Module 3, Lecture 1: Classical Neural Networks

M. Vidyasagar

Distinguished Professor, IIT Hyderabad
Email: m.vidyasagar@iith.ac.in
Website: www.iith.ac.in/∼m_vidyasagar/

Outline

Outline

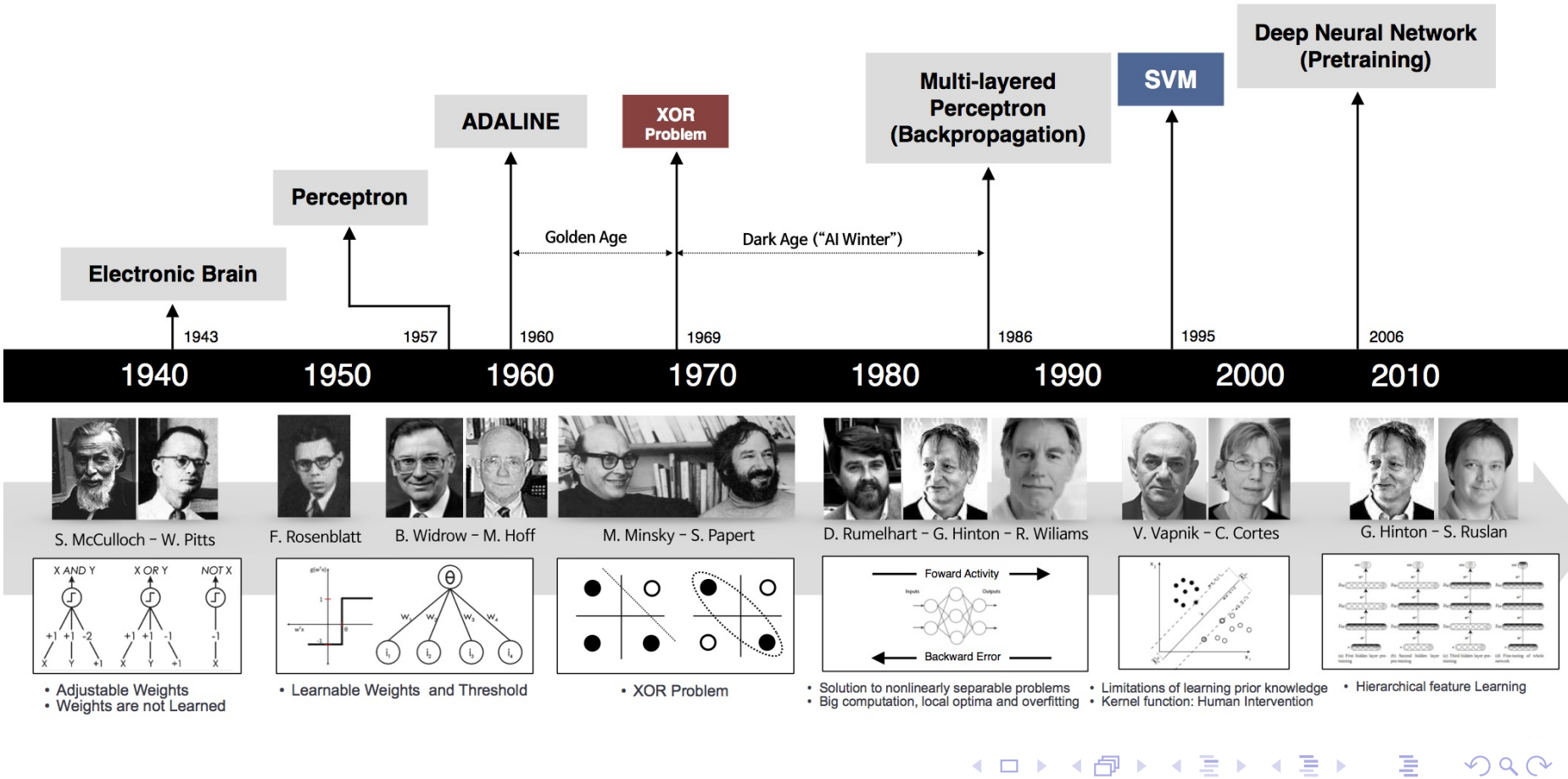
IIT Hyderabad

Outline

IIT Hyderabad

Neural Network Development: Some Milestones

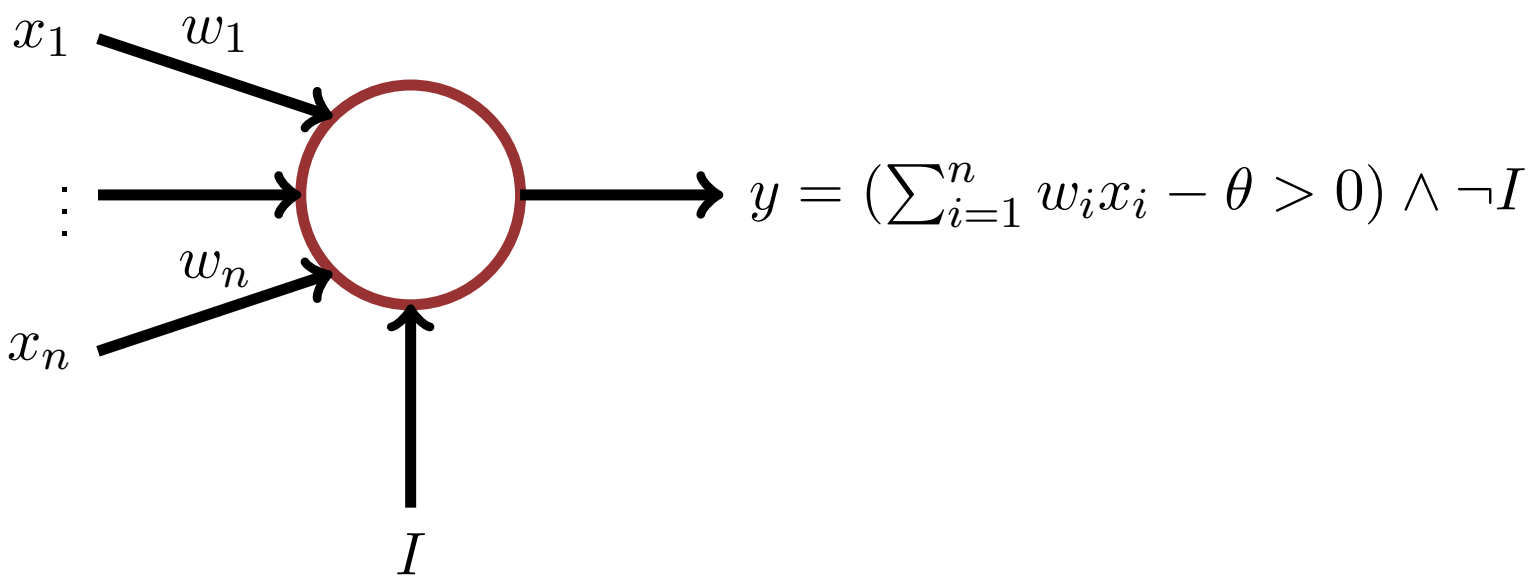
Some milestones in the development of neural networks.



