Lecture 13: Classification: Neural Network

Course: Biomedical Data Science

Parisa Rashidi
Fall 2018

Methods

- k-NN
- Decision Tree
- Support Vector Machines
- Neural Networks
- Deep Learning

Neural Network

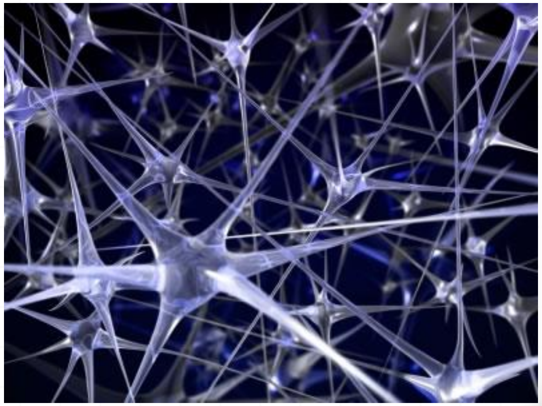
Material partially based on:

-Raschka, Sebastian. Python Machine Learning (p. 18). Packt Publishing.

-Stanford CS231n: Convolutional Neural Networks for Visual Recognition, 2017.

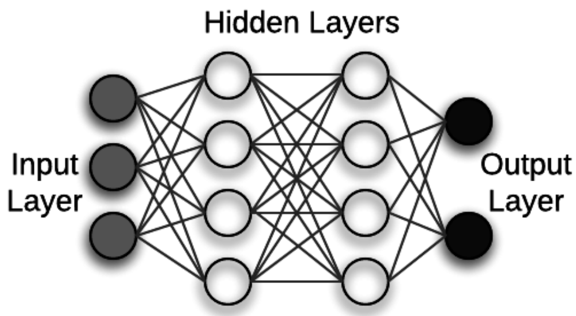
Human Brain

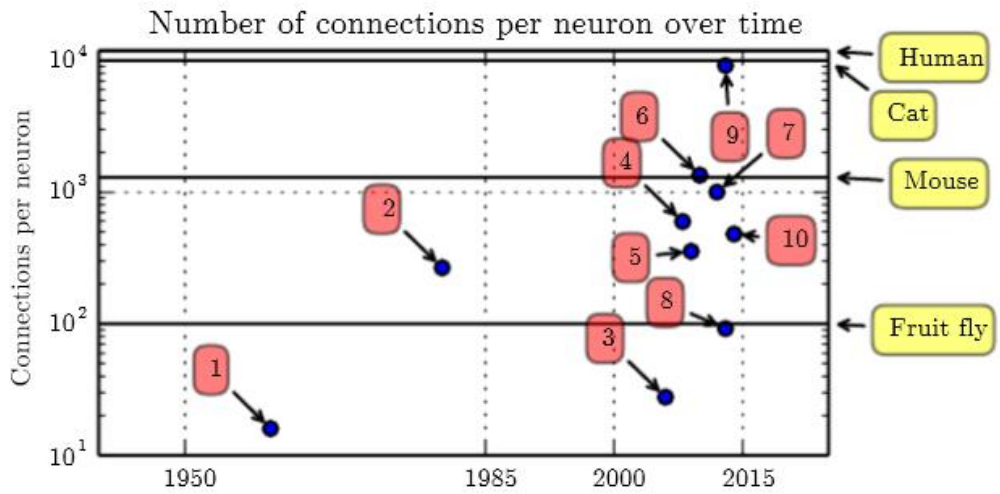
- Networks of processing units (neurons) with connections (synapses) between them
- Large number of neurons: 10^{10}
- Large connectitivity for each neuron: 10^{4}
- Parallel processing
- Distributed coupled computation/memory
- Robust to noise, failures



Comparison

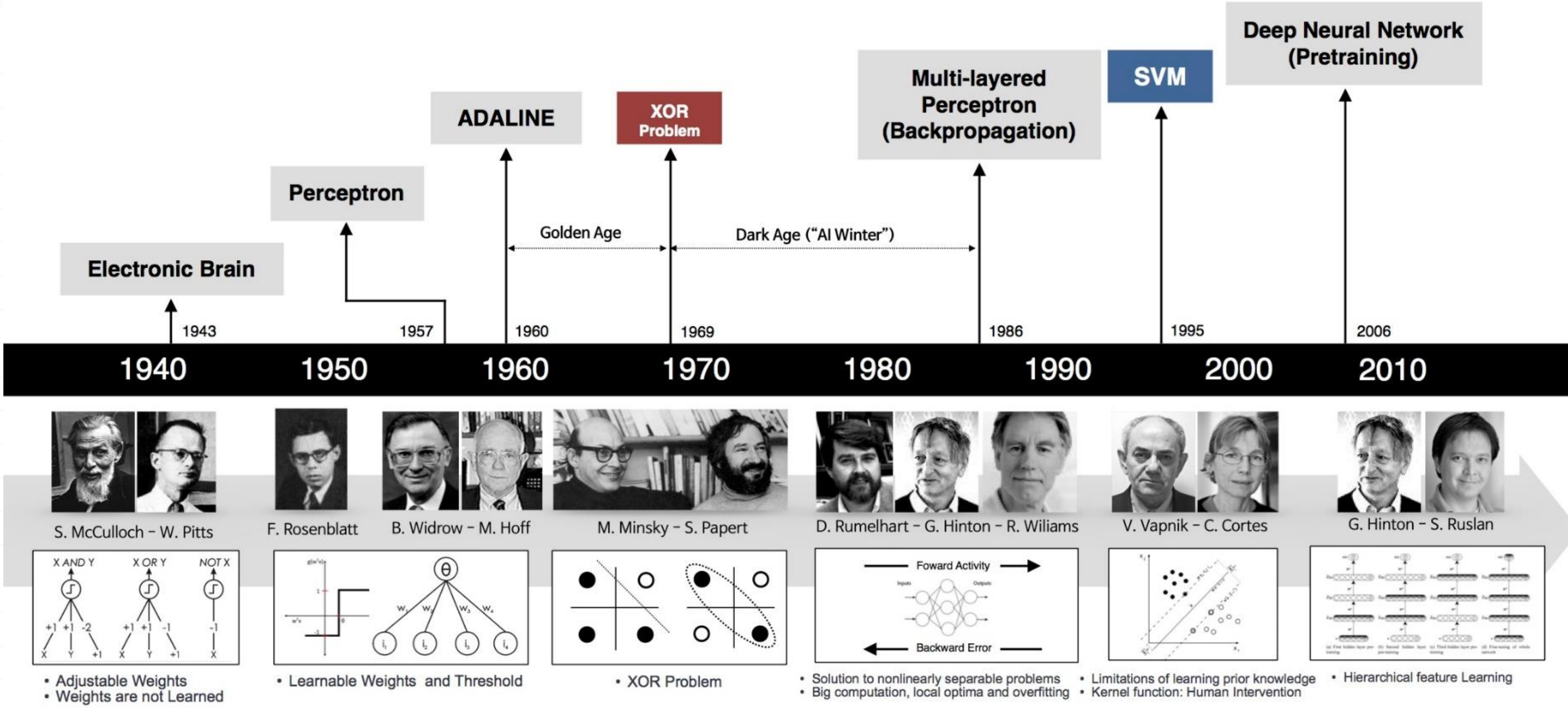
It is not just about the number of neurons. Brain is much more complex!





1. Adaptive linear element (Widrow and Hoff, 1960)
2. Neocognitron (Fukushima, 1980)
3. GPU-accelerated convolutional network (Chellapilla *et al.*, 2006)
4. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
5. Unsupervised convolutional network (Jarrett *et al.*, 2009)
6. GPU-accelerated multilayer perceptron (Ciresan *et al.*, 2010)
7. Distributed autoencoder (Le *et al.*, 2012)
8. Multi-GPU convolutional network (Krizhevsky *et al.*, 2012)
9. COTS HPC unsupervised convolutional network (Coates *et al.*, 2013)
10. GoogLeNet (Szegedy *et al.*, 2014a)

*Deep learning textbook, Goodfellow et al.



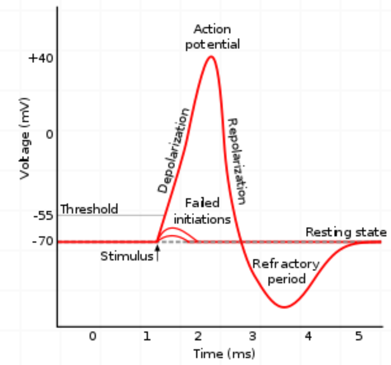
Neural Networks History

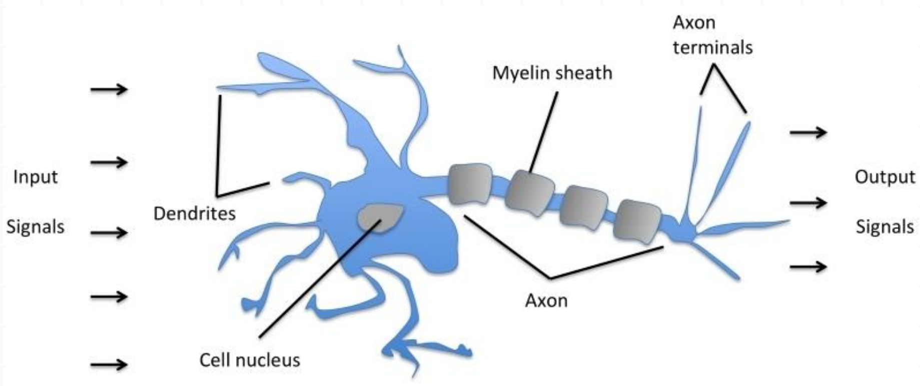
GOES BACK TO 1940, WITH SEVERAL DARK AI WINTERS

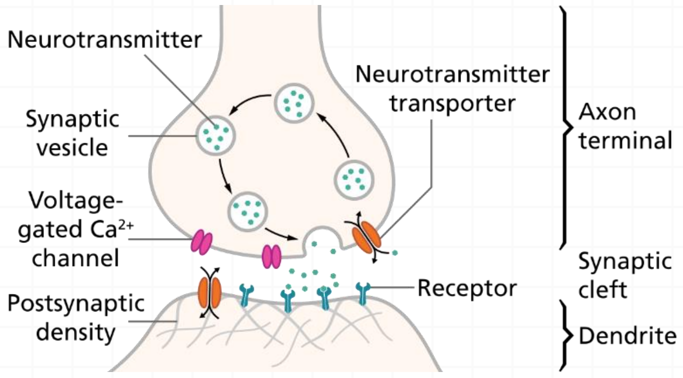
Image from https://beamandrew.github.io/deeplearning/2017/02/23/deep_learning_101_part1.html

Electronic Brain - 1943

- Warren McCulloch and Walter Pitts published the first concept of a simplified brain cell (1943).
  - A simple logic gate with binary outputs
  - Multiple signals arrive at the dendrites,
  - Signals then integrated into the cell body,
  - If the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

