Data 188: Introduction to Deep Learning

Fully-connected networks, optimization, initialization

Speaker: Eric Kim
Lecture 06 (Week 03)
2026-02-05, Spring 2026. UC Berkeley.

Acknowledgement: slides are based on CMU's Deep Learning Systems course (10-414/714, Fall 2025). Thanks to: Prof. Zico Kolter, Prof. Tianqi Chen, Prof. Tim Dettmers.

Revision02: 2026-02-08

Announcements

- HW0 due tonight!
- HW1 is released
    - Warning: this homework is substantially more work than HW0. Start early!

https://phdcomics.com/comics/archive.php?comicid=820

How to succeed in this class

- Step 1: Attend ALL course activities. **Don't fall behind!**
  - An easy way to ensure success: **regularly attend lecture and discussion**. Everything else will follow naturally.
  - It'll be very difficult to succeed in this class without doing at least this. "Why play on hard mode?"

- Step 2: Actively study material
  - Ex: "passive" learning would be simply watching lecture and moving on.
  - "Active" learning: revisit lecture slides and discussion exercises. Recreate any tricky derivations, ask questions on Ed, go to office hours, use Google...
  - **Make course concepts "yours".** "Can you explain ConceptX to someone not taking the course?"

- Step 3: Reinforce concepts by doing homework assignments
  - In this course, homework assignments are sort of the "first class citizens". In a sense, lectures and discussions exist to support you for the homework.
  - These homework assignments are a great way to get a deep, hands-on understanding of the course concepts
  - Homework assignments may reveal holes in your knowledge. This is natural! Forge on ahead, it'll be worth it!

Some good advice that you may enjoy: ["Doing well in your courses: a guide by Andrej Karpathy"]().

Outline

Fully connected networks

Optimization

Initialization

Outline

Fully connected networks

Optimization

Initialization

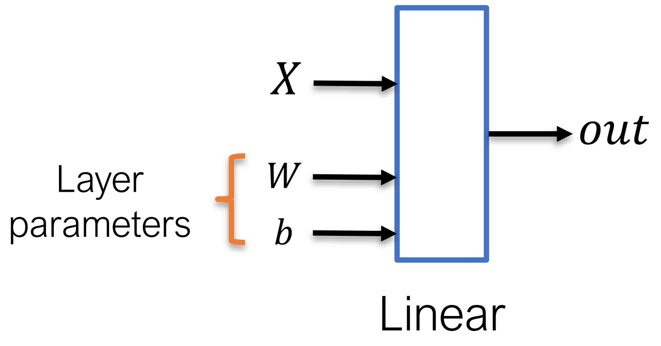
Linear layer

In our MNIST digit classifier (one, two-layer), the primary operation is a matrix multiply (eg ops.MatMul):

n: input dimensionality
k: number of output classes
d: hidden dimension size

One layer classifier	$h(X) = X\theta$	X has shape=[batchsize, n], θ has shape=[n, k]
Two layer classifier	$h(X) = \sigma(XW_1)W_2$	W_1 has shape=[n, d], W_2 has shape=[d, k]

In deep learning, it's common to augment MatMul with a learnable **bias** term. This is commonly called a "Linear" (or FullyConnected) layer:



$$\text{linear}(X) = XW + b$$

X shape=[batchsize, d_in]
W shape=[d_in, d_out]
b shape=[1, d_out]

https://docs.pytorch.org/docs/stable/generated/torch.nn.Linear.html

Linear layer: illustrated

Linear(X) takes an input with dimensionality `d_in` and transforms it to dimensionality `d_out`

$$\text{linear}(X) = XW + b$$

X shape=[batchsize, d_in]
W shape=[n, d_out]
b shape=[1, d_out]

$$W = \begin{bmatrix} 2 & 2 & 0 & 3 \\ 0 & 1 & 1 & 5 \\ 1 & 4 & 2 & 0 \end{bmatrix}$$

$$b = \begin{bmatrix} 5 & -5 & 0 & 1 \end{bmatrix}$$

$$\text{linear}\left(\begin{bmatrix} 1 & 2 & 1 \\ 3 & 4 & 0 \end{bmatrix}\right) \rightarrow \begin{bmatrix} 8 & 3 & 4 & 14 \\ 11 & 5 & 4 & 30 \end{bmatrix}$$

$$\text{linear}(X) = XW + \mathbf{1}b$$

MatMul $$\begin{bmatrix} 1 & 2 & 1 \\ 3 & 4 & 0 \end{bmatrix}\begin{bmatrix} 2 & 2 & 0 & 3 \\ 0 & 1 & 1 & 5 \\ 1 & 4 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 3 & 8 & 4 & 13 \\ 6 & 10 & 4 & 29 \end{bmatrix}$$

+ bias $$\begin{bmatrix} 3 & 8 & 4 & 13 \\ 6 & 10 & 4 & 29 \end{bmatrix} + \begin{bmatrix} 5 & -5 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 8 & 4 & 13 \\ 6 & 10 & 4 & 29 \end{bmatrix} + \begin{bmatrix} 5 & -5 & 0 & 1 \\ 5 & -5 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 8 & 3 & 4 & 14 \\ 11 & 5 & 4 & 30 \end{bmatrix}$$

Broadcast sum

Bias broadcasting

Linear(X) takes an input with dimensionality `d_in` and transforms it to dimensionality `d_out`

$$\mathrm{linear}(X) = XW + b$$

X shape=[batchsize, d_in]
W shape=[n, d_out]
b shape=[1, d_out]

+ bias $\begin{bmatrix} 3 & 8 & 4 & 13 \\ 6 & 10 & 4 & 29 \end{bmatrix} + \begin{bmatrix} 5 & -5 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 8 & 4 & 13 \\ 6 & 10 & 4 & 29 \end{bmatrix} + \begin{bmatrix} 5 & -5 & 0 & 1 \\ 5 & -5 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 8 & 3 & 4 & 14 \\ 11 & 5 & 4 & 30 \end{bmatrix}$

Broadcast sum

Tip: ndarray libraries like numpy and pytorch will not actually copy the bias vector like this. As an optimization, they implement broadcast semantics that achieves this behavior without having to explicitly copy the bias vector.

If you wanted to write linear(X) more formally:

$$\mathrm{linear}(X) = XW + \mathbf{1}b$$

Math-y way of saying: "create `batchsize` copies of b, row-wise"

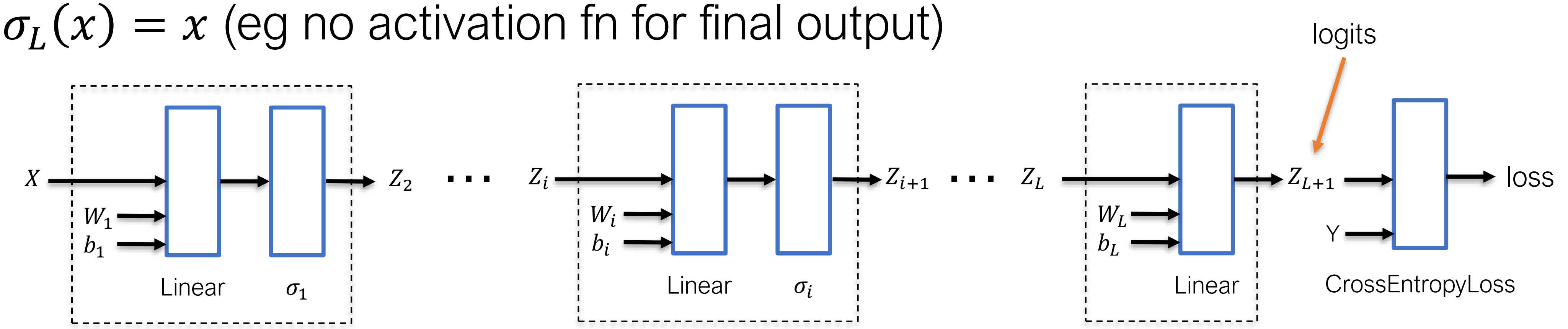
Where $\mathbf{1}$ has shape=[batchsize, 1], b has shape=[1, d_out]

Fully connected networks

An L*-layer, fully connected network,* a.k.a. multi-layer perceptron (MLP), with a bias term, is defined by the iteration

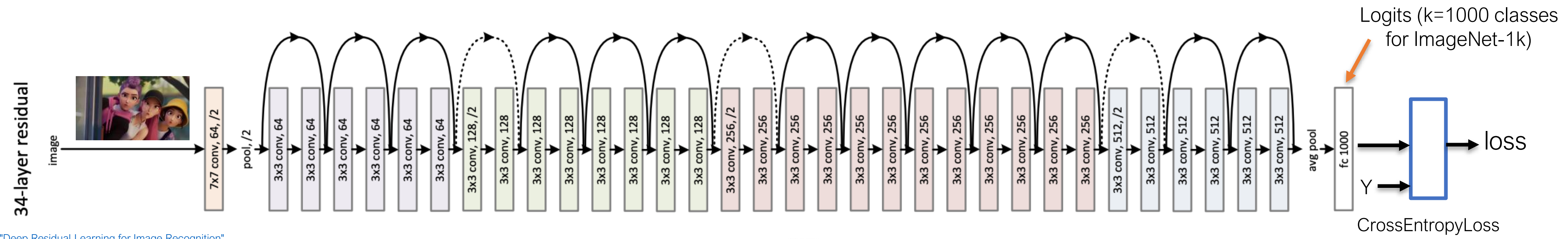
$$Z_{i+1} = \sigma_i(Linear_i(X)), \qquad i = 1, \dots, L$$
$$h_\theta(X) \equiv Z_{L+1}$$
$$Z_1 \equiv X$$
$$Linear_i(X) = XW_i + b_i$$

with model parameters $\theta = \{W_{1:L}, b_{1:L}\}$. $\sigma_i(x)$ is the (nonlinear!) activation, usually with $\sigma_L(x) = x$ (eg no activation fn for final output)

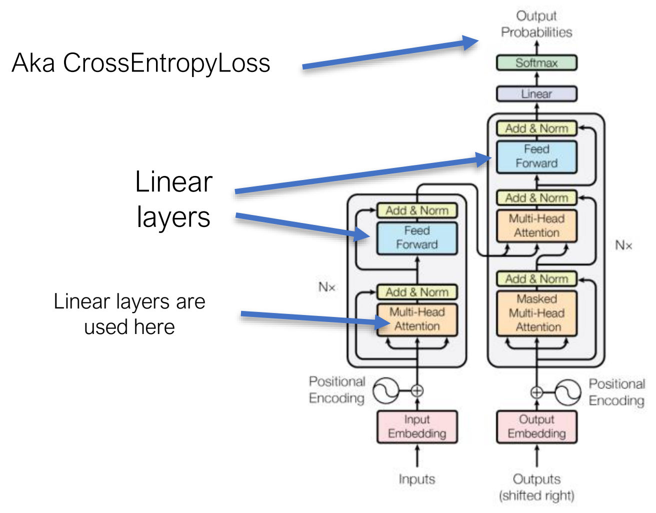


Deep learning: modularity

The Linear and CrossEntropyLoss layers are ubiquitous layers in deep learning, including: image/text classification, large language models, etc.



Resnet paper, "Deep Residual Learning for Image Recognition".



Transformers paper: https://arxiv.org/pdf/1706.03762

Figure 1: The Transformer - model architecture.

Deep learning: aka building models with modular "lego blocks".

Common layers: Linear, CrossEntropy, EWiseAdd/Mul, Conv2D, ...)

The "art" of deep learning

In order to actually train a fully-connected network (or any deep network), we need to address a certain number of questions:

- How do we choose the width and depth of the network?
- How do we actually optimize the objective? ("SGD" is the easy answer, but not the algorithm most commonly used in practice)
- How do we initialize the weights of the network?
- How do we ensure the network can continue to be trained easily over multiple optimization iterations?

