Optimization

and the

Death of Optimization

4

1

A lecture on optimization for deep learning

Aaron Defazio

Facebook AI Research

1

2

Covering

**• Gradient descent & stochastic gradient descent**

**• Momentum**

**• Diagonal rescaling: RMSprop & Adam**

**• Normalization layers**

**• Solving optimization problems without**

**optimizing**

**• Application: MRI reconstruction**

**Gradient Descent Proximal Point Method**

−1.0 −0.5 0.0 0.5 1.0

−1.0 −0.5 0.0 0.5 1.0

1

3

Gradient descent

“*The worst optimization method in the world”*

= *x*2

= *k*(*I A*) Gradient descent 1 4

*x*ˆ= *B*(*y*) minimize

*x*ˆ= *B*(*y*)

*rf*(*w*) = *A w⇤* = *A*1

Problem

*wf*(*w*)

minimize

We denote the optimizing input

*wf*(*w*)

*wk*+1 = *wk krf*(*wk*) *f*(*w*) = 12*wT Aw bT w*

Step size

Solution (iterative)

*wk*+1 = *wk krf*(*wk*)

*kwk*+1  *w⇤k* = *kwk* (*A*

*f*(*w*) = 12*wT Aw bT w*

*rf*(*w*) = *Aw b w⇤* = *A*1*b*

Next iterate Previous iterate

Gradient of *f*

= *wk  A*(

*rf*(*w*) = *Aw b w⇤* = *A*1*b*

= *kwk  A*(

Side note: You should probably never use gradient descent directly,

*kwk*+1  *w⇤k* = *kwk* (*Awk  b*) *w⇤k*

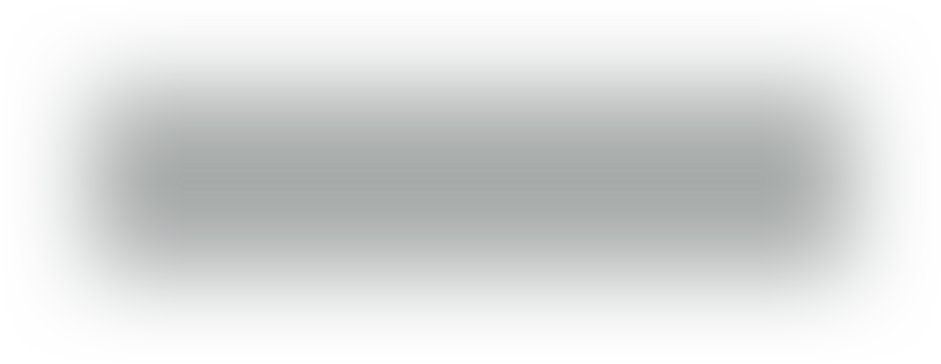
consider it a building block for other methods.

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

**PROXIMAL VS GRADIENT**

**Solution can not be in this region**

0.4

0.4

0.0

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

y

y

**Gradient Descent Proxi**

−1.0 −0.5 0.0

−1.0 −0.5 0.0

0.0

−0.4 −0.4

*fw* = 2*wAw bw*

*wk*+1 = *~~w~~k krf*(*wk*)

⇣

*f*(*w*) = 12*w>Aw b>w*

1 *µL*⌘*kwk  w⇤k*

Theory in the quadratic (positive definite) case

*rf*(*w*~~)~~ = *Aw b*

*w⇤* = *A*1*b*

*rf*(*w*) = *Aw b*

Details:

*w⇤* = *A*1*b*

4:55pm Monday,

*kwk*+1  *w⇤k* = *kwk* (*Awk  b*) *w⇤k kwk*+1  *w⇤k* = *kwk* (*Awk  b*) *w⇤k*

*kwk*+1  *w⇤k *

*wk*+1 = *wk  rf*(*wk*)

