Part 1: Basics

Introduction to Neural Networks

Deep Learning for NLP

Kevin Patel

ICON 2017

December 21, 2017

Overview

Our brains are so awesome, that we cannot replicate their computation

Our brains are so awesome, that we cannot replicate their computation

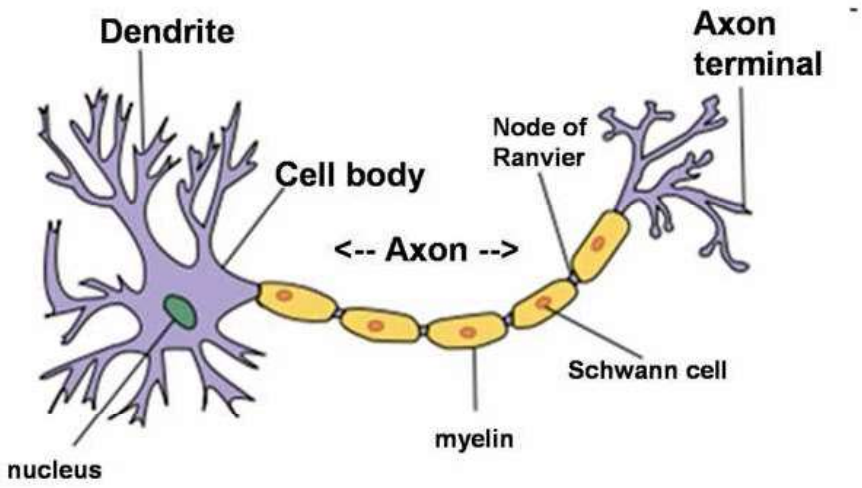
Our brains' capability is so limited, that we have failed to replicate their computation

Purpose of Artificial Intelligence

- Human Like AI: An AI which functions like a human and has similar characteristics
- Beneficial AI: An AI which works in a fundamentally different manner, but gets the job done

The Human Brain

- Brain - a large network of neurons
- Neuron - a cell capable of receiving, processing and transmitting information via electric and chemical signals

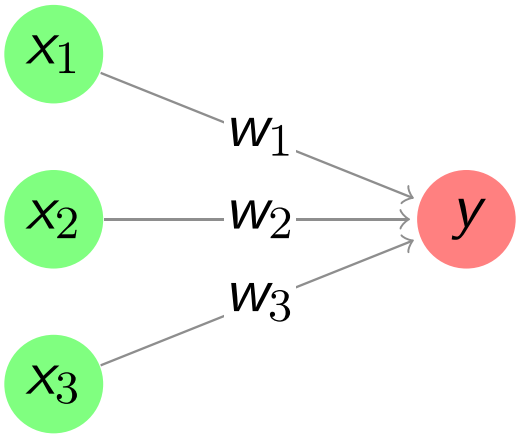


- Dendrites receive input
- Axon transmits output

Perceptron

- A simple artificial neuron Rosenblatt (1958)
- Input: one or more binary values x_i
- Output: single binary value y
- Output computed using weighted sum of inputs and a threshold
- Giving different weights to different features while making a decision

Perceptron (Contd.)



$$y = \begin{cases} 1, & \text{if } \sum w_i x_i > \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

Some Conventions

- Inputs also treated as neurons (no input, output is the actual value of the feature)
- Rewrite $\sum w_i x_i$ as $w.x$
- Move threshold to the other side of the equation; Call it bias $b = -\text{threshold}$

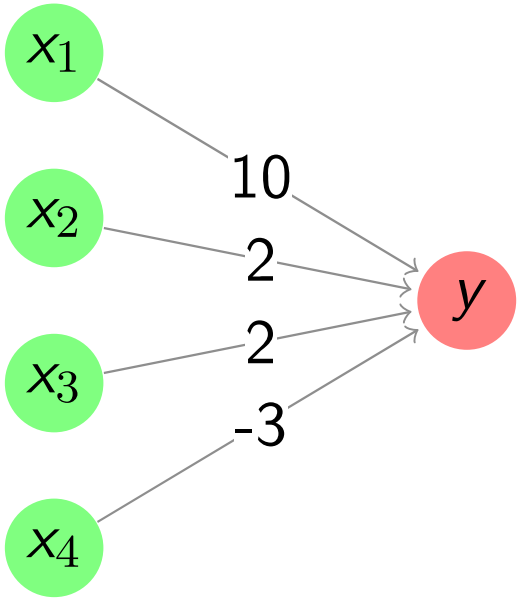
$$y = \begin{cases} 1, & \text{if } w.x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

Bias

- An indication of how easy it is for a neuron to fire
- The higher the value of bias, the easier it is for the neuron to fire
- Consider it as a prior inclination towards some decision
  - The higher your initial inclination, the smaller the amount of extra push needed to finally make some decision

Perceptron Example

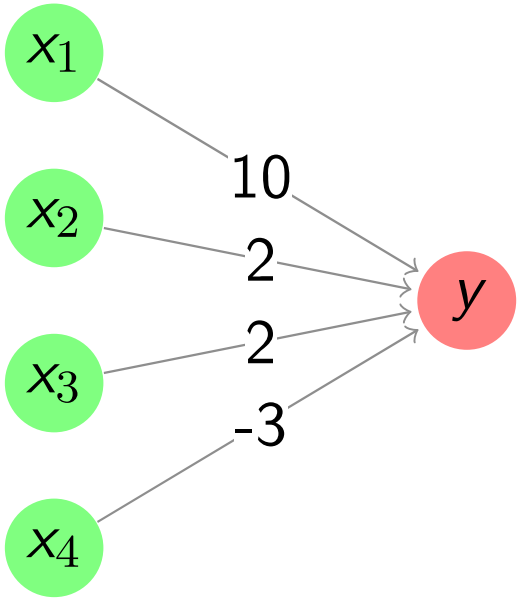
Should I go to lab?



- x_1: My guide is here
- x_2: Collaborators are in lab
- x_3: The buses are running
- x_4: Tasty tiffin in the mess
- b: My inclination towards going to the lab no matter what

Perceptron Example

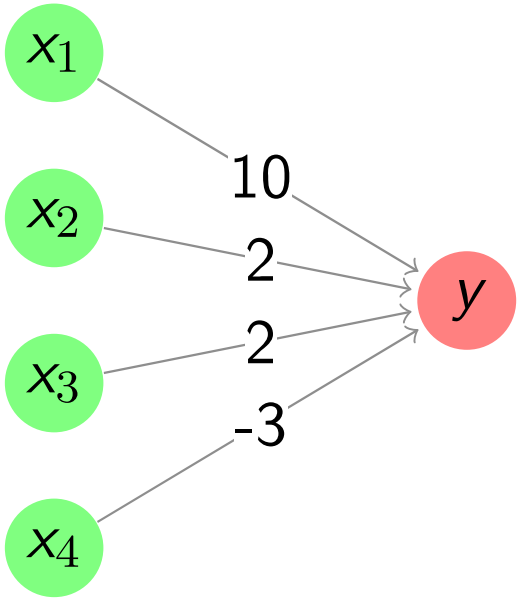
Should I go to lab?



- x_1: My guide is here
- x_2: Collaborators are in lab
- x_3: The buses are running
- x_4: Tasty tiffin in the mess
- b: My inclination towards going to the lab no matter what
- What if $b = -3$?

