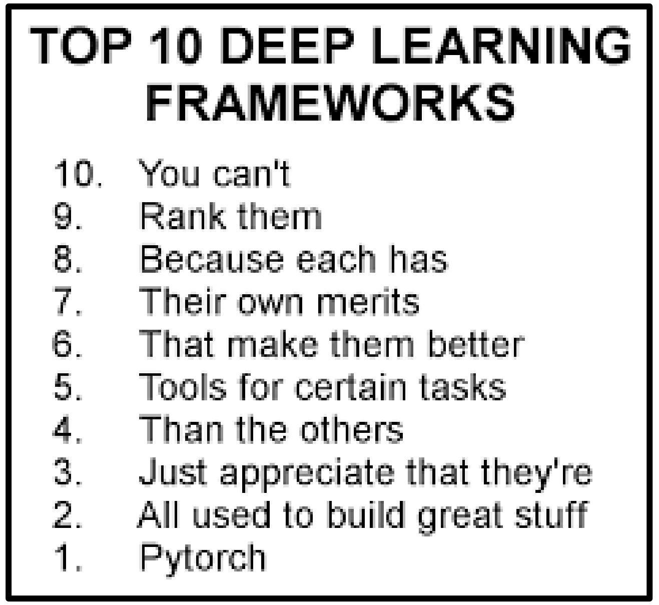
Intermediate pytorch concepts

Alberto TONDA, Ph.D. (Senior permanent researcher, DR)

UMR 518 MIA-PS, INRAE, AgroParisTech, Université Paris-Saclay
UAR 3611, Institut des Systèmes Complexes de Paris Île-de-France

Outline

- Where is the SoftMax?
- Monitor performance
- Tensorboard
- Stochastic Gradient Descent
- Activation functions
- Checkpointing

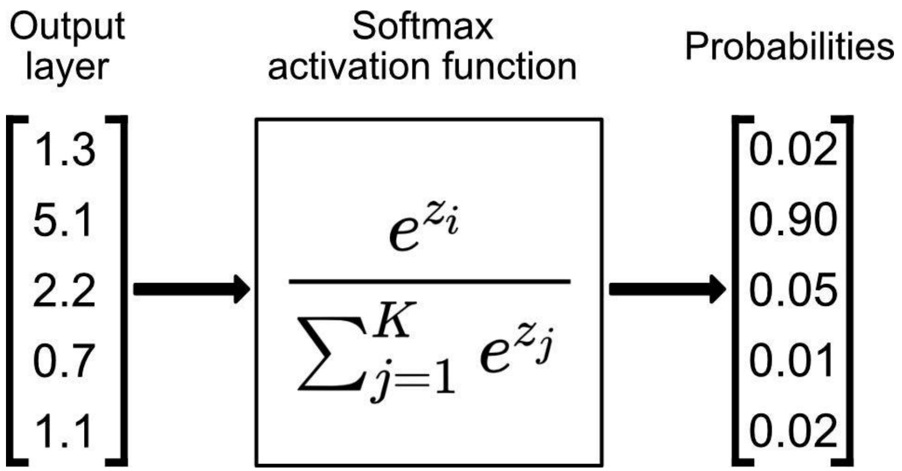


TOP 10 DEEP LEARNING FRAMEWORKS

10. You can't
9. Rank them
8. Because each has
7. Their own merits
6. That make them better
5. Tools for certain tasks
4. Than the others
3. Just appreciate that they're
2. All used to build great stuff
1. Pytorch

Where is the SoftMax?

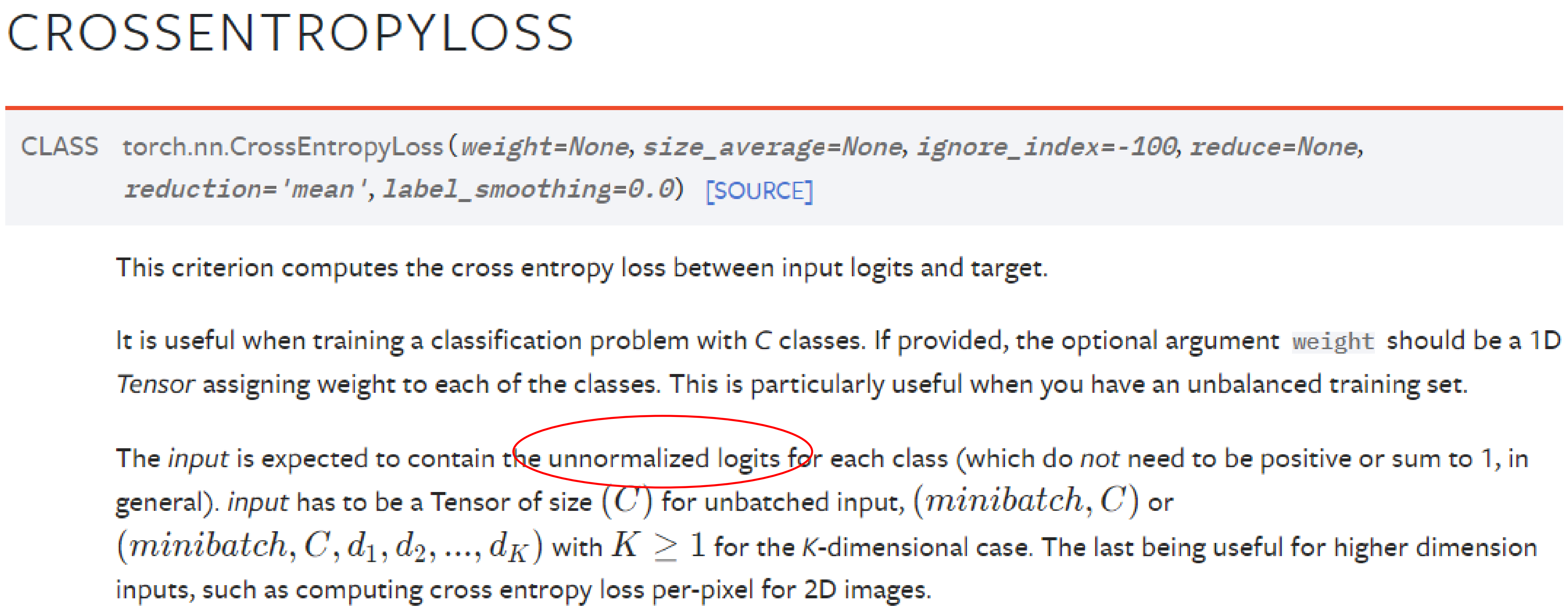
- We grazed over classification
  - But in literature, output tensor is sent through SoftMax
  - SoftMax is used to get values in [0.0,1.0] who add to 1.0
  - Sometimes called “class probabilities”, but **they are not**
- Where was the SoftMax? Was there an error in the code?



