CRIPAC
智能感知与计算研究中心
Center for Research on Intelligent Perception and Computing

中国科学院自动化研究所
Institute of Automation
Chinese Academy of Sciences

2019

"Deep Learning Lecture"

Lecture 3 : Deep Feedforward Network

Wang Liang

Center for Research on Intelligent Perception and Computing (CRIPAC)

National Laboratory of Pattern Recognition (NLPR)

Institute of Automation, Chinese Academy of Science (CASIA)

Outline

1. Course Review
2. Overview
3. Basic Components
4. Architectural Considerations
5. Back-Propagation

Review: Linear Algebra

- Scalar, Vector, Matric, Tensor
- Matrix Transpose, Matrix (Dot) Product, Identity Matrix
- Systems of Equations, Solving Systems of Equations
- Matrix Inversion, Invertibility, Norm
- Eigendecomposition, Trace

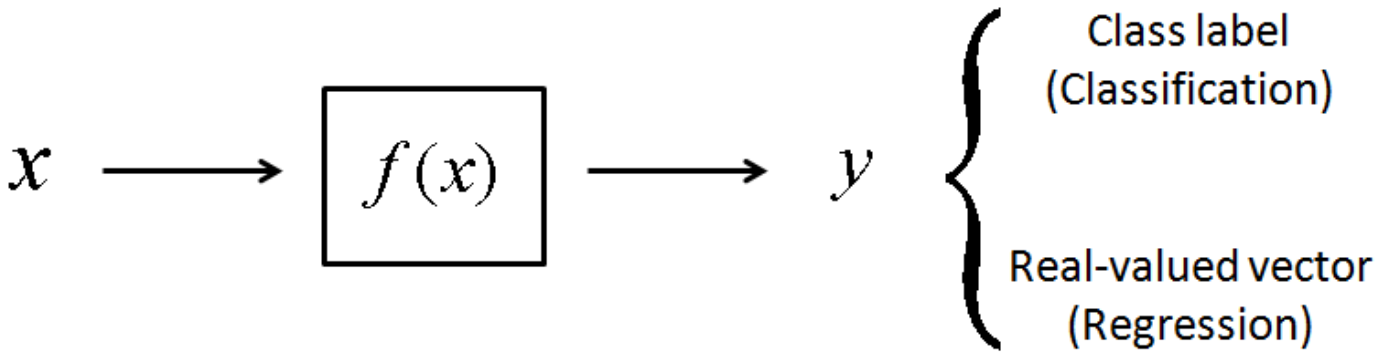
Review: Probability and Information Theory

- Probability Mass Function, Probability Density Function
- Marginal Probability, Conditional Probability
- Chain Rule of Probability, Independence
- Expectation, Variance and Covariance

Review: Numerical concerns for DL

- Algorithms are often specified in terms of real numbers; real numbers cannot be implemented in a finite computer
  - Does the algorithm still work when implemented with a finite number of bits?
- Do small changes in the input to a function cause large changes to an output?
  - Rounding errors, noise, measurement errors can cause large changes
  - Iterative search for best input is difficult

Review: Machine Learning Basics



Object recognition

{dog, cat, horse, flower, …}

Low-resolution image

Super resolution

High-resolution image

Outline

1. Course Review
2. Overview
3. Basic Components
4. Architectural Considerations
5. Back-Propagation

Historical Background

- Pioneering work on the mathematical model of neural networks
  - McCulloch and Pitts 1943
  - Include recurrent and non-recurrent (with “circles”) networks
  - Use thresholding function as nonlinear activation
  - No learning
- Early works on learning neural networks
  - Starting from Rosenblatt 1958
  - Using thresholding function as nonlinear activation prevented computing derivatives with the chain rule, and so errors could not be propagated back to guide the computation of gradients
- Backpropagation was developed in several steps since 1960
  - The key idea is to use the chain rule to calculate derivatives
  - It was reflected in multiple works, earliest from the field of control

Historical Background

- Standard backpropagation for neural networks
  - Rumelhart, Hinton, and Williams, Nature 1986. Clearly appreciated the power of backpropagation and demonstrated it on key tasks, and applied it to pattern recognition generally
  - In 1985, Yann LeCun independently developed a learning algorithm for three-layer networks in which target values were propagated, rather than derivatives. In 1986, he proved that it was equivalent to standard backpropagation
- Prove the universal expressive power of three-layer neural networks
  - Hecht-Nielsen 1989
- Convolutional neural network
  - Introduced by Kunihiko Fukushima in 1980
  - Improved by LeCun, Bottou, Bengio, and Haffner in 1998

Historical Background

- Deep belief net (DBN)
  - Hinton, Osindero, and Tech 2006
- Auto encoder
  - Hinton and Salakhutdinov 2006 (Science)
- Deep learning
  - Hinton. Learning multiple layers of representations. Trends in Cognitive Sciences, 2007
  - Unsupervised multilayer pre-training + supervised fine-tuning (BP)
- Large-scale deep learning in speech recognition
  - Geoff Hinton and Li Deng started this research at Microsoft Research Redmond in late 2009.
  - Generative DBN pre-training was not necessary
  - Success was achieved by large-scale training data + large deep neural network (DNN) with large, context-dependent output layers

Historical Background

- Unsupervised deep learning from large scale images
  - Andrew Ng et al. 2011
  - Unsupervised feature learning
  - 16000 CPUs
- Large-scale supervised deep learning in ImageNet image Classification
  - Krizhevsky, Sutskever, and Hinton 2012
  - Supervised learning with convolutional neural network
  - No unsupervised pre-training

Classic application

- Application of ANN
  - Function approximation (modelling)
  - Pattern classification (analysis of time-series, customer databases, etc)
  - Object recognition (e.g. character recognition)
  - Data compression
  - Security (credit card fraud)
  - ……

Classic application

- Pattern Classification
  - In some literature, the set of all input values is called the input pattern, and the set of output values the output pattern

$$x = (x_1, \cdots, x_m) \rightarrow y = (y_1, \cdots, y_n)$$

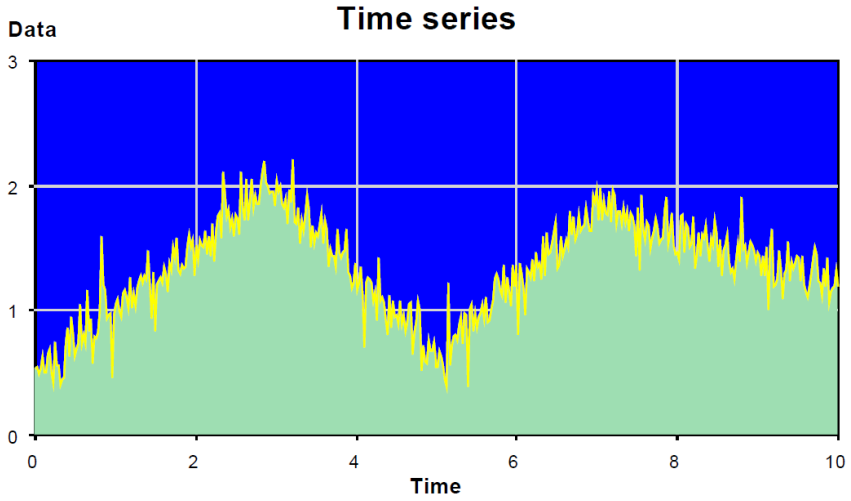
  - A neural network 'learns' the relation between different input and output patterns
  - Thus, a neural network performs pattern classification or pattern recognition (i.e. classifies inputs into output categories)

Classic application

- Time Series Analysis

  A time series is a recording of some variable (e.g. a share price, temperature) at different time moments:

$$x(t_1), x(t_2), \cdots, x(t_m)$$



- The aim of the analysis is to learn to predict the future values

Classic application

Time Series (Cont.)

- We may use a neural network to analyse time series:

  Input: consider m values in the past $x(t_1), x(t_2), \cdots, x(t_m)$ as m input variables.

  Output: consider n future values $\mathrm{y}(t_{m+1}), y(t_{m+2}), \cdots, y(t_{m+n})$ as n output variables.
- Our goal is to find the following model: $(\mathrm{y}(t_{m+1}), y(t_{m+2}), \cdots, y(t_{m+n})) = f(x(t_1), x(t_2), \cdots, x(t_m))$
- By training a neural network with m inputs and n outputs on the time series data, we can create such a model

Advantages and Limitations

- Advantages of Neural Networks:
  - Can be applied to many problems, as long as there is some data
  - Can be applied to problems, for which analytical methods do not yet exist
  - Can be used to model non-linear dependencies
  - If there is a pattern, then neural networks should quickly work it out, even if the data is ‘noisy’
  - Always gives some answer even when the input information is not complete
  - Networks are easy to maintain

Advantages and Limitations

- Limitations of Neural Networks:
  - Like with any data-driven models, they cannot be used if there is no or very little data available
  - There are many free parameters, such as the number of hidden nodes, the learning rate, minimal error, which may greatly influence the final result.
  - Not good for arithmetics and precise calculations
  - Neural networks do not provide explanations. If there are many nodes, then there are too many weights that are difficult to interprete (unlike the slopes in linear models, which can be seen as correlations). In some tasks, explanations are crucial (e.g. air traffic control, medical diagnosis).

Feedforward Neural Network

- Goal: Approximate some unknown ideal function

$$f^* : \mathcal{X} \rightarrow \mathcal{Y}$$

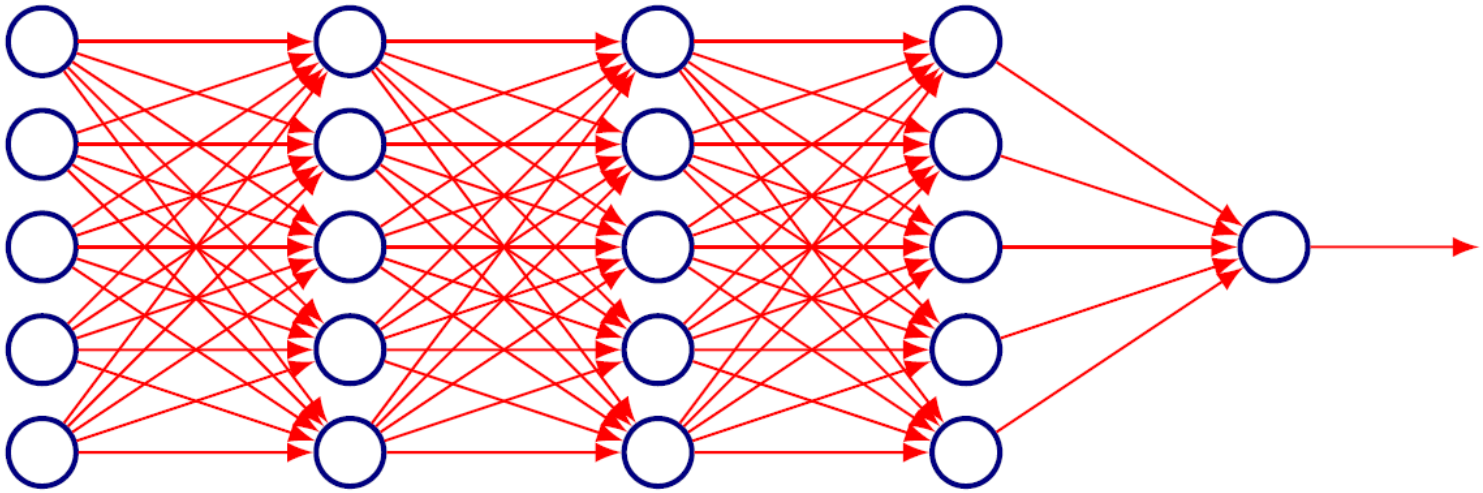
- Ideal classier: $y = f^*(\mathbf{x})$ with x and category y
- Feedforward Network: Define parametric mapping

$$y = f(\mathbf{x}, \theta)$$

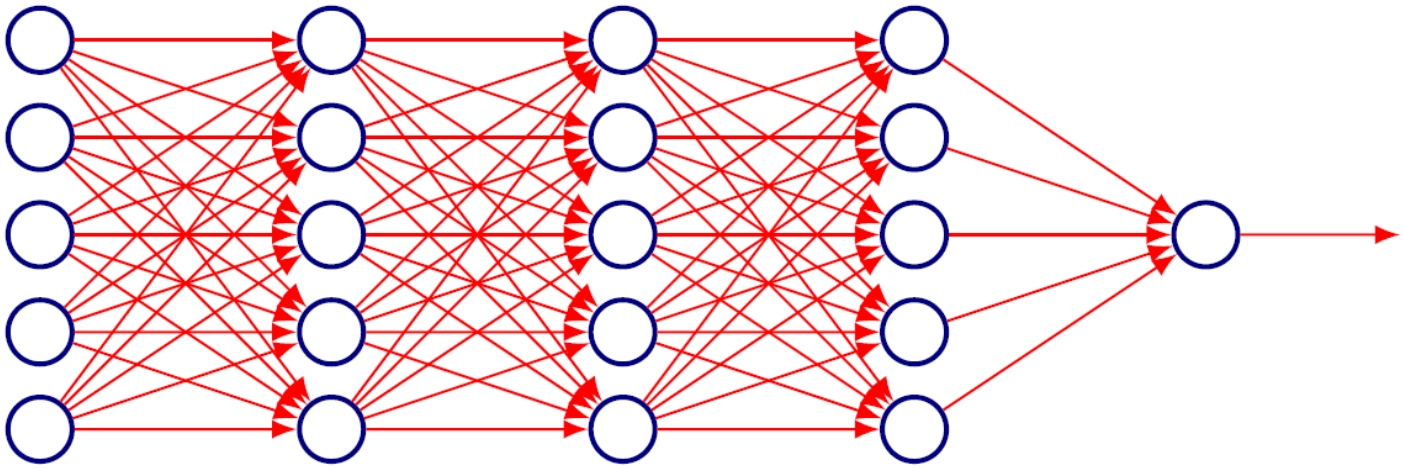
- Learn parameters θ to get a good approximation to f* from available sample
- Naming: Information flow in function evaluation begins at input, flows through intermediate computations (that define the function), to produce the category
- No feedback connections (Recurrent Networks!)

Feedforward Neural Network

- Function f is a composition of many different functions $f(\mathbf{x}) = f^{(3)}(f^{(2)}|(f^{(1)}(\mathbf{x})))$
- e.g.



Feedforward Neural Network



- Function composition can be described by a directed acyclic graph (hence feedforward networks)
- $f^{(1)}$ is the first layer, f^2 the second layer and so on.
- Depth is the maximum i in the function composition chain
- Final layer is called the *output* layer

Feedforward Neural Network

- **Training**: Optimize θ to drive f(x; θ) closer to f*(x)
- **Training Data**: f* evaluated at different x instances (i.e. expected outputs)
- Only specify the output of the *output* layers
- Output of intermediate layers is not specified by $\mathcal{D}$, hence the nomenclature *hidden* layers
- **Neural**: Choices of $f^{(i)}$'s and layered organization, loosely inspired by neuroscience

Back to Linear Models

- Optimization is convex or closed form !
- Model can't understand interaction between input variables!
- **Extension**: Do nonlinear transformation $\mathbf{x} \rightarrow \phi(\mathbf{x})$; apply linear model to $\phi(\mathbf{x})$
- ϕ gives features or a representation for x
- How do we choose ϕ ?

Choosing ϕ

- Option 1: Use a generic ϕ
- **Example**: Infinite dimensional ϕ implicitly used by kernel machines with RBF kernel
- **Positive**: Enough capacity to fit training data
- **Negative**: Poor generalization for highly varying f*
- Prior used: Function is locally smooth.

Choosing ϕ

- Option 2: Engineer ϕ for problem
- Still convex!

