Deep Learning

Ali Ghodsi

University of Waterloo

Deep Learning

Deep learning attempts to learn representations of data with multiple levels of abstraction. Deep learning usually refers to a set of algorithms and computational models that are composed of multiple processing layers. These methods have significantly improved the state-of-the-art in many domains including, speech recognition, classification, pattern recognition, drug discovery, and genomics.

Success Stories

Deep Learning Machine Teaches Itself Chess in 72 Hours, Plays at International Master Level.

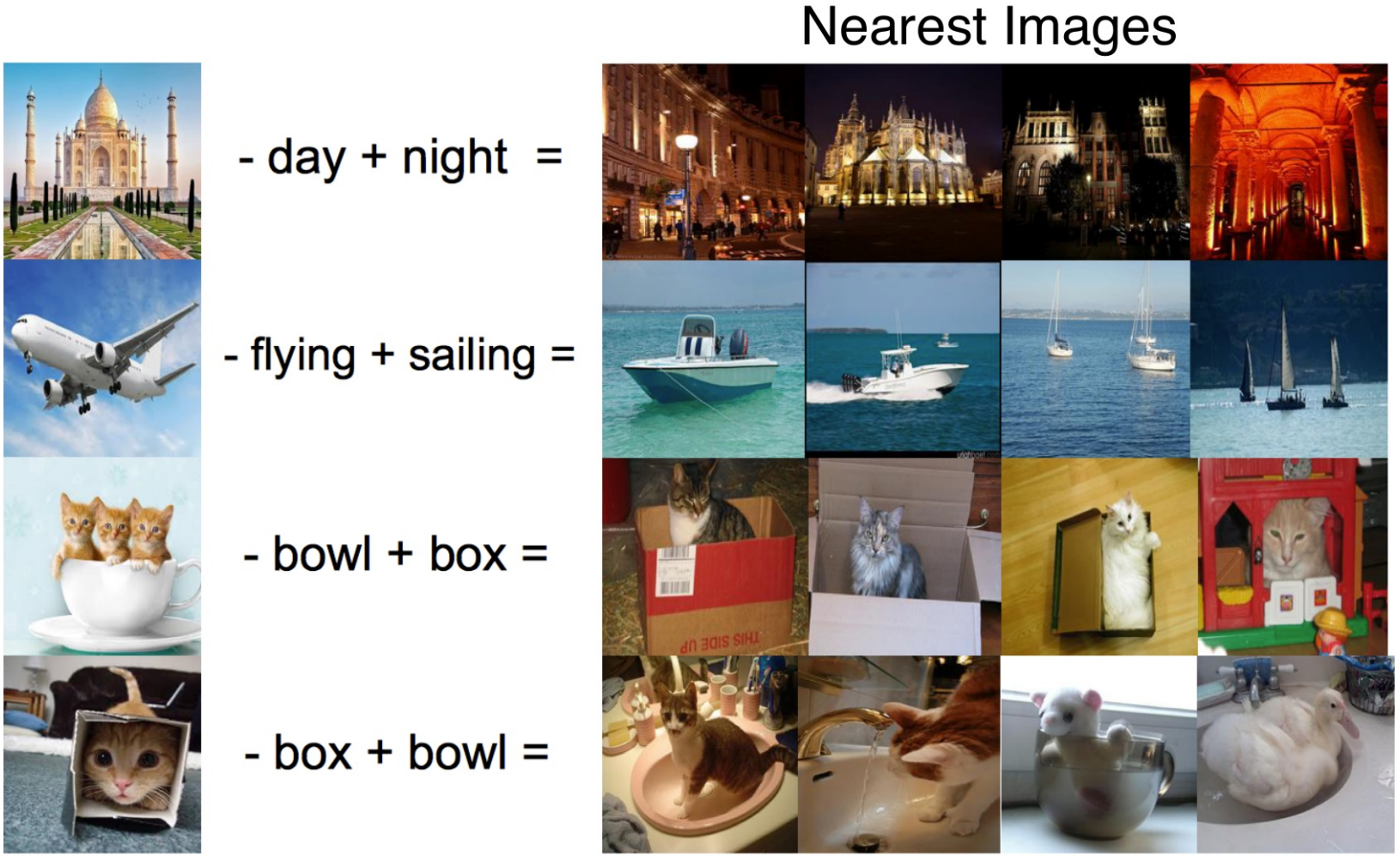
An artificial intelligence machine plays chess by evaluating the board rather than using brute force to work out every possible move.

Success Stories

Word2vec , Mikolov, 2013.

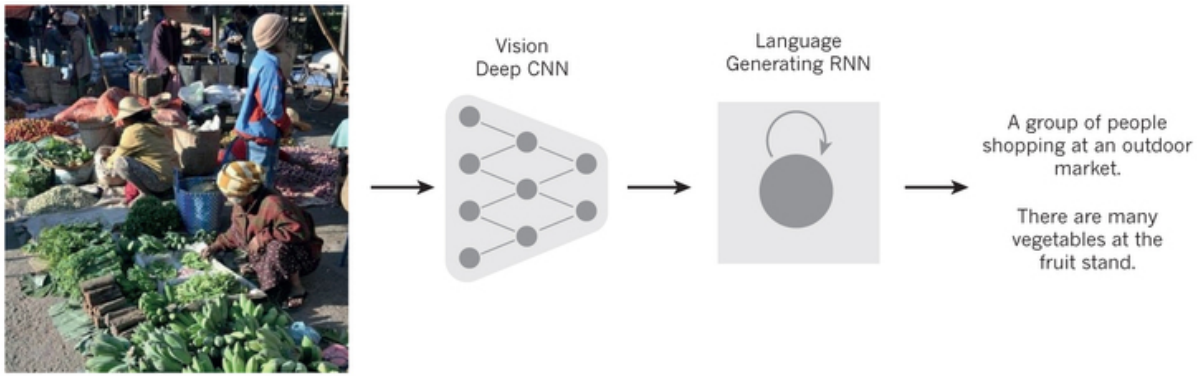
$$\text{king} - \text{man} + \text{woman} = \text{queen}$$

Success Stories



(Kiros, Salakhutdinov, Zemel, TACL 2015)

Success Stories




A woman is throwing a **frisbee** in a park.


A **dog** is standing on a hardwood floor.

A **stop** sign is on a road with a mountain in the background


A little **girl** sitting on a bed with a teddy bear.

