Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

An Introduction to Neural Networks

Neural Networks and Deep Learning, Springer, 2018
Chapter 1, Sections 1.1–1.2

Neural Networks

- Neural networks have seen an explosion in popularity in recent years.
  - Victories in eye-catching competitions like the *ImageNet* contest have brought them fame.
  - The potential of neural networks is now being realized because of fundamental shifts in hardware paradigms and data availability.
- The new term *deep learning* is a re-birth of the field of neural networks (although it also emphasizes a specific aspect of neural networks).

Overview of the Presented Material

- The videos are based on the book: C. Aggarwal. Neural Networks and Deep learning, *Springer*, 2018.
  - Videos not meant to be exhaustive with respect to book.
  - Helpful in providing a firm grounding of important aspects.
  - Videos can provide the initial background to study more details from the book.
  - Slides available for download at http://www.charuaggarwal.net (with latex source).

Overview of Book

- The book covers both the old and the new in neural networks.
  - The core learning methods like backpropagation, traditional architectures, and specialized architectures for sequence/image applications are covered.
  - The latest methods like variational autoencoders, Generative Adversarial Networks, Neural Turing Machines, attention mechanisms, and reinforcement learning are covered.
    * Reinforcement learning is presented from the view of deep learning applications.
  - The "forgotten architectures" like Radial Basis Function networks and Kohonen self organizing maps are also covered.

Neural Networks: Two Views

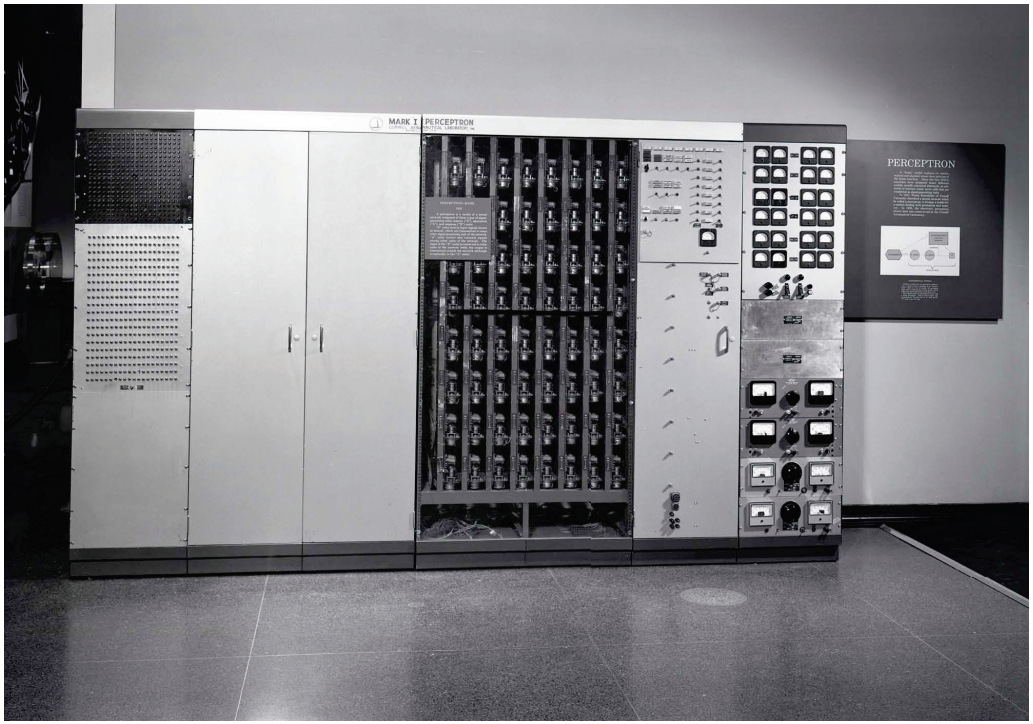
- A way to simulate biological learning by simulating the nervous system.

- A way to increase the power of known models in machine learning by stacking them in careful ways as *computational graphs*.
  - The number of nodes in the computational graph controls learning capacity with increasing data.
  - The specific architecture of the computational graph incorporates domain-specific insights (e.g., images, speech).
  - The success of deep computational graphs has led to the coining of the term "*deep learning*."

Historical Origins

- The first model of a computational unit was the *perceptron* (1958).
  - Was roughly inspired by the biological model of a neuron.
  - Was implemented using a large piece of hardware.
  - Generated great excitement but failed to live up to inflated expectations.
- Was not any more powerful than a simple linear model that can be implemented in a few lines of code today.

The Perceptron [Image Courtesy: Smithsonian Institute]



The First Neural Winter: Minsky and Papert's Book

- Minsky and Papert's Book "*Perceptrons*" (1969) showed that the perceptron only had limited expressive power.
  - Essential to put together multiple computational units.
- The book also provided a pessimistic outlook on training multilayer neural networks.
  - Minsky and Papert's book led to the first winter of neural networks.
  - Minsky is often blamed for setting back the field (fairly or unfairly).
- Were Minsky/Papert justified in their pessimism?

Did We Really Not Know How to Train Multiple Units?

- It depends on who you ask.
  - AI researchers didn't know (and didn't believe it possible).
  - Training computational graphs with dynamic programming had already been done in control theory (1960s).
- Paul Werbos proposed backpropagation in his 1974 thesis (and was promptly ignored–formal publication was delayed).
  - Werbos (2006): "*In the early 1970s, I did in fact visit Minsky at MIT. I proposed that we do a joint paper showing that MLPs can in fact overcome the earlier problems if (1) the neuron model is slightly modified to be differentiable; and (2) the training is done in a way that uses the reverse method, which we now call backpropagation in the ANN field. But Minsky was not interested. In fact, no one at MIT or Harvard or any place I could find was interested at the time.*"

General View of Artificial Intelligence (Seventies/Eighties)

- It was the era or work on logic and reasoning (discrete mathematics).
  - Viewed as the panacea of AI.
  - This view had influential proponents like Patrick Henry Winston.
- Work on continuous optimization had few believers.
  - Researchers like Hinton were certainly not from the mainstream.
  - This view has been completely reversed today.
  - The early favorites have little to show in spite of the effort.

Backpropagation: The Second Coming

- Rumelhart, Hinton, and Williams wrote two papers on back-propagation in 1986 (independent from prior work).
  - Paul Werbos's work had been forgotten and buried at the time.
- Rumelhart *et al*'s work is presented beautifully.
- It was able to at least partially resurrect the field.

The Nineties

- Acceptance of backpropagation encouraged more research in multilayer networks.

