POLITECNICO DI TORINO

Master Degree in Electronic Engineering

Master Degree Thesis

Artificial Intuition

Supervisor:
Professor Mariagrazia Graziano

Candidate:
Gianluca Leone

April 2019

Table of contents

List of figures

Chapter 1

Abstract

The aim of this thesis is to explore a new field of Artificial Intelligence. The scientific literature is rich of systems able to emulate the logic human reasoning, although they are not proficient in answering by intuition, like humans continuously do. In the first chapters an investigation is carried out to understand what intuition is, and thanks to psychological researches the ambiguity around this term is clarified. The psychologists point of view show us a mind behavioural model, composed by two subsystems, which perform different tasks. The so called System 1, responsible of the intuition, is described as an associative machine, which surfs in an ocean of linked ideas. The approach of this thesis is so to implement both the ocean and the navigator: by this perspective the knowledge becomes a series of points of a semantic significant space, and the navigator an agent who learns which routes lead to the best results. The found outcomes show that this agent can move in the semantic space and recognizes valuable points, based on its past experience, getting intuitions.

Chapter 2

Introduction

Artificial Intelligence

Artificial Intelligence explores the implementability of human cognitive functions. The most popular applications, such as learning, decision-making, image processing and language understanding are studied since AI birth. A huge amount of materials has been produced up to date and scientific literature is rich of studies and working implementations of systems capable to satisfactorily emulate human features.

Human brain is the main character, responsible of all human-being's skills. It is not surprisingly that to reproduce brain capabilities, the architectures implied are inspired from brain structure. It is a made of fact that neural networks (NN) are the main engine of AI. Algorithm NNs implement are not write from developers, NNs learn how to behave starting from labelled examples (supervised learning) or with only a positive or negative feedbacks as consequence of their actions (unsupervised learning).

Recent examples of AI goals are DeepMind's Wavenet [1], which improves Google Assistant voice synthesis with deep neural networks (DNN); [2] automatically generates captions to describe images; [11] play to 49 different games of the Atari 2600 console with impressive results. A less recent example is [4], capable to extract subject, object and action from a rich number of different grammatical structures.

AI products are so successful to cover important roles in the ranks of economic giants. For instance, YouTube recently published and introduced a new algorithm for video recommendation based on DNNs which try to maximize the time users spent inside the platform [5].

Intuition

Our lives are continuous streams of new experiences. We learn to grab, to speak, to walk, to run... More we learn, more our background become useful to solve problems we will face. The most is copied, observed and replicated, but something, sometimes, it's new.

Understanding and reasoning are the main and most unknown human mind capabilities. But, even if they are largely studied by AI, they aren't the only way we use to solve problems, maybe not even the most frequently used. Intuition is "*The ability to understand something instinctively, without the need for conscious reasoning*" [6]. It's more and more quicker than logic reasoning and also powerful, under a certain point of view, because in zero time it jumps the unknowns and reaches the solution. Zero or almost zero time implies zero or almost zero computation, i.e. reasoning for humans. Therefore, seems to be acceptable the definition of intuition given by Herbert Simon in [7]: "*intuition is simply a form of recognition*". An intuition is not endorsed by rationales, it only comes to suggest a solution, based on our past experiences, without certainty for its correctness.

Summarizing, intuition is the ability to appreciate resemblance among the problem we are facing and problems we are faced, and reapply solutions we just saw in the past in similar, but non identical way.

Chapter 3

A psychological point of view

Intuition is poor studied from Artificial Intelligence communities. Therefore, to implement an intuitive network seems to be necessary to start from a psychological analysis of what intuition is. The analysis is based on the book *Thinking Fast and Slow* [8], written by the Nobel Prize Daniel Kahneman.

3.1 Thinking Fast and Slow

The book gives a behavioural model of the human mind. Kahneman agrees with the Dual Process Theory, according to which it is possible to divide the cognitive functionalities of the brain in two large sets, implemented by two different systems:

- System 1: fast, involuntary. It is responsible of intuitive understanding.
- System 2: slow. it is in charge to logically analyse thoughts and facts and to allocate attention, e.g. to solve complex computation.

The most interesting chapters of the book, by the point of view of our research, are summarised in the following.

3.1.1 System 1

The associative machine Every type of input the mind receives turn on an automatic mechanism of associations which links what we see, listen or touch to ideas, memories or others.

Associations are waked up quickly and uncontrollably. May be caused by a combination of many inputs and System 1 try to use them to make sense to the situation we are facing on, to prepare us to what come next. Every net activation provokes a big activity of which we aren't completely conscious.

Psychologists think our memory is like a network where objects are connected to their characteristics, causes to effects, facts to its habits [8].

Ease, Mood, and Intuition Sarnoff Mednick, around 1960, supposed the creativity is like an associative memory which works extremely well. He creates a test still used nowadays: the Remote Associate Test (RAT). The RAT consists in to find a word linked to other three words, e.g. given *cottage, Swiss* and *cake* guess *cheese.*

People mood, it has to be noticed, influences the test results. Good mood doubled the answer accuracy, instead, bad mood caused a very low correctness. Seems creativity and intuition work well during happy periods. Contrary, System 2, is easily kept off. So, logical errors become easier.

A machine for jumping to conclusions System 1 tries to guess responses and solutions when it faces problems. If the guess is easy and the prediction correct, the computation time is avoided, but sooner or later errors will come. The predictor learns from previous errors and gains experience. It bases its bets on recent events when available or on less recent contexts. In section 3.1.3 intuitive predictions are discussed more in depth.

3.1.2 System 2

Attention and effort System 2 operations need attention and effort, they are poorly executed when attention is missed or effort is low.

Attention and effort are allocated voluntary and have an upper limit. When System 2 has to deal with too many tasks, it gives priority to the most important, e.g. just think to be the passenger during a road trip, the driver is able to have a conversation with you but during a tough road, probably, it stops to talk so animately.

The lazy controller System 2 has to supervise actions hinted by System 1. Some actions may be executed exactly as come, others need to be modified or suppressed. System 1 is often leaved uncontrolled by System 2 and people answers become unchecked products of System 1. An example is the *bat and ball* puzzle proposed by [8]:

A bat and ball cost $1.10.
The bat costs one dollar more than the ball.
How much does the ball cost?

The first answer come to everyone mind is 10 cents. It arrives without reasoning or computations. It is an intuitive and wrong answer. In fact, if the price of the ball was 10 cents, the total would be 1.20$. The correct answer is 5 cents. Some people directly answer ten, others resist and start reasoning. In the latter, System 2 is prone to be activated, in the former is lazier. While System 1 is ever active, System 2 in general is not. For this reason, and for its role of controller, it takes the name of Lazy Controller.

3.1.3 Intuitive Prediction

Humans are able to forecast. Many examples exist, e.g. *"Economist forecast inflation"* [8], *"publishers and producers predict audiences"* [8]. Two main categories of predictions may be identified:

- Engineering Prediction: After several observations a mathematical model can be build and used to predict what will happen.
- Intuitive Prediction: After several observations the gained experience permits to predict what will happen.

We will focus on the latter one, in fact, this type of prediction come to mind without reasoning and are a type of intuition. Chess masters, doctors, and others, are able to predict intuitively the best play or the disease, respectively, simply taking a look to the board or to the patient, most of the times. It's not a trick, during years of practice, they gain a lot of experience and when face a new problem they are able to recognize the situation and automatically gets the answer thanks its System 1:

"The situation has provided a cue; this cue has given the expert access to information stored in memory, and the information provides the answer. Intuition is nothing more and nothing less than recognition" - H. Simon [7]

When the intuition is valid? It depends from who formulates the intuition, from its level of expertise and trustworthiness. But certainly, the environment in which the intuition is expressed has a main role. More an environment is predictable, more an intuition is reliable.

Summarize

Thinking, Fast and Slow [8] analyses human intuitions from different point of view, starting from the *Dual Process Theory*. We believe the associative memory and the intuitive prediction are the most promise types of intuition to be implemented.

In the following two chapters interesting materials from scientific literature will be explored, for both associative memory and intuitive prediction.

Chapter 4

Reinforcement Learning

Reinforcement learning is a technique to train neural networks. The net aims to maximise a score, but to improve its behaviour, labelled examples are not provided, it get better by analysing the system response to its actions. By observing reinforcement learning applications, like [11] and [12], it's clear how the networks learn by attempts, and gradually start to recognize just seen, or similar to just seen situations. So, it starts to select actions which get the highest scores.

We notice a similarity with *intuitive prediction* (3.1.3) in which the answer to a problem is automatically fetched from memory after a rapid look to the problem. For this reason Reinforcement Learning is analysed in this chapter.

4.1 Survey

Reinforcement Learning is a way to train agents by positive and negative feedbacks in response to agent behaviour. The feedbacks are named rewards or reinforces. Agents are usually implemented by NNs and its training don't need labelled examples, only rewards for its actions. The aim of the agent is to maximize the sum of received rewards along every episode. This survey section is based on [9].

Markov Decision Process The agent received the state of the environment s as input and produces an action a as output. The action changes the state and a scalar reward signal r is generated from the environment. The reward signal is used to train the agent. See image 4.1.

Formally the model is named Markov Decision Process (MDP) and consists of:

- A discrete set of states S;
- A discrete set of actions A;

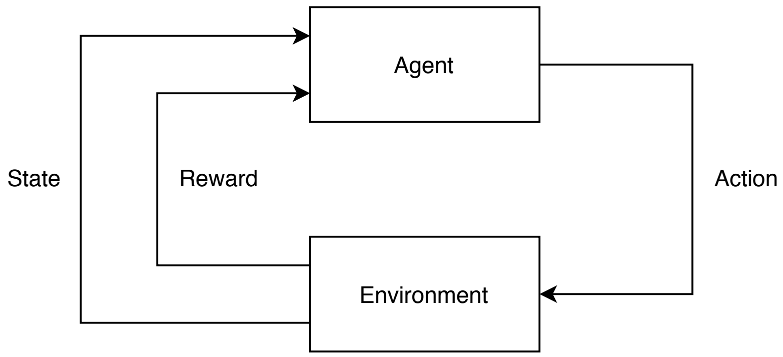

Figure 4.1: Reinforcement Learning Architecture. Agent receives as input the state of the environment and generates an output action. The action changes the state and a reward is received to improve the agent behaviour.

- A set of scalar reinforcement signals.

Agent aims to maximize the State-Value function V:

$$V(s) = E\Big(\sum_{t=0}^{\infty} \gamma^t \cdot r_t\Big) \tag{4.1}$$

Where $\gamma \in (0,1)$ is a discount factor, introduced to bound the infinite sum. Other interpretations are given in [9]. r_t is the reward signal received at time t. Therefore, the state-value function is the discounted sum of all the rewards received along an episode. Maximize V means correctly choose the best possible action in every visited state looking for a long term maximization instead of the short term best possible r_t.

The chosen actions are suggested from an agent, implemented by a function $\pi : S \rightarrow A$ named policy. Following the optimal policy, the optimal state-value function, i.e. the highest, is reached:

$$V^*(s) = \max_{\pi} V(s) = \max_{\pi} \left(E\Big(\sum_{t=0}^{\infty} \gamma^t \cdot r_t\Big) \right) \tag{4.2}$$

4.1.1 Find a Policy Given a Model

The model may be represented by the next two functions:

1. $T : S \times A \times S \rightarrow \Re$: the probability to arrive in a state s' starting from s executing action a;

2. $R : S \times A \rightarrow \Re$: the reward given by executing the action a in the state s.

Hence, the optimal state-value function 4.2, may be rewritten as following:

$$V^*(s) = \max_a E\Bigg(R(s,a) + \gamma \cdot \sum_{s' \in s} T(s,a,s') \cdot V^*(s')\Bigg) \tag{4.3}$$

Given images 4.2 and 4.3 it's easier to explain eq 4.2 and eq 4.3.

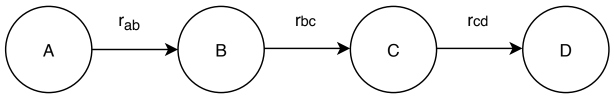

Figure 4.2: A simple graph representing environment states.

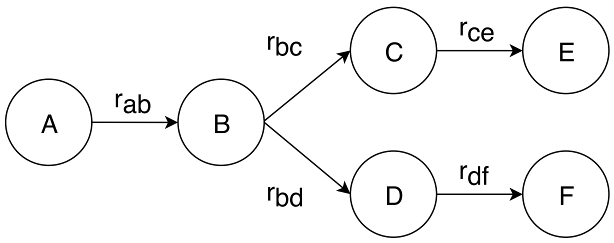

Figure 4.3: A forked graph representing environment states.

Every node of the graphs of images 4.2 and 4.3 are a state and every arch represent a transition. State to state transitions are rewarded. To estimate $V(A)$, in the case of graph 4.2, means to estimate the discounted sum of the rewards r_{ab}, r_{bc} and r_{cd}:

$$V(A) = r_{ab} + \gamma r_{bc} + \gamma^2 r_{cd} \tag{4.4}$$

In image 4.3 there are two different possible paths among states of the environment. Actions determine the next state with a certain probability T. Knowing the probability of the states transitions T and the immediate reward R, it is possible to find V applying eq 4.3:

$$V(B) = R(B,a) + \gamma\Bigg(T(B,a,C)V(C) + T(B,a,D)V(D)\Bigg) \tag{4.5}$$

Eq 4.2 and eq 4.3 may be used to find indirectly the optimal policy. For every state s, the state-value function is evaluated for all the possible actions. The argument of the higher state-value function is the same action the optimal policy would have chosen.

4.1.2 Model-Free Methods

Reinforcement Learning agents "*must learn behaviour through trial-and-error interactions with a dynamic environment*" [9]. In fact, when RL is applied, usually the mathematical model is not available and then, section 4.1.1 hypotheses are not helpful. In the following, temporal difference (TD) model-free method is briefly explained, then Q-networks are introduced.

Temporal Difference Methods: TD(0) and TD(λ) Without a model are just known the states, before and after action execution and the reinforce signal.

Temporal difference Method exploits relationships between state-value functions of neighbour states to improve state-value function estimation accuracy. Equation 4.6, referred to image 4.4, clarify the bound:

$$\begin{aligned}
\hat{V}(S) &= E\big[r + \gamma r'\big] \\
\hat{V}(S') &= E\big[r'\big] \\
\hat{V}(S) &= E\big[r + \gamma \hat{V}(s')\big]
\end{aligned} \tag{4.6}$$

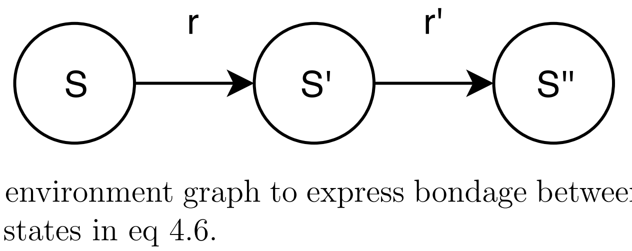

Figure 4.4: Simple environment graph to express bondage between state-value functions of neighbour states in eq 4.6.

Starting from an arbitrary distribution of the state-value function, it's possible to use the received reinforce signal of a state transition to improve the accuracy of

V in the following way:

$$\begin{aligned} V(s) &= V(s) + \alpha(r + \gamma V(s') - V(s)) \\ &= (1-\alpha)V(s) + \alpha(r + \gamma V(s')) \end{aligned} \tag{4.7}$$

Equation 4.7 is simply a weight average in α between the direct and indirect estimations of $V(s)$. The indirect evaluation involves the guess of V(s') and r. r is a correct value given from the environment, it's not a guess. Therefore, thanks to r the accuracy of V slowly increases. Algorithm 4.7 implements the TD(0) method. The more general class of TD(λ) algorithm, introduce a lambda factor which modulate the state value functions update frequency. The eligibility is a measure which determines the state significance in terms of number of visitations, giving higher weights to the more recent:

$$e(s) = \sum_{k=1}^{t} (\lambda\gamma)^{t-k} \delta_{s,s_k} \tag{4.8}$$

Where δ is one iff $s = s_k$.