= *wk  A*(*wk  A*1*b*) *w⇤*

= *wk  A*(*wk  A*1*b*) *w⇤*

= *kwk  A*(*wk  w⇤*) *w⇤k*

= *kwk  A*(*wk  w⇤*) *w⇤k*

= *k*(*I A*) (*wk  w⇤*)*k*

= *k*(*I A*) (*wk  w⇤*)*k*

* kI Ak kwk  w⇤k*

* kI Ak kwk  w⇤k*

et *µ* be the minimum eigenvalue of *A* and *L* the max. Then the extremal eigenvalues of *I A* are 1 *µ* and 1 *L*, so: Let *µ* be the minimum eigenvalue of *A* and *L* the max. Then the extremal

1

eigenvalues of *I A* are 1 *µ* and 1 *L*, so:

*kI Ak* = max *{|*1 *µ| , |*1 *L|}*

*kI Ak* = max *{|*1 *µ| , |*1 *L|}*

we use two large a step size 1 *L* becomes negative. A standard value is = 1*/L* which gives:

⇣

1 *µL*⌘*kwk  w⇤k*

If we use two large a step size 1 *L* becomes negative. A standard value

is = 1*/L* which gives:

*kwk*+1  *w⇤k * ⇣

1 *µL*⌘*kwk  w⇤k*

*wk*+1 = *wk  rf*(*wk*)

*kwk*+1  *w⇤k *

= *wk  w⇤*

* kI Ak kwk  w⇤k*

The convergence of optimization methods

Let *µ* be the minimum eigenvalue of *A* and *L* the max. Then the extremal depends on the condition number of a

problem

eigenvalues of *I A* are 1 *µ* and 1 *L*, so:

*kI Ak* = max *{|*1 *µ| , |*1 *L|}*

Inverse of the condition number

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

⇣

*kwk*+1  *w⇤k *

1 *µL*⌘*kwk  w⇤k*

*wk*+1 = *wk  rf*(*wk*)

The condition number only formally makes sense on simple problems (“strongly convex”)

Details:

But we often talk about “**poorly conditioned**” and “**well conditioned**” problems in 4:55pm Monday,

machine learning informally

*<* opt = opt *>* opt ' 2opt

,

⇣

Typically we don’t have a good

1 *µL*⌘*kwk  w⇤k*

Let *µ* be the minimum singular value of *A* and *L* th

*kwk*+1  *w⇤k *

singu~~lar~~ values of *I A* are 1 *µ* and 1 *L*, so:

*kI Ak* = max *{|*1 *~~µ~~| , |*1 *L|}*

estimate of the learning rate!

*wk*+1 = *wk  rf*(*w*

If we use two large a step size 1 *L* becomes negative. A standard value

**Batch Gradient**

Standard practice is to try a is = 1*/L* which gives:

Details:

*wk*+1 = *wk  rf*(*wk*)

*kI Ak* = max *{|*1 *µ| , |*1

bunch of values on a log scale **There is an optimal**

If we use two large a step size 1 *L* becomes n4:55pm Monday,

is = 1*/L* which gives:

⇣

and use the one that gave the

Details:

1 *µL*⌘*kwk  w⇤k*

*<* opt

**learning rate**

best final result

4:55pm Monday,

*kwk*+1  *w⇤k *

⇣

*kwk*+1  *w⇤k *

1 *µL*⌘*kwk*

**Equal to inverse 2nd** Learning rates that are too

**derivative**

large cause “divergence”, where the function value

Details:

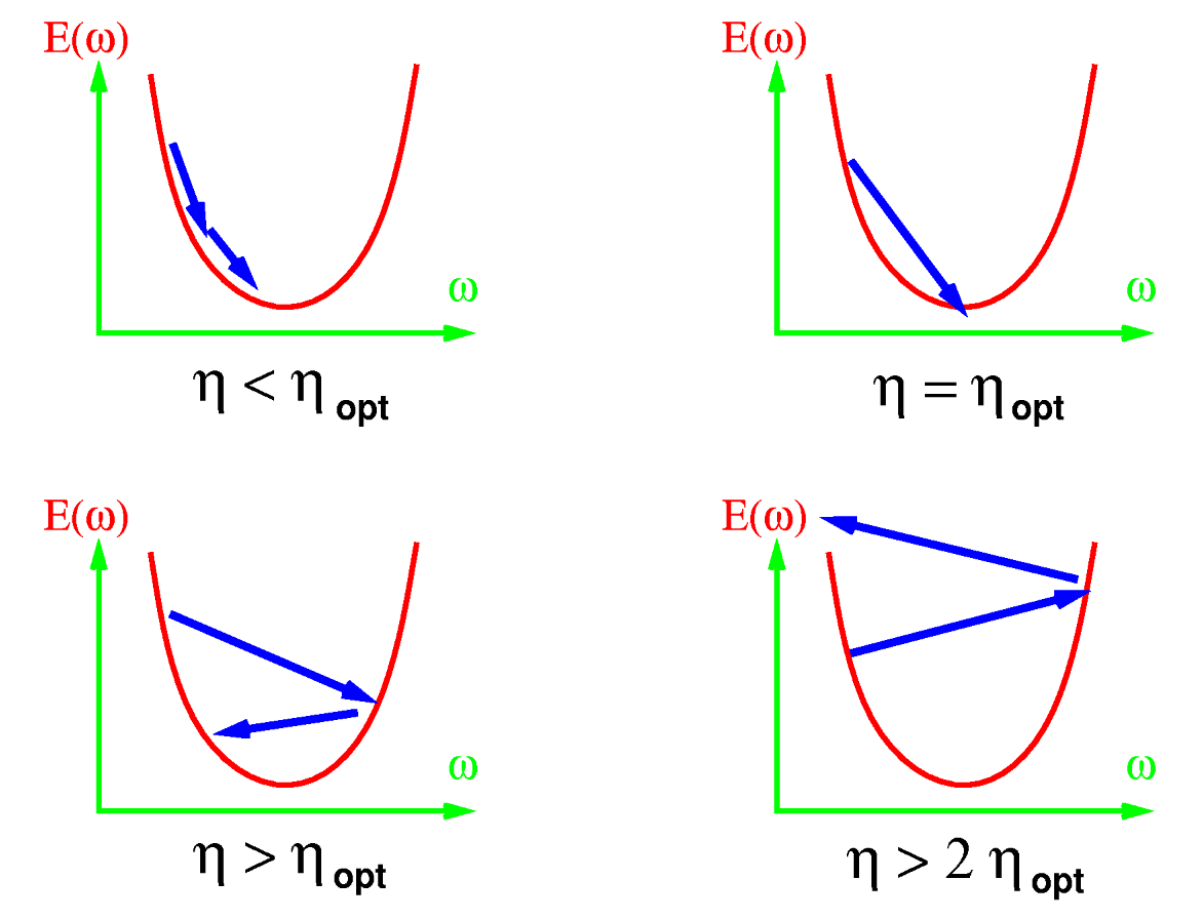
(loss) explodes

4:55pm Monday,

The optimal learning rate can change during

optimization! Often

decreasing it over time is necessary

*<* opt 

*wk*+1 = *wk  rf*(*wk*)

= opt

Details:

*>* opt

4:55pm Monday,

' 2opt

*<* opt

= opt

*>* opt

' 2opt

= opt

*wk*+1 = *wk  rf*(*wk*) *>* opt

' 2opt

*<* opt

= opt

*>* opt

' 2opt

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 Optimization3 11 Or: why a rough estimate now is better than a good estimate in a week

*x*

= opt

*wk*+1 = *wk krfi*(*wk*)

*wf*(*w*) = 1*n*X*nifi*(*w*) = 1*n*X*niL*(*xi, yi, w*)

Stochastic optimization *x*ˆ= *B*(*y*)

*>* opt

minimize

' 2opt

(*i* chosen randomly)

*wk*+1 = *wk  rfi*(*w*)

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

minimize

*fi*(*w*) = *`*(*xi, yi, w*)

scent

*wf*(*w*)

1

E[*rfi*(*wk*)] = *rf*(*wk*)

*Note: the optimization community uses f by convention, so I follow that notation here*(*w*) = 1*n*X*nifi*(*w*) = 1*n*X*n~~i~~L*(*xi, yi, w*)

*wk*+1 = *wk krf*(*wk*)

