# Introduction to Machine Learning in Geosciences GEO371/GEO398D.1

## Artificial Neural Networks

Mrinal K. Sen
Geosciences
UT Austin

October 3, 2023

# Content

- Introduction
- History
- Mathematical Model
- Training and prediction
- Bayesian NN

https://www.tutorialspoint.com/artificial_neural_network/artificial_neural_network_basic_concepts.htm

https://sebastianraschka.com/books.html

# Introduction

- Linear classifiers (e.g., logistic regression) classify inputs based on linear combinations of features $x_i$. [see my lecture on linear regression]
- Many decisions involve non-linear functions of the input.
- We would like to construct non-linear discriminative classifiers that utilize functions of input variables.
- Use a large number of simpler functions
  - If these functions are fixed (Gaussian, sigmoid, polynomial basis functions), then optimization still involves linear combinations of (fixed functions of) the inputs.
  - Or, we can make these functions depend on additional parameters ! need an efficient method of training extra parameters.

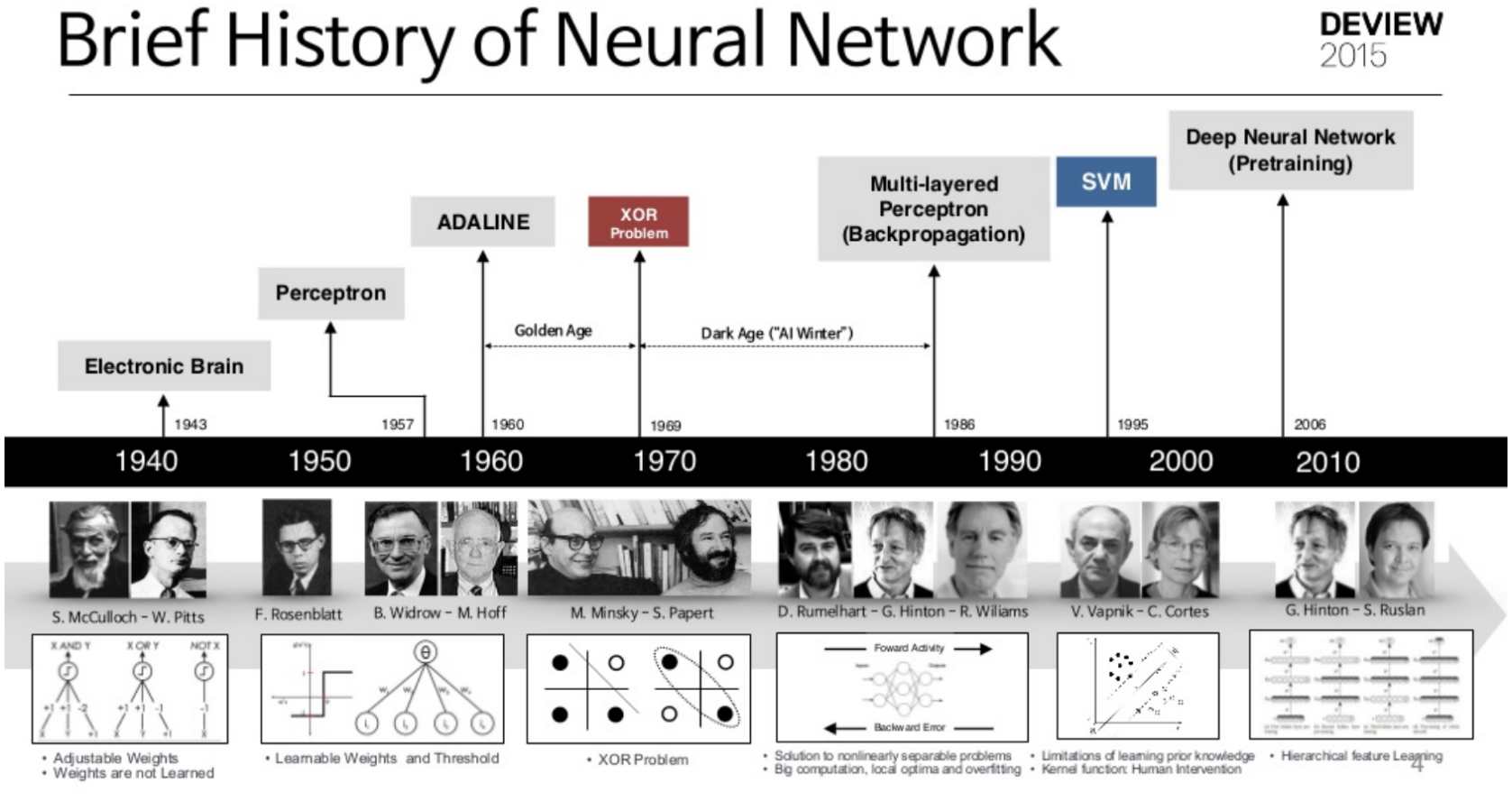
https://www.cs.toronto.edu/~urtasun/courses/CSC411_Fall16/10_nn1.pdf

# Introduction

- Many machine learning methods inspired by biology, e.g., the (human) brain.
- Our brain has nearly $10^{11}$ neurons, each of which communicates (is connected) to nearly $10^4$ other neurons.
- Parallel Computing devices - an attempt to model the brain.
- Efficient computing system - central theme borrowed from Biological neural networks

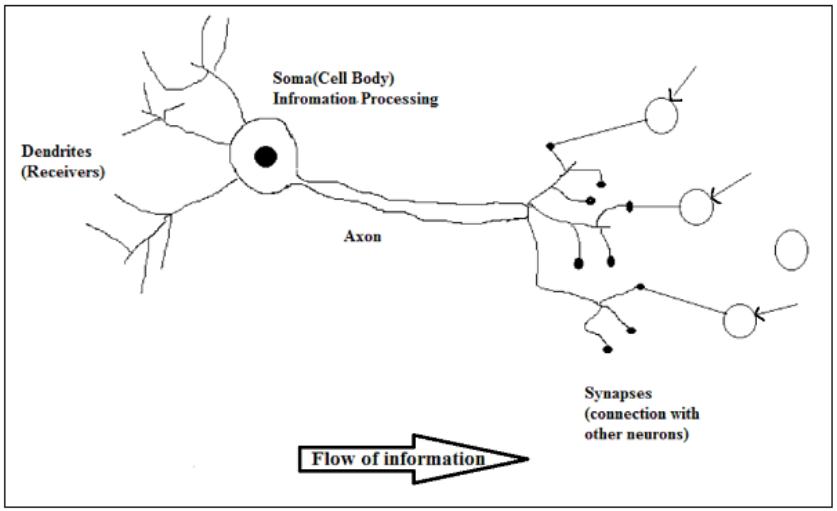
- 1940s - The beginning of Neural Networks (Electronic Brain)
- 1950s and 1960s - The first golden age of Neural Networks (Perceptron)
- 1970s - The winter of Neural Networks (XOR problem)
- 1980s - Renewed enthusiasm (Multilayered Perceptron, backpropagation)
- 1990s - Subfield of Radial Basis Function Networks was developed
- 2000s - The power of Neural Networks Ensembles  Support Vector Machines is apparent
- 2006 - Hinton presents the Deep Belief Network (DBN)
- 2009 - Deep Recurrent Neural Network
- 2010 - Convolutional Deep Belief Network (CDBN)
- 2011 - Max-Pooling CDBN

# History of ANN



# Biological Neuron

- **Neuron**: a special biological cell that processes information.




- Dendrites: They are tree-like branches, responsible for receiving the information from other neurons it is connected to.
- Soma: It is the cell body of the neuron and is responsible for processing of information, they have received from dendrites.
- Axon: It is just like a cable through which neurons send the information.
- Synapses: It is the connection between the axon and other neuron dendrites.

# A Mathematcal Model of a Neuron

- Neural networks define functions of the inputs (hidden features), computed by neurons.
- Artificial neurons are called **units**.

