Deep Learning

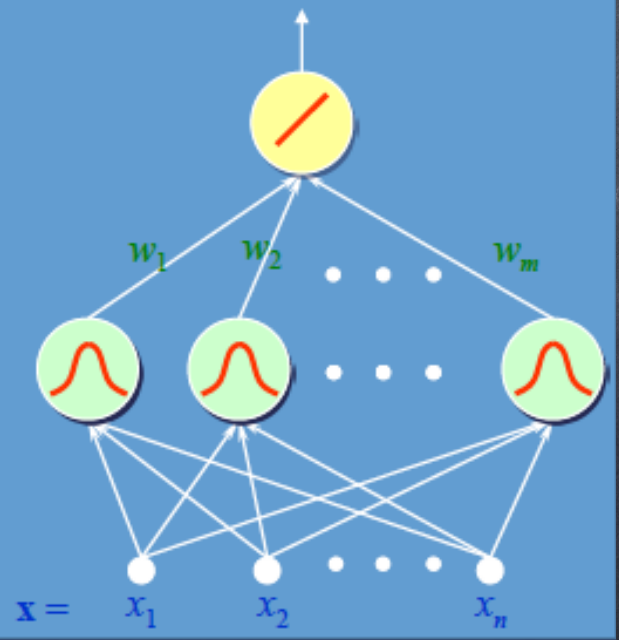
Mayank Vatsa

Training Neural Network

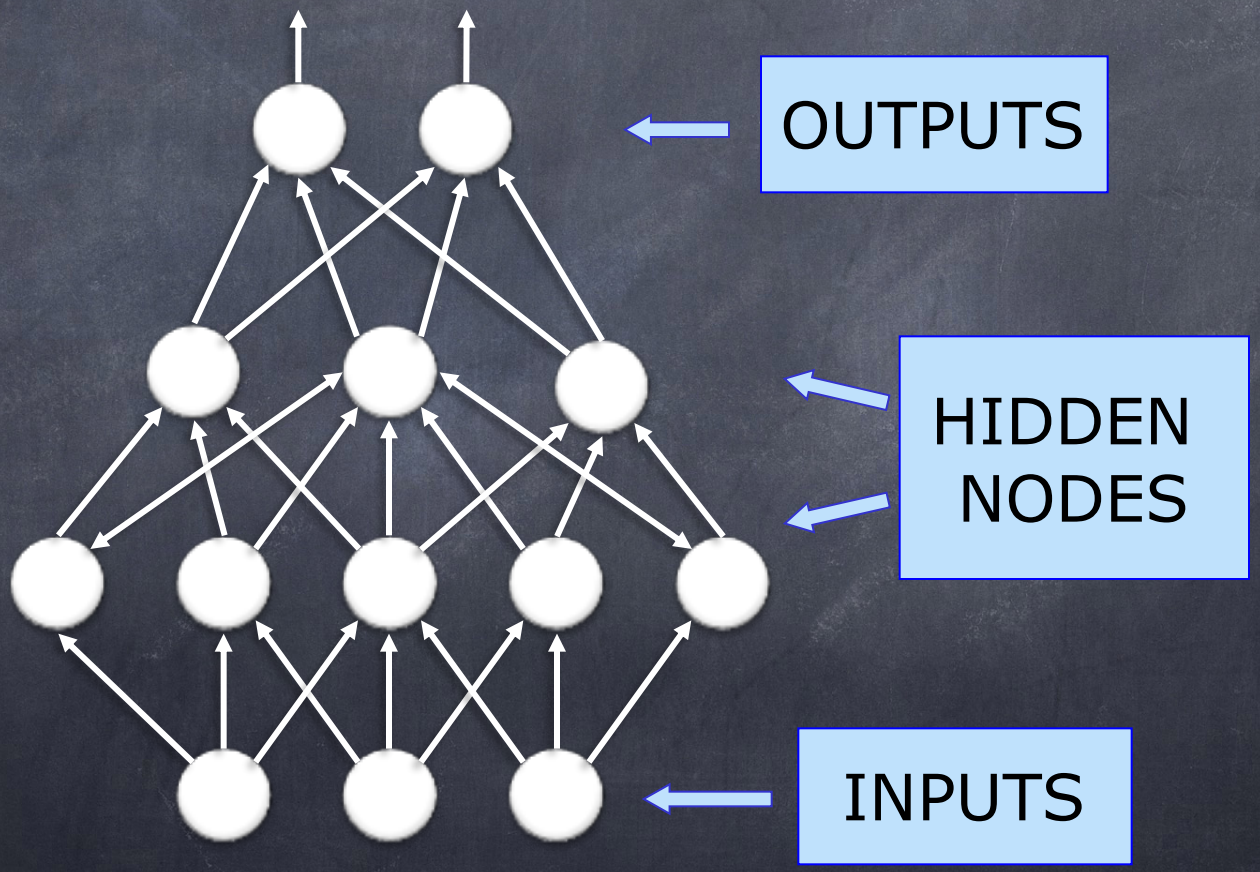
Training set $\mathcal{T} = \left\{ \left(\mathbf{x}^{(k)}, y^{(k)} \right) \right\}_{k=1}^{p}$

Goal $y^{(k)} \approx f\left(\mathbf{x}^{(k)} \right)$ for all k

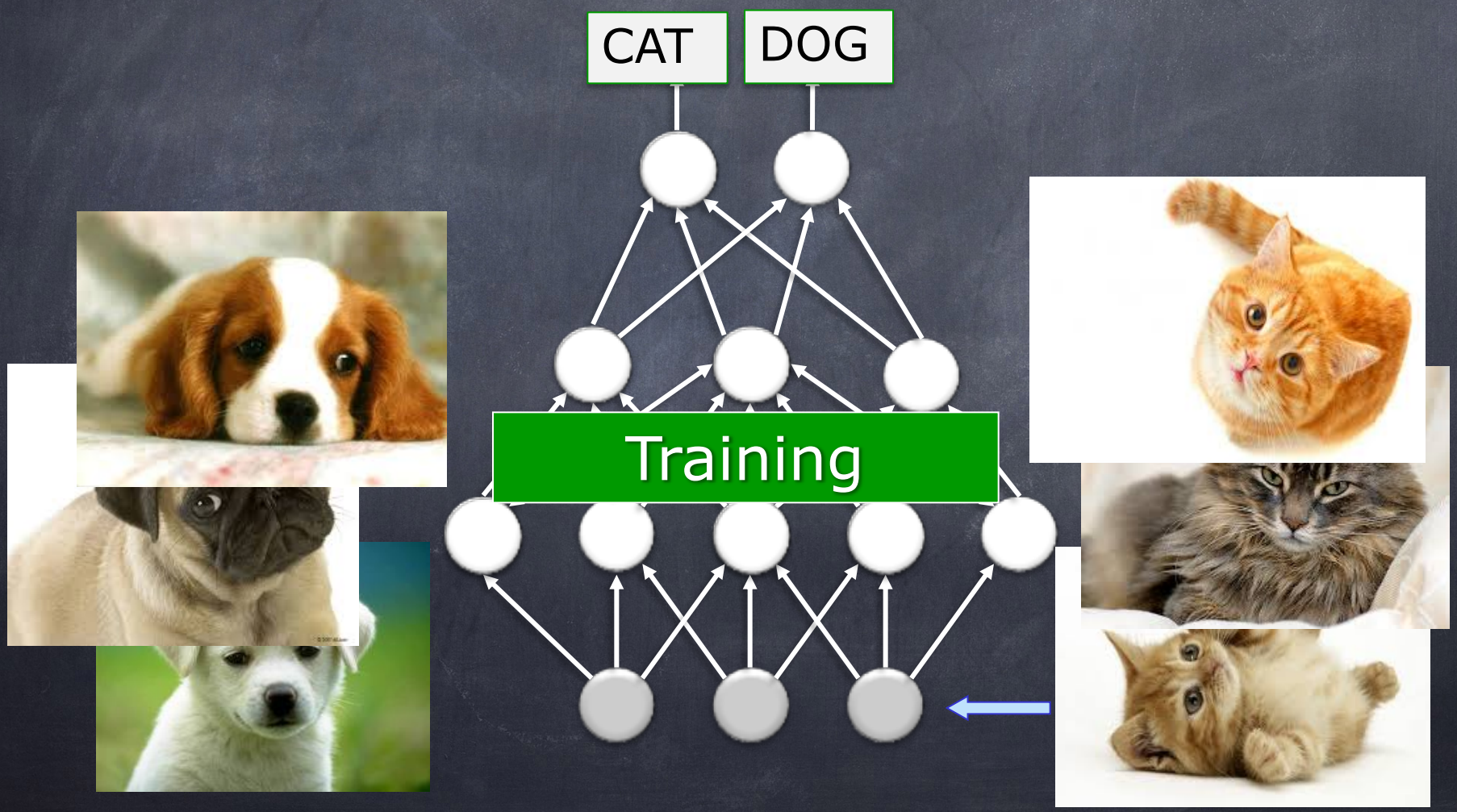
$$\min E = \frac{1}{2} \sum_{k=1}^{p} \left[y^{(k)} - f\left(\mathbf{x}^{(k)} \right) \right]^2$$

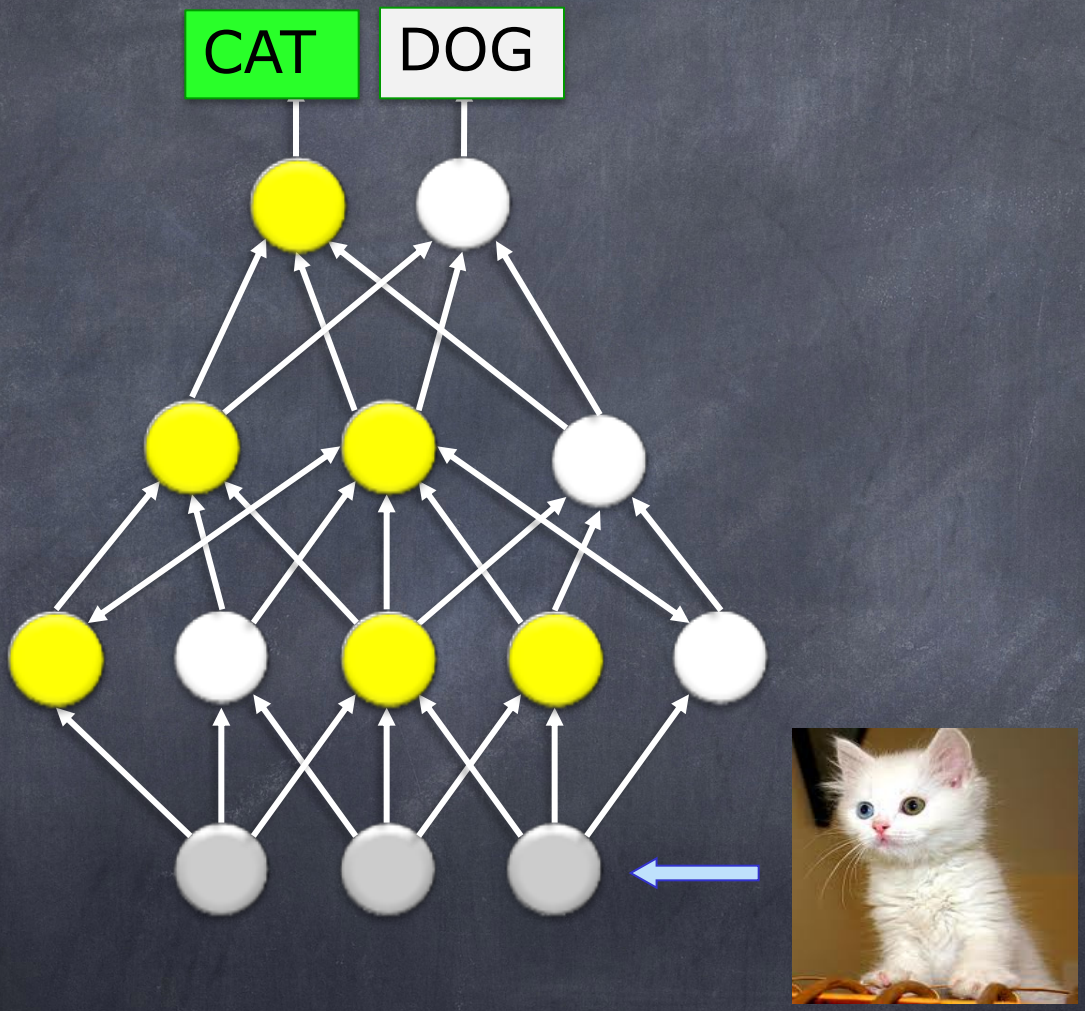


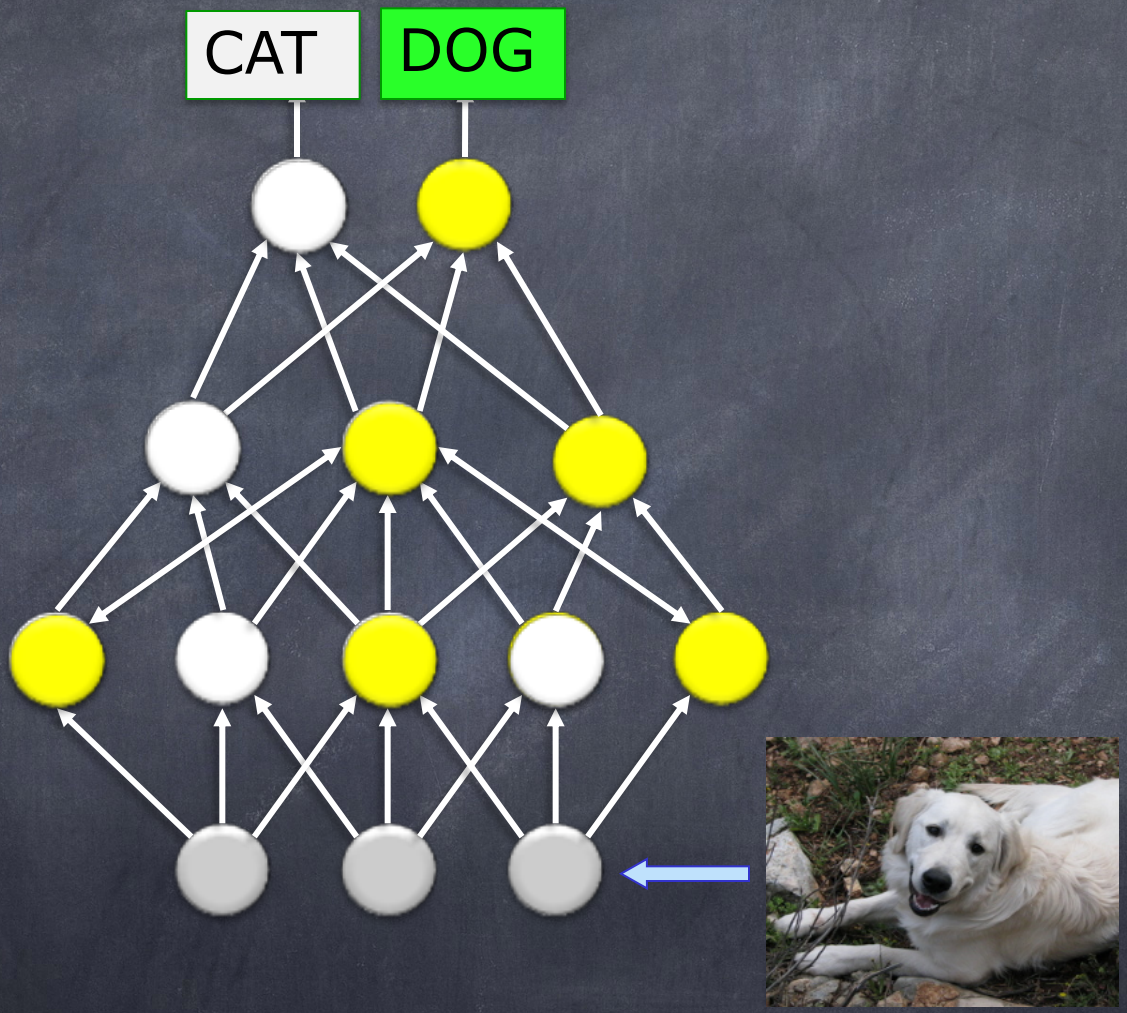
Some useful things to keep in mind while doing the coding



Two Class Classification




CAT
DOG


CAT
DOG

Activation function

- What is "Activation function"?

