Introduction

Outline

Basic Units

Nonlinearly Separable Problems

Backprop

Types of Units

Putting Things Together

CSCE 970 Lecture 2: Artificial Neural Networks and Deep Learning

Stephen Scott and Vinod Variyam

(Adapted from Ethem Alpaydin and Tom Mitchell)

sscott@cse.unl.edu

Introduction
Properties
History
Outline
Basic Units
Nonlinearly Separable Problems
Backprop
Types of Units
Putting Things Together

Introduction

Deep learning is based in *artificial neural networks*

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

CSCE 970 Lecture 2: Artificial Neural Networks and Deep Learning

Stephen Scott and Vinod Variyam

Introduction
Properties
History
Outline
Basic Units
Nonlinearly Separable Problems
Backprop
Types of Units
Putting Things Together

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

CSCE 970 Lecture 2: Artificial Neural Networks and Deep Learning

Stephen Scott and Vinod Variyam

Introduction
Properties
History
Outline
Basic Units
Nonlinearly Separable Problems
Backprop
Types of Units
Putting Things Together

- **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

Introduction
History of ANNs (cont'd)

- **The Resurgence:** Deep architectures (2000s)
  - Better hardware 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 Problem:** Skynet (TBD)
  - Sorry.

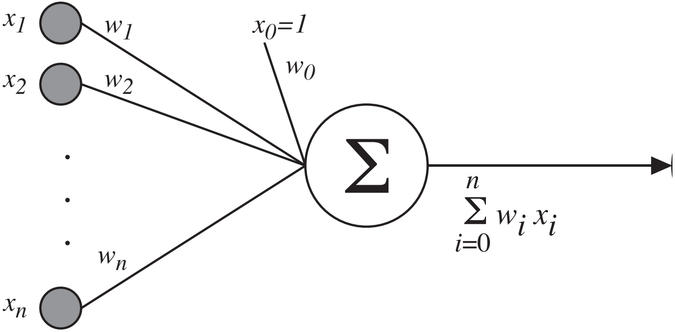
When to Consider ANNs

- 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

Outline

- Basic units
    - Linear unit
    - Linear threshold units
    - Perceptron training rule
- Nonlinearly separable problems and multilayer networks
- Backpropagation
- Putting everything together

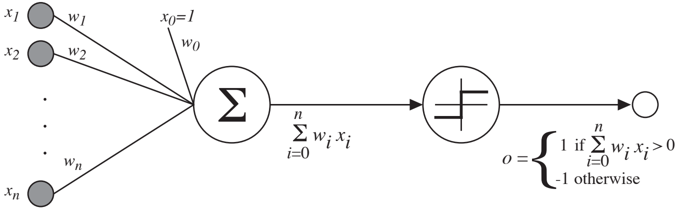
Linear Unit



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

- If set $w_0 = b$, can simplify above
- Forms the basis for many other activation functions

Linear Threshold Unit



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

(sometimes use 0 instead of -1)

Linear Threshold Unit
Decision Surface

(a) (b)

Represents some useful functions

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

But some functions not representable

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

Perceptron Training Rule

$$w_j^{t+1} \leftarrow w_j^t + \Delta w_j^t \ , \text{ where } \Delta w_j^t = \eta \left(y^t - \hat{y}^t\right) x_j^t$$

and

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

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

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

Where Does the Training Rule Come From?

- Recall initial *linear unit*, where output

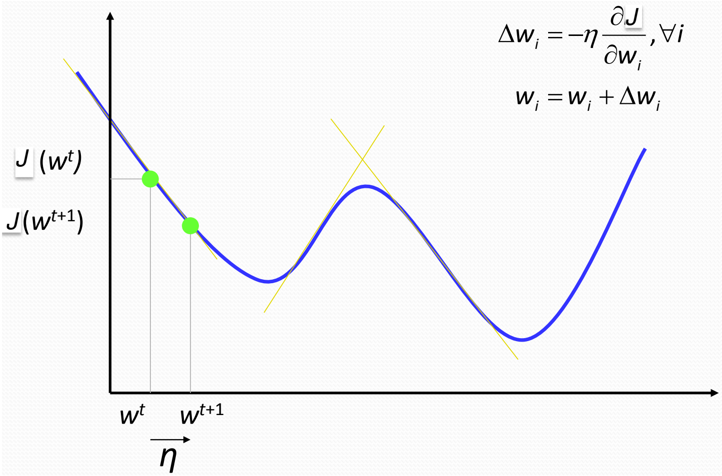
$$\hat{y}^t = f(\mathbf{x}; \mathbf{w}, b) = \mathbf{x}^\top \mathbf{w} + b$$

