my note about

Basic (and Maybe advanced) Deep Learning stuff

theory and samples

Valgi0 (me as always)

July 7, 2020

Contents

List of Figures

1Why this documents

I feel the exigence to put together all my notes and my experience in this wide word of Deep Neural Network. I want them in one place with a good organization allowing me and the reader to access to what I need quickly. This thought came to my mind while I was working to my dissertation thesis, and for the haste, I used to write notes every where and when I needed them they were hard to find. Btw I don't care to write why I'm doing this so

1.1 Where Information come from

I'm going to use some books freely distributed and one day I'll put the links here. Stay tuned

1.2 Chapters organization

Each chapter contains arguments to one topic and information are organized following this schema:

- Introduction to the topic
- Theory about topic
- Examples maybe
- Related links or papers (Idk)

1.3 Notation

- a,b,c are scalars
- **a,b,c** are vectors
- **A,B,C** are matrices
- **0** it is a zero vector
- $P(x = x_i), x \sim P(x_i)$ Probability mass function over a discrete variable x
- $p(x = x_i), x \sim p(x_i)$ Probability density function over a discrete variable x where x_i is a range of value
- $x \perp y$ the two variables are independent

2Has someone said Math?

In this chapter we are going to explore all math stuff we need to fully understand the concepts. If God Exists he must be a mathematician.

2.1 Linear Algebra OMG

Let's start by defining what is a vector.. a vector is a vector. $\mathbf{v} = [i_0, i_1, i_2, ..., i_n]$. In this case v is a vector composed by elements $i \in \mathbb{I}$ with size n. So vector comes from $\mathbb{R}^n$ space. This is the n-dimensional Euclidean Space. A vector with all zeros is called **zero vector** and it will be represented as $\mathbf{0}$

2.1.1 Vector Space

The Vector Space $\mathbb{X}$ is a set of vectors that satisfies the following conditions:

- **Addition Closure**. This means that:
  - if $\mathbf{x}, \mathbf{y} \in \mathbb{X}$ so $\mathbf{x} + \mathbf{y} = \mathbf{j}$ and $\mathbf{j} \in \mathbb{X}$ for each **x,y**
  - $\mathbf{x} + \mathbf{y} = \mathbf{y} + \mathbf{x}$
  - $\mathbf{x} + (\mathbf{y} + \mathbf{c}) = (\mathbf{x} + \mathbf{y}) + \mathbf{c}$
  - $\mathbf{x} + \mathbf{0} = \mathbf{x}$
  - For each $x \in \mathbb{X}$ exists a unique vector called **-x** such that $\mathbf{x} +$ (**-x**) $= \mathbf{0}$
- **Multiplication Closure**. This means that if $\mathbf{x} \in \mathbb{X}$ and y a scalar so $\mathbf{x} * y = \mathbf{j}$ and $\mathbf{j} \in \mathbb{X}$ for each **x,y**
  - $\mathbf{x} * 1 = \mathbf{x}$
  - For any a, b scalars $(ab) * \mathbf{x} = a(b * \mathbf{x})$
  - $(a + b) * \mathbf{x} = a * \mathbf{x} + b * \mathbf{x}$
  - $a(\mathbf{x} + \mathbf{y}) = a * \mathbf{x} + a * \mathbf{y}$

From the above conditions we can extract some considerations. The first is that each $\mathbb{R}^n$ is a vector space. A subset of this euclidean space can be a vector space only if it his unbounded. For example:

$$\mathbb{X} = ((x, y) : x < 5y < 5, (x, y) \in \mathbb{R}^2)$$

$$v_0 = (4, 4), v_1 = (3, 3) \in \mathbb{X}$$

$$v_0 + v_1 = v_2 = (7,7) not \in \mathbb{X}$$

So $\mathbb{X}$ is not a vector space.

2.1.2 Linear Independence

Linear combination is a very cool stuff. Let's have a vector $\mathbf{x}$ and some different vectors **y,z,k**. If there exists a set of three scalars such that $a_0 * \mathbf{y} + a_1 * \mathbf{z} + a_2 * \mathbf{k} = \mathbf{x}$ so $a_0 * \mathbf{y} + a_1 * \mathbf{z} + a_2 * \mathbf{k}$ is a linear combination. Linear indipendence is a propery between more vectors. $\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_n$ are linearly dependent just if there exist n scalars at least one of which is not zero, that:

$$a_0\mathbf{x}_0 + a_1\mathbf{x}_1 + ... + a_n\mathbf{x}_n = \mathbf{0}$$

Otherwise, if all a_i are 0, vectors are *linearly independent.* **Attention** if a set of vectors is linearly dependent so one or more that tis vectors can be obtained by a linear combination of others.
Let's do some examples

$$\mathbf{v}_0 = [1, -1], \mathbf{v}_1 = [-1, 1], \mathbf{v}_2 = [-1, -1]$$

These are three vectors. Are they linearly independent?

$$a_0 * \mathbf{v}_0 + a_1 * \mathbf{v}_1 + \mathbf{v}_2 * a_2 = 0 \rightarrow \begin{bmatrix} a_0 - a_1 - a_2 = 0 \\ -a_0 + a_1 - a_2 = 0 \end{bmatrix}$$

Now we need to solve those equations:

$$a_0 = a_1, a_2 = 0$$

So they are a linear dependent vectors. This means that we can select one of them and express it as a linear combination of the others:

$$\mathbf{v}_0 = \mathbf{v}_1 * a_1 + \mathbf{v}_2 * a_2$$

That is true for $a_1 = -1$ and $a_2 = 0$.

2.1.3 Spanning a Vectors space

To create a vector space we can grab some vectors and create this space as the set of all linear combinations of these vectors. It the given vectors are linearly independent they are called **Basis set** for that vector space. The number of vectors in the basis set is the dimension of the vector space. This means that we can express each vector inside that space using a number of independent vectors equal to the dimension of the space. **For Example** $\mathbb{R}^2$ can have as basis $(1,0),(0,1)$. Using these two vectors we can create each vector inside the space. Try to believe.

A vector space can have infinite basis. But the basis composed by binary vectors

2.1.4 Some functions

Some functions can be defined between vectors. the first is the **Inner Product** that is the product between two vector and it is represented with $(\mathbf{x}, \mathbf{y})$. This function must follow these properties:

- $(\mathbf{x}, \mathbf{y}) = (\mathbf{y}, \mathbf{x})$
- $\mathbf{x}(a\mathbf{y} + b\mathbf{j}) = a(\mathbf{x}, \mathbf{y}) + b(\mathbf{x}, \mathbf{j})$
- $(\mathbf{x}, \mathbf{y}) \geq 0$ if $\mathbf{x}$ is not the zero vector.

The standard inner product adopted the respect the above properties is:

$$\mathbf{xy} = x_0y_0 + x_1y_1 + ... + x_ny_n \tag{2.1}$$

Another function that can be defined is the **Norm** represented by $\parallel \mathbf{x} \parallel$. This function must respect these properties:

- $\parallel \mathbf{x} \parallel \geq 0$
- $\parallel \mathbf{x} \parallel = 0$ if $\mathbf{x}$ is the zero vector
- $\parallel a\mathbf{x} \parallel = | a | \parallel \mathbf{x} \parallel$
- $\parallel \mathbf{x} + \mathbf{y} \parallel \leq \parallel \mathbf{x} \parallel + \parallel \mathbf{y} \parallel$

There are many function that satisfy these conditions. The most common **norm** is:

$$\parallel \mathbf{x} \parallel = (\mathbf{xx})^{\frac{1}{2}} = \sqrt{x_0^2 + x_1^2 + ... + x_n^2} \tag{2.2}$$

Using both of the previous function we can define the *angle* between two vectors. In particular we can calculate the *cosine*:

$$cosine(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{xy}}{\parallel \mathbf{x} \parallel \parallel \mathbf{y} \parallel} \tag{2.3}$$

2.2 Linear Transformation

For **Transformation** we identify the operation that relate a value x to a value y. Let's define:

- *Domain* the set of $x_i \in \mathbb{X}$ values.
- *Range* or Co-domain the set of $y_i \in \mathbb{Y}$ values
- *Transformation* as the rule relating each x_i to an y_i

A transformation A is *Linear* if:

- for all x_0, x_1 $A(x_0 + x_1) = A(x_0) + A(x_1)$
- for all x_0 and for all a scalars $A(x_0 * a) = A(x_0) * a$

IMPORTANT
For any transformation between two finite dimensions vector space can be represented by a Matrix. This Matrix is build using the basis of the vector spaces so changing them the matrix change

2.2.1 Eigenvalues eigenvectors

All vectors from a Vector space not 0 and scalars that make this equation true:

$$A((z)) = a(z) \tag{2.4}$$

are called Eigenvalues and eigenvectors. What that means?? Graphically the eingenvector of a Linear transformation is the direction for which vectors in that direction that after the transformation will have the same direction but a different length.

