Lecture 6: Artificial Neural Networks

Dr.-Ing. Sudchai Boonto, Assistant Professor

March 18, 2018

Department of Control System and Instrument Engineering, KMUTT

Introduction

Why (Artificial) Neural Networks

- (Neuro-)Biology / (Neuro-)Physiology / Psychology:
  - Exploit similarity to real (biological) neural networks
  - Build models to understand nerve and brain operation by simulation.
- Computer Science/ Engineering / Economics
  - Mimic certain cognitive capabilities of human beings.
  - Solve learning/ adaptation, prediction, and optimization problems.
- Physics / Chemistry
  - Use neural network models to describe physical phenomena.
  - Special case: spin glasses (alloys of magnetic and non-magnetic methods).

Why Neural Networks in AI?

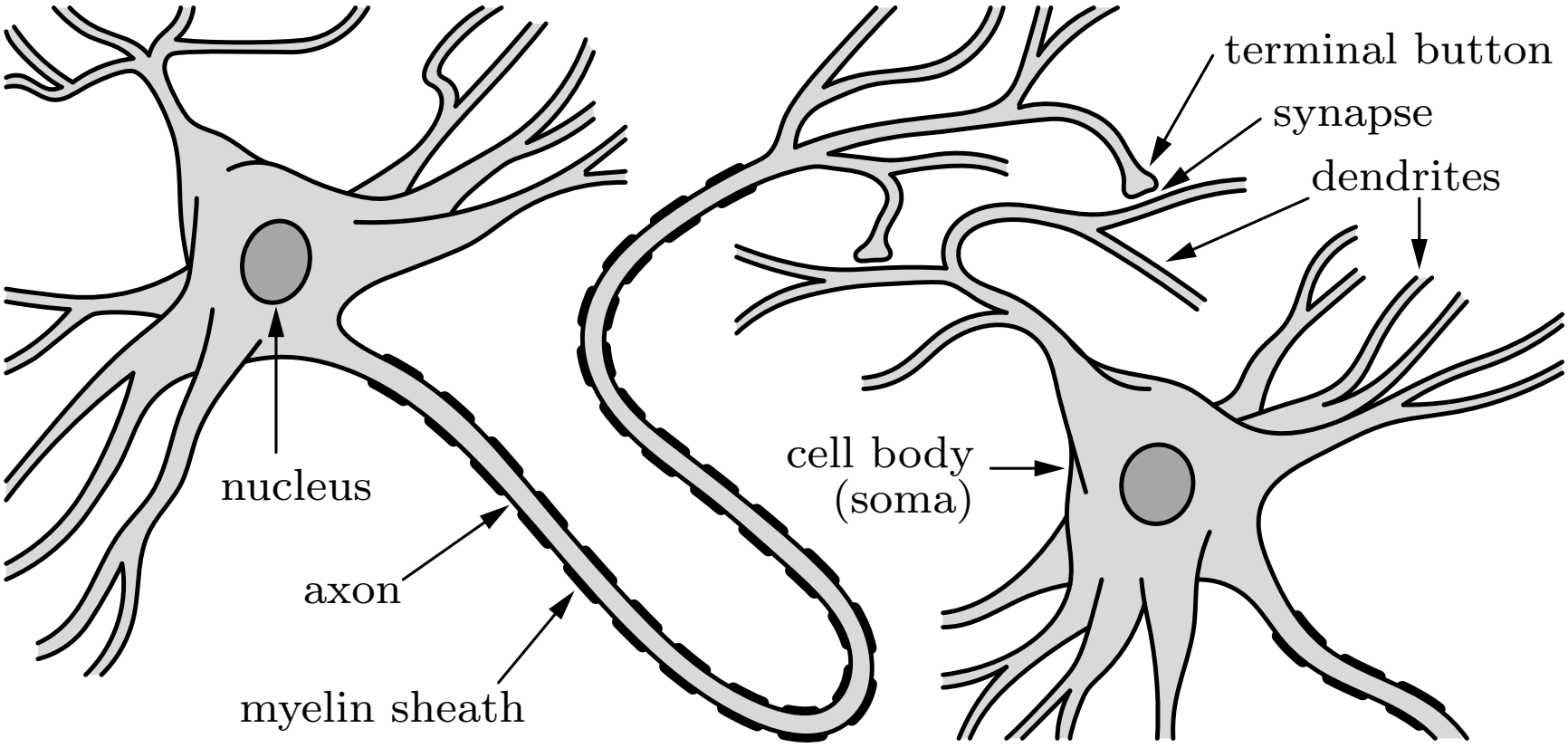
Physical-Symbol System Hypothesis [Newell and Simon 1976] A physical-symbol system has the necessary and sufficient means for general intelligent action.

Neural networks process simple signals, not symbols

So why study neural networks in Artificial Intelligence?

- Symbol-based representations work well for inference tasks, but are fairly bad for perception tasks.
- Symbol-based expert systems tend to get slower with growing knowledge, human experts tend to get faster.
- Neural networks allow for highly parallel information processing.
- There are several successful applications in industry and finance.

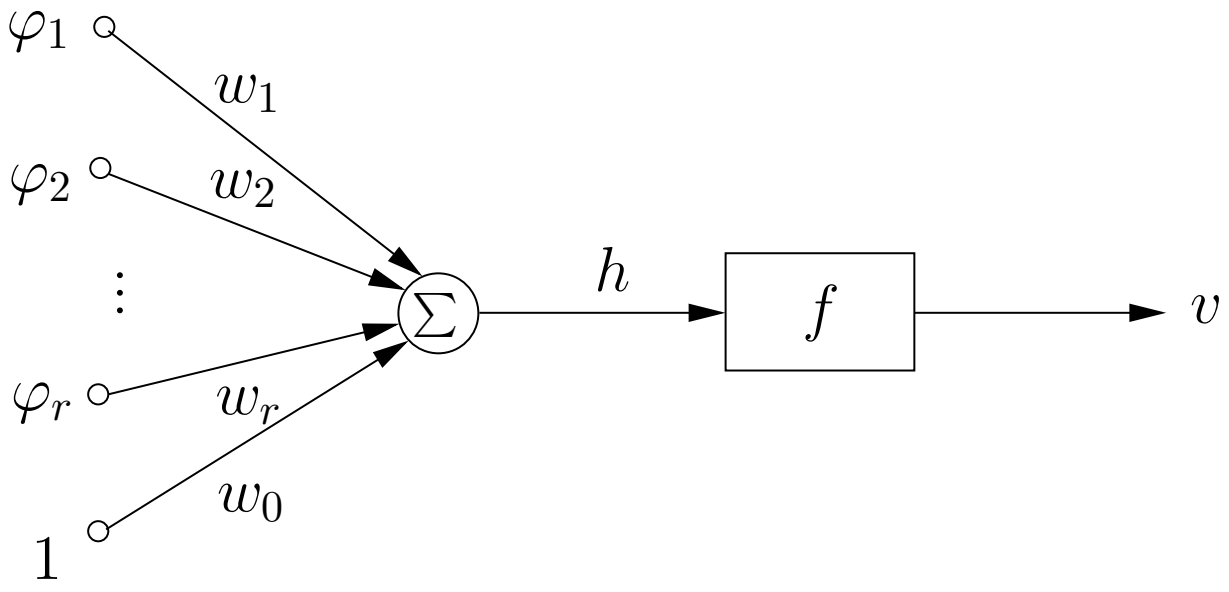
Biological Neuron



Biological Neuron cont.

- **dendrites** represent a highly branching tree of fibres that carry electrical signals to the cell body. 10^3 to 10^4 dendrites per neuron. The input channels of a neuron.
- **soma** or cell body realizes the logical functions of the neuron.
- **axon (nucleus)** is a single long nerve fibre attached to the soma that serves as the output channel of the neuron.
- **synapse** is a point of contact between an axon of one cell and a dendrite of another cell.

Artificial Neurons (Single-layer Perceptron)



- the neuron is modelled as a multi-input nonlinear device with r inputs $\varphi_1, \varphi_2, \ldots, \varphi_r$, one output v and weighted interconnections w_i.
- an extra input with a value fixed to 1 is provided that can be used to generate a *bias* w_0.

Artificial Neurons cont.