- By the year 2000, most of the modern architectures had already been set up in some form.
  - They just didn't work very well!
  - The winter continued after a brief period of excitement.

- It was the era of the support vector machine.
  - The new view: SVM was the panacea (at least for supervised learning).

What Changed?

- Modern neural architectures are similar to those available in the year 2000 (with some optimization tweaks).

- **Main difference:** Lots of data and computational power.

- Possible to train large and deep neural networks with millions of neurons.

- Significant Events: Crushing victories of deep learning methods in *ImageNet* competitions after 2010.

- *Anticipatory Excitement:* In a few years, we will have the power to train neural networks with as many computational units as the human brain.

  – Your guess is as good as mine about what happens then.

A Cautionary Note

- Deep learning has now assumed the mantle of the AI panacea.
  - We have heard this story before.
  - Why should it be different this time?
  - Excellent performance on richly structured data (images, speech), but what about others?
- There are indeed settings where you are better off using a conventional machine learning technique like a random forest.

How it All Started: The Biological Inspiration

- Neural networks were originally designed to simulate the learning process in biological organisms.

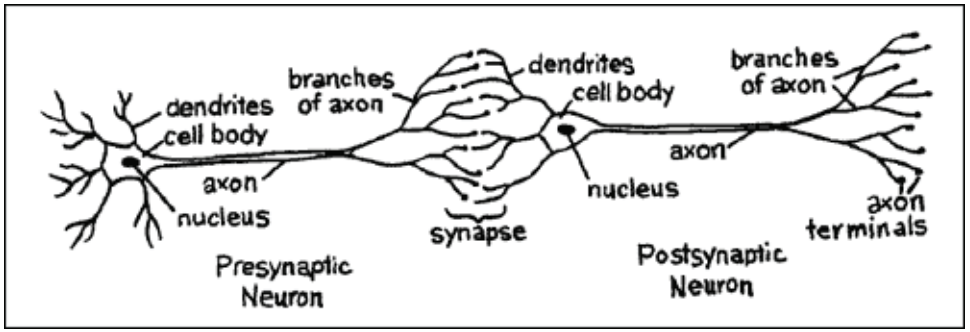
- The human nervous system contains cells called *neurons*.

- The neurons are connected to one another with the use of *synapses*.

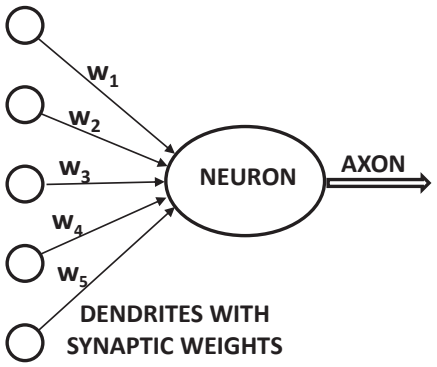
  – The strengths of synaptic connections often change in response to external stimuli.

  – This change causes learning in living organisms.

Neural Networks: The Biological Inspiration



(a) Biological neural network



(b) Artificial neural network

- Neural networks contain computation units $\Rightarrow$ Neurons.
- The computational units are connected to one another through weights $\Rightarrow$ Strengths of synaptic connections in biological organisms.
- Each input to a neuron is scaled with a weight, which affects the function computed at that unit.

Learning in Biological vs Artificial Networks

- In living organisms, synaptic weights change in response to external stimuli.
  - An unpleasant experience will change the synaptic weights of an organism, which will train the organism to behave differently.
- In artificial neural networks, the weights are learned with the use of training data, which are input-output pairs (e.g., images and their labels).
  - An error made in predicting the label of an image is the unpleasant "stimulus" that changes the weights of the neural network.
  - When trained over many images, the network learns to classify images correctly.

Comments on the Biological Paradigm

- The biological paradigm is often criticized as a very inexact caricature.
  - The functions computed in a neural network are very different from those in the brain. $\Rightarrow$ No one exactly knows how the brain works.
- Nevertheless, there are several examples, where the principles of neuroscience have been successfully applied in designing neural networks.
  - Convolutional neural networks are based on architectural principles drawn from the cat's visual cortex.
- There has been a renewed focus in recent years in leveraging the principles of neuroscience in neural network design $\Rightarrow$ The "caricature" view is not fully justified.

An Alternative View: The Computational Graph Extension of Traditional Machine Learning

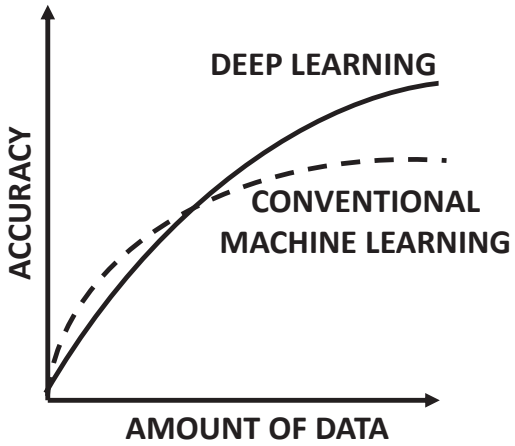
- The elementary units compute similar functions to traditional machine learning models like linear or logistic regression.

- *Much of what you know about optimization-based machine learning can be recast as shallow neural models*.
  - When large amounts of data are available, these models are unable to learn all the structure.

- Neural networks provide a way to increase the capacity of these models by connecting multiple units as a computational graph (with an increased number of parameters).

- At the same time, connecting the units in a particular way can incorporate domain-specific insights.

Machine Learning versus Deep Learning



- For smaller data sets, traditional machine learning methods often provide slightly better performance.

- Traditional models often provide more choices, interpretable insights, and ways to handcraft features.

- For larger data sets, deep learning methods tend to dominate.

