Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Learning Robot Task Planning Primitives by means of Long Short-Term Memory

Student
Federico Vendramin

Advisor
Dr. Stefano Ghidoni

Co-Advisor
Dr.ssa Elisa Tosello

16 Aprile 2018
Anno Accademico 2017/2018

If you always do what you've always done, you'll always get what you've always got.

Henry Ford

Abstract

This work is an attempt to create a robot task planner by exploiting increasingly popular Deep Neural Networks. The aim is to learn how to achieve a robotic manipulation task by selecting the appropriate action to perform, along with its arguments, by observing the robot workspace.

This work proposes a model that uses Long Short-Term Memory, along with an expert policy that is able to generate an artificial dataset used for training. The network learns sequences composed by robotics action primitives like picking and placing objects. These sequences aim to achieve a specific task, in this work we chose swapping the positions of two cubes.

The network's ability to learn this kind of tasks using an artificial dataset as well as the ability to generalize to unseen cases is questioned and explored in this work. Accuracy of action selection up to 97% was obtained on sequences of the same length of the one used during training and up to 91% on longer task sequences.

Sommario

Questo progetto è un tentativo di creazione di un task planner tramite l'utilizzo delle sempre più popolari Deep Neural Network. L'obiettivo è quello di eseguire un task di manipolazione tramite robot selezionando l'azione appropriata da svolgere, con i relativi argomenti, basandosi sull'osservazione dell'ambiente di lavoro del robot.

Questa tesi propone un modello che utilizza le Long Short-Term Memory, insieme ad una expert policy in grado di generare un dataset artificiale per allenamento. La rete apprende sequenze composte da azioni primitive robotiche, come il pick e il place di oggetti. Queste sequenze hanno la finalità di compiere uno specifico task, in questo progetto il task consiste nell'invertire le posizioni di due cubi.

In questa tesi ci si interroga sulle capacità delle reti di apprendere questo tipo di task tramite un dataset artificiale, ovvero le capacità di generalizzare in presenza di casi nuovi. Una precisione nella selezione delle azioni del 97% è stata ottenuta su sequenze della stessa lunghezza di quelle utilizzate per il training e fino al 91% su sequenze di lunghezza superiore.

Contents

Chapter 1

Introduction

The ability to reason about an either partial or complete observation of the environment is very useful for an autonomous robot. This ability requires the robot to understand the environment and allows it to choose the right action to perform to accomplish a previously instructed task.

Planning is an essential ability needed by a robot to carry out tasks: a rover navigating and exploring an unknown environment while avoiding obstacles, a robotic arm grasping an object, a humanoid robot standing and walking. These are just some examples of tasks that need planning. This concept can be broken down into more specific ones which are **Task Planning** and **Motion Planning**.

> *"Task Planning is the process of generating a discrete sequence of actions that are required to achieve a desired task."*

Suppose the assignment of a task T to a robot R. A task planner TP : $(s_0, s_G, A) \rightarrow p$ aims to find a plan $p \in P$ solving T. p moves R from its start state $s_0 \in S$ to a goal state $s_G \in S$ by combining the set of actions A that R is able to perform according to its capabilities.

Each action is defined as a sentence with a set precon$(a) = \{precon_0(a), ...,$ $precon_N(a)\}$ of preconditions and a set effect$(a) = \{effect_0(a), ..., effect_M(a)\}$ of effects, described as conjunctive lists of literals in first-order logic.

TP computes a set of plans P where $p \in P$ is defined as

$$p =< s_0, a_0, ..., s_{N-1}, a_{N-1}, s_N >, \quad s_N = s_G \tag{1.1}$$

and $(s_i, a_i) \rightarrow s_{i+1}$ iff precon(a_i) is satisfied by s_i and effect(a_i) brings to s_{i+1}.

> *"Motion Planning is the process of generating collision-free trajectories to reach a desired position."*

A motion planner MP: $(s_0, s_G, A) \rightarrow t$ tries to find a path $t \in \tau$ that lets R move from $s_0 \in S$ to $s_G \in S$ while avoiding collisions. The problem is deterministic if the working space is fully observable. MP can find a set of paths τ, where $t \in \tau$ is a path in the free space:

$$\tau : [0,1] \rightarrow C_{free}, \quad \tau(0) = s_0, \quad \tau(1) = s_G \tag{1.2}$$

Historically Task Planning and Motion Planning have always been treated as two separate, different problems in a stacked, hierarchical architecture. Both have been exclusively working on their domain: the task planner works in a discrete, often hierarchical domain based on preconditions and effects, while the motion planner works in a continuous, geometric domain called *Configuration Space*.
A trending research topic in robotics right now is called **Combined Task and Motion Planning**, often shortened to CTMP or TMP. The main challenge of TMP is to let go the standard hierarchical approach where the task and motion are approached as two different problem, one on top of the other. This is usually done by developing a planner that is capable of generating task and motion plan without treating their domains independently.
Suppose the existence of a Task and a Motion Planner. Suppose that T is assigned to R. TMP: $(s_0, s_G, A) \rightarrow t$ finds the plan $p \in P$ performing T and returns the trajectory $t \in T$ executing p.

The Task Motion Kit [1] (TMKit) is a framework for TMP based on *Satisfiability Modulo Theories* (SMT) and was used as a starting point for this work. A more in depth description of this framework is given in Chapter 2. The fundamental idea of TMKit is to leverage the incremental solving of a SMT solver, like Microsoft Z3 [2] library, for task planning. In this way it is able to efficiently add and remove constraint. These constraint regards motion feasibility and come from an off the shelf motion planner, like the sampling-based planners in the **Open Motion Planning Library** [3] (OMPL).

1.1 Motivations

The main limitation when approaching the TMP problem like TMKit or other similar frameworks do is the incapability of handling motion failures, like an object slipping from the robot's hand, or other type of unexpected changes in the robot environment, like a person moving, that are typical of real world situations. This is due because a full plan is computed using a geometric domain before executing it in real time using a simulator or a real robot. This type of approaches are not capable of adapting to environment changes to achieve their goal.

At the same time, the rise in popularity of **Deep Learning** (DL), a subset of Machine Learning inspired by the human brain, is giving increasingly positive results in very different fields. Its popularity has been fueled by the higher GPU computing power, that allows matrices operations used during the training to be computed with less time than ever before.
Deep Learning is different than standard Machine Learning algorithms that are very specific and do not generalize enough while also requiring features to be hand-engineered. DL is able to learn which features are important by it self, by observing the input data. It also features the ability to train end to end, i.e. learning internal parameter to reflect a specific behavior shown through a high number of demonstrations.
This technique has given researchers a new powerful tool that can be applied in many fields like, but not restricted to, Computer Vision, Natural Language Processing, Recommender Systems and Robotics. The results obtained so far are impressive especially in solving specific tasks.

In this work we try to bring together some concepts of two fields, Task Motion Planning and Artificial Intelligence. Can neural networks be exploited to learn a task and produce an online planner by selecting the correct action, along with its parameters, to be performed? Will this planner be able to avoid computing the entire plan beforehand so that it can react to changes in the environment or unexpected failures? Will it be able to learn how to change the execution of the task acknowledging motion-related failures?

1.2 Challenges

The main challenges and goals that this work addresses are reported in this section.

- **Online Planning and Executing**: the main goal is to use neural networks as main building block to explore the feasibility of an online task planner. The ability of the network to reason about the environment and react to unexpected changes is questioned.

- **Learning**: since neural networks are trained end to end, we question the ability to learn with supervision how to predict the next action to achieve a specific task.

- **Dataset**: a public dataset to use for training the network is lacking and collecting this type of data can be very resource expensive.

1.3 Contributions

This work provides a study of **Recurrent Neural Networks** (RNNs), **Long Short-Term Memory** (LSTM) networks. It is also an attempt to exploit this kind of networks to produce an agent able to learn how to reason about environment observations. From this observations it should be able to produce a sequence of actions that fulfill a task. This sequence can be seen as a program that invokes primitive functions, or APIs, and also provides them with parameters.

We will focus on a simple robotic manipulation task with a low number of objects carried out by a robotic arm with a gripper. The experiments were executed using an artificial environment generated by an expert policy. This policy produces a high-level representation of the environment composed, for example, from the position of the objects, the status of the gripper indicating if full or empty, and so on. This work can be used with either a physics simulator or in real world by adding the necessary encodings that extract the relevant high-level feature needed from sensors data. The most common example is an object detector using a robot's camera to estimate the position of a specific object.

- **Online Planning and Execution**: the proposed model is able to detect some robotic manipulation failures like an unsuccesful grasp or a wrong placing of an object. It is also able to react to these action failures, that can modify preconditions of the failed action, by selecting the actions that restore the right preconditions.

- **Learning**: accuracy of up to 97% has been reached when evaluating the network on sequences of the same length of those used for training, with length $maxlen = 30$. Accuracy of up to 91% has been reached while evaluating the network on sequences of unseen, higher length of $maxlen = 100$.

- **Dataset**: an artificial dataset generator is proposed. It is able to produce the training examples used to train the network. The issues that arise by using this approach are also analysed.

1.4 Structure of the thesis

The remainder of the thesis is organized as follows: Chapter 2 describes the current state of the art and gives a more in depth explanation of the frameworks and publications used for this work. Moreover, possible applications of Deep Neural Networks to Task Planning and Task-Motion Planning are introduced as well as the concept of meta-learning.
In Chapter 3 an overview of Neural Networks and Deep Learning will be presented along with Recurrent Neural Networks and Long Short-Term Memory networks.
In Chapter 4 the experiments of this thesis will be presented and discussed and in Chapter 5 conclusions will be reported along with possible future works.

Chapter 2

State of the art

There are many different strands that solve the task planning problem, most of them are evolution of **STRIPS** [4], the **ST**anford **R**esearch **I**nstitute **P**roblem **S**olver, an automatic planner developed in 1971 by Fikes and Nilsson at Stanford. The task of the problem solver is to find some composition of operators that transforms an initial configuration of the world into the goal configuration.
Since those years one of the biggest difficulties has been how to model the world surrounding the robot. Traditional task planning relies on a representation of the task domain using a formal task language composed of a variety of notations and logics, like the **Planning Domain Definition Language** [5] (PDDL). This language is used to define the initial and goal states and parameterized actions with preconditions and effects based on first-order logic. Common techniques used to solve the task planning problems are heuristic search [6] and constraint satisfaction [7].