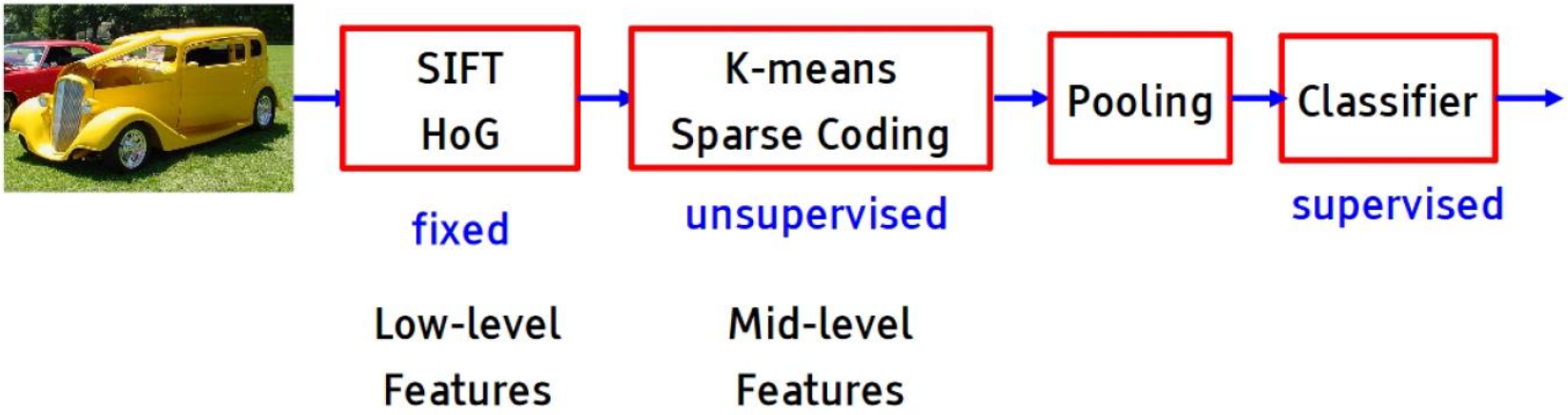
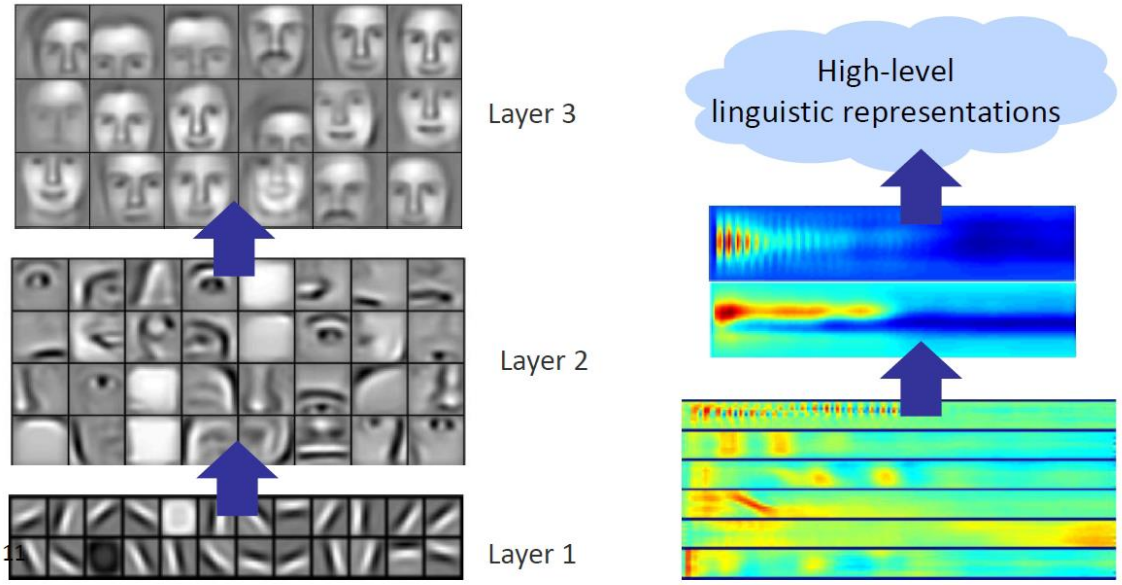


Illustration: Yann LeCun

Choosing ϕ

- Option 3: Learn ϕ from data
- Gives up on convexity
- Combines good points of first two approaches: ϕ can be highly generic and the engineering effort can go into architecture



Design Decisions

- Need to choose optimizer, cost function and form of output
- Choosing activation functions
- Architecture design (number of layers etc)

XOR Example

XOR

Exclusive-OR gate

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

- Let XOR be the target function f*(x) that we want to learn
- We will adapt parameters θ for f(x;θ) to try and represent f*
- Our Data:

$$(X, Y) = \{([0, 0]^T, 0), ([0, 1]^T, 1), ([1, 0]^T, 1), ([1, 1]^T, 0)\}$$

XOR

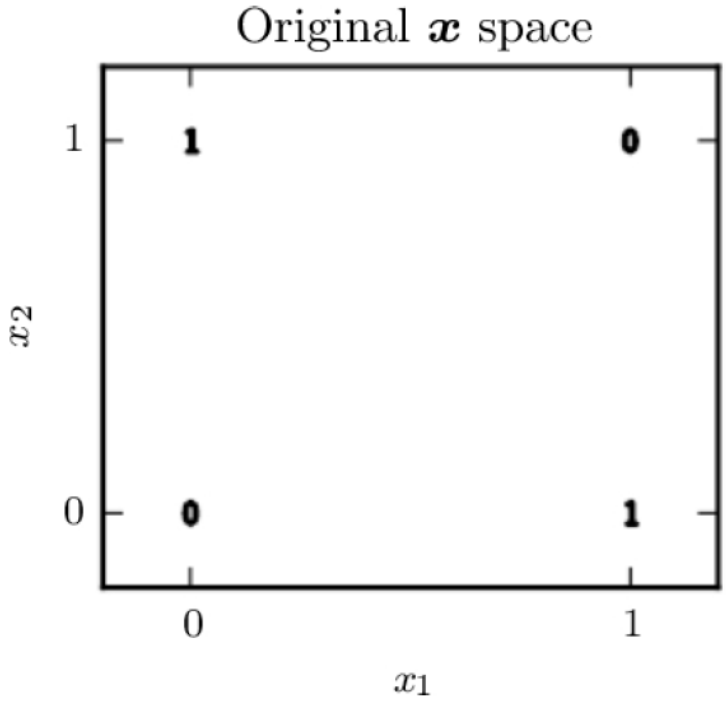
- Our Data:
  $$(X, Y) = \{([0,0]^T, 0), ([0,1]^T, 1), ([1,0]^T, 1), ([1,1]^T, 0)\}$$
- Not concerned with generalization, only want to fit this data
- For simplicity consider the squared loss function

$$J(\theta) = \frac{1}{4} \sum_{x \in X} (f^*(\mathbf{x}) - f(\mathbf{x}; \theta))^2$$

- Need to choose a form for f(x; θ): Consider a linear model with θ being w and b
- Our model $f(\mathbf{x}; \mathbf{w}, b) = \mathbf{x}^T \mathbf{w} + b$

Linear Model

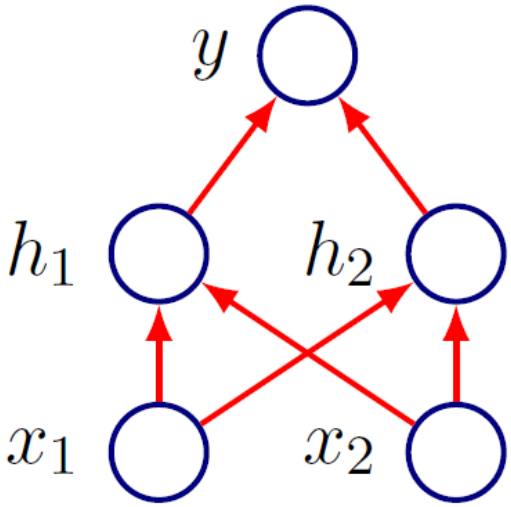
- Recall previous lecture: Normal equations give w = 0 and b = $\frac{1}{2}$
- A linear model is not able to represent XOR, outputs 0.5 everywhere



Solving XOR

- How can we solve the XOR problem?
- Idea: Learn a different feature space in which a linear model will work

Solving XOR



- Define a feedforward network with a vector of hidden units **h** computed by $f^{(1)}(x; W, c)$
- Use hidden unit values as input for a second layer i.e. to compute output $y = f^{(2)}(h; w, b)$
- Complete model: $f(x; W, c, w, b) = f^{(2)}(f^{(1)}(x))$
- What should be $f^{(1)}$? Can it be linear?

Solving XOR

- Let us consider a non-linear activation $g(z) = \max\{0, z\}$
- Our complete network model:

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

- Note: The activation above is applied element-wise

A Solution

- Let

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0$$

- Our design matrix is:

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

A Solution

- Compute the first layer output, by first calculating *XW*

$$XW = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

- Find *XW* + c

$$XW + \mathbf{c} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

- Note: Ignore the type mismatch

A Solution

- Next step: Rectify output

$$\max\{0, XW + \mathbf{c}\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

- Finally compute $\mathbf{w}^T \max\{0, XW + \mathbf{c}\} + b$

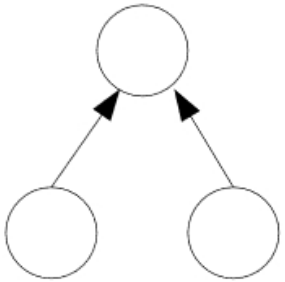
$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

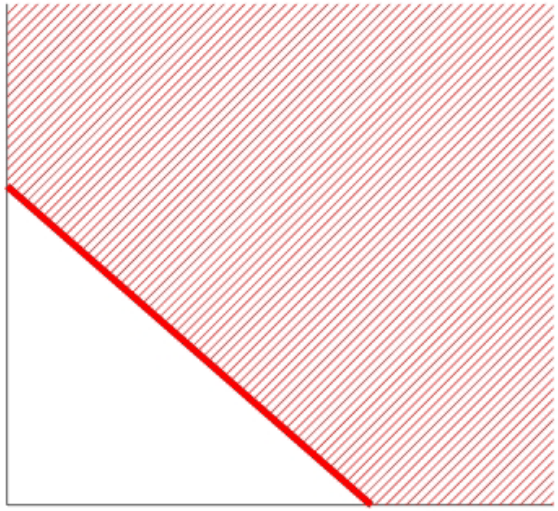
A Solution

- Able to correctly classify every example in the set
- This is a hand coded; demonstrative example, hence clean
- For more complicated functions, we will proceed by using gradient based learning

An Aside:

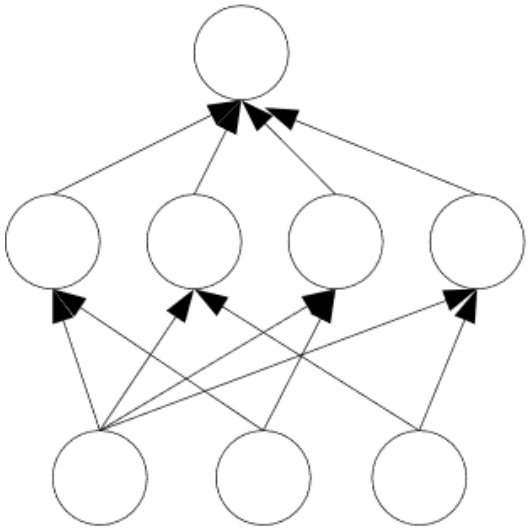
1 layer of trainable weights

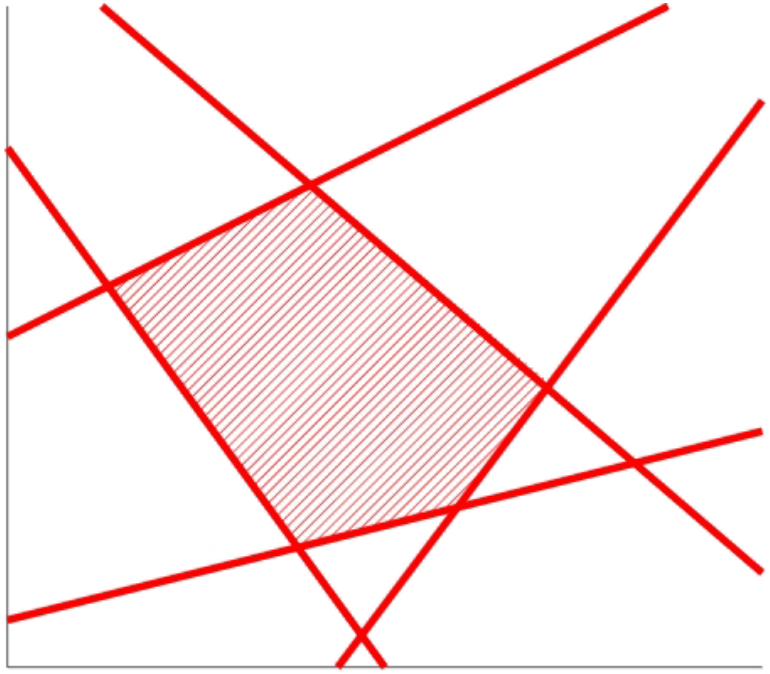




separating hyperplane

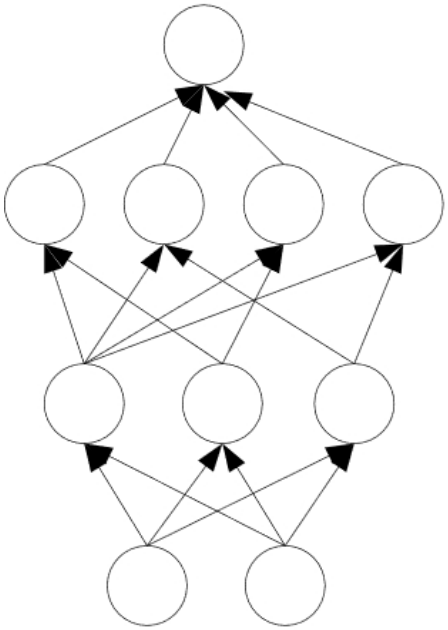
An Aside:

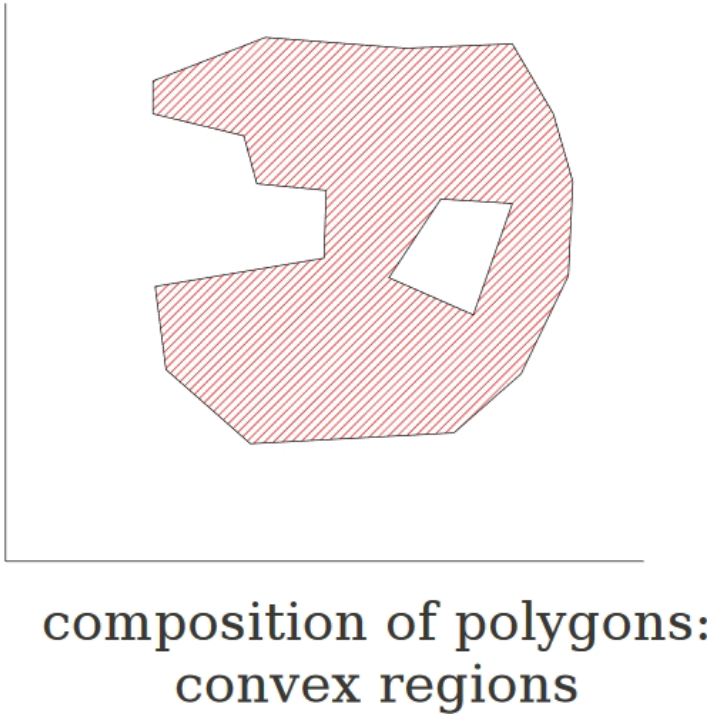




convex polygon region

An Aside:





composition of polygons:
convex regions

An Aside:

- Designing and Training a Neural Network is not much different from training any other Machine Learning model with gradient descent
- Largest difference: Most interesting loss functions become non-convex
- Unlike in convex optimization, no convergence guarantees
- To apply gradient descent: Need to specify cost function, and output representation

