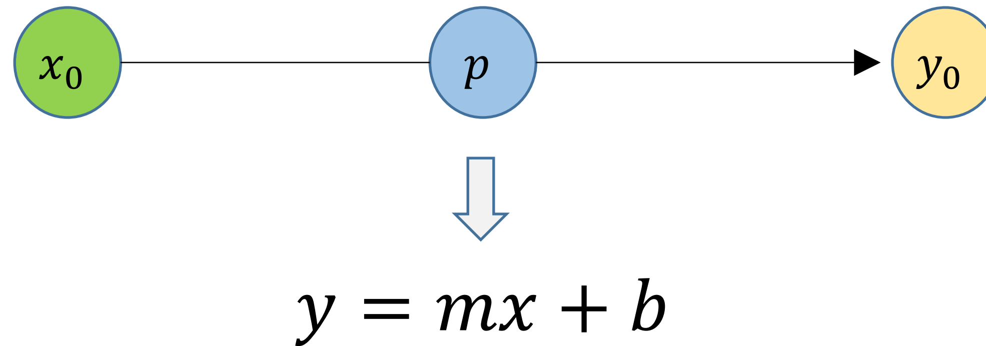
CS535
Deep Visual Computing:

Deep Basics

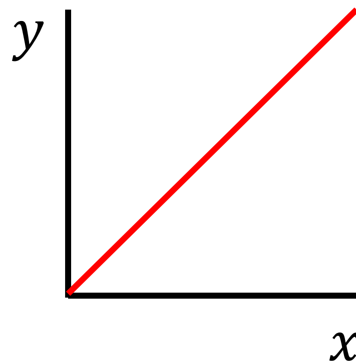
Daniel G. Aliaga

Perceptron

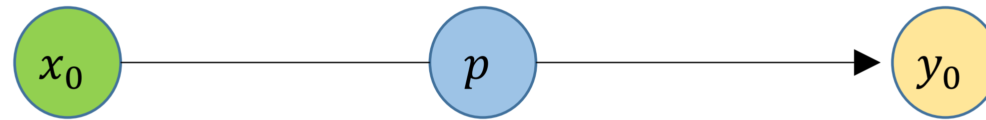


$$y = mx + b$$

Example: $b = 0, m = 1$ → $y = x$



Perceptron

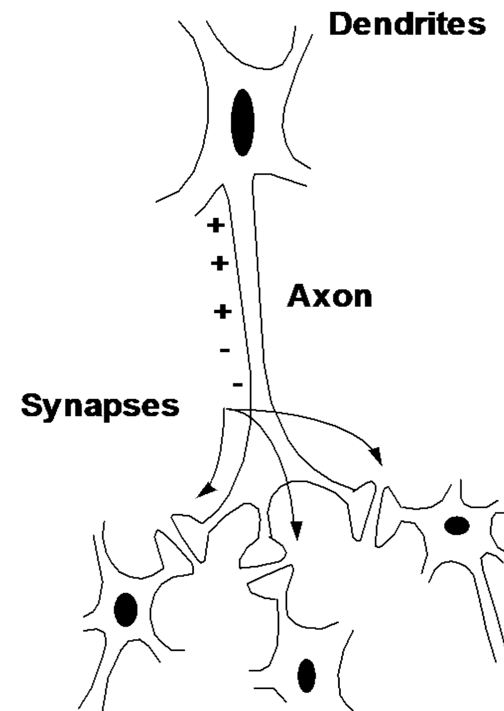


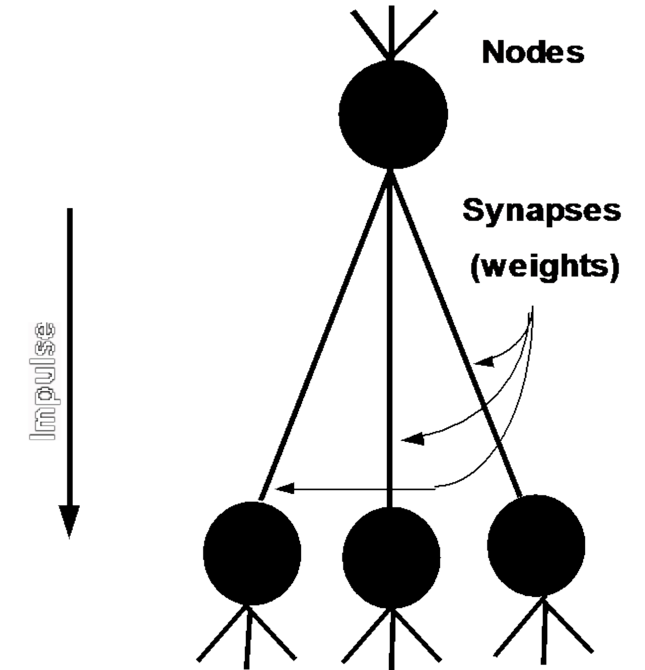
$$y = mx + b$$

Activation Function

Biology 101

- In human brain:
  - Neuron switching time
    ~ 0.001 second
  - Number of neurons
    ~ 10^{10}
  - Connections per neuron
    ~ 10^{4-5}
  - Scene recognition time
    ~ 0.1 second
  - Huge amount of parallel computation
    → 100 inference steps is not enough