Output layer
Softmax activation function
Probabilities
1.3
5.1
2.2
0.7
1.1
$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
0.02
0.90
0.05
0.01
0.02

Where is the SoftMax?

- The SoftMax is *inside* the loss function



CROSSENTROPYLOSS

CLASS torch.nn.CrossEntropyLoss(*weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean', label_smoothing=0.0*) [SOURCE]

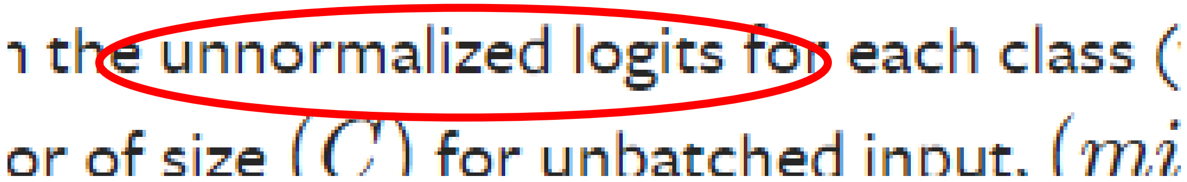
This criterion computes the cross entropy loss between input logits and target.

It is useful when training a classification problem with C classes. If provided, the optional argument `weight` should be a 1D *Tensor* assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* is expected to contain the unnormalized logits for each class (which do *not* need to be positive or sum to 1, in general). *input* has to be a Tensor of size (C) for unbatched input, $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case. The last being useful for higher dimension inputs, such as computing cross entropy loss per-pixel for 2D images.

Where is the SoftMax?

- What is a *logit*?!?

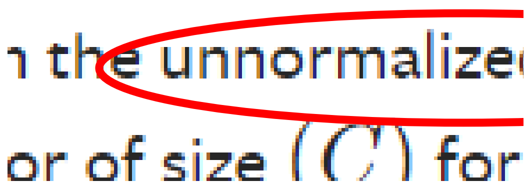


n the unnormalized logits for each class (

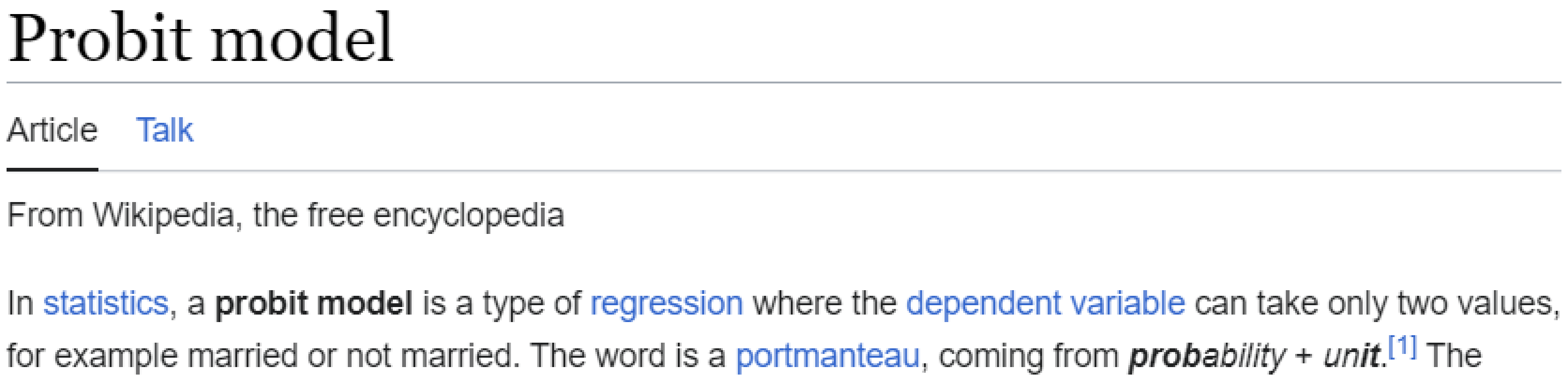
or of size (C) for unbatched input. $(mi$

Where is the SoftMax?

- What is a *logit*?!?


n the unnormalize

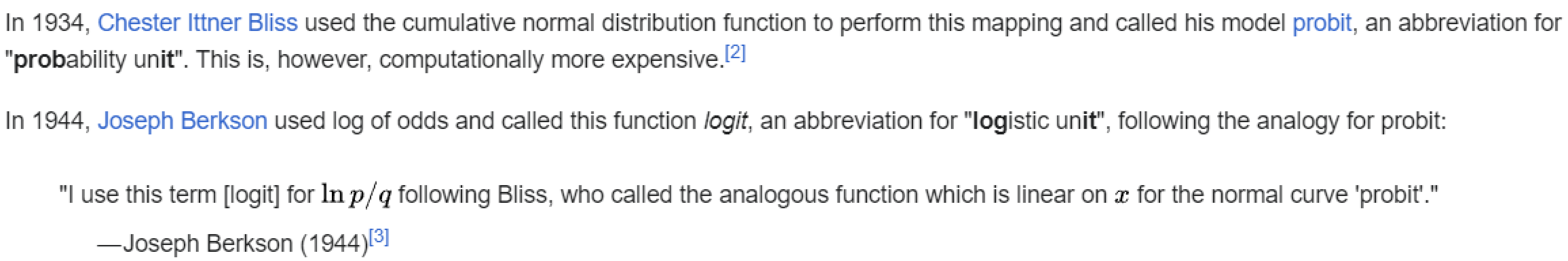
or of size (C) for


Probit model

Article Talk

From Wikipedia, the free encyclopedia

In statistics, a **probit model** is a type of regression where the dependent variable can take only two values, for example married or not married. The word is a portmanteau, coming from ***probability*** *+* ***unit***.[1] The


