Deep Learning

Introduction to Neural Networks
w/ Neurolab Lib

Dom Huh

Libraries to download

pip install neurolab

Documentation:

https://pythonhosted.org/neurolab/

https://pythonhosted.org/neurolab/lib.html

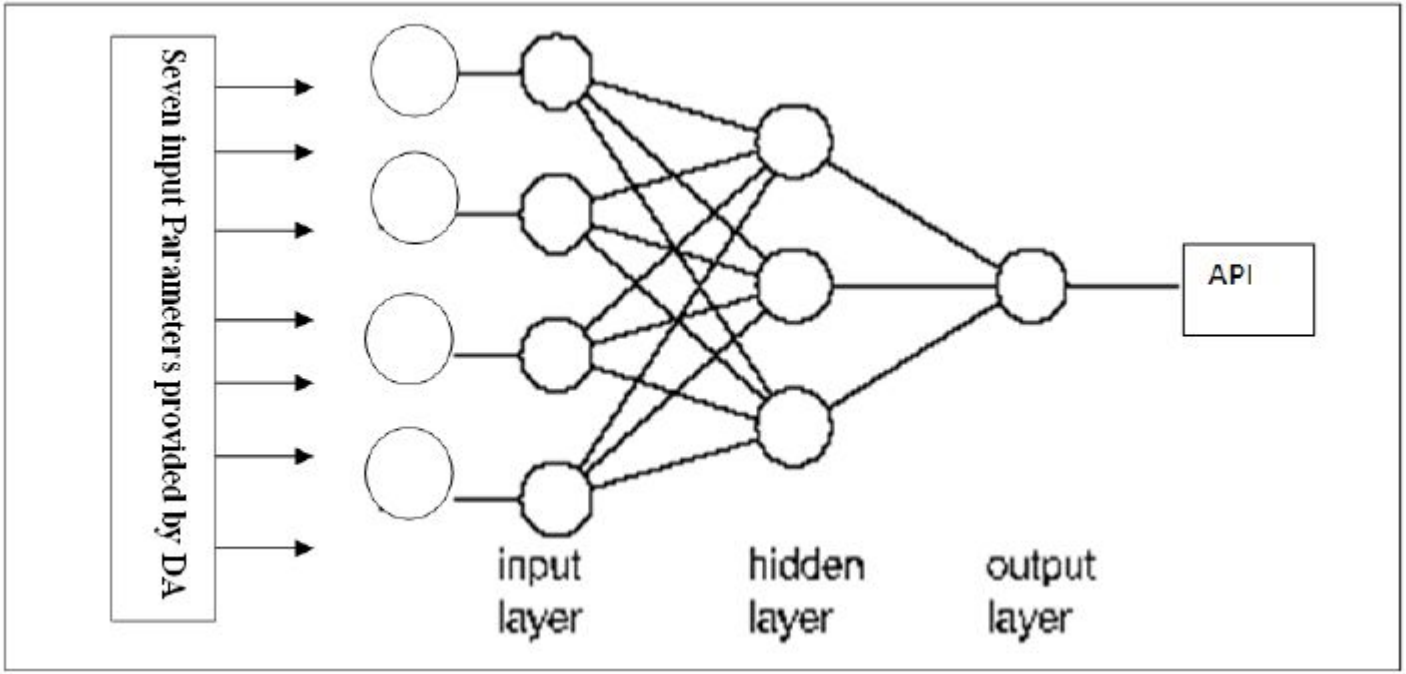
New divisions we are focusing on

Supervised - maps an input to an output based on example input-output pairs. It infers a function from labeled training data consisting of a set of training examples

Unsupervised - draw inferences from datasets consisting of input data without explicit labels, classification or category

Reinforced - take actions in an environment so as to maximize some notion of cumulative reward with discount factor with consideration of value and action-value