How to compute them

In order to compute them let's take a look to the formula:

$$A(\mathbf{z}) = a\mathbf{z}$$

We said that a linear transformation can be represented by a matrix so the formula becomes:

$$\mathbf{A}\mathbf{z} = a\mathbf{z}$$

$$(\mathbf{A} - a\mathbf{I}) * \mathbf{z} = 0$$

Let's define **A** matrix as:

$$\begin{vmatrix} -1 & 1 \\ 0 & -2 \end{vmatrix}$$

so $(\mathbf{A} - a\mathbf{I}) = 0$ (Now we look for determinant that is 0,.. dont ask why) is:

$$\begin{vmatrix} -1-a & 1 \\ 0 & -2-a \end{vmatrix} = 0$$

The solutions are found solving:

$$a^2 + 3a + 2 = (a+1)(a+2) = 0 \Rightarrow a_0 = -1 a_1 = -2$$

Those are the eigenvalues. To find the vectors you must solve the equation

$$(\mathbf{A} - a\mathbf{I}) * \mathbf{z} = 0$$

using each eigenvalue.

2.3 Probability

Let's talk about probability. In machine learning and Deep learning probability is essential cause the are stochastic fields and not deterministic ones. Let's define a variable x_r as a random variable on a dice domain. This means that it can be each value within 1,2,3,4,5,6 with the same probability.

Let's speak about **Probability Mass Function** as the probability that some variables can be found in a certain state. For example $P(x_r = 1) = 1/6$ (For completeness it's possible to write this function in other ways: $x_r \sim P(1)$ or $P_{x_r}(1)$.

We can have more dices and we can define **Joint Probability** as a Probability Mass Function defined on more variables: $P(x_r = 1, y_r = 2)$ that denotes the probability that $x_r = 1$ and $y_r = 2$ at once.

$P(x)$ must follow some properties:

- $\forall x_i \in x_r 0 < P(x_i) < 1$
- $\forall x_i \in x_r \sum P(x_i) = 1$

But htis works for discrete variable, for continuous variable defined on $\mathbb{R}$ things become harder. We define the probability to find the variable in a range of value. Let's define the range as $[a, b]$ with $a, b \in \mathbb{R}$ and $b >= a$. Let's define p the **Probability density function** as the function that denotes the probability of the input variable to be found with a value in the given range: $p(x_r; a, b)$. This function draws a linear function that must follow some properties:

- Area beneath the function must be 1: $\int p(x)dx = 1$
- $p(x) >= 0$

2.3.1 Conditional Probability

Let's define two events x_0, y_0 and for some reason they are connected in some way. Now we want to compute the probability of $y_r = y_0$ knowing that $x_r = x_0$ has happened. In this case we speak about **Conditional Probability** and it's defined:

$$P(y_r = y_0 | x_r = x_0) = \frac{P(y_r = y_0, x_r = x_0)}{P(x_r = x_0)} \tag{2.5}$$

For examples let's use ours dices. We want to know the probability to get $y_r = 1$ if we already got $x_r = 1$. So $P(1|1) = \frac{P(1,1)}{P(1)}$ in this case we get: $\frac{1/36}{1/6} = \frac{6}{36} = \frac{1}{6}$ what we get in this case is the two dices are independent, the probability to have 1 in the second dice doesn't change respect the first.

Some times we know the probability of $P(x|y)$ and the probability of $P(x)$. We would like to know the probability of $P(x|y)$. To get that we can use the **Bayes' Rule**:

$$P(x_r = x_0 | y_r = y_0) = \frac{P(x_0)P(y_0|x_0)}{P(y_0)} \tag{2.6}$$

It's time to do some exercises.

Exercise Let's have Robert a funny boys who is going to get a dog. In the shop there are 6 dogs: 4 males and 2 females and the shopper has to chose the dog for him randomly. Furthermore he has already thought about names: if dog is male the name is one of: Spike, Mike, Andy. Otherwise, if the animal is female, the name could be: Sally or Andy. In this situation we want to know the probability the dog will be called Sally, the probability that if the dog was called Andy it would be male.

First we need compute the probability to get a dog named Sally:

$$P(name = Sally) = P(dog = male) * P(name = Sally|dog = male) = 4/6 * 1/2 = 1/3$$

$P(dog = male|name = Andy)$ for this case we have to use Bayes' Rule: $\frac{P(dog=male)P(name=Andy|dog=male)}{P(name=Andy)}$.

We need the probability $P(name = Andy)$ and it is:

$P(dog = male) * P(name = Andy|dog = male) + P(dog = female) * P(name = Andy|dog = female) = 7/18$

Let's resolve this:

$\frac{4/6*1/3}{7/18} = 2/9 * 18/7 = 4/7$

2.3.2 Chain rule of probability

The chain rule let us decompose a joint distribution in a conditional distributions over only one variables. For examples we have 3 discrete variables:

$P(a, b, c)$ can be rewritten over c: $P(c) * P(b|c) * P(a|b, c)$

We can generalize this property using this formula:

$$P(x_0, ..., x_n) = P(x_0) \prod_{i=1}^{n} P(x_i|x_0, ..., x_{i-1}) \tag{2.7}$$

That is. Nothing more and nothing less

2.3.3 Marginal Probability

Let's imagine we have a set of discrete variables $x_0, ..., x_n \in \mathbb{S}$ and we know the probability distribution over them. Now we want to know the probability distribution over a subset of them $x_k, .., x_m \in \mathbb{S}'$. This distribution is called **Marginal distribution** and can be calculated using the **sum rule**:

$$\forall x_i \in \mathbb{S}', P(x_i) = \sum_{x \neq x_i \in \mathbb{S}} P(x_0, ..., x_i, ..., x_n) \tag{2.8}$$

For continuous variables the formula exploits the derivatives. Let's imagine we have two variable x, y:

$$p(x) = \int p(x, y) * dy \tag{2.9}$$

2.3.4 Relationships between variables

Let's have two variables x, y. We can say they are **independent** if their probability distribution can be expressed as the product of the tow factors:

$$p(x = x_i, y = y_i) = p(x = x_i)p(y = y_i)\forall x_i, y_i \tag{2.10}$$

we also can say they are **Conditional Independent** if given a event z the probability distribution can be factorized as:

$$p(x = x_i, y = y_i|z) = p(x = x_i|z)p(y = y_i|z) \tag{2.11}$$

If two variable are are independent we can denote that using: $x \perp y$. If two variable are conditional independent we can denote that using $x \perp y|z$

3 Let's start

In this chapter we are going to see the very basic of Neural Network like *Neurons*, *Links*, ...

3.1 Single Neuron

Neuron is the basic unit of Neural Network and is just a mathematical function applied to an input. Let's define an input a scalar i a scalar weight w, a scalar bias b and a function *f*. Neuron function is:

$$a = f(iw + b) \tag{3.1}$$

Function f is called **Activation Function** or sometimes **Transfer function** and usually is a non linear function.

3.1.1 Example of neuron

$i = 5$, $w = 0.5$, $b = 2$ and *f(x)* $= x + 1$

$$f(5 * 0.5 + 2) = f(4,5) = 5.5 \tag{3.2}$$

3.2 Transfer functions

Or Activation function are chosen during model creation in fact they are Hyper-Parameters. They can be linear or non-linear. Now some of the most important linear functions are listed:

- **Hard Limit Transfer Function**. This function produce 0 if input value is negative or 1 if value is equal or greater that 0. It creates a step and it is used for binary classification problems
- **Linear Function**. $f(x) = x * m + q$. It is the function of the line.

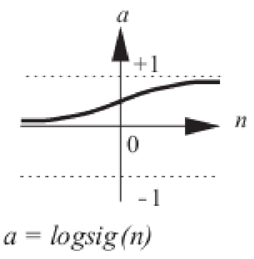
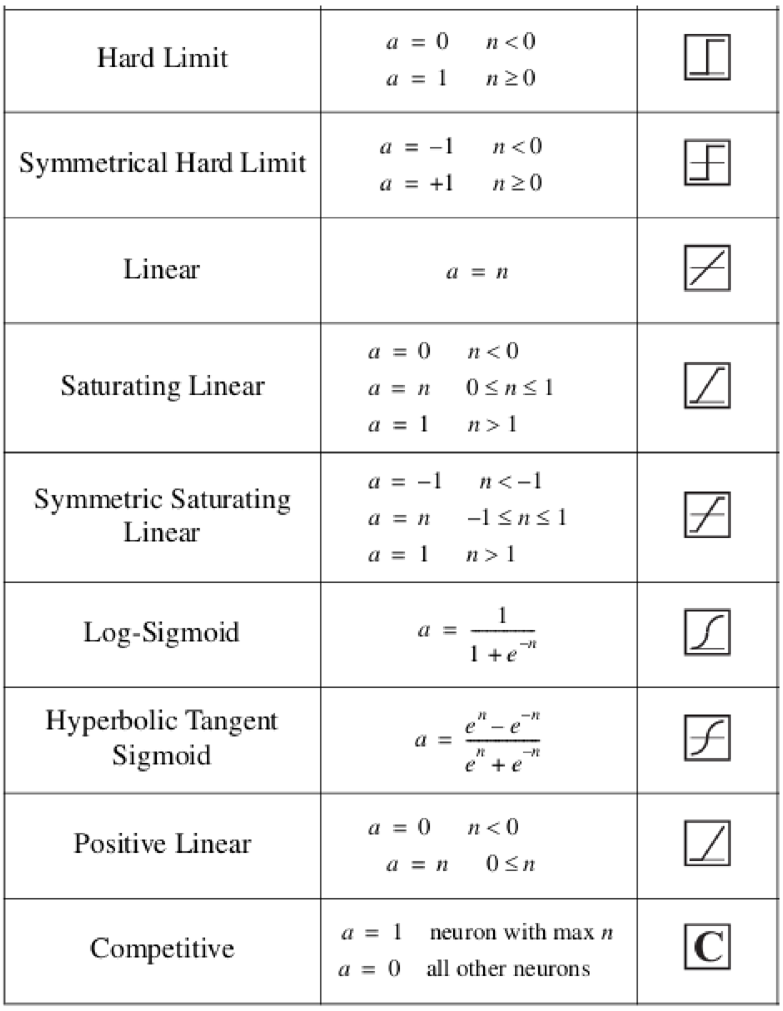


