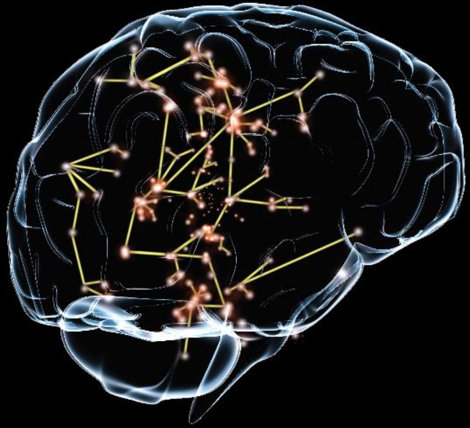


Artificial neural networks

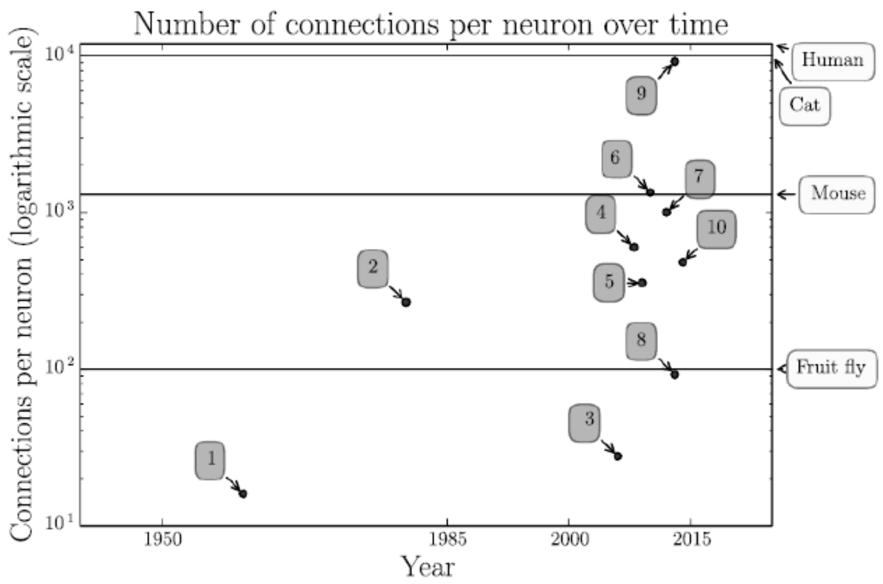
Now

- Neurons
- Neuron models
- Perceptron learning
- Multi-layer perceptrons
- Backpropagation

It all starts with a neuron

Some facts about human brain

- ~ 86 billion neurons
- ~ 10^{15} synapses
-



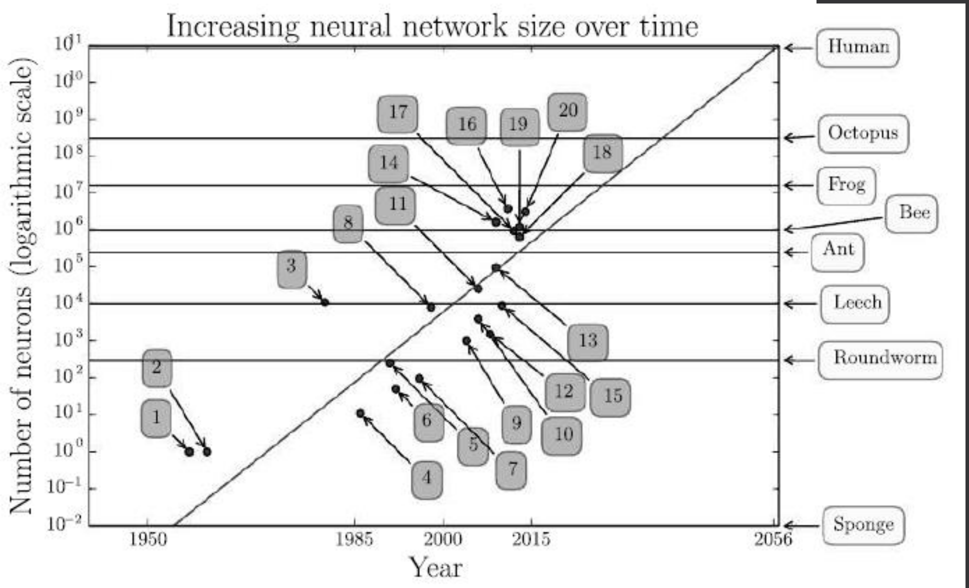
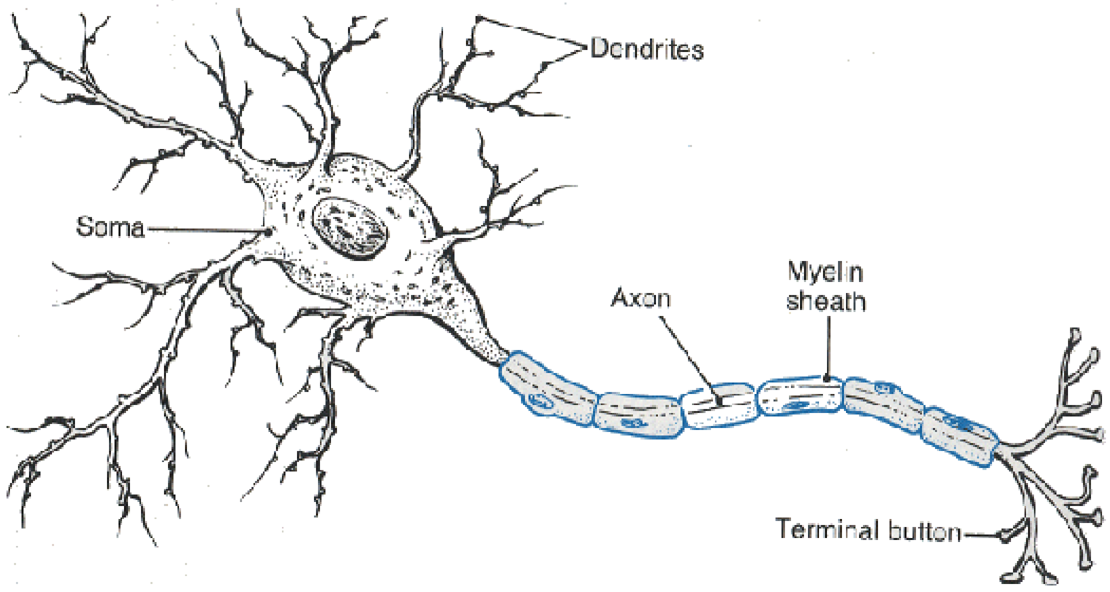


Fig: I. Goodfellow

Neuron

The basic information processing element of neural systems. The neuron

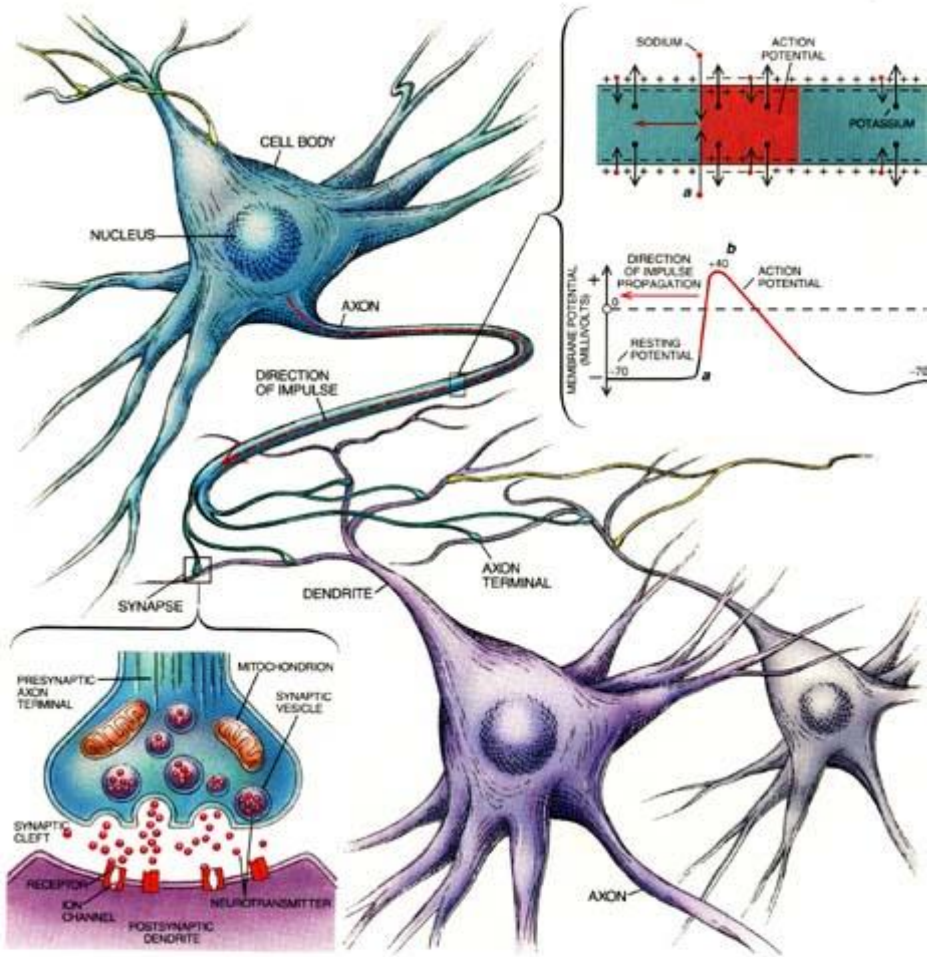
- receives input signals generated by other neurons through its dendrites,
- integrates these signals in its body,
- then generates its own signal (a series of electric pulses) that travel along the axon which in turn makes contacts with dendrites of other neurons.
- The points of contact between neurons are called synapses.



http://animatlab.com/Help/Documentation/Neural-Network-Editor/Neural-Simulation-Plug-ins/Firing-Rate-Neural-Plug-in/Neuron-Basics

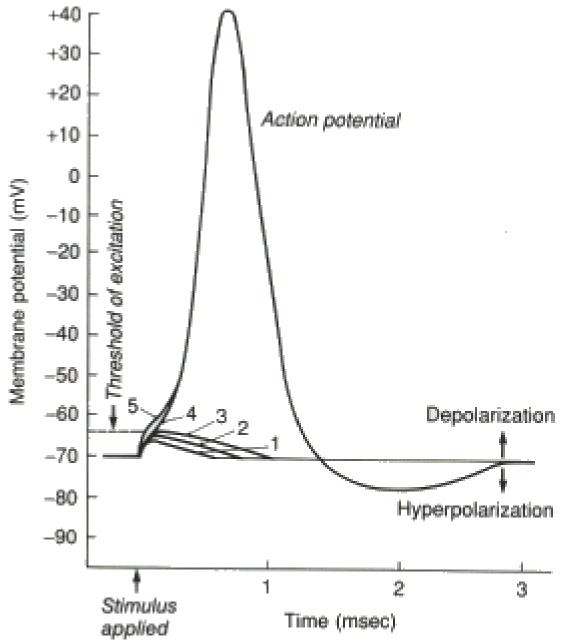
Neuron

- The pulses generated by the neuron travels along the axon as an electrical wave.
- Once these pulses reach the synapses at the end of the axon open up chemical vesicles exciting the other neuron.

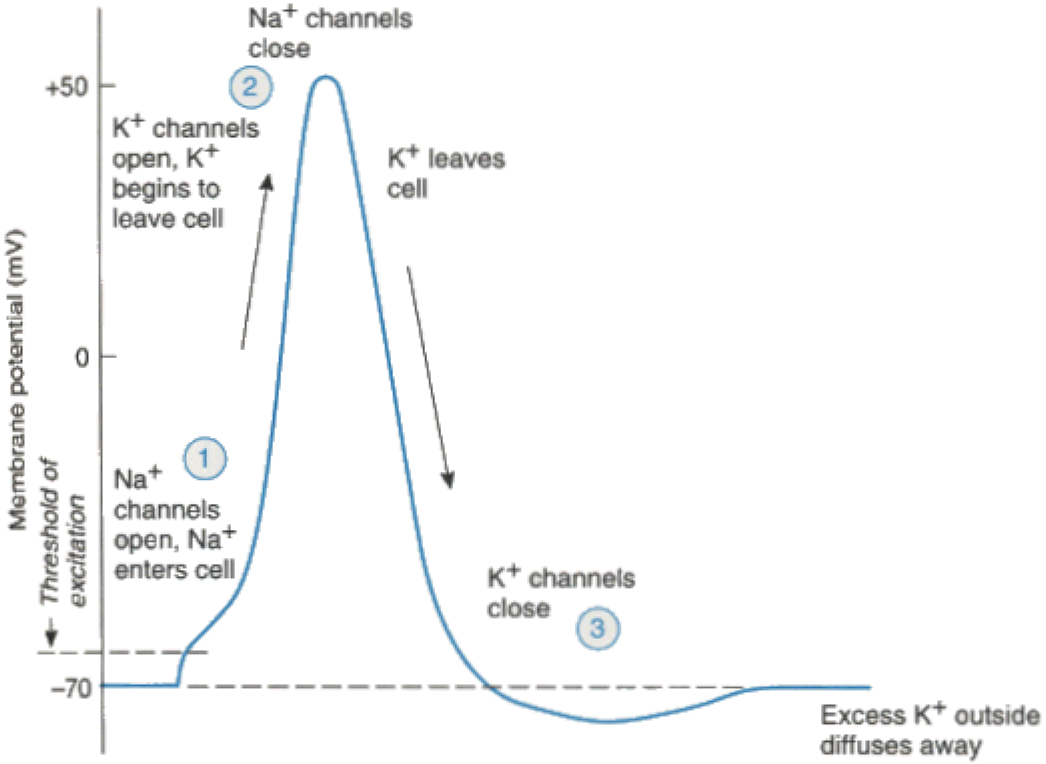


Slide credit: Erol Sahin

Neuron



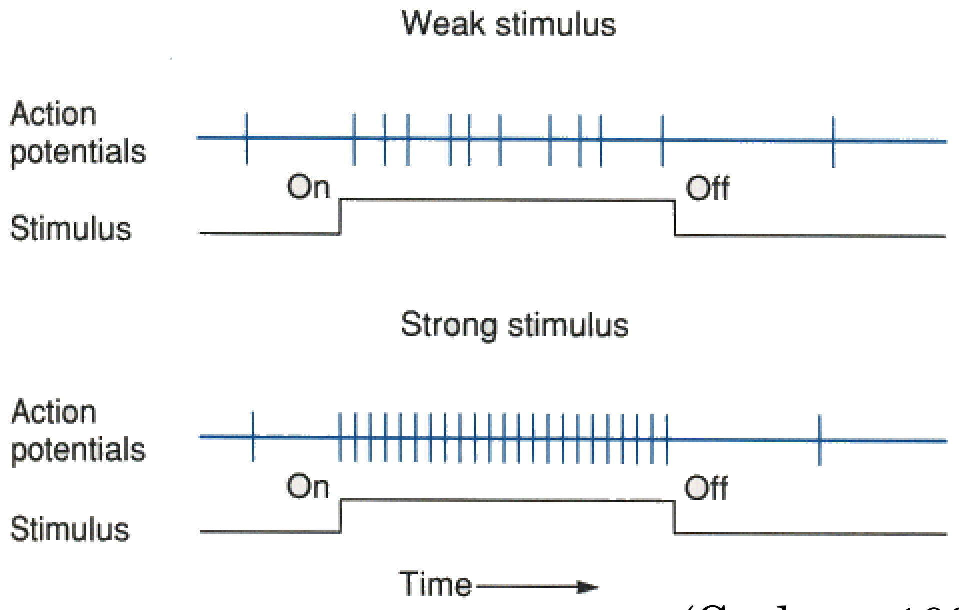
(Carlson, 1992)



(Carlson, 1992)

http://animatlab.com/Help/Documentation/Neural-Network-Editor/Neural-Simulation-Plug-ins/Firing-Rate-Neural-Plug-in/Neuron-Basics

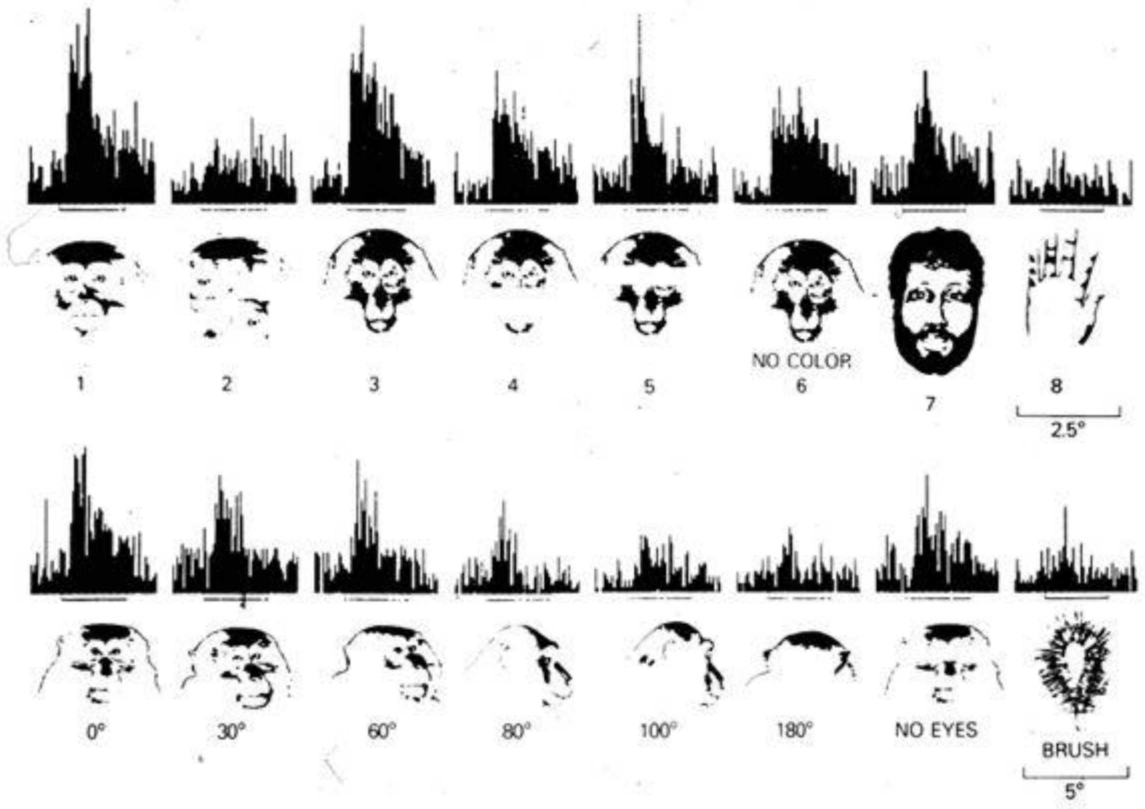
The biological neuron - 2



(Carlson, 1992)

http://animatlab.com/Help/Documentation/Neural-Network-Editor/Neural-Simulation-Plug-ins/Firing-Rate-Neural-Plug-in/Neuron-Basics

Face selectivity in IT



http://www.billconnelly.net/?p=291

Artificial neuron

History of artificial neurons

- Threshold Logic Unit, or Linear Threshold Unit, a.k.a. McCulloch Pitts Neurons – 1943
- Perceptron by Rosenblatt
  - "This model already considered more flexible weight values in the neurons, and was used in machines with adaptive capabilities. The representation of the threshold values as a bias term was introduced by Bernard Widrow in 1960 – see ADALINE."
- "In the late 1980s, when research on neural networks regained strength, neurons with more continuous shapes started to be considered. The possibility of differentiating the activation function allows the direct use of the gradient descent and other optimization algorithms for the adjustment of the weights. Neural networks also started to be used as a general function approximation model. The best known training algorithm called backpropagation has been rediscovered several times but its first development goes back to the work of Paul Werbos"

The history

- 1962 - Frank Rosenblatt: “Back-propagating error-correction procedures”. In Principles of Neurodynamics.
- 1974 - Paul Werbos: “Beyond regression: new tools for prediction and analysis in the behavioral sciences”. Ph.D. thesis. Harvard University.
- 1986 - D. E. Rumelhart, G. E. Hinton, and R. J. Williams. “Learning Internal Representations by Error Propagation”, published in “Parallel Distributed Processing” volume I and II, by the PDP group of UCSD.
- 1986- today - the interest about the neural networks is on the rise..

BRAIN AND COGNITION **33,** 295–305 (1997)
ARTICLE NO. BR970869

Bain on Neural Networks

ALAN L. WILKES AND NICHOLAS J. WADE

University of Dundee, Dundee, Scotland

In his book *Mind and body* (1873), Bain set out an account in which he related the processes of associative memory to the distribution of activity in neural groupings—or neural networks as they are now termed. In the course of this account, Bain anticipated certain aspects of connectionist ideas that are normally attributed to 20th-century authors—most notably Hebb (1949). In this paper we reproduce Bain's arguments relating neural activity to the workings of associative memory which include an early version of the principles enshrined in Hebb's neurophysiological postulate. Nonetheless, despite their prescience, these specific contributions to the connectionist case have been almost entirely ignored. Eventually, Bain came to doubt the practicality of his own arguments and, in so doing, he seems to have ensured that his ideas concerning neural groupings exerted little or no influence on the subsequent course of theorizing in this area.

Alexander Bain (1818–1903), see Fig. 1, is best known for his textbooks *The senses and the intellect* (1855) and *The emotions and the will* (1859), in which he offered an interpretation of mental phenomena within an associationist framework (for further biographical detail, see Hearnshaw, 1964). Specifically, he tried to match quantitative estimates of the associations held in memory to the neural structure of the brain. It was this exercise that first drew Bain into confronting the potential properties of neural groupings or networks. In the course of thinking about these issues, he was led to speculate on how the internal structure of neural groupings could *physically grow* to reflect the contingencies of experience and how this same internal structure could come to support the variety of associative links typically found in memory.

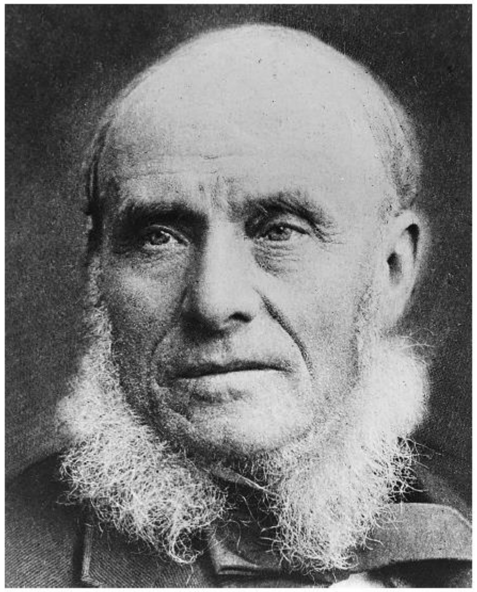


FIG. 1. Alexander Bain in 1892 from a photograph in his *Autobiography* (1904).

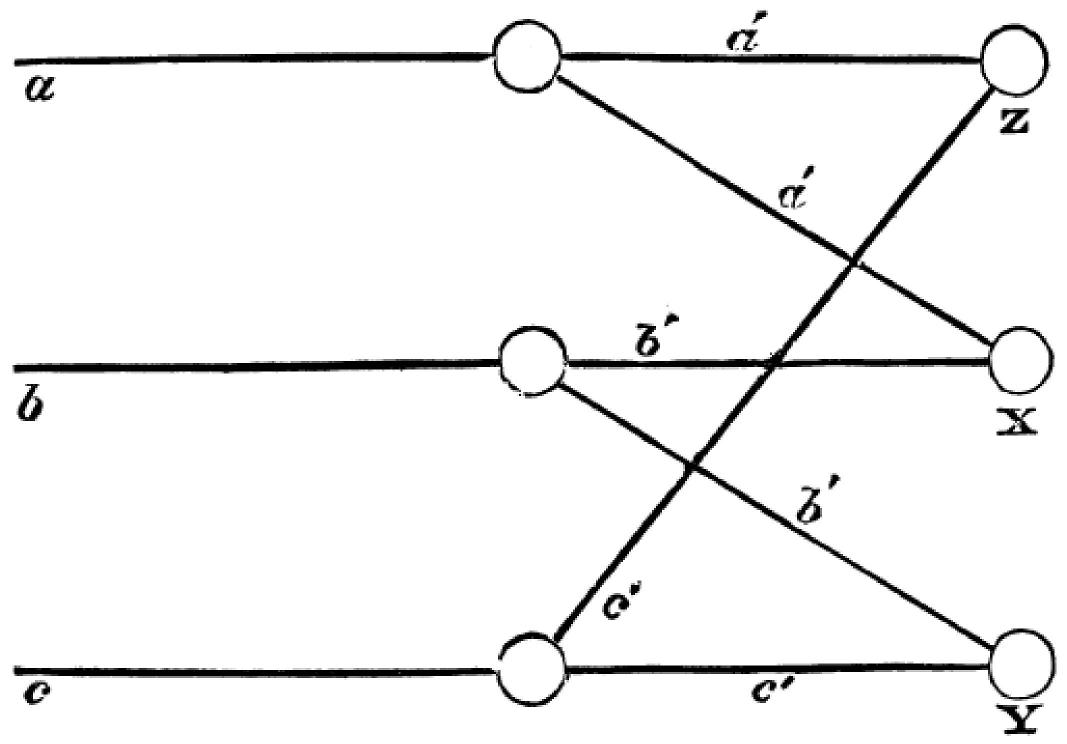


FIG. 2. Bain's diagram illustrating the way in which the connections in a neural network can channel activation in different directions:

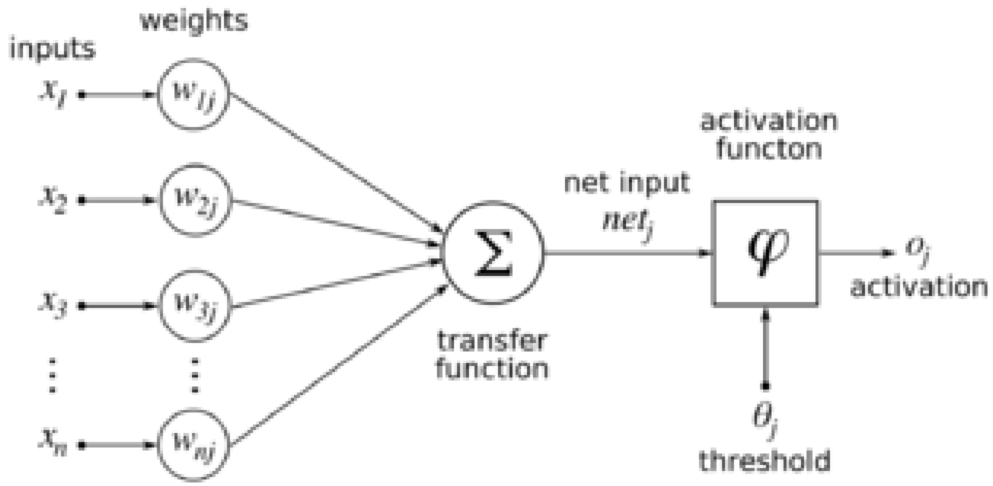
> It requires us to assume, not merely fibres multiplying by ramification through the cell junctions, but also an extensive arrangement of cross connections. We can typify it in this way. Suppose *a* enters a cell junction, and is replaced by several branches, *a*′, *a*′ etc; *b* in like manner, is multiplied into *b*′, *b*′ etc. Let one of the branches of *a* or *a*′, pass into some second cell, and a branch of *b,* or *b*′, pass into the same, and let one of the emerging branches be *X,* we have then a means of connecting united *a* and *b* with *X;* and in some other crossing, a branch of *b* may unite with a branch of *c,* from which crossing *Y* emerges and so on. . . . By this plan we comply with the primary condition of assigning a separate outcome to every different combination of sensory impressions.
>
> The diagram shows the arrangement. The fibre *a* branches into two *a*′, *a*′; the fibre *b* into *b*′, *b*′; *c*′, *c*′. One of the branches of *a* unites with one of the branches of *b,* or *a*′, *b*′ in a cell *X; b*′, *c*′ unite in *Y; a*′, *c*′ in *Z.* (1873, pp. 110, 111)

McCulloch-Pitts Neuron (McCulloch & Pitts, 1943)

- Binary input-output
- Can represent Boolean functions.
- No training.

$$net = \sum_i (w_{y,x_i} x_i) + w_{y,x_b}$$

$$f(net) = \begin{cases} 0, & net < 0 \\ 1, & net \geq 0 \end{cases}$$

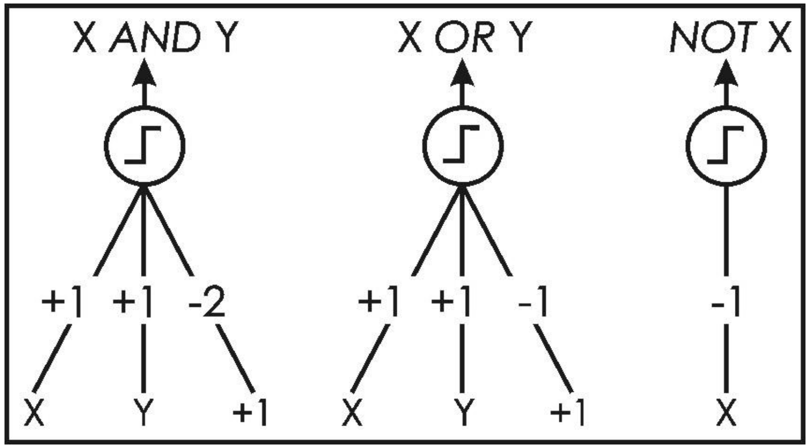


http://www.tau.ac.il/~tsirel/dump/Static/knowino.org/wiki/Artificial_neural_network.html

McCulloch-Pitts Neuron

- Implement AND:
  - Let w_{x1} and w_{x2} to be 1, and w_{xb} to be -2.

- When input is 1 & 1; net is 0.
- When one input is 0; net is -1.
- When input is 0 & 0; net is -2.



http://www.webpages.ttu.edu/dleverin/neural_network/neural_networks.html

http://www.tau.ac.il/~tsirel/dump/Static/knowino.org/wiki/Artificial_neural_network.html

McCulloch-Pitts Neuron

Wikipedia:

- “Initially, only a simple model was considered, with binary inputs and outputs, some restrictions on the possible weights, and a more flexible threshold value. Since the beginning it was already noticed that any boolean function could be implemented by networks of such devices, what is easily seen from the fact that one can implement the AND and OR functions, and use them in the disjunctive or the conjunctive normal form. Researchers also soon realized that cyclic networks, with feedbacks through neurons, could define dynamical systems with memory, but most of the research concentrated (and still does) on strictly feed-forward networks because of the smaller difficulty they present.”

McCulloch-Pitts Neuron

- Binary input-output is a big limitation
- Also called

  “[…] *caricature models since they are intended to reflect one or more neurophysiological observations, but without regard to realism* […]”

  -- Wikipedia

- No training! No learning!
- They were useful in inspiring research into connectionist models

Hebb's Postulate (Hebb, 1949)

- "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased"

In short: Neurons that fire together, wire together.

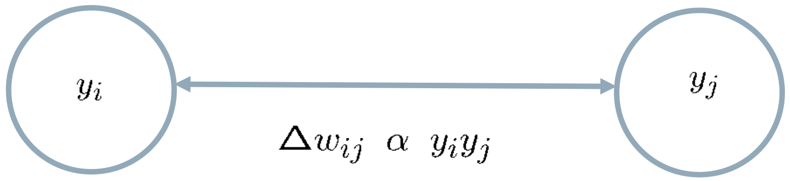
In other words:

$$w_{ij} \propto x_i x_j$$

THIS ALSO APPLIES TO NIGHT'S WATCH
NIGHTSMEN THAT FIGHT TOGETHER DIE TOGETHER
memegenerator.net

Hebb’s Learning Law

- Very simple to formulate as a learning rule:



- If the activation of the neurons, y1 and y2 , are both on (+1) then the weight between the two neurons grow. (Off: 0)
- Else the weight between remains the same.

- However, when bipolar activation {-1,+1} scheme is used, then the weights can also decrease when the activation of two neurons does not match.

Slide credit: Erol Sahin

Psychological Review
Vol. 65, No. 6, 1958

THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN[1]

F. ROSENBLATT

Cornell Aeronautical Laboratory

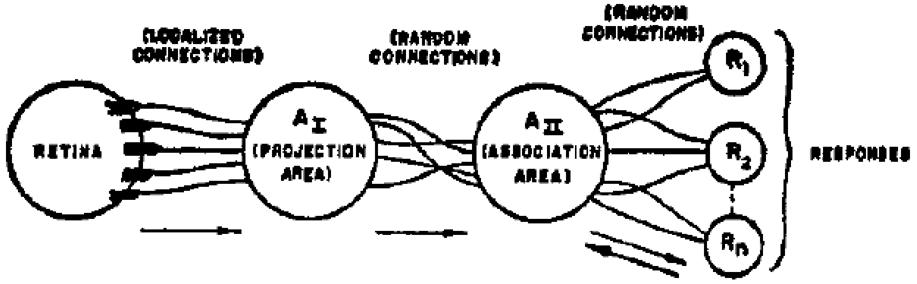
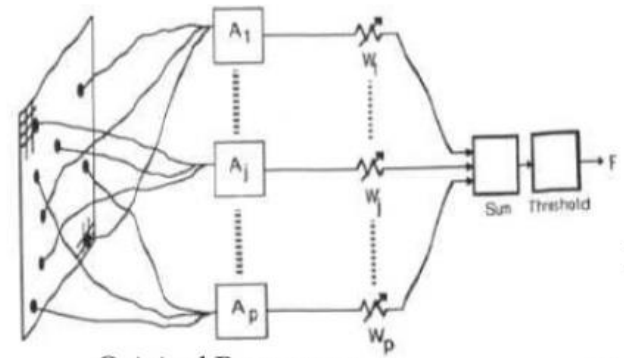


FIG. 1. Organization of a perceptron.

Perceptron (1957)

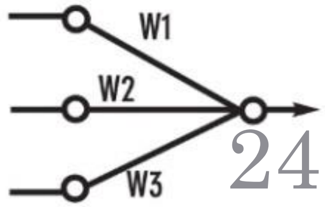


Original Perceptron

(From Perceptrons by M. L Minsky and S. Papert, 1969, Cambridge, MA: MIT Press. Copyright 1969 by MIT Press.