Outline

1. Course Review
2. Overview
3. Basic Components
4. Architectural Considerations
5. Back-Propagation

Cost Functions

Cost Functions

- Choice similar to parameteric models from earlier: Define a distribution $p(\mathbf{y}|\mathbf{x};\theta)$ and use principle of maximum likelihood
- We can just use cross entropy between training data and the model's predictions as the cost function:

$$J(\theta) = \mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{data}} \log p_{model}(\mathbf{y}|\mathbf{x})$$

- Specific form changes depending on form of $\log p_{model}$
- Example: If $p_{model}(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}; f(\mathbf{x};\theta), I)$, then we recover:

$$J(\theta) = \frac{1}{2}\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{data}} \|\mathbf{y} - f(\mathbf{x};\theta)\|^2 + \text{Constant}$$

Cost Functions

- Advantage: Need to specify $p(\mathrm{y}|\mathrm{x})$, and automatically get a cost function $\log p(\mathrm{y}|\mathrm{x})$
- Choice of output units is very important for choice of cost function

Output Units

Linear Units

- Given features h, a layer of linear output units gives:

$$\hat{y} = W^T h + b$$

- Often used to produce the mean of a conditional Gaussian distribution:

$$p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}; \hat{\mathbf{y}}, I)$$

- Maximizing log-likelihood $\Longrightarrow$ minimizing squared error

Sigmoid Units

- Task: Predict a binary variable y
- Use a sigmoid unit:

$$\hat{y} = \sigma(\mathbf{w}^T \mathbf{h} + b)$$

- Cost:

$$J(\theta) = -\log p(y|\mathbf{x}) = -\log \sigma((2y-1)(\mathbf{w}^T \mathbf{h} + b))$$

- Positive: Only saturates when model already has right answer i.e. when y = 1 and ($w^T h + b$) is very positive and vice versa
- When ($w^T h + b$) has wrong sign, a good gradient is returned

Softmax Units

- Need to produce a vector $\hat{\mathbf{y}}$ with $\hat{y}_i = p(y = i|\mathbf{x})$
- Linear layer first produces unnormalized log probabilities:

  $\mathbf{z} = W^T\mathbf{h} + \mathbf{b}$
- Softmax:

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

- Log of the softmax (since we wish to maximize $p(y = i; \mathbf{z})$):

$$\log \text{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j)$$

Benefits

$$\log \text{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j)$$

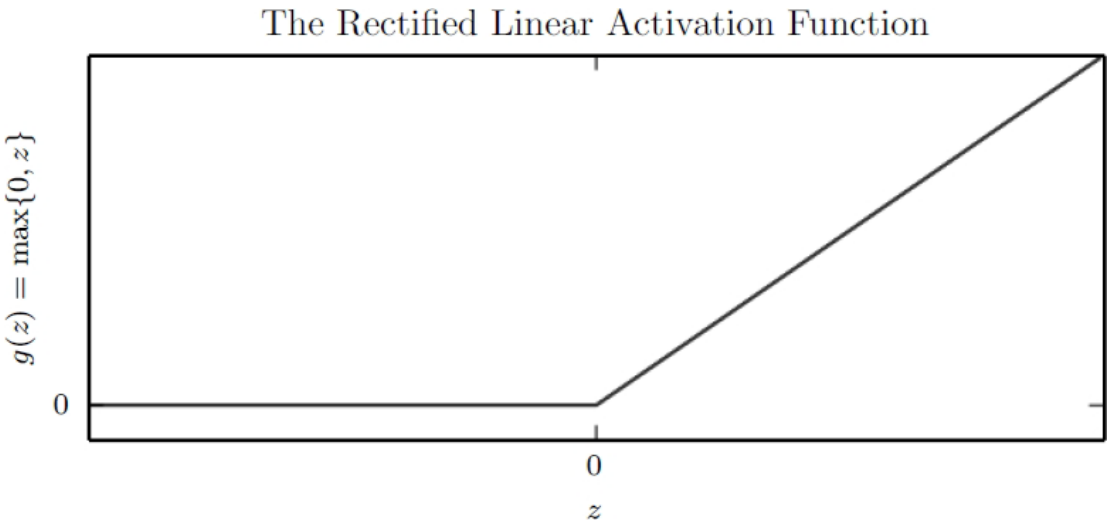
- z_i term never saturates, making learning easier
- Maximizing log-likelihood encourages z_i to be pushed up, while encouraging all z to be pushed down (Softmax encourages competition)
- More intuition: Think of $\log \sum_j \exp(z_j) \approx \max_j z_j$
- log-likelihood cost function ($\sim z_i - \max_j z_j$) strongly penalizes the most active incorrect prediction
- If model already has correct answer then
- $\log \sum_j \exp(z_j) \approx \max_j z_j$ and z_i will roughly cancel out
- Progress of learning is dominated by incorrectly classified examples

Hidden Units

Hidden Units

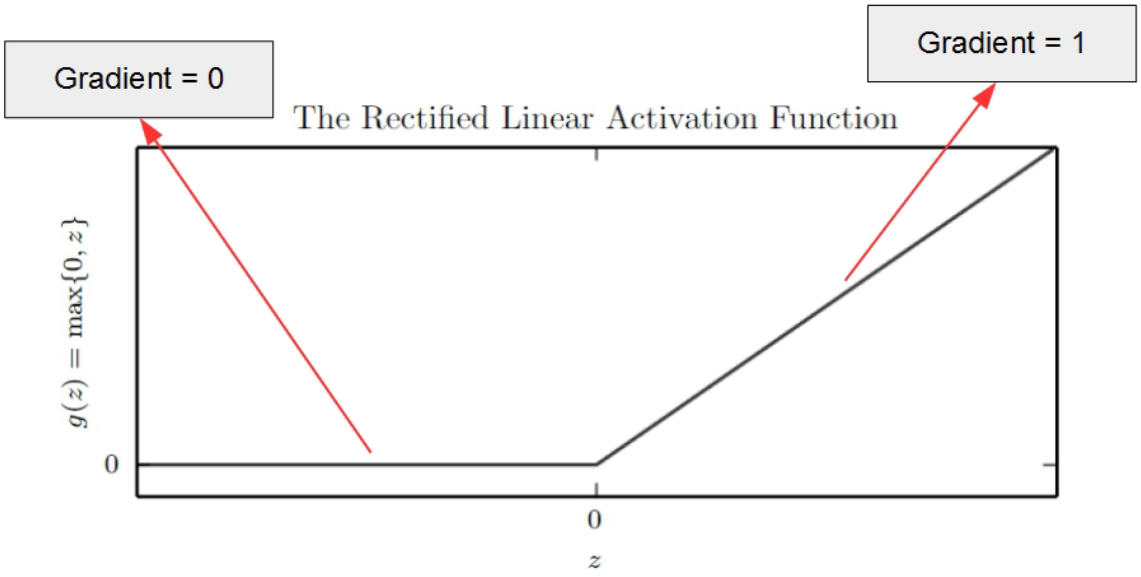
- Accept input x → compute affine transformation z = W^T x + b → apply elementwise non-linear function $g(z)$ → obtain output $g(\mathbf{z})$
- Choices for g?
- Design of Hidden units is an active area of research

Rectified Linear Units



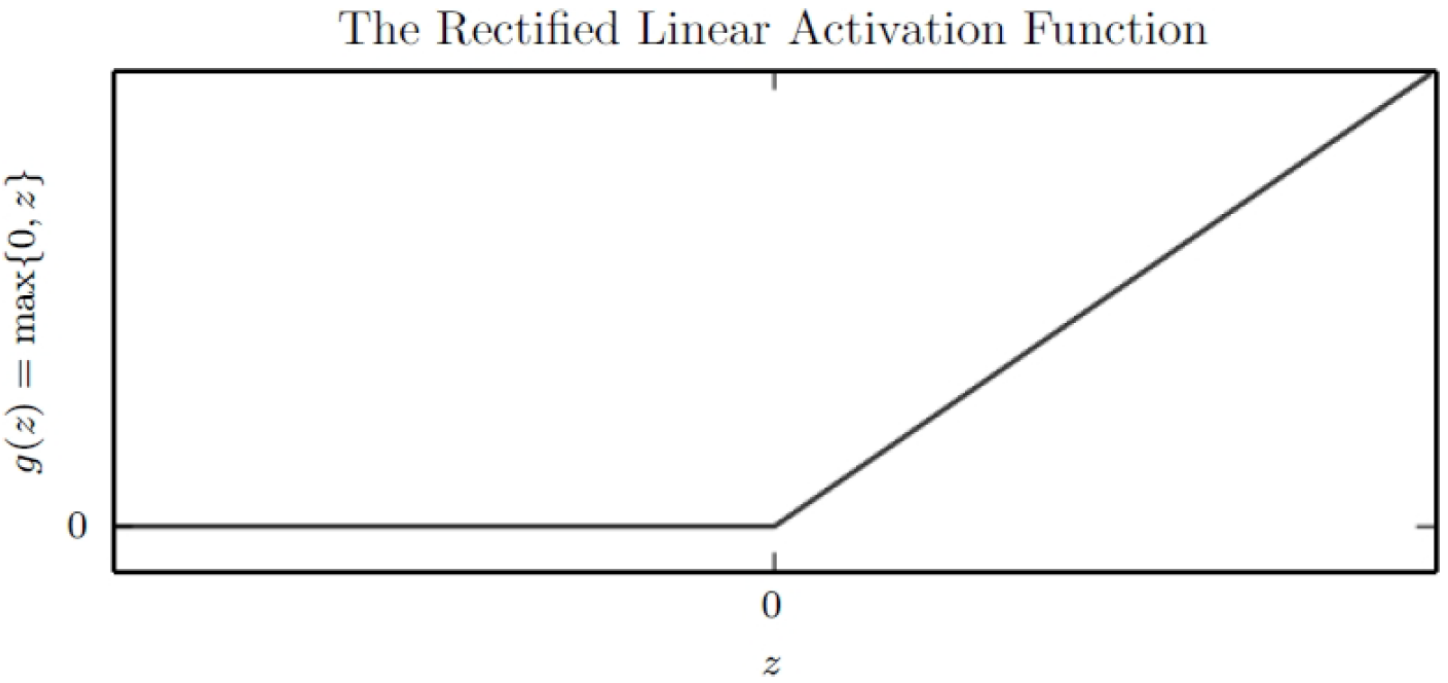
- Activation function: $g(z) = \max\{0, z\}$ with $z \in \mathbb{R}$
- On top of a affine transformation $\max\{0, W\mathbf{x} + \mathbf{b}\}$
- Two layer network: First layer $\max\{0, W_1^T\mathbf{x} + \mathbf{b}_1\}$
- Second layer: $W_2^T \max\{0, W_1^T\mathbf{x} + \mathbf{b}_1\} + \mathbf{b}_2$

Rectified Linear Units



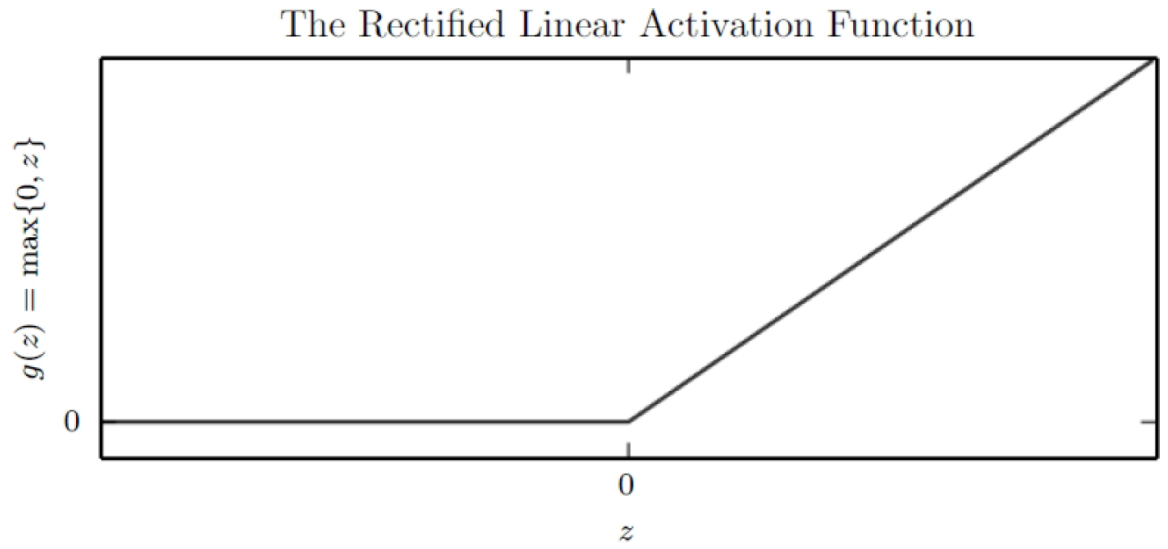
- Similar to linear units. Easy to optimize!
- Give large and consistent gradients when active
- **Good practice**: Initialize b to a small positive value (e.g. 0.1)
- Ensures units are initially active for most inputs and derivatives can pass through

Rectified Linear Units



- Not everywhere differentiable. Is this a problem?
  - In practice not a problem. Return one sided derivatives at z = 0
  - Gradient based optimization is subject to numerical error anyway

Rectified Linear Units



- Positives:
  - Gives large and consistent gradients (does not saturate) when active
  - Efficient to optimize, converges much faster than sigmoid or tanh
- Negatives:
  - Non zero centered output
  - Units "die" i.e. when inactive they will never update

Generalized Rectified Linear Units

- Get a non-zero slope when $z_i < 0$
- $g(z, a)_i = \max\{0, z_i\} + a_i \min\{0, z_i\}$
  - **Absolute value rectification**: (*Jarret et al, 2009*) $a_i = 1$ gives $g(z) = |z|$
  - **Leaky ReLU**: (*Maas et al., 2013*) Fix a_i to a small value e.g. 0.01
  - **Parametric ReLU**: (*He et al., 2015*) Learn a_i
  - **Randomized ReLU**: (*Xu et al., 2015*) Sample a_i from a fixed range during training, fix during testing
  - ….