The sum h of the r weighted inputs and the bias is passed through a static nonlinear function $f(h)$ according to

$$v = f(h) = f\left(\sum_{i=1}^{r} w_i \varphi_i + w_0\right)$$

Introducing the column vectors

$$w = \begin{bmatrix} w_1 \\ \vdots \\ w_r \end{bmatrix}, \qquad \varphi = \begin{bmatrix} \varphi_1 \\ \vdots \\ \varphi_r \end{bmatrix}$$

this can be written as

$$v = f(w^T \varphi + w_0)$$

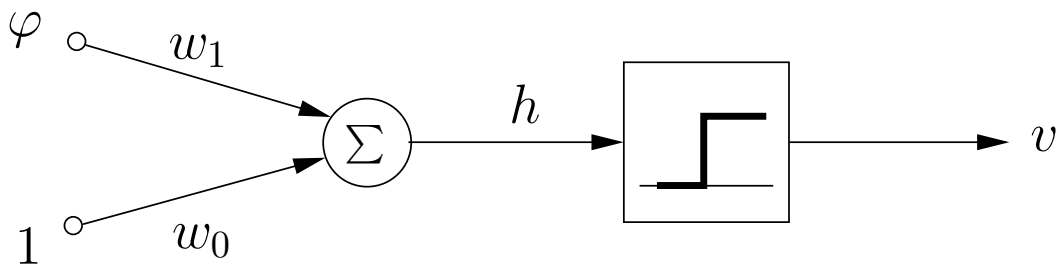
Activation Functions

- The function f is called the **activation function** of the neuron, which can be linear or nonlinear function.
- The activation function is used as a decision making body at the output of a neuron. The neuron learns linear of nonlinear decision boundaries based on the activation function. It also has a normalizing effect (maps the resulting values in between 0 to 1 or -1 to 1 etc.) on the neuron output which prevents the output of neurons after several layers to become very large, due to the cascading effect.
- Several types of activation function are commonly used:
    - step functions
    - linear functions
    - sigmoid functions
    - softmax functions
    - recified linear functions
    - leaky ReLU functions
    - softplus functions

Step Activation Function

The step function as activation function is defined by

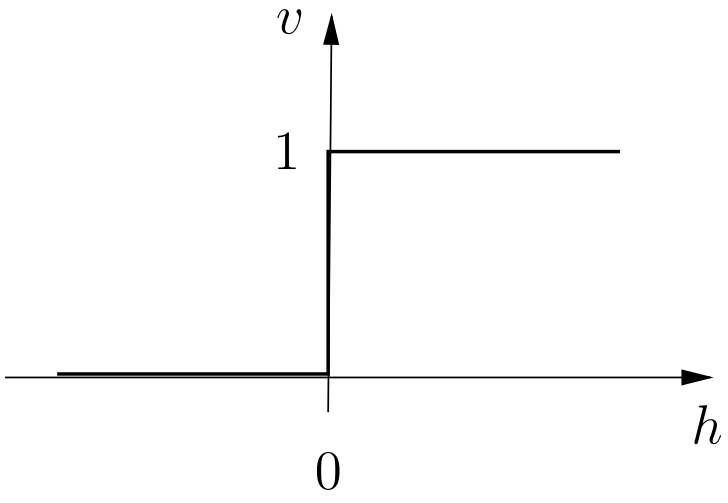
$$v(h) = \sigma(h) = \begin{cases} 1, & h \geq 0, \\ 0, & h < 0. \end{cases}$$

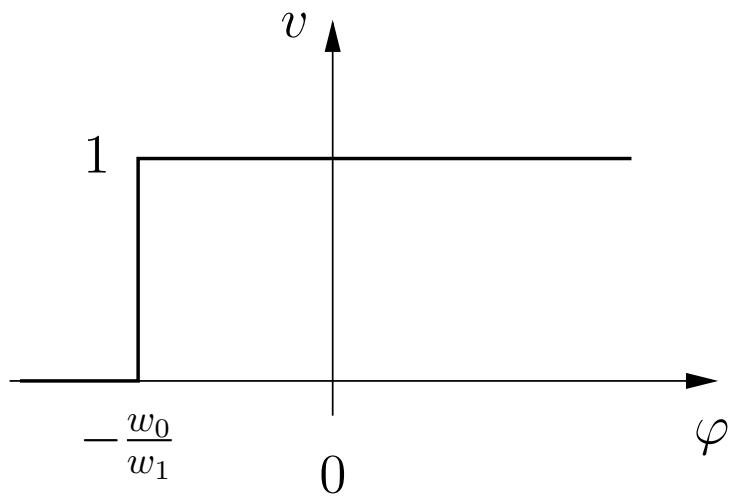


The output is

$$v = \sigma(w_1\varphi + w_0)$$

Step Activation Function cont.