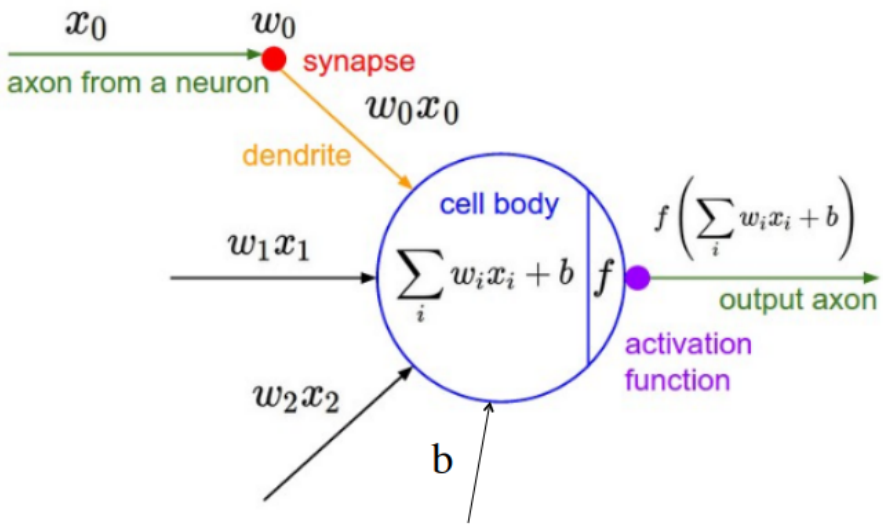


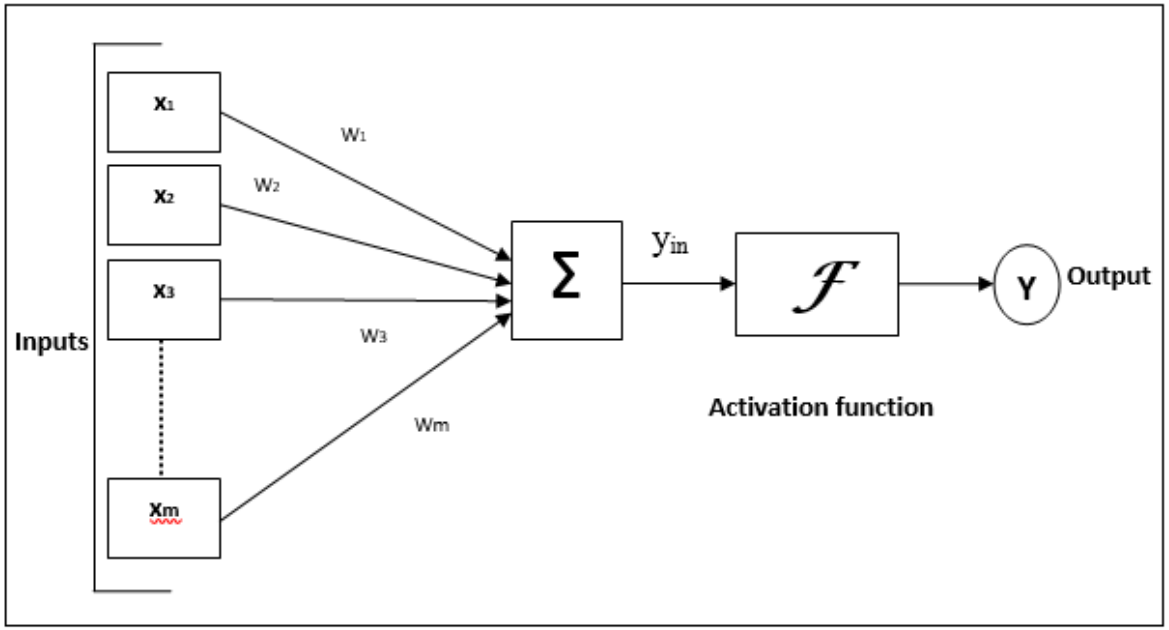

Figure: A mathematical model of the neuron in a neural network

[Pic credit: http://cs231n.github.io/neural-networks-1/]

# A Mathematcal Model of a Neuron

## Model of Artificial Neural Network

The following diagram represents the general model of ANN followed by its processing.




$$y_{in} = \sum_{i}^{m} x_i w_i.$$

$$Y = f(y_{in}).$$

# Network Topology

## Feedforward ANN

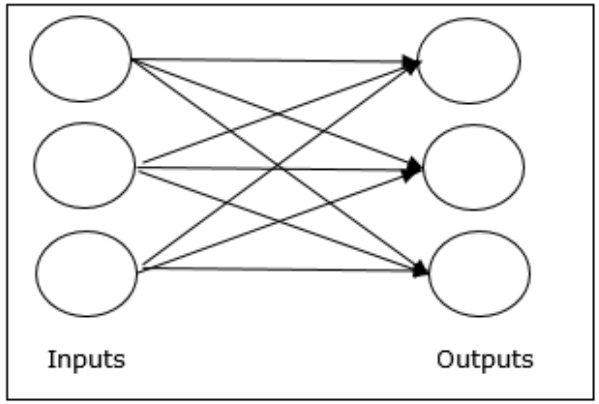



Figure: Single Layer

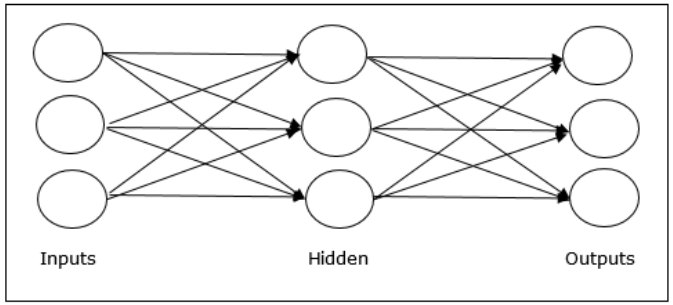
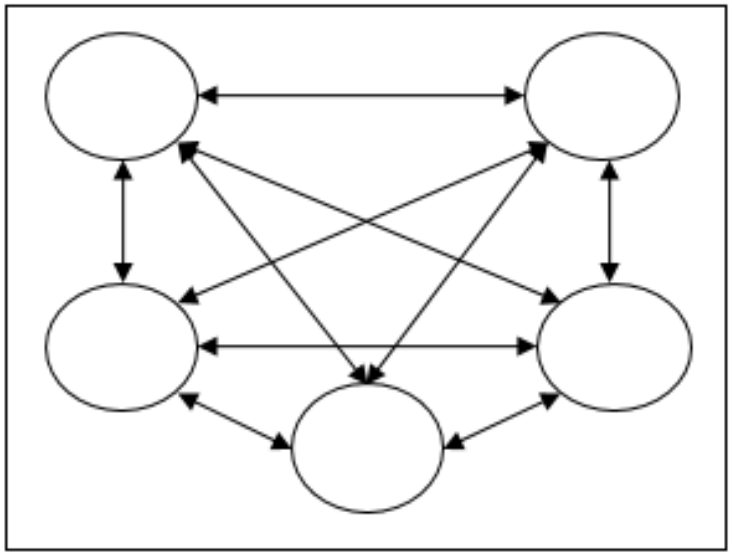



Figure: Multilayer

# Network Topology

## Feedback Network

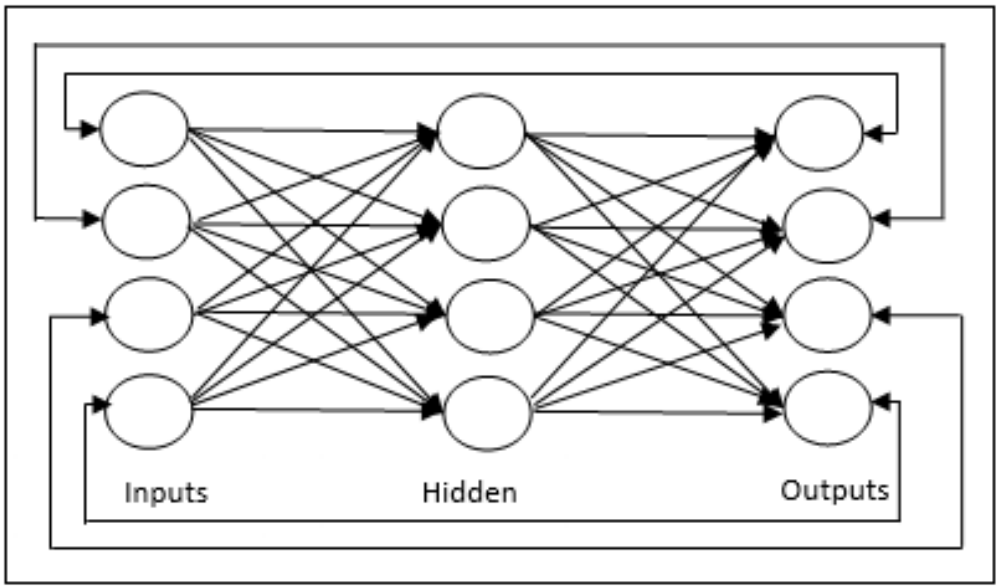
- Recurrent networks  These are feedback networks with closed loops.
  - **Fully recurrent network** – It is the simplest neural network architecture because all nodes are connected to all other nodes and each node works as both input and output.



# Network Topology

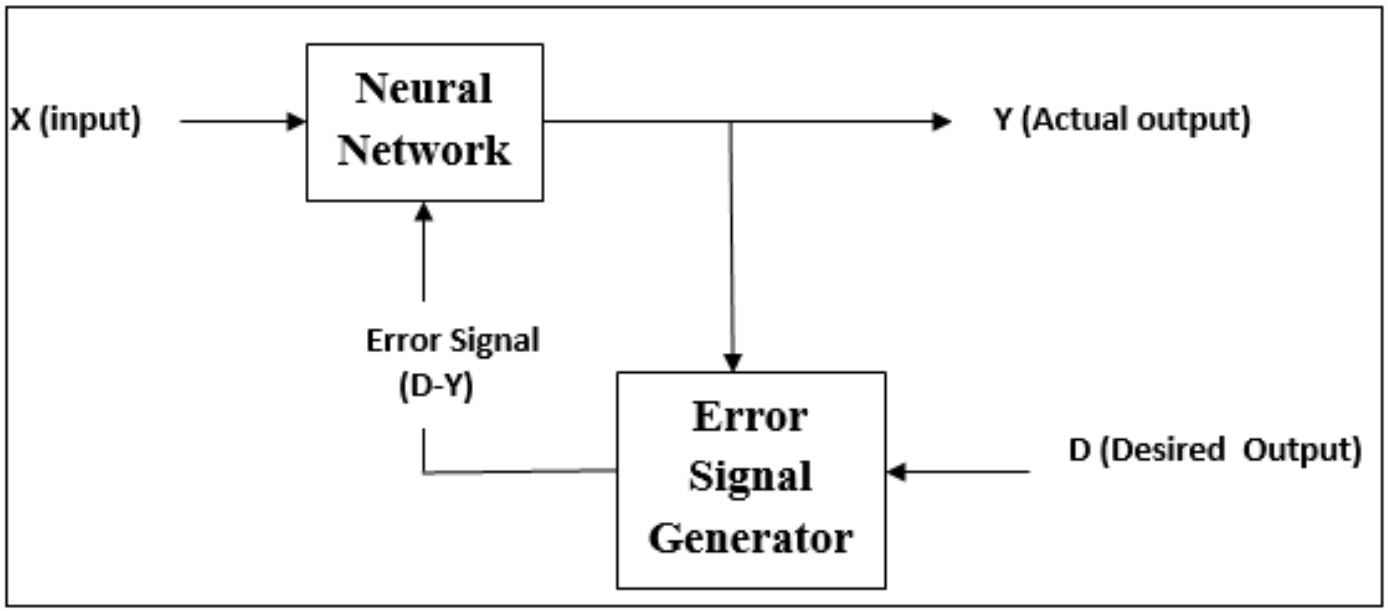
## Feedback Network

- Recurrent networks  These are feedback networks with closed loops.
  - **Jordan network** – It is a closed loop network in which the output will go to the input again as feedback as shown in the following diagram.



