An Introduction to Artificial Neural Network

Ramaseshan Ramachandran

Ramaseshan

An algorithm is a sequence of instructions to solve a problem

- The steps to solve problems are well defined
- Steps are coded in some ordered sequence to transform the input from one form to another
- Rules are unambiguous
- Sufficient Knowledge is available to fully solve the problem

HUMAN/MACHINE LEARNING

- There are problems whose solutions cannot be formulated using standard rule-based algorithms
- Problems that require subtle inputs cannot be solved using standard algorithmic approach - face recognition, speech recognition, hand-written character recognition, etc
- Finding Examples and using experience gained in similar situations are useful
- Examples provide certain underlying patterns
- Patterns give the ability to predict some outcome or help in constructing an approximate model
- **Learning** is the key to the ambiguous world

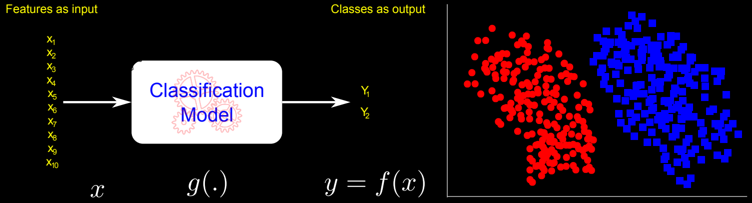
CLASSIFICATION

Classification is the task of assigning predefined dis-joint categories to objects

- Detect Spam emails
- Find the set of mobile phones $<$ Rs.10000 and received 5* reviews
- Identify the category of the incoming document as sports, politics, entertainment or business
- Determine whether a movie review is a positive or negative review

DEFINITION OF CLASSIFICATION

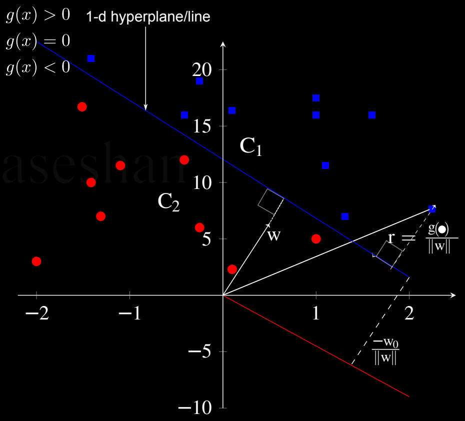
- The input is a collection of records
- Each record is represented by a tuple $(\mathbf{x}, \mathbf{y})$
- $\mathbf{x} = x_1, x_2, \ldots, x_n$ and $\mathbf{y} = y_1, y_2, \ldots, y_n$ are the input features and the classes respectively
- $\mathbf{x} \in \mathbf{R}^2$ is a vector - the set of observed variables
- (x, y) are related by an unknown function. The goal is to estimate the unknown function $g(.)$, also known as a classifier function, such that $g(x) = f(x), \forall x$



WHAT DOES THE CLASSIFIER FUNCTION DO?

Assuming that we have a linearly separable X, the linear classifier function $g(.)$ implements decision rule

- Fitting a straight line to a given data set requires two parameters (w_0 *and* w)
- The decision rule divides the data space into two sub-spaces - separating two classes using a boundary
- The distance of the boundary from the origin $= \dfrac{w_0}{||w||}$
- Distance of any point from the boundary $= d = \dfrac{g(x)}{||w||}$

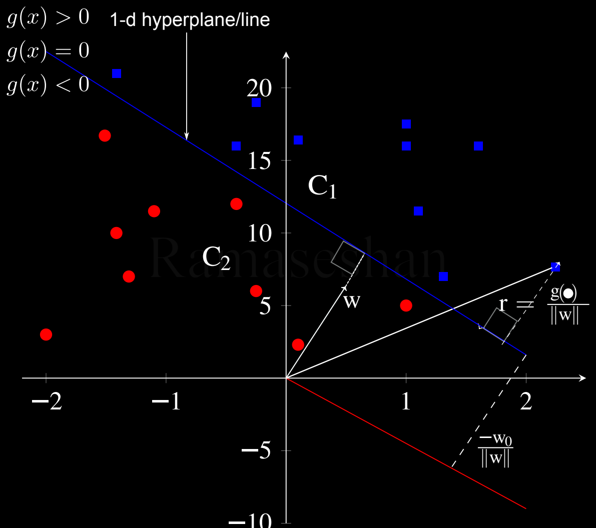


LINEAR MODELS FOR CLASSIFICATION

The goal of classification is to take a vector X and assign it to one of the N discrete classes $\mathbb{C}_n$, where $n = 1, 2, 3, \ldots, N$.

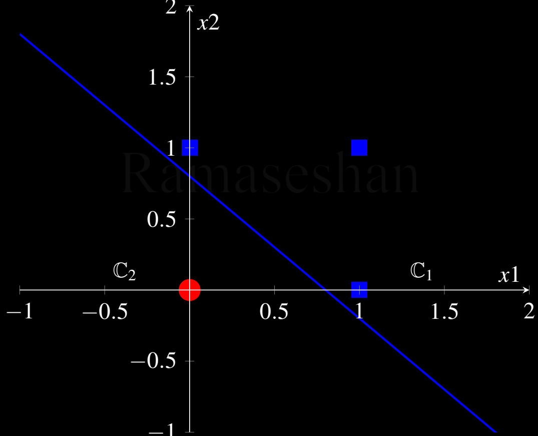
- The classes are disjoint and an input is assigned to only one class
- The input space is divided into *decision regions*
- The boundaries are called as *decision boundaries or decision surfaces*
- In general, if the input space is N dimensional, then $g(x)$ would define an $N-1$ hyperplane