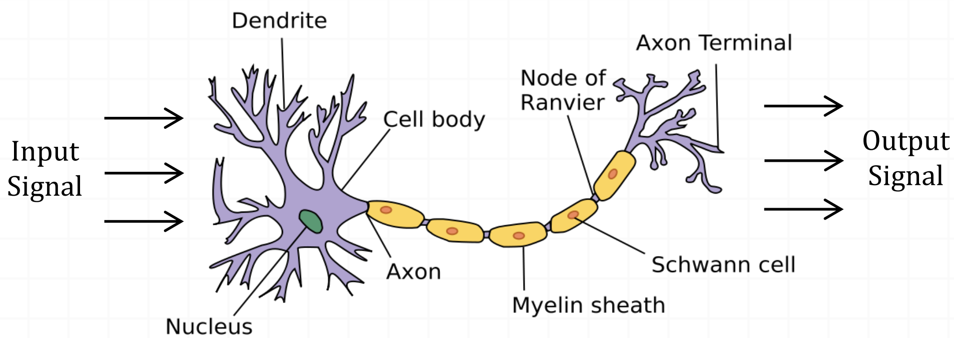


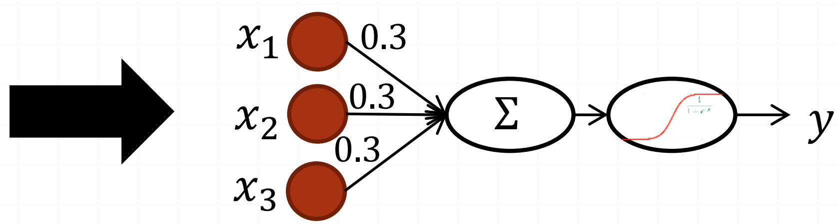




Electronic Brain - 1943

- Activation function as a step function $\Phi(x)$

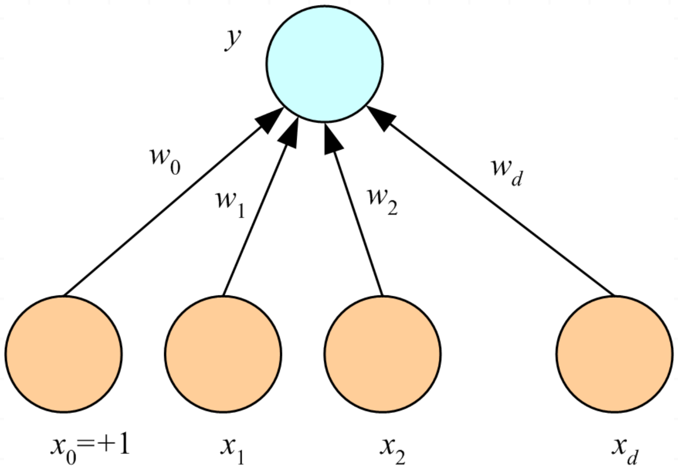




$$\Phi(x) = \begin{cases} 1 \; if \; x \geq \theta \\ -1 \; otherwise \end{cases}$$

PERCEPTRON - 1957

- The basic processing element (Rosenblat, 1957)
- Associated with each input x_i is a connection weight w_i
- Rosenblatt proposed an algorithm to automatically learn the optimal weights



$$y = \sum_{j=1}^{d} w_j x_j + w_0 = \mathbf{w}^T \mathbf{x}$$

$$\mathbf{w} = [w_0, w_1, \ldots, w_d]^T$$

$$\mathbf{x} = [1, x_1, \ldots, x_d]^T$$

PERCEPTRON - 1957

A bit of history

The **Mark I Perceptron** machine was the first implementation of the perceptron algorithm.

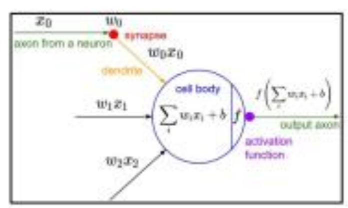
The machine was connected to a camera that used 20×20 cadmium sulfide photocells to produce a 400-pixel image.

recognized
letters of the alphabet

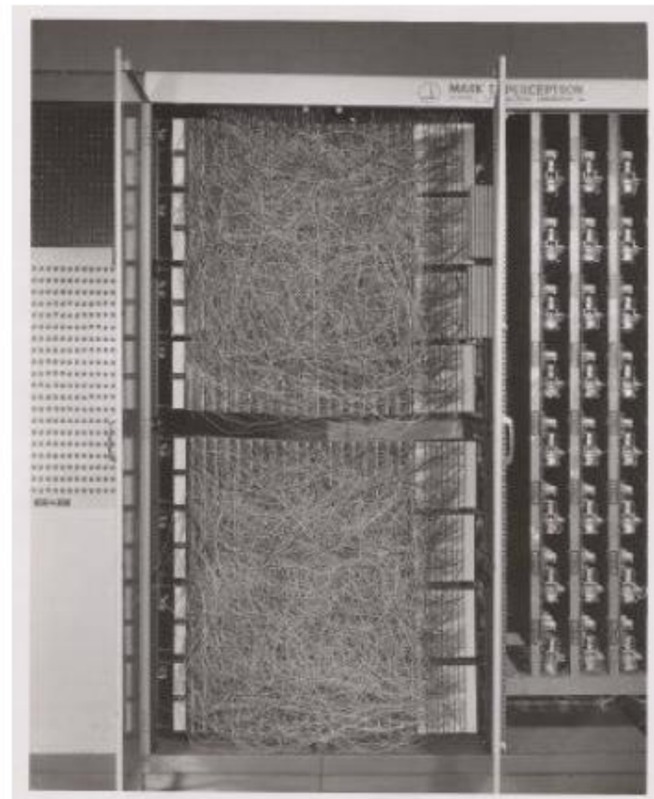
$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

update rule:

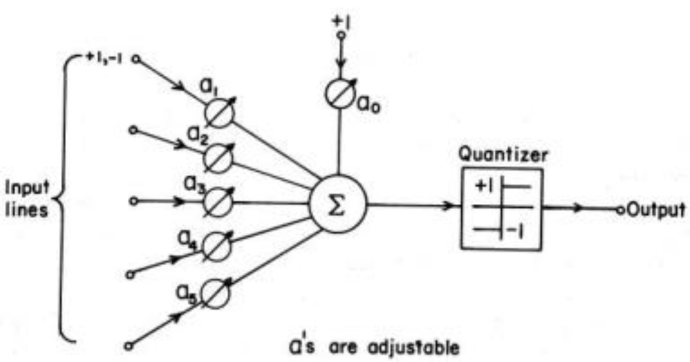
$$w_i(t+1) = w_i(t) + \alpha(d_j - y_j(t))x_{j,i},$$

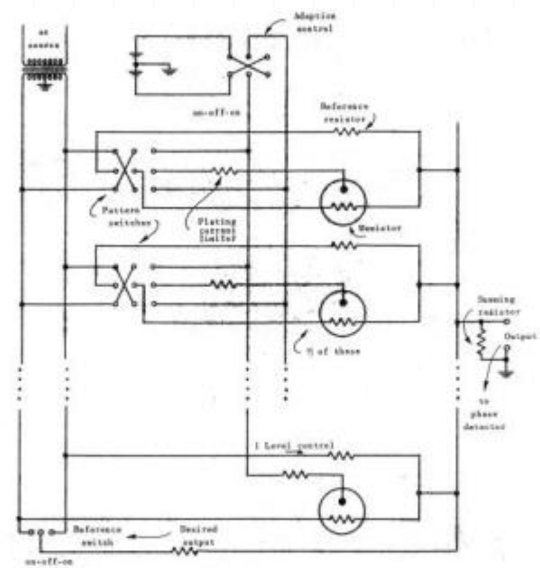


Frank Rosenblatt, ~1957: Perceptron



ADALINE - 1960





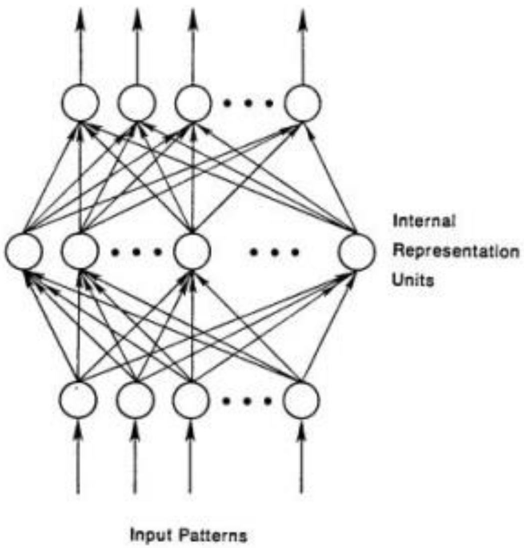
Widrow and Hoff, ~1960: Adaline/Madaline

AI WINTER (1974 – 1980)

Picture & quote from HBO series, Game of Thrones

Backpropagation - 1986

A bit of history



To be more specific, then, let

$$E_p = \frac{1}{2}\sum_j (t_{pj} - o_{pj})^2 \tag{2}$$

be our measure of the error on input/output pattern p and let $E = \sum E_p$ be our overall measure of the error. We wish to show that the delta rule implements a gradient descent in E when the units are linear. We will proceed by simply showing that

$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} i_{pi},$$

which is proportional to $\Delta_p w_{ji}$ as prescribed by the delta rule. When there are no hidden units it is straightforward to compute the relevant derivative. For this purpose we use the chain rule to write the derivative as the product of two parts: the derivative of the error with respect to the output of the unit times the derivative of the output with respect to the weight.

$$\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial w_{ji}}. \tag{3}$$

The first part tells how the error changes with the output of the jth unit and the second part tells how much changing w_{ji} changes that output. Now, the derivatives are easy to compute. First, from Equation 2

$$\frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj}) = -\delta_{pj}. \tag{4}$$

Not surprisingly, the contribution of unit u_j to the error is simply proportional to δ_{pj}. Moreover, since we have linear units,

$$o_{pj} = \sum_i w_{ji} i_{pi}, \tag{5}$$

from which we conclude that

$$\frac{\partial o_{pj}}{\partial w_{ji}} = i_{pi}.$$

Thus, substituting back into Equation 3, we see that

$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} i_i \tag{6}$$

recognizable maths

Rumelhart, Hinton, Williams (1986)

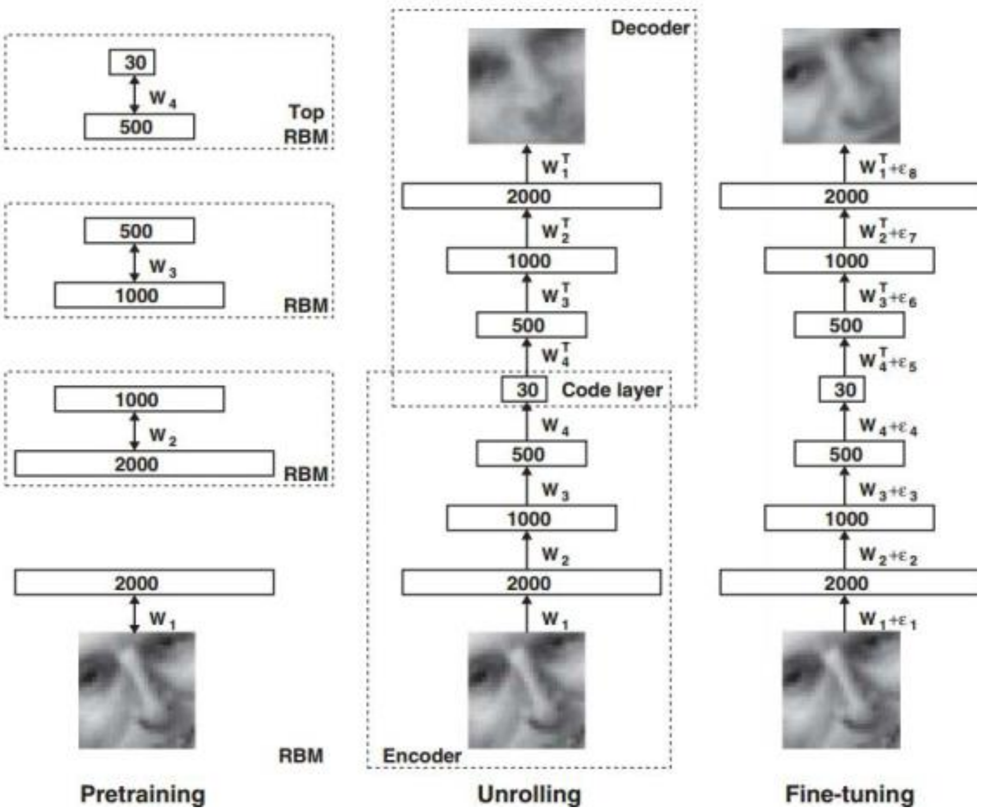
AI WINTER (1987 – 1993)

Picture & quote from HBO series, Game of Thrones

DEEP LEARNING IS REBORN

REIGNITED DEEP LEARNING IN 2006

Ruslan Salakhutdinov and Geoffrey Hinton, 2006

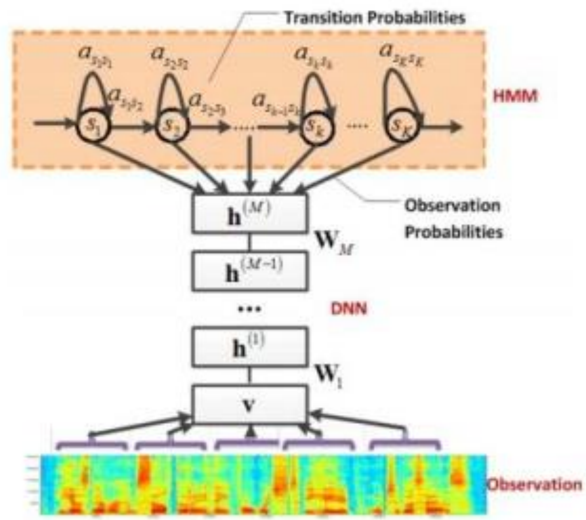


An Efficient Learning Procedure for Deep Boltzmann Machines. Ruslan Salakhutdinov and Geoffrey Hinton. Neural Computation August 2012, Vol. 24, No. 8: 1967 -- 2006.

