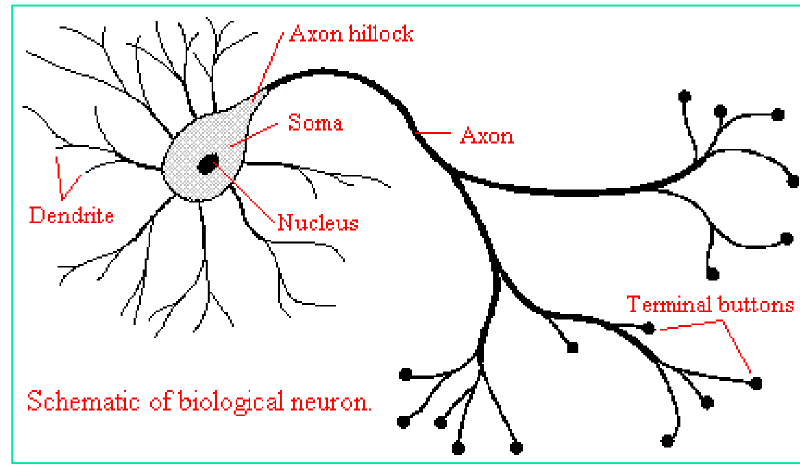
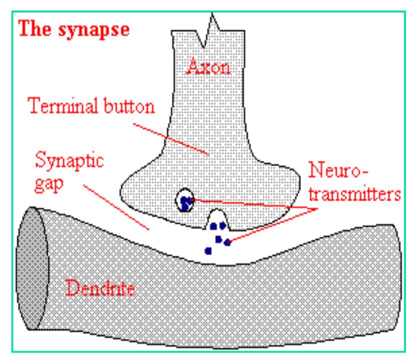
Lecture No. 7 – Artificial Neural Networks

- Overview
- Classification by Backpropagation
- Deep Neural Networks

Artificial Neural Networks
Biological Motivation

- Human brain contains about 10^{11} neurons
- Each neuron is connected, on average, to 10^4 other neurons
- Neuron switching times: about 10^{-3} seconds (computers: 10^{-10} seconds)
- Complex human decisions (e.g., face recognition): **very fast (~0.1 sec.)**
- Massive Parallel Computation



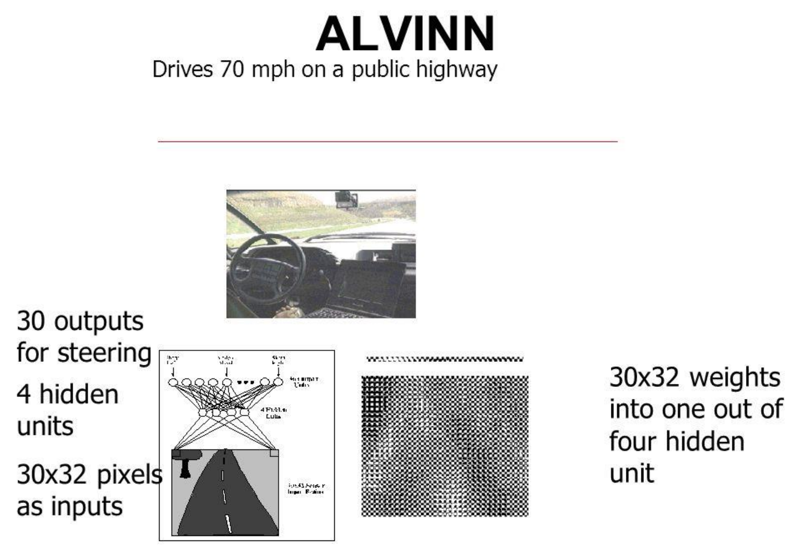


Properties of Artificial Neural Networks

- Many neural-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically
- Basic idea
  - Model the target as a nonlinear function of multiple features (also known as …)

Neural Network Representation Example: Autonomous Vehicle

- Input image: 30 X 32 = 960 pixels
- Each input node is associated with one pixel
- There are four hidden units and 30 output units
- Each input node is connected to each hidden unit with a positive / negative weight
- Each output unit is associated with a command (left, straight, right, etc.)



ALVINN
(**A**utonomous **L**and **V**ehicle **I**n a **N**eural **N**etwork)

Appropriate Problems for ANN

- Input and target attributes can be discrete or real-valued (**have to be** numeric)
- Output may include more than one target attribute (“a vector of attributes”)
- The training data may contain errors
- Each target attribute is a smooth, continuous function of input attributes
- Form of target function is unknown
- Long training times are acceptable
- Human readability of results is unimportant

Sample Applications of ANN
(Define inputs and outputs)

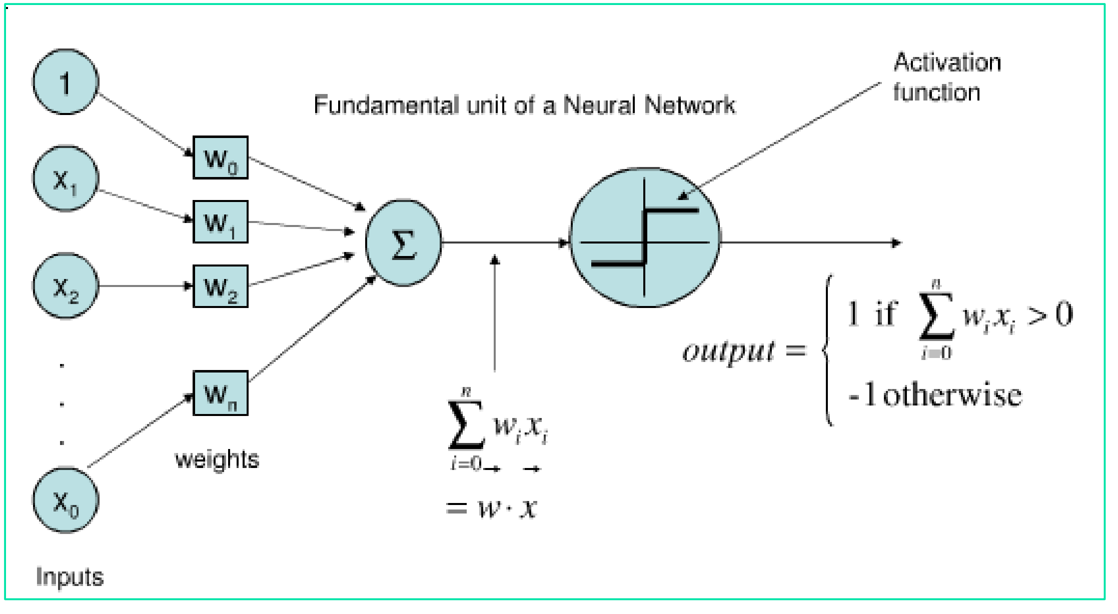
- Optical Character Recognition (OCR)
- Voice Recognition
- Image Classification
- Information Retrieval
- Financial Prediction
- Natural Language Processing (Word Embedding)

