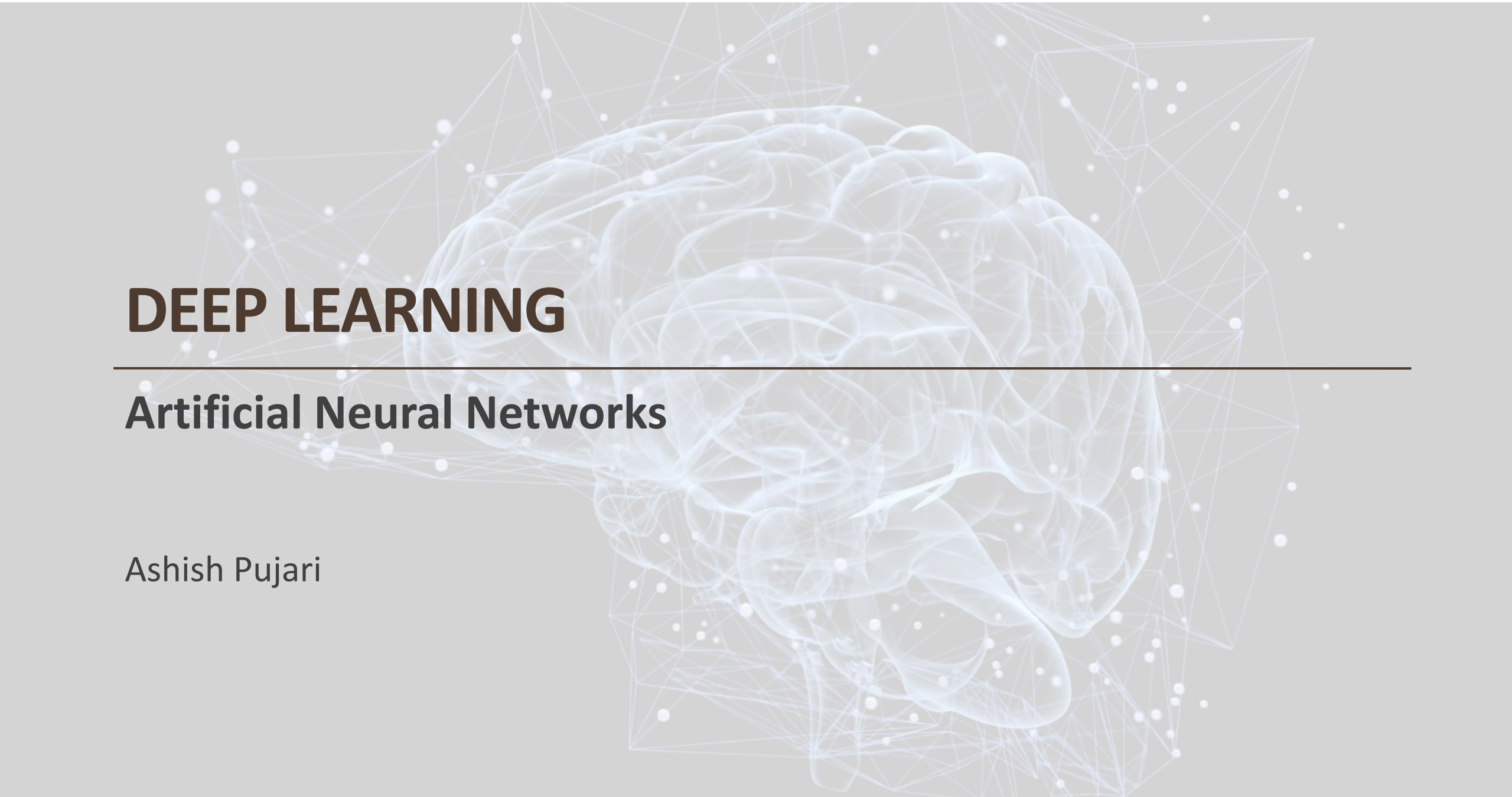


DEEP LEARNING

Artificial Neural Networks

Ashish Pujari

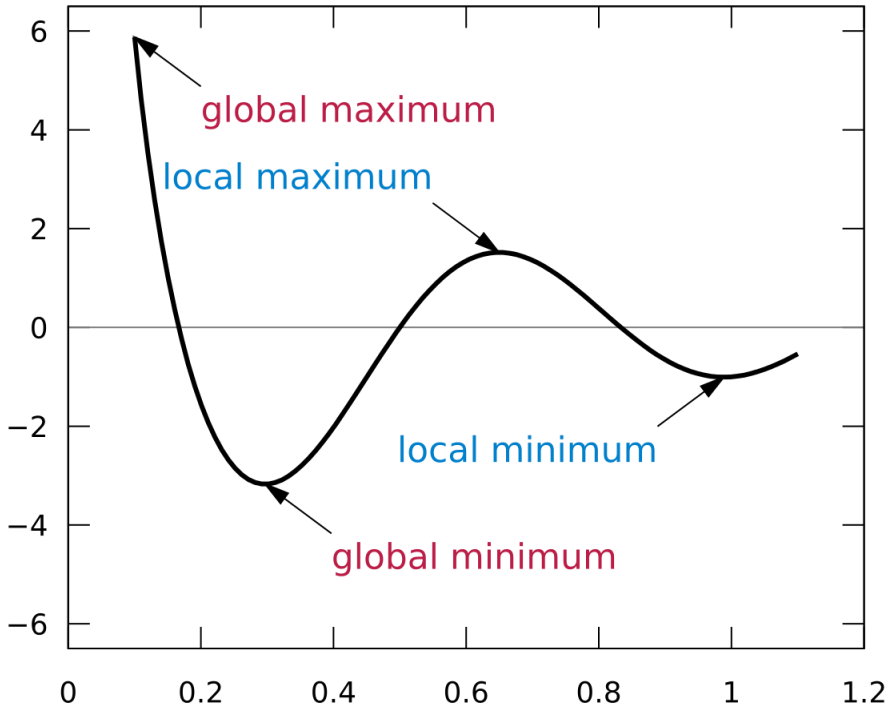
Lecture Outline

- Gradient Descent
- Artificial Neural Networks (ANNs)
- ANN Training

GRADIENT DESCENT (GD)

Optimization

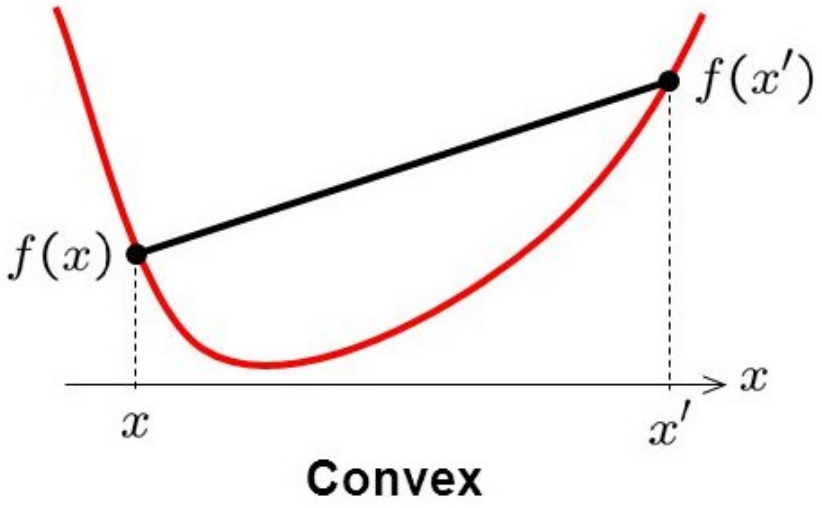
- Optimization deals with finding the maxima and minima of a function that depends on one or more variables.



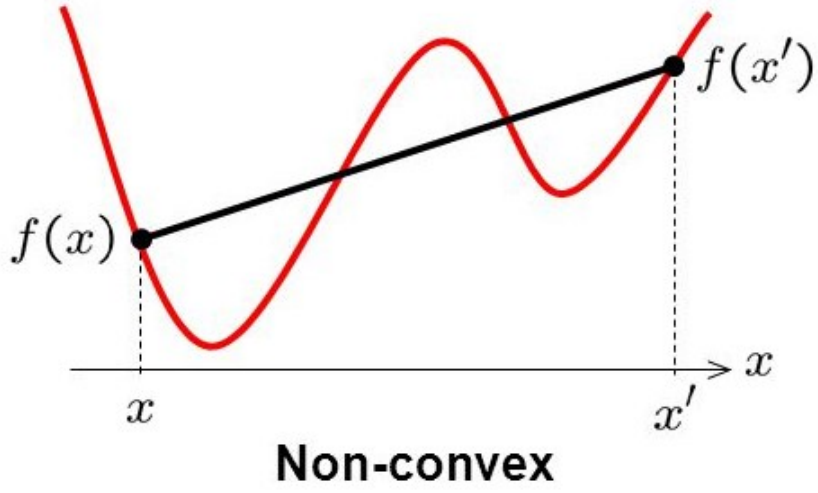
Convex vs Non-Convex Functions

- A function $f: A \subseteq \mathbb{X} \rightarrow \mathbb{R}$ defined over a convex set A is called Convex if

$$f(\lambda x + (1-\lambda)x') \leq \lambda f(x) + (1-\lambda) f(x') \qquad \text{For any } x, x' \epsilon\, \mathbb{X} \;\; and \;\; \lambda \in [0,1]$$



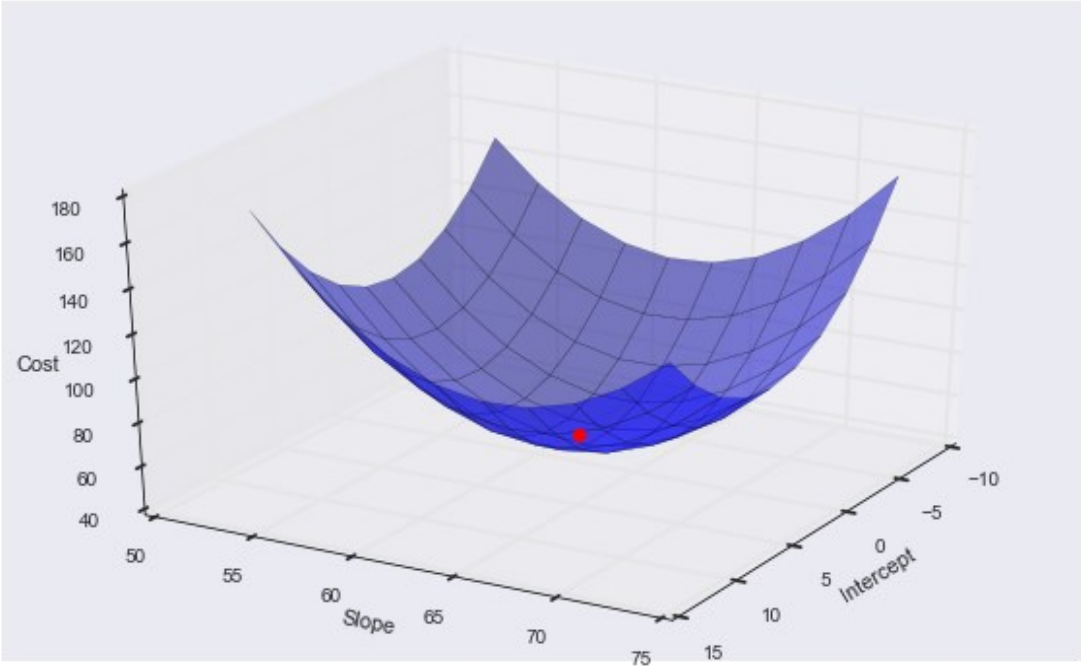
Convex



Non-convex

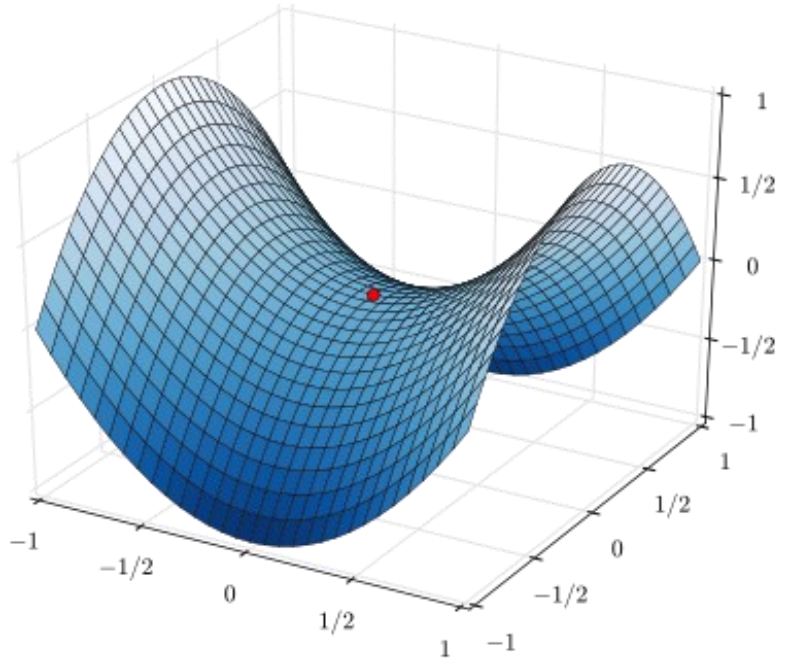
Convex vs Non-Convex Functions

Convex



Error surface for simple linear model with quadratic error

Non-convex



A saddle point on the graph of $z=x^2-y^2$ (red)

Gradient

- Derivative of $f(x)$ is scalar valued

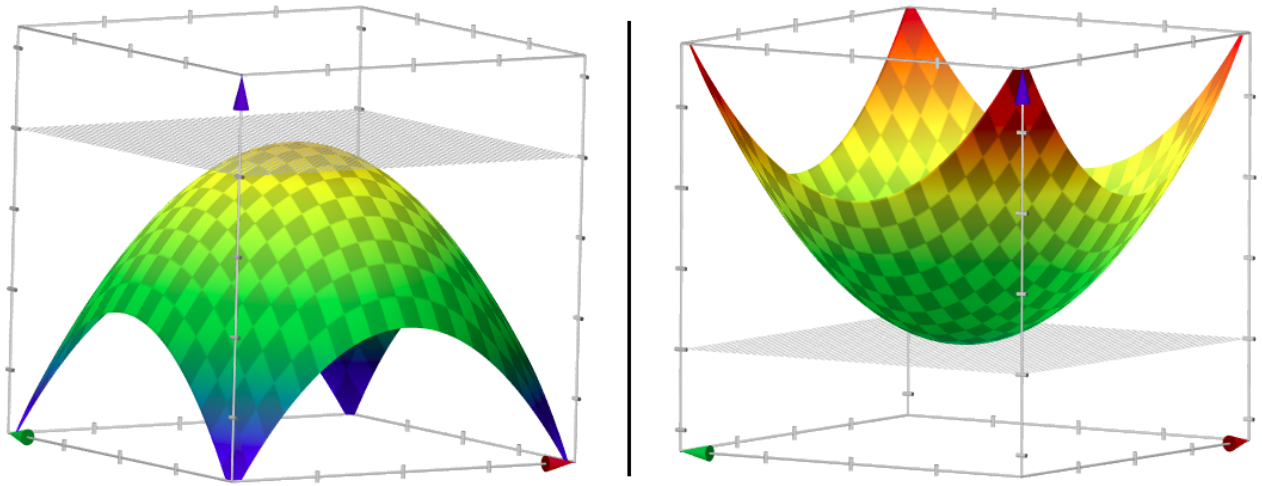
$$f'(x) = \frac{dF}{dx}$$

- Gradient of $f(x, y)$ is vector-valued function which contains partial derivatives

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

Gradient

- A gradient points in the direction of greatest increase of a function
- It is zero at a local maximum or local minimum

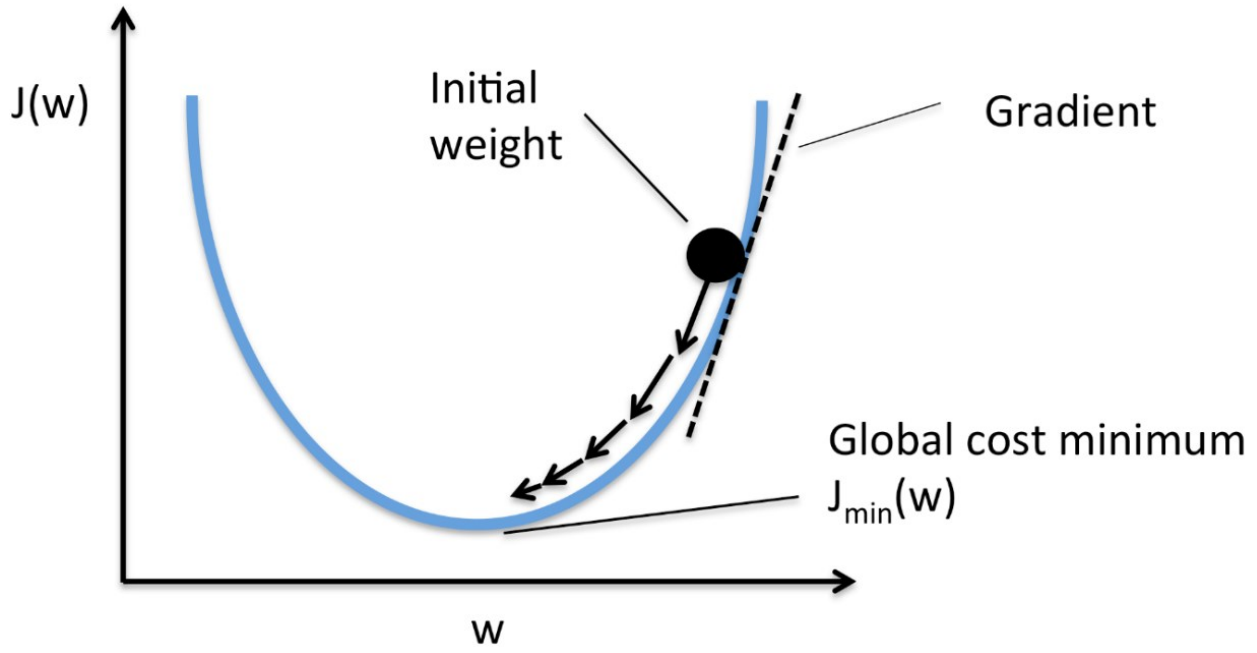


The tangent planes to the surfaces $z=-x^2-y^2$ and $z=x^2+y^2$ at their maximum and minimum respectively.