At the same time, a lot of research around motion planning is present. MP has as its main goals to compute collision-free trajectories and to make a robot reach the goal location as fast as possible. The problem of motion planning can be stated as the problem of finding a path that moves the robot gradually from start to goal without colliding with anything. If this is not possible the task planner does not make anything more than stating "A valid path can exists but *I* couldn't find one".
Sampling-based motion planners are widely used for high dimensional systems [8]. Such sampling-based planners offer *probabilistic completeness*, guaranteeing that the planner will eventually find a solution if one exists. However, if a solution does not exists, the planner cannot prove this. In this case the planner runs until a timeout occurs.

Prior work in TMP includes applying geometric constraints to limit the motion planning space or prove motion infeasibility [9]. Hierarchical Planning in the Now (HPN) [10] interleaves planning and execution, reducing search depth but requiring reversible actions when backtracking. Combining symbolic and geo-metric planning to synthesize human-aware plans [11] extends a hierarchical task planner with geometric primitves, using shared literals that relate task-level symbols with motion-level geometric entities. Combined task and motion planning through an extensible planner-independent interface layer [12] interfaces an off the-shelf task planner and motion planning using a herustic to remove objects that potentially block the robot's path. Other works like [13] formulate the motion side of TMP as a constraint satisfaction problem over a discretized, preprocessed subset of the configuration space. The Robosynth framework [14] uses a Satisfiability Modulo Theories (SMT) solver to generate task and motion plans from a static roadmap, employing plan outlines to guide the planning process. FFRob [15] develops an FF-like [6] task layer herustic based on a lazily-expanded roadmap.

2.1 The Task-Motion Kit

The **Task-Motion Kit** [1] is an end-to-end system developed at the *Kavraki Lab* of Houston for probabilistically complete task-motion planning and real-time execution enabling the coupling of task planning, motion planning, and real time estimation and control. This framework has been the starting point in this work.

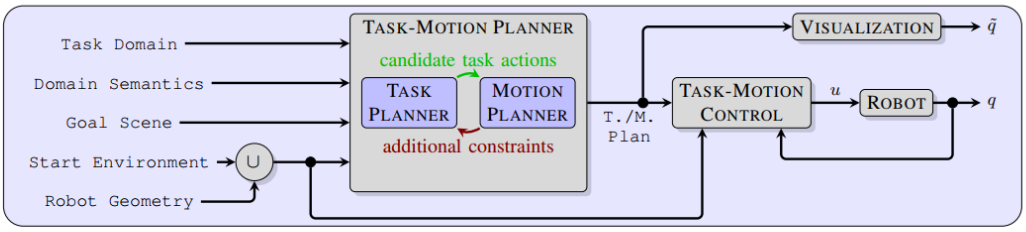


Figure 2.1: High-Level Planning and Execution Block Diagram. The inputs are the task domain definition, the environment and robot geometries, combined to produce the scene graph, and the domain semantics that relate the task and motion layers. The Task–Motion Planner generates a plan based on these inputs. The Task–Motion Control layer executes the plan, sharing a geometric representation, the scene graph, with the planning layer

TMKit follows the high-level design shown in Fig. 2.1 and it is able to compute a discrete sequence of actions along with their collision-free motion plans by alternating task and motion planning.
The following inputs are needed to compute a plan (See Fig. 2.2):

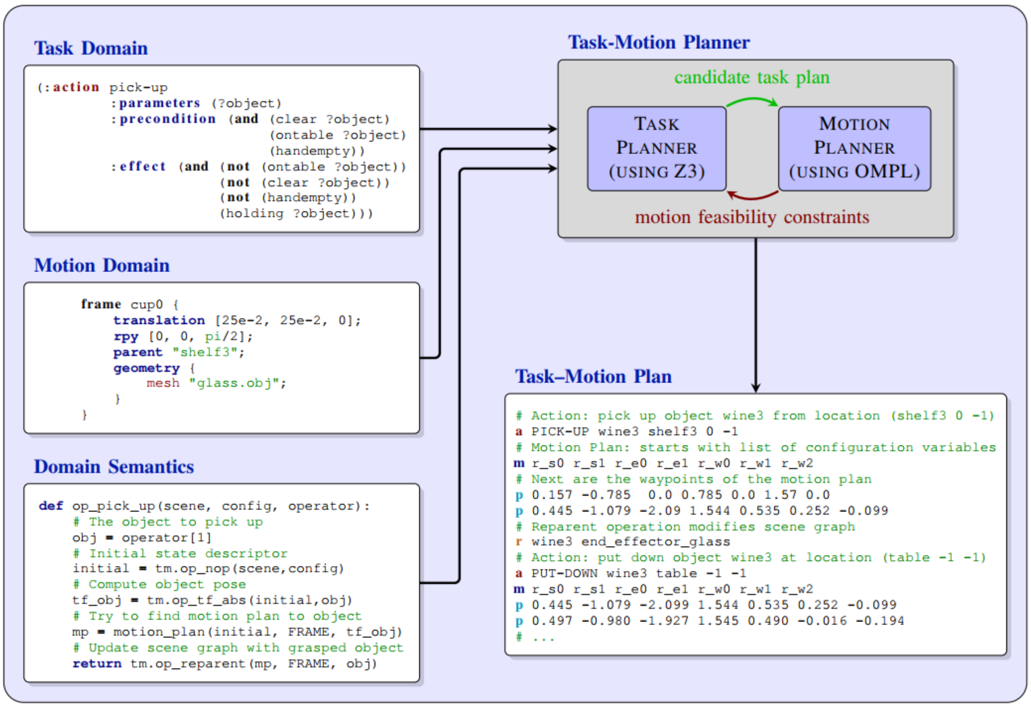
- **Task Domain**: Defines the discrete actions that the robot can perform including their *preconditions* and *effects*. As an example a *Pick-Up* action could have as precondition that the object to be picked up has to be on a table and that the robot's gripper has to be empty. It could also have the occupied gripper as an effect to that action.

- **Motion Domain**: Defines the 3D poses of the objects inside the robot's environment, its kinematic structure, its geometry and object meshes in the working environment. The robot and the environment, together, take the name of *Scene*.

- **Domain Semantics**: Defines the relationships between the task domain and the motion domain in such a way that it represents specific geometric configurations at the task level: as an example we can consider when grasping an object, a high level, discrete variable called *holding(obj)* should be set to *True* (Preconditions). In the other way around relationships from task to motion happens by executing the selected actions (Effects). These effects will in fact modify the predicates that will be used as preconditions for the successive actions.

In particular TMKIT present a novel algorithm for combined task and motion planning called **Iteratively Deepened Task and Motion Planning** (IDTMP) that uses a constraint-based task planner to compute different candidate plans (Fig. 2.3).
The task planner relies on an incremental **Satisfiability Modulo Theories** [16] [17] (SMT) solver that is capable of efficiently generating alternate task plans. Incremental SMT solvers maintain a stack of constraints or assertions and can efficiently perform repeated satisfiability checks as constraints are pushed onto and popped from the constraint stack. TMKit uses as SMT solver **Z3** [2] the standard the facto library developed by Microsoft.

The motion planner is basically composed by a variety of sampling-based motion planners through the **Open Motion Planning Library** [3] (OMPL) in synergy with the **Flexible Collision Library** [18] (FCL) for collision checkings.



```
(:action pick-up
        :parameters (?object)
        :precondition (and (clear ?object)
                           (ontable ?object)
                           (handempty))
        :effect (and (not (ontable ?object))
                     (not (clear ?object))
                     (not (handempty))
                     (holding ?object)))
```

```
    frame cup0 {
        translation [25e-2, 25e-2, 0];
        rpy [0, 0, pi/2];
        parent "shelf3";
        geometry {
            mesh "glass.obj";
        }
    }
```

```
def op_pick_up(scene, config, operator):
    # The object to pick up
    obj = operator[1]
    # Initial state descriptor
    initial = tm.op_nop(scene,config)
    # Compute object pose
    tf_obj = tm.op_tf_abs(initial,obj)
    # Try to find motion plan to object
    mp = motion_plan(initial, FRAME, tf_obj)
    # Update scene graph with grasped object
    return tm.op_reparent(mp, FRAME, obj)
```

```
# Action: pick up object wine3 from location (shelf3 0 -1)
a PICK-UP wine3 shelf3 0 -1
# Motion Plan: starts with list of configuration variables
m r_s0 r_s1 r_e0 r_e1 r_w0 r_w1 r_w2
# Next are the waypoints of the motion plan
p 0.157 -0.785  0.0 0.785 0.0 1.57 0.0
p 0.445 -1.079 -2.09 1.544 0.535 0.252 -0.099
# Reparent operation modifies scene graph
r wine3 end_effector_glass
# Action: put down object wine3 at location (table -1 -1)
a PUT-DOWN wine3 table -1 -1
m r_s0 r_s1 r_e0 r_e1 r_w0 r_w1 r_w2
p 0.445 -1.079 -2.099 1.544 0.535 0.252 -0.099
p 0.497 -0.980 -1.927 1.545 0.490 -0.016 -0.194
# ...
```

Figure 2.2: Task-Motion Planner Implementation Diagram, showing fragments of the planner's input: the task domain, domain semantics, motion domain. As well as the output consisting of the generated task-motion plan

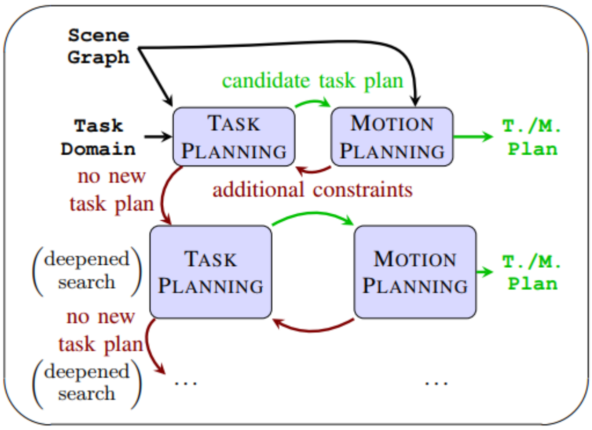


Figure 2.3: Diagram of IDTMP. Incrementally incorporate motion feasibility information into the task planner via incremental constraint solving

A diagram of TMKit's main components and internals is shown in Fig. 2.4.

