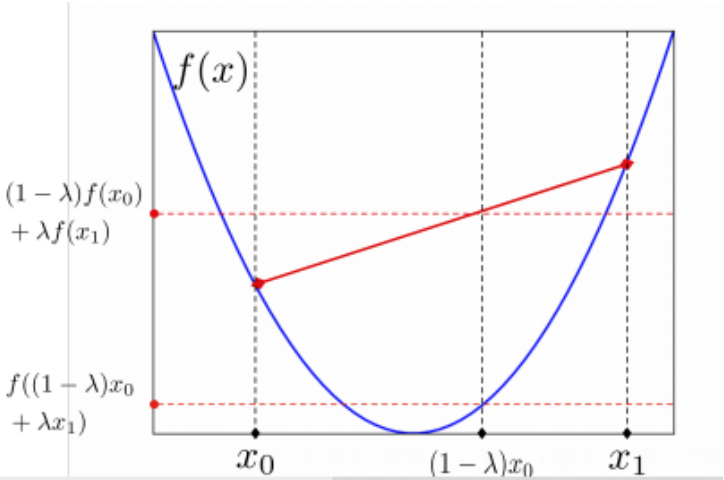
Feature 1: Convexity of Linear Models

Linear regression and logistic regressions are **convex** problems—i.e. its loss function is convex.

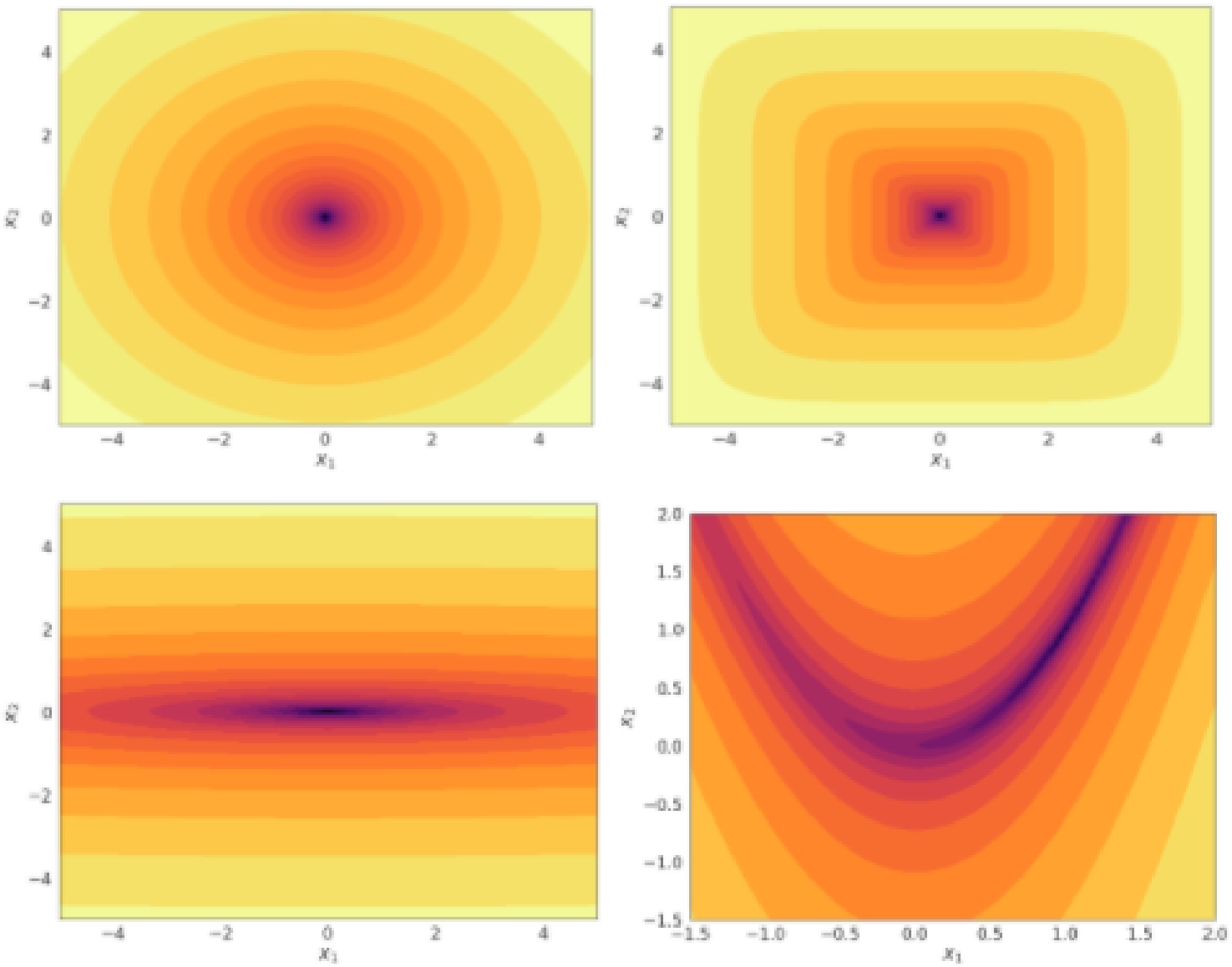
A function f is convex if for any $a \in (0, 1)$

$$f(ax + (1 - a)y) \leq af(x) + (1 - a)f(y)$$

- The cost function only has one minima.
- There are no local minima that is not global minima.
- Intuitively: the cost function is "bowl-shaped".



Q: Are these loss surfaces convex?



Convexity in 1D

How do we know if a function $f : \mathbb{R} \to \mathbb{R}$ is convex?

Convexity in 1D

How do we know if a function $f : \mathbb{R} \to \mathbb{R}$ is convex?