Gradient Descent (GD)

$$\Delta\omega = -\eta\nabla J(\omega)$$

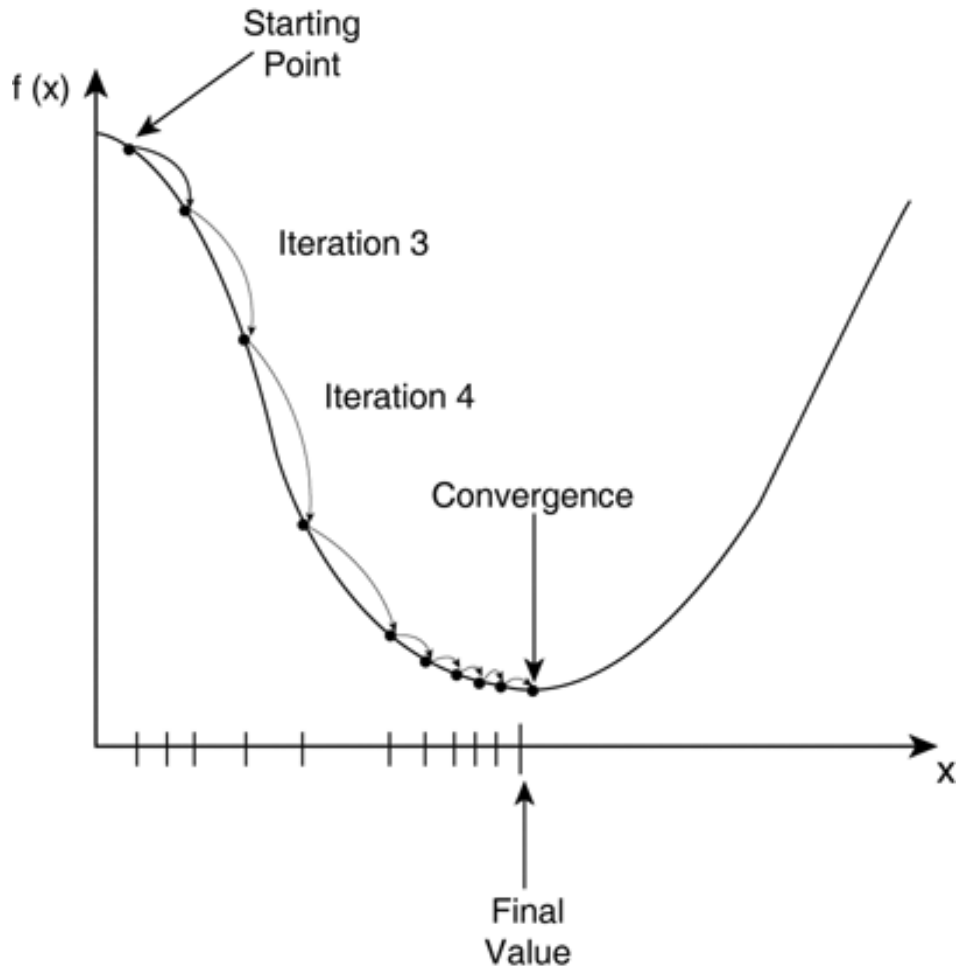
$$\Delta\omega_i(t+1) = -\eta\frac{\partial J}{\partial w_i}$$



Gradient Descent (GD)