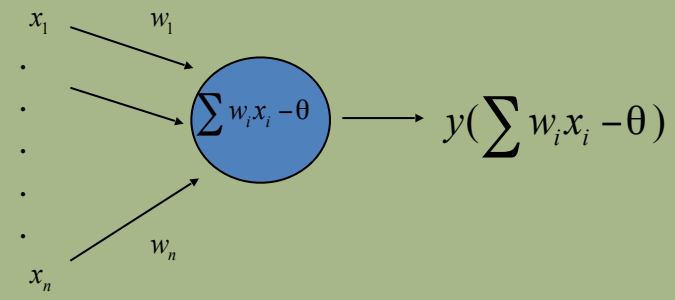
Activation function

- The activation function decides whether a neuron should be activated or not
- Why we need this?

Activation function

- A neural network without an activation function is essentially just a linear regression model
- The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks

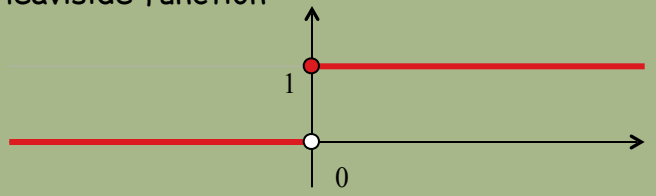
Activating function



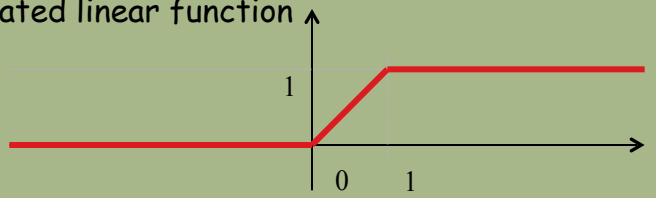
$$y(\xi) = \frac{1}{1+e^{-\xi}}$$

$$y(\xi) = \frac{1-e^{-\xi}}{1+e^{-\xi}}$$

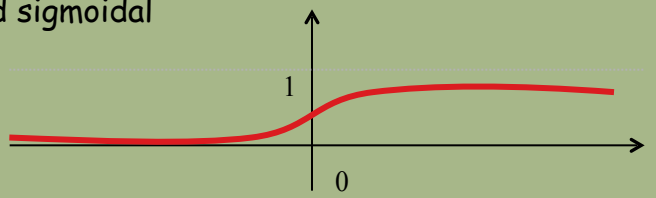
Heaviside function



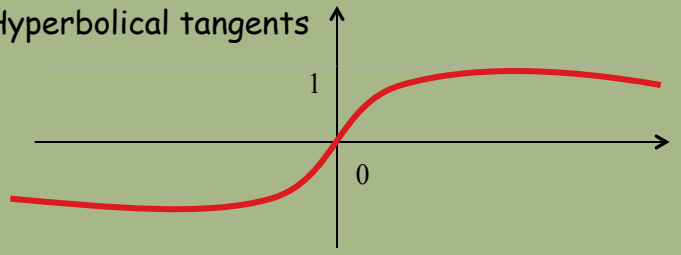
Saturated linear function



Standard sigmoidal function

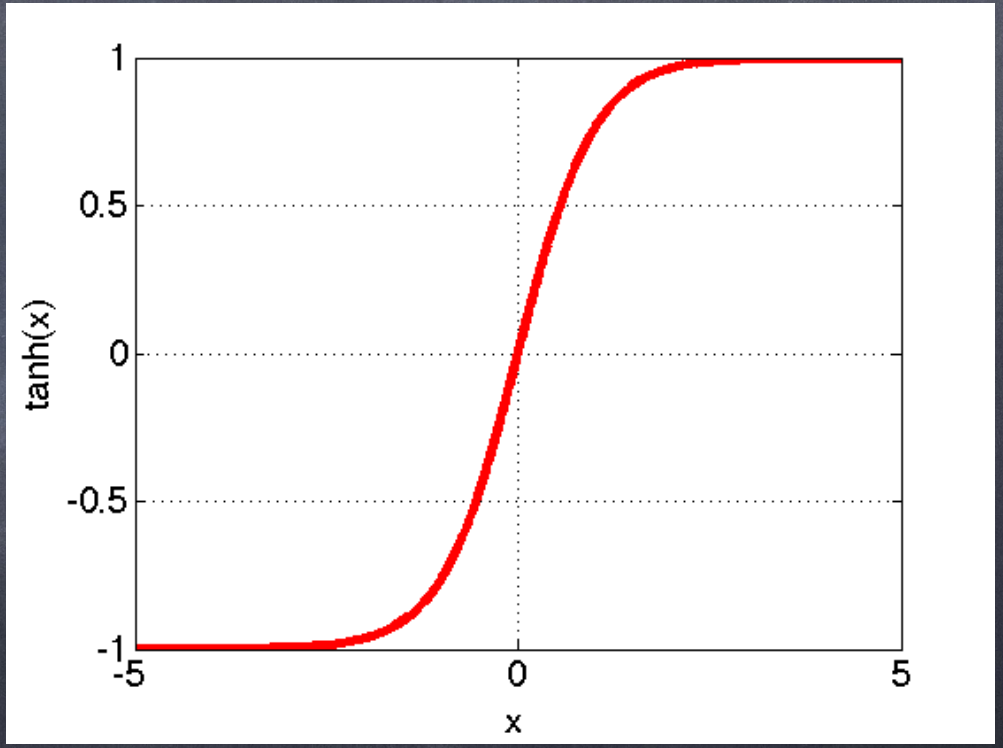


Hyperbolical tangents



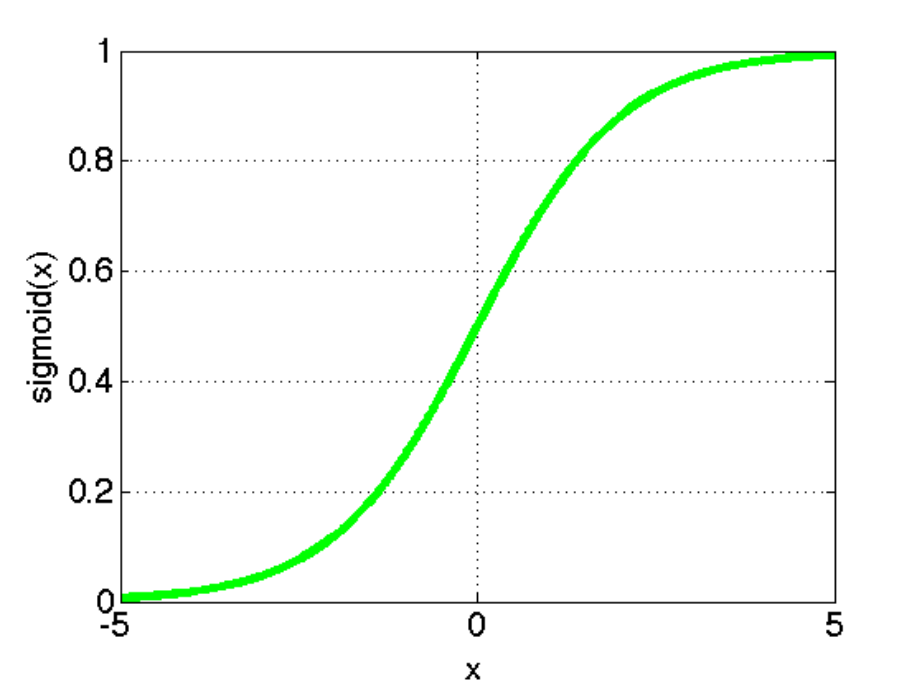
Non Linearity via activation function

- Tanh



Non Linearity via activation function

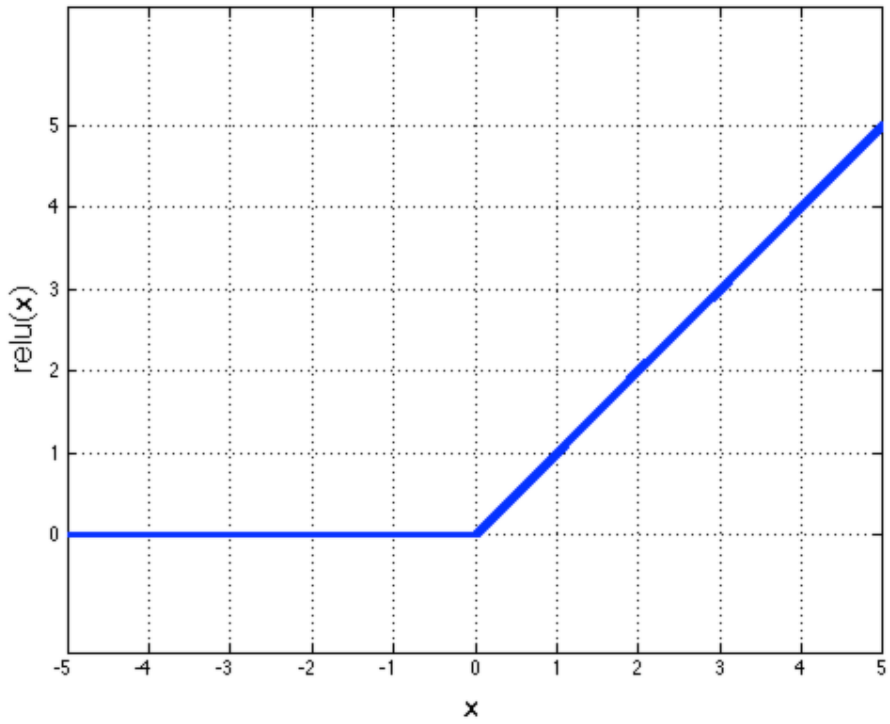
- Sigmoid: 1/(1+exp(-x))



ReLU

- Rectified linear (ReLU) : max(0,x)

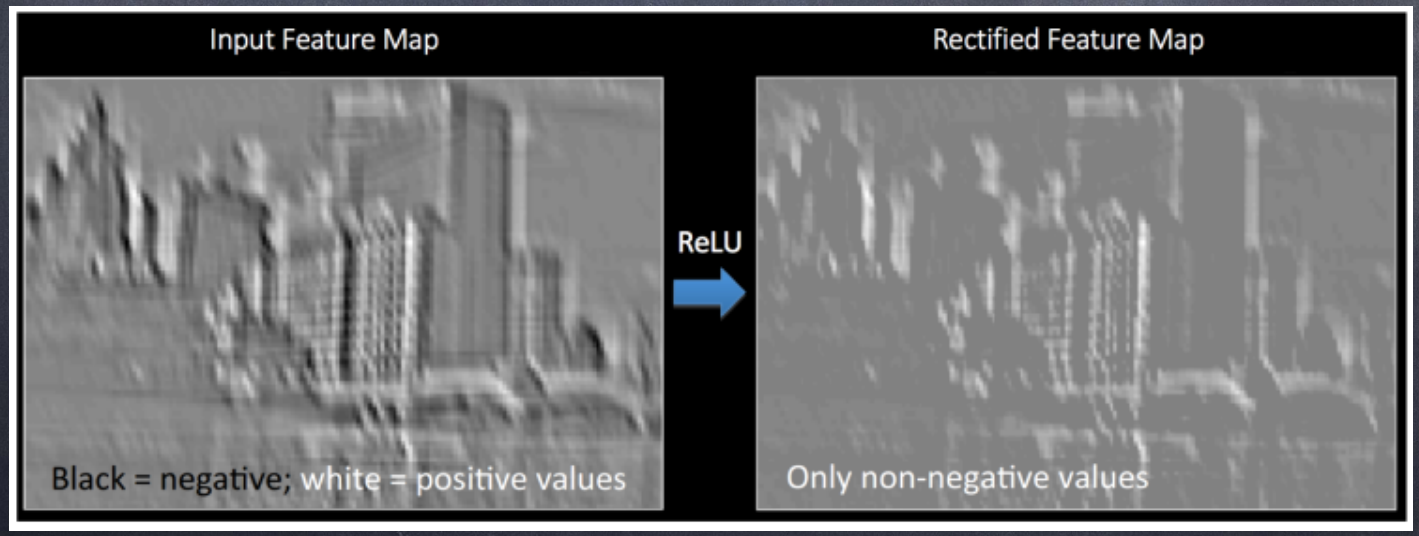
$$h^{(i)} = \max(w^{(i)T}x, 0) = \begin{cases} w^{(i)T}x & w^{(i)T}x > 0 \\ 0 & \text{else} \end{cases}$$




```
1 if input > 0:
2       return input
3 else:
4       return 0
```

ReLU

- Easy to implement and backprop



ReLU

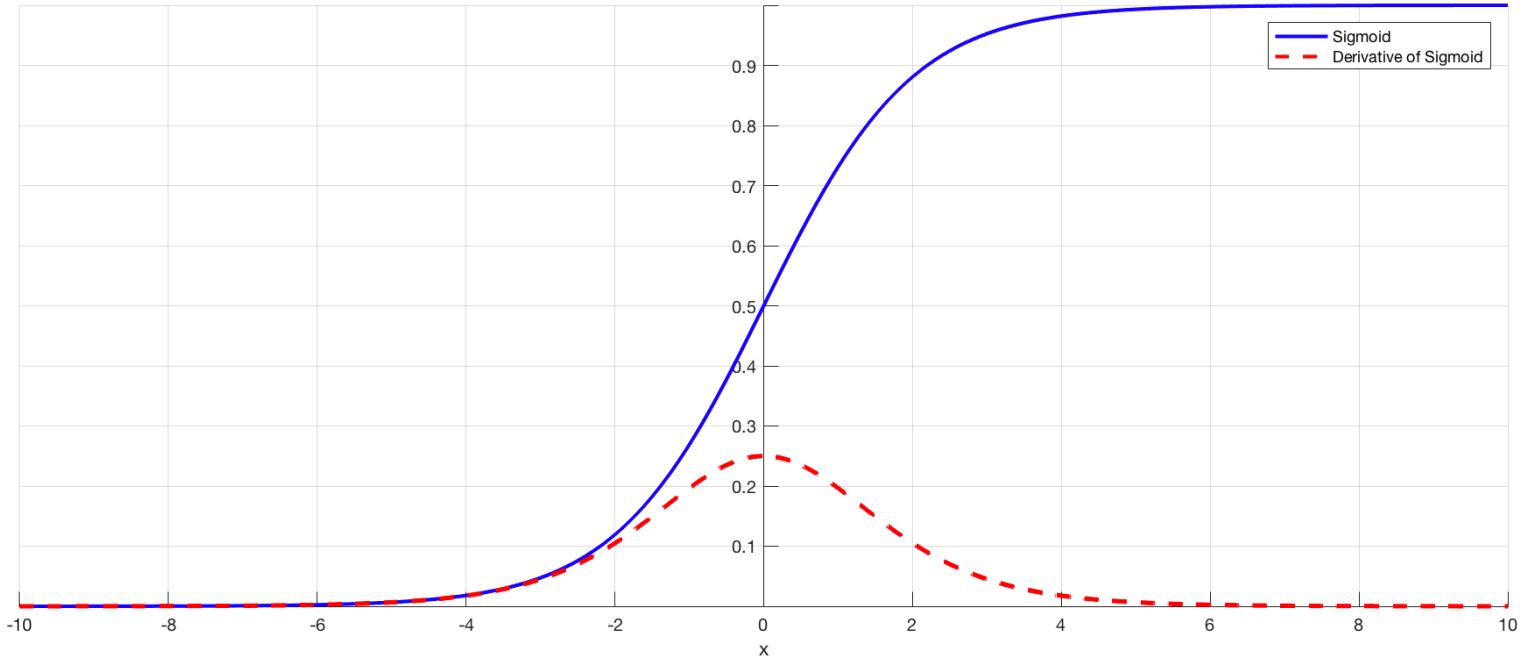
- It is a general purpose activation function
- It helps in addressing Vanishing Gradients

Vanishing Gradients

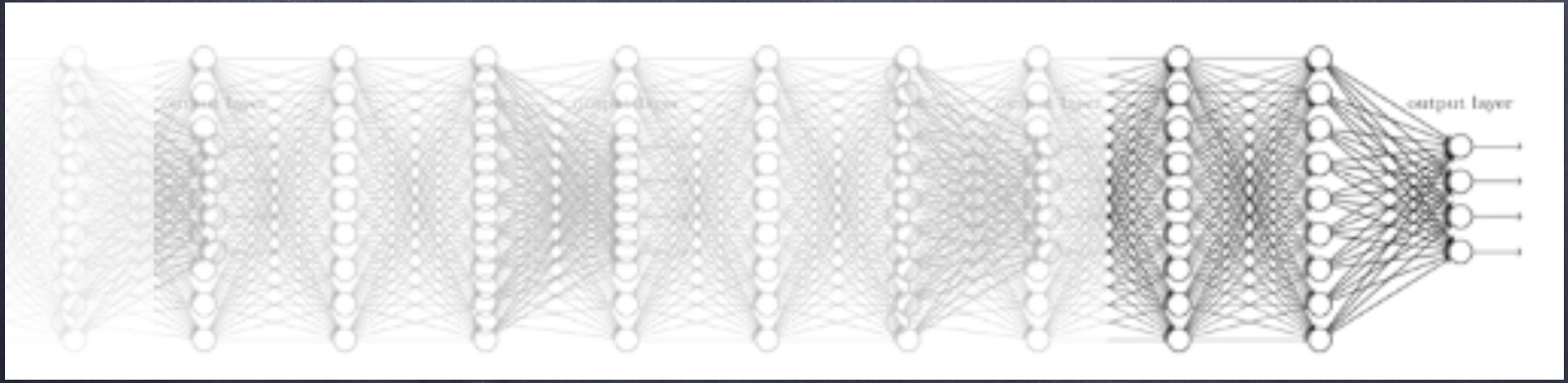
- What do you mean by Vanishing Gradients?