McCulloch-Pitts Model of Brain Neurons (1943)

- The device has a binary output.
- The "excitatory" inputs can come by multiple channels, each with a weight of one. So effectively the weight is a positive integer.
- The "inhibitory" input has a weight of one.
- There is also a threshold.
- If weighted inputs exceed threshold and there is no inhibitory input, the neuron "fires."

Depiction of the McCulloch-Pitts Neuron Model

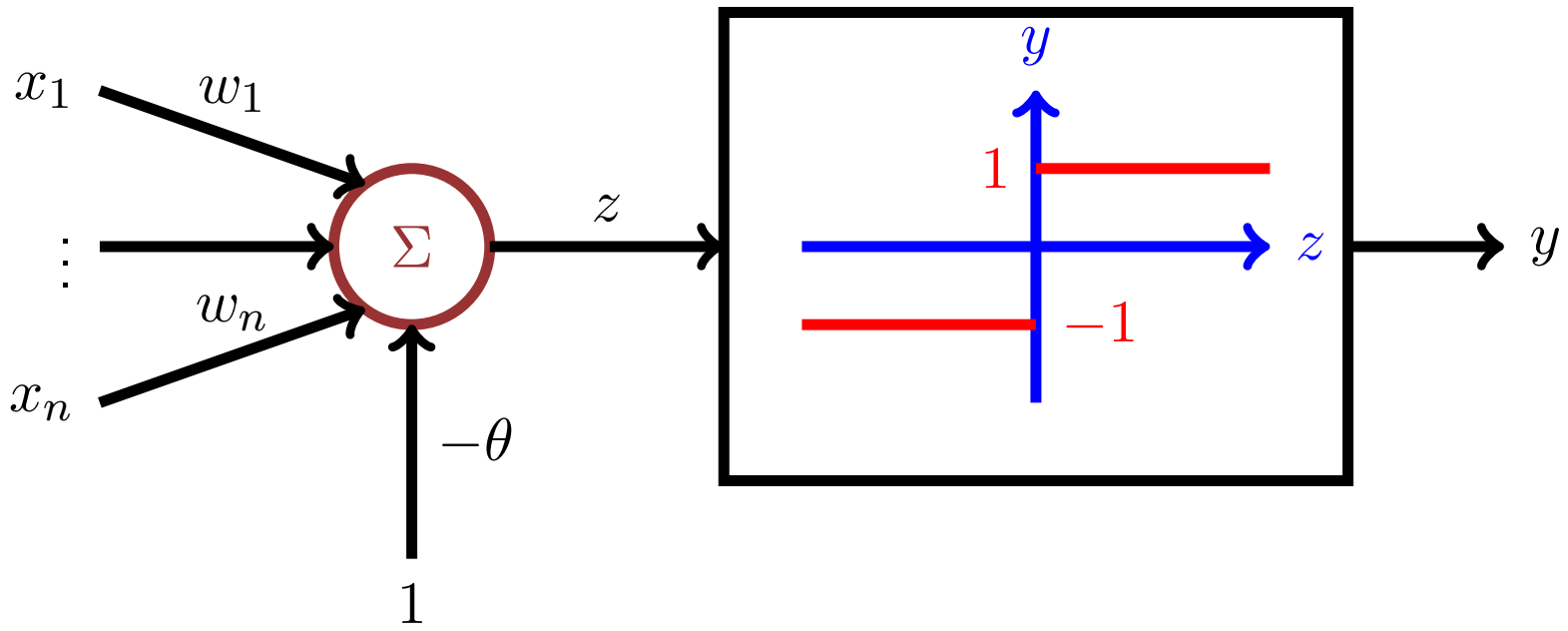


The weights are positive integers.

The neuron can also have "quiescence" – it can't fire again until after a delay.

The Perceptron of Frank Rosenblatt (1950s)

Same as the McCulloch-Pitts Neuron without (i) the inhibitory neuron, and (ii) quiescence.



Other types of "neuron" characterstics are discussed later.

Outline

IIT Hyderabad

Training a Perceptron

Suppose we are given labelled inputs $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$, where each $\mathbf{x}_i \in \mathbb{R}^n$, and $y_i \in \{-1, 1\}$.

The **perceptron training problem** is to find, if it exists, a pair $(\mathbf{w}, \theta)$ such that

$$\mathbf{w}^\tau \mathbf{x}_i - \theta = \sum_{j=1}^{n} w_j (\mathbf{x}_i)_j - \theta \begin{cases} \geq 0 & \text{if } y_i = +1, \\ < 0 & \text{if } y_i = -1. \end{cases}, i = 1, \ldots, m.$$

This statement is equivalent to

$$y_i [\mathbf{w}^\top \quad -\theta] \begin{bmatrix} \mathbf{x}_i \\ 1 \end{bmatrix} \geq 0, i = 1, \ldots, m.$$

Training a Perceptron (Cont'd)

This is just checking whether a set of *linear inequalities* in $\mathbf{w}, \theta$ has a solution, i.e., whether the data is linearly separable.

But in 1960, there were no efficient methods for doing this. (Karmarkar's algorithm wasn't invented until 1984.)

So Rosenblatt invented the "perceptron training algorithm."

The Perceptron Training Algorithm

Let $\mathbf{w}^k, \theta^k$ be the weight and threshold at iteration k. Initialize with whatever $\mathbf{w}, \theta$ you wish.

- Start with $i = 1$, and check whether

$$y_i(\mathbf{w}^k \mathbf{x}_i - \theta^k) \geq 0.$$

- If yes, do nothing. If not, replace

$$\mathbf{w}^k \leftarrow \mathbf{w}^k + y_i \delta \mathbf{x}_i, \theta^k \leftarrow \theta^k - y_i \delta,$$

where δ is a fixed "step size."