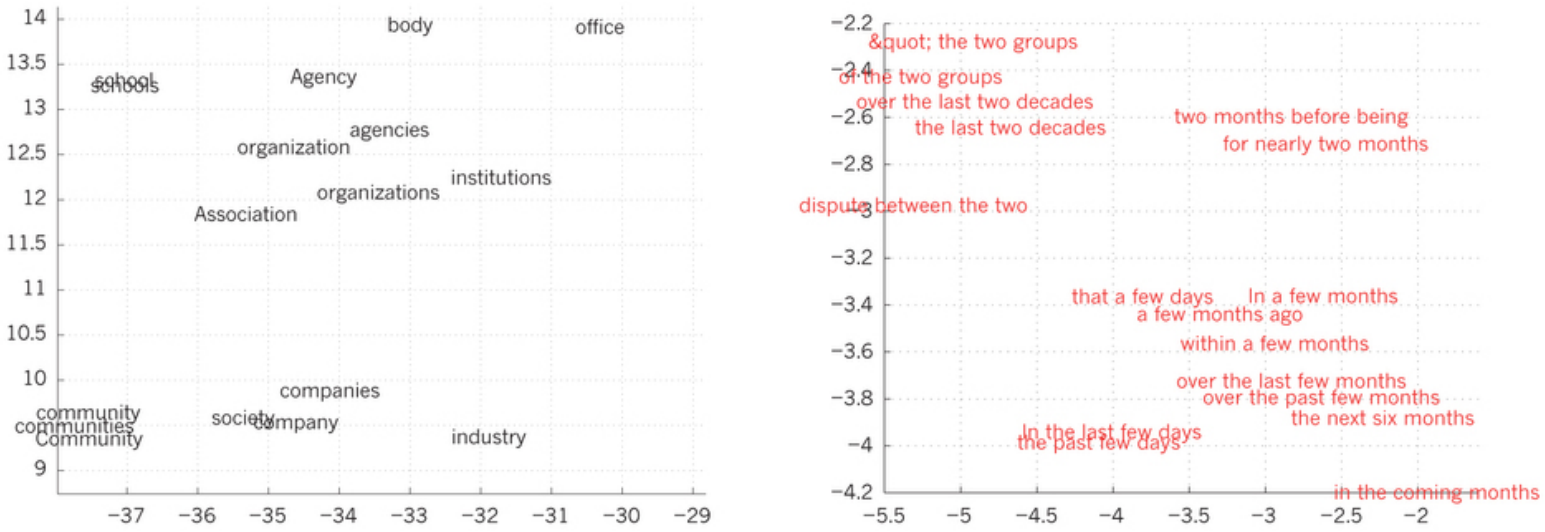

A group of **people** sitting on a boat in the water.


A giraffe standing in a forest with **trees** in the background.

Vinyals et. al 2014

Captions generated by a recurrent neural network.

Success Stories



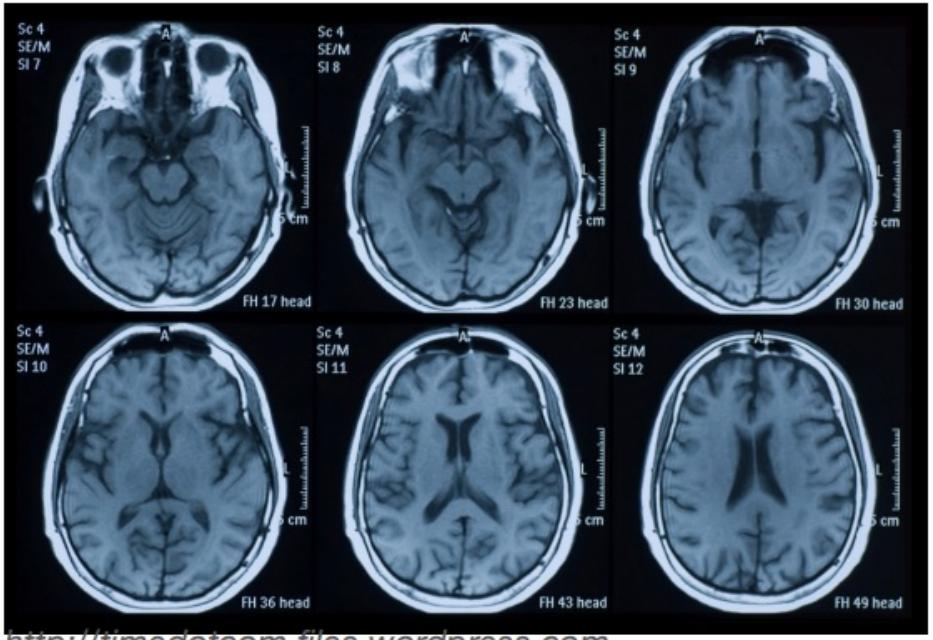
Credit: LeCun, et. al., 2015, Nature

On the left is an illustration of word representations learned for modelling language, non-linearly projected to 2D for visualization using the t-SNE algorithm. On the right is a 2D representation of phrases learned by an English-to-French encoder–decoder recurrent neural network. One can observe that semantically similar words or sequences of words are mapped to nearby representations.

Success Stories

PayPal is using deep learning via H2O, an open source predictive analytics platform, to help prevent fraudulent purchases and payment transactions.

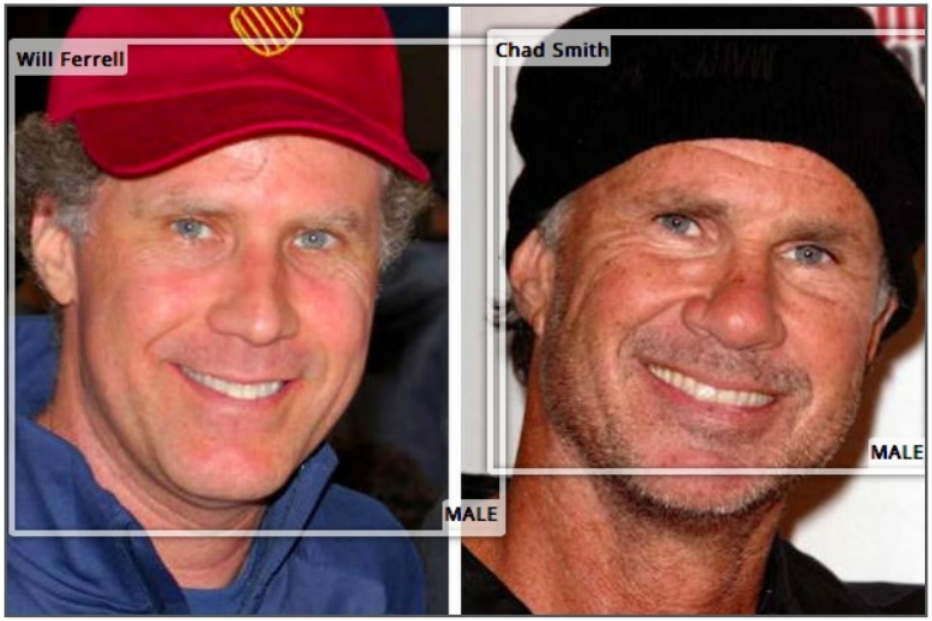
Success Stories



http://timedotcom.files.wordpress.com

New startup Enlitic is using deep learning to process X-rays, MRIs, and other medical images to help doctors diagnose and treat complicated diseases. Enlitic uses deep learning algorithms that "are suited to discovering the subtle patterns that characterize disease profiles."