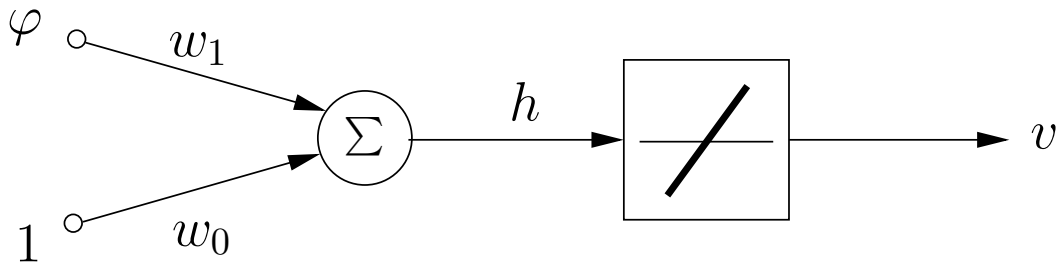


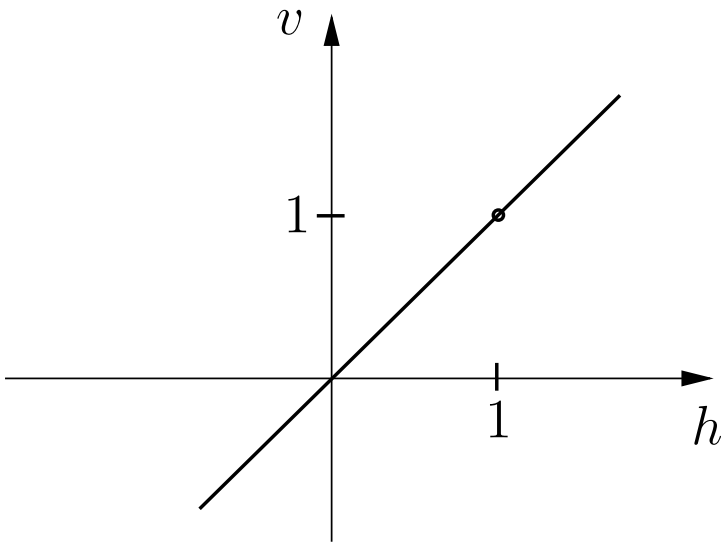


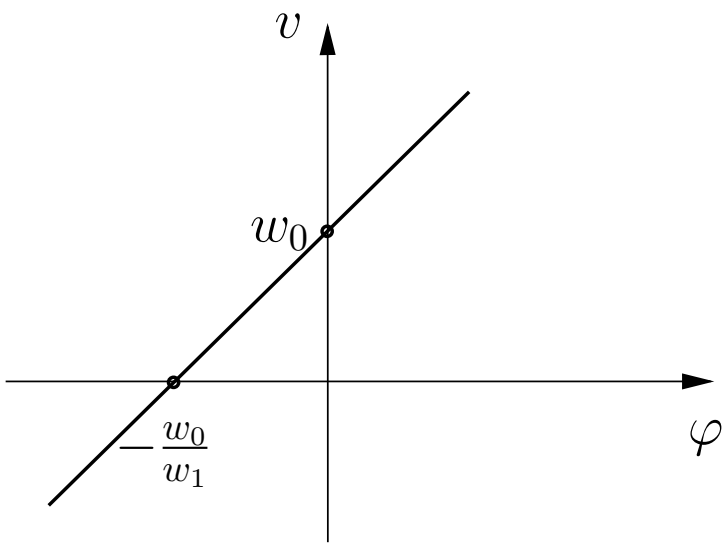
Linear Activation Function

The output of a linear activation function is equal to its input

$$v(h) = h$$







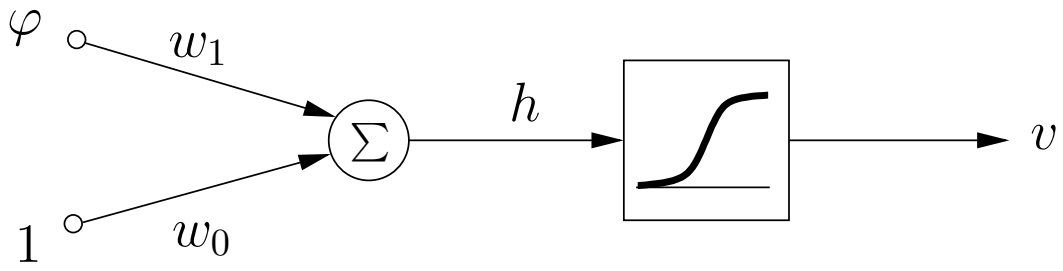
Nonlinear Activation Function

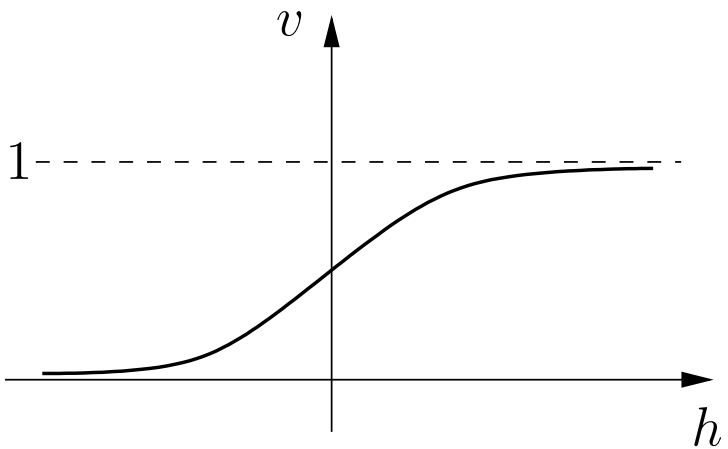
- In theory, when an activation function is non-linear; a two-layer Neural Network can approximate any function (given a sufficient number of units in the hidden layer). The nonlinear activation functions are the most used activation functions.
- The main terminologies needed to understand for nonlinear functions are:
  - **Derivative or Differential:** Change in y-axis with respect to change in x-axis.
  - **Monotonic:** A function varying in such a way that it either never decreases or never increases.
- The nonlinear activation functions are mainly divided on the basis of their **range or curves**
- A function whose range is finite leads to a more stable performance with respect to gradient-based methods.
- Smooth functions are preferred (empirical evidence) and Monolithic functions for a single layer lead to convex error surfaces.
- Are symmetric around the origin and behave like identity functions near the origin ($f(x) = x$).

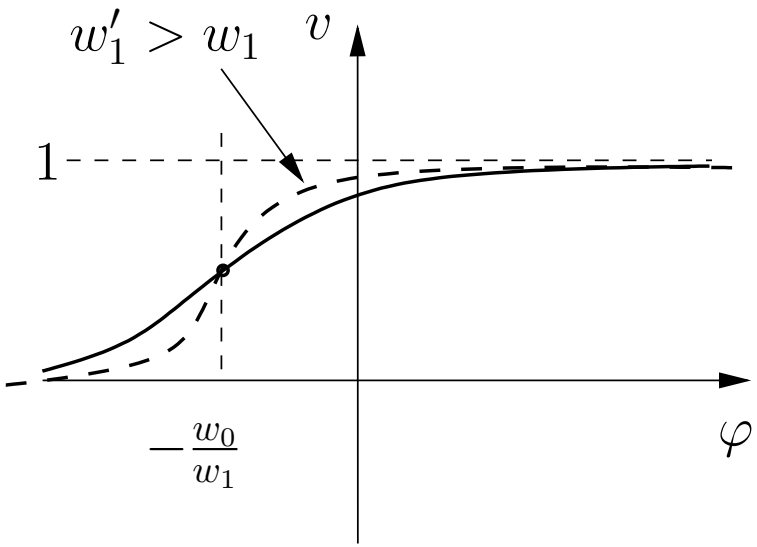
Sigmoid (logistic) Activation Function

The logistic sigmoid activation function is

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







Sigmoid (logistic) Activation Function

Derivative of the sigmoid function

$$
\begin{aligned}
f(h) &= \left(1 + e^{-h}\right)^{-1} \\
\frac{df(h)}{dh} &= (-1)\left(1 + e^{-h}\right)^{-2} \frac{d}{dh}\left(1 + e^{-h}\right) \\
&= \left(1 + e^{-h}\right)^{-2}\left(e^{-h}\right) \\
&= \frac{e^{-h}}{\left(1 + e^{-h}\right)^2}
\end{aligned}
$$

Nicer form

$$
\begin{aligned}
\frac{df(h)}{dh} &= \frac{e^{-h} + 1 - 1}{\left(1 + e^{-h}\right)^2} \\
&= \frac{\left(1 + e^{-h}\right)}{\left(1 + e^{-h}\right)^2} - \frac{1}{\left(1 + e^{-h}\right)^2} \\
&= \frac{1}{1 + e^{-h}} - \frac{1}{\left(1 + e^{-h}\right)^2} = \frac{1}{1 + e^{-h}}\left(1 - \frac{1}{1 + e^{-h}}\right) = f(h)(1 - f(h))
\end{aligned}
$$

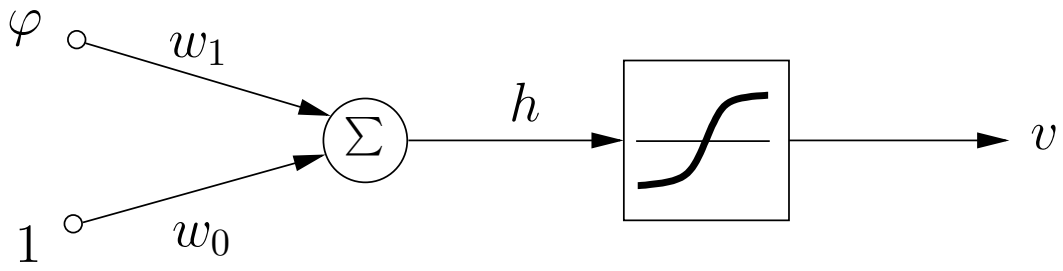
Sigmoid (logistic) Activation Function cont.

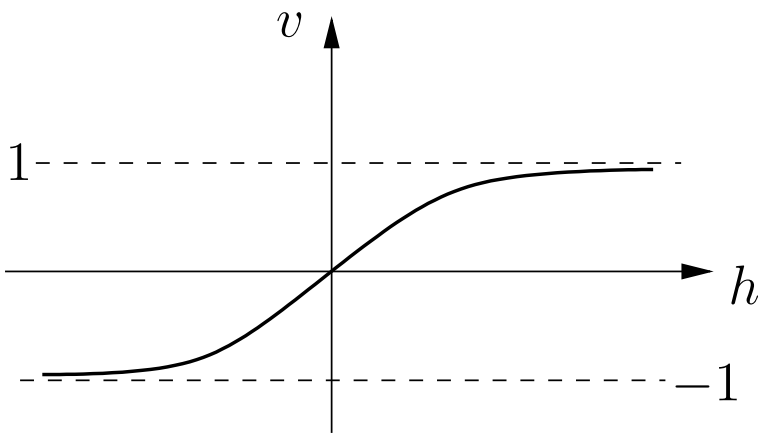
- It is real value, existing between (0 to 1).
- It is especially used for models where we have to **predict the probability** as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice.
- The function is **differentiable**. That means, we can find the slope of the sigmoid curve at any two points. It is non-negative value, if a number is greater than or equal to zero. It is non-positive, if a number is less than or equal to zero.
- The function is **monotonic** but functions's derivative is not.
- The logistic sigmoid function can cause a neural network to get stuck at the training time.

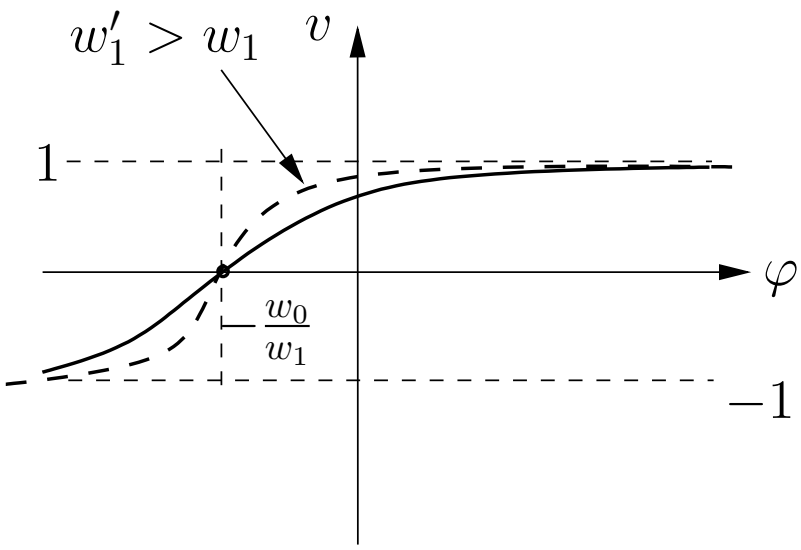
Hyperbolic tangent (tanh) Activation Function

