Machine Learning

Session 10 Deep Learning

- Introduction
- Feed-Forward Neural Networks
- Training (backpropagation)

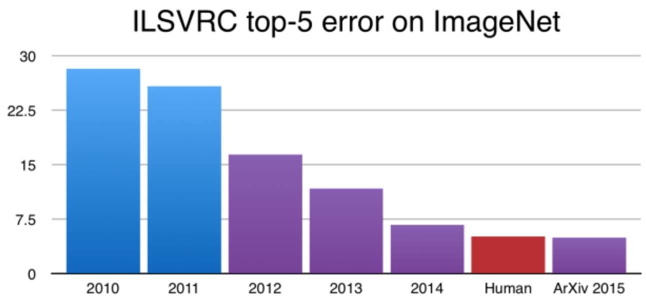
Chap 5 of C. Bishop book
DL book and slides from I. Goodfellow

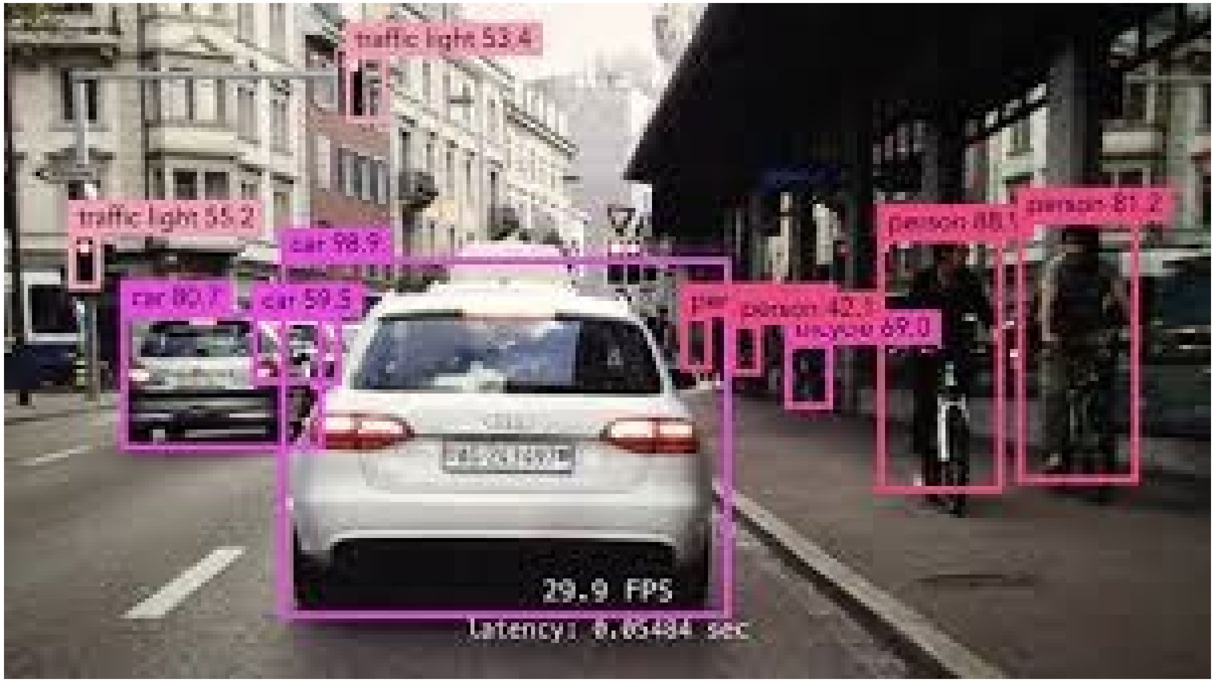
Deep Learning: Introduction

- **Major achievements:** **image Recognition, Text generation, language translation, Game Playing**

Image recognition

- ImageNet: ~14 million hand-annotated images, 20,000 categories,
  2012, major success with **CNN (convolutional Neural Networks)**
  A standard in industry: autonomous cars, drones, …
- Large Visual Recognition Challenge (ILSVRC) editions:2010-2017





- 2018 up to now: classifying 3D objects using natural language: for robotics and Virtual Reality
- And much more..

Deep Learning: Introduction

- **Major achievements**

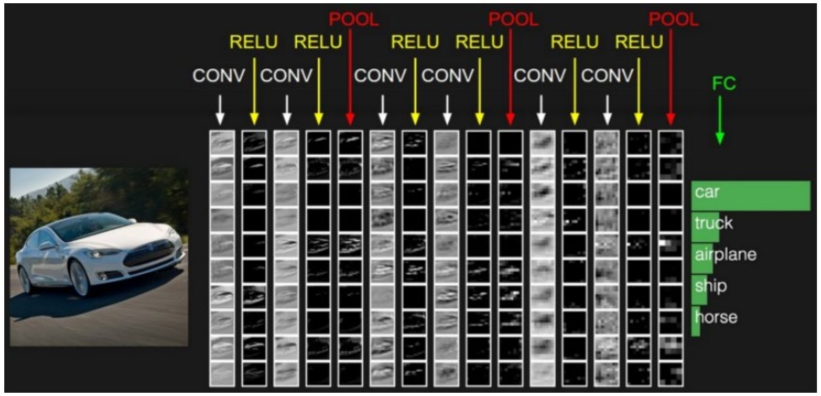
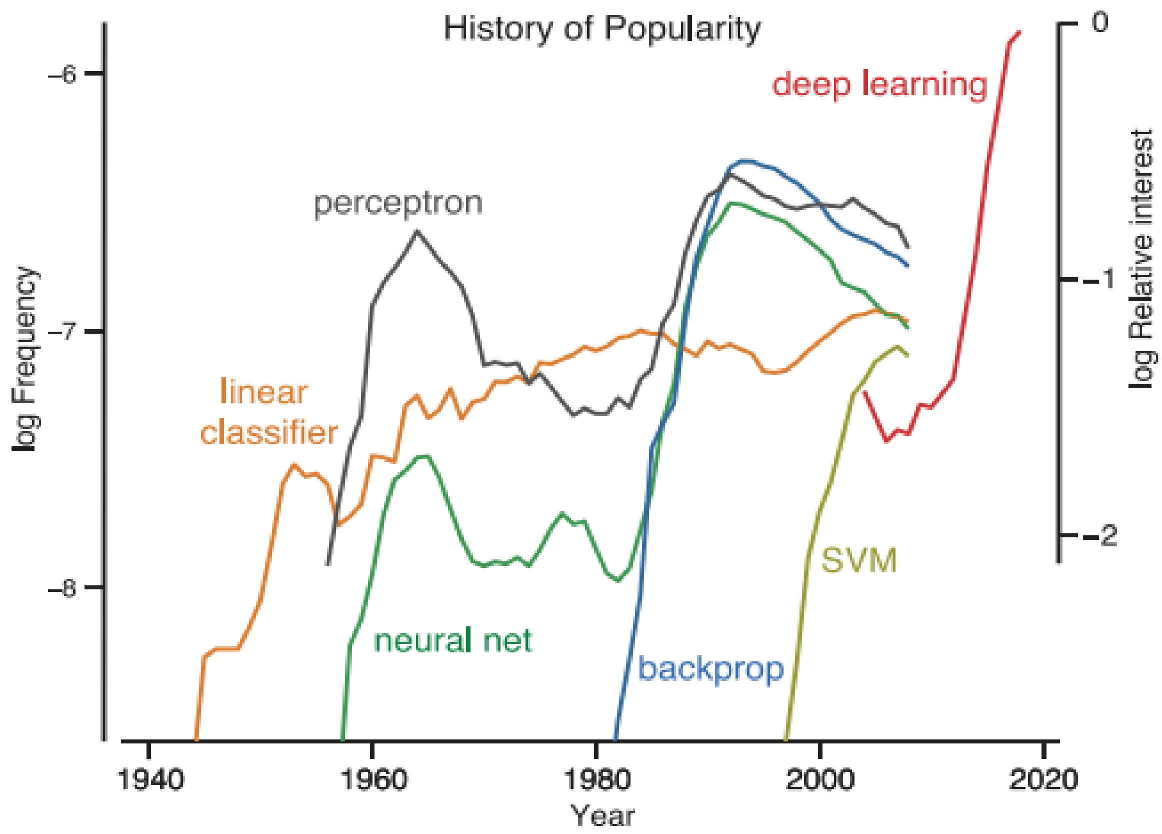


Image recognition

- ImageNet: ~14 million hand-annotated images,
  20,000 categories,
  2012, major success
  with **CNN (convolutional Neural Networks)**
  A standard in industry: autonomous cars, drones, …