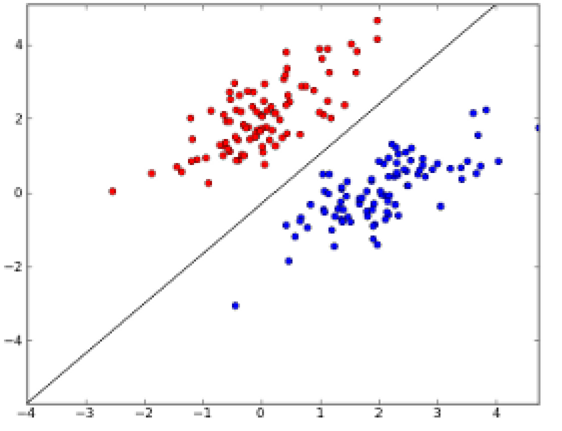
Lecture No. 7 – Artificial Neural Networks

- Overview
- Classification by Backpropagation
- Deep Neural Networks

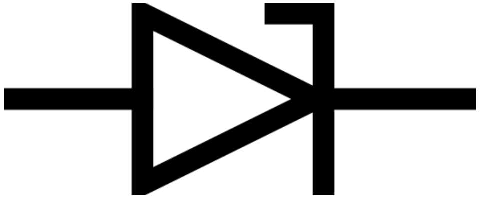
Classification by Backpropagation

- A neural network: A set of connected input/output units (*neurons*) where each connection has a **weight** associated with it
- During the learning phase, the **network learns by adjusting the weights**
- Also referred to as **connectionist learning**

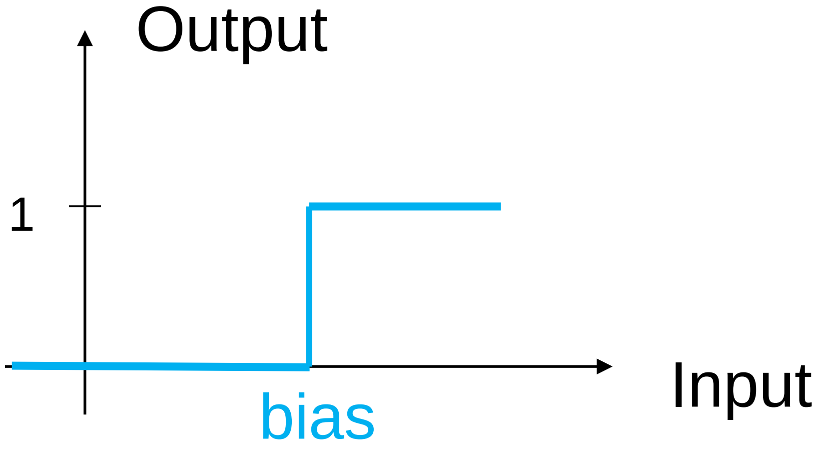




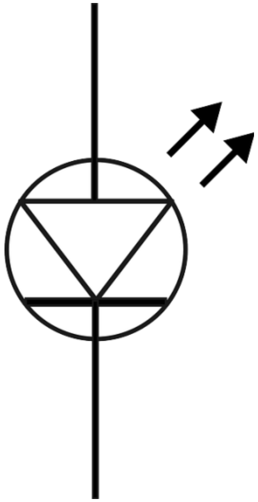
Node Biases



Recall: A node's output is weighted function of its inputs <u>and a 'bias' term</u>



These *biases* also need to be learned!



Training Biases (Θ's)

A node's output (assume 'step function' for simplicity)

$$1 \text{ if } W_1 X_1 + W_2 X_2 + \ldots + W_n X_n \geq \Theta$$

$$0 \text{ otherwise}$$

<u>Rewriting</u>

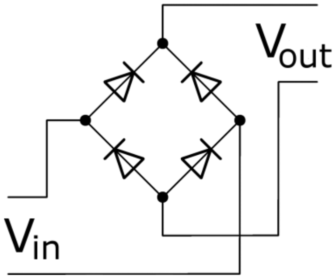
$$W_1 X_1 + W_2 X_2 + \ldots + W_n X_n - \Theta \geq 0$$

$$W_1 X_1 + W_2 X_2 + \ldots + W_n X_n + \Theta \times (-1) \geq 0$$

'activation'

weight

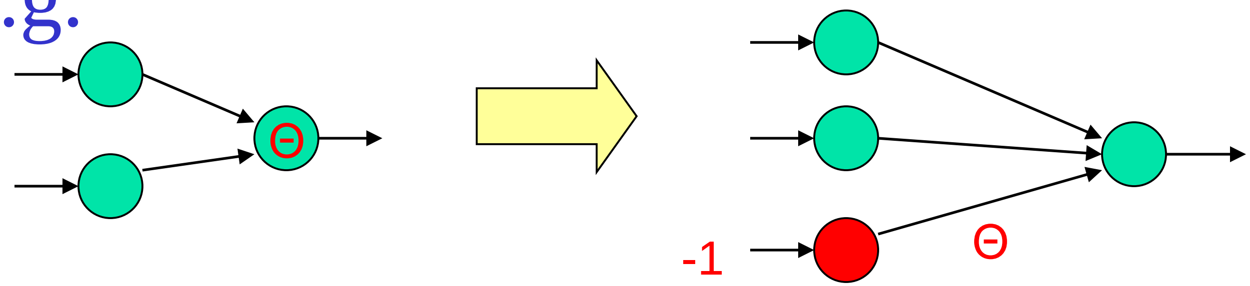
Training Biases (cont.)



Hence, add another unit whose activation is always -1

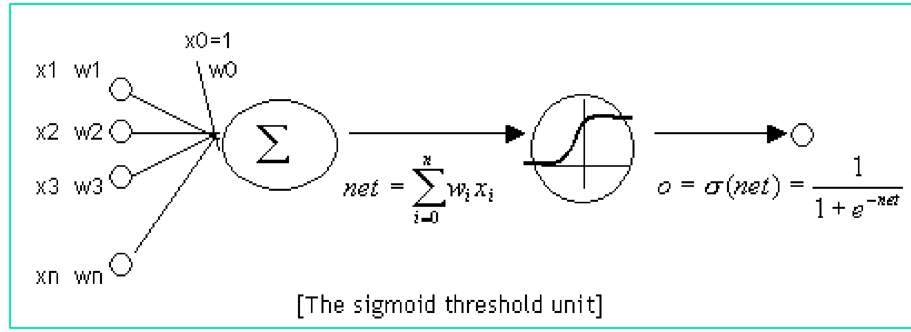
The bias is then just another weight!

E.g.



Sigmoid Activation Units

(Input: ? Output: ?)



