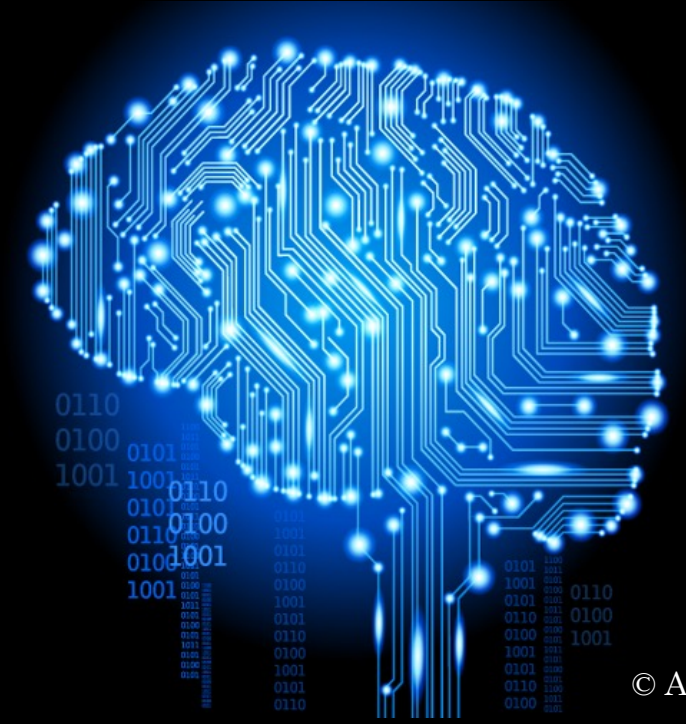


CENG 501
Deep Learning:
A GAME OF NEURONS

Sinan Kalkan

Today

- Loss Functions
- Activation Functions
- Optimization Perspective
- Challenges of the Loss Surface
- Setting the Learning Rate
- Representational Capacity

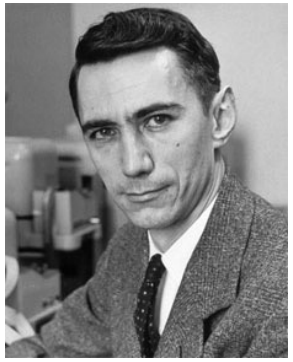
Administrative Notes

- Reading assignment
  - CH1-7 of the Hundred-Page Machine Learning Book by Andriy Burkov. https://themlbook.com/
- Quiz
  - Next week [since registrations are still ongoing]
- Paper Selection
  - https://forms.gle/SXm33dFc9GHnhWje8
  - Deadline next week (1st of March, midnight)

More on loss functions

Softmax or Logistic CLASSIFIERS

Information Entropy



Claude Elwood Shannon
(1916-2001)

("A Mathematical Theory of Communication", 1948)

- Number of bits to represent a coin-pair:

$$\log_2 4 = 2$$

- In fact, this is:

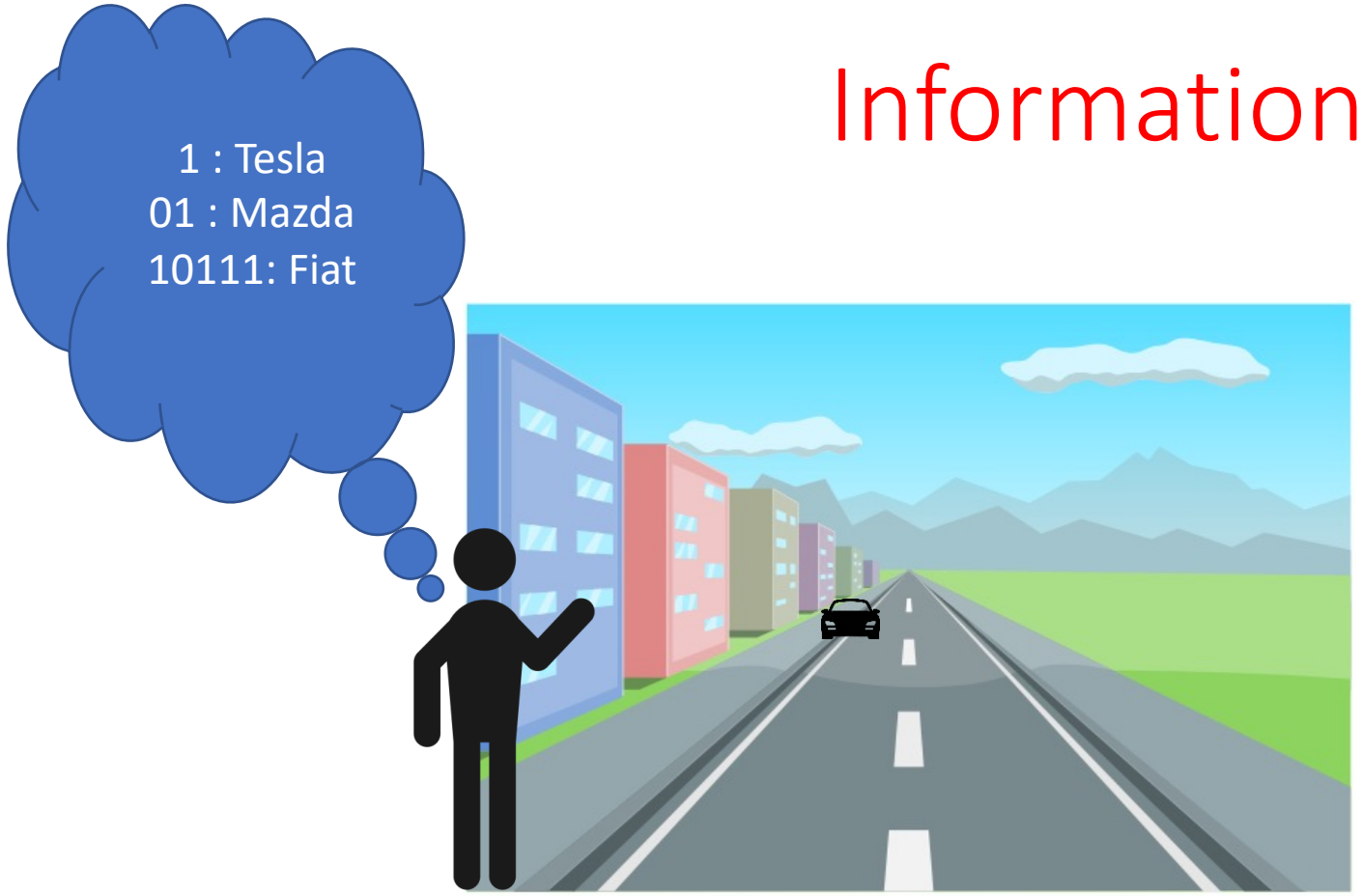
$$\log_2 \frac{1}{p_{coin}} = \log_2 \frac{1}{0.25} = 2$$

- Optimal number of bits to represent an event with probability p:

$$\log_2 \frac{1}{p}$$

H	H
H	T
T	H
T	T

Information Entropy



- Problem:
  - Transmit information about the labels of cars to another person with least the number of bits
- Assume that each bit is expensive
  - So, we are interested in the minimal/optimal coding

Example from: https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/

Information Entropy

- For an optimal setting, we can assign bits to code information based on their probabilities

- The smallest number of bits on avg. to represent an event with probability p: $\log_2 1/p$

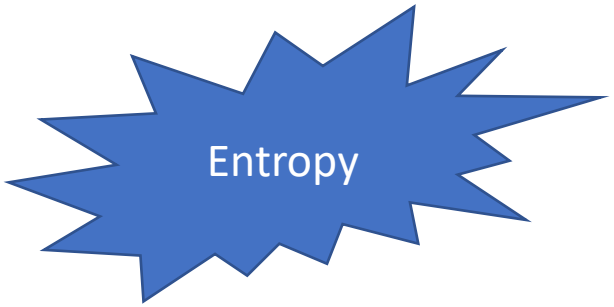
- Optimal # of bits to represent Fiat cars:

$$b_{\text{fiat}} = \log_2 \frac{1}{p_{\text{fiat}}}$$

- The optimal encoding then requires:

$$H(p) = E_p\left[\log_2 \frac{1}{p}\right] = \sum_i p_i \log_2 \frac{1}{p_i} = -\sum_i p_i \log_2 p_i$$

Car	Probability	# of bits
Fiat	0.80	0.32
Mazda	0.15	2.74
Tesla	0.05	4.32



Example from: https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/

Cross-entropy, Entropy

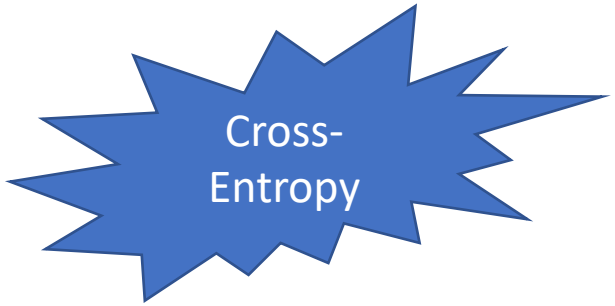
- Entropy assumes that the data follows the «correct» distribution.

Entropy:

$$H(p) = \sum_i p_i \log_2 \frac{1}{p_i} = -\sum_i p_i \log_2 p_i$$

- If the estimated/current distribution (call it q) is somewhat “wrong”, how can we quantify the number of bits required?

$$H(p,q) = E_p\left[\log_2 \frac{1}{q}\right] = \sum_i p_i \log_2 \frac{1}{q_i} = -\sum_i p_i \log_2 q_i$$



Example from: https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/

Kullback-Leibler Divergence

- Difference between cross-entropy and entropy (this is zero when p_i equals q_i):

$$KL(p \,||\, q) = \sum_i p_i \log \frac{1}{q_i} - \sum_i p_i \log \frac{1}{p_i}$$
$$= \sum_i p_i \log \frac{p_i}{q_i}$$

More on xentropy, entropy and KL-divergence

https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/

https://www.youtube.com/watch?v=ErfnhcEV1O8

Softmax classifier – Cross-entropy loss

- Cross-entropy: $H(p,q) = E_p\left[\log_2 \frac{1}{q}\right] = \sum_i p_i \log_2 \frac{1}{q_i} = -\sum_i p_i \log_2 q_i$
- In our case,
  - p denotes the correct probabilities of the categories. In other words, $p_j = 1$ for the correct label and $p_j = 0$ for other categories.
  - q denotes the estimated probabilities of the categories
- But, our scores are not probabilities!
  - One solution: Softmax function: $sm(s_i) = \frac{e^{s_i}}{\sum_j e^{s_j}}$
  - It maps arbitrary ranges to probabilities
- Using the normalized values, we can define the cross-entropy loss for classification problem now:

$$\mathcal{L}_i = -\log_e\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right) = -s_{y_i} + \log_e \sum_j e^{s_j}$$

Derive the gradients of NLL loss

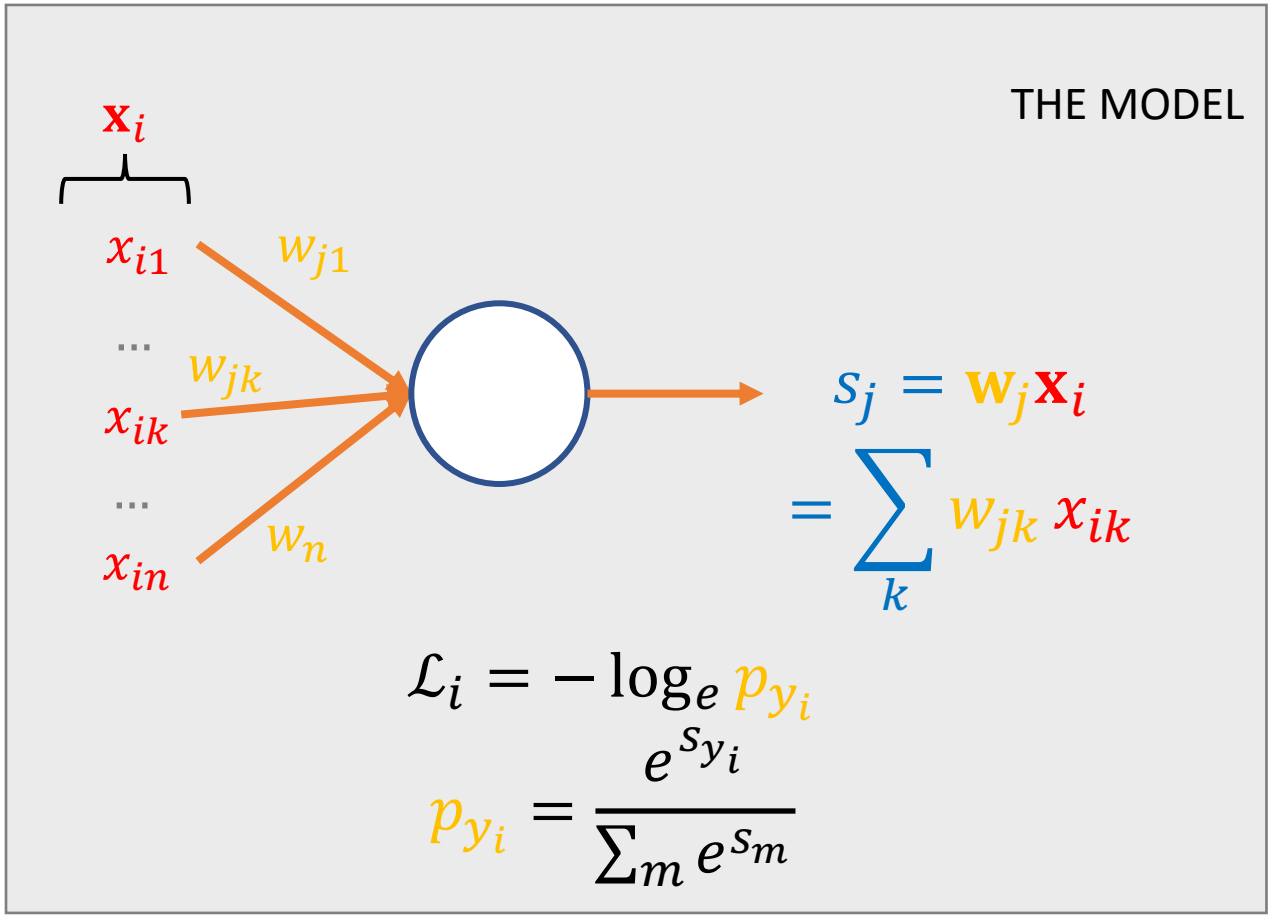
$$\frac{\partial \mathcal{L}_i}{\partial w_{jk}} = ?$$

If $j \neq y_i$:

$$\frac{\partial \mathcal{L}_i}{\partial w_{jk}} = \frac{\partial \mathcal{L}_i}{\partial p_{y_i}} \frac{\partial p_{y_i}}{\partial s_j} \frac{\partial s_j}{\partial w_{jk}}$$

$-\frac{1}{p_{y_i}}$ $\quad -p_{y_i} p_j$ $\quad x_{ik}$

$$\frac{\partial \mathcal{L}_i}{\partial w_{jk}} = p_j x_{ik}$$



This assumed that $j \neq y_i$.
What happens if that's not the case?
See the next page.

Derive the gradients of NLL loss

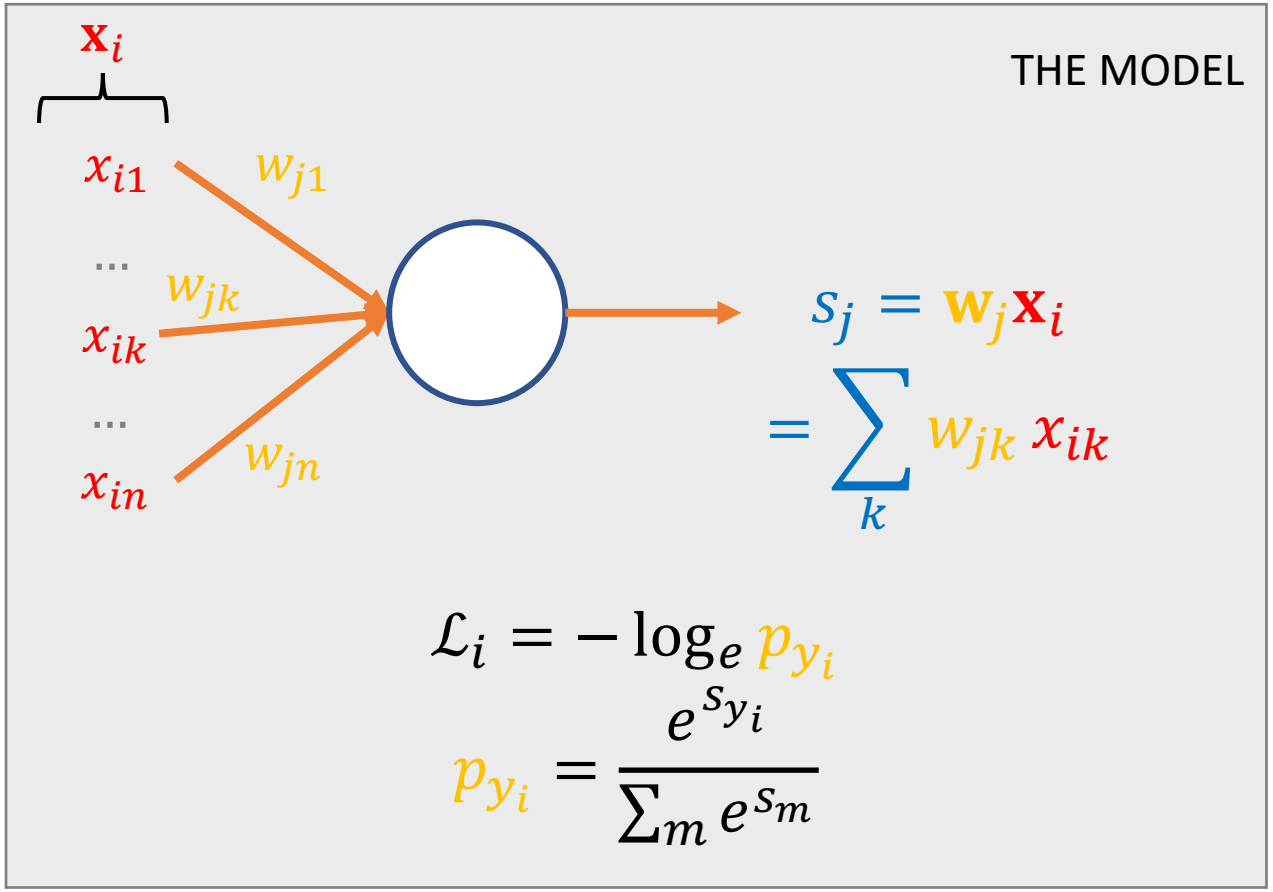
$$\frac{\partial \mathcal{L}_i}{\partial w_{jk}} = ?$$

If $j = y_i$:

$$\frac{\partial \mathcal{L}_i}{\partial w_{y_i k}} = \frac{\partial \mathcal{L}_i}{\partial p_{y_i}} \frac{\partial p_{y_i}}{\partial s_{y_i}} \frac{\partial s_{y_i}}{\partial w_{y_i k}}$$

$$-\frac{1}{p_{y_i}} \qquad p_{y_i}(1 - p_{y_i}) \qquad x_{ik}$$

$$\frac{\partial \mathcal{L}_i}{\partial w_{y_i k}} = (p_{y_i} - 1) x_{ik}$$



logistic loss

- A special case of cross-entropy for binary classification:

$$H(p,q) = -\sum_j p_j \log q_j = -p \log q - (1-p)\log(1-q)$$

- Softmax function reduces to the logistic function (see [1] for the derivation):

$$\frac{1}{1+e^{-x}}$$

Logistic regression!
(with a linear model)

[1] http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/

Softmax classifier: One interpretation

- Information theory
  - Cross-entropy between a true distribution and an estimated one:

$$H(p,q) = -\sum_{x} p(x) \log q(x)\,.$$

  - In our case, $p = [0, \ldots, 1, 0, \ldots 0]$, containing only one 1, at the correct label.
  - Since $H(p,q) = H(p) + D_{KL}(p||q)$, we are minimizing the Kullback-Leibler divergence.

$$D_{\mathrm{KL}}(P\|Q) = \sum_{i} P(i) \log \frac{P(i)}{Q(i)}.$$

http://cs231n.github.io/

Softmax classifier: Another interpretation

- Probabilistic view

$$P(y_i \mid x_i; W) = \frac{e^{s_{y_i}}}{\sum_j e^{s_j}}.$$

- In our case, we are minimizing the negative log likelihood.
- This corresponds to Maximum Likelihood Estimation (MLE).

http://cs231n.github.io/

Maximum Likelihood Estimation (MLE) vs Maximum A Posteriori Estimation (MAP)

Baye's Theorem:

Likelihood

$$p(\theta|Y) = \frac{p(Y|\theta)p(\theta)}{p(Y)}$$

Posterior

Since $p(Y)$ is constant:
$p(\theta|Y) \propto p(Y|\theta)p(\theta)$

Maximum Likelihood Estimation (MLE)

$$\theta_{MLE} \leftarrow \arg\max p(Y|\theta)$$
$$\leftarrow \arg\max \Pi_i \, p(y_i|\theta)$$

Taking the logarithm for numerical stability:

$$\theta_{MLE} \leftarrow \arg\max \log p(Y|\theta)$$
$$\leftarrow \arg\max \log \Pi_i \, p(y_i|\theta)$$
$$\leftarrow \arg\max \sum_i \log p(y_i \mid \theta)$$

Maximum A Posteriori Estimation (MAP)

$$\theta_{MAP} \leftarrow \arg\max \log p(\theta|Y)$$
$$\leftarrow \arg\max \log p(Y|\theta)p(\theta)$$
$$\leftarrow \arg\max \log p(Y|\theta) + \log p(\theta)$$
$$\leftarrow \arg\max \log \Pi_i \, p(y_i|\theta) + \log p(\theta)$$
$$\leftarrow \arg\max \sum_i \log p(y_i \mid \theta) + \log p(\theta)$$

For more details and derivations, see e.g.: https://wiseodd.github.io/techblog/2017/01/01/mle-vs-map/

Hinge loss vs. cross-entropy loss

- Hinge loss is “happy” (= zero) when the classification satisfies the margin
  - Ex: if score values = [10, 9, 9] or [10, -10, -10]
    - Hinge loss is “happy” if the margin is 1

- Cross-entropy is more ambitious: it wants more than a margin

More on softmax

- Softmax is a smooth version of arg max:

$$\arg\max\ (s_1, s_2, \dots, s_n) = (y_1, y_2, \dots, y_n) = (0,0,..,0,1,0..0)$$

- The base in softmax can be changed to have more "peaky" (or distributed) values for the largest input ($e^{\beta} = b$):

$$sm_{\beta}(s_i) = \frac{e^{\beta s_i}}{\sum_j e^{\beta s_j}}$$

- When $\beta \to \infty$, softmax converges to arg max.

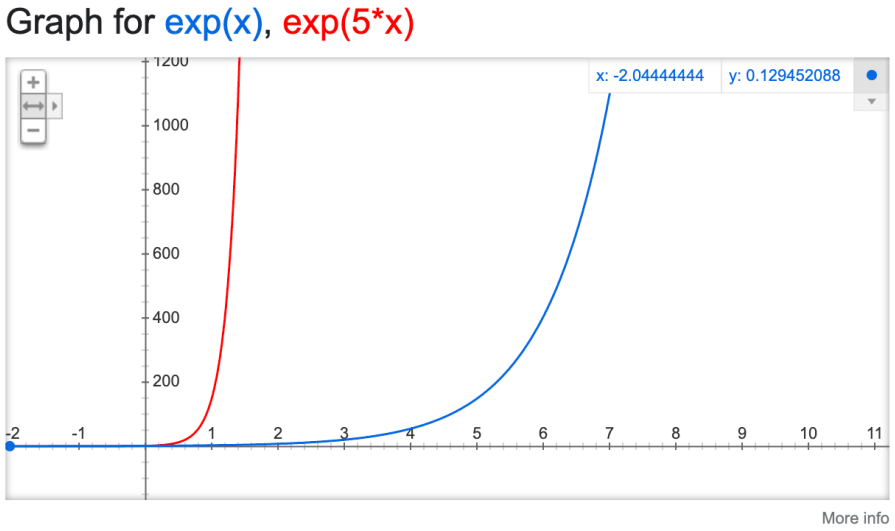
e.g.