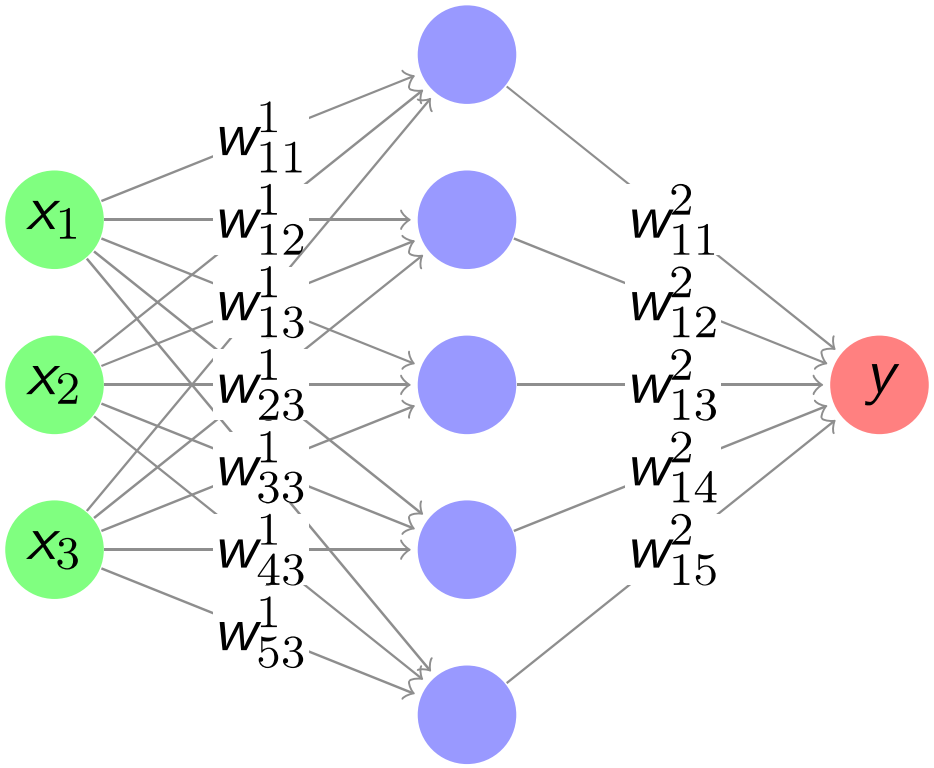
Perceptron Example

Should I go to lab?



- x_1: My guide is here
- x_2: Collaborators are in lab
- x_3: The buses are running
- x_4: Tasty tiffin in the mess
- b: My inclination towards going to the lab no matter what
- What if $b = -3$?
- What if $b = 1$?

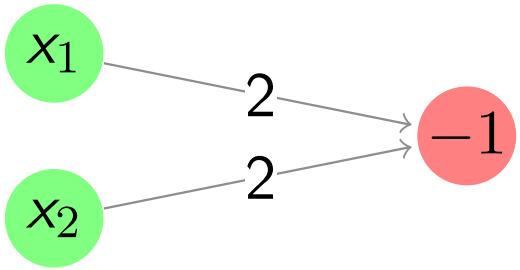
Network of Perceptrons



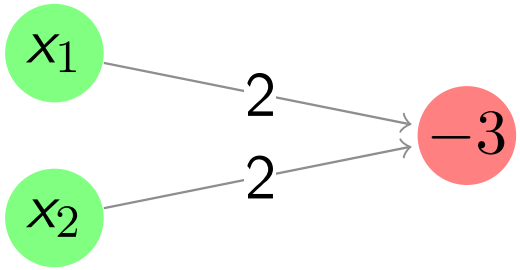
- Outputs of perceptrons fed into next layer
- Simpler decisions made at initial layers used as features for making complex decisions.

Perceptrons and Logic Gates

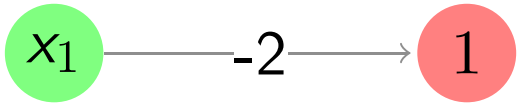
OR Gate



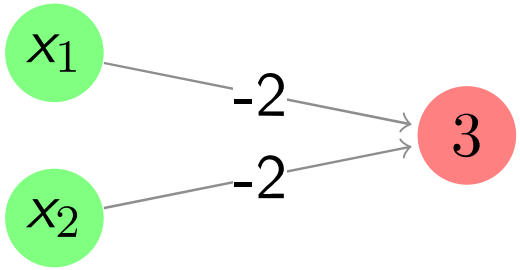
AND Gate



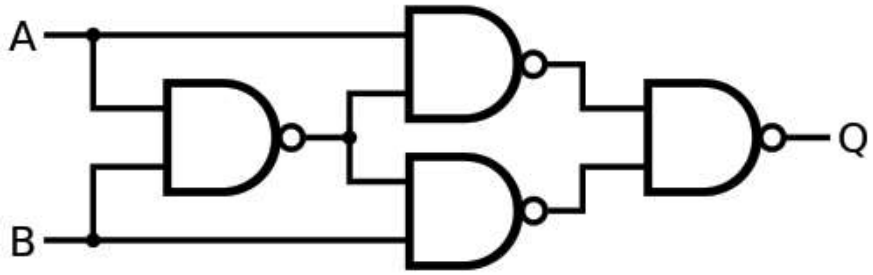
NOT Gate



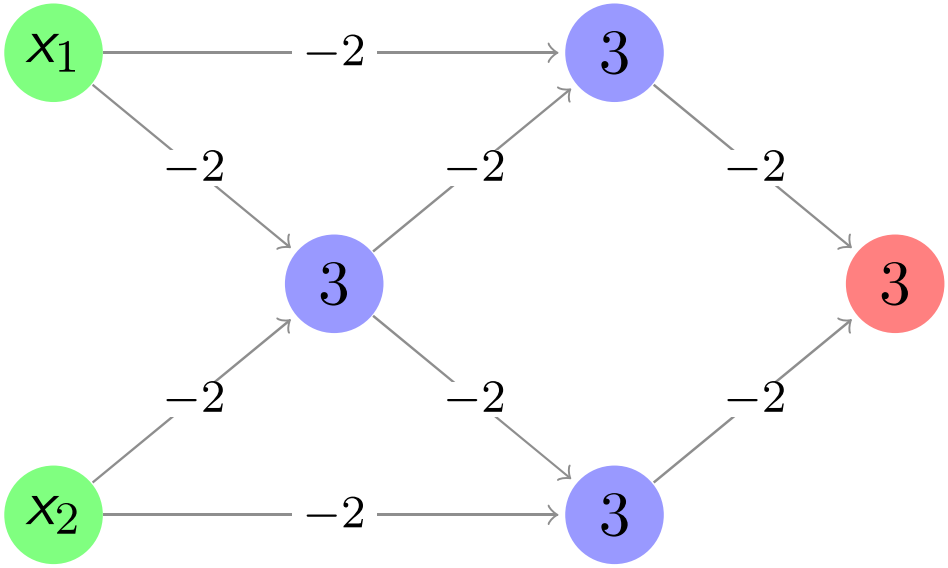
NAND Gate



Perceptrons and Logic Gates (contd.)



XOR Gate



Perceptrons and Logic Gates (contd.)

- NAND gates are universal gates
- Perceptrons can simulate NAND gates
- Therefore, Perceptrons are universal for computation
- Good News: Network of perceptrons is as capable as any other computing device
- Bad News: Is it just another fancy NAND gate?
  - Silver Lining: Learning algorithms can automatically figure out weights and biases, whereas we need to explicitly design circuits using NAND gates

Learning

- Simple weight update rule to learn parameters of a single perceptron
- Perceptron Convergence Theorem guarantees that learning will converge to a correct solution in case of linearly separable data.
- However, learning is difficult in case of network of perceptrons
  - Ideally, a learning process involves changing one of the input parameters by a small value, hoping that it will change the output by a small value.
  - For instance, in handwritten digit recognition, if the network is misclassifying *9* as *8*, then we want
  - Here, a small change in parameters of a single perceptron $\Rightarrow$ flipped output $\Rightarrow$ change behavior of entire network
  - Need some machinery such that gradual change in parameters lead to gradual change in output

Learning (contd.)

- If y is a function of x, then change in y *i.e.* Δy is related to change in x *i.e.* Δy as follows (Linear Approximation)

$$\Delta y \approx \frac{dy}{dx}\Delta x$$

- Example

$$\begin{aligned} f(x) &= x^2 \\ f'(x) &= 2x \\ f(4.01) &\approx f(4) + f'(4)(4.01 - 4) \\ &= 16 + 2 \times 4 \times 0.01 \\ &= 16.08 \end{aligned}$$

Learning (contd.)

