SDU Summer School

Deep Learning

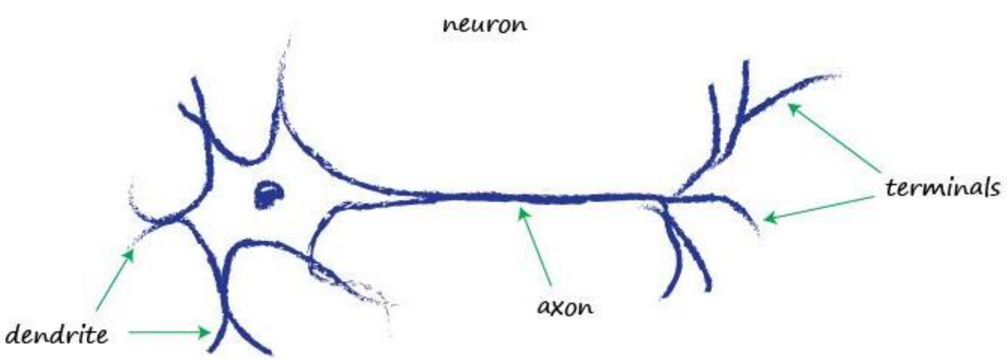
Summer 2018

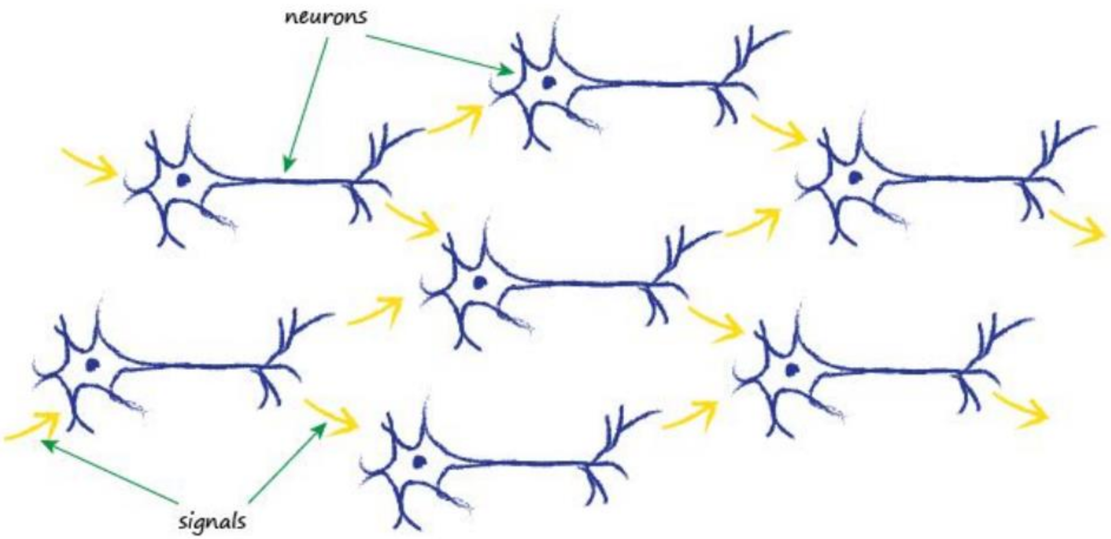
Deep Feedforward Networks

Deep Feedforward Networks

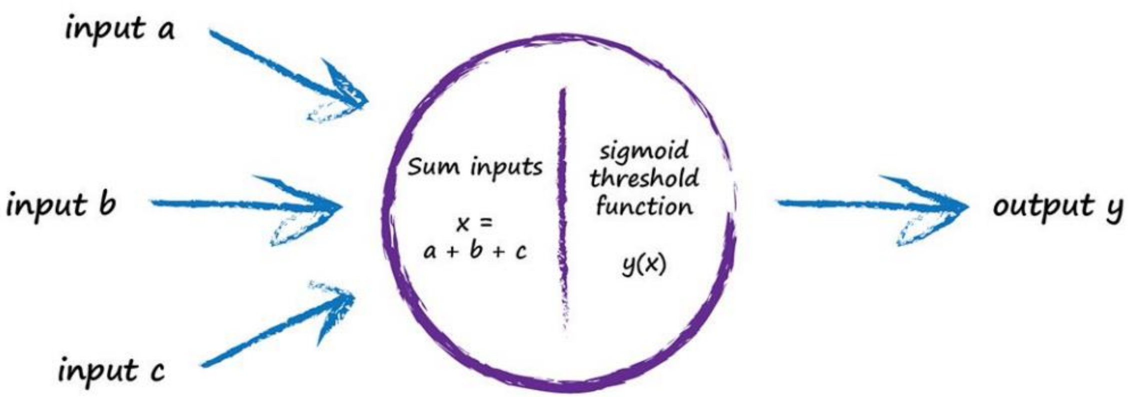
- **PART I**
  - **Feedforward Networks**
  - Output Units
  - Hidden Units
  - Architecture Design
- **PART II**
  - Gradient-Based Learning
  - Backpropagation

The Neuron … in Nature





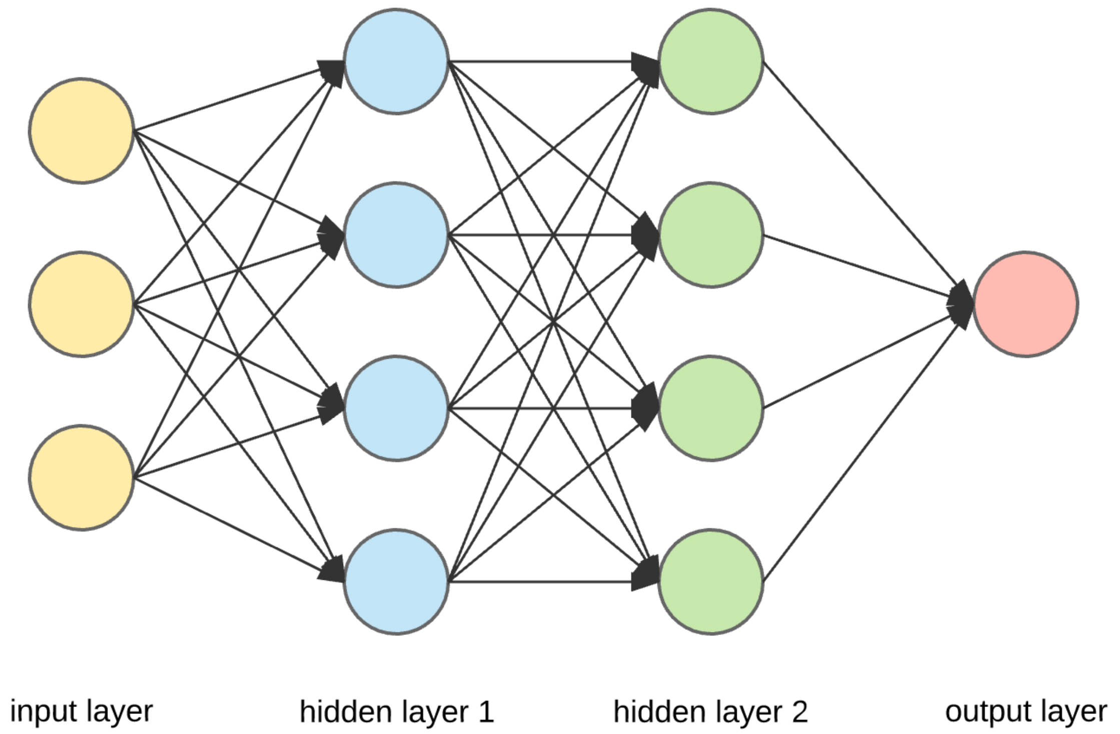
The Artificial Neuron



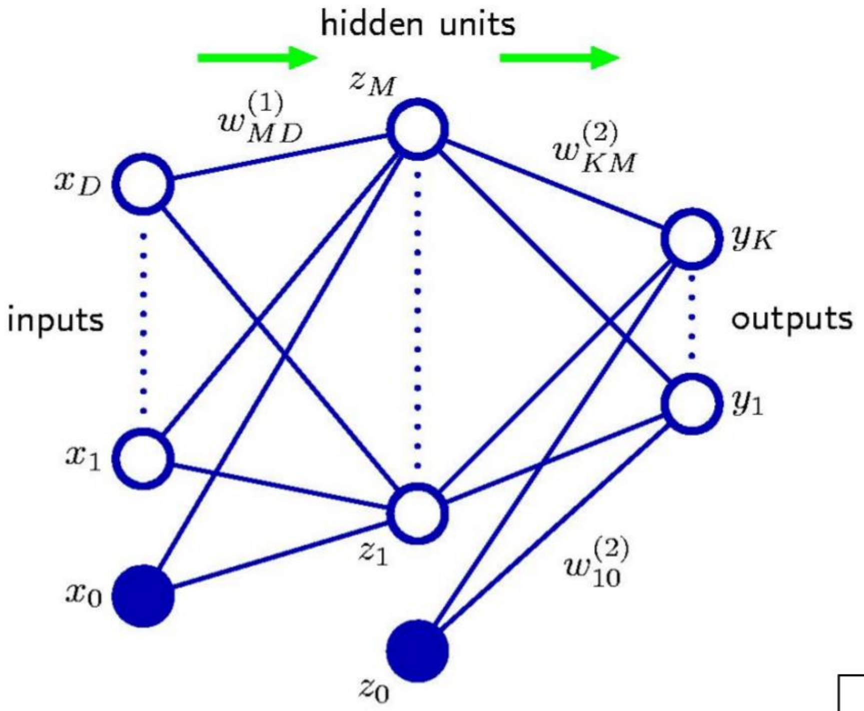
- An artificial Neuron consists of
  - A number of weighted inputs
  - An activation function
  - The generated output