Figure 3.1: log-sigmoid function



Hard Limit	$a = 0 \quad n < 0$ $a = 1 \quad n \geq 0$	
Symmetrical Hard Limit	$a = -1 \quad n < 0$ $a = +1 \quad n \geq 0$	
Linear	$a = n$	
Saturating Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n \leq 1$ $a = 1 \quad n > 1$	
Symmetric Saturating Linear	$a = -1 \quad n < -1$ $a = n \quad -1 \leq n \leq 1$ $a = 1 \quad n > 1$	
Log-Sigmoid	$a = \frac{1}{1 + e^{-n}}$	
Hyperbolic Tangent Sigmoid	$a = \frac{e^{n} - e^{-n}}{e^{n} + e^{-n}}$	
Positive Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n$	
Competitive	$a = 1$ neuron with max n $a = 0$ all other neurons	$\boxed{C}$

Figure 3.2: Overview of basic activation functions

- **log-sigmoid transfer function**. $f(x) = \frac{1}{1+e^{-x}}$. This function produce a value always positive and you can see the graphic at the figure 3.1

Diagonalization

A matrix that represents a linear transformation can be *Diagonilized* by changing the basis in the one composed with only Eigenvectors. Each dimension is represented by one of those vectors so the matrix has to contain all eigenvalues to perform such operation. In this way it become a diagonal matrix filled by eigenvalues.

3.3 Multi-input Neuron

Usually neuron has more than one input at time. Each of them want a private w while b the bias is one for all. Under this light we can say that input $\mathbf{i}$ is a column vector of

$i_0, .., i_n$ elements and neuron contains a scalar b and a vectors of $\mathbf{w}$ of n weights. Using f as linear function $f(x) = x + 1$ the neuron perform this equation:

$$a = f(\mathbf{wi} + b) = f((\sum_{j=0}^{n} w_j * i_j) + b) \tag{3.3}$$

3.3.1 Example

$\mathbf{i} = 5, 6, 7$ $\mathbf{w} = 1, 2, -1$ $b = 2$ and *f(x)* $= x + 1$

$$f(5 * 1 + 6 * 2 + 7 * -1 + 2) = f(12) = 13 \tag{3.4}$$

3.4 Multi-neurons

Often Input vector goes to multiple neurons and these neurons are a Layer. It is important keep in mind that:

- All Neurons take all input. So w weights is a vector with the same dimension of the input vector
- All Neuron produce a scalar so the number of neurons in a layer is the number of outputs.
- Each neuron has its bias so a layer has a vector of bias one per neurons

Let's define input $\mathbf{i}$ a column vector and weight $\mathbf{W}$ as a matrix $m * n$ where m is the number of the neuron and n the number of the input elements in $\mathbf{i}$. f is the activation function:

$$\mathbf{a} = f(\mathbf{Wi} + b) \tag{3.5}$$

From the above equation we can see that a matmul is performed between weights matrix and input column vector. So row j, that is the weight vector of the neuron j is multiplied and summed with the whole input column vector.

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n} \\ w_{1,0} & w_{1,0} & \dots & w_{1,n} \\ \vdots & \vdots & \vdots & \vdots \\ w_{m,0} & w_{m,1} & \dots & w_{m,n} \end{bmatrix} * \begin{bmatrix} i_0 \\ i_1 \\ \vdots \\ i_n \end{bmatrix} \tag{3.6}$$

So a neural layer can be seen as a matrix of weight and a Bias vector:

$$Layer_w = \begin{bmatrix} n_0 \\ n_1 \\ n_2 \\ \vdots \\ n_m \end{bmatrix} = \begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n} \\ w_{1,0} & w_{1,0} & \dots & w_{1,n} \\ \vdots & \vdots & \vdots & \vdots \\ w_{m,0} & w_{m,1} & \dots & w_{m,n} \end{bmatrix} \tag{3.7}$$

$$Layer_b = [b_0, b_1, b_2, \ldots, b_m] \tag{3.8}$$

Mathematically a layer can be represents just by **W** the weight matrix and **b** the bias vector.

3.5 Multi Layers

A neural network has more layers. The first layer is called **Input layer** the last in called **Output layer** and the ones in the middle are **Hidden Layers**. Each layers can have different number of neurons and different activation function. So a basic multi-layer network with l layers is defined by $f_0, ..., f_l$ activation functions, $\mathbf{W}_0, ..., \mathbf{W}_l$ weight matrices and $\mathbf{b}_0, ..., \mathbf{b}_l$ or just **B** bias (matrix contains in row j the bias vector for the layer j). This network take inputs with the first layer, generate output and then gives this output to the second layer and so on. A Multi layer Network performs:

$$\mathbf{a} = f_l(f_{l-1}(\ldots * \mathbf{W}_{l-1} + \mathbf{b}_{l-1}) * \mathbf{W}_l + \mathbf{b}_l) \tag{3.9}$$

This Multi-layer neural Network is called **Feed Forward Neural Network**

3.6 Recurrent Neural Network

Feed forward neural network has layers and layer has neurons. Each Neuron take a input vector **i** and produce a scalar a. Differently, Recurrent Neural Network takes two input vector the input vector **i** and some *a'* that is the output of the previous input. Let's imagine we have a time series of data $i_0, ..., i_t$. This data must be given to the model respecting the sequence but we want model somehow remember previous data. For this kind of task Recurrent Neural Network was created. They have layers and neurons but each neuron has two vectors and a bias: u weights for the input, w weights for its layer output at the previous step and b the bias. Let me explain in a intuitive way before using math: the input is a series of vectors. First vector i_0 is taken and is given to the first layer of the model. Each neurons of this layer get the input vector i_0 and produce a scalar a_0 and with other neurons output of that layer it creates the layer output a Nothing new until now. Output goes to the second layer and so on until last layer. Second input is taken i_1 and neuron receives it and the previous layer output a. Using both of them it compute the next scalar output $nexta_0$. And so on!

Let's define its behavior using some cool math. Our network is composed by 1 layer with 1 neuron:

$$inputsequence = \mathbf{i}_0, \ldots, \mathbf{i}_k \tag{3.10}$$

$$f_{neuron} = ActivationFunc(b + \mathbf{w} * a'_{t-1} + \mathbf{u} * \mathbf{i}_t) = a_t \tag{3.11}$$

The neuron takes at time t in input $a'_{0,t-1}$ and $i_{0,t}$ and produce a_t. In this case with just one neuron the layer output coincide with the neuron output. After that the following equation is performed to compute a'_t

$$a'_t = c + \mathbf{v}a_t \tag{3.12}$$

Let's imagine to have a Recurrent Neural Network with more layers and more neurons for each layers. The above equations become:

$$\mathbf{a}_t = ActivationFunc(\mathbf{b} + \mathbf{W} * \mathbf{a'}_{l,t-1} + \mathbf{U} * \mathbf{i}_t) \tag{3.13}$$

$a'_{l,t-1}$ means the column vector output from the layer l at time step $t-1$

$$\mathbf{a'}_{l,t} = \mathbf{c} + \mathbf{V}\mathbf{a}_{l,t} \tag{3.14}$$

That is all my dear.. Not true, we are going to expand this kind of network later

3.6.1 Keep in mind

Some tips to keep in mind when you work with Recurrent Neural Network

- Externally are similar to feed forward network.. it has layers and neurons
- The inputs are sequence of inputs vectors
- Output is a series of vectors one for each inputs vector but if a prediction is what u want last output must be used.
- At any time neuron takes exactly 2 input vectors in input and they are the output from prior layer and output for its layer but a time t-1.
- For each input vectors neurons has a weight vector this means that a layer has a **W** and **U** weight. The first for its own output and the second for the output of the previous layer.
- Output of a layer is turned in a input performing $c + \mathbf{V}a_l$ where c is a bias and **V** is a weight matrix.

3.7 Some Applications

In this section we are going to see how the previously explained network are used in a very easy problem: *We have some fruits composed by oranges and apples and we need a model capable to tell us if a given fruit is one or it is the other. Fruits is defined by size, dimension, weight.*
Let be $i_a = \begin{vmatrix} 1 & -1 & -1 \end{vmatrix}$ the vector that represent an apple and $i_o = \begin{vmatrix} 1 & 1 & -1 \end{vmatrix}$ the one that represent an orange.

3.7.1 Perceptron

Perceptron is a binary classifier based on single layer **Feed Forward Network** with **Hard Limit** as activation function. So if the output is minor than 0 the function produces 0 otherwise 1.

$$output = HardLims(\mathbf{Wi} + \mathbf{b}) \tag{3.15}$$

W is a matrix *nxm* where n is the dimension of the input vector (3 in this case) and m the number of neurons in the layer.