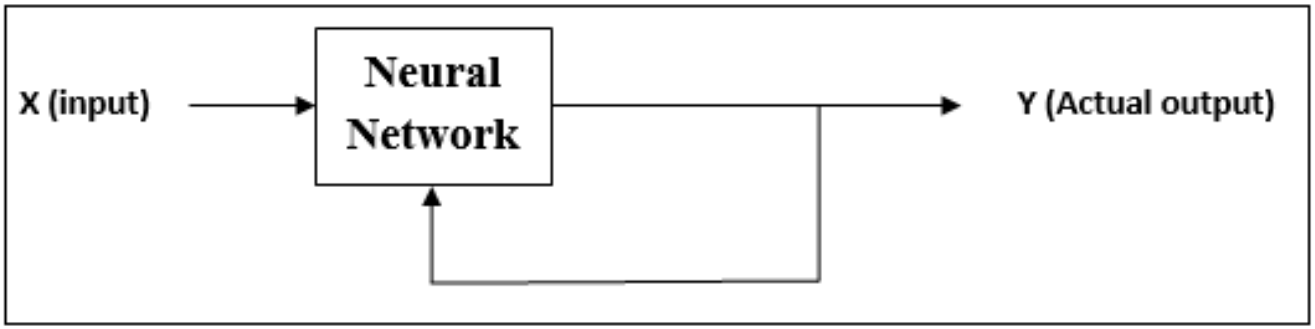
# Learning/Weight Updates

## Supevised Learning



# Learning/Weight Updates

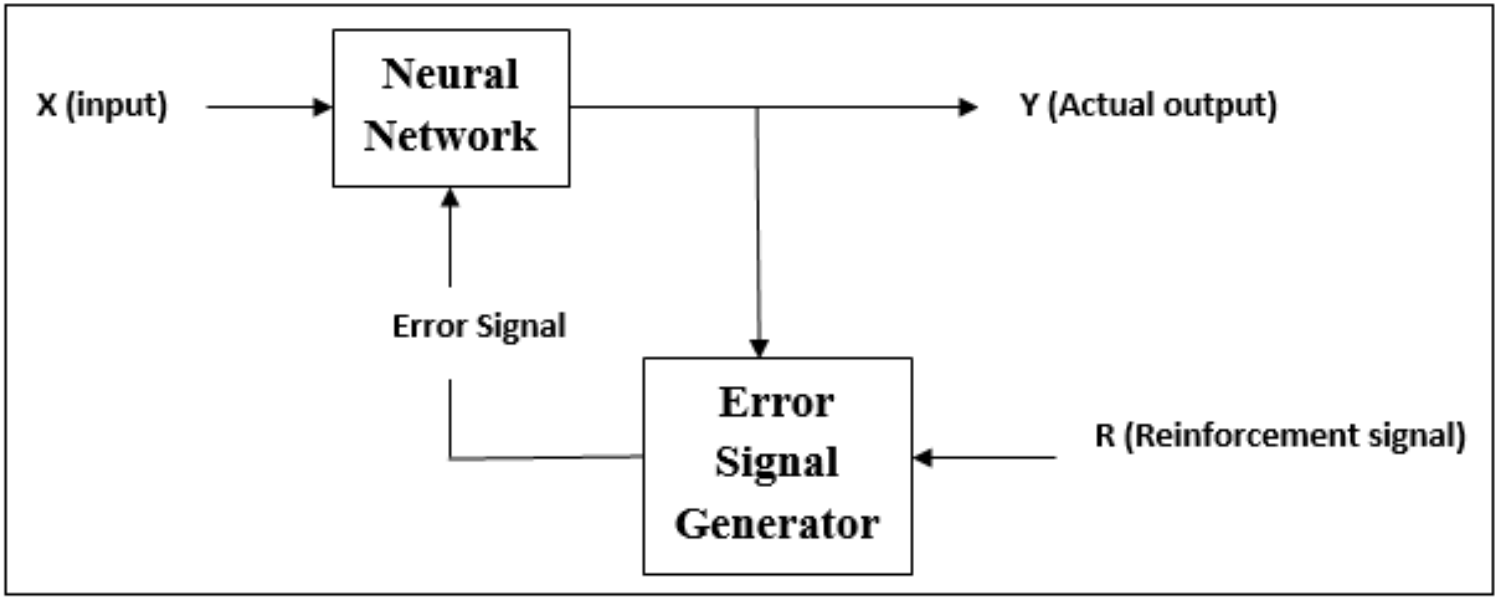
## Unsupevised Learning




- Hopefield NN
- Autoencoder

# Learning/Weight Updates

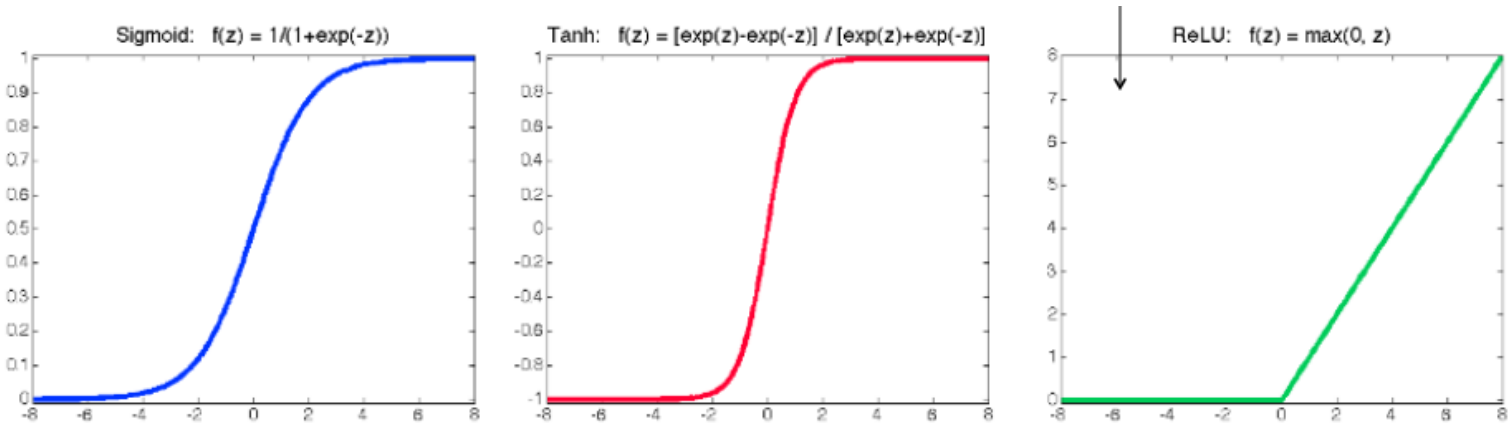
## Reinforment Learning




- During the training of network under reinforcement learning, the network receives some feedback from the environment. This makes it somewhat similar to supervised learning. However, the feedback obtained here is evaluative not instructive, which means there is no teacher as in supervised learning. After receiving the feedback, the network performs adjustments of the weights to get better critic information in future.

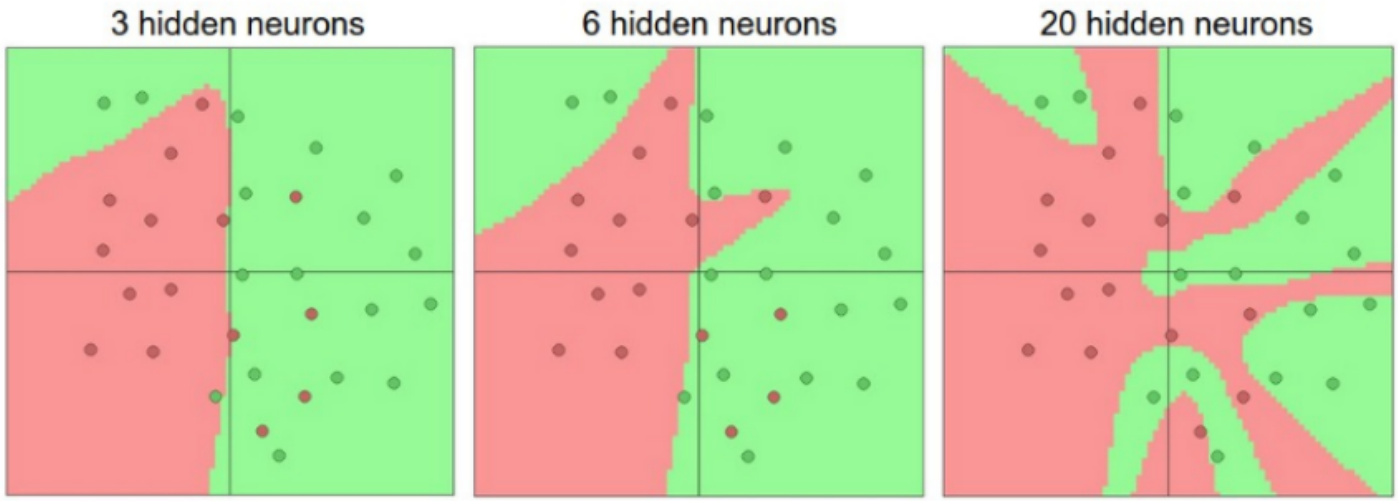
# Activation Functions

- **Sigmoid:** $F(x) = \frac{1}{1+\exp(-x)}$.
- **tanh** $\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$.
- **ReLU** (Rectified Linear Unit): $\mathrm{ReLU}(x) = max(0, x)$.