[The sigmoid threshold unit]

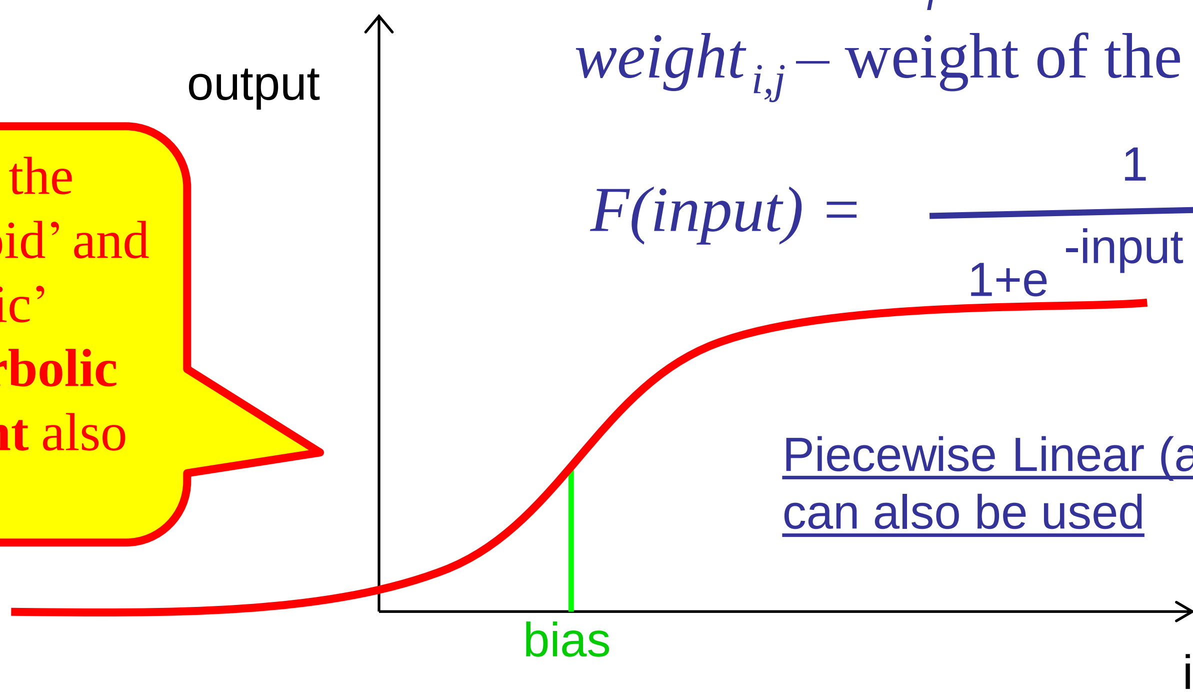
Individual Units' Computation

$$output_j = F(\sum_i weight_{i,j} \times output_i + bias_j)$$

$weight_{i,j}$ – weight of the connection $i \rightarrow j$

$$F(input) = \frac{1}{1+e^{-input}}$$

Called the 'sigmoid' and 'logistic' (**hyperbolic tangent** also used)

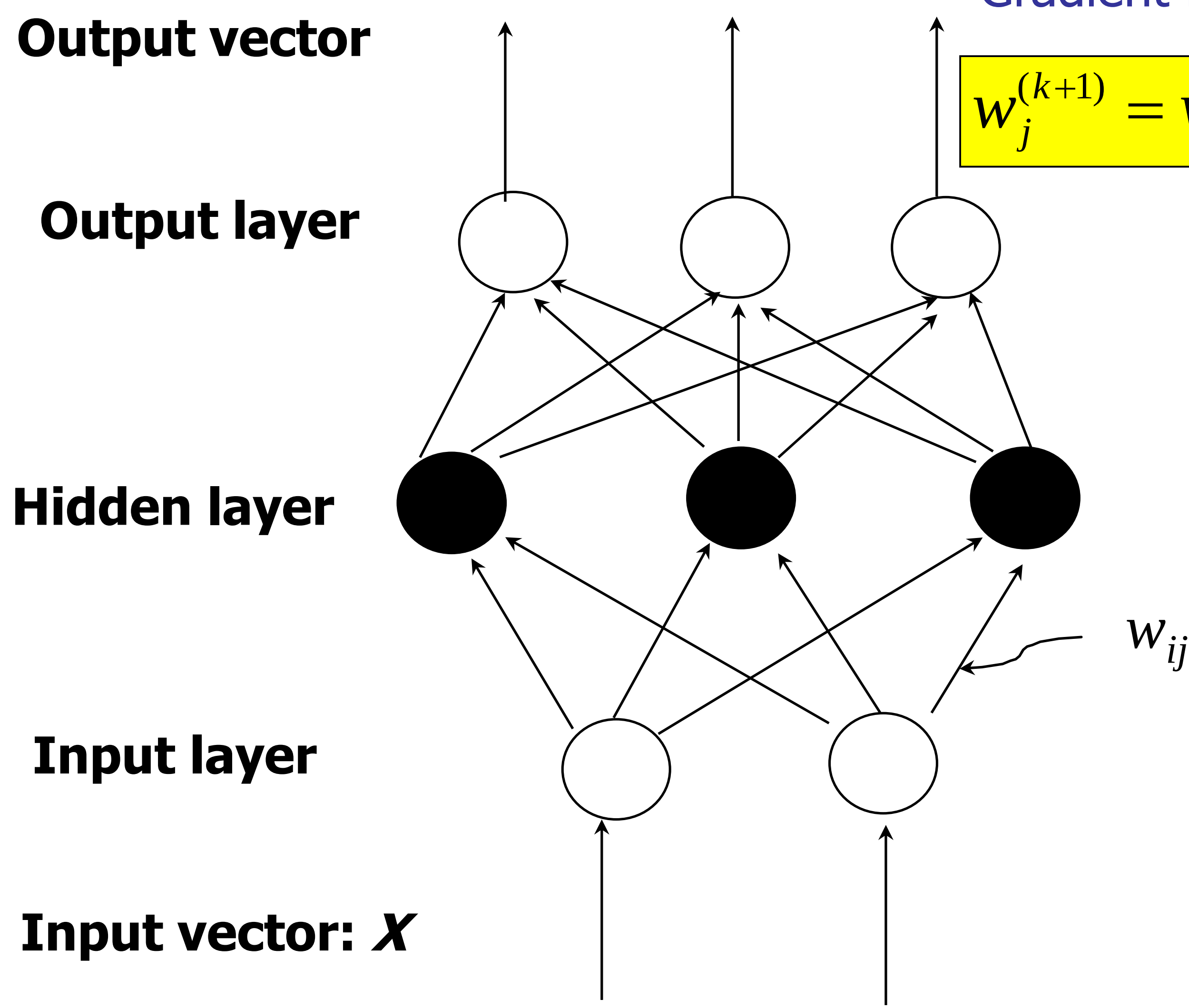


Piecewise Linear (and Gaussian) nodes can also be used

Neural Network as a Classifier

- Weakness
  - Long training time
  - Require a number of parameters typically best determined empirically
  - Poor interpretability
- Strength
  - High tolerance to noisy data
  - Ability to classify untrained patterns
  - Well-suited for continuous-valued inputs and outputs
  - Successful on an array of real-world data, e.g., hand-written letters
  - Algorithms are inherently parallel
  - Techniques have recently been developed for the extraction of rules from trained neural networks

A Multi-Layer Feed-Forward Neural Network



Gradient Descent Rule:

$$\boxed{w_j^{(k+1)} = w_j^{(k)} + \lambda(y_i - \hat{y}_i^{(k)})x_{ij}}$$

k – iteration

x_{ij} - input vector

y_i – target value

λ - learning rate (e.g., 0.05)

How A Multi-Layer Neural Network Works

- The **inputs** to the network correspond to the attributes measured for each training tuple
- Inputs are fed simultaneously into the units making up the **input layer**
- They are then weighted and fed simultaneously to a **hidden layer**
- The number of hidden layers is arbitrary (one, two, or more)
- The weighted outputs of the last hidden layer are input to units making up the **output layer**, which emits the network's prediction
- The network is **feed-forward**: None of the weights cycles back to an input unit or to an output unit of a previous layer
- From a statistical point of view, networks perform **nonlinear regression**: Given enough hidden units and enough training samples, they can closely approximate any function

