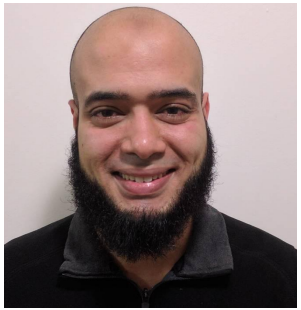
Machine Learning

Training Neural Networks: The Big Picture

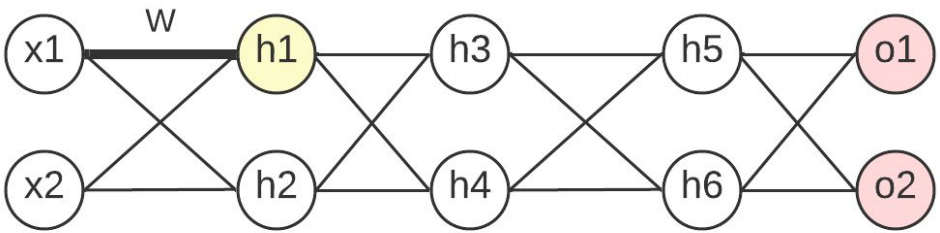
Mostafa S. Ibrahim
Teaching, Training and Coaching for more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / MSc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)

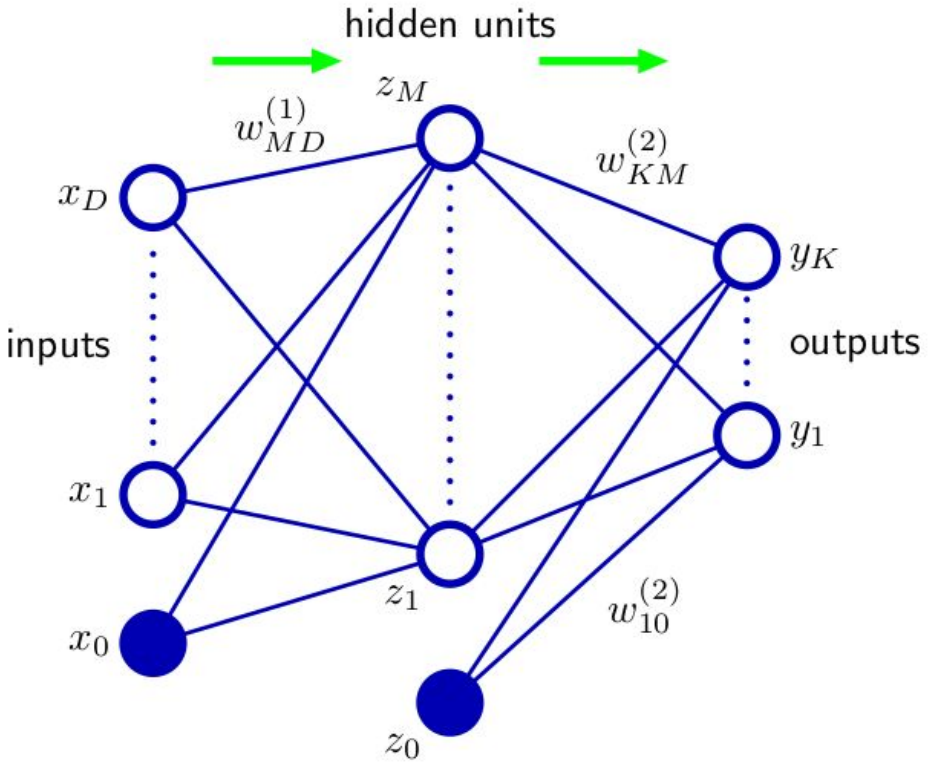


Training Neural Networks with Gradient Descent

- The algorithm **flow** for training an NN is **exactly** the same as optimizing **any** function, such as how we did it with linear regression
- While not stopping the training:
  - Feedforward example/mini-batch into your network
  - Get the outputs
  - For each weight w, update it using the formula:
    - w -= learning rate * $\partial E / \partial w$
- The issue now: the error function is **complex**!



Mathematical Notation



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

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \qquad z_j = h(a_j).$$

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \qquad y_k = \sigma(a_k)$$

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \right)$$

Src: Bishop book

Computing Partial Derivatives

- When training a neural network, we need to compute the partial derivative of the error/cost/loss function with respect to each weight w

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

- One way to do this is **numerically**, which is slow and impractical
  - Let J be our function and theta is one of the weights and c is a small value e.g. 1e-5

$$\frac{\partial}{\partial \theta_i} J(\boldsymbol{\theta}) \approx \frac{J(\boldsymbol{\theta}_{i+c}) - J(\boldsymbol{\theta}_{i-c})}{2c}$$

