

Feedforward Neural Networks and Backpropagation

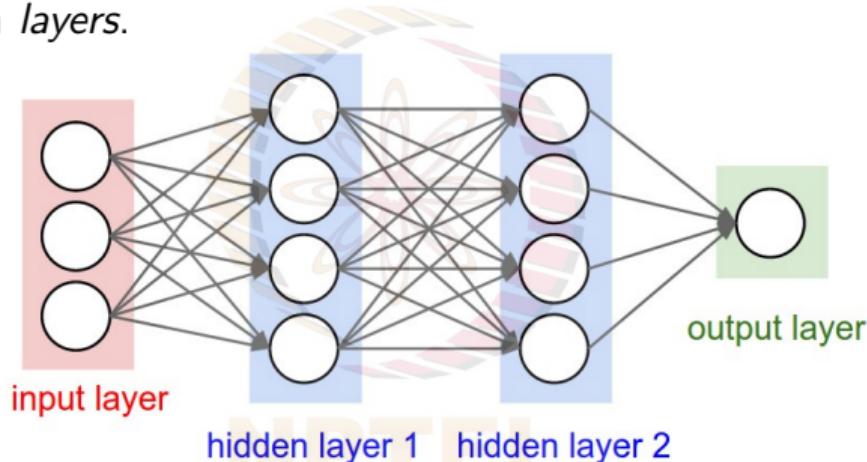
Vineeth N Balasubramanian

Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad



Feedforward Networks

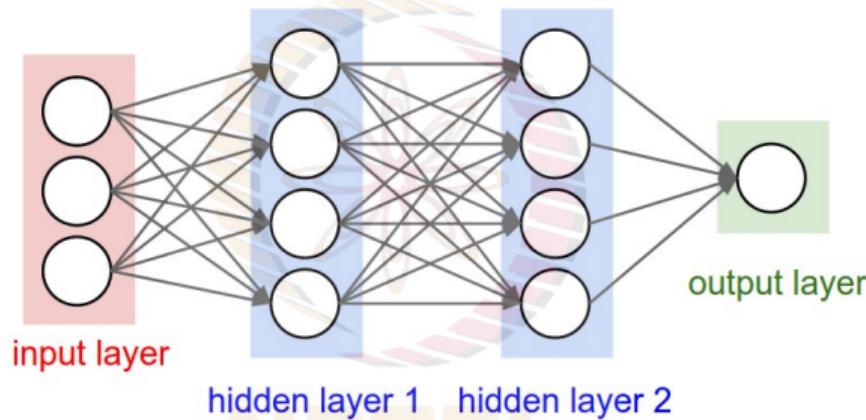
- A feedforward neural network, also called a multi-layer perceptron, is a collection of neurons, organized in *layers*.



- It is used to approximate some function f^* . For instance, f^* could be a classifier that maps an input vector x to a category y .
- The neurons are arranged in the form of a directed acyclic graph i.e., the information only flows in one direction - input x to output y . Hence the term **feedforward**.

Feedforward Networks

- The number of layers in the network (excluding the input layer) is known as **depth**



- Each neuron can be seen as a **vector-to-scalar** function which takes a vector of inputs from the previous layer and computes a scalar value.
- Above network can be seen as a composition of functions $y = f^{(3)}(f^{(2)}(f^{(1)}(x)))$, $f^{(1)}$ being the first hidden layer, $f^{(2)}$ being the second and $f^{(3)}$ being the final output layer.

Feedforward Networks

- To approximate some function f^* , we are generally given noisy estimates of $f^*(\mathbf{x})$ at different points, in the form of a dataset $\{\mathbf{x}_i, y_i\}_{i=1}^M$.
- Our neural network defines a function $y = f(\mathbf{x}; \boldsymbol{\theta})$. Our goal is to learn the parameters (weights and biases) $\boldsymbol{\theta}$ such that f best approximates f^* .
- How to find the values of the parameters i.e., train the network?
- In this lecture, we introduce **Gradient Descent**, the go-to method to train neural networks

Gradient Descent: A 1D Example

- Neural networks are usually trained by minimizing a loss function, such as mean square error:

$$Loss_{MSE} = \frac{1}{M} \sum_{i=1}^M (f^*(\mathbf{x}) - f(\mathbf{x}; \boldsymbol{\theta}))^2$$

- Let us consider a simple 1D example, where we try to minimize the function $f(x) = x^2$. Specifically, we find out the value x^* gives the smallest value for $f(x)$ i.e., $f(x^*)$.

$$x^* = \arg \min_x f(x)$$

Gradient Descent: A 1D Example

- We can obtain the slope of the function $f(x)$ at x by taking its derivative i.e., $f'(x)$.