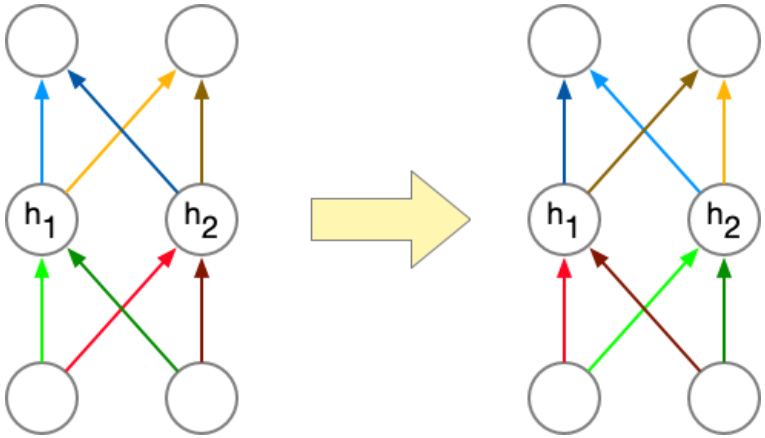
When $f''(x)$ is positive everywhere!

Likewise, analyzing the Hessian matrix H tells us whether a function $f : \mathbb{R}^D \to \mathbb{R}$ is convex. (Hint: H needs to have only positive eigenvalues)

Neural Networks are Not Convex

In general, neural networks are **not convex**.

One way to see this is that neural networks have **weight space symmetry**:



- Suppose you are at a local minima θ.
- You can swap two hidden units, and therefore swap the corresponding weights/biases, and get θ',
- then θ' must also be a local minima!

Video: https://play.library.utoronto.ca/watch/78367fbca4c4a42a30ec5862cdd0c756

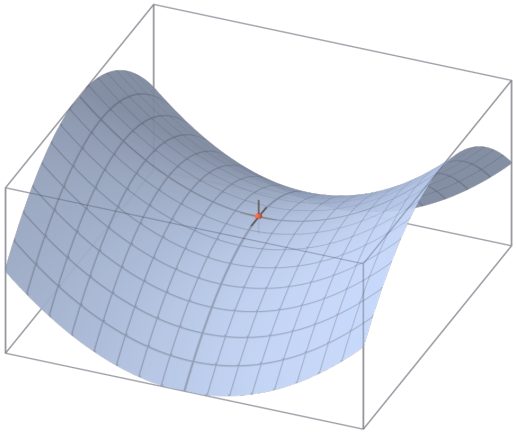
Feature 2: Local Minima in Neural Networks

Even though any multilayer neural net can have local optima, we usually don't worry too much about them.

It's possible to construct arbitrarily bad local minima even for ordinary classification MLPs.

Over the decade, theoreticians have made lots of progress proving gradient descent converges to global minima for some non-convex problems, including some specific neural net architectures.

Feature 3: Saddle Points



A saddle point has $\nabla_\theta \mathcal{E} = \mathbf{0}$, even though we are not at a minimum.

Minima with respect to some directions, maxima with respect to others. In other words, H has some positive and some negative eigenvalues.

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.

Initialization

- If we initialize all weights/biases to the same value, (e.g. 0)
- . . . then all the hidden states in the same layer will have the same value, (e.g. $\mathbf{h}$ will be a vector containing the same value repeated)
- . . . then all of the error signals for weights in the same layer are the same. (e.g. each row of $\overline{W^{(2)}}$ will be identical)

$$\begin{aligned}
\bar{\mathbf{y}} &= \overline{\mathcal{L}}(\mathbf{y} - \mathbf{t}) \\
\overline{W^{(2)}} &= \bar{\mathbf{y}}\mathbf{h}^T \\
\bar{\mathbf{h}} &= {W^{(2)}}^T \bar{\mathbf{y}} \\
\bar{\mathbf{z}} &= \bar{\mathbf{h}} \circ \sigma'(\mathbf{z}) \\
\overline{W^{(1)}} &= \bar{\mathbf{z}}\mathbf{x}^T
\end{aligned}$$

Random Initialization

Solution: don't initialize all your weights to zero!

Instead, *break the symmetry* by using small random values.

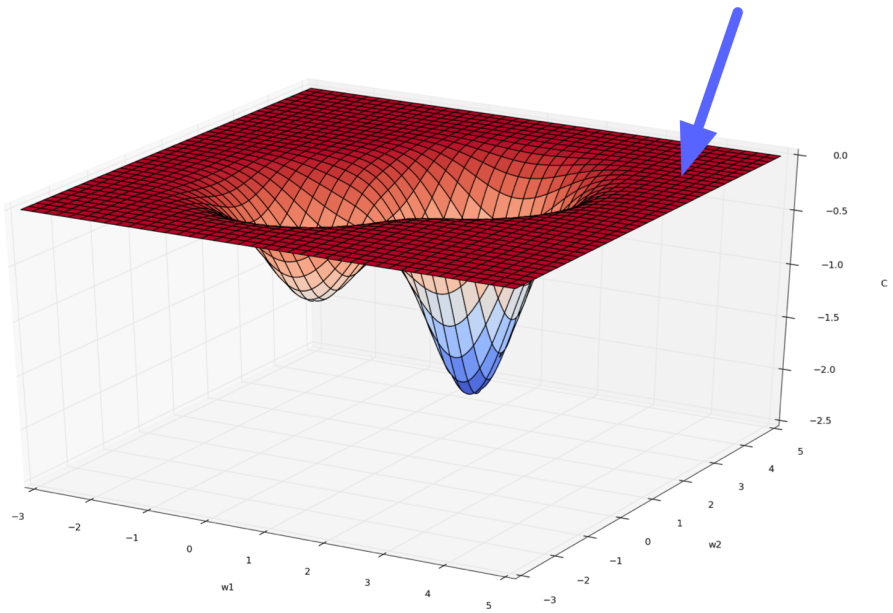
For example, we can initialize the weights by sampling from a random normal distribution with:

- Mean $= 0$
- Variance $= \frac{2}{\text{fan\_in}}$ where `fan_in` is the number of input neurons that feed into this feature. (He et al. 2015)

`openaccess.thecvf.com/content_iccv_2015/html/He_Delving_Deep_into_ICCV_2015_paper.html`

Feature 4: Plateaux

A flat region in the cost is called a **plateau**. (Plural: plateaux)