$$\Delta\omega = -\eta\nabla J(\omega)$$

$$\Delta\omega_i(t+1) = -\eta\frac{\partial J}{\partial w_i}$$



Batch Gradient Descent

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

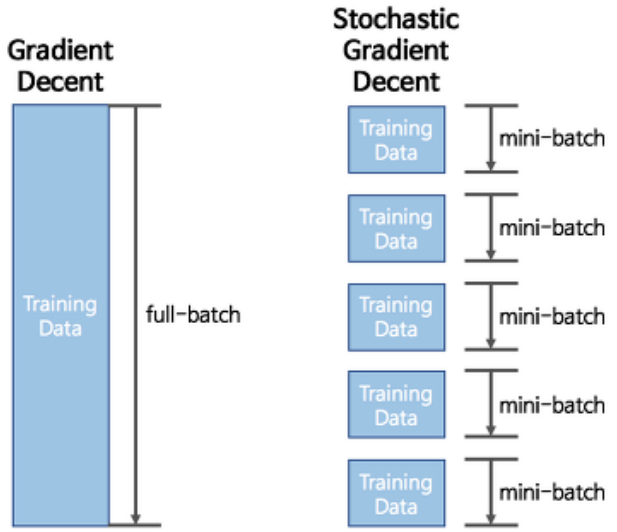
```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

http://ruder.io/optimizing-gradient-descent/

Stochastic Gradient Descent (SGD)

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

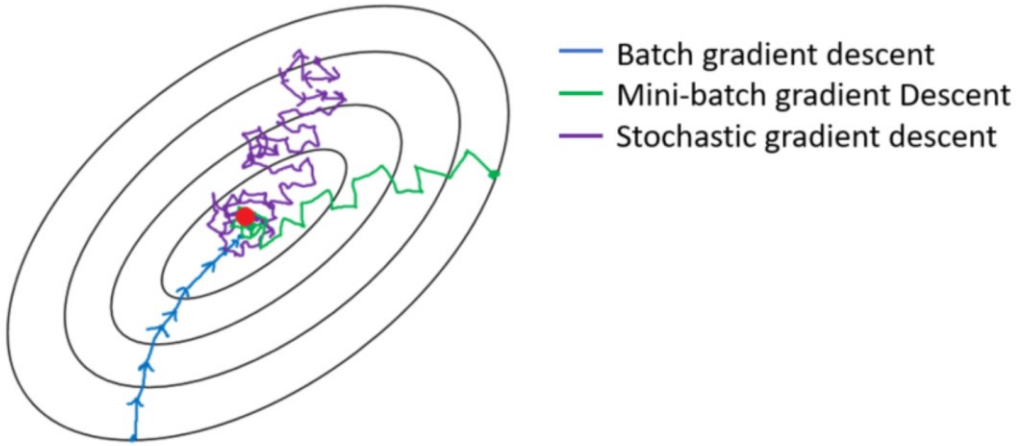


http://ruder.io/optimizing-gradient-descent/

Mini-Batch Gradient Descent

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```



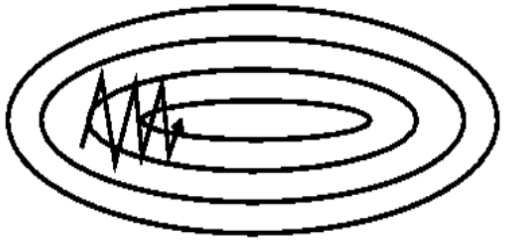
http://ruder.io/optimizing-gradient-descent/

Momentum

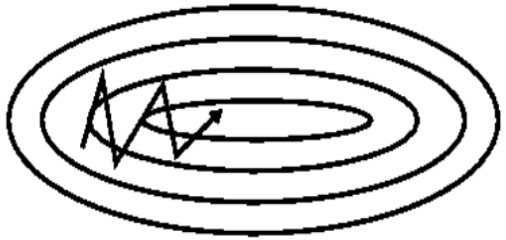
- Momentum helps accelerate SGD in the relevant direction and dampens oscillations

$$\Delta\omega_i(t+1) = -\eta\frac{\partial J}{\partial w_i} + \alpha\Delta\omega_i(t)$$

learning rate momentum



SGD Without momentum

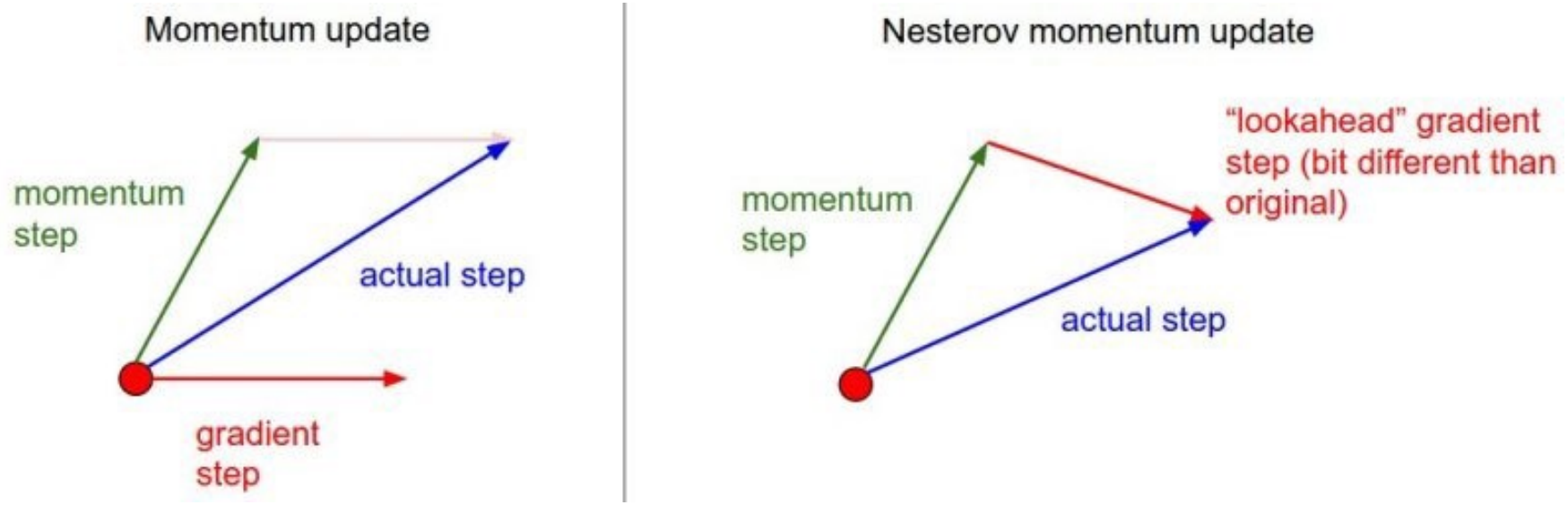


SGD with momentum

https://distill.pub/2017/momentum/

Nesterov Momentum

- Gradient term is not computed from the current position ω_t in parameter space but instead from a position $\omega_{\text{intermediate}} = \omega_t + \mu\, v_t$.

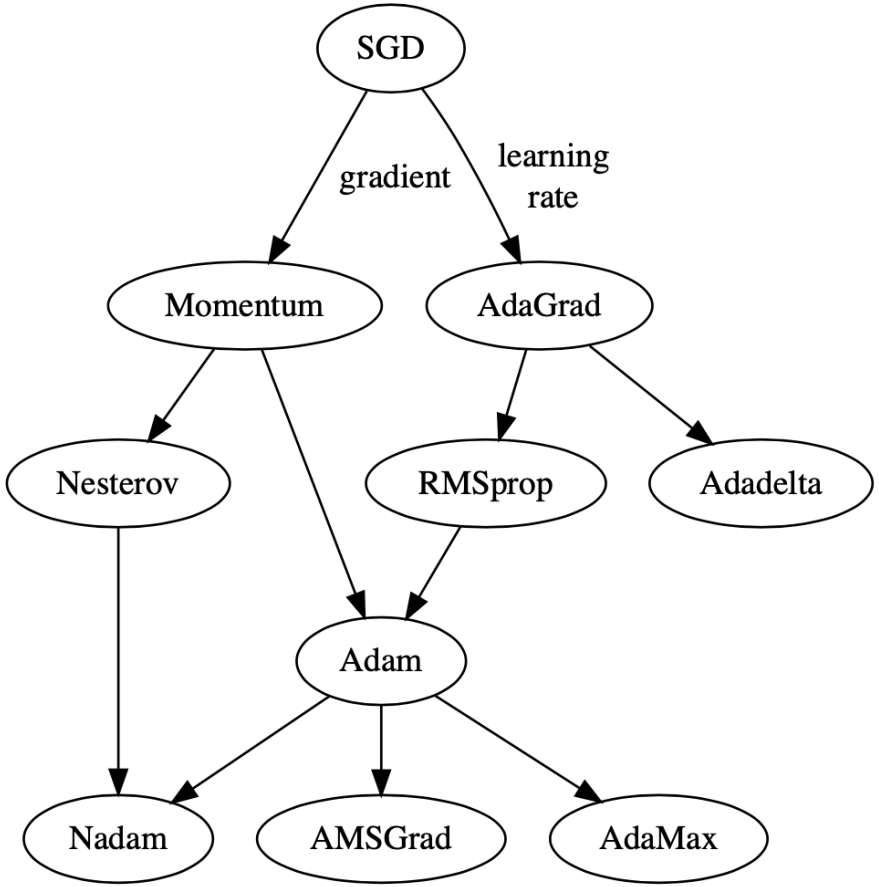


https://distill.pub/2017/momentum/

SGD: Variants

- Optimizers can act upon SGD in following ways :
  - modifying the learning rate component
  - modifying the gradient component,
  - modifying both.

Optimiser	Year	Learning Rate	Gradient
Momentum	1964		✓
AdaGrad	2011	✓	
RMSprop	2012	✓	
Adadelta	2012	✓	
Nesterov	2013		✓
Adam	2014	✓	✓
AdaMax	2015	✓	✓
Nadam	2015	✓	✓
AMSGrad	2018	✓	✓



https://towardsdatascience.com/10-gradient-descent-optimisation-algorithms-86989510b5e9

SGD: Variants

Algorithm	Description	Details
AdaGrad	• Adapts learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. • Uses different learning rate for every parameter θ_i at every time step t • Well-suited for sparse data and less sensitive to hyperparameters	$g_{t,i} = \nabla_\theta J(\theta_{t,i})$. $\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$. $\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$.
AdaDelta	• Extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate • Restricts the window of accumulated past gradients to some fixed size	$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$. $\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$.
RMSProp	• Root mean square prop (Hinton et al., 2012) is another adaptive learning rate that is an improvement of AdaGrad. • Identical to the first update vector of AdaDelta. Like AdaDelta resolves AdaGrad's radically diminishing learning rates	$E[g^2]_t = 0.9 E[g^2]_{t-1} + 0.1 g_t^2$ $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$
Adam	• First-order gradients with little memory requirement • Adaptive Moment Estimation • AdaGrad + RMSProp	$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$ $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$

http://ruder.io/optimizing-gradient-descent/

Adam (2014)

