CSCE 496/896 Lecture 2: Basic Artificial Neural Networks

Stephen Scott

(Adapted from Vinod Variyam, Ethem Alpaydin, Tom Mitchell, Ian Goodfellow, and Aurélien Géron)

sscott@cse.unl.edu

Introduction
Supervised Learning

- **Supervised learning** is most fundamental, "classic" form of machine learning
- "Supervised" part comes from the part of *labels* for examples (instances)
- Many ways to do supervised learning; we'll focus on **artificial neural networks**, which are the basis for deep learning

Introduction
ANNs

Consider humans:

- Total number of neurons $\approx 10^{10}$
- Neuron switching time $\approx 10^{-3}$ second (vs. 10^{-10})
- Connections per neuron $\approx 10^4$–10^5
- Scene recognition time ≈ 0.1 second
- 100 inference steps doesn't seem like enough

$\Rightarrow$ massive parallel computation

Introduction
Properties

Properties of artificial neural nets (ANNs):

- Many "neuron-like" switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically

Strong differences between ANNs for ML and ANNs for biological modeling

When to Consider ANNs

- Input is high-dimensional discrete- or real-valued (e.g., raw sensor input)
- Output is discrete- or real-valued
- Output is a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant
- Long training times acceptable

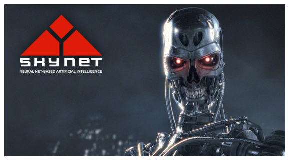
Introduction
Brief History of ANNs

- **The Beginning:** Linear units and the Perceptron algorithm (1940s)
  - Spoiler Alert: stagnated because of inability to handle data not *linearly separable*
  - Aware of usefulness of multi-layer networks, but could not train
- **The Comeback:** Training of multi-layer networks with Backpropagation (1980s)
  - Many applications, but in 1990s replaced by large-margin approaches such as support vector machines and boosting

CSCE 496/896 Lecture 2: Basic Artificial Neural Networks
Stephen Scott
Introduction
Supervised Learning
Basic Units
Gradient Descent
Nonlinearly Separable Problems
Backprop
Types of Units
Putting Things Together

Introduction
Brief History of ANNs (cont'd)

- **The Resurgence:** Deep architectures (2000s)
  - Better hardware[1] and software support allow for deep ($>$ 5–8 layers) networks
  - Still use Backpropagation, but
    - Larger datasets, algorithmic improvements (new loss and activation functions), and deeper networks improve performance considerably
  - Very impressive applications, e.g., captioning images
- **The Inevitable:** (TBD)
  - Oops



[1]Thank a gamer today.

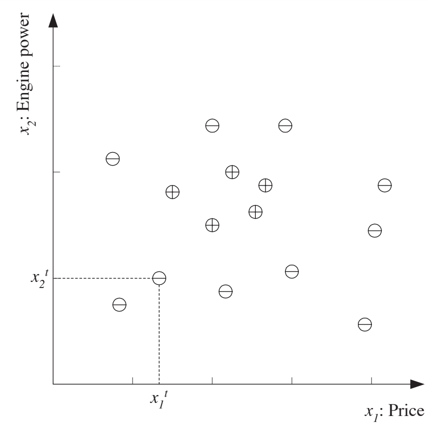
Outline

- Supervised learning
- Basic ANN units
  - Linear unit
  - Linear threshold units
  - Perceptron training rule
- Gradient Descent
- Nonlinearly separable problems and multilayer networks
- Backpropagation
- Types of activation functions
- Putting everything together

Learning from Examples

- Let C be the **target function** (or **target concept**) to be learned
  - Think of C as a function that takes as input an **example** (or **instance**) and outputs a **label**
- **Goal:** Given **training set** $\mathcal{X} = \{(\boldsymbol{x}^t, y^t)\}_{t=1}^N$ where $y^t = C(\boldsymbol{x}^t)$, output **hypothesis** $h \in \mathcal{H}$ that approximates C in its classifications of new instances
- Each instance $\boldsymbol{x}$ represented as a vector of **attributes** or **features**
  - E.g., let each $\boldsymbol{x} = (x_1, x_2)$ be a vector describing attributes of a car; $x_1 =$ price and $x_2 =$ engine power
  - In this example, label is binary (positive/negative, yes/no, 1/0, $+1/-1$) indicating whether instance $\boldsymbol{x}$ is a "family car"

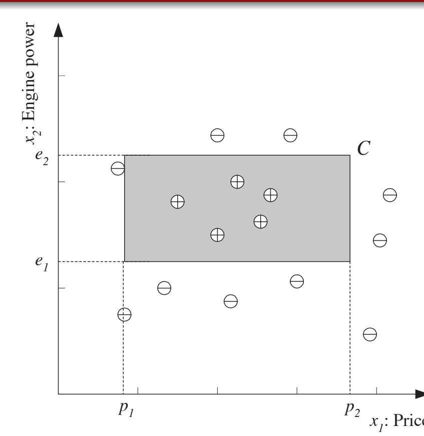
Learning from Examples (cont'd)



Thinking about C

- Can think of target concept C as a **function**
  - In example, C is an axis-parallel box, equivalent to upper and lower bounds on each attribute
  - Might decide to set $\mathcal{H}$ (set of candidate hypotheses) to the same family that C comes from
  - Not required to do so
- Can also think of target concept C as a **set** of positive instances
  - In example, C the continuous set of all positive points in the plane
- Use whichever is convenient at the time