Q-Networks and Deep Q-networks Q-networks are actually the most common used form of RL algorithms. They are based on the computation of the Q-value, close relative of the state-value function:

$$Q^* : S \times A \to \Re.$$

Q is the expected discounted sum of the rewards starting from state s, choosing action a and then acting following the optimal policy π^*. But by initially selecting the best possible action a, the optimal policy is followed just from the beginning, so the Q-value becomes equal to the optimal state-value function:

$$V^*(s) = \max_a Q^*(s,a) \tag{4.9}$$

A typical Q-networks training strategy may be the temporal difference method, seen in section 4.1.2:

$$\begin{aligned} Q(s,a) &= Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a)) \\ &= (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a')) \end{aligned} \tag{4.10}$$

RL agents may be used as standard controllers. They generate a control action starting from the system state. Nowadays, thanks to deep neural networks, the class of the possible states belong to a wider range. It's in fact possible to use images, audio, video and text in addition to common state vectors. This new form of Q-networks are named deep Q-networks.

4.2 DeepMind

DeepMind is one of the leaders in the field of artificial intelligence in the world [10]. It was born in 2010 and it has been acquired by Google in 2014. In the following sections two works of DeepMind are reported as examples of reinforcement learning agents implementation.

4.2.1 Human-level control through deep reinforcement learning

A demonstration of deep-RL effectiveness is given in [11]. An agent becomes capable to play 49 different games of the console Atari 2600 with impressive results. By receiving the raw video frames and the game score it outperforms the results of all existing RL agents in all the 49 games, reaching at least the 75% of professional play testers scores in 29 games. A video demonstration of the agent ability is available at this link.

Strategy and Architecture The raw video frames are took 4 by 4, they are compressed in terms of dimensionality (from 210x160 pixels to 84x84 pixels) and colours (from a 128-colour palette to the only luminance) and used as state s of the network.

The agent selects the action to perform starting from the 4x84x84 data packet, the Atari state. Multiple action selection strategies are possible. The one implemented in [11] is explained in the following. The preprocessed image feeds three convolutional and two fully connected layers:

1. Convolutional layer (CL) with 32 8x8 filters with stride 4 and a rectifier of the type max(0,x).
2. CL with 64 4x4 filters with stride 2 and a rectifier of the type max(0,x).
3. CL with 64 3x3 filters with stride 1 and a rectifier of the type max(0,x).
4. Fully connected linear layer with 512 rectifier units.
5. The output layer is another fully connected linear layer which computes the Q-value for all the 18 possible actions.

Therefore, at every time step, eighteen Q-values are computed and the action of the higher one is chosen.

Training During episodes, at every state transition, the agent action is stored together with state, next state and the reinforce signal. This type of records are named tuples: $\langle s_t, a_t, r_t, s_{t+1} \rangle$. Tuples are stored inside Replay Memory (RM). Between games, random tuples are sampled from RM and used to train the network with a TD-like method, analysed in section 4.1.2.

The policy is determined indirectly, the eighteen Q-values are evaluated and the action which lead to the highest one is selected. The agent choice is followed with a probability $1 - \epsilon$. To ensure state-space exploration a random action is taken with a probability equal to ϵ. ϵ starts from a value of 0.9 and linearly decreases down to 0.1.

Discussion DeepMind work [11] demonstrates the effectiveness of reinforcement learning agents and their flexibility to different environments. It proves how a model-free controller may learn how to behaves simply receiving a feedback from the environment for its actions, like humans do.

4.2.2 Deep Reinforcement Learning in Large Discrete Action Spaces

In the previous case at issue (4.2.1), the action space is small and a Q-value based approach, where the Q-value is evaluated for every possible action, it's possible. The complexity scale linearly with actions, so, the approach shown, become impractical where the actions are one thousand, or one million, or even more. The work [12] proves and shows how to deal with larger action spaces. Three cases are taken under examination and it is shown the agent needs a different implementation to work with respect to [11].

Strategy The policy π is directly implemented, instead to be extracted indirectly selecting the higher Q-value. Hence, a neural network implementing π is used. The NN produces actions in a continuous space. It's easy for the continuous actions to be out of the discrete action space. Therefore, it's necessary to select a discrete action from the discrete space, starting from the continuous action given by the NN implementing the policy π. Nearest Neighbour algorithms are used to solve the problem. If only the nearest discrete action is taken, the algorithm may not converge: the dynamics of the system under control with respect to input actions may be very varying. So, the best approach consists in to select a certain number k of actions, and then to start a Q-value based approach with the reduced action space with dimensionality k.

Experiments and Results In the following three cases of study with different actions spaces and dynamics are reported. The main agent parameters are the number k of actions selected from the Nearest-Neighbour algorithm and the type of Nearest-Neighbour algorithm used: exact or approximated (slow-99% accuracy, medium-90% accuracy, fast-70% accuracy).

1. **Cart-Pole** The agent has to balance a pole attached to a cart by applying force to the cart. The action, therefore, is the force intensity and there are 1 million possibilities. To verify the performance of a Q-values based approach it's possible to set k equal to the action space dimension, so, 1 million. With this setting the algorithm doesn't converge. The pole responses to similar

actions, it's only slightly different. So, it's possible to set k at 1 and skip the Q evaluations. For the same reason a fast-70% accuracy nearest neighbour algorithm demonstrates to be sufficient. A video demonstration of the agent behaviour is available here.

2. **Multi-Step Plan Environment** The agent has to plan a path to arrive to a goal position. The number of possible movements i: up, down, right and left or only down and left, is a parameter of the algorithm. Moreover, the number of actions to plan n is an extra parameter. The Q-value based approach should tract with a complexity of i^n, and quickly diverges. With n equal to 1, the algorithm still diverges. It's necessary to increase n up to 20 to met the optimal policy. k equal to 1 is sufficient but with a slow-70% or exact nearest neighbour algorithm.

3. **Recommender Environment** The agent has to recommend items to the user on the basis of the object currently in use. Three item sets of different size are taken into account. For every set it has proved to be necessary increases k with respect to the Cart-Pole and Multi-step Plan Environment cases ($k = 1$). The Recommender Environment dynamics is quite irregular. The agent converges for k equal to 20, 83 and 656, for action spaces of 49, 835 and 13138 items respectively.

Discussion It has been shown by [12] reinforcement learning agent efficacy in large action spaces. [12] gives the green to a new class of problems for RL agents.

Summarize

Reinforcement Learning seems to be particularly interesting to emulate human-like learning and behaviour when unknown problems are faced. In fact, the aptitude to learn without examples it's a fundamental human ability.

Chapter 5

Associative Memory

According to the *Associative Machine* theory (3.1.1): knowledge, ideas, words, memories, dreams are stored in a network in which they are heavily interconnected: objects to their characteristics, causes to effects, facts to its habits [8]. Inside this net, some nodes may be activated from the outside from images, sounds, perceptions and in turn they activate other nodes, to whom they are connected, and so on. This structure explains why we remember something we had forgot thanks to a small cue. It is moreover responsible of the majority of actions we do, especially when lost in thought (and system is 2 shut off or completely busy). The chapter starts analysing the associative memory implementation [13], then it focuses on Latent Semantic Analysis, a way to represent words and phrases which gives similar representations to words with similar meanings. Finally the *Word2Vec* project is described. It is a step forward with respect to Latent Semantic Analysis. In fact, thanks to it, it become possible to express analogies among words.

5.1 A Spiking Neuron Associative Memory Example

A spiking neural network associative memory, powered by the Neural Engineering Framework (NEF), is introduced by [13]. The system was developed to have an estimation of the internal biological mechanisms who regulate associations in the human brain. Given that, spiking neurons are used and also, some constraints are introduced in the synapses and post-synapses current models to respect biological structures and behaviours. The system is tested on the Remote Associates Test (RAT), a practice still used by psychologists to measure people creativity. Subjects under examination have to identify the word which connects three other words, given by the test, under a certain time limit. To meet the goal an associative memory is essential, to connect words among them and consequently to generate the puzzle answer. The words associations database, provided by [14], is used from [13] to generate words associations.

5.1.1 Associative Memory Architecture

Inside the Neural Engineering Frameworks words are represented by vectors, input stimuli are sensed by the network and they activate groups of neurons which represent data entries. The NEF, furthermore, encodes associations among words as synapses between groups of neurons. Therefore, neurons trigger in turn other neurons areas generating associations.

Word Representation Word-vector pairs are generated off-line and mapped into the network. Differently from Latent Semantic Analysis (LSA) [18] or Word2Vec [20], analysed later in the chapter, encodings are not based on semantic similarities among words because the highly noisy spiking networks may easily confuse inputs among each others. For this reason words are encoded as almost orthogonal vectors. To achieve this purpose the vectors dimension rapidly raise; to correctly solve the RAT, vectors of 2048 dimensions are used to represent 5018 words.

Imagine a surface of neurons, where each neuron is connected to the same inputs. If the input vector is normalized, and the weight vectors are normalized, the activation will be higher where the alignment between input and weights is greater. In fact, neuron activation is proportional to the product:

$$w_i^t \cdot x(t) \tag{5.1}$$

Where w_i is the weight vector of the i^{th} neuron and $x(t)$ the input vector at time t.

Starting from this simple principle the NEF is able to represent each word as a neural activation.

Making Associations Neural response to input stimuli has to be some way elaborated to trigger associated neurons. Then, neuronal activities has to be decoded.

Skipping the decoding phase, because closely related to the type of neurons used by [13], it's possible to express associations starting from the following definitions:

- The Dictionary D, an NxD matrix. Where N is the number of words and D the vectors dimensionality.
- The Associations Matrix A, an NxN matrix of normalized real numbers.

Thanks to the NEF it's possible to implement the output activity $y(t) = \hat{A}x(t)$ where:

$$\hat{A} = D^t AD \tag{5.2}$$

Hence, the output activity y(t) may be rewritten as:

$$y(t) = (D^t AD)x(t) \tag{5.3}$$

By analysing one by one the partial products, some truths may be inferred:

- $Dx(t)$ return a N dimensional vector where, the i-th element gives the alignment between the word $x(t)$ and the word $D(i)$ of the dictionary.
- $A(Dx(t))$ generates a vector of real numbers of dimension N, where both alignment with the input and associations strength among words are taken into account.
- $D^t(ADx(t))$ remaps back the informations in a vector of dimension D, like the dictionary. The i-th element is the activation of the word $D(i)$.

Summarize

The implementation [13] is an example of associative memory selected among many others present in literature ([16], [17]). They prove the possibility to implement an associative memory: a system which stored data and link among data. The behaviour implemented from this networks seem to be the basic kernel to arrive to create an artificial intuition system according to the theories of Kahneman and Simon.

5.2 Latent Semantic Analysis

Words may be represented as vectors of a multidimensional space, as shown in [13] and explained in sec 5.1. Latent Semantic Analysis (LSA) studies semantic similarities among words and suggests similar encodings for words with similar meanings. The advantages and the applications of this technique range from the language processing ([19]) to the database query ([18]) and in this section they will be investigated. The first step consists into create a matrix of associations among the objects of interest, e.g. words. The second step instead, it's the Singular Value Decomposition (SVD).

5.2.1 Association Matrix

An association matrix is a collection of association strengths between object pairs. Starting from a dictionary of N words, it's possible to define an Association Matrix A, of dimension $N \times N$, where each element a_{ij} represents the strength of the association between the word i and the word j [19].

We use a free available associations database of 5019 words [15]. Informations are collected by asking to more than 6,000 participants to write the first word come to his mind by reading a cue word, e.g. the volunteer read *"dog"* and write *"cat"*. But, if the word *"dog"* often produces the word *"cat"*, the opposite is not necessary true. So, the matrix isn't symmetric and reasonably full of quasi-zero numbers. It is similar in shape to what reported in table 5.1.

	human	interface	computer	user
human	1	0.8	0	0.6
interface	0.8	1	0.9	0.9
computer	0.2	0.6	1	0.7
user	0.7	0.9	0.8	1

Table 5.1: Example of Word Associations Matrix

Diagonal values may be set to one or to zero, it depends by the application. The main assumption in LSA is that similar words are supposed to have similar representative rows. The study [19] shows association matrix manipulations starting from the data collected in the free associations database [15]. Starting from a matrix like 5.1 it is possible to get other association matrices:

- S' matrix: its elements are obtained by adding forward and backward association strength for each and every element:

$$s'_{ij} = a_{ij} + a_{ji} \tag{5.4}$$

  Therefore, S' become symmetric.

- S'' matrix: for every connection are taken into account backward and forward strength, like for S', but also indirect connections:

$$s''_{ij} = s'_{ij} + \sum_{k=1}^{N} s'_{ik} s'_{kj} \tag{5.5}$$

5.2.2 Singular Value Decomposition

The core of the LSA is the Singular Value Decomposition (SVD). The algorithm starts from an arbitrary matrix of associations A, of dimension $N \times N$, where N is the length of the dictionary, arrives to the Word Association Space (WAS) [19]. The WAS is a k-dimension space where words are represented according to its similarity. An example of Singular Value Decomposition is available in [18]. With the purpose to construct queries which go beyond common strings comparisons, an association matrix is constructed to bound documents stored in the database with all the words contained inside the texts. A matrix, quite similar to that shown in tab 5.1, is constructed. However, words are arranged along the rows and documents along the columns. Every item of the matrix is a count of how many times such word appears inside each document, a frequency measure, then the items are normalized. Hence, words which appeared in common documents have similar rows and documents which contain similar words have similar columns. Singular Value Decomposition reduces the matrix dimensionality from N to k. To accomplish this dimension lowering the A matrix is decomposed into the product of three matrices:

$$A = T_0 S_0 D'_0 \tag{5.6}$$

Such that T_0 and D_0 have orthonormal columns and S_0 is diagonal. Then, all the singular values of S_0 are ordered by size and only the k highest are kept. If all the other values are set to zero and the multiplication executed a new matrix $\hat{A}$ is obtained. It can be demonstrated that $\hat{A}$ is the least square approximation of A and it has rank k. By removing rows and columns of the set to zero elements in S_0 and the corresponding column in T_0 and D_0 three new simplified matrix are obtained: T, D and S. By multiplying T, D and S is still obtained $\hat{A}$ [18]. Rows of $\hat{A}$ may be used as vectorial representation of the words and its columns as vectorial representation of the documents. Although, $\hat{A}$ is obtained by an approximation of A it still maintains the same dimensionality as illustrated by image 5.1.

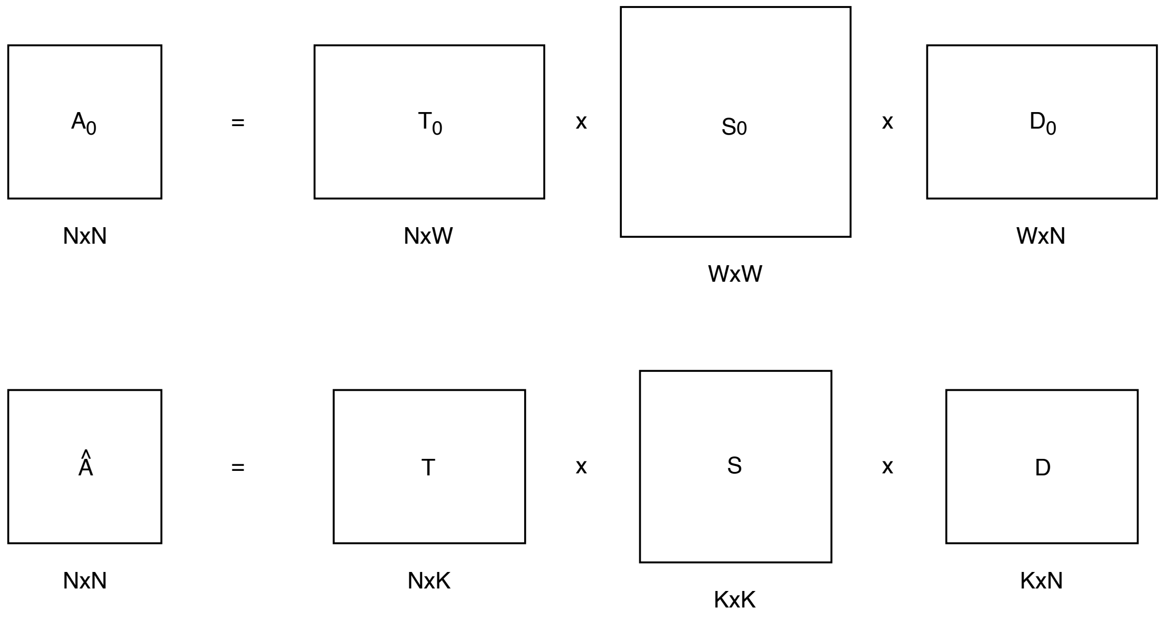

Figure 5.1: Graphical illustration to clarify the equal dimension of A and $\hat{A}$

So, new queries may be processed differently. Each word is substituted by its vectorial representation and an average vector obtained by the query's entries is used to match results inside the database. The query is compared with the documents by a scalar product. Indeed, scalar products among normalized vectors return the cosine of the angle between the vectors which states how much the two elements are superimposed. A query example is reported in [18]. It shows how the query *"human computer interaction"* without SVD doesn't return two important matches which are instead returned thanks to SVD query improvements. The same SVD technique may be applied to any association matrices, like for example S' and S''. [19] reports how semantic similarities between words may be enhanced thanks to SVD and simple scalar products.