All related questions that affect each other

There are (still) no definite answers to these questions, and for deep learning they wind up being problem-specific, but we will cover some basic principles

Outline

Fully connected networks

Optimization

Initialization

Optimization

Definition: an **objective function** is a scalar-valued function $f: \mathbb{R}^n \to \mathbb{R}$ that we would like to minimize.

Definition: an optimization problem consists of an objective function and optional constraints. We express it as:

$$\min_{\theta} f(\theta)$$

"Find the parameters θ that minimize $f(\theta)$ (aka "argmin")

Subject to:

$$g_i(\theta) \leq 0, i = 1, \dots, m$$
$$h_j(\theta) = 0, j = 1, \dots, p$$

$\leftarrow$ Fortunately, in this class (and deep learning in general) we typically don't have constraints, so we can drop these

Example: let $f(\theta, \mathcal{D}_{train}) = loss_{ce}(h(\theta), \mathcal{D}_{train})$. Minimizing $f(\theta, \mathcal{D}_{train})$ is precisely what we do when we train an ML model!

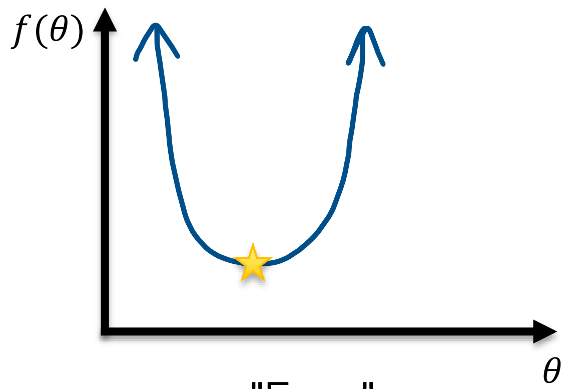
Objective function: difficulty

What kinds of objective functions are "easy" to optimize? What makes a function "hard" to optimize?

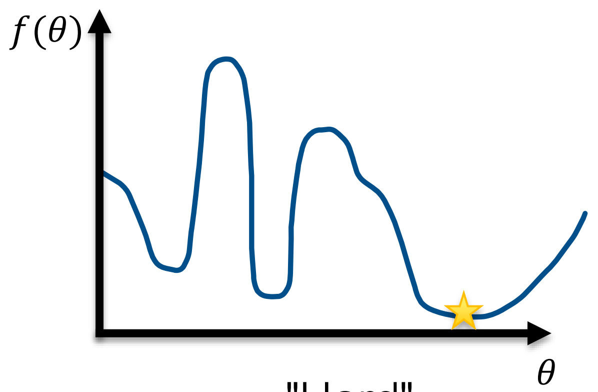
$$\min_{\theta} f(\theta)$$

Subject to: $g_i(\theta) \leq 0, i = 1, \ldots, m$

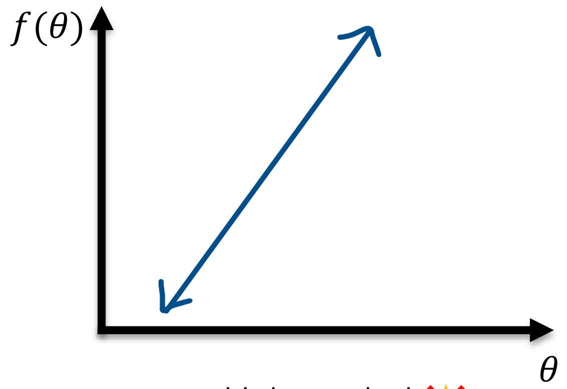
$h_j(\theta) = 0, j = 1, \ldots, p$



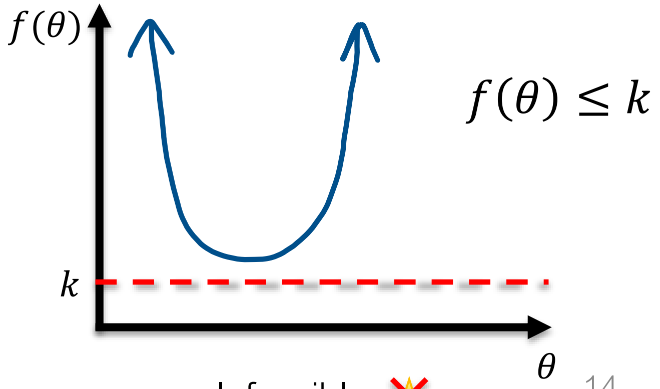
"Easy"



"Hard"



Unbounded



Infeasible

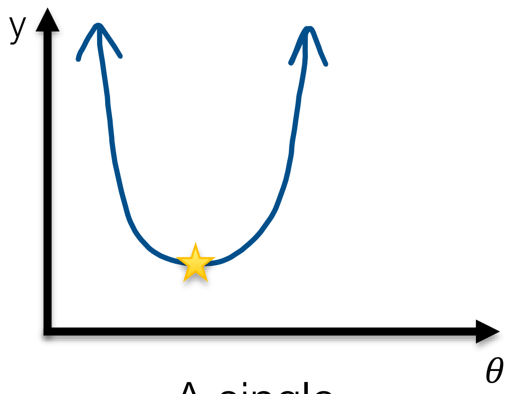
Global vs local optima

If you find a minima (say, through gradient descent), is it a global minima ("best")?

$$\min_{\theta} f(\theta)$$

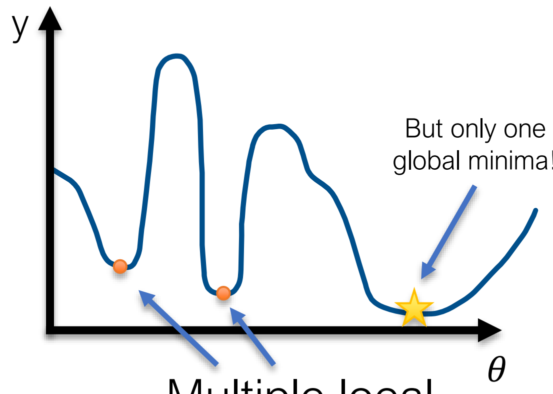
Subject to: $g_i(\theta) \leq 0, i = 1, \dots, m$

$h_j(\theta) = 0, j = 1, \dots, p$



A single minima

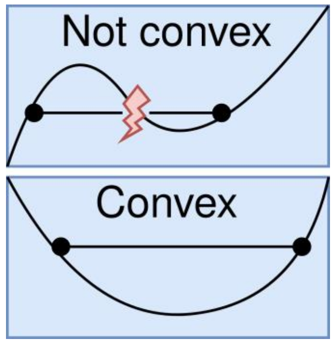
aka "any local optimum is a global optimum"



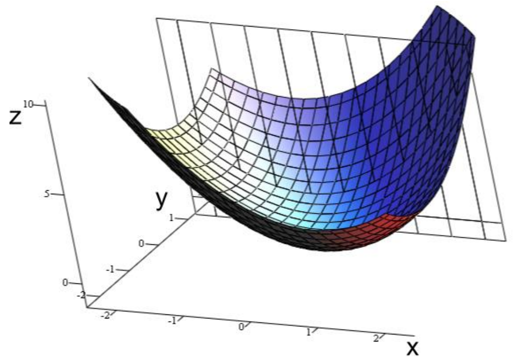
Multiple local minima.

Convex functions

In optimization, there are several "classes" of functions that are easy to optimize. One popular class of functions is called "convex functions".



By Varagk - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=124668501



By Indeed123 - commons This diagram was created with Mathematica., CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=5508870

Many interesting, important problems can be formulated as minimizing a convex optimization problem!

One lovely property of convex objective functions: if a convex function has a local optima, it is also a global optima*!

→

Gradient descent on convex objective functions will lead you to a global optima**!

UCLA's EE236B - Convex Optimization (Professor Lieven Vandenberghe) is an excellent graduate-level course on optimization. The textbook, "Convex Optimization", is also excellent!

*there may exist "saddle" points, see this post.
**assuming you use "appropriate" step size.

Objective functions in deep learning

Unfortunately, the objective functions used in deep learning models are (very much) not convex.

Thus, many of the insights/techniques from convex optimization simply don't directly translate over.

Deep learning objective functions have many (many) local minima. The loss landscape is hard (impossible) to interpret, with very little guarantees.

Ex: gradient descent is only guaranteed to give you a **local** minima. No **global** minima guarantees.

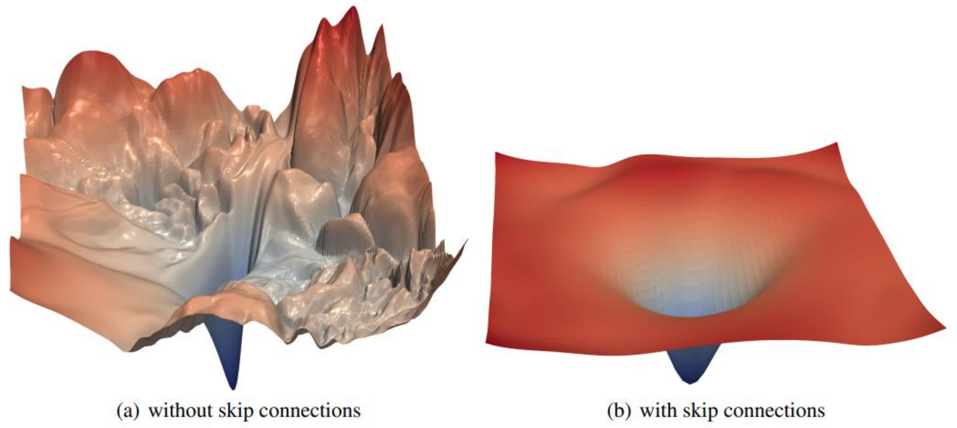


Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

"Visualizing the Loss Landscape of Neural Nets", https://arxiv.org/abs/1712.09913

Yet! Deep learning is still wildly successful despite this!

Optimization: closing thoughts

Optimization is a rich field with tons of interesting theory and applications.

$$\min_{\theta} f(\theta)$$

Subject to: $g_i(\theta) \leq 0, i = 1, \ldots, m$
$h_j(\theta) = 0, j = 1, \ldots, p$

In this course: we won't really study optimization that deeply. Mostly we'll study and apply gradient descent to train our models.