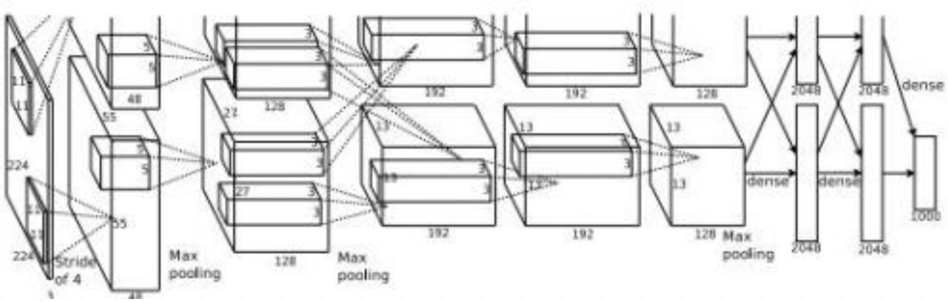
History

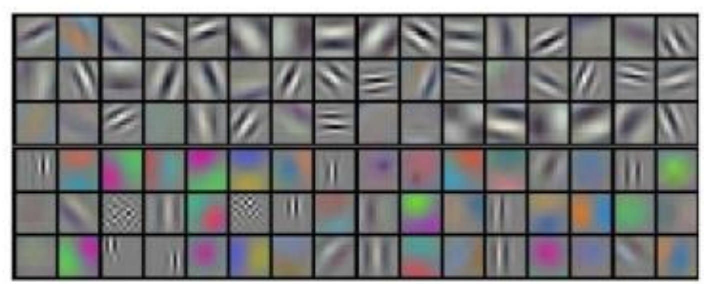
First strong results

Context-Dependent Pre-trained Deep Neural Networks for Large Vocabulary Speech Recognition
George Dahl, Dong Yu, Li Deng, Alex Acero, 2010



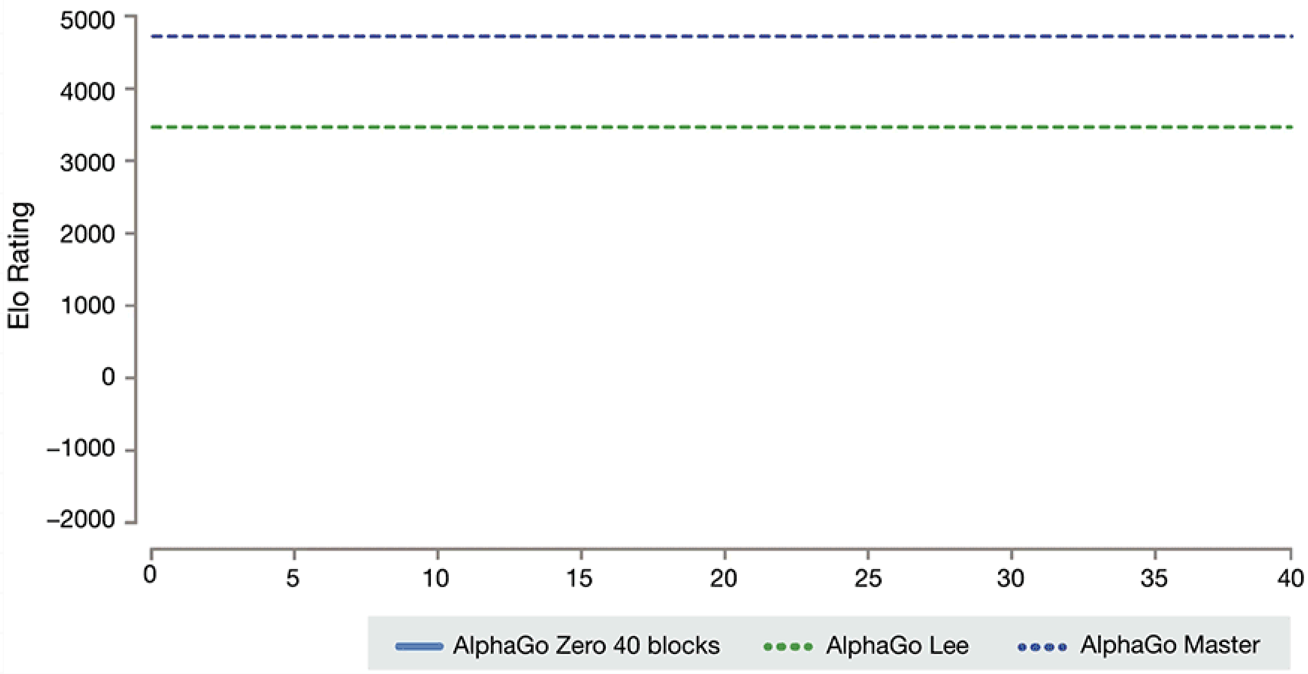
Imagenet classification with deep convolutional neural networks
Alex Krizhevsky, Ilya Sutskever, Geoffrey E Hinton, 2012





History

- Progress in many areas, such as image recognition, segmentation, reinforcement learning

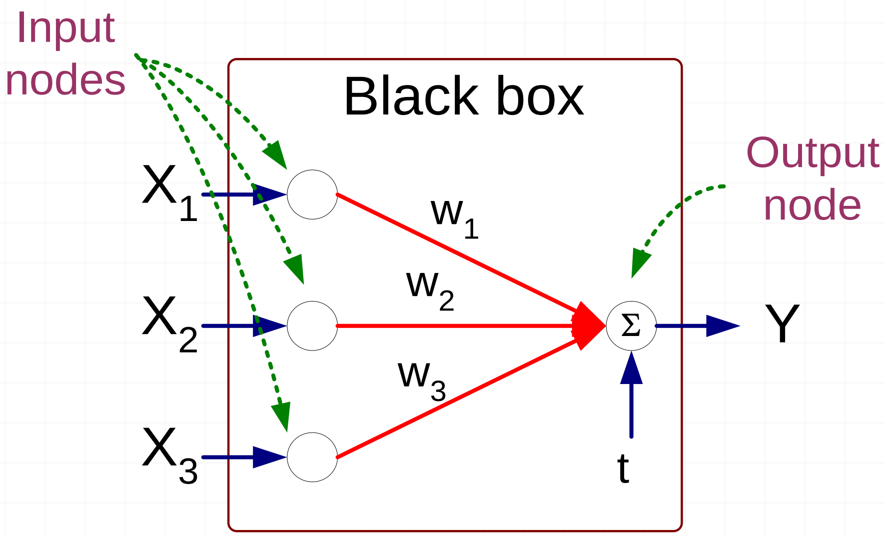


Neural Network Basics

Using Perceptron

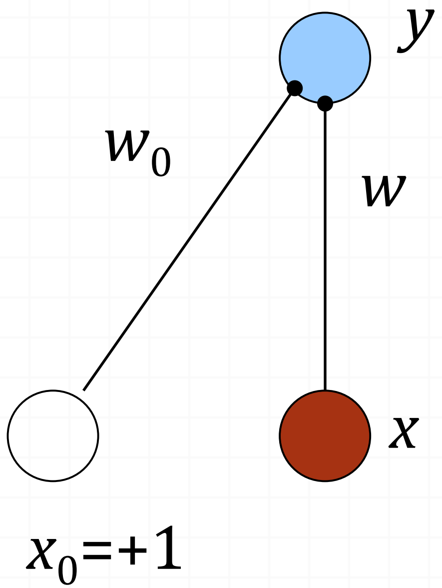
- To use perceptron, for each new input **x**, we compute y using the following equation:

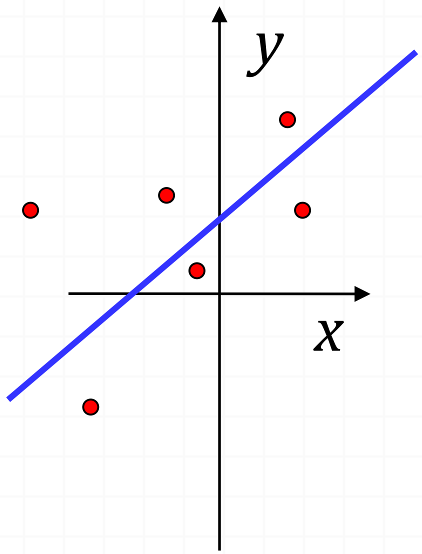
$$y = \sum_{j=1}^{d} w_j x_j + w_0 = \mathbf{w}^T \mathbf{x}$$



Regression

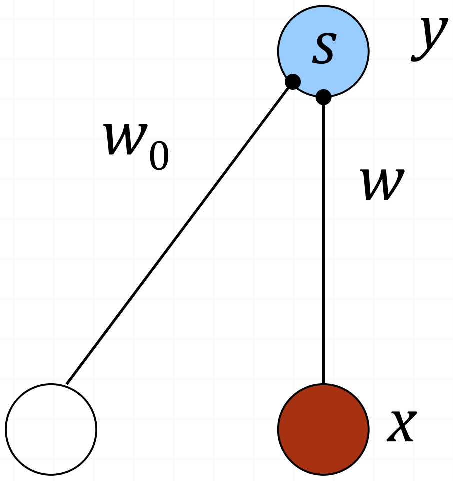
- Regression: $y=wx+w_0$

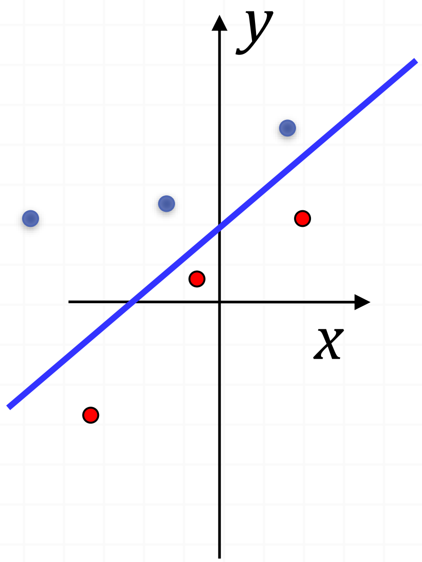




Classification

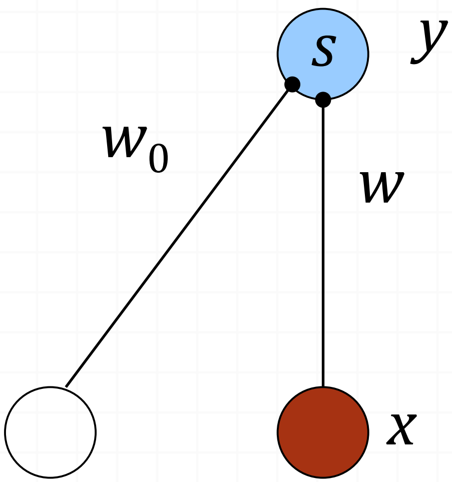
- Classification: $y=\text{sign}(wx+w_0)$

