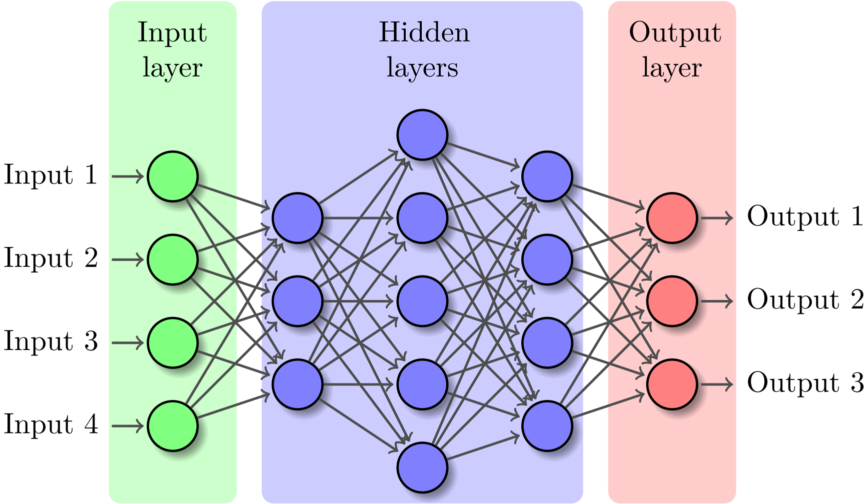
Neural Networks

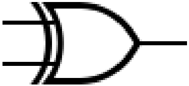
October 4, 2017

Inspiration

Feed-forward Artificial Neural Network (ANN)



Motivating example: XOR (Exclusive Or)



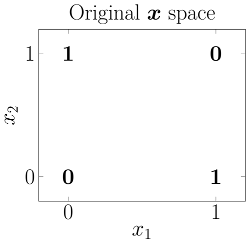
$$\mathcal{O} = (A + B) \cdot \overline{(A \cdot B)}$$

A	B	$\mathcal{O}$
0	0	0
0	1	1
1	0	1
1	1	0

Choosing a linear model:

$$f\left(\boldsymbol{x}; \boldsymbol{w}, b\right) = \boldsymbol{x}^T \boldsymbol{w} + b$$

we get $\boldsymbol{w} = \boldsymbol{0}$ and $b = \frac{1}{2}$, *i.e.* the linear model outputs 0.5 everywhere.

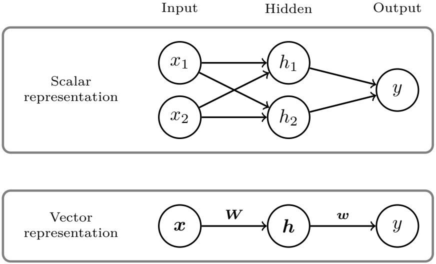


Learning with simple Feedforward Network

- One hidden layer with two units.
- Two step process:
  - $\boldsymbol{h} = f^{(1)}\left(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}\right)$
  - $y = f^{(2)}\left(\boldsymbol{h}; \boldsymbol{w}, b\right)$
- The complete model looks like:

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

- What should $f^{(1)}$ and $f^{(2)}$ look like?



Choosing both $f^{(1)}$ and $f^{(2)}$ as linear models we end up with a linear model.

$$f^{(1)} = \boldsymbol{W}^T \boldsymbol{x}, \quad f^{(2)} = \boldsymbol{h}^T \boldsymbol{w} \quad \Longrightarrow \quad f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{W}^T \boldsymbol{x}$$

Clearly we need one of the functions to be non-linear!

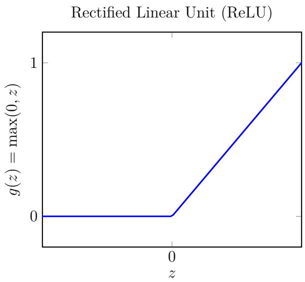
Rectified Linear Unit (ReLU)

- We want to chose $f^{(1)}$ as a non-linear function.
- Modern neural networks employs Rectified linear units.