Success Stories



Credit: Hansen, 2014

AlchemyVision's Face Detection and Recognition service is able to distinguish between look-alikes such as actor Will Ferrell and Red Hot Chili Peppers' drummer, Chad Smith.

Recovering sound waves from the vibrations

Davis, A., Rubinstein, M., Wadhwa, N., Mysore, G., Durand, F., and Freeman, W. T.(2014). The visual microphone: Passive recovery of sound from video. ACM Transac-tions on Graphics (Proc. SIGGRAPH), 33(4), 79:1–79:10.

▸ The visual microphone

Demos

http://deeplearning.net/demos/

Tentative topics:

- Feedforward Deep Networks
- Optimization for Training Deep Models
- Convolutional Networks
- Sequence Modeling: Recurrent and Recursive Nets
- Auto-Encoders
- Representation Learning
- Restricted Boltzmann Machines
- Deep Generative Models
- Deep learning for Natural Language Processing

Tentative Marking Scheme

Group Project 50%

Paper critiques 30%

Paper presentation 20%

Communication

All communication should take place using the Piazza discussion board.

You will be sent an invitation to your UW email address. It will include a link to a web page where you may complete the enrollment process.

Note 1: You cannot borrow part of an existing thesis work, nor can you re-use a project from another course as your final project.

Note 2: We will use wikicoursenote for paper critiques and possibly for course note (details in class).

History, McCulloch and Pitts network

1943

The first model of a neuron was invented by McCulloch (physiologists) and Pitts (logician).

The model had two inputs and a single output.

A neuron would not activate if only one of the inputs was active.

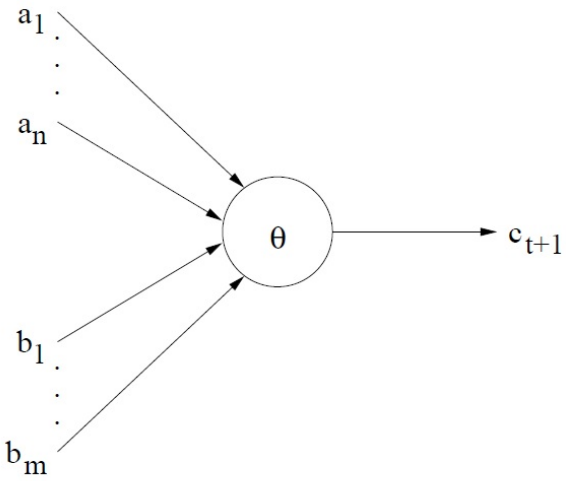
The weights for each input were equal, and the output was binary.

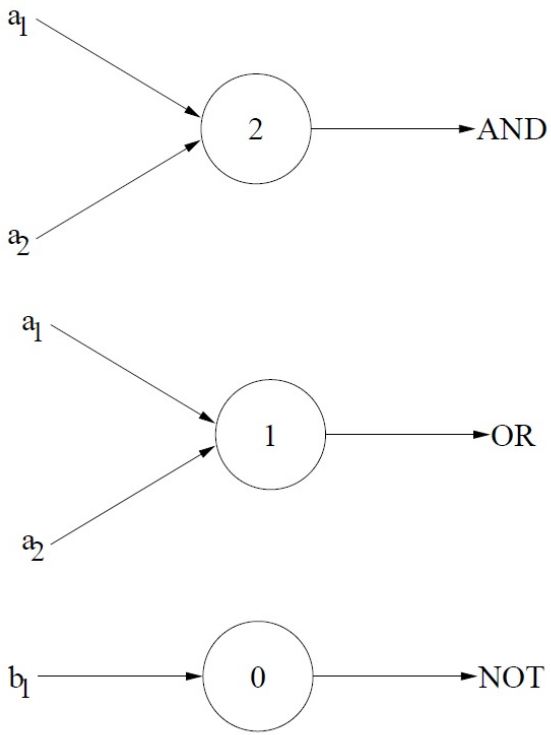
Until the inputs summed up to a certain threshold level, the output would remain zero.

The McCulloch and Pitts' neuron has become known today as a logic circuit.

History, McCulloch and Pitts network (MPN)

1943



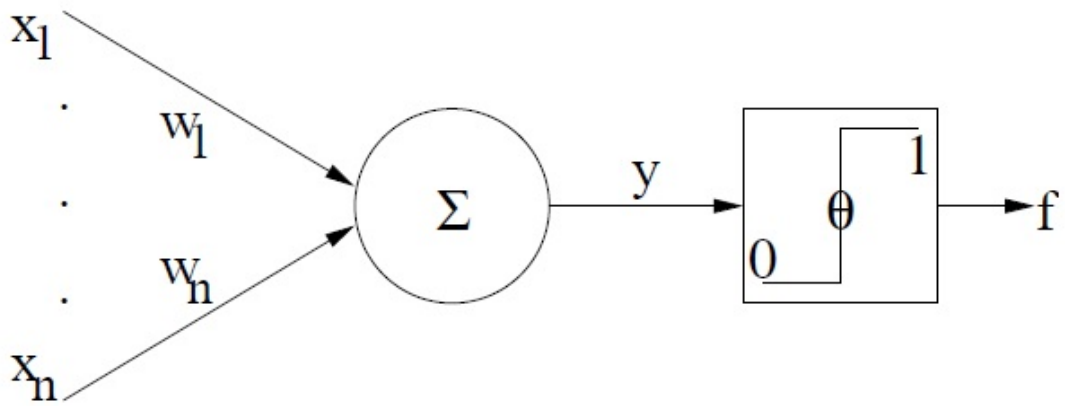


logic functions can be modeled by a network of MP-neurons

History, Perceptron

1958

The perceptron was developed by Rosenblatt (physiologist).



Credit:

Perceptron, the dream

1958

Rosenblatt randomly connected the perceptrons and changed the weights in order to achieve "learning."

Based on Rosenblatt's statements in a press confrence in 1958, The New York Times reported the perceptron to be
'the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.'

MPN vs Perceptron

Apparently McCulloch and Pitts' neuron is a better model for the electrochemical process inside the neuron than the perceptron.

But perceptron is the basis and building block for the modern neural networks.

History, obtimization

1960

Widrow and Hoff proposed a method for adjusting the weights. They introduced a gradient search method based on minimizing the error squared (Least Mean Squares).

In the 1960's, there were many articles promising robots that could think.

It seems there was a general belief that perceptrons could solve any problem.

History, shattered dream

1969

Minsky and Papert published their book *Perceptrons*. The book shows that perceptrons could only solve linearly separable problems.

They showed that it is not possible for perseptron to learn an XOR function.

After Perceptrons was published, researchers lost interest in perceptron and neural networks.

History, obtimization

1969

Arthur E. Bryson and Yu-Chi Ho described proposed Backpropagation as a multi-stage dynamic system optimization method. (Bryson, A.E.; W.F. Denham; S.E. Dreyfus. Optimal programming problems with inequality constraints. I: Necessary conditions for extremal solutions. AIAA J. 1, 11 (1963) 2544-2550)

1972

Stephen Grossberg proposed networks capable of learning XOR function.