- For a fixed datapoint with two features (x_1, x_2), the change in output of the perceptron depends on the corresponding changes in weights w_1 and w_2 and the bias b
- Thus, change in y - Δy is

$$\Delta y \approx \frac{\partial y}{\partial w_1}\Delta w_1 + \frac{\partial y}{\partial w_2}\Delta w_2 + \frac{\partial y}{\partial b}\Delta b$$

- Partial derivative ill-defined in case of perceptron, which creates hurdle for learning

Sigmoid Neurons

- Another simple artificial neuron McCulloch and Pitts (1943)
- Input: one or more real values x_i
- Output: single real value y
- Output computed by applying sigmoid function σ on the weighted sum of inputs and bias

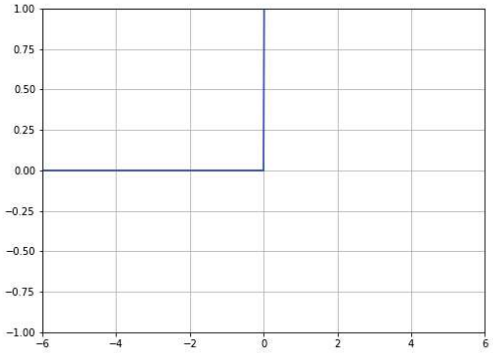
$$y = \sigma(w.x + b) = \frac{1}{1 + e^{-(w.x+b)}}$$

- Decision making using sigmoid:
  - Given real valued output, use threshold
  - If $y > 0.5$, output 1, else 0

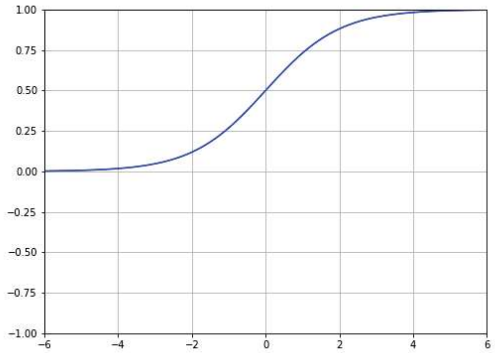
Sigmoid vs. Perceptron

- When $z = w.x + b >> 0$
  - $e^{-z} \approx 0$
  - $\sigma(z) \approx 1$
  - Similar to perceptron producing 1 as output when z is large and positive
- When $z = w.x + b << 0$
  - $e^{-z} \approx \infty$
  - $\sigma(z) \approx 0$
  - Similar to perceptron producing 0 as output when z is large and negative
- Different primarily when absolute value of z is small.

Sigmoid vs. Perceptron (contd.)



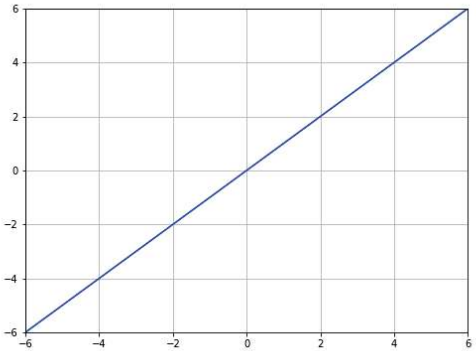
Step (Perceptron)



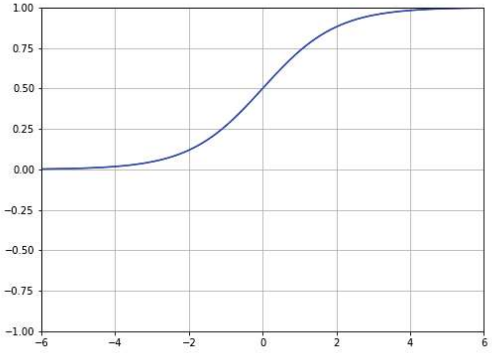
Sigmoid

- Sigmoid is continuous and differentiable over its domain
- Learning is possible via small changes in parameters and using linear approximation

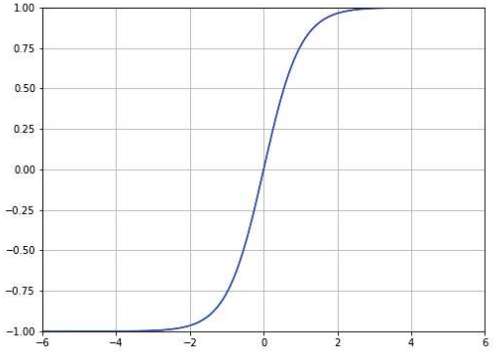
Activation Functions



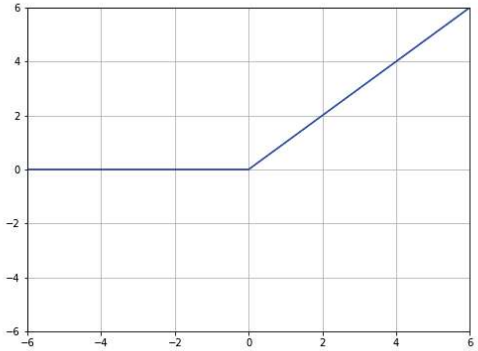
Linear



Sigmoid



Tanh

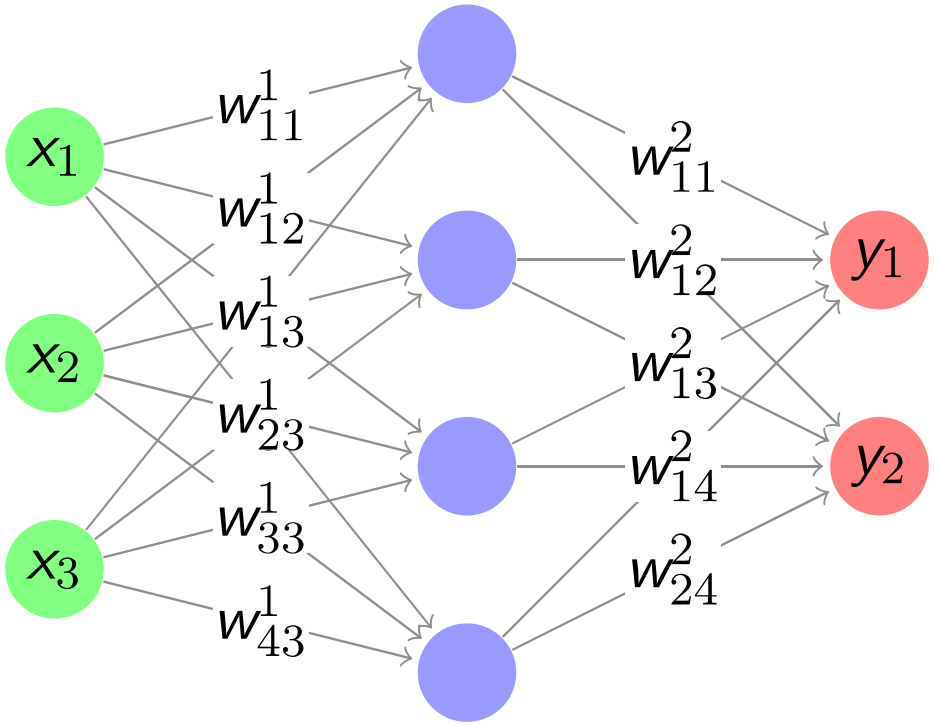


ReLU

Notations

- x: Input
- w_{ij}^{l}: weight from j^{th} neuron in $(l-1)^{th}$ layer to i^{th} neuron in l^{th} layer
- b_{j}^{l}: bias of the $j^{t}h$ neuron in the l^{th} layer
- z_{j}^{l}: $w^{l}.a^{l-1} + b_{j}^{l}$
- a_{j}^{l}: $f(z_{j}^{l})$

Neural Network Architecture



- For MNIST Digit recognition
  - Input Layer $28 \times 28 = 784$ neurons
  - Output Layer?

