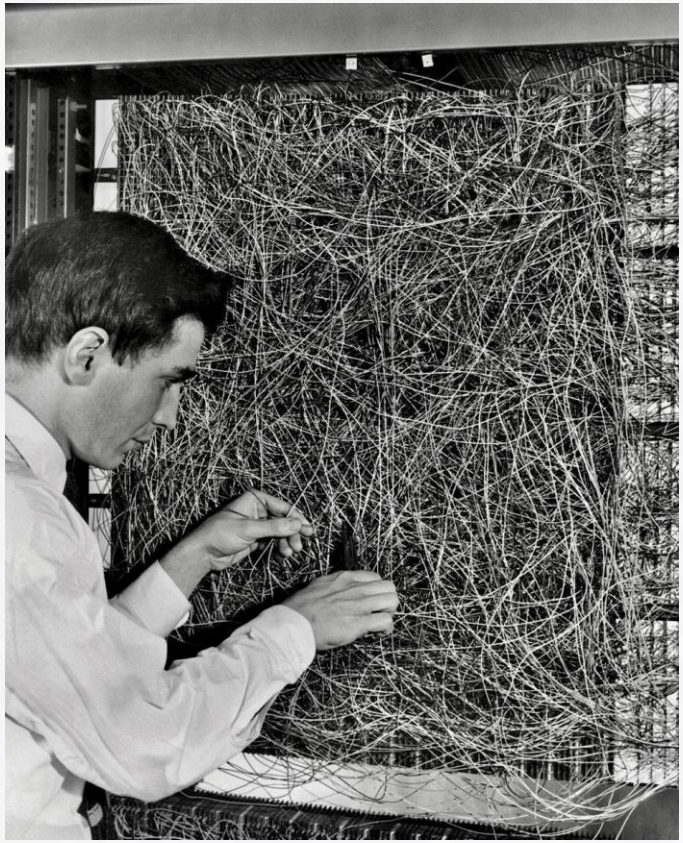
Neural Networks, Backpropagation and Deep Learning

CS 410/510: CV & DL

Outline



- Historical Notes
- UAT
- Gradient Descent
- Backpropagation & Computational Graphs
- Automatic Differentiation
- Activations, Weight Initialization
- Deep Learning Challenges
- Data Augmentation, Feature Pre-Processing
- SGD, Momentum, AdaGrad, Adam, Second-Order Methods
- Backpropagation Derivation

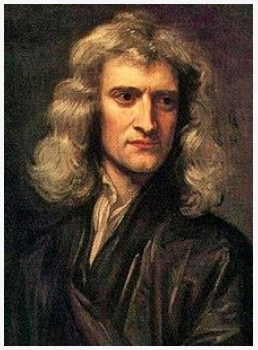
Historical Notes

• Feedforward networks can be seen as efficient non-linear function approximators based on using gradient descent to minimize the error in a function approximation.

•As such, the modern feedforward NN is the culmination of centuries of progress on **the general function approximation task**.

• The *chain rule* underlying backprop was invented by Leibniz (1796), and due naturally to foundations also laid by Newton.

• Calculus and algebra have been used to solve optimization problems in closed form since their inception, but *gradient descent* was not introduced as a technique for iteratively approximating the solution to optimization problems until 19C (Cauchy, 1847).



Newton

Leibniz

Cauchy

Al-Khwarizmi

Galois

$$\frac{\partial E}{\partial w_{ij}} = \delta_j * x_i$$

$$o_j' = o_j * (1 - o_j)$$

$$\varphi_j' = (1 - \varphi_j) * (1 + \varphi_j)$$

$$\varphi_j' = \varphi_j * (1 - \varphi_j)$$

$$e_j = (o_j - t_j)$$

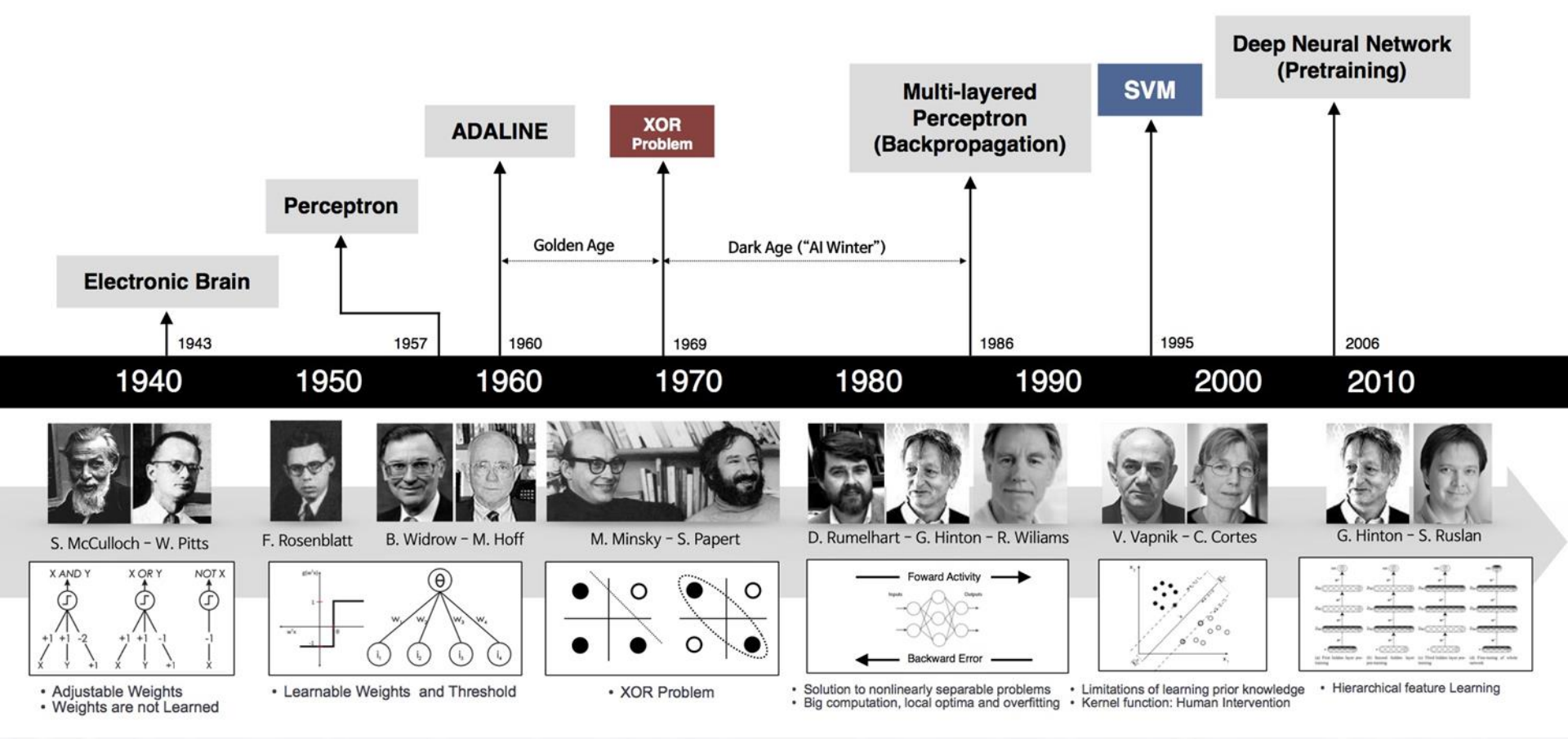
$$\delta_j = e_j * o_j'$$

$$\delta_j = (\sum \delta_j w_j) * \varphi_j'$$

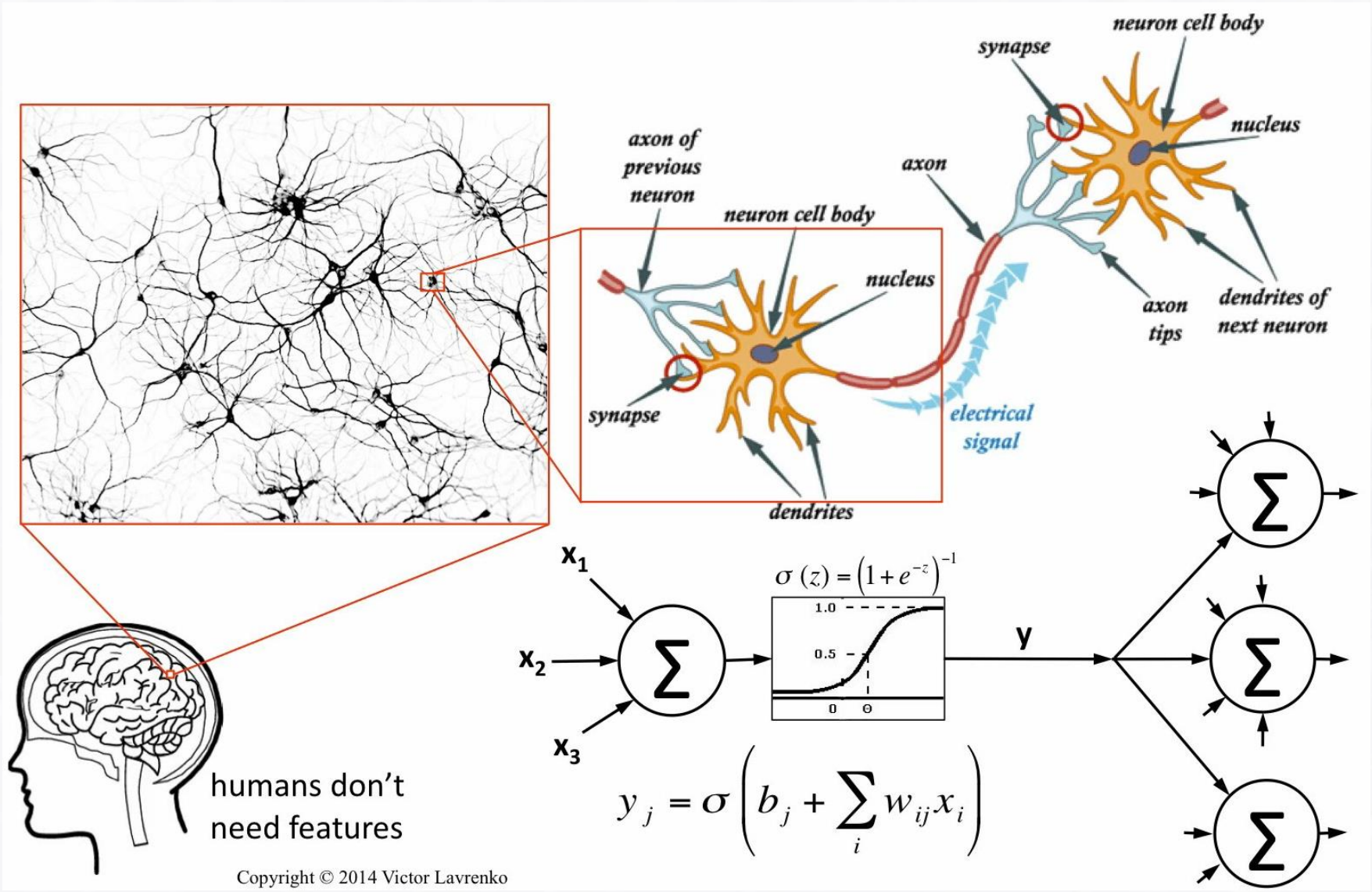
$$\Delta w_{ij} = \alpha * \frac{\partial E}{\partial w_{ij}}$$

$$w_{ij}' = w_{ij} + \Delta w_{ij}$$

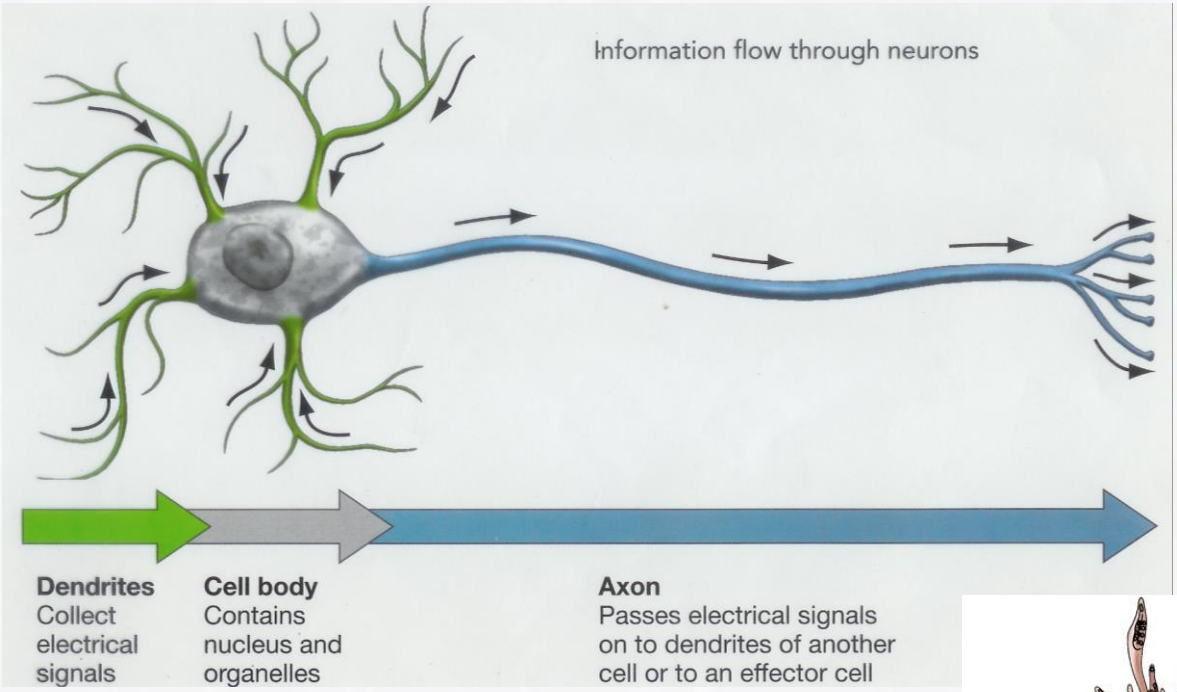
Historical Notes

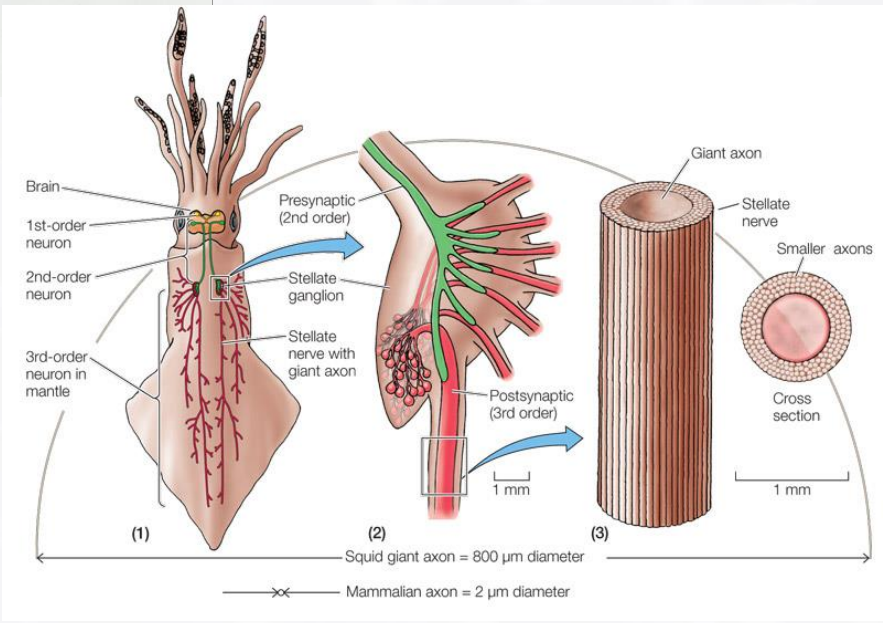


Neurons & the Brain



Neurons & the Brain



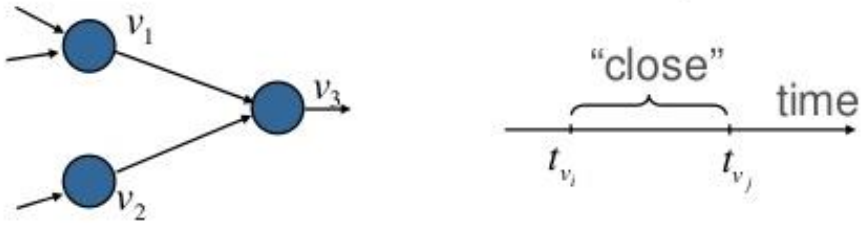


Hebb's Postulate

"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."

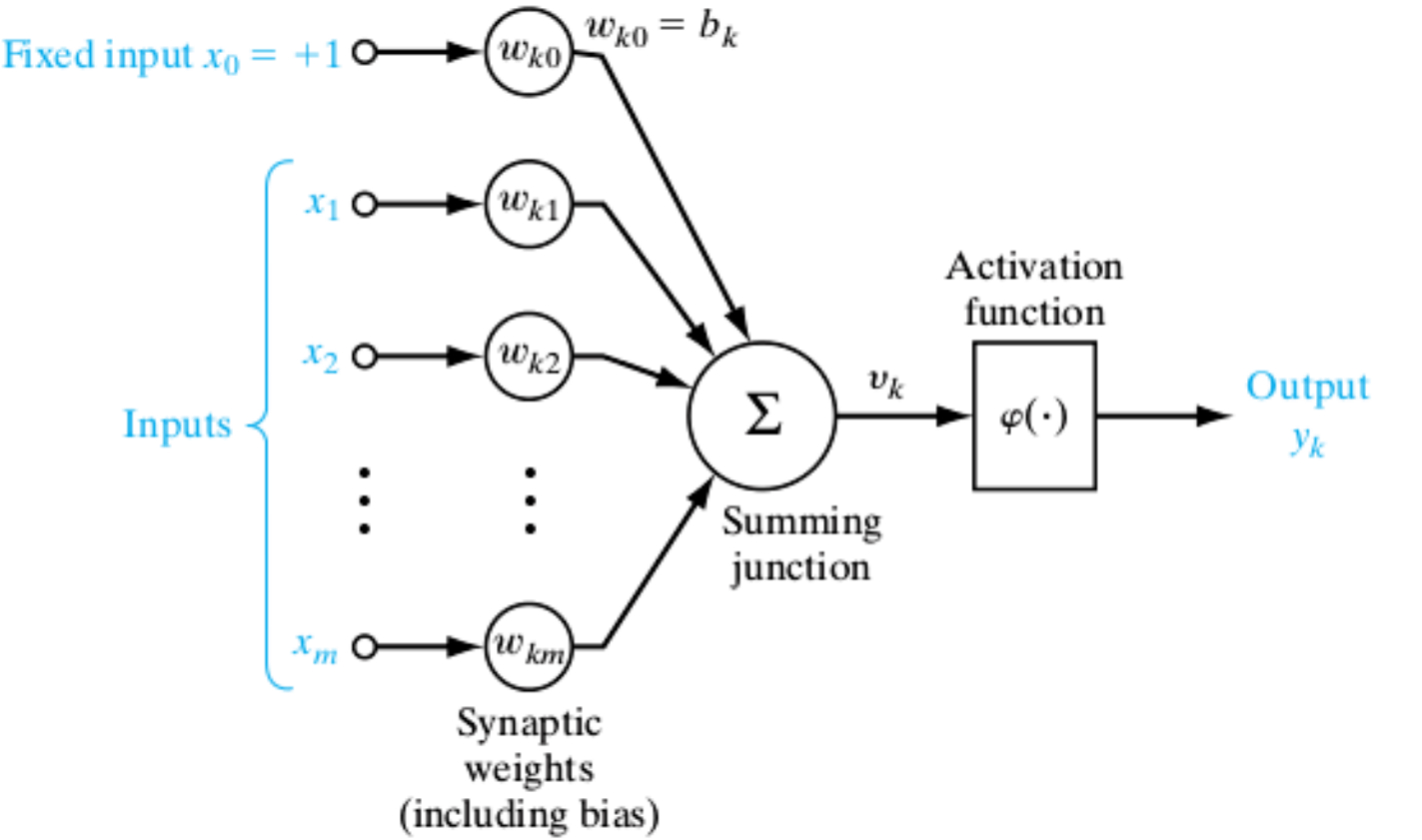
- In other words: if two neurons fire "close in time" then strength of synaptic connection between them increases.

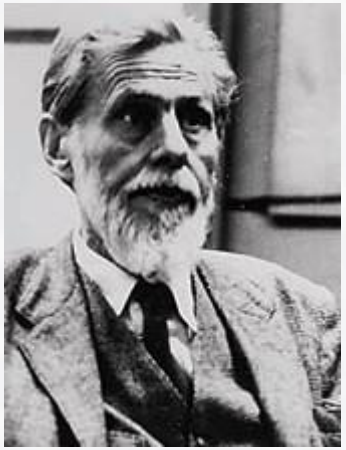
$$\Delta w_{ij}(t) = \eta v_i v_j g(t_{v_i}, t_{v_j})$$

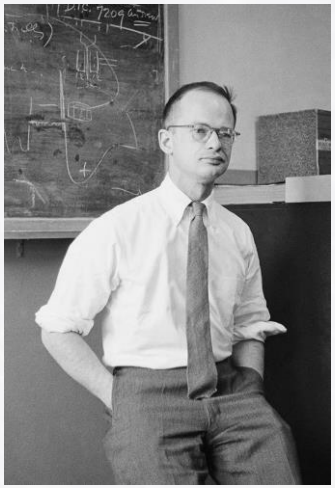


- Weights reflect correlation between firing events.

McCulloch & Pitts Neuron Model (1943)







(3) Components:

(1) Set of weighted inputs $\{w_i\}$ that correspond to synapses

(2) An "adder" that sums the input signals (equivalent to membrane of the cell that collects the electrical charge)

(3) An activation function (initially a threshold function) that decides whether the neuron fires ("spikes") for the current inputs.

McCulloch & Pitts Neuron Model (1943)

Limitations & Deviations of the M-P Neuron Model:

- Summing is linear.
- No explicit model of "spike trains" (sequence of pulses that encodes information in biological neuron).
- Threshold value is usually fixed.
- Sequential updating implicit (biological neurons usually update themselves asynchronously)
- Weights can be positive (excitatory) or negative (inhibitory); biological neurons do not change in this way.
- Real neurons can have synapses that link back to themselves (e.g. feedback loop) – see RNNs (recurrent neural networks).
- Other biological aspects ignored: chemical concentrations, refractory periods, etc.

Historical Notes

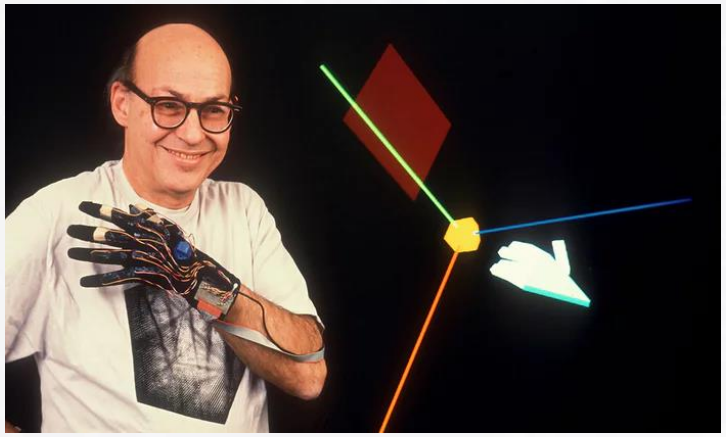
• Beginning in the 1940s, these function approximation techniques were used to motivate ML models such as the **percepton**. However, the earliest models were based on linear models.

• In the 1960s **Rosenblatt** proved that the perceptron learning rule converges to correct weights in a finite number of steps, provided the training examples are linearly separable.

•Critics including Marvin Minsky point out several of the flaws of the linear model family, such as its inability to learn the XOR function, which led to a backlash against the entire NN approach.

• Learning non-linear functions required the development of a MLP (multi-layer perceptron) and a means of computing the gradient through such a model. Efficient applications of the chain rule based on DP (dynamic programming) began to appear in the 1960s and 1970s.

Rosenblatt



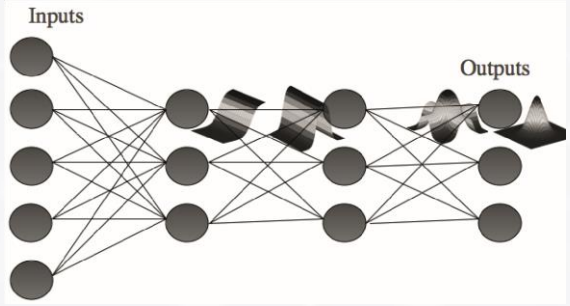
Minsky

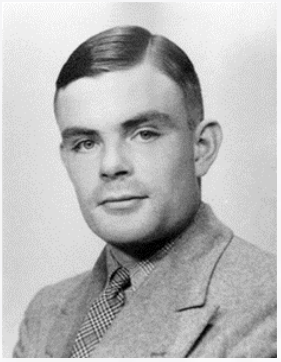
Historical Notes

- 1969: **Minsky and Papert** proved that perceptrons cannot represent non-linearly separable target functions.

- However, they showed that adding a fully connected hidden layer makes the network more powerful.
  - i.e., Multi-layer neural networks can represent non-linear decision surfaces.
- Later it was shown that by using continuous activation functions (rather than thresholds), a fully connected network with a single hidden layer can in principle represent any function.
- 1986: "rediscovery" of backprop algorithm: **Hinton** et al.

- The **Universal Approximation Theorem (1989)** states that <u>one hidden layer is sufficient to approximate any function to arbitrary accuracy</u> with a NN. (we say: "NNs are universal function approximators"); RNNs are *Turing Complete*.





Universal Approximation Properties

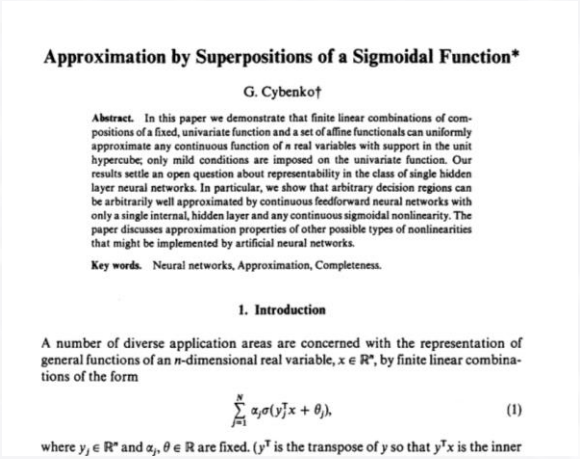
- A linear model, mapping from features to outputs via matrix multiplication, can by definition represent only linear functions. It has the advantage of being easy to train because many loss functions result in convex optimization problems when applied to linear models.

- The **universal approximation theorem** (UAT) states that a feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function can approximate any *Borel measurable* (e.g. a continuous function on a closed and bounded subset of R^n) function from one finite-dimensional space to another with any desired non-zero amount of error, provided the network is given enough hidden units.

- The UAT states that regardless of what function we are trying to learn, we know that a sufficiently large MLP will be able to represent this function. We are not guaranteed, however, that the training algorithm will be able to learn the function.

Universal Approximation Properties

- The UAT states that regardless of what function we are trying to learn, we know that a sufficiently large MLP will be able to represent this function. We are not guaranteed, however, that the training algorithm will be able to learn the function.
- Cybneko (1989) proved UAT for sigmoid activations.



Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

1. Introduction

A number of diverse application areas are concerned with the representation of general functions of an n-dimensional real variable, $x \in \mathbb{R}^n$, by finite linear combinations of the form

$$\sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j), \qquad (1)$$

where $y_j \in \mathbb{R}^n$ and $\alpha_j, \theta \in \mathbb{R}$ are fixed. (y^T is the transpose of y so that $y^T x$ is the inner

- Hornik (1991) proved that the network itself gives rise to universal approximation property -- not specific choice of activation (so no long as activation is non-linear).
- Classical UAT related to depth-bounded networks (e.g. depth-2). Lu et al. (2017) proved UAT for width-bound NNs (width: n+4 with RELU, where n is the input dimension).

Universal Approximation Properties

- Even if the MLP is able to represent the function. Learning can fail for (2) different reasons:

(1) The optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function.

(2) The training algorithm might choose the wrong function as a result of overfitting.

Universal Approximation Properties

- Feedforward networks provide a universal system for representing functions in the sense that, given a function, there exists a feedforward network that approximates the function; there is **no universal procedure** for examining a training set of specific examples and choosing a function that will generalize to points not in the training set.

Universal Approximation Properties

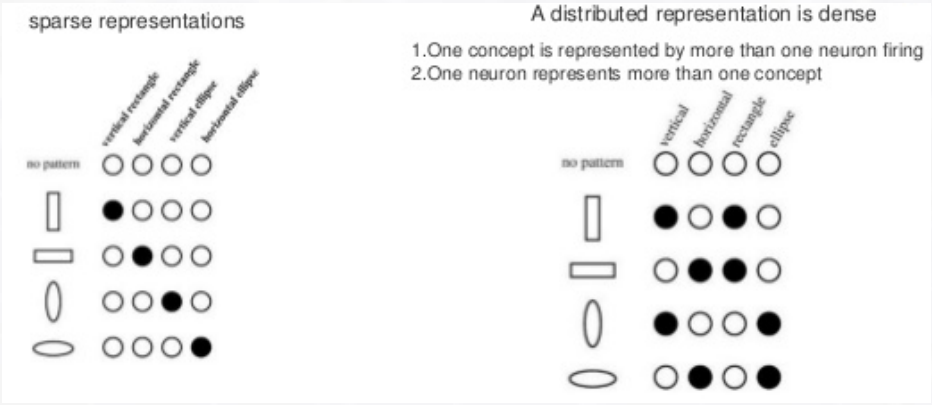
- Feedforward networks provide a universal system for representing functions in the sense that, given a function, there exists a feedforward network that approximates the function; there is **no universal procedure** for examining a training set of specific examples and choosing a function that will generalize to points not in the training set.

*Note also that the theorem does not prescribe the size of the network (some bounds can be approximated); unfortunately, in the worst case, an exponential number of hidden units may be required.

