Deep Learning for NLP

Week 6: Training NNs — regularization & optimization

Sharid Loáiciga — May 26th, 2020

Slide credits: Hande Celikkanat and Miikka Silfverberg

Agenda

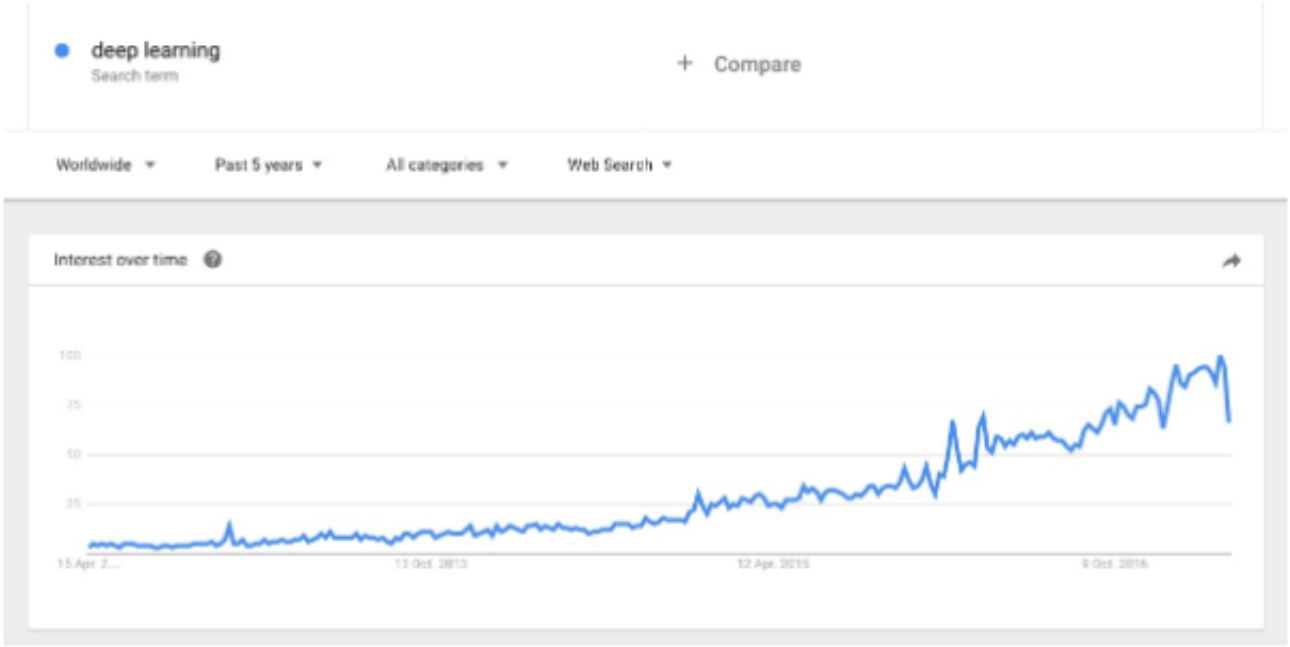
- Regularization
    - Mini-batching
    - Keep the model simple
    - Ensemble learning
    - Dropout
    - Early-stopping
    - Multi-task learning
    - Adversarial training
    - Data augmentation
- Optimization:
    - Choice of activation function
    - Batch normalization
    - Momentum
    - Adaptive learning rate
    - Parameter initialization

A bit of History

Prehistoric times – ~2000: NNs didn’t work.

~2000 – ~2010: ???

~2010 – Present: NNs work quite ok.



https://www.import.io/post/history-of-deep-learning/

Regularization: Why?

Big issue #1 in ML: How to make the model perform well on data that it didn’t see?

Keep variance in control, at the expense of increasing bias a bit: “Regularization”

From Batches to Minibatches
(Because we had to mess this up, too.)

First the jargon:

When we use a single data point at each update step, it's called **stochastic**, or **online**: Stochastic Gradient Descent

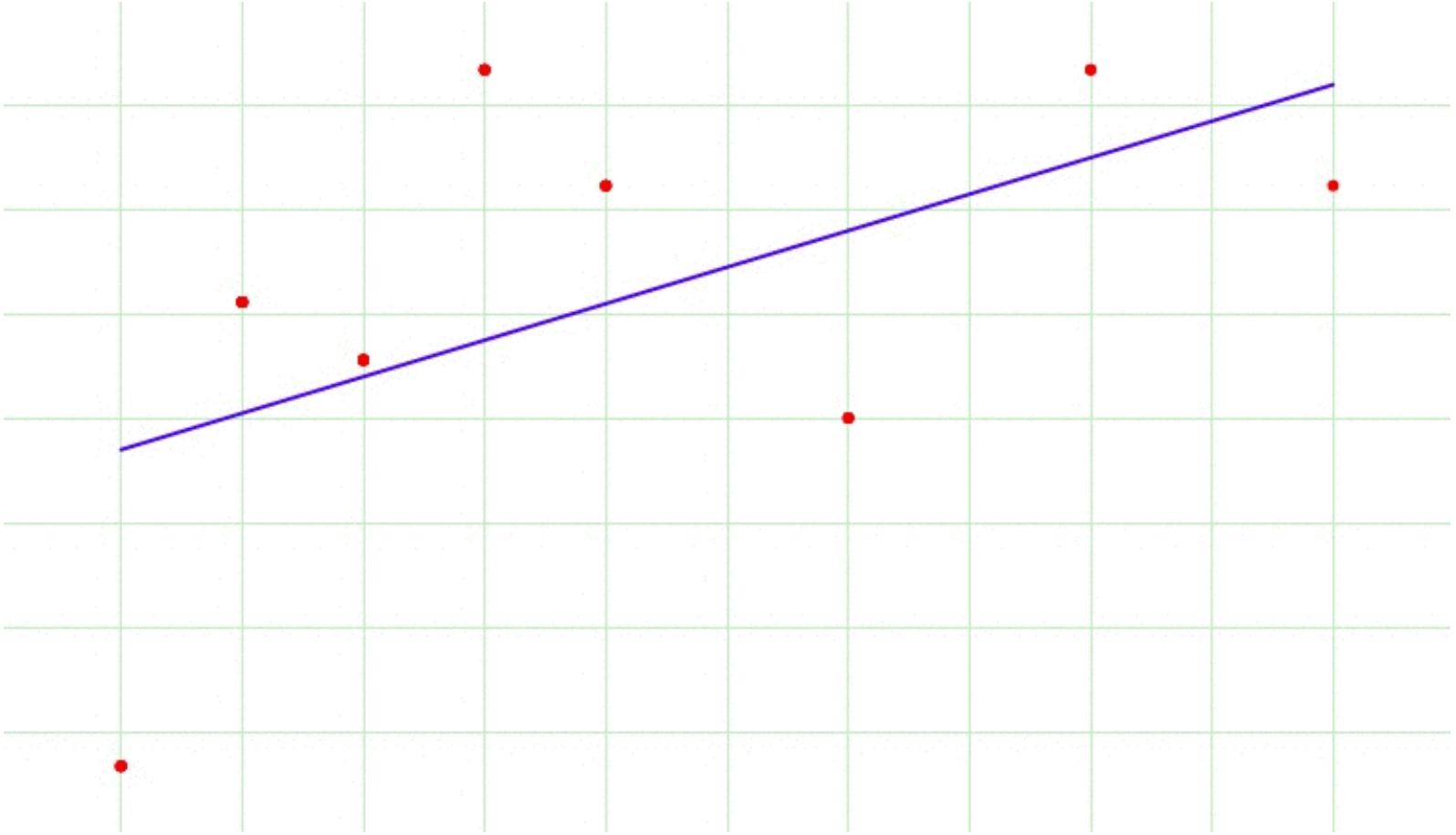
If we use ALL of the data points for each update, it's called a **batch** (from old CS jargon) or **deterministic** gradient method

But the general practice is to use something in between: which is a **minibatch**

Batching Notes
(But don’t let them hinder your explorations.)

- Larger “batches” (=read “minibatch” from now on) provide a more accurate estimate of the gradient
- Multicore architectures are generally under-utilized by extremely small batches (Generally motivates a “minimum” meaningful batch size to get any speed up)
- But then there is a “meaningful” maximum too, which is your memory size
- ! Small batches can have a regularizing effect, due to the intrinsic noise! We already got this effect in our code until now!
- Small batches require a lower learning rate in general, due to this noise.
- With bigger batches, use a higher learning rate, and also more number of epochs! (Generally done in NLP)

Oldest Trick in the Book: “Line”, “the Superstar”

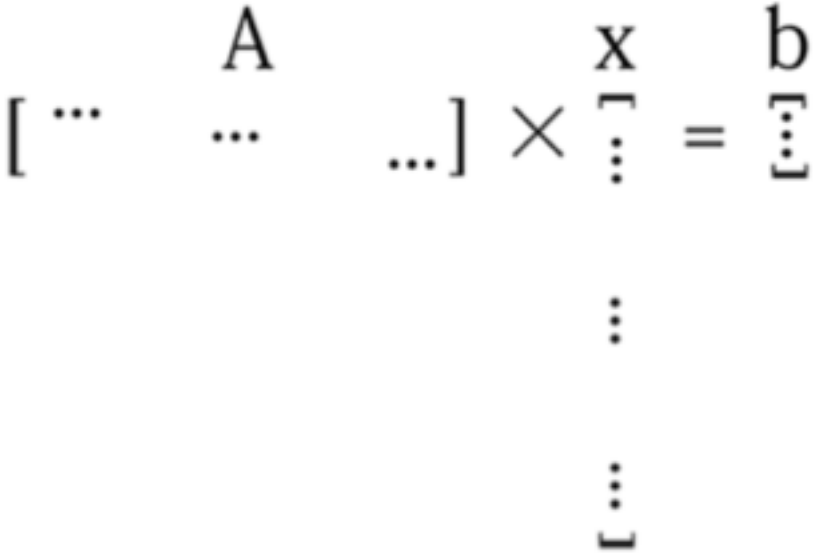


Ever wonder why the linear models do so well?

Keeping your network simple

The more parameters there are, the more easy it is for the network to overfit.

You need a number of times of data as the parameters in general. How is this a problem with deep networks?



But be careful with this one!

In practice, we never have an idea of what the true distribution looks like.

”To some extent, we are always trying to fit a square peg (the data-generating process) into a round hole (our model family).”

[1] Deep Learning Book, Goodfellow et al., pg. 225

But be careful with this one!

Then, which model is simple enough to not overfit, while also strong enough to learn?

In practice, the best model is very often a large model that has been regularized appropriately.

[1] Deep Learning Book, Goodfellow et al., pg. 225

Ensemble Learning

Idea: Combine the strengths of multiple methods!

Train several models **separately**, then have all of them **vote** on test input.

Which models to combine?

1. different learning algorithms

2. same learning algorithm trained in different ways

3. even *same learning algorithm trained the same way*

https://www.d.umn.edu/~rmaclin/cs5751/notes/Ensemble-6PerPage.pdf

Ensemble Learning

Idea: Combine the strengths of multiple methods!

Train several models **separately**, then have all of them **vote** on test input.

How to combine?

1. majority vote

2. confidence-weighted vote

3. *learning* how to combine, too

https://www.d.umn.edu/~rmaclin/cs5751/notes/Ensemble-6PerPage.pdf

Ensemble Learning

Why it works:

1. Different models will usually make **different errors**.
2. If we combine enough models, we can out-weight the erroneous model every time! (Genius, no?)

Important! Assumptions:

If we can assume classifiers are **independent in predictions** and **accuracy > 50%**, we can push accuracy arbitrarily high by combining more classifiers. (Hansen and Salamon, 1990)

https://www.d.umn.edu/~rmaclin/cs5751/notes/Ensemble-6PerPage.pdf

Ensemble Learning

To keep in mind: Ensemble Learning performs SO WELL

that it is often used as a benchmark for new algorithms in papers.

Because if your model performs reasonably well, at the cost of training time and memory, you can **always** push its accuracy up by ensembling by training multiple copies of it.

