ECBM E4040
Neural Networks and Deep Learning

Deep Feedforward Networks

Zoran Kostić
Columbia University
Electrical Engineering Department &
Data Sciences Institute

Outline

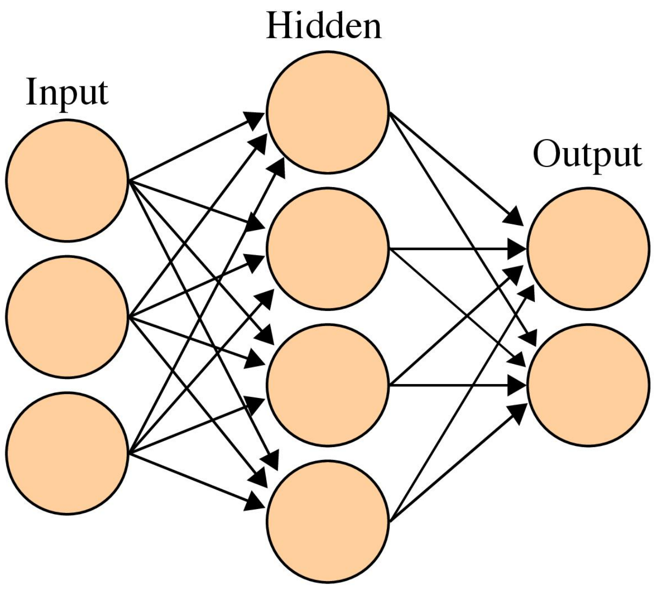
- Multilayer perceptrons (MLPs)
- Connections, weights and activation functions
- Stochastic transformations and reparametrization trick
- Universal approximation theorem
- Deep models
- ReLU activation

References and Acknowledgments

- **Deep Learning by Ian Goodfellow, Yoshua Bengio and Aaron Courville, http://www.deeplearningbook.org/ , chapter 6.**
- **Lecture material by bionet group / Prof. Aurel Lazar (http://www.bionet.ee.columbia.edu/).**

Deep Feedforward Networks or Multilayer Perceptrons

hidden layers



Multilayer Perceptrons (MLPs)
are multi-input and multi-output
parametric functions,
defined by composing together many parametric functions.

1980's.

diagram from Colah's blog:
http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Deep Forward Network (MLP)

The goal:

Approximate function f^*

How:

MLP defines a mapping

$$y = f(x; \theta)$$

and learns the value of parameters

$$\theta$$

that result in the best approximation of function f^*

Deep Networks, Training

A function is represented by composing together many different functions.

$$f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$$

Each component function is represented by a separate layer.

During ANN training, we drive

$$f(\boldsymbol{x})$$

to match

$$f^*(\boldsymbol{x})$$

Each example $\boldsymbol{x}$ is accompanied by a label $y \approx f^*(\boldsymbol{x})$, which tells what the network should do to get as close as possible to y.

Deep Networks - Many Layers

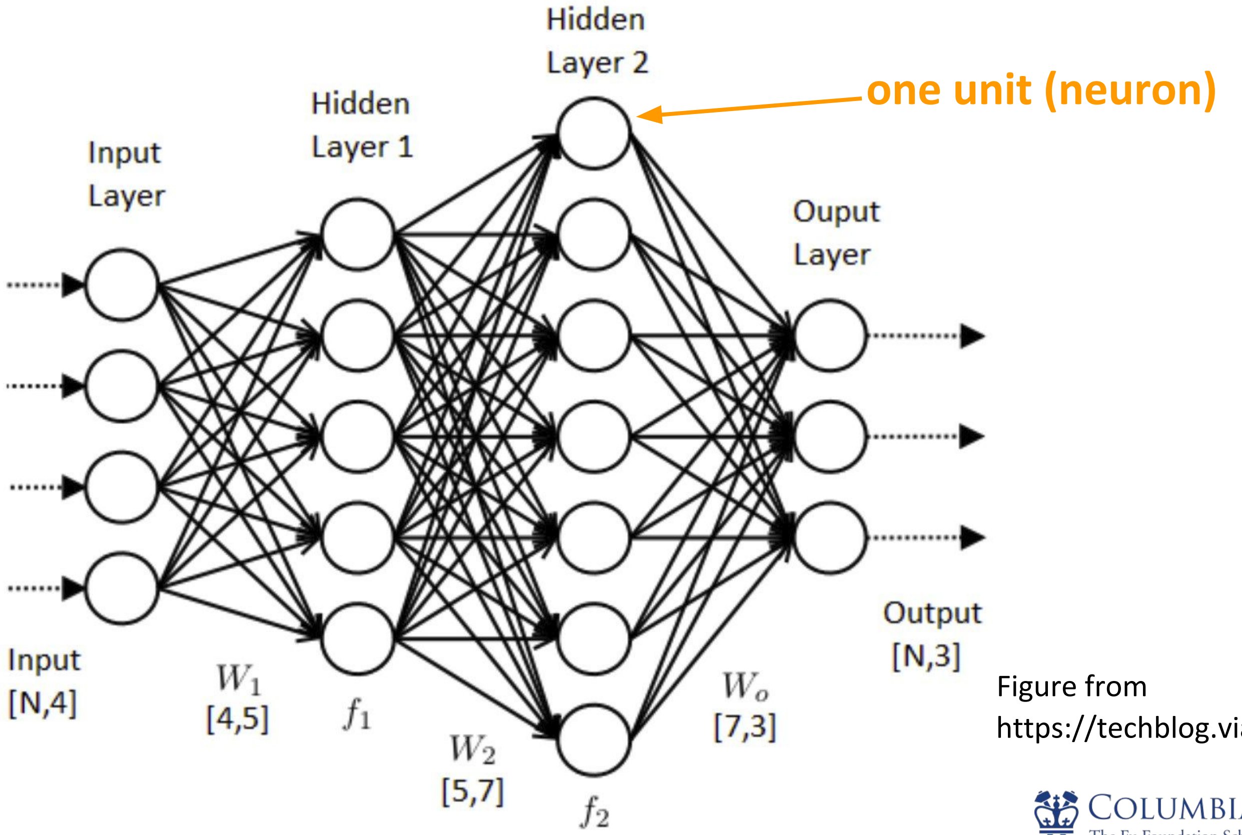
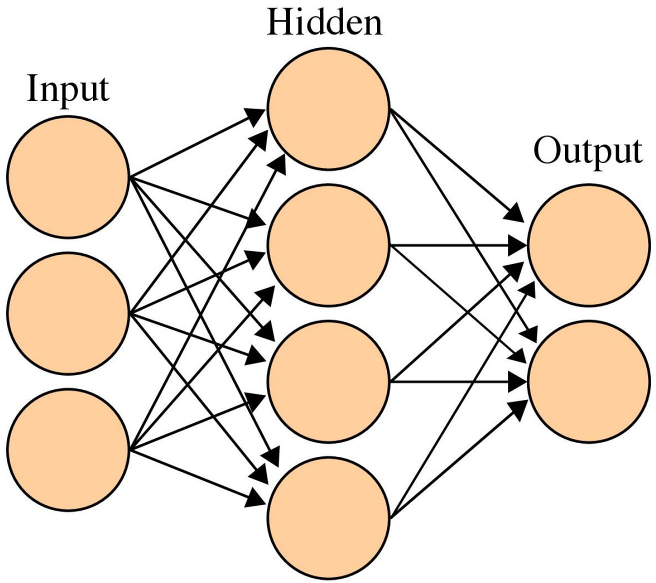