Connecting to a Network

- We normally have more than one node
- Multiple Nodes are arranged in layers
- Each layer receives the generated output from the previous layer



More Mathematically View



K outputs $y_1, ..y_K$ for a given input x
Hidden layer consists of M units

$$y_k(\mathrm{x},\mathrm{w}) = \sigma\left(\sum_{j=1}^{M} w^{(2)}_{kj} h\left(\sum_{i=1}^{D} w^{(1)}_{ji} x_i + w^{(1)}_{j0}\right) + w^{(2)}_{k0}\right)$$

$$\boxed{\begin{array}{l} f_m{}^{(1)} = z_m = h(\mathrm{x},\mathrm{w}_m{}^{(1)}),\ m=1,..M \\ f_k{}^{(2)} = \sigma\ (\mathrm{z},\mathrm{w}_k{}^{(2)}),\ k=1,..K \end{array}}$$

Note that there are M versions of $f^{(1)}$ and K versions of $f^{(2)}$

More Compact Representation

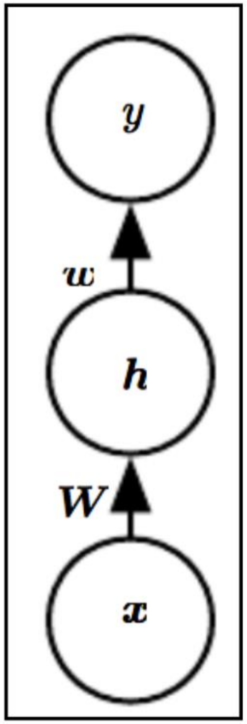
- We can summarize these networks to their essential parts:

- We receive the vector $\boldsymbol{x}$ and perform an affine transformation according to the weight matrix

$$\boldsymbol{h}' = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

  - Does this ring a bell? Linear regression anyone?

- Afterward, the internal representation is component wise feed through an activation function $h_i = g(h_i')$ to generate the input vector $\boldsymbol{h}$ for the next layer



Composition of Complicated Functions

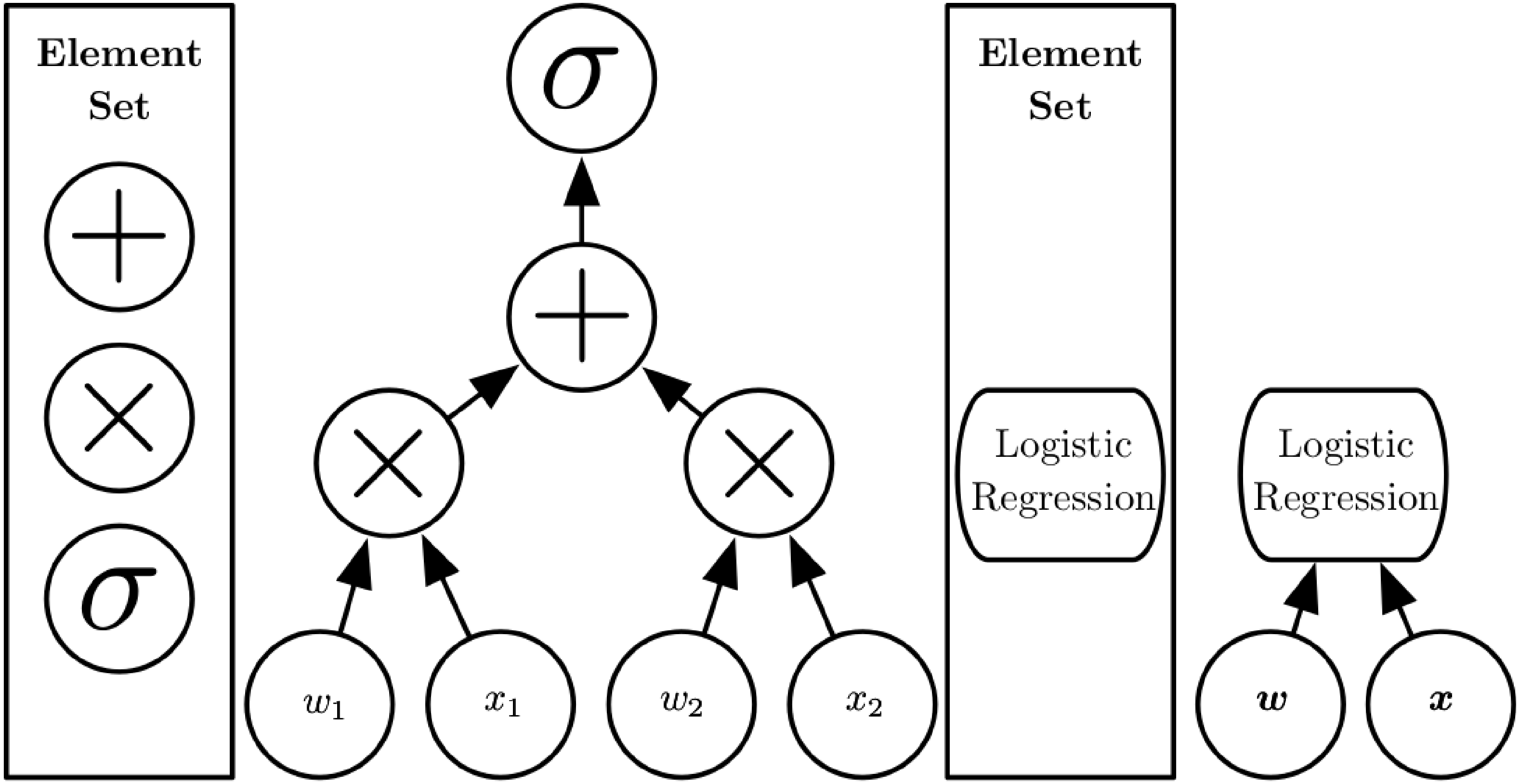
- Model is associated with a directed acyclic graph describing how functions composed
  - E.g., functions $f^{(1)}$, $f^{(2)}$, $f^{(3)}$ connected in a chain to form

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

  - $f^{(1)}$ is the first layer, $f^{(2)}$ the second, etc.
  - The length of the chain is the depth of the model

- The name deep learning arises from this terminology
- Final layer of a feedforward network, (here $f^{(3)}$) is the output layer

Regarding the Depth of a Model



$$p(y = 1|\boldsymbol{x};\boldsymbol{w}) = \sigma\left([w_1, w_2]\begin{bmatrix}x_1\\x_2\end{bmatrix}\right)$$

What are Hidden Layers

- Behavior of other layers is not directly specified by the data

- Learning algorithm must decide how to use those layers to produce a final value that is close to y

- Training data does not say what individual layers should do
  - This is one of the main tasks: Define the appropriate structure of the network and the hidden layers

- Since the desired output for these layers is not shown, they are called hidden layers

Extending Linear Models

To represent non-linear functions of $\boldsymbol{x}$

- Apply linear model to transformed input $\phi(\boldsymbol{x})$ with non-linear ϕ
  - Equivalently kernel trick of SVM obtains nonlinearity

- Function is nonlinear wrt x but linear wrt $\phi(x)$
- We can think of ϕ as providing a set of features providing a new representation for x

- Get ϕ
  - Generic functions: There exists a set of useful kernel functions
  - Manually engineer ϕ: Laborious and not transformable
  - **Deep Learning**: Learn ϕ

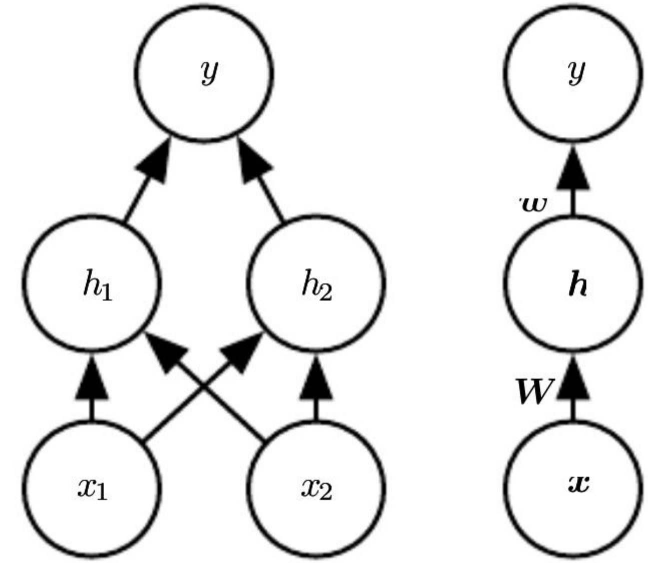
The XOR Problem

- We are trying to approximate the XOR function
  - Our training points: $X = \{[0,0]^T, [1,0]^T, [0,1]^T, [1,1]^T\}$
  - Using linear regression: $y = f([x_1, x_2]; \boldsymbol{w}, b) = \boldsymbol{x^T w} + b$ to learn XOR
  - When using "mean squared error" as loss function, the result would be $\boldsymbol{w} = 0$ and $b = \frac{1}{2}$ … horizontal line outputting 0.5 everywhere

The XOR Problem

- We are trying to approximate the XOR function
  - Our training points: $X = \{[0,0]^T, [1,0]^T, [0,1]^T, [1,1]^T\}$
  - Using linear regression: $y = f([x_1, x_2]; \boldsymbol{w}, b) = \boldsymbol{x^T w} + b$ to learn XOR
  - When using "mean squared error" as loss function, the result would be $\boldsymbol{w} = 0$ and $b = \frac{1}{2}$ … horizontal line outputting 0.5 everywhere

- Now lets use a FFN
  - One hidden layer, containing two units
  - Both representations are equivalent
    - Matrix $\boldsymbol{W}$ describes mapping $\boldsymbol{x}$ to $\boldsymbol{h}$
    - Vector $\boldsymbol{w}$ describes mapping $\boldsymbol{h}$ to y
    - (biases omitted)