# Representation Power

A feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly. — Ian Goodfellow, DLB




- The capacity of the network increases with more hidden units and more hidden layers

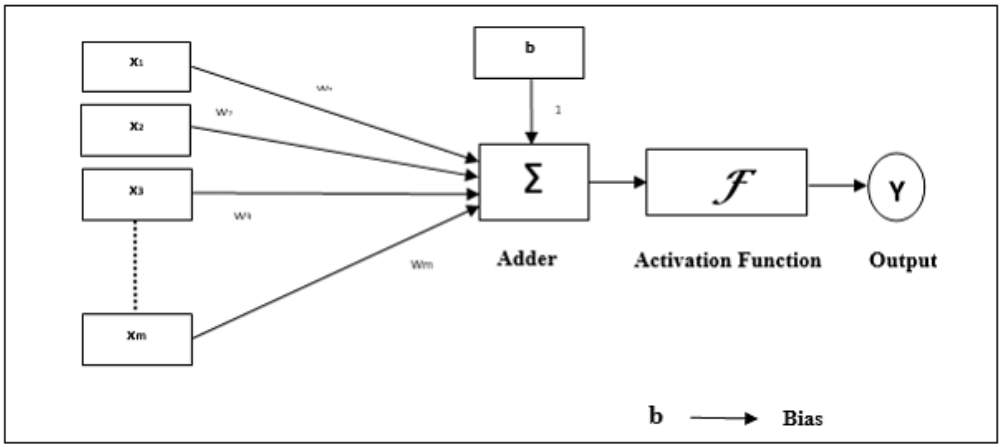
# Representation Power

- Neural Networks are POWERFUL, it's exactly why with recent computing power there is a renewed interest in them.
- "With great power comes great overfitting." – Boris Ivanovic, 2016

# Perceptron

- Developed by Frank Rosenblatt by using McCulloch and Pitts model
- Basic operational unit of artificial neural networks.
- Employs supervised learning rule and is able to classify the data into two classes.

- https://www.tutorialspoint.com/artificial_neural_network/artificial_neural_network_supervised_learning.htm




Perceptron thus has the following three basic elements −

- **Links** − It would have a set of connection links, which carries a weight including a bias always having weight 1.
- **Adder** − It adds the input after they are multiplied with their respective weights.
- **Activation function** − It limits the output of neuron. The most basic activation function is a Heaviside step function that has two possible outputs. This function returns 1, if the input is positive, and 0 for any negative input.

# Perceptron LEARNING RULE

## The Perceptron Learning Algorithm

Let

$$\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, ..., \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0, 1\})^n$$

1. Initialize $\mathbf{w} := 0^m$ (assume notation where weight incl. bias)
2. For every training epoch:
   A. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$:
      (a) $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]T} \mathbf{w})$
      (b) $\text{err} := (y^{[i]} - \hat{y}^{[i]})$
      (c) $\mathbf{w} := \mathbf{w} + err \times \mathbf{x}^{[i]}$

Sebastian Raschka STAT 479: Deep Learning SS 2019 28

- Output value is the class level predicted by the unit step function!

# Gradient of the Loss function

$$y_{pred} = w_1 x_1 + w_2 x_2 + w_3 x_3 + ..... + w_n x_n + b$$

Loss function

$$L = \frac{1}{2}(y_{actual} - y_{pred})^2$$

$$\frac{\partial L}{\partial y_{pred}} = (y_{actual} - y_{pred})$$

$$\frac{\partial y_{pred}}{\partial x_i} = w_i$$

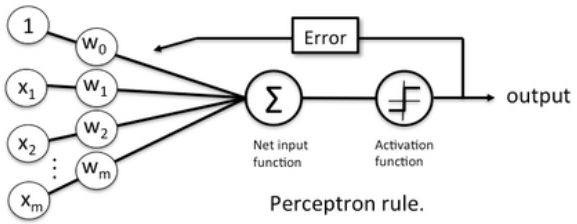
$$\frac{\partial y_{pred}}{\partial w_i} = x_i$$

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y_{pred}} \frac{\partial y_{pred}}{\partial w_i}$$

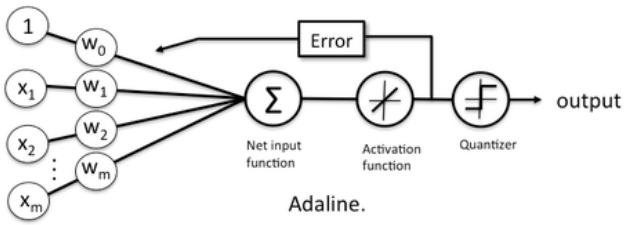
$$\boxed{\frac{\partial L}{\partial w_i} = x_i(y_{actual} - y_{pred})}$$

# ADALINE

- Particularly Interesting - illustrates the key concepts of defininining and minimizing continuous cost functions!
- Groundwork for more advanced ML methods such as *logistic regression, SVM, and regression models*.
- Perception uses a unit step function while Adaline uses a linear activaion function
- The Adaline (Adaptive Linear Element) and the Perceptron are both linear classifiers when considered as individual units. They both take an input, and based on a threshold, output e.g. either a 0 or a 1.
- The main difference between the two, is that a Perceptron takes that binary response (like a classification result) and computes an error used to update the weights, whereas an Adaline uses a continous response value to update the weights (so before the binarized output is produced).
- The fact that the Adaline does this, allows its updates to be more repesentative of the actual error, before it is thresholded, which in turn allows a model to converge more quickly.



Perceptron rule.



Adaline.

# Minimizing Cost Function with Gradient Descent

- The Cost function: $J(\mathbf{W}) = \frac{1}{2}\sum_i (y^i - \phi(x^i))^2$.
- Using GD, we update weights by taking a step in the opposite direction of the gradient, $\nabla J(\mathbf{W})$:

$$\mathbf{W} \leftarrow \mathbf{W} + \Delta\mathbf{W}.$$

- The weight change $\nabla J(\mathbf{W})$ is defined as the negative of the gradient multiplied by the learning rate $\eta$

$$\Delta\mathbf{W} = -\eta\nabla J(\mathbf{W}).$$

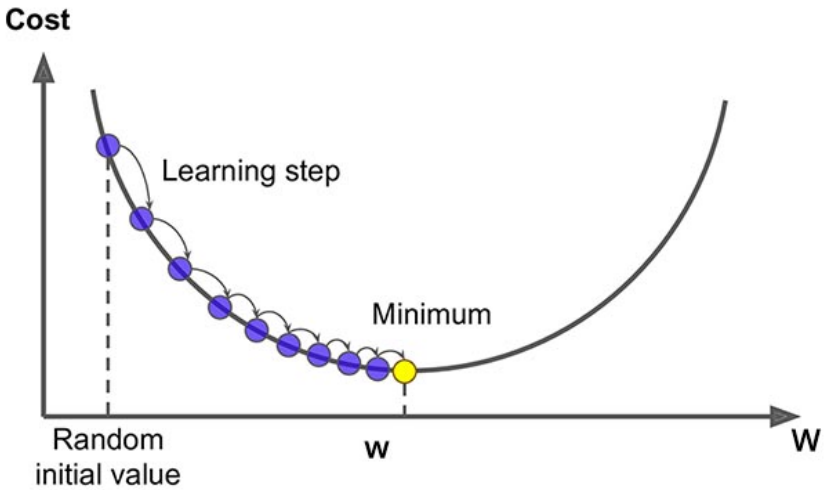
$$\frac{\partial J}{\partial w_i} = -\sum_i (y^i - \phi(x^i))x_j^i.$$

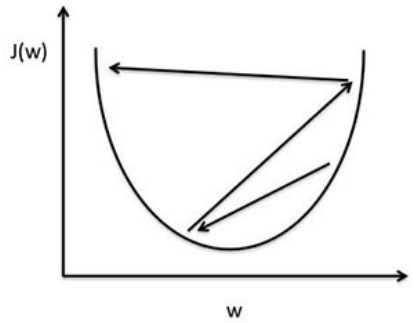
$$\Delta w_j = -\eta\frac{\partial J}{\partial w_i} = \eta\sum_i (y^i - \phi(x^i))x_j^i.$$

# Minimizing Cost Function with Gradient Descent

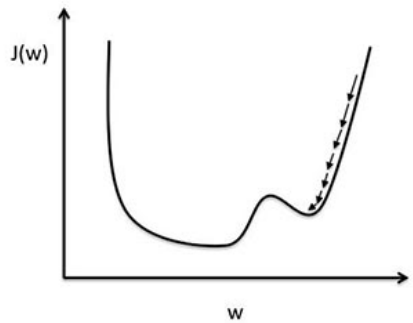
Hyperparameters:

- The learning rate $\eta$
- The number of epochs $=$ *n_iters*




Large Learning Rate

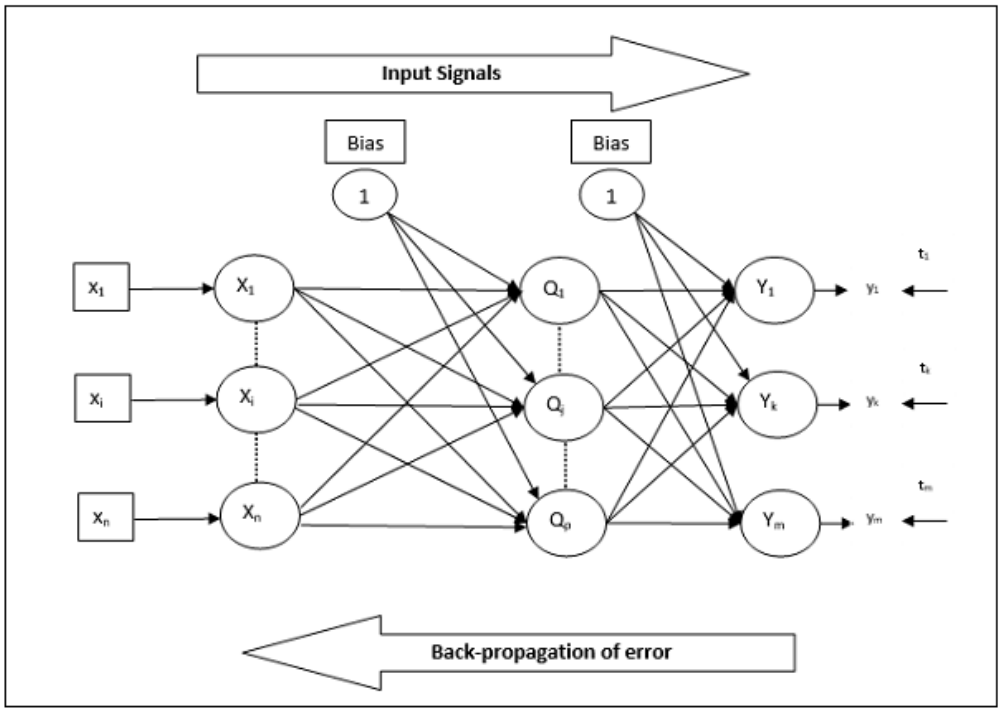



Small Learning Rate

# ANN/DNN

Goals:

- A learning rule that is more robust than the perception
- Combine multiple Neurons and layers of neurons (deep neural nets) to learn more complex decision boundaries (most real world problems are not linear)
- Handle multiple categories (not just binary) in classification.




Back Propagation Neural Network

# Feedforward Neural Network

## High-Level Overview

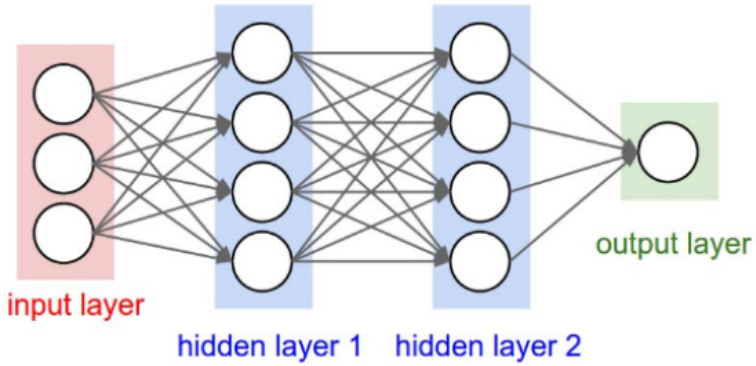



Figure: A 3-layer neural net with 3 input units, 4 hidden units in the first and second hidden layer and 1 output unit

- Naming conventions; a N-layer neural network:
  - $N-1$ layers of hidden units
  - One output layer

[http://cs231n.github.io/neural-networks-1/]

Neuron Breakdown

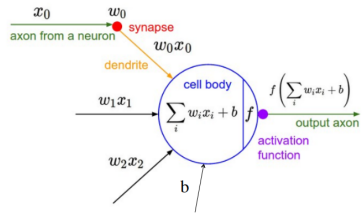



Figure: A mathematical model of the neuron in a neural network

[Pic credit: http://cs231n.github.io/neural-networks-1/]

# Training Neural Network

Find Optimal Weights:

$$w^* = \text{argmin}_{\mathbf{w}} \sum_{n=1}^{N} (\mathbf{o}^n, \mathbf{t}^n),$$

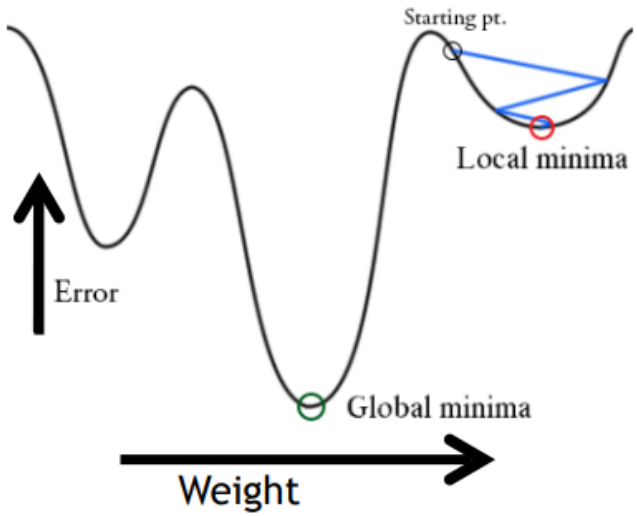
- $\mathbf{o} = f(x; \mathbf{w})$: output of the network.
- Loss function, e.g.,
  - Squared loss: $\sum_k \frac{1}{2}(o_k^{(n)} - t_k^{(n)})^2$, [For Regression]
  - Cross-entropy Loss: $-\sum_k t_k^{(n)} \log o_k^{(n)}$. [For classification]

Gradient Descent Minimization

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{\partial E}{\partial \mathbf{w}^t}$$

## Training Compared to Other Models

- Training Neural Networks is a **NON-CONVEX OPTIMIZATION PROBLEM**.
- This means we can run into many local optima during training.




https://www.cs.toronto.edu/ jlucas/teaching/csc411/lectures/tut5_handout.p

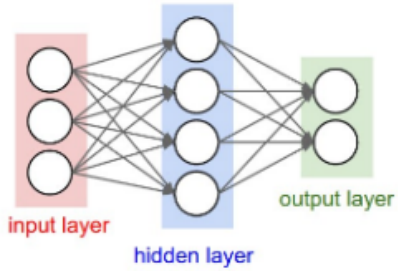
# Training Neural Network

## Training Neural Networks (Implementation)

- We need to first perform a **forward pass**
- Then, we update weights with a **backward pass**

https://www.cs.toronto.edu/ jlucas/teaching/csc411/lectures/tut5_handout.p

# Forward Pass (AKA "Inference")




- Output of the network can be written as:

$$h_j(\mathbf{x}) = f(v_{j0} + \sum_{i=1}^{D} x_i v_{ji})$$

$$o_k(\mathbf{x}) = g(w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj})$$

($j$ indexing hidden units, $k$ indexing the output units, $D$ number of inputs)

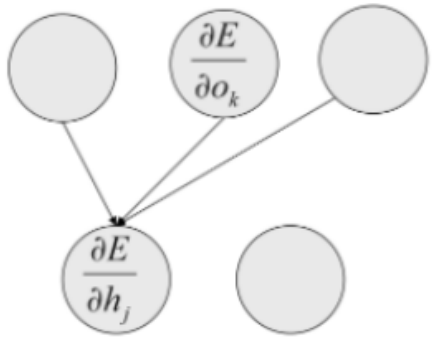
- Activation functions $f$, $g$: sigmoid/logistic, tanh, or rectified linear (ReLU)

$$\sigma(z) = \frac{1}{1+\exp(-z)}, \quad \tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}, \quad \mathrm{ReLU}(z) = \max(0, z)$$

https://www.cs.toronto.edu/ jlucas/teaching/csc411/lectures/tut5_handout.p

# Backward Pass (AKA "Backprop.")

- Compute error derivatives in each hidden layer from error derivatives in layer above. [assign blame for error at $k$ to each unit $j$ according to its influence on k (depends on $w_{kj}$)]




- Use error derivatives w.r.t. activities to get error derivatives w.r.t. the weights.

https://www.cs.toronto.edu/ jlucas/teaching/csc411/lectures/tut5$_h$*andout.p*

## Learning Weights during Backprop

- Do exactly what we've been doing!
- Take the derivative of the error/cost/loss function w.r.t. the weights and minimize via gradient descent!