GD: *The WORST method in virtually all situations*

E [*wk*+1] = *wk krf*(*wk*)

*f*(*w*) = 12*wT Aw bT w*

*wk*+~~1~~ ~~=~~ *~~w~~~~k~~ ~~k~~~~rf~~~~i~~*~~(~~*~~w~~~~k~~*~~) +~~ *~~k~~* ~~(~~*~~w~~~~k~~  ~~w~~k*1)

*wk*+1 = *wk krfi*(*wk*) *rf*(*w*) = *Aw b*

(*i* chosen uniformly at random)

SGD: *Often the BEST method available! WHY?*

*pk*+1 = ˆ*kpk* + *rfi*(*wk*)

(*i* chosen randomly)

*w⇤* = *A*1*b*

*wk*+1 = *wk kpk*+1

mnmze

*ww* = *niiw*

SGD is GD in expectation:

(*i* chosen randomly)

Since:

*wk*+1 = *wk krfi*(*wk*)

*fi*(*w*) = *`*(*xi, yi, w*)

E[*rfi*(*wk*)] = *rf*(*wk*)

(*i* chosen randomly)

E [*wk*+1] = *wk krf*(*wk*)

*fi*(*w*) = *`*(*xi, yi, w*)

*wk*+1 = *wk krfi*(*wk*) + *k* (*wk  wk*1)

The SGD step’s expectation is just the gradient step: E[*rfi*(*wk*)] = *rf*(*wk*)

E [*wk*+1] = *wk krf*(*wk*)

*pk*+1 = ˆ*kpk* + *rfi*(*wk*)

*wk*+1 = *wk krfi*(*wk*) + *k* (*wk  wk*1)

*wk*+1 = *wk kpk*+1

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

0 * <* 1

*pk*+1 = ˆ*kpk* + *rfi*(*wk*)

*pk*+1 = ˆ*kpk* + *rfi*(*wk*)

*wk*+1 = *wk kpk*+1

ˆ

*wni*

' 2opt

*ii*

Advantages of SGD

Stochastic gradient descent

*wk*+1 = *wk krfi*(*wk*)

*wk*+1 = *wk krfi*(*wk*)

*wf*(*w*) = 1*n*X*nifi*(*w*) = 1*n*X*niL*(*xi, yi, w*)

minimize

(*i* chosen randomly)

(*i* chosen randomly)

*fi*(*w*) = *`*(*xi, yi, w*)

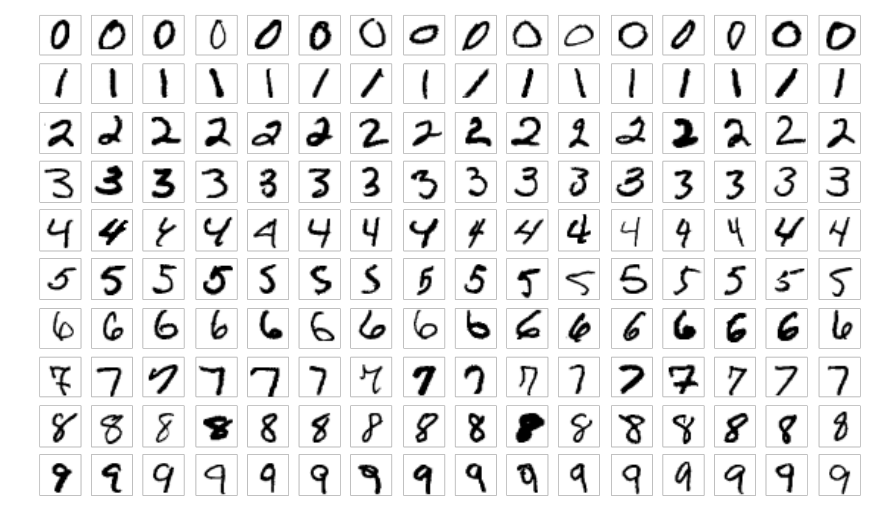
*wk*+1 = *wk  rfi*(*w*)

E[*rfi*(*wk*)] = *rf*(*wk*)