(i.e., no threshold)

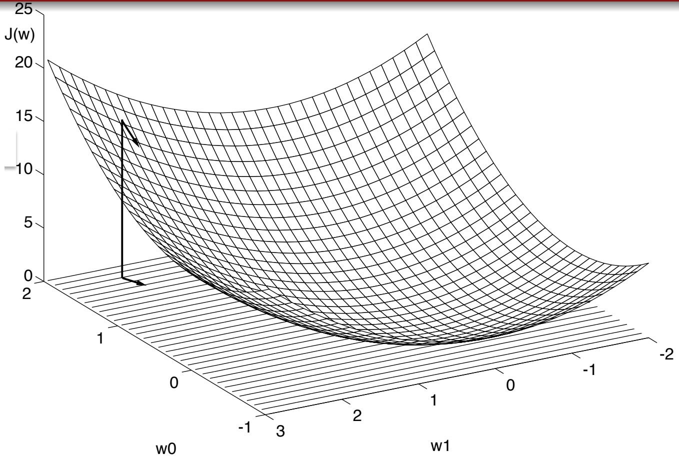
- For each training example, compromise between **correctiveness** and **conservativeness**
  - **Correctiveness:** Tendency to improve on $\mathbf{x}^t$ (reduce loss)
  - **Conservativeness:** Tendency to keep $\mathbf{w}^{t+1}$ close to $\mathbf{w}^t$ (minimize distance)
- Use **cost function** that measures both (let $w_0 = b$):

$$J(\mathbf{w}) = dist\left(\mathbf{w}^{t+1}, \mathbf{w}^t\right) + \eta\, loss\left(y^t, \overbrace{\mathbf{w}^{t+1} \cdot \mathbf{x}^t}^{\text{curr ex, new wts}}\right)$$

Gradient Descent



Gradient Descent (cont'd)



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

Gradient Descent (cont'd)

$$\begin{aligned} J(\mathbf{w}) &= \overbrace{\|\mathbf{w}^{t+1} - \mathbf{w}^t\|_2^2}^{conserv.} + \overbrace{\eta}^{coef.} \overbrace{(y^t - \mathbf{w}^{t+1} \cdot \mathbf{x}^t)^2}^{corrective} \\ &= \sum_{j=1}^{n} \left(w_j^{t+1} - w_j^t\right)^2 + \eta \left(y^t - \sum_{j=1}^{n} w_j^{t+1} x_j^t\right)^2 \end{aligned}$$

Take gradient w.r.t. $\mathbf{w}^{t+1}$ (i.e., $\partial J / \partial w_i^{t+1}$) and set to $\mathbf{0}$:

$$0 = 2\left(w_i^{t+1} - w_i^t\right) - 2\eta \left(y^t - \sum_{j=1}^{n} {\color{red} w_j^{t+1}} x_j^t\right) x_i^t$$

Gradient Descent (cont'd)

Approximate with

$$0 = 2\left(w_i^{t+1} - w_i^t\right) - 2\eta\left(y^t - \sum_{j=1}^{n} w_j^t x_j^t\right) x_i^t \ ,$$

which yields

$$w_i^{t+1} = w_i^t + \overbrace{\eta\left(y^t - \hat{y}^t\right) x_i^t}^{\Delta w_i^t}$$

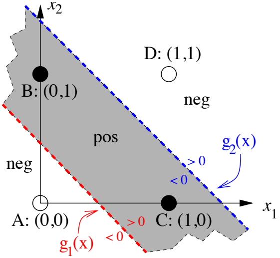
Given other loss function and unit activation function, can derive weight update rule

CSCE 970 Lecture 2: Artificial Neural Networks and Deep Learning

Stephen Scott and Vinod Variyam

Introduction
Outline
Basic Units
Nonlinearly Separable Problems
XOR
General Nonlinearly Separable Problems
Backprop
Types of Units
Putting Things Together

Using linear threshold units (book uses rectified linear units)



Represent with **intersection** of two linear separators

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

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

pos $= \{\mathbf{x} \in \mathbb{R}^2 : g_1(\mathbf{x}) > 0$ <u>AND</u> $g_2(\mathbf{x}) < 0\}$

neg $= \{\mathbf{x} \in \mathbb{R}^2 : g_1(\mathbf{x}), g_2(\mathbf{x}) < 0$ <u>OR</u> $g_1(\mathbf{x}), g_2(\mathbf{x}) > 0\}$

