Neural Networks I

Dr. Alex Williams

October 26, 2020

COSC 425: Introduction to Machine Learning
Fall 2020 (CRN: 44874)

Today's Agenda

We will address:

1. Feedforward Neural Networks
2. Recurrent Neural Networks

	Sept 4	ID Learning II	
4	Sept 7	[No Lecture]	
	Sept 9	Perceptron I	[HD] Ch.4
	Sept 11	Perceptron II	
5	Sept 14	Linear Models I	[HD] Ch.7

Perceptron

Learn weights for features.

Learning with Neurons

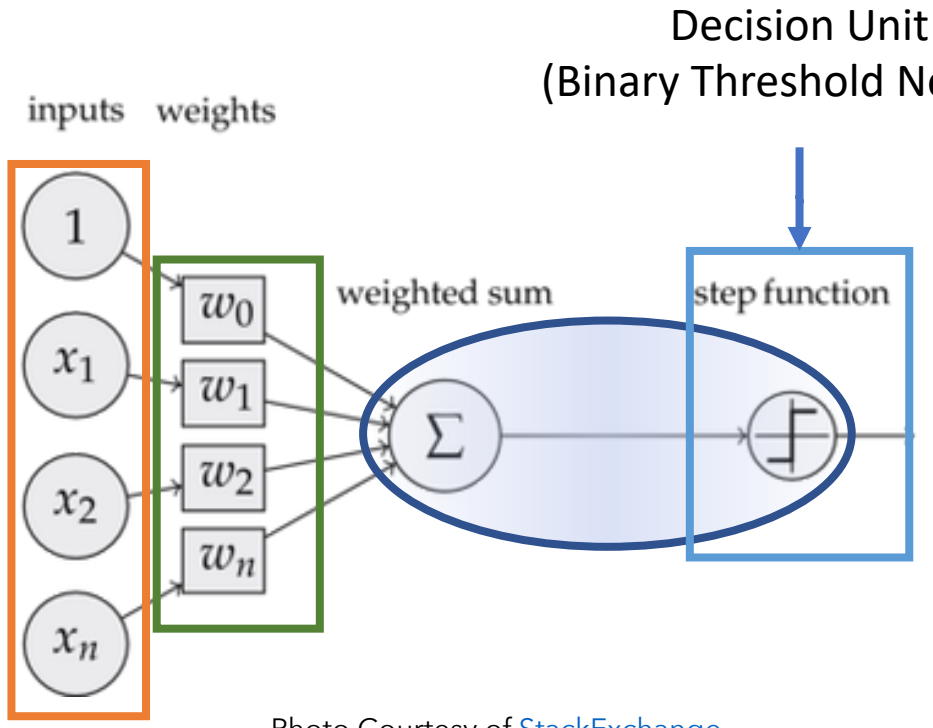


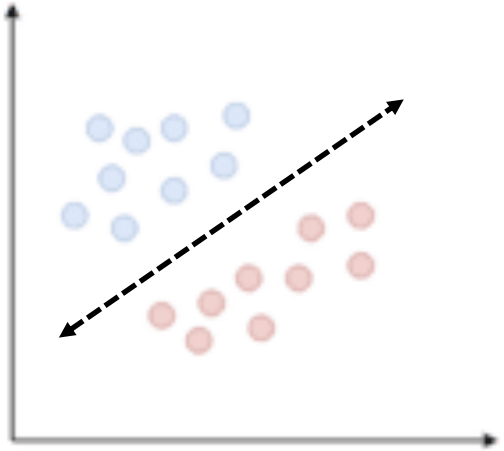
Photo Courtesy of StackExchange.

Perceptron: Usage

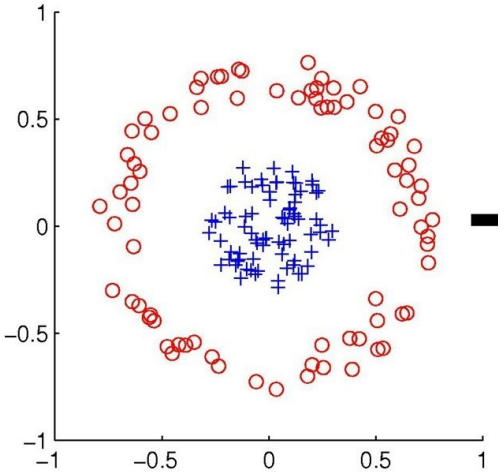
The Perceptron algorithm is useful for scenarios where your data is **linearly separable**.

→ Plot 1 is linearly separable.
A line can be drawn between red and blue classes.

→ Plot 2 is not linearly separable.
A line cannot be drawn to separate classes.



Plot 1



Plot 2

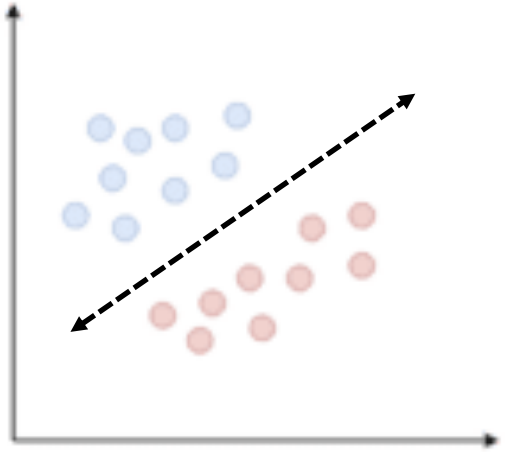
Note: We'll briefly cover how to get around inseparability later.

Perceptron: Algorithm

Let's revisit and think about Perceptron algorithmically in 2d space.
→ **Note**: 2D space indicates we are using two features.

Input: A vector of X feature vectors.
(Vector size is always # of dimensions + 1).

$x = [1, x_1, x_2]$. (Note: Each feature has one of these!)
→ 1 is the bias (i.e. constant term).



Parameters: The algorithm has two parameters:
→ A vector of weights for each input.
$\omega = [\omega_0, \omega_1, \omega_2]$

→ A hyperparameter for "how fast Perceptron converges" to the line that best separates the data.
η = Learning Rate.

Perceptron: Example

Let's do an example in 2d space.
→ **Note**: 2D space indicates we are using two features.

Start with Small, Arbitrary Weights in ω.
→ Take the dot product of the ω vector and *x* vector.

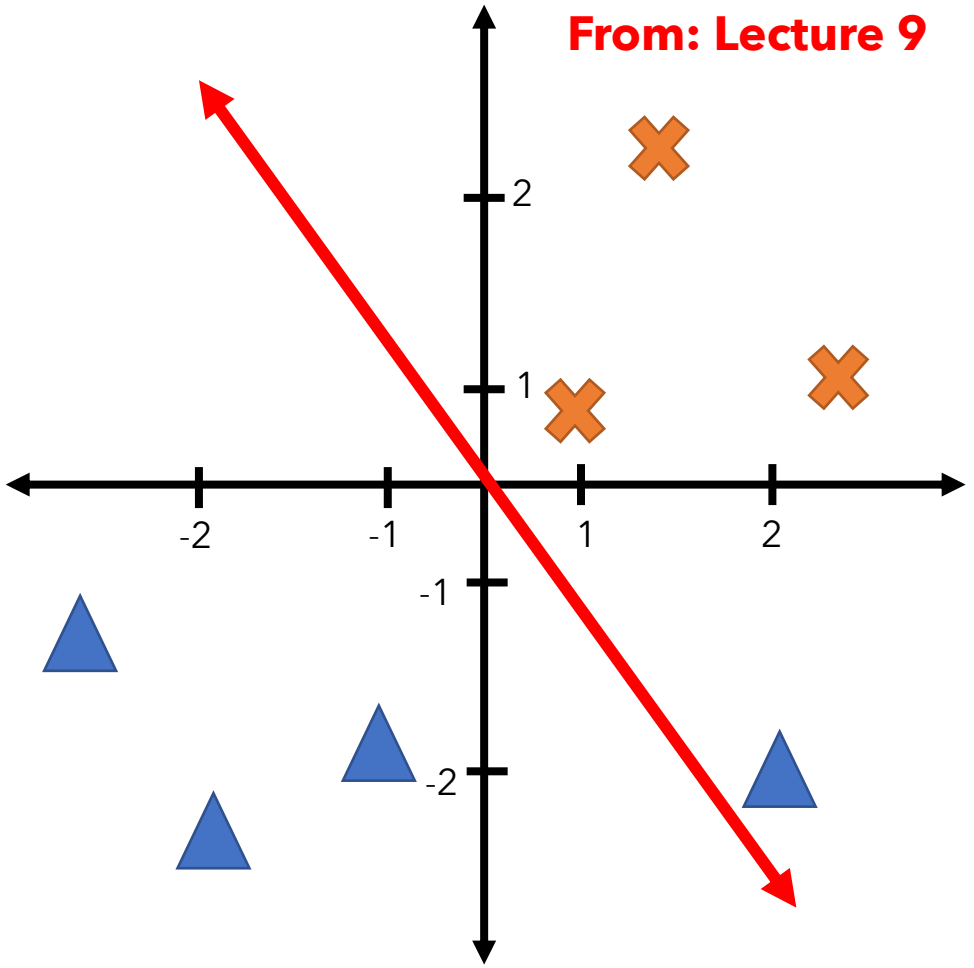
$$X = \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix} \qquad \omega = \begin{pmatrix} 0 \\ 1 \\ 0.5 \end{pmatrix}$$

$$0 \cdot 1 + 1 \cdot x_1 + 0.5 \cdot x_2$$

$$x_1 + \frac{x_2}{2} > 0$$

$$x_2 > -2x_1$$

Our Boundary!



Perceptron: Example

New Weights? New Dot Product!

