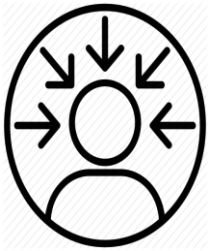
ML Basics
Neural Network

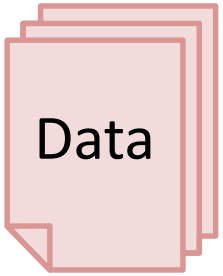
Sudeshna Sarkar

25 July 2019

Machine Learning

– Provide systems the ability to automatically learn and improve from experience





- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning

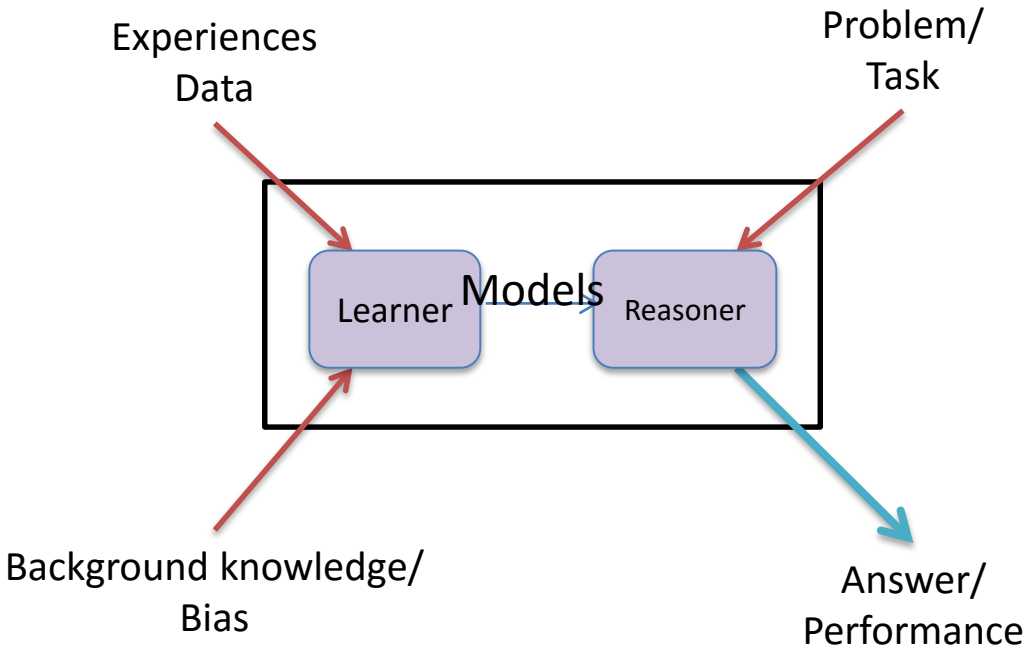
- Decision Trees
- Support Vector Machine
- **Neural Networks**

Deep Learning

Machine Learning

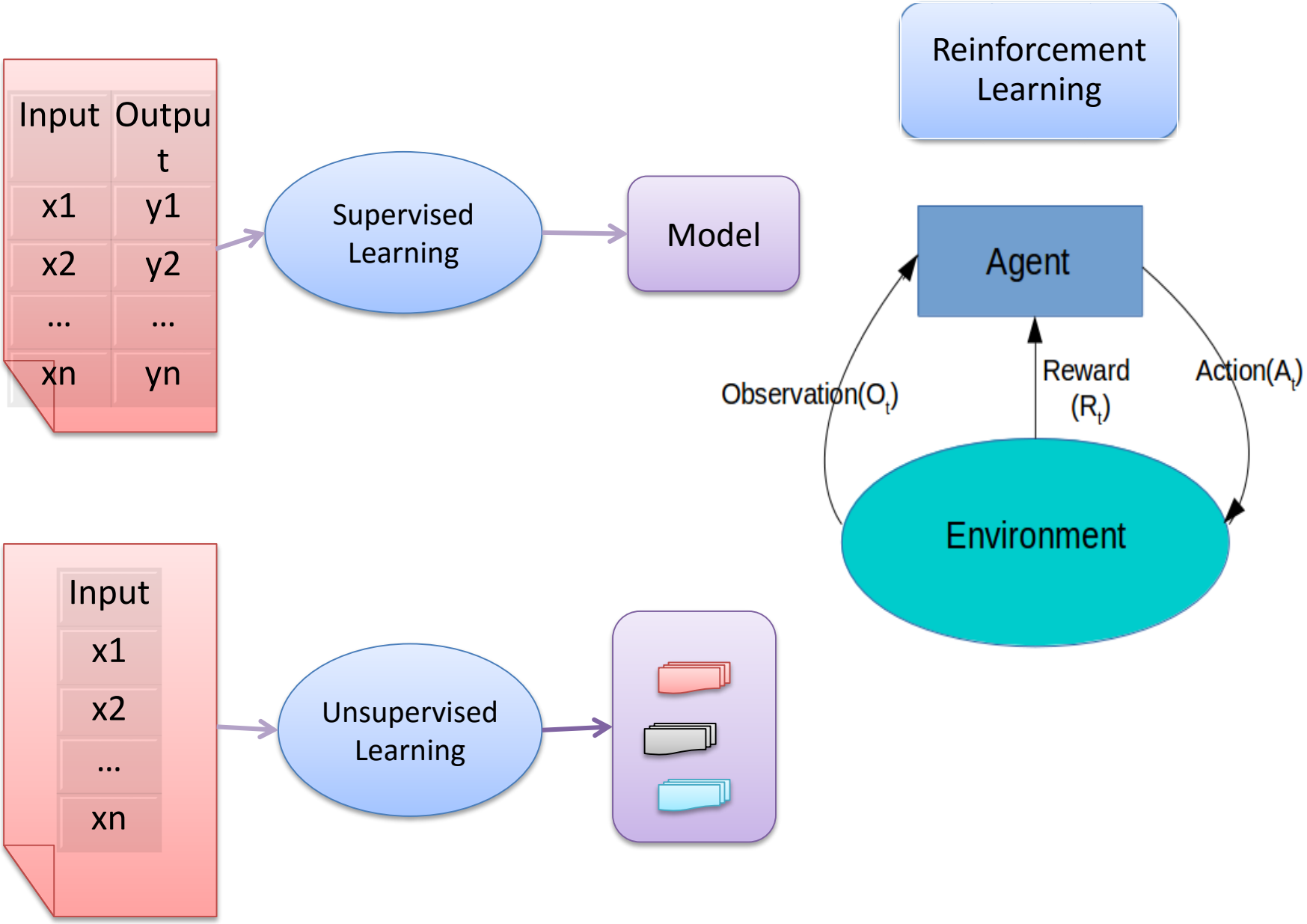
Algorithms that can

- learn from data / build a model from data
- use the model for prediction, decision making or solving some tasks



Designing a Learner

1. Choose the data
2. Choose the features
3. Choose the target function (that is to be learned)
4. Choose how to represent the target function
5. Choose a learning algorithm to infer the target function


Input
Output
x1
y1
x2
y2
...
...
xn
yn
Supervised Learning
Model
Reinforcement Learning
Agent
Observation(O_t)
Reward (R_t)
Action(A_t)
Environment
Input
x1
x2
...
xn
Unsupervised Learning

ML Methods

Supervised learning

- **Linear classifier**
- **Parametric**

Naïve Bayes, Hidden Markov models (HMM), Probabilistic graphical models

- **Non-parametric** (Instance-based functions)
  - *K*-nearest neighbors, Kernel regression, Kernel density estimation,
  - Classification and regression tree (CART), decision tree
- **Aggregation**
  - Bagging (bootstrap + aggregation), Adaboost, Random forest

Unsupervised learning

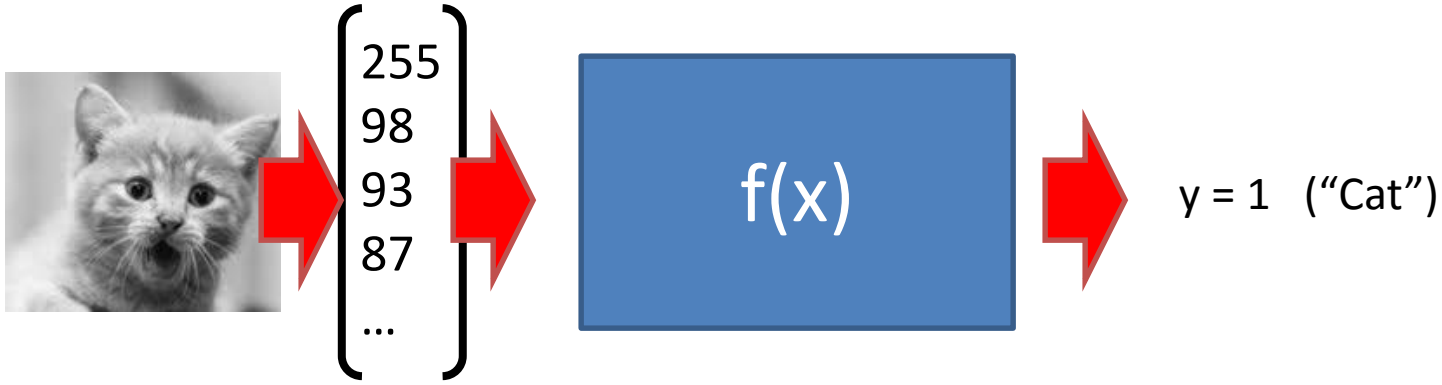
- **Clustering**
  - K-means clustering
  - Spectral clustering
- **Density Estimation**
  - Gaussian mixture model (GMM)
  - Graphical models
- **Dimensionality reduction**
  - Principal component analysis (PCA)
  - Factor analysis

Supervised Learning

- Given *labeled* training examples:

$$\mathcal{X} = \{(x^{(i)}, y^{(i)}) : i = 1, \ldots, m\}$$

- For instance: $x^{(i)}$ = vector of pixel intensities.
  $y^{(i)}$ = object class ID.