In 1934, Chester Ittner Bliss used the cumulative normal distribution function to perform this mapping and called his model probit, an abbreviation for "**prob**ability **unit**". This is, however, computationally more expensive.[2]

In 1944, Joseph Berkson used log of odds and called this function *logit*, an abbreviation for "**log**istic un**it**", following the analogy for probit:

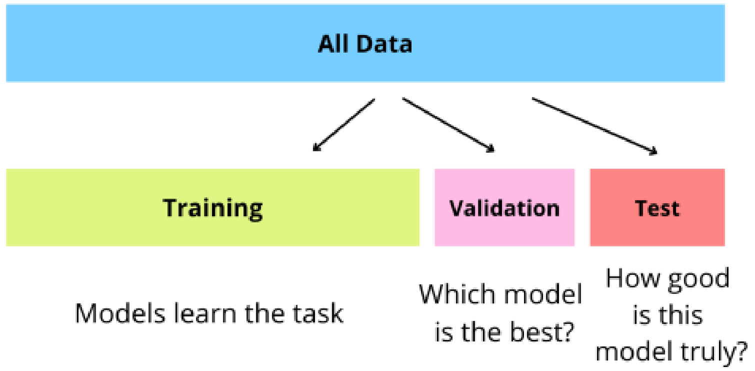
> "I use this term [logit] for $\ln p/q$ following Bliss, who called the analogous function which is linear on x for the normal curve 'probit'."
>
> —Joseph Berkson (1944)[3]

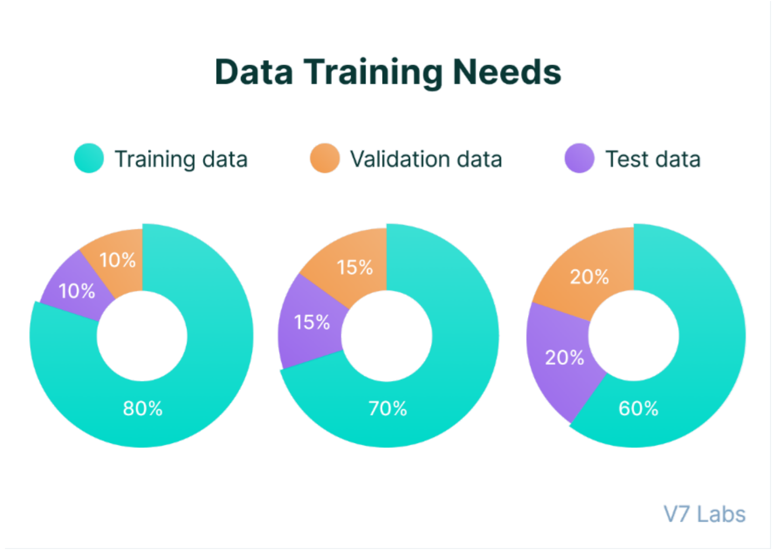
Monitor performance

- Assess performance during training
  - But performance is better evaluated on **test**!
  - Evaluating performance on training is not very informative
  - Overfitting gives the illusion of increasing performance on training
- We want to know performance on *unseen* data!
- But at the same time, we don't want to use the test set!
- How can we solve this conundrum?

Monitor performance

- Further split in the data
  - Three parts: training, **validation**, test
  - Validation is used to assess performance at a given epoch
  - Commonly called “validation loss” (vs “training loss”)





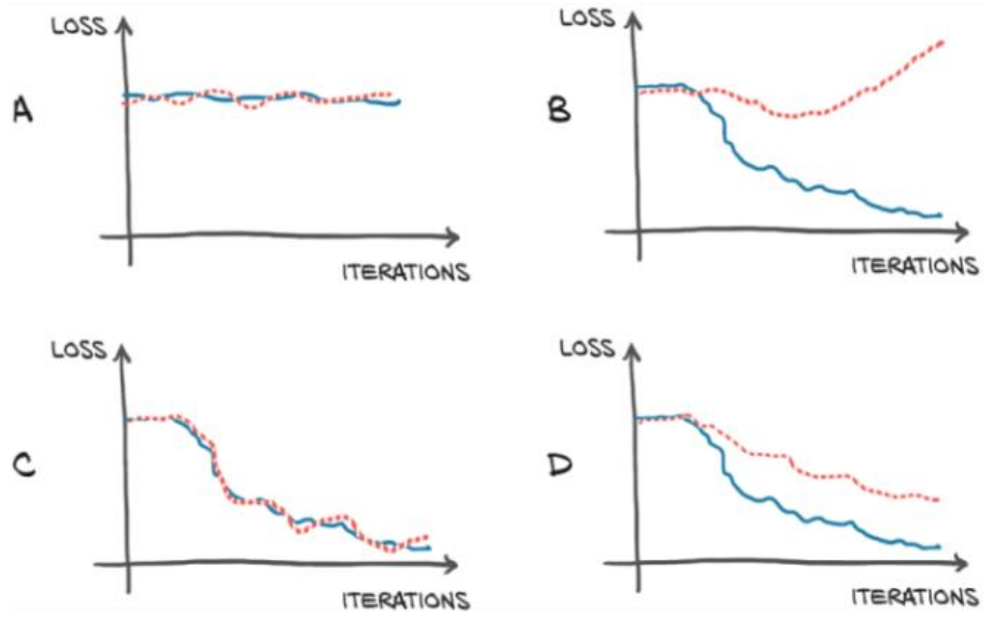
Monitor performance

- Using training loss and validation loss, **detect overfitting**!
  - How?