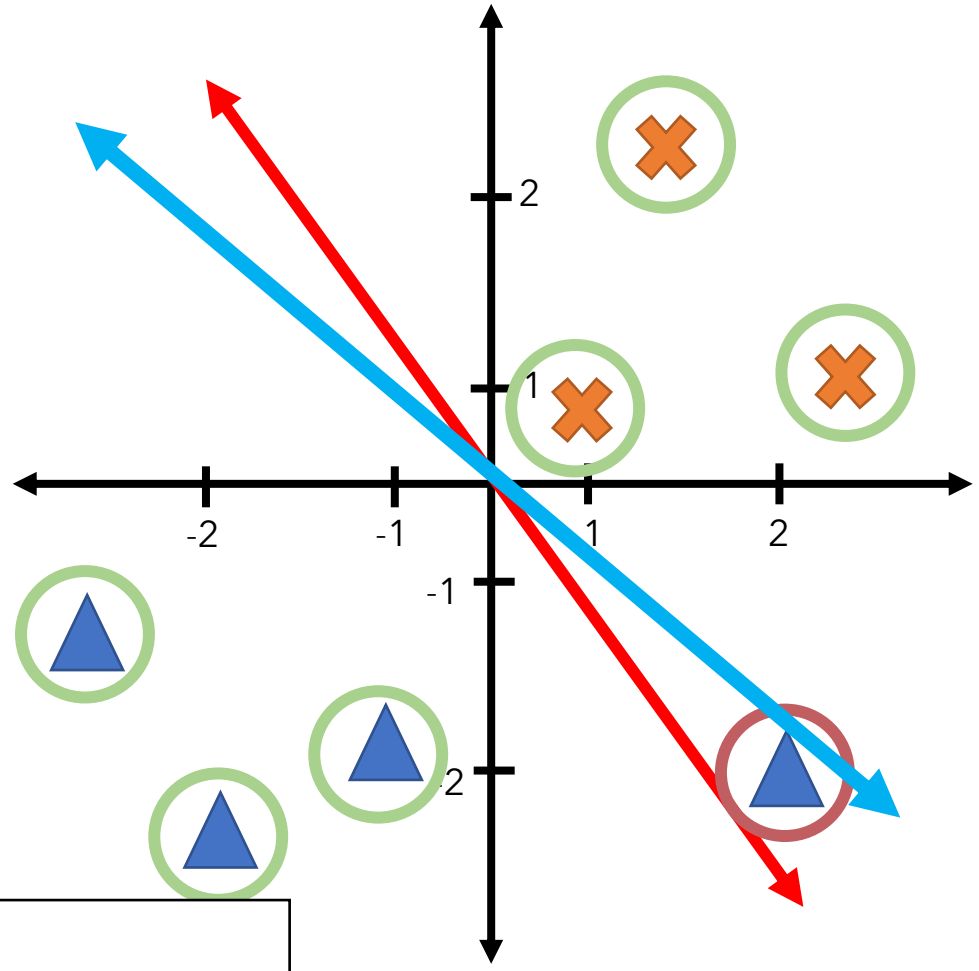
→ Now, it's time to calculate a new linear boundary..

$$X = \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix} \qquad \omega = \begin{pmatrix} -0.2 \\ 0.6 \\ 0.9 \end{pmatrix}$$

$$\begin{aligned} & -0.2 \cdot 1 \\ & + 0.6 \cdot x_1 \\ & + 0.9 \cdot x_2 \end{aligned}$$

$$-0.2 + 0.6 \cdot x_1 + 0.9 \cdot x_2$$

Reduced: $x_2 > -\frac{2}{3}x_1 + \frac{2}{9}$



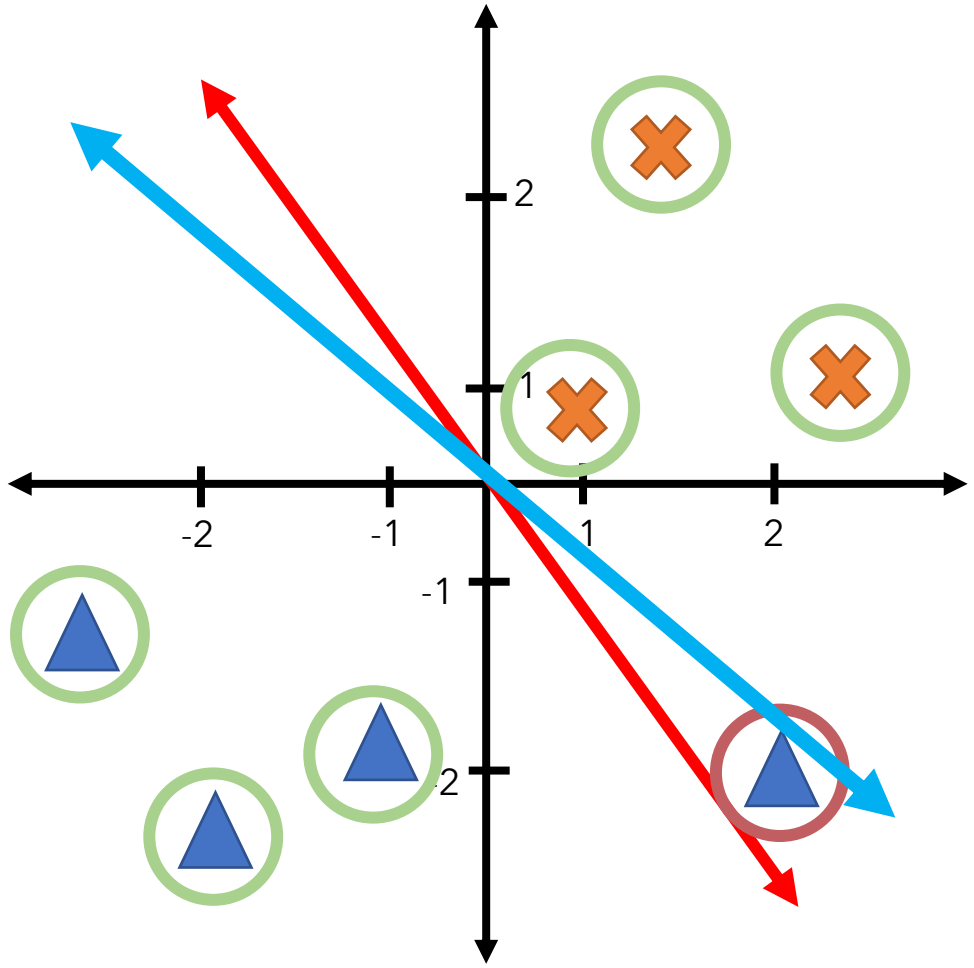
Perceptron: Example

More than 2 Features? Generally no problem.
→ Extend your **x** and **ω** vectors accordingly.

$$X = \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \qquad \omega = \begin{pmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix}$$

An example in 3-dimensional space.

→ **Remember:** Size is always Num of Dimensions + 1



This plot is a 2D space example.

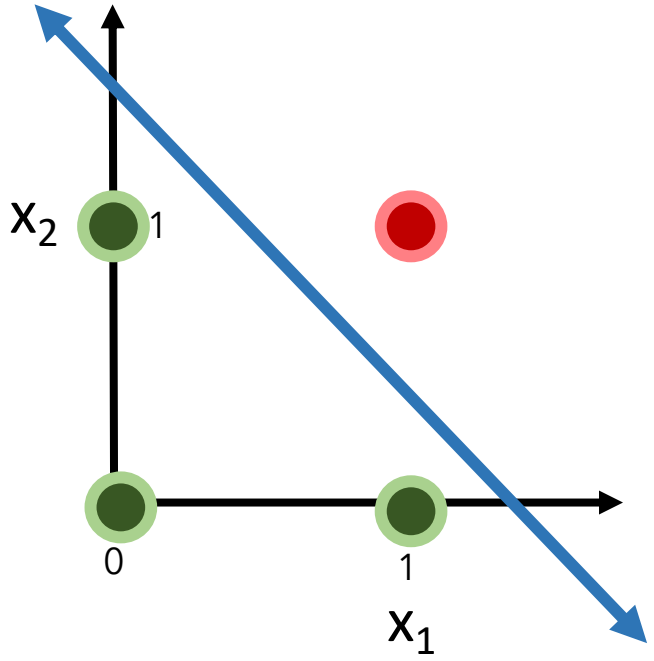
Perceptron : Encoding

Logic Gates

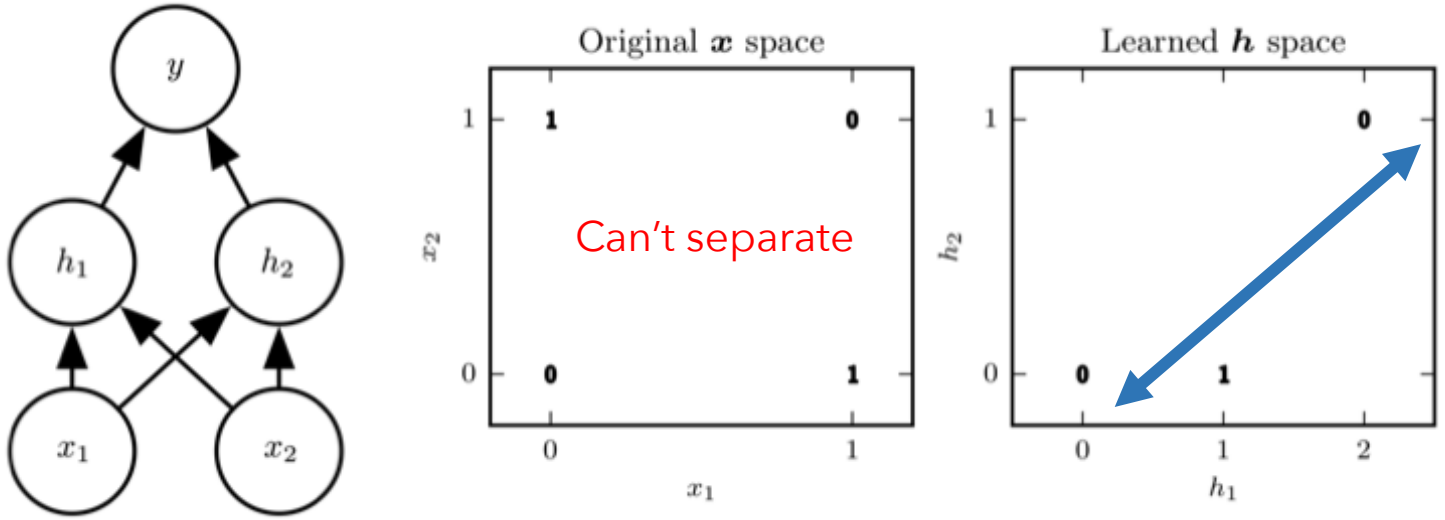
→ Suppose we have two Boolean features, and we want to use perceptron to encode the NAND function.

