

Statistical Machine Learning
Methods for Bioinformatics
**III. Neural Network & Deep
Learning Theory**

William and Nancy Thompson Missouri Distinguished
Professor

Department of Electrical Engineering & Computer Science
University of Missouri

Free for Academic Use. Copyright @ Jianlin Cheng & original sources of some materials.

Classification Problem

Input

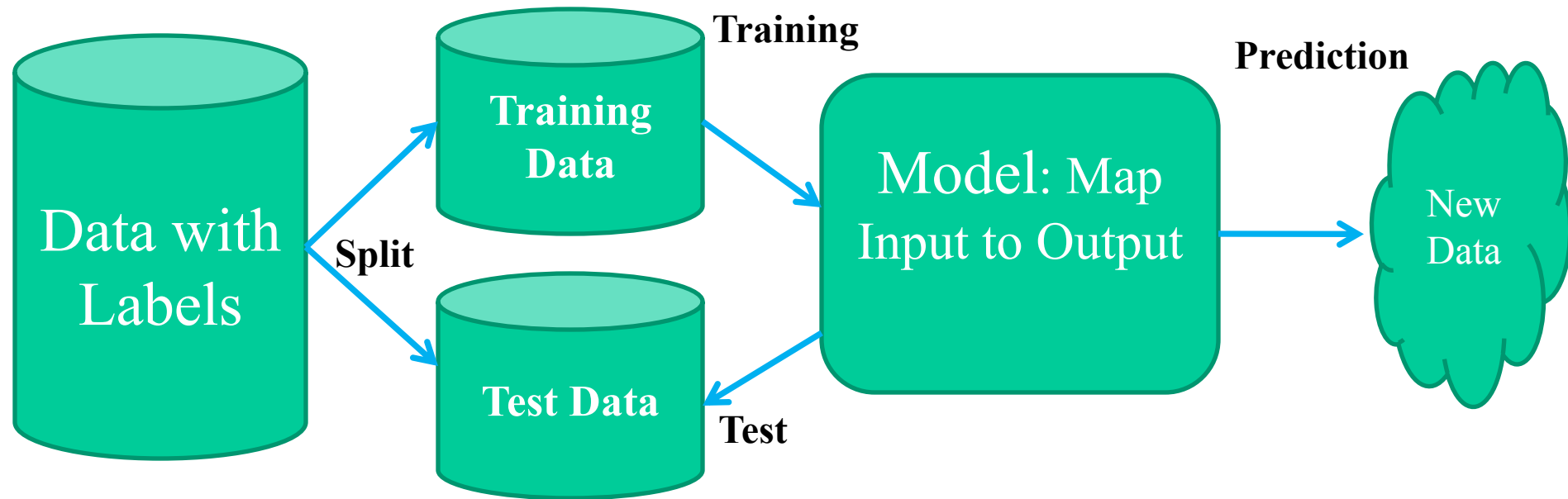
Output

Legs	weight	size	...				Feature m	Category / Label
4	100							Mammal
80	0.1							Bug

Question: How to automatically predict output given input?

Idea: Learn from known examples and generalize to unknown ones.

Data Driven Machine Learning Approach



Input: words of news

Output: politics, sports, entertainment,

...

Training: Build a model (classifier)

Test: Test the model

Key idea: Learn from known data and **Generalize** to unseen data

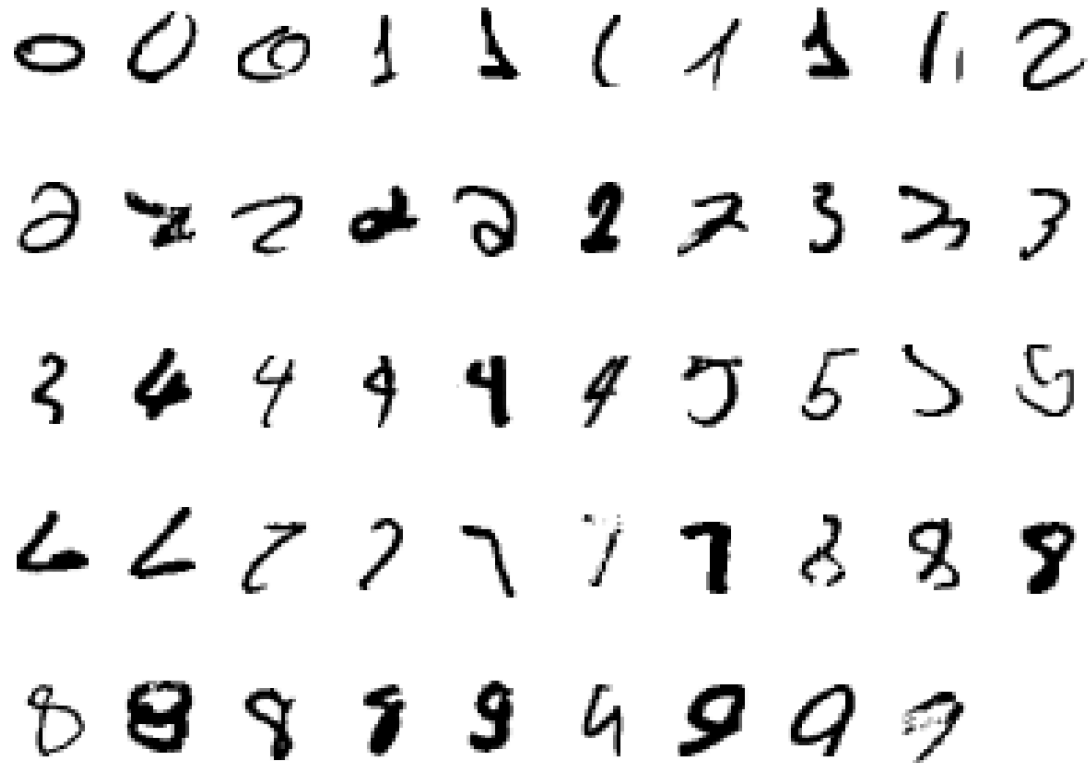
Outline

- Introduction
- Linear regression
- Linear Discriminant function (classification) and one node neural network / perceptron
- Multi-layer network
- Prevent overfitting & speedup learning & stochastic gradient descent
- Deep neural network
- Convolutional neural network
- Recurrent neural network
- Advanced networks (GAN, inception, capsule, LSTM, deep auto-encoder)
- Deep belief network

Machine Learning

- **Supervised learning** (training with labeled data), **un-supervised learning** (clustering un-labeled data), and **semi-supervised learning** (use both labeled and unlabeled data)
- Supervised learning: **classification** and **regression**
- **Classification**: output is discrete value
- **Regression**: output is real value

Learning Example: Recognize Handwriting



Classification: recognize each number

Clustering: cluster the same numbers together

Regression: predict the index of Dow-Jones

Neural Network

- Neural Network can do both supervised learning and un-supervised learning
- Neural Network can do both regression and classification
- Neural Network has both statistical and artificial intelligence roots

History of Neural Networks

- 1957 – perceptron (Rosenblatt)
- 1960s – almost died
- 1980s – neural networks (multi-layer perceptron)
- 1990-2000s – fell out of favor
- 2010s – deep learning (hottest)



Geoffrey Hinton

[FOLLOW](#)

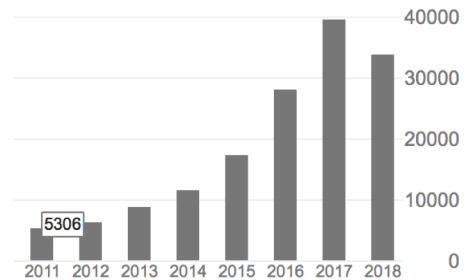
Emeritus Prof. Comp Sci, U.Toronto & Engineering Fellow, Google
 Verified email at cs.toronto.edu - [Homepage](#)

[machine learning](#) [neural networks](#) [artificial intelligence](#) [cognitive science](#) [computer science](#)

TITLE	CITED BY	YEAR
Learning internal representations by error-propagation DE Rumelhart, GE Hinton, RJ Williams Parallel Distributed Processing: Explorations in the Microstructure of ...	44486 *	1986
Learning representations by back-propagating errors DE Rumelhart, GE Hinton, RJ Williams Nature 323, 533-536	39834 *	1986
Imagenet classification with deep convolutional neural networks A Krizhevsky, I Sutskever, GE Hinton Advances in neural information processing systems, 1097-1105	28519	2012

Cited by [VIEW ALL](#)

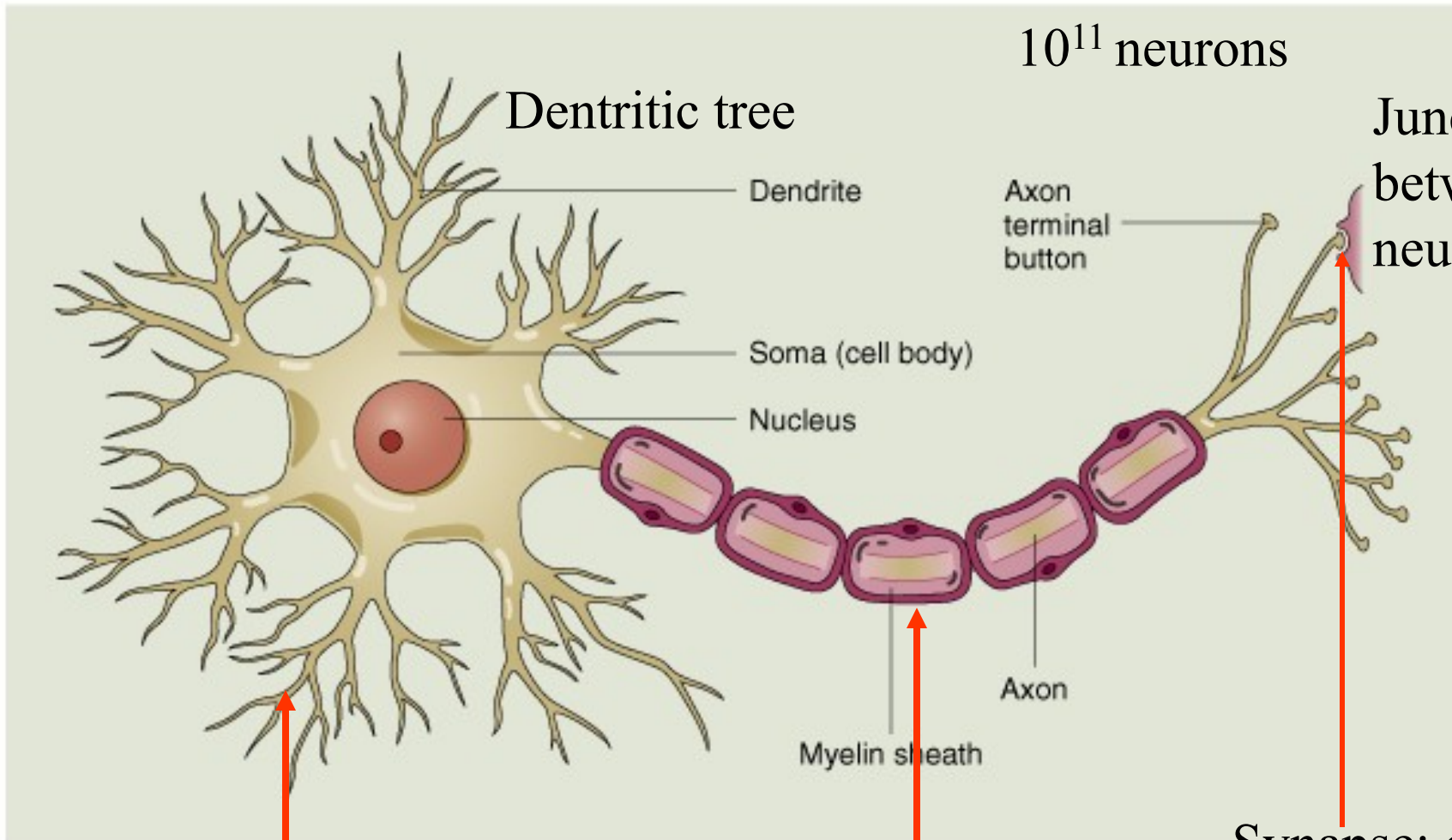
	All	Since 2013
Citations	240261	139376
h-index	140	104
i10-index	329	235



Roots of Neural Network

- Artificial intelligence root (neuron science)
- **Statistical root** (linear regression, generalized linear regression, discriminant analysis. This is our focus.)

A Typical Cortical Neuron



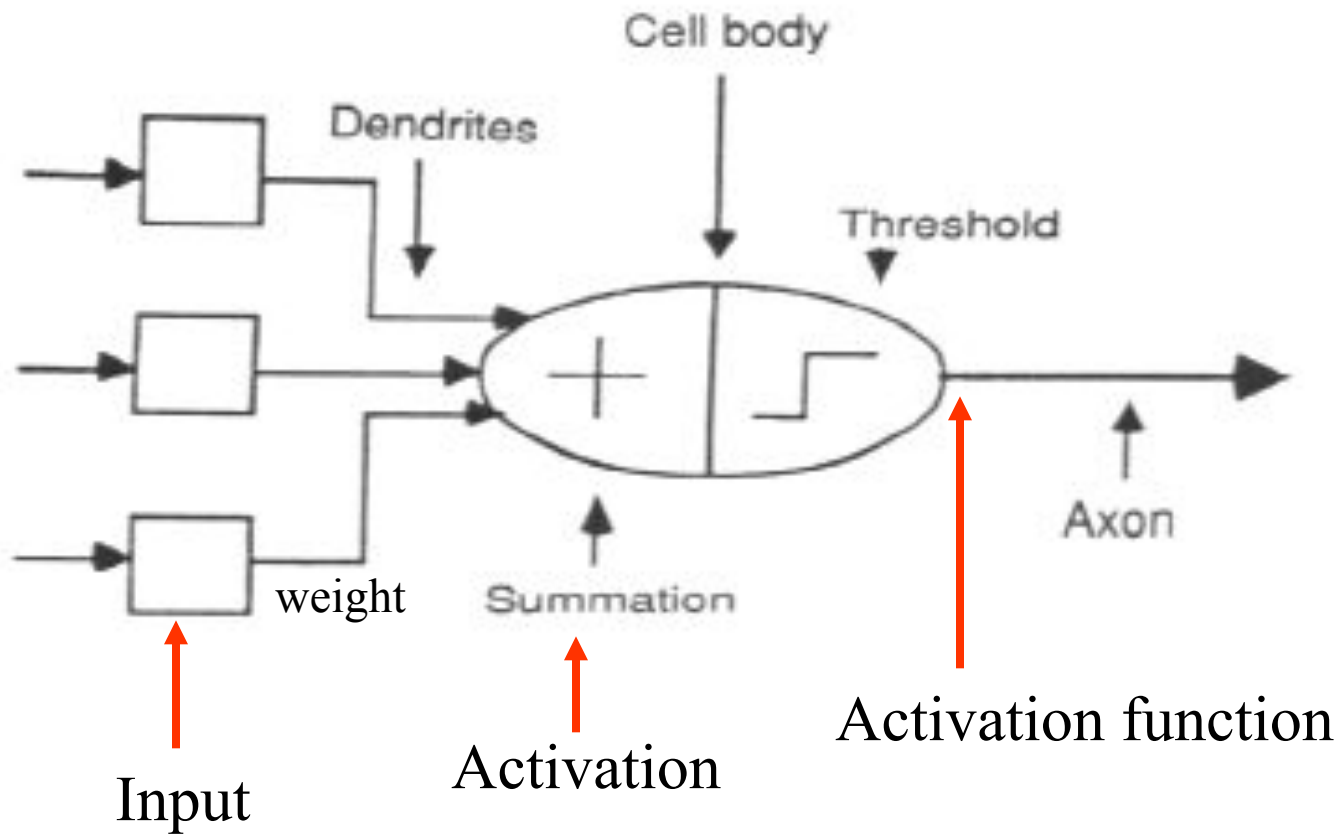
© 2000 John Wiley & Sons, Inc.

Collect chemical signals

Axon: generate Potentials (Fire/not Fire)

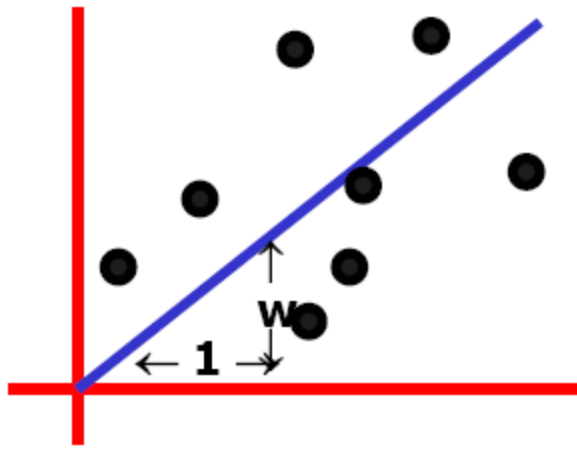
Synapse: control release chemical transmitters.

A Neural Model



Statistics Root: Linear Regression

Example



DATASET

inputs	outputs
$x_1 = 1$	$y_1 = 1$
$x_2 = 3$	$y_2 = 2.2$
$x_3 = 2$	$y_3 = 2$
$x_4 = 1.5$	$y_4 = 1.9$
$x_5 = 4$	$y_5 = 3.1$

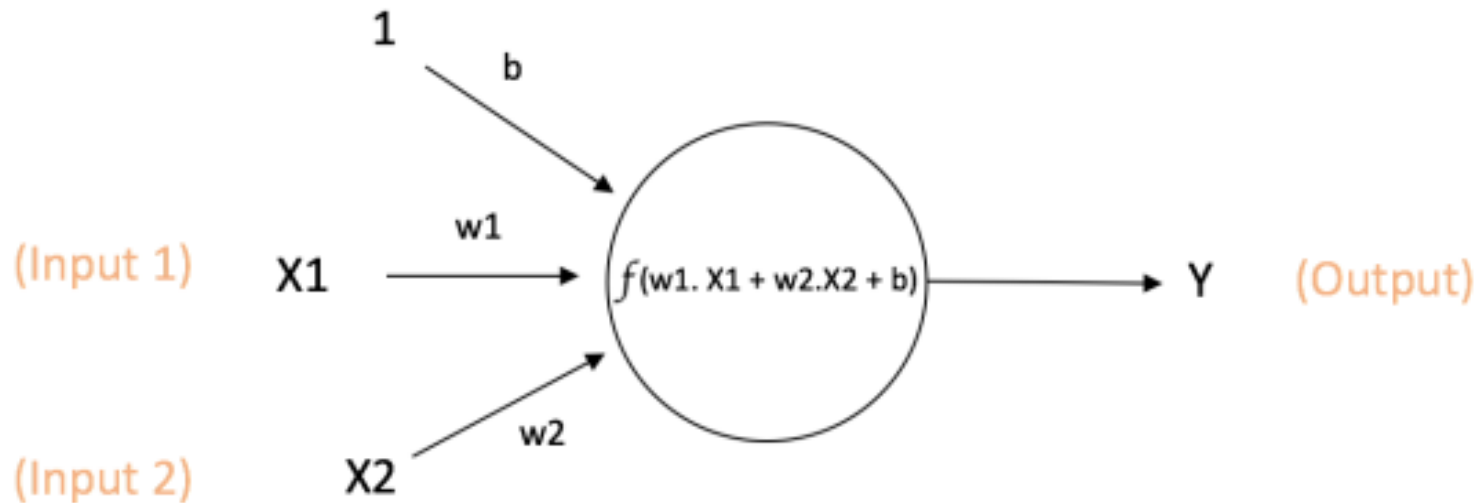
Fish length vs. weight?

X : input or predictor

Y : output or response

Goal: learn a linear function $E[y|x] = wx + b$.

One Node Neural Network



$$\text{Output of neuron} = Y = f(w_1 \cdot X_1 + w_2 \cdot X_2 + b)$$

Linear Regression

Definition of a linear model:

- $y = wx + b + \text{noise}$.
- $\text{noise} \sim \text{N}(0, \sigma^2)$, assume σ is a constant.
- $y \sim \text{N}(wx + b, \sigma^2)$
- Estimate expected value of y given x ($\text{E}[y|x] = wx + b$).
- Given a set of data $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$, to find the optimal parameters w and b .

Objective Function

- Least square error: $\sum_{i=1}^N (y_i - wx_i - b)^2$
- Maximum Likelihood: $\prod_{i=1}^N P(y_i | x_i, w, b)$
- Minimizing square error is equivalent to maximizing likelihood

Maximize Likelihood

$$\prod_{i=1}^N P(y_i | x_i, w, b) = \prod_{i=1}^N \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i - wx_i - b)^2}{2\sigma^2}}$$

Minimize negative log-likelihood:

$$\begin{aligned} -\log\left(\prod_{i=1}^N P(y_i | x_i, w, b)\right) &= -\log\left(\prod_{i=1}^N \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i - wx_i - b)^2}{2\sigma^2}}\right) = -\sum_{i=1}^N \left(-\log(\sqrt{2\pi\sigma^2}) - \frac{(y_i - wx_i - b)^2}{2\sigma^2}\right) \\ &= \sum_{i=1}^N \left(\log(\sqrt{2\pi\sigma^2}) + \frac{(y_i - wx_i - b)^2}{2\sigma^2}\right) \end{aligned}$$

Note: σ is a constant.

1-Variable Linear Regression

$$\text{Minimize } E = \sum_{i=1}^N (y_i - wx_i - b)^2$$

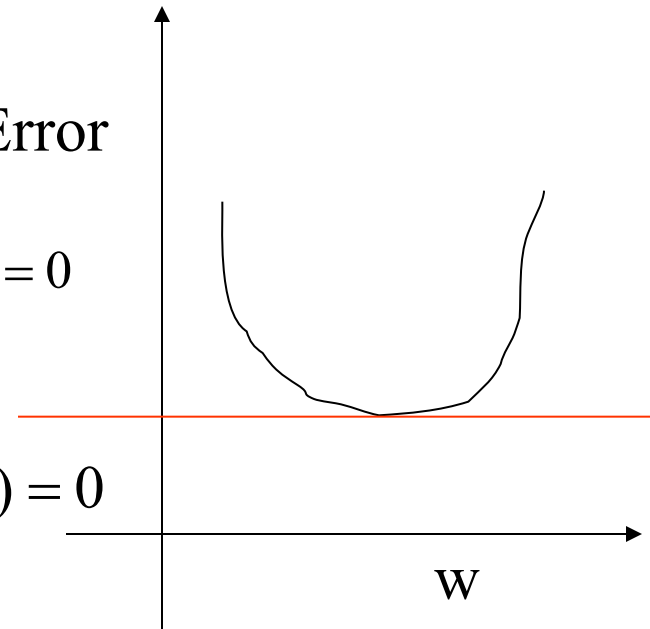
Error

$$\frac{\partial E}{\partial w} = \sum_{i=1}^N 2(y_i - wx_i - b) * (-x_i) = \sum_{i=1}^N 2(-y_i x_i + wx_i^2 + bx_i) = 0$$

$$\frac{\partial E}{\partial b} = \sum_{i=1}^N 2(y_i - wx_i - b) * (-1) = \sum_{i=1}^N 2(-y_i + wx_i + b) = 0$$

$$w = \frac{\sum_{i=1}^N x_i y_i - N \bar{x} \bar{y}}{\sum_{i=1}^N x_i^2 - N \bar{x} \bar{x}}$$

$$b = \frac{\sum_{i=1}^N (y_i - wx_i)}{N}$$



Multivariate Linear Regression

- How about multiple predictors: (x_1, x_2, \dots, x_d) .
- $y = w_0 + w_1x_1 + w_2x_2 + \dots + w_dx_d + \varepsilon$
- For multiple data points, each data point is represented as (y_i, x_i) , x_i consists of d predictors $(x_{i1}, x_{i2}, \dots, x_{id})$.
- $y_i = w_0 + w_1x_{i1} + w_2x_{i2} + \dots + w_dx_{id} + \varepsilon$

A Motivating Example

- Each day you get lunch at the cafeteria.
 - Your diet consists of fish, chips, and beer.
 - You get several portions of each
- The cashier only tells you the total price of the meal
 - After several days, you should be able to figure out the price of each portion.
- Each meal price gives a linear constraint on the prices of the portions:

$$price = x_{fish} w_{fish} + x_{chips} w_{chips} + x_{beer} w_{beer}$$

Matrix Representation

n data points, d dimension

$$\begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & \dots & x_{1d} \\ 1 & x_{21} & \dots & x_{2d} \\ \dots & \cdot & \dots & \cdot \\ 1 & x_{n1} & \dots & x_{nd} \end{pmatrix} \times \begin{pmatrix} w_0 \\ w_1 \\ \dots \\ w_d \end{pmatrix} + \varepsilon$$

$n \times 1$

$n \times (d+1)$

$(d+1) \times 1$

Matrix Representation: $\mathbf{Y} = \mathbf{XW} + \varepsilon$

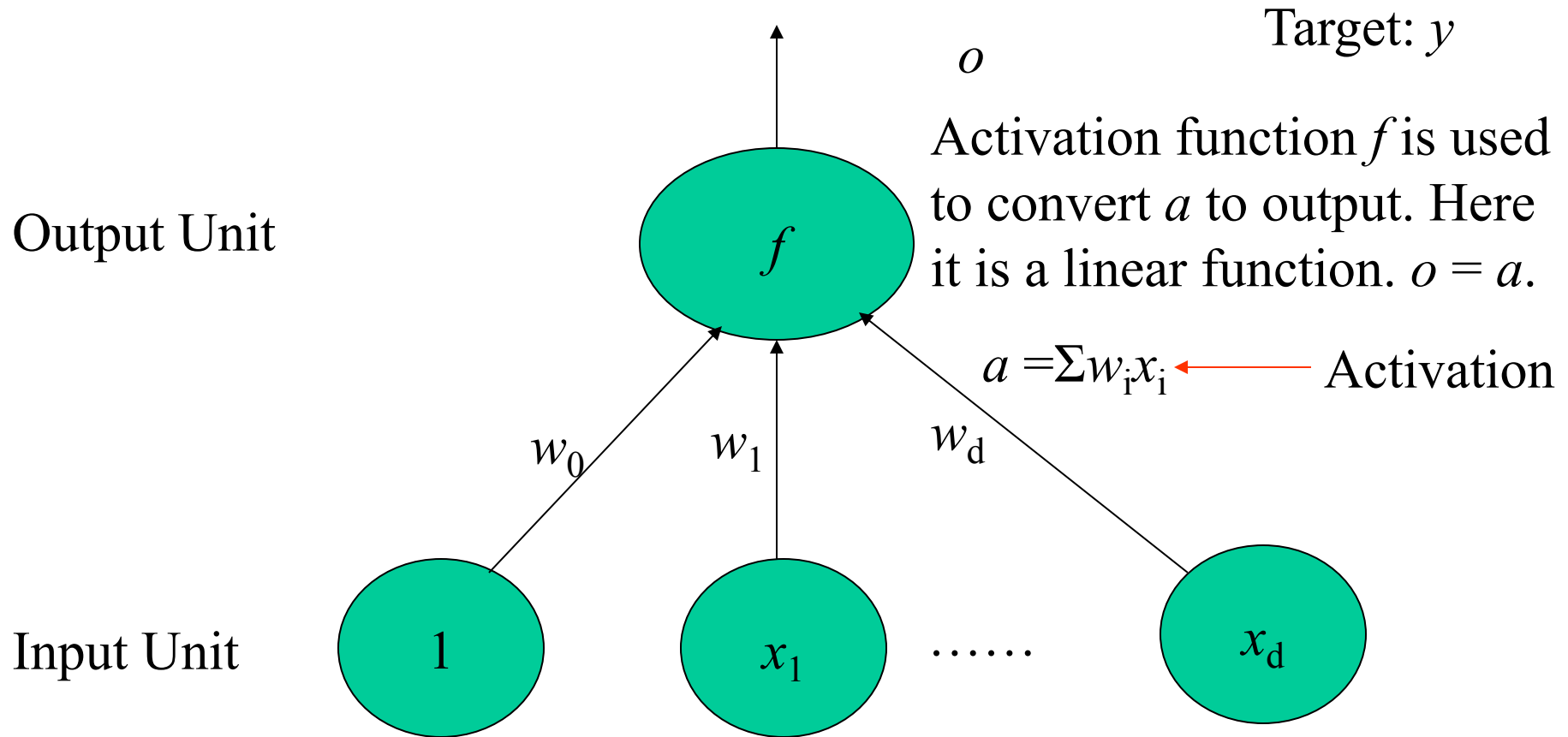
Multivariate Linear Regression

- Goal: minimize square error = $(Y - XW)^T(Y - XW) = Y^T Y - 2X^T W Y + W^T X^T X W$
- Derivative: $-2X^T Y + 2X^T X W = 0$
- $W = (X^T X)^{-1} X^T Y$
- Thus, we can solve linear regression using matrix inversion, transpose, and multiplication.

Difficulty and Generalization

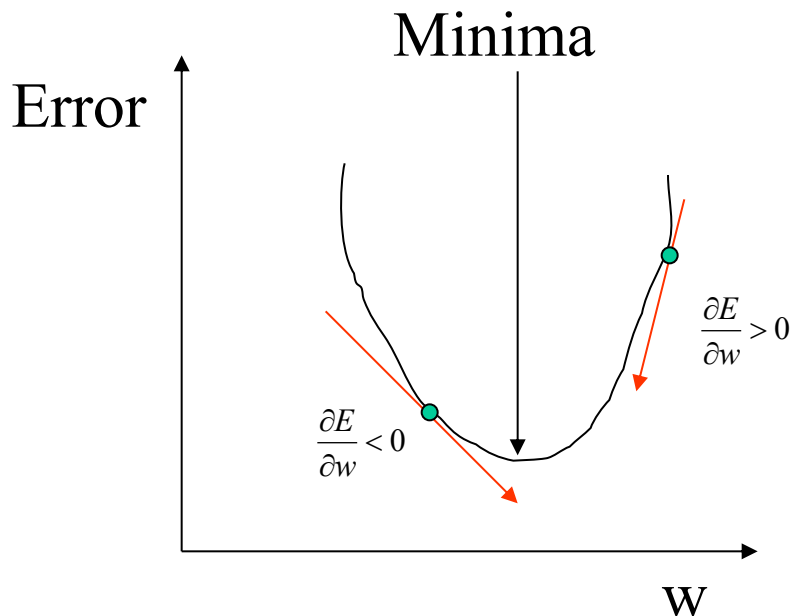
- Numerical computation issue. (a lot data points. Matrix inversion is impossible.)
- Singular matrix (determinant is zero) : no inversion
- How to handle non-linear data?
- Turns out neural network and its *iterative learning algorithm* can address this problem.

Graphical Representation: One Layer Neural Network for Regression



Gradient Descent Algorithm

- For a data $x = (x_1, x_2, \dots, x_d)$, error $E = (y - o)^2 = (y - w_0x_0 - w_1x_1 - \dots - w_dx_d)^2$
- Partial derivative: $\nabla E|_{w_i} = \frac{\partial E}{\partial w_i} = 2(y - o) \frac{\partial o}{\partial w_i} = 2(y - o)(-x_i) = -2(y - o)x_i$



Update rule:

$$w_i^{(t+1)} = w_i^{(t)} + \eta(y - o)x_i$$

Famous Delta Rule

Algorithm of One-Layer Regression Neural Network

- Initialize weights w (small random numbers)

- **Repeat**

Present a data point $x = (x_1, x_2, \dots, x_d)$ to the network and compute output o .