The Hyperbolic tangent (tanh) activation function is

$$v(h) = f(h) = \frac{e^h - e^{-h}}{e^h + e^{-h}}$$







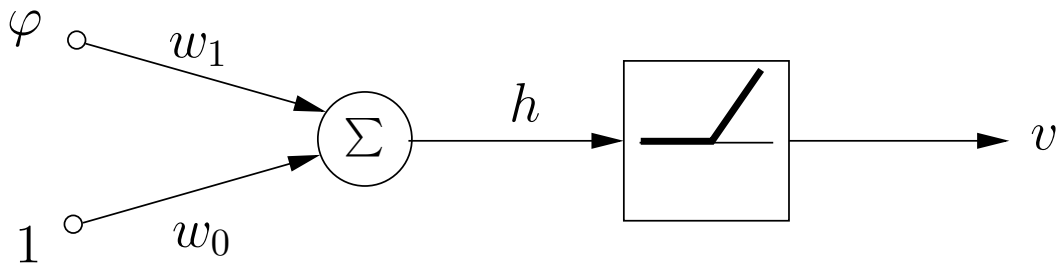
Hyperbolic tangent (tanh) Activation Function

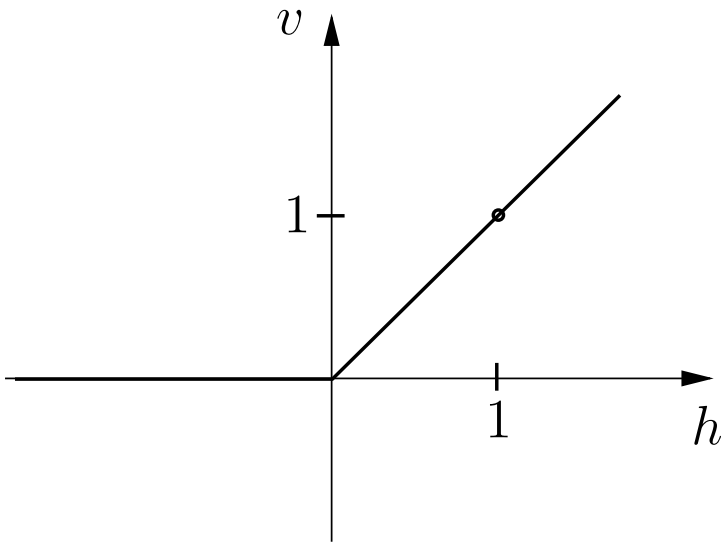
- $\tanh$ is also like logistic sigmoid but better in term of the range of the $\tanh$ function is from -1 to 1.
- $\tanh$ is also sigmoidal (s-shaped).
- The advantage of this activation function is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the $\tanh$ graph.
- The function is **monotonic** but functions's derivative is not.
- The $\tanh$ function is mainly used classification between two classes.

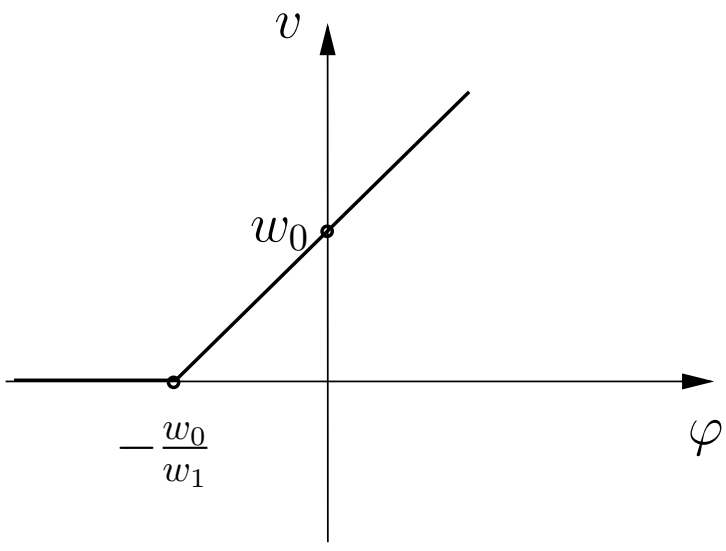
ReLU (rectified linear unit) Activation Function

The ReLU (Rectified Linear Unit) activation function is

$$v(h) = f(h) = \max(0, h)$$







Variant ReLU (rectified linear unit) Activation Function

Noisy ReLUs

Rectified linear units can be extended to include Gaussian noise, making them noisy RELus, giving

$$v(h) = f(h) = \max(0, h + Y), \text{ with } Y \sim \mathcal{N}(0, \sigma(h))$$

Leaky ReLUs

Leaky ReLUs allow a small, non-zero gradient when the unit is not active.

$$v(h) = f(h) = \begin{cases} h, & h > 0 \\ 0.01h, & \text{otherwise} \end{cases}$$

Parametric ReLUs take this idea further by making the coefficient of leakage into a parametr that is learned alogn with the other neural network parameters.

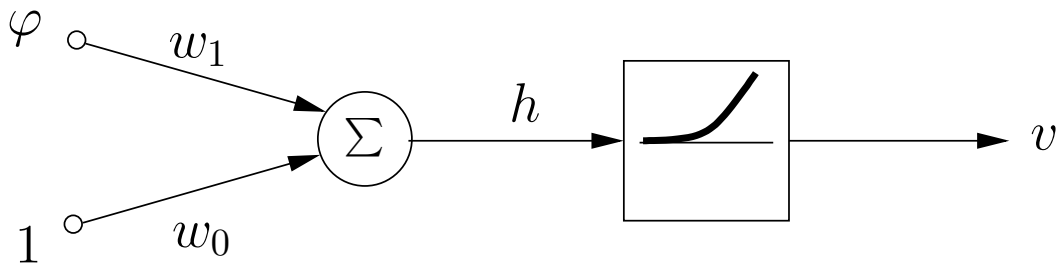
$$v(h) = f(h) = \begin{cases} h & h > 0 \\ ax & \text{otherwise} \end{cases}$$

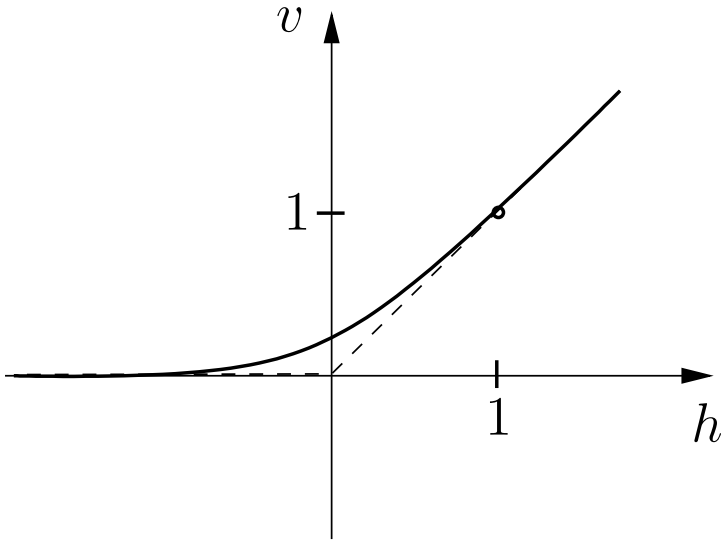
Note that for $a \leq 1$, this is equivalent to $f(h) = \max(h, ah)$.

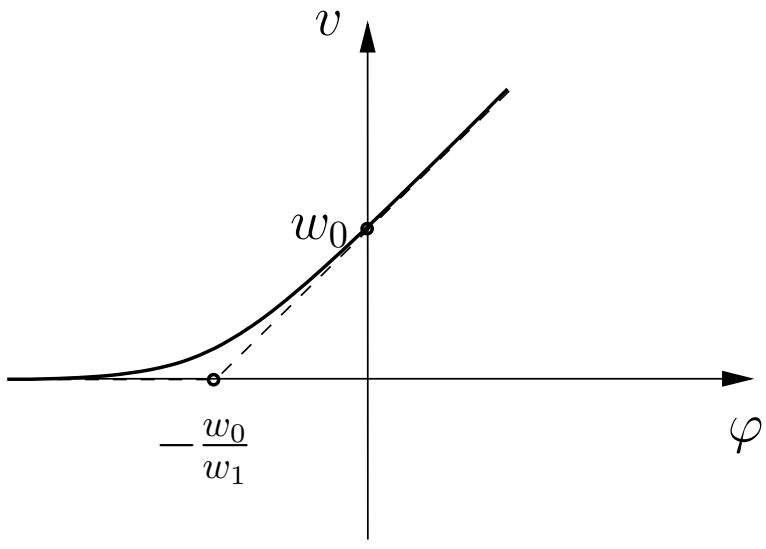
Softplus Activation Function

This activation function is considered to be the smoother version of ReLU. The softplus activation function is

$$v(h) = f(h) = \ln(1 + e^x)$$







Softmax Activation Function

The softmax activation function is

$$v(h) = f(h) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}, \; i = 0, 1, 2, \ldots, k$$