Behaviour of the Perceptron Training Algorithm

If the data is linearly separable, then within "a finite number of steps," the algorithm produces a suitable $\mathbf{w}, \theta$ pair.

If not it runs forever.

This algorithm per se is useless, but it inspired "kernel-based learning."

Outline

IIT Hyderabad

Formulation as a Learning Problem

Many of the issues in neural network learning can be illustrated using the humble perceptron!

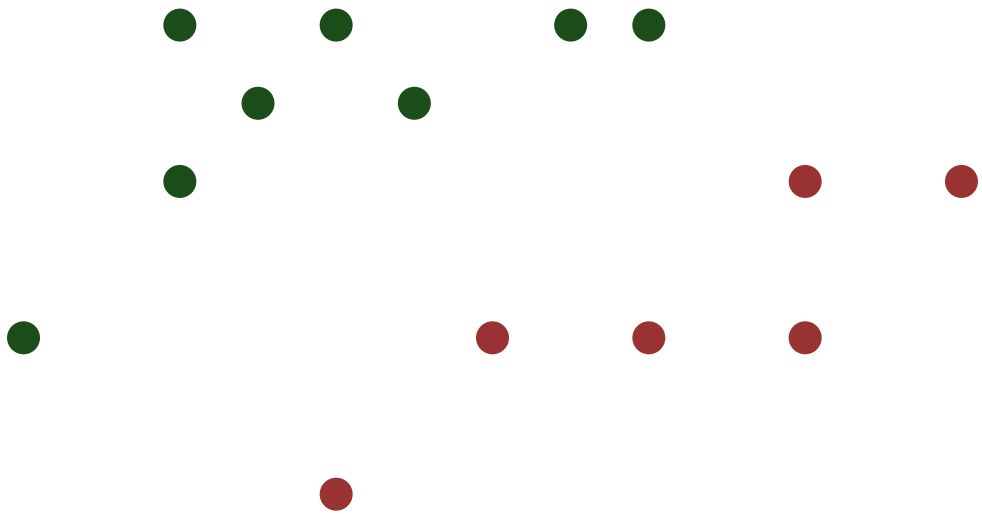
Problem: Suppose we have labelled data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$ *generated by a true put unknown perceptron*.

Can we find the "true" perceptron (or equivalently, the "true" $\mathbf{w}, \theta$) that generate the data?

A little more precisely: Suppose that there is a "true but unknown" perceptron with parameters $(\mathbf{w}, \theta)$. Each time we present it with a random input $\mathbf{x}$, it tells us the sign of $\mathbf{w}^\top \mathbf{x} - \theta$.

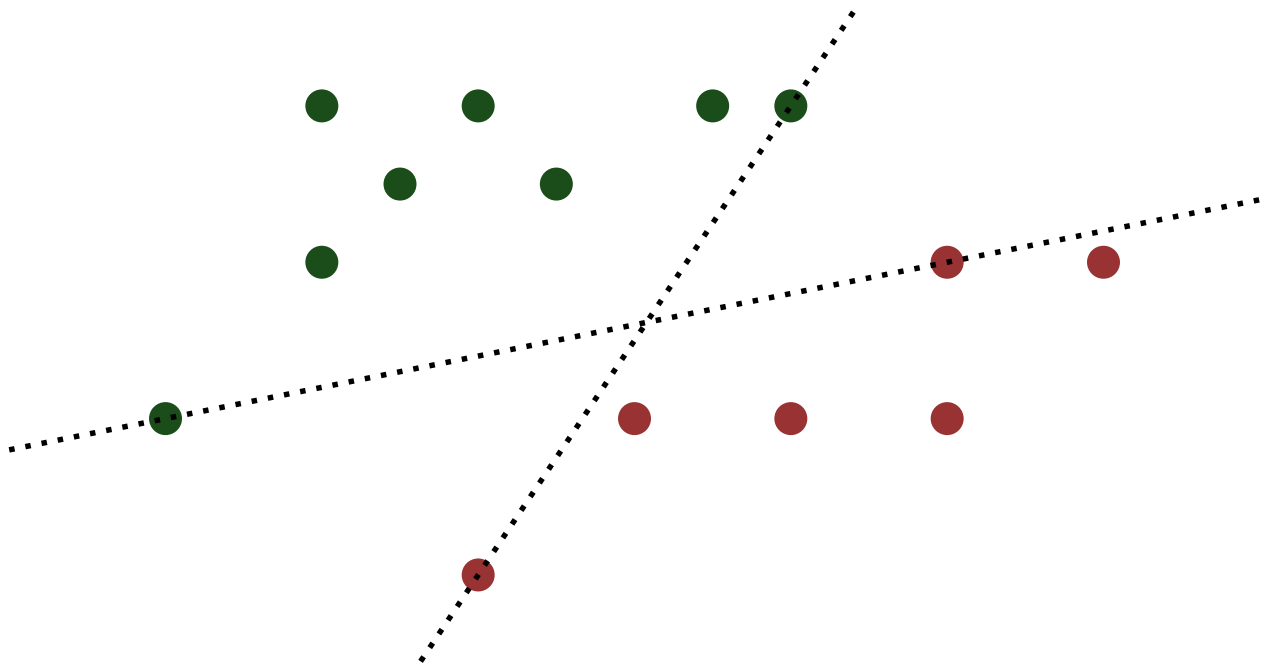
From this information, after "enough" experiments, can we determine $\mathbf{w}, \theta$?

Depiction of the Problem: Find $\mathbf{w}$ and θ



Linear separability of the data is guaranteed (why?) But there are *infinitely many* separating lines! Which one is "true"?

Nonlearnability After Finitely Many Samples



Any straight line passing between the two dotted lines is "consistent" with the data – the true perceptron could be *any one* of them.

Reformulation as a Generalization Problem

Suppose that there is a "true but unknown" perceptron with parameters $(\mathbf{w}, \theta)$. Each time we present it with a random input $\mathbf{x}$, it tells us the sign of $\mathbf{w}^\top \mathbf{x} - \theta$.

After training with m labelled (random) inputs, we generate a random "test" input $\mathbf{x}_{m+1}$ *using the same probability distribution* that generated the training samples. What is the probability that the test input is correctly classified?