Generalized Rectified Linear Units

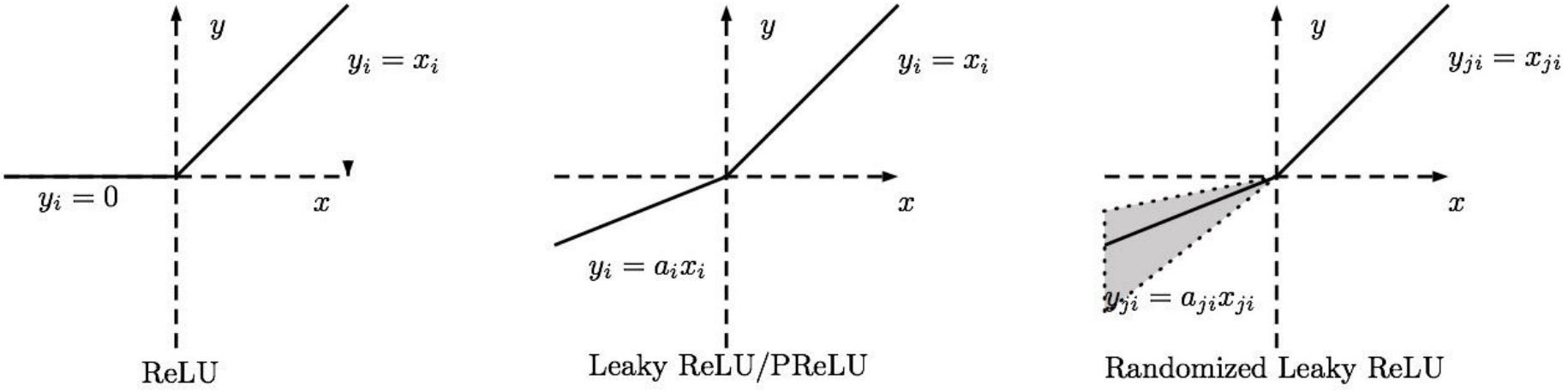


Figure: Xu et al. "Empirical Evaluation of Rectified Activations in Convolutional Network"

Exponential Linear Units (ELUs)

$$g(z) = \begin{cases} z \text{ if } z > 0 \\ \alpha(\exp z - 1) \text{ if } z \leq 0 \end{cases}$$

- All the benefits of ReLU + does not get killed
- Problem: Need to exponentiate

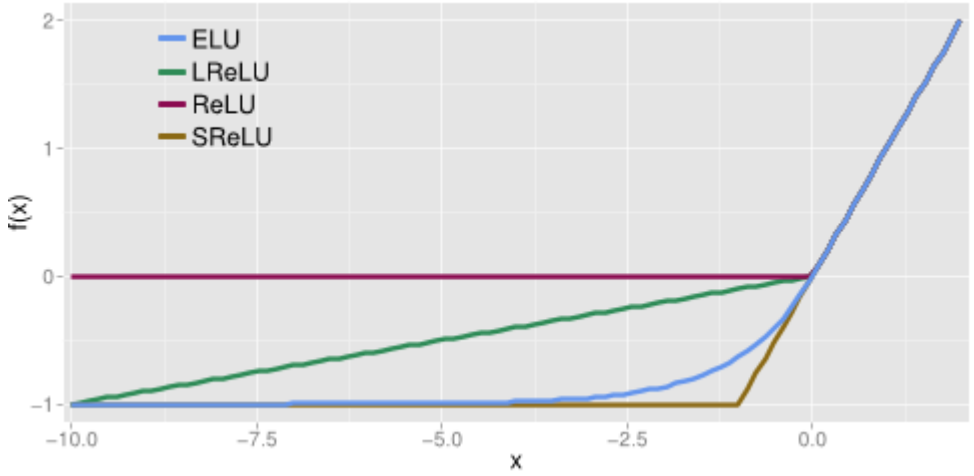


Figure: Clevert et al. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)", 2016

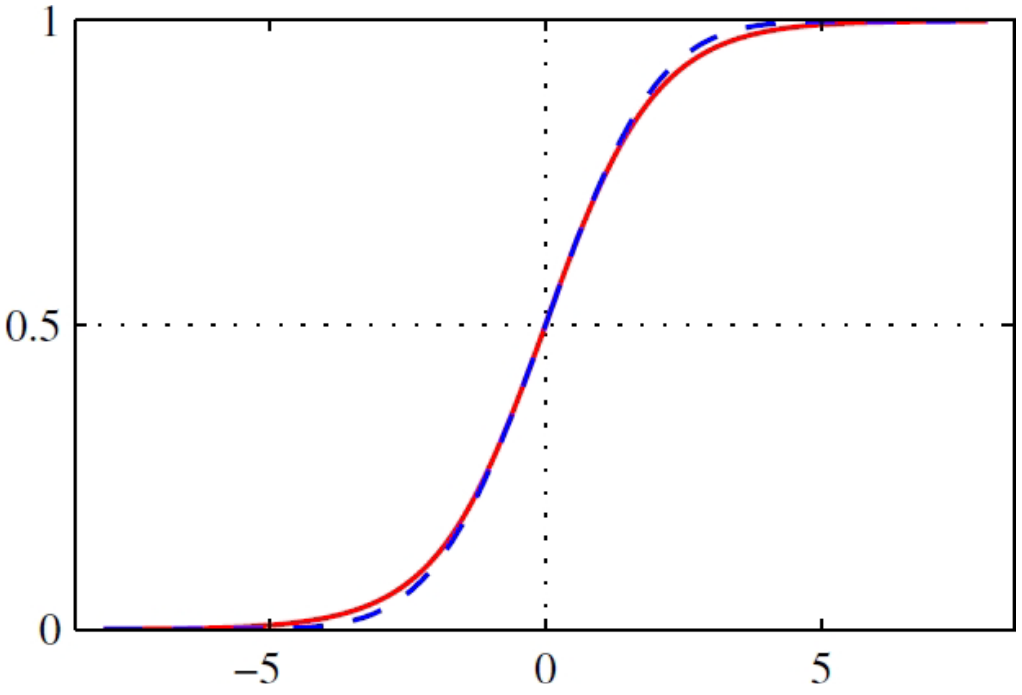
Maxout Units

- Generalizes ReLUs further but does not fit into the (dot product → nonlinearity) mold
- Instead of applying an element-wise function g(z), divide vector z into k groups (more parameters!)
- Output maximum element of one of k groups $g(z)_i = max_{j \in G(i)} z_j$
- $g(z)_i = \max\{w_1^T x + b_1, \dots, w_k^T x + b_k\}$
- A maxout unit makes a piecewise linear approximation (with k pieces) to an arbitrary convex function
- Can be thought of as learning the activation function itself
- With $k = 2$ we CAN recover absolute value rectification, or ReLU or PReLU
- Each unit parameterized by k weight vectors instead of 1, needs stronger regularization

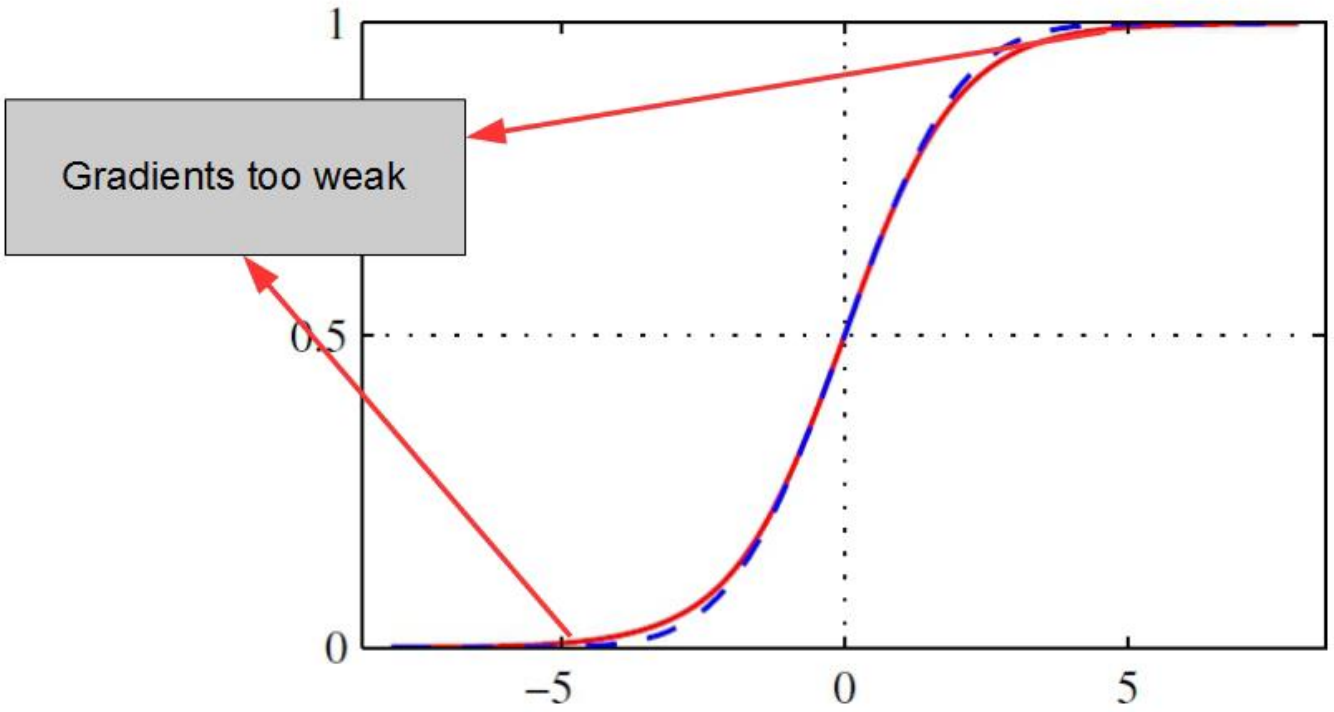
Sigmoid Units

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

- Squashing type non-linearity: pushes outputs to range [0, 1]



Sigmoid Units



- Problem: Saturate across most of their domain, strongly sensitive only when z is closer to zero
- Saturation makes gradient based learning difficult

Tanh Units

- Related to sigmoid: $g(z) = \tanh(z) = 2\sigma(2z) - 1$
- Positives: Squashes output to range [-1, 1], outputs are zero-centered
- Negative: Also saturates
- Still better than sigmoid as $\hat{y} = w^T \tanh(U^T \tanh(V^T x))$ resembles $\hat{y} = w^T U^T V^T x$ when activations are small

Other Units

- **Radial Basis Functions**: $g(z)_i = \exp(\frac{1}{\sigma_i^2}\left\|W_{:,i}x\right\|^2)$
- Function is more active as x approaches a template $W_{:,i}$. Also saturates and is hard to train
- **Softplus**: $g(z) = \log(1 + e^z)$. Smooth version of rectifier (*Dugas et al., 2001*), although differentiable everywhere, empirically performs worse than rectifiers
- **Hard Tanh**: $g(z) = \max(-1, \min(1, z))$, like the rectifier, but bounded (*Collobert, 2004*)

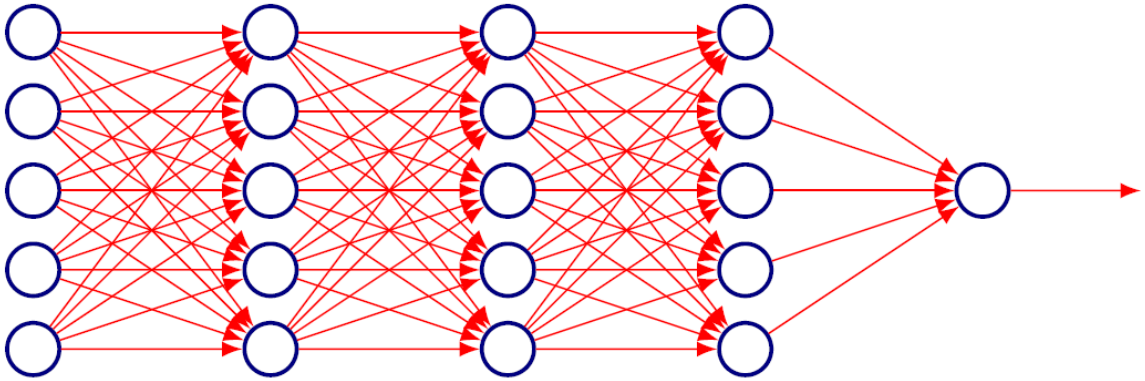
Summary

- In Feedforward Networks don't use Sigmoid
- When a sigmoidal function must be used, use tanh
- Use ReLU by default, but be careful with learning rates
- Try other generalized ReLUs and Maxout for possible improvement

Outline

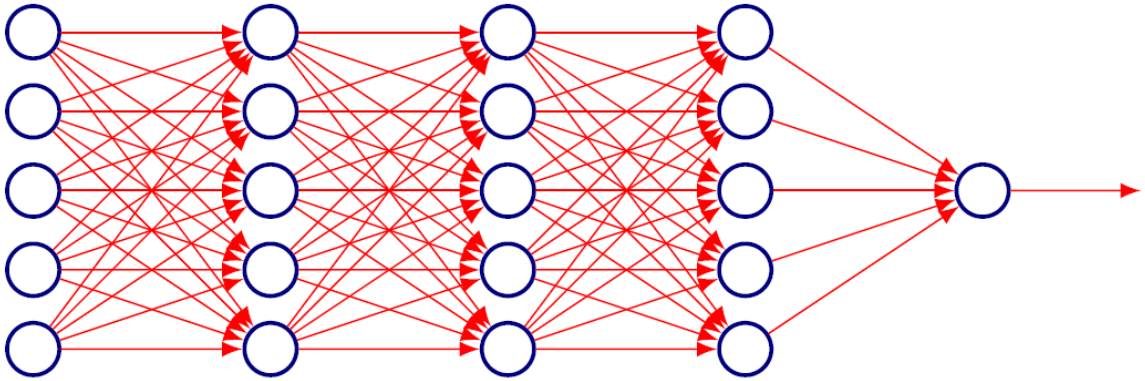
1. Course Review
2. Overview
3. Basic Components
4. Architectural Considerations
5. Back-Propagation

Architecture Design



- First layer: $\mathrm{h}^{(1)} = g^{(1)}(W^{(1)^T} x + b^{(1)})$
- Second layer: $\mathrm{h}^{(2)} = g^{(2)}(W^{(2)^T} x + b^{(2)})$
- How do we decide depth, width?
- In theory how many layers suffice?

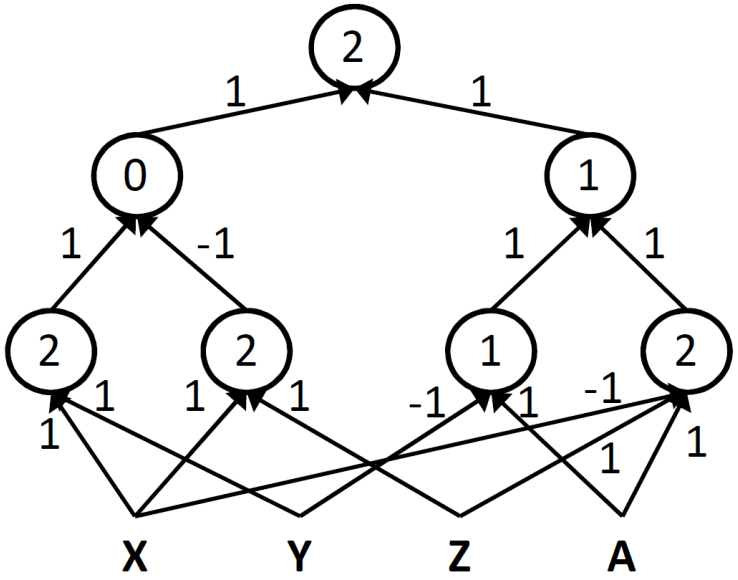
Depth and Width

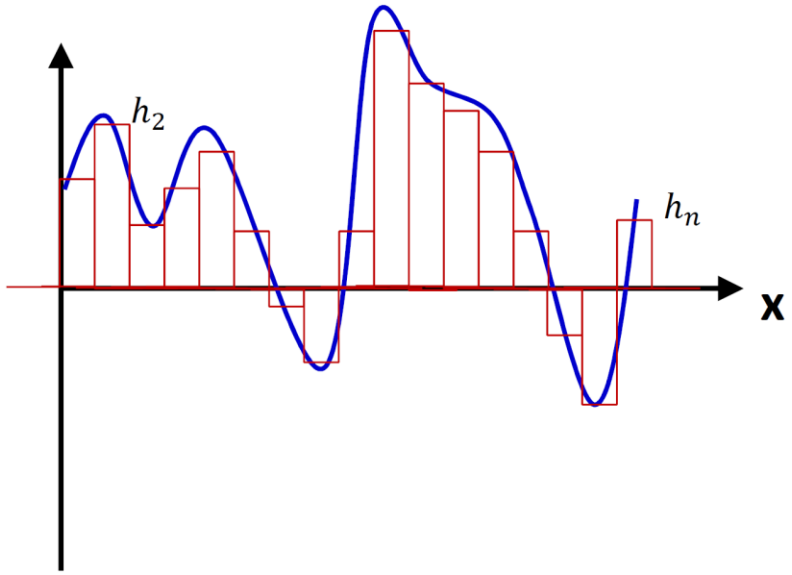


- The length of the chain of functions in a neural network is called its depth
- The dimensionality of the hidden layers of a neural network is called its width