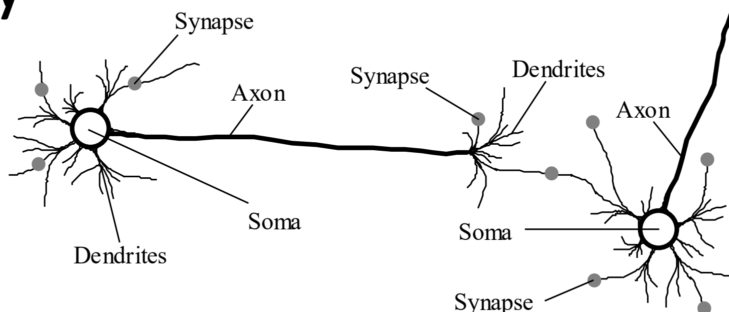


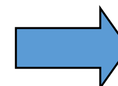


© Eric Xing @ CMU, 2006-2011

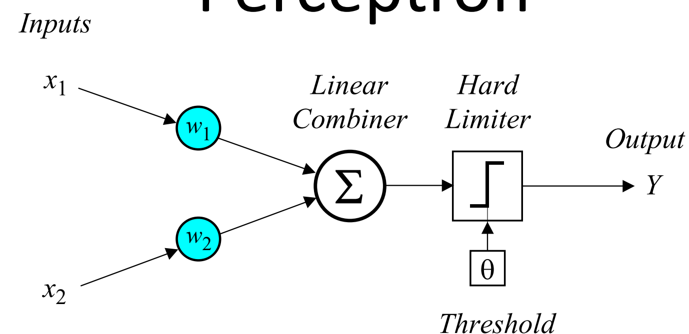
From Biology to Computers…

- Biology





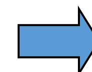
Perceptron

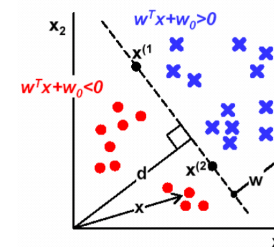


- Activation function

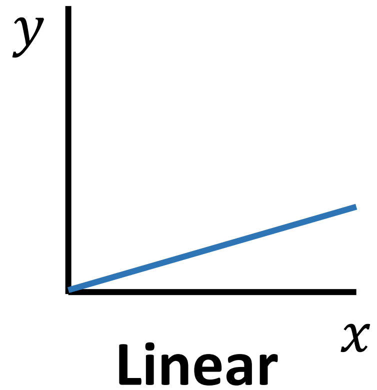
$$X = \sum_{i=1}^{n} x_i w_i$$

$$Y = \begin{cases} +1, & \text{if } X \geq \omega_0 \\ -1, & \text{if } X < \omega_0 \end{cases}$$



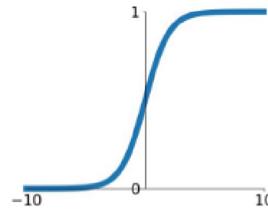


Activation Functions



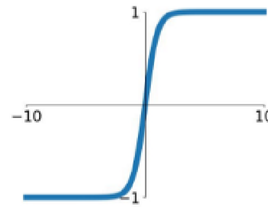
Sigmoid

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



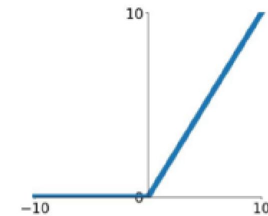
tanh

$\tanh(x)$



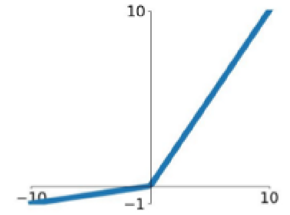
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

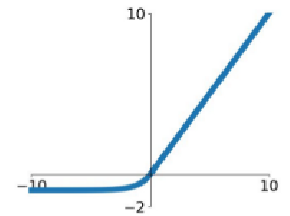


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

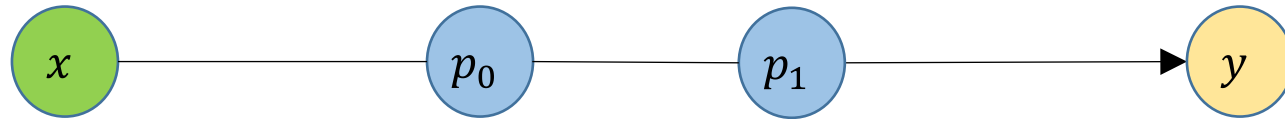
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



NOTE: ReLU = Rectified Linear Unit, ELU = Exponential Linear Unit

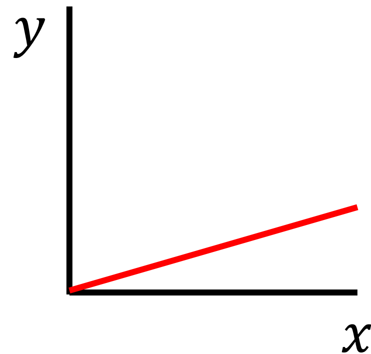
Multilayer Perceptron



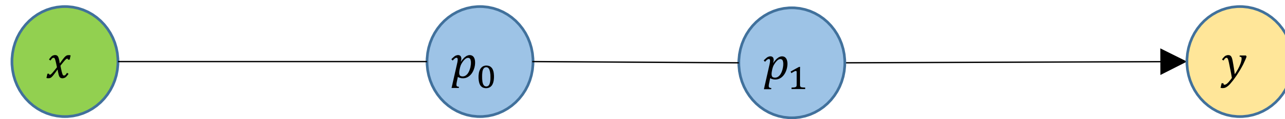
$$h = m_0 x + b_0 \qquad y = m_1 h + b_1$$

$$y = m_1(m_0 x + b_0) + b_1$$

Example: $\mathrm{b}_0 = \mathrm{b}_1 = 0, m_0 = m_1 = 0.5 \rightarrow y = 0.25x$