Structure of neural networks

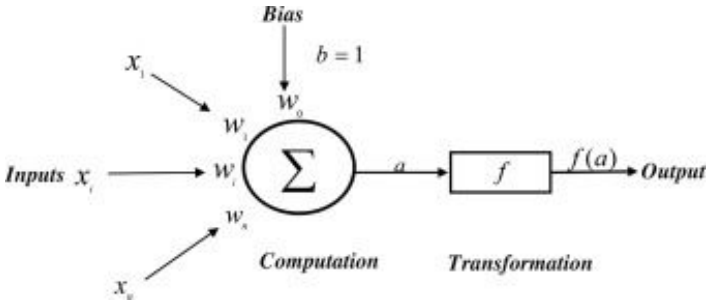


Perceptrons aka artificial neurons

Receives input

Adds weight and bias (aka does computation on it)

Produces an output



If we only did this, what would the output look like?

What else should we do? Why would this lead to potentially a terrible model?

Things to consider beforehand

Data has to numerical

Labeled data set to numerical data to NN.

Nonlinearity

Input-Output Mapping

Adaptability

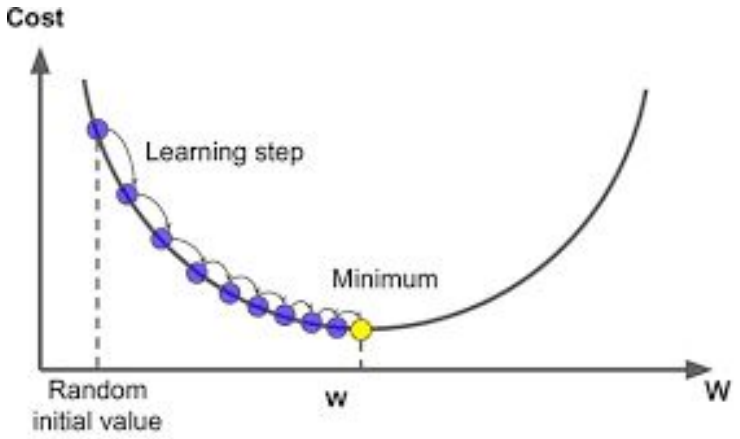
Evidential Response

Fault Tolerances

Goal: Minimize error from actual values from data set, aka lower difference between predicted outputs and actual values. How?

Cost Function + Backpropagation

Use an algorithm to calculate cost and adjust the weights of the “neurons” based on the error of the outputs.



Cost functions to Backpropagation

Cost function acts to calculate the difference in expected vs observed output.

$$C(\theta) = \frac{1}{n}\sum_{i=1}^{n} L(f_\theta, z_i)$$

Training loss
OR
Generalization loss

$$C(\theta) = \int L(f_\theta, z) P(z) dz$$

Gradient is calculated with cost function

$$\frac{\partial C(\theta)}{\partial \theta} = \lim_{\delta\theta \to 0} \frac{C(\theta + \delta\theta) - C(\theta)}{\delta\theta}$$

Gradient Descent wants to minimize this value

$$\frac{\partial C(\theta)}{\partial \theta} = 0$$

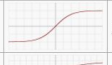
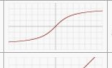
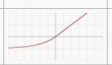
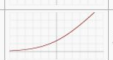
With either batch gradient descent

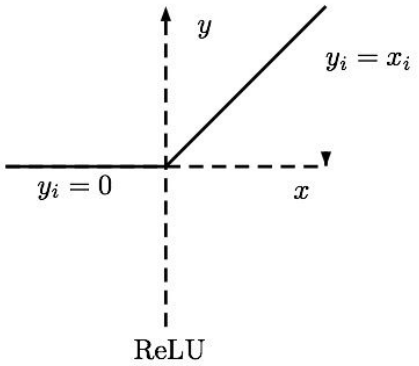
$$\theta^{k+1} = \theta^k - \epsilon_k \frac{\partial C(\theta^k)}{\partial \theta^k}$$

OR
With Adam:
https://arxiv.org/pdf/1412.6980.pdf

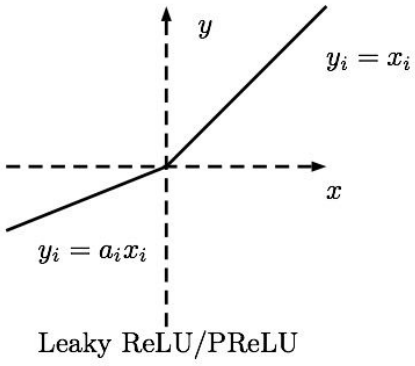
Activation Function aka Squashing

When we do not have the activation function, the weights and bias would simply do a linear transformation.