*Recall that any time we choose a specific ML algorithm, we are implicitly imposing some set of prior beliefs we have about what kind of function the algorithm should learn (this is the so-called **inductive bias** of the learning algorithm); choosing a deep model generally indicates that we want to learn a composition of several simpler functions.

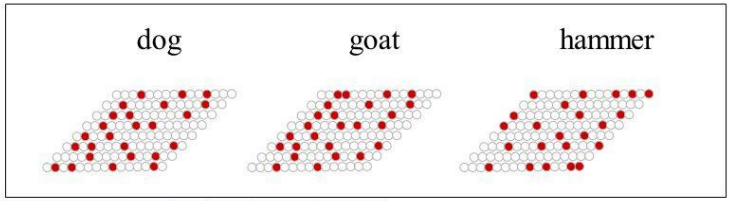
Historical Notes

•The "rediscovery" of the backpropagation algorithm (Hinton & Rumelhardt) ushered in a very active period of research for MLPs. In particular, "**connectionism**" took root in the ML community, which placed emphasis on connections between neurons as the locus of learning and memory (cf. **distributed representation**: each concept is represented by many neurons, each neuron participates in the representation of many concepts.

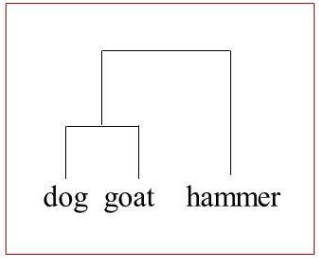


http://www.cs.toronto.edu/~bonner/courses/2014s/csc321/lectures/lec5.pdf

Distributed Representations in the Brain: Overlapping Patterns for Related Concepts (Kiani et al, 2007)



- Many hundreds of single neurons recorded in monkey IT.
- 1000 different photographs were presented twice each to each neuron.
- Hierarchical clustering based on the distributed representation of each picture:
  - The pattern of activation over all the neurons



http://www.jneurosci.org/content/35/13/5180

Historical Notes

• Following the success of backprop, NN research gained popularity and reached a peak in the early 1990s. Afterwards, other ML techniques became more popular until the modern deep learning renaissance that began in 2006.

• The core ideas behind modern feedforward nets have not changed substantially since the 1980s. The same backprop algorithm and the same approaches to gradient descent are still in use. Most of the improvement in NN performance from 1986-2018 **can be attributed to two factors**:

Historical Notes

• Following the success of backprop, NN research gained popularity and reached an (initial) apex in the early 1990s.Afterwards, other ML techniques became more popular until the modern deep learning renaissance that began in 2006.

• The core ideas behind modern feedforward nets have not changed substantially since the 1980s. The same backprop algorithm and the same approaches to gradient descent are still in use. Most of the improvement in NN performance from 1986-2018 **can be attributed to two factors**:

(1) Larger datasets have reduced the degree to which statistical generalization is a challenge for NNs.

(2) NNs have come much larger because of more powerful computer (including the use of GPUs) and better software infrastructure.

Historical Notes

• Nevertheless, a number of algorithmic changes have also contributed to subsequent improvements in the performance of NNs.

• One of these algorithmic changes was the replacement of mean squared error (MSE) with the **cross-entropy family of loss functions**. MSE was popular in the 1980s and 1990s but was gradually replaced by cross-entropy losses and the principles of MLE as ideas spread between the statistics community and ML community.

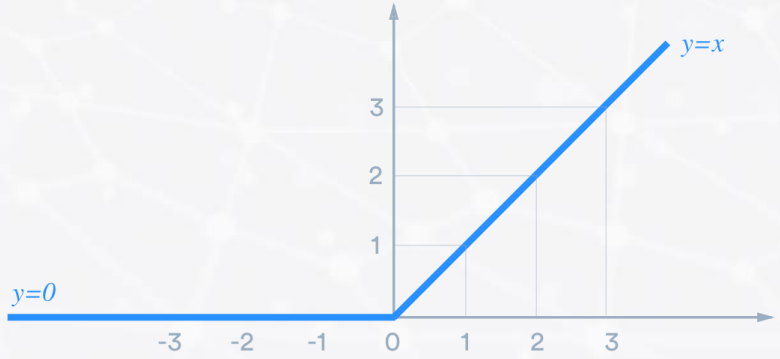
• The use of cross-entropy losses greatly improved the performance of models with sigmoid and softmax outputs, which had previously suffered from saturation and slow learning when using MSE.

Historical Notes

• The other major algorithmic change that has greatly improved the performance of feedforward networks was the replacement of sigmoid hidden units with **piecewise linear hidden units**, such as *rectified linear units* (**RELU**s). Rectification using the $\max\{0,z\}$ function was introduced in early NN models.

• As of the early 2000s, rectified linear units were avoided due to the belief that activation functions with non-differentiable points must be avoided.

• For small datasets, Jarrett et al. (2009) observed that using rectifying non-linearities is even more important than learning the weights of the hidden layers. Random weights are sufficient to propagate useful information through a rectified linear network, enabling the classifier layer at the top to learn how to map different feature vectors to class identities.



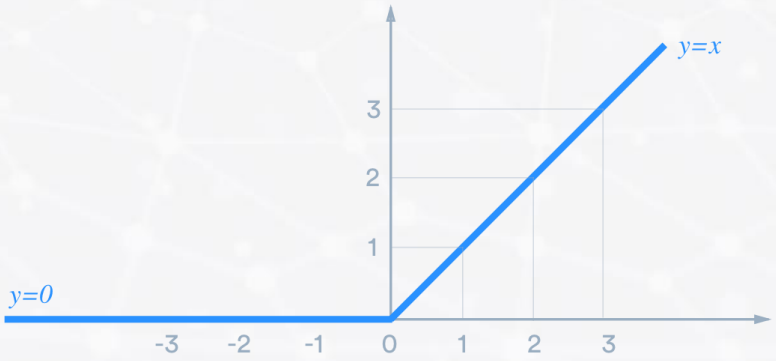
Historical Notes

• RELUs are also of historical interest because they show that neuroscience has continued to have an influence on the development of deep learning algorithms. Glorot et al. (2011) motivated RELUs from biological considerations. The half-rectifying non-linearity was intended to captured these properties of biological neurons:

(1) For some inputs, biological neurons are completely inactive.

(2) For some inputs, a biological neuron's output is proportional to its inputs.

(3) Most of the time, biological neurons operate in the regime where they are inactive (i.e. they should have sparse activations).

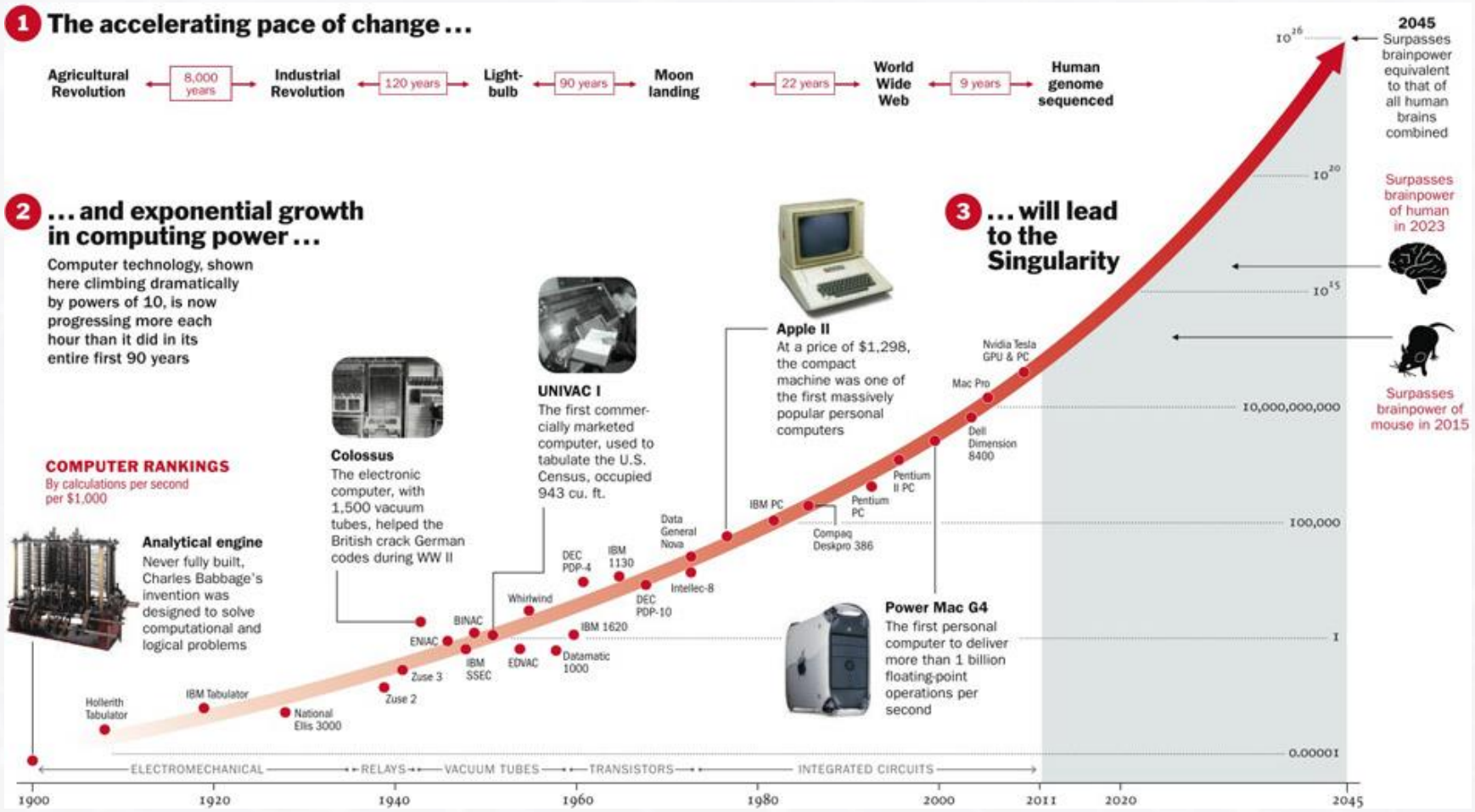


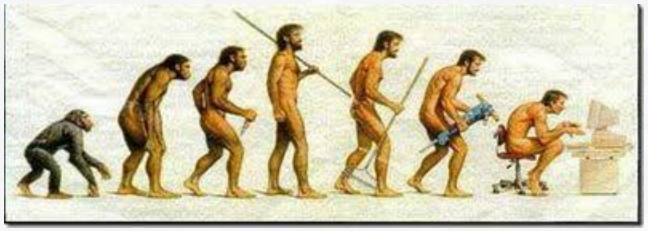
Neurons & the Brain

- Human brain contains ~10^{11} neurons
- Each individual neuron connects to ~10^{4} neuron
- ~10^{14} total synapses!

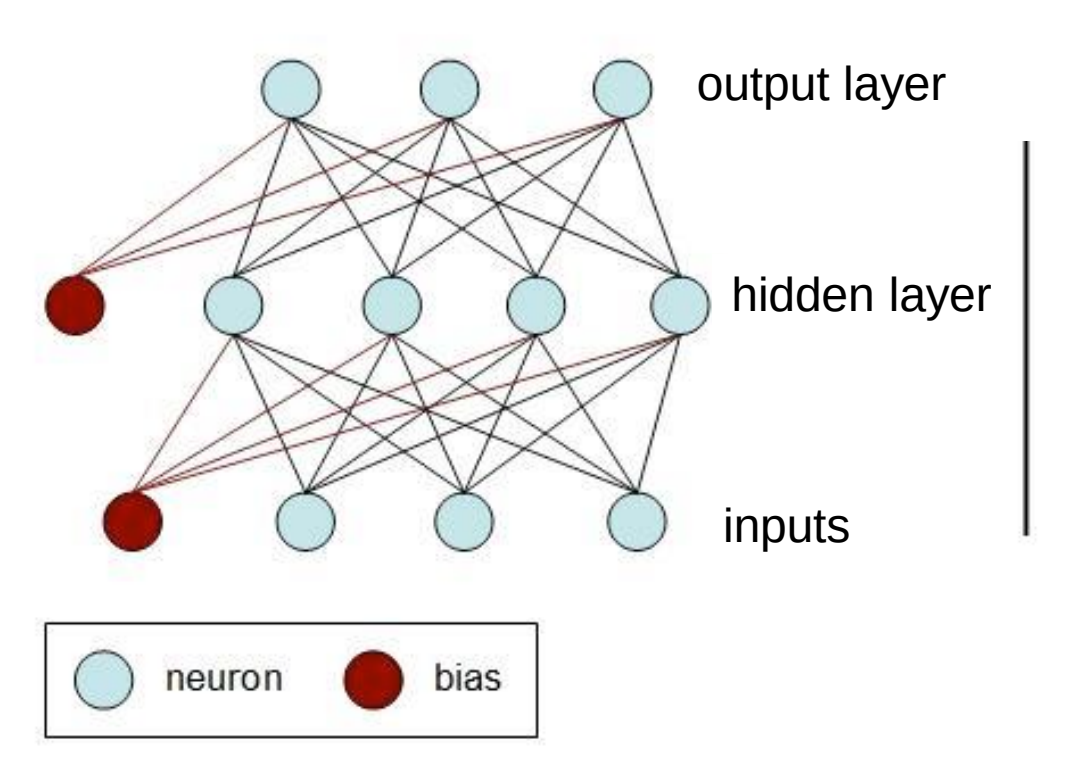
	Brain	Computer
Number of Processing Units	$\approx 10^{11}$	$\approx 10^{9}$
Type of Processing Units	Neurons	Transistors
Form of Calculation	Massively Parallel	Generally Serial
Data Storage	Associative	Address-based
Response Time	$\approx 10^{-3}$s	$\approx 10^{-9}$s
Processing Speed	Very Variable	Fixed
Potential Processing Speed	$\approx 10^{13}$ FLOPS [14]	$\approx 10^{18}$ FLOPS
Real Processing Speed	$\approx 10^{12}$ FLOPS	$\approx 10^{10}$ FLOPS
Resilience	Very High	Almost None
Power Consumption per Day	20W	300W [15]

Historical Notes





A "two"-layer neural network



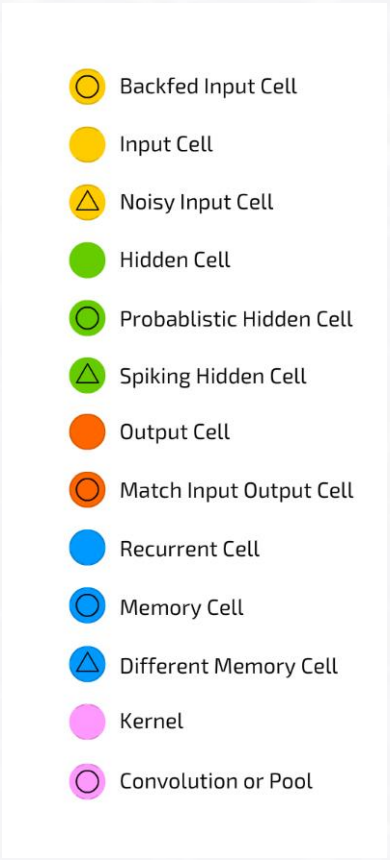
(activation represents classification)

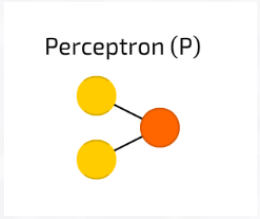
(internal representation)

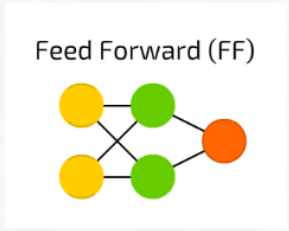
(activations represent feature vector for one training example)

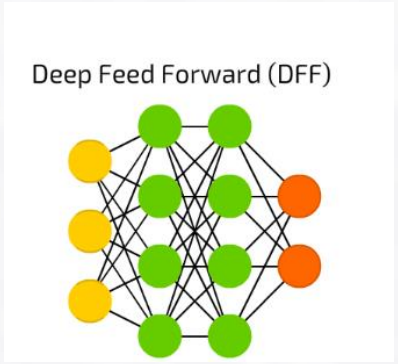
•**Input layer**—It contains those units (artificial neurons) which receive input from the outside world on which network will learn, recognize about or otherwise process.
•**Output layer**—It contains units that respond to the information about how it's learned any task.
•**Hidden layer**—These units are in between input and output layers. The job of hidden layer is to transform the input into something that output unit can use in some way.
Most neural networks are fully connected that means to say each hidden neuron is fully connected to the every neuron in its previous layer(input) and to the next layer (output) layer.

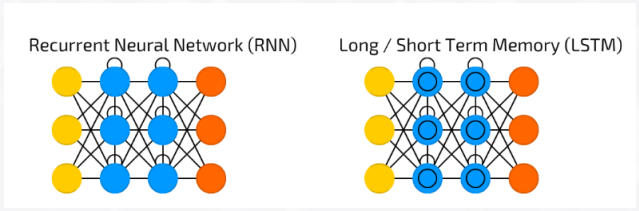
A Neural Network "Zoo"

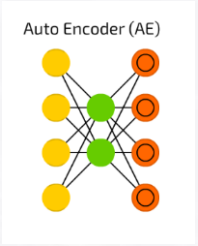


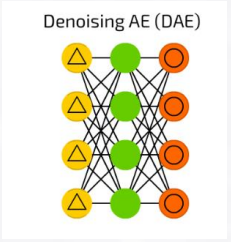


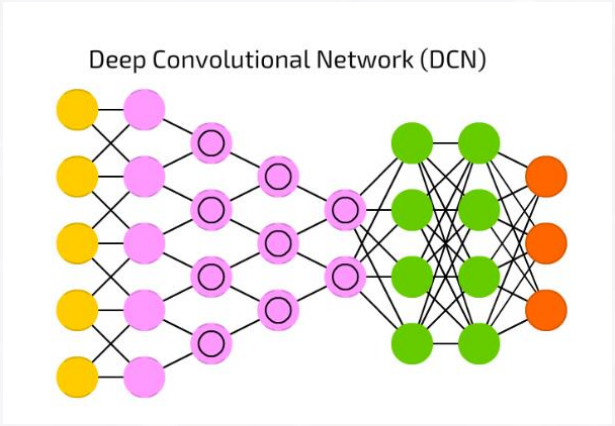




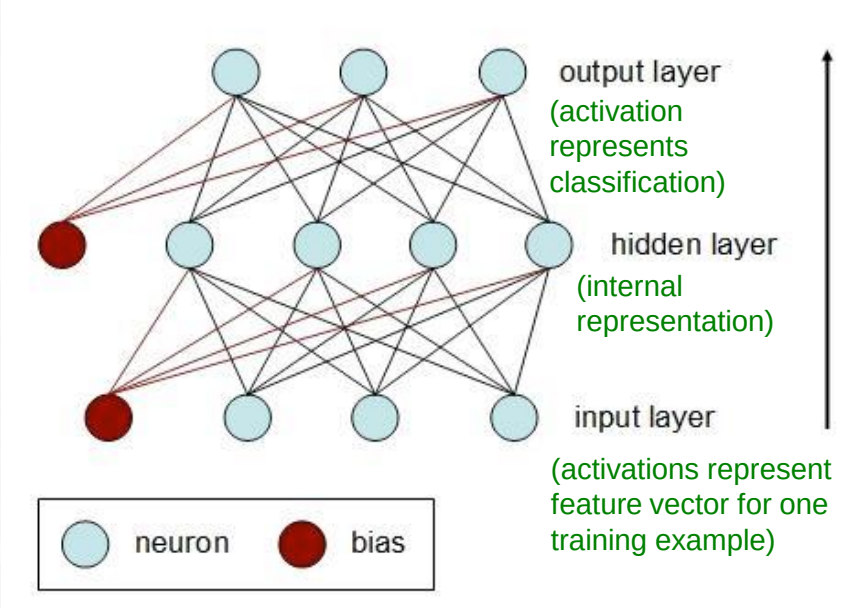








Neural network notation



x_i : activation of **input** node *i*.

h_j : activation of **hidden** node *j*.

o_k : activation of **output** node *k*.

w_{ji} : weight from node *i* to node *j*.

σ : "sigmoid function".

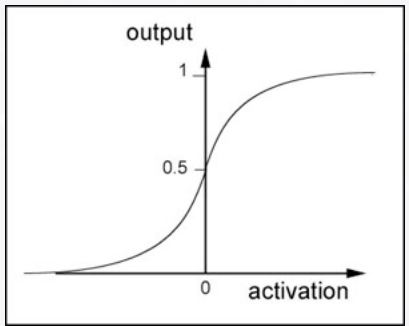
For each node *j* in hidden layer,

$$h_j = s\left(\sum_{i \in \text{input layer}} w_{ji} x_i + w_{j0}\right)$$

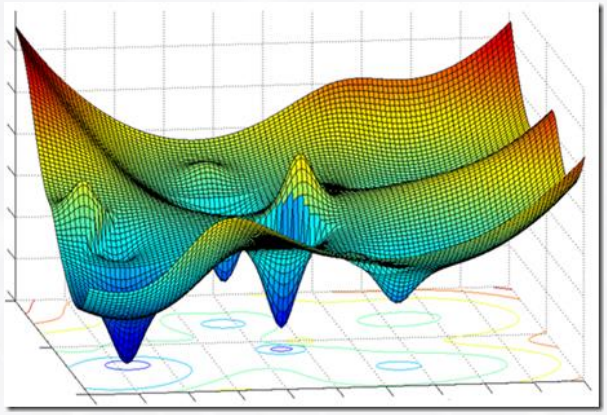
For each node *k* in output layer,

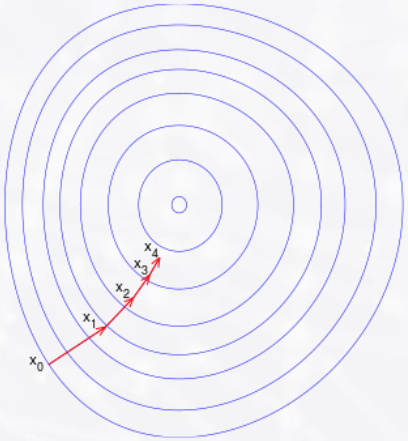
$$o_k = s\left(\sum_{j \in \text{hidden layer}} w_{kj} h_j + w_{k0}\right)$$

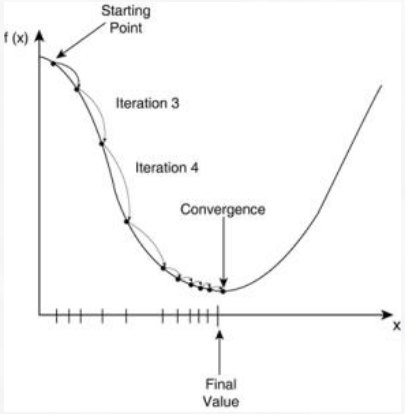
Sigmoid function:

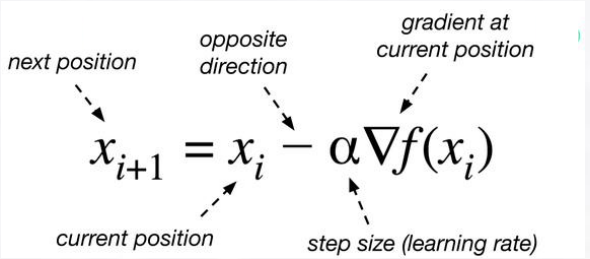


Gradient Descent







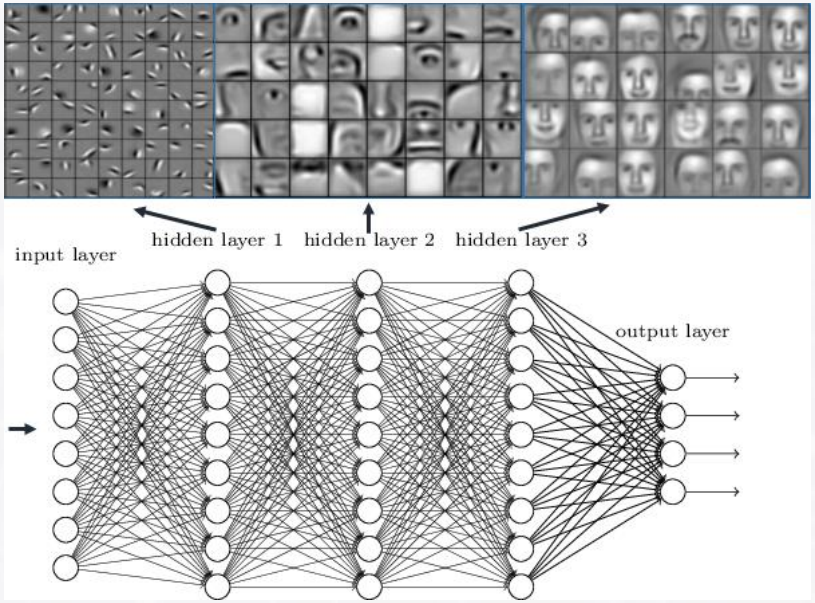


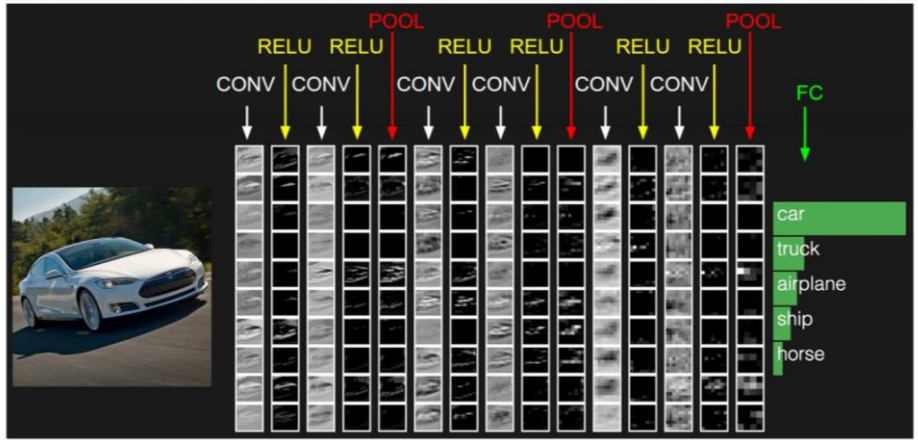
(*) Backpropagation is one particular instance of a larger paradigm of optimization algorithms know as **Gradient Descent** (also called "hill climbing").

(*) There exists a large array of nuanced methodologies for efficiently training NNs (particularly DNNs), including the use of **regularization**, **momentum**, **dropout**, **batch normalization**, pre-training regimes, initialization processes, etc.

(*) Traditionally, the backpropagation algorithm has been used to efficiently train a NN; more recently the **Adam stochastic optimization method** (2014) has eclipsed backpropagation in practice: https://arxiv.org/abs/1412.6980

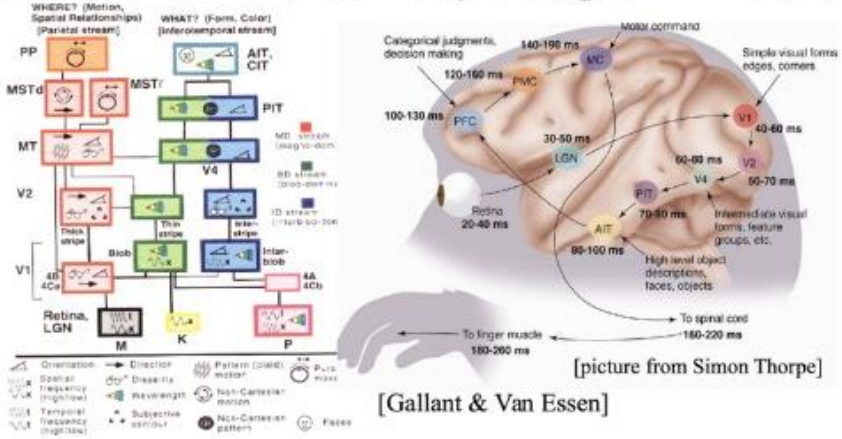
DNNs Learn Hierarchical Feature Representations

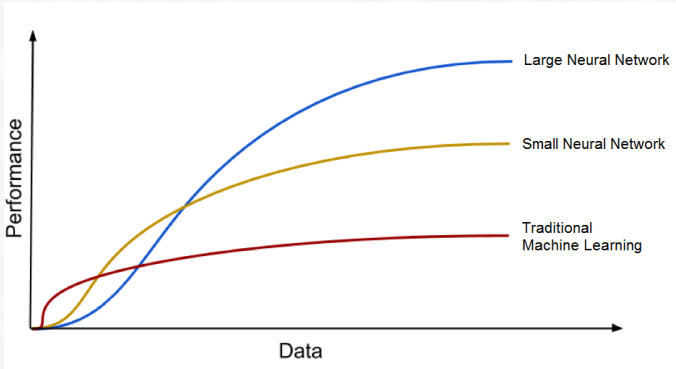




Deep Learning = Brain "inspired"

Audio/Visual Cortex has multiple stages == Hierarchical





Backpropagation

• Backpropagation is the engine behind most (but not all) deep learning training algorithms.

Backpropagation consists of two alternating steps:

(1) Forward step: Propagate the input vector through the network (this consists primarily of dot product operations followed by non-linear activation operations).

(2) Backward step: Using the output computed in step (1); the "error" (according to some prescribed *loss function*) is propagated backward through the network. The backward step assigns an attribution value to the edges in the network based on the loss calculated.

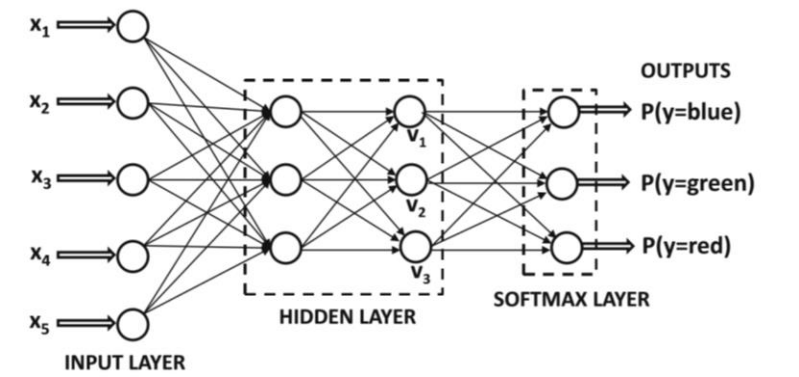
Backpropagation

• Backpropagation is the engine behind most (but not all) deep learning training algorithms.