Gradient descent

In optimization, gradient descent is a popular and successful way to optimize functions. For a function f and iterate number t, the gradient descent update equation is:

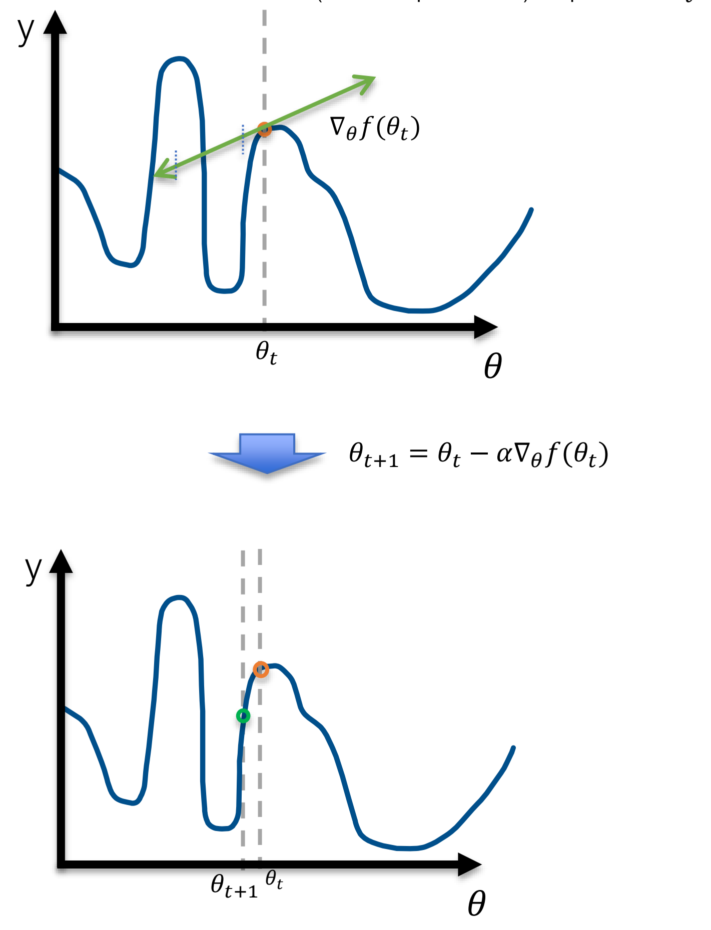
$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta f(\theta_t)$$

where $\alpha > 0$ is step size (learning rate), $\nabla_\theta f(\theta_t)$ is gradient evaluated at the parameters θ_t

Intuition: takes the “steepest descent direction” locally

Gradient $\nabla_\theta f(\theta_t)$ only tells you the **direction** to travel. Step size α dictates how **"far"** to go in that direction.

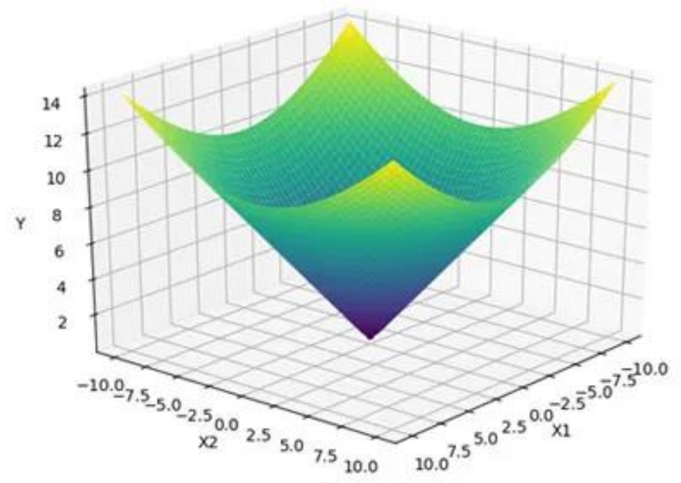
$\nabla_\theta f(\theta_t)$ means: gradient of f (with respect to θ) at position θ_t


y
$\nabla_\theta f(\theta_t)$
θ_t
θ
$\theta_{t+1} = \theta_t - \alpha \nabla_\theta f(\theta_t)$
y
θ_{t+1}
θ_t
θ

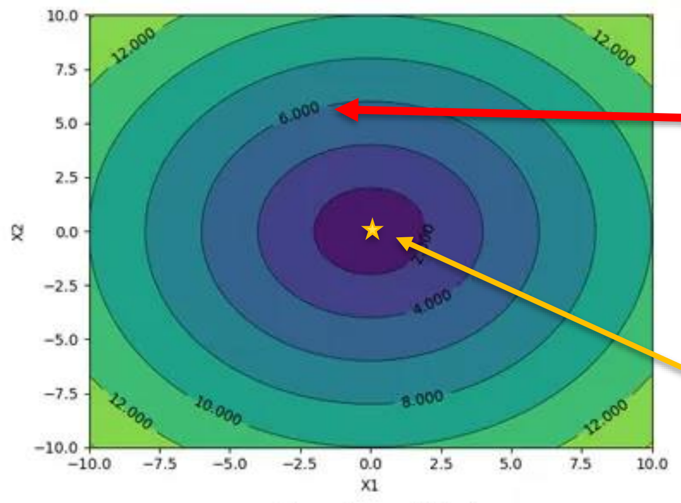
Contour plots

In optimization, contour plots are a way to visualize the loss landscape with respect to the optimization parameters.

Suppose our loss function is y = f(x1, x2), where x1, x2, y are scalars



3D Plot



Contour Plot

This circle (eg contour, or "level set") means: "these values of x1, x2 lead to a loss value of 6"

Global optimum

X1: param 1
X2: param 2
Y: loss value

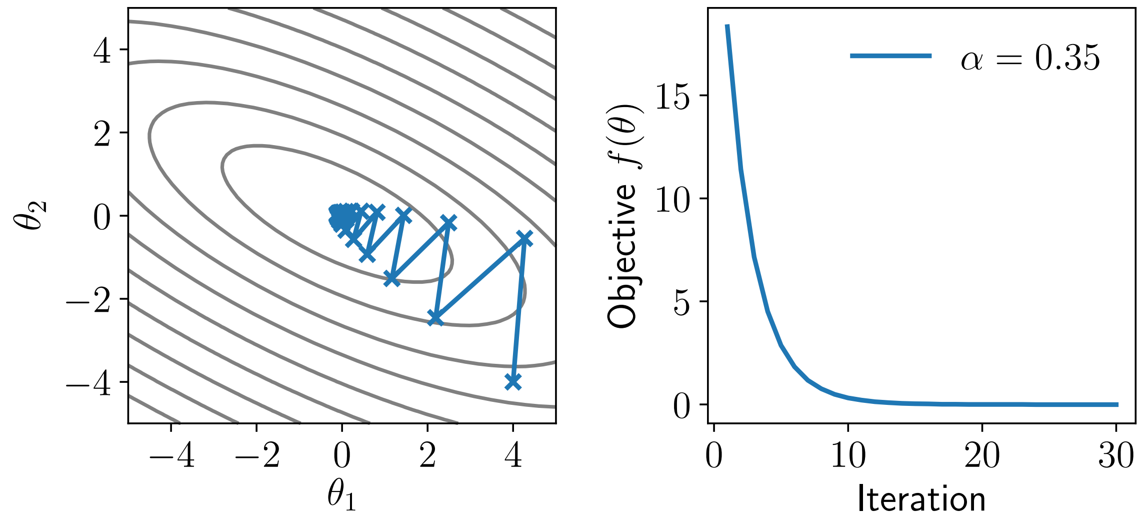
https://adeveloperdiary.com/data-science/how-to-visualize-gradient-descent-using-contour-plot-in-python/

Illustration of gradient descent

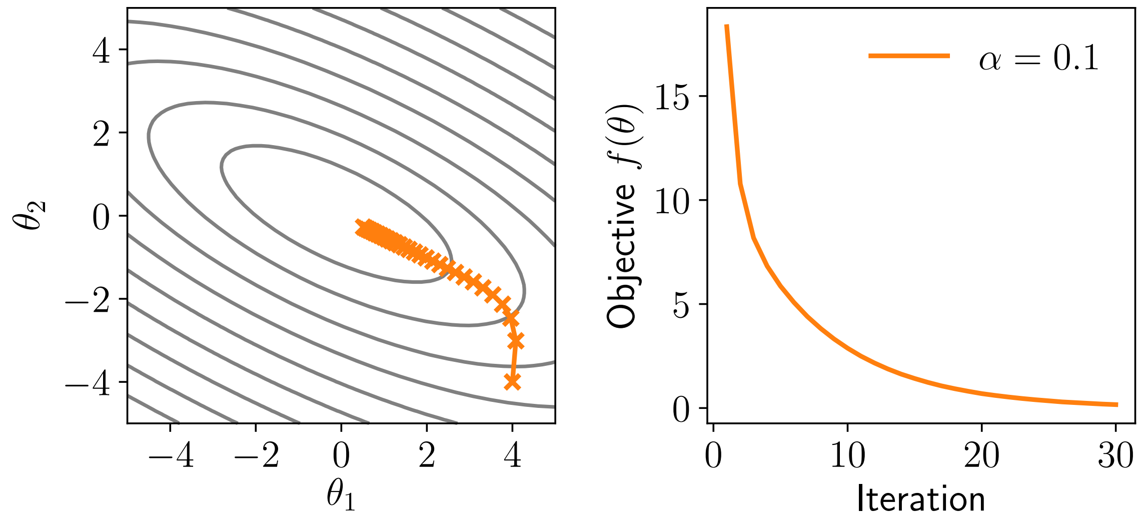
(Data188) don't worry too much about this equation. If you're curious, this is the multidimensional generalization of the scalar quadratic function.

For $\theta \in \mathbb{R}^2$, consider quadratic function $f(\theta) = \frac{1}{2}\theta^T P\theta + q^T\theta$, for P positive definite (all positive eigenvalues)

Illustration of gradient descent with different step sizes:



Updates are a little erratic, but still reaches the minimum



Smoother update trajectory. A "better" learning rate

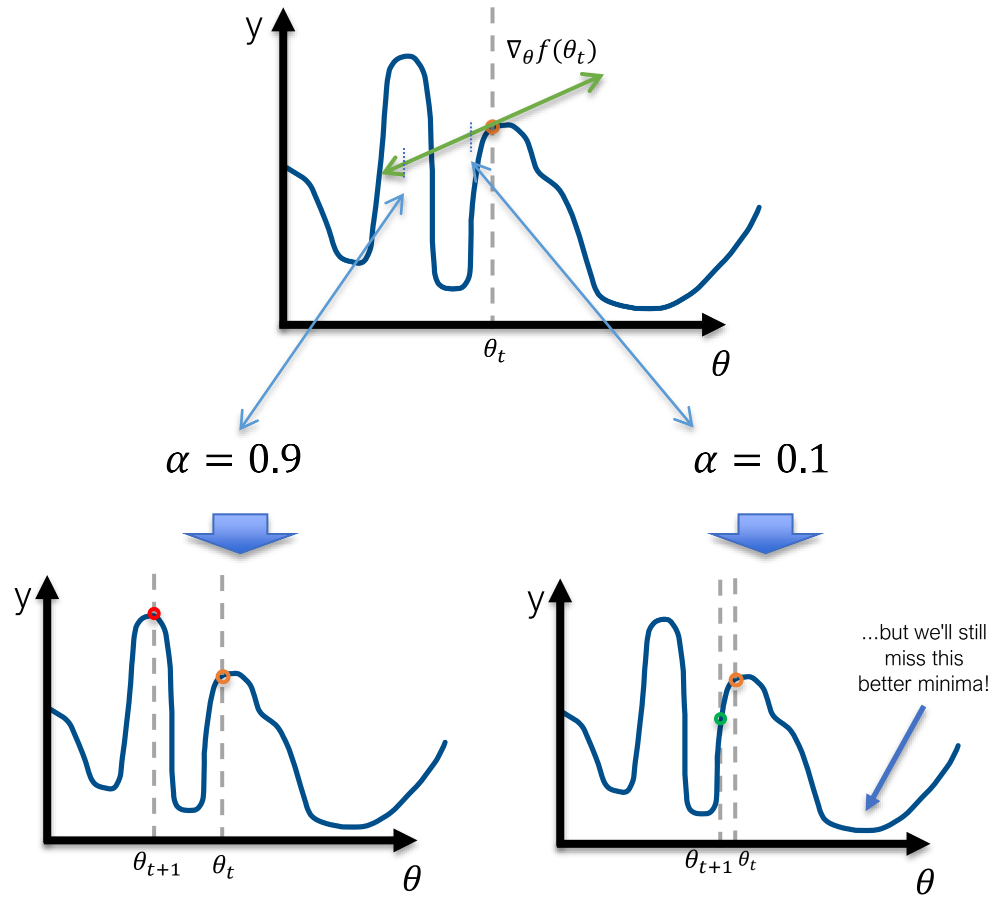
Gradient descent: local linear approximation

- **Gradient descent assumption**: approximate objective function via a locally-linear function.
    - Gradient $\nabla_\theta f(\theta_t)$ is tangent line of f at location θ_t. Tells us the direction that objective function decreases (within local linear window)

- **Pro**: fast, easy to compute!