MLPs approximate functions

$$((A\&\bar{X}\&Z)|(A\&\bar{Y}))\&((X\ \&\ Y)|\overline{(X\&Z)})$$

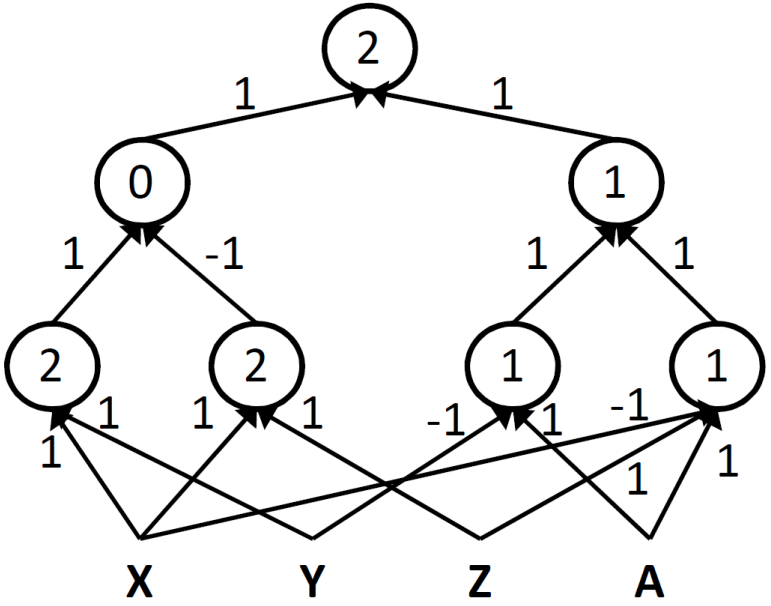


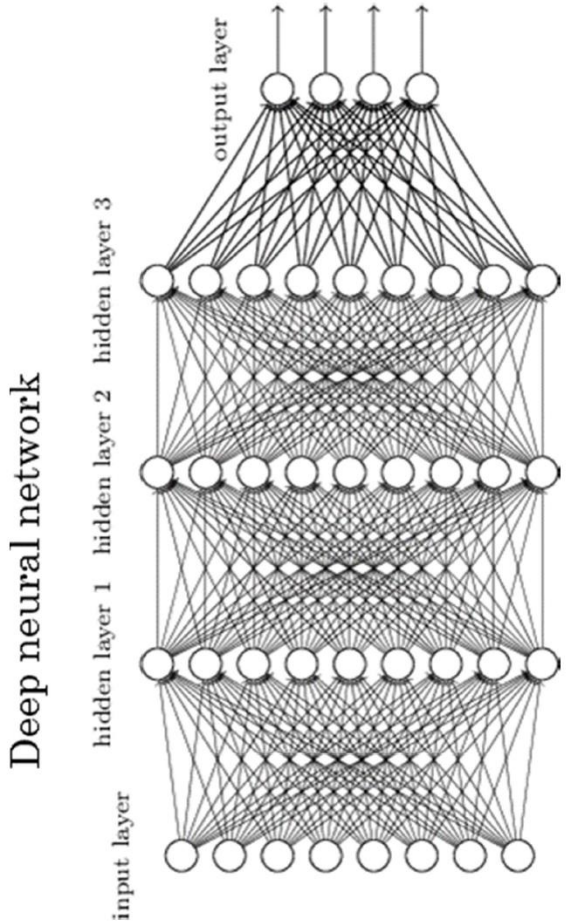


- MLPs can compose universal boolean functions
- MLPs can compose universal classifiers
- MLPs can compose universal approximators

MLP as Boolean Functions

$$((A\&\bar{X}\&Z)|(A\&\bar{Y}))\&((X\ \&\ Y)|\overline{(X\&Z)})$$





- MLPs are universal Boolean functions
  - Any function over any number of inputs and any number of outputs
- But how many “layers” will they need?

How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows all input combinations for which output is 1

- A Boolean function is just a truth table

How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows all input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$

- Expressed in disjunctive normal form

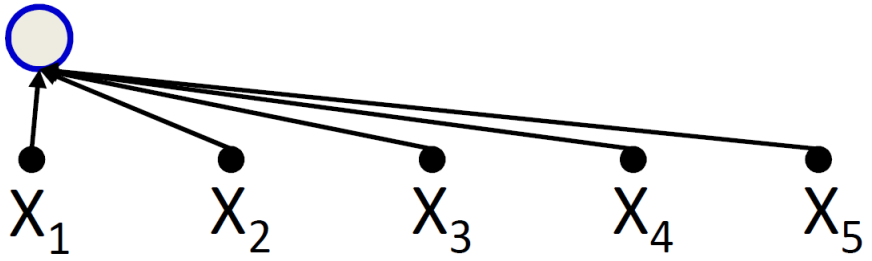
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows all input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

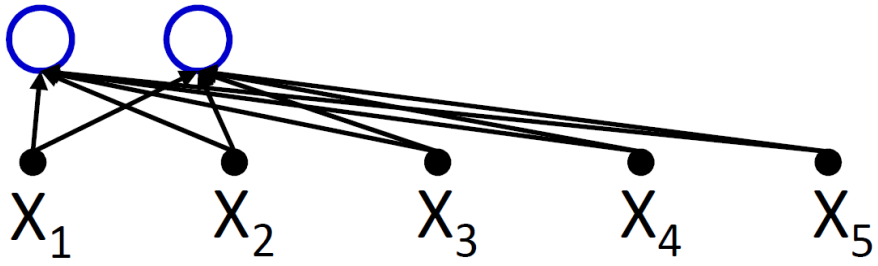
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows all input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

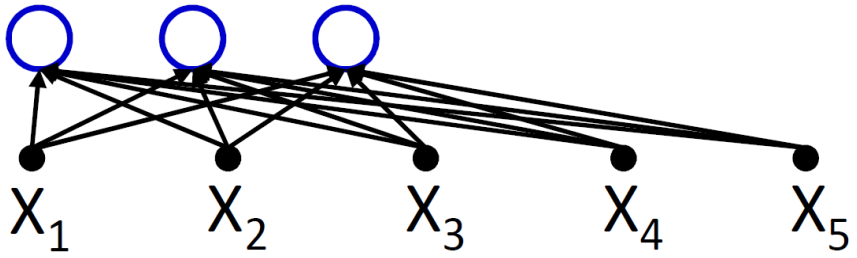
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows all input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2X_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

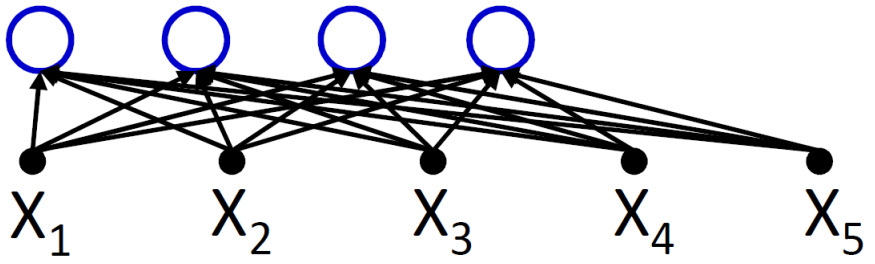
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows all input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

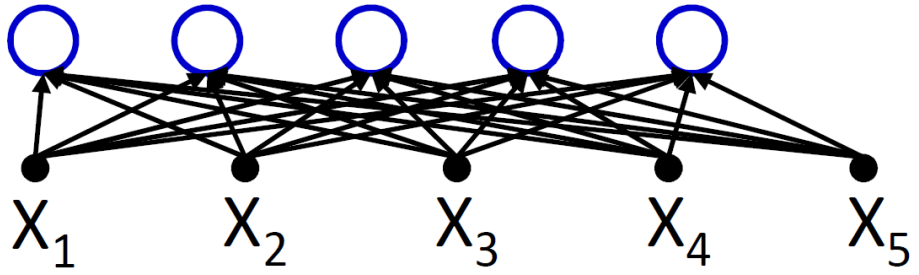
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows all input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

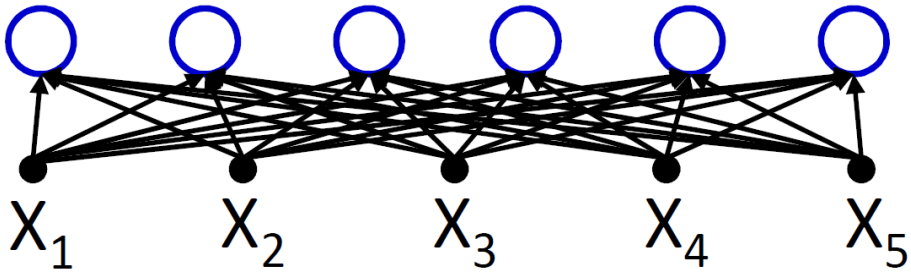
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows all input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

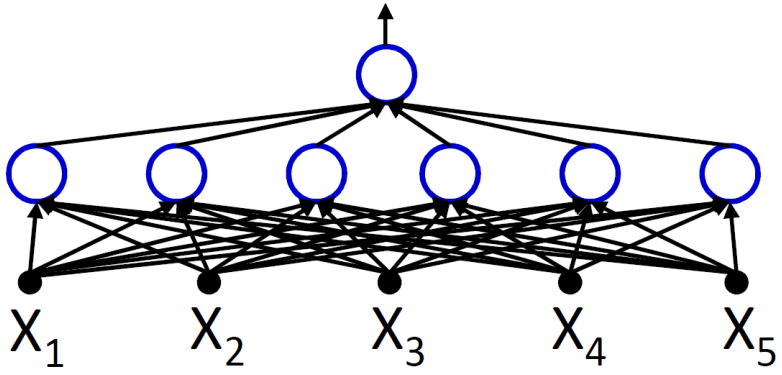
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows all input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

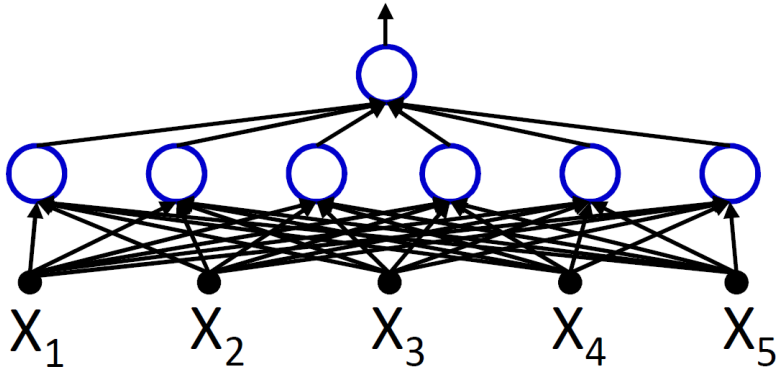
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

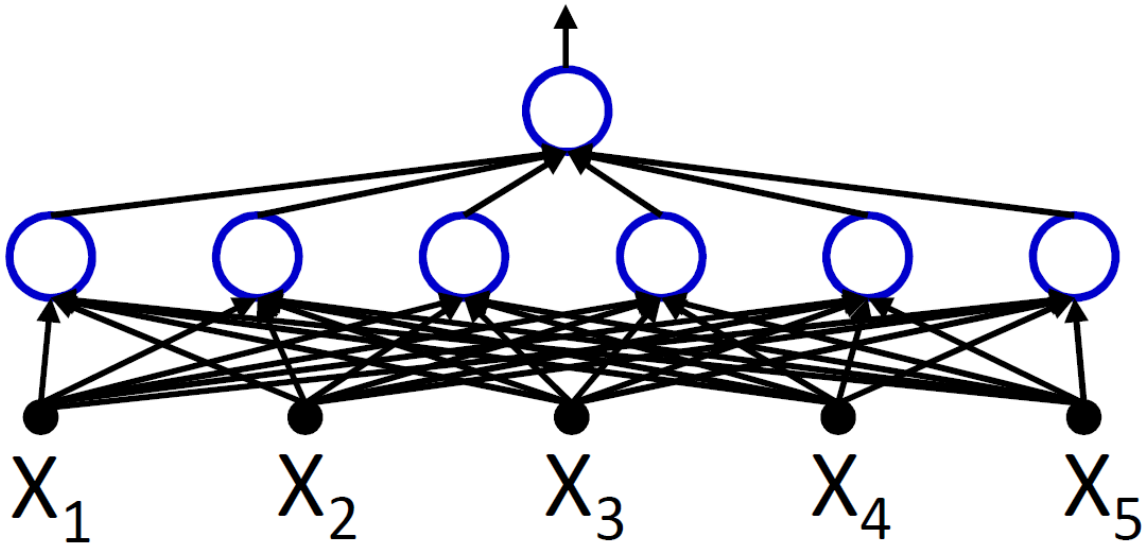
Truth table shows all input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Any truth table can be expressed in this manner!
- A one-hidden-layer MLP is a Universal Boolean Function

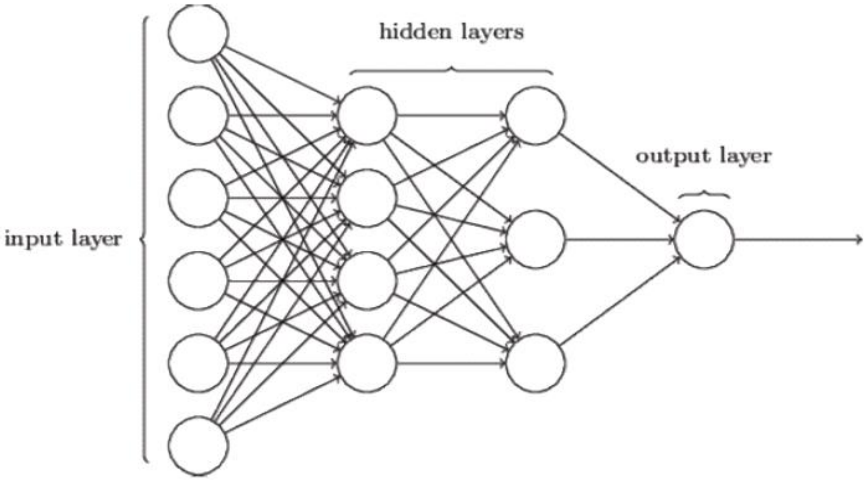
Number of parameters in a network

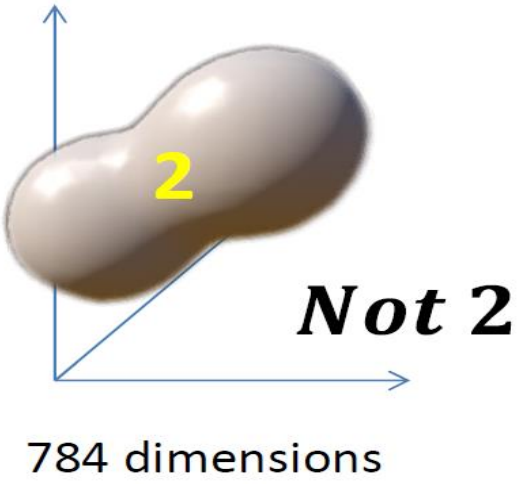


- The actual number of parameters in a network is the number of connections
  - In this example there are 30
- This is the number that really matters in software or hardware implementations

MLP as universal classifier

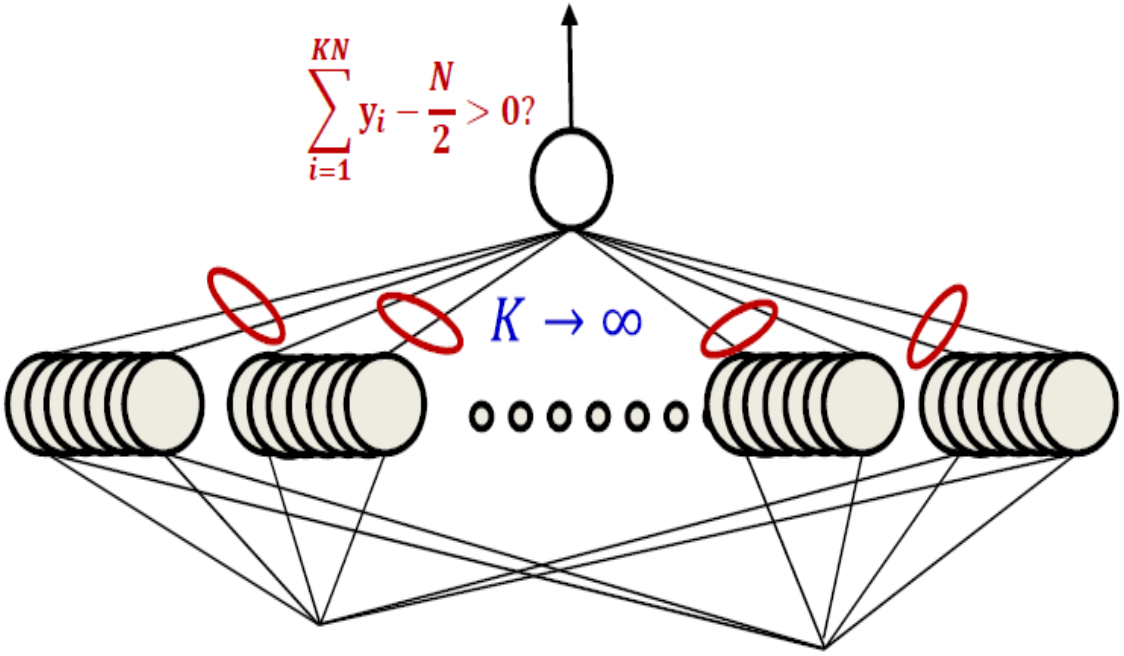
784 dimensions
(MNIST)