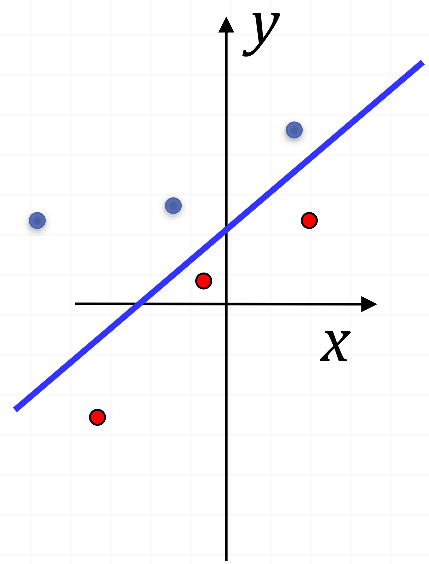


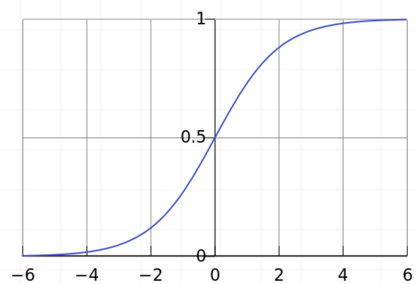


Classification + posterior probability

- Classification: y=sigmoid(wx+w_0)







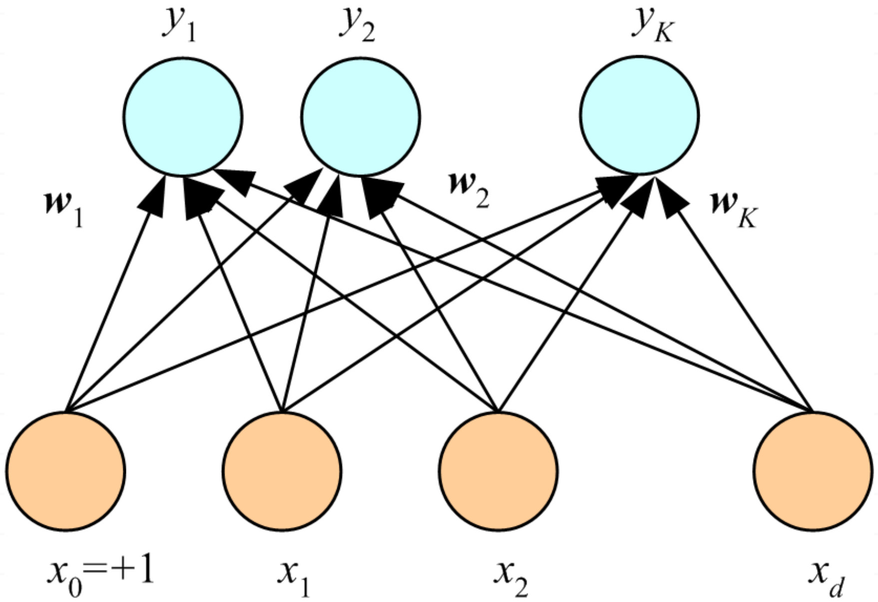
$$y = \text{sigmoid}(\mathbf{w}^T\mathbf{x}) = \frac{1}{1+\exp[\mathbf{w}^T\mathbf{x}]}$$

Multi-Output Regression

- There will be k neurons, each with a weight vector $\mathbf{w}_i$

$$y_i = \sum_{j=1}^{d} w_{ij} x_j + w_{i0} = \mathbf{w}_i^T \mathbf{x}$$

$$\mathbf{y} = \mathbf{W}\mathbf{x}$$



Multi-class Classification

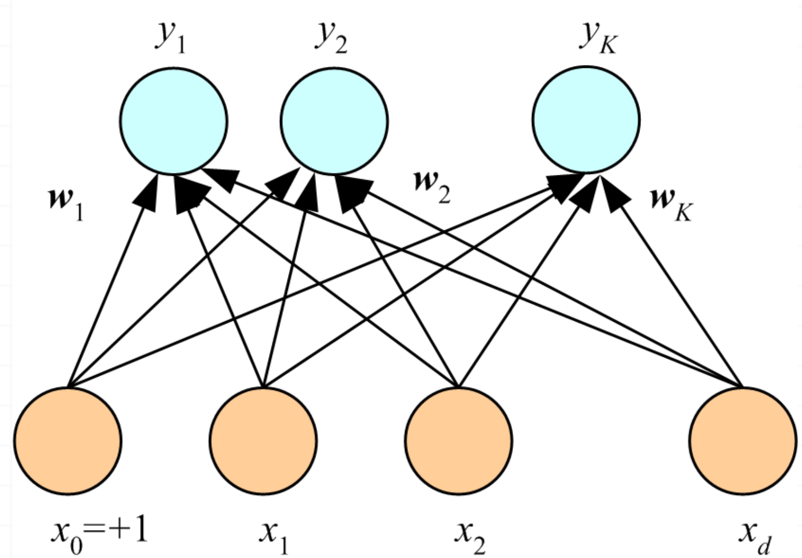
- There will be k neurons, each with a weight vector $\mathbf{w}_i$
- Choose class C_i for i:

$$y_i = \max_k y_k$$

- If posterior is needed, report y_i

$$o_i = \mathbf{w}_i^T \mathbf{x}$$

$$y_i = \frac{\exp o_i}{\sum_k \exp o_k}$$



Training a Perceptron

- We need to learn the weights **w** (the parameters of the system)
  - The weights are computed online
    - you are given the instances one by one

Perceptron Online Learning

- We do not write the error function over the whole sample at once
  - But on individual instances at each step

1. Start from random weights
2. At each iteration, adjust parameters a little bit to minimize error based on current input

Perceptron Online Learning

- Update after observing a data point

Learning Rate

Input Feature Value

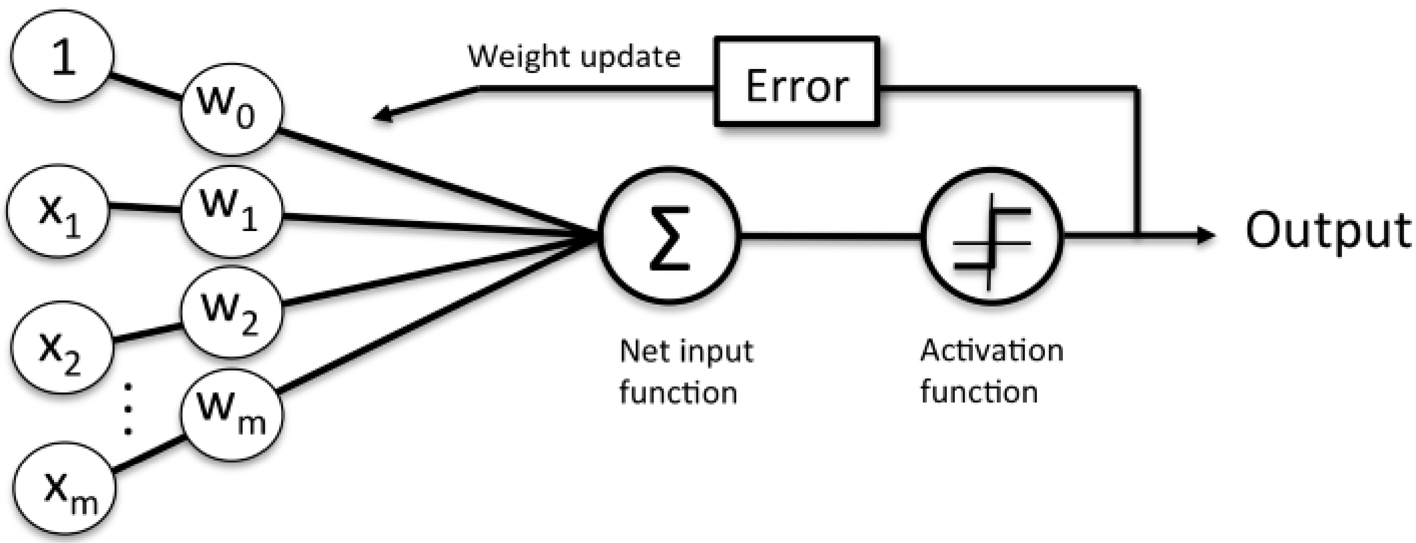
$$\Delta w_j = \eta(y_i - \tilde{y}_i)x_i^j$$

Weight update

Actual value

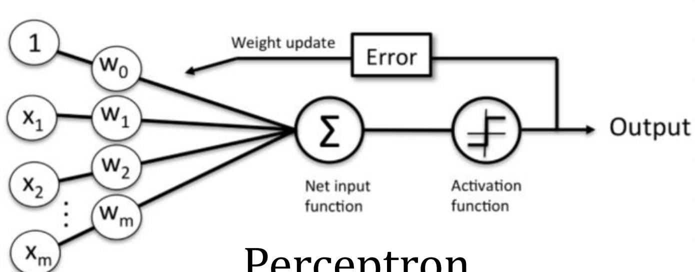
Predicted value

Training Perceptron

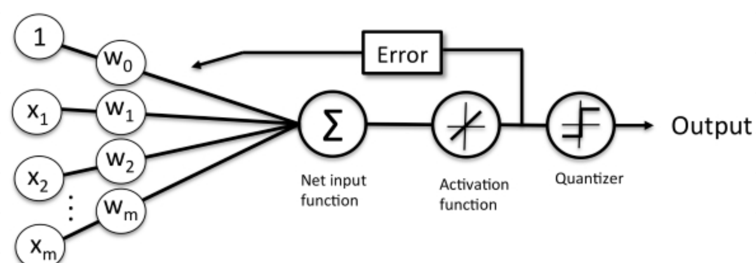


Adaptive Linear Neuron (Adaline)

- Proposed by Widrow & Hoff (1960)
  - Illustrates the key concept of defining and minimizing a cost function
    - The groundwork for understanding more advanced techniques
  - Key difference from perceptron
    - Compared to perceptron, we can use a continuous function to compute the error
    - Differentiable



Perceptron



Adaline

Adaline

- One of the key ingredients of machine learning
  - Objective function to be optimized during learning
  - This objective function is often a cost function
  - Adaline cost function

$$E(w) = \frac{1}{2}\sum_i \left(y_i - \Phi(z_i)\right)^2$$

Gradient Descent

- A simple, yet powerful optimization algorithm
- Think of it as climbing down a hill until a local or global minimum is reached.
- In each iteration, we take a step away from the gradient where the step size is determined by the value of the learning rate as well as the slope of the gradient.

Gradient Descent

- We will update the weights as following

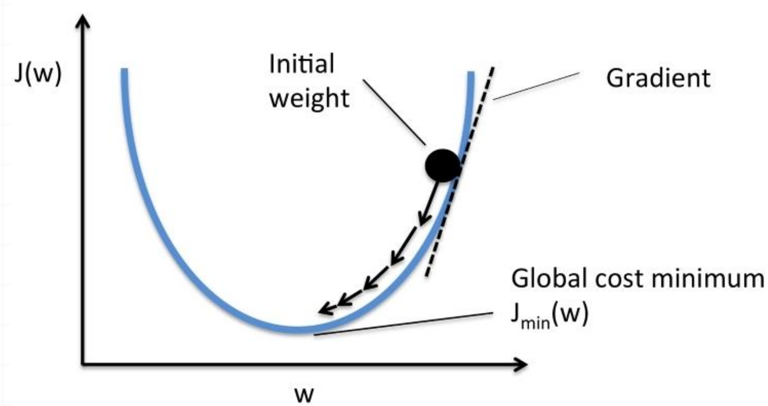
$$y = \sum_{j=1}^{d} w_j x_j + w_0 = \boldsymbol{w}^T \boldsymbol{x}$$

$$\boxed{\Delta w = -\eta \nabla E(w)} \quad \text{Update}$$

- Gradient

$$\nabla E(w) = \frac{\partial E}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i \left(y_i - \Phi(z_i)\right)^2$$

$$= -\sum_i \left(y_i - \Phi(z_i)\right) x_i^j$$



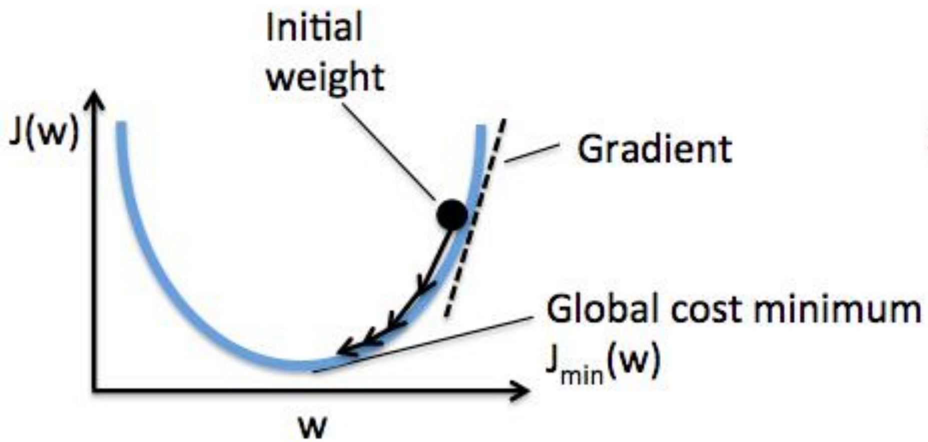
Perceptron vs. Adaline

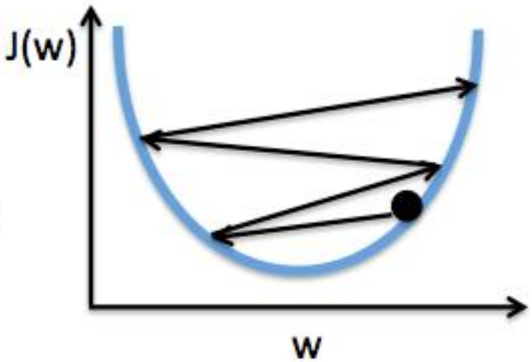
- While the update formula might look the same, it is not! (integer versus real number used in error computation)
- Perceptron uses one data point at a time to update its weight
- Adaline uses the entire batch before making an update
  - Hence the name, Batch gradient descent