Monitor performance

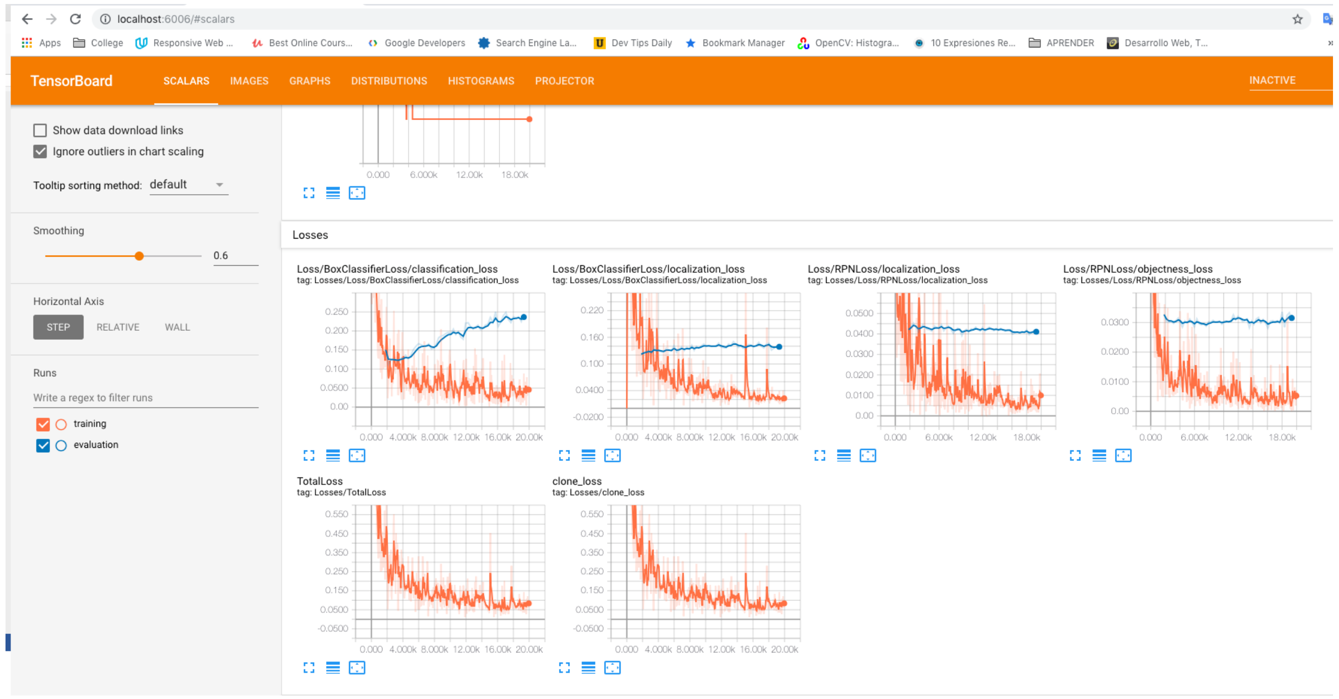
- Using training loss and validation loss, **detect overfitting**!
  - When training loss keeps decreasing, but validation loss *increases*
  - …or remains stationary for a long time

Training loss
Validation loss



Tensorboard

- Software developed by Google (part of Tensorflow)
  - During training, write logs to text files
  - Read text files and present a visualization


localhost:6006/#scalars
TensorBoard
SCALARS
IMAGES
GRAPHS
DISTRIBUTIONS
HISTOGRAMS
PROJECTOR
INACTIVE
Losses
Loss/BoxClassifierLoss/classification_loss
Loss/BoxClassifierLoss/localization_loss
Loss/RPNLoss/localization_loss
Loss/RPNLoss/objectness_loss
TotalLoss
clone_loss
STEP
RELATIVE
WALL

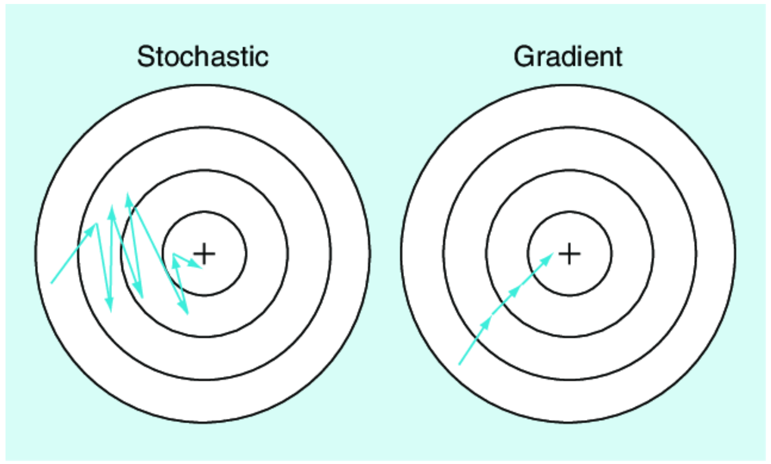
Show data download links
✓ Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0.6
Horizontal Axis: STEP
Runs
Write a regex to filter runs
✓ training
✓ evaluation

Stochastic Gradient Descent?

- You already used SGD in the exercises
- What is *stochastic* about SGD?