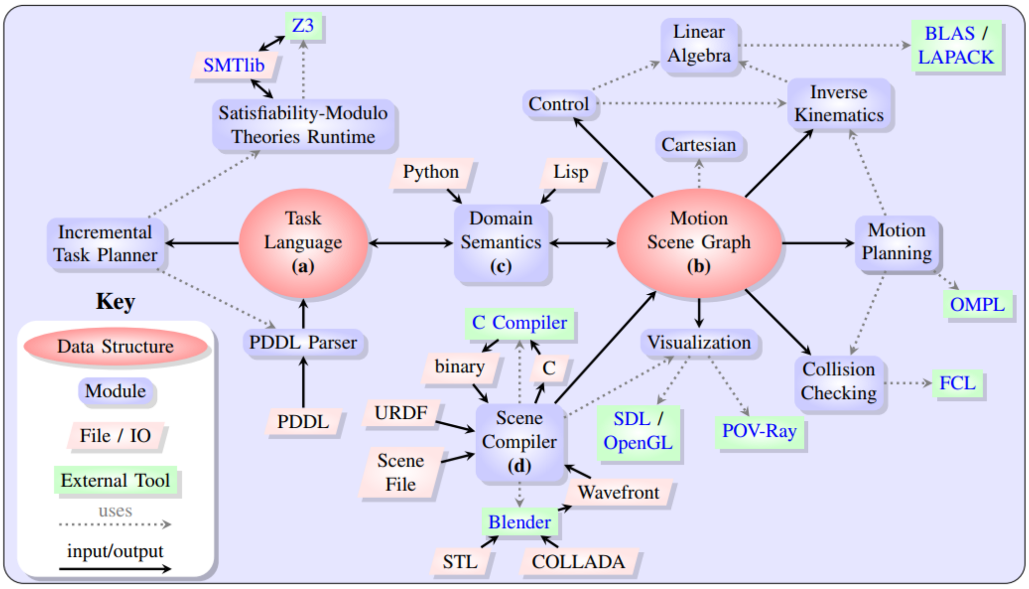


Figure 2.4: Map of software components. The key data structures are (a) the task language and (b) the scene graph. These data structures are connected by (c) the domain semantics definitions. (d) The scene compiler is used for efficient visualization

2.2 Neural Network for Task Planning

Due to the rising popularity and effectiveness of deep learning in the last couple of years novel techniques based on neural networks have been experimented in a plethora of different fields. Combined task and motion planning is among those fields even though researchers are either publishing papers from a robotics point of view or from the Artificial Intelligence one.
In the past applying Artificial Intelligence to Robotics seemed an overly complex problem. Therefore, the two branches was separated: robotics became *"a body without a brain"* while AI became *"a brain without a body"*. Nowadays we are going towards the embodiment of AI back into Robotics since both fields are beginning to be mature enough to face the challenge.
Some related works like the **Neural Programmer-Interpreter** [19] (NPI), although from a pure AI point of view, seems to point to this direction even though it is not strictly connected to robotics nor it is embodied in any machine. This work has become really popular among researchers in the field

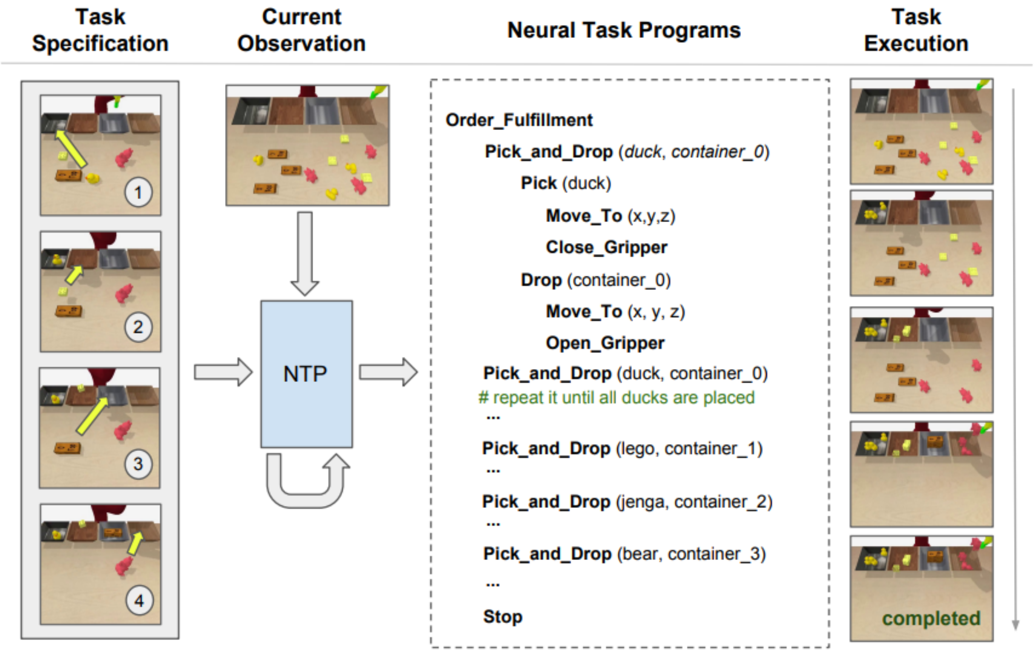
and is inspiring more and more research in this direction.
NPI is a recurrent and compositional neural network that learns to represent and execute programs. It has three learnable components: a task-agnostic recurrent core, a persistent key-value program memory, and domain specific encoders that enable a single NPI to operate in multiple perceptually diverse environments with distinct affordances.
The core of NPI is a LSTM network that at each timestep i selects the next program to run conditioned on the current observation o_i of the environment.
By learning to compose lower-level programs to express higher-level programs, NPI reduces sample complexity and increases generalization ability compared to sequence-to-sequence LSTMs.
NPI is trained with fully supervised execution traces, learning from a small number of rich samples.

Another useful work was done in [20] that selects subgoals using Deep Learning in *Minecraft*. The main concept is based on the interleaving of perception, goal, reasoning, and acting with the observations of the environment given by the images seen from the in-game character's camera.
Basic motion primitives have been implemented inside the Minecraft game such as looking, moving, jumping, and placing or destroying blocks. These motion primitives compose the operational plans for the four sub-goals: walking forward, creating stairs, removing obstacles, and bridging obstacles. The selection of the subgoal is done by a CNN trained on the environment observations and accuracy up to 87.1% was achieved.

Other approaches use Deep Reinforcement Learning like [21] that learns both low-level control policies and high-level action selection. It then use these multi-level policies as part of a heuristic search algorithm to achieve a complex task like asking a vehicle to drive down a road in traffic, avoid collisions, and navigate an intersection, all while obeying given rules of the road. The work proposed in [21] investigates the ability of neural networks to learn both linear temporal logic constraints and control policies in order to generate task plans in complex environments.

2.3 Meta-Learning

Since it's not always possible to collect a large dataset to train the models, different ways to avoid this necessity are being investigated, one of the most



```
Order_Fulfillment
    Pick_and_Drop (duck, container_0)
        Pick (duck)
            Move_To (x,y,z)
            Close_Gripper
        Drop (container_0)
            Move_To (x, y, z)
            Open_Gripper
    Pick_and_Drop (duck, container_0)
    # repeat it until all ducks are placed
    ...
    Pick_and_Drop (lego, container_1)
    ...
    Pick_and_Drop (jenga, container_2)
    ...
    Pick_and_Drop (bear, container_3)
    ...
    Stop
```

Figure 2.5: An order fulfillment task that illustrates the pipeline of the model. The meta-learning framework learns to instantiate a neural program from a task specification. In this example, the goal is to move all objects from the same category into the corresponding shipping container stated in the task specification.

interesting is called *meta-learning* where a first network, called the meta-learner, trains another network, called the *learner* [22, 23] .
The aim of meta-learning is to achieve better generalization and to develop a more efficient learning where the model does not need to be trained on a high number of examples but rather on a few, even just one, from which it should be able to learn the correct behavior.

The **Neural Task Programming** [24] (NTP) is a robot learning framework based on the work proposed in NPI. This tool was used with robotic manipulators for tasks like block stacking and order fulfillment, where the goal is to transport all objects from each category into a specified shipping container (Fig. 2.5).
The key idea of this work is to learn reusable representations shared across tasks and domains. NTP interprets a task specification (Fig. 2.5 left) and instantiates a hierarchical policy as a neural program (Fig. 2.5 middle), where bottom-level programs are primitive actions that are executable in the environment. Each program call takes as input the environment observation and

a task specification, producing the next sub-program and a corresponding sub-task specification.

The model executes the program to accomplish the task described in the task specification. The lowest level of the hierarchy is made of symbolic actions executed by robot API.

Even more interesting is the fact that NTP bridges the idea of few-shot learning from demonstration [25] and neural program induction by learning to execute a task from a single demonstration. It is of course necessary to train using a high number, possibly infinite, of different tasks to achieve strong task generalization.

Chapter 3

Deep Learning

DL [26] is a subset of ML that uses neural networks to learn to recognize common patterns from data. The idea behind it date back to 1943 when neurophysiologist Warren McCulloch and mathematician Walter Pitts wrote a paper [27] on how neurons might work. They then created a simple neural network model using electrical circuits.

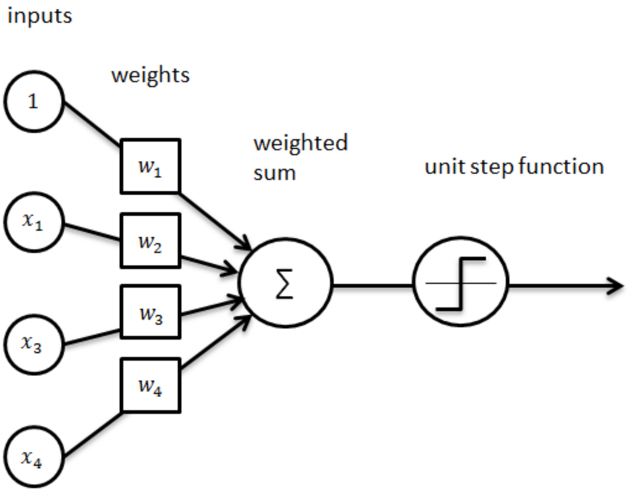


Figure 3.1: The Perceptron Model

Source: http://www.ataspinar.com/

Based on the work of McCulloch and Pitts, in 1957 psychologist Frank Rosenblatt created the *Perceptron* [28] (Fig. 3.1). It is a simplified math-

ematical model of how a neuron works, taking a set of binary inputs, multiplying each input by a weight and thresholding the sum of these weighted inputs to a output equal to 1 if the sum is above a certain level, to 0 otherwise. As computers technology grew, the first simulations of neural networks started to take place giving exciting results: in 1959 Bernard Widrow and Marcian Hoff of Stanford developed a model capable of recognize binary patterns to predict the next bit from a stream (ADALINE [29]).
After this years, researchers settled on the fact that networks with multiple layers of perceptrons could not be achieved; it was also shown that it was theoretically impossible for a perceptron to learn the XOR function, this brought neural networks to their first 'winter'.