Defining a Network Topology

- Decide the **network topology:** Specify # of units in the *input layer*, # of *hidden layers* (if > 1), # of units in *each hidden layer*, and # of units in the *output layer*
- Normalize the input values for each attribute measured in the training tuples to [0.0—1.0]
- One **input** unit per domain value, each initialized to 0
- **Output**, if for classification and more than two classes, one output unit per class is used
- Once a network has been trained and its accuracy is **unacceptable**, repeat the training process with a *different network topology* or a *different set of initial weights*

Backpropagation

- Iteratively process a set of training tuples & compare the network's prediction with the actual known target value
- For each training tuple, the weights are modified to **minimize the mean squared error** between the network's prediction and the actual target value
- Modifications are made in the "**backwards**" direction: from the output layer, through each hidden layer down to the first hidden layer, hence "**backpropagation**"
- Steps
  - Initialize weights to small random numbers, associated with biases
  - Propagate the inputs forward (by applying activation function)
  - Backpropagate the error (by updating weights and biases)
  - Terminating condition (when error is very small, etc.)

Backpropagation Algorithm

Algorithm: Backpropagation. Neural network learning for classification or prediction, using the backpropagation algorithm.

Input:

- D, a data set consisting of the training tuples and their associated target values;
- l, the learning rate;
- *network*, a multilayer feed-forward network.

Output: A trained neural network.

Method:

Initialize all weights and biases in *network*;
`while` terminating condition is not satisfied {
    `for` each training tuple X in D {
        // Propogate the inputs forward:
        `for` each input layer unit j {
            $O_j = I_j$; // output of an input unit is its actual input value
        `for` each hidden or output layer unit j {
            $I_j = \Sigma_i w_{ij} O_i + \theta_j$; // compute the net input of unit j with respect to the previous layer, i
            $O_j = \frac{1}{1+e^{-I_j}}$; } // compute the output of each unit j
        // Backpropogate the errors;
        `for` each unit j in the output layer
            $Err_j = O_j(1 - O_j)(T_j - O_j)$; // compute the error
        `for` each unit j in the hidden layers, from the last to the first hidden layer
            $Err_j = O_j(1 - O_j)\Sigma_k Err_k w_{jk}$; // compute the error with respect to the next higher layer, k
        `for` each weight w_{ij} in network {
            $\Delta w_{ij} = (l) Err_j O_i$; // weight increment
            $w_{ij} = w_{ij} + \Delta w_{ij}$; } // weight update
        `for` each bias θ_j in network {
            $\Delta\theta_j = (l) Err_j$; // bias increment
            $\theta_j = \theta_j + \Delta\theta_j$; } // bias update
        } }

Weight Space

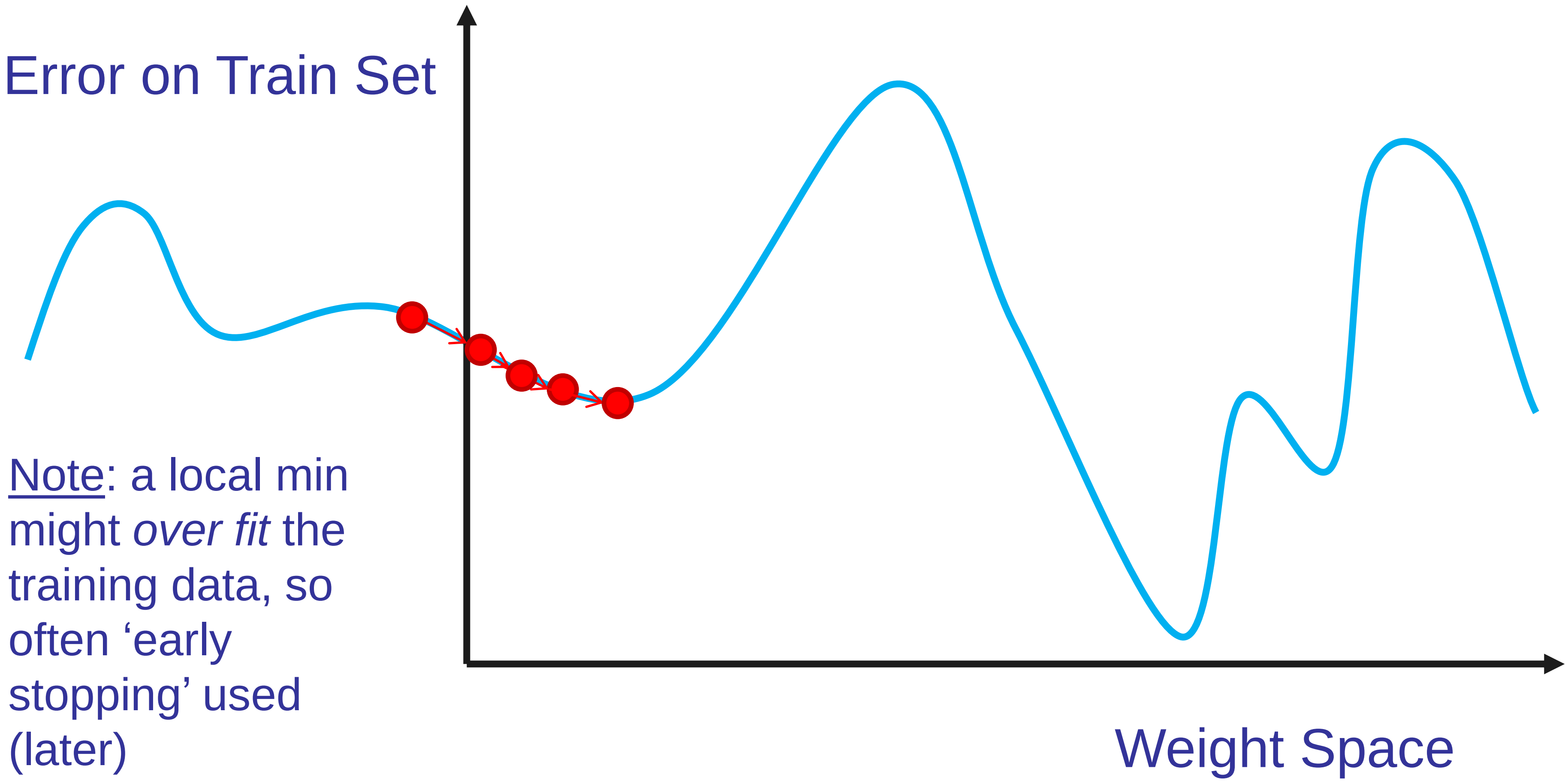
- Given a neural-network layout, the weights and biases are free parameters that define a *space*
- Each point in this *Weight Space* specifies a network

  weight space is a *continuous* space we search

- Associated with each point is an *error rate*, **E**, over the training data
- Backprop performs gradient descent in weight space

Backprop Seeks LOCAL Minima

(in a *continuous* space)



<u>Note</u>: a local min might *over fit* the training data, so often ‘early stopping’ used (later)