History

1974

Backpropagation was reinvented / applied in the context of neural networks by Paul Werbos, David E. Rumelhart, Geoffrey E. Hinton and Ronald J. Williams.

Back propagation allowed perceptrons to be trained in a multilayer configuration.

History

1980s

The filed of artificial neural network research experienced a resurgence.

2000s

Neural network fell out of favor partly due to BP limitations.

Backpropagation Limitations

It requires labeled training data.

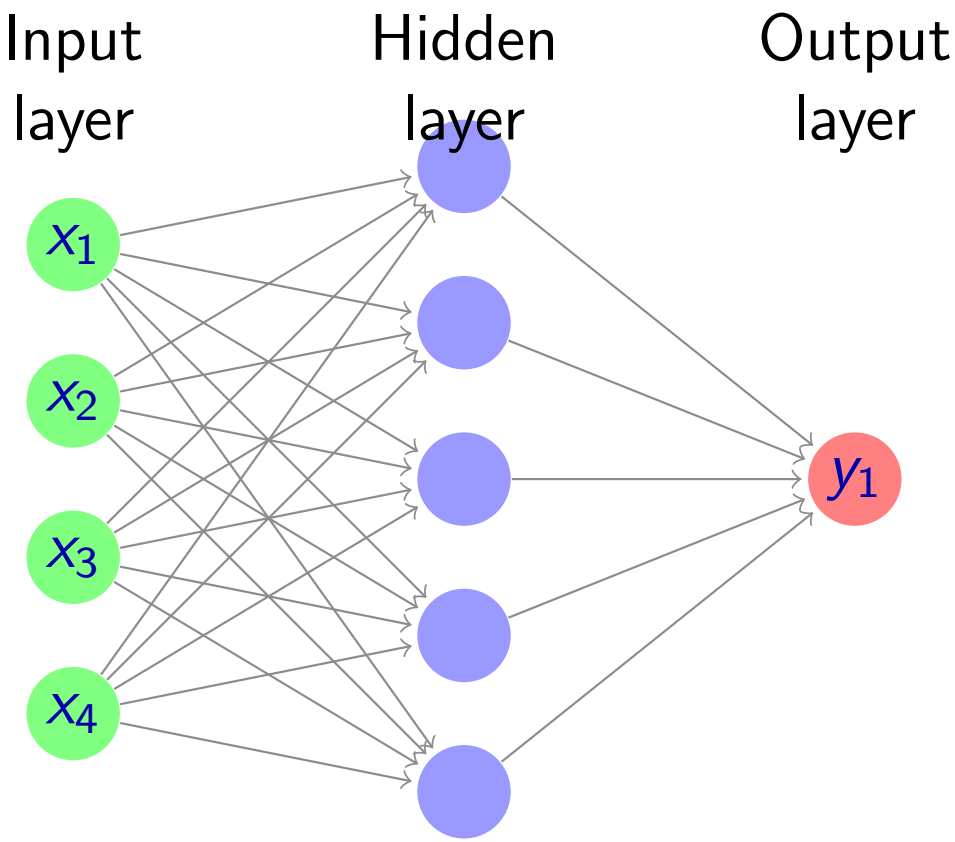
It is very slow in networks with multiple layers (doesn't scale well).

It can converge to poor local minima.

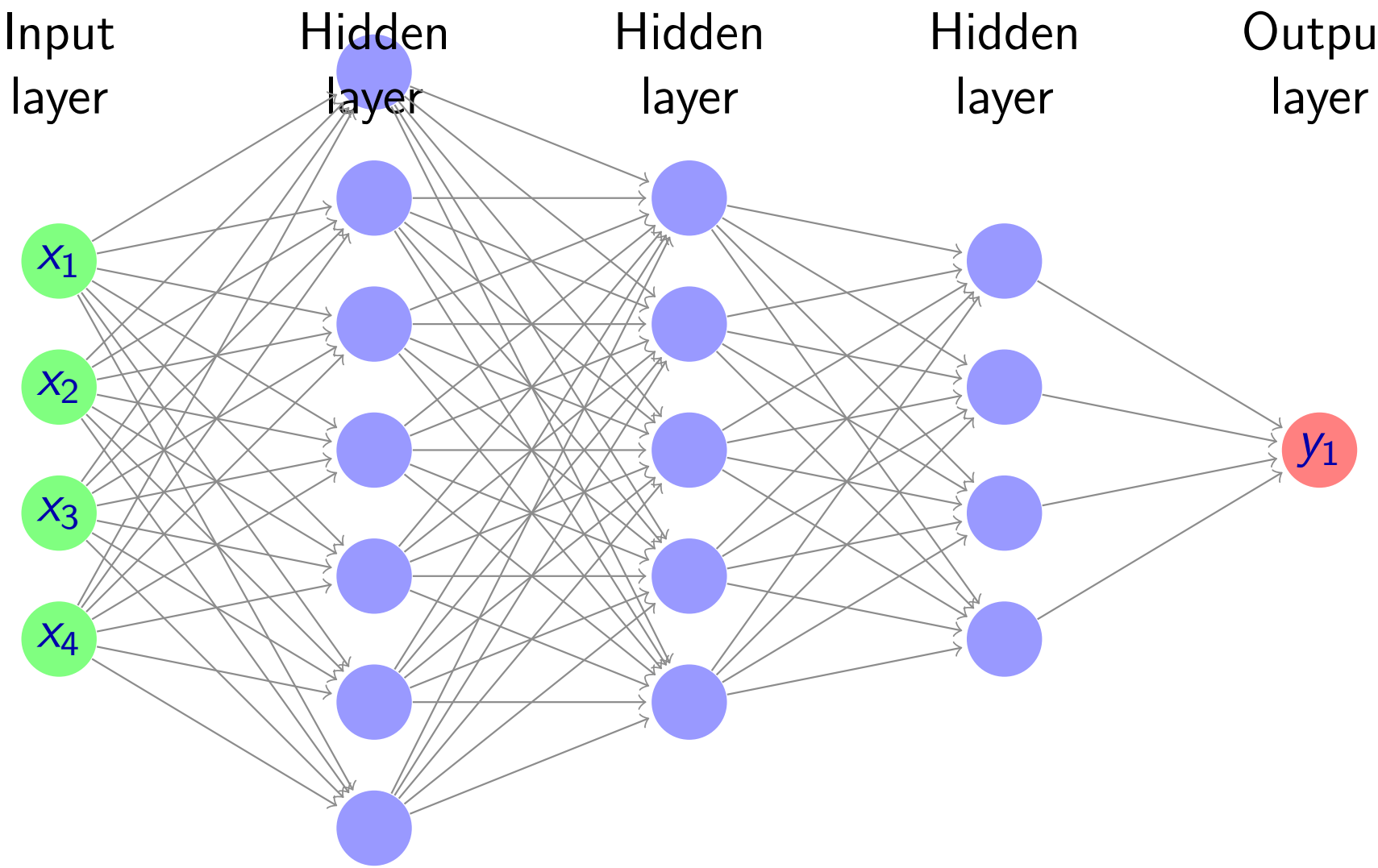
History, obtimization

but has returned again in the 2010s, now able to train much larger networks using huge modern computing power such as GPUs. For example, in 2013 top speech recognisers now use backpropagation-trained neural networks.