Frank Rosenblatt
(1928-1971)

Simplified model:



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

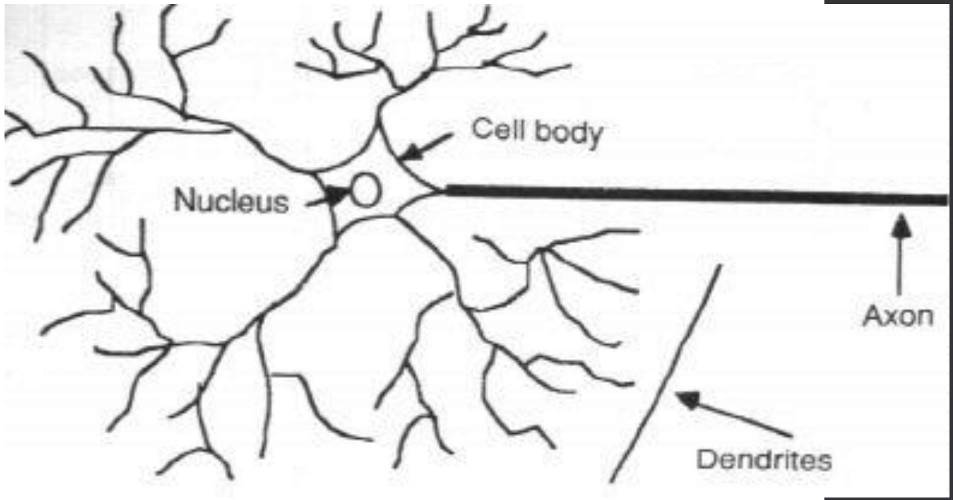
https://www.youtube.com/watch?v=aygSMgK3BEM

And many others

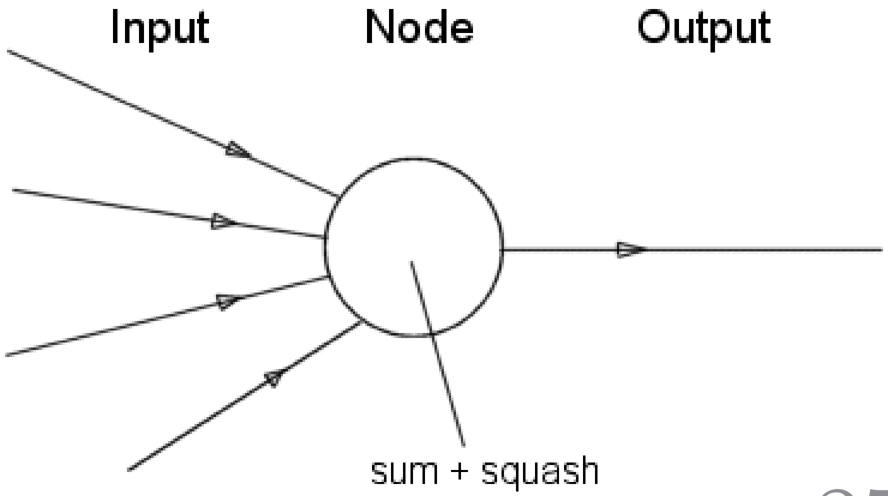
- Widrow & Hoff, 1962
- Grossberg, 1969
- Kohonen, 1972
- von der Malsburg, 1973
- Narendra & Thathtchar, 1974
- Palm, 1980
- Hopfield, 1982

Let's go back to a biological neuron

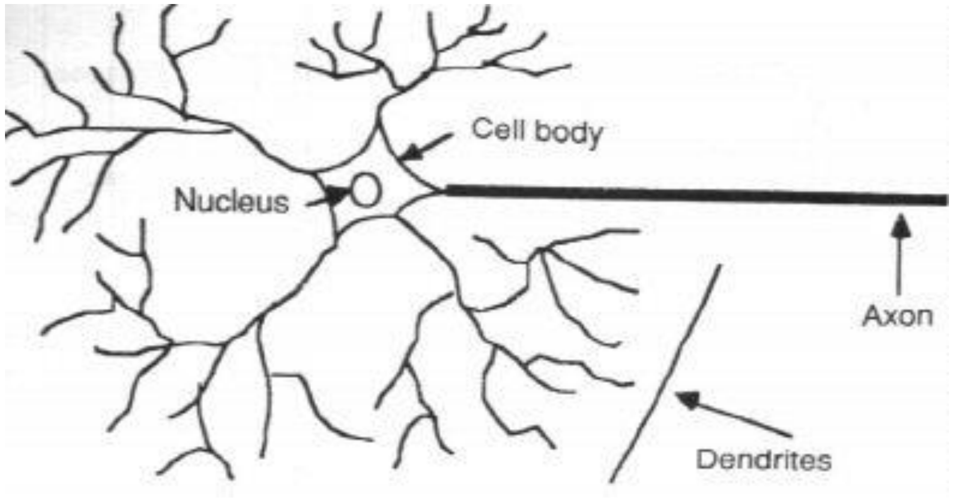
- A biological neuron has:
  - Dendrites
  - Soma
  - Axon

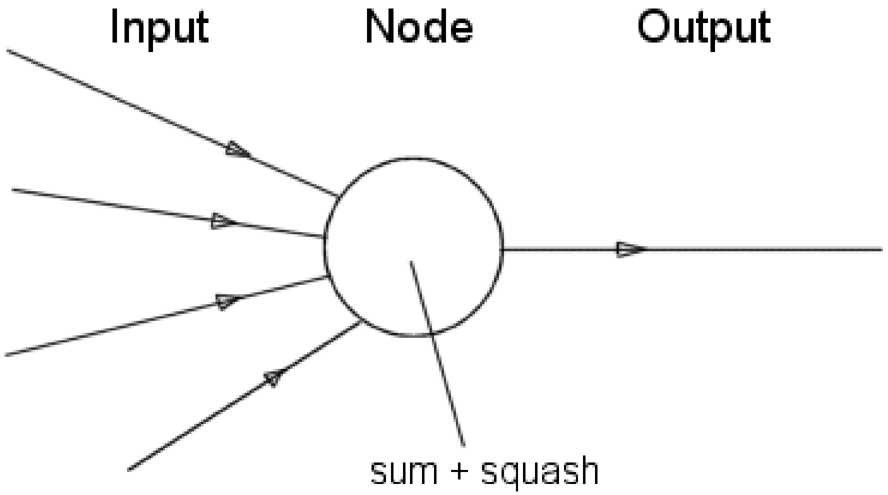


- Firing is continuous, unlike most artificial neurons
- Rather than the response function, the firing rate is critical



- Neurone vs. Node





- Very crude abstraction
- Many details overseen

“Spherical cow” problem!

Spherical cow

Q: How does a physicist milk a cow?
A: Well, first let us consider a spherical cow...

Or

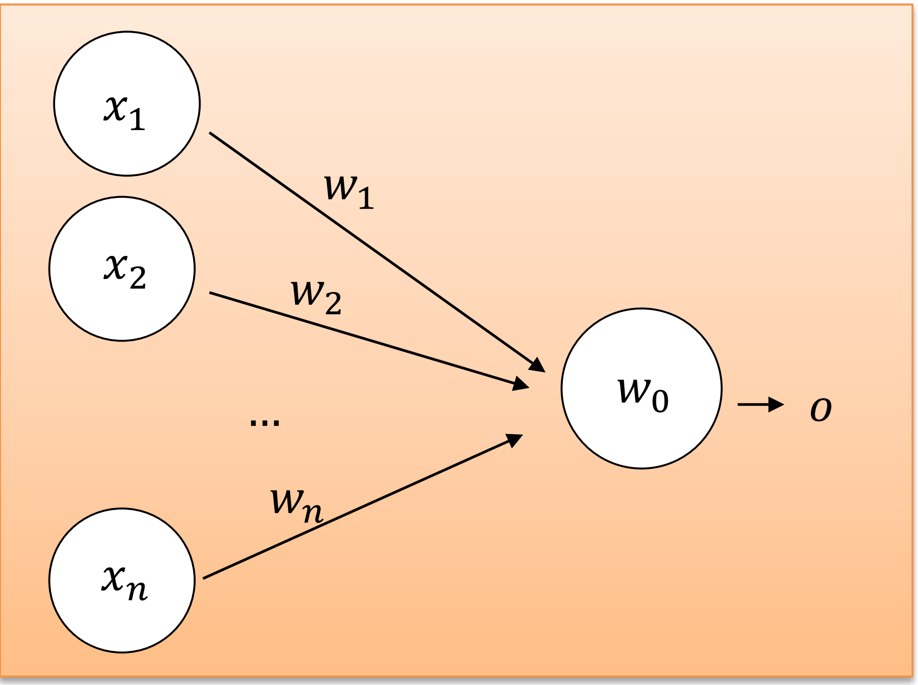
https://en.wikipedia.org/wiki/Spherical_cow

“Milk production at a dairy farm was low, so the farmer wrote to the local university, asking for help from academia. A multidisciplinary team of professors was assembled, headed by a theoretical physicist, and two weeks of intensive on-site investigation took place. The scholars then returned to the university, notebooks crammed with data, where the task of writing the report was left to the team leader. Shortly thereafter the physicist returned to the farm, saying to the farmer, "I have the solution, but it only works in the case of spherical cows in a vacuum".”

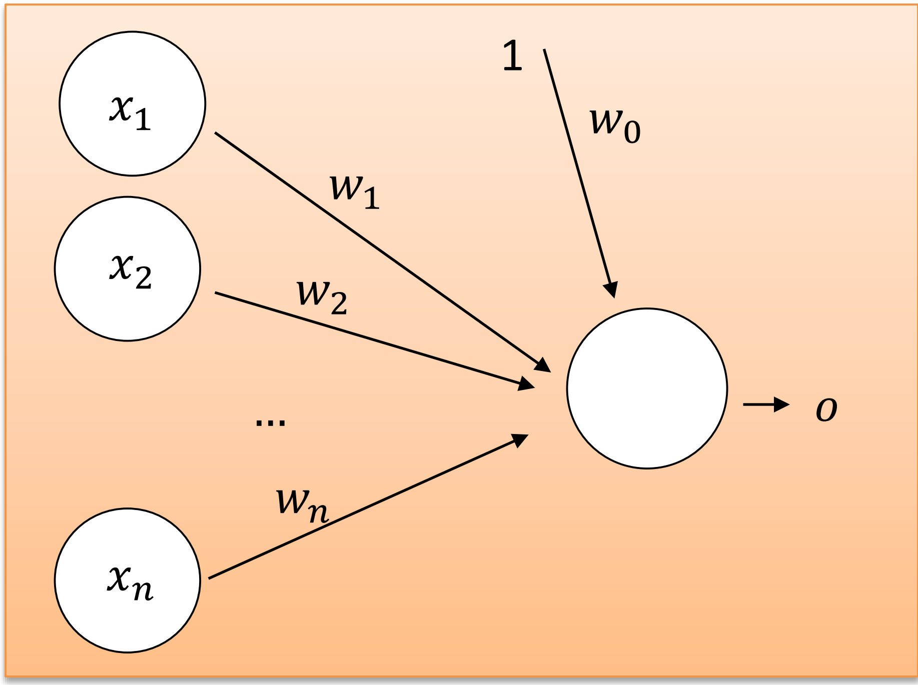
https://www.washingtonpost.com/news/wonk/wp/2013/09/04/the-coase-theorem-is-widely-cited-in-economics-ronald-coase-hated-it/

Let us take a closer look at perceptrons

- Initial proposal of connectionist networks
- Rosenblatt, 50's and 60's
- Essentially a linear discriminant composed of nodes and weights



or



Activation Function

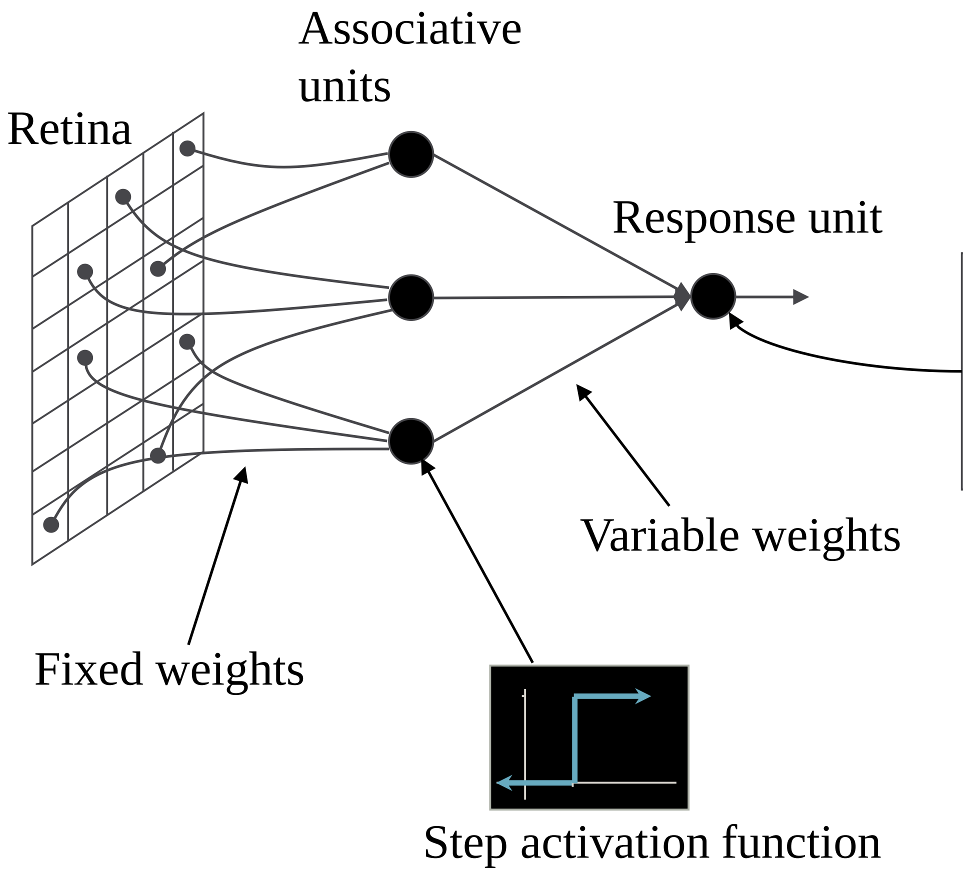
$$o(\mathbf{x}) = \begin{cases} 1, & w_0 + w_1x_1 + .. w_nx_n > 0 \\ 0, & \text{otherwise} \end{cases}$$

Or, simply

$$o(\mathbf{x}) = \text{sgn}(\mathbf{w} \cdot \mathbf{x})$$

where $sgn(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise} \end{cases}$

Perceptron – clearer structure

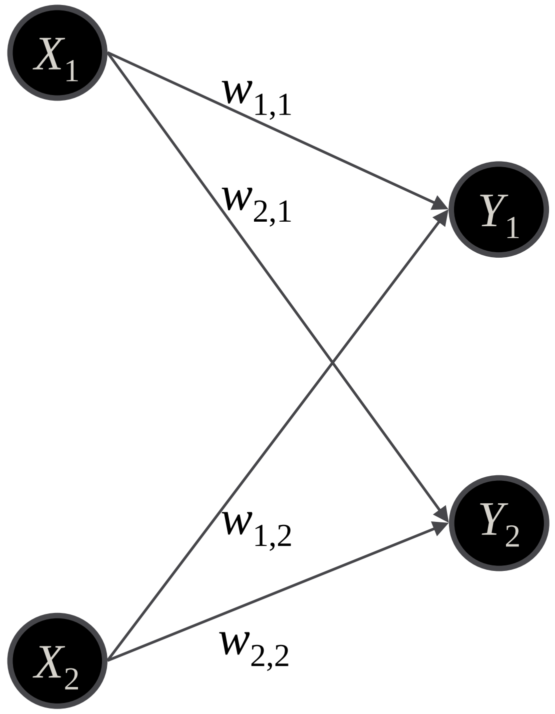


$$f(y\_in) = \begin{cases} 1 & \text{if } y\_in > \theta \\ 0 & \text{if } -\theta \le y\_in \le \theta \\ -1 & \text{if } y\_in < -\theta \end{cases}$$

Slide adapted from Dr. Nigel Crook from Oxford Brookes University
Slide credit: Erol Sahin

Perceptron - activation

Simple matrix multiplication, as we have seen in the previous lecture



$$\mathbf{Wx} = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$= \begin{bmatrix} w_{1,1}x_1 + w_{1,2}x_2 \\ w_{2,1}x_1 + w_{2,2}x_2 \end{bmatrix}$$

$$= \begin{bmatrix} \sum_{j=1}^{2} w_{1,j}x_j \\ \sum_{j=1}^{2} w_{2,j}x_j \end{bmatrix}$$

Slide adapted from Dr. Nigel Crook from Oxford Brookes University
Slide credit: Erol Sahin

Motivation for perceptron learning

(No gradient descent yet)

- We have estimated an output o
  - But the target was t
- Error (simply): $t - o$
- Let us update each weight such that we "learn" from the error:
  - $w_i \leftarrow w_i + \Delta w_i$
  - where $\Delta w_i \propto (t - o)$
- We somehow need to distribute the error to the weights. How?
  - Distribute the error according to how much they contributed to the error: Bigger input contributes more to the error.
  - Therefore: $\Delta w_i \propto (t - o) x_i$

An example

- Consider $x_i = 0.8, t = 1, o = -1$
  - Then, $(t - o)x_i = 1.6$
  - Which will increase the weight
  - Which makes sense considering the output and the target

Perceptron training rule

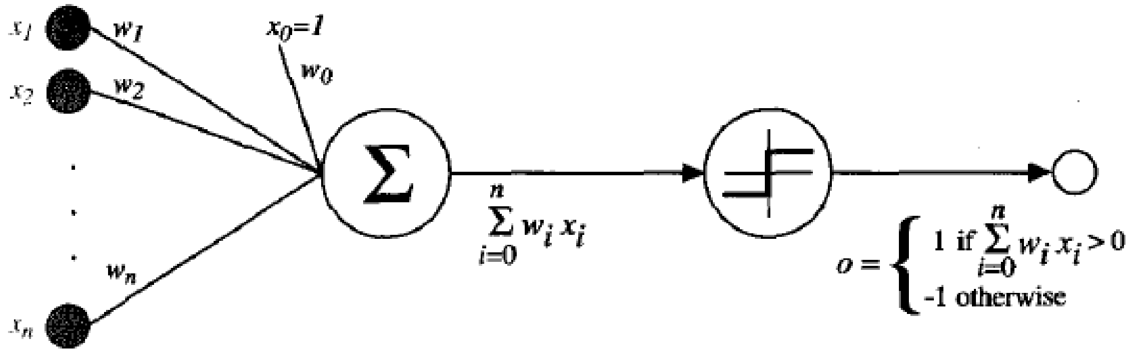
- Update weights

$$w_i \leftarrow w_i + \Delta w_i$$

- How to determine Δw_i?

$$\Delta w_i \leftarrow \eta(t - o)x_i$$

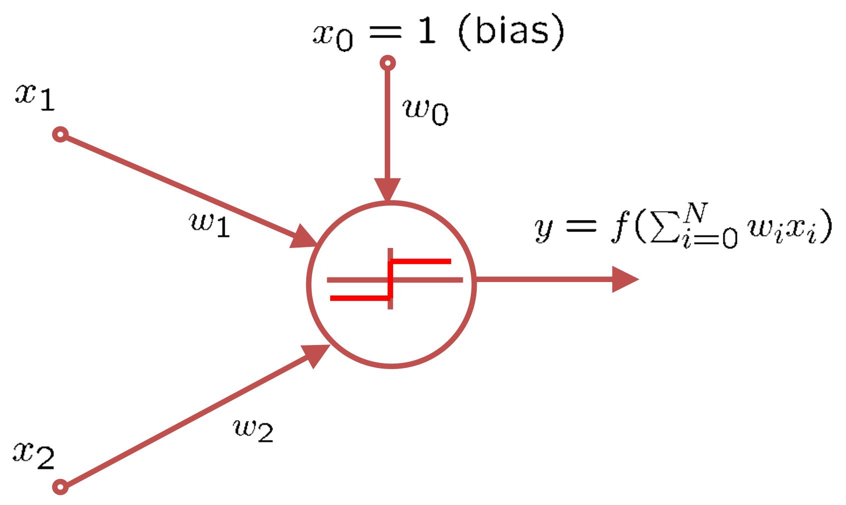
  - η: learning rate – can be slowly decreased
  - t: target/desired output
  - o: current output

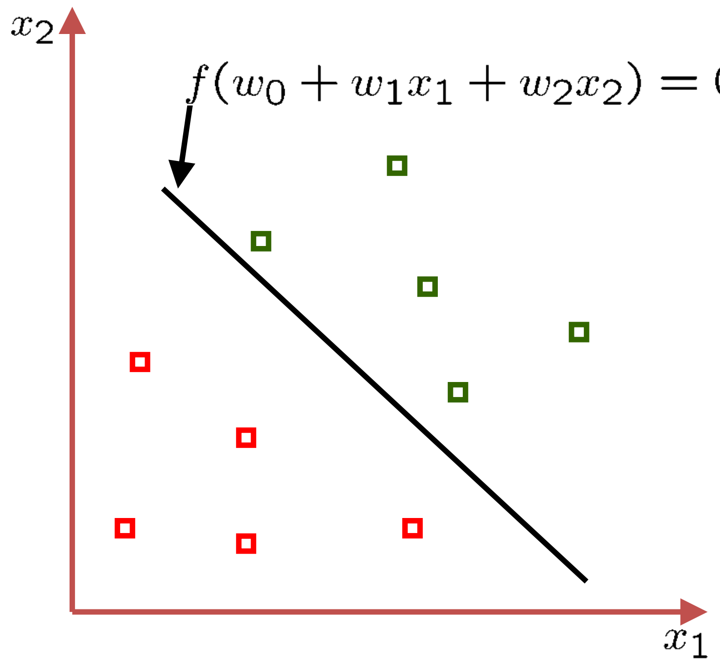


x_1
x_2
x_n
w_1
w_2
w_n
$x_0=1$
w_0
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases}$

Perceptron - intuition

- A perceptron defines a hyperplane in N-1 space: a line in 2-D (two inputs), a plane in 3-D (three inputs),….
- The perceptron is a linear classifier: It's output is -1 on one side of the plane, and 1 for the other.
- Given a linearly separable problem, the perceptron learning rule guarantees convergence.





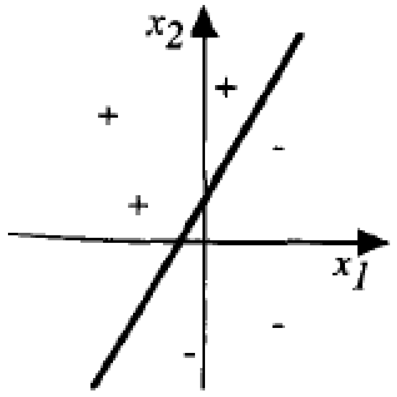
Slide credit: Erol Sahin

Problems with perceptron

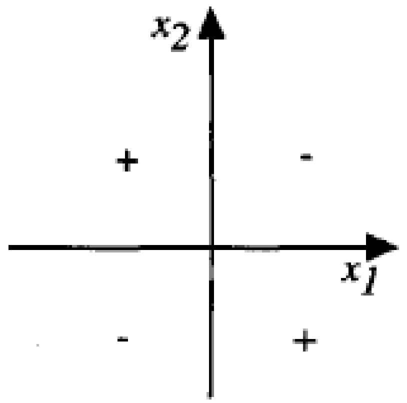
- Perceptron unit is non-linear
- However, it is not differentiable (due to thresholding), which makes it unsuitable to gradient descent in multi-layer networks.

Problems with perceptron learning

- Can only learn linearly separable classification.



linearly separable



not linearly separable

Gradient Descent

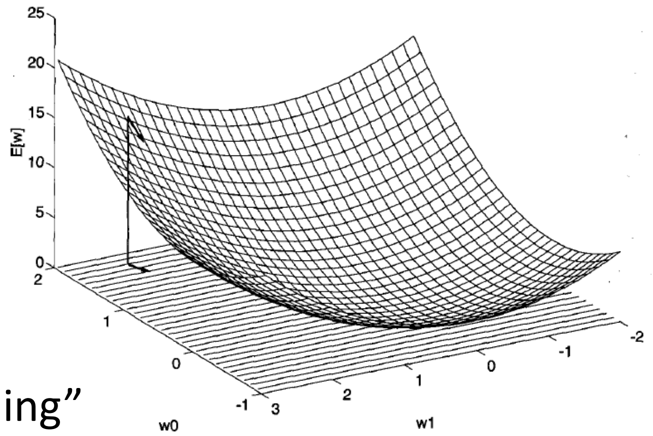
- Consider unthresholded perceptron:

$$o(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$$

- We can calculate the error of the perceptron:

$$E(\boldsymbol{w}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

- We can guide the search for better weights by using the gradient of this error function.



T. M. Mitchell, “Machine Learning”

Gradient Descent Rule

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \Delta\boldsymbol{w}$$

- Determine $\Delta\boldsymbol{w}$ based on the error function:

$$\Delta\boldsymbol{w} \leftarrow -\eta\nabla E(\boldsymbol{w})$$

- For the individual weights:

$$\boxed{\Delta w_i \leftarrow -\eta \frac{\partial E}{\partial w_i}}$$

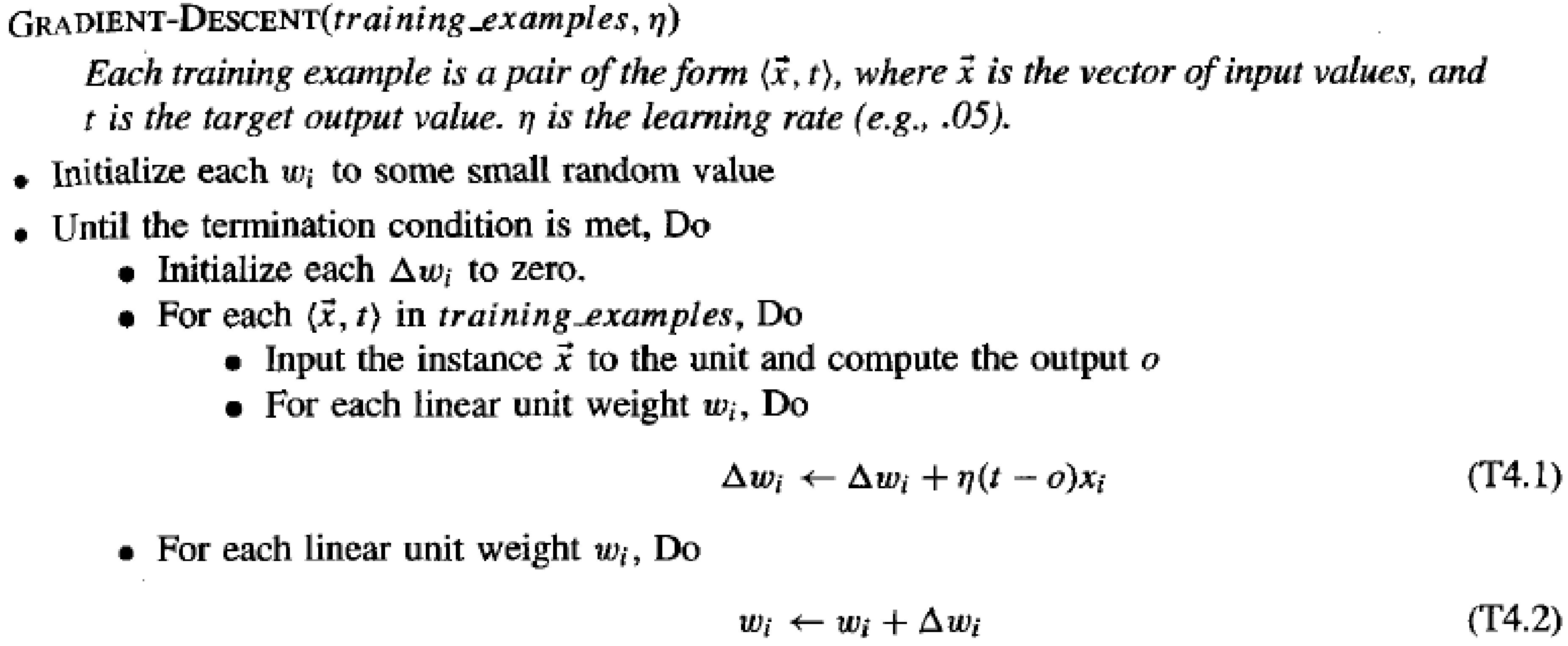
$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x}_d)
\end{aligned}$$

$$\boxed{\frac{\partial E}{\partial w_i} = \sum_{d\in D}(t_d - o_d)(-x_{id})}$$

$$\Delta w_i \leftarrow \eta \sum_{d\in D}(t_d - o_d)x_{id}$$

T. M. Mitchell, “Machine Learning”

Gradient Descent Training Algorithm



```
GRADIENT-DESCENT($training\_examples$, $\eta$)
    Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector of input values, and
    $t$ is the target output value. $\eta$ is the learning rate (e.g., .05).
• Initialize each $w_i$ to some small random value
• Until the termination condition is met, Do
      • Initialize each $\Delta w_i$ to zero.
      • For each $\langle \vec{x}, t \rangle$ in $training\_examples$, Do
            • Input the instance $\vec{x}$ to the unit and compute the output $o$
            • For each linear unit weight $w_i$, Do
                  $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$          (T4.1)
      • For each linear unit weight $w_i$, Do
                  $w_i \leftarrow w_i + \Delta w_i$          (T4.2)
```

T. M. Mitchell, "Machine Learning"

Stochastic Gradient Descent or Incremental Gradient Descent

- Difficulties of gradient descent:
  - Convergence to a local minimum can be quite slow
  - If there are multiple local minima, no guarantee on finding the global minimum
- One alternative:
  - update the weight after seeing each sample.

$$\Delta w_i \leftarrow \eta(t - o)x_i$$

Compare to (in standard GD):

$$\Delta w_i \leftarrow \eta \sum_{d \in D} (t_d - o_d) x_{id}$$

- Error effectively becomes (per data):

$$E_d(w) = \frac{1}{2}(t_d - o_d)^2$$

(Delta rule, least-mean-square-rule, Adaline rule or Widrof-Hoff rule)

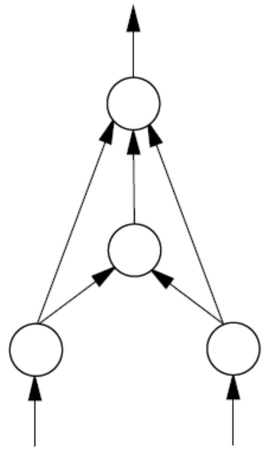
Notes on convergence

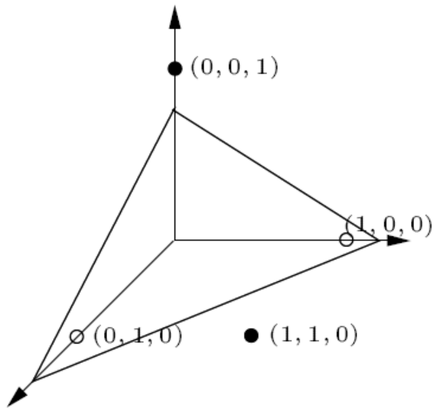
- Perceptron learning:
  - Output is thresholded
  - Converges after a finite number of iterations to a hypothesis that perfectly classifies the data
  - Condition: data is linearly separable

- Gradient descent (delta rule):
  - Output is not thresholded
  - Converges asymptotically to the minimum error hypothesis
  - Condition: unbounded time
  - Does not require linear separation.

The limitations of a perceptron: A hidden neuron may help

Input		Output
001	⇒	0
010	⇒	1
100	⇒	1
110	⇒	0





LET’S GET MULTI-LAYER

Multi-layer Networks

Information flow is unidirectional

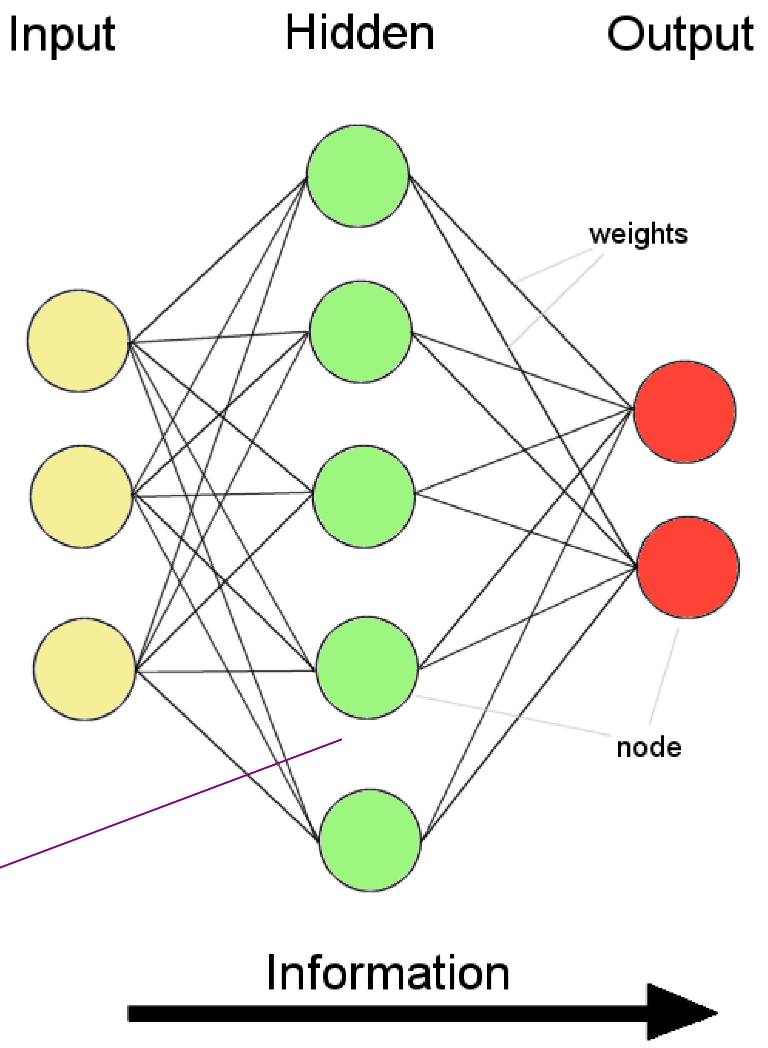
Data is presented to *Input layer*

Passed on to *Hidden Layer*

Passed on to *Output layer*

Information is distributed

Information processing is parallel



Internal representation (interpretation) of data

Multi-layered Networks

- To be able to have solutions for linearly non-separable cases, we need a non-linear and differentiable unit.

$$o = \sigma(\vec{w} \cdot \vec{x})$$

where:

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

$$\frac{d\sigma(y)}{dy} = \sigma(y) \cdot (1 - \sigma(y))$$

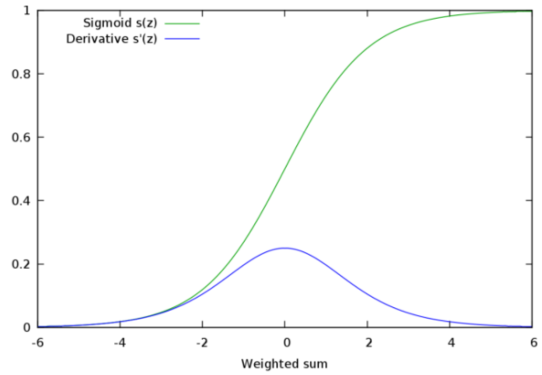
Let's denote the sigmoid function as $\sigma(x) = \frac{1}{1+e^{-x}}$.

The derivative of the sigmoid is $\frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x))$.

Here's a detailed derivation:

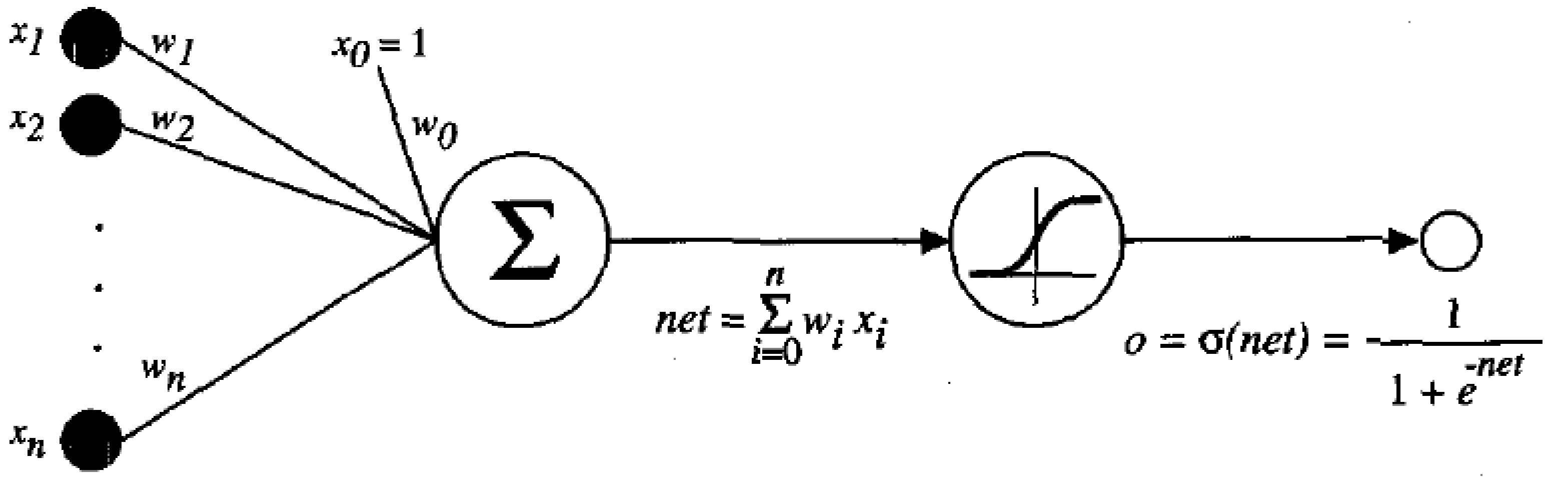
$$\begin{aligned}
\frac{d}{dx}\sigma(x) &= \frac{d}{dx}\left[\frac{1}{1+e^{-x}}\right] \\
&= \frac{d}{dx}(1+\mathrm{e}^{-x})^{-1} \\
&= -(1+e^{-x})^{-2}(-e^{-x}) \\
&= \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{(1+e^{-x})-1}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) \\
&= \sigma(x) \cdot (1-\sigma(x))
\end{aligned}$$

- Sigmoid (logistic) function
- Output is in (0,1)
- Since it maps a large domain to (0,1) it is also called squashing function
- Alternatives: *tanh*



(Eq: M. Percy)

Perceptron with sigmoid function



x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

Why do we need to learn backpropagation?