$sm_{\beta=1}([1, 1.1]) = [0.475\ 0.524]$

$sm_{\beta=2}([1, 1.1]) = [0.451\ 0.550]$

$sm_{\beta=5}([1, 1.1]) = [0.378\ 0.622]$

$sm_{\beta=100}([1, 1.1]) = [4.5e{-}05\ 9.9e{-}01]$



More on softmax

- Softmax with temperature is softmax with $\beta = 1/T$:

$$sm_{1/T}(s_i) = \frac{e^{s_i/T}}{\sum_j e^{s_j/T}}$$

- Interpretation:
  - Increase T => decrease β => decrease the peak around the largest value.
- Lower T yields more confident (may be over confident) probability distribution.
- Especially in training sequence models where we perform sampling from the output distribution, in order to allow diversity, we can increase T.

More on softmax

- Exponentials may become very large. A trick:

$$\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} = \frac{Ce^{f_{y_i}}}{C\sum_j e^{f_j}} = \frac{e^{f_{y_i}+\log C}}{\sum_j e^{f_j+\log C}}$$

- Set $\log C = -\max_j f_j$.

See the following link for more information:

http://www.nowozin.net/sebastian/blog/streaming-log-sum-exp-computation.html

http://cs231n.github.io/

Classification Loss functions

- **A single correct** label case (classification):
  - Hinge loss:
    - $\mathcal{L}_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$
  - Cross-entropy (negative log-likelihood) loss:
    - $\mathcal{L}_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$

Classification Loss functions

- **Many correct** labels case:
  - Binary prediction for each label, independently:
    - $\mathcal{L}_i = \sum_j \max(0, 1 - y_{ij} s_j)$
    - $y_{ij} = +1$ if example i is labeled with label j; otherwise $y_{ij} = -1$.
  - Alternatively, train binary Cross Entropy (logistic) loss for each label (0 or 1):

$$\mathcal{L}_i = \sum_j \left[y_{ij} \log\left(\sigma(s_j)\right) + \left(1 - y_{ij}\right) \log\left(1 - \sigma(s_j)\right) \right]$$

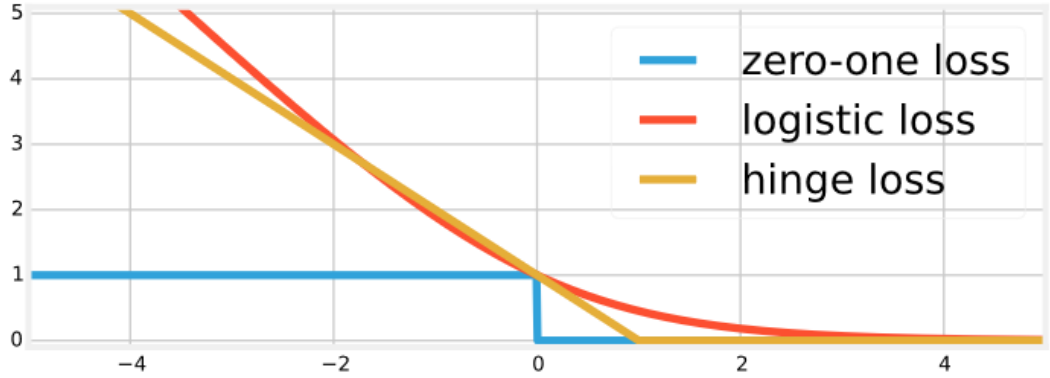
0-1 Loss

- Minimize the # of cases where the prediction is wrong:

$$\mathcal{L} = \sum_{i} \mathbb{I}\left(f(\mathbf{x}_i; W, b)_{y_i} \neq y_i\right)$$

Or equivalently,

$$\mathcal{L} = \sum_{i} \mathbb{I}\left(y_i f(\mathbf{x}_i; W, b)_{y_i} < 0\right)$$



$$t = y_i f(\mathbf{x}_i; W, b)_{y_i}$$

Figure source: https://fa.bianp.net/talks/trento_may_2015/#/step-6

Absolute Value Loss, Squared Error Loss

$$\mathcal{L}_i = \sum_j |s_j - y_j|^q$$

- $q = 1$: absolute value or L1 loss.
- $q = 2$: square error or L2 loss.

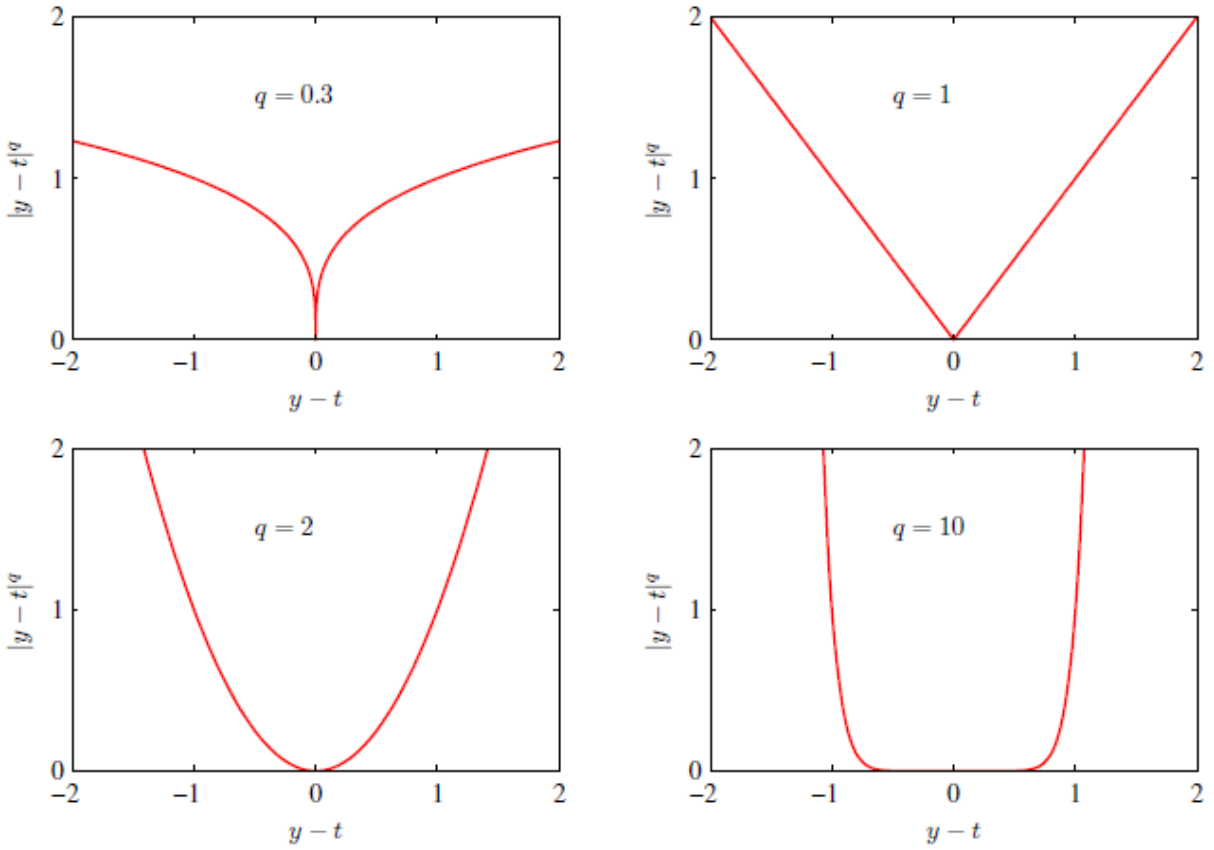


Figure 1.29 Plots of the quantity $L_q = |y - t|^q$ for various values of q.

Bishop

Structured Loss functions

- What if we want to predict a graph, tree etc.? Something that has structure.
  - Structured loss: formulate loss such that you minimize the distance to a correct structure

Visualizing Loss Functions

- If you look at one of the example loss functions:

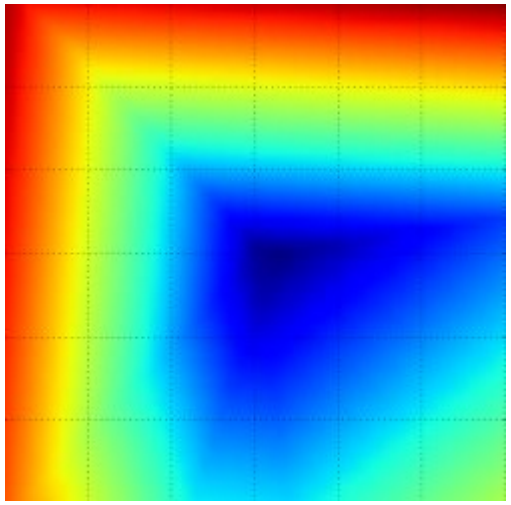
$$\mathcal{L}_i = \sum_{j \neq y_i} \max(0, \mathbf{w}_j^T \mathbf{x}_i - \mathbf{w}_{y_i}^T \mathbf{x}_i + 1)$$

- Since W has too many dimensions, this is difficult to plot.
- We can visualize this for one weight direction though, which can give us some intuition about the shape of the function.
  - E.g., start from an arbitrary W_0, choose a direction W_1 and plot $\mathcal{L}(W_0 + \alpha W_1)$ for different values of α.

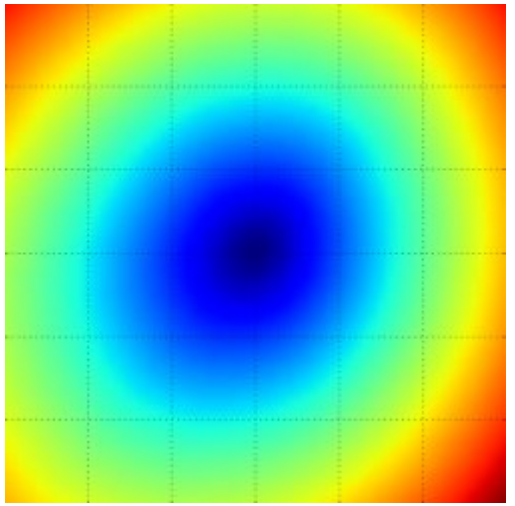
http://cs231n.github.io/

Visualizing Loss Functions

Loss along one direction



Loss along two directions



Loss along two directions
(averaged over many samples)

- You see that this is a convex function.
  - Nice and easy for optimization
- When you combine many of them in a neural network, it becomes non-convex.

Another approach for visualizing loss functions

- 0-1 loss:

$$\mathcal{L} = \mathbb{I}(f(x) \neq y)$$

or equivalently as:

$$\mathcal{L} = \mathbb{I}(yf(x) < 0)$$

- Square loss:

$$\mathcal{L} = (f(x) - y)^2$$

in binary case:

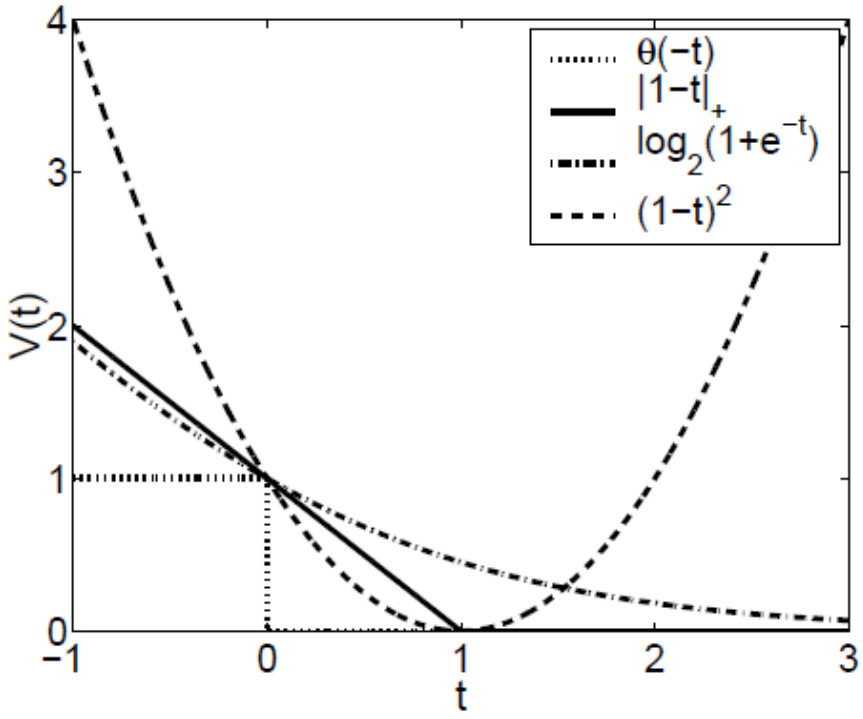
$$\mathcal{L} = \left(1 - yf(x)\right)^2$$

- Hinge-loss

$$\mathcal{L} = \max(1 - yf(x), 0)$$

- Logistic loss (binary Cross Entropy Loss):

$$\mathcal{L} = -\log_2\left(\frac{1}{1 + e^{-yf(x)}}\right)$$

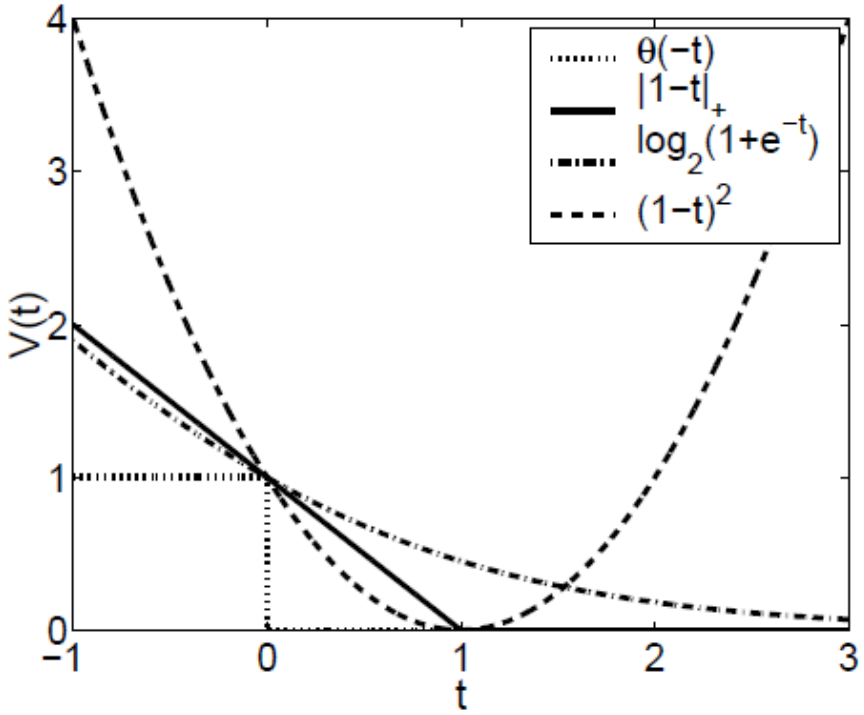


Various loss functions used in classification. Here $t = yf(\mathbf{x})$.

Rosacco et al., 2003

https://web.mit.edu/lrosasco/www/publications/loss.pdf

All losses approximate 0-1 loss



Various loss functions used in classification. Here $t = yf(\mathbf{x})$.

Rosacco et al., 2003

Loss Functions: Sum up

- 0-1 loss is not differentiable/helpful at training
  - It is used in testing
- Other losses try to cover the “weakness” of 0-1 loss
- Hinge-loss imposes weaker constraint compared to cross-entropy
- For classification: use hinge-loss or cross-entropy loss
- For regression: use squared-error loss, or absolute difference loss

Activation Functions

Activation function: Sigmoid/logistic

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Output is in range (0,1)
- Since it maps a large domain to (0,1) it is also called squashing function
- Simple derivative

$$\frac{d\sigma(x)}{dx} = \sigma(x) \cdot \big(1 - \sigma(x)\big)$$

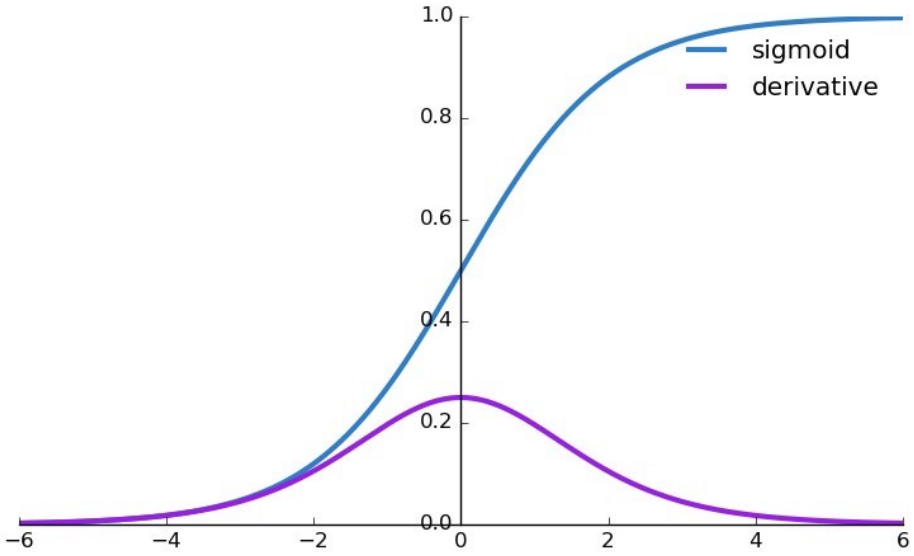


Fig: https://medium.com/@omkar.nallagoni/activation-functions-with-derivative-and-python-code-sigmoid-vs-tanh-vs-relu-44d23915c1f4

Activation function: tanh

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

- Output is in range (-1,1)
- A squashing function
- Simple derivative

$$\frac{d\tanh(x)}{dx} = (1 - \tanh^{2}(x))$$

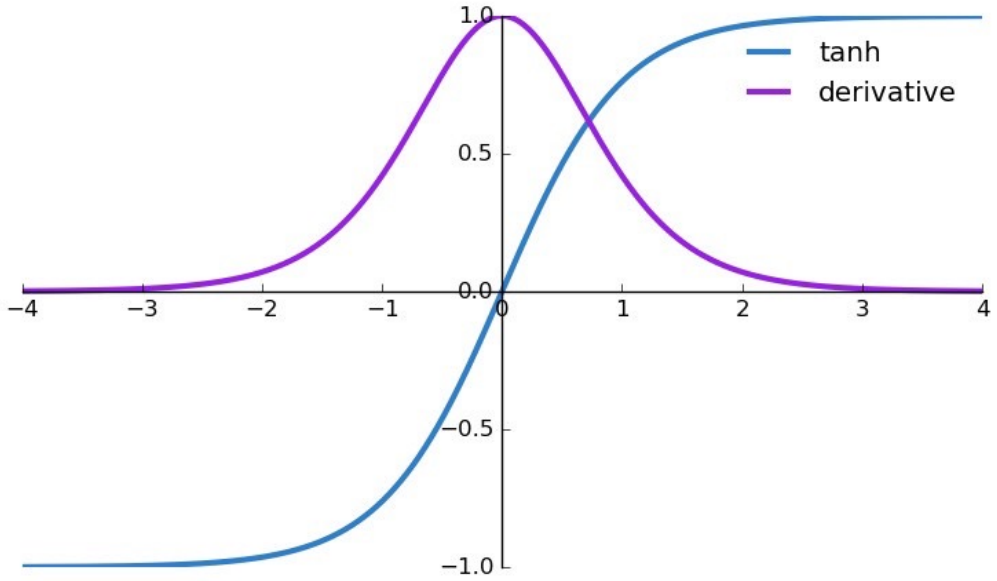


Fig: https://medium.com/@omkar.nallagoni/activation-functions-with-derivative-and-python-code-sigmoid-vs-tanh-vs-relu-44d23915c1f4

Activation Functions: Sigmoid vs. tanh

- Sigmoid is a historically important activation function
  - But nowadays, rarely used
  - Drawbacks:
    1. It gets saturated, if the activation is close to zero or one
       - This leads to very small gradient, which affects the feedback to earlier layers
       - Initialization is also very important for this reason
    2. It is not zero-centered (not very severe)

They are both non-convex!

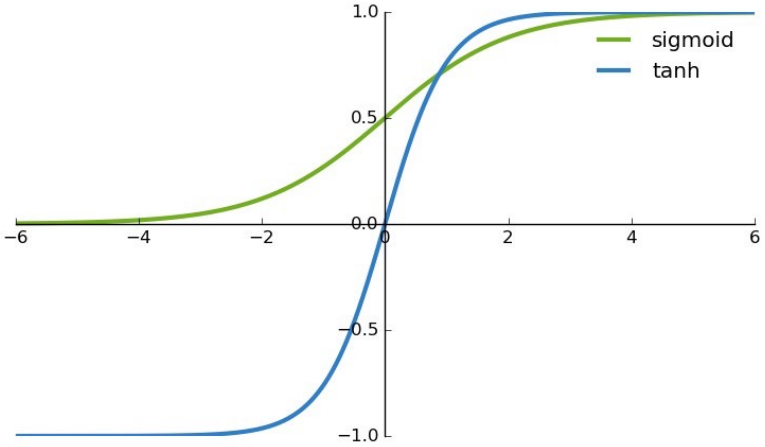


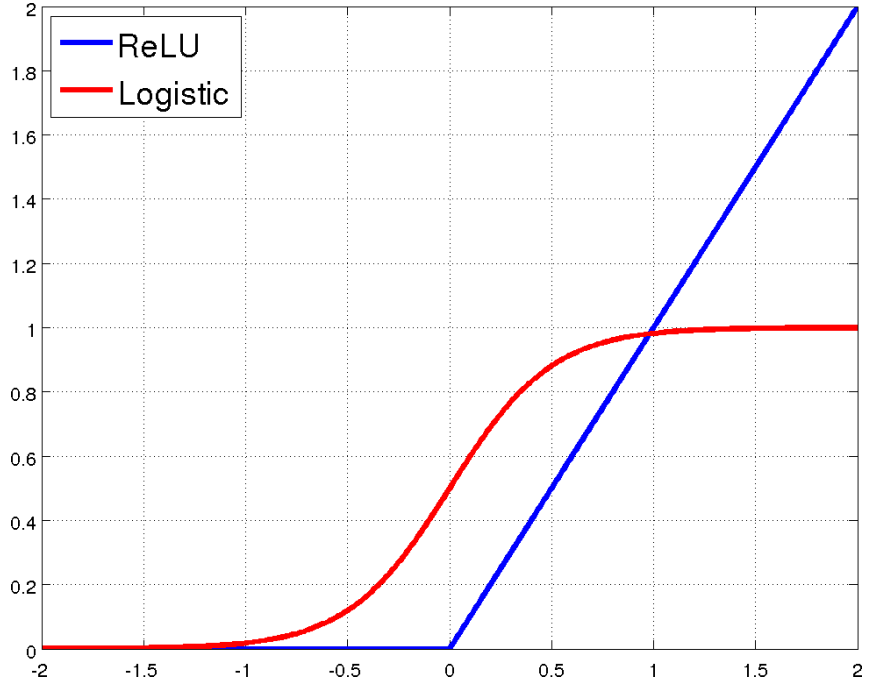
Fig: https://medium.com/@omkar.nallagoni/activation-functions-with-derivative-and-python-code-sigmoid-vs-tanh-vs-relu-44d23915c1f4

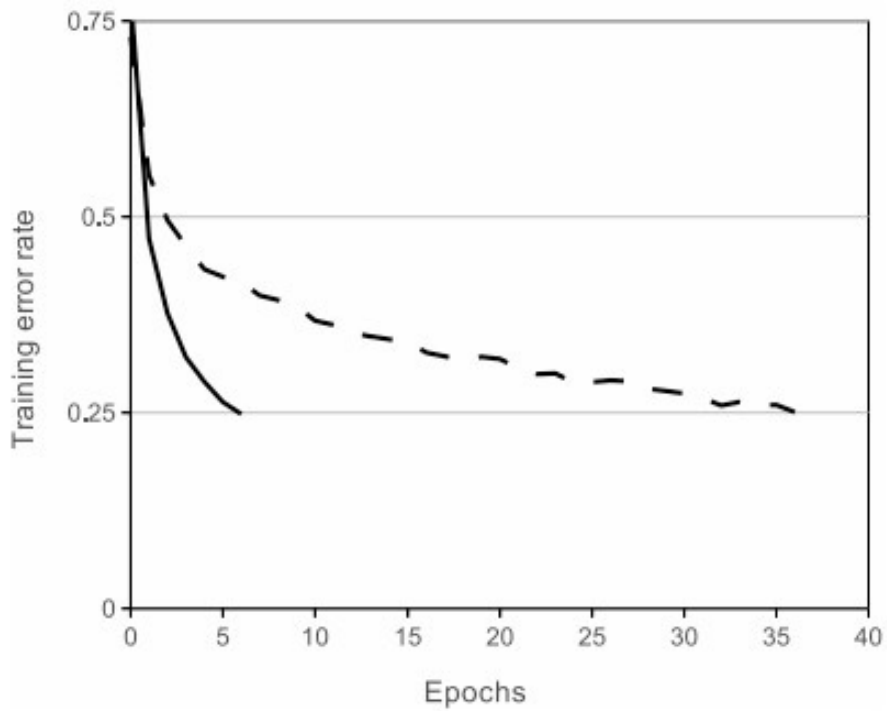
- Tanh
  - Similar to the sigmoid, it saturates
  - However, it is zero-centered.
  - Tanh is generally preferred over sigmod
  - Note: $\tanh(x) = 2\sigma(2x) - 1$

http://cs231n.github.io/neural-networks-1/

Activation Functions: Rectified Linear Units (ReLU)

Vinod Nair and Geoffrey Hinton (2010). Rectified linear units improve restricted Boltzmann machines, ICML.