What is gonna be computed

- Layer 1 (hidden layer): vector of hidden units $\boldsymbol{h}$ computed by function

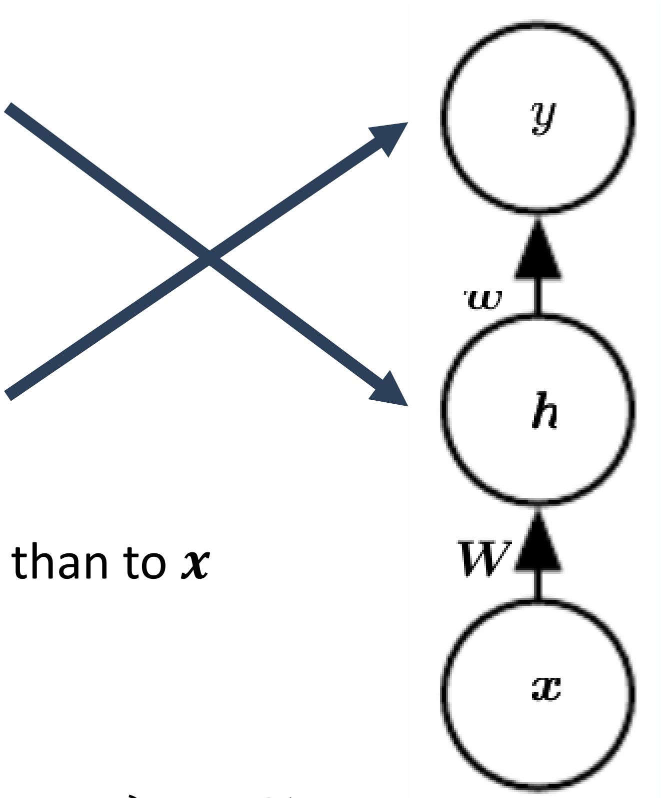
$$\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, c)$$

  - c are bias variables

- Layer 2 (output layer) computes

$$y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$$

  - $\boldsymbol{w}$ are linear regression weights
  - Output is linear regression applied to $\boldsymbol{h}$ rather than to $\boldsymbol{x}$



- Complete model is

$$f(\boldsymbol{x}; \boldsymbol{W}, c, \boldsymbol{w}, b) = f^{(2)}(f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, c), \boldsymbol{w}, b)$$

Activation Function

- If we choose both $f^{(1)}$ and $f^{(2)}$ to be linear, the total function will still be linear; which is insufficient (as already seen)

Activation Function:

- In linear regression we used a vector of weights $\boldsymbol{w}$ and scalar bias b

$$y = \boldsymbol{x}^{\boldsymbol{T}}\boldsymbol{w} + b$$

- Now we describe an affine transformation from a vector $\boldsymbol{x}$ to a vector $\boldsymbol{h}$, so an entire vector of bias parameters is needed
- Activation function g is typically chosen to be applied element-wise

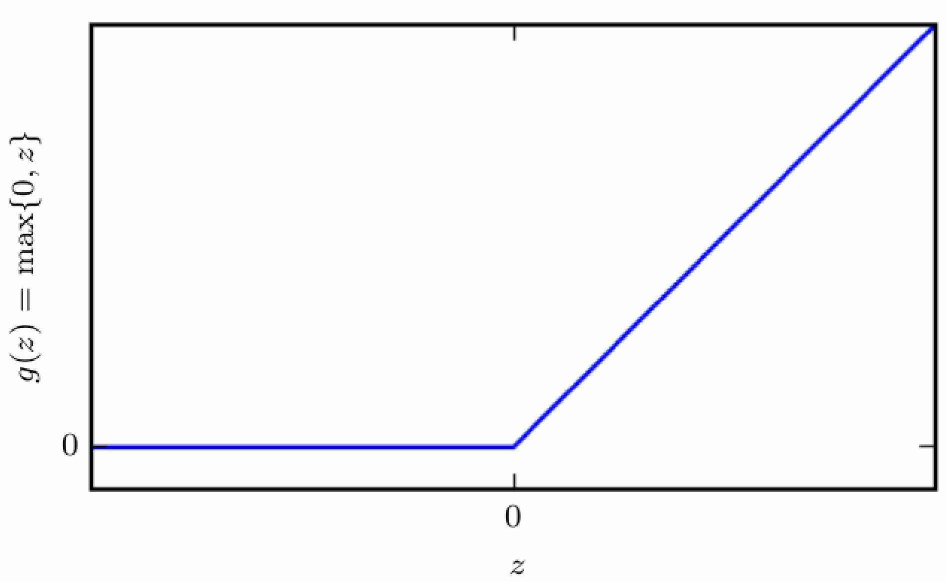
$$h_i = g(x^T W_{:i} + c_i)$$

Default Activation Function: ReLU

- Activation:

$$g(z) = \max\{0, z\}$$

- This already yields to a non-linear transformation
  - But still piecewise linear
  - Preserves couple of nice properties making gradient-based learning easy
  - Generalizes well



Back to our XOR Problem

- The complete equation is now:

$$f(\boldsymbol{x};\boldsymbol{W},\boldsymbol{c},\boldsymbol{w},b) = \boldsymbol{w}^T \max\{0, \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c}\} + b$$

Lets use the following weights:

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0 \ and\ \boldsymbol{X} = \begin{bmatrix} 0101 \\ 0011 \end{bmatrix}^T$$

Back to our XOR Problem

- The complete equation is now:

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^T \max\{0, \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{c}\} + b$$

Lets use the following weights:

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0 \ and \ \boldsymbol{X} = \begin{bmatrix} 0101 \\ 0011 \end{bmatrix}^T$$

$$\boxed{\boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}}$$

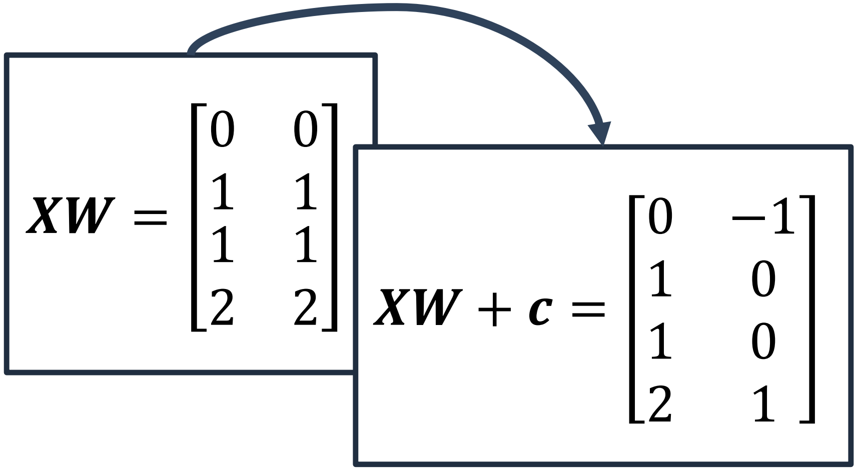
Back to our XOR Problem

- The complete equation is now:

$$f(\boldsymbol{x};\boldsymbol{W},\boldsymbol{c},\boldsymbol{w},b)=\boldsymbol{w}^T\max\{0,\boldsymbol{W}^T\boldsymbol{x}+\boldsymbol{c}\}+b$$

Lets use the following weights:

$$\boldsymbol{W}=\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix},\boldsymbol{c}=\begin{bmatrix}0\\ -1\end{bmatrix},\boldsymbol{w}=\begin{bmatrix}1\\ -2\end{bmatrix},b=0\ and\ \boldsymbol{X}=\begin{bmatrix}0101\\ 0011\end{bmatrix}^T$$



$$\boldsymbol{XW}=\begin{bmatrix}0 & 0\\ 1 & 1\\ 1 & 1\\ 2 & 2\end{bmatrix}$$

$$\boldsymbol{XW}+\boldsymbol{c}=\begin{bmatrix}0 & -1\\ 1 & 0\\ 1 & 0\\ 2 & 1\end{bmatrix}$$

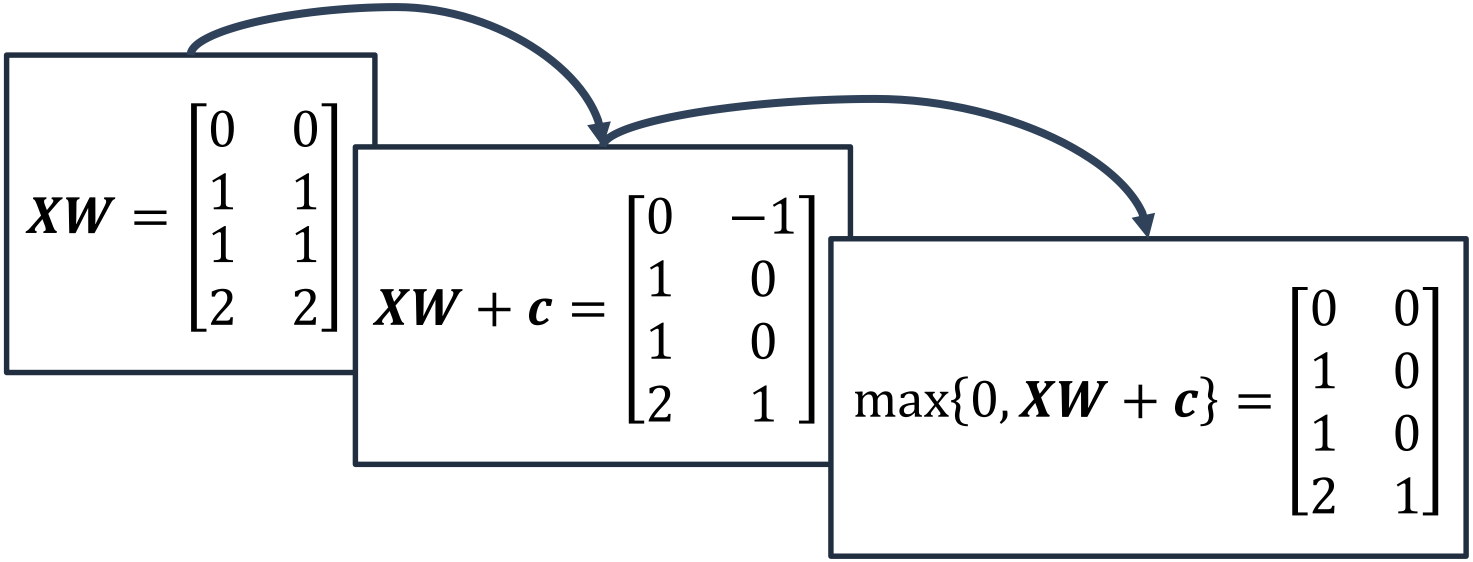
Back to our XOR Problem

- The complete equation is now:

$$f(\boldsymbol{x};\boldsymbol{W},\boldsymbol{c},\boldsymbol{w},b)=\boldsymbol{w}^T\max\{0,\boldsymbol{W}^T\boldsymbol{x}+\boldsymbol{c}\}+b$$