(Really) Stochastic Gradient Descent

- It was NOT STOCHASTIC AT ALL!
- Difference between SGD and GD
  - SGD updates gradients after seeing a *random subset* of the data
  - Random subset is called **batch**
  - Smaller, more frequent updates are more robust and *faster*
  - Typically SGD uses a **smaller learning rate**



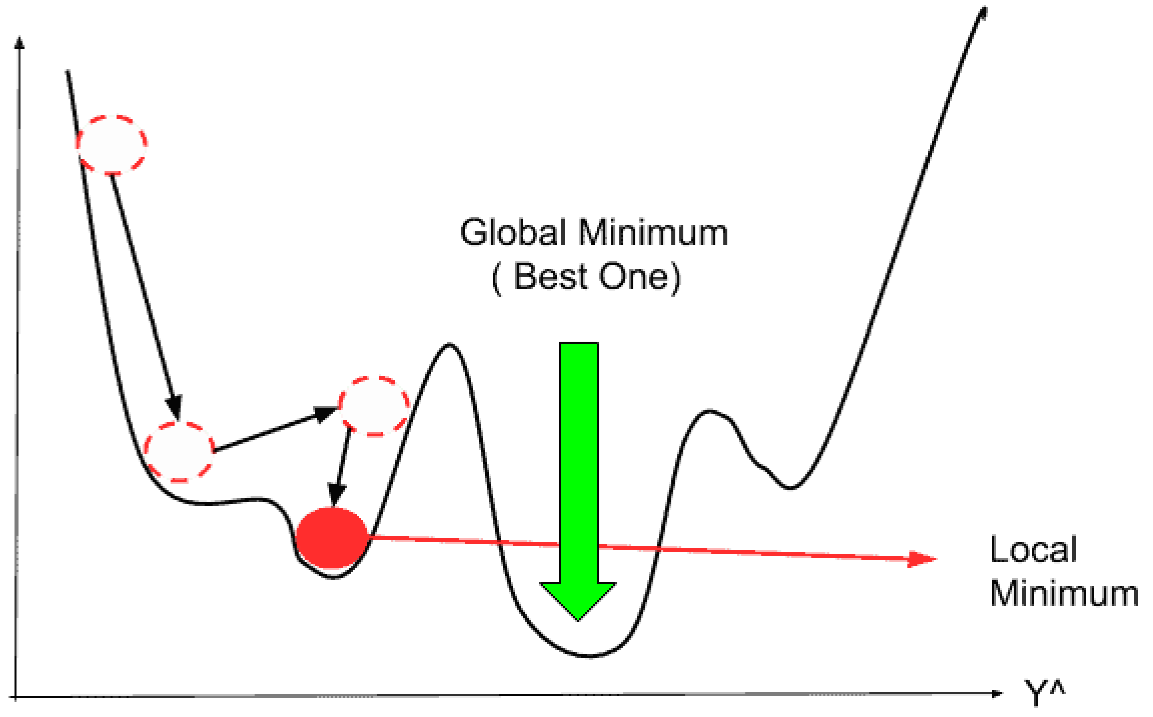
Beyond Stochastic Gradient Descent

- Nobody uses SGD anymore
  - However, its descendants thrive!
  - A cumulative research effort over generations to overcome issues

- Issues of gradient-based techniques
  - Hard/impossible to get out of local optima
  - Starting point of exploration matters
  - Step size? Too small / too large leads to convergence issues

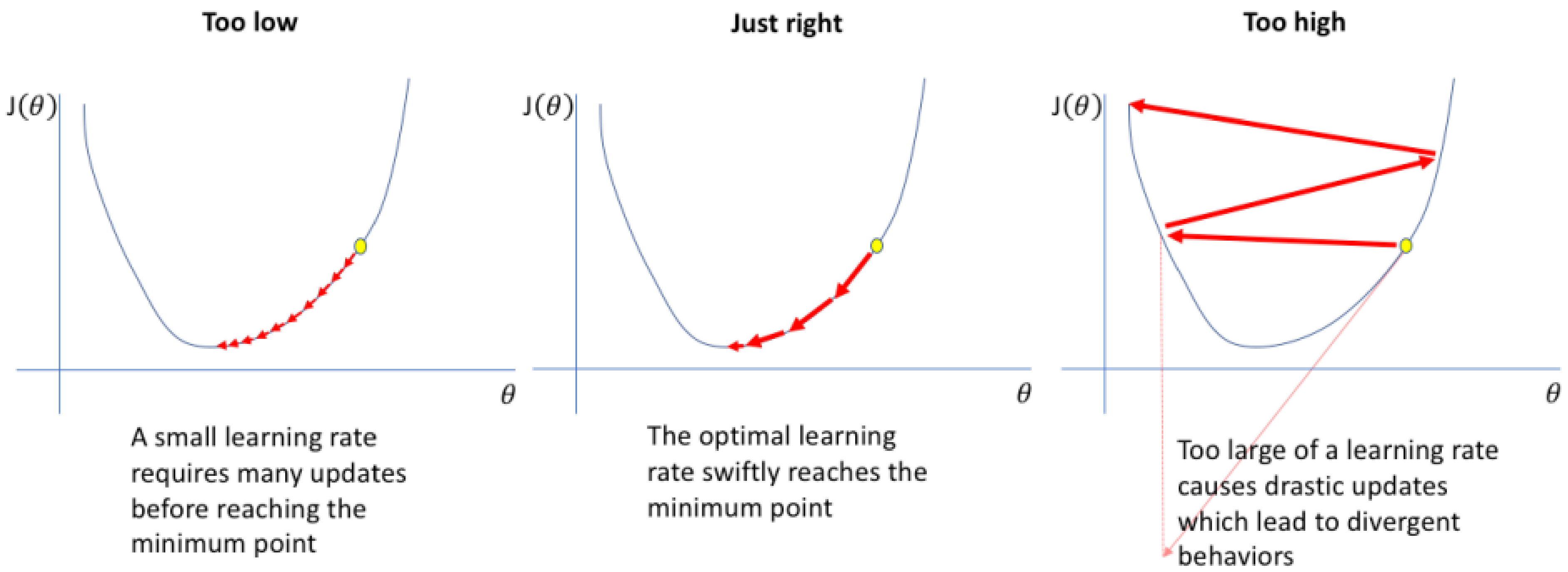
Issues with gradient-based techniques

- Starting point of exploration matters



Issues with gradient-based techniques

- Step size? Too small / too large lead to convergence issues


Too low
Just right
Too high
J(θ)
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors

Modern gradient-based techniques

- Momentum

$$\mathbf{v}^{(k+1)} = \beta\mathbf{v}^{(k)} - \alpha\mathbf{g}^{(k)}$$
$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{v}^{(k+1)}$$