One-hot output vs. Binary encoded output

- Given 10 digits, we fixed 10 neurons in output layer
  - Why not 4 neurons, and generate binary representation of digits?
- The task is to observe features and learn to decide whether it is a particular digit
- Observing visual features and trying to predict, say, most significant bit, will be hard
  - Almost no correlation there

Feed Forward Computation

- Given input x
  - $z^1 = w^1.x + b^1$
  - $a^1 = \sigma(z^1)$
  - $z^l = w^l.a^{l-1} + b^1$
  - $a^l = \sigma(z^l)$
  - a^L is the output, where L is the last layer
- Note that output contains real numbers (due to σ function)

Loss functions

- Consider a network with parameter setting P1

True			Predicted			Correct
0	1	0	0.3	0.4	0.3	yes
1	0	0	0.1	0.2	0.7	no
0	0	1	0.3	0.3	0.4	yes

- Number of correctly classified examples $= \frac{2}{3}$
- Classification error $= 1 - \frac{2}{3} = \frac{1}{3}$
- Consider same network with parameter setting P2

True			Predicted			Correct
0	1	0	0.1	0.7	0.2	yes
1	0	0	0.3	0.4	0.3	no
0	0	1	0.1	0.2	0.7	yes

- Classification error still the same

Loss functions

- Mean Squared Error : $MSE = \frac{1}{M}\sum(y_i - t_i)^2$

True			Predicted			Correct
0	1	0	0.3	0.4	0.3	yes
1	0	0	0.1	0.2	0.7	no
0	0	1	0.3	0.3	0.4	yes

- Mean Squared Error $= (0.54 + 0.54 + 1.34)/3 = 0.81$

True			Predicted			Correct
0	1	0	0.1	0.7	0.2	yes
1	0	0	0.3	0.4	0.3	no
0	0	1	0.1	0.2	0.7	yes

- Mean Squared Error $= (0.14 + 0.14 + 0.74)/3 = 0.34$
- Indicates that second parameter setting is better

Loss functions

- Mean Cross Entropy
  $MCE = \frac{1}{M}\sum(-t_i log y_i - (1 - t_i) log(1 - y_i))$

True			Predicted			Correct
0	1	0	0.3	0.4	0.3	yes
1	0	0	0.1	0.2	0.7	no
0	0	1	0.3	0.3	0.4	yes

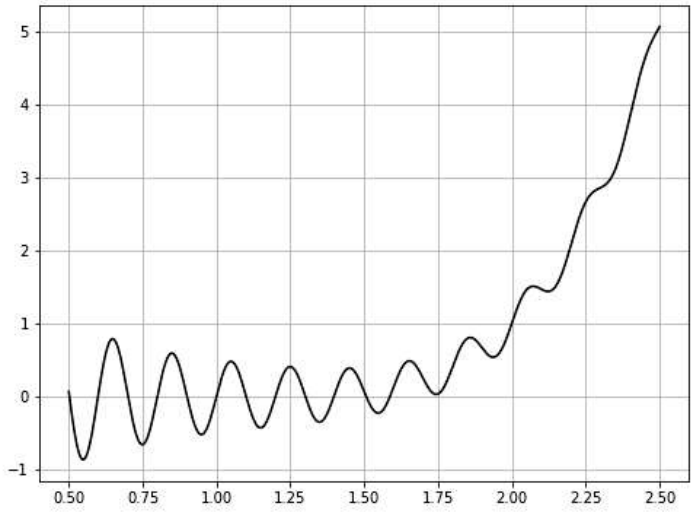
- Mean Cross Entropy $= -(ln(0.4) + ln(0.4) + ln(0.1))/3 = 1.38$

True			Predicted			Correct
0	1	0	0.1	0.7	0.2	yes
1	0	0	0.3	0.4	0.3	no
0	0	1	0.1	0.2	0.7	yes

- Mean Cross Entropy $= -(ln(0.7) + ln(0.7) + ln(0.3))/3 = 0.64$
- Indicates that second parameter setting is better

Minimizing Loss

- Consider a function C that depends on some parameter x as shown below:



- How to find the value of x for which C is minimum?
- Idea: Choose a random value for x, place an imaginary ball there. It will eventually lead to a valley

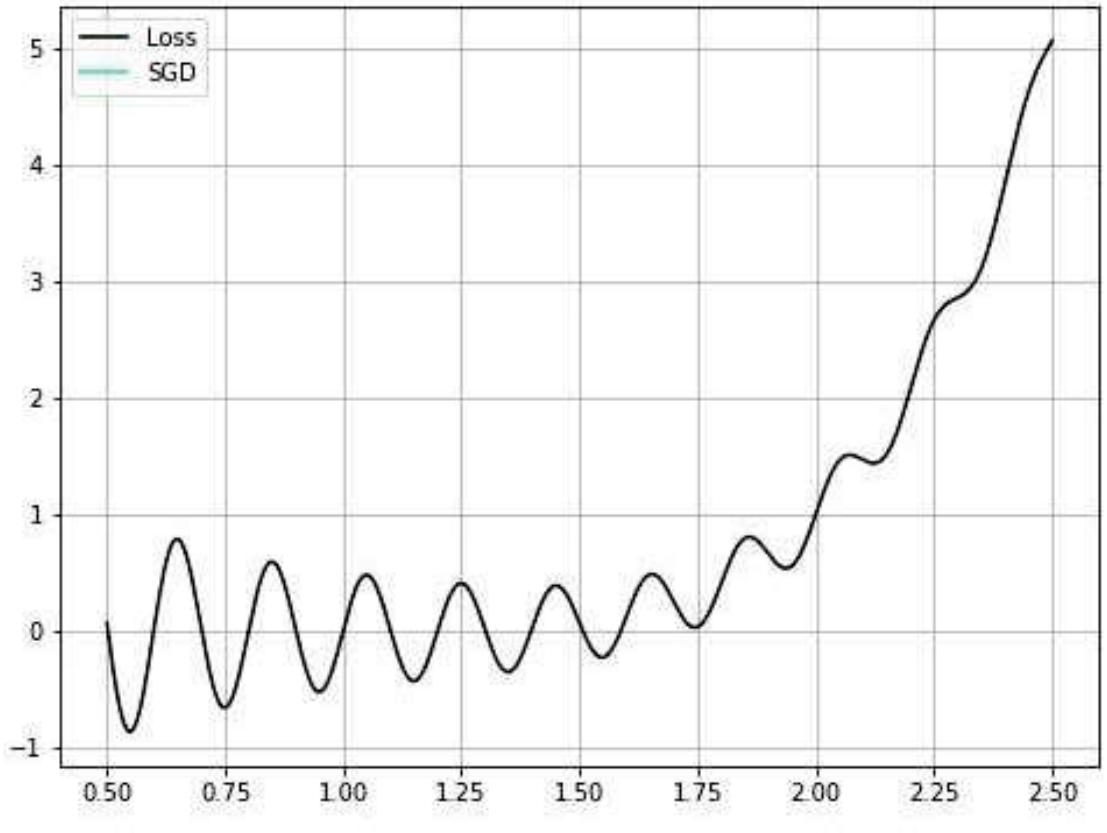
Gradient Descent

- Recall $\Delta C \approx \frac{dC}{dx}.\Delta x$
- We want to change x such that C is reduced *i.e.* ΔC has to be always negative
- What if we choose $\Delta x = -\eta \frac{dC}{dx}$?

$$\begin{aligned}\Delta C &\approx \frac{dC}{dx}.\Delta x \\ &= \frac{dC}{dx}.(-\eta \frac{dC}{dx}) \\ &= -\eta.(\frac{dC}{dx})^2 \\ &\leq 0\end{aligned}$$

- Gradient Descent: $x_{t+1} = x_t - \eta \frac{dC}{dx}$

Gradient Descent



Back Propagation

- Effective use shown in Williams et al. (1986)
- Every neuron taking part in the decision
- Every neuron shares the blame for error
- Decision made by a neuron dependent on its weights and biases
- Thus error is caused due to these weights and biases
- Need to change weights and biases such that overall error is reduced
  - Can use gradient descent here
  - For a weight w_{ij}^k, the weight update will be

$$w_{ij}^k \leftarrow w_{ij}^k - \eta \frac{\partial C}{\partial w_{ij}^k}$$

Back Propagation (contd.)