- “Many frameworks implement backpropagation for us, why do we need to learn?”
  - This is not a blackbox. There are many problems/issues involved. You can only deal with them if you have a good understanding of backpropagation.

https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b#.7zawffou2

Backpropagation algorithm

- Let us re-define the error function since we have many outputs:

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{d \in D}\sum_{k \in outputs}(t_{kd} - o_{kd})^2$$

- For one data:

$$E_d(\boldsymbol{w}) = \frac{1}{2}\sum_{k \in outputs}(t_{kd} - o_{kd})^2$$

- For each output unit k, calculate its error term δ_k:

$$\delta_k = -\partial E_d(w)/\partial o_k$$
$$\delta_k = \boxed{o_k(1 - o_k)}(t_k - o_k)$$

Derivative of the Sigmoid function

- For each hidden unit h, calculate its error term δ_h:

$$\delta_h = o_h(1 - o_h)\sum_{k \in outputs} w_{kh}\delta_k$$

- Update every weight w_{ji}

$$w_{ji} = w_{ji} + \eta\delta_j x_{ji}$$

Derivation of backpropagation

- Derivation of the <u>output unit</u> weights

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

- Expand $\frac{\partial E_d}{\partial w_{ji}}$:

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \boxed{\frac{\partial net_j}{\partial w_{ji}}} \longrightarrow x_{ji}$$

- Expand $\frac{\partial E_d}{\partial net_j}$:

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} \boxed{\frac{\partial o_j}{\partial net_j}} \longrightarrow$$

Derivative of sigmoid $o_j(1 - o_j)$

$$\frac{\partial}{\partial o_j} \frac{1}{2} \sum_k (t_k - o_k)^2 = \frac{\partial}{\partial o_j} \frac{1}{2} (t_k - o_k)^2 = -(t_j - o_j)$$

Therefore:

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

$$= \eta \underbrace{(t_j - o_j) o_j (1 - o_j)}_{} x_{ij}$$

Derivation of backpropagation

- Derivation of the output unit weights

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

Therefore:

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

$$= \eta \underbrace{(t_j - o_j)o_j(1 - o_j)}_{\substack{\delta_j \\ \text{(error term for unit j)}}} x_{ij}$$

Derivation of backpropagation

- Derivation of the hidden unit weights

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

- Expand $\frac{\partial E_d}{\partial w_{ji}}$:

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j}\boxed{\frac{\partial net_j}{\partial w_{ji}}} \longrightarrow x_{ji}$$

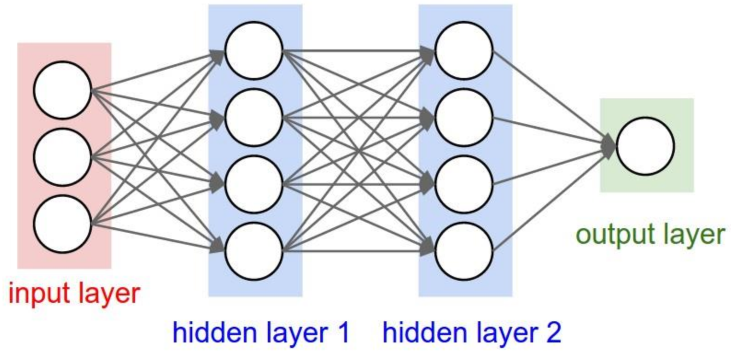
- Expand $\frac{\partial E_d}{\partial net_j}$:

$$\frac{\partial E_d}{\partial net_j} = \sum_k \underbrace{\boxed{\frac{\partial E_d}{\partial net_k}}}_{-\delta_k}\underbrace{\boxed{\frac{\partial net_k}{\partial net_j}}}_{= \frac{\partial net_k}{\partial o_j}\frac{\partial o_j}{\partial net_j} = w_{kj}o_j(1-o_j)}$$

Therefore:

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta \delta_j x_{ji} = \eta x_{ji}\left[o_j(1-o_j)\textstyle\sum_k \delta_k w_{kj}\right]$$

Forward pass



```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

Backpropagation vs. numerical differentiation

- Backpropagation:
  - $O(|W|)$

- Numerical differentiation
  - $O(|W|^2)$

Problems of back propagation with sigmoid

- It is extremely slow, if it does converge.
- It may get stuck in a local minima.
- It is sensitive to initial conditions.
- It may start oscillating.

Backprop in deep networks

- Local minima may not be as severe as it is feared
  - If one weight gets into local minima, other weights provide escape routes
  - The more weights, the more escape routes
- Add a momentum term
- Use stochastic gradient descent, rather than true gradient descent
  - This means we have different error surfaces for each data
  - If stuck in local minima in one of them, the others might help
- Train multiple networks with different initial weights
  - Select the best one

Backprop

- Very powerful - can learn any function, given enough hidden units!
- Have the same problems of Generalization vs. Memorization.
  - With too many units, we will tend to memorize the input and not generalize well. Some schemes exist to “prune” the neural network.
- Networks require extensive training, many parameters to fiddle with. Can be extremely slow to train. May also fall into local minima.
- Inherently parallel algorithm, ideal for multiprocessor hardware.
- Despite the cons, a very powerful algorithm that has seen widespread successful deployment.

Now, let us look at alternative aspects

- Loss functions
  - Hinge-loss, softmax loss, squared-error loss, …
  - We will not look at them here again

- Activation functions
  - Sigmoid, tanh, ReLU, Leaky ReLU, parametric ReLU, maxout

- Backpropagation strategy:
  - True Gradient Descent, Stochastic Gradient Descent, Mini-batch Gradient Descent, RMSpop, AdaDelta, AdaGrad, Adam

Activation Functions

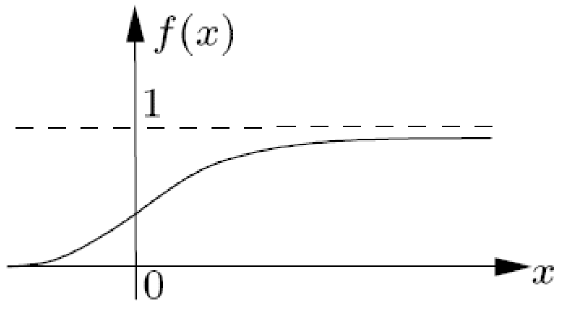
Activation function

- Sigmoid / logistic function

The computational power is increased by the use of a squashing function. In the original paper the logistic function:

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

is used.



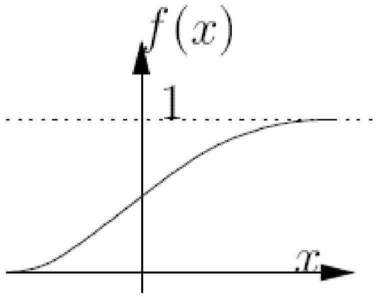
Activation function

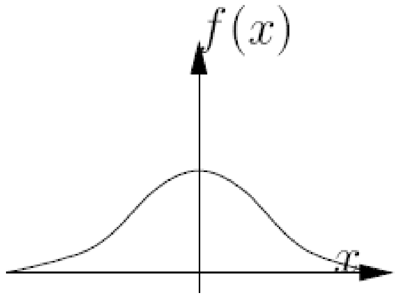
Logistic function $o_{pj} = \frac{1}{1+e^{-net_{pj}}}$ has nice features:

- The derivative is expressable in terms of the function itself:

$$\frac{\partial o_{pj}}{\partial net_{pj}} = o_{pj}(1 - o_{pj})$$

- The derivative is a “bump” that pushes uncommitted nodes to change weights.
- Likewise, weights are prevented from blowing up.
- The downside is that it is hard to change large weights!

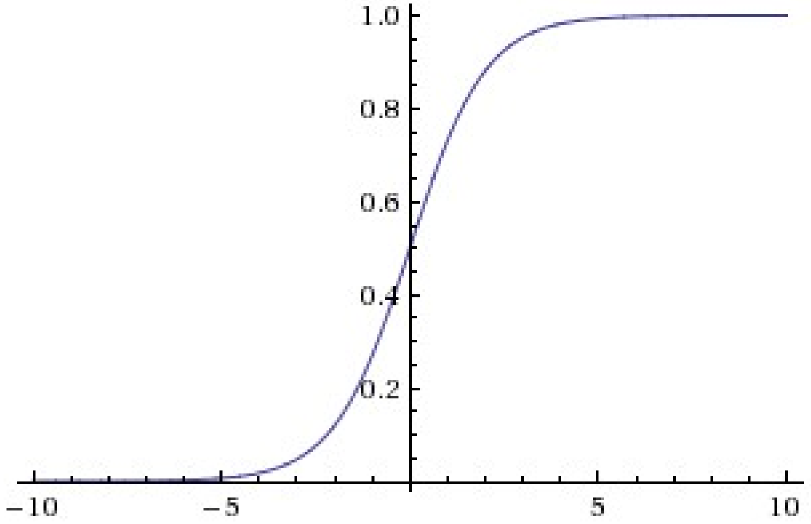


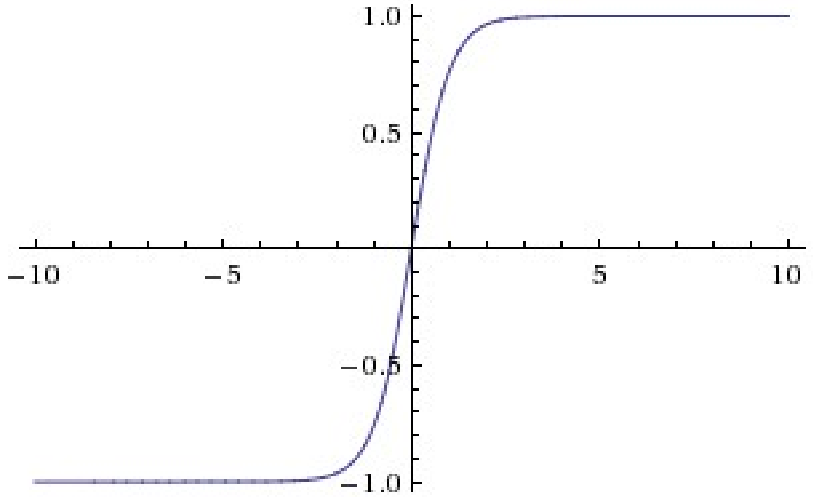


- How about $f(\cdot) = tanh(\cdot)$ which squashes the input into the $[-1 : +1]$?
- All we need to make sure it that $f(\cdot)$ is differentiable.

Activation Functions

- sigmoid vs tanh

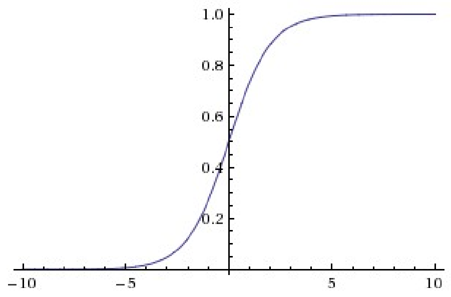




Derivative: $\sigma(x)(1-\sigma(x))$

Derivative: $(1-\tanh^2(x))$

Pros and Cons

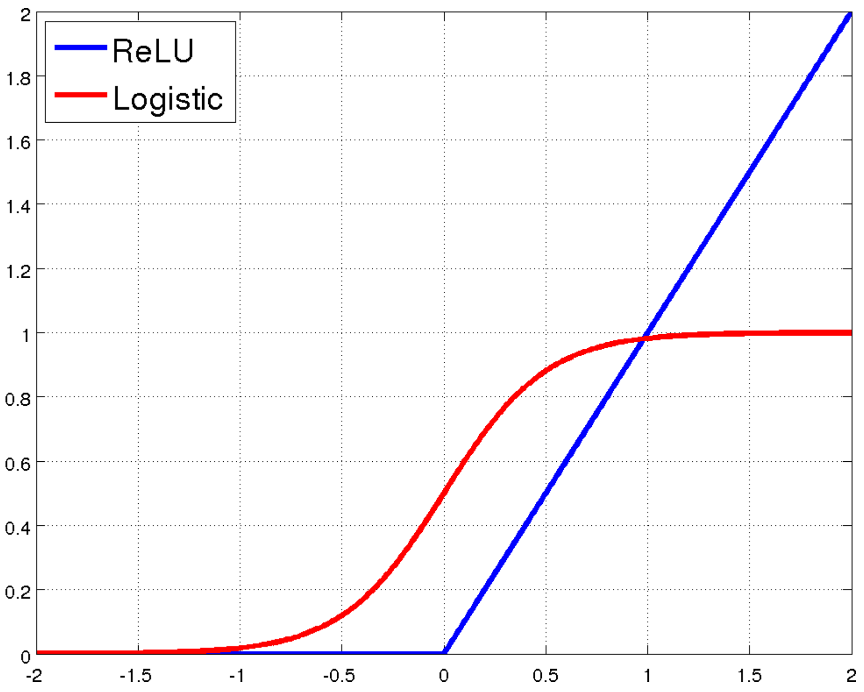


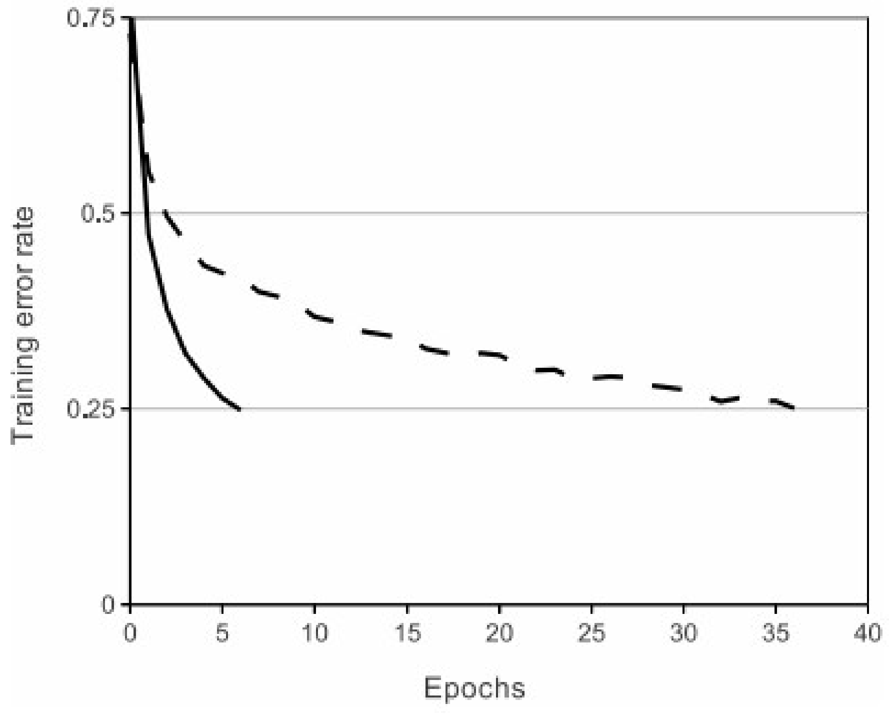
- Sigmoid is an historically important activation func
  - But nowadays, rarely used
- Sigmoid drawbacks
  1. It gets saturated, if the activation is close to zero or one
     - This leads to very small gradient, which disallows "transfer"ing the feedback to earlier layers
     - Initialization is also very important for this reason
  2. It is not zero-centered (not very severe)
- Tanh
  - Similar to the sigmoid, it saturates
  - However, it is zero-centered.
  - Tanh is always preferred over sigmod
  - Note: $\tanh(x) = 2\sigma(2x) - 1$

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

Rectified Linear Units (ReLU)

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





[Krizhevsky et al., NIPS12]

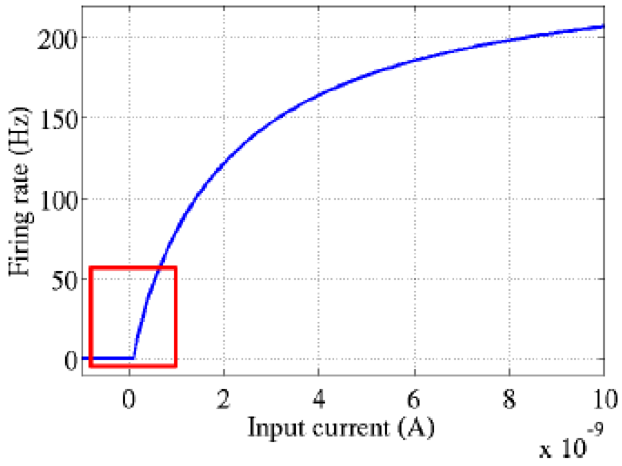
$$f(x) = \max(0, x)$$

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

ReLU: biological motivation

$$f(I) = \begin{cases} \left[\tau \log\left(\frac{E+RI-V_r}{E+RI-V_{th}}\right) + t_{ref}\right]^{-1}, \\ \qquad\qquad \text{if } E+RI > V_{th} \\ 0\,, \qquad\quad \text{if } E+RI \le V_{th} \end{cases}$$

where t_{ref} is the refractory period (minimal time between two action potentials), I the input current, V_r the resting potential and V_{th} the threshold potential (with $V_{th} > V_r$), and R, E, τ the membrane resistance, potential and time constant. The most commonly used activation func-



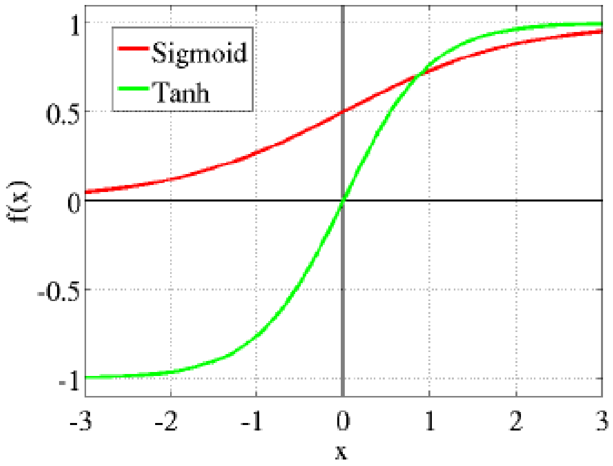
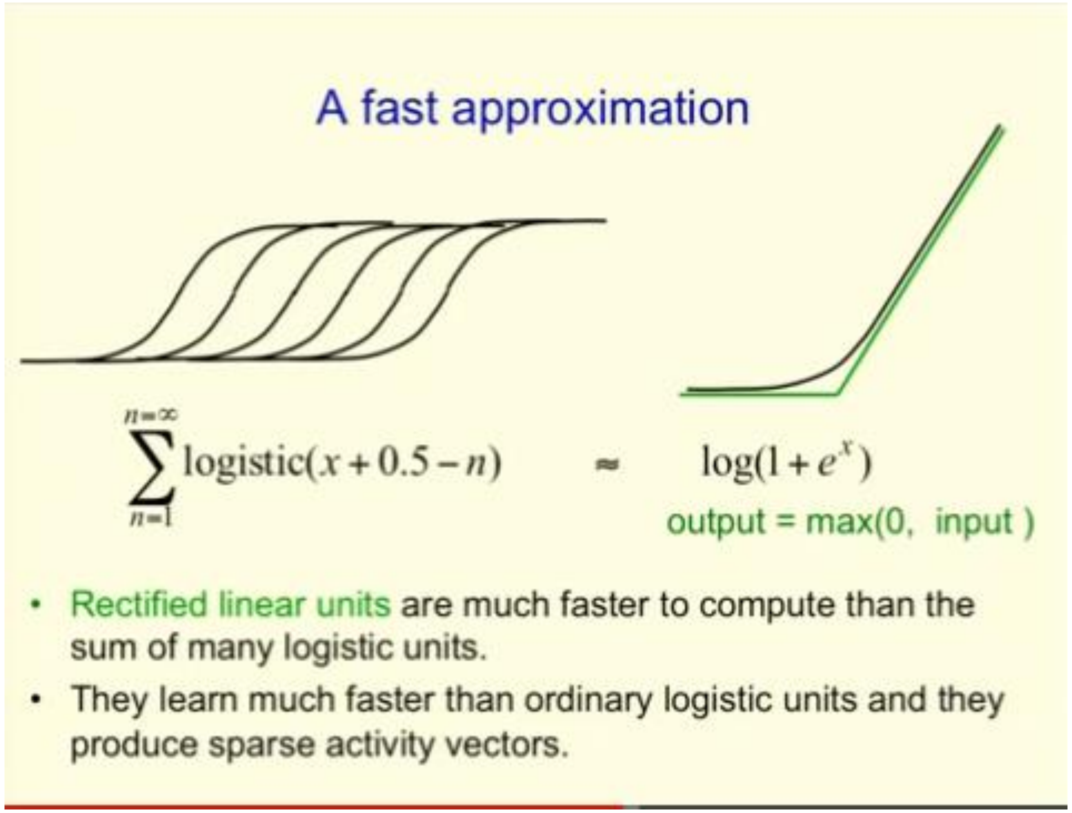


Figure 1: *Left:* Common neural activation function motivated by biological data. *Right:* Commonly used activation functions in neural networks literature: logistic sigmoid and hyperbolic tangent (*tanh*).

Glorot et al., “Deep Sparse Rectier Neural Networks”, 2011.

Rectified Linear Units: Another Perspective

Hinton argues that this is a form of model averaging



A fast approximation

$$\sum_{n=1}^{n=\infty} \text{logistic}(x + 0.5 - n) \quad \approx \quad \log(1 + e^{x})$$

output = max(0, input)

- Rectified linear units are much faster to compute than the sum of many logistic units.
- They learn much faster than ordinary logistic units and they produce sparse activity vectors.

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 (simple comparison)
- 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

ReLU

- See the following site for more in-depth analysis

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

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

Leaky ReLU

- $f(x) = \mathbf{1}(x < 0)(\alpha x) + \mathbf{1}(x \geq 0)(x)$
  - When x is negative, have 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

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!

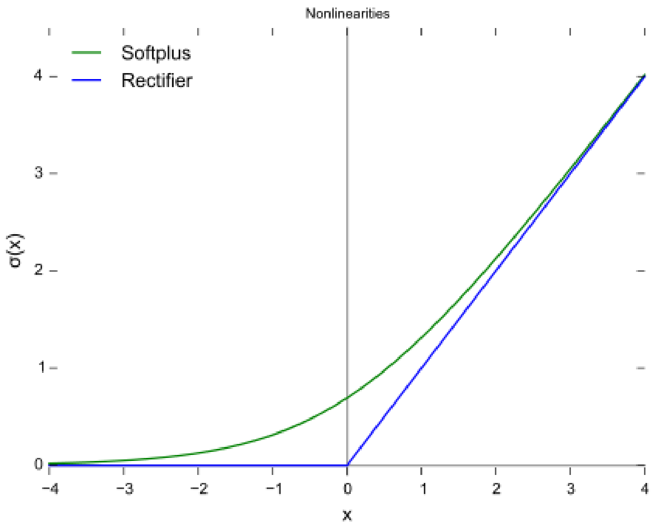
Softplus

- 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: To sum up

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

DEMO

- 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

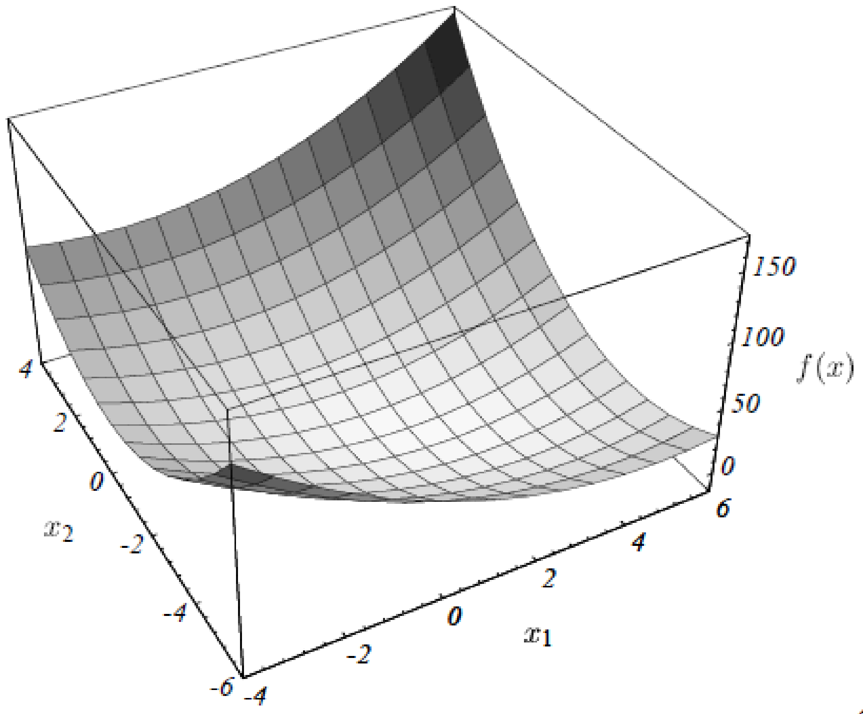
https://jalammar.github.io/visual-interactive-guide-basics-neural-networks/

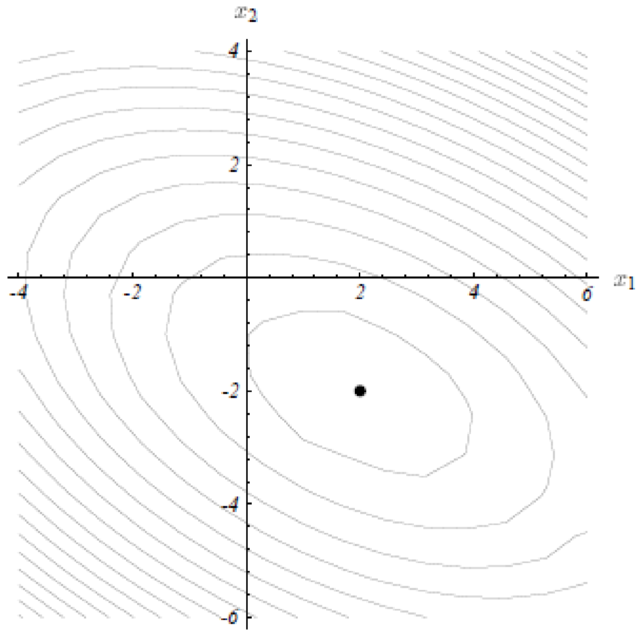
BACK PROPAGATION / MINIMIZATION STRATEGIES

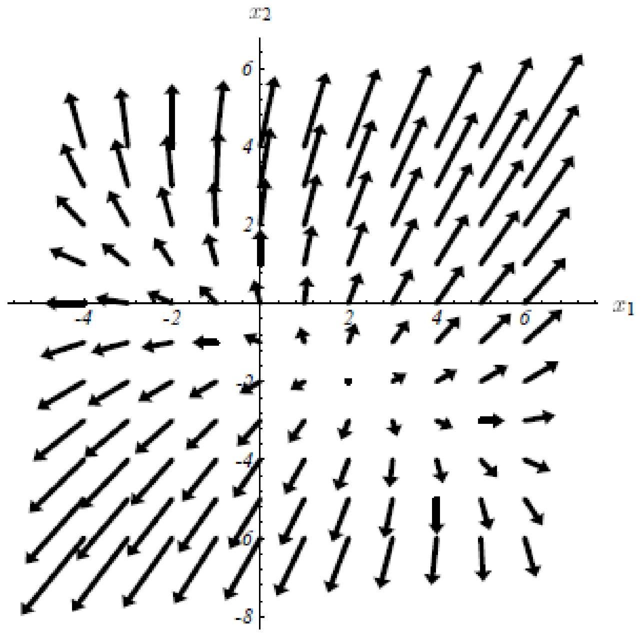
Schemes of training

- True/Standard Gradient Descent
- Stochastic Gradient Descent
- Steepest Gradient Descent
- Momentum Gradient Descent

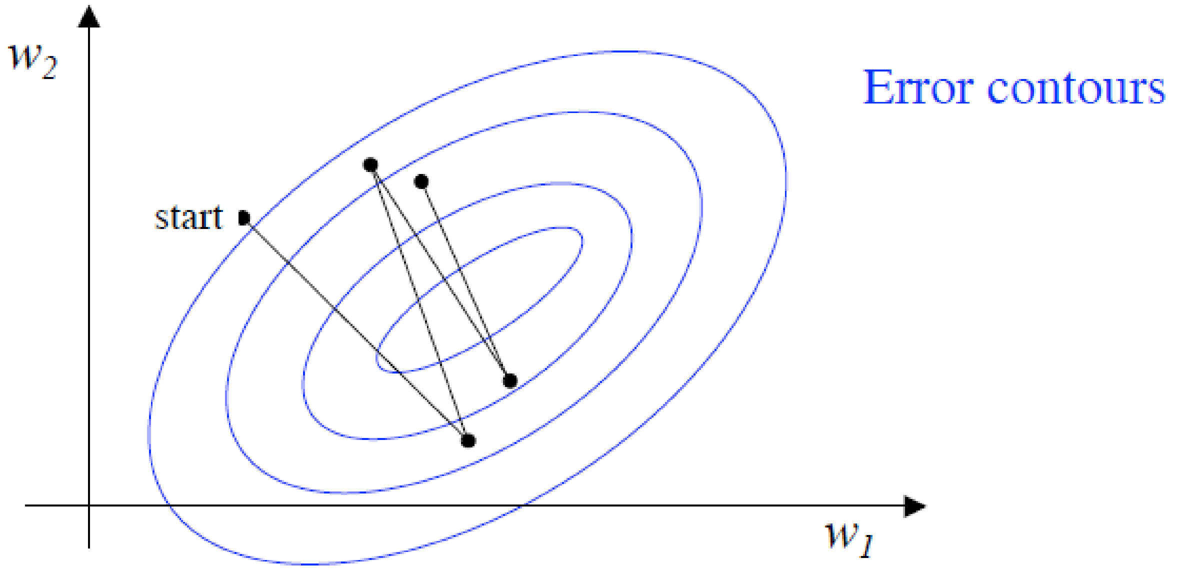
- Curricular training

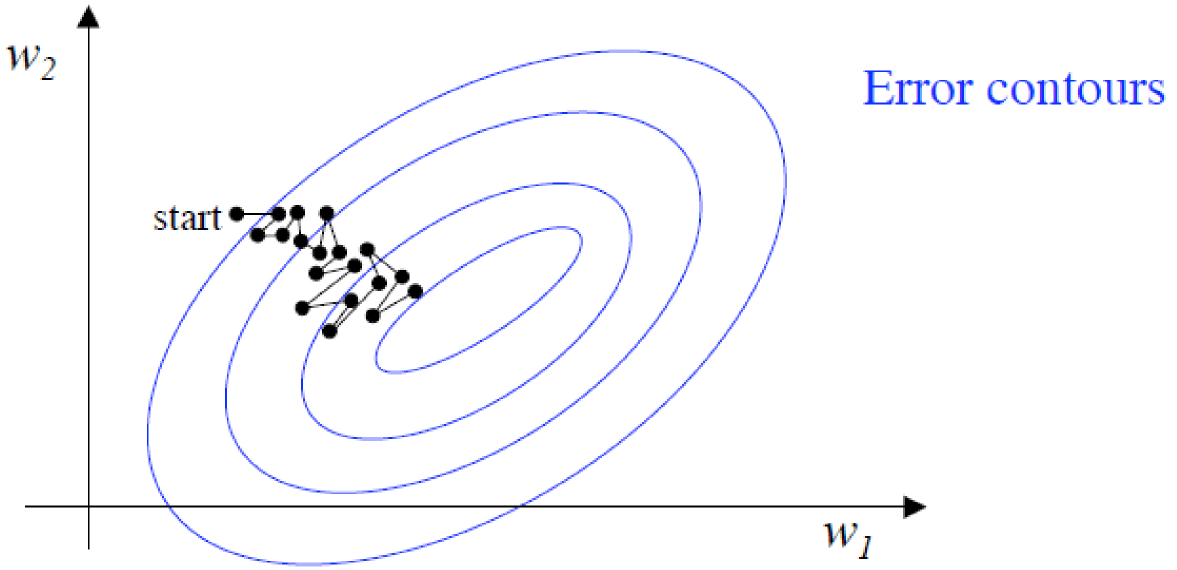




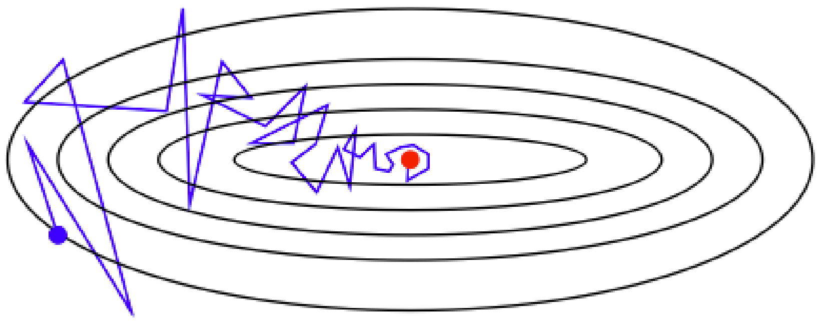


Jonathan Richard Shewchuk

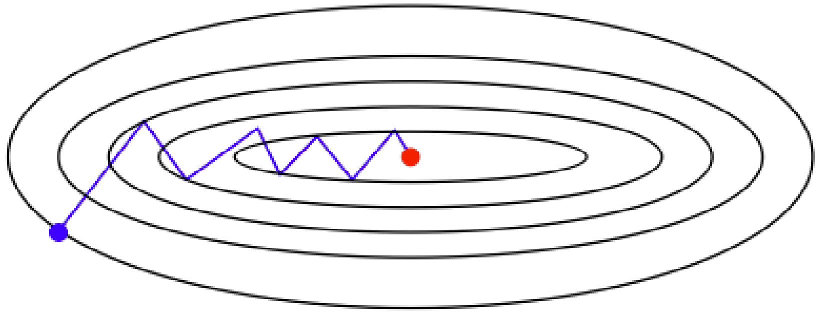

w_2
Error contours
start
w_1


w_2
Error contours
start
w_1

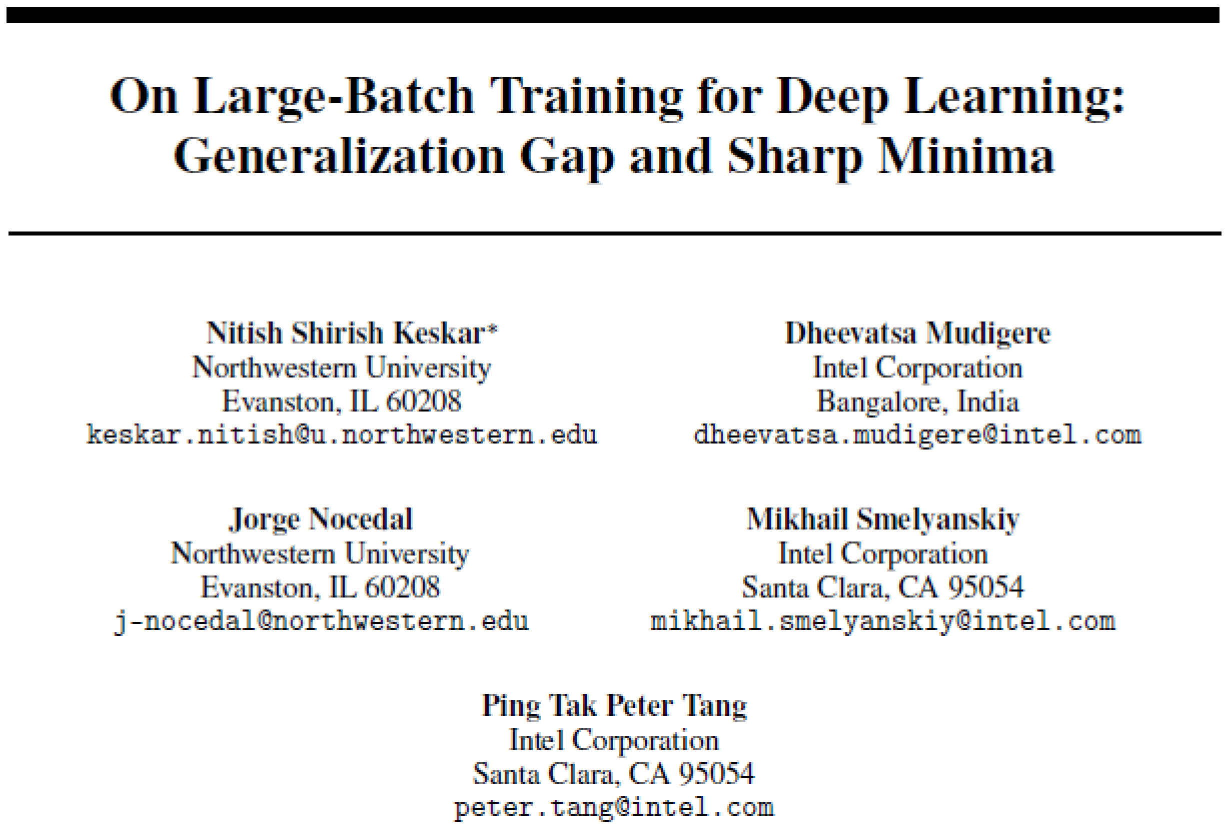
Stochastic Gradient Descent



Batch Gradient Descent



F. Bach



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

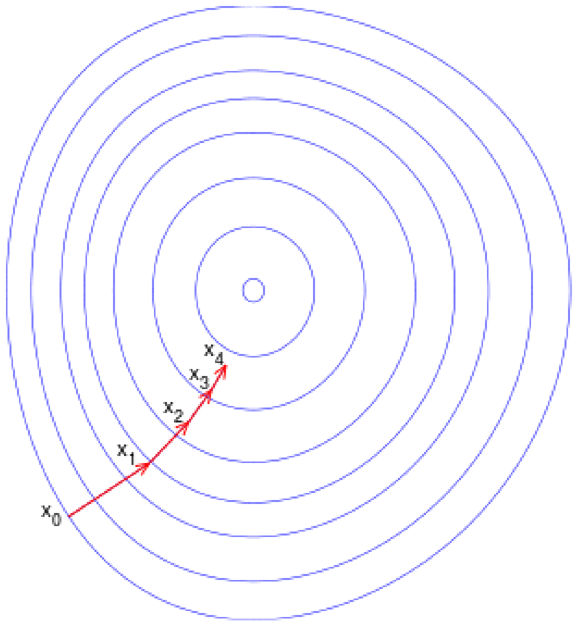
Dheevatsa Mudigere
Intel Corporation
Bangalore, India
dheevatsa.mudigere@intel.com

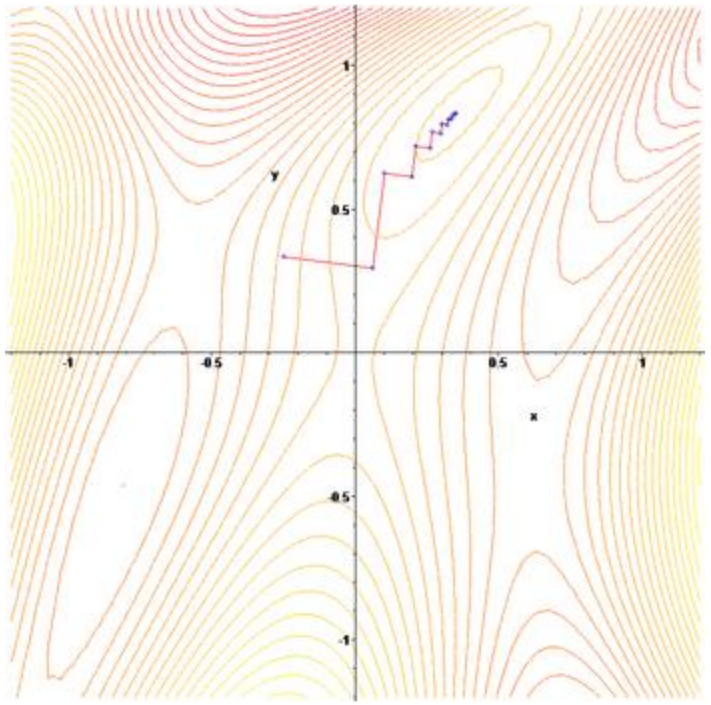
Jorge Nocedal
Northwestern University
Evanston, IL 60208
j-nocedal@northwestern.edu

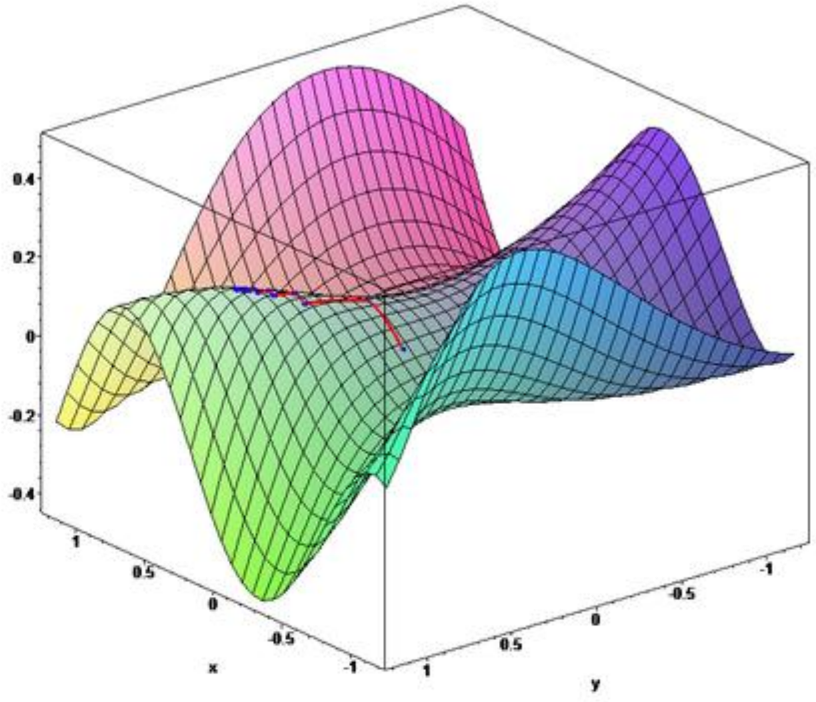
Mikhail Smelyanskiy
Intel Corporation
Santa Clara, CA 95054
mikhail.smelyanskiy@intel.com

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

Gradient descent







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

Second order methods

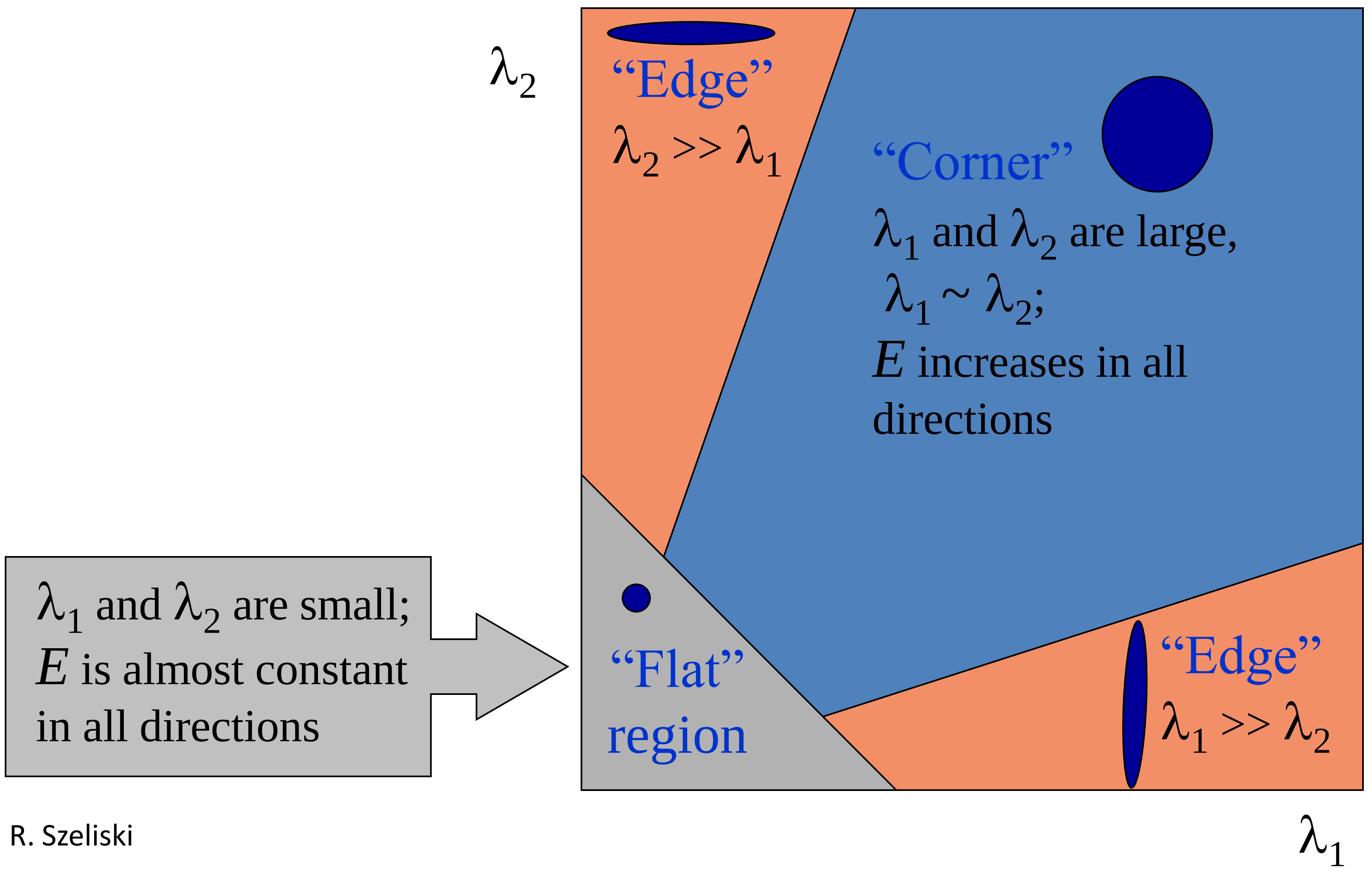
- Newton's method for optimization:
  - $w \leftarrow w - [Hf(w)]^{-1} \nabla f(w)$
  - where $Hf(w)$ is the Hessian

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

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

Hessian for an image window

The eigenvalues of the Hessian matrix:



λ_2
"Edge"
$\lambda_2 >> \lambda_1$
"Corner"
λ_1 and λ_2 are large,
$\lambda_1 \sim \lambda_2$;
E increases in all directions
λ_1 and λ_2 are small;
E is almost constant in all directions
"Flat" region
"Edge"
$\lambda_1 >> \lambda_2$
λ_1

Source: R. Szeliski

Newton's method for optimization

- $w \leftarrow w - [Hf(w)]^{-1} \nabla f(w)$
  - Makes bigger steps in shallow curvature
  - Smaller steps in steep curvature
- Note that there is no hyper-parameter!
- 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)
    - Active research area ➔ A suitable project topic ☺

RPROP (Resilience Propagation)

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

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

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

- Motivation: If the sign of a weight 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

RPROP (Resilience Propagation)