1

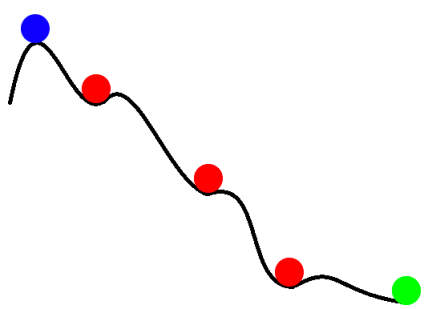
• There is redundant information across instances

1

E [*wk*+1] = *wk krf*(*wk*) 

(i.e. MNIST has thousands of near identical digits)

the noise is lower when there is higher redundancy.

*wk*+1 = *wk krfi*(*wk*) + *k* (*wk  wk*1) 

• 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

*pk*+1 = ˆ*kpk* + *rfi*(*wk*)

• 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.

*wk*+1 = *wk kpk*+1 0 * <* 1

Mini-batching

*y* =*a*

(*x µ*) + *b*

*vt*+1 = *↵vt  ↵ iwt wt*+1 = *wt rfi*(*wt*)

*~~p~~~~v~~t*+1 + *✏*

*wk*+1 = *wk k*1*|Bi|*X

ADAM (without bias correction):

*j2Bi*

Often we are able to make better use of our hardware

*rfj*(*wk*) 

*mt*+1 = *vt* + (1 )*rfi*(*wt*)

*vt*+1 = *↵vt* + (1 *↵*) *rfi*(*wt*)2

*wt*+1 = *wt mt*

*~~p~~~~v~~t*+1 + *✏*

“RMSProp with momentum generates its parameter updates using a momentum on the rescaled gradient”

by using mini batches instead of single instances

torch.addcmul(input, value=1, tensor1, tensor2, out=None)

For instance, modern graphics cards are poorly utilized

*outi* = *inputi* + *value ⇥ tensor*1*i ⇥ tensor*2*i*

if we try and calculate single instance batches.

*avg* =

q

*vt*+1  *m*2*t*+1 + *✏*

*y* =*a*

(*x µ*) + *b*

The most common distributing training technique on a cluster involves splitting a large mini batch between the

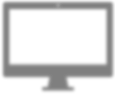
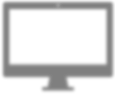
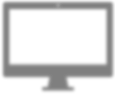
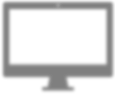
*wk*+1 = *wk k*1*|Bi|*X

*j2Bi*

*rfj*(*wk*)

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" Very efficient in practice

Momentum 3 17 Who says there is no such thing as a free lunch?

*w*= *nii*= *nii, i,*

*wk*+1 = *wk krfi*(*wk*) + *k* (*wk  wk*1)

*E*[*rfi*(*wk*)] = *rf*(*w*

Momentum

*wk*+1 = *wk krfi*(*wk*)

*E* [*wk*+1] = *wk krf*(

The most misunderstood idea in optimization?

*wk*+1 = *wk krfi*(*wk*) + *k* (

Momentum

(*i* chosen randomly)

*pk*+1 = ˆ*kpk* + *rfi*(*wk*)

*fi*(*w*) = *L*(*xi, yi, w*)

*wk*+1 = *wk kpk*+1

*E*[*rfi*(*wk*)] = *rf*(*wk*)

0 * <* 1

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

*E* [*wk*+1] = *wk krf*(*wk*)

*wk*+1 = *wk krfi*(*wk*) + *k* (*wk  wk*1)

*pk* = ˆ*kmk*1 + *rfi*(*wk*)

*wk*+1 = *wk kpk*

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

*pk* = ˆ*kmk*1 + *rfi*(*wk*)

**Key idea**: The next step becomes a combination of the

*wk*+1 = *wk kpk*

previous step’s direction and the new negative gradient

Extra term on top of SGD *pk* = ˆ*kmk*1 + *rfi*(*wk*+1 = *wk kpk*

0 * <* 1

Intuition

lyak’s momentum

*pk*+1 = ˆ*kpk* + *rfi*(*wk*)

m gradient descent, or the heavy ball algorithm was first proposed in the 60s. It combines the currentory of the previous step to accelerate the convergence of the algorithm. For example, the images be