Let's define $W = [0, 1, 0]$ for a single neuron layer. This is called **Single-Neuron Perceptron**. Now we can test it against our data.
Apple defined as $i_a = \begin{vmatrix} 1 & -1 & -1 \end{vmatrix}$ is multiplied by **W**

$$[0, 1, 0] * \begin{vmatrix} 1 & -1 & -1 \end{vmatrix}^T = 0 * 1 + 1 * -1 * 0 * 1 = -1 \tag{3.16}$$

For simplicity input vector is not in column but it should be. Don't judge me

$$hardlims(-1) = -1 \tag{3.17}$$

Ok! model classify apple as -1 let's check if orange are classified as 1

$$[0, 1, 0] * \begin{vmatrix} 1 & 1 & -1 \end{vmatrix}^T = 0 * 1 + 1 * 1 * 0 * 1 = 1 \tag{3.18}$$

$$hardlims(1) = 1 \tag{3.19}$$

OMG!!!! That is amazing... or not but anyway with a weight vector $[0, 1, 0]$ model is able to classify the fruits. Now try with banana and tell me if it works! COME ON!

4Basic Concepts about learning rules

In this chapter we are going to study in deep the basic concepts about learning rule. With *learning rule* it means all the rules and algorithms used to modify the weight and the bias of the network in order to improve the results. There are three main approaches:

- *Supervised learning.* In this case the net is trained by using a set of examples composed by (p_0, t_0) where p_0 is the input for the network and t_0 the corresponding correct result (target). The result produced is compared to the target. The learning rules are used to modify weights to make model produce the right result.

- *Reinforcement Learning.* The algorithm produce a score or a grade according to the network performance and the learning rule seek to adjust weights in order to get higher scores. The network continues to learn also during its application.

- *Unsupervised Learning.* The data does not have the target but they are somehow clustered. The cluster becomes the target to predict like the one in the supervised learning.

Probably, we will speak about all this approaches later.

4.1 Learning rules for Perceptron Architecture

4.1.1 Perceptron Network again

The perceptron network has a single layer of S neurons and it use as activation function the **Hardlim** (it produce 0 or 1 according to the output of the neuron). The weights of the network are:

$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_0 \\ \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_s \end{bmatrix} \tag{4.1}$$

So each neuron has a list of $w_{n0}, ..., w_{nm}$ weights where n it the neuron and m is the number of weight and it is the same for the number of inputs.

Each neuron produce a result o_n and the list with all results from all neurons is given to the activation function.

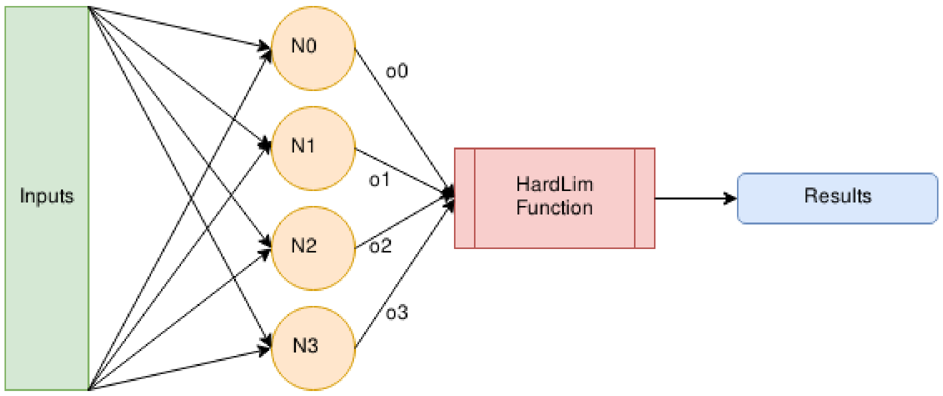


Figure 4.1: Overview of a Perceptron Network

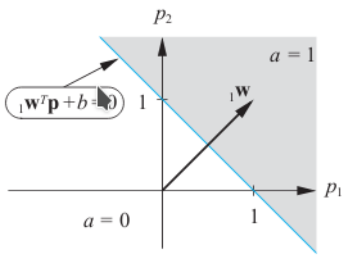


Figure 4.2: Graphic representation of the decision boundary

the decision boundary

Let's use a Perceptron Network with one neuron and 2 inputs. The result is obtained by this formula:

$$o = hardlim((w_0 * p_0 + w_1 * p_1) + b) \tag{4.2}$$

The decision boundary of the network is determined by the sub set of inputs that make the neurons produce zeros.
Example: Let's have $w_0 = 1$, $w_1 = 1$ and $b = -1$. So the decision boundary will be:

$$(w_0 * p_0 + w_1 * p_1) + b = p_0 + p_1 - 1 = 0 \Rightarrow p_0 + p_1 = 1 \tag{4.3}$$

The decision boundary define a line in the input space. On one side of that line, the network output will be 0, on the other side will be one. Modifying the weights and the bias of the network this line is moved up and down changing also its orientation.

4.1.2 Decision Boundary for Multiple Neurons

Network with S neurons has S lines, one for each of them. So each neuron can classify each input in just two categories. A network with one neuron has one decision boundary so it can produce just 2 category, while a network with S neurons can produce 2^s different output vectors so it can be used to classify an input to these categories.

4.2 Learning rules for Perceptron Network

There is a precise algorithm that tells us how to update the weights and it guarantees the best solution is found. The idea behind is if the model predict the wrong result weight must be modified to let the model classify that input. For this case let's define $e = t - a$ where t is the target and a is the output of the model. Each weight is updated according to this formula:

$$\mathbf{w}^{new} = \mathbf{w}^{old} + e\mathbf{p} \tag{4.4}$$

We can do the same for bias: Bias are just single weight for an input that is always one.

$$b^{new} = b^{old} + e \tag{4.5}$$

So, the perceptron rule can be written for the matrix of weight in case the network has more than one neuron.

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{ep} \tag{4.6}$$

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e} \tag{4.7}$$

That is all by now, but there is a proof of convergence that demonstrates this algorithm always find the best solution.

4.2.1 Example

Let's have a singular-neuron perceptron network for 3-input vectors: $i_0 = [1, -1, 0]$, $i_1 = [1, 1, -1]$, $i_2 = [1, 0, 0]$. For all of them we have the corresponding targets $t_0 = 1, t_1 = 0, t_2 = 1$. Let's initialize our net with random values: $w = [0, 0.5, -0.2], b = 0.3$. First, it can be useful find out the decision boundary:

$$(0 * i_{n0} + 0.5 * i_{n1} - 0.2 * i_{n2}) + 0.3) = 0 \Rightarrow \frac{i_{n1}}{2} - \frac{i_{n2}}{5} = 0.3 \tag{4.8}$$

Now we can proceed giving the first input to the network and see what happend:

$$hardlim((0*1+0.5*-1-0.2*0)+0.3) = hardlim(0-0,5+0+0.3) = hardlim(-0,2) = 0 \tag{4.9}$$

So $e = t - a = 1 - 0 = 1$. Let's apply the updating formula :

$$\mathbf{w}^{new} = \mathbf{w}^{old} + e\mathbf{p} = [0, 0.5, -0.2] + (1)[1, -1, 0] = [1, -0.5, -0.2] \tag{4.10}$$

$$b^{new} = b^{old} + e = 0.3 + (1) = 1.3 \tag{4.11}$$

Let's do it again but with the second input. (I gonna skip some calculus)

$$hardlim((1 * 1 + -0.5 * 1 - 0.2 * -1) + 1.3) = hardlim(2) = 1 \tag{4.12}$$

Again the model mistakes. $e = t - a = -1$. So let's repeat the update phase:

$$\mathbf{w}^{new} = \mathbf{w}^{old} + e\mathbf{p} = [1, -0.5, -0.2] + (-1)[1, 1, -1] = [0, -1.5, 0.8] \tag{4.13}$$

$$b^{new} = b^{old} + e = 1.3 + (-1) = 0.3 \tag{4.14}$$

Let's do it again but with the third input. (I gonna skip some calculus)

$$hardlim((0 * 1 - 1.5 * 0 + 0.8 * 0) + 0.3) = hardlim(0.3) = 1 \tag{4.15}$$

This time the prediction is good and $e = 0$
Let's test the new weights with the first input:

$$hardlim((0 * 1 - 1.5 * -1 + 0.8 * 0) + 0.3) = hardlim(1.8) = 1 \tag{4.16}$$

Good, it is correct. And at last, let's check for second input:

$$hardlim((0 * 1 - 1.5 * 1 + 0.8 * -1) + 0.3) = hardlim(-2) = 0 \tag{4.17}$$

That is cool! Now model can predict the right result for all of them!

4.3 Perceptron Network Limitation

This network can find the best solution of classification problem in a finite number of steps, however it can cut the input space using lines (hyperplane). It can't model create decision boundary not-linear, so if data are a long some curve they can't be classified. For example this network can learn to produce AND but it can't reproduce XOR. Try to think why by yourself.