- Goal: find f(x) to predict y from x on training data.
  - Hopefully: learned predictor works on “test” data.

Linear Regression

- Predict the value of a stock tomorrow
- Find $f: \mathbb{R}^d \rightarrow \mathbb{R}$ s.t. $f(x) \approx y$ for all test examples x
- A linear regression function is a linear function of the feature vectors, i.e.,

$$f(x;\ \theta, \theta_0) = \theta . x + \theta_0 = \sum_{i=1}^{d} \theta_i x_i + \theta_0$$

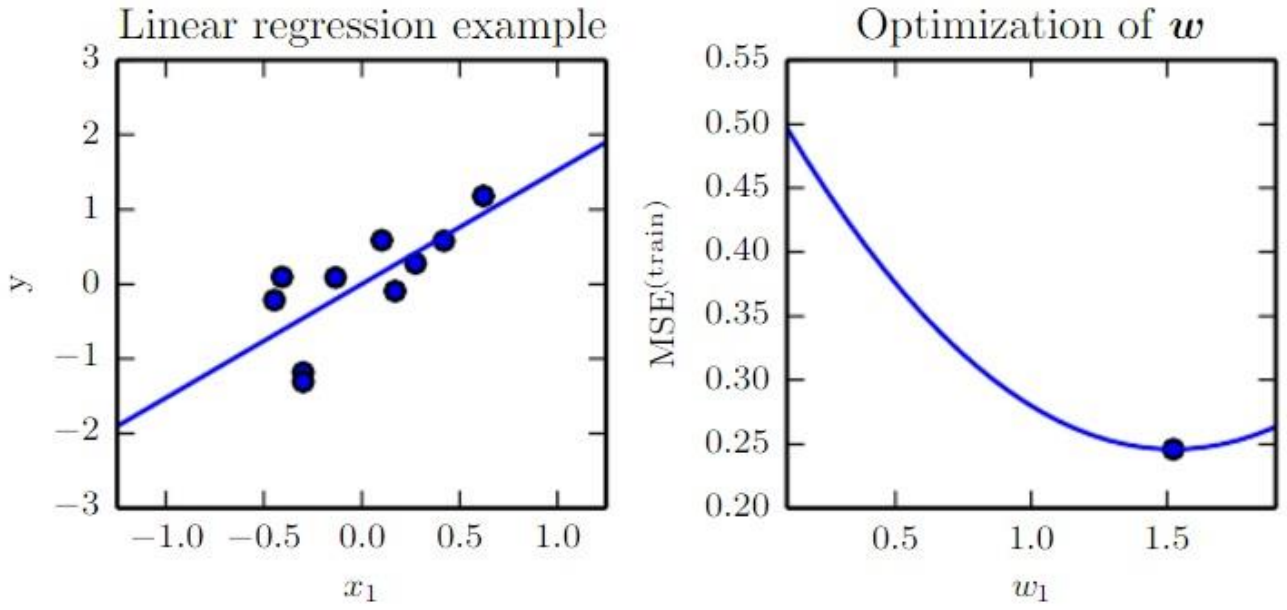


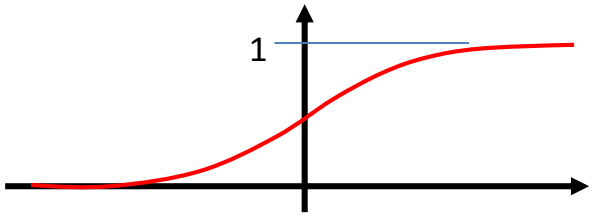
Figure 5.1: A linear regression problem, with a training set consisting of ten data points, each containing one feature. Because there is only one feature, the weight vector $\boldsymbol{w}$ contains only a single parameter to learn, w_1. *(Left)*Observe that linear regression learns to set w_1 such that the line $y = w_1 x$ comes as close as possible to passing through all the training points. *(Right)*The plotted point indicates the value of w_1 found by the normal equations, which we can see minimizes the mean squared error on the training set.

Logistic Regression

- Simple binary classification algorithm
  - Start with a function of the form:

$$f(x;\theta) \equiv \sigma(\theta^\top x) = \frac{1}{1+\exp(-\theta^\top x)}$$

  - Interpretation: f(x) is probability that y = 1.
    - Sigmoid "nonlinearity" squashes linear function to [0,1].



  - Find θ that minimizes objective:

$$\mathcal{L}(\theta) = -\sum_{i}^{m} 1\{y^{(i)}=1\}\log(f(x^{(i)};\theta)) + 1\{y^{(i)}=0\}\log(1-f(x^{(i)};\theta))$$

$\mathbb{P}(y^{(i)}=1|x^{(i)})$

$\mathbb{P}(y^{(i)}=0|x^{(i)})$

Optimization

- How do we tune θ to minimize $\mathcal{L}(\theta)$?
- One algorithm: gradient descent
  - Compute gradient:

$$\nabla_\theta \mathcal{L}(\theta) = \sum_i^m x^{(i)} \cdot (y^{(i)} - f(x^{(i)}; \theta))$$

  - Follow gradient "downhill":

$$\theta := \theta - \eta \nabla_\theta \mathcal{L}(\theta)$$

- Stochastic Gradient Descent (SGD): take step using gradient from only small batch of examples.

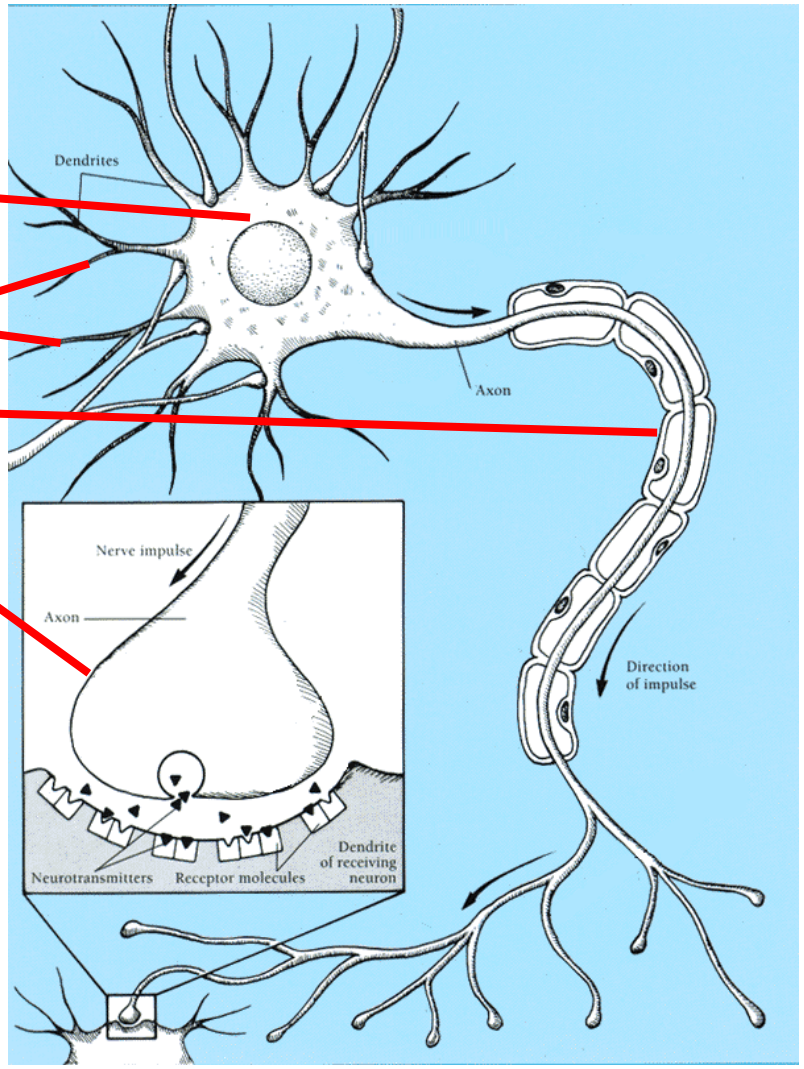
Neural Network

Neural Network Learning

- Learning approach based on modeling adaptation in biological neural systems.
- Perceptron: Initial algorithm for learning simple neural networks (single layer) developed in the 1950's.
- Backpropagation: More complex algorithm for learning multi-layer neural networks developed in the 1980's.