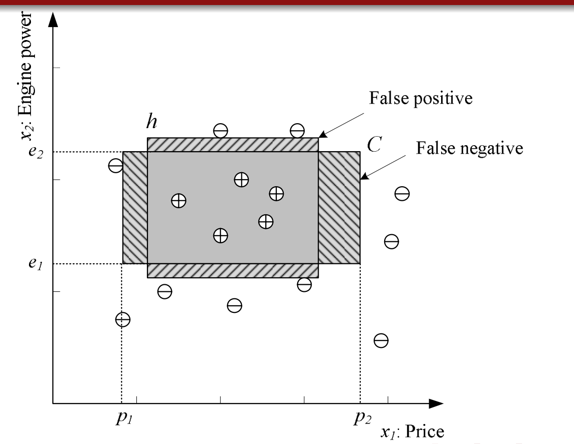
Thinking about C (cont'd)



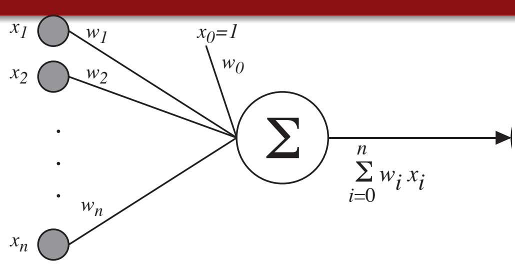
Hypotheses and Error

- A learning algorithm uses training set $\mathcal{X}$ and finds a hypothesis $h \in \mathcal{H}$ that approximates C
- In example, $\mathcal{H}$ can be set of all axis-parallel boxes
- If C guaranteed to come from $\mathcal{H}$, then we know that a perfect hypothesis exists
  - In this case, we choose h from the **version space** = subset of $\mathcal{H}$ consistent with $\mathcal{X}$
  - What learning algorithm can you think of to learn C?
- Can think of two types of **error** (or **loss**) of h
  - **Empirical error** is fraction of $\mathcal{X}$ that h gets wrong
  - **Generalization error** is probability that a new, randomly selected, instance is misclassified by h
    - Depends on the probability distribution over instances
  - Can further classify error as **false positive** and **false negative**

Hypotheses and Error (cont'd)



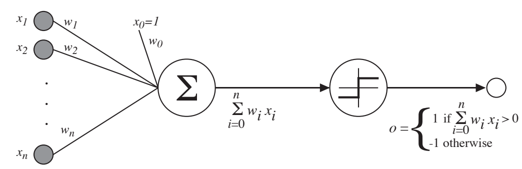
Linear Unit (Regression)



$$\hat{y} = f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^\top \boldsymbol{w} + b = w_1 x_1 + \cdots + w_n x_n + b$$

- Each weight vector $\boldsymbol{w}$ is different h
- If set $w_0 = b$, can simplify above
- Forms the basis for many other activation functions

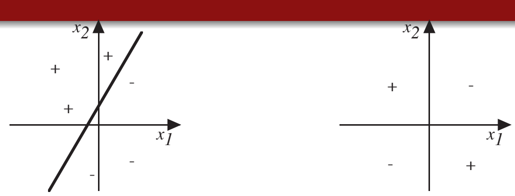
Linear Threshold Unit (Binary Classification)



$$y = o(\boldsymbol{x}; \boldsymbol{w}, b) = \begin{cases} +1 & \text{if } f(\boldsymbol{x}; \boldsymbol{w}, b) > 0 \\ -1 & \text{otherwise} \end{cases}$$

(sometimes use 0 instead of -1)

Linear Threshold Unit
Decision Surface



(a) (b)

Represents some useful functions

- What parameters $(\boldsymbol{w}, b)$ represent $g(x_1, x_2; \boldsymbol{w}, b) = AND(x_1, x_2)$?

But some functions not representable

- I.e., those not **linearly separable**
- Therefore, we'll want **networks** of units

Linear Threshold Unit
Non-Numeric Inputs

- What if attributes are not numeric?
- **Encode** them numerically
- E.g., if an attribute *Color* has values *Red*, *Green*, and *Blue*, can encode as **one-hot** vectors $[1, 0, 0]$, $[0, 1, 0]$, $[0, 0, 1]$
- Generally better than using a single integer, e.g., *Red* is 1, *Green* is 2, and *Blue* is 3, since there is no implicit ordering of the values of the attribute

Perceptron Training Rule (Learning Algorithm)

$$w_j' \leftarrow w_j + \eta\,(y^t - \hat{y}^t)\,x_j^t$$

where

- x_j^t is jth attribute of training instance t
- y^t is label of training instance t
- $\hat{y}^t$ is Perceptron output on training instance t
- $\eta > 0$ is small constant (e.g., 0.1) called **learning rate**

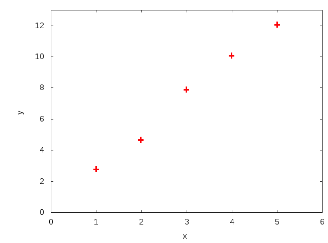
I.e., if $(y - \hat{y}) > 0$ then increase w_j w.r.t. x_j, else decrease

Can prove rule will converge if training data is linearly separable and η sufficiently small

