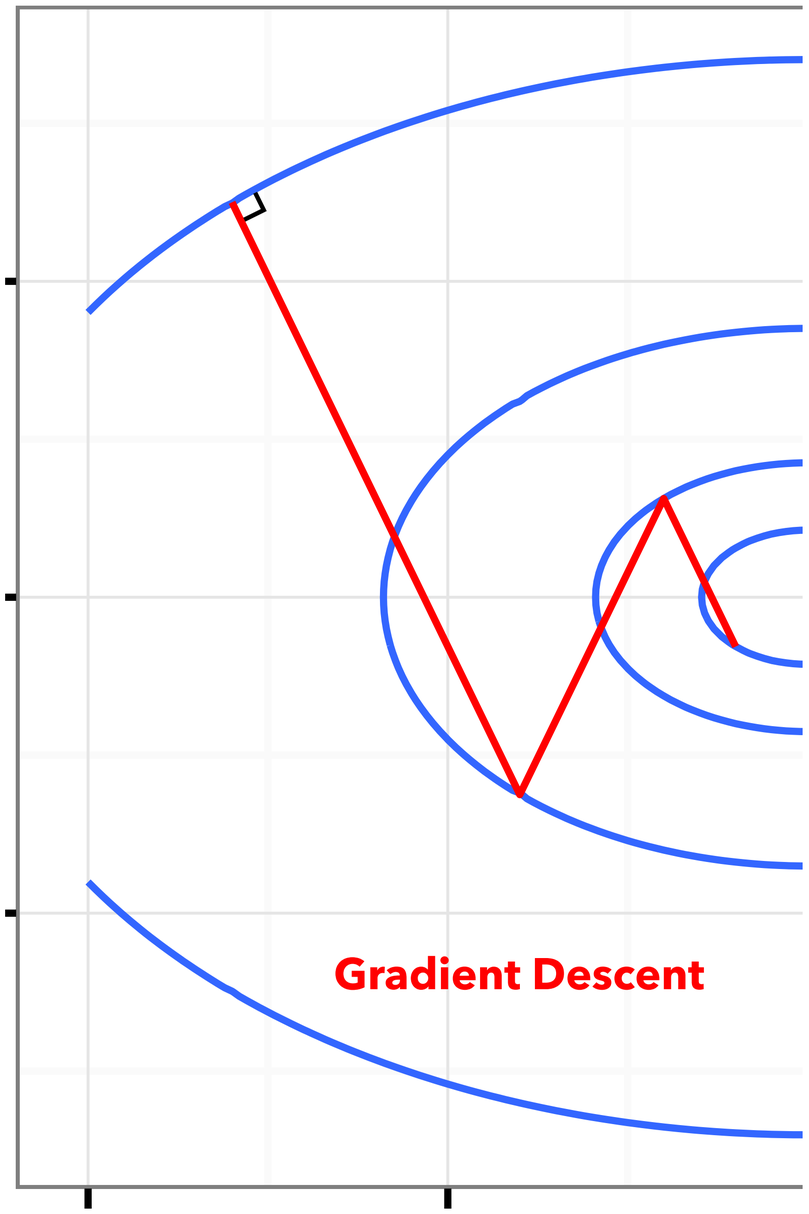
Optimization and the Death of Optimization

A lecture on optimization for deep learning

Aaron Defazio

Facebook AI Research



Covering

- Gradient descent & stochastic gradient descent
- Momentum
- Diagonal rescaling: RMSprop & Adam
- Normalization layers
- Solving optimization problems without optimizing
- Application: MRI reconstruction

Gradient descent

“The worst optimization method in the world”

Problem $\underset{w}{\text{minimize}}\ f(w)$

We denote the optimizing input w_*

Solution (iterative)

$$w_{k+1} = w_k - \gamma_k \nabla f(w_k)$$

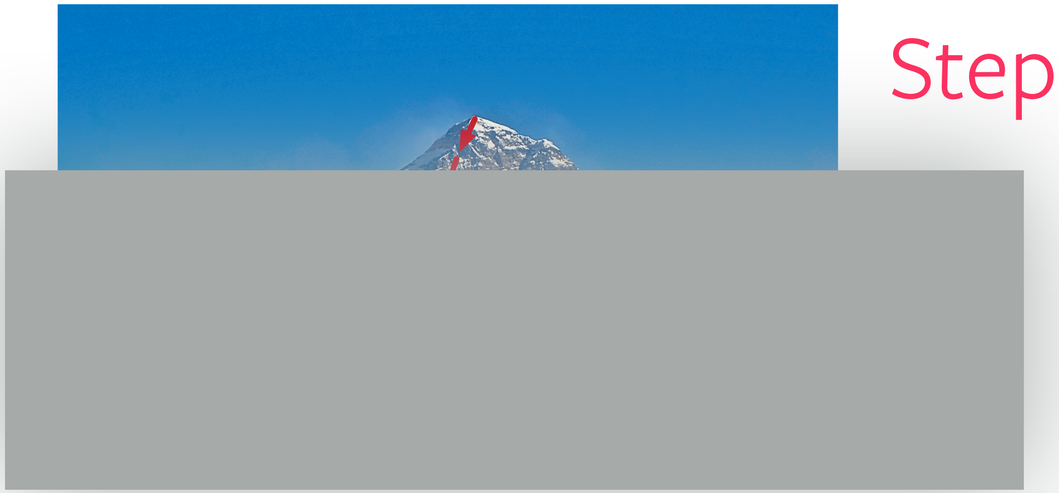
Step size

Next iterate

Previous iterate

Gradient of *f*

Side note: You should probably never use gradient descent directly, consider it a building block for other methods.



Step 1

Intuition: We can't tell from looking "locally" the exact Direction to the solution
The negative gradient direction is the steepest direction locally, taking a small negative gradient step can only take you closer to the minimum

Step 2

Step 3

Source: https://www.deepideas.net/deep-learning-from-scratch-iv-gradient-descent-and-backpropagation/

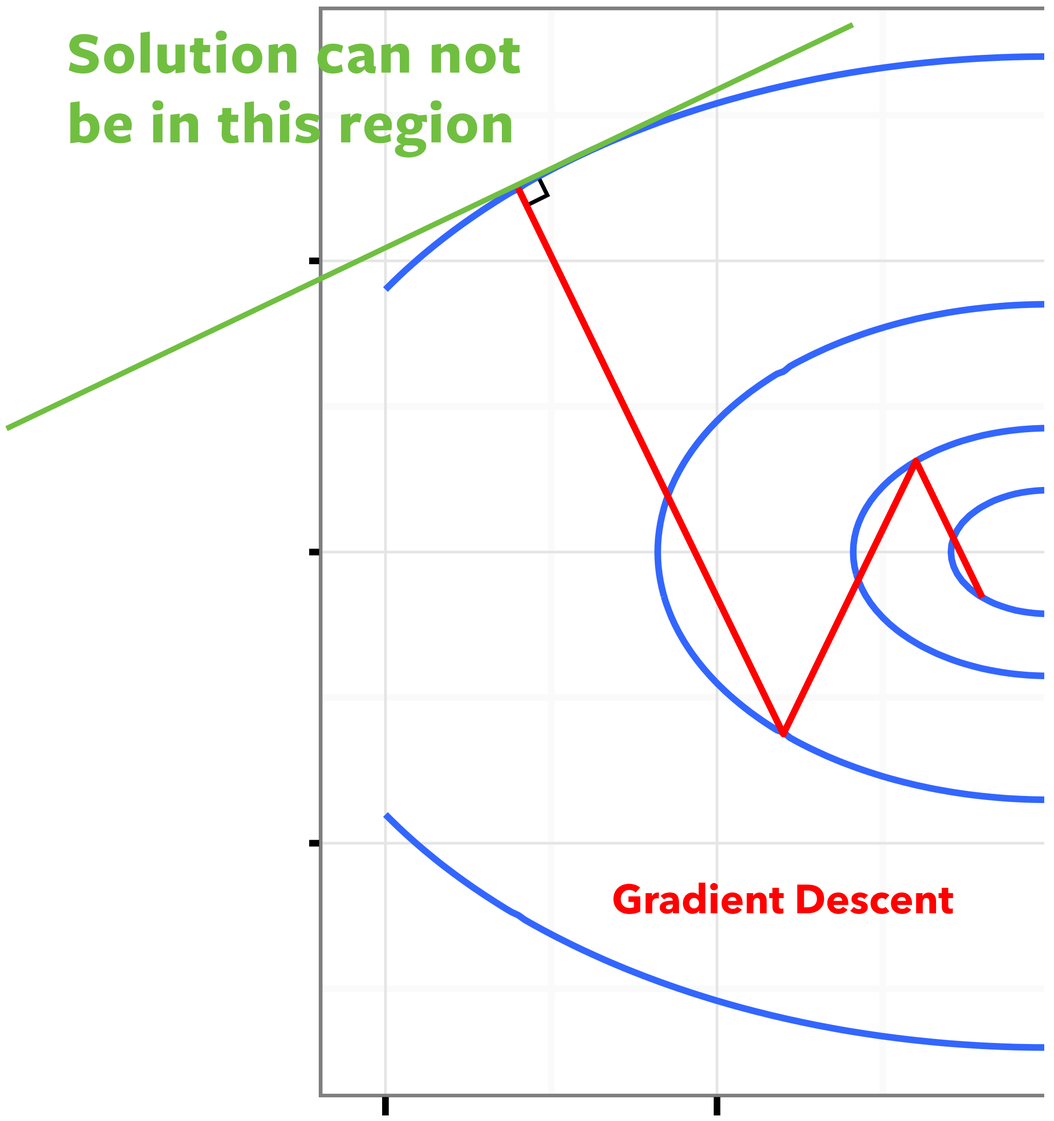
Intuition in **quadratic** case

Blue lines are the level sets of the problem
(Lines of constant function value)

Think: Topographic map of a 2D function,
Where solution is the bottom of a valley.