Figure from https://techblog.viasat.com

Deep Feedforward Networks or Multilayer Perceptrons

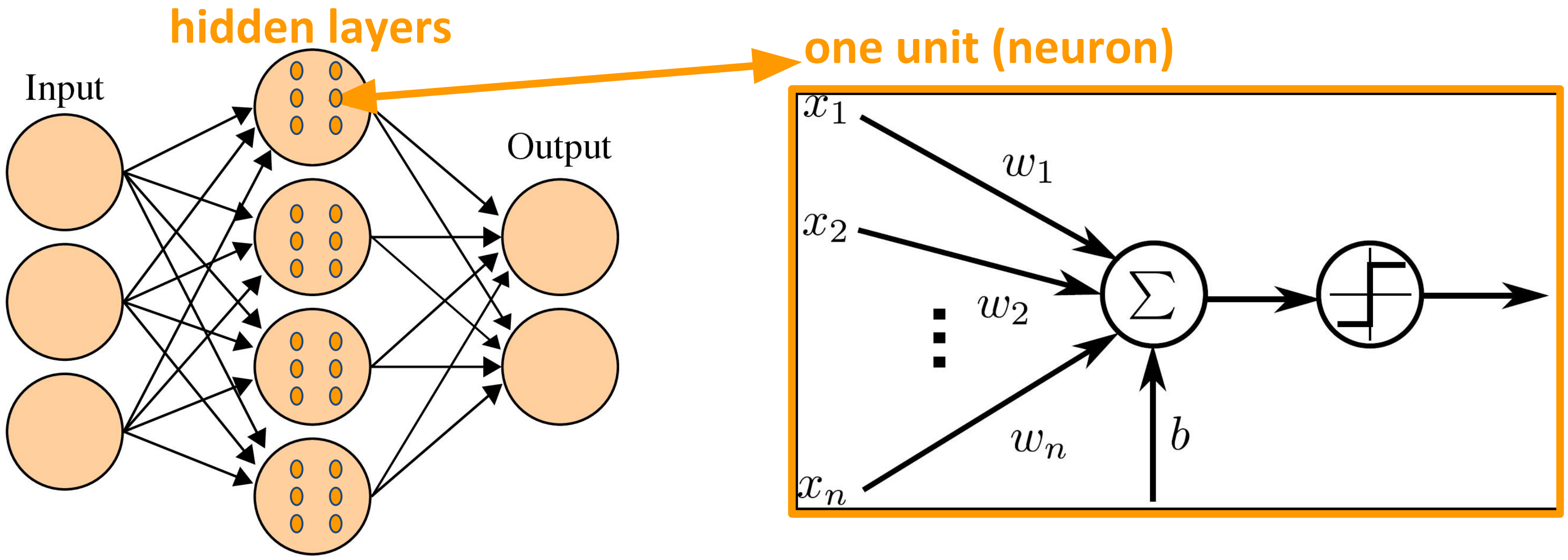
hidden layers



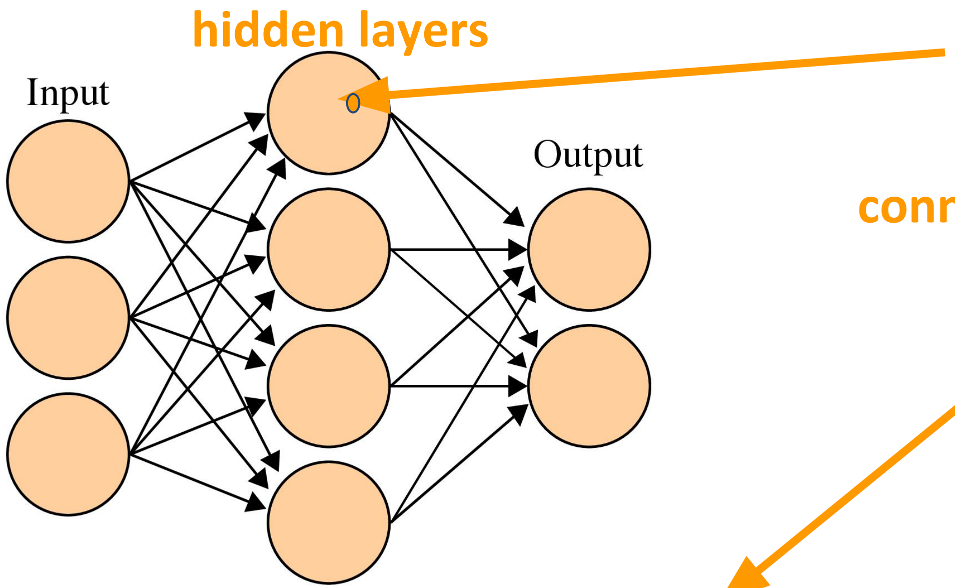
Each sub-function is known as a layer of the network, and each scalar layer output of one of these functions as a unit or feature.

The number of units in each layer is called the width of a machine learning model, and the number of layers is called the depth.

Deep Feedforward Networks or Multilayer Perceptrons



Deep Feedforward Networks or Multilayer Perceptrons



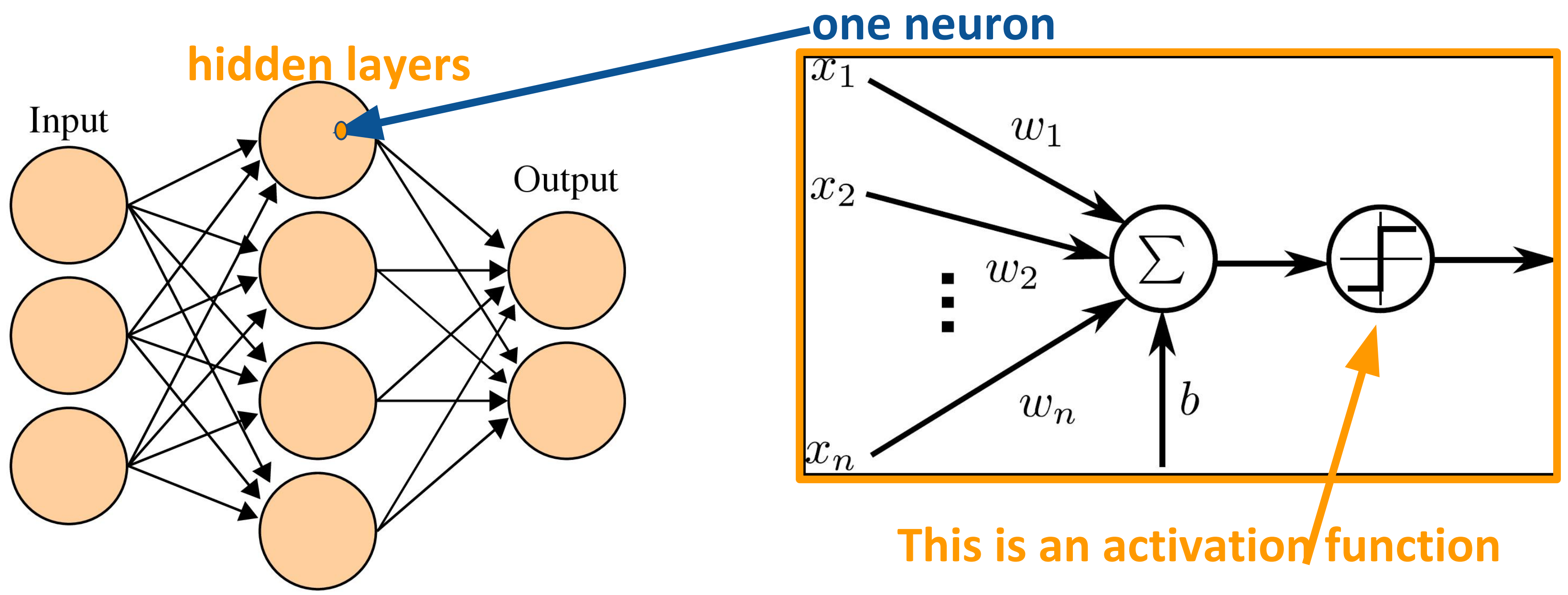
one hidden unit (neuron)

connectivity of ANN and the output

$$\boldsymbol{x} \in \mathbb{R}^{n_i} \qquad \mathbf{h} \in \mathbb{R}^{n_h} \qquad \hat{\mathbf{y}} \in \mathbb{R}^{n_o}$$

$$\mathbf{W} \in \mathbb{R}^{n_h \times n_i} \qquad \mathbf{V} \in \mathbb{R}^{n_o \times n_h}$$

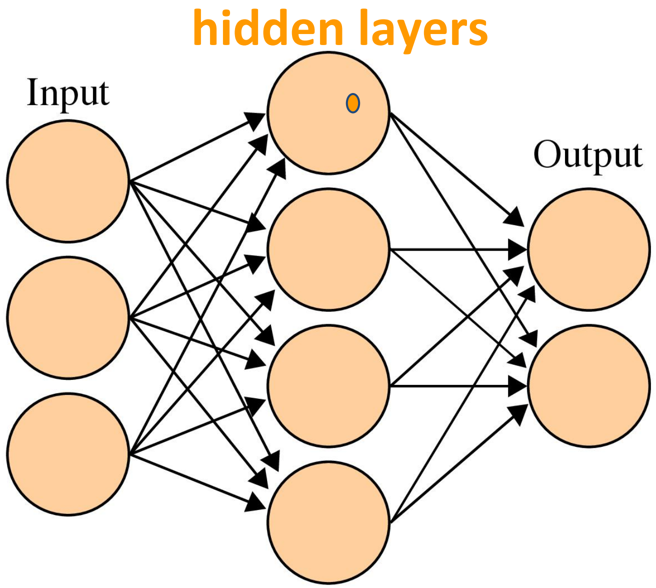
Deep Feedforward Networks or Multilayer Perceptrons



This is an activation function

Deep Feedforward Networks or Multilayer Perceptrons

sigmoid is one possible activation function



$\boldsymbol{x} \in \mathbb{R}^{n_i}$ $\mathbf{h} \in \mathbb{R}^{n_h}$ $\hat{\mathbf{y}} \in \mathbb{R}^{n_o}$

$\mathbf{W} \in \mathbb{R}^{n_h \times n_i}$ $\mathbf{V} \in \mathbb{R}^{n_o \times n_h}$

The hidden unit vector is $\mathbf{h}$

$$\mathbf{h} = \text{sigmoid}(\mathbf{c} + \mathbf{W}\boldsymbol{x})$$

with weight matrix $\mathbf{W}$ and offset vector $\mathbf{c} \in \mathbb{R}^{n_h}$.