GEOMETRY OF THE LINEAR DISCRIMINANT FUNCTION



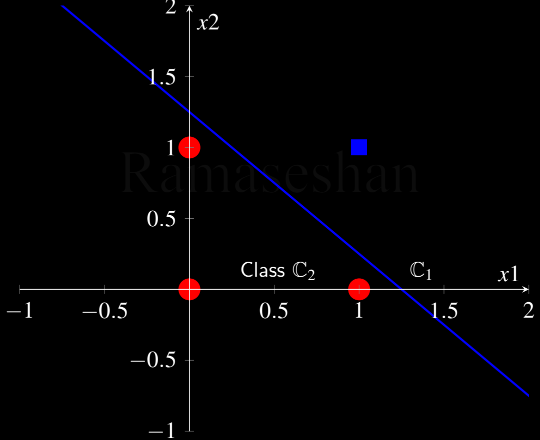
1D-DECISION BOUNDARY FOR OR GATE

The decision regions are separated by a hyperplane and it is defined by $g(x) = 0$. This separates linearly separable classes $\mathbb{C}_1$ and $\mathbb{C}_2$



1D-DECISION BOUNDARY FOR AND GATE

The decision regions are separated by a hyperplane and it is defined by $g(x) = 0$. This separates linearly separable classes $\mathbb{C}_1$ and $\mathbb{C}_2$

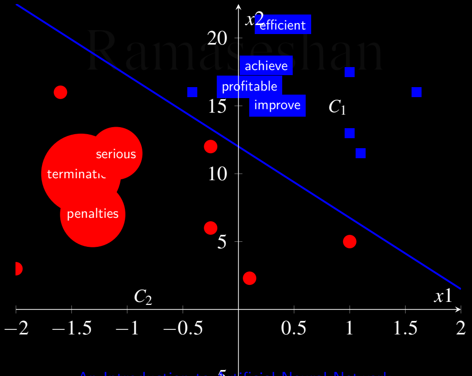


DECISION BOUNDARY FOR SENTIMENTS

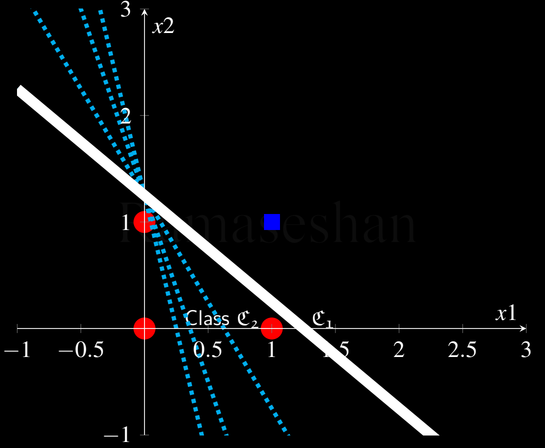
Let us consider some positive and negative sentiment terms which are contained in two classes $\mathbb{C}_P$ *and* $\mathbb{C}_N$

$$\mathbb{C}_P = [achieve\ efficient\ improve\ profitable] = +1$$

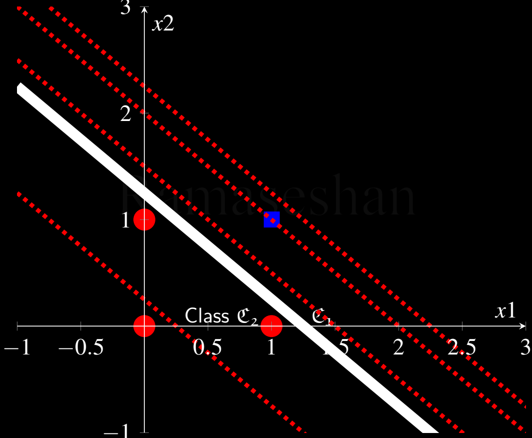
$$\mathbb{C}_N = [termination\ penalties\ misconduct\ serious] = -1$$