Red lines are gradient descent path

The negative gradient indicates the
direction of “steepest descent”,
Orthogonal to the level set at the current
point



Theory in the quadratic (positive definite) case

$$f(w) = \frac{1}{2} w^\top A w - b^\top w$$
$$\nabla f(w) = Aw - b$$
$$w_* = A^{-1} b$$

$$w_{k+1} = w_k - \gamma \nabla f(w_k)$$

$$\begin{aligned}
\|w_{k+1} - w_*\| &= \|w_k - \gamma(Aw_k - b) - w_*\| \\
&= \left\|w_k - \gamma A(w_k - A^{-1}b) - w_*\right\| \\
&= \|w_k - \gamma A(w_k - w_*) - w_*\| \\
&= \|(I - \gamma A)(w_k - w_*)\| \\
&\leq \|I - \gamma A\| \, \|w_k - w_*\|
\end{aligned}$$

Let μ be the minimum eigenvalue of A and L the max. Then the extremal eigenvalues of $I - \gamma A$ are $1 - \mu\gamma$ and $1 - L\gamma$, so:

$$\|I - \gamma A\| = \max\{|1 - \mu\gamma|, |1 - L\gamma|\}$$

If we use two large a step size $1 - L\gamma$ becomes negative. A standard value is $\gamma = 1/L$ which gives:

$$\|w_{k+1} - w_*\| \leq \left(1 - \frac{\mu}{L}\right) \|w_k - w_*\|$$

The convergence of optimization methods depends on the condition number of a problem

Inverse of the condition number

$$\|w_{k+1} - w_*\| \leq \left(1 - \frac{\mu}{L}\right) \|w_k - w_*\|$$

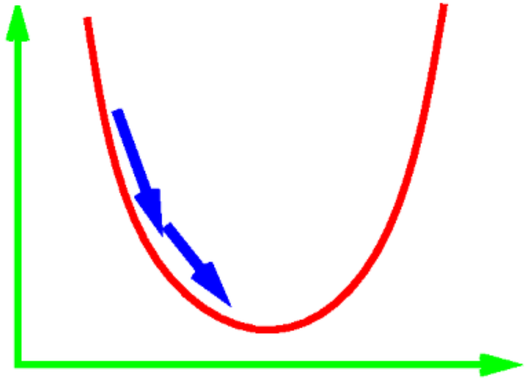
The condition number only formally makes sense on simple problems (“strongly convex”)
But we often talk about “**poorly conditioned**” and “**well conditioned**” problems in machine learning informally

Typically we don't have a good estimate of the learning rate!

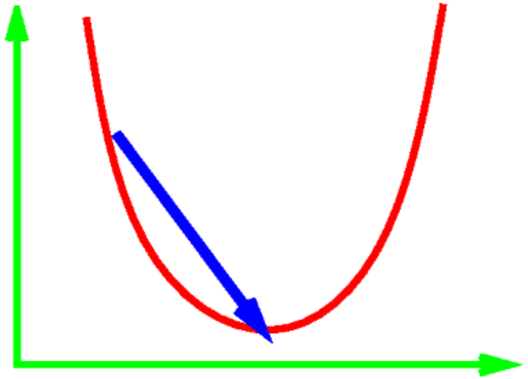
Standard practice is to try a bunch of values on a log scale and use the one that gave the best final result

Learning rates that are too large cause "divergence", where the function value (loss) explodes

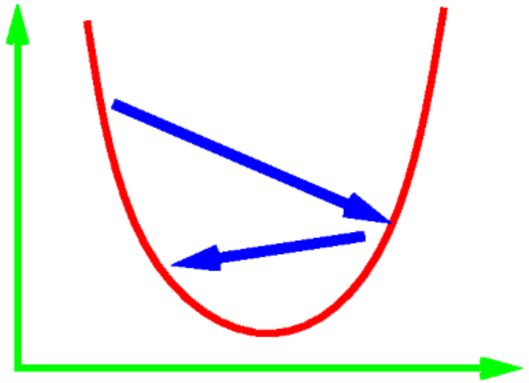
The optimal learning rate can change during optimization! Often decreasing it over time is necessary



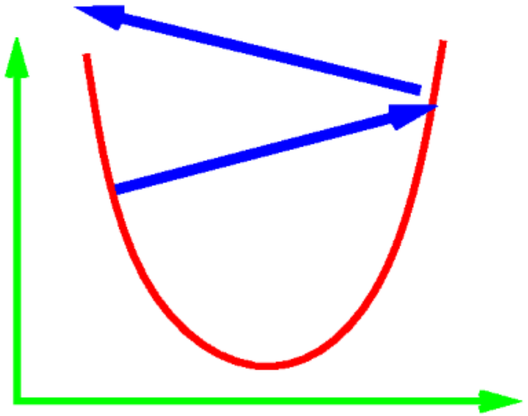
$\gamma < \gamma_{\text{opt}}$



$\gamma = \gamma_{\text{opt}}$

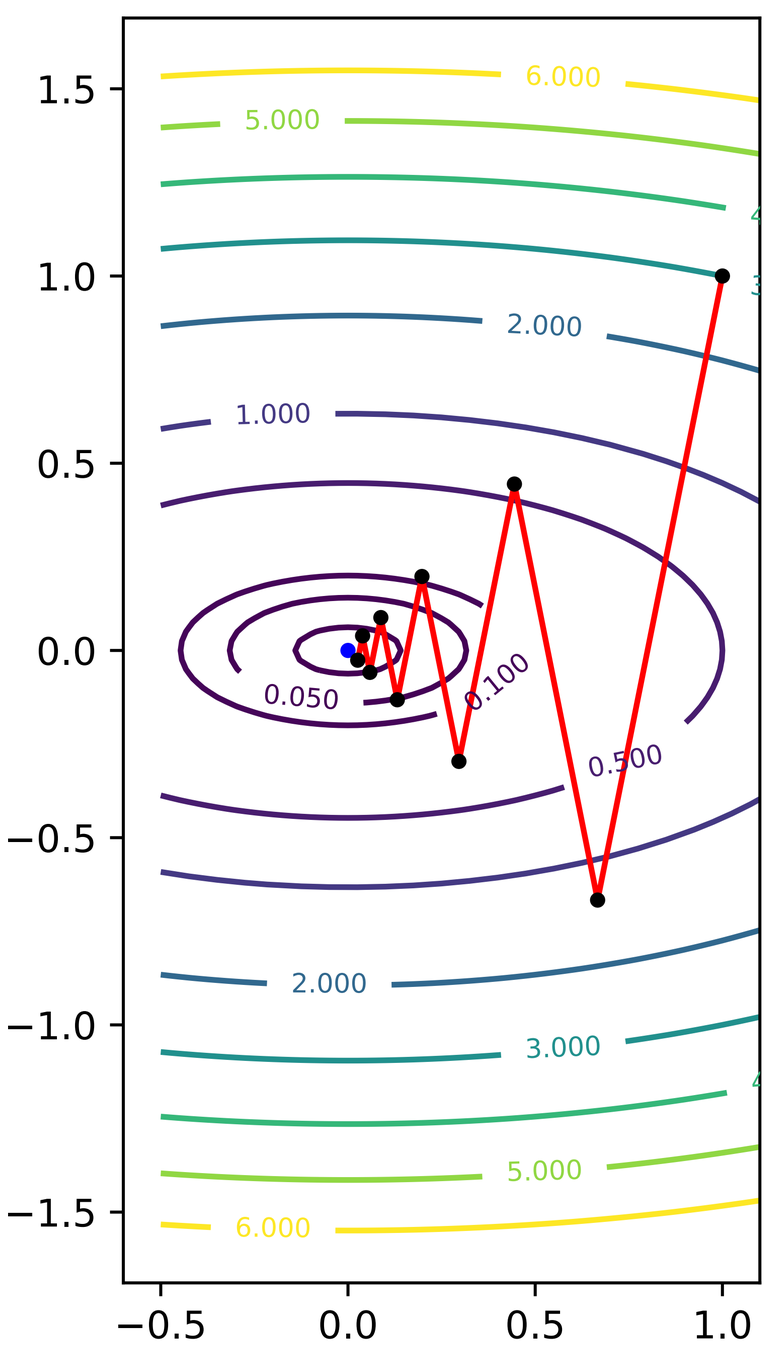


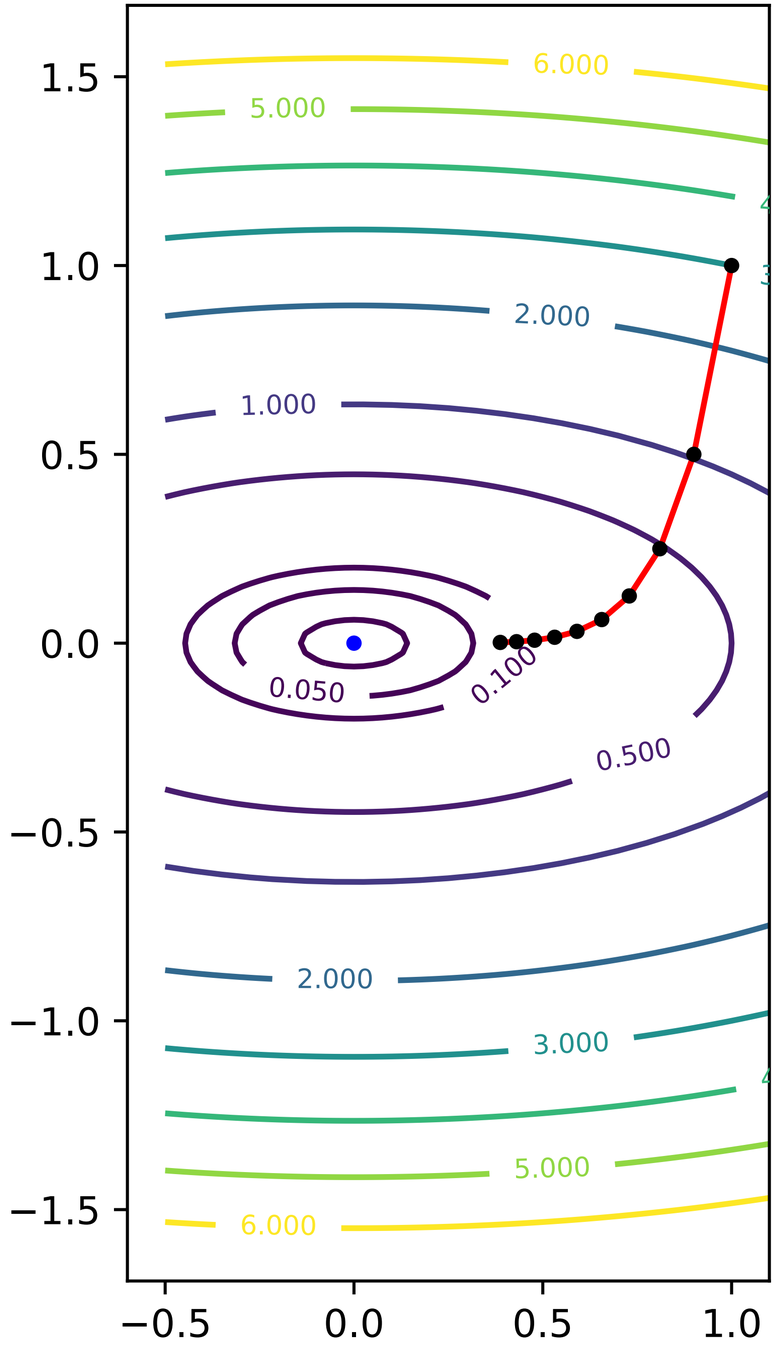
$\gamma > \gamma_{\text{opt}}$



$\gamma \gtrapprox 2\gamma_{\text{opt}}$

Unfortunately for us, in practice the fastest convergence is using a learning rate that is close to diverging.
Use as **large as LR as possible**, but not larger!





