Deep learning

Feedforward deep networks & backpropagation

Hamid Beigy

Sharif University of Technology

October 16, 2022

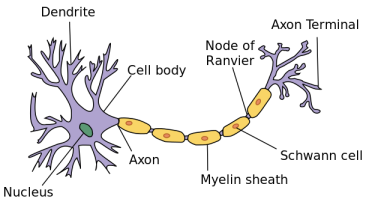
Table of contents

1. Introduction
2. History of neural networks
3. Activation function
4. Gradient based learning
5. Deep feed-forward networks
6. Backpropagation algorithm
7. Reading

Introduction

Neuron

1. The idea of neural networks began as a model of how neurons function in the brain.
2. Connected circuits was used to simulate its intelligent behavior.
3. The brain is made up of neurons.
   - a cell body
   - dendrites (inputs)
   - an axon (outputs)
   - synapses

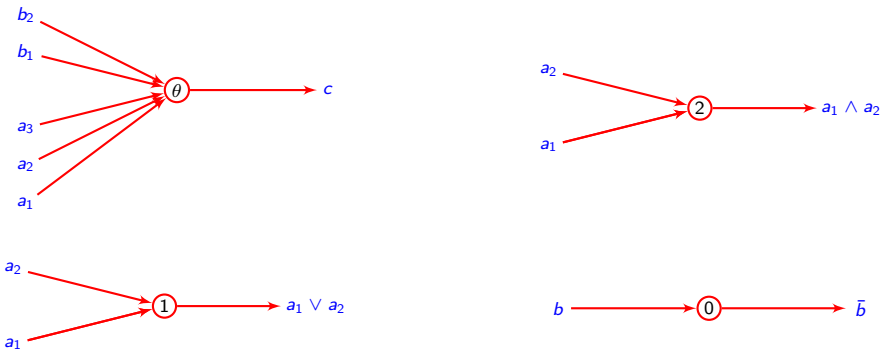


4. Synapses can be excitatory or inhibitory and may change over time.
5. When the sum of the inputs reach to some threshold, an electrical pulse will be sent on the axon.

History of neural networks

McCulloch and Pitts network (1943)

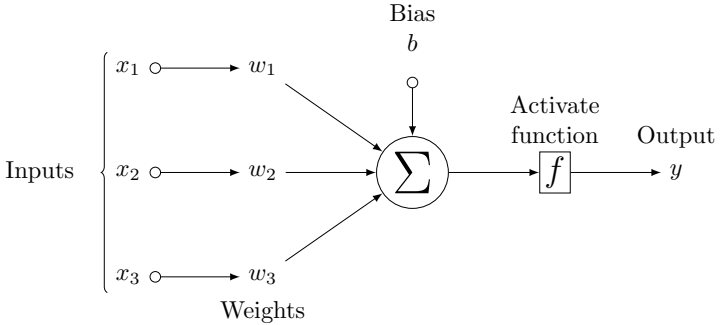
1. The first model of a neuron was invented by McCulloch and Pitts.
2. Inputs are binary.
3. This neuron has two types of inputs: Excitatory (a) and Inhibitory (b).
4. Excitatory connections have positive weights and inhibitory connections have negative weights.
5. The output is binary: fires (1) and not fires (0).
6. Until inputs summed up to a certain threshold level, output would remain zero.



Perceptron (Frank Rosenblat (1958))

1. Problems with McCulloch and Pitts -neurons
   - Weights and thresholds are analytically determined (cannot learn them).
   - Very difficult to minimize size of a network.
   - What about non-discrete and/or non-binary tasks?
2. Perceptron solution.
   - Weights and thresholds can be determined analytically or by a learning algorithm.
   - Continuous, bipolar and multiple-valued versions.
   - Rosenblatt randomly connected the perceptrons and changed the weights in order to achieve learning.
   - Efficient minimization heuristics exist.

1. The Perceptron has the following architecture.

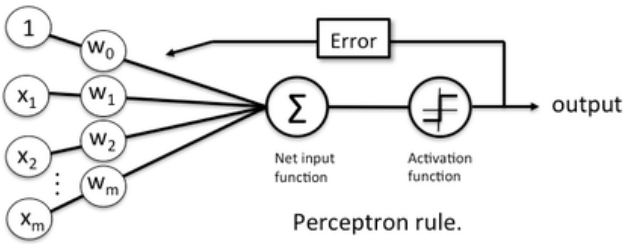


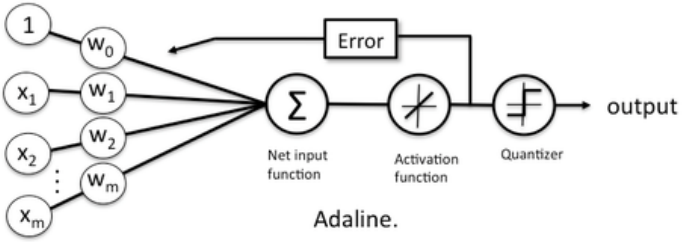
2. Let t be the correct output and y the output function of the network.
3. Perceptron updates weights (Rosenblatt 1960)

$$w_j^{(t)} \leftarrow w_j^{(t)} + \alpha x_j (t - y)$$

4. McCulloch and Pitts' neuron is a better model for the electrochemical process inside the neuron than the Perceptron.
5. But Perceptron is the basis and building block for the modern neural networks.

1. The model is same as perceptron, but uses different learning algorithm



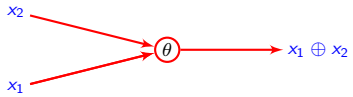


2. A multilayer network of Adaline units is known as a MAdaline.

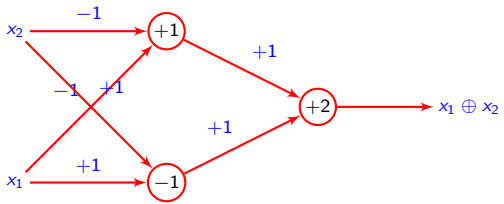
Adaline learning (Bernard Widrow and Ted Hoff (1960))

1. Let t be the correct output, and $y = \sum_{j=0}^{n} w_j x_j$.
2. Adaline updates weights
$$w_j^{(t+1)} \leftarrow w_j^{(t)} + \alpha x_j (t - y)$$
3. The Adaline converges to the least squares error which is $(t - y)^2$. This update rule is in fact the stochastic gradient descent update for linear regression.
4. In the 1960's, there were many articles promising robots that could think.
5. It seems there was a general belief that Perceptrons could solve any problem.