Gradient Descent: A 1D Example

- We can obtain the slope of the function $f(x)$ at x by taking its derivative i.e., $f'(x)$.
- This means, if we give a very small *push* to x in the direction (sign) of the slope, we're sure that the function will increase.

$$f(x + p \cdot \text{sign}(f'(x))) > f(x) \text{ for an infinitesimally small } p$$

NPTEL

Gradient Descent: A 1D Example

- We can obtain the slope of the function $f(x)$ at x by taking its derivative i.e., $f'(x)$.
- This means, if we give a very small *push* to x in the direction (sign) of the slope, we're sure that the function will increase.

$$f(x + p \cdot \text{sign}(f'(x))) > f(x) \text{ for an infinitesimally small } p$$

- The reverse is also true i.e.,

$$f(x - p \cdot \text{sign}(f'(x))) < f(x) \text{ for an infinitesimally small } p$$

- This forms the basis for gradient descent - we start off at a random x , and take small steps in the direction of the **negative** gradient.

Gradient Descent: A 1D Example

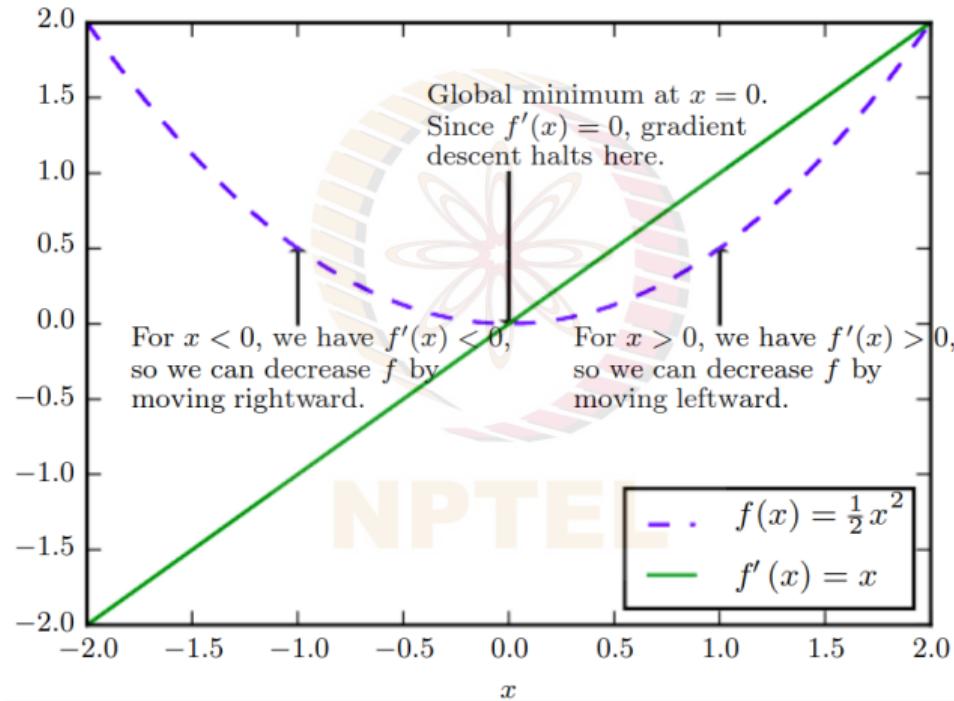
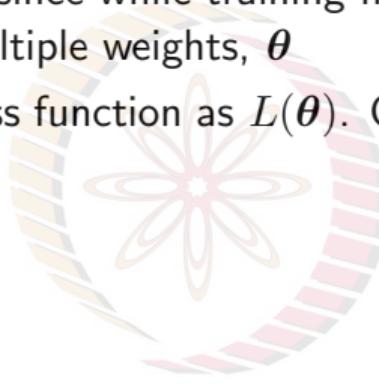


Figure Credit: Goodfellow et al, DL Book Ch 4

Why Negative Gradient?

- Consider the multivariate case, since while training neural networks, the loss function we minimize is parametrized by multiple weights, θ
- For simplicity, we denote our loss function as $L(\theta)$. Our aim is to find the weight vector θ which minimizes $L(\theta)$



Why Negative Gradient?

- Consider the multivariate case, since while training neural networks, the loss function we minimize is parametrized by multiple weights, θ
- For simplicity, we denote our loss function as $L(\theta)$. Our aim is to find the weight vector θ which minimizes $L(\theta)$
- Let \mathbf{u} , a unit vector, be the direction that takes us to the minimum, i.e.:

$$\min_{\mathbf{u}, \mathbf{u}^T \mathbf{u}=1} \mathbf{u}^T \nabla_{\theta} L(\theta)$$

NPTEL

Why Negative Gradient?