CSCE 970 Lecture 2: Artificial Neural Networks and Deep Learning

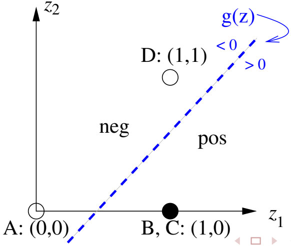
Stephen Scott and Vinod Variyam

Introduction
Outline
Basic Units
Nonlinearly Separable Problems
XOR
General Nonlinearly Separable Problems
Backprop
Types of Units
Putting Things Together

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

Class	(x_1, x_2)	$g_1(\mathbf{x})$	z_1	$g_2(\mathbf{x})$	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 $g(\mathbf{z}) = 1 \cdot z_1 - 2 \cdot z_2 - 1/2$

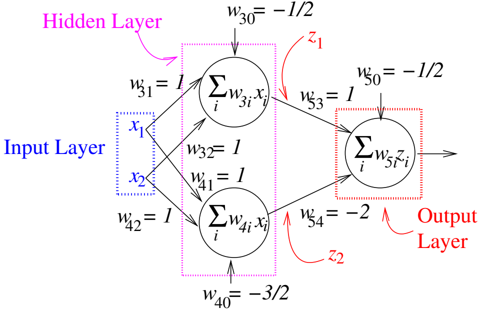


CSCE 970 Lecture 2: Artificial Neural Networks and Deep Learning

Stephen Scott and Vinod Variyam

Introduction
Outline
Basic Units
Nonlinearly Separable Problems
XOR
General Nonlinearly Separable Problems
Backprop
Types of Units
Putting Things Together

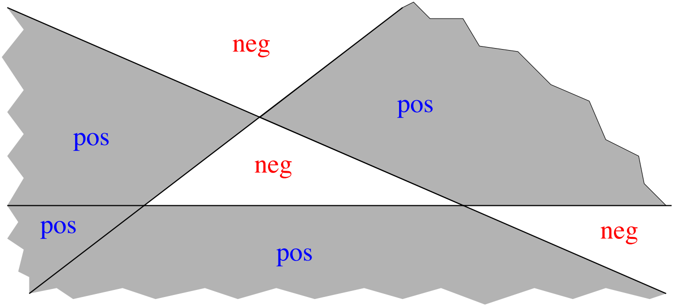
In other words, we **remapped** all vectors $\mathbf{x}$ to $\mathbf{z}$ such that the classes are linearly separable in the new vector space



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

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

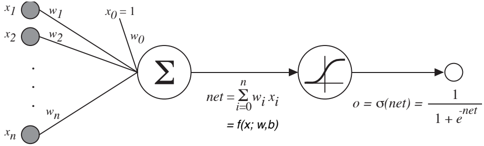
By adding up to 2 **hidden layers** of linear threshold units, can represent any **union** of **intersection of halfspaces**



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

The Sigmoid Unit

[Rarely used in deep ANNs, but continuous and differentiable]



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

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

(a type of **sigmoid** function)

Squashes net into $[0, 1]$ range

Nice property:

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

CSCE 970 Lecture 2: Artificial Neural Networks and Deep Learning

Stephen Scott and Vinod Variyam

Introduction
Outline
Basic Units
Nonlinearly Separable Problems
Backprop
- Sigmoid Unit
- Multilayer Networks
- Training Multilayer Networks
- Backprop Alg

Types of Units
Putting Things Together

Again, use squared loss for correctiveness:

$$E(\mathbf{w}^t) = \frac{1}{2}\left(y^t - \hat{y}^t\right)^2$$

(folding $1/2$ of correctiveness into loss func)

Thus $\dfrac{\partial E}{\partial w_j^t} = \dfrac{\partial}{\partial w_j^t}\dfrac{1}{2}\left(y^t - \hat{y}^t\right)^2$

$$= \frac{1}{2}\, 2\left(y^t - \hat{y}^t\right)\frac{\partial}{\partial w_j^t}\left(y^t - \hat{y}^t\right) = \left(y^t - \hat{y}^t\right)\left(-\frac{\partial \hat{y}^t}{\partial w_j^t}\right)$$

Sigmoid Unit
Gradient Descent (cont'd)