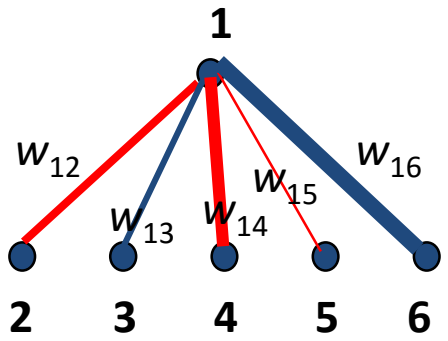
Real Neurons

- Cell structures
  - Cell body
  - Dendrites
  - Axon
  - Synaptic terminals



Simple Artificial Neuron Model
(Linear Threshold Unit)

- Model network as a graph with cells as nodes and synaptic connections as weighted edges from node *i* to node *j*, w_{ji}



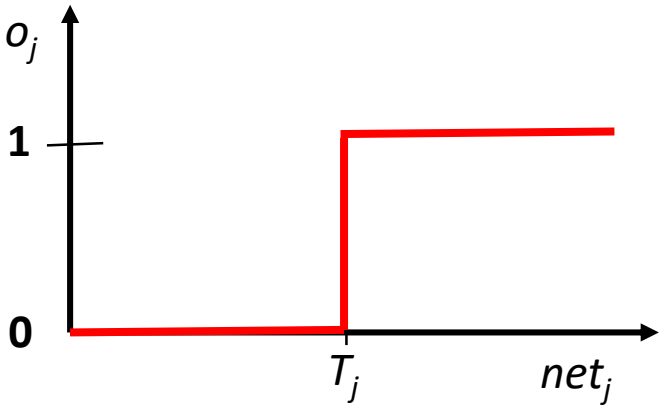
- Model net input to cell as

$$net_j = \sum_i w_{ji} o_i$$

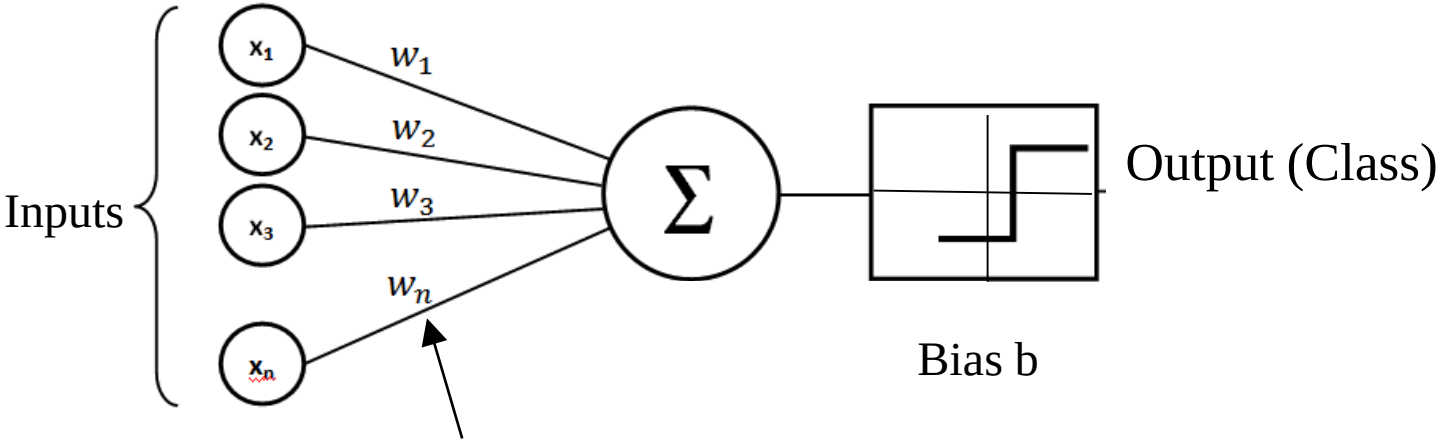
- Cell output is:

$$o_j = \begin{cases} 0 \text{ if } net_j < T_j \\ 1 \text{ if } net_i \geq T_j \end{cases}$$

(T_j is threshold for unit *j*)



Perceptron


Inputs
x1
x2
x3
xn
w1
w2
w3
wn
Σ
Output (Class)
Bias b
Weights: arbitrary,
learned parameters

Perceptron Learning Rule

- Update weights by:

$$w_{ji} = w_{ji} + \eta(t_j - o_j)o_i$$

where η is the "learning rate"

- Equivalent to rules:
  - If output is correct do nothing.
  - If output is high, lower weights on active inputs
  - If output is low, increase weights on active inputs
- Also adjust threshold to compensate:

$$T_j = T_j - \eta(t_j - o_j)$$

Perceptron Learning Algorithm

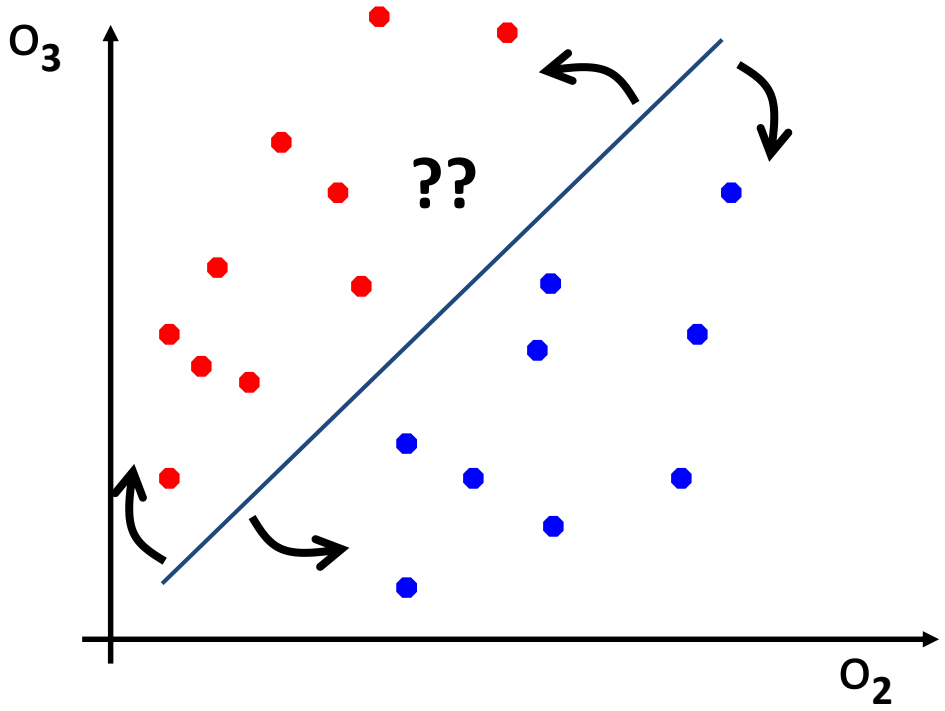
- Iteratively update weights until convergence.

Initialize weights to random values
Until outputs of all training examples are correct
    For each training pair, *E*, do:
        Compute current output o_j for *E* given its inputs
        Compare current output to target value, t_j, for *E*
        Update synaptic weights and threshold using learning rule

- Each execution of the outer loop is typically called an *epoch*.

Perceptron as a Linear Separator

- Since perceptron uses linear threshold function, it is searching for a linear separator that discriminates the classes.



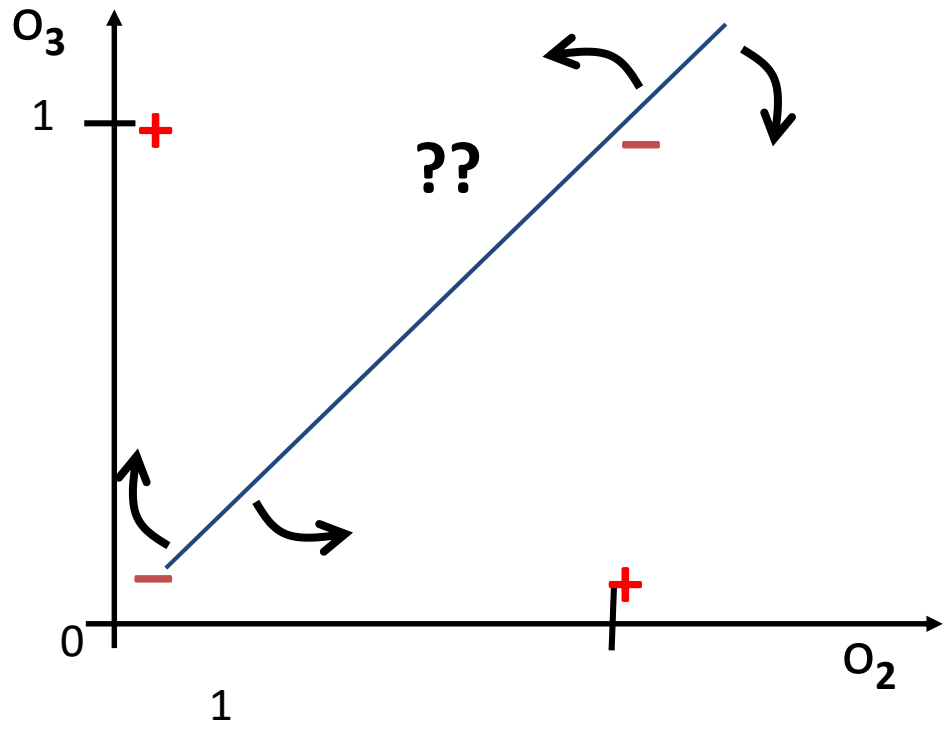
$$w_{12}o_2 + w_{13}o_3 > T_1$$

$$o_3 > -\frac{w_{12}}{w_{13}}o_2 + \frac{T_1}{w_{13}}$$

Or hyperplane in n-dimensional space

Concept Perceptron Cannot Learn

- Cannot learn exclusive-or, or parity function in general.