Backpropagation consists of two alternating steps:

(1) Forward step: Propagate the input vector through the network (this consists primarily of dot product operations followed by non-linear activation operations).

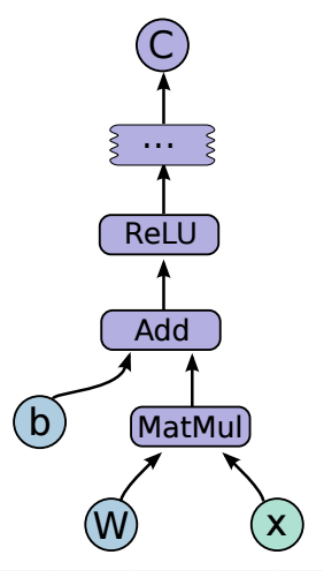
(2) Backward step: Using the output computed in step (1); the "error" (according to some prescribed *loss function*) is propagated backward through the network. The backward step assigns an attribution value to the edges in the network based on the loss calculated.



(*) For an alternative to backpropagation methods, see, for example: ELM "**extreme learning machines**" methodologies (which are considered controversial in mainstream ML. https://www.researchgate.net/publication/264273594_Extreme_learning_machines

Backpropagation: Computational Graphs

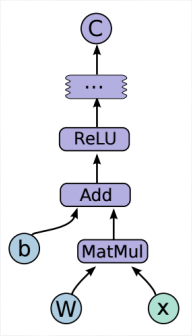
• A neural network (NN can be modeled as a **computational graph,** in which a unit of computation is the neuron.



• NNs are <u>fundamentally more powerful than their building blocks</u> because the parameters of these models are learned jointly to create a highly optimized composition function of these models. In addition, the non-linear activations between the different layers enhance the expressive power of the network.

Backpropagation: Computational Graphs

- A neural network (NN) is a **computational graph,** in which a unit of computation is the neuron.



- A multi-layer NN evaluates **compositions of functions** computed at individual nodes. For instance, a path of length 2 in the NN in which the activation function g(·) follows a basic *affine transformation* (i.e., matrix multiplication plus a "bias" shift) results in the composition:

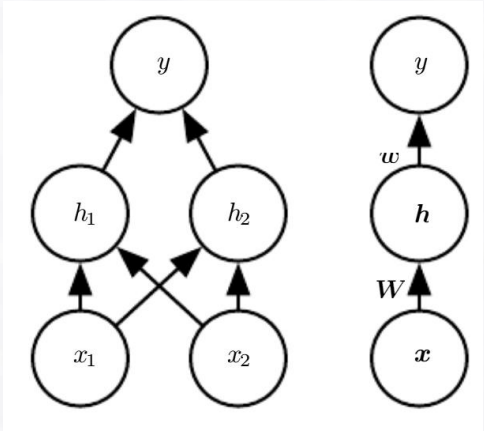
$$\mathbf{h} = g\left(\mathbf{W}^T\mathbf{x} + \mathbf{b}\right)$$

- Weight updates are traditionally computed using gradient descent (or a related variant), in which case, one applies the **chain rule** of differential calculus with respect to the the various function compositions defined across the layers of the network.

Computational Graph: Example

- Next, we consider a simple example of learning the XOR function in 2D.

x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	0

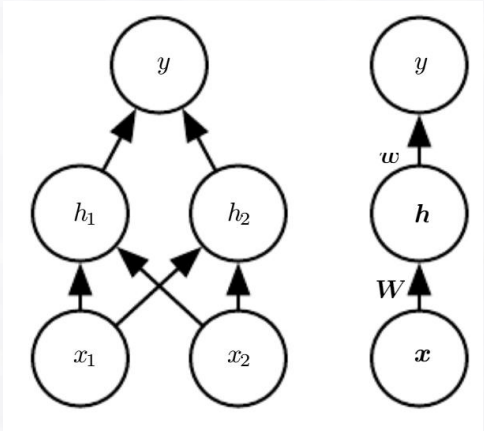


• (Right image, left-side) Every unit in computational graph is shown; (Right image right-side) More compactly, each node represents a layer.

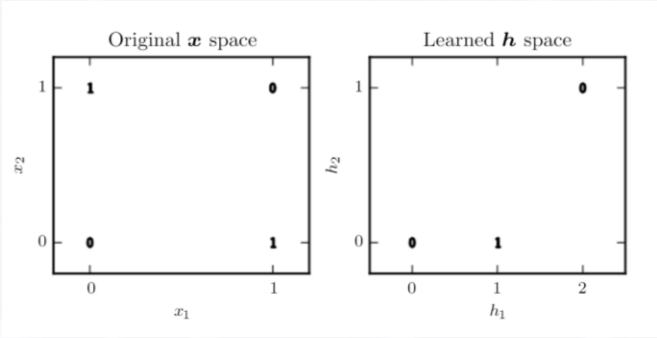
Computational Graph: Example

- Next, we consider a simple example of learning the XOR function in 2D.

x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	0



- (Right image, left-side) Every unit in computational graph is shown; (Right image right-side) More compactly, each node represents a layer.

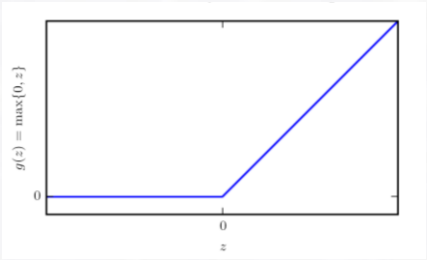


- (Left) XOR represented in original space (notice the data are not linearly separable); (Right) By introducing non-linearity, the data are linearly separable in the learned space.

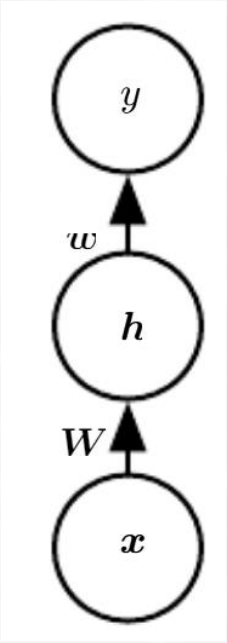
Computational Graph: Example

• Denote the *i*th element activation: $h_i = g\left(\boldsymbol{x}^T W_{:,i} + c_i\right)$, where g is our activation function – here we'll use the standard RELU depicted below, defined: $g(z) = max\{0, z\}$

• Notice that the complete (mathematical) specification of our network is given as:



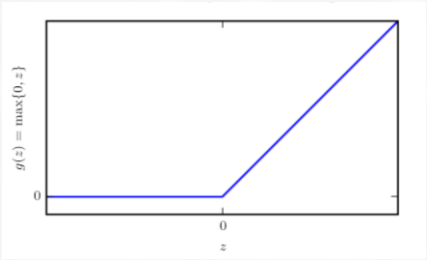
$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^T max\{0, \boldsymbol{W}^T x + c\} + b$$



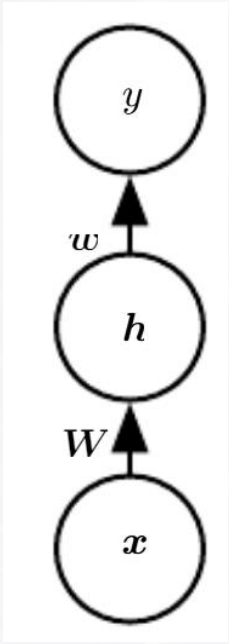
Computational Graph: Example

• Denote the ith element activation: $h_i = g\left(\boldsymbol{x}^T W_{:,i} + c_i\right)$, where g is our activation function – here we'll use the standard RELU depicted below, defined: $g(z) = max\{0, z\}$

• Notice that the complete (mathematical) specification of our network is given as:



$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^T max\{0, \boldsymbol{W}^T x + c\} + b$$



• Let: $\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$, $\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$, $\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$, $b = 0$, $\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$ (XOR input)

$$\boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} \rightarrow \boldsymbol{XW} + \boldsymbol{c} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \rightarrow max\{0, \boldsymbol{W}^T x + c\} \rightarrow \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\rightarrow \boldsymbol{w}^T max\{0, \boldsymbol{W}^T x + c\} + b = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Predicted output over XOR dataset.

Computational Graph: Example

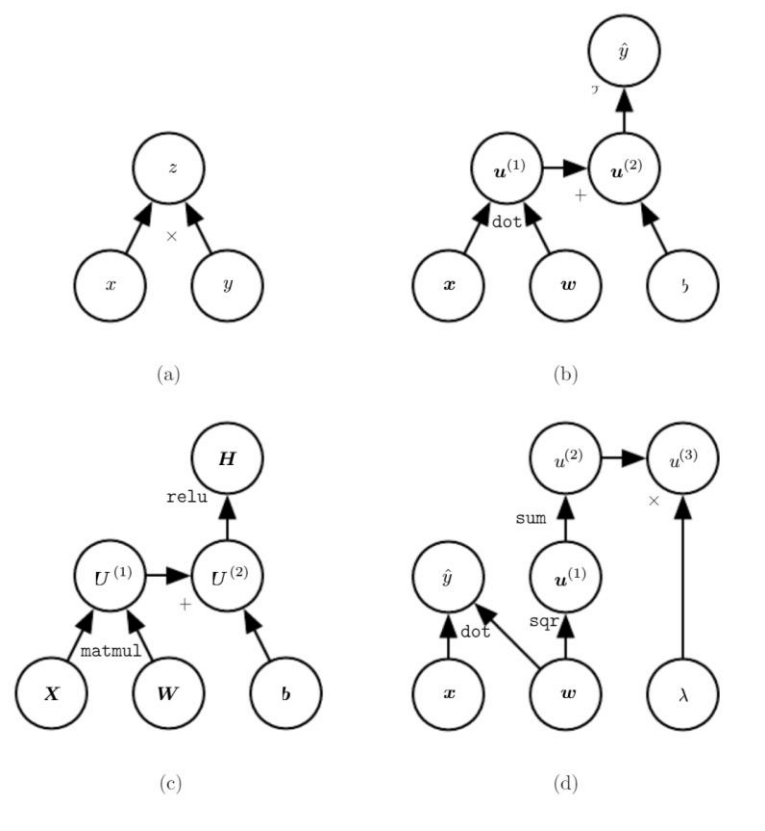
- Some example computation graphs:

(a) $z = xy$

(b) $y = \sigma\left(\boldsymbol{x}^T \boldsymbol{w} + b\right)$ (logistic regression)

(c) $\mathbf{H} = \max\{0, \boldsymbol{XW} + \boldsymbol{b}\}$

(d) linear regression model with regularization (L2 weight decay penalty), i.e., $\hat{y} = \boldsymbol{wx} + \lambda\sqrt{\sum_i w_i^2}$.

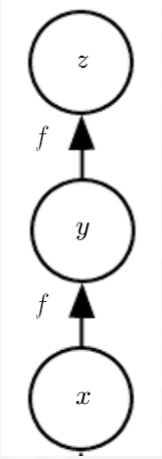


Backpropagation

• Recall the Chain Rule of Calculus:

Let $z = f(g(x)) = f(y)$, then the Chain Rule states:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$



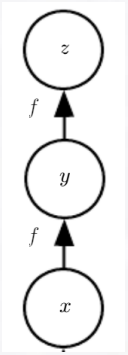
:

Backpropagation

• Recall the Chain Rule of Calculus:

Let $z = f(g(x)) = f(y)$, then the Chain Rule states:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$



• For functions of several variables, we introduce the analogue of the derivative, termed the **partial derivative**. Partial derivatives entail computing the derivative of a multivariate function wrt (“*with respect to*”) a single variable, while treating all other variables as constants. Let $f(x, y, z, \ldots)$; partial derivatives are commonly denoted:

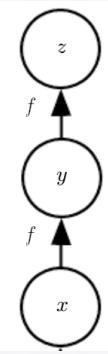
$$\frac{\partial f}{\partial x} \text{ or equivalently: } f_x \text{ or } D_x; \text{ recall that in general: } f_{xy} = f_{yx}, \text{ etc.}$$

Backpropagation

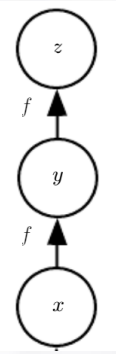
• We can thus generalize the chain rule to vector-valued functions. Suppose that $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n$; if $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$, then:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i} \text{ or equivalently in vector notation: } \nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z$$

Jacobian of g



Backpropagation



• We can thus generalize the chain rule to vector-valued functions. Suppose that $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n$; if $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y}) = f(g(\boldsymbol{x}))$ then:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i} \text{ or equivalently in vector notation: } \nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z$$

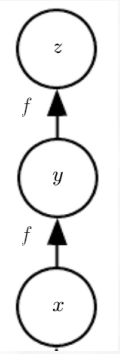
Jacobian of g

• Consider the following example:

Let $\mathbf{y} = g(\mathbf{x}) = g(\langle x_1, x_2 \rangle) = \langle x_1, x_1 \sin(x_2) \rangle$, $f(\mathbf{y}) = f(\langle y_1, y_2 \rangle) = y_1 + 2y_2$

$z = f(g(\mathbf{x})) = x_1 + 2x_1 \sin(x_2)$

Backpropagation



• We can thus generalize the chain rule to vector-valued functions. Suppose that $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n$; if $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y}) = f(g(\boldsymbol{x}))$ then:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i} \text{ or equivalently in vector notation: } \nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z$$

Jacobian of g

• Consider the following example:

Let $\mathbf{y} = g(\mathbf{x}) = g(\langle x_1, x_2 \rangle) = \langle x_1, x_1 \sin(x_2) \rangle$, $f(\mathbf{y}) = f(\langle y_1, y_2 \rangle) = y_1 + 2y_2$

$z = f(g(\mathbf{x})) = x_1 + 2x_1 \sin(x_2)$

$$\frac{\partial z}{\partial x_1} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x_1} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x_1} = 1 \cdot 1 + 2 \cdot \sin(x_2)$$

$$\frac{\partial z}{\partial x_2} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x_2} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x_2} = 1 \cdot 0 + 2 \cdot x_1 \cos(x_2)$$

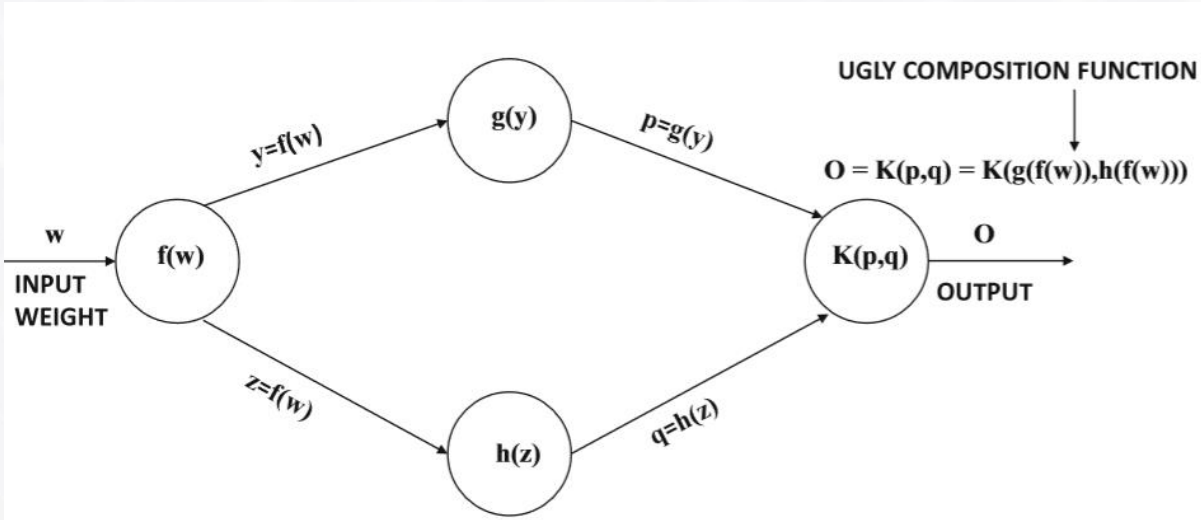
$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \end{bmatrix}^T \begin{bmatrix} \frac{\partial z}{\partial y_1} \\ \frac{\partial z}{\partial y_2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 2\sin(x_2) & x_1 \cos(x_2) \end{bmatrix}^T \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 & 2\sin(x_2) \\ 0 & x_1 \cos(x_2) \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 + 2\sin(x_2) \\ +2x_1 \cos(x_2) \end{bmatrix}$$

Jacobian of g

Backpropagation

• The main computational challenge for backpropagation relates to the multivariate chain rule. (see **pathwise aggregation lemma**, next slides)

$$\frac{\partial f(g_1(w), \ldots g_k(w))}{\partial w} = \sum_{i=1}^{k} \frac{\partial f(g_1(w), \ldots g_k(w))}{\partial g_i(w)} \cdot \frac{\partial g_i(w)}{\partial w}$$



$$\frac{\partial o}{\partial w} = \frac{\partial o}{\partial p} \cdot \frac{\partial p}{\partial w} + \frac{\partial o}{\partial q} \cdot \frac{\partial q}{\partial w} \quad \text{[Multivariable Chain Rule]}$$

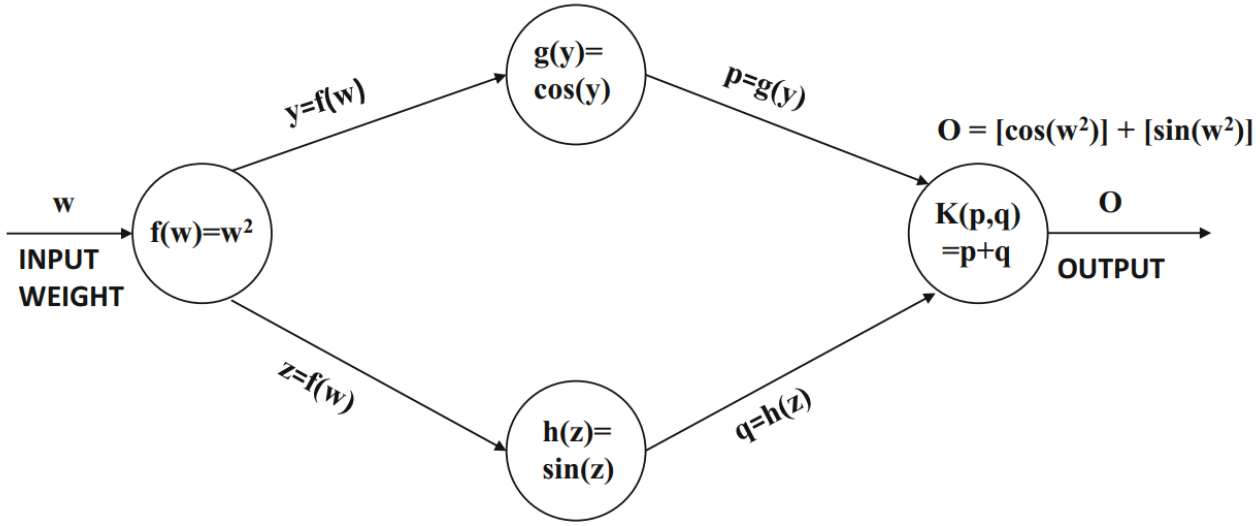
$$= \frac{\partial o}{\partial p} \cdot \frac{\partial p}{\partial y} \cdot \frac{\partial y}{\partial w} + \frac{\partial o}{\partial q} \cdot \frac{\partial q}{\partial z} \cdot \frac{\partial z}{\partial w} \quad \text{[Univariate Chain Rule]}$$

$$= \underbrace{\frac{\partial K(p,q)}{\partial p} \cdot g'(y) \cdot f'(w)}_{\text{First path}} + \underbrace{\frac{\partial K(p,q)}{\partial q} \cdot h'(z) \cdot f'(w)}_{\text{Second path}}$$

Backpropagation

• The main computational challenge for backpropagation relates to the multivariate chain rule. (see **pathwise aggregation lemma**, next slides)

$$\frac{\partial f(g_1(w), \ldots g_k(w))}{\partial w} = \sum_{i=1}^{k} \frac{\partial f(g_1(w), \ldots g_k(w))}{\partial g_i(w)} \cdot \frac{\partial g_i(w)}{\partial w}$$



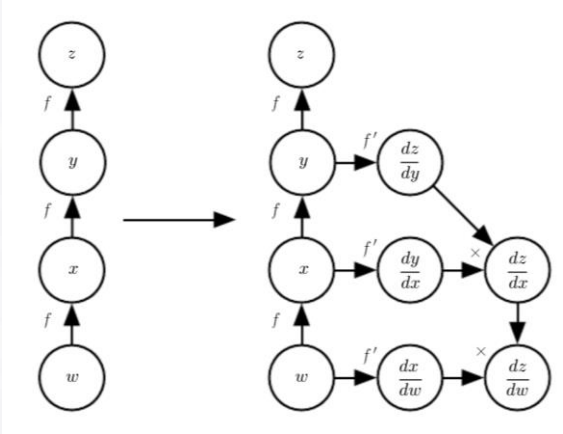
$$\frac{\partial o}{\partial w} = \underbrace{\frac{\partial K(p,q)}{\partial p}}_{1} \cdot \underbrace{g'(y)}_{-\sin(y)} \cdot \underbrace{f'(w)}_{2w} + \underbrace{\frac{\partial K(p,q)}{\partial q}}_{1} \cdot \underbrace{h'(z)}_{\cos(z)} \cdot \underbrace{f'(w)}_{2w}$$

$$= -2w \cdot \sin(y) + 2w \cdot \cos(z)$$

$$= -2w \cdot \sin(w^2) + 2w \cdot \cos(w^2)$$

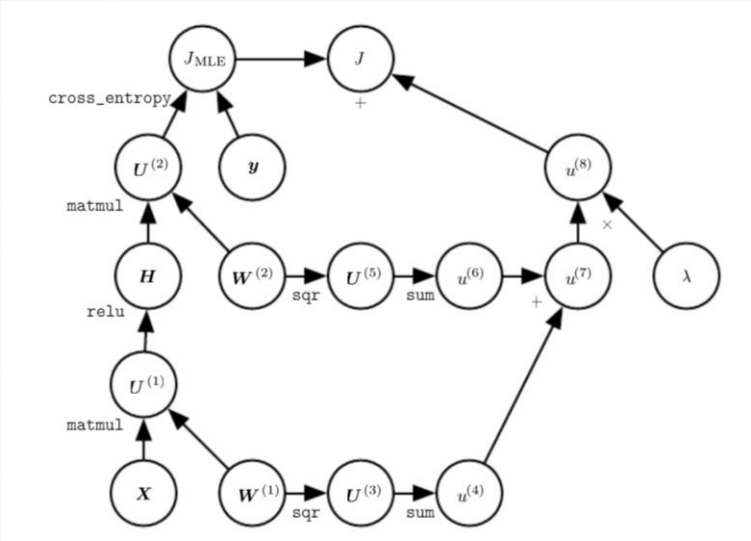
Computational Graphs & Backpropagation

• Here is an example schematic of symbol-to-symbol computation of derivatives from a computation graph.



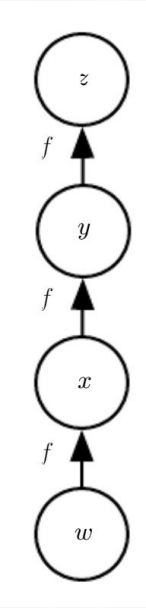
• Another example schematic of the computational graph used to train a single-layer NN using cross-entropy loss and weight decay:

$$J = J_{\text{MLE}} + \lambda \left(\sum_{i,j} \left(W^{(1)}_{i,j} \right)^2 + \sum_{i,j} \left(W^{(2)}_{i,j} \right)^2 \right)$$



Computational Graphs & Backpropagation

• Notice that the calculation of the chain rule along a path in a computational graph typically **admits of many redundancies**. Let $x = f(w), y = f(x), z = f(y)$:



$$\begin{aligned}
&\frac{\partial z}{\partial w} \\
=&\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\
=&f'(y)f'(x)f'(w) \\
=&f'(f(f(w)))f'(f(w))f'(w).
\end{aligned}$$

• Notice that the calculation of $\frac{\partial z}{\partial w}$ requires that we compute that value $f(w)$ many times. Naturally, a more efficient approach is to simply compute this value only once and store it in order to avoid these redundant calculations. This is the key idea behind applying **dynamic programming** to backpropagation.

Backpropagation

• **Pathwise Aggregation Lemma**: Consider a directed acyclic computational graph (DAG) in which the ith node contains variable $y(i)$. The local derivative $z(i,j)$ of the directed edge (i,j) in the graph is defined as: $\frac{\partial y(j)}{\partial y(i)}$. Let a non-null set of paths P exist from **variable w** in the graph **to output node** containing variable o. Then, the value of $\frac{\partial o}{\partial w}$ is given by computing the product of the local gradients along each path in P, and summing these products over all paths:

$$\frac{\partial o}{\partial w} = \sum_{p \in P} \prod_{(i,j) \in p} z(i,j)$$

Backpropagation

• **Pathwise Aggregation Lemma**: Consider a directed acyclic computational graph (DAG) in which the ith node contains variable y(i). The local derivative **z(i,j)** of the directed edge (i,j) in the graph is defined as: $\frac{\partial y(j)}{\partial y(i)}$. Let a non-null set of paths P exist from variable w in the graph to output node containing variable o. Then, the value of $\frac{\partial o}{\partial w}$ is given by computing the product of the local gradients along each path in P, and summing these products over all paths:

$$\frac{\partial o}{\partial w} = \sum_{p \in P} \prod_{(i,j) \in p} z(i,j)$$

An Exponential-Time Algorithm

The fact that we can compute the composite derivative as an aggregation of the products of local derivatives along all paths in the computational graph leads to the following exponential-time algorithm:

1. Use computational graph to compute the value $y(i)$ of each nodes i in a forward phase.
2. Compute the local partial derivatives $z(i,j) = \frac{\partial y(j)}{\partial y(i)}$ on each edge in the computational graph.
3. Let $\mathcal{P}$ be the set of all paths from an input node with value w to the output. For each path $P \in \mathcal{P}$ compute the product of each local derivative $z(i,j)$ on that path.
4. Add up these values over all paths in $\mathcal{P}$.

Backpropagation: Dynamic Programming

• Although the summation discussed previously has an exponential number of paths, one can nonetheless compute it efficiently using dynamic programming.

• We want to compute the product of $z(i,j)$ over each path p ε P from source node w to output o and then add them:

$$S(w,o) = \sum_{p \in P} \prod_{(i,j) \in p} z(i,j)$$

Backpropagation: Dynamic Programming

• Although the summation discussed previously has an exponential number of paths, one can Nevertheless, compute this result efficiently using dynamic programming.

• We want to compute the product of z(i,j) over each path p ε P from source node *w* to output *o* and then add them:

$$S(w,o) = \sum_{p \in P} \prod_{(i,j) \in p} z(i,j)$$

(*) In practice, when using dynamic programming for backpropagation for redundant calculations required for enumerating all paths.

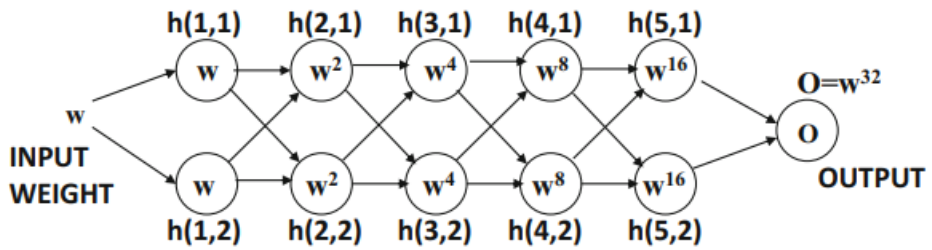
Backpropagation: Dynamic Programming

• Pathwise Aggregation – in this example, explicit computation of the partial derivative of the output (o) wrt to the input (w), requires "pathwise aggregation" over all $2^5 = 32$ paths in the network!

$$\frac{\partial o}{\partial w} = \sum_{p \in P} \prod_{(i,j) \in p} z(i,j)$$

$$\frac{\partial o}{\partial w} = \sum_{j_1,j_2,j_3,j_4,j_5 \in \{1,2\}^5} \underbrace{\prod h(1,j_1)}_{w} \underbrace{h(2,j_2)}_{w^2} \underbrace{h(3,j_3)}_{w^4} \underbrace{h(4,j_4)}_{w^8} \underbrace{h(5,j_5)}_{w^{16}}$$

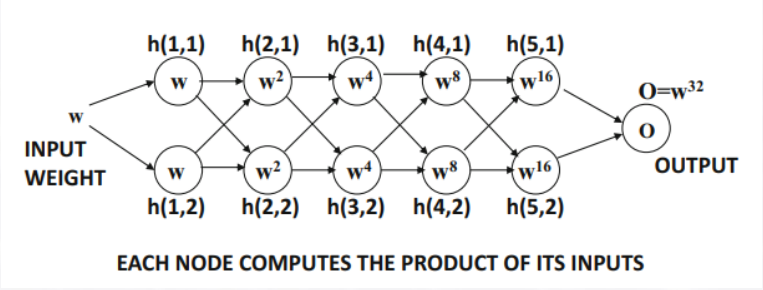
$$= \sum_{\text{All 32 paths}} w^{31} = 32w^{31}$$

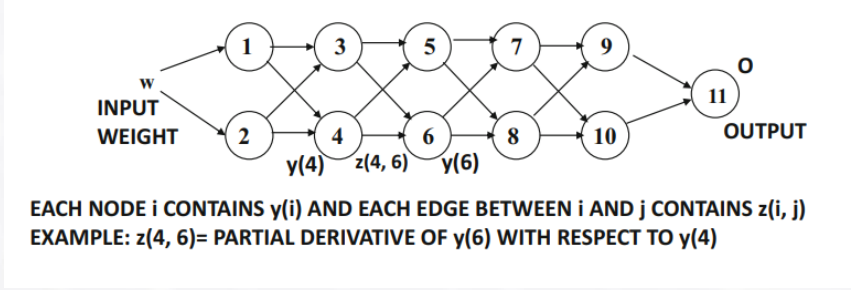


EACH NODE COMPUTES THE PRODUCT OF ITS INPUTS

Backpropagation: Dynamic Programming

• Notice that the given network represents a DAG (so it admits of a topological ordering), so we can apply dynamic programming (DP) to generate an efficient solution for the calculation of: $\frac{\partial o}{\partial w}$. Recall that $\mathrm{z(i,j)} = \frac{\partial y(j)}{\partial y(i)}$.