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

- We then get:

$$f^{(1)} = \max(0, \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{c})$$



Our complete model becomes:

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

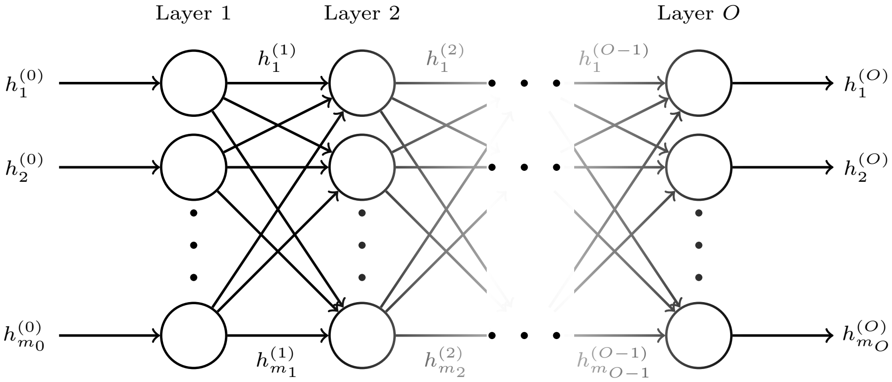
The complete XOR model

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

$$\boldsymbol{W}=\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}\qquad \boldsymbol{c}=\begin{bmatrix}0\\ -1\end{bmatrix}\qquad \boldsymbol{w}=\begin{bmatrix}1\\ -2\end{bmatrix}\qquad b=0$$

$$X:\begin{bmatrix}0 & 0\\ 0 & 1\\ 1 & 0\\ 1 & 1\end{bmatrix}\xrightarrow{\boldsymbol{W}}\begin{bmatrix}0 & 0\\ 1 & 1\\ 1 & 1\\ 2 & 2\end{bmatrix}\xrightarrow{+\boldsymbol{c}}\begin{bmatrix}0 & -1\\ 1 & 0\\ 1 & 0\\ 2 & 1\end{bmatrix}\xrightarrow{\textbf{ReLU}}\begin{bmatrix}0 & 0\\ 1 & 0\\ 1 & 0\\ 2 & 1\end{bmatrix}\xrightarrow{\boldsymbol{w}^T}\begin{bmatrix}0\\ 1\\ 1\\ 0\end{bmatrix}$$

General Feedforward network



- A feedforward network consist of layers of neurons.
- The output of neurons in layer n is passed as input to the neurons in layer $n+1$

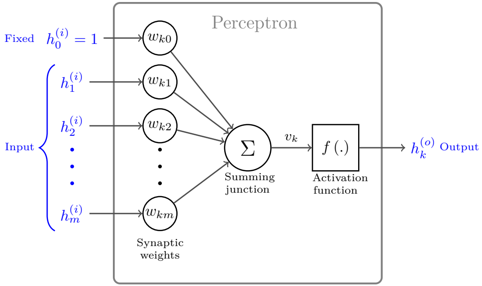
Neuron/Perceptron

1. **Perceptron** recieves input $\boldsymbol{h}^{(i)} = \left(h_1^{(i)}, h_2^{(i)}, \ldots, h_m^{(i)}\right)$ from previous layer. If a **bias** is included in the model add a fixed $h_0^{(i)} = 1$ to the input.
2. Inputs are multiplied by the **Synaptic Weights**, and added together at the **Summing Junction** (Linear/Affine transformation).

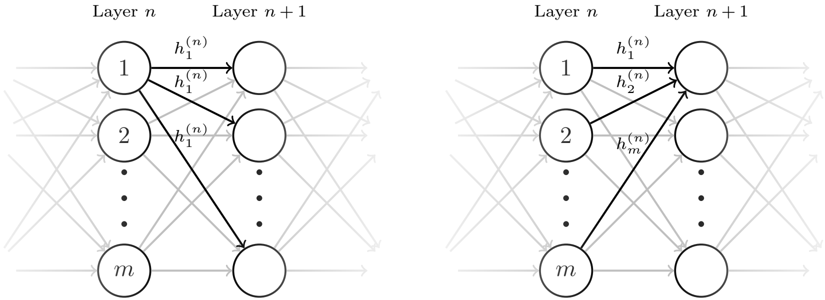
$$v_k = \boldsymbol{w}_k^T \boldsymbol{h}^{(i)} = \sum_{j=0}^{m} w_{kj} h_j^{(i)}$$

3. **Activation function** is evaluated to give **Perceptron** output (Non-linear transformation).

$$h_k^{(o)} = f\left(v_k\right)$$



Feeding forward



- Each Perceptron in layer n sends its output to each Perceptron in layer $n+1$.
- Each Perceptron in layer $n+1$ recieves input from all Perceptrons in layer n.