[Krizhevsky et al., NIPS12]

$$f(x) = \max(0, x)$$

Derivative: $\mathbf{1}(x > 0)$

Activation Functions:
ReLU – biological motivation

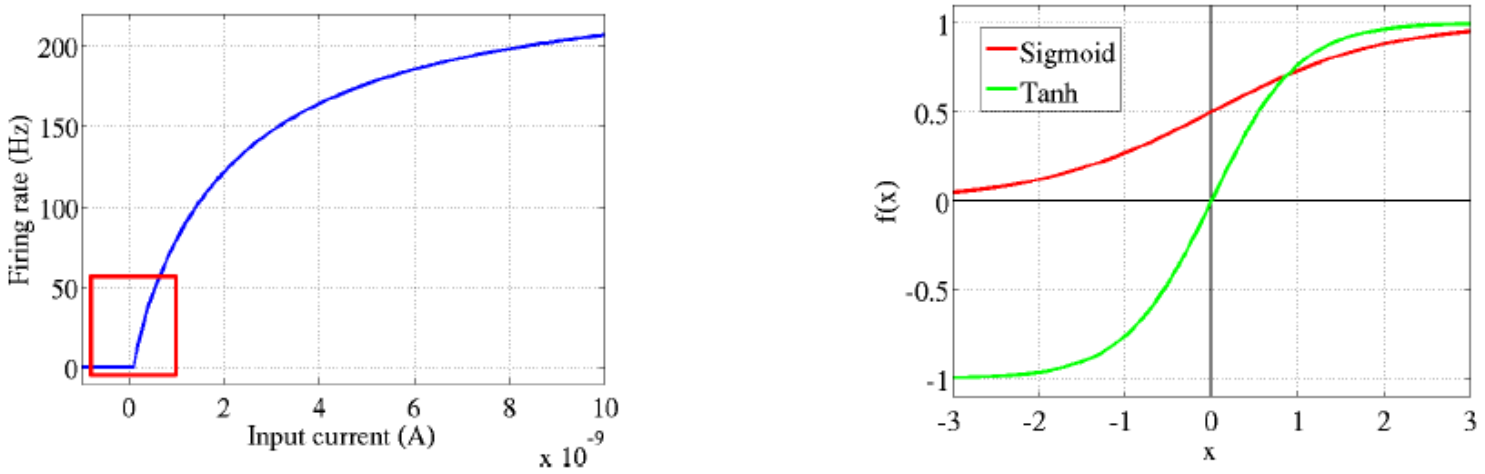
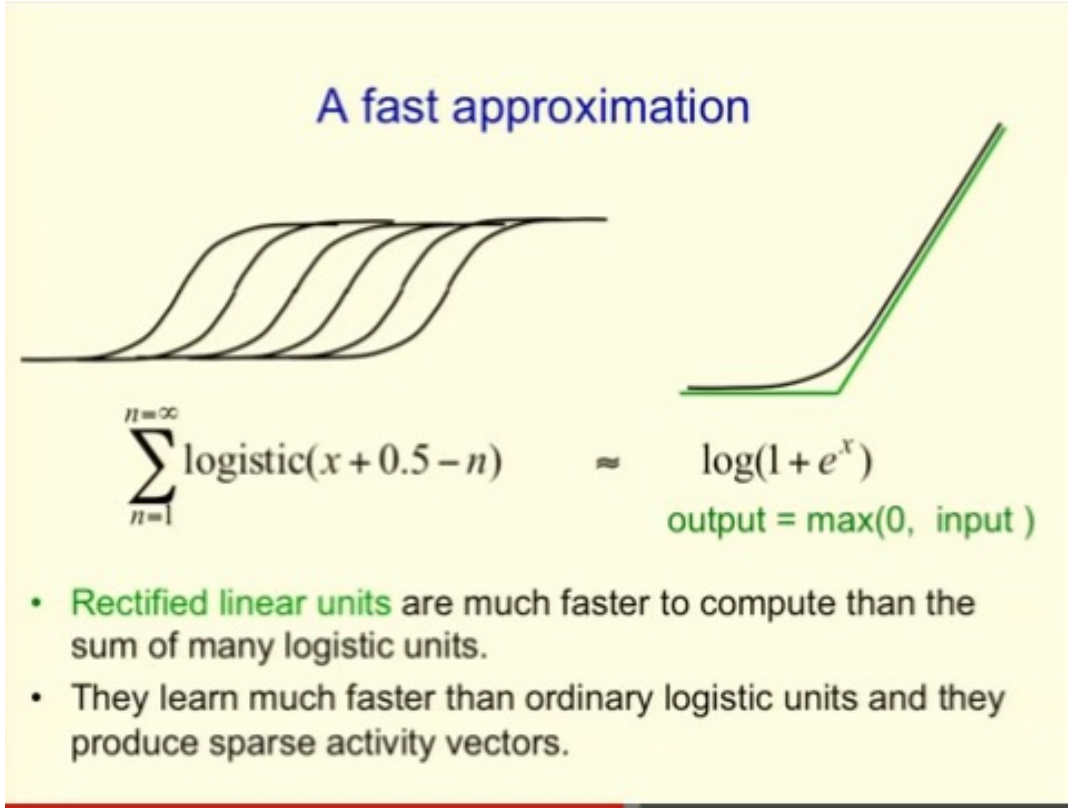


Figure 1: *Left:* **Common neural activation function motivated by biological data.** *Right:* **Commonly used activation functions** in neural networks literature: logistic sigmoid and hyperbolic tangent (*tanh*).

Glorot et al., "Deep Sparse Rectifier Neural Networks", 2011.

Activation Functions: ReLU – biological motivation

Hinton argues that this is a form of model averaging



Activation Functions: ReLU: Pros and Cons

- Pros:
  - It converges much faster (claimed to be 6x faster than sigmoid/tanh)
    - It overfits very fast and when used with e.g. dropout, this leads to very fast convergence
  - It is simpler and faster to compute as it performs a simple comparison with zero

- Cons:
  - A ReLU neuron may “die” during training
  - A large gradient may update the weights such that the ReLU neuron may never activate again
    - Avoid large learning rate

- See also:

  http://www.jefkine.com/general/2016/08/24/formulating-the-relu/

Activation Functions: Leaky ReLU

Andrew L. Maas, Awni Y. Hannun, Andrew Y. Ng (2014). Rectifier Nonlinearities Improve Neural Network Acoustic Models

- $f(x) = \mathbf{1}(x < 0)(\alpha x) + \mathbf{1}(x \geq 0)(x)$
  - When x is negative, Leaky ReLU has a non-zero slope (α)

- If you learn α during training, this is called parametric ReLU (PReLU)

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun (2015) Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

Activation Functions: maxout

"Maxout Networks" by Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, Yoshua Bengio, 2013.

- $\max(w_1^T x + b_1, w_2^T x + b_2)$

- ReLU, Leaky ReLU and PReLU are special cases of this
- Drawback: More parameters to learn!

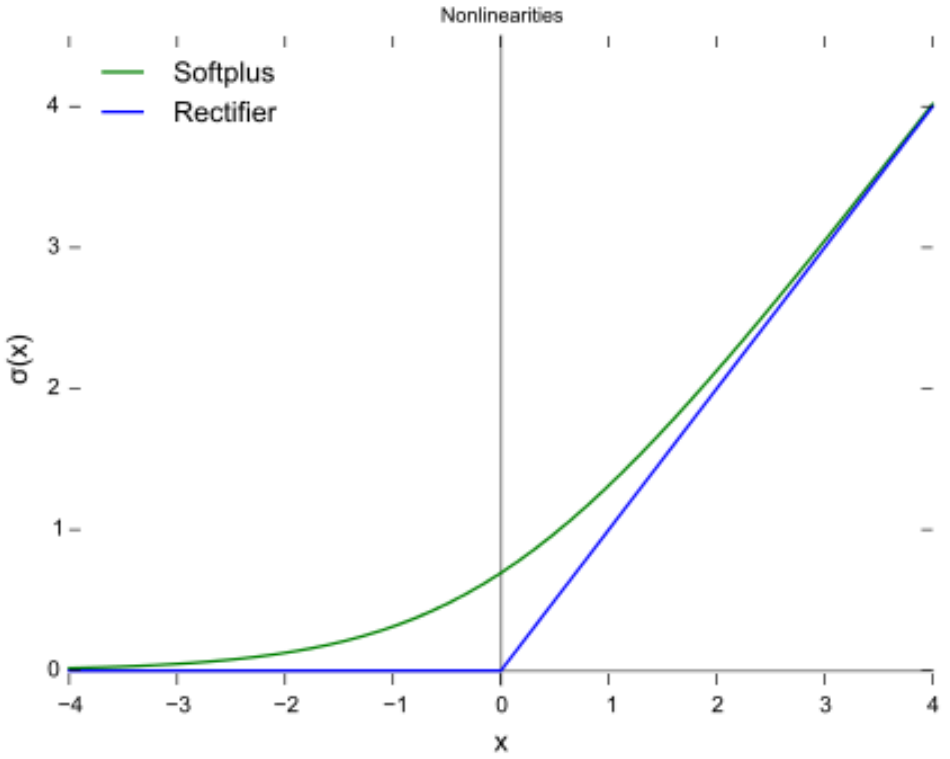
Activation Functions: Softplus

- A smooth approximation to the ReLU unit:

$$f(x) = \ln(1 + e^x)$$

- Its derivative is the sigmoid function:

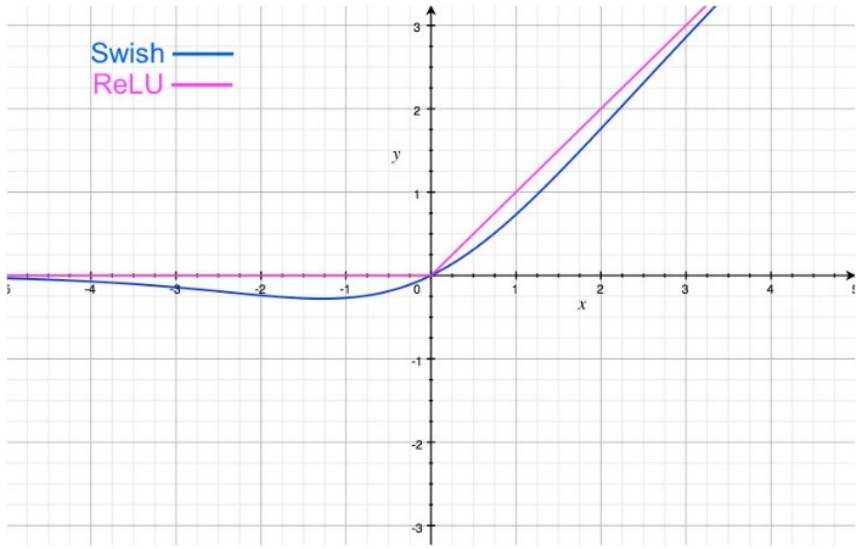
$$f'(x) = 1/(1 + e^{-x})$$



Activation Functions:
Swish: A Self-Gated Activation Function

SEARCHING FOR ACTIVATION FUNCTIONS

Prajit Ramachandran*, Barret Zoph, Quoc V. Le
Google Brain
{prajit,barretzoph,qvl}@google.com



"The choice of activation functions in deep networks has a significant effect on the training dynamics and task performance. Currently, the most successful and widely-used activation function is the Rectified Linear Unit (ReLU). Although various alternatives to ReLU have been proposed, none have managed to replace it due to inconsistent gains. In this work, we propose a new activation function, named Swish, which is simply $f(x)=x \cdot sigmoid(x)$. *Our experiments show that Swish tends to work better than ReLU on deeper models across a number of challenging datasets. For example, simply replacing ReLUs with Swish units improves top-1 classification accuracy on ImageNet by 0.9% for Mobile NASNet-A and 0.6% for Inception-ResNet-v2. The simplicity of Swish and its similarity to ReLU make it easy for practitioners to replace ReLUs with Swish units in any neural network."*

Activation Functions: Exponential Linear Unit

- Similar to the Swish function

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

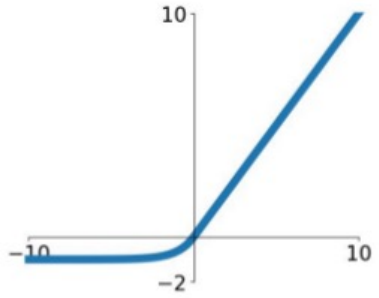
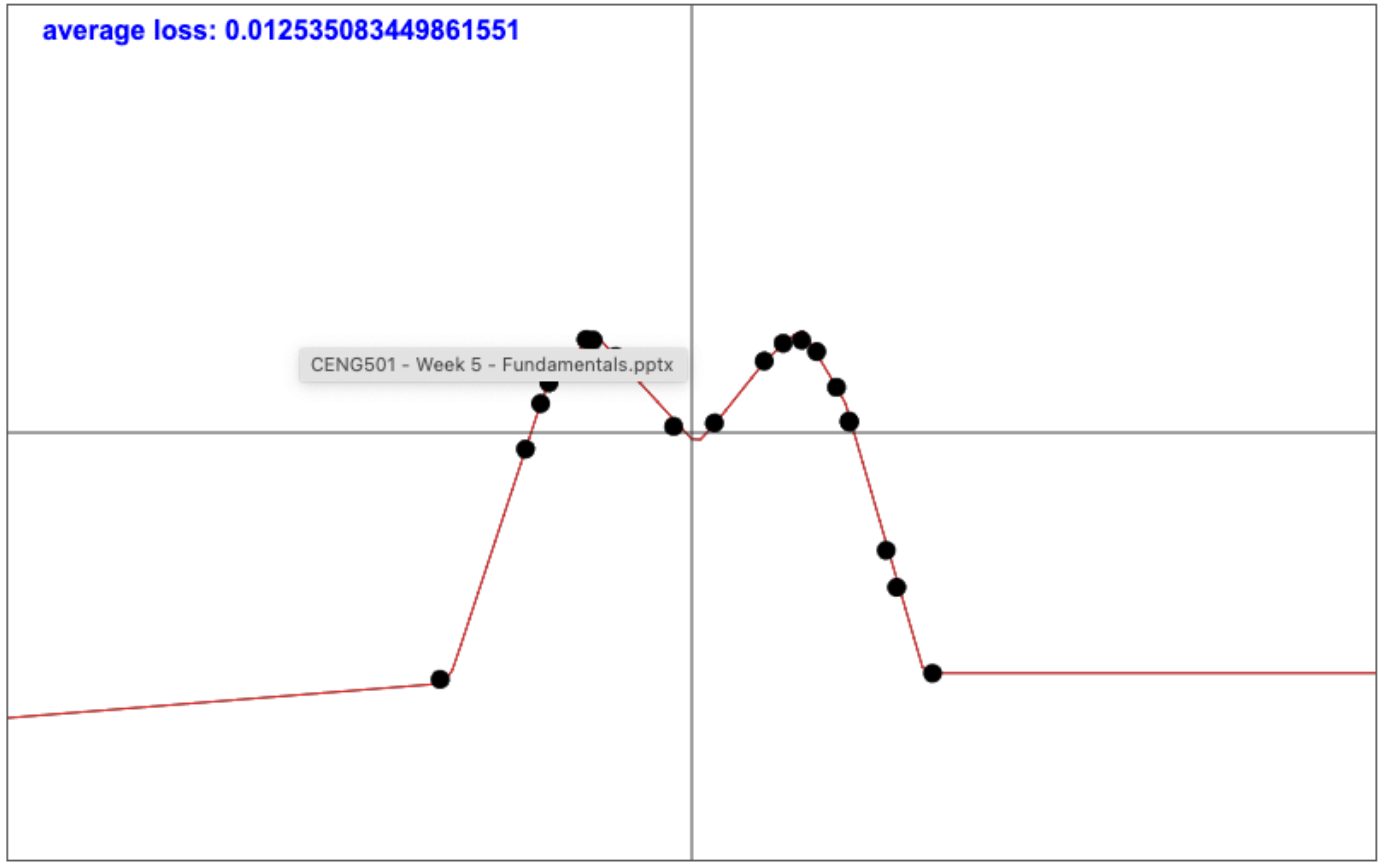


Fig: https://medium.com/@krishnakalyan3/introduction-to-exponential-linear-unit-d3e2904b366c

Activation Functions: To sum up

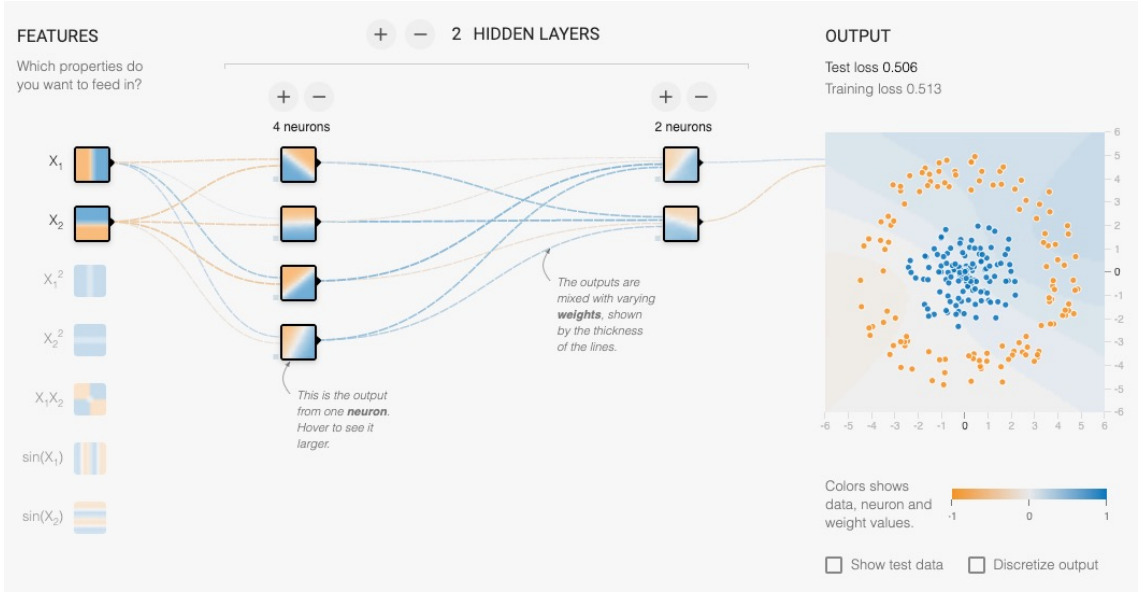
- Don’t use sigmoid
- If you really want to, use tanh but it is worse than ReLU and its variants
- ReLU: be careful about dying neurons
- Leaky ReLU and Maxout: Worth trying

DEMO 1



https://cs.stanford.edu/people/karpathy/convnetjs/demo/regression.html

DEMO 2



http://playground.tensorflow.org/#activation=tanh&regularization=L2&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=4,2&seed=0.24725&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification

Interactive introductory tutorial

https://jalammar.github.io/visual-interactive-guide-basics-neural-networks/

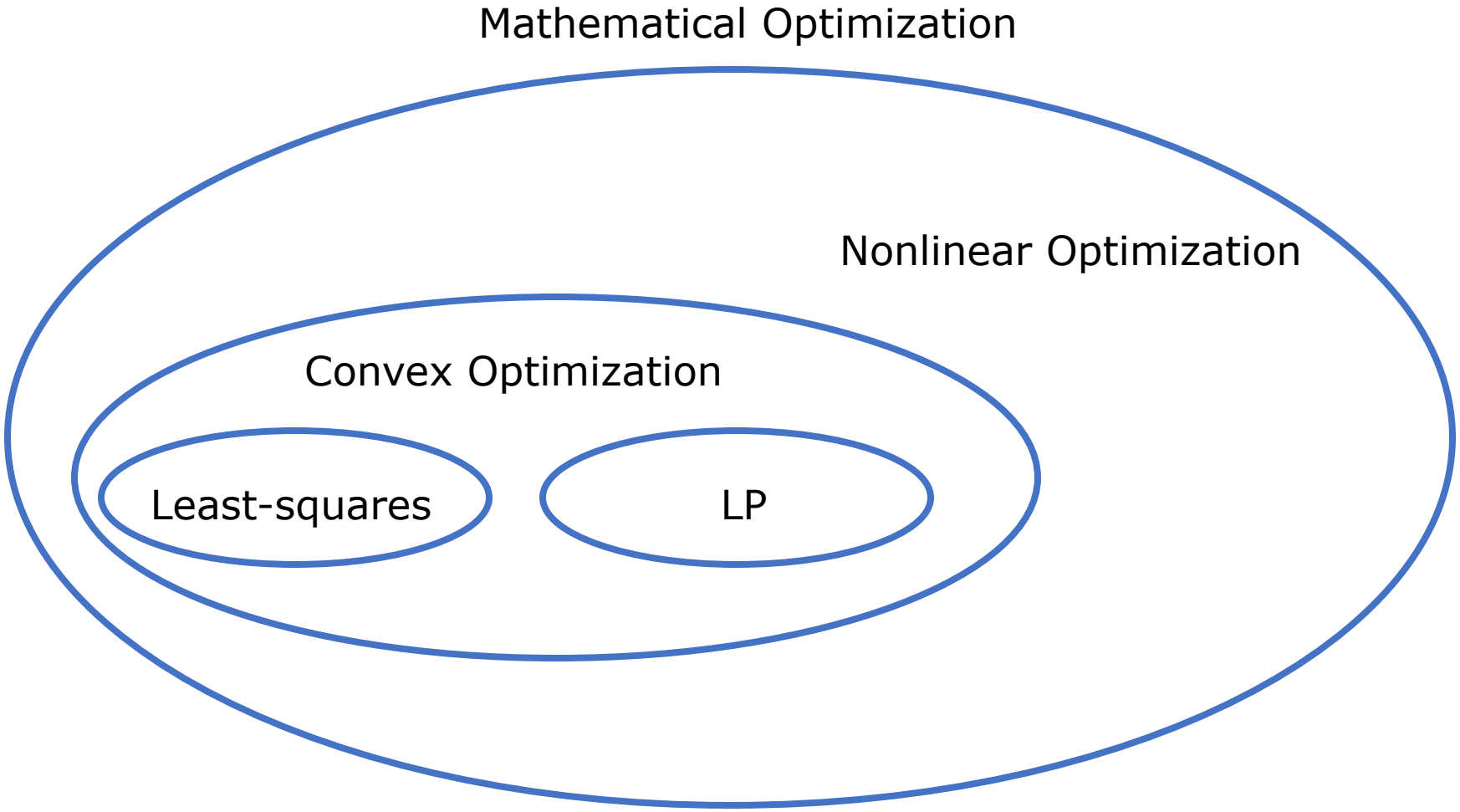
Courses/tutorials:

- METU IAM771: Optimization Methods for Machine Learning
  https://catalog.metu.edu.tr/course.php?course_code=9700771

- EPFL: Optimization for Machine Learning:
  https://github.com/epfml/OptML_course

- Optimization Algorithms in Machine Learning:
  http://videolectures.net/nips2010_wright_oaml/

A General Look at Optimization


Mathematical Optimization
Nonlinear Optimization
Convex Optimization
Least-squares
LP

Rong Jin

Mathematical Optimization

(mathematical) optimization problem

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq b_i, \quad i = 1, \ldots, m \end{array}$$

- $x = (x_1, \ldots, x_n)$: optimization variables
- $f_0 : \mathbf{R}^n \to \mathbf{R}$: objective function
- $f_i : \mathbf{R}^n \to \mathbf{R}$, $i = 1, \ldots, m$: constraint functions

optimal solution $x^\star$ has smallest value of f_0 among all vectors that satisfy the constraints

Convex Optimization

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq b_i, \quad i = 1, \ldots, m \end{array}$$

- objective and constraint functions are convex:

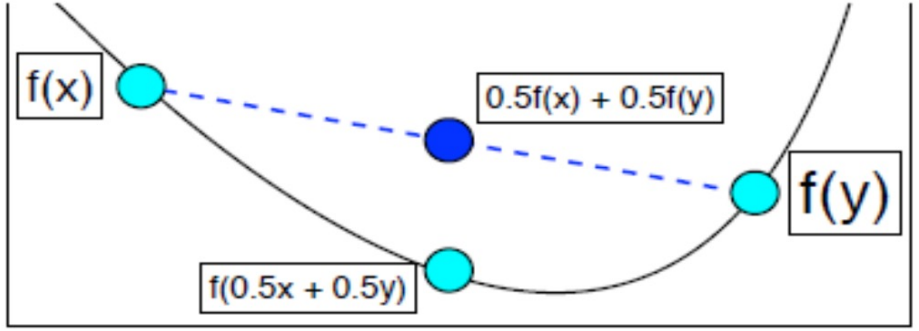
$$f_i(\alpha x + \beta y) \leq \alpha f_i(x) + \beta f_i(y)$$

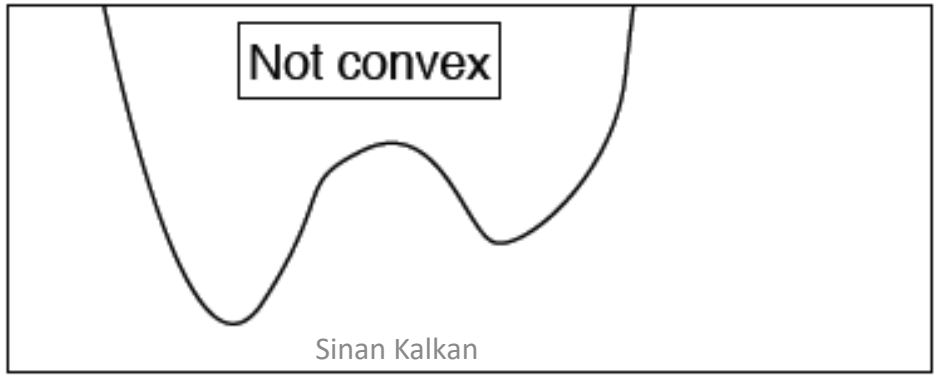
  if $\alpha + \beta = 1$, $\alpha \geq 0$, $\beta \geq 0$

- includes least-squares problems and linear programs as special cases

Interpretation

- Function's value is below the line connecting two points





Sinan Kalkan

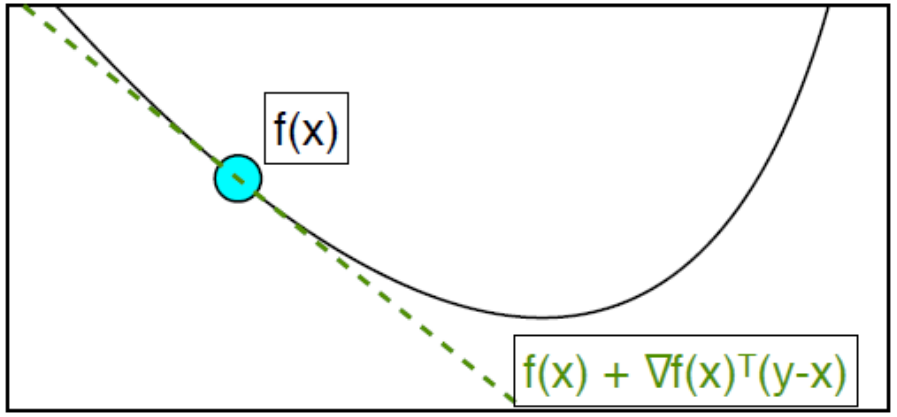
Mark Schmidt

Another interpretation

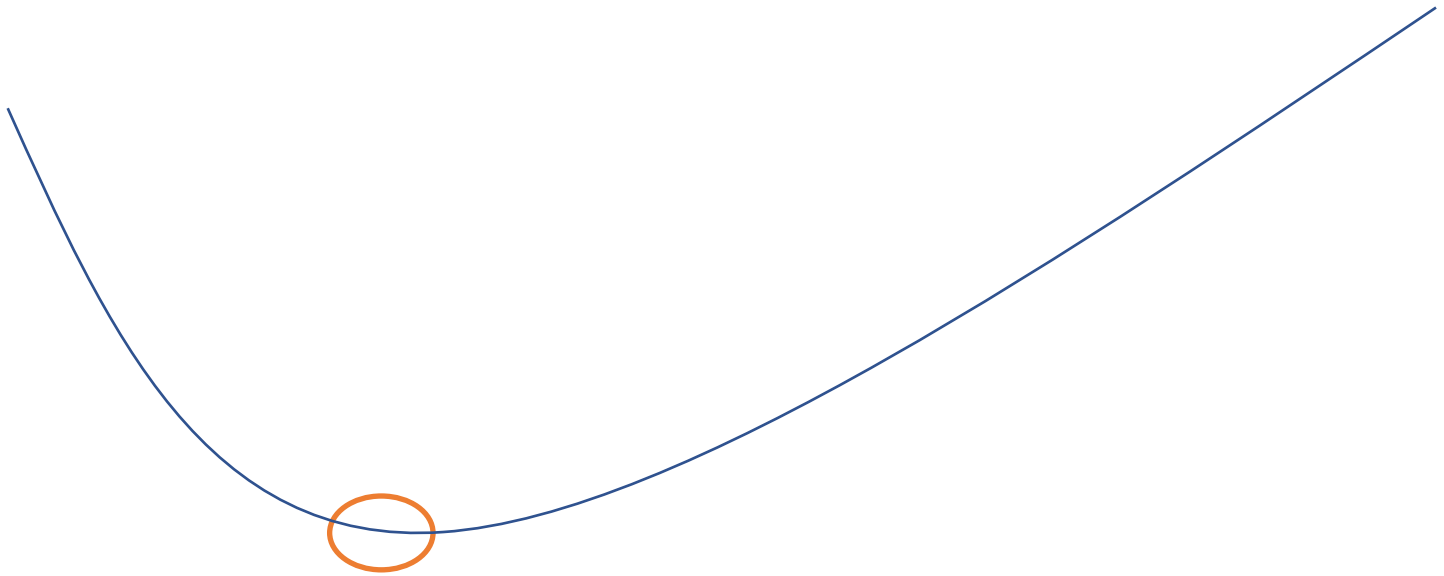
A differentiable function f is convex if for all x and y we have

$$f(y) \geq f(x) + \nabla f(x)^T (y - x),$$

- The function is globally above the tangent at x.

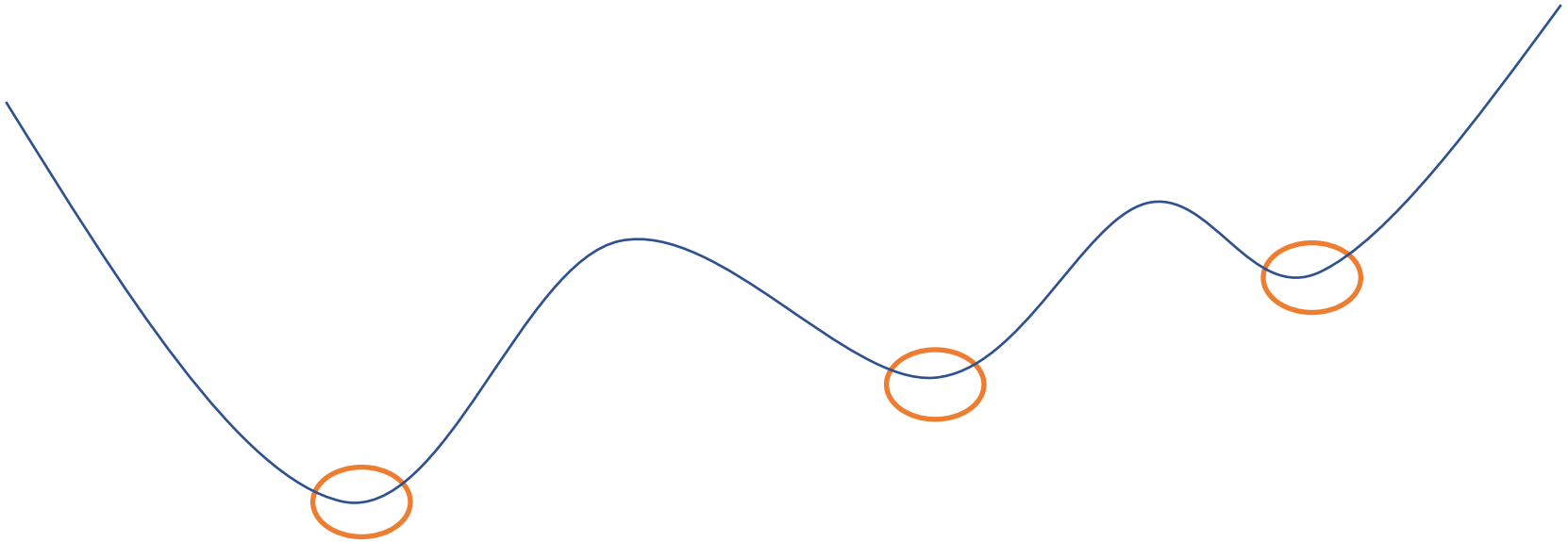


Convex vs. Non-convex Ex.



Convex => Easy to minimize

Convex vs. Non-convex Ex.



- Non-convex => Local minima => Easy to get stuck in a local min.
- **Can’t rely on only local search techniques**

Example convex functions

Some simple convex functions:

- $f(x) = c$
- $f(x) = a^T x$
- $f(x) = x^T A x$ (for $A \succeq 0$)
- $f(x) = \exp(ax)$
- $f(x) = x \log x$ (for $x > 0$)
- $f(x) = \|x\|^2$
- $f(x) = \|x\|_p$
- $f(x) = \max_i \{x_i\}$

Some other notable examples:

- $f(x, y) = \log(e^x + e^y)$
- $f(X) = \log\det X$ (for X positive-definite).
- $f(x, Y) = x^T Y^{-1} x$ (for Y positive-definite)

Operations that conserve convexity

1. Non-negative weighted sum:

$$f(x) = \theta_1 f_1(x) + \theta_2 f_2(x).$$

2. Composition with affine mapping:

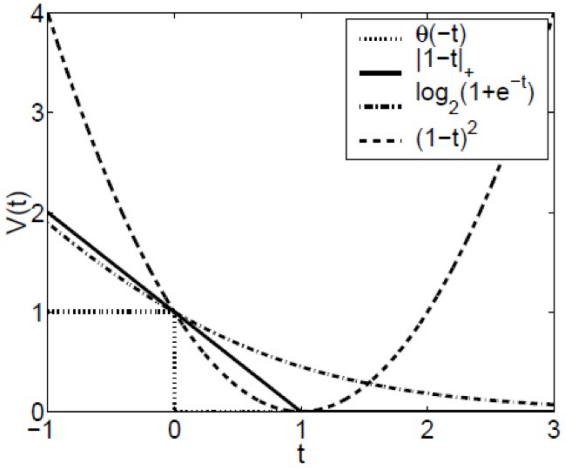
$$g(x) = f(Ax + b).$$

3. Pointwise maximum:

$$f(x) = \max_i \{f_i(x)\}.$$

Deep learning functions

- Wx - convex
- ReLU – convex
- Softmax – non-convex
  - Log-sum-exp (normalization of softmax) – convex
- Sigmoid, tanh – non-convex
- Loss functions (cross-entropy, max-margin, squared-error loss) are convex
- How about NNs?
  - NNs without non-linearities are convex. The parameters just model a hyper-plane.
  - NNs with non-linearities are NOT convex.



Various loss functions used in classification. Here $t = yf(\mathbf{x})$.

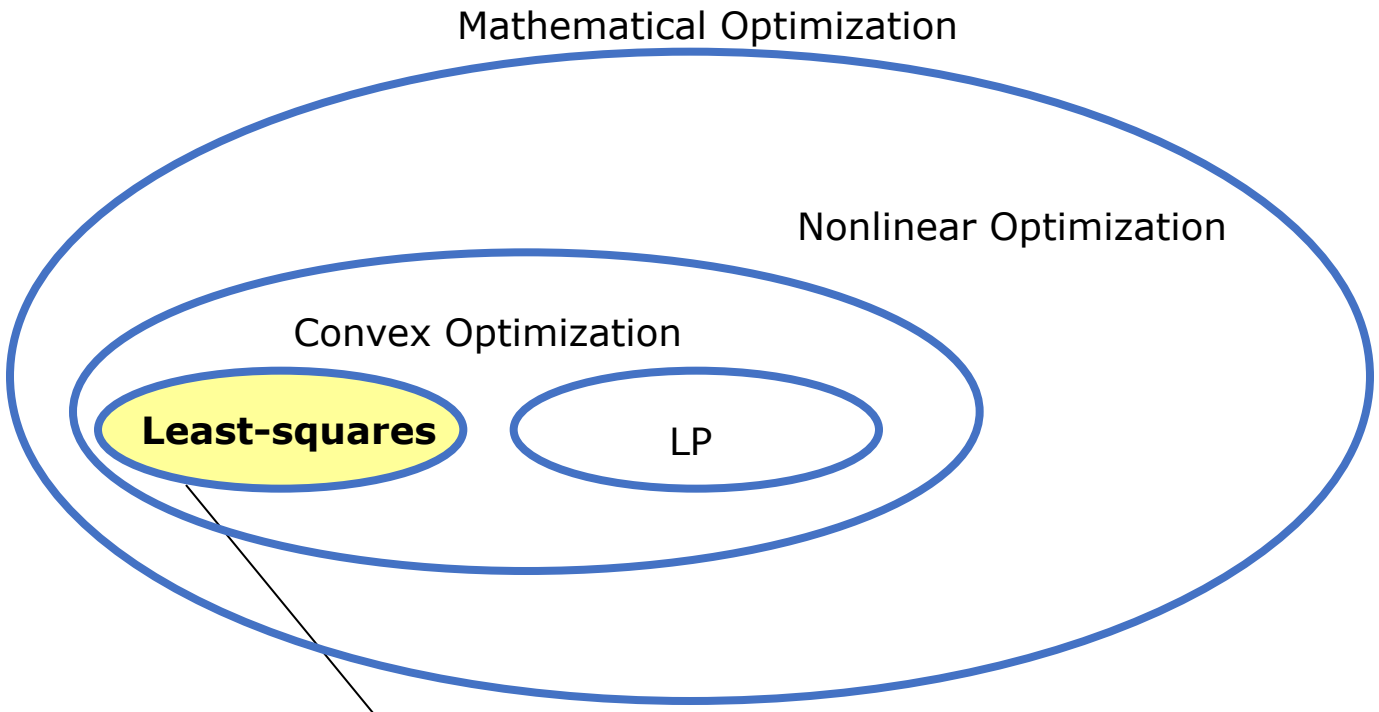
Rosacco et al., 2003
https://web.mit.edu/lrosasco/www/publications/loss.pdf

Why convex optimization?

- Can’t solve most OPs
  - E.g. NP Hard, too slow
- Convex OPs
  - (Generally) No analytic solution
  - Efficient iterative algorithms to find (global) solution
- Easy to see why convexity allows for efficient solution
  - Just “slide” down the objective function as far as possible and will reach a minimum

Non-convex Problems

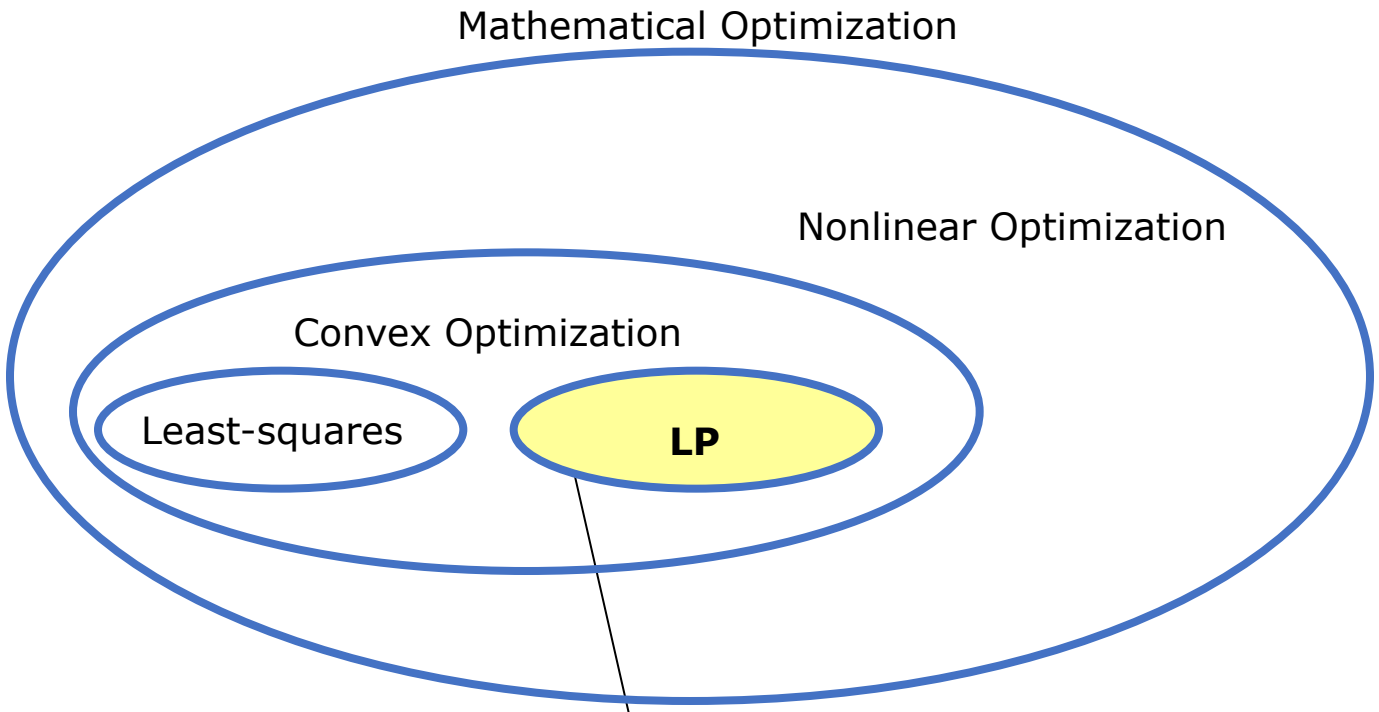
- Some non-convex problems highly multi-modal, or NP hard
- Could be forced to search all solutions, or hope stochastic search is successful
- Cannot guarantee best solution, inefficient
- Harder to make performance guarantees with approximate solutions



minimize $\|Ax - b\|_2^2$

- Analytical solution
- Good algorithms and software
- High accuracy and high reliability

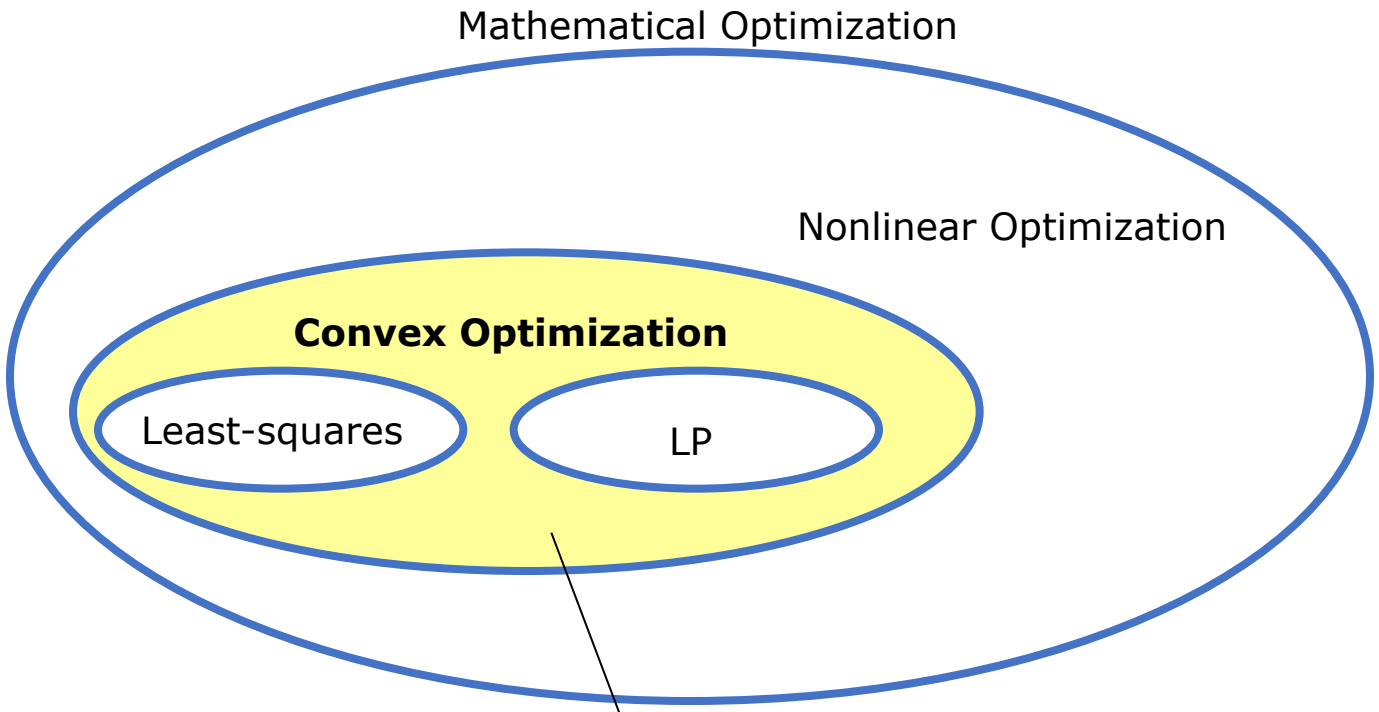
A mature technology!