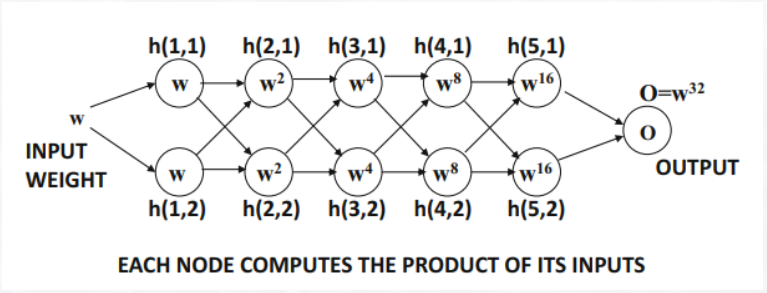


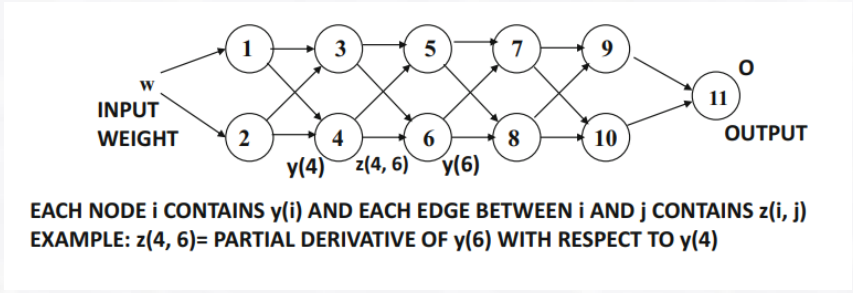
• We will use a common DP methodology – compute $\mathbf{S(w,}\boldsymbol{j}\mathbf{)}$ for all nodes w in the graph beginning with $\mathrm{w} := o$ (so we traverse right to left); see the formula below for the general calculation of $\mathrm{S(i,}o\mathrm{)}$. Note that $\mathbf{A(}\boldsymbol{i}\mathbf{)}$ symbolizes the set of nodes at the endpoints of outgoing edge for each intermediate node i. Let $\mathrm{S(11,11)} = 1$ by default.

$$S(i,o) \Leftarrow \sum_{j \in A(i)} S(j,o)z(i,j)$$

Backpropagation: Dynamic Programming

• We will use a common DP methodology – compute $S(w, j)$ for all nodes w in the graph beginning with $w := o$ (so we traverse right to left); see the formula below for the general calculation of $S(i, o)$. Note that $A(i)$ symbolizes the set of nodes at the end points of outgoing edge for each intermediate node i. Let $S(11,11) = 1$.





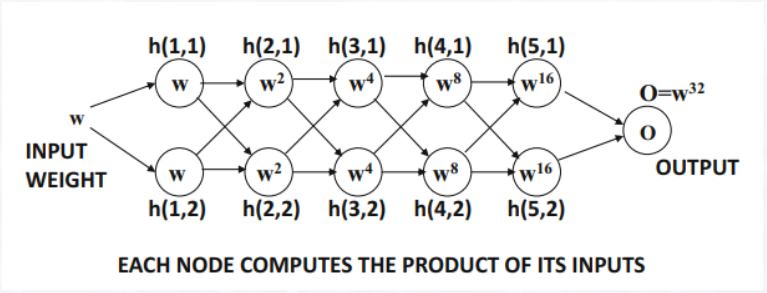
$$S(i,o) \Leftarrow \sum_{j \in A(i)} S(j,o) z(i,j)$$

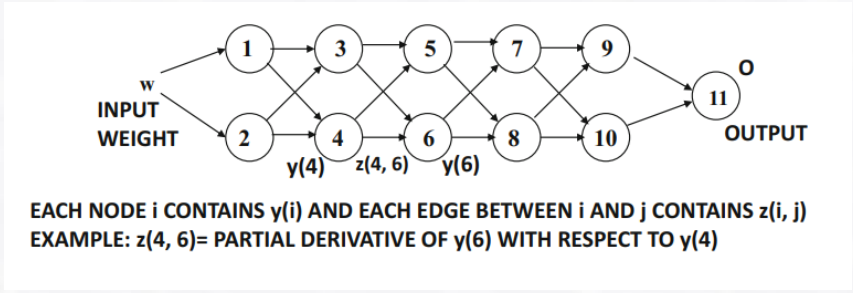
• Next, we compute $S(9,11) = S(11,11) \cdot z(9,11) = 1 \cdot \frac{\partial y(11)}{\partial y(9)} = \frac{\partial w^{32}}{\partial w^{16}} = 2w^{16}$

• Similarly, $S(10,11) = S(11,11) \cdot z(10,11) = 1 \cdot \frac{\partial y(11)}{\partial y(10)} = \frac{\partial w^{32}}{\partial w^{16}} = 2w^{16}$

Backpropagation: Dynamic Programming

• We will use a common DP methodology – compute $S(w, j)$ for all nodes w in the graph beginning with $w := o$ (so we traverse right to left); see the formula below for the general calculation of $S(i, o)$. Note that $A(i)$ symbolizes the set of nodes at the end points of outgoing edge for each intermediate node i. You should see that $S(11,11) = 1$.





$$S(i,o) \Leftarrow \sum_{j \in A(i)} S(j,o) z(i,j)$$

• Next, we compute $S(9,11) = S(11,11) \cdot z(9,11) = 1 \cdot \frac{\partial y(11)}{\partial y(9)} = \frac{\partial w^{32}}{\partial w^{16}} = 2w^{16}$

• Similarly, $S(10,11) = S(11,11) \cdot z(10,11) = 1 \cdot \frac{\partial y(11)}{\partial y(10)} = \frac{\partial w^{32}}{\partial w^{16}} = 2w^{16}$

and $S(7,11) = \boxed{S(9,11)} \cdot z(7,9) + \boxed{S(10,11)} \cdot z(7,10) = 2w^{16} \frac{\partial y(9)}{\partial y(7)} +$

$2w^{16} \frac{\partial y(10)}{\partial y(7)} = 2w^{16} \frac{\partial w^{16}}{\partial w^{8}} + 2w^{16} \frac{\partial w^{16}}{\partial w^{8}} = 4w^{24} + 4w^{24} = 8w^{24}.$

• You should verify that iterating this DP strategy to completion yields: $\frac{\partial o}{\partial w} = 32w^{31}$; note that this method avoids exponential path aggregation calculations, as was to be shown.

Automatic Differentiation

• **Automatic Differentiation** (AD) is a set of techniques to numerically evaluate the derivative of a function. Many contemporary ML and DL libraries (e.g., Pytorch, TensorFlow) include AD capabilities.

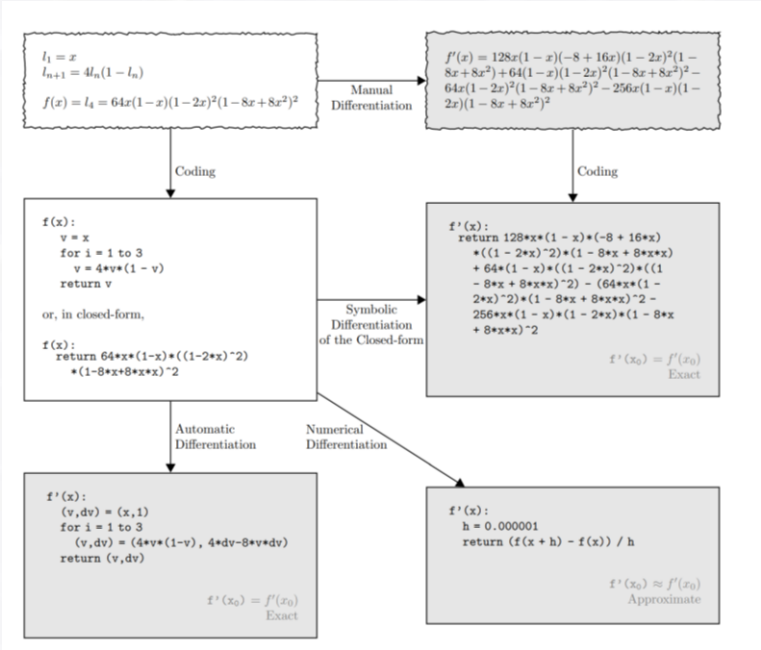
• Different from **symbolic differentiation** (i.e., directly using mathematical expression) and **numerical differentiation** (e.g., an iterative algorithm to estimate the derivative of a function), AD replaces the domain of variables to incorporate derivatives per the chain rule.

Automatic Differentiation

• **Automatic Differentiation** (AD) is a set of techniques to numerically evaluate the derivative of a function. Many contemporary ML and DL libraries (e.g., Pytorch, TensorFlow) include AD capabilities.

• Different from **symbolic differentiation** (i.e., directly using mathematical expression) and **numerical differentiation** (e.g., an iterative algorithm to estimate the derivative of a function), AD replaces the domain of variables to incorporate derivatives per the chain rule.

• AD computes derivatives through the accumulation of values during code execution to generate numerical derivative evaluations (rather than derivative expressions).



Baydin et al., "Automatic Differentiation: A Survey" (JMLR 2018)

Automatic Differentiation

• AD is a deep topic, for brevity we note that at a high-level AD uses the chain rule to compute the accumulation of derivatives. This is done for two fundamental processes: (1) **forward accumulation** (i.e., AD for forward pass through a NN) and (2) **reverse accumulation** (used for backprop).

Automatic Differentiation

• AD is a deep topic, for brevity we note that at a high-level AD uses the chain rule to compute the accumulation of derivatives. This is done for two fundamental processes: (1) **forward accumulation** (i.e., AD for forward pass through a NN) and (2) **reverse accumulation** (used for backprop).

• Consider the following example (https://sidsite.com/posts/autodiff/):

```
a = 4
b = 3
c = a + b  # = 4 + 3 = 7
d = a * c  # = 4 * 7 = 28
```

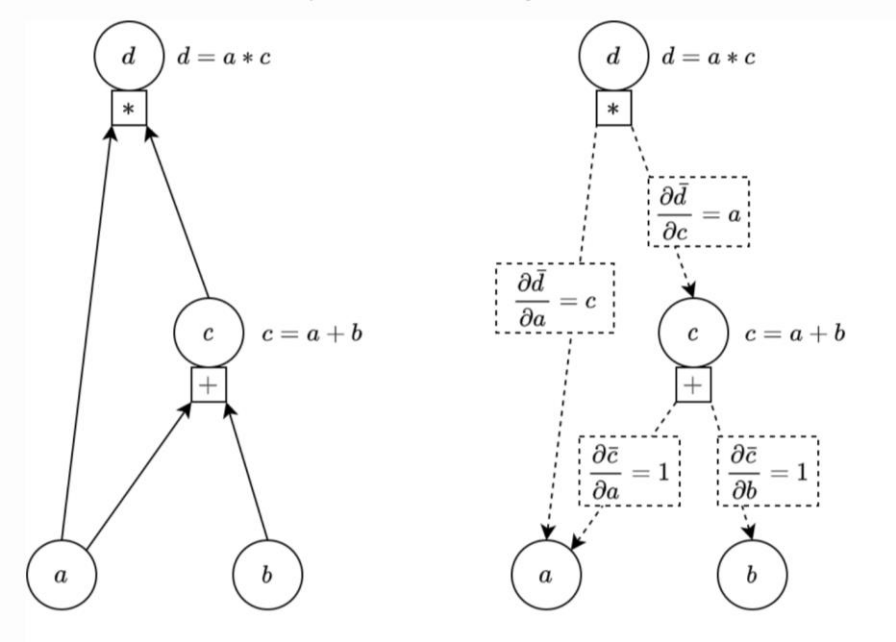
where we wish to compute $\frac{\partial d}{\partial a}$. Using the product rule we have:

$$
\begin{aligned}
d &= a * c \\
\frac{\partial d}{\partial a} &= \frac{\partial a}{\partial a} * c + a * \frac{\partial c}{\partial a} \\
\frac{\partial d}{\partial a} &= c + a * \frac{\partial c}{\partial a} \\
\frac{\partial d}{\partial a} &= (a + b) + a * \frac{\partial (a + b)}{\partial a} \\
\frac{\partial d}{\partial a} &= a + b + a * (\frac{\partial a}{\partial a} + \frac{\partial b}{\partial a}) \\
\frac{\partial d}{\partial a} &= a + b + a * (1 + 0) \\
\frac{\partial d}{\partial a} &= a + b + a \\
\frac{\partial d}{\partial a} &= 2a + b \\
\frac{\partial d}{\partial a} &= 2 * 4 + 3 = 11
\end{aligned}
$$

• Note that if we wish to compute $\frac{\partial d}{\partial b}$ a similar tedious process is required.

Automatic Differentiation

• Let's now contrast the computation of $\frac{\partial d}{\partial a}$ using AD.




```
a = 4
b = 3
c = a + b   # = 4 + 3 = 7
d = a * c   # = 4 * 7 = 28
```

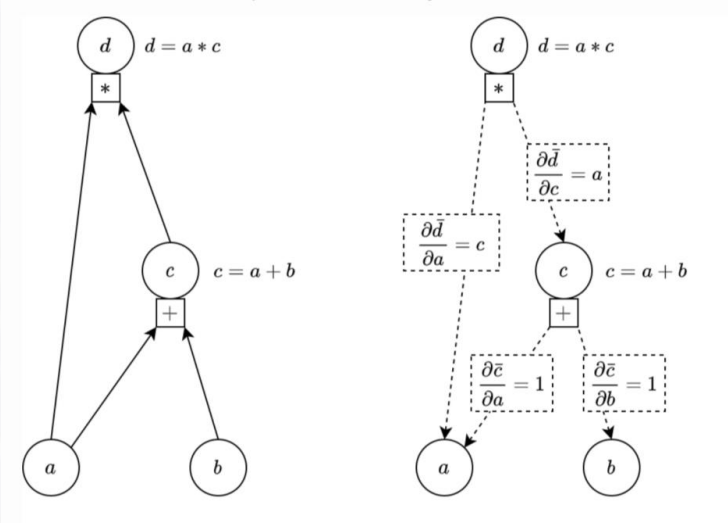
• The left image denotes the computational graph for this problem. On the right, we see the AD methodology. Consider the derivatives appearing on the edges as **local derivatives**.

• The basic idea through *reverse accumulation* is to begin at the output node of the computational graph, and then consider each path in the computational graph from the output node to the input nodes.

• We follow two simple rules: (1) add together different path accumulations and (2) multiply local derivatives along each path.

Automatic Differentiation

• Let's now contrast the computation of $\frac{\partial d}{\partial a}$ using AD.




```
a = 4
b = 3
c = a + b   # = 4 + 3 = 7
d = a * c   # = 4 * 7 = 28
```

• We follow two simple rules: (1) add together different path accumulations and (2) multiply local derivatives along each path.

$$\frac{\partial d}{\partial a} = \frac{\partial \bar{d}}{\partial a} + \frac{\partial \bar{d}}{\partial c} * \frac{\partial \bar{c}}{\partial a}$$
$$\frac{\partial d}{\partial a} = c + a * 1$$
$$\frac{\partial d}{\partial a} = a + b + a$$
$$\frac{\partial d}{\partial a} = 2a + b$$
$$\frac{\partial d}{\partial a} = 11$$

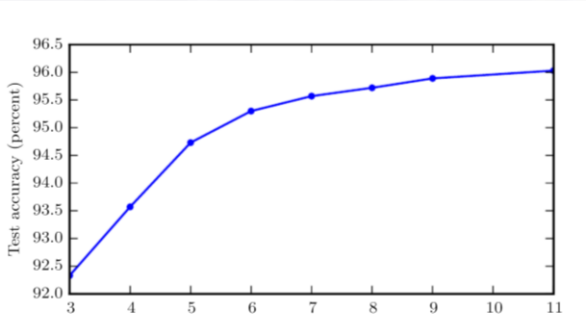
AD execution

$$d = a * c$$
$$\frac{\partial d}{\partial a} = \frac{\partial a}{\partial a} * c + a * \frac{\partial c}{\partial a}$$
$$\frac{\partial d}{\partial a} = c + a * \frac{\partial c}{\partial a}$$
$$\frac{\partial d}{\partial a} = (a + b) + a * \frac{\partial (a + b)}{\partial a}$$
$$\frac{\partial d}{\partial a} = a + b + a * (\frac{\partial a}{\partial a} + \frac{\partial b}{\partial a})$$
$$\frac{\partial d}{\partial a} = a + b + a * (1 + 0)$$
$$\frac{\partial d}{\partial a} = a + b + a$$
$$\frac{\partial d}{\partial a} = 2a + b$$
$$\frac{\partial d}{\partial a} = 2 * 4 + 3 = 11$$

conventional Chain Rule execution

Architectural Considerations

• Vanilla NNs consist of chains of feed-forward layers, with the main considerations being the depth of the network and width of each layer.



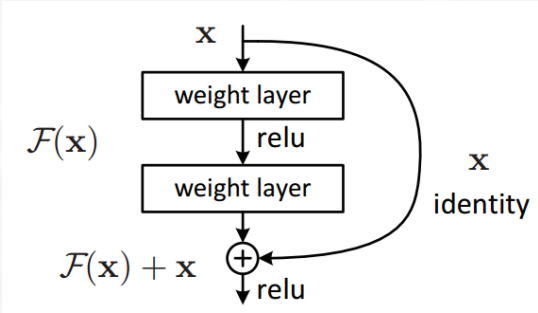
• In practice, though, NN architectures can be very diverse (see the NN "Zoo", shown previously). Ultimately, NN design should be intentional and developed with consideration for the specific task at hand.

• Special architectures for computer vision called convolutional neural networks (CNNs) are described later in our course. FF networks can be generalized to the recurrent neural networks (RNNs) for sequence processing, which have their own architectural considerations.

http://playground.tensorflow.org/

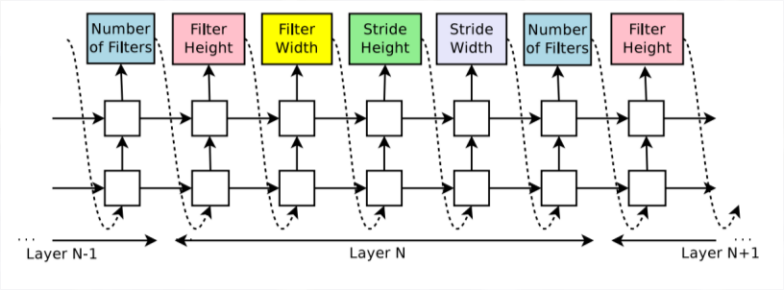
Architectural Considerations

• Observe that layers need not be connected in a sequential chain; many architectures make use of **skip connections** and **residual layers** to benefit gradient flow in the network.



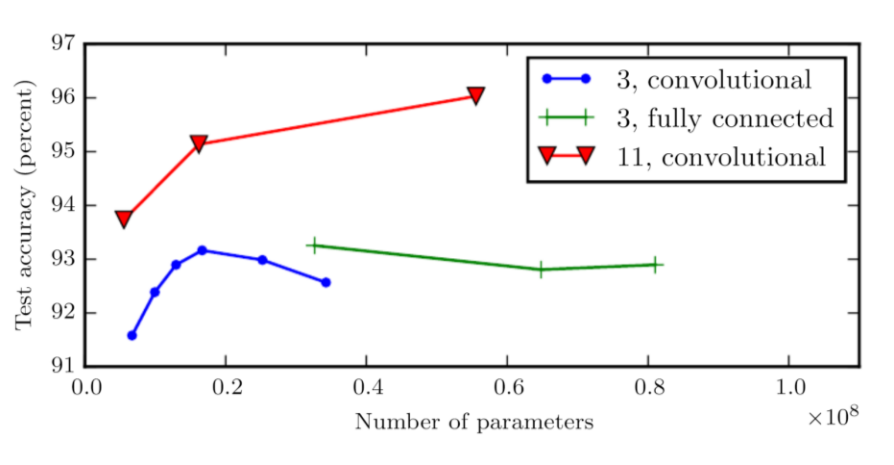
• One can additionally vary the connectivity strategy of the network. Many specialized networks have **fewer connections** (than dense networks) or they admit of some other form of compression. Note that CNNs utilize **parameter sharing** to this end**.**

• Recently, **Neural Architecture Search** (NAS) has emerged as a new paradigm for automating the design of DNNs.



Architectural Considerations

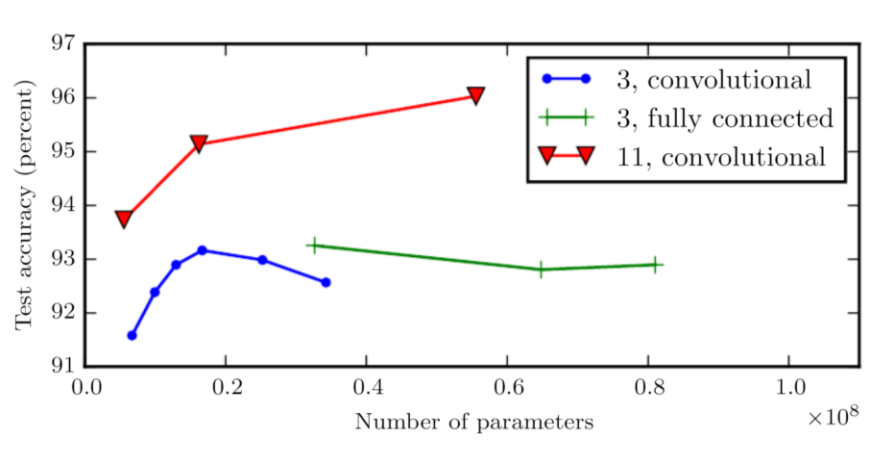
• DNNs frequently embody large and unwieldy, overparameterized models. Thus, recent research has focused on transforming DNNs into more sustainable network designs.



• This effort is catalyzed by several factors, including: the desire to conserve memory and compute overhead for the deployment of commercial DL models, energy sustainability, the need for greater model interpretability, and the aspiration to port DL models to low compute environments, including edge and IOT devices.

Architectural Considerations

• DNNs frequently embody large and unwieldy, overparameterized models. Thus, recent research has focused on transforming DNNs into more sustainable network designs.



• This effort is catalyzed by several factors, including: the desire to conserve memory and compute overhead for the deployment of commercial DL models, energy sustainability, the need for greater model interpretability, and the aspiration to port DL models to low compute environments, including edge and IOT devices.

• Today there exist a large variety of DL model compression techniques, due to the desirability of compact models with state-of-the-art functionality.

• Roughly, these techniques fall into several generic categories, comprising **pruning**, **quantization**, **low-rank and sparse approximations**, and **knowledge distillation**.

Weight Initialization

• Training algorithms for DNN models are usually iterative and thus require the user to specify some initial point from which to begin the iterations. Moreover, training deep models is a sufficiently difficult task that most algorithms are strongly affected by the choice of initialization.

• The **initial point can determine whether the algorithm converges at all**, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether. When learning does converge, **the initial point can determine how quickly learning converges and whether it converges to a point with high or low cost**.

Weight Initialization

• Modern initialization strategies are usually simple and heuristic; designing improved initialization strategies is a difficult task because NN optimization is not yet well understood.

• The most general guideline agreed upon by most practitioners is known as "**symmetry-breaking**." If two hidden units with the same activation function are connected to the same inputs, then these units have different initial parameters. If the training is deterministic, "symmetric" units will update identically (and hence be useless); even if the training is stochastic, it is usually best to initialize each unit to compute a different function from all the other units.

• Note that the scale of the initial distribution does have a large effect on both the outcome of the optimization procedure and the ability of the network to generalize.

Weight Initialization

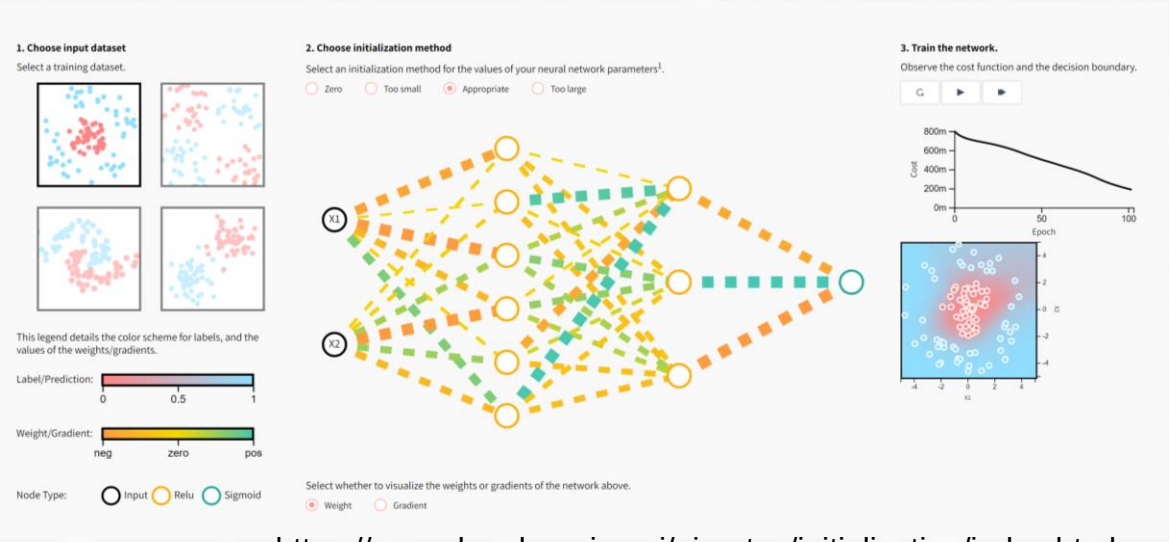
• Larger initial weights will yield a strong symmetry-breaking effect, helping to avoid redundant units; in addition, they will also potentially help avoid the problem of vanishing gradients. Nevertheless, they may conversely exacerbate the exploding gradient problem; in RNNs, large initial weights can manifest *chaotic behavior*.

* **Sparse initialization** (Martens, 2010) fixes the number of non-zero weights for initialization; **Xavier initialization** draws random initial values from a distribution with zero mean and variance inversely proportional to the size of the previous layer in the network.

Weight Initialization

• Another related approach is to initialize the weights to generate random values from a Gaussian distribution with zero mean and small standard deviation (e.g. 10^{-2}). This will result in small random values that are both positive and negative.

• One problem with this initialization is that **it is not sensitive to the number of inputs to a specific neuron**. For example, if one neuron has only 2 inputs and another has 100 inputs, the output of the latter is far more sensitive to the average weight because of the additive effect of more inputs (which will manifest itself through a much larger gradient).



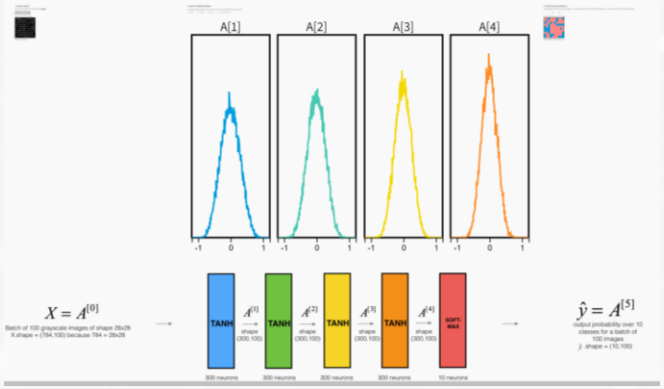
https://www.deeplearning.ai/ai-notes/initialization/index.html

Weight Initialization

(*) It can be shown that the variance of outputs scales with the number of inputs, and therefore the standard deviation scales with the square root of the number of inputs.

• To balance this fact, each weight can be initialized by a value drawn from $N\left(0, \frac{1}{\sqrt{r}}\right)$, where r indicates the number of inputs to that neuron.

• **Xavier initialization** is somewhat more sophisticated, so that initial weights are drawn from $N\left(0, \sqrt{\frac{2}{r_{in}+r_{out}}}\right)$, where r_{in} and r_{out} are the fan-in and fan-out values of a particular neuron, respectively.

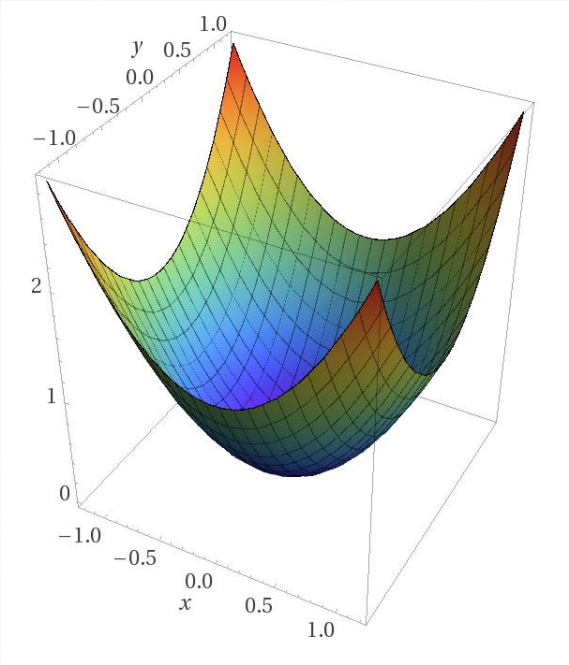


https://www.deeplearning.ai/ai-notes/initialization/index.html

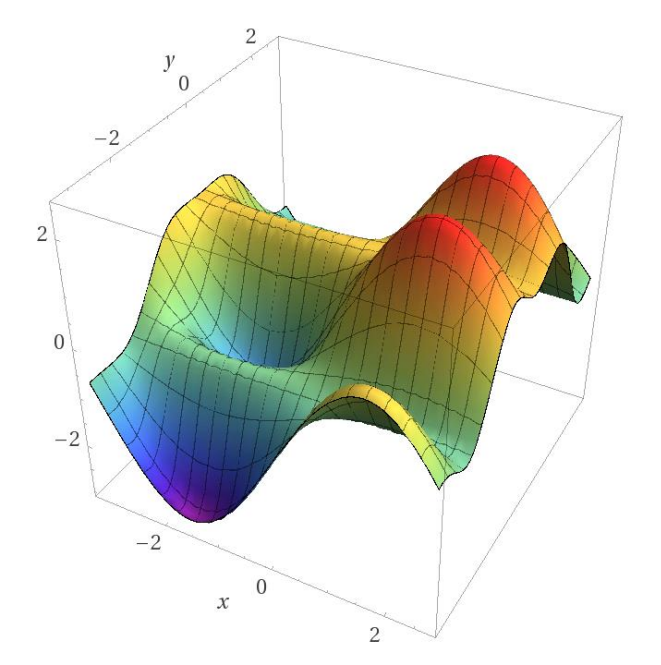
Challenges for DNN Optimization

• Traditionally, ML implementations avoid the difficulty of general optimization by carefully designing the objective function and constraints to ensure that the optimization problem is convex.

