How does gradient descent work?
The loss “landscape”

\[ \theta^* \leftarrow \arg \min_\theta \sum_i \log p_\theta(y_i|x_i) \]

let’s say \( \theta \) is 2D

An algorithm:

1. Find a direction \( \mathbf{v} \) where \( \mathcal{L}(\theta) \) decreases
2. \( \theta \leftarrow \theta + \alpha \mathbf{v} \)

some small constant called “learning rate” or “step size”
Gradient descent

An algorithm:
1. Find a direction $v$ where $\mathcal{L}(\theta)$ decreases
2. $\theta \leftarrow \theta + \alpha v$

Which way does $\mathcal{L}(\theta)$ decrease?

- Negative slope = go to the right
- Positive slope = go to the left

In general:
- For each dimension, go in the direction opposite the slope along that dimension

Gradient:

$$\nabla_\theta \mathcal{L}(\theta) = \begin{pmatrix}
\frac{d\mathcal{L}(\theta)}{d\theta_1} \\
\frac{d\mathcal{L}(\theta)}{d\theta_2} \\
\vdots \\
\frac{d\mathcal{L}(\theta)}{d\theta_n}
\end{pmatrix}$$

$$v_1 = -\frac{d\mathcal{L}(\theta)}{d\theta_1} \quad v_2 = -\frac{d\mathcal{L}(\theta)}{d\theta_2} \quad \text{etc.}$$
Visualizing gradient descent

level set contours for all $\theta$ values along a line $\mathcal{L}(\theta)$ takes on the same value

visualizations based on Gabriel Goh’s distill.pub article: https://distill.pub/2017/momentum/
Demo time!

visualizations based on Gabriel Goh’s distill.pub article: https://distill.pub/2017/momentum/
What’s going on?

we don’t always move toward the optimum!

the steepest direction is not always best!

more on this later…
The loss surface

Logistic regression:

\[ p_\theta(y = i|x) = \frac{\exp(x^T \theta_i)}{\sum_{j=1}^{m} \exp(x^T \theta_j)} \]

Negative likelihood loss for logistic regression is guaranteed to be **convex**

(this is **not** an obvious or trivial statement!)

Convexity:

A function is convex if a line segment between any two points lies entirely “above” the graph.

Convex functions are “nice” in the sense that simple algorithms like gradient descent have strong guarantees.

The **doesn’t** mean that gradient descent works well for all convex functions!
The loss surface...
...of a neural network

pretty hard to visualize, because neural networks have very large numbers of parameters

but let’s give it a try!

...though some networks are better!

the monster of the plateau
Oh no...

the dragon of local optima

(b) with skip connections
The geography of a loss landscape

- the local optimum
- the plateau
- the saddle point
Local optima

the most obvious issue with non-convex loss landscapes

one of the big reasons people used to worry about neural networks!

very scary in principle, since gradient descent could converge to a solution that is arbitrarily worse than the global optimum!

a bit surprisingly, this becomes less of an issue as the number of parameters increases!

for big networks, local optima exist, but tend to be not much worse than global optima

Choromanska, Henaff, Mathieu, Ben Arous, LeCun. The Loss Surface of Multilayer Networks.
Plateaus

Can’t just choose tiny learning rates to prevent oscillation!

Need learning rates to be large enough not to get stuck in a plateau

We’ll learn about momentum, which really helps with this
Saddle points

the gradient here is very small
it takes a long time to get out of saddle points

this seems like a very special structure,
does it really happen that often?

Yes! in fact, most critical points in neural net loss landscapes are saddle points
Saddle points

Critical points:

any point where $\nabla_\theta \mathcal{L}(\theta) = 0$

is it a maximum, minimum, or saddle?

In higher dimensions:

Hessian matrix:

$$
\begin{bmatrix}
\frac{d^2 \mathcal{L}}{d\theta_1 d\theta_1} & \frac{d^2 \mathcal{L}}{d\theta_1 d\theta_2} & \frac{d^2 \mathcal{L}}{d\theta_1 d\theta_3} \\
\frac{d^2 \mathcal{L}}{d\theta_2 d\theta_1} & \frac{d^2 \mathcal{L}}{d\theta_2 d\theta_2} & \frac{d^2 \mathcal{L}}{d\theta_2 d\theta_3} \\
\frac{d^2 \mathcal{L}}{d\theta_3 d\theta_1} & \frac{d^2 \mathcal{L}}{d\theta_3 d\theta_2} & \frac{d^2 \mathcal{L}}{d\theta_3 d\theta_3}
\end{bmatrix}
$$

only maximum or minimum if all diagonal entries are positive or negative!

how often is that the case?
Which way do we go?

we don’t always move toward the optimum!

the steepest direction is not always best!