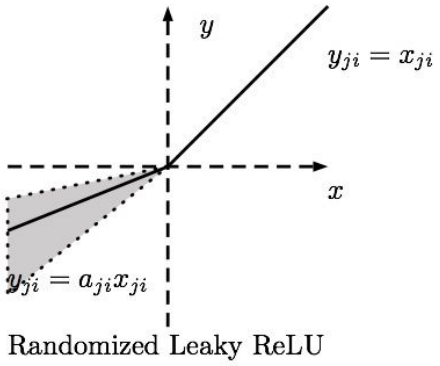
Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU)[2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU)[3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$



ReLU



Leaky ReLU/PReLU



Randomized Leaky ReLU

Overview of Propagation

Input layer:

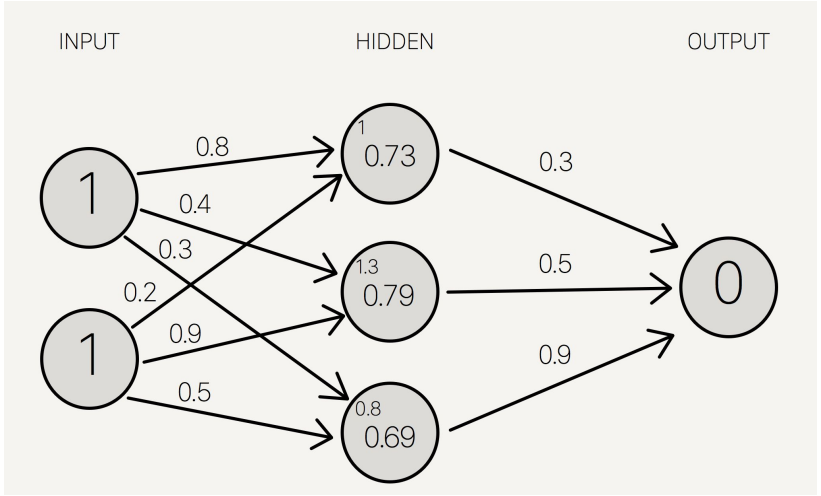
Data is split into a fixed number of nodes

Each node contains some sort of numerical representation of the data

Hidden Layer:

Data is passed through a fixed number of layers of nodes with various weights and biases (Forward propagation)

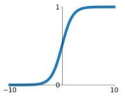
Activation Functions (ReLU)



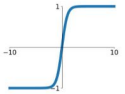
$$Y = \text{Activation}(\Sigma(weight * input) + bias)$$

Activation Functions

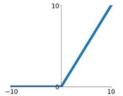
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



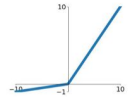
tanh
$\tanh(x)$



ReLU
$\max(0, x)$

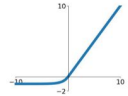


Leaky ReLU
$\max(0.1x, x)$



Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$



Overview of Propagation

Output Layer:

Activation function

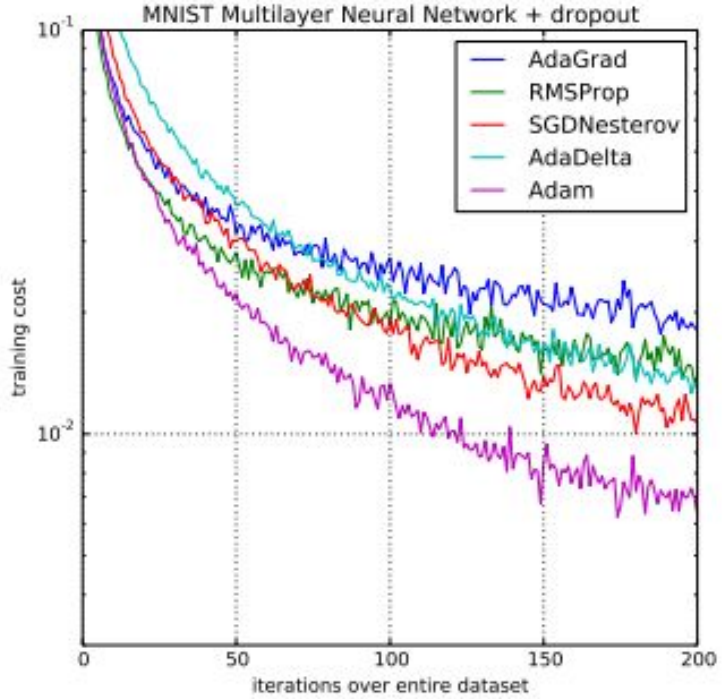
Cost function

Outputs prediction

Backpropagation - Update the weights and biases of the neurons on the basis of the error.

Hidden layer:

Optimization



Single Layer Network

Train with Delta rule aka hill-climbing gradient descent.

Involves using derivative of the perceptrons' weights with respect to the output error to adjust the weights to better classify training examples.

For better mathematical intuition: https://blog.yani.io/deltarule/

$$\Delta w_{ji} = \alpha(t_j - y_j)g'(h_j)x_i$$

α is a small constant called *learning rate*

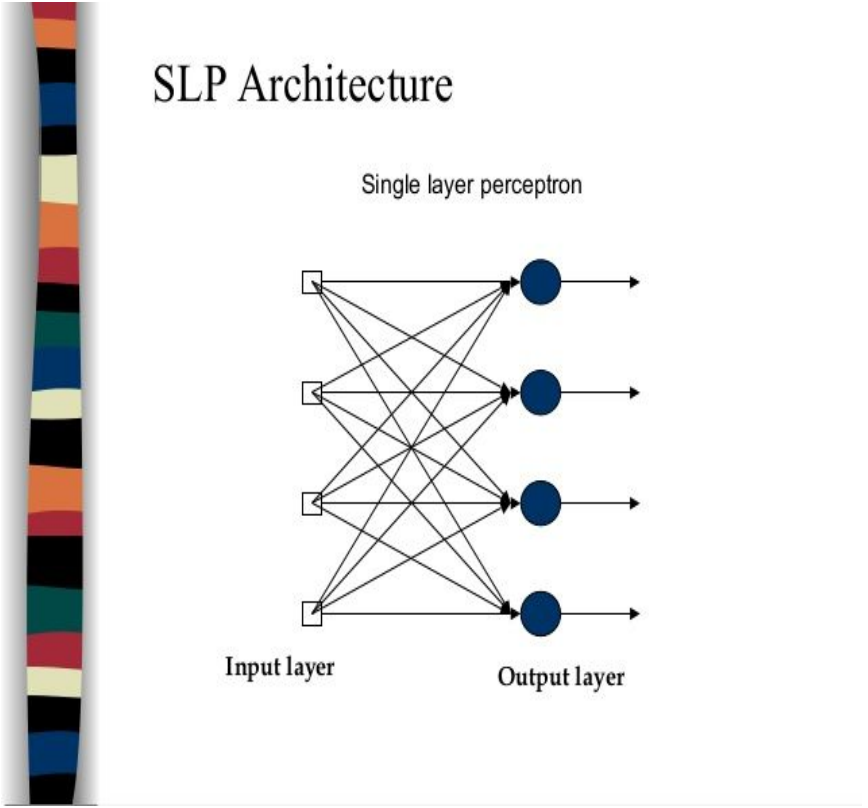
$g(x)$ is the neuron's activation function