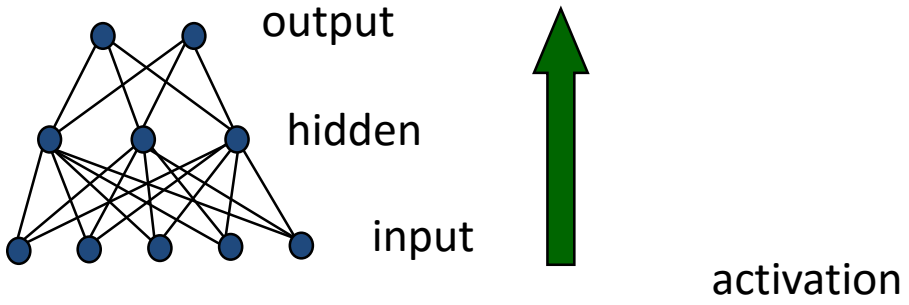


Perceptron Convergence

- **Perceptron convergence theorem**: If the data is linearly separable and therefore a set of weights exist that are consistent with the data, then the Perceptron algorithm will eventually converge to a consistent set of weights.

Multi-Layer Feed-Forward Networks

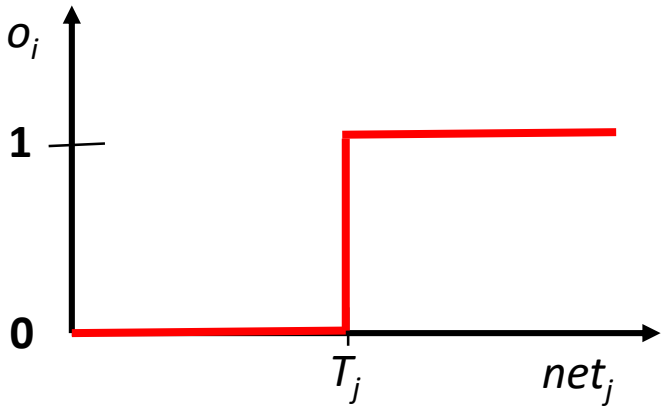
- Multi-layer networks can represent arbitrary functions.
- A typical multi-layer network consists of an input, hidden and output layer, each fully connected to the next, with activation feeding forward.



- The weights determine the function computed. Given an arbitrary number of hidden units, any boolean function can be computed with a single hidden layer.

Multi-Layer Nets

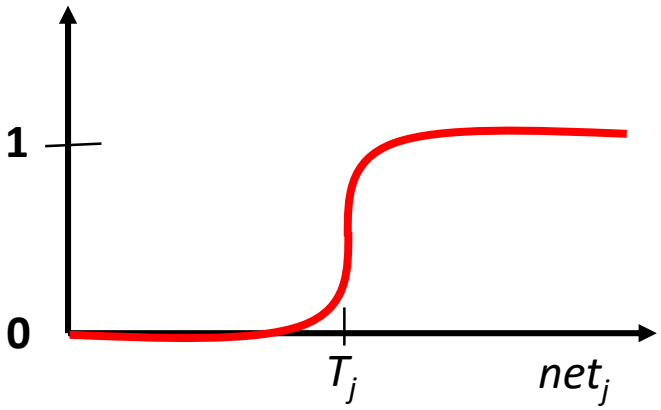
- To do gradient descent, we need the output of a unit to be a differentiable function of its input and weights.
- Standard linear threshold function is not differentiable at the threshold.



Differentiable Output Function

- Need non-linear output function to move beyond linear functions.
  - A multi-layer linear network is still linear.
- Standard solution is to use the non-linear, differentiable sigmoidal "logistic" function:

$$o_j = \frac{1}{1 + e^{-(net_j - T_j)}}$$

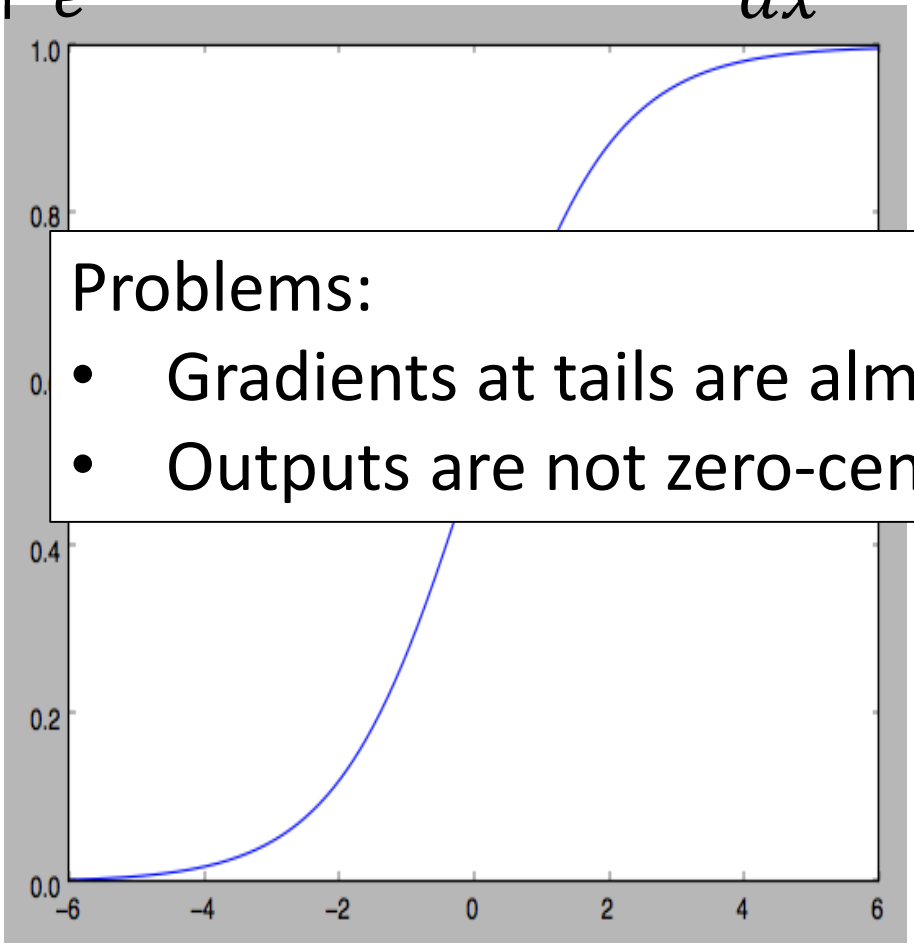


Can also use tanh or Gaussian output function

Activation Functions: Sigmoid / Logistic

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

$$\frac{df}{dx} = f(x)(1 - f(x))$$

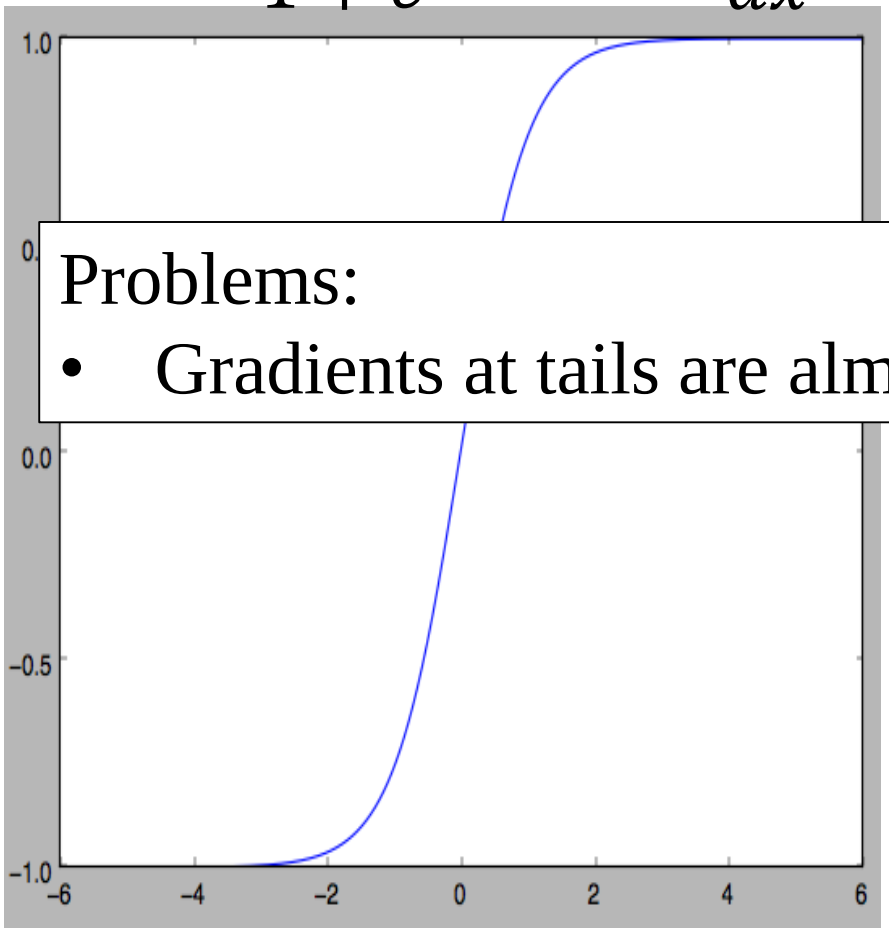


Problems:

- Gradients at tails are almost zero
- Outputs are not zero-centered

Activation Functions: Tanh

$$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \qquad \frac{df}{dx} = 1 - f(x)^2$$



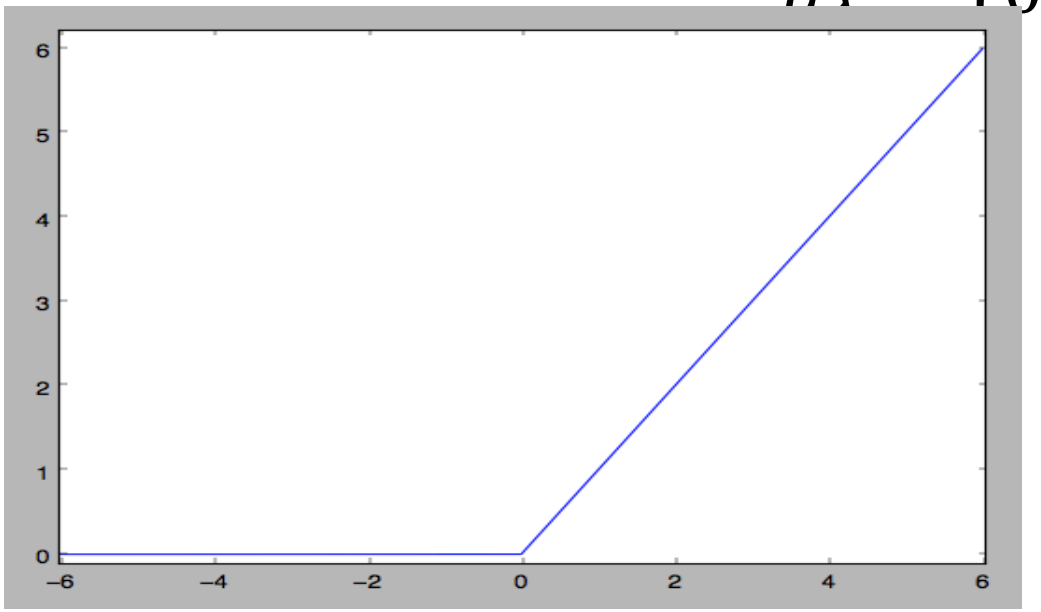
Problems:

- Gradients at tails are almost zero

Activation Functions:
ReLU (Rectified Linear Unit)

$$f(x) = \max(x, 0)$$

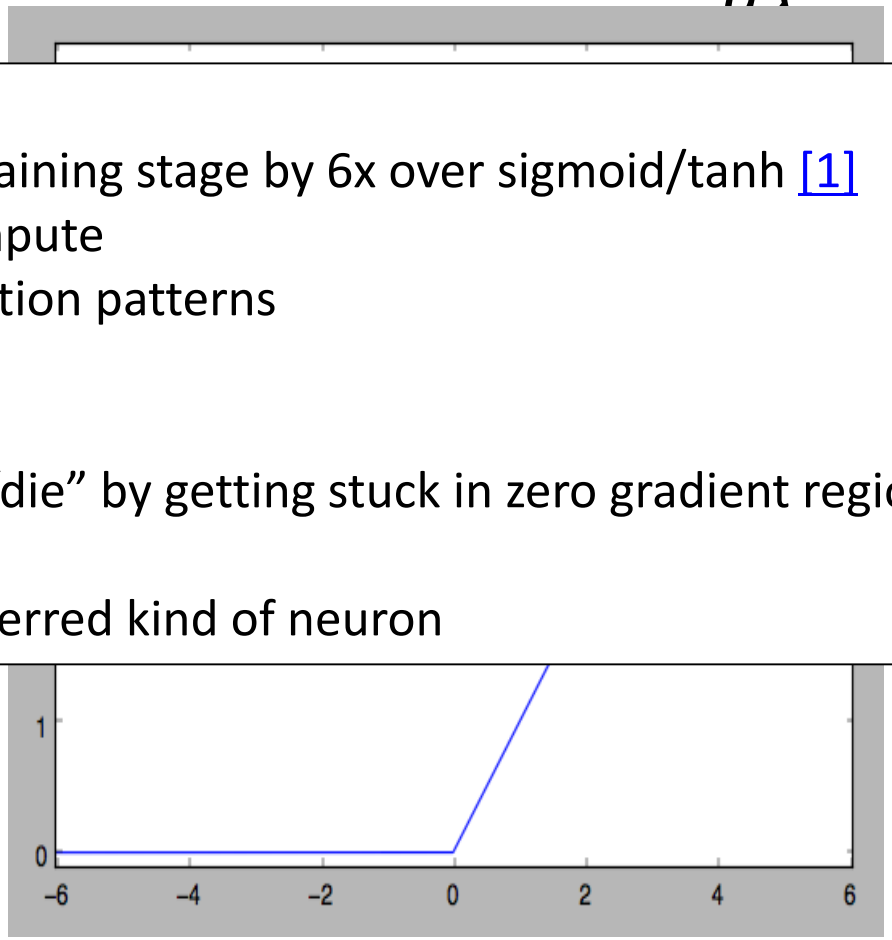
$$\frac{df}{dx} = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x < 0 \end{cases}$$



Activation Functions:
ReLU (Rectified Linear Unit)

$$f(x) = \max(x, 0)$$

$$\frac{df}{dx} = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x < 0 \end{cases}$$



Pros:

- Accelerates training stage by 6x over sigmoid/tanh [1]
- Simple to compute
- Sparser activation patterns

Cons:

- Neurons can “die” by getting stuck in zero gradient region

Summary:

- Currently preferred kind of neuron

Universal Approximation Theorem

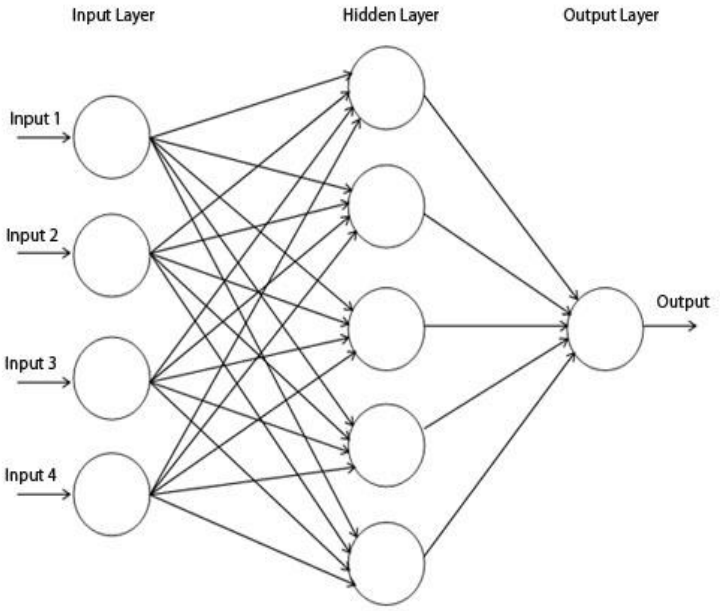
- Multilayer perceptron with a single hidden layer and linear output layer can approximate any continuous function on a compact subset of $\mathbb{R}^n$ to within any desired degree of accuracy.
- Assumes activation function is bounded, non-constant, monotonically increasing.
- Also applies for ReLU activation function.

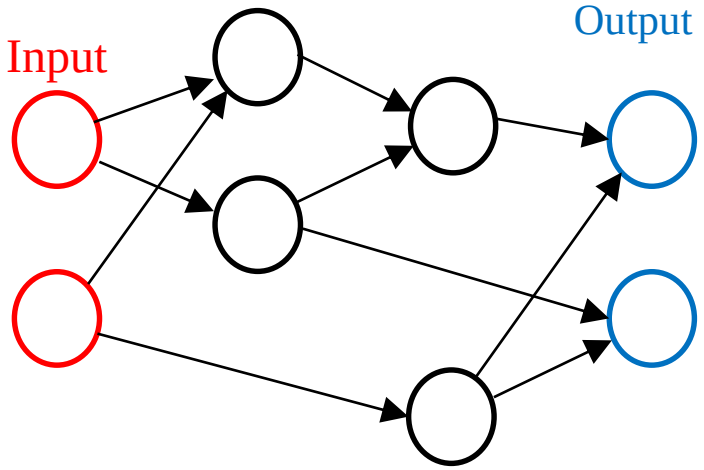
Why Use Deep Networks?