Where Does the Training Rule Come From?
Linear Regression

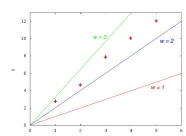
- Recall initial *linear unit* (no threshold)
- If only one feature, then this is a **regression** problem
- Find a straight line that best fits the training data
  - For simplicity, let it pass through the origin
  - Slope specified by parameter w_1

x^t	y^t
1	2.8
2	4.65
3	7.9
4	10.1
5	12.1



Where Does the Training Rule Come From?
Linear Regression

- If we use hypothesis $w_1 = 1$, then **square loss** is



$$J(1) = \sum_{t=1}^{m} (\hat{y}^t - y^t)^2$$

$$= \sum_{t=1}^{m} (1x^t - y^t)^2 = (1-2.8)^2 + (2-4.65)^2 + (3-7.9)^2$$

$$+(4-10.1)^2 + (5-12.1)^2 = 121.8925$$

- If we use $w_2 = 2$, then we get $J(2) = 13.4925$
- Can plot $J(w_1)$ versus w_1
- Goal is to find w_1 to minimize $J(w_1)$

Where Does the Training Rule Come From?
Linear Regression

- Can write $J(w_1)$ in general:

$$J(w_1) = \sum_{t=1}^{m} (\hat{y}^t - y^t)^2 = \sum_{t=1}^{m} (w_1 x^t - y^t)^2$$

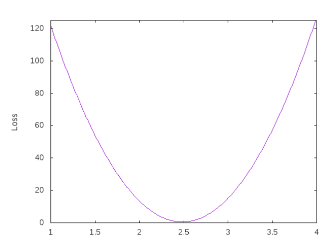
$$= (1w_1 - 2.8)^2 + (2w_1 - 4.65)^2 + (3w_1 - 7.9)^2$$

$$+(4w_1 - 10.1)^2 + (5w_1 - 12.1)^2$$

$$= 55w_1^2 - 273.4w_1 + 340.293$$

Where Does the Training Rule Come From?
Convex Quadratic Optimization

$$J(w_1) = 55w_1^2 - 273.4w_1 + 340.293$$



- Minimum is at $w_1 \approx 2.485$, with loss ≈ 0.53
- What's special about that point?

Where Does the Training Rule Come From?
Gradient Descent

- Recall that a function has a (local) minimum or maximum where the derivative is 0

$$\frac{d}{dw_1} J(w_1) = 110w_1 - 273.4$$

- Setting this $= 0$ and solving for w_1 yields $w_1 \approx 2.485$
- Motivates the use of **gradient descent** to solve in high-dimensional spaces with nonconvex functions:

$$\mathbf{w}' = \mathbf{w} - \eta \nabla J(\mathbf{w})$$

- η is **learning rate** to moderate updates
- Gradient is a vector of partial derivatives: $\left[\frac{\partial J}{\partial w_i}\right]_{i=1}^{n}$
- $\frac{\partial J}{\partial w_i}$ is how much a change in w_i changes J

Where Does the Training Rule Come From?
Gradient Descent Example

- In our example, initialize w_1, then repeatedly update

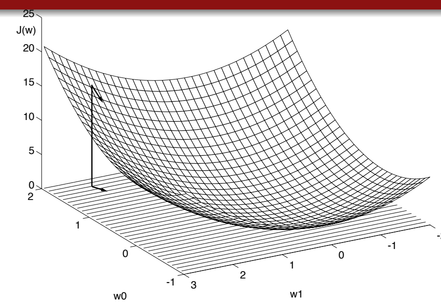
$$w_1' = w_1 - \eta(110\,w_1 - 273.4)$$

eta	0.01			
round	w	J	grad	update
0	1	121.893	-163.4	1.634
1	2.634	1.74498	16.34	-0.1634
2	2.4706	0.5434998	-1.634	0.01634
3	2.48694	0.531485	0.1634	-0.001634
4	2.485306	0.53136485	-0.01634	0.0001634
5	2.4854694	0.53136365	0.001634	-1.634E-05
6	2.48545306	0.53136364	-0.0001634	1.634E-06
7	2.48545469	0.53136364	1.634E-05	-1.634E-07
8	2.48545453	0.53136364	-1.634E-06	1.634E-08
9	2.48545455	0.53136364	1.634E-07	-1.634E-09
10	2.48545455	0.53136364	-1.634E-08	1.634E-10
11	2.48545455	0.53136364	1.634E-09	-1.634E-11
12	2.48545455	0.53136364	-1.634E-10	1.6337E-12
13	2.48545455	0.53136364	1.6314E-11	-1.631E-13
14	2.48545455	0.53136364	-1.592E-12	1.5916E-14
15	2.48545455	0.53136364	0	0

- Could also update one at a time: $\frac{\partial J}{\partial w_1} = 2w_1\,(x^t)^2 - 2x^t y^t$

  $\Rightarrow$ **Stochastic gradient descent** (SGD)

Where Does the Training Rule Come From?
Gradient Descent

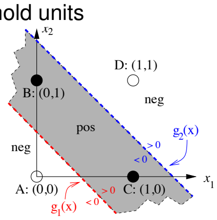


$$\frac{\partial J}{\partial \boldsymbol{w}} = \left[\frac{\partial J}{\partial w_0}, \frac{\partial J}{\partial w_1}, \cdots, \frac{\partial J}{\partial w_n}\right]$$

In general, define loss function J, compute gradient of J w.r.t. J's parameters, then apply gradient descent

