

# Introduction to Deep Learning for Scientists and Engineers

Abhijat Vatsyayan <sup>1</sup>

November 19, 2020

# Summary

- Part I
  - Problem definition (rely on supervised learning)
  - Compute graph and gradients
  - A little about deep learning libraries.
- Part II
  - Need for different architectures
  - Convolution networks
  - Recurrent networks
- Not covered
  - Diagram of neurons
  - History
  - Recent advances and business context
  - Tutorial on pytorch or keras
- Expectation and whats next

# Extending derivatives to functions that accept vectors

For a function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ , or, equivalently,  $y = f(\vec{x})$ , the gradient is:

$$\nabla_x f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \quad (1)$$

For finding minima, use gradient descent (iterate):

$$\vec{x} = \vec{x} - \eta \nabla_x f \quad (2)$$

Extending this to  $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ , or  $\vec{y} = f(\vec{x})$ , we define the Jacobian as:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial y_1}{\partial x_{11}} & \frac{\partial y_1}{\partial x_{12}} & \cdots & \frac{\partial y_1}{\partial x_{1n}} \\ \vdots & & & \\ \frac{\partial y_m}{\partial x_{11}} & \frac{\partial y_m}{\partial x_{12}} & \cdots & \frac{\partial y_m}{\partial x_{1n}} \end{bmatrix} \quad (3)$$

Note: Gradient is defined for  $y \in \mathbb{R}$ .

- Can we define another function  $L(\vec{y}) : \mathbb{R}^m \rightarrow \mathbb{R}$ ?
- What does this function mean?

# Matrix multiplication as a function

$$x \in \mathbb{R}^n, y \in \mathbb{R}^m, w \in \mathbb{R}^{m \times n}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & & \vdots \\ w_{m1} & w_{m2} & \dots & w_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$
$$\vec{y} = w\vec{x}$$

# Derivatives

For  $m = 2, n = 3$

$$y_1 = w_{11}x_1 + w_{12}x_2 + w_{13}x_3 \quad (4)$$

$$y_2 = w_{21}x_1 + w_{22}x_2 + w_{23}x_3 \quad (5)$$

$$\begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \quad (6)$$

$$\frac{d\vec{y}}{d\vec{x}} = w \quad (7)$$

# Element wise operations

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (8)$$

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x)) \quad (9)$$

$$\sigma\left(\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_m \end{bmatrix}\right) = \begin{bmatrix} \sigma(z_1) \\ \sigma(z_2) \\ \vdots \\ \sigma(z_m) \end{bmatrix} \quad (10)$$