- Functions representable with a deep rectifier network can require an exponential number of hidden units with a shallow (one hidden layer) network
- Piecewise linear networks (e.g. using ReLU) can represent functions that have a number of regions exponential in depth of network.
  - Can capture repeating / mirroring / symmetric patterns in data.
  - Empirically, greater depth often results in better generalization.

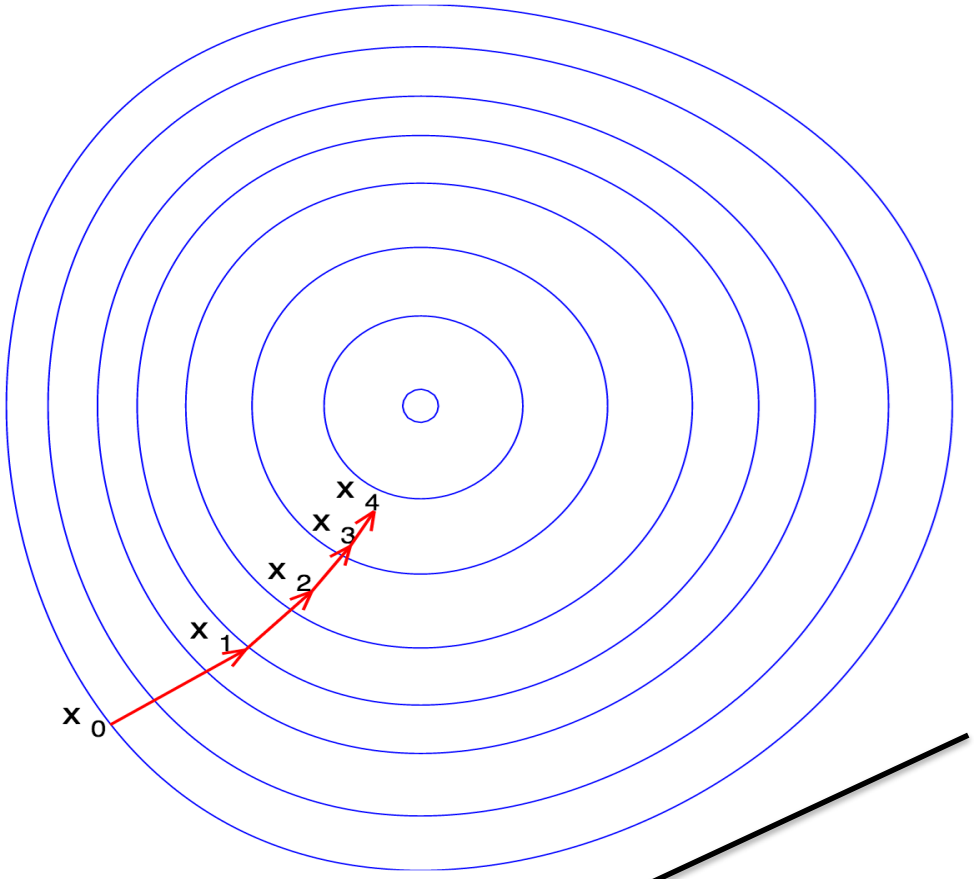
Neural Network Architecture

- **Architecture**: refers to which parameters (e.g. weights) are used in the network and their topological connectivity.
- **Fully connected:** A common connectivity pattern for multilayer perceptrons. All possible connections made between layers *i*-1 and *i*.





Gradient Descent



Step size
or
Learning rate

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n)$$

Gradient Descent

- Define objective to minimize error:

$$E(W) = \sum_{d \in D} \sum_{k \in K} (t_{kd} - o_{kd})^2$$

where D is the set of training examples, K is the set of output units, t_{kd} and o_{kd} are, respectively, the teacher and current output for unit k for example d.

- The derivative of a sigmoid unit with respect to net input is:

$$\frac{\partial o_j}{\partial net_j} = o_j(1 - o_j)$$

- Learning rule to change weights to minimize error is:

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}}$$

Stochastic gradient descent

- Stochastic gradient descent (Wikipedia)
- Gradient of sum of n terms where n is large
- Sample rather than computing the full sum
  - Sample size *s* is "mini-batch size"
  - Could be 1 (very noisy gradient estimate)
  - Could be 100 (collect photos 100 at a time to find each noisy "next" estimate for the gradient)
- Use same step as in gradient descent to the estimated gradient

Problem Statement

- Take the gradient of an arbitrary program or model (e.g. a neural network) with respect to the parameters in the model (e.g. weights).

Review: Chain Rule in One Dimension

- Suppose $f: \mathbb{R} \to \mathbb{R}$ and $g: \mathbb{R} \to \mathbb{R}$
- Define $h(x) = f(g(x))$

- Then what is $h'(x) = dh/dx$?

$$h'(x) = f'\big(g(x)\big)g'(x)$$

Chain Rule in Multiple Dimensions

- Suppose $f: \mathbb{R}^m \to \mathbb{R}$ and $g: \mathbb{R}^n \to \mathbb{R}^m$, and $\mathbf{x} \in \mathbb{R}^n$
- Define

$$h(\mathbf{x}) = f(g_1(\mathbf{x}), \dots, g_m(\mathbf{x}))$$

- Then we can define partial derivatives using the multidimensional chain rule:

$$\frac{\partial f}{\partial x_i} = \sum_{l=1}^{m} \frac{\partial f}{\partial g_j} \frac{\partial g_j}{\partial x_i}$$

Solution for Simplified Chain of Dependencies

- Suppose $\pi(i) = i - 1$
- The computation:

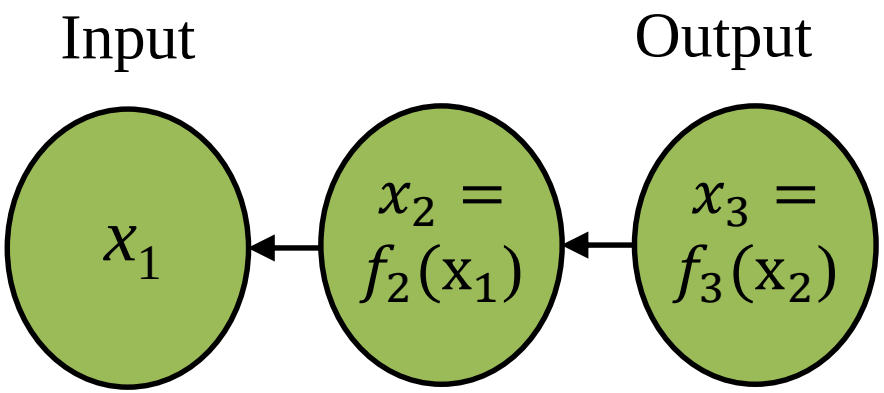
  For $i = n + 1, \ldots, N$

  $$x_i = f_i(\mathrm{x}_{i-1})$$

- What is $\dfrac{dx_N}{dx_N}$?

$$\frac{dx_N}{dx_N} = 1$$

For example:



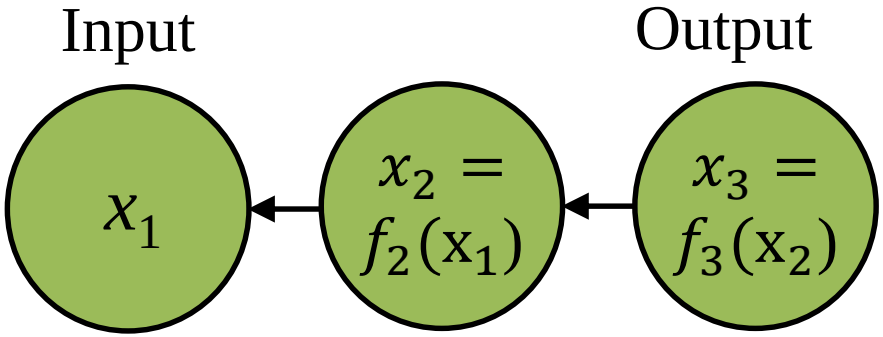
Solution for Simplified Chain of Dependencies

- Suppose $\pi(i) = i - 1$
- The computation:

For $i = n + 1, \ldots, N$

$$x_i = f_i(\mathrm{x}_{i-1})$$

For example:



What is $\frac{dx_N}{dx_i}$ in terms of $\frac{dx_N}{dx_{i+1}}$?

$$\frac{dx_N}{dx_i} = \frac{dx_N}{dx_{i+1}}\left(\frac{\partial x_{i+1}}{\partial x_i}\right)$$

$$dx_{i+1} = dx_i\left(\frac{\partial x_{i+1}}{\partial x_i}\right) \Rightarrow \frac{dx_{i+1}}{dx_N} = \frac{dx_i}{dx_N}\left(\frac{\partial x_{i+1}}{\partial x_i}\right)$$

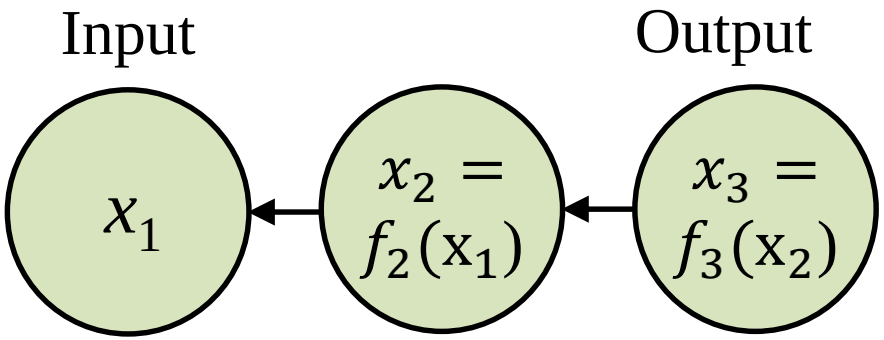
Solution for Simplified Chain of Dependencies

- Suppose $\pi(i) = i - 1$
- The computation:

For $i = n + 1, \ldots, N$

$$x_i = f_i(\mathrm{x}_{i-1})$$

For example:



What is $\frac{dx_N}{dx_i}$ in terms of $\frac{dx_N}{dx_{i+1}}$?

$$\frac{dx_N}{dx_i} = \frac{dx_N}{dx_{i+1}}\left(\frac{\partial x_{i+1}}{\partial x_i}\right)$$

Conclusion: run the computation forwards. Then initialize $\frac{dx_N}{dx_N} = 1$ and work **backwards** through the computation to find $\frac{dx_N}{dx_i}$ for each *I* from $\frac{dx_N}{dx_{i+1}}$.