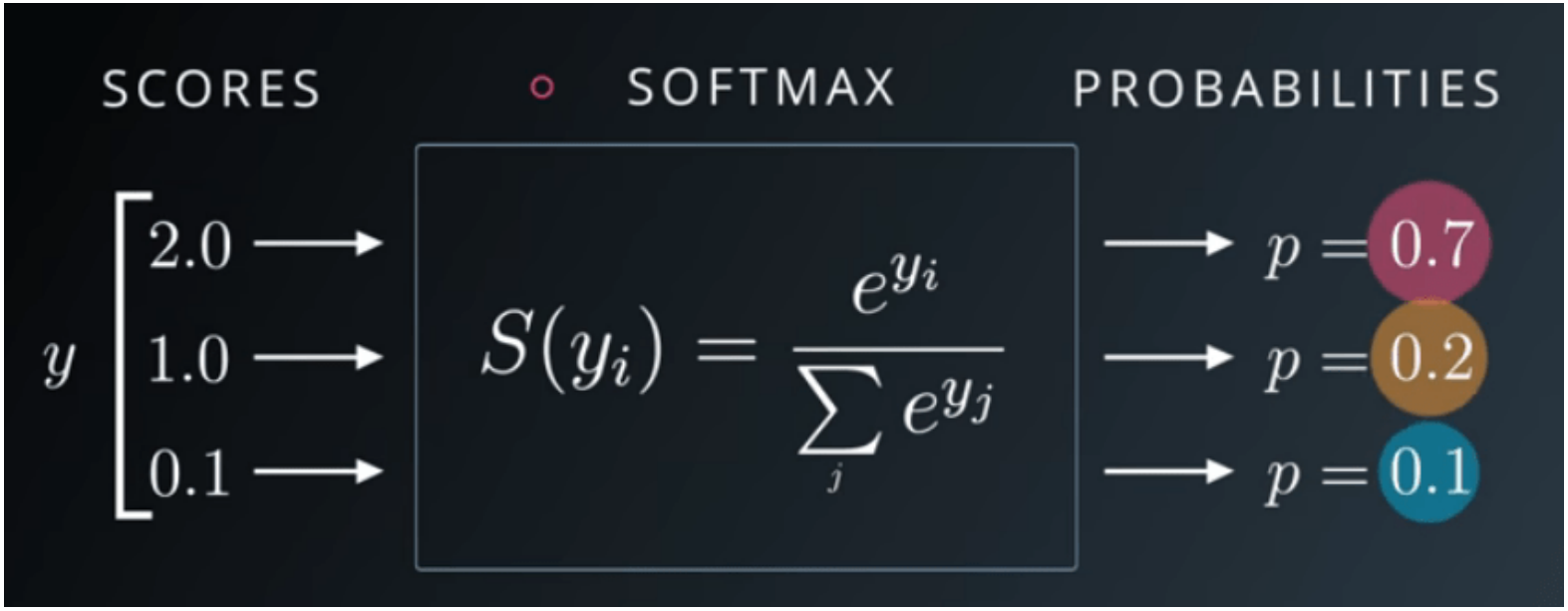


Figure 1:

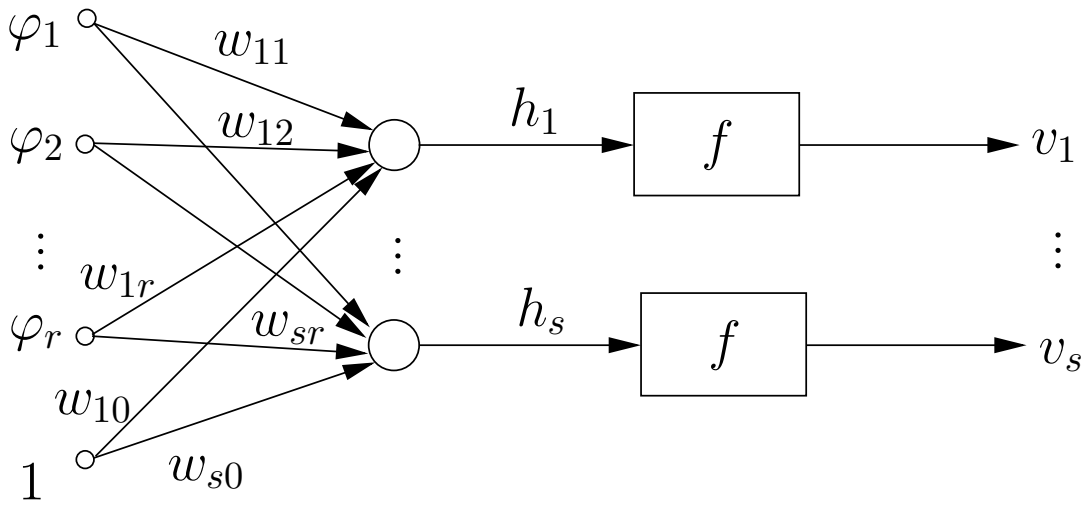
https://sefiks.com/2017/11/08/softmax-as-a-neural-networks-activation-function/

Softmax Activation Function

- Softmax is a generalization of logistic regression in as much as it can be applied to continuous data (rather than classifying binary) and can contain multiple decision boundaries.
- It handles multinomial labeling systems. It is the function you will often find at the output layer of a classifier.
- The softmax activation function returns the probability distribution over mutually exclusive output classes.
- If we have a multiclass modeling problem yet we care only about the best score across these classes, we'd use softmax output layer with a function to get the highest score of all the classes.

Artificial Neural Network

We can turn to networks formed by connecting single neurons.



The network has r inputs $\varphi_1, \ldots, \varphi_r$, a bias and s outputs $v_1, \ldots, v_s$.

Artificial Neural Network cont.

The output of the summing junction of the i^{th} neuron is

$$h_i = \begin{bmatrix} w_{i1} & w_{i2} & \cdots & w_{ir} \end{bmatrix} \begin{bmatrix} \varphi_1 \\ \vdots \\ \varphi_2 \end{bmatrix} + w_{i0}$$

where w_{ij} is the gain from input j to the i^{th} neuron. Defining the weight vector

$$w_i^T = \begin{bmatrix} w_{i1} & w_{i2} & \cdots & w_{ir} \end{bmatrix}$$

we have

$$\begin{bmatrix} h_1 \\ \vdots \\ h_s \end{bmatrix} = \begin{bmatrix} w_1^T \\ \vdots \\ w_s^T \end{bmatrix} \varphi + \begin{bmatrix} w_{10} \\ \vdots \\ w_{s0} \end{bmatrix}$$

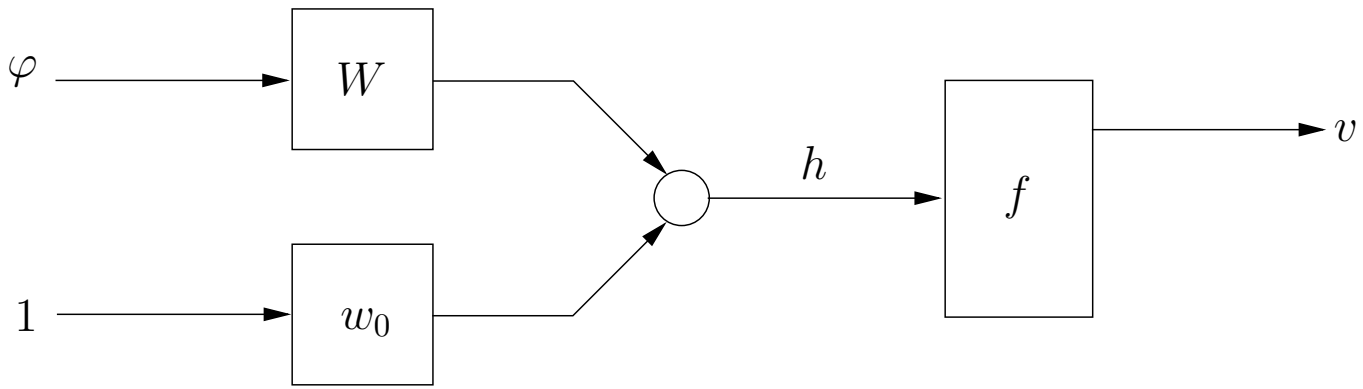
$$h = W\varphi + w_0$$

Artificial Neural Network cont.