Reasons for Recent Popularity

- The recent success of neural networks has been caused by an increase in data and computational power.
  - Increased computational power has reduced the cycle times for experimentation.
  - If it requires a month to train a network, one cannot try more than 12 variations in an year on a single platform.
  - Reduced cycle times have also led to a larger number of successful tweaks of neural networks in recent years.
  - Most of the models have not changed dramatically from an era where neural networks were seen as impractical.
- We are now operating in a data and computational regime where deep learning has become attractive compared to traditional machine learning.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

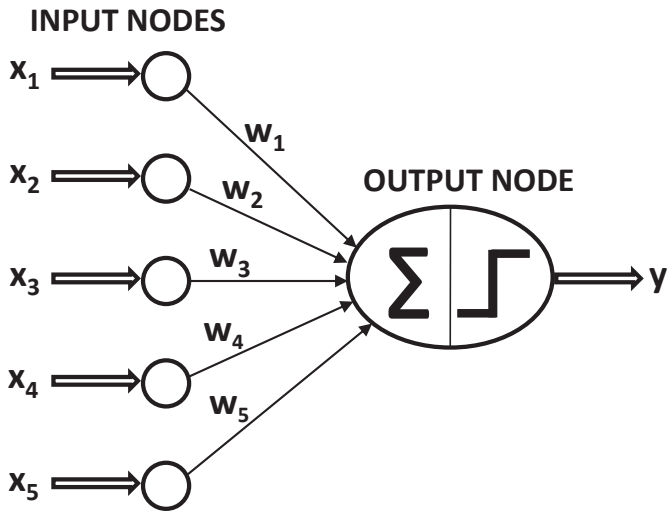
Single Layer Networks: The Perceptron

Neural Networks and Deep Learning, Springer, 2018
Chapter 1, Section 1.3

Binary Classification and Linear Regression Problems

- In the binary classification problem, each training pair $(\overline{X}, y)$ contains feature variables $\overline{X} = (x_1, \ldots x_d)$, and label y drawn from $\{-1, +1\}$.
  - Example: Feature variables might be frequencies of words in an email, and the class variable might be an indicator of spam.
  - Given labeled emails, recognize incoming spam.
- In linear regression, the *dependent* variable y is real-valued.
  - Feature variables are frequencies of words in a Web page, and the dependent variable is a prediction of the number of accesses in a fixed period.
- Perceptron is designed for the binary setting.

The Perceptron: Earliest Historical Architecture



- The d nodes in the input layer only transmit the d features $\overline{X} = [x_1 \ldots x_d]$ without performing any computation.

- Output node multiplies input with weights $\overline{W} = [w_1 \ldots w_d]$ on incoming edges, aggregates them, and applies *sign activation*:

$$\hat{y} = \text{sign}\{\overline{W} \cdot \overline{X}\} = \text{sign}\{\sum_{j=1}^{d} w_j x_j\}$$

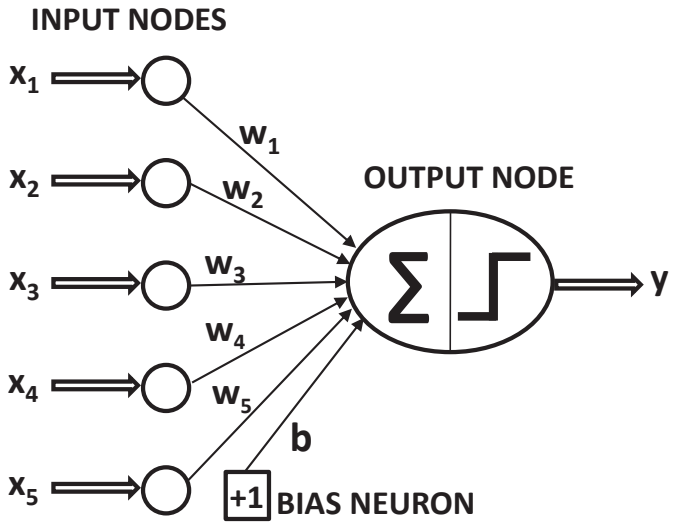
What is the Perceptron Doing?

- Tries to find a *linear separator* $\overline{W} \cdot \overline{X} = 0$ between the two classes.

- Ideally, all positive instances ($y = 1$) should be on the side of the separator satisfying $\overline{W} \cdot \overline{X} > 0$.

- All negative instances ($y = -1$) should be on the side of the separator satisfying $\overline{W} \cdot \overline{X} < 0$.

Bias Neurons



- In many settings (e.g., skewed class distribution) we need an invariant part of the prediction with bias variable b:

$$\widehat{y} = \text{sign}\{\overline{W} \cdot \overline{X} + b\} = \text{sign}\{\sum_{j=1}^{d} w_j x_j + b\} = \text{sign}\{\sum_{j=1}^{d+1} w_j x_j\}$$

- On setting $w_{d+1} = b$ and x_{d+1} as the input from the bias neuron, it makes little difference to learning procedures $\Rightarrow$ Often implicit in architectural diagrams

Training a Perceptron

- Go through the input-output pairs $(\overline{X}, y)$ one by one and make updates, if predicted value $\hat{y}$ is different from observed value $y \Rightarrow$ Biological readjustment of synaptic weights.

$$\overline{W} \Leftarrow \overline{W} + \alpha \underbrace{(y - \hat{y})}_{\text{Error}} \overline{X}$$

$$\overline{W} \Leftarrow \overline{W} + (2\alpha) y \overline{X} \text{ [For misclassified instances } y - \hat{y} = 2y\text{]}$$

- Parameter α is the learning rate $\Rightarrow$ Turns out to be irrelevant in the special case of the perceptron

- One cycle through the entire training data set is referred to as an *epoch* $\Rightarrow$ Multiple epochs required

- How did we derive these updates?

What Objective Function is the Perceptron Optimizing?

- At the time, the perceptron was proposed, the notion of loss function was not popular $\Rightarrow$ Updates were heuristic

- Perceptron optimizes the perceptron criterion for ith training instance:

$$L_i = \max\{-y_i(\overline{W} \cdot \overline{X_i}), 0\}$$

  - Loss function tells us how far we are from a desired solution $\Rightarrow$ Perceptron criterion is 0 when $\overline{W} \cdot \overline{X_i}$ has same sign as y_i.

- Perceptron updates use *stochastic gradient descent* to optimize the loss function and reach the desired outcome.

  - Updates are equivalent to $\overline{W} \Leftarrow \overline{W} - \alpha \left(\frac{\partial L_i}{\partial w_1} \ldots \frac{\partial L_i}{\partial w_d} \right)$

Perceptron vs Linear SVMs

- Perceptron criterion is a shifted version of hinge-loss in SVM:

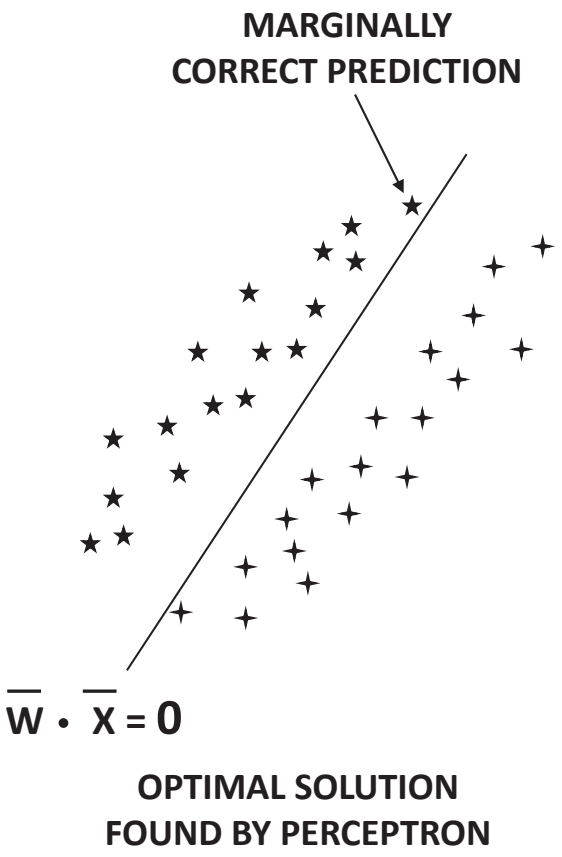
$$L_i^{svm} = \max\{1 - y_i(\overline{W} \cdot \overline{X_i}), 0\}$$

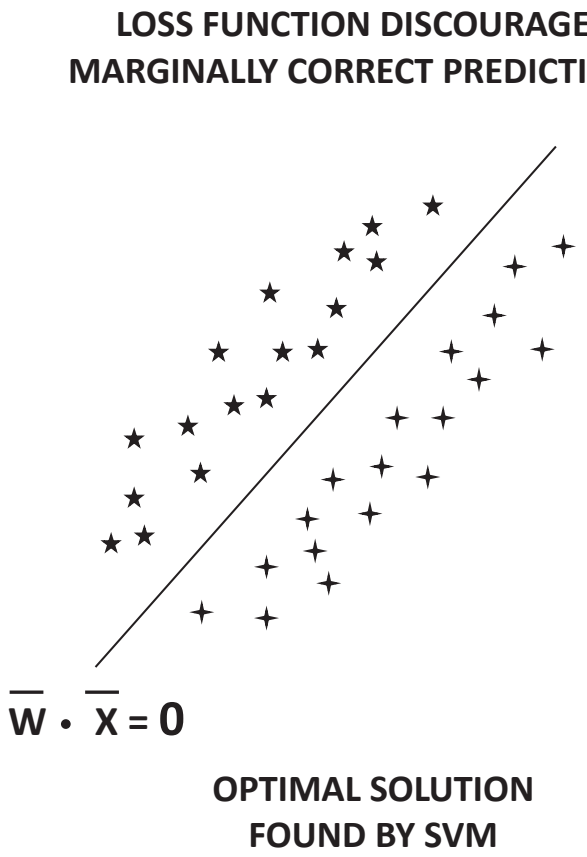
  - The pre-condition for updates in perceptron and SVMs is different:
  - In a perceptron, we update when a misclassification occurs: $-y_i(\overline{W} \cdot \overline{X_i}) > 0$
  - In a linear SVM, we update when a misclassification occurs or a classification is "barely correct": $1 - y_i(\overline{W} \cdot \overline{X_i}) > 0$

- Otherwise, *primal* updates of linear SVM are *identical* to perceptron:

$$\overline{W} \Leftarrow \overline{W} + \alpha y \overline{X}$$

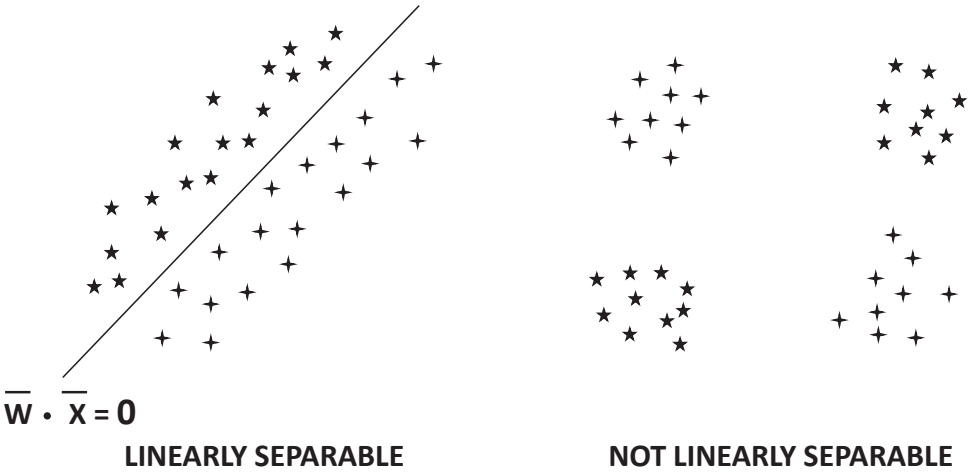
Perceptron vs Linear SVMs





- The more rigorous condition for the update in a linear SVM ensures that points near the decision boundary *generalize* better to the test data.

Where does the Perceptron Fail?



- The perceptron fails at similar problems as a linear SVM
  - **Classical solution:** Feature engineering with Radial Basis Function network $\Rightarrow$ Similar to kernel SVM and good for noisy data
  - **Deep learning solution:** Multilayer networks with non-linear activations $\Rightarrow$ Good for data with a lot of structure

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Activation and Loss Functions

Neural Networks and Deep Learning, Springer, 2018
Chapter 1, Section 1.2

Why Do We Need Activation Functions?

- An activation function $\Phi(v)$ in the output layer can control the nature of the output (e.g., probability value in $[0, 1]$)

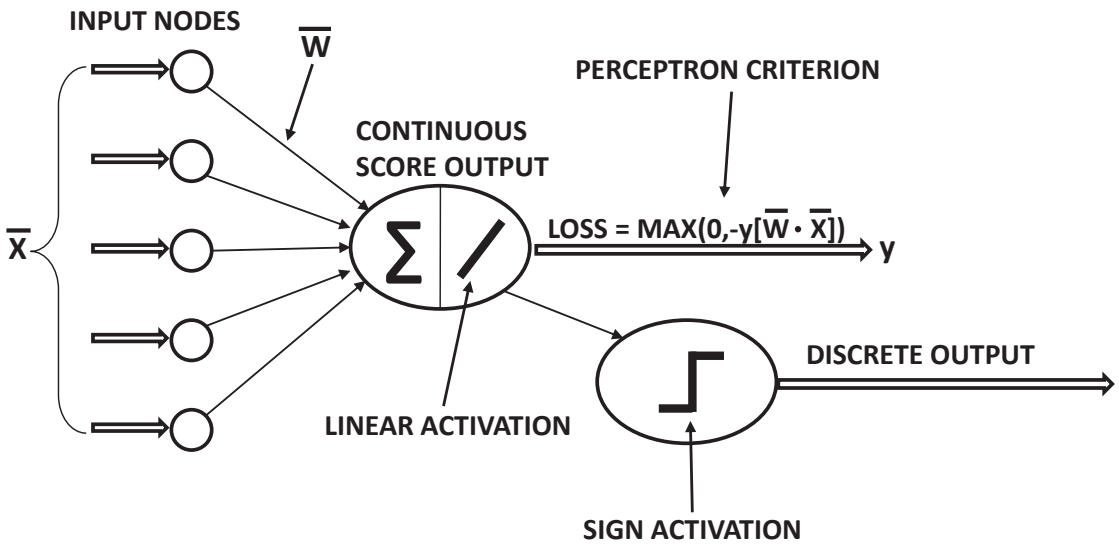
- In *multilayer* neural networks, activation functions bring non-linearity into hidden layers, which increases the complexity of the model.