Learning Rate

- If too large
  - Updates too much dependent on recent updates (i.e. short memory)
- If too small
  - Many updates needed, slow convergence
- Can be adapted over time
  - The learning factor is gradually decreased in time for convergence

Learning Rate

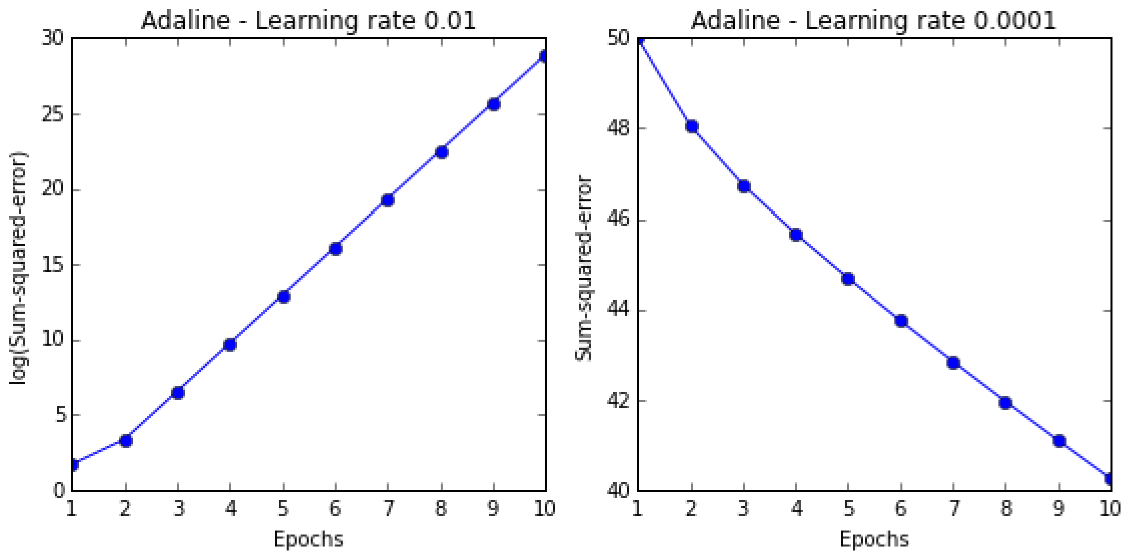




A large learning rate:
we overshoot the global
minimum

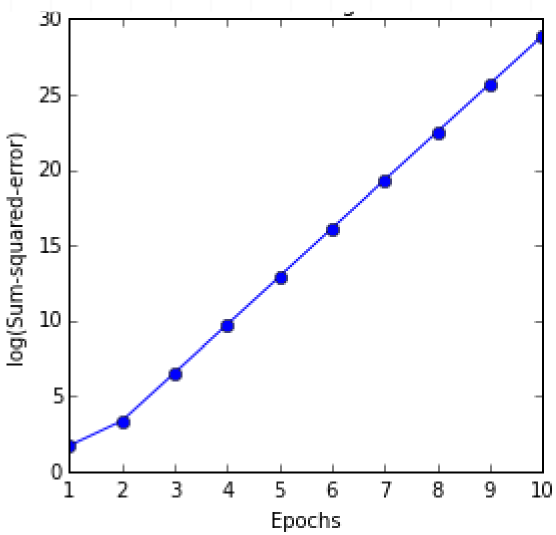
Learning Rate

- Learning rate is very important

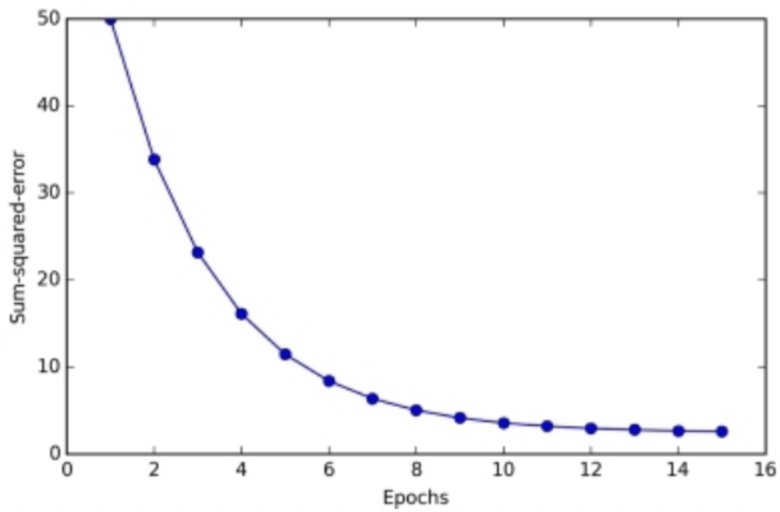


Preprocessing

- Preprocessing matters
  - Standardize your features
  - Hint: look at Adaline equation

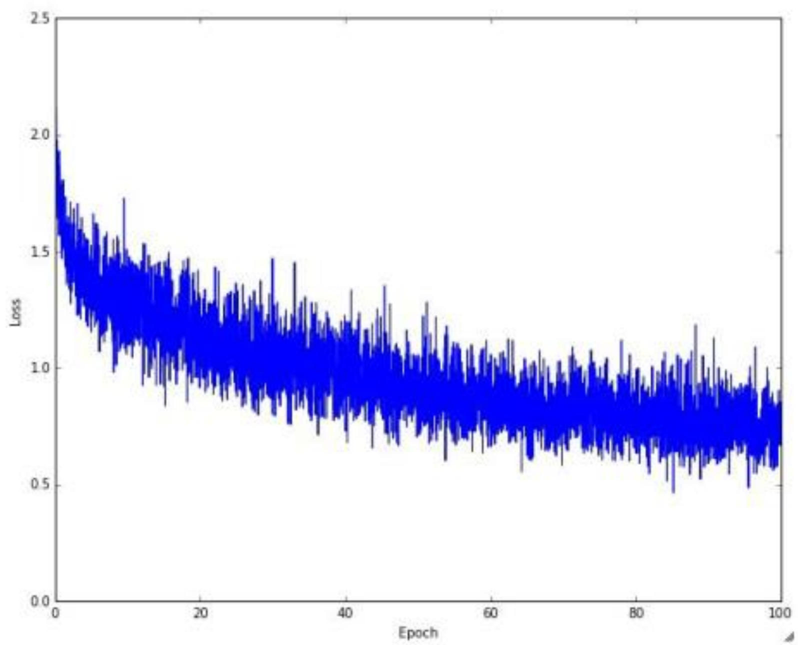


Learning rate 0.01 before standardizing



Learning rate 0.01 after standardizing

Loss function Over Time

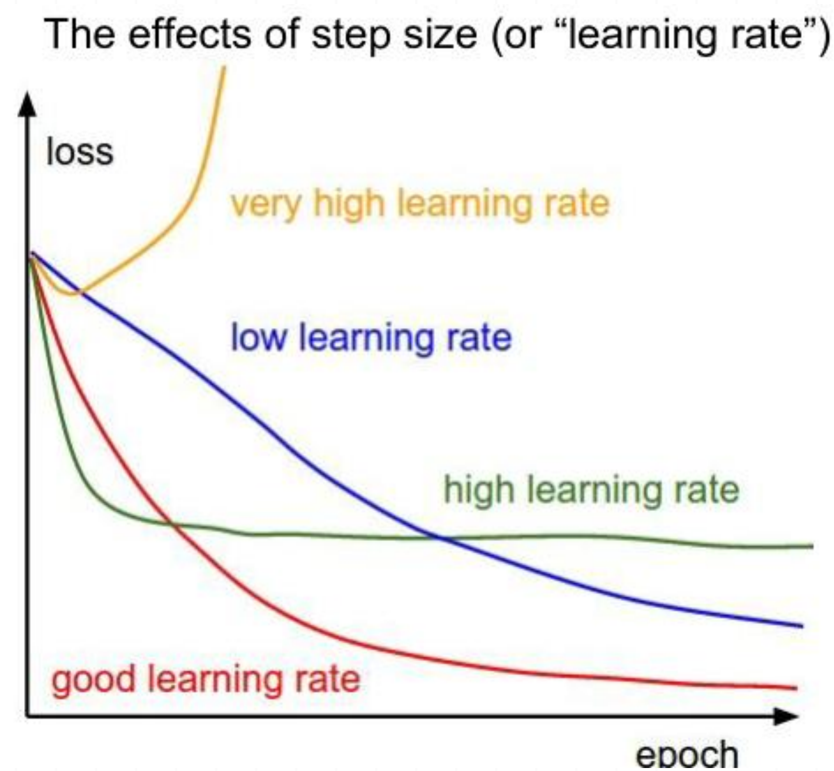


Example of optimization progress while training a neural network.

(Loss over mini-batches goes down over time.)

Learning Rate

- Loss decrease over time



Wait a minute ..

- Why we are looking at these super-old models?
  - The basis for modern neural networks

Stochastic Gradient Descent

- Alternative to batch gradient descent
- Batch gradient descent is great, but we might have millions of data points
  - Very costly computation before making a single step towards the global minimum
- Instead of updating the weights based on the sum of the accumulated errors over all samples, we update the weights incrementally for each training sample.

$$\eta\left(y^{(i)} - \phi\left(z^{(i)}\right)\right)\boldsymbol{x}^{(i)}$$

Stochastic Gradient Descent

- It typically reaches convergence much faster because of the more frequent weight updates.
- Since each gradient is calculated based on a single training example, the error surface is noisier than in gradient descent,
  - But can also have the advantage that stochastic gradient descent can escape shallow local minima more readily.

Stochastic Gradient Descent

- Order of examples matters, so shuffle your examples in each epoch.

- Stochastic gradient descent can be use for online learning. In online learning, our model is trained on-the-fly as new training data arrives.

Mini-Batch

- A compromise between batch gradient descent and stochastic gradient descent is the so-called mini-batch learning.
- applying batch gradient descent to smaller subsets of the training data— for example, 50.

