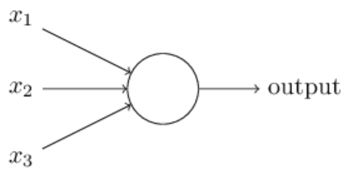
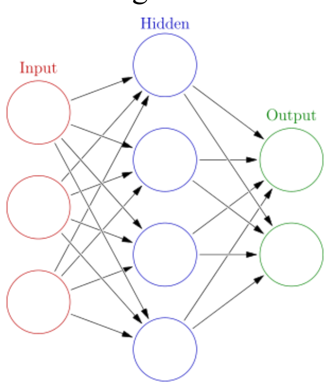
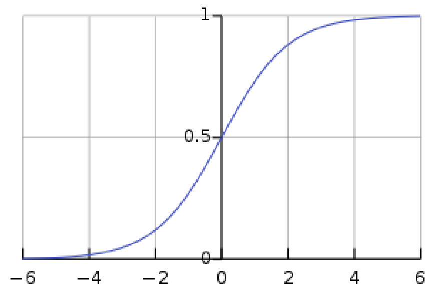
Backpropagation in Feed Forward Artificial Neural Networks

- Perceptron model
  - Takes several binary inputs and produces single binary output
  - 

  - Ordered set of weights *w*
  - Output determined if $\sum_j w_j x_j = \vec{w} \cdot \vec{x} > b$ greater than threshold value
    - 1 if W•x + b > 0 where b is bias term (threshold moved to lhs)
      - Bias is how easy it is for perceptron to fire or activate
      - Positive bias means easier to fire, negative bias means harder to fire
  - Network of perceptrons allow for subtle decision making
- Neural Network
  - Set of Nodes: Input, Hidden, Output
  - Set of weights between nodes
  - 

  - “Learns” by feeding training data through network and then correcting weights appropriately by working back through the network => Backpropagation
    - Feed forward → no loops, loops in recurrent neural networks that have certain neurons that recurrently fire for some duration and slowly die out, lots of use in natural language processing where context is relevant
  - Perceptron model uses binary output, so crossing a threshold might change that perceptron for the better but it may drastically change the rest of the network in unwanted ways
- Sigmoid Neuron
  - Continuous inputs and continuous output

- Output is σ(w · x+b) where σ is the sigmoid or logistic function
    - "Activation function"
    - $\sigma(z) = \dfrac{1}{1+e^{-z}}$
    - 

- $$\Delta\text{output} = \sum \frac{\partial\,\text{output}}{\partial w_j}\Delta w_j + \frac{\partial\,\text{output}}{\partial b}\Delta b$$

- How to learn
    - Backpropagation
    - Adjust weights to optimize output ⇒ Need to define a loss/cost/objective function
    - How to minimize cost function
        - Why not just do some FOCs and SOCs? C is a function of many, many variables; doing FOCs and SOCs for single variable is fine, doing it for 2 or 3 is harder, doing it for more… fuhgettaboutit
- Gradient Descent
    - Imagine you're on a hilly terrain and you want to get down to the bottom before sun down, but it's super foggy so you can't see far ahead. How do you do so?
    - Move in random directions
        - This is bad because you might end up moving up the hill!
    - Randomly place your foot until it is downward from your first foot
        - Better, but still not great
    - If only you could figure out which direction to move in… you want to move in direction of quickest descent (away from direction of quickest ascent)... gradient!
    - $\Delta C = \dfrac{\partial C}{\partial v_1}\Delta v_1 + \dfrac{\partial C}{\partial v_2}\Delta v_2$ ... ⇒ $\Delta C = \nabla C \cdot \Delta v$
    - $\Delta v = -\alpha \nabla C$
        - α is hyperparameter learning rate
    - $\Delta C = -\alpha \nabla C \cdot \nabla C = -\alpha ||\nabla C||^2$
        - Since gradient is squared, the change in the cost function will always be negative (down)
    - Update: $v' = v - \alpha \nabla C$