Multilayer Perceptron



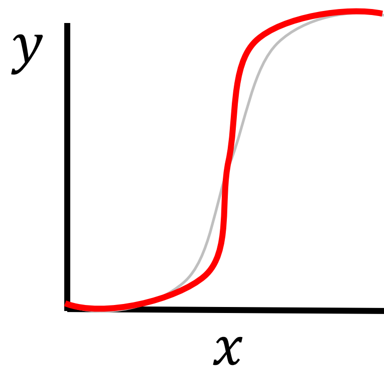
$$h = m_0 x + b_0$$

$$y = \frac{1}{1 + e^{-m_1(h+b_1)}}$$

Example: $\mathrm{b}_0 = \mathrm{b}_1 = 0, m_0 = 2, m_1 = 1$

$$y = \frac{1}{1 + e^{-2x}}$$

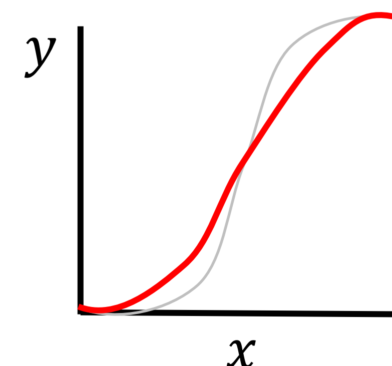
Intuitively: y will be "high" for smaller values of x



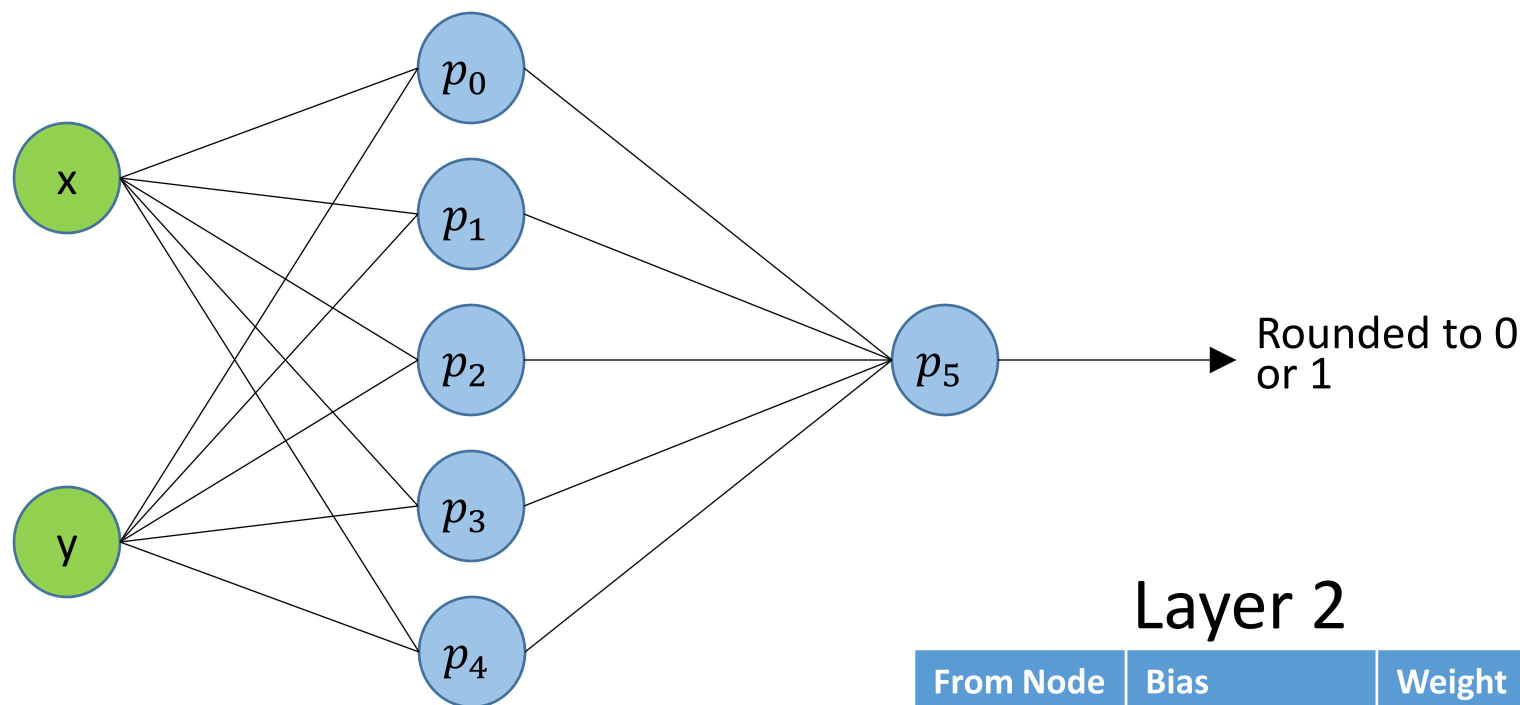
Example: $\mathrm{b}_0 = \mathrm{b}_1 = 0, m_0 = 0.5, m_1 = 1$

$$y = \frac{1}{1 + e^{-0.5x}}$$

Intuitively: y will be "high" for larger values of x



Multilayer Perceptron



(Sigmoid activation functions)