• When training NNs, however, we must confront **the general non-convex case**.



Convex Function



Non-Convex Function

Challenges for DNN Optimization: Local Minima

• For a convex function, any local minimum is guaranteed to be a global minimum.

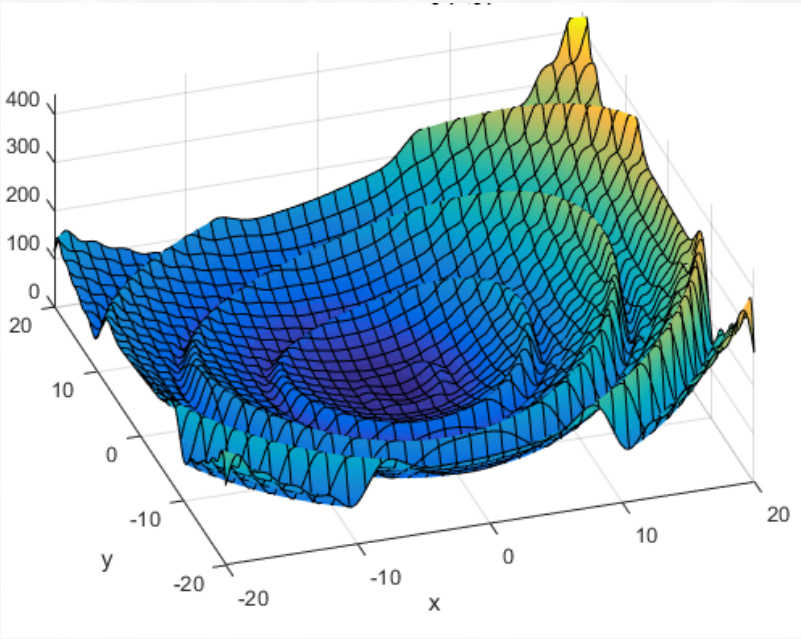
• With non-convex functions, such as with loss functions of NNs, it is possible to have many local minima. Moreover, nearly any DNN is essentially guaranteed to have a very large number of local minimal (even uncountably many).

• One of the chief reasons for the presence of many local minima for NNs, is due to the problem of model **identifiability**. A model is said to be identifiable if a sufficiently large training set can rule out all but one setting of the mode's parameters.

Challenges for DNN Optimization: Local Minima

• Models with latent variables (e.g. hidden neurons) are not in general identifiable because we can obtain equivalent models by exchanging latent variables with one another.

• Local minima are problematic if they correspond with high cost (vis-à-vis the global minimum). ***Note** that local minima are typically less problematic for DNN training than saddle points (this concept is not always well-appreciated by ML practitioners).



Challenges for DNN Optimization: Plateaus, Saddle Points

• For many high-dimensional, non-convex functions, local minima (and maxima) are in fact rare compared to **saddle points**.

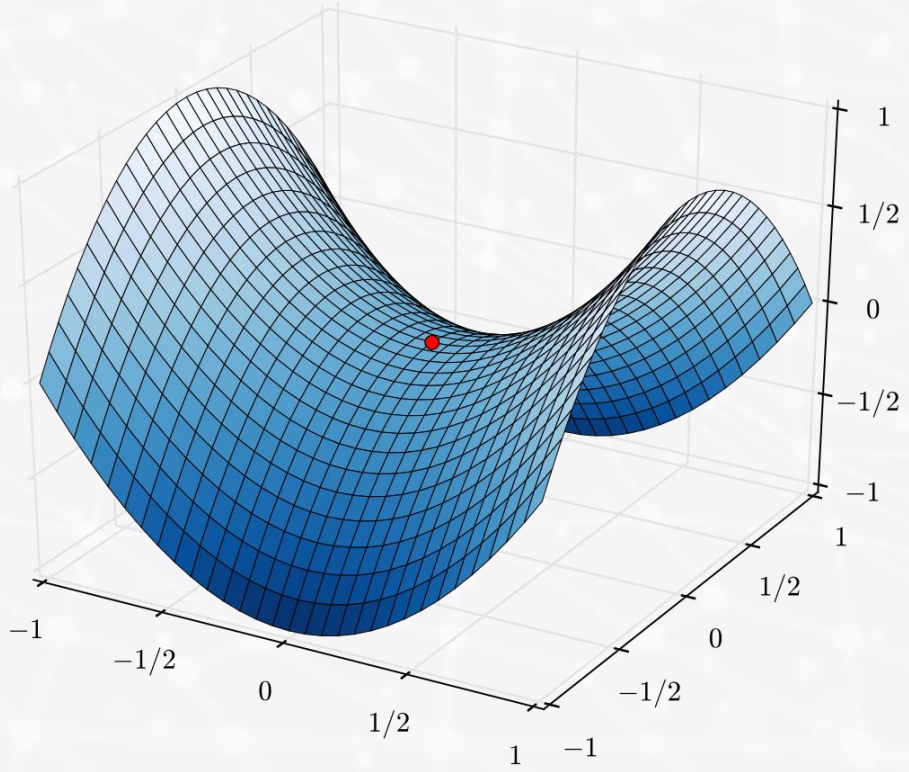
• Some points around a saddle point have greater cost than the saddle point, while others have lowers cost. At a saddle point, the **Hessian matrix** has both positive and negative eigenvalues.

• Why are saddle points more common than local extrema in high dimensions? The basic intuition is this: in order to render a local extreme value, all of the eigenvalues must be of the same sign (naturally, this is very unlikely – all things being equal – in high dimensions).

Challenges for DNN Optimization: Plateaus, Saddle Points

• For first-order optimization, saddle points are not necessarily a significant problem (Goodfellow); however, for second-order methods, they clearly constitute a problem.

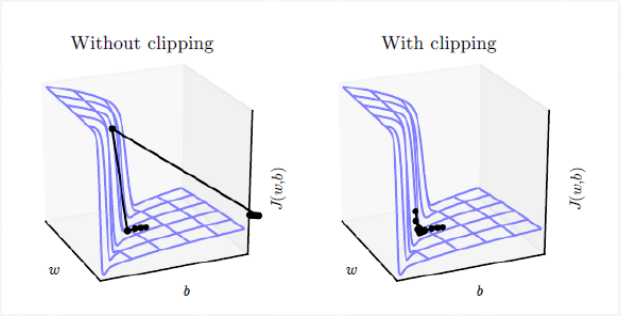
• Degenerate locations such as *plateaus* can pose major problems for all numerical algorithms.

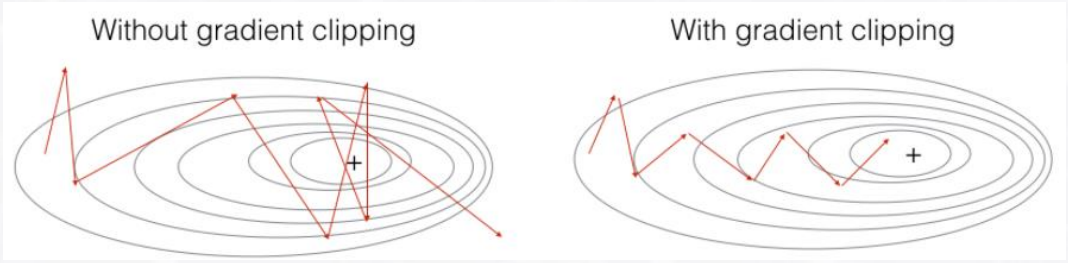


Challenges for DNN Optimization: Cliffs, Exploding and Vanishing Gradients

• NNs with many layers often have extremely steep regions resembling cliffs in the parameter space. This is due to the multiplication of several large weights together. On the face of an extremely steep cliff structure, the gradient update step can alter the parameters drastically.

• **Gradient clipping**, a heuristic technique, can help avoid this issue. When the traditional gradient descent algorithm proposes making a large step, the gradient clipping heuristic intervenes to reduce the step size, thereby making it less likely to go outside the region where the gradient indicates the direction of approximately steepest descent.

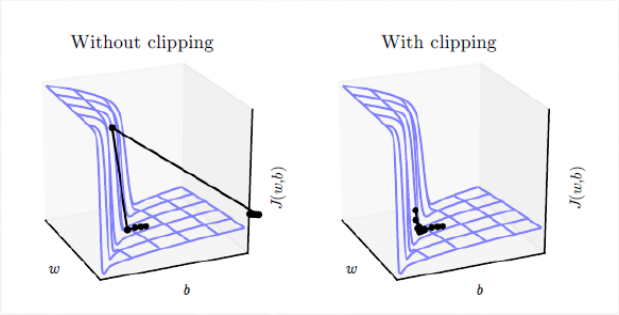


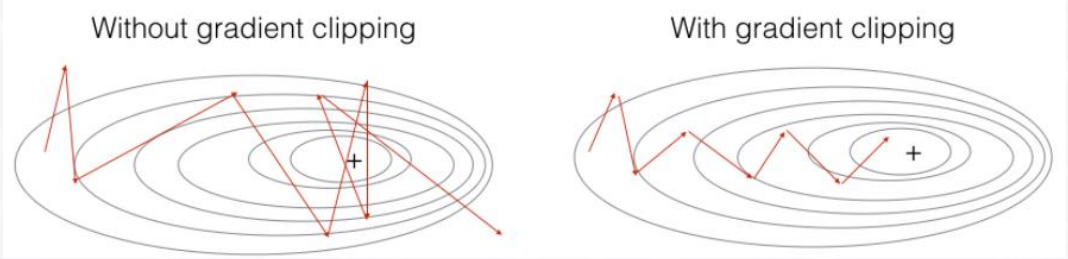


Challenges for DNN Optimization: Cliffs, Exploding and Vanishing Gradients

• NNs with many layers often have extremely steep regions resembling cliffs in the parameter space. This is due to the multiplication of several large weights together. On the face of an extremely steep cliff structure, the gradient update step can alter the parameters drastically.

• **Gradient clipping**, a heuristic technique, can help avoid this issue. When the traditional gradient descent algorithm proposes making a large step, the gradient clipping heuristic intervenes to reduce the step size, thereby making it less likely to go outside the region where the gradient indicates the direction of approximately steepest descent.



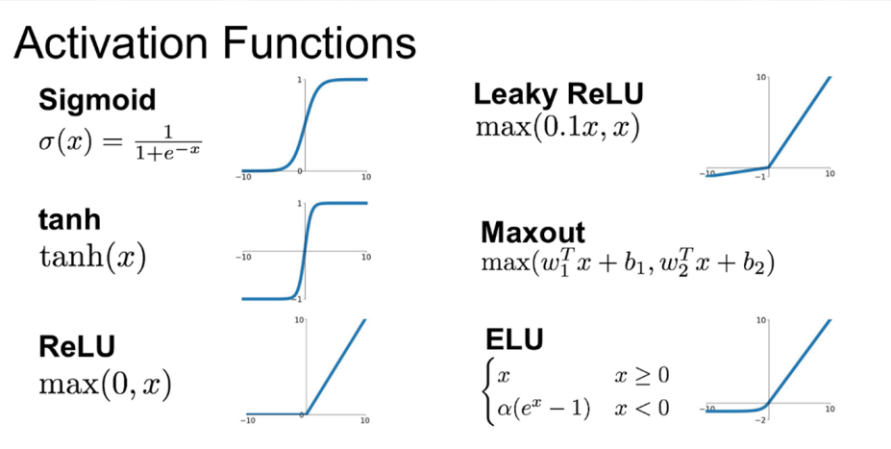


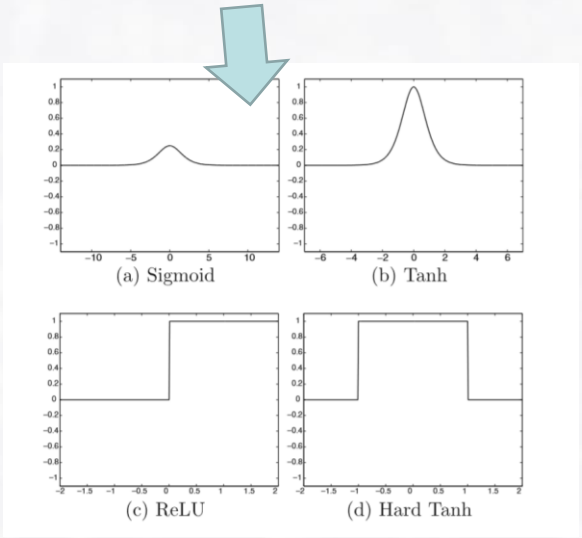
• When the computational graph for a NN becomes very large (e.g. RNNs), the issue of **exploding/vanishing gradients** can arise. Vanishing gradients make it difficult to known which direction the parameters should move to improve the cost function, while exploding gradients can make learning unstable.

*LSTMs, RELU, and ResNet (Microsoft) have been applied to solve the vanishing gradient problem.

Challenges for DNN Optimization: Hill-Climbing

• Potentially compounding this problem, many activation functions have small derivatives.





• The specific choice of activation function often has a considerable effect on the severity of the vanishing gradient problem.

• In recent years, the sigmoid and tanh activation functions have been increasingly supplanted by the **ReLU** and the **hard tanh** functions (see subsequent slides on variants of the ReLU activation).

(*) NB: The computational requirements to generate the derivate of a piecewise linear function are naturally significantly less than that required for a *transcendental function* (e.g. e^x) – where a sigmoid is defined as the composition of a transcendental function.

Cross-Entropy Loss

• As mentioned, **cross-entropy loss** is generally preferred to MSE, particularly for classification problems with DNNs.

Cross-entropy loss is defined:

$$E = -\sum c_i \log(p_i) + (1 - c_i) \log(1 - p_i)$$

Where c refers to one hot encoded classes (or labels), whereas p refers to softmax applied probabilities

Cross-Entropy Loss

• As mentioned, **cross-entropy loss** is generally preferred to MSE, particularly for classification problems with DNNs.

Cross-entropy loss is defined:

$$E = -\sum c_i \log(p_i) + (1 - c_i) \log(1 - p_i)$$

Where *c* refers to one hot encoded classes (or labels), whereas *p* refers to softmax applied probabilities

(2) Properties make cross-entropy a natural loss function:

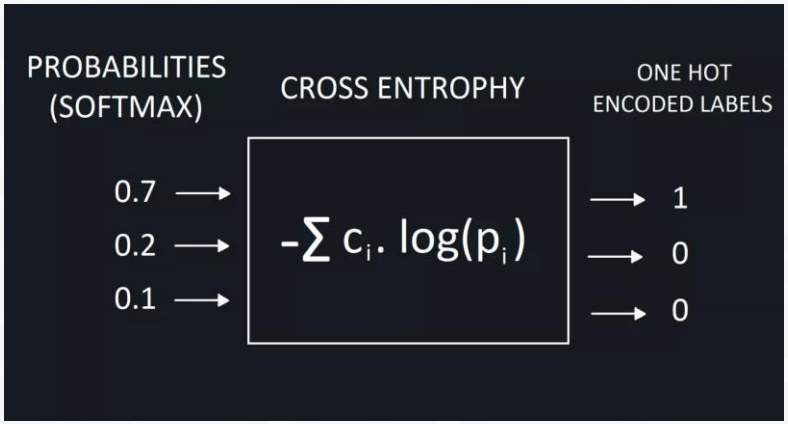
(1) $E \geq 0$; all individual terms are negative and there is a minus outside.

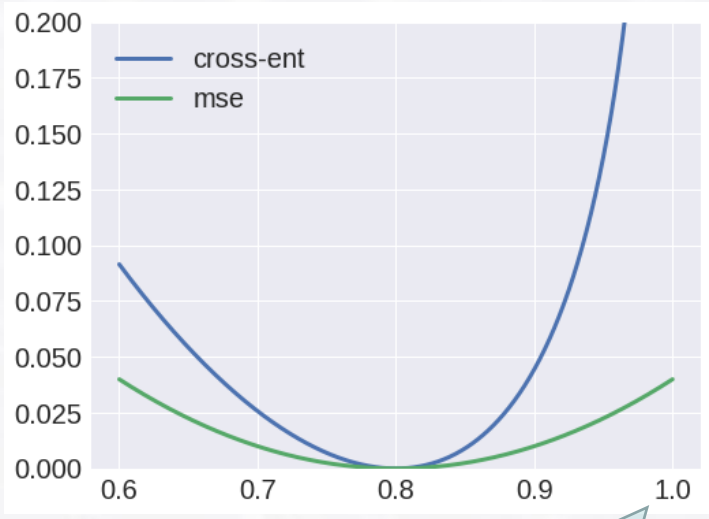
(2) If the neuron's actual output is close to the desired output for all training inputs, x, then the cross-entropy will be close to zero. To demonstrate this, we assume (WLOG) that the desired outputs c are all either 0 or 1. Suppose for example that $c = 0$ and $p \approx 0$, for some input x (so the neuron has done well on this input). The first term in E vanishes, while the second term is close to zero; a similar analysis holds when $c = 1$ and $p \approx 1$.

Cross-Entropy Loss

• Cross-entropy loss is defined:

$$E = -\sum c_i \log(p_i) + (1 - c_i) \log(1 - p_i)$$





One can show that, for example, that the partial derivative of the cross-entropy loss function is:

$$\frac{\partial E}{\partial w_j} = \sum_x x_j \left(\sigma(z) - y\right)$$

*(σ denotes the sigmoid function) Which indicates that the gradient is larger (i.e. learning is faster) the larger the error; in addition, the cross-entropy loss function does not in general "bottom out" like the MSE loss.

RELU & Their Generalizations

• Rectified linear units use the activation function $g(z) = \max\{0, z\}$.

• These units are easy to optimize because they are so similar to linear units; the only difference being the RELU is zero across half of its domain. This makes the derivatives through a RELU remain large whenever the unit is active.

• The gradients are therefore not only large but consistent.

RELUs are typically used on top of an *affine transformation*:

$$\mathbf{h} = g\left(\mathbf{W}^T\mathbf{x} + \mathbf{b}\right)$$

•One drawback of RELU: is that they cannot learn via gradient-based methods on examples for which their activation is zero; various generalizations of RELUs guarantee they receive gradient everywhere.

*affine transformations preserve points, straight lines, planes, and parallelism.

RELU & Their Generalizations

(3) Generalizations of RELUs are based on using a non-zero slope α_i when $z_i < 0$:

$$h_i = g(\mathbf{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

(1) **Absolute value rectification** fixes $\alpha_i = -1$, to obtain $g(z) = |z|$; this method has been used for object recognition from images, where it makes sense to seek features that are invariant under polarity reversal of the input illumination.

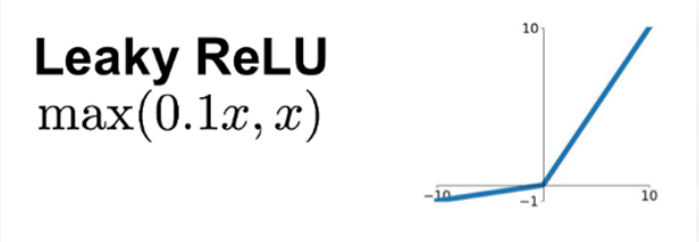
RELU & Their Generalizations

(3) Generalizations of RELUs are based on using a non-zero slope α_i when $z_i < 0$:

$$h_i = g(\mathbf{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

(1) **Absolute value rectification** fixes $\alpha_i = -1$, to obtain $g(z) = |z|$; this method has been used for object recognition from images, where it makes sense to seek features that are invariant under polarity reversal of the input illumination.

(2) **Leaky RELU** fixes α_i to a small value like 0.01.



• Note that the gradient of a standard ReLU is zero for negative values of its argument. While this inactivity is arguably biological-plausible -- since in real brains, neuron firing is often sporadic and followed by refractory periods (see previous slides) -- it can nevertheless lead to undesirable, pathological behavior for artificial NNs.

• In artificial NNs, zero outputs can cause some ReLU units to be "knocked out", in which case they can reach a state in which they are never further updated during training. Such a neuron can be considered **dead**, which is a kind of permanent "brain damage" in biological parlance.

(*) The problem of **dying neurons** can be partially ameliorated by the leaky ReLU.

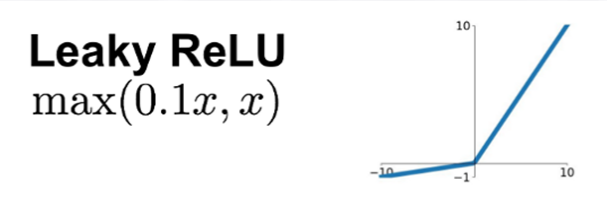
RELU & Their Generalizations

(3) Generalizations of RELUs are based on using a non-zero slope α_i when $z_i < 0$:

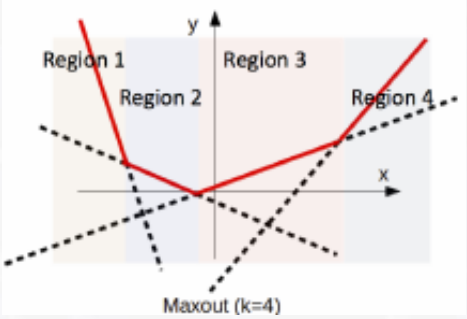
$$h_i = g(\mathbf{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

(1) **Absolute value rectification** fixes $\alpha_i = -1$, to obtain $g(z) = |z|$; this method has been used for object recognition from images, where it makes sense to seek features that are invariant under poliartiy reversal of the input illumination.

(2) **Leaky RELU** fixes α_i to a small value like 0.01.



(3) **Maxout** units (Goodfellow, 2013); instead of applying an element-wise function g(z), maxout units divide z into groups of k values. Each maxout unit then outputs the maximum element of one of those groups.



This provides a way of learning a piecewise linear function that responds to multiple directions in the input x space. Each maxout unit can learn a piecewise linear, convex function with up to *k* pieces; maxout units can thus be seen as learning the activation function itself rather than just the relationship between units; with enough k, a maxout unit can learn to approximate any convex function with arbitrary fidelity.

“Swish” Activations

• In 2018 Google Brain introduced “swish” activation functions (https://arxiv.org/pdf/1710.05941.pdf); swish a smooth, non-monotonic function matching/outperforming RELU in experiments.

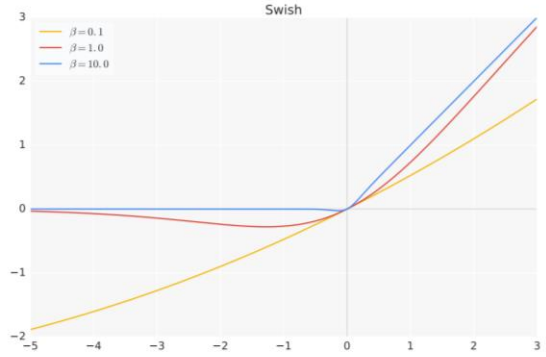


Figure 4: The Swish activation function.

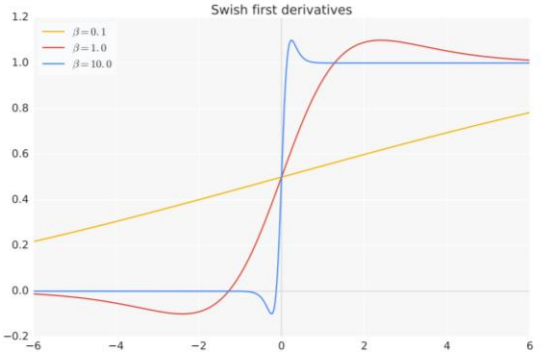
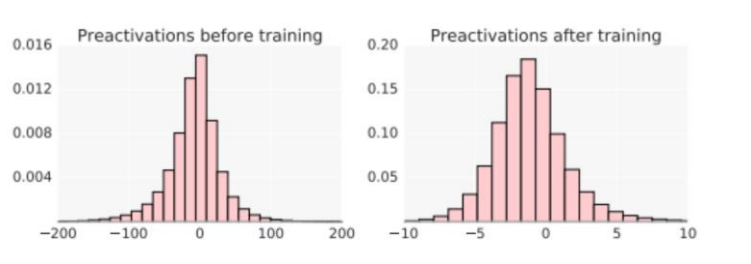


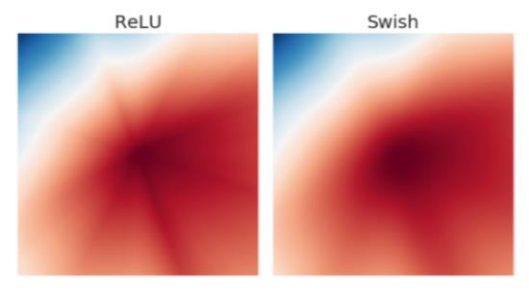
Figure 5: First derivatives of Swish.

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

• Of note, swish activation introduces a non-monotonic “bump” for $x < 0$ (the shape of this bump is modulated by the parameter β), as this regularizes large initial negative parameter weights.

• Non-monotonic feature increases the “expressivity” of activations; smoothness helps improve network optimization efficiency by making output space smoother and thus easier to traverse for optimization.





Feature Preprocessing

• There are two general forms of feature preprocessing:

(1) **Additive preprocessing and mean-centering**. It can be useful to mean-center the data to remove certain types of bias effects (recall that PCA does this); mean-centering is often paired with *standardization*.

(*) If it is desirable for all feature values to be non-negative (e.g. χ^2 test for feature selection), then one can simply add the absolute value of the maximum negative feature to the data set.

Feature Preprocessing

(1) **Additive preprocessing and mean-centering**. It can be useful to mean-center the data to remove certain types of bias effects (recall that PCA does this); mean-centering is often paired with *standardization*.

(2) **Feature normalization**. *Standardization* is a default feature normalization technique:

$$x_i \leftarrow \frac{x_i - \mu}{\sigma}$$

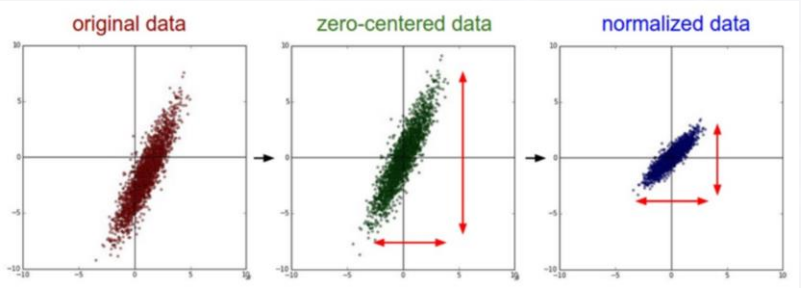
This assumes that each feature is drawn from a *standard normal Gaussian* (i.e., $N(0,1)$).

(*) Another, common form of feature normalization is **min-max normalization**:

$$x_i \leftarrow \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

This data transformation maps the dataset to [0,1].

(*) In general, feature normalization often ensures better performance, as it safeguards against **ill-conditioning** (where the loss function is more sensitive to some parameters vs. others).



Feature Preprocessing: Whitening

• **Whitening** is a linear data transformation that transforms a vector of random variables with known covariance matrix into a set of new variables whose covariance is the identity matrix (i.e. this procedure produces decorrelated variables with variances equal to 1). This procedure is called "whitening" because it changes the input vector into a white noise vector.

• Suppose X is a random column vector with non-singular covariance matrix M and mean equal to zero (that is to say, assume the data has been mean-centered).

Then the transformation Y=WX for the **whitening matrix** where W satisfies: $WW^T=M^{-1}$ yields the whitened random vector Y with unit diagonal covariance matrix.

Feature Preprocessing: Whitening

• Whitening is a linear data transformation that transforms a vector of random variables with known covariance matrix into a set of new variables whose covariance is the identity matrix (i.e. this procedure produces decorrelated variables with variances equal to 1). (this procedure is called "whitening" because it changes the input vector into a white noise vector).

• Suppose X is a random column vector with non-singular covariance matrix M and mean equal to zero (that is to say, assume the data has been mean-centered).

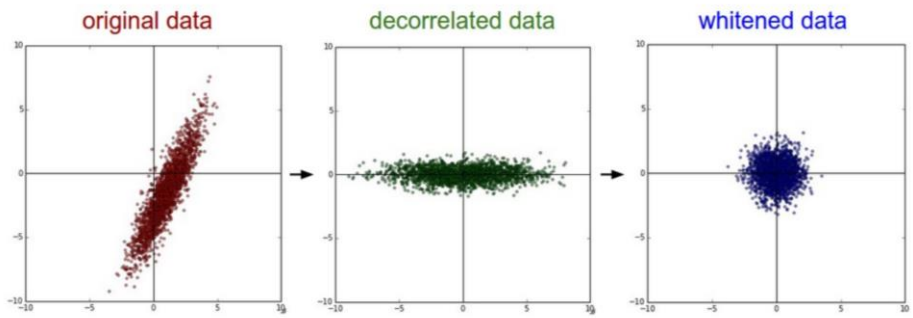
Then the transformation Y=WX for the **whitening matrix** where W satisfies: $WW^T=M^{-1}$ yields the whitened random vector Y with unit diagonal covariance matrix.

$$Cov(Y) = E\left[YY^T\right] = E\left[WX\left(WX\right)^T\right] = E\left[WXX^TW^T\right] = WW^T E\left[XX^T\right]$$

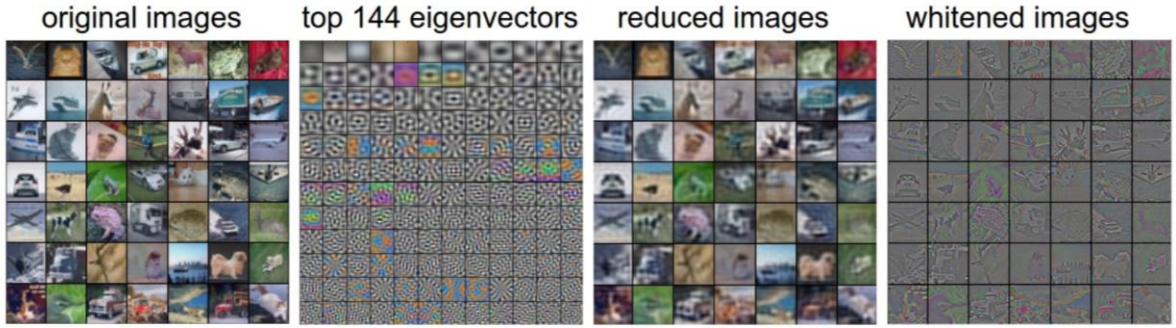
$$= Cov\left(X\right)^{-1} Cov\left(X\right) = I$$

(*) Note that the choice of the whitening matrix W is not unique. Common choices include: $W=M^{-1/2}$ (**Mahalanobis whitening**), **Choleksy** decomposition-based whitening, where $W=M^{-1}$ and the eigen-system of M (**PCA** whitening).