Feedforward NeuralNetwork



Feedforward Deep Networks

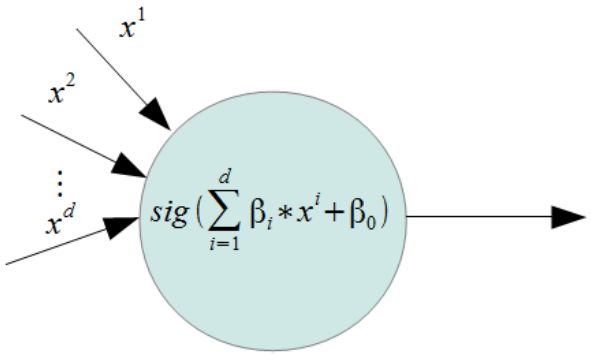


Feedforward Deep Networks

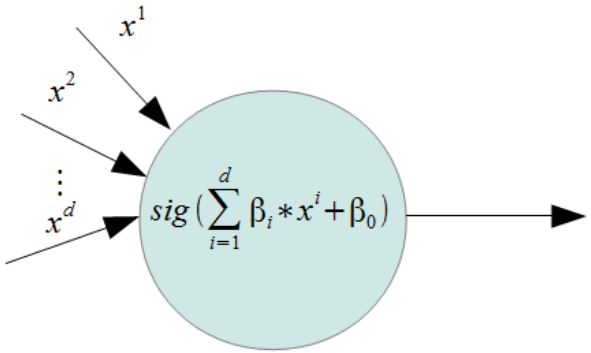
- Feedforward deep networks, a.k.a. multilayer perceptrons (MLPs), are parametric functions composed of several parametric functions.
- Each layer of the network defines one of these sub-functions.
- Each layer (sub-function) has multiple inputs and multiple outputs.
- Each layer composed of many units (scalar output of the layer).
- We sometimes refer to each unit as a feature.
- Each unit is usually a simple transformation of its input.
- The entire network can be very complex.

Perceptron

- The perceptron is the building block for neural networks.
- It was invented by Rosenblatt in 1957 at Cornell Labs, and first mentioned in the paper 'The Perceptron – a perceiving and recognizing automaton'.
- Perceptron computes a linear combination of factor of input and returns the sign.



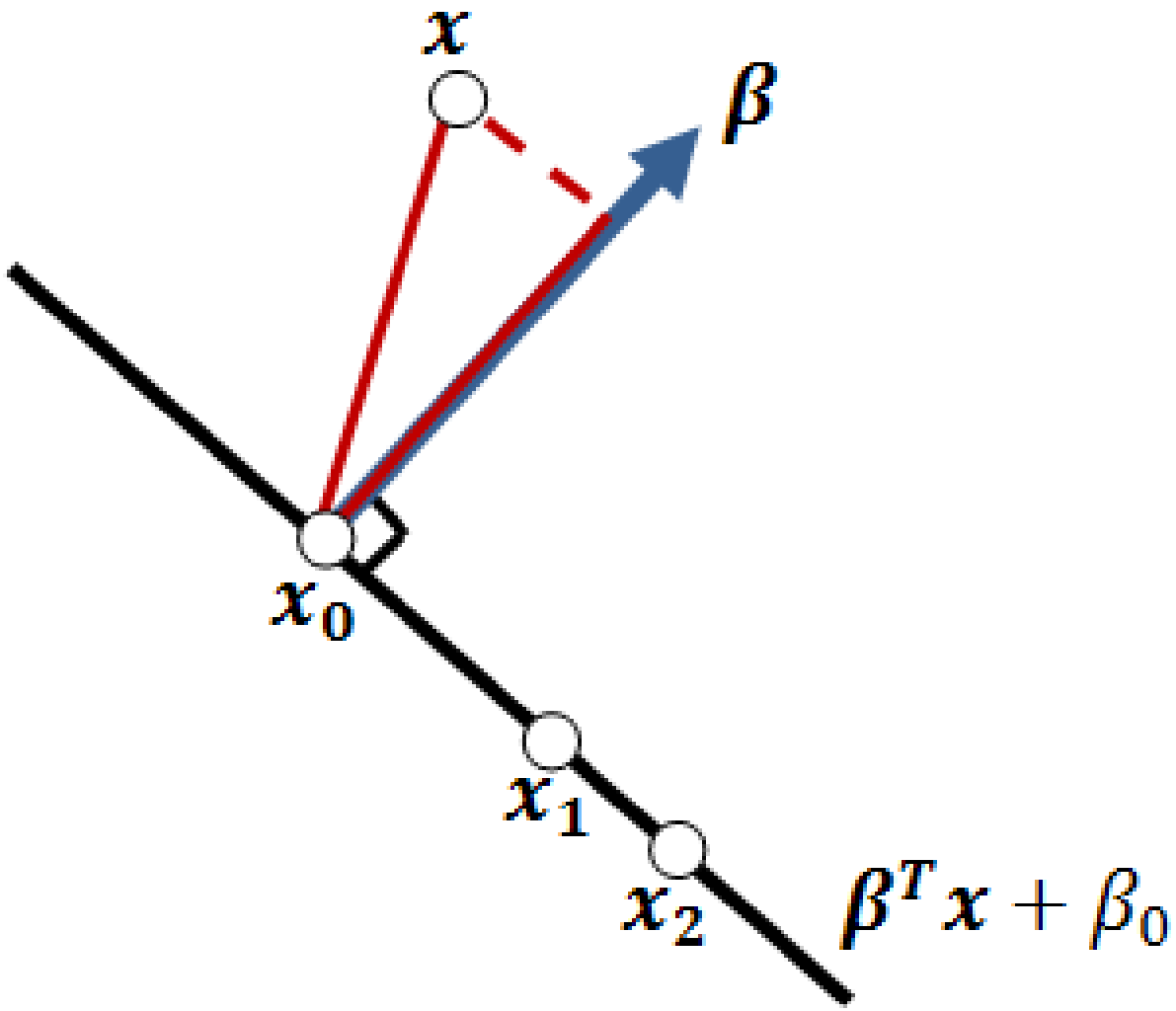
Simple perceptron



Simple perceptron

x^i is the *i*-th feature of a sample and β_i is the *i*-th weight. β_0 is defined as the bias. The bias alters the position of the decision boundary between the 2 classes. From a geometrical point of view, Perceptron assigns label "1" to elements on one side of $\beta^T x + \beta_0$ and label "-1" to elements on the other side

- define a cost function, $\phi(\boldsymbol{\beta}, \beta_0)$, as a summation of the distance between all misclassified points and the hyper-plane, or the decision boundary.
- To minimize this cost function, we need to estimate $\boldsymbol{\beta}, \boldsymbol{\beta_0}$.
  $\min_{\beta,\beta_0} \phi(\boldsymbol{\beta}, \beta_0) = \{\text{distance of all misclassified points}\}$



Distance between the point and the decision boundary hyperplane (black line).