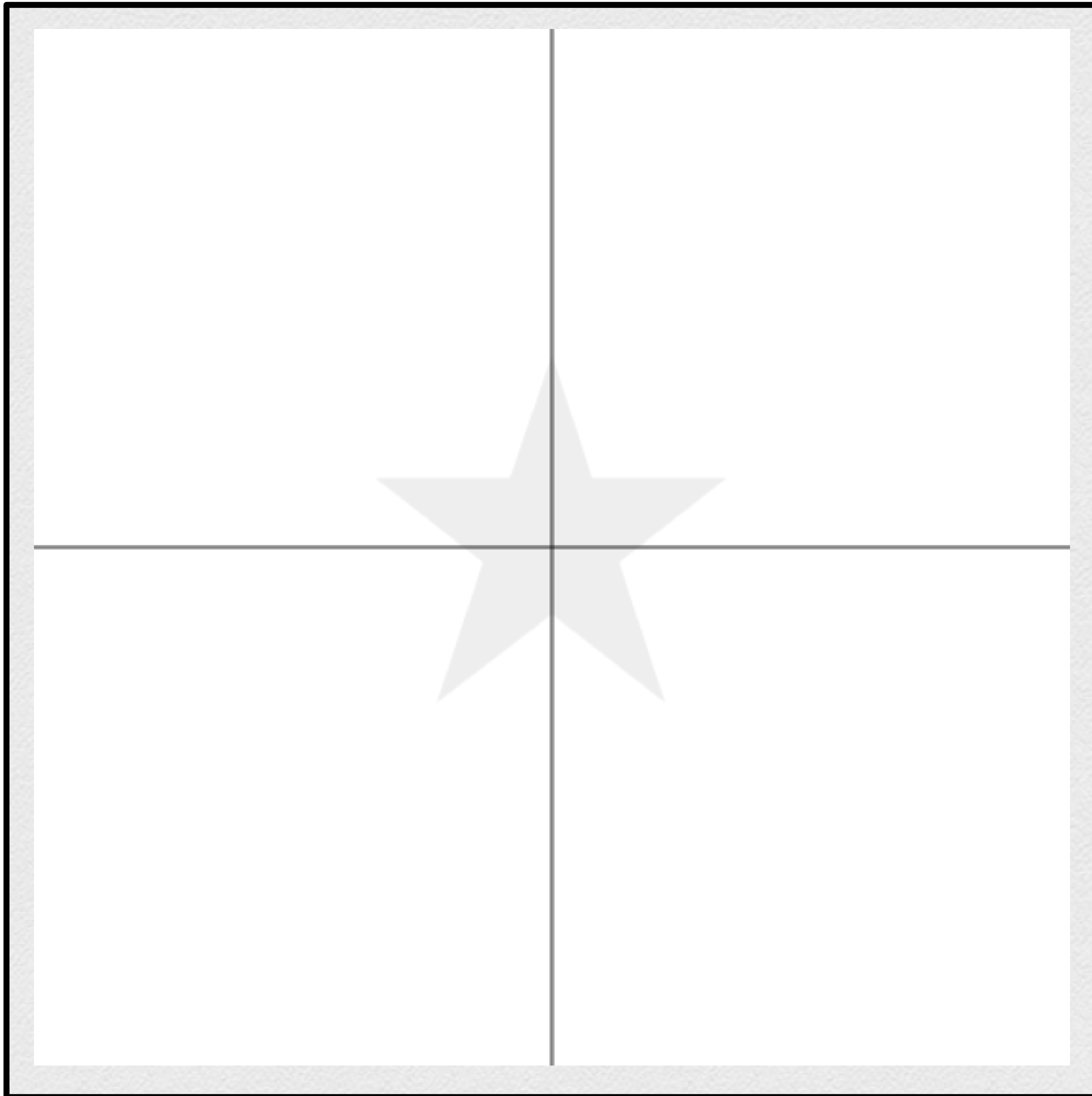
Layer 1

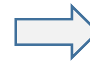
Node	Bias	x-Weight	y-Weight
0	-0.375	-3	1
1	-0.125	0	1
2	-0.375	3	1
3	0.125	-0.75	1
4	0.125	0.75	1

