Deep Learning for Computer Vision

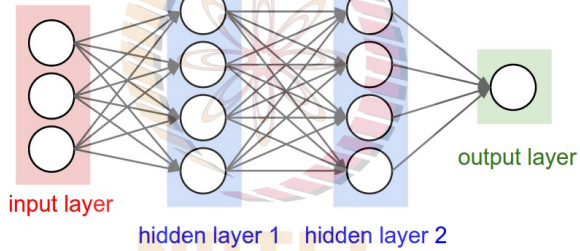
Feedforward Neural Networks and Backpropagation

Vineeth N Balasubramanian

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

Feedforward Networks

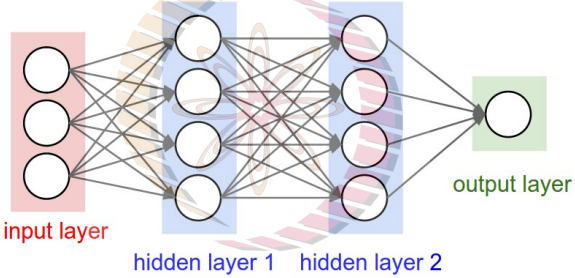
- A feedforward neural network, also called a multi-layer perceptron, is a collection of neurons, organized in *layers*.



- It is used to approximate some function f^*. For instance, f^* could be a classifier that maps an input vector $\boldsymbol{x}$ to a category $\boldsymbol{y}$.
- The neurons are arranged in the form of a directed acyclic graph i.e., the information only flows in one direction - input $\boldsymbol{x}$ to output $\boldsymbol{y}$. 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 $f^*(\boldsymbol{x})$ at different points, in the form of a dataset $\{\boldsymbol{x}_i, y_i\}_{i=1}^{M}$.
- Our neural network defines a function $y = f(\boldsymbol{x}; \boldsymbol{\theta})$. 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:

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

- 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.

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

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.

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

- The reverse is also true i.e.,

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

- 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

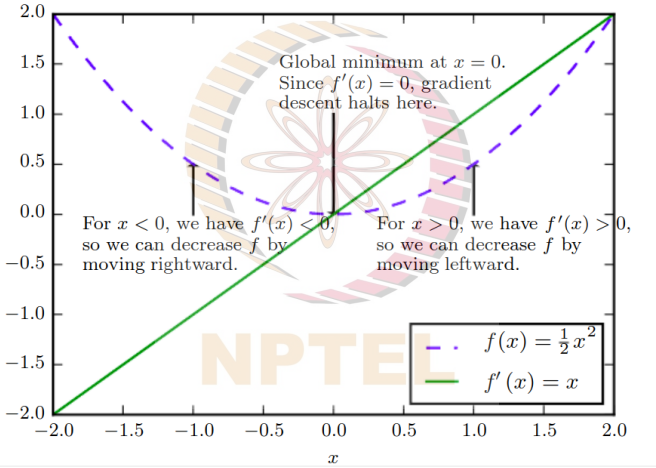


Figure Credit: Goodfellow et al, DL Book Ch 4

Why Negative Gradient?

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

NPTEL

Why Negative Gradient?

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

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

NPTEL

Why Negative Gradient?

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

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

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

- Since $\|\boldsymbol{u}\|_2 = 1$, we can minimize the above function when $\beta = 180^0$, i.e. when **u** is the direction of **negative** gradient

How to use Gradient Descent

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

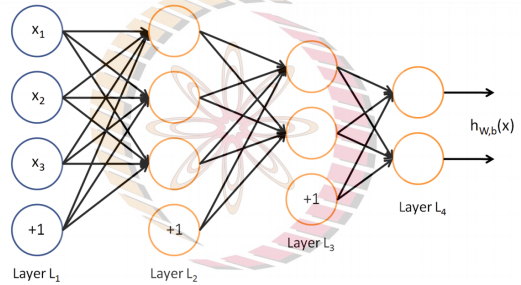
- Start with a random weight vector $\boldsymbol{\theta}$.
- Compute the loss function over the dataset, i.e., $L(\theta)$ with the current network, using a suitable loss function such as mean-squared error
- Compute the gradients of the loss function with respect to each weight value $\frac{\delta L}{\delta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

$$\boldsymbol{\theta}_i^{next} = \boldsymbol{\theta}_i^{curr} - \eta \frac{\delta L}{\delta \boldsymbol{\theta}_i^{curr}}$$

