Lecture 8: June 2

Lecturer: Yishay Mansour *Scribe: Elad Hazan and Eyal Zach*

8.1 Lecture's Overview

In this lecture, we present the concept of *artificial neural networks*. We introduce the *perceptron* basic unit and the *gradient descent* learning rule. We also present multi-layer networks based on units such as the *sigmoid unit* and the *backpropagation* algorithm.

8.2 Biological Motivation

The study of *artificial neural networks (ANNs)* has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected neurons. In rough analogy, artificial neural networks are built out of densely interconnected set of simple units, where each unit can take a number of real-values inputs (possibly the outputs of other units) and produces a single real-valued output (which may become the input to many other units). In the next section, we present such a unit.
To develop a feel for this analogy, let us consider a few facts from neurobiology. The human brain, for example, is estimated to contain a densely interconnected network of approximately 10^{10} neurons, each connected, on average, to 10^4 other neurons. The fastest neuron switching time are known to be on the order of 10^{-3} seconds - quite slow compared to computer switching speeds of 10^{-10} seconds. Yet humans are able to make surprisingly complex decisions, surprisingly quickly. For example, it requires approximately 10^{-1} seconds to visually recognize a face. Many speculate that the information processing abilities of biological neural systems must follow from highly parallel processes operating on representations that are distributed over many neurons. In analogy, most ANNs software runs on sequential machines emulating distributed processes, although some algorithms have also been implemented on highly parallel machines and hardware designed specifically for ANN applications. Some researchers were motivated by the goal of using ANNs to study and model biological learning process, although ANNs do not model many complexities of biological neural networks, and are known to be inconsistent with biological systems.

8.3 Perceptron

In this lecture we introduce a unit called *Perceptron*, illustrated in Figure 8.1.
In essence, a perceptron is a weighted summation of several boolean inputs, which gives a boolean classification according to a threshold. We can view the perceptron as a hyperplane decision surface. In that case, the input $\vec{x}$ is a vector of points in the n-dimensional space.

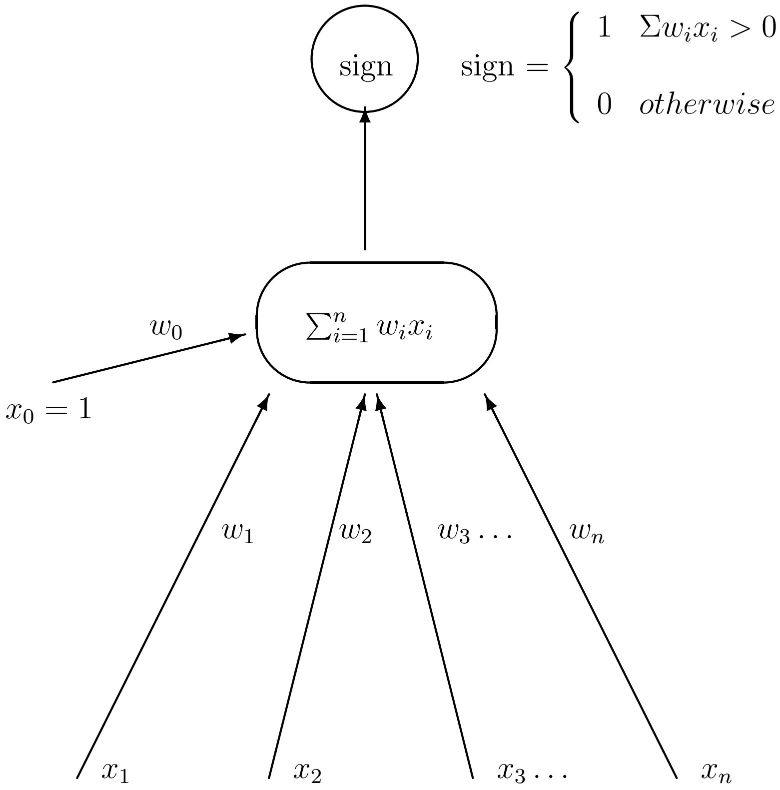

Figure 8.1: A perceptron. To simplify notation, we imagine an additional constant $x_0 = 1$, allowing us to write the inequality as $\sum_{i=0}^{n} w_i x_i > 0$

The development of such structures has been spurred due to:

- Simple structure.
- Efficient computation.
- Relation to models of the brain (i.e. Neural Networks).