Gradient descent:

$$\mathbf{w}^{t+1} = \mathbf{w}^{t} - \eta \frac{\partial E}{\partial \mathbf{w}^{t}}$$

where $\eta$ is the learning rate (and $E$ is error/loss)

https://www.cs.toronto.edu/ jlucas/teaching/csc411/lectures/tut5_handout.p

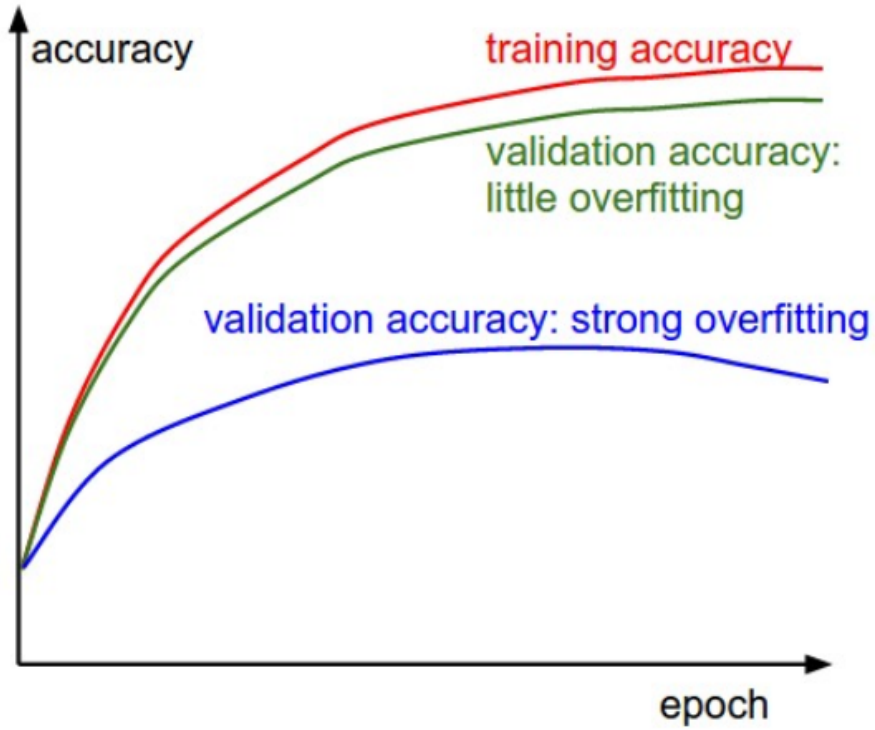
# Useful Derivatives

| name | function | derivative |
|---|---|---|
| Sigmoid | $\sigma(z) = \frac{1}{1+\exp(-z)}$ | $\sigma(z) \cdot (1 - \sigma(z))$ |
| Tanh | $\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$ | $1/\cosh^2(z)$ |
| ReLU | $\mathrm{ReLU}(z) = \max(0, z)$ | $\begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{if } z \leq 0 \end{cases}$ |

https://www.cs.toronto.edu/ jlucas/teaching/csc411/lectures/tut5_handout.p

# Monitoring Accuracy

- Check how your desired performance metrics behaves during training




[http://cs231n.github.io/neural-networks-3/]

# Demonstration

http://neuralnetworksanddeeplearning.com/chap4.html

# Preventing Overfitting

Standard ways to limit the capacity of a neural net:

- Limit the number of hidden units.
- Limit the size of the weights. Weight decay
- Stop the learning before it has time to overfit. Early stop

https://www.cs.toronto.edu/ jlucas/teaching/csc411/lectures/tut5_handout.p

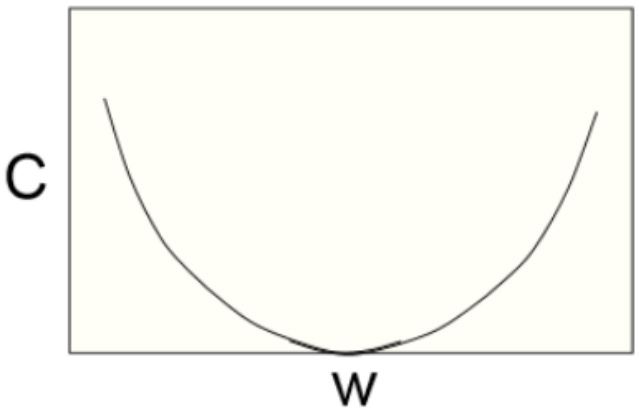
# Limiting the Size of the Weights

Weight-decay involves adding an extra term to the cost function that penalizes the squared weights.

- Keeps weights small unless they have big error derivatives.

$$C = E + \frac{\lambda}{2} \sum_i w_i^2$$

$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$$




$$when \quad \frac{\partial C}{\partial w_i} = 0, \quad w_i = -\frac{1}{\lambda} \frac{\partial E}{\partial w_i}$$

https://www.cs.toronto.edu/ jlucas/teaching/csc411/lectures/tut5_handout.p

# The Effects of Weight-Decay

- It prevents the network from using weights that it does not need
  - This can often improve generalization a lot.
  - It helps to stop it from fitting the sampling error.
  - It makes a smoother model in which the output changes more slowly as the input changes.

- But, if the network has two very similar inputs it prefers to put half the weight on each rather than all the weight on one → other form of weight decay?


https://www.cs.toronto.edu/ jlucas/teaching/csc411/lectures/tut5_handout.p

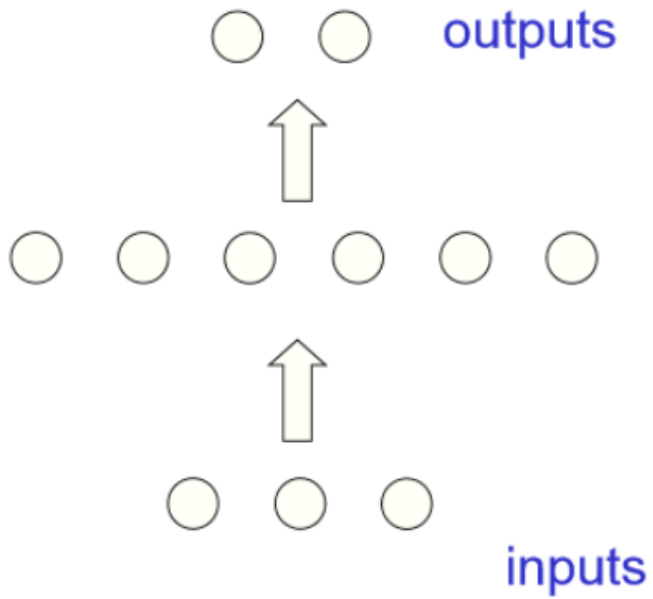
# Early Stopping

- If we have lots of data and a big model, its very expensive to keep re-training it with different amounts of weight decay
- It is much cheaper to start with very small weights and let them grow until the performance on the validation set starts getting worse
- The capacity of the model is limited because the weights have not had time to grow big.

https://www.cs.toronto.edu/ jlucas/teaching/csc411/lectures/tut5$_h$*andout*.*p*

# Why Early Stopping Works

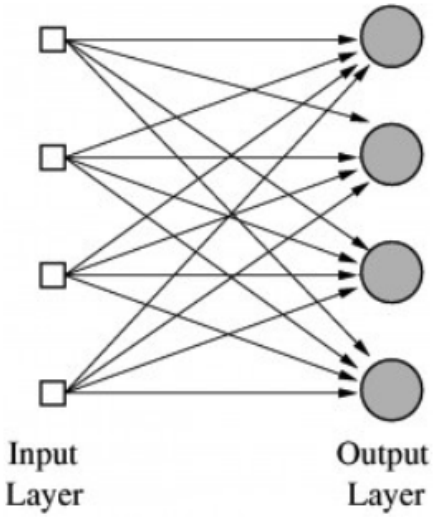
- When the weights are very small, every hidden unit is in its linear range.
  - So a net with a large layer of hidden units is linear.
  - It has no more capacity than a linear net in which the inputs are directly connected to the outputs!
- As the weights grow, the hidden units start using their non-linear ranges so the capacity grows.




https://www.cs.toronto.edu/ jlucas/teaching/csc411/lectures/tut5_handout.p

# Special Case

- What is a single layer (no hiddens) network with a sigmoid act. function?




- Network:

$$o_k(\mathbf{x}) = \frac{1}{1+\exp(-z_k)}$$

$$z_k = w_{k0} + \sum_{j=1}^{J} x_j w_{kj}$$

- Logistic regression!

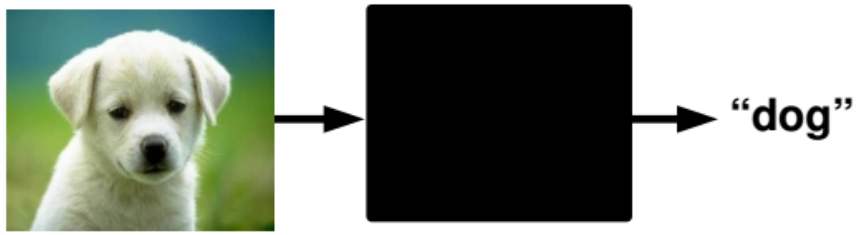
# Choosing Activation and Loss Functions

- When using a neural network for regression, sigmoid activation and MSE as the loss function work well.
- For classification, if it is a binary (2-class) problem, then cross-entropy error function often does better (as we saw with logistic regression).