Let us understand it from sigmoid activation function

- It squishes a large input space into a small input space between 0 and 1
- a large change in the input of the sigmoid function will cause a small change in the output. Hence, the derivative becomes small.



Vanishing Gradients



Solution

- One solution - change the activation function from sigmoid to ReLU
- More solutions - we will discuss all these with respect to CNNs

Leaky ReLU

$$h^{(i)} = \max(w^{(i)T}x, 0) = \begin{cases} w^{(i)T}x & w^{(i)T}x > 0 \\ 0.01w^{(i)T}x & \text{else} \end{cases}$$

Learning Rate

- Recall GDR ...

Derivation of GDR …

The vector of $\frac{\partial E}{\partial w_i}$ derivatives that form the gradient can be obtained by differentiating E

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x}_d) \\
\frac{\partial E}{\partial w_i} &= \sum_{d\in D}(t_d - o_d)(-x_{id})
\end{aligned}$$

The weight update rule for standard gradient descent can be summarized as

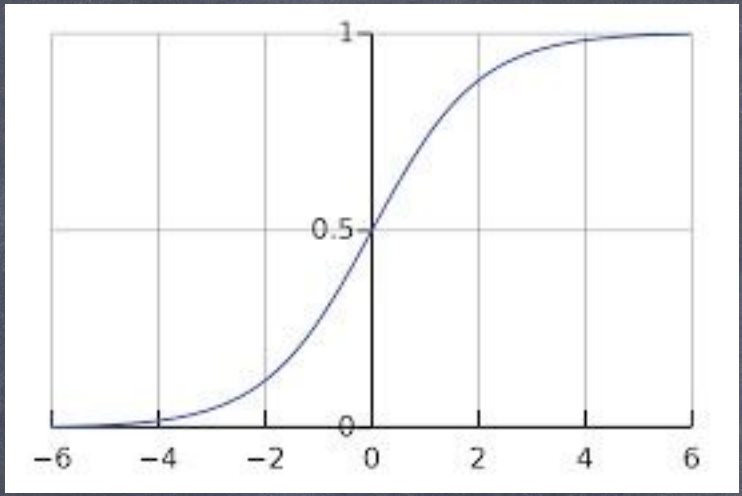
$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i}$$

$$\Delta w_i = \eta\sum_{d\in D}(t_d - o_d)x_{id}$$

Bias in Neural Network

- In simple words, neural network bias can be defined as the constant which is added to the product of features and weights
- It is used to offset the result
- It helps the models to shift the activation function towards the positive or negative side

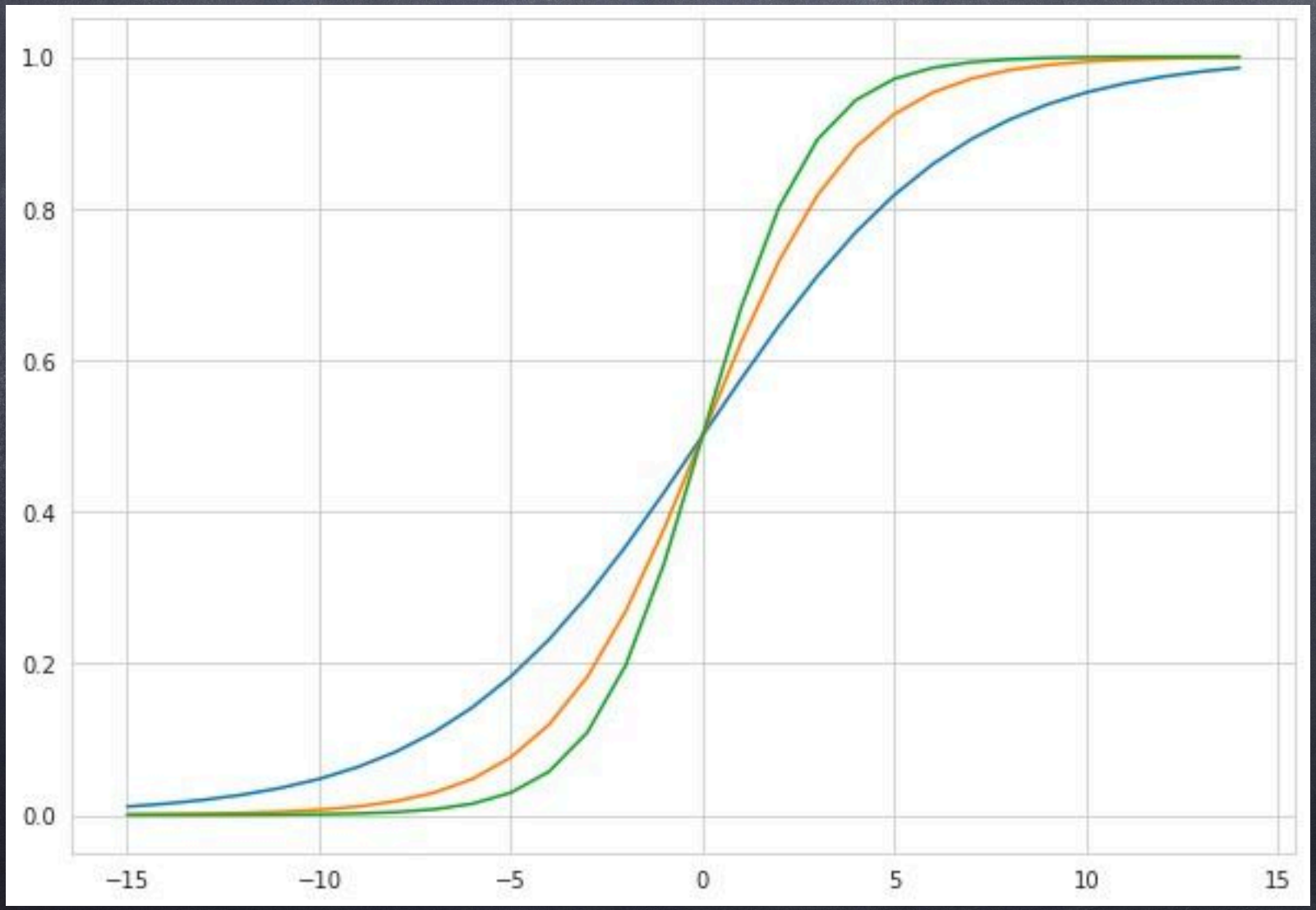
Let us understand it by using sigmoid function



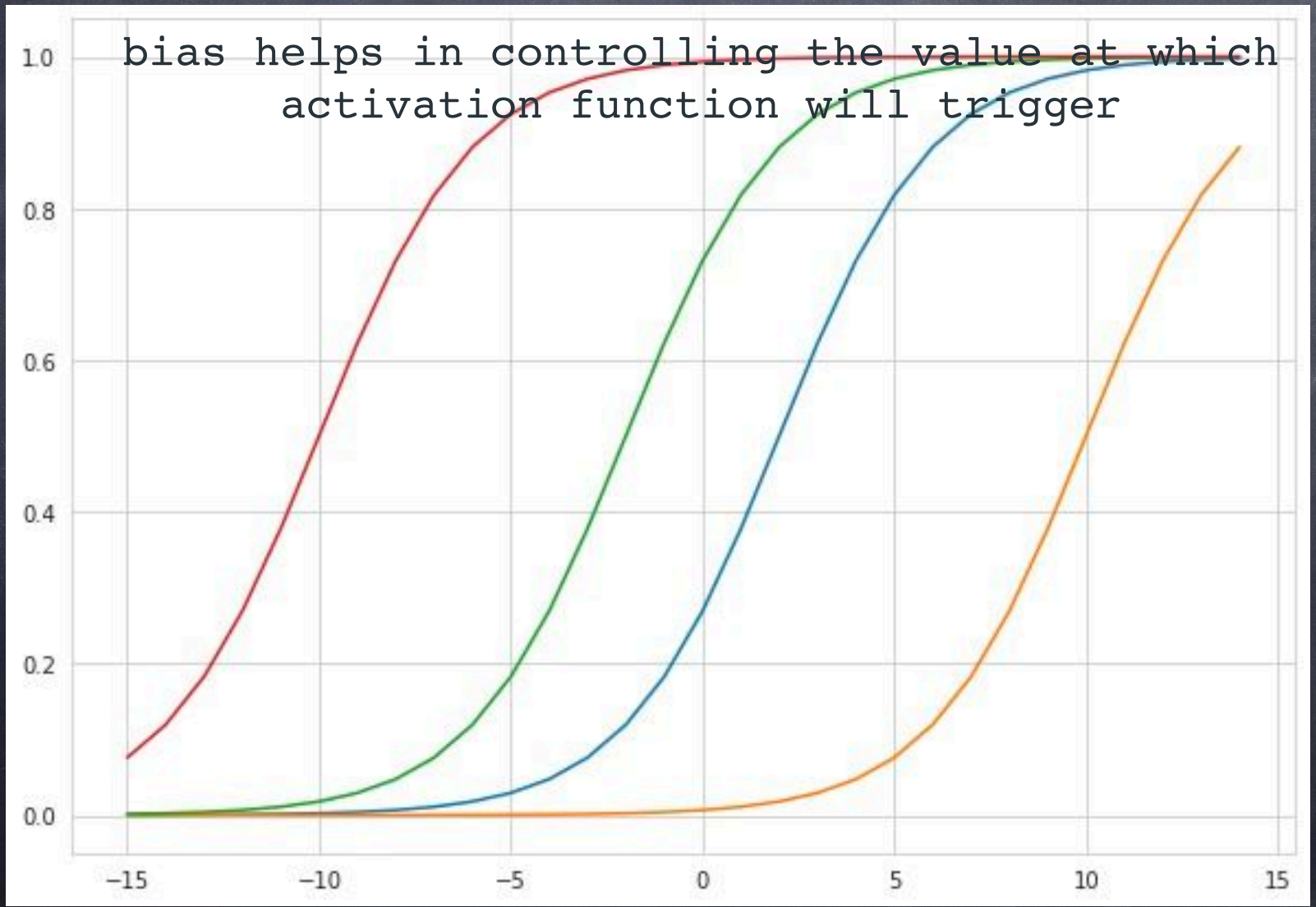
$$sigmoid\ function = \frac{1}{1 + e^{-x}}$$

$$sigmoid\ function = \frac{1}{1 + e^{-(w*x+b)}}$$

Fix b = 0 and vary w

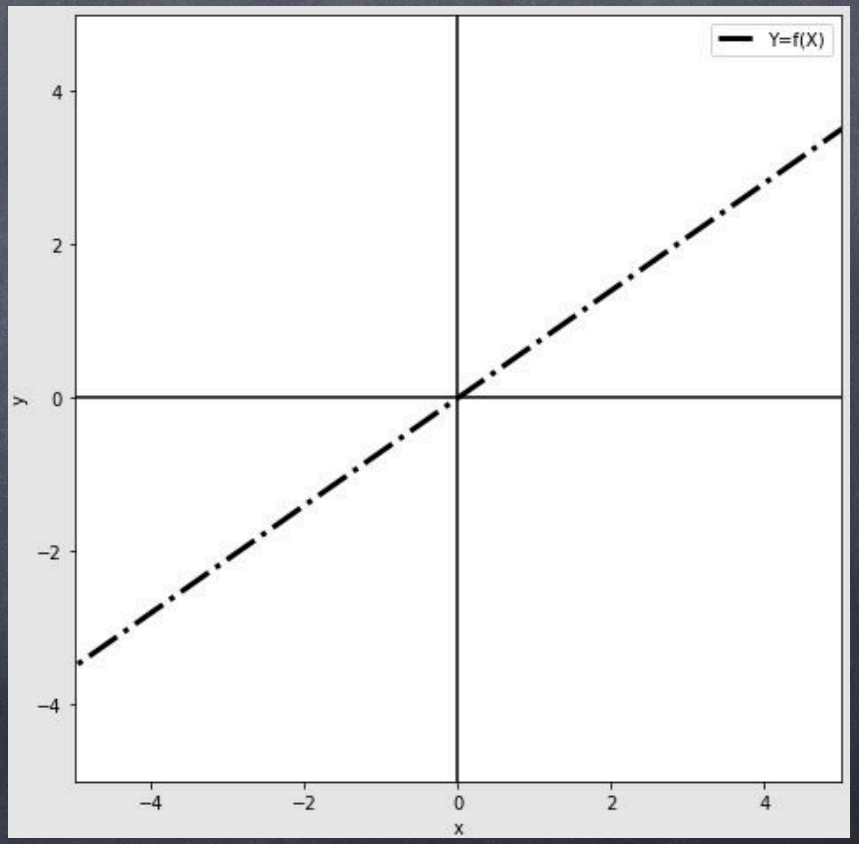


Fix w and vary b



One bias per layer vs one bias per node

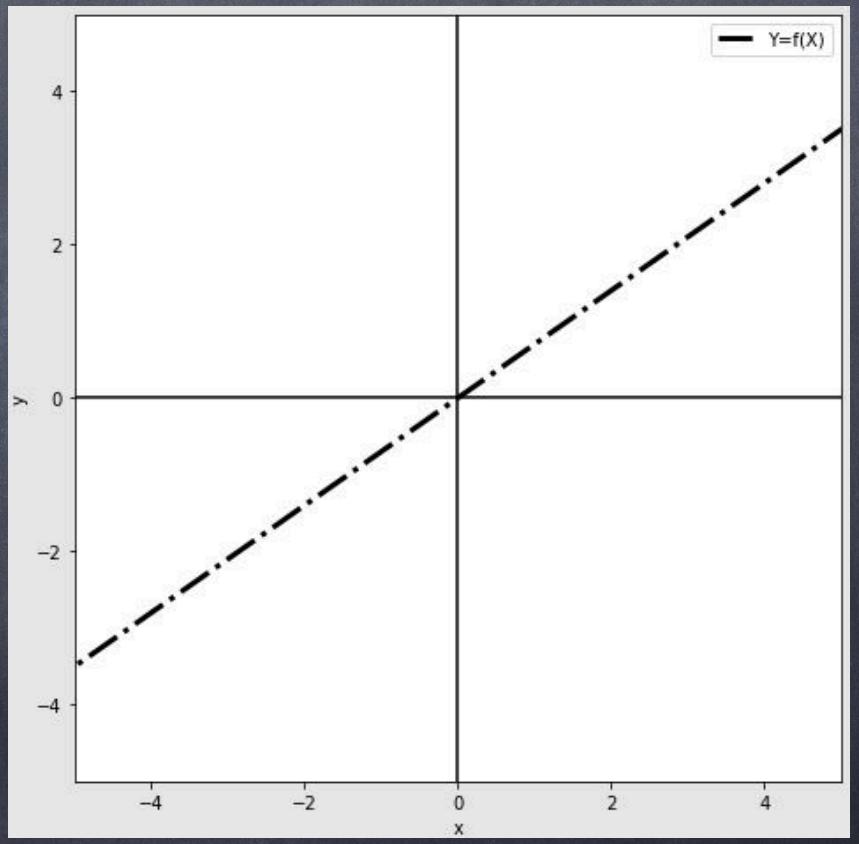
- A given neural network computes the function Y=f(X), where X and Y are feature vector and output vector
- If the given neural network has weight 'W' then it can also be represented as Y=f(X,W)
- If the dimensionality of both X and Y is equal to 1, the function can be plotted in a two-dimensional plane