# Derivatives in multiple dimensions

Partial derivatives of  $f(x, y, z) = \sin(x^2y) + e^z$

$$\frac{\partial f}{\partial x} = 2xy \cos(x^2y) \quad (11)$$

$$\frac{\partial f}{\partial y} = x^2 \cos(x^2y) \quad (12)$$

$$\frac{\partial f}{\partial z} = e^z \quad (13)$$

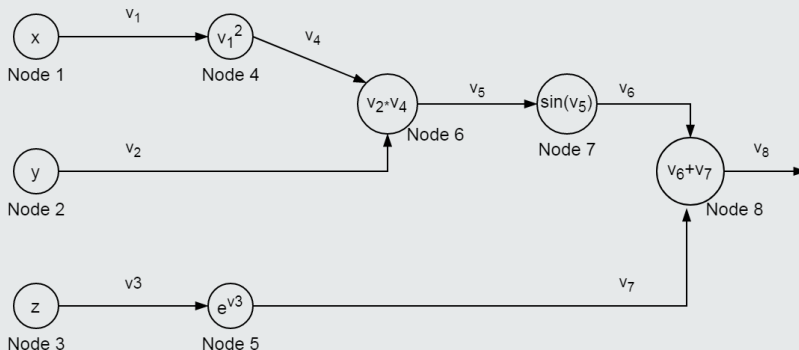
Computing the derivative

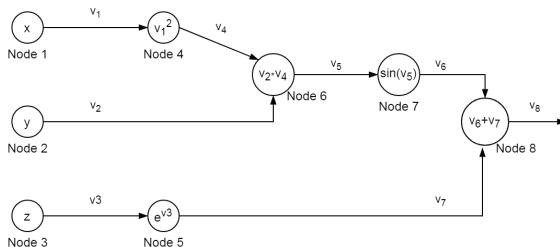
- Analytically (shown above)
- Numerically  $f'(x) \approx \frac{f(x+\delta x) - f(x-\delta x)}{2\delta x}$
- Auto-diff. Deep learning libraries use reverse mode auto-diff.



# Computation graph

Compute graph of  $f(x, y, z) = \sin(x^2 y) + e^z$





$$\frac{\partial v_8}{\partial z} = \frac{\partial v_8}{\partial v_7} \frac{\partial v_7}{\partial v_3} \frac{\partial v_3}{\partial z} \quad (14)$$

$$= 1 \cdot e^{v_3} \cdot 1 \quad (15)$$

$$\frac{\partial v_8}{\partial z} = \frac{\partial f}{\partial z} = e^{v_3} = e^z \quad (16)$$

$$\frac{\partial v_8}{\partial y} = \frac{\partial v_8}{\partial v_6} \frac{\partial v_6}{\partial v_5} \frac{\partial v_5}{\partial v_2} \frac{\partial v_2}{\partial y} \quad (17)$$

$$= 1 \cdot \cos(v_5) \cdot v_4 \cdot 1 \quad (18)$$

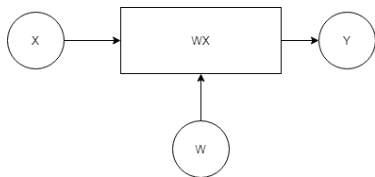
$$\frac{\partial f}{\partial y} = \frac{\partial v_8}{\partial y} = \cos(v_5) v_4 \quad (19)$$

$$= \cos(v_2 v_4) v_4 \quad (20)$$

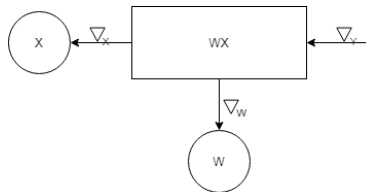
$$= \cos(v_1^2 y) v_1^2 \quad (21)$$

$$= \cos(x^2 y) x^2 \quad (22)$$

# Unit operations (and a little bit of python)



```
1 class MultiplicationLayer1D:
2     def __init__(self):
3         self.cache = {}
4
5     def forward(self, x, w):
6         self.cache['x'] = x
7         self.cache['w'] = w
8         return w * x
9
10    def backprop(self, incoming_grad):
11        x_grad = self.cache['w']
12        w_grad = self.cache['x']
13        x_grad *= incoming_grad
14        w_grad *= incoming_grad
15        return x_grad, w_grad
```



```
1 class AdditionLayer:
2     def __init__(self):
3         self.cache = {}
4
5     def forward(self, x, w):
6         self.cache['x'] = x
7         self.cache['w'] = w
8         return w + x
9
10    def backprop(self, incoming_grad):
11        x_grad = incoming_grad
12        w_grad = incoming_grad
13        return x_grad, w_grad
```

# Fitting a function to data

## Description (supervised learning)

Given a set of data  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ , can we find a function  $y = f(x)$  that "fits" this data?

## Questions

- What is this function  $f(x)$ ?
- What does "fit" mean?
- How do we know this works?
- What kinds of problems can we solve?

# More about $f(x)$

## Class of functions

Starting with a function  $f(x; \theta_1, \theta_2, \dots, \theta_n)$  where  $x$  is the input to the function and  $\theta$ s are its parameters, we need to find the set of  $\theta$ s that best "fits" the give data  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

## Class of linear functions

Consider  $f(x) = ax + b$ . If we can say, with some confidence, that our data is linearly related, we need to find  $\theta_1 = a$ ,  $\theta_2 = b$  that *fits* the given data. We can also write it as  $f(x; a, b) = ax + b$ .