Deep Learning: Introduction

Journal of Vision (2018) 18(13):2, 1–13 Majaj & Pelli

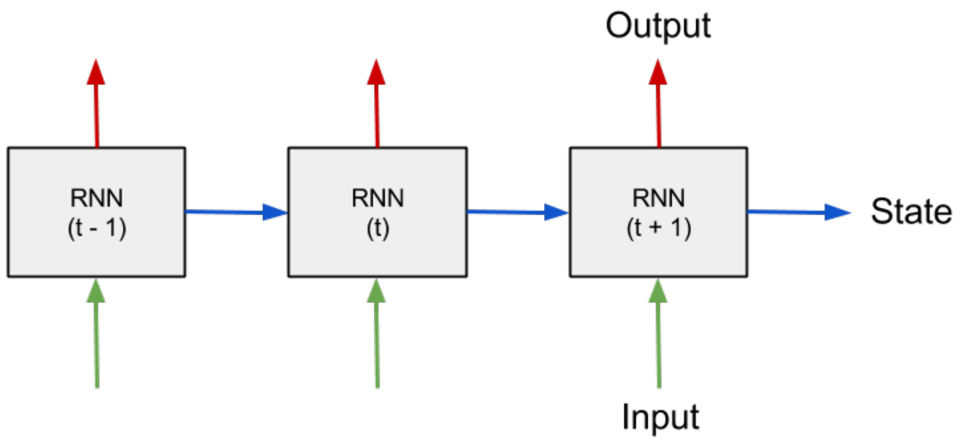


Deep Learning: Introduction

- **Major achievements**

Text Generation

- Predicts next characters/words from large sequences of text
- Output is the shifted input text
- Uses Another type of model : Recurrent Neural Networks (RNNs)

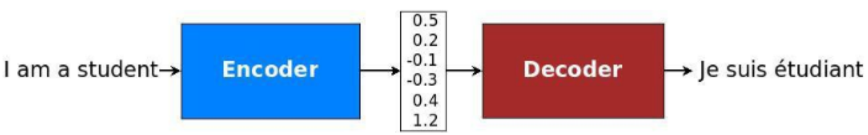


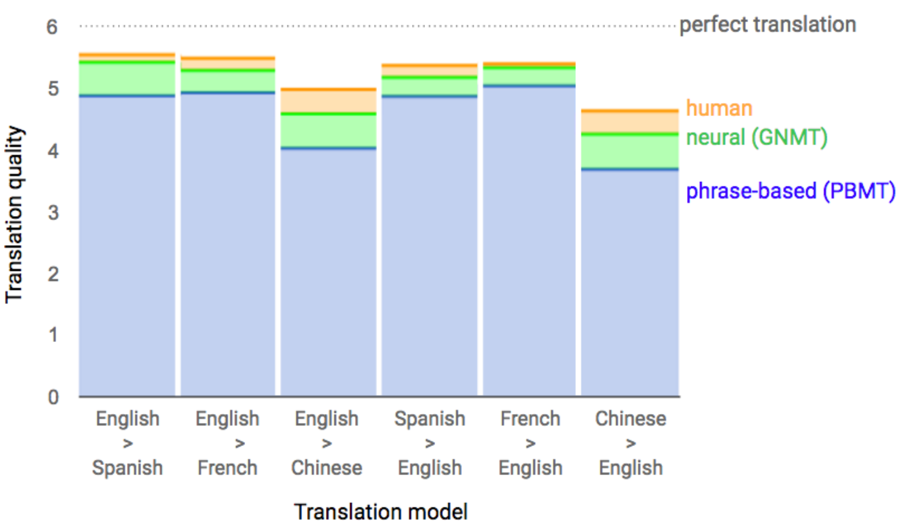
Deep Learning: Introduction

- **Major achievements**

Language translation

- Learns from pairs of sentences in two languages
- Output is the shifted input text
- Also uses RNNs



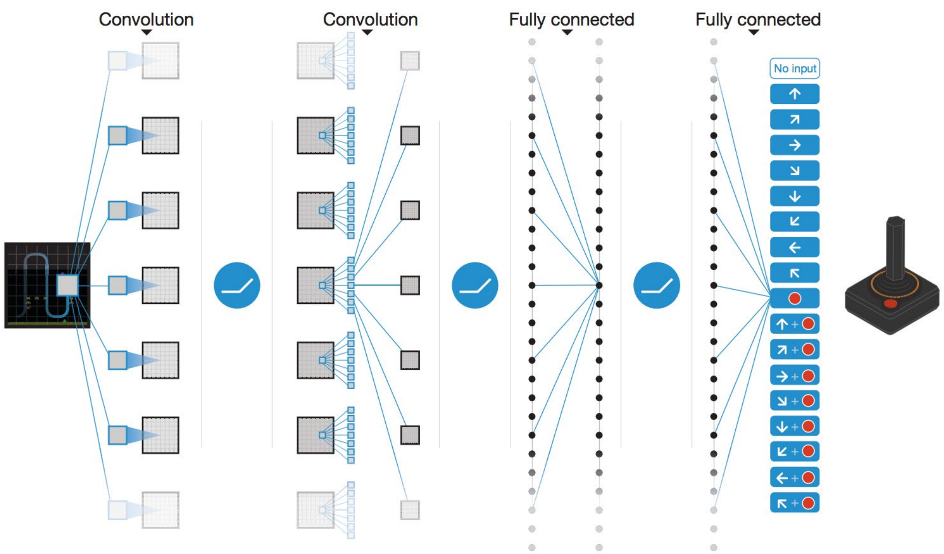


Deep Learning: Introduction

- **Major achievements**

Game playing

- Combined with Reinforcement Learning
- Learns how to play Atari videogames

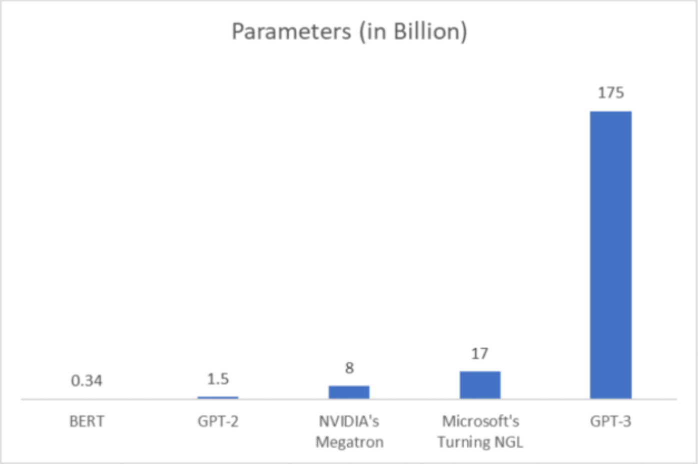


Deep Learning: Introduction

- **Major achievements**

Language Models

- OpenAI GPT-3 Language Model.
- 175 Billion parameters.
- Many use cases: coding, poetry, blogging, news articles, chatbots
- Controversial discussions
- Remained closed (private API) until November 18, 2021.



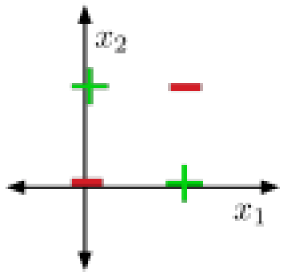
Deep Learning: Introduction

Feedforward Networks

Limits of linear classifiers

- The exclusive-OR (XOR) is a non-linearly separable function

x_1	x_2	XOR
0	0	0
1	0	1
0	1	1
1	1	0



- We visited this problem using SVMs and showed how projecting the inputs in a higher dimensional feature space could solve the problem

- Now we will take a **compositional** approach.
  We can compose simple linear functions to represent a complex one.
  How?

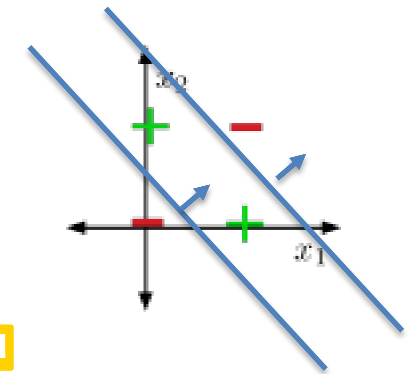
Limits of linear classifiers

- The exclusive-OR (XOR) is a non-linearly separable function

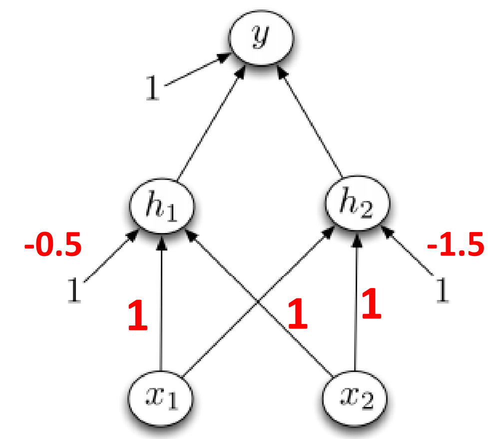
x_1 x_2	XOR	h_1	h_2	
0 0	0	0	0	
1 0	1	1	0	
0 1	1	1	0	
1 1	0	1	1	