1) A hyper-plane L can be defined as

$$L = \{x : f(x) = \beta^T x + \beta_0 = 0\},$$

For any two arbitrary points x_1 and x_2 on L , we have

$$\beta^T x_1 + \beta_0 = 0 \ ,$$

$$\beta^T x_2 + \beta_0 = 0 \ ,$$

such that

$$\beta^T (x_1 - x_2) = 0 \ .$$

Therefore, β is orthogonal to the hyper-plane and it is the normal vector.

2) For any point x_0 in L,

$\beta^T x_0 + \beta_0 = 0$, which means $\beta^T x_0 = -\beta_0$.

3) We set $\beta^* = \frac{\beta}{||\beta||}$ as the unit normal vector of the hyper-plane L . For simplicity we call β^* norm vector. The distance of point x to L is given by

$$\beta^{*T}(x - x_0) = \beta^{*T}x - \beta^{*T}x_0 = \frac{\beta^T x}{||\beta||} + \frac{\beta_0}{||\beta||} = \frac{(\beta^T x + \beta_0)}{||\beta||}$$

Where x_0 is any point on L . Hence, $\beta^T x + \beta_0$ is proportional to the distance of the point x to the hyper-plane L .

4) The distance from a misclassified data point x_i to the hyper-plane L is

$$d_i = -y_i(\boldsymbol{\beta}^T x_i + \beta_0)$$

where y_i is a target value, such that $y_i = 1$ if $\boldsymbol{\beta}^T x_i + \beta_0 < 0$, $y_i = -1$ if $\boldsymbol{\beta}^T x_i + \beta_0 > 0$

Since we need to find the distance from the hyperplane to the *misclassified* data points, we need to add a negative sign in front. When the data point is misclassified, $\boldsymbol{\beta}^T x_i + \beta_0$ will produce an opposite sign of y_i . Since we need a positive sign for distance, we add a negative sign.

Learning Perceptron

The gradient descent is an optimization method that finds the minimum of an objective function by incrementally updating its parameters in the negative direction of the derivative of this function. That is, it finds the steepest slope in the D-dimensional space at a given point, and descends down in the direction of the negative slope. Note that unless the error function is convex, it is possible to get stuck in a local minima. In our case, the objective function to be minimized is classification error and the parameters of this function are the weights associated with the inputs, β

The gradient descent algorithm updates the weights as follows:

$$\beta^{\text{new}} \leftarrow \beta^{\text{old}} \rho \frac{\partial Err}{\partial \beta}$$

ρ is called the *learning rate*.¡br /¿ The Learning Rate ρ is positively related to the step size of convergence of $\min \phi(\boldsymbol{\beta}, \beta_0)$. i.e. the larger ρ is, the larger the step size is. Typically, $\rho \in [0.1, 0.3]$.

The classification error is defined as the distance of misclassified observations to the decision boundary:

To minimize the cost function $\phi(\boldsymbol{\beta}, \beta_0) = -\sum_{i \in M} y_i(\boldsymbol{\beta}^T x_i + \beta_0)$ where $M = \{\text{all points that are misclassified}\}$ ¡br¿

$$\frac{\partial \phi}{\partial \boldsymbol{\beta}} = -\sum_{i \in M} y_i x_i \text{ and } \frac{\partial \phi}{\partial \beta_0} = -\sum_{i \in M} y_i$$

Therefore, the gradient is

$$\nabla D(\beta, \beta_0) = \begin{pmatrix} -\sum_{i \in M} y_i x_i \\ -\sum_{i \in M} y_i \end{pmatrix}$$

Using the gradient descent algorithm to solve these two equations, we have

$$\begin{pmatrix} \boldsymbol{\beta}^{\text{new}} \\ \beta_0^{\text{new}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{\beta}^{\text{old}} \\ \beta_0^{\text{old}} \end{pmatrix} + \rho \begin{pmatrix} y_i x_i \\ y_i \end{pmatrix}$$

If the data is linearly-separable, the solution is theoretically guaranteed to converge to a separating hyperplane in a finite number of iterations.

In this situation the number of iterations depends on the learning rate and the margin. However, if the data is not linearly separable there is no guarantee that the algorithm converges.

Features

- A Perceptron can only discriminate between two classes at a time.
- When data is (linearly) separable, there are an infinite number of solutions depending on the starting point.
- Even though convergence to a solution is guaranteed if the solution exists, the finite number of steps until convergence can be very large.
- The smaller the gap between the two classes, the longer the time of convergence.

- When the data is not separable, the algorithm will not converge (it should be stopped after N steps).
- A learning rate that is too high will make the perceptron periodically oscillate around the solution unless additional steps are taken.
- Learning rate affects the accuracy of the solution and the number of iterations directly.

Separability and convergence

The training set D is said to be linearly separable if there exists a positive constant γ and a weight vector β such that $(\beta^T x_i + \beta_0) y_i > \gamma$ for all $1 < i < n$. That is, if we say that β is the weight vector of Perceptron and y_i is the true label of x_i , then the signed distance of the x_i from β is greater than a positive constant γ for any $(x_i, y_i) \in D$.