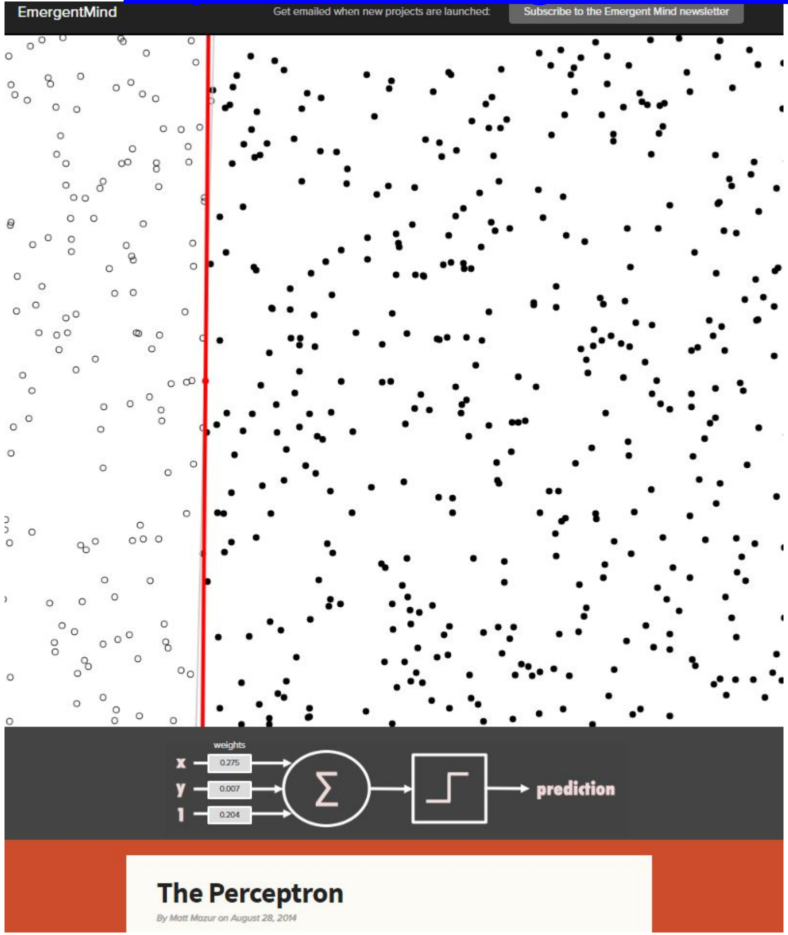
The output vector is obtained via the learned affine transformation

$$\hat{\mathbf{y}} = \mathbf{b} + \mathbf{V}\mathbf{h},$$

with weight matrix $\mathbf{V}$ and output offset vector $\mathbf{b} \in \mathbb{R}^{n_0}$.

Perceptron

http://www.emergentmind.com/the-perceptron


EmergentMind
Get emailed when new projects are launched:
Subscribe to the Emergent Mind newsletter
weights
x
y
1
0.275
0.007
0.204
Σ
prediction

The Perceptron

By Matt Mazur on August 28, 2014

Multilayer Perceptron
What Can it Do

Regression

Classification

Shallow Multilayer Perceptron - Regression

MLP model has many coefficients which can be changed

- **in an iterative fashion,**
- **the goal being to get the best possible representation of a function.**

How to change the coefficients?

- **By comparing the output (obtained by the ANN) against the target value (label),**
- **i.e. by minimizing the error/loss function.**

Shallow Multilayer Perceptron - Regression Function Representation and Loss Definition

We consider the family of input-output functions

$$\mathbf{f}(\boldsymbol{x};\boldsymbol{\theta}) = \mathbf{b} + \mathbf{V}\text{sigmoid}(\mathbf{c} + \mathbf{W}\boldsymbol{x}),$$

where $\text{sigmoid}(a) = 1/(1+e^{-a})$ is applied element-wise. The input vector is $\boldsymbol{x} \in \mathbb{R}^{n_i}$, and the output vector is $\mathbf{f} \in \mathbb{R}^{n_o}$. The parameters are the flattened vectorized version of the tuple

$$\boldsymbol{\theta} = (\mathbf{b}, \mathbf{c}, \mathbf{V}, \mathbf{W}).$$

complete description of an ANN

For the loss function

$$L(\hat{\mathbf{y}}, \mathbf{y}) = ||\hat{\mathbf{y}} - \mathbf{y}||^2$$

one can show (using calculus of variations) that $\hat{\mathbf{y}} = \mathbb{E}[\mathbf{y}|\boldsymbol{x}]$.

Shallow Multilayer Perceptron - Regression
Cost Minimize->IterativeWeightUpdate/Training

Define the weights $\boldsymbol{\omega}$ as the concatenated elements of matrices $\mathbf{W}$ and $\mathbf{V}$ and $||\boldsymbol{\omega}||^2 = \sum_{i,j} W_{ij}^2 + \sum_{ki} V_{ki}^2$. The regularized cost function typically considered

$$J(\boldsymbol{\theta}) = \lambda||\boldsymbol{\omega}||^2 + \frac{1}{n}\sum_{t=1}^{n} ||\mathbf{y}^t - (\mathbf{b} + \mathbf{V}\text{sigmoid}(\mathbf{c} + \mathbf{W}\boldsymbol{x}^t))||^2,$$

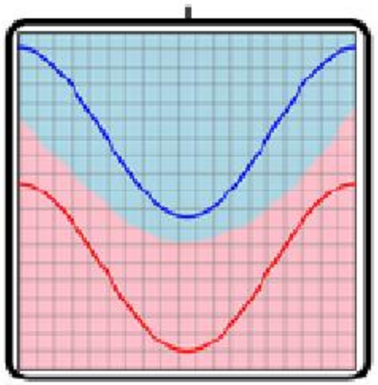
where $(\boldsymbol{x}^t, \mathbf{y}^t)$ is the t-th training (input, target) pair.
The classical training procedure iteratively updates $\boldsymbol{\theta}$ as in

$$\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \epsilon(2\lambda\boldsymbol{\omega} + \nabla_{\boldsymbol{\omega}} L(\mathbf{f}(\boldsymbol{x}^t; \boldsymbol{\theta}), \mathbf{y}^t))$$
$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \epsilon\nabla_{\boldsymbol{\beta}} L(\mathbf{f}(\boldsymbol{x}^t; \boldsymbol{\theta}), \mathbf{y}^t),$$

where $\boldsymbol{\beta} = (\mathbf{b}, \mathbf{c})$ contains the offset parameters, ϵ is the learning rate and t is incremented after each training example, modulo n.

Multilayer Perceptron - Classification

Can a line* be the class decision (separator) in this case?



input
(2)

The red and blue solid curves are where the training examples come from.

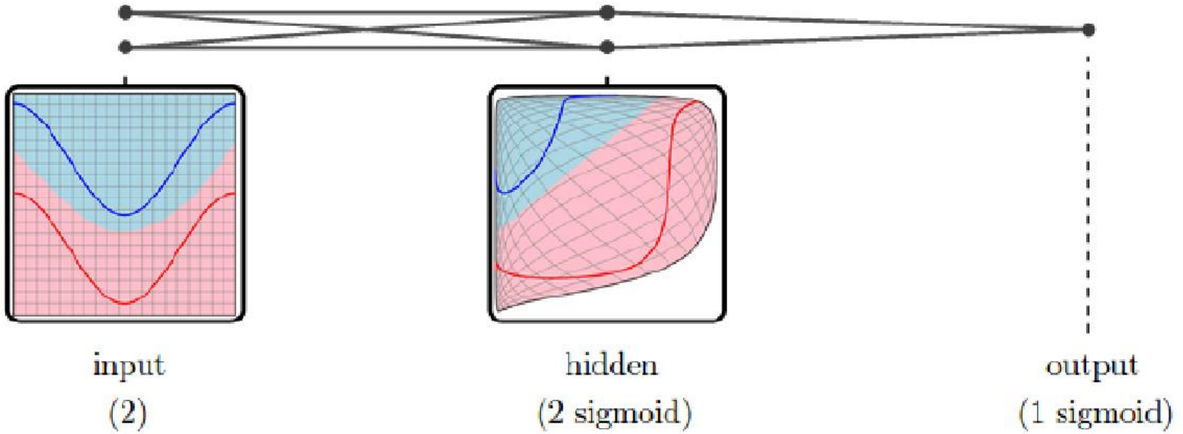
Non-linear boundary.

* a line in 2D space corresponds to a linear surface in multidimensional space

Multilayer Perceptron - Classification

Strategy: do not try to "bend the decision surface", but transform the data (by MLP).

When data is properly transformed non-linearly by the first hidden layer, the good decision surface becomes a linear one, which can be implemented by the MLP output.



<- 2-layer MLP

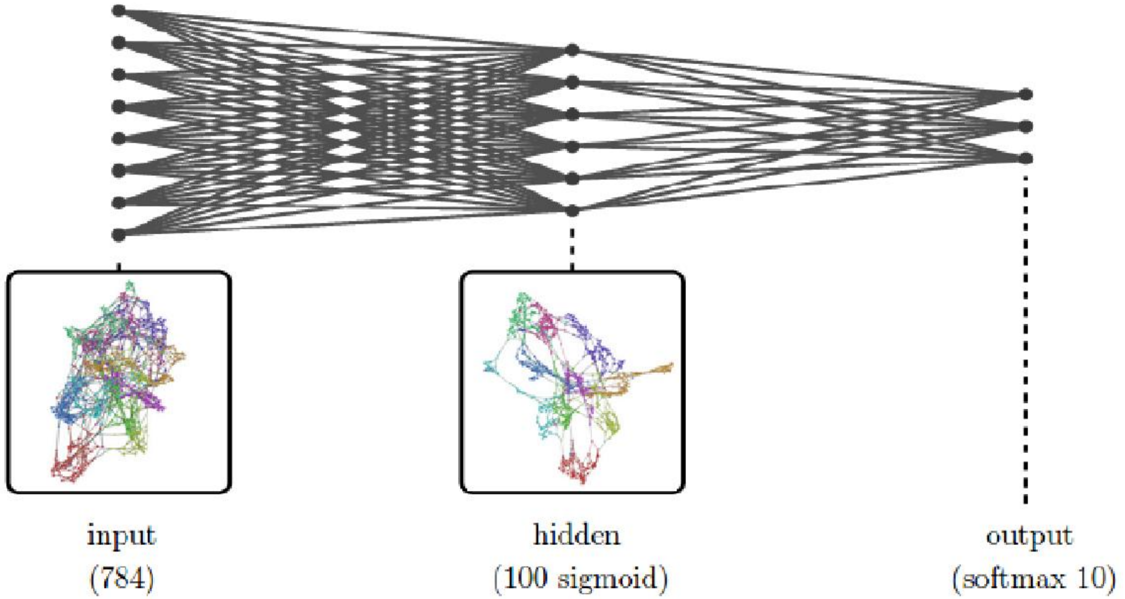
Multilayer Perceptron - Classification
Effect of Non-Linear Transformations

Intuition via Dimensionality Reduction

784 inputs (pixels of 28 x 28 image of a digit), 100 hidden units,

10 outputs

= 10 digit classes.