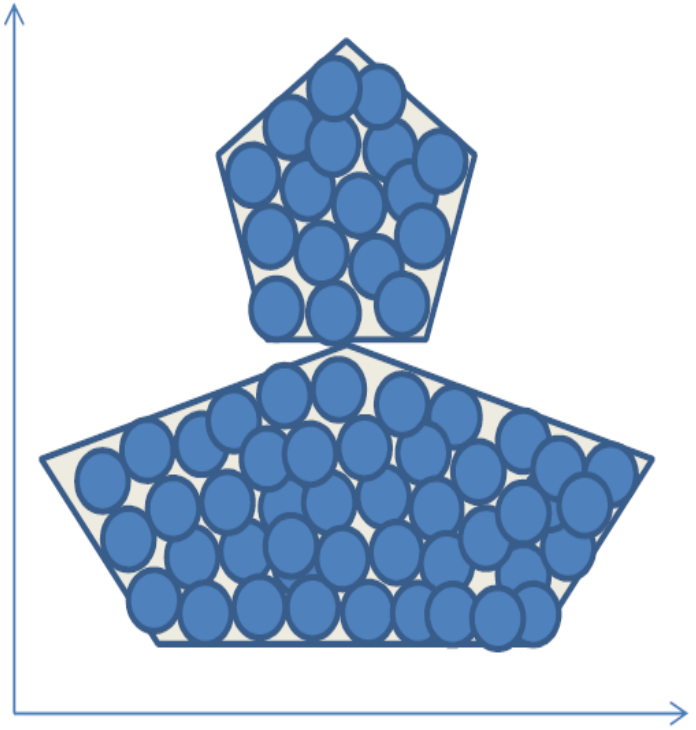




- MLP as a function over real inputs
- MLP as a function that finds a complex “decision boundary” over a space of reals

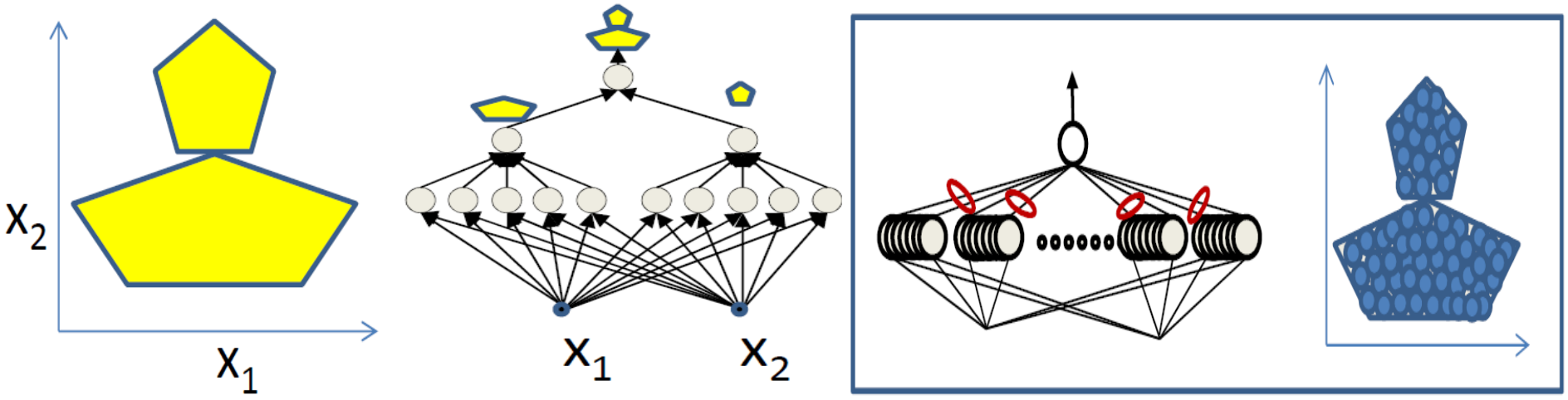
MLP as universal classifier





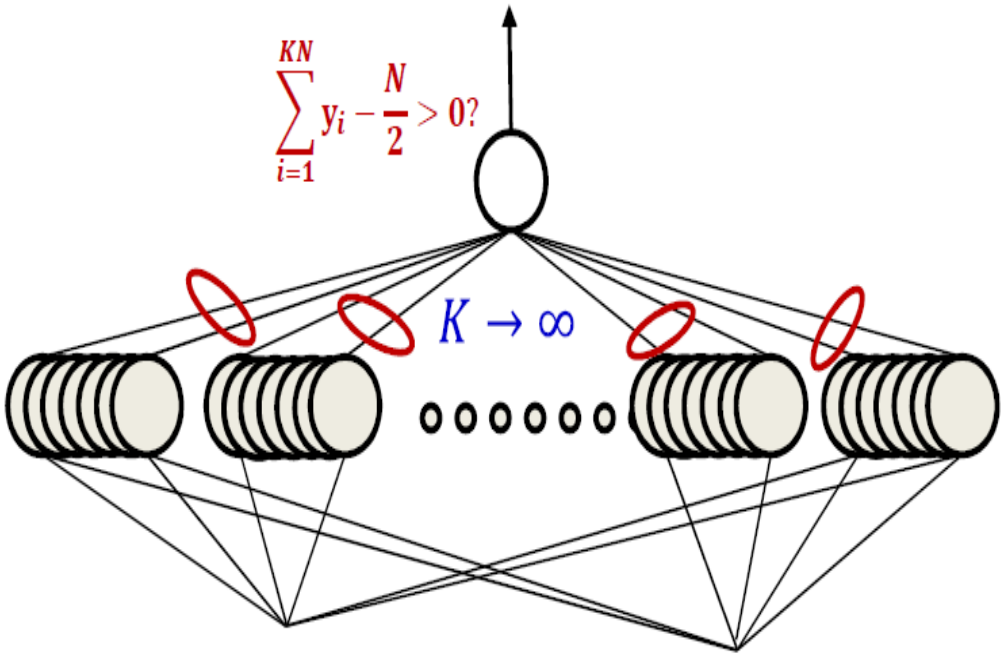
- MLPs can capture any classification boundary
- A one-layer MLP can model any classification boundary
- MLPs are universal classifiers

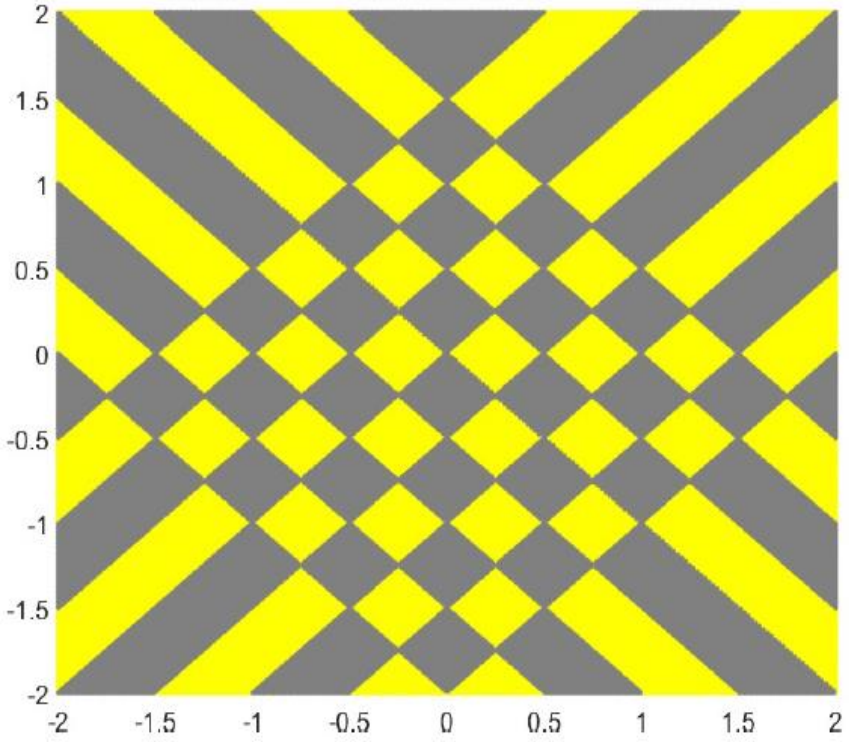
Depth and the universal classifier



- Deeper networks can require far fewer neurons

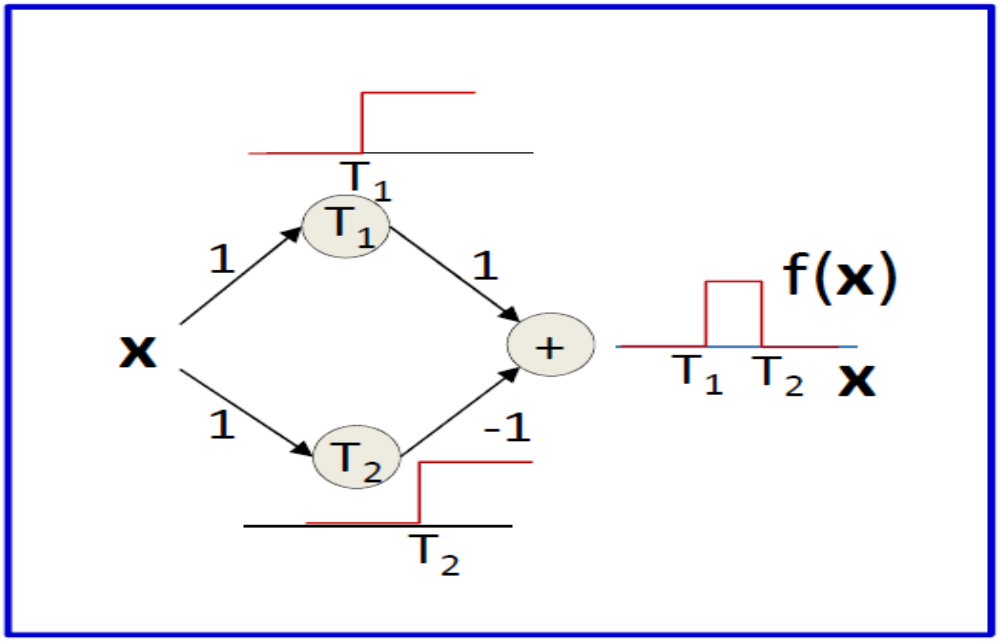
Optimal depth





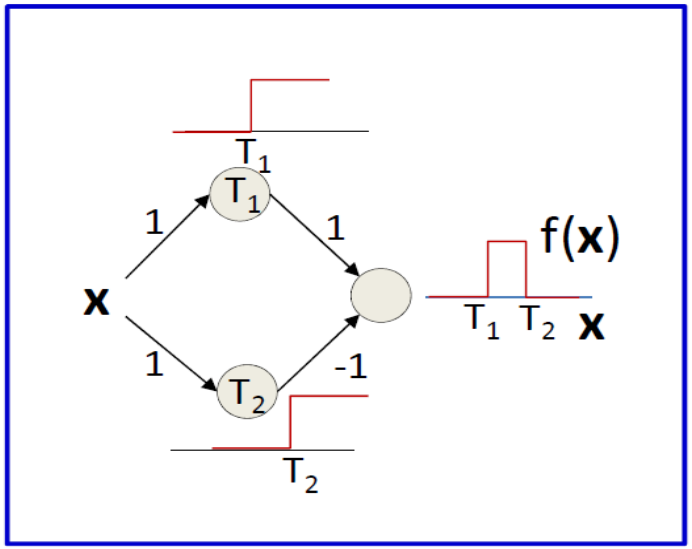
- A one-hidden-layer neural network will required infinite hidden neurons

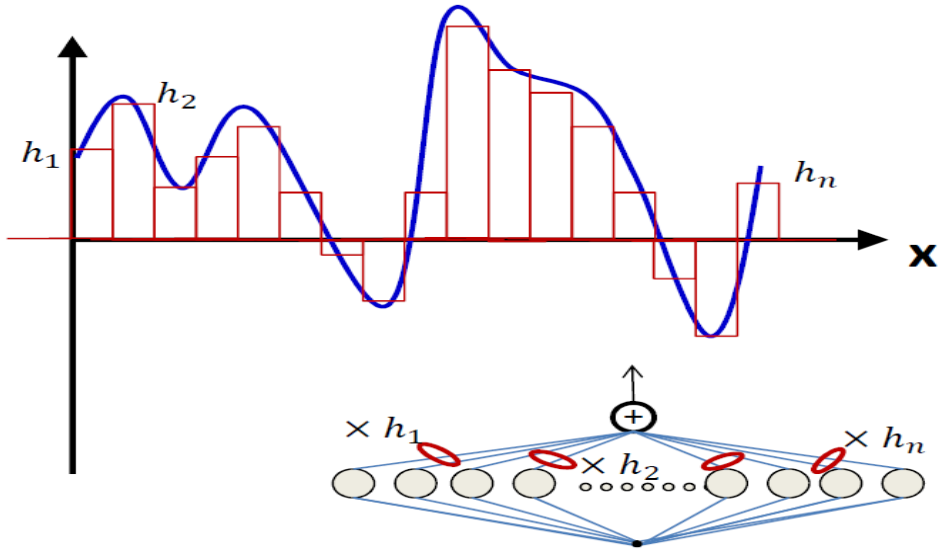
MLP as a continuous-valued regression



- A simple 3-unit MLP with a "summing" output unit can generate a "square pulse" over an input
  - Output is 1 only if the input lies between T_1 and T_2
  - T_1 and T_2 can be arbitrarily specified

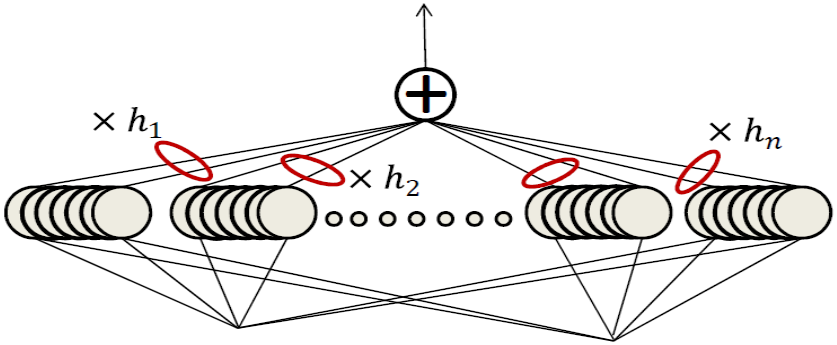
MLP as a continuous-valued regression

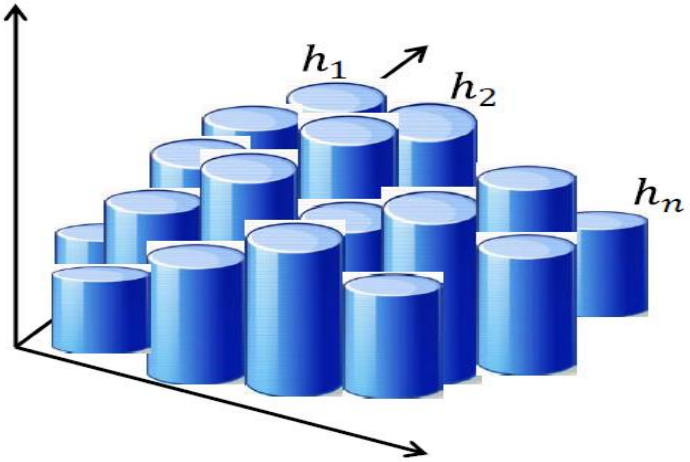




- A simple 3-unit MLP can generate a “square pulse” over an input
- An MLP with many units can model an arbitrary function over an input
  - — To arbitrary precision
    - Simply make the individual pulses narrower
- A one-layer MLP can model an arbitrary function of a single input