The New Hotshot in Town: Dropout “the Nutcase”

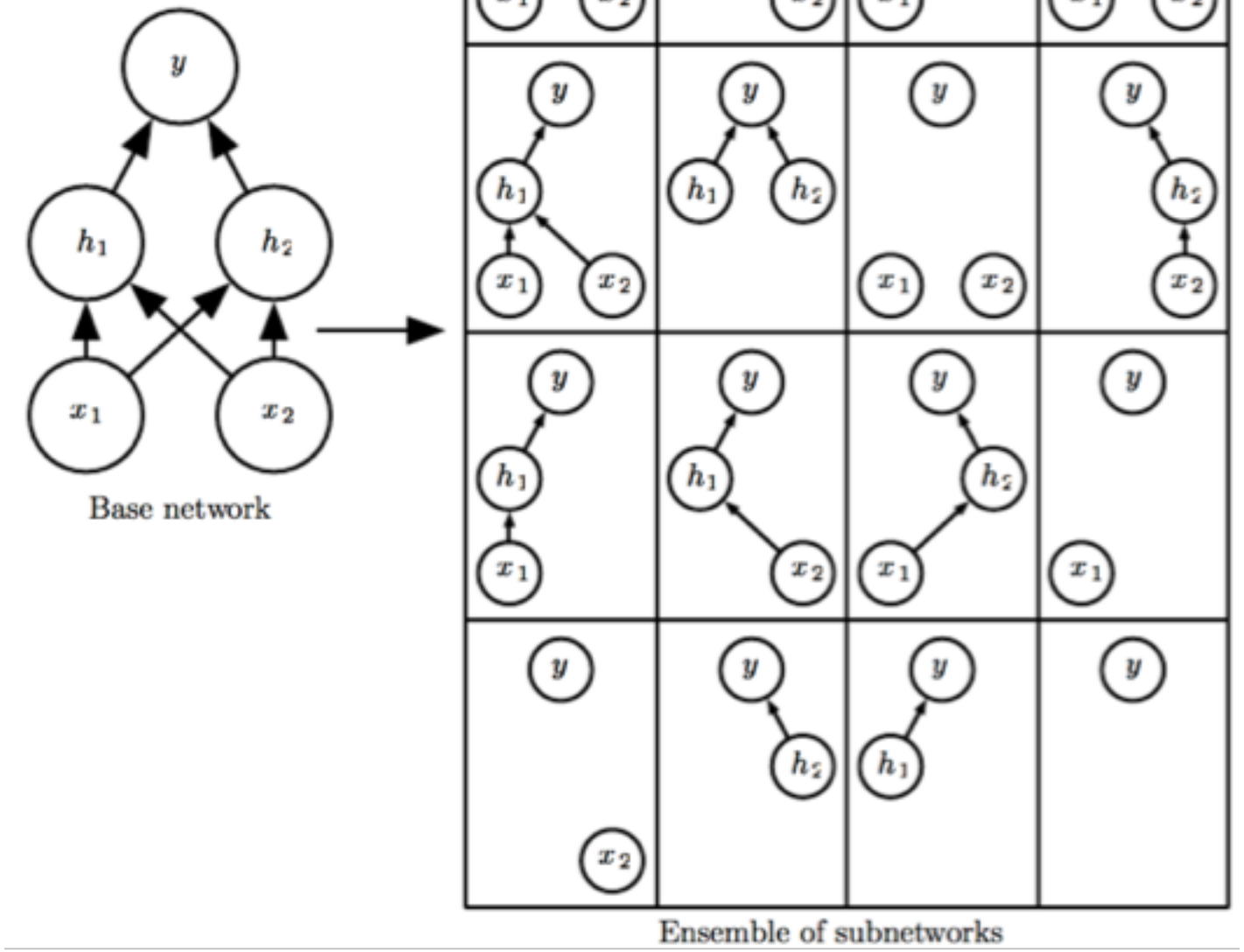
Now this is where things get crazy.

This is a way of practical ensemble learning for NNs.

Because we don’t have time to train so many models really.

Dropout is an **inexpensive approximation** to training and evaluating an ensemble of exponentially-many networks.

Every time we load a minibatch



We can “remove” a unit by multiplying its output by 0.

Hyperparam!

Typical “include” probs:
Input Unit: 0.8
Hidden Unit: 0.5

Deep Learning Book, Goodfellow et al., pg. 252

Dropout Ju-jutsu: How does it work?

Dropout Ju-jutsu: How does it work?

We “drop” randomly in training time. At every minibatch, we apply an independent filter.

This prevents the neurons from placing “*too much trust*” in one single input, or one single other neuron’s activation.

Since it may have to do without it, too.

This prevents overfitting to some “seemingly relevant” inputs.

The model needs to look deeper, and require “more robust” assurances.

Dropout Magic

How does this work *exactly*?

We are not sure :)

Empirically:

It REALLY is MORE EFFECTIVE

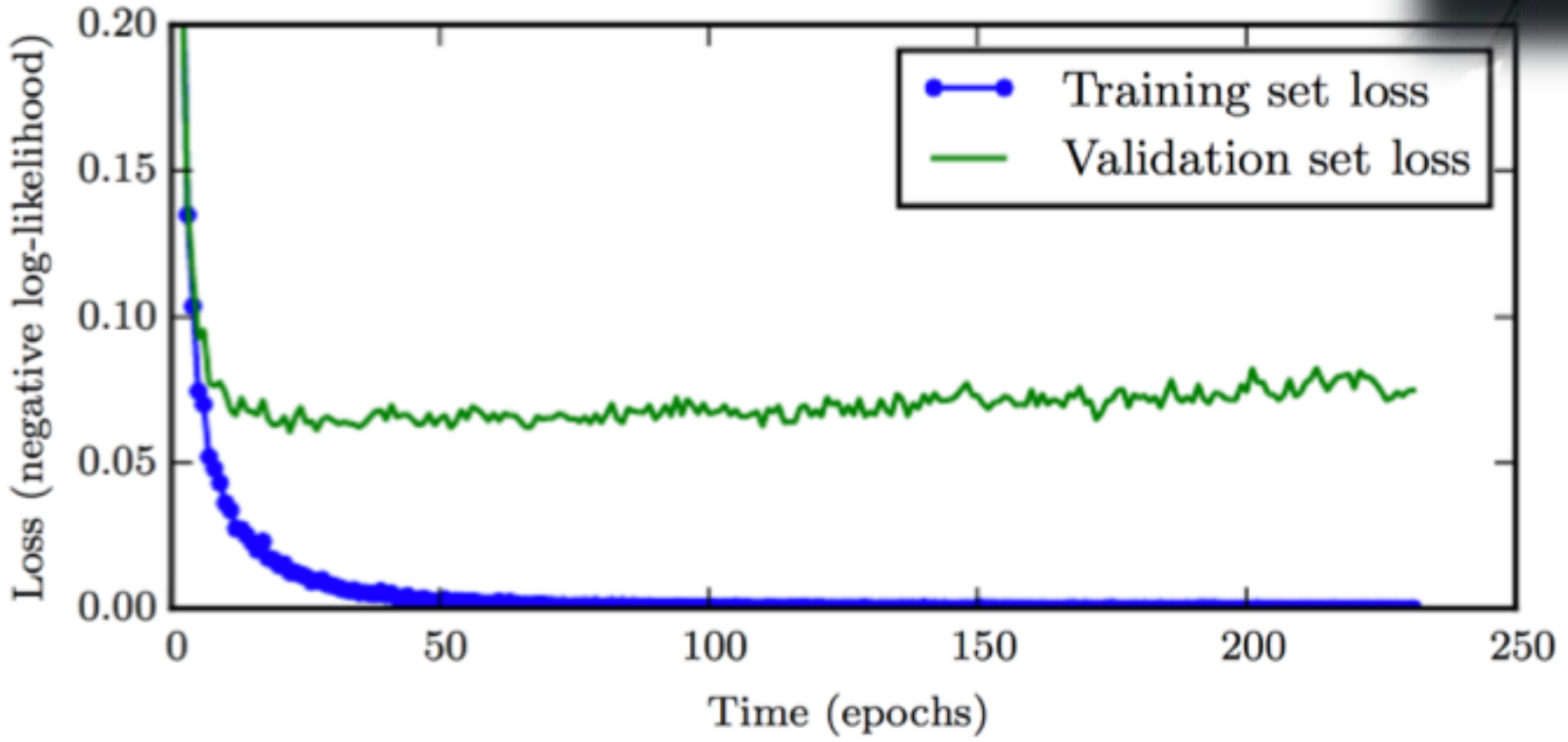
than other regularizers.

And idea makes sense.

Theoretically tho:

Don’t have a formula on why it is so good:)

Seriously??
Early-stopping “the I-swear-I-thought-of-this-one”

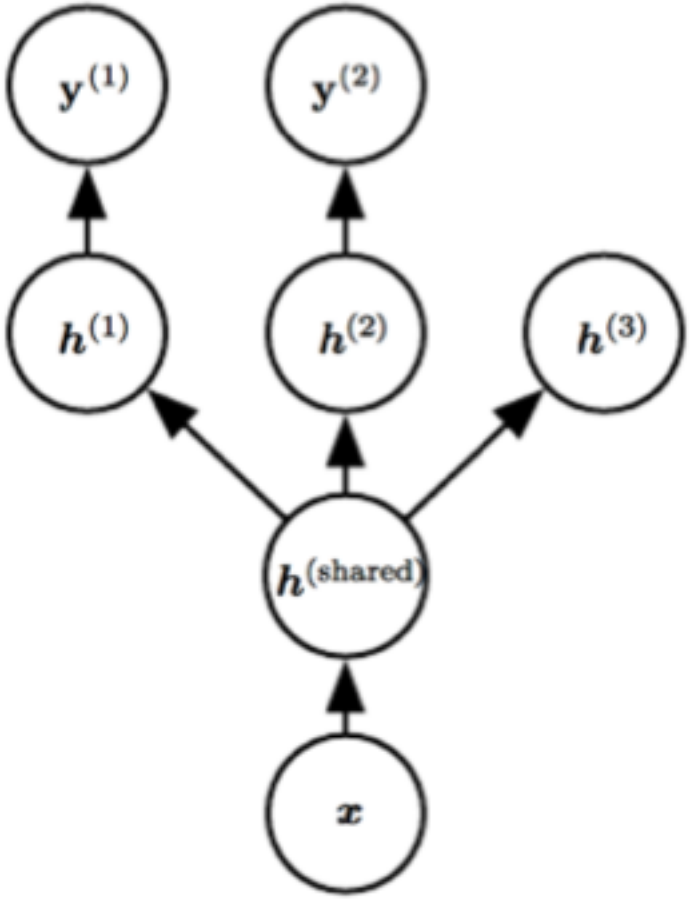


Because why not learn it all: Multi-Task Learning “the Valedictorian”

Problem: I don’t want my model to memorize the perfect solution on the training data.

Genius Solution: So I’ll just confuse it by making it have to solve many problems at once!

Because why not learn it all: Multi-Task Learning “the Valedictorian”



Problem: I don’t want my model to memorize the perfect solution on the training data.

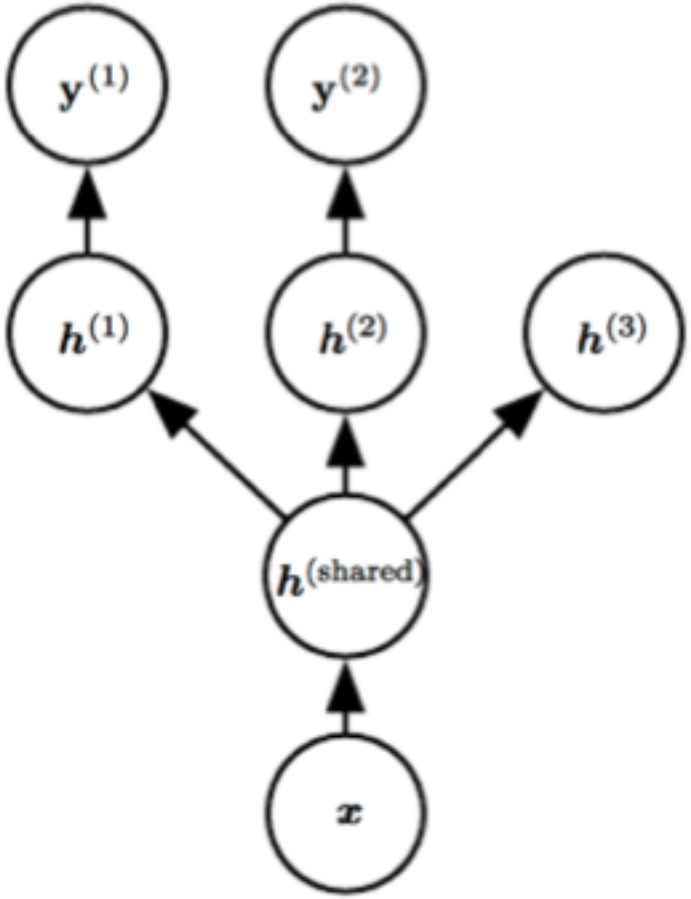
Genius Solution: So I’ll just confuse it by making it have to solve many problems at once!