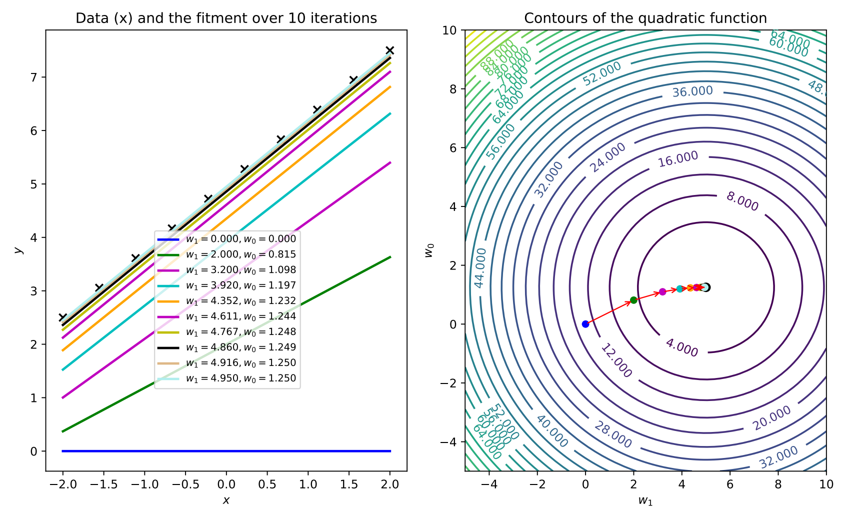
DECISION BOUNDARY- VARIATION OF W_J



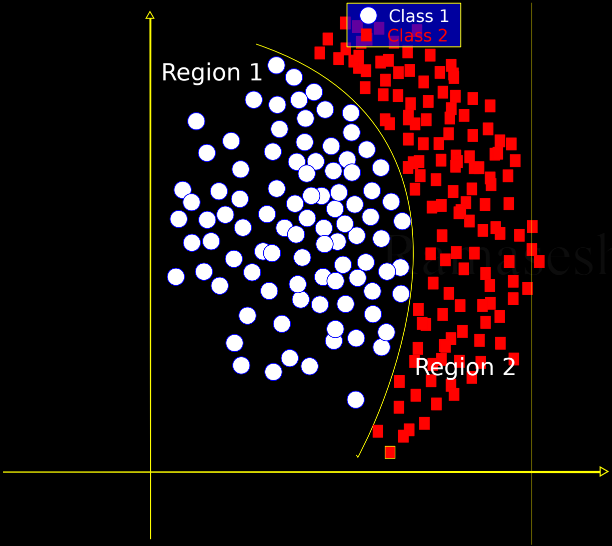
The contribution of bias to the the creation of the decision boundary



DECISION BOUNDARY AND GRADIENT DESCENT



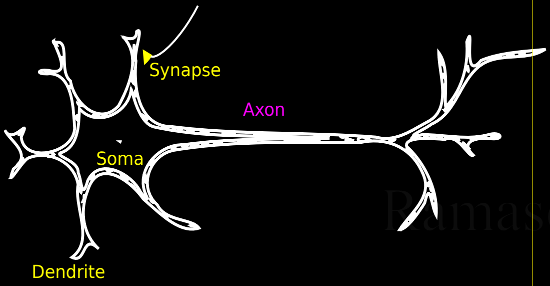
LINEARLY SEPARABLE?



Is this separable?

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

NEURAL NETWORK



- Each individual neuron can form thousands of links with other neurons.
- A typical brain has well over 100 trillion synapses

- Functionally related neurons connect to each other to form neural networks
- The electro-chemical connections between neurons are not static
- The more signals sent between two neurons, the stronger the connection grows and with each new experience and each remembered event, the brain slightly re-wires its physical structure.
- Our brains form a million new connections for every second of our lives

LAWS OF ASSOCIATION

Aristotle's attempts on fundamental laws of learning and memory

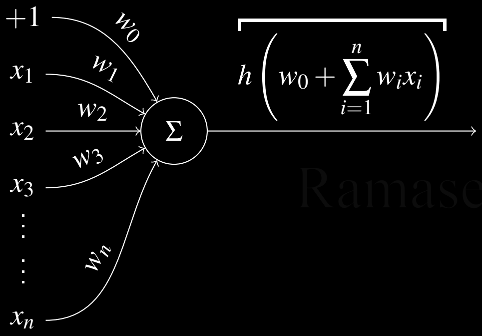
The law of similarity	If two things are similar, the thought of one will tend to trigger the thought of the other - word2vec If you recollect one birthday, you may find yourself thinking about others as well
The law of contrast	Seeing or recalling something may also trigger the recollection of something completely opposite
The law of contiguity	Things or events that occur close to each other in space or time tend to get linked together in the mind If you found a snake in the corner of the street, every time you cross the corner, you tend to look for one. Events are conditioned based on the time and space
The law of frequency	The more often two things or events are linked, the more powerful will be that association - think of next word prediction - strength of the association decides who is the probable candidate

PERCEPTRON

Neuron	**Perceptron**
Biological	A mathematical model of a biological neuron
Dendrites receive electrical signals	Perceptron receives mathematical values as input
Electro-chemical signals between Dendrites and axons	The weighted sum represents the total strength of the signal
The electro-chemical signals are not static	Weights change during the training process

Activation Function

$$\underbrace{h\left(w_0 + \sum_{i=1}^{n} w_i x_i\right)}$$



$$\underbrace{y' = \begin{cases} +1 & if\, h(.) \geq \theta \\ -1 & \text{otherwise} \end{cases}}$$

Feature Vectors
$x_1, x_2, x_3, \ldots x_n$

Learned Weights
$w_0, w_1, w_2, w_3, \ldots w_n$

Decision Function

PERCEPTRON LEARNING

- Perceptron learns the weights
- They are adjusted until the output is consistent with the target output in the training examples
- $w^{(k+1)} \propto (y - \hat{y})$
- The weights are updated as below
$$w_j^{(k+1)} = w_j^{(k)} - \eta(y_i - \hat{y}^{(k)})x_{ij}$$
where $w^{(k)}$ is the weight parameter associated with the i^{th} input at k^{th} iteration

  η is the learning parameter and x_{ij} is the j^{th} attribute of the i^{th} training sample
- If $(y - \hat{y}) \approxeq 0$, no prediction error
- During the training the weights contributing most to the error require adjustments

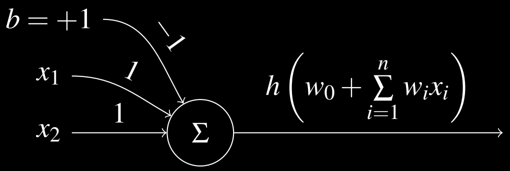
ALGORITHM FOR PERCEPTRON LEARNING

1: Total number of input vectors $= k$
2: Total number of features $= n$
3: Learning parameter $\eta = 0.01$, where $0 < \eta < 1$
4: epoch[1] count $t = 1, j = 1$
5: Initialize weights w_i with random numbers
6: Initialize the input layer with $\vec{x}_j$
7: Calculate the output using $\Sigma w_i x_i + w_0$
8: Calculate the error $(y - \hat{y})$.
9: Update the weights $w_j(t+1) = w_j - \eta(y - \hat{y})x_j$
10: Repeat steps 7 and 9 until: the error is less than θ or a predetermined number of epochs have been completed.