NAND Truth Table

x1	x2	y
0	0	1
1	0	1
0	1	1
1	1	0



Decision Surface of a Perceptron

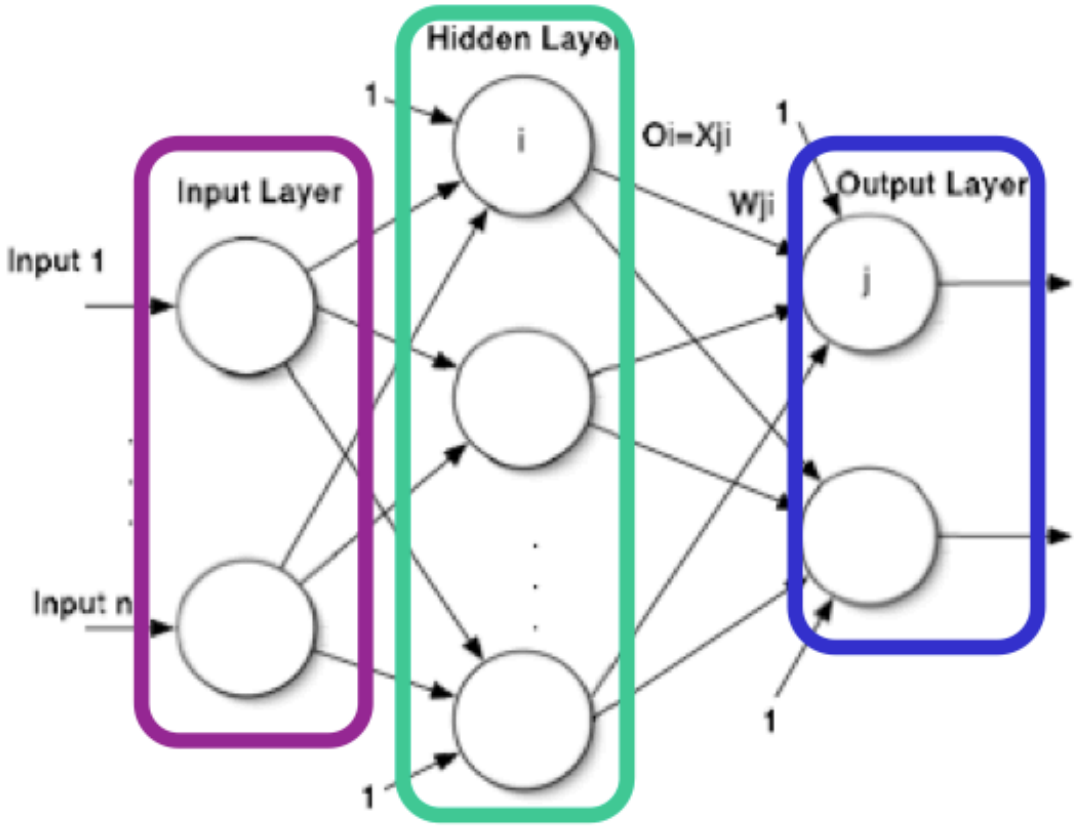


Networks of Perceptrons

→ To represent non-linearly separable functions (e.g. XOR), we could use a network of perceptron-like elements.

→ If we connect perceptrons into networks, the error surface for the network is not differentiable (because of the hard threshold).

Feed-Forward Neural Networks



Defining Neural Networks

→ They go by many names.

→ "Multilayer Perceptrons" (MLP)

→ "Feedforward Neural Networks"

→ "Deep Feedforward Neural Networks"

Layer Types

→ ***Input Layer*: (Layer 1)**

→ Copies inputs. Nothing else.

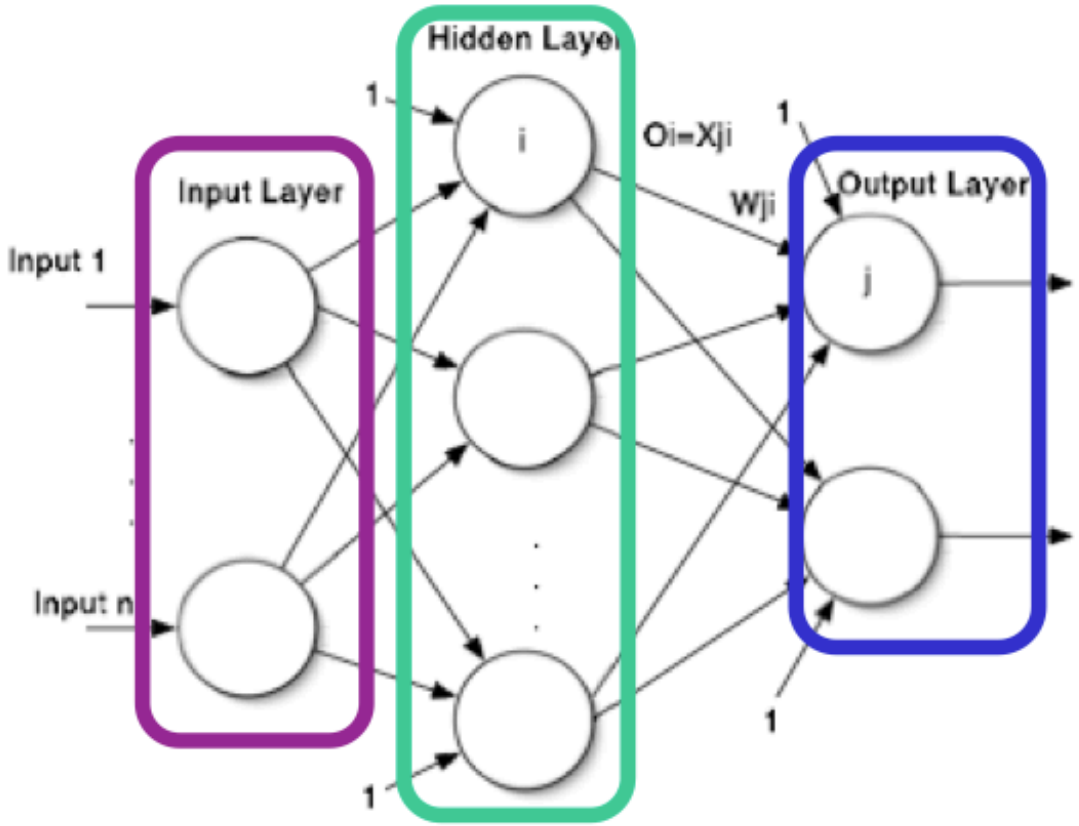
→ ***Hidden Layers*: (Layers *k-1*)**

→ Manipulate inputs. Can't detect them.

→ ***Output Layers*: (Layer *K*)**

→ Final layers whose units provide outputs.

Feed-Forward Neural Networks



Understanding "Feedforward"

→ Information flows through the function being evaluated (i.e. from your inputs).

→ There are no "feedback" connections.

→ Information flows <u>one way</u>.

"Fully Connected" → All units in layer ***k*** provide input to all units in layer ***k+1***.

Networks ←→ Functions

NNs can be thought of as chains of functions.

$$f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$$

$f^{(1)}$ is first layer, $f^{(2)}$ is second layer, etc.

→ Length of the chain is referred to as "depth".

→ "Deep Learning" = massive depth.

Computing Network Output

To use the network to make a prediction about instance <x,y=?>, run a **forward pass** through the network.

For layer k = 1 → K

1. Computing the output of all neurons in layer *k*: $o_j = \sigma(w_j \cdot x_j), \forall j \in \text{Layer } k$

2. Copy this output as the input to the next layer. $x_{j,i} = o_i, \forall i \in \text{Layer } k, \forall j \in \text{Layer } k+1$

The output of the final layer is the predicted output ***y***.

Learning in Feed Forward Neural Networks

Assume the network structure (units + connections) is given.
→ You can endlessly tweak this!

The learning problem is finding a good set of weights that minimize the error at the output of the network.
→ Minimize loss on test set.

Approach: Gradient Descent!
→ Because the form of the hypothesis formed by the network, h_w, is:

1. **Differentiable:** Because we're using sigmoid functions for activation.
2. **Very Complex!:** Hence, direct computation of the optimal weights is not possible.

Gradient Descent Preliminaries for NN

Assume we have a fully connected network:
- ***N*** input units (index 1, … , ***N***)
- ***H*** hidden units in a single layer (index ***N+1***, …, ***N+H***)
- One output unit (index ***N+H+1***)

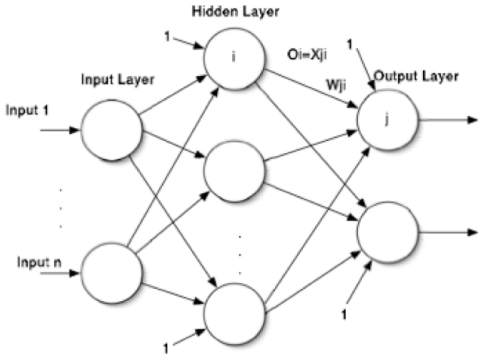
Suppose you want to compute the weight update after seeing instance <x,y>.

Let o_i, $i = 1, \ldots, H+N+1$ be the outputs of all units in the network for the given input *x*.

The sum-squared error function is:

$$J(w) = \frac{1}{2}(y - h_w(x))^2 = \frac{1}{2}(y - o_{N+H+1})^2$$

Backpropagation



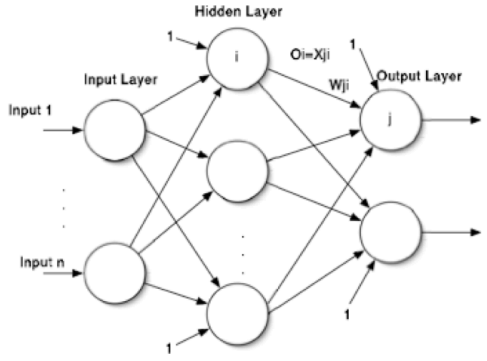
Backpropagation = Gradient descent over **all parameters**.
→ Measure the margin of error of the output and adjust weights accordingly to decrease the error.

The Neural Learning Process (One "Epoch")

1. **Forward Propagation:**
   → Apply a set of weights and calculate output.

2. **Back Propagation:**
   → Calculate error and adjust weights.