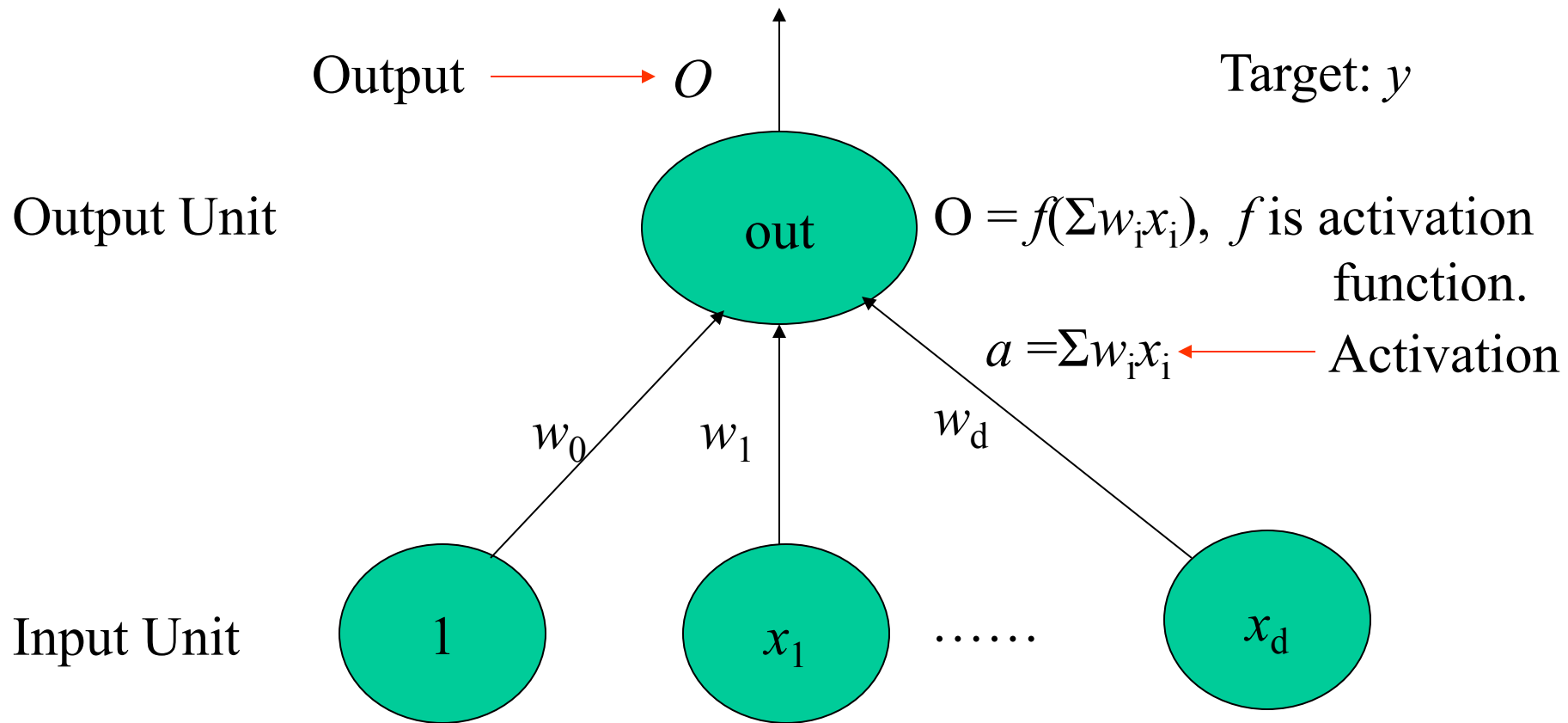
if $y > o$, add ηx_i to w_i .

if $y < o$, add $-\eta x_i$ to w_i .

- **Until** $\Sigma(y_k - o_k)^2$ is zero or below a threshold or reaches the predefined number of iterations.

Comments: online learning: update weight for every x . batch learning: update weight every batch of x (i.e. $\Sigma \eta x_i$).

Graphical Representation: One Layer Neural Network for Regression

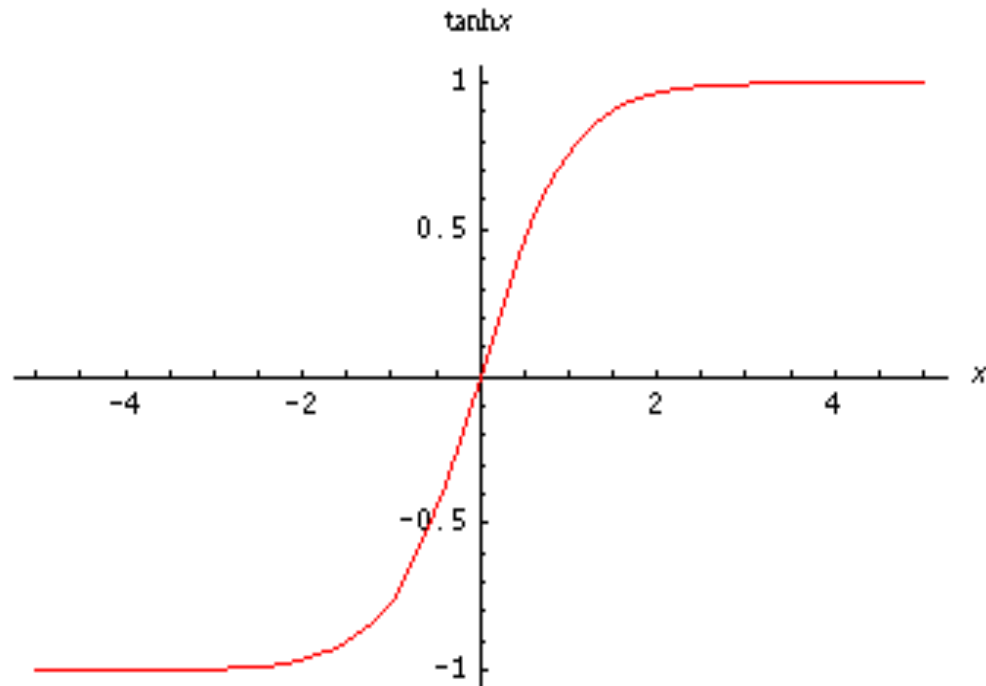


What about Hyperbolic Tanh Function for Output Unit

- Can we use activation function other than linear function?
- For instance, if we want to limit the output to be in $[-1, +1]$, we can use hyperbolic Tanh function:

$$\frac{e^x - e^{-x}}{e^x + e^{-x}}$$

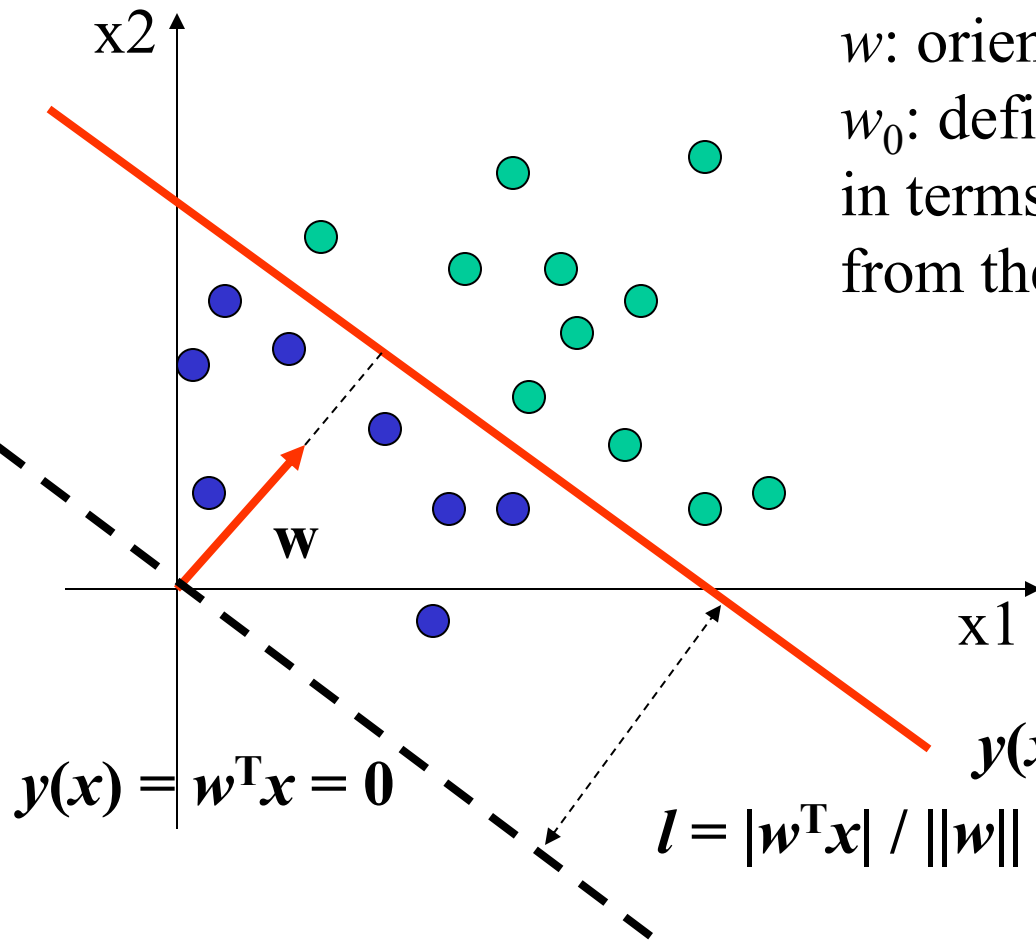
- The only thing to change is to use the new gradient.



Two-Category Classification

- Two classes: C_1 and C_2 .
- Input feature vector: x .
- Define a discriminant function $y(x)$ such that x is assigned to C_1 if $y(x) > 0$ and to class C_2 if $y(x) < 0$.
- Linear discriminant function: $y(x) = w^T x + w_0 = \bar{w}^T \bar{x}$, where $\bar{x} = (1, x)$.
- w : weight vector, w_0 : bias.

A Linear Decision Boundary in 2-D Input Space

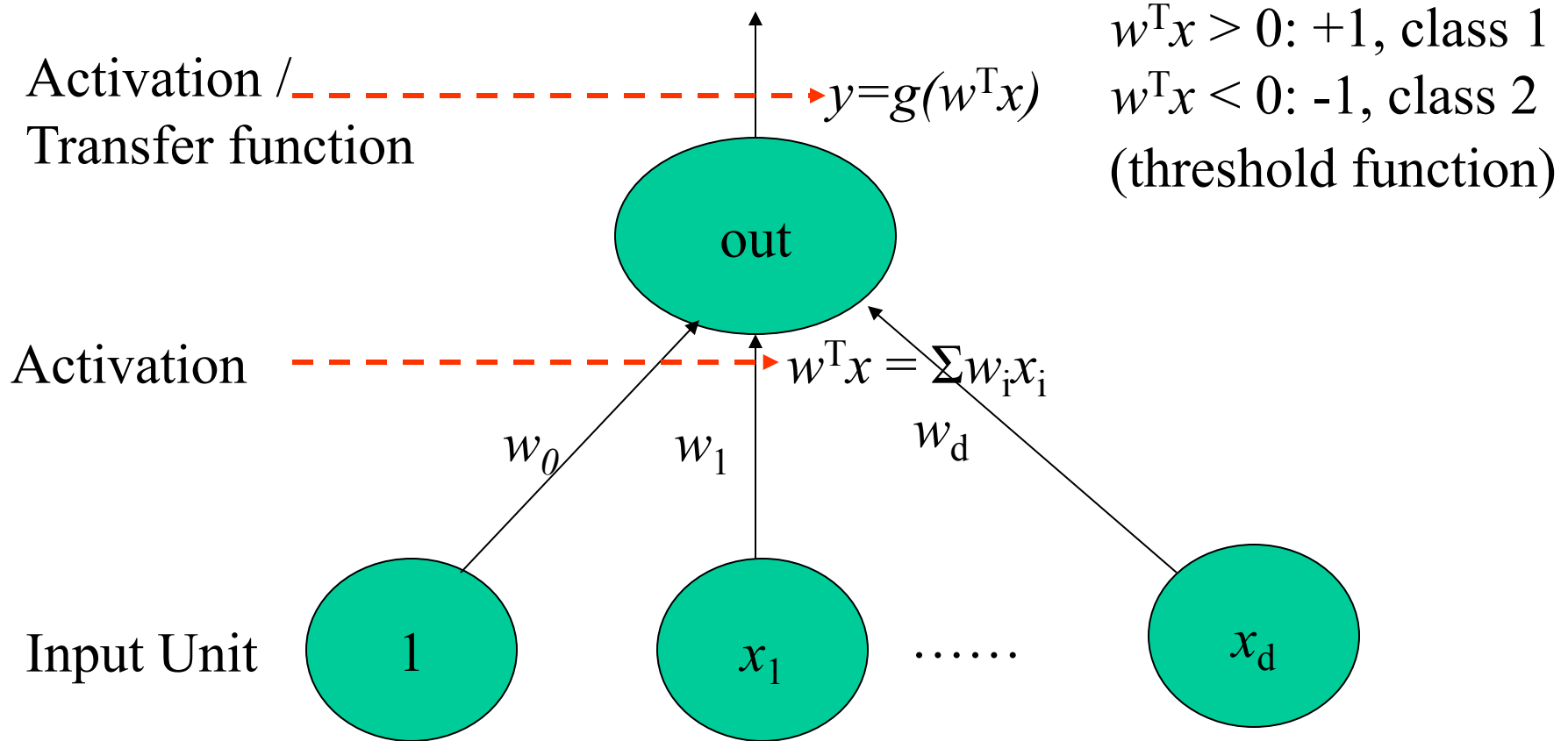


w : orientation of decision boundary
 w_0 : defines the position of the plan
in terms of its perpendicular distance
from the origin.

$$y(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0 = 0$$

$$l = |\mathbf{w}^T \mathbf{x}| / \|\mathbf{w}\| = w_0 / \|\mathbf{w}\|$$

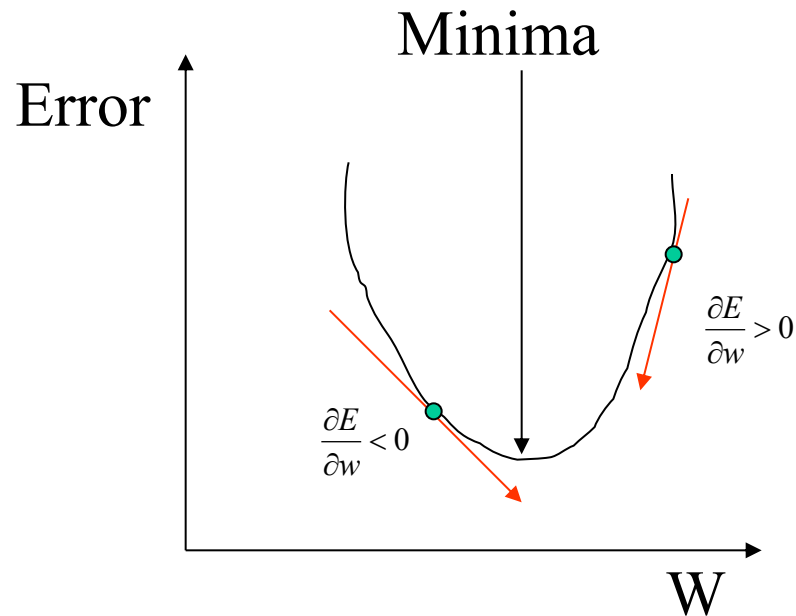
Graphical Representation: Perceptron, One-Layer Classification Neural Network



Perceptron Criterion

- Minimize classification error
- Input data (vector): x^1, x^2, \dots, x^N and corresponding target value t^1, t^2, \dots, t^N .
- Goal: for all x in C_1 ($t = 1$), $w^T x > 0$, for all x in C_2 ($t = -1$), $w^T x < 0$. Or for all x : $w^T x t > 0$.
- Error (loss): $E^{\text{perc}}(w) = \sum_{x^n \in M} w^T x^n t^n$. M is the set of misclassified data points.

Gradient Descent



For each misclassified data point, adjust weight as follows:

$$w = w - \frac{\partial E}{\partial w} \times \eta = w + \eta x^n t^n$$

Perceptron Algorithm

- Initialize weight w
- **Repeat**
For each data point (x^n, t^n)
Classify each data point using current w .
If $w^T x^n t^n > 0$ (correct), do nothing
If $w^T x^n t^n < 0$ (wrong), $w^{new} = w + \eta x^n t^n$
 $w = w^{new}$
- **Until** w is not changed (all the data will be separated correctly, if data is linearly separable) or error is below a threshold.

Perceptron Convergence Theorem

- For any data set which is linearly separable, the algorithm is guaranteed to find a solution in a finite number of steps (Rosenblatt, 1962; Block 1962; Nilsson, 1965; Minsky and Papert 1969; Duda and Hart, 1973; Hand, 1981; Arbib, 1987; Hertz et al., 1991)

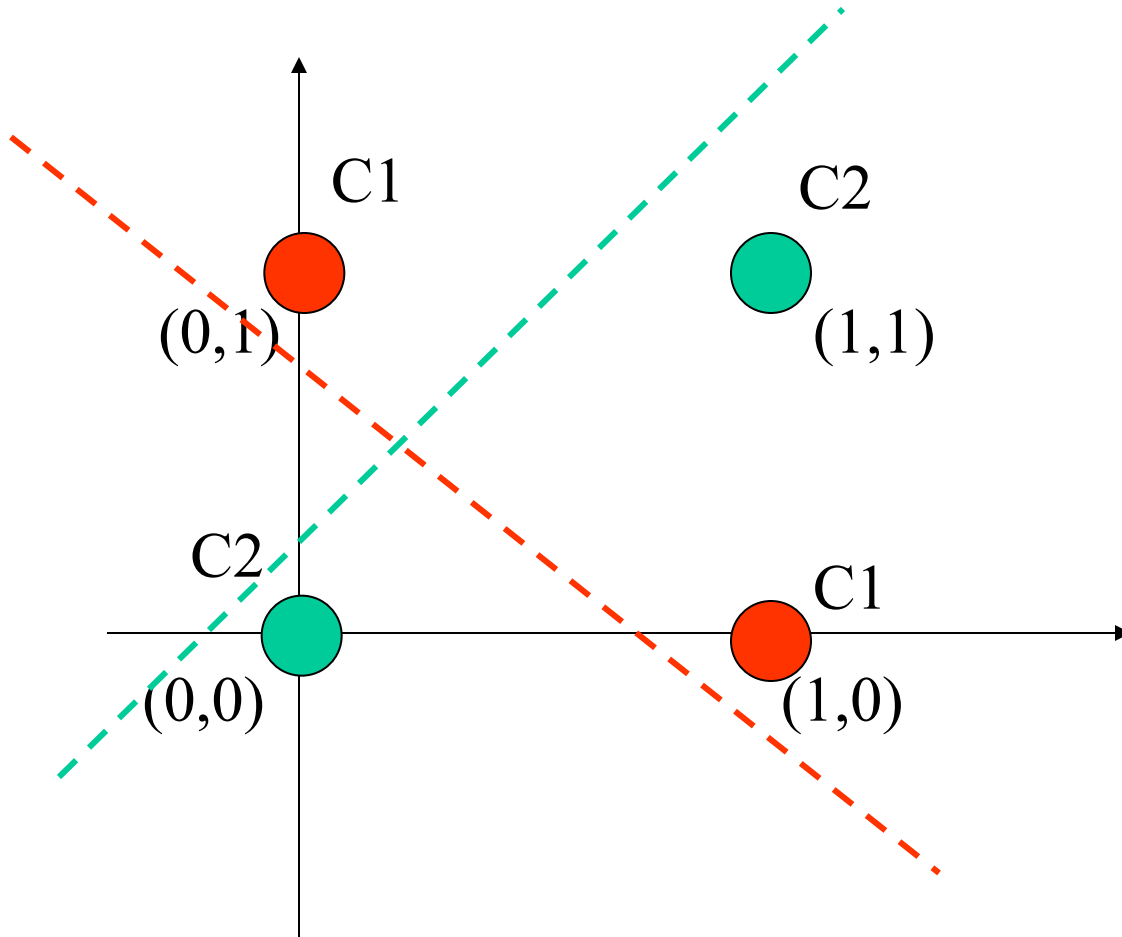
Perceptron Demo

- [https://www.youtube.com/watch?v=vGwemZ
hPlsA](https://www.youtube.com/watch?v=vGwemZ
hPlsA)

Limitation of the Perceptron

- Can't not separate non-linear data completely.
- Or can't not fit non-linear data well.
- Two directions to attack the problem: (1) extend to multi-layer neural network (2) map data into high dimension (SVM approach)

Exclusive OR Problem



Perceptron (or one-layer neural network) can not learn a function to separate the two classes perfectly.

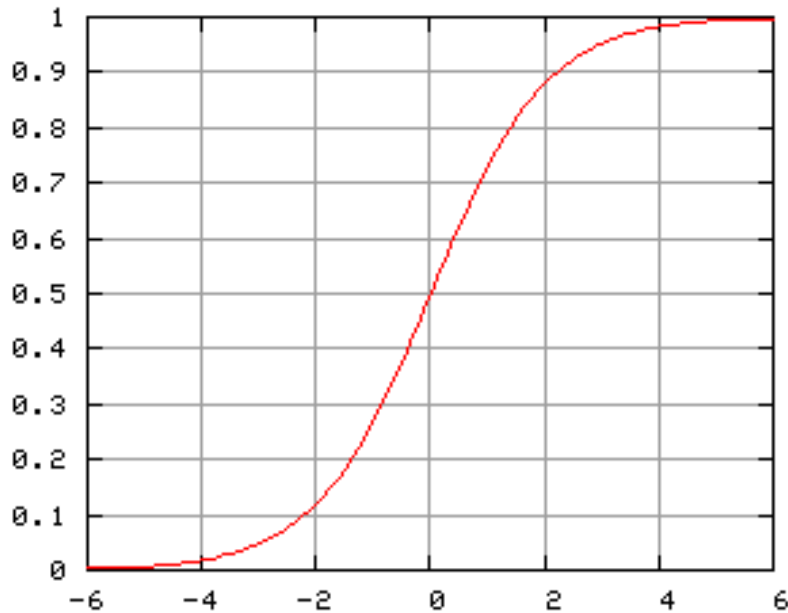
Logistic Regression

- Estimate posterior distribution: $P(C_1|x)$
- Dose – response estimation: in bioassay, the relation between dose level and death rate $P(\text{death} | x)$.
- We can not use 0/1 hard classification.
- We can not use unconstrained linear regression because $P(\text{death} | x)$ must be in $[0,1]$?

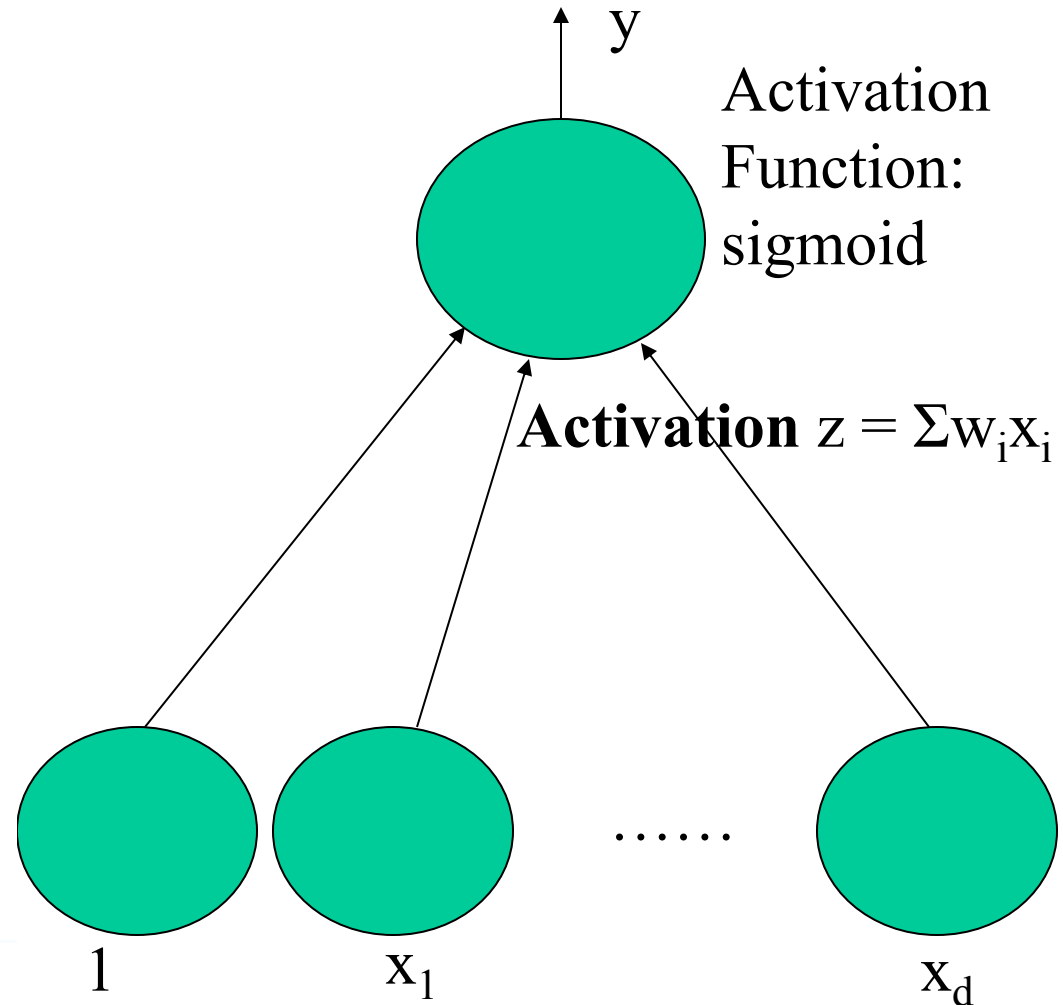
Logistic Regression and One Layer Neural Network With Sigmoid Function.

$$P(\text{death} | x) = \frac{1}{1 + e^{-wx}}$$

(Sigmoid function)



Target: t (0 or 1)



How to Adjust Weights?

- Minimize error $E=(t-y)^2$. For simplicity, we derive the formula for one data point. For multiple data points, just add the gradients together.

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_i} = -2(t-y)y(1-y)x_i$$

Notice:
$$\frac{\partial y}{\partial z} = \frac{\partial\left(\frac{1}{1+e^{-z}}\right)}{\partial z} = \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}}\right) = y(1-y)$$

Error / Loss Function and Learning

- Least Square
- Maximum likelihood: output y is the probability of being in C_1 ($t=1$). $1-y$ is the probability of being in C_2 . So what is probability of $P(t|x) = y^t(1-y)^{1-t}$.
- Maximum likelihood is equivalent to minimize negative log likelihood:
$$E = -\log P(t|x) = -t \log y - (1-t) \log(1-y).$$
 (cross / relative entropy)

How to Adjust Weights?

- Minimize error $E = -t \log y - (1-t) \log(1-y)$. For simplicity, we derive the formula for one data point. For multiple data points, just add the gradients together.

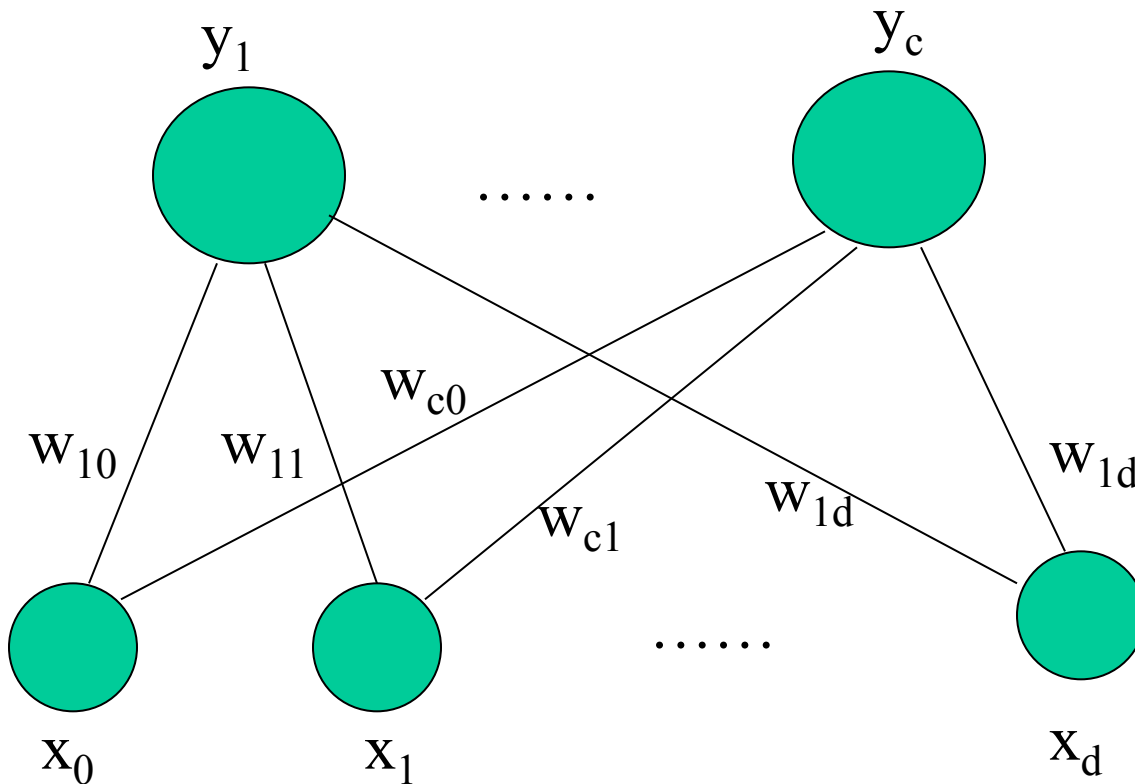
$$\frac{\partial E}{\partial y} = -\frac{t}{y} - \frac{1-t}{1-y}(-1) = -\frac{t}{y} + \frac{t-1}{1-y} = \frac{y-t}{y(1-y)}$$

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_i} = \frac{y-t}{y(1-y)} y(1-y)x_i = (y-t)x_i$$

Update rule: $w_i^{(t+1)} = w_i^t + \eta(t - y)x_i$

Multi-Class Logistic Regression

- Transfer (or activation) function is normalized exponentials (or **soft max**)



$$y_i = \frac{e^{a_i}}{\sum_{j=1}^c e^{a_j}}$$

Activation Function

$$a_i = \sum_{j=1}^d w_{ij} * x_j$$

Activation to Node O_i

How to learn this network? Once again, gradient descent.

Questions?

- Is logistic regression a linear regression?
- Can logistic regression handle non-linearly separable data?
- How to introduce non-linearity?