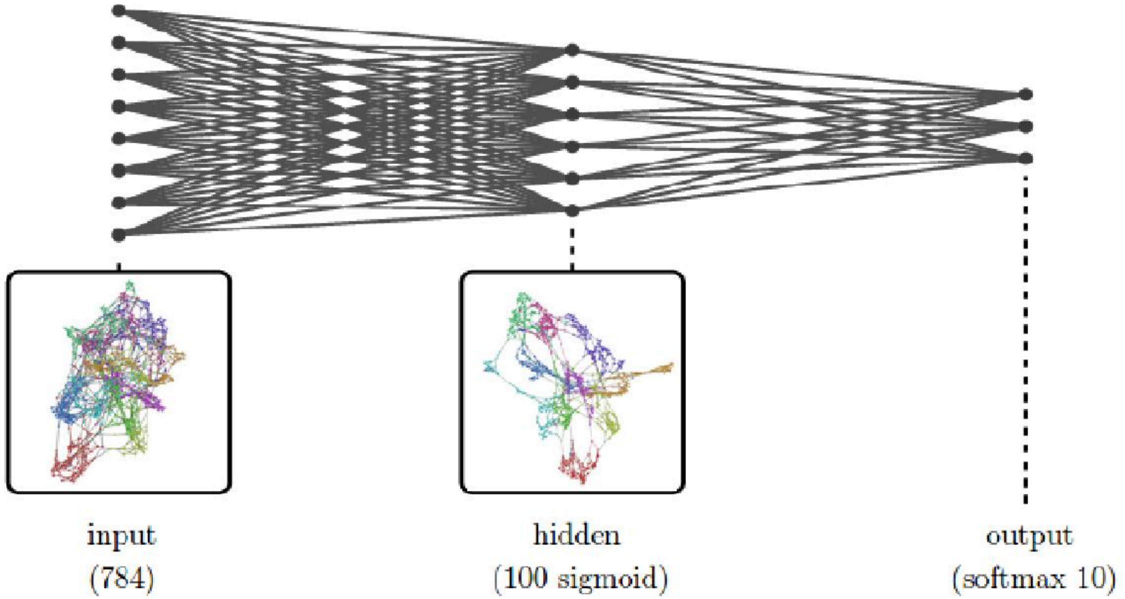
from: colah's blog

http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Multilayer Perceptron - Classification
Effect of Non-Linear Transformations

Intuition via Dimensionality Reduction

Windows below input and hidden layers show where the training examples are in the reduced space, with one point per example, colored per digit class.

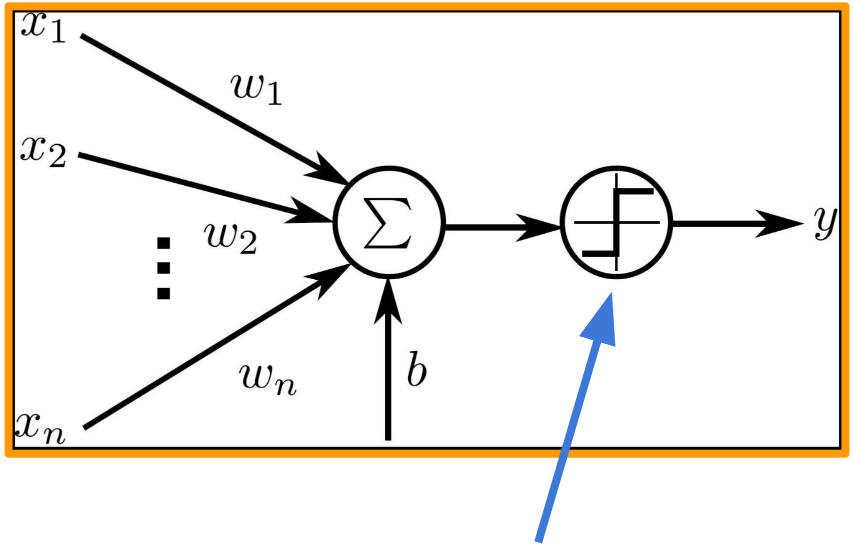


linear boundary surfaces

Activation Functions Enable Simple Transformations -> Complex Nonlinear Transformations

Every neuron imposes a “simple” transformation on its input data.

Successive layers of neurons create highly complex nonlinear transformations



This is an activation function

(Non-Linear) Activation Functions

Combination of

- **an affine transform**

$$\mathbf{a} = \mathbf{b} + \mathbf{W}\boldsymbol{x}$$

- **and an activation function**

$$\mathbf{h} = \phi(\mathbf{a})$$

Output (final layer) activation function is usually a nonlinearity different from other layers

- **depends on the type of output to be predicted, and the associated loss function**

$$\mathbf{h} = \phi(\mathbf{a}) \Leftrightarrow h_i = \phi(a_i) = \phi(b_i + \mathbf{W}_{i}.\boldsymbol{x})$$

Activation Functions - Biologically Inspired

Biologically inspired neural networks: the activation function is an abstraction representing the rate of action potential firing in the cell.

Sigmoid activation function: stays at zero until input current is received, at which point the firing frequency increases quickly at first, but gradually approaches an asymptote at 100% firing rate.

Hyperbolic tangent function: used in multilayer perceptrons.

Activation Functions - Inventory

Biologically inspired:

- **Binary**
- **Sigmoid**
- **Hyperbolic tangent**

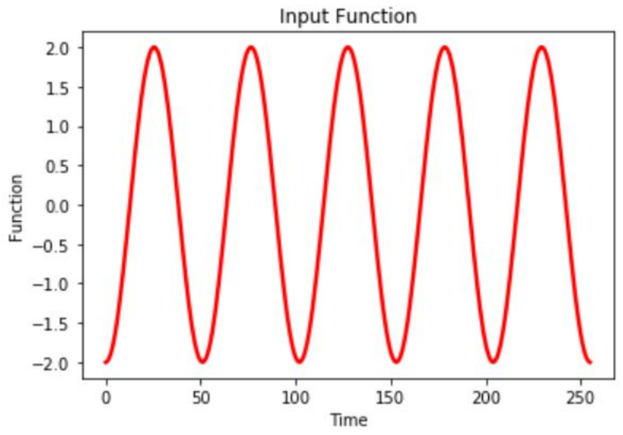
Plus others used in ANNs:

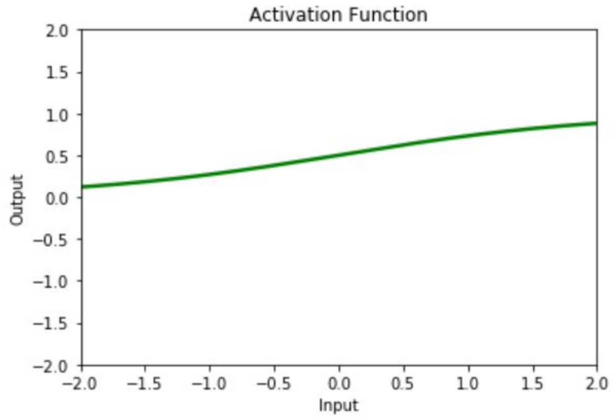
- **Rectified Linear Unit (ReLU)**
- **Leaky ReLU**
- **Softmax**
- **Radial Basis Function (RBF)**
- **Hard tanh**
- **Absolute Value Rectification**
- **Maxout**
- **…**