5Training a model

The main ingredients of a training phase are:

- The Model: the neural network that contains weight and parameters
- The dataset: it is the source where extract information and knowledge
- Loss function: the function that tells the model what is learning
- Optimizer: the technique used to update the weights

When u want to train a model u can use different approaches:

- Supervised Learning: when you have labeled data
- Unsupervised Learning when your data are not labeled
- Reinforcement Learning: when your model try to learn through experience and not data

But for each kind of learning you have to set up a function that tell to the model how to improve him self. This function is called **Objective Function, Loss Function, Target Function etc etc** and it is fundamental for:

- Tell the model how to update weights in order to get better result
- Control the training of the model
- Add extra information to the model
- Avoid bad outcome (Overfitting, not-convergence)

5.1 Objective function: overview

5.2 Different kinds of Loss Function

5.3 Optimizer

Let's say we have a very easy model that we can represent using the equation:

$$M(\mathbf{x}) = Tanh(\mathbf{x}\mathbf{W} + \mathbf{b}) \tag{5.1}$$

Let's say the input is a vector of 2 dimension (x_0, x_1) and the output is a 2 dims vector (o_0, o_1). So the matrix of weights $\mathbf{W}$ is a 2x2 matrix. Let's define a loss function like the MSE $= \frac{\sum_{i=0}^{2}(t_i - o_i)^2}{2}$

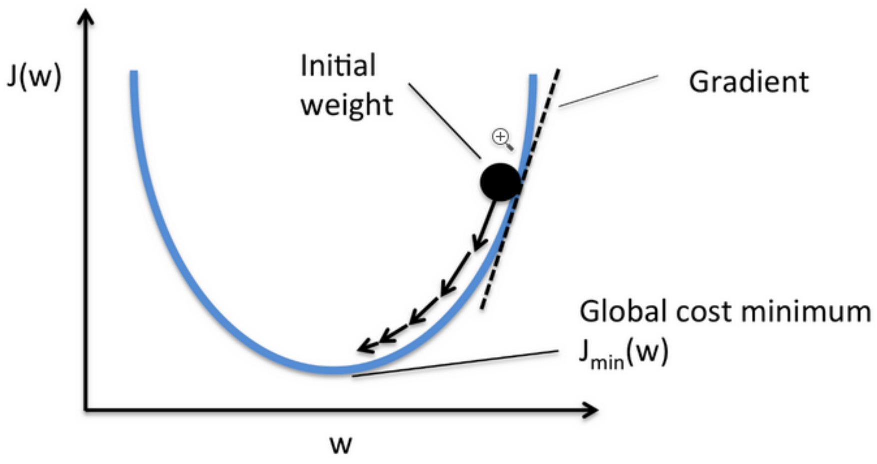


Figure 5.1: How the backpropagation works

Now we have our sample $(x_n, t_n) = ([1, 1], [0, 0])$ and the model generates $[1, -1]$ so we compute the MSE $= 1$. We want our model itself and reduce the loss but how. In this case in order to reduce the loss we would need to reduce the difference between our model prediction and target $0 - 1, 0 - (-1)$. So we would need to reduce the first value and raise the second. The first value is obtained through the equation:

$$Tanh(\mathbf{x} * \mathbf{w}_{c1} + b) \tag{5.2}$$

where $\mathbf{w}_{c1}$ is the first column of W and b is the first value of the bias vector. In the same way we know the second value is obtained using the equation:

$$Tanh(\mathbf{x} * \mathbf{w}_{c2} + b) \tag{5.3}$$

So in order to minimize the loss function we need to reduce $\mathbf{w}_{c1}$ and raise $\mathbf{w}_{c2}$.

5.3.1 BackPropagation and lr

So the idea is we needs to adjust each parameters in a model in order to minimize the loss. In order to automatize this process we need a technique that determines for each parameter if it needs to be raised or lowered. This technique is called **Back Propagation of the Gradient**. The idea is: the derivative of a function respect a variable tells us if moving forward or backward that variable the output of the function will raise or decrease. The is exactly what we need.

Let's define the **Gradient** as the vectors that contains the partial derivative of each parameters respect the loss function. If the partial derivative is positive so we know that, in order to lower the loss, we need to reduce the value of that parameter otherwise we need to raise the value of that parameter. Geometrically the gradient is the vector that points to the direction where function raise most. So the update step is doing by:

$$w_i^t = w_i^{t-1} + \alpha \nabla_{wi} \tag{5.4}$$

where w_i^t is the weight i at the time t, $\bigtriangledown_{wi}$ is the gradient of the function, in this case it represent the partial derivative of the loss function respect the parameter wi. The α represent how much the weight will be incresed or decreased. This value is called **Learning Rate**

So this is the overview on how weight are update but what is an Optimizer? The optimizer is the selected technique to update weight, the Back propagation of the Gradient is a technique but also the basic technique, the others will extend this concept in order to make the process faster.

5.3.2

6Markov Decision Processes

Paper: A survey of Application of Markov decision processes Paper: Markov decision Process: concepts and algorithms

7 Generative Models

In this chapter I'm gonna put all stuff about generative field. I'd lie to speak about GANs network and other advanced stuff but before I need to put together some backgrounds concepts like Boltzman machine Energy-based model and so on.

7.1 Energy based model

Let's say that a probability function $p(x)$ needs to respects this rules:

- $\forall x, p(x) >= 0$
- $\sum_x p(x) = 1$

You can have a more deep definition in the probability chapter. In order to define this $p(x)$ function we need to guarantee the respect of those constrains. Let's imagine $p(x) = x^2$ In this case the first rule is respected but not the second, so this function can be used. So, to use a positive function we need to normalize it. Let's define $p'(x)$ as the normalized version of $p(x)$ in order to respect both the rule:

$$p'(x) = \frac{p(x)}{\int p(x)dx} \tag{7.1}$$

We have normalized $p(x)$ by the area above the the function.

Now we can define the probability function $p(x)$ the function:

$$p(x) = \frac{exp(f(x)}{\int exp(f(x))dx} \tag{7.2}$$

We can define:

$$Z = \int exp(f(x))dx \tag{7.3}$$

As you can see it was used $exp()$ function in order to guarantee the first rule. Furthermore this function allow to better capture variations, many distribution can be written in exponential form.

In this system we can call $f(x)$ the **The energy function** and we see the method to turn it in to a probability function

7.1.1 How it work?

The **Energy based model** has to associate to each configuration of the input variables a scalar that indicates how much energy has this configuration. A good configuration has low energy, so a low value while a bad one has a greater energy.

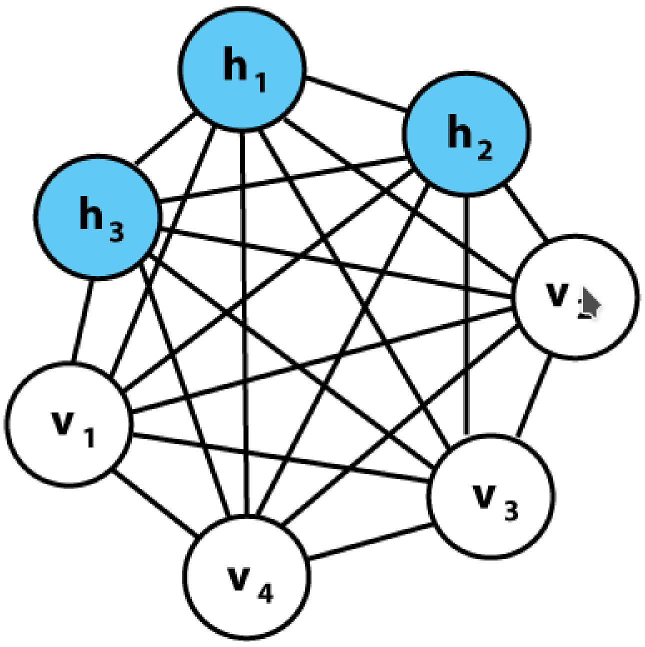


Figure 7.1: boltzman machine model

For example let's say we have in imagine M and a set of object $(s_0, s_1, ..s_5)$. The imagine contains one object from the set and we want create a model capable to identify it. Let's say that we want a model capable to give to the right s_i lesser energy than the other.

7.2 Boltzman Machine

Boltzman machine is a symmetrically connected neurons model. Each of these neurons has a binary state so they can be on or off. A Boltzman machine is composed by nodes, called, neurons, connected to the others with a weighted connection. Nodes can be visible of hidden. This means the user will be able to set input in the visible neurons not in the hidden ones.

Each neurons compute its state according to a probability function that means the probability of the neuron to change state:

$$prob(s_i = 1) = \frac{1}{1 + e^{-z_i}} \tag{7.4}$$

As we can see this formula uses a parameter z_i that is computed by summing all weighted states of input nodes plus a bias:

$$z_i = b_i + \sum_j s_j w_{ij} \tag{7.5}$$

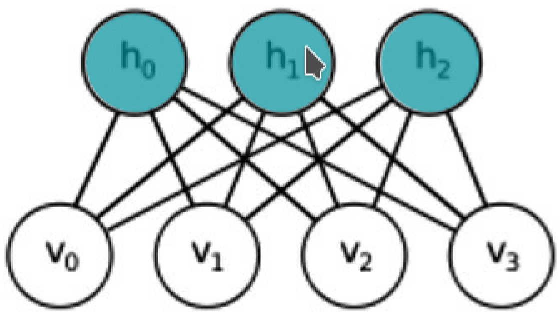


Figure 7.2: restricted boltzman machine model

Now, once all nodes are update, we can compute the **Energy of the system** using a **Boltzman Distribution** that is a energy probability function for discrete value.