Feature Preprocessing: Whitening



PCA / Whitening. **Left**: Original toy, 2-dimensional input data. **Middle**: After performing PCA. The data is centered at zero and then rotated into the eigenbasis of the data covariance matrix. This decorrelates the data (the covariance matrix becomes diagonal). **Right**: Each dimension is additionally scaled by the eigenvalues, transforming the data covariance matrix into the identity matrix. Geometrically, this corresponds to stretching and squeezing the data into an isotropic gaussian blob.



Left: An example set of 49 images. **2nd from Left:** The top 144 out of 3072 eigenvectors. The top eigenvectors account for most of the variance in the data, and we can see that they correspond to lower frequencies in the images. **2nd from Right:** The 49 images reduced with PCA, using the 144 eigenvectors shown here. That is, instead of expressing every image as a 3072-dimensional vector where each element is the brightness of a particular pixel at some location and channel, every image above is only represented with a 144-dimensional vector, where each element measures how much of each eigenvector adds up to make up the image. In order to visualize what image information has been retained in the 144 numbers, we must rotate back into the "pixel" basis of 3072 numbers. Since U is a rotation, this can be achieved by multiplying by U.transpose()[:144,:], and then visualizing the resulting 3072 numbers as the image. You can see that the images are slightly blurrier, reflecting the fact that the top eigenvectors capture lower frequencies. However, most of the information is still preserved. **Right:** Visualization of the "white" representation, where the variance along every one of the 144 dimensions is squashed to equal length. Here, the whitened 144 numbers are rotated back to image pixel basis by multiplying by U.transpose()[:144,:]. The lower frequencies (which accounted for most variance) are now negligible, while the higher frequencies (which account for relatively little variance originally) become exaggerated.

Data Augmentation

• The best way to make an ML model generalize better is to **train it on more data**. Of course, data are limited/expensive.

• One way to get around this problem is to generate synthetic data and add it to the training set.

• This approach is easiest for classification. A classifier needs to take a complicated, high-dimensional input **x** and summarize it with a single category identity *y*. This means that the main task facing a classifier is to be invariant to a wide variety of transformations; we can generate new (**x**, *y*) pairs easily by transforming the **x** inputs in our training set.

Data Augmentation

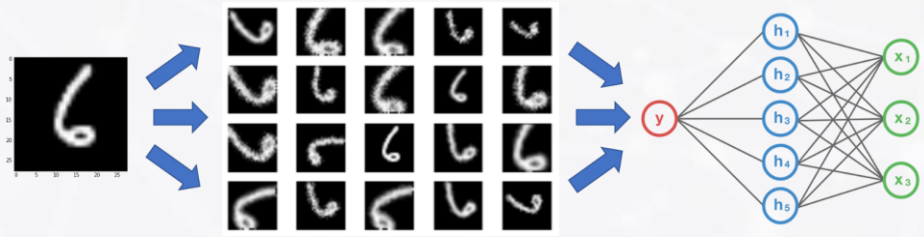
• The best way to make an ML model generalize better is to train it on more data. Of course, data are limited/expensive.

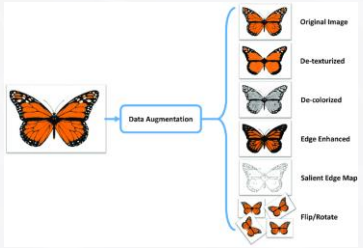
• One way to get around this problem is to generate synthetic data and add it to the training set.

• This approach is easiest for classification. A classifier needs to take a complicated, high-dimensional input **x** and summarize it with a single category identity *y*. This means that the main task facing a classifier is to be invariant to a wide variety of transformations; we can generate new (**x**, *y*) pairs easily by transforming the **x** inputs in our training set.

• Dataset augmentation has been particularly effective for **object recognition**; operations like translating the training images a few pixels in each direction can often greatly improve generalization; many operations such as rotating the image or scaling the image are also quite effective (one needs to be careful that the transformation does not alter the correct image class).

• **Injecting noise** in the input to a NN can also be seen as a form of data augmentation; one way to improve the robustness of a NN is to simply train them with random noise applied to their inputs.

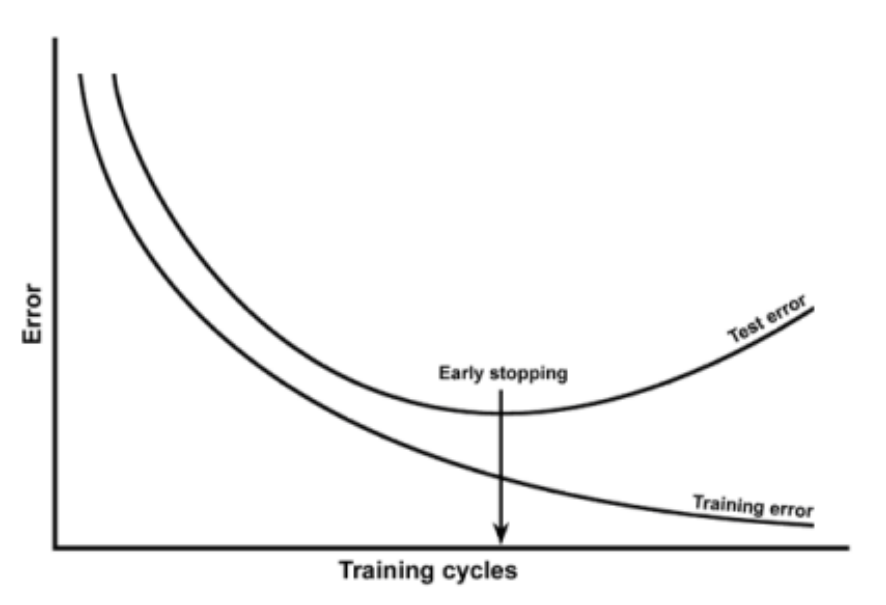




Early Stopping

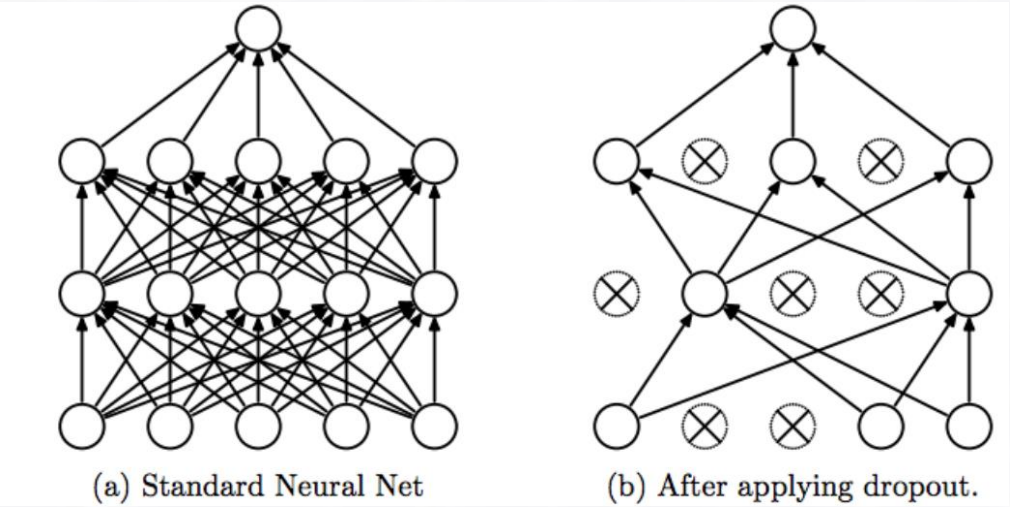
• When training large models with sufficient representation capacity to overfit the task, we often observe that training error decreases steadily over time, but validation set error begins to rise again.

• This means we can obtain a model with better validation set error (and hopefully better test error) by returning to the parameter setting at the point in time with the lowest validation set error. Every time the error on the validation set improves, we store a copy of the model parameters; when the training terminates, we return these parameters, rather than the latest parameters.



* This strategy is known as **early stopping**; it is one of the most common forms of regularization used in deep learning.

Dropout



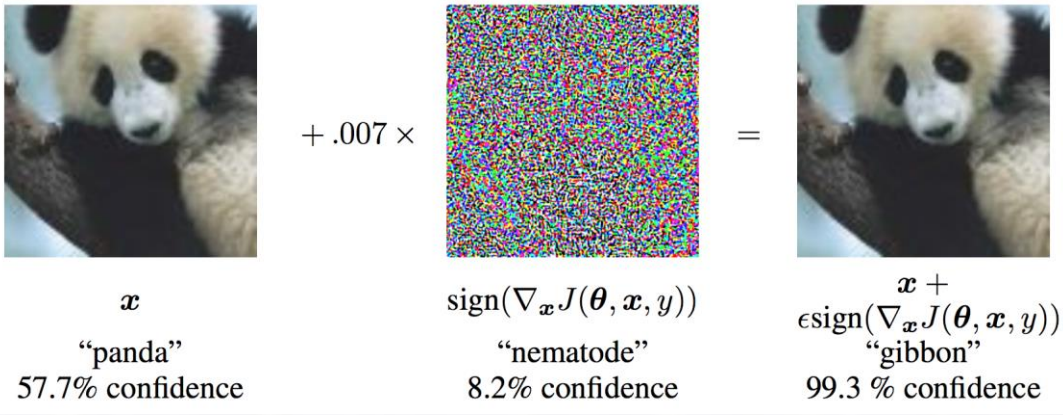
• Dropout (Srivastava et al., 2014) provides a computationally inexpensive but powerful method of regularizing a broad family of models (it is akin to *bagging*).

• Dropout trains the ensemble consisting of all subnetworks that can be formed by removing non-output units from an underlying base network. Recall that to learn with bagging, we define k different models, construct k different datasets by sampling from the training set with replacement, and then train model i on dataset i. Dropout aims to approximate this process, but with an exponentially large number of NNs.

• In practice, each time we load an example into a minibatch for training, we randomly sample a different binary mask to apply to all input and hidden units in the network; the mask is sampled independently for each unit (e.g. 0.8 probability for including an input unit and 0.5 for hidden units).

• In the case of bagging, the models are all independent; for dropout, the models share parameters.

Adversarial Training



• Szegedy et al. (2014) found that even NNs that perform at human level accuracy have a nearly 100 percent error rate on examples that are intentionally construction by using an optimization procedure to search for an input x' near a data point x such that the model output is very different from x' (oftentimes such **adversarial examples** are indiscernible to humans).

• In the context of regularization, one can reduce the error rate on the original i.i.d. test set via **adversarial training** – training on adversarially perturbed examples from the training set.

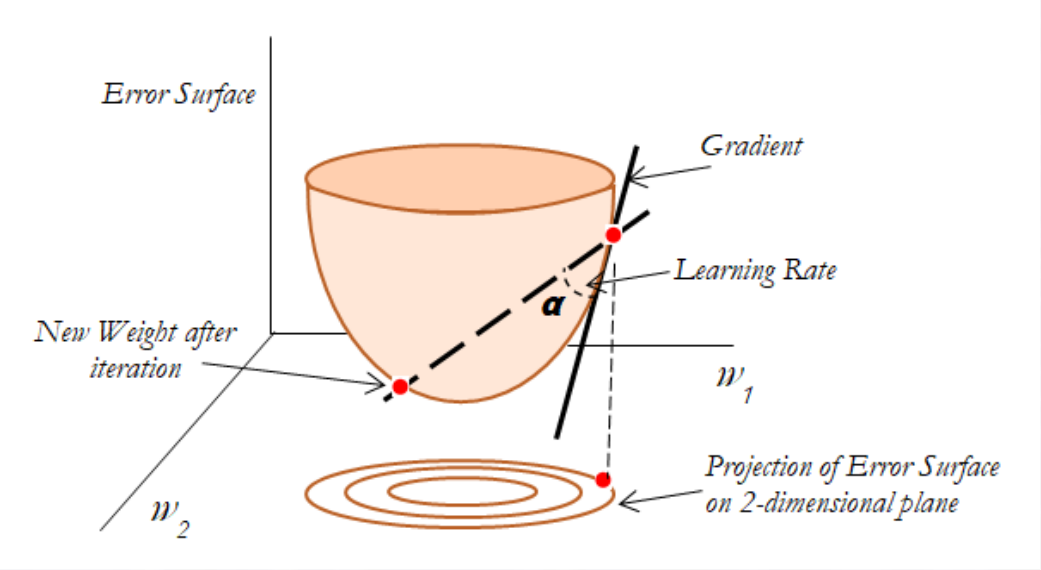
• Goodfellow et al. (2014), showed that one of the primary cause of these adversarial examples is excessive linearity. NNs are primarily built out of linear parts, and so the overall function that they implement proves to be highly linear as a result.

• Adversarial training help to illustrate the power of using a large function family in combination with aggressive regularization – a major theme in contemporary deep learning.

Basic Algorithms: SGD

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while



Basic Algorithms: SGD

• **Stochastic Gradient Descent** (SGD) and its variants are some of the most frequently used optimization algorithms in ML. Using a minibatch of i.i.d. samples, one can obtain an unbiased estimate of the gradient (where examples are drawn from the data-generating distribution).

•A crucial parameter for the SGD algorithm is the **learning rate**, ε. In practice, it is necessary to gradually decrease the learning rate over time. This is because the SGD gradient estimator introduces a source of noise (the random sampling of m training examples) that does not vanish even when we arrive at a minimum.

A sufficient condition to guarantee convergence of SGD is that:

$$\sum_{k=1}^{\infty} \varepsilon_k = \infty \text{ and } \sum_{k=1}^{\infty} \varepsilon_k^2 < \infty$$

In practice, it is common to decay the learning rate linearly until iteration τ:

$$\varepsilon_k = (1-\alpha)\varepsilon_0 + \alpha\varepsilon_\tau \text{ with } \alpha = \frac{k}{\tau}$$

* Note that for SGD, the computation time per update does not grow with the number of training examples. This allows convergence even when the number of training examples becomes very large.

Momentum

• The method of **momentum** is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients.

• The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

• Formally, the momentum algorithm introduces a variable **v** that plays the role of *velocity* – it is the direction and speed at which the parameters move through parameter space. The velocity is set to an exponentially decaying average of the negative gradient.

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}.$$

Momentum

• The method of **momentum** is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients.

• The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

• Formally, the momentum algorithm introduces a variable **v** that plays the role of *velocity* – it is the direction and speed at which the parameters move through parameter space. The velocity is set to an exponentially decaying average of the negative gradient.

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}.$$

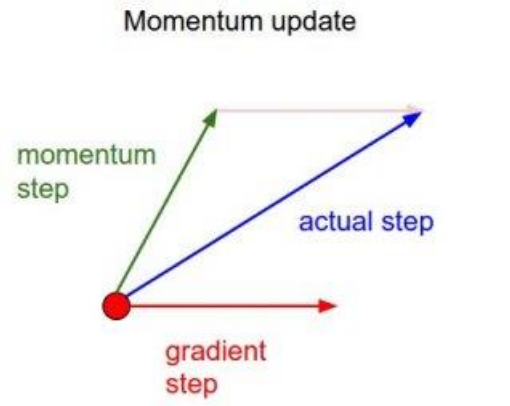
• The name momentum derives from a physical analogy, in which the negative gradient is a force moving a particle through parameter space, according to Newton's laws of motion. If the only force is the gradient of the cost function, then the particle might never come to rest. To resolve this problem, we add one other force, proportional to v(t); in physics terminology this force corresponds to *viscous drag*, as the if the particle must push through a resistant medium such as syrup.

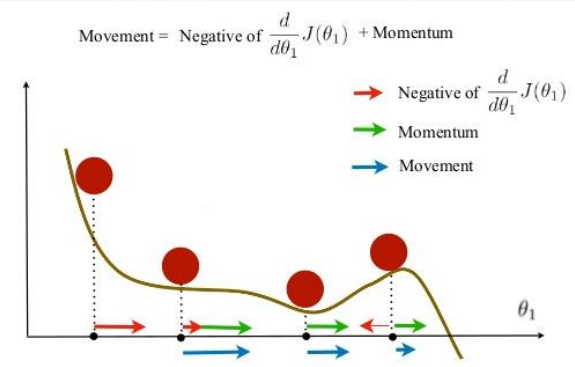
• The velocity **v** accumulates the gradient elements; the larger alpha is relative to epsilon, the more previous gradients affect the current direction.

Momentum

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while





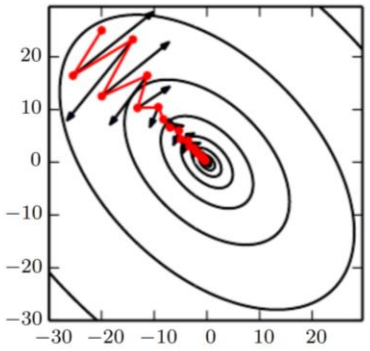


Figure 8.5: Momentum aims primarily to solve two problems: poor conditioning of the Hessian matrix and variance in the stochastic gradient. Here, we illustrate how momentum overcomes the first of these two problems. The contour lines depict a quadratic loss function with a poorly conditioned Hessian matrix. The red path cutting across the contours indicates the path followed by the momentum learning rule as it minimizes this function. At each step along the way, we draw an arrow indicating the step that gradient descent would take at that point. We can see that a poorly conditioned quadratic objective looks like a long, narrow valley or canyon with steep sides. Momentum correctly traverses the canyon lengthwise, while gradient steps waste time moving back and forth across the narrow axis of the canyon. Compare also figure 4.6, which shows the behavior of gradient descent without momentum.

Algorithms with Adaptive Learning Rates

• It is well known that the learning rate is reliably one of the most challenging hyperparameters to set because it significantly affects model performance. The cost function is often highly sensitive to some directions in parameters space and insensitive to others.

• While the momentum algorithm mitigates these issues somewhat, it does so at the expense of introducing another hyperparameter.

• Recently, a number of incremental methods have been introduced that adapt the learning rates of model parameters.

AdaGrad

• The **AdaGrad** algorithm (Duchi et al, 2011) **individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all the historical squared values of the gradient**.

• The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivates have a relatively small decrease in their learning rate. **The net effect is greater progress in the more gently sloped directions of parameter space**.

*Note: empirically, for training DNNs, the accumulation of squared gradients *from the beginning of training* can result in premature and excessive decrease in the effective learning rate.

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, perhaps 10^{-7}, for numerical stability

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

while stopping criterion not met **do**

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

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

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

RMSProp

• The **RMSProp** algorithm (Hinton, 2012) modifies AdaGrad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially-weighted moving average. Where AdaGrad shrinks the learning rate according to the entire history of the squared gradient, **RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl**.

• Empirically, RMSProp has been to shown to be an effective and practical optimization algorithm for DNNs.

Algorithm 8.5 The RMSProp algorithm

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

Adam

• **Adam** (Kingman and Ba, 2014) is another adaptive learning rate optimization algorithm ("adaptive moments"). It can be seen as **a variant on the combination of RMSProp and momentum with several distinctions**.

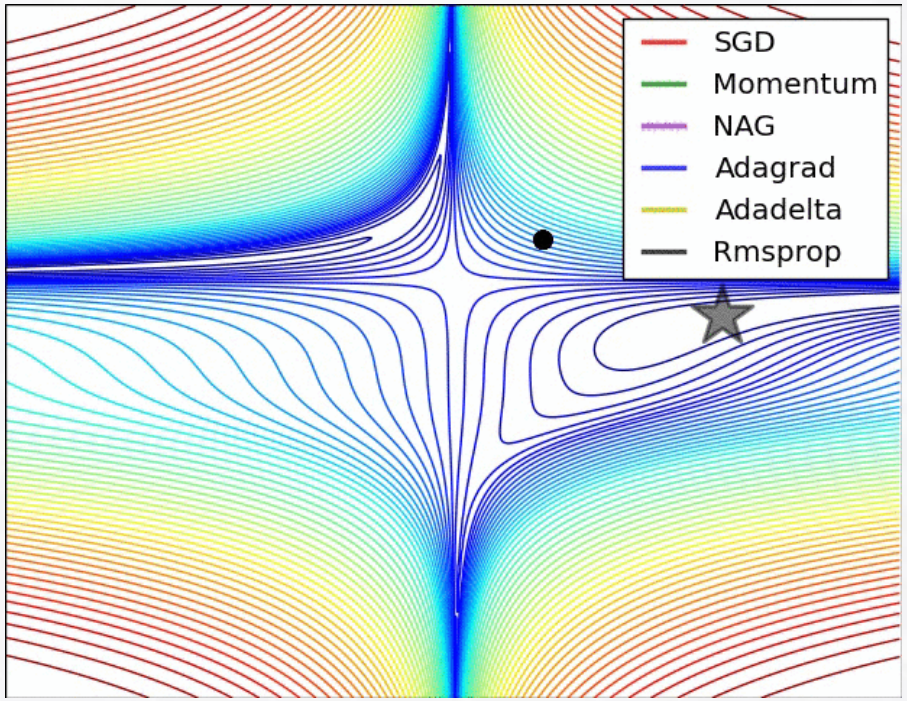
• First, in Adam, **momentum is incorporated** directly as an estimate of the first-order moment (with exponential weighting) of the gradient. Second, Adam includes **bias corrections** to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin.

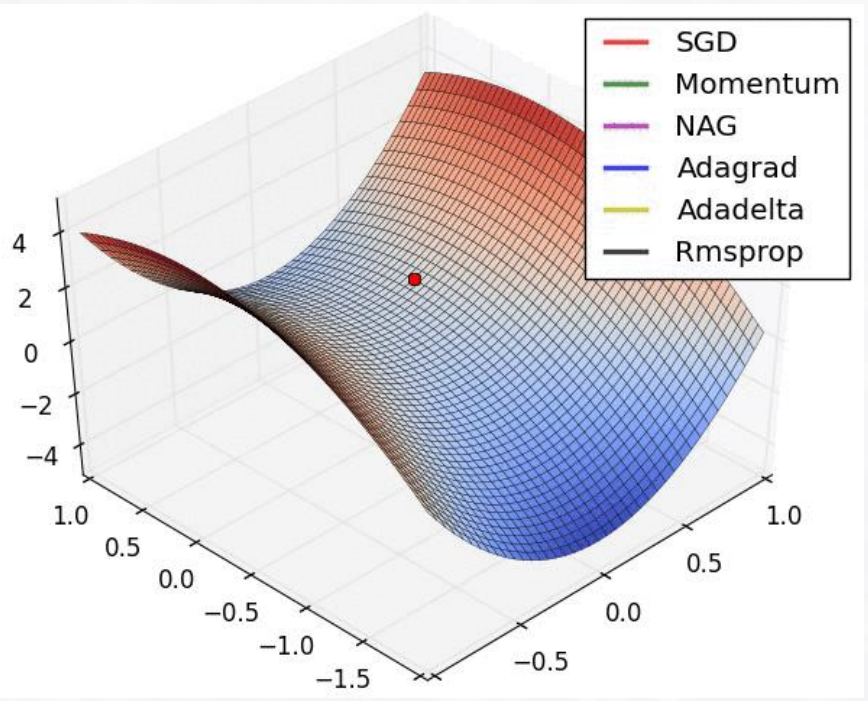
• RMSProp also incorporates an estimate of the (uncentered) second-order moment; however, it lacks the correction factor. Thus, unlike in Adam, the RMSProp second-order moment estimate may have high bias early in training. *__Adam is generally regarded as being fairly robust to the choice of hyperparameters__.

Algorithm 8.7 The Adam algorithm

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

DL Optimization Comparison





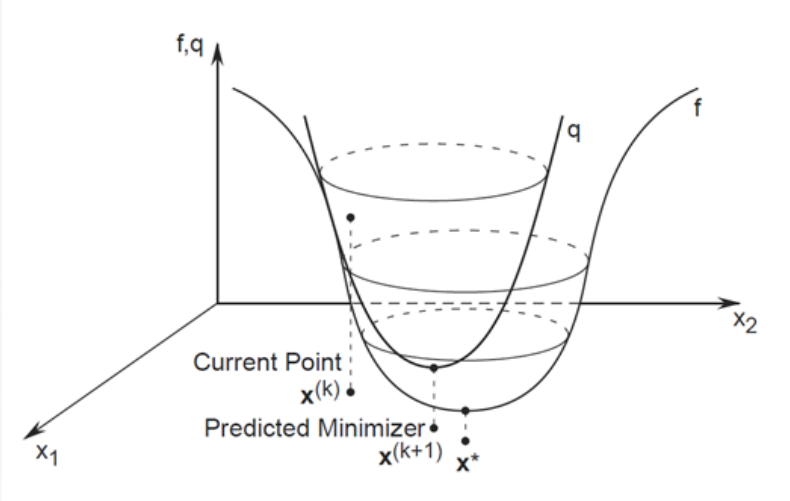
Left: Contours of a loss surface and time evolution of different optimization algorithms. Notice the "overshooting" behavior of momentum-based methods, which make the optimization look like a ball rolling down the hill. **Right:** A visualization of a saddle point in the optimization landscape, where the curvature along different dimension has different signs (one dimension curves up and another down). Notice that SGD has a very hard time breaking symmetry and gets stuck on the top. Conversely, algorithms such as RMSprop will see very low gradients in the saddle direction. Due to the denominator term in the RMSprop update, this will increase the effective learning rate along this direction, helping RMSProp proceed. Images credit: Alec Radford.

Second-Order Methods

• A number of methods have been proposed in recent years for using second-order derivatives for optimization (consider this scenario as incorporating an approximation of the curvature of the loss function into the optimization problem).

• Such methods can partially alleviate some of the problems caused by curvature of the loss function, including cliffs, and the necessity of many course correction steps for hill climbing.

• **Newton's method** is a classical second-order iterative approximation method. In contrast to first-order methods, second-order methods make use of second derivatives (i.e. the curvature of the loss function) to improve optimization.



Second-Order Methods: Newton's Method

• Newton's method is a classical second-order iterative approximation method. In contrast to first-order methods, second-order methods make use of second derivatives (i.e. the curvature of the loss function) to improve optimization.

• Newton's method is an optimization scheme based on using a second-order Taylor series expansion to approximate J(θ) near some point θ_0, ignoring derivatives of higher order:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Where **H** is the Hessian of *J* wrt θ evaluated at θ_0. If we then solve for the critical point of this function, we obtain the Newton parameter update rule:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - H^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

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

Second-Order Methods: Newton's Method

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0) \qquad \boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - H^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

• If the objective function is convex but not quadratic, this update can be iterated, yielding a training algorithm. For surfaces that are not quadratic, as long as the Hessian remains positive definite, Newton's method can be applied iteratively. This implies a two-step procedure: (1) update or compute the inverse Hessian; (2) update the parameters according to the equation above.

* In deep learning, the surface of the objective function is usually non-convex; with many features and potential saddle points, this is a potential problem for Newton's Method.

Second-Order Methods: Newton's Method

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0) \qquad \boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - H^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

• If the objective function is convex but not quadratic, this update can be iterated, yielding a training algorithm. For surfaces that are not quadratic, as long as the Hessian remains positive definite, Newton's method can be applied iteratively. This implies a two-step procedure: (1) update or compute the inverse Hessian; (2) update the parameters according to the equation above.

* In deep learning, the surface of the objective function is usually non-convex; with many features and potential saddle points, this is a potential problem for Newton's Method.

• Commonly, researchers apply a regularization strategy, for which the update becomes (this regularization is used in approximations to Newton's Method including the *Levenberg-Marquardt algorithm*):

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \left[H\left(f(\boldsymbol{\theta}_0) \right) + \alpha \mathbf{I} \right]^{-1} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0)$$

• Beyond the challenges of saddle points, the application of Newton's method for training large NNs is limited by its **significant computational requirements**; ostensibly, Newton's method requires the inversion of a matrix ($O(n^3)$); as a consequence, only networks with a very small number of parameters can be practically trained via Newton's method.

(*) In practice, it is common to apply a second-order method using a "**Hessian-free**" approach, meaning that the full Hessian is either approximated with a low-rank matrix or eigen-vector methods are applied (see **conjugate gradients**).

Second-Order Methods: Newton's Method

Algorithm 8.8 Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$.