One obvious drawback of such a structure is its limited ability to represent many function classes. For example, the XOR function cannot be represented by a *single* perceptron, because it is not linearly separable. However, every boolean function can be represented by some network of interconnected perceptrons.

8.3.1 Learning Rule

Generally, learning a perceptron involves choosing values for the weights $w_1, \ldots, w_n$. We apply an *On-line* learning model, which gradually learns a concept, by improving hypothesis at each learning step.
We observe that the VCdim of a perceptron (which is actually a n-dimentional hyperplane) is **n+1**[1]. Therefore, using the PAC learning model we need a relatively small number of examples in order to achieve an accurate approximation with high confidence. Most of the algorithms we have seen were *batch* based, in the sense that there was a separation between the training phase and the testing phase. The *on-line* model accepts examples successively and using the current hypothesis and example, generates a new hopefully improved hypothesis.
More formally, the learner is given an instance $x_i^n \in X^n$, which it classifies using its current hypothesis. Then the learner is given the correct classification of x_i^n by the target concept $c \in C$. If the prediction was incorrect, the learner has made a mistake, and the current hypothesis is updated in the following way:

$$w_i = w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Here t is the target output for the current training example, o is the output generated by the preceptron, and η is a positive constant called the *learning rate*.

In other words, the hypothesis is calculated incrementally and the performance of the algorithm is evaluated by the number of mistakes occurring during the learning. The aim is to achieve a minimal number of mistakes, without maintaining previous examples.

[1]See lecture 6

8.3.2 Perceptron Algorithm

The algorithm is initiated with some set of weights $\mathbf{w} = (w_1, \ldots, w_n)$ *and* $\theta = 1$, which form an initial hypothesis h. The learning process proceeds as follows:
Given an example $\vec{x}$, it uses $h(\vec{x})$ to classify it and the outcome is compared to the correct one, i.e. $f(\vec{x})$. If $h(\vec{x}) = f(\vec{x})$, i.e. the algorithm was correct, $h(\vec{x})$ is maintained. If $h(\vec{x}) \neq f(\vec{x})$, i.e. a mistake was done, we update $h(\vec{x})$.
There are two classification mistakes which are possible:

1. False Positive

Our hypothesis yields +1 where the target concept yields -1,
that is: $h(\vec{x}) = +1$ and $f(\vec{x}) = -1$.
The update procedure in this case will be: $\vec{w}_{j+1} = \vec{w}_j - \vec{x}_j$.
Explanation: Before update: $< \vec{w}_j, \vec{x}_j > \; = \; \sum_i w_{ij} \cdot x_{ij} > 0$
After update: $< \vec{w}_{j+1}, \vec{x}_j > \; = \; < \vec{w}_j - \vec{x}_j, \vec{x}_j > \; = \; < \vec{w}_j, \vec{x}_j > \; - \; < \vec{x}_j, \vec{x}_j >$.
We note that $< \vec{x}_j, \vec{x}_j >$ is positive for every $\vec{x}_j$ and therefore $\vec{w}_{j+1}$ updates $\vec{w}_j$ towards the correct direction.

2. False Negative

Our hypothesis yields -1 where the target concept yields +1,
that is: $h(\vec{x}) = -1$ and $f(\vec{x}) = +1$.
The update in this case: $\vec{w}_{j+1} = \vec{w}_j + \vec{x}_j$.
We note, as above: $< \vec{w}_{j+1}, \vec{x}_j > \; = \; < \vec{w}_j + \vec{x}_j, \vec{x}_j > \; = \; < \vec{w}_j, \vec{x}_j > \; + \; < \vec{x}_j, \vec{x}_j >$, and $\vec{w}_{j+1}$ updates $\vec{w}_j$ towards the correct direction.
It is clear that accepting the same example successively will produce a hypothesis which will classify it correctly. (The same is true for a repeating sequence of examples).