Will use calculus chain rule to obtain the partial derivatives

1. First find error for each neuron on the last layer

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

2. Then find error for each neuron on the interior neurons

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

3. Update weights and biases using following gradients:

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

Vanishing Gradient Problem

- Observed by Hochreiter (1998)
- Important point: the $\sigma'(z^l)$ term in the previous steps
- Derivative of the activation function
- Will be multiplied at each layer during back propagation
- Example: 3 layer network

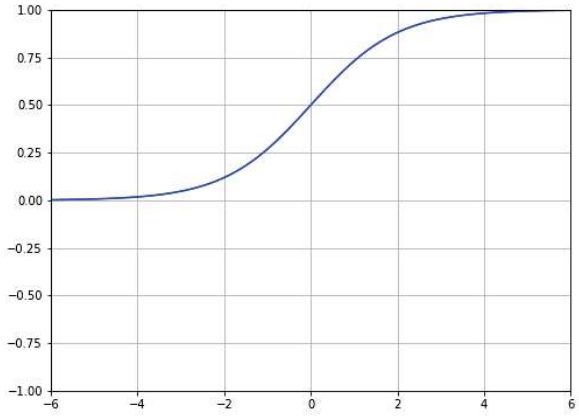
$$\delta^4 = A.\sigma'()$$

$$\delta^3 = X.\delta^4.\sigma'() = X.A.\sigma'().\sigma'()$$

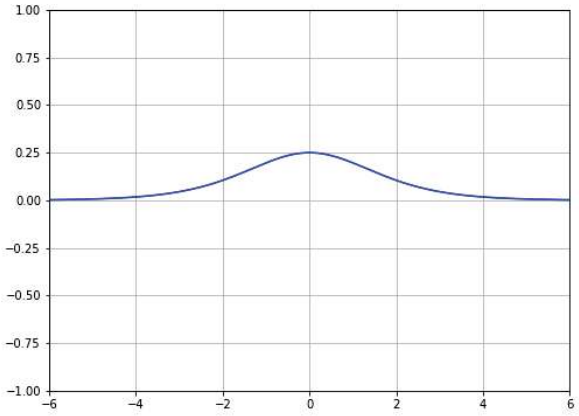
$$\delta^2 = Y.\delta^3.\sigma'() = Y.X.A.\sigma'().\sigma'().\sigma'()$$

$$\delta^1 = Z.\delta^2.\sigma'() = Z.Y.X.A.\sigma'().\sigma'().\sigma'().\sigma'()$$

Vanishing Gradient Problem (contd.)



Sigmoid



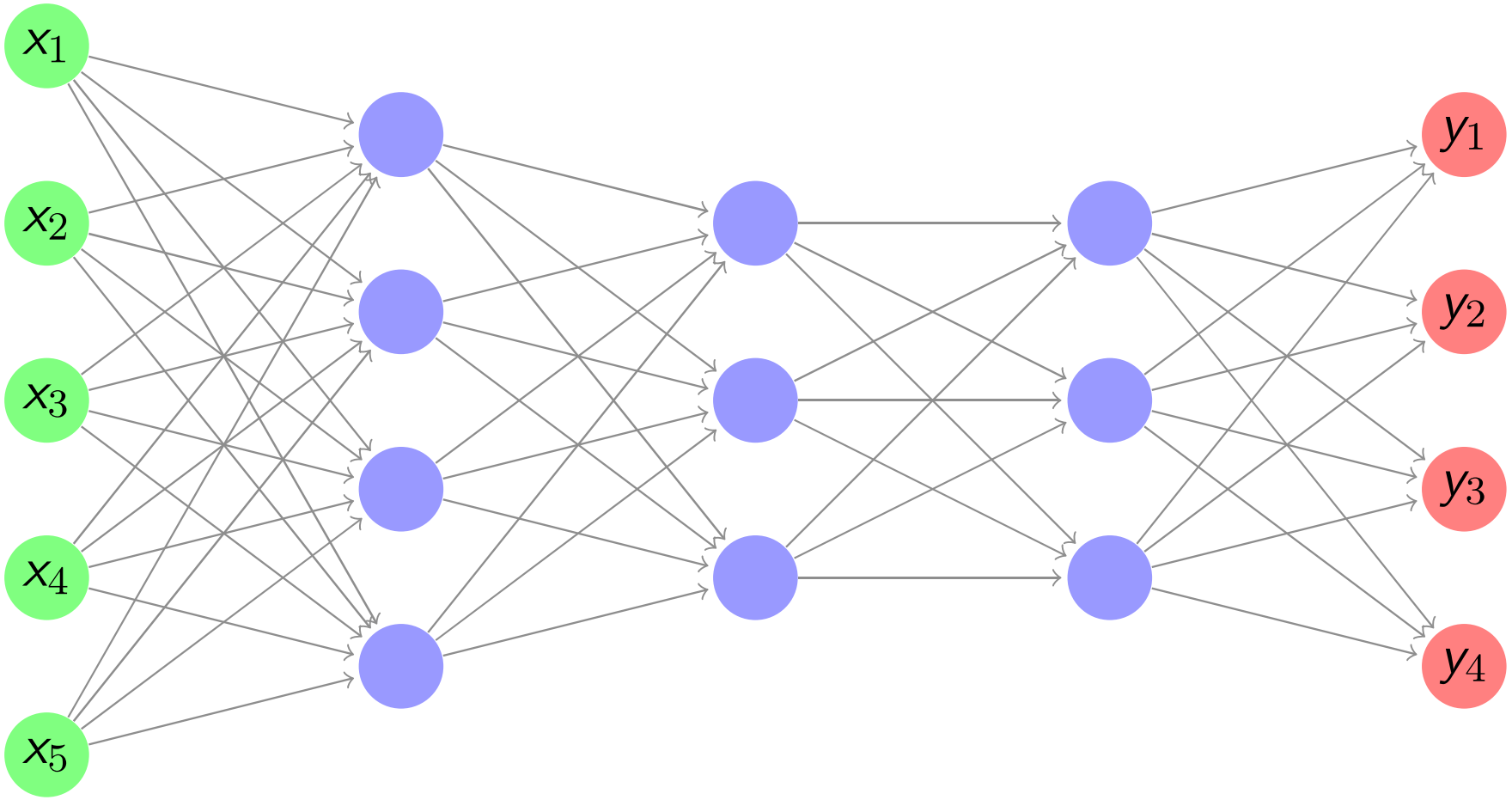
Derivative of Sigmoid

- Maximum value of sigmoid's derivative = 0.25
- $0.25^n \approx 0$ as $n \to \infty$
- Gradient tends to 0 *i.e. vanishes*