https://www.turing.com/kb/necessity-of-bias-in-neural-networks

One bias per layer vs one bias per node

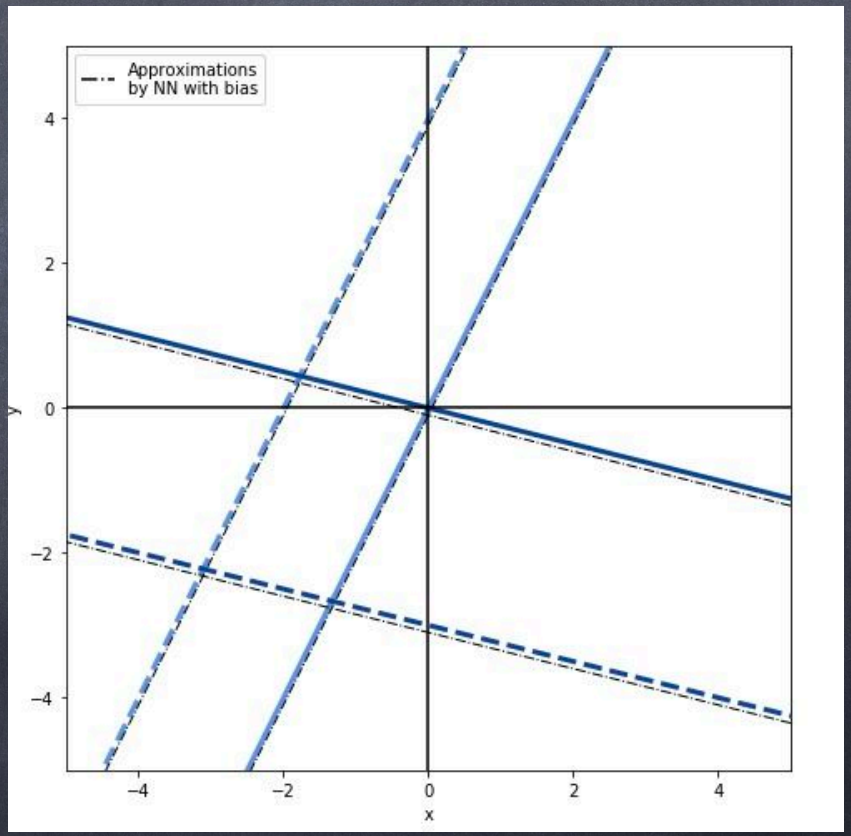
- A given neural network computes the function Y=f(X), where X and Y are feature vector and output vector
- If the given neural network has weight 'W' then it can also be represented as Y=f(X,W)
- If the dimensionality of both X and Y is equal to 1, the function can be plotted in a two-dimensional plane
- Such a neural network can approximate any linear function of the form y=mx + c
- When c=0, then y=f(x)=mx and the neural network can approximate only the functions passing through origin



https://www.turing.com/kb/necessity-of-bias-in-neural-networks

One bias per layer vs one bias per node

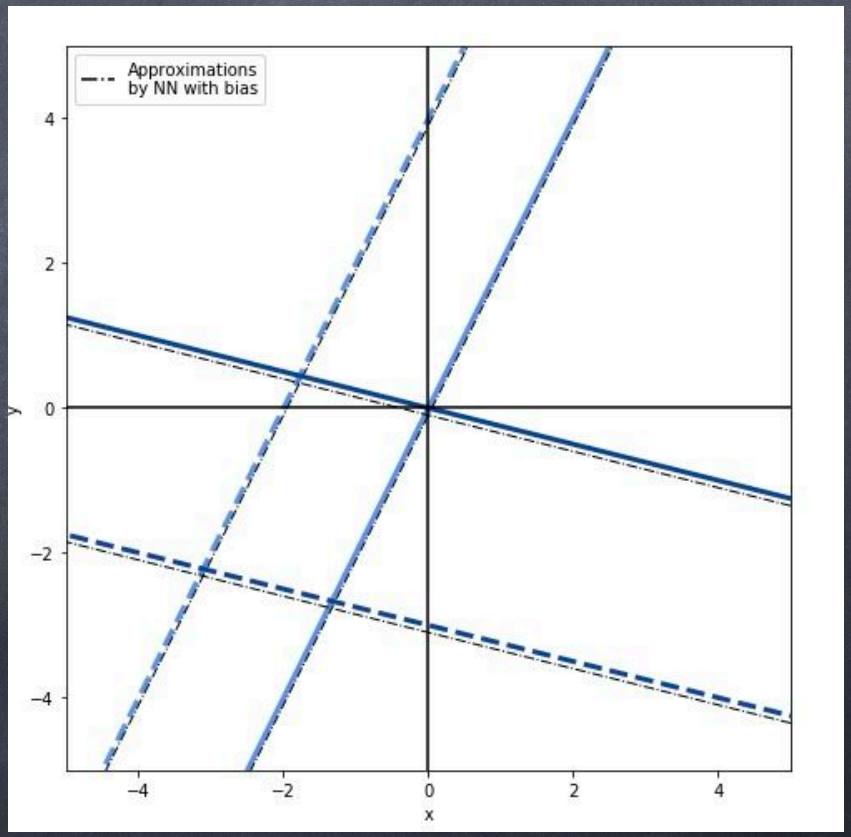
- If a function includes the constant term c (or b), it can approximate any of the linear functions in the plane
- the neural network would compute y=f(x) + b which includes all the predictions by the neural network shifted by the constant 'b'



https://www.turing.com/kb/necessity-of-bias-in-neural-networks

One bias per layer vs one bias per node

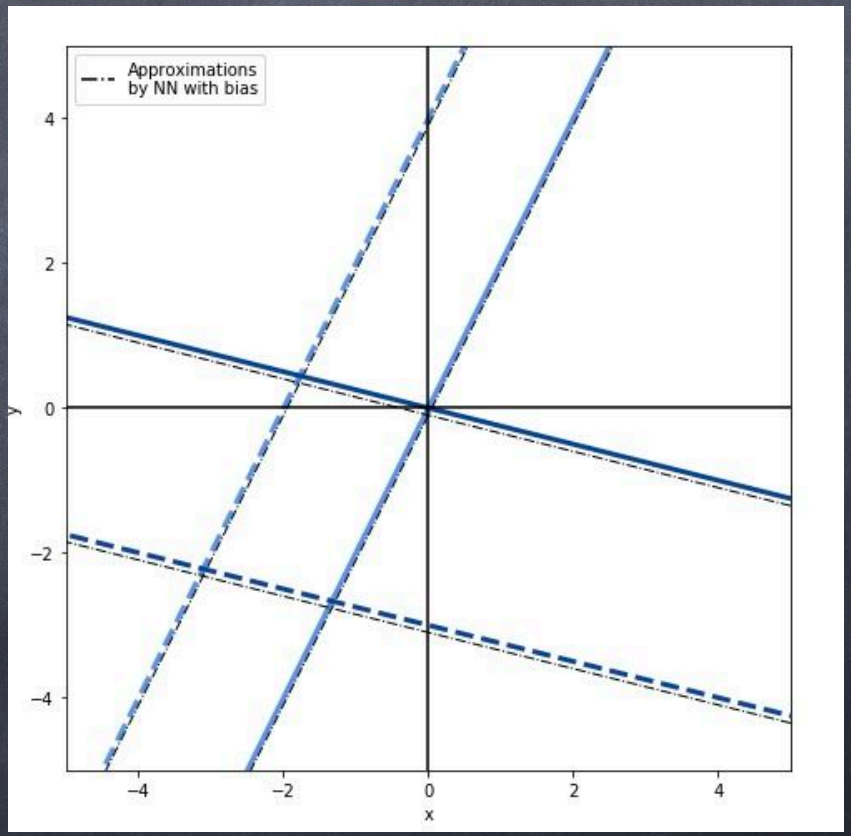
- if we add one more input to the neural network layer, the function is defined as y=f(x1,x2)
- Since both x1 and x2 are independent components, they should have independent biases b1 and b2, respectively
- Thus, the neural networks can be represented as y=f(x1, x2) + b1 + b2



https://www.turing.com/kb/necessity-of-bias-in-neural-networks

One bias per layer vs one bias per node

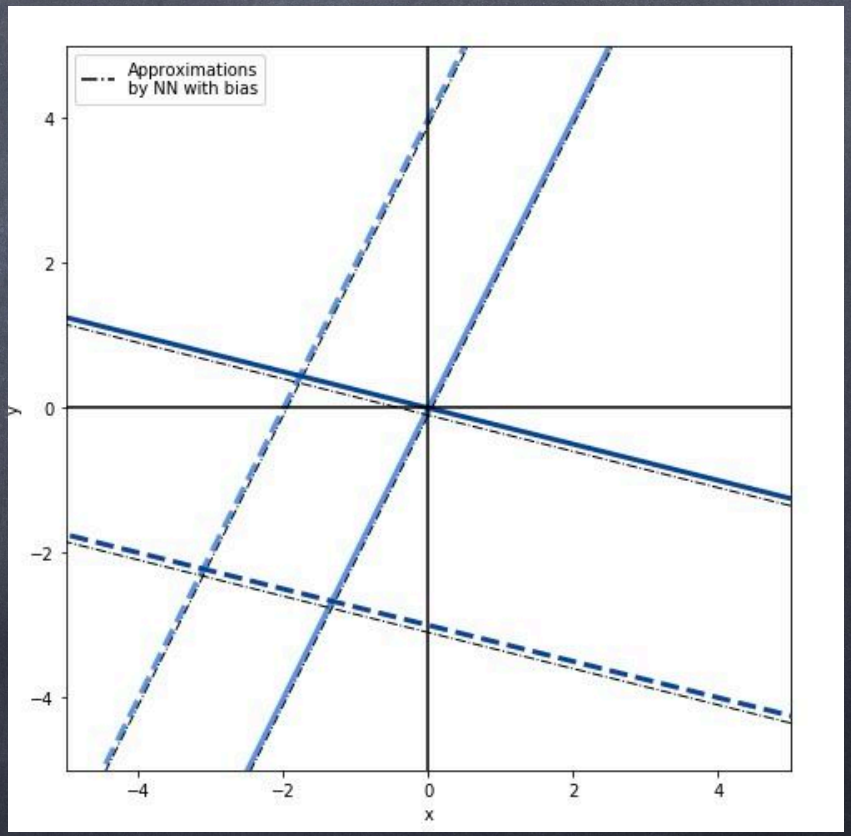
- Consider a feature vector X =[x1, x2, ….,xn] and their associated systematic errors (b1+ b2+ ….bn) for n inputs
- A single layer neural network computing a function
- Y =f(X,W) + (b1+ b2+ ….bn)



https://www.turing.com/kb/necessity-of-bias-in-neural-networks

One bias per layer vs one bias per node

- Consider a feature vector X =[x1, x2, ….,xn] and their associated systematic errors (b1+ b2+ ….bn) for n inputs
- A single layer neural network computing a function
- Y =f(X,W) + b



https://www.turing.com/kb/necessity-of-bias-in-neural-networks

One bias per layer vs one bias per node

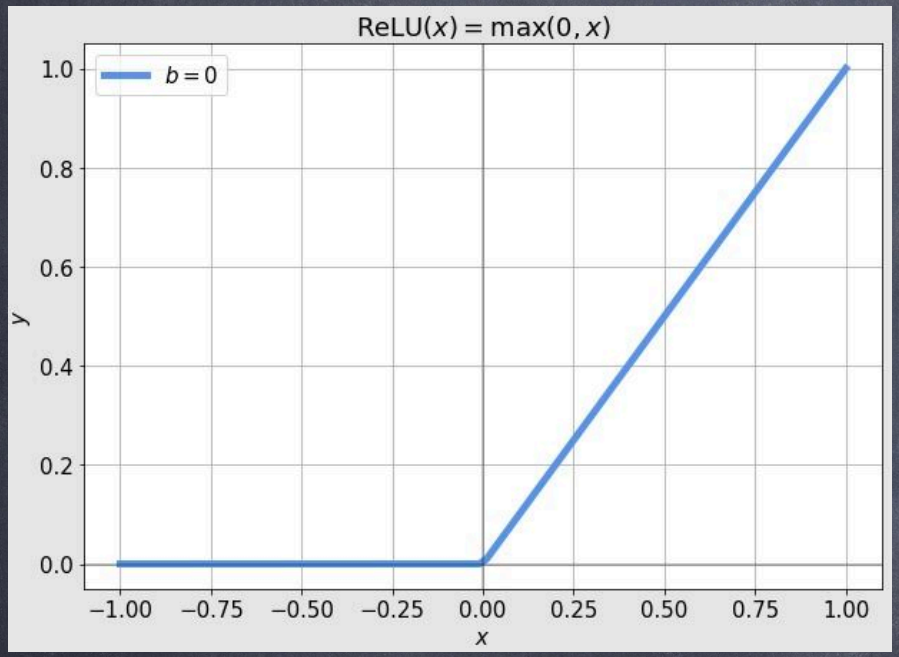
- Expand this to the network with multiple layers

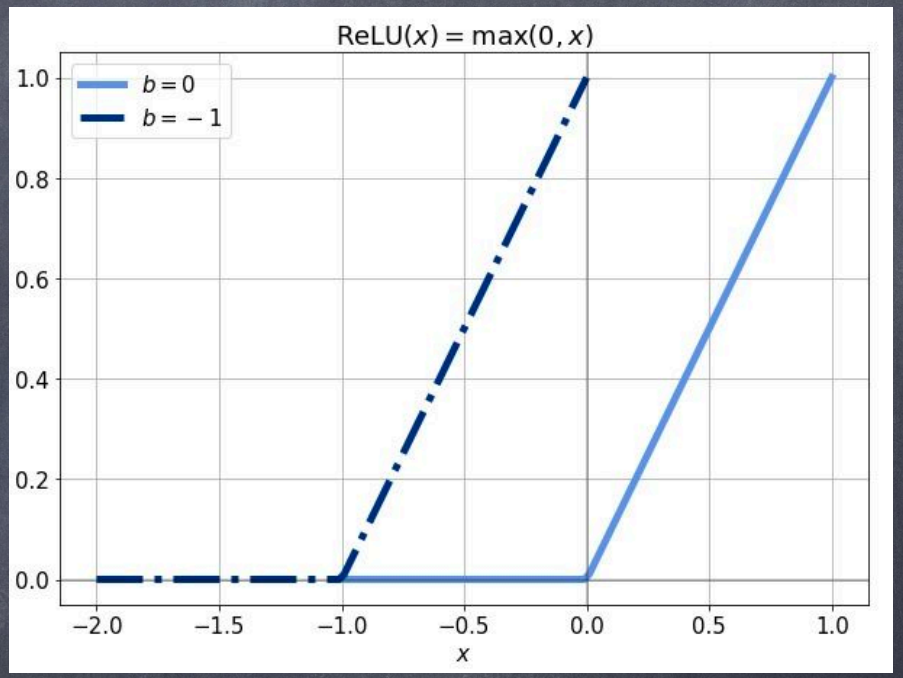
$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

$$f(X) = \sigma_n(W_n\sigma_{n-1}(...(W_2\sigma_1(W_1X + b_1) + b_2) + ...) + b_n)$$

https://www.turing.com/kb/necessity-of-bias-in-neural-networks

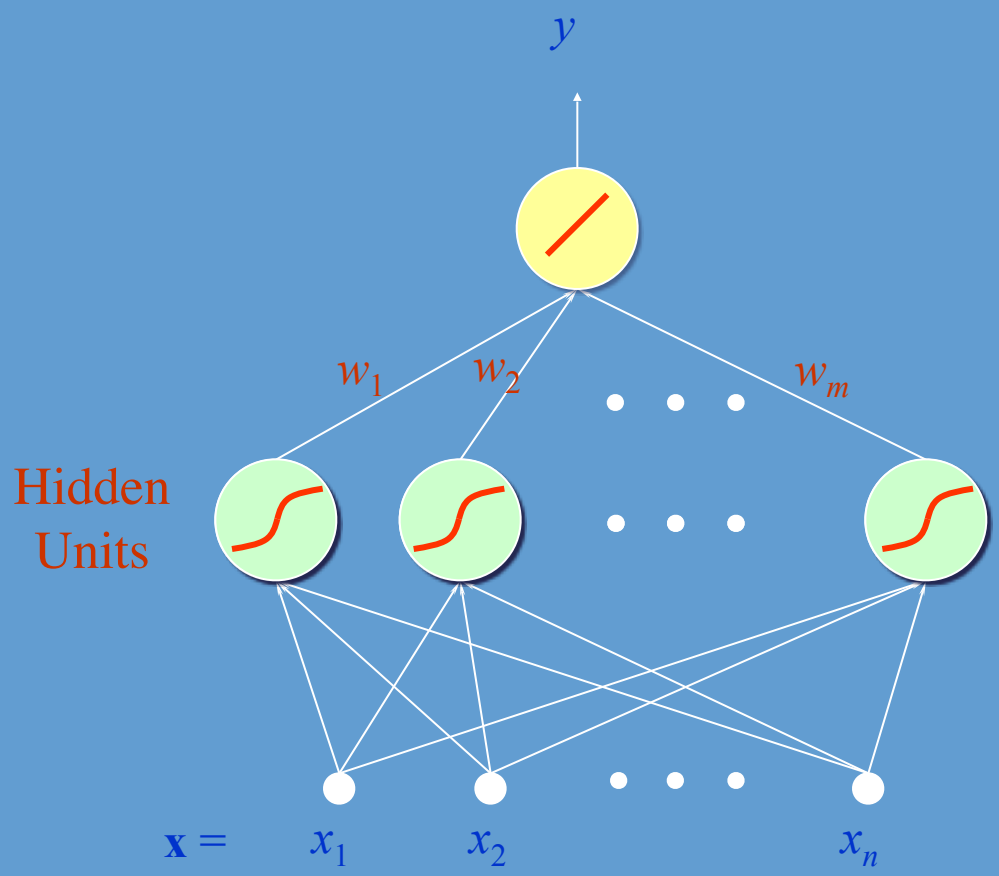
Including bias within the activation function