# Why Deep?

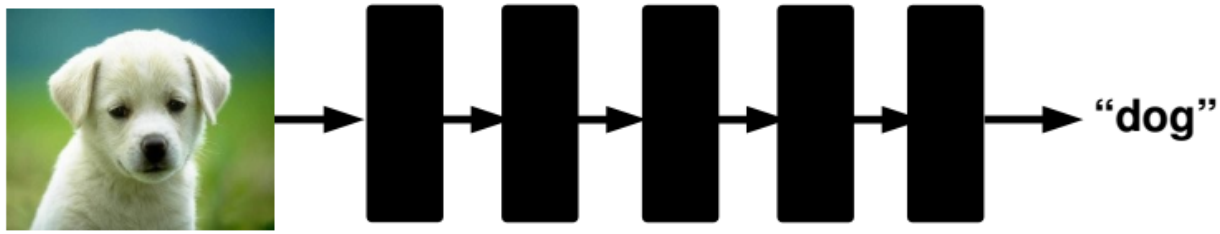
## Supervised Learning: Examples

**Classification**




## Supervised Deep Learning

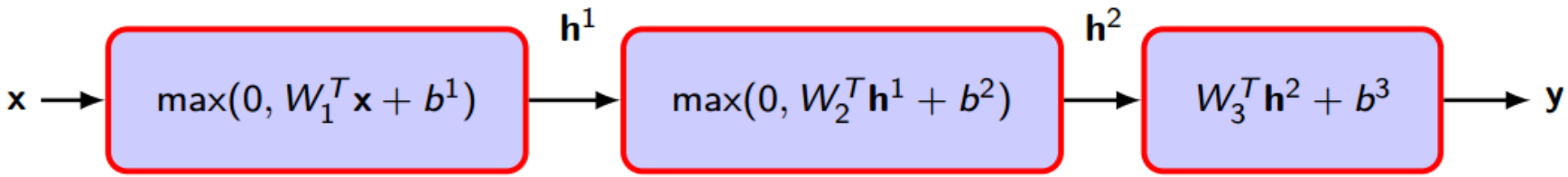
**Classification**




[Picture from M. Ranzato]

# Deep Learning

- Deep learning uses composite of simple functions (e.g., ReLU, sigmoid, tanh, max) to create complex non-linear functions
- Note: a composite of linear functions is linear!
- Example: 2 hidden layer NNet (now matrix and vector form!) with ReLU as nonlinearity



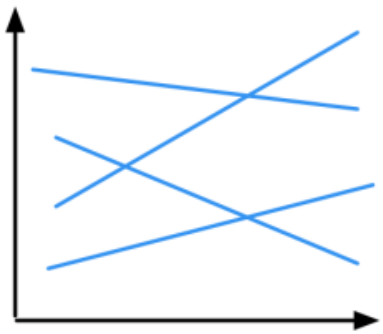

- **x** is the input
- **y** is the output (what we want to predict)
- $\mathbf{h}^i$ is the $i$-th hidden layer
- $W_i$ are the parameters of the $i$-th layer

# What about Uncertainty?

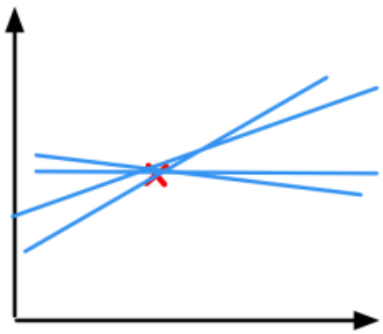
**Why do we care for uncertainty?**

- Instead of considering a single answer to a question, we will have an entire distribution of answers.
- We will be able to put error bars on the output of the work.
- Smooth out the predictions by averaging over lots of plausible explanations (just like ensembles!)
- Assign confidences to predictions (i.e. calibration)
- Make more robust decisions (e.g. medical diagnosis)
- Guide exploration (focus on areas you're uncertain about)
- Detect out-of-distribution examples, or even adversarial example!

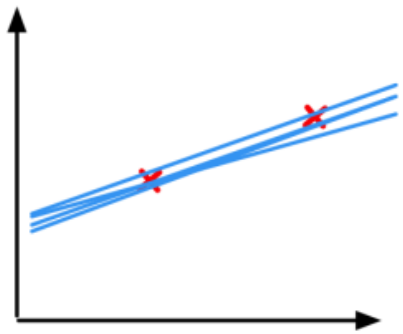
# What about Uncertainty?



no observations

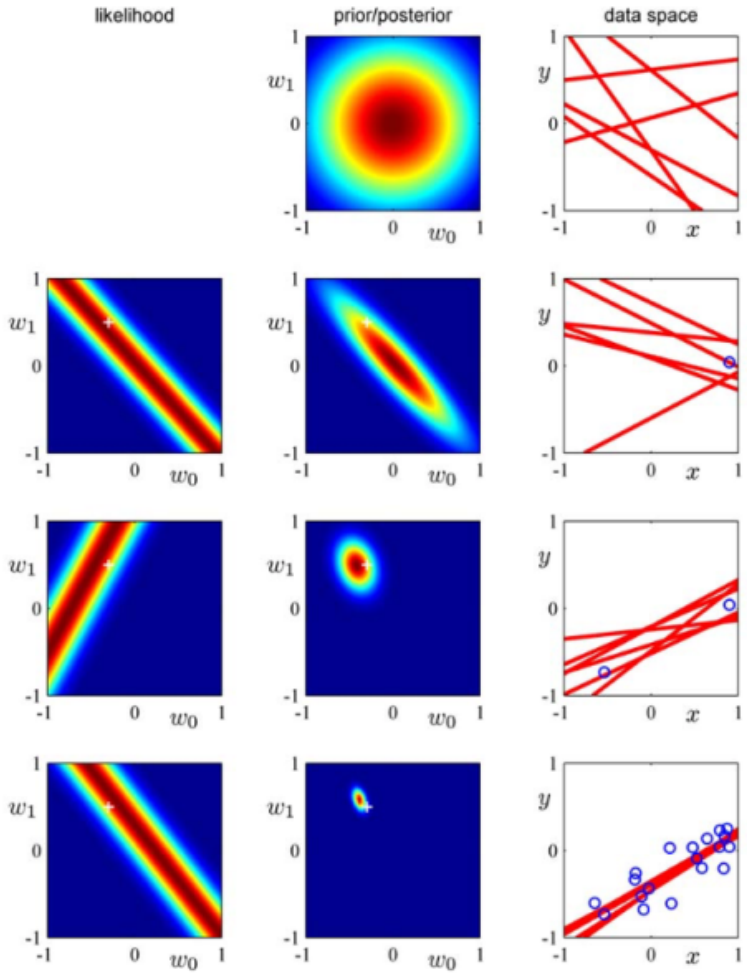


one observation



two observations

# What about Uncertainty?

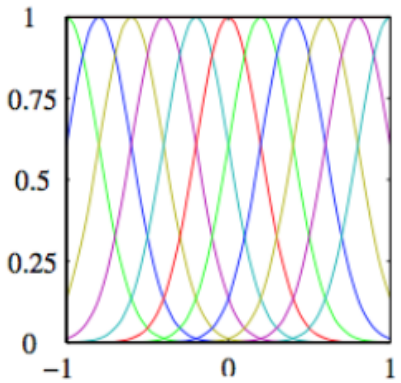



— Bishop, Pattern Recognition and Machine Learning

# What about Uncertainty?

- Example with radial basis function (RBF) features

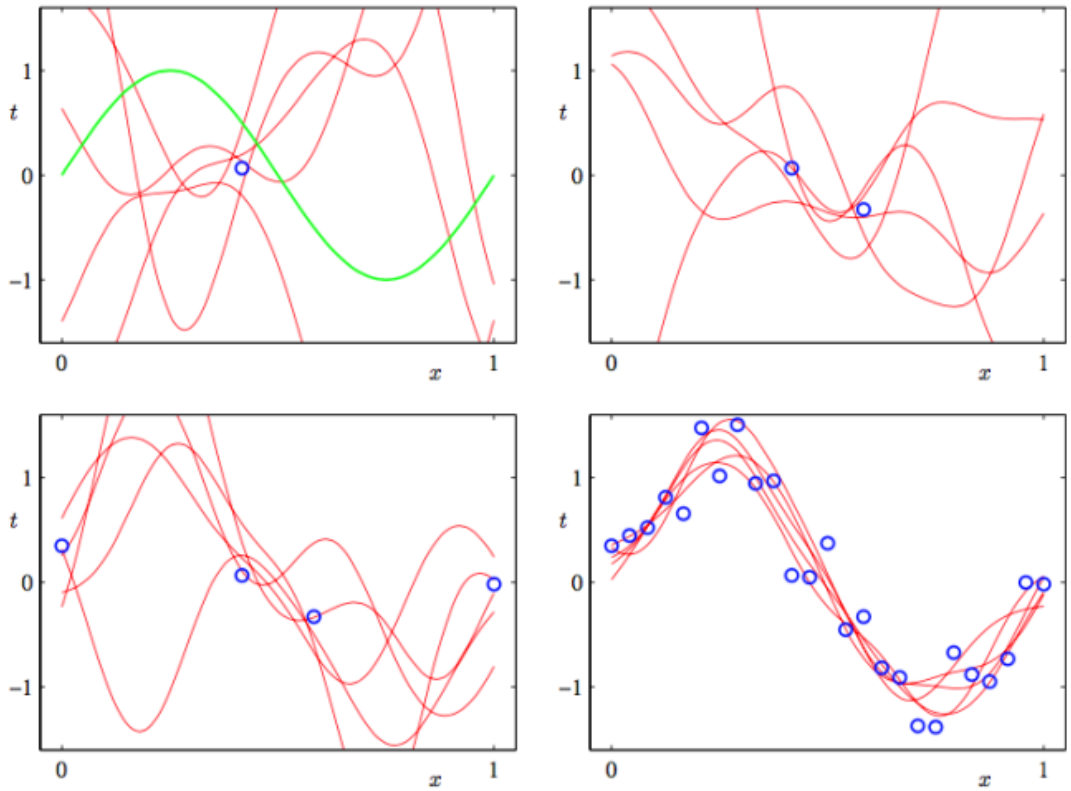
$$\phi_j(x) = \exp\left(-\frac{(x-\mu_j)^2}{2s^2}\right)$$