$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & a_i^T x \leq b_i \\ & i = 1, \ldots, m \end{array}$$

- No analytical solution
- Algorithms and software
- Reliable and efficient

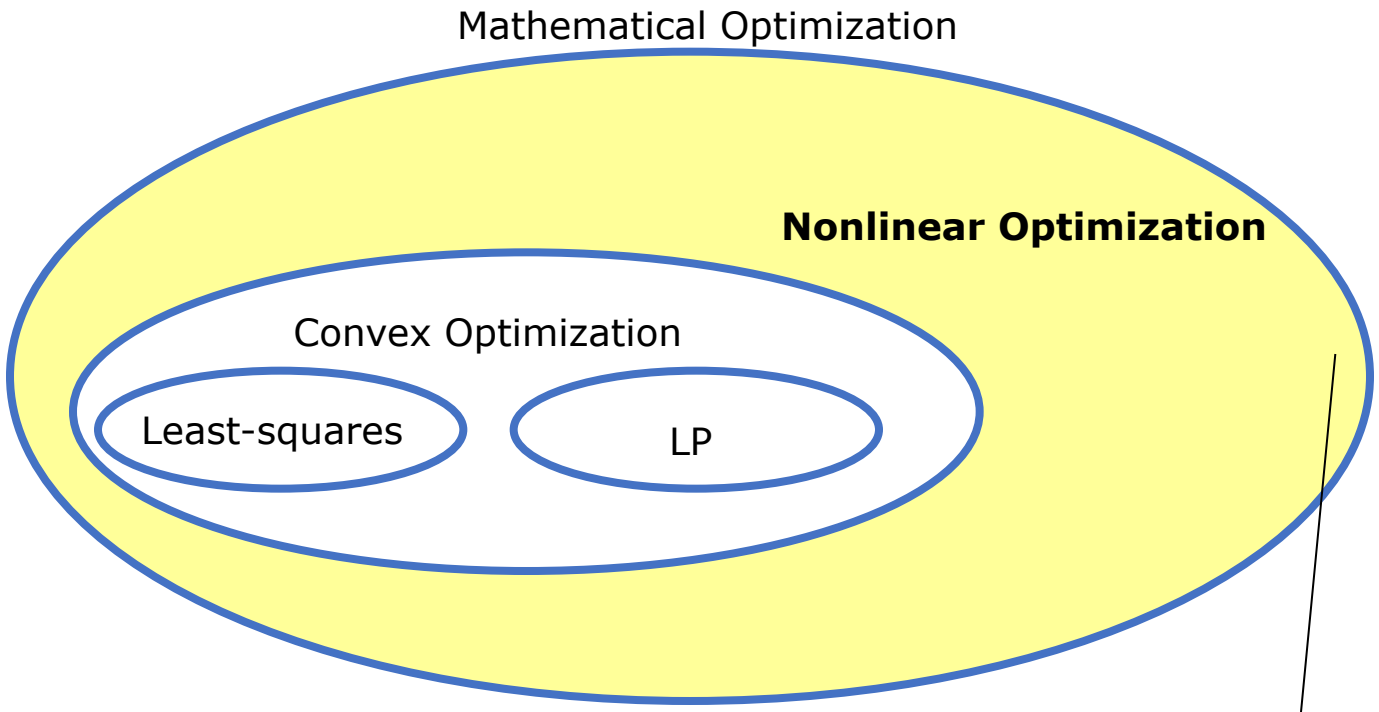
Also a mature technology!



minimize $f_0(x)$
subject to $f_i(x) \leq b_i, \quad i = 1, \ldots, m$

- No analytical solution
- Algorithms and software
- Reliable and efficient

Almost a ~~mature~~ technology!



- Sadly, no effective methods to solve
- Only approaches with some compromise
- Local optimization: *"more art than technology"*
- Global optimization: greatly compromised efficiency
- Help from convex optimization
  1) Initialization 2) Heuristics 3) Bounds

Far from a technology! (something to avoid)

Why Study Convex Optimization

With only a bit of exaggeration, we can say that, if you formulate a practical problem as a convex optimization problem, then you have solved the original problem. If not, there is little chance you can solve it.

-- Section 1.3.2, p8, *Convex Optimization*

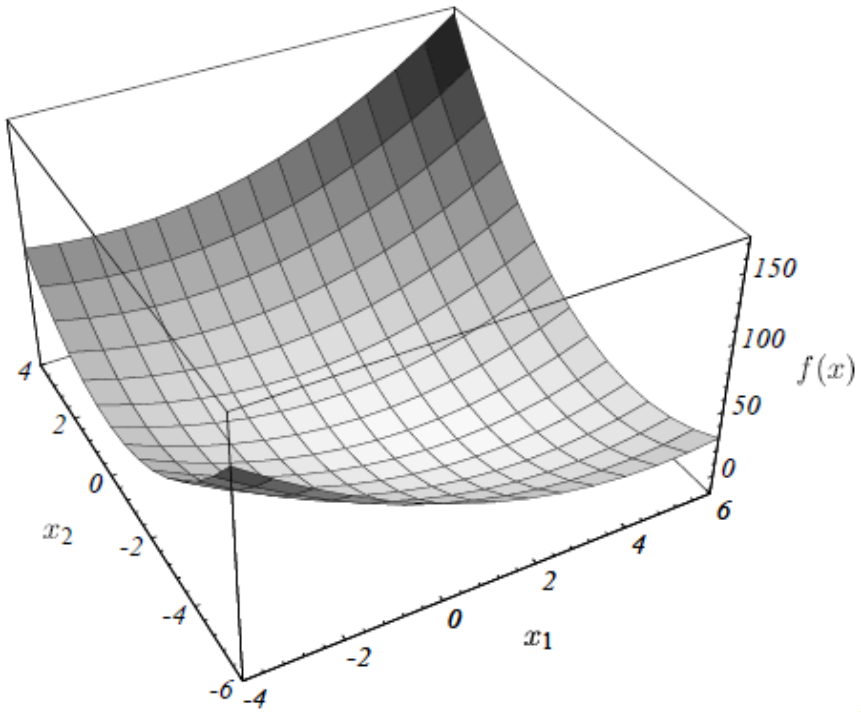
Recommended:

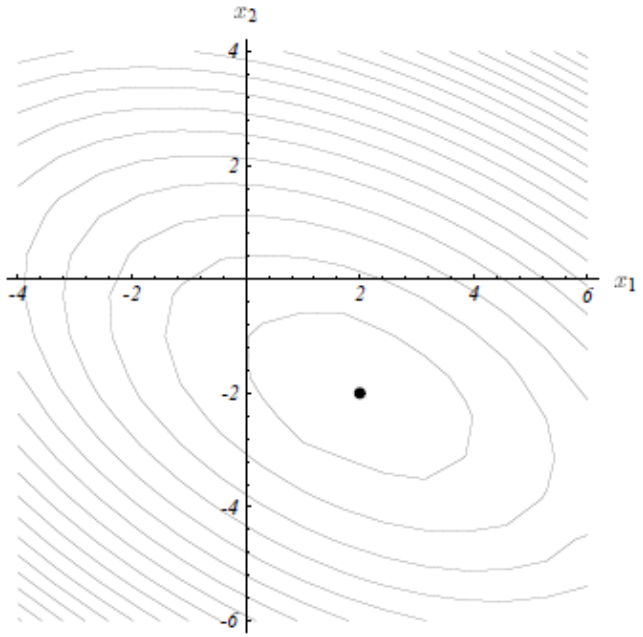
Course on Neural Net Training Dynamics
https://www.cs.toronto.edu/~rgrosse/courses/csc2541_2021/

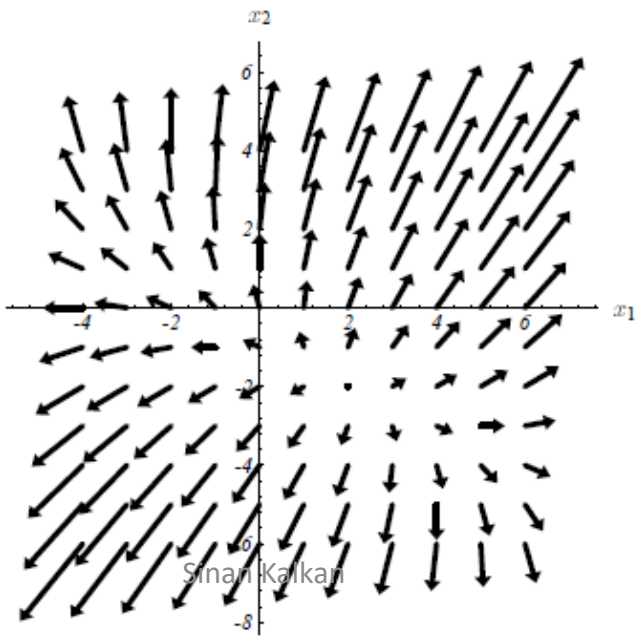
Gradient DESCENT strategies

Schemes of training

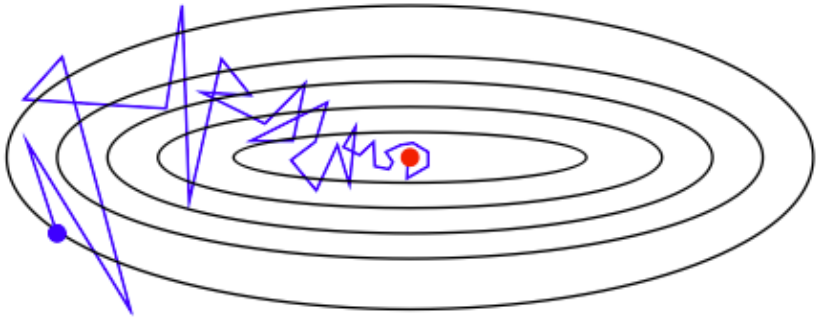
- True/Stochastic/Batch Gradient Descent
- Effect of batch size
- Second-order Methods
- Steepest/Conjugate Gradient Descent
- Momentum Gradient Descent


150
100
$f(x)$
50
0
4
2
0
x_2
-2
-4
-6
-4
-2
0
2
4
6
x_1

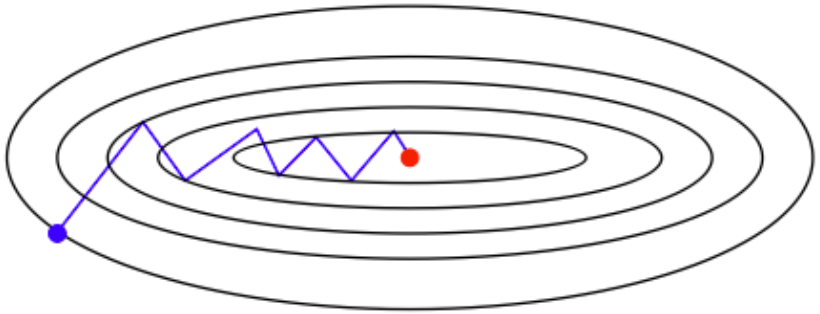

x_2
4
2
-2
-4
-6
-4
-2
2
4
6
x_1


x_2
6
4
2
-2
-4
-6
-8
-4
-2
2
4
6
x_1

Stochastic Gradient Descent



Batch Gradient Descent



F. Bach

Large vs. small batch sizes

forth the following as possible causes for this phenomenon: (i) LB methods over-fit the model; (ii) LB methods are attracted to saddle points; (iii) LB methods lack the *explorative* properties of SB methods and tend to zoom-in on the minimizer closest to the initial point; (iv) SB and LB methods converge to qualitatively different minimizers with differing generalization properties. The data presented in this paper supports the last two conjectures.

The main observation of this paper is as follows:

> The lack of generalization ability is due to the fact that large-batch methods tend to converge to *sharp minimizers* of the training function. These minimizers are characterized by a significant number of large positive eigenvalues in $\nabla^2 f(x)$, and tend to generalize less well. In contrast, small-batch methods converge to *flat minimizers* characterized by having numerous small eigenvalues of $\nabla^2 f(x)$. We have observed that the loss function landscape of deep neural networks is such that large-batch methods are attracted to regions with sharp minimizers and that, unlike small-batch methods, are unable to escape basins of attraction of these minimizers.

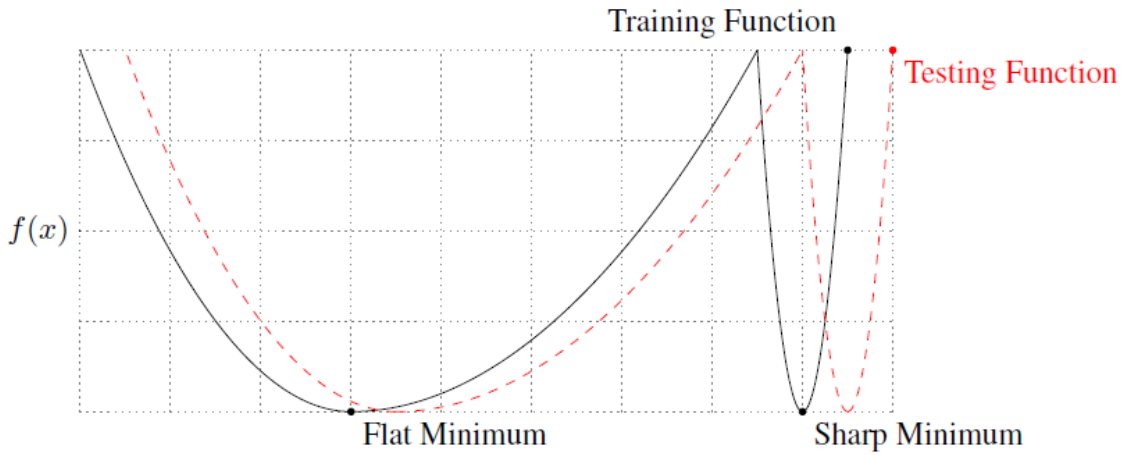


Figure 1: A Conceptual Sketch of Flat and Sharp Minima. The Y-axis indicates value of the loss function and the X-axis the variables (parameters)

On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima

Nitish Shirish Keskar*
Northwestern University
Evanston, IL 60208
keskar.nitish@u.northwestern.edu

Dheevatsa Mudigere
Intel Corporation
Bangalore, India
dheevatsa.mudigere@intel.com

Jorge Nocedal
Northwestern University
Evanston, IL 60208
j-nocedal@northwestern.edu

Mikhail Smelyanskiy
Intel Corporation
Santa Clara, CA 95054
mikhail.smelyanskiy@intel.com

Ping Tak Peter Tang
Intel Corporation
Santa Clara, CA 95054
peter.tang@intel.com

ICLR, 2017

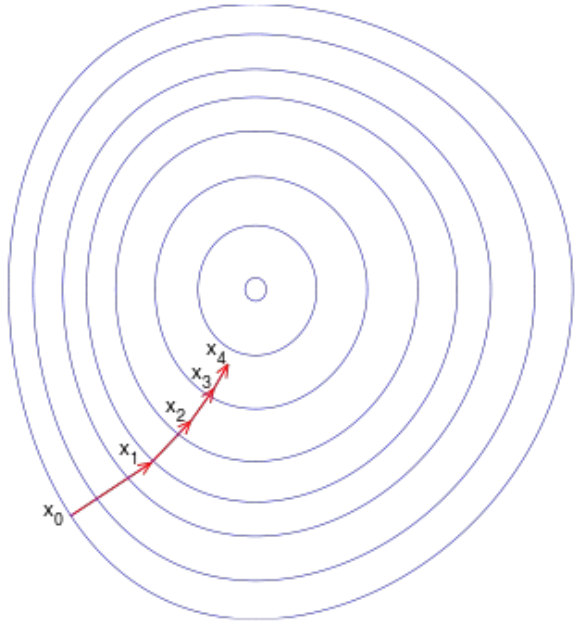
Large vs. small batch sizes

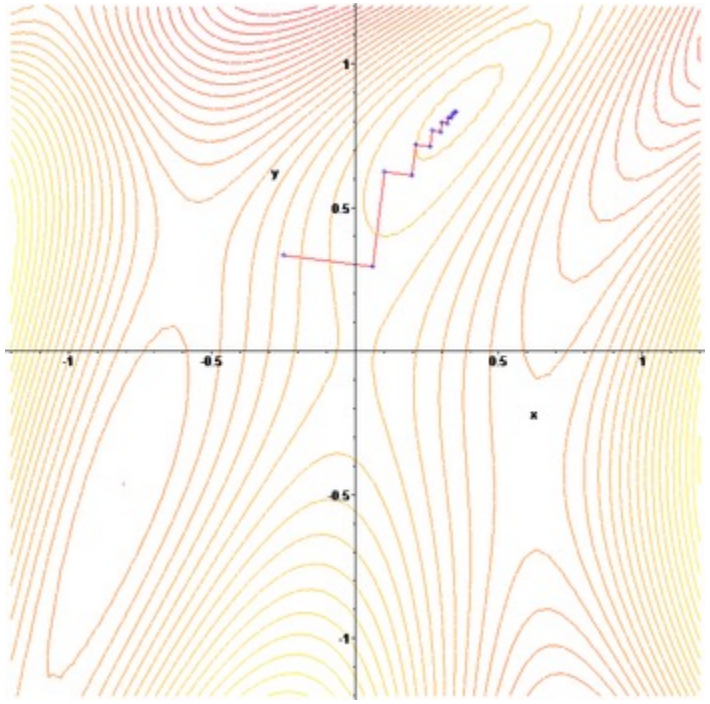
- Stability [1]:
  - Large batch sizes introduce stability in terms of gradient directions. But this increases the changes of getting stuck in local minima.
  - Small batch sizes introduce noisy gradients which make it difficult to get stuck in local minima.
- Local convergence & width of the minima [2]:
  - Small batch sizes tend to converge to solutions that are farther away from the initial position whereas large batch sizes lead to solutions close to the initial position.

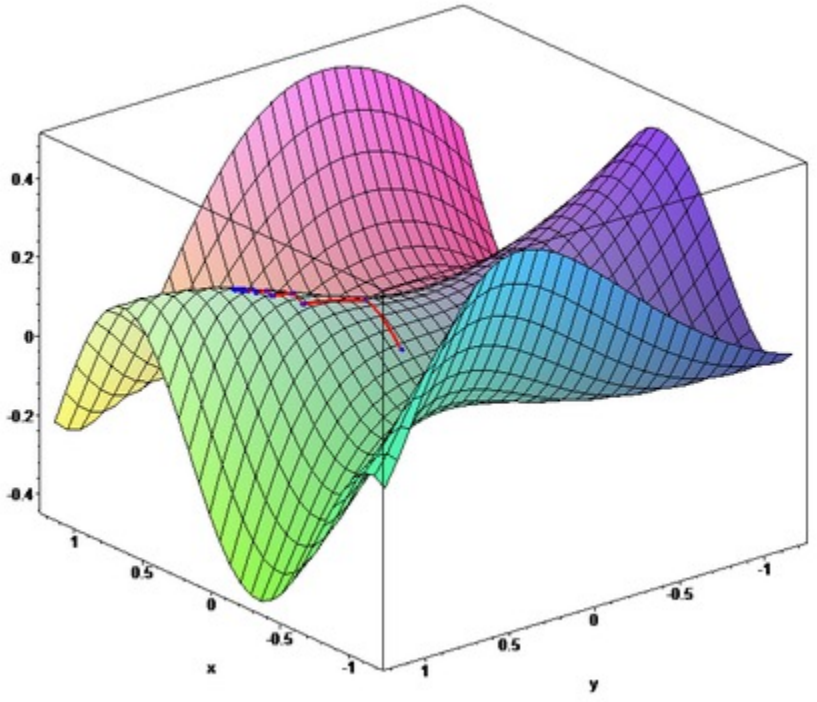
[1] Takase et al., “Why Does Large Batch Training Result in Poor Generalization? A Comprehensive Explanation and a Better Strategy from the Viewpoint of Stochastic Optimization”, Neural Computation, 2018.

[2] Keskar et al., “On large batch training for deep learning: Generalization gap and sharp minima”, ICLR 2017.

Gradient descent







https://en.wikipedia.org/wiki/Gradient_descent

Second order methods

- Newton's method for optimization:
  - $x \leftarrow x - [Hf(x)]^{-1}\nabla f(x)$
  - where $Hf(x)$ is the Hessian

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

Eq: https://en.wikipedia.org/wiki/Hessian_matrix

- Hessian gives a better feeling about the surface
  - It gives information about the curvature of surface

Intuition behind Newton's method

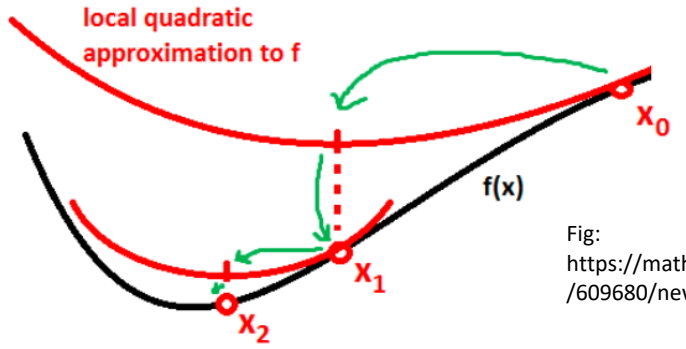


Fig:
https://math.stackexchange.com/questions/609680/newtons-method-intuition

- Newton's method assumes that the function ($f(x)$) we are trying to minimize is quadratic, and aims to find the minimum ($x+\delta$), where $f'(x+\delta)=0$.
- From Taylor expansion:

$$f(x+\delta) \cong f(x) + f'(x)\delta + \frac{1}{2}f''(x)\delta^2$$

- Solving for δ using $f'(x+\delta)=0$:

$$\frac{d}{d\delta}\left[f(x) + f'(x)\delta + \frac{1}{2}f''(x)\delta^2\right] = 0$$

which yields:

$$\delta \cong -f'(x)/f''(x)$$

- In high-dimensional cases, $f'(x)$ is replaced by $\nabla f(x)$ and $f''(x)$ by $Hf(x)$.