To provide a stable weight update for this step, $w_j(t+1) = w_j - \eta(y - \hat{y})x_j$, we require a small η. This results in slow learning. Bigger η would be good for fast learning. What are the problems? . What is the compromise?

[1] An epoch is one complete presentation of the data set to be learned to a learning machine.

LOGICAL AND

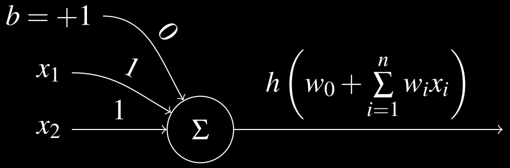


$$y' = \begin{cases} 0 & if\, h(.) \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

Input x_1	Input x_2	$x_1.w_1 + x_2.w_2 + b.w_b$	output-y
0	0	0.1+0.1-1	0
0	1	0.1+1.1-1	0
1	0	1.1+0.1-1	0
1	1	1.1+1.1-1	1

Here, the perceptron is already trained and the learned weights are shown in the diagram

LOGICAL OR



$$y' = \begin{cases} 0 & if\, h(.) \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

Input x_1	Input x_2	$x_1.w_1 + x_2.w_2 + b.w_b$	output-y

SENTIMENT ANALYSIS - USING PERCEPTRON

- Ability to classify reviews as positive or negative
- Positive and negative words for training
- Glove word embedding as features - input
  - 50 element word embedding[2]
  - Training Data generated using the intersection of the sentiment word list and word embedding from Glove

[2] data from https://nlp.stanford.edu/projects/glove/

GENERATE TRAINING DATA

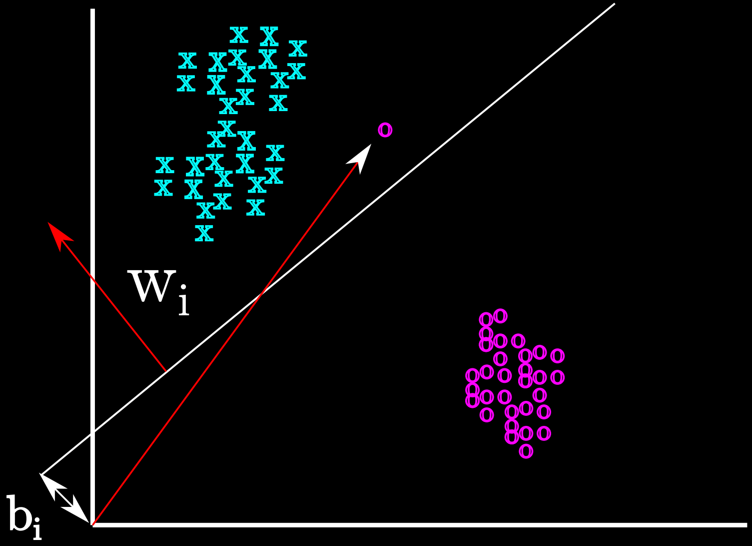
```
def generate_data():
    #data from https://nlp.stanford.edu/projects/glove/
    #...
    #...
    for pos_word in positives:
        positive_words.append(pos_word.rstrip())

    for neg_word in negatives:
        negative_words.append(neg_word.rstrip())
    for line in glove:
        values = line.split()
        word = values[0]
        vector = np.asarray(values[1:], dtype='float32')
        if word in positive_words:
            vector = np.append(vector,[1.0])
            emb_dict[word] = vector
        elif word in negative_words:
            vector = np.append(vector,[0.0])
            emb_dict[word] = vector
    #...
    dump(emb_dict,data_dir,'SentiWordEmbedding.bin')
```

BUILD MODEL

```python
def combine_input_and_weights(self, X):
    # linearly combine input vectors and weight vectors
    return np.dot(X, self.weights)


def build_model(self, X, y):
    # Build a model using the training data X and the class associated with each word embedding
    # X contains the word embeddings of sentiment words
    #y array contains the sentiment labels for every word - positive=1, negative=0
    X = self.normalize_feature_values(X)
    self.initialize_weights(X)
    for i in range(self.epochs):
        predicted_output = self.activation_function(self.combine_input_and_weights(X))
        errors = y - predicted_output
        self.weights += (self.eta * X.T.dot(errors))
        # compute the cost function
        cost_function = (errors ** 2).sum() / 2.0
        self.cost.append(cost_function)
    return self
```