How it works:

By having **one basic set** of parameters (shared between tasks!)

and **another set fine-tuned** to each task.

Because why not learn it all: Multi-Task Learning “the Valedictorian”



Problem: I don’t want my model to memorize the perfect solution on the training data.

Genius Solution: So I’ll just confuse it by making it have to solve many problems at once!

Why it works:

1. Model **has to reuse** shared params, so they better be generic (useful) ones!
2. More data for training by pooling in from all tasks.

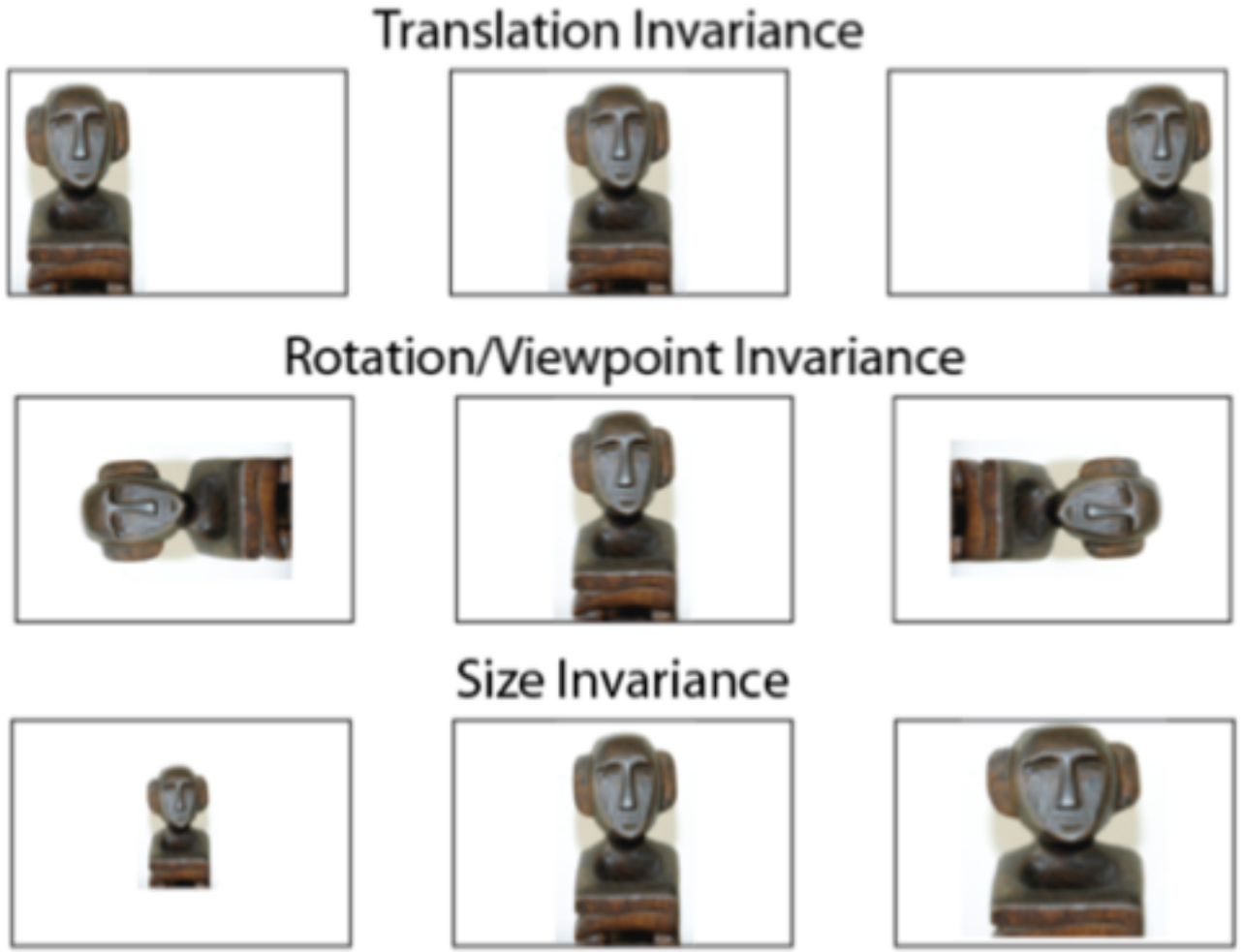
Parameter Sharing “the All or None!”

We force sets of parameters to be **equal** to each other at each step of learning.

So, either all of them move to the same direction (read: learn), or none of them.

Parameter Sharing

But why?

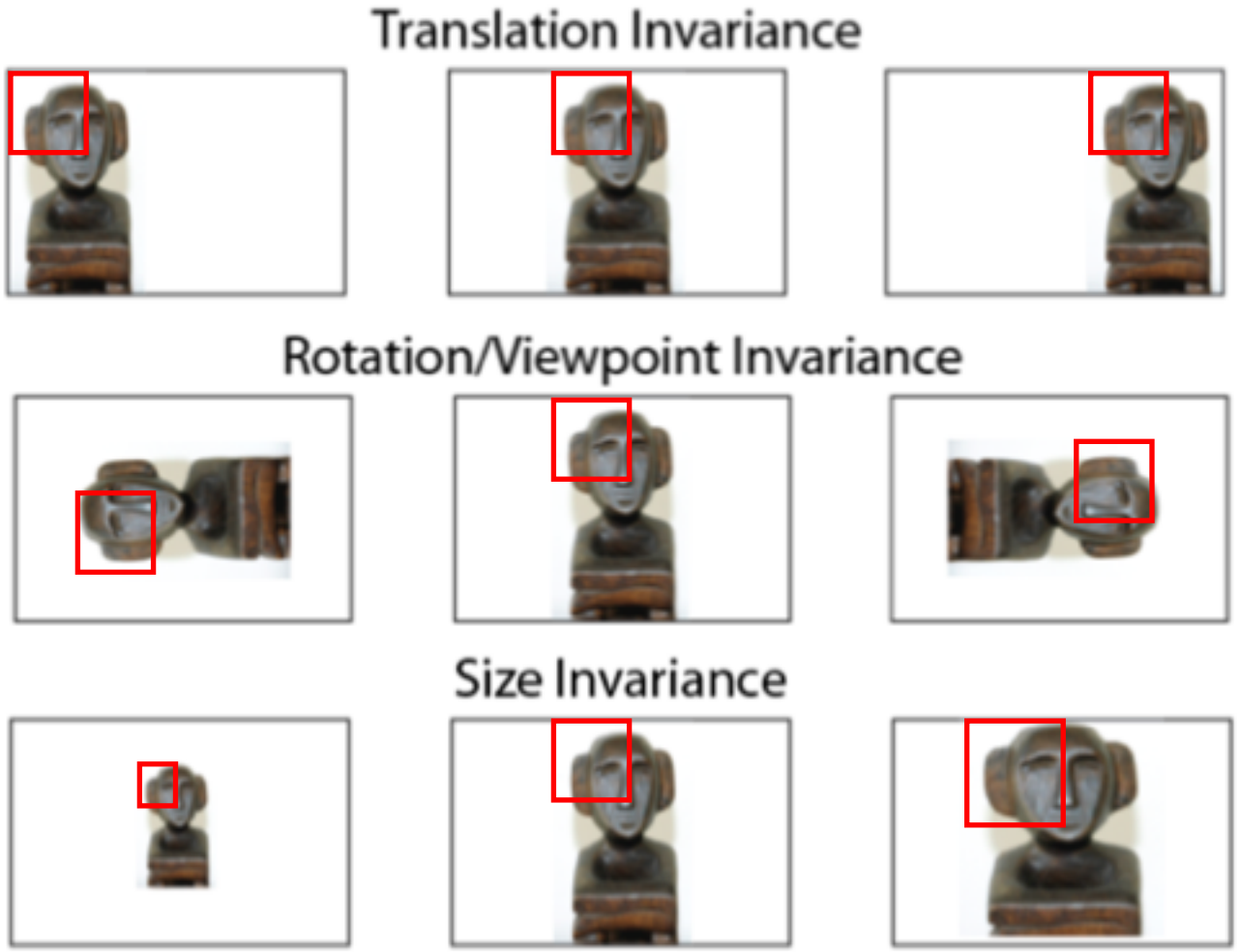


https://stats.stackexchange.com/questions/208936/what-is-translation-invariance-in-computer-vision-and-convolutional-neural-netwo

Parameter Sharing

This red square has to detect the same input everywhere.

“Invariance” is a big problem with images.

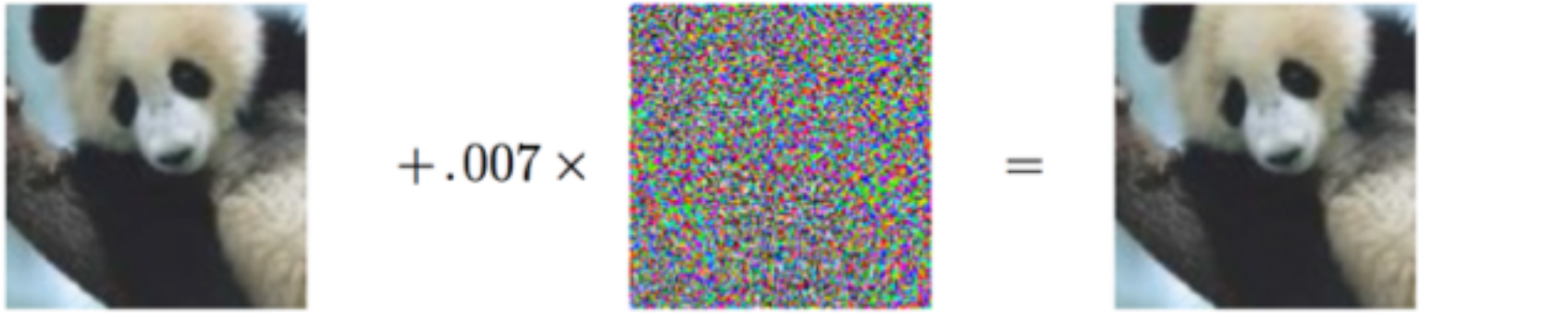


Hence Convolutional Neural Networks (CNNs) are the most famous example!

https://stats.stackexchange.com/questions/208936/what-is-translation-invariance-in-computer-vision-and-convolutional-neural-netwo

Playing the (advocate of the) Devil: Adversarial Training “the Psychopath”

Q: Are we REALLY learning what we think we are learning?