  - Accumulates “velocity” like a ball rolling down an incline
  - Much faster at traversing nearly flat areas of search space
- However, it adds an extra parameter (α, β)
  - α is still the learning rate
  - β represents the importance given to velocity during update

Modern gradient-based techniques

- However, now it cumulates *too much* momentum!
- Nesterov momentum

$$\mathbf{v}^{(k+1)} = \beta \mathbf{v}^{(k)} - \alpha \nabla f(\mathbf{x}^{(k)} + \beta \mathbf{v}^{(k)})$$
$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{v}^{(k+1)}$$

  - Evaluate the gradient at the point planned to end in
  - Velocity is rescaled accordingly

Modern gradient-based techniques

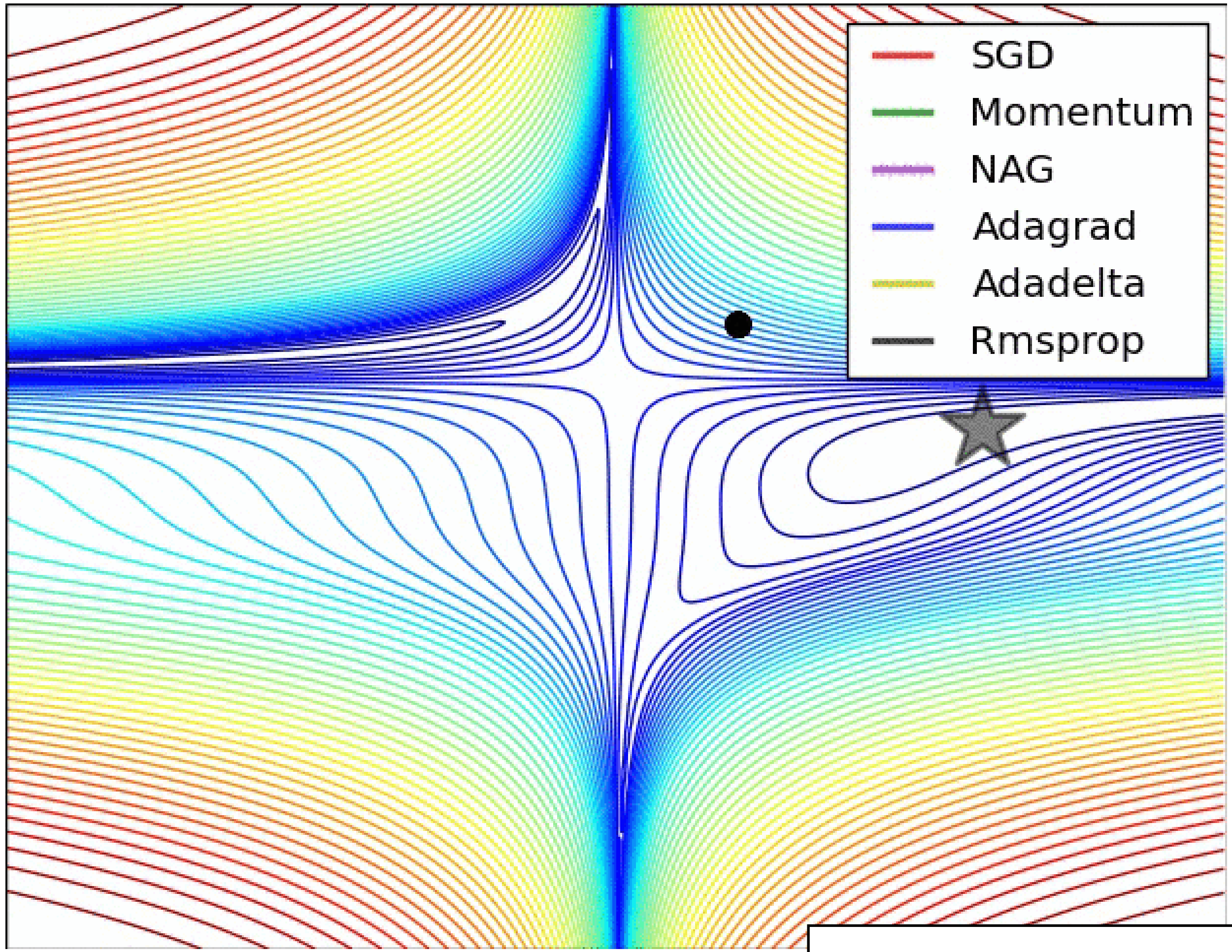
- Further issues motivated further advances (**Adagrad**)
  - The step size could be different in each dimension
  - Maintain a "memory" of the gradient values in each direction