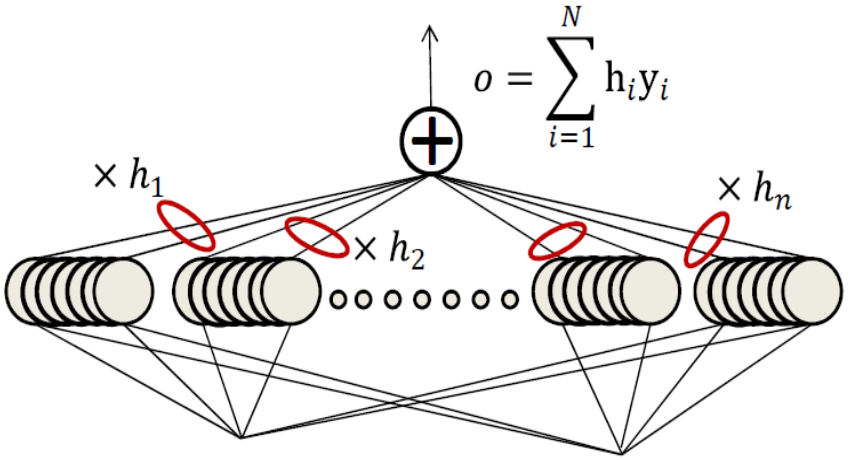
MLP as a continuous-valued function

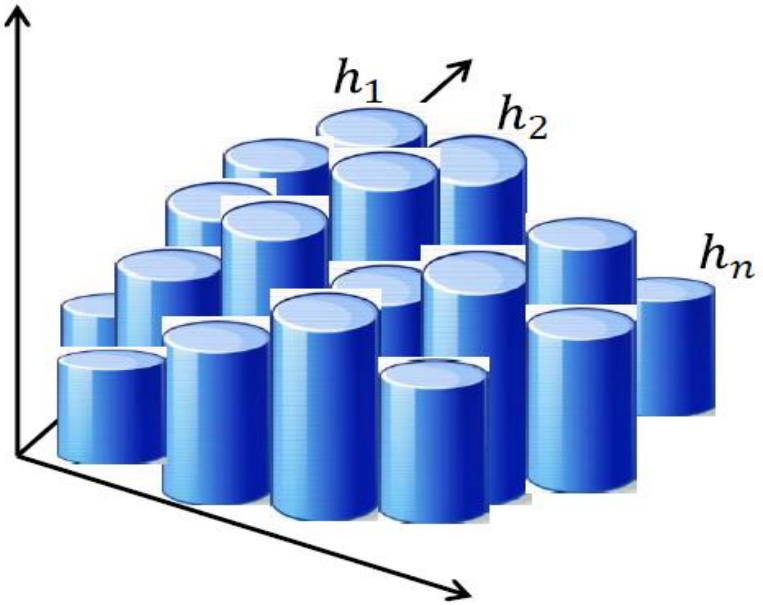




- MLPs can actually compose arbitrary functions in any number of dimensions!
  - — Even with only one layer
    - As sums of scaled and shifted cylinders
  - — To arbitrary precision
    - By making the cylinders thinner
  - — The MLP is a universal approximator!

MLP as universal approximator





- MLPs can actually compose arbitrary functions
- But explanation so far only holds if the output unit only performs summation
  — i.e. does not have an additional “activation”

Universality

- Theoretical result [Cybenko, 1989]: 2-layer net with linear output with some squashing non-linearity in hidden units can approximate any continuous function over compact domain to arbitrary accuracy (given enough hidden units!)
- Implication: Regardless of function we are trying to learn, we know a large MLP can represent this function
- But not guaranteed that our training algorithm will be able to learn that function
- Gives no guidance on how large the network will be (exponential size in worst case)
- Talked of some suggestive results earlier:

Advantages of Depth

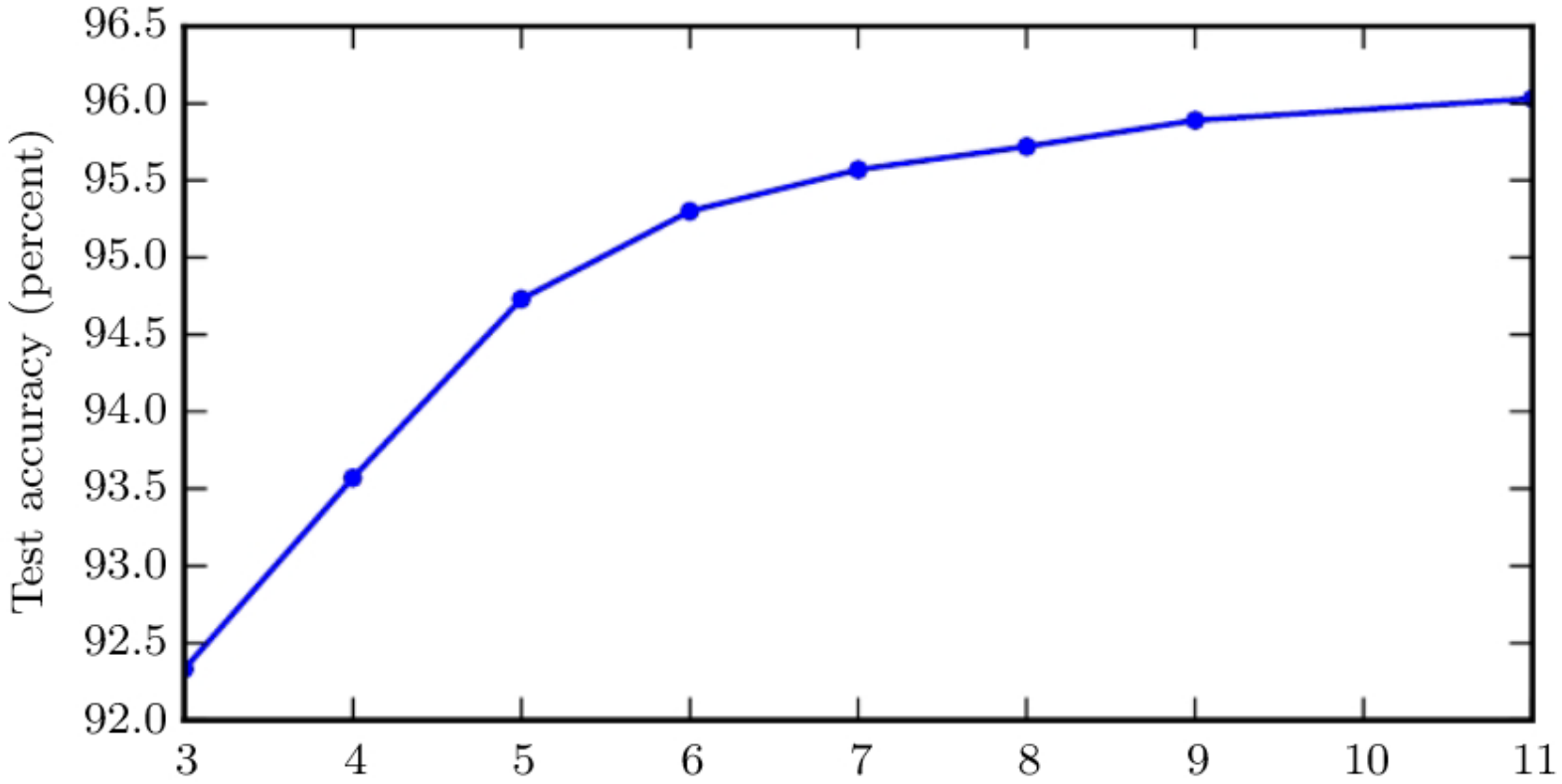
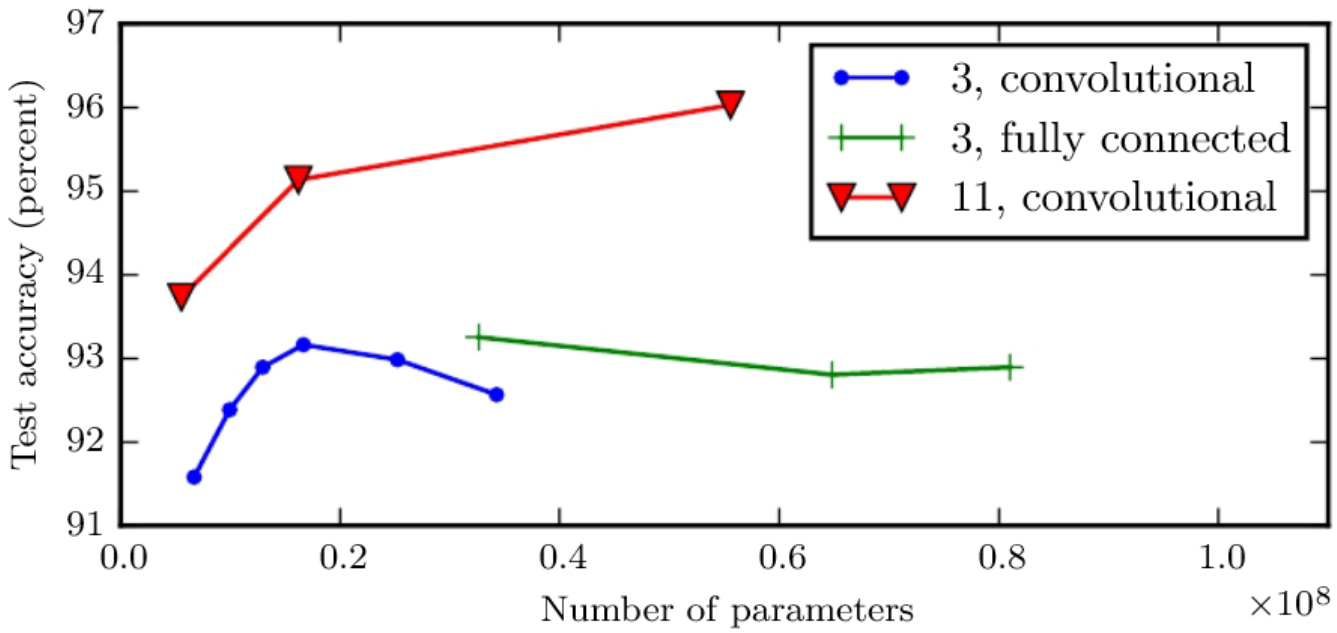


Figure: Goodfellow et al., 2014

Advantages of Depth



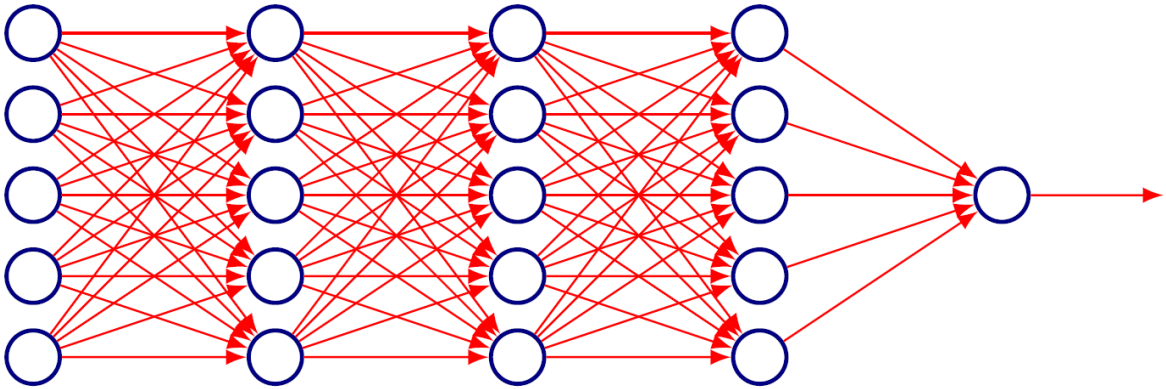
- Control experiments show that other increases to model size don't yield the same effect

Figure: Goodfellow et al., 2014

Outline

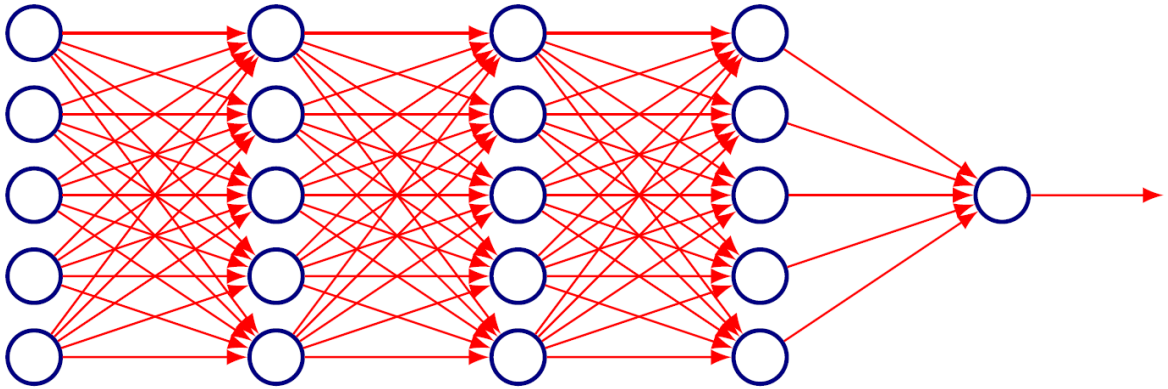
1. Course Review
2. Overview
3. Basic Components
4. Architectural Considerations
5. Back-Propagation

How do we learn weights?



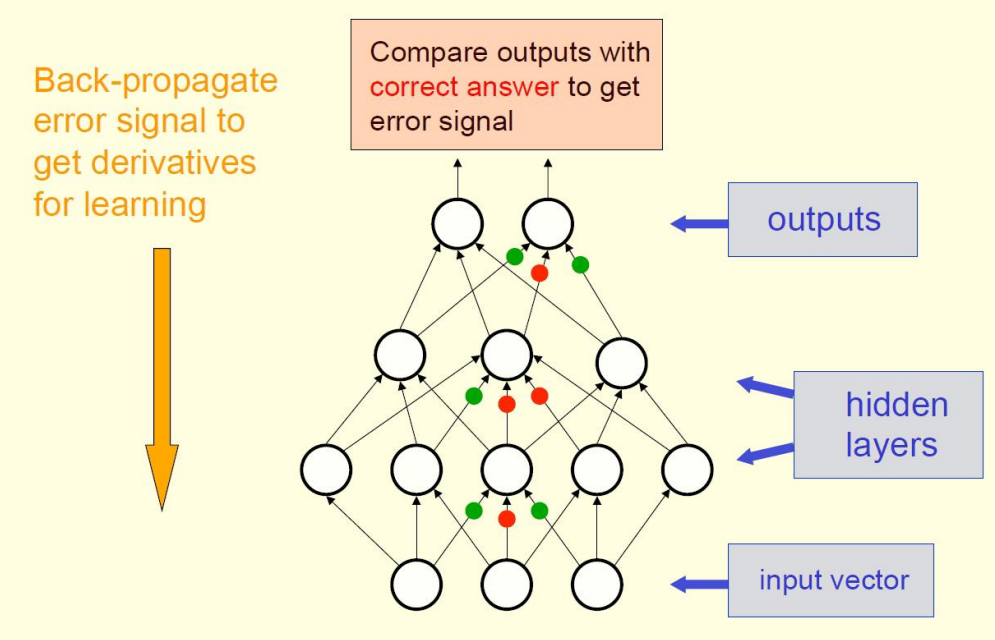
- First Idea: Randomly perturb one weight, see if it improves performance, save the change
- **Very inefficient**: Need to do many passes over a sample set for just one weight change
- What does this remind you of?