Backpropagation Learning Rule

- Each weight changed by:

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

$$\delta_j = o_j(1-o_j)(t_j - o_j) \qquad \text{if } j \text{ is an output unit}$$

$$\delta_j = o_j(1-o_j)\sum_k \delta_k w_{kj} \qquad \text{if } j \text{ is a hidden unit}$$

where η is a constant called the learning rate

t_j is the correct teacher output for unit j

δ_j is the error measure for unit j

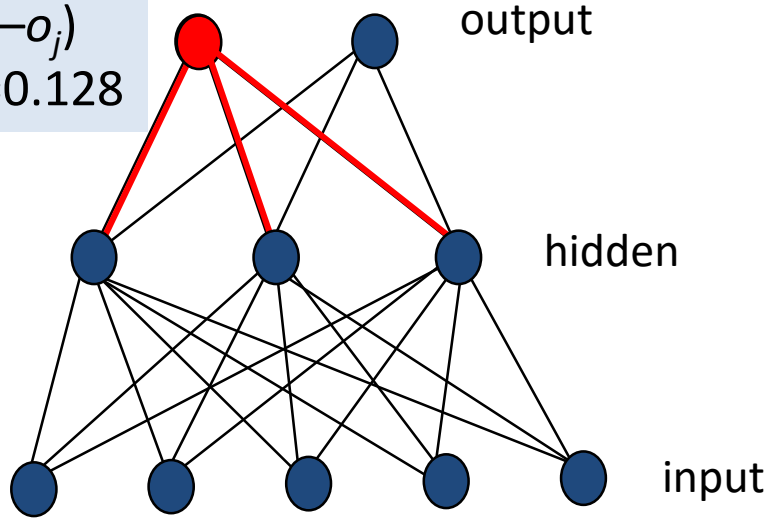
Error Backpropagation

- First calculate error of output units and use this to change the top layer of weights.

Current output: o_j=0.2
Correct output: t_j=1.0
Error $\delta_j = o_j(1–o_j)(t_j–o_j)$
0.2(1–0.2)(1–0.2)=0.128

Update weights into *j*

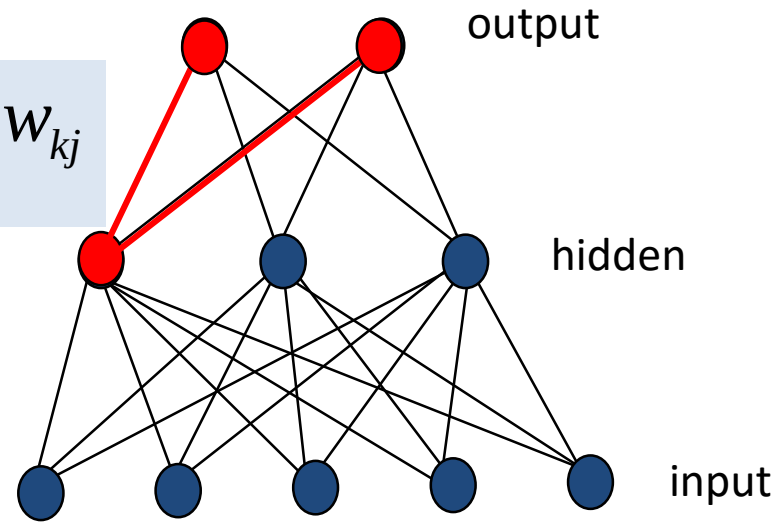
$$\Delta w_{ji} = \eta \delta_j o_i$$



Error Backpropagation

- Next calculate error for hidden units based on errors on the output units it feeds into.

$$\delta_j = o_j(1-o_j)\sum_k \delta_k w_{kj}$$



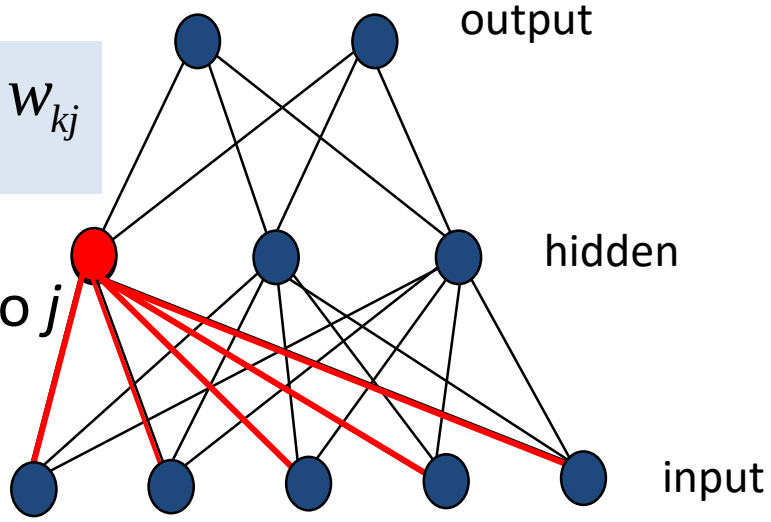
Error Backpropagation

- Finally update bottom layer of weights based on errors calculated for hidden units.

$$\delta_j = o_j(1-o_j)\sum_k \delta_k w_{kj}$$

Update weights into *j*

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



Backpropagation Training Algorithm

Create the 3-layer network with *H* hidden units with full connectivity between layers. Set weights to small random real values.

Until all training examples produce the correct value (within ε), or mean squared error ceases to decrease, or other termination criteria:

Begin epoch

For each training example, *d*, do:

Calculate network output for *d*'s input values

Compute error between current output and correct output for *d*

Update weights by backpropagating error and using learning rule

End epoch

Comments on Training Algorithm

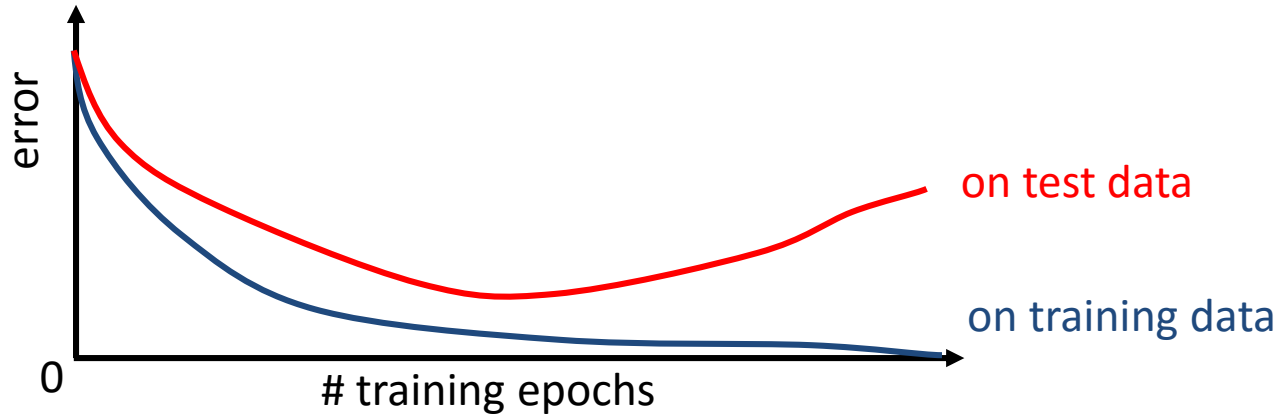
- Not guaranteed to converge to zero training error, may converge to local optima or oscillate indefinitely.
- However, in practice, does converge to low error for many large networks on real data.
- Many epochs (thousands) may be required, hours or days of training for large networks.
- To avoid local-minima problems, run several trials starting with different random weights (*random restarts*).
  - Take results of trial with lowest training set error.
  - Build a committee of results from multiple trials (possibly weighting votes by training set accuracy).

Hidden Unit Representations

- Trained hidden units can be seen as newly constructed features that make the target concept linearly separable in the transformed space.
- On many real domains, hidden units can be interpreted as representing meaningful features such as vowel detectors or edge detectors, etc..
- However, the hidden layer can also become a distributed representation of the input in which each individual unit is not easily interpretable as a meaningful feature.