During the 80s the interest in the field of DL was renewed due to the rise of multilayer neural networks and during these years the Backpropagation algorithm was developed. This algorithm allows to propagate the error from a layer to the previous one starting from the output until the input layer and therefore update the weights and biases of the network to learn a specific behavior.
The limited computational power of the CPUs available in those years, that made the networks training times too long, along with the lack of public datasets to use is what held back the development of this technology.

In recent years multilayer neural networks or deep networks (and from this the name Deep Learning is used) have seen, once again, a renovated excitement. This was possible by many factors, among those the most important were: the increasing appearance of large high quality labeled dataset, also known as "Big Data", used for training and the availability of powerful parallel computing GPUs that lowered the training time by a huge margin. Since the introduction of these new technologies, DL has had an amazing progress with a lot of new different network and, lately, compositions of those networks, proposed by many scientists from all around the world.

3.1 Artificial Neural Networks

An Artificial Neural Network (ANN) is a computational model inspired by the brain capable of learning to reflect a specific behavior by observing labeled data. Neural Networks are composed by nodes, called neurons, interconnected between each other and that can be gathered together into groups called layers, as depicted in Fig. 3.2. Given an input set of values X the NN computes the output values Y by applying a series of operations from left to

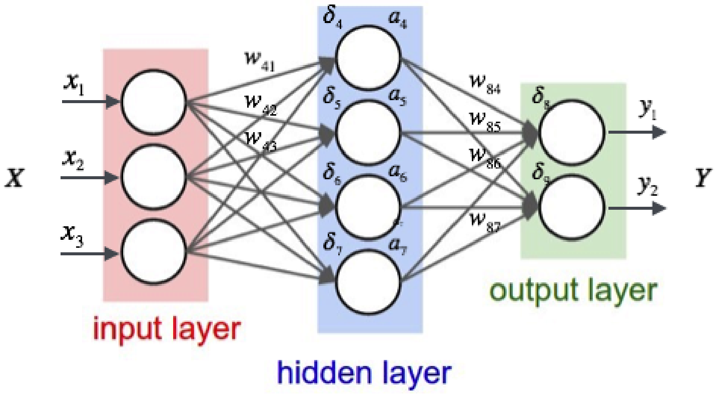


Figure 3.2: Neural Network architecture

Source: http://cs231n.github.io/neural-networks-1/

right, hence why this kind of networks are called **Feedforward Networks**. The neuron, which is the basic element of computation, can be modeled as:

$$output = \sigma(w \cdot x + b) \tag{3.1}$$

Where σ is called sigmoid function, and is defined by:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3.2}$$

This can be rewritten in a less compact, but more intuitive form:

$$\frac{1}{1 + e^{-\sum\limits_j w_j x_j - b}} \tag{3.3}$$

In simple words each neuron's input x_i is multiplied by its weight w_i and summed along with the bias b. Then the sigmoid function is applied to obtain a normalized value between 0 and 1.

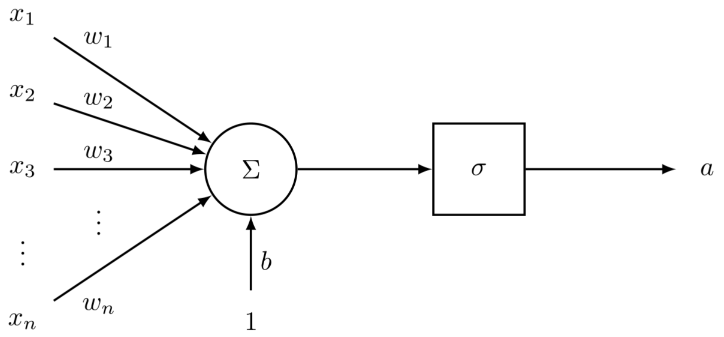


Figure 3.3: The Neuron model

Source: http://diffsharp.github.io

The most important thing to observe is that the sigmoid function is differentiable, meaning that we can compute the gradient that tells us how the output changes in relation to the inputs. By exploiting this we will be able to iteratively update the weights and biases to reflect the desired behavior, specifically by minimizing a cost function of the output obtained by feeding examples to the network: this is in fact how the network learns.
Since a lot of iterations are needed to achieve good results, existance of a dataset with enough examples is of fundamental importance. The most interesting algorithm that does this parameters' update after computing the gradients is called Gradient Descent, discussed next.

3.1.1 Gradient Descent

Gradient Descent (GD) is a iterative algorithm for finding the minimum of a function, in particular it lets us find weights and biases. To this purpose we need a *cost function*, for example the Mean Squared Error (MSE):

$$C(\theta) = C(w, b) = \frac{1}{2n} \sum_{i=1}^{n} (f_\theta(x^i) - \hat{y}^i)^2 \tag{3.4}$$

Where θ denotes all ours parameters in the network, such as the weights w and the biases b, n is the total number of training inputs, $\hat{y}^i$ is the i-th desired output value while $f(x^i)$ is the one predicted by the network. We can observe that the cost $C(w, b)$ is non-negative, since every term in the sum is non-negative. Furthermore, $C(w, b) \approx 0$ when $f(x^i) \approx \hat{y}^i$ for all training inputs x.

Since we have defined a cost function we are now able to compute its gradient, the vector of partial derivatives:

$$\nabla C = (\frac{\partial C}{\partial \theta_1}, ..., \frac{\partial C}{\partial \theta_m})^T \tag{3.5}$$

Observing that the variation of the cost function is given by:

$$\Delta C \approx \sum_{i=1}^{m} \frac{\partial C}{\partial \theta_i} \Delta \theta_i \tag{3.6}$$

We can rewrite it in a more compact way using the gradient vector:

$$\Delta \approx \nabla C \cdot \Delta \theta \tag{3.7}$$

we now have a formula that describes how the cost function changes as the parameters θ change. Since we want to minimize the cost, we want ΔC to be negative. To achieve this we choose:

$$\Delta \theta = -\alpha \Delta C \tag{3.8}$$

Where α is a small, positive parameter known as **Learning Rate**. By substituting the equation above we obtain:

$$\Delta C \approx \ \alpha \Delta C \cdot \Delta C = \ \alpha \|\Delta C\|^2 \tag{3.9}$$

That guarantess that ΔC is always decreasing and so our learning process will not diverge. By repeatingly updating the parameters θ by the amount $-\alpha\Delta C$ we can make the network learn. We can graphically represent the gradient descent using a trivial case with θ of just one dimension, so we only have a single parameter to be trained, as in Fig. 3.4.

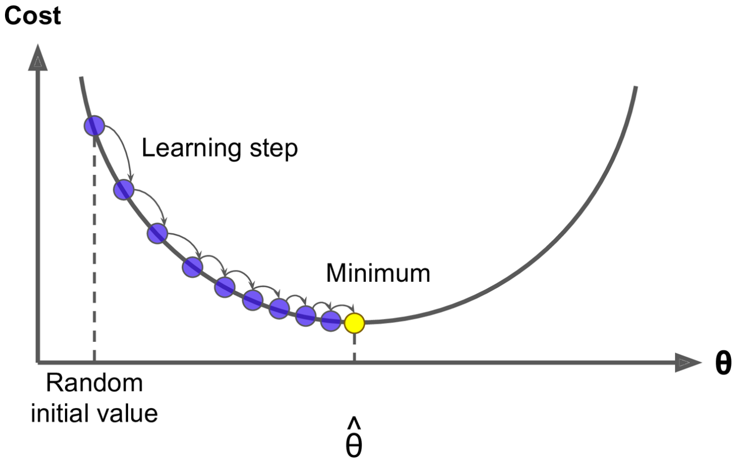
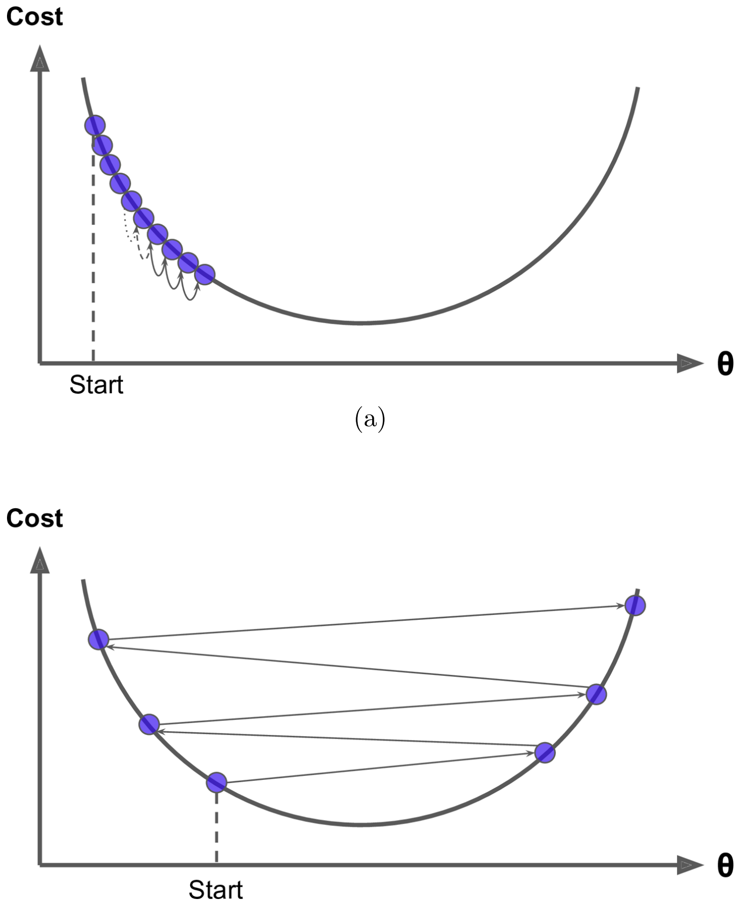


Figure 3.4: Visual representation of 1-dimensional Gradient Descent

Source: https://www.safaribooksonline.com/

We can observe that the learning rate (or step) decreases as it approaches the minimum point of convergence. This is because the gradient of the cost function ΔC decreases, since it represents the slope of the curve. Therefore, there is no need to adjust the learning rate α through iterations.
Even though α does not need adjustment during training it needs to be set at a reasonable value. If α is too little a lot of iterations are needed to get to the minimum and the algorithm become very inefficient. On the other hand setting α too large will result in the divergence of the algorithm.