more on this later…
Improvement directions
A better direction...

can we find this direction?
yes, with Newton’s method!
we won’t use Newton’s method (can’t afford it)
but it’s an “ideal” to aspire to
Newton’s method

Taylor expansion:

\[ f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(x_0)(x - x_0)^2 \]

multivariate case:

\[ \mathcal{L}(\theta) \approx \mathcal{L}(\theta_0) + \nabla_\theta \mathcal{L}(\theta_0)(\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)^T \nabla^2_\theta \mathcal{L}(\theta_0)(\theta - \theta_0) \]

Gradient:

\[ \begin{bmatrix} \frac{d^2 \mathcal{L}}{d\theta_1 d\theta_1} & \frac{d^2 \mathcal{L}}{d\theta_1 d\theta_2} & \frac{d^2 \mathcal{L}}{d\theta_1 d\theta_3} \\ \frac{d^2 \mathcal{L}}{d\theta_2 d\theta_1} & \frac{d^2 \mathcal{L}}{d\theta_2 d\theta_2} & \frac{d^2 \mathcal{L}}{d\theta_2 d\theta_3} \\ \frac{d^2 \mathcal{L}}{d\theta_3 d\theta_1} & \frac{d^2 \mathcal{L}}{d\theta_3 d\theta_2} & \frac{d^2 \mathcal{L}}{d\theta_3 d\theta_3} \end{bmatrix} \]

Hessian

can optimize this analytically!

set derivative to zero and solve:

\[ \theta^* \leftarrow \theta_0 - (\nabla^2_\theta \mathcal{L}(\theta_0))^{-1} \nabla_\theta \mathcal{L}(\theta_0) \]
Tractable acceleration

Why is Newton’s method not a viable way to improve neural network optimization?

gradient descent: \[ \theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta \mathcal{L}(\theta_k) \]

runtime? \[ \mathcal{O}(n) \]

Hessian

\[
\begin{bmatrix}
\frac{d^2 \mathcal{L}}{d\theta_1 d\theta_1} & \frac{d^2 \mathcal{L}}{d\theta_1 d\theta_2} & \cdots & \frac{d^2 \mathcal{L}}{d\theta_1 d\theta_n} \\
\frac{d^2 \mathcal{L}}{d\theta_2 d\theta_1} & \frac{d^2 \mathcal{L}}{d\theta_2 d\theta_2} & \cdots & \frac{d^2 \mathcal{L}}{d\theta_2 d\theta_n} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{d^2 \mathcal{L}}{d\theta_3 d\theta_1} & \frac{d^2 \mathcal{L}}{d\theta_3 d\theta_2} & \cdots & \frac{d^2 \mathcal{L}}{d\theta_3 d\theta_n}
\end{bmatrix}
\]

\[
\nabla_\theta \mathcal{L}(\theta) = \begin{pmatrix}
\frac{d\mathcal{L}(\theta)}{d\theta_1} \\
\frac{d\mathcal{L}(\theta)}{d\theta_2} \\
\vdots \\
\frac{d\mathcal{L}(\theta)}{d\theta_n}
\end{pmatrix}
\]

\[ \theta^* \leftarrow \theta_0 - (\nabla_\theta^2 \mathcal{L}(\theta_0))^{-1} \nabla_\theta \mathcal{L}(\theta_0) \]

runtime? \[ \mathcal{O}(n^3) \]

if using naïve approach, though fancy methods can be much faster if they avoid forming the Hessian explicitly

because of this, we would really prefer methods that don’t require second derivatives, but somehow “accelerate” gradient descent instead
Momentum

averaging together successive gradients seems to yield a much better direction!

Intuition: if successive gradient steps point in different directions, we should cancel off the directions that disagree

if successive gradient steps point in similar directions, we should go faster in that direction
Momentum

update rule:

$$\theta_{k+1} = \theta_k - \alpha g_k$$

before: $g_k = \nabla_{\theta} \mathcal{L}(\theta_k)$

now: $g_k = \nabla_{\theta} \mathcal{L}(\theta_k) + \mu g_{k-1}$

“blend in” previous direction

this is a very simple update rule

in practice, it brings some of the benefits of Newton’s method, at virtually no cost

does carry very appealing guarantees (in practice, we usually just momentum)

this kind of momentum method has few guarantees

a closely related idea is “Nesterov accelerated gradient,” which does carry very appealing guarantees
Momentum Demo

visualizations based on Gabriel Goh’s distill.pub article: https://distill.pub/2017/momentum/
Gradient scale

**Intuition:** the sign of the gradient tells us which way to go along each dimension, but the magnitude is not so great.

**Even worse:** overall magnitude of the gradient can change drastically over the course of optimization, making learning rates hard to tune.