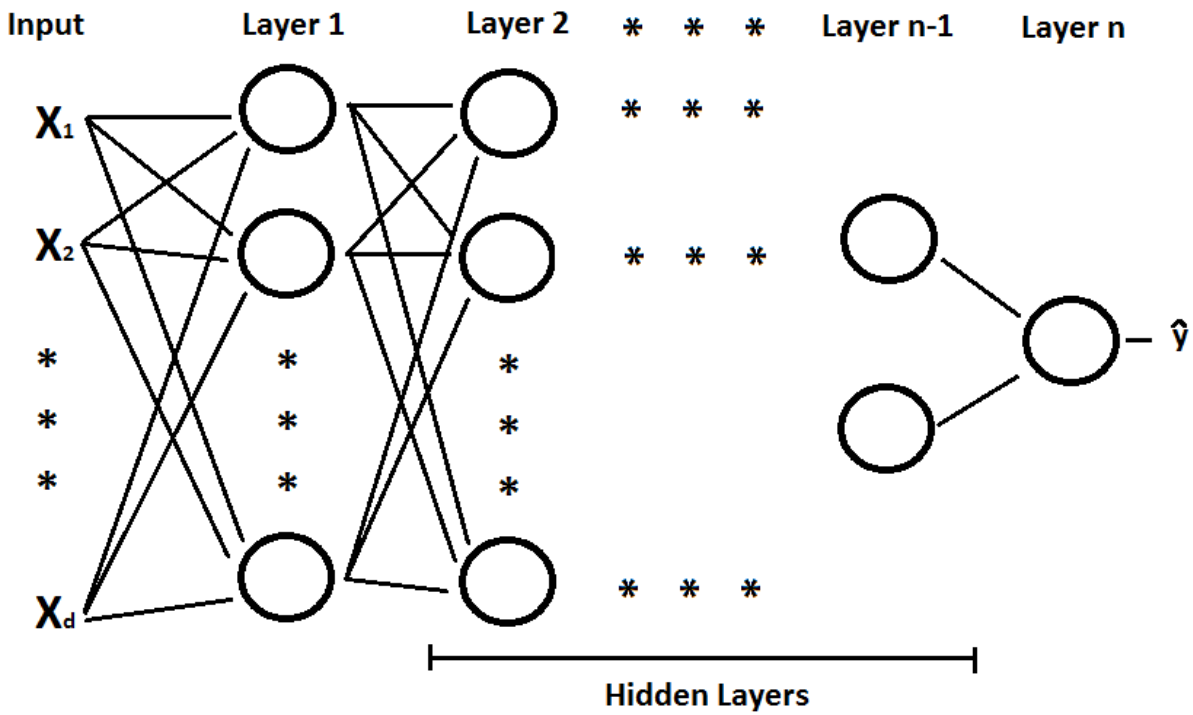
Separability and convergence

Novikoff (1962) proved that the perceptron algorithm converges after a finite number of iterations if the data set is linearly separable. The idea of the proof is that the weight vector is always adjusted by a bounded amount in a direction that it has a negative dot product with, and thus can be bounded above by $O(\sqrt{t})$ where t is the number of changes to the weight vector. But it can also be bounded below by $O(t)$ because if there exists an (unknown) satisfactory weight vector, then every change makes progress in this (unknown) direction by a positive amount that depends only on the input vector. This can be used to show that the number t of updates to the weight vector is bounded by $(\frac{2R}{\gamma})^2$, where R is the maximum norm of an input vector.

See http://en.wikipedia.org/wiki/Perceptron for details.

Neural Network

- A neural network is a multistate regression model which is typically represented by a network diagram.

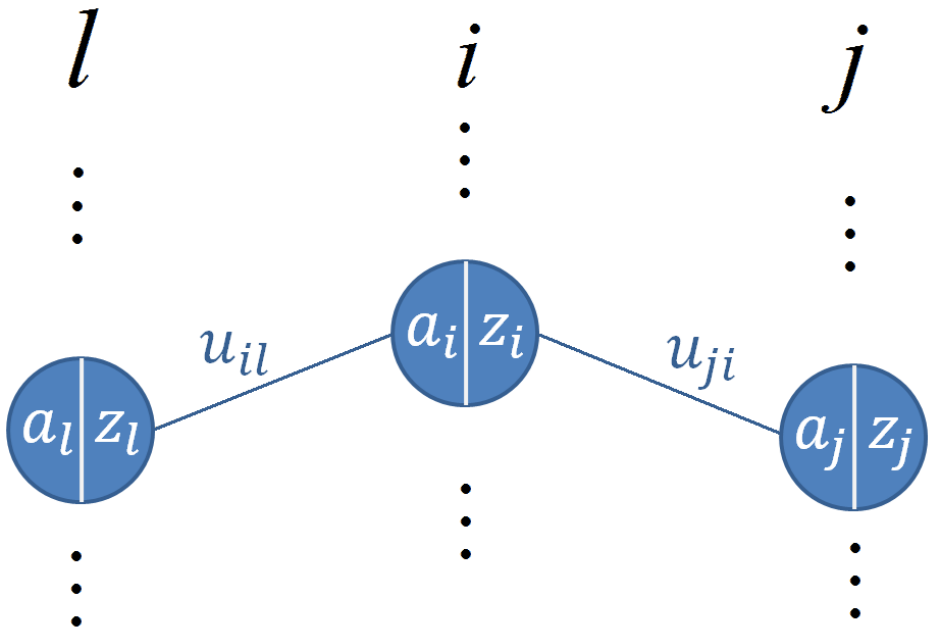


Feed Forward Neural Network

- For regression, typically k $= 1$ (the number of nodes in the last layer), there is only one output unit y_1 at the end.

- For c-class classification, there are typically c units at the end with the cth unit modelling the probability of class c, each y_c is coded as 0-1 variable for the cth class.

Backpropagation



Nodes from three hidden layers within the neural network. Each node is divided into the weighted sum of the inputs and the output of the activation function.

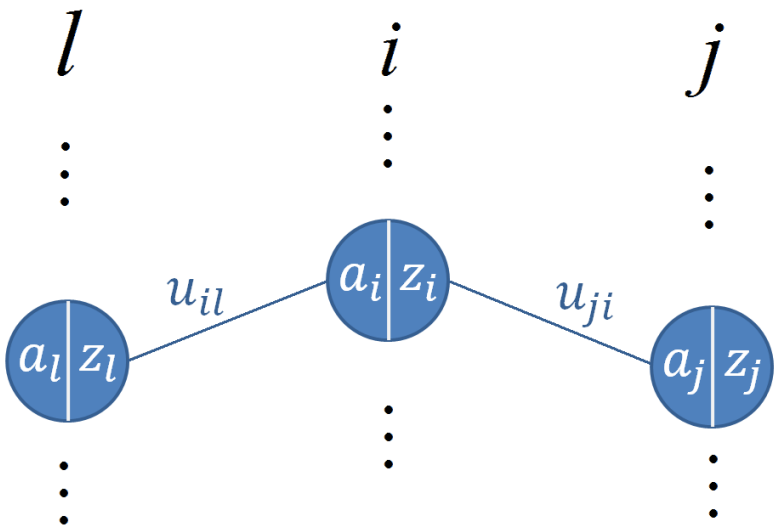
$$a_i = \sum_l z_l u_{il}$$
$$z_i = \sigma(a_i)$$
$$\sigma(a) = \frac{1}{1+e^{-a}}$$