Compare this to Newton's method for finding the roots

- To find a root r of a function ($f(x)$), i.e., $f(r) = 0$:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

- In optimization, we wish to end up with $f'(x) = 0$ with:

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

Newton's method for optimization

- $x \leftarrow x - [Hf(x)]^{-1}\nabla f(x)$
  - Makes bigger steps in shallow curvature
  - Smaller steps in steep curvature

- Note that there is no (learning rate) hyper-parameter! (if you wish you can add a step size, but this is not necessary)
- Disadvantage:
  - Too much memory requirement
  - For 1 million parameters, this means a matrix of 1 million x 1 million ➔ ~ 3725 GB RAM
  - Alternatives exist to get around the memory problem (quasi-Newton methods, Limited-memory BFGS -- short for Broyden–Fletcher–Goldfarb–Shanno)

RPROP (Resilience Propagation)

- Instead of the magnitude, use the sign of the gradients

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^+ * \Delta_{ij}^{(t-1)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ \eta^- * \Delta_{ij}^{(t-1)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ \Delta_{ij}^{(t-1)} & , \text{ else} \end{cases} \quad (4)$$

$$\text{where } 0 < \eta^- < 1 < \eta^+$$

- Motivation: If the sign of a gradient has changed, that means we have "overshot" a minima
- Advantage: Faster to run/converge
- Disadvantage: More complex to implement

A Direct Adaptive Method for Faster Backpropagation Learning:
The RPROP Algorithm

Martin Riedmiller Heinrich Braun

1993

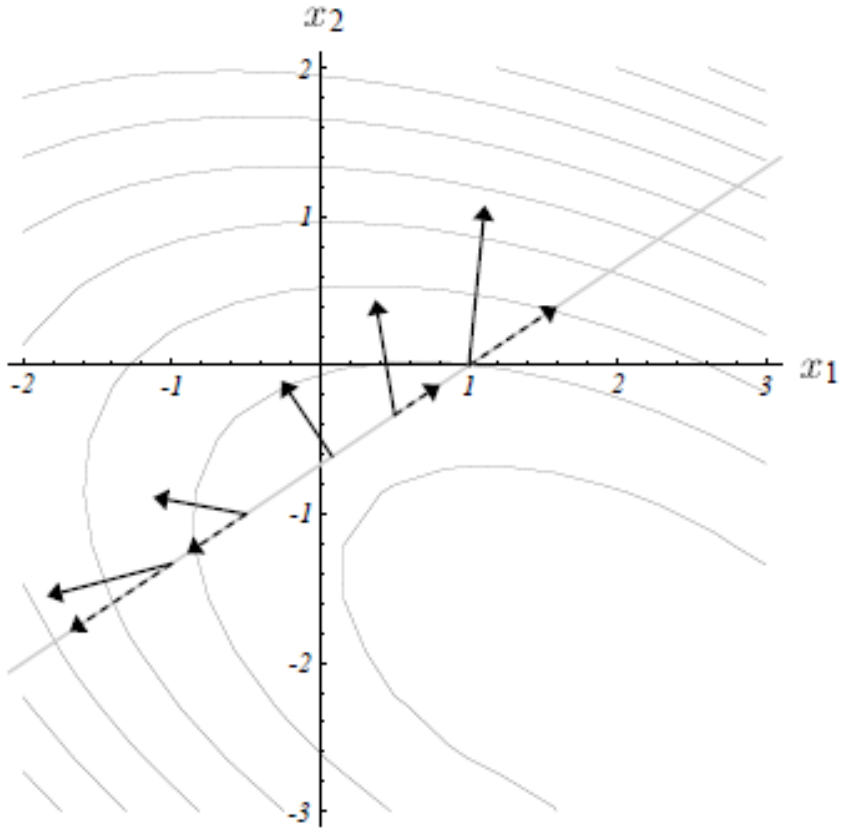
Gradient Descent with Line Search

- Gradient descent:

$$w_{ij}^{t} = w_{ij}^{t-1} + s\, dir_{ij}^{t-1}$$

where $dir_{ij}^{t-1} = -\partial L / \partial w_{ij}$

- Gradient descent with line search:
  - Choose s such that L is minimized along dir_{ij}^{t-1}.
  - Set $\frac{dL\left(w_{ij}^{t}\right)}{ds} = 0$ to find the optimal s.



Jonathan Richard Shewchuk

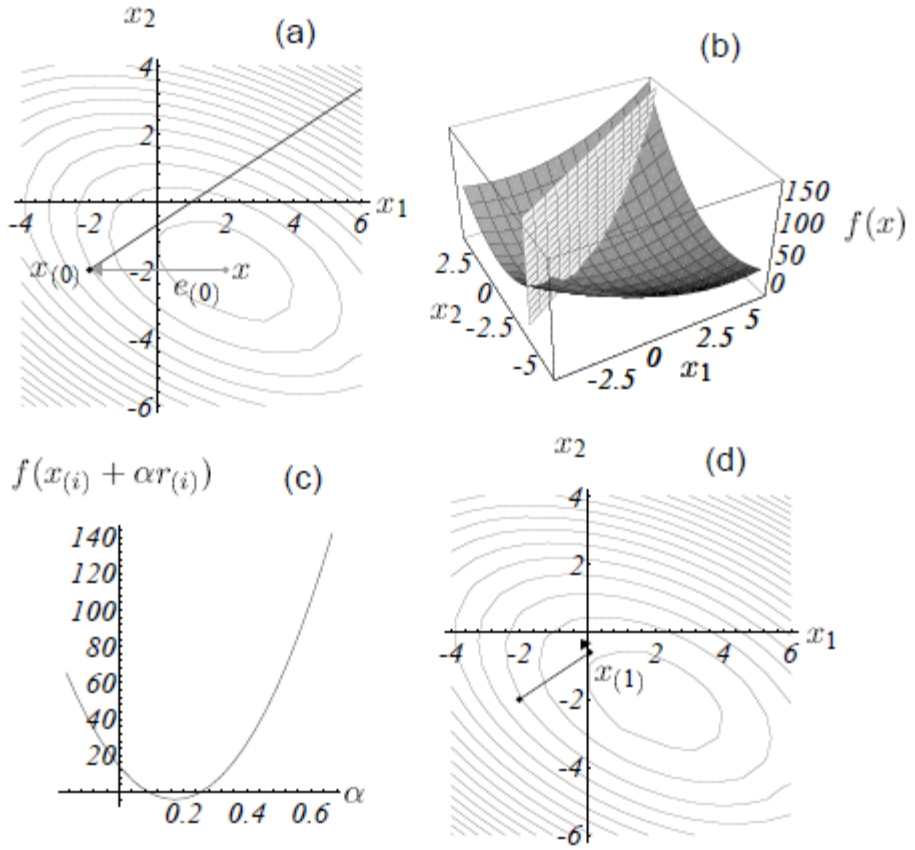


Figure 6: The method of Steepest Descent. (a) Starting at $[-2, -2]^T$, take a step in the direction of steepest descent of f. (b) Find the point on the intersection of these two surfaces that minimizes f. (c) This parabola is the intersection of surfaces. The bottommost point is our target. (d) The gradient at the bottommost point is orthogonal to the gradient of the previous step.

Gradient Descent with Line Search

$$w_{ij}^{t} = w_{ij}^{t-1} + s\, dir_{ij}^{t-1}$$

- Set $\frac{dL\left(w_{ij}^{t}\right)}{ds} = 0$ to find the optimal s.
- $\frac{dL\left(w_{ij}^{t}=w_{ij}^{t-1}+s\, dir_{ij}^{t-1}\right)}{ds} = \frac{dL}{dw_{ij}^{t}}\frac{dw_{ij}^{t}}{ds} = \frac{dL}{dw_{ij}^{t}} dir_{ij}^{t-1} = 0$

$$\frac{dL}{dw_{ij}^{t}}\frac{dw_{ij}^{t}}{ds} = \frac{dL}{dw_{ij}^{t}} dir_{ij}^{t-1} = 0$$

- Interpretation:
  - Choose s such that: the gradient direction at the new position is orthogonal to the current direction
- This is called steepest gradient descent
- Problem: makes zig-zag

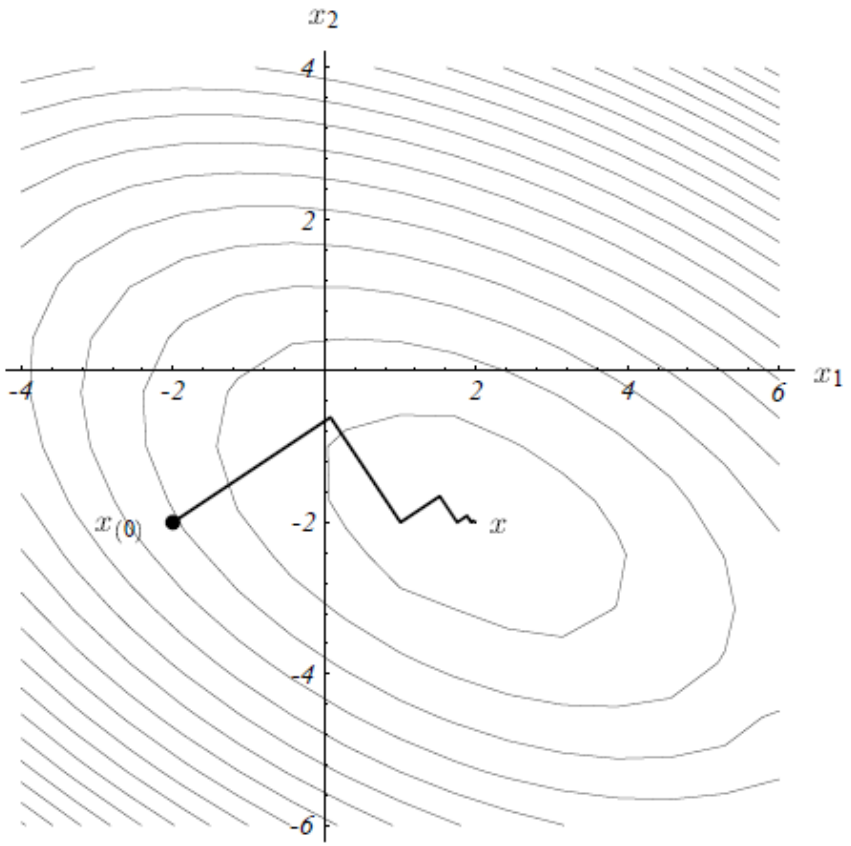
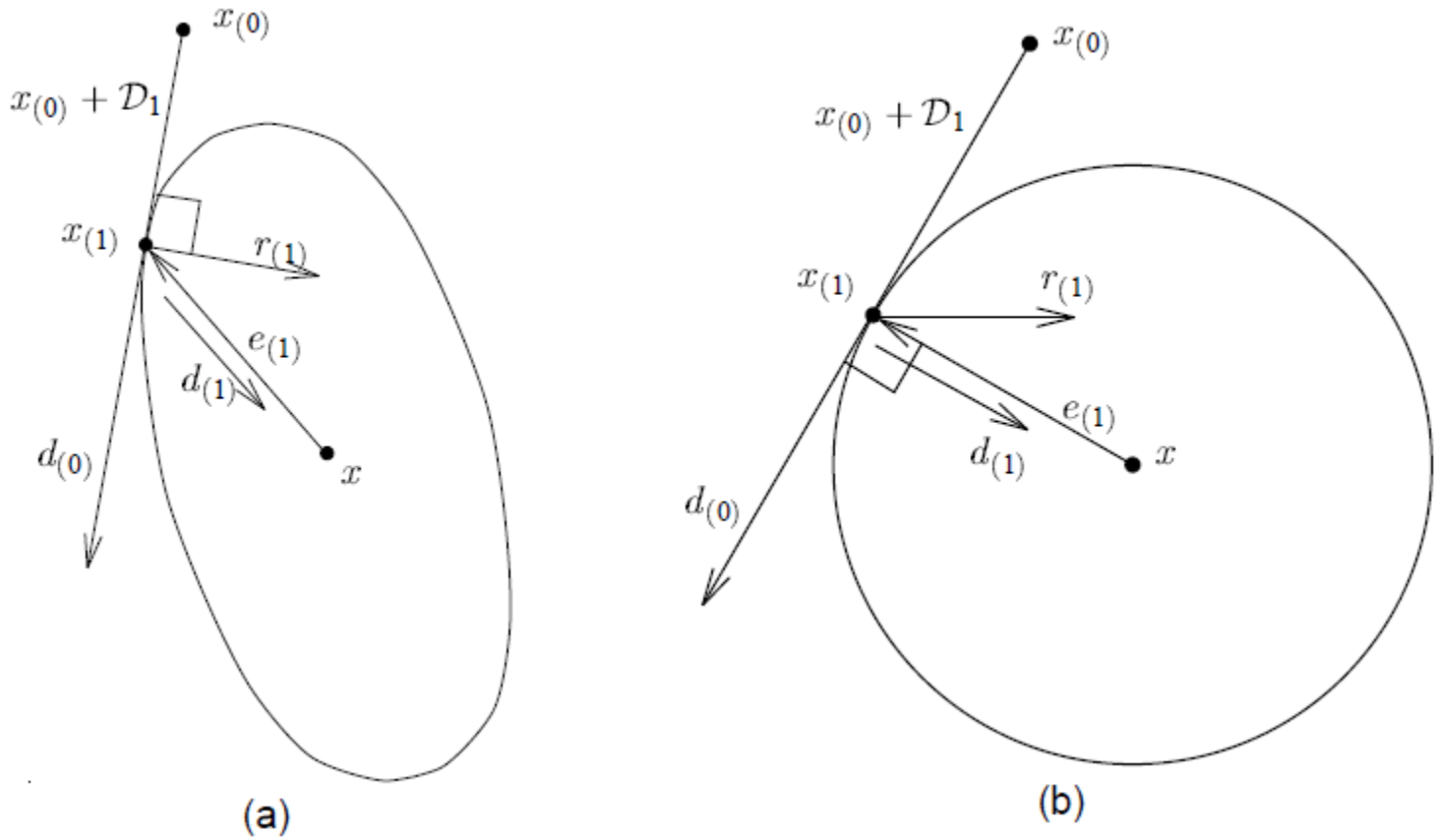


Figure 8: Here, the method of Steepest Descent starts at $[-2, -2]^T$ and converges at $[2, -2]^T$.

Conjugate Gradient Descent

- Motivation



Jonathan Richard Shewchuk

Conjugate Gradient Descent

- Two vectors are conjugate (A-orthogonal) if:

$$u^T A v = 0$$

- We assume that the error surface has the quadratic form:

$$f(x) = \frac{1}{2} x^T A x - b^T x + c$$

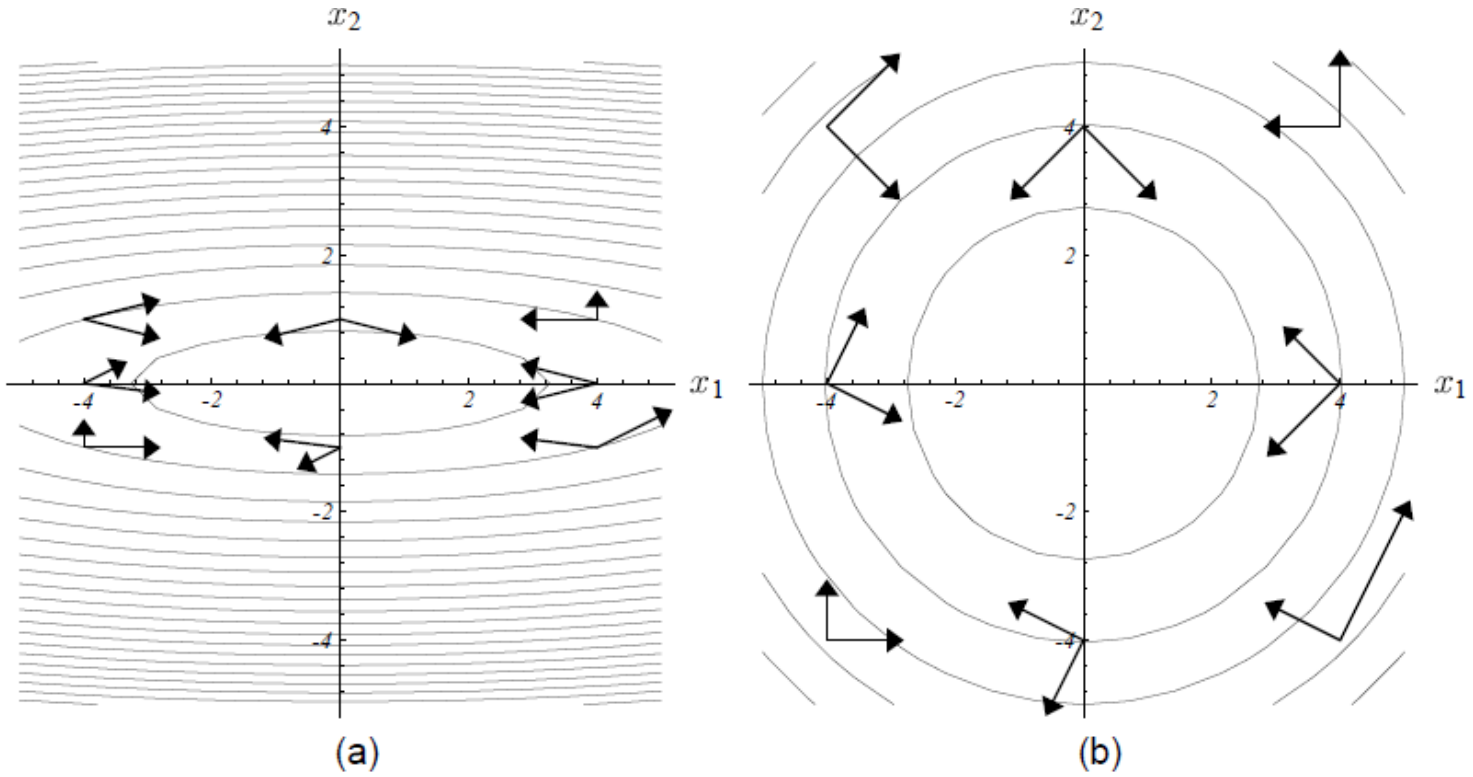


Figure 22: These pairs of vectors are A-orthogonal . . . because these pairs of vectors are orthogonal.

Jonathan Richard Shewchuk

Conjugate Gradient Descent

- $dir_{ij}^{t} = -\frac{\partial E\left(w_{ij}^{t}\right)}{\partial w_{ij}^{t}} + \beta\, dir_{ij}^{t-1}$
- By assuming quadratic form etc.:

$$\beta = \frac{\sum_{i,j}\left(\frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)} - \frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}\right)\cdot\frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)}}{\sum_{i,j}\frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}\cdot\frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}}$$

Jonathan Richard Shewchuk

Conjugate Gradient Descent

- Or simply as:

$$\beta = \frac{(\nabla E_{new} - \nabla E_{old}) \cdot \nabla E_{new}}{(\nabla E_{old})^2}$$

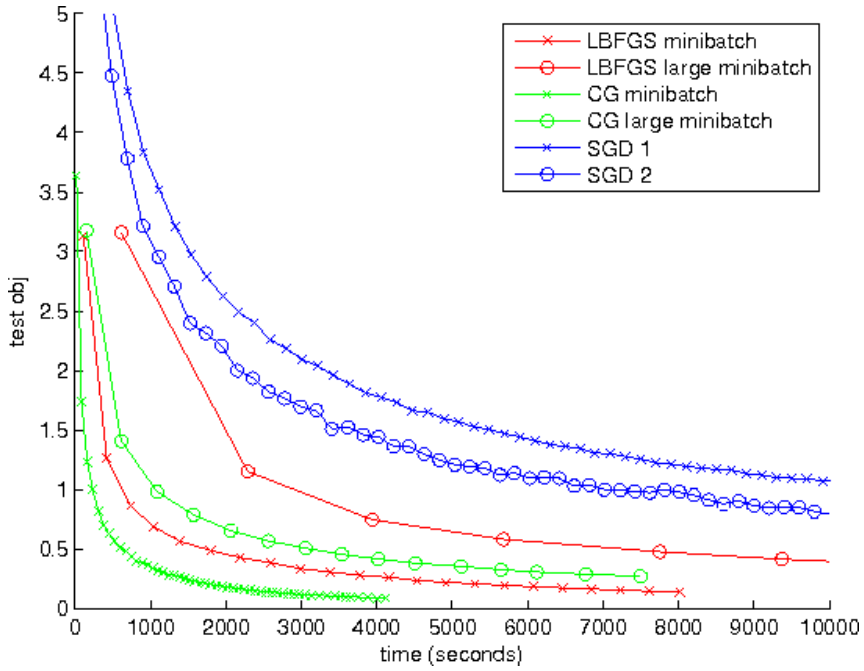
- Interpretation:
  - Rewrite this as:

$$\beta = \frac{\nabla E_{new}^2}{\nabla E_{old}^2} - \frac{\nabla E_{old} . \nabla E_{new}}{\nabla E_{old}^2}$$

- For more detailed motivation and derivations, see:
  - Jonathan Richard Shewchuk, “An Introduction to the Conjugate Gradient Method Without the Agonizing Pain”, 1994.
  - Jan A. Snyman, Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms, CH2, 2005.

Steepest and Conjugate Gradient Descent: Cons and Pros

- Pros:
  - Faster to converge than, e.g., stochastic gradient descent (even mini-batch)
- Cons:
  - They don't work well on saddle points
  - Computationally more expensive
  - In 2D:
    - Steepest descent is $O(n^2)$
    - Conjugate descent is $O(n^{3/2})$



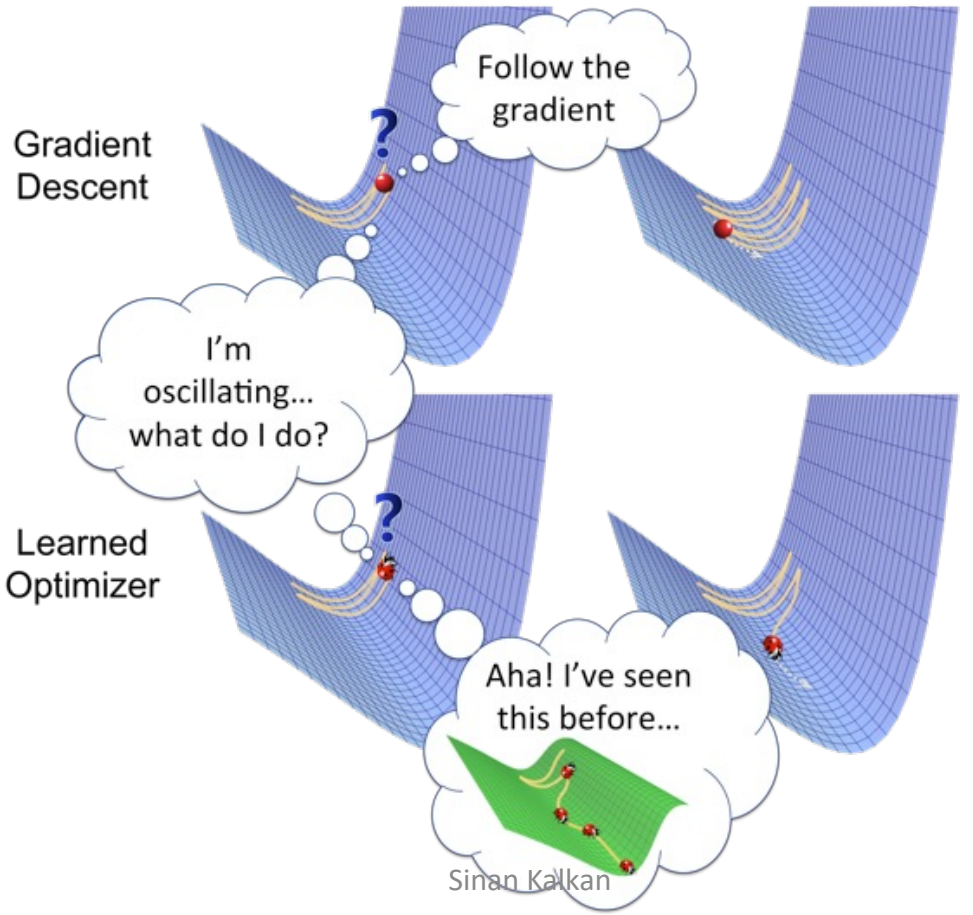
Le et al., "On optimization methods for deep learning", 2011.

Jonathan Richard Shewchuk, "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain", 1994.