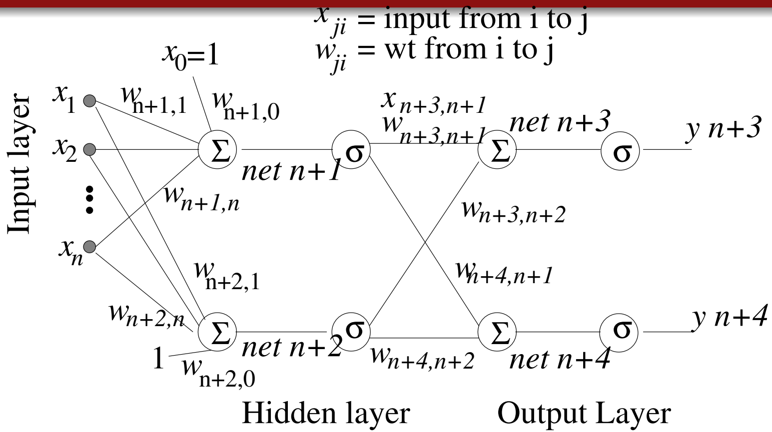
Since $\hat{y}^t$ is a function of $f(\mathbf{x}; \mathbf{w}, b) = net^t = \mathbf{w}^t \cdot \mathbf{x}^t$,

$$\begin{aligned}\frac{\partial E}{\partial w_j^t} &= -\left(y^t - \hat{y}^t\right) \frac{\partial \hat{y}^t}{\partial net^t} \frac{\partial net^t}{\partial w_j^t} \\ &= -\left(y^t - \hat{y}^t\right) \frac{\partial \sigma\left(net^t\right)}{\partial net^t} \frac{\partial net^t}{\partial w_j^t} \\ &= -\left(y^t - \hat{y}^t\right) \hat{y}^t \left(1 - \hat{y}^t\right) x_j^t\end{aligned}$$

Update rule:

$$\boxed{w_j^{t+1} = w_j^t + \eta\, \hat{y}^t \left(1 - \hat{y}^t\right) \left(y^t - \hat{y}^t\right) x_j^t}$$

Multilayer Networks



For now, using sigmoid units

$$E^t = E(\mathbf{w}^t) = \frac{1}{2} \sum_{k \in outputs} \left(y_k^t - \hat{y}_k^t\right)^2$$

Adjust weight w_{ji}^t according to E^t as before

For output units, this is easy since contribution of w_{ji}^t to E^t when j is an output unit is the same as for single neuron case[1], i.e.,

$$\frac{\partial E^t}{\partial w_{ji}^t} = -\left(y_j^t - \hat{y}_j^t\right) \hat{y}_j^t \left(1 - \hat{y}_j^t\right) x_{ji}^t = -\delta_j^t \, x_{ji}^t$$

where $\delta_j^t = -\frac{\partial E^t}{\partial net_j^t} =$ **error term** of unit j

[1] This is because all other outputs are constants w.r.t. w_{ji}^t

CSCE 970 Lecture 2: Artificial Neural Networks and Deep Learning

Stephen Scott and Vinod Variyam

Introduction
Outline
Basic Units
Nonlinearly Separable Problems
Backprop
Sigmoid Unit
Multilayer Networks
Training Multilayer Networks
Backprop Alg
Types of Units
Putting Things Together

- How can we compute the error term for hidden layers when there is no target output $\mathbf{r}^t$ for these layers?
- Instead **propagate back** error values 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"

CSCE 970 Lecture 2: Artificial Neural Networks and Deep Learning

Stephen Scott and Vinod Variyam

Introduction
Outline
Basic Units
Nonlinearly Separable Problems
Backprop
Sigmoid Unit
Multilayer Networks
Training Multilayer Networks
Backprop Alg
Types of Units
Putting Things Together

The impact that w_{ji}^t has on E^t is only through net_j^t and units immediately "downstream" of j:

$$\frac{\partial E^t}{\partial w_{ji}^t} = \frac{\partial E^t}{\partial net_j^t} \frac{\partial net_j^t}{\partial w_{ji}^t} = x_{ji}^t \sum_{k \in down(j)} \frac{\partial E^t}{\partial net_k^t} \frac{\partial net_k^t}{\partial net_j^t}$$

$$= x_{ji}^t \sum_{k \in down(j)} -\delta_k^t \frac{\partial net_k^t}{\partial net_j^t} = x_{ji}^t \sum_{k \in down(j)} -\delta_k^t \frac{\partial net_k^t}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial net_j^t}$$

$$= x_{ji}^t \sum_{k \in down(j)} -\delta_k^t \, w_{kj} \frac{\partial \hat{y}_j}{\partial net_j^t} = x_{ji}^t \sum_{k \in down(j)} -\delta_k^t \, w_{kj} \, \hat{y}_j \, (1 - \hat{y}_j)$$