— Bishop, Pattern Recognition and Machine Learning

# What about Uncertainty?

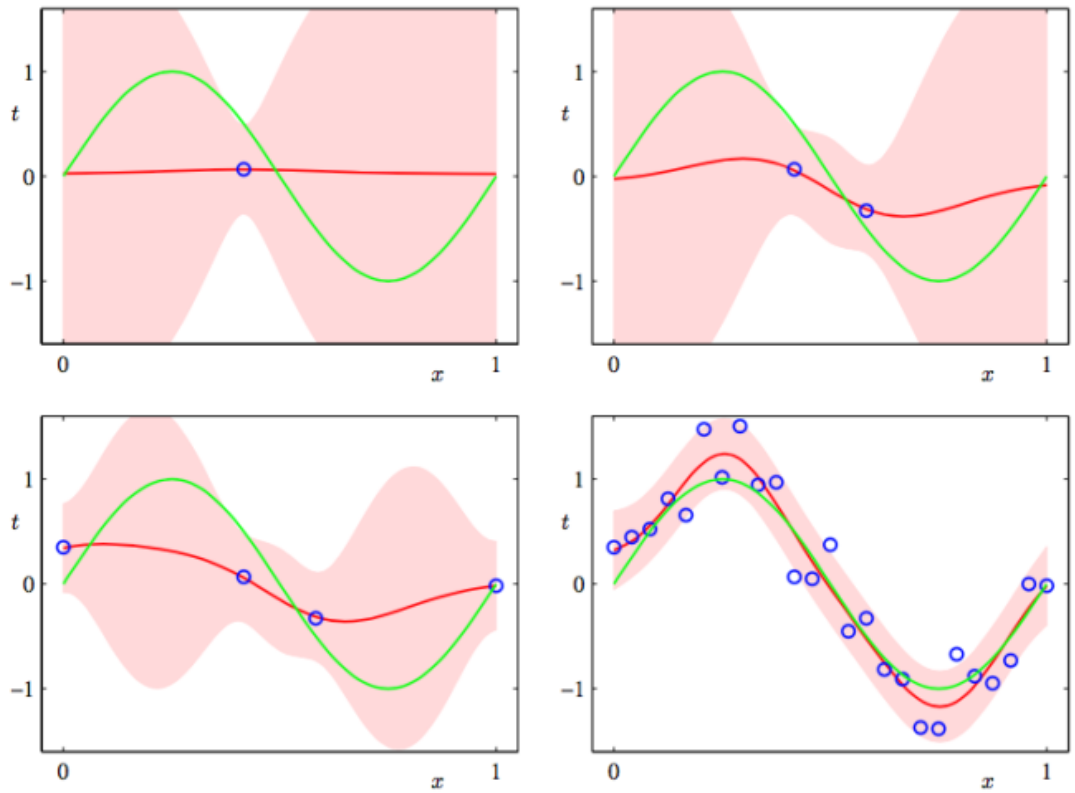
Functions sampled from the posterior:




— Bishop, Pattern Recognition and Machine Learning

# What about Uncertainty?

Here we visualize confidence intervals based on the posterior predictive mean and variance at each point:



# What about Uncertainty?

- Fixed basis functions are limited. Can we combine the advantages of neural nets and Bayesian models?

## Bayesian Inference?

- $p(\mathbf{w})$ : **Prior probability** of the parameter $\mathbf{w}$ independent of the data being analyzed.
- $p(\mathbf{d}|\mathbf{w})$ : **Likelihood** - the probability of data $\mathbf{d}$ given $\mathbf{w}$.
- We use Bayes' rule to obtain the posterior probability of $\mathbf{w}$ given the data $\mathbf{d}$,

$$p(\mathbf{w}|\mathbf{d}) = \frac{p(\mathbf{d}|\mathbf{w})p(\mathbf{w})}{p(\mathbf{d})}.$$

This will provide us with a distribution over possible values of $\mathbf{w}$ rather than the single most likely value of $\mathbf{w}$.

# Bayesian Neural Networks

- $p(\mathbf{w}|\mathbf{d}, \mathcal{H})$: distribution over weights $\mathbf{w}$ given the data and a fixed model $\mathcal{H}$.
- $p(\mathbf{y}|\mathbf{d}, \mathcal{H})$: distribution over network outputs $\mathbf{y}$ given the data and a fixed model $\mathcal{H}$ (Regression Problems).
- $p(\mathbf{C}|\mathbf{d}, \mathcal{H})$: distribution over network outputs $\mathbf{y}$ given the data and a fixed model (Classification Problems).

# Bayesian Neural Networks

- **Epistemic Uncertainty**: Uncertainty about the model parameters **w**.
- **Aleatoric Uncertainty**: Uncertainty in the output (prediction)!

# Example of Bayesian Classification

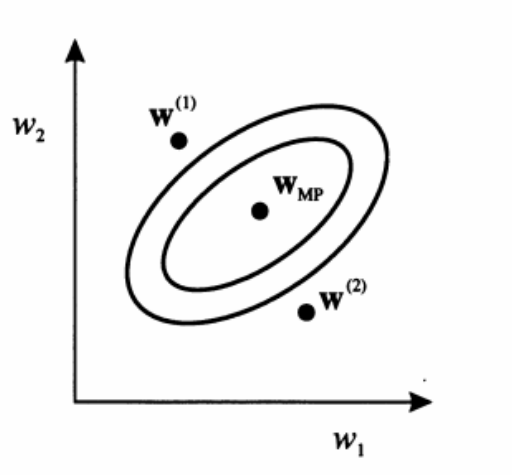



Figure 1

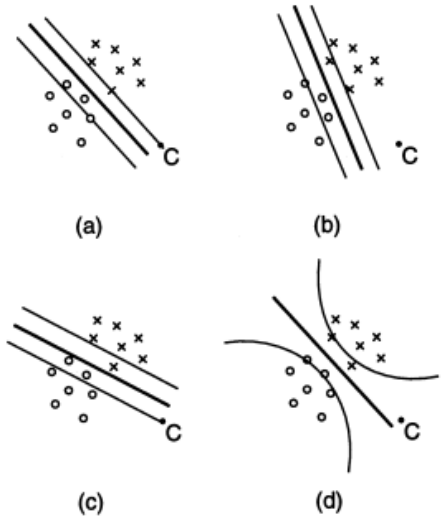



Figure 2

The three lines in Figure 2 correspond to network outputs of 0.1, 0.5, and 0.9. (a) shows the predictions made by $\mathbf{w}_{MP}$. (b) and (c) show the predictions made by the weights $\mathbf{w}^{(1)}$ and $\mathbf{w}^{(2)}$. (d) shows $P(\mathcal{C}_1|\mathbf{x}, \mathcal{D})$, the prediction after marginalizing over the distribution of weights; for point C, far from the training data, the output is close to 0.5.

Bayesian Methods for Neural Networks – p.20/29

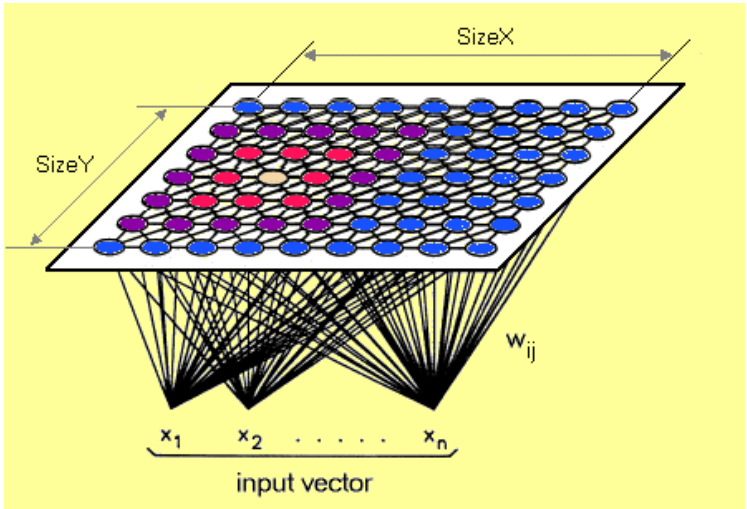
# BNN: Posterior Inference

- One way to use posterior uncertainty is to sample a set of values $\mathbf{w}_1, \mathbf{w}_2, ..., \mathbf{w}_k$, from the posterior $p(\mathbf{w}|\mathbf{d})$,and then average their predictive distributions,
- We can't sample exactly from the posterior, but we can do so approximately using Markov chain Monte Carlo (MCMC).
- Variants of MCMC : Hamiltonian Monte Carlo (HMC), Langevin Monte Carlo!
- Variational Bayes: Less accurate but more scalable!

# Unsupervised Neural Network

## Self-Organizing Maps

- Self Organizing Map(SOM) by Teuvo Kohonen provides a data visualization technique which helps to understand high dimensional data by reducing the dimensions of data to a map. SOM also represents clustering concept by grouping similar data together.
- Unlike other learning technique in neural networks, training a SOM requires no target vector. A SOM learns to classify the training data without any external supervision.



# Unsupervised Neural Network

## Self-Organizing Maps - Algorithm

Each data from data set recognizes themselves by competeting for representation. SOM mapping steps starts from initializing the weight vectors. From there a sample vector is selected randomly and the map of weight vectors is searched to find which weight best represents that sample. Each weight vector has neighboring weights that are close to it. The weight that is chosen is rewarded by being able to become more like that randomly selected sample vector. The neighbors of that weight are also rewarded by being able to become more like the chosen sample vector. From this step the number of neighbors and how much each weight can learn decreases over time. This whole process is repeated a large number of times, usually more than 1000 times.

- Each node's weights are initialized.
- A vector is chosen at random from the set of training data.
- Every node is examined to calculate which one's weights are most like the input vector. The winning node is commonly known as the Best Matching Unit (BMU).
- Then the neighbourhood of the BMU is calculated. The amount of neighbors decreases over time.
- The winning weight is rewarded with becoming more like the sample vector. The neighbors also become more like the sample vector. The closer a node is to the BMU, the more its weights get altered and the farther away the neighbor is from the BMU, the less it learns.
- Repeat step 2 for N iterations.

# Unsupervised Neural Network

## Autoencoders