- Consider the multivariate case, since while training neural networks, the loss function we minimize is parametrized by multiple weights, θ
- For simplicity, we denote our loss function as $L(\theta)$. Our aim is to find the weight vector θ which minimizes $L(\theta)$
- Let \mathbf{u} , a unit vector, be the direction that takes us to the minimum, i.e.:

$$\min_{\mathbf{u}, \mathbf{u}^T \mathbf{u}=1} \mathbf{u}^T \nabla_{\theta} L(\theta)$$

$$= \min_{\mathbf{u}, \mathbf{u}^T \mathbf{u}=1} \|\mathbf{u}\|_2 \|\nabla_{\theta} L(\theta)\|_2 \cos \beta$$

- Since $\|\mathbf{u}\|_2 = 1$, we can minimize the above function when $\beta = 180^\circ$, i.e. when \mathbf{u} is the direction of **negative** gradient

How to use Gradient Descent

We can use Gradient Descent to train neural networks as follows:

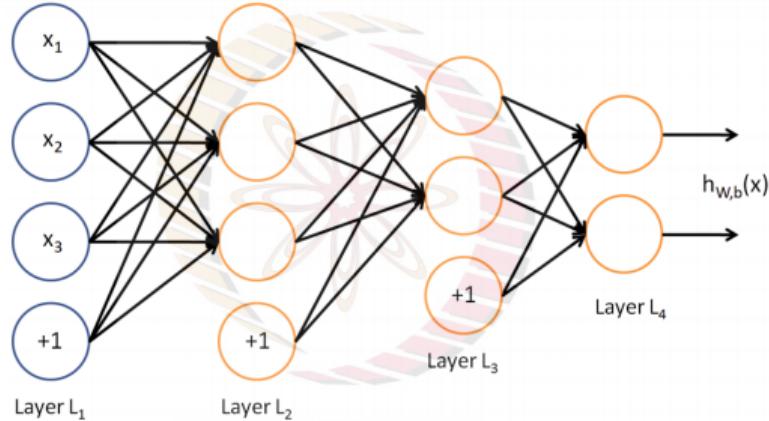
- Start with a random weight vector θ .
- Compute the loss function over the dataset, i.e., $L(\theta)$ with the current network, using a suitable loss function such as mean-squared error
- Compute the gradients of the loss function with respect to each weight value $\frac{\delta L}{\delta \theta_i}$.
- Update the weights as follows, where η is the learning rate i.e., the amount by which the weight is changed in each step:

$$\theta_i^{next} = \theta_i^{curr} - \eta \frac{\delta L}{\delta \theta_i^{curr}}$$

We can repeat the above steps until the gradient is zero.

Gradient Descent

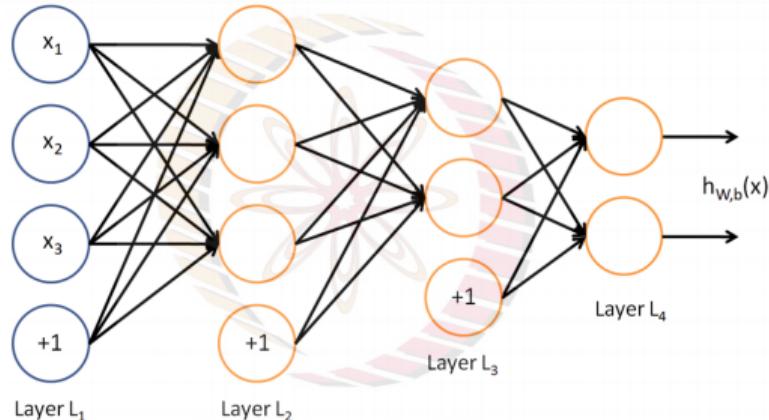
- A feedforward neural network is a composition of multiple functions, organized as layers



- What do we need to implement gradient descent? Compute gradient of loss function w.r.t. each weight in the network. How to do this?

Gradient Descent

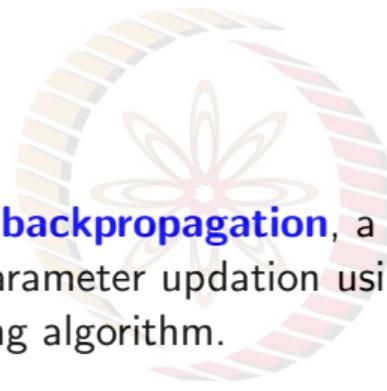
- A feedforward neural network is a composition of multiple functions, organized as layers



- What do we need to implement gradient descent? Compute gradient of loss function w.r.t. each weight in the network. How to do this?
- Using the **chain rule in calculus**
 - Computing gradient w.r.t. a weight in layer i requires computation of gradients with respect to outputs which involve that weight i.e., all activations from layer $i + 1$ to last layer, n_l

Gradient Descent

In the next few slides, we introduce **backpropagation**, a procedure which combines gradient computation using chain rule and parameter updation using Gradient Descent, thus fully describing the neural network training algorithm.