Stochastic Optimization

3

Or: why a rough estimate now is better than a good estimate in a week

Stochastic optimization

$$\underset{w}{\text{minimize}}\ f(w) = \frac{1}{n}\sum_{i}^{n} f_i(w)$$

Typically each f_i is the loss of the neural network on a single instance

$$f_i(w) = \ell(x_i, y_i, w)$$

Note: the optimization community uses f by convention, so I follow that notation here

$$w_{k+1} = w_k - \gamma_k \nabla f(w_k)$$ GD: *The WORST method in virtually all situations*

$$w_{k+1} = w_k - \gamma_k \nabla f_i(w_k)$$ SGD: *Often the BEST method available!*

(i chosen uniformly at random)

WHY?

SGD is GD in expectation:

Since:

$$\mathbb{E}[\nabla f_i(w_k)] = \nabla f(w_k)$$

The SGD step's expectation is just the gradient step:

$$\mathbb{E}\left[w_{k+1}\right] = w_k - \gamma_k \nabla f(w_k)$$

It's useful to think of SGD as GD with **noise**.

Advantages of SGD

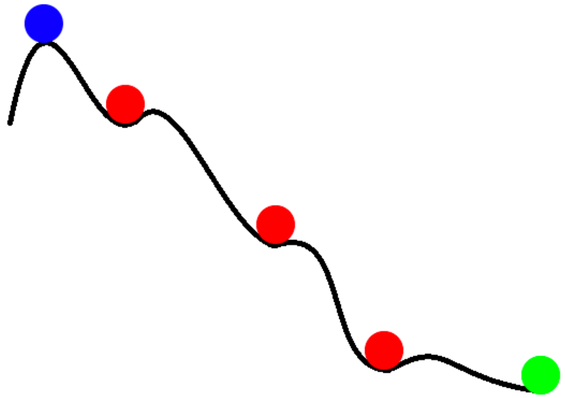
$$w_{k+1} = w_k - \gamma_k \nabla f_i(w_k)$$

$$\underset{w}{\text{minimize}}\ f(w) = \frac{1}{n}\sum_{i}^{n} f_i(w)$$

$$f_i(w) = \ell(x_i, y_i, w)$$

- There is redundant information across instances (i.e. MNIST has thousands of near identical digits) the noise is lower when there is higher redundancy.

- At the early stages of optimization, the noise is small compared to the information in the gradient, so a SGD step is virtually as good as a GD step.



- The noise can prevent the optimizing converging to bad local minima, a phenomena called annealing

- Stochastic gradients are drastically cheaper to compute (proportional to your dataset size), so you can often take thousands of SGD steps for the cost of one GD step.

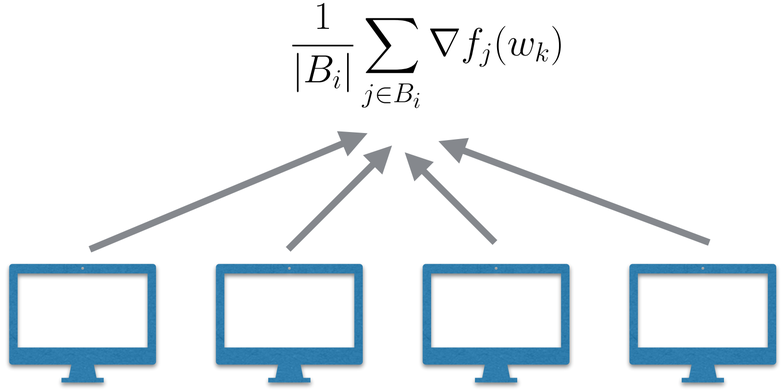
Mini-batching

$$w_{k+1} = w_k - \gamma_k \frac{1}{|B_i|} \sum_{j \in B_i} \nabla f_j(w_k)$$

Often we are able to make better use of our hardware by using mini batches instead of single instances

For instance, modern graphics cards are poorly utilized if we try and calculate single instance batches.

The most common distributing training technique on a cluster involves splitting a large mini batch between the machines and aggregating the resulting gradient.



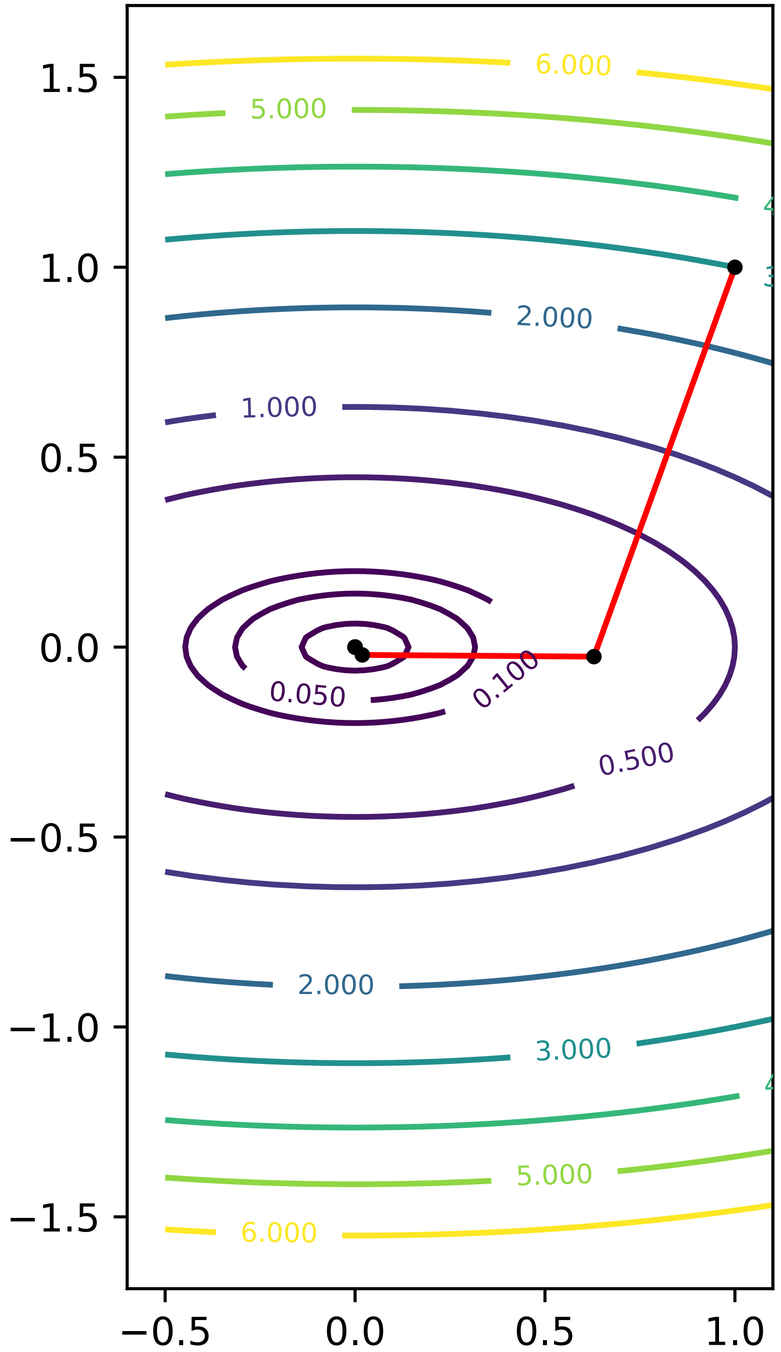
All methods we discuss for stochastic optimization work with mini batches without issue

If you must use a full-batch method ...

For batch optimization, **LBFGS** & Conjugate-Gradients are far better than vanilla gradient descent

`torch.optim.LBFGS` **scipy.optimize.fmin_l_bfgs_b**

BUT we don't know how to best combine these techniques with stochasticity!
Plain SGD (with momentum) is still the best method for training many state-of-the-art neural networks such as ResNet models.



Liu, D. C.; Nocedal, J. (1989). "On the Limited Memory Method for Large Scale Optimization"

Momentum

3

Who says there is no such thing as a free lunch?

Momentum

The most misunderstood idea in optimization?

$$p_{k+1} = \hat{\beta}_k p_k + \nabla f_i(w_k)$$
$$w_{k+1} = w_k - \gamma_k p_{k+1}$$

Extra term on top of SGD

SGD + Momentum = Stochastic **heavy ball** method

$$w_{k+1} = w_k - \gamma_k \nabla f_i(w_k) + \beta_k (w_k - w_{k-1})$$

$$0 \leq \beta < 1$$

This is mathematically equivalent to the previous form for a particular value of beta.

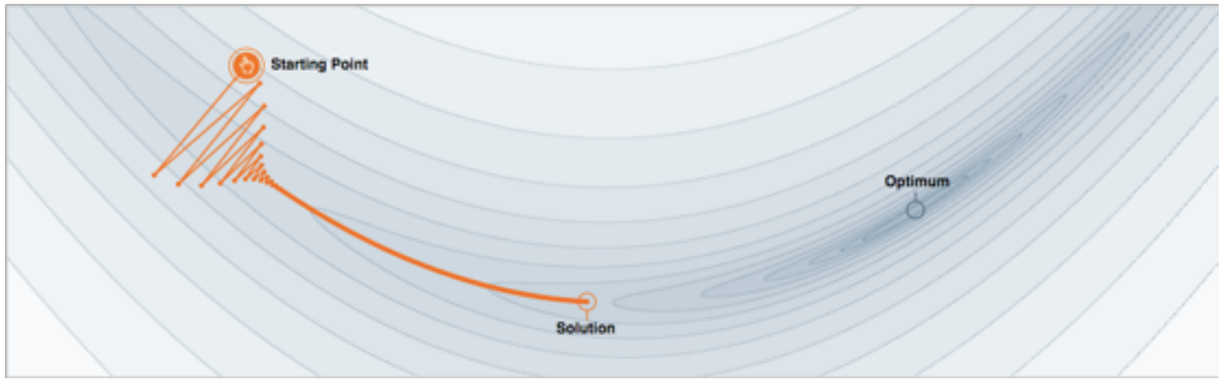
Key idea: The next step becomes a combination of the previous step's direction and the new negative gradient

Intuition

$$p_{k+1} = \hat{\beta}_k p_k + \nabla f_i(w_k)$$
$$w_{k+1} = w_k - \gamma_k p_{k+1}$$

The optimization process resembles a heavy ball rolling down a hill. The ball has momentum, so it doesn't change direction immediately when it encounters changes to the landscape!