Summarize

Latent Semantic Analysis permits to assign meaningful vectorial representations to words but, as seen in [18], also to more complex entities. *Is it possible to represent thoughts, concepts and memories inside the word association space and then build an associative memory using the resulting vectors as nodes?*

5.3 Word2Vec

Latent Semantic Analysis and Singular Value Decomposition are not the only way to find meaningful vectorial representations for words. Word2Vec (W2V) [20] project aims to reach the same goal but with a totally different approach. Several types of neural networks are trained to guess next words given a phrase context. When training ends, the encodings built inside the inner layers of the NNs are taken and used as vectorial representation of the words. W2V is not only another way to achieve the same targets, its encodings are near for words of similar meanings but, furthermore, distances among encodings have as well importance. Semantic and syntactic properties are extracted, e.g. the difference between vectorial representations of *king* and *man* summed to the vector *woman* return something very close to *queen* in the semantic space [21].

5.3.1 Training and Architecture

Different classes of NNs are trained to extract vectorial representations of words, also named embeddings, with slightly different characteristics.

Feed-Forward Model Let's start talking about the Feed-Forward Model proposed in [23]. The following points shed light on its architecture:

- Input layer: T words are encoded using one of the N possible coding (N is the size of the vocabulary);
- Projection layer: the T words input produces an output of $T \times D$ (D is the word-vector dimension);
- Hidden layer: Compute probability distribution over all the N words in the vocabulary (N is the output dimension);
- Output layer: the output word is selected.

Recurrent Neural Network Model The Recurrent Neural Network (RNN), presented in [24], it is made up by:

- Input layer: a single word is encoded.
- Inner layer: thanks to the recursive connections the network earn a memory and the input words received previously play a role in the output evaluation, jointly to the actual input word.

  The probability of each word to be the next with respect to the context is given.
- Output layer: the output word is selected.

The recursive network general equation are reported in the following:

$$\begin{aligned} s(t) &= f\big(Uw(t) + Ws(t-1)\big) \\ y(t) &= g\big(Vs(t)\big) \end{aligned} \tag{5.7}$$

Where:

- s: state; y: output; w: input;
- U,W,V: weights matrices.
- $f(z) = \frac{1}{1+e^{-z}}$ and $g(z_m) = \frac{e^{z_m}}{\sum_k e^{-z_k}}$: Sigmoid and Softmax non-linear functions respectively.

Thanks to recursive connections, fixed length context of Feed-Forward Model is overcome.

Continuous Bag-of-Words Model The Continuous Bag-of-Words (CBW), presented in [21], it's similar to the Feed-Forward model. Even though, the projection layer is missing. The architecture:

- Input layer: it receives T words as input;
- Hidden layer: the T words are averaged and the probability of the next word computed for each and every words of the dictionary.
- Output layer: It selects the output word.

The name comes from the fact that the averaging delete words order. It's like they are putted inside a bag.

Continuous Skip-gram Model Continuous Skip-gram Model (CSG) operates differently from other models. It receives only a word as input and guesses previous and future words, it guesses the k words context of the input word. Its architecture:

- Input layer: A single word of dimension D is received;
- Projection layer: It predicts the probability of each word of the dictionary, for every possible position in the context.
- Output layer: It selects the k words of the context.

5.3.2 From Words to Phrases

The main limitation of previous presented works is the impossibility to represent something more complex than single word. Not always the sum of the single words meanings correctly represent a more complicated concept. This limitation brought to new studies, ideas and results, like [22]. Here, to solve the problem, the texts used for the NN training are pre-analysed and the most common phrases are treated as *"individual tokens"* [22]. In this way the dictionary size increases, but more complex concepts may be correctly represented. Furthermore, the [22] study, presents a new characteristic of W2V encodings. The normalized sum of two vectors permits to obtain interesting results, for instance the sum between the vector *river* and *Russia* returns the vector *Volga river*.

5.3.3 LSA and W2V comparison

Latent Semantic Analysis and Word2Vec projects are quite similar and it's difficult to decide at priori who to bet on. [25] compares the encodings under several expects:

1. Semantic relatedness: the cosine distance among words is compared with an average votes on relatedness expressed from people.
2. Synonym detection: the synonym has to be detected among four words. The answer is chose by means of scalar product.
3. Concept categorization: space items have to be clustered in natural categories.
4. Selectional preferences: the most common noun as subject or object of the action described by the given verb have to be supplied. Answers are compared with people's answers.
5. Analogy: an example pair and a test word are assigned, e.g. *brother*, *sister* as example pair and *granddaughter* as test word, the word *grandson* is expected.

W2V approach outperforms LSA in almost all the tasks. Only in the Selectional preferences task LSA approach W2V achievement.

Summarize

Techniques described in this chapter show as words may be represented in a vector space in which relatedness, semantics and syntactic properties may be embedded. Especially [22], which introduces phrase encodings, transforms from a word to a concept space, it seems to be a good starting point to build an intuitive architecture.

Chapter 6

Visual Embedding Space

Figured out a way to implement intuitive reasoning, it's necessary to equip the system with human senses to initially excite the associative machine. Common visual classifiers ([28], [29]) are trained to recognize a certain number of different categories of objects. Real word is not so rigid partitioned and it's common to find items for which the classification is ambiguous. A way to address the problem is to represent classifier categories as point in a semantic space ([19], [22], [20]) and classify input objects as points in the same space. Thanks to the properties of semantic spaces it's possible to correctly classify categories of objects never seen during the training. This feature is called Zero-Shot Learning. Two examples of Visual Semantic Embeddings (SE) classifiers are reported in this chapter [26], [27]. This systems cover a fundamental gap arisen during the conceptual developing of the intuitive network, i.e. how to shift from a visual to a semantic space, inside which associations are possible?

6.1 DeViSE

The Deep Visual Semantic Embedding (DeViSE) is a visual classifier which takes advantage of semantic analysis of textual data. It uses the embeddings to give a semantic representation of its categories and attributes a new embedding to each visual input it receives.

Architecture DeViSE is obtained by two distinct systems, i.e. the Mikolov's neural language model [21] and a deep convolutional neural network AlexNet able to classify up to 1000 classes of objects [28]. The two networks are distinctly pre-trained. Then, the last softmax layer of the visual classifier is removed and the resulting 4096 floating point vector is attached to a new block, which apply a linear shift from 4096 to 1000, the output dimensionality of the embeddings. The projection layer has to be trained to reduce mapping errors. The loss functions introduced in [26] aims to maximise the difference among the scalar product between the image and its label and the image and all the other labels:

$$loss(image,label) = \sum_{j \neq label} \max[0; margin - t_{label} M v(image) + t_j M v(image)] \quad (6.1)$$

Where:

- t_{label} is the embedding of the classifier label corresponding to the *image* and t_j the embedding of a generic label. All the embeddings are unit normed;
- $v(.)$ is the classifier output, removed the softmax layer.
- M is the projection layer matrix, it contains the parameters that have to be tuned to reduce the mapping error.
- The margin is set to 0.1 and avoid the *loss* is equal to zero for $t_j M v(image)$ products equals to $t_{label} M v(image)$.

As mentioned previously, classifiers which embeds semantic labels may correctly classify never seen categories of objects: Zero-Shot Learning. Indeed, for example, DeViSE is able to give an optimal semantic response to images of sharks, even thought it was never trained to recognize images labelled as *shark*, but only more specific categories like *tiger shark*, *bull shark*, and *blue shark* [26].

6.2 ConSE

The CONvex combination of Semantic Embeddings (ConSE), to be precise is not a Visual Semantic Embeddings classifier but a method to implement them. In [27] is explained how to create SE classifiers starting from whatever visual classifier and embedding vectors. The only necessary condition is that the classifier labels have to be contained among the embedding space.

ConSE method Consider an N-way visual classifier which returns a probability p_0, for an image x, to be a member of a class $y \in Y$, and $\sum_{y \in Y} p_0(x|y) = 1$. The t highest $p_0(x|y)$ are expressed as $p_0(x|t)$ with $t \in T < Y$ and are considered to map the images into the embedding space:

$$s(x) = \frac{1}{Z} \sum_{t=1}^{T} p_0(x|t) \cdot s(t) \tag{6.2}$$

Where:

- $s(\cdot)$ returns the embedding of its argument;
- $Z = \sum_{t=1}^{T} p_0(x,t)$ is a normalization factor.

Therefore, 6.2 is simply a convex combination of the T labels of the classifier with highest probabilities. The coefficients are the same probabilities.

ConSE implementation and comparison ConSE method is applied to the AlexNet [28] classifier and [21] embeddings to be directly comparable with DeViSE [26] on the ImageNet dataset [30]. Comparison among the two networks shown how DeViSE successes in the classification of the trained classes but ConSE performs better in the Zero-Shot task, more details are available in [27].

Chapter 7

Algorithm Overview

The algorithm working principles heavily weights on the ideas of Simon and Kahneman [7], [8]. The associative structure which permits to surf among past and conscious experiences in an unconscious and uncontrollable flow precisely hit our idea of what intuition is. We discover a way to represent human knowledge in the vectorial representation of words. It maps thousands of words and phrases in significant points of the space [22]. Thanks to this basic alphabet becomes possible to represent new concepts as convex combination of the others.

The free association database [15] permits to set up a reasonable associative structure, since associations to 5019 words are collected with the support of 6000 participants. The database is free available and associations are given in different formats. We chose for Appendix B which order alphabetically the words and gives all the associated words for each and every entry. The database of Appendix B is used after a minimal reformatting made by a Python script.

Once the associative skeleton is set up, it becomes necessary to surf on its paths. With infinite resources, starting from a cue word would be possible to take all the associations as next states, and in turn recurs on all of them. In this way resources soon terminate, it's necessary to define both a width and a depth for the research. The width determines how many words are evaluated at every step, the depth instead, establish after how many steps the algorithm is stopped. They define the research space. Set a limit on the width lead to select the best associations to kept and the worst associations to thrash. We decide to insert an agent, Q-network based, which receives words pairs in input and returns a measure of the transition quality, the Q-value. The agent filters the new states.

Algorithm outputs are also based on the agent, in fact, when the Q-value between the starting word and one of the arrival states of the algorithm is sufficiently high, the state becomes an output. The algorithm, which implements the System 1 of the *Dual Process Theory* [8] suggests words, phrases, ideas to the listener, the logic System 2 in the theory. The suggestions are called intuition when System 2 believes they solve the problem faced.

In the following sections the algorithm is explained in-depth, meanwhile the Matlab code is available in Appendix A. The first section 7.1 clarifies Q-agent implementation, then the associative machine algorithm is described in section 7.2. Last, a way to map images on word embeddings is investigated in section 7.3.

7.1 Q-network Implementation with Matlab

The Q-network agent is fundamental to filter among associated words and to select which associations suggest to a hypothetical System 2 of the *Dual Process Theory* [8]. As explained in section 4.1.2 the agent receives in input the system state and the action and returns an estimation of the final score given by executing that specific action and then the optimal policy. The state of the system, initially is the cue word, after the first iteration w words among the associated words are kept. The w words are the new state of the system.

Thinking about the DeepMind examples, shown in section 4.2, it's possible to analyse the architecture of the agent with respect to known and working schemes:

1. The states and actions sets are superimposed, just like in the Recommender Environment example, section 4.2.2, then the chosen actions coincide with the system next state.

2. Differently from the Atari example, reported in section 4.2.1, where the possible actions are only eighteen and the Q-agent directly returns in parallel the eighteen Q-values given by following action a_i with $i \in [1 : 18]$, in our case the actions are many more, so the agent takes as input the couple state-action, just like in the reinforcement learning theory in section 4.1.2, and evaluates transition quality one by one. It has to be noted that the evaluations may be performed in parallel.

3. The case of study is close to the large discrete action space example of section 4.2.2, but it is simplified by the restricted number of actions available in every state. For this reason it's not necessary to implement a policy π and then a near neighbour algorithm to arrive at the Q-value estimation. It makes sense to limit the agent evaluations to the words associated to the current state or to its subset.

String words are substituted by their embeddings. So, the agent input is a couple of embeddings of which it estimates the transition quality. Therefore, if a state transition receives good or bad rewards, similar state transitions are trained automatically.

Matlab Implementation

Network Structure Deepmind Atari research [11] shown its Q-agent structure, as recap in section 4.2.1. Skipping the first convolutional layers which have the role to extract the features of the console raw frames, to implement the Q-values computation are left two fully connected layers. So, thanks to the Matlab Deep Learning Tool Box a feed-forward neural network is implemented by means of the *fitnet* Class. The Q-network internal structure is made up by 2 fully connected layers, exactly as the Atari-net [11]. Input size changes according to the embedding size, the output width is constantly kept to one. The number of hidden units per layer is a very critical parameter which has to be tuned very carefully (see Methods ch.8).

Reply Memory Like in [11] a Reply Memory is used to keep track of the system transitions and to train the net among episodes. In this case starting and arriving states are saved as strings and only when the training time comes they are converted to embeddings with the *word2vec* method. In this way a big amount of memory is saved but, most of all, the Reply Memory access time is reduced moving all the strings to embeddings conversions in a not real time environment, among episodes, when training is performed. Moreover, the Q-values estimated by the net are stored together with the states. Summarising, each entry of the Reply Memory is made up by tuples with the following structure:

$$\langle \text{ current state, next state, Q(current state, next state) } \rangle$$

Training Phase At training time, temporal difference method TD(0), readapted to Q-networks and described by equation 4.10 is used to update the agent. Looking at 4.10, some informations are missing to apply temporal difference method:

1. The discount factor γ is a parameter tuned very close to one according to the Deepmind directives [10], [12];
2. The learning rate α, to be tuned during algorithm testing phase;
3. The reinforce signal r possible values are two: $+1$ for positive rewards and -1 for negative rewards. In this way the feedback is straightforward: it becomes

a simple thumb signal and after the training phase the network outputs stays very near the range [-1,+1] since all the training set entries are bound in the range.

Rewards are not stored inside Reply Memory, in fact they are not even known during an episode, they are discovered only when some suggestions is proposed as output. Or also, as it is explained in section 7.2, when some of the suggested words proposed from the outside have be found during special training episodes. Long story short, the training function reads the Reply Memory and receives an array of thumb up strings. It looks for this words and follows all their paths updating their Q-values positively, meanwhile all the other transitions are negative rewarded. Anyway, also the negative rewarded paths have to be followed to apply TD(0) method and to update the Q-values.

If for instance the cue word, i.e. the starting word, is *hat* and the width and the depth are set respectively to two and three, a possible system evolution may be given by image 7.1.

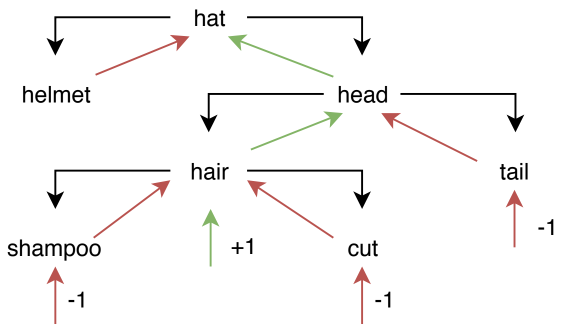

Figure 7.1: Graph of transitions and rewards starting from the cue word *hat* with width equal to two and depth equal to three.

Such graph leads to the Reply Memory state given by table 7.1. Where *cs* and *ns* stay for current state and next state.