$\boldsymbol{x}$

$y =$“panda”
w/ 57.7%
confidence

$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

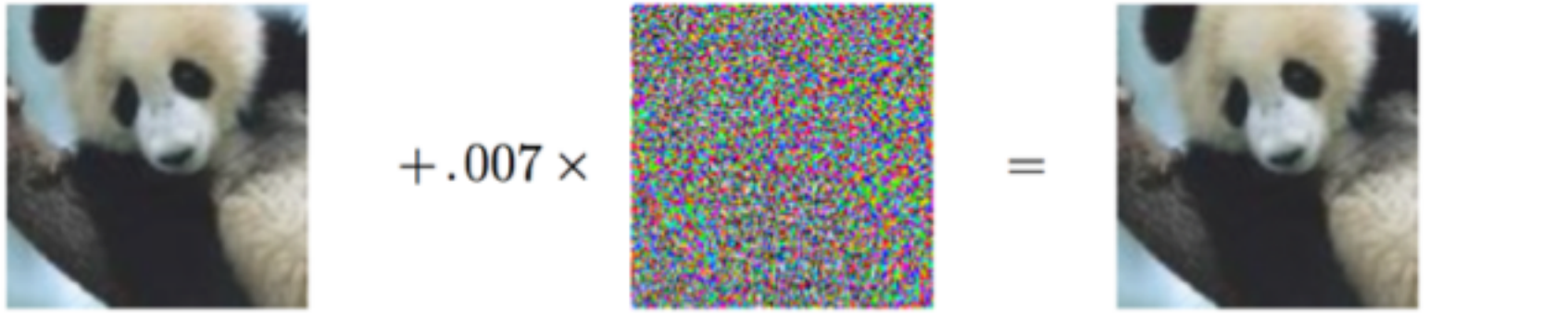
“nematode”
w/ 8.2%
confidence

$\boldsymbol{x} +$
$\epsilon\,\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
“gibbon”
w/ 99.3%
confidence

Playing the (advocate of the) Devil: Adversarial Training “the Psychopath”

Wait, but how? Here’s the trick:

The “panda-that-is-not-a-panda” is NOT a coincidence!
It was intentionally constructed!



$\boldsymbol{x}$

$y =$“panda”

$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

“nematode”

$\boldsymbol{x} + \epsilon\,\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

“gibbon”

Playing the (advocate of the) Devil: Adversarial Training “the Psychopath”

Wait, but, how? Here’s the trick:

The “panda-that-is-not-a-panda” is NOT a coincidence!
It was intentionally constructed!

By **optimizing** for an input x' very close to input $x = panda$, but for which the model output is very different.

Basically, it’s a maniac teacher that tries to cheat the student. (Upper right corner.)

Adversarial training: Training on “adversarially” perturbed examples.

Data Augmentation “the Shady”

Problem: It’s impossible to find any more data.

Solution: The unbearable lightness of being able to fake data.

Examples:

- Invariant transformations of your image, or speech, or etc.

Data Augmentation “the Shady”

Problem: It’s impossible to find any more data.

Solution: The unbearable lightness of being able to fake data.

Examples:

- Invariant transformations of your image, or speech, or etc.
- **Injecting noise** in your data to create noisy forms of the original examples. Small random noise should be both **solvable**, and **beneficial** (in forcing robustness).

Optimization: Why?

Big issue #2 in ML: How to make the model perform well at all really?

First, we will see the challenges, especially for NNs.

[Inner Thought]

Heh, how serious can an optimization problem be, to render NNs practically NOT WORKING for decades?

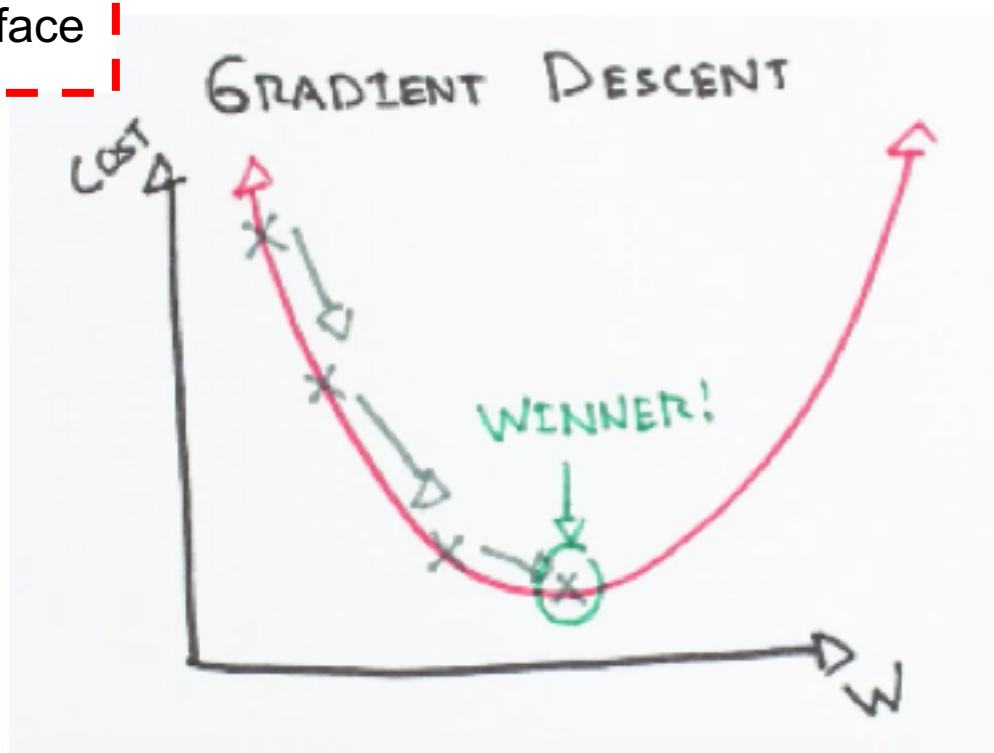
Are they serious?

Well, we will see that it’s almost a MIRACLE that NNs are working finally, against all these odds!

Challenge #1: Plateaus and other ne'er-do-well minima

You can run but you cannot escape.

Nice convex error surface

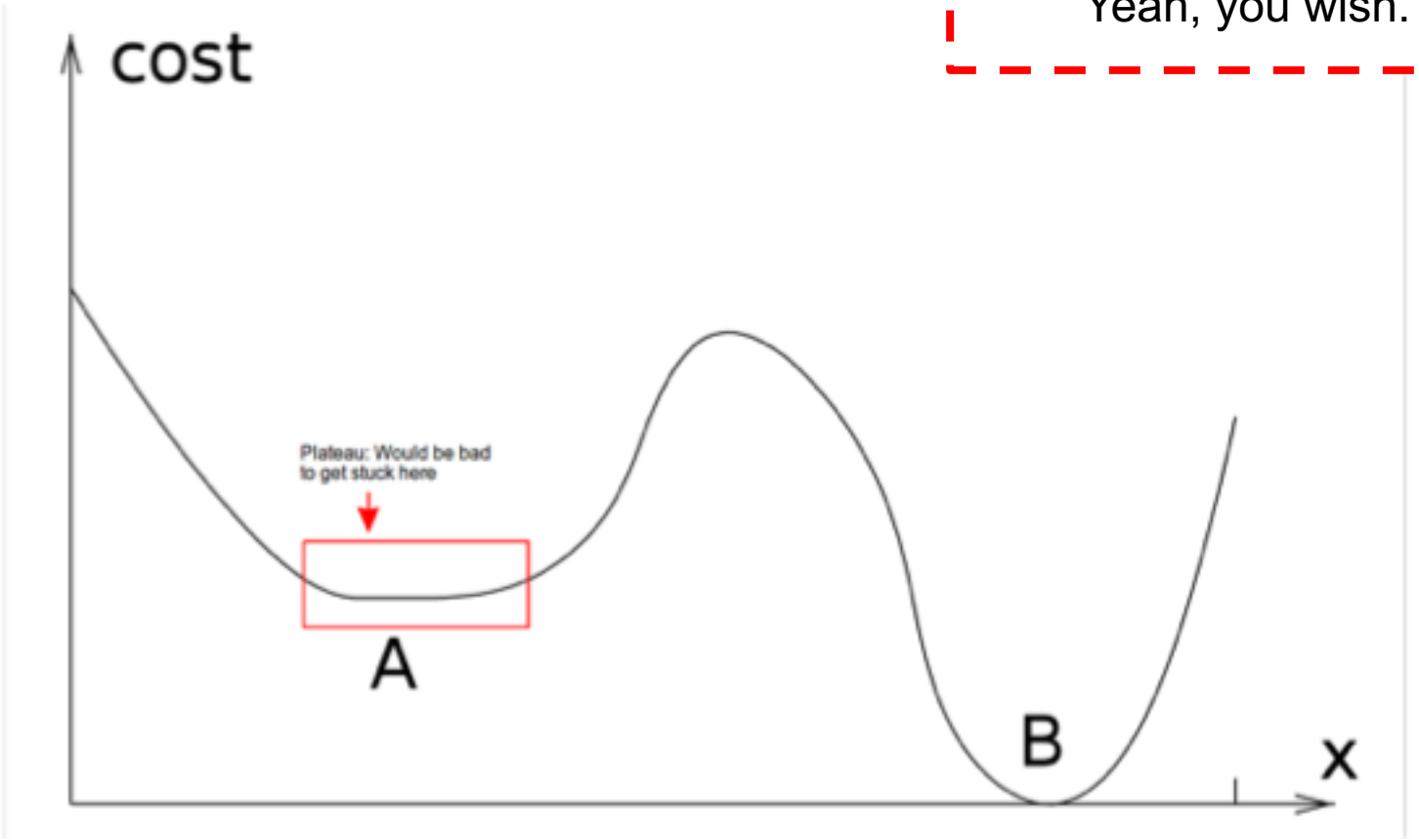


https://medium.com/inveterate-learner/deep-learning-book-chapter-8-optimization-for-training-deep-models-part-i-20ae75984cb2

Challenge #1: Plateaus and other ne'er-do-well minima

You can run but you cannot escape.

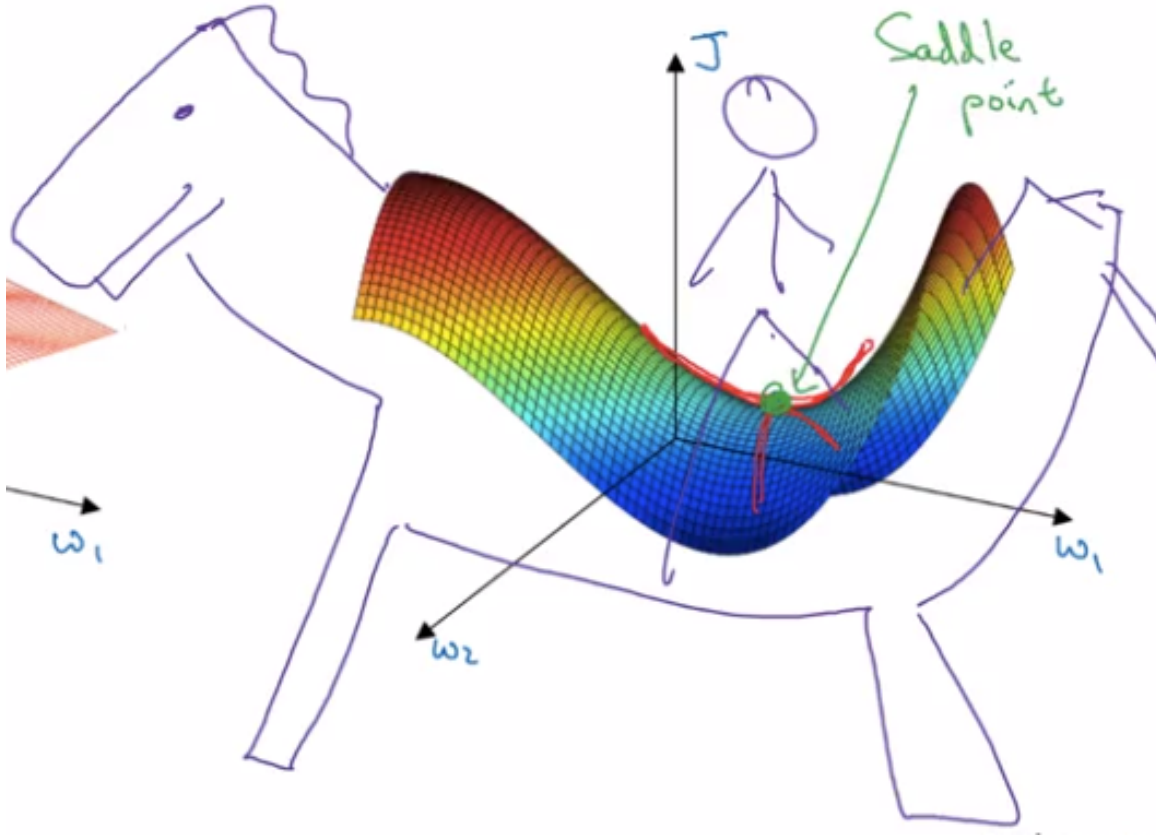
Yeah, you wish.



https://medium.com/inveterate-learner/deep-learning-book-chapter-8-optimization-for-training-deep-models-part-i-20ae75984cb2

Challenge #1: Plateaus and other ne'er-do-well minima

You can run but you cannot escape.