Neural Network Approach

- Multi-Layer Perceptrons
- In addition to input nodes and output nodes, some hidden nodes between input / output nodes are introduced.
- Use hidden units to learn internal features to represent data. Hidden nodes can learn internal representation of data that are not explicit in the input features.
- Transfer function of hidden units are non-linear function

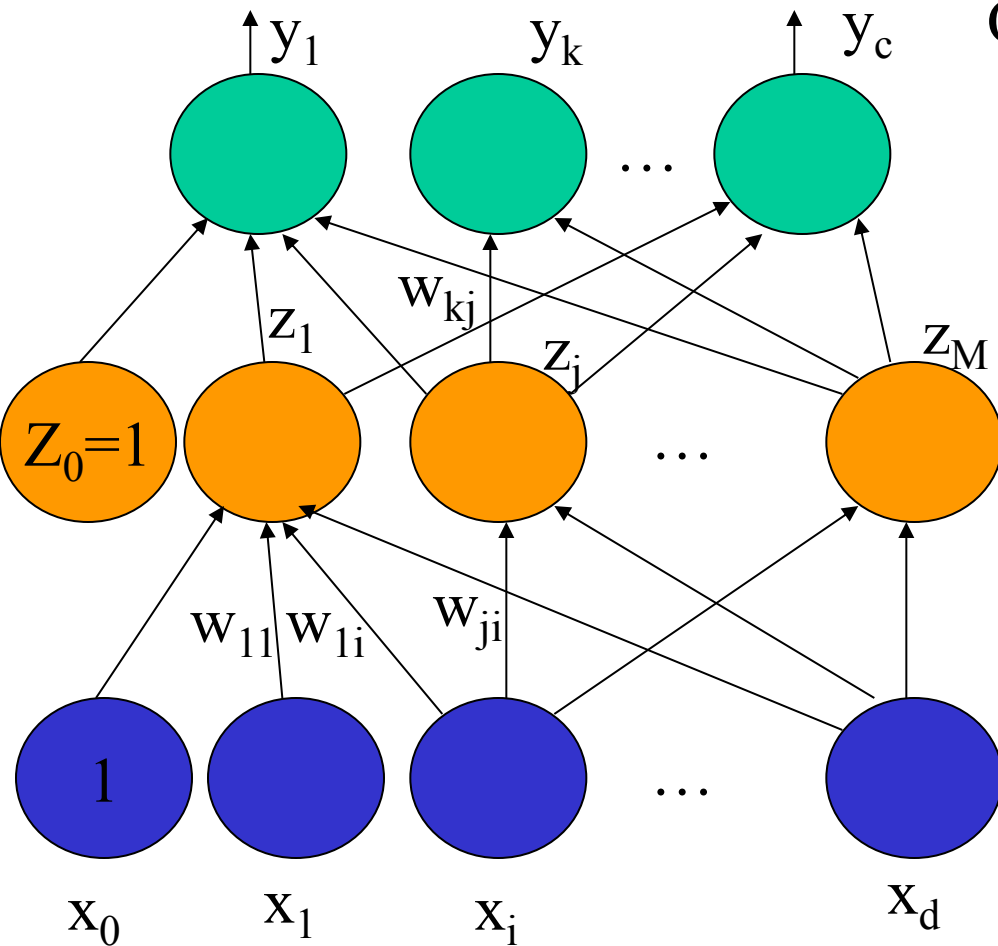
Multi-Layer Perceptron

- Connections go from lower layer to higher layer. (usually from input layer to hidden layer, to output layer)
- Connection between input/hidden nodes, input/output nodes, hidden/hidden nodes, hidden/output nodes are arbitrary as long as there is no loop (must be feed-forward).
- However, for simplicity, we usually only allow connection from input nodes to hidden nodes and from hidden nodes to output nodes. The connections within a layer are disallowed.

Multi-Layer Perceptron

- Two-layer neural network (one hidden and one output) with non-linear activation function is a **universal function approximator** (see Baldi and Brunak 2001 or Bishop 96 for the proof), i.e. it can approximate any numeric function with arbitrary precision given a set of appropriate weights and hidden units.
- In early days, people usually used two-layer (or three-layer if you count the input as one layer) neural network. Increasing the number of layers was occasionally helpful.
- Later expanded into deep learning with many layers!!!

Two-Layer Neural Network



Output

Activation function: f (linear, sigmoid, softmax)

Activation of unit a_k :

$$\sum_{j=0}^M w_{kj} z_j$$

Activation function: g (linear, tanh, sigmoid)

Activation of unit a_j :

$$\sum_{i=0}^d w_{ji} x_i$$

$$y_k = f\left(\sum_{j=0}^M w_{kj} \times g\left(\sum_{i=0}^d w_{ji} x_i\right)\right)$$

Adjust Weights by Training

- How to adjust weights?
- Adjust weights using known examples (training data) $(x_1, x_2, x_3, \dots, x_d, t)$.
- Try to adjust weights so that the difference between the output of the neural network y and t (target) becomes smaller and smaller.
- Goal is to minimize Error (difference) as we did for one layer neural network

Adjust Weights using Gradient Descent (Back-Propagation)

Known:

Data: $(x_1, x_2, x_3, \dots, x_n)$ target t .

Unknown weights w :

w_{11}, w_{12}, \dots

Randomly initialize weights

Repeat

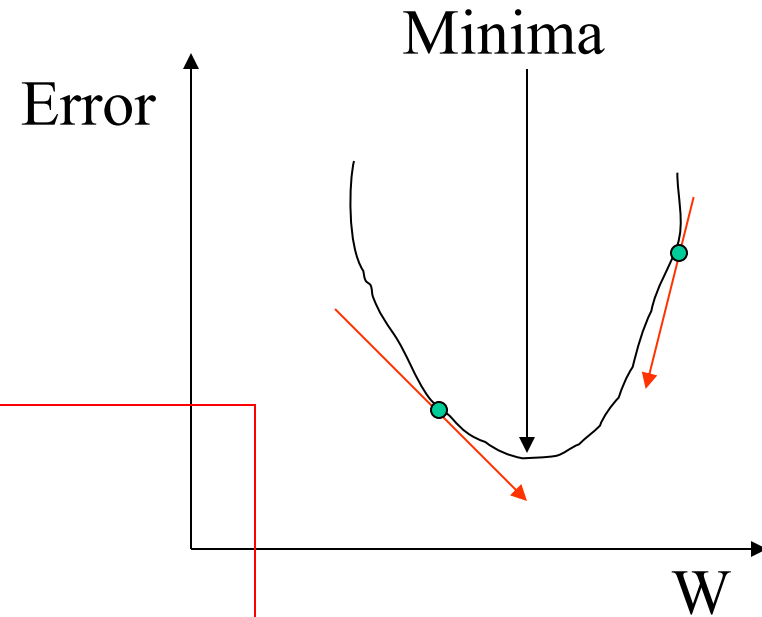
for each example, compute output y

calculate error $E = (y-t)^2$

compute the derivative of E over w : $dw = \frac{\partial E}{\partial w}$

$w_{\text{new}} = w_{\text{prev}} - \eta * dw$

Until error doesn't decrease or max num of iterations



Note: η is learning rate or step size.

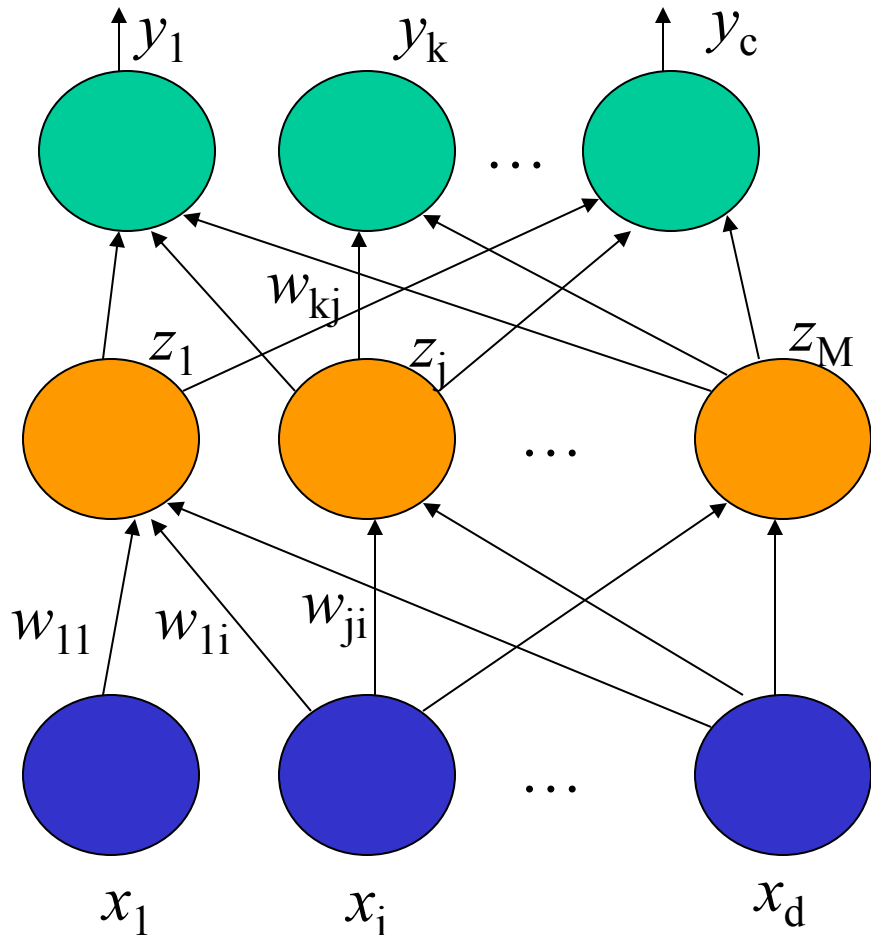
Insights

- We know how to compute the derivative of one layer neural network? How to change weights between input layer and hidden layer?
- Should we compute the derivative of each w separately or we can reuse intermediate results? We will have an efficient back-propagation algorithm.
- We will derive learning for one data example. For multiple examples, we can simply add the derivatives from them for a weight parameter together.

Neural Network Learning: Two Processes

- Forward propagation: present an example (data) into neural network. Compute activation into units and output from units.
- Backward propagation: propagate error back from output layer to the input layer and compute derivatives (or gradients).

Forward Propagation



Output

Activation function: f (linear, sigmoid, softmax)

Activation of unit a_k : $\sum_{j=1}^M w_{kj} z_j$

Activation function: g (linear, tanh, sigmoid)

Activation of unit a_j :

$$\sum_{i=1}^d w_{ji} x_i$$

Time complexity?
 $O(dM + MC) = O(W)$

Neural Network Calculations are essentially matrix operations

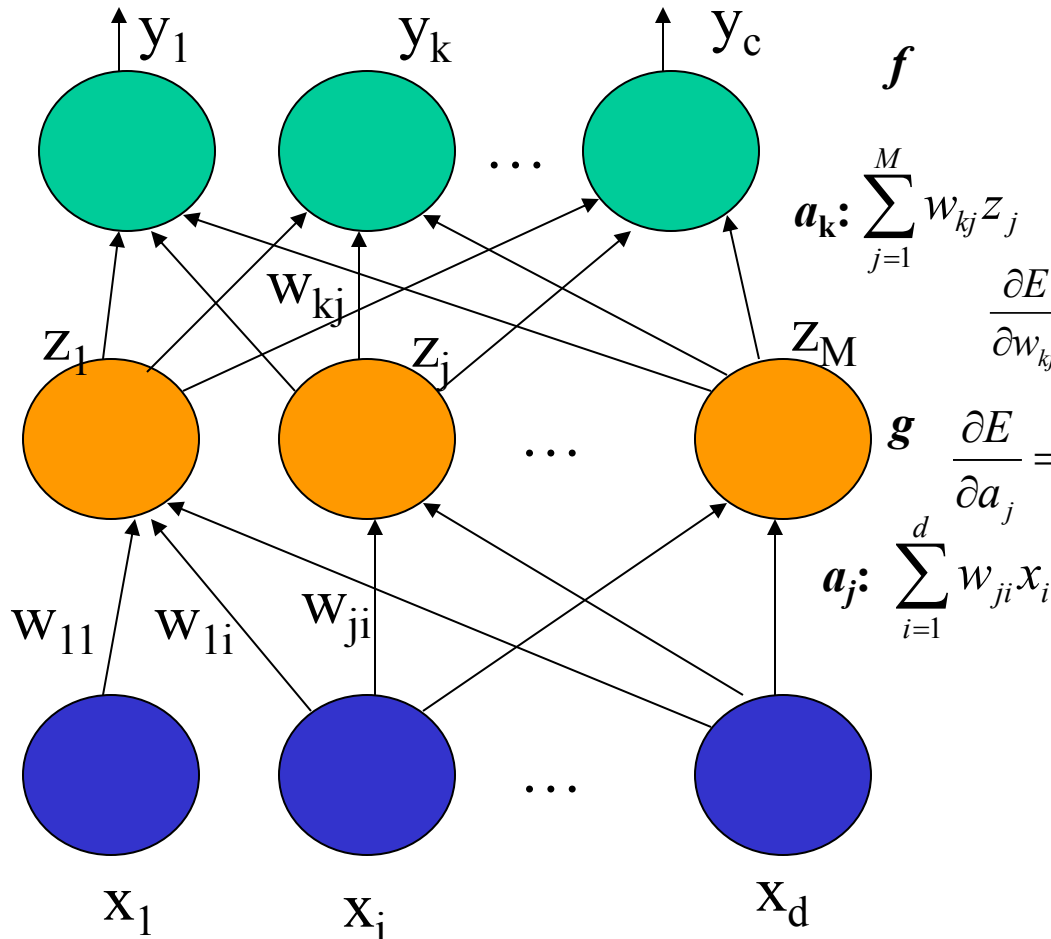
Layer 1: $XW_1 = A_1$

Layer 2: $f(A_1) = Z$

**Layer 3: $ZW_2 = A_2$
 $Y = f(A_2)$**



Backward Propagation



$$E = \frac{1}{2} \sum_{k=1}^c (y_k - t_k)^2$$

$$\frac{\partial E}{\partial y_k} = y_k - t_k$$

$$\frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial a_k} = (y_k - t_k) f'(a_k) = \delta_k$$

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial w_{kj}} = \delta_k z_j$$

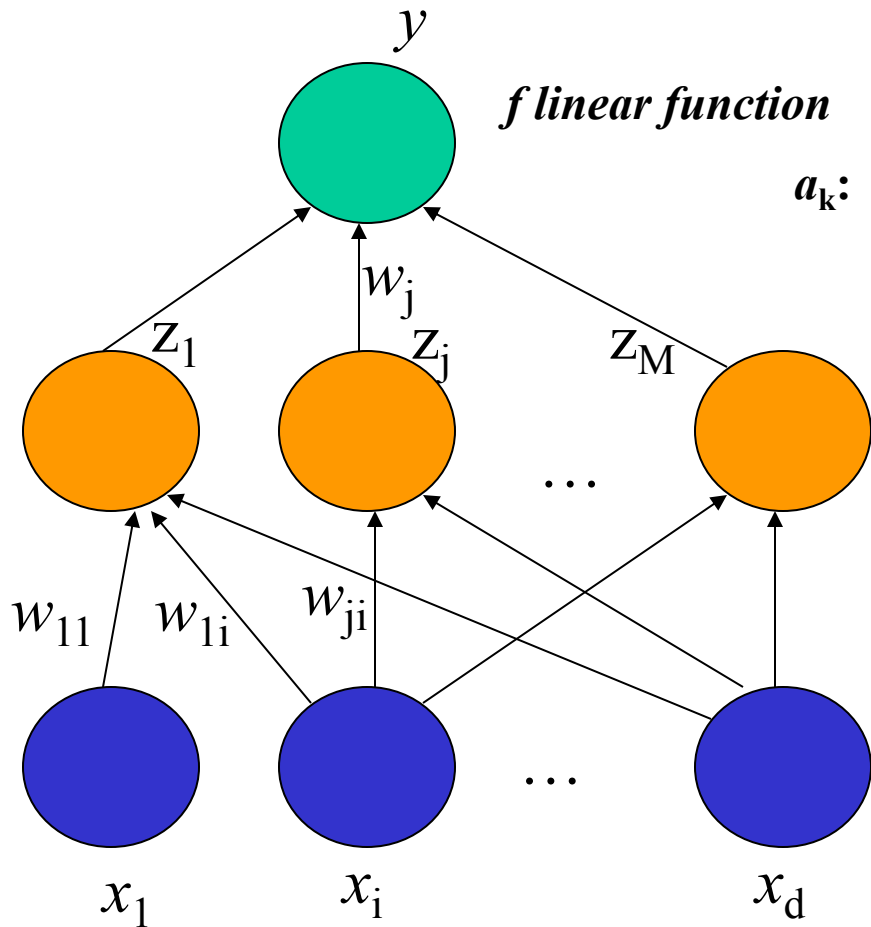
$$\frac{\partial E}{\partial a_j} = \sum_{k=1}^c \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial a_k} \frac{\partial a_k}{\partial z_j} \frac{\partial z_j}{\partial a_j} = \sum_{k=1}^c \delta_k w_{kj} g'(a_j) = \delta_j$$

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} = \delta_j x_i$$

If no back-propagation, time complexity is: $(MdC + CM)$

Time complexity?
 $O(CM + Md) = O(W)$

Example



$$E = \frac{1}{2} (y - t)^2$$

$$\delta = \frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial a_k} = (y - t)$$

$$g \text{ is sigmoid: } \frac{\partial E}{\partial w_j} = \delta z_j$$

$$\delta_j = \delta w_j g'(a_j) = (y - t) w_j z_j (1 - z_j)$$

$$\frac{\partial E}{\partial w_{ji}} = \delta_j x_i = (y - t) w_j z_j (1 - z_j) x_i$$

Algorithm

- **Initialize** weights w

- **Repeat**

For each data point x , do the following:

Forward propagation: compute outputs and activations

Backward propagation: compute errors for each output units and hidden units. Compute gradient for each weight.

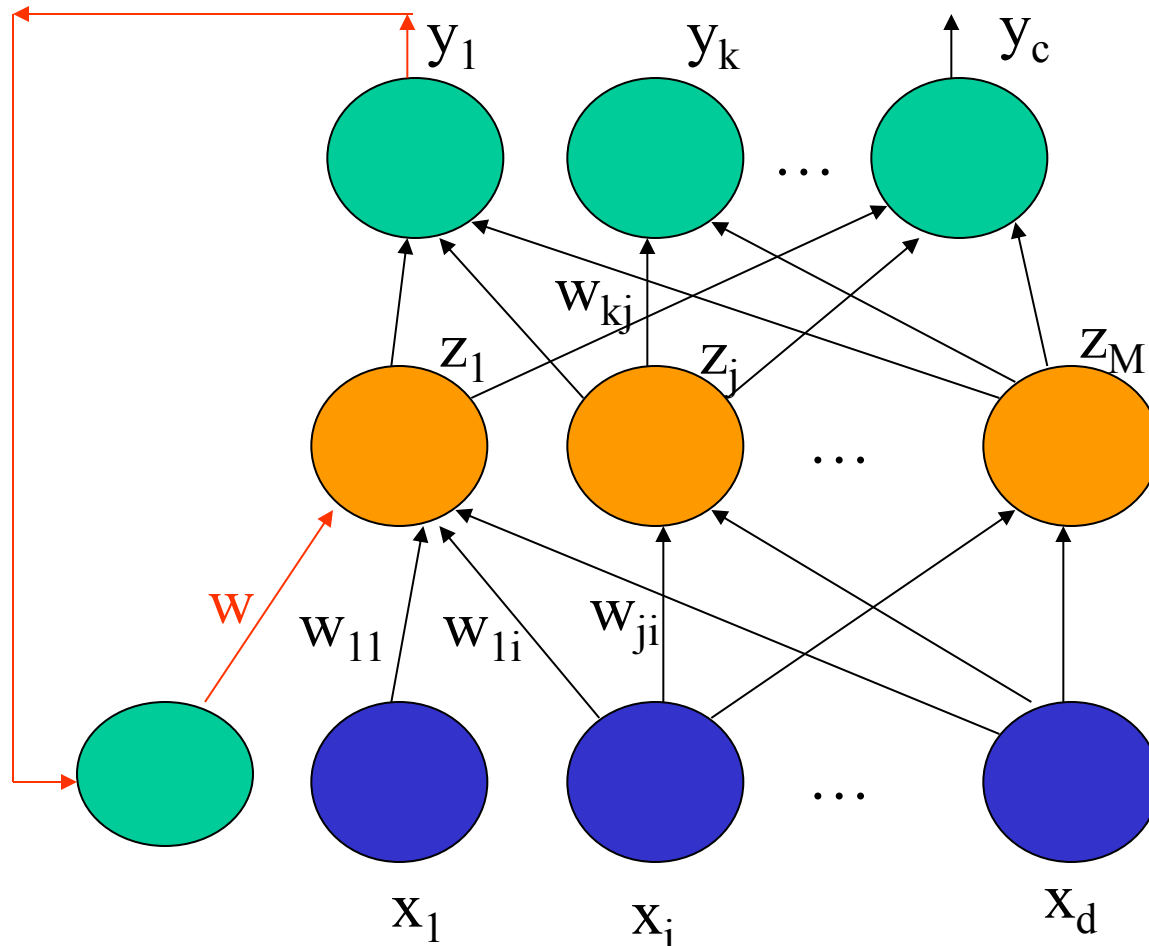
Update weight $w = w - \eta (\partial E / \partial w)$

- **Until** a number of iterations or errors drops below a threshold.

Implementation Issue

- What should we store?
- An input vector x of d dimensions
- A $M \times d$ matrix $\{w_{ji}\}$ for weights between input and hidden units
- An activation vector of M dimensions for hidden units
- An output vector of M dimensions for hidden units
- A $C \times M$ matrix $\{w_{kj}\}$ for weights between hidden and output units
- An activation vector of C dimensions for output units
- An output vector of C dimensions for output units
- An error vector of C dimensions for output units
- An error vector of M dimensions for hidden units

Recurrent Network



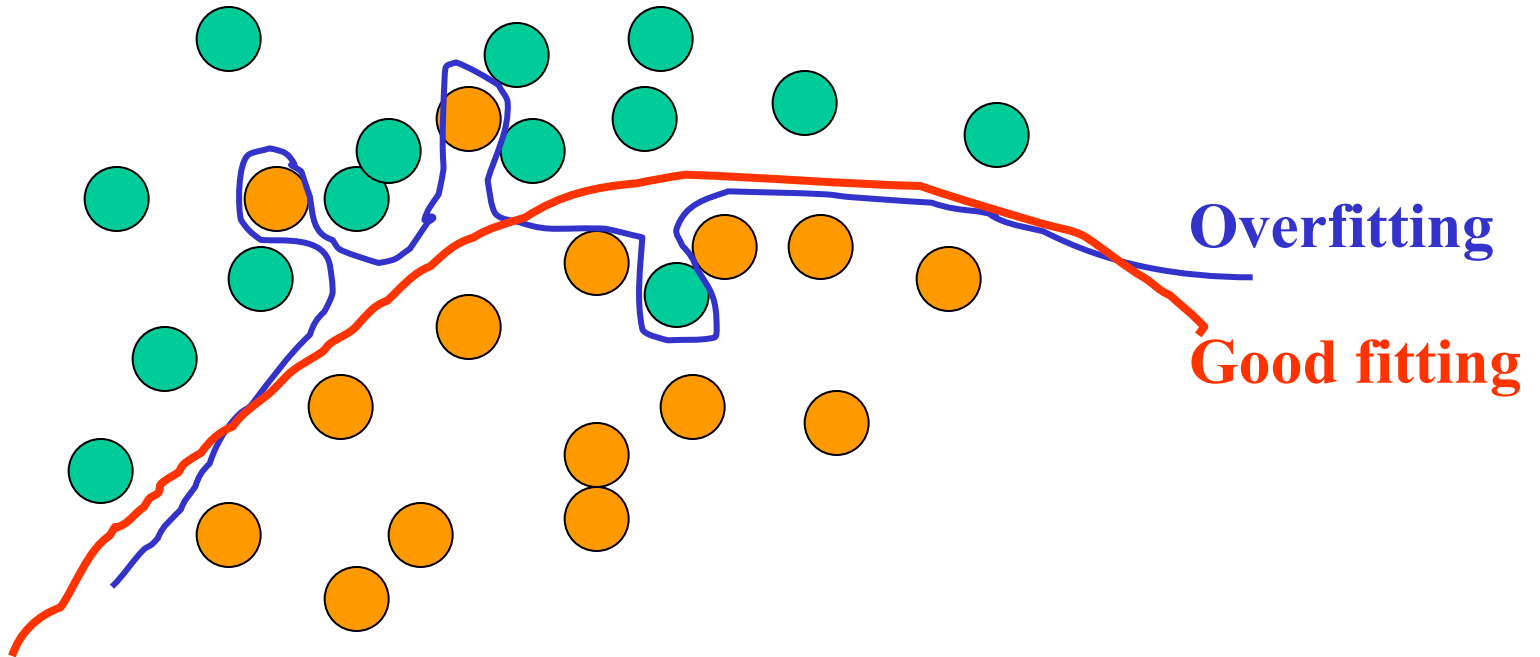
Forward:

At time 1: present $x_1, 0$
At time 2: present x_2, y_1
.....

Backward:

Time t : back-propagate
Time $t-1$: back-propagate with
Output errors and errors from previous step

Example of Overfitting and Good Fitting



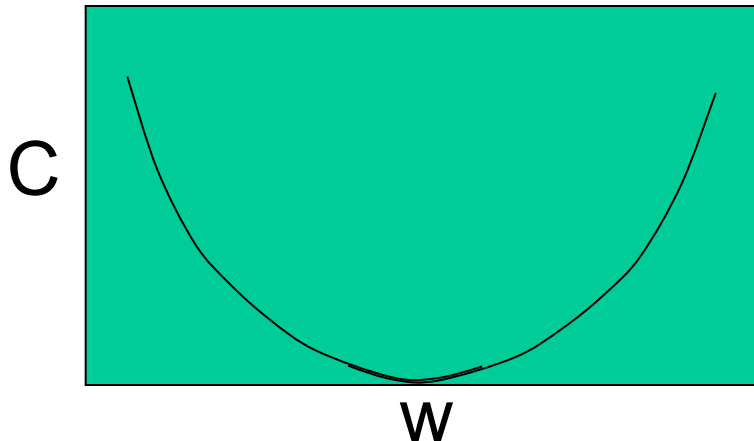
Overfitting function can not generalize well to unseen data.

Preventing Overfitting

- Use a model that has the right capacity:
 - enough to model the true regularities
 - not enough to also model the spurious regularities (assuming they are weaker).
- Standard ways to limit the capacity of a neural net:
 - Limit the number of hidden units.
 - Limit the size of the weights.
 - Stop the learning before it has time to overfit.

Limiting the Size of the Weights

- Weight-decay involves adding an extra term to the cost function that penalizes the squared weights.
 - Keeps weights small unless they have big error



$$C = E + \frac{\lambda}{2} \sum_i w_i^2$$

$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$$

when $\frac{\partial C}{\partial w_i} = 0$, $w_i = -\frac{1}{\lambda} \frac{\partial E}{\partial w_i}$

The Effect of Weight-Decay

- It prevents the network from using weights that it does not need.
 - This can often improve generalization a lot.
 - It helps to stop it from fitting the sampling error.
 - It makes a smoother model in which the output changes more slowly as the input changes.
- If the network has two very similar inputs it prefers to put half the weight on each rather than all the weight on one.



Deciding How Much to Restrict the Capacity

- How do we decide which limit to use and how strong to make the limit?
 - If we use the test data we get an unfair prediction of the error rate we would get on new test data.
 - Suppose we compared a set of models that gave random results, the best one on a particular dataset would do better than chance. But it won't do better than chance on another test set.
- So use a separate **validation set** to do model selection.

Using a Validation Set

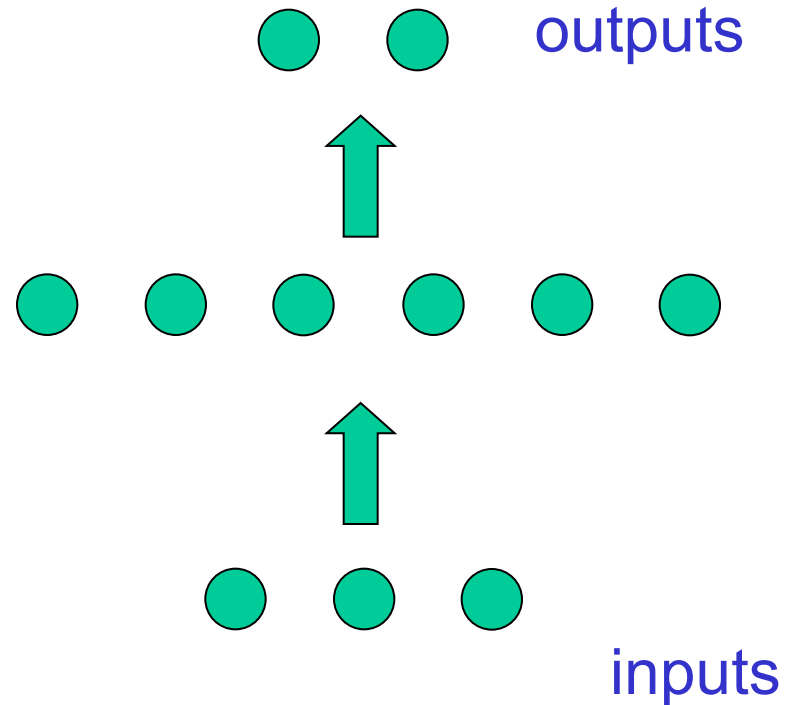
- Divide the total dataset into three subsets:
 - **Training data** is used for learning the parameters of the model.
 - **Validation data** is not used of learning but is used for deciding what type of model and what amount of regularization works best.
 - **Test data** is used to get a final, unbiased estimate of how well the network works. We expect this estimate to be worse than on the validation data.
- We could then re-divide the total dataset to get another unbiased estimate of the true error rate.

Preventing Overfitting by Early Stopping

- If we have lots of data and a big model, its very expensive to keep re-training it with different amounts of weight decay.
- It is much cheaper to start with very small weights and let them grow until the performance on the validation set starts getting worse (but don' t get fooled by noise!)
- The capacity of the model is limited because the weights have not had time to grow big.

Why Early Stopping Works

- When the weights are very small, every hidden unit is in its linear range.
 - So a net with a large layer of hidden units is linear.
 - It has no more capacity than a linear net in which the inputs are directly connected to the outputs!
- As the weights grow, the hidden units start using their non-linear ranges so the capacity grows.



Combining Networks

- When the amount of training data is limited, we need to avoid overfitting.
 - Averaging the predictions of many different networks is a good way to do this.
 - It works best if the networks are as different as possible.
 - Combining networks reduces variance
- If the data is really a mixture of several different “regimes” it is helpful to identify these regimes and use a separate, simple model for each regime.
 - We want to use the desired outputs to help cluster cases into regimes. Just clustering the inputs is not as efficient.

How the Combined Predictor Compares with the Individual Predictors

- On any one test case, some individual predictors will be better than the combined predictor.
 - But different individuals will be better on different cases.
- If the individual predictors **disagree** a lot, the combined predictor is typically better than all of the individual predictors when we average over test cases.
 - So how do we make the individual predictors disagree? (without making them much worse individually).

Ways to Make Predictors Differ

- Rely on the learning algorithm getting stuck in a different local optimum on each run.
 - A dubious hack unworthy of a true computer scientist (but definitely worth a try).
- Use lots of different kinds of models:
 - Different architectures
 - Different learning algorithms.
- Use different training data for each model:
 - **Bagging**: Resample (with replacement) from the training set: a,b,c,d,e -> a c c d d
 - **Boosting**: Fit models one at a time. Re-weight each training case by how badly it is predicted by the models already fitted.
 - This makes efficient use of computer time because it does not bother to “back-fit” models that were fitted earlier.

How to Speedup Learning?

The Error Surface for a Linear Neuron

- The error surface lies in a space with a horizontal axis for each weight and one vertical axis for the error.

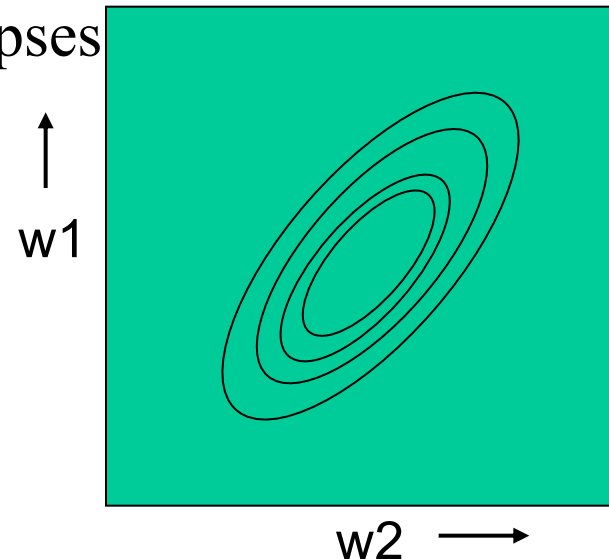
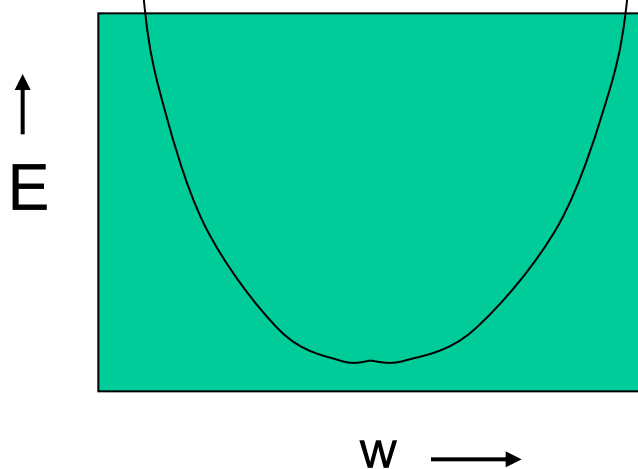
- It is a quadratic bowl.