Notes:

1. One may assume that the instances $\vec{x}_i$ are unit-vectors, because $c \cdot \vec{x}$ is classified the same as $\vec{x}$.

2. If $h(\vec{x}) \neq f(\vec{x})$ then $h(-\vec{x}) \neq f(-\vec{x})$.

3. After an update, h does not necessarily classify the last example correctly.

8.3.3 Analysis

We show that if the data is linearly separable, then the number of errors of the preceptron algorithm is bounded.

Theorem 8.1 *If there exists a hyperplane which separates the positive examples from the negative examples, the perceptron learning algorithm will converge to this hyperplane after a finite number of examples.*

Proof: We assume without loss of generality that all of the examples are positive.
(If we get a negative example $\vec{x}$ we use $-\vec{x}$ which would be a positive example.)
For this reason, all of our updates are for *false negative.*
Let $\vec{v}$ be the weight vector of a separating hyperplane (which exists by the theorem assumption), and suppose $\|\vec{v}\| = 1$. (note that $\vec{v}$ is not unique!)
There exists a constant δ such that,

$$\forall x : f(\vec{x}) = +1 \;\Rightarrow\; \vec{v} \cdot \vec{x} > \delta.$$

δ may be $\delta(n)$ but it does not depend on the examples.
Let us consider the function $\Psi(\vec{w}) = \frac{\vec{w} \cdot \vec{v}}{\|w\|}$ which is the *cos* of the angle between the planes $\vec{v}$ and $\vec{w}$. Therefore $\Psi(\vec{w}) \leq 1$.

After an update: $\tilde{w} \;=\; w \,+\, x$.
We consider how $\vec{w} \cdot \vec{v}$ and $\|w\|^2$ change as the result of the update.

$$< v, \tilde{w} > \;=\; < v, w + x > \;=\; < v, w > \;+\; < v, x >$$

but

$$< v, x > \geq \;\delta \quad\Longrightarrow\; < v, \tilde{w} > \;-\; < v, w > \;\geq\; \delta.$$

On the other hand,

$$\|\tilde{w}\|^2 \;= < \tilde{w}, \tilde{w} > = < w + x, w + x > = < w, w > \; + 2 < w, x > \;+\; < x, x >$$

(we assumed the input vectors are normalized, i.e. $< x, x >= 1$.)

Since there was a mistake, $< w, x > \;<\; 0$ Hence:

$$\|\tilde{w}\|^2 \leq \|w\|^2 \;+\; 1.$$

After t updates:

- $< v, w_t > \ \geq \ \ \delta \cdot t$

- $\|w_t\|^2 \ \leq \ t \Rightarrow \|w_t\| \ \leq \ \sqrt{t}$

Consequently: $\Psi(w_t) \geq \frac{\delta \cdot t}{\sqrt{t}}$. Since for every t, $\Psi(w_t) \leq 1$
$1 \geq \delta \cdot \sqrt{t}$, which implies that $t \leq \frac{1}{\delta^2}$

Therefore the maximum number of mistakes of the algorithm is $\frac{1}{\delta^2}$.
□

8.4 Gradient Descent Learning Rule

The perceptron algorithm is not assured to converge if the examples are not linearly separable. A second learning rule, called gradient descent[2], is designed to overcome this difficulty by converging toward a best-fit approximation to the target concept.
A *linear unit* corresponds to the first stage of a perceptron, without the threshold.
We consider a linear unit with continuous output o such that $o = w_0 + w_1 x_1 + \ldots + w_n x_n$
Our aim is to minimize the squared error

$$E[w_1, \ldots, w_n] = \frac{1}{2} \sum_{d \in S} (t_d - o_d)^2$$

where S is the set of training examples, t_d is the target output and o_d is the output of the linear unit for training example d. Note that the squared error is a function easy to derive. The gradient descent algorithm determines a weight vector that minimizes E by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps. At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface. This process continues until the global minimum error is reached. This is done by computing the derivative of E with respect to each component of the weight vector $\vec{w}$. The vector derivative is called the *gradient* of E, written $\nabla E(\vec{w})$.