Backpropagation: Convergence

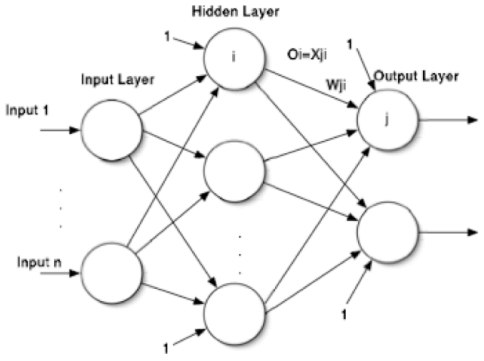


Backpropagation = Gradient descent over **all parameters**.
→ Measure the margin of error of the output and adjust weights accordingly to decrease the error.

If Learning Rate = Appropriate, Algorithm converges to a local minimum.
→ Not the global minimum. (Could be much worse!)
→ There can be MANY local minimum.
→ Could use random restarts = train multiple nets with different initial weights.
→ In practice, the learned solution is often good with a few restarts.

Training can take thousands of iterations. → Very slow! (BUT: Prediction is very fast.)

Backpropagation: Learning Rate

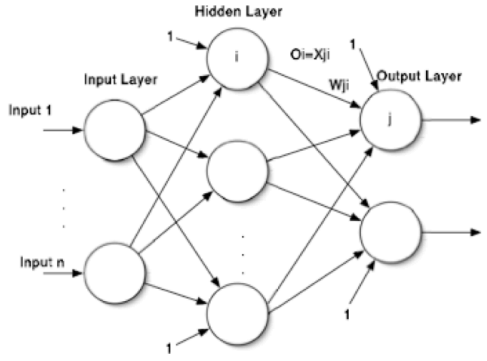


Backpropagation = Gradient descent over **all parameters**.
→ Measure the margin of error of the output and adjust weights accordingly to decrease the error.

Backprop is sensitive to the choice of learning rate.
→ Too large = Divergence.
→ Too small = Very slow learning.
→ The learning rate also influences the ability to escape local optima.

Very often, different learning rates are used for units in different layers. Difficult to tune by hand.
→ **Heuristic**: Track performance on validation set. When it stabilizes, divide the rate by 2.

Backpropagation: AdaGrad



Backpropagation = Gradient descent over **all parameters**.
→ Measure the margin of error of the output and adjust weights accordingly to decrease the error.

AdaGrad: A Technique for Adaptive Learning
→ Calculate adaptive learning rate per parameter.
→ **Intuition:** Adapt the learning rate depending on previous updates to that parameter.
  → Learn slowly for frequent features.
  → Learn faster for rare, but informative features.

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

Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*

John Duchi JDUCHI@CS.BERKELEY.EDU
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720 USA
Elad Hazan EHAZAN@IE.TECHNION.AC.IL
Technion - Israel Institute of Technology
Technion City
Haifa, 32000, Israel
Yoram Singer SINGER@GOOGLE.COM
Google
1600 Amphitheatre Parkway
Mountain View, CA 94043 USA

Editor: Tong Zhang

Abstract

We present a new family of subgradient methods that dynamically incorporate knowledge of the geometry of the data observed in earlier iterations to perform more informative gradient-based learning. Metaphorically, the adaptation allows us to find needles in haystacks in the form of very predictive but rarely seen features. Our paradigm stems from recent advances in stochastic optimization and online learning which employ proximal functions to control the gradient steps of the algorithm. We describe and analyze an apparatus for adaptively modifying the proximal function, which significantly simplifies setting a learning rate and results in regret guarantees that are provably as good as the best proximal function that can be chosen in hindsight. We give several efficient algorithms for empirical risk minimization problems with common and important regularization functions and domain constraints. We experimentally study our theoretical analysis and show that adaptive subgradient methods outperform state-of-the-art, yet non-adaptive, subgradient algorithms.

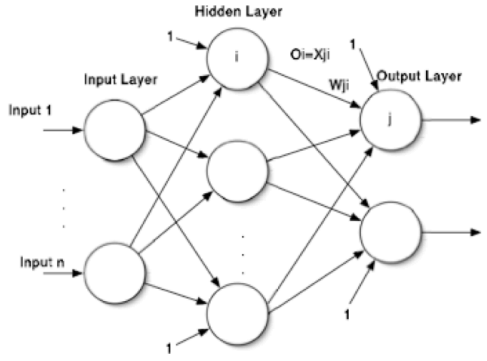
Keywords: subgradient methods, adaptivity, online learning, stochastic convex optimization

1. Introduction

In many applications of online and stochastic learning, the input instances are of very high dimension, yet within any particular instance only a few features are non-zero. It is often the case, however, that infrequently occurring features are highly informative and discriminative. The informativeness of rare features has led practitioners to craft domain-specific feature weightings, such as TF-IDF (Salton and Buckley, 1988), which pre-emphasize infrequently occurring features. We use this old idea as a motivation for applying modern learning-theoretic techniques to the problem of online and stochastic learning, focusing concretely on (sub)gradient methods.

*. A preliminary version of this work was published in COLT 2010.

Backpropagation: Momentum



Backpropagation = Gradient descent over **all parameters**.
→ Measure the margin of error of the output and adjust weights accordingly to decrease the error.

Momentum: A Supplement for Learning Rate
→ On the t-th training sample, instead of the update: → $\Delta w_{ij} \leftarrow \alpha_{ij}\delta_j x_{ij}$
→ Do: $\Delta w_{ij}(t) \leftarrow \alpha_{ij}\delta_j x_{ij} + \underbrace{\beta \Delta w_{ij}(t-1)}_{\text{momentum}}$

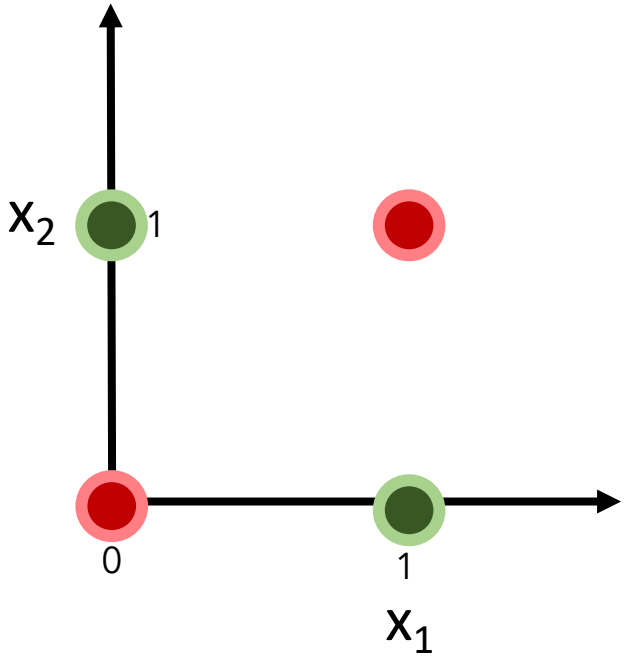
Advantages: Easy to pass small local minima & keeps weights moving in areas where error = 0.
Disadvantages: Too much momentum, and you can get out of a global maximum.
→ One more parameter to tune.

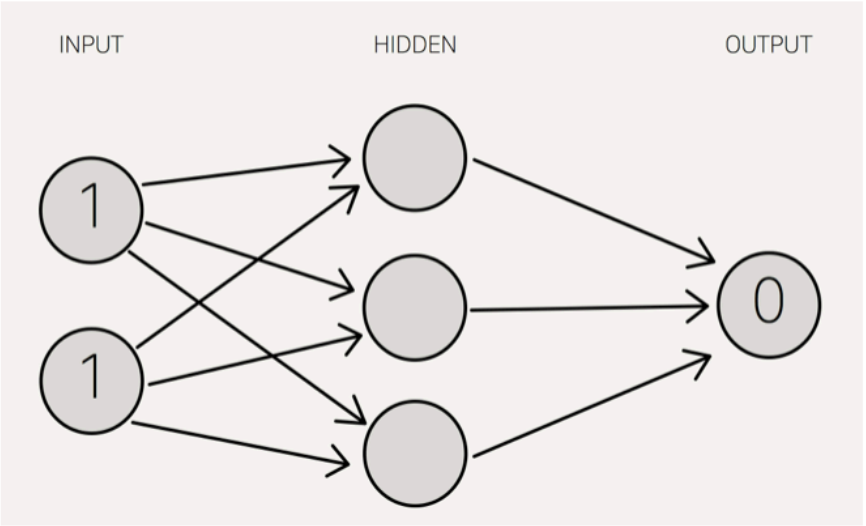
Neural Network: Encoding

Logic Gates
→ Suppose we have two Boolean features, and we want to use perceptron to encode the XOR function.