If we ask for a single suggestion, the algorithm output is *hair*, since it has the higher Q-value in the column *Q(hat,ns)*. Filled the Reply Memory, it is possible to apply the TD(0) method, implemented by equation 4.10. In order to correctly update the net it's necessary to start from the bottom. The first step consists in to check if the last *ns* is a positive rewarded word. If it is, its *Q(cs,ns)* is set to

cs	ns	Q(cs,ns)	Q(*hat*,ns)
hat	helmet	0.43	0.92
hat	head	0.71	0.90
head	hair	0.59	0.96
head	tail	0.84	0.31
hair	shampoo	0.40	0.20
hair	cut	0.57	0.31

Table 7.1: Reply Memory status given graph 7.1

1, otherwise to -1. Then, the associated *cs* has to be found in the *ns* column to reconstruct the path and to apply equation 4.10. So, Q(*hair,cut*) is updated to -1, then the algorithm looks for *hair* in the *ns* column. It founds *head-hair* transition, positive rewarded, so the negative updates are stopped because all the previous transitions have led to arrive to a thumb-up state: *hair*. The algorithm restarts from the last but one transition: *hair-shampoo*. Negative rewarded as well. It sets Q(*hair,shampoo*) to -1 since it is at the end of the graph. Then the algorithm starts looking for *hair* in the *ns* column. It founds *head-hair* transition, positive rewarded. As previously, the negative updates are stopped. The algorithm restarts from transition *head-tail* and negatively updates Q(*hat,head*). Then restart from *head-hair* which positively updates Q(*hat,head*) deleting all the previous negative updates. Last, *hat-head* transition updates Q(*hat,helmet*).

Every fork in a path produces a collision, i.e. in the example both *head-hair* and *head-tail* transitions have to update Q(*hat,head*), which is the collision policy? Among thumb-up and thumb-down rewards, as seen in the example, thumb-up wins, otherwise a good suggestion is negative rewarded. Instead, who wins between two negative updates? Is q$_i$ updated by q_j or by q_l? Theoretically the answer is from both, and given that both the path have proven useless, q$_i$ is doubled decreased by the effects of q_j and q_l as in equation 7.1.

$$q_i = (1-\alpha)q_i + \alpha(r_j + r_k + \gamma q_j + \gamma q_k) \tag{7.1}$$

We will see in chapter 8 that the neural network doesn't fit well such strategy, so countermeasures have to be adopted.

7.2 Associative Machine Implementation with Matlab

The Associative Machine Class retains control of the entire system. It supplies Q-network and Reply Memory inputs and read the net responses to the input stimulus. It receives a cue word from outside to run the simulation and reads the free association database [15] to pick the associated states to submit to the Q-network as next possible states. It implements a set of three filters:

1. Among all the associated words, only that present inside the embedding dictionary are kept. In fact, if not present it is impossible to evaluate the Q-values in which the words are involved.
2. To avoid to lose algorithm cycles bouncing among two bonded states, a list of the just visited states is saved and the associations list is lighten by them.
3. Finally, after the Q-values computation, only w next state are kept. The w with the highest Q-value. w stays for the width of the research.

The class implements two different kinds of episodes, normally the simulation starts with a cue word and it lasts for a number of iterations given by the depth of the research and with a parallelism given by the width of the research, as explained in sec 7.1. In this case, the highest Q-values among the cue word and the visited states become the System 1 suggestions. However, another kind of episode is available: the training episode. In such episodes, it's added a training phase in which good System 1 outputs are directly trained thanks to a list of final words received in inputs. More details are available in Methods, chapter 8.

Association Database reformatting with Python

The type of associations of our interest are given by [15] in the Appendix B. A list of all the words is given in alphabetical order in eight files:

- A-B.html
- C.html

- D-F.html
- G-K.html
- L-O.html
- P-R.html
- S.html
- T-Z.html

Inside the files, for each and every entry a table with all its associated words is given, like in table 7.2.

A	FSG BSG
GRADE	0.270.0590
VITAMIN	0.060.0000
ALPHABET	0.040.0660
TYPE	0.030.0000
ACHIEVEMENT	0.020.0000
OUTSTANDING	0.010.0000
LOT	0.010.0000

Table 7.2: Table example from [15] Appendix B. Entry: A, on the left column the list of associated words, on the right forward and backward association strength: 0.Forward.Backward → 0.Forward, 0.Backward.

Where the first word is A, the owner of the table, while the left column host the list of associations. In the right column the encoded values of the backward and forward associations strength respectively, formatted as follow:

$$0.\text{Forward}.\text{Backward} \rightarrow 0.\text{Forward},\ 0.\text{Backward}.$$

A Python script, available in Appendix A, has been developed to convert the eight files in a set of files, one for each entry, with the list of the associations in the first row and a table with forward and backward associations strength starting from the second row. In this way is straightforward for the Associative Machine class to extract the needed informations thanks to the *fgetl* and *readcsv* Matlab embedded methods.

Matlab Implementation

The Matlab implementation of the Associative Machine class is quite simple and is available in the appendix A. It consists mainly in a for loop, whose iterations are dependent. So, no optimization such like unrolling or Matlab *parfor* are possible to speed up the simulation. However a certain parallelism, of which the *width* of the research is an indicator, exists inside every iteration and it is then possible to increase performances of the loop body. The gains will be multiplied by the number of iterations, i.e. the *depth* of the research. Let's start to analyse the steps of the algorithm by its flowchart shown by figure 7.2.

The first computational block, `get associations`, reads from the files the associations of the current state, keep in mind the current state may be composed by up to *width* words. Therefore, it isn't matter of a simple file reading. To maximise the parallelism of the readings a Matlab *parfor* statement is used. In fact, given that the readings are independent, it's possible to avoid to waste time waiting for the completion of the previous loop iterations.

The first filter remove from the list of associations, given by the files reading, all the entries which are not present in the embedding dictionary, for reasons explained before. Then the second filter delete all the entries just visited before. Random actions are taken with a probability equal to epsilon, if it is the case the Q-values among the chosen words and their previous states are computed, otherwise the Q-values are evaluated for all the couples and only the highest are kept. Last, the Reply Memory is updated and Q-values among the *cue word*, i.e. the starting word, and the next state, i.e. the chosen associations, are computed. The highest are stored and suggested. For all this computational blocks it's possible to exploit Matlab methods embedded parallelism. Loops are avoided where possible so computational time decreases.

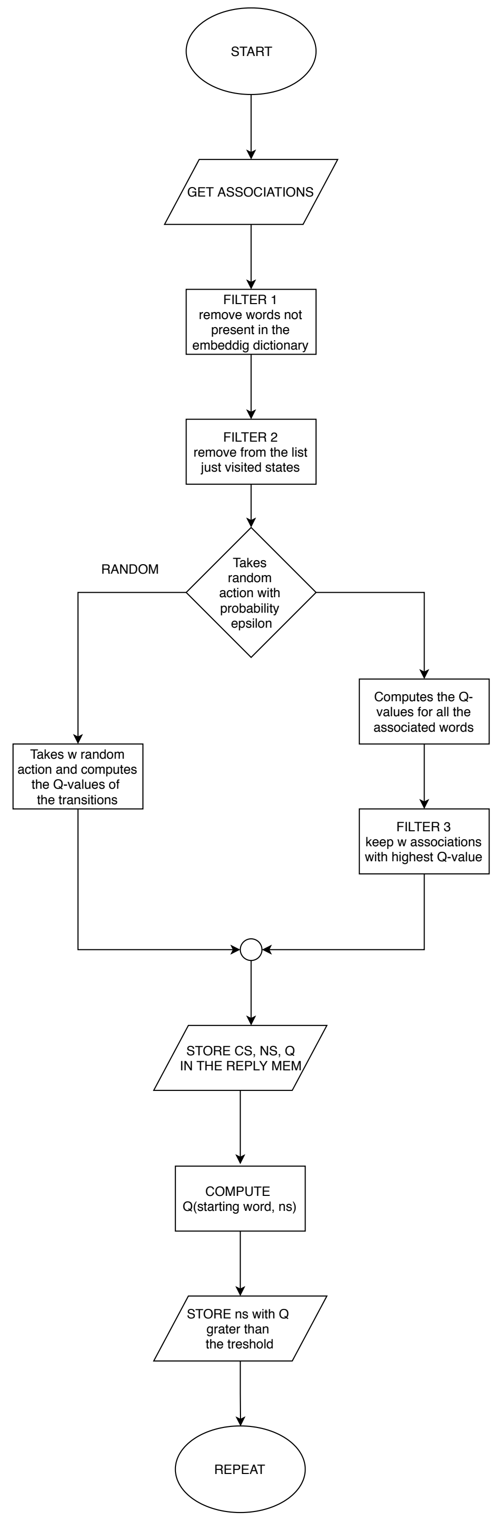
START
GET ASSOCIATIONS
FILTER 1 remove words not present in the embeddig dictionary
FILTER 2 remove from the list just visited states
Takes random action with probability epsilon
RANDOM
Takes w random action and computes the Q-values of the transitions
Computes the Q-values for all the associated words
FILTER 3 keep w associations with highest Q-value
STORE CS, NS, Q IN THE REPLY MEM
COMPUTE Q(starting word, ns)
STORE ns with Q grater than the treshold
REPEAT

Figure 7.2: Associative Machine Class loop body flowchart. It is executed a number of times equal to *depth*.

7.3 Intuitive Machine Eyes Implementation with Matlab

In line with the zero-shot classifiers [26] and [27], taken as an example of how to shift from images to word embeddings, a visual embedding neural network object is designed for the Matlab platform. The VENN (Visual Embedding Neural Network) Matlab class is highly inspired from DeViSE [26]. The choice to adopt it as a model, instead of ConSE [27], it is justified because we are not interested in the zero-shot learning feature, but into recognize images as points of the embedding space.

Matlab Implementation

The implemented VENN class, as suggested in [27], starts from a pretrained image classifier and a ready embedding dictionary; it's possible to work with whatever classifiers and embeddings dictionaries. Every image classifier final layers are composed by a fully connected layer and a softmax layer. This two layers are substituted by a fully connected layer with a number of outputs equal to the embedding vectors dimensionality followed by an output regression layer. Only this two layers need to be trained, so, the training is very fast. This technique known as transfer learning is the same adopted by [27]. Furthermore, a very small set of images per class is sufficient, then the training time is even more reduced.

Untrained Class Recognition We choose to implement a new architecture for our image classifier, but it was also possible to use a common classifier with fully-connected and softmax layers. In fact, after the classification, it would be very simple to get the embedding of the class by the embedding dictionary. Anyway, also this structure needs to retrain its final layers to classify the classes of our interest. However, there is a reason for which we decide for this solution. Actually, thanks to the output type of our classifier, becomes possible to obtain the classification of untrained classes by a manipulation of the embedding dictionary. Let's explain with an example, if the classifier is trained to recognize the classes: *man*, *woman* and *king* among the others, it generates their embeddings when images of these classes are supplied in input. Even if a non trained class is supplied, the classifier anyway generates an embedding. If the input image belong to the class *queen*, out of the instructed classes, unsurprisingly the output embedding will be something between *king* and *woman*. Therefore, by applying a sufficient number of images and computing the average of the classifier outputs, it's possible to obtain a new embedding that, if saved in the dictionary, permits to recognize a class for which the classifier is never trained.

Chapter 8

Methods

This chapter aims to shed light on the methods used to solve the issues encountered during the algorithm development. Such parameters tuning strategies, network architectures choice and results accuracy improvement.

8.1 Q-network Architecture

Deepmind studies, in particular the Atari project [11], suggests the agent could be implemented by two fully-connected layers. The number of neurons of such layers is not revealed but, actually, it's not a useful information. In fact, the number of neurons per layer highly depend by the training set and the number of output [39]. There exist different methods to identify the best architecture, some are based on heuristics, others to genetic algorithms [40]. The easiest one to implement is the exhaustive test, unfortunately very time-consuming. Anyway, [39] presents two equations, which give the number of neurons for both first and second hidden layers beyond which the neural network precision stop to increase. So, equations 8.1 gives un upper bound to the exhaustive test, making it possible:

$$\begin{aligned} N_1 &= \sqrt{\frac{m+2}{N}} + 2\sqrt{\frac{N}{m+2}} \\ N_2 &= m\sqrt{\frac{N}{m+2}} \end{aligned} \tag{8.1}$$

Where N_1 and N_2 are the number of hidden neurons, respectively for the first and second layers, m is the number of outputs and N is the training set size.

So, in Appendix B is available the Matlab script which tests and compares the architectures in terms of mean square error by a double loop which iterates up to the two hidden neurons bound N_1 and N_2 given by eq 8.1.

To better understand the problem let's try to run the exhaustive test. The training set is generated from the Association Machine Class, by setting *width* and *depth* the same and equal to twelve. Finally, the training set size N is of forty entries. Then, the outer and the inner loop iterate up to eight and four respectively, in fact by applying equation 8.1 with $m = 1$ and $N = 40$:

$$N_1 = 7.45 \rightarrow 8;$$
$$N_2 = 3.65 \rightarrow 4.$$

Unfortunately in this way the method is not deterministic. Due to the initial random distributions of weights and biases. By running three times the exhaustive test the mean square errors are different, as depicted by image 8.1.

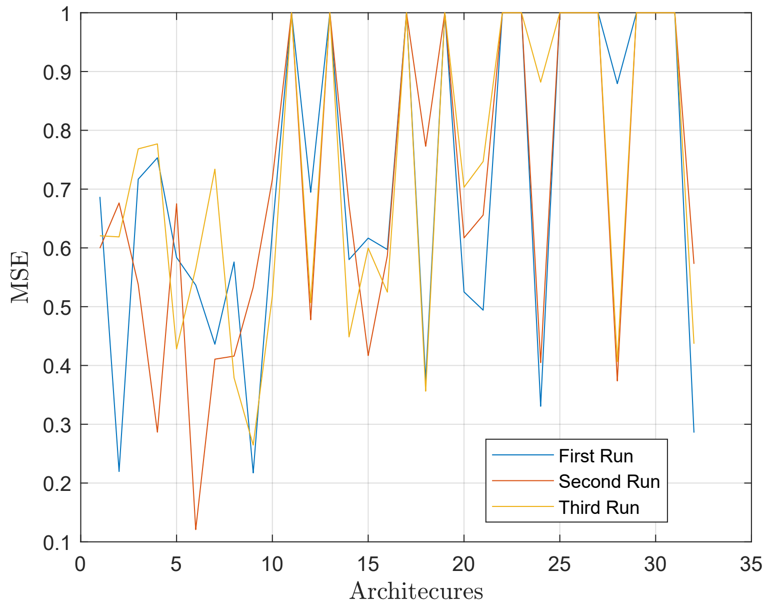

Figure 8.1: The exhaustive test didn't give deterministic results due to initial weights distributions: in the figure the test is run three times with the same training set but the results are quite different.

Note that the architectures are enumerated along the horizontal axis according to the iteration in which they are tested. To clarify, look at table 8.1 in which is written the architecture to iteration mapping.

(N_1,N_2)	iteration	(N_1,N_2)	iteration	(N_1,N_2)	iteration	(N_1,N_2)	iteration
(1,1)	1	(3,1)	9	(5,1)	17	(7,1)	25
(1,2)	2	(3,2)	10	(5,2)	18	(7,2)	26
(1,3)	3	(3,3)	11	(5,3)	19	(7,3)	27
(1,4)	4	(3,4)	12	(5,4)	20	(7,4)	28
(2,1)	5	(4,1)	13	(6,1)	21	(8,1)	29
(2,2)	6	(4,2)	14	(6,2)	22	(8,2)	30
(2,3)	7	(4,3)	15	(6,3)	23	(8,3)	31
(2,4)	8	(4,4)	16	(6,4)	24	(8,4)	32

Table 8.1: Horizontal axis mapping among architectures and iterations.

To exclude weights and biases initial state from the measure, another loop is added. In this very inner loop the same architecture is tested L times and then the mean square error is considered as the average of the L mean square errors. The exhaustive test now implies L times more but the results accuracy justify the time impact. The triple inner loop effect is evident in image 8.2. We discover in L equal to twenty a good compromise between accuracy and waiting time.