1. Minsky and Papert published their book Perceptrons. The book shows that Perceptrons could only solve linearly separable problems.
2. They showed that it is not possible for Perceptrons to learn an XOR function.



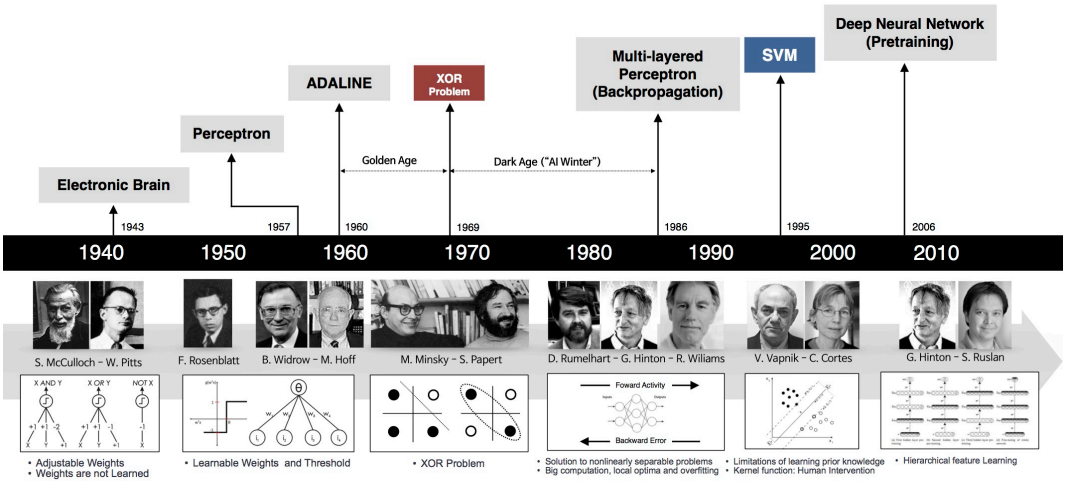
3. After Perceptrons was published, researchers lost interest in Perceptrons and neural networks.
4. The following multi-layer Perceptron can solve XOR problem.



5. The middle layer is a hidden layer.

1. Optimization
   - In 1969, Bryson and Ho described proposed Backpropagation as a multi-stage dynamic system optimization method.
   - In 1972, Stephen Grossberg proposed networks capable of learning XOR function.
   - In 1974, Paul Werbos, David E. Rumelhart, Geoffrey E. Hinton and Ronald J. Williams reinvented Backpropagation and applied in the context of neural networks. Backpropagation allowed Perceptron to be trained in a multilayer configuration.
2. In 1980s, the filed of artificial neural network research experienced a resurgence.
3. In 2000s, neural networks fell out of favor partly due to BP limitations.
4. In 2010, we are now able to train much larger networks using huge modern computing power such as GPUs.

History of neural networks



Activation function

Activation function

1. An activation function is added into a neural network to help it for learning complex patterns in the data.
2. An activation function converts the output of a neuron into another form used as input to the next neuron.
3. Why is there a need for activation functions?
   - Used to keep the output of a neuron to a certain interval as per our requirement.
   - Used to add non-linearity into a neural network.
4. Desirable features of an activation function

   Non-vanishing gradient We train neural networks using gradient-based algorithms. Hence, gradient must be nozero in all domain points. If the gradient tends to zero the problem is called vanishing gradient problem.

   Zero-centered Output of an activation function should be symmetrical at zero. This property prevents the gradients to shift to a particular direction.

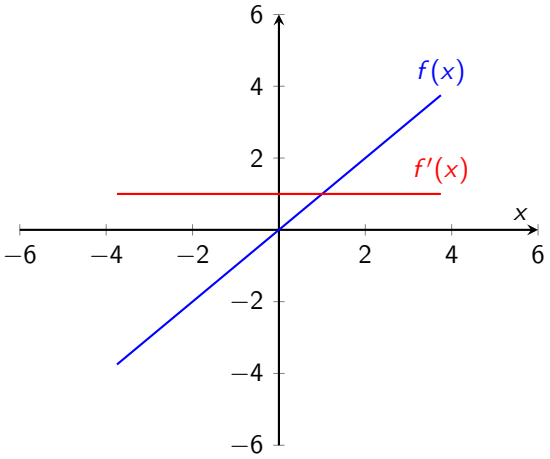
   Computational expense Activation functions are applied several times. They should be computationally inexpensive to calculate it and its derivative.

   Differentiable In gradient-based learning, we must calculate the gradient of activation functions. Hence, activation functions need to be differentiable or at least differentiable in parts.

Linear activation function

$$f(x) = x$$

$$f'(x) = 1$$



Advantages

1. Its calculation is simple.
2. Its derivative is nonzero.

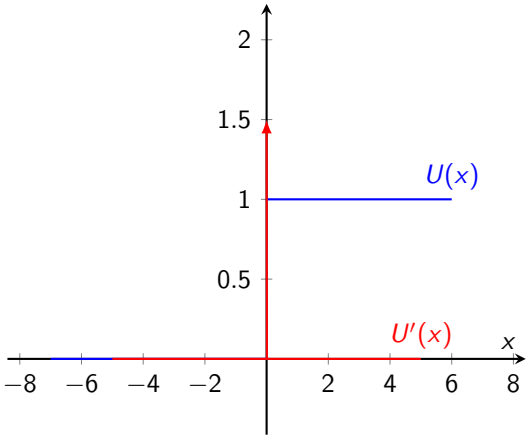
Disadvantages

1. Output is not in a range.
2. No results in nonlinearity.

Threshold activation function

$$U(x) = \begin{cases} 0 & \text{if } x > 0 \\ \\ 0 & \text{if } x < 0 \end{cases}$$

$$U'(x) = \begin{cases} 0 & \text{if } x > 0 \\ \delta(0) & \text{if } x = 0 \\ 0 & \text{if } x < 0 \end{cases}$$