- Adaptive learning rate optimization algorithm that utilizes both momentum and scaling, combining the benefits of RMSProp and SGD with Momentum

$$\omega_t = \omega_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad , \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) {g_t}^2$$

where ϵ is a small number,
β_1, β_2 are forgetting parameters with values 0.9 and 0.999

https://arxiv.org/pdf/1412.6980v9.pdf

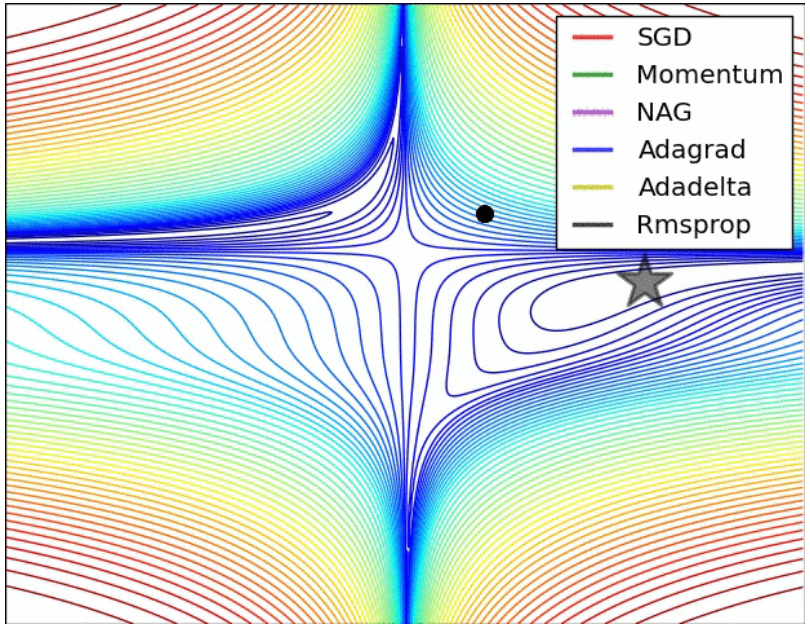
Adam (2014)

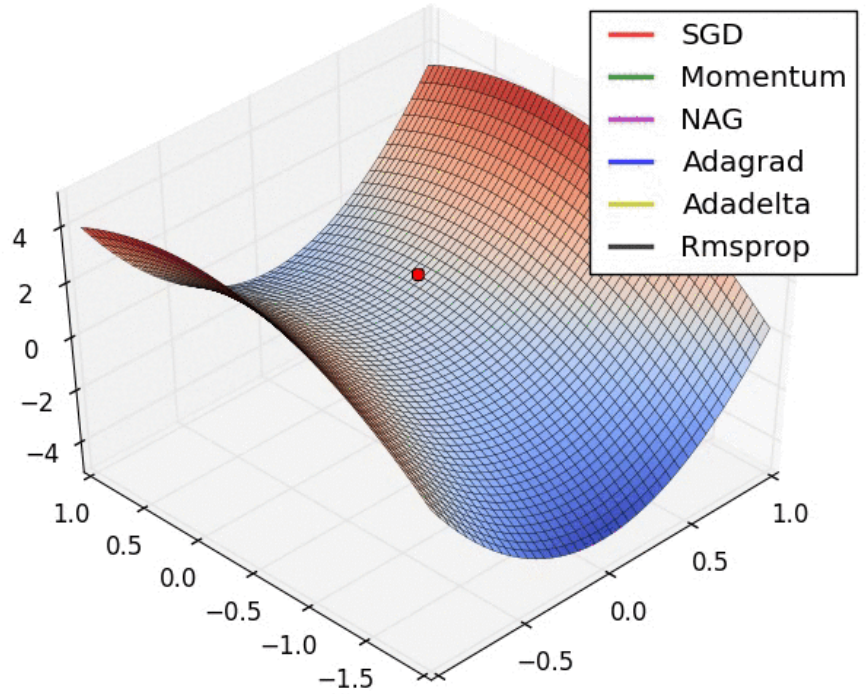
Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

https://arxiv.org/pdf/1412.6980v9.pdf

SGD: Variants





Alec Radford

Optimizing Gradient Descent - http://ruder.io/optimizing-gradient-descent/

SGD: Best Practices

- Rapid prototyping
  - Use adaptive techniques like Adam/Adagrad - quicker results without much hyper parameter tuning.
- Best results
  - Use vanilla gradient descent with momentum - slower to converge, but better than adaptive techniques.
- Very small dataset
  - Can be fit in a single iteration, use 2nd order techniques like l-BFGS.

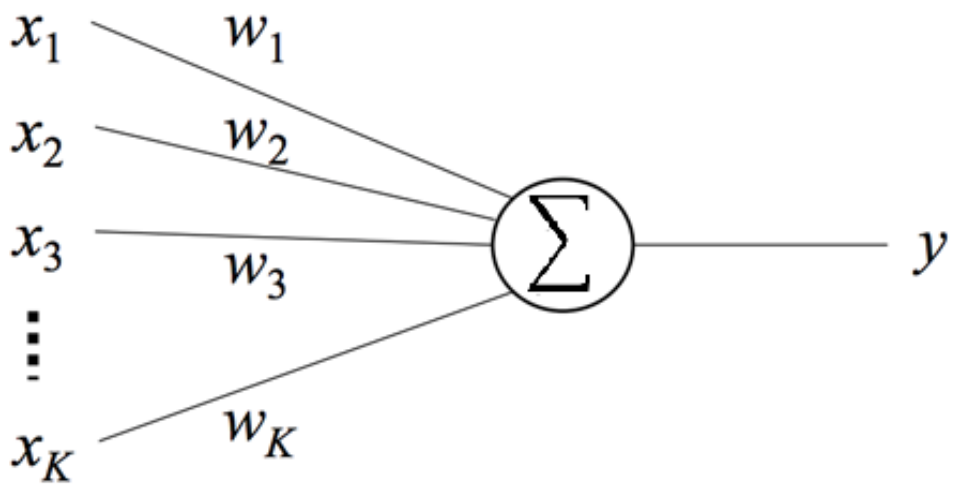
ARTIFICIAL NEURAL NETWORKS

Linear Model, Biology, Perceptron Model

Linear Model

- Linear model assumes linear relationship between variables
- Relies on matrix multiplications which are easy to compute

$$y = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \sum_{i=1}^{n} w_i x_i = \boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{x}$$



Affine Transformation

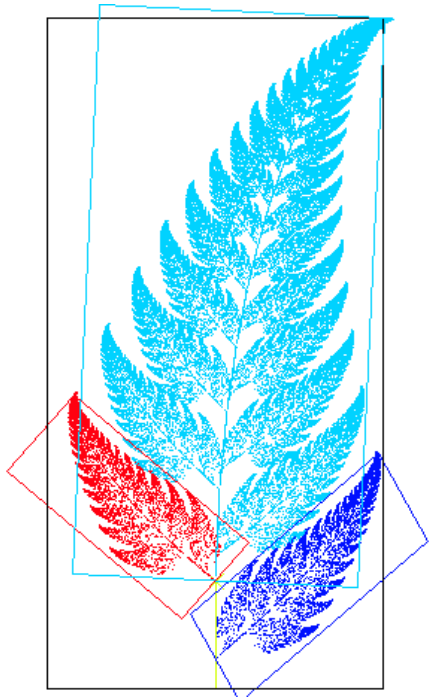
- Preserves collinearity (i.e., all points lying on a line initially still lie on a line after transformation) and ratios of distances (e.g., the midpoint of a line segment remains the midpoint after transformation).

$$xW + b$$

$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} \end{bmatrix}$$

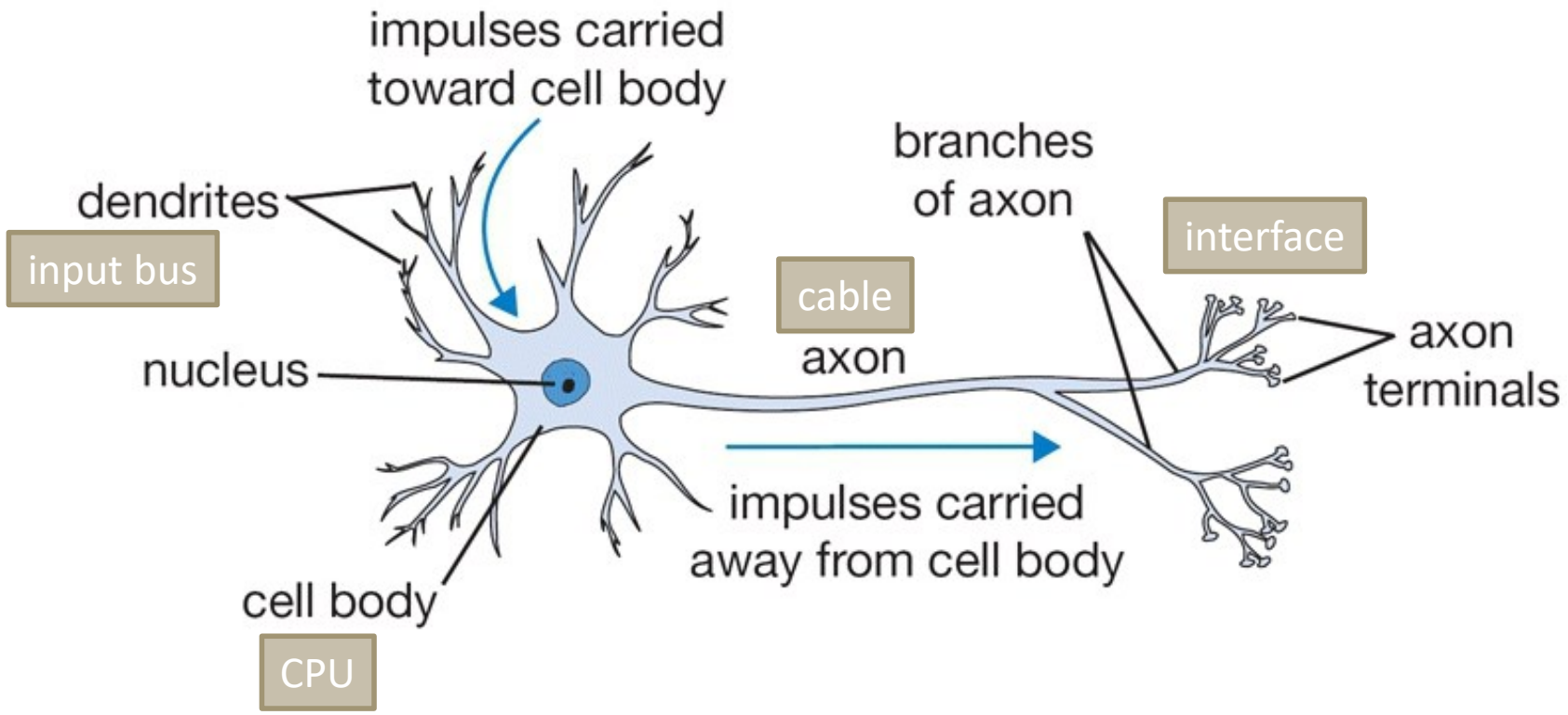
$$\begin{bmatrix} x_1 w_{11} + x_2 w_{21} + b_{11} & x_1 w_{12} + x_2 w_{22} + b_{12} \end{bmatrix}$$

$$\begin{bmatrix} z_1 & z_2 \end{bmatrix}$$



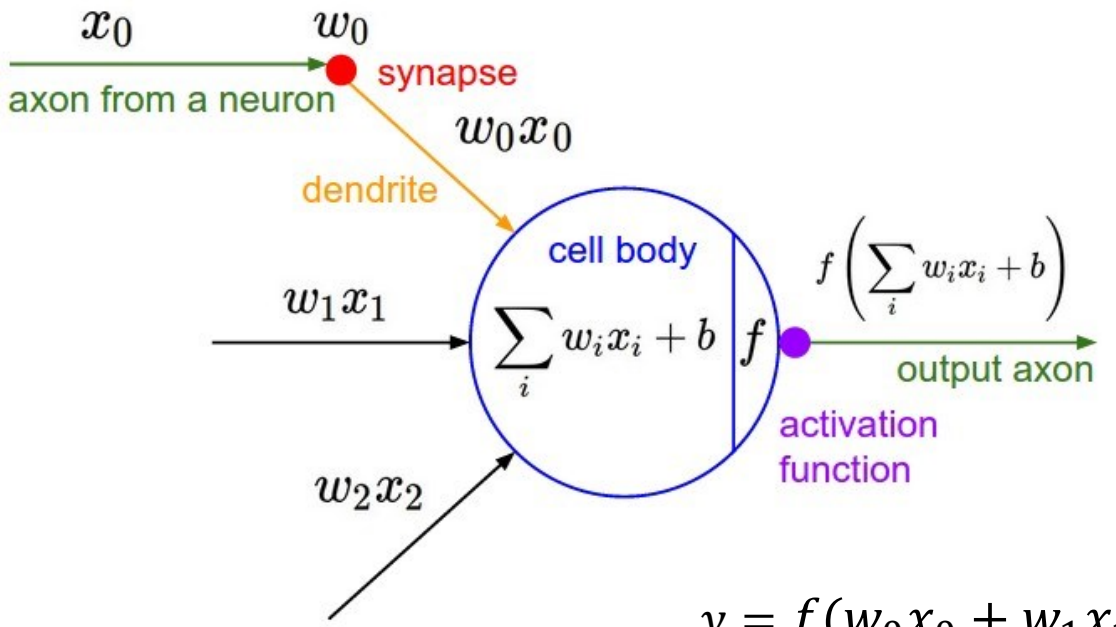
https://en.wikipedia.org/wiki/Affine_transformation

Biological Neurons



Perceptron

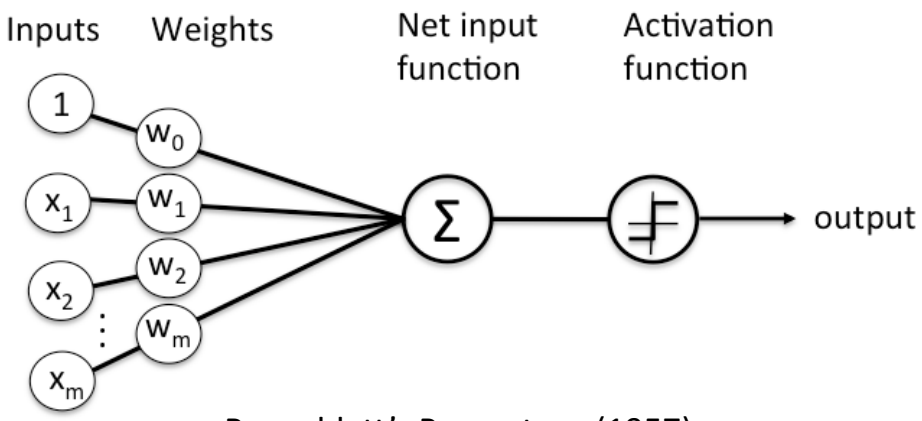
- A neuron is a function that maps an input vector $\{x_0, \dots, x_n\}$ to a scalar output y via a weight vector $\{w_0, \dots, w_n\}$ and a nonlinear function f



$$y = f(w_0 x_0 + w_1 x_1 + \cdots + w_n x_n) = f(\sum_{i=1}^{n} w_i x_i) = \boldsymbol{f(w^T x)}$$

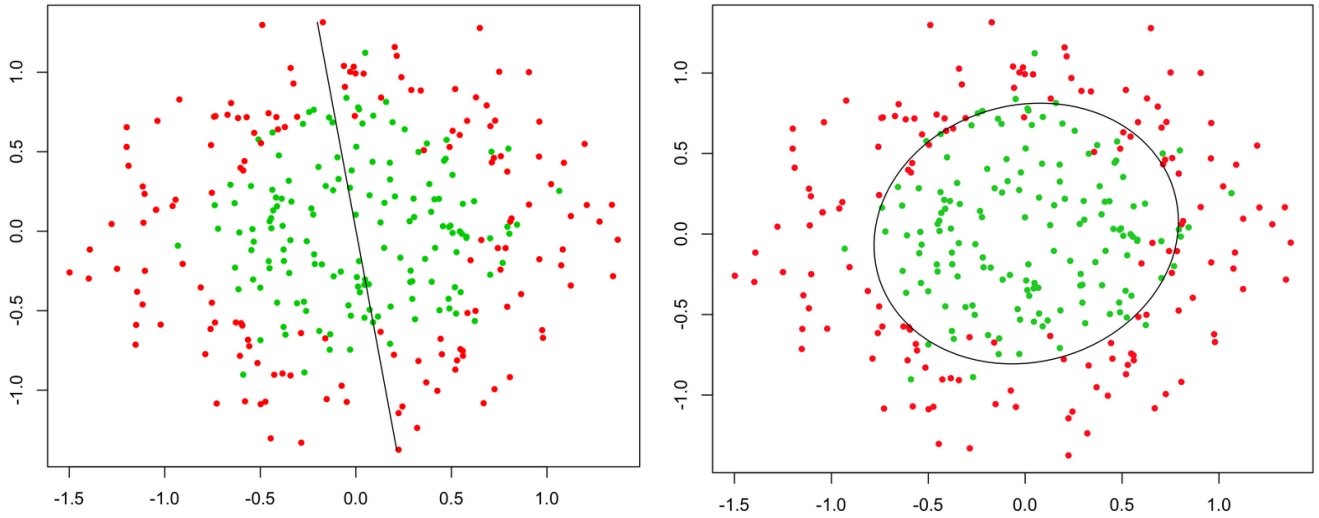
Perceptron

- Perceptron is a single-layer binary linear classifier
- Convergence is only guaranteed if the two classes are linearly separable



Rosenblatt's Perceptron (1957)

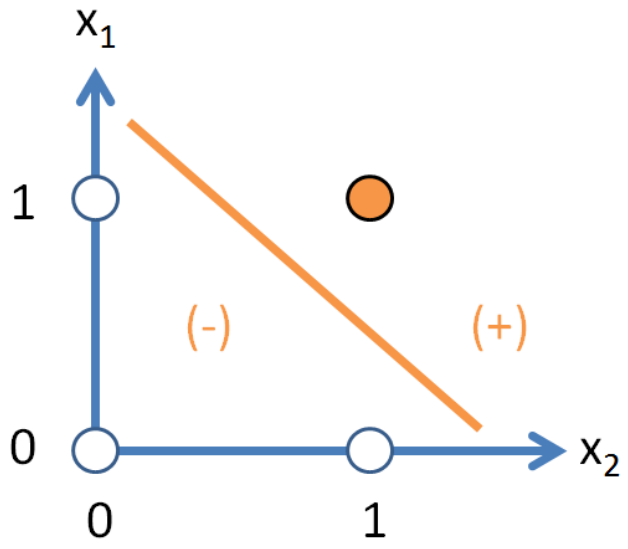
$$g(\mathbf{z}) = \begin{cases} 1 & \text{if } \mathbf{z} \geq \theta \\ -1 & \text{otherwise.} \end{cases}$$



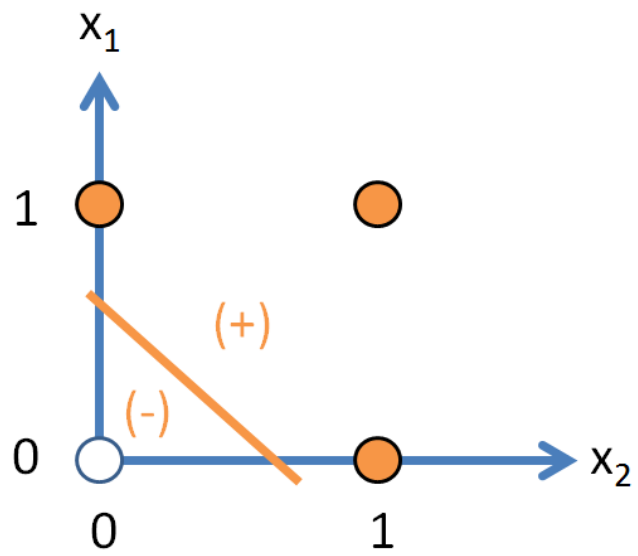
A perceptron cannot classify non-linear decision boundaries (1970)

Perceptron and Logical Gates

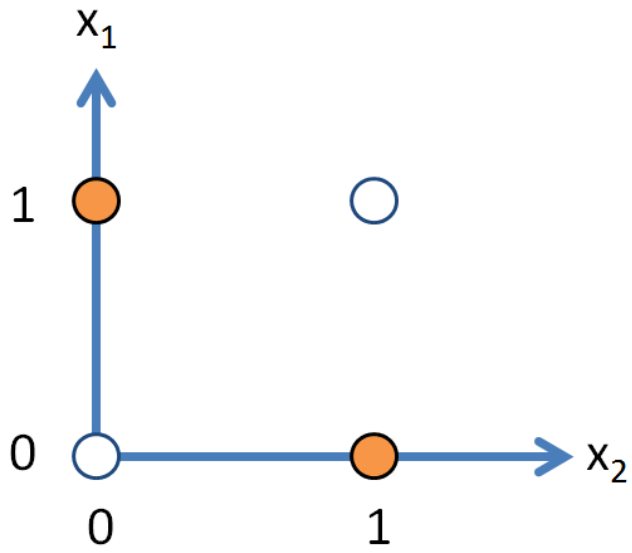
Perceptron Logical “AND”



Perceptron Logical “OR”



Perceptron Logical “XOR”



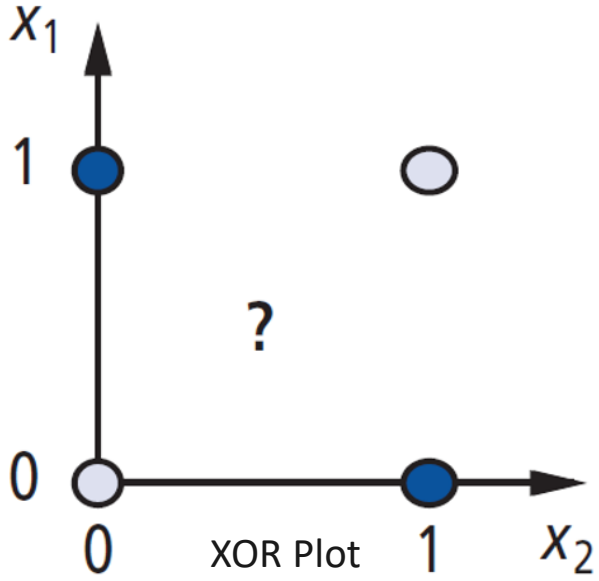
x	*y*	*AND(x,y)*	*OR(x,y)*	*XOR(x,y)*