NAND Truth Table

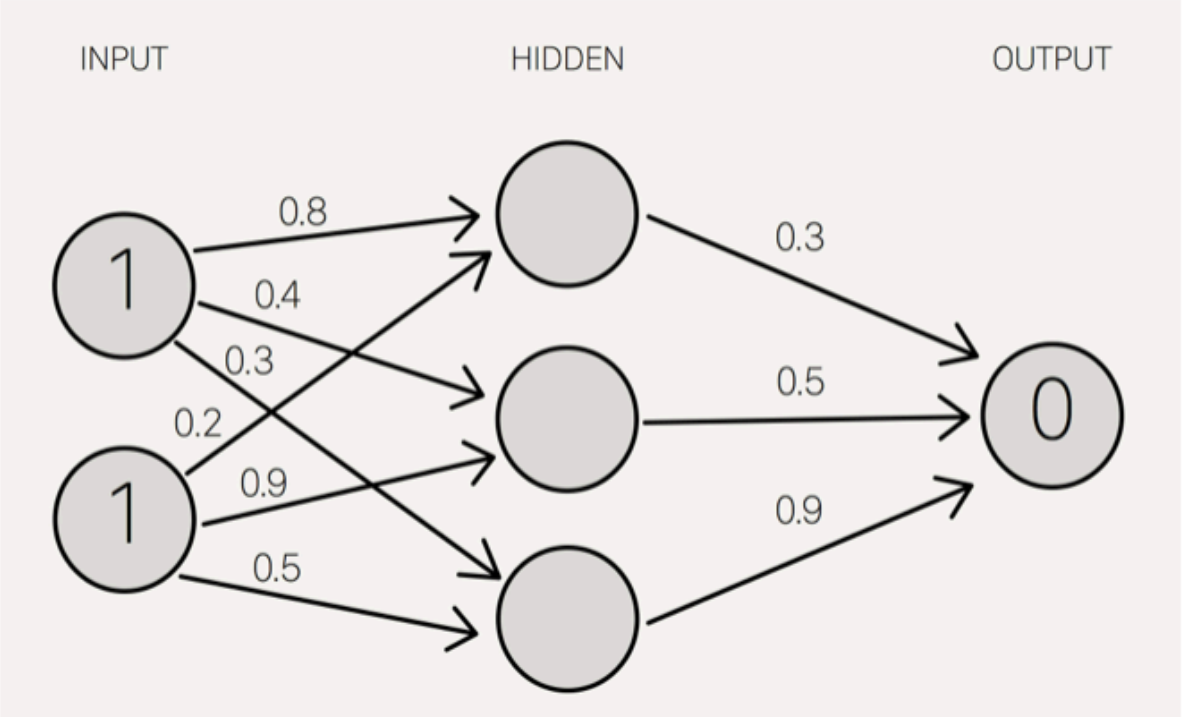
x1	x2	y
0	0	0
1	0	1
0	1	1
1	**1**	**0**





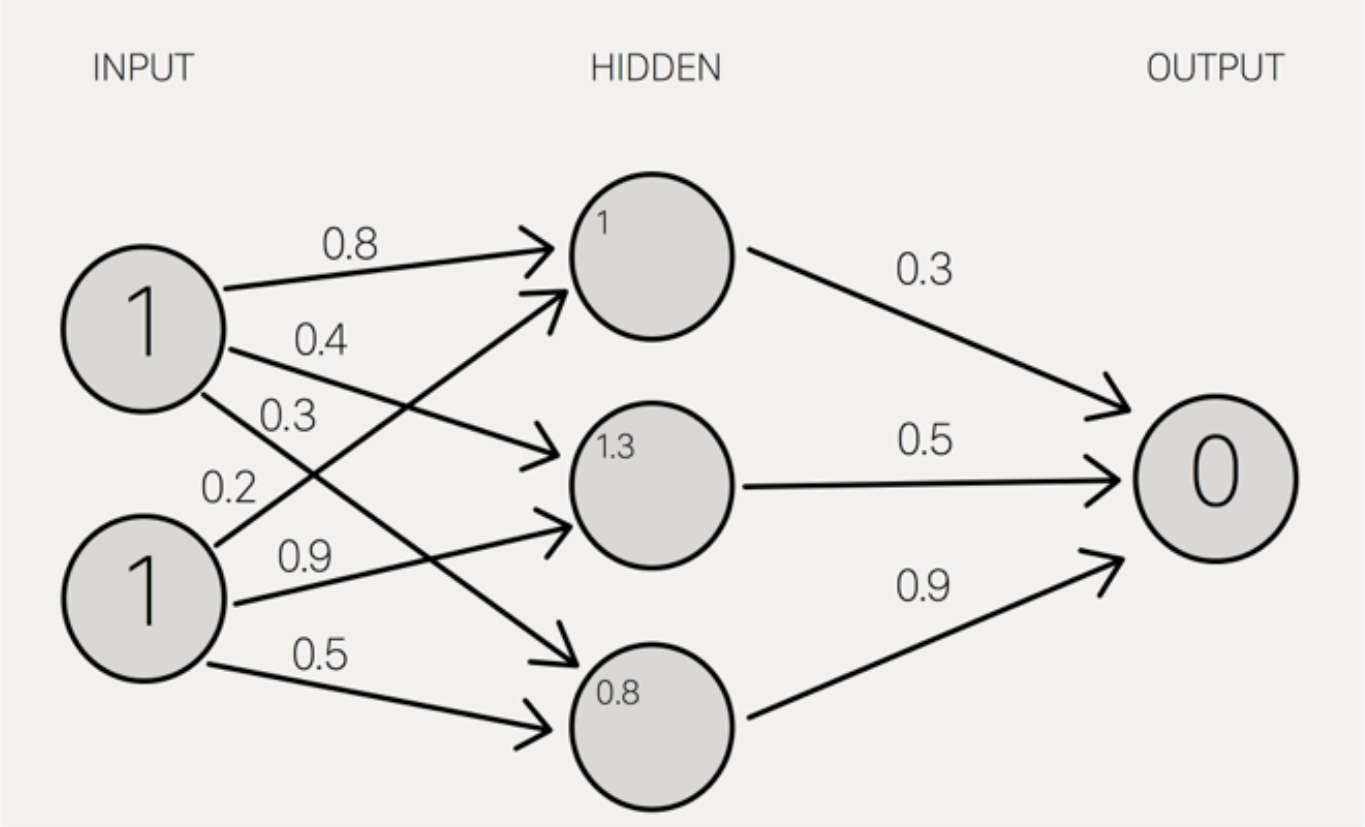
Focus: $x_1 = 1, x_2 = 1, y = 0$

Feed Forward: Random Weights



Step 1: Randomly assign weights.

Feed Forward: Hidden Sums



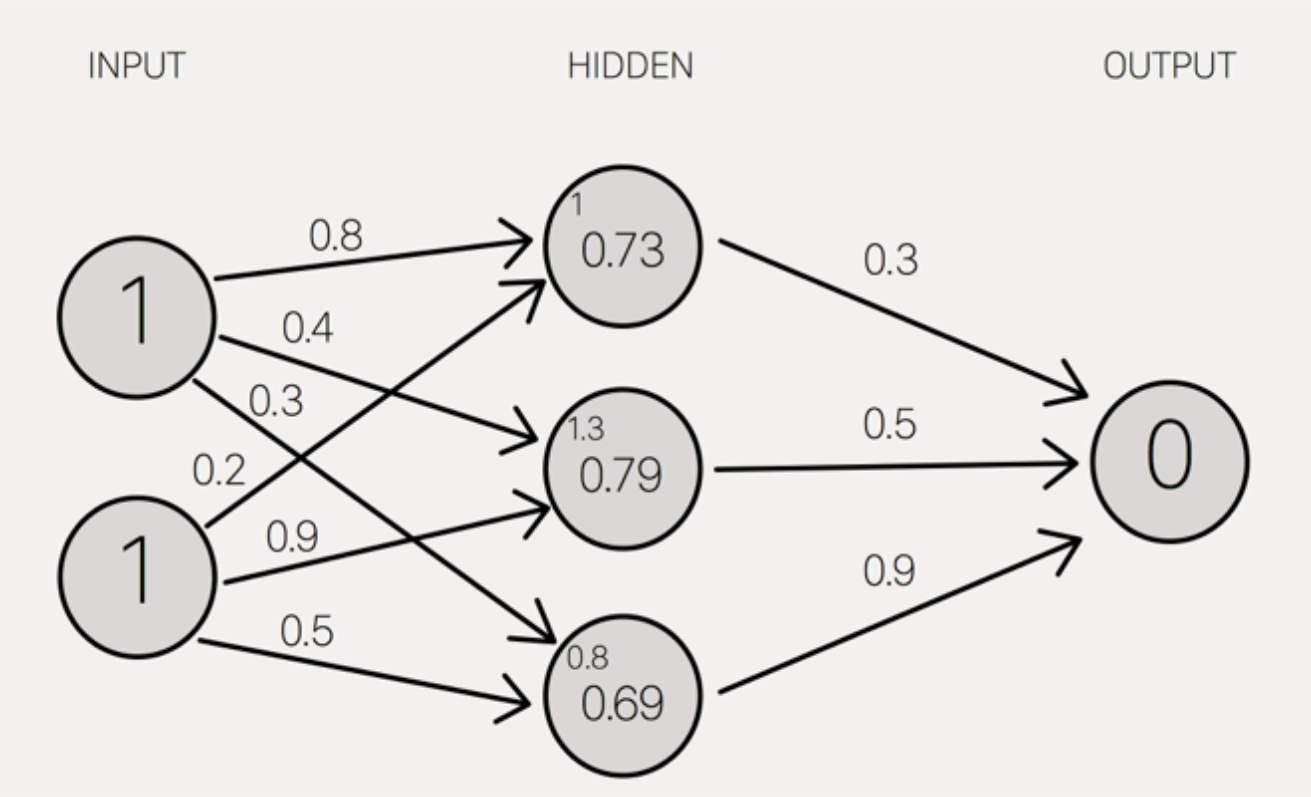
Hidden Unit 1:
1 * 0.8 + 1 * 0.2 = 1

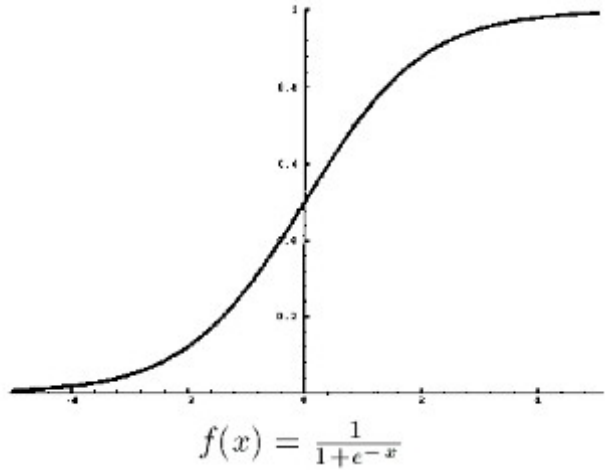
Hidden Unit 2:
1 * 0.4 + 1 * 0.9 = 1.3

Hidden Unit 3:
1 * 0.3 + 1 * 0.5 = 0.8

Step 2: Compute hidden sums and assign them to units.