(a)

(b)

Figure 3.5: Hyperparameter α too small (3.5a) and too large (3.5b)

3.1.2 Deep Neural Networks

After a brief introduction to NNs and gradient descent we can finally introduce Deep Neural Networks. The evolution from the basic, shallow network is based on a rather simple concept: the introduction of more hidden layers in cascade, also called stacked layers, between the input and output layers. The novelty is the capacity for networks to learn abstraction, recognizing low level patterns, such as edges, and build up new, higher level features at each layer to obtain a hierarchical learning.

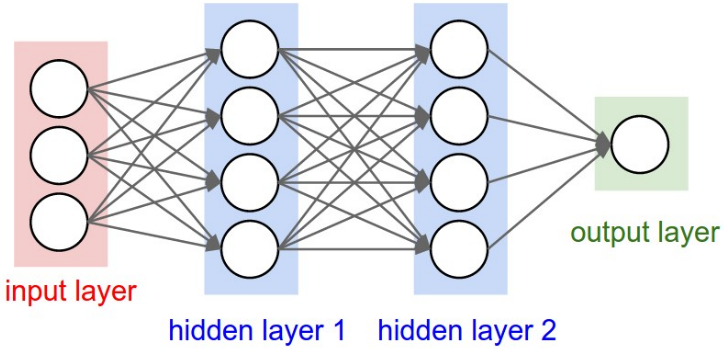


Figure 3.6: A simple Deep Netork with 2 hidden layers

Source: http://cs231n.github.io/neural-networks-1/

To sum up we can call Deep Neural Network every network composed by two or more hidden layers (Fig. 3.6).

3.1.3 Backpropagation

In paragraph 3.1.1 Gradient Descent was presented as a way to update the weights and biases of a feed forward neural network. In this paragraph, instead, how to compute the gradient vector ∇C through the Backpropagation algorithm will be explained.
This algorithm is of extreme importance, in particular for deep networks, since it allows to compute each partial derivative $\frac{\partial C}{\partial \theta_i}$ that composes ΔC starting from the output layer. We can apply this algorithm to every intermediate layer, as the name suggested, by back propagating a quantity tied to the partial derivatives, called error, from the successive layer to the current. We can denote the error in the j^{th} neuron in the l^{th} layer as δ^l_j and the weighted input of the same neuron as z^l_j so its output is $\sigma(z^l_j + \Delta z^l_j)$ and causing the overall cost to change by an amount $\frac{\partial C}{\partial z^l_j}\Delta z^l_j$.
We can define the error δ^l_j as:

$$\delta^l_j = \frac{\partial C}{\partial z^l_j} \tag{3.10}$$

The next four equations describe the actual Backpropagation algorithm in mathematical terms, based on the previous intuitions.
The first equation lets us compute the error δ_j^L of the output layer L:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \tag{3.11}$$

This is given by the multiplication of how fast the cost is changing as a function of the j^{th} output activation and how fast the activation function σ is changing at z_j^L.
The next equation is the core of the backpropagation and is in fact applied to all the hidden layers in the network. It lets us compute the error δ^l as a function of the error in the next layer δ^{l+1}. $(w^{l+1})^T$ is the transpose of the weight matrix w^{l+1} for the $(l+1)^{th}$ layer. This can be interpreted as backpropagating the error from the next layer l^{th+1} to the output of the current l^{th} layer and then again through the activation function in the same layer. The error δ^l in the weighted input to layer l is obtained in this way.

$$\delta^l = ((w^{l+1})^T\delta^{l+1}) \odot \sigma'(z^l) \tag{3.12}$$

The last two equations let us compute the rate of change of the cost with respect to the bias and with respect to the weight.

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{3.13}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{3.14}$$

3.2 Recurrent Neural Network

In this section we take a step further towards the problem addressed in this work. While throughout section 2.1 a brief overview of feed forward neural network has been given, another kind of network is introduced here: Recurrent Neural Networks.
This type of networks was created in the 80's and leverage a different concept that others simpler networks are not able to exploit. They make the strong assumption that input sets are incorrelated. This can not be a problem for some tasks like image classification where only spatial features are required to provide correct answers.
But what about other, more complex and time related tasks, where the

previous (and potentially future) states and conditions are relevant to make current choices?
Examples of these kind of tasks are action recognition, time series prediction, speech recognition, natural language processing, robot control and so on.

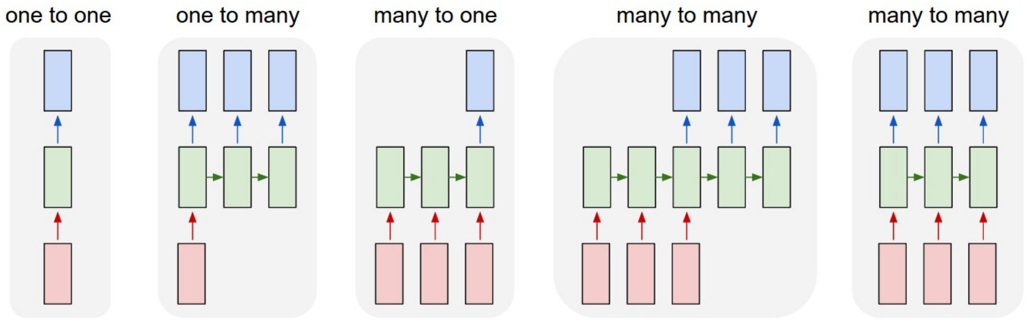


Figure 3.7: RNNs can work with sequences as either input, output or both

Source: http://karpathy.github.io/

Recurrent Neural Networks (RNNs), as stated above, exploit the fact that input sets are not independent from each other, hence RNNs are capable of operating with sequence of vectors.
But even more interesting is the fact that they are able to retain an internal state that updates at each iteration thanks to the recurrent connections.

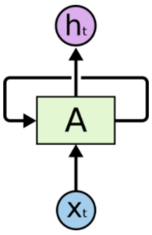


Figure 3.8: Simple representation of a RNN: A is a neural network, x_t is the input at time t, h_t is the internal state at time t.

Source: http://colah.github.io/

The basic idea lies in **Sharing Parameters** across different parts of a model. In particular to be able to process sequences of different lengths the network must be able to use the same parameters across different time steps. Basically, each member of the output is computed using the same parameters, i.e. the internal state that is in turn a function of the previous outputs.

There are two ways of representing RNNs: the first one (Fig. 3.8) is a diagram containing one element for every component that might exists in an implementation of the model. The other way is by using a computational graph where each component is represented by many different variables with one variable per time step; each variable is a separate node of the computational graph (Fig. 3.9).

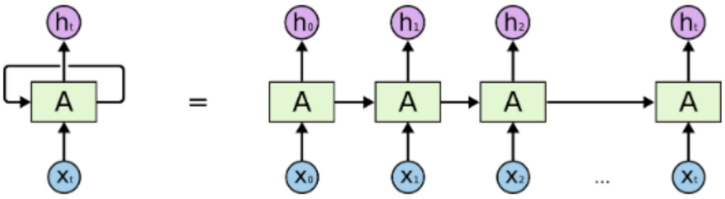


Figure 3.9: Unfolding of the previous RNN representation

A RNN can be unfolded into a computational graph obtaining the graph in Fig. 3.9 that represent a deep network structure where each chunk A share the same parameters, as discussed before. This is possible due to the fact that this network can be thought as a recurrent equation like the following:

$$h^t = f(h^{t-1}, x^t, \theta) \tag{3.15}$$

Where h^t is the hidden state at time t and x^t is the input at time t. By definition the network can be unfolded for a finite number of time steps τ obtaining a computation graph like the one in Fig. 3.9. For $\tau = 3$ the following non-recurrent equation is obtained:

$$h^{(3)} = f(h^{(2)}, x^{(3)}, \theta) = f(f(h^{(1)}, x^{(2)}, \theta), x^{(3)}, \theta) \tag{3.16}$$

θ was explicitly put in the notation to remind that f is a function of θ where θ doesn't change in time.
A key observation here is that we are not defining a different function, say g^t for each timestep $t \in \tau$ that takes as input all the sequence up to t:

$$h^{(t)} = g^{(t)}(x^t, x^{t-1}, ..., x^1) \tag{3.17}$$

We are defining a single function f (Equation 2.15) that only takes as input the current timestep of the sequence x^t, θ and the hidden status h^t. This single shared model allows generalization to sequence lengths that did not appear during training.
Intuitively, the hidden state h^t can be thought as a lossy encoding of all the task-relevant aspects of the past sequence of inputs up to that timestep that the network learns to generate.

3.2.1 General RNN model

We present a general model of a Recurrent Neural Network even though this is not the only design pattern for RNNs. This model is a recurrent neural network that produce an output at each time step and have recurrent connections between hidden units.
We introduce this model as a reference to discuss the Back Propagation Through Time algorithm, in the next subsection.

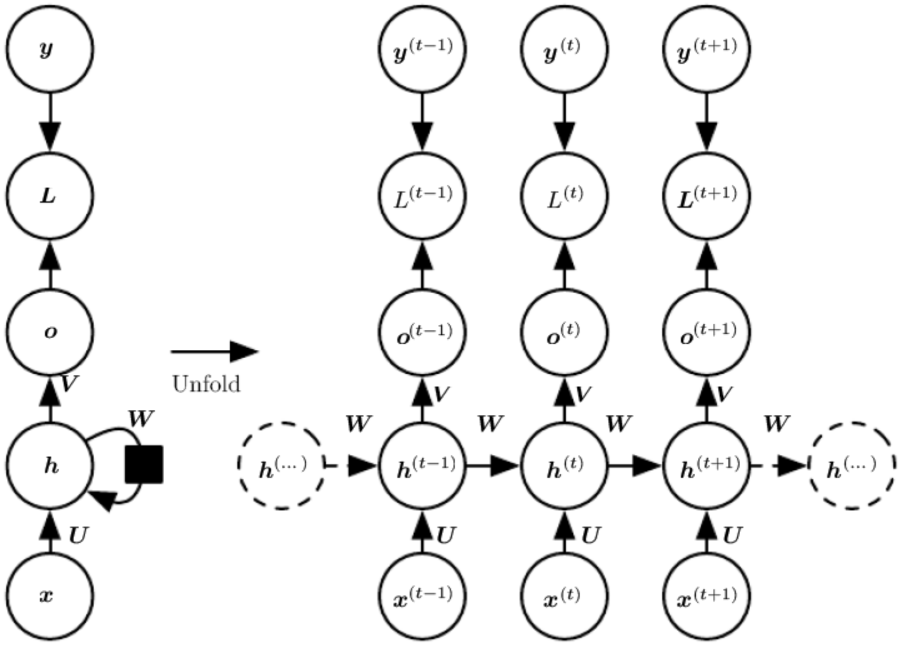


Figure 3.10: The unfolding of a general RNN model

Source: http://www.deeplearningbook.org/

Fig.3.10 depicts the unfolding of a general RNN model that maps an input sequence of **x** values to a corresponding sequence of **o** output values. The *loss* L measures how far each output is from the corrisponding training target **y**.
The matrices **U**, **W** and **V** are the weight matrix of input to hidden connections, the weight matrix of hidden to hidden recurrent connections and the weight matrix of hidden to output connections, respectively.