Looking at image 8.2 it's possible to observe how the mean square error changes suddenly among very near architectures. The last configuration with layers of N_1 and N_2 neurons is the best one only in the third run. Therefore, given an initial arbitrary distribution of the weights, the best architecture isn't the bigger one at all. The trend seems to be unpredictable a priori. Furthermore, the dataset size is not known due to the constraints of our application, in which the net has supposed to be retrained during its life. The constraint makes very difficult to understand which architecture is a good choice. The nets which well fit the training set are many, but as the dataset size changes, the accuracy is no more guaranteed. So, we decide to keep a net until it's mean square error and variance are under certain limits. Then, we substitute the net when it starts to be too much inaccurate with another architecture which better fit the new dataset.

Since the exhaustive test needs a lot of runs to complete the exploration we choose to implement a very simple heuristic. The starting point is the (N_1,N_2) architecture, it is tested H times and if it doesn't satisfy the requirements of mean square error and variance the hidden neurons number is slightly modified by decreasing it. The

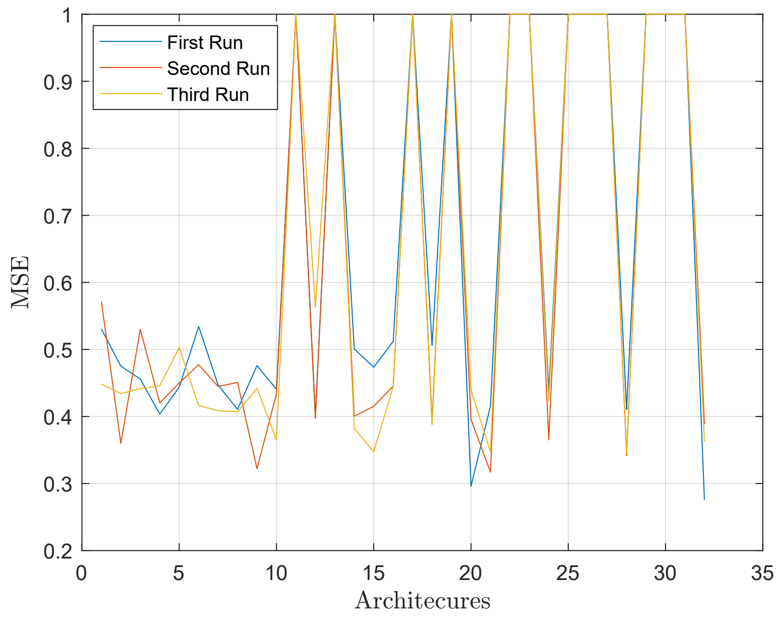

Figure 8.2: The image shown how the inner loop introduced, which repeat the training of every architecture multiple times, permits to find deterministic results by running the exhaustive test.

H parameter and the mean square error and variance are selected very carefully:

- We set H in the range [8, 16], it's a good compromise among heuristic convergence and waiting time, especially when the training set dimension increases.

- We leaved the mean square error unconstrained and decide to check the variance of the Q-value computations. Actually, we use the standard deviation, very close to the variance, set in the range [0.18, 0.20], about the 10% of the output range, i.e. $[-1, +1]$. This range permits to meet sufficiently precise results avoiding too restrictively requirements for the training, which lead to a long waiting time for the heuristic convergence. Lower the standard deviation, higher the H parameter to converge.

8.2 Relevant Transitions

Every transition of the system is stored inside the Reply Memory in a tuple with the following structure:

$$\langle \text{ current state, next state, q(current state, next state) } \rangle$$

At training time, as explained in section 7.1, the memory is processed and TD(0) method is applied. Given that the space exploration maintains an high degree of parallelism, it is not unusual which a single transition generates up to 64 or 128 transitions. By choosing to keep in consideration all the transitions, TD(0) method returns values very far from the minus and plus one range in which we try to trap the network results. In fact, by looking at equation 7.1, it's clear how if the received rewards are 64 or 128, it's very easy to exceed those bounds.

The number of rewards received is a very simple index of the transition importance, indeed, it permits to identify the main hubs, which are the most important transitions to update. It's crucial to understand how to train the net, in fact, transitions with too few negative rewards are not sufficiently explored to understand if they are really bad transitions or if some more explorations may lead to find good results. This fact brought us to choose to updates only the main transitions of each episode, by using as discriminant the number of received rewards per transition.

This choice greatly decreases the size of the training set and highly reduces the number of times it's necessary to instance another Q-network which better fit the dataset. Moreover, we fix a maximum dimension for the training set equal to 512 and 1024 entries during the experiments. By applying equation 8.1, it's visible how the maximum size of the network able to handle these datasets changes:

- 512 Entries Dataset: $N_1 = 26$ and $N_2 = 13$
- 1024 Entries Dataset: $N_1 = 37$ and $N_2 = 19$

8.3 Training Episodes

Training episodes are introduced to get examples to the agent and speed up the training. The structure of the episode is quite similar but no suggestions are generated and an array of good suggestions is given in input. All the Q-value among the array elements and the starting word are trained to one then, the Associative Machine is run and also all the found paths which connect the cue word with the array elements are automatically good rewarded.

A new memory called Reply Buffer is added with the only role to store cue word and array element couples to update those transitions to one and to not affect the Reply Memory which is processed to apply the TD(0) method.

8.4 Visual Embedding Neural Network

The Visual Embedding Neural Network is tested to evaluate both its accuracy as classifier and its capability to recognize new classes without training, but thanks only to dictionary manipulations. As explained previously VENN needs a trained classifier, from which it's possible to observe the inputs of the fully-connected layer. We instance a pretrained Googlenet, since it works better than the Alexnet [29] used in both DeViSE and ConSE [26], [27].

Moreover it's necessary an embedding dictionary to complete the architecture. Many free works are available, like Word2Vec [21] or FastText [37]. We choose for GloVe [38] since it is made available in different sizes, both for the dictionary entries and the space dimensionality. Especially the space dimension is a very critic factor by the point of view of the processing time. Usually the dictionaries are released encoding words in a 300-dimension space. GloVe releases also a 50-dimension database which we will use to test the classifier.

The datasets on which we perform the test is the Caltech 101 [31].

Chapter 9

Results

9.1 Associative Machine

In this chapter the Associative Machine is tested, the system parameters are set like explained in Methods, chapter 8. The algorithm is undergoing to three different tests to understand the agent capabilities:

1. Single Application Field Without Initial Training: The algorithm is trained to get suggestions in a single application area without any initial training episode;

2. Single Application Field With Initial Training: Differently from the previous test, the agent is initially trained with four user examples;

3. More Than One Application Fields: it is asked the agent to learn to suggest in different set of domains with initial user examples.

Every test is considered passed when at least eight suggestions are pertinent to the starting cue word. Pertinent means a system response like *China* if the theme are *Where to go for the holidays?* Furthermore, *Chinese* si also accepted since in the embedding dictionary the two words are very close, so it's very difficult to distinguish the cases.

9.1.1 Single Application Field Without Initial Training

In this section the algorithm is trained to suggest the user in specific and unique application field. Different domains are investigated and results are reported in the following.

Travel Idea In a series of episodes the agent are asked for suggestions about where to go for holidays. Every Associative Machine response is rewarded in the view of the user and the interview continues until the system recommends eight pertinent suggestions.

As starting point is considered an untrained instance of the agent which casually suggests a single pertinent word. In the example the agent satisfy the test requirements after six attempts. The entire system evolution is available on table C.1 in Appendix C. It is here shown the array of suggestions which conclude the test:

island, California, Japan, bay, glacier, coast, sea, castle.

Furthermore, in 9.1 is reported the number of times the heuristic iterates to find an agent architecture which fit the dataset and satisfy the deviation standard requirement of 0.18 given in Methods, chapter 8.

Episode	#architectures tested	Q-net structure N_1,N_2
1	12	14,7
2	28	17,9
3	8	23,12
4	0	23,12
5	0	23,12
6	0	23,12

Table 9.1: In order from left to right the episode index, the number of architectures tested before to converge and the architecture structures.

For every episode, identified by the left column, the table 9.1 shows the number of architectures trained before the required standard deviation is met on the new dataset on the second column. The right column shows the neurons number in the first and second layer, N_1 and N_2 respectively. Zero architectures tested means that the previous trained Q-network, retrained on the new transitions, still fits the new dataset under the given accuracy requirements.

What to play? With the same structure of the previous example here it is asked the agent suggestions to which instrument to play. Table C.2 in Appendix C shows the list of suggestions among the episodes. The test finishes at the fifth episode with the following array of suggestions:

Keyboard, drum, guitar, saxophone, trumpet, cello, clarinet, string.

Table 9.2 reports the iteration of the heuristic to converge at every episode and the architecture structure, like before.

Episode	#architectures tested	Q-net structure N_1,N_2
1	8	14,7
2	29	17,9
3	0	17,9
4	0	17,9
5	0	17,9

Table 9.2: In order from left to right the episode index, the number of architectures tested before to converge and the architecture structures.

9.1.2 Single Application Field With Initial Training

The same tests, previously accomplished without initial training, was here reproduced but four words are suggested to the agent before the first episode, just like they was a set of user preferences.

Travel Idea The travel destination test was reproduced training the system to four good types of destinations:

Africa, mountain, sea, Italy.

The test was stopped at the fifth attempt. The full system evolution is available in table C.3, in Appendix C. Conversely, the final suggestions are reported below:

mountain, Chinese, Russia, Asia, sea, China, Japan, coast.

Finally, the table 9.3 reports the heuristic iterations count to converge.

Episode	#architectures tested	Q-net structure N_1,N_2
1	13	19,10
2	10	23,12
3	19	24,12
4	0	24,12
5	0	24,12

Table 9.3: In order from left to right the episode index, the number of architectures tested before to converge and the architecture structures.

What to play? During the last single field experiment was reproduced the instrument problem with an initial training on the following four user examples:

Guitar, trumpet, drum, piano.

The agent satisfied the test requirements at the second run, with the following array of suggestions:

saxophone, tenor, guitar, organ, cello, viola, flute, trumpet.

The first try of the agent is reported in table C.4, in Appendix C. As in the previous cases the table 9.4 shown the heurstic iterations per episode and the network final structure.

Episode	#architectures tested	Q-net structure N_1,N_2
1	4	18,9
2	0	18,9

Table 9.4: In order from left to right the episode index, the number of architectures tested before to converge and the architecture structures.

9.1.3 More Than One Application Fields

The experiment investigates if a single agent is able to correctly operates in two different domains. The first test is performed by training sequentially the agent in the two operating fields. In the second test instead, the training is performed in parallel. The training sets generated by the two previously trained agents 9.1.1, are joined and used to train the Q-network.

Serial Training The test starts from an architecture trained to operate in a single application field. Then, the agent is trained in another domain. Later, the suggestions of the Q-network in both the domains are observed.

Initially the network is able to suggest travel ideas. Actually, the network trained previously in the example 9.1.2 is taken. Then, the agent is trained to suggest an instrument to play. The complete list of training episodes is shown in table C.5, in Appendix C. The heuristic report, as usual, is available below, in table 9.5.

Episode	#architectures tested	Q-net structure N_1,N_2
1	1	27,14
2	1	27,14
3	1	27,14
4	1	27,14
5	2	27,14
6	1	27,14
7	1	27,14
8	0	27,14
9	1	27,14

Table 9.5: In order from left to right the episode index, the number of architectures tested before to converge and the architecture structures.

The training elapses for nine episodes. At the ninth attempt the network generates the following results for the starting cue word *play*:

saxophone, tenor, guitar, organ, cello, viola, flute, trumpet.

If the agent is run with the cue word *travel*, after the second training, it generates the following array of suggestions:

rim, Asia, thick, China, crater, Russia, waves, slope.

Parallel Training Conversely from all the previous cases, here it was tested the agent behaviour with two different training set sizes. The former one has a training set of 512 entries, the latter 1024 entries. To train the Q-network the training sets of the two agents trained in 9.1.2 are joined. Especially for the 512 entries dataset a lot of items are dropped, since the two dataset joined are both of 512 entries. In the second case no items are dropped.

To find a neural network structure which fit the dataset, the heuristic is run. In both the cases the heuristic doesn't converge until the deviation standard requirement is increased from 0.18 to 0.20. Table 9.6 shown the heuristics iterations and the final network structure.

	#architectures tested	Q-net structure N_1,N_2
512 Entries Dataset	13	27,14
1024 Entries Dataset	60	32,16

Table 9.6: Parallel training heuristic report. Two different agents are trained, in the first row the agent own the same 512 entries dataset size of the previous examples but the standard deviation is increased from 0.18 to 0.20, to allow the heuristic convergence. In the second row the size of the dataset is increased to 1024 and the standard deviation is also increased to 0.20.

Finally, the agent results for the cue words *travel* and *play* are shown below:

- 512 Entries Dataset *travel* Suggestions: moth, bark, glacier, Russia, swimmer, Arctic, volcano, tropical;
- 512 Entries Dataset *play* Suggestions: bark, moth, swimmer, singer, chart, royal, glacier, trumpet;
- 1024 Entries Dataset *travel* Suggestions: tongue, earthquake, rim, canyon, snake, mouth, snail, typhoon;
- 1024 Entries Dataset *play* Suggestions: cyclone, tissue, pour, minute, metal, alto, blow, bottle.

9.2 Visual Embedding Neural Network

The classifier is tested to prove its accuracy on the Caltech 101 image dataset [31]. It is trained on 30 images per each classes and tested on 10. The VENN instance is than tried against untrained classes to see if the dictionary manipulations discussed in Methods are effective, chapter 8.

9.2.1 Caltech 101

The Caltech 101 dataset test is reduced from 101 categories to 68 (the full list is available in table C.6, Appendix C) because some classes are not present inside the GloVe dictionary [38], others has too few images to both train and test the classifier in line with our parameters. The classifier trained on 2040 images and tested on 670 images reach an accuracy of 88.66%.

9.2.2 Untrained Class Recognition

The classes excluded by the training, because with too few images, are here introduced in the dictionary, as explained in Methods, chapter 8, and classified. The number of untrained classes is 10 (as well listed in table C.7, in Appendix C) and the new test set is made of 354 images. The classifier accuracy just in the untrained classes is about 81.92%.

Chapter 10

Discussion

Associative Machine The core of the system is based on a neural network, trained with the Reinforcement Learning technique, which has to surf among all the known points of the embedding dictionary to meet the best answers to a cue word. The tests performed have the purpose to investigate the agent capability to effectively carry out the job. By looking at the single application field experiments is clear how te agent operates inside the embedding space. All the suggestions received are semantically near to the good rewarded words or to the words suggested by the user, in the last two tests, where user preferences are taken into account. To verify the assumption it's sufficient to check if the semantic closeness is maintained among the embeddings by looking at the results of the normalized scalar products between couples of words. Let's start to multiply the user preferences in the *travel* problem, *Africa*, *mountain*, *sea* and *Italy*, to the agent suggestions:

	Africa	Mountain	Sea	Italy
Mountain	0.1035	1.0000	0.4337	0.0903
Chinese	0.3764	0.0685	0.2300	0.3192
Russia	0.3072	0.0379	0.2923	0.4542
Asia	0.6526	0.1384	0.3582	0.3587
Sea	0.3039	0.4337	1.0000	0.2451
China	0.5535	0.1010	0.3031	0.4656
Japan	0.3479	0.0407	0.2677	0.4332
Coast	0.4657	0.4175	0.7634	0.3147

Table 10.1: Scalar product among the training words and the agent outputs in the *travel* problem.

The highlighted values propose a possible association among hint words and algorithm outputs. Blue results point out user preferences arrived in output, red results demonstrate closeness among training word and algorithm suggestions embeddings.

This result explain how the algorithm is able to extend the outputs set coherently with the examples received.

Given that the training of the *play* test has demonstrated to be very easy, in fact the training elapses for only two episodes, it's interesting to see if the embedding closeness among examples and outputs is higher or lower with respect to the previous case.

	Guitar	Trumpet	Drum	Piano
Saxophone	0.8602	0.8971	0.6837	0.8058
Tenor	0.6727	0.8183	0.4378	0.7779
Guitar	1.0000	0.8145	0.7434	0.7172
Organ	0.4346	0.5496	0.4807	0.5542
Cello	0.6831	0.7596	0.4545	0.9098
Viola	0.4938	0.6502	0.3068	0.7914
Flute	0.6566	0.8398	0.5940	0.8125
Trumpet	0.8145	1.0000	0.7534	0.8053

Table 10.2: Scalar product among the training words and the agent outputs in the *play* problem.