Online Interactive Tutorial

http://www.benfrederickson.com/numerical-optimization/

- http://bair.berkeley.edu/blog/2017/09/12/learning-to-optimize-with-rl



Challenges of the Loss surface

and How to Avoid Them

Challenges

- Local minima
- Saddle points
- Cliffs
- Valleys

Local minima

- Solutions
  - Large training data
  - Stochastic gradient descent
  - Momentum
  - Adaptive learning rate
  - Good initialization
  - Different minimization strategies

- For smaller networks, local minima are more problematic

- For large-size networks, most local minima are equivalent and yield similar performance on a test set.
- The probability of finding a "bad" (high value) local minimum is non-zero for small-size networks and decreases quickly with network size.
- Struggling to find the global minimum on the training set (as opposed to one of the many good local ones) is not useful in practice and may lead to overfitting.

2014

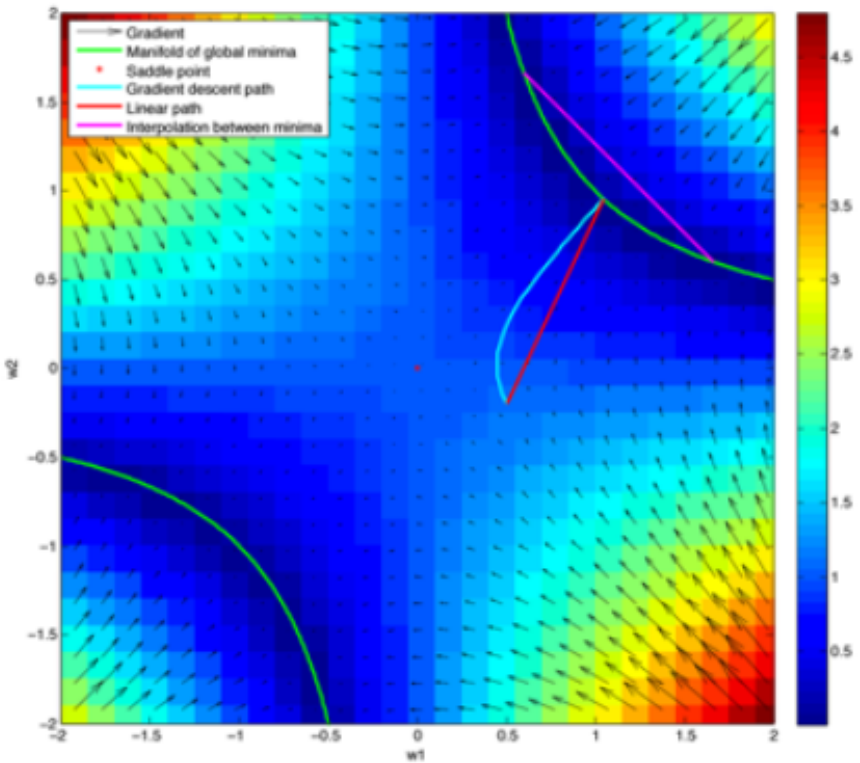
The Loss Surfaces of Multilayer Networks

Anna Choromanska achoroma@cims.nyu.edu
Mikael Henaff mbh305@nyu.edu
Michael Mathieu mathieu@cs.nyu.edu
Gérard Ben Arous benarous@cims.nyu.edu
Yann LeCun yann@cs.nyu.edu

Google

Do neural nets have saddle points?

- Saxe et al, 2013:
- neural nets without non-linearities have many saddle points
- all the minima are global
- all the minima form a connected manifold



Google

Do neural nets have saddle points?

- Dauphin et al 2014: Experiments show neural nets do have as many saddle points as random matrix theory predicts
- Choromanska et al 2015: Theoretical argument for why this should happen
- Major implication: **most minima are good, and this is more true for big models.**
- Minor implication: the reason that *Newton's method* works poorly for neural nets is its attraction to the ubiquitous saddle points.

I. Goodfellow

Valleys, Cliffs and Exploding Gradients

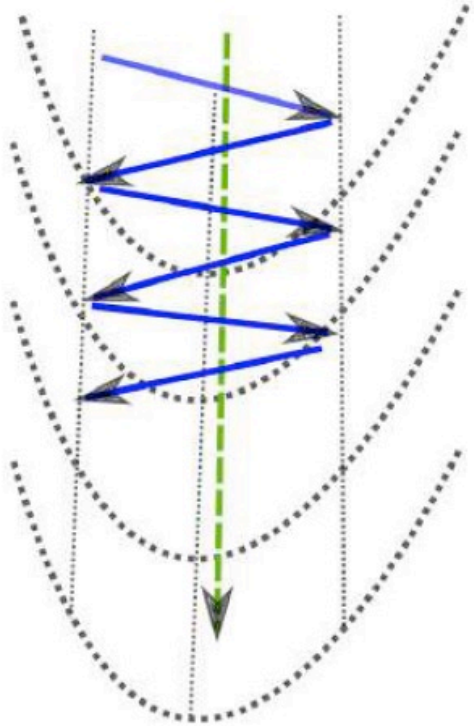


Figure 8.1: One theory about the neural network optimization is that poorly conditioned Hessian matrices cause much of the difficulty in training. In this view, some directions have a high curvature (second derivative), corresponding to the quickly rising sides of the valley (going left or right), and other directions have a low curvature, corresponding to the smooth slope of the valley (going down, dashed arrow). Most second-order methods, as well as momentum or gradient averaging methods are meant to address that problem, by increasing the step size in the direction of the valley (where it pays off the most in the long run to go) and decreasing it in the directions of steep rise, which would otherwise lead to oscillations (blue full arrows). The objective is to smoothly go down, staying at the bottom of the valley (green dashed arrow).

Valleys, Cliffs and Exploding Gradients

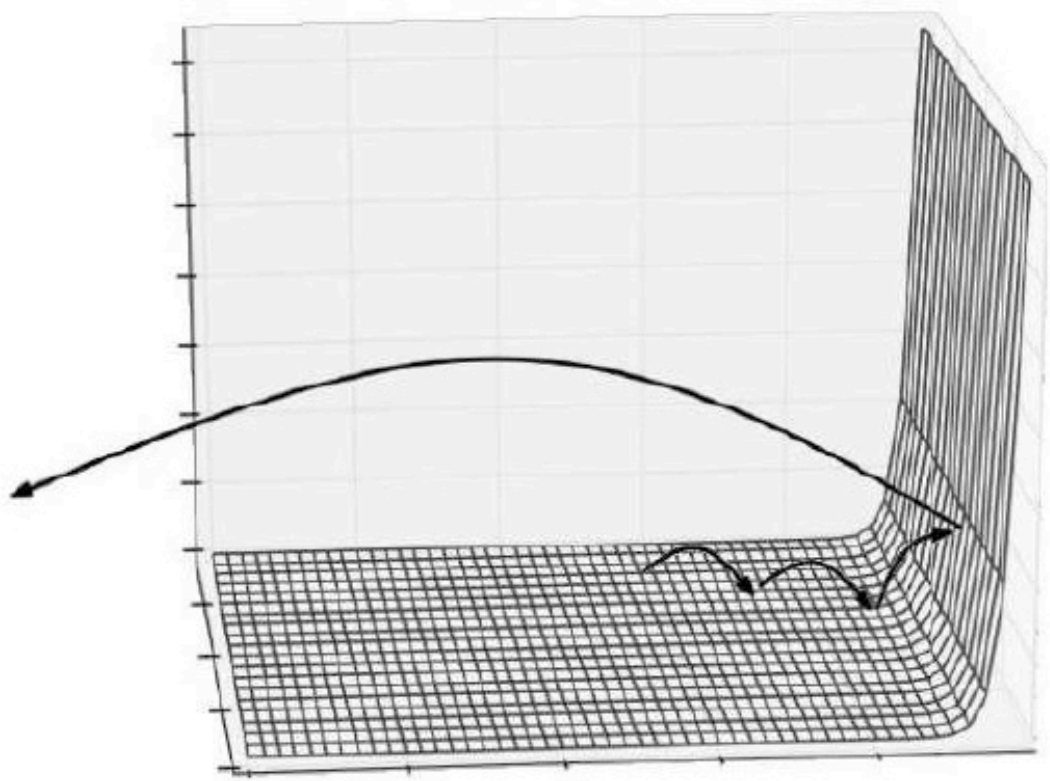


Figure 8.2: Contrary to what is shown in Figure 8.1, the objective function for highly non-linear deep neural networks or for recurrent neural networks is typically not made of symmetrical sides. As shown in the figure, there are sharp non-linearities that give rise to very high derivatives in some places. When the parameters get close to such a cliff region, a gradient descent update can catapult the parameters very far, possibly ruining a lot of the optimization work that had been done. Figure graciously provided by Razvan Pascanu (Pascanu, 2014).

Valleys, Cliffs and Exploding Gradients

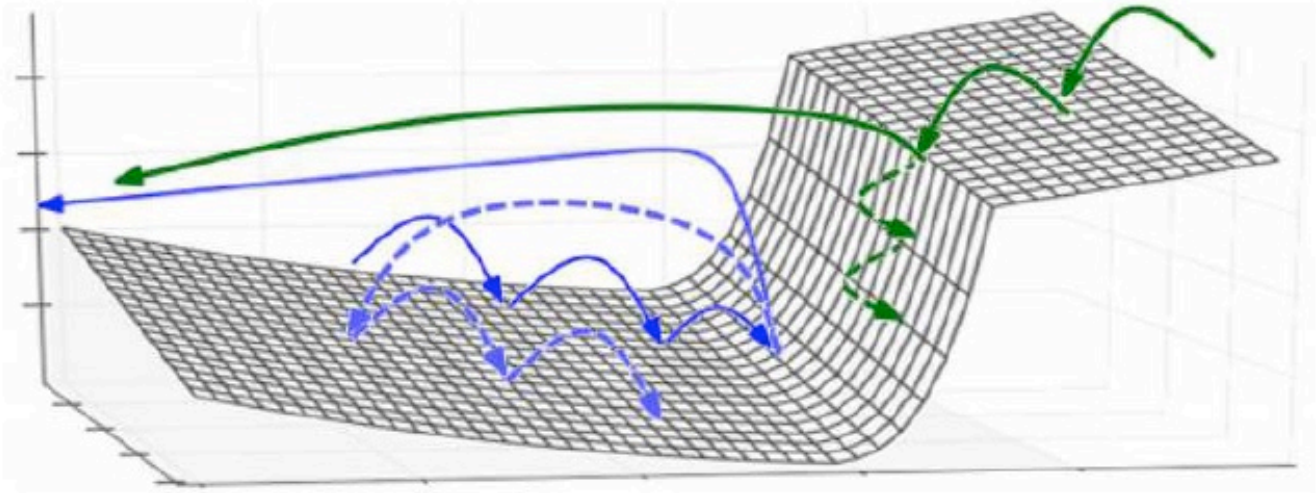


Figure 8.3: To address the presence of cliffs such as shown in Figure 8.2, a useful heuristic is to clip the magnitude of the gradient, only keeping its direction if its magnitude is above a threshold (which is a hyperparameter, although not a very critical one). Using such a gradient clipping heuristic (dotted arrows trajectories) helps to avoid the destructive big moves which would happen when approaching the cliff, either from above or from below (bold arrows trajectories). Figure graciously provided by Razvan Pascanu (Pascanu, 2014).

Using momentum to improve steps

Momentum

Update at $t-1$

Update recommended by the gradient at t

- Maintain a "memory"

$$\Delta\theta_t \leftarrow \mu\,\Delta\theta_{t-1} + (-\eta\,\nabla_{\theta_{t-1}}\mathcal{L})$$

where μ is called the momentum weight/coefficient

- Momentum filters oscillations on gradients (i.e., oscillatory movements on the error surface)

- μ is typically initialized to 0.9.
  - It is better if it anneals from 0.5 to 0.99 over multiple epochs

Momentum

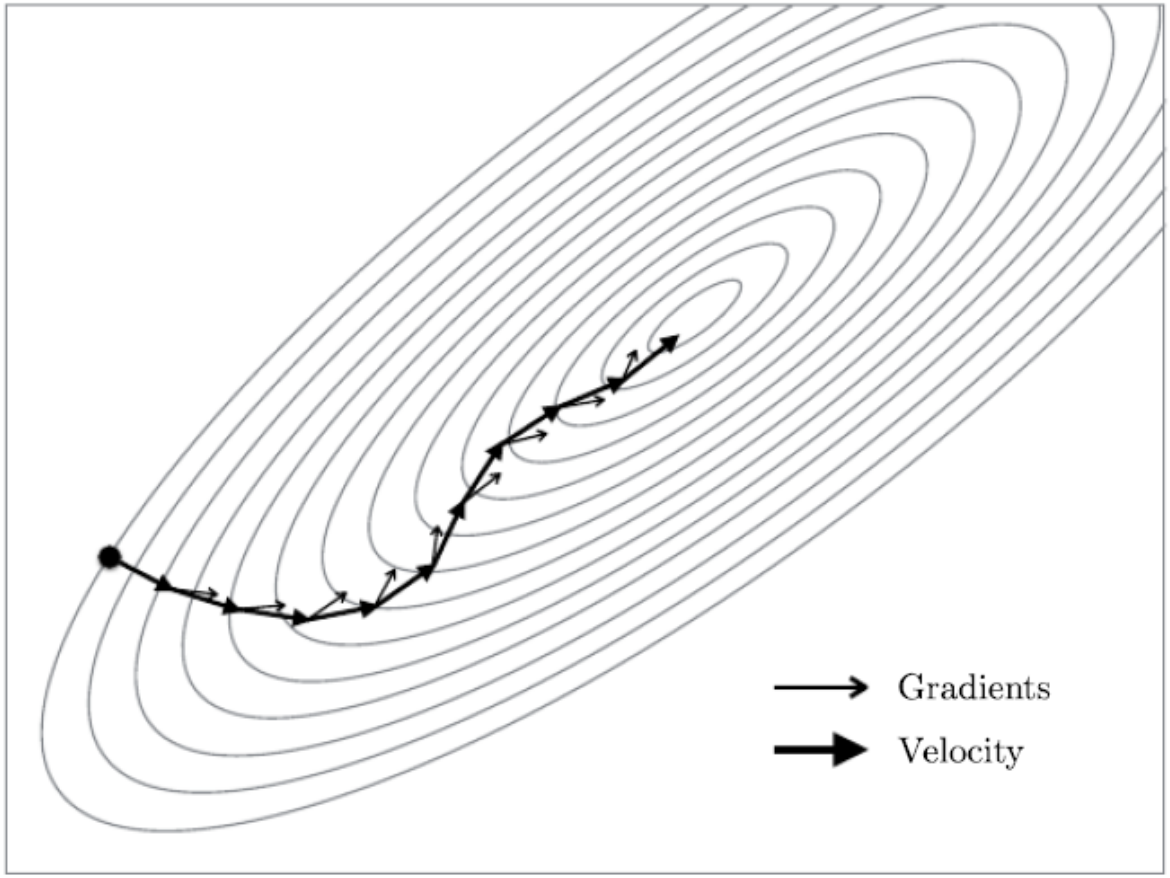
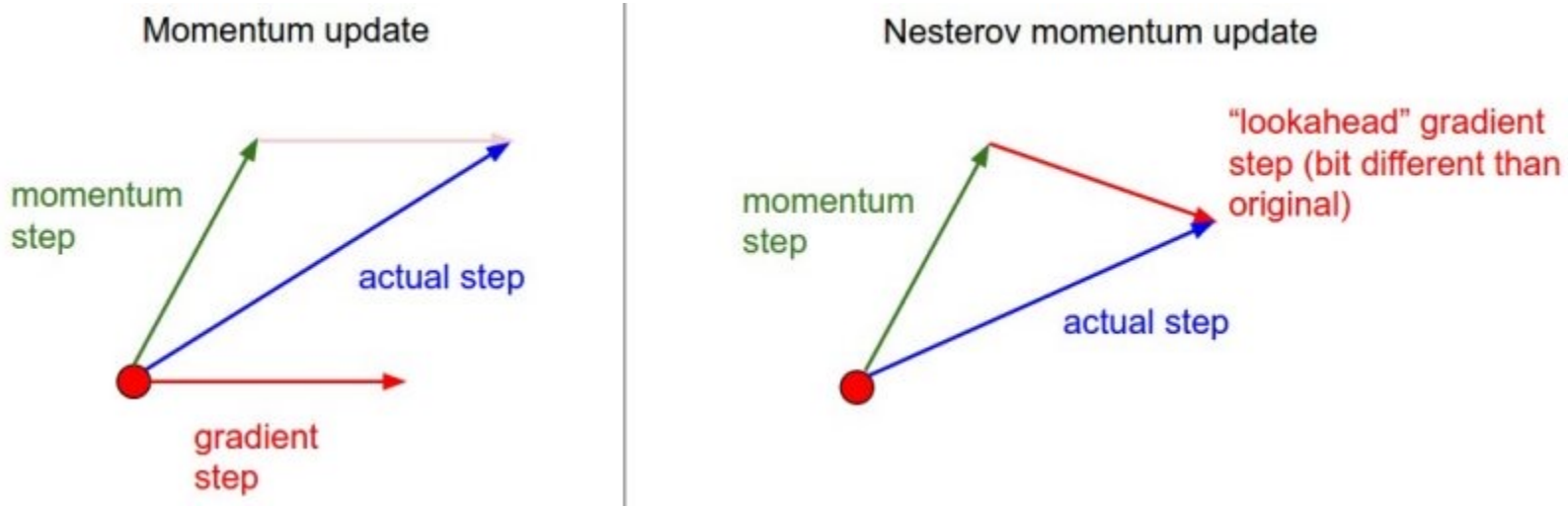


Figure 8.5: The effect of momentum on the progress of learning. Momentum acts to accumulate gradient contributions over training iterations. Directions that consistently have positive contributions to the gradient will be augmented.

Nesterov Momentum

- Use a "lookahead" step to update:

$$\theta_{\text{ahead}} \leftarrow \theta_t + \mu\, \Delta\theta_{t-1}$$
$$\Delta\theta_t \leftarrow \mu\, \Delta\theta_{t-1} - \eta\, \nabla_{\theta_{\text{ahead}}} \mathcal{L}_{\text{ahead}}$$
$$\theta_{t+1} \leftarrow \theta_t + \Delta\theta_t$$



http://cs231n.github.io/neural-networks-3/

Nesterov Momentum (alternative formulation)

Classical Momentum

$$v_{t+1} = \mu_t v_t - \alpha_t J'(\theta_t)$$
$$\theta_{t+1} = \theta_t + v_{t+1}$$

Nesterov's Momentum

$$y_{t+1} = (1 + \mu_t)\theta_t - \mu_t \theta_{t-1}$$
$$\theta_{t+1} = y_{t+1} - \alpha_t J'(y_{t+1})$$

Equations from: Botev, A., Lever, G., & Barber, D. (2017). Nesterov's accelerated gradient and momentum as approximations to regularised update descent. IJCNN.

- Uses smoothed weights
- Uses future gradient to update
- Guaranteed optimal convergence rate for convex functions (if first-order gradient based methods are used)

Y. Nesterov. A method of solving a convex programming problem with convergence rate O(1/k2). In Soviet Mathematics Doklady, volume 27, pages 372–376, 1983.

Momentum vs. Nesterov Momentum

- When the learning rate is very small, they are equivalent.
- When the learning rate is sufficiently large, Nesterov Momentum performs better (it is more responsive).
- See for an in-depth comparison:

On the importance of initialization and momentum in deep learning

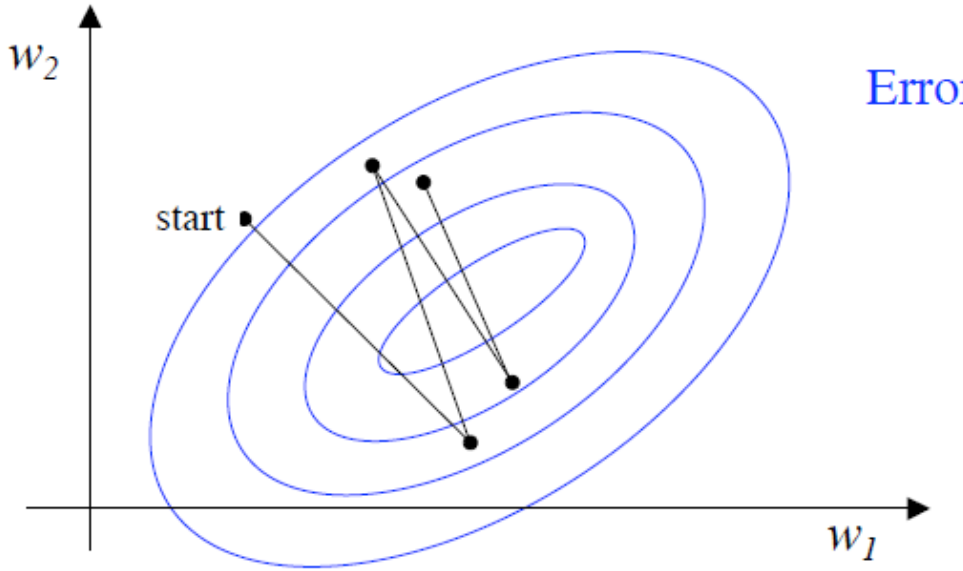
---

Ilya Sutskever[1] ILYASU@GOOGLE.COM
James Martens JMARTENS@CS.TORONTO.EDU
George Dahl GDAHL@CS.TORONTO.EDU
Geoffrey Hinton HINTON@CS.TORONTO.EDU

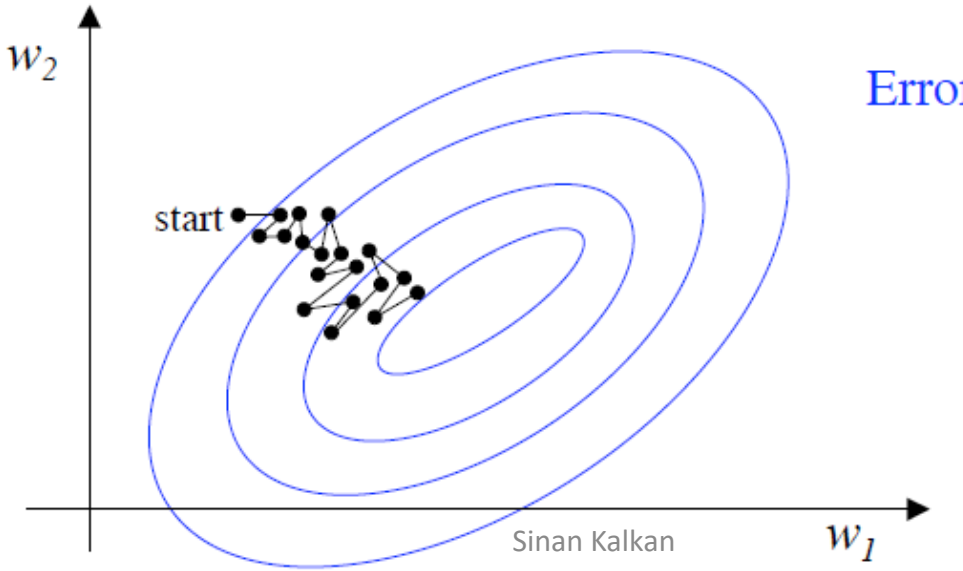
Demo (and further reading)

http://distill.pub/2017/momentum/

Setting the learning rate



Error contours



Error contours

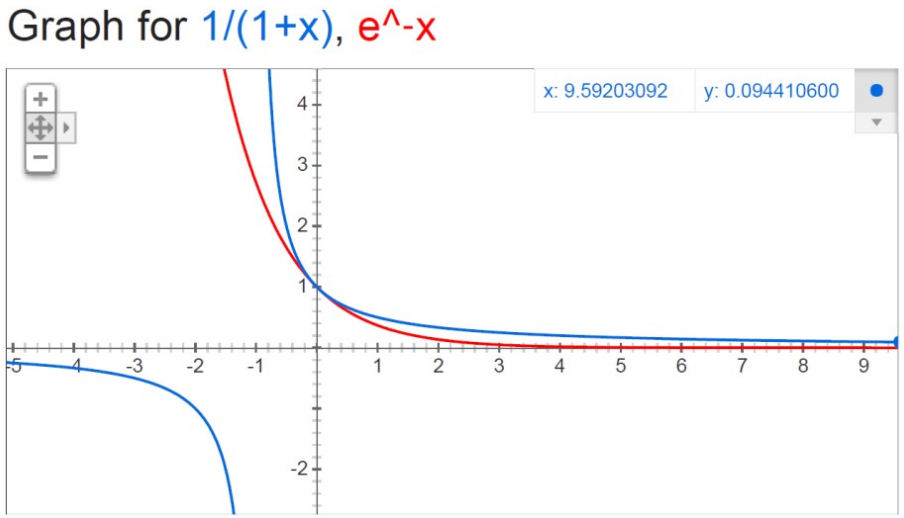
Alternatives

- Single global learning rate
  - Constant Learning Rate
  - Scheduling Learning Rate

- Per-parameter learning rate
  - AdaGrad
  - RMSprop
  - Adam
  - AdaDelta

Global Methods: Scheduling the learning rate

- Step decay
  - $\eta' \leftarrow \eta \times c$, where c could be 0.5, 0.4, 0.3, 0.2, 0.1 etc.
- Exponential decay:
  - $\eta = \eta_0 e^{-kt}$, where t is iteration number
  - η_0, k: hyperparameters
- 1/t decay:
  - $\eta = \eta_0/(1 + kt)$
- If you have time, keep decay small and train longer



Global Methods: warm-up

- Start with a small learning rate [1]
  - Constant learning rate
  - Gradually increasing
- Why? The first steps of learning appear to be very critical [2]

[1] Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., ... & He, K. (2017). Accurate, large minibatch sgd: Training imagenet in 1 hour. arXiv preprint arXiv:1706.02677. (this is not the first paper to do so)

[2] Achille, A., Rovere, M., & Soatto, S. (2018). Critical learning periods in deep networks. In International Conference on Learning Representations.