We can assume that the activation function of the hidden units is the hyperbolic tangent function, instead of the sigmoid function used when introducing the neuron model in section 3.1. The main difference is that the output value is not limited between 0 and 1 anymore, but between -1 and 1 instead. Also we assume that the output is discrete. Forward propagation begins with a specification of the initial state h^0.

For each timestep t, $1 \leq t \leq \tau$ the following update equations are used, with parameters bias vectors $\mathbf{b}$ and $\mathbf{c}$:

$$\mathbf{a}^t = \mathbf{b} + \mathbf{W}\mathbf{h}^{t-1} + \mathbf{U}\mathbf{x}^t \tag{3.18}$$

$$\mathbf{h}^t = tanh(\mathbf{a}^t) \tag{3.19}$$

$$\mathbf{o}^t = \mathbf{c} + \mathbf{V}\mathbf{h}^t \tag{3.20}$$

$$\hat{\mathbf{y}}^t = softmax(\mathbf{o}^t) \tag{3.21}$$

3.2.2 Back Propagation Through Time

The **Back-Propagation Through Time** [30] (BPTT) is the algorithm used for computing the gradient through a recurrent neural network. It is the application of the general Back-Propagation algorithm to the unfolded computational graph.

We take the previous model as a reference to explain the algorithm, for each node in the graph the gradient has to be computed recursively, based on the gradient computed on the nodes ahead.

The last nodes, i.e. the ones immediatly preceding the loss, are given by:

$$\frac{\partial L}{\partial L^t} = 1 \tag{3.22}$$

Where we assumed that the outputs o^t were used to compute the softmax function to obtain the output $\hat{y}$ and that the loss function is the negative log-likelihood of the objective y^t. The gradient of the output at timestep t, for all i as:

$$(\nabla^t_{\mathbf{o}} L)_i = \frac{\partial L}{\partial o^t_i} = \frac{\partial L}{\partial L^t}\frac{\partial L^t}{\partial o^t_i} = \hat{y}_i - \mathbf{1}_{i,y^t} \tag{3.23}$$

The idea is to propagate backwards through the sequence, starting from the end $t = \tau$:

$$\nabla_{\mathbf{h}^\tau} L = \mathbf{V}^T \nabla_{\mathbf{o}^\tau} L \tag{3.24}$$

From now the actual back propagation through time can be applied, starting from $t = \tau - 1$ down to $t = 1$. The gradient now depends on both $\mathbf{h}^{t+1}$ and $\mathbf{o}^t$, obtaining:

$$\nabla_{\mathbf{h}^t} L = (\frac{\partial \mathbf{h}^{t+1}}{\partial \mathbf{h}^t})^T (\nabla_{\mathbf{h}^{t+1}} L) + (\frac{\partial \mathbf{o}^t}{\partial \mathbf{h}^t})(\nabla_{\mathbf{o}^t} L) \tag{3.25}$$

$$= \mathbf{W}^T(\nabla_{\mathbf{h}^{t+1}}L)diag(1-(\mathbf{h}^{t+1})^2)+\mathbf{V}^T(\nabla_{\mathbf{o}^t}L) \tag{3.26}$$

The gradients of the parameter nodes can now be computed, observing that they are shared across the timesteps:

$$\nabla_{\mathbf{c}}L=\sum_t(\frac{\partial\mathbf{o}^t}{\partial\mathbf{c}})^T\nabla_{\mathbf{o}^t}L=\sum_t\nabla_{[o]^t}L \tag{3.27}$$

$$\nabla_{\mathbf{b}}L=\sum_t(\frac{\partial\mathbf{h}^t}{\partial\mathbf{b}^t})^T\nabla_{\mathbf{h}^t}L=\sum_t diag(1-(\mathbf{h}^t)^2)\nabla_{\mathbf{h}^t}L \tag{3.28}$$

$$\nabla_{\mathbf{V}}L=\sum_t\sum_i(\frac{\partial L}{\partial_{o_i^t}})\nabla_{\mathbf{V}}o_i^t=\sum_t(\nabla_{\mathbf{o}^t}L)\mathbf{h}^{t^T} \tag{3.29}$$

$$\nabla_{\mathbf{W}}L=\sum_t\sum_i(\frac{\partial L}{\partial_{h_i^t}})\nabla_{\mathbf{W}^t}h_i^t \tag{3.30}$$

$$=\sum_i diag(1-(\mathbf{h}^t)^2)(\nabla_{\mathbf{h}^t}L)\mathbf{h}^{t-1^T} \tag{3.31}$$

$$\nabla_{\mathbf{U}}L=\sum_t\sum_i(\frac{\partial L}{\partial h_i^t})\nabla_{\mathbf{U}^t}h_i^t \tag{3.32}$$

$$=\sum_t diag(1-(\mathbf{h}^t)^2)(\nabla_{\mathbf{h}^t}L)\mathbf{x}^{t^T} \tag{3.33}$$

3.2.3 Vanishing and Exploding Gradient Problem

Optimization algorithms may encounter difficulties when the computation graph becomes very deep. RNNs have deep computation graph that can be built by repeatedly applying the same operation at each time step of a long temporal sequence.

Lets consider this operation is a multiplication by a matrix W. After t steps this is equivalent to applying W^t. We can decompose this as:

$$W^t=(Vdiag(\lambda)V^{-1})^t=Vdiag(\lambda)^tV^{-1} \tag{3.34}$$

It can be observed that any eigenvalues λ_i that is, in absolute value, greater then 1 will eventually explode leading to the saturation of the gradient during training. On other hand, if it is smaller than 1, it will vanish leading to a loss of information about which direction to take for correct optimization. The first behavior causes unstable learning while the second, more frequent, makes it difficult for the network to learn correlation between temporally distant events and consequently to learn the long term dependencies inside the sequences.

Recurrent Neural Netorks don't differ much from the previous example and we can describe the recurrence relation for the hidden state, where we ignore the bias b and input x as:

$$h^{(t)} = W^T h^{(t-1)} \tag{3.35}$$

the previous equation is also called the *power method* and it can be simplified to:

$$h^{(t)} = (W^t)^T h^{(0)} \tag{3.36}$$

if W admits an eigendecomposition of the form $W = Q\Lambda Q^T$ with orthogonal Q, it can be further simplified to:

$$h^{(t)} = Q^T \Lambda^t Q h^{(0)} \tag{3.37}$$

We can observe that the eigenvalues are raised to the power of t causing eigenvalues with magnitude less than one to decay to zero and eigenvalues with magnitude greater than one to explode. This does not mean that is impossible to learn long term dependencies but that it might take a very long time to learn them. Indeed, the signals about these dependencies will tend to be overridden by even the smallest fluctuations arising from short term dependencies.

3.3 Long Short-Term Memory Network

As seen in the previous section the error computed using Back Propagation Through Time in RNNs tends to either explode or vanish, **Long Short-Term Memory** [31] (LSTM), a particular kind of recurrent network that aim to solve these issues is presented in this section.
LSTM were proposed by Hochreiter & Schmidhuber in 1997 to solve the vanishing and exploding gradient problem and they are able to remember really long-term interactions. They are based on a simple principle called *Gating*: it is a simple multiplication between two vectors. Say x our input vector and v our gating vector, $x^T v$ is the output vector. This is a simple weighting process used to select the element of the input vector and it is used by LSTMs and, in general, by **Gated Recurrent Units** (GRUs).

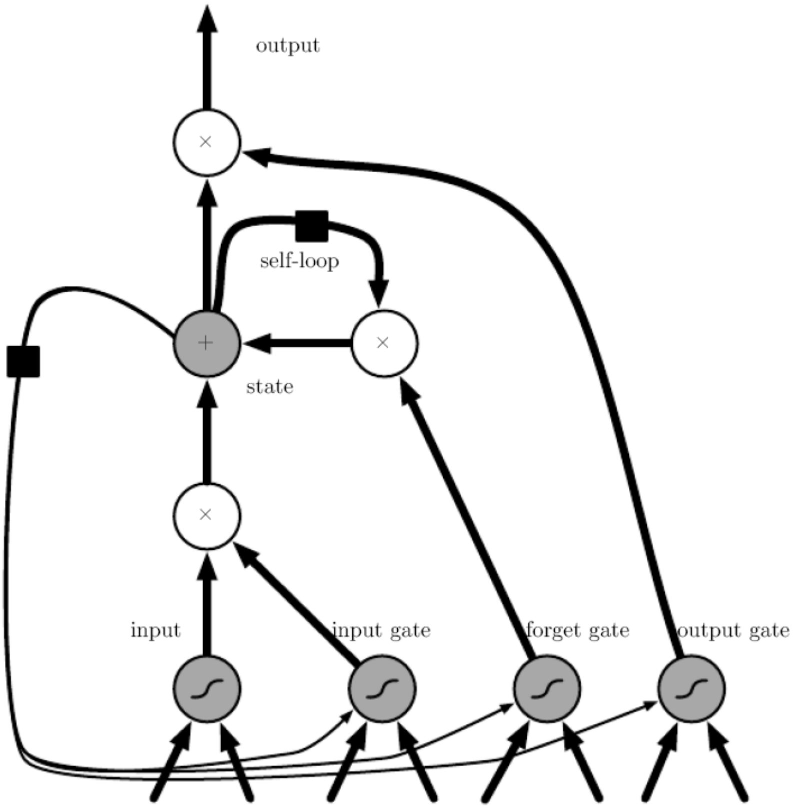


Figure 3.11: A block diagram of the LSTM recurrent network cell

Source: http://www.deeplearningbook.org/

The basic idea is to let the network learn the weights of these gates instead of choosing fixed parameters, in this way LSTMs are able to learn how to control the flow of information inside the cell and choose what to remember and what to forget.
In Fig. 3.11 a block diagram of an LSTM cell is presented. Many recurrent connections can be observed straight away, the black box stands for a one step delay. In the bottom left of the image a feature is computed from the input and the previous state (called *Context*) using a regular artificial neuron unit. Then the first of three gates, called the *Input Gate*, selects if the just computed value can be accumulated into the state. The second gate, the *Forget Gate*, selects which part of the previous timestep state has to be forgot and which not and can flow inside the state. The state value is computed by adding the gated input and gated previous state, for this reason the state has to be squashed to normalize its values.