Advantages

1. Its calculation is simple.

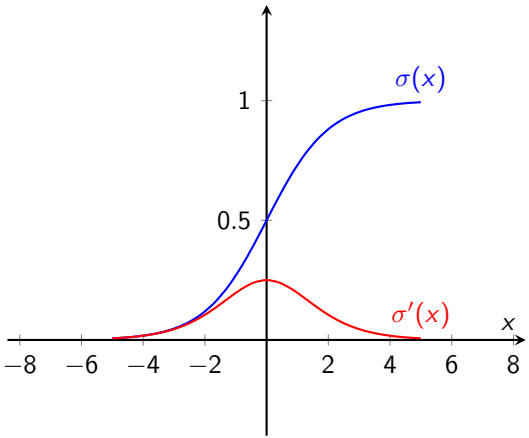
Disadvantages

1. Its derivative is zero except at the origin.

Sigmoid activation function

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

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$



Advantages

1. Output is in interval $[0, 1]$.
2. It is differentiable.

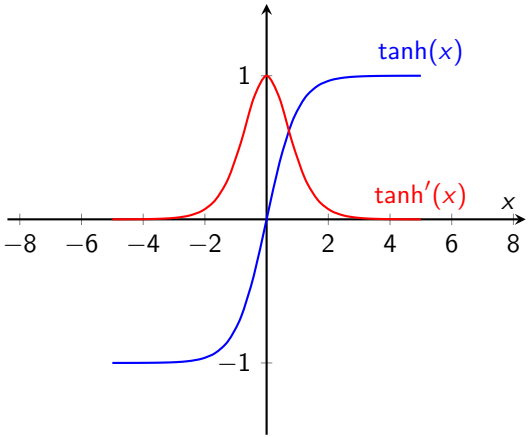
Disadvantages

1. Saturates and kills gradients.
2. Its output is not zero-centered.

Tanh activation function

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

$$\tanh'(x) = 1 - (tanh(x)^2)$$



Advantages

1. Output is in interval $[0, 1]$.
2. It is differentiable.
3. Its output is zero-centered.

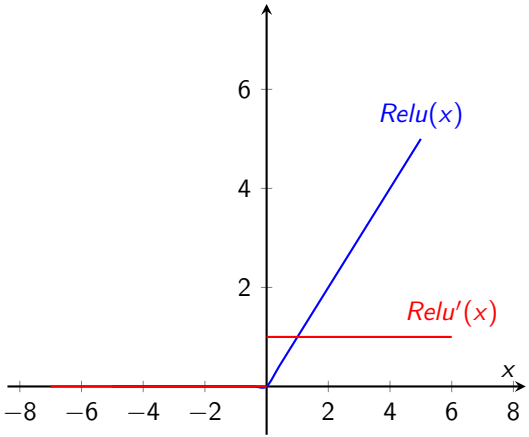
Disadvantages

1. Saturates and kills gradients.

Relu activation function

$$Relu(x) = \max(0, x)$$

$$Relu'(x) = \begin{cases} 1 & \text{if } x > 0 \\ ? & \text{if } x = 0 \\ 0 & \text{if } x < 0 \end{cases}$$



Advantages

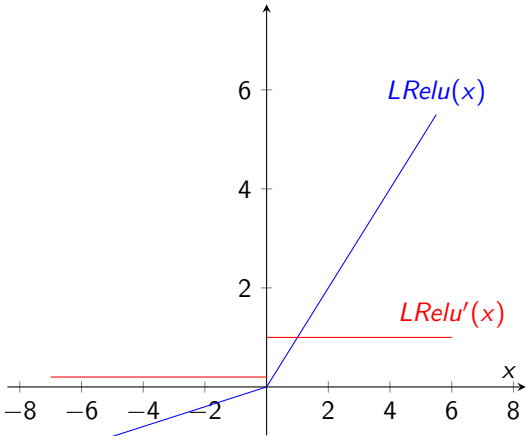
1. Its calculation is simple.
2. Its derivative is nonzero when $x > 0$.
3. It is computationally nonexpensive.

Disadvantages

1. Output is not in a range.
2. Gradient vanishes when $x < 0$.

$$LRelu(x) = \begin{cases} x & \text{if } x > 0 \\ -ax & \text{if } x < 0 \end{cases}$$

$$LRelu'(x) = \begin{cases} 1 & \text{if } x > 0 \\ ? & \text{if } x = 0 \\ -a & \text{if } x < 0 \end{cases}$$



Advantages

1. Its calculation is simple.
2. Its derivative is nonzero.
3. It is computationally nonexpensive.

Disadvantages

1. Output is not in a range.

1. Softmax is a more generalized form of the sigmoid.
2. Softmax is used in the output layer of neural networks for multi-class classification problems.

$$a_i(x) = \frac{\exp^{z_i}}{\sum_k \exp^{z_k}}$$

3. Let $\mathbf{x} = [1.60, 0.55, 0.98]^\top$.
4. Applying softmax results in $a_i = [0.51, 0.18, 0.31]^\top$.

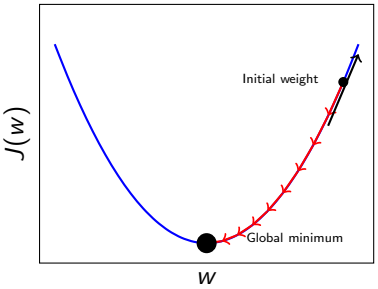
Gradient based learning

Cost function

1. The goal of machine learning algorithms is to construct a model (hypothesis) that can be used to estimate t based on x.
2. Let the model be in form of
$$h(x) = w_0 + w_1 x$$
3. The goal of creating a model is to choose parameters so that $h(x)$ is close to t for the training data, x and t.
4. We need a function that will minimize the parameters over our dataset. A function that is often used is mean squared error,
$$J(w) = \frac{1}{2m} \sum_{i=1}^{m} (h(x_i) - t_i)^2$$
5. How do we find the minimum value of cost function?

Gradient descent

1. Gradient descent is by far the most popular optimization strategy, used in machine learning and deep learning at the moment.
2. Cost (error) is a function of the weights (parameters).
3. We want to reduce/minimize the error.
4. Gradient descent: move towards the error minimum.
5. Compute gradient, which implies get direction to the error minimum.
6. Adjust weights towards direction of lower error.