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

Gradient Descent

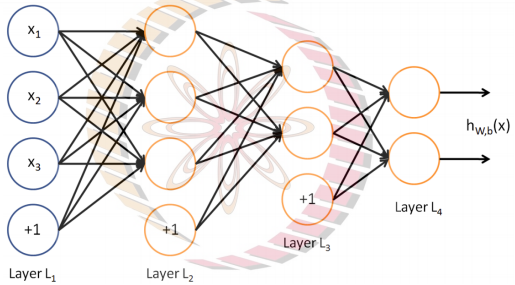
- A feedforward neural network is a composition of multiple functions, organized as layers



- 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.to 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.

NPTEL

Backpropagation

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

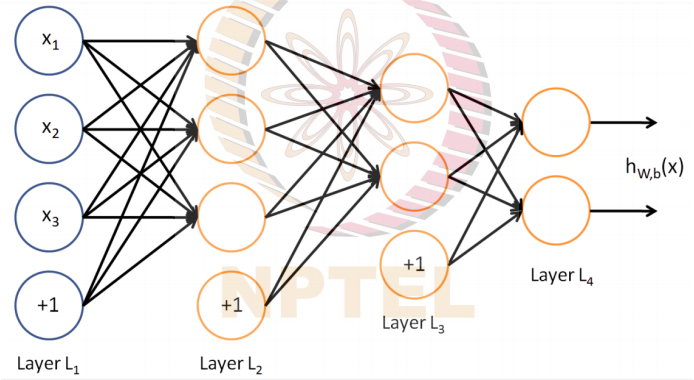
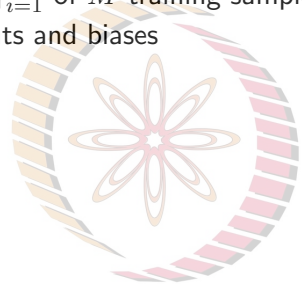


Figure Credit: Stanford UFLDL Tutorial

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:

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

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:

$$L(\theta) = \frac{1}{M}\sum_{i=1}^{M} L(\theta; x^{(i)}, y^{(i)})$$

$$= \frac{1}{2M}\sum_{i=1}^{M} \|h_\theta(x^{(i)} - y^{(i)}\|^2$$

Backpropagation: Notations

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

Backpropagation: Notations

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

NPTEL

Backpropagation: Notations

- We have n_l layers in the network, $l = 1, 2, .., n_l$
- 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 $W^{(l)}$

NPTEL

Backpropagation: Notations

- We have n_l layers in the network, $l = 1, 2, .., n_l$
- 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, $l = 1, 2, .., n_l$
- 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:

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

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

NPTEL

Backpropagation: Notations

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

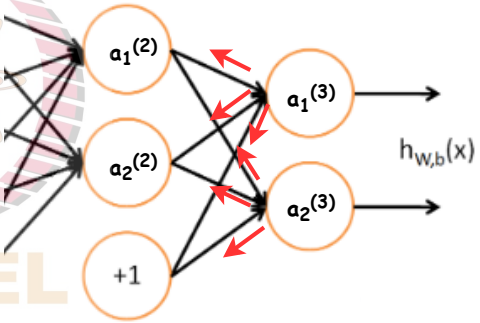
- We have n_l layers in the network, $l = 1, 2, .., n_l$
- We denote activation of node i at layer l as $a_i^{(l)}$
- We denote weight connecting node i in layer l and node j in layer $l+1$ as $W_{ij}^{(l)}$. The weight matrix between layer l and layer $l+1$ is denoted as $W^{(l)}$
- For a 3-layer network 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 $n_l, n_l - 1, ..., 1$.
- Once the derivatives are computed, we use Gradient Descent to update the parameters.