The third and last gate is called the *Output Gate* that selects which parts of the current internal state can flow through the output.

Now that we developed the intuition behind this kind of network a formal definition can be given. Of course the most important part of the network is the state cell so let's begin from the computation of the state unit value s_i^t of the i^{th} cell at time t. Looking at the diagram above we can recall that it is given by the sum of two terms, so lets start with them. First we define the external input gate:

$$g_i^t = \sigma(b_i^g + \sum_j U_{i,j}^g x_j^t + \sum_j W_{i,j}^g h_j^{t-1}) \tag{3.38}$$

it is computed using gating on the current input vector x^t and the previous time step hidden state vector h^{t-1} by multiplying with the i^{th} row of matrices U^g of input weights and W^g of recurrent weights, respectively. Then the sigmoid function is applied to obtain a value between 0 and 1.
The forget gate can be computed in almost the same way:

$$f_i^t = \sigma(b_i^f + \sum_j U_{i,j}^f x_j^t + \sum_j W_{i,j}^f h_j^{t-1}) \tag{3.39}$$

We can now give a formal definition of the internal state:

$$s_i^t = f_i^t s_i^{t-1} + g_i^t \sigma(b_i + \sum_j U_{i,j} x_j^t + \sum_j W_{i,j} h_j^{t-1}) \tag{3.40}$$

The key observation is that we do not multiply many times by the same operation W^t as discussed in the vanishing and exploding gradient problem section but the recurrent update is given by the sum of two components instead.
Finally, we can provide the last two equations for the output to obtain the current output vector h_i^t:3

$$h_i^t = tanh(s_i^t) q_i^t \tag{3.41}$$

$$q_i^t = \sigma(b_i^o + \sum_j U_{i,j}^o x_j^t + \sum_j W_{i,j}^o h_j^{t-1}) \tag{3.42}$$

Chapter 4

Experiments

In this chapter the main experiments and results are reported and analysed. We tried to make the network learn how to swap two cubes positions and how to treat unexpected failures. We exhibit the insights emerged in each of them, and useful for the progress of the research. In Appendix A.1 some simple, preliminary experiments are reported and discussed. Those experiments were needed to isolate the training to a minimal task level, so that we were able to determine if LSTMs were capable of learning the basic behaviors needed.

The main experiments are divided in two parts: the first part uses a continuous domain and led to the development of the second part where a discretization of the domain is used instead. Each part of the experiments can be further divided into two parts: the first one tries to make the network learn the action prediction and argument generation when no failures can happen. The second, instead, introduces the possibility of actions failing.

4.1 Configuration

4.1.1 Tensorflow

TensorFlow [32] [33] is an OpenSource software library for Numerical Computation largely used for Machine Learning and training Neural Networks.

It was developed internally by the *Google Brain* team at first but in 2015 it was released under a open source license.

TensorFlow uses data flow graphs where nodes represent mathematical operations and edges represent the multidimensional data arrays, called *Tensors*, that are passed between nodes. The flexible architecture allows the user to

Figure 4.1: The TensorFlow logo

Source: https://www.tensorflow.org/

deploy computation to one or more CPUs or GPUs in a desktop, server or mobile device using a single API.

4.1.2 Keras

Keras [34] is a high-level neural networks API, written in Python and capable of running on top of **TensorFlow**, **CNTK** [35] or **Theano** [36]. In this work we used TensorFlow as the backend.
Keras was developed with the aim of enabling fast experimentation as it allows easy and fast prototyping of an idea through user friendliness, modularity, and extensibility.

Figure 4.2: The Keras logo

Source: https://keras.io/

4.1.3 Platform Specifications

All of the experiments have been run on a machine with 8GB DDR3 memory and an Intel i5 2500k at 4.2GHz CPU using Ubuntu 16.04 Operating System. Tensorflow was compiled from source to take advantage of the SSE instructions and gain some performance improvements. The training times range from few minutes for the simpler cases where there are hundreds of parameters up to 2 or 3 hours for cases where there are thousands of parameters.

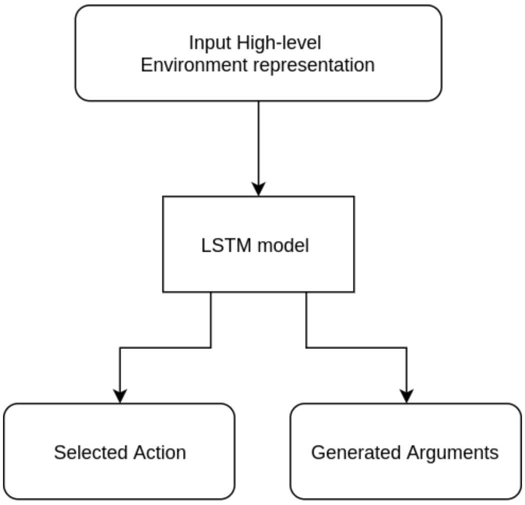


Figure 4.3: Main building block of the architecture

Tensorflow with GPU support was also tried as backend for Keras using a nVidia GTX 970 but in fact the performances deteriorated. This is probably due to the low batch size and low cardinality of the input, that does not yield a good speed up factor to improve training times. This combined with the overhead needed to load the GPU's memory makes it not enough to switch to GPU and it is why CPU was preferred.

4.1.4 Architecture

The main building block of our model is a LSTM, to train the network an expert policy was developed for each of the cases. This policy produces the training examples which are composed by a pair of two sequences of length $maxlen$: the first is a sequence of vectors which elements describe the environment, the second of vectors which elements describe the target output instead. The input vector has n_{dim} features that represent the environment observation while the output vector is composed of o_{dim} features.
The i^{th} training example can be defined as

$$\begin{aligned} \{(In, Out)_i | In_t = \{in_1, in_2, ..., in_n\}, \\ Out_t = \{out_1, out_2, ..., out_o\}, \\ \forall t \in [0, maxlen]\} \end{aligned} \tag{4.1}$$

The expert policy is used to generate an artificial dataset used to train the network and to evaluate its performances over a set of unseen examples.

Fig. 4.3 describes the general architecture where the LSTM model uses as its inputs the high level representation of the environment and provide the selected action and generated arguments using two separate output layers. Initial experiments were executed using a single output layer but results quickly pointed out that using a single loss function for two different output often leads to bad results.
Therefore the output is divided in two different layers, each with its own loss function and accuracy metrics.

The architecture can be expressd in a mathematical form by considering the problem of an agent performing actions to interact with an environment to accomplish a complex task. Let $\epsilon_t \in \mathcal{E}$ be the environment observation at time t and let $s_t \in \mathcal{R}^D$ be a fixed-length state encoding extracted by a domain-specific encoder $f_{enc} : \mathcal{E} \rightarrow \mathcal{R}^D$.
The network takes the environment representation s_t as input, possibly along with other type of inputs like a motion planner failure, and computes a hidden state $h_t \in \mathcal{R}^M$ that summarizes the progression of the task in the current environment, different decoders can produce different outputs.
There are two types of outputs: the first is needed to select the next action to perform and can be computed as $f_{Act} : \mathcal{R}^M \rightarrow \mathcal{A}$, where $\mathcal{A}$ denotes the finite set of actions.
The second output type is the parameter generator and it is computed in different ways, in general each i^{th} argument can be computed as either a function $f^i_{Arg} : \mathcal{R}^M \rightarrow \mathcal{U}$ where $\mathcal{U}$ is a finite set of arguments, or a function $f^i_{Arg} : \mathcal{R}^M \rightarrow \mathcal{R}^k$, where $\mathcal{R}$ is the set of real numbers. Then the softmax activation function can be applied to obtain a k-dimensional vector with each value between 0 and 1 and which sum is equal to 1. The argument with the maximum value is then selected as output.
The first output type is used when the arguments are generated using a continuous domain, and the problem can be seen as a linear regression. The second one is instead used when the arguments are generated using a discrete domain, and the problem can be seen as a classification.

An important choice made was to avoid the domain-specific encoder by directly providing the network with a high-level representation of the environment. In this way we can assume that the input position of the object is correct and does not include any error due to imprecision on the prediction of an external encoder. Moreover this allows us not to restrict to a specific environment and therefore to expand to a variety of different domains.

4.2 Object Swapping Task

The use case of this thesis is a simple robotic task where the scene is composed of two cubes on a table and the robot should swap the cubes' positions.
The environment can be restricted to a square table where the cubes are placed on. When considering this table as a continuous environment, the cubes can have any (X, Y) position inside the side the cells, except for $(0, 0)$ that is used as a symbolic position when the cube has been grasped by the robot. When using a discrete environment the table can be thought as a chessboard where the cube can only be positioned at the chessboard cell centers.
The robot should pick the first cube, place it in a temporary location, pick the second one and place it on the first cube original position. Finally, it has to pick up again the first cube from the temporary location and place it in the second cube's original position (Fig. 4.4).

Of course, learning the task means: learning the sequence of actions to perform in order to achieve the target task, but also learning how to recover from failures.
An action can fail, e.g. the grasped cube can slip from the gripper of the robot or can be placed in a wrong position. This implies that the action has to be performed again and eventually other actions have to be executed before to restore the right preconditions.

4.2.1 Expert Policy - Dataset Generator

Since a dataset for these kind of experiments is not available and hard to acquire from real data, each of the next experiments uses an **artificial dataset**. This dataset is randomly generated right before training and it is based on the fixed structure for the selected task. Whenever a pick fails it is executed again until it succeds and whenever a place fails, the cube has to be picked up again, and that pick action can fail as well.

The Dataset Generator has been developed using PYTHON and it is able to create a dataset using a specified input parameter for the error probability of each primitive.
It generates a sequence of Input-Output pairs called ENVIRONMENT and OUTPUT that represent the observed high level encoding of the environment and the desired output of the network, respectively.