Backpropagation

Consider a simple feed forward neural network (or multilayer perceptron)

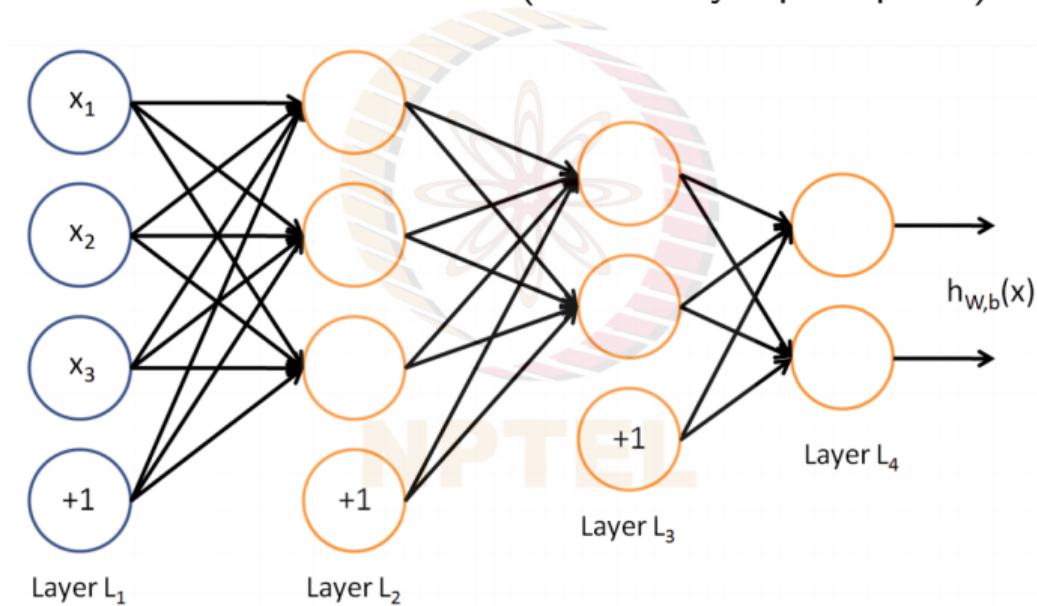


Figure Credit: Stanford UFLDL Tutorial

Backpropagation

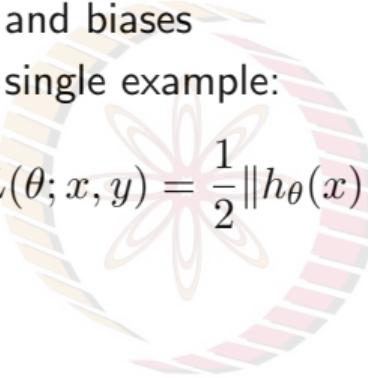
- A fixed training set $\{x^{(i)}, y^{(i)}\}_{i=1}^M$ of M training samples
- Parameters $\theta = \{W, b\}$, weights and biases



Backpropagation

- A fixed training set $\{x^{(i)}, y^{(i)}\}_{i=1}^M$ of M training samples
- Parameters $\theta = \{W, b\}$, weights and biases
- Mean square cost function for a single example:

$$L(\theta; x, y) = \frac{1}{2} \|h_\theta(x) - y\|^2$$

The logo consists of a circular emblem with a stylized orange and red floral or leaf-like pattern in the center, surrounded by concentric circles of the same colors. Below the emblem, the word "NPTEL" is written in a large, bold, orange sans-serif font.

NPTEL

Backpropagation

- A fixed training set $\{x^{(i)}, y^{(i)}\}_{i=1}^M$ of M training samples
- Parameters $\theta = \{W, b\}$, weights and biases
- Mean square cost function for a single example:

$$L(\theta; x, y) = \frac{1}{2} \|h_\theta(x) - y\|^2$$

- Overall cost function is given by:

$$\begin{aligned} L(\theta) &= \frac{1}{M} \sum_{i=1}^M L(\theta; x^{(i)}, y^{(i)}) \\ &= \frac{1}{2M} \sum_{i=1}^M \|h_\theta(x^{(i)}) - y^{(i)}\|^2 \end{aligned}$$