1. We have the following hypothesis and we need fit to the training data

$$h(x) = w_0 + w_1 x$$

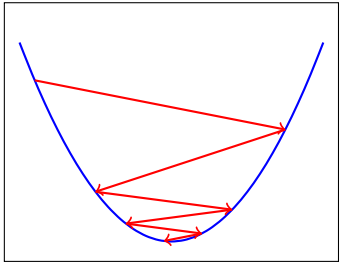
2. We use a cost function such Mean Squared Error

$$J(w) = \frac{1}{2m} \sum_{i=1}^{m} (h(x_i) - t_i)^2$$

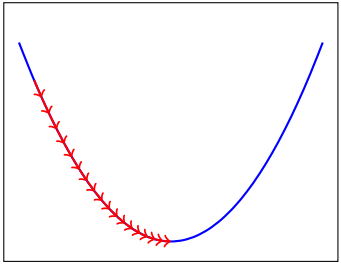
3. This cost function can be minimized using gradient descent with step (learning) rate α.

$$w_0^{(t+1)} = w_0^{(t)} - \alpha \frac{\partial J(w^{(t)})}{\partial w_0}$$
$$w_1^{(t+1)} = w_1^{(t)} - \alpha \frac{\partial J(w^{(t)})}{\partial w_1},$$

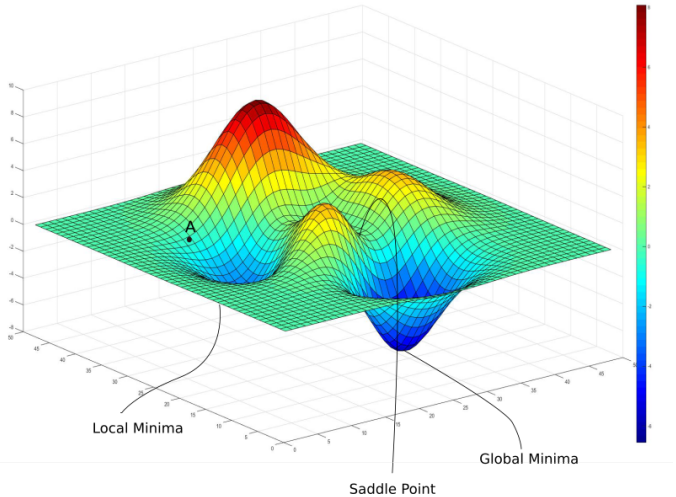
Big step size



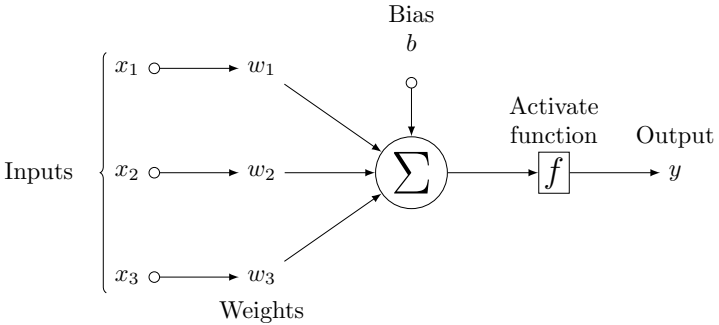
Small step size



1. Local minimim: A local minimum is a minimum within some neighborhood that need not be (but may be) a global minimum.
2. Saddle points: For non-convex functions, having the gradient to be 0 is not good enough. Example: $f(x) = x_1^2 - x_2^2$ at $x = (0, 0)$ has zero gradient but it is clearly not a local minimum as $x = (0, \epsilon)$ has smaller function value. The point $(0, 0)$ is called a saddle point of this function.



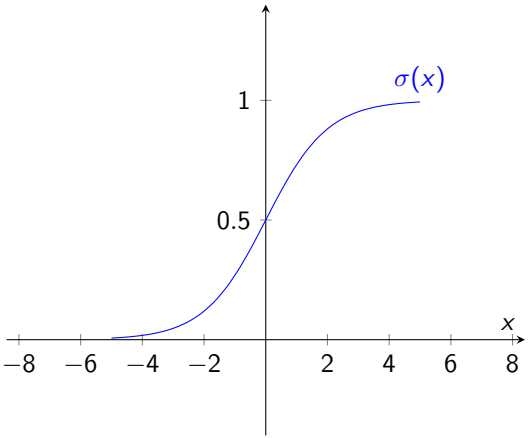
1. Considering the following single neuron



1. We want to train this neuron to minimize the following cost function

$$J(w) = \frac{1}{2m}\sum_{i=1}^{m}(h(x^i) - t^i)^2$$

2. Considering the sigmoid activation function $f(z) = \frac{1}{1+e^{-z}}$



3. We want to calculate $\frac{\partial J(w)}{\partial w_i}$

1. We want to calculate $\frac{\partial J(w)}{\partial w_i}$
2. By using the chain rule, we obtain

$$\frac{\partial J(w)}{\partial w_j} = \frac{\partial J(w)}{\partial f(z)} \times \frac{\partial f(z)}{\partial z} \times \frac{\partial z}{\partial w_j}$$

$$\frac{\partial J(w)}{\partial f(z^i)} = \frac{1}{m} \sum_{i=1}^{m} (f(z^i) - t^i)$$

$$\frac{\partial f(z)}{\partial z} = \frac{e^{-z}}{(1+e^{-z})^2} = f(z)(1-f(z))$$

$$\frac{\partial z}{\partial w_j} = x^j$$

$$w_j^{(t+1)} = w_j^{(t)} - \alpha \frac{\partial J(w)}{\partial w_j}$$

α is the learning rate.

1. We want to train this neuron to minimize the following cost function

$$J(w) = \sum_{i=1}^{m} \left[-t^i \ln h(x^i) - (1 - t^i) \ln(1 - h(x^i)) \right]$$

2. Computing the gradients of $J(w)$ with respect to w, we obtain

$$\nabla J(w) = \sum_{i=1}^{m} t^i x^i (h(x^i) - t^i)$$

3. Updating the weight vector using the gradient descent rule will result in

$$w^{(t+1)} = w^{(t)} - \alpha \sum_{i=1}^{m} t^i x^i (h(x^i) - t^i)$$

α is the learning rate.