- Composition of linear classifiers followed with a **non-linearity** $\mathrm{Th}(\cdot)$

$$x_1 + x_2 - 1.5 = 0$$



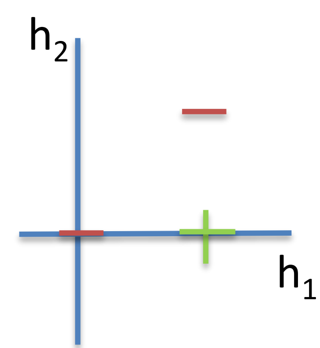
$$x_1 + x_2 - 0.5 = 0$$



$$\mathrm{Th}(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$h_1 = \mathrm{Th}\left((1,1,-0.5) \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} \right)$$

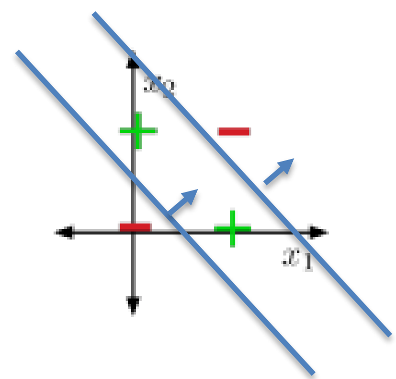
$$h_2 = \mathrm{Th}\left((1,1,-1.5) \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} \right)$$



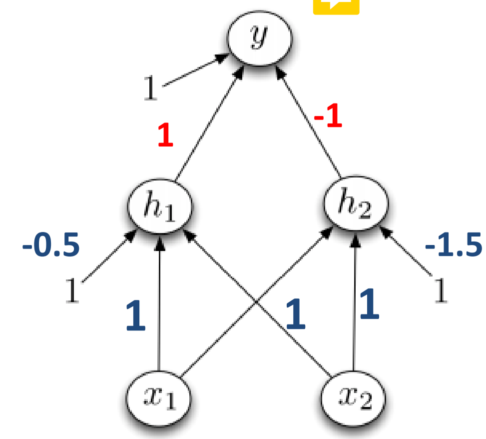
Limits of linear classifiers

- The exclusive-OR (XOR) is a non-linearly separable function

$x_1 + x_2 - 1.5 = 0$



$x_1 + x_2 - 0.5 = 0$



x_1	x_2	XOR	h_1	h_2	y
0	0	0	0	0	**0**
1	0	1	1	0	**1**
0	1	1	1	0	**1**
1	1	0	1	1	**0**

- Composition of linear classifiers followed with a **non-linearity** $\mathrm{Th}(\cdot)$

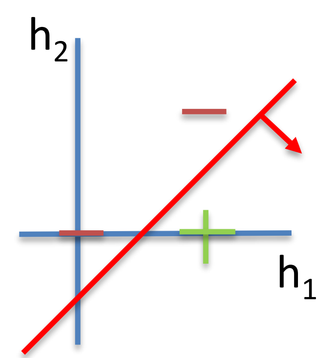
- In this new representation (h_1, h_2) the points are linearly separable

$$y = (1, -1, -0.5)\begin{pmatrix} h_1 \\ h_2 \\ 1 \end{pmatrix}$$

$$\mathrm{Th}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$h_1 = \mathrm{Th}\left((1,1,-0.5)\begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}\right)$$

$$h_2 = \mathrm{Th}\left((1,1,-1.5)\begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}\right)$$



Deep Learning inspiration: The Brain

- Our Brain has $\sim 10^{11}$ neurons, each of which communicates (is connected) to $\sim 10^{4}$ neurons

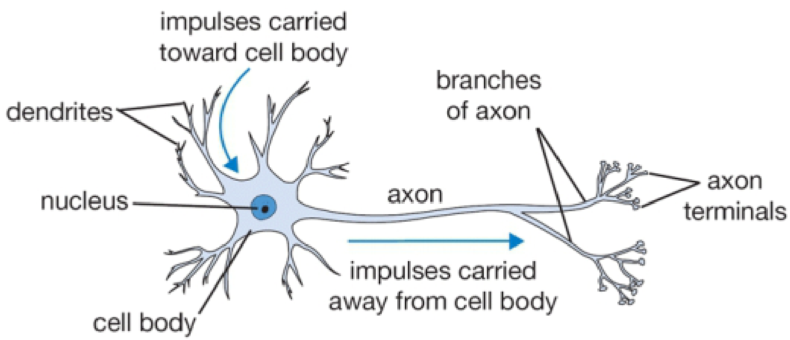
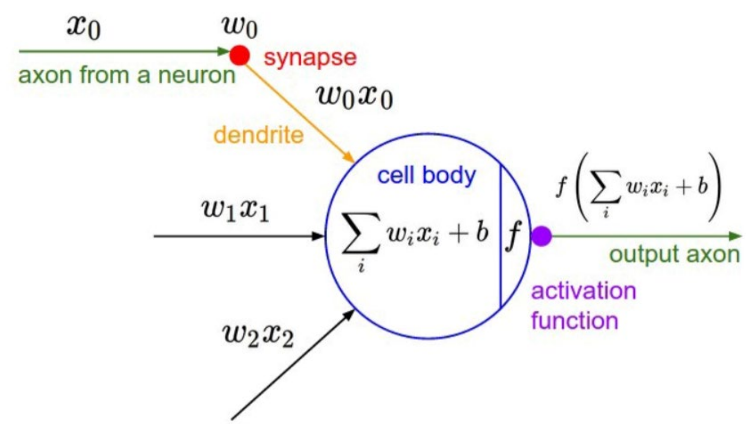


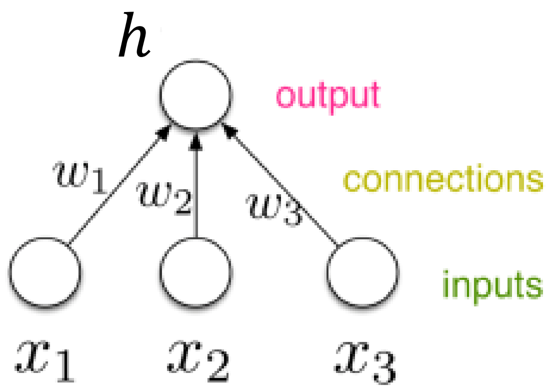
Figure: The basic computational unit of the brain: Neuron

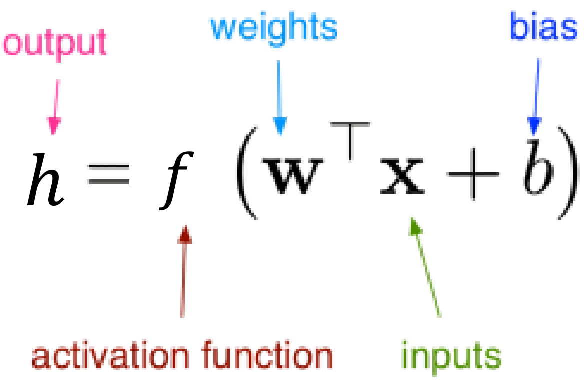
[Pic credit: http://cs231n.github.io/neural-networks-1/]



Deep Learning inspiration: The Brain

- For Neural Nets, we use a much simpler model, neuron or unit:



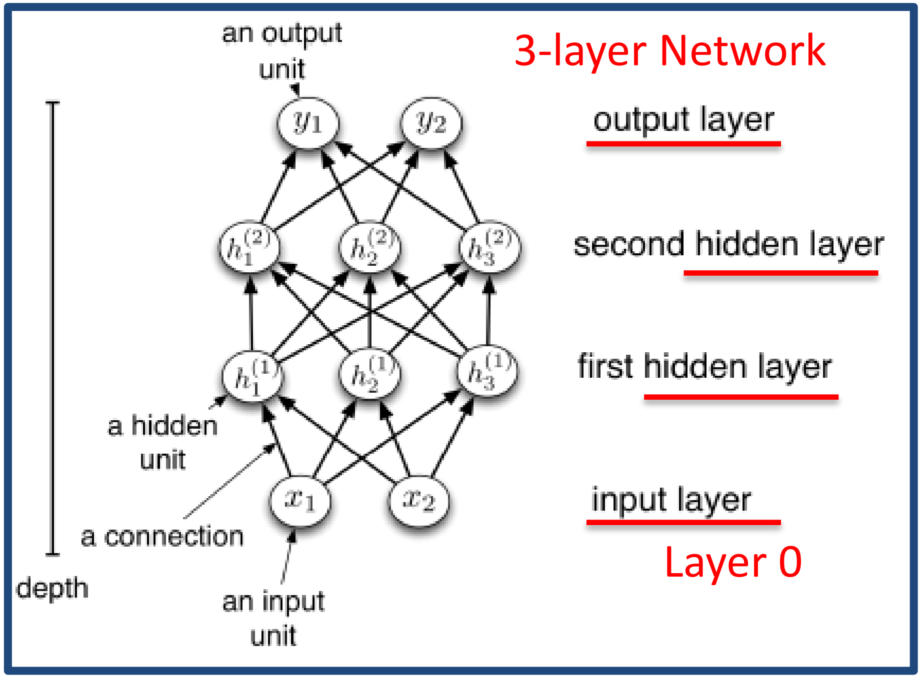


- Compare with logistic regression

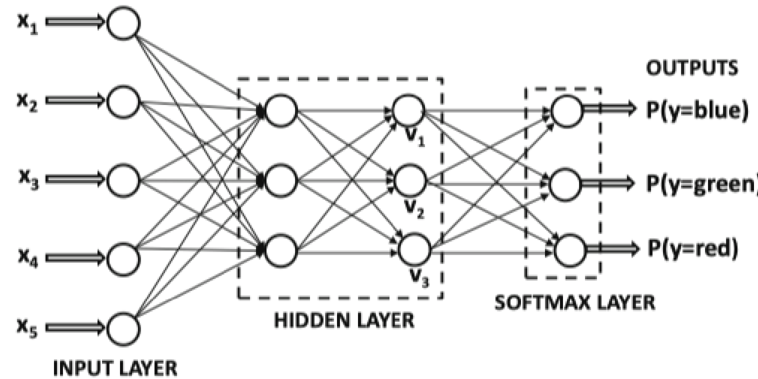
$$h_{\mathbf{w}}(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + w_0)$$