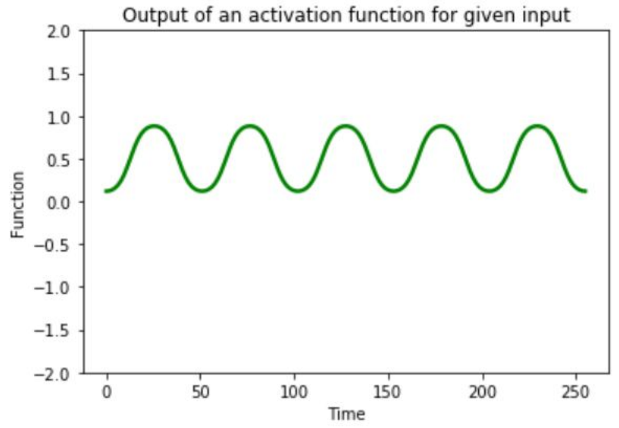
wikipedia inventory with properties

Activation Function
Sigmoid

$$\text{sigmoid}(a) = 1/(1 + e^{-a})$$

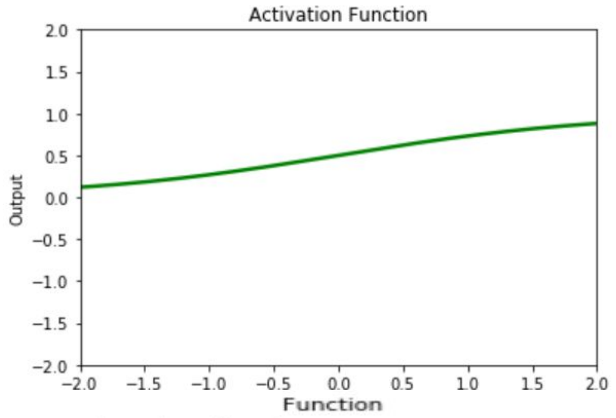


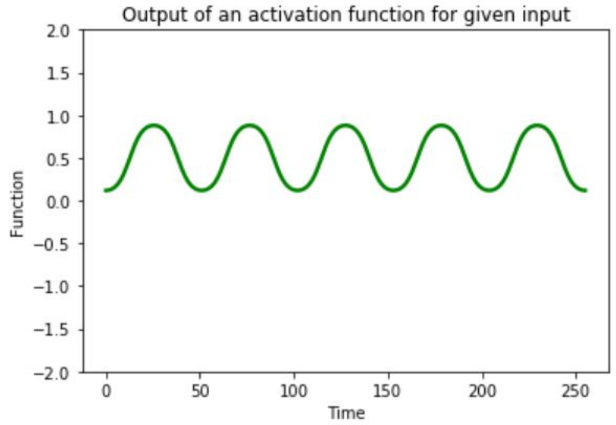


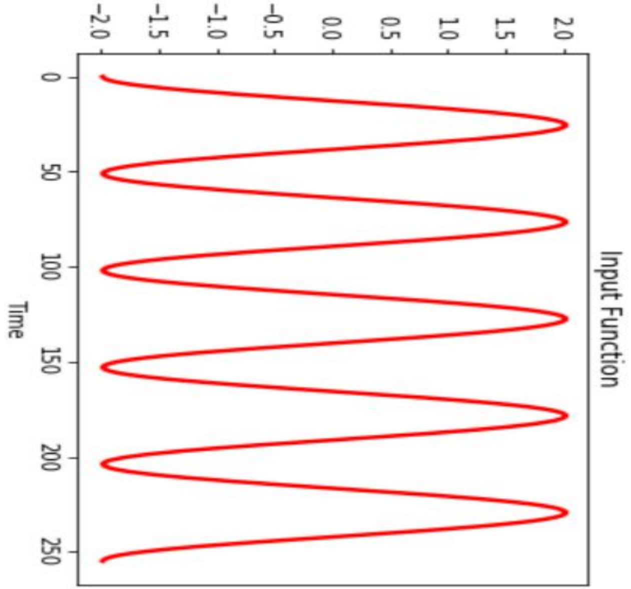


Activation Function Sigmoid

$$\text{sigmoid}(a) = 1/(1+e^{-a})$$



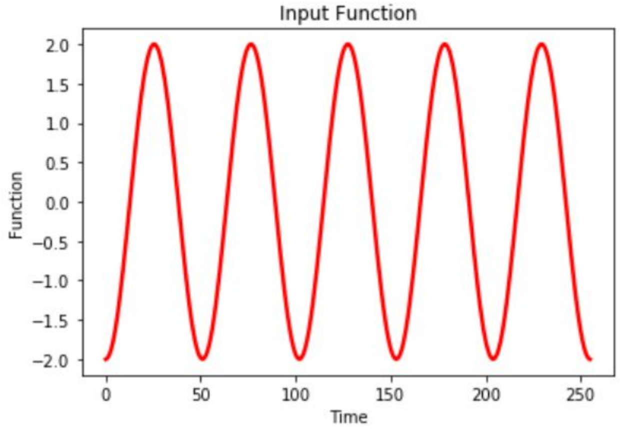


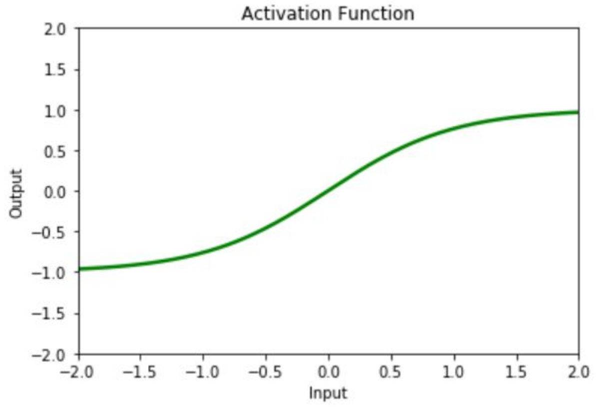


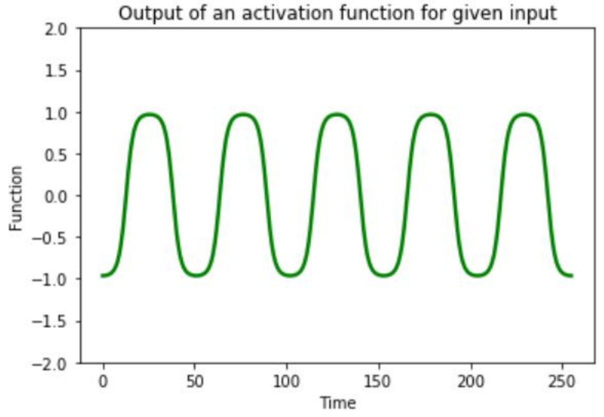
Activation Function
Hyperbolic Tangent

$$\mathbf{h}^k = \tanh(\mathbf{b}^k + \mathbf{W}^k \mathbf{h}^{k-1}),$$

where $\mathbf{h}^0 = \boldsymbol{x}$ is the input of the neural net, $\mathbf{h}^k, k > 0$ is the output of the k-th hidden layer, which has weight matrix $\mathbf{W}^k$ and offset (or bias) vector $\mathbf{b}^k$.

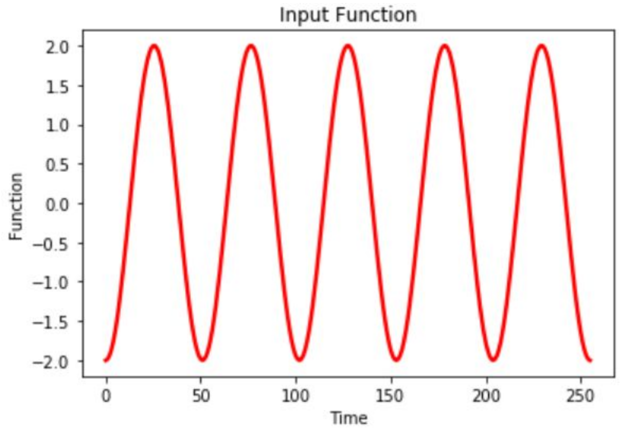


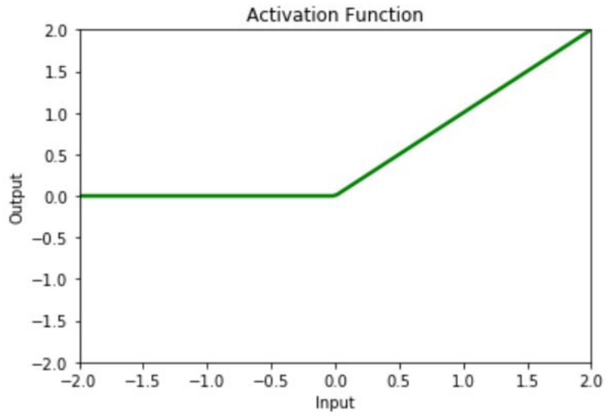


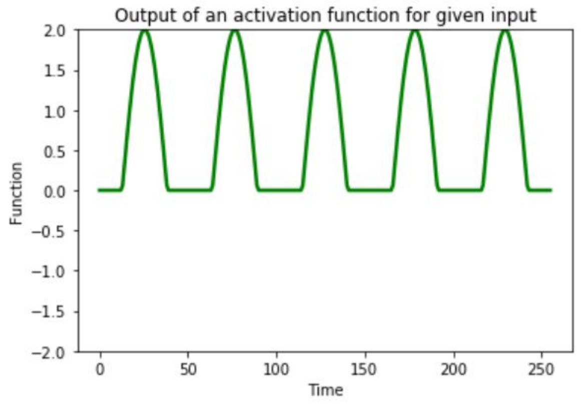


Activation Function
Rectified Linear Unit (ReLU) or Positive Part

$\phi(\mathbf{a}) = \max(0, \mathbf{a})$, also written $\phi(\mathbf{a}) = \mathbf{a}^{+}$.

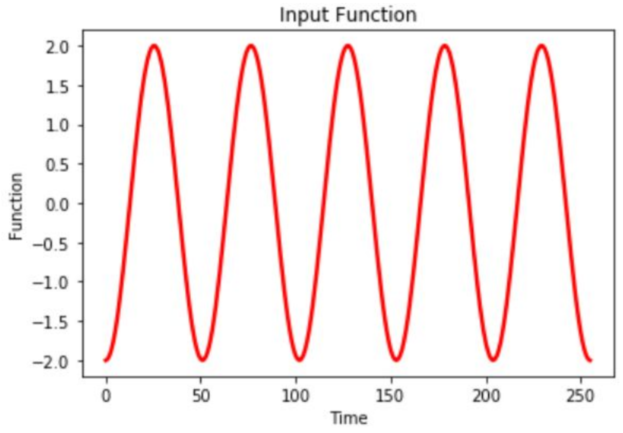


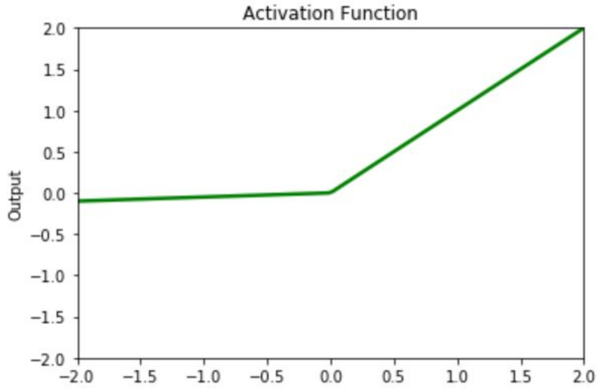


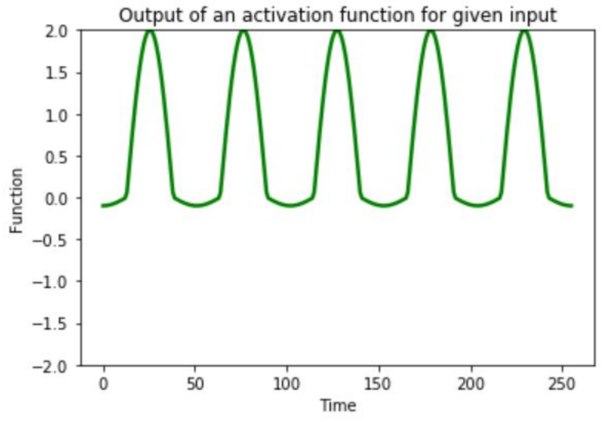


Activation Function
Leaky Rectified Linear Unit (ReLU)

$$f(x) = \begin{cases} 0.01x & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$$