```
For all weights and biases{
    if ($\frac{\partial E}{\partial w_{ij}}(t-1) * \frac{\partial E}{\partial w_{ij}}(t) > 0$) then {
        $\triangle_{ij}(t) = \textbf{minimum}\ (\triangle_{ij}(t-1) * \eta^{+}, \triangle_{max})$
        $\triangle w_{ij}(t) = -\ \textbf{sign}\ (\frac{\partial E}{\partial w_{ij}}(t)) * \triangle_{ij}(t)$
        $w_{ij}(t+1) = w_{ij}(t) + \triangle w_{ij}(t)$
    }
    else if ($\frac{\partial E}{\partial w_{ij}}(t-1) * \frac{\partial E}{\partial w_{ij}}(t) < 0$) then {
        $\triangle_{ij}(t) = \textbf{maximum}\ (\triangle_{ij}(t-1) * \eta^{-}, \triangle_{min})$
        $w_{ij}(t+1) = w_{ij}(t) - \triangle w_{ij}(t-1)$
        $\frac{\partial E}{\partial w_{ij}}(t) = 0$
    }
    else if ($\frac{\partial E}{\partial w_{ij}}(t-1) * \frac{\partial E}{\partial w_{ij}}(t) = 0$) then {
        $\triangle w_{ij}(t) = -\ \textbf{sign}\ (\frac{\partial E}{\partial w_{ij}}(t)) * \triangle_{ij}(t)$
        $w_{ij}(t+1) = w_{ij}(t) + \triangle w_{ij}(t)$
    }
}
```

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

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

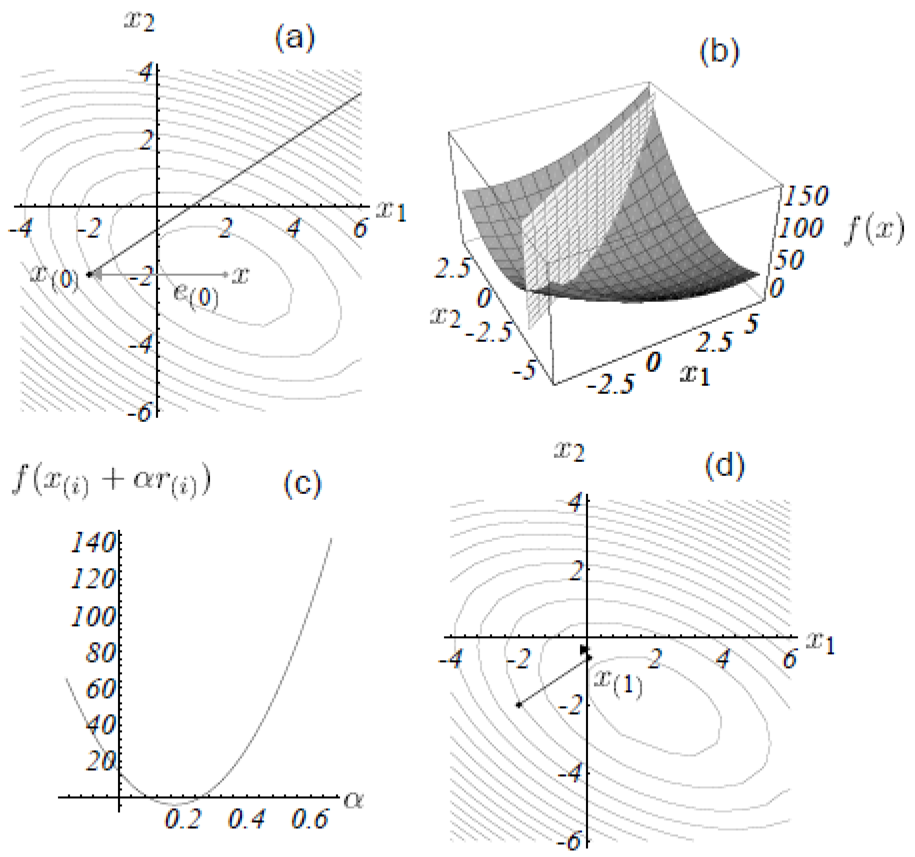


Figure 6: The method of Steepest Descent. (a) Starting at $[-2, -2]^T$, take a step in the direction of steepest descent of f. (b) Find the point on the intersection of these two surfaces that minimizes f. (c) This parabola is the intersection of surfaces. The bottommost point is our target. (d) The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

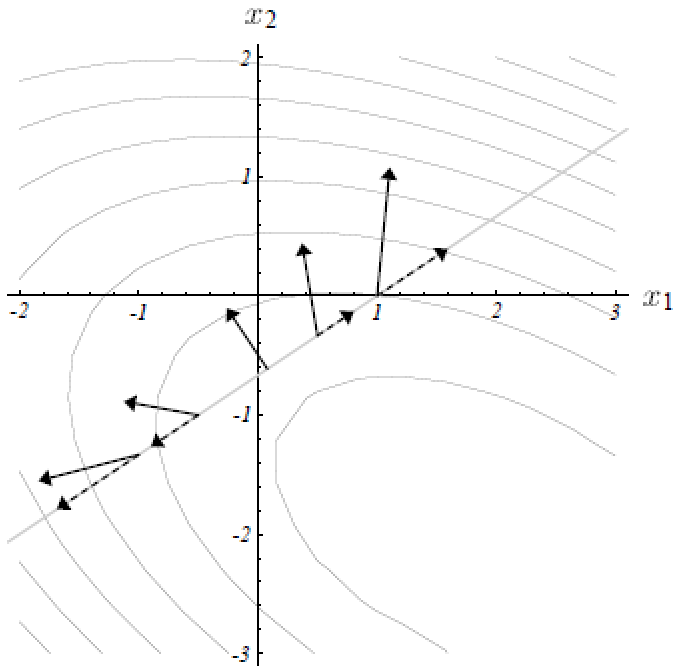


Figure 7: The gradient f' is shown at several locations along the search line (solid arrows). Each gradient's projection onto the line is also shown (dotted arrows). The gradient vectors represent the direction of steepest increase of f, and the projections represent the rate of increase as one traverses the search line. On the search line, f is minimized where the gradient is orthogonal to the search line.

Gradient Descent with Line Search

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

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

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

- 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

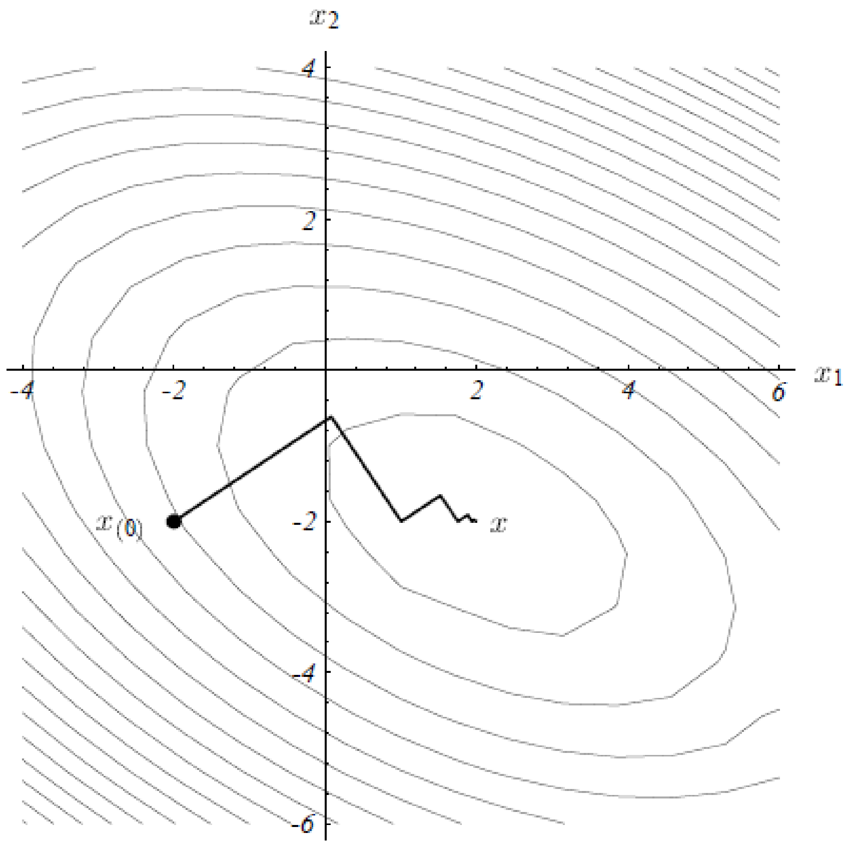
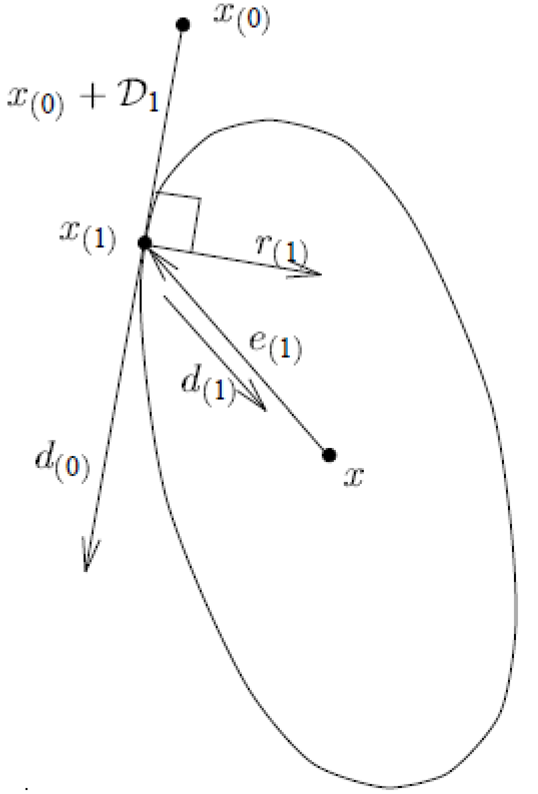


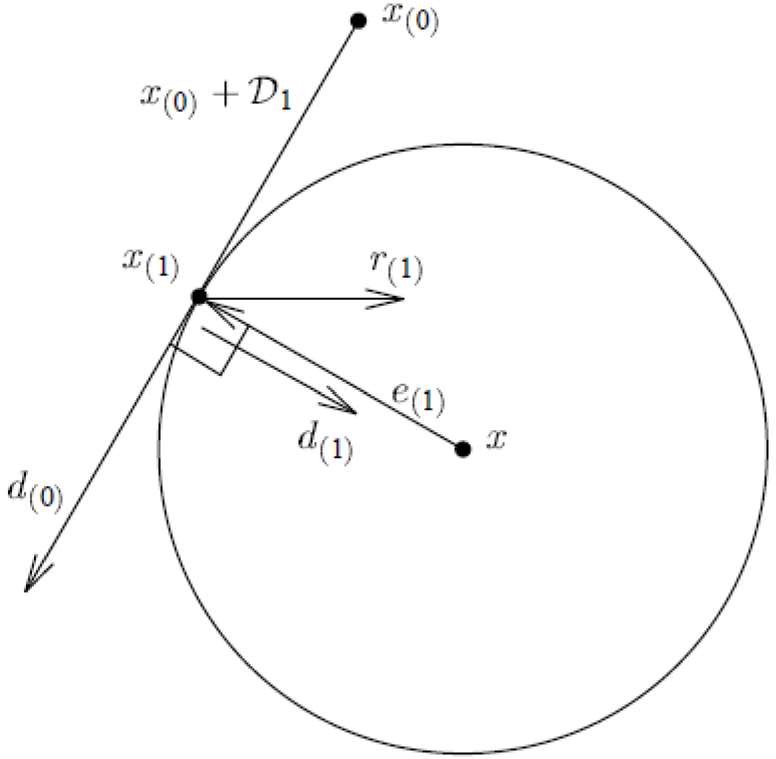
Figure 8: Here, the method of Steepest Descent starts at $[-2, -2]^T$ and converges at $[2, -2]^T$.

Conjugate Gradient Descent

- Motivation



(a)



(b)

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:

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

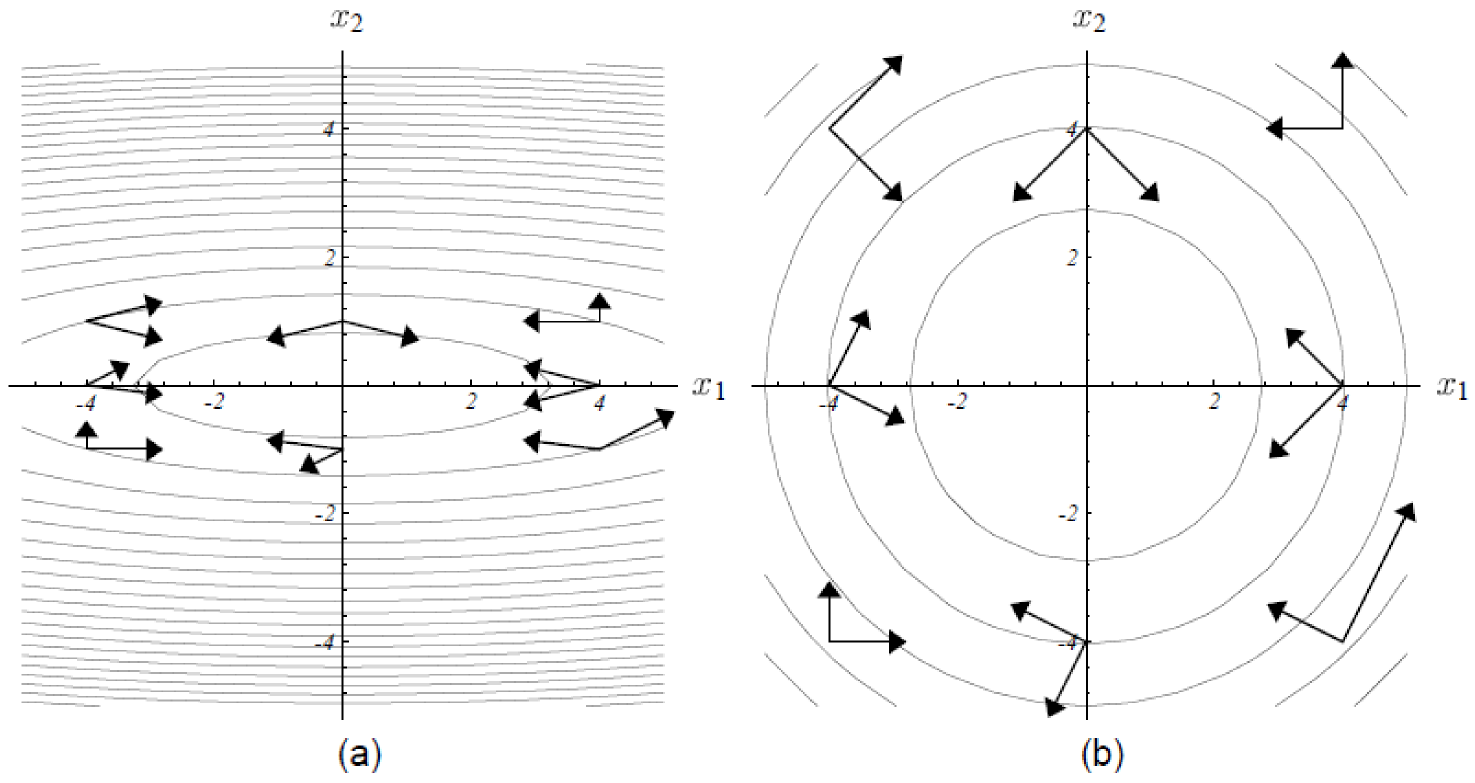


Figure 22: These pairs of vectors are A-orthogonal . . . because these pairs of vectors are orthogonal.

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:

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

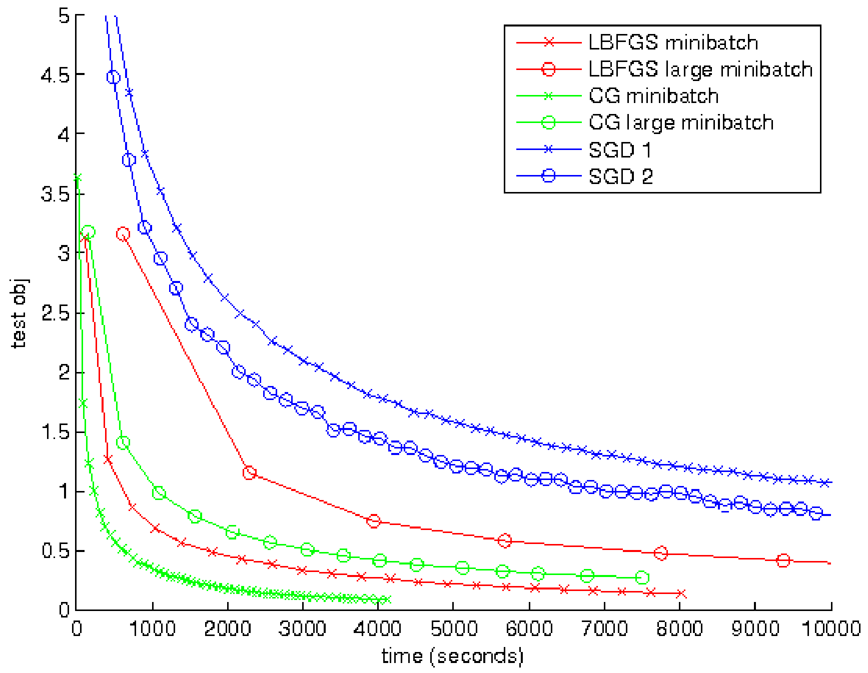
  - If the new direction suggests a radical turn, rely more on the old direction!

- For more detailed motivation and derivations, see:

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

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.

Online Interactive Tutorial

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

Genetic Algorithms

General strategy:

- Randomly choose weights and encode them on a string of bits (chromosomes).
- Determine a “fitness” function (e.g. error function).
- Use genetic operators *mutation, crossover) to construct new strings.
- Use “survival of the fittest” to produce better and better strings.

Some observations/comments:

- Selection of fitness and operators is crucial to its effectiveness.
- Search is global, not fooled by local minima.
- Fitness (or error in this case) function need not be differentiable.
- Search is rather blind, since it does not use the ∇ info.
- It can be a good method for initialization, to be used for a gradient method.

Slide credit: E. Sahin

CHALLENGES OF THE ERROR SURFACE

Challenges

- Local minima
- Saddle points
- Cliffs
- Valleys

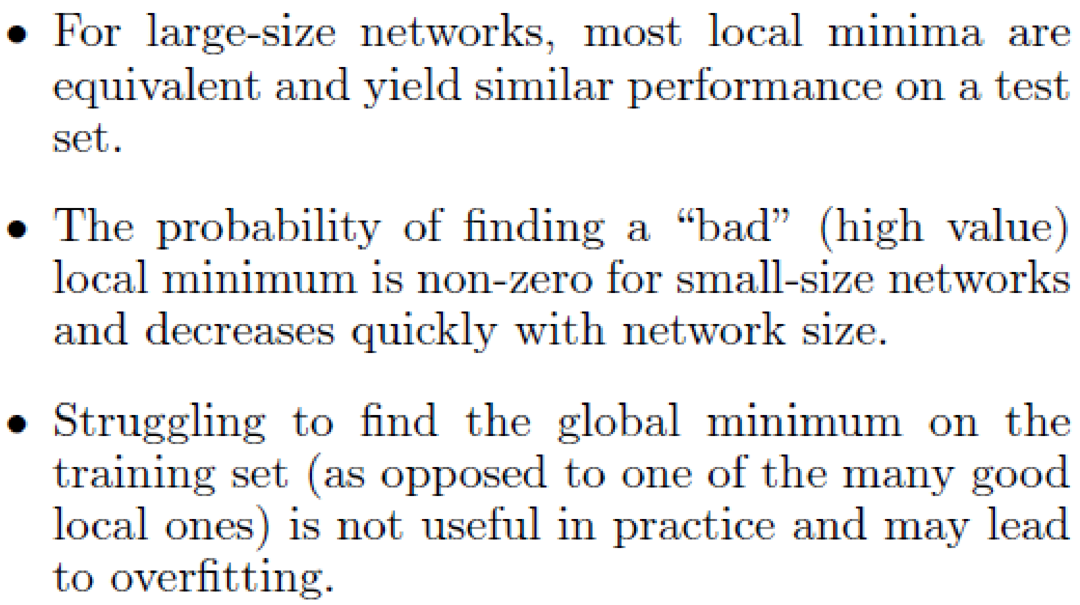
Local minima

- Solutions
  - Momentum
    - Make weight update depend on the previous one as well:

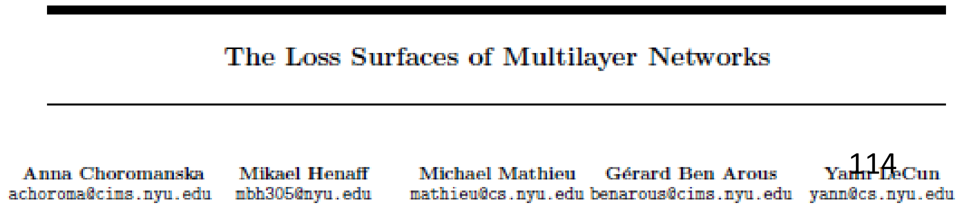
$$\Delta w_{ij}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1)$$

    - $0 \leq \alpha < 1$: momentum (constant)
  - Incremental update
  - Large training data
  - 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.



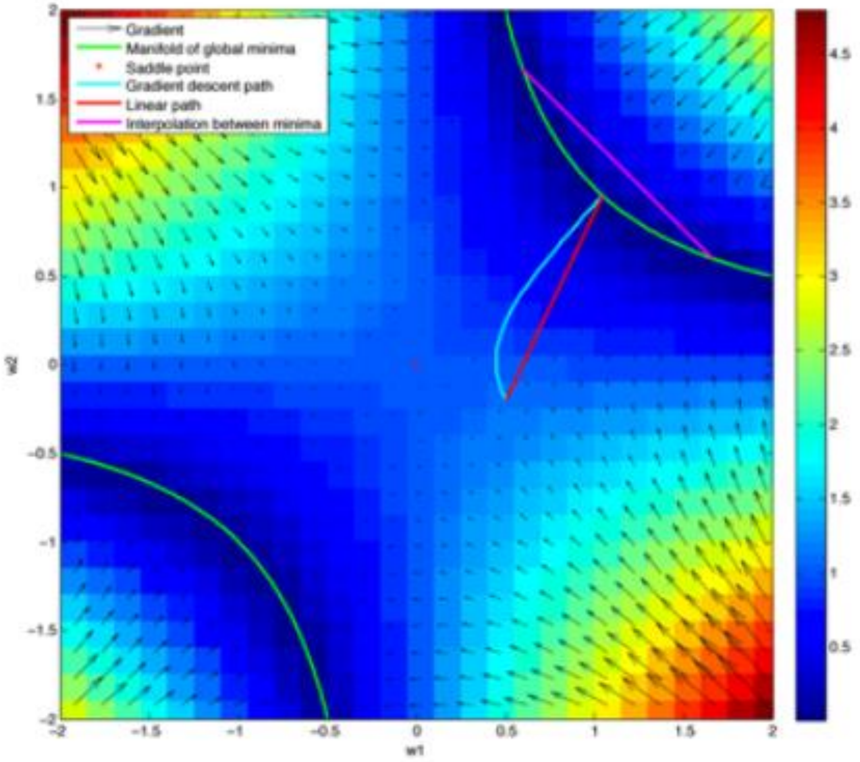
The Loss Surfaces of Multilayer Networks

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

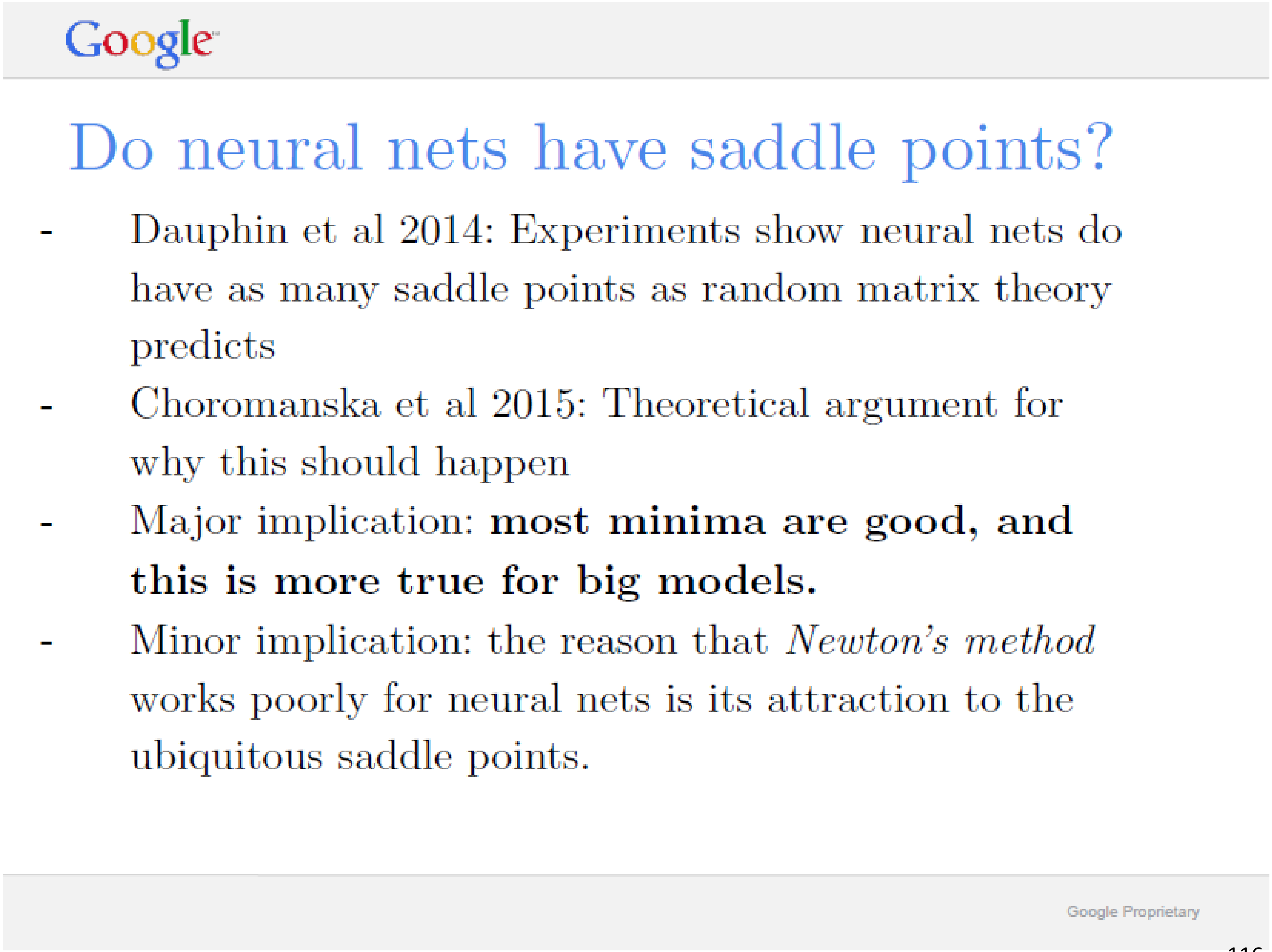
Google

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



I. Goodfellow



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.

Valleys, Cliffs and Exploding Gradients

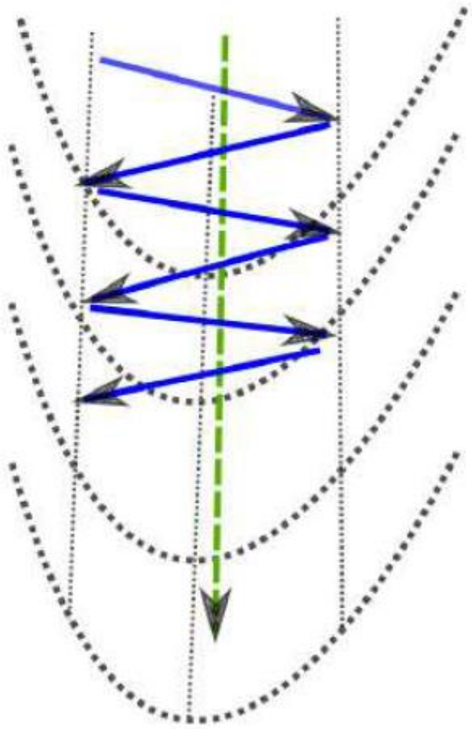


Figure 8.1: One theory about the neural network optimization is that poorly conditioned Hessian matrices cause much of the difficulty in training. In this view, some directions have a high curvature (second derivative), corresponding to the quickly rising sides of the valley (going left or right), and 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

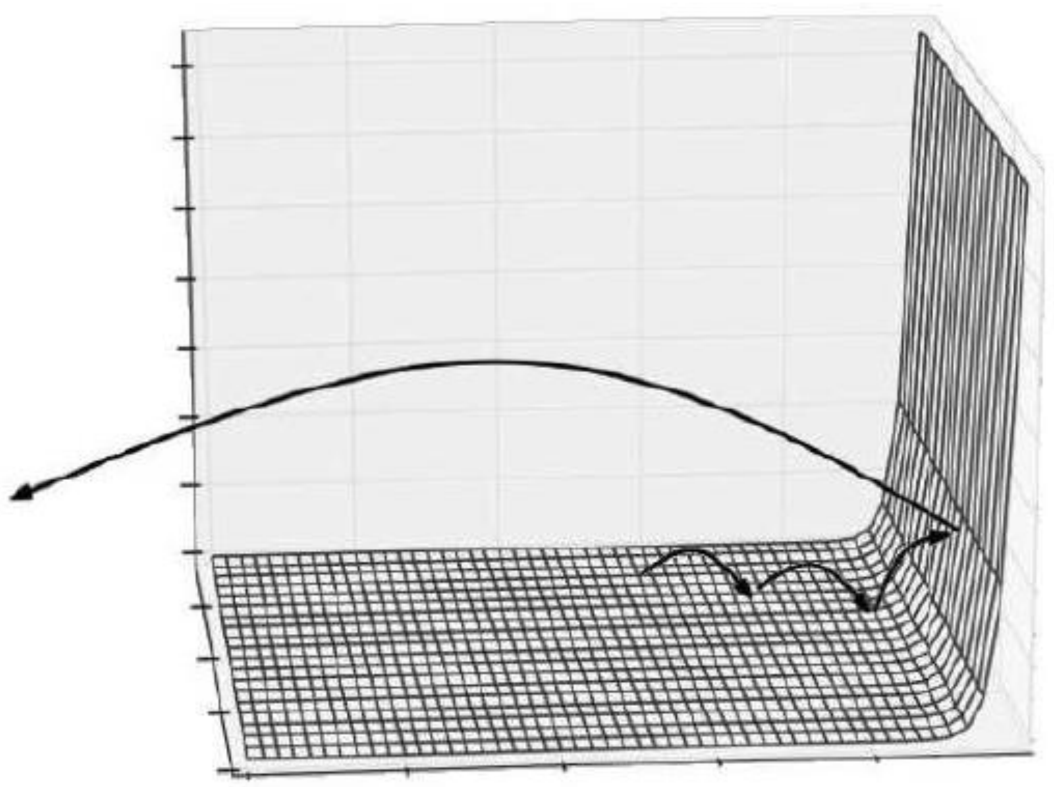


Figure 8.2: Contrary to what is shown in Figure 8.1, the objective function for highly non-linear deep neural networks or for recurrent neural networks is typically not made of symmetrical sides. As shown in the figure, there are sharp non-linearities that give rise to very high derivatives in some places. 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

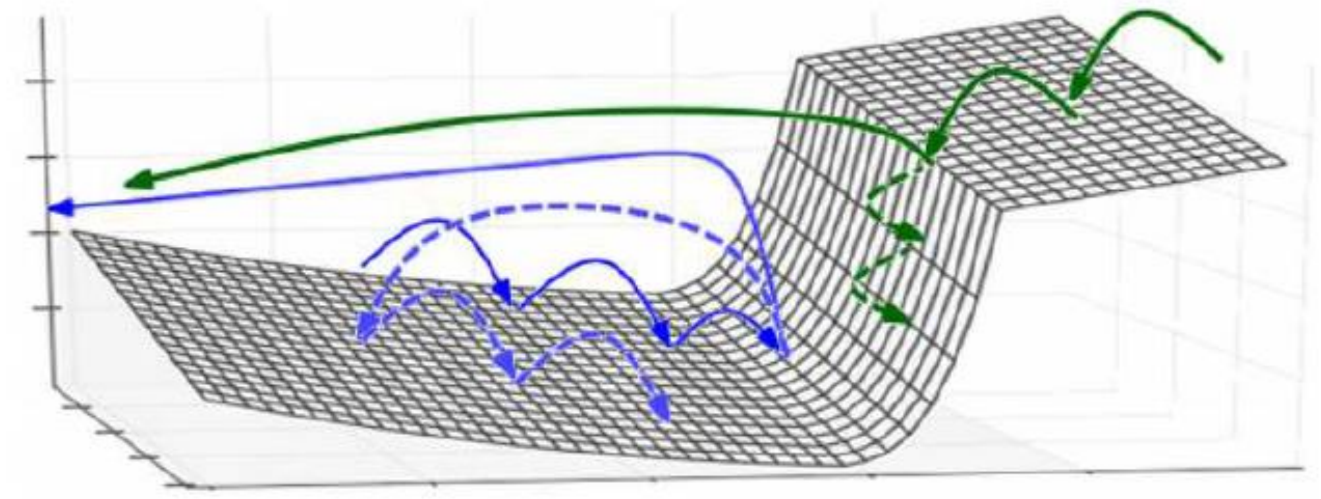


Figure 8.3: To address the presence of cliffs such as shown in Figure 8.2, a useful heuristic is to clip the magnitude of the gradient, only keeping its direction if its magnitude is above a threshold (which is a hyperparameter, although not a very critical one). Using such a gradient clipping heuristic (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

- Maintain a "memory"
$$\Delta w(t+1) \leftarrow \mu\, \Delta w(t) - \eta\, \nabla E$$
where μ is called the momentum term
- 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

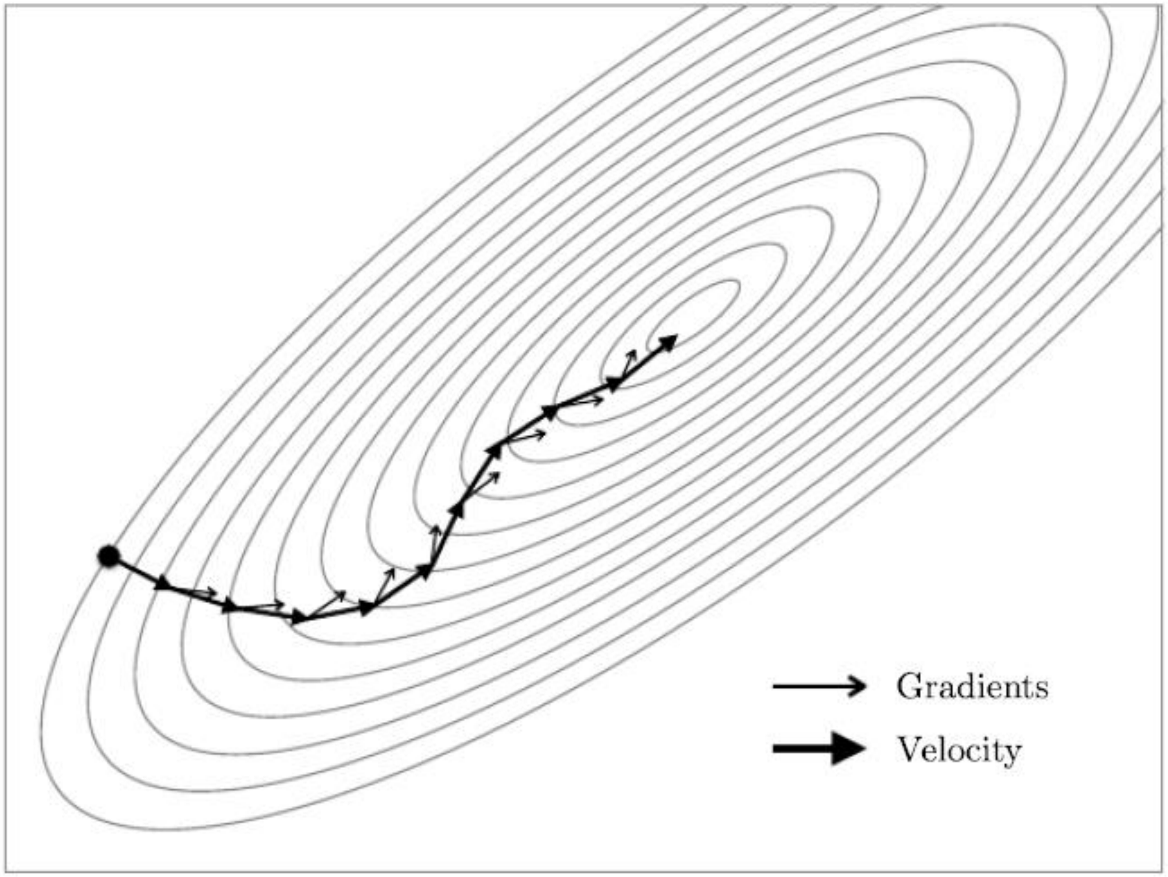
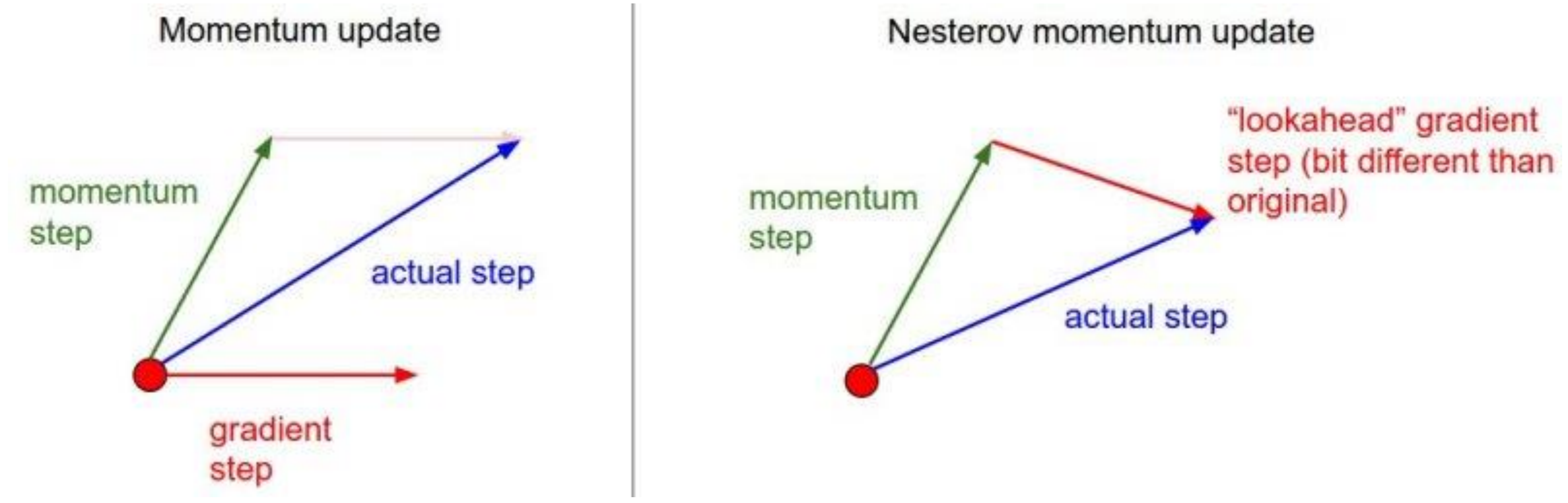


Figure 8.5: The effect of momentum on the progress of learning. Momentum acts to accumulate gradient contributions over training iterations. Directions that consistently have positive contributions to the gradient will be augmented.

Nesterov Momentum

- Use a "lookahead" step to update:

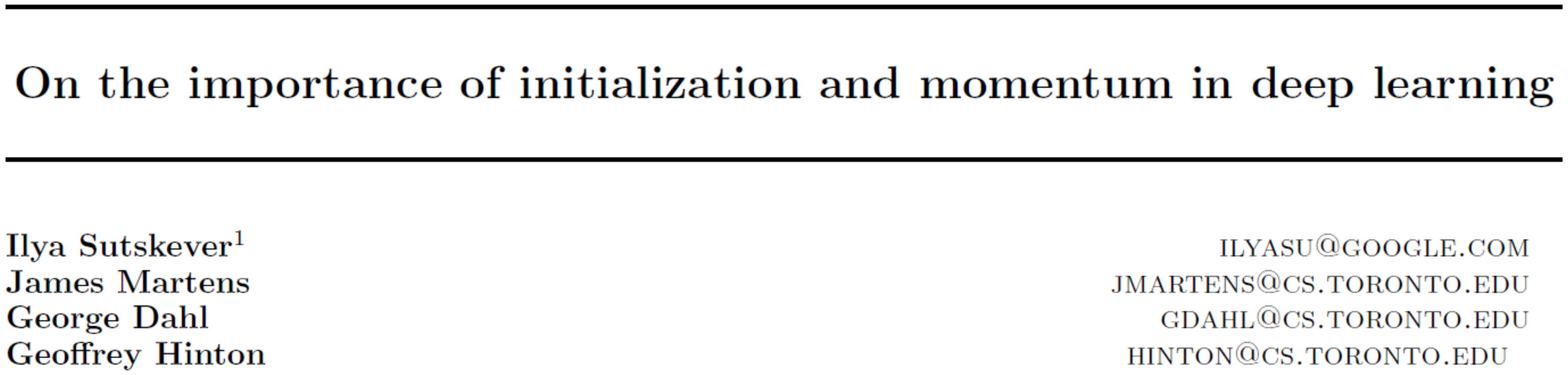
$$w_{\text{ahead}} \leftarrow w + \mu\, \Delta w(t)$$
$$\Delta w(t+1) \leftarrow \mu\, \Delta w(t) - \eta\, \nabla E_{\text{ahead}}$$
$$w \leftarrow w + \Delta w(t+1)$$



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

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

SETTING THE LEARNING RATE

Alternatives

- Single global learning rate
  - Adaptive Learning Rate
  - Adaptive Learning Rate with Momentum

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

Adaptive Learning Rate (Global)

Choice of η and α is not always easy. Also different problems require different choices. Alternative: change η and/or α during training.
Typical rule: after each Δw, check the change in error:

$$\Delta E = E(t) - E(t-1)$$

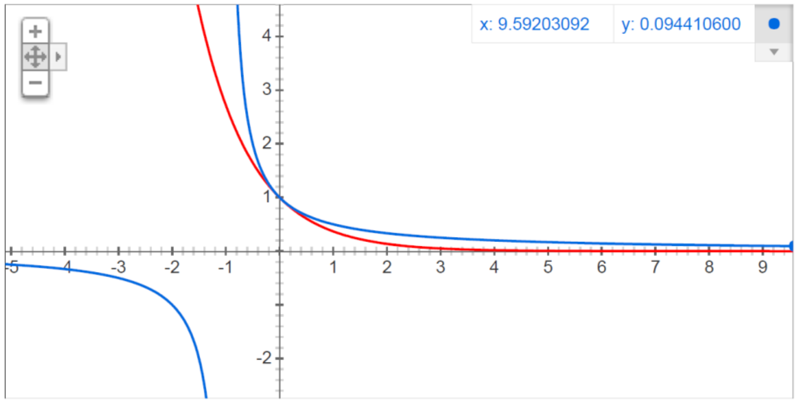
- If ΔE decreases consistently, increse η.
- If ΔE increases, rapidly decrease η.

Similar methods can be used for the adaptation of the momentum parameter, α.

Annealing the learning rate (Global)

- 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

Graph for 1/(1+x), e^-x



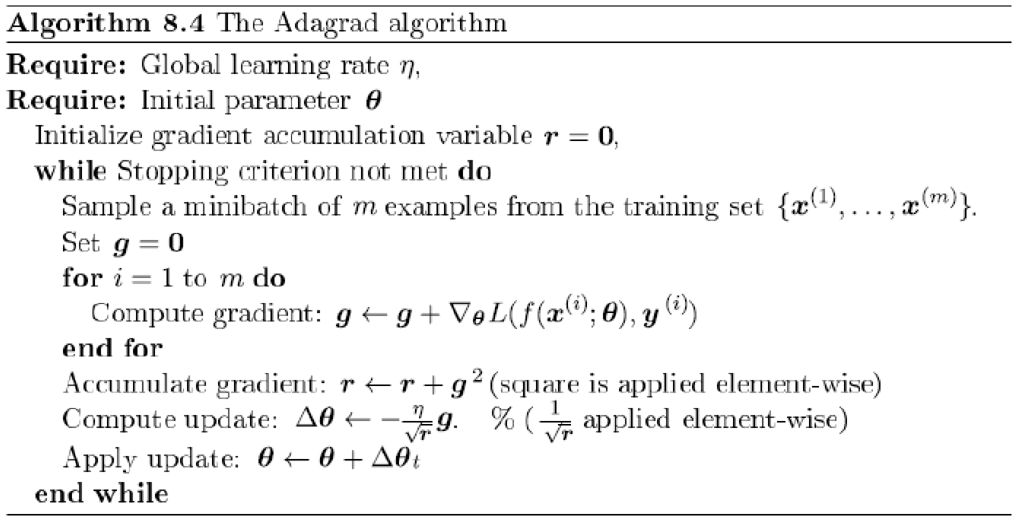
Adagrad (Per parameter)

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

$$r \leftarrow r + \left(\sum_{i=1:M} \frac{\partial L(x_i; W, b)}{\partial W} \right)^2$$

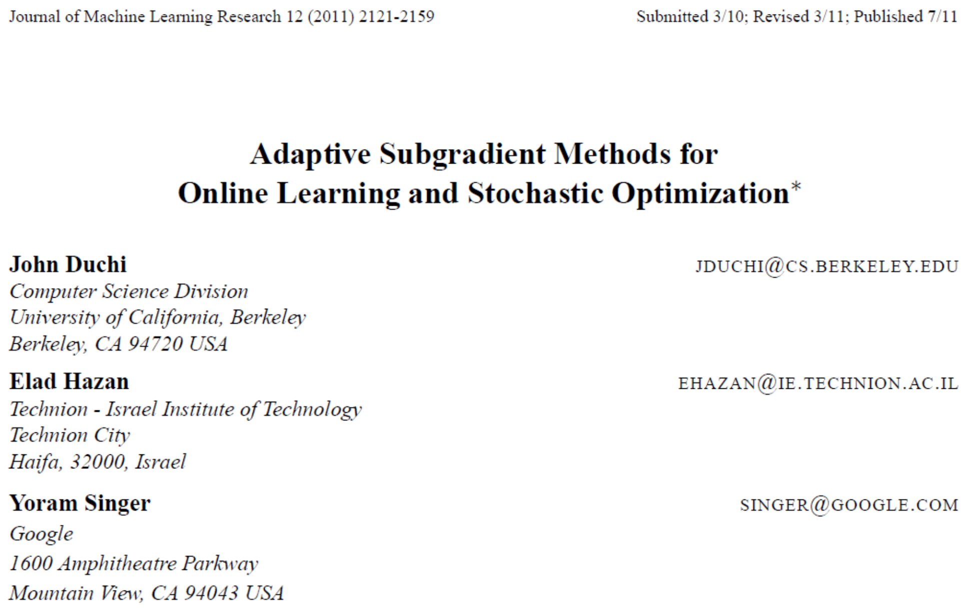
- Update each parameter/weight based on the gradient on that:

$$\Delta W \leftarrow -\frac{\eta}{\sqrt{r}} \sum_{i=1:M} \frac{\partial L(x_i; W, b)}{\partial W}$$



Algorithm 8.4 The Adagrad algorithm