Support Vector Machines (SVMs)

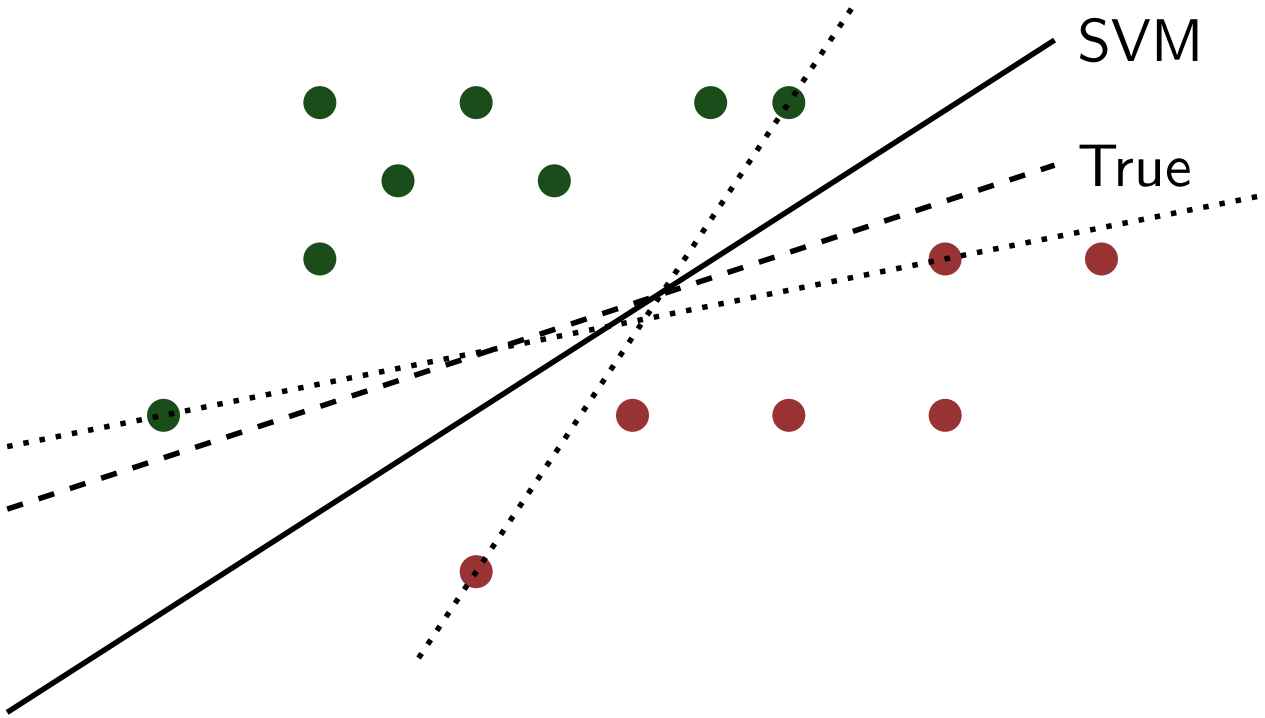
A Support Vector Machine (SVM) chooses the straight line such that the closest point is as far away as possible (see figure on next slide).

Still, the "true" perceptron could be any one in the region shown.

The probability of misclassification depends on the "measure" of the region between the two lines (the likelihood that the test input belongs to this region).

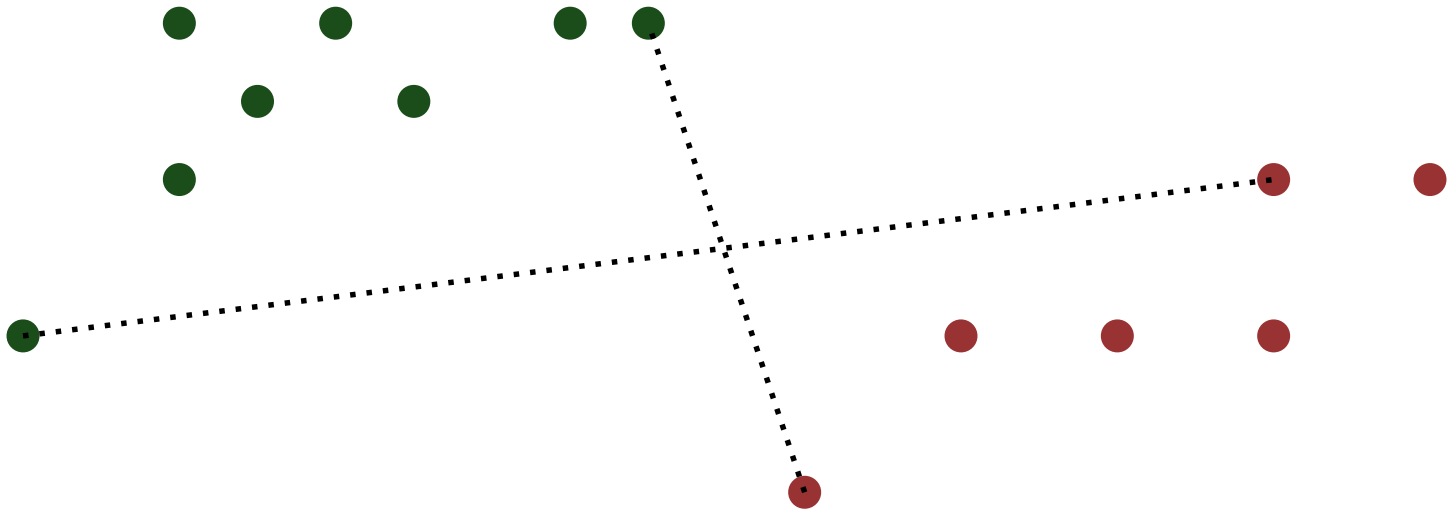
It is *impossible* for the generalization error to equal zero after finitely many samples, for *every* true but unknown perceptron (next slide).

SVM and Generalization Error



The area between SVM and True is the probability of misclassifying a test input. It cannot equal zero after finitely many samples, for *every* true but unknown perceptron.