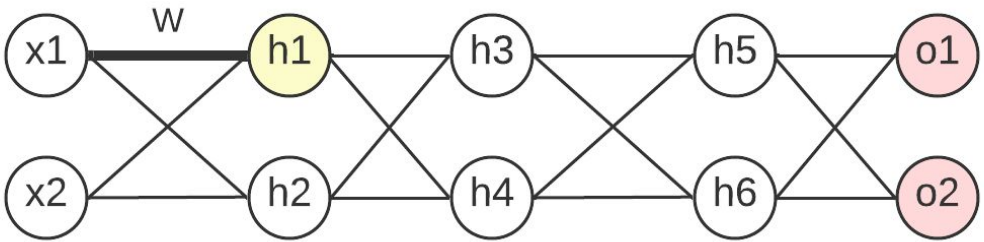
Backpropagation Algorithm

- Backpropagation algorithm is an **efficient** method for **computing gradients** in directed graphs of computations, such as neural networks.
- This is not a **learning method**, but rather a nice computational trick
- The trick is simple. As we process a DAG, we can build the results bottom-up using dynamic programming, which **caches** subresults
  - Interestingly, many lecturers do not emphasize the DP point, instead choosing to explain the caching idea directly
    - Its cashing is very common in competitive programming!

Prerequisites

- To properly understand the details one needs:
  - Understanding bottom-up **dynamic programming** on DAG
    - You can understand the caching anyway as it is logical
    - Caching is the key property of backpropagation algorithm
  - **Multivariate** chain rule

Computing the derivative: the hard way!

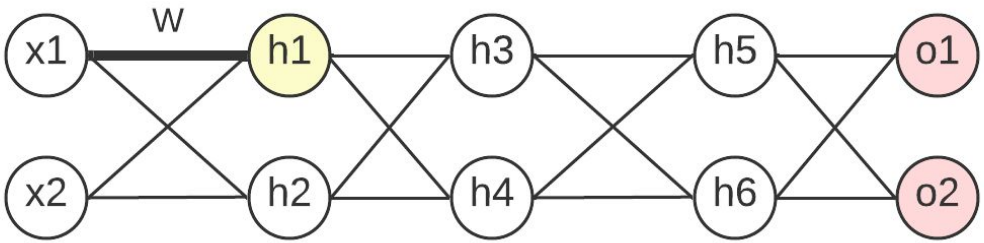


$$E = E1 + E2$$
$$E1 = (o1 - t1)^2$$
$$E2 = (o2 - t2)^2$$

Compute $\partial E / \partial w$

- To compute this partial derivatives, we need to consider all paths from + edge w to the output nodes (hence node(h2) and all its connections are discarded)
- We can use the chain rule to simplify this process like so:
  - e.g. one path is: o1 ⇒ h5 ⇒ h4 ⇒ h1 ⇒ w
- But we will face 3 issues!
  - Tedious math even for a simple network: there are **8 paths** from h1 to an output nodes!
    - Hence, O(weights) to update a single weight ⇒ O(weights^2) for the whole network
  - Computations **duplications**!

Caching Trick

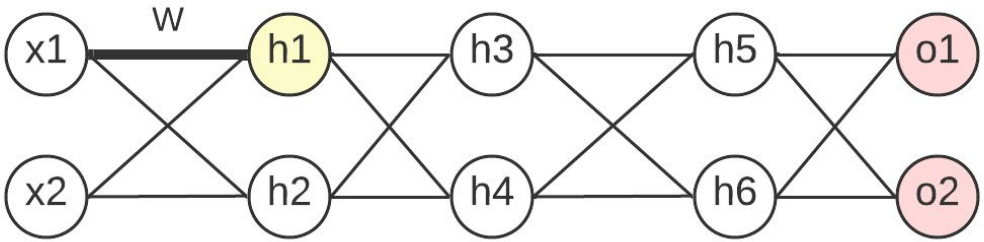


E = E1 + E2
E1 = (o1 - t1)^2
E2 = (o2 - t2)^2

Compute $\partial E / \partial w$

- Given the graph is DAG, this means the paths of node(h1) and h(4) are part of the paths of node(h1). Hence, we are **recomputing** results heavily!
- This is where backpropagation comes **with just caching** results
- The general rule is computing the partial derivative of the error function relative to any node/edge by preparing the results of its dependencies
    - We can/**should** implement that in a bottom-up fashion