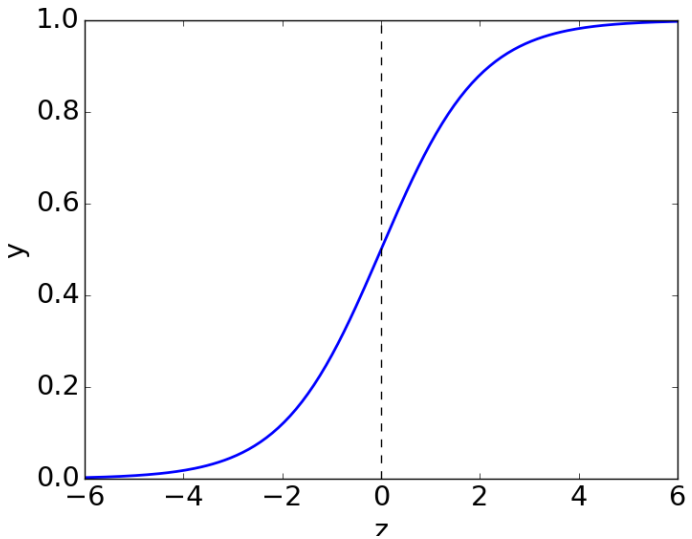
Can you think of examples?

- logistic activation with least squares
- 0-1 loss
- ReLU activation (potentially)

Plateaux and Saturated Units

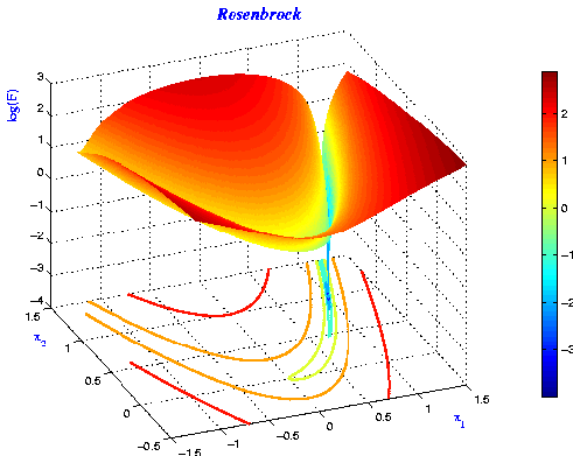
An important example of a plateau is a **saturated unit**. This is when activations always end up in the flat region of its activation function. Recall the backprop equation for the weight derivative:

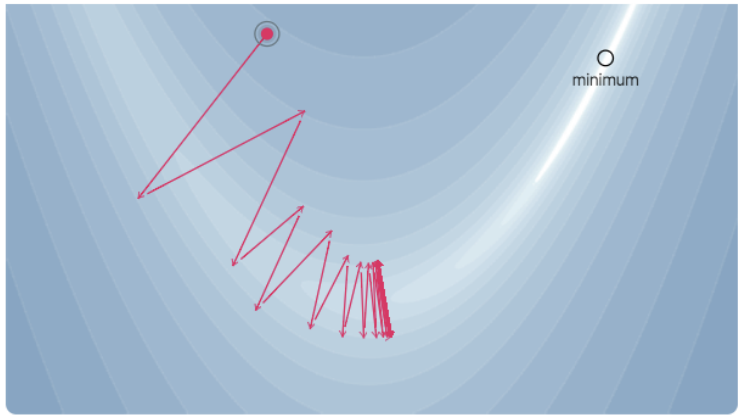
$$\overline{z_i} = \overline{h_i}\,\phi'(z), \qquad \overline{w_{ij}} = \overline{z_i}\,x_j$$



- If $\phi'(z)$ 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 neuron a **dead unit**.

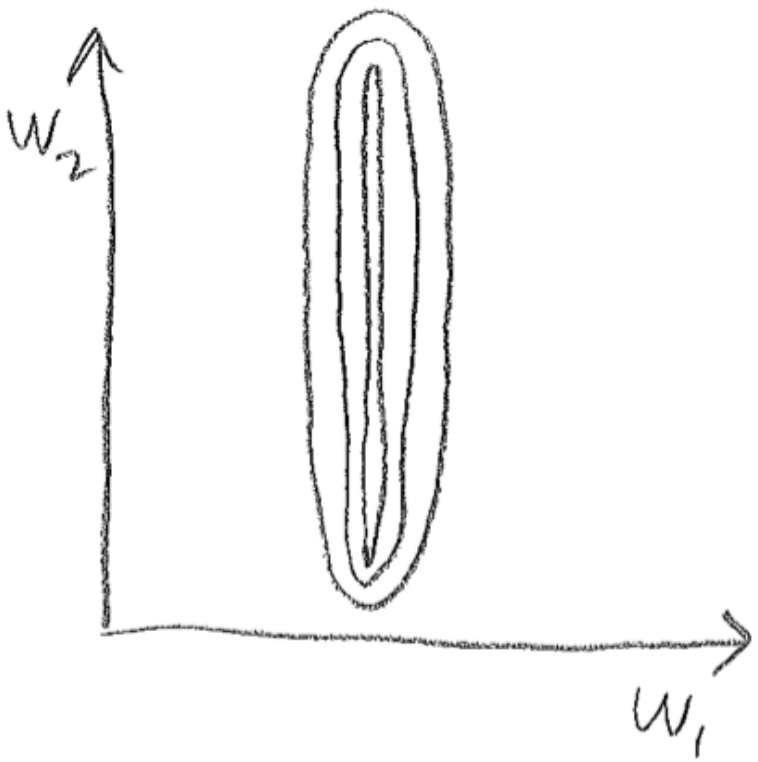
Ravines





Lots of sloshing around the walls, only a small derivative along the slope of the ravine's floor.