Lets use the following weights:

$$\boldsymbol{W}=\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix},\boldsymbol{c}=\begin{bmatrix}0\\-1\end{bmatrix},\boldsymbol{w}=\begin{bmatrix}1\\-2\end{bmatrix},b=0\ and\ \boldsymbol{X}=\begin{bmatrix}0101\\0011\end{bmatrix}^T$$



$$\boldsymbol{XW}=\begin{bmatrix}0 & 0\\1 & 1\\1 & 1\\2 & 2\end{bmatrix}$$

$$\boldsymbol{XW}+\boldsymbol{c}=\begin{bmatrix}0 & -1\\1 & 0\\1 & 0\\2 & 1\end{bmatrix}$$

$$\max\{0,\boldsymbol{XW}+\boldsymbol{c}\}=\begin{bmatrix}0 & 0\\1 & 0\\1 & 0\\2 & 1\end{bmatrix}$$

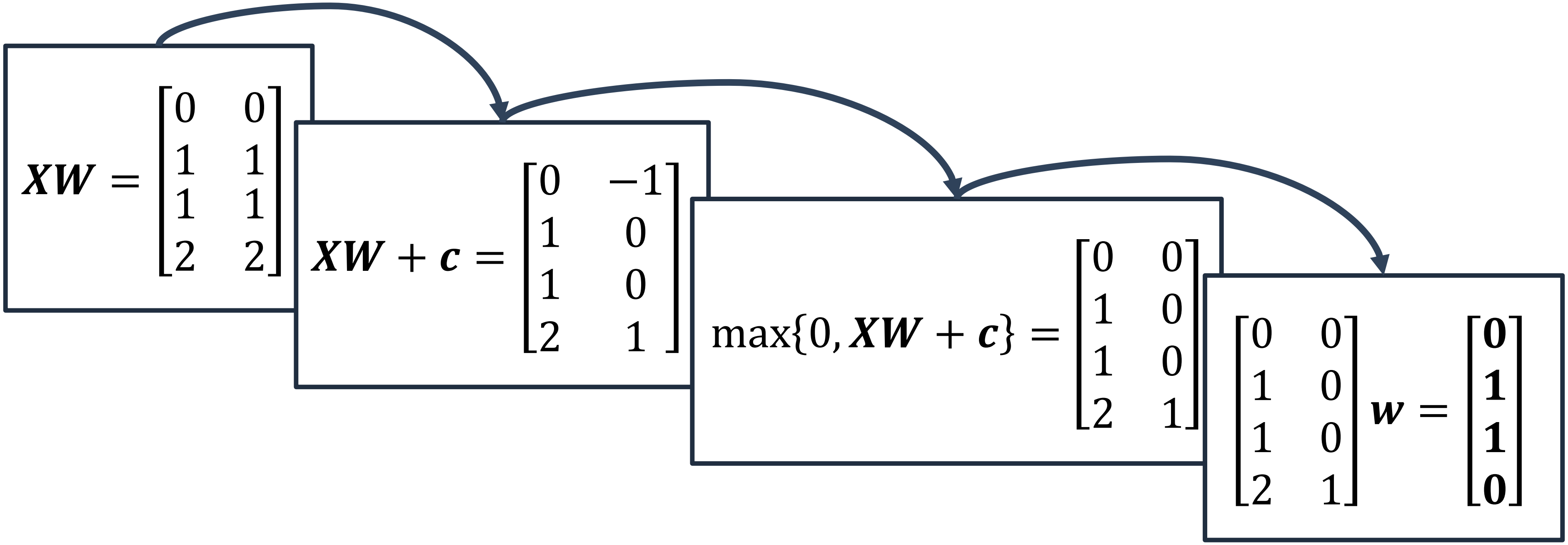
Back to our XOR Problem

- The complete equation is now:

$$f(\boldsymbol{x};\boldsymbol{W},\boldsymbol{c},\boldsymbol{w},b) = \boldsymbol{w}^T \max\{0, \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c}\} + b$$

Lets use the following weights:

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0 \; and \; \boldsymbol{X} = \begin{bmatrix} 0101 \\ 0011 \end{bmatrix}^T$$



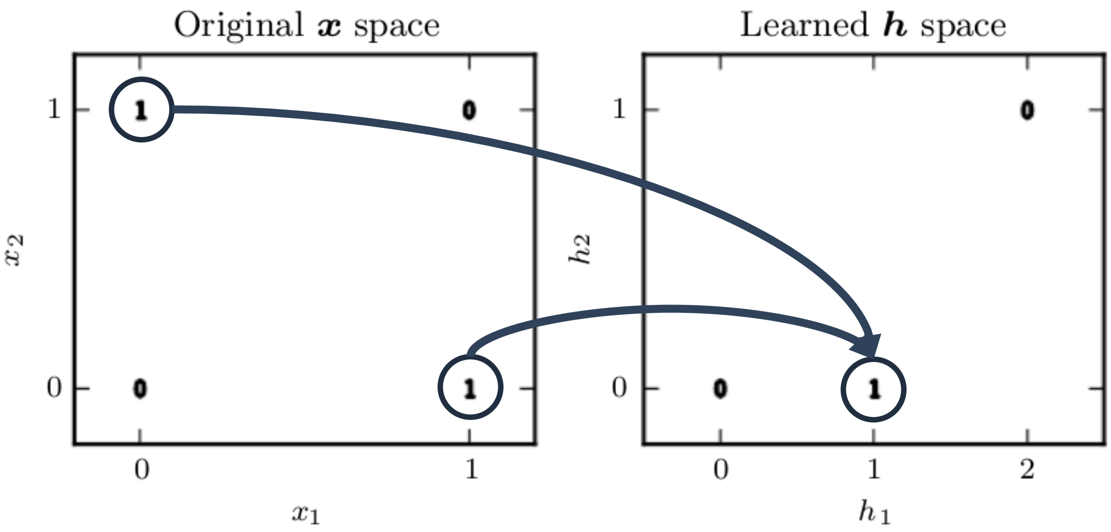
$$\boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

$$\boldsymbol{XW} + \boldsymbol{c} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\max\{0, \boldsymbol{XW} + \boldsymbol{c}\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \boldsymbol{w} = \begin{bmatrix} \mathbf{0} \\ \mathbf{1} \\ \mathbf{1} \\ \mathbf{0} \end{bmatrix}$$

“Learned XOR”



- Two points that must have output 1 have been collapsed into one
- Can be separated in a linear model:
  - For fixed h_2 output has to increase in h_1
- Of course, we “cheated”: We have “guessed” the parameters correctly which lead to the desired result
- In reality: Millions of parameters which we have to actually learn

Deep Feedforward Networks

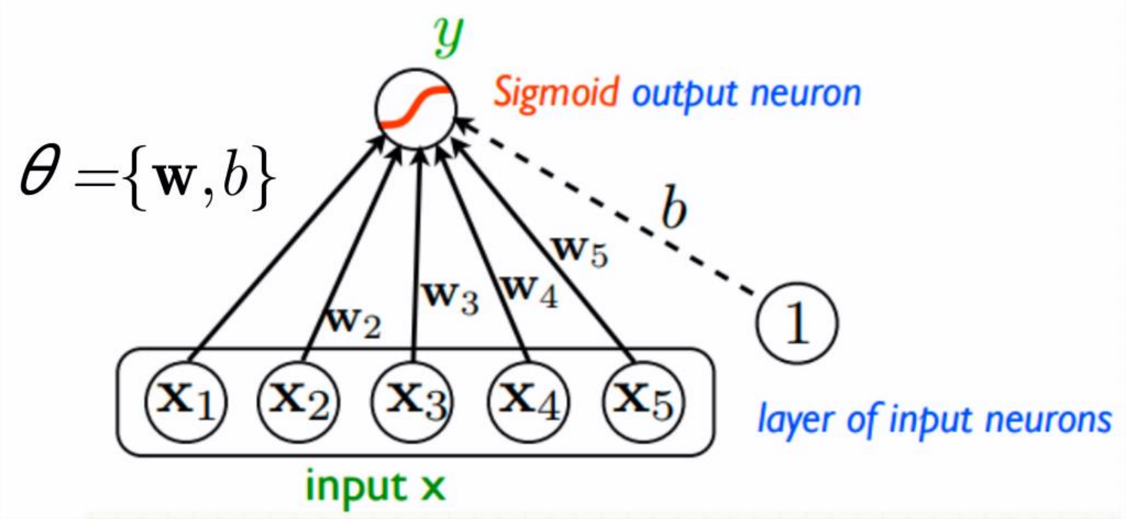
- **PART I**
  - Feedforward Networks
  - **Output Units**
  - Hidden Units
  - Architecture Design
- **PART II**
  - Gradient-Based Learning
  - Backpropagation