Activation Function
Softmax

$$\phi(\mathbf{a}) = \text{softmax}(\mathbf{a}) \Leftrightarrow [\phi(\mathbf{a})]_i = e^{a_i} / \sum_j e^{a_j}$$

such that $\sum_i [\phi(\mathbf{a})]_i = 1$ and $[\phi(\mathbf{a})]_i > 0$

Activation Function
Radial Basis Function (RBF)

Radial Basis Function (RBF) unit: acts on $\boldsymbol{x}$ using $\mathbf{h}_i = \exp(-||\mathbf{w}_i - \boldsymbol{x}||^2/\sigma_i^2)$ that corresponds to template matching. Example: $\mathbf{w}_i = \boldsymbol{x}^t$ for some assignment of samples t to hidden unit templates i.

Activation Function
Softplus

$$\phi(a) = \zeta(a) = \log(1 + e^{a})$$

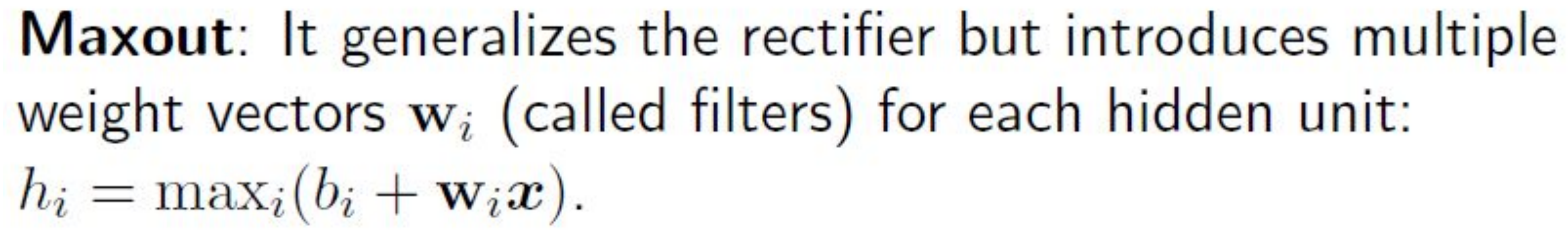
Activation Function
Hard tanh

Hard tanh: this is shaped similarly to the tanh and the rectifier but unlike the latter, it is bounded,
$\phi(a) = \max(-1, \min(1, a))$.

Activation Function
Absolute Value Rectification

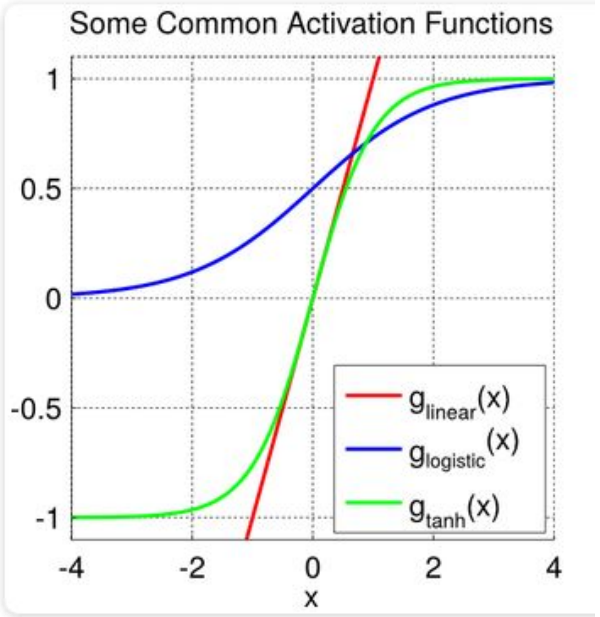
Absolute Value Rectification: $\phi(a) = |a|$

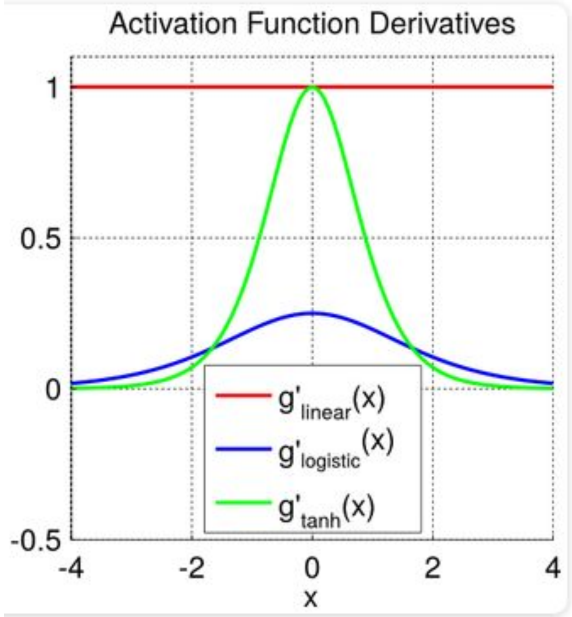
Activation Function
Maxout



Maxout: It generalizes the rectifier but introduces multiple weight vectors $\mathbf{w}_i$ (called filters) for each hidden unit:
$h_i = \max_i(b_i + \mathbf{w}_i \boldsymbol{x})$.

Activation Functions - Comparison

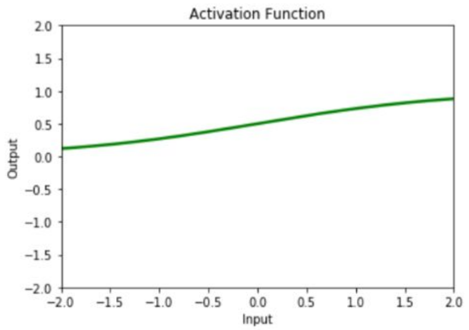




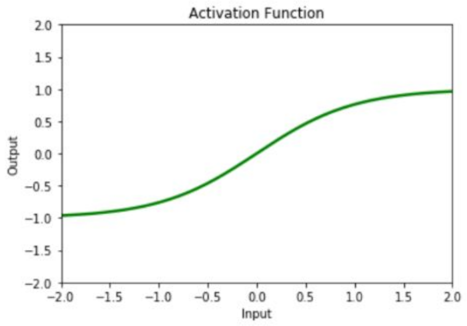
https://theclevermachine.wordpress.com/tag/tanh-function/

Activation Functions - Comparison

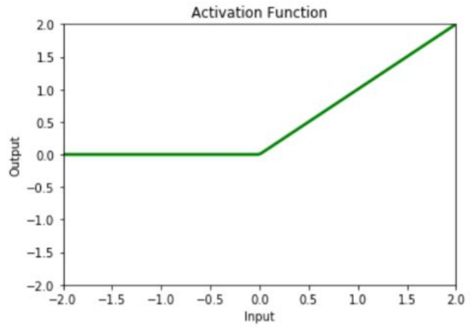
sigmoid

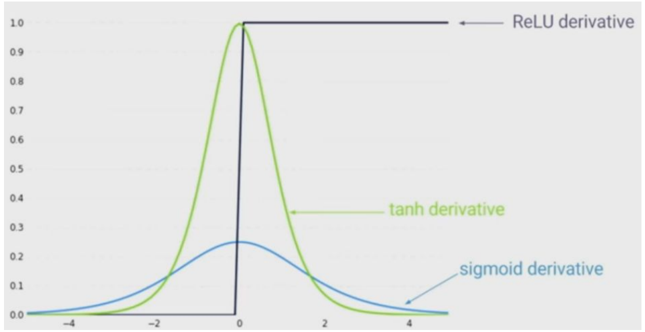


tanh



ReLU





Activation Functions - Comparison

The Identity/Linear	Useful for non-linear regression problems: can predict continuous target values using a linear combination of signals that arise from one or more layers of nonlinear transformations of the input.
Logistic sigmoid	Motivated somewhat by biological neurons and can be interpreted as the probability of an artificial neuron "firing" given its inputs. Can cause a neural network to get "stuck" during training. This is due in part to the fact that if a strongly-negative input is provided to the logistic sigmoid, it outputs values very near zero.
Hyperbolic Tangent	Strongly negative inputs to the tanh will map to negative outputs. Additionally, only zero-valued inputs are mapped to near-zero outputs. These properties make the network less likely to get "stuck" during training.

https://theclevermachine.wordpress.com/tag/tanh-function/

Feedforward Coefficients of Multi-Layer Neural Networks

Coefficients need to be (iteratively) modified, … to get (an ANN) model which represents input data well.