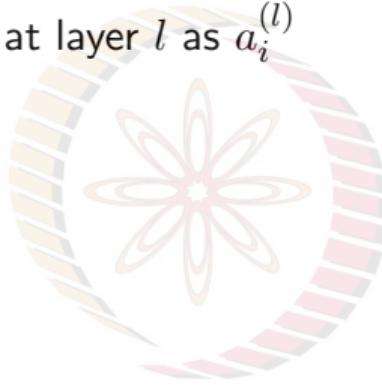
Backpropagation: Notations

- We have n_l layers in the network, $l = 1, 2, \dots, n_l$



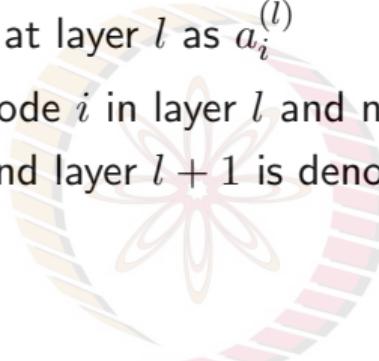
Backpropagation: Notations

- We have n_l layers in the network, $l = 1, 2, \dots, n_l$
- We denote activation of node i at layer l as $a_i^{(l)}$



Backpropagation: Notations

- We have n_l layers in the network, $l = 1, 2, \dots, n_l$
- We denote activation of node i at layer l as $a_i^{(l)}$
- We denote weight connecting node i in layer l and node j in layer $l + 1$ as $W_{ij}^{(l)}$. The weight matrix between layer l and layer $l + 1$ is denoted as $W^{(l)}$



Backpropagation: Notations

- We have n_l layers in the network, $l = 1, 2, \dots, n_l$
- We denote activation of node i at layer l as $a_i^{(l)}$
- We denote weight connecting node i in layer l and node j in layer $l + 1$ as $W_{ij}^{(l)}$. The weight matrix between layer l and layer $l + 1$ is denoted as $W^{(l)}$
- For a 3-layer network shown earlier, compact vectorized form of a **forward pass** to compute neural network's output is shown below:

$$z^{(2)} = W^{(1)}x + b^{(1)}$$

NPTEL

Backpropagation: Notations

- We have n_l layers in the network, $l = 1, 2, \dots, n_l$
- We denote activation of node i at layer l as $a_i^{(l)}$
- We denote weight connecting node i in layer l and node j in layer $l + 1$ as $W_{ij}^{(l)}$. The weight matrix between layer l and layer $l + 1$ is denoted as $W^{(l)}$
- For a 3-layer network shown earlier, compact vectorized form of a **forward pass** to compute neural network's output is shown below:

$$\begin{aligned}z^{(2)} &= W^{(1)}x + b^{(1)} \\a^{(2)} &= f(z^{(2)})\end{aligned}$$

Backpropagation: Notations

- We have n_l layers in the network, $l = 1, 2, \dots, n_l$
- We denote activation of node i at layer l as $a_i^{(l)}$
- We denote weight connecting node i in layer l and node j in layer $l + 1$ as $W_{ij}^{(l)}$. The weight matrix between layer l and layer $l + 1$ is denoted as $W^{(l)}$
- For a 3-layer network shown earlier, compact vectorized form of a **forward pass** to compute neural network's output is shown below:

$$z^{(2)} = W^{(1)}x + b^{(1)}$$

$$a^{(2)} = f(z^{(2)})$$

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$

Backpropagation: Notations

- We have n_l layers in the network, $l = 1, 2, \dots, n_l$
- We denote activation of node i at layer l as $a_i^{(l)}$
- We denote weight connecting node i in layer l and node j in layer $l + 1$ as $W_{ij}^{(l)}$. The weight matrix between layer l and layer $l + 1$ is denoted as $W^{(l)}$
- For a 3-layer network shown earlier, compact vectorized form of a **forward pass** to compute neural network's output is shown below:

$$z^{(2)} = W^{(1)}x + b^{(1)}$$

$$a^{(2)} = f(z^{(2)})$$

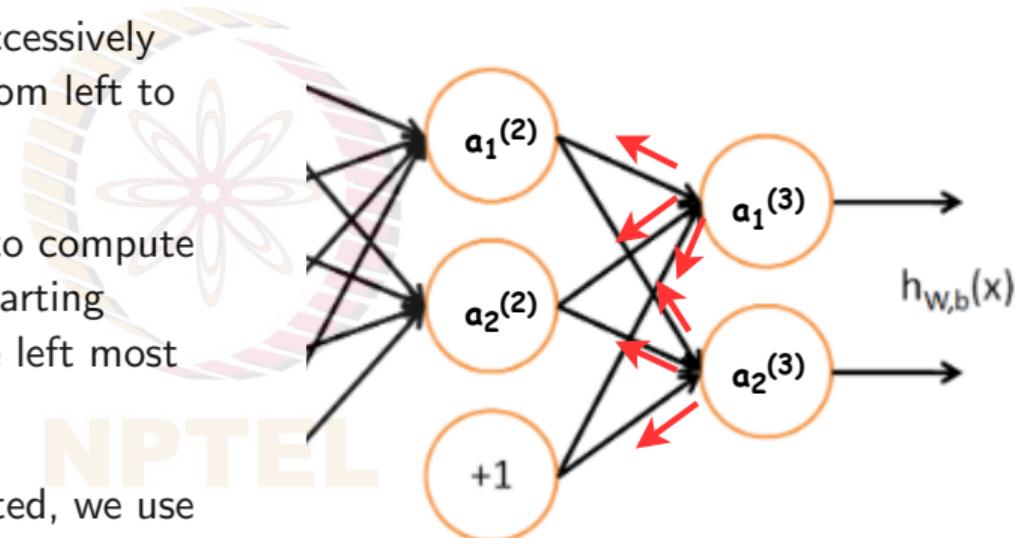
$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$