Deep Learning

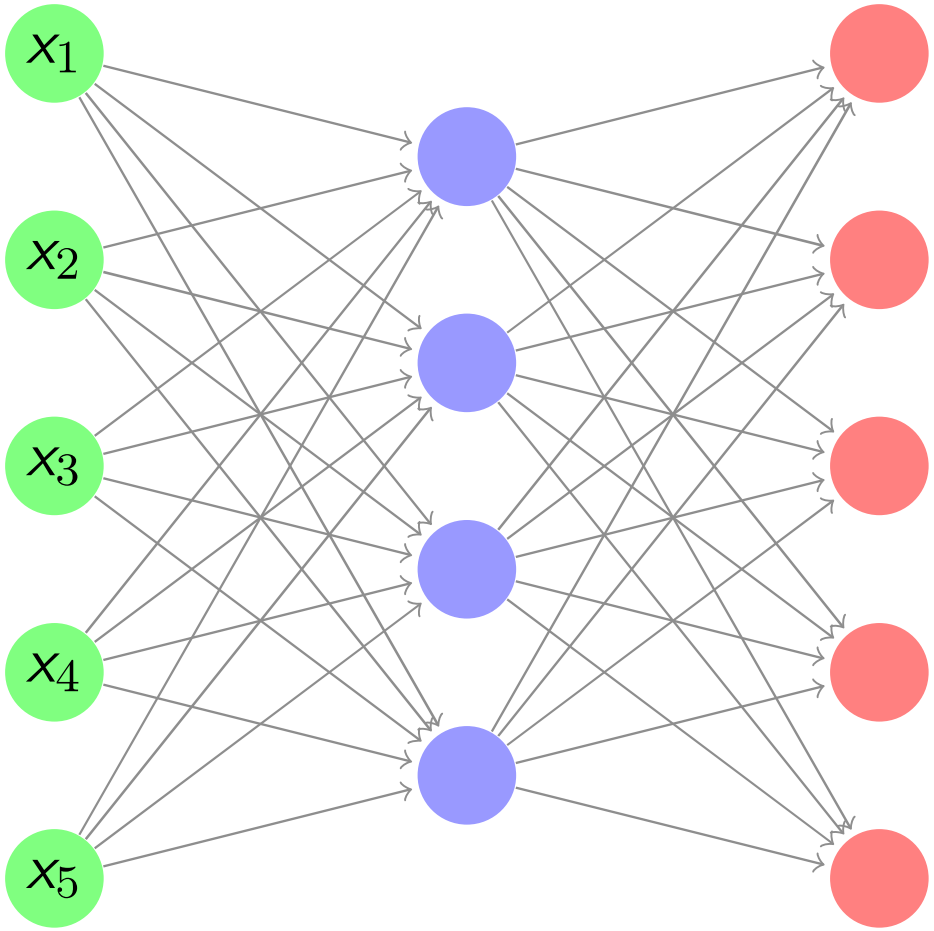
- Set of techniques and architectures that tackles such learning problems and helps to reach optimal parameters faster
- Various methods:
  - Start at near optimal values of parameters so smaller updates due to vanishing gradients is not much of a problem
  - Use better activation functions which can avoid such problems
  - Use better optimizers than standard gradient descent
  - Use novel architectures

Tackling Vanishing Gradients via Greedy Unsupervised Pretraining



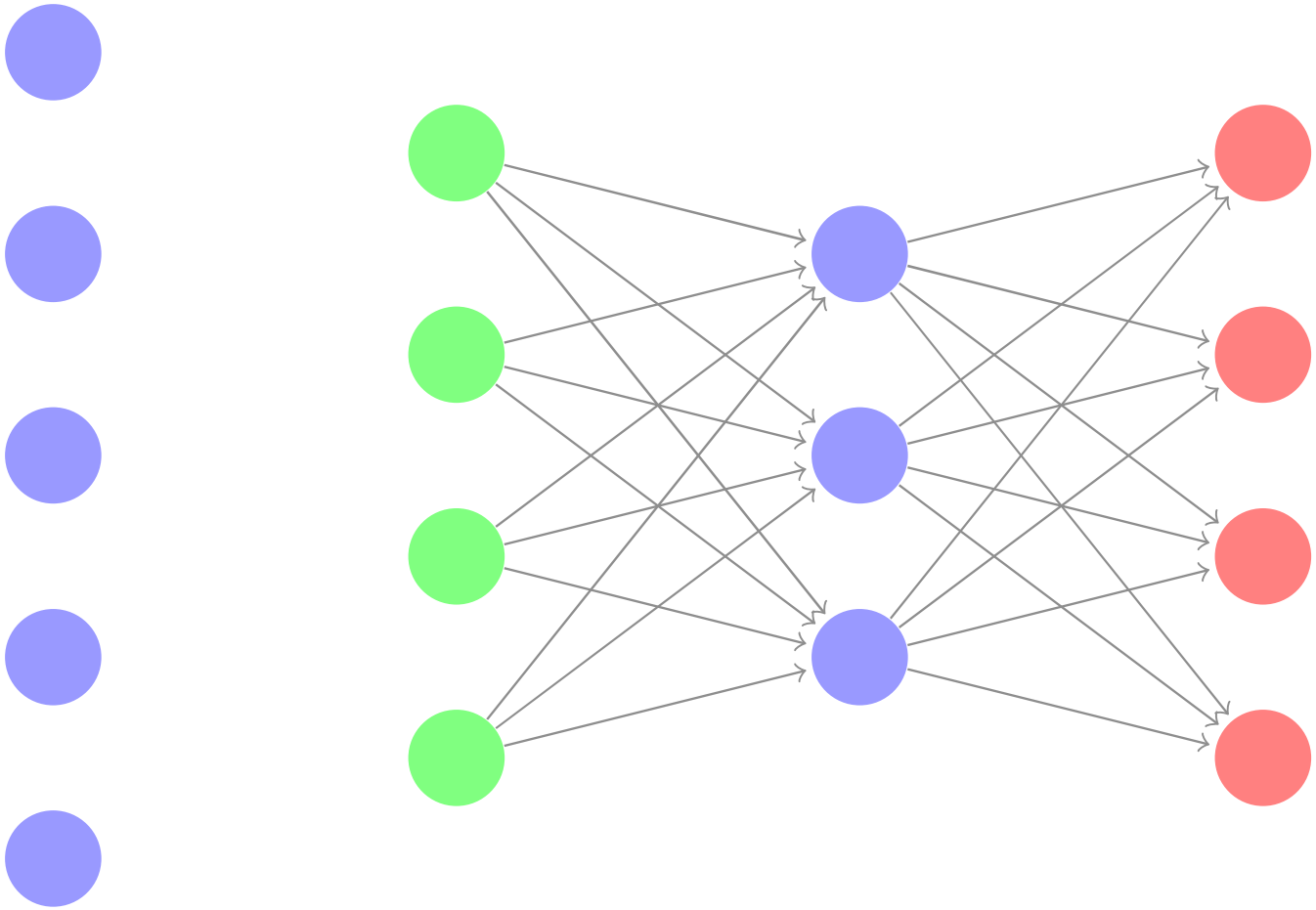
- Proposed by Bengio et al. (2007)

Tackling Vanishing Gradients via Greedy Unsupervised Pretraining



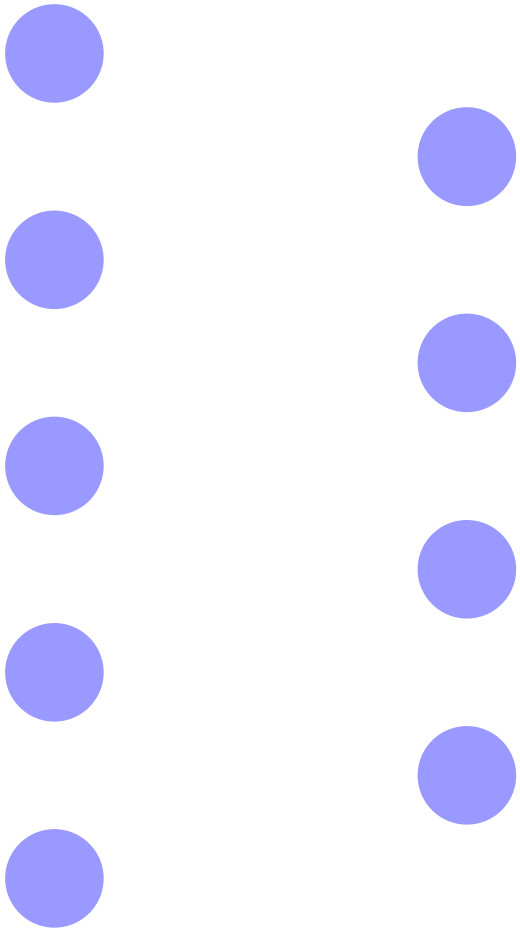
- Proposed by Bengio et al. (2007)