Ravines (2D Intuition)



- Gradient component $\frac{\partial \mathcal{E}}{\partial w_1}$ is large
- Gradient component $\frac{\partial \mathcal{E}}{\partial w_2}$ is small

Gradient Descent Dynamics at an Ill-Conditioned Curvature

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

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

  where $\mathbf{A}$ is PSD.
- Gradient descent update:

$$\begin{aligned}\theta_{k+1} &\leftarrow \theta_k - \alpha\nabla\mathcal{E}(\theta_k)\\ &= \theta_k - \alpha\mathbf{A}\theta_k\\ &= (\mathbf{I} - \alpha\mathbf{A})\theta_k\end{aligned}$$

- Solving the recurrence,

$$\theta_k = (\mathbf{I} - \alpha\mathbf{A})^k\theta_0$$

Gradient Descent Dynamics at an Ill-Conditioned Curvature

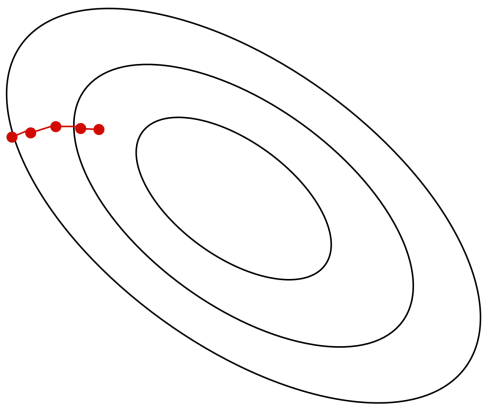
- We can analyze matrix powers such as $(\mathbf{I} - \alpha\mathbf{A})^k\theta_0$ using the spectral decomposition.
- Let $\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top$ be the spectral decomposition of $\mathbf{A}$.

$$\begin{aligned}(\mathbf{I} - \alpha\mathbf{A})^k\theta_0 &= (\mathbf{I} - \alpha\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top)^k\theta_0 \\ &= [\mathbf{Q}(\mathbf{I} - \alpha\mathbf{\Lambda})\mathbf{Q}^\top]^k\theta_0 \\ &= \mathbf{Q}(\mathbf{I} - \alpha\mathbf{\Lambda})^k\mathbf{Q}^\top\theta_0\end{aligned}$$

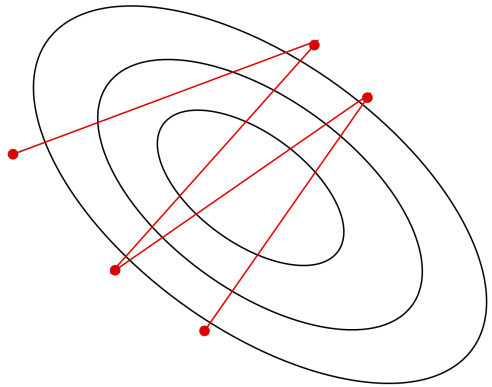
- Hence, in the $\mathbf{Q}$ basis, each coordinate gets multiplied by $(1 - \alpha\lambda_i)^k$, where the λ_i are the eigenvalues of $\mathbf{A}$.
- Cases:
  - $0 < \alpha\lambda_i \leq 1$: decays to 0 at a rate that depends on $\alpha\lambda_i$
  - $1 < \alpha\lambda_i \leq 2$: oscillates
  - $\alpha\lambda_i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

Tuning Learning Rate

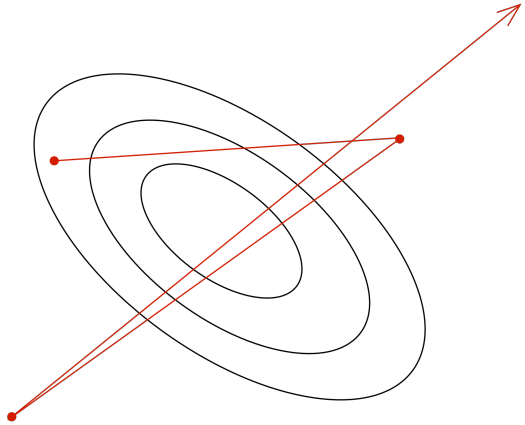
- How can spectral decomposition help?
- The learning rate α is a hyperparameter we need to tune. Here are the things that can go wrong:



α too small: slow progress



α too large: oscillations



α much too large: instability (diverges)

Gradient Descent Dynamics at an Ill-Conditioned Curvature

- 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)
- **Ill-conditioned curvature bounds the maximum learning rate choice**. 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.

Gradient Descent Dynamics at an Ill-Conditioned Curvature

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

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

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

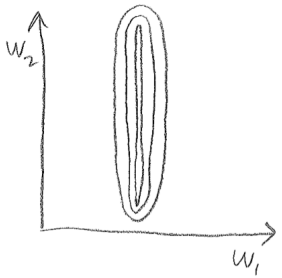
$$\mathcal{E}(\theta) = \frac{1}{2}\theta^\top \mathbf{A}\theta + \mathbf{b}^\top\theta + c$$

- 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.

Ravines: Example

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
⋮	⋮	⋮

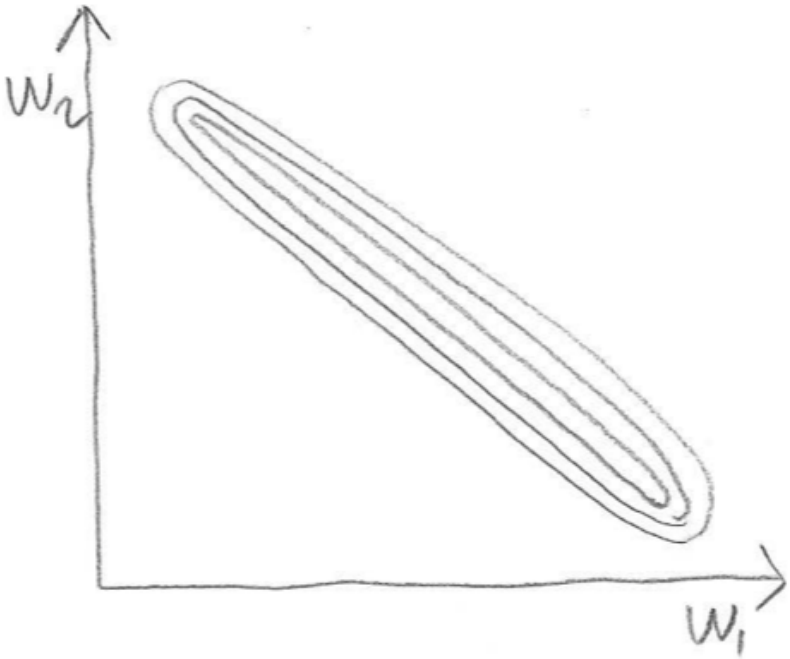


$$\overline{w_i} = \overline{y}\, x_i$$

- 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!