Without momentum



With momentum

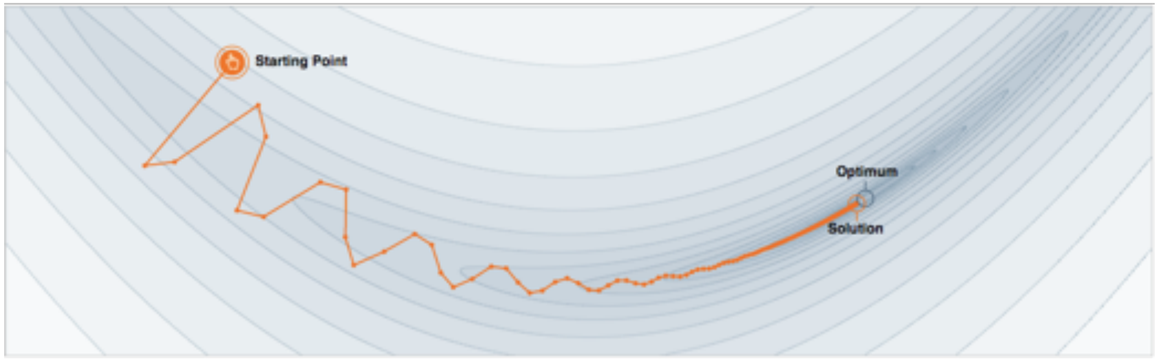
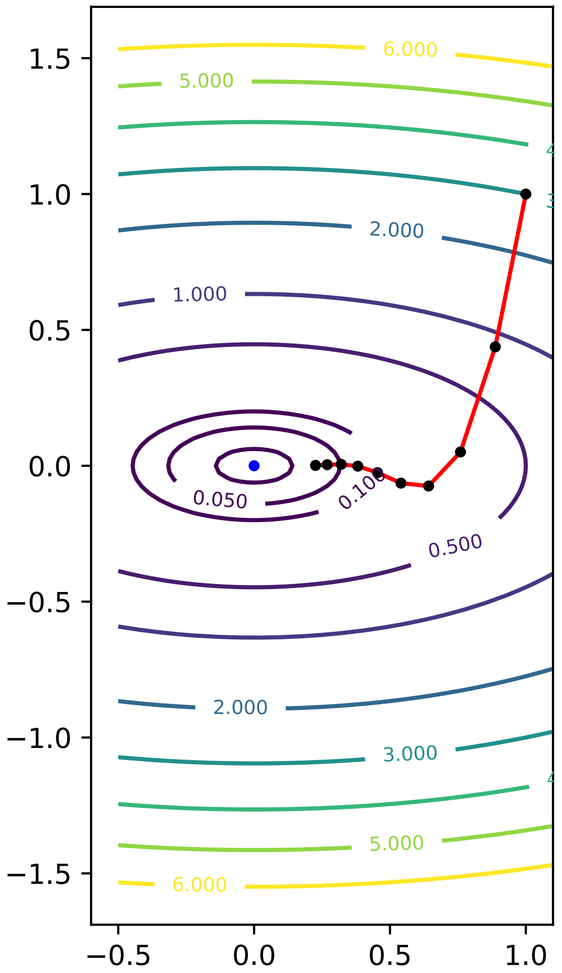


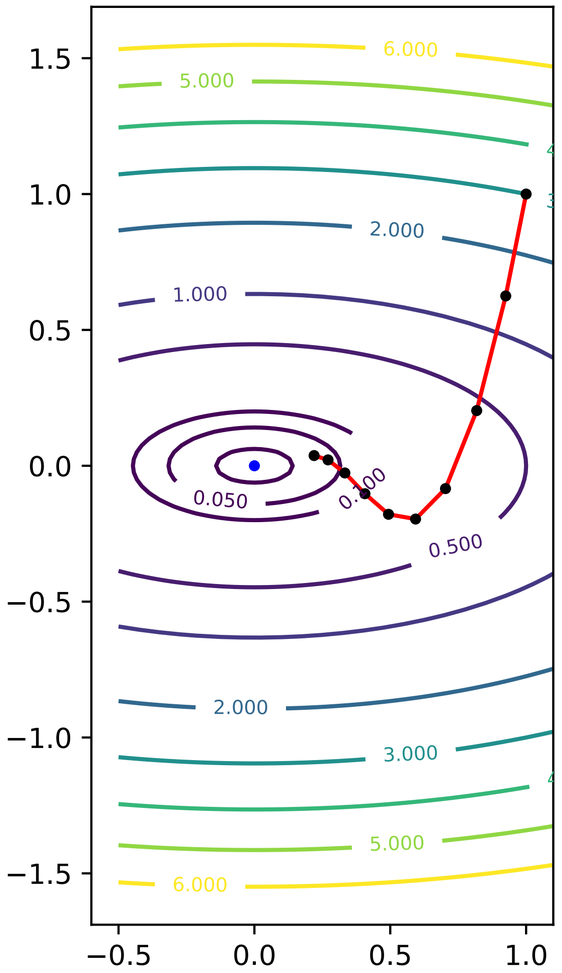
Image Source: https://distill.pub/2017/momentum/

Larger momentum = slower reaction to change in the landscape

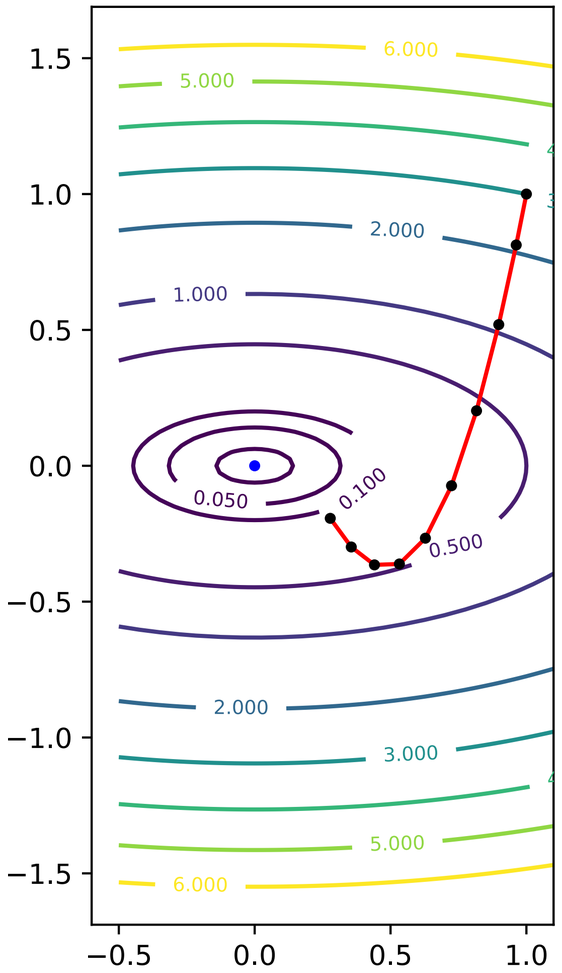
Beta 0.25



Beta 0.5



Beta 0.75



Practical Aspects of momentum

$$p_{k+1} = \hat{\beta}_k p_k + \nabla f_i(w_k)$$
$$w_{k+1} = w_k - \gamma_k p_{k+1}$$

It's basically "free lunch", in almost all situations, SGD + momentum is better than SGD, and very rarely worse!

Recommended Parameters:

Beta = **0.9** or **0.99** almost always works well. Sometimes slight gains can be had by tuning it.

The step size parameter usually needs to be decreased when the momentum parameter is increased to maintain convergence.

Explanation one: **Acceleration**

The momentum method is often conflated with Nesterov’s momentum

Regular momentum

$$p_{k+1} = \hat{\beta}_k p_k + \nabla f_i(w_k)$$
$$w_{k+1} = w_k - \gamma_k p_{k+1}$$

Nesterov’s momentum

$$p_{k+1} = \hat{\beta}_k p_k + \nabla f_i(w_k)$$
$$w_{k+1} = w_k - \gamma_k \left(\nabla f_i(w_k) + \hat{\beta}_k p_{k+1} \right)$$

`torch.optim.SGD(…, nesterov=True)`

Nesterov’s momentum, when using **VERY** carefully chosen constants, provably “**accelerates**” convergence on problems with simple structure (convex)

Regular momentum is also accelerated: but only on **quadratics**!

There is no theory to suggest this acceleration occurs when training neural networks, although a practical speedup is often observed.

Surprisingly, Nesterov’s momentum usually performs the same as regular momentum when training neural networks.

Acceleration alone does not explain momentum!

Explanation two: **Noise Smoothing**

In optimization, we usually take the last w as our estimate of the best parameters at the end.

When using **SGD**, this is suboptimal! We should actually take an **average** over past time steps (with weights ideally depending on the problem structure)

$$\bar{w}_K = \frac{1}{K} \sum_{k=1}^{K} w_k$$

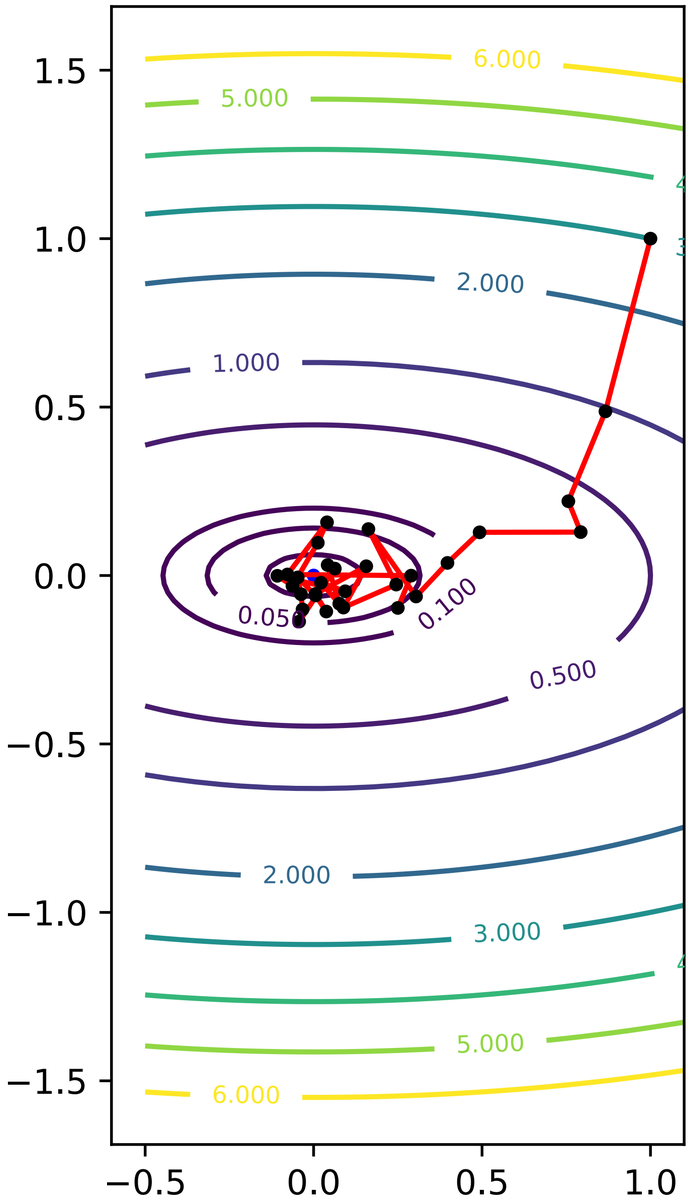
In contrast, theory suggests that SGD + Momentum requires **no averaging**, the **last value** may directly be returned!