Node Derivatives

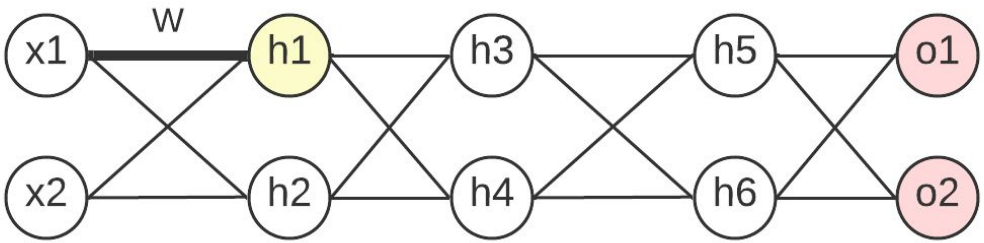


E = E1 + E2
E1 = (o1 - t1)^2
E2 = (o2 - t2)^2

Compute $\partial E / \partial w$

- Compute partial derivatives for output layer: o1 and o2
  - Directly based on the error function
- Compute partial derivatives for hidden layer of h5 and h6
  - h5 uses cached results from o1 and o2. Same for h6
- Compute partial derivatives for hidden layer of h3 and h4
  - h3 uses cached results from h5 and h6. Same for h3
- Compute partial derivatives for hidden layer of h1 and h2
  - h1 uses cached results from h3 and h4. Same for h2
- Now, we computed partial derivatives for **all nodes**

Weights derivative

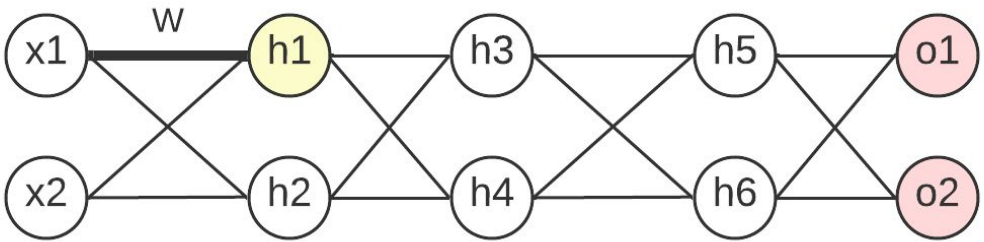


E = E1 + E2
E1 = (o1 - t1)^2
E2 = (o2 - t2)^2

Compute $\partial E / \partial w$

- Start from backward again and update **each weight**
- Let W4 represents all the weights of the last layer (from o1/2 to h5/6)
  - Let Wij denote the edge from node (i) at some layer L to node(j) in layer L-1
- Update W4[o1][h5], W4[o1][h6], W4[o2][h5], W4[o2][h6]
- Update W3[h5][h3], W4[h5][h4], W3[h6][h3], W4[h6][h4]
- The same for W2 and W1
- Overall **4 matrices** represent this network

Weights derivative

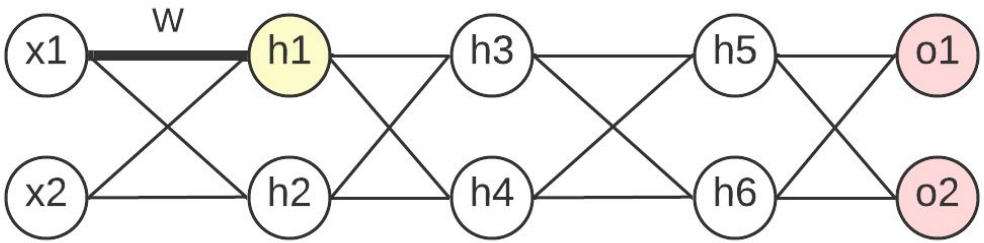


E = E1 + E2
E1 = (o1 - t1)^2
E2 = (o2 - t2)^2

Compute **∂E / ∂w**

- To update a single weight, e.g. let w = W1[h1][x1]
- Compute derv = **∂E / ∂w**
  - Just use the cached results of h1, which were based on h3 and h4 up to the outputs
- Update the weight
  - w = w - learning_rate x derv
- This means we need O(weights) to update the whole network!

Overall Procedure



E = E1 + E2
E1 = (o1 - t1)^2
E2 = (o2 - t2)^2

Compute $\partial E / \partial w$

- Overall
  - Compute and store partial derivatives for every node/layer
  - Compute the partial derivative of every weight and update it based on the current weights
- Common **implementation mistake**
  - A common implementation error is to mix these two steps, which can lead to incorrect results
  - Any weight update must be based on the current weights NOT the updated weights
  - It is safer to perform the 2 steps separately (full compute, then update)

Why learn the details of backpropagation?

- Backpropagation is a challenging topic to learn and implement
- Students may wonder why they should bother with the low-level details when modern frameworks like Tensorflow, Pytorch, and Keras make it easy to build and train large neural networks
- However, mastering the details of backpropagation is crucial for several reasons!
- Because you will find it hard to **understand** some of the Deep Learning issues (vanishing/Exploding gradients, Dying ReLUs).
- It will be hard to **build or debug** complex deep networks. It will be hard to **make sense** about how some networks really work!
- You may face NANs due to gradient problems

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."