By evaluating the average values of the tables 10.1 and 10.2 are obtained 0.3649 and 0.7054. Therefore, in the *play* test the net outputs are closer to the examples and probably this fact leads to a faster training. The reason for which the average values are so different has to be inspected starting from the origin of the data, the embeddings. The *vec2word* Matlab method permits to obtain the vocabulary entry much close to the vector passed in input. If of interest, it's possible to specify the number of vocabulary entries to return. Thanks to this feature it's possible to obtain an arbitrary number of close words, not only the most. So, by asking for the closest embeddings to our example words in both the *travel* and *play* cases it's possible to evaluate the average distance among every example word and it's neighbourhood. Finally, by taking the average values of all the evaluated distances it's possible to define a value which denotes the richness of similar words given a set of example words. With a number of neighbours equal to ten, for each and every example word, the indexes of similarity with the neighbourhood in the *travel* test is 0.7851, in the *play* test instead is greater and equal to 0.8306. These values justify the assumption, i.e. the net converges quicker if the example words have a multitude of similar words inside the embedding dictionary. However, it's also true that since the agent finds the suggestions by surfing along the links given by the association database [15], non connected or very far words are difficult to suggest, even if the agent response to the word is very high. It's simply possible that it doesn't reach such word.

The last set of tests investigate the capacity of the net to be used in very far domains. During the first attempt the agent is serially trained to different application fields and it demonstrates to have the potentiality to operate in both, even if not perfectly.

The second training strategy was unsuccessful. In fact, just from the beginning, it has proven necessary to accept an architecture with an higher output variance to permit the heuristic to converge. Probably, if the heuristic which looks for the new neural network architectures has convergence problems, even though [39] demonstrates the number of neurons in the hidden layers are sufficient, the training set includes contrasting entries, collected among the training episodes on different domains.

Visual Embedding Neural Network The classifier experiment aims to understand if it were possible to get embeddings starting from images. The VENN outperforms the accuracy of many other classifiers tested on the Caltech 101 dataset ([34], [35], [36],), even thought all the convolutional layers are leaved untouched. In fact, DeViSE [26], which inspire VENN, raises its accuracy by partially retraining the convolutional layers. The classifier reaches an accuracy of 88.66% meanwhile state of the art result is 93.42% [33]. It is better than expected. Furthermore, the untrained classes recognition strategy has demonstrated to work satisfactory achieving an accuracy of 81.92%.

Chapter 11

Conclusion

The work had the purpose to implement an algorithm able to emulate human intuition. We found out that a dual layers neural network can accomplish the space exploration, bringing out the most significant points it crossed. The studies proved that the training set needs a special attention, since contradictory entries may lead to convergence problems. Moreover, we explores the possibility of achieving embeddings by considering no longer words, but for instance images, sounds, tastes or smells. We focused on images, modifying the standard images classifier final layers. It was obtained a system able to map images in the vectorial space in which the agent works. The training was accelerated thanks to the transfer learning technique, and in addition, a method to identify untrained classes were successfully exploited. The final results explained how, thanks to the support of the embedding space, the agent other than suggesting past solutions, pushes itself over the boundaries of what it has learned. Actually, the intuition described by the System 1 model *"is nothing more and nothing less than recognition"*, [7]. This sentence reflects our founding, and if the *Dual Process Theory* is right about that, i.e. respecting the biological processes, the difference between intuition and suggestion stays in the latter quality.

Appendices

Appendix A

Algorithms

A.1 Q-network Class

```matlab
classdef Qnetwork < matlab.mixin.SetGet
    %QNETWORK: implements a typical qnetwork trained with the TD(0) method.
    %State transitions are stored inside a Reply Memory and the
    %training method rewards all the transition stored in the Reply Memory
    %starting from the last one.
    properties
        emb      % embedding dictionary
        replyMem % to store transitions
        replyBuf % to store transition to reward directly to +/- 1
        qnet     % Qnetwork
        n1n2     % number of neurons in the 2 layers [N1 N2]
        TS       % Training Set to retrain a new network when accuracy decreases
        max_err  % max standard deviation
        n_of_shift   % reduce the number of transitions to train
        try_per_arch % architecture heuristic: training per arch before to change
            arch
        half_TS_size % half dimension of the training set
    end

    methods
        function obj = Qnetwork(emb)
            % Qnetwork: initializes propoerties to standard values.
            % It receives emb to set the net inputs.
            obj.emb = emb;
            obj.n1n2 = [4 2];
            obj.max_err = 0.18;
            obj.CreateQnet(obj.n1n2);
            obj.n_of_shift = -4;
            obj.try_per_arch = 16;
            obj.half_TS_size = 128;
        end

        function CreateQnet(obj, n1n2)
            % CreateQnet: instances the agent.
            % input width: emb.Dimension*2 (current and next word, aka current and
                next state)
            % 2 fully connected layers, see CheckAccuracy method
            % output width: 1, the Q-value
            width = 2*obj.emb.Dimension;
            obj.qnet = fitnet(n1n2); % 2 hidden layer
            obj.qnet = init(obj.qnet);
            obj.qnet.trainFcn = 'trainlm'; % otherwise 'trainbr';
            % the input width is setted here
            obj.qnet = configure(obj.qnet,rand(width,2),rand(1,2));
        end

        function qvalue = Q_value(obj,cs,ns)
            %computes q value starting from current state
            %and next state, which is also the action
            cs_emb = word2vec(obj.emb,cs);
            ns_emb = word2vec(obj.emb,ns);

            input = [cs_emb,ns_emb]';

            qvalue = obj.qnet(input);
        end

        function ReplyMem(obj,cs,ns,q)
            %store cs, ns/action in string format and q estimation
            %to train the network once the reward comes.
            if isempty(obj.replyMem)
                obj.replyMem = [cs ns q];
            else
                obj.replyMem = [obj.replyMem; cs, ns, q];
            end
        end

        function ReplyBuf(obj,cs,ns,r)
            %store cs, ns/action in string format and the reward
            %to train the passed transition directly.
            if isempty(obj.replyBuf)
                obj.replyBuf = [cs ns r];
            else
                obj.replyBuf = [obj.replyBuf; cs, ns, r];
            end
        end

        function HandleTS (obj,cs,ns,td0,r)
            % HandleTS: keeps only  2*TS_half_size transitions. The most
            % significant are considered the most negative rewarded and the
            % positive rewarded.

            if isempty(obj.TS)
                obj.TS = [cs ns td0 r];
            else
                obj.TS = [obj.TS; cs, ns, td0, r]; % add all the new transitions
                [~, ii] = unique( flipud( obj.TS(:,1)+obj.TS(:,2) ) ); % remove
                    equal transition, keep the newer
                obj.TS = flipud( obj.TS );
                obj.TS = obj.TS(ii,:); % unique
                %obj.TS = flipud( obj.TS );

                i_1 = find(str2double(obj.TS(:,4)) > 0); % takes thumb up
                    transitions
                residual =  obj.half_TS_size - length(i_1);
                [~, i_2] =
                    mink(str2double(obj.TS(:,4)),obj.half_TS_size+residual); %
                    takes worst rewarded transitions
                i_new = [i_1;i_2];
                obj.TS = obj.TS(i_new,:);
                i_new = randperm(length(obj.TS));
                obj.TS = obj.TS(i_new,:);
            end
        end

        function ResetReplyMem(obj)
            % ResetReplyMem: Reset ReplyMem.
            obj.replyMem = [];
        end

        function ResetReplyBuf(obj)
            % ResetReplyBuf: Reset ReplyBuf.
            obj.replyBuf = [];
        end

        function [training_set] = TrainNetwork(obj,alpha,gamma,suggestions,rewards)
            % TrainNetwork: train the net using the Reply Memory and
            % TD(0) method: q(cs,ns)=reward(cs,ns)+gamma*max_q
            %  it directly trains the transition stored in the ReplyBuf

            % get data from replyMem
            cs = obj.replyMem(:,1); %[cs,ns]
            ns = obj.replyMem(:,2); %[cs,ns]
            q = str2double(obj.replyMem(:,3));

            % TD(0)
            rm_length = length(q);
            td0 = zeros(rm_length,1);
            r = zeros(rm_length,1);
            for kk=1:rm_length
                k = rm_length + 1 - kk;
                w = cs(k);
                % training episode: if ns(k) is a leaf of the path q <- r
                % which is 1 if is a finalWord, -1 otherwise
                if td0(k) == 0
                    l = find(strcmp(suggestions, ns(k)));
                    if ~isempty(l)
                        q_old = rewards(l);
                        td0(k) = rewards(l);
                        r(k) = rewards(l);
                        r_old = rewards(l);
                    else % maybe useless
                        q_old = -1.0;
                        td0(k) = -1.0;
                        r(k) = -1;
                        r_old = -1;
                    end
                % otherwise update using TD(0) equation
                else
                    td0(k) = (1-alpha)*q(k) + alpha*(gamma*td0(k)+sign(r(k))); % TD(0) equation
                    q_old = td0(k);
                    r_old = r(k);
                end
                % start find the previous transition along the ReplyMem
                for ii=kk:rm_length-1
                    i = rm_length + 1 - ii - 1;

                    % walking up the path ... when the connection is found
                    % if it's a suggestion break, it's found in future
                    % iterations and set to 1. Stop the TD(0) propagation
                    % on the path.
                    % if it is to update positevly set +1 both q and r
                    % otherwise substitute the q (or take the average of
                    % the q received, better) and increment r to understand
                    % how many paths cross the state and negative reward it
                    if ns(i) == w
                        if ismember(ns(i),suggestions)
                             break;
                        else
                            if r_old > 0
                                r(i) = r_old;
                                td0(i) = q_old;
                                break;
                            else
                                if r(i) > 0
                                    break;
                                end
                            end
                            r(i) = r(i) + r_old;
                            td0(i) = q_old;
                        end
                        break;
                    end
                end % for ii
            end % for kk

            % reduce the number of transition to update
            s = bitshift(rm_length, obj.n_of_shift);
            if s == 0
                s=1;
            end
            fprintf("RM size: %i, TS size: %i (n_of_shift =
                %i)\n",rm_length,s,obj.n_of_shift)

            [~, i_1] = mink(r,s);    % takes worst rewarded transitions
            [~, i_2] = find(r>0);% takes good rewarded transitions
            i_s = [i_1;i_2];

            r_s = r(i_s);
            td0_s = td0(i_s);
            cs_s = cs(i_s);
            ns_s = ns(i_s);
            q_s = q(i_s);

            % insert replyBuf transition in the training set
            if (~isempty(obj.replyBuf))
                cs_s = [cs_s;obj.replyBuf(:,1)];
                ns_s = [ns_s;obj.replyBuf(:,2)];
                rb_r = str2double(obj.replyBuf(:,3));
                r_s = [r_s; rb_r];
                q_s = [q_s; rb_r];
                td0_s = [td0_s; rb_r];
            end

            ts_length = length(cs_s);
            fprintf("training set: %i\n",ts_length);

            % mix the training set
            rp = randperm(ts_length);

            r_s = r_s(rp);
            td0_s = td0_s(rp);
            cs_s = cs_s(rp);
            ns_s = ns_s(rp);
            q_s = q_s(rp);

            input_s(:,1:50) = word2vec(obj.emb,cs_s);
            input_s(:,51:100) = word2vec(obj.emb,ns_s);

            %train the net
            obj.qnet = train(obj.qnet,input_s',td0_s','showResources','yes');

            new_q = obj.qnet(input_s')';
            err = td0_s-new_q;
            disp(table(cs_s,ns_s,q_s,r_s,td0_s,new_q, err));

            % update TS
            obj.HandleTS(cs_s,ns_s,td0_s,r_s);

            training_set = [input_s,td0_s];
            % check if the agent accuracy is sufficient, run until a good
            % agent is found. Time consuming.
            obj.CheckAccuracy();
        end

        function CheckAccuracy(obj)
            % CheckAccuracy: heuristic to find an agent architecture which
            % fit the training set with standard deviation constraint

            % take the training set
            cs = obj.TS(:,1); %[cs,ns]
            input(:,1:50) = word2vec(obj.emb,cs);
            ns = obj.TS(:,2); %[cs,ns]
            input(:,51:100) = word2vec(obj.emb,ns);
            t = str2double(obj.TS(:,3));
            ts_size = length(cs);
            % compute std deviation and mean error
            q = obj.qnet(input');
            err = t'-q;
            me = mean(err);
            ds = std(err);
            % set the threshold above which the agent is not accepted
            e = obj.max_err;
            e_neg = -1*e;
            % If it is not necessary to change the net
            if  and(ds > e_neg, ds < e)
                fprintf("Still good, unchanged (%i,%i):\nmean err:\t%i\nstd
                    dev\t%f\n\n",obj.n1n2, me,ds);
                return;
            end

            % compute theoric number of neurons per layer, check thesis
            [N, ~] = size(obj.TS);
            m = 1;
            N1 = ceil( sqrt((m+2)/N) + 2*sqrt(N/(m+2)) );
            N2 = ceil( m*sqrt(N/(m+2)) );
            fprintf("N1 = %i, N2 = %i\n", N1, N2);
            obj.n1n2 = [N1,N2];
            % run the heuristic to find a good agent
            maxitr = N2*obj.try_per_arch;
            fprintf("maxitr: %i\n",maxitr);
            for i=1:maxitr
                    % after some iterations change the architecture
                    if and(mod(i,obj.try_per_arch) == 0, N2>1)
                       N1=N1-2;
                       N2=N2-1;
                    end

                    net = fitnet([N1,N2],'trainlm'); % otherwise 'trainbr'
                    net = init(net);

                    net = configure(net,input',t');
                    net = train(net,input',t','showResources','yes');
                    % compute the error
                    q = net(input');
                    err = t'-q;
                    me = mean(err);
                    ds = std(err);
                    %check the net
                    if  and(ds > e_neg, ds < e)
                        fprintf("#%d Good performances on %i entries
                            (%i,%i):\nmean err:\t%i\nstd dev\t%f\n",i,ts_size, N1,
                            N2, me,ds);
                        obj.qnet = net;
                        return;
                    end

                    fprintf("#%d Bad performances on %i entries (%i,%i):\nmean
                        err:\t%i\nstd dev\t%f\n",i,ts_size, N1, N2, me,ds);
            end
            fprintf("failed\n");
        end

    end

end
```