Momentum **smooths** the noise from the stochastic gradients

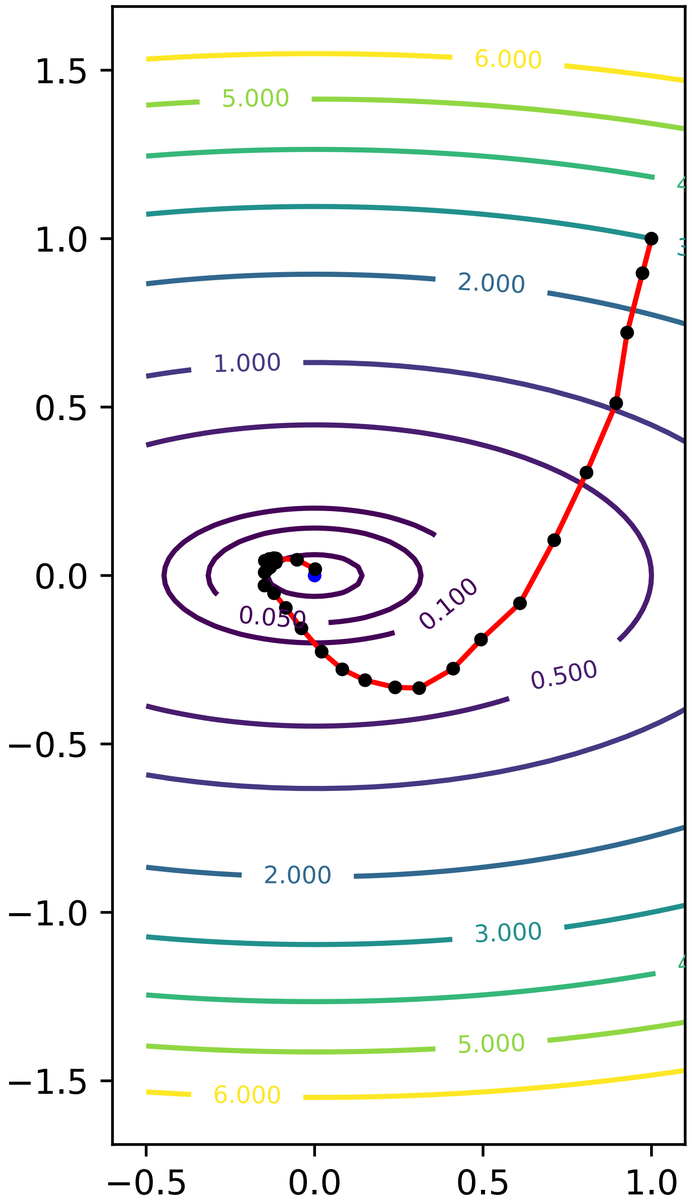
Both noise smoothing and acceleration contribute to the high performance of momentum

Quadratic with STD 1.0 noise injected into b

SGD



Momentum 0.8

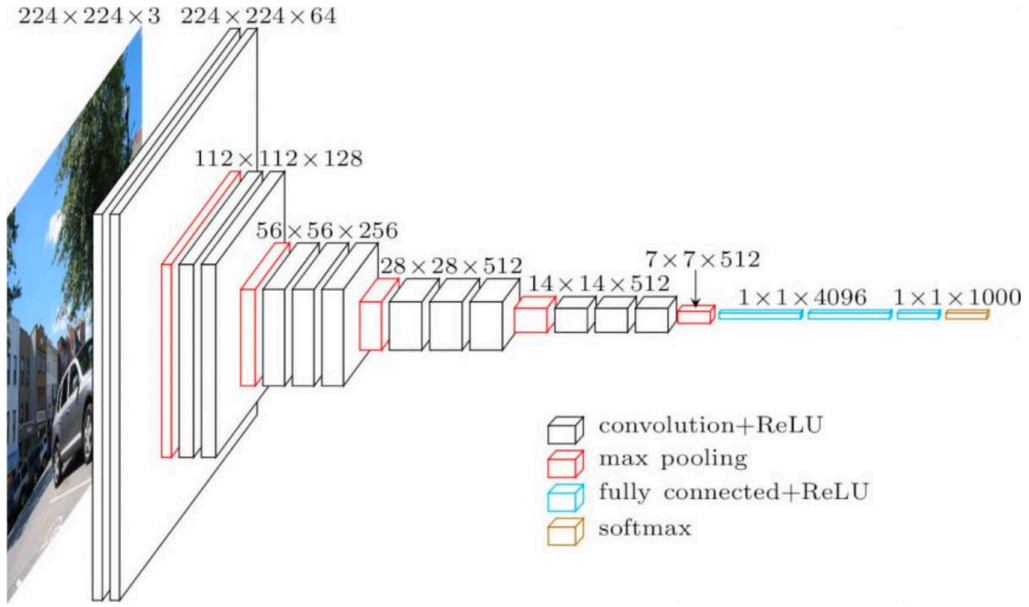


Adaptive methods

3

Adaptive methods

The magnitude of the gradients often varies highly between layers due to, so a global learning rate may not work well.



General **IDEA**: Instead of using the same learning rate for every weight in our network, **maintain an estimate of a better rate separately for each weight**.

The exact way of adapting to the learning rates varies between algorithms,
But most methods either **adapt** to the **variance** of the weights, or to the l**ocal curvature** of the problem.

RMSprop

Key **IDEA**: normalize by the **root-mean-square** of the gradient

Exponential moving average

2nd moment estimate

Notation for element-wise squaring. i.e. square each element separately

$$v_{t+1} = \alpha v_t + (1 - \alpha) \nabla f_i(w_t)^2$$

$$w_{t+1} = w_t - \gamma \frac{\nabla f_i(w_t)}{\sqrt{v_{t+1}} + \epsilon}$$

Global LR

RMS estimate + epsilon (epsilon avoids divide-by-zero)

Most adaptive methods use a moving average, it's a simple way to estimate a noisy quantity that changes over time.

If the expected value of gradient is small, this is similar to dividing by the standard deviation (i.e. whitening)

Adam: RMSprop with a kind of momentum

"Adaptive Moment Estimation"

Presented here without **bias-correction** (i.e. steady state Adam)

RMSProp

$$v_{t+1} = \alpha v_t + (1-\alpha)\,\nabla f_i(w_t)^2$$
$$w_{t+1} = w_t - \gamma \frac{\nabla f_i(w_t)}{\sqrt{v_{t+1}} + \epsilon}$$

Momentum (Exponential moving average)

2nd moment estimate (same as RMSProp)

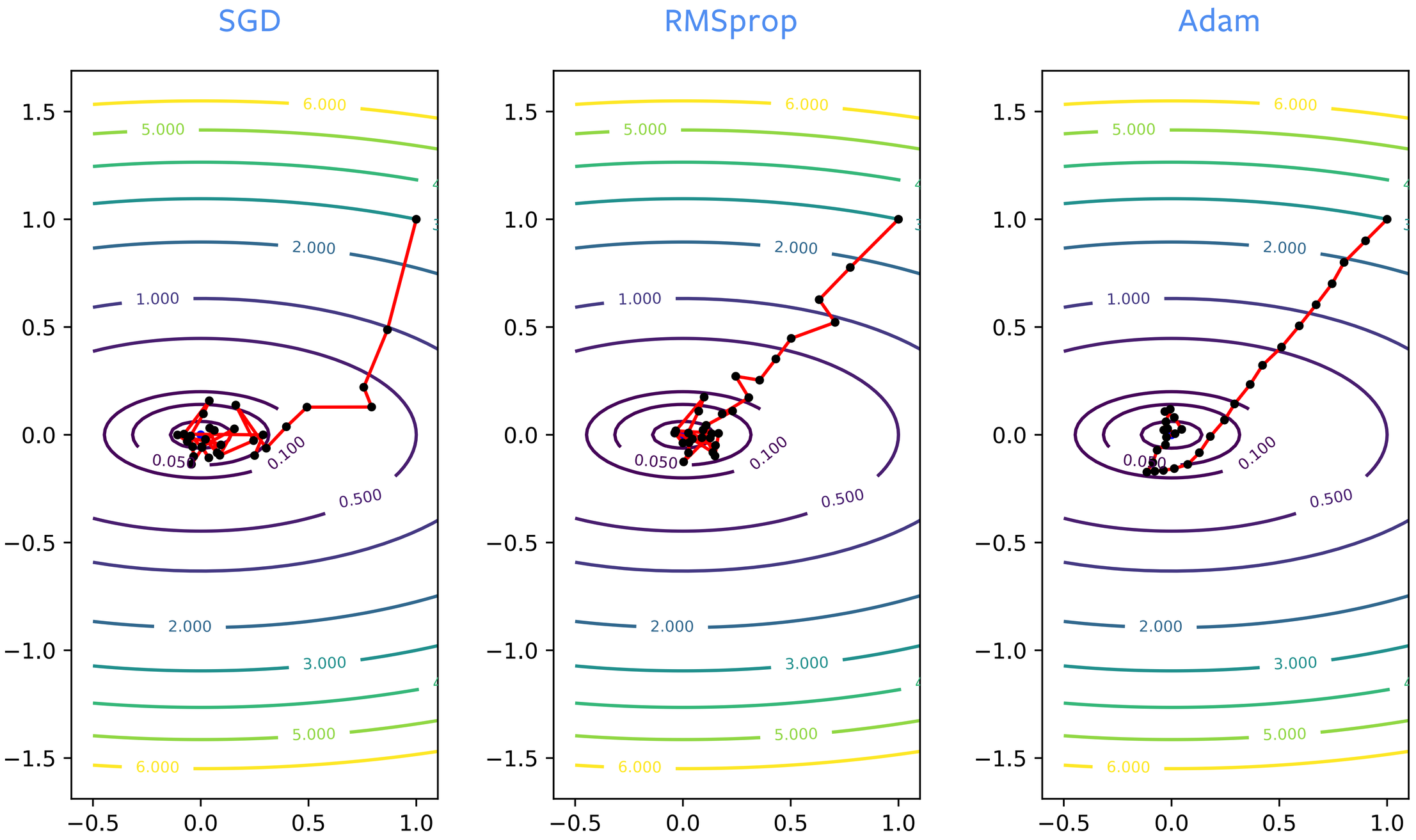
$$m_{t+1} = \beta v_t + (1-\beta)\nabla f_i(w_t)$$
$$v_{t+1} = \alpha v_t + (1-\alpha)\,\nabla f_i(w_t)^2$$
$$w_{t+1} = w_t - \gamma \frac{m_t}{\sqrt{v_{t+1}} + \epsilon}$$

Use m instead of the gradient

Just as momentum improves SGD, it improves RMSProp as well.
The exponential-moving-average method of updating momentum is equivalent to the standard form under rescaling. Nothing mysterious.