$$h(x) = a^{(3)} = f(z^{(3)})$$

- Function f can denote any activation function such as sigmoid, ReLU, identity, etc.

Backpropagation: Backward Pass

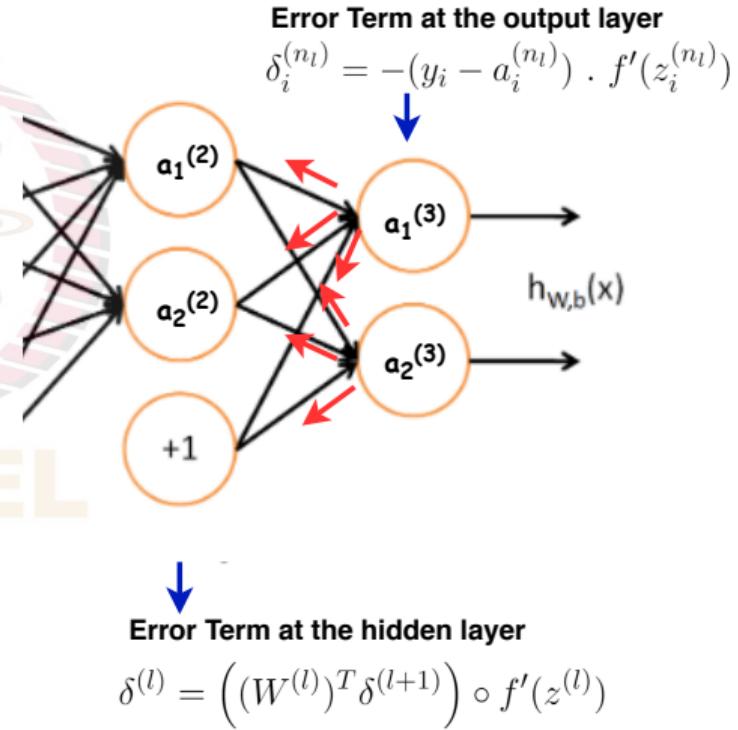
- During the forward pass, we successively compute each layer's outputs from left to right.
- During backward pass, we aim to compute derivatives of each parameter starting from the right most layer to the left most one i.e., layer $n_l, n_l - 1, \dots, 1$.
- Once the derivatives are computed, we use Gradient Descent to update the parameters.



Backpropagation: Backward Pass

- For each node, we define an **error term** $\delta_i^{(l)}$ to denote how much the node was responsible for the loss computed
- If $l = n_l$ i.e., **last layer**, error term computation is straightforward, since we directly take derivative of loss function (MSE, in this case, between output and target values)