Let's define $\vec{v}$ the vector containing all states and $\mathbb{U}$ all the possible state vectors. Now we can compute the probability of the system using:

$$P(\vec{v}) = \frac{exp(-E(\vec{v}))}{\sum_{\vec{u}\in\mathbb{U}} exp(-E(\vec{u}))} \tag{7.6}$$

$E()$ is the boltzman energy function:

$$E(\vec{v}) = -\sum_i s_i^v b_i - \sum_{i<j} s_i^v s_j^v w_{ij} \tag{7.7}$$

The goal of this machine is to set weighs in order to minimize the energy of the system for the right configuration of states, and maximize it for the wrong configuration. **Problems:** Nodes are all connected to each other and that makes the model learning really long. That why **Restricted Boltzman Machines (RBM)** came into the picture

7.2.1 Restricted Boltzman Machine

The main difference between Boltzman Machine and Restricted Boltzman Machine is that visible nodes are link just to hiddens one and viceversa. Less connections less learning time. The State vector $\vec{v'}$ is composed by the visible state vector $\vec{v}$ and the hidden vector $\vec{h}$. The energy is computed:

$$E(v,h) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum v_i h_i w_i j \tag{7.8}$$

Where a_i and b_i are the bias. The probability of the system can be expressed as:

$$p(v,h) = \frac{exp(-E(v,h))}{\sum_{v,h} exp(-E(v,h))} \tag{7.9}$$

Now it si possible to calculate the probability of a single hidden neuron to be activated:

$$p(h_j = 1|v) = \frac{1}{1 + exp(-(b_j + W - jv_j))} = \sigma(a_i + \sum_j h_j w_i j) \tag{7.10}$$

As we can see from the above equation it can be expressed as a logistic function

8EXTRA: Neural Symbolic Integration

8.1 Introduction to Neural Symbolic Integration

Well, it is a real chaos out there but let's try to put things together. With **Neural Symbolic Integration** we mean the set of techniques used to combine Symbolic Approach with Neural network Approach. At the very beginning of the AI (1970 more or less) there were two different directions: **Symbolic AI** and **Connectionist AI** (or sub-symbolic). The former wants to explain the knowledge and the reasoning to get a task done by using Logic Rules as Prolog, or Descriptive Logic (Cond, Cond => State). The latter, on the other hand, wants to get best results with less effort. From this direction was born Machine Learning and in particular Deep Learning.

So, on one hand we have symbolic approaches that are clear, transparent but less precises, less powerful, less applicable in real context due to scalability, distribution and so on. Those approaches are based on Logic (Symbols, Relation between them, abstract connection) and the manipulation of them and this approach is considered *Top-Down* knowledge is from abstract representation using Logic and not in raw data and experience.

On the other hand we have connectionist approaches or sub-symbolic approaches that totally lacks of transparency but they can handle big data, they can scale and be distributed, they are power with good result. They also lack of a real reasoning. This approach use statistical function to extract information from raw data so it is called **Bottom-Up** because knowledge comes from raw data. They learn trough experience.

So the **Neural Symbolic Integration** is the research fields that studies technique to put them together and get the most from both. The field also called Neural Symbolic Computation is one of the many that exist but it is the most promising so we speak about it as the only one. This field obtained new interesting by the research community very recently so it lacks of common tasks, common formalism and also common objectives that can make possible clear comparison between solutions. There are a lot of different solutions in the wild very different and the only thing in common is they try to put some logic with neural network.

But in this heterogeneous mixing we can cluster solution in two groups:

- **Logic as Constrain** where logic is inserted directly into the neural model that to guide the training process, working as regularization factor. In this way a trained network has high-level knowledge inside.

- **Differentiable Programming** or **Logic as Task**. The idea is to create a mapping between logic and numerical domain in order to use Neural network on the second to solve problems in the first. Logic guides the construction of an Equivalent Neural Network.

8.1.1 Logic As Constrain

8.1.2 Logic As Task

8.2 Neural Symbolic Cognitive Agent (NSCA)

from paper *A Neural-Symbolic Cognitive Agent for Online Learning and Reasoning*. The first point is existing models are too simplified and require too time to be used in online learning and reasoning. Furthermore, some high-order concepts have temporal relation hard to represent by hand. They propose a new model capable to learn new hypotheses from observed data in complex environment. More precisely the model is able to:

- Perform learning of complex temporal relations from uncertain observations
- Reason probabilistically about the knowledge learnd
- Represent the agent's knowledge in a logic-based format for validation purposes.

It takes advance of Statistical learning and Symbolic Representation

8.2.1 References

https://arxiv.org/pdf/1905.06088.pdf - Neural-Symbolic Computing: An Effective Methodology for Principled Integration of Machine Learning and Reasoning

9Reinforcement Learning

This chapter is my notes from a Udemy Course called **Deep Reinforcement Learning 2.0**. During the course I will see theory notions but implementations of some AI that keeps learn. I will face Q-Learning and other cool stuff.
The structure of the course will be:

- Extra theory
- Foundamentals
- Twin Delayed DDPG - Theory
- Twin Delayed DDPG - Implementation

9.1 Resources

`https://arxiv.org/pdf/1802.09477.pdf`. It is the paper that we will discuss dring the course

9.2 Extra theory: basic knowledge

In this section we will see what is reinforcement learning, Bellman equation, Markov decision process, plans, living penalty, q-learning, temporal difference.

9.2.1 Reinforcement Learning

The main ingredients of a RL systems are:

- Agent: the AI that has to fulfill a task by taking actions
- Environment: the place where Agent lives
- Action and rewards: mechanism that make Agent do thing and discriminate if it is right or not. Action are hard coupled with time.. a correct action in a wrong time leads a bad rewards.

Training AI is like training a dog, if it does right it get a reward otherwise nothing.
RL create RL agents that differs from Pre-Programmed Agents by the knowledge they have inside before start learning. The PPAgents have a series of rule that describe all actions they can do, in which case etc etc... while RL has to figure out those rule, his own behavior.

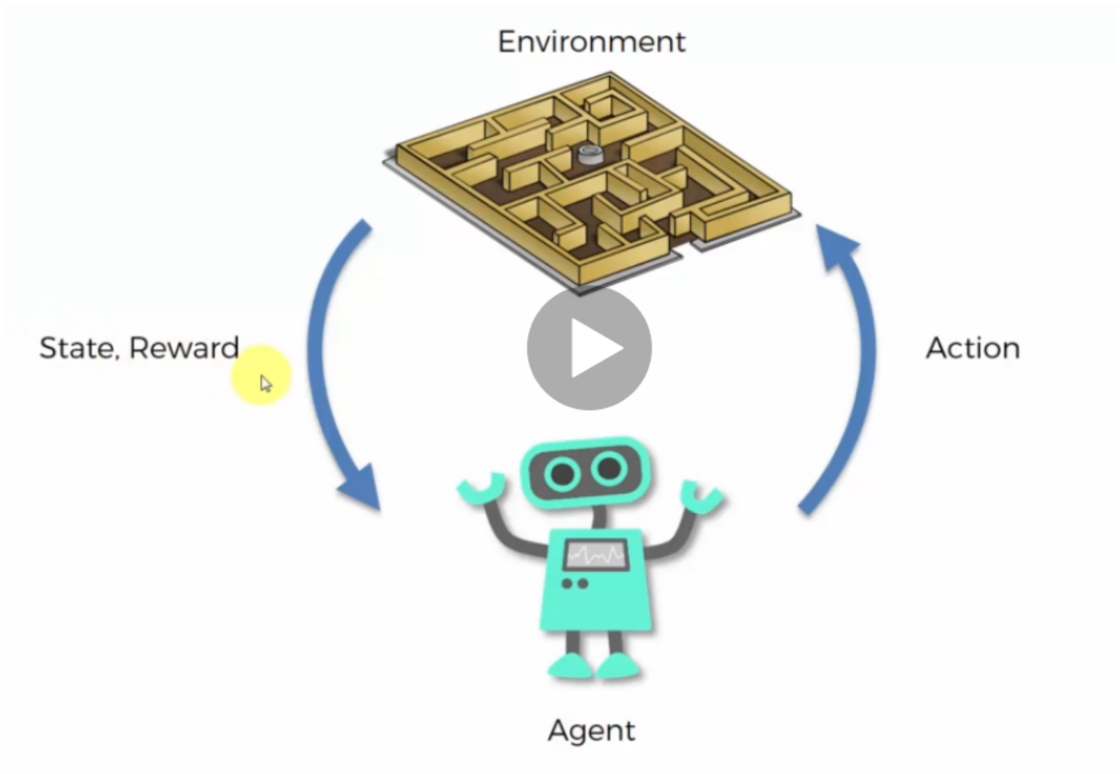


Figure 9.1: Overview of Reinforcement Learning system

9.2.2 BellMan Equation

Notions:

- s the state of the agent
- a action that agent can make
- R reward agent gets for entering in a State
- y Discount.