  – A neural network with any number of layers but only linear activations can be shown to be equivalent to a single-layer network.

- Activation functions required for inference may be different from those used in loss functions in training.

  – Perceptron uses sign function $\Phi(v) = \text{sign}(v)$ for prediction but does not use any activation for computing the perceptron criterion (during training).

Perceptron: Activation Functions



- It is not quite correct to suggest that the perceptron uses sign activation $\Rightarrow$ Only for inference.
  - The perceptron uses *identity activation* (i.e., no activation function) for computing the loss function.
- Typically, a smooth activation function (unlike the sign function) is used to enable a differentiable loss function.

Why Do We Need Loss Functions?

- The loss function is typically paired with the activation function to quantify how far we are from a desired result.

- An example is the perceptron criterion.

$$L_i = \max\{-y(\overline{W} \cdot \overline{X}), 0\}$$

- Note that loss is 0, if the instance $(\overline{X}, y)$ is classified correctly.

- Even though many machine learning problems have discrete outputs, a smooth and continuous loss function is required to enable *gradient-descent* procedures.

- Gradient descent is at the heart of neural network parameter learning.

Identity Activation

- Identity activation $\Phi(v) = v$ is often used in the output layer, when the outputs are real values.

- For a single-layer network, if the training pair is $(\overline{X}, y)$, the output is as follows:

$$\hat{y} = \Phi(\overline{W} \cdot \overline{X}) = \overline{W} \cdot \overline{X}$$

- Use of the squared loss function $(y - \hat{y})^2$ leads to the *linear regression* model with numeric outputs and *Widrow-Hoff learning* with binary outputs.

- Identity activation can be combined with various types of loss functions (e.g., perceptron criterion) even for discrete outputs.

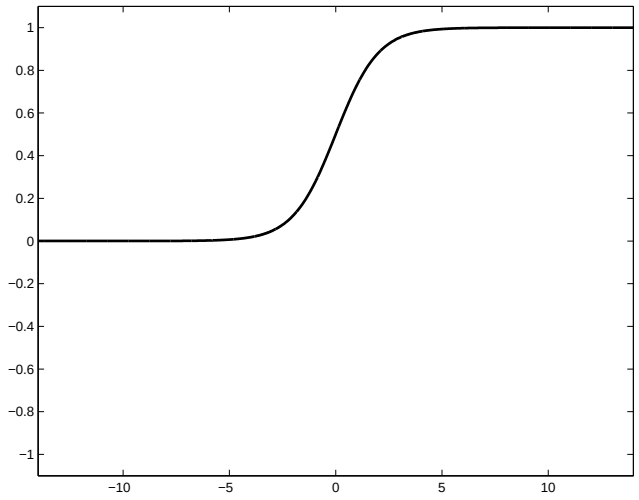
Sigmoid Activation

- Sigmoid activation is defined as $\Phi(v) = 1/(1 + \exp(-v))$.
- For a training pair $(\overline{X}, y)$, one obtains the following prediction in a single-layer network:

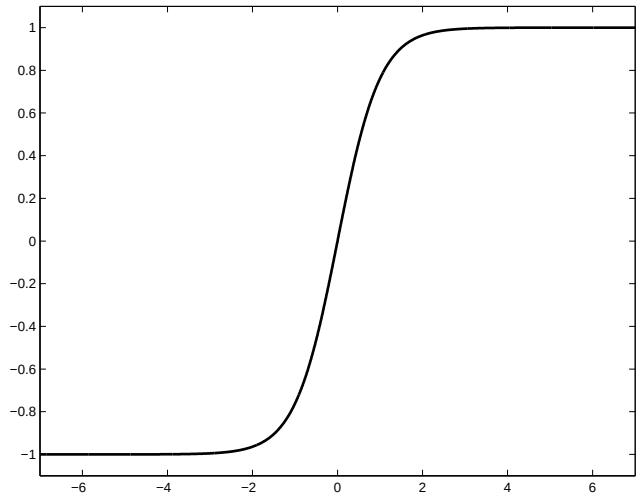
$$\hat{y} = 1/(1 + \exp(-\overline{W} \cdot \overline{X}))$$

- Prediction is the *probability* that class label is $+1$.
- Paired with *logarithmic loss*, which $-\log(\hat{y})$ for positive instances and $-\log(1 - \hat{y})$ for negative instances.
- Resulting model is *logistic regression*.

Tanh Activation



(a) Sigmoid



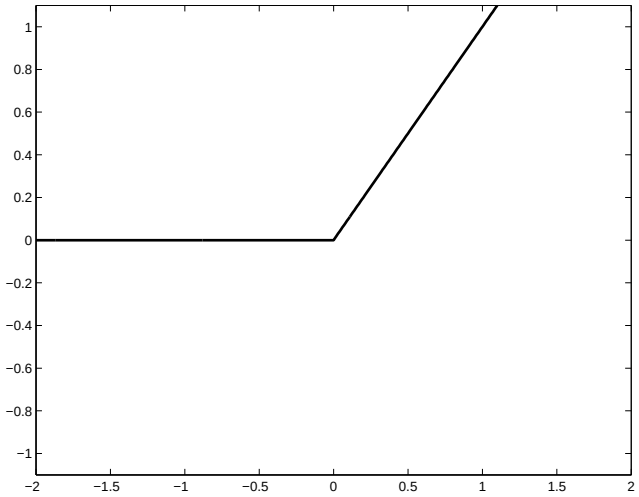
(b) Tanh

- The tanh activation is a scaled and translated version of sigmoid activation.