Backpropagation

Take the derivative with respect to weight u_{il} :

$$\frac{\partial |y - \hat{y}|^2}{\partial u_{il}} = \frac{\partial |y - \hat{y}|^2}{\partial a_j} \cdot \frac{\partial a_j}{\partial u_{il}}$$

$$\frac{\partial |y - \hat{y}|^2}{\partial u_{il}} = \delta_j \cdot z_l$$



where $\delta_j = \frac{\partial |y - \hat{y}|^2}{\partial a_j}$

Backpropagation

$$\delta_i = \frac{\partial |y - \hat{y}|^2}{\partial a_i} =$$

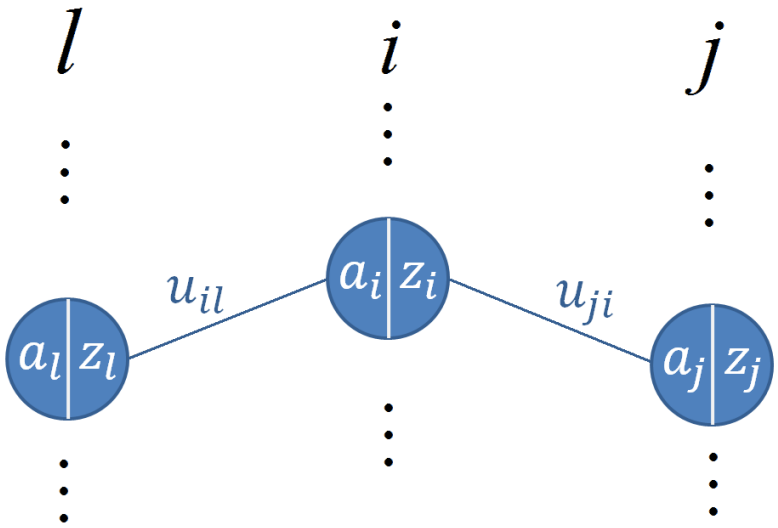
$$\sum_j \frac{\partial |y - \hat{y}|^2}{\partial a_j} \cdot \frac{\partial a_j}{\partial a_i}$$

$$\delta_i = \sum_j \delta_j \cdot \frac{\partial a_j}{\partial z_i} \cdot \frac{\partial z_i}{\partial a_i}$$

$$\delta_i = \sum_j \delta_j \cdot u_{ji} \cdot \sigma'(a_i)$$

where

$$\delta_j = \frac{\partial |y - \hat{y}|^2}{\partial a_j}$$



Backpropagation

Note that if $\sigma(x)$ is the sigmoid function, then $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

The recursive definition of δ_i

$\delta_i = \sigma'(a_i) \sum_j \delta_j \cdot u_{ji}$

Backpropagation

Now considering δ_k for the output layer:

$$\delta_k = \frac{\partial (y - \hat{y})^2}{\partial a_k} .$$

where $a_k = \hat{y}$

Assume an activation function is not applied in the output layer.

$$\delta_k = \frac{\partial (y - \hat{y})^2}{\partial \hat{y}}$$

$$\delta_k = -2(y - \hat{y})$$

Backpropagation

$$u_{il} \leftarrow u_{il} - \rho \frac{\partial (y - \hat{y})^2}{\partial u_{il}}$$

The network weights are updated using the backpropagation algorithm when each training data point x is fed into the feed forward neural network (FFNN).

Backpropagation

Backpropagation procedure is done using the following steps:

- First arbitrarily choose some random weights (preferably close to zero) for your network.
- Apply x to the FFNN's input layer, and calculate the outputs of all input neurons.
- Propagate the outputs of each hidden layer forward, one hidden layer at a time, and calculate the outputs of all hidden neurons.
- Once x reaches the output layer, calculate the output(s) of all output neuron(s) given the outputs of the previous hidden layer.
- At the output layer, compute $\delta_k = -2(y_k - \hat{y}_k)$ for each output neuron(s).

- Compute each δ_i , starting from $i = k - 1$ all the way to the first hidden layer, where $\delta_i = \sigma'(a_i) \sum_j \delta_j \cdot u_{ji}$.

- Compute $\dfrac{\partial (y - \hat{y})^2}{\partial u_{il}} = \delta_i z_l$ for all weights u_{il} .

- Then update $u_{il}^{\text{new}} \leftarrow u_{il}^{\text{old}} - \rho \cdot \dfrac{\partial (y - \hat{y})^2}{\partial u_{il}}$ for all weights u_{il} .

- Continue for next data points and iterate on the training set until weights converge.

Epochs

It is common to cycle through the all of the data points multiple times in order to reach convergence. An epoch represents one cycle in which you feed all of your datapoints through the neural network. It is good practice to randomized the order you feed the points to the neural network within each epoch; this can prevent your weights changing in cycles. The number of epochs required for convergence depends greatly on the learning rate & convergence requirements used.

Stochastic gradient descent

Suppose that we want to minimize an objective function that is written as a sum of differentiable functions.

$Q(w) = \sum_{i=1}^{n} Q_i(w)$

Each term Q_i is usually associated with the i_th data point.

Standard gradient descent (batch gradient descent):
$w = w - \eta \nabla Q(w) = w - \eta \sum_{i=1}^{n} \nabla Q_i(w)$

where η is the learning rate (step size).

Stochastic gradient descent

Stochastic gradient descent (SGD) considers only a subset of summand functions at every iteration.

This can be quite effective for large-scale problems.

Bottou, Leon; Bousquet, Olivier (2008). The Tradeoffs of Large Scale Learning. Advances in Neural Information Processing Systems 20. pp. 161–168.

The gradient of $Q(w)$ is approximated by a gradient at a single example: $w = w - \eta \nabla Q_i(w)$.

This update needs to be done for each training example.

Several passes might be necessary over the training set until the algorithm converges.

η might be adaptive.

Stochastic gradient descent

- Choose an initial value for w and η.
- Repeat until converged
  - Randomly shuffle data points in the training set.
  - For $i = 1, 2, ..., n$, do:
  - $w = w - \eta \nabla Q_i(w)$.

Example

Suppose $y = w_1 + w_2 x$

The objective function is:

$$Q(w) = \sum_{i=1}^{n} Q_i(w) = \sum_{i=1}^{n} (w_1 + w_2 x_i - y_i)^2.$$

Update rule will become:

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} := \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} - \eta \begin{bmatrix} 2(w_1 + w_2 x_i - y_i) \\ 2x_i(w_1 + w_2 x_i - y_i) \end{bmatrix}.$$

Example from Wikipedia

Mini-batches

Batch gradient decent uses all n data points in each iteration.

Stochastic gradient decent uses 1 data point in each iteration.

Mini-batch gradient decent uses b data points in each iteration.

b is a parameter called Mini-batch size.

Mini-batches

- Choose an initial value for w and η.
- Say $b = 10$
- Repeat until converged
  - Randomly shuffle data points in the training set.
  - For $i = 1, 11, 21, ..., n - 9$, do:
  - $w = w - \eta \sum_{k=i}^{i+9} \nabla Q_i(w)$.

Tuning η

If η is too high, the algorithm diverges.

If η is too low, makes the algorithm slow to converge.

A common practice is to make η_t a decreasing function of the iteration number t. e.g. $\eta_t = \frac{constant1}{t+constant2}$

The first iterations cause large changes in the w, while the later ones do only fine-tuning.

Momentum

SGD with momentum remembers the update Δw at each iteration.

Each update is as a (convex) combination of the gradient and the previous update.

$$\Delta w := \eta \nabla Q_i(w) + \alpha \Delta w$$

$$w := w - \eta \Delta w$$

Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (8 October 1986). "Learning representations by back-propagating errors". Nature 323 (6088): 533–536.