        - Note on selecting a learning rate: Too small and it takes forever, too large and you might overshoot
    - In a ANN, you want to update your weights and biases to minimize C:
        -
        -
    - Question: How do you compute the gradient in practice?
        - Compute partial for each variable and take average ⇒ long and computationally expensive
        - Solution: Stochastic gradient descent
- Stochastic gradient descent
    - Randomly select some variables and compute partials with respect to those variables and average
    - Set of selected variables called mini-batch of size *m*
    - (This is based on an assumption about C that we will discuss in a second when we define C)
        -
        -
    - One iteration of learning on training inputs called "Epoch"
        - Number of epochs used in training, mini-batch size, and learning rate are all hyperparameters which can be optimized separately
- Backpropagation
    - Notation:
        - is the activation of the neuron in the layer
        - is the weight from the neuron in the layer to the neuron in the layer
        - Activation of a neuron is related to the sum of its inputs, which is related to the activations of the neurons in the previous layer
            -

- is a weight matrix containing weights, is a vector containing biases

  ⇒

- Weighted input vector

- Cost function
  - Could define function like number of correct responses, but you want a continuous function so that you can make small adjustments to your weights
  - Mean Squared Error (Quadratic Cost Function)
    - Extra ½ doesn't affect the optimization but simplifies computation later on
  -
    - This holds true for the MSE loss function, but this is not true for all the loss functions you encounter in the wild
    - Note: only a function of since y is a fixed parameter associated with training data
- Objective: Understand how changing the weights (and biases) in a network affect the cost function
- is the error term in neuron in the layer, so is vector with error terms
  - , then the effect on the cost function would be
- **Error in the output layer:**
  - By definition of the error term, the output error term can be found as
  - Summing over all output neurons,

- Remember that is a function of if , so if , the partials term

  is zero. Therefore,
  - This makes sense as the partial with respect to the activation is how much a change in the activation affects the cost. The sigmoid derivative shows how fast the activation function is changing at that point (the weighted input of the output activation)
  - Note: These components are fairly simple to compute
    -
    -
    - Side note:

      where A(x) is a matrix who has x along its diagonal and zeroes everywhere else
- **Error in a layer in terms of error in the subsequent layer:**
  - By definition,
  - Expanding,
  - Given , the error term in *l* can be rewritten as
  - By definition, , so differentiating yields
  - Back into the original equation shows that
- **Partial of Cost with respect to weights**

- ~~Definition:~~
- ~~Applying the chain rule:~~
- ~~Simplifying:~~
- ~~Definition:~~
-
- ~~Substituting back into the initial equation:~~
- ~~Given the definition of~~
- ~~Remember that only one of the activations in a layer is affected by~~ ~~, so~~

  ~~it can be rewritten as~~ ~~z^L_j~~
-

- **Partial of Cost with respect to biases**
  -

- Application of fundamental equations to backpropagation (using stochastic gradient descent)
  - 1. Input x (compute for input layer)
  - 2. Feedforward (For each subsequent layer *l* for 2, 3... through L, computing and)
  - 3. Output error (compute)
  - 4. Backpropagate Error (for every *l* before L, compute)
  - 5. Output (gradient of cost is and).
  - 6. Update weights and biases moving along gradient of cost

        -

        -

- Modern uses
    - “Deep learning” ⇒ ANNs with many hidden layers, often 10-15
    - “Convolutional Neural Networks”
        - Break problem further into sub-problems, thus “convoluting” within hidden layers
        - Super useful for computer vision, image compression, stock market prediction, even really good estimation of travelling salesman’s problem
    - GPU speed up allows parallel training of the network
    - We used sigmoid activation function, which is nice because it makes the math (relatively) simple and easy to understand. Often not recommended because it saturates neurons very quickly (goes to either 1 or 0) which makes it like a perceptron
        - Solved with variety of methods, including varied activation functions
        - Rectified Linear Unit (ReLU) accelerates SGD by factor of 6 compared to sigmoid or tanh activation functions because it saturates less and uses less computationally expensive operations
        - ReLUs can die, making large parts of the network inactive
            - Solved with Leaky ReLUs

Sources accessed:

- Stanford CS231n (http://cs231n.stanford.edu/)
- Backpropagation derivation (http://neuralnetworksanddeeplearning.com/chap2.html)