It is well-reasoned that saddle points are much more common for NNs then local minima.

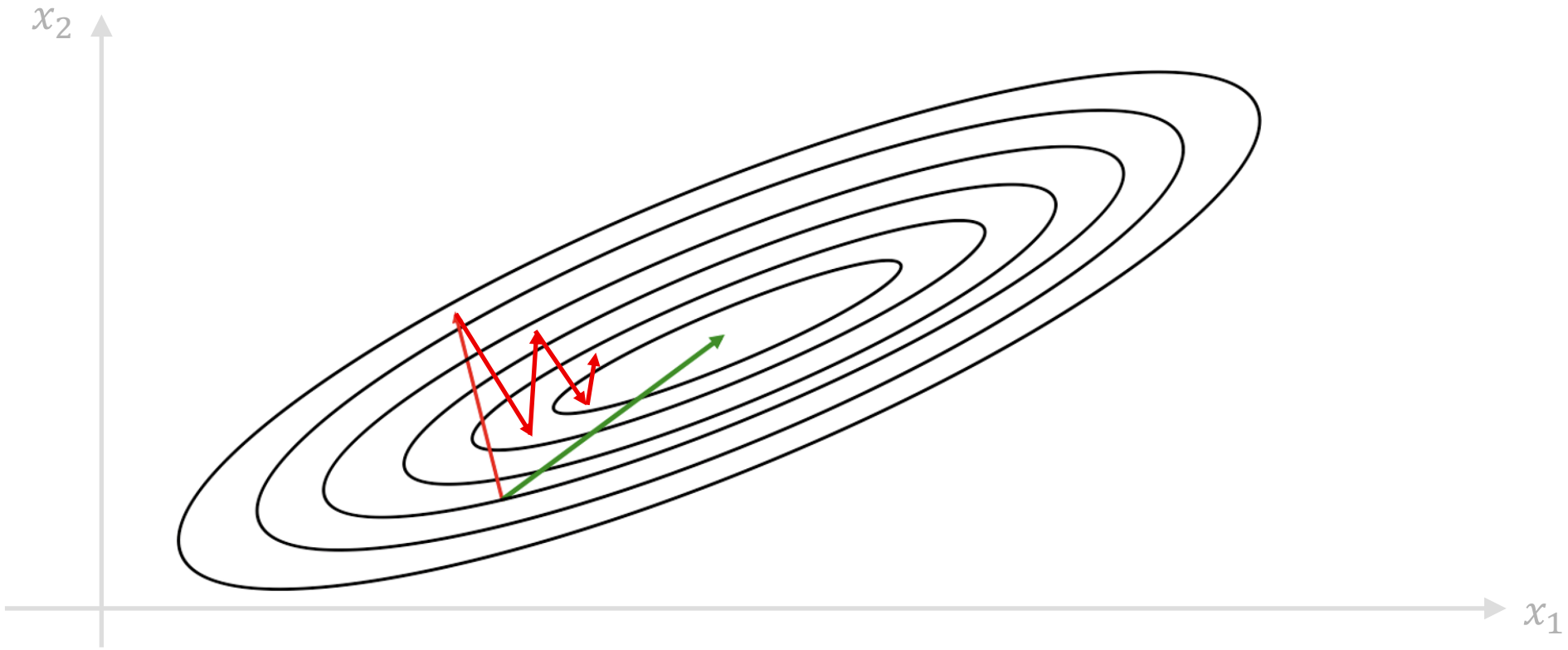
⇒ We have SO many dimensions that it is indeed difficult to find an ACTUAL minimum in all the dimensions.

https://medium.com/inveterate-learner/deep-learning-book-chapter-8-optimization-for-training-deep-models-part-i-20ae75984cb2

Challenge #2: Ill-conditioning

An ill-conditioned problem is a big problem. (Even if it is nicely convex!)

It is one where the error surface looks like a valley. Not cool. Poor SGD can get "stuck".

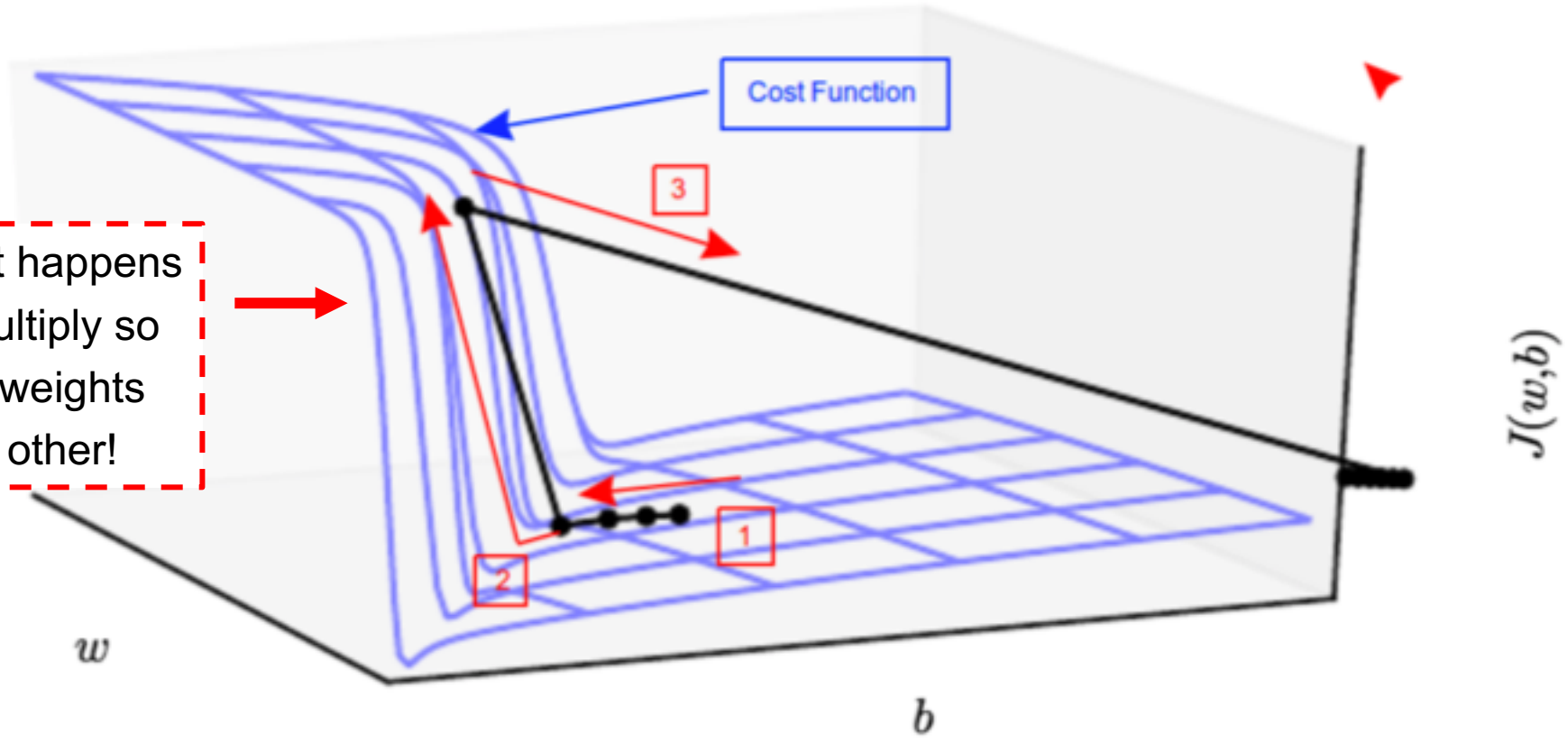


Challenge #3: Cliffs and Exploding Gradients

Mission Impossible-style.

("Gradient clipping" is one solution.)

"This" is what happens when you multiply so many large weights with each other!



https://medium.com/inveterate-learner/deep-learning-book-chapter-8-optimization-for-training-deep-models-part-i-20ae75984cb2

Challenge #4: Long-Term Dependencies

The infamous **vanishing and exploding** gradients problem.

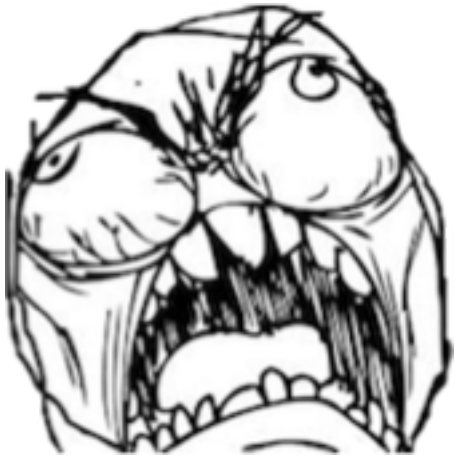
Problem: When network is too deep. ESPECIALLY WITH RNNS! (We will see this.)

To calculate the gradient at a lower layer, we multiply the Loss with some weights MANY TIMES

If weights > 1 ⇒⇒⇒ Gradients EXPLODE!

If weights < 1 ⇒⇒⇒ Gradients VANISH!

WHAT, AM I GONNA KEEP EVERY WEIGHT AT EXACTLY 1???



Challenge #4: Long-Term Dependencies

Feed-Forward NNs: Every Weight is different at each layer, so the problem is not so severe.

So as to say, large and small weights can cancel each other out.

But with RNNS: We multiply the SAME weight matrix with itself, when unfolding through time.

(We will see this.)

The same matrix multiplies its own worst side, and we end up with a serious issue.

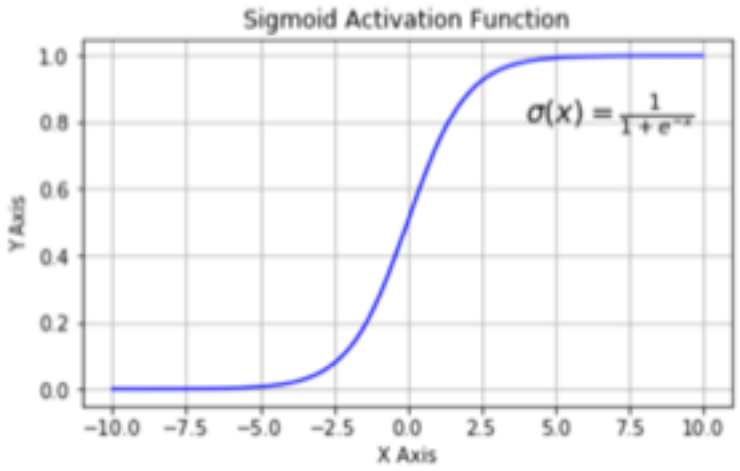
Optimization Approach 1:

Choose your non-linearity wisely

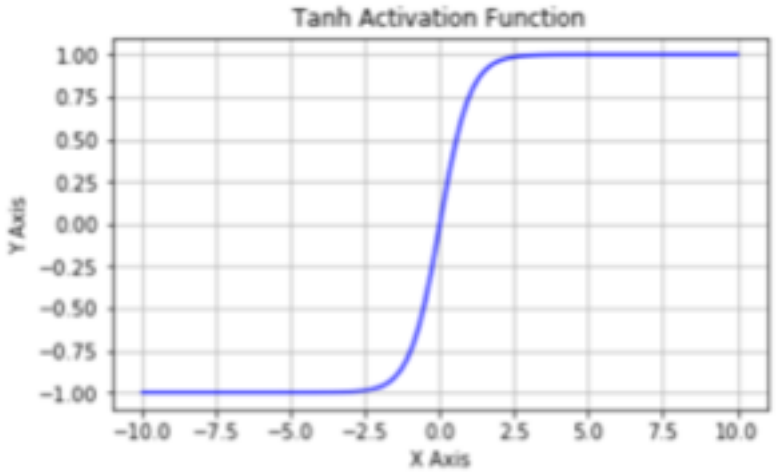
When `sigmoid` vs `relu` vs `tanh` matters

(Yes, apparently this can really matter sometimes. Nobody guessed, tbh.)