- i.e. the height can be expressed as a function of the weights without using powers higher than 2. Quadratics have constant curvature (because the second derivative must be a constant)

- Vertical cross-sections are parabolas.

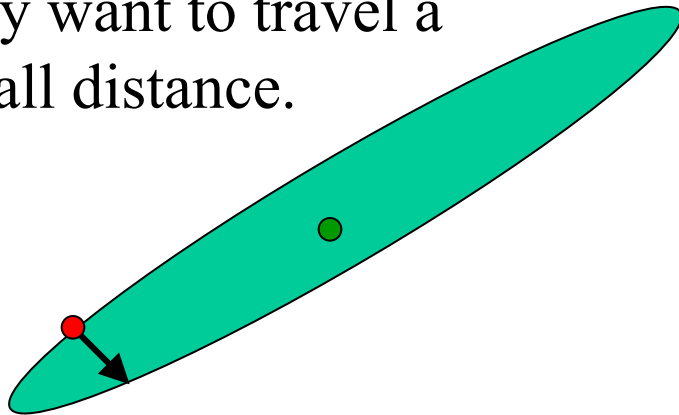
G. Hinton, 2006

- Horizontal cross-sections are ellipses



Convergence Speed

- The direction of steepest descent does not point at the minimum unless the ellipse is a circle.
 - The gradient is big in the direction in which we only want to travel a small distance.



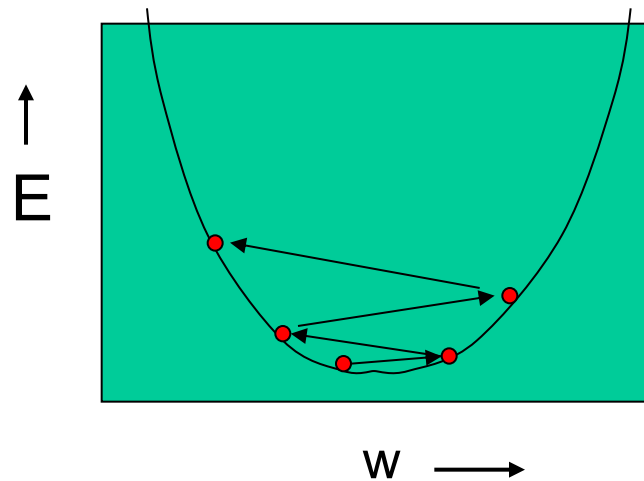
- The gradient is small in the direction in which we want to travel a large distance.

$$\Delta w_i = -\varepsilon \frac{\partial E}{\partial w_i}$$

This equation is sick.

How the Learning Goes Wrong

- If the learning rate is big, it sloshes to and fro across the ravine. If the rate is too big, this oscillation diverges.
- How can we move quickly in directions with small gradients without getting divergent oscillations in directions with big gradients?



Five Ways to Speed up Learning

- Use an adaptive global learning rate
 - Increase the rate slowly if its not diverging
 - Decrease the rate quickly if it starts diverging
- Use separate adaptive learning rate on each connection
 - Adjust using consistency of gradient on that weight axis
- Use momentum
 - Instead of using the gradient to change the **position** of the weight “particle”, use it to change the **velocity**.
- Use a stochastic estimate of the gradient from a few cases
 - This works very well on large, redundant datasets.

The Momentum Method

Imagine a ball on the error surface with velocity v .

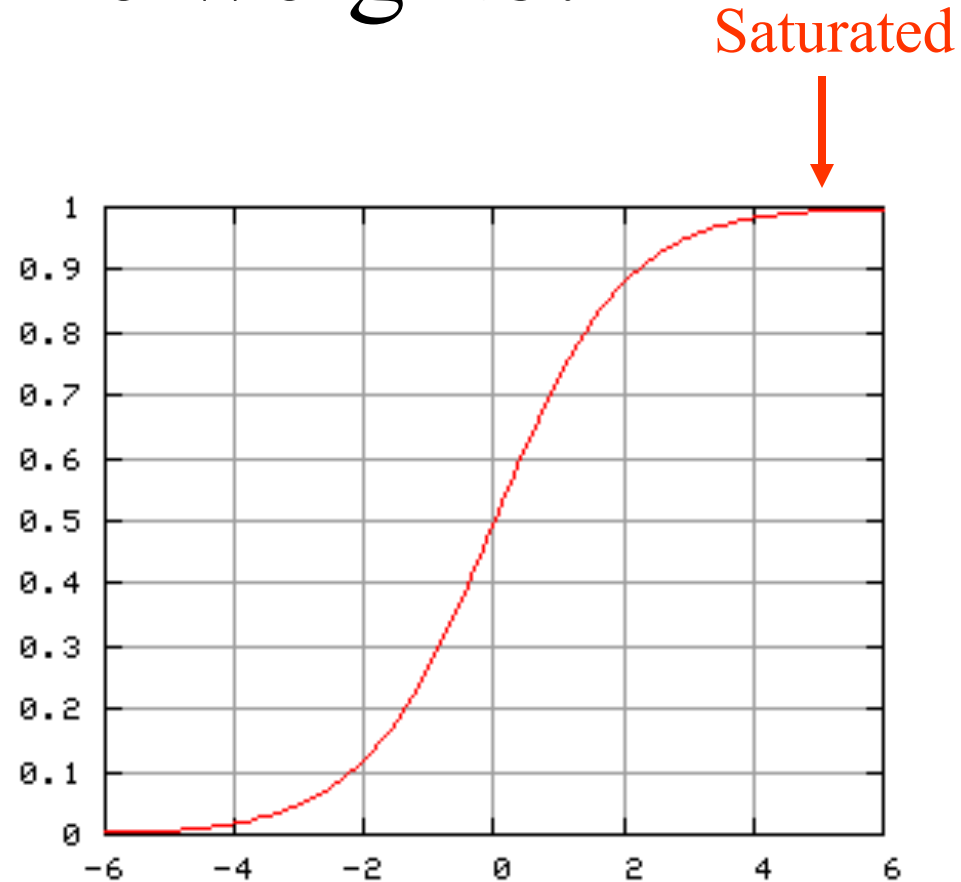
– It starts off by following the gradient, but once it has velocity, it no longer does steepest descent.

- It damps oscillations by combining gradients with opposite signs.
- It builds up speed in directions with a gentle but consistent gradient.

$$\begin{aligned}\Delta w(t) &= v(t) \\ &= \alpha \Delta w(t-1) - \varepsilon \frac{\partial E}{\partial w}(t)\end{aligned}$$

How to Initialize weights?

- Use small random numbers. For instance small numbers between $[-0.2, 0.2]$.
- Some numbers are positive and some are negative.
- Why are the initial weights should be small?



$$\frac{1}{1 + e^{-wx}}$$

Stochastic Gradient Descent

Ensemble, Dropout and Batch Normalization

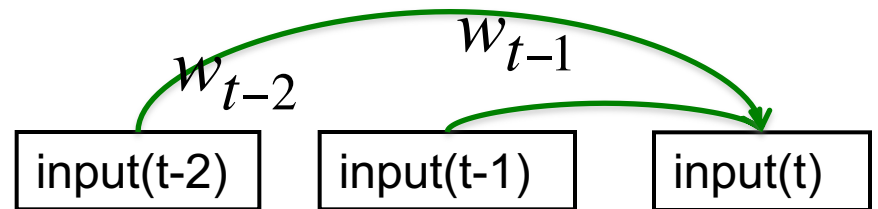
Recurrent Neural Networks

Getting targets when modeling sequences

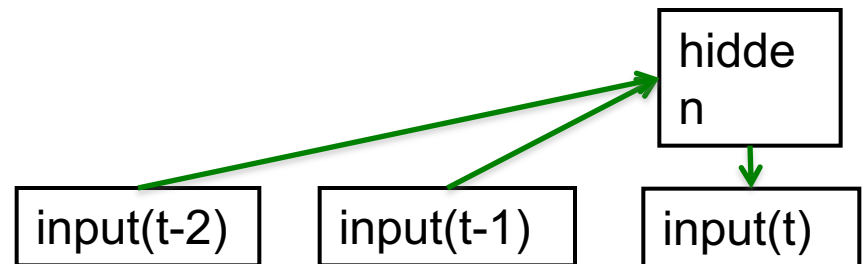
- When applying machine learning to sequences, we often want to turn an input sequence into an output sequence that lives in a different domain.
 - *E. g.* turn a sequence of sound pressures into a sequence of word identities.
- When there is no separate target sequence, we can get a teaching signal by trying to predict the next term in the input sequence.
 - The target output sequence is the input sequence with an advance of 1 step.
 - This seems much more natural than trying to predict one pixel in an image from the other pixels, or one patch of an image from the rest of the image.
 - For temporal sequences there is a natural order for the predictions.
- Predicting the next term in a sequence blurs the distinction between supervised and unsupervised learning.
 - It uses methods designed for supervised learning, but it doesn't require a separate teaching signal.

Memoryless models for sequences

- **Autoregressive models** Predict the next term in a sequence from a fixed number of previous terms using “delay taps”.



- **Feed-forward neural nets** These generalize autoregressive models by using one or more layers of non-linear hidden units.

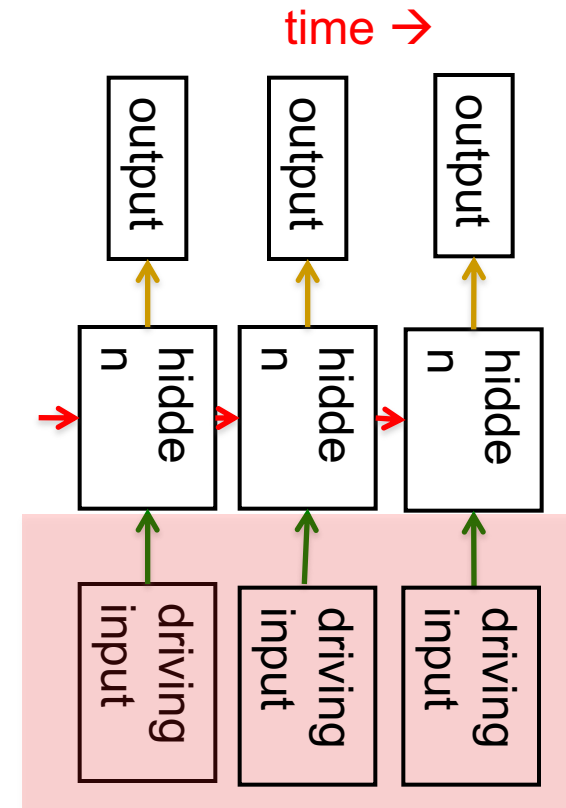


Beyond memoryless models

- If we give our generative model some hidden state, and if we give this hidden state its own internal dynamics, we get a much more interesting kind of model.
 - It can store information in its hidden state for a long time.
 - If the dynamics is noisy and the way it generates outputs from its hidden state is noisy, we can never know its exact hidden state.
 - The best we can do is to infer a probability distribution over the space of hidden state vectors.
- This inference is only tractable for two types of hidden state model.

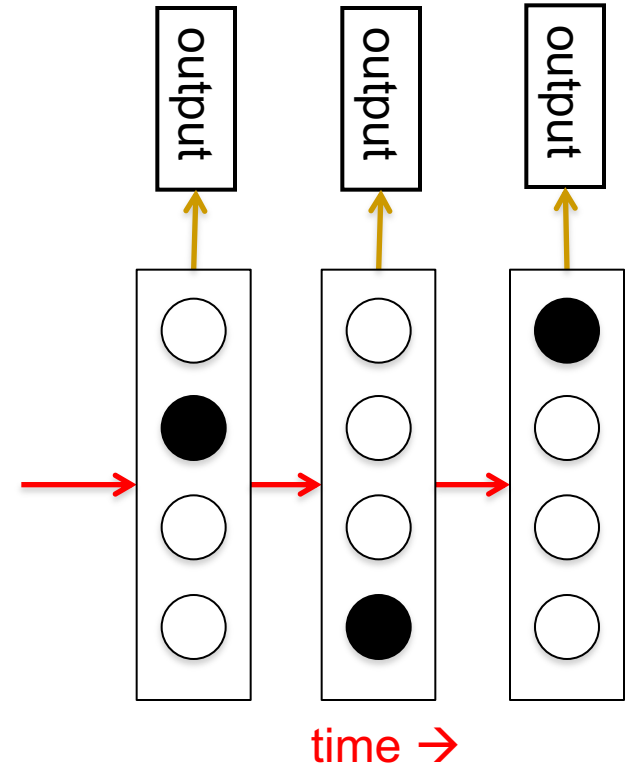
Linear Dynamical Systems (engineers love them!)

- These are generative models. They have a real-valued hidden state that cannot be observed directly.
 - The hidden state has linear dynamics with Gaussian noise and produces the observations using a linear model with Gaussian noise.
 - There may also be driving inputs.
- To predict the next output (so that we can shoot down the missile) we need to infer the hidden state.
 - A linearly transformed Gaussian is a Gaussian. So the distribution over the hidden state given the data so far is Gaussian. It can be computed using “Kalman filtering”.



Hidden Markov Models (computer scientists love them!)

- Hidden Markov Models have a discrete one-of-N hidden state. Transitions between states are stochastic and controlled by a transition matrix. The outputs produced by a state are stochastic.
 - We cannot be sure which state produced a given output. So the state is “hidden”.
 - It is easy to represent a probability distribution across N states with N numbers.
- To predict the next output we need to infer the probability distribution over hidden states.
 - HMMs have efficient algorithms for inference and learning.

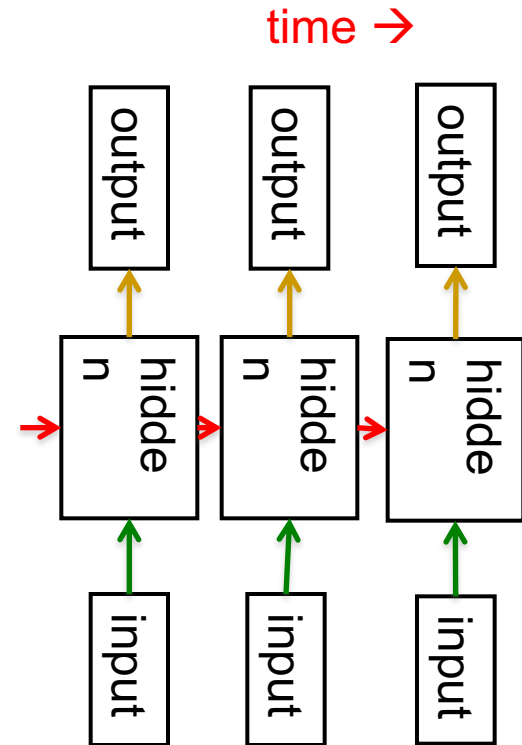


A fundamental limitation of HMMs

- Consider what happens when a hidden Markov model generates data.
 - At each time step it must select one of its hidden states. So with N hidden states it can only remember $\log(N)$ bits about what it generated so far.
- Consider the information that the first half of an utterance contains about the second half:
 - The syntax needs to fit (e.g. number and tense agreement).
 - The semantics needs to fit. The intonation needs to fit.
 - The accent, rate, volume, and vocal tract characteristics must all fit.
- All these aspects combined could be 100 bits of information that the first half of an utterance needs to convey to the second half. 2^{100} is big!

Recurrent neural networks

- RNNs are very powerful, because they combine two properties:
 - Distributed hidden state that allows them to store a lot of information about the past efficiently.
 - Non-linear dynamics that allows them to update their hidden state in complicated ways.
- With enough neurons and time, RNNs can compute anything that can be computed by your computer.



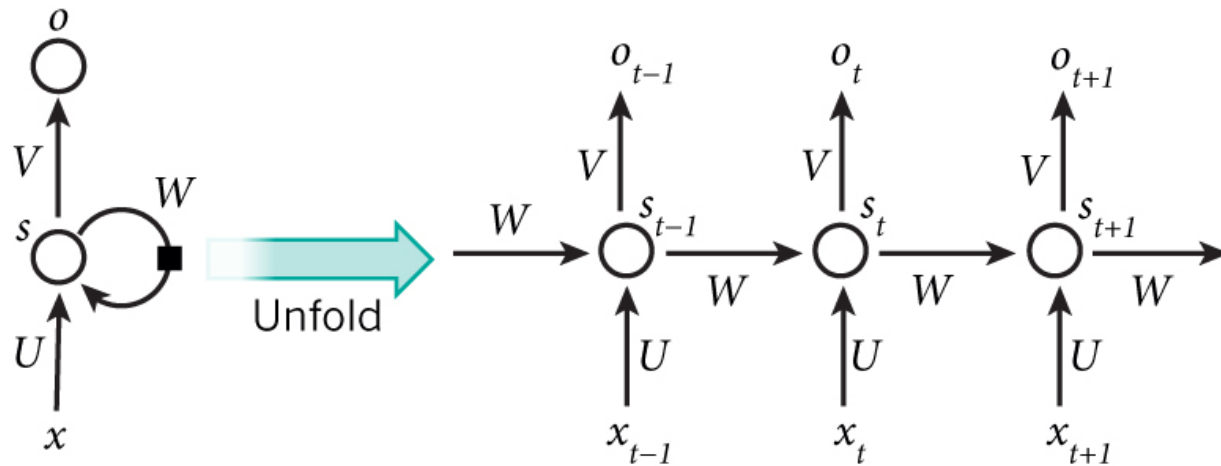
Do generative models need to be stochastic?

- Linear dynamical systems and hidden Markov models are stochastic models.
 - But the posterior probability distribution over their hidden states given the observed data so far is a deterministic function of the data.
- Recurrent neural networks are deterministic.
 - So think of the hidden state of an RNN as the equivalent of the deterministic probability distribution over hidden states in a linear dynamical system or hidden Markov model.

Recurrent neural networks

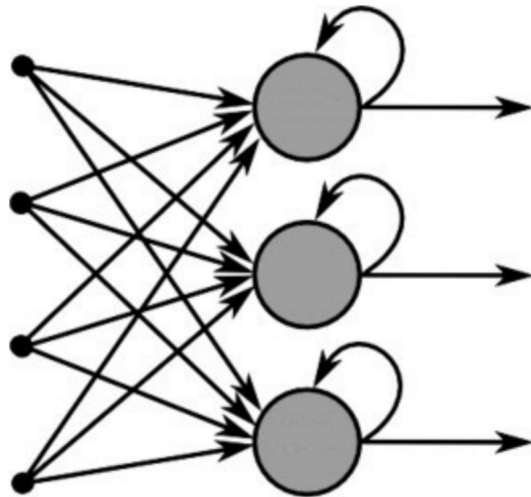
- What kinds of behaviour can RNNs exhibit?
 - They can oscillate. **Good for motor control?**
 - They can settle to point attractors. **Good for retrieving memories?**
 - They can behave chaotically. **Bad for information processing?**
 - RNNs could potentially learn to implement lots of small programs that each capture a nugget of knowledge and run in parallel, interacting to produce very complicated effects.
- But the computational power of RNNs makes them very hard to train.
 - For many years we could not exploit the computational power of RNNs despite some heroic efforts (e.g. Tony Robinson's speech recognizer).

The equivalence between feedforward nets and recurrent nets

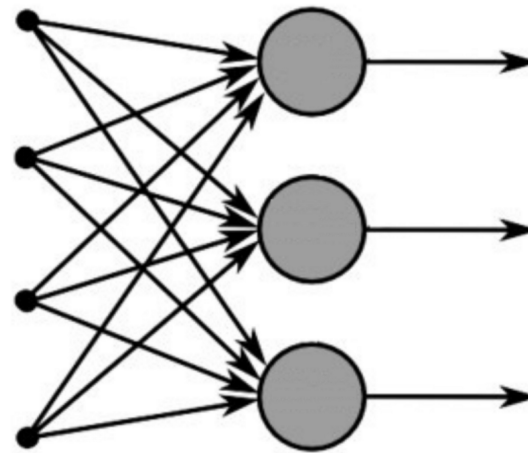


The recurrent net is just a layered net that keeps reusing the same weights.

An Example



Recurrent Neural Network



Feed-Forward Neural Network

Reminder: Backpropagation with weight constraints

- It is easy to modify the backprop algorithm to incorporate linear constraints between the weights.
- We compute the gradients as usual, and then modify the gradients so that they satisfy the constraints.
 - So if the weights started off satisfying the constraints, they will continue to satisfy them.

To constrain: $w_1 = w_2$

we need: $\Delta w_1 = \Delta w_2$

compute: $\frac{\partial E}{\partial w_1}$ and $\frac{\partial E}{\partial w_2}$

use $\frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2}$ *for* w_1 *and* w_2

Backpropagation through time

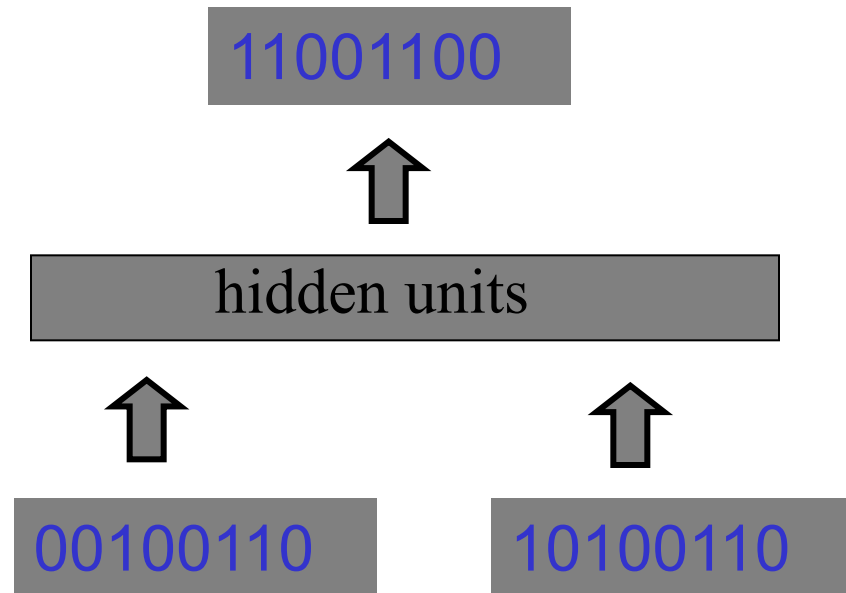
- We can think of the recurrent net as a layered, feed-forward net with shared weights and then train the feed-forward net with weight constraints.
- We can also think of this training algorithm in the time domain:
 - The forward pass builds up a stack of the activities of all the units at each time step.
 - The backward pass peels activities off the stack to compute the error derivatives at each time step.
 - After the backward pass we add together the derivatives at all the different times for each weight.

An irritating extra issue

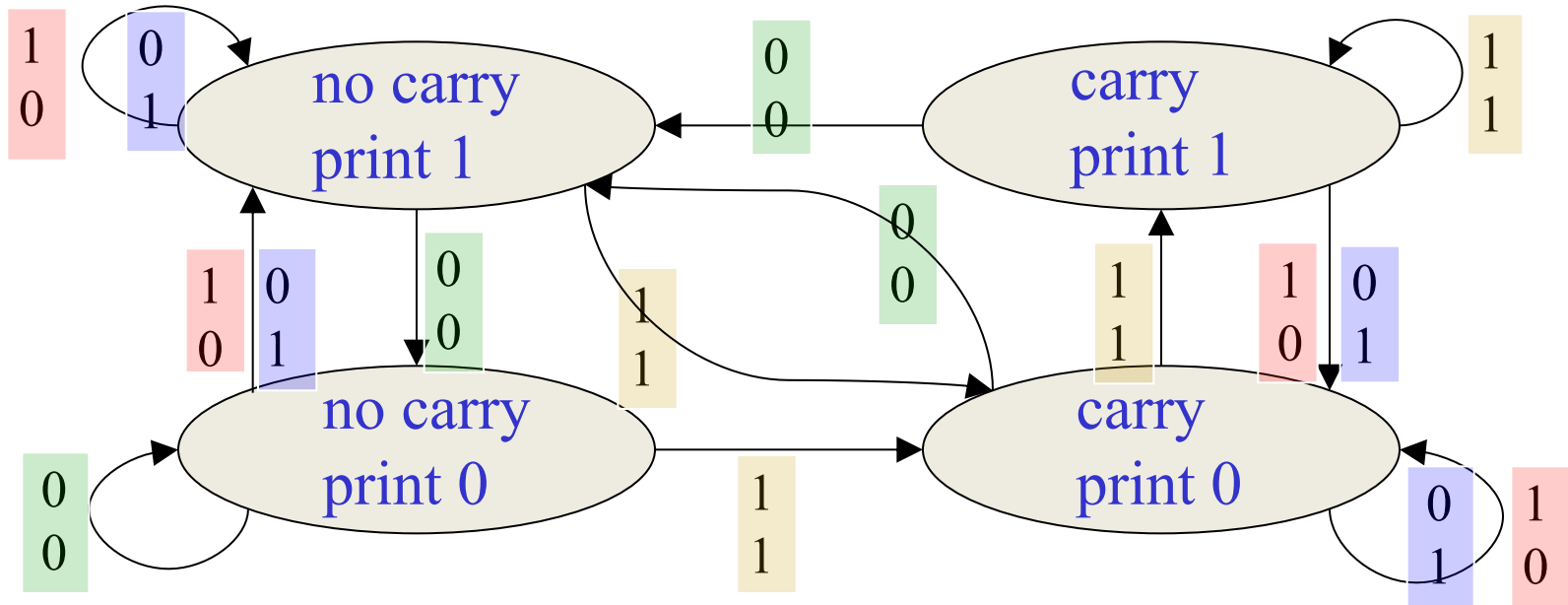
- We need to specify the initial activity state of all the hidden and output units.
- We could just fix these initial states to have some default value like 0.5.
- But it is better to treat the initial states as learned parameters.
- We learn them in the same way as we learn the weights.
 - Start off with an initial random guess for the initial states.
 - At the end of each training sequence, backpropagate through time all the way to the initial states to get the gradient of the error function with respect to each initial state.
 - Adjust the initial states by following the negative gradient.

A good toy problem for a recurrent network

- We can train a feedforward net to do binary addition, but there are obvious regularities that it cannot capture efficiently.
 - We must decide in advance the maximum number of digits in each number.
 - The processing applied to the beginning of a long number does not generalize to the end of the long number because it uses different weights.
- As a result, feedforward nets do not generalize well on the binary addition task.



The algorithm for binary addition



This is a finite state automaton. It decides what transition to make by looking at the next column. It prints after making the transition. It moves from right to left over the two input numbers.

A recurrent net for binary addition

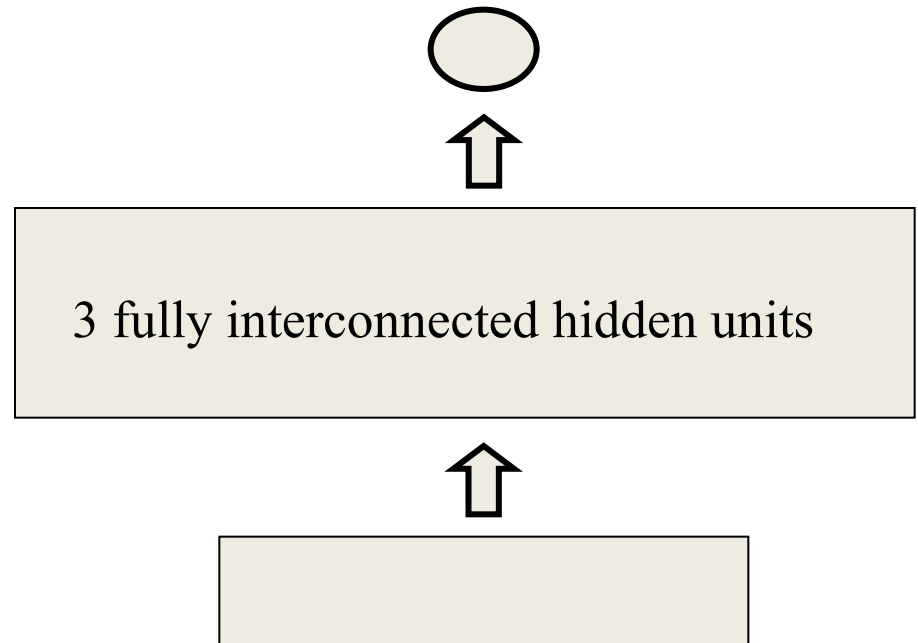
- The network has two input units and one output unit.
- It is given two input digits at each time step.
- The desired output at each time step is the output for the column that was provided as input two time steps ago.
 - It takes one time step to update the hidden units based on the two input digits.
 - It takes another time step for the hidden units to cause the output.

$$\begin{array}{r} 00110100 \\ 01001101 \\ \hline 10000001 \end{array}$$

← time

The connectivity of the network

- The 3 hidden units are fully interconnected in both directions.
 - This allows a hidden activity pattern at one time step to vote for the hidden activity pattern at the next time step.
- The input units have feedforward connections that allow them to vote for the next hidden activity pattern.

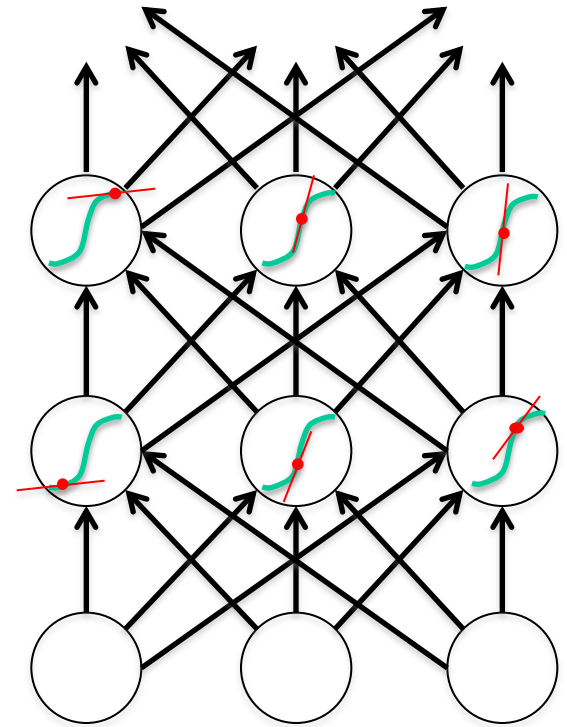


What the network learns

- It learns four distinct patterns of activity for the 3 hidden units. These **patterns** correspond to the nodes in the finite state automaton.
 - Do not confuse units in a neural network with nodes in a finite state automaton. Nodes are like activity vectors.
 - The automaton is restricted to be in exactly one **state** at each time. The hidden units are restricted to have exactly one **vector** of activity at each time.
- A recurrent network can emulate a finite state automaton, but it is exponentially more powerful. With N hidden neurons it has 2^N possible binary activity vectors (but only N^2 weights)
 - This is important when the input stream has two separate things going on at once.
 - A finite state automaton needs to square its number of states.
 - An RNN needs to double its number of **units**.

The backward pass is linear

- There is a big difference between the forward and backward passes.
- In the forward pass we use squashing functions (like the logistic) to prevent the activity vectors from exploding.
- The backward pass, is completely **linear**. If you double the error derivatives at the final layer, all the error derivatives will double.
 - The forward pass determines the slope of the **linear** function used for backpropagating through each neuron.



The problem of exploding or vanishing gradients

- What happens to the magnitude of the gradients as we backpropagate through many layers?
 - If the weights are small, the gradients shrink exponentially.
 - If the weights are big the gradients grow exponentially.
- Typical feed-forward neural nets can cope with these exponential effects because they only have a few hidden layers.
- In an RNN trained on long sequences (*e.g.* 100 time steps) the gradients can easily explode or vanish.
 - We can avoid this by initializing the weights very carefully.
- Even with good initial weights, it's very hard to detect that the current target output depends on an input from many time-steps ago.
 - So RNNs have difficulty dealing with long-range dependencies.