- **Con**: locally-linear assumption can be (wildly!) inaccurate

- Sensitive to step size
    - Large step size -> "trust" local linear approx.
    - Smaller step size -> don't trust it too much



Step size α too large: our loss actually increased after param update (oh no!)

Better step size α : our loss went down, and we're on our way to a decent local minima

First order vs second order methods

- Gradient descent is known as a "first order method", because it only uses local gradient information (aka "first order derivatives of the objective function")

- There exist a class of more sophisticated methods that utilize additional information (local curvature) to try to create a better (more accurate) local approximation of the loss function
    - These are called "second order methods"
    - The "additional curvature information" is: second-derivatives!

The Hessian

Definition: for a function $f\colon \mathbb{R}^n \to \mathbb{R}$ (aka a scalar-valued function), the **Hessian matrix** H (sometimes denoted ∇^2) is the $n \times n$ matrix of all second derivatives.

$$H = \nabla^2 f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 x_n} \\ \frac{\partial^2 f}{\partial x_2 x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n x_1} & \frac{\partial^2 f}{\partial x_n x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Shape=[n, n]

Intuition: Hessian describes local curvature of the function.

Vs **Jacobian** (aka the "gradient", as f is scalar-valued) gives you a linear approximation of the function at a point x (eg tangent line/plane/hyperplane for $\mathbb{R}^2, \mathbb{R}^3, \mathbb{R}^{n>3}$ respectively).

$$J = \nabla f(x) = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} & \frac{\partial f(x)}{\partial x_2} & \cdots & \frac{\partial f(x)}{\partial x_n} \end{bmatrix}$$

Shape=[1, n]

Newton's Method

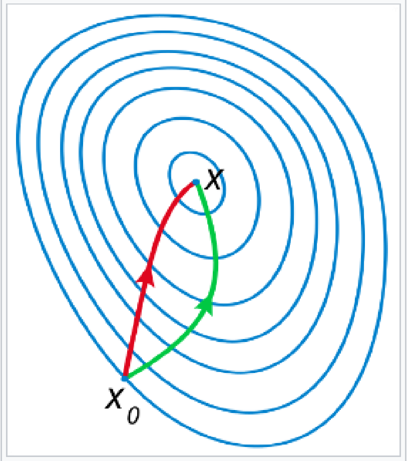
One way to integrate more "global" structure into optimization methods is Newton's method, which scales gradient according to inverse of the Hessian (matrix of second derivatives)

$$\theta_{t+1} = \theta_t - \alpha \left(\nabla_\theta^2 f(\theta_t)\right)^{-1} \nabla_\theta f(\theta_t)$$

where $\nabla_\theta^2 f(\theta_t)$ is the *Hessian*, $n \times n$ matrix of all second derivatives

Equivalent to approximating the function as quadratic using second-order Taylor expansion, then solving for optimal solution

Full step given by $\alpha = 1$, otherwise called a *damped* Newton method



A comparison of gradient descent (green) and Newton's method (red) for minimizing a function (with small step sizes). Newton's method uses curvature information (i.e. the second derivative) to take a more direct route.

By Oleg Alexandrov - self-made with en:Matlab. Tweaked in en:Inkscape, Public Domain, https://commons.wikimedia.org/w/index.php?curid=2284243

Recall: second-order Taylor expansion of f(x) around a is:

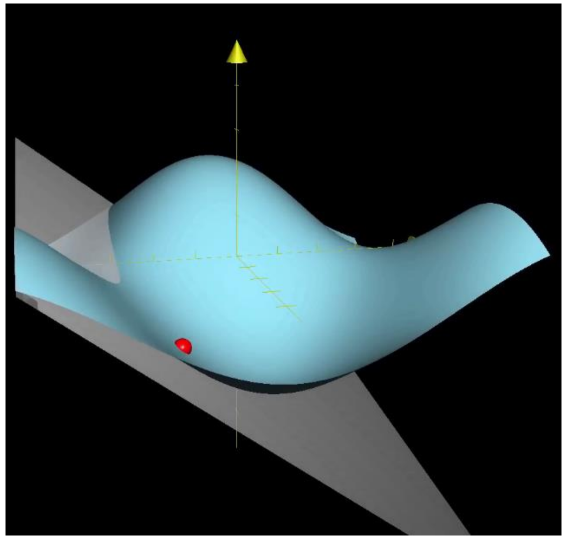
$$f(x) \approx f(a) + \nabla_\theta f(\theta_t)^T (x-a) + \frac{1}{2}(x-a)^T H_f(a)(x-a)$$

Where is the Hessian of f at point a: $H_f(a) = \nabla_\theta^2 f(\theta_t)$

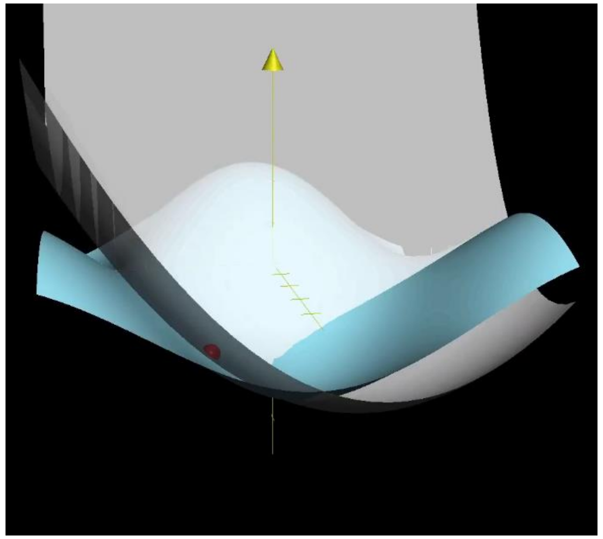
Intuition: approximate f(x) via a quadratic, with a linear offset (first order term) and a constant offset.

(optional) to learn more, see:
https://en.wikipedia.org/wiki/Newton%27s_method_in_optimization

Linear vs Quadratic local approximations



Linear approximation
(in 3D, a 2D surface
or tangent plane)



Quadratic
approximation

Here is a neat visualization of linear vs quadratic local approximations: "What do quadratic approximations look like?"

(optional) If you're interested in learning more about linear/quadratic approximations and its application to optimization (eg Newton's method), see this "Quadratic Approximation" Khan Academy lesson.
(this is out-of-scope for this class though)

Illustration of Newton's method

Newton's method (will $\alpha = 1$) will optimize quadratic functions in one step

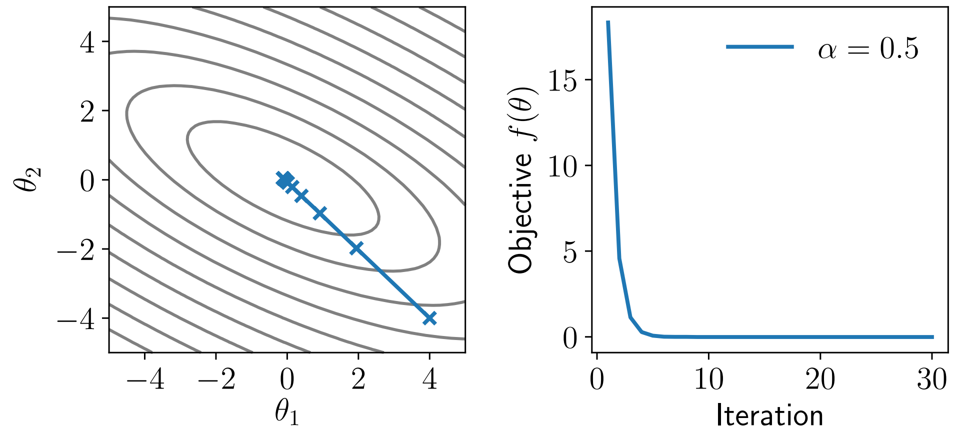
Not of that much practical relevance to deep learning for two reasons

1. We can't efficiently solve for Newton step, even using automatic differentiation (though there are tricks to approximately solve it)
2. For non-convex optimization, it's very unclear that we even *want* to use the Newton direction

Ex: a second-order approximation can also be wildly inaccurate!

In 2026: models with **hundreds of millions of parameters** are easily accessible.
Billion parameter models are easily productionized at scale at companies. (Ex: Llama3-8B has 8B params)
Trillion parameter models exist (albeit expensive to serve).

For a 1B param model: storing the Hessian matrix (1B x 1B) requires **4,000,000,000 gigabytes**. Not practical*!



***Disclaimer**: I'm not fully familiar with modern second order methods. There exist sophisticated second order methods that don't have to explicitly construct the Hessian, but these approaches are outside the scope for this course

Back to first order methods

In deep learning (and this course!), we'll primarily stick to first order methods like gradient descent, with the following update rule:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta f(\theta_t)$$

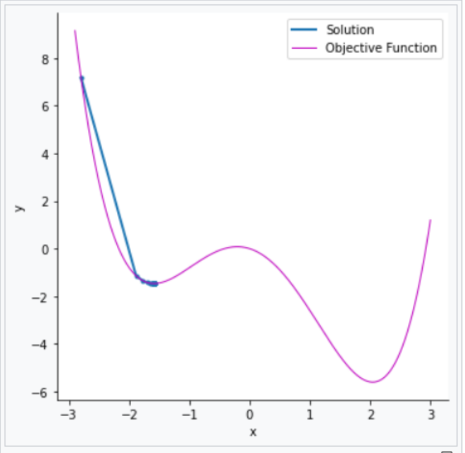
The above equation is known as "vanilla" gradient descent. Next, let's explore some common improvements to vanilla gradient descent!

Gradient descent deficiencies

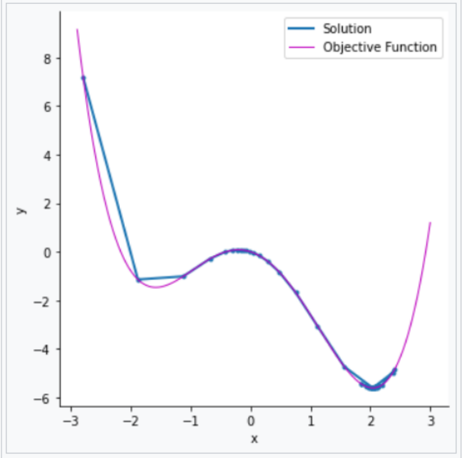
- One problem with "vanilla" gradient descent: noisy / oscillating gradients
    - **Possible root cause**: batchsize too small, dataset not representative enough

- Another issue: getting stuck in local minima

- **Idea**: maintain a history of gradients. Perform gradient update using a weighted average of current gradient $\nabla_\theta f(\theta_t)$ and previous gradients $[\nabla_\theta f(\theta_{t-1}), \nabla_\theta f(\theta_{t-2}), \ldots]$



Stochastic gradient descent without momentum stops at a local minimum.



Stochastic gradient descent with momentum stops at the global minimum.

https://optimization.cbe.cornell.edu/index.php?title=Momentum

Momentum

This is known as the *momentum* update, that takes into account a moving average of *multiple* previous gradients

$$u_{t+1} = \beta u_t + (1 - \beta)\nabla_\theta f(\theta_t)$$
$$\theta_{t+1} = \theta_t - \alpha u_{t+1}$$

β is typically between 0.8 and 0.99. 0.9 is a decent starting value.