The full version of Adam has **bias-correction** as well, which just keeps the moving averages **unbiased** during early iterations. The algorithm quickly approaches the above steady state form.

Diederik, Kingma; Ba, Jimmy (2014). "Adam: A method for stochastic optimization"


SGD
RMSprop
Adam
6.000
5.000
2.000
1.000
0.500
0.100
0.050
3.000
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−0.5
0.0
0.5
1.0

Practical side

For poorly conditioned problems, Adam is often **much** better than SGD.
I recommend using Adam over RMSprop due to the clear advantages of momentum

BUT, Adam is poorly understood theoretically, and has known disadvantages:

- Does not converge at all on some simple example problems!
- Gives worse generalization error on many computer vision problems (i.e. ImageNet)
- Requires more memory than SGD
- Has 2 momentum parameters, so some tuning may be needed

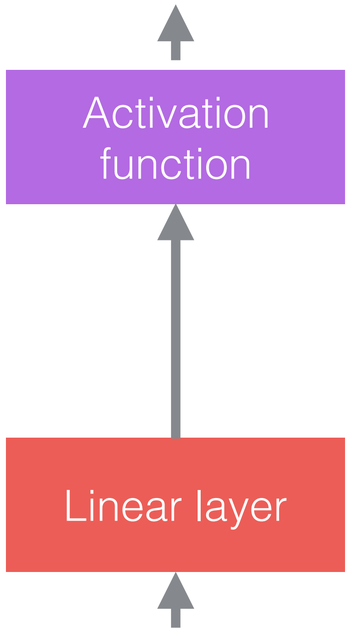
It’s a good idea to try both SGD + momentum and Adam with a sweep of different learning rates, and use whichever works better on held out data

Normalization layers:

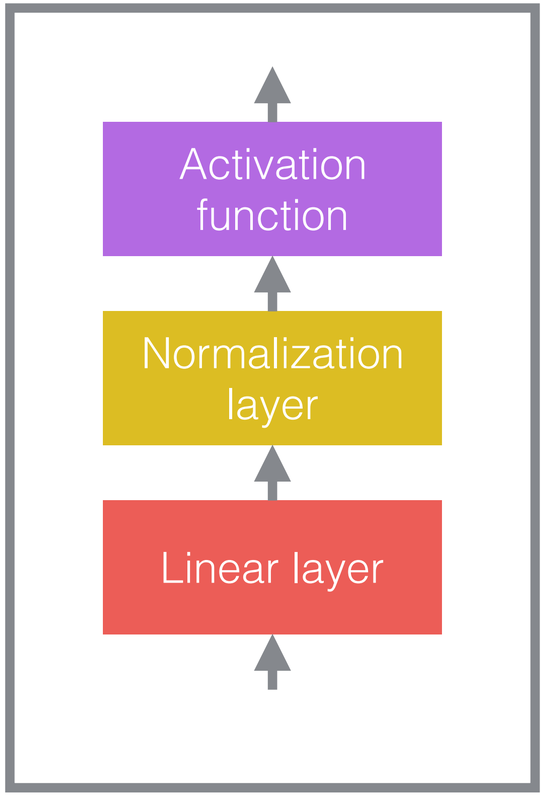
Instead of making our optimization algorithm smarter, can we make the task easier?

3

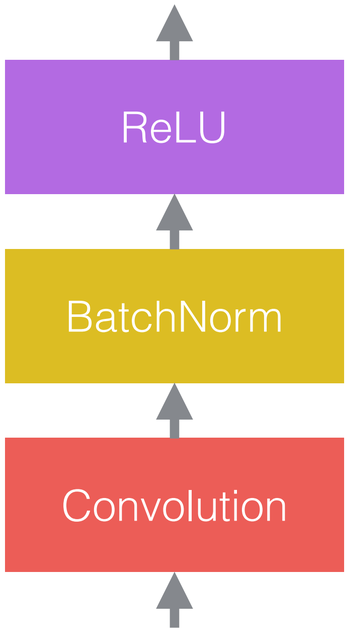
Normalization layers



They are added in-between existing layers of a neural network

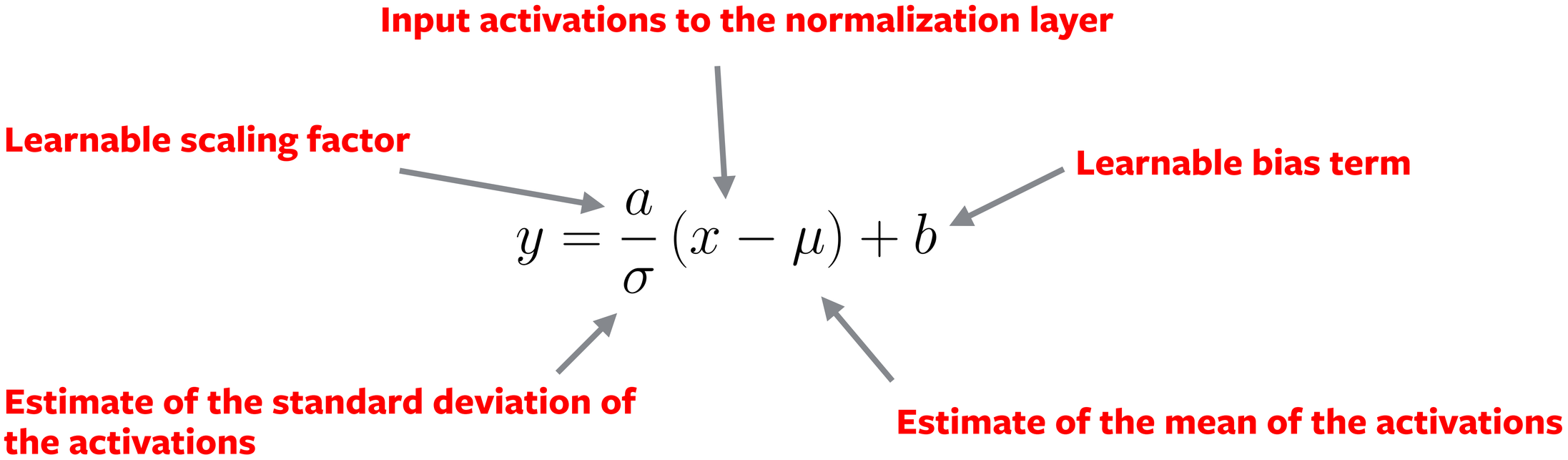


Typical image classification structure



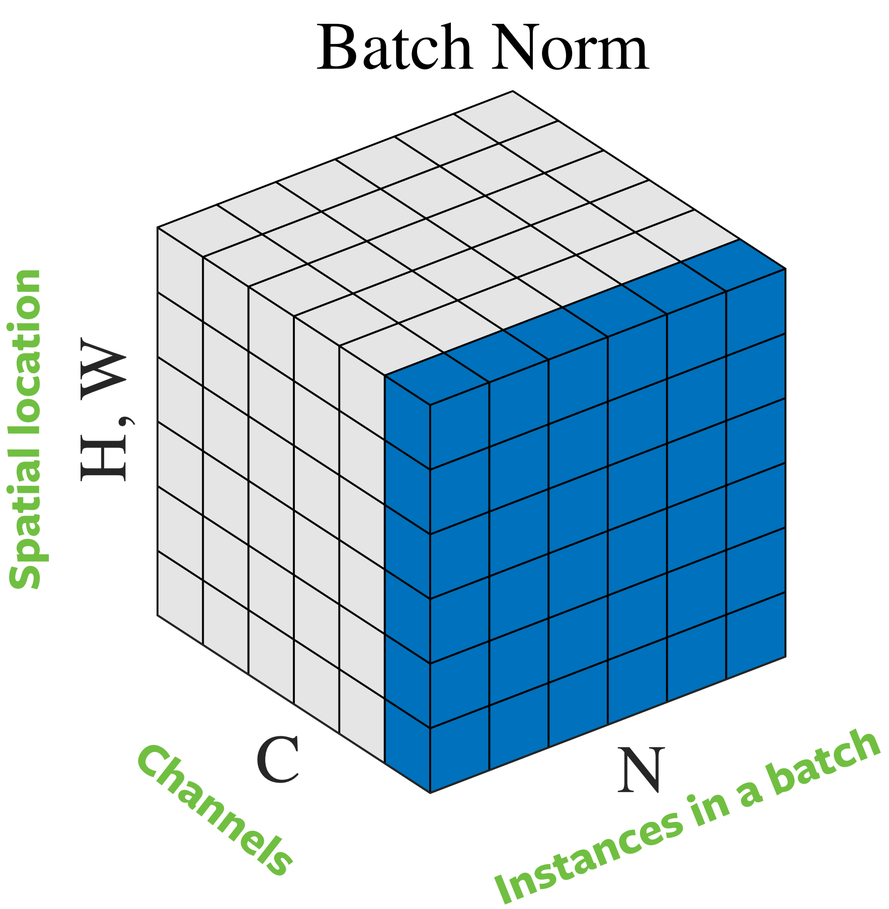
What does it mean to normalize?

Most normalization layers **whiten** activations

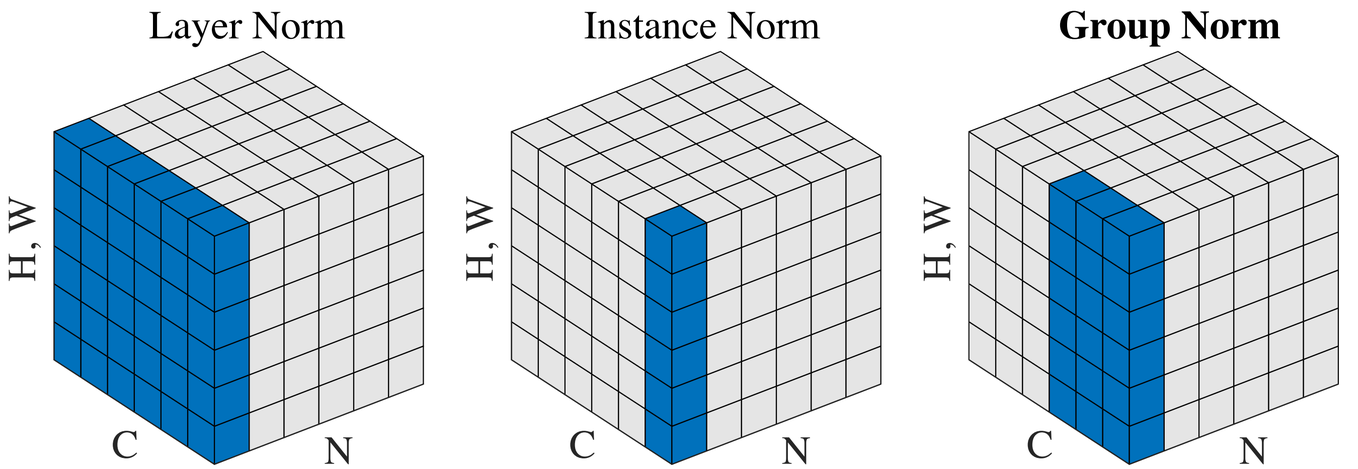


The extra a & b parameters keep the representation power of the network the same

But how do we estimate the mean and standard deviation? Does it differ across channels, spatial location, instances?