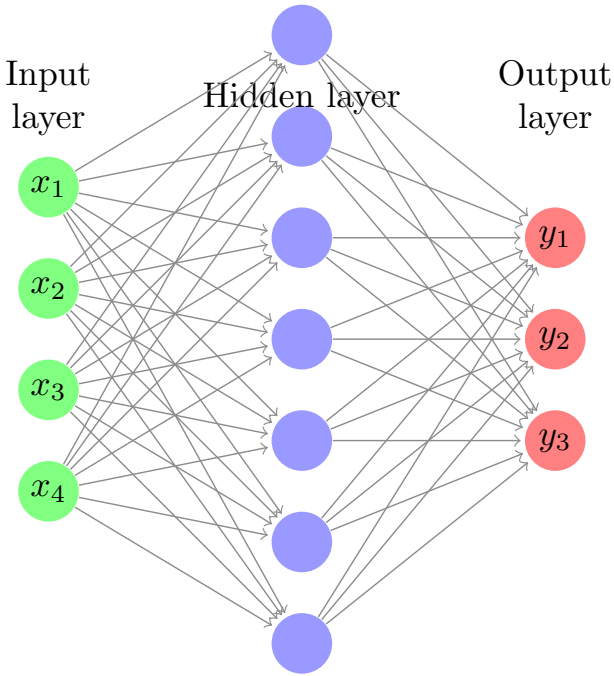
Tuning learning rate (α)

1. If α is too high, the algorithm diverges.
2. If α is too low, makes the algorithm slow to converge.
3. A common practice is to make α_k a decreasing function of the iteration number k. e.g.

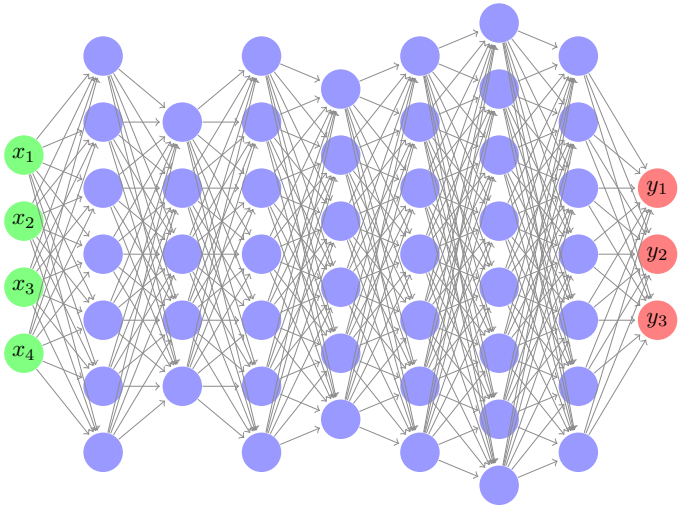
$$\alpha_k = \frac{c_1}{k + c_2}$$

   where c_1 and c_2 are two constants.
4. The first iterations cause large changes in the w, while the later ones do only fine-tuning.

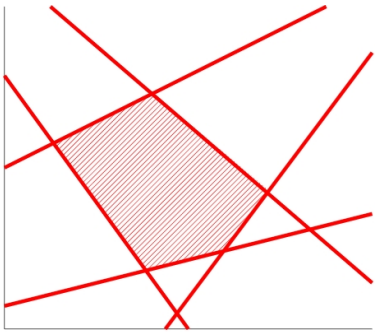
Deep feed-forward networks

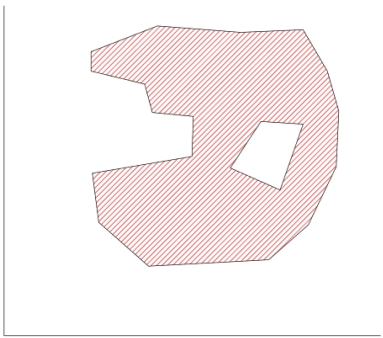

Input layer
Hidden layer
Output layer
x_1
x_2
x_3
x_4
y_1
y_2
y_3

Deep feed-forward networks



1. What is the topology of network for the given problem?
2. Can we build a network to create every decision boundary?
3. Neural networks are universal approximators.

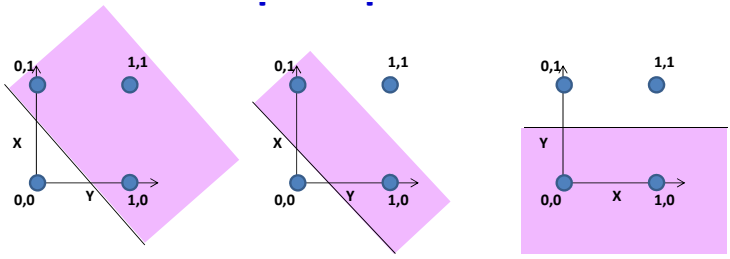


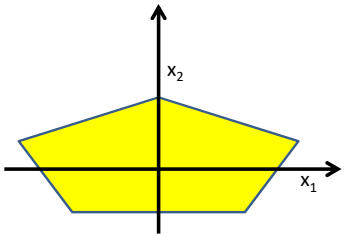


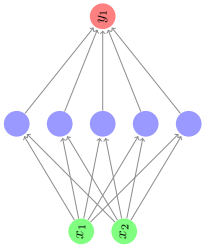
4. Can we build a network without local minimia in cost function?

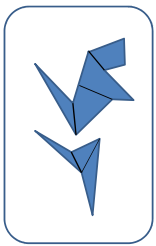
Decision surface of perceptron

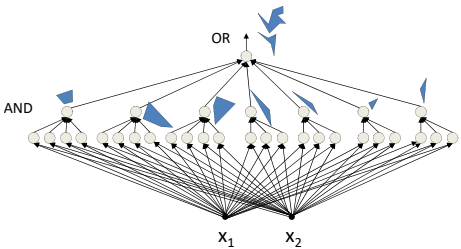
1. What is the decision surface of perceptron?











Can you build such region with one hidden layer network?