Four effective ways to learn an RNN

- **Long Short Term Memory**
Make the RNN out of little modules that are designed to remember values for a long time.
- **Hessian Free Optimization:** Deal with the vanishing gradients problem by using a fancy optimizer that can detect directions with a tiny gradient but even smaller curvature.
 - The HF optimizer (Martens & Sutskever, 2011) is good at this.
- **Echo State Networks:** Initialize the input→hidden and hidden→hidden and output→hidden connections very carefully so that the hidden state has a huge reservoir of weakly coupled oscillators which can be selectively driven by the input.
 - ESNs only need to learn the hidden→output connections.
- **Good initialization with momentum**
Initialize like in Echo State Networks, but then learn all of the connections using momentum.

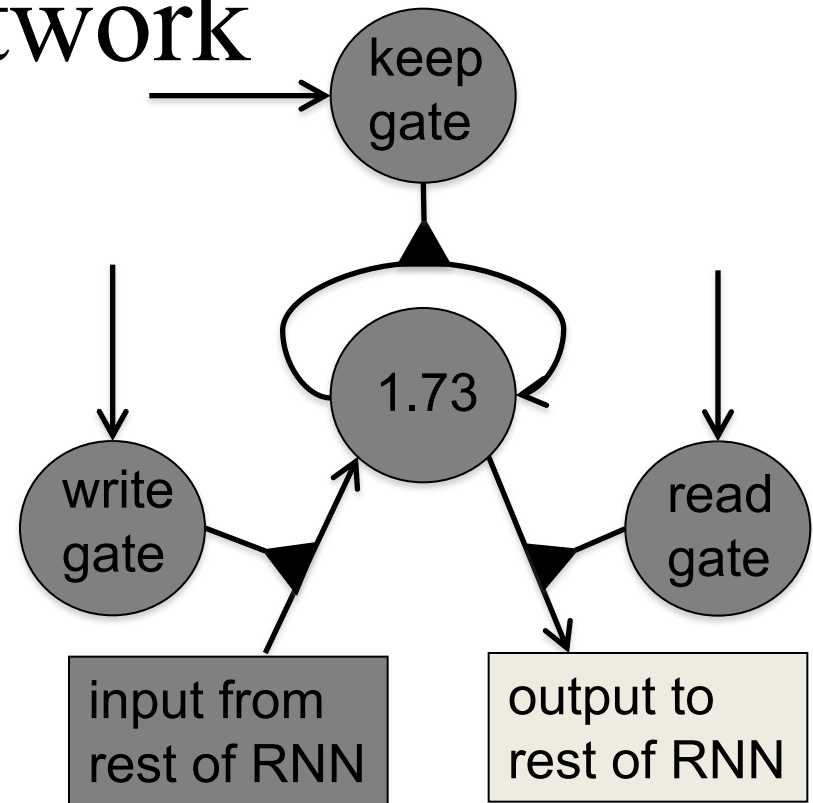
Long Short Term Memory (LSTM)

- Hochreiter & Schmidhuber (1997) solved the problem of getting an RNN to remember things for a long time (like hundreds of time steps).
- They designed a memory cell using logistic and linear units with multiplicative interactions.
- Information gets into the cell whenever its “write” gate is on.
- The information stays in the cell so long as its “keep” gate is on.
- Information can be read from the cell by turning on its “read” gate.

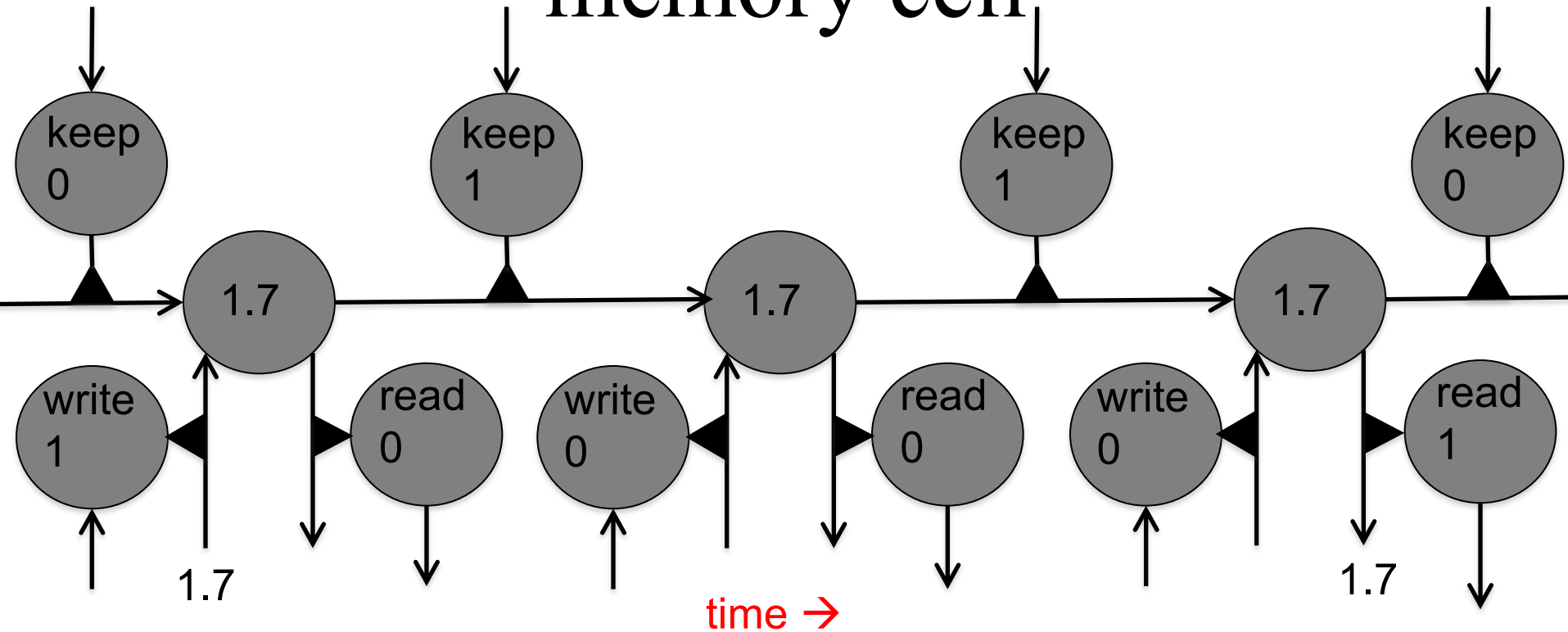
Implementing a memory cell in a neural network

To preserve information for a long time in the activities of an RNN, we use a circuit that implements an analog memory cell.

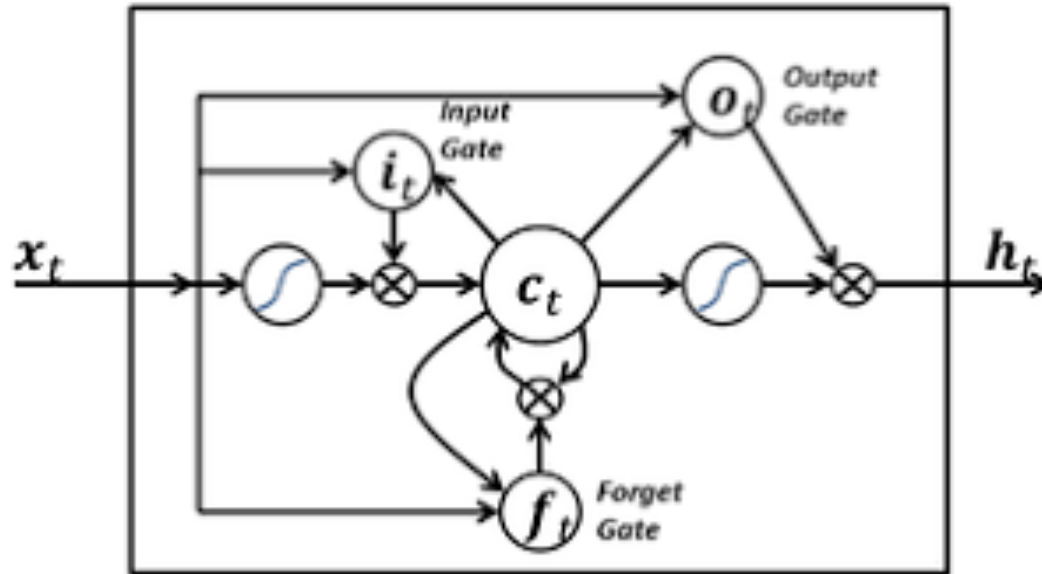
- A linear unit that has a self-link with a weight of 1 will maintain its state.
- Information is stored in the cell by activating its write gate.
- Information is retrieved by activating the read gate.
- We can backpropagate through this circuit because logistics have nice derivatives.



Backpropagation through a memory cell

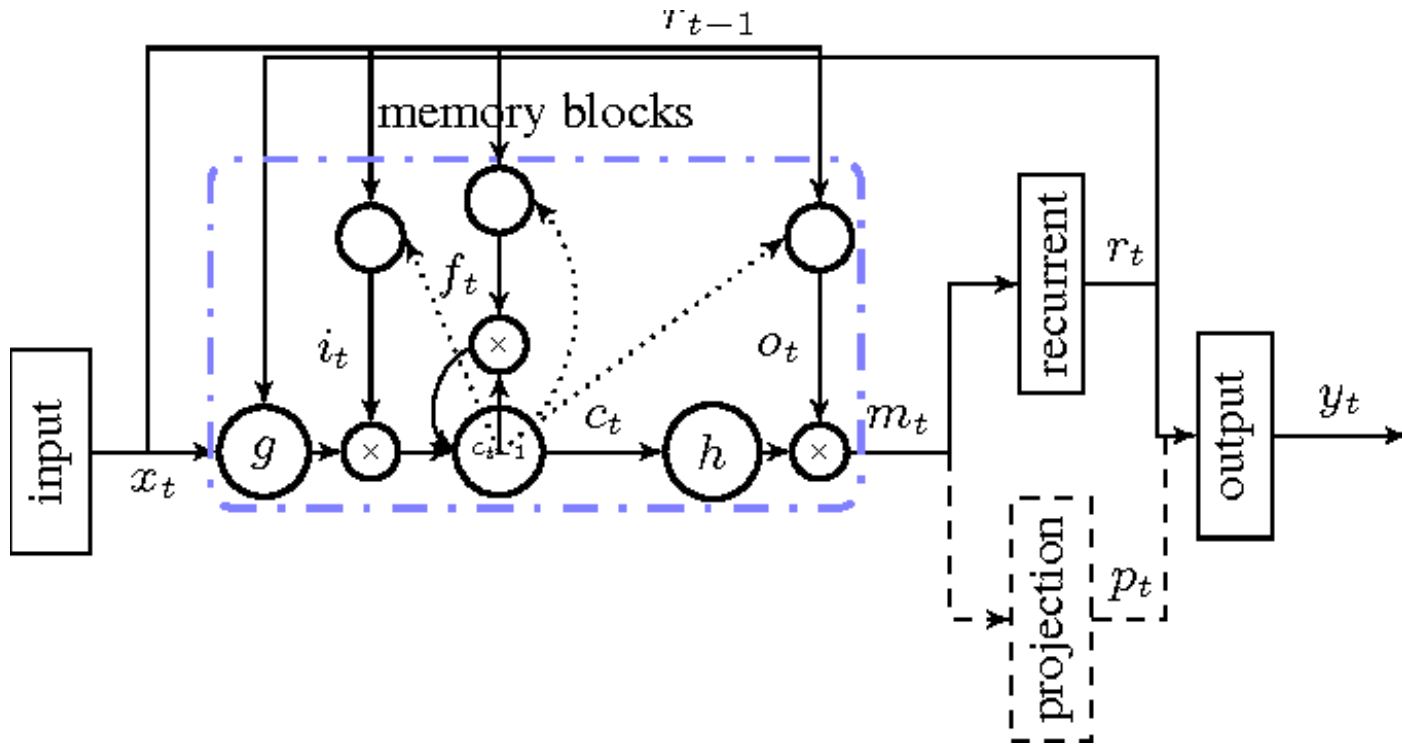


A simple LSTM block with only input, output, and forget gates

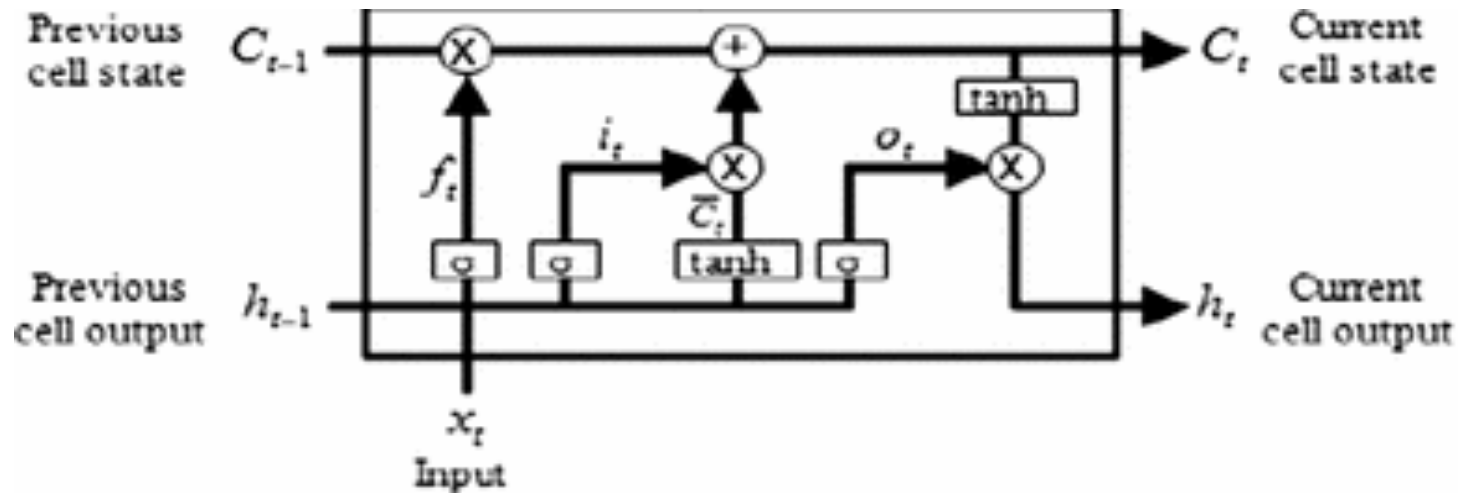


https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/Long_short-term_memory.html

A Simple LSTM Network Example



Another Example



Reading cursive handwriting

- This is a natural task for an RNN.
- The input is a sequence of (x,y,p) coordinates of the tip of the pen, where p indicates whether the pen is up or down.
- The output is a sequence of characters.
- Graves & Schmidhuber (2009) showed that RNNs with LSTM are currently the best systems for reading cursive writing.
 - They used a sequence of small images as input rather than pen coordinates.

A demonstration of online handwriting recognition by an RNN with Long Short Term Memory (from Alex Graves)

Transfer 1m to my paypal
now or i will kill your cat

SHOW ALEX GRAVES' MOVIE

<https://www.cs.toronto.edu/~graves/>

How to generate character strings from the model

- Start the model with its default hidden state.
- Give it a “burn-in” sequence of characters and let it update its hidden state after each character.
- Then look at the probability distribution it predicts for the next character.
- Pick a character randomly from that distribution and tell the net that this was the character that actually occurred.
 - i.e. tell it that its guess was correct, whatever it guessed.
- Continue to let it pick characters until bored.
- Look at the character strings it produces to see what it “knows”.

He was elected President during the Revolutionary War and forgave Opus Paul at Rome. The regime of his crew of England, is now Arab women's icons in and the demons that use something between the characters' sisters in lower coil trains were always operated on the line of the **ephemeral** street, respectively, the graphic or other facility for deformation of a given proportion of large segments at RTUS). The B every chord was a "strongly cold internal palette pour even the white blade."

Some completions produced by the model

- Sheila thrunges (most frequent)
- People thrunge (most frequent next character is space)
- Shiela, Thrungelini del Rey (first try)
- The meaning of life is literary recognition. (6th try)
- The meaning of life is the tradition of the ancient human reproduction: it is less favorable to the good boy for when to remove her bigger. (one of the first 10 tries for a model trained for longer).

What does it know?

- It knows a huge number of words and a lot about proper names, dates, and numbers.
- It is good at balancing quotes and brackets.
 - It can count brackets: **none, one, many**
- It knows a lot about syntax but its very hard to pin down exactly what form this knowledge has.
 - Its syntactic knowledge is not modular.
- It knows a lot of weak semantic associations
 - E.g. it knows Plato is associated with Wittgenstein and cabbage is associated with vegetable.

RNNs for predicting the next word

- Tomas Mikolov and his collaborators have recently trained quite large RNNs on quite large training sets using BPTT.
 - They do better than feed-forward neural nets.
 - They do better than the best other models.
 - They do even better when averaged with other models.
- RNNs require much less training data to reach the same level of performance as other models.
- RNNs improve faster than other methods as the dataset gets bigger.
 - This is going to make them very hard to beat.

Problem of Traditional Neural Network

- Vanishing gradients \rightarrow shallow network
- Exploding gradient
- Cannot use unlabeled data
- Hard to understand the relationship between input and output
- Cannot generate data

Deep Network Versus Shallow Network

- With the same number of weights, deep network is more expressive than shallow network
- Deep network generalizes better than shallow network
- So should we go deep?

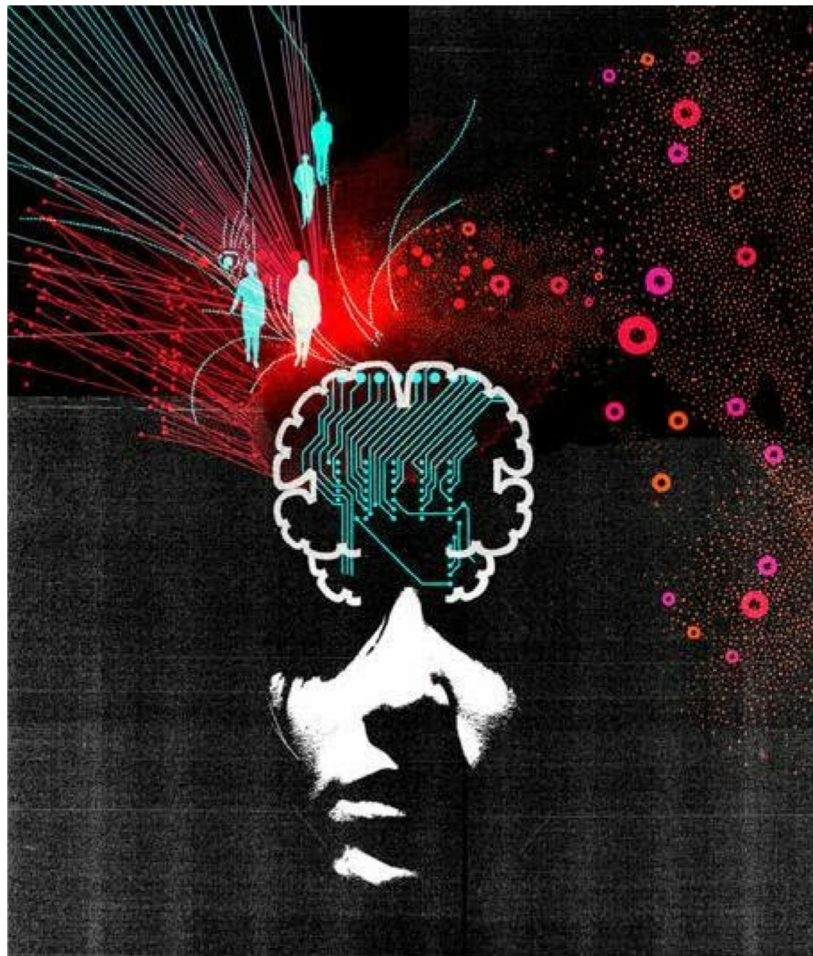
How to go deep?

- ReLu activation function
- Residual network
- Weight sharing (reduce number of parameters)
- Long- and Short-Memory network

- **Big Data**
- **Deep Learning**



Deep Learning Network



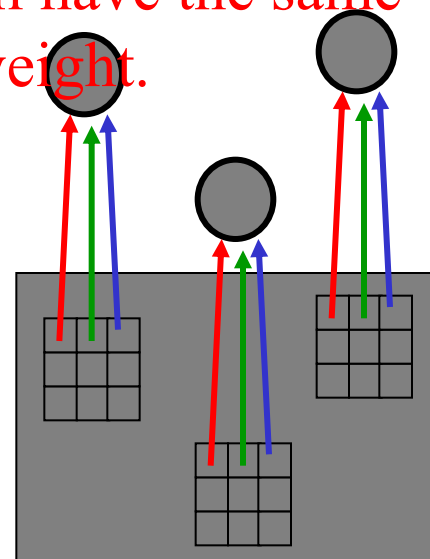
Convolutional Neural Network

The replicated feature approach for hand writing recognition

(currently the dominant approach for neural networks)

- Use many different copies of the same feature detector with different positions.
 - Could also replicate across scale and orientation (*tricky and expensive*)
 - Replication greatly reduces the number of free parameters to be learned.
- Use several different feature types, each with its own map of replicated detectors.
 - Allows each patch of image to be represented in several ways.

The red connections all have the same weight.



Backpropagation with weight constraints

- It's easy to modify the backpropagation algorithm to incorporate linear constraints between the weights.

*To constrain: $w_1 = w_2$
we need: $\Delta w_1 = \Delta w_2$*

compute: $\frac{\partial E}{\partial w_1}$ and $\frac{\partial E}{\partial w_2}$

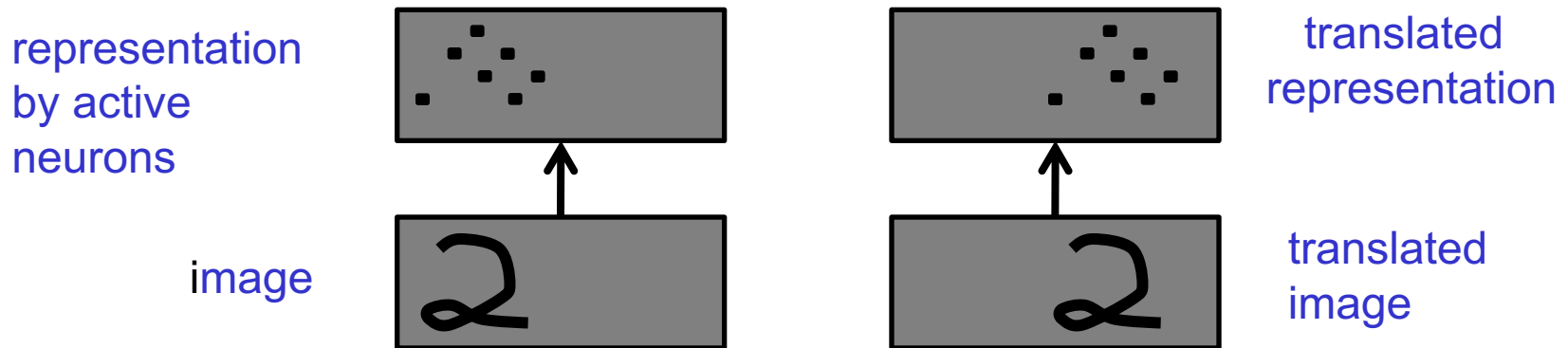
- We compute the gradients as usual, and then modify the gradients so that they satisfy the constraints.

use $\frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2}$ for w_1 and w_2

- So if the weights started off satisfying the constraints, they will continue to satisfy them.

What does replicating the feature detectors achieve?

- **Equivariant activities:** Replicated features do **not** make the neural activities invariant to translation. The activities are equivariant.



- **Invariant knowledge:** If a feature is useful in some locations during training, detectors for that feature will be available in all locations during testing.

Pooling the outputs of replicated feature detectors

- Get a small amount of translational invariance at each level by averaging four neighboring replicated detectors to give a single output to the next level.
 - This reduces the number of inputs to the next layer of feature extraction, thus allowing us to have many more different feature maps.
 - Taking the maximum of the four works slightly better.
- **Problem:** After several levels of pooling, we have lost information about the precise positions of things.
 - This makes it impossible to use the precise spatial relationships between high-level parts for recognition.

Le Net

- Yann LeCun and his collaborators developed a really good recognizer for handwritten digits by using backpropagation in a feedforward net with:
 - Many hidden layers
 - Many maps of replicated units in each layer.
 - Pooling of the outputs of nearby replicated units.
 - A wide net that can cope with several characters at once even if they overlap.
 - A clever way of training a complete system, not just a recognizer.
- This net was used for reading ~10% of the checks in North America.
- Look the impressive demos of LENET at <http://yann.lecun.com>

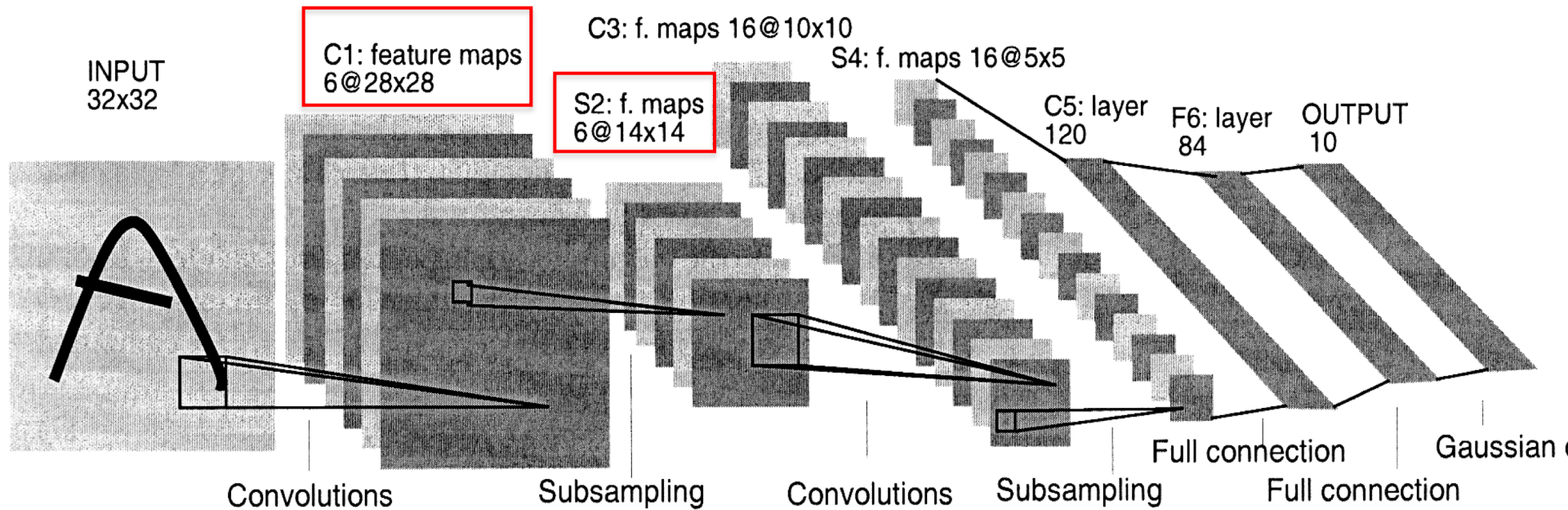
LeNet Demo

- <http://yann.lecun.com/exdb/lenet/index.html>

How LeNet Capture Various Invariance (Demo)

- <http://yann.lecun.com/exdb/lenet/rotation.html>

The architecture of LeNet5






































Priors and Prejudice

- We can put our prior knowledge about the task into the network by designing appropriate:
 - Connectivity.
 - Weight constraints.
 - Neuron activation functions
- This is less intrusive than hand-designing the features.
 - But it still prejudices the network towards the particular way of solving the problem that we had in mind.
- Alternatively, we can use our prior knowledge to create a whole lot more training data.
 - This may require a lot of work (Hofman&Tresp, 1993)
 - It may make learning take much longer.
- It allows optimization to discover clever ways of using the multi-layer network that we did not think of.
 - And we may never fully understand how it does it.

The brute force approach

- LeNet uses knowledge about the invariances to **design**:
 - the local connectivity
 - the weight-sharing
 - the pooling.
- This achieves about 80 errors.
 - This can be reduced to about 40 errors by using many different transformations of the input and other tricks (Ranzato 2008)
- Ciresan *et. al.* (2010) inject knowledge of invariances by creating a huge amount of carefully designed extra training data:
 - For each training image, they produce many new training examples by applying many different transformations.
 - They can then train a large, deep, dumb net on a GPU without much overfitting.
- They achieve about 35 errors.

The errors made by the Ciresan *et. al.* net

 2 2 1 7	 1 1 7 1	 9 8 9 8	 9 9 5 9	 9 9 7 9	 5 5 3 5	 3 8 2 3
 4 9 4 9	 3 5 3 5	 9 4 9 7	 4 9 4 9	 4 4 9 4	 2 2 0 2	 3 5 3 5
 6 6 1 6	 4 4 9 4	 0 0 6 0	 6 6 0 6	 6 6 8 6	 1 1 7 9	 1 1 7 1
 9 9 4 9	 0 0 5 0	 3 5 3 5	 8 8 9 8	 9 9 7 9	 7 7 1 7	 1 1 6 1
 2 7 2 7	 8 8 5 8	 2 2 7 8	 6 6 1 6	 5 5 6 5	 4 4 9 4	 0 0 6 0

The top printed digit is the right answer. The bottom two printed digits are the network's best two guesses.

The right answer is **almost** always in the top 2 guesses.

With model averaging they can now get about 25 errors.

How to detect a significant drop in the error rate

- Is 30 errors in 10,000 test cases significantly better than 40 errors?
 - It all depends on the particular errors!

	model 1 wrong	model 1 right
model 2 wrong	29	1
model 2 right	11	9959

uses the
erful that

	model 1 wrong	model 1 right
model 2 wrong	15	15
model 2 right	25	9945

n
e

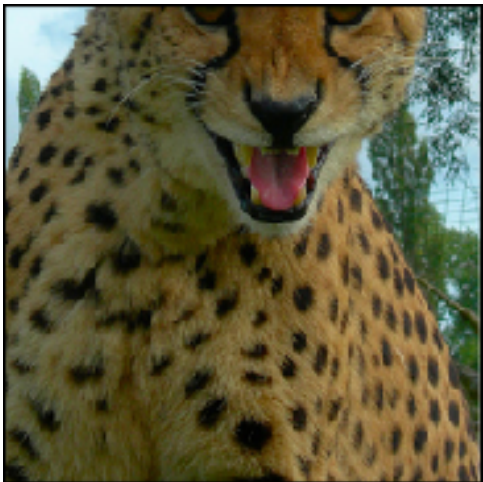
From hand-written digits to 3-D objects

- Recognizing real objects in color photographs downloaded from the web is much more complicated than recognizing hand-written digits:
 - Hundred times as many classes (1000 vs 10)
 - Hundred times as many pixels (256 x 256 color vs 28 x 28 gray)
 - Two dimensional image of three-dimensional scene.
 - Cluttered scenes requiring segmentation
 - Multiple objects in each image.
- Will the same type of convolutional neural network work?

The ILSVRC-2012 competition on ImageNet

- The dataset has 1.2 million high-resolution training images.
- **The classification task:**
 - Get the “correct” class in your top 5 bets. There are 1000 classes.
- **The localization task:**
 - For each bet, put a box around the object. Your box must have at least 50% overlap with the correct box.
- Some of the best existing computer vision methods were tried on this dataset by leading computer vision groups from Oxford, INRIA, XRCE, ...
 - Computer vision systems use complicated multi-stage systems.
 - The early stages are typically hand-tuned by optimizing a few parameters.