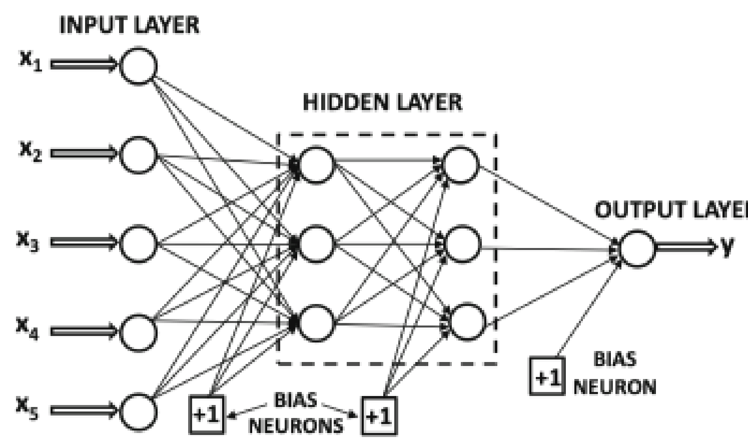
Using lots of these simplistic neuron like processing units, we can do some powerful computations!

Deep Learning: Multilayer Perceptrons



Other representations

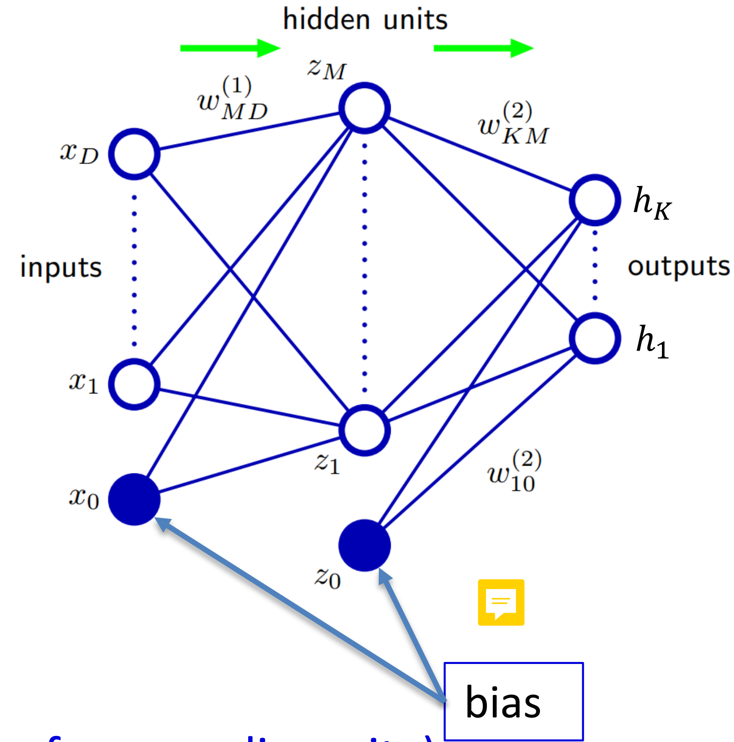




- We connect many units, grouped into layers
- This gives a neural network
- When all input units are connected to all output units we call this fully connected layer
- Constraint: No feedback (loop) connections

Deep Learning

Feedforward multi-layer Networks

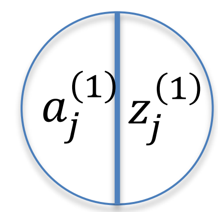


- Example: (1 input layer, 1 hidden layer, and 1 output layer with K outputs)

$w_{kj}^{(2)}$ (layer number) j source unit k target unit

$$a_j^{(1)} = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$ activation (before non-linearity)

$$z_j^{(1)} = \sigma\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right)$$ hidden output

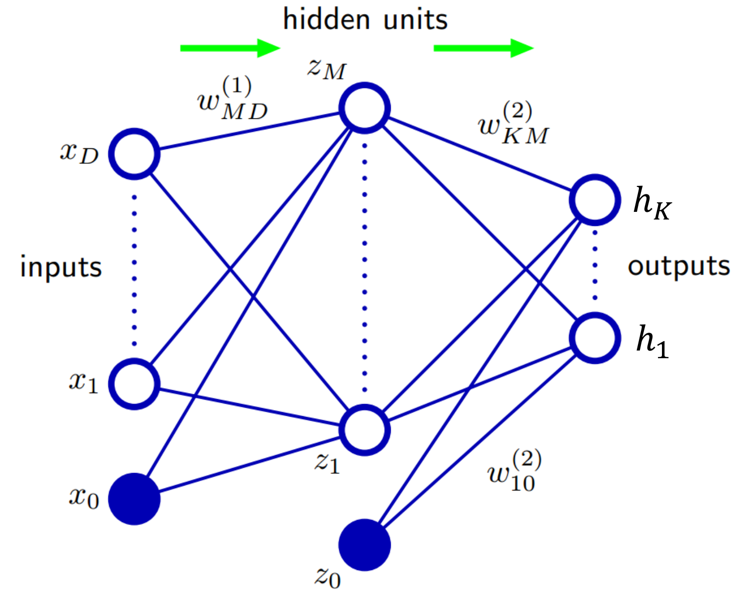


- Activation function $\sigma(\cdot)$ **must be nonlinear,** Otherwise, an equivalent single-layer network exists
- Forward pass: evaluation of $h_k(\mathbf{x})$
- $E(\mathbf{w})$ Error function : difference between desired and actual outputs

Deep Learning

Feedforward multi-layer Networks

- Example (1 hidden layer, 1 input layer and one utput layer with K units)



Full network

bias

$$h_k(\mathbf{x}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} \underbrace{\sigma\left(\overbrace{\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}}^{a_j^{[1]}}\right)}_{z_j^{[1]}} + w_{k0}^{(2)}\right)$$

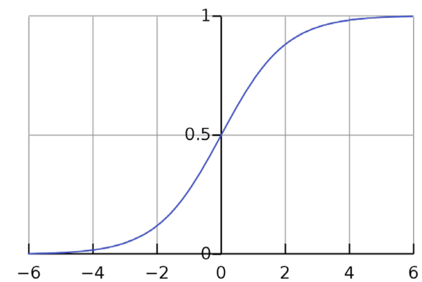
- Activation function $\sigma(\cdot)$ **must be nonlinear,** Otherwise, an equivalent single-layer network exists
- Forward pass: evaluation of $h_k(\mathbf{x})$
- $E(\mathbf{w})$ Error function : difference between desired and actual outputs

Deep Learning

Feedforward multi-layer Networks

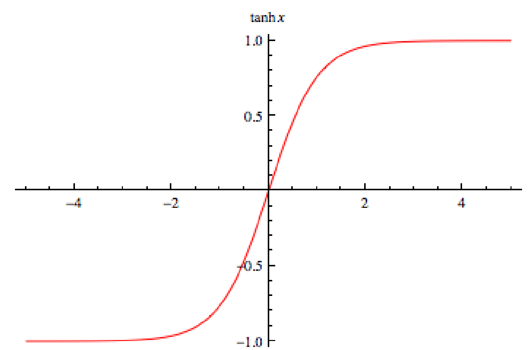
- Examples of activation functions
  - Logistic sigmoid

$$\sigma(a) = \frac{1}{1+\exp(-a)} \quad \text{derivative: } \sigma'(a)=\sigma(a)(1-\sigma(a))$$



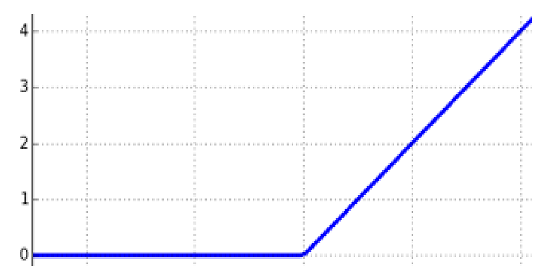
  - Tanh

$$\tanh(a) = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}} \quad \text{derivative: } \tanh'(a)=1-\tanh(a)^2$$