$$\tanh(v) = \frac{e^{2v} - 1}{e^{2v} + 1} = 2 \cdot \text{sigmoid}(2v) - 1$$

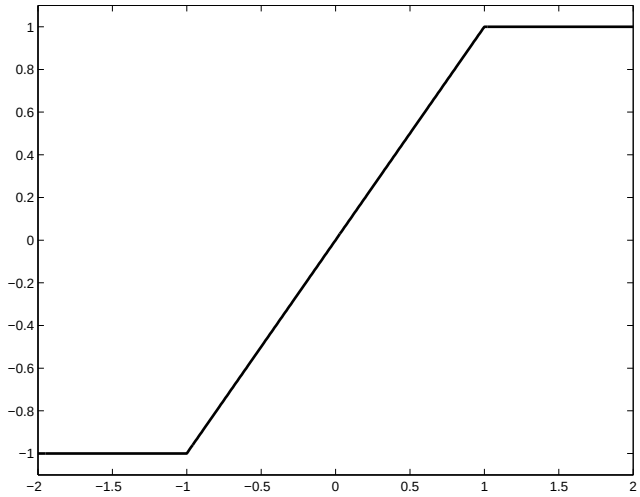
- Often used in hidden layers of multilayer networks

Piecewise Linear Activation Functions



(a) ReLU

$\Phi(v) = \max\{v, 0\}$



(b) Hard Tanh

$\Phi(v) = \max\{\min[v, 1], -1\}$

- Piecewise linear activation functions are easier to train than their continuous counterparts.

Softmax Activation Function

- All activation functions discussed so far map scalars to scalars.

- The softmax activation function maps vectors to vectors.

- Useful in mapping a set of real values to probabilities.

  – Generalization of sigmoid activation, which is used in *multiway* logistic regression.

- Discussed in detail in later lectures.

Derivatives of Activation Functions

- Neural network learning requires gradient descent of the loss.

- Loss is often a function of the output o, which is itself obtained by using the activation function:

$$o = \Phi(v) \tag{1}$$

- Therefore, we often need to compute the partial derivative of o with respect to v during neural network parameter learning.

- Many derivatives are more easily expressed in terms of the output o rather than input v.

Useful Derivatives

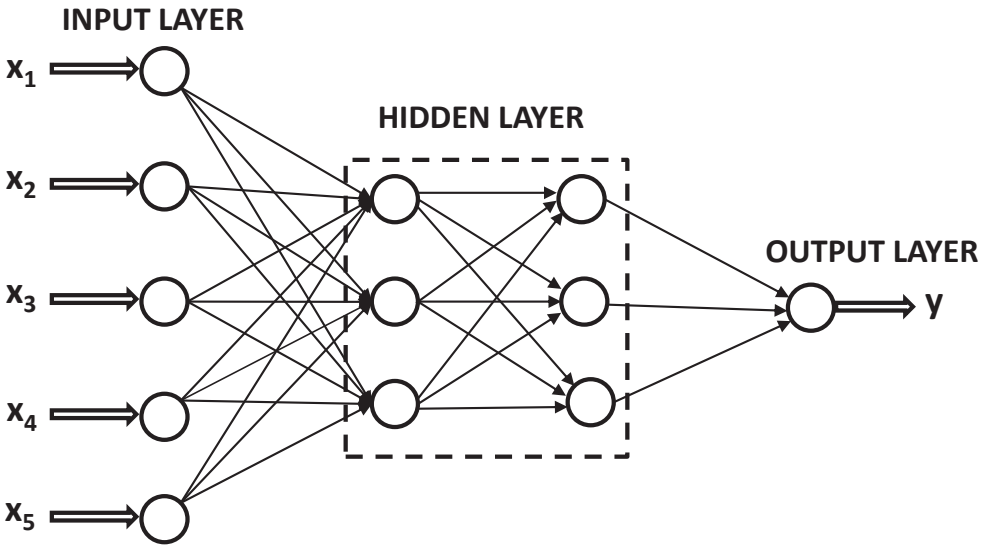
- Sigmoid: $\frac{\partial o}{\partial v} = o(1 - o)$
- Tanh: $\frac{\partial o}{\partial v} = 1 - o^2$
- ReLU: Derivative is 1 for positive values of v and 0 otherwise.
- Hard Tanh: Derivative is 1 for $v \in (-1, 1)$ and 0 otherwise.
- Best to commit to memory, since they are used repeatedly in neural network learning.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Multilayer Neural Networks

Neural Networks and Deep Learning, Springer, 2018
Chapter 1, Section 1.3–1.6

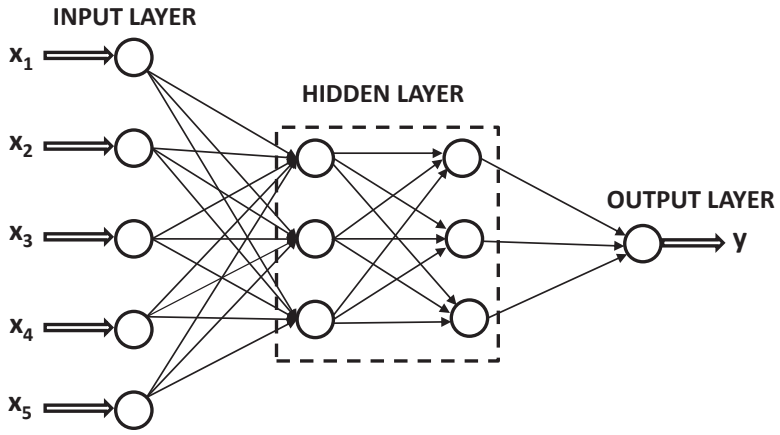
Multilayer Neural Networks



- In multilayer networks, the output of a node can feed into other *hidden* nodes, which in turn can feed into other hidden or output nodes.

- Multilayer neural networks are *always* directed acyclic graphs and *usually* arranged in layerwise fashion.