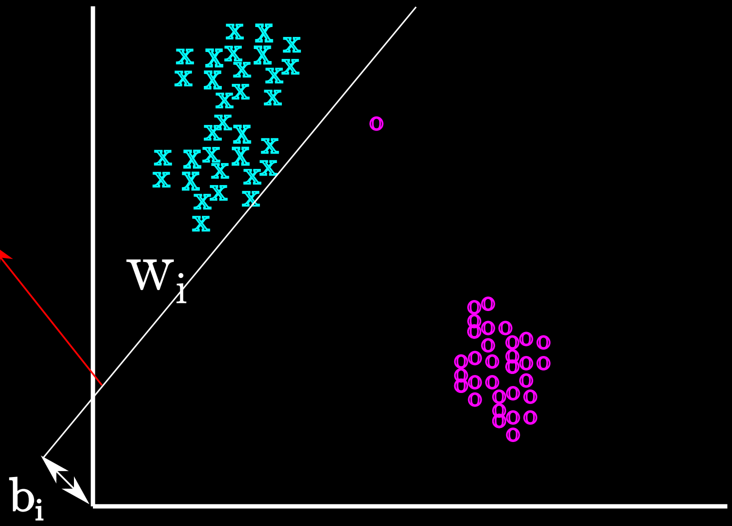
PREDICT SENTIMENTS

```
def predict(self, X):
    # predict the output corresponding to the input vector X
    X = self.normalize_feature_values(X)
    return np.where(self.activation_function(self.combine_input_and_weights(X)
    ) >= 0.0, 1, 0)

classifier = Perceptron(eta=0.00001, epochs=5000)
classifier.build_model(np.array(X),np.array(y))


test = sent_embedding_dict['terrible']
sentiment = classifier.predict(X_test)
print(sentiment)#0
```

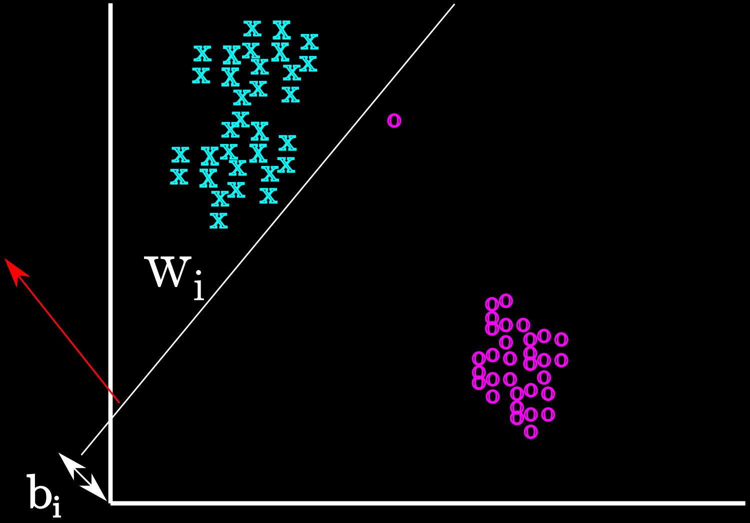


Figure

PERCEPTRON LEARNING



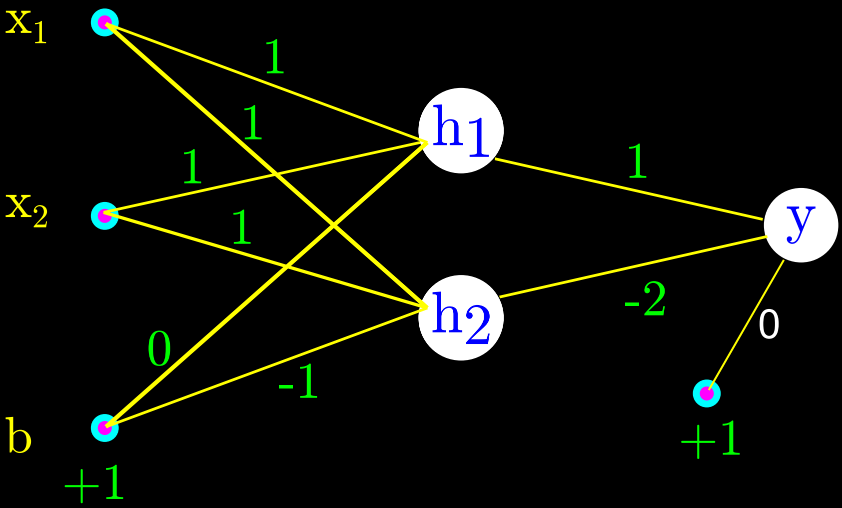
PERCEPTRON LEARNING



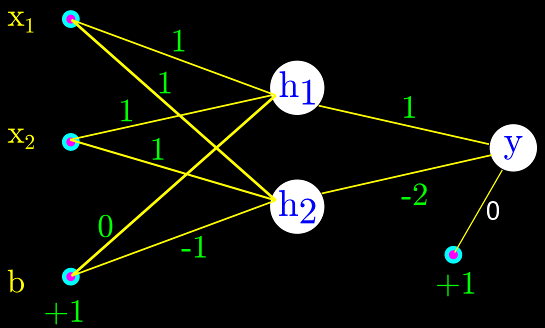
PERCEPTRON LIMITATIONS

- It is based on the linear combination of fixed basis functions
- Updates the model only based on misclassification
- Documents that are linearly separable are classified

LOGICAL XOR



LOGICAL XOR



Input x_1	Input x_2	b	h_1	h_2	output-y

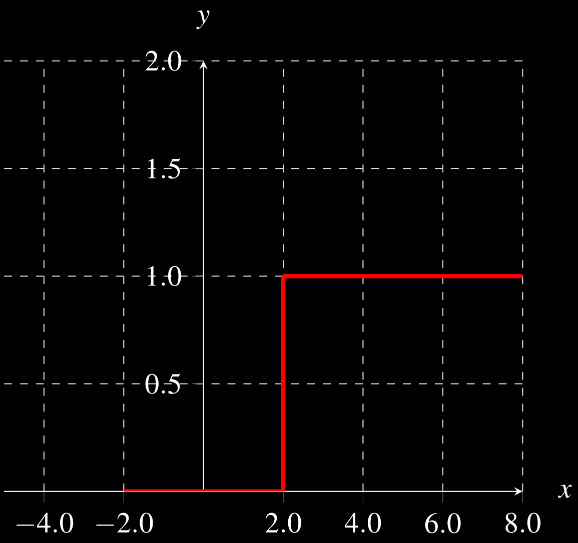
INTUITION

- Input space is transformed into hidden space
- Hidden layer represents the input layer
- Learns automatically the input representation and patterns
- $(0, 1)$ and $(1, 0)$ are merged into one in the h-space
- Patterns yielding similar results are merged into one
- Dimensionality reduction
- Are hidden layer neurons joining piecewise linear representations to create non-linear boundaries?