Normalize across H,W,N
i.e. across the **batch**, and
Within each separate image channel



Normalize within
H,W and a **group** of
channels

In practice group/batch norm work well for computer vision problems, and instance/layer norm are heavily used for language problems.

IMG SOURCE: https://arxiv.org/pdf/1803.08494.pdf

Why does normalization help?

This is still disputed

The original paper said that it “reduces internal covariate shift” whatever that means

As usual, we are using something we don’t fully understand.

But, it’s clearly a combination of a number of factors:

- Networks with normalization layers are easier to optimize, allowing for the use of larger learning rates. (Normalization has a **optimization** effect)
- The mean/std estimates are noisy, this extra “noise” results in better generalization in some cases (Normalization has an **regularization** effect)
- Normalization reduces sensitivity to weight initialization

Normalization lets you be more “careless”, you can combine almost any neural network building blocks together and have a good chance of training it without having to consider how poorly conditioned it might be.

For more info: https://myrtle.ai/how-to-train-your-resnet-7-batch-norm

Practical considerations

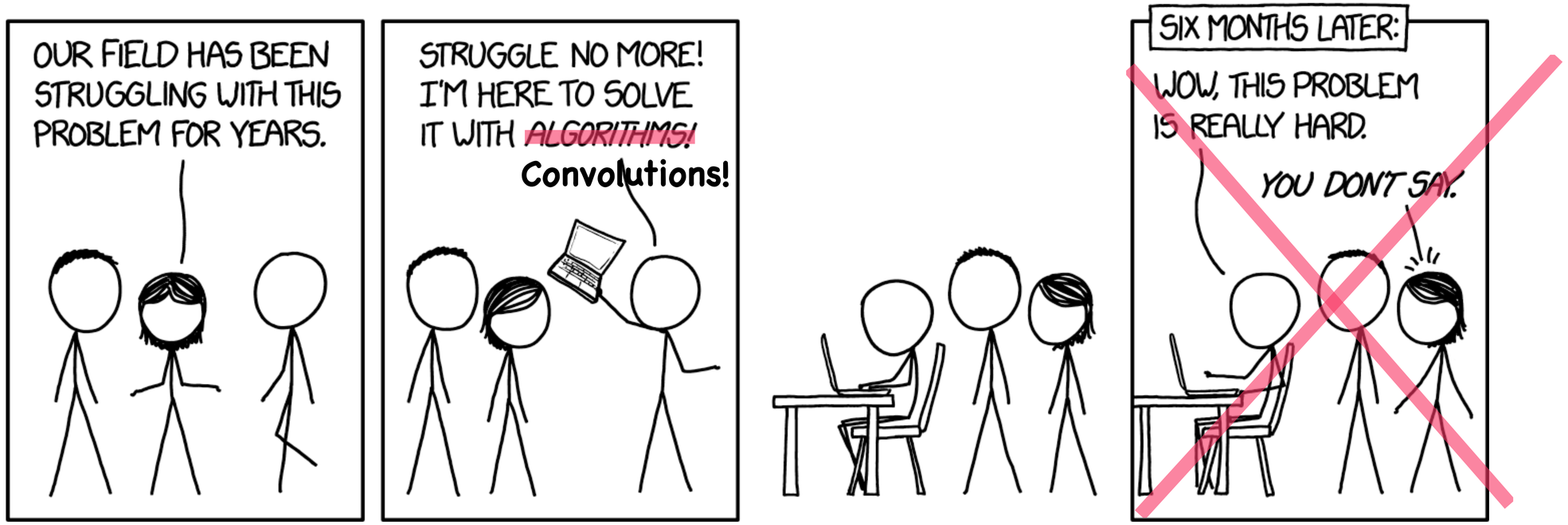
- It's important that back-propagation is done through the calculation of the mean and std, as well as the application of the normalization. It diverges otherwise.
- BatchNorm was the first developed and is the most widely known, however I **recommend using GroupNorm** instead, it's more stable, theoretically simpler, and usually works better. Try group-size 32 as a good default.
- For batch/instance Norm, the mean/std used are fixed after training, rather than recomputed every time the network is evaluated. This is not necessary for group/layer norm

`torch.nn.BatchNorm2d`

`torch.nn.GroupNorm(num_groups, num_channels, …)`

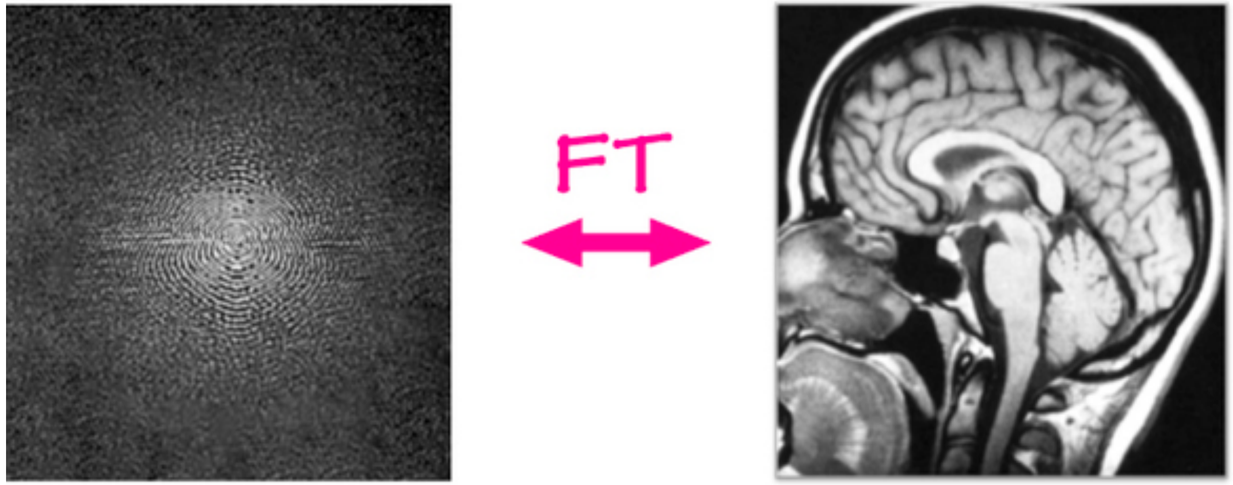
The Death of Optimization

3



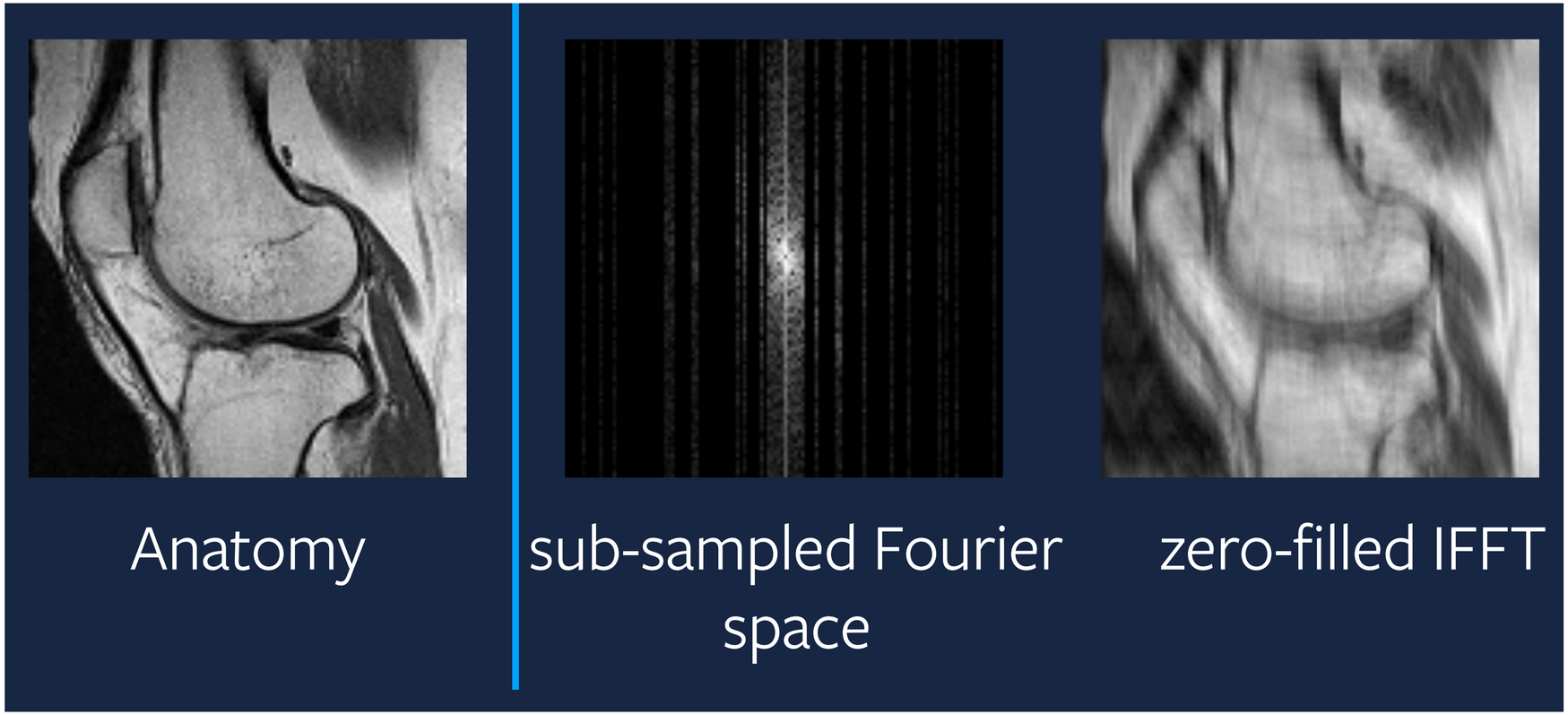
Sometimes it actually works!