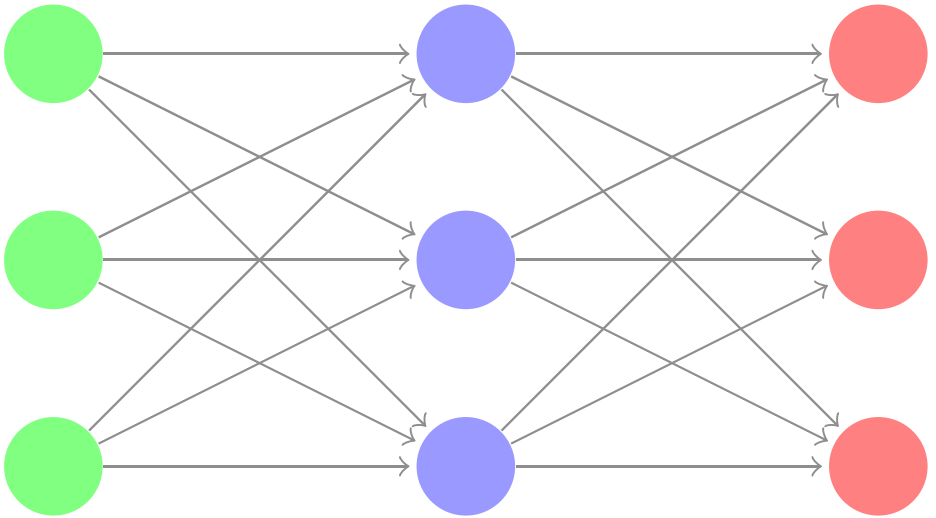
Tackling Vanishing Gradients via Greedy Unsupervised Pretraining



- Proposed by Bengio et al. (2007)

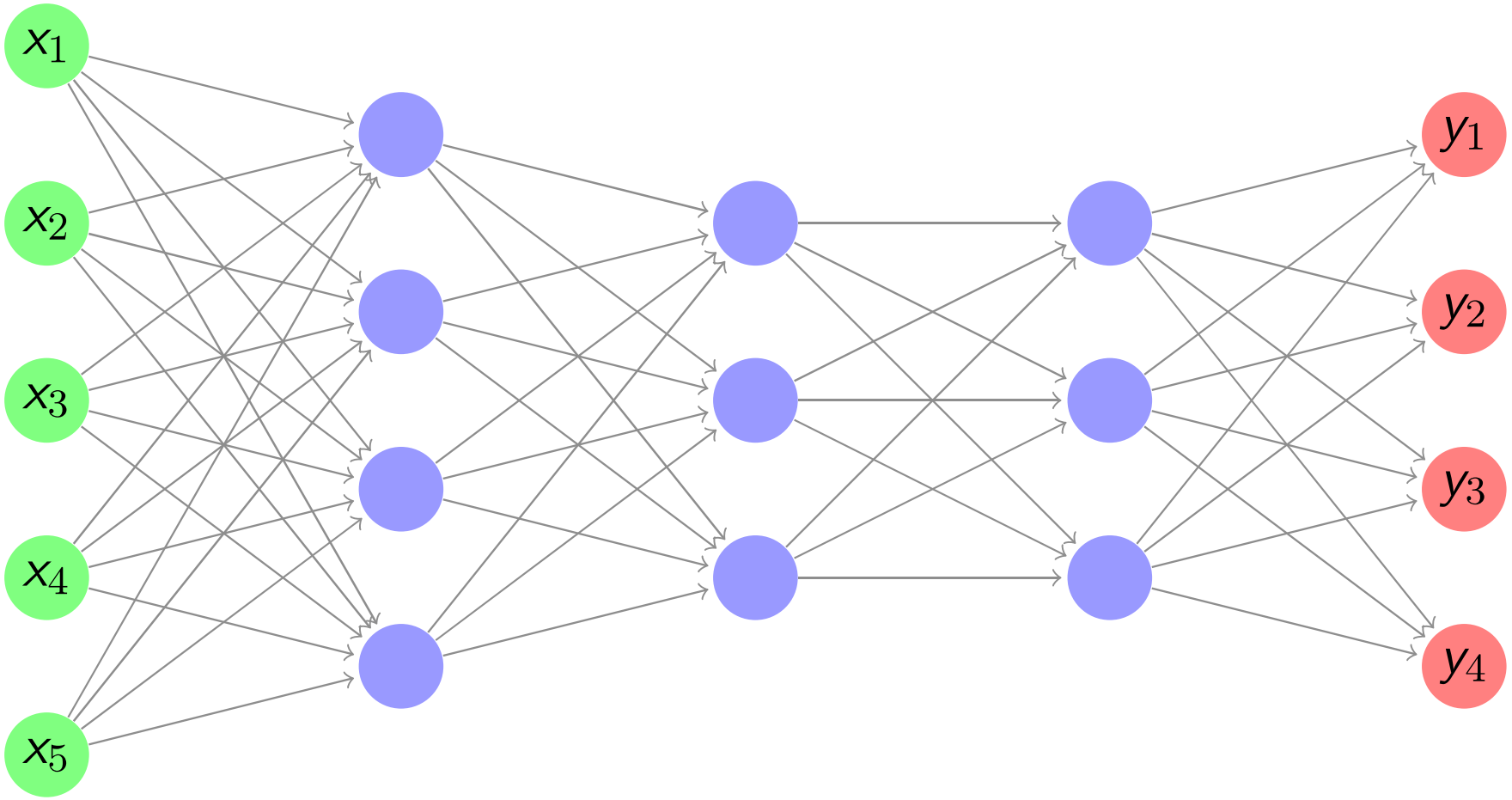
Tackling Vanishing Gradients via Greedy Unsupervised Pretraining





- Proposed by Bengio et al. (2007)

Tackling Vanishing Gradients via Greedy Unsupervised Pretraining



- Proposed by Bengio et al. (2007)

Tackling Vanishing Gradients via Novel Activation Functions

- Rectified Linear Unit Nair and Hinton (2010): Derivative = 1 when non-zero, else 0
- Product of derivatives does not vanish
- But once a ReLU gets to 0, it is difficult to get it to one again (Dead ReLU problem)
  - Addressed by better variants such as Leaky ReLU Maas et al. (2013), Parametric ReLU He et al. (2015) *etc.*

Types of Gradient Descent

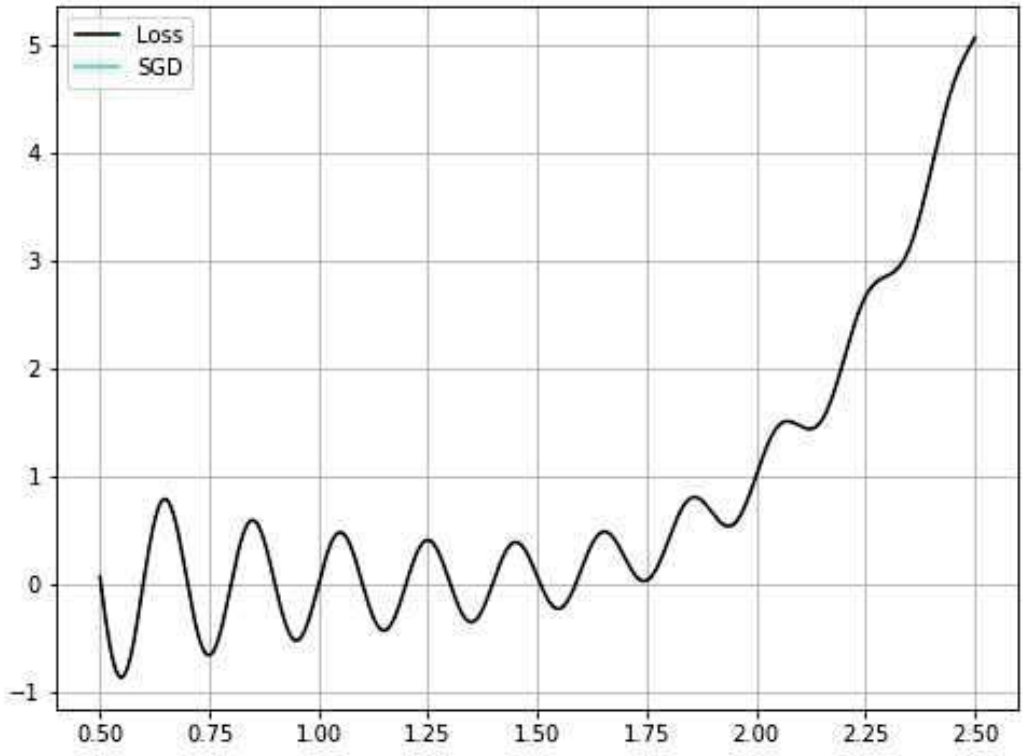
Based on the amount of data used for training