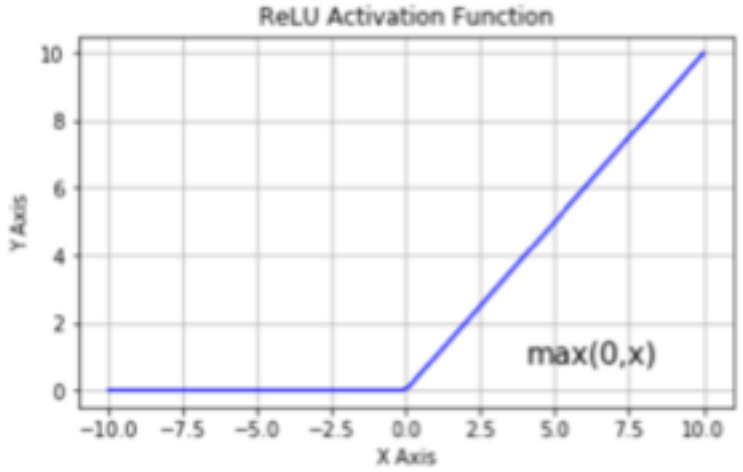
`sigmoid`

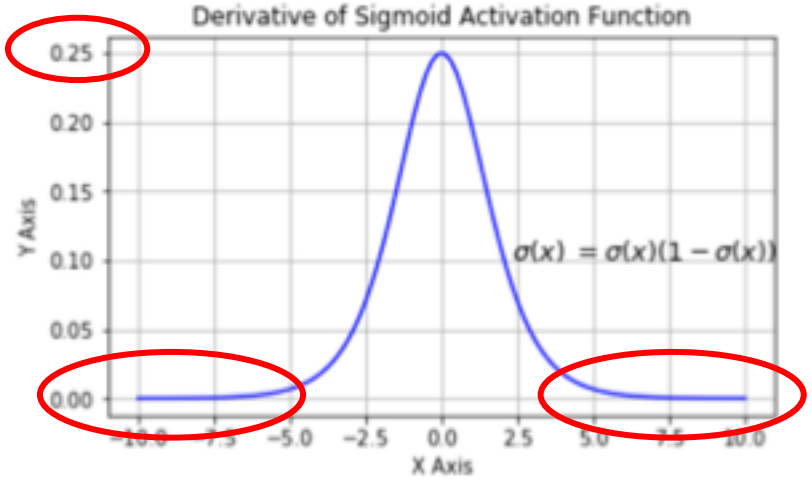


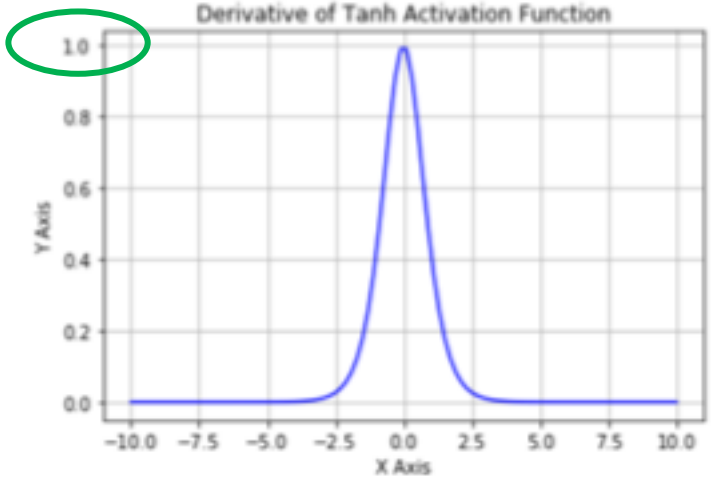
`tanh`

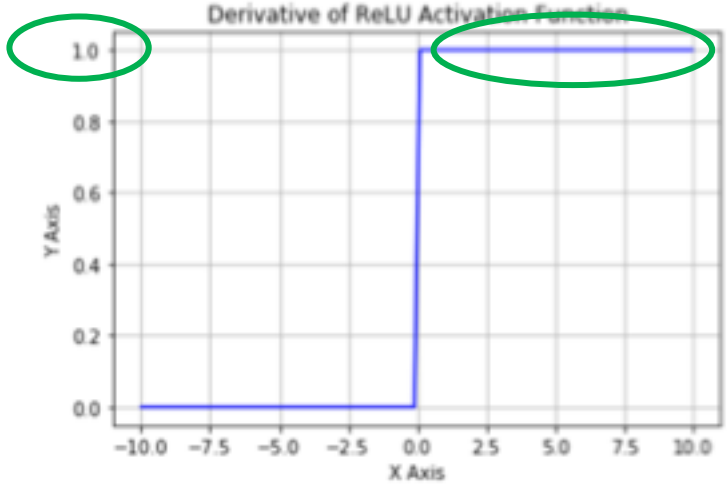


`relu`







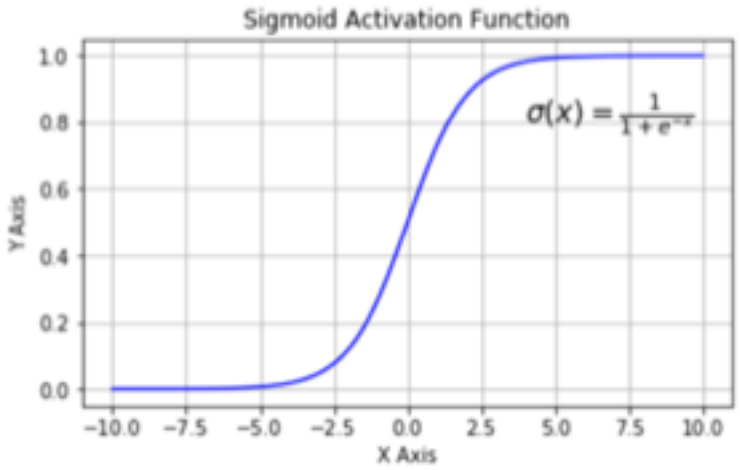


leaky relu

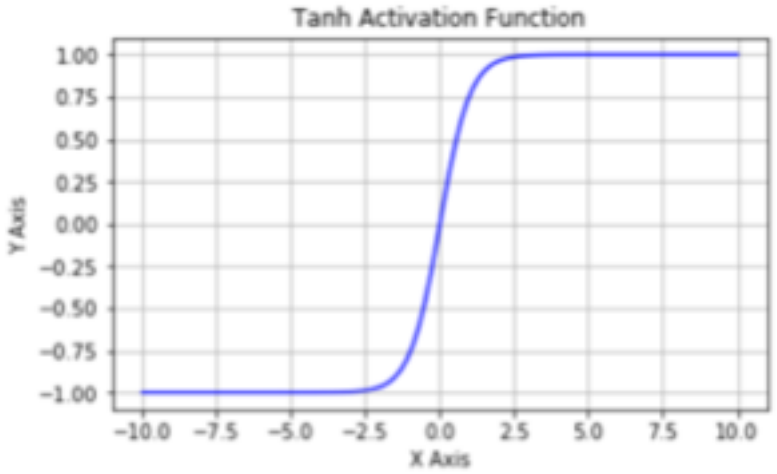
When sigmoid vs relu vs tanh ma

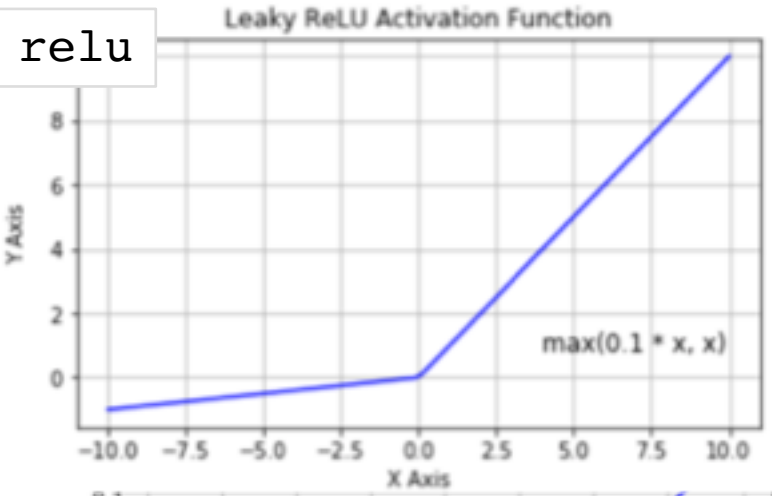
(Yes, apparently this can really matter sometimes. Nobo