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

XOR Gate

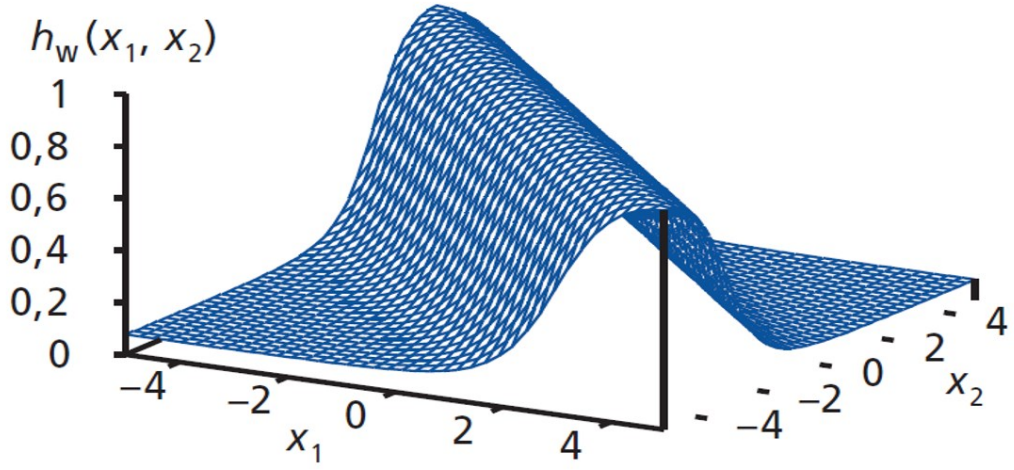
- Exclusive OR (exclusive disjunction) is a logical operation that outputs true only when inputs differ (one is true, the other is false). Exclusive OR gate is not linearly separable
- It is impossible for a classifier with linear decision boundary to learn an XOR function

INPUT		OUTPUT
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Truth Table

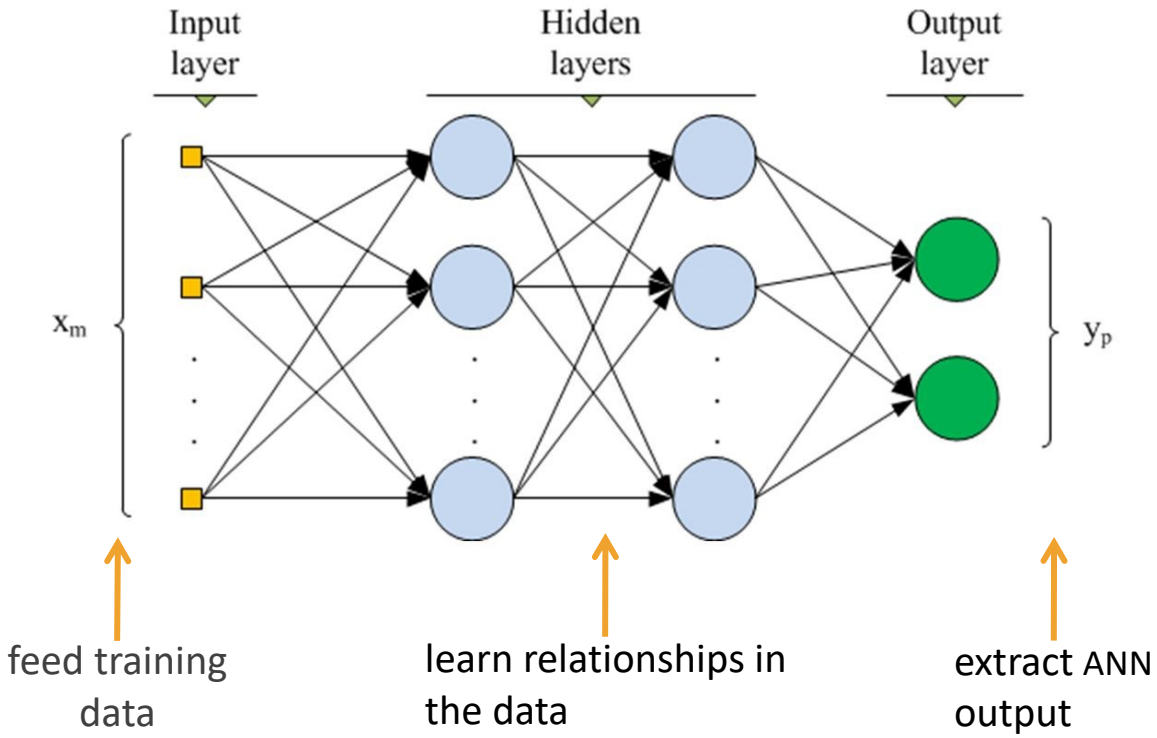


XOR Plot



XOR Cost Function

Feedforward Network (Multi Layer Perceptron)

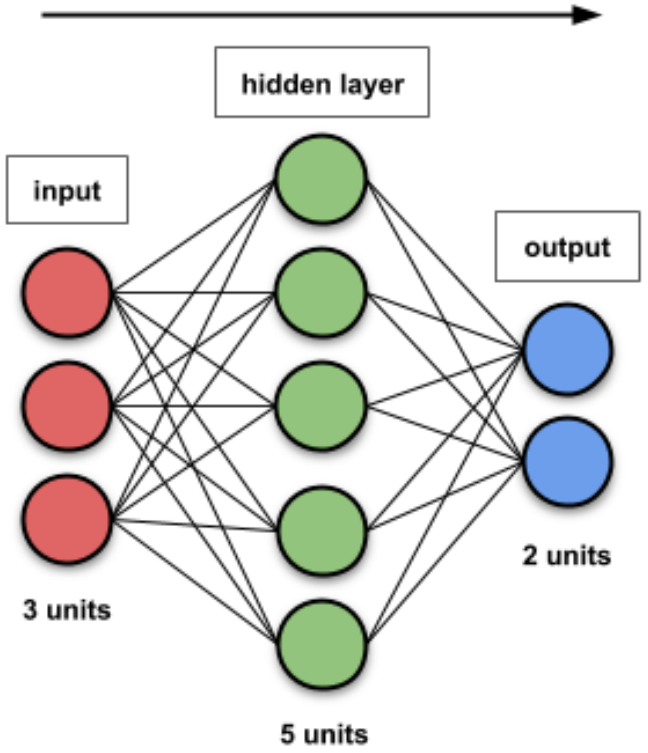


$$f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$$

Feedforward Network - Number of Parameters

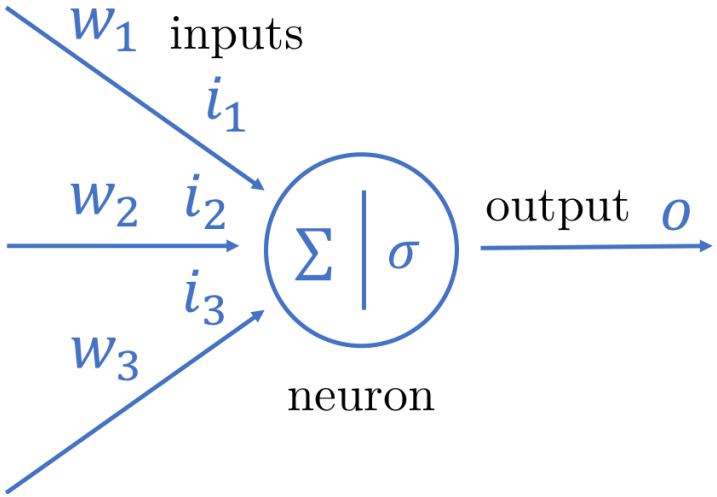
- ***i***, input size
- ***h***, size of hidden layer
- ***o***, output size

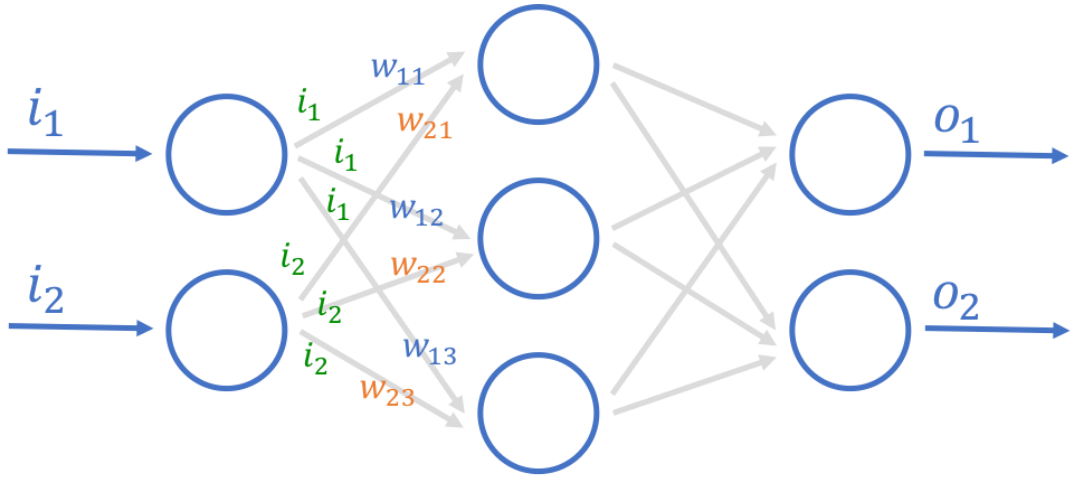
Num of Parameters
= connections between layers + biases in every layer
= (i×h + h×o) + (h+o)



$$Number\ of\ Parameters = (3 \times 5 + 5 \times 2) + (5 + 2) = 32$$

Feedforward Network - Matrix Multiplication

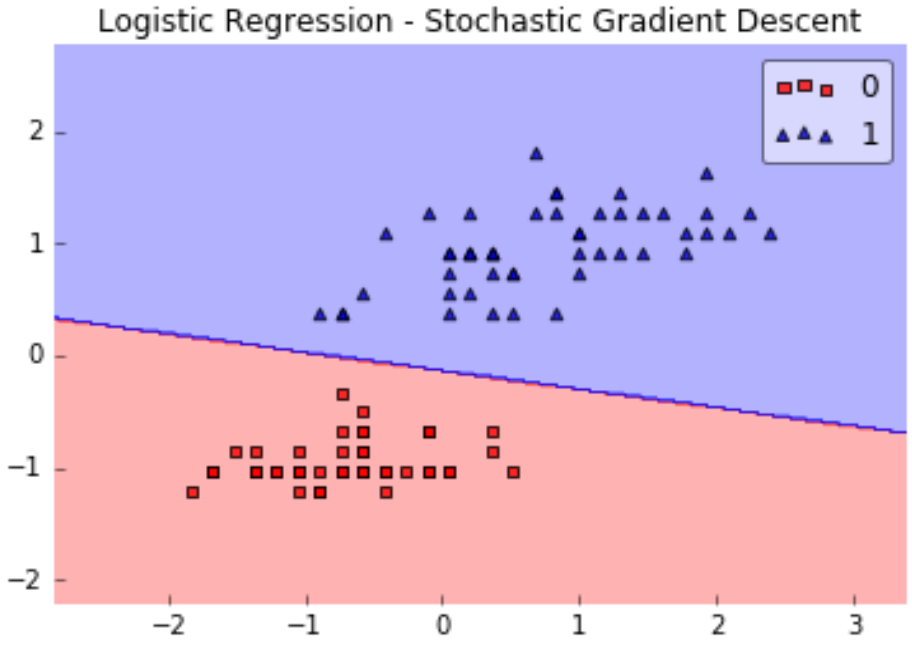




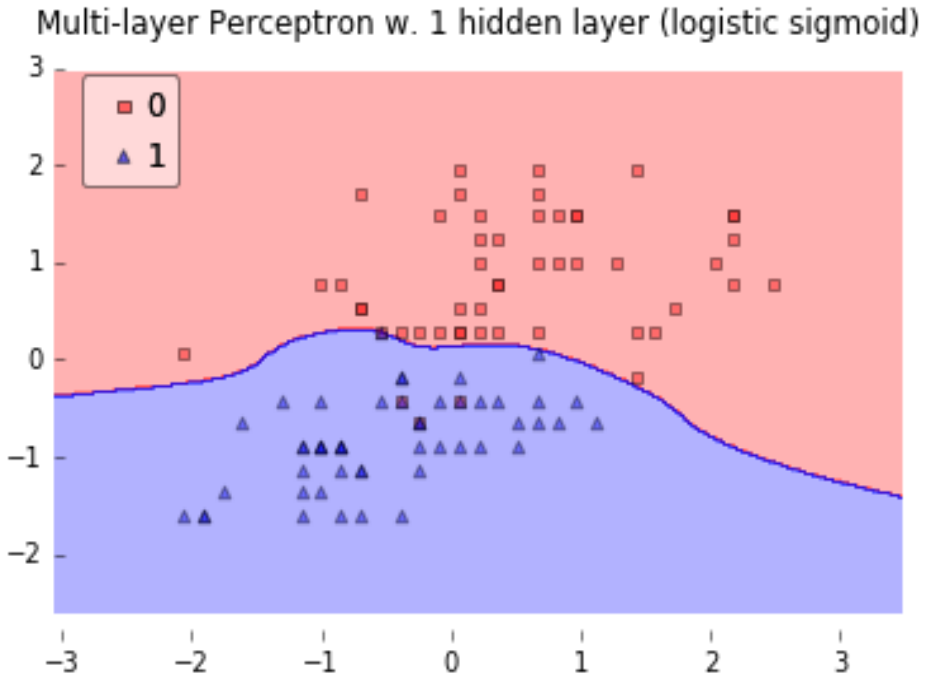
$$\begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} \cdot \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} = \begin{bmatrix} (w_{11} \times i_1) + (w_{21} \times i_2) \\ (w_{12} \times i_1) + (w_{22} \times i_2) \\ (w_{13} \times i_1) + (w_{23} \times i_2) \end{bmatrix}$$

Logistic Regression vs Neural Networks

Logistic Regression



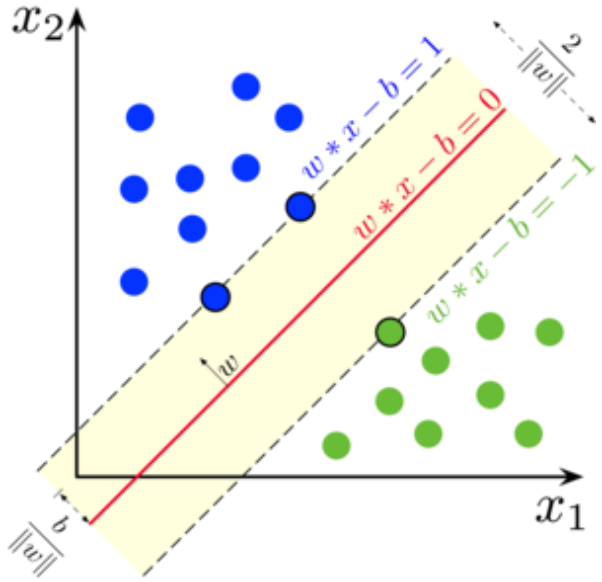
ANN



Support Vector Machines Vs Neural Networks

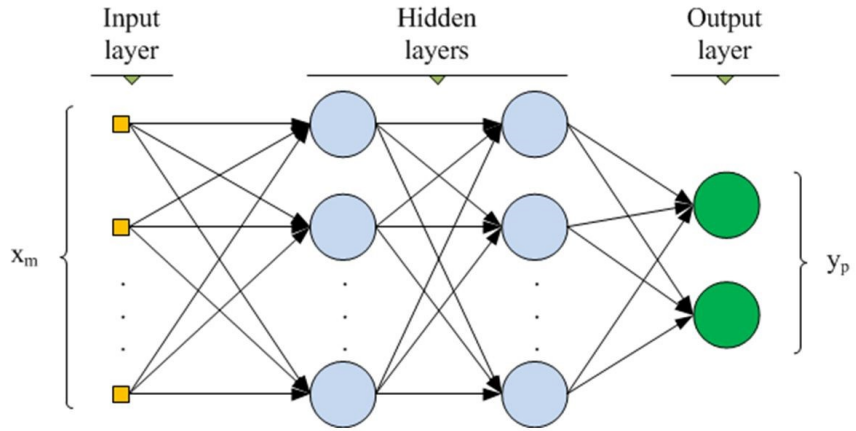
SVM

- SVMs generally train faster on small datasets and do not suffer from local-minima like ANNs
- The number of support vectors can grow as the size of the training set grows..



ANN

- ANNs are universal function approximators and a kernel doesn't have to be guessed
- DNNs are known to perform better than SVMs for large datasets

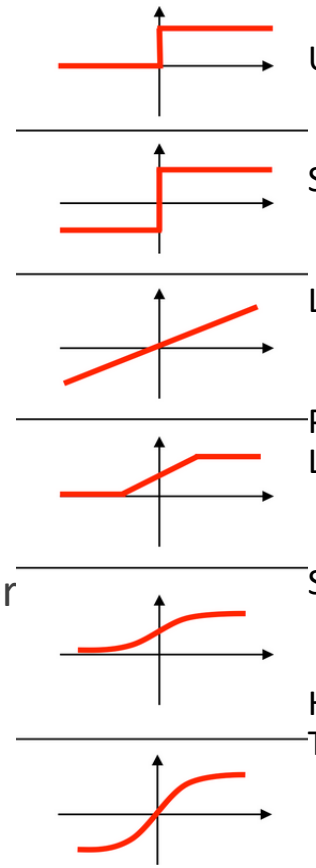


ACTIVATION FUNCTIONS

Sigmoid, Tanh, ReLU, Softmax

Activation Function

- Activation function converts the input signal into a more useful output
- Considerations:
  - Differentiability for gradient calculation
  - Computationally efficient
  - Speed of convergence during optimization
  - Symmetry around the origin
    - because they tend to produce zero-mean inputs to the next layer
  - Vanishing or exploding gradients



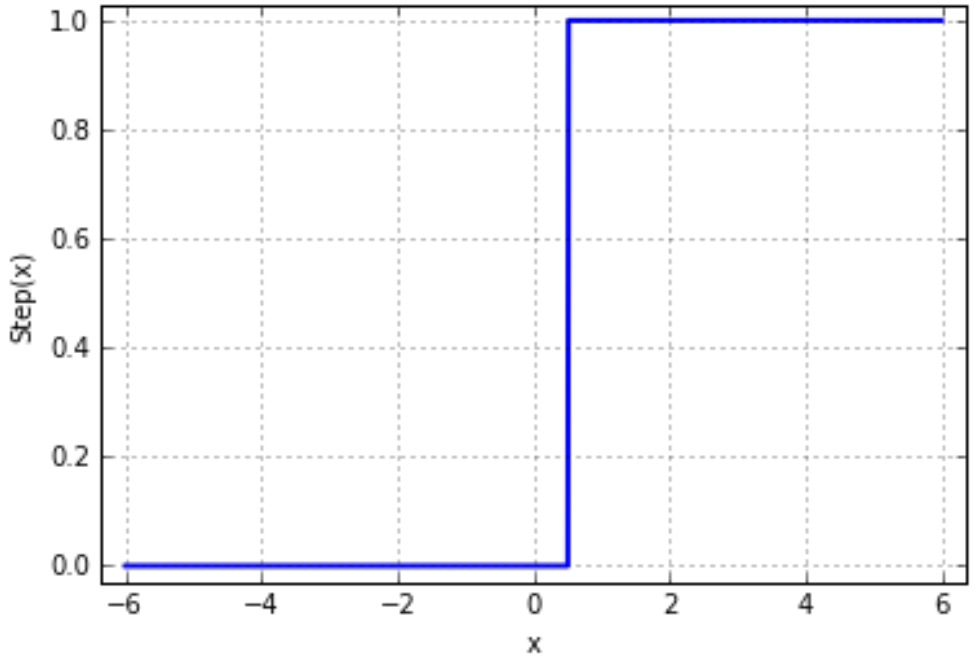
- `tf.nn.relu`
- `tf.nn.relu6`
- `tf.nn.crelu`
- `tf.nn.elu`
- `tf.nn.softplus`
- `tf.nn.softsign`
- `tf.nn.dropout`
- `tf.nn.bias_add`
- `tf.sigmoid`
- `tf.tanh`

Step

- Most basic form of an activation function is a simple binary function that has only two possible results (0 or 1)
- Simple thresholding function

Cons:

- Small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip
- No convergence guaranteed for non-linear decision boundaries

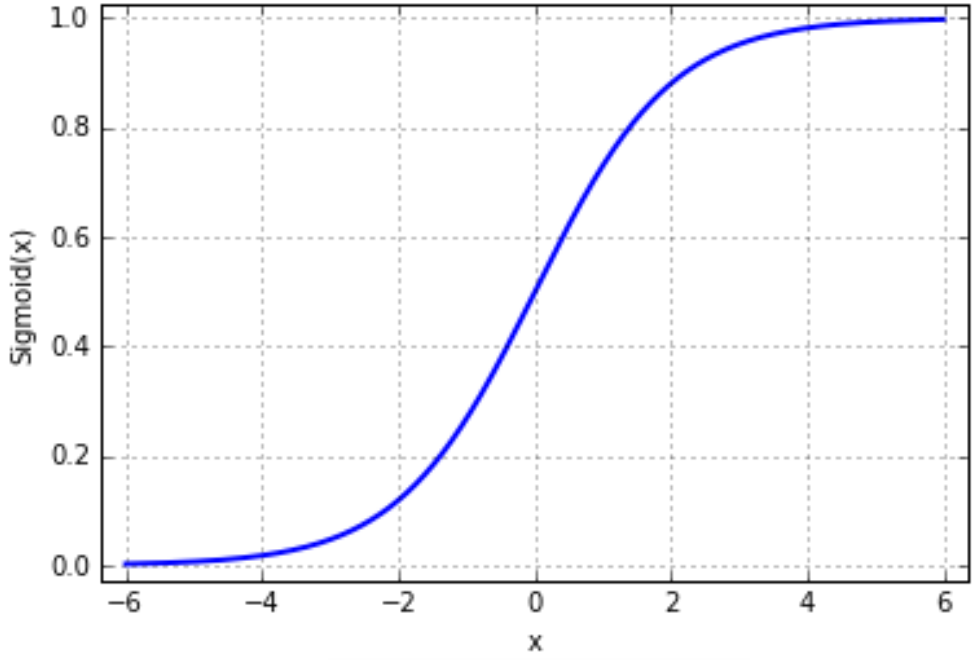


$$f(x) = \begin{cases} 0, & \text{if } x < 0.5 \\ 1, & \text{if } x >= 0.5 \end{cases}$$

Sigmoid

- Special case of the Logistic function