Preferred,

$$f(x; \theta_1, \theta_2) = \theta_1 x + \theta_2 \quad (23)$$

## Mean squared Euclidean distance as one possible measure of fit

Let  $L_i$  be the squared Euclidean distance between the predicted value,  $\hat{y}_i = f(x_i)$  and the actual,  $y_i$ . Then,

$$L_i = z_i^2 \quad (24)$$

$$z_i = y_i - f(x_i) \quad (25)$$

$$= y_i - \theta_1 x_i - \theta_2 \quad (26)$$

Minimizing  $L_i$  with respect to the parameters  $\theta_1$  and  $\theta_2$ ,

$$\frac{\partial L_i}{\partial \theta_1} = \frac{\partial z_i^2}{\partial z_i} \frac{\partial z_i}{\partial \theta_1} \quad (27)$$

$$\frac{\partial L_i}{\partial \theta_2} = \frac{\partial z_i^2}{\partial z_i} \frac{\partial z_i}{\partial \theta_2} \quad (28)$$

For  $n$  data points, mean loss is

$$L = \frac{1}{n} \sum_{i=1}^{i=n} L_i = \frac{1}{n} \sum_{i=1}^{i=n} (y_i - \theta_1 x_i - \theta_2)^2$$

In practice

- Choose a small (64 or 128) random subset of training data.
- Compute predicted values, then loss.
- Compute gradients of loss W.R.T. parameters then update parameters.

An **epoch** refers to a single iteration over all training data.

## Two different spaces

- Space spanned by  $x$  and  $y$ . Optimization tries to find the surface (model) in this space that best fits the data.
- Space spanned by  $\theta_s$ . We minimize the loss function in this space.

# Using sigmoid

## Definition

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (29)$$

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x)) \quad (30)$$

Assuming a non-linear function (sigmoidal + affine prior)

Let our function be  $y = \sigma(\theta_1 x + \theta_2)$ .



# Making a non-linear function

## New function and the squared error

$$z = \sigma(\hat{y}) \quad (31)$$

$$\hat{y} = \theta_1 x + \theta_2 \quad (32)$$

Error  $L_i$  for the  $i$ th data point

$$L_i = (\sigma(\theta_1 x_i + \theta_2) - y_i)^2 \quad (33)$$

# Finding best $\theta$ s

## Minimizing $L_i$

$$L_i = \Delta_i^2 \quad (34)$$

$$\Delta_i = z_i - y_i \quad (35)$$

$$z_i = \sigma(\hat{y}_i) \quad (36)$$

$$\hat{y}_i = \theta_1 x_i + \theta_2 \quad (37)$$

$$\frac{\partial L_i}{\partial \theta_1} = \underbrace{\frac{\partial \Delta_i^2}{\partial \Delta_i} \frac{\partial \Delta_i}{\partial z_i} \frac{\partial z_i}{\partial \hat{y}_i}}_{\text{chain rule}} \frac{\partial \hat{y}_i}{\partial \theta_1} \quad (38)$$

$$\frac{\partial L_i}{\partial \theta_2} = \underbrace{\frac{\partial \Delta_i^2}{\partial \Delta_i} \frac{\partial \Delta_i}{\partial z_i} \frac{\partial z_i}{\partial \hat{y}_i}}_{\text{chain rule}} \frac{\partial \hat{y}_i}{\partial \theta_2} \quad (39)$$