Formally,

[2] Also called *delta rule*

Gradient-Descent(S:training-examples, η:learning-rate)
Until TERMINATION Do
- $\forall i \Delta w_i \leftarrow 0$
- For each $< x, t > \in S$ Do
  - $\diamond$ Compute $o =< x, w >$
  - $\diamond$ For each weight w_i Do
    - $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$
- For each weight w_i Do
  - $\diamond$ $w_i \leftarrow w_i + \Delta w_i$

The key practical difficulties in applying gradient descent are:

- converging to local minimum can sometimes be quite slow
- if there are multiple local minima in the error surface, then there is no guarantee that the algorithm will find the global minimum.

One common variation on gradient descent intended to alleviate these difficulties is called *incremental stochastic gradient descent*.

8.4.1 Comparison of Perceptron and Gradient Descent Learning Rules

Perceptron learning rule guaranteed to succeed if

- Training examples are linearly separable
- No guarantee otherwise

In comparison, for linear unit using gradient descent:

- Given sufficiently small learning rate η, converges to hypothesis with minimum squared error, even when training data contains noise or training data not linearly separable

8.5 Multilayer Networks and the Backpropagation Algorithm

As noted in the previous section, single perceptrons can only express linear decision surfaces. This section discusses multi-layered networks that are capable of learning a variety of non-linear decision surfaces.

The main focus of the section is the Backpropagation algorithm, a gradient descent algorithm, that enables learning multi-layered networks.

8.5.1 The Sigmoid Unit of Multi-Layered Networks

Each basic unit has n inputs $(x_1, \ldots, x_n)$. Note that $w_1, \ldots, w_n$ are the parameters of the unit, i.e. they are constant for any set of inputs.
The function $\sigma(S) : \Sigma x_i w_i \mapsto \{0, 1\}$ is called the threshold function. In the case of the perceptron, it is simply:

$$\sigma(S) = \begin{cases} 1 & S \geq \theta \\ \\ 0 & otherwise \end{cases}$$

For some threshold parameter θ.

For a gradient descent algorithm, such as the Backpropagation algorithm to be described shortly, the output of the unit must be continuous and differentiable. Therefore, the unit of choice is the *sigmoid unit* with the threshold function of:

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

Note that this function output ranges from 0 to 1 and increases monotonically.
An important and interesting property of it is that its derivative is easily expressed as its output:

$$\frac{\partial \sigma(y)}{\partial y} = \sigma(y) \cdot (1 - \sigma(y))$$

8.5.2 The Backpropagation Algorithm

The Backpropagation algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs. The square error is defined as:

$$E := \frac{1}{2} \sum_{d \in S} \sum_{k \in outputs} (t_d^k - o_d^k)^2$$

Where S is the training set, *outputs* is the set of output units in the network, and t_d^k and o_d^k are the target and output values associated with the k'th output unit and training example d.
The Backpropagation algorithm is now defined as:

- Given a multi-layered network, initialize all weights to small random numbers.
- Until the termination condition is met, for each $(\vec{x}, \vec{t})$ in the training set, Do:
  Propagate the input forward through the network:
  1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

  Propagate the errors backward through the network:
  1. For each network output unit k, compute its error term:
  $$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$
  2. For each hidden unit h, calculate its error term:
  $$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_k^h \delta_k$$
  3. Update each network weight w_j^i:
  $$w_j^i \leftarrow w_j^i + \Delta w_j^i$$
  Where:
  $$\Delta w_j^i = \eta \delta_j x_j^i$$
  (η is an adjustable learning rate)

8.5.3 Derivation of the Backpropagation Rule

The gradient descent rule, as described previously, involves iterating through the training examples one at a time, and for each example d, descending the gradient of the error E_d with respect to this single example. That is, for every example, each weight is updated:

$$\Delta w_i^j = -\eta \frac{\partial E_d}{\partial w_i^j}$$

Where the error is:

$$E_d(\vec{w}) := \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

First, we introduce a few notations:

- x_i^j - the i'th input to unit j

- w_i^j - the weight associated with the i'th input to unit j
- $net_j := \sum_i w_i^j x_i^j$ - the weighted sum of inputs for unit j
- o_j - the output computed by unit j
- t_j - the target output for unit j
- σ - the sigmoid function
- *outputs* - the set of units in the final layer of the network
- $Downstream(j)$ - the set of units whose immediate inputs include the output of unit j

We proceed to analyze two different cases:

Case 1: Training Rule for Output Unit Weights

Note that weight w_i^j can influence the rest of the network only through net_j, and net_j can influence the rest of the network only through o_j. Therefore, according to the chain rule:

$$\frac{\partial E_d}{\partial w_i^j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_i^j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}x_i^j$$

Now simply calculate the other derivatives of the error, and for the first we get:

$$\begin{aligned}\frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k\in outputs}(t_k - o_k)^2 \\ &= \frac{1}{2}2(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j} \\ &= -(t_j - o_j)\end{aligned}$$

As for the second derivative, recall that $o_j = \sigma(net_j)$, and therefore the derivative $\frac{\partial o_j}{\partial net_j}$ is the derivative of the sigmoid function, which is as we saw before $\sigma(net_j)(1-\sigma(net_j))$. Therefore:

$$\begin{aligned}\frac{\partial E_d}{\partial w_{ji}} &= \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_i^j} \\ &= \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}x_{ji} \\ &= -(t_j - o_j)o_j(1-o_j)x_i^j\end{aligned}$$

And the gradient descent rule is to update the weights according to:

$$\Delta w_i^j = -\eta\frac{\partial E_d}{\partial w_i^j} = \eta(t_j - o_j)o_j(1-o_j)x_i^j$$

Case 2: Training Rule for Hidden Unit Weights

In this case, note that net_j can influence the network outputs (and therefore E_d) only through the units in $Downstream(j)$. Therefore we can write:

$$\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k}\frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j}\frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k w_k^j \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k w_k^j o_j(1-o_j)
\end{aligned}$$

Rearanging terms and using δ_j to denote $-\frac{\partial E_d}{\partial net_j}$, we have:

$$\delta_j = o_j(1-o_j) \sum_{k \in Downstream(j)} \delta_k w_k^j$$

And:

$$\Delta w_i^j = \eta \delta_j x_i^j$$

8.5.4 Summary on the Backpropagation Algorithm

As shown above, the Backpropagation algorithm implements a gradient descent search through the space of possible network weights. Because the error surface for multilayer networks may contain many local minima, Backpropagation may get trapped in any of these. Therefore, Backpropagation is guarantied to converge toward some local minimum in E and not necessarily to the global minimum.
Nonetheless, Backpropagation performs well in practice.
Several methods may be deployed to avoid local minima:

- Training multiple networks using the same data, but initializing the weights of each differently (randomly).

- Adding a "momentum" to the weight update rule:

$$\Delta w_i^j(n) = \eta \delta_j x_i^j + \alpha \Delta w_i^j(n-1)$$

The factor $\alpha \Delta w_i^j(n-1)$ is called the momentum factor, and intuitively keeps the gradient change in the same direction as the previous step.

The Backpropagation algorithm can also be easily generalized to arbitrary multi-layered networks.

8.6 Expressive Capabilities of Neural Networks

8.6.1 Upper Bounds

- Every boolean function can be represented by network with single hidden layer, but might require exponential (in number of inputs) hidden units.
- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989, Hornik 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988]

8.6.2 Notations

Given basic units as described in previous subsections, we can combine them to create Neural Networks (NN) much like combining and/or gates creates boolean circuit. The NN is an acyclic directed graph for which the output of an edge is marked by an outgoing vetrex, and the input by ingoing one. Each vertex in layer $l+1$ receives its inputs from layer l.

The NN has three basic parameters:

- The number of inputs denoted n.
- The number of nodes denoted s.
- The in degree of a non-input node denoted r.

8.6.3 Lower Bounds - the VC-dimension of Neural Networks

We consider a general case in which we have a family of concepts C over $\Re^r$ to $\{0,1\}$,and each node has a concept $c \in C$. Since each node can output only zero or one, the input values of all inner nodes (including the root) will be from $\{0,1\}^r$.