Possibility of Getting Bad Training Samples



Because training samples are randomly drawn, the above situation could result – not likely but not impossible. The worst-case generalization error would be *huge!*

Outline

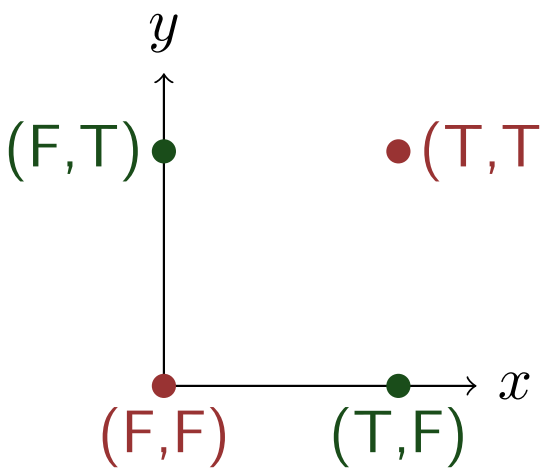
IIT Hyderabad

Outline

IIT Hyderabad

End of the Perceptron: The XOR Problem

Perceptrons can be used only with *lineary separable* data. The XOR problem is not linearly separable. This observation by Minsky and Papert (1969) effectively killed off the perceptron. The subject then went into a hiatus until 1986.



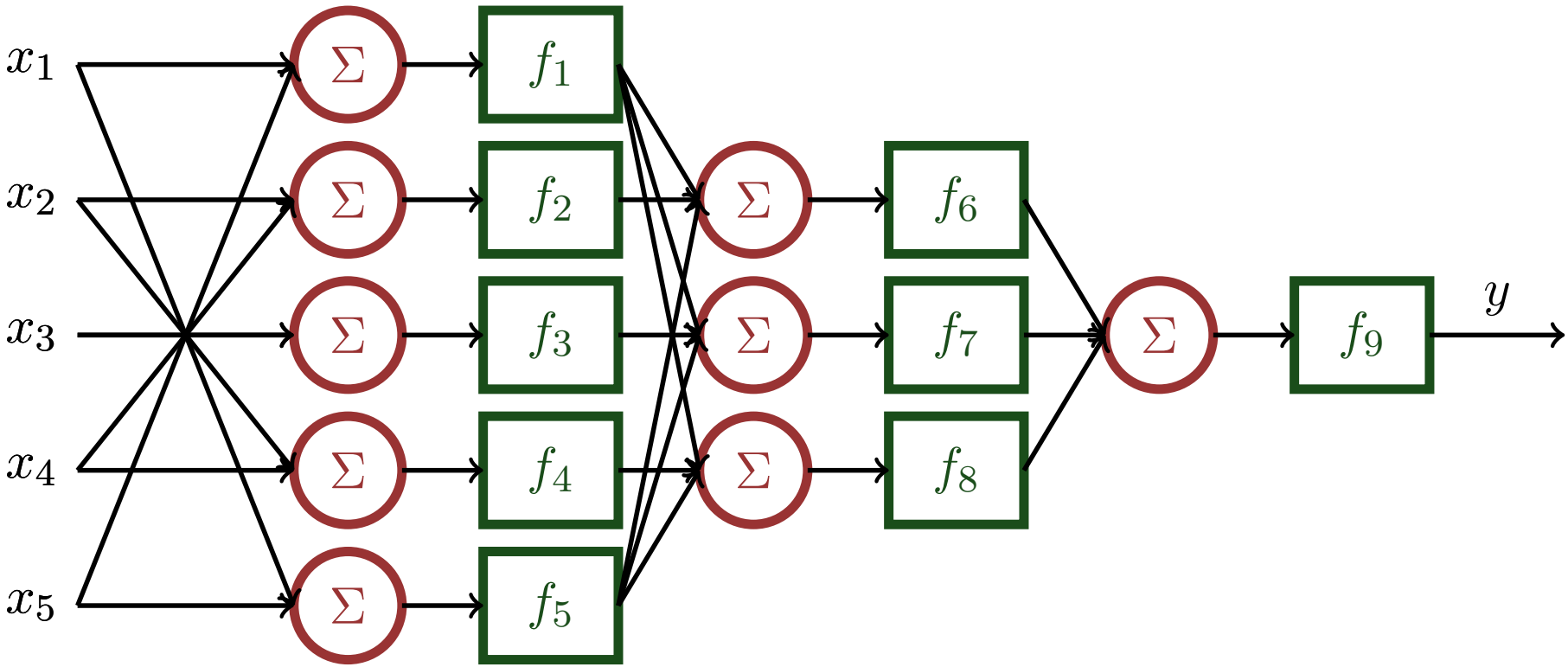
Multi-Layer Perceptron Networks (MLPNs)

In 1986, Rumelhart, Hinton and Williams proposed to use *multi-layer* neural networks, where the "neuron" could be a perceptron, or anything else. (Figure on next slide.)

In early days, the neuron was a perceptron or a sigmoid. In recent times other types of neurons are used. (Figures on next slides).

It is also possible to use multiple types of neurons in the same network.

MLPN Architecture



Each f_i is a neuron of some kind; not all need to be the same.

Various Neuron Characteristics – 1: Sigmoidal Functions

We have already seen the perceptron characteristic, namely

$$y = \text{sign}(z) = \begin{cases} +1 & \text{if } z \geq 0, \\ -1 & \text{if } z < 0. \end{cases}$$

Other possibilities: The "standard sigmoid"