Output Units

- The choice of cost function is **tightly** coupled with the choice of output unit (later…)

- Most of the time we use cross-entropy between data distribution and model distribution (later …)

- Choice of how to represent the output then determines the form of the cross-entropy function



Role of a Output Unit

- Any output unit is in principal also usable as a hidden unit

- A feedforward network provides a hidden set of features

$$\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$$

- Role of output layer is to provide some additional transformation from the features to the task that network must perform

- Common Output Units
  - **Linear units**: no non-linearity (used for Gaussian distributions)
  - **Sigmoid units** (used for Bernoulli distributions)
  - **Softmax units** (used for Multinoulli distributions)

Linear Units for Gaussian Output Distributions

- Linear unit: simple output based on affine transformation with no nonlinearity

- Given features h, a layer of linear output units produces a vector

$$\hat{\boldsymbol{y}} = \boldsymbol{W}^T \boldsymbol{h} + \boldsymbol{b}$$

- Linear units are often used to produce mean $\hat{\boldsymbol{y}}$ of a conditional Gaussian distribution

$$P(\boldsymbol{y}|\boldsymbol{x}) = N(\boldsymbol{y}; \hat{\boldsymbol{y}}, \sigma^2)$$

Sigmoid Units for Bernoulli Output Distributions

- Task of predicting value of binary variable y
  - Classification problem with two classes

- Maximum likelihood approach is to define a Bernoulli distribution over y conditioned on $\boldsymbol{x}$

- Neural net needs to predict $p(y = 1|\boldsymbol{x}) \in [0,1]$:

$$P(y = 1|\boldsymbol{x}) = \max\{0, \min\{1, \boldsymbol{w}^T\boldsymbol{h} + \boldsymbol{b}\}\}$$

  - We would define a valid conditional distribution, but cannot train it effectively with gradient descent
  - A gradient of 0: learning algorithm cannot be guided

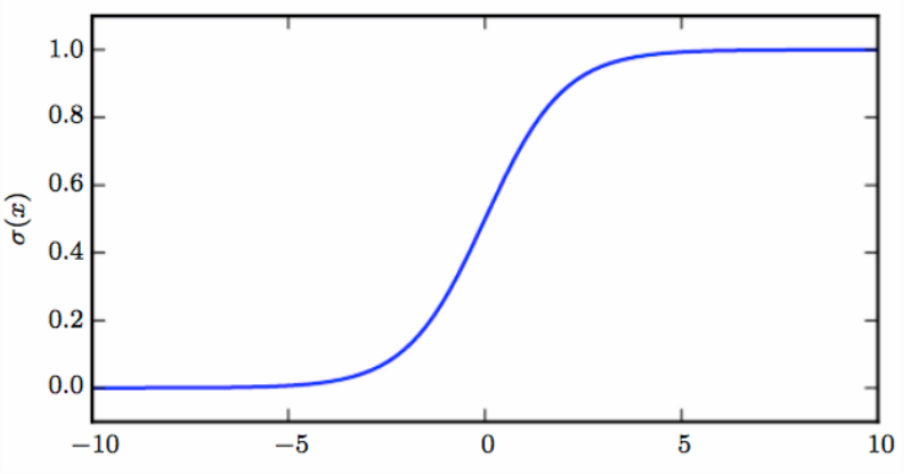
Sigmoid Units

- Sigmoid always gives a gradient

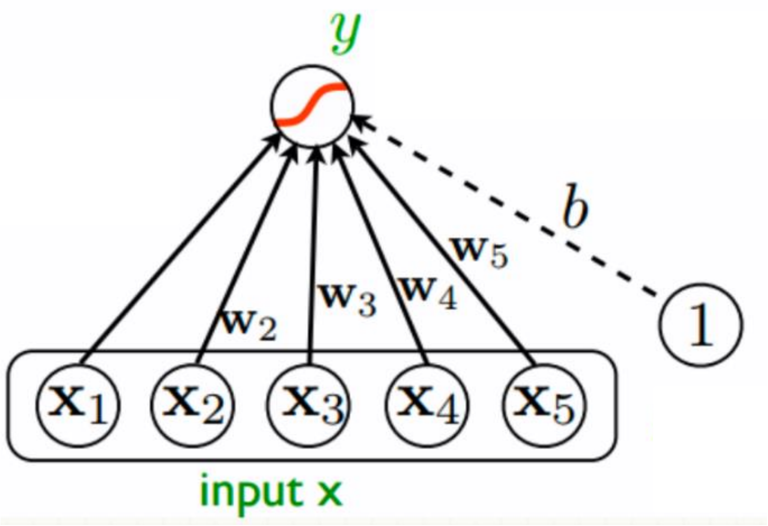
$$\hat{y} = \sigma(\boldsymbol{w}^T \boldsymbol{h} + b)$$

with

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



- Sigmoid Unit has two components:
  - A linear layer to compute
    $$z = \boldsymbol{w}^T \boldsymbol{h} + b$$
  - A sigmoid activation function to convert z into a probability



Softmax units for Multinoulli Output

- Any time we want a probability distribution over a discrete variable with n values we may us the softmax function

- Softmax most often used for output of classifiers to represent distribution over n classes
  - Also inside the model itself when we wish to choose between one of n options

- For the Binary case we have produced a single number

$$\hat{y} = P(y = 1|\boldsymbol{x})$$

- Now, we have to produce a vector $\hat{\boldsymbol{y}}$

$$\hat{y}_i = P(y = i|\boldsymbol{x})$$

Softmax Defintion

- The same procedure as for the Bernoulli distribution

- We have a linear layer predicting unnormalized log probabilities

$$\boldsymbol{z} = \boldsymbol{W}^T \boldsymbol{h} + \boldsymbol{b}$$

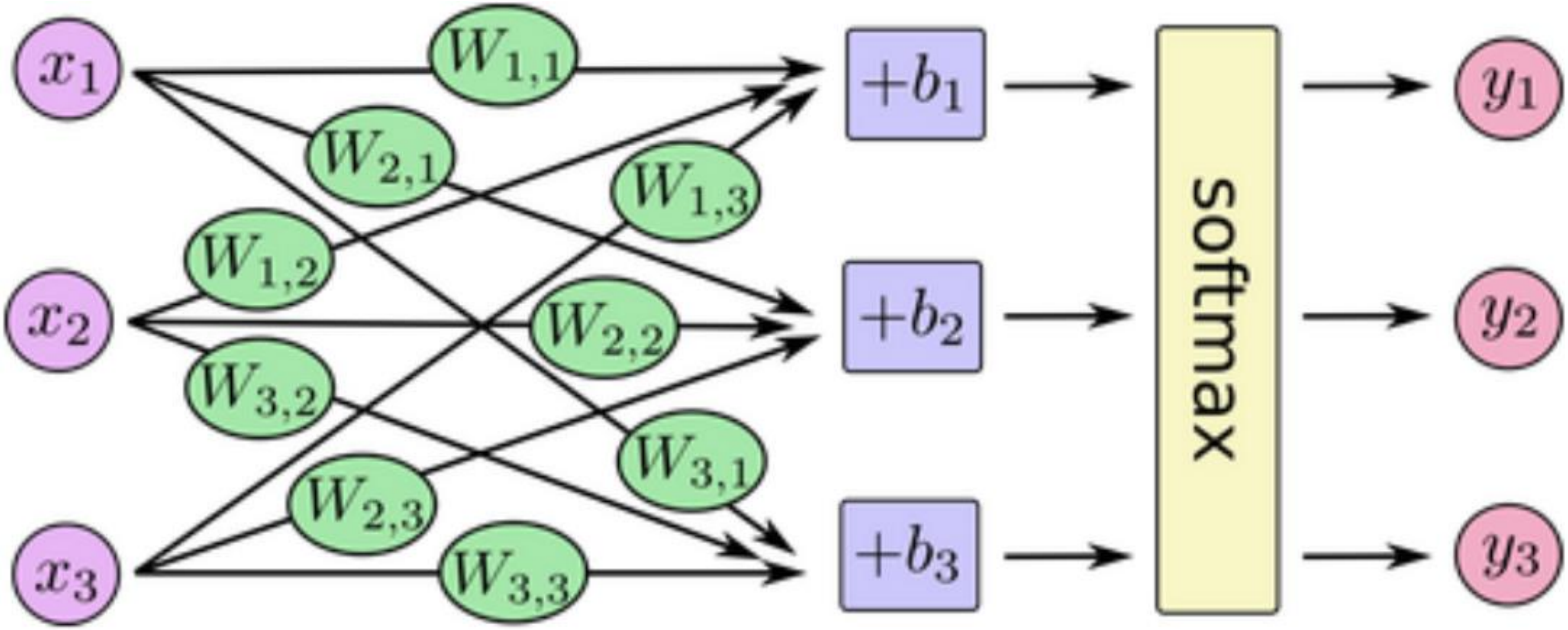
with

$$z_i = \log \hat{P}(y = i | \boldsymbol{x})$$

- Softmax can then exponentiate and normalize $\boldsymbol{z}$ to get $\hat{\boldsymbol{y}}$

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Softmax Regression



Deep Feedforward Networks

- **PART I**
  - Feedforward Networks
  - Output Units
  - **Hidden Units**
  - Architecture Design
- **PART II**
  - Gradient-Based Learning
  - Backpropagation

General Construction of a Hidden Unit

- Very Similar to an output unit

- Accepts a vector of inputs $\boldsymbol{x}$ and computes an affine transformation $\boldsymbol{z} = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$

- Computes an element-wise non-linear function $g(\boldsymbol{z})$

- Most hidden units are distinguished from each other by the choice of activation function $g(z)$
  - We look at: ReLU, Sigmoid and tanh, and other hidden units

Choice of Hidden Unit

- We now look at how to choose the type of hidden unit in the hidden layers of the model

- Design of hidden units is an active research area that does not have many definitive guiding theoretical principles

- ReLU is an excellent default choice
- But there are many other types of hidden units available

- When to use which kind?
  - Impossible to predict in advance which will work best
  - Design process is trial and error

Is Differentiability necessary?