  - Rectified Linear Unit (ReLU)

$$\sigma(a) = \max\{0, a\}$$



- ReLU is mostly used nowadays. Why?
  - Discontinuity at zero is not an actual problem numerically
  - Gradient information does not saturate

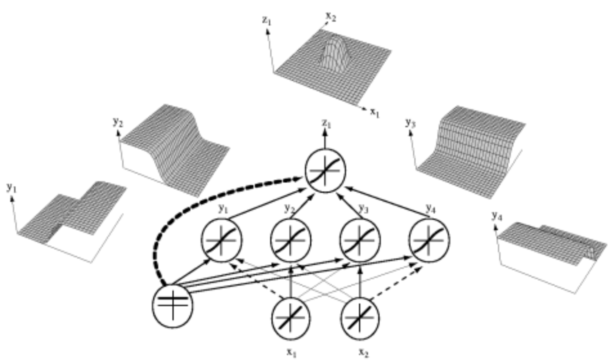
Deep Learning

Feedforward multi-layer Networks

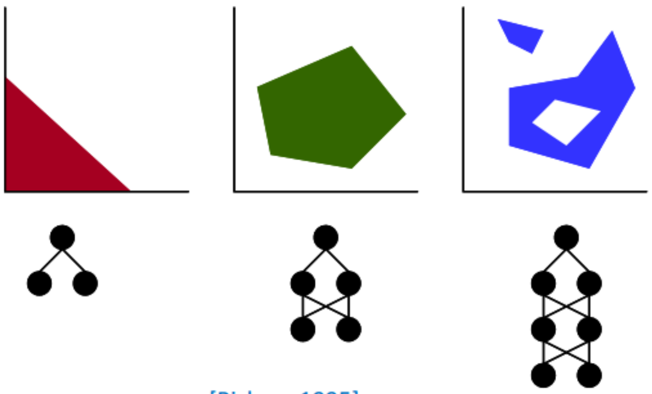
- **Universal Approximator Theorem:**

  One hidden layer is enough to represent (not learn) an approximation of any function to an arbitrary degree of accuracy

- Intuition: four hidden units can produce a *bump* at the output. A large number of *bumps* can approximate any surface.



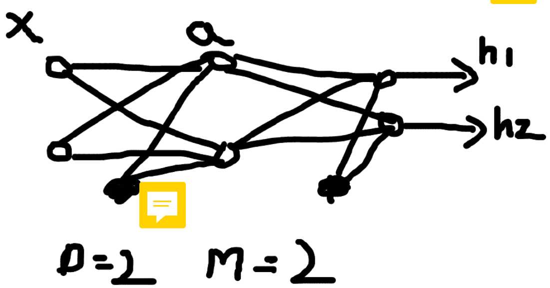
[Duda, Hart and Stork, 2001]



[Bishop, 1995]

Example

a) Draw the architecture of a neural network with 2 inputs, 1 hidden layer with 2 units, and one output layer with 2 units. Use sigmoid as the activation function for all units and the following weight matrices:



$$\mathbf{W}^{(1)} = \begin{pmatrix} 0.25 & 0.30 \\ 0.15 & 0.20 \end{pmatrix}, \qquad \mathbf{b}^{(1)} = \begin{pmatrix} 0.35 \\ 0.35 \end{pmatrix}$$

$$\mathbf{W}^{(2)} = \begin{pmatrix} 0.45 & 0.40 \\ 0.50 & 0.55 \end{pmatrix}, \qquad \mathbf{b}^{(2)} = \begin{pmatrix} 0.60 \\ 0.60 \end{pmatrix}$$

For a given data point consisting of input $\mathbf{x}$ and output $\mathbf{y}$

$$\mathbf{x} = \begin{pmatrix} 0.05 \\ 0.10 \end{pmatrix}, \qquad \mathbf{y} = \begin{pmatrix} 0.01 \\ 0.99 \end{pmatrix}$$

b) Compute the forward pass (output $h_{\mathbf{W}}(\mathbf{x})$) of the neural network model

c) Compute the error as the squared norm of $h_{\mathbf{W}}(\mathbf{x}) - \mathbf{y}$

Example

For a given data point consisting of input $\mathbf{x}$ and output $\mathbf{y}$

$$\mathbf{x} = \begin{pmatrix} 0.05 \\ 0.10 \end{pmatrix}, \qquad \mathbf{y} = \begin{pmatrix} 0.01 \\ 0.99 \end{pmatrix}$$

b) Compute the forward pass (output $h_{\mathbf{W}}(\mathbf{x})$) of the neural network model:

$$\mathbf{a}^{(1)} = \mathbf{W}^{(1)} \cdot \mathbf{x} + \mathbf{b}^{(1)} = \begin{pmatrix} 0.25 & 0.30 \\ 0.15 & 0.20 \end{pmatrix} \cdot \begin{pmatrix} 0.05 \\ 0.10 \end{pmatrix} + \begin{pmatrix} 0.35 \\ 0.35 \end{pmatrix} = \begin{pmatrix} 0.39 \\ 0.38 \end{pmatrix}$$

$$\mathbf{z}^{(1)} = \sigma\left(\mathbf{a}^{(1)}\right) = \frac{1}{1 + e^{-\mathbf{a}^{(1)}}} = \begin{pmatrix} 0.60 \\ 0.59 \end{pmatrix}$$

Example

For a given data point consisting of input $\mathbf{x}$ and output $\mathbf{y}$

$$\mathbf{x} = \begin{pmatrix} 0.05 \\ 0.10 \end{pmatrix}, \qquad \mathbf{y} = \begin{pmatrix} 0.01 \\ 0.99 \end{pmatrix}$$

b) Compute the forward pass (output $h_{\mathbf{W}}(\mathbf{x})$) of the neural network model:

$$\mathbf{a}^{(1)} = \mathbf{W}^{(1)} \cdot \mathbf{x} + \mathbf{b}^{(1)} = \begin{pmatrix} 0.25 & 0.30 \\ 0.15 & 0.20 \end{pmatrix} \cdot \begin{pmatrix} 0.05 \\ 0.10 \end{pmatrix} + \begin{pmatrix} 0.35 \\ 0.35 \end{pmatrix} = \begin{pmatrix} 0.39 \\ 0.38 \end{pmatrix}$$

$$\mathbf{z}^{(1)} = \sigma\left(\mathbf{a}^{(1)}\right) = \frac{1}{1 + e^{-\mathbf{a}^{(1)}}} = \begin{pmatrix} 0.60 \\ 0.59 \end{pmatrix}$$

$$\mathbf{a}^{(2)} = \mathbf{W}^{(2)} \cdot \mathbf{z}^{(1)} + \mathbf{b}^{(2)} = \begin{pmatrix} 0.45 & 0.40 \\ 0.50 & 0.55 \end{pmatrix} \cdot \begin{pmatrix} 0.60 \\ 0.59 \end{pmatrix} + \begin{pmatrix} 0.60 \\ 0.60 \end{pmatrix} = \begin{pmatrix} 1.11 \\ 1.22 \end{pmatrix}$$

$$\mathbf{h} = \sigma\left(\mathbf{a}^{(2)}\right) = \frac{1}{1 + e^{-\mathbf{a}^{(1)}}} = \begin{pmatrix} 0.75 \\ 0.77 \end{pmatrix}$$

Example

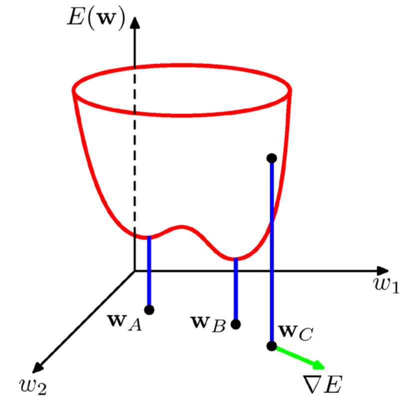
For a given data point consisting of input $\mathbf{x}$ and output $\mathbf{y}$

$$\mathbf{x} = \begin{pmatrix} 0.05 \\ 0.10 \end{pmatrix}, \qquad \mathbf{y} = \begin{pmatrix} 0.01 \\ 0.99 \end{pmatrix}$$

c) Compute the error as the squared norm of $h_{\mathbf{W}}(\mathbf{x}) - \mathbf{y}$

$$|(h_{\mathbf{W}}(\mathbf{x}) - \mathbf{y})|^2 = \left|\begin{pmatrix} 0.04 \\ -0.89 \end{pmatrix}\right|^2 = 0.79$$

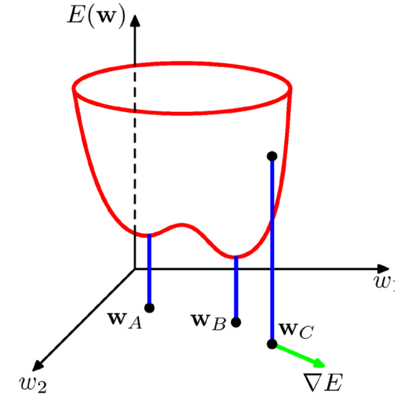
Deep Learning



Network training

- The role of the NN is to learn a mapping from $\mathbf{x}$ to outputs across the weight parameters $\mathbf{W} = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \ldots\}$ of the NN