```
Require: Global learning rate $\eta$,
Require: Initial parameter $\boldsymbol{\theta}$
  Initialize gradient accumulation variable $\boldsymbol{r} = \boldsymbol{0}$,
  while Stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
    Set $\boldsymbol{g} = \boldsymbol{0}$
    for $i = 1$ to $m$ do
      Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    end for
    Accumulate gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g}^2$ (square is applied element-wise)
    Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\eta}{\sqrt{\boldsymbol{r}}}\boldsymbol{g}$.   % ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}_t$
  end while
```



Journal of Machine Learning Research 12 (2011) 2121-2159 Submitted 3/10; Revised 3/11; Published 7/11

Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*

John Duchi JDUCHI@CS.BERKELEY.EDU
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720 USA

Elad Hazan EHAZAN@IE.TECHNION.AC.IL
Technion - Israel Institute of Technology
Technion City
Haifa, 32000, Israel

Yoram Singer SINGER@GOOGLE.COM
Google
1600 Amphitheatre Parkway
Mountain View, CA 94043 USA

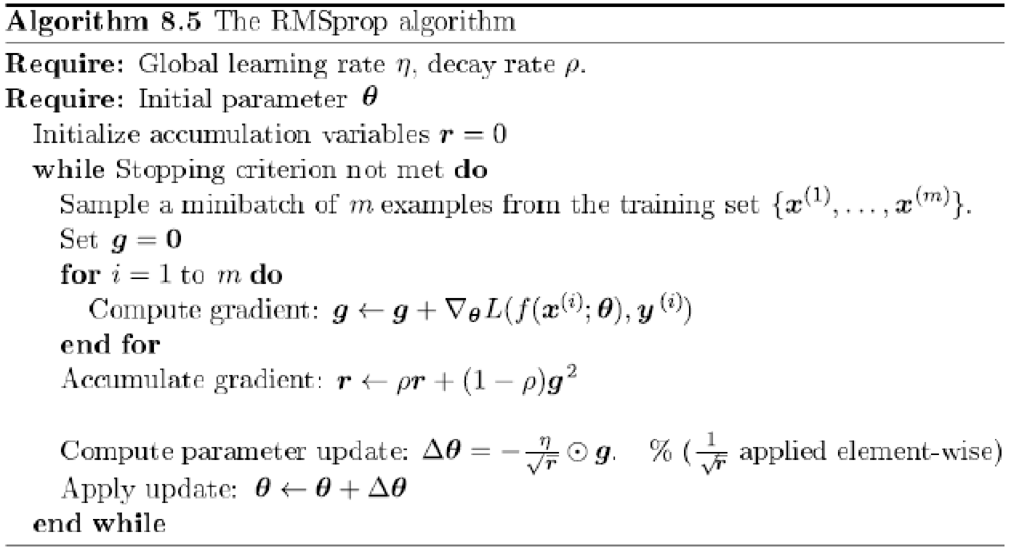
RMSprop (Per parameter)

- Similar to Adagrad
- Calculates a moving average of square of the gradients
- Accumulate square of gradients (initially $r = 0$):

$$r \leftarrow \rho r + (1-\rho)\left(\sum_{i=1:M} \partial L(x_i; W, b)/\partial W\right)^2$$

- ρ is typically [0.9, 0.99, 0.999]
- Update each parameter/weight based on the gradient on that:

$$\Delta W \leftarrow -\frac{\eta}{\sqrt{r}} \sum_{i=1:M} \frac{\partial L(x_i; W, b)}{\partial W}$$



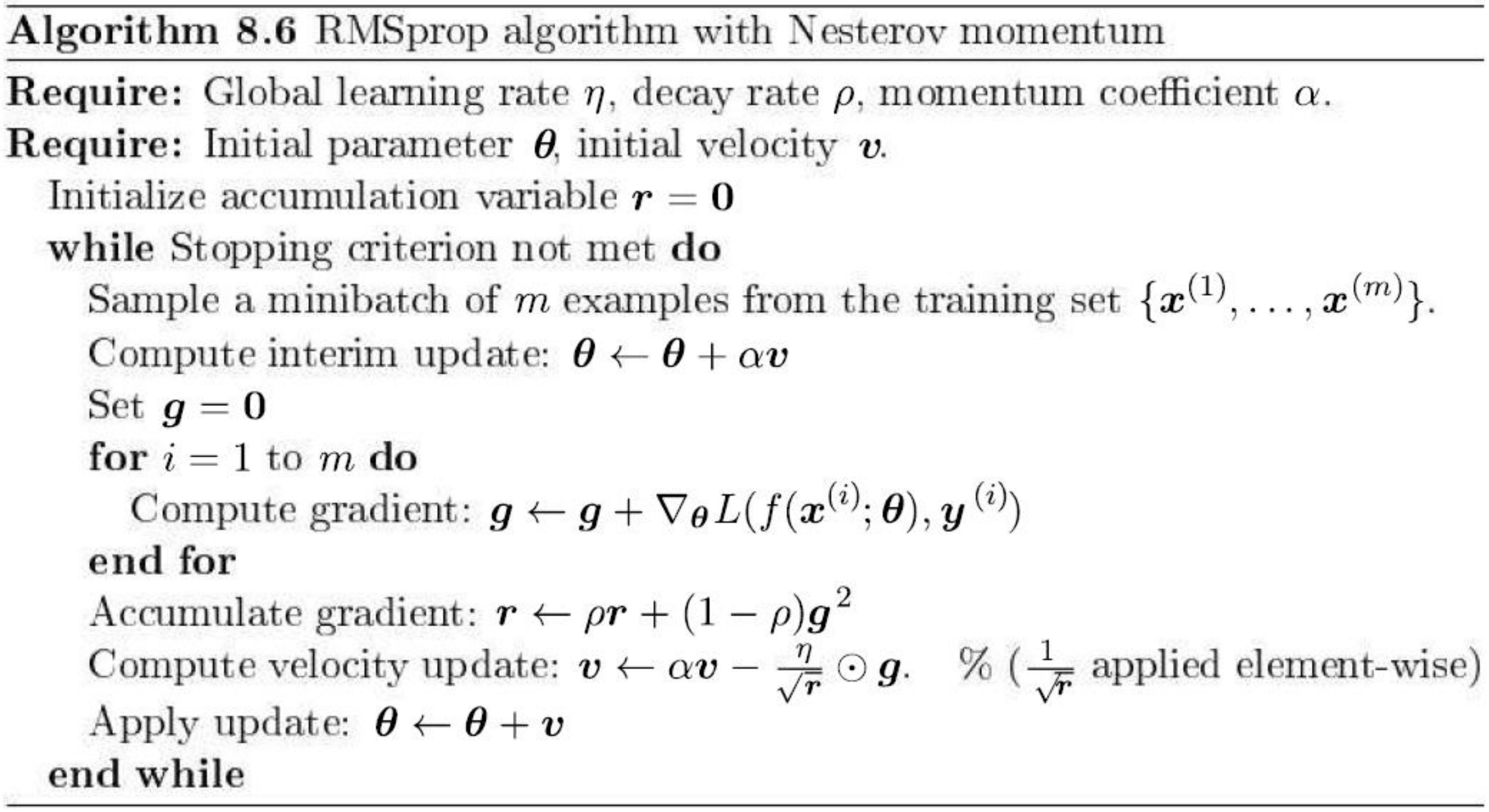
Algorithm 8.5 The RMSprop algorithm

```
Require: Global learning rate $\eta$, decay rate $\rho$.
Require: Initial parameter $\boldsymbol{\theta}$
  Initialize accumulation variables $\boldsymbol{r} = 0$
  while Stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$.
    Set $\boldsymbol{g} = \boldsymbol{0}$
    for $i = 1$ to $m$ do
      Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    end for
    Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)\boldsymbol{g}^2$

    Compute parameter update: $\Delta\boldsymbol{\theta} = -\frac{\eta}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$.   % ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

Currently, unpublished.

RMSprop with Nesterov Momentum

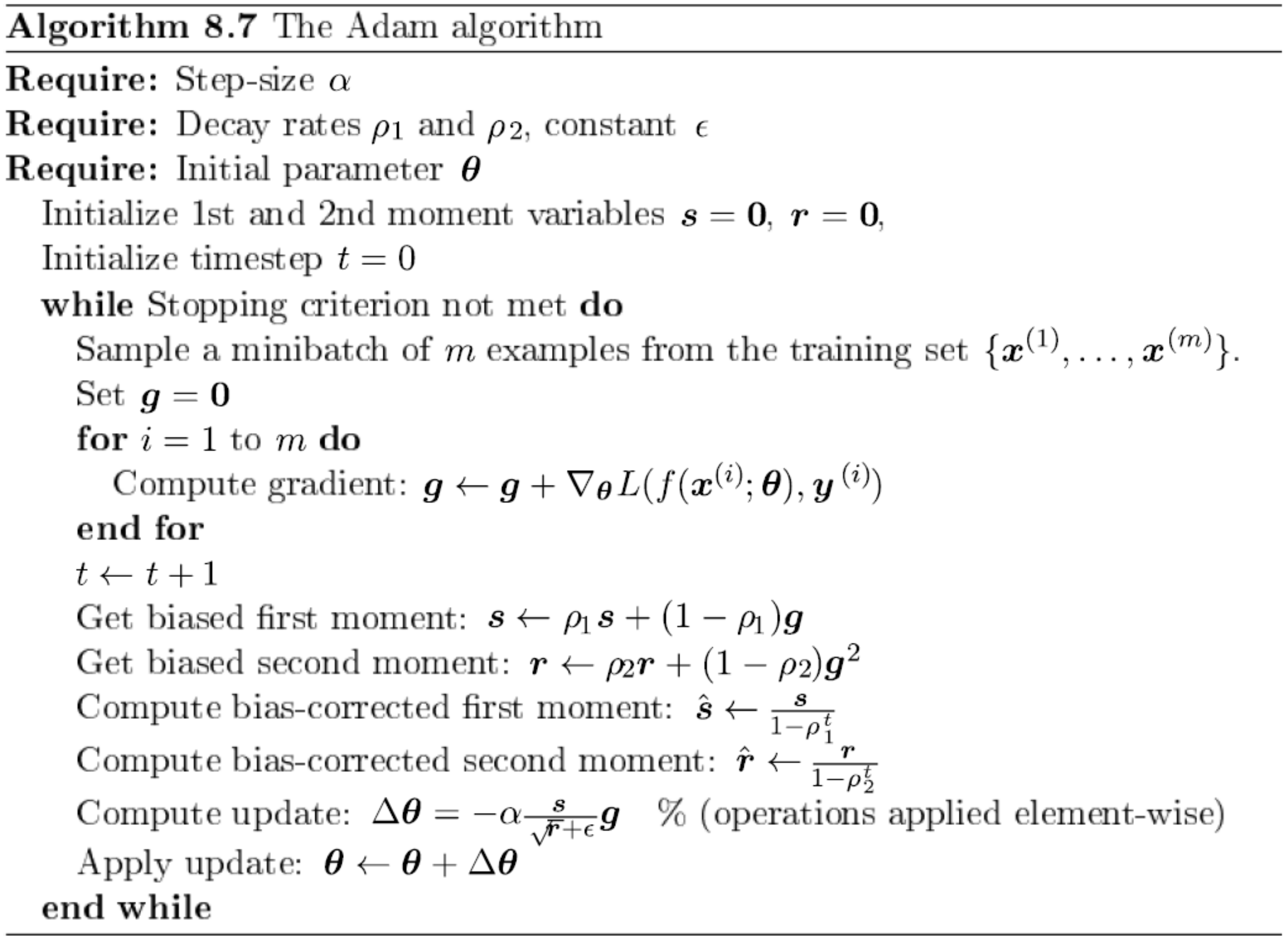


Algorithm 8.6 RMSprop algorithm with Nesterov momentum

```
Require: Global learning rate $\eta$, decay rate $\rho$, momentum coefficient $\alpha$.
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 $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$.
    Compute interim update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
    Set $\boldsymbol{g} = \mathbf{0}$
    for $i = 1$ to $m$ do
      Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    end for
    Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho)\boldsymbol{g}^2$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\eta}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$.   % ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while
```

Adam (per parameter)

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

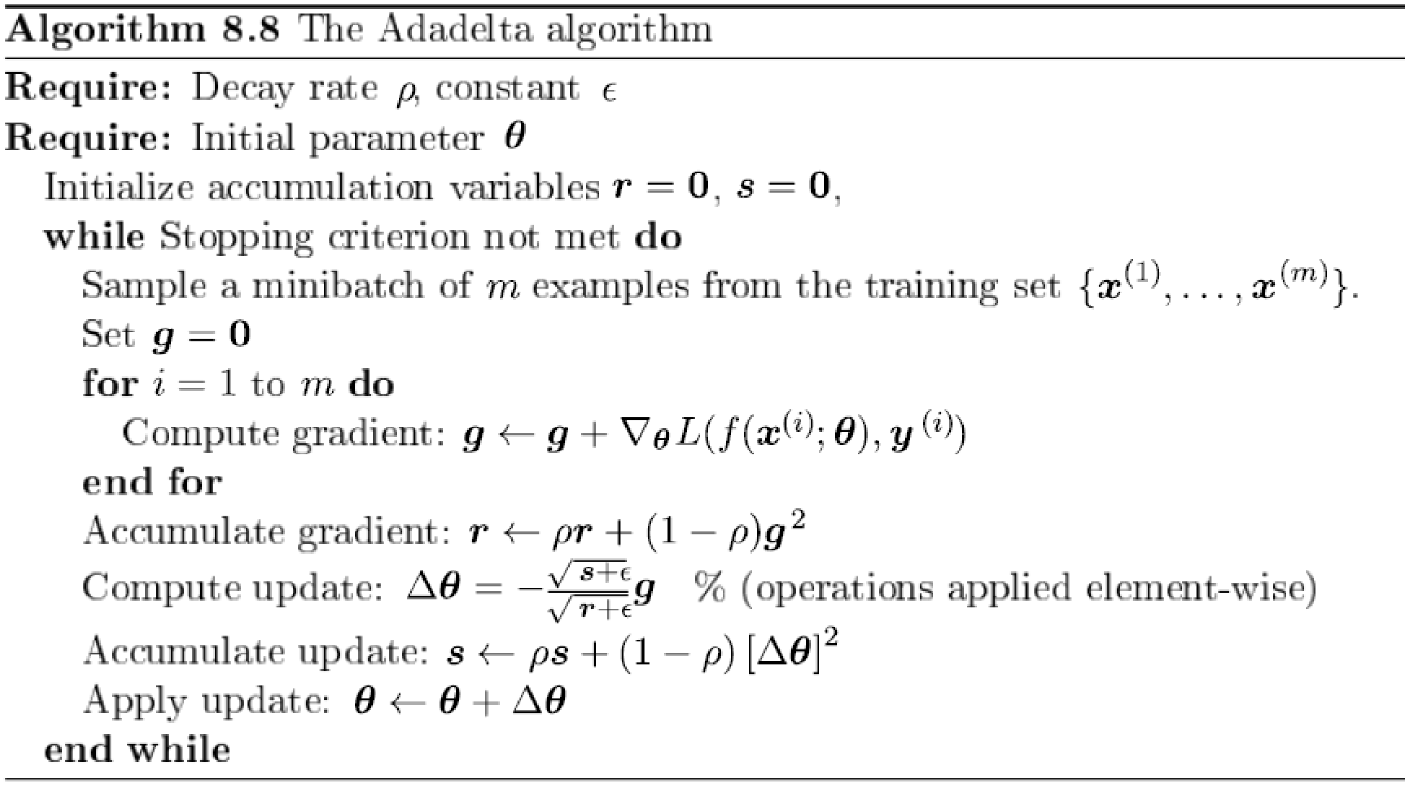


Algorithm 8.7 The Adam algorithm

```
Require: Step-size $\alpha$
Require: Decay rates $\rho_1$ and $\rho_2$, constant $\epsilon$
Require: Initial parameter $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$,
  Initialize timestep $t = 0$
  while Stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
    Set $\boldsymbol{g} = \boldsymbol{0}$
    for $i = 1$ to $m$ do
      Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    end for
    $t \leftarrow t + 1$
    Get biased first moment: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Get biased second moment: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g}^2$
    Compute bias-corrected first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$
    Compute bias-corrected second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\alpha \frac{\boldsymbol{s}}{\sqrt{\boldsymbol{r}}+\epsilon} \boldsymbol{g}$   % (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

Adadelta (per parameter)

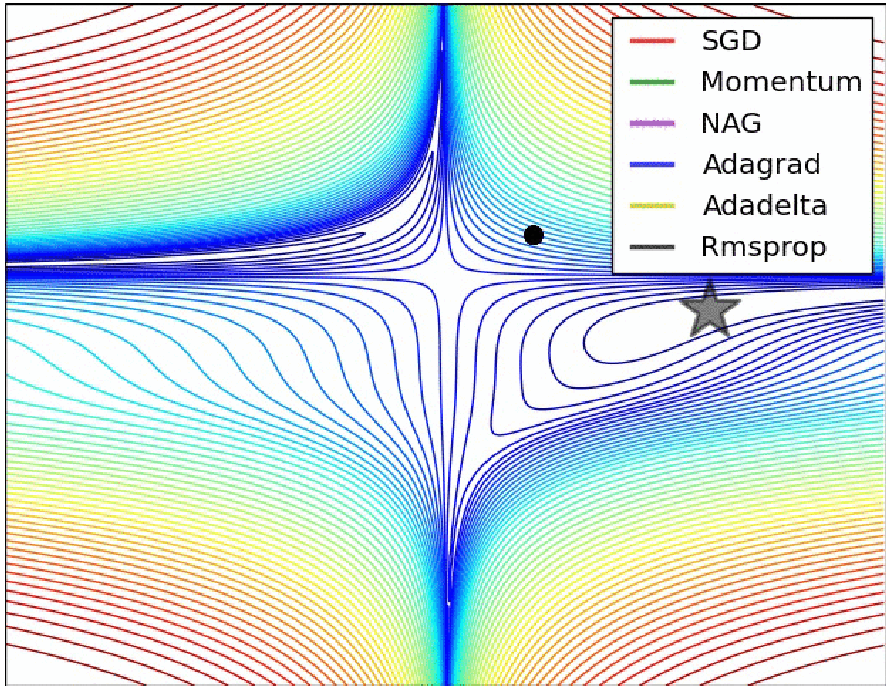
- Incorporates second-order gradient information

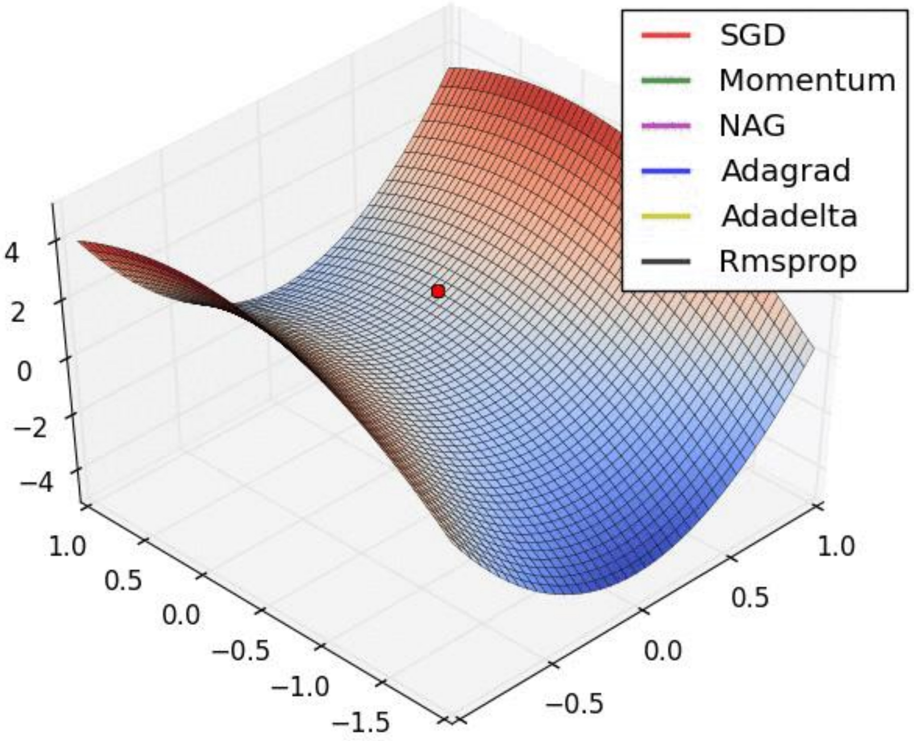


Algorithm 8.8 The Adadelta algorithm