ACTIVATION FUNCTIONS

- Hard threshold
- Sigmoid
- Tanh
- ReLu - Rectified Linear Unit
- Leaky ReLu
- Softmax

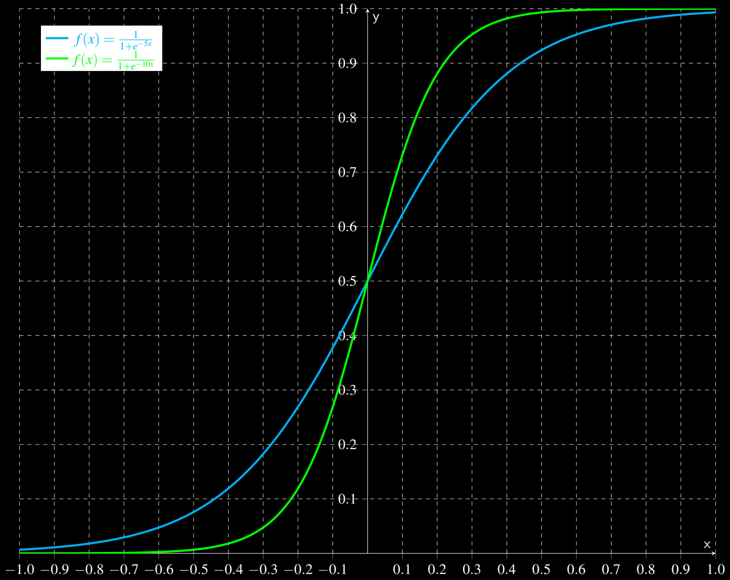
HARD THRESHOLD



SIGMOID 1/2

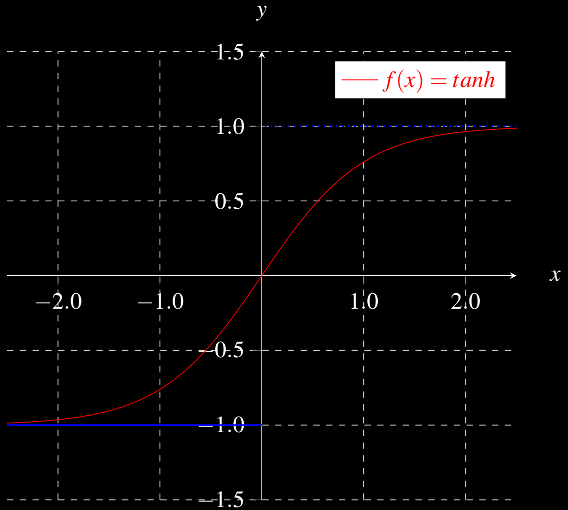
- The sigmoid is a non-linear function
- Better than hard threshold function as it squashes the net output into the range $[0, 1]$
- The values closer to the tails become 0 or 1
- In some cases, the values quickly saturate at 0 or 1
- At the bottom tail, most values become zero during the training and hence the most important aspect of learning of neural network is inhibited
- Sigmoid outputs are not zero-centered - $[01]$
- It is undesirable to have all the values squashed near the tails, where the gradient is 0

SIGMOID 2/2


$f(x) = \frac{1}{1+e^{-5x}}$
$f(x) = \frac{1}{1+e^{-10x}}$
y
x
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
−1.0 −0.9 −0.8 −0.7 −0.6 −0.5 −0.4 −0.3 −0.2 −0.1 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

TANH

This is a zero based non-linear function.



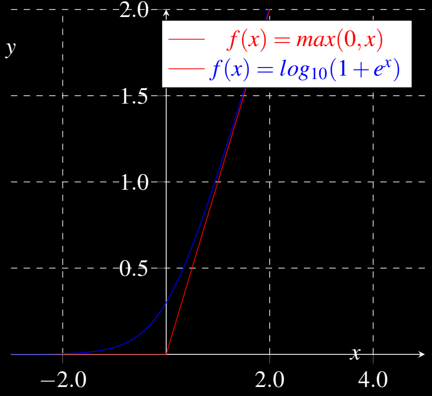
RELU - RECTIFIED LINEAR UNIT

- There is a continuous gradient for the neurons to be in active state
- Produces a non-zero gradient for values closer to zero
- Leaky ReLu

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

- Produces efficient propagation of the gradient
- Computationally efficient
- Scale invariant
- Unbounded and not zero centered

Learning rate (usually, very small) has to be fine tuned to minimize the death of neurons



ACTIVATION FUNCTION - PYTHON CODE

```
import numpy as np

def sigmoid(X,W,b):
    return 1.0/(1.0+ np.exp(-(np.dot(W.T,X)+b)))

def tanh(X,W,b):
    z = np.exp(-(np.dot(W.T,X)+b))
    return (np.exp(z) - np.exp(-z))/(np.exp(z) + np.exp(-z))

def relu(X,W,b):
    x = np.dot(W.T,X) + b
    return np.maximum(x,0)


def softmax(X,W,b):
    z_exp = np.exp(np.dot(W,X)+b)
    z_exp_sum = np.sum(z_exp)
    return z_exp/z_exp_sum


W=np.array([0.1, 0.2, 0.6])
X=np.array([0.2, 0.1, 0.3])
b=1.5
```