Ravines: Another Example

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



Avoiding Ravines

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

Hidden units may have non-centred activations, and this is harder to deal with.

A method called **batch normalization** explicitly centres each hidden activation. It often speeds up training by 1.5-2x.

Method: Batch Normalization

Idea: Normalize the activations **per batch** during training, so that the activations have zero mean and unit variance.

What about during test time (i.e. during model evaluation)?

- Keep track of the activation mean μ and variance σ during training.
- Use that μ and σ at test time: $z' = \frac{z-\mu}{\sigma}$.

Batch Normalization Video: https://play.library.utoronto.ca/watch/3e2b87ac8e5730f404893ce9270b4b75

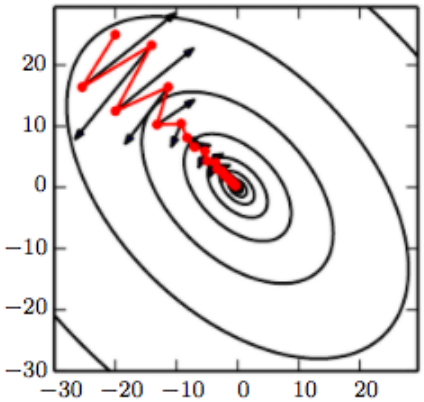
Method: Momentum

Momentum is a simple and highly effective method to deal with narrow ravines. Imagine a hockey puck on a frictionless surface (representing the cost function). It will accumulate momentum in the downhill direction:

$$m_{k+1} \leftarrow \beta_1 m_k + \alpha \frac{\partial \mathcal{E}(\theta_k)}{\partial \theta}$$
$$\theta \leftarrow \theta - m_{k+1}$$

- α is the learning rate, just like in gradient descent.
- β_1 is a damping parameter. It should be slightly less than 1 (e.g. 0.9 or 0.99).
  - If $\beta_1 = 1$, conservation of energy implies it will never settle down.
- m is a (weighted) average of the recent gradients.

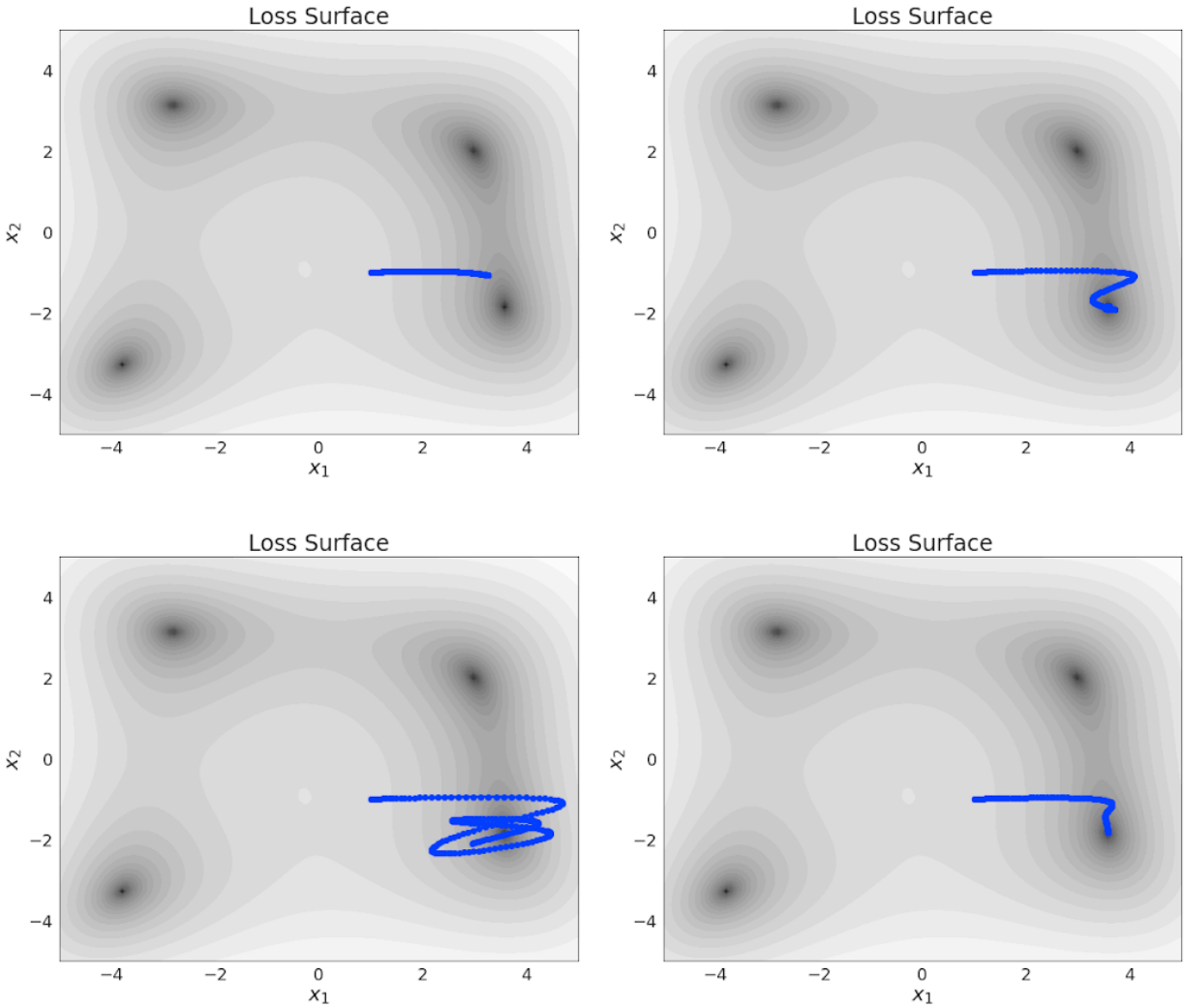
Why Momentum Works



- 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 $-\frac{\alpha}{1-\beta_1} \cdot \frac{\partial \mathcal{E}}{\partial \theta}$ This suggests if you increase β_1, you should lower α to compensate.
- Momentum sometimes helps a lot, and almost never hurts.