- Squashes the input space from (- ∞, ∞) to (0,1)
- Derivative easy to compute

$$
\begin{aligned}
g'(z) &= \frac{d}{dz}\frac{1}{1+e^{-z}} \\
&= \frac{1}{(1+e^{-z})^2}\left(e^{-z}\right) \\
&= \frac{1}{(1+e^{-z})}\cdot\left(1-\frac{1}{(1+e^{-z})}\right) \\
&= g(z)(1-g(z)).
\end{aligned}
$$



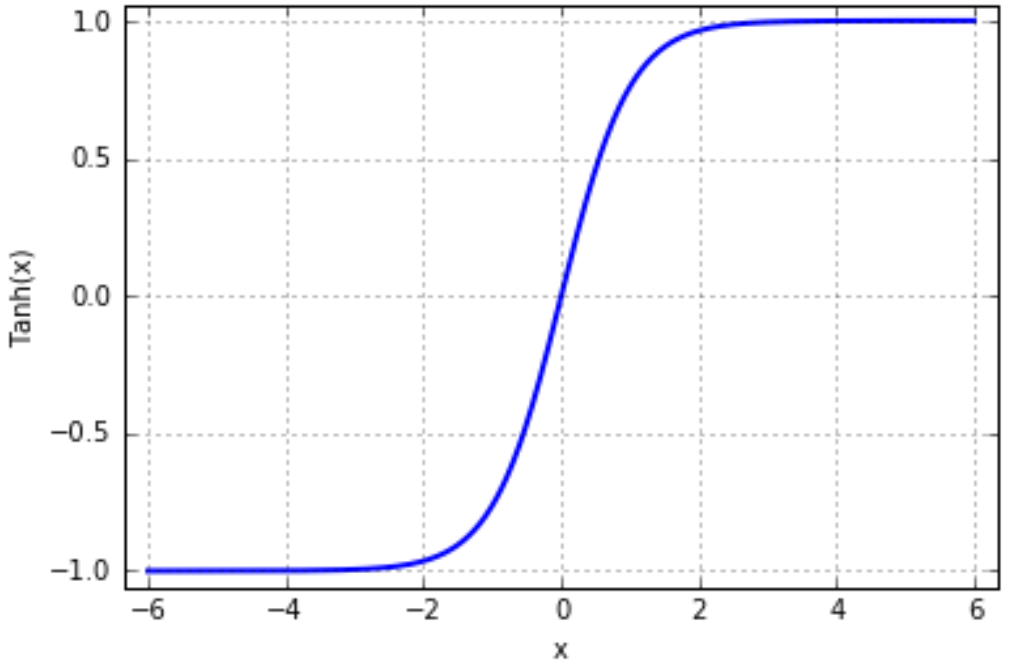
$$f(x) = \frac{1}{1+e^{-x}}$$

Cons:

- Tends to vanish gradient
- Output suffers from symmetric bias

Tanh (Hyperbolic Tangent)

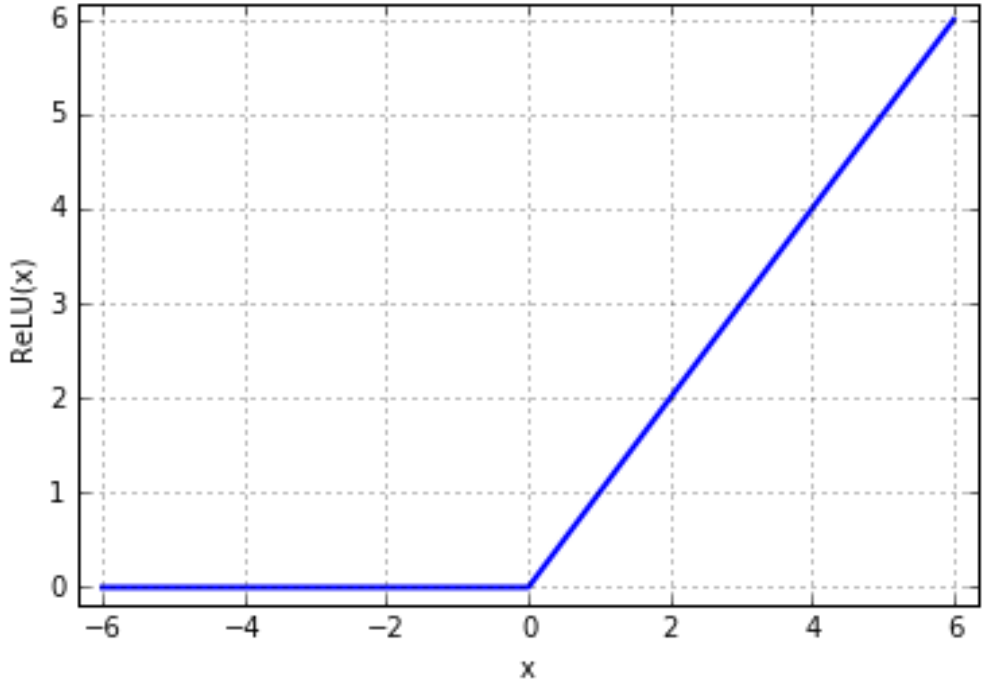
- Outputs are 0 centered in range (-1,1)
- Generally converges faster than sigmoid due to symmetry around zero*



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

ReLU (Rectified Linear Unit)

- Reduced likelihood of vanishing gradient
- Converges faster than Sigmoid or Tanh
- Non-differentiable at the origin
- Popular in deep convolutional neural networks

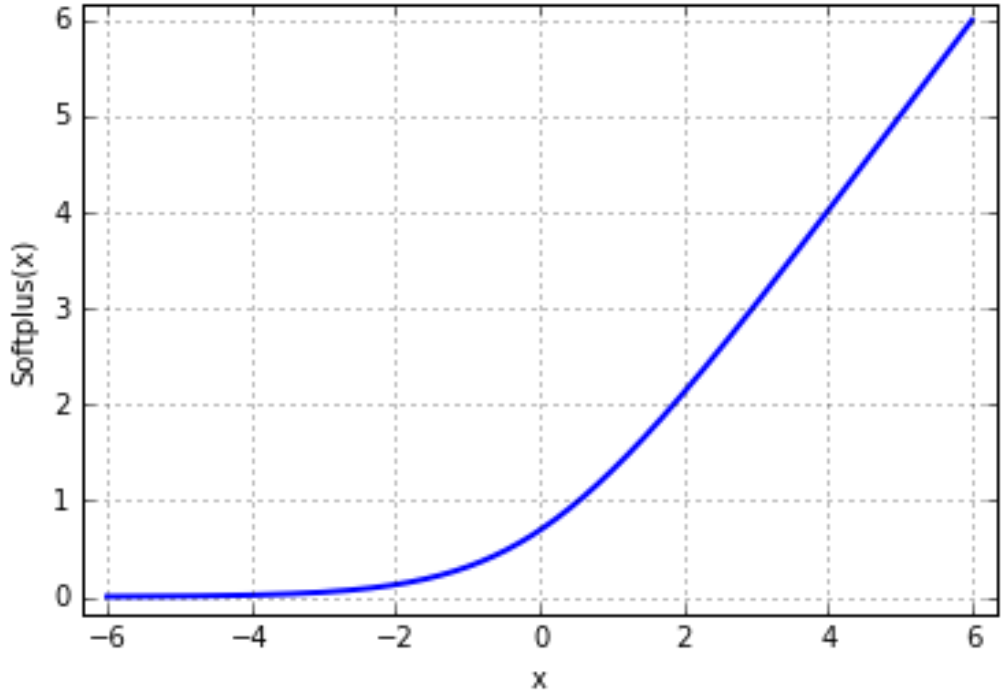


$$f(x) = max(0, x)$$

ReLU Variants - Softplus

- Smooth non-linear version of the ReLU
- Differentiable everywhere
- The derivative of softplus is the logistic function

$$f'(x) = e^x/(e^x + 1) = 1/(1 + e^{-x})$$



$$f(x) = log(1 + e^x)$$

ReLU Variants

Noisy ReLUs
Rectified linear units can be extended to include Gaussian noise, making them noisy ReLUs

$$f(x) = \max(0, x + Y)$$
$$\text{with } Y \sim \mathcal{N}(0, \sigma(x))$$

Leaky ReLUs
Allow a small, non-zero gradient when the unit is not active

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$$

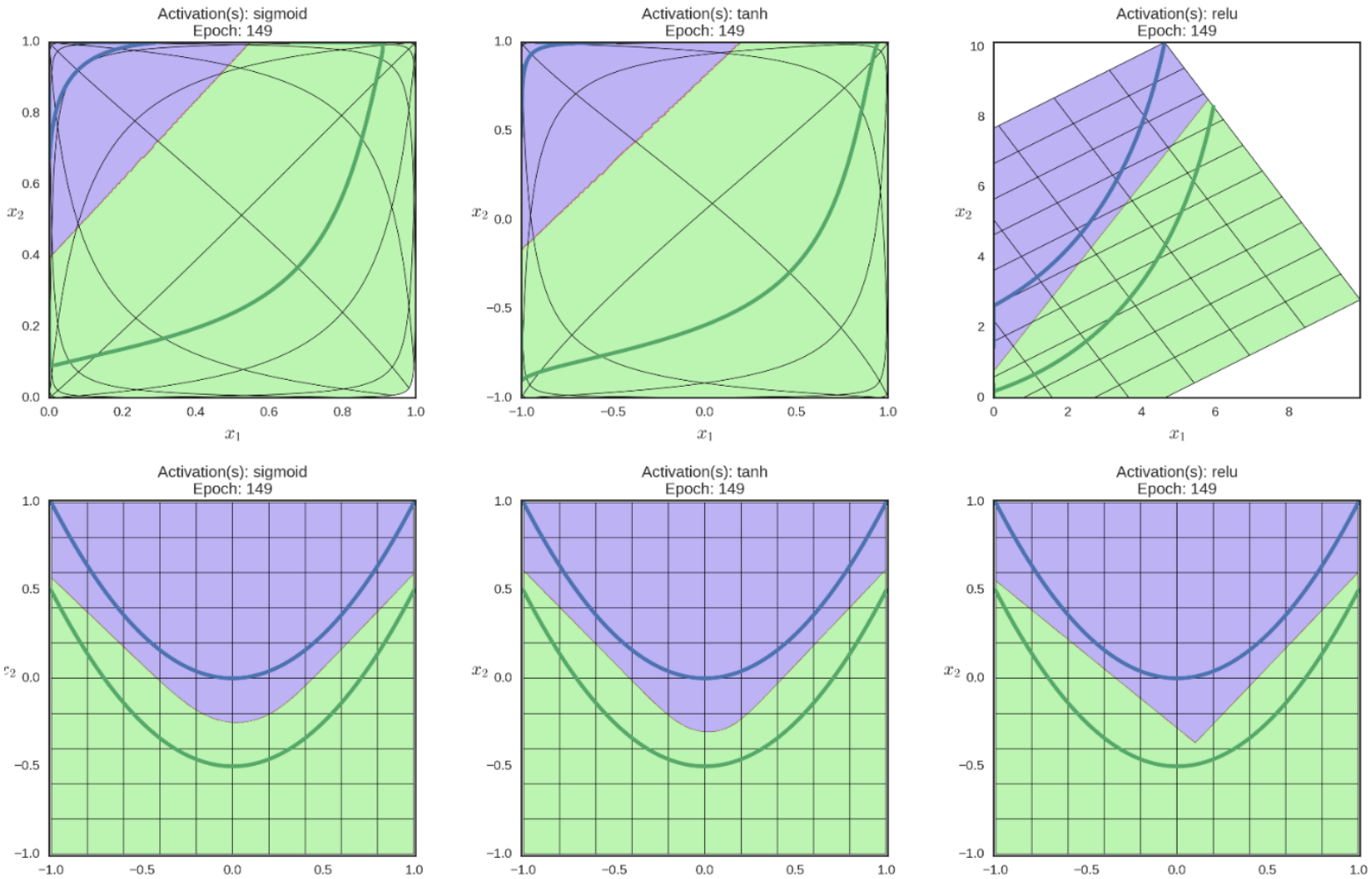
Parametric ReLUs
Make the coefficient of leakage into a parameter that is learned along with the other neural network parameters

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{otherwise} \end{cases}$$

Exponential ReLUs
Make the mean activations closer to zero which speeds up learning. It has been shown that ELUs obtain higher classification accuracy than ReLUs

$$f(x) = \begin{cases} x & \text{if } x >= 0 \\ a(e^x - 1) & \text{otherwise} \end{cases}$$

Activation Functions - Compared



Linear boundaries on transformed feature space (top row), non-linear boundaries on original feature space (bottom row)

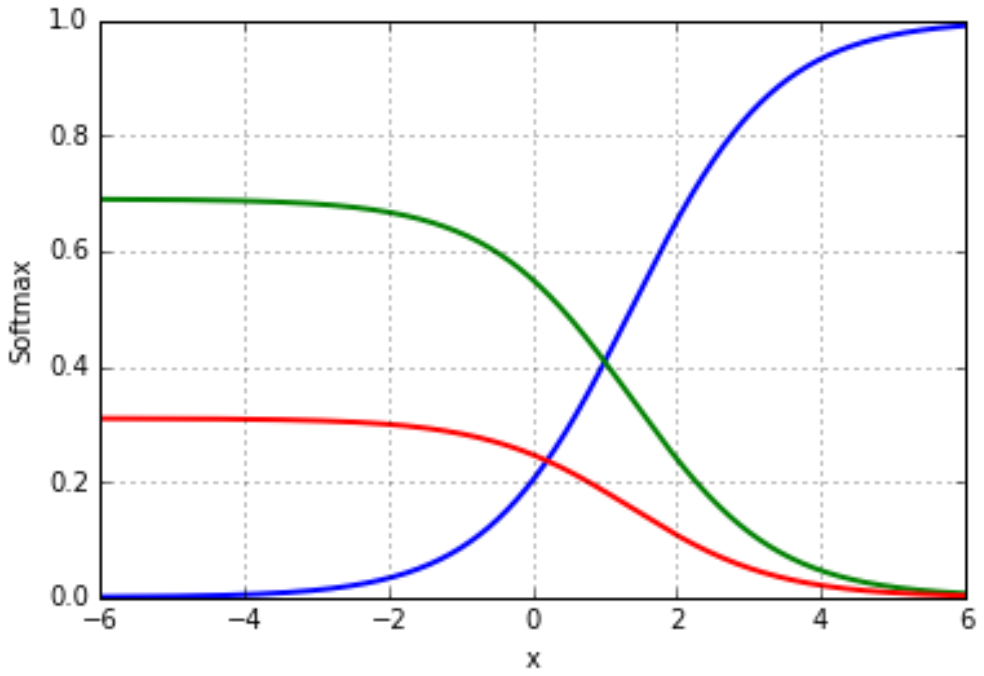
https://towardsdatascience.com/hyper-parameters-in-action-a524bf5bf1c

Softmax

- Normalized Exponential – generalization of logistic function.
- Logistic sigmoid is special case of the softmax for just two classes.
- Used for multiclass classification
- Converts raw scores into probabilities
- Differentiable everywhere

$$\text{evidence}_i = \sum_j W_{i,\,j} x_j + b_i$$

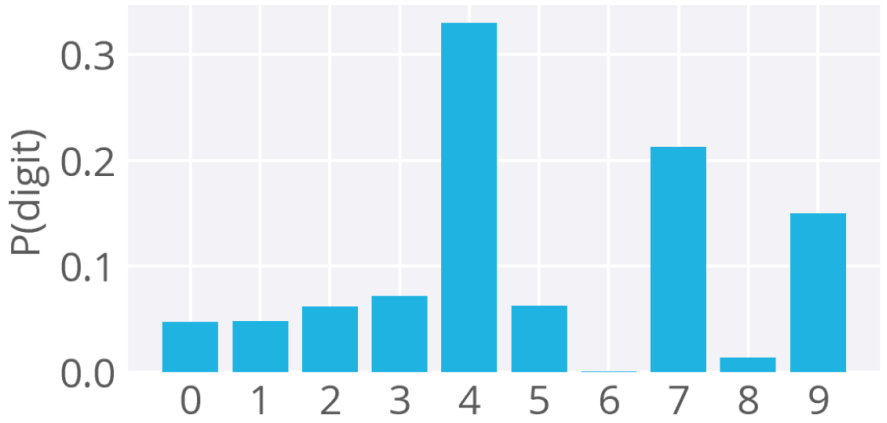
$$y = \text{softmax}(\text{evidence}) = \text{normalize}(\exp(x))$$



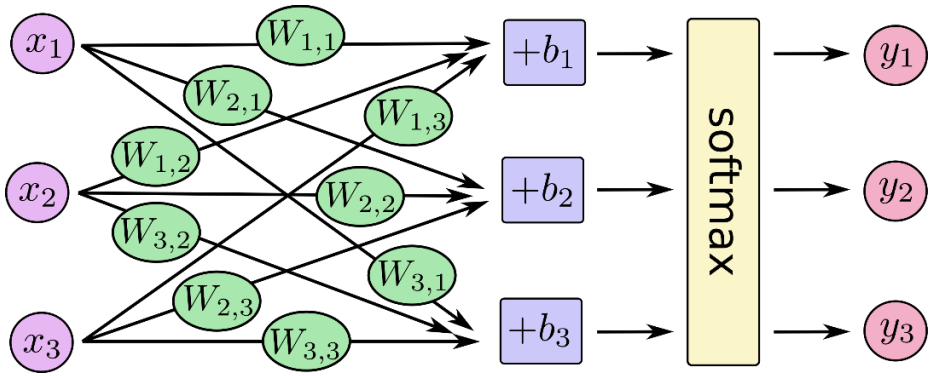
$$\text{softmax}(\mathbf{x})_i = \frac{e^{\mathbf{X}_i}}{\sum_{j=1}^{n} e^{\mathbf{X}_j}}$$

Softmax Example

$$\begin{bmatrix} 1.2 \\ 0.9 \\ 0.4 \end{bmatrix} \rightarrow \boxed{\text{Softmax}} \rightarrow \begin{bmatrix} 0.46 \\ 0.34 \\ 0.20 \end{bmatrix}$$



Softmax Linear Algebra



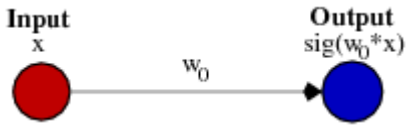
$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{bmatrix} \right)$$