MULTICLASS DECISION FUNCTION

- All linear classifier are used for binary classifications
- In NLP problems, we need to identify more than two classes
  - Document classification
  - Sentiment Analysis - positive, negative, neutral and non-sentiment word
- We need a decision function that predicts more than two classes by providing appropriate values
- An extension of the case function would be hard to manage

SOFTMAX

- Need a function that takes as input a vector of of size with N real numbers, and normalizes it into a K classes.
- Need a function that normalizes the net output and classes well separated (ideal condition)
- Need a function that fits the classes using probability and distributes the probability density

$$Softmax(a_j) = P(C_k|x_j) = \frac{e^{a_j}}{\sum_{j=1}^{K} e^{a_k}}, \tag{1}$$

where $k = 1, K$ and x_j is the j^{th} input vector belonging to class k and $a_j = x_j.w_{ij}$

SOFTMAX - PYTHON IMPLEMENTATION

```python
import numpy as np
def softmax(X,W,b):
    z = np.exp(np.dot(W,X)+b)
    return z/np.sum(z)


W=np.array([0.1, 0.2, 0.6])
X=np.array([0.2, 0.1, 0.3])
b=1.5
W = np.array([[1,2,3],[2,3,8],[1,5,7]])

print(softmax(X,W,b))#[ 0.08672022  0.52462674  0.38865305]
```

LOSS FUNCTION

$$\hat{y} = \sum_{i=1}^{m} w_i^T x_i + w_0 \tag{2}$$

where $\hat{y}$ is the predicted value
w_0 is the bias
$\mathbf{x}$ is the input vector $\mathbf{w}$ is the weight vector
if y is the target, then the loss function is defined as a squared function

$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2 \tag{3}$$

The main idea is to reduce the residual $(y - \hat{y})$. When the value of L becomes negligible, we have predicted the vector to belong to a known class.
The loss function computes the error for a single training example

COST FUNCTION

Let us assume that we have a set of vectors for training. In the case of the sentiment analysis, these are the vectors obtained using any of the word embedding methods, representing the sentiment words. We could also use one-hot vectors representing the same

$$\mathbf{X} = [x_1, x_2, x_3, \ldots, x_N] \tag{4}$$

Combining the model parameters $w_0, w_1, w_2, w_3, \ldots, w_n$ with the loss function, we get a **Cost function**, averaged over all the input training samples

$$J(\theta) = \frac{1}{2} \sum_i L(y_i, \hat{y}_i)^2 \tag{5}$$

- The loss is a function of prediction and target values
- The cost is a function of model parameters and bias

GRADIENT DESCENT

- GD iteratively used to adjust the weights and as a result to minimize the cost function
- Initialize the weights to random values
- Iteratively adjust the weights in the direction of the steepest descent or in the direction that most decreases the cost function. To update the weights in the steepest descent, a learning parameter η is used

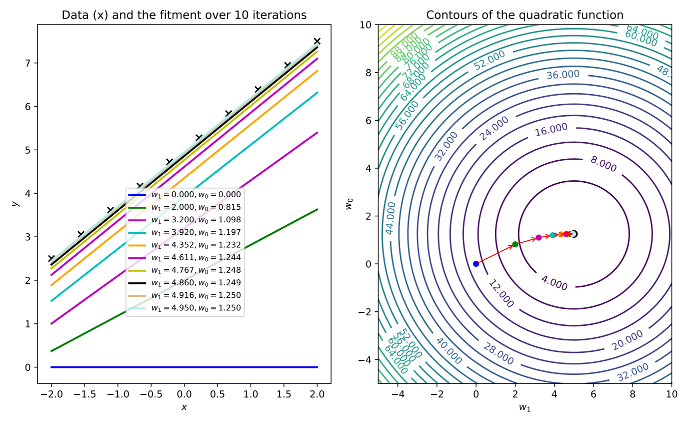
$$w_j \leftarrow w_j - \eta \frac{\partial J(\theta)}{\partial w_j} \tag{6}$$

$$= w_j - \eta \sum_{i=1}^{N} x_j{}^i (y - \hat{y}) \tag{7}$$

where η is the learning parameter and usually takes the value between 0.01 and 0.001

GD ADVANTAGES

- Iterative
- Computationally efficient
- Generic and could be used to solve even non-linear equations
- Suitable for large models
- It works!
- It is very slow when it reaches close to the the local minima



SEQUENTIAL NATURE OF DATA

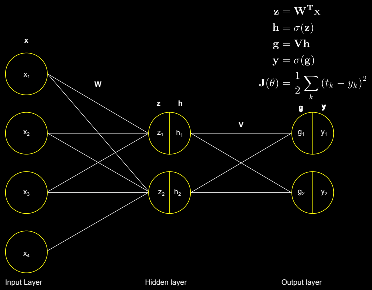
- Speech
- Documents
- Videos
- Weather forecast
- Financial - Stock market

PROPERTIES OF ANN

- Massively parallel distributed structure
- Ability to learn
- Ability to learn from training samples
- Ability to find latent patterns in the data
- Generalize and associate data

BACKPROPAGATION MODEL

The goal of backpropagation is to change the weights so that the *estimated target* $\approx$ target, thereby minimizing the error for each neuron and the network as a whole.