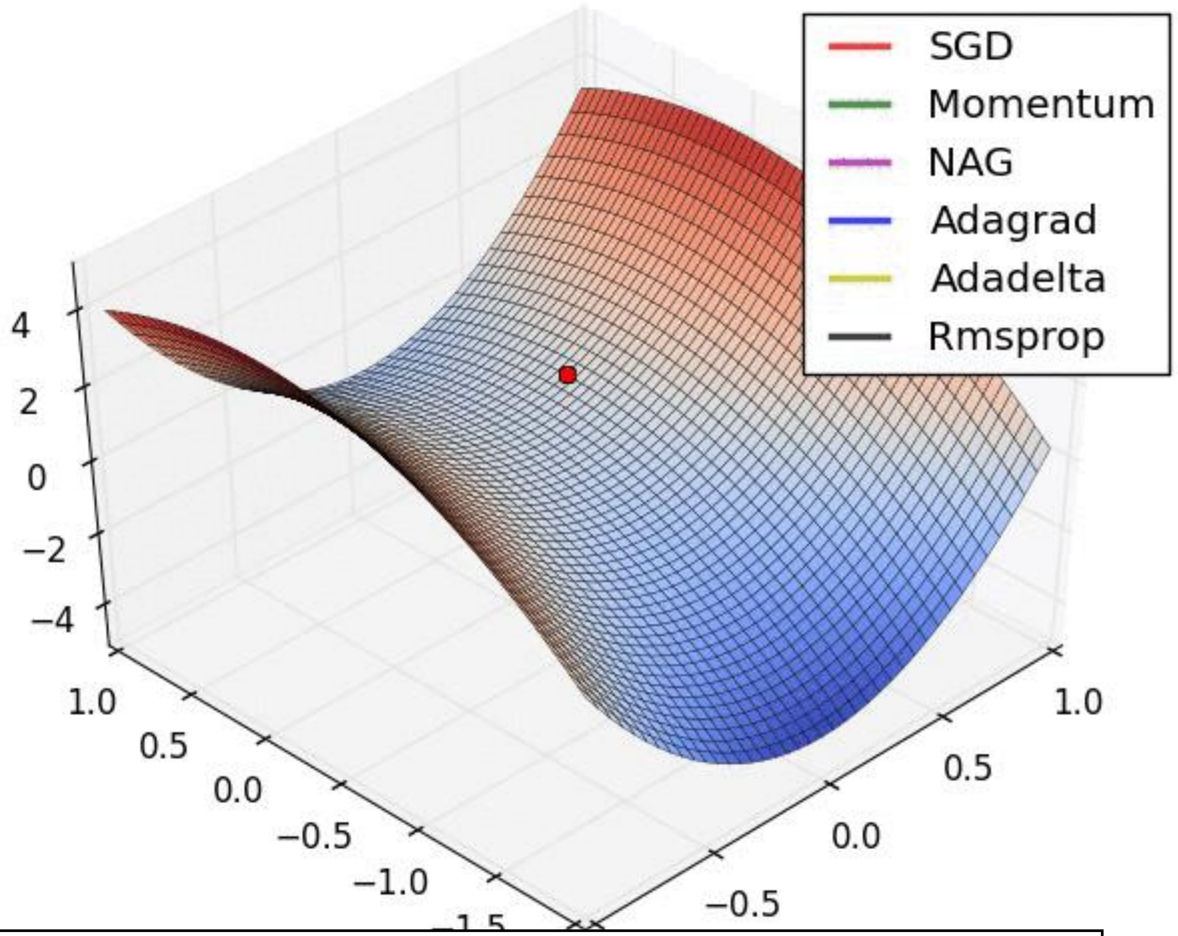
$$x_i^{(k+1)} = x_i^{(k)} - \frac{\alpha}{\epsilon + \sqrt{s_i^{(k)}}} g_i^{(k)}$$

$$s_i^{(k)} = \sum_{j=1}^{k} \left(g_i^{(j)}\right)^2$$

  - However, this leads to the step size always decreasing

Modern gradient-based techniques





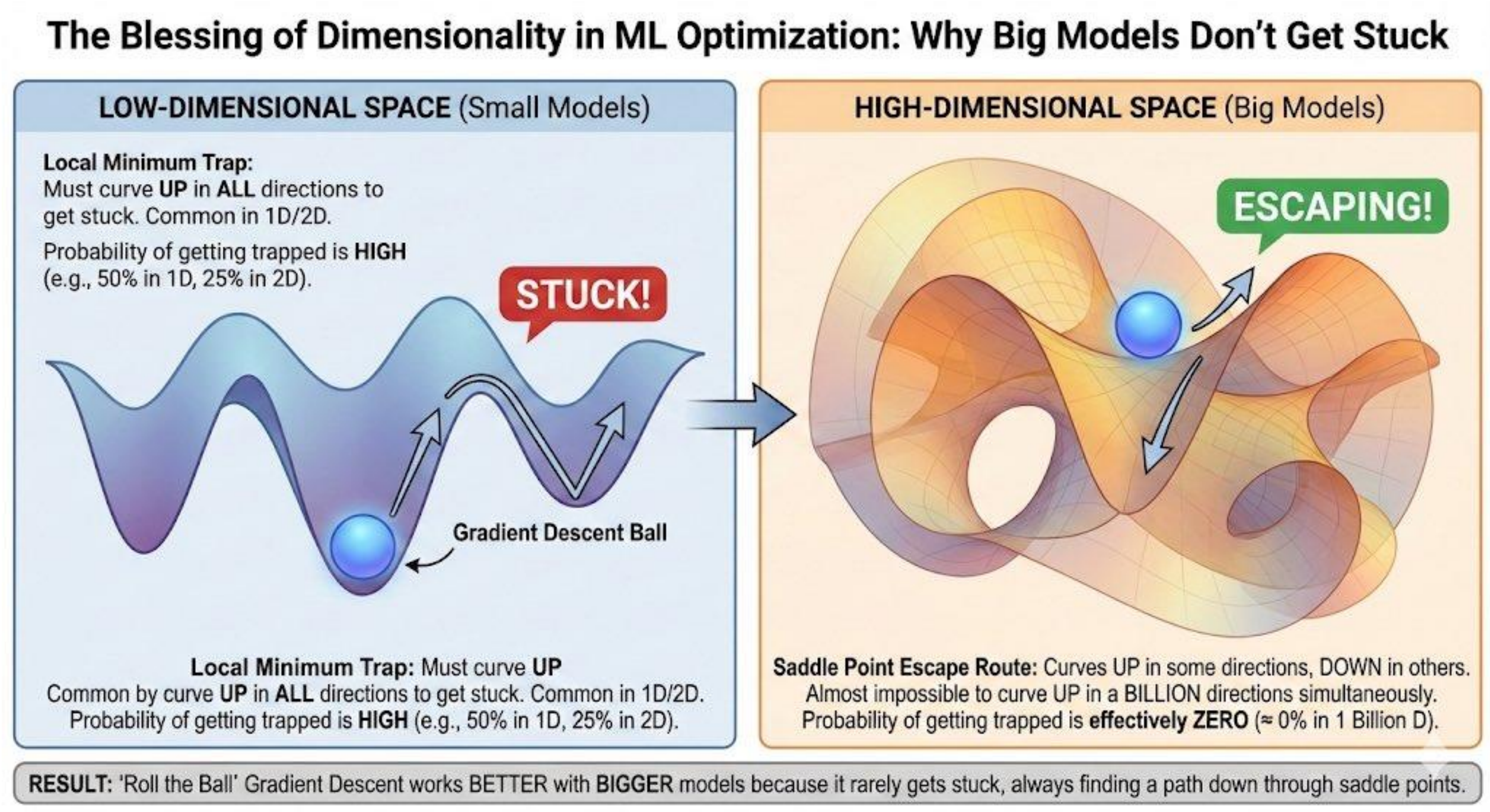
https://deepdatascience.wordpress.com/2017/02/03/optimiser-choice/

Modern gradient-based techniques

- Solutions to issues of previous algorithms create new issues
  - Commonly used **Adam** and **RMSProp** have 3-4 hyperparameters
  - Also take more memory (history of gradient values)
  - Default values work reasonably well (but not always)

- Practical advice
  - Use **Adam** (Adaptive Moment Estimation) in most cases
  - Check out new algorithms! torch.optim.*

Why does SGD work?