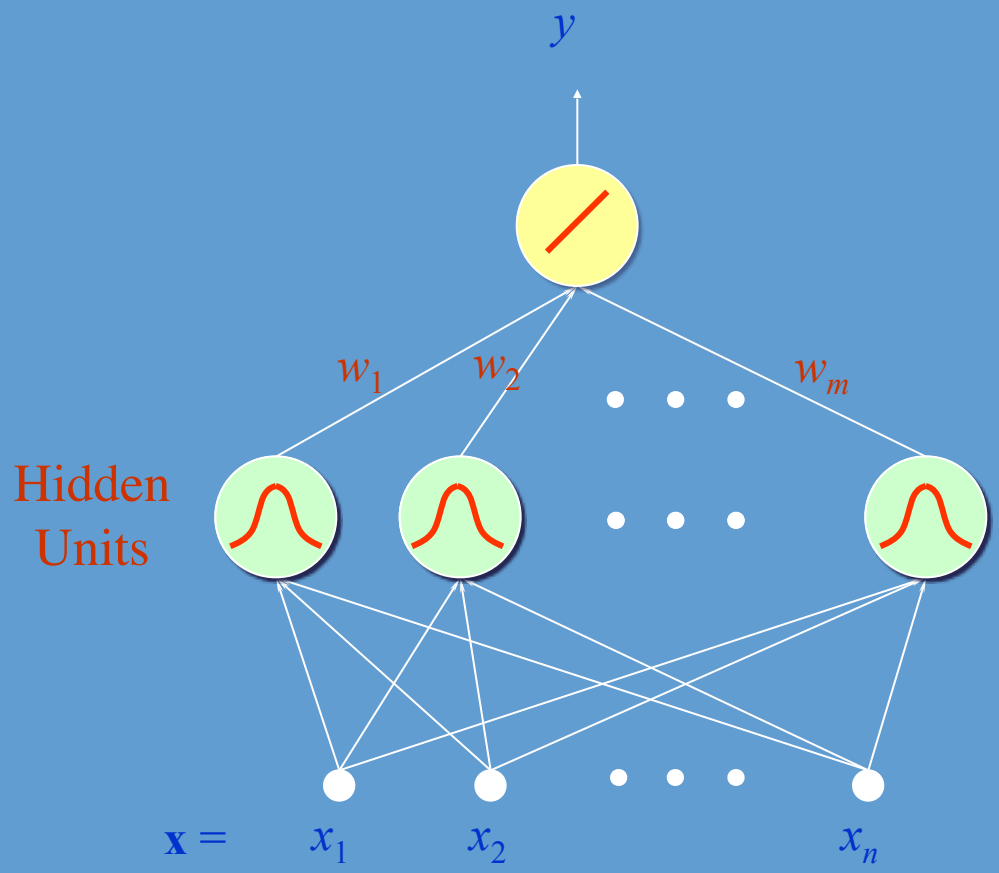
- This will allow neurons to be activated even when the input is zero. This in turn will allow backpropagation to occur even when input is zero

Single-Hidden Layer NNet



Radial Basis Function Networks

$$f(\mathbf{x}) = \sum_{i=1}^{m} w_i \phi_i(\mathbf{x})$$



Non-Linear Models

$$f(\mathbf{x}) = \sum_{i=1}^{m} w_i \phi_i(\mathbf{x})$$

$$f(\mathbf{x}) = \sum_{i=1}^{m} \overbrace{w_i}^{\text{Weights}} \underbrace{\phi_i(\mathbf{x})}_{\text{Adjusted by the Learning process}}$$

Typical Radial Functions

- Gaussian

$$\phi(r) = e^{-\frac{r^2}{2\sigma^2}} \quad \sigma > 0 \quad \text{and} \quad r \in \Re$$

- Hardy Multiquadratic

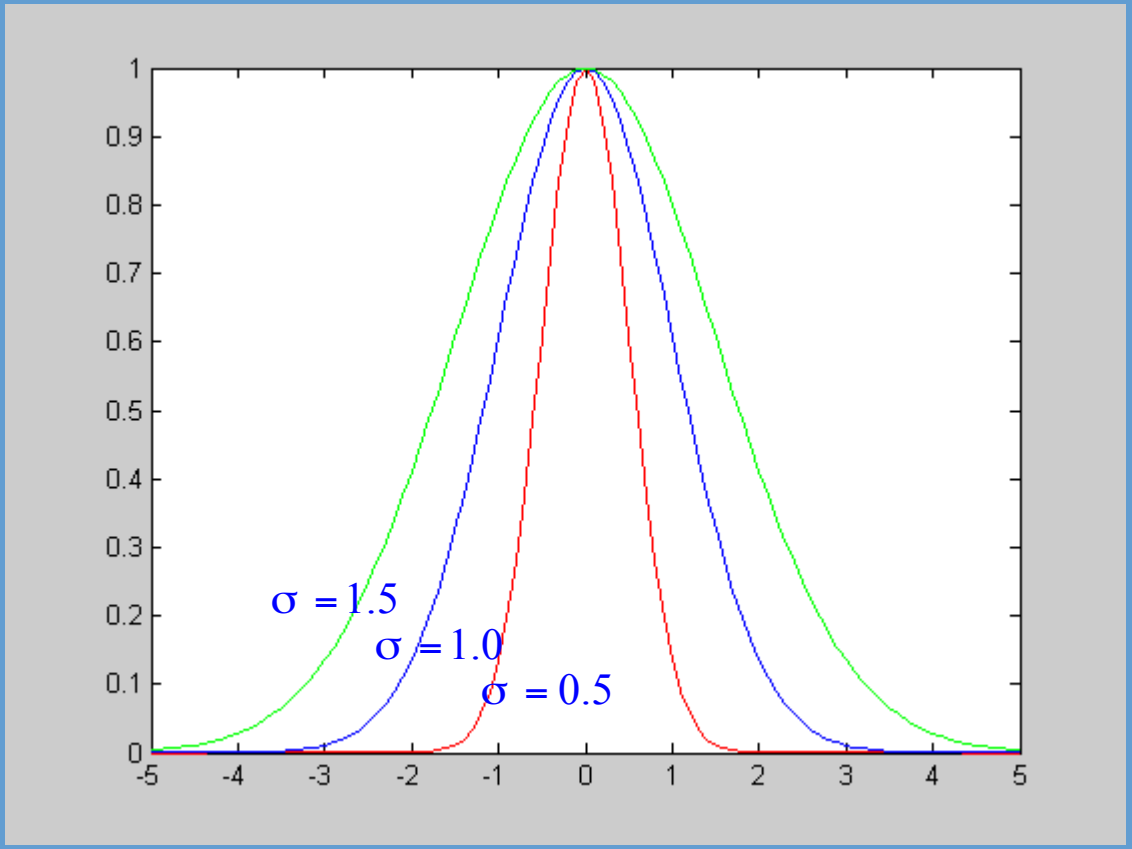
$$\phi(r) = \sqrt{r^2 + c^2}\,/\,c \quad c > 0 \quad \text{and} \quad r \in \Re$$

- Inverse Multiquadratic

$$\phi(r) = c\,/\,\sqrt{r^2 + c^2} \quad c > 0 \quad \text{and} \quad r \in \Re$$

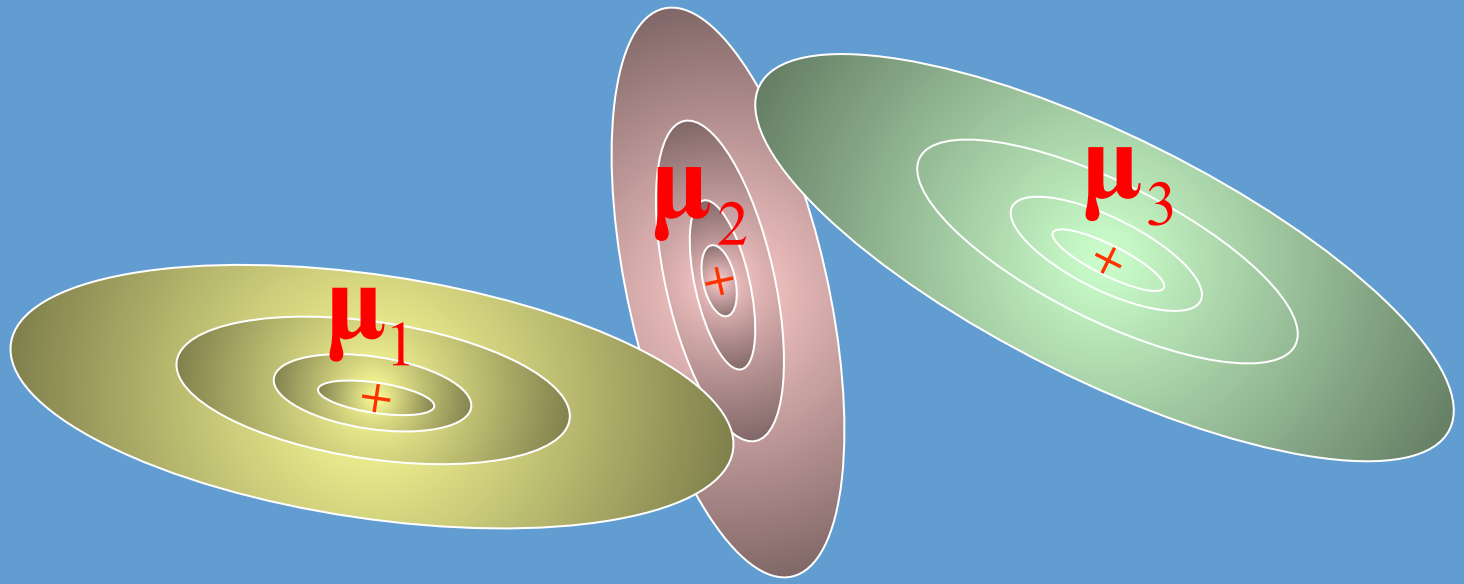
Gaussian Basis Function (σ=0.5,1.0,1.5)

$$\phi(r) = e^{-\frac{r^2}{2\sigma^2}} \quad \sigma > 0 \quad \text{and} \quad r \in \Re$$

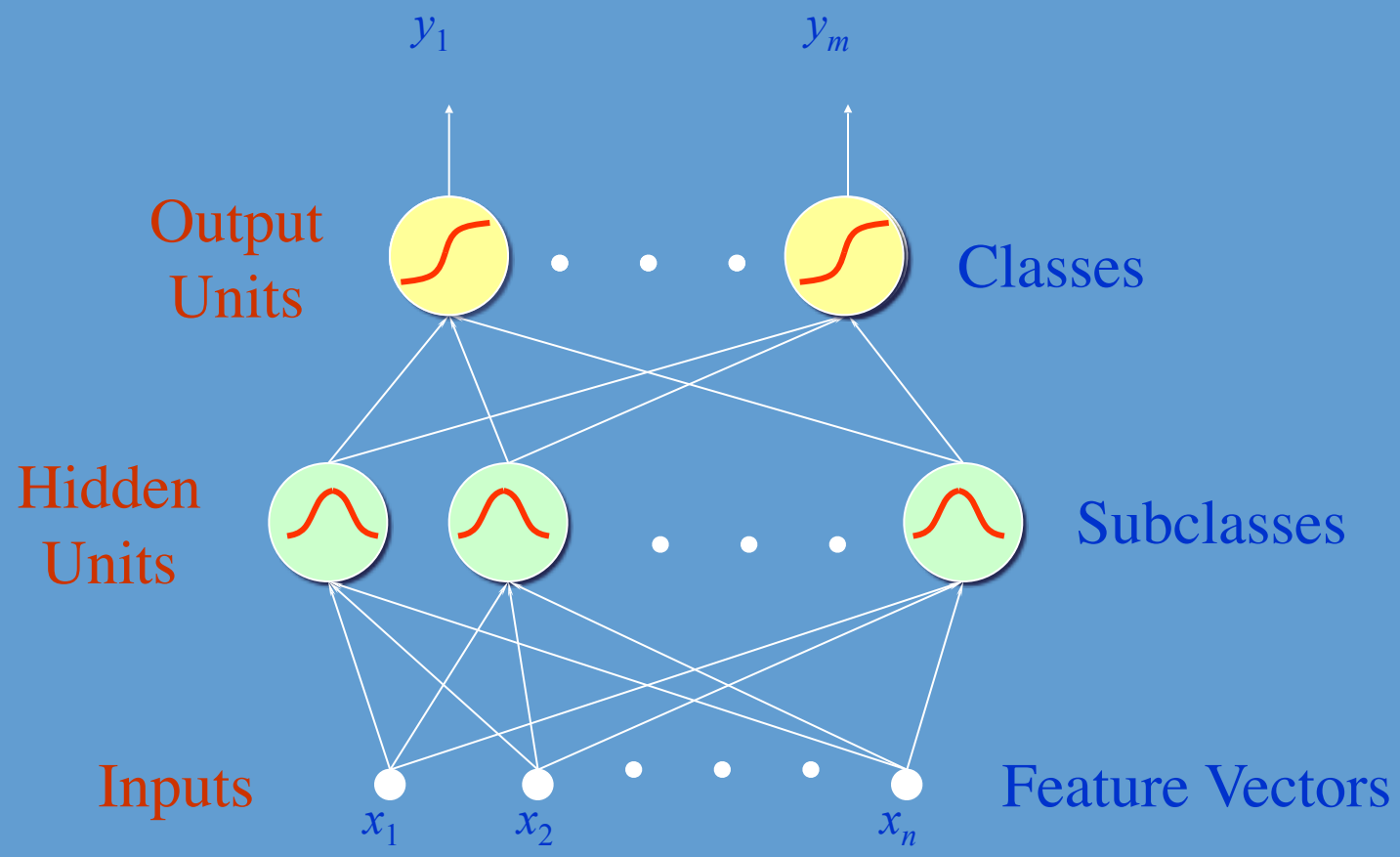


Most General RBF

$$\phi_i(\mathbf{x}) = \phi\left((\mathbf{x}-\boldsymbol{\mu}_i)^T \Sigma^{-1} (\mathbf{x}-\boldsymbol{\mu}_i)\right)$$



The Topology of RBF NNet



Radial Basis Function Networks

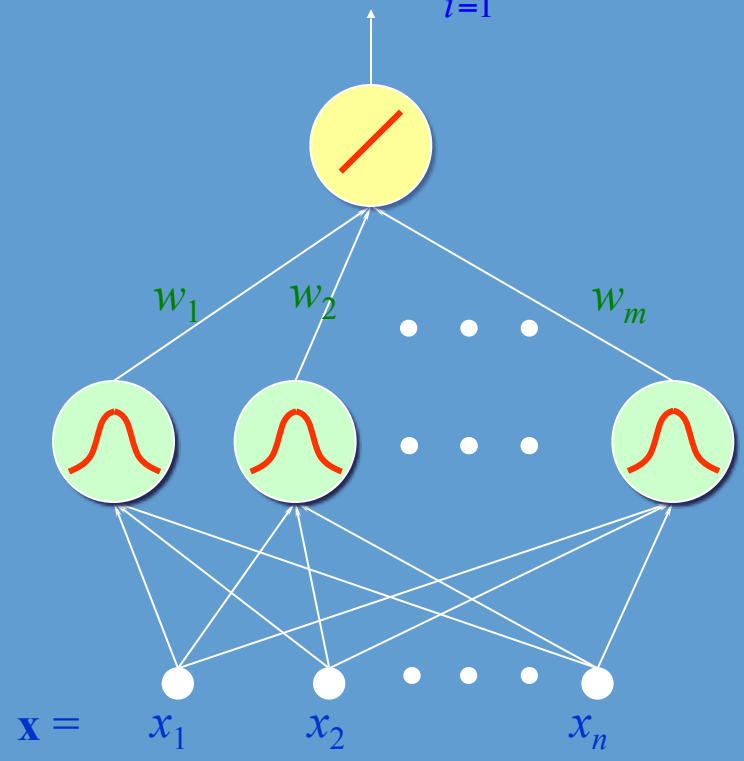
Training set $\mathcal{T} = \left\{\left(\mathbf{x}^{(k)}, y^{(k)}\right)\right\}_{k=1}^{p}$

$$y = f(\mathbf{x}) = \sum_{i=1}^{m} w_i \phi_i(\mathbf{x})$$

Goal $y^{(k)} \approx f\left(\mathbf{x}^{(k)}\right)$ for all k

$$\min E = \frac{1}{2}\sum_{k=1}^{p}\left[y^{(k)} - f\left(\mathbf{x}^{(k)}\right)\right]^2$$

$$= \frac{1}{2}\sum_{k=1}^{p}\left[y^{(k)} - \sum_{i=1}^{m} w_i \phi_i\left(\mathbf{x}^{(k)}\right)\right]^2$$