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

or the hyperbolic tangent function

$$y = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

Note that

$$2\sigma(z) - 1 = \tanh z/2.$$

Sign and Tanh Nonlinearities

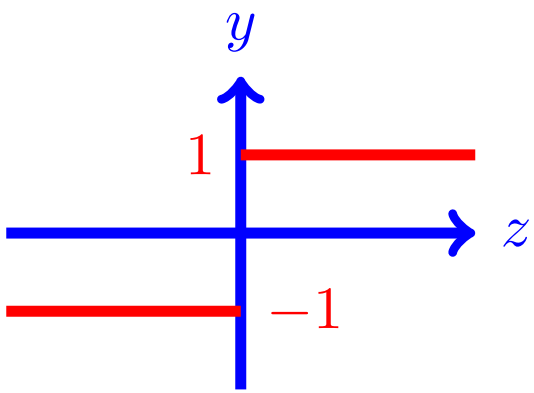


Figure: Sign Function

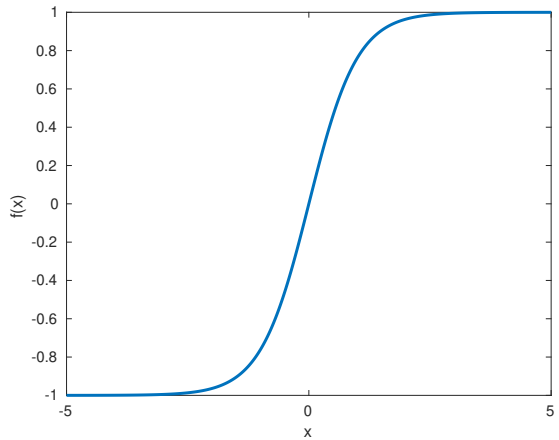
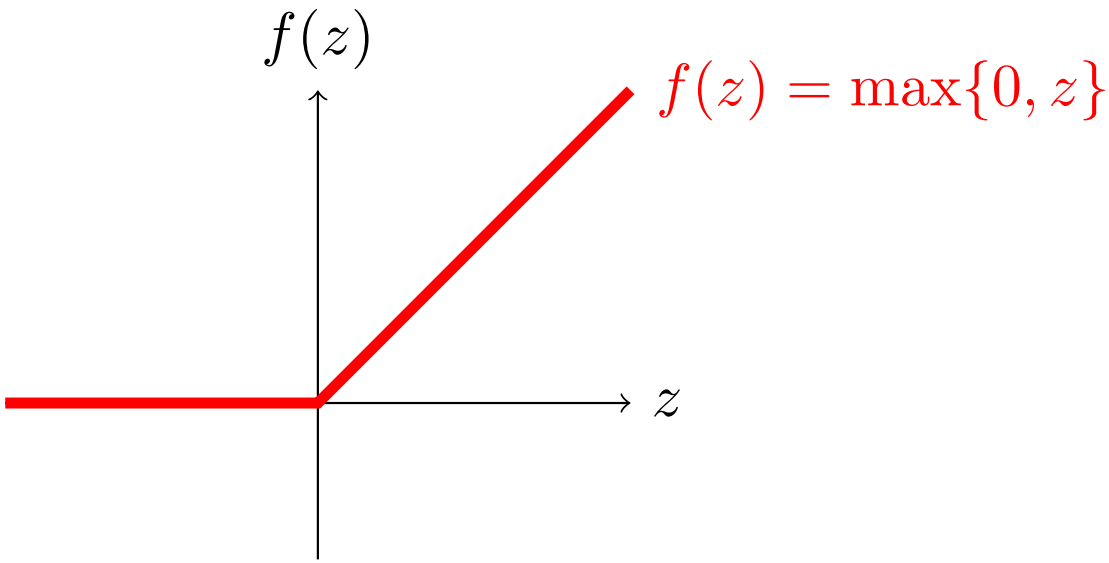


Figure: Tanh Nonlinearity

Various Neuron Characteristics – 2: ReLU

The Rectified Linear Unit (ReLU) is defined by

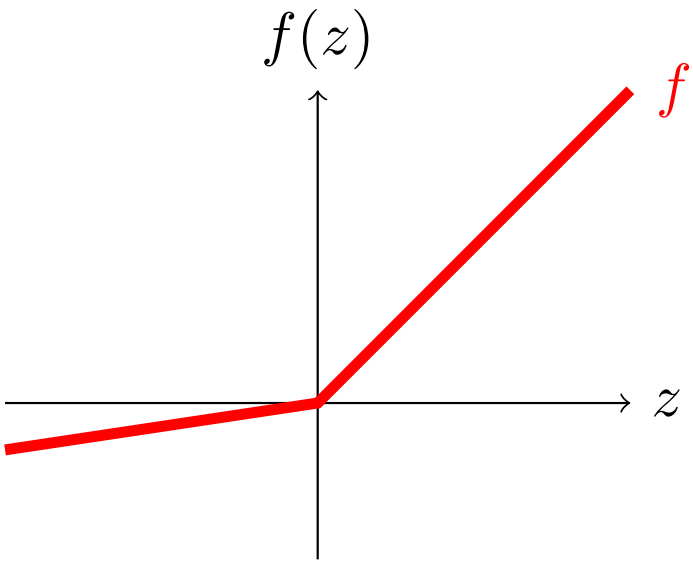
$$y = \begin{cases} z & \text{if } z \geq 0, \\ 0 & \text{if } z < 0. \end{cases}$$



Various Neuron Characteristics – 2: "Leaky" ReLU

The "Leaky" Rectified Linear Unit (ReLU) is defined by

$$y = \begin{cases} z & \text{if } z \geq 0, \\ 0 & \text{if } \alpha z < 0. \end{cases}$$

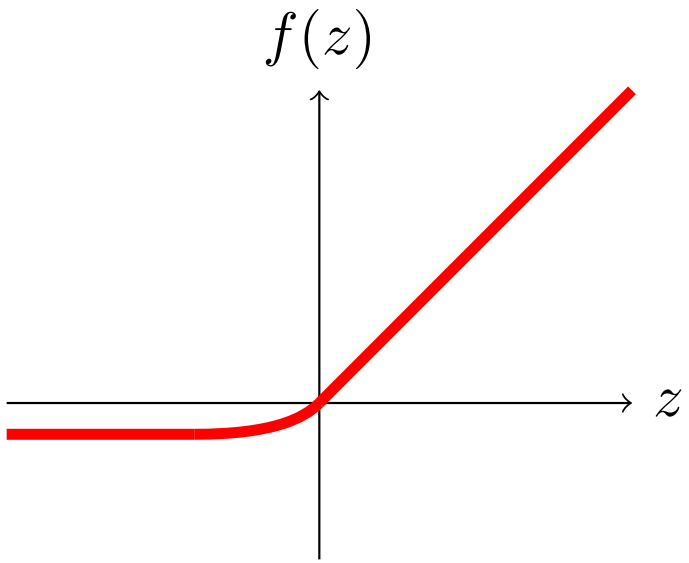


$f(z) = \max\{\alpha z, z\}, \alpha \ll 1$

Various Neuron Characteristics – 3: ELU

The Exponential Linear Unit (ELU) is defined by

$$y = \begin{cases} z & \text{if } z \geq 0, \\ \alpha(e^{-z} - 1) & \text{if } z < 0. \end{cases}$$



Outline

IIT Hyderabad

Types of Neural Networks

Usually neural networks are categorized into two types:

- If the output is *bipolar* (± 1), then the NN is of *classification type.*
- If the output is a *real number*, then the NN is of *regression type*.