The Gradient-Descent Rule

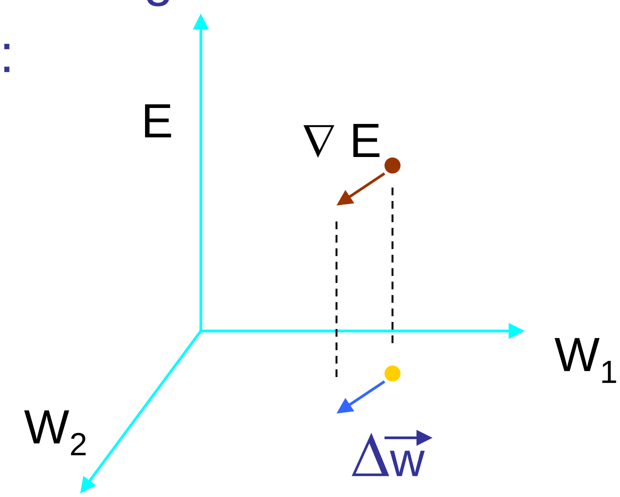
$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \cdots, \frac{\partial E}{\partial w_N} \right]$$

The 'gradient'

This is a N+1 dimensional vector (ie, the 'slope' in weight space)

Since we want to **reduce** errors, we want to go 'down hill'

We'll take a finite step in weight space:



$$\Delta \vec{w} = - \eta \nabla E(\vec{w})$$

'delta' = the change to $\vec{w}$

$$\text{or } \Delta w_i = - \eta \frac{\partial E}{\partial w_i}$$

'On Line' vs. 'Batch' Backprop

- Technically, we should look at the error gradient for the entire training set, before taking a step in weight space ('batch' backprop)

- *However*, in practice we take a step after each example ('on-line' backprop)
  - Much faster convergence (learn after each example)
  - Called 'stochastic' gradient descent
  - Stochastic gradient descent quite popular at Google, Facebook, Microsoft, etc. due to easy parallelism

Gradient Descent for the Perceptron

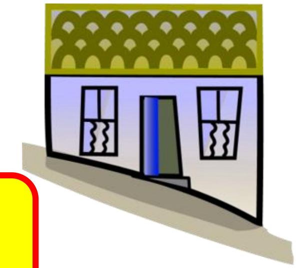
(for the simple case of <u>linear</u> output units)

$$\text{Error} \equiv \tfrac{1}{2} \times (\, T - o \,)^2$$

Network's **o**utput

Teacher's answer
(a <u>constant</u> wrt the weights)

$$\frac{\partial E}{\partial W_k} = (T - o) \frac{\partial (T - o)}{\partial W_k} = -\,(T - o) \frac{\partial o}{\partial W_k}$$



Continuation of Derivation

Stick in formula for output

$$\frac{\partial E}{\partial W_k} = -(T - o)\,\frac{\partial(\sum w_j \times x_j)}{\partial W_k}$$

$$= -(T - o)\, x_k$$

Recall $\Delta W_k \equiv -\eta \dfrac{\partial E}{\partial W_k}$

So $\boxed{\Delta W_k = \eta\,(T - o)\, x_k}$ **The Perceptron Rule**

We'll use for both LINEAR and STEP-FUNCTION activation

Also known as the **delta rule** and other names (with some variation in the calculation)

Perceptron Example: Autonomous Vehicle

Training Set

Current Speed	Speed Limit	Correct Output
40	50	1 (acceleration)
55	50	0 (deceleration)
75	90	1 (acceleration)

Correct solution?

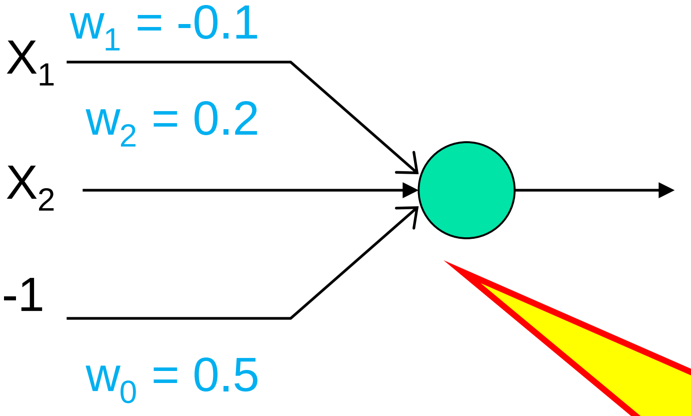
Perceptron Learning Rule

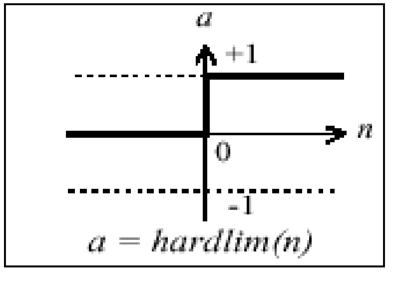
$$\Delta W_k = \eta\,(T - Out)\,x_k$$
$$\eta = 0.01$$

Out = Step Function:

Cur_Sp*w_1 + Sp_Limit*w_2 – w_0 =

40*-0.1 + 50*0.2 - 1*0.5 = 5.5 => 1





No wgt changes, since correct

Perceptron Example: Autonomous Vehicle

Training Set

Current Speed	Speed Limit	Correct Output
40	50	1 (acceleration)
55	50	0 (deceleration)
75	90	1 (acceleration)

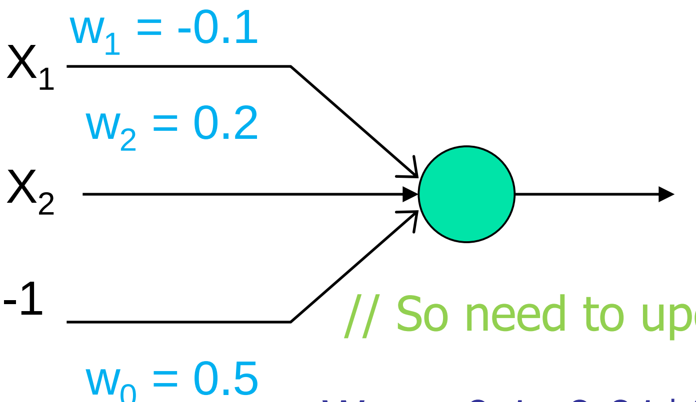
Perceptron Learning Rule

$$\Delta W_k = \eta\,(T - Out)\,x_k$$
$$\eta = 0.01$$

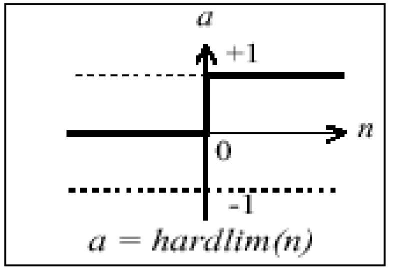
Out = Step Function:

Cur_Sp*w_1 + Sp_Limit*w_2 – w_0 =

55*-0.1 + 50*0.2 - 1*0.5 = 4 => 1



// So need to update weights



W_1 = -0.1+0.01*(-1)*55 = -0.65
W_2 = 0.2+0.01*(-1)*50 = -0.30
W_0 = 0.50+0.01*(-1)*(-1) = 0.51

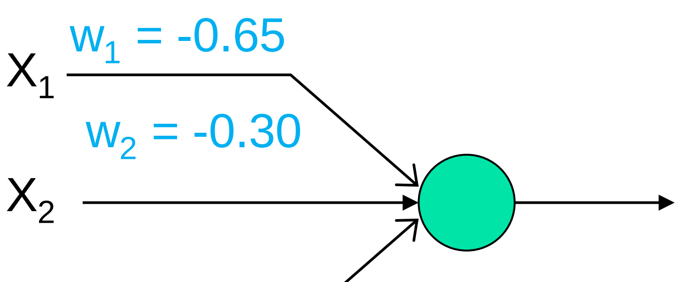
Perceptron Example: Autonomous Vehicle

Training Set

Current Speed	Speed Limit	Correct Output
40	50	1 (acceleration)
55	50	0 (deceleration)
75	90	1 (acceleration)