Before start let's define a task. We have an agent (the robot) in a maze composed by a grid with three different kinds of cells:

- Empy: the withe cell where agent is free to go
- Grey: the wall, agent cant cross this cell
- Red: The fire, agent can go there but it loses
- Green: the target, agent can go there and it wins

You can see this amazing maze in the picture 9.3 The agent can do 4 actions:

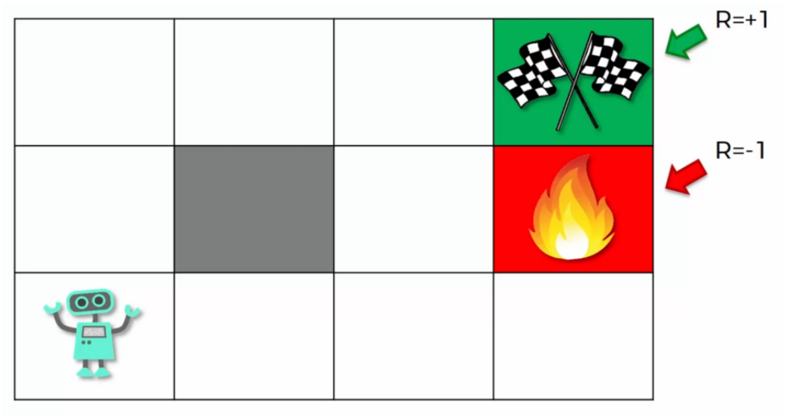


Figure 9.2: RL - Maze task

- Go Up
- Go Down
- Go Right
- Go Left

Basic Idea Of Bellman equation. When agent goes to the green cell it gets a $+1$ rewards. It likes rewards, it is addicted, it cant live without so it think: "how i get the reward?" in order to be able to get it each time. So it learns that from the previous cell (3,3) it has to just move right (4,3) and he get the $+1$. Now each time it goes in (3,3) he knows how to get reward. He is happy but he could be more happy, more addicted. He start to think: "how i get here in the (3,3)" in order to be able to get this cell every time. So it learn for example that it can reach this cell from (3,2). Then it ask the same until it has learned a path from the start point to the green cell. Now it is really happy.

The bellman equation. This is the bellman equation:

$$V(s) = max_a(R(s,a) + yV(s')) \tag{9.1}$$

This equation gives a value V to a State s reached by the agent. The value is based on the best action it can take from that state. Let dig into it. We know agent in a state can take actions a, in our case it can chose between 4 actions each of them leads the action is a new state s'. Each action, took in a state has a reward for example if agent is in (3,3) and it goes right it get $+1$ as reward, if it is in (3,2) and it goes right it get -1 as reward because it ends in the fire. So each action, according to the current state, leads to different rewards $R(a,s)$. In the bellman equation we select the action that give the maximum value. So we cycle on all possible action computing the reward $R(s,a)$ and we select the action that brings agent to the greatest reward. We also consider the value of the reached state. Take an action brings agent from s to s' so the value of s' must be taken in account $yV(s')$. So for each action we compute the reward $R(s,a)$ plus

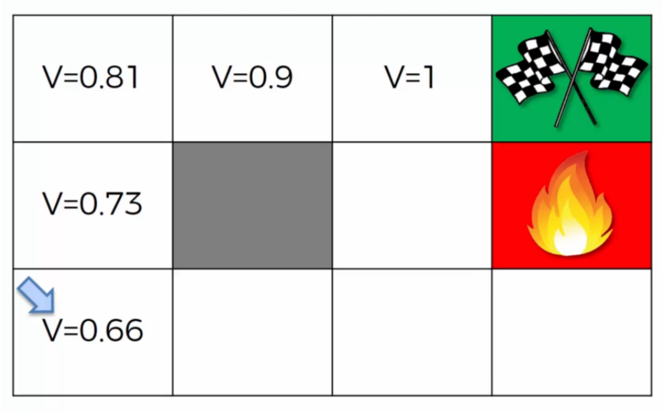


Figure 9.3: RL - Maze task with values from bellman equation

the value of the reached state $yV(s')$ and we select the action that gives the agent the greatest value because we want agent be optimal, so we can consider just best solutions.

Let's do some example Let's compute the $V(s)$ of the (3,3) using a $y = 0.9$. $V((3,3)) = R((3,3), goRight) + yV((3,4))$. We already know the best action because it is the only action that leads agent to the target (3,4). So considering this action we compute the value.

$V((3,3)) = 1 + y0$ V(3,4) is zero cause it is the target cell and we don't need it to have a value So $V((3,3)) = 1$

We can repeat this for the V(2,3), we know the best action is go right so $V(2,3) = 0 + 0.9 * 1 = 0.9$. We can repeat the it for each cell.

It easy to see that value decreases according to the distance to the target cell (4,3).

9.2.3 Stochastic variable... OMG

Untill now we speak about the maze a a very kind one. U want to go up, then u go up with 100% possibility. But this is not how real life, out there, works. Let's add some randomness, let's create a stochastic maze where the action is a stochastic process. For example in cell (2,3) ground is broken so if u want to go up you have:

- 80% of probability to go up
- 10% of fall down
- 10% of fall in the cell with the hell's fire and die.

This is a **Markov Process** because the state s' is expressed just according to the current state and not the previous decision of the agent. (Check the chapter about Markov Process).

That is how real work works. Our robot, that is addicted wish to get to the cell (3,3) because it perfectly know how to reach is drug from there. So from (3,2) decide to go up but it risk to fail and worsening his position. So the value of this cell must decrease.

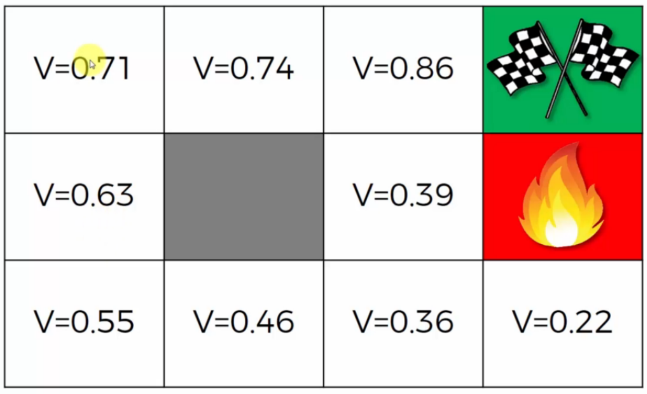


Figure 9.4: RL - Maze task with all values changed because in each cell there is a probability to fall in two wrong cells

Let's define the new Bellman Equation considering the Markov Decision Process:

$$V(s) = max_a(R(s,a) + y\sum_n P(s,a,s'_n)V(s'_n)) \tag{9.2}$$

Let's explain that, now agent can end in different states $s'_1 s'_2 ... s'_n$ and it can end there with a certain probability. Let's define P() as a function that gives to us this probability: $P(s,a,s'_n)$ gives to us the probability from s to reach s'_n by action a. For each possible state s'_n we need to compute this probability and multiply it with the value of that state and sum all values obtained.

Let's do an example. Let's compute the value of (3,2) considering the hard ground. $V((3,2)) = 0 + 0.9 * ((0.8 * 1) + (0.1 * 0.73) + (0.1 * -1)) = 0.6957$

From pics 9.4 it's possbile to see the new cell values considering the probability to end in the wrong cells (80%, 10%, 10%). That is why the agent in the cell (3,2) wont try to go up because he know that he have 10% of probability to end in the fire an die. But it is more convenient for him to try to go in the wall an stay in the cell (3,2) but have a 20% of probability to fall in a better cell without risk to end in the fire. It's possible to see it in the picture 9.5.

9.2.4 Rewarding

In our maze, by now, agent get two rewards the +1 if it arrives in the (4,3) cell (the green) or -1 if it ends in the fire. But what takes the agent away from waste time? Maybe it wants to walk a bit before get the reward? But we want agent be more addicted possible, we want it suffering the pain of hell if it doesn't get fast his drug!! Let's introduce **The Living Penalty**, a negative reward that it gets each time it take and actions. More action it takes more penalty if collects. Using this it select different action, for example it stops to try to go in the wall, has it does without penalty. You can see this differences in the picture 9.6

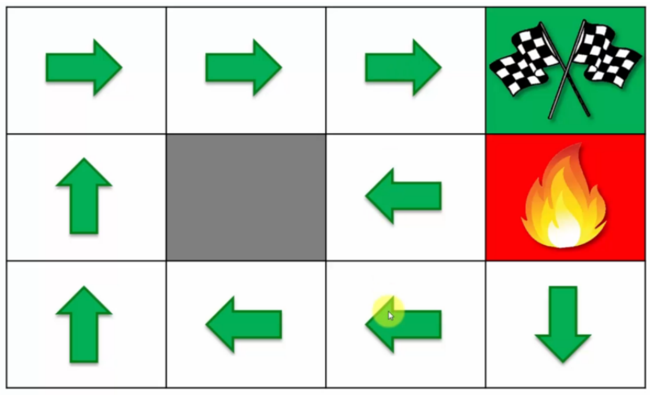


Figure 9.5: RL - Maze task with all values changed because in each cell there is a probability to fall in two wrong cells. We can see the actions that agent will try to take in each cell

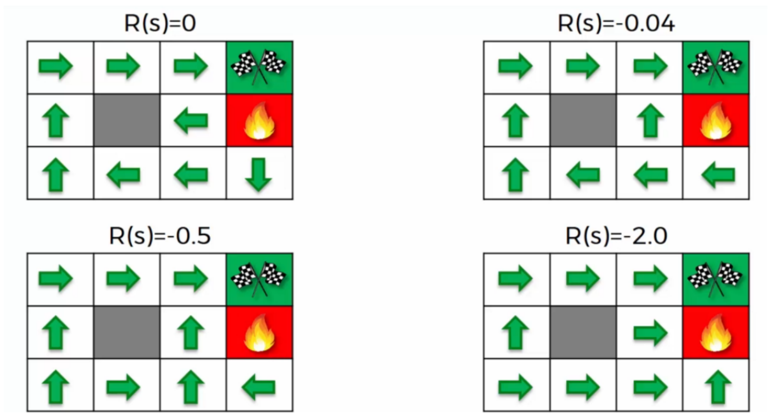


Figure 9.6: RL - Actions for the maze with different Living Penalty

9.2.5 Q-Learning

By now, using bellman equation, the best action is chosen by knowing the value of a state and selecting the action that brings agent it that state. In Q-Learning approach the idea is to evaluate directly the action using a *Quality function* to measure the quality of an action $Q(s,a)$ where s is the state of the agent and a is the action. The $Q(s,a)$ equation come from this equation:

$$Q(s,a) = R(s,a) + y\sum_{n} P(s,a,s'_n)V(s'_n) \tag{9.3}$$

As we can see it is identical to the final piece of bellman equation. But we still have $V(s)$ in our equation and we don't want it because it focuses on value of state and we want focus on value of possible action:

$$Q(s,a) = R(s,a) + y\sum_{n} P(s,a,s'_n)max_{a'}Q(s'_n,a') \tag{9.4}$$

9.2.6 Temporal Difference

Now it obvious ask ourselves if the same action at different time can have different value or precisely different q-function results. Let's introduce the temporal difference concept.