A.2 Association Engine Class

```
classdef AssociationEngine < matlab.mixin.SetGet
    %ASSOCAITION ENGINE: it uses free-associations to move from a starting
    %state. The free associations are collected by University of South
    %Florida. Association engine uses Appendix B, reformatted thanks to a
    %Python script.
    %It takes associations and choose the w most signficant transitions
    %using a qnetwork as discrimant.

    properties
        emb                    %embedding dictionary
        qnet                   %qnetwork
        reward                 %reward
        startingWord           %cue word
        suggestions            %suggestions/intuitions/outputs
        last_training_set      % last transitions used to train the net
        ts                     %training set
        n_suggestions          %number of suggestions
    end

    properties (Constant)
        alpha = 0.1;      %bellman learning parameter
        gamma = 0.9999;   %discount factor
        n = 100;          %number of associated words
        eps = 0.1;        %probability to take random cues independetly from qvalues
    end

    methods

        function obj = AssociationEngine(emb)
            %ASSOCAITIONENGINE: instances the Q-network, stores the emb
            %dictionary.
            obj.emb = emb;
            obj.qnet = Qnetwork(emb);
            obj.n_suggestions = 8;
        end

        function Episode(obj,maxItr,w,startingWord,finalWord)
            % starting from startingWord the network has to reach one of
            % the finalWord. The Q-Network is trained to achieve desired
            % connections-path.
            obj.startingWord = startingWord;
            obj.qnet.ResetReplyMem();
            obj.qnet.ResetReplyBuf();
            %if starting word isn't inside emb dictionary: premature
            %finishing
            if ~isVocabularyWord(obj.emb,startingWord)
                disp("The starting cue isn't a vocabulary word");
                return;
            end

            % if some finalword isn't known exit
            if ~isempty(finalWord)
                to_exit = isVocabularyWord(obj.emb,finalWord);
                if any(to_exit == 0)
                    disp("missing words:"); disp(finalWord(~to_exit));
                    return;
                end

                %if finalword is not empty, we are in a training episode.
                %Therefore, we have to train the qnet to consider
                %every finalword as a goal for the startingword
                % store inside ReplyBuf training suggestions, they are
                % trained as q=1 and are stored here because ReplyMem
                % transition are post processed with TD(0) method
                obj.qnet.ReplyBuf(repmat(startingWord,[length(finalWord) 1]),...
                                     finalWord', ones(length(finalWord),1));
            end

            cs = startingWord; % cs = current state
            old_states = cs; % just visited states
            obj.suggestions = []; % suggestions found list
            qvalue_suggestions = []; % qvalues of the suggestions

            for i=1:maxItr
                fprintf('### Iteration %i ###\n',i);
                %get possible actions/nextstate
                [input,~] = obj.FindAssociations(cs',obj.n);
                cs = input(1,:);
                cues = input(2,:);
                % filter1 removes old states from possible new states
                filter1 = ~ismember(cues,old_states);
                cues = cues(filter1);
                cs = cs(filter1);
                % filter2 removes missing vocabulary words
                filter2 = isVocabularyWord(obj.emb,cues);
                cues = cues(filter2);
                cs = cs(filter2);

                % if all the possible next states were just visited or are
                % not present inside vocabulary: premature finishing
                if (isempty(cues))
                        fprintf("premature finishing, zero cues\n");
                        break;
                end
                % with a ceratin probability epsilon takes random actions.
                % otherwhise computes Q value for each and every cues and
                % choose the w highest
                if(obj.eps > rand)
                   filter3rand = randperm(length(cues), min([w,length(cues)]));
                   cs = cs(filter3rand);
                   cues = cues(filter3rand);
                   max_q = obj.qnet.Q_value(cs,cues);
                else
                    % q values evaluation
                    qvalue = obj.qnet.Q_value(cs,cues);
                    % filter3: selects the w highest q values
                    [max_q, filter3] = maxk(qvalue,min([w,length(cues)]));
                    cues = cues(filter3); %next state takes the action with the
                        best qvalue
                    cs = cs(filter3);
                end

                % assign next states after 3 level of filters
                ns = cues;

                % Update Reply Memory to update the net once rewards come
                obj.qnet.ReplyMem(cs',ns',max_q');

                % Check if qvalue(startingword,cues(i)) it's greater than
                % the saved thumb up states. if it is save it here.
                % Only if finalword is empty, otherwise it's a training
                % episode and the aim is to reach finalword.
                if (isempty(finalWord))
                    qvalue_startingword = obj.qnet.Q_value(repmat(startingWord,[1
                        length(cues)]),cues);
                    qvalue_suggestions = [qvalue_suggestions;
                        qvalue_startingword'];
                    [qvalue_suggestions, ll] = maxk(qvalue_suggestions
                        ,obj.n_suggestions);
                    obj.suggestions = [obj.suggestions; cues'];
                    obj.suggestions = obj.suggestions(ll);
                end

                %disp([cs', ns', max_q']);

                % Set current state as next state
                cs=ns;
                % Add old states to avoid returning on it
                old_states = unique([old_states cs], 'stable');

            end

            if isempty(finalWord)
                disp("suggestions:");
                disp([obj.suggestions qvalue_suggestions]);
            else
                obj.TrainQnet(finalWord, ones(1,length(finalWord)));
            end
        end

        function TrainQnet(obj, suggestions, rewards)
            %TrainQnet: for training episodes. It runs TD(0) method
            %on the Reply Memory and also trains suggested transitions
            %stored in the ReplyBuf.
            obj.last_training_set = obj.qnet.TrainNetwork(obj.alpha,obj.gamma,
                suggestions, rewards);
            obj.qnet.ResetReplyMem();
            obj.qnet.ResetReplyBuf();
        end

        function Reward (obj,rewards)
            %Reward for non-training episodes. Suggestions are rewarded and
            %TD(0) method is applied starting from the ReplyMem. The
            %suggestions are trained to +1/-1 thanks to the ReplyBuf, it
            %depends by the rewards content.
            obj.qnet.ReplyBuf(repmat(obj.startingWord,[length(rewards) 1]),...
                                 obj.suggestions, rewards');
            obj.TrainQnet(obj.suggestions, rewards);
        end
    end

    methods(Static)

        function [cues,fas] = FindAssociations(word,n)
            % FindAssociations: returns the vector of the n most associated
            % words to each word inside the input vector. Every output words
            % is coupled with the input word.

            % free-associations appendix-B reformatted in Python directory
            path = "C:\Users\gianl\Documents\Python\lsa\data";
            wordPath = fullfile(path,word + ".csv");

            % since every word(i) produces potentially a different number
            % of associations to parallelise the for loop a cell matrix of
            % length(word)*n is used. In this way every iteration is
            % independet from the others.
            parfor i=1:length(word)
                if isfile(wordPath(i))

                    %how to takes forward associations, for the moment
                    %useless.
                    %%M = csvread(wordPath(i),1,0);
                    %%nmax = min([n length(M(:,1))]);
                    %%fas(i,:) = M(1:nmax,1); %forward association strenght
                    % associated words loading, inside files are ordered by
                    % forward association value. The stronger the upper.
                    fid = fopen(wordPath(i),'r');
                    words = lower(string(strsplit(fgetl(fid),",")));
                    nmax = min([n length(words)-1]);
                    cues_cell{i,:}  = words(2:nmax+1);
                    cs_cell{i,:}    = repmat(word(i),[1 nmax]);
                    fclose(fid);
                else
                   cues_cell{i,:} = "";
                   cs_cell{i,:} = "";
                   %fas(i,:) = 0;
                end
            end
            % now parfor output has to be reformatted...
            cues = zeros(2,numel(cues_cell));
            for i=1:length(cues_cell)
                cues = [cues(1,:), string(cs_cell{i,:}); ...
                        cues(2,:), string(cues_cell{i,:})];
            end
            %fas = rmmissing(fas(:));
            cues = cues(:,length(word)+1:end);
            [~, i_temp] = unique(cues(2,:)); % remove equal outputs
            cues = cues(:,i_temp);
            fas = 0;
        end

    end
end
```

A.3 Visual Embedding Neural Netowrk Class

```
classdef Venn < matlab.mixin.SetGet
    %VENN classifies images, to be initilized needs an embedding dictionary
    %and pretrained classifier

    properties
        emb                   % embedding dictionaries
        pretrained_net        % pretrained classifier (googlenet, alexnet, etc..)

        nImgs                 % training images per class
        classes               % classes to train took from the rootFolder
        residual_classes      % classes with too few images: img < nImgs+10
        classes_to_remove     % classes of the rootFolder not present as embedding
        imageSize             % pretrained net input image size

        trainingSet            % for final layers
        finegrainTrainingSet  % for final layers and pretrained net
        testSet                % to test the net

        trainingImages
        finegrainTrainingImages
        testImages

        trainingEmb
        finegrainEmb
        testEmb

        testNetEmbOutput

        threeLayerGraph   % layer graph of the 3 final layers
        threeLayerNet     % 3final layer trained
    end

    methods
        function obj = Venn(emb,pretrained_net)
            %VENN basic initialization
            obj.emb = emb;
            obj.pretrained_net = pretrained_net;
            obj.imageSize = pretrained_net.Layers(1).InputSize;
            obj.threeLayerGraph = obj.BuildThreeLayerGraph();
        end

        function lgraph = BuildThreeLayerGraph(obj)
            % buildThreeLayerGraph returns the 3 layers
            % with regression output equal to the emb dimension
            InputLayer = imageInputLayer([1 1 1024], 'Name', 'vennInput');
            FCLayer = fullyConnectedLayer(obj.emb.Dimension, ...
                'Name','fc300out', ...
                'WeightLearnRateFactor',10, ...
                'BiasLearnRateFactor',10);
            RegressionOutputLayer = regressionLayer('Name','routput');
            layers = [InputLayer, FCLayer, RegressionOutputLayer];
            lgraph = layerGraph(layers);
         end

        function residual_classes = TrainingSettings(obj,rootFolder,nImgs)
            %trainingSettings set training and test sets

            % get valid classes from the image folder
            obj.classes = Venn.GetValidClasses(rootFolder,obj.emb);
            imds = imageDatastore(fullfile(rootFolder, obj.classes),
                'LabelSource', 'foldernames');

            % set number of images for each training classes
            tbl = countEachLabel(imds);
            c = tbl{tbl{:,2}> nImgs+9,1};
            obj.residual_classes = tbl{tbl{:,2}< nImgs+10,1};
            residual_classes = obj.residual_classes;
            obj.classes = c;
            imds2 = imageDatastore(fullfile(rootFolder, string(c)), 'LabelSource',
                'foldernames');
            imds2 = splitEachLabel(imds2, nImgs+10, 'randomize');

            r = nImgs/(nImgs+10);
            %Split images in Training and Test Sets
            [obj.trainingSet,obj.finegrainTrainingSet ,obj.testSet] =
                splitEachLabel(imds2, r, 0.00001,'randomize');

            %resize images...
            obj.trainingImages = obj.ResizeImages(obj.trainingSet);
            obj.finegrainTrainingImages =
                obj.ResizeImages(obj.finegrainTrainingSet);
            obj.testImages = obj.ResizeImages(obj.testSet);
            %get classes labels
            obj.trainingEmb = obj.GetEmb(obj.trainingSet);
            obj.finegrainEmb = obj.GetEmb(obj.finegrainTrainingSet);
            obj.testEmb = obj.GetEmb(obj.testSet);
        end

        function imageSet = ResizeImages(obj,set)
            %resizeImages: prepare an image set

            imgs_temp = readall(set);
            imageSet = uint8(  zeros( obj.imageSize(1), obj.imageSize(2),
                obj.imageSize(3), length(imgs_temp) )  );
            for i=1:length(imgs_temp)
```

```
                img=imgs_temp{i};
                [~,~,c] = size(img);
                if c == 1
                    img = cat(3,img,img,img);
                end
            imageSet(:,:,:,i) = imresize(img,obj.imageSize(1:2));
            end
        end

        function embSet = GetEmb(obj,set)
            %get embeddings

            labels_string = strrep(string(set.Labels),'_','');
            embSet = zeros(length(labels_string),obj.emb.Dimension);
            for i = 1:length(labels_string)
                embSet(i,:) = word2vec(obj.emb,labels_string(i,:));
            end
        end

        function net = TrainFinalLayer(obj,imageSet,embSet)
            %TrainFinalLayer: train final layers

            % get activations of the pretrained classifier fc layer
            PNet_layer_of_interest = obj.pretrained_net.Layers(end-3).Name;
            PNout = activations(obj.pretrained_net,imageSet,...
                                PNet_layer_of_interest,'OutputAs','channels');

            % Train the Network
            options = trainingOptions('sgdm', ...
                                    'MaxEpochs',20,...
                                    'InitialLearnRate',1e-4, ...
                                    'Verbose',false, ...
                                    'Plots','training-progress');

            obj.threeLayerNet =
                trainNetwork(PNout,embSet,obj.threeLayerGraph,options);
            net = obj.threeLayerNet;
        end

        function [stringResults, embResults] = TestNet(obj,imageSet,embSet)
            %TestNet: tests the trained net on the test set

            % get activations
            PNet_layer_of_interest = obj.pretrained_net.Layers(end-3).Name;
            PNout = activations(obj.pretrained_net,imageSet,...
                                PNet_layer_of_interest,'OutputAs','channels');
            % test net
            prediction = predict(obj.threeLayerNet,PNout);
            obj.testNetEmbOutput = prediction;
            embResults = prediction;
            prediction_string = strings(length(prediction(:,1)),1);
            for i=1:length(prediction(:,1))
                prediction_string(i) = vec2word(obj.emb,prediction(i,:));
            end

            emb_string = vec2word(obj.emb,embSet)';
            stringResults = [prediction_string,emb_string];
        end

        function [images, emb,str] = GetNewSet(obj, rootFolder, class)
          %GetNewSet: useful to prepare new single class test set
            imds = imageDatastore(rootFolder);
            str = string(imds.Labels);
            images = obj.ResizeImages(imds);
            emb = repmat(word2vec(obj.emb,class),[imds.numpartitions, 1]);
         end
    end

    methods(Static)
         function [vennObj, results, residual_classes] = Example0 (emb, n)
             % run an example on caltech 101
             v = Venn(emb,googlenet);
             residual_classes =
                 v.TrainingSettings(fullfile('C:\Users\gianl\Documents\MATLAB\img',
                 '101_ObjectCategories'),n);
             v.TrainFinalLayer(v.trainingImages,v.trainingEmb);
             [results, ~] = v.TestNet(v.testImages,v.testEmb);
             vennObj = v;
         end

         function [vennObj, results] = ExampleBasic (n)
             % run an example on caltech 101 with the matlab emb
             emb = readWordEmbedding('exampleWordEmbedding.vec');
             [vennObj, results, ~] = Venn.Example0(emb,n);
         end

         function [vennObj, results, qResults, qImg] = ExampleNewDict (emb,n)
             % Create a new reduced dictionary with only the entries which
             % represent the class to recognize.
             % Then run ExampleBasic and test the net on the residual
             % classes, i.e. classes with too few images to be trained.

             % to save the dictionary
             rootFolder = 'C:\Users\gianl\Documents\MATLAB';
             [newDicPath,~] = Venn.StoreNewEmbDB(rootFolder,emb);

             % get new dictionary
             emb = readWordEmbedding(newDicPath);

             % run example0
             [vennObj, results, residual_classes] = Venn.Example0(emb,n);

             % get the embedding for each untrained class (residual classes)
             for i=1:length( residual_classes)
                c = string(residual_classes(i));
                [im, em] = vennObj.GetNewSet(fullfile( ...
                     'C:\Users\gianl\Documents\MATLAB\img\101_ObjectCategories',c),c);
                [~, qResultsEmb] = vennObj.TestNet(im,em);
                % find the average prediction for the untrained class
                newEm = sum(qResultsEmb);
                newEm = newEm/norm(newEm);
                % store it
                [newDicPath,~] = Venn.AddEmbsToEmbDB(newDicPath,c,newEm);
             end

             % get new dictionary
             vennObj.emb = readWordEmbedding(newDicPath);
             qResults=[];
             % for every untrained class
              for i=1:length( residual_classes)
                c = string(residual_classes(i));
                % get test set
                [qImg, qEmb, str] =
                    vennObj.GetNewSet(fullfile('C:\Users\gianl\Documents\MATLAB\img\101_ObjectCategories',c)
                % test on the class
                [qResults2, ~] = vennObj.TestNet(qImg,qEmb);
                % append
                qResults = [qResults;qResults2];
              end

         end

         function [filePath,emb2store_string] = StoreNewEmbDB(rootFolder,emb)
             %get all the valid classes of a folder and store its embedding
             %in a new dictionary

             imgPath = fullfile(rootFolder,'img\101_ObjectCategories');
             embPath = fullfile(rootFolder,'emb\embDB.txt');
             classes = Venn.GetValidClasses(imgPath,emb);

             emb2store_string = classes;
             emb2store = word2vec(emb,classes(:));

             % no duplicates, alphabetical order
             [emb2store_string,index] = sort(emb2store_string);
             emb2store = emb2store(index,:);
             [emb2store_string, index] = unique(emb2store_string, 'rows');
             emb2store = emb2store(index,:);
             %emb2store = normalize(emb2store','norm')'; %normalize, its
             %correct but performance decreases so, we can avoid it
             emb2store = rmmissing(emb2store);
             emb2store_string = rmmissing(emb2store_string);
             fid = fopen(embPath,'w');
             for ii = 1:size(emb2store,1)
                fprintf(fid,'%s %g ',emb2store_string(ii),emb2store(ii,:));
                fprintf(fid,'\n');
             end
            fclose(fid);
            filePath = embPath;
         end

         function [filePath,emb2store_string] =
             AddEmbsToEmbDB(embPath,str2store,emb2store)
             % AddEmbsToEmbDB: Starting from the embedding dictionary stored at
                 the path
             % 'rootFolder' add a new entry

             emb = readWordEmbedding(embPath);
             strVoc = emb.Vocabulary';
             embVoc = word2vec(emb,strVoc(:));

             for ii = 1:size(emb2store,1)
                 if (isVocabularyWord(emb,str2store(ii)))
                        embVoc(strVoc==str2store(ii),:) = emb2store(ii,:);
                        str2store(ii) = "";
                 end
             end

             index = str2store(:) ~= "";
             str2store = str2store(index);
             emb2store = emb2store(index,:);
             strVoc = [strVoc; str2store];
             embVoc = [embVoc;emb2store];
             [strVoc, index2] = sort(strVoc);
             embVoc = embVoc(index2,:);

             fid = fopen(embPath,'w');
             for ii = 1:size(embVoc,1)
                fprintf(fid,'%s %g ',strVoc(ii,:),embVoc(ii,:));
                fprintf(fid,'\n');
             end

            fclose(fid);
            filePath = embPath;
            emb2store_string = str2store;
         end

         function classes = GetValidClasses(rootFolder,emb)
            % GetValidClasses: get valid classes from rootFolder
            % Remove classes not present as embeddings
```

```

            classes = categorical(cellstr(ls (rootFolder)));
            classes_to_remove = {'.','..'};
            j=3;
            for i=1:length(classes)
                if(isVocabularyWord(emb,strrep(string(classes(i)),'_','')))
                    continue
                else
                    classes_to_remove{j} = char(classes(i));
                    j=j+1;
                end
            end
            % get string array of valid classes
            classes = string(categories(removecats(classes,classes_to_remove)));
        end

    end % end static methods
end % end class



%
```