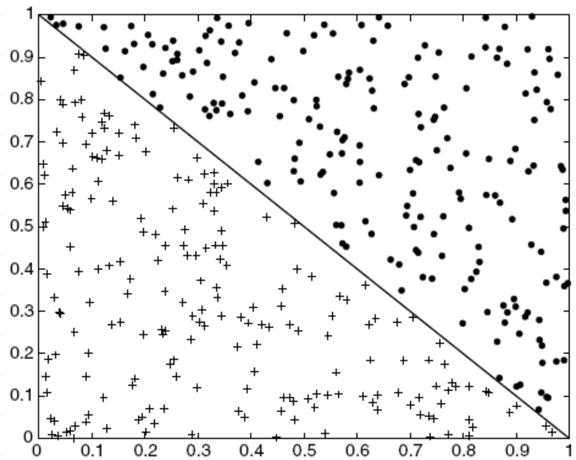
```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Epoch

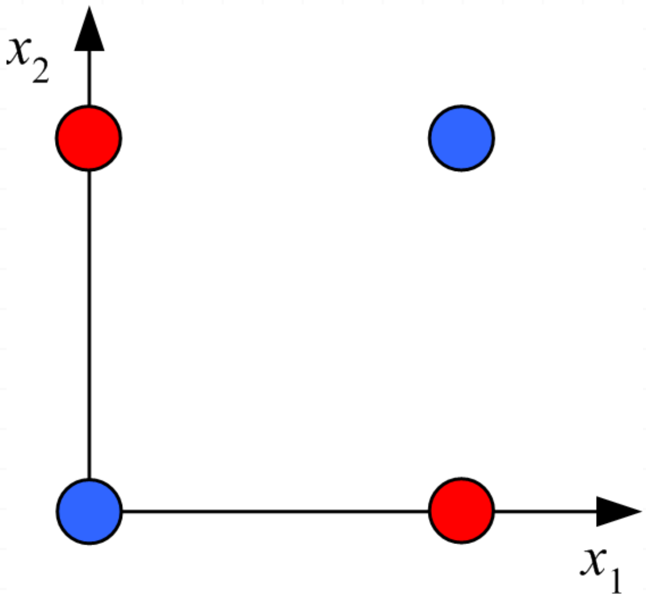
- Epoch
  - One round of updating the model for the entire training dataset
- Iteration
  - One round of updating the model for the number of examples in the batch set

Perceptron/Adaline Decision Boundary

- It is a linear classifier



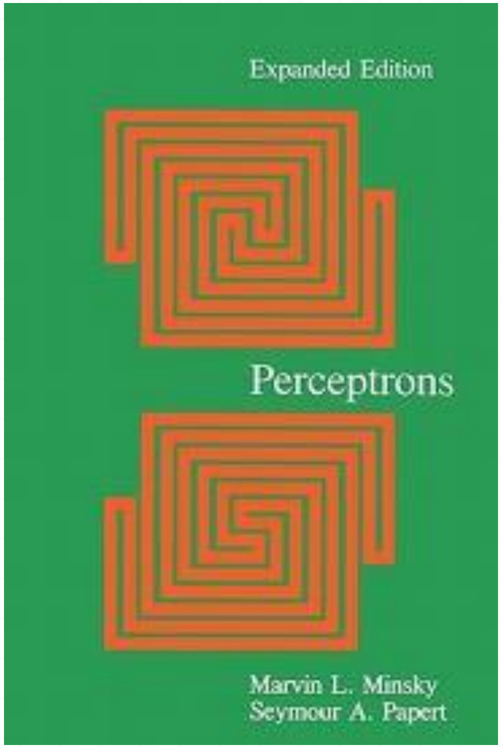
Yes! We can learn this with a linear classifier.



Impossible to learn with a linear classifier (XOR).

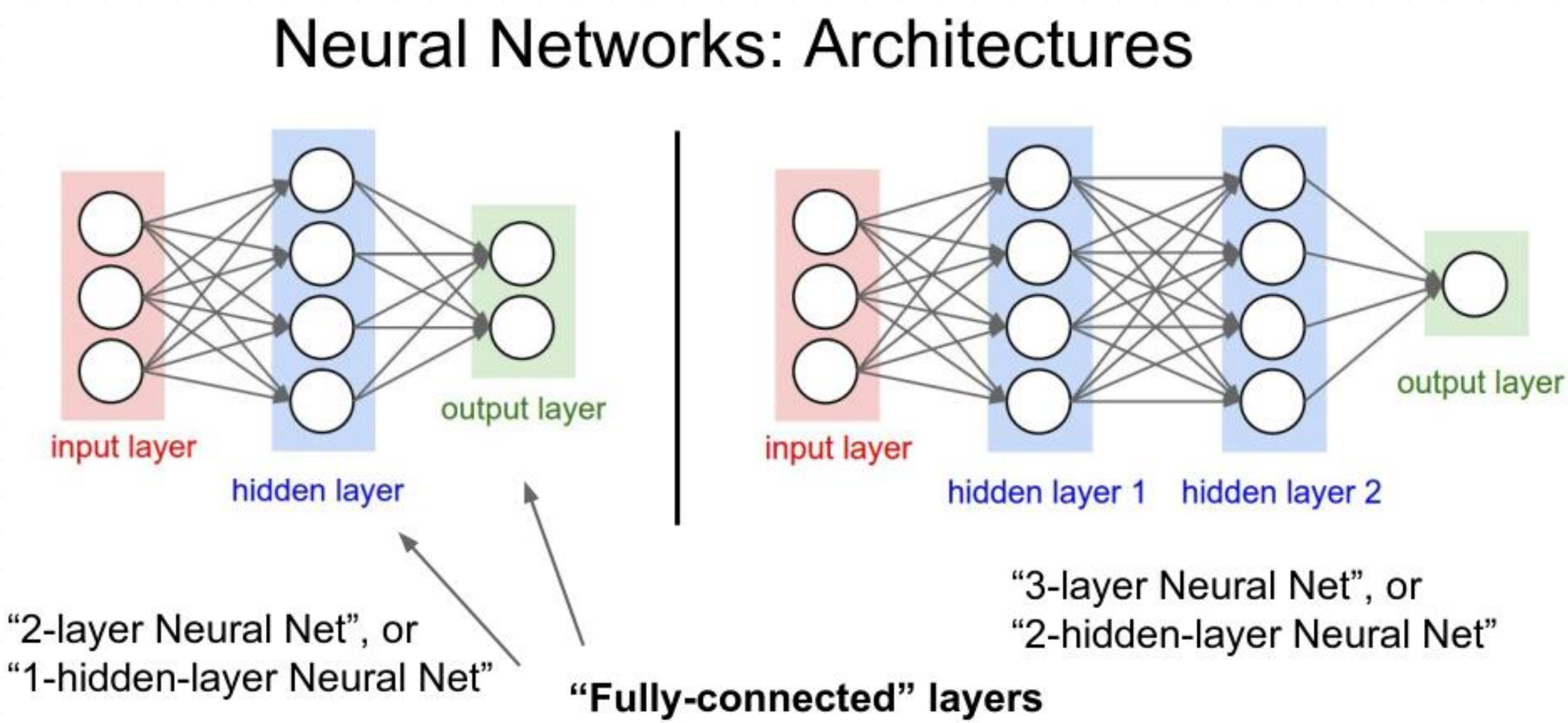
AI Winter

- Pessimistic predictions made by the authors
  - Change of direction to symbolic systems



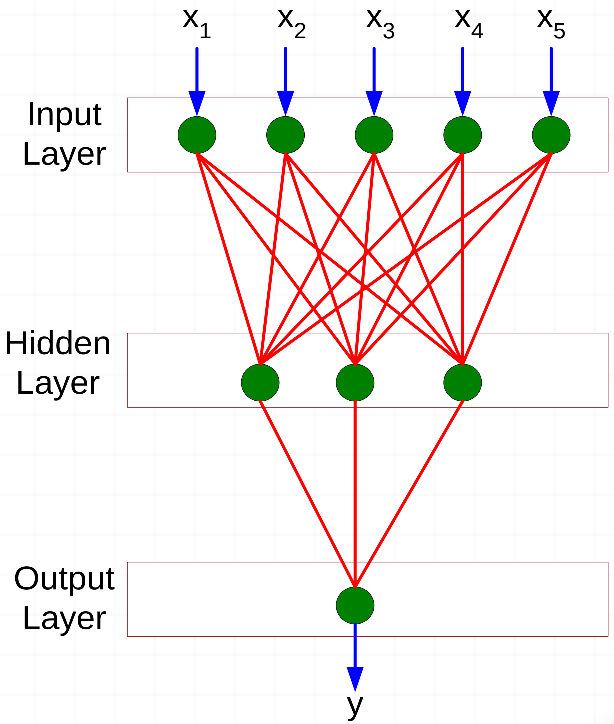
Modern Fully Connected Neural Networks

Fully Connected



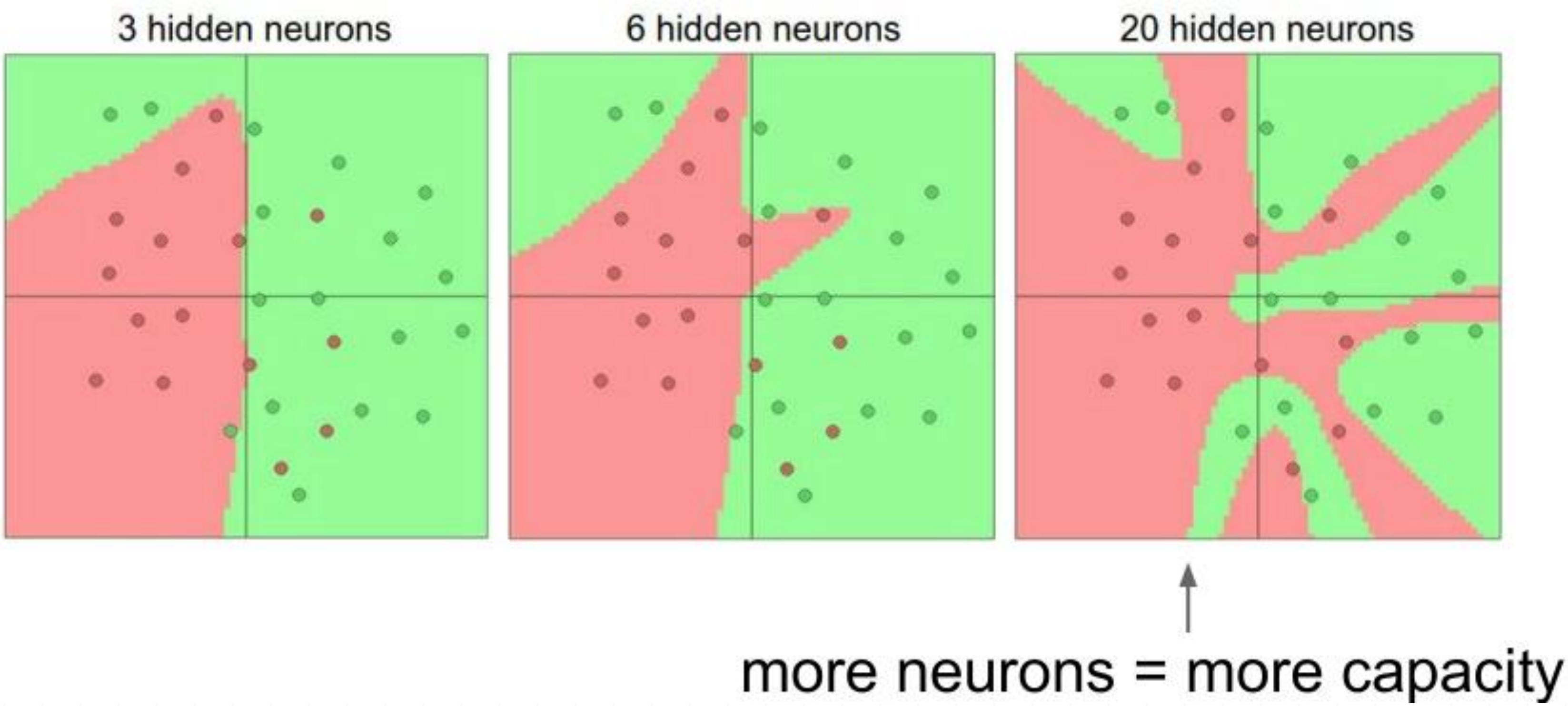
Role of Hidden units

- A linear combination of nonlinear functions
- Transforming from a d-dimensional space to an H-dimensional space
  - i.e. creating new latent features



Complexity

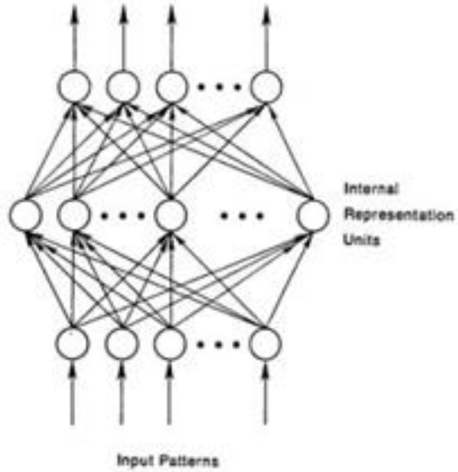
Setting the number of layers and their sizes



Multiple Layers

- A single-layer perceptron Can only predict linear functions
- Multilayer layers
  - Can predict nonlinear functions
  - Is trained using the back-propagation technique

BACKPROPAGATION - 1986



To be more specific, then, let

$$E_p = \frac{1}{2}\sum_j (t_{pj} - o_{pj})^2 \quad (2)$$

be our measure of the error on input/output pattern p and let $E = \sum E_p$ be our overall measure of the error. We wish to show that the delta rule implements a gradient descent in E when the units are linear. We will proceed by simply showing that

$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} i_{pi},$$

which is proportional to $\Delta_p w_{ji}$ as prescribed by the delta rule. When there are no hidden units it is straightforward to compute the relevant derivative. For this purpose we use the chain rule to write the derivative as the product of two parts: the derivative of the error with respect to the output of the unit times the derivative of the output with respect to the weight.

$$\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial w_{ji}}. \quad (3)$$

The first part tells how the error changes with the output of the jth unit and the second part tells how much changing w_{ji} changes that output. Now, the derivatives are easy to compute. First, from Equation 2

$$\frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj}) = -\delta_{pj}. \quad (4)$$