Over-Training Prevention

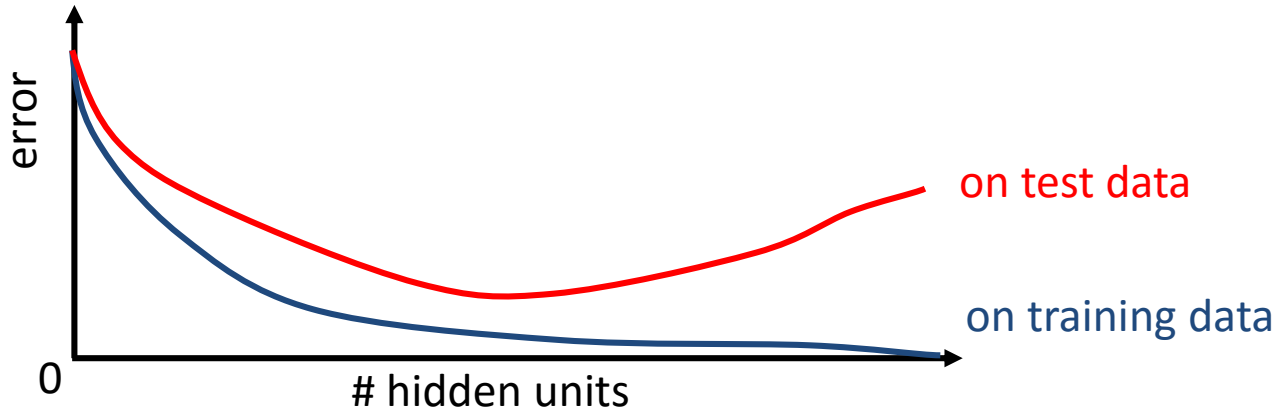
- Running too many epochs can result in over-fitting.



- Keep a hold-out validation set and test accuracy on it after every epoch. Stop training when additional epochs actually increase validation error.
- To avoid losing training data for validation:
  - Use internal 10-fold CV on the training set to compute the average number of epochs that maximizes generalization accuracy.
  - Train final network on complete training set for this many epochs.

Determining the Best Number of Hidden Units

- Too few hidden units prevents the network from adequately fitting the data.
- Too many hidden units can result in over-fitting.



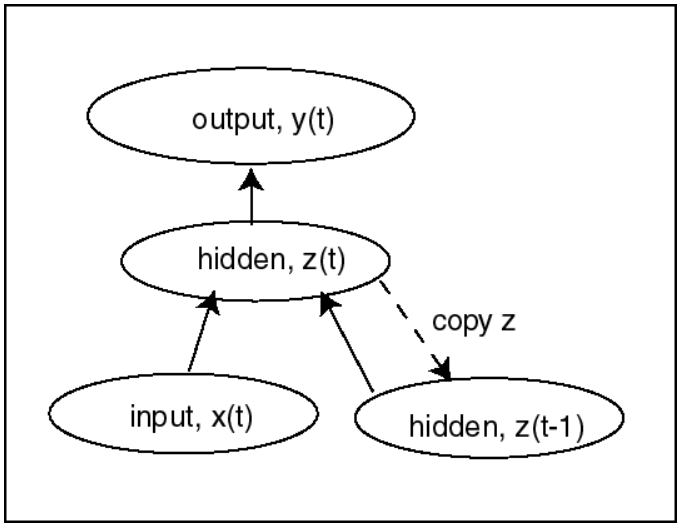
- Use internal cross-validation to empirically determine an optimal number of hidden units.

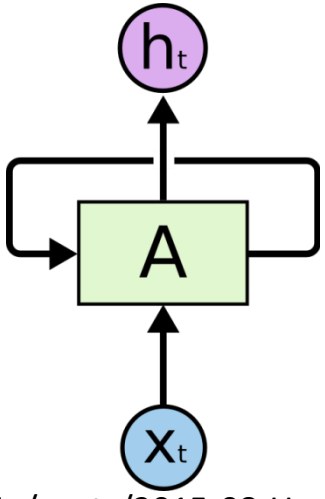
Recurrent Neural Networks (RNN)

- Add feedback loops where some units' current outputs determine some future network inputs.
- RNNs can model dynamic finite-state machines, beyond the static combinatorial circuits modeled by feed-forward networks.

Simple Recurrent Network (SRN)

- Initially developed by Jeff Elman ("*Finding structure in time*," 1990).
- Additional input to hidden layer is the state of the hidden layer in the previous time step.

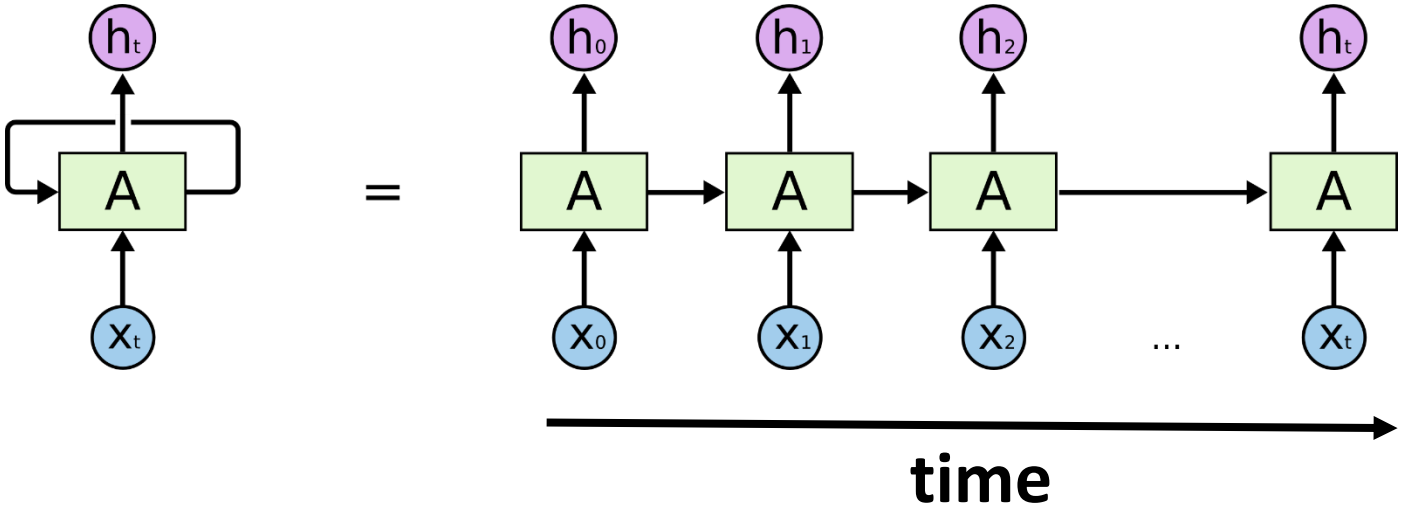




http://colah.github.io/posts/2015-08-Understanding-LSTMs/

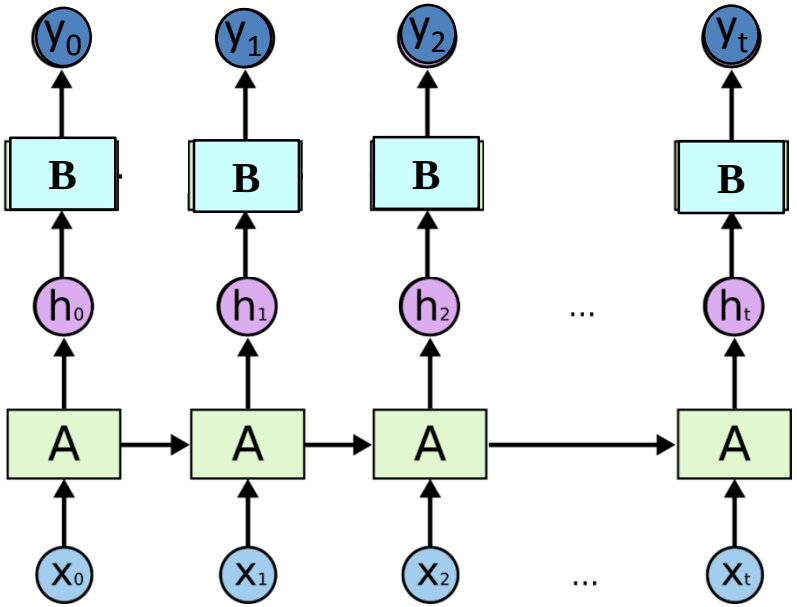
Unrolled RNN

- Behavior of RNN is perhaps best viewed by "unrolling" the network over time.



Training RNN's

- RNNs can be trained using "backpropagation through time."
- Can viewed as applying normal backprop to the unrolled network.



training outputs

training inputs

backpropagated errors

Conclusions

- “Feed forward” neural networks are a powerful machine learning technique for feature-vector classification.
- Training becomes increasingly difficult as the number of neural layers increases.
  - Perceptron for training a single layer network
  - Backpropagation for multi-layer networks
- Recurrent neural networks can perform sequence modeling and labeling, but backpropagation thru time has problems training unrolled networks that are “deep in time.”