$$\delta_i^{(n_l)} = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$



Backpropagation: Backward Pass

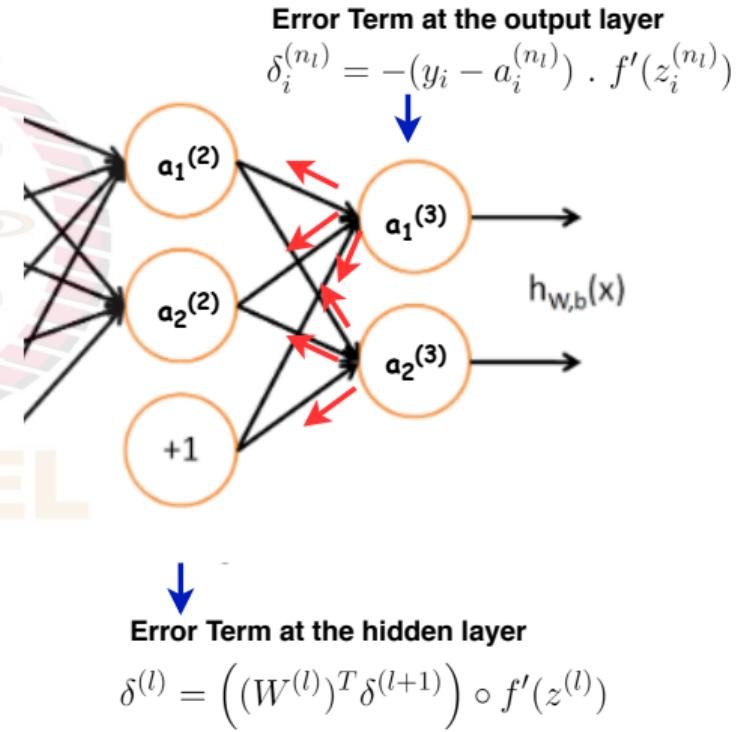


- To compute error term for **hidden layers**,
 $l = n_l - 1, n_l - 2, \dots$, we rely on error terms from subsequent layers

Backpropagation: Backward Pass

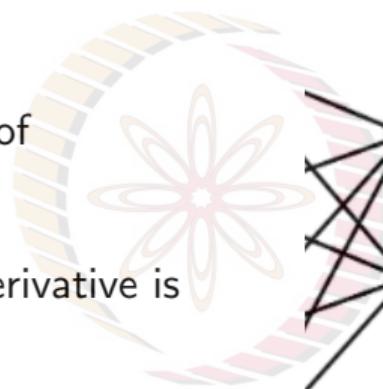
- To compute error term for **hidden layers**,
 $l = n_l - 1, n_l - 2, \dots$, we rely on error terms from subsequent layers
- In particular, we compute error term as sum of error terms in next layer, weighted by weights along connections to next layer:

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ij}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

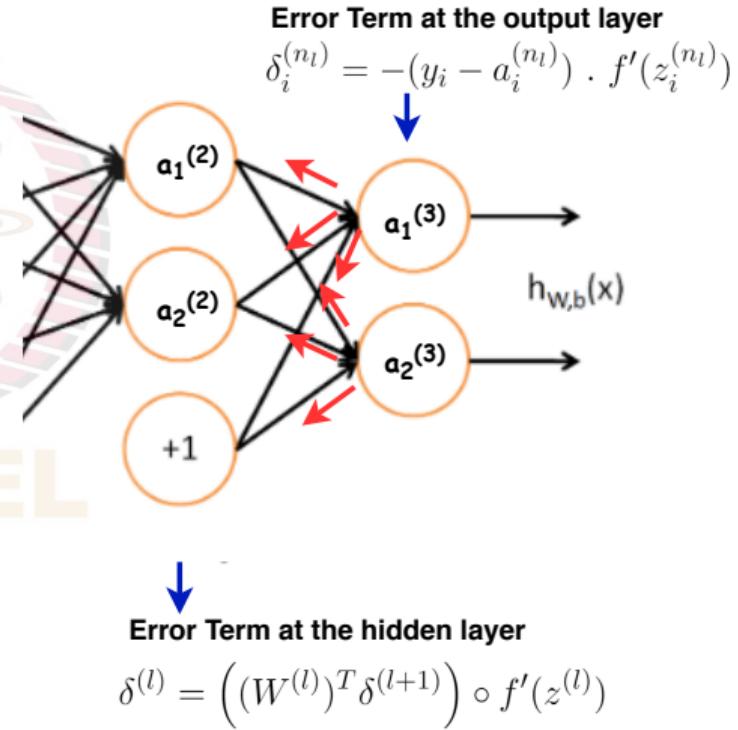


Backpropagation: Backward Pass

- Note that f' denotes derivative of activation function
- For a linear neuron $f(x) = x$, derivative is 1
- For a sigmoid neuron $f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$, derivative turns out to be $\sigma(x)(1 - \sigma(x))$ (How? Homework!)



NPTEL



Backpropagation: Algorithm

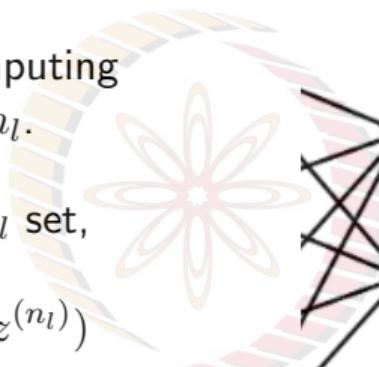
- Perform a feedforward pass, computing the activations for layers $1, 2, \dots, n_l$.