Multilayer Neural Networks

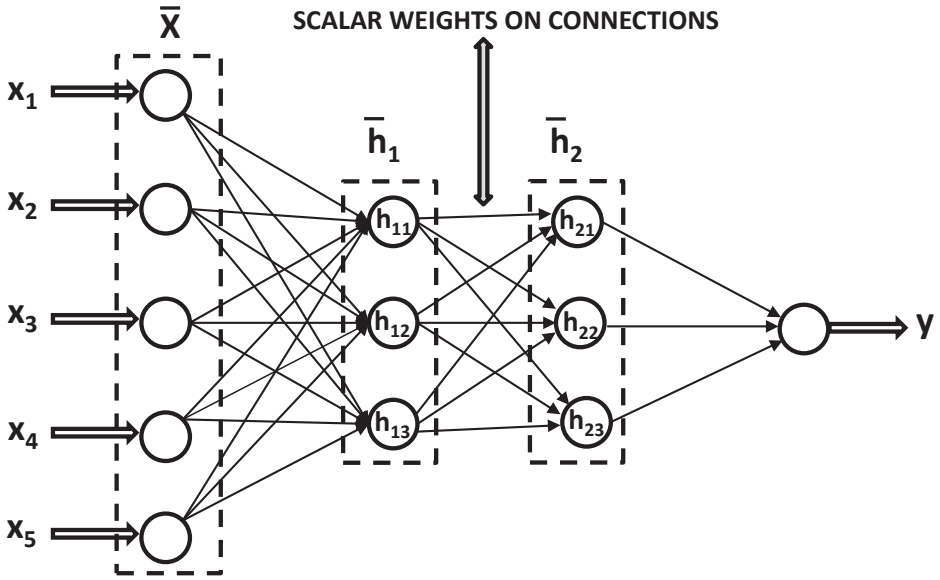


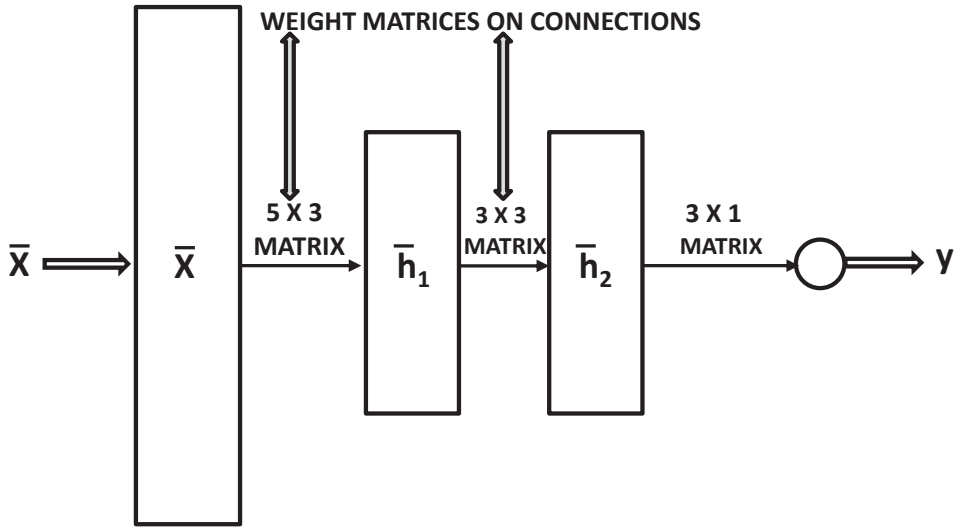
- The layers between the input and output are referred to as *hidden* because they perform intermediate computations.

- Each hidden node uses a combination of a linear transformation and an activation function $\Phi(\cdot)$ (like the output node of the perceptron).

- The use of *nonlinear* activation functions in the hidden layer is crucial in increasing learning capacity.

Scalar versus Vector Diagrams





- Hidden vectors $\overline{h}_1 \ldots \overline{h}_k$ and weight matrices $W_1 \ldots W_{k+1}$.

$$\overline{h}_1 = \Phi(W_1^T \overline{x}) \qquad \text{[Input to Hidden Layer]}$$
$$\overline{h}_{p+1} = \Phi(W_{p+1}^T \overline{h}_p) \quad \forall p \in \{1 \ldots k-1\} \qquad \text{[Hidden to Hidden Layer]}$$
$$\overline{o} = \Phi(W_{k+1}^T \overline{h}_k) \qquad \text{[Hidden to Output Layer]}$$

- $\Phi(\cdot)$ is typically applied element-wise (other than softmax).

Using Activation Functions

- The nature of the activation in output layers is often controlled by the nature of output
  - Identity activation for real-valued outputs, and sigmoid/softmax for binary/categorical outputs.
  - Softmax almost exclusively for output layer and is paired with a particular type of *cross-entropy* loss.
- Hidden layer activations are almost always nonlinear and often use the same activation function over the entire network.
  - Tanh often (but not always) preferable to sigmoid.
  - ReLU has largely replaced tanh and sigmoid in many applications.

Why are Hidden Layers Nonlinear?

- A multi-layer network that uses only the identity activation function in all its layers reduces to a single-layer network that performs linear regression.

$$\begin{aligned}
\overline{h}_1 &= \Phi(W_1^T \overline{x}) = W_1^T \overline{x} \\
\overline{h}_{p+1} &= \Phi(W_{p+1}^T \overline{h}_p) = W_{p+1}^T \overline{h}_p \quad \forall p \in \{1 \ldots k-1\} \\
\overline{o} &= \Phi(W_{k+1}^T \overline{h}_k) = W_{k+1}^T \overline{h}_k
\end{aligned}$$

- We can eliminate the hidden variable to get a simple linear relationship:

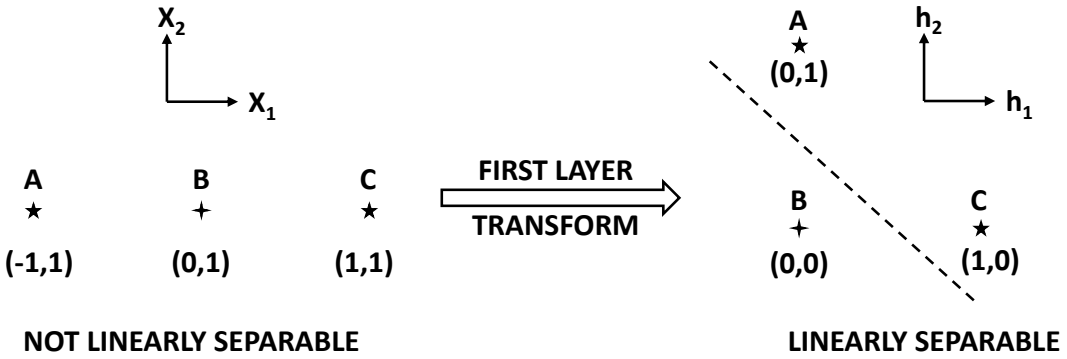
$$\begin{aligned}
\overline{o} &= W_{k+1}^T W_k^T \ldots W_1^T \overline{x} \\
&= \underbrace{(W_1 W_2 \ldots W_{k+1})^T}_{W_{xo}^T} \overline{x}
\end{aligned}$$