Not surprisingly, the contribution of unit u_j to the error is simply proportional to δ_{pj}. Moreover, since we have linear units,

$$o_{pj} = \sum_i w_{ji} i_{pi}, \quad (5)$$

from which we conclude that

$$\frac{\partial o_{pj}}{\partial w_{ji}} = i_{pi}.$$

Thus, substituting back into Equation 3, we see that

$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} i_{pi} \quad (6)$$

Rumelhart, Hinton, Williams (1986)

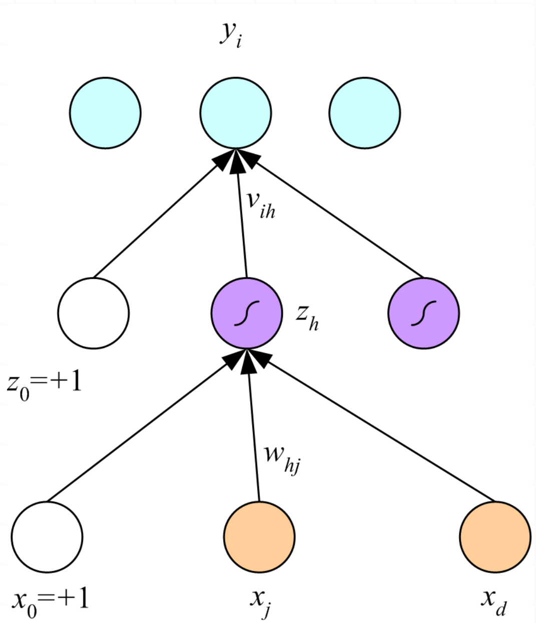
- Proposed by Rumelhart, Hinton, and Williams
- Paved the way for training deep networks

How it works …

- Mini-batch SGD
- Loop:
  - 1. Sample a batch of data
  - 2. Forward prop it through the graph, get loss
  - 3. Backprop to calculate the gradients
  - 4. Update the parameters using the gradient

Training MLP: Back-propagation

- The error propagates from output y back to the inputs and hence the name back-propagation.
- Function composition



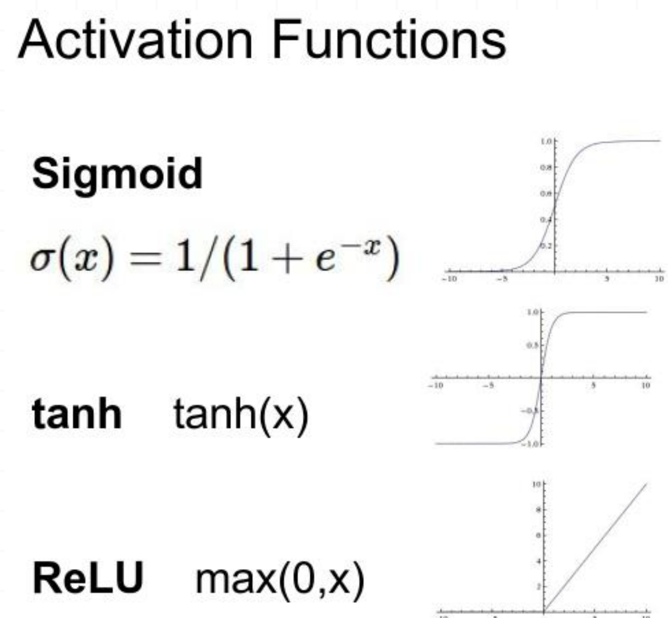
E is the error (i.e. the difference between desired and actual output)

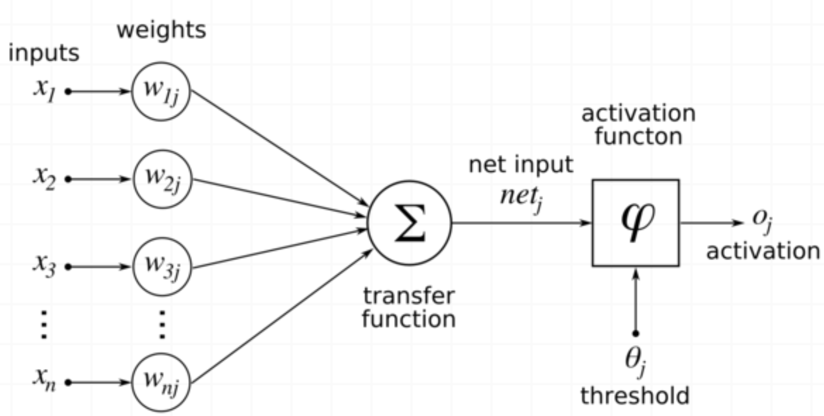
$$\frac{\partial E}{\partial w_{hj}} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial z_h}\frac{\partial z_h}{\partial w_{hj}}$$

Chain Rule

Activation Function

- Initially many were using Sigmoid since it is the differentiable version of threshold (step function)
- This days we usually use ReLU for hidden layers and sigmoid in the output layer



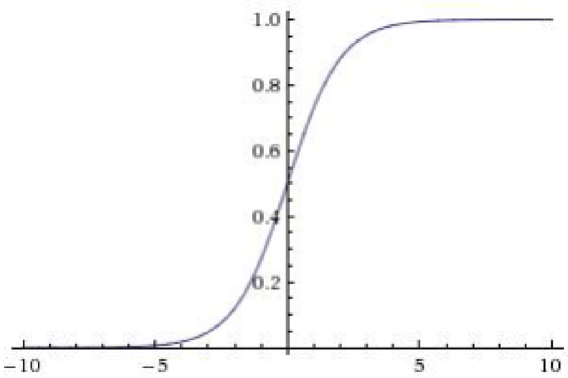


Activation Function

Activation Functions

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

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

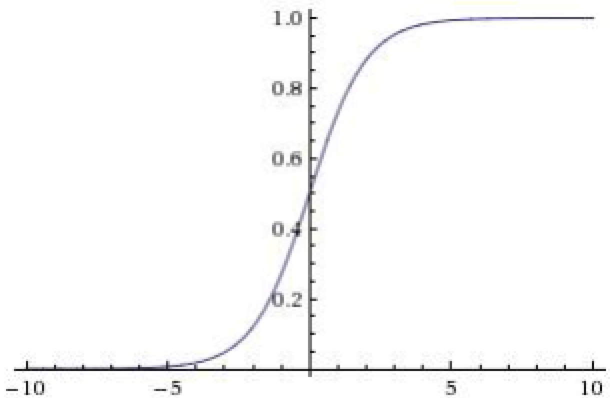


Sigmoid

Activation Function

Activation Functions

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



Sigmoid

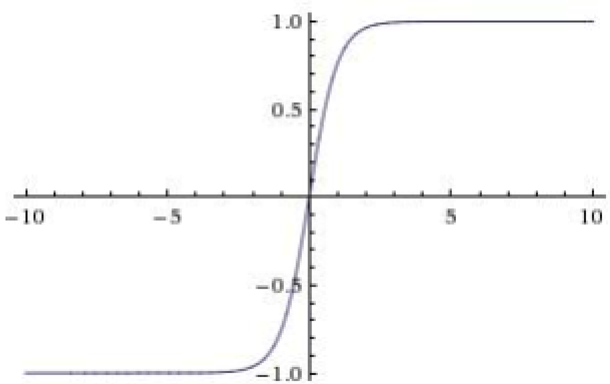
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients

Activation Function

Activation Functions



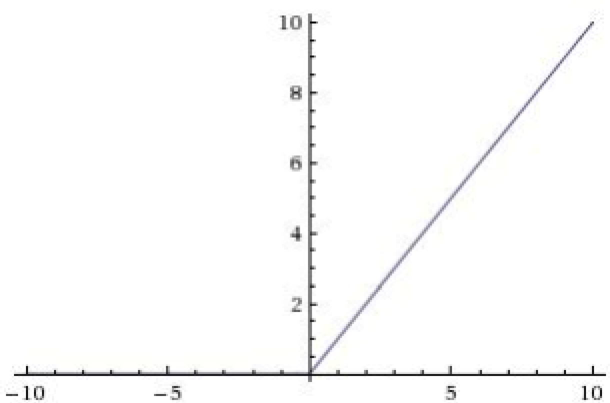
tanh(x)

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

[LeCun et al., 1991]

Activation Function

Activation Functions



ReLU
(Rectified Linear Unit)

- Computes **f(x) = max(0,x)**

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

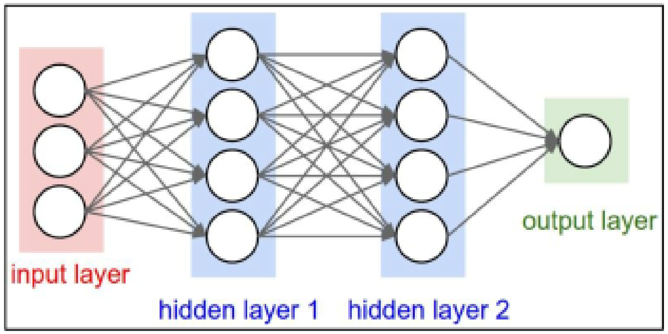
[Krizhevsky et al., 2012]

NN Training

Mini-batch SGD

Loop:

1. **Sample** a batch of data
2. **Forward** prop it through the graph, get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient



Simple NN Code

Training a neural network, main loop:

```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
```

Algorithm for learning ANN

- Standardize your data
- Initialize the weights (w_0, w_1, ..., w_k)
  - Important how to initialize

- Compute the direction/magnitude in which each parameter needs to be changed
  - Mini-Batch mode

Layers

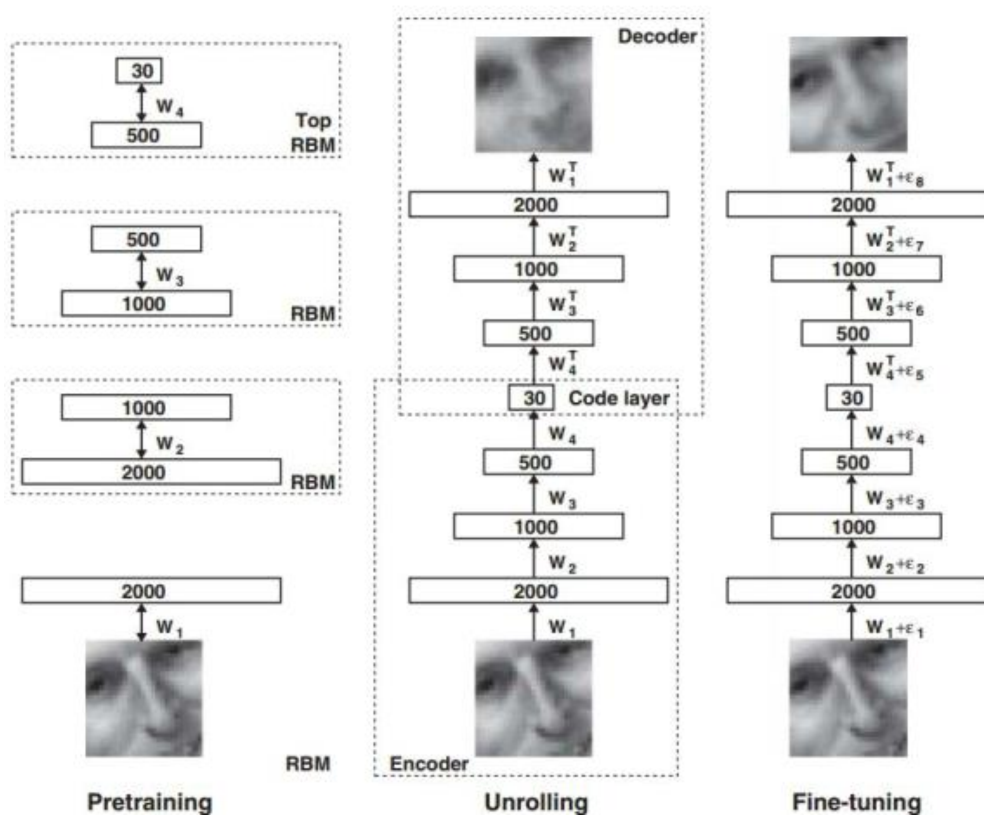
- How many layers?
  - Very hard question
  - Mostly based on heuristic and best practice
  - Trial and error experimentation

DEEP LEARNING IS REBORN

REIGNITED DEEP LEARNING IN 2006

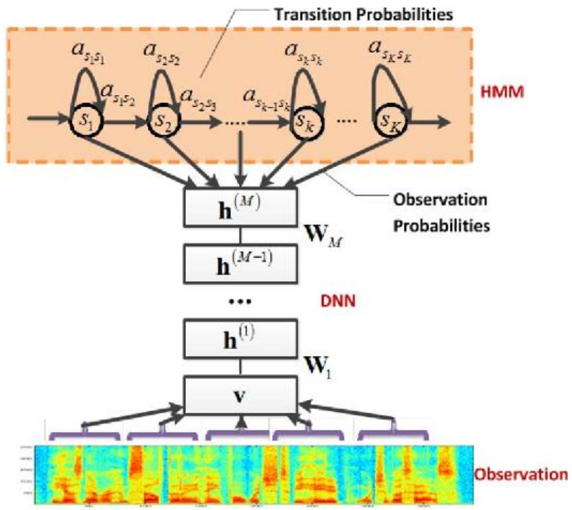
Ruslan Salakhutdinov and Geoffrey Hinton, 2006



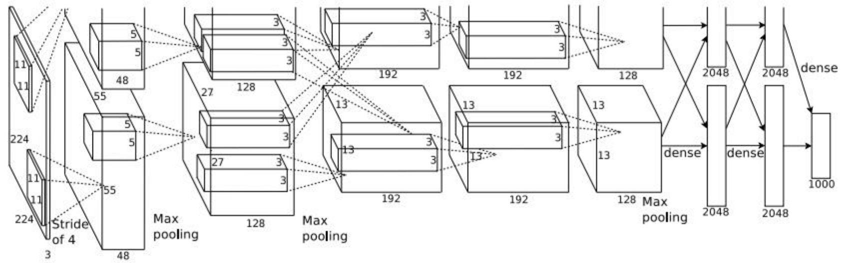
An Efficient Learning Procedure for Deep Boltzmann Machines. Ruslan Salakhutdinov and Geoffrey Hinton. Neural Computation August 2012, Vol. 24, No. 8: 1967 -- 2006.

FIRST IMPRESSIVE RESULTS

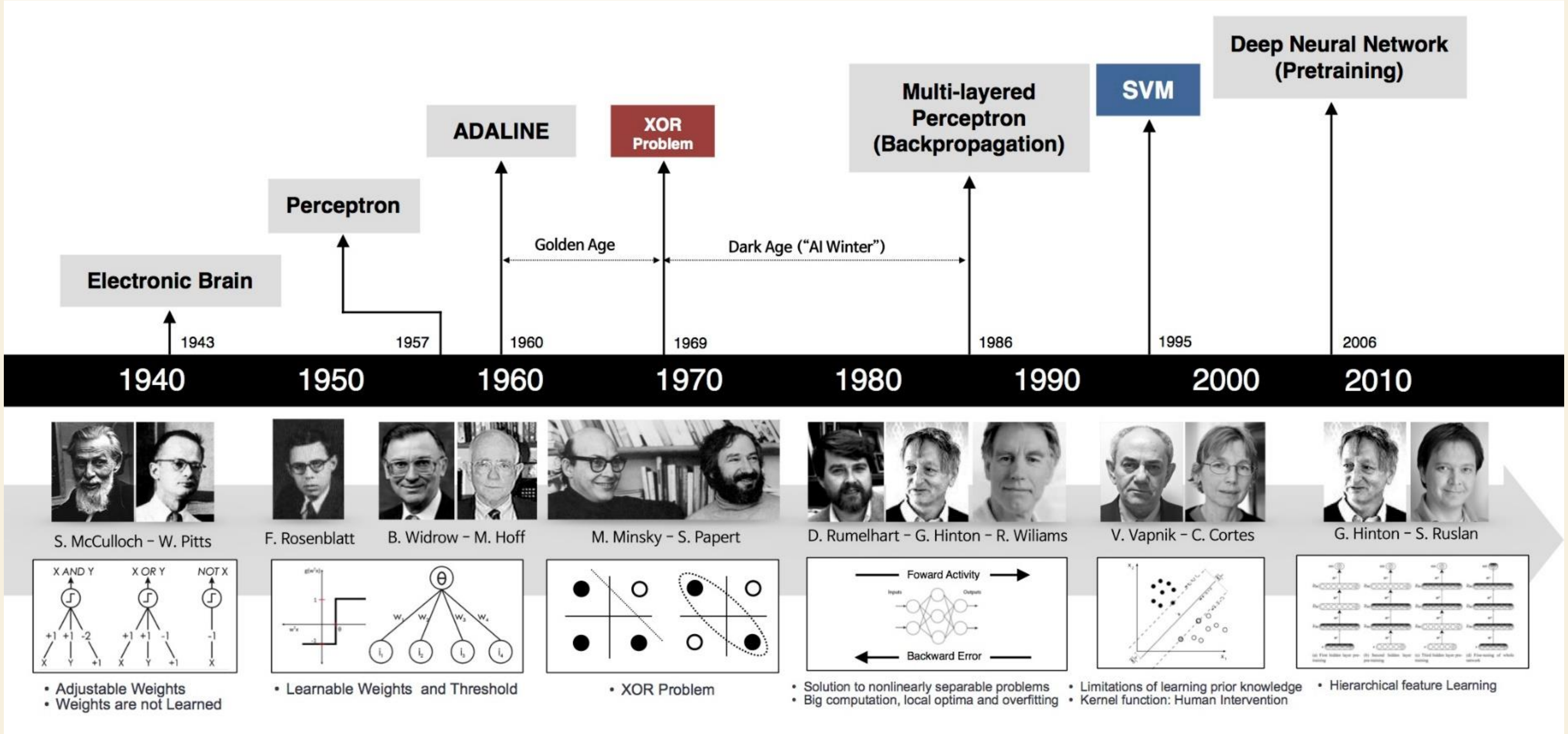
STARTING IN 2010-2012



Dahl, George E., Dong Yu, Li Deng, and Alex Acero. "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition." IEEE Transactions on audio, speech, and language processing 20, no. 1 (2012): 30-42.



Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." In Advances in neural information processing systems, pp. 1097-1105. 2012.



NEURAL NETWORKS HISTORY

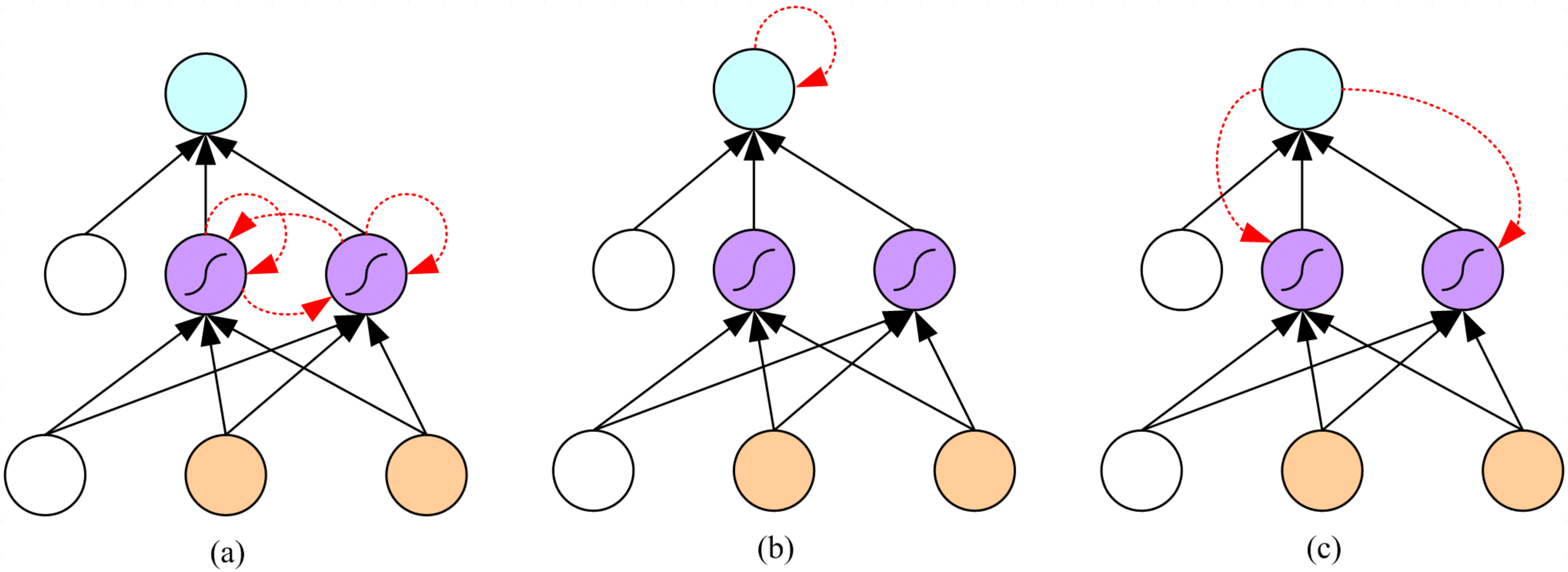
GOES BACK TO 1940, WITH SEVERAL DARK AI WINTERS

Image from https://beamandrew.github.io/deeplearning/2017/02/23/deep_learning_101_part1.html

Learning in Time

- Applications:
  - Sequence recognition: Speech recognition
  - Sequence reproduction: Time-series prediction
  - Sequence association
- Network architectures
  - Time-delay networks (Waibel et al., 1989)
  - Recurrent networks (Rumelhart et al., 1986)

Recurrent Networks



Keras

- A minimalist Python library for deep learning
  - Can work with CPUs and GPUs
  - Can use Theano, CNTK, or TensorFlow as backend
  - Keras is compatible with: Python 2.7-3.6.

Keras

- You have to install Keras on your machine
- Before installing Keras, install one of its backend engines: TensorFlow, Theano, or CNTK.
  - TensorFlow backend is recommended

https://keras.io/#installation

Keras

1. Define: Typically, first you need to define your model as a sequence of layers called Sequential
2. Compile: Once defined, you need to compile your model
   - This step uses other libraries to optimize the computations
3. Fit: Once compiled, model can be fit to data
   - The actual computations happens here
4. Predict: After training, we can use it to make predictions

Keras

1. Define: create a Sequential model
2. Compile: Sepcify loss function and optimizers and call compile()
3. Fit: call fit()
4. Predict: After training, call predict()

Keras

- Fully connected layers are defined using the Dense class

Data Preparation

- Usual preparation: everything needs to be a number (e.g. use one-hot encoding), scale your data

Let’s see some code

- Notebook on Canvas

In Summary ..

- NN with at least 1 hidden layer are universal approximators
  - Over-fitting
- The topology should be chosen (not easy!)
- Weights should be initialized
- Sensitive to noise
- Local minima
- Training is time consuming