Feed Forward: Activation



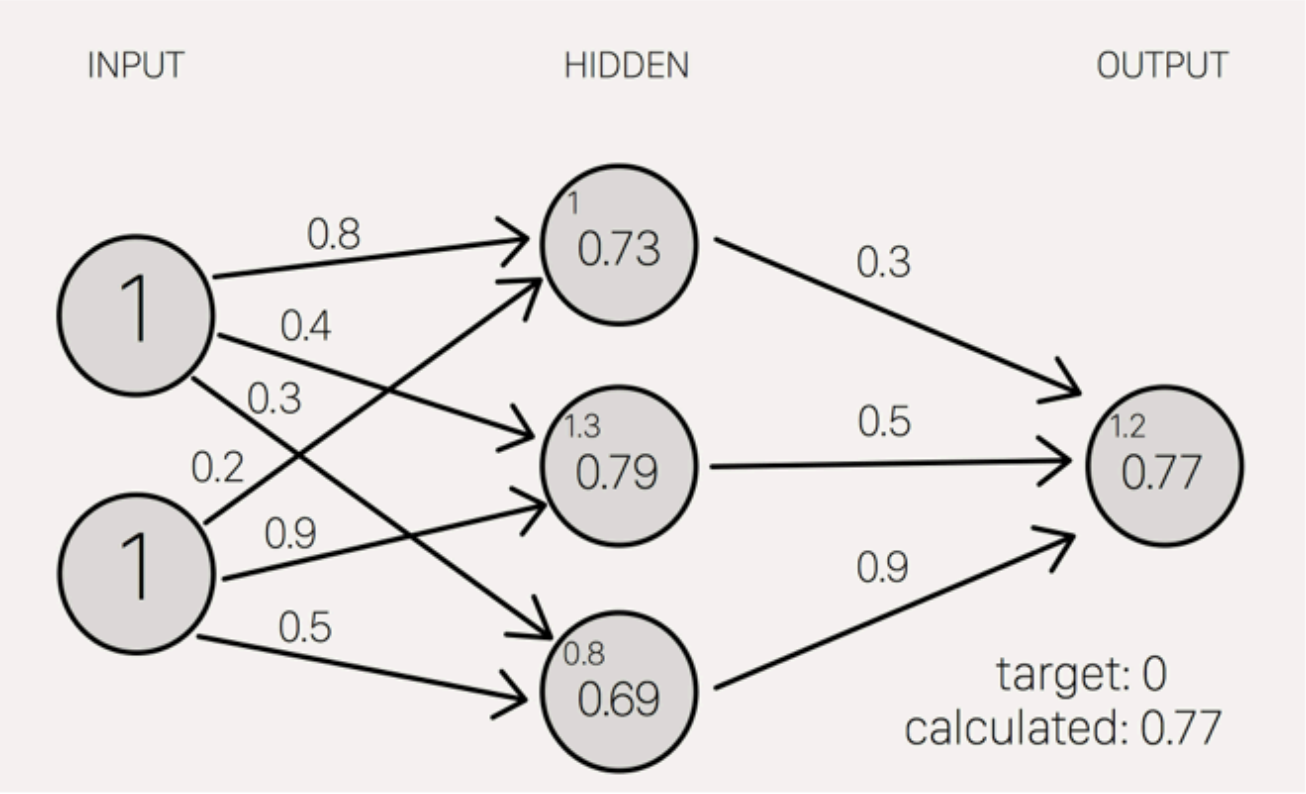


Activation Function: Sigmoid

S(1.0) = 0.73105…
S(1.3) = 0.78583…
S(0.8) = 0.68997…

Step 3: Apply activation function to hidden units and capture result.

Feed Forward: Repeat for Layers



Repeat for Output Layer

→ Sum the weighted connections

0.73 * 0.3 + 0.79 * 0.5 + 0.69 * 0.9 = 1.235

→ Apply the activation function.

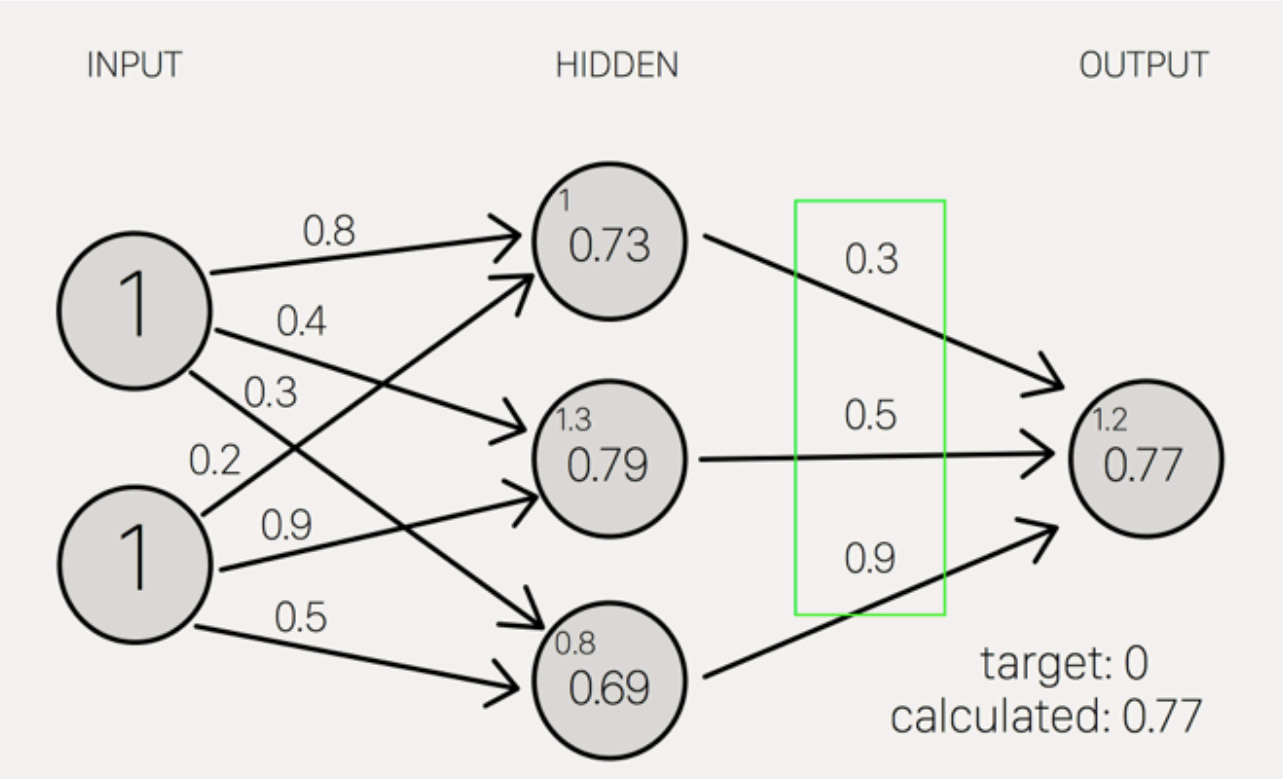
S(1.235) = 0.77469…

If we stopped here***: Inaccurate***.

→ Our initial weights were bad!

Step 4: Repeat the procedure for the next layer.

Backpropagation: Hidden-Output Weights



Compute "Delta Output Sum"

$$\frac{dsum}{dresult} \times (\text{target result} - \text{calculated result}) = \triangle \text{sum}$$

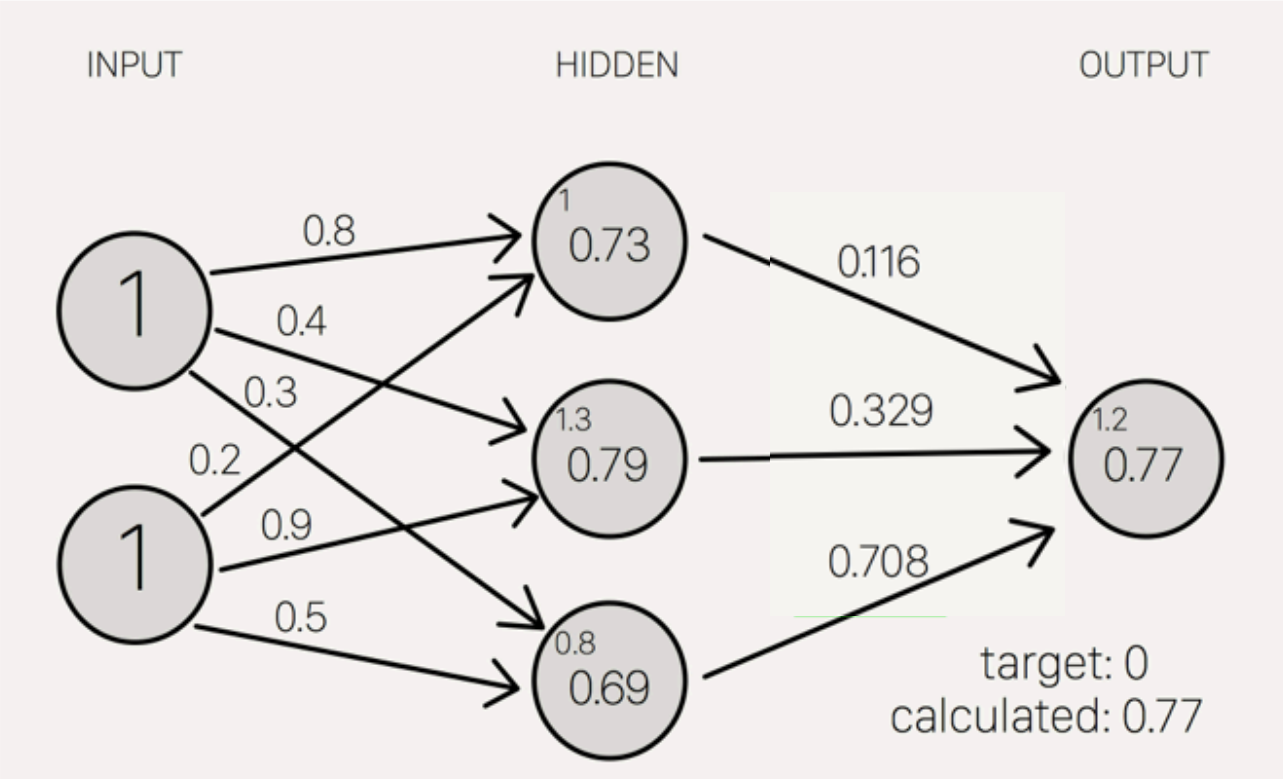
$S'(1.235) * (0 - 0.77) = -0.1343989$.