**Idea:** “normalize” out the magnitude of the gradient along each dimension.
Algorithm: RMSProp

Estimate per-dimension magnitude (running average):

\[ s_k \leftarrow \beta s_{k-1} + (1 - \beta)(\nabla_{\theta} \mathcal{L}(\theta_k))^2 \]

this is roughly the squared length of each dimension

\[ \theta_{k+1} = \theta_k - \alpha \frac{\nabla_{\theta} \mathcal{L}(\theta_k)}{\sqrt{s_k}} \]

each dimension is divided by its magnitude
Algorithm: AdaGrad

Estimate per-dimension cumulative magnitude:

$$s_k \leftarrow s_{k-1} + (\nabla_\theta L(\theta_k))^2$$

$$\theta_{k+1} = \theta_k - \alpha \frac{\nabla_\theta L(\theta_k)}{\sqrt{s_k}}$$

RMSProp:

$$s_k \leftarrow \beta s_{k-1} + (1 - \beta)(\nabla_\theta L(\theta_k))^2$$

How does AdaGrad and RMSProp compare?

AdaGrad has some appealing guarantees for convex problems

Learning rate effectively “decreases” over time, which is good for convex problems

But this only works if we find the optimum quickly before the rate decays too much

RMSProp tends to be much better for deep learning (and most non-convex problems)
Algorithm: Adam

**Basic idea:** combine **momentum** and **RMSProp**

\[
m_k = (1 - \beta_1) \nabla_\theta \mathcal{L}(\theta_k) + \beta_1 m_{k-1}
\]

\[
v_k = (1 - \beta_2) (\nabla_\theta \mathcal{L}(\theta_k))^2 + \beta_2 v_{k-1}
\]

\[
\hat{m}_k = \frac{m_k}{1 - \beta_1^k}
\]

why? \(
m_0 = 0 \quad v_0 = 0
\)

\[
\hat{v}_k = \frac{v_k}{1 - \beta_2^k}
\]

\[
\theta_{k+1} = \theta_k - \alpha \frac{\hat{m}_k}{\sqrt{\hat{v}_k} + \epsilon}
\]

first moment estimate (“momentum-like”)

second moment estimate

so early on these values will be small, and this correction “blows them up” a bit for small \(k\)

good default settings:

\[
\alpha = 0.001 \\
\beta_1 = 0.9 \\
\beta_2 = 0.999
\]

small number to prevent division by zero

\[
\epsilon = 10^{-8}
\]
Stochastic optimization
Why is gradient descent expensive?

\[ \mathcal{L}(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \log p_\theta(y_i | x_i) \approx -E_{p_{\text{data}(x,y)}}[\log p_\theta(y_i | x_i)] \approx -\frac{1}{B} \sum_{j=1}^{B} \log p_\theta(y_{ij} | x_{ij}) \]

requires summing over all datapoints in the dataset could simply use fewer samples, and still have a correct (unbiased) estimator

ILSVRC (ImageNet), 2009: 1.5 million images
Stochastic gradient descent

with minibatches

1. Sample $B \subset D$

2. Estimate $g_k \leftarrow -\nabla_{\theta} \frac{1}{B} \sum_{i=1}^{B} \log p(y_i | x_i, \theta) \approx \nabla_{\theta} \mathcal{L}(\theta)$

3. $\theta_{k+1} \leftarrow \theta_k - \alpha g_k$

draw $B$ datapoints at random from dataset of size $N$

(where sum is over elements in $B$)

can also use momentum, ADAM, etc.

each iteration samples a different minibatch

Stochastic gradient descent in practice:

- sampling randomly is slow due to random memory access
- instead, shuffle the dataset (like a deck of cards...) once, in advance
- then just construct batches out of consecutive groups of $B$ datapoints
Learning rates

Low learning rates can result in convergence to worse values!
This is a bit counter-intuitive.
Decaying learning rates

AlexNet trained on ImageNet

Learning rate decay schedules usually needed for best performance with SGD (+momentum)

Often not needed with ADAM

Opinions differ, some people think SGD + momentum is better than ADAM if you want the very best performance (but ADAM is easier to tune)
Tuning (stochastic) gradient descent

Hyperparameters:

- **batch size:** $B$
  - larger batches = less noisy gradients, usually “safer” but more expensive
- **learning rate:** $\alpha$
  - best to use the biggest rate that still works, decay over time
- **momentum:** $\mu$
  - keep the defaults (usually)
  - 0.99 is good
- **Adam parameters:** $\beta_1$, $\beta_2$

What to tune hyperparameters on?

Technically we want to tune this on the **training** loss, since it is a parameter of the optimization

Often tuned on **validation** loss

Relationship between stochastic gradient and regularization is complex – some people consider it to be a good regularizer!

(this suggests we should use validation loss)