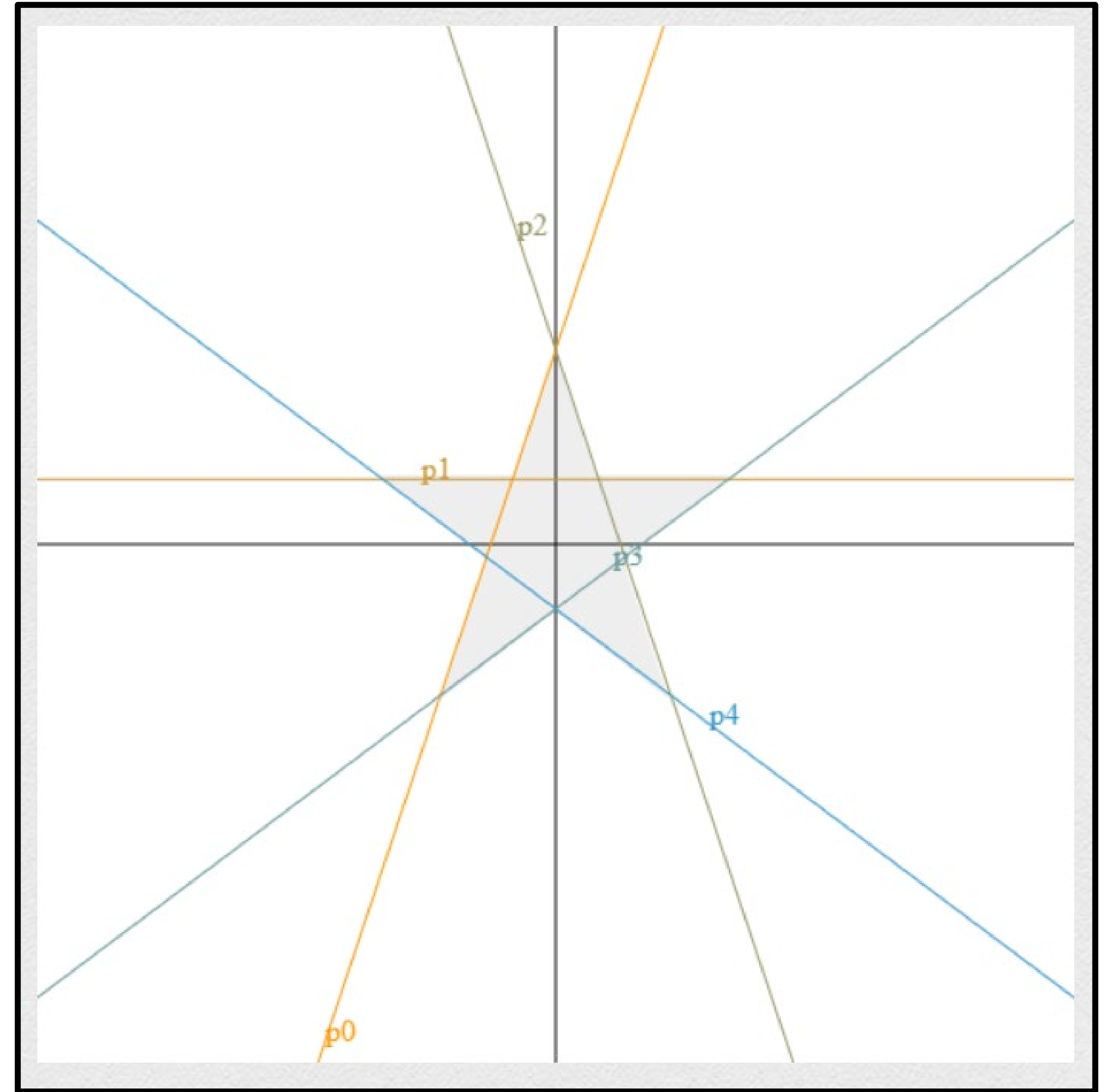
Layer 2

From Node	Bias	Weight
0	-0.2	1
1	-0.2	1
2	-0.2	1
3	-0.2	1
4	-0.2	1

Star Classifier: https://www.cs.utexas.edu/~teammco/misc/mlp







Basic Machine Learning Recipe

1. Obtain training data
2. Choose decision and loss functions
3. Define goal
4. Optimize!

Training Data

$\{x_i, y_i\}$ for i $\in [1, N]$

Fundamental categories:

1. Synthetic data
2. Real data (annotated)
3. Real data (unannotated) <- tricky!

Properties:

1. Data should span/populate the distribution of expected input values
2. Data should be plenty – kinda same as above
3. Data should have low errors/noise (ideally)

Decision and Loss Functions

$$\hat{y} = \mathrm{f}_{\theta}(\mathrm{x_i})$$

The function you wish to "decide" that given the inputs, and the parameters θ, yields an output $\hat{y}$ that is equal or close to desired values; thus, you seek

$$l(\hat{y}, y_i) \rightarrow 0$$

Properties:

1. Decision should be "doable" so that convergence is possible
2. Loss function should exploit as much as possible of domain knowledge

Define (Training) Goal

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{N} l(f_\theta(x_i), y_i)$$

Define a function to find parameters θ^* that minimize the loss function for the entire training data set; i.e., find network weights and biases that make the network “learn” the desired (high-dimensional) function

Optimize!

- Perform small steps (opposite the gradient)…

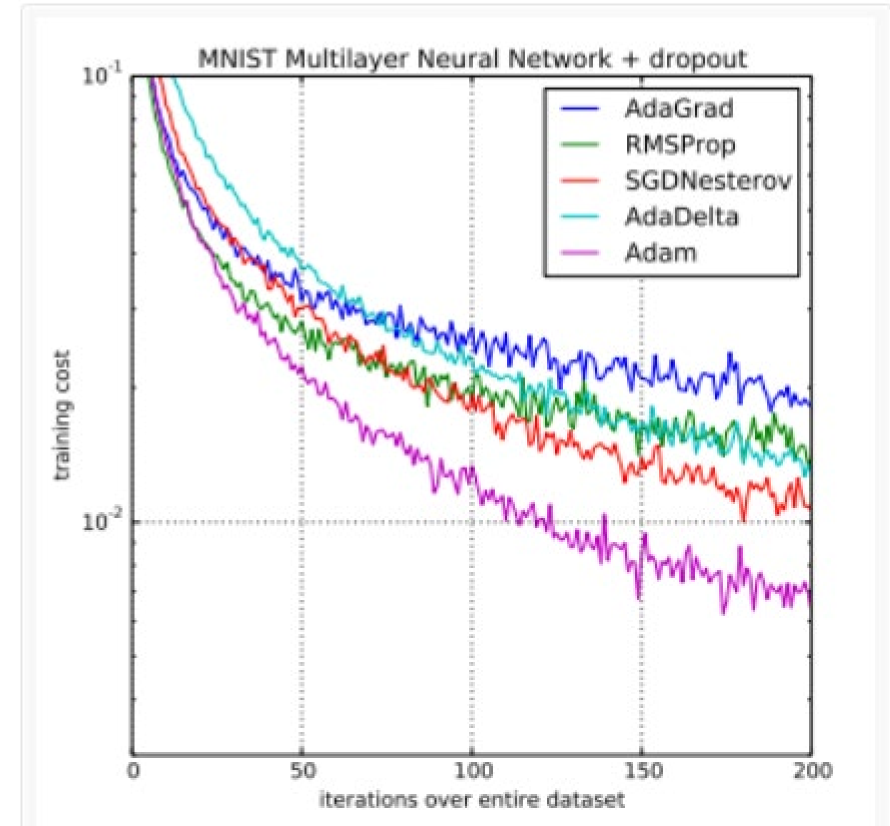
$$\theta^{t+1} = \theta^{t} - \alpha_t \nabla l(\mathrm{f}_\theta(x_i), y_i)$$

Move a small step against the gradient to eventually reach a set of network parameters that minimize the loss function

Optimize!