Example problem:
MRI reconstruction

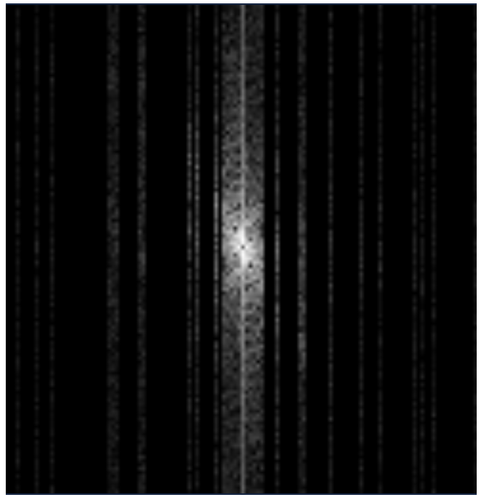


MRI machines acquire data one row or column at time of the Fourier domain representation.
The inverse Fourier transform is applied at the end to get the anatomical image

Accelerated MRI



Breakthrough of the decade: compressed sensing

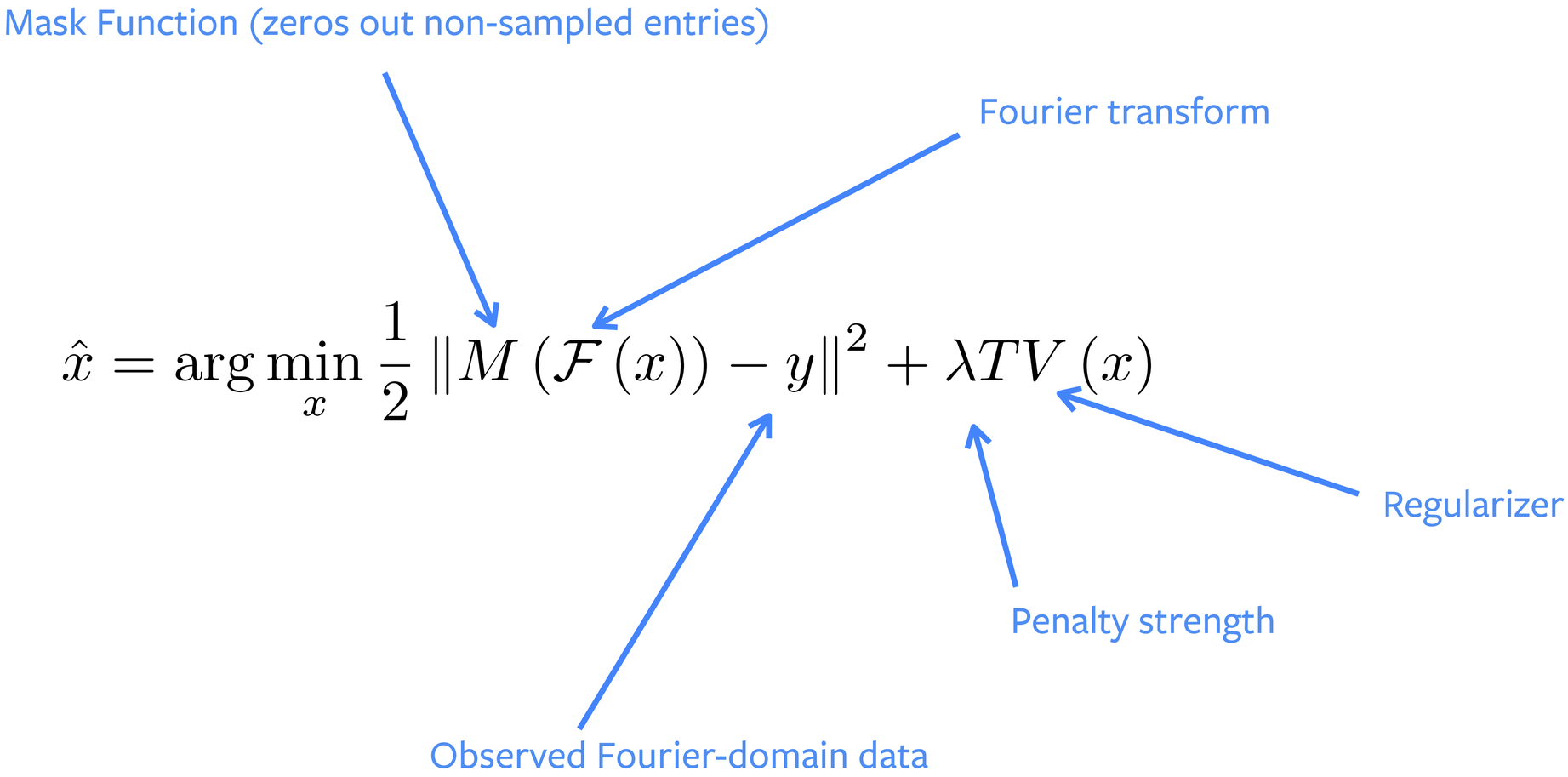


"Incoherent measurements"

When the signal you are trying to reconstruction is sparse or sparsely structured, then it's possible to reconstruct it from fewer measurements!

Stable Signal Recovery from Incomplete and Inaccurate Measurements
-Emmanuel Candes, Justin Romberg, Terence Tao (2004)

Compressed sensing MRI in 1 slide



This optimization problem must be solved for each “slice” in an MRI scan, often taking longer than the scan!

Who needs optimization?

$$\hat{x} = B(y)$$

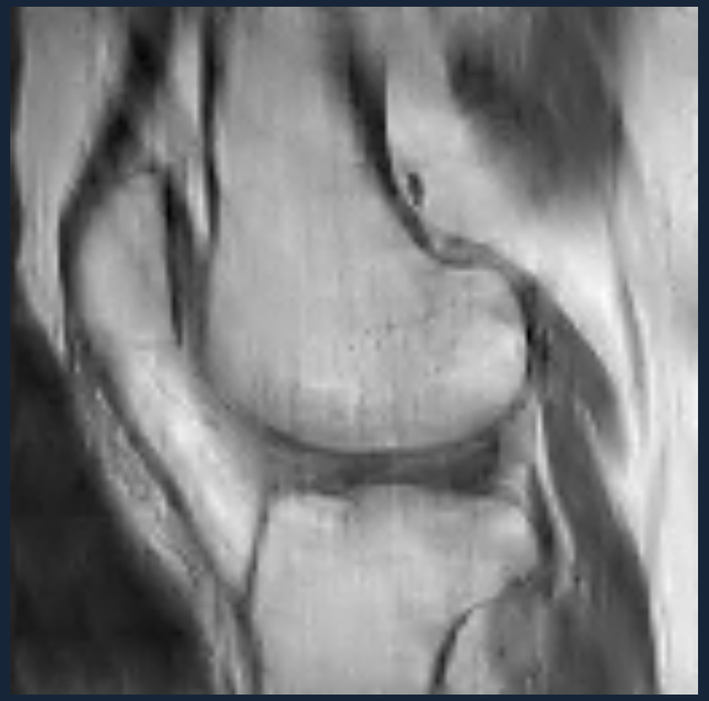
Impressively large neural network

Observed Fourier-domain data

We can just train a deep learning model to produce the required solution directly!

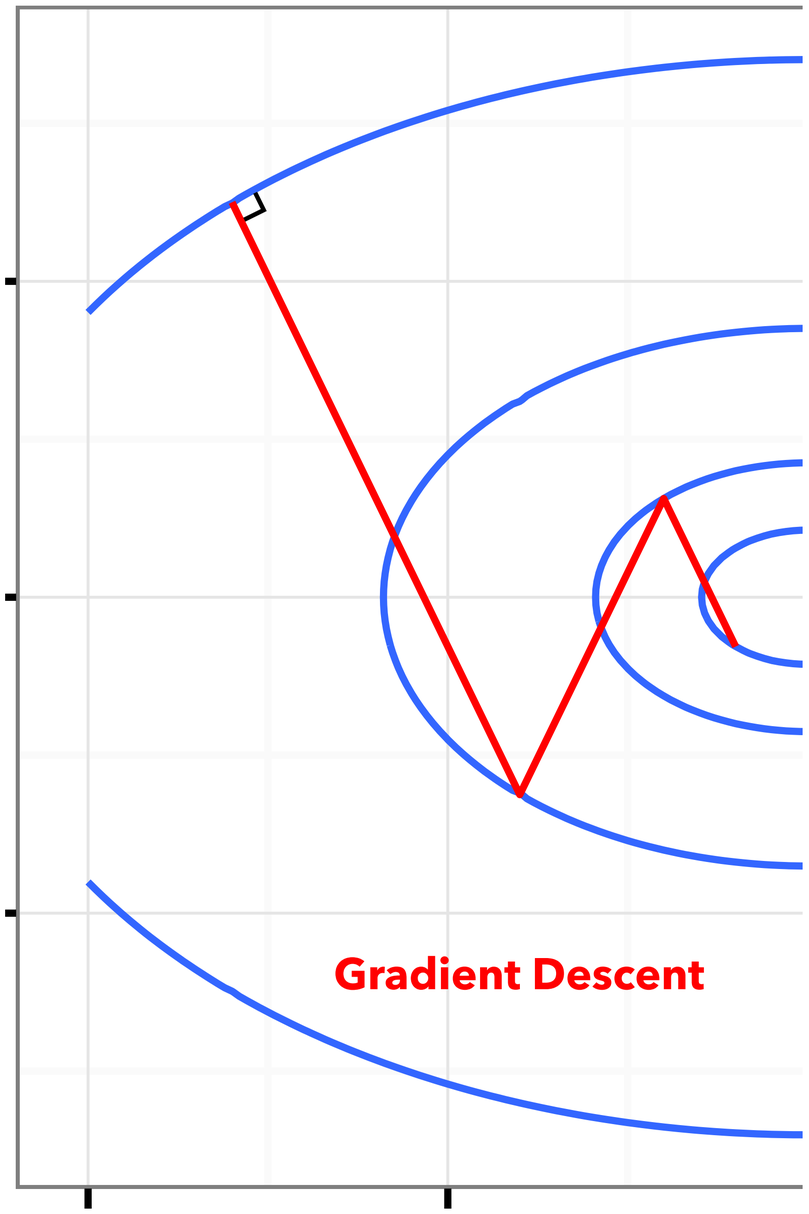
Deep learning approach

Anatomy



Classical compressed sensing reconstruction

Deep learning reconstruction (~300m parameters)



Overview

- Gradient descent & stochastic gradient descent
- Momentum
- Diagonal rescaling: RMSprop & Adam
- Normalization layers
- Solving optimization problems without optimizing
- Application: MRI reconstruction