Intuition. Let's say our agent is for the first time in the maze. Let's place him in (1,1) at the very start. Now he can go in both direction (2,1) o (1,2). By now he has no idea of the path for the reward so he evaluates them equally: $Q(a_0,s) = Q(a_1,s)$. After some exploration he is back in the first cell (1,1). The cell is the same as before, the possible actions are the same as before but now he knows the best path so $Q(a_0,s) > Q(a_1,s)$. That means that $Q_{t_0}(a_0,s) < Q_{t_n}(a_0,s)$ because agent learned the best path. The same action from the same cell can have different importance according to the time it is performed because agent learn at each time.

Let's define this concept introducing t the time:

$$Q_{t_1}(s,a) = Q_{t_0}(s,a) + Q_{t_1}(s,a) - Q_{t_0}(s,a) \tag{9.5}$$

From the above equation we can see that the Q at time t is the Q at time $t-1$ plus the difference between them. Very easy .. doesn't it?. This difference is what agent has learned. Let's call $Q_{t_1}(s,a) - Q_{t_0}(s,a) = TD(s,a)$ the temporal difference between the same action a in the same state s but in different times. We can now rewrite the equation as:

$$Q_{t_1}(s,a) = Q_{t_0}(s,a) + \alpha TD(s,a) \tag{9.6}$$

α is a parameter that tell to the agent how much care to the past Q_{t_0} or to the knowledge learned TD In order to better understand let's expand the equation:

$$Q_{t_1}(s,a) = Q_{t_0}(s,a) + \alpha(R(s,a) + max_a Q(s',a') - Q_{t_0}(s,a)) \tag{9.7}$$

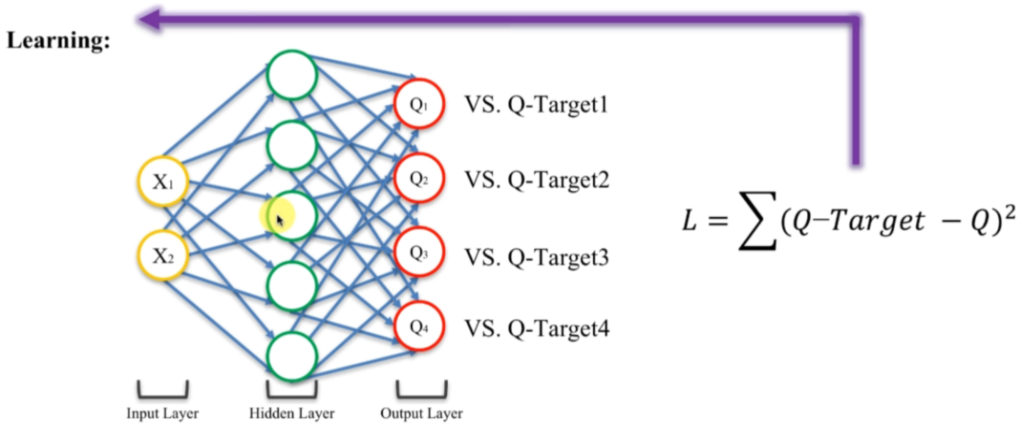


Figure 9.7: RL - Deep Neural Network overview for deep q-learning

From this equation we can see that if α is set to 1 we have this:

$$Q_{t_1}(s,a) = R(s,a) + max_a Q(s',a') \tag{9.8}$$

So the entire old value of Q_{t_0} is replaced be the new value that comes from what agent has learned (that is inside: $max_a Q(s',a')$). Otherwise, if we put α to 0 we have:

$$Q_{t_1}(s,a) = Q_{t_0}(s,a) \tag{9.9}$$

From this equation we can see that nothing has changed so agent has learned nothing. This means that α is a coefficient that weight how much agent has to update its knowledge according to his new discoveries. Let's call α **learning rate** and this process **Q-Learning Process**. At the beginning of this process $Q(s,a)$ will be very low (or wrong) and the TD very high. After some passages, when agent has learned the optimal path so the optimal action $Q(s,a)$ will be high (or optimal) and TD will be low because agent can't learn something new.

9.3 Deep Q-Learning

The idea of **Deep Q-Learning** is to use a neural network to learn the best action to do in a given place. So the model should take in input s expressed as a pair $< x_0, x_1 >$ and it tries to predict the right action. So it gives in output a vector $< q_0, q_1, q_2, q_3 >$ that represent the q-learning value associated to each action. *This algorithm (deep q-learning) can be applied just when you have discrete actions like in this case.*

9.3.1 Experience Replay

The agent performs some actions and save them in a memory as experience. Then it select some of the past action, using a uniform distribution, and evaluate them.

`To read: Prioritized Experience Replay - google deepmind 2016`
Experience Replay has different implementation so we have to define some policies like: max number of experience to save, how to discharge past experience for new ones, how to select them etc etc.

9.3.2 Action Selection Policy

9.3.3 Deep Q-Learning: The algorithm

First of all we need to put agent in a context. Then we need to define **Experience Replay Policy** and we are ready to start. So our agent will be in a state s and he can do the same 4 actions: up, down, right, left. So he will try to do the best action predicting the Q-Value for each possible action. Let's say he predicts $[Q^p(s,up), Q^p(s,down), Q^p(s,right), Q^p(s,left)]$. The first time he will predict random value for instance: $[0.1, 0.1, 0.4, 0.05]$. According to our action selection policy he will select one action to do, let's say it select the one with the best score so: right. Now agent moves right and enters in a new state s'. Let's say he also get a reward $r = 1$ so now he can compute a better q-value for the action *right* from state s: $Q^t(s,right) = R(s,right) + ymax_a Q(s',a) = r + 0 = 1$. Now agent can save in memory the experience using the tuple $(state = s, action = right, reward = r, end_s tate = s')$, for clarity let's express this tuple using time because agent can experience the same situation more times so $(state_t = s, action_t = right, reward_t = r, end_s tate_t = s')$. He also save for that experience the predicted q-values.

It's time for the agent to learn. He predicted for this action the q-value $Q^p_{right} = 0.4$ then he performed the action and he discovered that a more precise q-value was $Q^t(s,right) = 1$ so he can use the loss function $Loss(Q^p, Q^t) = (Q^t - Q^p)^2$ and than he can back-propagate the gradient of the function to update weights as in a normal Neural Network.

Let's add a new complexity, the memory. When agent has to learn he doesn't use just that experience but he select from his memory also N - 1 past experiences $e_0, e_1, ..., e_{n-1}$. He adds the new experience he has just performed and he get a pool of N experiences. Now it try to learn also form those experiences using the loss function:

$$Loss = \frac{1}{2}\sum_e ((Q^t(s,a) - Q^p_e(s,a))^2 \tag{9.10}$$

9.4 Formula Summary

Bellman Equation:

$$V(s) = max_a(R(s,a) + yV(s')) \tag{9.11}$$

Bellman Equation in a Markov Decision Process:

$$V(s) = max_a(R(s,a) + y\sum_n P(s,a,s'_n)V(s'_n)) \tag{9.12}$$

Q-Learning Equation:

$$Q(s,a) = R(s,a) + y\sum_n P(s,a,s'_n)V(s'_n) = R(s,a) + y\sum_n P(s,a,s'_n)max_{a'}Q(s'_n,a') \tag{9.13}$$

Q-Learning Equation simplified (*This equation substitutes the previous just because it is shorter, for simplicity*):

$$Q(s,a) = R(s,a) + ymax_{a'}Q(s'_n,a') \tag{9.14}$$

Temporal Difference:

$$TD(s,a) = Q_{t1}(s,a) - Q_{t0}(s,a) \tag{9.15}$$

$$Q_{t1}(s,a) = Q_{t0}(s,a) + \alpha(Q_{t1}(s,a) - Q_{t0}(s,a)) = Q_{t0}(s,a) + \alpha TD(s,a) \tag{9.16}$$

$$Q_{t_1}(s,a) = Q_{t_0}(s,a) + \alpha(R(s,a) + max_a Q(s',a') - Q_{t_0}(s,a)) \tag{9.17}$$