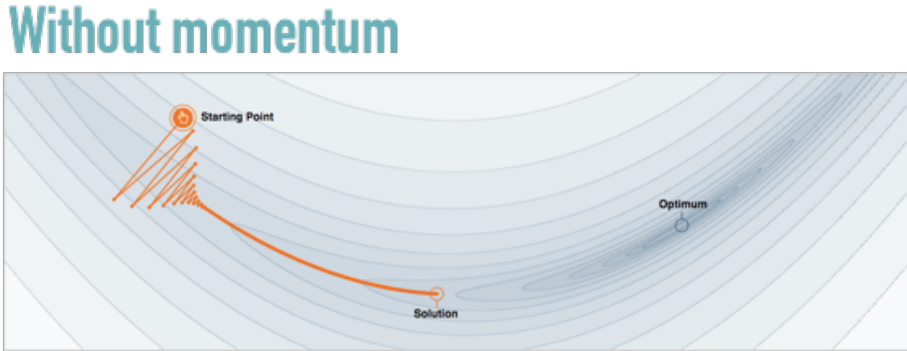
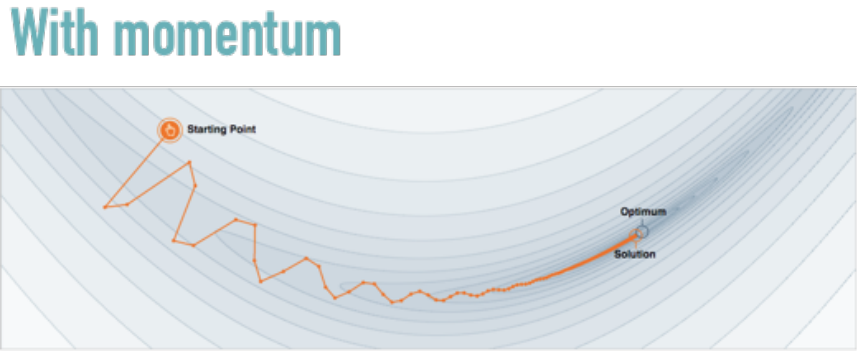
*wk*+1 = *wk kpk*+1

ke landscape, where the algorithm wants to reach the optimal point. Without momentum, gradie0 * <* 1

, whereas with momentum, we find that it converges much closer to the optimal point in the same

.

The optimization process resembles a heavy ball rolling down a hill. The ball has momentum, so *pk* = ˆ*kmk*1 + *rfi*(*wk*)

it doesn’t change direction immediately when it encounters changes to the landscape! *wk*+1 = *wk kpk*

omentum also known as the “heav ball method” introduces a “momentum” ~~term~~ *~~x~~ ~~x~~*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

*pk*+1 = ˆ*kpk* + *rfi*(*wk*) *wk*+1 = *wk kpk*+1

0 * <* 1

It’s basically “free lunch”, in almost all situations, SGD + momentum

is better than SGD, and very rarely worse! Recommended Parameters:

*pk* = ˆ*kmk*1 + *rfi*(*wk*) *wk*+1 = *wk kpk*

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.

Why “The most misunderstood idea in optimization?"

*E* [*wk*+1] = *wk krf*(*wk*)

Explanation one: **Acceleration**

*wk*+1 = *wk krfi*(*wk*) + *k* (*wk  wk*1) The momentum method is often conflated with Nesterov’s momentum

*pk*+1 = ˆ*kpk* + *rfi*(*wk*) *wk*+1 = *wk kpk*+1

0 * <* 1

Regular momentum Nesterov’s momentum

*pk*+1 = ˆ*kpk* + *rfi*(*wk*)

*pk*+1 = ˆ*kpk* + *rfi*(*wk*)

*wk*+1 = *wk kpk*+1

⇣

*wk*+1 = *wk k*

*rfi*(*wk*) + ˆ*kpk*+1⌘

0 * <* 1

torch.optim.SGD(*…, nesterov=True*)

Nesterov’s momentum, when using **VERY** carefully chosen constants, provably

*pk* = ˆ*kmk*1 + *rfi*(*wk*)

*pk*+1 = ˆ*kpk* + *rfi*(*wk* + *k*ˆ*k*1*pk*)

“**accelerates**” convergence on problems with simple structure (convex)

Regular momentum is also accelerated: but only on **quadratics**! *wk*+1 = *wk kpk*

*wk*+1 = *wk kpk*+1

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 1

when training neural networks.