t_j is the target output

h_j is the weighted sum of the neuron's inputs

y_j is the actual output

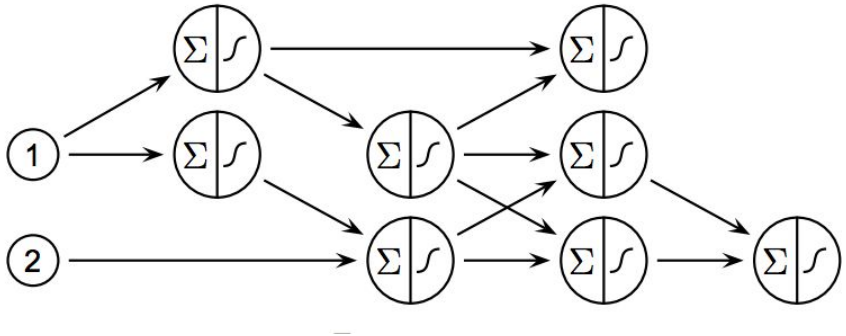
x_i is the ith input.



SLN implementation with NeuroLabs

Multilayer feed forward perceptron

A finite directed acyclic graph



More in-depth look into MLP:

http://ml.informatik.uni-freiburg.de/former/_media/teaching/ss10/05_mlps.printer.pdf

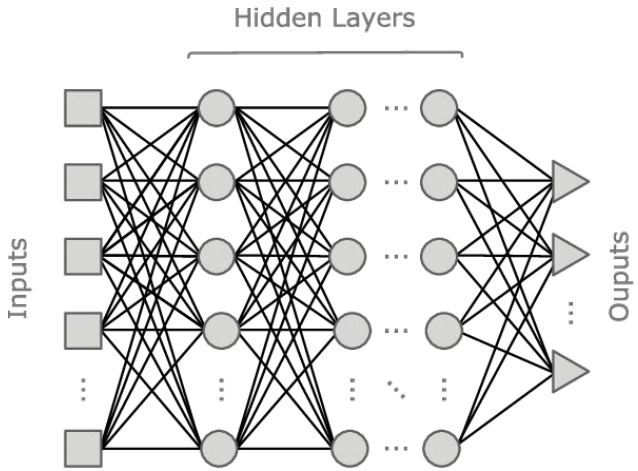
Multilayer feed forward perceptron

Optimizations:

Broyden, Fletcher, Goldfarb, Shanno (BFGS) method

Newton-CG method

Conjugate gradient algorithm



Vector Quantization (VQ) and Learning Vector Quantization (LVQ)

Applies simple competitive learning to solve supervised learning tasks where class membership is known for each training pattern. Each node is associated with a predetermined class label and the number of nodes chosen for each class is roughly proportional to the number of training patterns in that class.

Essentially, this is what is used in KNN, K-means

LVQ1 train function

LVQ2 train function

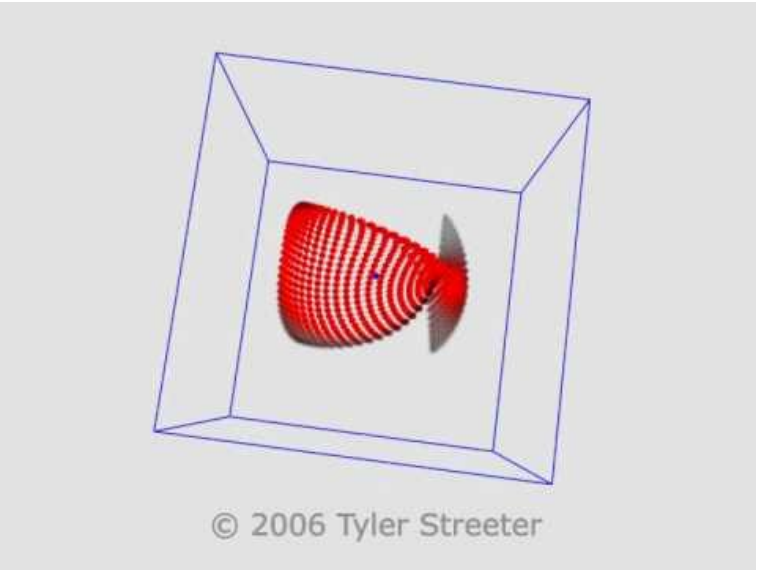
SOM (Self Organizing Map)

Unsupervised algorithm for dimension reduction on high dimensioned dataset, transforming into lower dimensional discrete representation of the input space of the training samples, called a map.