Learn the Optimal Weight Vector

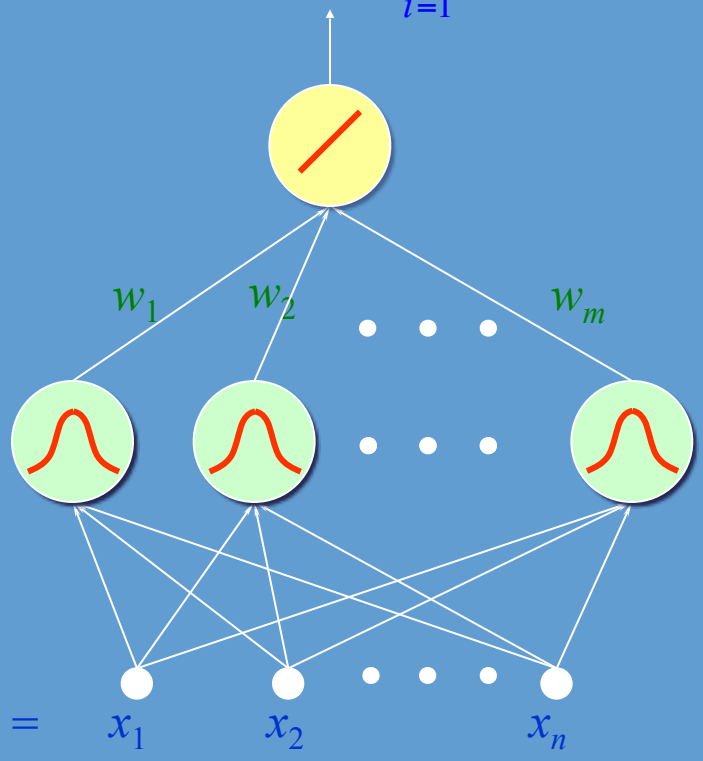
Training set $T = \left\{\left(\mathbf{x}^{(k)}, y^{(k)}\right)\right\}_{k=1}^{p}$

$$y = f(\mathbf{x}) = \sum_{i=1}^{m} w_i \phi_i(\mathbf{x})$$

Goal $y^{(k)} \approx f\left(\mathbf{x}^{(k)}\right)$ for all k

$$\min E = \frac{1}{2}\sum_{k=1}^{p}\left[y^{(k)} - f\left(\mathbf{x}^{(k)}\right)\right]^2$$

$$= \frac{1}{2}\sum_{k=1}^{p}\left[y^{(k)} - \sum_{i=1}^{m} w_i \phi_i\left(\mathbf{x}^{(k)}\right)\right]^2$$



Regularization

$$\lambda_i \geq 0$$

Training set $T = \left\{ \left(\mathbf{x}^{(k)}, y^{(k)} \right) \right\}_{k=1}^{p}$

If regularization is not needed, set $\lambda_i = 0$

Goal $y^{(k)} \approx f\left(\mathbf{x}^{(k)}\right)$ for all k

$$\min C = \frac{1}{2} \sum_{k=1}^{p} \left[y^{(k)} - f\left(\mathbf{x}^{(k)}\right) \right]^2 + \frac{1}{2} \sum_{i=1}^{m} \lambda_i w_i^2$$

$$= \frac{1}{2} \sum_{k=1}^{p} \left[y^{(k)} - \sum_{i=1}^{m} w_i \phi_i\left(\mathbf{x}^{(k)}\right) \right]^2 + \frac{1}{2} \sum_{i=1}^{m} \lambda_i w_i^2$$

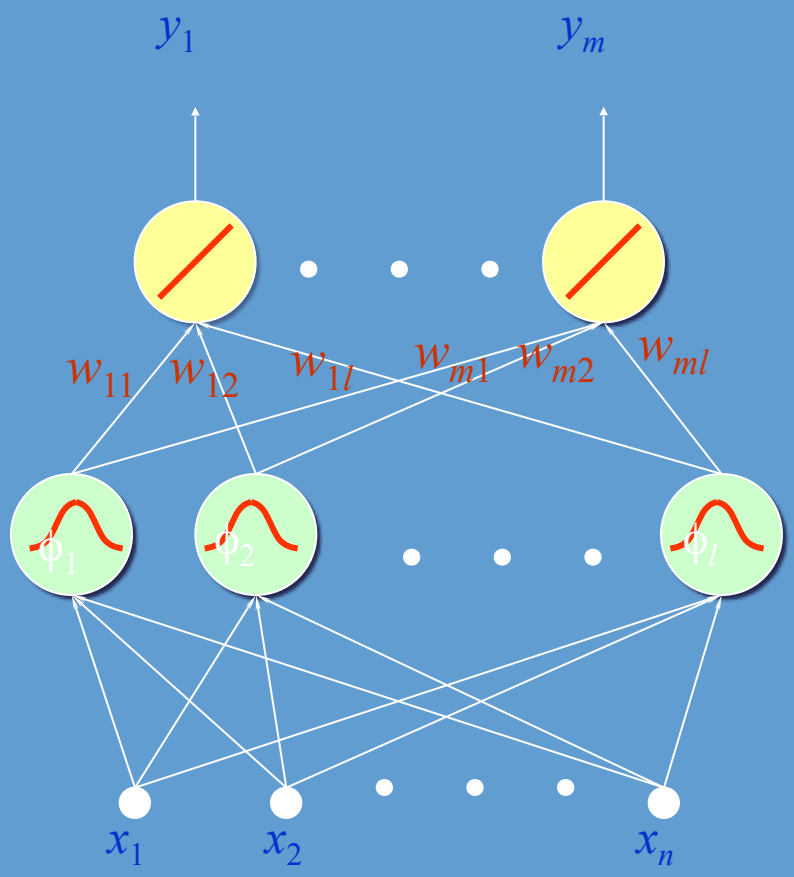
Learn the Optimal Weight Vector

$$f(\mathbf{x}) = \sum_{i=1}^{m} w_i \phi_i(\mathbf{x})$$

Minimize $$C = \frac{1}{2}\sum_{k=1}^{p}\left[y^{(k)} - f\left(\mathbf{x}^{(k)}\right)\right]^2 + \frac{1}{2}\sum_{i=1}^{m}\lambda_i w_i^2$$

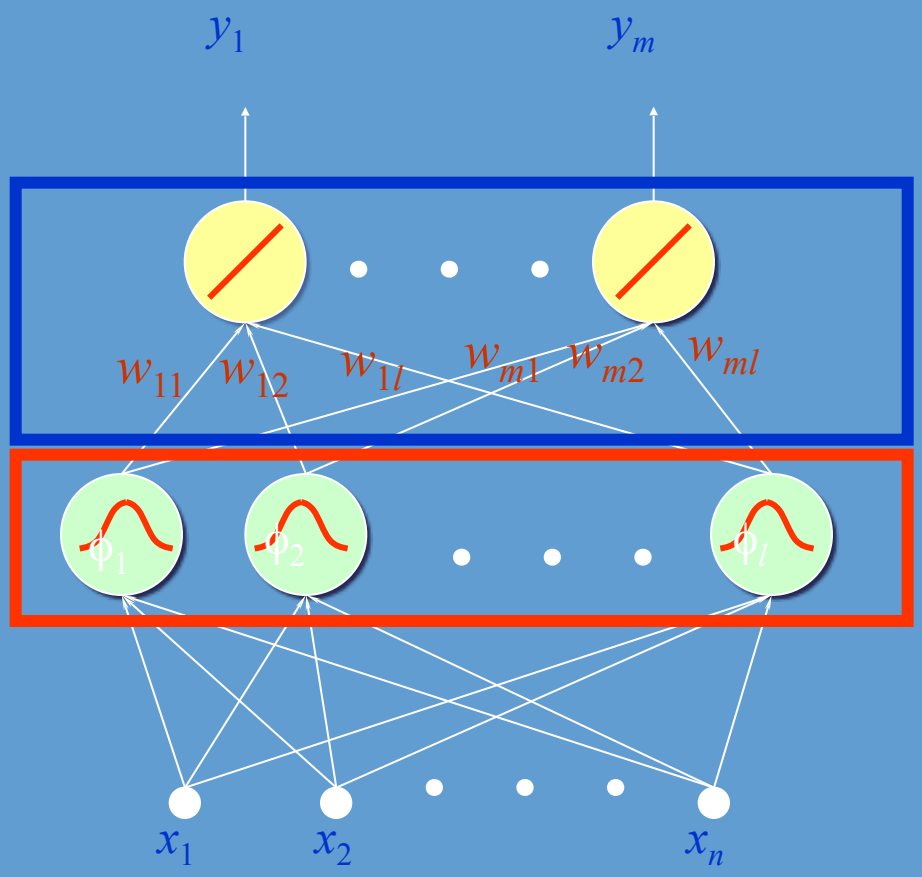
$$0 = \frac{\partial C}{\partial w_j} = -\sum_{k=1}^{p}\left[y^{(k)} - f\left(\mathbf{x}^{(k)}\right)\right]\frac{\partial f\left(\mathbf{x}^{(k)}\right)}{\partial w_j} + \lambda_j w_j$$

What to Learn?



- Weights w_{ij}'s
- Centers μ_j's of ϕ_j's
- Widths σ_j's of ϕ_j's

Two-Stage Training



Step 2

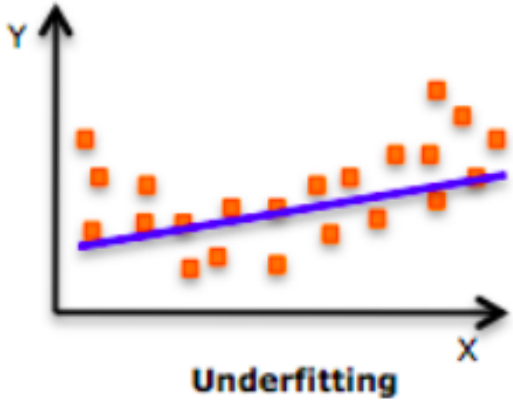
Determines w_{ij}'s.

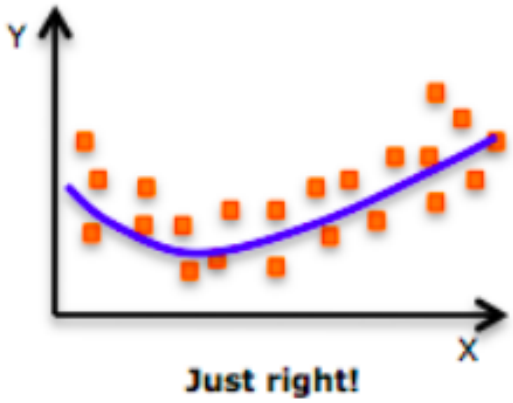
Step 1

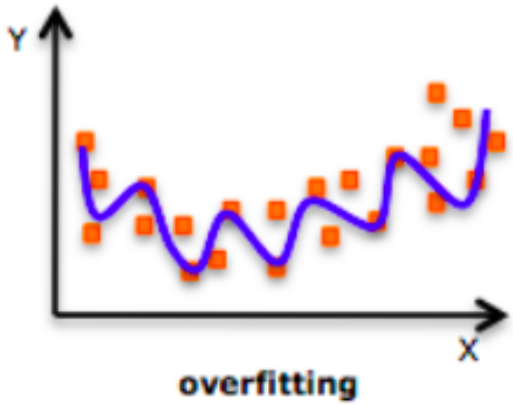
Determines

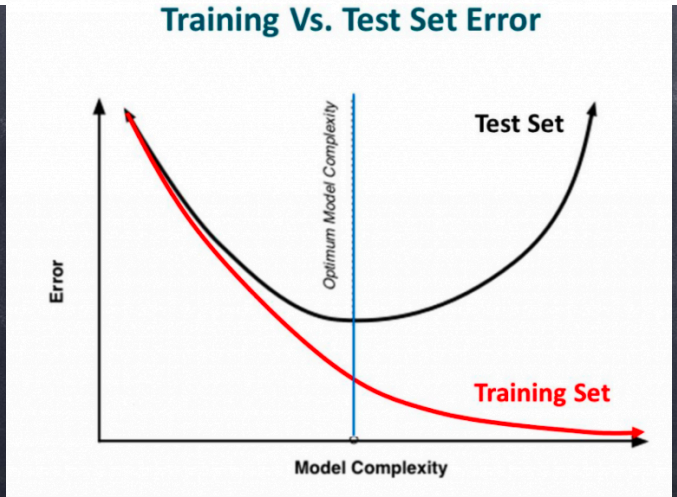
- Centers μ_j's of ϕ_j's.
- Widths σ_j's of ϕ_j's.