- Methods:
  - Stochastic Gradient Descent (SGD),
  - Adam, or
  - Others

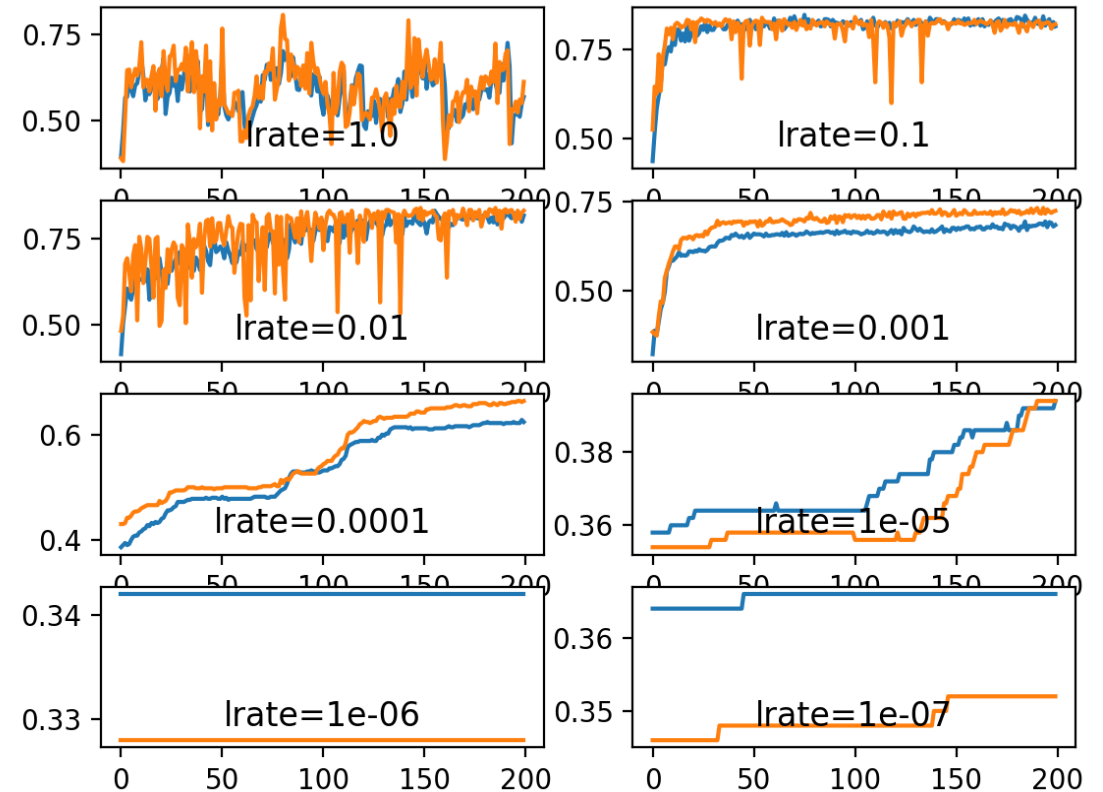
- Adam: an adaptive moment estimation based optimization – the learning rate changes during the optimization [Kingma and Ba, 2015]



Comparison of Adam to Other Optimization Algorithms Training a Multilayer Perceptron
Taken from Adam: A Method for Stochastic Optimization, 2015.

SGD: Learning Rate

- It is a scale factor for how much the network parameters (θ) are updated during training
- To the right is a graph of different learning rates for 3-blob classification trained on multilayer perceptron of 50 nodes, using ReLU, for 200 epochs
  - Orange = train
  - Blue = test
  - Best is 0.1 to 0.001 (in this case)

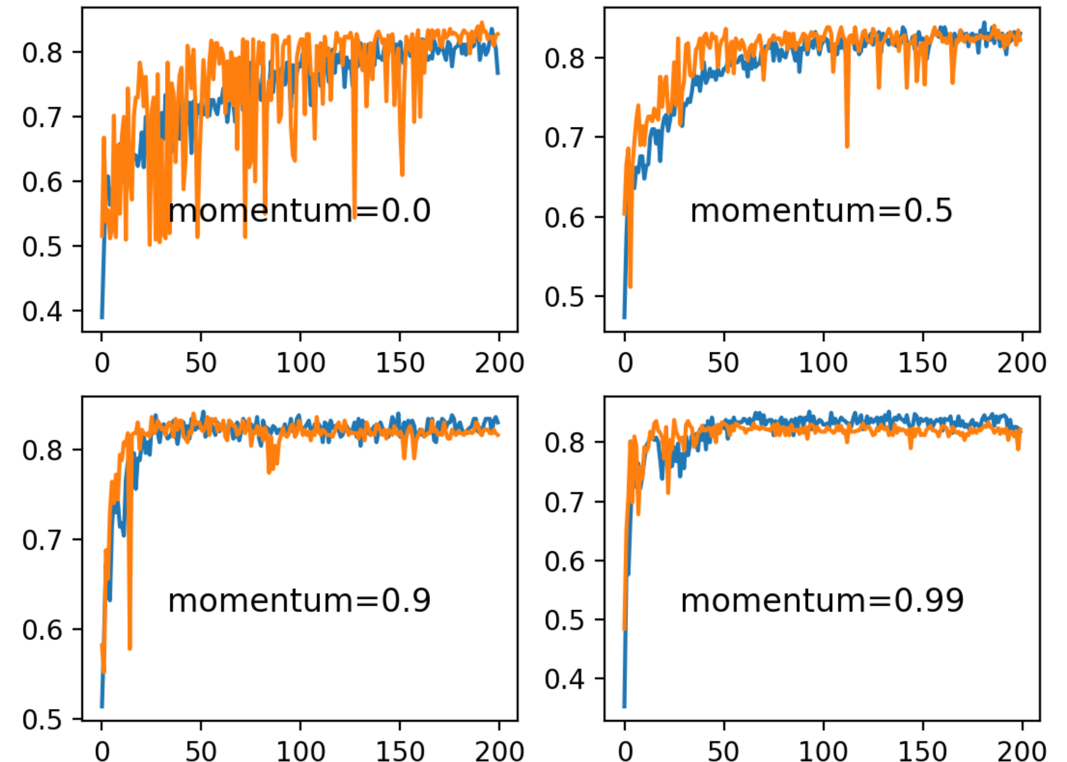


SGD: Moment

- It is like giving the optimization step short term memory and keeping it partially moving the direction it was going – in a sense a dynamic learning rate
- Effect of moment-based SGD on the same example as previous slide:
  - Best is 0.9 or 0.99 in this case because converged and fastest
  - Formulation?