Global Methods: Cyclic Learning Rates

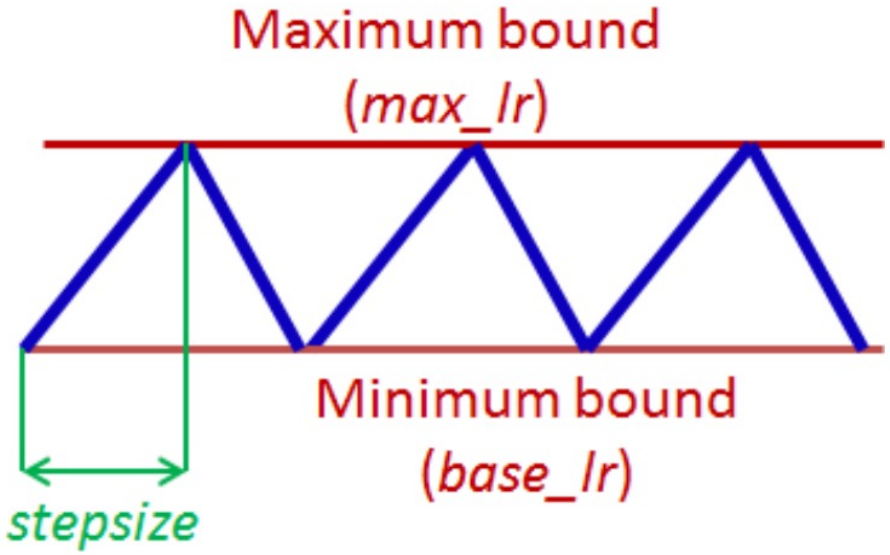


Figure 2. Triangular learning rate policy. The blue lines represent learning rate values changing between bounds. The input parameter *stepsize* is the number of iterations in half a cycle.

Smith, L. N. (2017). Cyclical learning rates for training neural networks. In 2017 IEEE winter conference on applications of computer vision (WACV) (pp. 464-472). IEEE.

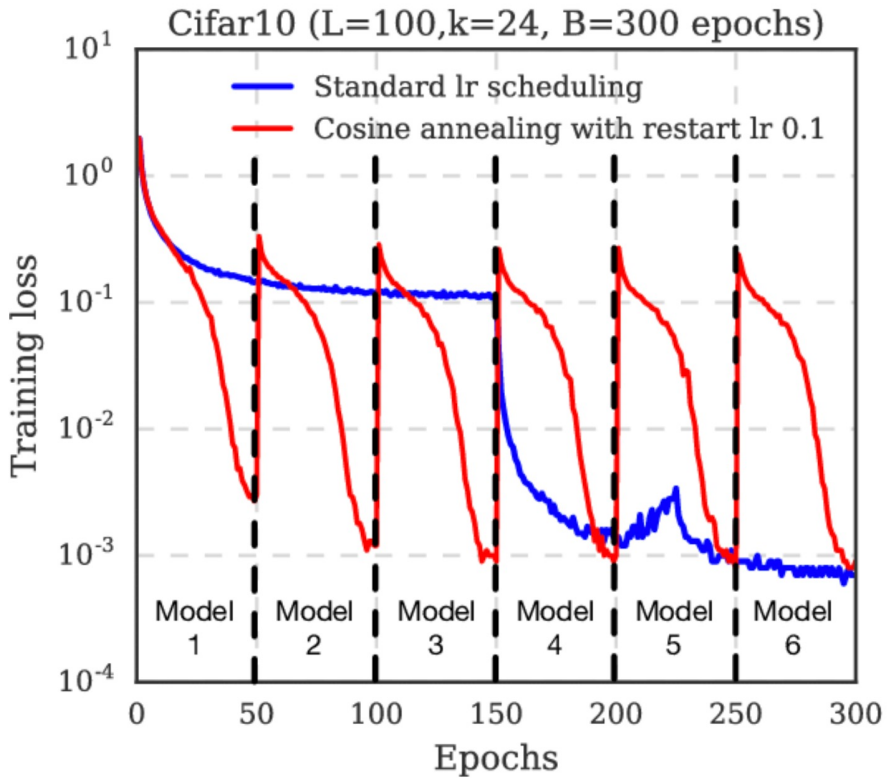
Global Methods: Cosine Scheduling

Cosine Annealing is a type of learning rate schedule that has the effect of starting with a large learning rate that is relatively rapidly decreased to a minimum value before being increased rapidly again. The resetting of the learning rate acts like a simulated restart of the learning process and the re-use of good weights as the starting point of the restart is referred to as a "warm restart" in contrast to a "cold restart" where a new set of small random numbers may be used as a starting point.

$$\eta_t = \eta^i_{min} + \frac{1}{2}(\eta^i_{max} - \eta^i_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_i}\pi\right)\right)$$

Where where η^i_{min} and η^i_{max} are ranges for the learning rate, and T_{cur} account for how many epochs have been performed since the last restart.

Text Source: Jason Brownlee



https://paperswithcode.com/method/cosine-annealing

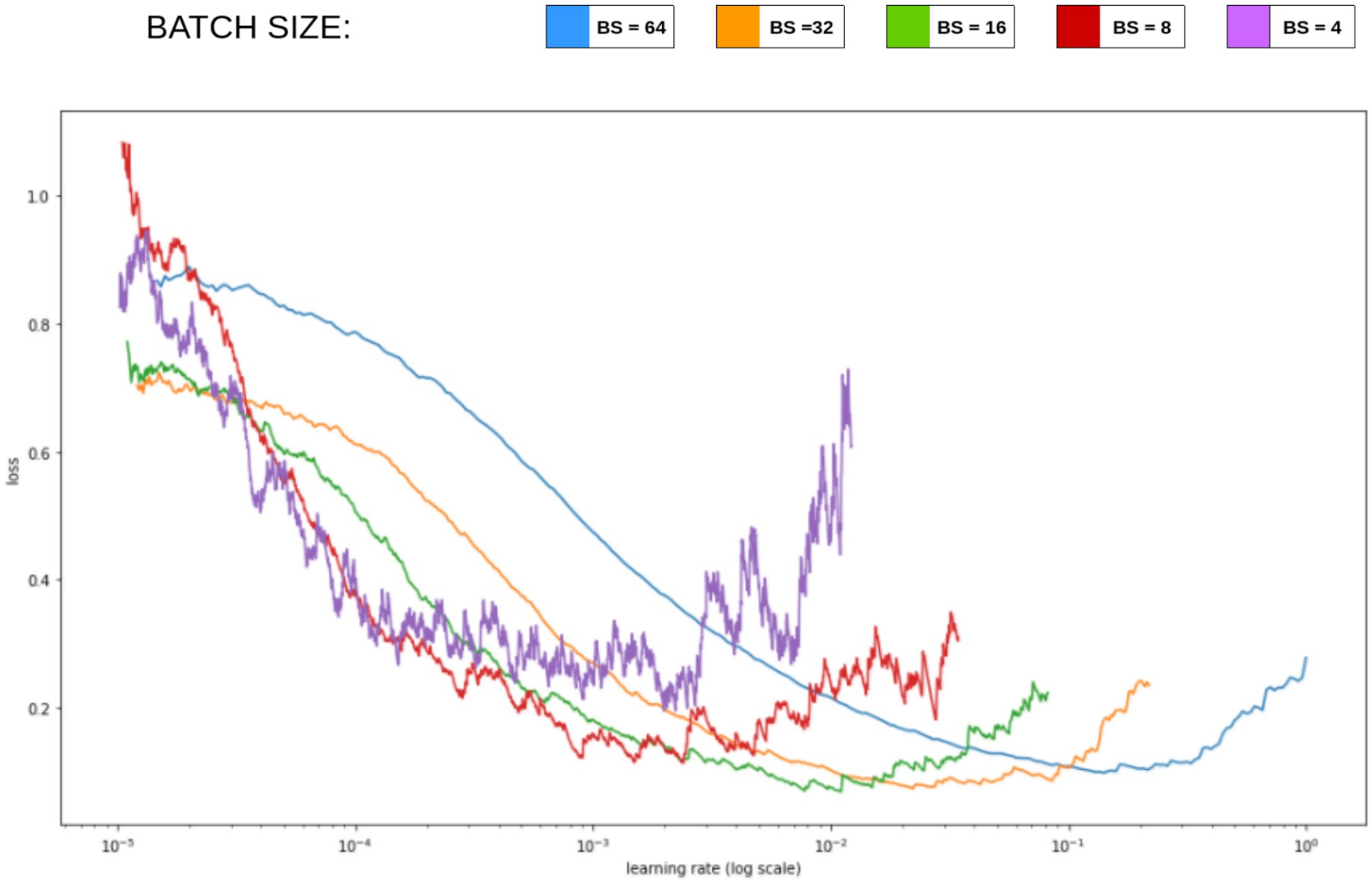
Global Methods: learning rate & the batch size

- Bigger batch size, bigger learning rate
- Increase batch size => increase learning rate
  - If you increase batch size from N to kN, learning rate should be scaled by:
    - sqrt(k) [1]
    - k [2]
- Two interpretations:
  - Bigger batch means more stable gradient => Safer to make large steps.
  - Bigger batch means less number of update steps => increase learning rate to compensate.

[1] Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. arXiv preprint arXiv:1404.5997.

[2] Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., ... & He, K. (2017). Accurate, large minibatch sgd: Training imagenet in 1 hour. arXiv preprint arXiv:1706.02677.

Global Methods: learning rate & the batch size



https://miguel-data-sc.github.io/2017-11-05-first/

Per-parameter Methods: Adagrad

- Higher the gradient, lower the learning rate
- Accumulate square of gradients **elementwise** (initially $r = 0$):

$$r_t \leftarrow r_{t-1} + \left(\sum_{i=1:M} \nabla_{\theta_t} \mathcal{L}(\mathbf{x}_i; \theta_t) \right)^2$$

- Update each parameter/weight based on the gradient on that:

$$\Delta\theta_{t+1} \leftarrow -\frac{\eta}{\sqrt{r_t}} \sum_{i=1:M} \nabla_{\theta_t} \mathcal{L}(\mathbf{x}_i; \theta_t)$$

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$.
Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.
end while

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. Journal of machine learning research, 12(7).

Algorithm taken from: Goodfellow et al., Deep Learning, 2016.

Per-parameter Methods: Root-Mean-Squared Propagation (RMSprop)

- Similar to Adagrad. Adagrad uses the whole history of gradients, which can be a limitation when training converges to a nice "basin".

- RMSprop handles this by weighted/moving averaging (again, **elementwise**):

$$r_t \leftarrow \rho r_{t-1} + (1-\rho)\left(\sum_{i=1:M} \nabla_{\theta_t} \mathcal{L}(\mathbf{x}_i; \theta_t)\right)^2$$

- ρ is typically one of: 0.9, 0.99, 0.999.

- Update each parameter/weight based on the gradient on that:

$$\Delta\theta_{t+1} \leftarrow -\frac{\eta}{\sqrt{r_t}} \sum_{i=1:M} \nabla_{\theta_t} \mathcal{L}(\mathbf{x}_i; \theta_t)$$

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
  Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
  Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$.
  Compute parameter update: $\Delta\boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta+\boldsymbol{r}}}$ applied element-wise)
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.
end while

Currently, unpublished. Proposed by Hinton in one of his lectures.

Algorithm taken from: Goodfellow et al., Deep Learning, 2016.

Per-parameter Methods: RMSprop with Nesterov Momentum

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$

Initialize accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$.

Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.

end while

Algorithm taken from: Goodfellow et al., Deep Learning, 2016.

Per-parameter Methods: Adaptive Moments (Adam)

- A variation of RMSprop + momentum
- Incorporates first & second order moments
- Bias correction needed to get rid of bias towards zero at initialization

Algorithm taken from:
Goodfellow et al., Deep Learning, 2016.

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
Initialize time step $t = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
$t \leftarrow t + 1$
Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

Add this from NeurIPS2024 (?)

Algorithm 1 ADOPT algorithm

Require: Learning rate $\{\alpha_t\}$, initial parameter $\boldsymbol{\theta}_0$

Require: Exponential decay rate $0 \leq \beta_1 < 1, 0 \leq \beta_2 \leq 1$, small constant $\epsilon > 0$

$\boldsymbol{v}_0 \leftarrow \boldsymbol{g}_0 \odot \boldsymbol{g}_0, \boldsymbol{m}_1 \leftarrow \boldsymbol{g}_1 / \max\left\{\sqrt{\boldsymbol{v}_0}, \epsilon\right\}$

for $t = 1$ to T **do**

$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \alpha_t \boldsymbol{m}_t$

$\boldsymbol{v}_t \leftarrow \beta_2 \cdot \boldsymbol{v}_{t-1} + (1 - \beta_2)\, \boldsymbol{g}_t \odot \boldsymbol{g}_t$

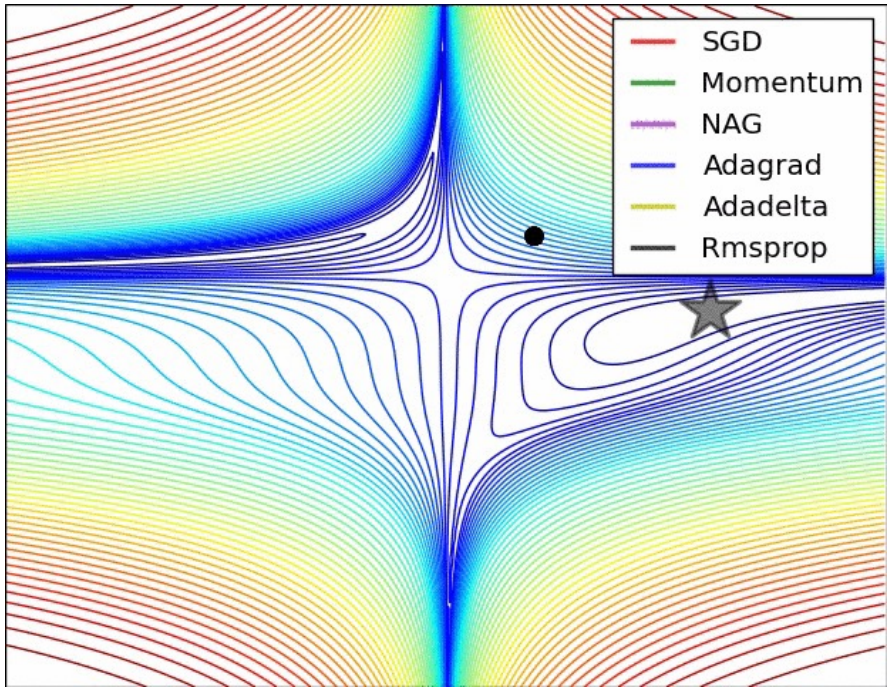
$\boldsymbol{m}_{t+1} \leftarrow \beta_1 \cdot \boldsymbol{m}_t + (1 - \beta_1) \frac{\boldsymbol{g}_{t+1}}{\max\{\sqrt{\boldsymbol{v}_t}, \epsilon\}}$

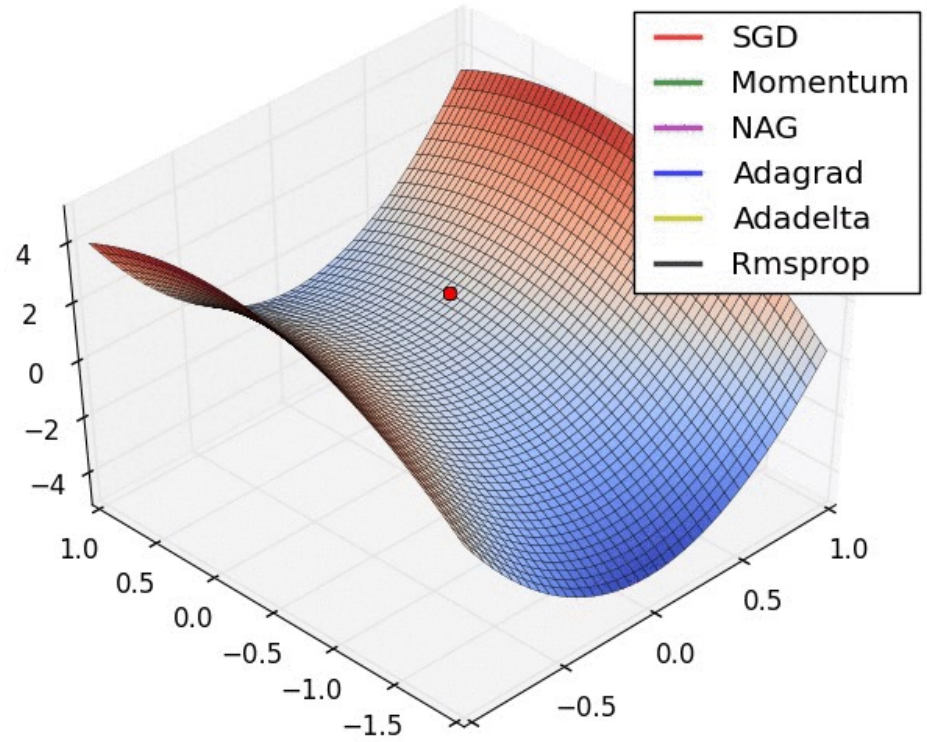
end for

return $\{\boldsymbol{\theta}_t\}_{t=1}^{T}$

https://arxiv.org/pdf/2411.02853

Comparison

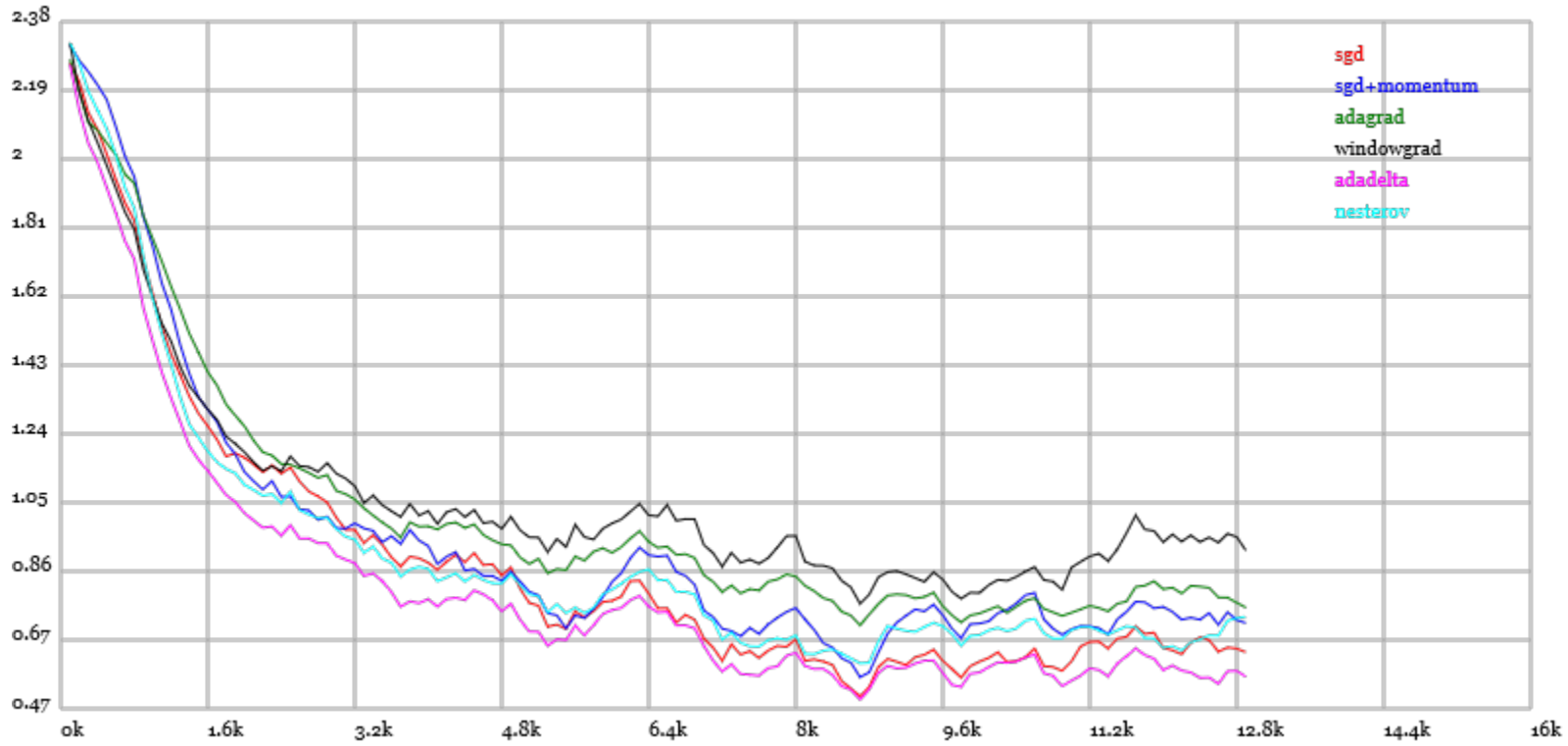




NAG: Nesterov's Accelerated Gradient

https://twitter.com/alecrad

Comparison


2.38
2.19
2
1.81
1.62
1.43
1.24
1.05
0.86
0.67
0.47
0k
1.6k
3.2k
4.8k
6.4k
8k
9.6k
11.2k
12.8k
14.4k
16k
sgd
sgd+momentum
adagrad
windowgrad
adadelta
nesterov

- When SGD+momentum is tuned for hyperparameters, it can outperform Adam etc.
- There are methods that try to finetune the hyper-parameters:

YellowFin and the Art of Momentum Tuning

https://arxiv.org/abs/1706.03471

To sum up

- Different problems seem to favor different per-parameter methods
- Adam seems to perform better among per-parameter adaptive learning rate algorithms
- SGD+Nesterov momentum seems to be a fair alternative