```
Require: Decay rate $\rho$, constant $\epsilon$
Require: Initial parameter $\boldsymbol{\theta}$
  Initialize accumulation variables $\boldsymbol{r} = \mathbf{0}$, $\boldsymbol{s} = \mathbf{0}$,
  while Stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
    Set $\boldsymbol{g} = \mathbf{0}$
    for $i = 1$ to $m$ do
      Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    end for
    Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho)\boldsymbol{g}^2$
    Compute update: $\Delta\boldsymbol{\theta} = -\frac{\sqrt{\boldsymbol{s}+\epsilon}}{\sqrt{\boldsymbol{r}+\epsilon}}\boldsymbol{g}$   % (operations applied element-wise)
    Accumulate update: $\boldsymbol{s} \leftarrow \rho \boldsymbol{s} + (1 - \rho)\left[\Delta\boldsymbol{\theta}\right]^2$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

Comparison

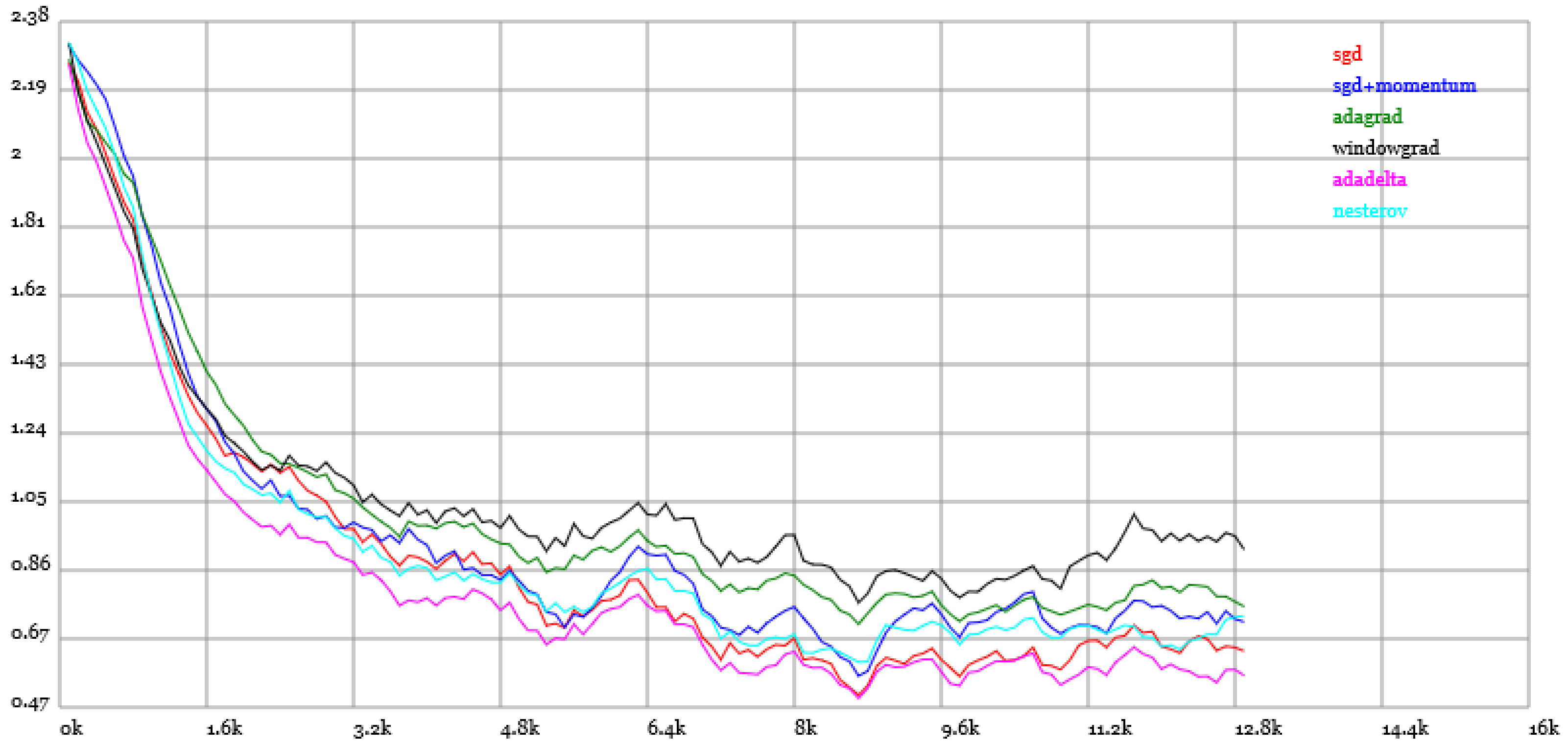




NAG: Nesterov's Accelerated Gradient

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

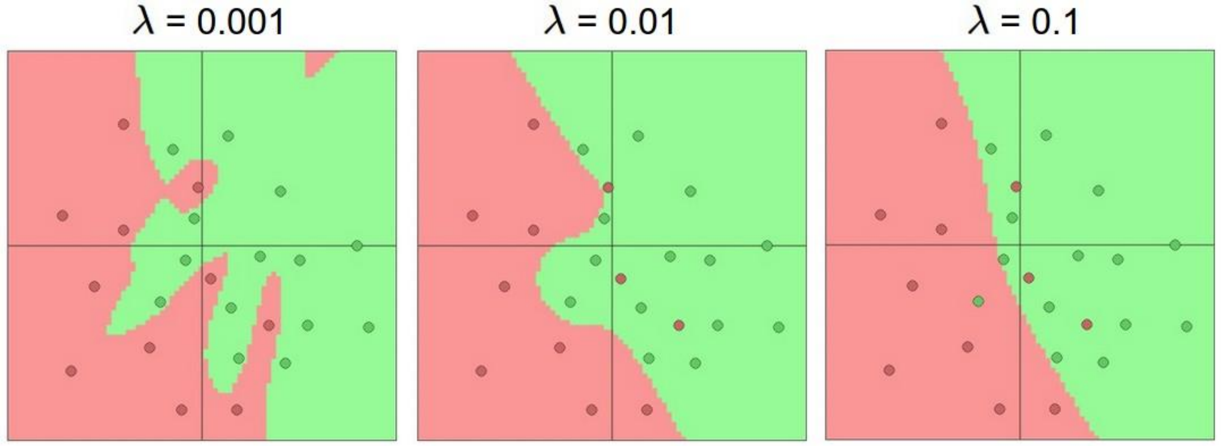
To sum up

- 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

OVERFITTING, CONVERGENCE, AND WHEN TO STOP

Overfitting

- Occurs when training procedure fits not only regularities in training data but also noise.
  - Like memorizing the training examples instead of learning the statistical regularities
- Leads to poor performance on test set
- Most of the practical issues with neural nets involve avoiding overfitting



Adapted from Michael Mozer

Avoiding Overfitting

- Increase training set size
  - Make sure effective size is growing; redundancy doesn't help
- Incorporate domain-appropriate bias into model
  - Customize model to your problem
- Set hyperparameters of model
  - number of layers, number of hidden units per layer, connectivity, etc.
- Regularization techniques
  - "smoothing" to reduce model complexity

Slide Credit: Michael Mozer

Incorporating Domain-Appropriate Bias Into Model

- Input representation
- Output representation
  - e.g., discrete probability distribution
- Architecture
  - # layers, connectivity
  - e.g., family trees net; convolutional nets
- Activation function
- Error function

Slide Credit: Michael Mozer

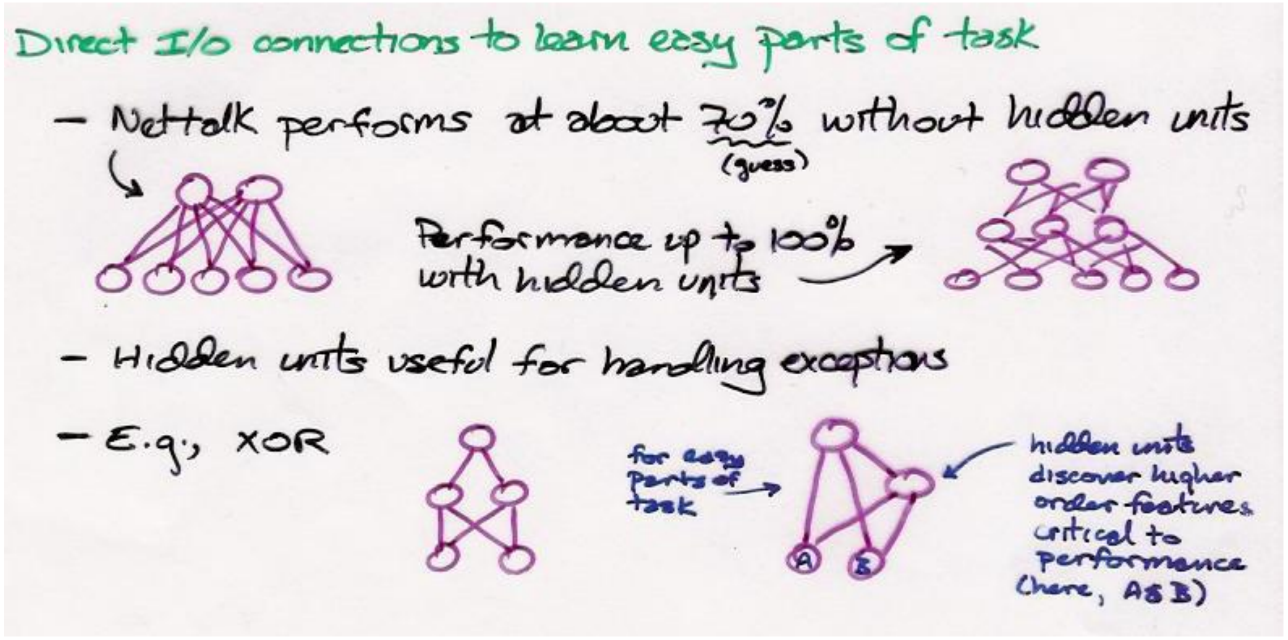
Customizing Networks

- Neural nets can be customized based on understanding of problem domain
  - choice of error function
  - choice of activation function
- Domain knowledge can be used to impose domain-appropriate bias on model
  - bias is good if it reflects properties of the data set
  - bias is harmful if it conflicts with properties of data

Slide Credit: Michael Mozer

Adding bias into a model

- Adding hidden layers or direct connections based on the problem



Direct I/O connections to learn easy parts of task

- Nettalk performs at about 70% (guess) without hidden units

Performance up to 100% with hidden units

- Hidden units useful for handling exceptions
- E.g., XOR

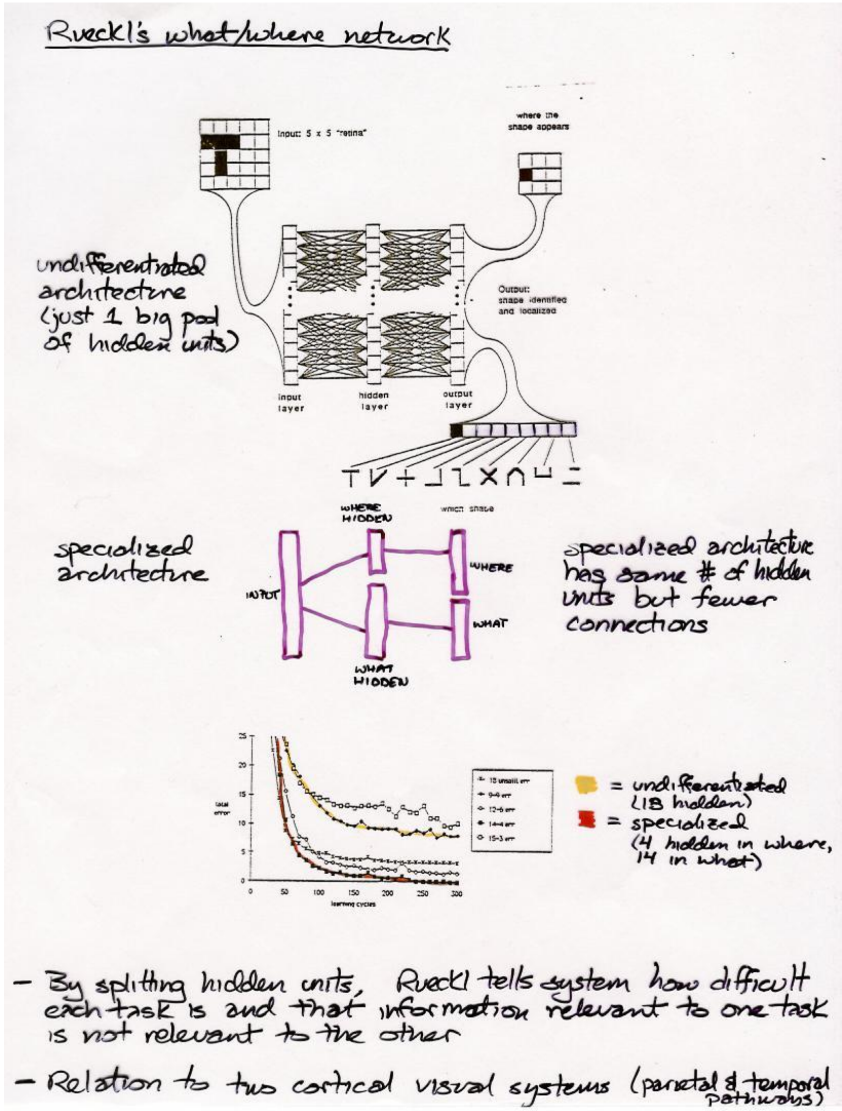
for easy parts of task

hidden units discover higher order features critical to performance (here, A & B)

Slide Credit: Michael Mozer

Adding bias into a model

- Modular architectures
  - Specialized hidden units for special problems



Rueckl's what/where network

undifferentiated architecture (just 1 big pool of hidden units)

specialized architecture

specialized architecture has same # of hidden units but fewer connections

🟨 = undifferentiated (18 hidden)

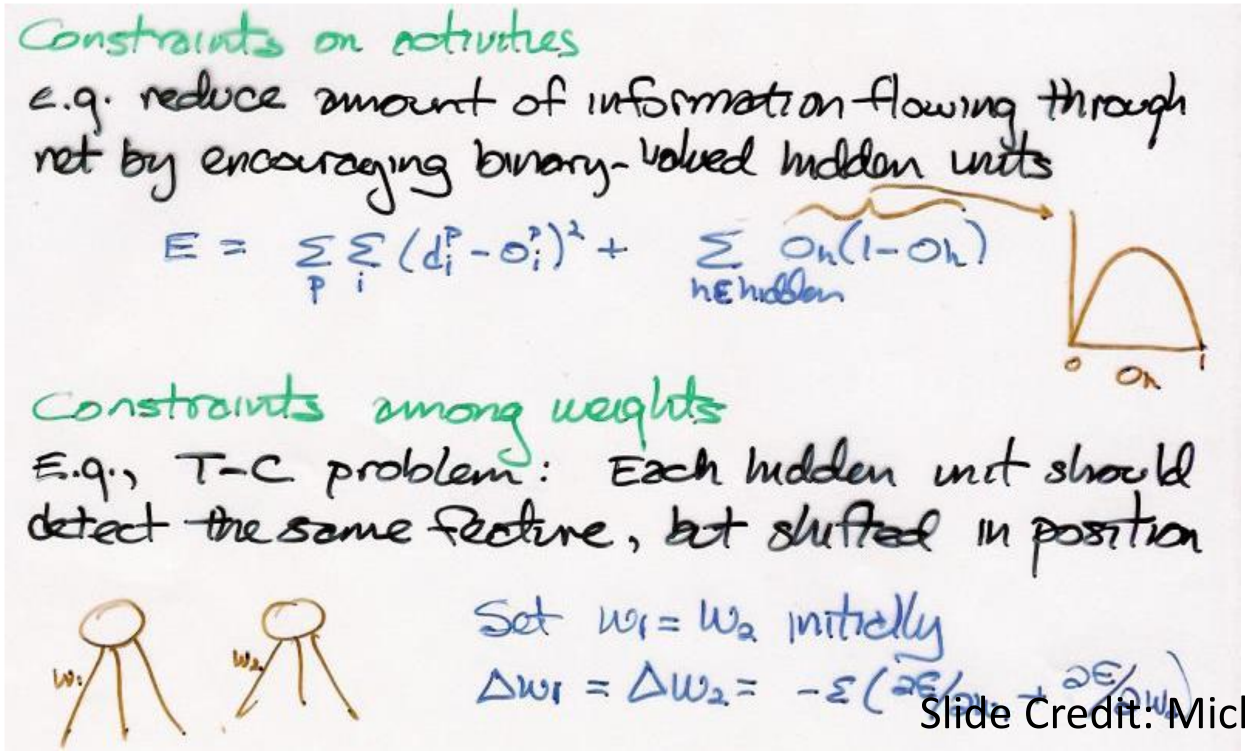
🟥 = specialized (4 hidden in where, 14 in what)

- By splitting hidden units, Rueckl tells system how difficult each task is and that information relevant to one task is not relevant to the other
- Relation to two cortical visual systems (parietal & temporal pathways)

Slide Credit: Michael Mozer

Adding bias into a model

- Local or specialized receptive fields
  - E.g., in CNNs
- Constraints on activities
- Constraints on weights



Constraints on activities

e.g. reduce amount of information flowing through net by encouraging binary-valued hidden units

$$E = \sum_p \sum_i (d_i^p - o_i^p)^2 + \underbrace{\sum_{h \in \text{hidden}} o_h(1-o_h)}$$

Constraints among weights

E.g., T-C problem: Each hidden unit should detect the same feature, but shifted in position

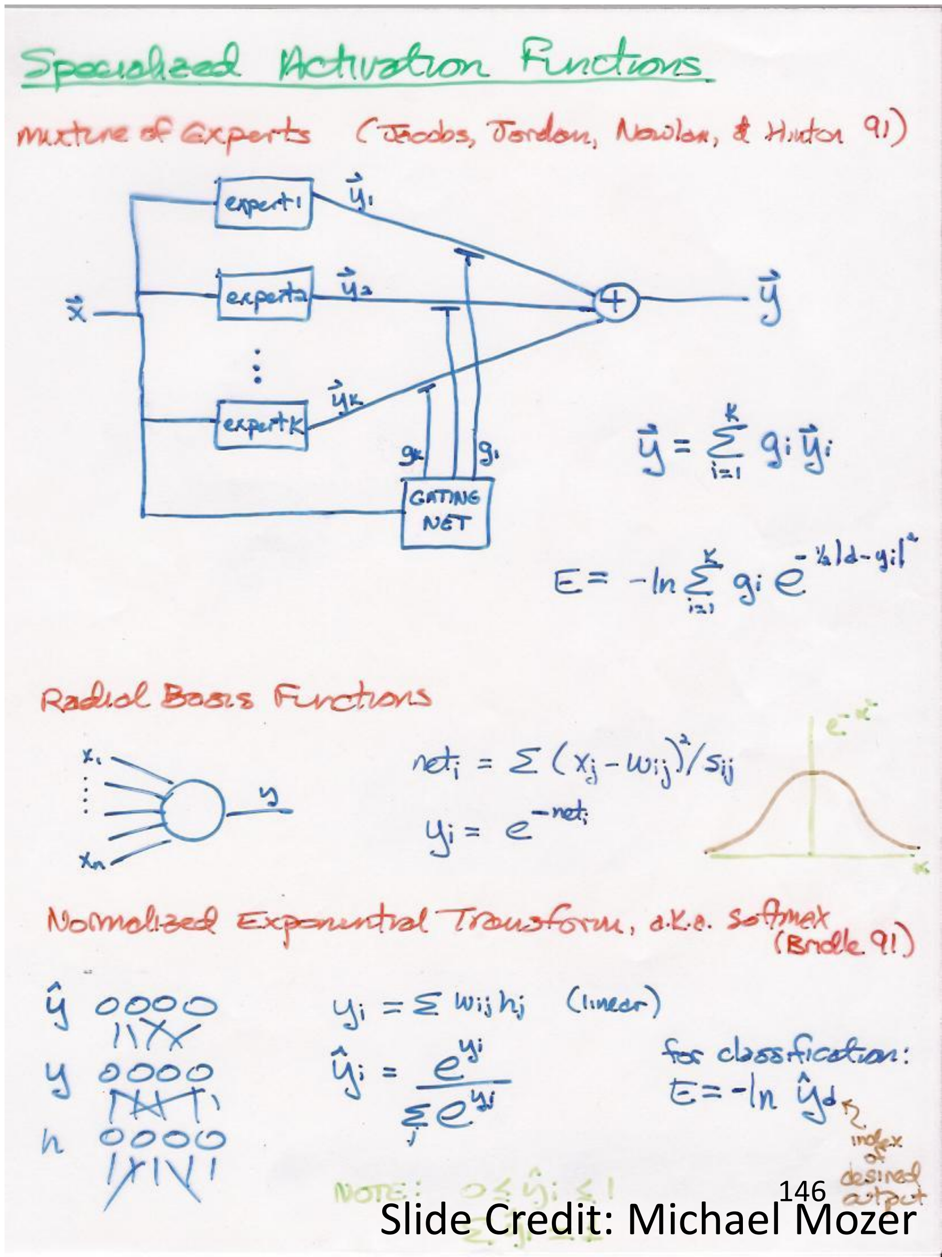
Set $w_1 = w_2$ initially

$$\Delta w_1 = \Delta w_2 = -\varepsilon\left(\partial E/\partial w_1 + \partial E/\partial w_2\right)$$

Slide Credit: Michael Mozer

Adding bias into a model

- Use different error functions (e.g., cross entropy)
- Use specialized activation functions



Specialized Activation Functions

Mixture of Experts (Jacobs, Jordan, Nowlan, & Hinton 91)

$$\vec{y} = \sum_{i=1}^{K} g_i \vec{y}_i$$

$$E = -\ln \sum_{i=1}^{K} g_i e^{-\frac{1}{2}|d - y_i|^2}$$

Radial Basis Functions

$$net_i = \sum (x_j - w_{ij})^2 / s_{ij}$$

$$y_i = e^{-net_i}$$

Normalized Exponential Transform, a.k.a. softmax (Bridle 91)

$$y_i = \sum w_{ij} h_j \quad \text{(linear)}$$

$$\hat{y}_i = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

for classification:

$$E = -\ln \hat{y}_d$$

index of desired output

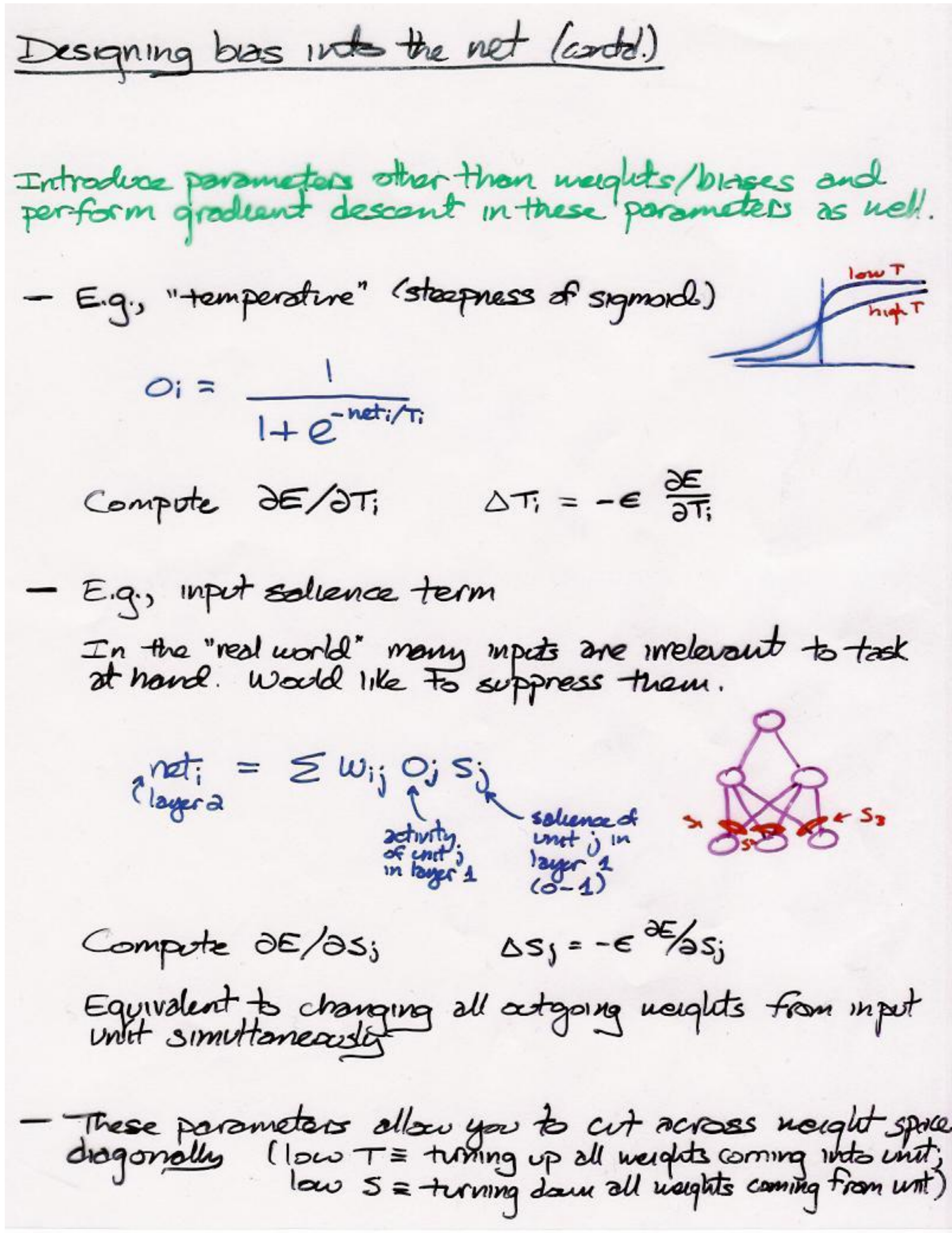
NOTE: $0 \le \hat{y}_i \le 1$

146

Slide Credit: Michael Mozer

Adding bias into a model

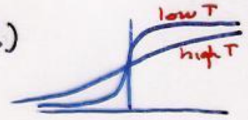
- Introduce other parameters
  - Temperature
  - Saliency of input



Designing bias into the net (contd.)

Introduce parameters other than weights/biases and perform gradient descent in these parameters as well.

– E.g., "temperature" (steepness of sigmoid)



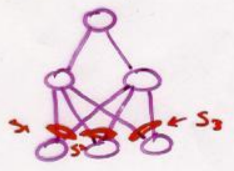
$$o_i = \frac{1}{1+e^{-net_i/T_i}}$$

Compute $\partial E/\partial T_i$ $\Delta T_i = -\epsilon \frac{\partial E}{\partial T_i}$

– E.g., input salience term

In the "real world" many inputs are irrelevant to task at hand. Would like to suppress them.

$$\underset{\text{layer 2}}{net_i} = \sum w_{ij}\, \underset{\text{activity of unit } j \text{ in layer 1}}{o_j}\, \underset{\text{salience of unit } j \text{ in layer 1 (0–1)}}{s_j}$$



Compute $\partial E/\partial s_j$ $\Delta s_j = -\epsilon\, \partial E/\partial s_j$

Equivalent to changing all outgoing weights from input unit simultaneously

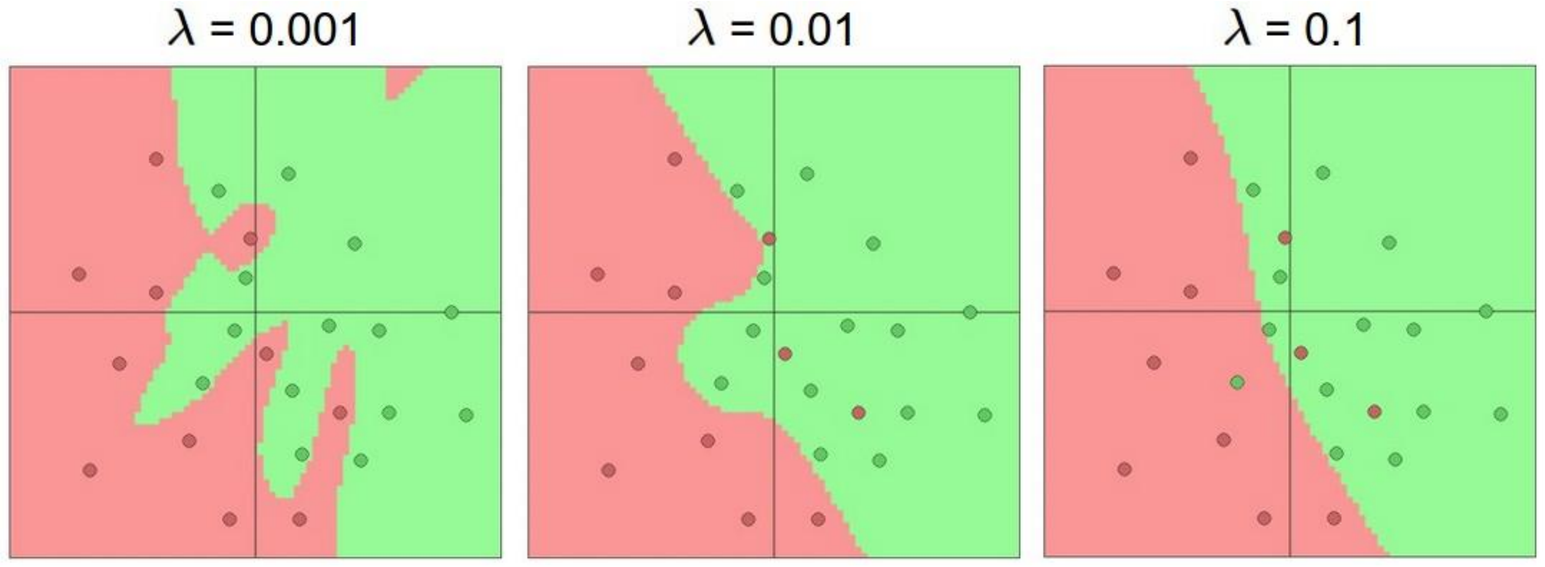
– These parameters allow you to cut across weight space diagonally (low T ≡ turning up all weights coming into unit; low s ≡ turning down all weights coming from unit)

Slide Credit: Michael Mozer

Regularization

- Regularization strength can effect overfitting

$$\frac{1}{2}\lambda w^2$$

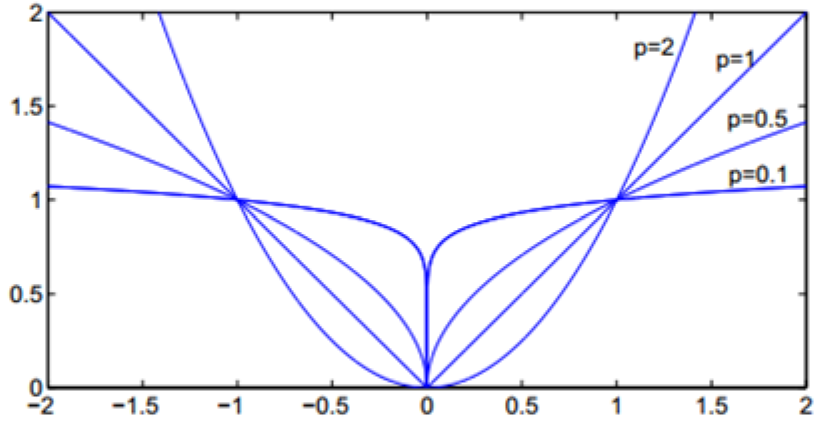


Regularization

- L2 regularization: $\frac{1}{2}\lambda w^2$
  - Very common
  - Penalizes peaky weight vector, prefers diffuse weight vectors
- L1 regularization: $\lambda|w|$
  - Enforces sparsity (some weights become zero)
  - Leads to input selection (makes it noise robust)
  - Use it if you require sparsity / feature selection
- Can be combined: $\lambda_1|w| + \lambda_2 w^2$
- Regularization is not performed on the bias; it seems to make no significant difference

L0 regularization

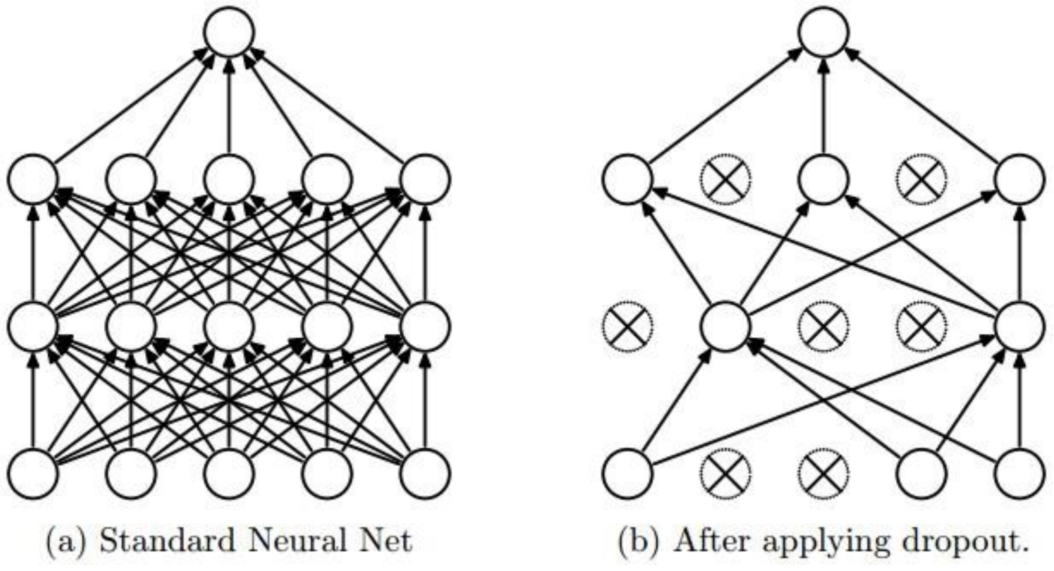
- $L_0 = \left(\sum_i x_i^0\right)^{1/0}$
- How to compute the zeroth power and zeroth-root?
- Mathematicians approximate this as:
  - $L_0 = \#(i \mid x_i \neq 0)$
  - The cardinality of non-zero elements
- This is a strong enforcement of sparsity.
- However, this is non-convex
  - L1 norm is the closest convex form



Regularization

- Enforce an upper bound on weights:
  - Max norm:
    - $||w||_2 < c$
    - Helps the gradient explosion problem
    - Improvements reported
- Dropout:
  - At each iteration, *drop* a number of neurons in the network
  - **<u>Use a neuron's activation</u>** with probability p (a hyperparameter)
  - Adds stochasticity!