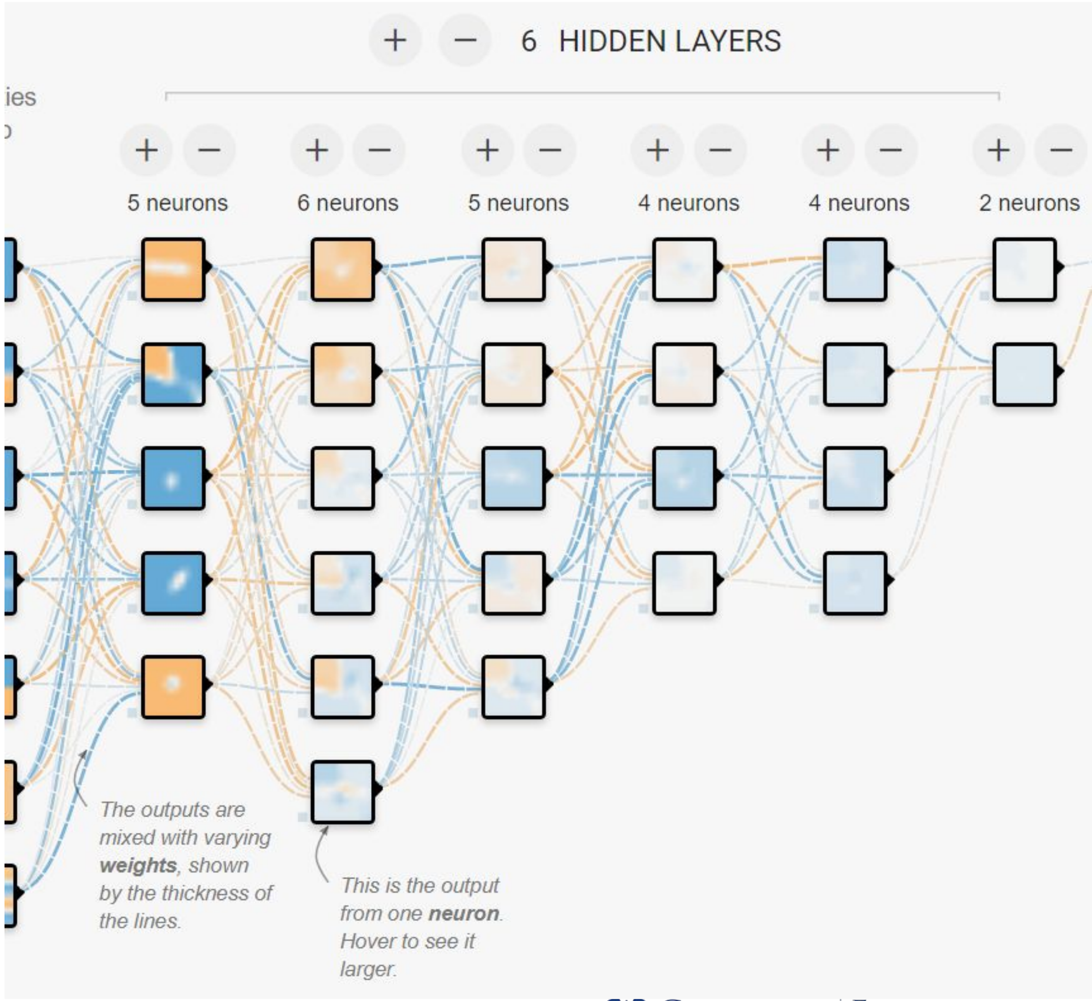
That will be accomplished by training, using Back-Propagation.

Deep Feedforward Networks or Multilayer Perceptrons

Many layers

with various number of neurons

[Playground.tensorflow]

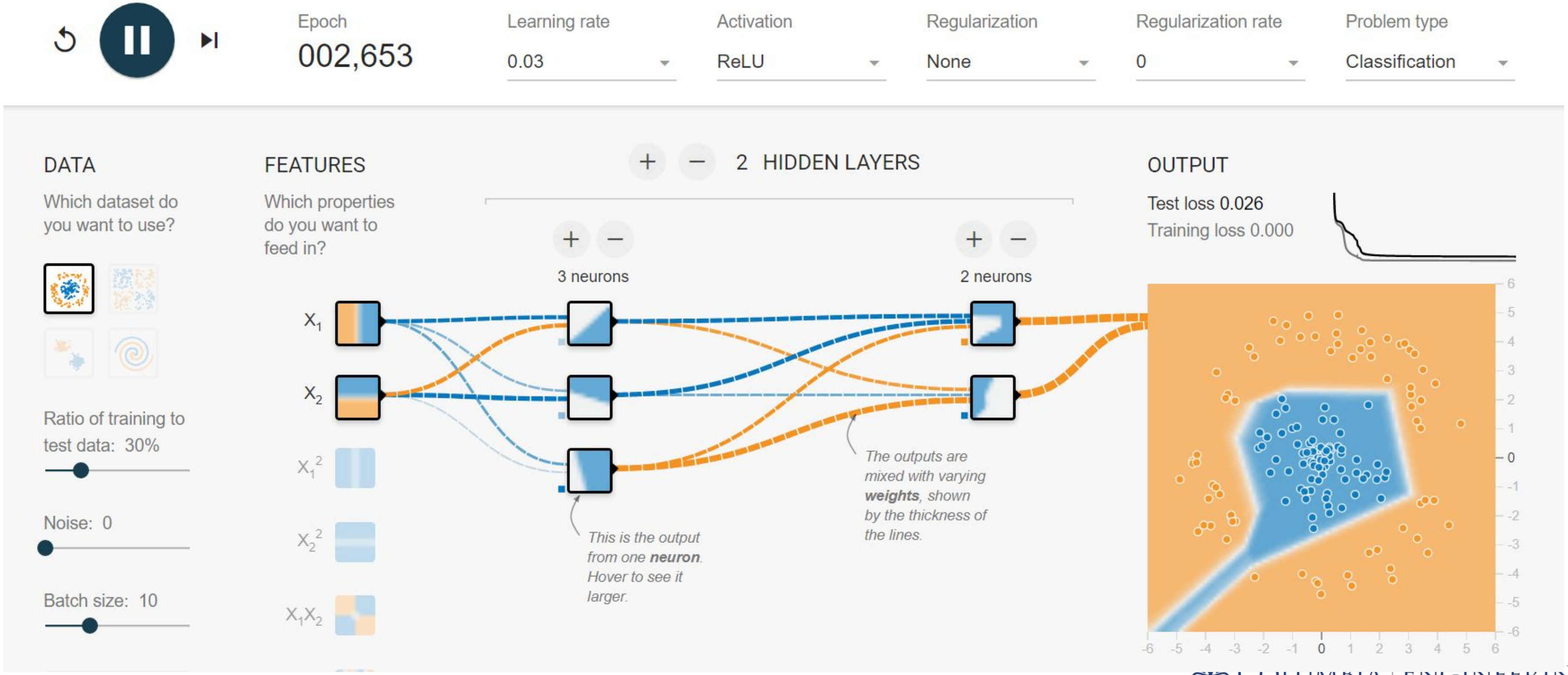


The outputs are mixed with varying ***weights****, shown by the thickness of the lines.*

This is the output from one ***neuron****. Hover to see it larger.*

© ZK

What do Individual Neurons Do (Playground)
Monitor Visual Area Intersections



Epoch 002,653

Learning rate 0.03

Activation ReLU

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 30%

Noise: 0

Batch size: 10

FEATURES

Which properties do you want to feed in?

2 HIDDEN LAYERS

3 neurons

2 neurons

*This is the output from one **neuron**. Hover to see it larger.*

*The outputs are mixed with varying **weights**, shown by the thickness of the lines.*

OUTPUT

Test loss 0.026

Training loss 0.000

Tensorflow Playground

An example of a complete neural network code repository:

- Source code, visualization

github code:

https://github.com/tensorflow/playground

https://github.com/tensorflow/playground/tree/master/src

- dataset.ts
- heatmap.ts
- linechart.ts
- nn.ts
- playground.ts
- seedrandom.d.ts
- state.ts

Artificial Neural Networks can Implement Stochastic Transformations

Traditional neural networks implement a deterministic transformation of some input variables $\boldsymbol{x}$. We can extend neural networks to implement stochastic transformations of $\boldsymbol{x}$ by simply augmenting the neural network with randomly sampled inputs $\boldsymbol{z}$.

The neural network can then continue to perform deterministic computation internally, but the function $f(\boldsymbol{x}, \boldsymbol{z})$ will appear stochastic to an observer who does not have access to $\boldsymbol{z}$. Provided that f is continuous and differentiable, we can then generally apply back-propagation as usual.

Let y be obtained by sampling the Gaussian distribution $\mathcal{N}(\mu, \sigma)$. We interpret the sampling process as the transformation

$$y = \mu + \sigma z,$$

where z is a Gaussian random variable $\mathcal{N}(0, 1)$.

Artificial Neural Networks
Back-Propagation for Stochastic Transformations

We are now able to back-propagate through the sampling operation, by regarding it as a deterministic operation with an additional input z.

Back-propagation through the sampling operation enables us to incorporate the transformation into a larger graph and, thereby, compute the derivatives of the loss function of interest $J(y)$.

We can also introduce functions that shape the distribution, e.g., $\mu = f(\boldsymbol{x}; \boldsymbol{\theta})$ and $\sigma = g(\boldsymbol{x}; \boldsymbol{\theta})$ and use back-propagation through this functions to derive $\nabla_{\boldsymbol{\theta}} J(y)$.