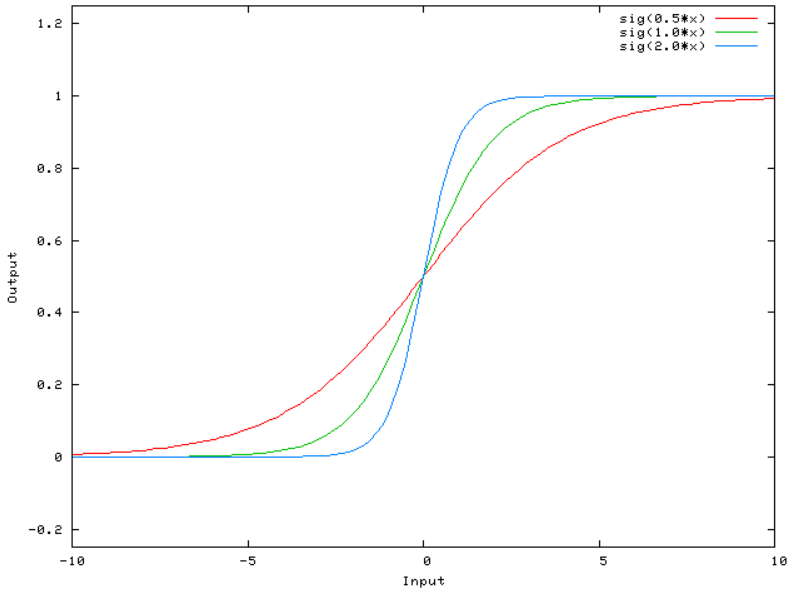
$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

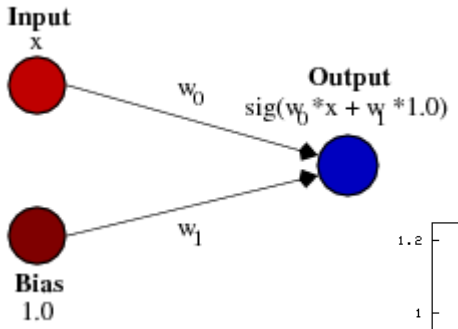
https://www.tensorflow.org/get_started/mnist/beginners

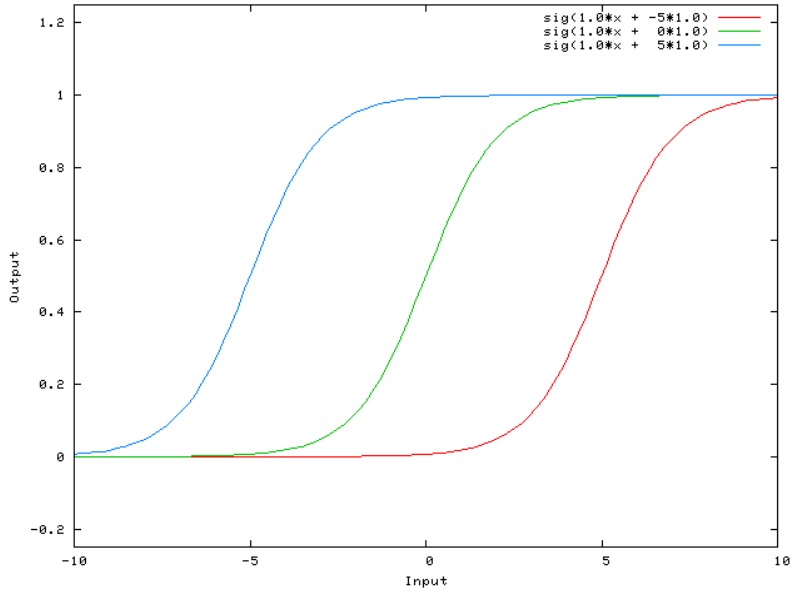
Effect of Weight vs Bias

- Changing the weight w_0 essentially changes the "steepness" of the sigmoid.
- However, to be able to shift the entire curve to the right/left use a bias w_1









ANN TRAINING

Feedforward, Backpropagation

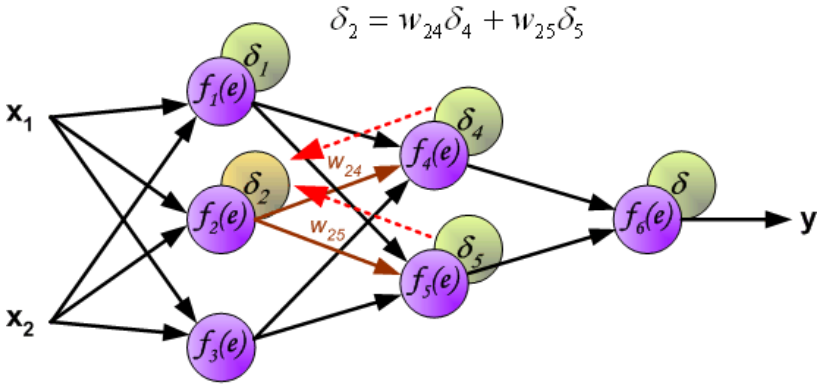
Training Algorithms

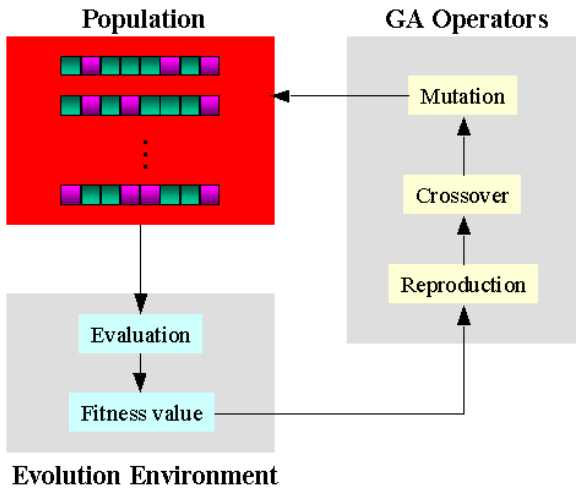
Neural Network Training

Backpropagation

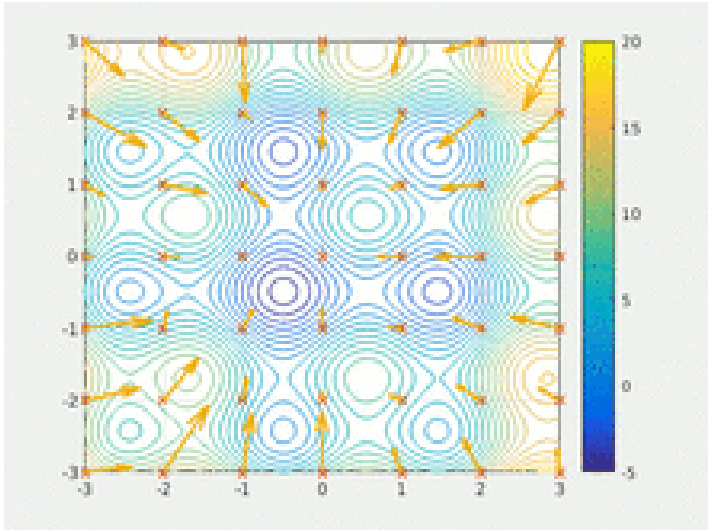
Genetic Algorithms

Particle Swarm Optimization





Genetic Algorithm Evolution Flow



ANN Learning Problem

- Learn the network weights (parameters) which minimizes the loss function
- Gradient Descent weight update equation:

$$w_{i+1} = w_i - \eta \frac{\partial E}{\partial w_i} + \alpha \Delta w_i - \lambda \eta w_i$$

learning rate momentum weight decay

Calculus Chain Rule

Calculating the derivative of nested functions

$$y = e^{\sin x^2}$$

$$\begin{aligned} y &= f(u) = e^u, \\ u &= g(v) = \sin v, \\ v &= h(x) = x^2. \end{aligned}$$

$$\begin{aligned} \frac{dy}{du} &= f'(u) = e^u, \\ \frac{du}{dv} &= g'(v) = \cos v, \\ \frac{dv}{dx} &= h'(x) = 2x. \end{aligned}$$

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx}.$$

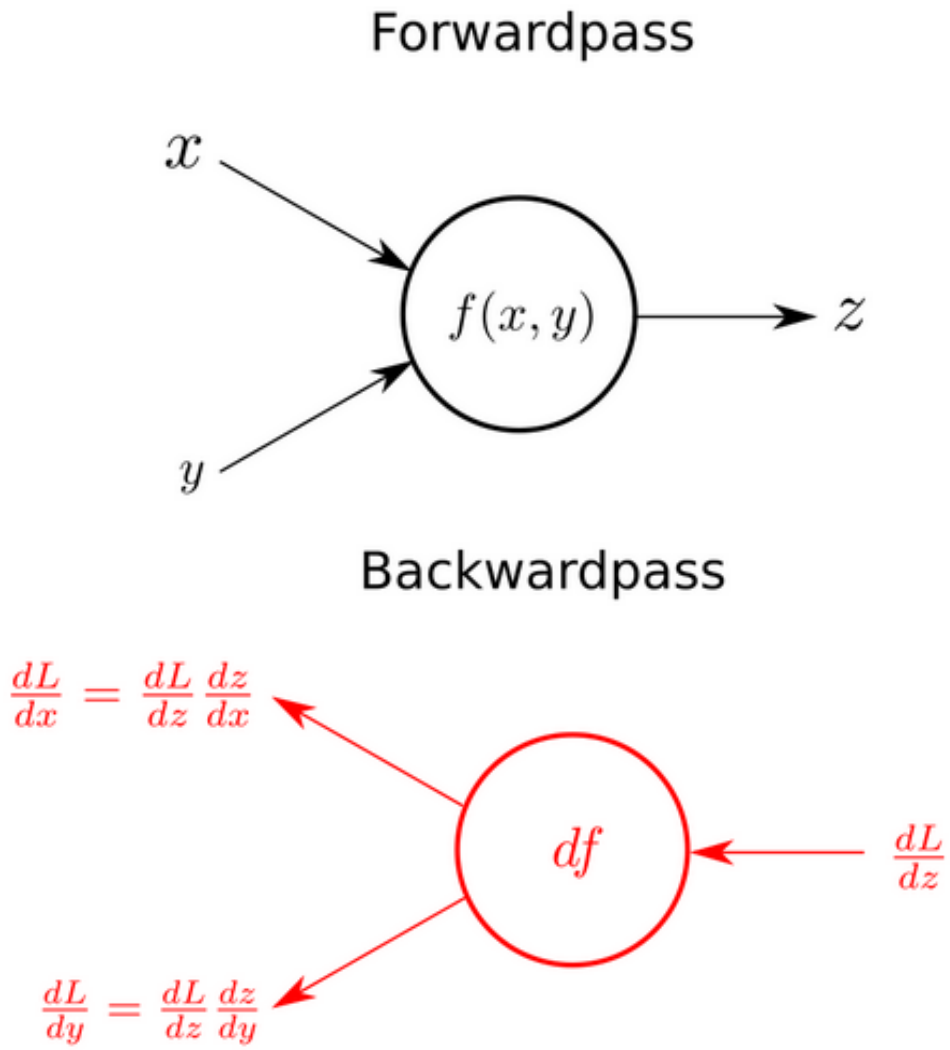
$$\frac{dy}{dx} = e^{\sin x^2} \cdot \cos x^2 \cdot 2x.$$

https://en.wikipedia.org/wiki/Chain_rule

Backpropagation Algorithm

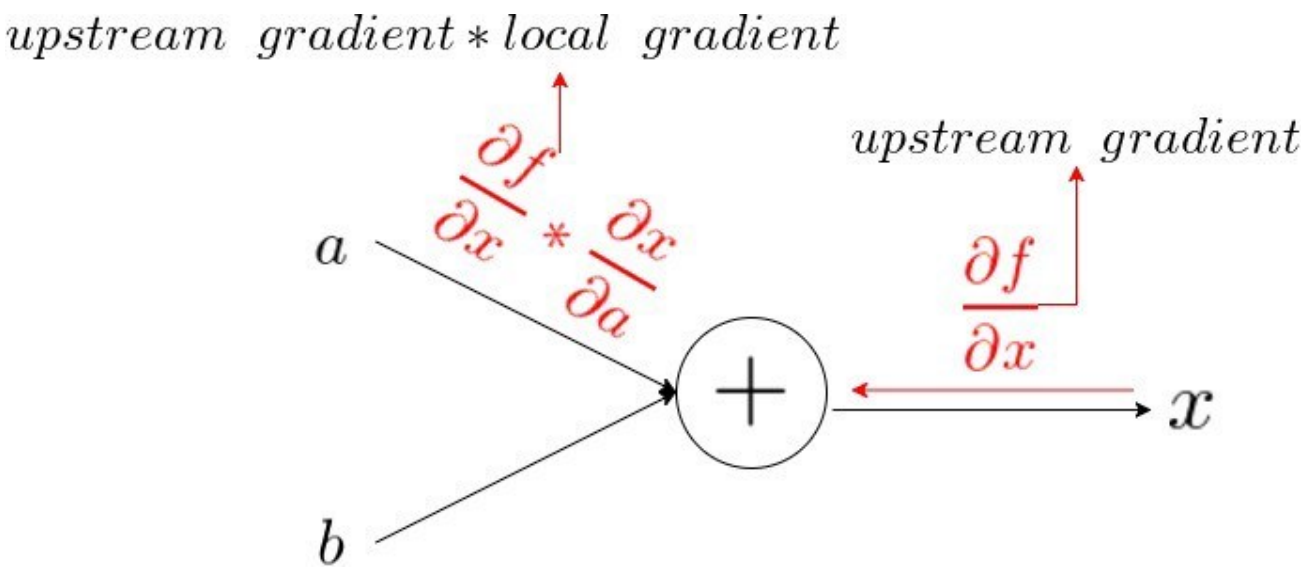
Backpropagation (or reverse differentiation) consists of :

1. Gradient Descent
   - Minima of the error function in weight space
2. Chain Rule
   - Since the decision function of the ANN is a function of functions, we use the chain rule to compute its gradient

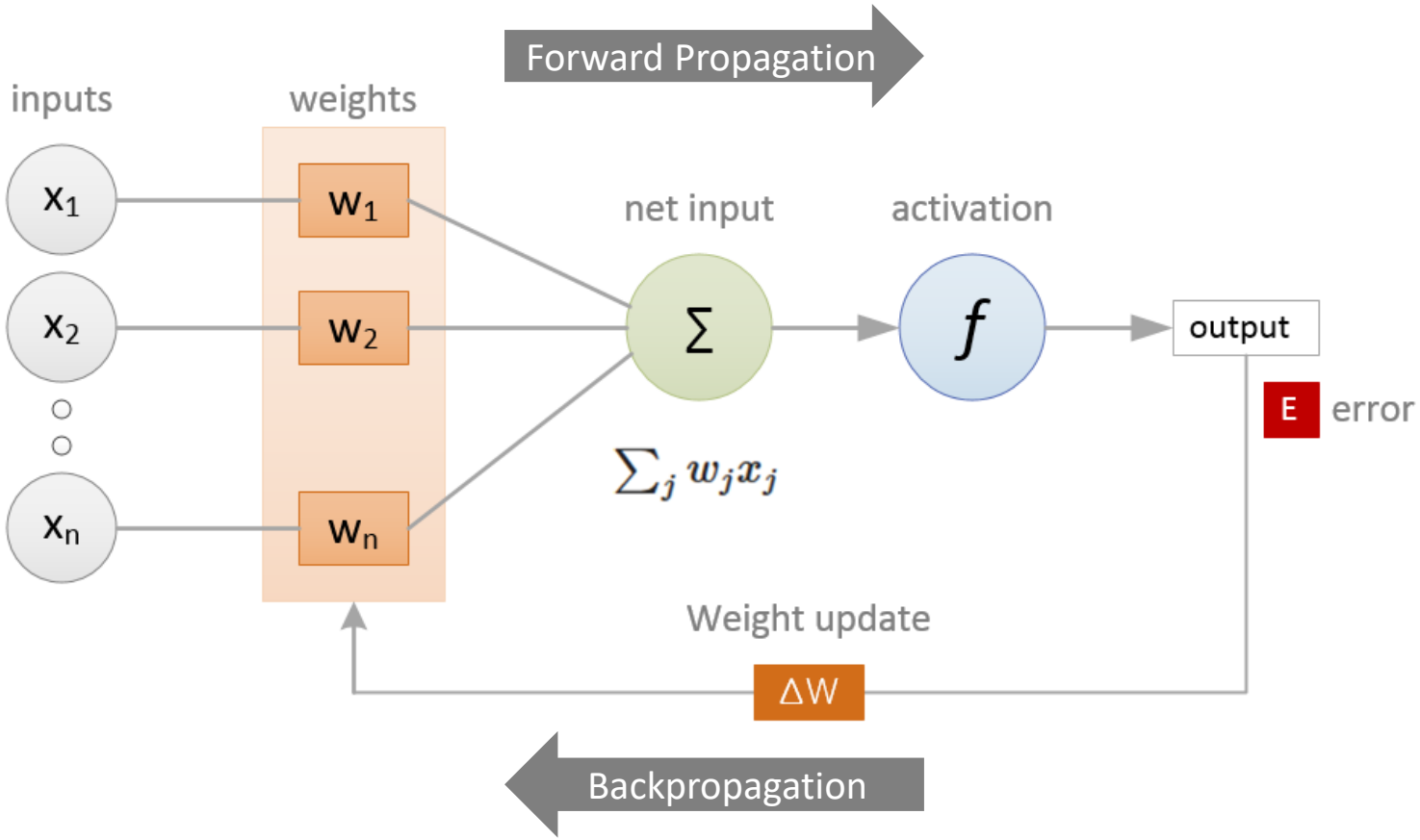


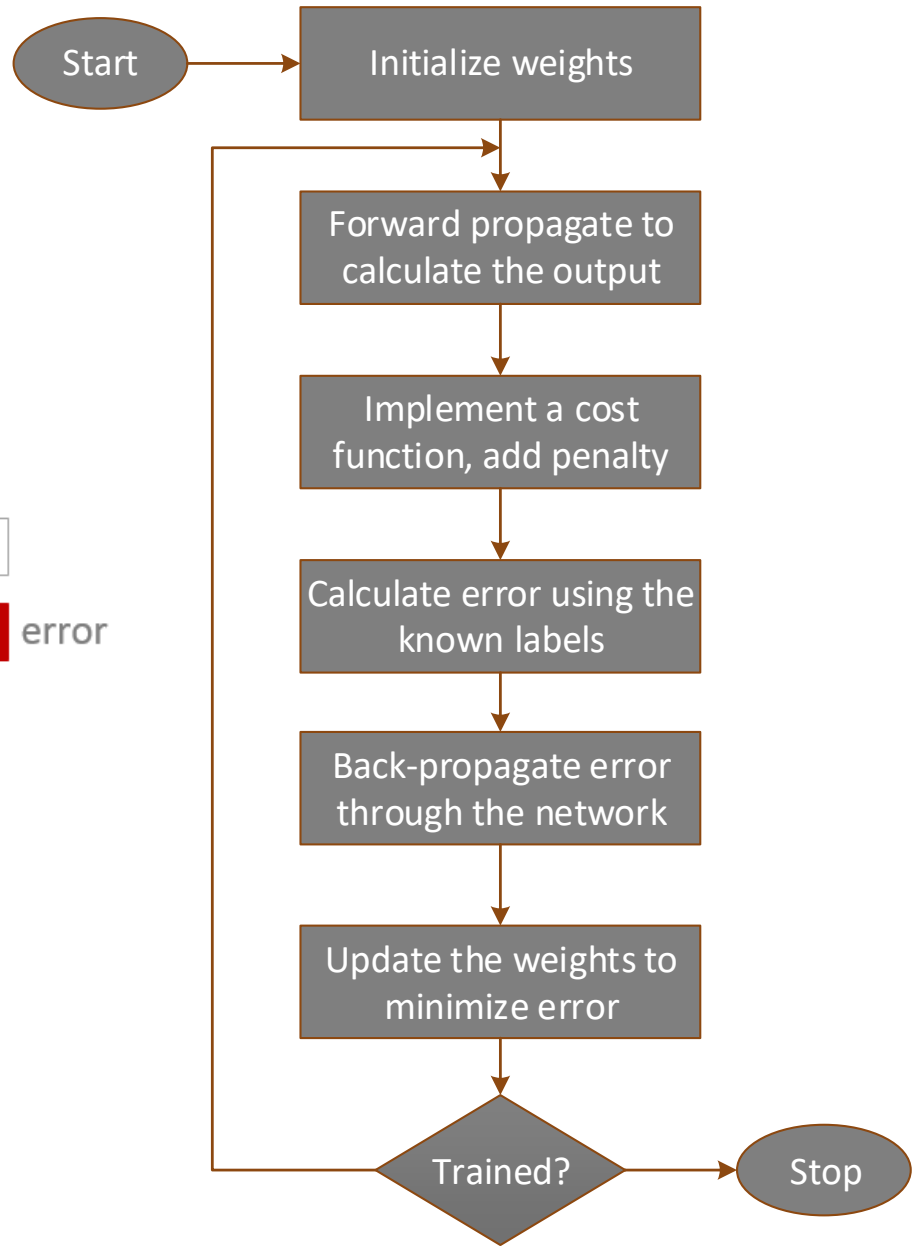
Backpropagation: Compute Graph

- Every node in a computational graph can compute two things - the output of the node and the local gradient of the node without even being aware of the rest of the graph.



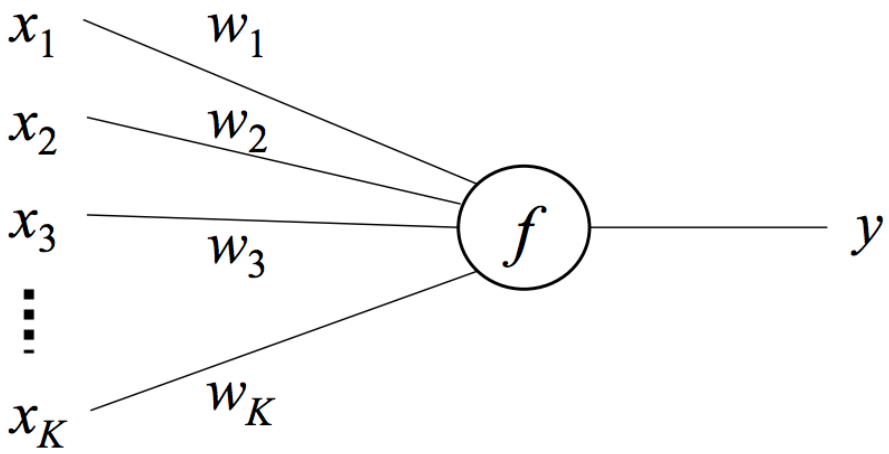
Backpropagation: Algorithm





Backpropagation

Single Neuron



Target Output t

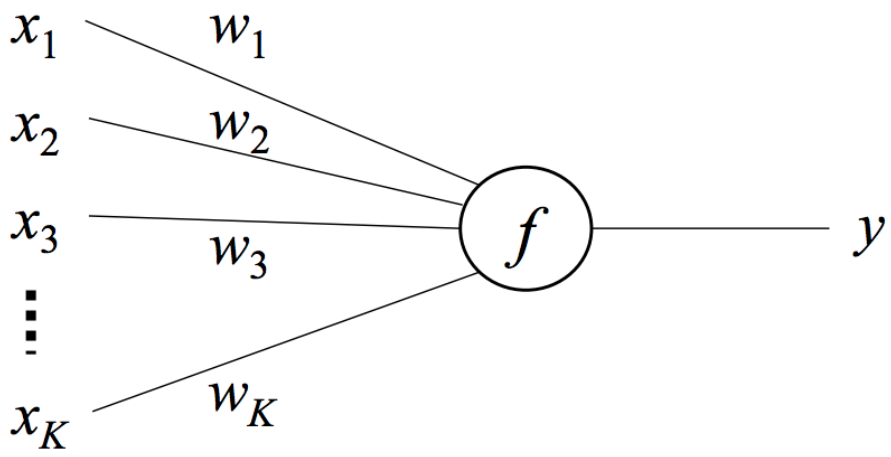
Predicted Output $$y = f(\sum_{i=0}^{K} w_i x_i) = f(\mathbf{w}^\mathbf{T}\mathbf{x})$$

Error $$E = \frac{1}{2}(t-y)^2 = \frac{1}{2}(t - f(\mathbf{w}^\mathbf{T}\mathbf{x}))^2$$

Activation $$f(u) = \frac{1}{1+e^u} \Rightarrow y = \frac{1}{1+e^{\mathbf{w}^\mathbf{T}\mathbf{x}}}$$

Backpropagation

Single Neuron



Delta Rule : Degree to which a slight change in a weight causes a slight change in the error :

$$\Delta W \propto -\frac{dE}{dW} = \frac{dE}{dActivation} \cdot \frac{dActivation}{dW}$$

Compute gradient of E with respect to an arbitrary element of weight vector w_i

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial w_i} \\ &= (y - t) \cdot y(1 - y) \cdot x_i\end{aligned}$$

Iteratively update weight parameters in the direction of the gradient of the loss function until minimum reached