The Blessing of Dimensionality in ML Optimization: Why Big Models Don't Get Stuck

LOW-DIMENSIONAL SPACE (Small Models)

Local Minimum Trap:
Must curve **UP** in **ALL** directions to get stuck. Common in 1D/2D.
Probability of getting trapped is **HIGH** (e.g., 50% in 1D, 25% in 2D).

Local Minimum Trap: Must curve **UP**
Common by curve **UP** in **ALL** directions to get stuck. Common in 1D/2D.
Probability of getting trapped is **HIGH** (e.g., 50% in 1D, 25% in 2D).

HIGH-DIMENSIONAL SPACE (Big Models)

Saddle Point Escape Route: Curves UP in some directions, DOWN in others.
Almost impossible to curve UP in a BILLION directions simultaneously.
Probability of getting trapped is **effectively ZERO** (≈ 0% in 1 Billion D).

RESULT: 'Roll the Ball' Gradient Descent works BETTER with **BIGGER** models because it rarely gets stuck, always finding a path down through saddle points.

Carlos Alberto Haro
@haro_ca_

Schedulers

- Adapt optimizer hyperparameters **during training**?
  - Especially learning rate!
  - In general, large(r) initial learning rate, small(er) at the end
  - “Exploration vs Exploitation”
- Example: torch.optim.lr_scheduler.ExponentialLR
  - All hyperparameters, epoch k: $h^{(k+1)} = h^{(k)} \cdot \gamma; \gamma < 1.0$
  - In practice, all hyperparameter values slowly become smaller
  - Common values: $\gamma = 0.9, \gamma = 0.99$
- Other ideas: torch.optim.lr_scheduler.*

Pseudo-random number generation in pytorch

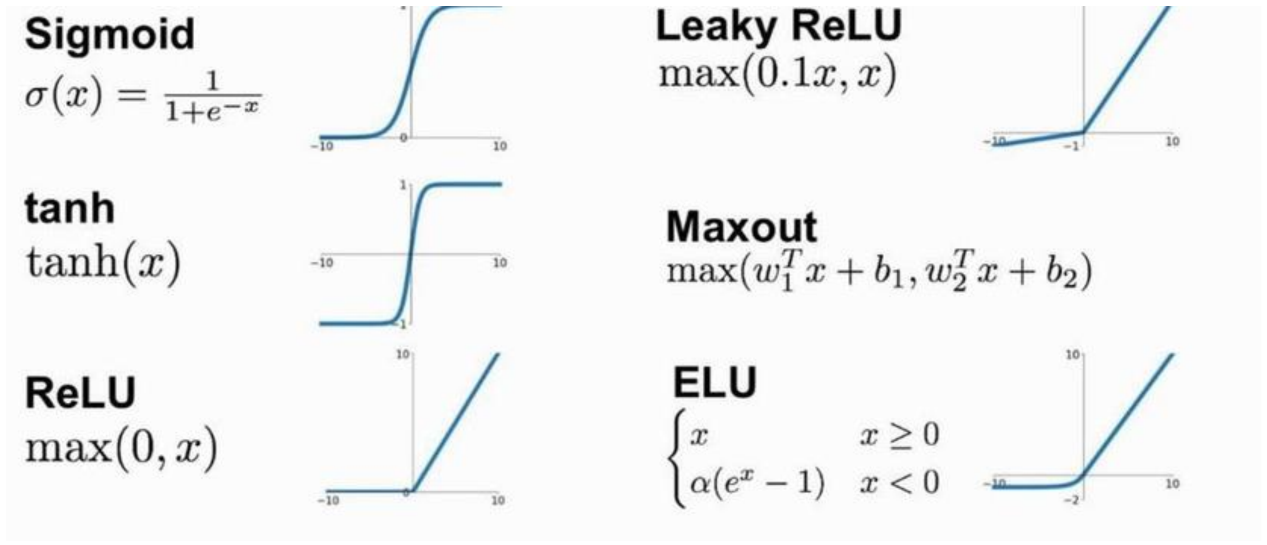
- Unfortunately, it's not easy
  - Computing on GPUs makes consistent PRNG difficult
  - Libraries optimized for *speed*, not consistent behavior

- Still, good practices:

  https://pytorch.org/docs/stable/notes/randomness.html

Activation functions

- So far, we used one of the most basic activation functions
  - But there are several others, more modern
  - Better theoretical properties and empirical results



Activation functions

- Practical advice
  - Check function typically used in literature for target application
  - For recurrent neural networks, TanH or Sigmoid
  - When in doubt, go for ReLUs

- Lots of available activation functions
  - https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity
  - https://pytorch.org/docs/stable/nn.html#non-linear-activations-other

Checkpointing

- Training neural networks can take a long time
  - We can **save the state of the network** during training!
  - State depends only on weight values (+ optimizer, + scheduler)
  - Stop and resume training
  - **Share weights** with other people!

- Checkpoints are one of the foundations of **transfer learning**

Questions?

Bibliography

- pytorch FAQs, https://pytorch.org/docs/stable/notes/faq.html

Images and videos: unless otherwise stated, I stole them from the Internet. I hope they are not copyrighted, or that their use falls under the Fair Use clause, and if not, I am sorry. Please don't sue me.