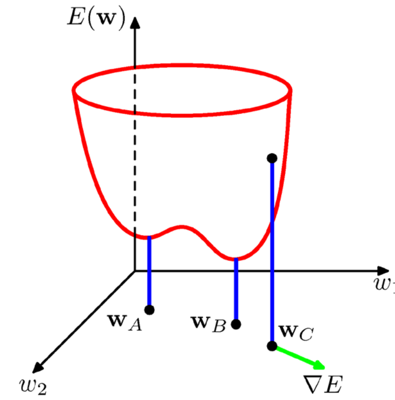
Deep Learning



Network training

- The role of the NN is to learn a mapping from $\mathbf{x}$ to outputs across the weight parameters $\mathbf{W} = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \ldots\}$ of the NN
- We need the $\mathbf{W}$ that minimizes the error function
  - No closed-form solution exists.
  - For very large models, only first-order (gradient-based) methods are feasible

Deep Learning

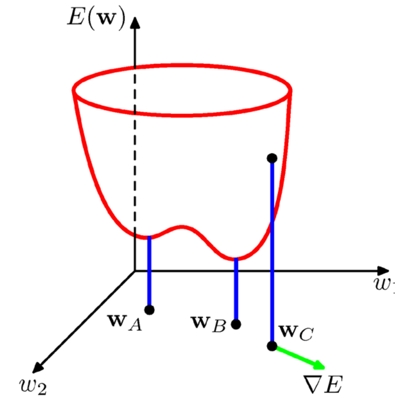


Network training

- The role of the NN is to learn a mapping from $\mathbf{x}$ to outputs across the weight parameters $\mathbf{W} = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \dots\}$ of the NN
- We need the $\mathbf{W}$ that minimizes the error function
  - No closed-form solution exists.
  - For very large models, only first-order (gradient-based) methods are feasible
- **In practice**: **stochastic gradient descent** starting from a smart initialization $\mathbf{W}^{[0]}$

$$\mathbf{W}^{[t+1]} = \mathbf{W}^{[t]} - \eta \, \nabla_{\mathbf{W}} E\left(\mathbf{W}^{[t]}\right)$$

Deep Learning



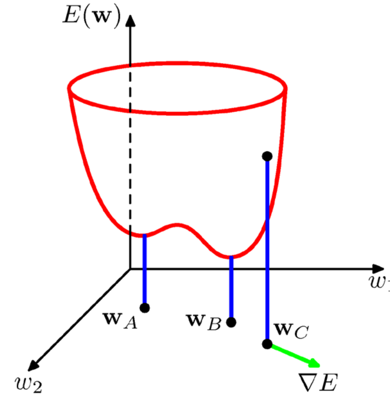
Network training

- The role of the NN is to learn a mapping from $\mathbf{x}$ to outputs across the weight parameters $\mathbf{W} = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \ldots\}$ of the NN
- We need the $\mathbf{W}$ that minimizes the error function
  - No closed-form solution exists.
  - For very large models, only first-order (gradient-based) methods are feasible
- **In practice**: **stochastic gradient descent** starting from a smart initialization $\mathbf{W}^{[0]}$

$$\mathbf{W}^{[t+1]} = \mathbf{W}^{[t]} - \eta \nabla_{\mathbf{W}} E\left(\mathbf{W}^{[t]}\right)$$

- Two steps for each iteration:
  1. Forward step: from the input layer to the outputs and calculating $E\left(\mathbf{W}^{[t]}\right)$
  2. Error Backpropagation: recalculate $\mathbf{W}$ to decrease the Error

Deep Learning



Network training

- The role of the NN is to learn a mapping from $\mathbf{x}$ to outputs across the weight parameters $\mathbf{W} = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \dots\}$ of the NN
- We need the $\mathbf{W}$ that minimizes the error function
  - No closed-form solution exists.
  - For very large models, only first-order (gradient-based) methods are feasible
- **In practice**: **stochastic gradient descent** starting from a smart initialization $\mathbf{W}^{[0]}$

$$\mathbf{W}^{[t+1]} = \mathbf{W}^{[t]} - \eta \nabla_{\mathbf{W}} E\left(\mathbf{W}^{[t]}\right)$$

- Two steps for each iteration:
  1. Forward step: from the input layer to the outputs and calculating $E\left(\mathbf{W}^{[t]}\right)$
  2. Error Backpropagation: recalculate $\mathbf{W}$ to decrease the Error

- In practice:
  - Shuffle all data points
  - Select batches of data points and update parameters in parallel for each batch
  - Run for several epochs (an epoch is one full pass over the training data)

Backpropagation

- A method to **compute the gradient** vector $\nabla_{\mathbf{W}} E(\mathbf{W})$ efficiently

- Just like the forward pass *forward propagates* computations to obtain output for a given the input, this algorithm ***backward propagates*** computations to obtain the gradients with respect to every parameter

Backpropagation

- A method to **compute the gradient** vector $\nabla_{\mathbf{W}} E(\mathbf{W})$ efficiently

- Just like the forward pass *forward propagates* computations to obtain output for a given the input, this algorithm ***backward propagates*** computations to obtain the gradients with respect to every parameter

- **Automatic differentiation**: Nowadays implemented efficiently in all deep learning libraries that make us of GPUs (tensorflow, pytorch, etc.)

Backpropagation

- A method to **compute the gradient** vector $\nabla_{\mathbf{W}} E(\mathbf{W})$ efficiently
- Just like the forward pass *forward propagates* computations to obtain output for a given the input, this algorithm ***backward propagates*** computations to obtain the gradients with respect to every parameter
- **Automatic differentiation**: Nowadays implemented efficiently in all deep learning libraries that make us of GPUs (tensorflow, pytorch, etc.)
- Requires **chain rule of calculus**

$$f\big(g(x)\big)' = f'\big(g(x)\big) \cdot g'(x)$$

- For partial derivatives

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial x}$$

Backpropagation

- **Objective**: Compute the gradient of error of one data point.
  Example: $E_n = \frac{1}{2}\left(h_{\mathbf{W}}\left(\mathbf{x}^{(n)}\right) - y^{(n)}\right)^2$

- Remember (neuron output and neuron activation)

$$z_j^{(\ell)} = \sigma\left(a_j^{(\ell)}\right), \qquad a_j^{(\ell)} = \sum_i w_{ji}^{(\ell)} z_i^{(\ell-1)}$$

- We apply chain rule to compute

$$\frac{\partial E_n}{\partial w_{ji}^{(\ell)}} = \frac{\partial E_n}{\partial a_j^{(\ell)}} \cdot \frac{\partial a_j^{(\ell)}}{\partial w_{ji}^{(\ell)}}$$

- We define $\frac{\partial E_n}{\partial a_j^{(\ell)}}$ as the **local error** $\delta_j^{(\ell)}$ (to be derived later)

- The second term is just $\frac{\partial a_j^{(\ell)}}{\partial w_{ji}^{(\ell)}} = z_i^{(\ell-1)}$

Backpropagation

- The derivative is the local error $\delta_j^{(\ell)}$ times the output of the in- unit

$$\frac{\partial E_n}{\partial w_{ji}^{(\ell)}} = \delta_j^{(\ell)} \cdot z_i^{(\ell-1)}$$

- How to compute the **local error** $\delta_j^{(\ell)} = \frac{\partial E_n}{\partial a_j^{(\ell)}}$?
- We proceed **recursively**

1. **Output unit** (for ℓ the final layer) can be done directly.