Perceptron Learning Rule

$$\Delta W_k = \eta\,(T - Out)\,x_k$$
$$\eta = 0.01$$



Out = Step Function:

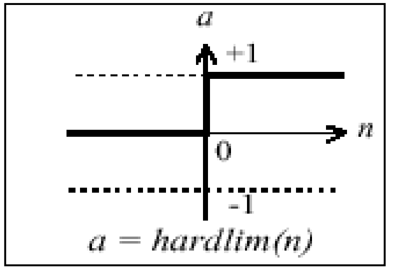
Cur_Sp*w_1 + Sp_Limit*w_2 – w_0 =

75*(-0.65) + 90*(-0.30) - 1*0.51 = -76.3

=> 0

// So need to update weights

$W_1 = -0.65+0.01*1*75 = 0.10$
$W_2 = -0.30+0.01*1*90 = 0.60$
$W_0 = 0.51+0.01*1*(-1) = 0.50$



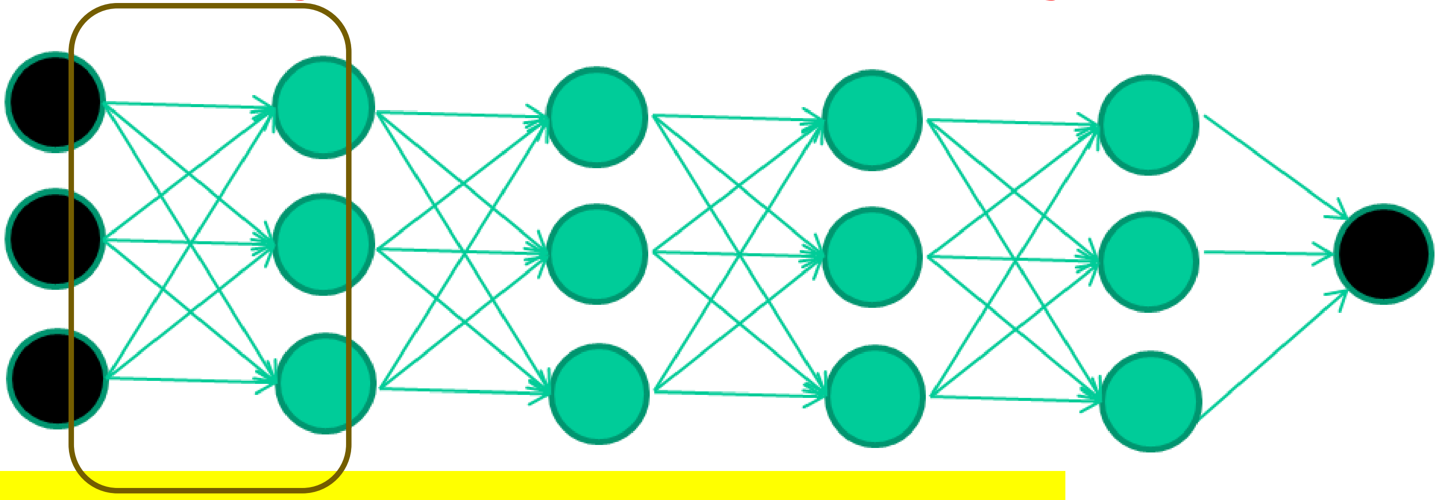
Efficiency and Interpretability

- **<u>Efficiency</u>** of backpropagation: Each epoch (one iteration through the training set) takes O(|D| * *w*), with |D| tuples and *w* weights, but # of epochs can be exponential to n, the number of inputs, in worst case
- For easier comprehension: **<u>Rule extraction</u>** by network pruning
  - Simplify the network structure by removing weighted links that have the least effect on the trained network
  - Then perform link, unit, or activation value clustering
  - The set of input and activation values are studied to derive rules describing the relationship between the input and hidden unit layers
- **<u>Sensitivity analysis</u>**: assess the impact that a given input variable has on a network output. The knowledge gained from this analysis can be represented in rules