Require: Initial parameter $\boldsymbol{\theta}_0$

Require: Training set of m examples

while stopping criterion not met **do**

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

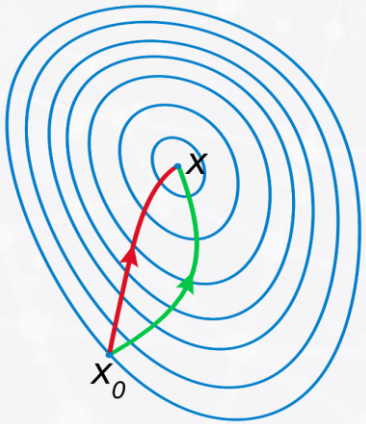
Compute Hessian: $\boldsymbol{H} \leftarrow \frac{1}{m}\nabla^2_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

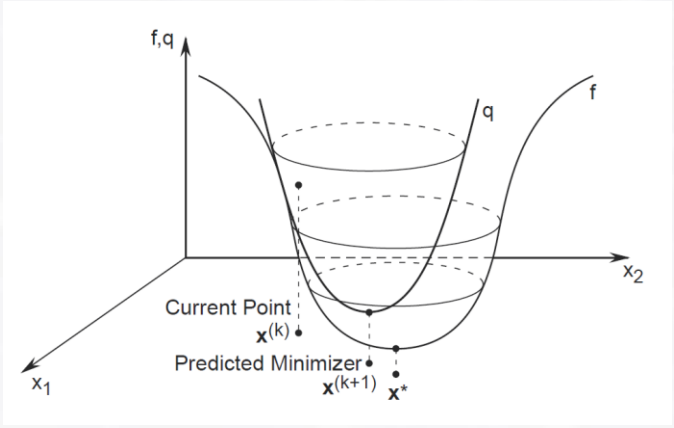
Compute Hessian inverse: $\boldsymbol{H}^{-1}$

Compute update: $\Delta\boldsymbol{\theta} = -\boldsymbol{H}^{-1}\boldsymbol{g}$

Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while





Supplemental Backpropagation Derivation

Backpropagation Algorithm

- Initialize the network weights **w** to small random numbers (e.g., between −0.05 and 0.05).

- Until the termination condition is met, Do:
  - For each (**x**,**t**) ∈ training set, Do:
    1. *Propagate the input forward:*

       - Input **x** to the network and compute the activation h_j of each hidden unit j.

       - Compute the activation o_k of each output unit k.

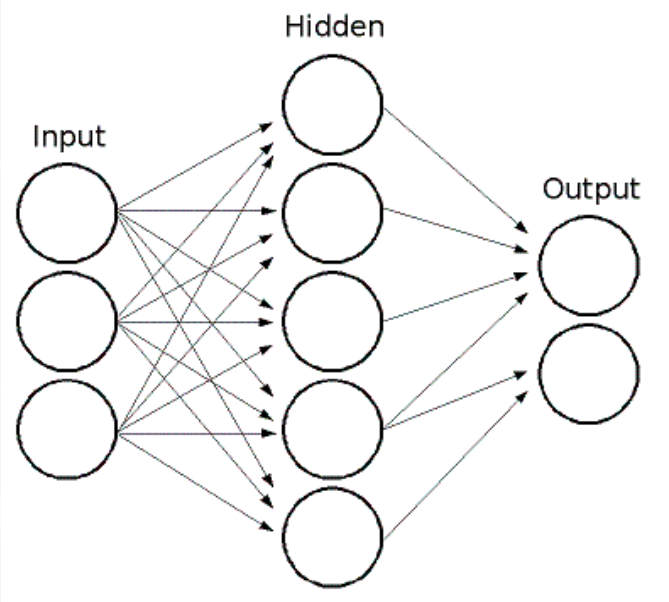
2. *Calculate error terms*

For each **output** unit k, calculate error term δ_k :

$$\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$$

For each **hidden** unit j, calculate error term δ_j:

$$d_j \leftarrow h_j(1-h_j)\left(\sum_{k \in \text{output units}} w_{kj}\, d_k\right)$$



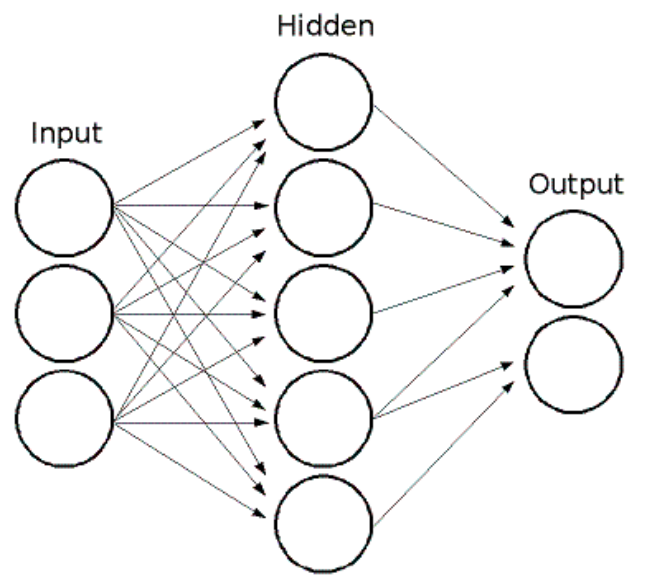
2. *Calculate error terms*

For each **output** unit k, calculate error term δ_k :

$$\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$$

For each **hidden** unit j, calculate error term δ_j :

$$d_j \leftarrow h_j(1-h_j)\left(\sum_{k \in \text{output units}} w_{kj}\, d_k\right)$$



3. Update weights

Hidden to Output layer: For each weight w_{kj}

$$w_{kj} \leftarrow w_{kj} - \Delta w_{kj}$$

where

$$\Delta w_{kj} = \eta \delta_k h_j$$

Input to Hidden layer: For each weight w_{ji}

$$w_{ji} \leftarrow w_{ji} - \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \delta_j x_i$$

Backpropagation Algorithm (BP)

- **Forwards Phase**: compute the activation of each neuron in the hidden layers and outputs using:

$$h_j = s\left(\sum_{i \in \text{input layer}} w_{ji} x_i + w_{j0}\right) \qquad o_k = s\left(\sum_{j \in \text{hidden layer}} w_{kj} h_j + w_{k0}\right)$$

- **Backwards pass**
- Compute the error at the output using: $\delta_k \leftarrow o_k(1-o_k)(t_k - o_k)$
- Compute the error at the hidden layer(s) using: $d_j \leftarrow h_j(1 - h_j)\left(\sum_{k \in \text{output units}} w_{kj}\, d_k\right)$
- Update the output layer weights using: $w_{kj} \leftarrow w_{kj} - \Delta w_{kj}$
  where $\Delta w_{kj} = h d_k h_j$
- Update the hidden layer weights using: $w_{ji} \leftarrow w_{ji} - \Delta w_{ji}$
  where $\Delta w_{ji} = h d_j x_i$
- (If using *sequential updating*) randomize the order of the input vectors so that you don't train in exactly the same order each iteration.

Backprop Example

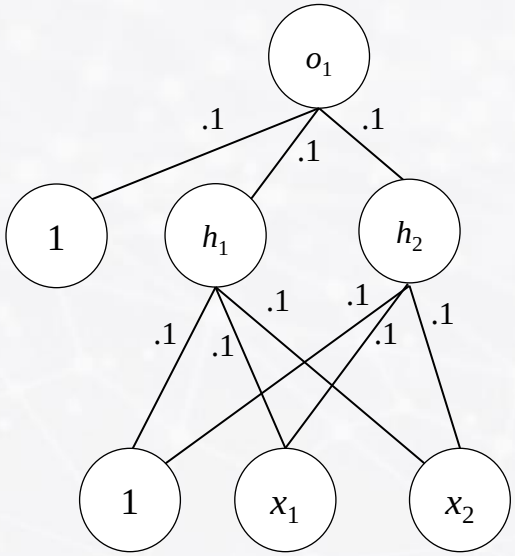
Training set:

1 0 Label: 0.9

0 1 Label: -.3

Test set:

1 1 Label: .8



Training set:

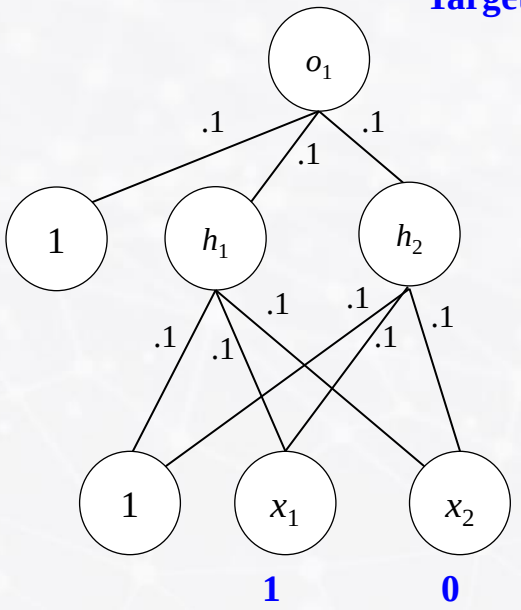
1 0 Label: .9

0 1 Label: -.3

Test set:

1 1 Label: .8

Target: .9



Training set:

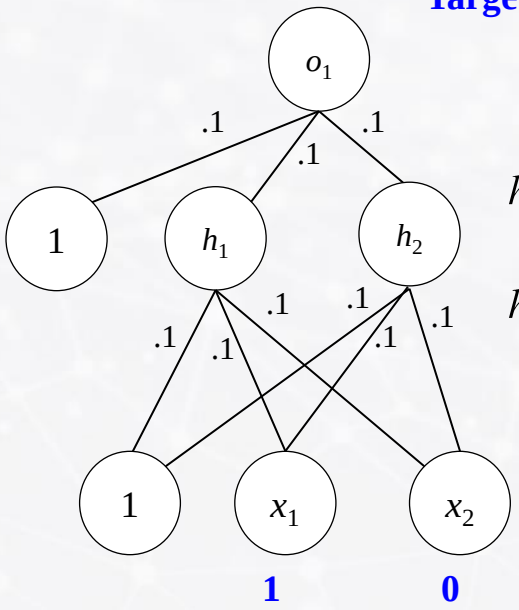
1 0 Label: .9

0 1 Label: -.3

Test set:

1 1 Label: .8

Target: .9



$$h_1 = \sigma\big((1)(.1)+(1)(.1)+(0)(.1)\big) = \sigma(.2) = \frac{1}{1+e^{-.2}} = .55$$

$$h_2 = \sigma\big((1)(.1)+(1)(.1)+(0)(.1)\big) = \sigma(.2) = \frac{1}{1+e^{-.2}} = .55$$

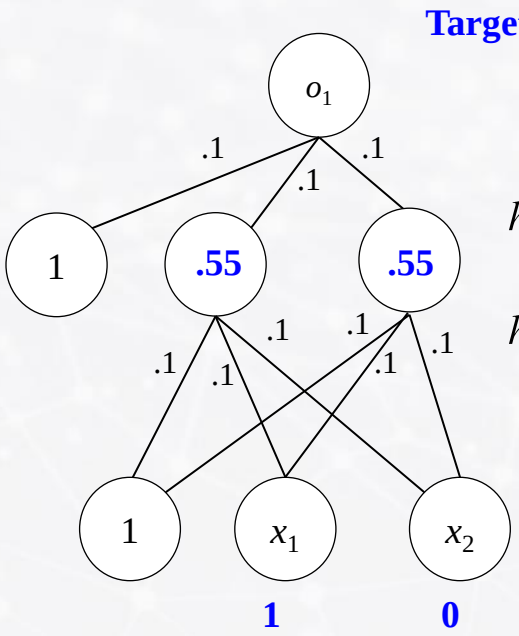
Training set:

1 0 Label: .9

0 1 Label: -.3

Test set:

1 1 Label: .8



$$h_1 = \sigma\big((1)(.1)+(1)(.1)+(0)(.1)\big) = \sigma(.2) = \frac{1}{1+e^{-.2}} = .55$$

$$h_2 = \sigma\big((1)(.1)+(1)(.1)+(0)(.1)\big) = \sigma(.2) = \frac{1}{1+e^{-.2}} = .55$$

"Forward Phase" – hidden layers

Training set:

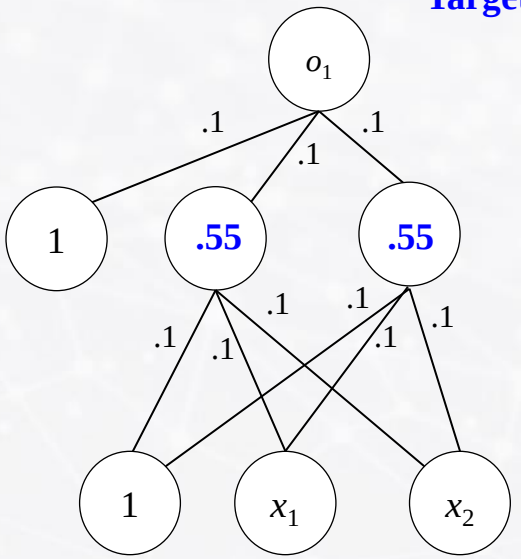
1 0 Label: .9

0 1 Label: -.3

Test set:

1 1 Label: .8

Target: .9



1 0

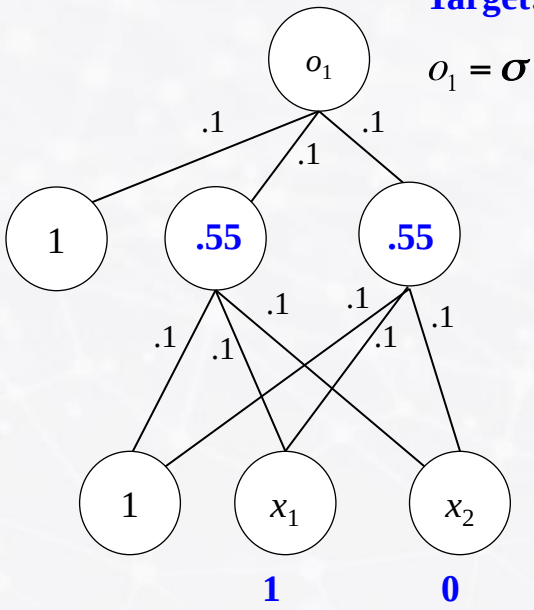
Training set:

1 0 Label: .9

0 1 Label: -.3

Test set:

1 1 Label: .8



Target: .9

$$o_1 = \sigma\left((1)(.1)+(.55)(.1)+(.55)(.1)\right) = \sigma(.21) = \frac{1}{1+e^{-.21}} = .552$$

"Forward Phase" – output layer

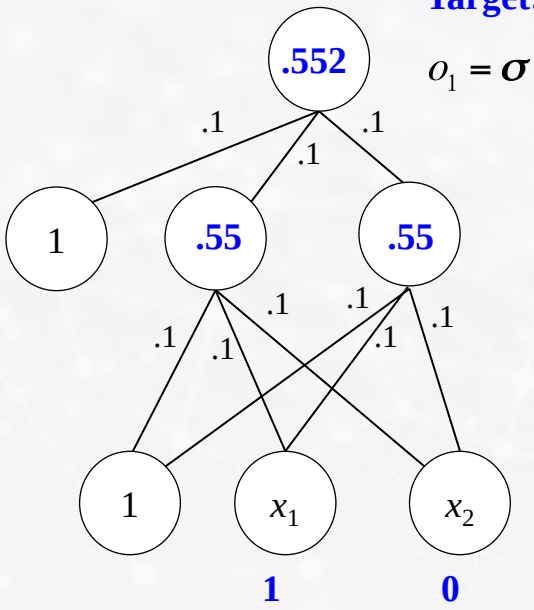
Training set:

1 0 Label: .9

0 1 Label: -.3

Test set:

1 1 Label: .8



Target: .9

$$o_1 = \sigma\left((1)(.1)+(.55)(.1)+(.55)(.1)\right) = \sigma(.21) = \frac{1}{1+e^{-.21}} = .552$$

"Forward Phase" – output layer

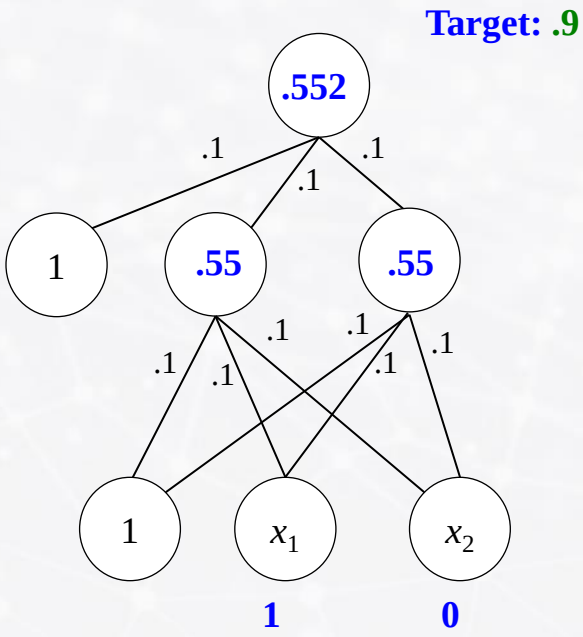
Training set:

1 0 Label: .9

0 1 Label: -.3

Test set:

1 1 Label: .8



Output weight Updates

$$\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$$

Hidden weight Updates

$$d_j \leftarrow h_j(1-h_j)\left(\sum_{k \in \text{output units}} w_{kj}\, d_k\right)$$

"Backward Phase"

Training set:

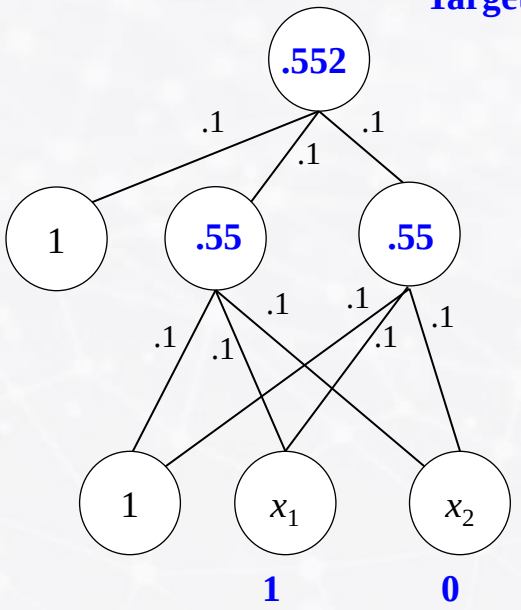
1 0 Label: .9

0 1 Label: -.3

Test set:

1 1 Label: .8

Target: .9



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)(.086) = .002$$

"Backward Phase"

$$\delta_k \leftarrow o_k(1-o_k)(t_k - o_k)$$

Output weight Updates

$$\delta_j \leftarrow h_j(1-h_j)\left(\sum_{k \in \text{output units}} w_{kj}\,\delta_k\right)$$

Hidden weight Updates

Training set:

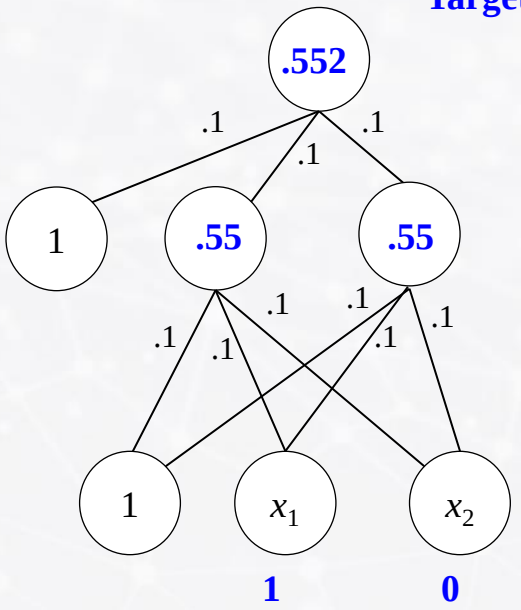
1 0 Label: .9

0 1 Label: -.3

Test set:

1 1 Label: .8

Target: .9



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)(.086) = .002$$

"Backward Phase"

$$\delta_k \leftarrow o_k(1-o_k)(t_k - o_k)$$

Output weight Updates

$$\delta_j \leftarrow h_j(1-h_j)\left(\sum_{k \in \text{output units}} w_{kj}\,\delta_k\right)$$

Hidden weight Updates

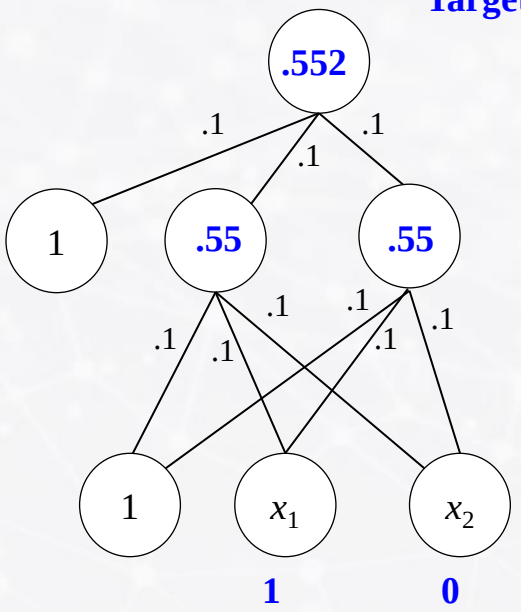
Training set:

1 **0** **Label: Positive**

0 **1** **Label: Negative**

Test set:

1 **1** **Label: Positive**



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)(.086) = .002$$

$$\Delta w_{kj} = \eta \delta_k h_j + \alpha \Delta w_{kj}^{t-1}$$

***Update hidden-to-output weights* (*learning rate* = 0.2; *momentum* = 0.9):**

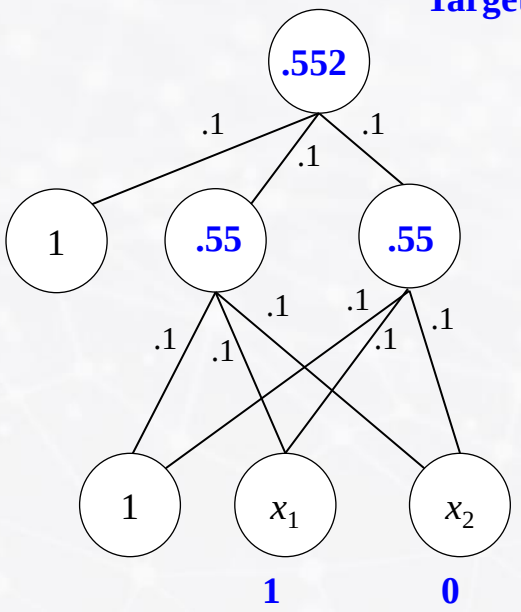
Training set:

1 0 Label: .9

0 1 Label: -.3

Test set:

1 1 Label: .8



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)(.086) = .002$$

$$\Delta w_{kj} = \eta \delta_k h_j + \alpha \Delta w_{kj}^{t-1}$$

***Update hidden-to-output weights* (*learning rate* = 0.2; *momentum* = 0.9): Hidden unit j=1**

$$\Delta w_{k=1,j=0}^{1} = (.2)(.086)(1) + (.9)(0) = .0172$$

$$\Delta w_{k=1,j=1}^{1} = (.2)(.086)(.55) + (.9)(0) = .0095$$

$$\Delta w_{k=1,j=2}^{1} = (.2)(.086)(.55) + (.9)(0) = .0095$$

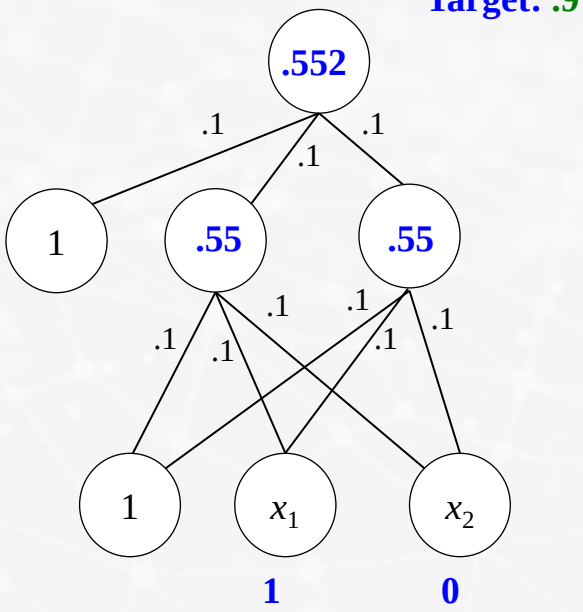
Training set:

1 0 Label: .9

0 1 Label: -.3

Test set:

1 1 Label: .8



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)(.086) = .002$$

$$\Delta w_{kj} = \eta \delta_k h_j + \alpha \Delta w_{kj}^{t-1}$$

***Update hidden-to-output weights* (*learning rate* = 0.2; *momentum* = 0.9):**

$$\Delta w^1_{k=1,j=0} = (.2)(.086)(1) + (.9)(0) = .0172 \qquad w^1_{k=1,j=0} = .1 - .0172 = .0828$$

$$\Delta w^1_{k=1,j=1} = (.2)(.086)(.55) + (.9)(0) = .0095 \qquad w^1_{k=1,j=1} = .1 - .0095 = .0905$$

$$\Delta w^1_{k=1,j=2} = (.2)(.086)(.55) + (.9)(0) = .0095 \qquad w^1_{k=1,j=2} = .1 - .0095 = .0905$$

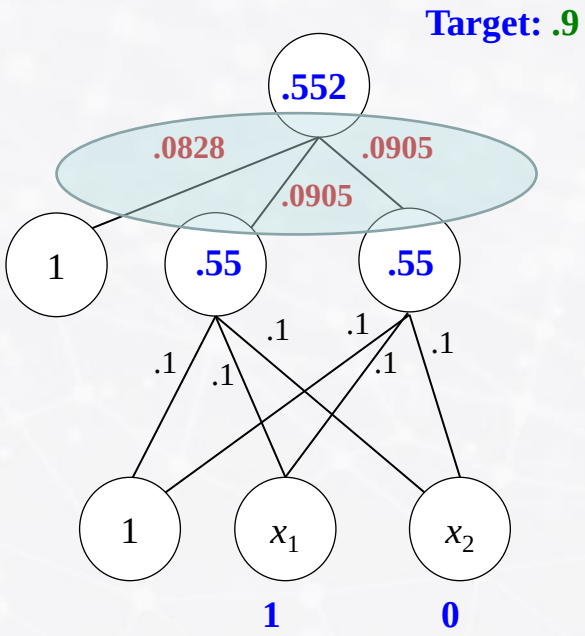
Training set:

1 0 Label: .9

0 1 Label: -.3

Test set:

1 1 Label: .8



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)(.086) = .002$$

***Update hidden-to-output weights* (*learning rate* = 0.2; *momentum* = 0.9):**

$$\Delta w^1_{k=1,j=0} = (.2)(.086)(1) + (.9)(0) = .0172 \qquad w^1_{k=1,j=0} = .1 - .0172 = .0828$$

$$\Delta w^1_{k=1,j=1} = (.2)(.086)(.55) + (.9)(0) = .0095 \qquad w^1_{k=1,j=1} = .1 - .0095 = .0905$$

$$\Delta w^1_{k=1,j=2} = (.2)(.086)(.55) + (.9)(0) = .0095 \qquad w^1_{k=1,j=2} = .1 - .0095 = .0905$$

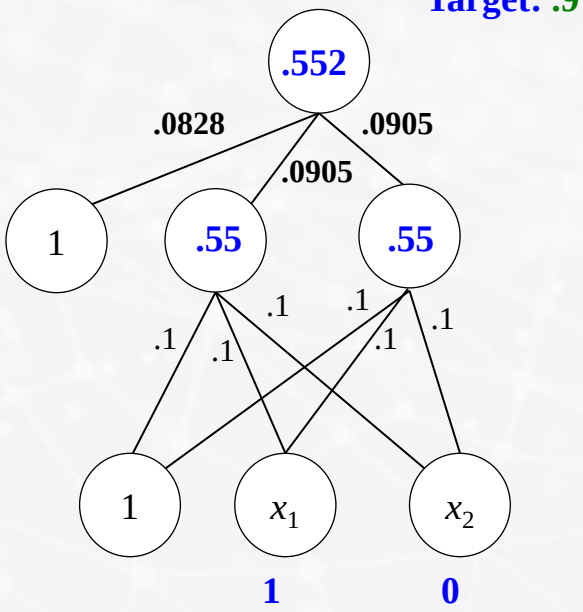
Training set:

1 0 Label: .9

0 1 Label: -.3

Test set:

1 1 Label: .8



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)(.086) = .002$$

$$\Delta w_{ji} = \eta \delta_j x_i + \alpha \Delta w_{ji}^{t-1}$$

Update input-to-hidden weights (learning rate = 0.2; momentum = 0.9):

$$\Delta w^1_{j=1,i=0} = (.2)(.002)(1) + (.9)(0) = .0004 \qquad w^1_{j=1,i=0} = .1 - .0004 = .9996$$

$$\Delta w^1_{j=1,i=1} = (.2)(.002)(1) + (.9)(0) = .0004 \qquad w^1_{j=1,i=1} = .1 - .0004 = .9996$$

$$\Delta w^1_{j=1,i=2} = (.2)(.002)(0) + (.9)(0) = 0 \qquad w^1_{j=1,i=2} = .1$$

Training set:

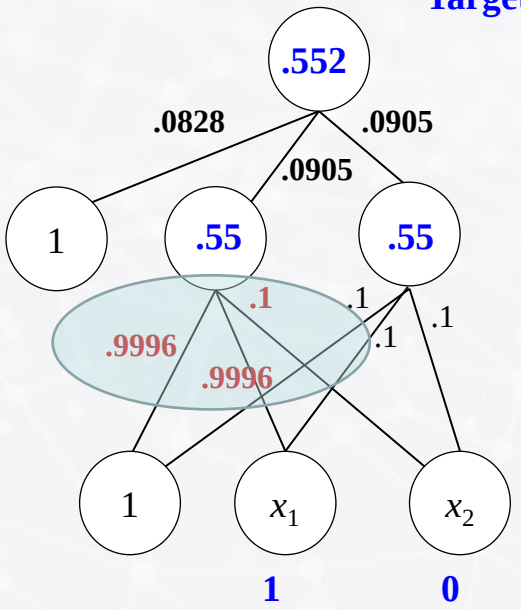
1 0 Label: .9

0 1 Label: -.3

Test set:

1 1 Label: .8

Target: .9



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)(.086) = .002$$

$$\Delta w_{ji} = \eta \delta_j x_i + \alpha \Delta w_{ji}^{t-1}$$

Update input-to-hidden weights (learning rate = 0.2; momentum = 0.9):

$$\Delta w^1_{j=1,i=0} = (.2)(.002)(1) + (.9)(0) = .0004 \qquad w^1_{j=1,i=0} = .1 - .0004 = .9996$$

$$\Delta w^1_{j=1,i=1} = (.2)(.002)(1) + (.9)(0) = .0004 \qquad w^1_{j=1,i=1} = .1 - .0004 = .9996$$

$$\Delta w^1_{j=1,i=2} = (.2)(.002)(0) + (.9)(0) = 0 \qquad w^1_{j=1,i=2} = .1$$

Training set:

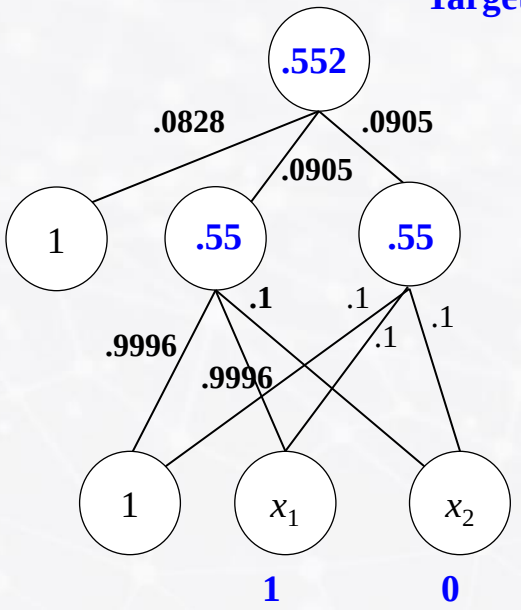
1 0 Label: .9

0 1 Label: -.3

Test set:

1 1 Label: .8

Target: .9



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)(.086) = .002$$

$$\Delta w_{ji} = \eta \delta_j x_i + \alpha \Delta w_{ji}^{t-1}$$

***Update input-to-hidden weights* (*learning rate* = 0.2; *momentum* = 0.9): Hidden unit j=2**

$$\Delta w_{j=2,i=0}^{1} = (.2)(.002)(1) + (.9)(0) = .0004 \qquad w_{j=2,i=0}^{1} = .1 - .0004 = .9996$$

$$\Delta w_{j=2,i=1}^{1} = (.2)(.002)(1) + (.9)(0) = .0004 \qquad w_{j=2,i=1}^{1} = .1 - .0004 = .9996$$

$$\Delta w_{j=2,i=2}^{1} = (.2)(.002)(0) + (.9)(0) = 0 \qquad w_{j=2,i=2}^{1} = .1$$

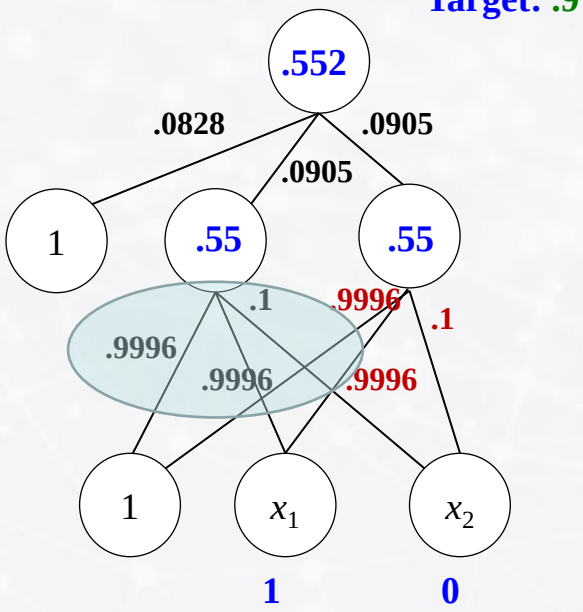
Training set:

1 **0** **Label: .9**

0 **1** **Label: -.3**

Test set:

1 **1** **Label: .8**



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)(.086) = .002$$

Training set:

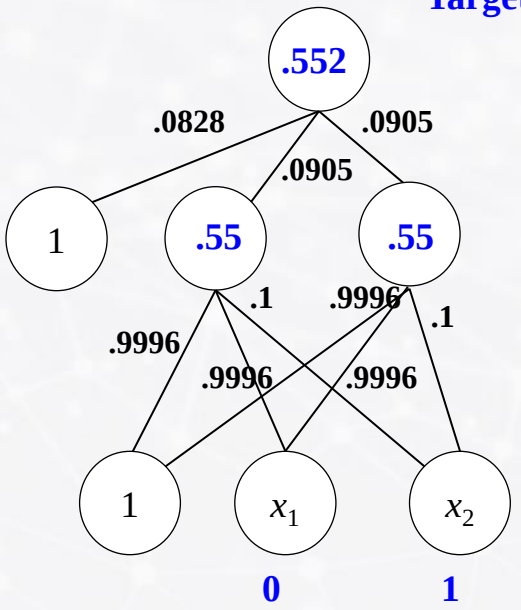
1 0 Label: .9

0 1 Label: -.3

Test set:

1 1 Label: .8

Target: -.3

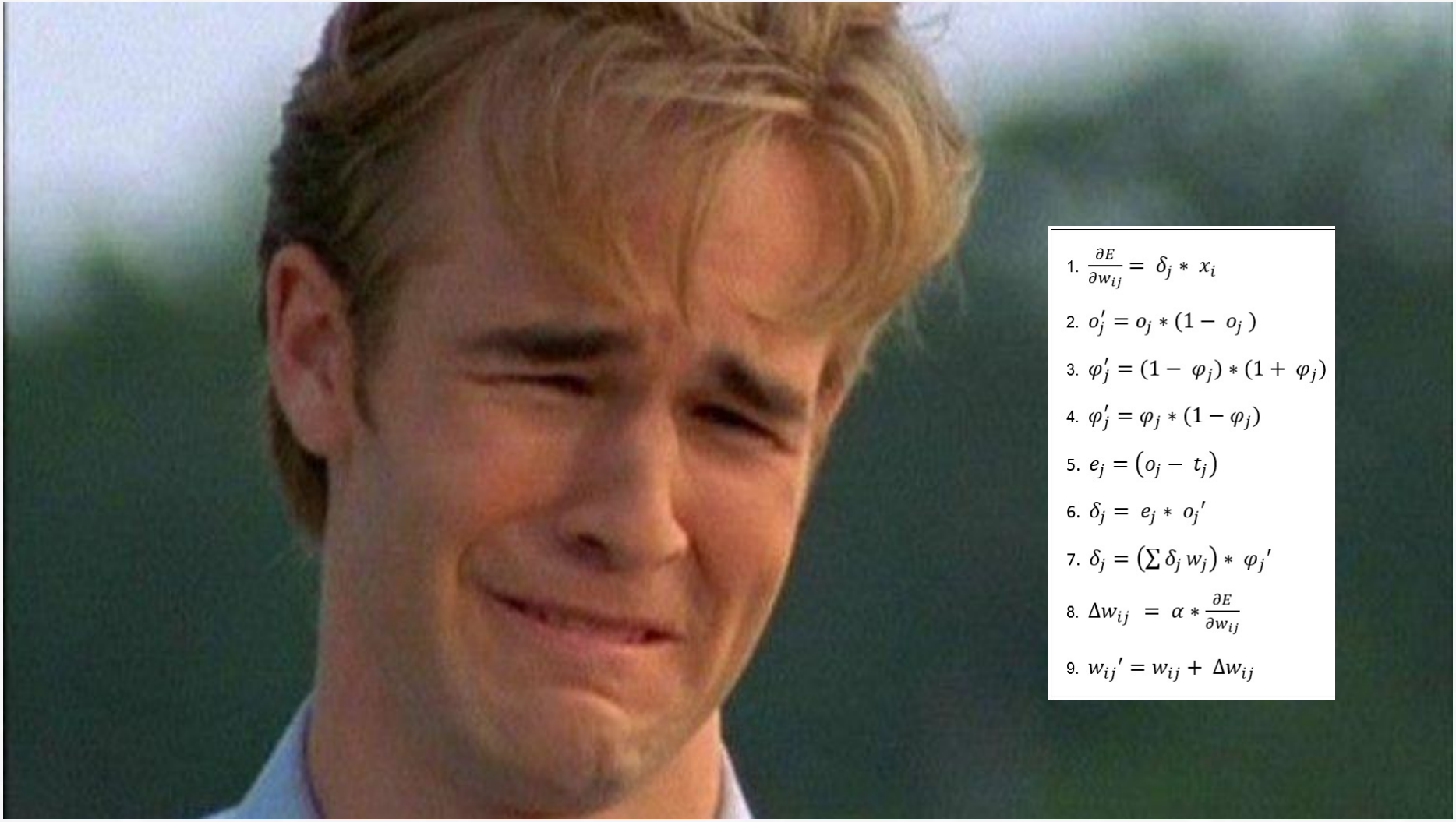


Note: This is time step 2, so momentum term will be nonzero…

Another detailed backprop example:
https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

(*) Here is a derivation (later slides contain a derivation with visuals) of BP (note our text also has a derivation pp. 101-108). Time permitting, I'll walk us through this. If you require further details don't hesitate to ask for help.

(*) "*Will this be on the exam?*" **No**, but understanding the material at this level makes you a better person – moreover, it will make your friends envious, your mother will love you more, and strangers at cocktail parties will be drawn to you like a magnet. You're welcome.


1. $\frac{\partial E}{\partial w_{ij}} = \delta_j * x_i$
2. $o_j' = o_j * (1 - o_j)$
3. $\varphi_j' = (1 - \varphi_j) * (1 + \varphi_j)$
4. $\varphi_j' = \varphi_j * (1 - \varphi_j)$
5. $e_j = (o_j - t_j)$
6. $\delta_j = e_j * o_j'$
7. $\delta_j = (\sum \delta_j w_j) * \varphi_j'$
8. $\Delta w_{ij} = \alpha * \frac{\partial E}{\partial w_{ij}}$
9. $w_{ij}' = w_{ij} + \Delta w_{ij}$

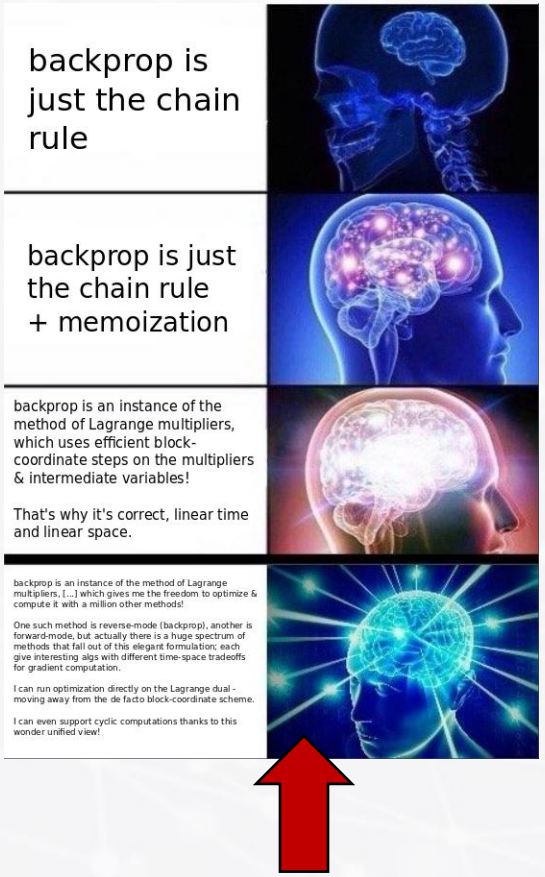
Backprop Derivation

What do we need to derive the backpropagation (BP) algorithm?

Only a basic knowledge of differential Calculus!

(*) Recall that we will use BP to update weights in both the hidden layer(s) and the output layer of our NN.

(*) We use the chain rule to "propagate" the error back through the network (following the "forward phase").



obligatory backprop meme

Backprop Derivation

(*) We'll call the current input **x** (a vector) and the output **y**; the activation function (throughout the network) will be denoted $g(\cdot)$.

(*) For simplicity, let's assume the NN contains only a single hidden layer (BP extends naturally for more layers); denote the weights of the network **v** and **w**, for the first and second layers respectively.

(*) Recall that "learning" entails tuning the weights of the network.

Backprop Derivation

(*) We wish to minimize the error function:

$$E(w) = \frac{1}{2}\sum_{k=1}^{N}(y_k - t_k)^2 = \frac{1}{2}\sum_{k=1}^{N}\left[g\left(\sum_{i=0}^{L} w_{ik} x_i\right) - t_k \right]^2$$

Where **y** is the output, t is the target; **N** is the data set size and **L** is the number of nodes (in a given layer).

(*) We use *gradient descent*. In particular, we wish to know how the error function changes with respect to the different weights:

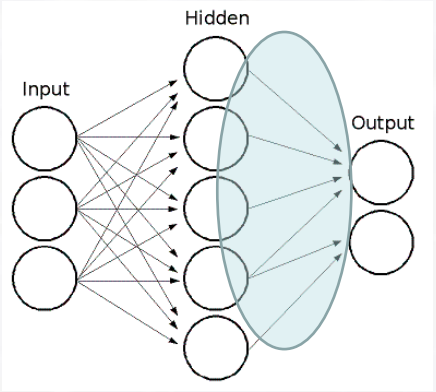
$$\frac{\partial E}{\partial w_{\varsigma\kappa}}$$

*Note: $j = \zeta, k = \kappa$ are fixed indices.

Backprop Derivation

(*) Let $a = g(h) = \frac{1}{1+e^{-h}}$ (the sigmoid function); recall that: $\boxed{a' = a(1-a)}$

(*) Using the *chain rule*, we have:



$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_{\kappa}} \frac{\partial h_{\kappa}}{\partial w_{\varsigma\kappa}}$$

where: $h_{\kappa} = \sum_{j=0}^{M} w_{j\kappa} a_j$

The input to output-layer neuron κ

The equation above says that **the error at the output changes as we vary the second-layer weights** as a function of the error change with respect to the input to the output neurons and the change in the input with respect to the weights.

Backprop Derivation

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_{\kappa}} \frac{\partial h_{\kappa}}{\partial w_{\varsigma\kappa}}$$

(*) Consider the (2)nd factor:

$$\frac{\partial h_{\kappa}}{\partial w_{\varsigma\kappa}} = \frac{\partial \sum_{j=0}^{M} w_{j\kappa} a_j}{\partial w_{\varsigma\kappa}} = \sum_{j=0}^{M} \frac{\partial w_{j\kappa} a_j}{\partial w_{\varsigma\kappa}} = a_{\zeta}$$

Why?

Backprop Derivation

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_\kappa}\frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}}$$

$$\frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}} = a_\zeta$$

(*) Consider the (2)nd factor:

$$\frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}} = \frac{\partial \sum_{j=0}^{M} w_{j\kappa} a_j}{\partial w_{\varsigma\kappa}} = \sum_{j=0}^{M} \frac{\partial w_{j\kappa} a_j}{\partial w_{\varsigma\kappa}} = a_\zeta$$

(*) Last step holds because $\frac{\partial w_{j\kappa}}{\partial w_{\varsigma\kappa}} = 0$, except in the case: $j = \zeta$

Backprop Derivation

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_{\kappa}} \frac{\partial h_{\kappa}}{\partial w_{\varsigma\kappa}}$$

(*) Consider the (1)st factor, which we short-hand as follows:

$$\delta_O(\kappa) = \frac{\partial E}{\partial h_{\kappa}}$$

(*) By the chain rule, we have:

$$\delta_O(\kappa) = \frac{\partial E}{\partial h_{\kappa}} = \frac{\partial E}{\partial y_{\kappa}} \frac{\partial y_{\kappa}}{\partial h_{\kappa}}$$

$$y_{\kappa} = g\left(h_{\kappa}^{output}\right) = g\left(\sum_{j=0}^{M} w_{j\kappa} a_j^{hidden}\right)$$

Note: $\mathbf{h_K}$ signifies the value of the output neuron prior to activation, whereas $\mathbf{y_K}$ denotes the value of the output neuron after activation.

Backprop Derivation

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_{\kappa}} \frac{\partial h_{\kappa}}{\partial w_{\varsigma\kappa}}$$

(*) Consider the (1)st factor, which we short-hand as follows:

$$\delta_O(\kappa) = \frac{\partial E}{\partial h_{\kappa}}$$

(*) By the chain rule, we have:

$$\delta_O(\kappa) = \frac{\partial E}{\partial h_{\kappa}} = \frac{\partial E}{\partial y_{\kappa}} \frac{\partial y_{\kappa}}{\partial h_{\kappa}}$$

(*) Also, note that: $y_{\kappa} = g\left(h_{\kappa}^{output}\right) = g\left(\sum_{j=0}^{M} w_{j\kappa} a_j^{hidden}\right)$

Backprop Derivation

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_\kappa}\frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}}$$

$$\delta_O(\kappa) = \frac{\partial E}{\partial h_\kappa} = \frac{\partial E}{\partial y_\kappa}\frac{\partial y_\kappa}{\partial h_\kappa} \qquad y_\kappa = g\left(h_\kappa^{output}\right) = g\left(\sum_{j=0}^{M} w_{j\kappa} a_j^{hidden}\right)$$

Continuing…

$$\delta_O(\kappa) = \frac{\partial E}{\partial g\left(h_\kappa^{output}\right)}\frac{\partial g\left(h_\kappa^{output}\right)}{\partial\left(h_\kappa^{output}\right)} = \frac{\partial E}{\partial g\left(h_\kappa^{output}\right)} g'\left(h_\kappa^{output}\right)$$

Backprop Derivation

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_\kappa}\frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}}$$

$$\delta_O(\kappa) = \frac{\partial E}{\partial h_\kappa} = \frac{\partial E}{\partial y_\kappa}\frac{\partial y_\kappa}{\partial h_\kappa} \qquad y_\kappa = g\left(h_\kappa^{output}\right) = g\left(\sum_{j=0}^{M} w_{j\kappa} a_j^{hidden}\right)$$

Continuing…

$$\delta_O(\kappa) = \frac{\partial E}{\partial g\left(h_\kappa^{output}\right)}\frac{\partial g\left(h_\kappa^{output}\right)}{\partial\left(h_\kappa^{output}\right)} = \frac{\partial E}{\partial g\left(h_\kappa^{output}\right)} g'\left(h_\kappa^{output}\right)$$

$$= \frac{\partial}{\partial g\left(h_\kappa^{output}\right)}\left[\frac{1}{2}\sum_{k=1}^{N}\left(g\left(h_\kappa^{output}\right) - t_k\right)^2\right] g'\left(h_\kappa^{output}\right)$$

$$E(w) = \frac{1}{2}\sum_{k=1}^{N}\left(y_k - t_k\right)^2 = \frac{1}{2}\sum_{k=1}^{N}\left[g\left(\sum_{i=0}^{L} w_{ik} x_i\right) - t_k\right]^2$$

Backprop Derivation

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_\kappa}\frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}}$$

$$\delta_O(\kappa) = \frac{\partial E}{\partial h_\kappa} = \frac{\partial E}{\partial y_\kappa}\frac{\partial y_\kappa}{\partial h_\kappa} \qquad y_\kappa = g\left(h_\kappa^{output}\right) = g\left(\sum_{j=0}^{M} w_{j\kappa} a_j^{hidden}\right)$$

Continuing…

$$\delta_O(\kappa) = \frac{\partial E}{\partial g\left(h_\kappa^{output}\right)}\frac{\partial g\left(h_\kappa^{output}\right)}{\partial\left(h_\kappa^{output}\right)} = \frac{\partial E}{\partial g\left(h_\kappa^{output}\right)} g'\left(h_\kappa^{output}\right)$$

$$= \frac{\partial}{\partial g\left(h_\kappa^{output}\right)}\left[\frac{1}{2}\sum_{k=1}^{N}\left(g\left(h_\kappa^{output}\right) - t_k\right)^2\right] g'\left(h_\kappa^{output}\right)$$

$$= \left(g\left(h_\kappa^{output}\right) - t_k\right) g'\left(h_\kappa^{output}\right)$$

Backprop Derivation

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_\kappa}\frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}}$$

$$\delta_O(\kappa) = \frac{\partial E}{\partial h_\kappa} = \frac{\partial E}{\partial y_\kappa}\frac{\partial y_\kappa}{\partial h_\kappa} \qquad y_\kappa = g\left(h_\kappa^{output}\right) = g\left(\sum_{j=0}^{M} w_{j\kappa} a_j^{hidden}\right)$$

Continuing…

$$\delta_O(\kappa) = \frac{\partial E}{\partial g\left(h_\kappa^{output}\right)}\frac{\partial g\left(h_\kappa^{output}\right)}{\partial\left(h_\kappa^{output}\right)} = \frac{\partial E}{\partial g\left(h_\kappa^{output}\right)} g'\left(h_\kappa^{output}\right)$$

$$= \frac{\partial}{\partial g\left(h_\kappa^{output}\right)}\left[\frac{1}{2}\sum_{k=1}^{N}\left(g\left(h_\kappa^{output}\right) - t_k\right)^2\right] g'\left(h_\kappa^{output}\right)$$

$$= \left(g\left(h_\kappa^{output}\right) - t_k\right) g'\left(h_\kappa^{output}\right) = \left(y_k - t_k\right) g'\left(h_\kappa^{output}\right)$$

Backprop Derivation

In Summary…

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_{\kappa}} \frac{\partial h_{\kappa}}{\partial w_{\varsigma\kappa}}$$

$$= \left(y_k - t_k \right) g' \left(h_{\kappa}^{output} \right) a_{\zeta}$$

Backprop Derivation

In Summary…

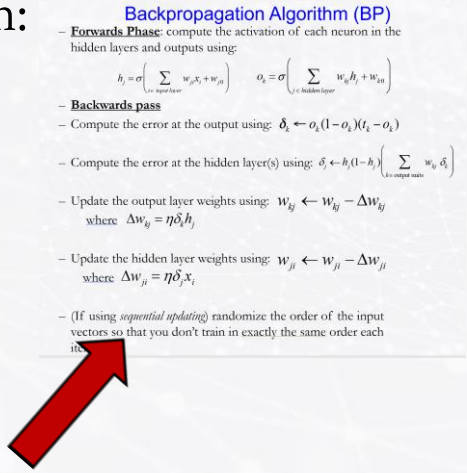
$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_\kappa}\frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}}$$

$$= \left(y_k - t_k\right) g'\left(h_\kappa^{output}\right) a_\zeta$$

(*) Recall, a "gradient descent" based weight update has the form:

$$w_{\varsigma\kappa} \leftarrow w_{\varsigma\kappa} - \eta \frac{\partial E}{\partial w_{\varsigma\kappa}}$$

$$= w_{\varsigma\kappa} - \eta\left(y_k - t_k\right) g'\left(h_\kappa^{output}\right) a_\zeta$$


Backpropagation Algorithm (BP)

– **Forwards Phase**: compute the activation of each neuron in the hidden layers and outputs using:

$$h_j = \sigma\left(\sum_{i \in \text{input layer}} w_{ji} x_i + w_{j0}\right) \qquad o_k = \sigma\left(\sum_{j \in \text{hidden layer}} w_{kj} h_j + w_{k0}\right)$$

– **Backwards pass**

– Compute the error at the output using: $\delta_k \leftarrow o_k(1-o_k)(t_k - o_k)$

– Compute the error at the hidden layer(s) using: $\delta_j \leftarrow h_j(1-h_j)\left(\sum_{k \in \text{output units}} w_{kj}\,\delta_k\right)$

– Update the output layer weights using: $w_{kj} \leftarrow w_{kj} - \Delta w_{kj}$
where $\Delta w_{kj} = \eta \delta_k h_j$

– Update the hidden layer weights using: $w_{ji} \leftarrow w_{ji} - \Delta w_{ji}$
where $\Delta w_{ji} = \eta \delta_j x_i$

– (If using *sequential updating*) randomize the order of the input vectors so that you don't train in exactly the same order each it

(*) Cool, but this doesn't look like the formulas for BP you showed us before.

Backprop Derivation

$$w_{\varsigma\kappa} \leftarrow w_{\varsigma\kappa} - \eta \frac{\partial E}{\partial w_{\varsigma\kappa}}$$

$$= w_{\varsigma\kappa} - \eta \left(y_k - t_k \right) g' \left(h_\kappa^{output} \right) a_\zeta$$

(*) Recall that g is the sigmoid! So what's our new formula?

Backprop Derivation

$$w_{\varsigma\kappa} \leftarrow w_{\varsigma\kappa} - \eta \frac{\partial E}{\partial w_{\varsigma\kappa}}$$

$$= w_{\varsigma\kappa} - \eta \left(y_k - t_k \right) g' \left(h_\kappa^{output} \right) a_\zeta$$

$$= w_{\varsigma\kappa} - \eta \left(y_k - t_k \right) y_k \left(1 - y_k \right) a_\zeta$$

$$\frac{ds(z)}{dz} = s(z) \times (1 - s(z))$$

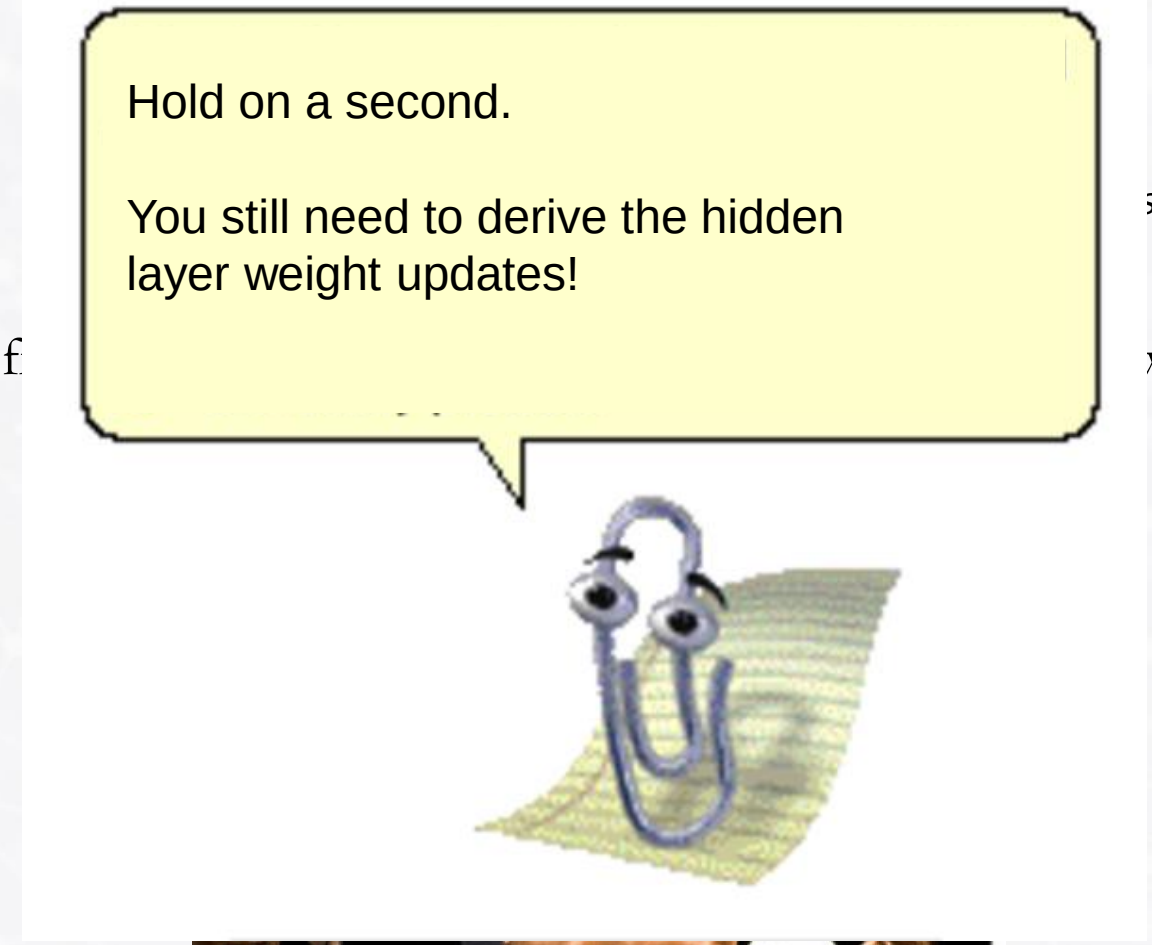
(*) This is the final formula for the BP update for the output layer weights!

Backprop Derivation

$\frac{\partial E}{\partial}$

$s(z) \times (1 - s(z))$

(*) This is the f… weights!



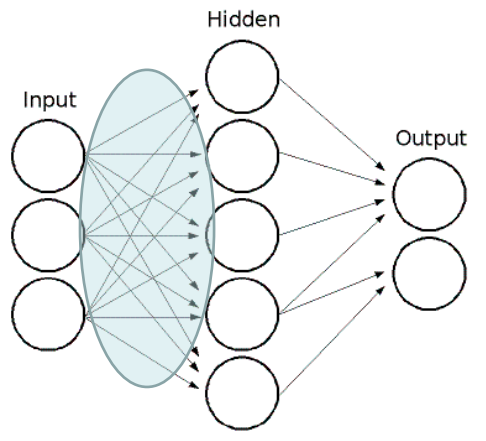
Backprop Derivation

(*)*Short version* of **input-to-hidden layer weight updates** for BP:

We compute: $$\delta_h(\zeta) = \frac{\partial E}{\partial h_\zeta^{hidden}} = \sum_{k=1}^{K} \frac{\partial E}{\partial h_k^{output}} \frac{\partial h_k^{output}}{\partial h_\zeta^{hidden}} = \sum_{k=1}^{K} \delta_O(k) \frac{\partial h_k^{output}}{\partial h_\zeta^{hidden}}$$

(*) This formula comes from the fact that each hidden node contributes to the activation of all the output nodes, and so we need to consider all of these contributions.

(*) From here, using the chain rule, differential properties of the sigmoid and the NN topology, it is not difficult (as we did before, analogously), to show:



$$\delta_h(\zeta) = a_\zeta(1-a_\zeta)\sum_{k=1}^{K} \delta_O(\kappa) w_\zeta$$

Backprop Derivation

$$\delta_h(\zeta) = a_\zeta\left(1-a_\zeta\right)\sum_{k=1}^{K_N}\delta_O(\kappa)\,w_\zeta$$

(*)This yields the following update rule for vertex v_t:

$$v_t \leftarrow v_t - \eta\frac{\partial E}{\partial v_t}$$

$$= v_t - \eta a_\zeta\left(1-a_\zeta\right)\left(\sum_{k=1}^{K_N}\delta_O(\kappa)\,w_\zeta\right)x_t$$

Derivation complete! (at least for NNs with one hidden layer)

Fin