Handling Nonlinearly Separable Problems
The XOR Problem

Using linear threshold units



Represent with **intersection** of two linear separators

$$g_1(\boldsymbol{x}) = 1 \cdot x_1 + 1 \cdot x_2 - 1/2$$

$$g_2(\boldsymbol{x}) = 1 \cdot x_1 + 1 \cdot x_2 - 3/2$$

pos $= \left\{\boldsymbol{x} \in \mathbb{R}^2 : g_1(\boldsymbol{x}) > 0 \text{ AND } g_2(\boldsymbol{x}) < 0\right\}$

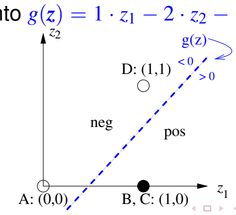
neg $= \left\{\boldsymbol{x} \in \mathbb{R}^2 : g_1(\boldsymbol{x}), g_2(\boldsymbol{x}) < 0 \text{ OR } g_1(\boldsymbol{x}), g_2(\boldsymbol{x}) > 0\right\}$

Handling Nonlinearly Separable Problems
The XOR Problem (cont'd)

Let $z_i = \begin{cases} 0 & \text{if } g_i(\boldsymbol{x}) < 0 \\ 1 & \text{otherwise} \end{cases}$

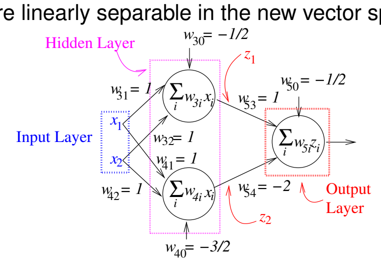
Class	(x_1, x_2)	$g_1(\boldsymbol{x})$	z_1	$g_2(\boldsymbol{x})$	z_2
pos	B: $(0, 1)$	$1/2$	1	$-1/2$	0
pos	C: $(1, 0)$	$1/2$	1	$-1/2$	0
neg	A: $(0, 0)$	$-1/2$	0	$-3/2$	0
neg	D: $(1, 1)$	$3/2$	1	$1/2$	1

Now feed z_1, z_2 into $g(\boldsymbol{z}) = 1 \cdot z_1 - 2 \cdot z_2 - 1/2$



Handling Nonlinearly Separable Problems
The XOR Problem (cont'd)

In other words, we **remapped** all vectors $\boldsymbol{x}$ to $\boldsymbol{z}$ such that the classes are linearly separable in the new vector space

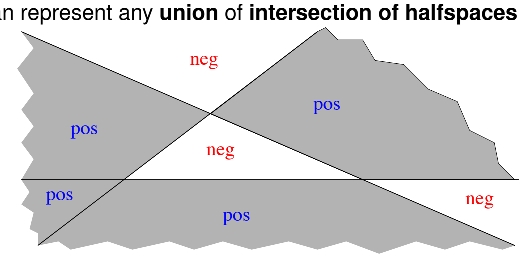


This is a **two-layer perceptron** or **two-layer feedforward neural network**

Can use many **nonlinear** activation functions in hidden layer

Handling Nonlinearly Separable Problems
General Nonlinearly Separable Problems

By adding up to 2 **hidden layers** of linear threshold units, can represent any **union** of **intersection of halfspaces**



First hidden layer defines halfspaces, second hidden layer takes intersection (AND), output layer takes union (OR)

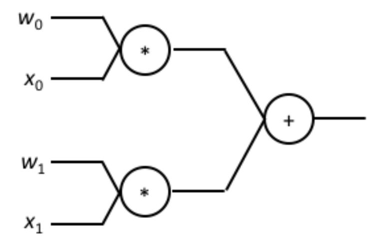
Training Multiple Layers

- In a multi-layer network, have to tune parameters in all layers
- In order to train, need to know the gradient of the loss function w.r.t. each parameter
- The **Backpropagation** algorithm first **feeds forward** the network's inputs to its outputs, then **propagates back** error via repeated application of **chain rule** for derivatives
- Can be decomposed in a simple, modular way

CSCE 496/896 Lecture 2: Basic Artificial Neural Networks
Stephen Scott
Introduction
Supervised Learning
Basic Units
Gradient Descent
Nonlinearly Separable Problems
Backprop
Computation Graphs
Sigmoid Unit
Multilayer Networks
Training Multilayer Networks
Backprop Alg
Types of Units

Computation Graphs

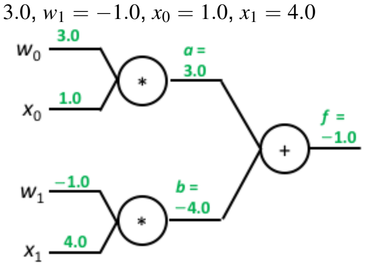
- Given a complicated function $f(\cdot)$, want to know its partial derivatives w.r.t. its parameters
- Will represent f in a modular fashion via a **computation graph** (like what we do in TensorFlow)
- E.g., let $f(\mathbf{w}, \mathbf{x}) = w_0 x_0 + w_1 x_1$



CSCE 496/896 Lecture 2: Basic Artificial Neural Networks
Stephen Scott
Introduction
Supervised Learning
Basic Units
Gradient Descent
Nonlinearly Separable Problems
Backprop
Computation Graphs
Sigmoid Unit
Multilayer Networks
Training Multilayer Networks
Backprop Alg
Types of Units