Lecture No. 7 – Artificial Neural Networks

- Overview
- Classification by Backpropagation
- Deep Neural Networks

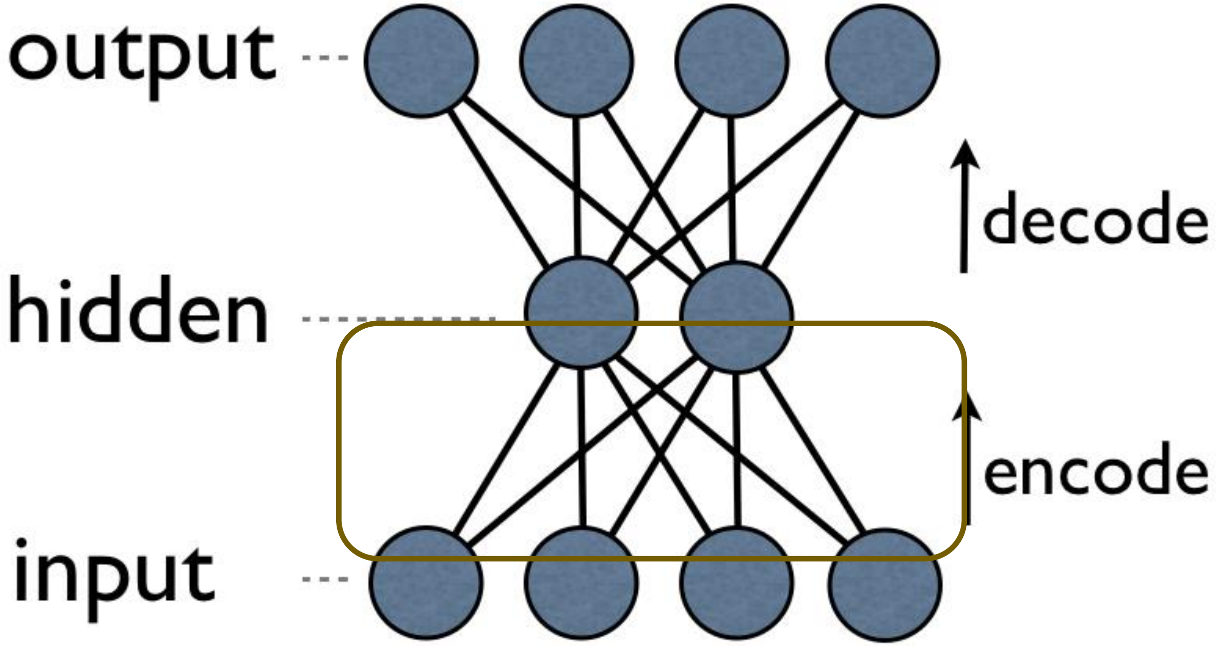
The new way to train multi-layer NNs…



EACH of the (non-output) layers is trained to be an ***auto-encoder***

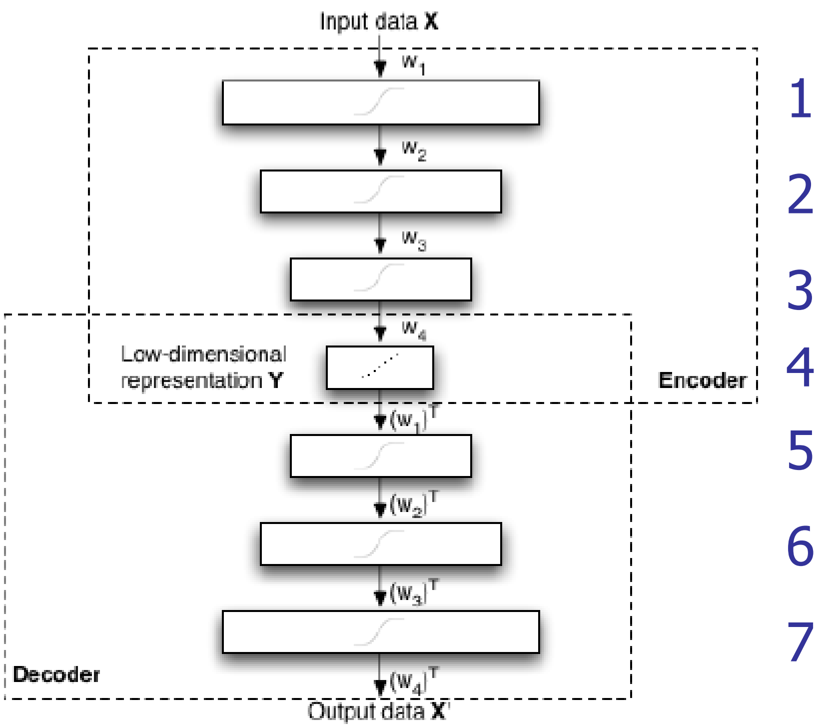
Basically, it is forced to learn good features that describe what comes from the previous layer

An auto-encoder is trained, with an absolutely standard weight-adjustment algorithm to <u>reproduce the input</u>



By making this happen with (many) fewer units than the inputs, this forces the 'hidden layer' units to become good feature detectors

Intermediate layers are each trained to be auto encoders (or similar)



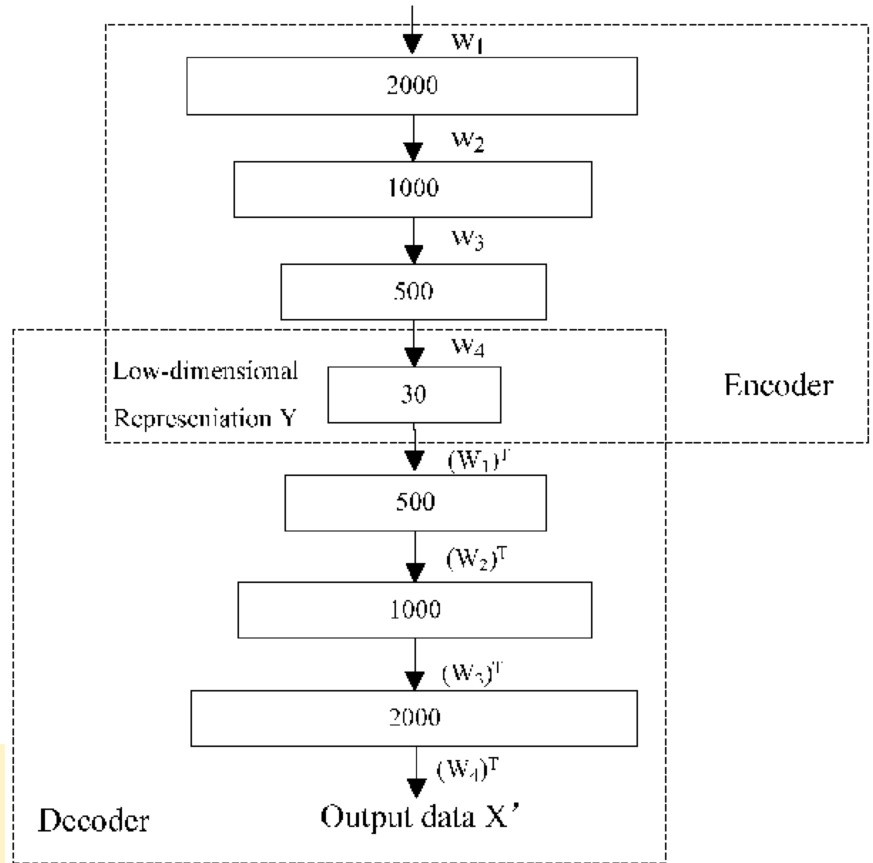


Figure 1. Structure of an autoencoder

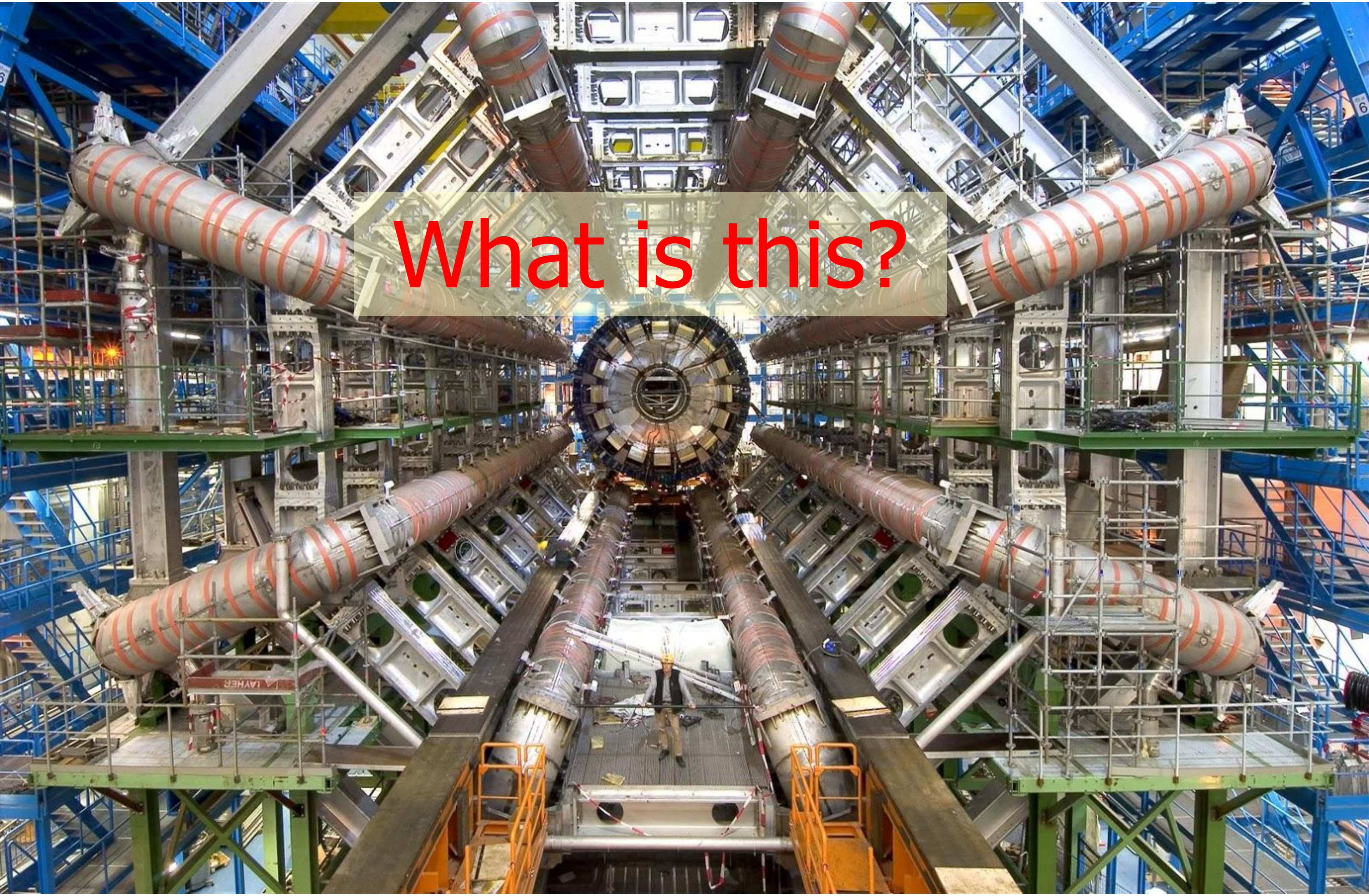
The number of units in the k-th layer is the same as that in the $(M - k + 1)$ th layer.
Aggarwal, Charu C.. *Neural Networks and Deep Learning: A Textbook*. 2018