Fig: Srivastava et al., 2014



http://cs231n.github.io/neural-networks-2/

Regularization: Dropout

- Feed-forward only on active units
- Can be trained using SGD with mini-batch
  - Back propagate only "active" units.
- One issue:
  - Expected output x with dropout:
  - $E[x] = px + (1-p)0$
- To have the same scale at testing time (no dropout), multiply test-time activations with p.

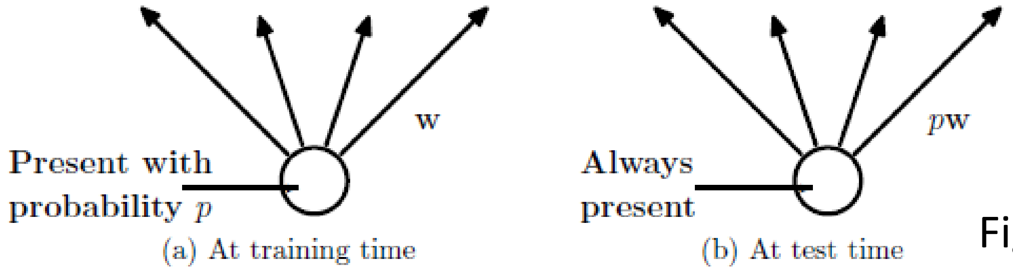


Fig: Srivastava et al., 2014

Regularization: Dropout

Training-time:

```
# forward pass for example 3-layer neural network
H1 = np.maximum(0, np.dot(W1, X) + b1)
U1 = np.random.rand(*H1.shape) < p # first dropout mask
H1 *= U1 # drop!
H2 = np.maximum(0, np.dot(W2, H1) + b2)
U2 = np.random.rand(*H2.shape) < p # second dropout mask
H2 *= U2 # drop!
out = np.dot(W3, H2) + b3
```

Test-time:

```
# ensembled forward pass
H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
out = np.dot(W3, H2) + b3
```

http://cs231n.github.io/neural-networks-2/

Regularization: Inverted Dropout

Perform scaling while dropping at training time!

Training-time:

```
# forward pass for example 3-layer neural network
H1 = np.maximum(0, np.dot(W1, X) + b1)
U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
H1 *= U1 # drop!
H2 = np.maximum(0, np.dot(W2, H1) + b2)
U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
H2 *= U2 # drop!
out = np.dot(W3, H2) + b3
```

Test-time:

```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

http://cs231n.github.io/neural-networks-2/

Regularization Summary

- L2 regularization
- Inverted dropout with $p = 0.5$ (tunable)

When To Stop Training

- 1. Train *n* epochs; lower learning rate; train *m* epochs
  - bad idea: can’t assume one-size-fits-all approach
- 2. Error-change criterion
  - stop when error isn’t dropping
  - recommendation: criterion based on % drop over a window of, say, 10 epochs
    - 1 epoch is too noisy
    - absolute error criterion is too problem dependent
  - Another idea: train for a fixed number of epochs after criterion is reached (possibly with lower learning rate)

Slide Credit: Michael Mozer

When To Stop Training

- 3. Weight-change criterion
  - Compare weights at epochs $(t-10)$ and t and test:

$$\max_i \left| w_i^t - w_i^{t-10} \right| < q$$

  - Don't base on length of overall weight change vector
  - Possibly express as a percentage of the weight
  - Be cautious: small weight changes at critical points can result in rapid drop in error

Slide Credit: Michael Mozer

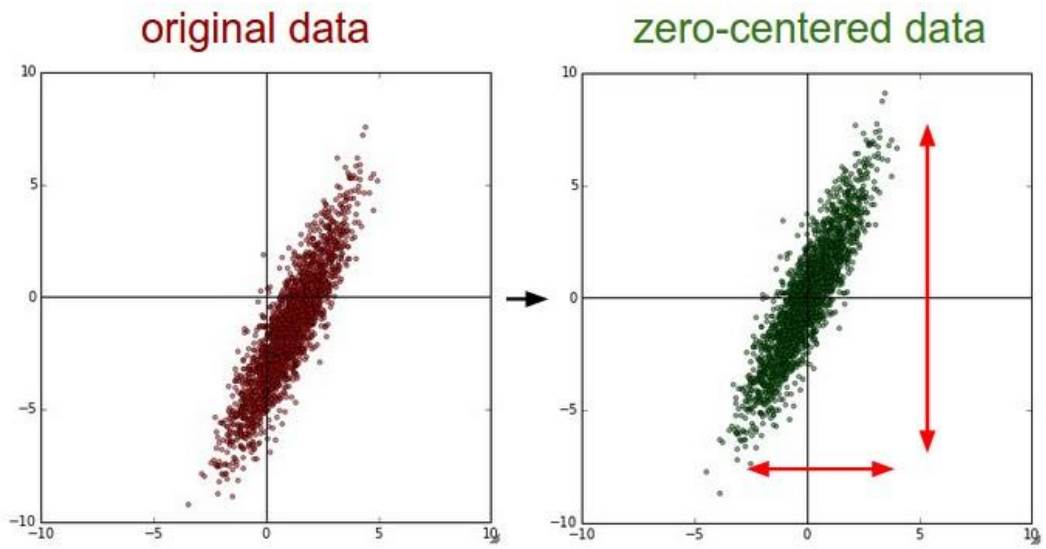
DATA PREPROCESSING AND WEIGHT INITIALIZATION

Data Preprocessing

- Mean subtraction
- Normalization
- PCA and whitening

Data Preprocessing: Mean subtraction

- Subtract the mean for each dimension:
$$x_i' = x_i - \hat{x}_i$$
- Effect: Move the data center (mean) to coordinate center



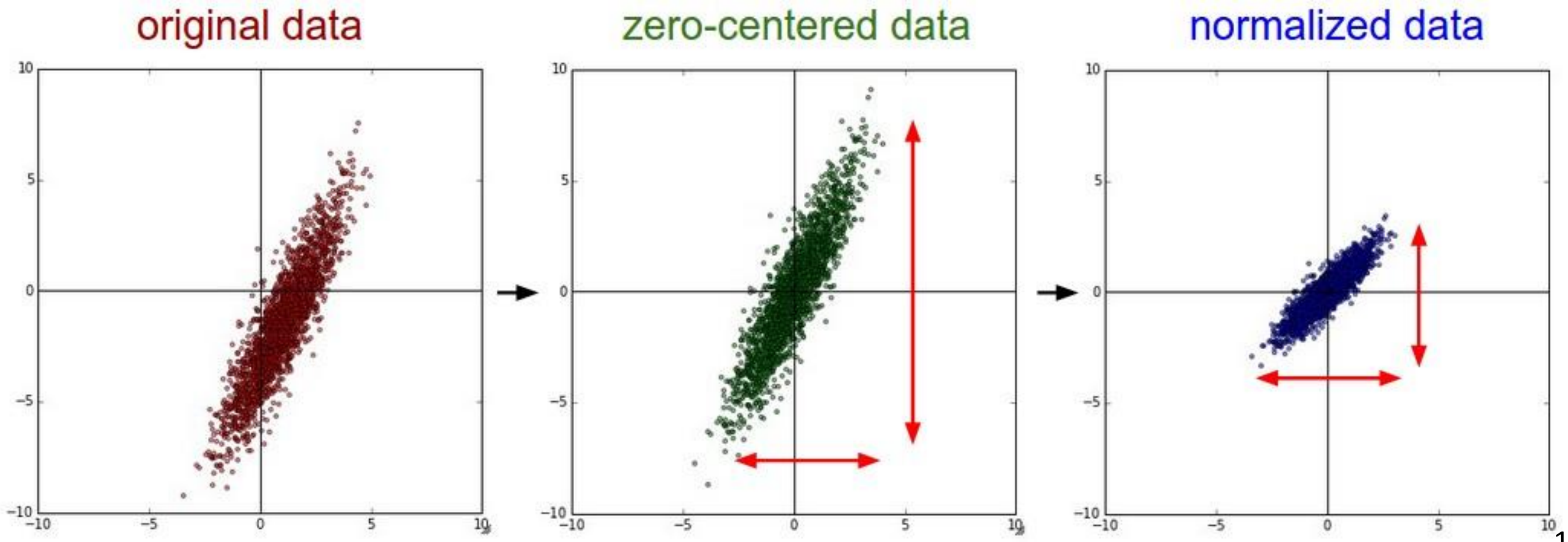
http://cs231n.github.io/neural-networks-2/

Data Preprocessing: Normalization (or conditioning)

- Necessary if you believe that your dimensions have different scales
  - Might need to reduce this to give equal importance to each dimension
- Normalize each dimension by its std. dev. after mean subtraction:

$$x_i' = x_i - \mu_i$$
$$x_i'' = x_i' / \sigma_i$$

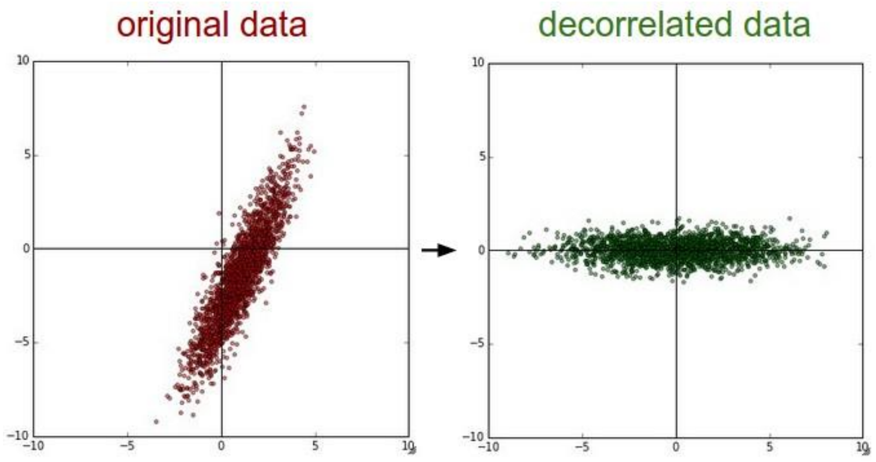
- Effect: Make the dimensions have the same scale



http://cs231n.github.io/neural-networks-2/

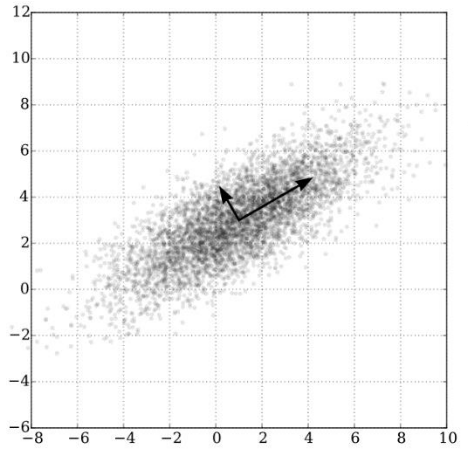
Data Preprocessing: Principle Component Analysis

- First center the data
- Find the eigenvectors $e_1, \dots, e_n$
- Project the data onto the eigenvectors:
  - $x_i^R = x_i \cdot [e_1, \dots, e_n]$
- This corresponds to rotating the data to have the eigenvectors as the axes
- If you take the first M eigenvectors, it corresponds to dimensionality reduction



http://cs231n.github.io/neural-networks-2/

Reminder: PCA



- Principle axes are the eigenvectors of the covariance matrix:

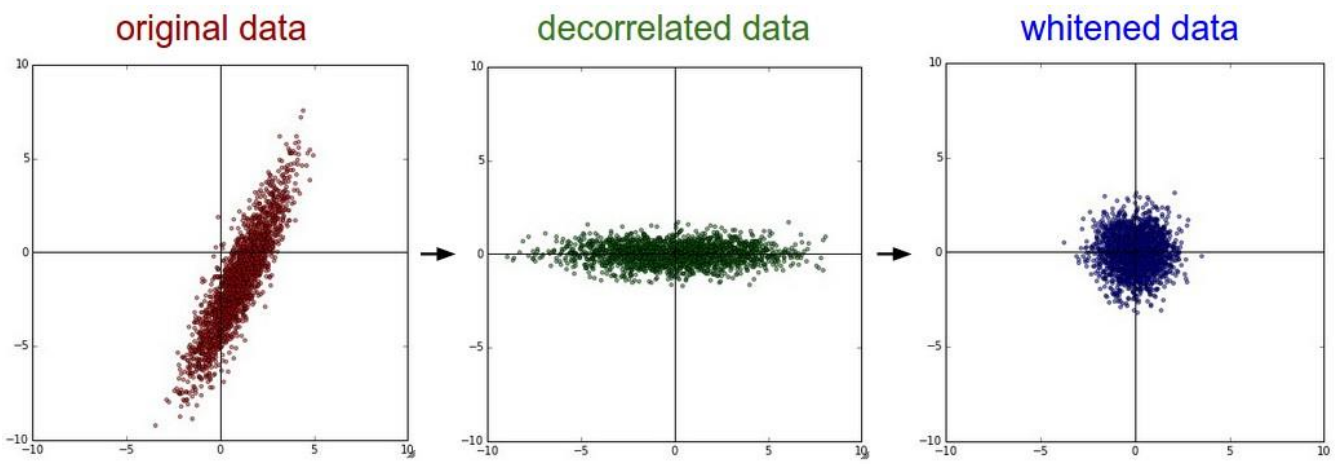
$$\Sigma = \begin{bmatrix} \mathrm{E}[(X_1-\mu_1)(X_1-\mu_1)] & \mathrm{E}[(X_1-\mu_1)(X_2-\mu_2)] & \cdots & \mathrm{E}[(X_1-\mu_1)(X_n-\mu_n)] \\ \mathrm{E}[(X_2-\mu_2)(X_1-\mu_1)] & \mathrm{E}[(X_2-\mu_2)(X_2-\mu_2)] & \cdots & \mathrm{E}[(X_2-\mu_2)(X_n-\mu_n)] \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{E}[(X_n-\mu_n)(X_1-\mu_1)] & \mathrm{E}[(X_n-\mu_n)(X_2-\mu_2)] & \cdots & \mathrm{E}[(X_n-\mu_n)(X_n-\mu_n)] \end{bmatrix}.$$

Data Preprocessing: Whitening

- Normalize the scale with the norm of the eigenvalue:

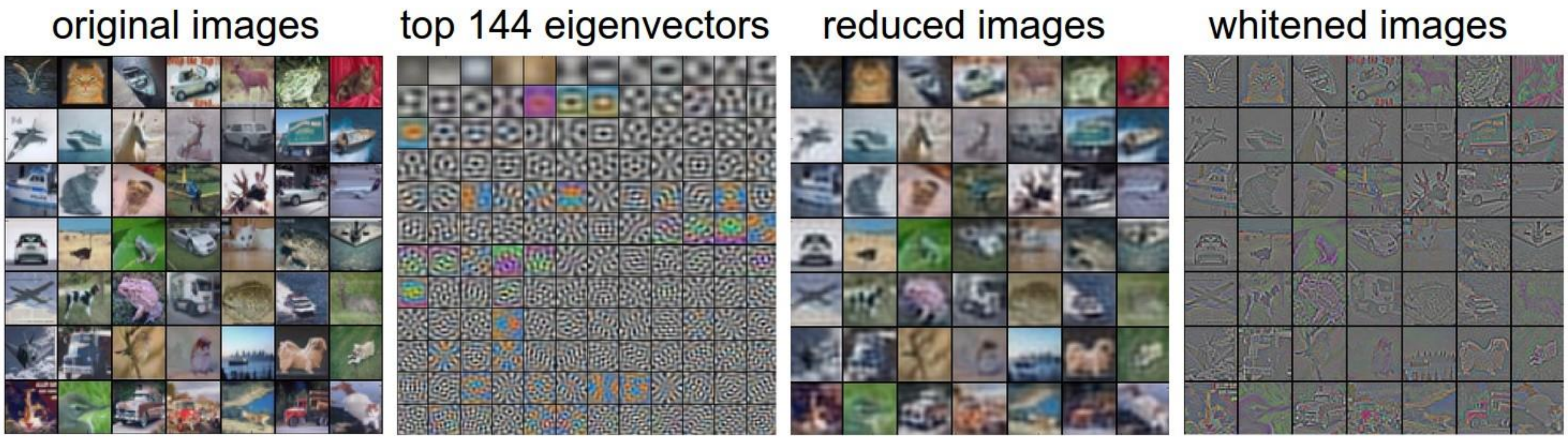
$$x_i^w = x_i^R / ([\lambda_1, \ldots, \lambda_n] + \epsilon)$$

- ϵ: a very small number to avoid division by zero
- This stretches each dimension to have the same scale.
- Side effect: this may exaggerate noise.



http://cs231n.github.io/neural-networks-2/

Data Preprocessing: Example



Data Preprocessing: Summary

- We mostly don’t use PCA or whitening
  - They are computationally very expensive
  - Whitening has side effects
- It is quite crucial and common to zero-center the data
- Most of the time, we see normalization with the std. deviation

Weight Initialization

- Zero weights
  - Wrong!
  - Leads to updating weights by the same amounts for every input
  - Symmetry!
- Initialize the weights randomly to small values:
  - Sample from a small range, e.g., Normal(0,0.01)
  - Don’t initialize too small
- The bias may be initialized to zero
  - For ReLU units, this may be a small number like 0.01.

Note: None of these provide guarantees. Moreover, there is no guarantee that one of these will always be better.

More on weight initialization

- Integrate the following

- http://www.jefkine.com/deep/2016/08/08/initialization-of-deep-networks-case-of-rectifiers/

Initial Weight Normalization

- Problem:
  - Variance of the output changes with the number of inputs
  - If $s = \sum_i w_i x_i$:

$$
\begin{aligned}
\mathrm{Var}(s) &= \mathrm{Var}\left(\sum_i^n w_i x_i\right) \\
&= \sum_i^n \mathrm{Var}(w_i x_i) \\
&= \sum_i^n [E(w_i)]^2 \mathrm{Var}(x_i) + E[(x_i)]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= \sum_i^n \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= (n\mathrm{Var}(w))\,\mathrm{Var}(x)
\end{aligned}
$$

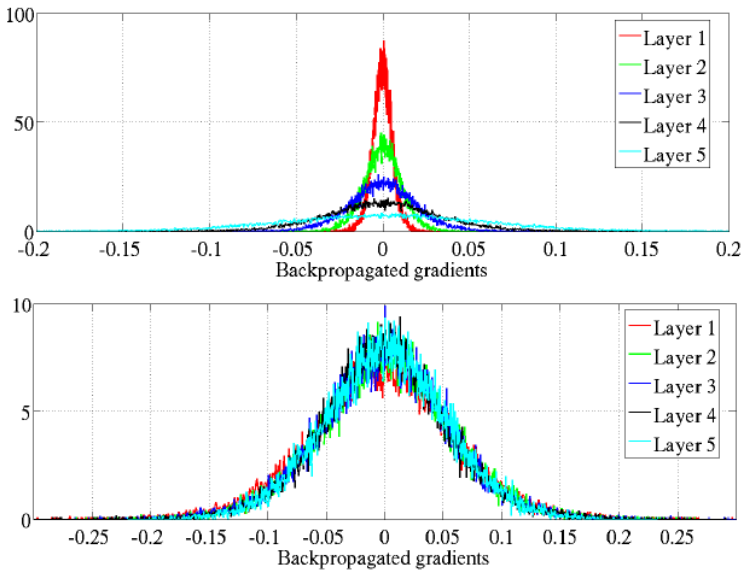


Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

Glorot & Bengio, “Understanding the difficulty of training deep feedforward neural networks”, 2010.

Initial Weight Normalization

- Solution:
  - Get rid of n in $Var(s) = (n \, \mathrm{Var(w)})\mathrm{Var(x)}$
  - How?
- $w_i = rand(0,1)/\sqrt{n}$
  - Why?
  - $Var(aX) = a^2 Var(X)$
- If the number of inputs & outputs are not fixed:
  - $w_i = rand(0,1) \times \frac{2}{\sqrt{n_{in}+n_{out}}}$

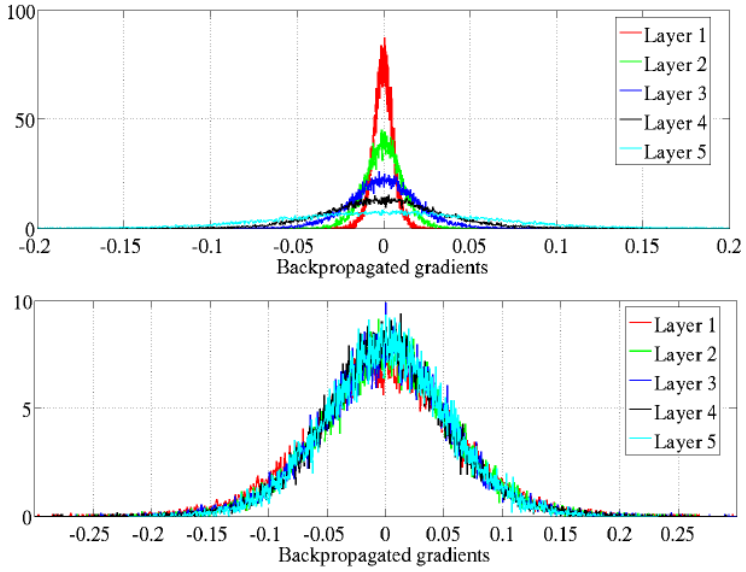
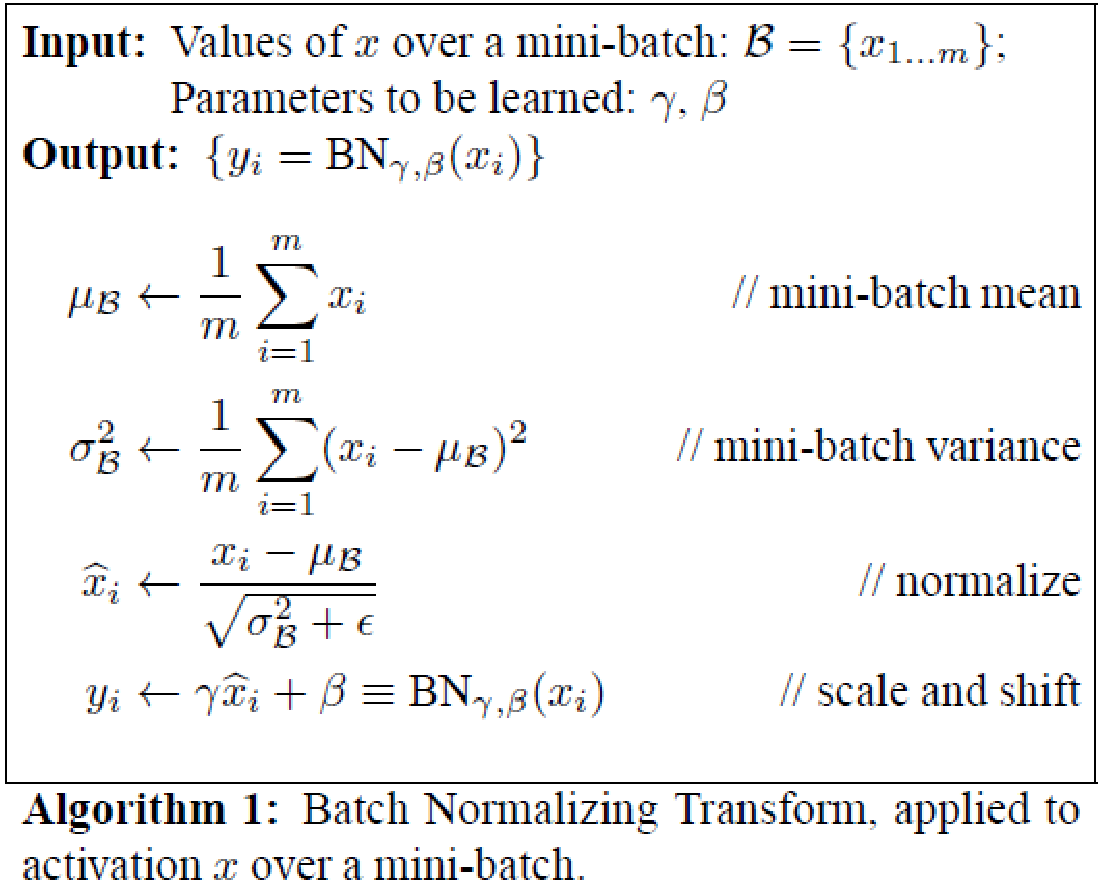


Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

Glorot & Bengio, "Understanding the difficulty of training deep feedforward neural networks", 2010.

Alternative: Batch Normalization

- Normalization is differentiable
  - So, make it part of the model (not only at the beginning)
  - I.e., perform normalization during every step of processing
- More robust to initialization



```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;
       Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