Remember K-Means, but in 2-D.

SOM therefore preserves the topology of the original data, because the distances in 2-D space reflect those in the high-dimensional space.

More details at:
http://www.pitt.edu/~is2470pb/Spring05/FinalProjects/Group1a/tutorial/som.html



Competing layer (Kohonen Layer)

Connections to SOMs

Nodes compete for the right to respond to a subset of the input data

A set of neurons that are all the same except for some randomly distributed synaptic weights, and which therefore respond differently to a given set of input patterns

A limit imposed on the "strength" of each neuron

A mechanism that permits the neurons to compete for the right to respond to a given subset of inputs, such that only one output neuron is active at a time. The neuron that wins the competition is called a "winner-take-all" neuron.

Winner Take All algorithm

- Can be adjusted to K-Winner Take All

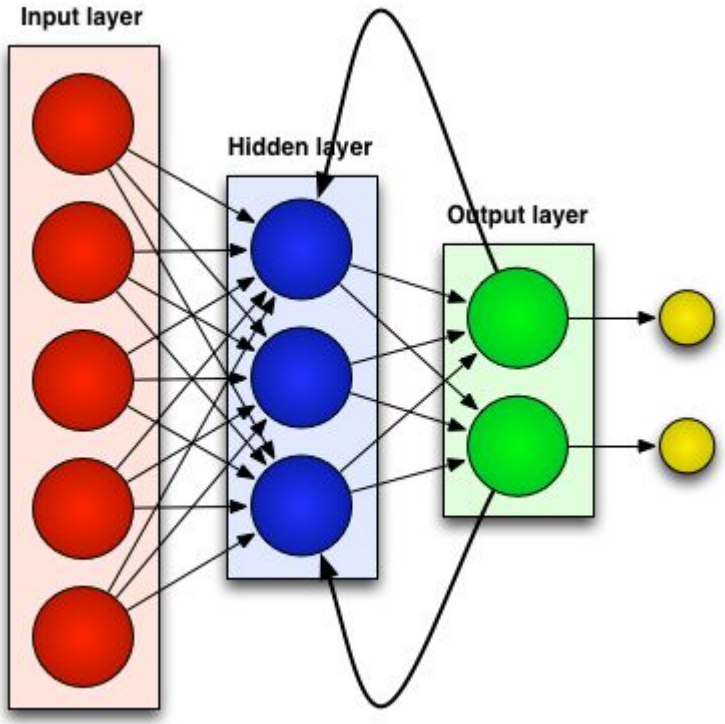
Competitive Learning (cont.)

More in-depth look into Competitive Learning

https://pdfs.semanticscholar.org/7f13/a0c932e32eb0dbe009dc86badfe8bed31e66.pdf

Recurrent Networks

Includes connections between nodes to form a directed graph along a sequence. This allows it to exhibit temporal dynamic behavior for a time sequence. Therefore, some sort of memory is implemented. (ie. LSTM, GRU)