   Example: for canonical output (e.g., without nonlinearity)

$$\frac{\partial E_n}{\partial a_j^{(\ell)}} = \left(h_k(\mathbf{x}) - y^{(n)}\right)$$

Backpropagation

- The derivative is the local error $\delta_j^{(\ell)}$ times the output of the in- unit

$$\frac{\partial E_n}{\partial w_{ji}^{(\ell)}} = \delta_j^{(\ell)} \cdot z_i^{(\ell-1)}$$

- How to compute the **local error** $\delta_j^{(\ell)} = \frac{\partial E_n}{\partial a_j^{(\ell)}}$?
- We proceed **recursively**

2. **Hidden unit** (for ℓ a hidden layer). Again, use the chain rule

$$\frac{\partial E_n}{\partial a_j^{(\ell)}} = \sum_k \frac{\partial E_n}{\partial a_k^{(\ell+1)}} \cdot \frac{\partial a_k^{(\ell+1)}}{\partial a_j^{(\ell)}} = \sum_k \delta_k^{(\ell+1)} \cdot \frac{\partial a_k^{(\ell+1)}}{\partial a_j^{(\ell)}}$$

Backpropagation

- The derivative is the local error $\delta_j^{(\ell)}$ times the output of the in- unit

$$\frac{\partial E_n}{\partial w_{ji}^{(\ell)}} = \delta_j^{(\ell)} \cdot z_i^{(\ell-1)}$$

- How to compute the **local error** $\delta_j^{(\ell)} = \frac{\partial E_n}{\partial a_j^{(\ell)}}$?
- We proceed **recursively**

2. **Hidden unit** (for ℓ a hidden layer). Again, use the chain rule

$$\frac{\partial E_n}{\partial a_j^{(\ell)}} = \sum_k \frac{\partial E_n}{\partial a_k^{(\ell+1)}} \cdot \frac{\partial a_k^{(\ell+1)}}{\partial a_j^{(\ell)}} = \sum_k \delta_k^{(\ell+1)} \cdot \frac{\partial a_k^{(\ell+1)}}{\partial a_j^{(\ell)}}$$

$$\frac{\partial a_k^{(\ell+1)}}{\partial a_j^{(\ell)}} = w_{kj}^{(\ell+1)} \cdot \sigma'\left(a_j^{(\ell)}\right)$$

$$\boxed{a_k^{(\ell+1)} = \sum_l w_{kl}^{(\ell+1)} \sigma\left(a_l^{(\ell)}\right)}$$

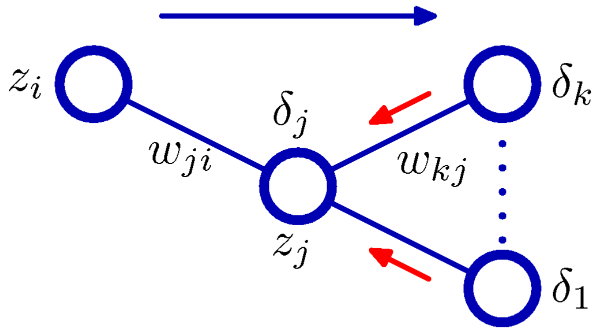
$$\delta_j^{(\ell)} = \sigma'\left(a_j^{(\ell)}\right) \sum_k w_{kj}^{(\ell+1)} \cdot \delta_k^{(\ell+1)}$$

Backpropagation

- The derivative is the local error $\delta_j^{(\ell)}$ times the output of the in- unit

$$\frac{\partial E_n}{\partial w_{ji}^{(\ell)}} = z_i^{(\ell-1)} \cdot \delta_j^{(\ell)} = z_i^{(\ell-1)} \cdot \left(\sigma' \left(a_j^{(\ell)} \right) \sum_k w_{kj}^{(\ell+1)} \cdot \delta_k^{(\ell+1)} \right)$$

- The **local error** can be obtained by propagating backwards the local errors δ's of subsequent layers



- The **backpropagation algorithm** can be seen as an instance of **dynamic programming**

Backpropagation

Summary

1. Apply an input vector to the network and **forward**-propagate

$$z_j^{(\ell)} = \sigma\left(a_j^{(\ell)}\right), \qquad a_j^{(\ell)} = \sum_i w_{ji}^{(\ell)} z_i^{(\ell-1)}$$

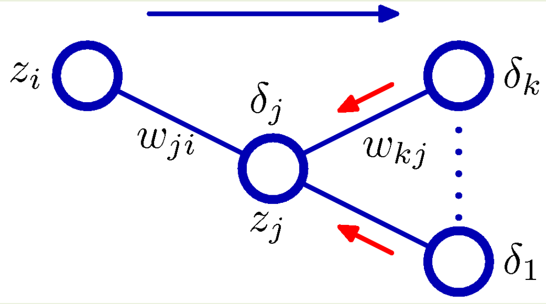
2. Evaluate the **local errors** $\delta_k^{(\ell)} = \frac{\partial E_n}{\partial a_k^{(\ell)}}$ for each **output** unit k

3. Back-propagate the errors to obtain the $\delta_j^{(\ell)}$ for each **hidden** unit j

$$\delta_j^{(\ell)} = \sigma'\left(a_j^{(\ell)}\right) \sum_k w_{kj}^{(\ell+1)} \cdot \delta_k^{(\ell+1)}$$

4. Evaluate the required derivatives using

$$\frac{\partial E_n}{\partial w_{ji}^{(\ell)}} = \delta_j^{(\ell)} \cdot z_i^{(\ell-1)}$$



Example

Consider the previous model

$$\mathbf{W}^{(1)} = \begin{pmatrix} 0.25 & 0.30 \\ 0.15 & 0.20 \end{pmatrix}, \qquad \mathbf{b}^{(1)} = \begin{pmatrix} 0.35 \\ 0.35 \end{pmatrix}$$

$$\mathbf{W}^{(2)} = \begin{pmatrix} 0.45 & 0.40 \\ 0.50 & 0.55 \end{pmatrix}, \qquad \mathbf{b}^{(2)} = \begin{pmatrix} 0.60 \\ 0.60 \end{pmatrix}$$

And given data point consisting of input $\mathbf{x}$ and output $\mathbf{y}$

$$\mathbf{x} = \begin{pmatrix} 0.05 \\ 0.10 \end{pmatrix}, \qquad \mathbf{y} = \begin{pmatrix} 0.01 \\ 0.99 \end{pmatrix}$$

d) Update parameters $\mathbf{W}$ using backpropagation

Example

d) Update all parameters using backpropagation

$$\mathbf{h} = \sigma\left(\mathbf{a}^{(2)}\right) = \frac{1}{1 + e^{-\mathbf{a}^{(1)}}} = \begin{pmatrix} 0.75 \\ 0.77 \end{pmatrix}$$

Local errors for **output units**:

$$\delta_1^{(2)} = 0.75 - 0.01 = 0.74, \qquad \delta_2^{(2)} = 0.77 - 0.99 = -0.22$$

Example

d) Update all parameters using backpropagation

$$\mathbf{h} = \sigma\left(\mathbf{a}^{(2)}\right) = \frac{1}{1 + e^{-\mathbf{a}^{(1)}}} = \begin{pmatrix} 0.75 \\ 0.77 \end{pmatrix}$$

Local errors for **output units**:

$$\delta_1^{(2)} = 0.75 - 0.01 = 0.74, \qquad \delta_2^{(2)} = 0.77 - 0.99 = -0.22$$

Local errors for **hidden layer** units:

$$\begin{aligned}
\delta_1^{(1)} &= \sigma'\left(a_1^{(1)}\right) \sum_k w_{k1}^{(2)} \cdot \delta_k^{(2)} \\
&= \sigma(0.39)\big(1 - \sigma(0.39)\big)(0.45 \cdot 0.74 + 0.50 \cdot (-0.22)) = 0.05 \\
\delta_2^{(1)} &= \sigma'\left(a_2^{(1)}\right) \sum_k w_{k2}^{(2)} \cdot \delta_k^{(2)} \\
&= \sigma(0.38)\big(1 - \sigma(0.38)\big)(0.40 \cdot 0.74 + 0.55 \cdot (-0.22)) = 0.03
\end{aligned}$$

Example

d) Update all parameters using backpropagation

$$\mathbf{h} = \sigma\left(\mathbf{a}^{(2)}\right) = \frac{1}{1 + e^{-\mathbf{a}^{(1)}}} = \begin{pmatrix} 0.75 \\ 0.77 \end{pmatrix}$$

Derivatives of 2nd layer:

$$\frac{\partial E_n}{\partial w_{11}^{(2)}} = \delta_1^{(2)} \cdot z_1^{(1)}, \qquad \frac{\partial E_n}{\partial w_{12}^{(2)}} = \delta_1^{(2)} \cdot z_2^{(1)}$$

$$\frac{\partial E_n}{\partial w_{21}^{(2)}} = \delta_2^{(2)} \cdot z_1^{(1)}, \qquad \frac{\partial E_n}{\partial w_{22}^{(2)}} = \delta_2^{(2)} \cdot z_2^{(1)}$$

In matrix form:

$$\frac{\partial E_n}{\partial \mathbf{W}^{(2)}} = \begin{pmatrix} \delta_1^{(2)} \\ \delta_2^{(2)} \end{pmatrix} \left(z_1^{(1)}, z_2^{(1)}\right) = \begin{pmatrix} 0.74 \\ -0.22 \end{pmatrix} (0.60, 0.59) = \begin{pmatrix} 0.44 & 0.44 \\ -13.2 & -12.8 \end{pmatrix}$$

Example

d) Update all parameters using backpropagation

$$\mathbf{h} = \sigma\left(\mathbf{a}^{(2)}\right) = \frac{1}{1 + e^{-\mathbf{a}^{(1)}}} = \begin{pmatrix} 0.75 \\ 0.77 \end{pmatrix}$$

Derivatives of 1st layer:

$$\frac{\partial E_n}{\partial w_{11}^{(1)}} = \delta_1^{(1)} \cdot x_1, \qquad \frac{\partial E_n}{\partial w_{12}^{(1)}} = \delta_1^{(1)} \cdot x_2$$

$$\frac{\partial E_n}{\partial w_{21}^{(1)}} = \delta_2^{(1)} \cdot x_1, \qquad \frac{\partial E_n}{\partial w_{22}^{(1)}} = \delta_2^{(1)} \cdot x_2$$