    $\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                          // mini-batch mean
    $\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2$     // mini-batch variance
    $\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$                   // normalize
    $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$          // scale and shift
```

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Ioffe & Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, 2015.

To sum up

- Initialization and normalization are crucial
- Different initialization & normalization strategies may be needed for different deep learning methods
  - E.g., in CNNs, normalization might be performed only on convolution etc.
- More on this later

Today

- Finish up the first part of the course
  - Loss functions again
  - Representational capacity
  - Other practical issues
- A crash course on Computer Vision & Human Vision
  - What can we learn from Human Vision?
- Autoencoders

LOSS FUNCTIONS, AGAIN

Loss functions

- **A single correct** label case (classification):
  - Hinge loss:
    - $L_i = \sum_{j \neq y_i} \max\left(0, f_j - f_{y_i} + 1\right)$
  - Squared hinge loss:
    - $L_i = \sum_j \left|f_j \ - y_{ji}\right|^2$
  - Soft-max:
    - $L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$

Loss functions

- **Many correct** labels case:
  - Binary prediction for each label, independently:
    - $L_i = \sum_j \max\left(0, 1 - y_{ij} f_j\right)$
    - $y_{ij} = +1$ if example i is labeled with label j; otherwise $y_{ij} = -1$.
  - Alternatively, train logistic regression classifier for each label (0 or 1):

$$L_i = \sum_j y_{ij} \log(\sigma(f_j)) + (1 - y_{ij}) \log(1 - \sigma(f_j))$$

Loss functions

- Regression case (a continuous label):
  - L2 Norm squared (L2 Loss):
    - $L_i = \left|\left|f - y_i\right|\right|_2^2$
    - $\frac{\partial L_i}{\partial f_j} = f_j - (y_i)_j$
  - L1 Norm:
    - $L_i = \left|\left|f - y_i\right|\right|_1 = \sum_j \left|f_j - (y_i)j\right|$
    - $\frac{\partial L_i}{\partial f_j} = sign\left(f_j - (y_i)_j\right)$

Reminder:

$\|x\|_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{\frac{1}{p}}$.

L2 Loss: Caution

- L2 loss asks for a more difficult constraint:
  - Learn to output a response that is exactly the same as the correct label
  - This is harder to train

- Compare, e.g., softmax:
  - Which asks only one response to be maximum than others.

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
  - Not very common

REPRESENTATIONAL CAPACITY

Representational capacity

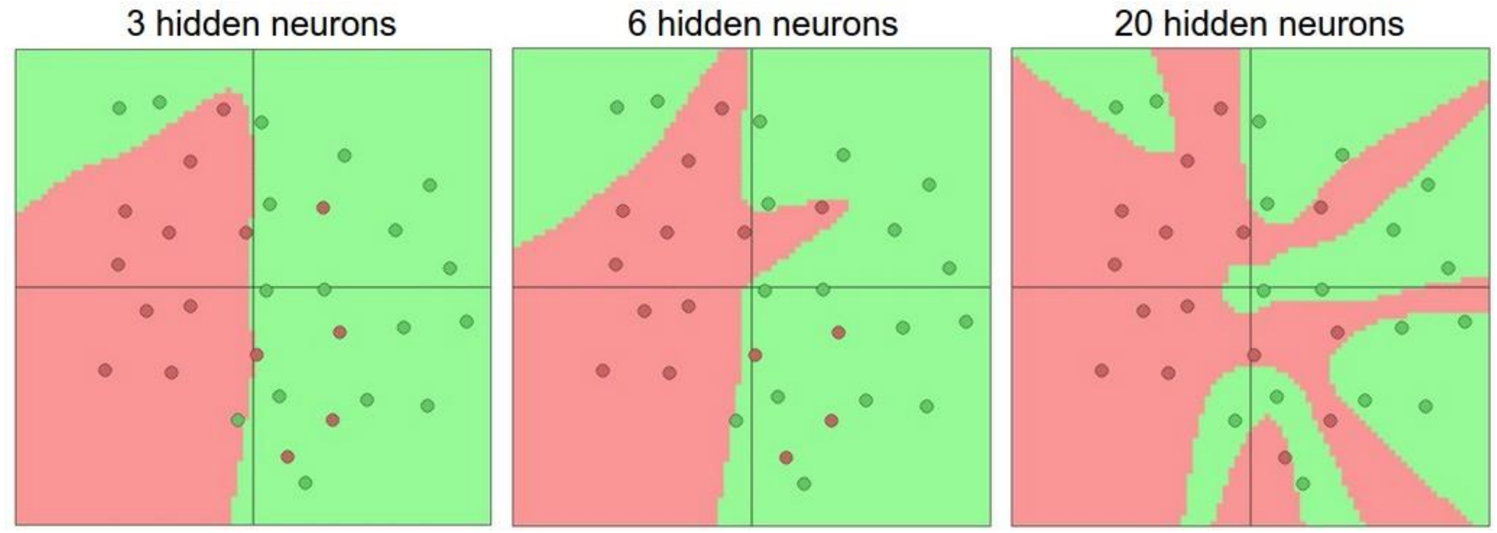
- Boolean functions:
  - Every Boolean function can be represented exactly by a neural network
  - The number of hidden layers might need to grow with the number of inputs
- Continuous functions:
  - Every bounded continuous function can be approximated with small error with two layers
- Arbitrary functions:
  - Three layers can approximate any arbitrary function

Representational Capacity: Why go deeper if 3 layers is sufficient?

- Going deeper helps convergence in “big” problems.
- Going deeper in “old-fashion trained” ANNs does not help much in accuracy
  - However, with different training strategies or with Convolutional Networks, going deeper matters

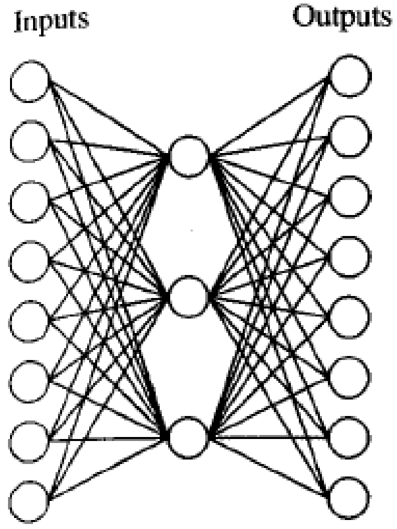
Representational Capacity

- More hidden neurons ➔ capacity to represent more complex functions



- Problem: overfitting vs. generalization
  - We will discuss the different strategies to help here (L2 regularization, dropout, input noise, using a validation set etc.)

What the hidden units represent



Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

FIGURE 4.7
Learned Hidden Layer Representation. This $8 \times 3 \times 8$ network was trained to learn the identity function, using the eight training examples shown. After 5000 training epochs, the three hidden unit values encode the eight distinct inputs using the encoding shown on the right. Notice if the encoded values are rounded to zero or one, the result is the standard binary encoding for eight distinct values.

Number of hidden neurons

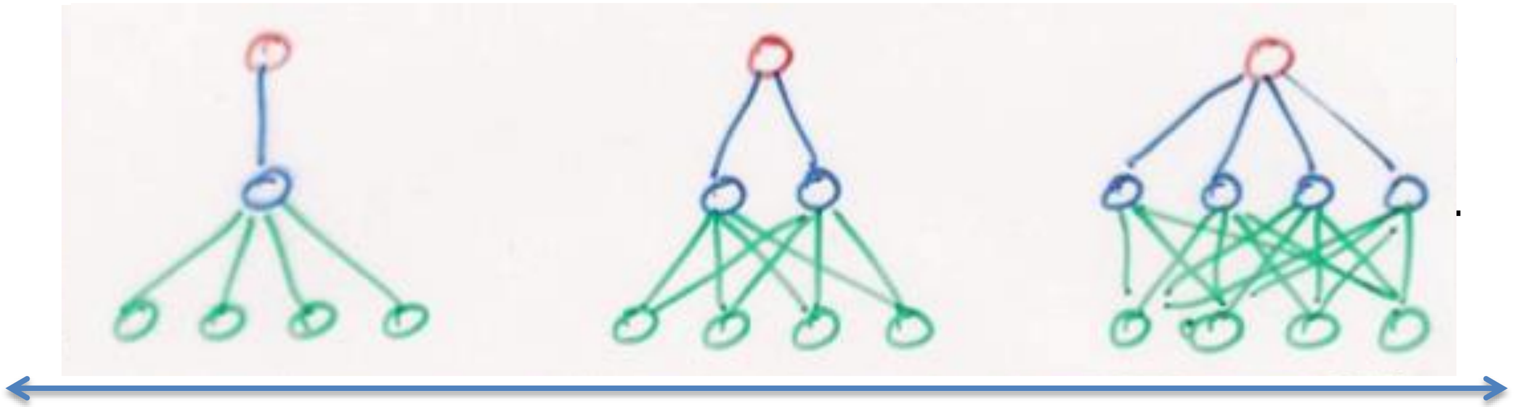
- Several rule of thumbs (Jeff Heaton)
  - The number of hidden neurons should be between the size of the input layer and the size of the output layer.
  - The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
  - The number of hidden neurons should be less than twice the size of the input layer.

Number of hidden layers

- Depends on the nature of the problem
  - Linear classification? ➔ No hidden layers needed
  - Non-linear classification?

Model Complexity

- Models range in their flexibility to fit arbitrary data



simple model
high bias

constrained
low variance

small capacity may prevent it from representing all structure in data

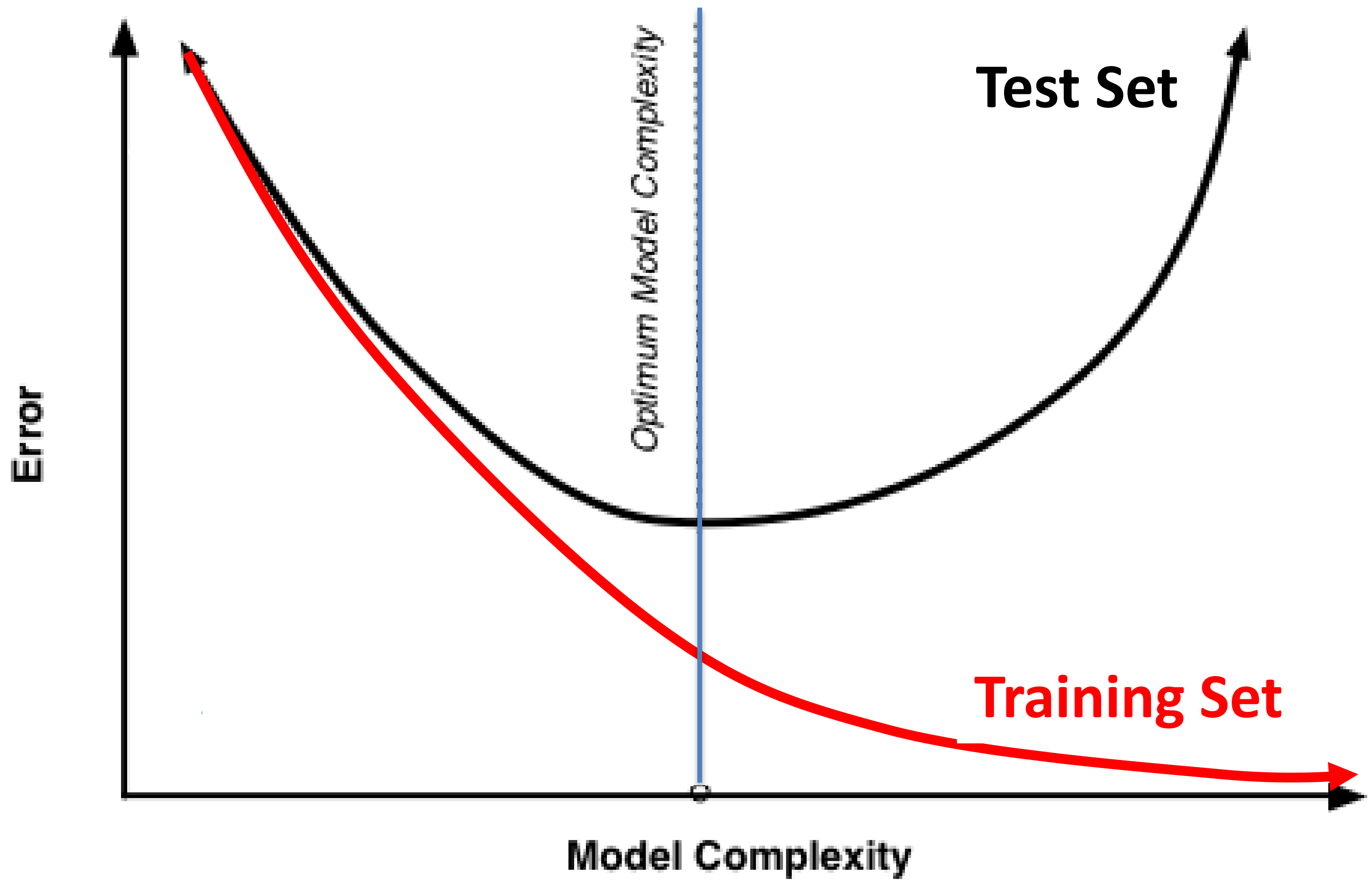
complex model
low bias

unconstrained
high variance

large capacity may allow it to memorize data and fail to capture regularities

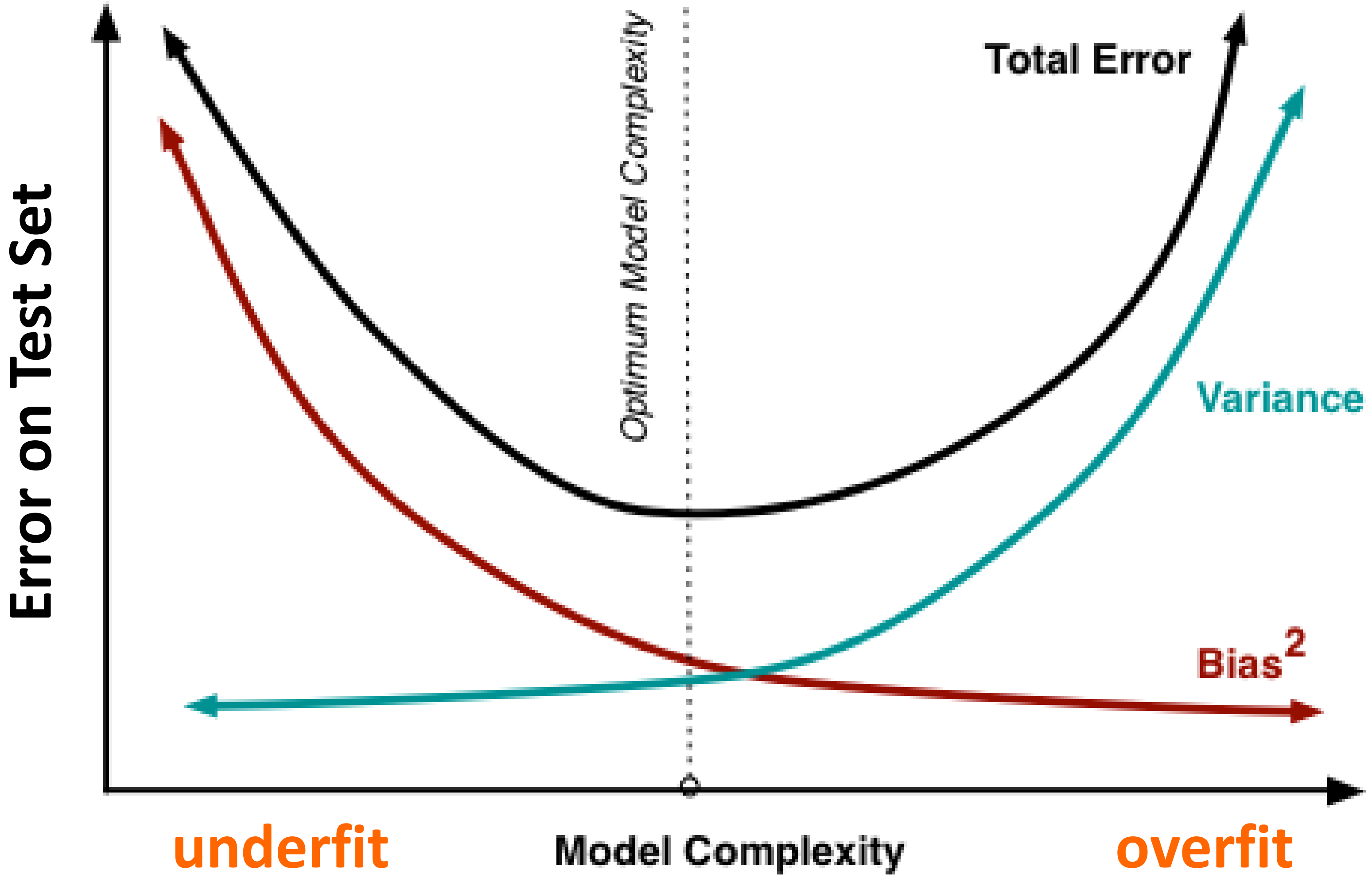
Slide Credit: Michael Mozer

Training Vs. Test Set Error



Slide Credit: Michael Mozer

Bias-Variance Trade Off



Slide Credit: Michael Mozer

image credit: scott.fortmann-roe.com

ISSUES & PRACTICAL ADVICES

Issues & tricks

- Vanishing gradient
  - Saturated units block gradient propagation (why?)
  - A problem especially present in recurrent networks or networks with a lot of layers
- Overfitting
  - Drop-out, regularization and other tricks.
- Tricks:
  - Unsupervised pretraining
- Batch normalization (each unit's preactivation is normalized)
  - Helps keeping the preactivation non-saturated
  - Do this for mini-batches (adds stochasticity)
  - Backprop needs to be updated

Unsupervised pretraining

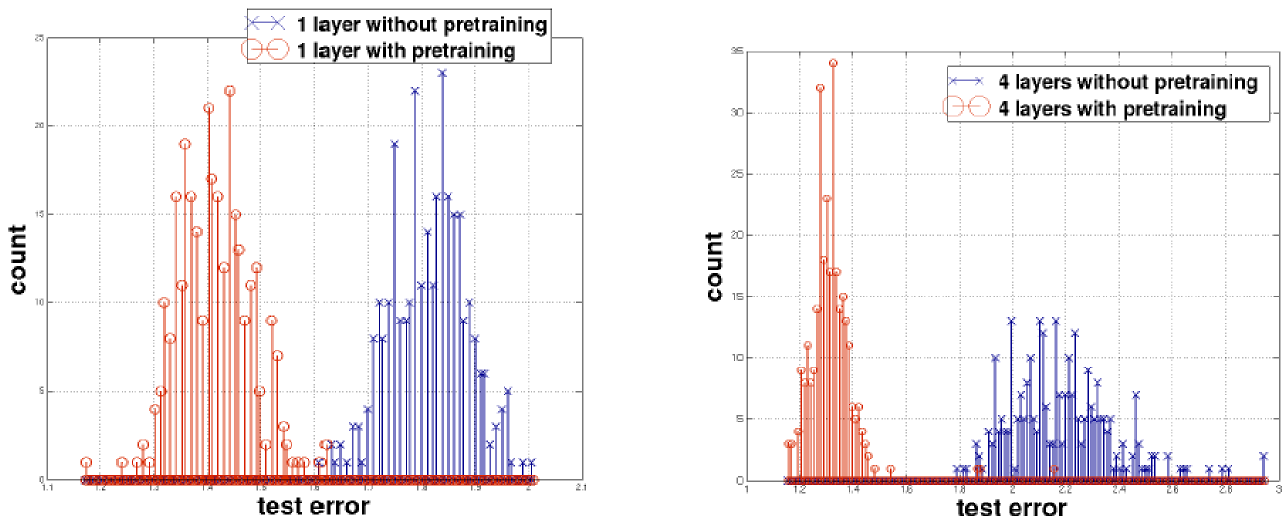


Figure 2: Histograms presenting the test errors obtained on MNIST using models trained with or without pre-training (400 different initializations each). **Left**: 1 hidden layer. **Right**: 4 hidden layers.

Journal of Machine Learning Research 11 (2010) 625-660 Submitted 8/09; Published 2/10

Why Does Unsupervised Pre-training Help Deep Learning?

Dumitru Erhan* DUMITRU.ERHAN@UMONTREAL.CA
Yoshua Bengio YOSHUA.BENGIO@UMONTREAL.CA
Aaron Courville AARON.COURVILLE@UMONTREAL.CA
Pierre-Antoine Manzagol PIERRE-ANTOINE.MANZAGOL@UMONTREAL.CA
Pascal Vincent PASCAL.VINCENT@UMONTREAL.CA
Département d'informatique et de recherche opérationnelle
Université de Montréal
2920, chemin de la Tour
Montréal, Québec, H3T 1J8, Canada

Samy Bengio BENGIO@GOOGLE.COM
Google Research
1600 Amphitheatre Parkway
Mountain View, CA, 94043, USA

Unsupervised pretraining

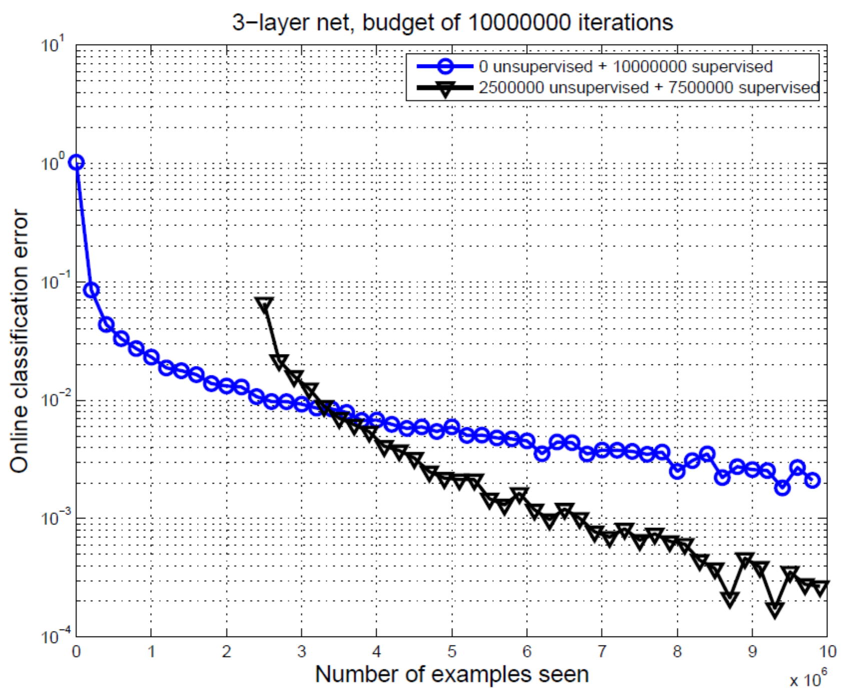


Figure 7: Deep architecture trained online with 10 million examples of digit images, either with pre-training (triangles) or without (circles). The classification error shown (vertical axis, log-scale) is computed online on the next 1000 examples, plotted against the number of examples seen from the beginning. The first 2.5 million examples are used for unsupervised pre-training (of a stack of denoising auto-encoders). The oscillations near the end are because the error rate is too close to zero, making the sampling variations appear large on the log-scale. Whereas with a very large training set regularization effects should dissipate, one can see that without pre-training, training converges to a poorer apparent local minimum: unsupervised pre-training helps to find a better minimum of the online error. Experiments performed by Dumitru Erhan.

Learning Deep Architectures for AI

Yoshua Bengio

What if things are not working?

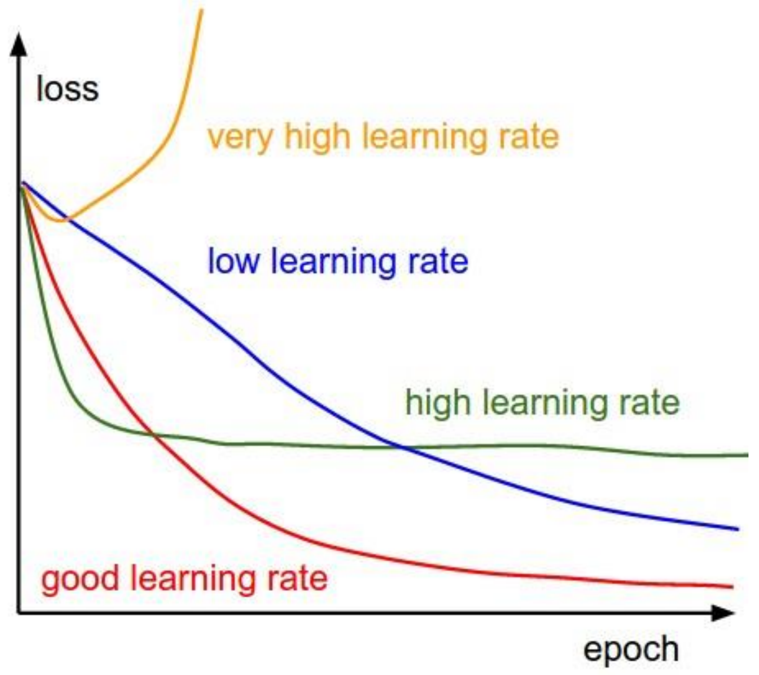
- Check your gradients by comparing them against numerical gradients
  - More on this at: http://cs231n.github.io/neural-networks-3/
  - Check whether you are using an appropriate floating point representation
    - Be aware of floating point precision/loss problems
  - Turn off drop-out and other “extra” mechanisms during gradient check
  - This can be performed only on a few dimensions

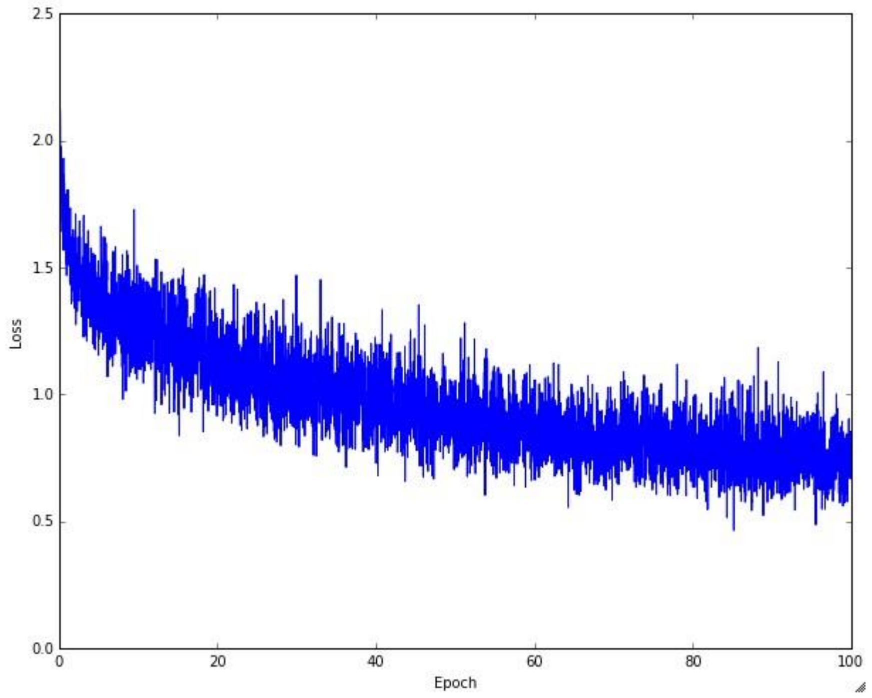
- Regularization loss may dominate the data loss
  - First disable regularization loss & make sure data loss works
  - Then add regularization loss with a big factor
  - And check the gradient in each case

What if things are not working?

- Have a feeling of the initial loss value
  - For CIFAR-10 with 10 classes: because each class has probability of 0.1, initial loss is –ln(0.1)=2.302
  - For hinge loss: since all margins are violated (since all scores are approximately zero), loss should be around 9 (+1 for each margin).
- Try to overfit on a tiny subset of the dataset
  - The cost should reach to zero if things are working properly

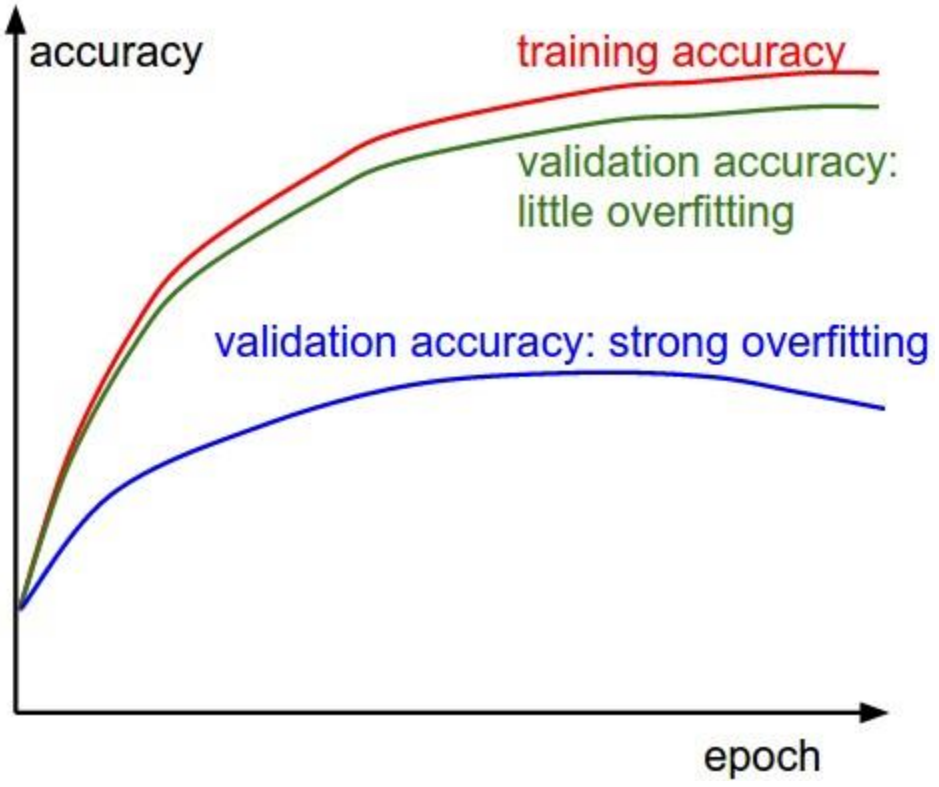
What if things are not working?





Learning rate might be too low;
Batch size might be too small

What if things are not working?

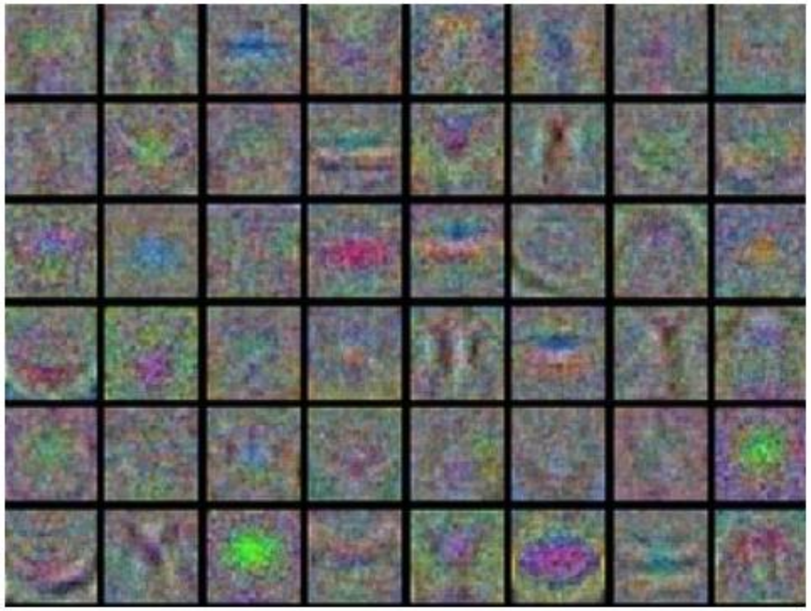
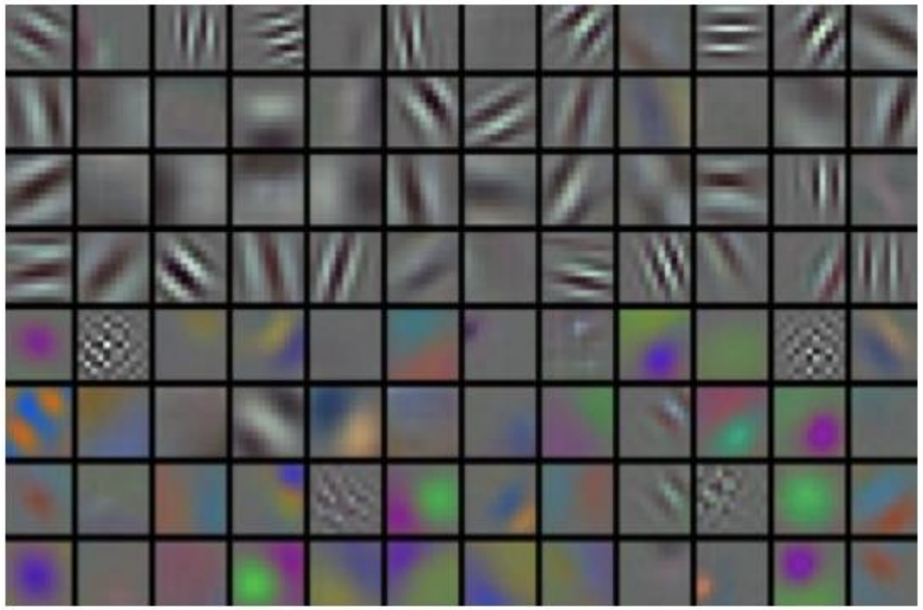


What if things are not working?

- Track the gradients and updates
  - E.g., the ratio between the norm of the update and the norm of the gradients for each weight.
  - This should be around 1e-3
    - If it is lower ➔ your learning rate might be too low
    - If it is higher ➔ your learning rate might be too high
- Plot the histogram of activations per layer
  - E.g., for tanh functions, we expect to see a diverse distribution of values between [-1,1]

What if things are not working?

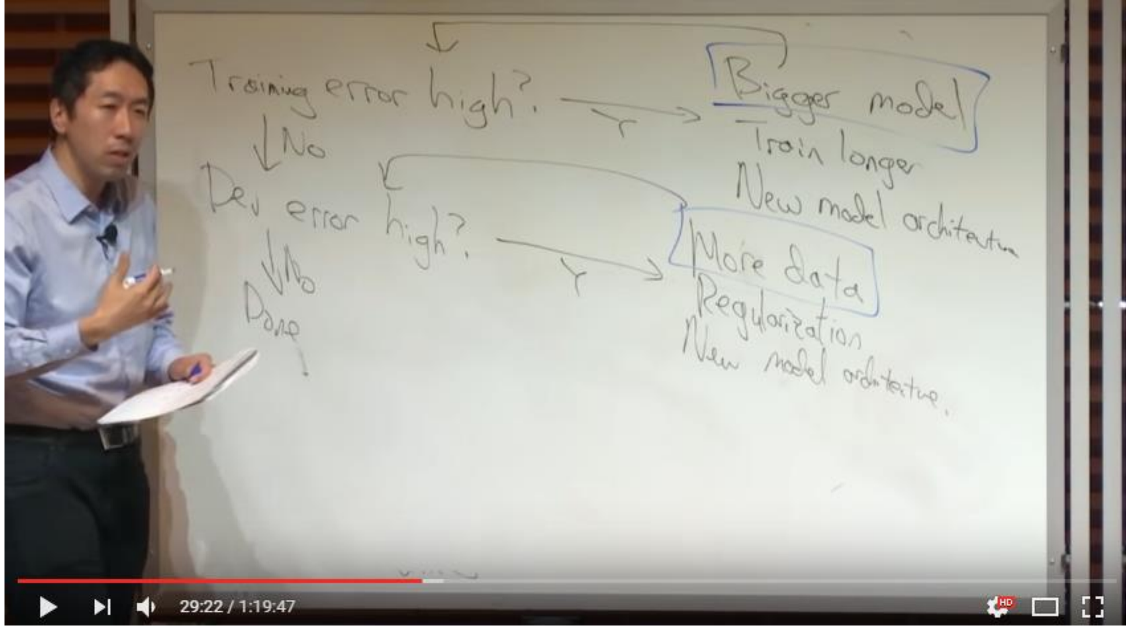
- Visualize your layers (the weights)

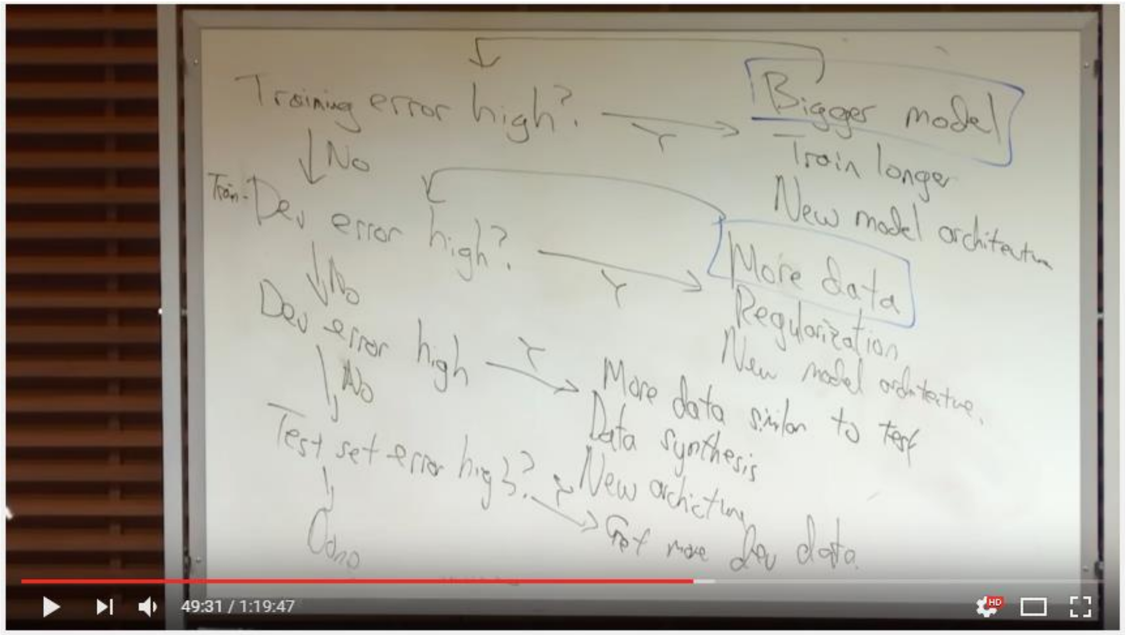



Examples of visualized weights for the first layer of a neural network. **Left**: Noisy features indicate could be a symptom: Unconverged network, improperly set learning rate, very low weight regularization penalty. **Right:** Nice, smooth, clean and diverse features are a good indication that the training is proceeding well.

Andrew Ng's suggestions

- "In DL, the coupling between bias & variance is weaker compared to other ML methods:
  - We can train a network to have high bias and variance."
- Dev(validation) and test sets should come from the same distribution. Dev&test sets are like problem specifications.
  - This requires especially attention if you have a lot of data from simulated environments etc. but little data from the real test environment.

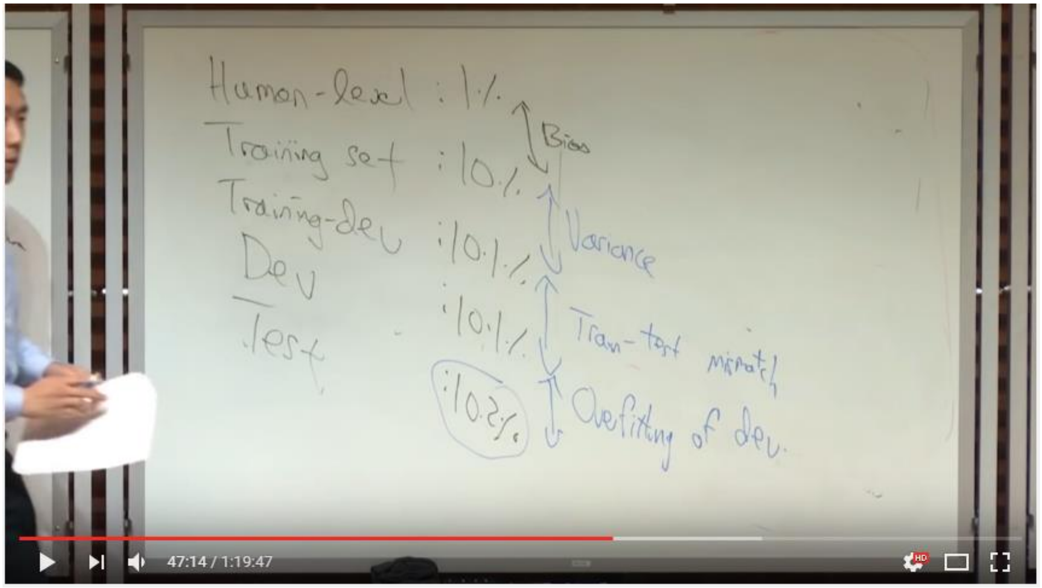


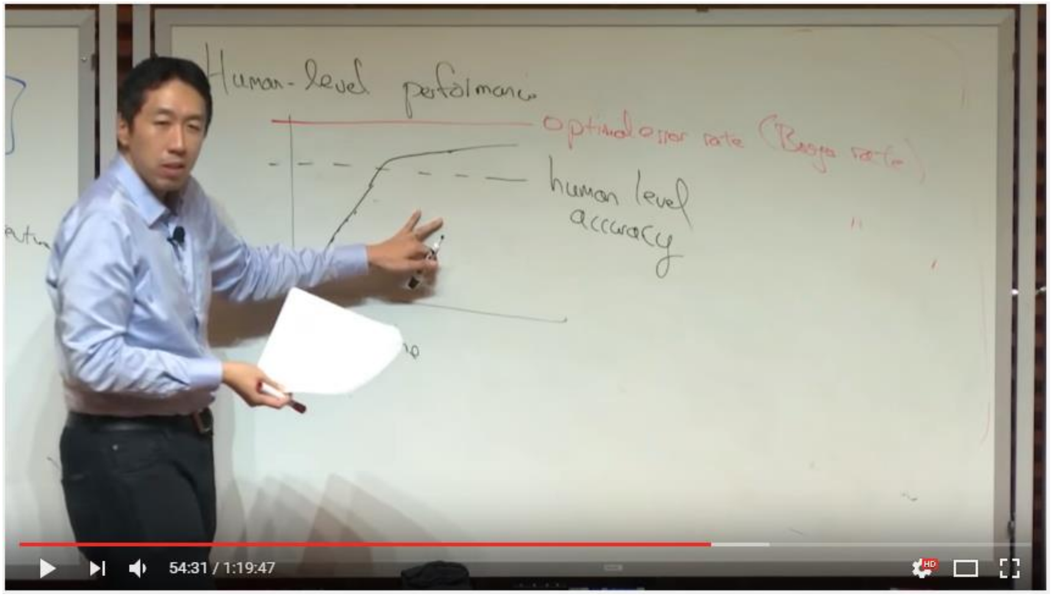


https://www.youtube.com/watch?v=F1ka6a13S9I

Andrew Ng's suggestions

- Knowing the human performance level gives information about the problem of your network: If training error is far from human performance, then there is a bias error. If they are close but validation has more error (compared to the diff between human and training error), then there is variance problem.
- After surpassing human level, performance increases only very slowly very difficult-ly. One reason: There is not much space for improvement (only tiny little details). Problem gets much harder. Another reason: We get labels from humans.



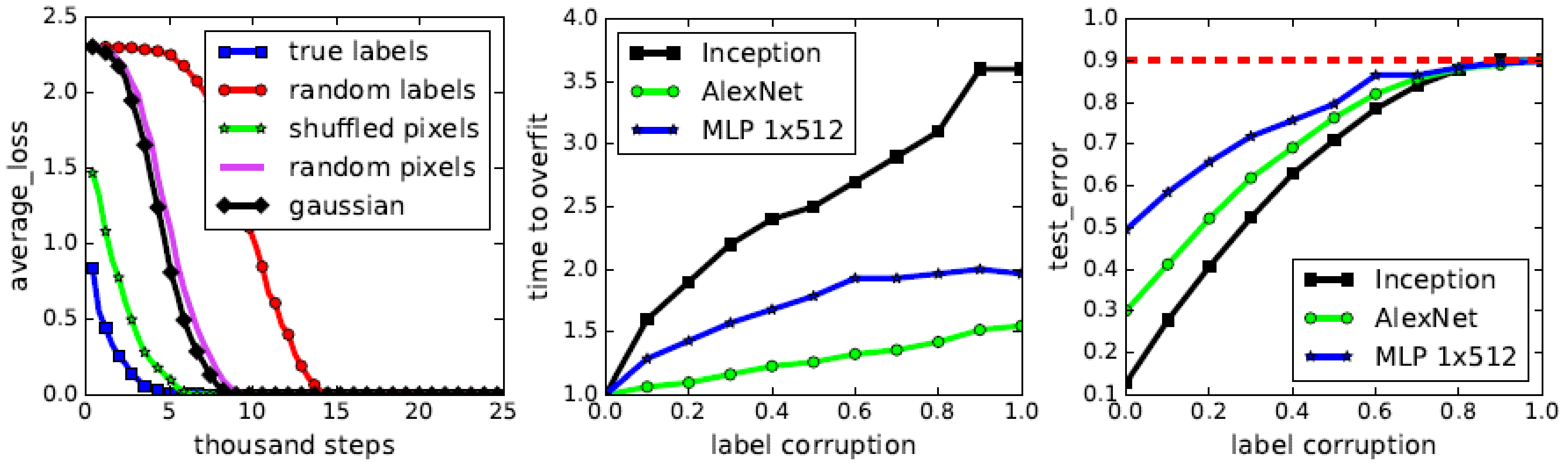


https://www.youtube.com/watch?v=F1ka6a13S9I

What is best then?

- Which algorithm to choose?
  - No answer yet
  - See Tom Schaul (2014)
  - RMSprop and AdaDelta seems to be slightly favorable; however, no best algorithm
- SGD, SGD+momentum, RMSprop, RMSprop+momentum, AdaDelta and Adam are the most widely used ones

Luckily, deep networks are very powerful



(a) learning curves (b) convergence slowdown (c) generalization error growth

Figure 1: Fitting random labels and random pixels on CIFAR10. (a) shows the training loss of various experiment settings decaying with the training steps. (b) shows the relative convergence time with different label corruption ratio. (c) shows the test error (also the generalization error since training error is 0) under different label corruptions.

Regularization is turned off in the experiments. When you turn on regularization, the networks perform worse.

UNDERSTANDING DEEP LEARNING REQUIRES RE-THINKING GENERALIZATION

Chiyuan Zhang*
Massachusetts Institute of Technology
chiyuan@mit.edu

Samy Bengio
Google Brain
bengio@google.com

Moritz Hardt
Google Brain
mrtz@google.com

Benjamin Recht†
University of California, Berkeley
recht@berkeley.edu

Oriol Vinyals
Google DeepMind
vinyals@google.com

Concluding remarks for the first part

- Loss functions
- Gradients of loss functions for minimizing them
  - All operations in the network should be differentiable
- Gradient descent and its variants
- Initialization, normalization, adaptive learning rate, …
- Overall, you have learned most of the tools you will use in the rest of the course.