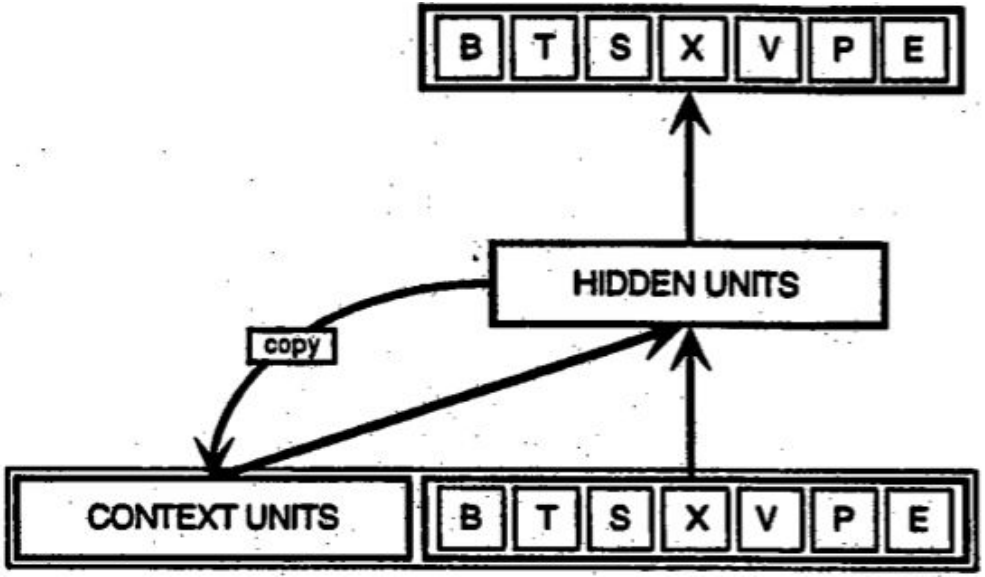


Simple recurrent networks (SRNs)

Diverges from MLP to have unit(s) of memory called Context

Types:

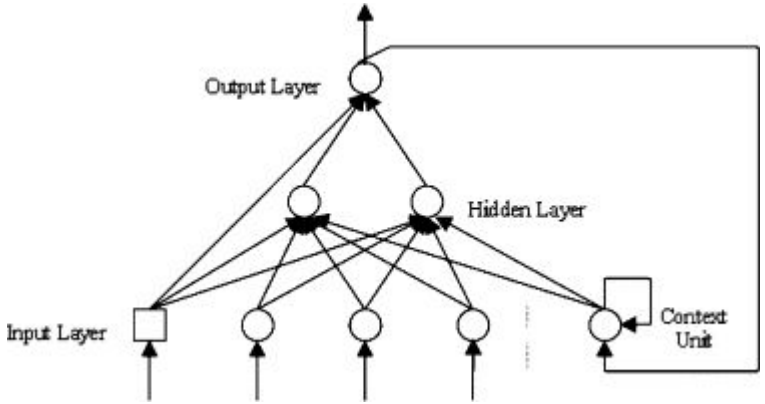
- Elman
- Jordan



Jordan Recurrent Networks

As the input is fed-forward and a learning rule is applied, the fixed back-connections save a copy of the previous values of the hidden units in the context units.

Note: The context units are fed from the output layer.

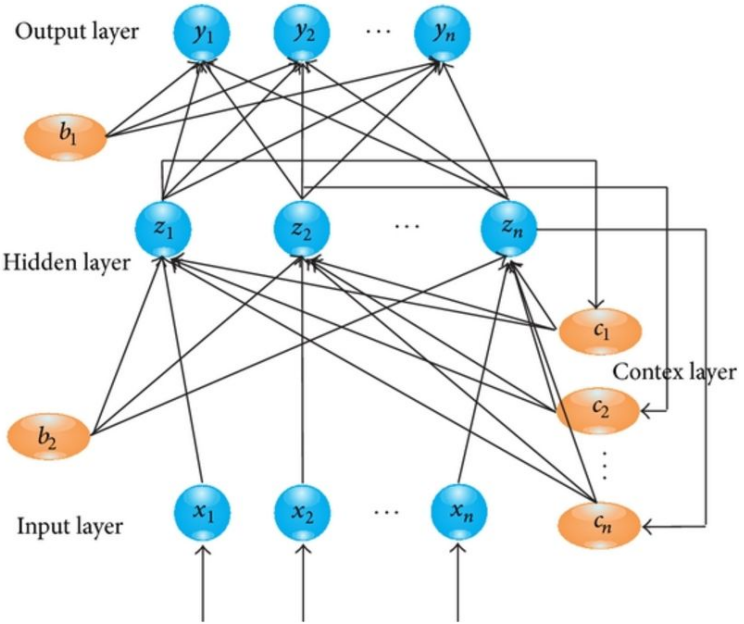


$$h_t = \sigma_h(W_h x_t + U_h y_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$

Elman Recurrent network

Same as JRN, but the context units are fed from the hidden layers.