Evaluating a feedforward network

Evaluating Perceptron k in layer n

$$h_k^{(n)} = f_k^{(n)} \left(\boldsymbol{w}_k^{(n)T} \boldsymbol{h}^{(n-1)} \right)$$

Evaluating the whole network is just a question of evaluating each layer in succession, i.e. **Feed-forward**.

$$h_k^{(1)} = f_k^{(1)} \left(\boldsymbol{w}_k^{(1)T} \boldsymbol{h}^{(0)} \right)$$

$$h_k^{(2)} = f_k^{(2)} \left(\boldsymbol{w}_k^{(2)T} \boldsymbol{h}^{(1)} \right)$$

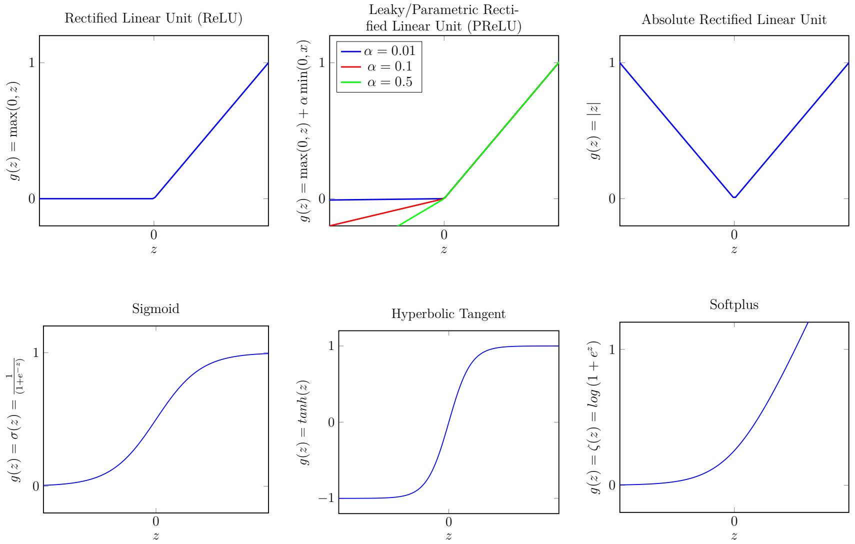
$$\cdots$$

$$h_k^{(O)} = f_k^{(O)} \left(\boldsymbol{w}_k^{(O)T} \boldsymbol{h}^{(O-1)} \right)$$

If all neurons in a layer has the same activation function we can rewrite the layer evaluation:

$$\boldsymbol{h}^{(n)} = f^{(n)} \left(\boldsymbol{W}^{(n)T} \boldsymbol{h}^{(n-1)} \right)$$

Activation functions



Minimization

Optimizing the weight of a feedforward neural network is a minimization problem!

Optimization

To optimize the weights of out network, we want, given a training set $\boldsymbol{x} \in \mathbb{X}$, to minimize a loss function $J(\boldsymbol{w})$ over the training set. This could e.g. be a sum of squares loss function

$$J(\boldsymbol{w}) = \frac{1}{2} \sum_{\boldsymbol{x} \in \mathbb{X}} \left(f^* (\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{w}) \right)^2$$

The equation is at an optimum if the gradient is w.r.t. the weights is $\mathbf{0}$

$$\frac{\partial}{\partial \boldsymbol{w}} J(\boldsymbol{w}) = \mathbf{0}$$

We thus need derivatives of the loss function with respect to all linear weights $\boldsymbol{w}_{kj}$ of the Perceptrons.

$$\frac{\partial J(\boldsymbol{w})}{\partial w_{kj}} = 0$$

Back-propagation: Chain-rule

Chain-rule

Let $\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$, $g : \mathbb{R}^m \to \mathbb{R}^n$, and $f : \mathbb{R}^m \to \mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$, then:

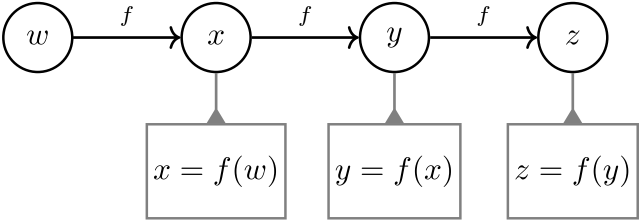
$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

Written in vector notation we get,

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{y}} z,$$

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n \times m$ Jacobian of g.