# Finding best $\theta$ s

Updates to  $\theta_1$  and  $\theta_2$

$$\frac{\partial L_i}{\partial \theta_1} = \underbrace{\frac{\partial \Delta_i^2}{\partial \Delta_i} \frac{\partial \Delta_i}{\partial z_i} \frac{\partial z_i}{\partial \hat{y}_i}}_{\text{chain rule}} \frac{\partial \hat{y}_i}{\partial \theta_1} \quad (40)$$

$$\frac{\partial L_i}{\partial \theta_2} = \underbrace{\frac{\partial \Delta_i^2}{\partial \Delta_i} \frac{\partial \Delta_i}{\partial z_i} \frac{\partial z_i}{\partial \hat{y}_i}}_{\text{chain rule}} \frac{\partial \hat{y}_i}{\partial \theta_2} \quad (41)$$

$$\theta_{1_{p+1}} = \theta_{1_p} - \eta \frac{\partial L_i}{\partial \theta_1} \quad (42)$$

$$\theta_{2_{p+1}} = \theta_{2_p} - \eta \frac{\partial L_i}{\partial \theta_2} \quad (43)$$

# Finding best $\theta$ s

Updates to  $\theta_1$  and  $\theta_2$

$$\theta_{1_{p+1}} = \theta_{1_p} - \eta \frac{\partial L_i}{\partial \theta_1} \quad (44)$$

$$\theta_{2_{p+1}} = \theta_{2_p} - \eta \frac{\partial L_i}{\partial \theta_2} \quad (45)$$

Vectorized format

$$\vec{\theta}_{p+1} = \vec{\theta}_p - \eta \nabla_{\vec{\theta}} L \quad (46)$$

# Using vectors

$$x \in \mathbb{R}^n, y \in \mathbb{R}^m, w \in \mathbb{R}^{m \times n}$$

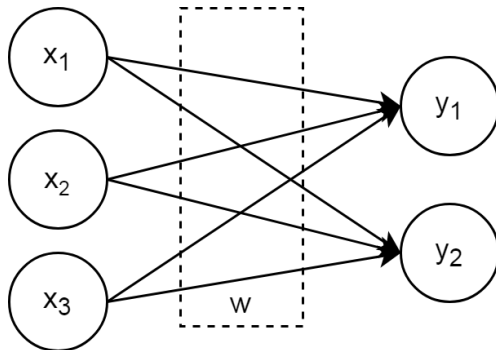
$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & & \vdots \\ w_{m1} & w_{m2} & \dots & w_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

For  $m = 2, n = 3$

$$y_1 = w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1 \quad (47)$$

$$y_2 = w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + b_2 \quad (48)$$

# Mandatory network diagram



# More layers

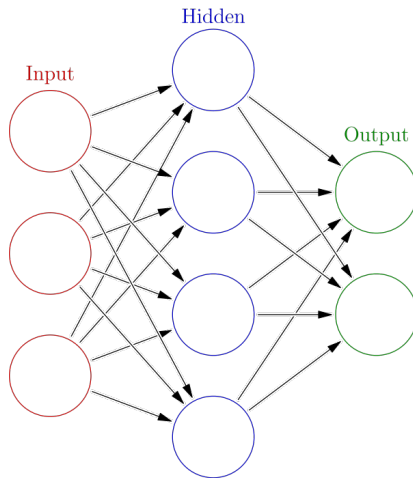


Image from [https://en.wikipedia.org/wiki/Artificial\\_neural\\_network](https://en.wikipedia.org/wiki/Artificial_neural_network)

# Multiple layers

## Layers (composition) of $ax + b$ style functions



$$y = \mathbb{W}x + b$$

$$z = \mathbb{U}y + c = \mathbb{U}(\mathbb{W}x + b) + c$$

$$= (\mathbb{U}\mathbb{W})x + (\mathbb{U}b + c) = \mathbb{V}x + d$$

## Introducing non-linearity with element-wise sigmoid

$$y = \sigma(\mathbb{W}x + b)$$

$$z = \sigma(\mathbb{U}y + c)$$



# Multi-layer, feed forward, non-polynomial

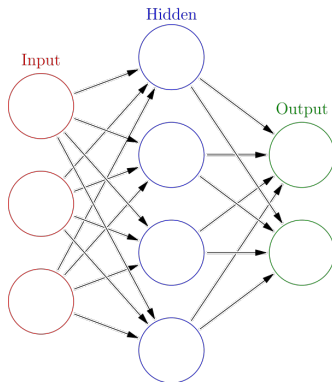
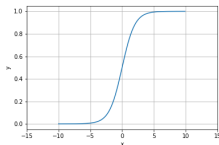