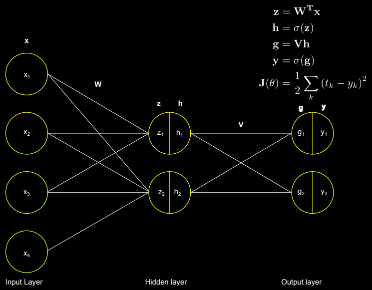


The goal is to minimize

$$J(\theta) = {}^1\!/_2\Big((t_1 - y_1)^2 + (t_2 - y_2)^2\Big) \tag{8}$$

* We want to the adjust weights coming in and going out of hidden layer so that $\mathbf{t} - \mathbf{y}$ is minimized
* $\Delta\mathbf{W} \propto -\dfrac{\partial J(\theta)}{\partial \mathbf{W}}$

BACK PROPAGATION MODEL - FORWARD PASS



$$z_1 = x_1.w_{11} + x_2.w_{21} + b_1 \quad (9)$$

$$z_2 = x_2.w_{12} + x_2.w_{22} + b_1 \quad (10)$$

$$h_1 = \sigma(z_1) = \frac{1}{1+e^{-z_1}} \quad (11)$$

$$h_2 = \sigma(z_2) = \frac{1}{1+e^{-z_2}} \quad (12)$$

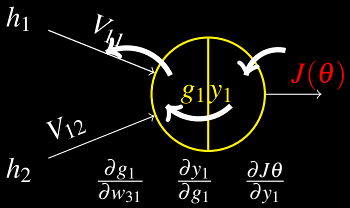
$$g_1 = h_1 * w_{31} + h_2.w_{41} \quad (13)$$

$$g_2 = h_2 * w_{32} + h_2.w_{42} \quad (14)$$

$$y_1 = \sigma(g_1) = \frac{1}{1+e^{-g_1}} \quad (15)$$

$$y_2 = \sigma(g_2) = \frac{1}{1+e^{-g_2}} \quad (16)$$

$$J(\theta) = 1/2\left((t_1 - y_1)^2 + (t_2 - y_2)^2\right)$$



$$\frac{\partial J(\theta)}{\partial V_{11}} = -\alpha \left(\frac{\partial J(\theta)}{\partial y_1} \frac{\partial y_1}{\partial g_1} \frac{\partial g_1}{\partial v_{11}} \right) \tag{17}$$

$$\frac{\partial J(\theta)}{\partial y_1} = -(t_1 - y_1) \tag{18}$$

$$\frac{\partial y_1}{\partial g_1} = y_1(1 - y_1) \tag{19}$$

$$\frac{\partial g_1}{\partial V_{11}} = h_1 \tag{20}$$

BACKWARD PASS-ADJUST HIDDEN-OUTPUT LAYER WEIGHTS

$$\frac{\partial J(\theta)}{\partial V_{11}} = -\alpha \left(\frac{\partial J(\theta)}{\partial y_1} \frac{\partial y_1}{\partial g_1} \frac{\partial g_1}{\partial V_{11}} \right) \tag{21}$$

$$= \alpha(t_1 - y_1)y_1(1 - y_1)h_1 \tag{22}$$

Now, the weights can be updated by $V_{11}^{t+1} = V_{11}^{t} - \eta * \frac{\partial J(\theta)}{\partial V_{11}^{t}}$. In the same fashion, compute the error to be propagated back to the other weights.

BACKWARD PASS – ADJUST INPUT-HIDDEN LAYER WEIGHTS

$$\frac{\partial J(\theta_1)}{\partial W_{11}} = -\alpha \frac{\partial J(\theta_1)}{\partial y_1} \frac{\partial y_1}{\partial g_1} \frac{\partial g_1}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial W_{11}}$$

$$\frac{\partial J(\theta_2)}{\partial W_{11}} = -\alpha \frac{\partial J(\theta_2)}{\partial y_2} \frac{\partial y_2}{\partial g_2} \frac{\partial y_2}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial W_{11}}$$

$$\frac{\partial J(\theta)}{\partial W_{11}} = \frac{\partial J(\theta_1)}{\partial W_{11}} + \frac{\partial J(\theta_2)}{\partial W_{11}} \tag{23}$$

$$z_1 = x_1 * W_{11} + x_2 * W_{21} \tag{24}$$

$$z_2 = x_2 * W_{12} + x_2 * W_{22} \tag{25}$$

$$h_1 = \sigma(z_1) \quad h_2 = \sigma(z_2) \tag{26}$$

$$g_1 = h_1 * V_{11} + h_2 * V_{21} \tag{27}$$

$$g_2 = h_2 * V_{12} + h_2 * V_{22} \tag{28}$$

$$y_1 = \sigma(g_1) \quad y_2 = \sigma(g_2) \tag{29}$$

$$\frac{\partial J(\theta_1)}{\partial y_1} = -(t_1 - y_1) \tag{30}$$

$$\frac{\partial y_1}{\partial g_1} = y_1(1 - y_1) \tag{31}$$

$$\frac{\partial g_1}{\partial h_1} = V_{11} \quad \frac{\partial z_1}{\partial W_{11}} = x_1 \tag{32}$$

$$\frac{\partial h_1}{\partial z_1} = z_1(1 - z_1) \tag{33}$$

$$\frac{\partial J(\theta_2)}{\partial y_2} = \dots \tag{34}$$

Now, input-hidden layer weights can be updated using $\mathbf{W}^{t+1} = \mathbf{W}^t - \eta * \frac{\partial \mathbf{J}(\theta)}{\partial \mathbf{W}}$

Once trained,

- The hidden layer of a trained model is a lookup table
- Hidden weights is an associative memory and captures the relational similarities
- The rows of the weight matrix represent the word vector