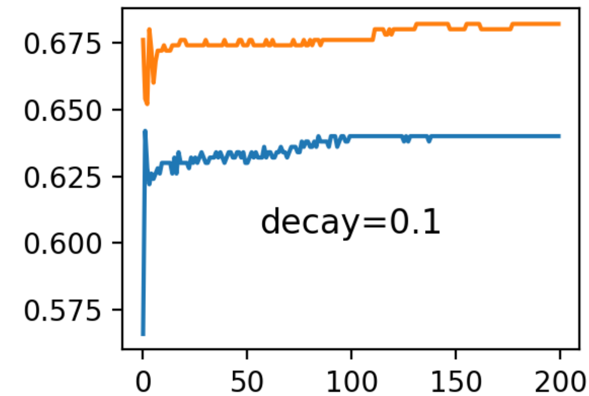
$$v_{t+1} = \mu v_t - \varepsilon \nabla f(\theta_t) \quad (1)$$
$$\theta_{t+1} = \theta_t + v_{t+1} \quad (2)$$

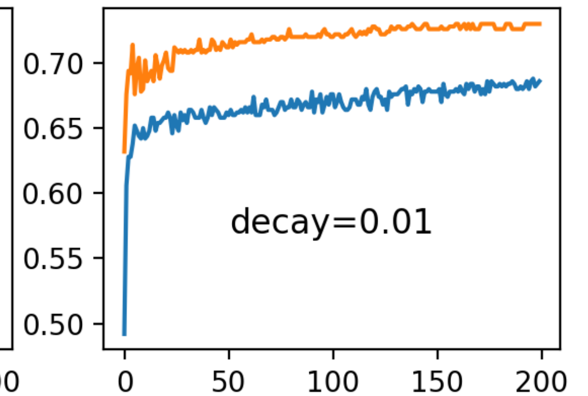
where $\varepsilon > 0$ is the learning rate, $\mu \in [0, 1]$ is the momentum coefficient, and $\nabla f(\theta_t)$ is the gradient at θ_t.

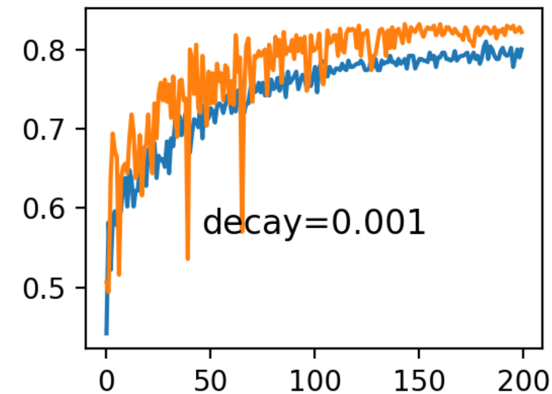


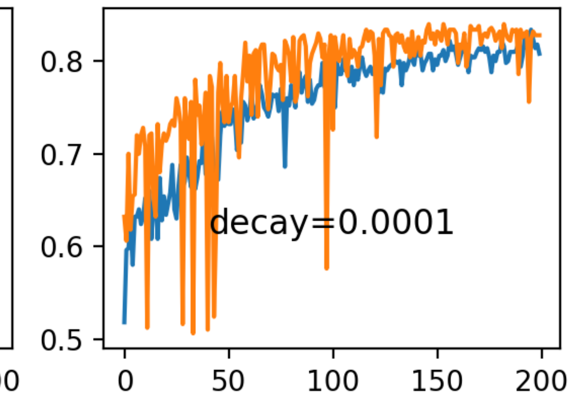
SGD: Learning Rate Decay

- Slowly reduce the learning rate
- Using same example from before, we experiment with different decay rates