Examples from the test set (with the network's guesses)



cheetah

cheetah

leopard

snow leopard

Egyptian cat



bullet train is like a plane, with in-train magazine and a jacket that you can plug your headphones into and listen to

bullet train

bullet train

passenger car

subway train

electric locomotive



hand glass

scissors

hand glass

frying pan

stethoscope

- University of Toronto (Alex Krizhevsky) • 16.4% 34.1%

Error rates on the ILSVRC-2012 competition

- | | classification | classification & localization |
|--|----------------|-------------------------------|
| • University of Tokyo | • 26.1% | |
| • Oxford University Computer Vision Group | • 26.9% | 53.6% |
| • INRIA (French national research institute in CS) + XRCE (Xerox Research Center Europe) | • 27.0% | 50.0% |
| • University of Amsterdam | • 29.5% | |

A neural network for ImageNet

- Alex Krizhevsky (NIPS 2012) developed a very deep convolutional neural net of the type pioneered by Yann Le Cun. Its **architecture** was:
 - 7 hidden layers not counting some max pooling layers.
 - The early layers were convolutional.
 - The last two layers were globally connected.
- The **activation functions** were:
 - Rectified linear units in every hidden layer. These train much faster and are more expressive than logistic units.
 - Competitive normalization to suppress hidden activities when nearby units have stronger activities. This helps with variations in intensity.

Tricks that significantly improve generalization

- Train on random 224x224 patches from the 256x256 images to get more data. Also use left-right reflections of the images.
 - At test time, combine the opinions from ten different patches: The four 224x224 corner patches plus the central 224x224 patch plus the reflections of those five patches.
- Use “dropout” to regularize the weights in the globally connected layers (which contain most of the parameters).
 - Dropout means that half of the hidden units in a layer are randomly removed for each training example.
 - This stops hidden units from relying too much on other hidden units.



Some more examples of how well the deep net works for object recognition.

mite container ship motor scooter leopard

--	--	--	--



grille mushroom cherry Madagascar cat

--	--	--	--

The hardware required for Alex's net

- He uses a very efficient implementation of convolutional nets on two Nvidia GTX 580 Graphics Processor Units (over 1000 fast little cores)
 - GPUs are very good for matrix-matrix multiplies.
 - GPUs have very high bandwidth to memory.
 - This allows him to train the network in a week.
 - It also makes it quick to combine results from 10 patches at test time.
- We can spread a network over many cores if we can communicate the states fast enough.
- As cores get cheaper and datasets get bigger, big neural nets will improve faster than old-fashioned (*i.e.* pre Oct 2012) computer vision systems.

Finding roads in high-resolution images

- Vlad Mnih (ICML 2012) used a non-convolutional net with local fields and multiple layers of rectified linear units to find roads in cluttered aerial images.
 - It takes a large image patch and predicts a binary road label for the central 16x16 pixels.
 - There is lots of labeled training data available for this task.
- The task is hard for many reasons:
 - Occlusion by buildings trees and cars.
 - Shadows, Lighting changes
 - Minor viewpoint changes
- The worst problems are incorrect labels:
 - Badly registered maps
 - Arbitrary decisions about what counts as a road.
- Big neural nets trained on big image patches with millions of examples are the only hope.

The best road-finder on the



1 19

Two ways to average models

- MIXTURE: We can combine models by averaging their output probabilities:

Model A: .3 .2 .5

Model B: .1 .8 .1

Combined .2 .5 .3

- PRODUCT: We can combine models by taking the geometric means of their output probabilities:

Model A: .3 .2 .5

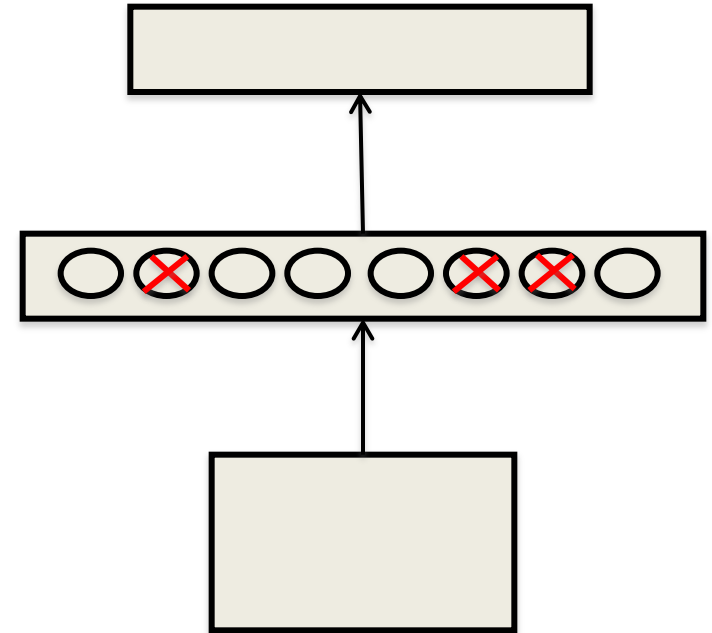
Model B: .1 .8 .1

Combined $\sqrt{.03}$ $\sqrt{.16}$ $\sqrt{.05}$ /sum

Dropout: An efficient way to average many large neural nets

(<http://arxiv.org/abs/1207.0580>)

- Consider a neural net with one hidden layer.
- Each time we present a training example, we randomly omit each hidden unit with probability 0.5.
- So we are randomly sampling from 2^H different architectures.
 - All architectures share weights.



Dropout as a form of model averaging

- We sample from 2^H models. So only a few of the models ever get trained, and they only get one training example.
 - This is as extreme as bagging can get.
- The sharing of the weights means that every model is very strongly regularized.
 - It's a much better regularizer than L2 or L1 penalties that pull the weights towards zero.

But what do we do at test time?

- We could sample many different architectures and take the geometric mean of their output distributions.
- It better to use all of the hidden units, but to halve their outgoing weights.
 - This exactly computes the geometric mean of the predictions of all 2^H models.

What if we have more hidden layers?

- Use dropout of 0.5 in every layer.
- At test time, use the “mean net” that has all the outgoing weights halved.
 - This is not exactly the same as averaging all the separate dropped out models, but it’s a pretty good approximation, and its fast.
- Alternatively, run the stochastic model several times on the same input.
 - This gives us an idea of the uncertainty in the answer.

What about the input layer?

- It helps to use dropout there too, but with a higher probability of keeping an input unit.
 - This trick is already used by the “denoising autoencoders” developed by Pascal Vincent, Hugo Larochelle and Yoshua Bengio.

How well does dropout work?

- The record breaking object recognition net developed by Alex Krizhevsky uses dropout and it helps a lot.
- If your deep neural net is significantly overfitting, dropout will usually reduce the number of errors by a lot.
 - Any net that uses “early stopping” can do better by using dropout (at the cost of taking quite a lot longer to train).
- If your deep neural net is not overfitting you should be using a bigger one!

Another way to think about dropout

- If a hidden unit knows which other hidden units are present, it can co-adapt to them on the training data.
 - But complex co-adaptations are likely to go wrong on new test data.
 - Big, complex conspiracies are not robust.
- If a hidden unit has to work well with combinatorially many sets of co-workers, it is more likely to do something that is individually useful.
 - But it will also tend to do something that is marginally useful given what its co-workers achieve.

Recent Progress on ImageNet Competition

Google acquires U of T neural networks company

SHARE THIS



Sara Franca

University Professor **Geoffrey Hinton** and two of his graduate students from the Department of Computer Science have sold their startup company to Google Inc.

Google acquired the company, incorporated by **Alex Krizhevsky, Ilya Sutskever** and Hinton in 2012, for its research on deep neural networks. Also known as "deep learning" for computers, this research involves helping machines understand context.

[Hinton](#) is world-renowned for his work with machine learning

and artificial intelligence. His neural networks research has profound implications for areas such as speech recognition, computer vision and language understanding.

"Geoffrey Hinton's research is a magnificent example of disruptive innovation with roots in basic research," said U of T's president, Professor **David Naylor**. "The discoveries of brilliant researchers, guided freely by their expertise, curiosity, and intuition, lead eventually to practical applications no one could have imagined, much less requisitioned.



From left: Ilya Sutskever, Alex Krizhevsky and University Professor Geoffrey Hinton of the University of Toronto's Department of Computer Science (photo by John Guatto)

Facebook Launches Advanced AI Effort to Find Meaning in Your Posts

A technique called deep learning could help Facebook understand its users and their data better.

By Tom Simonite on September 20, 2013



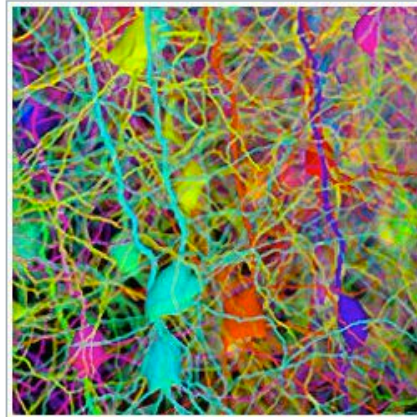
Deep Learning Comes of Age

By Gary Anthes

Communications of the ACM, Vol. 56 No. 6, Pages 13-15

10.1145/2461256.2461262

[Comments](#)



Rainbow brainwaves made from a computer simulation of pyramidal neurons found in the cerebral cortex.

Credit: Hermann Cuntz

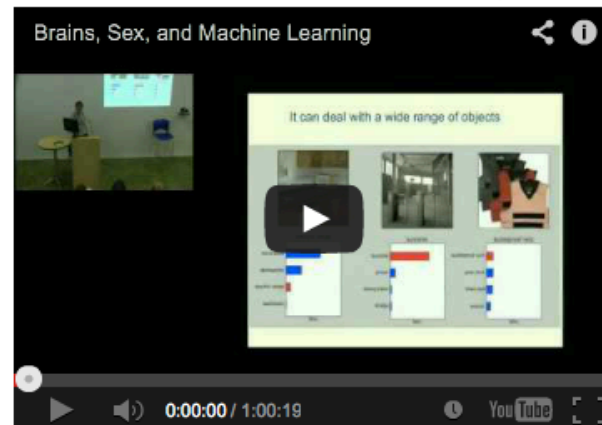
Improvements in algorithms and application architectures, coupled with the recent availability of very fast computers and huge datasets, are enabling major increases in the power of machine learning systems. In particular, multilayer artificial neural networks are producing startling improvements in the accuracy of computer vision, speech recognition, and other applications in a field that has become known as "deep learning."

Artificial neural networks ("neural nets") are patterned after the arrangement of neurons in the brain, and the connections, or synapses, between the neurons. Work on neural nets dates to the 1960s; although conceptually compelling, they proved difficult to apply effectively, and they did not begin to find broad commercial use until the early 1990s.

Neural nets are systems of highly interconnected, simple processing elements. The behavior of the net changes according to the "weights" assigned to each connection, with the output of any node determined by the weighted sum of its inputs. The nets

do not work according to hand-coded rules, as with traditional computer programs; they must be trained, which involves an automated process of successively changing the inter-nodal weights in order to minimize the difference between the desired output and the actual output. Generally, the more input data used for this training, the better the results.

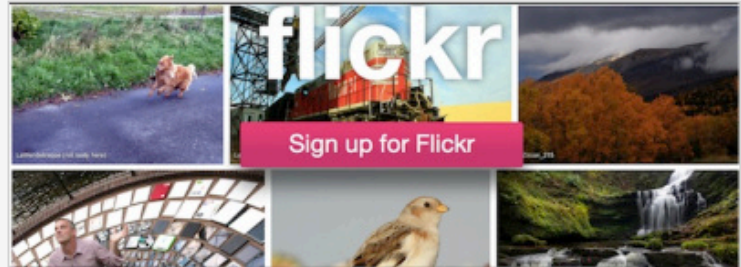
For years, most neural nets contained a single layer of "feature detectors" and were trained mainly with labeled data in a process called "supervised" training. In these kinds of networks, the system is shown an input and told what



Yahoo Acquires Startup LookFlow To Work On Flickr And ‘Deep Learning’

Posted Oct 23, 2013 by [Anthony Ha](#)

6 [Like](#) 184 [Tweet](#) 330 [Share](#) 41



LookFlow, a startup that describes itself as “an entirely new way to explore images you love,” just announced that it has been acquired by Yahoo and will be joining the Flickr team.

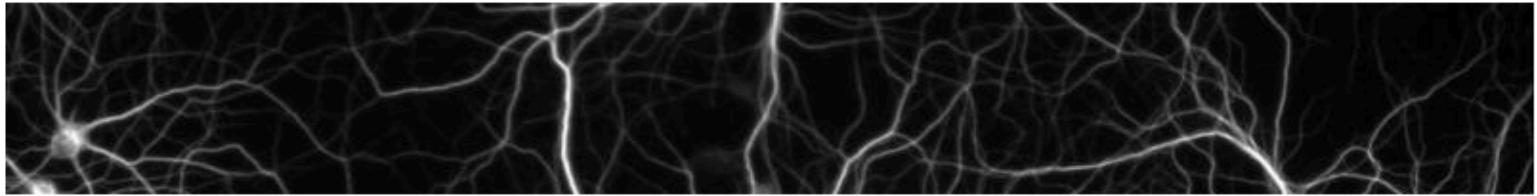
The company [writes on its homepage](#), “Fret not, LookFlow fans. Keep an eye out for our product in future versions of Flickr — with many more wonderful photos and all that Flickr awesomeness!” It also says it will be helping Yahoo to form a new “deep learning group.”

ADVERTISEMENT

Meet Venmo Touch.

[Learn more](#)

Braintree



Google's Large Scale Deep Learning Experiments

Google's new large-scale learning experimentation using 16000 CPU cores and deep learning as part of google brain project had made a big success on Imagenet dataset. This success had a wide media coverage. Some pointers to the news:

Google official blog, 26 June 2012 <http://googleblog.blogspot.com/2012/06/using-large-scale-brain-simulations-for.html>

NYT Front page on large scale neural network John Markoff, [...]

Deep Learning Revolution

**The
New York
Times**

2012: Is deep learning a revolution in artificial intelligence?

Accomplishments

Apple's Siri virtual personal assistant

Google's Street View & Self-Driving Car

Google/Facebook/Tweeter/Yahoo Deep Learning Acquisition

Hinton's Hand Writing Recognition

CASP10 protein contact map prediction



Outline

- Motivating factors for study
- RBMs
- Deep Belief Networks
- Applications



The Toolbox

We often reach for the familiar...

For discriminative tasks we have

- neural networks (~1980's, back-prop)
- SVM (~1990's, Vapnik)



But is there anything better out there???

Challenges with SVM/NN

Potential difficulties with SVM

- Training time for large datasets
- Large number of support vectors for hard classification problems

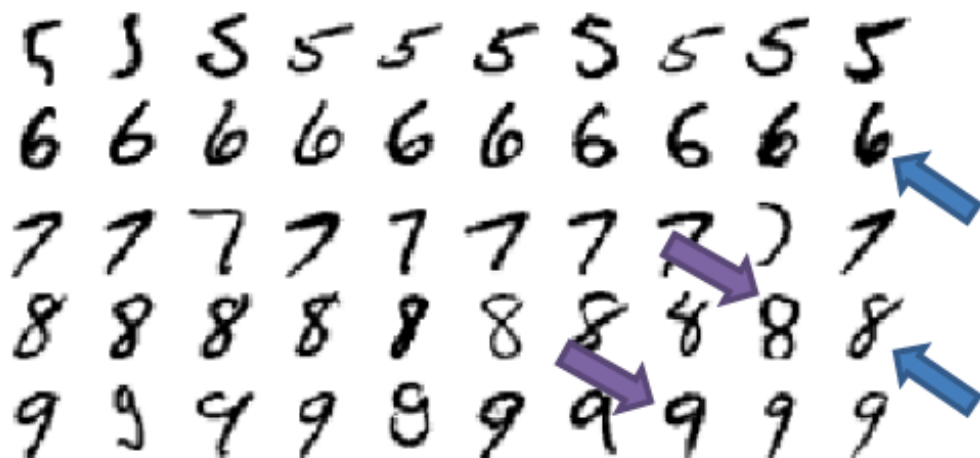
Potential difficulties with NN & back-prop

- Diminishing gradient inhibits multiple layers
- Can get stuck in local minimums
- Training time can be extensive

Challenges with SVM/NN

More general “problems” with NNs and SVM...

- Need labeled data (what about unlabeled data?)
- Amount of information restricted by labels (ie, hard to learn a complex model if we are limited by labels)



What if I could use “8”s
to learn to recognize
“6”s?

How to respond to these challenges

- Try to model the structure of the sensory input (ie, data), but keep the efficiency and simplicity of a gradient method
 - Adjust the weights to maximize the probability that a generative model would have produced the sensory input.
 - Learn $p(\text{data})$ not $p(\text{label} | \text{data})$
- So instead of learning a label, first learn how to generative your data

How to respond to these challenges

- Try to model the structure of the sensory input (ie, data), but keep the efficiency and simplicity of a gradient method
 - Adjust the weights to maximize the probability that a generative model would have produced the sensory input.

→ – Learn $p(\text{data})$ not $p(\text{label} | \text{data})$ ←

- So instead of learning $p(\text{label} | \text{data})$, first learn how to generative your data

Immediate benefit in that all data does not have to be label. Also reduces dependency on label.

Hinton, 2007

Recap

So, we are convinced we ...

1. recognize some concerns with “standard” tools and would like what other options are out there
2. like the idea of modeling the input first (ie, building a model of our data as oppose to an out right classifier)



Energy Based Models

$p(x)$ – probability of our data; data is represented by feature vector \mathbf{x} .

$$p(x) = \frac{e^{-E(x)}}{Z}$$

and

$$Z = \sum_x e^{-E(x)}$$

Attach an energy function (ie, $E(x)$) to score a configuration (ie, each possible input x).

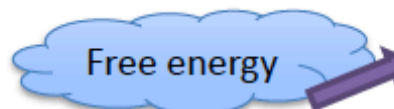
We want desirable data to have low energy. Thus, tweak the parameters of $E(x)$ accordingly.

EBMs with Hidden Units

To increase power of EBMs, add hidden variables.

$$P(x) = \sum_h P(x, h) = \sum_h \frac{e^{-E(x, h)}}{Z}.$$

By using the notation,


$$\mathcal{F}(x) = -\log \sum_h e^{-E(x, h)}$$

We can rewrite $p(x)$ in a form similar to the standard EBM,

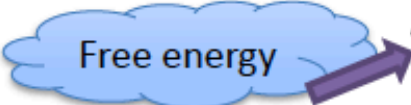
$$P(x) = \frac{e^{-\mathcal{F}(x)}}{Z} \text{ with } Z = \sum_x e^{-\mathcal{F}(x)}.$$

EBMs with Hidden Units

To increase power of EBMs, add hidden variables.

$$P(x) = \sum_h P(x, h) = \sum_h \frac{e^{-E(x,h)}}{Z}.$$

By using the notation,

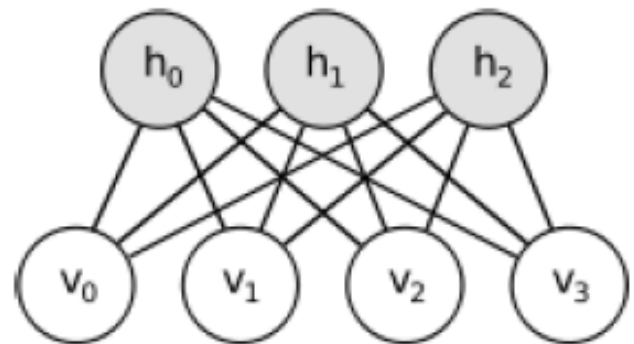

$$\mathcal{F}(x) = -\log \sum_h e^{-E(x,h)}$$

We can rewrite $p(x)$ in a form similar to the standard EBM,

$$P(x) = \frac{e^{-\mathcal{F}(x)}}{Z} \text{ with } Z = \sum_x e^{-\mathcal{F}(x)}.$$

RBM

- Represented by a bipartite graph, with symmetric, weighted connections
- One layer has visible nodes and the other hidden (ie, latent) variables.
- Nodes are often binary , stochastic units (ie, assume 0 or 1 based on probability)



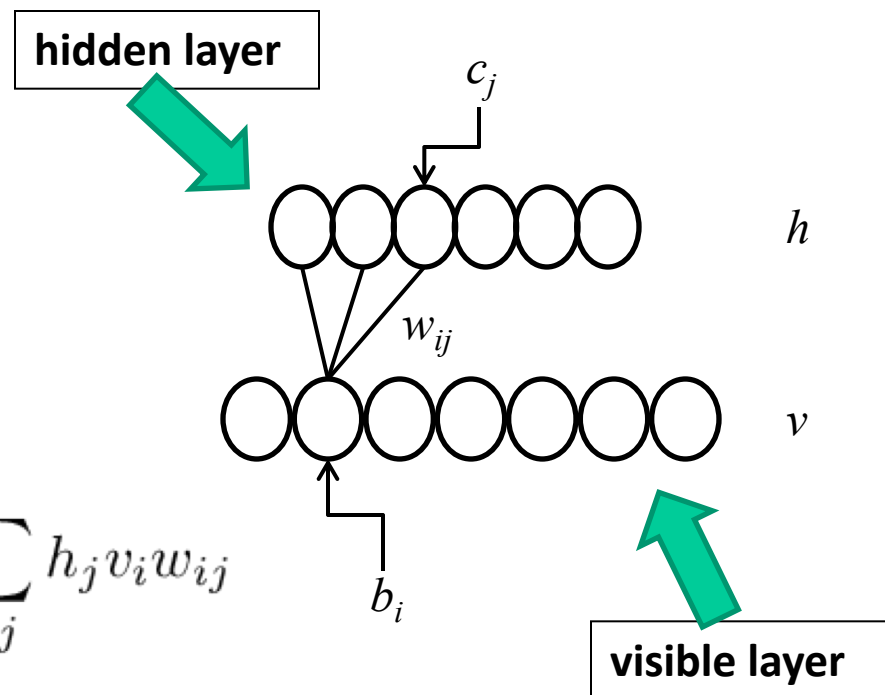
Restricted Boltzmann Machine (RBM)

- A model for a distribution over binary vectors
- Probability of a vector, v , under the model is defined via an “energy”

$$E(v, h) = - \sum_i b_i v_i - \sum_j c_j h_j - \sum_{i,j} h_j v_i w_{ij}$$

$$Z = \sum_v \sum_h e^{-E(v,h)}$$

$$p(v) = \sum_h \frac{e^{-E(v,h)}}{Z}$$



Training a RBM – Maximum Likelihood Approach

$$l(\theta) = \frac{1}{n} \sum_i \log\left(\sum_h e^{-E(v_i, h)}\right) - \log Z$$

$$\frac{\partial l(\theta)}{\partial \theta_j} = \frac{1}{n} \sum_i \frac{\sum_h e^{-E(v_i, h)} \frac{-\partial E}{\partial \theta_j}}{\sum_h e^{-E(v_i, h)}} - \frac{1}{Z} \sum_v \sum_h e^{-E(v, h)} \frac{-\partial E}{\partial \theta_j}$$

$$= \frac{1}{n} \sum_i \sum_h \frac{p(v_i, h)}{p(v_i)} \left(\frac{-\partial E}{\partial \theta_j} \right) - E \left[\frac{-\partial E}{\partial \theta_j} \right]_{p^\infty}$$

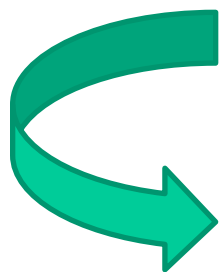
$$= \frac{1}{n} \sum_i \sum_h p(h|v_i) \left(\frac{-\partial E}{\partial \theta_j} \right) - E \left[\frac{-\partial E}{\partial \theta_j} \right]_{p^\infty}$$

$$= E \left[\frac{-\partial E}{\partial \theta_j} \right]_{p^0} - E \left[\frac{-\partial E}{\partial \theta_j} \right]_{p^\infty}$$

Training a RBM - Contrastive Divergence (CD)

Instead of attempting to sample from joint distribution $p(v,h)$ (i.e. p^∞), sample from

$$\Delta\theta_j \propto E \left[\frac{-\partial E}{\partial \theta_j} \right]_{p^0} - E \left[\frac{-\partial E}{\partial \theta_j} \right]_{p^\infty}$$



$$\Delta\theta_j \propto E \left[\frac{-\partial E}{\partial \theta_j} \right]_{p^0} - E \left[\frac{-\partial E}{\partial \theta_j} \right]_{p^1}$$

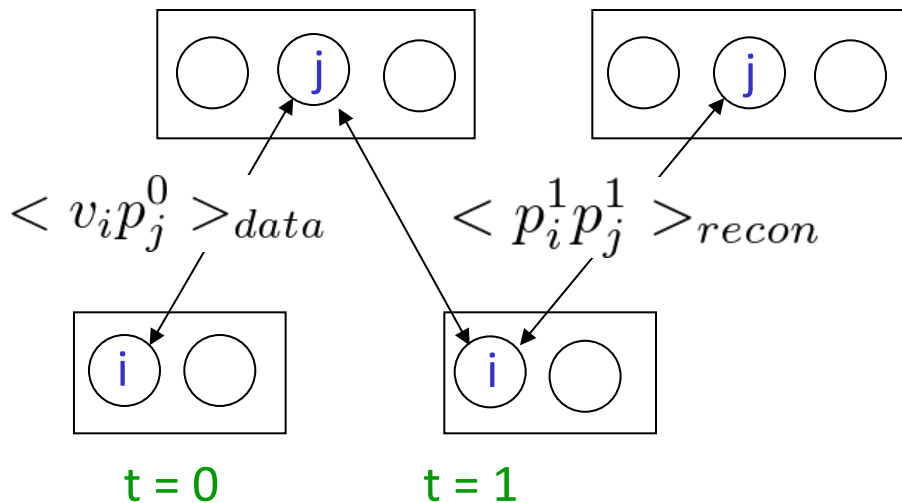
Hinton, *Neural Computation*(2002)

Faster and lower variance in sample.

Training a RBM

Partials of $E(v, h)$ easy to calculate.

$$\frac{\partial E}{\partial w_{ij}} = v_i h_j$$



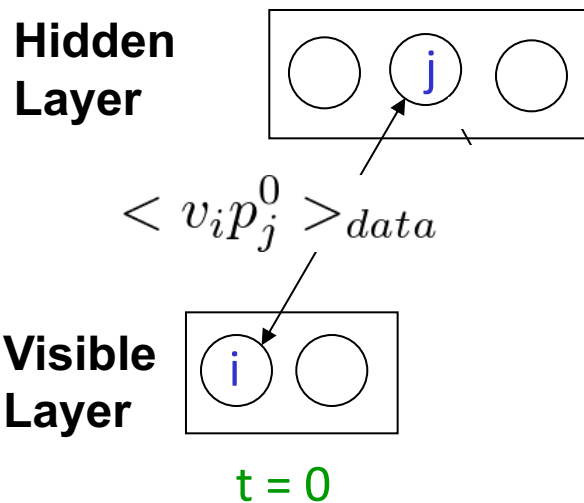
$$p_j^{(0)} = \sigma\left(\sum_i v_i w_{ij} + c_j\right)$$

$$p_i^{(1)} = \sigma\left(\sum_j h_j w_{ij} + b_i\right)$$

$$p_j^{(1)} = \sigma\left(\sum_i p_i^1 w_{ij} + c_j\right)$$

Training a RBM via Contrastive Divergence

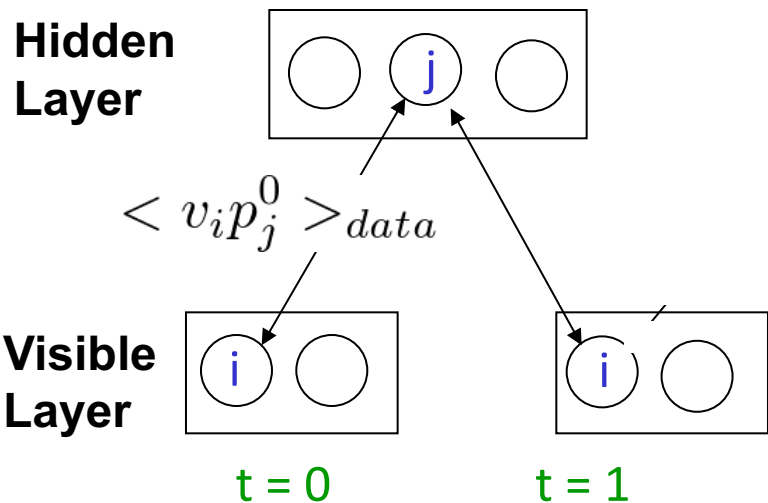
Gradient of the likelihood with respect to $w_{ij} \approx$ the difference between interaction of v_i and h_j at time 0 and at time 1.



$$p_j^{(0)} = \sigma\left(\sum_i v_i w_{ij} + c_j\right)$$

Training a RBM via Contrastive Divergence

Gradient of the likelihood with respect to $w_{ij} \approx$ the difference between interaction of v_i and h_j at time 0 and at time 1.

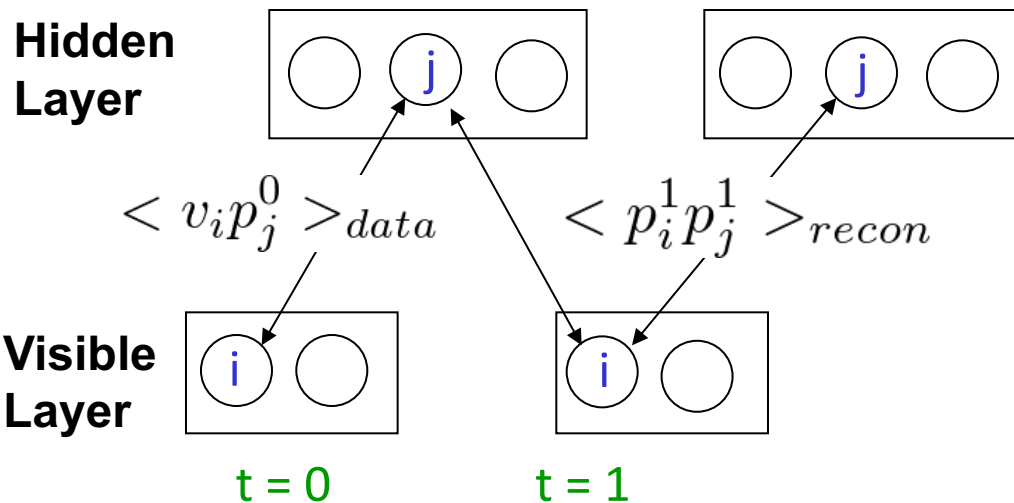


$$p_j^{(0)} = \sigma\left(\sum_i v_i w_{ij} + c_j\right)$$

$$p_i^{(1)} = \sigma\left(\sum_j h_j w_{ij} + b_i\right)$$

Training a RBM via Contrastive Divergence

Gradient of the likelihood with respect to $w_{ij} \approx$ the difference between interaction of v_i and h_j at time 0 and at time 1.



$$p_j^{(0)} = \sigma\left(\sum_i v_i w_{ij} + c_j\right)$$

$$p_i^{(1)} = \sigma\left(\sum_j h_j w_{ij} + b_i\right)$$

$$p_j^{(1)} = \sigma\left(\sum_i p_i^1 w_{ij} + c_j\right)$$

$$\Delta w_{i,j} = \langle v_i p_j^0 \rangle - \langle p_i^1 p_j^1 \rangle$$

Weight/Bias Updates

$$\Delta^{(n)} w_{ij} = \epsilon \{ (\langle v_i p_j^{(0)} \rangle - \langle p_i^{(1)} p_j^{(1)} \rangle) - \eta w_{ij} \} + \nu w_{ij}^{(n-1)}$$

$$\Delta^{(n)} b_i = \epsilon \{ (\langle v_i \rangle - \langle p_i^{(1)} \rangle) \} + \nu b_i^{(n-1)}$$

$$\Delta^{(n)} c_j = \epsilon \{ (\langle p_j^{(0)} \rangle - \langle p_j^{(1)} \rangle) \} + \nu c_j^{(n-1)}$$

ϵ is the learning rate, η is the weight cost, and ν the momentum.