How do we learn weights?



- Another Idea: Perturb all the weights in parallel, and correlate the performance gain with weight changes
- Very hard to implement
- Yet another idea: Only perturb activations (since they are fewer). Still very inefficient.

Backpropagation



- **Feedforward Propagation**: Accept input x, pass through intermediate stages and obtain output $\hat{y}$
- **During Training**: Use $\hat{y}$ to compute a scalar cost $J(\theta)$
- Backpropagation allows information to flow backwards from cost to compute the gradient

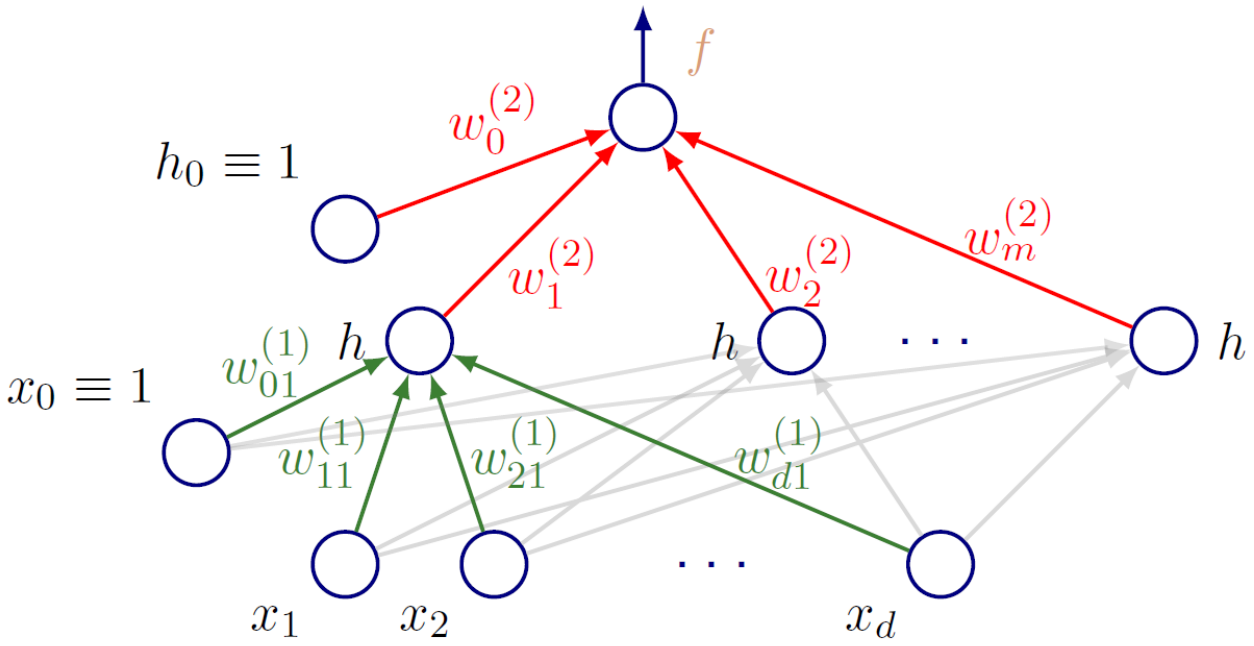
Figure: G. E. Hinton

Backpropagation

- From the training data we don't know what the hidden units should do
- But, we can compute how fast the error changes as we change a hidden activity
- Use error derivatives w.r.t hidden activities
- Each hidden unit can affect many output units and have separate effects on error — combine these effects
- Can compute error derivatives for hidden units efficiently (and once we have error derivatives for hidden activities, easy to get error derivatives for weights going in)

Slide: G. E. Hinton

Review: neural networks



- Feedforward operation, from input x to output $\hat{y}$:

$$\hat{y}(\mathbf{x};\mathbf{w}) = f\left(\sum_{j=1}^{m} w_j^{(2)} h\left(\sum_{i=1}^{d} w_{ij}^{(1)} x_i + w_{0j}^{(1)}\right) + w_0^{(2)}\right)$$

Slide adapted from TTIC 31020, Gregory Shakhnarovich

Training the network

- Error of the network on a training set:

$$L(X; \mathbf{w}) = \sum_{i=1}^{N} \frac{1}{2} \left(y_i - \hat{y}(\mathbf{x}_i; \mathbf{w})\right)^2$$

- Generally, no closed-form solution; resort to gradient descent
- Need to evaluate derivative of L on a single example
- Let's start with a simple linear model $\hat{y} = \sum_j w_j x_{ij}$:

$$\frac{\partial L(\mathbf{x}_i)}{\partial w_j} = \underbrace{(\hat{y}_i - y_i)}_{\text{error}} x_{ij}.$$

Slide adapted from TTIC 31020, Gregory Shakhnarovich

Initializing weights

- Randomly initialize weights in the linear region. But they should be large enough to make learning proceed.
  - The network learns the linear part of the mapping before the more difficult nonlinear parts
  - If weights are too small, gradients are small, which makes learning slow
- To obtain a standard deviation close to 1 at the output of the first hidden layer, we just need to use the recommended sigmoid and require that the input to the sigmoid also have a standard deviation $\sigma_y = 1$. Assuming the inputs to a unit are uncorrelated with variance 1, the standard deviation of σ_{y_i} is

$$\sigma_{y_i} = \left(\sum_{j=1}^{m} w_{ij}^2\right)^{1/2}$$

Initializing weights

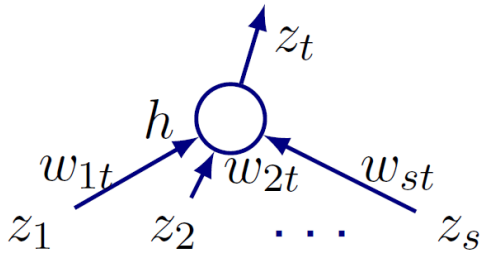
- To ensure $\sigma_{y_i} = 1$, weights should be randomly drawn from a distribution (e.g. uniform) with mean zero and standard deviation

$$\sigma_w = m^{-1/2}$$

Backpropagation

- General unit activation in a multilayer network:

$$z_t = h\left(\sum_j w_{jt} z_j\right)$$



- Forward propagation: calculate for each unit $a_t = \sum_j w_{jt} z_j$
- The loss L depends on w_{jt} only through a_t:

$$\frac{\partial L}{\partial w_{jt}} = \frac{\partial L}{\partial a_t}\frac{\partial a_t}{\partial w_{jt}} = \frac{\partial L}{\partial a_t} z_j$$

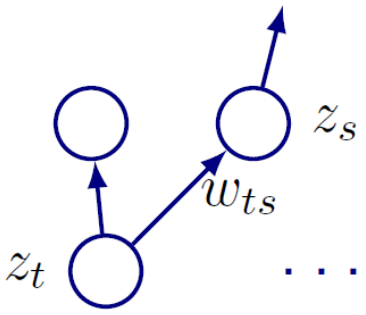
Slide adapted from TTIC 31020, Gregory Shakhnarovich

Backpropagation

$$\frac{\partial L}{\partial w_{jt}} = \frac{\partial L}{\partial a_t} z_j \qquad \frac{\partial L}{\partial w_{jt}} = \underbrace{\frac{\partial L}{\partial a_t}}_{\delta_t} z_j$$

- Output unit with linear activation: $\sigma_t = \hat{y} - y$
- Hidden unit $z_t = h(a_t)$ which sends inputs to units S:

$$\begin{aligned} \delta_t &= \sum_{s \in S} \frac{\partial L}{\partial a_s} \frac{\partial a_s}{\partial a_t} \\ &= h'(a_t) \sum_{s \in S} w_{ts} \delta_s \end{aligned}$$



$$a_s = \sum_{j:j \to s} w_{js} h(a_j)$$

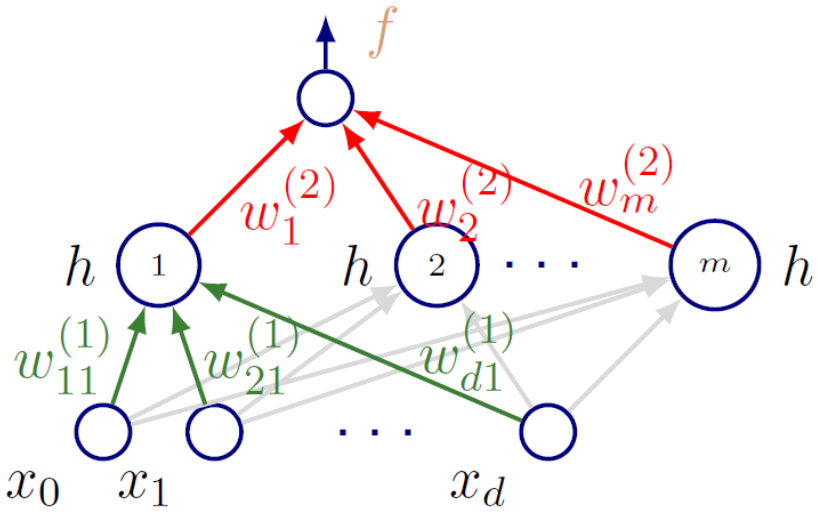
Slide adapted from TTIC 31020, Gregory Shakhnarovich

Backpropagation: example

- Output: $\mathrm{f}(a) = a$
- Hidden

$$h(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}},$$

$$h'(a) = 1 - h(a)^2.$$



- Given example x, feed-forward inputs:

input to hidden: $a_j = \sum_{i=0}^{d} w_{ij}^{(1)} x_i,$

hidden output: $z_j = \tanh(a_j),$

net output: $\hat{y} = a = \sum_{j=0}^{m} w_j^{(2)} z_j.$

Backpropagation: example

$$a_j = \sum_{i=0}^{d} w_{ij}^{(1)} x_i, \quad z_j = \tanh(a_j), \quad \hat{y} = a = \sum_{j=0}^{m} w_j^{(2)} z_j.$$

- Error on example x: $L = \frac{1}{2}(y - \hat{y})^2$.
- Output unit: $\delta = \frac{\partial L}{\partial a} = y - \hat{y}$
- Next, compute δs for the hidden units:

$$\delta_j = (1 - z_j)^2 w_j^{(2)} \delta$$

- Derivatives w.r.t. weights:

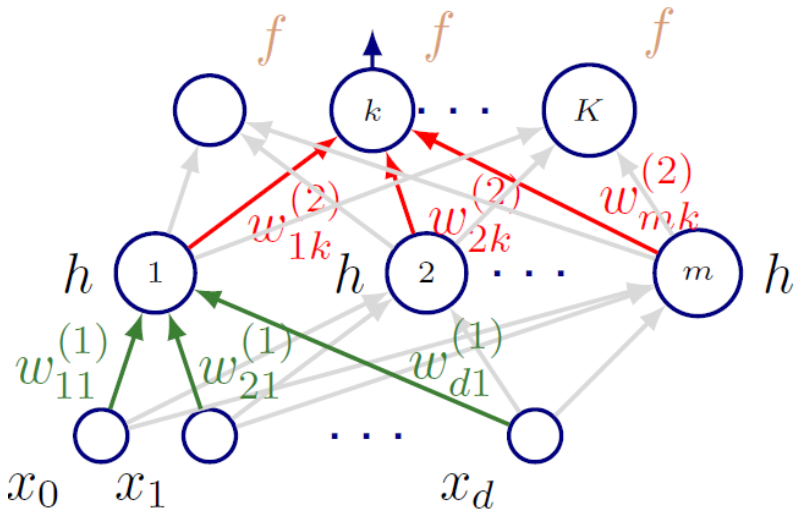
$$\frac{\partial L}{\partial w_{ij}^{(1)}} = \delta_j x_i, \qquad \frac{\partial L}{\partial w_j^{(2)}} = \delta z_j.$$

- Update weights: $\omega_j \leftarrow \omega_j - \eta \delta z_j$ and $\omega_{ij}^{(1)} \leftarrow \omega_{ij}^{(1)} - \eta \delta_j x_j$.η is called the weight decay

Multidimensional output

- Loss on example (x, y):

$$\frac{1}{2}\sum_{k=1}^{K}(y_k - \hat{y}_k)^2$$



- Now, for each output unit $\delta_k = y_k - \hat{y}_j$;
- For hidden unit j,

$$\delta_j = (1 - z_j)^2 \sum_{k=1}^{K} w_{jk}^{(2)} \delta_k.$$

Stochastic gradient descent

- Given *n* training samples, our target function can be expressed as

$$J(\mathbf{w}) = \sum_{p=1}^{n} J_p(\mathbf{w})$$

- Batch gradient descent

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \sum_{p=1}^{n} \nabla J_p(\mathbf{w})$$

- In some cases, evaluating the sum-gradient may be computationally expensive. Stochastic gradient descent samples a subset of summand functions at every step. This is very effective in the case of large-scale machine learning problems. In stochastic gradient descent, the true gradient of $J(\mathbf{w})$ is approximated by a gradient at a single example (or a mini-batch of samples):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla J_p(\mathbf{w})$$

Stochastic backpropagation

Algorithm 1 (Stochastic backpropagation)

1 **begin initialize** network topology (# hidden units), $\mathbf{w}$, criterion $\theta, \eta, m \leftarrow 0$
2 **do** $m \leftarrow m + 1$
3 $\mathbf{x}^m \leftarrow$ randomly chosen pattern
4 $w_{ij} \leftarrow w_{ij} + \eta\delta_j x_i; \quad w_{jk} \leftarrow w_{jk} + \eta\delta_k y_j$
5 **until** $\nabla J(\mathbf{w}) < \theta$
6 **return** w
7 **end**

(Duda et al. Pattern Classification 2000)

- In stochastic training, a weight update may reduce the error on the single pattern being presented, yet increase the error on the full training set.

Mini-batch Based SGD

- Divide the training set into mini-batches.
- In each epoch, randomly permute mini-batches and take a mini-batch sequentially to approximate the gradient
  - One epoch corresponds to a single presentations of all patterns in the training set
- The estimated gradient at each iteration is more reliable
- Start with a small batch size and increase the size as training proceeds

Batch Backpropagation

Algorithm 2 (Batch backpropagation)

1 **begin initialize** network topology (# hidden units), $\mathbf{w}$, criterion $\theta, \eta, r \leftarrow 0$
2 **do** $r \leftarrow r + 1$ (increment epoch)
3 $m \leftarrow 0$; $\Delta w_{ij} \leftarrow 0$; $\Delta w_{jk} \leftarrow 0$
4 **do** $m \leftarrow m + 1$
5 $\mathbf{x}^m \leftarrow$ select pattern
6 $\Delta w_{ij} \leftarrow \Delta w_{ij} + \eta \delta_j x_i$; $\Delta w_{jk} \leftarrow \Delta w_{jk} + \eta \delta_k y_j$
7 **until** $m = n$
8 $w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$; $w_{jk} \leftarrow w_{jk} + \Delta w_{jk}$
9 **until** $\nabla J(\mathbf{w}) < \theta$
10 **return** w
11 **end**

Summary

- Stochastic learning
  - Estimate of the gradient is noisy, and the weights may not move precisely down the gradient at each iteration
  - Faster than batch learning, especially when training data has redundance
  - Noise often results in better solutions
  - The weights fluctuate and it may not fully converge to a local minimum
- Batch learning
  - Conditions of convergence are well understood
  - Some acceleration techniques only operate in batch learning
  - Theoretical analysis of the weight dynamics and convergence rates are simpler

Next Lecture

- Convolutional Neural Network

Questions?

Thank You !