- Some hidden units are not differentiable at all input points
  - Most notably: Rectified Linear Function

$$g(z) = \max\{0, z\}$$

is not differentiable at $z = 0$

- May seem like it invalidates for use in gradient-based learning

- In practice gradient descent still performs well enough for these models to be used in ML tasks

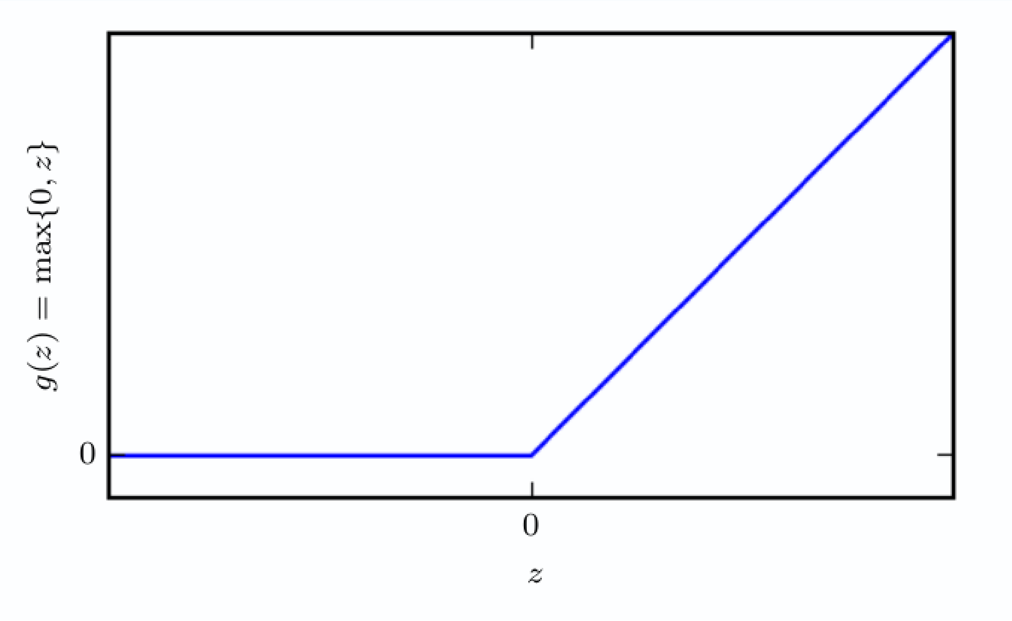
- Not differentiable hidden units are usually non-differentiable at only a small number of points

Left and Right Differentiability

- Function $g(x)$ has a left and right derivate:
  - defined by the slope Immediately left of x
  - defined by the slope Immediately right of x

- In case of $g(x) = \max\{0, z\}$ the left derivate at $z = 0$ is 0, the right derivate is 1.

- Software normally reports either of the values as derivate



The ReLU

$$\boldsymbol{h} = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b})$$
$$g(x) = \max\{0, x\}$$

- Good practice to set all elements of $\boldsymbol{b}$ to a small value such as 0.1
  - This makes it likely that ReLU will be initially active for most training samples and allow derivatives to pass through

- Generalizations of ReLU
  - Perform comparably to ReLU and occasionally perform better
  - ReLU cannot learn on examples for which the activation is zero
  - Generalizations guarantee that they receive gradient everywhere

Generalizations of ReLU

- Three methods based on using a non-zero slope α_i when $z_i < 0$:

$$h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

- Absolute-value rectification:
  - fixes $\alpha_i = -1$ to obtain $g(z) = |z|$

- Leaky ReLU:
  - fixes α_i to a small value like 0.01

- Parametric ReLU or PReLU
  - treats α_i as a learnable parameter

Maxout Units

- Maxout units further generalize ReLUs
- Instead of applying element-wise function $g(z)$, maxout units divide z into groups of k values
- Each maxout unit then outputs the maximum element of one of these groups:

$$g(\boldsymbol{z})_i = \max_{j \in G^{(i)}} z_j$$

with $G^{(i)}$ is the set of indices for group $i, \{(i-1)k+1, \dots, ik\}$

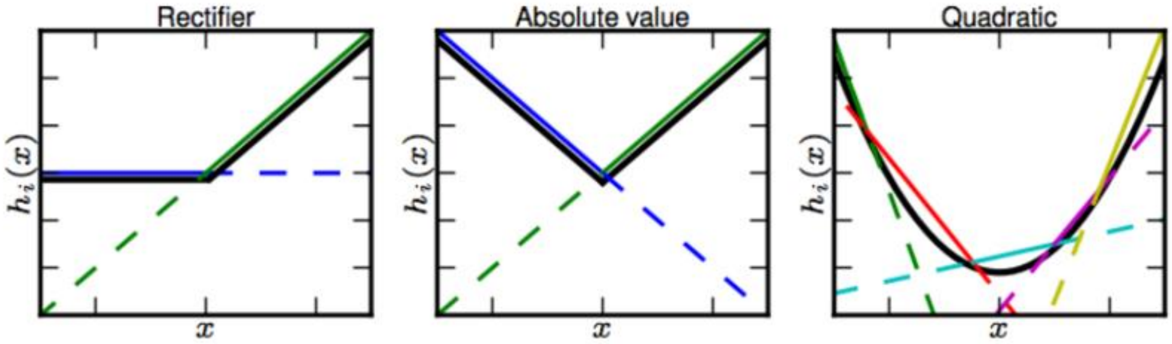
- A maxout unit can learn a piecewise linear, convex function with up to k pieces

Maxout

- Thus seen as **learning the activation function itself** rather than just the relationship between units

- With large enough k, approximate any convex function

- A maxout layer with two pieces can learn to implement the same function as a traditional layer using ReLU or its generalizations



Logistic Sigmoid

- Prior to introduction of ReLU, most neural networks used logistic sigmoid activation

$$g(z) = \sigma(z)$$

- Or the hyperbolic tangent

$$g(z) = \tanh(z)$$

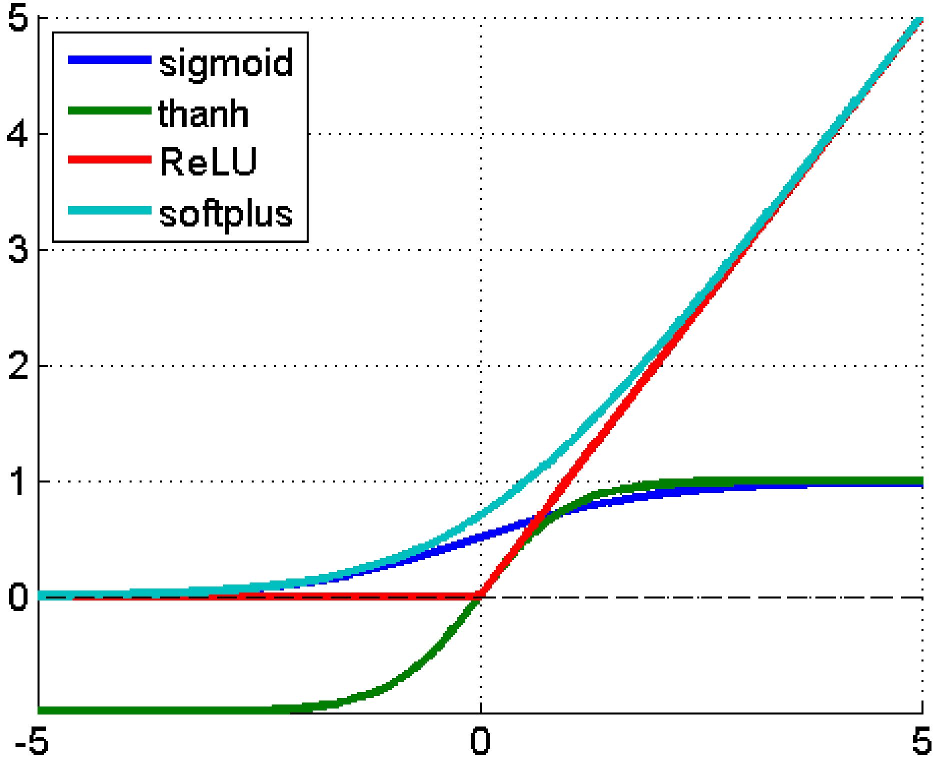
- These activation functions are closely related because

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

- Sigmoid units are used to predict probability that a binary variable is 1

Sigmoid Saturation

- Sigmoidals saturate across most of domain
  - Saturate to 1 when z is very positive and 0 when z is very negative
  - Strongly sensitive to input when z is near 0
  - Saturation makes gradient-learning difficult



- ReLU and Softplus increase for input >0

Sigmoid vs $tanh$ Activation

- Hyperbolic tangent typically performs better than logistic sigmoid

- It resembles the identity function more closely

$$\tanh(0) = 0 \quad \text{while} \quad \sigma(0) = 1 \,/\, 2$$

- Because tanh is similar to identity near 0, training a deep neural network

$$\hat{\boldsymbol{y}} \;=\; \boldsymbol{w}^T \tanh(\boldsymbol{U}^T \tanh(\boldsymbol{V}^T \boldsymbol{x}))$$

resembles training a linear model

$$\hat{\boldsymbol{y}} = \boldsymbol{w}^T \boldsymbol{U}^T \boldsymbol{V}^T \boldsymbol{x}$$

so long as the activations can be kept small

They are still useful

- Sigmoidal more common in settings other than feed-forward networks

- Recurrent networks, many probabilistic models and autoencoders have additional requirements that rule out piecewise linear activation functions