Computation Graphs

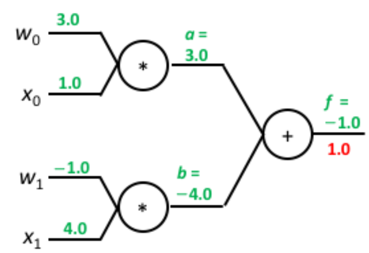
E.g., $w_0 = 3.0$, $w_1 = -1.0$, $x_0 = 1.0$, $x_1 = 4.0$



CSCE 496/896 Lecture 2: Basic Artificial Neural Networks
Stephen Scott
Introduction
Supervised Learning
Basic Units
Gradient Descent
Nonlinearly Separable Problems
Backprop
Computation Graphs
Sigmoid Unit
Multilayer Networks
Training Multilayer Networks
Backprop Alg
Types of Units

Computation Graphs

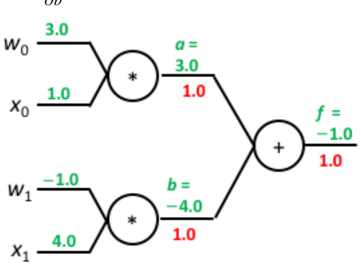
- So what?
- Can now decompose gradient calculation into basic operations
- $\frac{\partial f}{\partial f} = 1$



CSCE 496/896 Lecture 2: Basic Artificial Neural Networks
Stephen Scott
Introduction
Supervised Learning
Basic Units
Gradient Descent
Nonlinearly Separable Problems
Backprop
Computation Graphs
Sigmoid Unit
Multilayer Networks
Training Multilayer Networks
Backprop Alg
Types of Units

Computation Graphs

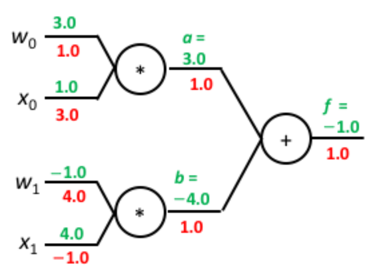
- If $g(y, z) = y + z$ then $\frac{\partial g}{\partial y} = \frac{\partial g}{\partial z} = 1$
- Via chain rule, $\frac{\partial f}{\partial a} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial a} = (1.0)(1.0) = 1.0$
- Same with $\frac{\partial f}{\partial b}$



CSCE 496/896 Lecture 2: Basic Artificial Neural Networks
Stephen Scott
Introduction
Supervised Learning
Basic Units
Gradient Descent
Nonlinearly Separable Problems
Backprop
Computation Graphs
Sigmoid Unit
Multilayer Networks
Training Multilayer Networks
Backprop Alg
Types of Units

Computation Graphs

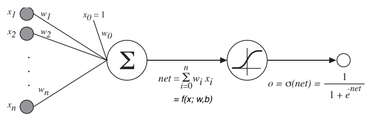
- If $h(y, z) = yz$ then $\frac{\partial h}{\partial y} = z$
- Via chain rule, $\frac{\partial f}{\partial x_0} = \frac{\partial f}{\partial a} \frac{\partial a}{\partial x_0} = 1.0 w_0 = 3.0$



So for $\mathbf{x} = [1.0, 4.0]^\top$, $\nabla f(\mathbf{w}) = [1.0, 4.0]^\top$

The Sigmoid Unit
Basics

- How does this help us with multi-layer ANNs?
- First, let's replace the threshold function with a continuous approximation



$\sigma(net)$ is the **logistic function**

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

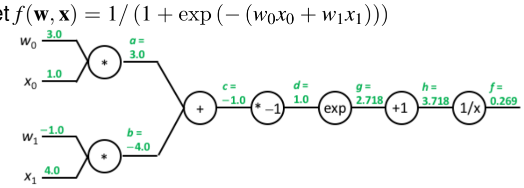
(a type of **sigmoid** function)

Squashes net into $[0,1]$ range

Nebraska Lincoln
CSCE 496/896 Lecture 2: Basic Artificial Neural Networks
Stephen Scott
Introduction
Supervised Learning
Basic Units
Gradient Descent
Nonlinearly Separable Problems
Backprop
Computation Graphs
Sigmoid Unit
Multilayer Networks
Training Multilayer Networks
Backprop Alg
Types of Units

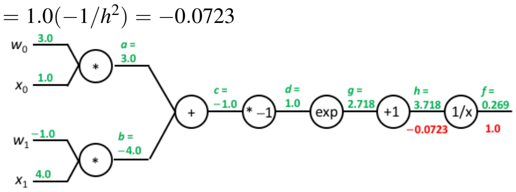
The Sigmoid Unit
Computation Graph

Let $f(\mathbf{w}, \mathbf{x}) = 1/\left(1 + \exp\left(-(w_0 x_0 + w_1 x_1)\right)\right)$



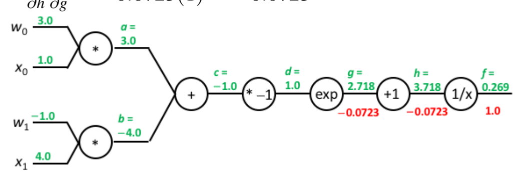
The Sigmoid Unit

$\frac{\partial f}{\partial h} = 1.0(-1/h^2) = -0.0723$



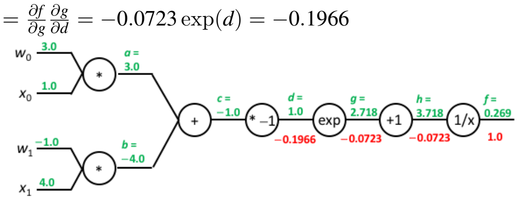
The Sigmoid Unit
Gradient

$\frac{\partial f}{\partial g} = \frac{\partial f}{\partial h}\frac{\partial h}{\partial g} = -0.0723(1) = -0.0723$