In matrix form:

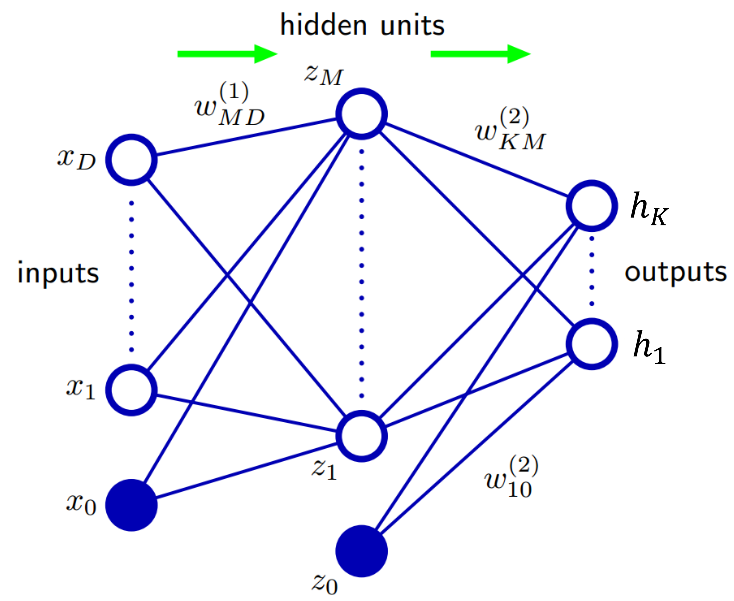
$$\frac{\partial E_n}{\partial \mathbf{W}^{(1)}} = \begin{pmatrix} \delta_1^{(1)} \\ \delta_2^{(1)} \end{pmatrix} (x_1, x_2) = \begin{pmatrix} 0.05 \\ 0.03 \end{pmatrix} (0.05, 0.01) = \begin{pmatrix} 2.5 \cdot 10^{-3} & 5 \cdot 10^{-4} \\ 1.5 \cdot 10^{-3} & 3 \cdot 10^{-4} \end{pmatrix}$$

The final update is

$$\mathbf{W}' \leftarrow \mathbf{W} - \eta \frac{\partial E_n}{\partial \mathbf{W}}$$

Example

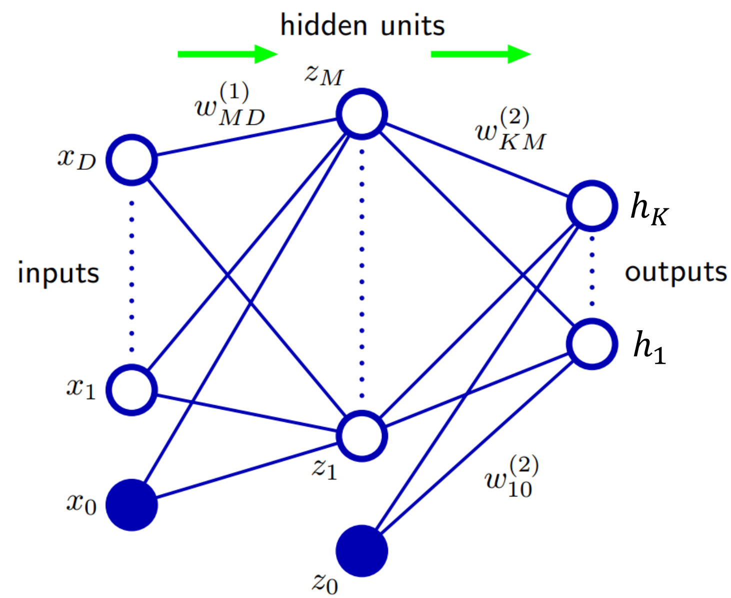
- Two layered network (1 hidden ,1 output)
- Sum-of-squares error $E_n = \frac{1}{2}\sum_k \left(h_k - y_k^{(n)}\right)^2$
- Linear outputs $h_k = a_k$
- Logistic hidden $h(a) = \tanh(a)$
  where $\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$



1. Apply an input vector to the network and **forward**-propagate

Example

- Two layered network (1 hidden ,1 output)
- Sum-of-squares error $E_n = \frac{1}{2}\sum_k \left(h_k - y_k^{(n)}\right)^2$
- Linear outputs $h_k = a_k$
- Logistic hidden $h(a) = \tanh(a)$
  where $\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$



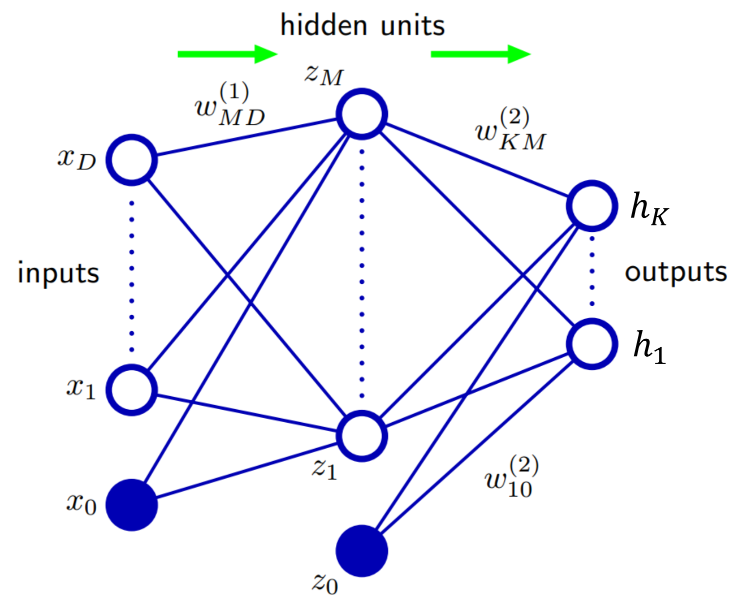
1. Apply an input vector to the network and **forward**-propagate

$$a_j^{(1)} = \sum_{i=0}^{D} w_{ji}^{(1)} x_i$$
$$z_j^{(1)} = \tanh\left(a_j^{(1)}\right)$$
$$h_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j^{(1)}$$

(Uses implicit bias $\mathbf{b}^{(1)} = \mathbf{w}_0^{(1)}$)

Example

- Two layered network (1 hidden ,1 output)
- Sum-of-squares error $E_n = \frac{1}{2}\sum_k \left(h_k - y_k^{(n)}\right)^2$
- Linear outputs $h_k = a_k$
- Logistic hidden $h(a) = \tanh(a)$
  where $\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$



2. Evaluate the $\delta_k^{(2)}$ for all the output units

$$\delta_k^{(2)} = h_k - y_k$$

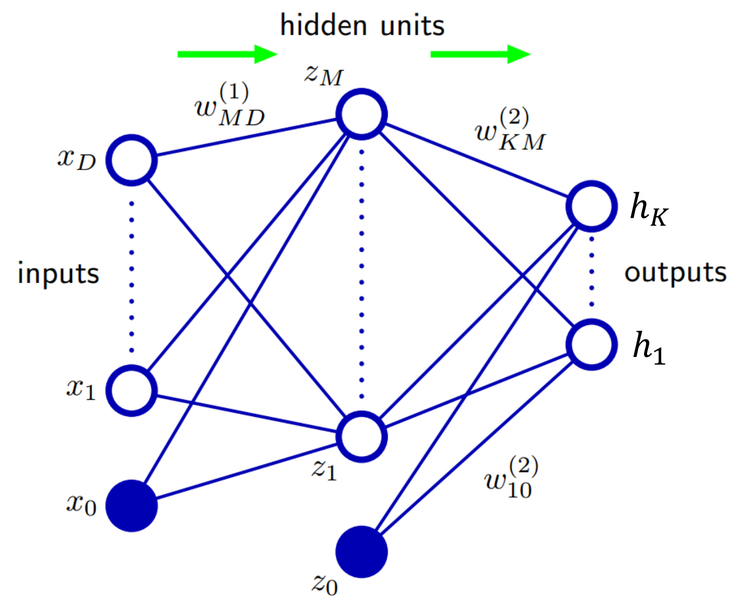
3. Back-propagate to obtain the δ's of the hidden units

$$\delta_j^{(1)} = \left(1 - {z^{(1)}_j}^2\right)\sum_{k=1}^{K} w_{kj}^{(2)}\delta_k^{(2)}$$

Example

- Two layered network (1 hidden ,1 output)
- Sum-of-squares error $E_n = \frac{1}{2}\sum_k \left(h_k - y_k^{(n)}\right)^2$
- Linear outputs $h_k = a_k$
- Logistic hidden $h(a) = \tanh(a)$

  where $\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$



4. Evaluate derivatives with respect to the 1st-layer and 2nd-layer

$$\frac{\partial E_n}{\partial w_{ji}^{(1)}} = \delta_j^{(1)} x_i \,, \qquad \frac{\partial E_n}{\partial w_{kj}^{(2)}} = \delta_k^{(2)} z_j^{(1)}$$