Compute Weight Change

$W_{D1} = -0.1344 / 0.73105 = -0.1838$
$W_{D2} = -0.1344 / 0.78583 = -0.1710$
$W_{D3} = -0.1344 / 0.69997 = -0.1920$

Step 1: Calculate Error Between Hidden and Output Layers

Backpropagation: Hidden-Output Weights



Compute Weight Change

W_{D1} = -0.1344 / 0.73105 = -0.1838

W_{D2} = -0.1344 / 0.78583 = -0.1710

W_{D3} = -0.1344 / 0.69997 = -0.1920

Compute New Weights

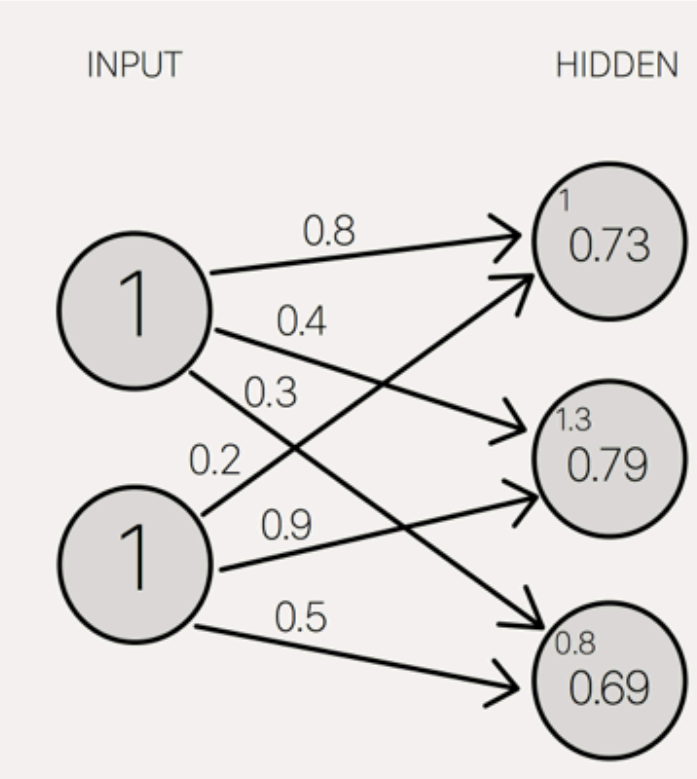
W_{N1} = 0.3 - 0.1838 = 0.1162

W_{N2} = 0.5 - 0.1710 = 0.329

W_{N3} = 0.9 - 0.1920 = 0.708

Step 2: Calculate Error Between Hidden and Output Layers

Backpropagation: Compute Hidden Sums



Compute "Delta Hidden Sum"

S`(hidden sum)

```
Delta hidden sum = delta output sum / hidden-to-outer weights * S'(hidden su
Delta hidden sum = -0.1344 / [0.3, 0.5, 0.9] * S'([1, 1.3, 0.8])
Delta hidden sum = [-0.448, -0.2688, -0.1493] * [0.1966, 0.1683, 0.2139]
Delta hidden sum = [-0.088, -0.0452, -0.0319]
```

Delta Output Sum = - 0.1344

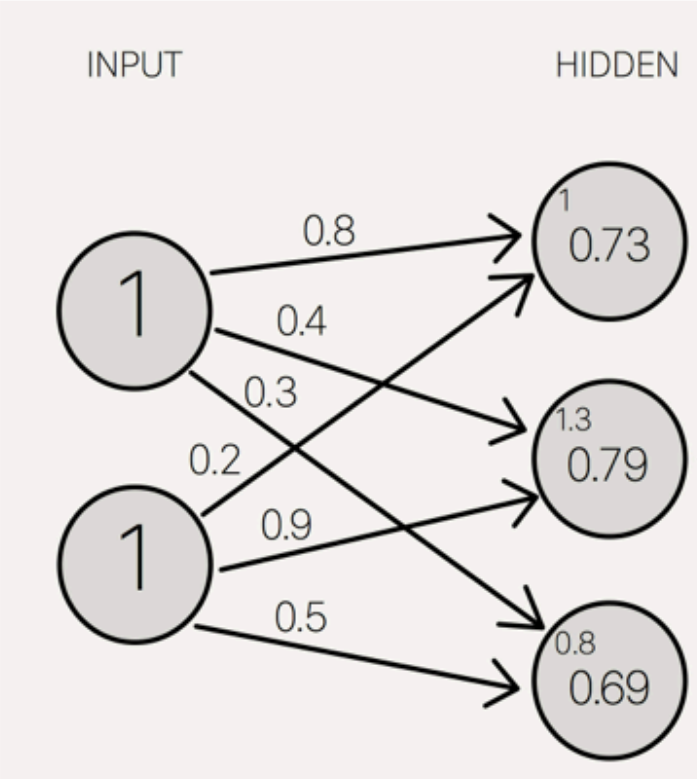
Hidden-to-outer weights = [0.3, 0.5, 0.9]

S`(Hidden Sums) = [1, 1.3, 0.8]

Delta Hidden Sums = [-0.088, -0.0452, -0.0319

Step 3: Compute Delta Hidden Sums

Backpropagation: Weight Change

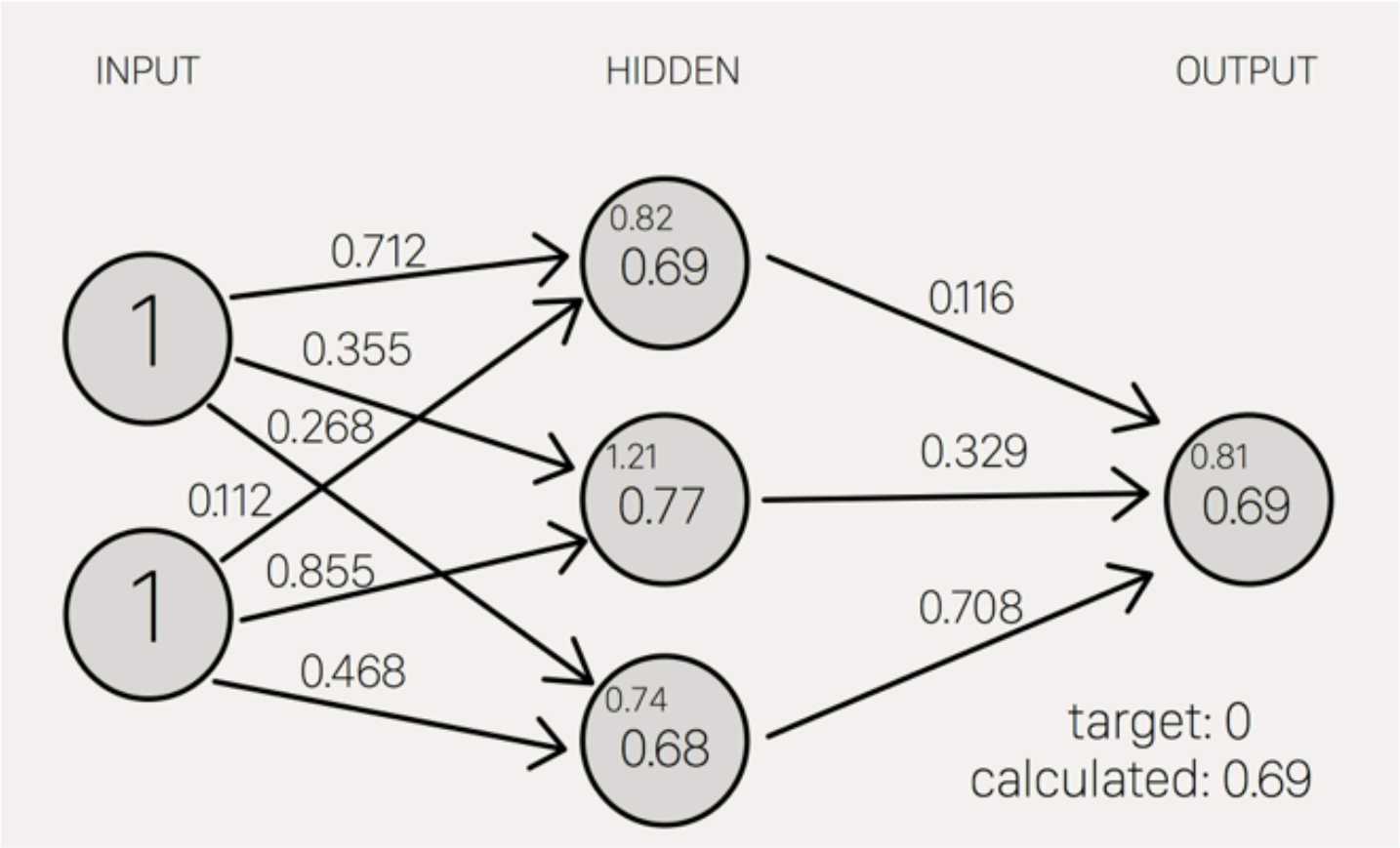


Compute "Delta Weight Changes"

```
input 1 = 1
input 2 = 1

Delta weights = delta hidden sum / input data
Delta weights = [-0.088, -0.0452, -0.0319] / [1, 1]
Delta weights = [-0.088, -0.0452, -0.0319, -0.088, -0.0452, -0.0319]

old w1 = 0.8
old w2 = 0.4
old w3 = 0.3
old w4 = 0.2
old w5 = 0.9
old w6 = 0.5

new w1 = 0.712
new w2 = 0.3548
new w3 = 0.2681
new w4 = 0.112
new w5 = 0.8548
new w6 = 0.4681
```

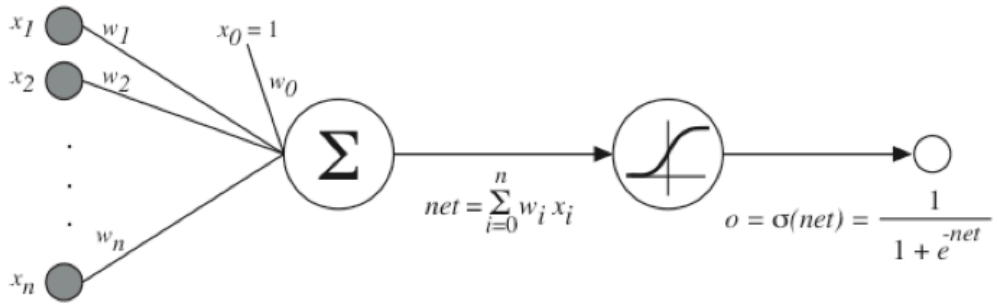
Step 4: Compute Weight Changes and Apply to Original Weights

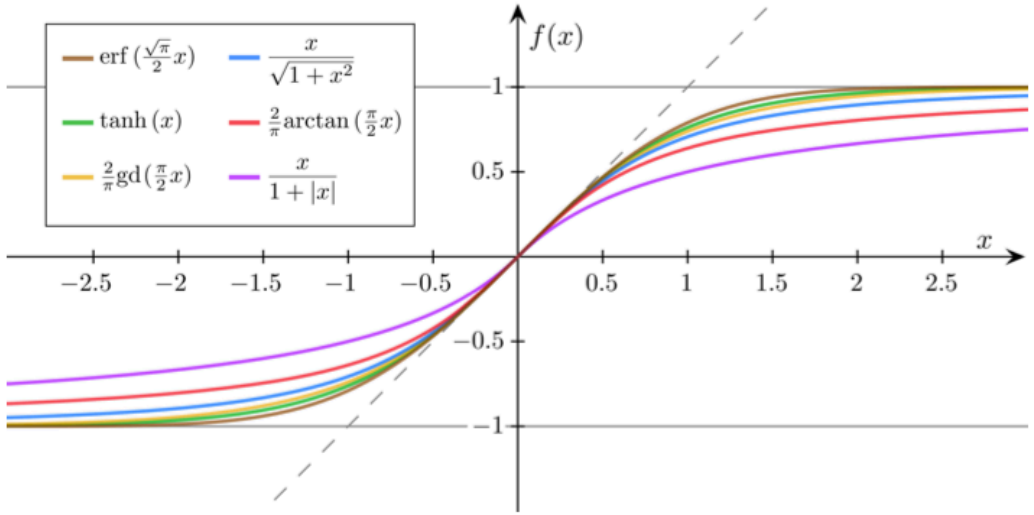
Backpropagation: Closing the First Iteration



Step 5: Do a forward pass to re-evaluate cost.