- They make sigmoid units appealing despite saturation

Other Output Units

- **Cosine**

$$\boldsymbol{h} = \cos(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

  - Feedforward networks on MNIST obtained error rate of less than 1%

- **Radial Basis**

$$h_i = \exp\left(-\frac{1}{\sigma^2}\left\|\boldsymbol{W}_{:j} - \boldsymbol{x}\right\|^2\right)$$

  - Becomes more active as $\boldsymbol{x}$ approaches a template $W_{:,i}$

- **Softplus**

$$g(a) = \zeta(a) = \log(1 + e^a)$$

  - Smooth version of the rectifier

- **Hard tanh**

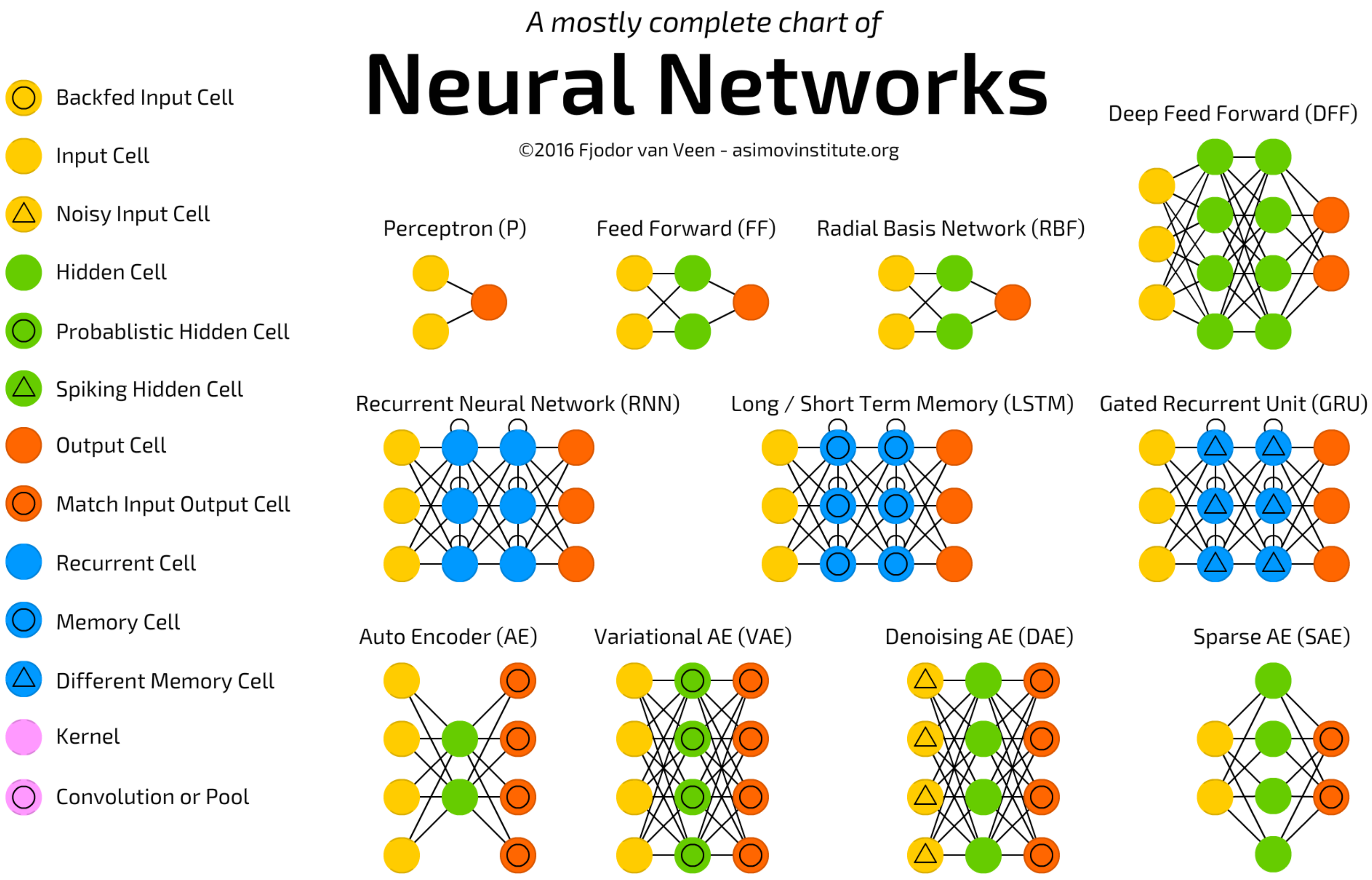
$$g(a) \ = \ \max(-1, \min(1, a))$$

  - Shaped similar to tanh and the rectifier but it is bounded

Deep Feedforward Networks

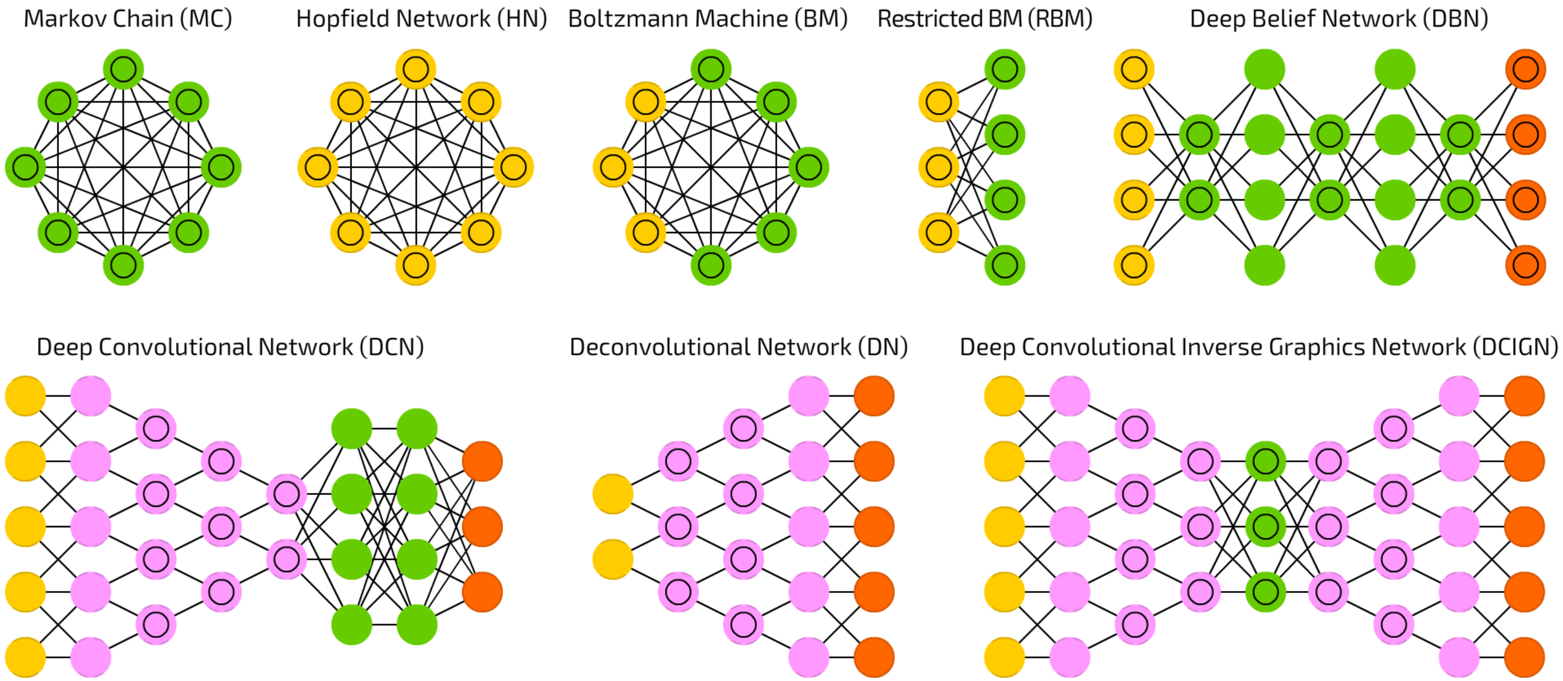
- **PART I**
  - Feedforward Networks
  - Output Units
  - Hidden Units
  - **Architecture Design**
- **PART II**
  - Gradient-Based Learning
  - Backpropagation

A mostly complete chart of Neural Networks



- https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464

A mostly complete chart of Neural Networks



Markov Chain (MC)

Hopfield Network (HN)

Boltzmann Machine (BM)

Restricted BM (RBM)

Deep Belief Network (DBN)