Artificial Neural Networks
Applications of Stochastic Transformations

More generally, using the representation

$$p(y|\boldsymbol{x};\boldsymbol{\theta}) = p(y|\boldsymbol{\omega}),$$

where $\boldsymbol{\omega}$ is a variable containing both parameters $\boldsymbol{\theta}$ and the inputs $\boldsymbol{x}$, we have

$$y = f(\boldsymbol{z}, \boldsymbol{\omega}),$$

where $\boldsymbol{z}$ is a source of randomness. Here, $\boldsymbol{\omega}$ must not be a function of $\boldsymbol{z}$, and $\boldsymbol{z}$ must not be a function of $\boldsymbol{\omega}$. This is often called the reparametrization trick.

In neural network applications, $\boldsymbol{z}$ is typically drawn from some simple distribution, such as a unit uniform or unit Gaussian distribution. Complex distributions are achieved by allowing the deterministic portion of the network to reshape its input.

Artificial Neural Networks
Reparametrization Trick Example

Assume we have a normal distribution q that is parameterized by θ, specifically $q_\theta(x) = N(\theta, 1)$. We want to solve the below problem

$$\min_\theta \quad E_q[x^2]$$

This is of course a rather silly problem and the optimal θ is obvious. We want to understand how the reparameterization trick helps in calculating the gradient of this objective $E_q[x^2]$.

One way to calculate $\nabla_\theta E_q[x^2]$ is as follows

$$\nabla_\theta E_q[x^2] = \nabla_\theta \int q_\theta(x) x^2 dx = \int x^2 \nabla_\theta q_\theta(x) \frac{q_\theta(x)}{q_\theta(x)} dx = \int q_\theta(x) \nabla_\theta \log q_\theta(x) x^2 dx$$
$$= E_q[x^2 \nabla_\theta \log q_\theta(x)]$$

Goker Erdogan, "Reparameterization Trick", http://gokererdogan.github.io/2016/07/01/reparameterization-trick/

Artificial Neural Networks Reparametrization Trick Example

For our example where $q_\theta(x) = N(\theta, 1)$, this method gives

$$\nabla_\theta E_q[x^2] = E_q[x^2(x - \theta)]$$

Reparameterization trick is a way to rewrite the expectation so that the distribution with respect to which we take the gradient is independent of parameter θ. To achieve this, we need to make the stochastic element in q independent of θ. Hence, we write x as

$$x = \theta + \epsilon, \qquad \epsilon \sim N(0, 1)$$

Then, we can write

$$E_q[x^2] = E_p[(\theta + \epsilon)^2]$$

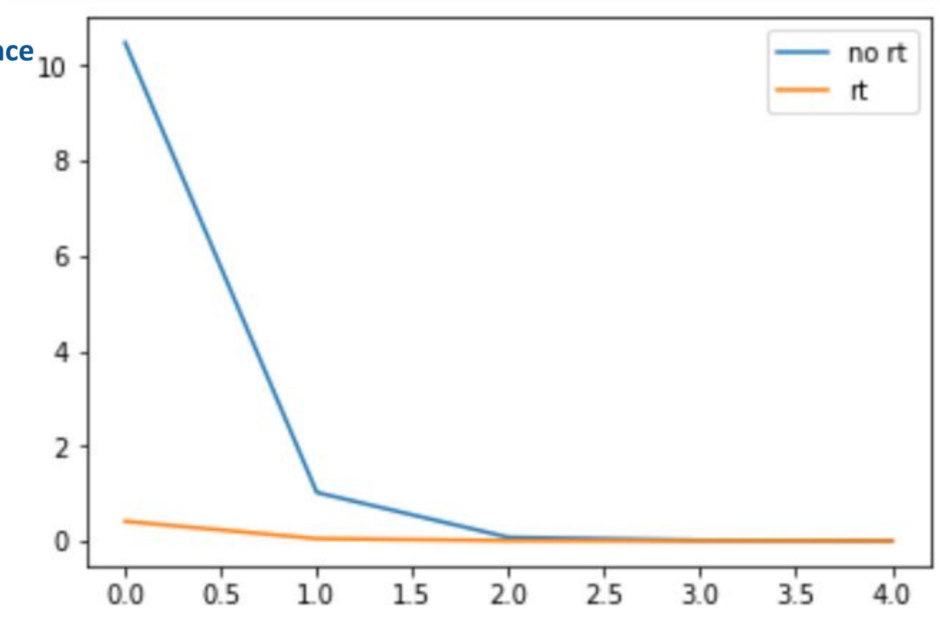
where p is the distribution of ϵ, i.e., $N(0, 1)$. Now we can write the derivative of $E_q[x^2]$ as follows

$$\nabla_\theta E_q[x^2] = \nabla_\theta E_p[(\theta + \epsilon)^2] = E_p[2(\theta + \epsilon)]$$

Artificial Neural Networks
Reparametrization Trick

Outcome: Reduced variance with reparametrization trick

Variance



X, where sample size=(10^{X+1})

Jupyter Notebook", https://github.com/gokererdogan/Notebooks/blob/master/Reparameterization%20Trick.ipynb

Artificial Neural Networks
The Universal Approximation Theorem

A feedforward network

can approximate any Borel measurable function

Artificial Neural Networks
The Universal Approximation Theorem

A feedforward network with a linear output layer and at least one hidden layer with any squashing activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function,

from one finite-dimensional space to another with any desired non-zero amount of error,

provided that the network is given enough hidden units.

The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well.

Artificial Neural Networks
The Universal Approximation Theorem - Power

Feedforward networks provide a universal system

we can find a large MLP that

approximates/represents the function

Artificial Neural Networks
The Universal Approximation Theorem - Power

Any continuous function on a closed and bounded subset of R is Borel measurable and therefore may be approximated by a neural network.

A neural network may also approximate any function mapping from any finite dimensional discrete space to another.

Feedforward networks provide a universal system for representing functions: given a function, we can find a large MLP that approximates/represents the function.

Artificial Neural Networks
The Universal Approximation Theorem - Limitations

no guarantees that

the training algorithm will be able to learn it.

no universal procedure

that will generalize

Artificial Neural Networks
The Universal Approximation Theorem - Limitations

Even if the MLP is able to represent a function, there are no guarantees that the training algorithm will be able to learn it.

Learning can fail for two different reasons:

- **the optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function;**
- **the training algorithm might choose the wrong function due to overfitting.**

There is no universal procedure for examining a training set of specific examples and choosing a function that will generalize to points not in the training set.

Artificial Neural Networks
The Universal Approximation Theorem - Limitations

there exists a network **to achieve any accuracy;** **does not say** **how large the network**

exponential number of hidden units

may be required

2^{2^n} **a**

© ZK

Artificial Neural Networks
The Universal Approximation Theorem - Limitations

The UAT says that there exists a network large enough to achieve any degree of accuracy; it does not say, however, how large the network will be.

Unfortunately, in the worst case, an exponential number of hidden units may be required.

This is easiest to see in the binary case: the number of possible binary functions of vectors v∈{0; 1}^{n} is 2^{2^n}. Selecting one such function requires 2^n bits, which will in general require O(2^n)degrees of freedom.

Artificial Neural Networks
Single Layer Can do All

A feedforward network with a single layer is sufficient to represent any function, but it may be unfeasibly large and may fail to learn and generalize correctly.

Both of these failure modes suggest the use of deeper models.

A visual proof that neural nets can compute any function:
http://neuralnetworksanddeeplearning.com/chap4.html

Artificial Neural Networks
Deep Models

representation learning

- **discovering a set of underlying factors** **described in terms of other, simpler underlying factors**
- **is a computer program consisting of multiple steps**

not necessarily factors of variation **instead be analogous to counters or pointers**

Artificial Neural Networks
Deep Models

From a representation learning view, learning with a deep architecture

- consists of discovering a set of underlying factors of variation that can be described in terms of other, simpler underlying factors of variation,
- expresses the belief that the function we want to learn is a computer program consisting of multiple steps, where each step makes use of the previous steps' output.

These intermediate outputs are not necessarily factors of variation, but can instead be analogous to counters or pointers that the network uses to organize its internal processing.

Using deep architectures does indeed express a useful prior over the space of functions the model learns.

Artificial Neural Networks
Why do They Work in 2010+

Recent performance improvements of deep neural networks can be attributed to

- increases in computational power
- increases in the size of datasets

Artificial Neural Networks
The Role of Piecewise Linear Units

Recent algorithmic improvements are due to the use of piecewise linear units, such as absolute value rectifiers and rectified linear units.

- **They consist of two linear pieces and their behavior is driven by a single weight vector.**
- **For small datasets, using rectifying nonlinearities is even more important than learning the weights of the hidden layers.**
  - **Random weights are sucient to propagate useful information through a rectified linear network, allowing the classifier layer at the top to learn how to map different feature vectors to class identities.**

Artificial Neural Networks
The Role of Piecewise Linear Units

Recent algorithmic improvements are due to the use of piecewise linear units, such as absolute value rectifiers and rectified linear units.

- **Learning is far easier in deep rectified linear networks than in deep networks that have curvature or two-sided saturation in their activation functions.**
- **Just as piecewise linear networks are good at propagating information forward, back-propagation in such a network is also piecewise linear**
  - **and propagates information about the error derivatives to all of the gradients in the network.**

Artificial Neural Networks
The Role of Piecewise Linear Units

The use of rectified linear units is strongly motivated by biological considerations. The half-rectifying non-linearity was intended to capture the following properties of biological neurons:

- **for some inputs, biological neurons are completely inactive.**
- **for some inputs, a biological neuron's output is proportional to its input.**
- **most of the time, biological neurons operate in the regime where they are inactive (e.g., they have sparse activations).**

Backup Slides

Various