Gradient

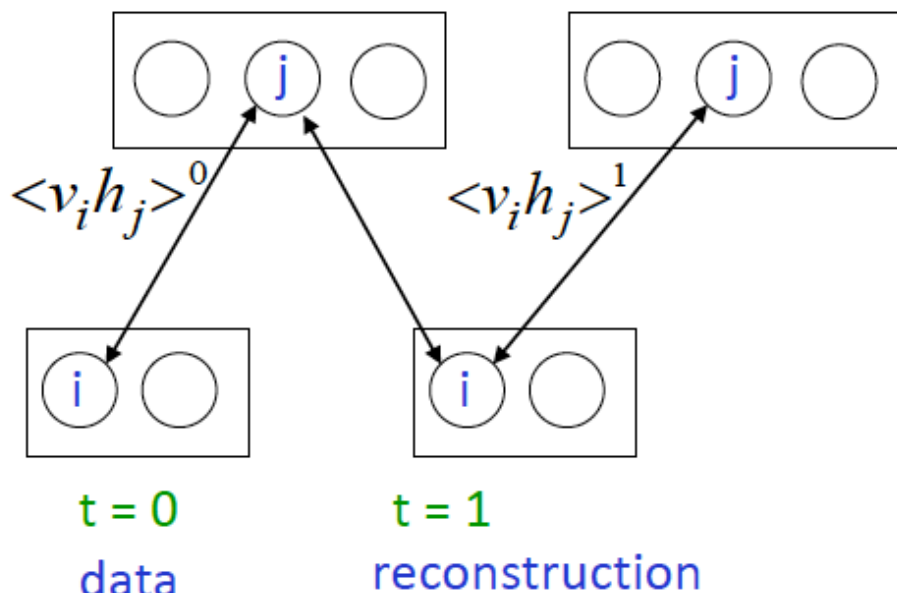


Smaller Weights



Avoid Local Minima

A quick way to learn an RBM



Start with a training vector on the visible units.

Update all the hidden units in parallel

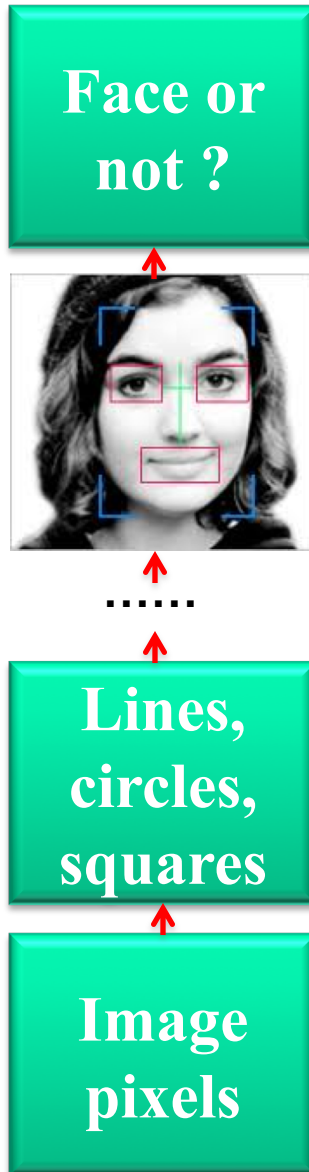
Update the all the visible units in parallel to get a “reconstruction”.

Update the hidden units again.

$$\Delta w_{ij} = \varepsilon (\langle v_i h_j \rangle^0 - \langle v_i h_j \rangle^1)$$

This is not following the gradient of the log likelihood. But it works well. It is approximately following the gradient of another objective function (Carreira-Perpinan & Hinton, 2005).

Why Deep Learning? – A Face Recognition Analogy



Brain Learning

Training a RBM – A Maximum Likelihood Approach

Objective of Unsupervised Learning:

Find $w_{i,j}$ to maximize the likelihood $p(\mathbf{v})$ of visible data

Iterative Gradient Descent Approach:

Adjust $w_{i,j}$ to increase the likelihood according to gradient

Why ???

Okay, we can model $p(x)$.

But how to...

1. Find $p(\text{label} | x)$. We want a **classifier!**
2. Improve the model for $p(x)$.

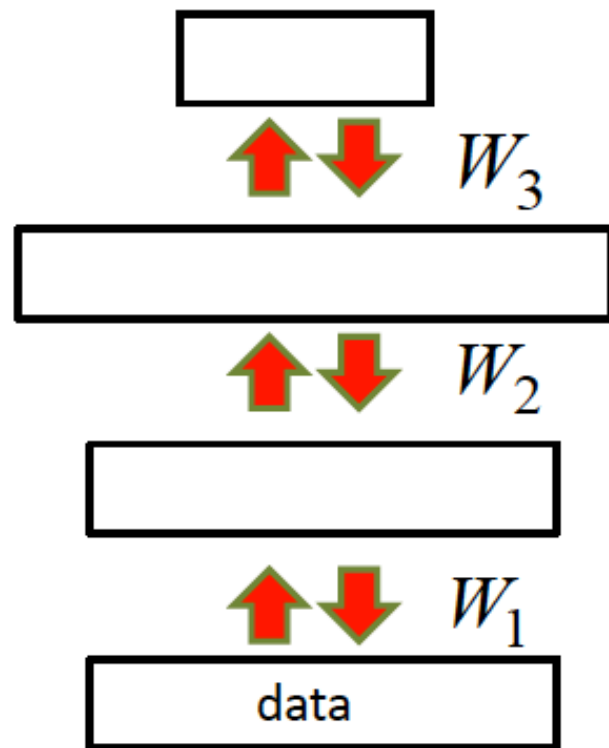


Deep Belief Nets

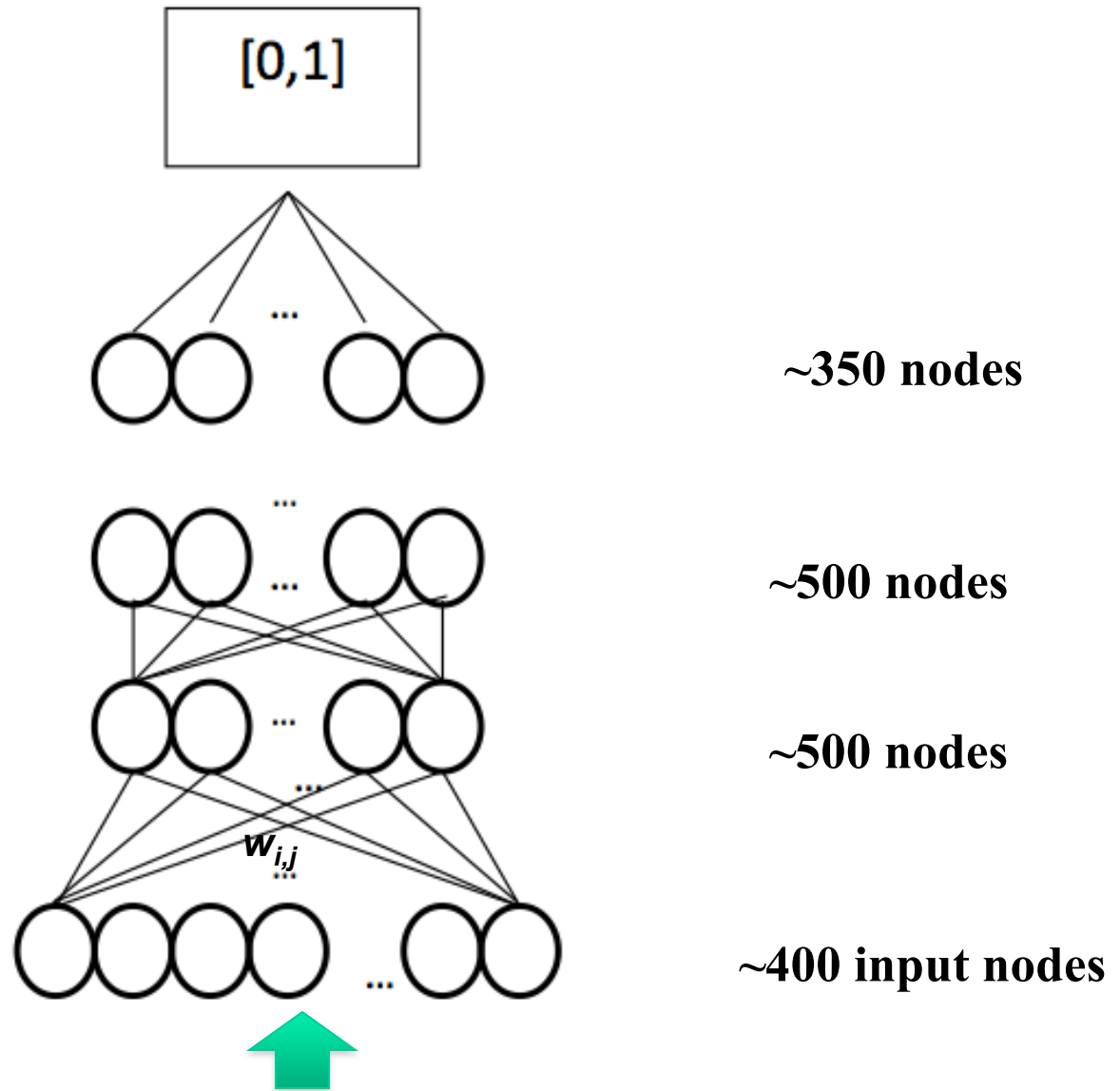
RBM's are typically used in stack

- Train them up one layer at a time
- Hidden units become visible units to the next layer up

If your goal is a discriminator, you train a classifier on the top level representation of your input.

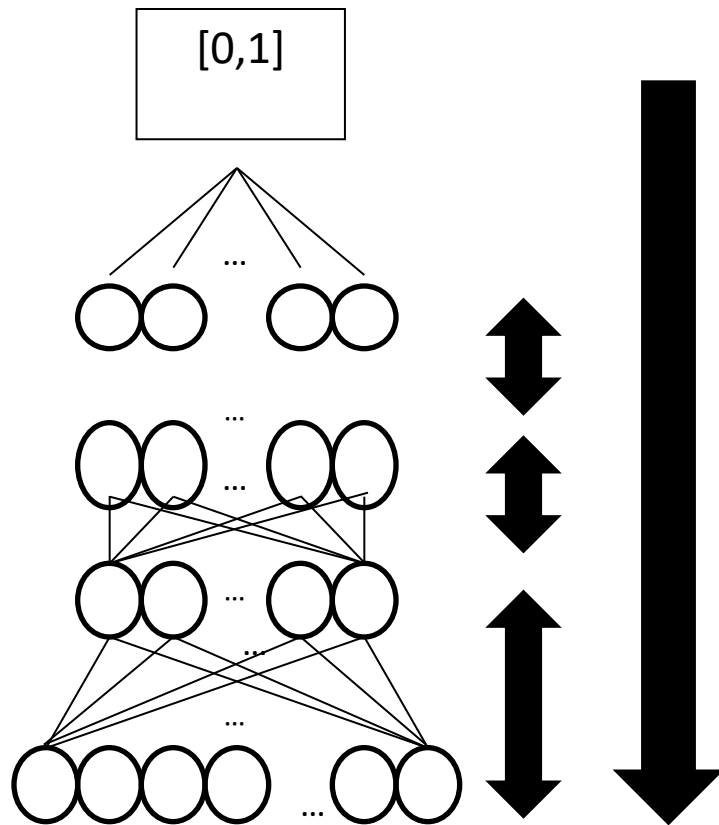


Deep Learning Network Architecture



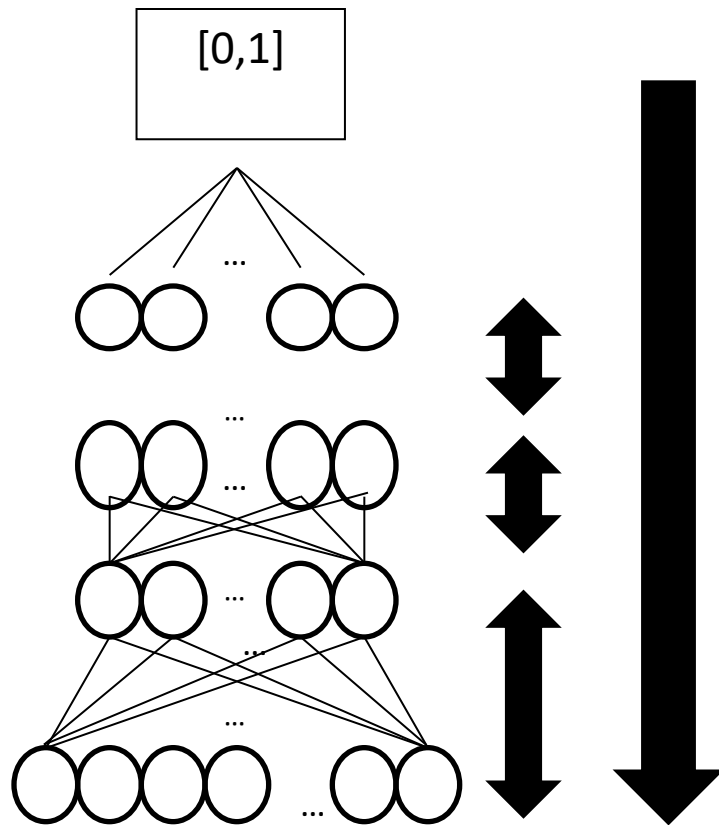
A Vector of ~400 Features (numbers between 0 and 1)

Training a Deep Network



1. **Weights are learned layer by layer via unsupervised learning.**
2. **Final layer is learned as a supervised neural network.**
3. **All weights are fine-tuned using supervised back propagation.**

Training a Deep Network

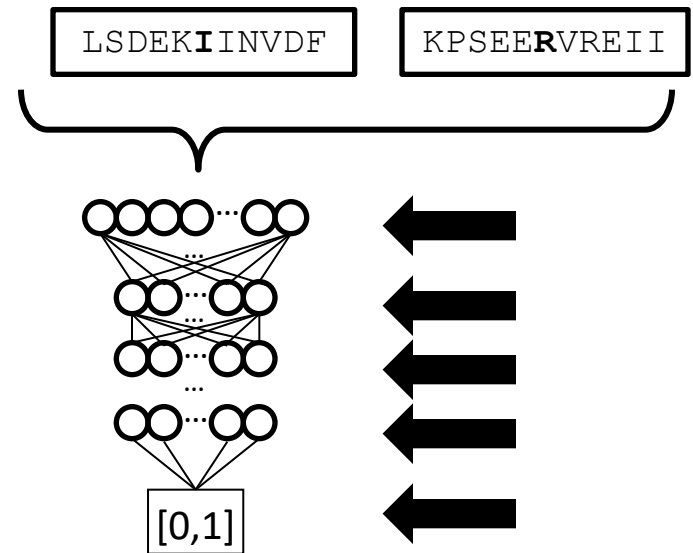


1. **Weights are learned layer by layer via unsupervised learning.**
2. **Final layer is learned as a supervised neural network.**
3. **All weights are fine-tuned using supervised back propagation.**

Specific Implementation on GPU

Speed up training by
CUDAMat and GPUs

Train DNs with over 1M
parameters in about an
hour

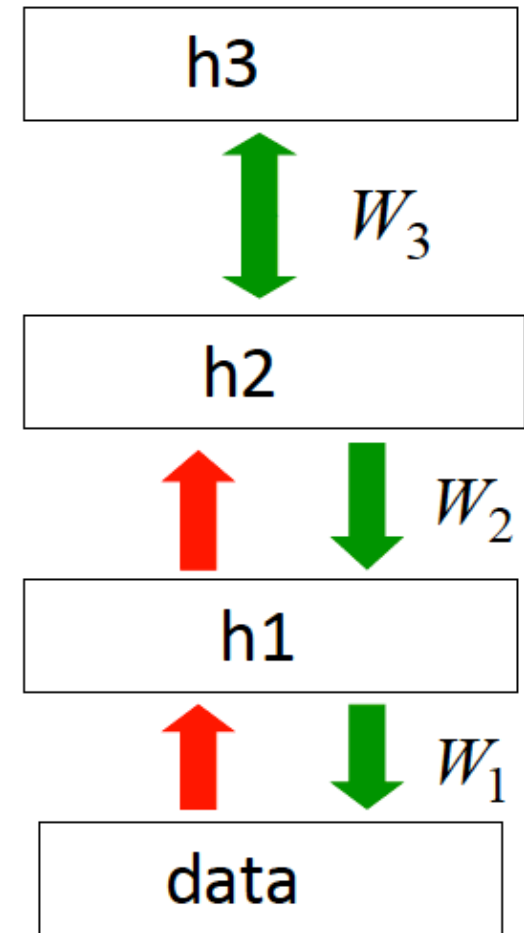


How to generate from the model

- To generate data:
 - Get an equilibrium sample from the top-level RBM by performing alternating Gibbs sampling for a long time.
 - Perform a top-down pass to get states for all the other layers.

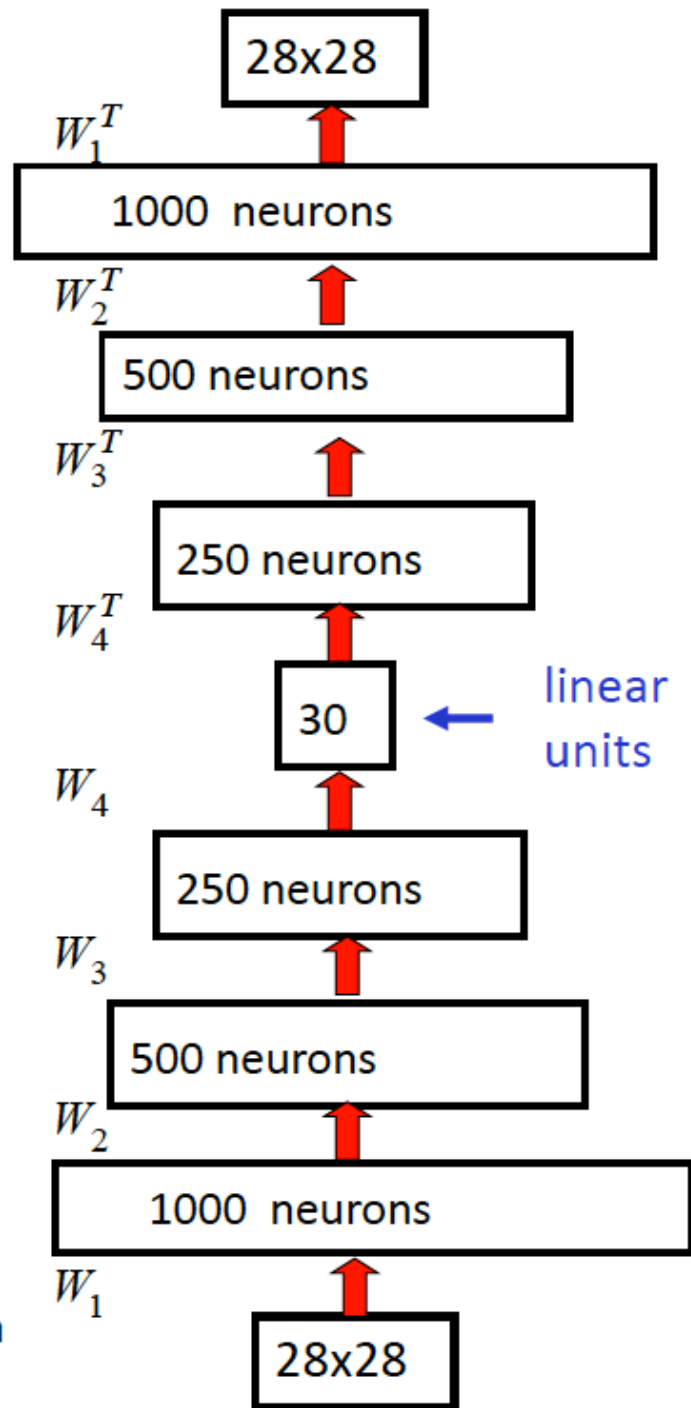
So the lower level bottom-up connections are not part of the generative model. They are just used for inference.

Bonus when modeling $p(x)$, we can see what the model believes in



Deep Autoencoders

- They always looked like a really nice way to do non-linear dimensionality reduction:
 - But it is **very** difficult to optimize deep autoencoders using backpropagation.
- We now have a much better way to optimize them:
 - First train a stack of 4 RBM's
 - Then “unroll” them.
 - Then fine-tune with backprop.



Applications: A model of digit recognition

- Classify digits (0 – 9)
- Input is a 28x28 image from MNIST (training 60k, test 10k examples)



Applications: A model of digit recognition

This is work from Hinton et al., 2006

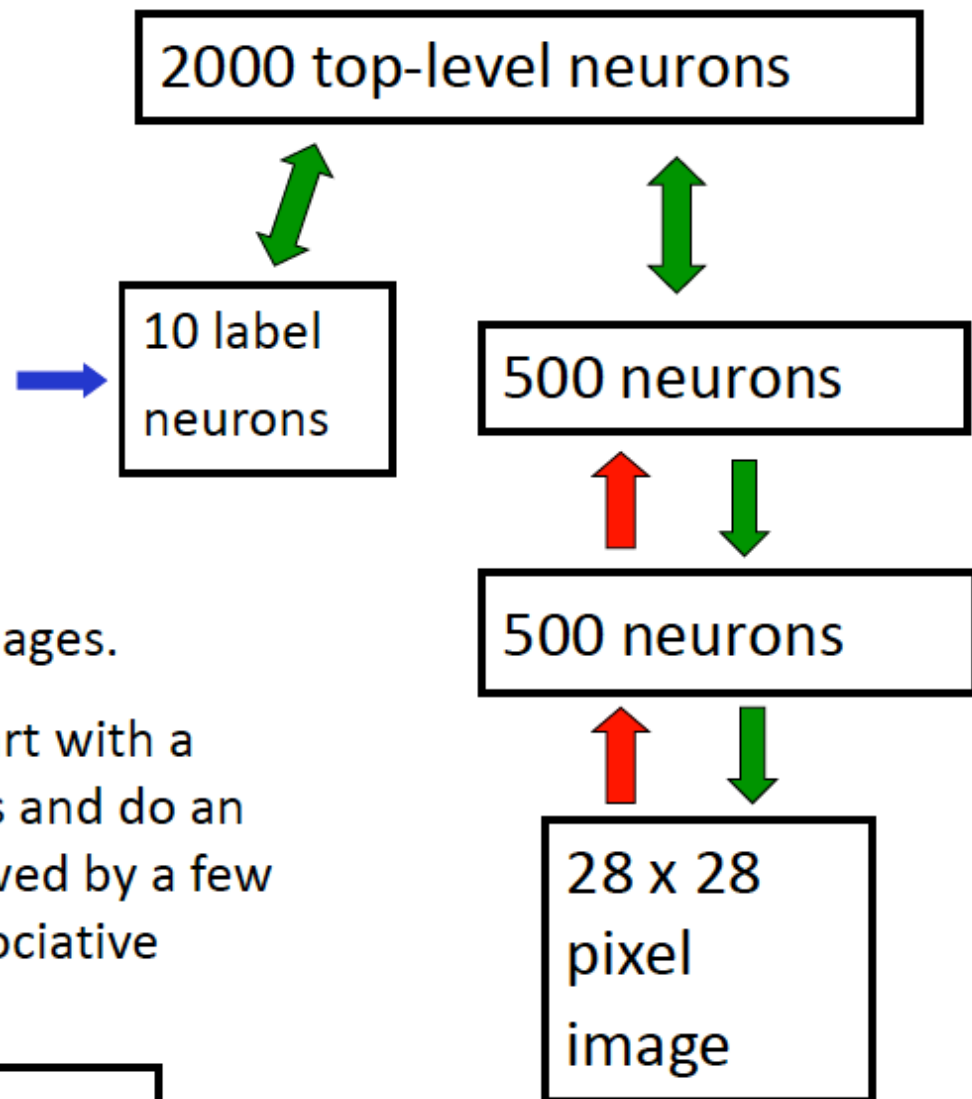
The top two layers form an associative memory whose energy landscape models the low dimensional manifolds of the digits.

The energy valleys have names

The model learns to generate combinations of labels and images.

To perform recognition we start with a neutral state of the label units and do an up-pass from the image followed by a few iterations of the top-level associative memory.

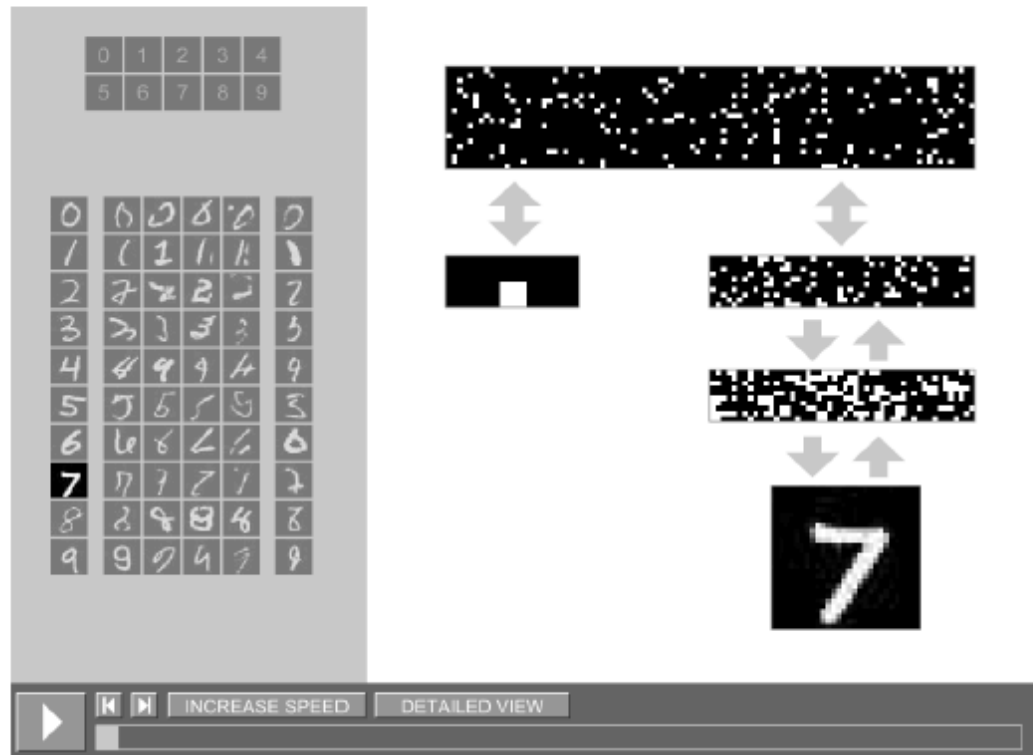
Matlab/Octave code available at <http://www.cs.utoronto.ca/~hinton/>



Slide modified from Hinton, 2007

Model in action

Hinton has provided an excellent way to view the model in action...



Demo:

<http://www.cs.toronto.edu/~hinton/digits.html>

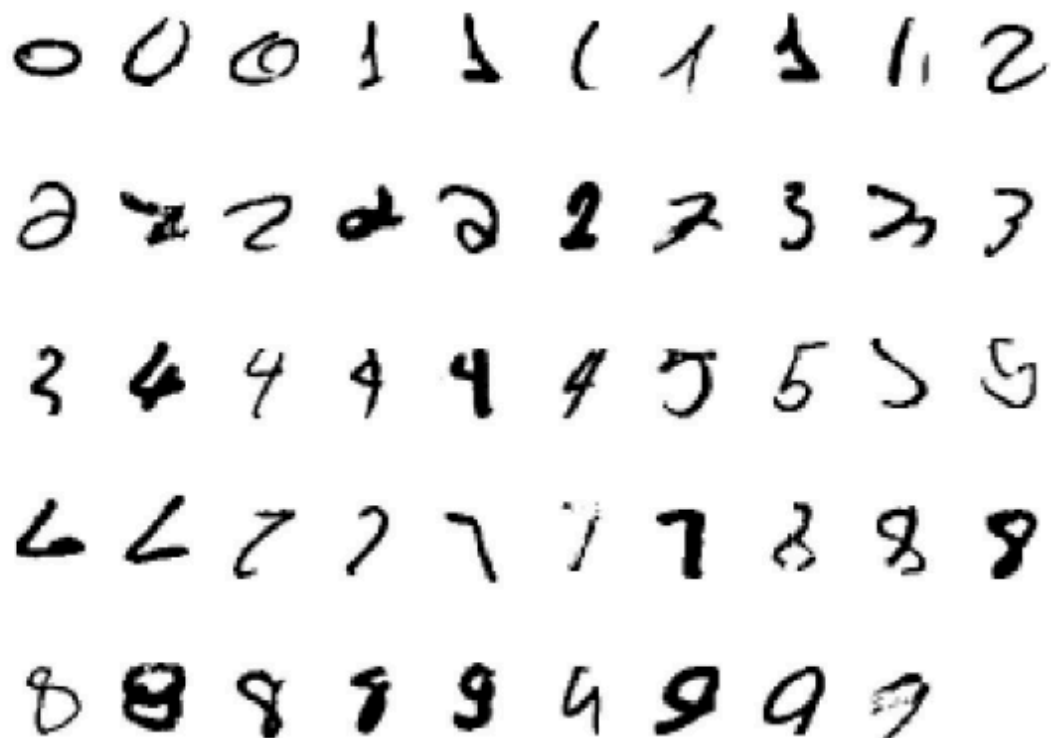
More Digits

Samples generated by letting the associative memory run with one label clamped. There are 1000 iterations of alternating Gibbs sampling between samples.



Even More Digits

Examples of correctly recognized handwritten digits that the neural network had never seen before

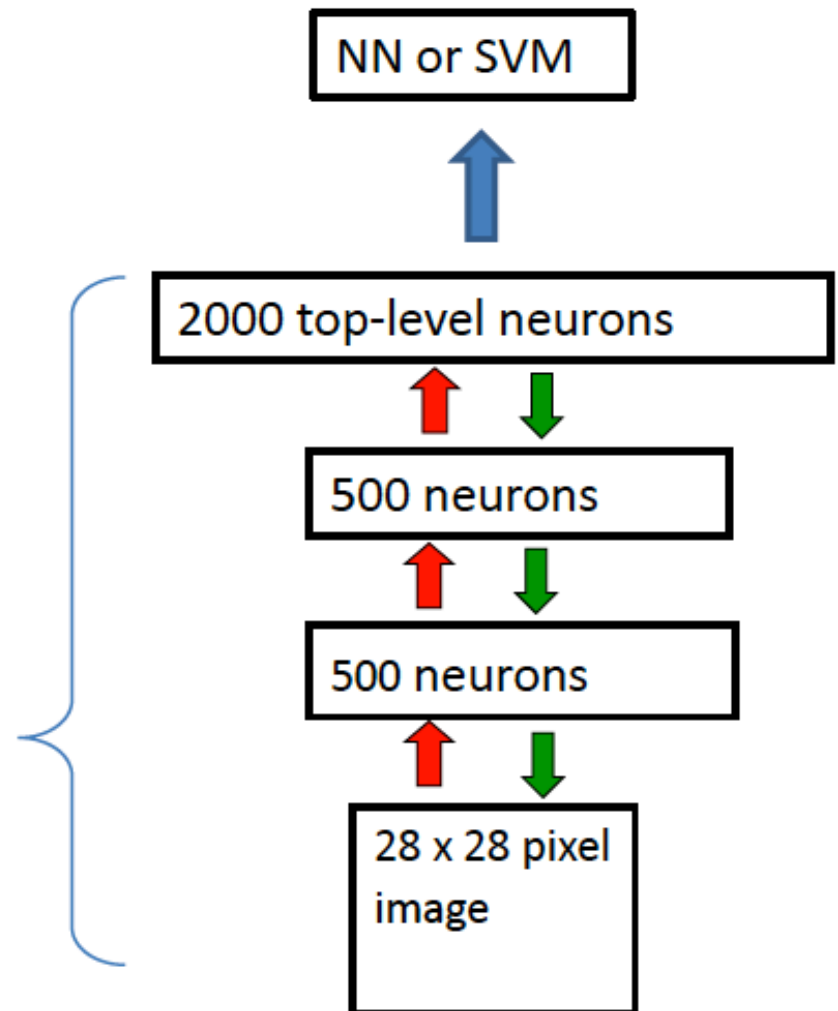


Extensions

Do classification.