A.4 Associations Reformat - Python Script

```python
# -*- coding: utf-8 -*-
def takeTables(inputFile,outputFile):
    fin = open(inputFile)
    buf = fin.read()
    buf = buf.split("\n")

    i = 0
    while i<len(buf):
        #row = buf[i].split(" ")
        #splittedRow = filter(None, row)
        name = buf[i][0:13].strip()
        values = buf[i][13:33]
        words = []
        strenght = []
        i+=1
        if "FSG" in values:
            words.append(name)
            i+=1
            name = buf[i][0:13].strip()
            values = buf[i][13:33]
            while len(name):
                words.append(name) # associated element
                newStrenght = values.split(".")
                newStrenght = [newStrenght[0]+ "." + item for item in newStrenght[1:]]
                strenght.append(newStrenght)
                i+=1
                name = buf[i][0:13].strip()
                values = buf[i][13:33]
            with open(outputFile + words[0] + ".csv", 'w') as f:
                print >> f, ','.join(words)
                for item in strenght:
                    print >> f, ','.join(item)
                f.write("\n\n")
    print "Bye!"
```

Appendix B

Methods

B.1 Q-network Architecture: Exhaustive Test

```
% N number of samples
% m number of output
% first hidden layer max size : sqrt((m+2)/N) + 2*sqrt(N/(m+2))
% second hidden layer max size : m*sqrt(N/(m+2))

replyMem = ae.last_training_set;
[a, b] = size(replyMem);
input = zeros(a,b-1);
input(:,1:100) = replyMem(:,1:100);
target = replyMem(:,101);

m = 1;
N = length(target);

f_size = ceil( sqrt((m+2)/N) + 2*sqrt(N/(m+2)) );
s_size = ceil( m*sqrt(N/(m+2)) );

y = zeros(a,f_size*s_size);
y_internal = zeros(a,10);
k=1;
index = zeros(f_size*s_size,2);
for i=1:f_size
   for j=1:s_size
     for l=1:20
          net = fitnet([i,j]);
          init(net);
          net.trainFcn = 'trainlm';

          net = train(net,input',target');
          y_internal(:,l) = net(input')';
     end
     y(:,i*j) = mean(y_internal')';
     index(k,1:2) = [i,j];
     k = k+1;
   end
end

v = vecnorm((repmat(target,1,f_size*s_size)-y))/norm(target);
[min_v, ind] = min(v);
disp(v);
disp(index(ind,1:2));
plot(v);
```

Appendix C

Agent responses to common problems

C.1 Single Application Field Without Initial Training

suggestions	rewards	suggestions	rewards	suggestions	rewards
hot	-1	lap	-1	commander	-1
wife	-1	cape	-1	reserve	-1
pack	-1	glacier	$+1$	command	-1
trailer	-1	freeway	-1	golf	-1
alley	-1	california	$+1$	battle	-1
garage	-1	telescope	-1	bay	$+1$
truck	-1	bay	$+1$	sailing	-1
california	$+1$	success	-1	sea	$+1$
california	$+1$	california	$+1$	island	$+1$
glacier	$+1$	glacier	$+1$	california	$+1$
therapy	-1	island	$+1$	japan	$+1$
sea	$+1$	bay	$+1$	bay	$+1$
island	$+1$	castle	$+1$	glacier	$+1$
bay	$+1$	sea	$+1$	coast	$+1$
traffic	-1	japan	$+1$	sea	$+1$
boundary	-1	lord	-1	castle	$+1$

Table C.1: Agent attempts to correctly suggest ideas for holidays without initial training. Episodes order: left to right, up to down.

suggestions	rewards	suggestions	rewards	suggestions	rewards
”children”	−1	”villain”	−1	”encyclopedia”	−1
”instrument”	−1	”swamp”	−1	”synagogue”	−1
”participate”	−1	”plot”	−1	”conquest”	−1
”perform”	−1	”superman”	−1	”dictionary”	−1
”performance”	−1	”monster”	−1	”jewish”	−1
”practice”	−1	”beetle”	−1	”museum”	−1
”program”	−1	”boulevard”	−1	”medieval”	−1
”saxophone”	+1	”deputy”	−1	”history”	−1
”away”	−1	”keyboard”	+1		
”keyboard”	+1	”drum”	+1		
”drum”	+1	”guitar”	+1		
”beat”	−1	”saxophone”	+1		
”pull”	−1	”trumpet”	+1		
”up”	−1	”cello”	+1		
”shop”	−1	”clarinet”	+1		
”back”	−1	”string”	+1		

Table C.2: Agent attempts to suggest which instrument to play without initial training. Episodes order: left to right, up to down.

C.2 Single Application Field With Initial Training

suggestions	rewards	suggestions	rewards	suggestions	rewards
weapon	-1	crater	-1	asia	$+1$
wave	$+1$	painter	-1	chinese	$+1$
drive	-1	polish	-1	america	$+1$
jesus	-1	astronomy	-1	cape	-1
dominant	-1	geology	-1	japan	$+1$
fbi	-1	asteroid	-1	cinema	-1
funeral	-1	weather	-1	africa	$+1$
opponent	-1	lava	-1	china	$+1$
mountain	$+1$	mountain	$+1$		
japan	$+1$	chinese	$+1$		
sea	$+1$	russia	$+1$		
chinese	$+1$	asia	$+1$		
hop	-1	sea	$+1$		
russia	$+1$	china	$+1$		
asia	$+1$	japan	$+1$		
china	$+1$	coast	$+1$		

Table C.3: The agent, initially trained on Africa, mountain, sea and Italy attempts to suggest holiday destinations. Episodes order: left to right, up to down.

suggestions	rewards	suggestions	rewards
act	-1	saxophone	$+1$
activity	-1	tenor	$+1$
actor	-1	guitar	$+1$
audience	-1	organ	$+1$
ballet	-1	cello	$+1$
band	-1	viola	$+1$
blocks	-1	flute	$+1$
card	-1	trumpet	$+1$

Table C.4: The agent, initially trained on guitar, trumpet, drum and piano attempts to suggest which instrument to play. Episodes order: left to right.

C.3 More Than One Application Fields

suggestions	rewards	suggestions	rewards	suggestions	rewards
”astronomy”	−1	”glacier”	−1	”typhoon”	−1
”carrots”	−1	”flute”	+1	”tropical”	+1
”earthquake”	−1	”keyboard”	+1	”cyclone”	−1
”medal”	−1	”crater”	−1	”depression”	−1
”waves”	−1	”earthquake”	−1	”carrots”	−1
”physics”	−1	”monastery”	−1	”fever”	−1
”greek”	−1	”alto”	+1	”crater”	−1
”tropical”	−1	”chess”	−1	”saxophone”	+1
”orchestra”	+1	”alto”	+1	”glacier”	−1
”conductor”	+1	”flute”	+1	”politician”	−1
”clarinet”	+1	”clarinet”	+1	”alto”	+1
”flute”	+1	”communist”	−1	”moth”	−1
”volcano”	−1	”tenor”	+1	”latin”	−1
”waves”	−1	”saxophone”	+1	”viola”	+1
”tenor”	+1	”trumpet”	+1	”cape”	−1
”snail”	−1	”violin”	+1	”conductor”	+1
”hop”	−1	”alto”	+1	alto”	+1
”jazz”	+1	”trumpet”	+1	violin”	+1
”choir”	+1	”bass”	+1	viola”	+1
”saxophone”	+1	”rhythm”	+1	piano”	+1
”painter”	−1	”drum”	+1	saxophone”	+1
”keyboard”	+1	”saxophone”	+1	clarinet”	+1
”drum”	+1	”tenor”	+1	conductor”	+1
”tenor”	+1	”hop”	−1	flute”	+1

Table C.5: The Q-network trained on the *travel* field is retrained for the domain *instrument to play*. Episodes order: left to right, up to down.

C.4 Visual Embedding Neural Network

Table C.6: List of classes, taken by the Caltech 101 dataset [31], used to train the Visual Embedding Neural Network.

accordion	cup	llama	stegosaurus
airplanes	dalmatian	lobster	sunflower
anchor	dolphin	lotus	tick
ant	dragonfly	mandolin	trilobite
barrel	elephant	mayfly	umbrella
bass	emu	menorah	watch
beaver	euphonium	minaret	wheelchair
bonsai	ewer	nautilus	
brain	ferry	pagoda	
brontosaurus	flamingo	pigeon	
buddha	gramophone	pizza	
butterfly	hawksbill	pyramid	
camera	headphone	revolver	
cannon	hedgehog	rhino	
cellphone	helicopter	rooster	
chair	ibis	saxophone	
chandelier	kangaroo	schooner	
crab	ketch	scorpion	
crayfish	lamp	stapler	
crocodile	laptop	starfish	

Table C.7: List of classes, taken by the Caltech 101 dataset [31], classified without training by the Visual Embedding Neural Network.

binocular
garfield
metronome
octopus
okapi
panda
platypus
scissors
snoopy
strawberry

Bibliography

[1] Aaron van den Oord, Sander Dieleman, Heiga Zeny, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, Koray Kavukcuoglu. Wavenet: A Generative Model for Raw Audio. 2016

[2] Oriol Vinyals, Alexander Toshev, Samy Bengio, Dumitru Erhan. Show and Tell: A Neural Image Caption Generator. 2015

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg & Demis Hassabis. Human-level control through deep reinforcement learning. 2015

[4] Xavier Hinaut, Peter Ford Dominey. Real-Time Parallel Processing of Grammatical Structure in the Fronto-Striatal System: A Recurrent Network Simulation Study Using Reservoir Computing. 2013

[5] Paul Covington, Jay Adams, Emre Sargin. Deep Neural Networks for YouTube Recommendations. 2016

[6] Oxford Dictionary. URL: https://en.oxforddictionaries.com/definition/intuition

[7] Herbert A. Simon. Explaining the Ineffable: AI on the Topics of Intuition, Insight and Inspiration. 1995

[8] Daniel Kahneman.Thinking Fast and Slow. 2011

[9] Leslie Pack Kaelbling, Michael L. Littman, Anrew W. Moore. Reinforcement Learning: A Survey. 1996

[10] DeepMind Web Site: https://deepmind.com/

[11] V.Mnih, K.Kavukcuoglu, D.Silver, A.Graves, I.Antonoglou, D.Wierstra, M.Riedmiller. Human-level control through deep reinforcement learning. 2013

[12] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, TheophaneWeber, Thomas Degris, Ben Coppin. Deep Reinforcement Learning in Large Discrete Action Spaces. 2016

[13] Ivana Kajic, Jan Gosmann, Terrence C. Stewart, Thomas Wennekers, Chris Eliasmith. A Spiking Neuron Model of Word Associations for the Remote Associates Test. 2017

[14] Douglas L. Nelson, Cathy L. Mcevoy, Thomas A. Schreiber. The University of South Florida free association, rhyme, and word fragment norms. 2004

[15] Douglas L. Nelson, Cathy L. McEvoy, Thomas A. Schreiber. The University of South Florida Word Association, Rhyme and Word Fragment Norms. `http://w3.usf.edu/FreeAssociation/`. 1998

[16] Kaoru Nakano. Associatron - A Model of Associative Memory. 1971

[17] Andrew G. Barto, Richard S. Sutton, Peter S. Brouwer. Associative Search Network: A Reinforcement Learning Associative Memory. 1981

[18] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, Richard Harshman. Indexing by Latent Semantic Analysis. 1990

[19] Mark Steyvers Richard M. Shiffrin Douglas L. Nelson. Word Association Spaces for Predicting Semantic Similarity Effects in Episodic Memory. 2004

[20] Tomas Mikolov, Wen-tau Yih, Geoffrey Zweig. Linguistic Regularities in Continuous SpaceWord Representations. 2013

[21] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space.2013

[22] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. 2013

[23] Y. Bengio, R. Ducharme, P. Vincent. A neural probabilistic language model. 2003

[24] Tomas Mikolov, Martin Karafiat, Jan Cernocky, and Sanjeev Khudanpur. Recurrent neural network based language model. 2010

[25] Marco Baroni, Georgiana Dinu, German Kruszewski. Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors. 2014

[26] Andrea Frome, Greg S. Corrado, Jonathon Shlens, Samy Bengio, Jeffrey Dean, Marc' Aurelio Ranzato, Tomas Mikolov. DeViSE: A Deep Visual-Semantic Embedding Model. 2013

[27] Mohammad Norouzi, Tomas Mikolov, Samy Bengio, Yoram Singer, Jonathon

Shlens, Andrea Frome, Greg S. Corrado, Jeffrey Dean. Zero-Shot Learning by Convex Combination of Semantic Embeddings. 2013

[28] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. 2012.

[29] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich. Going Deeper With Convolutions. 2015.

[30] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. 2009

[31] L. Fei-Fei, R. Fergus and P. Perona. Learning generative visual models from few training examples: an incremental Bayesian approach tested on 101 object categories. IEEE. CVPR 2004, Workshop on Generative-Model Based Vision. 2004

[32] Griffin, G. Holub, AD. Perona, P. The Caltech 256. Caltech Technical Report.

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition. 2015

[34] Qun Li, Student Member, IEEE, Honggang Zhang, Senior Member, IEEE, Jun Guo, Bir Bhanu, Fellow, IEEE, and Le An. Reference-Based Scheme Combined With K-SVD for Scene Image Categorization. 2013

[35] Liefeng Bo, Xiaofeng Ren, Dieter Fox. Multipath Sparse Coding Using Hierarchical Matching Pursuit. 2013

[36] Matthew D. Zeiler, Rob Fergus. Visualizing and Understanding Convolutional Networks. 2013

[37] P. Bojanowski, E. Grave, A. Joulin, T. Mikolov. Enriching Word Vectors with Subword Information. 2017

[38] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. GloVe: Global Vectors for Word Representation. 2014

[39] Guang-Bin Huang. Learning Capability and Storage Capacity of Two-Hidden-Layer Feedforward Networks. 2003

[40] D. Stathakis. How many hidden layers and nodes? 2008

Acknowledgments

Un ringraziamento sincero alla mia relatrice, per avermi coinvolto in questo progetto, per essere rimasta sempre positiva durante ogni fase della ricerca e per avermi aiutato a trovare la strada. Così a tutto il VLSI Lab per avermi ospitato e per avermi fatto sentire parte del team. Alla mia famiglia, sempre presente nonostante la distanza, per non aver mai smesso di credere in me, rendendo tutto questo possibile. Ai miei zii, che mi hanno accolto qui a Torino come un figlio. A Nicola, per aver condiviso ogni nanometro di questo percorso, a Riccardo, per avermi convinto di potercela fare e a Francesco, per non aver mai perso una chiamata. E a tutti gli altri amici, vecchi, nuovi e ritrovati. Per concludere un grazie di cuore a Camilla, il meglio arriva sempre alla fine, come il dolce a tavola. Mi hai sempre spinto a dare di più e mi hai fatto sentire a casa. Sei uno spettacolo.