- Batch GD: all training data per update, slow, not applicable in online setting, but guaranteed to converge to global minimum for convex and local minimum for non-convex
- Stochastic GD: one training datapoint per update, fluctuates a lot, allows to jump to new and potentially better local minima, this complicates convergence, has been shown that by decreasing learning rate almost certainly converges to global in convex and local in non-convex
- Mini-batch GD: batch of n datapoints per update, best of both worlds - relatively stable convergence and can use matrix operations for batches

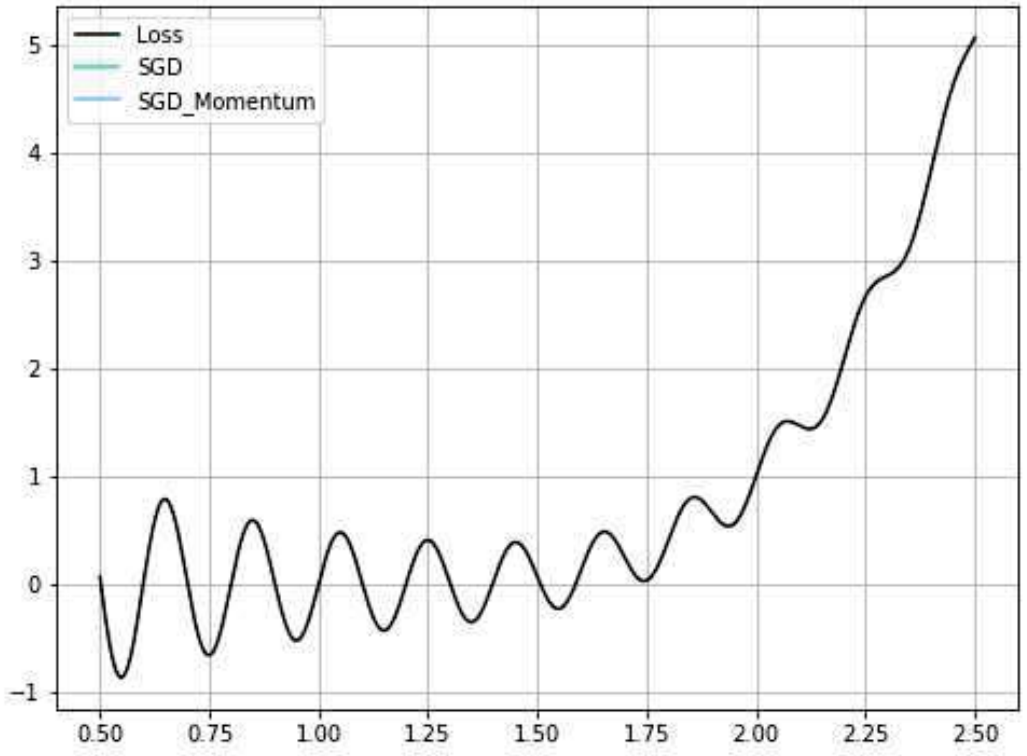
Tackling Learning Difficulties via Optimizers

- SGD mainly used for a long time
- Converges slowly
- Can get stuck in local minima

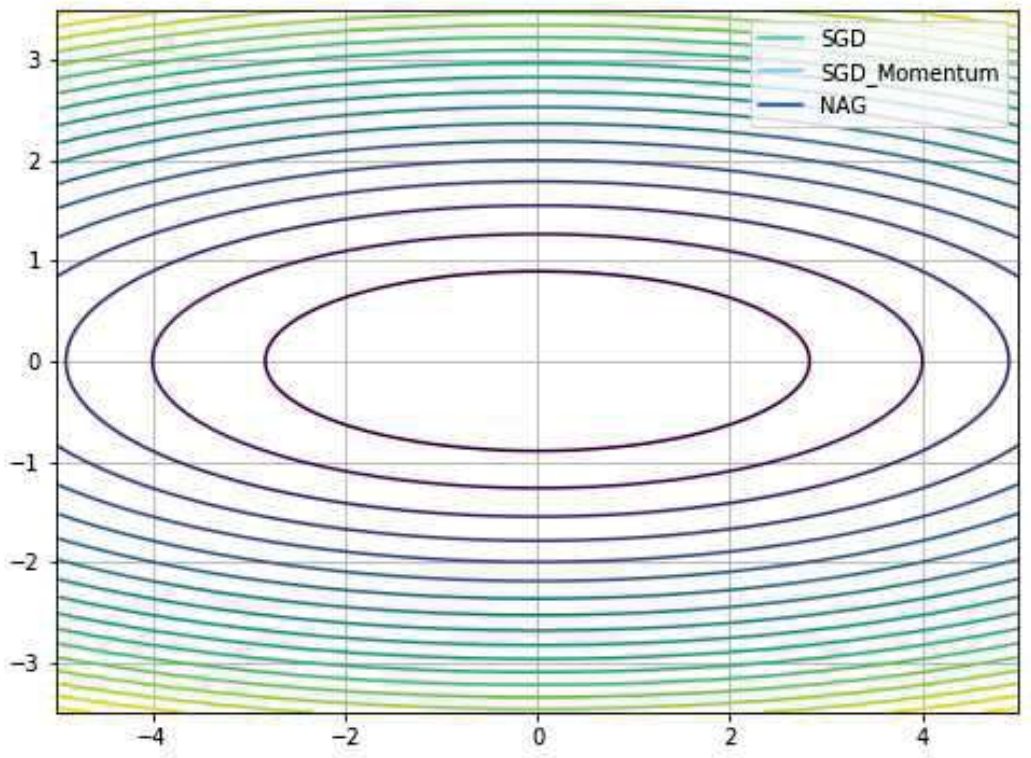
SGD



SGD + Momentum



Nesterov Accelerated Gradient



- Developed by Nesterov (1983)

Other Optimizers

- AdaGrad: Adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters Duchi et al. (2011)
- AdaDelta: Does not need an initial learning rate Zeiler (2012)
- RMSProp: Good with recurrent networks; Unpublished method from Hinton's Coursera Lectures
- Adam: Benefits of RMSProp and AdaDelta mixed with momemtum tricks Kingma and Ba (2014)

Tackling Vanishing Gradients via Novel Architectures

- Novel architectures made to specific problems
- Example:
  - Derivative of activation function in LSTM is identity function is 1. Gradient does not vanish
  - Effective weight depends on forget gate activation, whose derivative is never > 1.0. So Gradient does not explode
- Will be covered in other sessions

Conclusion

- Exciting area of research
- Heavy involvement from industry
- Many developments in each of those subareas: Activation functions, Optimizers, Architectures, *etc.*

References I

Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007). Greedy layer-wise training of deep networks. In *Advances in neural information processing systems*, pages 153–160.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034.

Hochreiter, S. (1998). Recurrent neural net learning and vanishing gradient. *International Journal Of Uncertainity, Fuzziness and Knowledge-Based Systems*, 6(2):107–116.

References II

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814.

References III

Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o (1/ƙ 2). In *Doklady AN USSR*, volume 269, pages 543–547.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Williams, R. J., Rumelhart, D. E., and Hinton, G. E. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533–538.

Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

Thank You