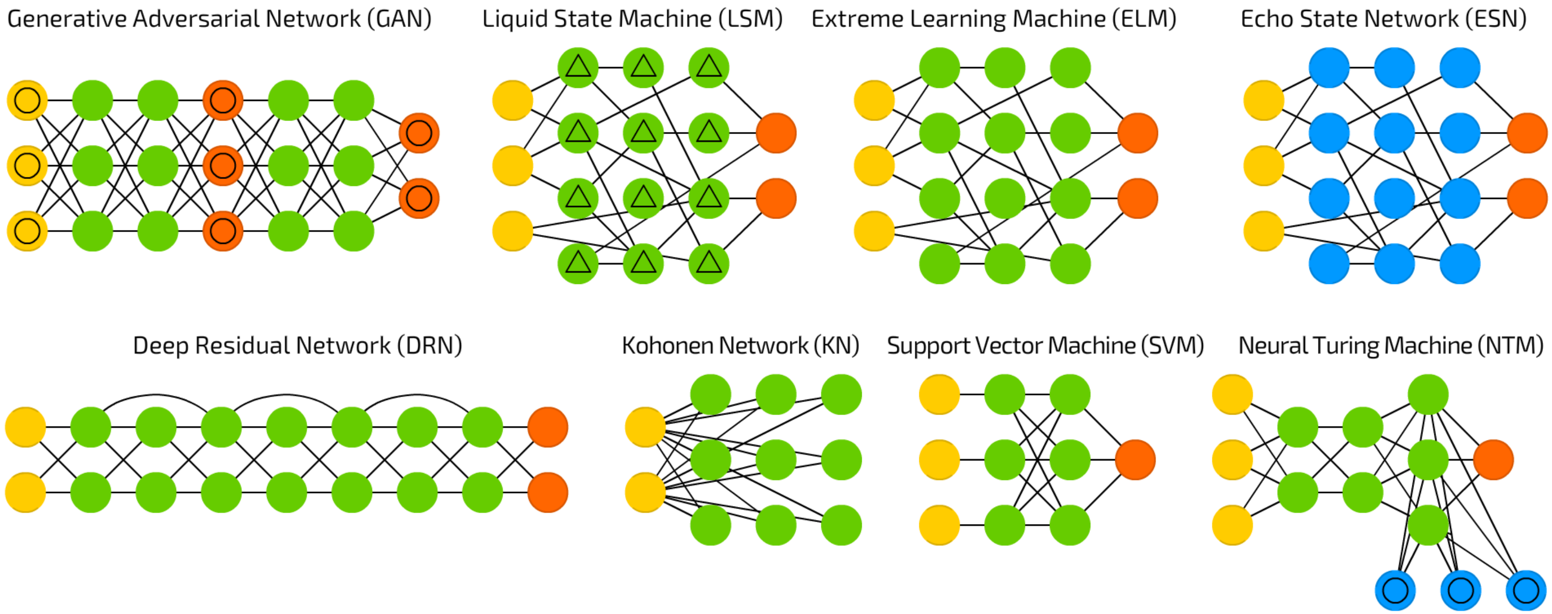
Deep Convolutional Network (DCN)

Deconvolutional Network (DN)

Deep Convolutional Inverse Graphics Network (DCIGN)

- https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464

A mostly complete chart of Neural Networks


Generative Adversarial Network (GAN)
Liquid State Machine (LSM)
Extreme Learning Machine (ELM)
Echo State Network (ESN)
Deep Residual Network (DRN)
Kohonen Network (KN)
Support Vector Machine (SVM)
Neural Turing Machine (NTM)

- https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464

Main Architectural Considerations

1. **Choice of depth of network**
2. **Choice of width of each layer**

- Deeper networks have
  - Far fewer units in each layer
  - Far fewer parameters
  - Often generalize well to the test set
  - But are often more difficult to optimize

- Ideal network architecture must be found via experimentation guided by validation set error

On the Power of Neural Networks

- A linear model, mapping from features to outputs via matrix multiplication, can by definition represent only linear functions.
  - Easy to learn
  - Cost function results in convex optimization problem

The **universal approximation theorem** states that a feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function and enough units (e.g. sigmoid) can approximate any Borel measurable function (with some restrictions).

- We don't need specialized model families for approximating functions
- However, it does not state how to learn and how to design the appropriate network

Implication of Theorem

- Whatever function we are trying to learn, a large MLP will be able to represent it

- However we are not guaranteed that the training algorithm will learn this function
  - Optimizing algorithms may not find the parameters
  - May choose wrong function due to over-fitting

- No Free Lunch: There is no universal procedure for examining a training set of samples and choosing a function that will generalize to points not in training set

Function Families and Depth

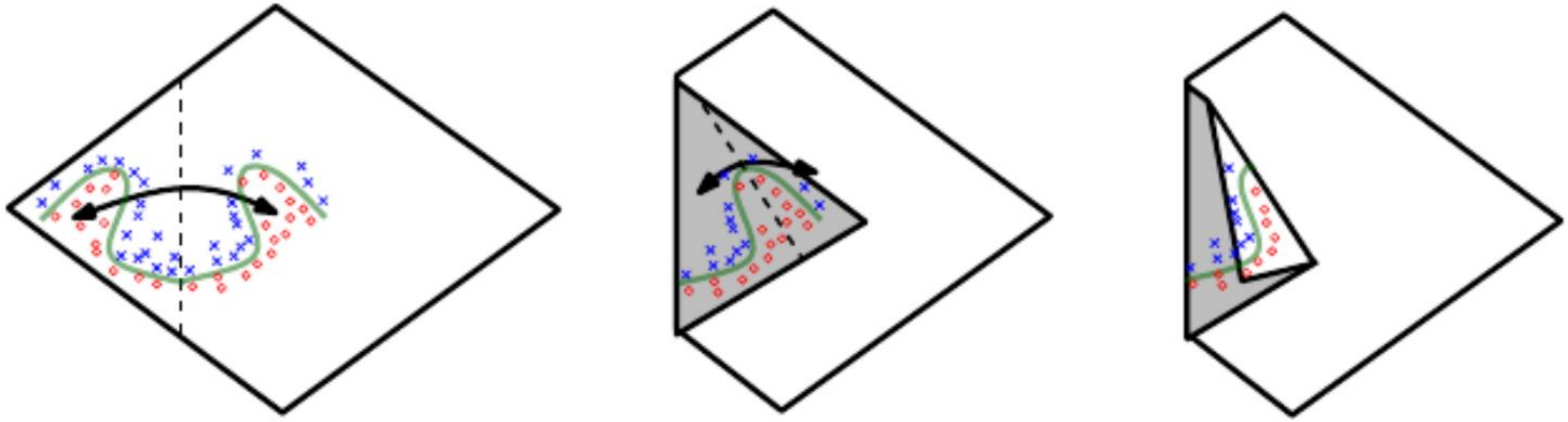
- Some families of functions can be represented efficiently in a deep network but require much larger networks if the model is shallow

- In some cases no. of hidden units required by shallow model is exponential in n

- Piecewise linear networks (which can be obtained from rectifier nonlinearities or maxout units) can represent functions with a no. of regions that is exponential in d

Advantage of deeper networks

- Absolute value rectification creates mirror images of function computed on top of some hidden unit, wrt the input of that hidden unit.
- Each hidden unit specifies where to fold the input space in order to create mirror responses.
- By composing these folding operations we obtain an exponentially large no. of piecewise linear regions which can capture all kinds of repeating patterns



Intuition of Depth

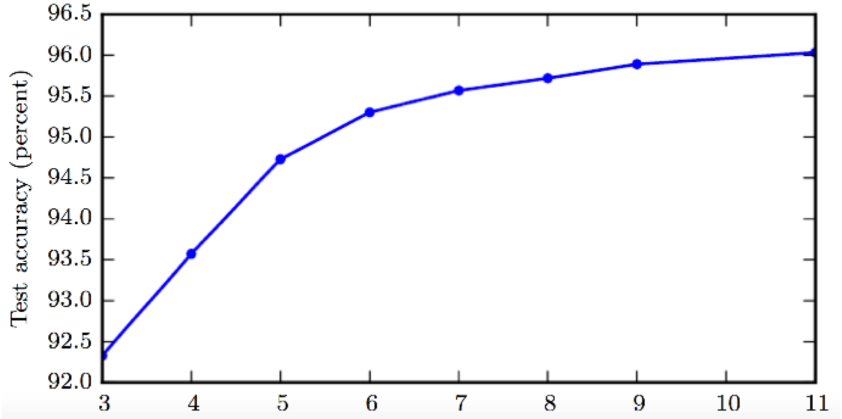
- Any time we choose a ML algorithm we are implicitly stating a set of beliefs about what kind of functions that algorithm should learn

- Choosing a deep model encodes a belief that the function should be a composition several simpler functions

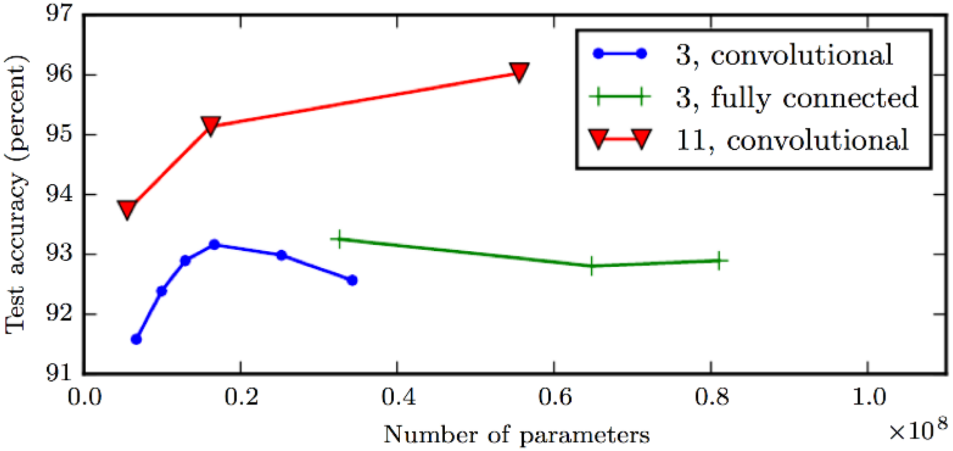
- The learning problem consists of discovering a set of underlying factors of variation that can in turn be described in terms of other, simpler underlying factors of variation.

- Empirically: Deeper Networks perform better ☺

Deeper Networks



Test accuracy consistently increases with depth



Increasing parameters without increasing depth is not as effective

Other architectural considerations

- Most important specialized architectures are

- Convolutional Networks
  - Used heavily for computer vision

- Recurrent Neural Networks
  - Used for sequence processing
  - Have their own architectural considerations

- Non-chain architectures
  - Skipping going from layer i to layer i+2 or higher