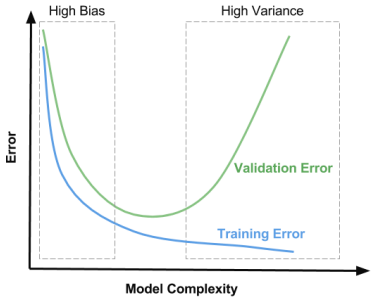
1. Specifying the topology of network.
   - #-layers
   - #-nodes in each layer
   - function of each node
   - activation of each node
2. What is the topology of network for the given problem?
3. Can we build a network to create every decision boundary?
4. Neural networks are universal approximators.
5. Can we build a network without local minimia in cost function?
6. Specifying the cost function.
7. We use gradient decent algorithm for training the network.
8. But, we don't have the true output of each hidden unit.

Choosing activation functions

1. Output layer
   - Linear activation function
   - ReLU activation function
   - Sigmoid activation function
   - Softmax activation function
2. Hidden layers
   - Linear activation function
   - ReLU activation function
   - Sigmoid activation function

Choosing the network topology

1. A simple approach for choosing the network topology is by trial and error.
2. We partition the available data into three parts : training data, validation data, and test data.
3. We choose a topology and train the network using the training data.
4. After training, we evalute the trained network using validation data.



Backpropagation algorithm

Training a neural network

Data: A training set $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$
Result: Weight matrices of the neural network
Initialize randomly weights in the network;
while *not at trained* **do**
  Create a batch $S_B \subseteq S$;
  Let $K \leftarrow |S_B|$;
  for $i \leftarrow 1$ **to** K **do**
    Give $\mathbf{x}_i$ to the network and $\hat{\mathbf{y}}_i$;
  end
  Compute $J(w)$;
  Compute $\nabla_w J(w)$;
  Compute $w^{t+1} \leftarrow w^t - \alpha \nabla_w J(w)$;
end

Definition (Epoch)

Epoch is defined as one forward pass and one backward pass of all training examples.

Definition (Batch size)

Batch size is defined as the number of training examples in one forward/backward pass.

Definition (Number of iterations)

The number of iterations is defined as the number of passes, each pass using the number of examples in the batch.

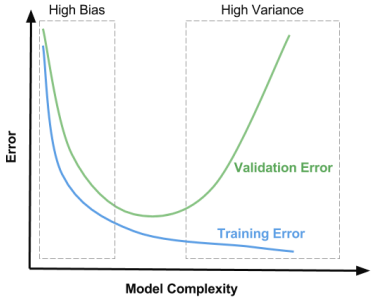
1. Gradient descent can be used with different batch sizes.
   - $K = 1$ Stochastic gradient descent
   - $K \ll m$ Mini-batch gradient descent
   - $K = m$ Batch gradient descent

Example (Batch size)

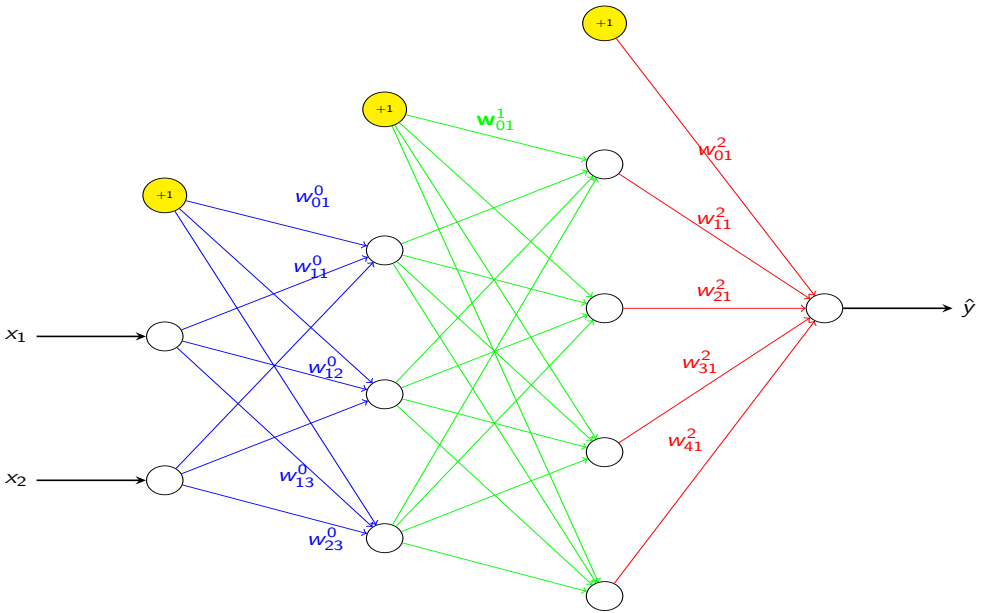
- Let the size of the training set be $m = 1000$.
- In stochastic gradient descent, we use 1000 batches of size 1.
- Let $K = 50$, in mini-batch gradient descent, we use 20 batches of size 50.
- In batch gradient descent, we use one batch of size 1000.

Stopping the training of the network

1. A simple approach for stopping criteria is to partition the available data into three parts : training data, validation data, and test data.
2. We traing the network using the training data and in some epochs, we evalute the trained network using validation data.



1. For every batch, give training examples one by one to the network.
2. Compute the output of each layer and pass it as the input of the next layer, until outputs are produced.



1. After computing the output of the network, loss is computed.

$$J(w) = \frac{1}{K} \sum_{k=1}^{K} \sum_{c=1}^{C} \ell(\hat{y}_k^c, y_k^c)$$

where

- $\hat{y}_k^c$ shows the cth output of the network for kth training example.
- y_k^c shows the cth desired output for kth training example.