What is regularization

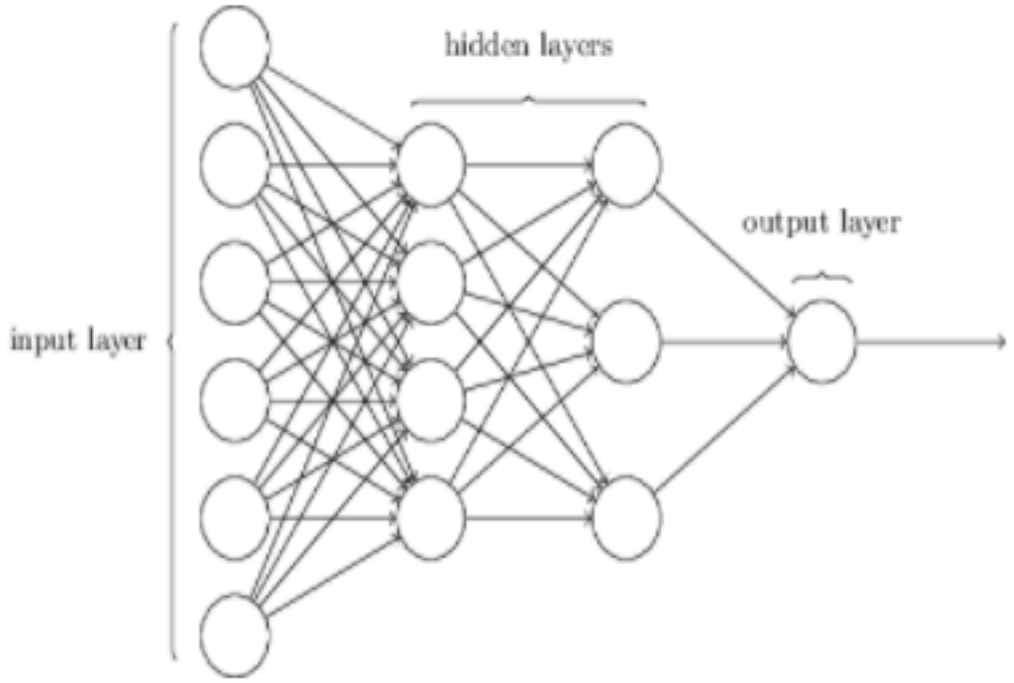


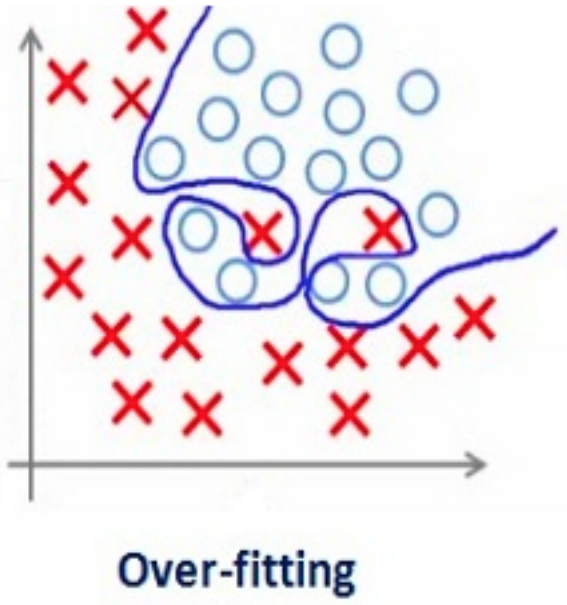




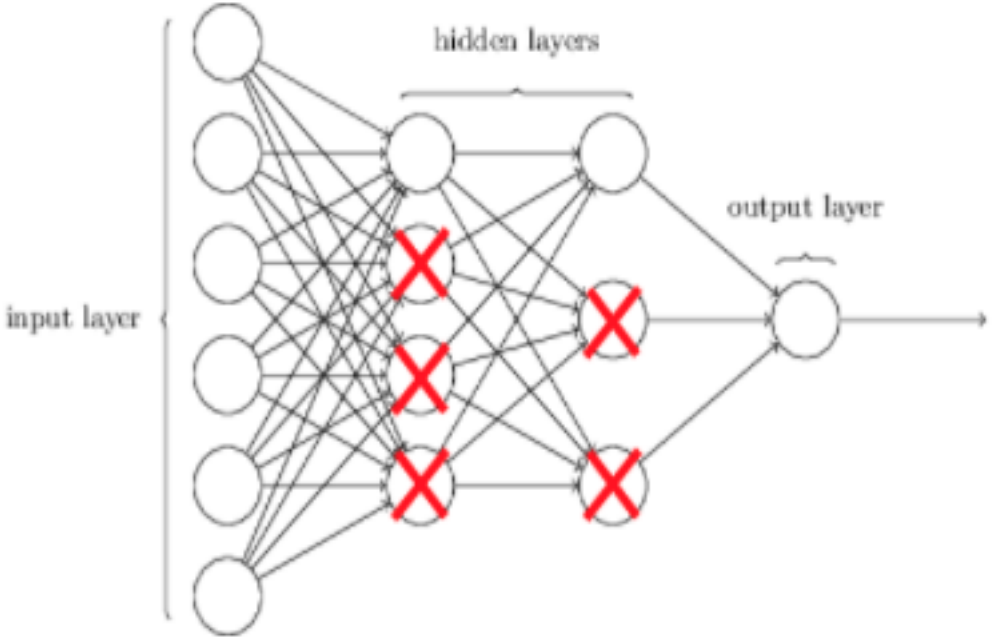


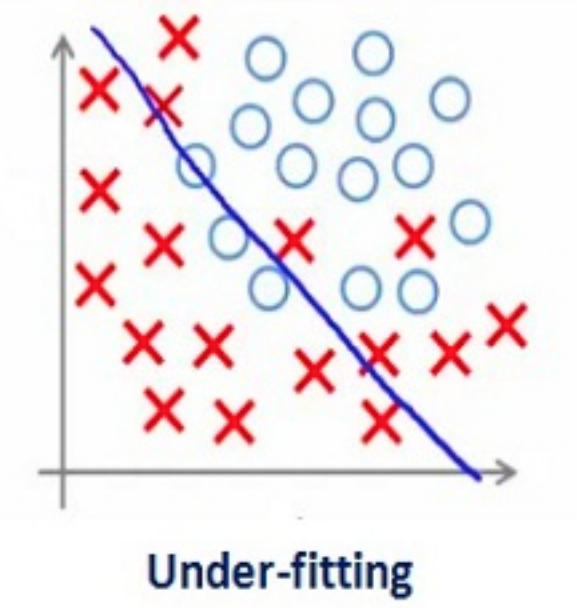
Overfitting

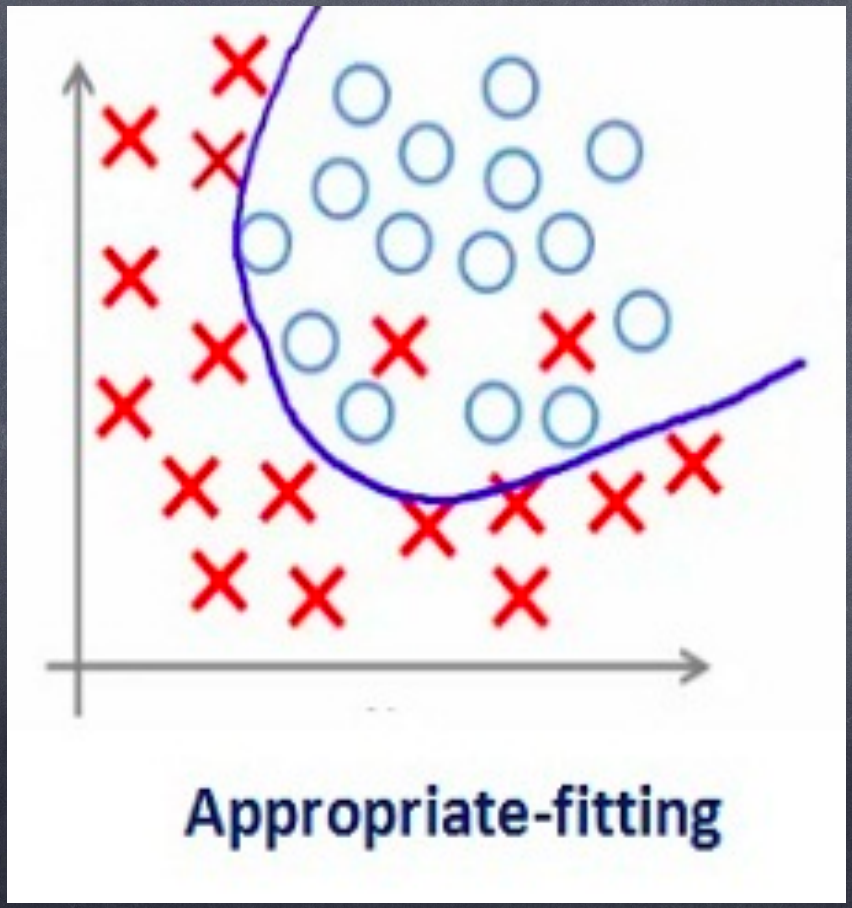


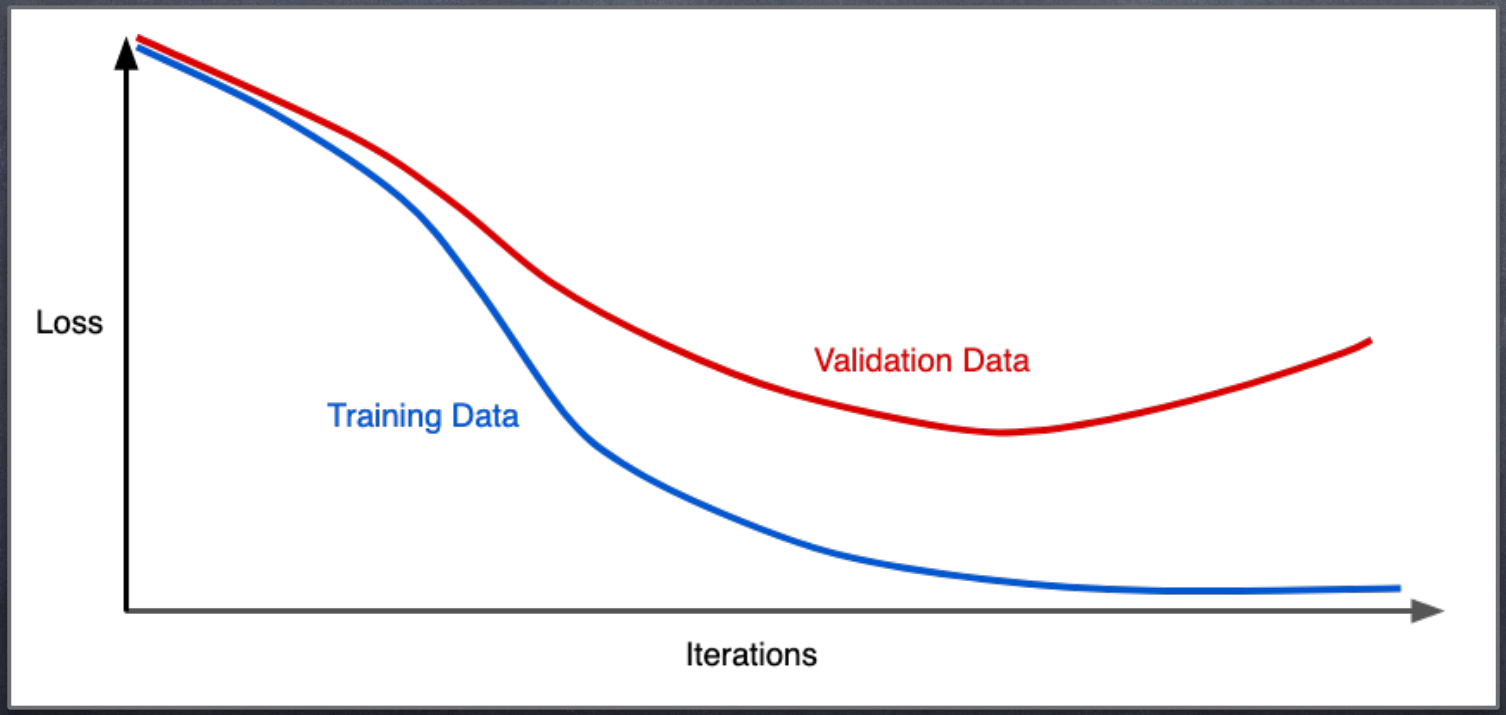


Under-fitting






Appropriate-fitting


Loss
Validation Data
Training Data
Iterations

$$0 = b + \sum_{j=1}^{n} w_j f_j$$

Should we allow all possible weights?

Any preferences?

instead of simply aiming to minimize loss (empirical risk minimization): minimize(Loss(Data|Model))

we'll now minimize loss+complexity, which is called **structural risk minimization**:

minimize(Loss(Data|Model) + complexity(Model))

two terms:

the **loss term**, which measures how well the model fits the data,

the **regularization term**, which measures model complexity

- Generally, we don't want huge weights
- If weights are large, a small change in a feature can result in a large change in the prediction
- Also gives too much weight to any one feature
- Might also prefer weights of 0 for features that aren't useful

$$\text{argmin}_{w,b} \sum_{i=1}^{n} loss(yy') + \lambda \; regularizer(w,b)$$

sum of the weights

$$r(w,b) = \sum_{w_j} |w_j|$$

sum of the squared weights

$$r(w,b) = \sqrt{\sum_{w_j} |w_j|^2}$$

$$w_j = w_j + \eta y_i x_{ij} \exp(-y_i(w \cdot x_i + b)) - \eta \lambda w_j$$

learning rate

direction to update

constant: how far from wrong

regularization

L1 Regularizer

$$\text{argmin}_{w,b} \sum_{i=1}^{n} \exp(-y_i(w \cdot x_i + b)) + \|w\|$$

$$\frac{d}{dw_j} objective = \frac{d}{dw_j} \sum_{i=1}^{n} \exp(-y_i(w \cdot x_i + b)) + \lambda \|w\|$$

$$= -\sum_{i=1}^{n} y_i x_{ij} \exp(-y_i(w \cdot x_i + b)) + \lambda sign(w_j)$$

L1 Regularization

$$w_j = w_j + \eta y_i x_{ij} \exp(-y_i(w \cdot x_i + b)) - \eta\lambda sign(w_j)$$

learning rate

direction to update

constant: how far from wrong

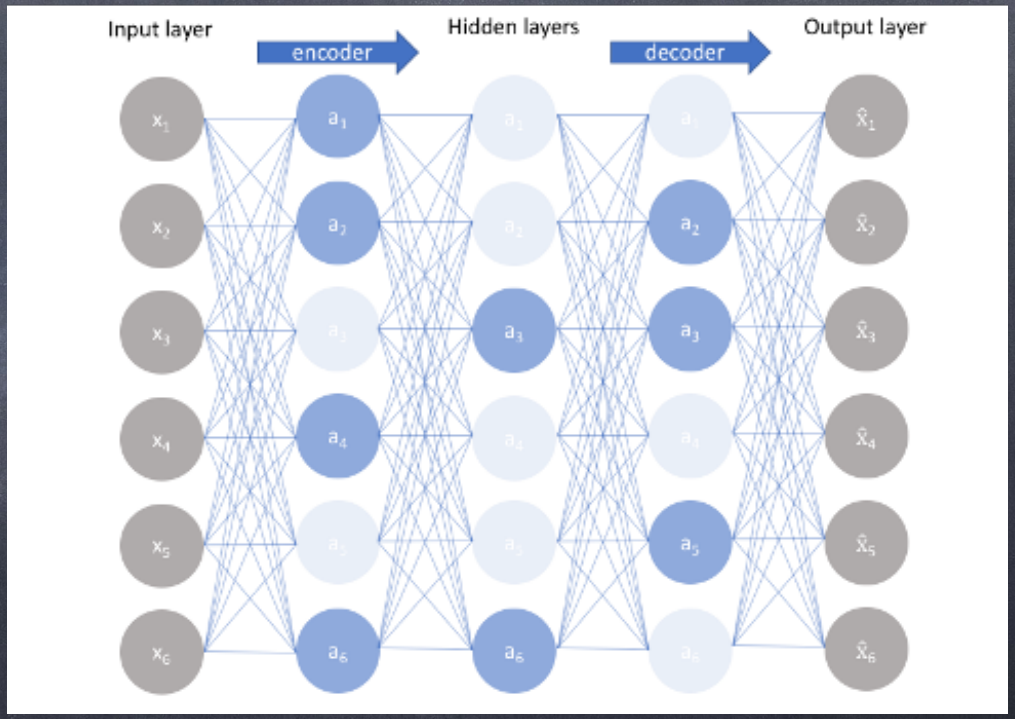
regularization

If w_i is positive, reduces by a constant

If w_i is negative, increases by a constant

In other words ...

- Construct a loss function to penalize activations
- L1 Regularization penalizes the absolute value of the vector of activations



Lambda value

- The goal is to strike the right balance between simplicity and training-data fit:
- If your lambda value is too high, your model will be simple, but you run the risk of underfitting your data. Your model won't learn enough about the training data to make useful predictions.
- If your lambda value is too low, your model will be more complex, and you run the risk of overfitting your data. Your model will learn too much about the particularities of the training data, and won't be able to generalize to new data.

Dropout and DropConnet

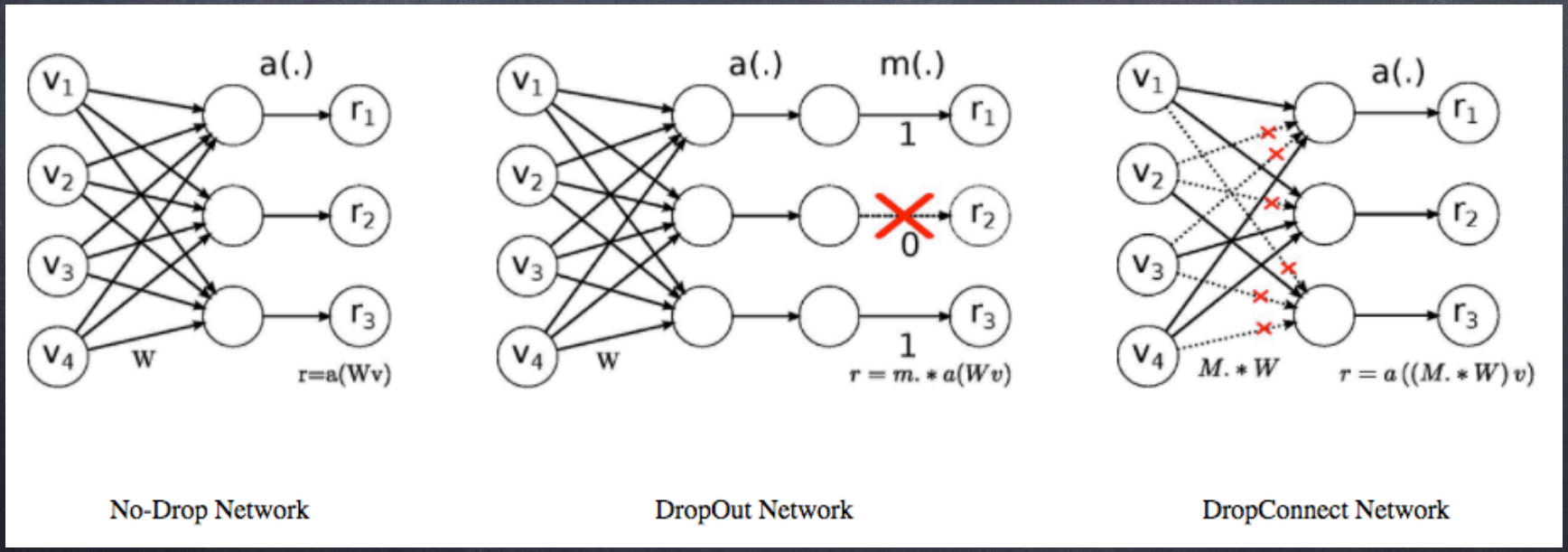
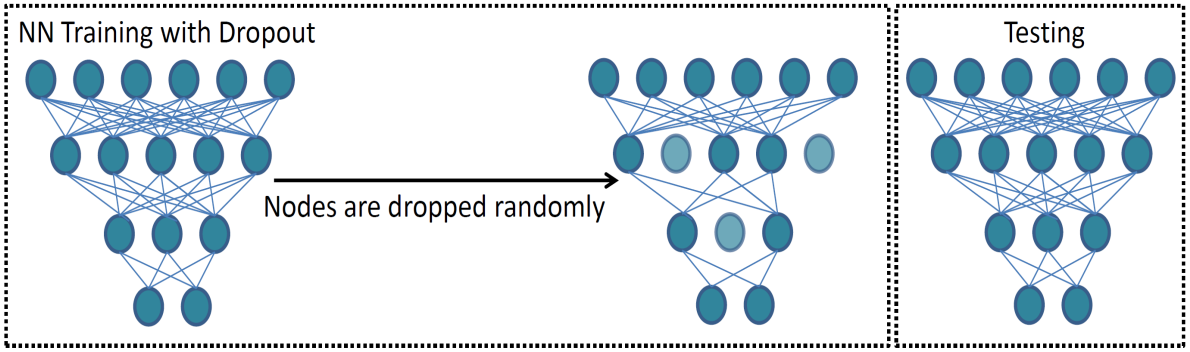


Image: Wan et al.