Gradient Descent with Momentum

Q: Which trajectory has the highest/lowest momentum setting?



Second-Order Information

Recall that the GD update for the quadratic function $\mathcal{E}(\theta) = \frac{1}{2}\theta^\top \mathbf{A}\theta$ is

$$\begin{aligned}\theta_{k+1} &\leftarrow \theta_k - \alpha\nabla\mathcal{E}(\theta_k)\\ &= \theta_k - \alpha\mathbf{A}\theta_k\\ &= (\mathbf{I} - \alpha\mathbf{A})\theta_k.\end{aligned}$$

The speed of convergence is greatly affected by $\mathbf{A}$ being ill-conditioned because we have to choose the learning rate α very small.

What if we used the second-order information?

Second-Order Information

The Newton method is

$$\begin{aligned}\theta_{k+1} &\leftarrow \theta_k - H(\theta_k)^{-1}\nabla f(\theta_k) \\ &= \theta_k - A^{-1}A\theta_k = 0.\end{aligned}$$

Intuition: The multiplication by the inverse of Hessian (A^{-1} here) rescales all dimensions. Even if A is "stretched" in some direction, hence has a ravine, its effect is removed after multiplication by the inverse.

Second-Order Information

Newton method is an example of a second-order optimization, which are methods that explicitly use curvature information (the Hessian H).

These method are computationally expensive and difficult to scale to large neural nets and large datasets.

- Q: What is the size of a Hessian for a fully connected neural network with input size 1000, the first hidden layer with size 500, the second hidden layer with size 500, and the output with the size of 10?

Can we use just a bit of second-order information?

- Let us consider a few examples: RMSProp and Adam

Method: RMSProp

SGD takes too large of a step in directions of high curvature and too small of a step 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 of the squared gradients.

`arxiv.org/abs/1308.0850`

Method: RMSProp

RMSProp is the following update

$$s_{k+1} \leftarrow \beta_2 s_k + (1-\beta_2)\left[\frac{\partial \mathcal{E}(\theta_k)}{\partial \theta}\right]^2$$

$$\theta_{k+1} \leftarrow \theta_k - \frac{\alpha}{\sqrt{s_{k+1}+\epsilon}}\frac{\partial \mathcal{E}(\theta_k)}{\partial \theta}$$

where the operations are performed component-wise:

- $(\frac{\partial \mathcal{E}(\theta_k)}{\partial \theta})^2$ is a vector with its j-th dimension being $(\frac{\partial \mathcal{E}(\theta_k)}{\partial \theta_j})^2$,
- and likewise for the division.

Here s is the weighted average of the squared size of the gradient along each dimension. The hyperparameer $\beta_2 \in (0,1)$ determines how the recency of the weighting.

If the eigenvectors of the Hessian are axis-aligned (dubious assumption), RMSprop can correct for the curvature. In practice, it typically works slightly better than SGD.

Method: Adam

Adam = RMSprop + Momentum

Adam is the very commonly used optimizer for neural network. Adam's update rule (simplified) is:

$$m_{k+1} \leftarrow \beta_1 m_k + (1-\beta_1)\frac{\partial \mathcal{E}(\theta_k)}{\partial \theta},$$
$$s_{k+1} \leftarrow \beta_2 s_k + (1-\beta_2)\left[\frac{\partial \mathcal{E}(\theta_k)}{\partial \theta}\right]^2,$$
$$\theta_{k+1} \leftarrow \theta_k - \frac{\alpha \cdot m_{k+1}}{\sqrt{s_{k+1}} + \epsilon}.$$

Adam uses both the momentum term (m) and the squared gradient term (s) in order to update the parameters. The algorithm itself is slightly different than what we have presented here. It has a bias-correction term.

`arxiv.org/abs/1412.6980`

Section 5

Stochasticity in Gradients and Stochastic Gradient Descent

From Gradient Descent to Stochastic Gradient Descent

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

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

- By linearity,

$$\frac{\partial\mathcal{E}}{\partial\theta} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial\mathcal{L}^{(i)}}{\partial\theta}.$$

- 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 $N \gg 1$ (think about millions of training examples)!

Stochastic Gradient Descent

- **Stochastic gradient descent (SGD)**: update the parameters based on the gradient for a single training example,
  1. Choose i uniformly at random
  2. $\theta \leftarrow \theta - \alpha \frac{\partial \mathcal{L}^i}{\partial \theta}$
- Cost of each SGD update is independent of N.
- SGD can make significant progress before even seeing all the data!

Stochastic Gradient Descent is Unbiased

- Mathematical justification: if you sample a training example uniformly at random, the stochastic gradient is an **unbiased estimate** of the batch gradient:

$$\mathbf{E}\left[\frac{\partial \mathcal{L}^i}{\partial \theta}\right] = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial \mathcal{L}^i}{\partial \theta} = \frac{\partial \mathcal{E}}{\partial \theta}.$$

- Problems:
  - Variance in this estimate may be high
  - If we only look at one training example at a time, we can't exploit efficient vectorized operations.

Stochastic Gradient Descent and Its Variance

- Compromise approach: compute the gradients on a randomly chosen medium-sized set of training examples $\mathcal{M} \subset \{1, \ldots, N\}$, called a **mini-batch**.
- Stochastic gradients computed on larger mini-batches have smaller variance.

$$\text{Var}\left[\frac{1}{|\mathcal{M}|}\sum_{i\in\mathcal{M}}\frac{\partial\mathcal{L}^{(i)}}{\partial\theta_j}\right] = \frac{1}{|\mathcal{M}|^2}\sum_{i\in\mathcal{M}}\text{Var}\left[\frac{\partial\mathcal{L}^{(i)}}{\partial\theta_j}\right] = \frac{1}{|\mathcal{M}|}\text{Var}\left[\frac{\partial\mathcal{L}^{(i)}}{\partial\theta_j}\right]$$