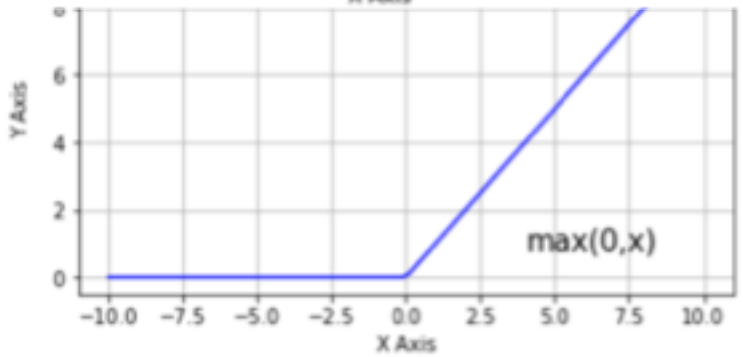
sigmoid

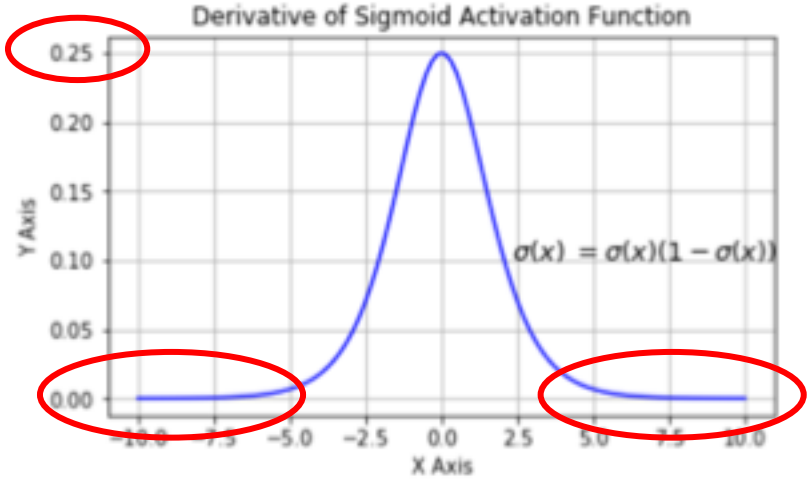


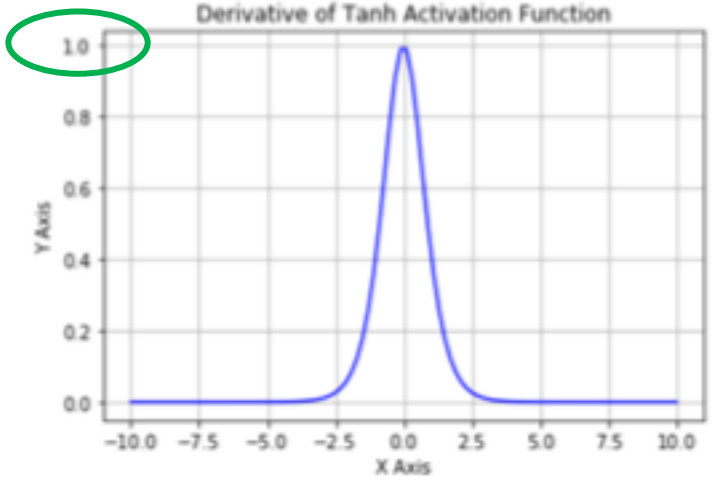
tanh

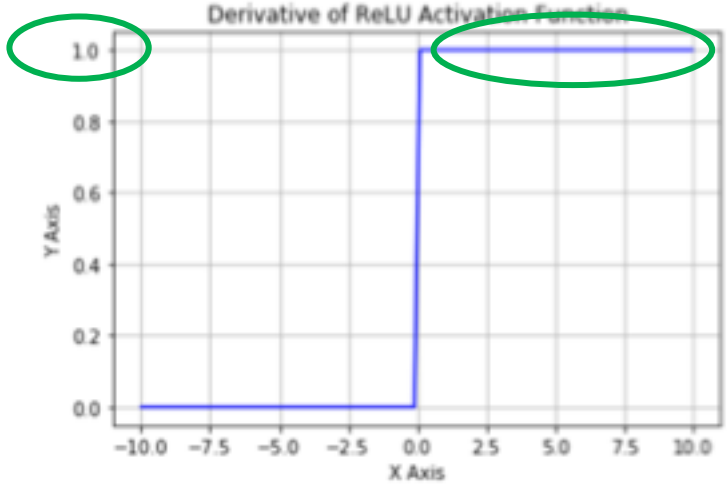












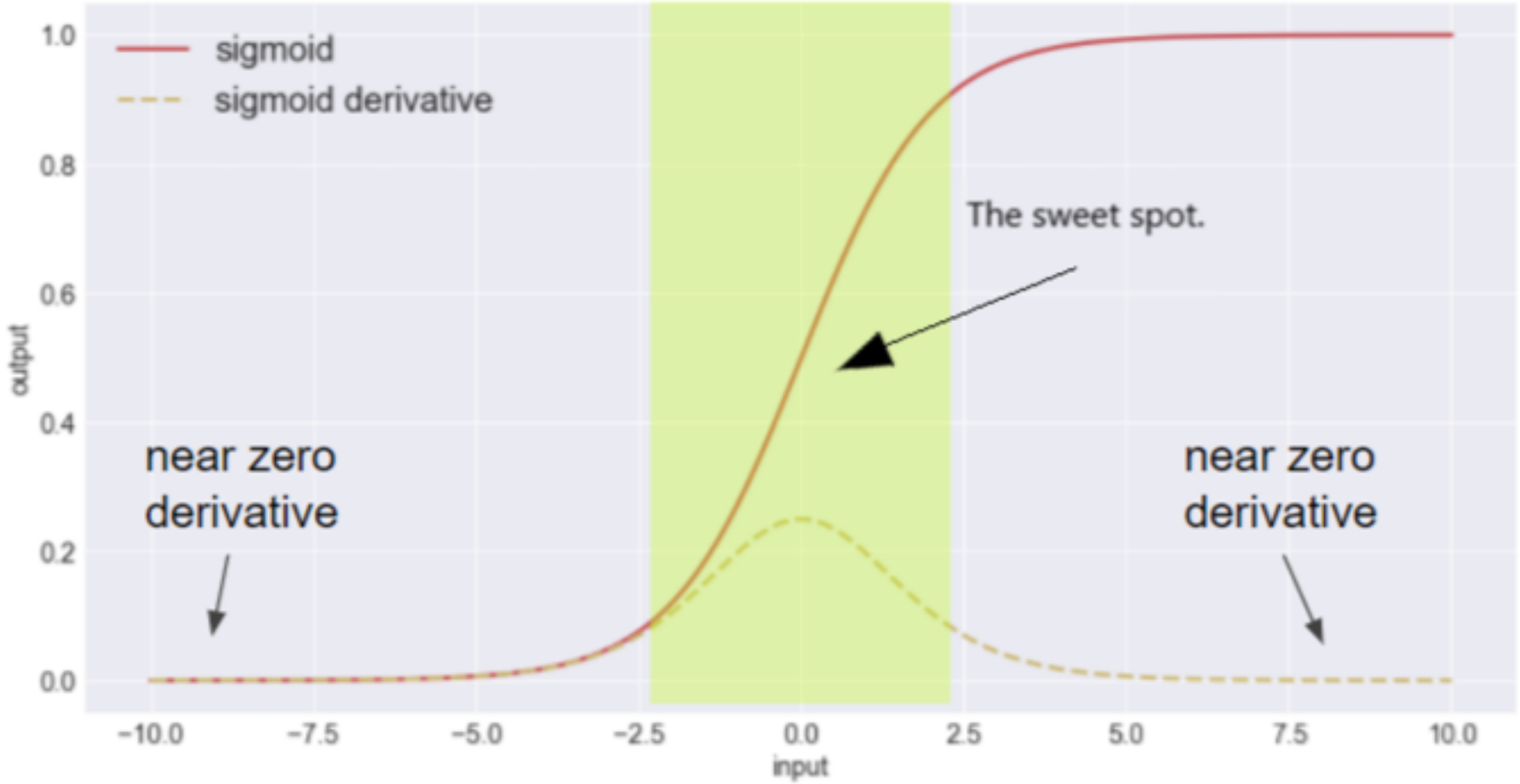
Optimization Approach 2:

Make your signals behave well

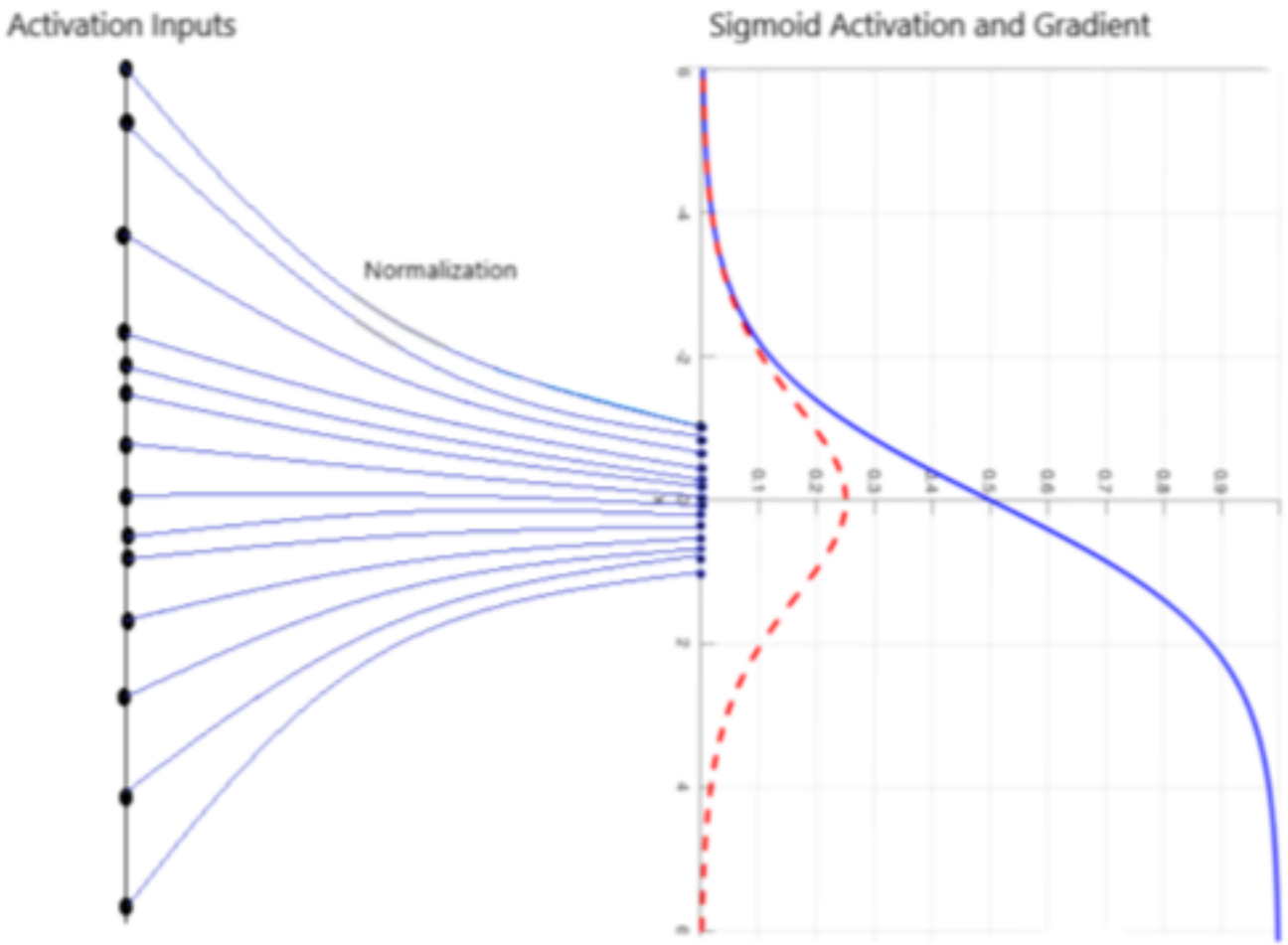
Batch Normalization

OK, “officially” this is not an optimization algorithm.

But it still happens to be one of the most effective optimization methods we ever bumped in so far!



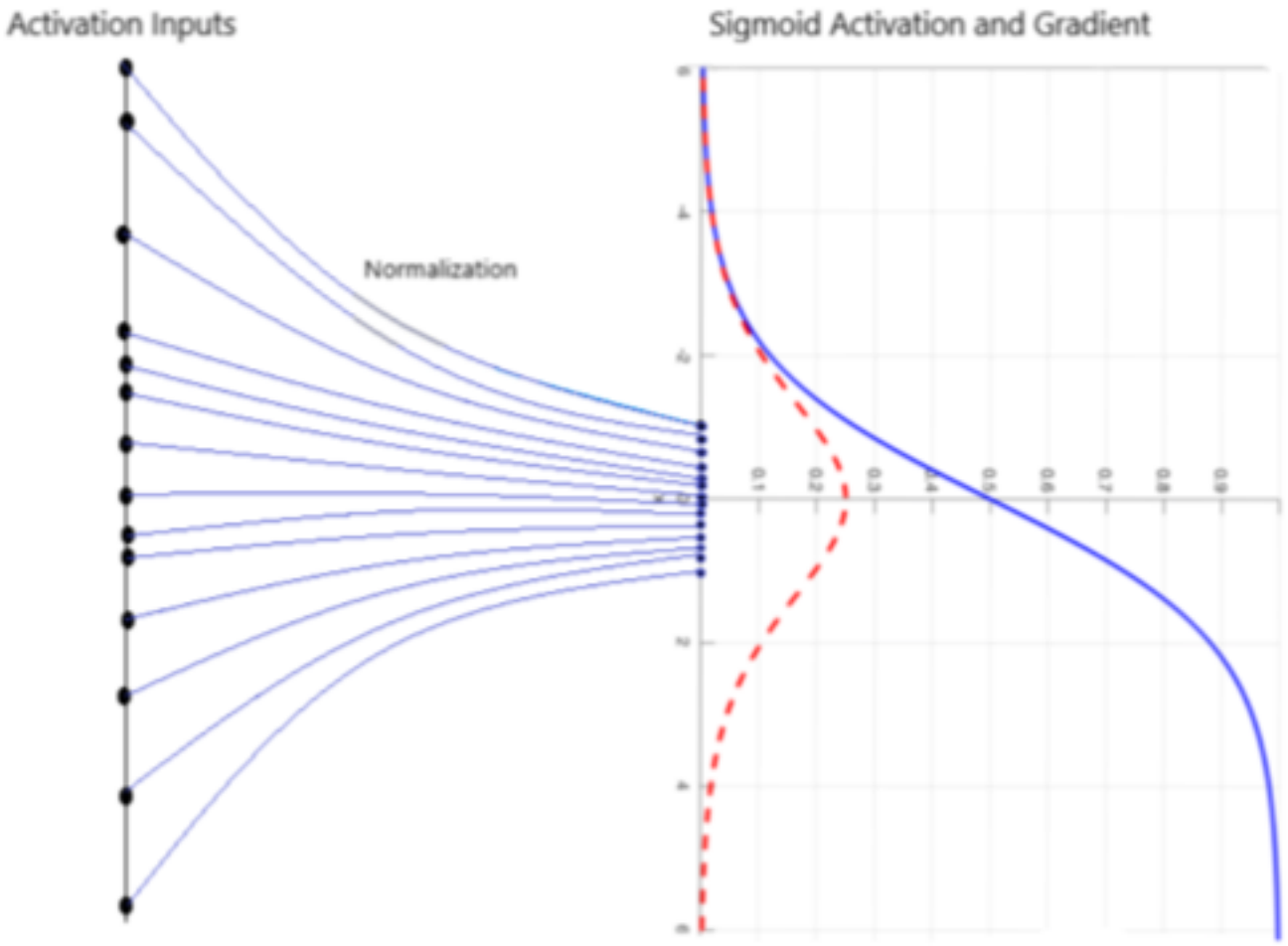
Batch Normalization



$$\hat{x} = \frac{x - \mu}{\sigma^2 + \epsilon}$$

- x: Original input
- μ: Original input's mean
- $\hat{x}$: Normalized input
- σ^2: Original input's variance
- ϵ: Super small number

Batch Normalization



1. Helps prevent vanishing gradients
2. Allows higher learning rates
3. May smooth-out the error surface (Claimed)
4. May regularize the model (Claimed)

Remember me?