The vector v of network outputs is then

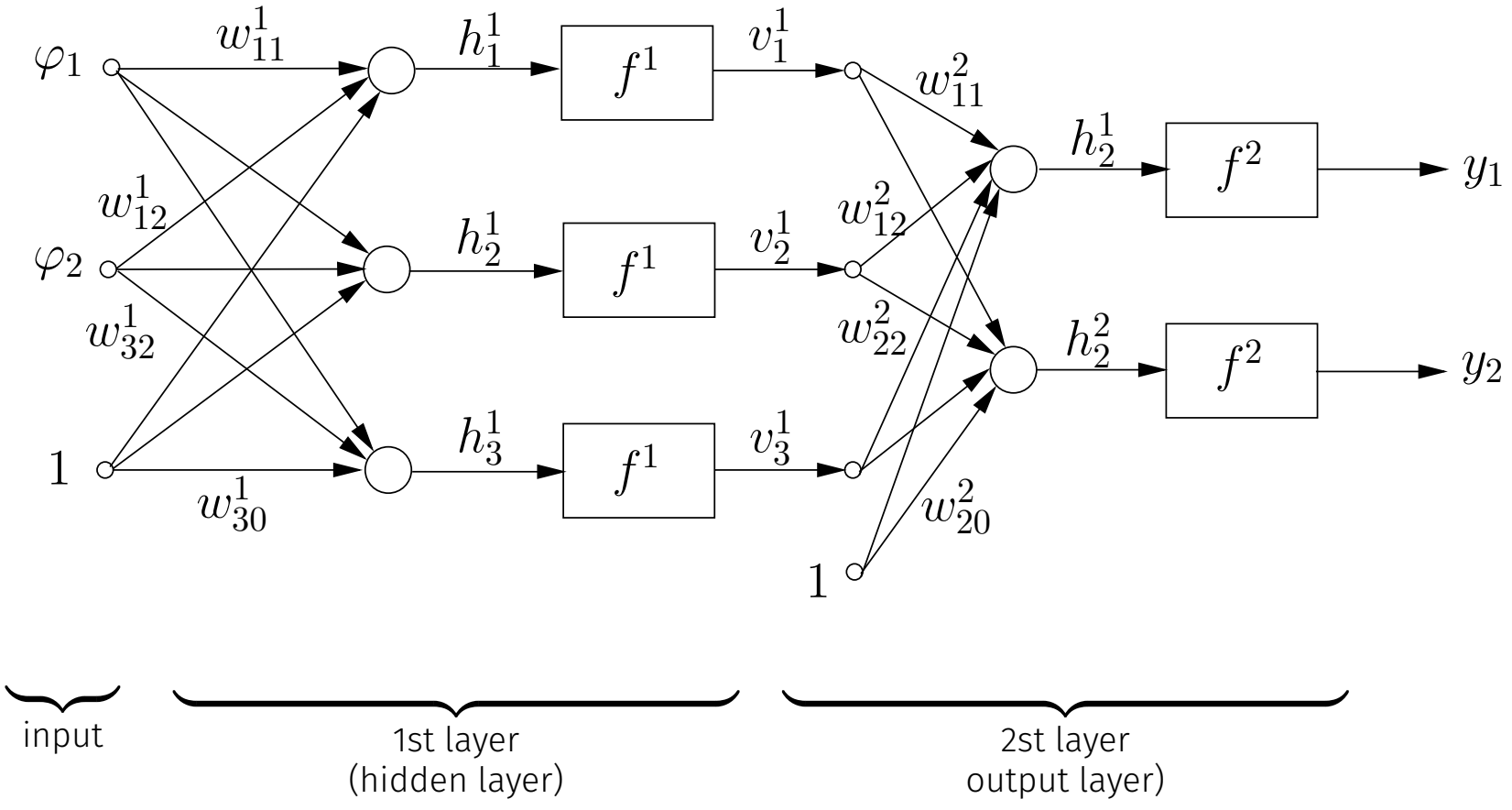
$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_s \end{bmatrix} = f(W\varphi + w_0)$$

A single-layer network of this form is called a perceptron network.



Multilayer Perceptron Network

Several perceptron layers can be connected in series to form a multilayer perceptron (MLP)



Multilayer Perceptron Network cont.

At the output of the first layer we have

$$v^1 = \begin{bmatrix} v_1^1 \\ v_2^1 \\ v_3^1 \end{bmatrix} = f^1(W^1\varphi + w_0^1)$$

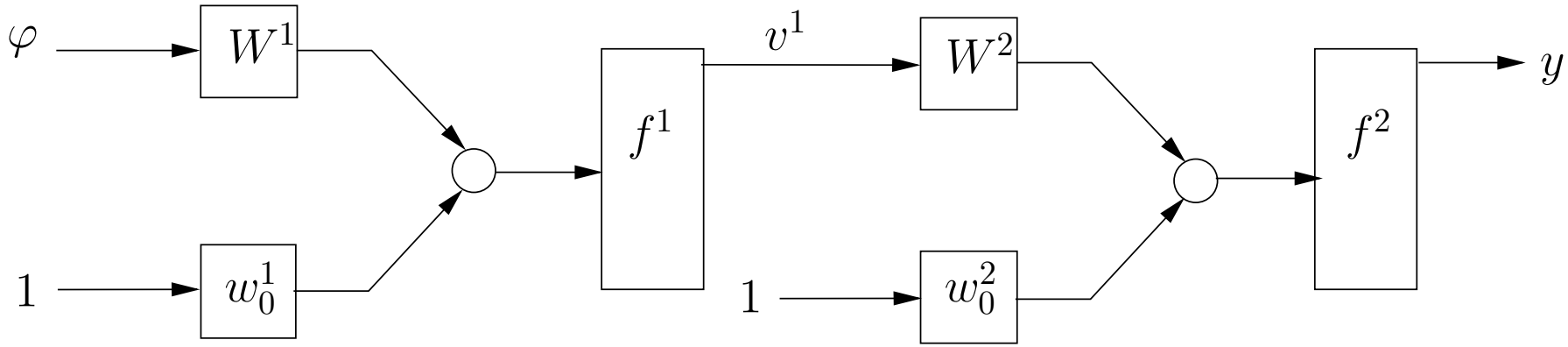
The network output – the output of the second layer is

$$y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = f^2(W^2 v^1 + w_0^2)$$

or in general

$$y = f^2(W^2 f^1(W^1\varphi + w_0^1) + w_0^2)$$

Multilayer Perceptron Network cont.



Universal Approximation

- a commonly used network structure is a two-layer perceptron network with sigmoidal activation functions in the hidden layer, and linear activation functions in the output layer.
- An important property of such networks is their **universal approximation capability**
- Any given real continuous function $g : \mathbb{R}^r \rightarrow \mathbb{R}$ can be approximated to any desired accuracy by a two-layer **sig-lin** networks
- however on indication about the number of hidden units required for achieving the desired accuracy.

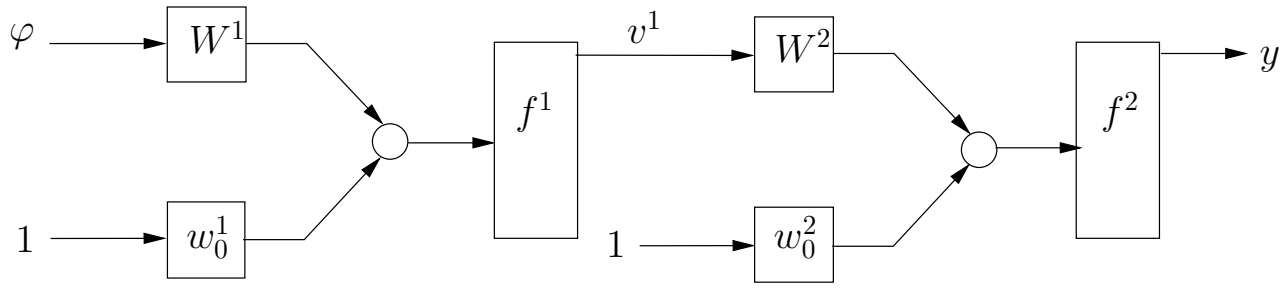
Training Neural Networks

General neural network training pseudocode

```
function neural-network-learning( training-records ) returns network
    network <- initialize weights (randomly)
    start loop
        for each example in training-records do
            network-output = neural-network-output( network, example )
            actual-output = observed outcome associated with example
            update weights in network based on
                { example, network-output, actual-output }
        end for
    end loop when all examples correctly predicted or hit stopping conditions
    return network
```

- The key is to distribute the blame for the error and divide it between the contributing weights.
- With the preceptron learning algorithm, it's easy because there is only one weight per input to influence the output value.
- With feed-forward multilayer networks learning algorithms have a bigger challenge. There are many weighs connecting each input to the output, so it becomes more difficult. Each weight contributes to more than one output, so the learning algorithm must be more clever.

Training Neural Networks



The network output is denoted by $\hat{y}$ and we have

$$\hat{y} = f^2(W^2 f^1(W^2 \varphi(k) + w_0^1) + w_0^2)$$

- Denote the collection of all the weights and bias terms of all layers of the network as θ and θ has been initialized with random values. Denote f_{NN} as the overall function representing the Neural Network.
- The cost (loss) function is $V(\hat{y}, y)$ or $V(f_{NN}(\varphi, \theta), y)$.