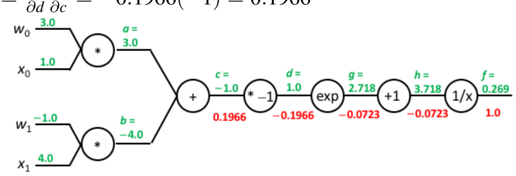
The Sigmoid Unit
Gradient

$\frac{\partial f}{\partial d} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial d} = -0.0723\exp(d) = -0.1966$

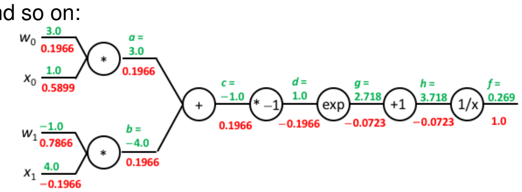


The Sigmoid Unit
Gradient

$\frac{\partial f}{\partial c} = \frac{\partial f}{\partial d}\frac{\partial d}{\partial c} = -0.1966(-1) = 0.1966$

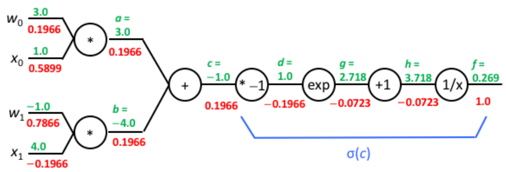


and so on:



So for $\mathbf{x} = [1.0, 4.0]^\top$, $\nabla f(\mathbf{w}) = [0.1966, 0.7866]^\top$

The Sigmoid Unit
Gradient



Note that $\frac{\partial f}{\partial c} = \sigma(c)(1-\sigma(c))$, so

$$\frac{\partial f}{\partial w_1} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial w_1} = \sigma(c)(1-\sigma(c))(1)x_1$$

This is **modular**, so once we have a formula for the gradient for this unit, we can apply it anywhere in a larger graph

Sigmoid Unit
Weight Update

- Let $\hat{y}^t = \sigma(\mathbf{w}\cdot\mathbf{x}^t)$ be prediction on training instance $\mathbf{x}^t$ with label y^t, and let loss be $J(\mathbf{w}) = \frac{1}{2}\left(\hat{y}^t - y^t\right)^2$, then

$$\begin{aligned}\frac{\partial J(\mathbf{w})}{\partial w_1} &= \left(\hat{y}^t - y^t\right)\left(\frac{\partial}{\partial w_1}\left(\hat{y}^t - y^t\right)\right)\\ &= \left(\hat{y}^t - y^t\right)\left(\frac{\partial}{\partial w_1}\hat{y}^t\right)\\ &= \left(\hat{y}^t - y^t\right)\left(\hat{y}^t\left(1-\hat{y}^t\right)x_1^t\right)\end{aligned}$$

- So update rule is

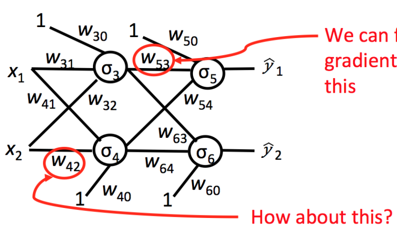
$$w_1' = w_1 - \eta\,\hat{y}^t\left(1-\hat{y}^t\right)\left(\hat{y}^t - y^t\right)x_1^t$$

- In general,

$$\boxed{\mathbf{w}' = \mathbf{w} - \eta\,\hat{y}^t\left(1-\hat{y}^t\right)\left(\hat{y}^t - y^t\right)\mathbf{x}^t}$$

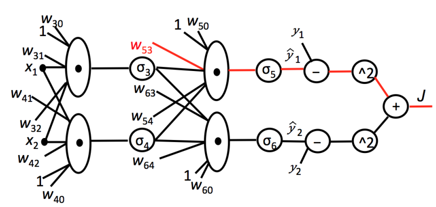
Multilayer Networks

- That update formula works for **output units** when we know the target labels $\mathbf{y}^t$ (here, a vector to encode **multi-class** labels)
- But for a **hidden unit**, we don't know its target output!



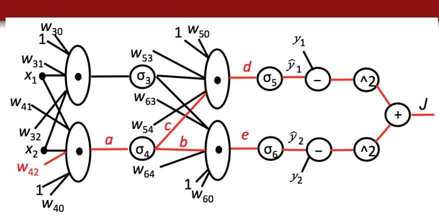
w_{ji} = weight from node i to node j

Training Multilayer Networks
Output Units



- Let loss on instance $(\mathbf{x}^t, \mathbf{y}^t)$ be $J(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}\left(\hat{y}_i^t - y_i^t\right)^2$
- Weights w_{5*} and w_{6*} tie to output units
- Gradients and weight updates done as before
- E.g., $w_{53}' = w_{53} - \eta\frac{\partial J}{\partial w_{53}} = w_{53} - \eta\hat{y}_1(1-\hat{y}_1)(\hat{y}_1 - y_1)\sigma_3$