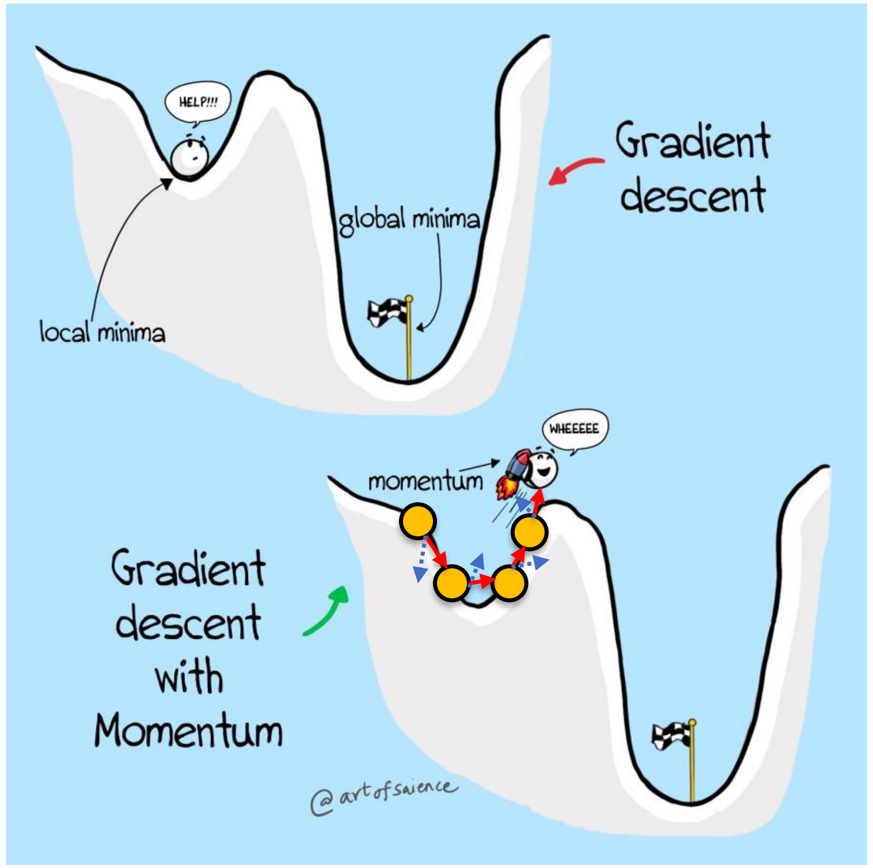
where $\alpha \in \mathbb{R}$ is step size as before, and $\beta \in \mathbb{R}$ is momentum averaging parameter

- Note: often written in alternative forms $u_{t+1} = \beta u_t + \nabla_\theta f(\theta_t)$ (or $u_{t+1} = \beta u_t + \alpha \nabla_\theta f(\theta_t)$) but I prefer above to keep u the same "scale" as gradient

In pytorch, `torch.optim.SGD` implements this via the `momentum=0, dampening=0` kwargs. Pytorch expresses momentum update slightly differently than we do, but the idea is the same

Momentum intuition: "rolling ball"

https://x.com/DSaience/status/1585239860566179840/photo/1



Momentum update:

$$u_{t+1} = \beta u_t + (1-\beta)\nabla_\theta f(\theta_t)$$
$$\theta_{t+1} = \theta_t - \alpha u_{t+1}$$

where $\alpha \in \mathbb{R}$ is step size as before, and $\beta \in \mathbb{R}$ is momentum averaging parameter

β is typically between 0.8 and 0.99. 0.9 is a decent starting value.

Visual: ball has inertia ("momentum"), some resistance to change to current gradient $\nabla_\theta f(\theta_t)$

Intuition: individual gradients $\nabla_\theta f(\theta_t)$ may be noisy/unreliable. But, a weighted average of gradients is (hopefully) more reliable!

$\dashrightarrow$ $\nabla_\theta f(\theta_t)$ Aka current ("noisy") gradient

$\rightarrow$ $u_{t+1} = \beta u_t + (1-\beta)\nabla_\theta f(\theta_t)$
Aka actual gradient used

Momentum: gradient history

Momentum update:

$$u_{t+1} = \beta u_t + (1-\beta)\nabla_\theta f(\theta_t)$$
$$\theta_{t+1} = \theta_t - \alpha u_{t+1}$$

where $\alpha \in \mathbb{R}$ is step size as before, and $\beta \in \mathbb{R}$ is momentum averaging parameter

$$\begin{aligned}
u_0 &= 0 \\
u_1 &= \beta u_0 + (1-\beta)\nabla_\theta f(\theta_0) = (1-\beta)\nabla_\theta f(\theta_0) \\
u_2 &= \beta u_1 + (1-\beta)\nabla_\theta f(\theta_1) = \beta\big((1-\beta)\nabla_\theta f(\theta_0)\big) + (1-\beta)\nabla_\theta f(\theta_1) \\
u_3 &= \beta u_2 + (1-\beta)\nabla_\theta f(\theta_2) = \beta\Big(\beta\big((1-\beta)\nabla_\theta f(\theta_0)\big) + (1-\beta)\nabla_\theta f(\theta_1)\Big) + (1-\beta)\nabla_\theta f(\theta_2) \\
&= \beta^2(1-\beta)\nabla_\theta f(\theta_0) + \beta(1-\beta)\nabla_\theta f(\theta_1) + (1-\beta)\nabla_\theta f(\theta_2)
\end{aligned}$$

$\ldots$

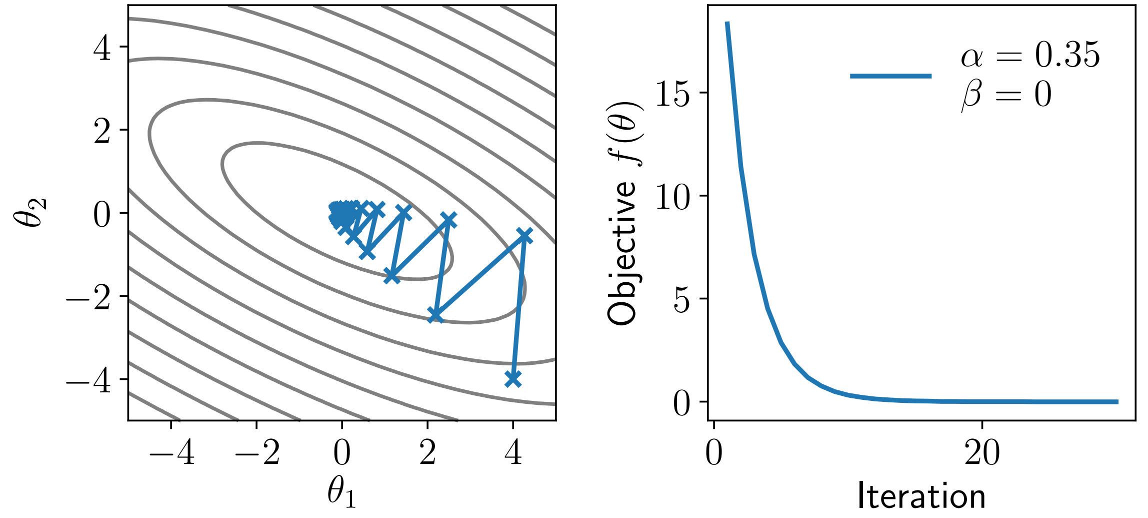
$$u_n = \underbrace{\beta^{n-1}(1-\beta)\nabla_\theta f(\theta_0) + \beta^{n-2}(1-\beta)\nabla_\theta f(\theta_1) + \cdots}_{\text{Exponential decay. Exponential "forgetting" of previous gradients, where memory strength is controlled by } \beta.} + \underbrace{(1-\beta)\nabla_\theta f(\theta_{n-1})}_{\text{Current gradient}}$$

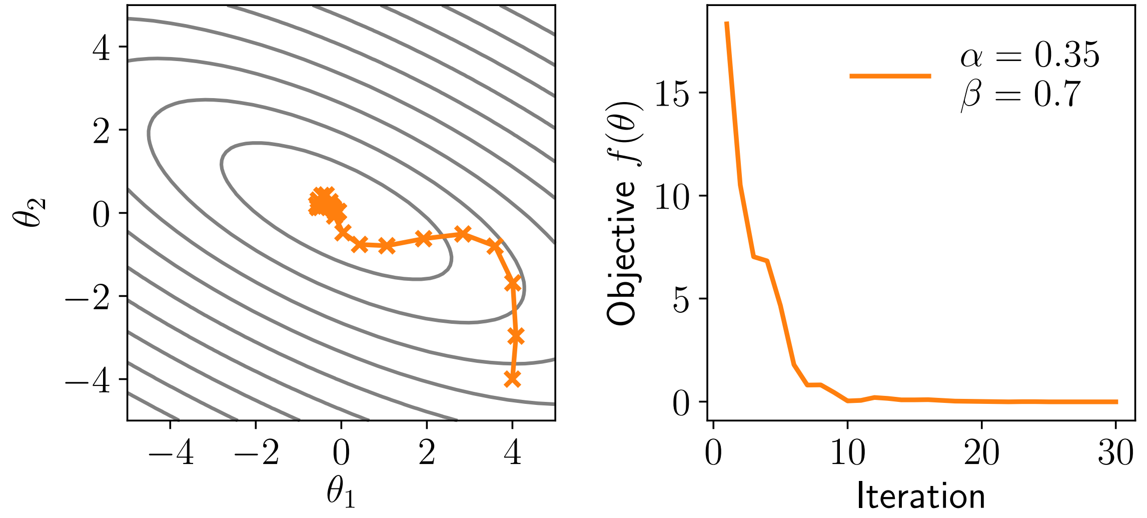
β is typically between 0.8 and 0.99. 0.9 is a decent starting value.

Illustration of momentum

Momentum “smooths” out the descent steps, but can also introduce other forms of oscillation and non-descent behavior

Frequently useful in training deep networks in practice





"Unbiasing" momentum terms

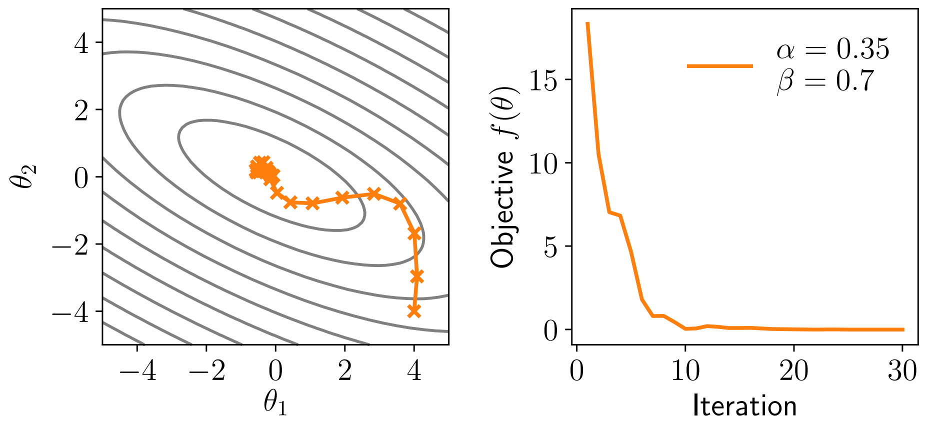
The momentum term u_t (if initialized to zero, as is common), will have smaller magnitude in initial iterations than in later ones. Results in the first few iterations taking smaller steps than vanilla GD, resulting in slower convergence.