Training Deep Neural Networks

Source: Aggarwal, 2018

- *Mini-batch stochastic gradient descent* often provides the best trade-off between stability, speed, and memory requirements.
  - Commonly used sizes of a mini-batch: 32, 64, 128, or 256.
- Use a *validation set* for tuning hyperparameters (e.g., network topology) and a separate *training set* for gradient descent . Why?
  - Hyperparameter optimization: coarse-to-fine-grained grid search
- Feature pre-processing
  - *Normalize* the values of each input feature
  - Apply *whitening*: extract a new set of de-correlated features (using … ?)
- Initialize each weight to a value drawn from a Gaussian distribution with standard deviation $\sqrt{\frac{1}{r}}$, where *r* is the number of inputs to that neuron. Why?
- Use ReLU activation function ($f(x) = x, x \geq 0, f'(x) = ?$)
- Use decaying learning rate
- Training acceleration and model compression. How?

What is this?



nature COMMUNICATIONS

ARTICLE

Received 19 Feb 2014 | Accepted 4 Jun 2014 | Published 2 Jul 2014

DOI: 10.1038/ncomms5308

Searching for exotic particles in high-energy physics with deep learning

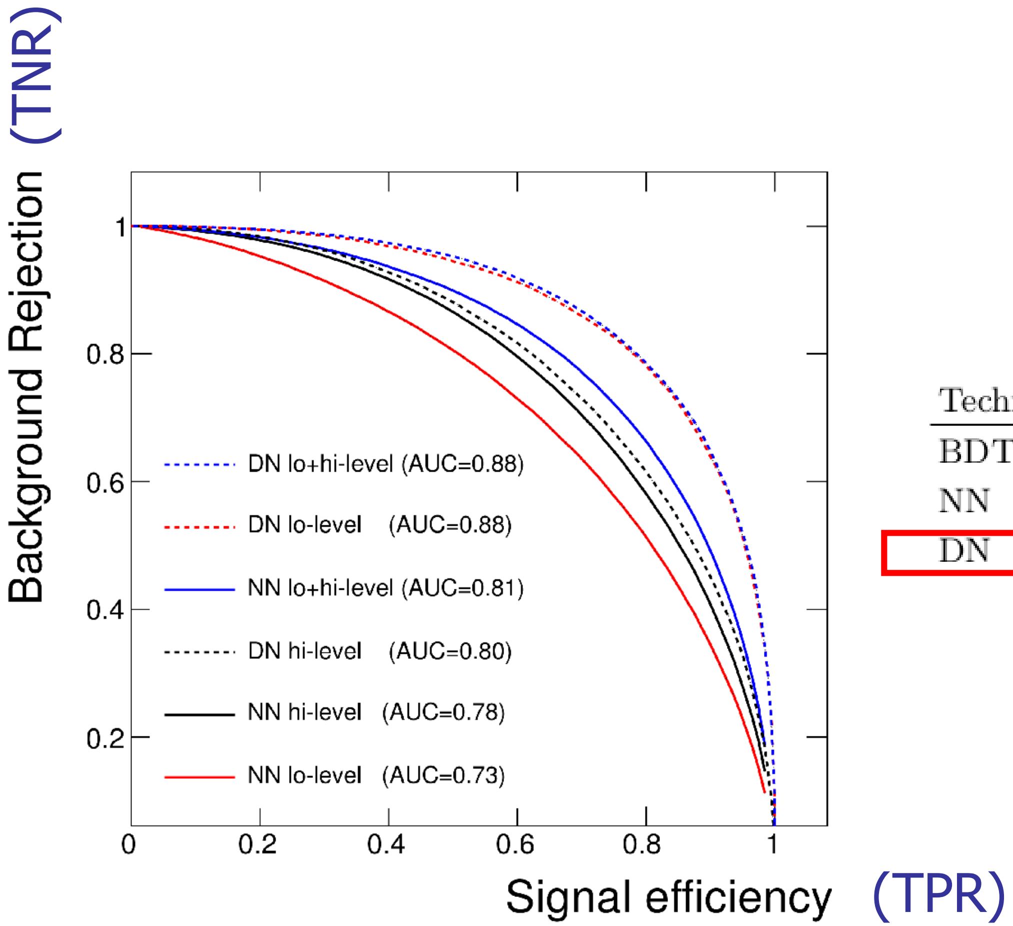
P. Baldi[1], P. Sadowski[1] & D. Whiteson[2]

Collisions at high-energy particle colliders are a traditionally fruitful source of exotic particle discoveries. Finding these rare particles requires solving difficult signal-versus-background classification problems, hence machine-learning approaches are often used. Standard approaches have relied on 'shallow' machine-learning models that have a limited capacity to learn complex nonlinear functions of the inputs, and rely on a painstaking search through manually constructed nonlinear features. Progress on this problem has slowed, as a variety of techniques have shown equivalent performance. Recent advances in the field of deep learning make it possible to learn more complex functions and better discriminate between signal and background classes. Here, using benchmark data sets, we show that deep-learning methods need no manually constructed inputs and yet improve the classification metric by as much as 8% over the best current approaches. This demonstrates that deep-learning approaches can improve the power of collider searches for exotic particles.

Daniel Whiteson

Peter Sadowski

Higgs Boson Detection



	AUC		
Technique	Low-level	High-level	Complete
BDT	0.73	0.78	0.81
NN	0.733 (0.007)	0.777 (0.001)	0.816 (0.004)
DN	0.880 (0.001)	0.800 (< 0.001)	0.885 (0.002)

Deep network improves AUC by 8%

BDT= Boosted Decision Trees in TMVA package

Nature Communications, July 2014