Training Multilayer Networks
Hidden Units



Multivariate chain rule says we sum paths from J to w_{42}:

$$\begin{aligned}\frac{\partial J}{\partial w_{42}} &= \frac{\partial J}{\partial a}\frac{\partial a}{\partial w_{42}} = \left(\frac{\partial J}{\partial c}\frac{\partial c}{\partial a} + \frac{\partial J}{\partial b}\frac{\partial b}{\partial a}\right)\frac{\partial a}{\partial w_{42}}\\ &= \left(\frac{\partial J}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{\partial a} + \frac{\partial J}{\partial e}\frac{\partial e}{\partial b}\frac{\partial b}{\partial a}\right)\frac{\partial a}{\partial w_{42}}\\ &= \left([\hat{y}_1(1-\hat{y}_1)(\hat{y}_1 - y_1)]\,[w_{54}]\,[\sigma_4(a)(1-\sigma_4(a))]\right.\\ &+ \left.[\hat{y}_2(1-\hat{y}_2)(\hat{y}_2 - y_2)]\,[w_{64}]\,[\sigma_4(a)(1-\sigma_4(a))]\right)x_2\end{aligned}$$

Training Multilayer Networks
Hidden Units

- Analytical solution is messy, but we don't need the formula; only need to **compute** gradient
- The modular form of a computation graph means that once we've computed $\frac{\partial J}{\partial d}$ and $\frac{\partial J}{\partial e}$, we can plug those values in and compute gradients for earlier layers
  - Doesn't matter if layer is output, or farther back; can run indefinitely backward
- **Backpropagation** of error from outputs to inputs
- Define **error term** of hidden node h as

$$\delta_h \leftarrow \hat{y}_h\left(1-\hat{y}_h\right)\sum_{k\in down(h)} w_{k,h}\,\delta_k\ ,$$

where $\hat{y}_k$ is output of node k and $down(h)$ is set of nodes immediately downstream of h
- Note that this formula is specific to sigmoid units

Training Multilayer Networks
Hidden Units

- We are **propagating back** error terms δ from output layer toward input layers, scaling with the weights
- Scaling with the weights characterizes how much of the error term each hidden unit is "responsible for"
- Process:
  1. Submit inputs $\mathbf{x}$
  2. **Feed forward** signal to outputs
  3. Comptue network loss
  4. Propagate error back to compute loss gradient w.r.t. each weight
  5. Update weights

Backpropagation Algorithm
Sigmoid Activation Units and Square Loss

Initialize weights

Until termination condition satisfied do

- For each training example $(\boldsymbol{x}^t, \boldsymbol{y}^t)$ do
  1. Input $\boldsymbol{x}^t$ to the network and compute the outputs $\hat{\boldsymbol{y}}^t$
  2. For each output unit k
  $$\delta_k^t \leftarrow \hat{y}_k^t\,(1-\hat{y}_k^t)\,(y_k^t-\hat{y}_k^t)$$
  3. For each hidden unit h
  $$\delta_h^t \leftarrow \hat{y}_h^t\,(1-\hat{y}_h^t)\sum_{k\in down(h)} w_{k,h}^t\,\delta_k^t$$
  4. Update each network weight $w_{j,i}^t$
  $$w_{j,i}^t \leftarrow w_{j,i}^t + \Delta w_{j,i}^t$$
  where $\Delta w_{j,i}^t = \eta\,\delta_j^t\,x_{j,i}^t$ and $x_{j,i}^t$ is signal sent from node i to node j

Backpropagation Algorithm
Notes

- Formula for δ assumes sigmoid activation function
  - Straightforward to change to new activation function via computation graph
- Initialization used to be via random numbers near zero, e.g., from $\mathcal{N}(0,1)$
  - More refined methods available (later)
- Algorithm as presented updates weights after each instance
  - Can also accumulate $\Delta w_{j,i}^t$ across multiple training instances in the same **mini-batch** and do a single update per mini-batch
    - ⇒ **Stochastic gradient descent** (SGD)
  - Extreme case: Entire training set is a single batch (**batch gradient descent**)

Types of Output Units

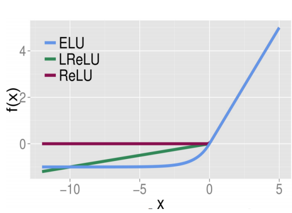
Given hidden layer outputs $\boldsymbol{h}$

- Linear unit: $\hat{y} = \boldsymbol{w}^\top \boldsymbol{h} + b$
  - Minimizing square loss with this output unit maximizes **log likelihood** when labels from normal distribution
    - I.e., find a set of parameters $\boldsymbol{\theta}$ that is most likely to generate the labels of the training data
  - Works well with GD training
- Sigmoid: $\hat{y} = \sigma(\boldsymbol{w}^\top \boldsymbol{h} + b)$
  - Approximates non-differentiable threshold function
  - More common in older, shallower networks
  - Can be used to predict probabilities
- Softmax unit: Start with $\boldsymbol{z} = W^\top \boldsymbol{h} + \boldsymbol{b}$
  - Predict probability of label i to be $\text{softmax}(\boldsymbol{z})_i = \exp(z_i)/\left(\sum_j \exp(z_j)\right)$
  - Continuous, differentiable approximation to argmax

Types of Hidden Units

Rectified linear unit (ReLU): $\max\{0, \boldsymbol{w}^\top \boldsymbol{x} + \boldsymbol{b}\}$

- Good default choice
- In general, GD works well when functions nearly linear
- Variations: **leaky ReLU** and **exponential ReLU** replace $z < 0$ side with $0.01z$ and $\alpha(\exp(z) - 1)$, respectively