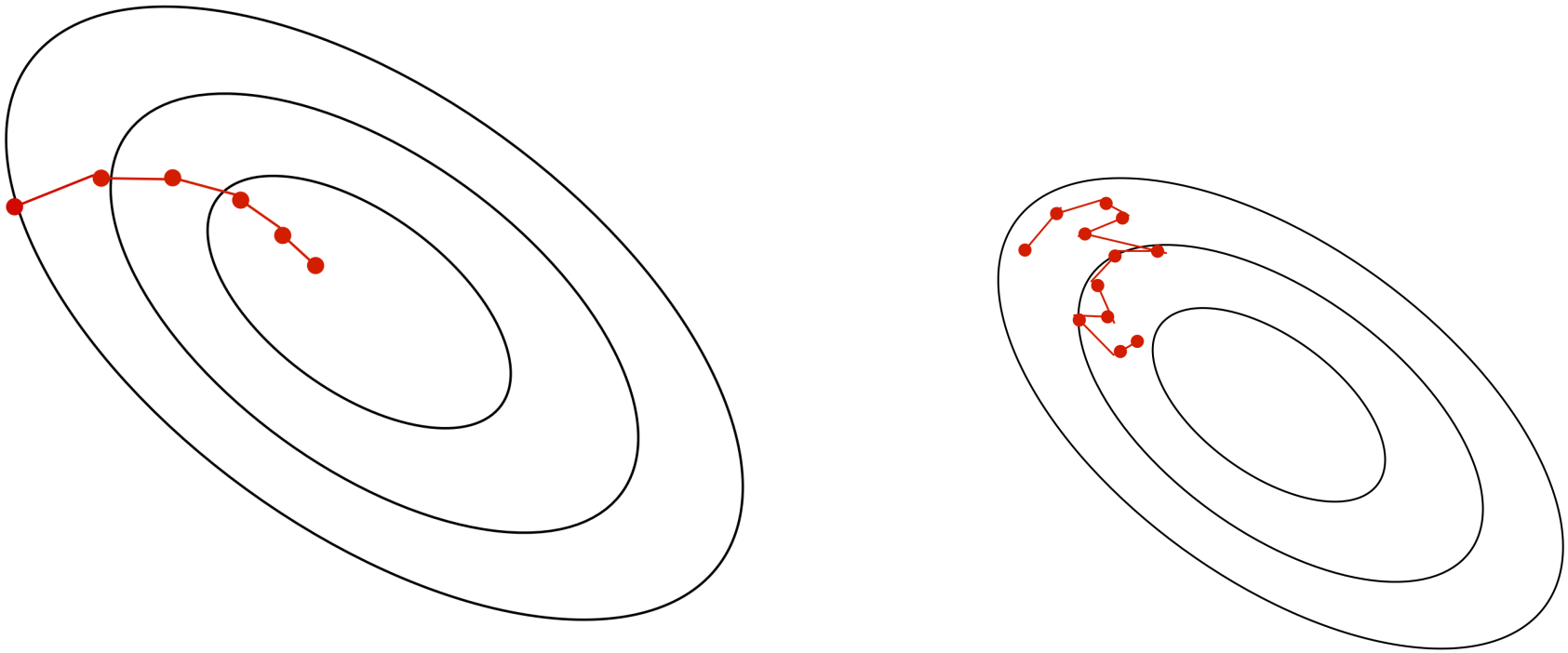
- Here we used the independence of data points in the first equality, and their having identical distribution in the second equality.

Stochastic Gradient Descent and Its Variance

- The mini-batch size $|\mathcal{M}|$ is a hyperparameter that needs to be set.
  - Too large: takes more computation, i.e. takes more memory to store the activations, and longer to compute each gradient update
  - Too small: can't exploit vectorization; has high variance
  - A reasonable value might be $|\mathcal{M}| = 100$.

Stochastic Gradient Descent

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



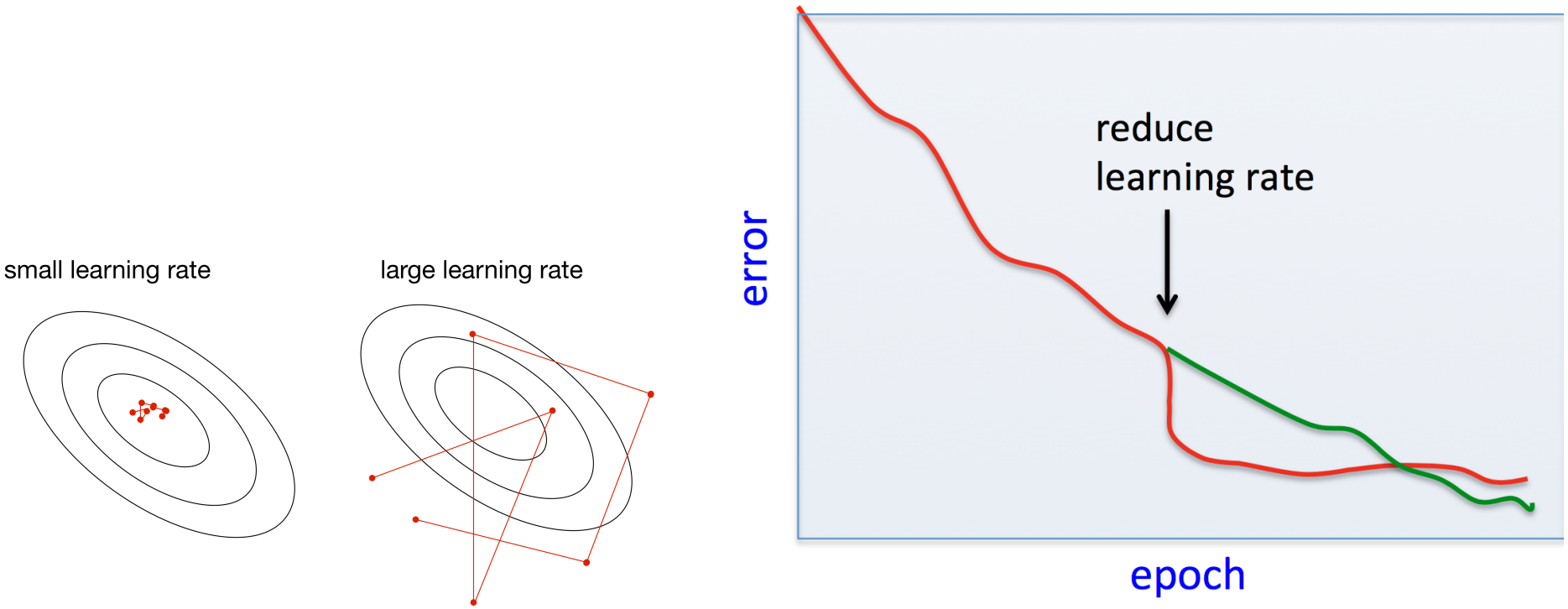
batch gradient descent **stochastic gradient descent**