Acceleration alone does not explain momentum!

⇣

*rfi*(*wk*) + ˆ*kpk*+1⌘

*wk*+1 = *wk k*

Explanation two: **Noise Smoothing**

In optimization, we usually take the last *w* as our estimate of the best parameters at the end. *pk*+1 = ˆ*kpk* + *rfi*(*wk* + *k*ˆ*k*1*pk*)

When using **SGD**, this is suboptimal! We should actually take an **average** over past time *wk*+1 = *wk kpk*+1

steps (with weights ideally depending on the problem structure)

*w*¯*K* =1*K*X*K k*=1

1

*wk*

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

SGD Momentum 0.8

Quadratic

with STD

1.0 noise

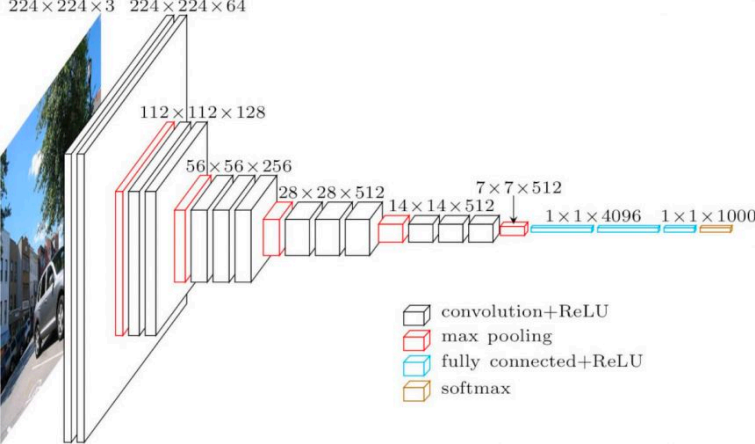
injected

into b

Adaptive methods3 25

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 estimateNotation for element-wise squaring. i.e. square each element separately**

*vt*+1 = *↵vt* + (1 *↵*) *rfi*(*wt*)2

*wt*+1 = *wt rfi*(*wt*)

*~~p~~~~v~~t*+1 + *✏*

tion):

**Global LR**

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

*mt*+1 = *vt* + (1 )*rfi*(*wt*)

Most adaptive methods use a moving average, it's a simple way to estimate a noisy quantity that changes over time. *vt*+1 = *↵vt* + (1 *↵*) *rfi*(*wt*)2

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

*vt*+1 = *↵vt* + (1 *↵*) *rfi*(*wt*)2

RMSProp

Adam: RMSprop with a kind of momentum *wt*+1 = *wt rfi*(*wt*)

RMSProp

*“Adaptive Moment Estimation”*

*~~p~~~~v~~t*+1 + *✏*

*vt*+1 = *↵vt* + (1 *↵*) *rfi*(*wt*)2

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

*wt*+1 = *wt rfi*(*wt*)

ction):

**Momentum (Exponential moving average)** ADAM (without bias correction):

*~~p~~~~v~~t*+1 + *✏*

**2nd moment estimate (same as RMSProp)**

*mt*+1 = *vt* + (1 )*rfi*(*wt*) *vt*+1 = *↵vt* + (1 *↵*) *rfi*(*wt*)2 *wt*+1 = *wt mt*

*~~p~~~~v~~t*+1 + *✏*

*mt*+1 = *vt* + (1 )*rfi*(*wt*) *vt*+1 = *↵vt* + (1 *↵*) *rfi*(*wt*)2 *wt*+1 = *wt mt*

*~~p~~~~v~~t*+1 + *✏*

**Use *m* instead of the gradient**

“RMSProp with momentum generates its parameter updates using a momentum on thetorch.addcmul(input, value=1, tensor1, tensor2, out=None)

generates its parameter updates using a momentum on the rescaled gradient” Just as momentum improves SGD, it improves RMSProp as well.