Typical Neural Network Learning Problem

- We are also given a family of models – earlier they were perceptrons, but the models could be more general neural networks. This is called the **architecture** of the neural network model.
- Each choice of network parameters leads to a different *network*.
- We are given labelled data $(\mathbf{x}_i, y_i), i = 1, \ldots, m$, where $\mathbf{x}_i \in \mathbb{R}^n, y_i \in \{-1, 1\}$ for classification networks, and $y_i \in \mathbb{R}$ for regression networks.

Typical Neural Network Learning Problem (Cont'd)

Each neural network architecture defines a function $f(\mathbf{w}, \mathbf{x})$ that maps $\mathbb{R}^n$ into either $\{-1, 1\}$ (for classification) or $\mathbb{R}$ (for regression).

Given the labelled data $(\mathbf{x}_i, y_i), i = 1, \ldots, m$, there are two distinct steps:

1. Choose a network NN in $\mathcal{N}$ that best matches the data.
   - With perceptrons this is easy (LP problem).
   - In general, even this step can be computationally infeasible.
2. Estimate the generalization error with a randomly selected test input.

Minimizing the Training Error

Let $\mathbf{w}$ denote the set of "weights" in the NN, or the set of adjustable parameters. Let $f(\mathbf{w}, \mathbf{x})$ denote the output produced by the network when the input vector is $\mathbf{x}$ and the weight vector is $\mathbf{w}$. In MLPNs, $f(\mathbf{w}, \mathbf{x})$ will be a very complicated nonlinear function.

In regression networks, both the network output $f(\mathbf{w}, \mathbf{x}_i)$ and the label y_i are real numbers. One chooses $\mathbf{w}$ by minimizing the *average least-squares error:*

$$\min_{\mathbf{w}} J(\mathbf{w}) = \frac{1}{m} \sum_{i=1}^{m} (y_i - f(\mathbf{w}, \mathbf{x}_i))^2.$$

Minimizing the Training Error (Cont'd)

In classification networks, both the network output $f(\mathbf{w}, \mathbf{x}_i)$ and the labe y_i are bipolar (or binary).

One chooses $\mathbf{w}$ by minimizing the *fraction of misclassified samples*.

$$\min_{\mathbf{w}} J(\mathbf{w}) = \frac{1}{m} \sum_{i=1}^{m} \#(y_i \neq f(\mathbf{w}, \mathbf{x}_i)).$$

Some Caveats

- To minimize a least-squares criterion (in regression), we would like the function $J(\mathbf{w})$ to be *continuously differentiable* so that we can use gradient techniques.
- But not all neuron characteristics have this property. So we may need to "smoothen" them.
- Minimizing the fraction of misclassified samples (in classification) is often NP-hard. So we need to resort to approximations.

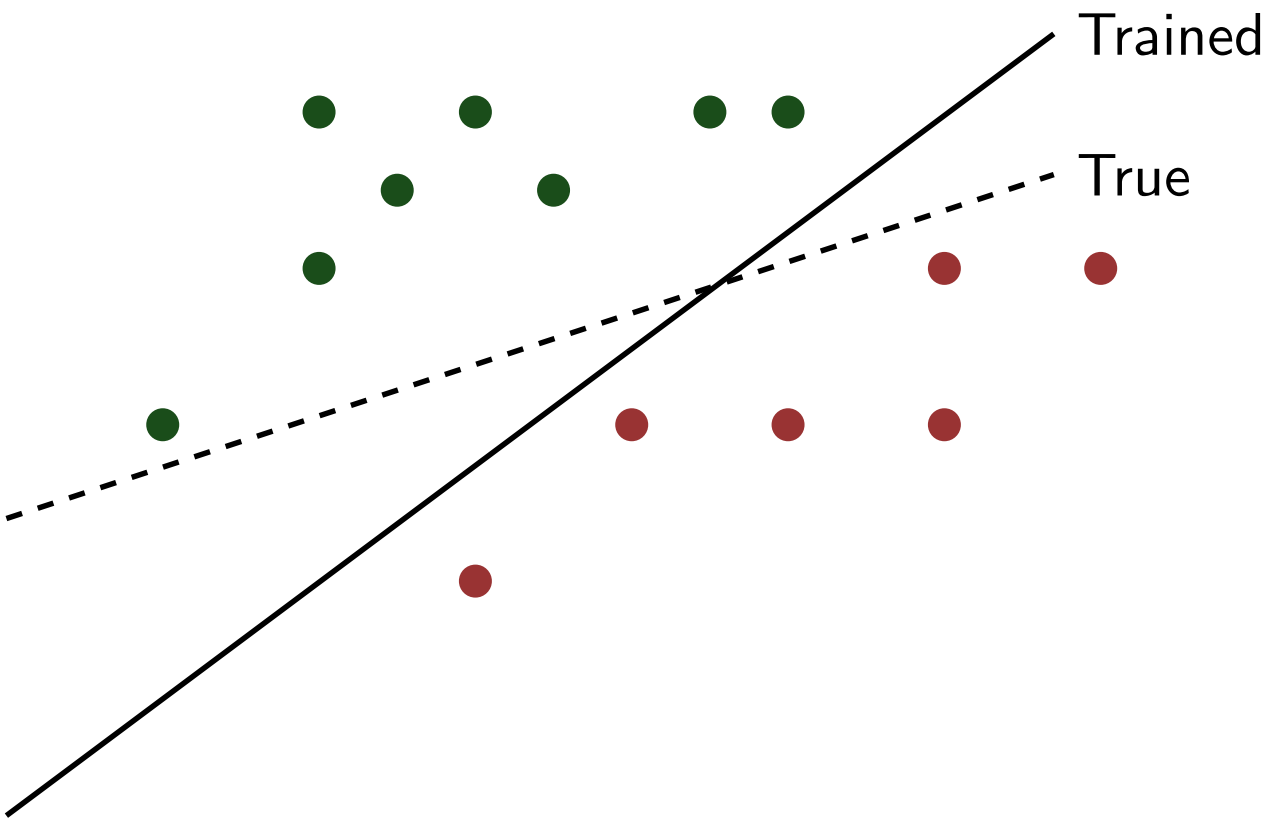
Difference Between Training and Testing Error

A low training error *does not imply* a low generalization error!

Example: Perceptron on next slide.

Later we will discuss methods for predicting the generalization error, which depends on the "richness" of the architecture and the number of training samples.

Difference Between Training and Generalization Error



The trained NN has zero training error but nonzero generalization error.