Training Neural Networks

- Compute the gradient of this loss function and denote it by $\nabla V(f_{NN}(\varphi, \theta), y)$.
- Update θ using steepest descent, for example, as

$$\theta_s = \theta_{s-1} - \alpha V(f_{NN}(\varphi, \theta), y),$$

  where s denotes a single step

If the cost function is

$$V(\theta, Z^N) = \frac{1}{2N} \sum_{k=p}^{N} (y(k) - \hat{y}(k|k-1, \theta))^2 = \frac{1}{2N} \sum_{k=p}^{N} \varepsilon^2(k, \theta),$$

where $Z^N = \{y(k), \varphi(k); k = 1, \ldots, N\}$ represents a set of N samples of measurement input and output data of the system.

Training Neural Networks

The i^{th} element of the gradient $\nabla V(\theta)$ of the cost function is

$$\frac{\partial V(\theta)}{\partial \theta_i} = \frac{1}{2N} \sum_{k=1}^{N} \frac{\partial (y(k) - \hat{y}(k|\theta))^2}{\partial \theta_i} = -\frac{1}{N} \sum_{k=1}^{N} \frac{\partial \hat{y}(k|\theta)}{\partial \theta_i} \varepsilon(k|\theta)$$

where $\varepsilon(k|\theta)$ is the prediction error at time k.
Output Layer we have

$$\hat{y}(k|\theta) = f^2 \left(\sum_{j=0}^{s^1} w_{1j}^2 v_j^1(k) \right) = f^2(h^2(k))$$

Note that the summation begins at $j = 0$: by defining $v_0^1(k) = 1$ we can treat the bias term $w_1 0^2$ as an additional weight parameter. Applying the chain rule, we obtain

$$\frac{\partial \hat{y}(k|\theta)}{\partial w_{1j}^2} = \left. \frac{\partial f^2(h^2)}{\partial h^2} \right|_k v_j^1(k)$$

If we define the **sensitivity**

$$\delta_1^2(k) = \left. \frac{\partial f^2(h^2)}{\partial h^2} \right|_k$$

of the first (and in this case only) output of layer 2 at sampling instant k , we can write

$$\frac{\partial \hat{y}(k|\theta)}{\partial w_{1j}^2} = \delta_1^2(k) v_j^1(k)$$

Hidden Layer

The derivative of the predicted output with respect to the weight and bias parameters in the hidden layer can be obtained from

$$\hat{y}(k|\theta) = f^2 \left(\sigma_{j=1}^{s^1} w_{1j}^2 f^1 \left(\sum_{i=0}^{r} w_{ji}^1 \varphi_i(k) \right) + w_0^2 \right)$$

The index i points to input channels, while the index j points to neurons in the hidden layer. Note that the inner summation begins at $i = 0$; we define $\varphi_0(k) = 1$ so that the bias terms of the hidden layer can be treated as additional weights.

Training Neural Networks

Applying the chain rule yields

$$\frac{\partial \hat{y}(k|\theta)}{\partial w_{ji}^1} = \left.\frac{\partial f^2(h^2)}{\partial h^2}\right|_k \left.\frac{\partial h^2}{\partial h^1}\right|_k \left.\frac{\partial h^1}{\partial w_{ji}^1}\right|_k$$

Observing that

$$\left.\frac{\partial h^1}{\partial w_{ji}^1}\right|_k = \varphi_i(k) \qquad \text{and} \qquad \left.\frac{\partial h^2}{\partial h^1}\right|_k = w_{1j}^2 \left.\frac{\partial f^1(h^1)}{\partial h^1}\right|_k$$

this can be written as

$$\frac{\partial \hat{y}(k|\theta)}{\partial w_{ji}^1} = \delta_j^1(k)\varphi_i(k)$$

where we defined the sensitivity of the j^{th} output in the first layer at time k

$$\delta_j^1(k) = \delta_1^2(k) w_{1j}^2 \left.\frac{\partial f^1(h^1)}{\partial h^1}\right|_k$$

Training Neural Networks

- From the computation above, we see that the derivatives in each layer are given by the corresponding sensitivity multiplied with the input of the respective layer.
- Starting at the output layer, we can compute its sensitivity and obtain the corresponding derivative.
- Having computed the output sensitivity, we can calculate the sensitivity of the hidden layer; the derivatives with respect to weights in the layer.
- If we have a network with more than two layers, we can proceed in the same manner.
- The predicted output with respect to weights in each layer are thus obtained by **backpropagating** the sensitivities.

Training a Two-Layer Sig-Lin Perceptron Network

The update of the estimate of the i^{th} parameter is then

$$\theta_i(l+1) = \theta_i(l) - \alpha \frac{\partial V}{\partial \theta_i}\bigg|_l$$

The partial derivative of V, we see that wee need the weighted average of the partial derivatives of the predicted output over all sampling instants. The sensitivity of the linear output layer at sampling instant k is

$$\delta_1^2(k) = \frac{\partial f^2(h^2)}{\partial h^2}\bigg|_k = 1$$

and thus

$$\frac{\partial \hat{y}(k|\theta)}{\partial w_{1j}^2} = \delta_1^2(k) v_j^1(k) = v_j^1(k)$$

Training a Two-Layer Sig-Lin Perceptron Network

If we have log-sig activation functions

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

in the hidden layer, where for simplicity we let h denote the weighted sum of inputs h_j^1 at the j^{th} hidden neuron, we obtain

$$\frac{\partial f_j^(h)}{\partial h} = \frac{\partial}{\partial h}\frac{1}{1+e^{-h}} = \frac{e^{-h}}{(1+e^{-h})^2} = \left(1 - \frac{1}{1+e^{-h}}\right)\frac{1}{1+e^{-h}}$$

The sensitivity at the j^{th} hidden neuron is therefore

$$\delta_j^1(k) = w_{1j}^2(1 - v_j^1(k))v_j^1(k)$$

where w_{ij}^2 refers to elements of $W^2(l)$ obtained at the previous iteration step.

Training a Two-Layer Sig-Lin Perceptron Network

Updating the Weights

The weight matrices W^m (where $m = 1$) represents the hidden layer and $m = 2$ the output layer) are updated at iteration step l according to

$$W^m(l+1)m = W^m(l) - \alpha \Delta W^m(l)$$

for the output layer

$$\Delta W^2(l) = -\frac{1}{N} \sum_{k=1}^{N} (v^1)^T(k)\varepsilon(k)$$

Note that because $v_0^1 = 1$ the bias update is equal to the negative average of prediction errors

$$\Delta w_0^2(l) = -\frac{1}{N} \sum_{k=1}^{N} \varepsilon(k)$$

Training a Two-Layer Sig-Lin Perceptron Network

Then we obtain for the hidden layer update

$$\Delta W^1(l) = -\frac{1}{N}\sum_{k=1}^{N}\delta^1(k)\varphi^T(k)\varepsilon(k)$$

and the bias update

$$\Delta w_0^1(l) = -\frac{1}{N}\sum_{k=1}^{N}\delta^1(k)\varepsilon(k)$$

Deriving Cost Functions using Maximum Likelihood

We will look into how loss function are derive using Maximum Likelihood:

- Binary cross entropy
- Cross entropy
- Squared error

Here a basic idea:

- Given a data set $Z^N = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ where $x \in \mathbb{R}^n$ and $y \in \{0, 1\}$, which is the target of interest. (can be binary or continuous value)
- The idea behind Maximum Likelihood is to find a θ that maximizes $P(Z^N|\theta)$

Binary Cross Entropy

Assuming a Bernoulli distribution and given that each of the examples $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ are independent, we have the following expression:

$$P(Z^N|\theta) = \prod_{i=1}^{n} f(x_i, \theta)^{y_i} (1 - f(x_i, \theta))^{(1-y_i)}$$

We can take a logarithm operation on both sides to arrive at the following:

$$\log P(Z^N|\theta) = \log \prod_{i=1}^{n} f(x_i, \theta)^{y_i} (1 - f(x_i, \theta))^{(1-y_i)}$$

$$\log P(Z^N|\theta) = \sum_{i=1}^{n} (y_i \log f(x_i, \theta) + (1 - y_i) \log(1 - f(x_i, \theta)))$$

Then the minimum is

$$-\log P(Z^N|\theta) = -\left(\sum_{i=1}^{n} (y_i \log f(x_i, \theta) + (1 - f(x_i, \theta))) \right)$$

We can use it for the context of binary classification.

Cross Entropy

The cross entropy loss function is used in the context of multi-classification.