The object of learning the NN is finding which concept resides in each node, i.e. the structure of the NN is known to the learning algorithm.

We shall denote the family of functions produced by assigning concepts from C to nodes of the graph G, by C_G.

The objective of the following theorem is to bound $VCdim(C_G)$ by a polynomial function of $VCdim(C)$.

Theorem 8.2 *Let G be an acyclic directed layered graph with n input nodes (zero entry level), $s \geq 2$ inner nodes with an entry level of r. Let C be a concept class over $\Re^r$ with $VCdim(C) = d$. Then*

$$VCdim(C_G) \leq 2ds \cdot log(es)$$

Proof: The idea is to bound $\Pi_{C_G}(m)$. Once we have an upper bound on it, then the first value of m for which $\Pi_{C_G}(m) < 2^d$ in necessarily an upper bound on $VCdim(C_G)$.

Let $S = \{\vec{x_1}, \ldots, \vec{x_m}\}$, where $\vec{x_i} \in \Re^n$. Each vector $\vec{x_i}$ is an input to the NN (the graph). Once we set the concepts for all the nodes in G, we can compute $G(\vec{x_i})$.

For each node in G we build a vector from $\{0,1\}^m$ by setting the i^{th} bit to be the note's output when $\vec{x_i} \in S$ is the input of the graph. i.e. this vector describes the outputs of a node with respect to each vector in S.

Consider all the possible functions in C_G, i.e., all assignments of concepts from C to all nodes of G. Two functions are distinct with respect to S, then there exists at least one node in G, for which the above mentioned vectors differ.

Let $T_{C_G}(S)$ denote the number of different computations of functions from C_G with S as inputs. In other words, this is the number of distinct functions in C_G (as described above).

Clearly $|\Pi_{C_G}(S)| \leq T_{C_G}(S)$, since $\Pi_{C_G}(S)$ considers two functions to be distinct only when the output vector (the vector of the root) is different, without considering all the vectors of the other nodes.

In order to bound $T_{C_G}(S)$ we extract the root of graph G (denoting the new graph G'). Let $T_{C_{G'}}(S)$ be the number of different computations of G' over S. Each computation can be extended into at most $\Pi_C(m)$ different computations of G over S (For each computation of G' the root gets $y_1, \ldots, y_r$ vectors of size m, which construct up to m different inputs. Assigning different concepts to the root can expand these inputs by magnitude of $\Pi_C(m)$.) Therefore

$$|\Pi_{C_G}(S)| \leq T_{C_G}(S) \leq T_{C_{G'}}(S) \cdot \Pi_C(m)$$

By continuing the process of root extracting we get:

$$|\Pi_{C_G}(S)| \leq |\Pi_C(M)|^s \leq \left(\frac{em}{d}\right)^{ds}$$

where the second inequality is using Sterling's formula. Note that we can use the depth of the NN instead of the number of nodes, s.

Our goal was to find m such that $\Pi_{C_G}(m) < 2^m$, and this m would bound the $VCdim(C_G)$. We can achieve this by finding an m for which

$$\left(\frac{em}{d}\right)^{ds} < 2^m.$$

Any m that satisfies,

$$m > 2ds \cdot log_2(es),$$

provided $s \geq 2$. □

We now use this theorem in order to bound the $VCdim$ of NN when $C = \{\sum_{i=1}^{r} w_i x_i \geq \theta\}$. Note that this concept class is actually the class of all hyper-planes over $\Re^r$, for which we already know that $VCdim(C) = r + 1$[3]. Therefore

$$VCdim(C_G) \leq 2(r+1)s \cdot log(es)$$

Combining this conclusion with the upper bound on the sample size needed for PAC learning we have

Theorem 8.3 *Let G be a acyclic directed graph, C_G the collection of all possible NN over G with s inner nodes and an in degree of r. The number of samples sufficient for PAC learning of C_G is*

$$O\left(\frac{1}{\epsilon}log\frac{1}{\delta} + \frac{(r+1)s \cdot log(s)}{\epsilon}log\frac{1}{\epsilon}\right)$$

[3]See lecture 6