Dropout as regularization



- Randomly drops nodes at every iteration of the training.
- Trained model behave as an ensemble of multiple models.

Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." *The Journal of Machine Learning Research* 15.1 (2014): 1929-1958.

Dropout and DropConnet

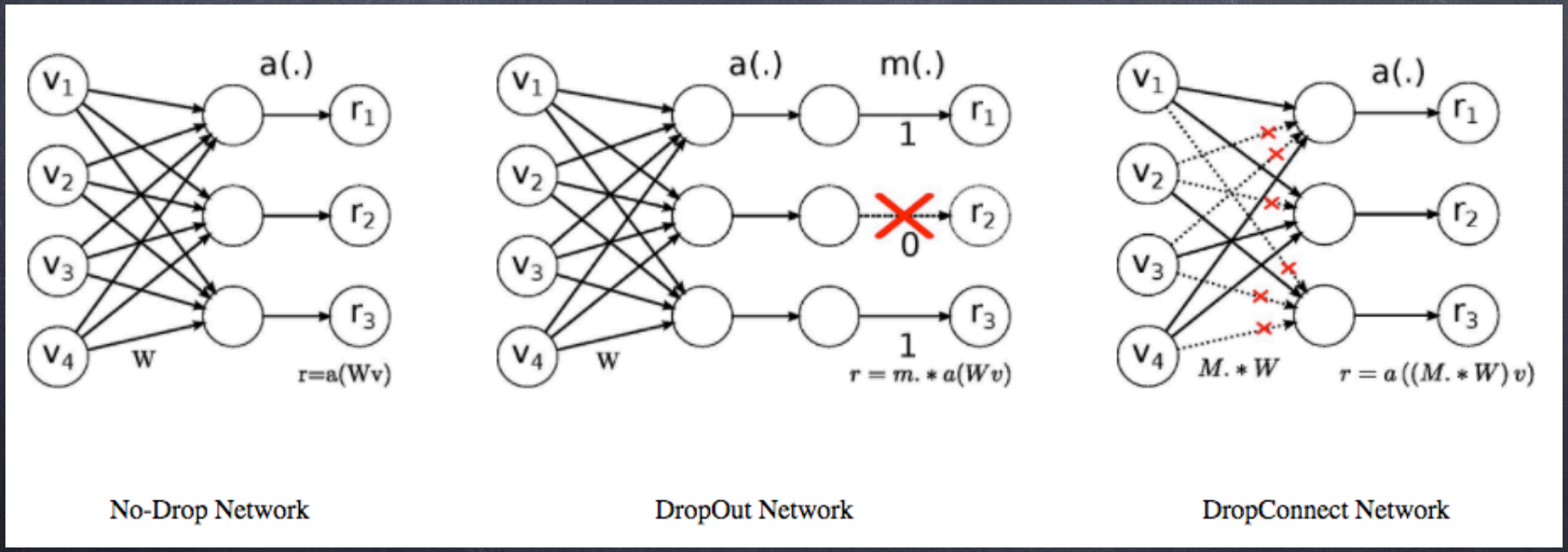


Image: Wan et al.

From Supervised Classification to Unsupervised feature learning

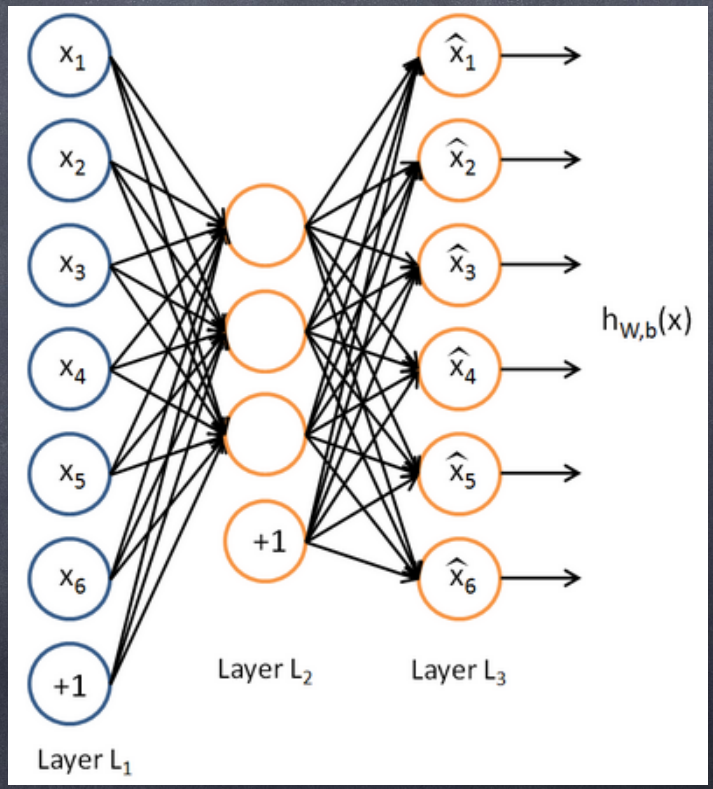
- Traditional neural network, including CNN: supervised approach for classification
- Different layers in the neural network can be viewed as "feature representation" of input data

From Supervised Classification to Unsupervised feature learning

- If input is raw data (huge amount of data available for training) and we "magically" allow our network to learn the "features" in an unsupervised manner

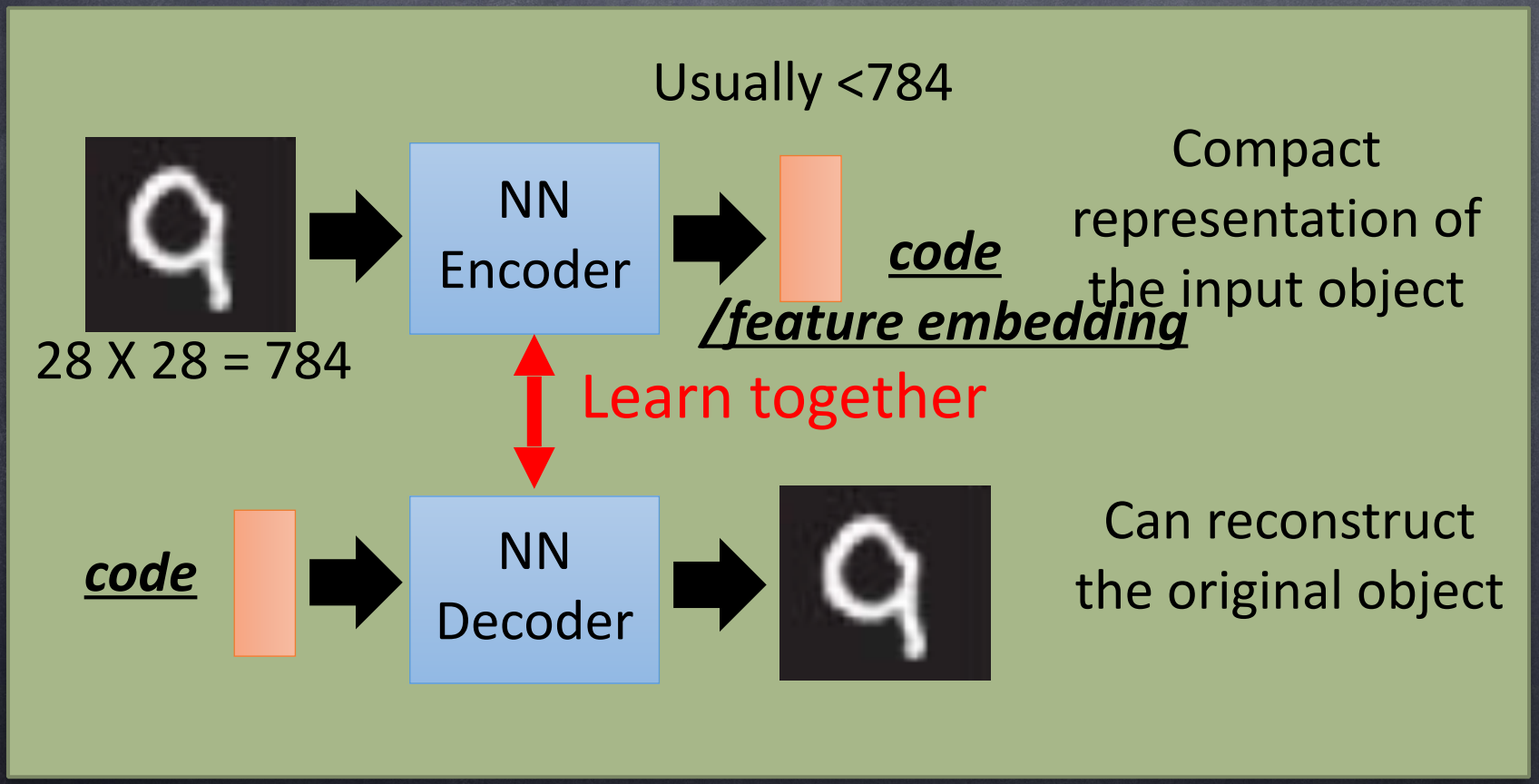
Let us take a look at simple Deep Architecture

- Auto-encoders
- Two parts: encoding and decoding
- Input layer: raw data X
- One hidden layer (encoding): feature learner
- Output layer (decoding): reconstruct $\hat{X}$ such that $||X-\hat{X}||^2$ is minimum



Credit: Andrew Ng

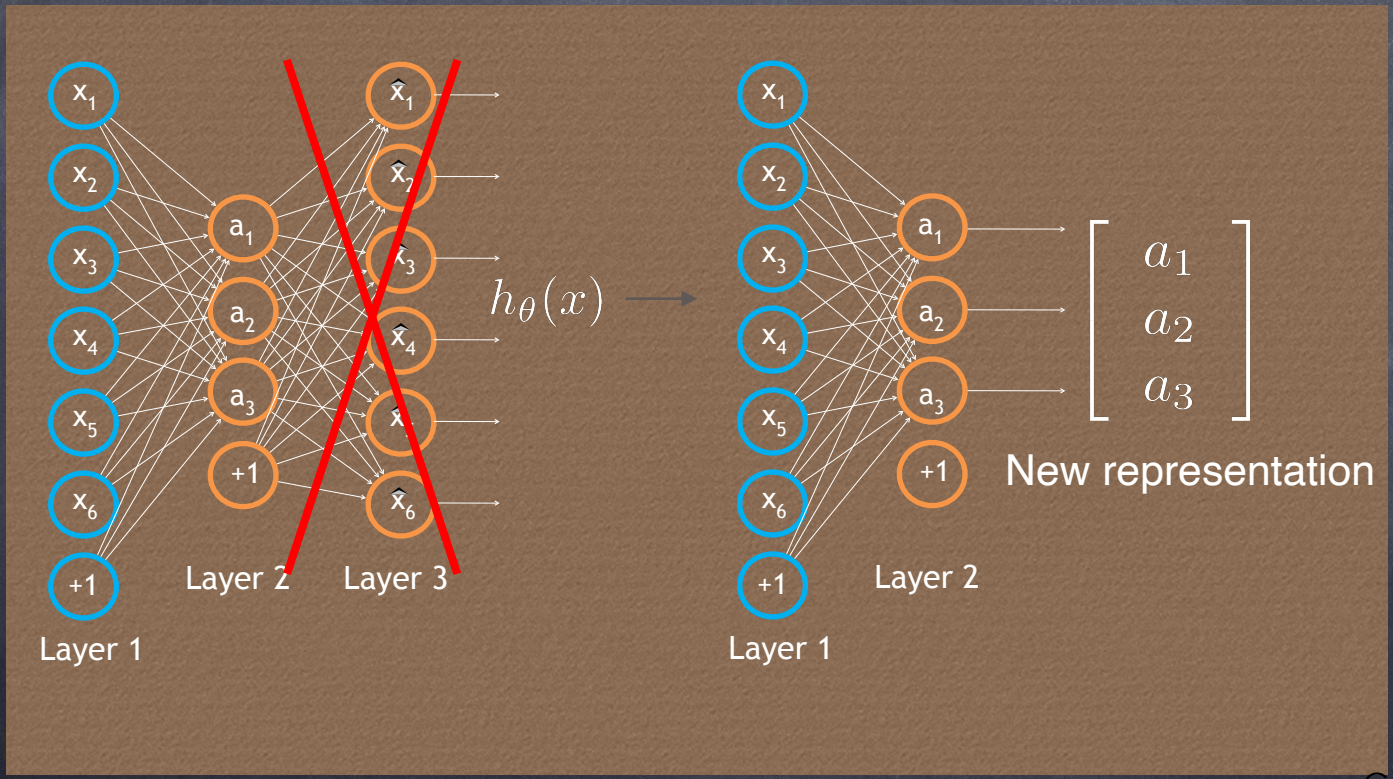
Auto-encoder



What this network is doing?

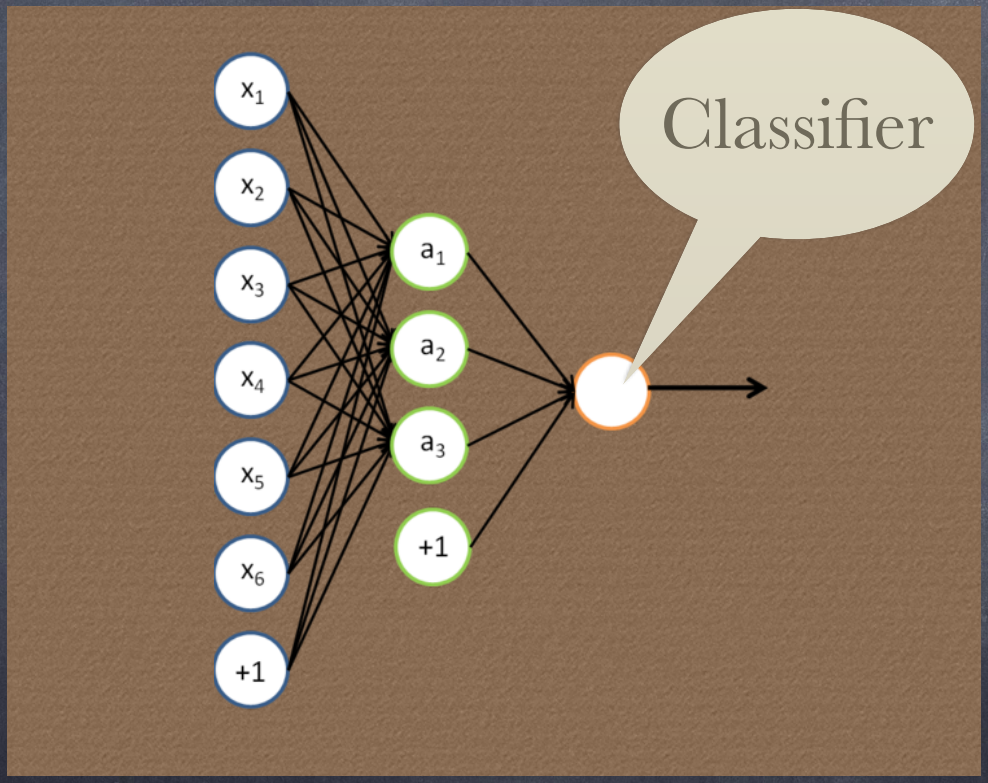
- a1 = sum(w_i*x_i)
- xcap = sum(w_1i*a_i)
- xcap = sum(w_1i*(sum(w_i*x_i))
- ||x - xcap||_2

Autoencoder



Credit: Andrew Ng

Autoencoder



Credit: Andrew Ng