Graph derivative: Example



Derivative

$$\boxed{\begin{aligned}\frac{\partial z}{\partial w} &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}\\ &= f'(y)f'(x)f'(w)\\ &= f'(f(f(w)))f'(f(w))f'(w)\end{aligned}}$$

Backpropagation derivation

Definition $\alpha_k^{(n)}$

Remember that the output of layer n can be written as

$$h_k^{(n)} = f_k^{(n)}\left(\boldsymbol{w}_k^{(n)T}\boldsymbol{h}^{(n-1)}\right) = f_k^{(n)}\left(\alpha_k^{(n)}\right)$$

with the definition

$$\alpha_k^{(n)} = \boldsymbol{w}_k^{(n)T}\boldsymbol{h}^{(n-1)} = \sum_i w_{ki}^{(n)} h_i^{(n-1)}$$

As $w_{ki}^{(n)}$ only enters the loss function through $\alpha_k^{(n)}$ we can, using the Chain rule, write the gradient of the loss function as:

$$\frac{\partial J(\boldsymbol{w})}{\partial w_{ki}^{(n)}} = \frac{\partial J(\boldsymbol{w})}{\partial \alpha_k^{(n)}}\frac{\partial \alpha_k^{(n)}}{\partial w_{ki}^{(n)}} = \delta_k^{(n)}\frac{\partial \alpha_k^{(n)}}{\partial w_{ki}^{(n)}}$$

where in the second step we have defined

$$\delta_k^{(n)} = \frac{\partial J(\boldsymbol{w})}{\partial \alpha_k^{(n)}}$$

Deriving and expression for $\delta_k^{(n)}$

First we apply the Chain rule:

$$\delta_k^{(n)} = \frac{\partial J(\boldsymbol{w})}{\partial \alpha_k^{(n)}} = \sum_l \frac{\partial J(\boldsymbol{w})}{\partial \alpha_l^{(n+1)}} \frac{\partial \alpha_l^{(n+1)}}{\partial \alpha_k^{(n)}}$$

First term is easy, it is just $\delta_l^{(n+1)}$. The second term gives:

$$\frac{\partial \alpha_l^{(n+1)}}{\partial \alpha_k^{(n)}} = \boldsymbol{w}_l^{(n)T} \frac{\partial \boldsymbol{h}^{(n)}}{\partial \alpha_k^{(n)}} = \sum_i w_{li}^{(n+1)} \frac{\partial h_i^{(n)}}{\partial \alpha_k^{(n)}} = w_{lk}^{(n+1)} h_k^{\prime(n)}$$

Combining first and second term we the expression:

$$\delta_k^{(n)} = h_k^{\prime(n)} \sum_l w_{lk}^{(n+1)} \delta_l^{(n+1)}$$

We see here why the method is known as **Back-propagation**, as $\delta_k^{(n)}$ depends on layer $n+1$! To initialize the back propagation we just need to find an expression for $\boldsymbol{\delta}^{(O)}$:

$$\delta_k^{(O)} = \frac{\partial J(\boldsymbol{w})}{\partial \alpha_k^{(O)}} = \sum_{\boldsymbol{x} \in \mathbb{X}} \left(f^*(\boldsymbol{x}) - \boldsymbol{h}^{(O)} \right)_k \frac{\partial h_k^{(O)}}{\partial \alpha_k^{(O)}} = \sum_{\boldsymbol{x} \in \mathbb{X}} \left(f^*(\boldsymbol{x}) - \boldsymbol{h}^{(O)} \right)_k h_k^{\prime(O)}$$

Backpropagated gradient

We now have an expression for the first term of our gradient expression:

$$\frac{\partial J(\boldsymbol{w})}{\partial w_{ki}^{(n)}} = \delta_k^{(n)} \frac{\partial \alpha_k^{(n)}}{\partial w_{ki}^{(n)}}$$

The second term is easy (remember that $\alpha_k^{(n)} = \sum_i w_{ki}^{(n)} h_i^{(n-1)}$):

$$\frac{\partial \alpha_k^{(n)}}{\partial w_{ki}^{(n)}} = h_i^{(n-1)}$$

Inserting this into our gradient expression we get an expression for our backpropagated gradient

$$\frac{\partial J(\boldsymbol{w})}{\partial w_{ki}^{(n)}} = \delta_k^{(n)} h_i^{(n-1)}$$

We can now use any gradient optimization algorithm (in fact we could also derive an expression for the Hessian. I leave this as an exercise 😎).