Works for arbitrary number of hidden layers

Backpropagation Algorithm

Initialize all weights to small random numbers

Until termination condition satisfied do

- For each training example $(\mathbf{y}^t, \mathbf{x}^t)$ do
  1. Input $\mathbf{x}^t$ to the network and compute the outputs $\hat{\mathbf{y}}^t$
  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

$$\Delta w_{j,i}^t = \eta \, \delta_j^t \, x_{j,i}^t$$

Backpropagation Algorithm

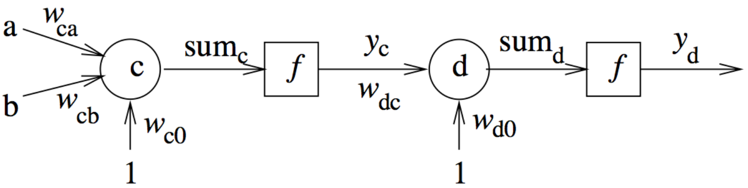
Example

target = r

$f(x) = 1 / (1 + \exp(-x))$

trial 1: a = 1, b = 0, r = 1

trial 2: a = 0, b = 1, r = 0



eta	0.3		
	trial 1	trial 2	
w_ca	0.1	0.1008513	0.1008513
w_cb	0.1	0.1	0.0987985
w_c0	0.1	0.1008513	0.0996498
a	1	0	
b	0	1	
const	1	1	
sum_c	0.2	0.2008513	
y_c	0.5498340	0.5500447	
w_dc	0.1	0.1189104	0.0964548
w_d0	0.1	0.1343929	0.0935679
sum_d	0.1549834	0.1997990	
y_d	0.5386685	0.5497842	

target	1	0
delta_d	0.1146431	-0.136083
delta_c	0.0028376	-0.004005

delta_d(t) = y_d(t) * (r(t) - y_d(t)) * (1 - y_d(t))

delta_c(t) = y_c(t) * (1 - y_c(t)) * delta_d(t) * w_dc(t

w_dc(t+1) = w_dc(t) + eta * y_c(t) * delta_d(t)

w_ca(t+1) = w_ca(t) + eta * a * delta_c(t)

Types of Output Units

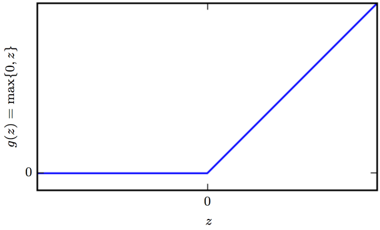
Given hidden layer outputs $\mathbf{h}$

- Linear unit (Sec 6.2.2.1): $\hat{y} = \mathbf{w}^\top \mathbf{h} + b$
  - Minimizing square loss with this output unit maximizes **log likelihood** when labels from normal distribution
    - I.e., find a set of parameters θ that is most likely to generate the labels of the training data
  - Works well with GD training
- Sigmoid (Sec 6.2.2.2): $\hat{y} = \sigma(\mathbf{w}^\top \mathbf{h} + b)$
  - Approximates non-differentiable threshold function
  - More common in older, shallower networks
  - Can be used to predict probabilities
- Softmax unit (Sec 6.2.2.3): Start with $\mathbf{z} = W^\top \mathbf{h} + \mathbf{b}$
  - Predict probability of label i to be $\text{softmax}(z)_i = \exp(z_i) / \left(\sum_j \exp(z_j) \right)$
  - Continuous, differentiable approximation to argmax

Types of Hidden Units

Rectified linear unit (ReLU) (Sec 6.3.1): $\max\{0, W^\top \mathbf{x} + \mathbf{b}\}$

- Good default choice
- Second derivative is 0 almost everywhere and derivatives large
- In general, GD works well when functions nearly linear



Logistic sigmoid (done already) and tanh (6.3.2)

- Nice approximation to threshold, but don't train well in deep networks
- Still potentially useful when piecewise functions inappropriate

Softmax (occasionally used as hidden)

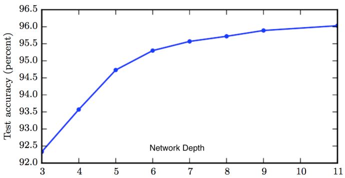
CSCE 970 Lecture 2: Artificial Neural Networks and Deep Learning

Stephen Scott and Vinod Variyam

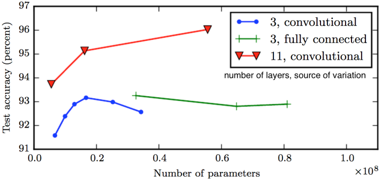
Introduction
Outline
Basic Units
Nonlinearly Separable Problems
Backprop
Types of Units
Putting Things Together
Hidden Layers

- How many layers to use?
  - Deep networks tend to build potentially useful representations of the 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



Introduction
Outline
Basic Units
Nonlinearly Separable Problems
Backprop
Types of Units
Putting Things Together
Hidden Layers

- 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