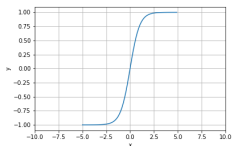
Image from [https://en.wikipedia.org/wiki/Artificial\\_neural\\_network](https://en.wikipedia.org/wiki/Artificial_neural_network)

# Activation functions

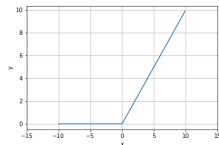
$$f(x) = \frac{1}{1 + e^{-x}}$$



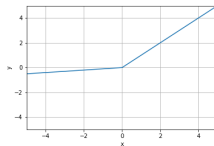
$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



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



$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0.1x, & \text{otherwise} \end{cases}$$



# What can these functions (neural networks) do?

*A standard multilayer feed-forward network with a locally bounded piecewise continuous activation function can approximate any continuous function to any degree of accuracy if and only if the network's activation function is not a polynomial.*

*-Leshno et al., 1993*

*In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feed forward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity.*

*-Cybenko, 1989*

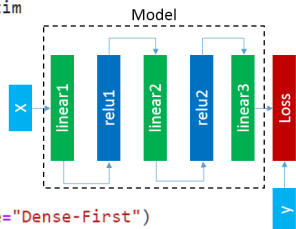
# Python code

```
# Build the network
import core.np.Activations as act
import core.np.Loss as loss
from core.np.utils import to_one_hot
import core.np.Optimization as autodiff_optim
import core.np.regularization as reg
from core import info, debug, log_at_info
import time
import matplotlib.pyplot as plt
```

```
x_node = node.VarNode('x')
y_target_node = node.VarNode('yt')
```

```
linear1 = node.DenseLayer(x_node, 100, name="Dense-First")
relu1 = act.RelUNode(linear1, name="ReLU-First")
linear2 = node.DenseLayer(relu1, 200, name="Dense-Second")
relu2 = act.RelUNode(linear2, name="ReLU-Second")
linear3 = node.DenseLayer(relu2, 10, name="Dense-Third")
```

```
loss_node = loss.LogitsCrossEntropy(linear3, y_target_node, name="XEnt")
```



# Steps

- Settle on a class of function with parameters to be optimized.
- Define a secondary function - the loss function.
- Split data into training set, validation set and test set. Use randomization. For the following discussion, we ignore validation set.
- Initialize model weights ( $W$ 's or  $\theta$ s), typically using small random numbers.
- Iterate over the data in batches
  - compute the predicted value
  - compute the loss
  - compute gradients
  - update parameters
- Keep track of loss and stop when loss stops decreasing.
- Compute loss for test set.

# Why now?

- Activation functions
- Initialization approaches
- New optimization algorithms\*
- Architectures for better propagation of gradients
- Availability of massive amounts of data - generation + storage
- Advances in processor speeds and GPUs
- New ways of working and collaborating

# Learning, libraries etc.

## Why use libraries

- Define compute graph easily
- calculates gradient
- Move between CPU and GPU
- Model libraries, trained models, utilities

## Libraries

- Torch and Theano
- Tensorflow
- Keras
- Pytorch

# Next

## Part II

- Autoencoders and PCA
- Convolution neural networks
- Recurrent neural networks
- Transfer learning
- GANs (possibly!)



# References



Automatic differentiation in machine learning: A survey

Atilim Gunes Baydin, Barak A. Pearlmutter & Alexey Andreyevich Radul

*arxiv* <http://arxiv.org/abs/1502.05767>