SGD Learning Rate

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

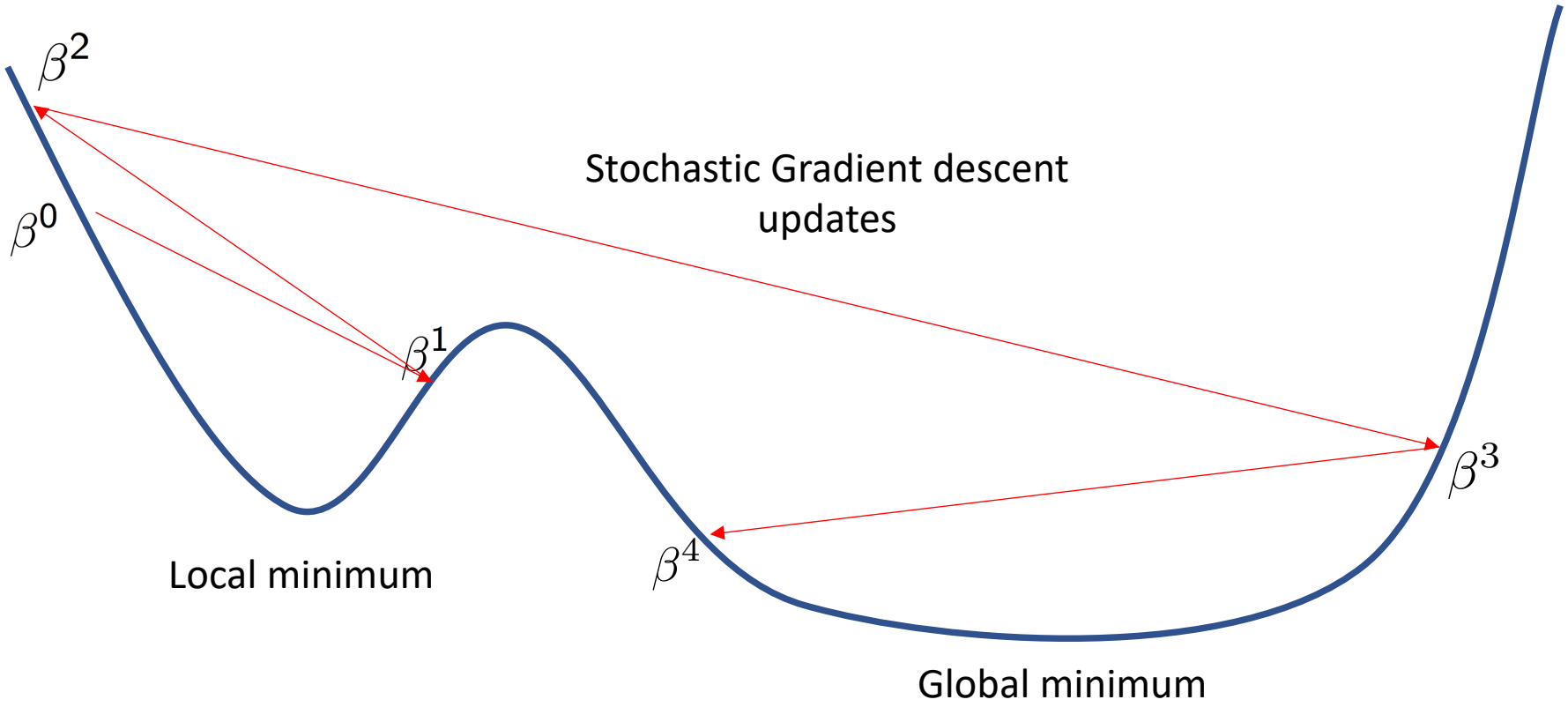
SGD Learning Rate

- 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.



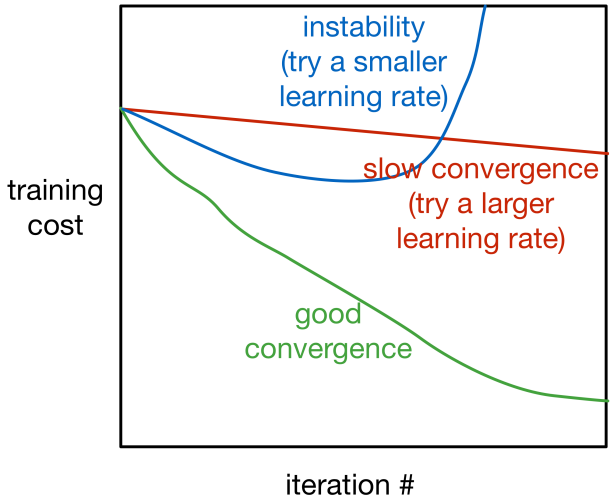
SGD and Non-convex optimization

- Stochastic methods have a chance of escaping from bad minima.
- Gradient descent with small step-size converges to first minimum it finds.



Training Curve (or Learning Curve)

To diagnose optimization problems, it's useful to look at **learning curves**: plot the training cost (or other metrics) as a function of iteration.



- **Note**: 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!
- **Note**: 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.

Visualizing Optimization Algorithms

You might want to check out these links

An overview of gradient descent algorithms
https://ruder.io/optimizing-gradient-descent

CS231n https://cs231n.github.io/neural-networks-3/

Why momentum really works https://distill.pub/2017/momentum/

Section 6

What to do this week?