One way (probably not the best), train generative model with labeled/unlabeled data

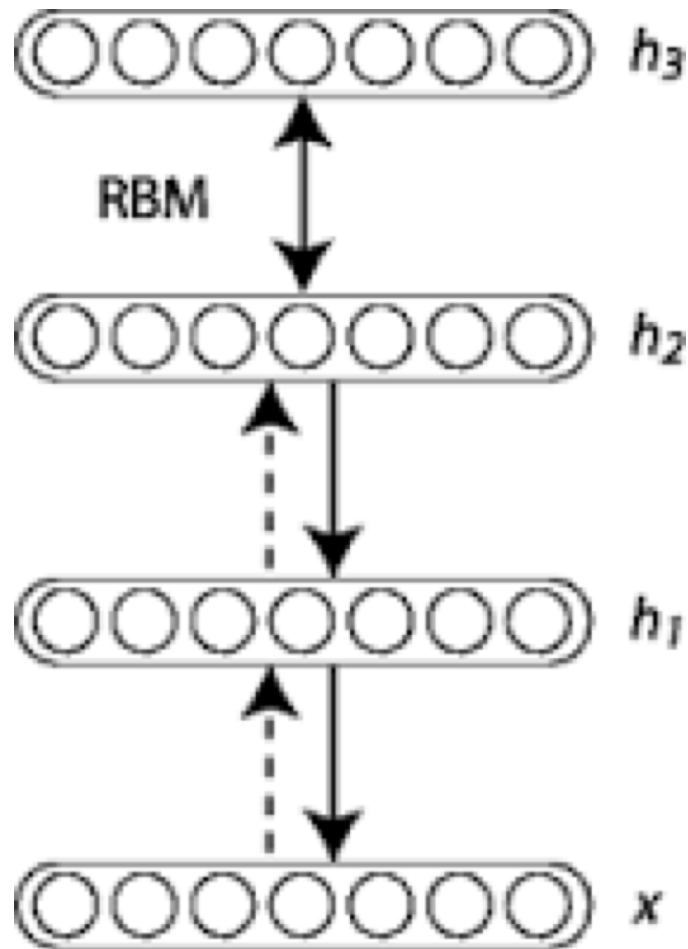
Then train a NN on higher dimensional representation.



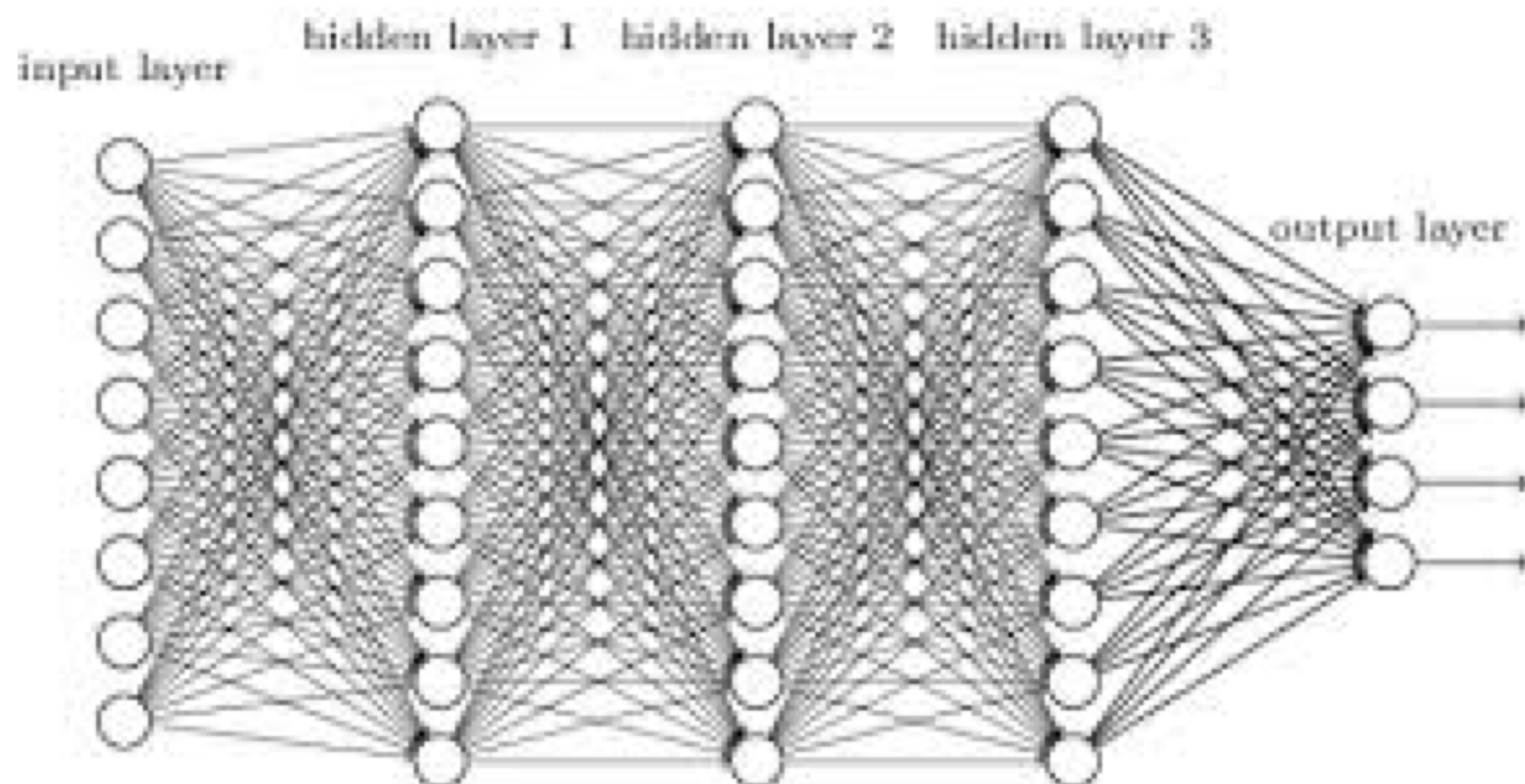
Various Deep Learning Architectures

- **Deep belief network**
- **Deep neural networks**
- **Deep autoencoder**
- **Deep convolution networks**
- **Deep residual network**
- **Deep recurrent network**

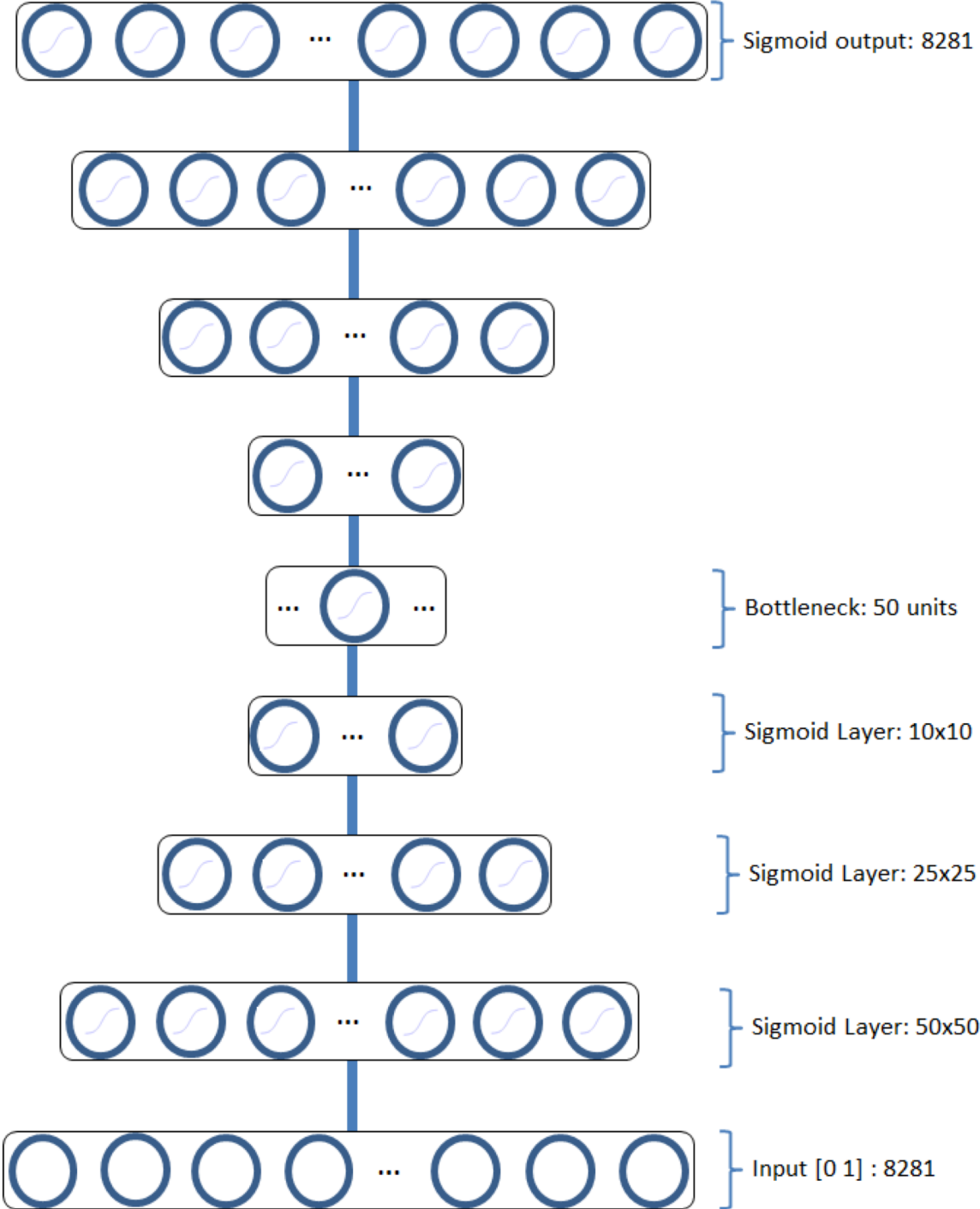
Deep Belief Network



Deep neural network

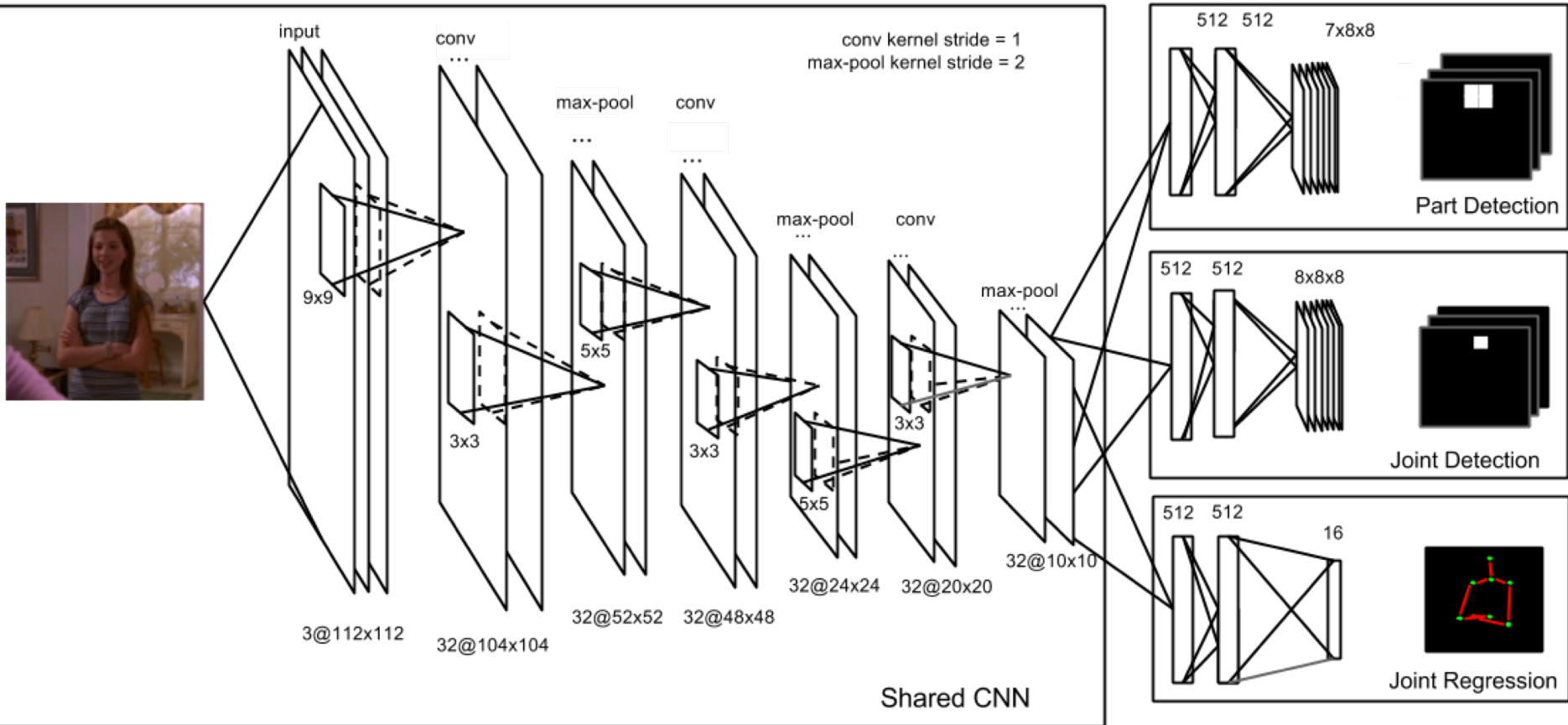


Deep AutoEncoder

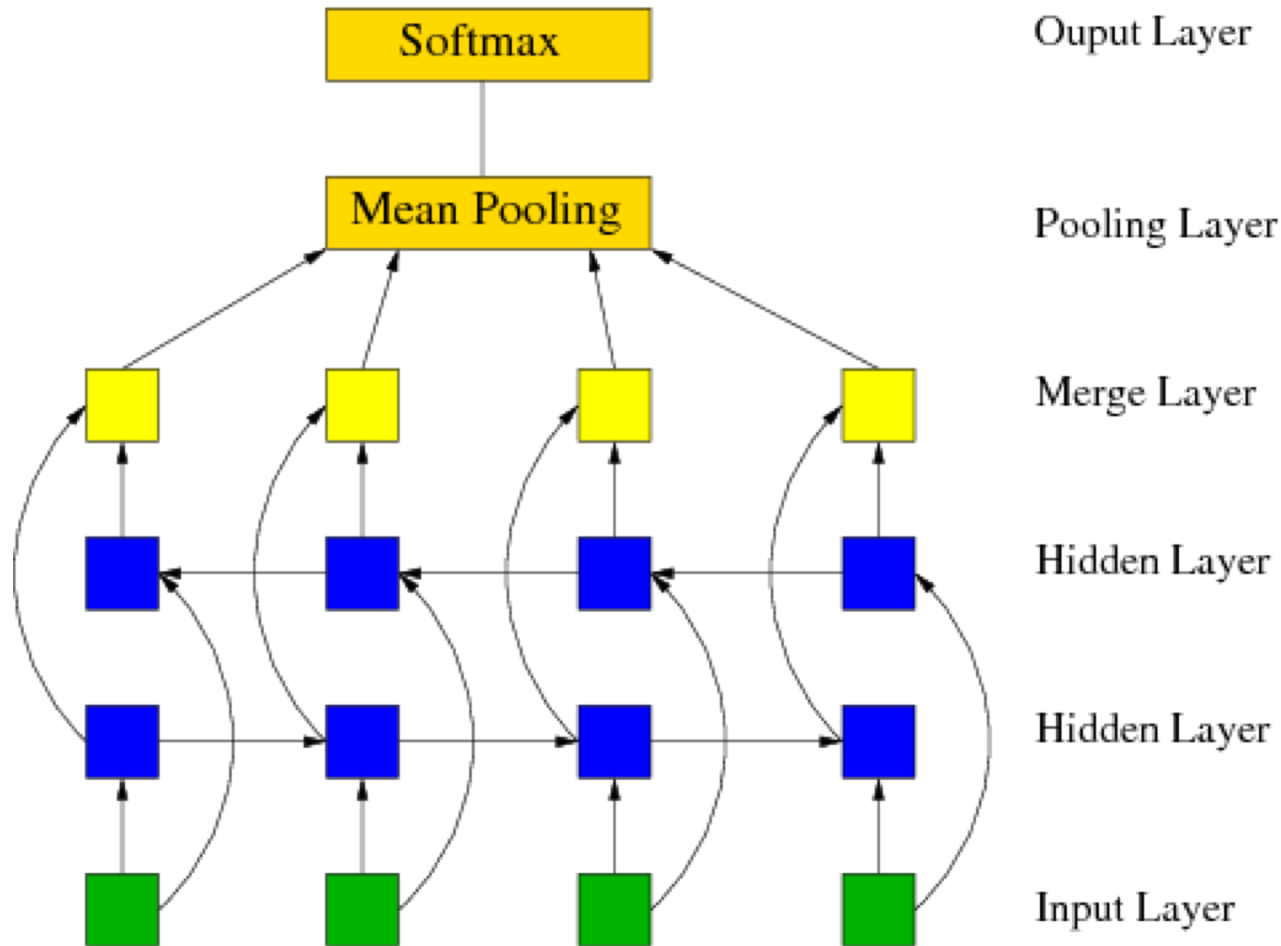


Deep Convolutional AutoEncoder

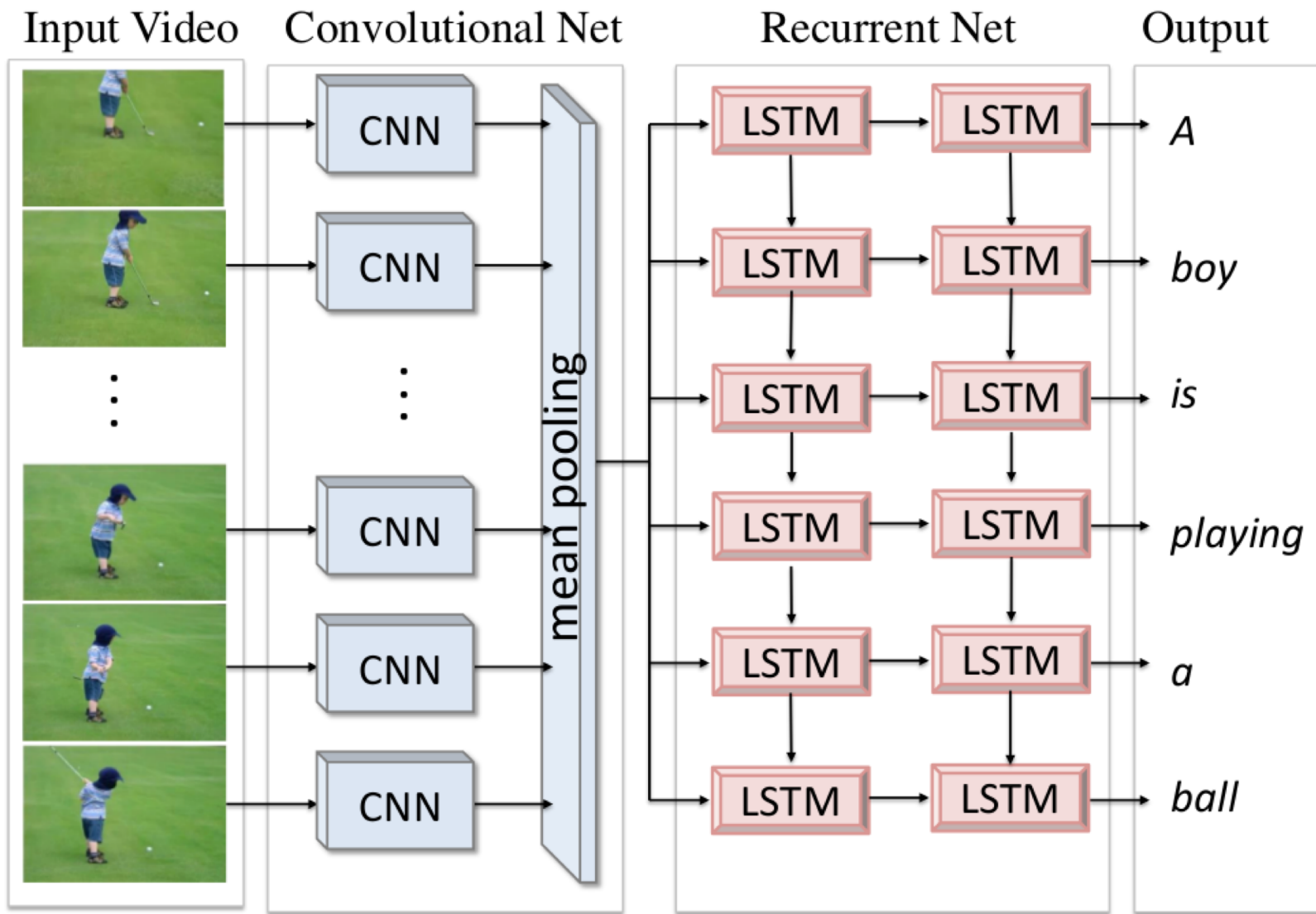
Deep Convolution Neural Network



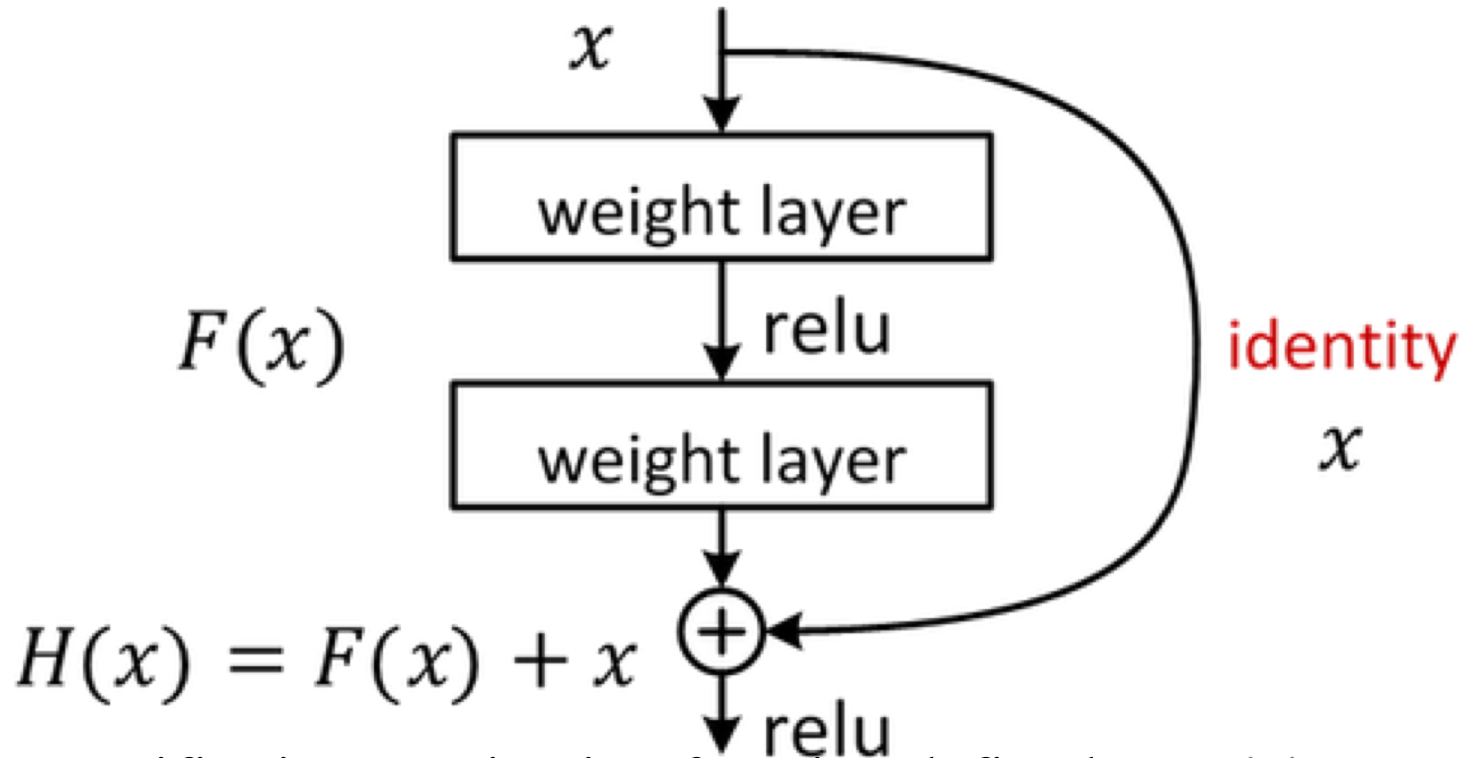
Deep Recurrent Neural Network



An Example of Network Combination



Deep Residual Network



the rectifier is an activation function defined as $f(x) = \max(0, x)$

A unit employing the rectifier is also called a rectified linear unit (ReLU)

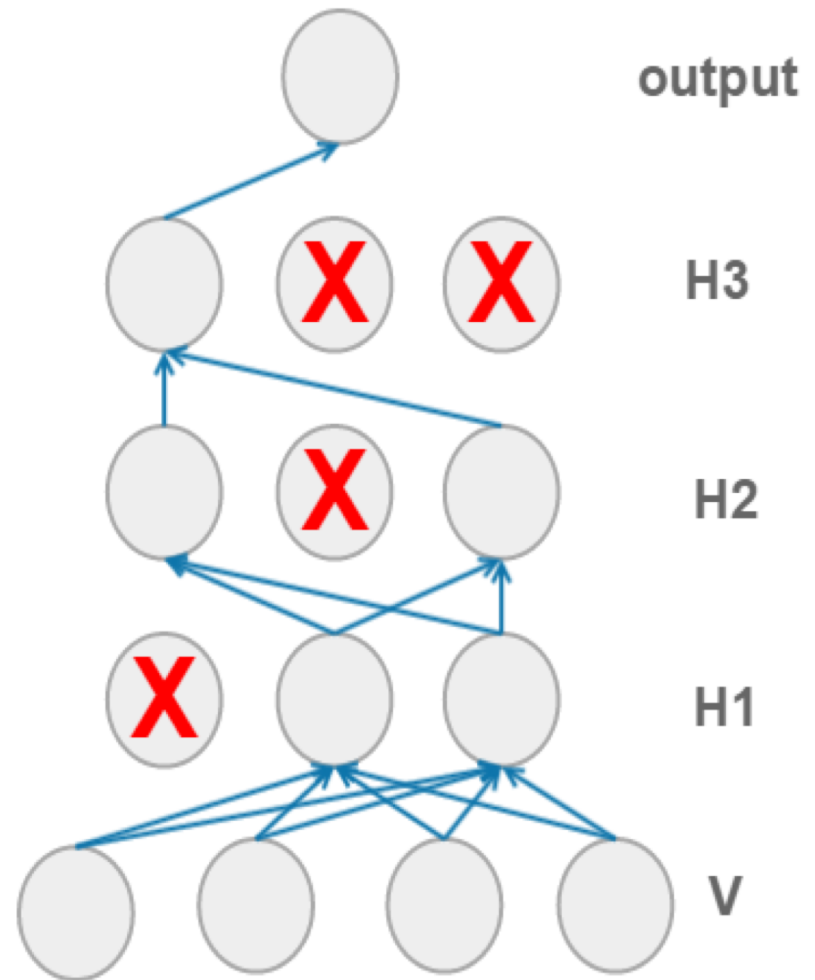
Generative-Adversarial Network (GAN)

Inception Network

Capsule Network

Dropout

- Prevent from over-fitting
- Prevent units from co-adapting
- Training: remove randomly selected units according to a rate (0.5)
- Testing: multiply all the units with dropout rate



- Pylearn2
- Theano
- Caffe
- Torch
- Cuda-convnet
- Deeplearning4j
- Keras

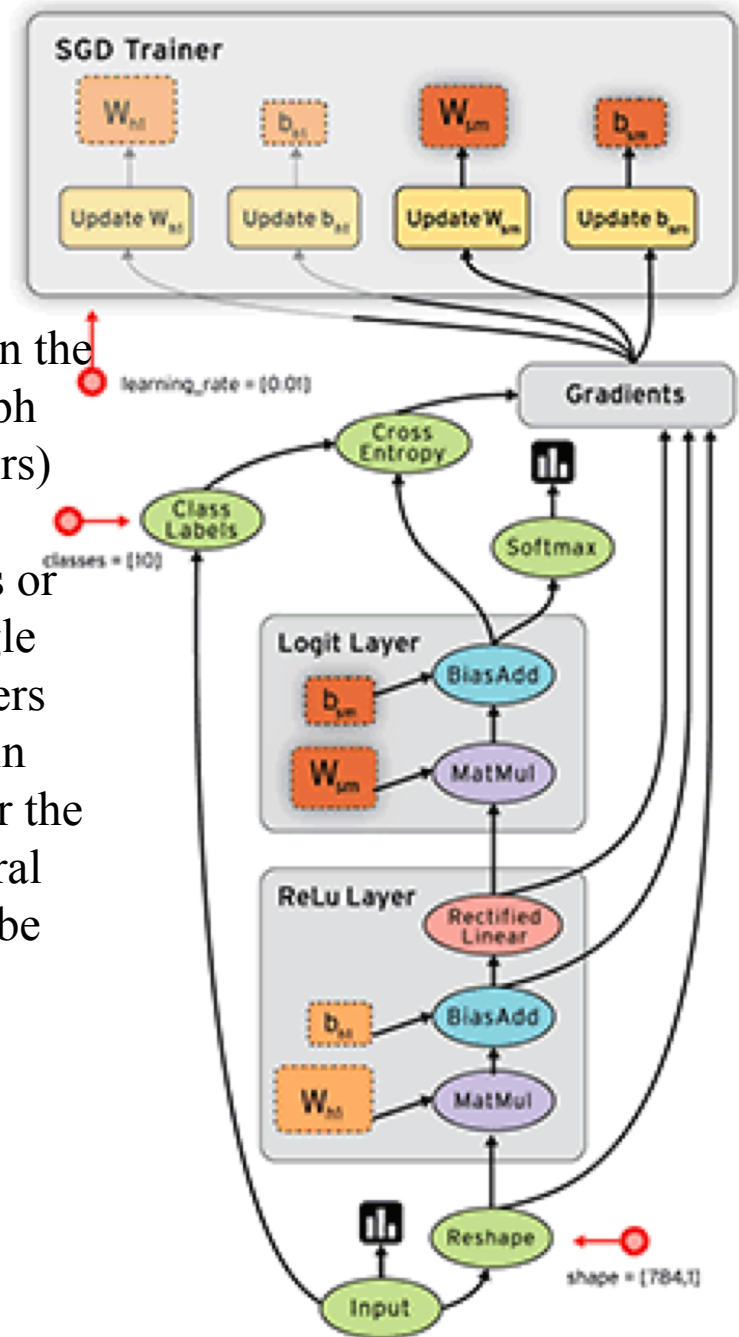
Deep Learning Tools

Framework	License	Core language	Binding(s)	CPU	GPU	Open source	Training	Pretrained models	Development
Caffe	BSD	C++	Python, MATLAB	✓	✓	✓	✓	✓	distributed
cuda-convnet [7]	unspecified	C++	Python		✓	✓	✓		discontinued
Decaf [2]	BSD	Python		✓		✓	✓	✓	discontinued
OverFeat [9]	unspecified	Lua	C++,Python	✓				✓	centralized
Theano/Pylearn2 [4]	BSD	Python		✓	✓	✓	✓		distributed
Torch7 [1]	BSD	Lua		✓	✓	✓	✓		distributed

Google's TensorFlow

TensorFlow™ is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well.

<https://www.tensorflow.org/>



Acknowledgements

- Geoffery Hinton's slides
- Jesse Eickholt's slides
- [Images.google.com](https://images.google.com)