*outi* = *inputi* + *value ⇥ tensor*1*i ⇥ tensor*2*i*

The exponential-moving-average method of updating momentum is =1, tensor1, tensor2, out=None)

equivalent to the standard form under rescaling. Nothing mysterious.

*avg* =

q

*vt*+1  *m*2*t*+1 + *✏*

*ensor*1*i ⇥ tensor*2*i*

q

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

*avg* =

approaches the above steady state form.

*vt*+1  *m*2*t*+1 + *✏*

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

SGD RMSprop Adam

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: 31

3

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

Normalization layers

Activation

function

They are added in

between existing layers

of a neural network

Linear layer

Activation

function

Normalization layer

Linear layer

Typical image

classification structure

ReLU

BatchNorm

Convolution

What does it mean to normalize?

Most normalization layers **whiten** activations

generates its parameter updates using a momentum on the rescaled gradient” 1, tensor1, tensor2, out=None)

*nsor*1*i ⇥ tensor*2*i*

**Input activations to the normalization layer** q

*avg* =

*vt*+1  *m*2*t*+1 + *✏*

**Learnable bias term Learnable scaling factor**

*y* =*a*

(*x µ*) + *b*

**Estimate of the mean of the activations Estimate of the standard deviation of the 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?

**n**

**o**

**it**

**a**

**c**

**o**

**l**

**l**

**a**

**it**

**a**

**pS**

W

,

H

Batch Norm C N

W

,

H

Layer Norm C N

W

,

H

Instance Norm C N

W

,

H

**Group Norm**

C N

Normalize within

**ChannelsInstances in a batch**

Figure 2. Normalization methods. Each subplot shows a feature map tensor, with *N* as the batch axis, *C* as the channel ax

H,W and a **group** of

as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values channels

alization methods. Each subplot shows a feature map tensor, with *N* as the batc

number. ShuffleNet [65] proposes a channel shuffle oper

3.1. Formulation

xes. The pixels in blue are normalized by the same mean and variance, computed Normalize across H,W,N  ation that permutes the axes of grouped features. These

i.e. across the **batch**, and

Within each separate image channel

We first describe a general formulation o

In practice group/batch norm work well for computer vision

methods all involve dividing the channel dimension into

malization, and then present GN in this formu

problems, and instance/layer norm are heavily used for language

groups. Despite the relation to these methods, GN does *not* require group convolutions. GN is a generic layer, as we

ily of feature normalization methods, includin

problems.

and GN erform the follo~~win comutation:~~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 Optimization3 37

The death of optimization 1 38 **Convolutions!** 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

fastMRI in 3 slides

Anatomy zero-filled IFFT

sub-sampled Fourier

space

Breakthrough of the decade: compressed sensing

(a)

fastMRI in 3 slides

Anatomy zero-filled IFFT **“Incoherent measurements”** 

sub-sampled Fourier

space

**When the signal you are trying to reconstruction is sparse or sparsely structured,**

**then it’s possible to reconstruct it from fewer measurements!**

(b)

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

Compressed sensing MRI in 1 slide

Mask Function (zeros out non-sampled entries)

Fourier transform

1

*x*ˆ = arg min *x*

2 *kM* (*F* (*x*)) *yk*2 + *T V* (*x*)

Penalty strength

Observed Fourier-domain data

Regularizer

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

Who needs optimization? 1

*x*ˆ = arg min *x*

2 *kM* (*F* (*x*)) *yk*2 + *T V* (*x*) *x*ˆ= *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)

1

45

Overview

**• Gradient descent & stochastic gradient descent**

**• Momentum**

**• Diagonal rescaling: RMSprop & Adam**

**• Normalization layers**

**• Solving optimization problems without**

**optimizing**

**• Application: MRI reconstruction**

**Gradient Descent Proximal Point Method**

−1.0 −0.5 0.0 0.5 1.0

−1.0 −0.5 0.0 0.5 1.0