- For each output unit i in layer n_l set,

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \circ f'(z^{(n_l)})$$

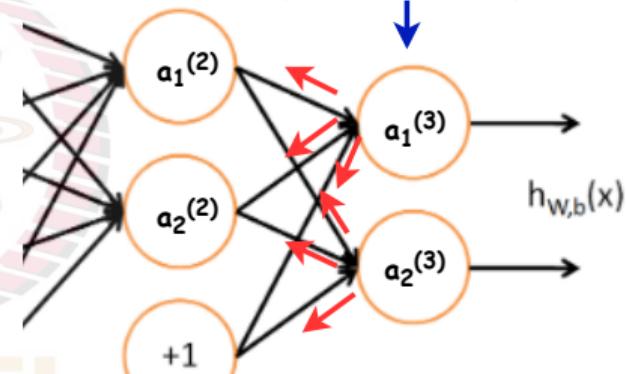
- For $l = n_l - 1, n_l - 2, n_l - 3, \dots, 2$
For each node in layer l set,

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)} \right) \circ f'(z^{(l)})$$



Error Term at the output layer

$$\delta_i^{(n_l)} = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$



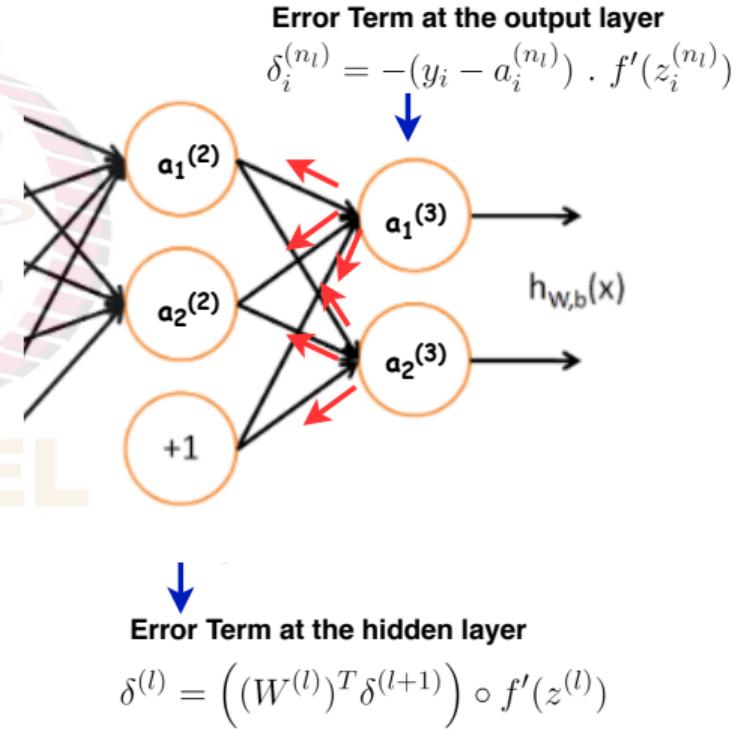
Error Term at the hidden layer

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)} \right) \circ f'(z^{(l)})$$

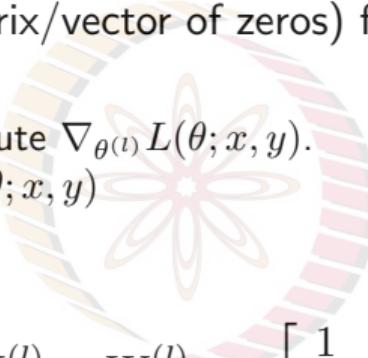
Backpropagation: Algorithm

- Compute the desired partial derivatives,
as:

$$\nabla_{W^{(l)}} L(W, b; x, y) = \delta^{l+1} (a^{(l)})^T$$
$$\nabla_{b^{(l)}} L(W, b; x, y) = \delta^{l+1}$$



Gradient Descent using Backpropagation

- 
- ① Set $\Delta W^{(l)} := 0, \Delta b^{(l)} = 0$ (matrix/vector of zeros) for all l .
 - ② For $i = 1$ to M
 - ① Use backpropagation to compute $\nabla_{\theta^{(l)}} L(\theta; x, y)$.
 - ② Set $\Delta \theta^{(l)} := \Delta \theta^{(l)} + \nabla_{\theta^{(l)}} L(\theta; x, y)$
 - ③ Update the parameters:

$$W^{(l)} = W^{(l)} - \eta \left[\frac{1}{M} \Delta W^{(l)} \right]$$

$$b^{(l)} = b^{(l)} - \eta \left[\frac{1}{M} \Delta b^{(l)} \right]$$

- ④ Repeat for all points until convergence.

Homework

Readings

- Chapter 4, Deep Learning Book
- Chapter 6, Deep Learning Book
- Stanford UFLDL tutorial
- Stanford CS231n Notes



Exercises

- Work out the derivative of the sigmoid and tanh activation functions
- Sigmoid activation function: $\sigma(x) = \frac{1}{1+e^{-x}}$
- Tanh activation function: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

References



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



NPTEL