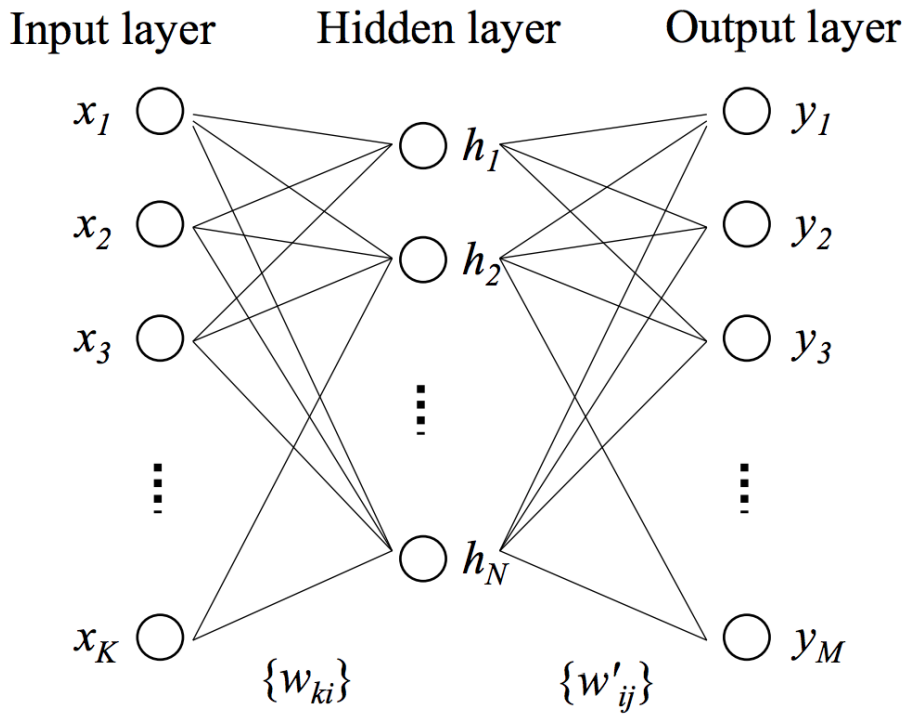
$$\mathbf{w}^{new} = \mathbf{w}^{old} - \eta \cdot (y - t) \cdot y(1 - y) \cdot \mathbf{x}$$

where η>0 is the step size

(**x**,y) data points are sequentially fed into this update equation until the weights **w** converge to their optimal value

Backpropagation

3 Layer Neural Network



1. Output of arbitrary neuron h_i

$$h_i = f(u_i) = f(\sum_{k=1}^{K} w_{ki} x_k)$$

2. Output of arbitrary neuron y_j

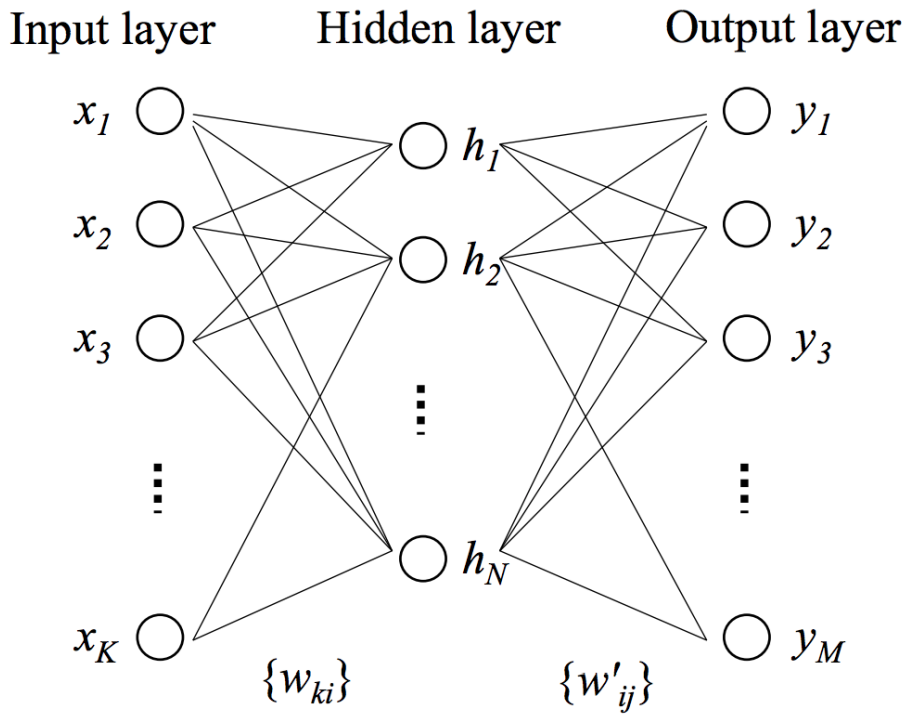
$$y_j = f(u'_j) = f(\sum_{i=1}^{N} w'_{ij} h_i)$$

3. Same cost function but summed up over all elements in output layer

$$E = \frac{1}{2} \sum_{j=1}^{M} (y_j - t_j)^2$$

Backpropagation

Update Hidden to Output Weights



hidden-to-output weights

4. Derivative of E w.r.t to w'_{ij}

$$\frac{\partial E}{\partial w'_{ij}} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial u'_j} \cdot \frac{\partial u'_j}{\partial w'_{ij}}$$

5. Derivative of E w.r.t to y_j

$$\frac{\partial E}{\partial y_j} = y_j - t_j \quad \text{where } y_j = f(u'_j)$$

6. Derivative of the logistic function

$$\frac{\partial y_j}{\partial u'_j} = y_j(1 - y_j)$$

7. Derivative of $u'_j = \sum_{i=1}^{N} w'_{ij} h_i$ w.r.t w'_{ij}

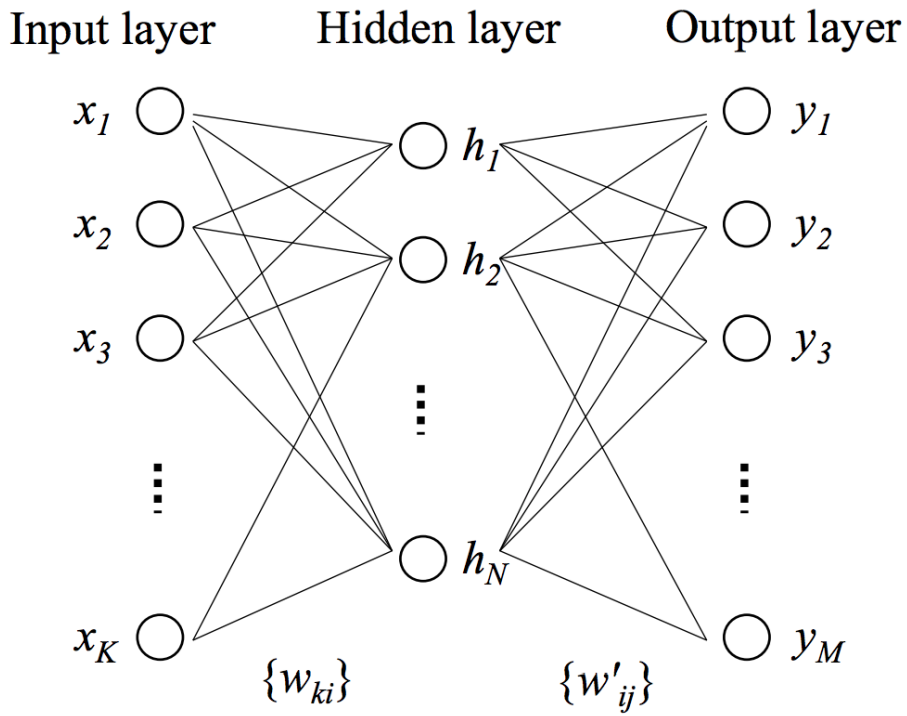
$$\frac{\partial E}{\partial w'_{ij}} = (y_j - t_j) \cdot y_j(1 - y_j) \cdot h_i$$

8. Weight update for w'_{ij}

$$w'^{new}_{ij} = w'^{old}_{ij} - \eta \cdot (y_j - t_j) \cdot y_j(1 - y_j) \cdot h_i$$

Backpropagation

Update Input to Hidden Weights



input-to-hidden weights

9. Derivative of E w.r.t to w_{ki} using chain rule

$$\frac{\partial E}{\partial w_{ki}} = \sum_{j=1}^{M}\left(\frac{\partial E}{\partial y_j}\cdot\frac{\partial y_j}{\partial u'_j}\cdot\frac{\partial u'_j}{\partial h_i}\right)\cdot\frac{\partial h_i}{\partial u_i}\cdot\frac{\partial u_i}{\partial w_{ki}}$$

10. Already computed that

$$\frac{\partial E}{\partial y_j}\cdot\frac{\partial y_j}{\partial u'_j} = (y_j - t_j)\cdot y_j(1-y_j)$$

11. Remaining derivatives

$$\frac{\partial u'_j}{\partial h_i} = \frac{\partial \sum_{i=1}^{N} w'_{ij}h_i}{\partial h_i} = w'_{ij}$$

$$\frac{\partial u_i}{\partial w_{ki}} = \frac{\partial \sum_{k=1}^{K} w_{ki}x_k}{\partial w_{ki}} = x_k$$

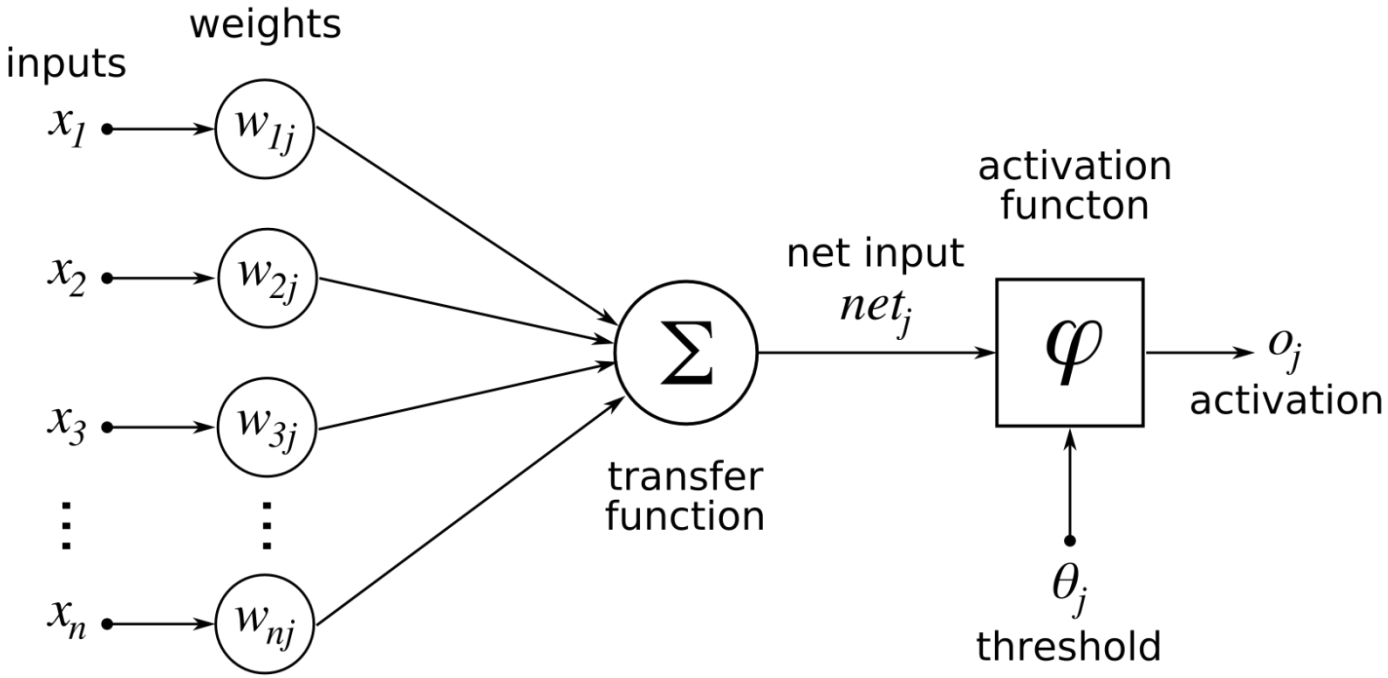
$$\frac{\partial E}{\partial w_{ki}} = \sum_{j=1}^{M}[(y_j - t_j)\cdot y_j(1-y_j)\cdot w'_{ij}]\cdot h_i(1-h_i)\cdot x_k$$

12. Weight update for w_{ki}

$$w_{ki}^{new} = w_{ki}^{old} - \eta\cdot\sum_{j=1}^{M}[(y_j - t_j)\cdot y_j(1-y_j)\cdot w'_{ij}]\cdot h_i(1-h_i)\cdot x_k$$

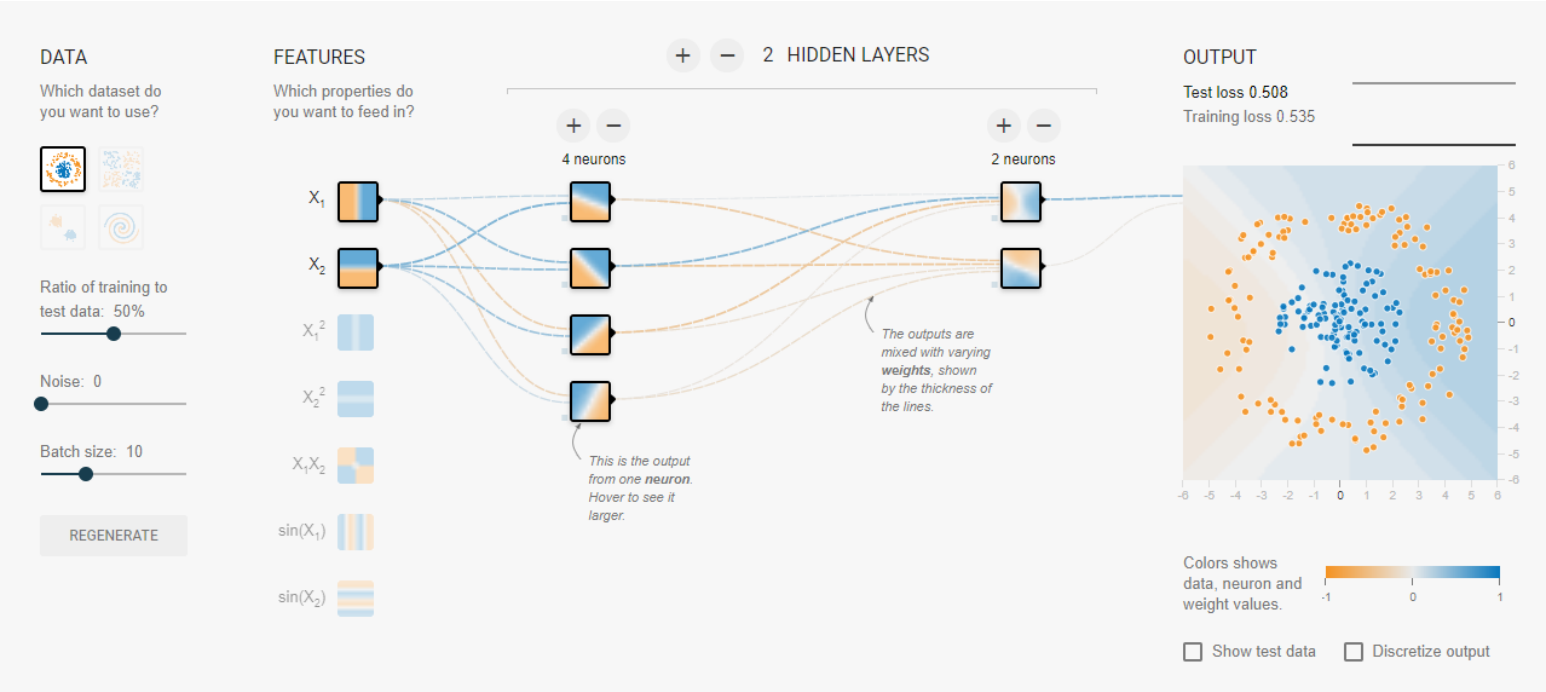
ANN Hyperparameter Tuning

- Network Architecture
  - No of hidden layers
  - No of hidden units
  - Activation Function
- Optimization
  - Weights Initialization
  - Learning Rate
  - Loss Function
  - Batch size
  - Momentum
  - Number of epochs



Exercises

1. Activation functions
2. Tensorflow Playground
3. ANN
   - Basic Neural Network
   - Sine Function, Logic gates
   - ANN Vs Logistic
   - IRIS Classification
4. Javascript
   - http://cs.stanford.edu/people/karpathy/convnetjs/



http://playground.tensorflow.org