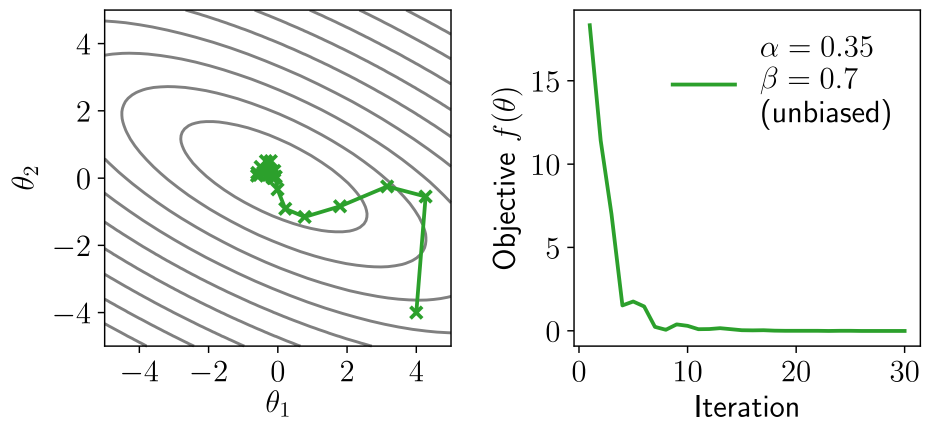
To "unbias" the update to have equal expected magnitude across all iterations, we can use the update

$$u_{t+1} = \beta u_t + (1-\beta)\nabla_\theta f(\theta_t)$$
$$\theta_{t+1} = \theta_t - \alpha u_{t+1}$$

$$u_{t+1} = \beta u_t + (1-\beta)\nabla_\theta f(\theta_t)$$
$$\theta_{t+1} = \theta_t - \alpha \frac{u_{t+1}}{(1-\beta^{t+1})}$$

Note: here, β^{t+1} means β raised to the power of (t+1)





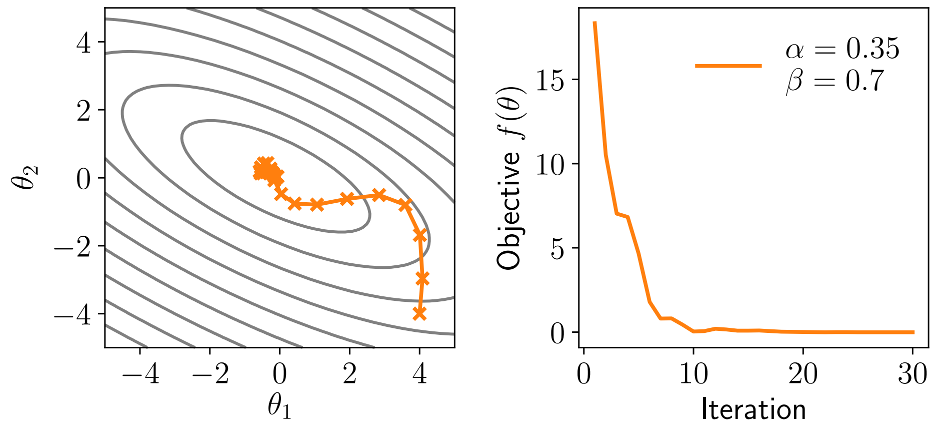
Nesterov Momentum

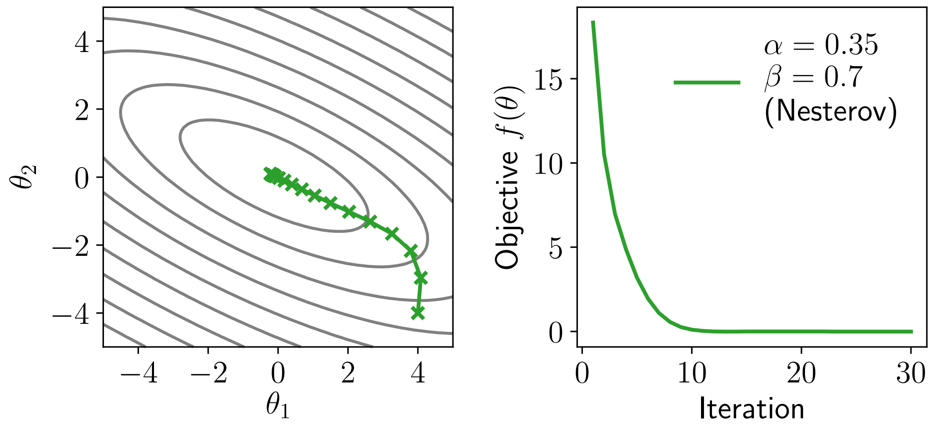
One (admittedly, of many) useful tricks is the notion of Nesterov momentum (or Nesterov acceleration), which computes momentum update at "next" point

$$\begin{aligned} u_{t+1} &= \beta u_t + (1-\beta)\nabla_\theta f(\theta_t) \\ \theta_{t+1} &= \theta_t - \alpha u_{t+1} \end{aligned} \implies \begin{aligned} u_{t+1} &= \beta u_t + (1-\beta)\nabla_\theta f(\theta_t - \alpha u_t) \\ \theta_{t+1} &= \theta_t - \alpha u_{t+1} \end{aligned}$$

"look ahead" one step

A "good" thing for convex optimization, and (sometimes) helps for deep networks





Feels like a simple tweak, but for convex functions this provably speeds up convergence (1/k to 1/k^2, where k is num iterations). Neat!

Gradient sparsity

Problem: sometimes, an iteration(s) of gradient descent will only update a **subset** of the model parameters.

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta f(\theta_t)$$

$$\begin{aligned}\theta_t &= [1 \quad 0 \quad -2 \quad 8] \\ \nabla_\theta f(\theta_t) &= [0.8 \quad 1 \quad \underbrace{0 \quad 0}_{\text{Sparse gradients}}]\end{aligned} \xrightarrow[\alpha = 0.1]{} \theta_{t+1} = [0.92 \quad -0.1 \quad \underbrace{-2 \quad 8}_{\substack{\text{Unchanged! Parameters} \\ \text{didn't get a chance to learn.}}}]$$

Further, due to dataset/batch characteristics, sometimes some of the model parameters are rarely updated, even after an entire epoch of training!

This is known as **"gradient sparsity"**, and results in poor learning for the parts of the model that rarely receive gradient updates.

Gradient sparsity: global learning rates

The fact that we're using the same learning rate for all model parameters ("global" learning rate) contributes to the gradient sparsity issue:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta f(\theta_t)$$

Gradient descent updates for a toy four-param model

$\alpha = 0.1$

Let θ_t mean: "θ at gradient descent iteration t"

$\theta_1 = [1 \quad 0 \quad -2 \quad 8]$

$\nabla_\theta f(\theta_1) = [0.8 \quad 1 \quad 0 \quad 0]$

$\theta_2 = [0.92 \quad -0.1 \quad -2 \quad 8]$

$\nabla_\theta f(\theta_2) = [0.1 \quad -0.2 \quad 0 \quad 0]$

$\theta_3 = [0.91 \quad -0.08 \quad -2 \quad 8]$

$\nabla_\theta f(\theta_3) = [0.2 \quad 0.5 \quad 1 \quad 2]$

$\theta_4 = [0.89 \quad -0.13 \quad -2.1 \quad 7.8]$

In these three minibatches (iterations):

$\theta[0], \theta[1]$ received: 3 updates

$\theta[2], \theta[3]$ received: 1 update ← Basically still at initial (random) point

All parameters θ use the same learning rate $\alpha = 0.1$.
Perhaps "unfair" to $\theta[2], \theta[3]$?

Idea: what if we had **per-parameter** learning rates?
Intuition: for params that rarely see gradients, scale up their gradients => increase their learning rates

Adam: "Adaptive Moment Estimation"

The *scale* of the gradients can vary widely for different parameters, especially e.g. across different layers of a deep network, different layer types, etc

So-called *adaptive gradient* methods attempt to estimate this scale over iterations and then re-scale the gradient update accordingly

Most widely used adaptive gradient method for deep learning is the **Adam** algorithm, which combines momentum and adaptive scale estimation

Gradient history ("momentum") $u_{t+1} = \beta_1 u_t + (1 - \beta_1)\nabla_\theta f(\theta_t)$

Gradient magnitude history $v_{t+1} = \beta_2 v_t + (1 - \beta_2)\left(\nabla_\theta f(\theta_t)\right)^2$ ← Square is done elementwise

$$\theta_{t+1} = \theta_t - \alpha \frac{u_{t+1}}{\left(\sqrt{v_{t+1}} + \epsilon\right)}$$

← Division is done elementwise

Square root is done elementwise

To avoid dividing by zero (common trick in deep learning) → $\epsilon = 10^{-6}$

Originally published in ICLR 2015: "Adam: A Method for Stochastic Optimization"

Adam intuition: sparse gradients

Adam update:

Gradient history ("momentum") $u_{t+1} = \beta_1 u_t + (1-\beta_1)\nabla_\theta f(\theta_t)$

Gradient magnitude history $v_{t+1} = \beta_2 v_t + (1-\beta_2)\left(\nabla_\theta f(\theta_t)\right)^2$ ← Square is done elementwise

$$\theta_{t+1} = \theta_t - \alpha \frac{u_{t+1}}{\left(\sqrt{v_{t+1}} + \epsilon\right)}$$

← Division is done elementwise

Square root is done elementwise

Intuition: Adam uses per-parameter learning rates (vs single global learning rate like SGD)
For parameters with a **history of large** gradient magnitudes: **downweight** it
For parameters with a **history of small** gradient magnitudes, **upweight** it

Suppose param 3 (in red) suffers from sparse gradient problem.

$u_{t+1} = [1, 2, 0.2, 4]$

$v_{t+1} = [1, 8, 0.4, 4]$

$$\frac{u_{t+1}}{\sqrt{v_{t+1}}} = [1, 0.707, 3.16, 2]$$

(ignoring epsilon)

Effectively scaled up gradient for param 3: 0.2 -> 3.16

Adam intuition: scale normalization

Adam update:

Gradient history ("momentum") $u_{t+1} = \beta_1 u_t + (1-\beta_1)\nabla_\theta f(\theta_t)$

Gradient magnitude history $v_{t+1} = \beta_2 v_t + (1-\beta_2)\left(\nabla_\theta f(\theta_t)\right)^2$ ← Square is done elementwise

$$\theta_{t+1} = \theta_t - \alpha \frac{u_{t+1}}{\left(\sqrt{v_{t+1}} + \epsilon\right)}$$ ← Division is done elementwise

Square root is done elementwise

Adam can also mitigate issues with differences in gradient magnitudes:
For parameters with **large** gradient magnitudes: **downweight** it
For parameters with **small** gradient magnitudes, **upweight** it

Note: it's not clear that discarding gradient magnitude information is always the "right" thing to do. Sometimes it's not! But, in practice Adam works quite well..."empirically"

(for fun, optional) SignSGD
Also: "Noise Is Not the Main Factor Behind the Gap Between SGD and Adam on Transformers, but Sign Descent Might Be"

Problem: gradient scale is massively different for different params (high dynamic range). If we're not careful, can result in training instability

$$\nabla_\theta f(\theta_t) = [1, 500, 0.0001, -8]$$

$$\left(\nabla_\theta f(\theta_t)\right)^2 = [1{,}250000{,}1e-08, 64]$$

$$\frac{\nabla_\theta f(\theta_t)}{\sqrt{\left(\nabla_\theta f(\theta_t)\right)^2}} = [1, 1, 1, -1]$$

(ignoring gradient history and gradient-magnitude history)

Now, the massive gradient magnitude scale differences are gone!