Batch normalization works folks.

It works very well.

Exactly how?

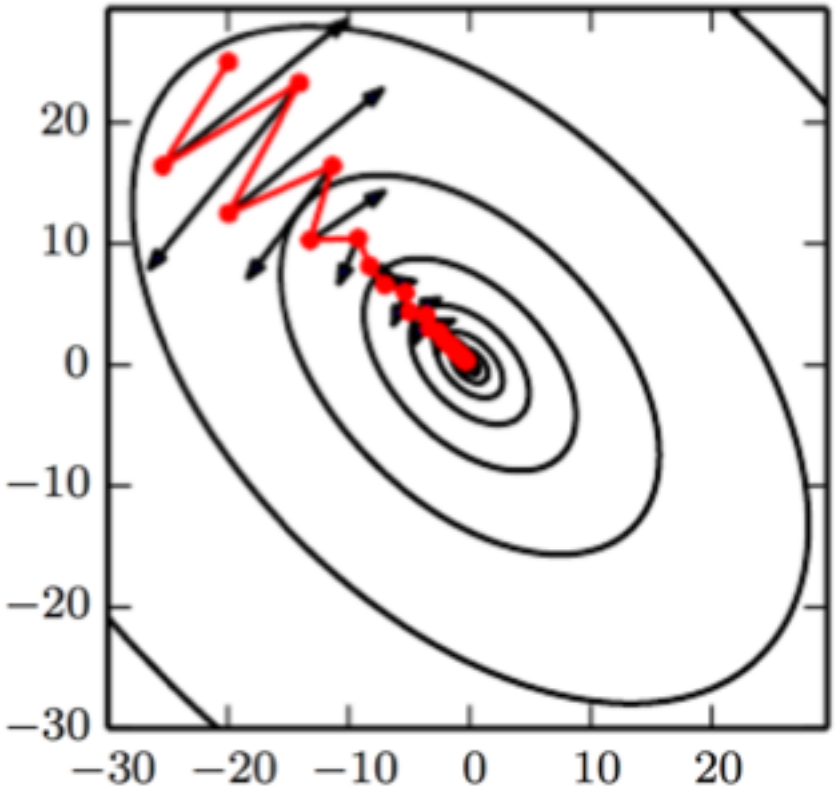
Maybe you will be the one to find out.

Optimization Approach 3:

Let’s gather some Momentum!

Momentum

Just what you need if you have ill-conditioning.



Black: Gradient descent without momentum
Red: Gradient descent with momentum

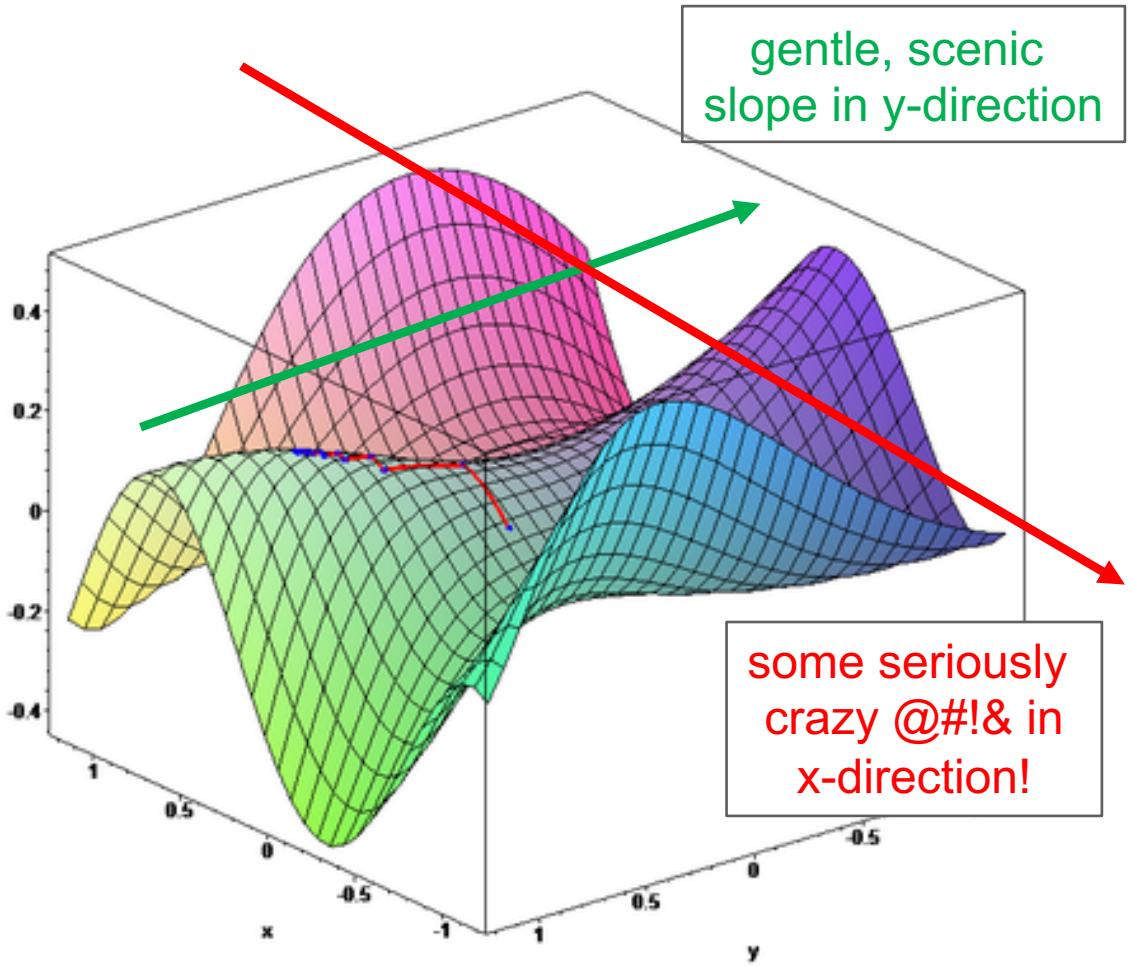
Optimization Approach 4:

In which you have not one, but many, many rates. (Adaptive too!)

Adaptive Learning Rate“s”

Turns out learning rate is difficult to tune.

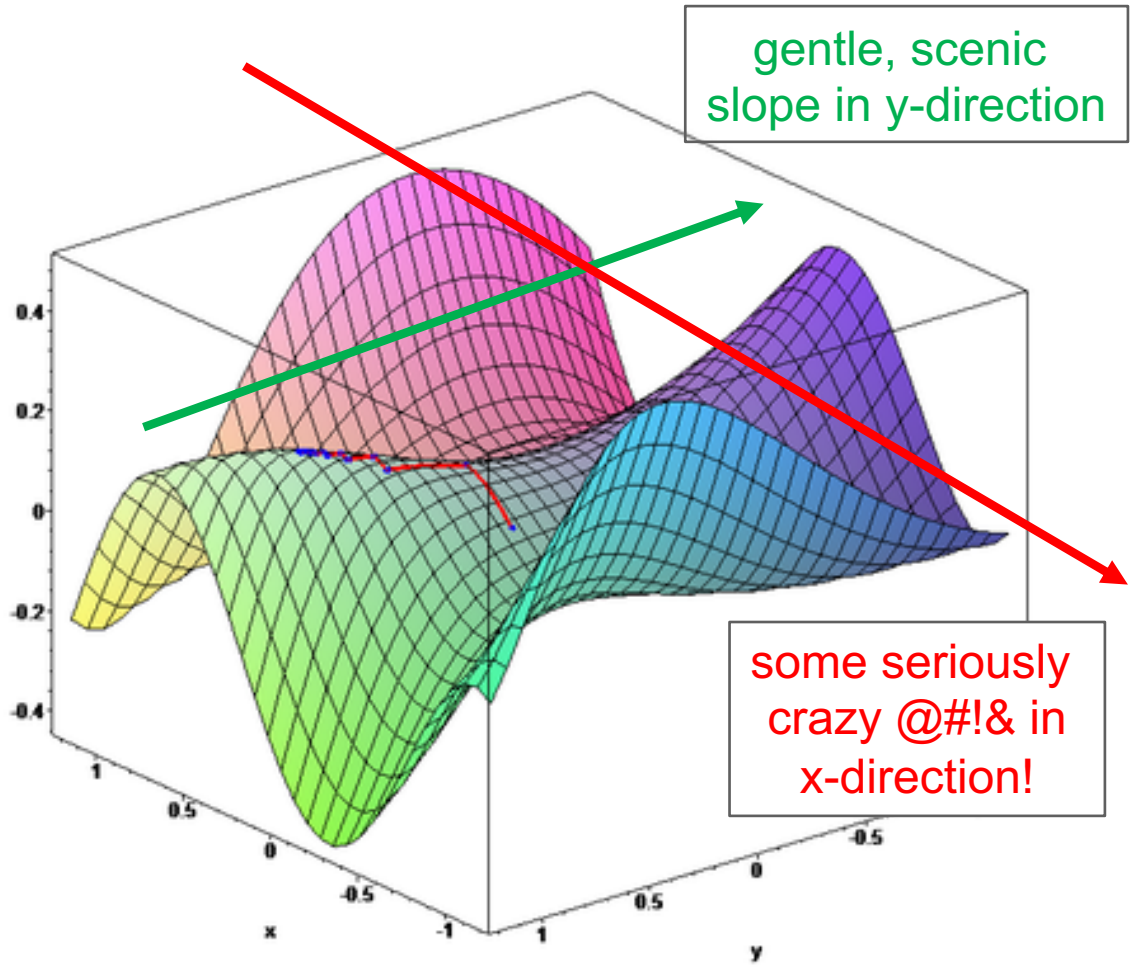
Moreover, the cost may be *super sensitive* to *some* directions in the parameter space, and *insensitive* to others.



Adaptive Learning Rate“s”

So, how about we set a *separate* learning rate for each parameter?

Let’s do this *incrementally* too, as we see more about the data.



#1: AdaGrad

Individually adapt the LR of each parameter,

by remembering the sum of all its history of gradient^2's:

$$r = (g_w^{t=1})^2 + (g_w^{t=2})^2 + (g_w^{t=3})^2 + (g_w^{t=4})^2 + \ \ldots$$

and "dividing" its LR to this r.

Idea: If one parameter is changing too rapidly, be careful in this direction, go slowly!

If another is changing slowly, you are on a gentler slope on this parameter, can go faster in that direction to save time.

#2: RMSProp

A modification of AdaGrad.

Instead of the sum of all its history of gradient^2 as before,

use a weighted moving average:

$r = \rho r + (1 - \rho)(g_w^t)^2$, at each time t.

and “dividing” its LR to this r.

#3: Adam

Adam = Adaptive Moments

= RMSProp + Momentum

1. use a weighted moving average:

$r = \rho r + (1 - \rho)(g_w^t)^2$, at each time t.

and “divide” its LR to this r

2. THEN apply momentum to this LR

Optimization Approach 6:

Get a head start if you can

The Secrets of Parameter Initialization

1. Gotta “break symmetry”:

    ⇒ If all initial weights are the same, they will ALL grow the same too!

    So we need to initialize each a bit differently.

Yeah, just draw them from a normal or uniform distribution or whatever.

And biases? Set them all the 0!

(There are some heuristics too, eg. normalized initialization, if you’d like to check.)

The Secrets of Parameter Initialization

2. You know what, I already know this point that is a pretty good solution.

   I'll start from right here, so basically any solution I can find around will be OK.

   Remember pretraining?

   - We train on a lot of data first (can be indep of our task!), then we can fine-tune

   - Or if your task is unsupervised (*well, good luck with that!*), then train on a related supervised task first.

Coming up next

Mini-batching example

RNNS