- Let us assume that $y \in \{0, 1, \ldots, k\}$, which are the classes.
- Denote $n_1, n_2, \cdots, n_k$ to be the observed counts of each of the k classes. Observe that $\sum_{i=1}^{k} n_i = n$
- Let us have generated a model that predicts the probability of y given x.
- The model denoted by $f(x, \theta)$, where θ represents the parameters of the model.
- Using the idea of Maximum Likelihood, we find θ that maximizes $P(Z|\theta)$
- Assuming a Multinomial distribution and given that each of the examples $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ are independent, we have the following expression:

$$P(Z^N|\theta) = \frac{n!}{n_1!n_2!\cdots n_k!} \prod_{i=1}^{n} f(x_i, \theta)^{y_i}$$

Cross Entropy

Take a logarithm operation on both sides to arrive at the following:

$$\log P(Z^N|\theta) = \log n! - \log\left(n_1!n_2!\cdots n_3!\right) + \sum_{i=1}^{n} y_i \log f(x_i, \theta)$$

The terms $\log n!$ and $\log n_1!n_2!\cdots n_k!$ are not parameterized by θ and can be ignored as we try to find a θ that maximizes $P(Z^N|\theta)$. Thus we have the following

$$\log P(Z^N|\theta) = \sum_{i=1}^{n} y_i \log f(x_i, \theta)$$

or

$$-\log P(Z^N|\theta) = -\sum_{i=1}^{n} y_i \log f(x_i, \theta)$$

Summary of LossFunctions

- The binary cross entropy given by the expression

$$-\sum_{i=1}^{n}(y_i \log f(x,\theta) + (1-y_i)\log(1-f(x_i,\theta)))$$

  is recommended loss function for binary classification. This loss function should typically be used when the Neural Network is designed to predict the probability of the outcome. In such cases, the output layer has a single unit with a suitable sigmoid as the activation function.
- The Cross entropy function given by the expression

$$-\sum_{i=1}^{n} y_i \log f(x_i,\theta)$$

  is the recommended loss function for multi-classification. This loss function should typically be used with the Neural Network and is designed to predict the probability of the outcomes of each of the class.
- The squared loss function given by $\sum_{i=1}^{n}(y-\hat{y})^2$ should used for regression problems. The output layer in this case will have a single unit.

Python

We need to install:

- Latest Anaconda with Python 3.6
- We need to install the packages by using `conda install` (with admin authorize)
  - `Scikit-learn`
  - `Theano`
  - `Autograd`
  - `Keras`
  - `PyOpenCL`
- Goto `Control Panel`, `System and Security`, `System`
- `Advanced system setting`, `Environment Variable`
- add new system variable name
- `MKL_THREADING_LAYER` and the variable value `GNU`
- restart the system.

A 2-Layer Neural Network for Regression

Necessary Library

```
import autograd.numpy as np
import autograd.numpy.random as npr
from autograd import grad

import sklearn.metrics
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
```

Generate Dataset Z^N, $N = 1000$ where $x, y \in \mathbb{R}^{100}$

```
examples = 1000
features = 100
Z = (npr.randn(examples, features), npr.randn(examples))
```

A 2-Layer Neural Network for Regression

Specify the network: 2 layers the layer 1 has 10 nodes and the layer 2 has 1 node
$W_1 \in \mathbb{R}^{100\times 10}$, $W_2 \in \mathbb{R}^{10\times 1}$

```
layer1_units = 10
layer2_units = 1
w1 = npr.rand(features, layer1_units)   #initial random weight
b1 = npr.rand(layer1_units)                #bias of layer 1
w2 = npr.rand(layer1_units, layer2_units)
b2 = 0.0

theta = (w1, b1, w2, b2)
```

Define the loss function with square error

```
def squared_loss(y, y_hat):
    return np.dot((y - y_hat), (y - y_hat))
```

Define the loss function with binary cross entropy

```
def binary_cross_entropy(y, y_hat):
    return np.sum(-((y * np.log(y_hat)) + ((1-y) * np.log(1 - y_hat))))
```

A 2-Layer Neural Network for Regression

Wrapper around the Neural Network

```
def neural_network(x, theta):
    w1, b1, w2, b2 = theta
    return np.than(np.dot((np.tanh(np.dot(x, w1) + b1)), w2) + b1)
```

Wrapper around the objective function to be optimised

```
def objective(theta, idx):
    return squared_loss(Z[1][idx], neural_network(Z[0][idx], theta))
```

A 2-Layer Neural Network for Regression

Update the Network

```
def update_theta(theta, delta, alpha):
    w1, b1, w2, b2 = theta
    w1_delta, b1_delta, w2_delta, b2_delta = delta
    w1_new = w1 - alpha * w1_delta
    b1_new = b1 - alpha * b1_delta
    w2_new = w2 - alpha * w2_delta
    b2_new = b2 - alpha * b2_delta
    new_theta = (w1_new, b1_new, w2_new, b2_new)

    return new_theta
```

Compute Gradient

```
grad_objective = grad(objective)
```

A 2-Layer Neural Network for Regression

Train the Neural Network

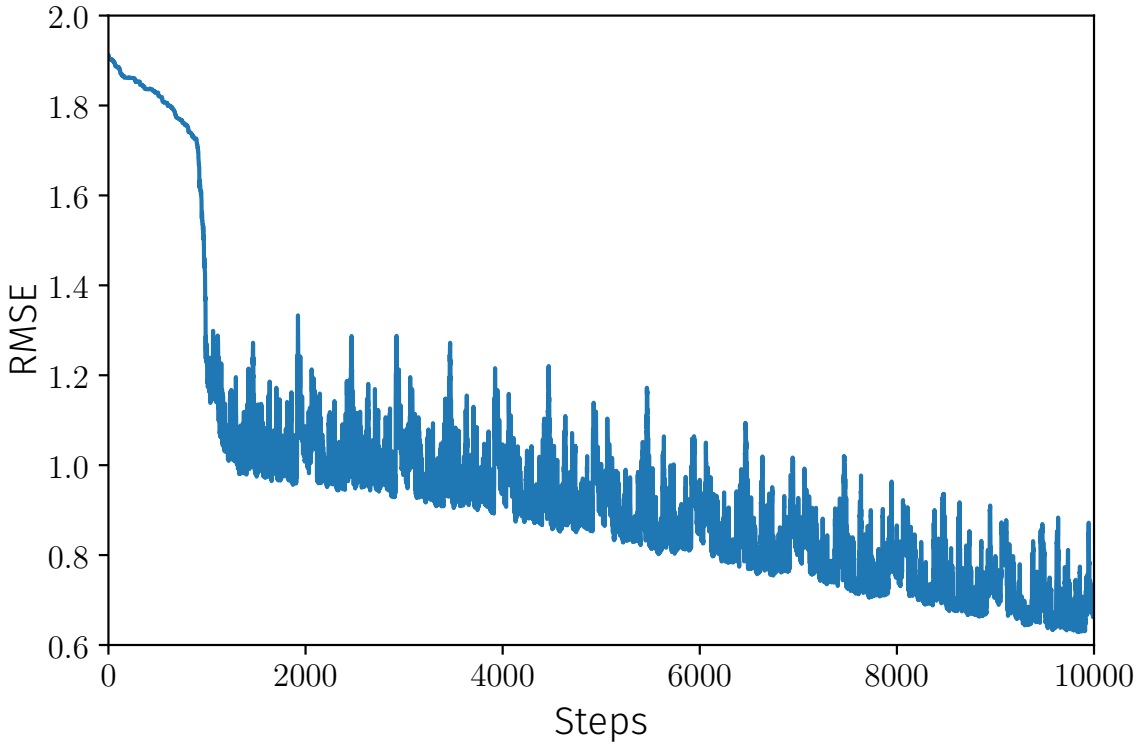
```
epochs = 10
print("RMSE before training:", sklearn.metrics.mean_squared_error(Z[1],
                                    neural_network(Z[0], theta))
rmse = []
for i in xrange(0, epochs):
    for j in xrange(0, examples):
        delta = grad_objective(theta, j)
        theta = update_theta(theta, delta, 0.01)
        rmse.append(sklearn.metrics.mean_squared_error(Z[1], neural_network(Z[0], theta)))

print("RMSE after training:", sklearn.metrics.mean_squared_error(Z[1],
                                    neural_network(Z[0], theta))
plt.plot(rmse)
```

One of the result can be

```
RMSE before training: 1.990198216168253
RMSE after training: 0.7514662947319568
```

A 2-Layer Neural Network for Regression

RMSE over training steps



Reference

1. Aurélien, Géron, "*Hands-On Machine Learning with Scikit-Learn & TensorFlow*", O'reilly, 2017
2. S.P.K. Spielberg, R. B. Gopaluni, P. D. Loewen, "Deep Reinforcement Learning Approaches for Process Control", 2017 6th International Symposium on Advanced control of Industrial Processes (AdCONIP), May 28-31, 2017, Taipei, Taiwan
3. Rudolf Kruse, Christian Borgelt, Christian Braune, Sanaz Mostaghim and Matthias Steinbrecher, "Computational Intellignece: A Methodological Introduction" 2nd , 2016
4. Sckit-Learn website `http://scikit-learn.org/stable/index.html`
5. Josh Patterson and Adam Gibson,"Deep Learning: A practitioner's approach", 2016
6. Nikhil Ketkar, "Deep Learning with Python: A Hands-on Introduction", 2017