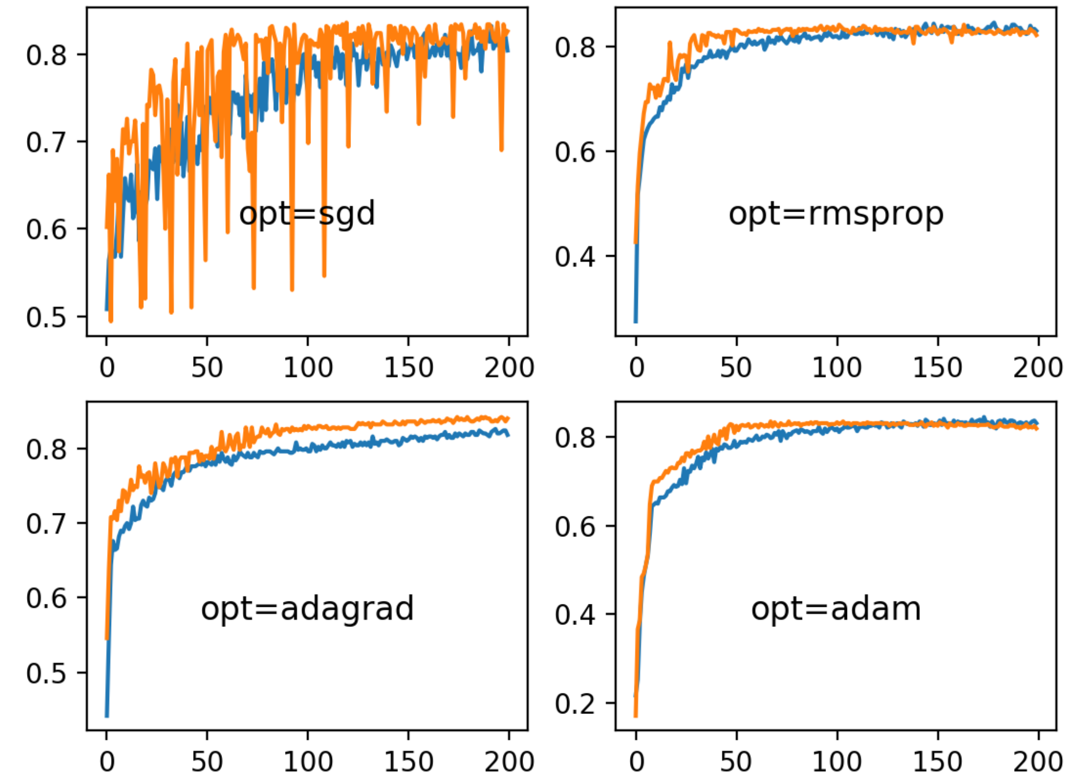






Adaptive Learning Rate Methods

- Different adaptive learning rate methods on the same example
  - SGD has fixed learning rate 0.01



Back Propagation

```
Initialize all network parameters with small random numbers (e.g., [-1,1])

REPEAT
FOR every pattern in the training set
    // Propagate the input forward through the network:
    Present the pattern to the network
    FOR each layer in the network
        FOR every node in the layer
            1. Calculate the weight sum of the inputs to the node
            2. Add the bias to the sum
            3. Calculate the activation for the node
        end
    end
```

```
 // Propagate the errors backward through the network
  FOR every node in the output layer
      Calculate the error signal
  end
  FOR all hidden starting at outmost layers
      FOR every node in the layer
          1. Calculate the node's signal error
          2. Update each node's weight in the network
      end
  end

  // Calculate Global Error
  Calculate the Error Function
end
UNTIL ((maximum number of iterations > than specified) OR
       (Error Function is < than specified))
```

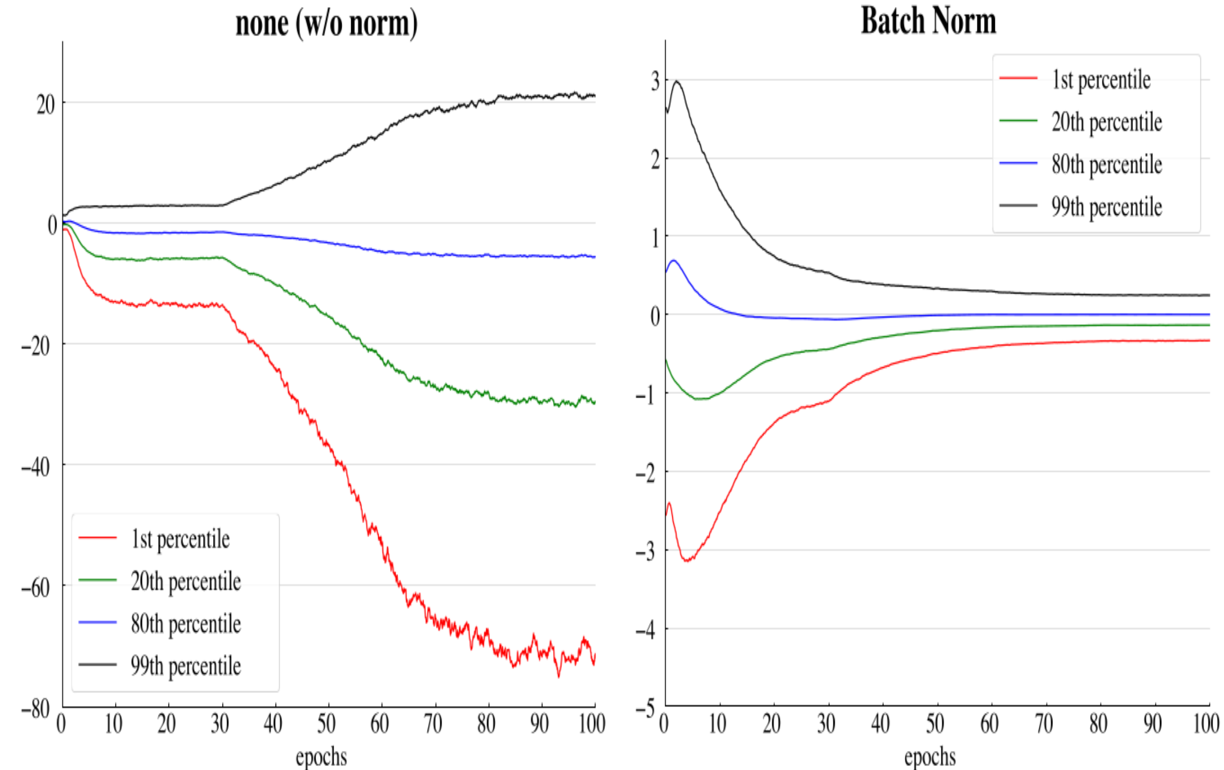
Batch Normalization

- Rather than compute gradient per (x_i, y_i):
  - compute for a "mini batch" and then,
  - normalize the output of the previous output layer by subtracting the mean over the batch divided by the standard deviation
  - This reduces internal covariant shift and makes things more "Gaussian"

  [Ioffe and Szegedy, 2015]

$$\theta^{t+1} = \theta^t - \alpha_t/m \sum_{i=1}^{m} \nabla l(\mathrm{f}_\theta(x_i), y_i)$$

  - Typical batch sizes are few to 100(?)



- https://playground.tensorflow.org

Batch Normalization

- One formulation:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Deep Graphics?

- Lets start with NERF:
  https://www.matthewtancik.com/nerf