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$



Hebbian networks

Implements associative learning

Hebbian theory is a neuroscientific theory claiming that an increase in synaptic efficacy arises from a presynaptic cell's repeated and persistent stimulation of a postsynaptic cell. It is an attempt to explain synaptic plasticity, the adaptation of brain neurons during the learning process("Cells that fire together wire together.").

Essentially, Hebbian learning reinforces connections from learned states

Hopfield Recurrent network

Maximally connected network with **symmetric** weights(binary thresholded) that fires randomly selected neurons and update the neurons every selection.

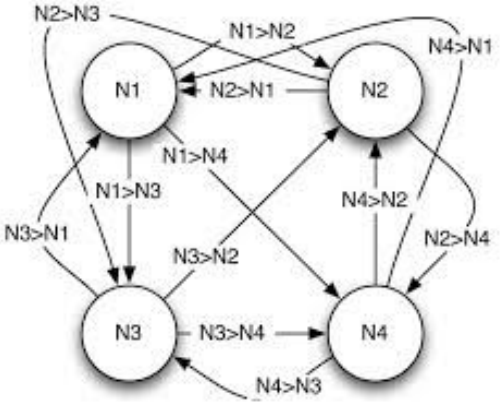
On: 1 Off: -1

Asynchronous vs Synchronous Firing

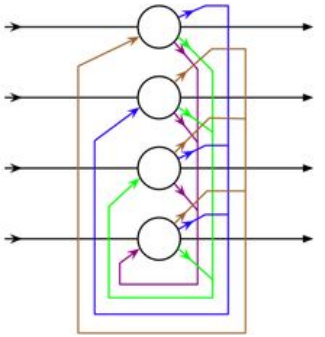
Hopfield Energy

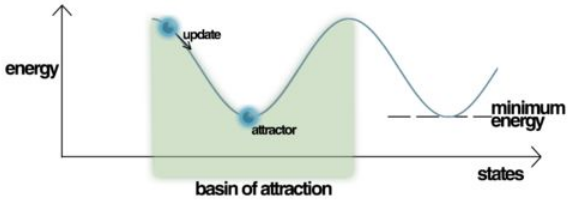
Issues with Hopfield

Spurious States (locals)



$$E = -\frac{1}{2}\sum_i \sum_j w_{ij} x_i x_j + \sum_i \theta_i x_i$$



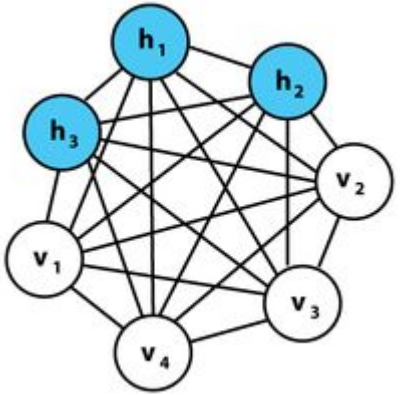


http://faculty.etsu.edu/knisleyj/neural/neuralnet3.htm

Boltzmann Network

Essentially a Monte Carlo algorithm for Hopfield, therefore incorporates “energy”, but is stochastic and generative

Search, represent, and learning



$$E = -\left(\sum_{i<j} w_{ij}\, s_i\, s_j + \sum_i \theta_i\, s_i\right)$$

Restricted Boltzmann machine

Deep Boltzmann machine

Next time: Backpropagation

Resilient Gradient Descent

Stochastic Gradient Descent

Gradient descent with momentum

Gradient descent with adaptive learning rate

Learning Decay

Nesterov Momentum

AdaDelta

AdaGrad

Adam

Hemming Recurrent network