Universal Approximation Theorem

How general is the Neural Network model?

The **universal approximation theorem** (Hornik et al.) states that a feedforward network with a linear input layer and at least one hidden layer with *any* "squashing" 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.

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

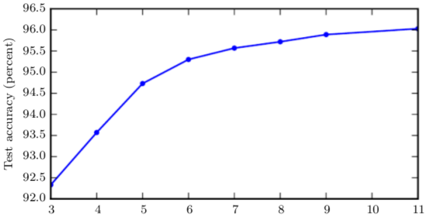
- Regardless of what function we are trying to learn, we know that a large neural network will be able to *represent* this.
- However, we are **not** guarenteed that the training algorithm will be able to *learn* this function.
- The theorem does not state how large the layer should be, and in **worst case** we need one hidden unit for each input configuration.

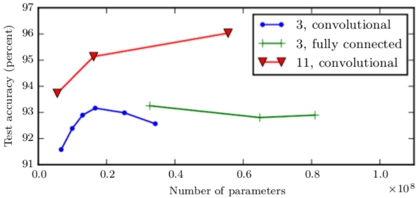
Architecture Design

Some architectures build a main chain, and then add extra features to it. These could include:

- Removing some connections between neurons to reduce number of parameters.
- Adding extra skip connections going from layer n to layer $n+2$.
- Choosing which activation function to use. One could utilize different activation functions in each layer.
- Many problems needs specialized neural network architectures, e.g. the Convolutional Neural Networks (CNNs) used in computer vision.

The depth of the network is also important. **Deeper** networks using many layers tend to perform better.



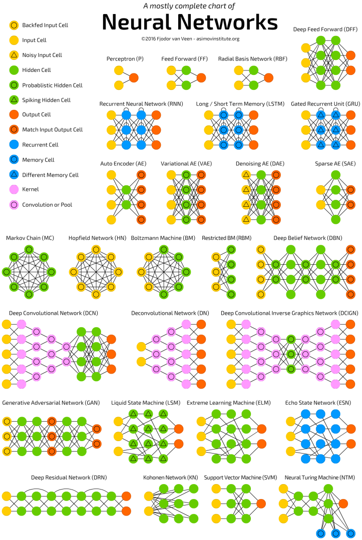


To sum up

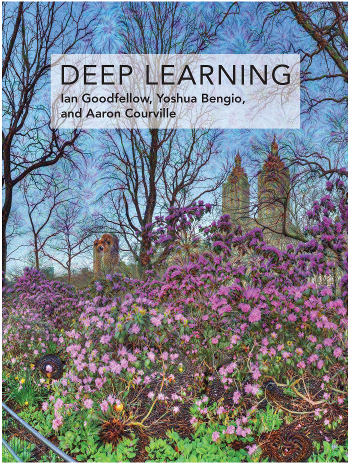
- A neural network is a function that takes a set of input $\{x_i\}$, and return a set of output $\{y_i\}$.
- This is done through a series of Linear transformations using linear weights $\boldsymbol{w}$, and non-linear function evaluations.
- Optimal weight parameters are found using an optimization/fitting algorithm, either using gradient or Hessian information.
- Network is evaluated by forward propagation.
- Derivatives of the linear weights are subsequently found by back propagation.
- Once optimal weight are found they can be stored and used for evaluating the network later.

Take home message: The Neural Network model is simply a non-linear function from a set of input variables $\{x_i\}$ to a set of output variables $\{y_i\}$ controlled by a vector $\boldsymbol{w}$ of adjustable parameters.

Flavors of Neural Networks



Resources



Available online: http://www.deeplearningbook.org/

And in pdf: https://github.com/janishar/mit-deep-learning-book-pdf

And with that...

... we are done for today!