Outline

IIT Hyderabad

Methods for Performining the Minimization

Suppose we wish to minimize a function $J(\mathbf{w})$ of the weight vector.

Commonly used methods for minimizing the training error:

- Back-propagation (doesn't work for very large NNs)
- Stochastic gradient descent
- Batch stochastic gradient descent

Back-propagation is described here; others in later lectures.

Steepest Descent Method

The **steepest descent** method is the oldest and simplest (but *not* the most efficient) method for minimization.

Suppose $J : \mathbb{R}^l \to \mathbb{R}$ is continuously differentiable. Here J is the cost function to be minimized, and l is the number of weights.

1. Initialize: Set $k = 0$, $\mathbf{w} = \mathbf{w}^0$.
2. Compute $\mathbf{g}^k = \nabla J(\mathbf{w}^k)$.
3. Set $\mathbf{w}^{k+1} = \mathbf{w}^k - t_k \mathbf{g}^k$, where t_k is the "step size."
4. Increment k and repeat until $\nabla J(\mathbf{w}^k)$ is sufficiently small.

Ways to choose t_k: (a) Choose a fixed value $t_k = \delta$. (b) Choose fixed $t_k \downarrow 0$ as $k \to \infty$. (c) Choose

$$t_k = \underset{t}{\operatorname{argmin}}\, J(\mathbf{w}^k - t\mathbf{g}^k).$$

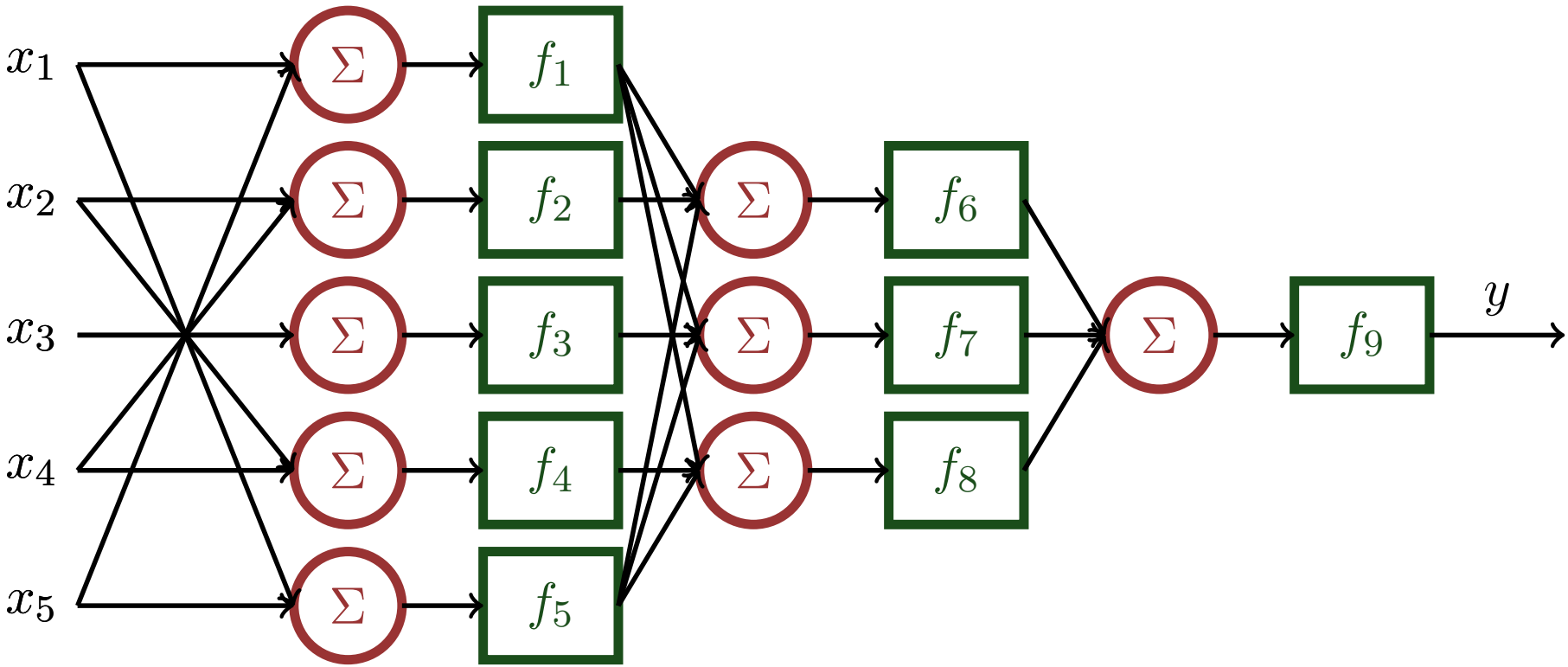
Back-Propagation

To implement steepest descent, we need to comput $\nabla J(\mathbf{w})$, for weights at *all levels* of the NN.

How to do this? Use the *chain rule!*

Recall the MLPN architecture (next slide).

MLPN Architecture (Reprise)



Some Useful Observations

- The output of each summing element is a *linear combination* of its inputs.
- If

$$z = \sum_{i=1}^{n} w_i x_i,$$

then

$$\frac{\partial z}{\partial w_i} = x_i.$$

- If $x = f(z)$, then $\partial f(z)/\partial x$ is readily calculated.
- By applying the chain rule as many times as needed, we can compute $\partial J/\partial w_i$ where w_i is a weight at *any* level!
- Conceptually simple, requires messy notation.

Back-Propagation with Momentum Term

This is known in optimization theory as the **conjugate gradient method.**

At iteration k, instead of choosing the search direction to be $-\nabla J(\mathbf{w}^k)$, choose it to be a linear combination of $-\nabla J(\mathbf{w}^k)$ and $-\nabla J(\mathbf{w}^{k-1})$.

Available software packages offer all these as options.

Outline

Database Resources

There are several publicly available databases, of which a couple are listed below:

- https://skymind.ai/wiki/open-datasets
- https://www.analyticsvidhya.com/blog/2018/03/comprehensive-collection-deep-learning-datasets/

IIT Hyderabad

Software Resources

`Matlab` has a `Neural Networks Toolbox` that is good enough for illustrative purposes.

Many major players have made their software publicly available. These include:

- Tensor-Flow (Google)
- Caffe2 (Facebook)
- Pytorch (Facebook)
- Keras (Google)