Adam with bias correction

Common to apply bias correction as well:

$$u_{t+1} = \beta_1 u_t + (1-\beta_1)\nabla_\theta f(\theta_t)$$
$$v_{t+1} = \beta_2 v_t + (1-\beta_2)\left(\nabla_\theta f(\theta_t)\right)^2$$
$$\theta_{t+1} = \theta_t - \alpha \frac{u_{t+1}}{\left(\sqrt{v_{t+1}} + \epsilon\right)}$$

$$u_{t+1} = \beta_1 u_t + (1-\beta_1)\nabla_\theta f(\theta_t)$$
$$\hat{u}_{t+1} = \frac{u_{t+1}}{(1-\beta_1^{t+1})}$$
$$v_{t+1} = \beta_2 v_t + (1-\beta_2)\left(\nabla_\theta f(\theta_t)\right)^2$$
$$\hat{v}_{t+1} = \frac{v_{t+1}}{(1-\beta_2^{t+1})}$$
$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{u}_{t+1}}{\left(\sqrt{\hat{v}_{t+1}} + \epsilon\right)}$$

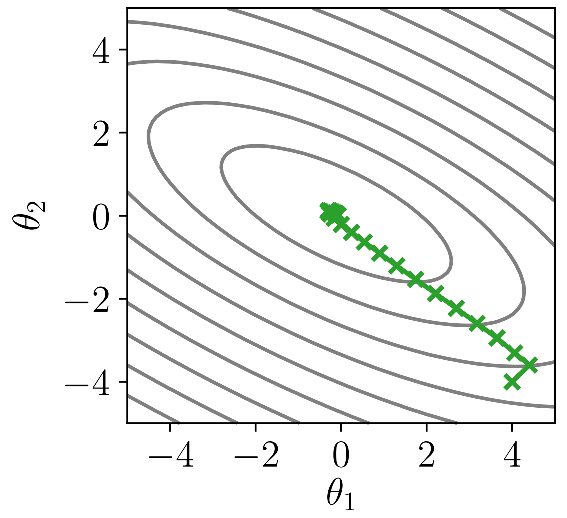
Typical choices: $\beta_1 = 0.9$, $\beta_2 = 0.999$.
Means: variance estimate $\hat{v}_{t+1}$ moves much more slowly than the momentum term $\hat{u}_{t+1}$.

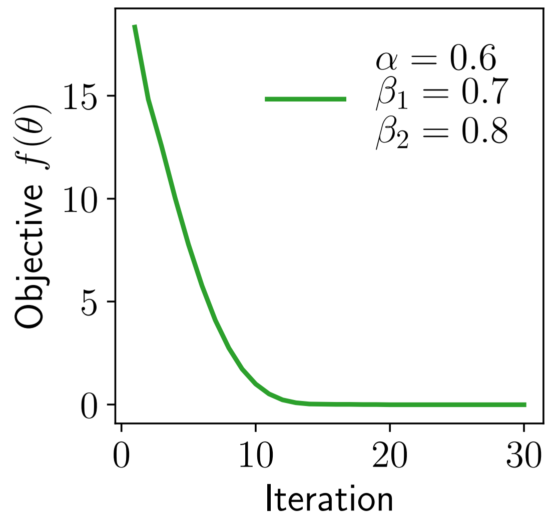
$\epsilon = 10^{-6}$

Notes on / illustration of Adam

Whether Adam is “good” optimizer is endlessly debated within deep learning, but it often seems to work quite well in practice (maybe?)

There are alternative universes where endless other variants became the “standard” (no unbiasing? average of absolute magnitude rather than squared? Nesterov-like acceleration?) but Adam is well-tuned and hard to uniformly beat



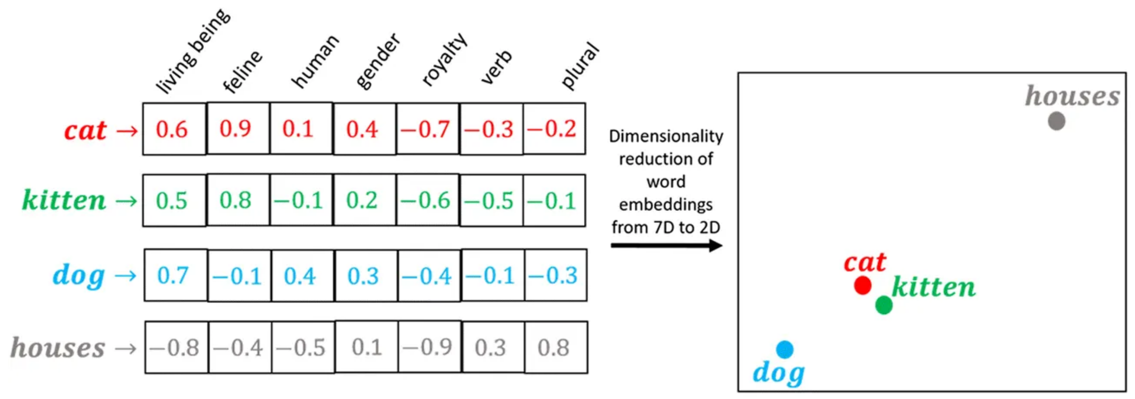


Word embeddings

In natural language processing, a common technique is to learn a representation for each word: **word embeddings.**

Goal: want word embeddings to encode **semantics**.

Example: want word embeddings for "cat" and "kitten" to be closer than, say, word embeddings for "cat" and "houses".



	living being	feline	human	gender	royalty	verb	plural
cat →	0.6	0.9	0.1	0.4	−0.7	−0.3	−0.2
kitten →	0.5	0.8	−0.1	0.2	−0.6	−0.5	−0.1
dog →	0.7	−0.1	0.4	0.3	−0.4	−0.1	−0.3
houses →	−0.8	−0.4	−0.5	0.1	−0.9	0.3	0.8

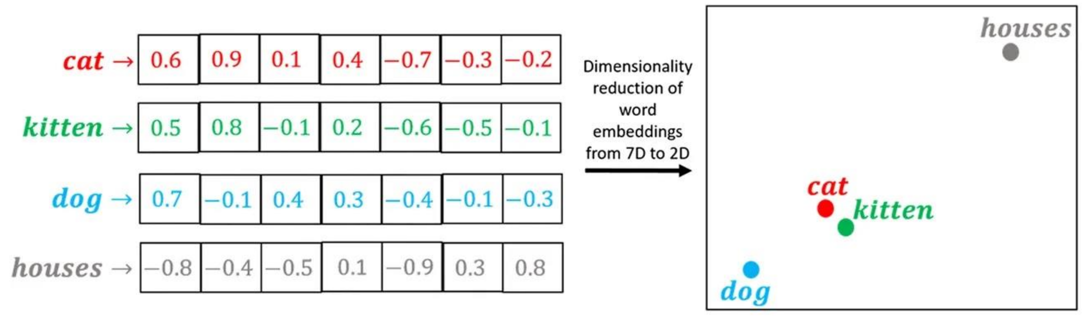
https://medium.com/@hari4om/word-embedding-d816f643140

Word embeddings: embedding tables

Word embeddings are typically implemented where we map each word in a vocabulary to model parameters. Called **"Embedding table"**.

Word embedding tables are trained in addition to subsequent layers (eg Linear layers).

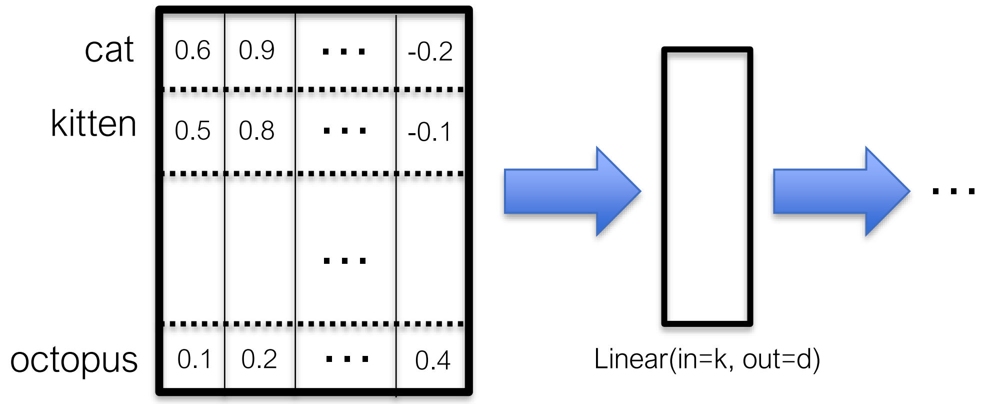
Key: model figures out what the word embeddings should be so that they "work" for downstream tasks (eg encode semantics).



cat →	0.6	0.9	0.1	0.4	−0.7	−0.3	−0.2
kitten →	0.5	0.8	−0.1	0.2	−0.6	−0.5	−0.1
dog →	0.7	−0.1	0.4	0.3	−0.4	−0.1	−0.3
houses →	−0.8	−0.4	−0.5	0.1	−0.9	0.3	0.8

https://medium.com/@hari4om/word-embedding-d816f643140

Word embedding table (with vocab size n, embed dim k)



cat	0.6	0.9	···	-0.2
kitten	0.5	0.8	···	-0.1
			···	
octopus	0.1	0.2	···	0.4

aka a **learnable** parameter matrix with shape=[n, k]
Ex: torch.nn.Embedding

Gradient sparsity, Adam, and word embeddings

Example: suppose "octopus" is a **rare** word in our training dataset.

Without Adam: the word embedding for "octopus" will receive very few gradient updates ("sparse gradients"), and each update has tiny impact (due to shared global learning rate).

Impact: "octopus" embedding vector will likely be poor quality (not far from random initialization).

With Adam: the "octopus" word embedding gradient updates use a larger effective learning rate, so each update is more impactful.

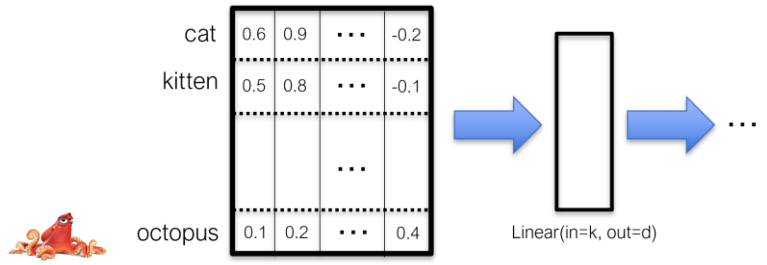
Question: exactly how does Adam mitigate the sparse gradient issue here? Eg for rare "octopus"?

Answer: Gradient magnitude history $\hat{v}_{t+1}$ for "octopus" params will be small (eg 0.01), and dividing by a small value -> scale by large value

Another great way to mitigate sparse gradients is to avoid them in the first place: have large datasets, large representative batches, etc.

Note: in practice, we typically tokenize text and learn "token" embeddings, rather than word embeddings. Reduces vocab size!
Note: We'll revisit embedding tables and natural language processing in more detail later in the course.

Word embedding table (with vocab size n, embed dim k)



cat	0.6	0.9	...	-0.2
kitten	0.5	0.8	...	-0.1
			...	
octopus	0.1	0.2	...	0.4

aka a **learnable** parameter matrix with shape=[n, k]
Ex: torch.nn.Embedding

$$u_{t+1} = \beta_1 u_t + (1-\beta_1)\nabla_\theta f(\theta_t)$$
$$\hat{u}_{t+1} = \frac{u_{t+1}}{(1-\beta_1^{t+1})}$$
$$v_{t+1} = \beta_2 v_t + (1-\beta_2)\left(\nabla_\theta f(\theta_t)\right)^2$$
$$\hat{v}_{t+1} = \frac{v_{t+1}}{(1-\beta_2^{t+1})}$$
$$\theta_{t+1} = \theta_t - \alpha\frac{\hat{u}_{t+1}}{\left(\sqrt{\hat{v}_{t+1}}+\epsilon\right)}$$

https://pixar.fandom.com/wiki/Hank

"Stochastic" gradient descent

"Standard" gradient descent would run through your entire dataset to calculate the gradient.