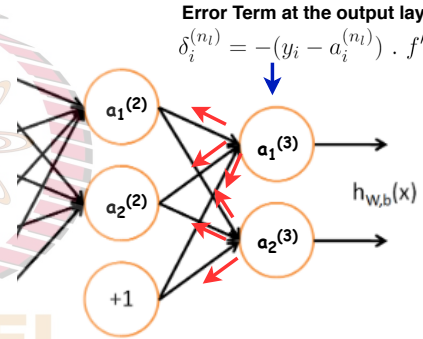


NPTEL

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)

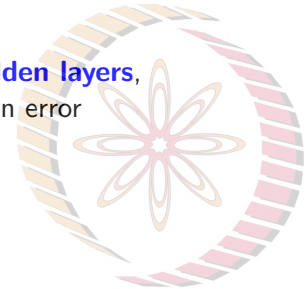
$$\delta_i^{(n_l)} = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$



NPTEL

Backpropagation: Backward Pass

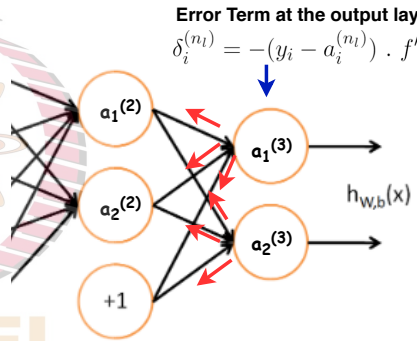
- To compute error term for **hidden layers**, $l = n_l - 1, n_l - 2, ..,$ we rely on error terms from subsequent layers



Backpropagation: Backward Pass

- To compute error term for **hidden layers**, $l = n_l - 1, n_l - 2, ..,$ 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:

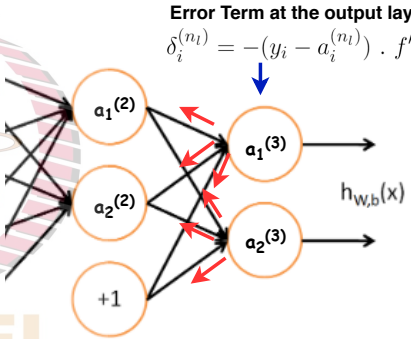
$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ij}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$



NPTEL

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

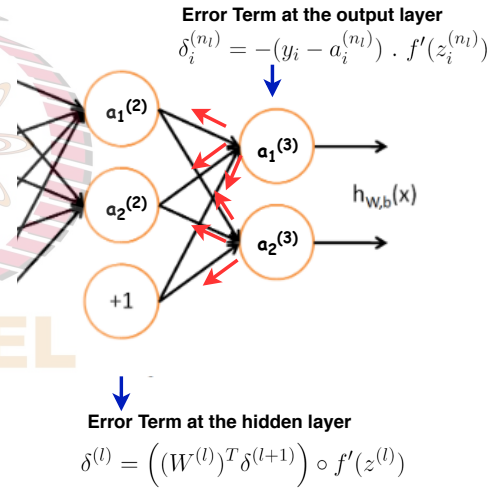
Backpropagation: Algorithm

- Perform a feedforward pass, computing the activations for layers $1, 2, ...n_l$.
- For each output unit i in layer n_l set,

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

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

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

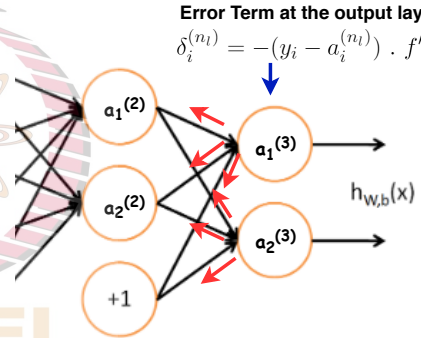


NPTEL

Backpropagation: Algorithm

- Compute the desired partial derivatives, as:

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



NPTEL

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

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

NPTEL