A Family of Sigmoid Functions





Example:

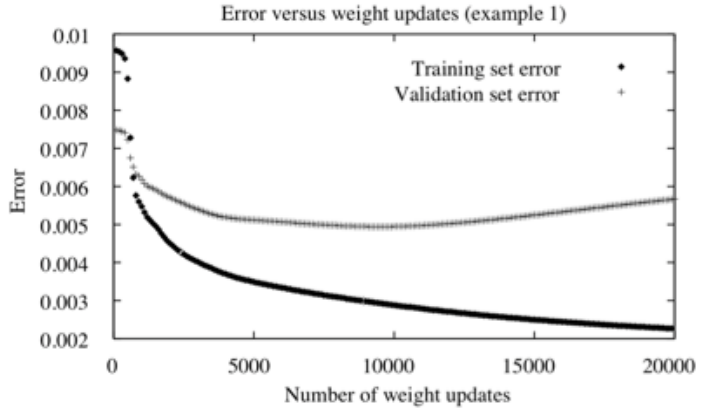
$$\sigma(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\frac{\delta\sigma(z)}{\delta z} = 1 - \sigma(z)^2$$

Overfitting and Overtraining

Traditional **overfitting** is concerned with the number of parameters vs. the number of instances.

Overtraining in Neural Networks
In Neural Nets, a related phenomenon occurs when weights take on large magnitudes, i.e. unit saturation.
→ As learning progresses, the network has more active parameters.



Use validation set to decide when to stop training.
→ **Training horizon** is a hyperparameter.

Regularization is also effective.

Managing Overfitting

$$J(w) = 0.5(y - h_w(x))^2 + 0.5\lambda w^T w$$

Option 1: Regularization (Hyperparameter)

→ Incorporate an L2 penalty, selected by cross-validation.

→ Could use different penalties, e.g. λ_1, λ_2, for each layer.

Option 2: Dropout (Hyperparameter)

→ Randomly "drop" some units from the network when training.

→ e.g. Each hidden unit is dropped with prob 0.5 in each iteration

→ e.g. Each input unit is dropped with prob 0.2 in each iteration

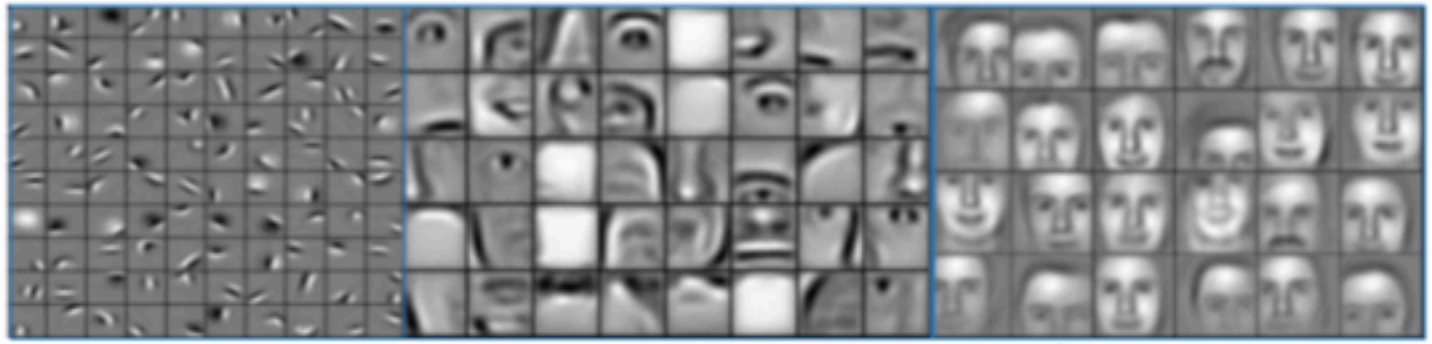
Option 3: Data Augmentation

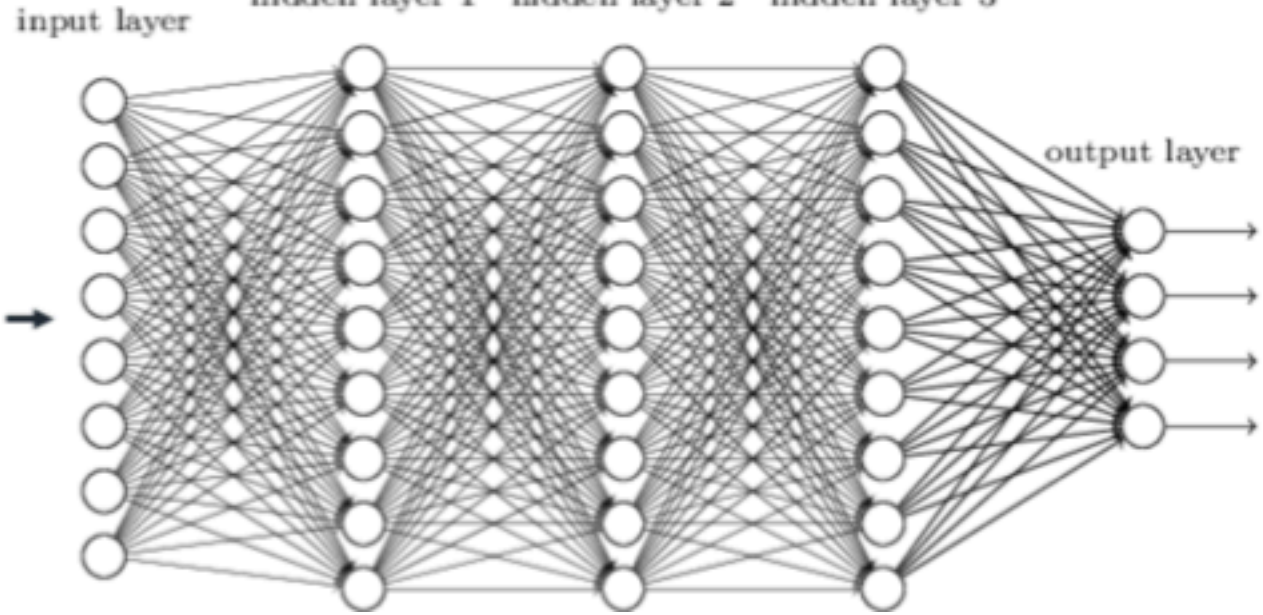
→ Get more data by making minor changes to the existing dataset.

→ e.g. Reflections, translations, rotations, scale, crop, simulated noise.

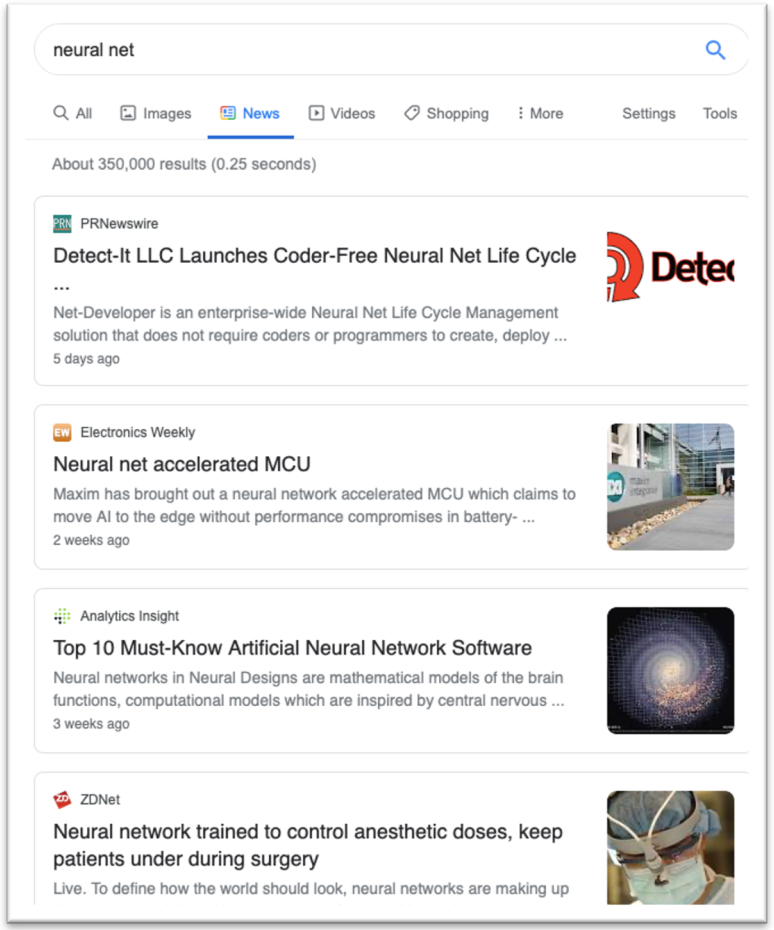
The Power of Depth

Deep neural networks learn hierarchical feature representations





Ubiquitous and Unexplainable



Future Lecture: Interpretability.

Opportune Uses for Neural Networks

You might ask: **When should use neural networks?**

→ Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
→ Output is discrete or real-valued, or a vector of values.

→ Possibly noisy data.

→ Training time isn't important.
→ Prediction time is important.

→ Form of target function is unknown.

→ Human readability is not important. (More on Interpretability later!)

Next Time

We will address:

1. Convolutional Neural Nets
2. Neural Networks with Keras and Tensorflow