However, in deep learning, it's common to instead calculate your gradient over a subset of the dataset, called a "minibatch". This is known as "stochastic" gradient descent. This is done for practical reasons.

Fortunately, the math for "full dataset" vs "minibatch" gradient descent works out: both produce unbiased estimates of the gradient.

	Quality of gradient $\nabla_\theta f(\theta_t)$	**Speed**
"Full batch" gradient descent	Better	Slower
"Stochastic" gradient descent	Worse (noisier)	Faster
"Stochastic" gradient descent with tiny batchsize (say, batchsize=1)	Worst (noisiest)	Fastest

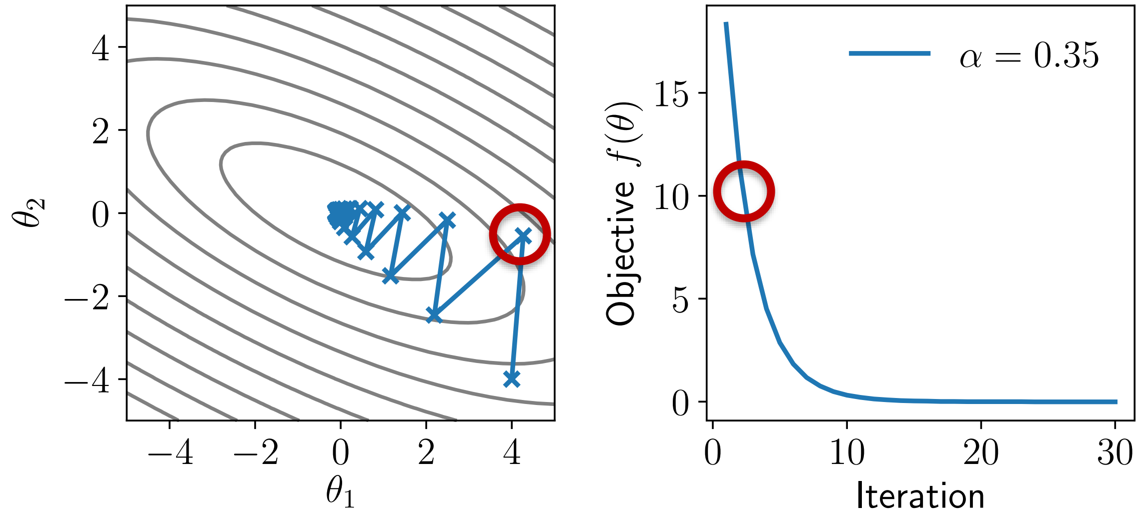
SGD: quality vs speed tradeoff

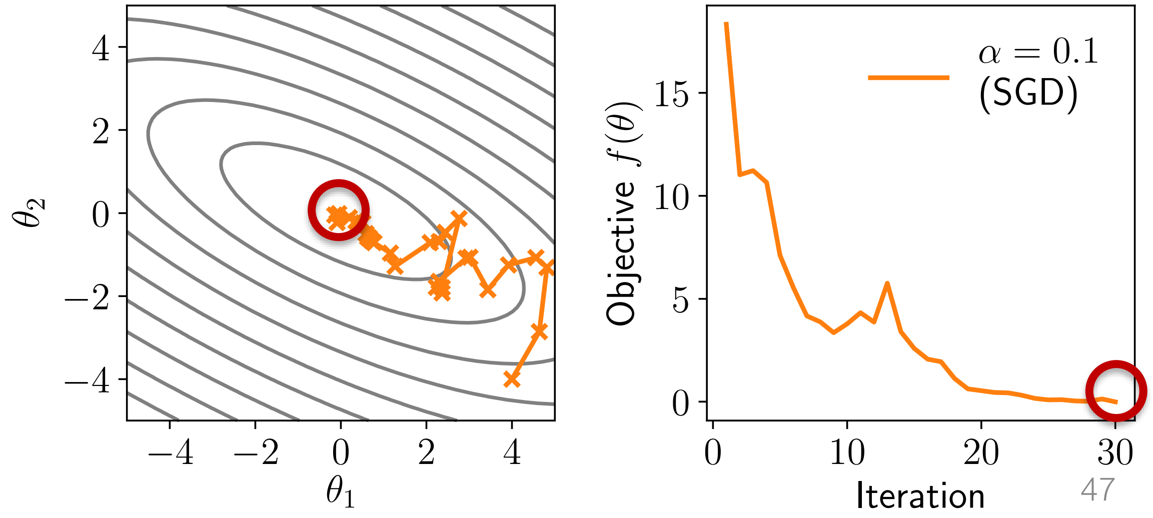
This leads us again to the SGD algorithm, repeating for batches $B \subset \{1, \ldots, m\}$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{B} \sum_{i \in B} \nabla_\theta \ell\left(h\left(x^{(i)}\right), y^{(i)}\right)$$

Tip: since decreasing batchsize leads to noisier gradients, we typically reduce learning rate accordingly (trust gradient less -> take smaller step). Linear scaling rule is a good first thing to try: double batchsize => half learning rate.

Instead of taking a few expensive, noise-free, steps, we take *many* cheap, noisy steps, which ends having much strong performance per compute





Optimization in deep learning: some takeaways

In deep learning, we pretty much always use "stochastic" (eg mini-batch) versions of optimization algorithms.

Regarding convex optimization: the amount of valid intuition that carriers over from simple (convex) optimization to complex (deep learning) problems is limited.

At the end of the day: it's all empirical. Run lots of experiments! It's valuable to keep up with research/trends to see what tends to work well for others.

Outline

Fully connected networks

Optimization

Initialization

Initialization of weights

Recall that we optimize parameters iteratively by stochastic gradient descent, e.g.

$$W_i := W_i - \alpha \nabla_{W_i} \ell(h_\theta(X), y)$$

But how do we choose the *initial* values of W_i, b_i? (maybe just initialize to zero?)

Recall the manual backpropagation forward/backward passes (without bias):

$$\begin{aligned} Z_{i+1} &= \sigma_i(Z_i W_i) \\ G_i &= \left(G_{i+1} \circ \sigma_i'(Z_i W_i)\right) W_i^T \end{aligned}$$

- If $W_i = 0$, then $G_j = 0$ for $j \leq i$, $\Longrightarrow \nabla_{W_i} \ell(h_\theta(X), y) = 0$
- I.e., $W_i = 0$ is a (very bad) local optimum of the objective

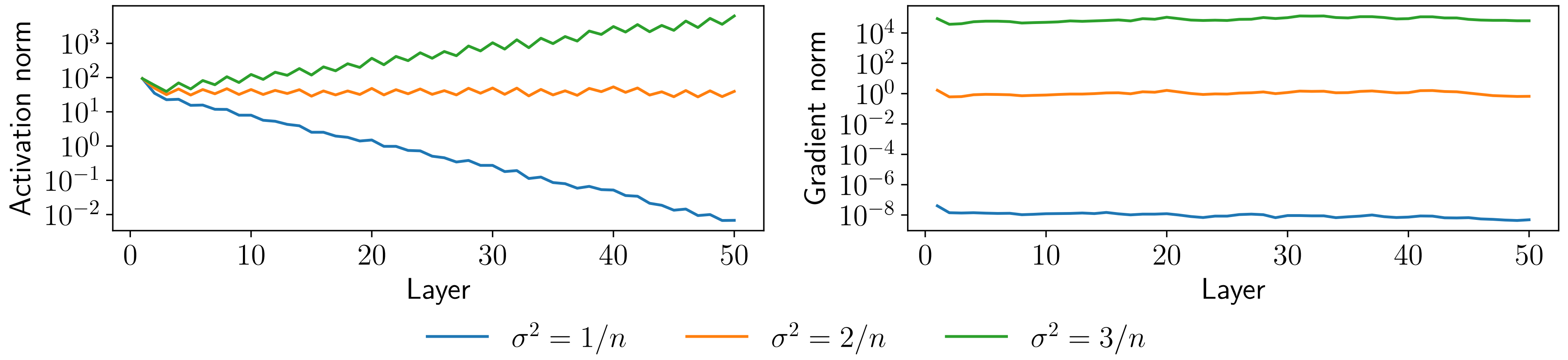
Key idea #1: Choice of initialization matters

Let's just initialize weights "randomly", e.g., $W_i \sim \mathcal{N}(0, \sigma^2 I)$

The choice of variance σ^2 will affect two (related) quantities:

1. The norm of the forward activations Z_i
2. The norm of the the gradients $\nabla_{W_i} \ell(h_\theta(X), y)$

Illustration on MNIST with $n = 100$ hidden units, depth 50, ReLU nonlinearities



$\sigma^2 = \frac{1}{n}$: activations shrink with model depth (bad! Can lead to slow training, "vanishing gradient problem")

$\sigma^2 = \frac{2}{n}$ is the "best" here: stable activation norms, and "healthy" gradient norm.

$\sigma^2 = \frac{3}{n}$: activations grow with model depth (bad! Can lead to exploding gradients, which can lead to overshooting and training instability)

Key idea #2: Weights don’t move “that much”

Might have the picture in your mind that the parameters of a network converge to some similar region of points regardless of their initialization

This is not true … weights often stay much closer to their initialization than to the “final” point after optimization from different

End result: initialization matters … we’ll see some of the practical aspects next lecture

Linear layer initialization strategy (1/2)

Suppose we have a linear layer L parameterized with weight matrix W, defined as: $L(x) = xW$, where x has shape=[1, n], and W has shape=[n, m].

Suppose the inputs to L are normally distributed: $x \sim \mathcal{N}(0,1)$

Goal: to keep activations "healthy" (eg don't grow/shrink with model depth), we'd like the outputs L(x) to also be normally distributed: $\mathrm{L}(x) \sim \mathcal{N}(0,1)$

Question: how do we initialize W to achieve $\mathrm{L}(x) \sim \mathcal{N}(0,1)$?

Hint: let $w \sim \mathcal{N}(0, \sigma^2)$. What would be a "good" value for σ^2?

Linear layer initialization strategy (2/2)

Question: how do we initialize W to achieve $\mathrm{L}(x) \sim \mathcal{N}(0,1)$?

Hint: let $w \sim \mathcal{N}(0, \sigma^2)$. What would be a "good" value for σ^2?

Answer: $\sigma^2 = \frac{1}{n}$.

n is shape of x

Proof: Consider independent random variables $x \sim \mathcal{N}(0,1)$, $w \sim \mathcal{N}\left(0, \frac{1}{n}\right)$; then

$$\mathbf{E}[x_i w_i] = \mathbf{E}[x_i]\mathbf{E}[w_i] = 0, \qquad \mathbf{Var}[x_i w_i] = \mathbf{Var}[x_i]\mathbf{Var}[w_i] = 1/n$$

Since x, w are independent

so $\mathbf{E}[w^T x] = 0$, $\mathbf{Var}[w^T x] = 1$ ($w^T x \to \mathcal{N}(0,1)$ by central limit theorem)

Thus, informally speaking, L(x) achieves our goal: $x_i \sim \mathcal{N}(0, I)$, $W_i \sim \mathcal{N}\left(0, \frac{1}{n} I\right)$ then $L(x) = W_i^T z_i \sim \mathcal{N}(0, I)$

If we use a ReLU nonlinearity: then "half" the components of $L(x)$ will be set to zero, so we need double the variance on W_i to achieve the same final variance, hence $W_i \sim \mathcal{N}\left(0, \frac{2}{n} I\right)$ (Kaiming normal initialization)