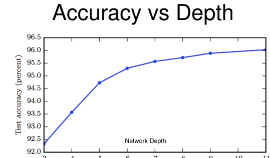
Logistic sigmoid (done already) and $\tanh$

- Nice approximation to threshold, but don't train well in deep networks since they saturate

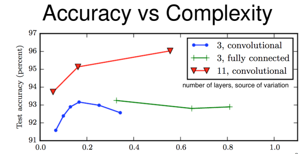
Putting Everything Together
Hidden Layers

- How many layers to use?
  - Deep networks build potentially useful representations of data via composition of simple functions
  - Performance improvement not simply from more complex network (number of parameters)
  - Increasing number of layers still increases chances of overfitting, so need significant amount of training data with deep network; training time increases as well

Accuracy vs Depth



Accuracy vs Complexity



Putting Everything Together
Universal Approximation Theorem

- Any boolean function can be represented with two layers
- Any bounded, continuous function can be represented with arbitrarily small error with two layers
- Any function can be represented with arbitrarily small error with three layers

Only an **EXISTENCE PROOF**

- Could need exponentially many nodes in a layer
- May not be able to find the right weights
- Highlights risk of overfitting and need for **regularization**

Putting Everything Together
Initialization

- Previously, initialized weights to random numbers near 0 (from $\mathcal{N}(0,1)$)
  - Sigmoid nearly linear there, so GD expected to work better
  - But in deep networks, this increases variance per layer, resulting in **vanishing gradients** and poor optimization
- **Glorot initialization** controls variance per layer: If layer has n_{in} inputs and n_{out} outputs, initialize via uniform over $[-r, r]$ or $\mathcal{N}(0, \sigma)$
  - $r = a\sqrt{\frac{6}{n_{in}+n_{out}}}$ and $\sigma = a\sqrt{\frac{2}{n_{in}+n_{out}}}$

Activation	a
Logistic	1
$\tanh$	4
ReLU	$\sqrt{2}$

Putting Everything Together
Optimizers

Variations on gradient descent optimization:

- Momentum optimization
- AdaGrad
- RMSProp
- Adam

Putting Everything Together
Momentum Optimization

- Use a **momentum** term β to keep updates moving in same direction as previous trials
- Replace original GD update $\mathbf{w}' = \mathbf{w} - \eta\nabla J(\mathbf{w})$ with

$$\mathbf{w}' = \mathbf{w} - \mathbf{m} \ ,$$

  where

$$\mathbf{m} = \beta\mathbf{m} + \eta\nabla J(\mathbf{w})$$

- Using sigmoid activation and square loss, replace $\Delta w_{ji}^t = \eta\,\delta_j^t\,x_{ji}^t$ with

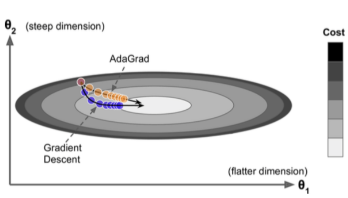
$$\Delta w_{ji}^t = \eta\,\delta_j^t\,x_{ji}^t + \beta\,\Delta w_{ji}^{t-1}$$

- Can help move through small local minima to better ones & move along flat surfaces

Putting Everything Together
AdaGrad

- Standard GD can too quickly descend steepest slope, then slowly crawl through a valley
- **AdaGrad** adapts learning rate by scaling it down in steepest dimensions:
  $\mathbf{w}' = \mathbf{w} - \eta\nabla J(\mathbf{w}) \oslash \sqrt{\mathbf{s}+\epsilon}$, where
  $\mathbf{s} = \mathbf{s} + \nabla J(\mathbf{w}) \otimes \nabla J(\mathbf{w})$,
  $\otimes$ and $\oslash$ are element-wise multiplication and division and $\epsilon = 10^{-10}$ prevents division by 0

$\mathbf{s}$ accumulates squares of gradient, and learning rate for each dimension scaled down



Putting Everything Together
RMSProp

- AdaGrad tends to stop too early for neural networks due to over-aggressive downscaling
- **RMSProp** exponentially decays old gradients to address this

$$\mathbf{w}' = \mathbf{w} - \eta\nabla J(\mathbf{w}) \oslash \sqrt{\mathbf{s}+\epsilon} \ ,$$

  where

$$\mathbf{s} = \beta\mathbf{s} + (1-\beta)\nabla J(\mathbf{w}) \otimes \nabla J(\mathbf{w})$$

Introduction
Supervised Learning
Basic Units
Gradient Descent
Nonlinearly Separable Problems
Backprop
Types of Units
Putting Things Together

Adam (adaptive moment estimation) combines Momentum optimization and RMSProp

1. $\mathbf{m} = \beta_1 \mathbf{m} + (1 - \beta_1) \nabla J(\mathbf{w})$
2. $\mathbf{s} = \beta_2 \mathbf{s} + (1 - \beta_2) \nabla J(\mathbf{w}) \otimes \nabla J(\mathbf{w})$
3. $\mathbf{m} = \mathbf{m}/(1 - \beta_1^t)$
4. $\mathbf{s} = \mathbf{s}/(1 - \beta_2^t)$
5. $\mathbf{w}' = \mathbf{w} - \eta \mathbf{m} \oslash \sqrt{\mathbf{s} + \epsilon}$

- Iteration counter t used in 3 and 4 to prevent $\mathbf{m}$ and $\mathbf{s}$ from vanishing
- Can set $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$