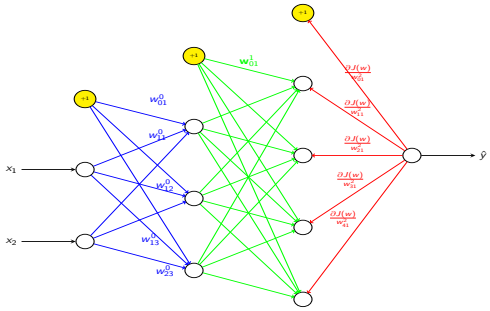
Example

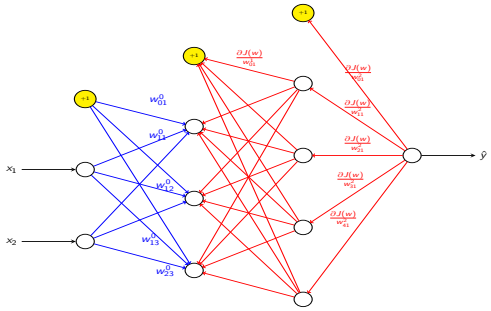
Assume that we have a multi-class classification problem for $C = 3$ classes. Then, each training example is in the form of (x, y), where $y \in \{1, 2, 3\}$. To map this to the network, we use a binary vector

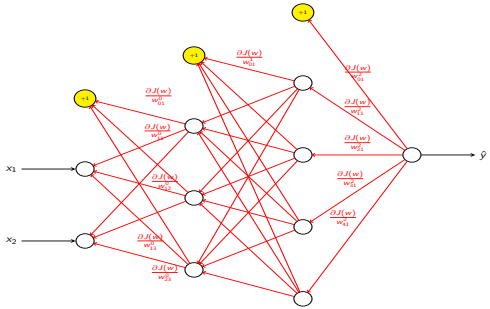
- $(1, 0, 0)$ to represent class 1.
- $(0, 1, 0)$ to represent class 2.
- $(0, 0, 1)$ to represent class 3.

Then use the above vectors for computing loss function.

1. After computing the loss function, we must compute $\nabla_w J(w)$.



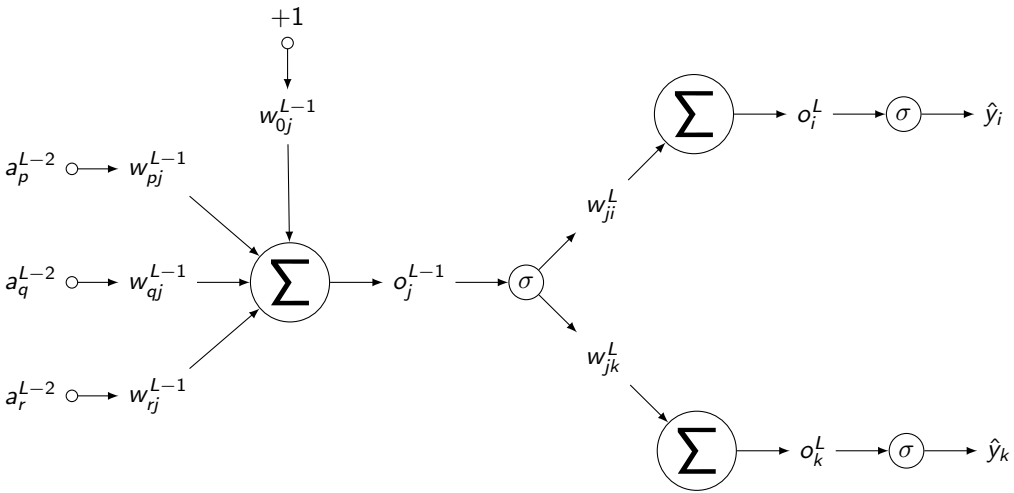




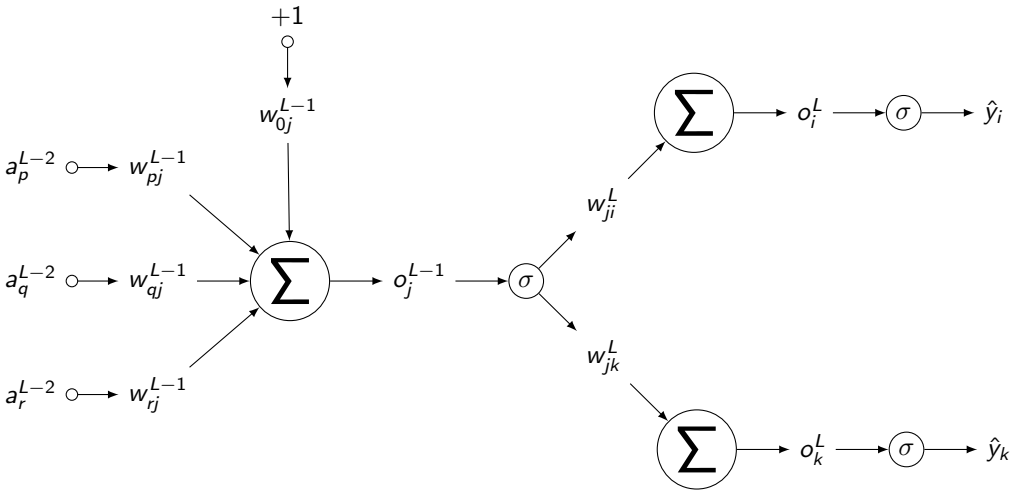
Then for every weight w, we use the following rule to update that weight.

$$w^{t+1} = w^t - \alpha \frac{\partial J(w)}{\partial w}$$

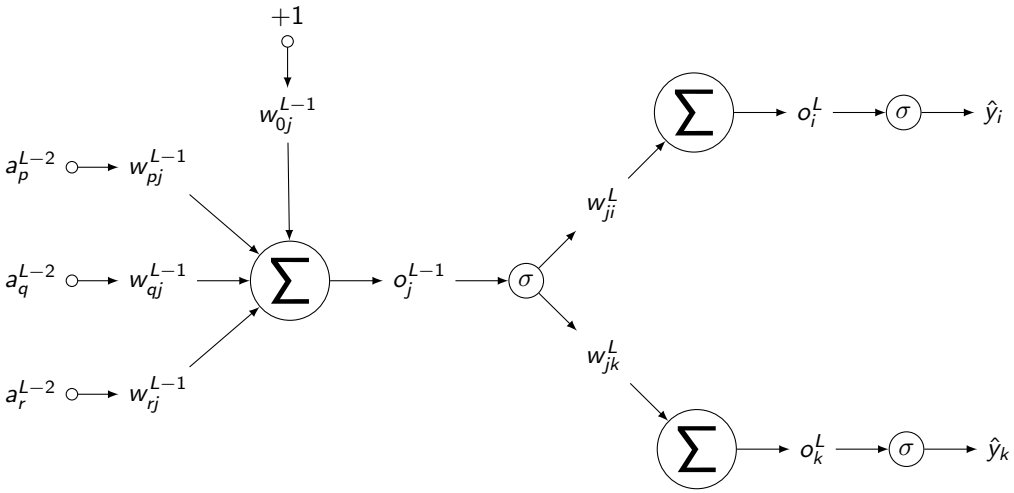
1. We use the following notation for calculating $\nabla_w J(w)$.