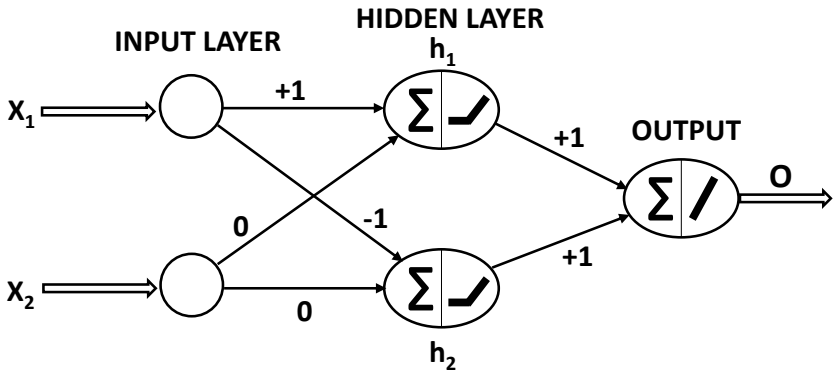
- We get a *single-layer* network with matrix W_{xo}.

Role of Hidden Layers

- Nonlinear hidden layers perform the role of hierarchical feature engineering.
  - Early layers learn primitive features and later layers learn more complex features
  - Image data: Early layers learn elementary edges, the middle layers contain complex features like honeycombs, and later layers contain complex features like a part of a face.
  - *Deep learners are masters of feature engineering*.
- The final output layer is often able to perform inference with transformed features in penultimate layer relatively easily.
- **Perceptron:** Cannot classify linearly inseparable data but can do so with *nonlinear* hidden layers.

Example of Classifying Inseparable Data

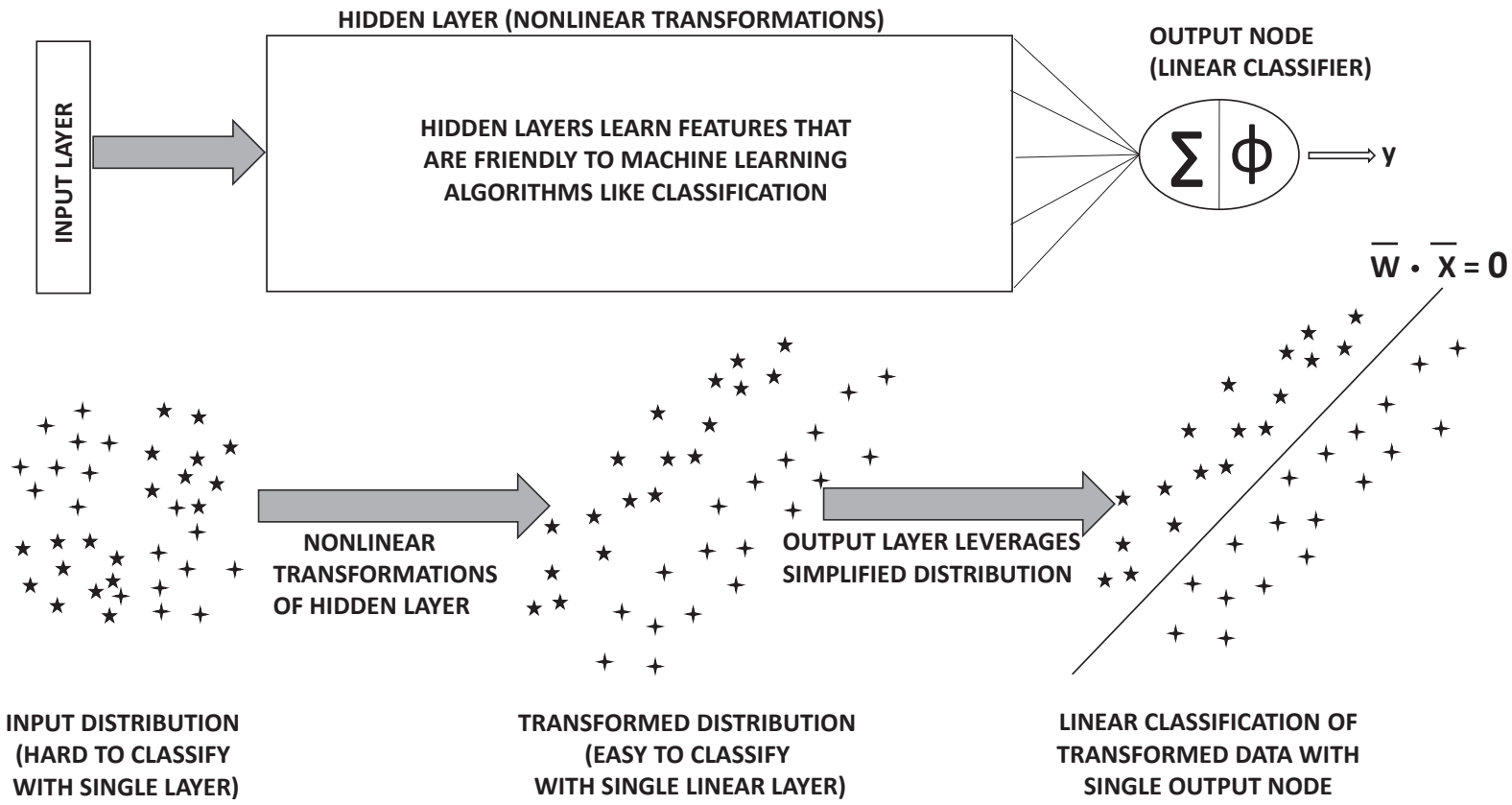




- The hidden units have ReLU activation, and they learn the two new features h_1 and h_2 with linear separator $h_1 + h_2 = 0.5$:

$$h_1 = \max\{x_1, 0\} \quad h_2 = \max\{-x_1, 0\}$$

The Feature Engineering View of Hidden Layers



- Early layers play the role of feature engineering for later layers.

Multilayer Networks as Computational Graphs

- Consider the case in which each layer computes the vector-to-vector function f_i.
- The overall *composition* function is $f_k o f_{k-1} o \ldots o f_1$
- Function of a function with k levels of recursive nesting!
- It is a complex and ugly-looking function, which is powerful and typically cannot be expressed in closed form.
- The neural network architecture is a directed acyclic computational graph in which each layer is often fully connected.

How Do We Train?

- We want to compute the derivatives with respect to the parameters in all layers to perform gradient descent.

- The complex nature of the composition function makes this difficult.

- The key idea to achieve this goal is *backpropagation*.

- Use the *chain rule of differential calculus* as a dynamic programming update on a directed acyclic graph.

- Details in later lectures.