1. Assume that we have a L-layer network, where the last layer has C output nodes.
2. Let all nodes use sigmoid activation functions.
3. We denote the output of jth node in the lth layer by a_j^l.



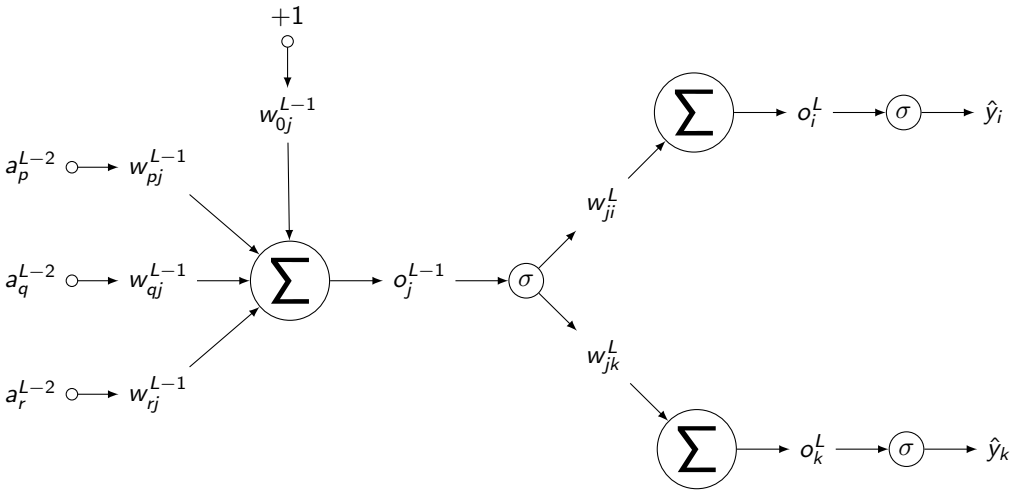
1. Consider the following notation.



2. We mus compute $\frac{\partial J(w)}{\partial w}$.

$$\frac{\partial J(w)}{\partial w_{pj}^{L-1}} = \frac{\partial \sum_{s=1}^{K}\sum_{c=1}^{C}\ell(\hat{y}_{cs}, y_{cs})}{\partial w_{pj}^{L-1}} = \frac{\partial \sum_{c=1}^{C}\sum_{s=1}^{K}\ell(\hat{y}_{cs}, y_{cs})}{\partial w_{pj}^{L-1}} = \sum_{c=1}^{C}\frac{\partial \sum_{s=1}^{K}\ell(\hat{y}_{cs}, y_{cs})}{\partial w_{pj}^{L-1}}$$

1. Consider the following notation.



2. How to drive $\sum_{c=1}^{C} \frac{\partial \sum_{s=1}^{K} \ell(\hat{y}_{cs}, y_{cs})}{\partial w_{pj}^{L-1}}$?

$$\sum_{c=1}^{C} \frac{\partial \sum_{s=1}^{K} \ell(\hat{y}_{cs}, y_{cs})}{\partial w_{pj}^{L-1}} = \sum_{c=1}^{C} \sum_{s=1}^{K} \frac{\partial \ell(\hat{y}_{cs}, y_{cs})}{\partial \hat{y}_{cs}} \frac{\partial \hat{y}_{cs}}{\partial o_c^L} \frac{\partial o_c^L}{\partial a_j^{L-1}} \frac{\partial a_j^{L-1}}{\partial o_j^{L-1}} \frac{\partial o_j^{L-1}}{\partial w_{pj}^{L-1}}$$

1. We have

$$\sum_{c=1}^{C} \frac{\partial \sum_{s=1}^{K} \ell(\hat{y}_{cs}, y_{cs})}{\partial w_{pj}^{L-1}} = \sum_{c=1}^{C} \sum_{s=1}^{K} \frac{\partial \ell(\hat{y}_{cs}, y_{cs})}{\partial \hat{y}_{cs}} \frac{\partial \hat{y}_{cs}}{\partial o_c^L} \frac{\partial o_c^L}{\partial a_j^{L-1}} \frac{\partial a_j^{L-1}}{\partial o_j^{L-1}} \frac{\partial o_j^{L-1}}{\partial w_{pj}^{L-1}}$$

2. Now we obtain

$$\begin{aligned}
\frac{\partial \ell(\hat{y}_{cs}, y_{cs})}{\partial \hat{y}_{cs}} &= ? \\
\frac{\partial \hat{y}_{cs}}{\partial o_c^L} &= \sigma\left(o_c^L\right)\left(1 - \sigma\left(o_c^L\right)\right) \\
\frac{\partial o_c^L}{\partial a_j^{L-1}} &= \frac{\partial \sum_l w_{lc}^L a_l^{L-1}}{\partial a_j^{L-1}} = w_{jk}^L \\
\frac{\partial a_j^{L-1}}{\partial o_j^{L-1}} &= \sigma\left(o_c^{L-1}\right)\left(1 - \sigma\left(o_c^{L-1}\right)\right) \\
\frac{\partial o_j^{L-1}}{\partial w_{pj}^{L-1}} &= \frac{\partial \sum_l w_{lj}^{L-1} a_l^{L-2}}{\partial w_{pj}^{L-1}} = a_p^{L-2}
\end{aligned}$$

3. How do you generalize the above equations to other layers?

Summary

1. Multi-layer Perceptron (MLP) uses error-backpropagation algorithm to propagate error to all layers.
2. Multi-layer Perceptron uses gradient descent (stochastic/mini-batch) to update weights.
3. Error function contains many local minima and there is no guarantee of convergence.
4. How well does MLP learn and how can we improve it (next lecture)?
5. How well will MLP generalize (outside training data)?

Reading

Readings

1. Chapter 6 of Deep Learning Book[1]

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. The MIT Press.

References

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. The MIT Press.

Questions?