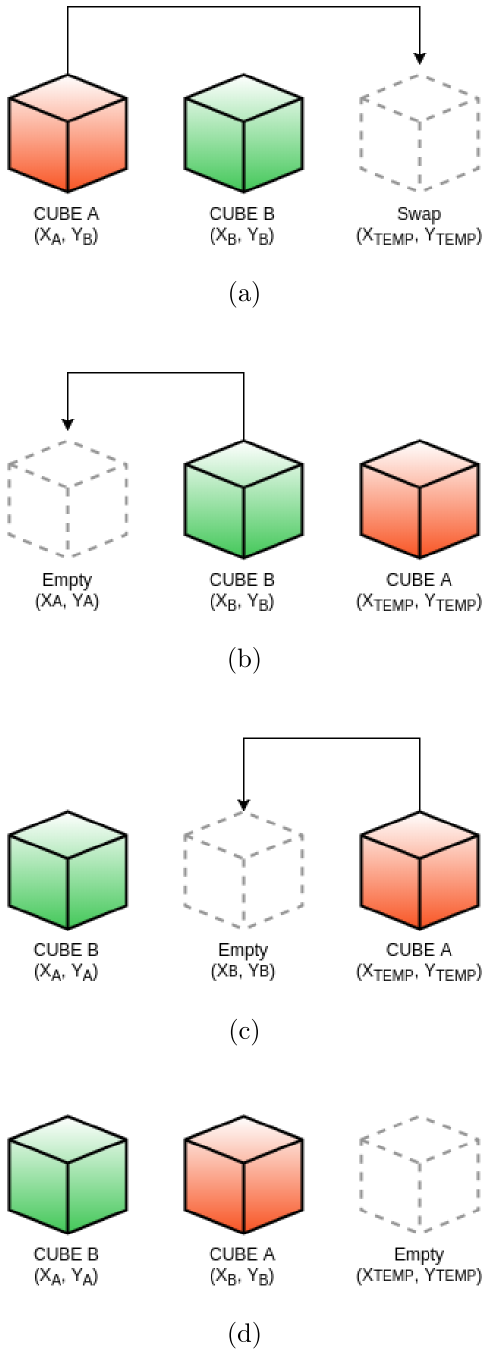


Figure 4.4: The complete task execution: Move A in a temporary position 4.4a, move B to A initial position 4.4b, move A to B initial position 4.4c and the scene after the completion of the swapping task 4.4d

The output is then further divided in Action Output and Argument Output as previously discussed.

The description of the environment is composed by the coordinates of the two cubes (X_A, Y_A) and (X_B, Y_B), as well as the status of the gripper. Each coordinate is a single input to the network and goes from 0 to 1 for the continuous domain. Fig. 4.5 shows a the sequence of pairs using the continuous domain. In the discrete domain each coordinate is represented by 10 inputs that are either 0 or 1 and represent the X or Y coordinate of the specified cube.

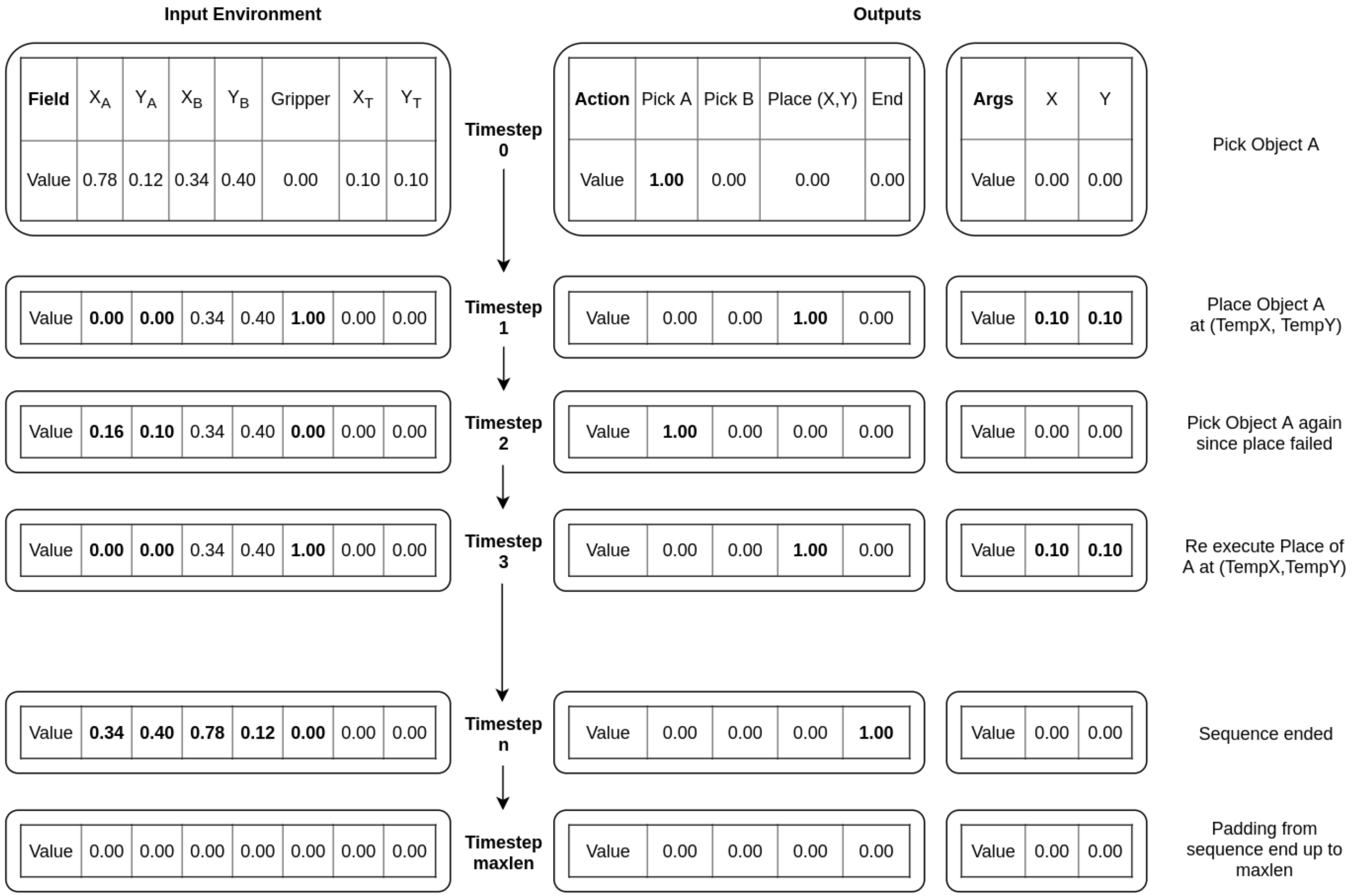
During the first experiments the (X, Y) temporary swap position coordinates had to be generated by the network, this led to problems during the evaluation of the network. This was due to the fact that the predicted temporary positions were nearly always different from the training example randomly generated positions. This behaviour is correct but was not expected at first, the generator has been modified to include the temporary position as input to avoid this type of problems. Specifically 2 inputs node for the continuous domain and 20 for the discrete domain, 10 for each input.

The desired temporary position is provided in input at the first timestep of the sequence and then is set to a specific value like 0 or -1 representing a null value.

The generator is able to introduce an error probability that simulates the failure of an action. For example the action of Picking a cube can fail, meaning that the cube falls from the gripper of the robot, and therefore the actions needed to pick up the object again are introduced to the sequence. This means that sequences will be, as expected, of different lengths. The more failures happens the more actions will be added into the sequence. In listing 4.1 the output of the network is reported where no failures happened. In listing 4.2, instead, the first pick failed and it has been executed a second time, that succeeded. The second place also fails and the cube B has to be picked up again and before attempting again a place.

```
PickA();
Place(tempX,tempY); // Temp position specified in input
PickB();
Place(A_x, A_y);    // at A's original position
PickA();
Place(B_x, B_y);    // at B's original position
```

Listing 4.1: The output of the network can be represented as a list of primitive invocations and relative arguments. In this example the primitives never fail therefore only 6 invocations are needed



Input Environment

Outputs

Timestep	Field	X_A	Y_A	X_B	Y_B	Gripper	X_T	Y_T	Action	Pick A	Pick B	Place (X,Y)	End	Args	X	Y	
Timestep 0	Value	0.78	0.12	0.34	0.40	0.00	0.10	0.10	Value	**1.00**	0.00	0.00	0.00	Value	0.00	0.00	Pick Object A
Timestep 1	Value	**0.00**	**0.00**	0.34	0.40	**1.00**	0.00	0.00	Value	0.00	0.00	**1.00**	0.00	Value	**0.10**	**0.10**	Place Object A at (TempX, TempY)
Timestep 2	Value	**0.16**	**0.10**	0.34	0.40	**0.00**	0.00	0.00	Value	**1.00**	0.00	0.00	0.00	Value	0.00	0.00	Pick Object A again since place failed
Timestep 3	Value	**0.00**	**0.00**	0.34	0.40	**1.00**	0.00	0.00	Value	0.00	0.00	**1.00**	0.00	Value	**0.10**	**0.10**	Re execute Place of A at (TempX,TempY)
Timestep n	Value	**0.34**	**0.40**	**0.78**	**0.12**	**0.00**	0.00	0.00	Value	0.00	0.00	0.00	**1.00**	Value	0.00	0.00	Sequence ended
Timestep maxlen	Value	0.00	0.00	0.00	0.00	0.00	0.00	0.00	Value	0.00	0.00	0.00	0.00	Value	0.00	0.00	Padding from sequence end up to maxlen

Figure 4.5: A part of a continuous training example. The input environment (left) with the (x,y) position of the two object A and B, the status of the gripper (1.0 if holding an object), the temporary swap position (TempX,TempY). The outputs (right) the selected action and the generated argument (X,Y) for the place primitive.

Sequences are padded with zeros if shorter than the provided MAXLEN, this is because Keras needs sequences of the same length during training. The model will then ignore the padding and the parameters of the network will not change in those cases.

The generator also has different methods that are able to generate slightly different datasets to be used for each different experiment, described later in detail. It is able to generate a dataset with discrete or continuos parameters to allow different experiments.

```
PickA();                  // Failed!
PickA();                  // Re executes
Place(tempX,tempY);
PickB();
Place(A_x, A_y);          // Failed!
PickB();                  // Pick B again, but this fails too
PickB();
Place(A_x, A_y);
PickA();
Place(B_x, B_y);
```

Listing 4.2: In this output example the primitives can fail, more than 6 actions could be needed

4.2.2 Object Swapping using error free continuous domain

The first experiment is executed using a continuous domain so that cube positions are a pair (X, Y) of real numbers that can each range from 0.0 to 1.0.

We also ignore, for now, the possibility of an action failing. This would require the network, when necessary, to first select the sequence of actions to restore the right preconditions and then to select the last action to be executed again.

The network (Fig. 4.6) composed by a LSTM layer of 50 units was trained using 3000 artificial training examples using a standard 60-20-20 split for training-validation-test sets. After some cross experiments using different hyperparameters, a batch size equal to 4 training samples has been selected. This was the best tradeoff value between training time and minimum loss. Using a batch size smaller than 4 does not bring appreciable improvements but really extends the training time. On the other hand, increasing the batch size to higher values like 16, 32 or 64 really speeds up the times but with a

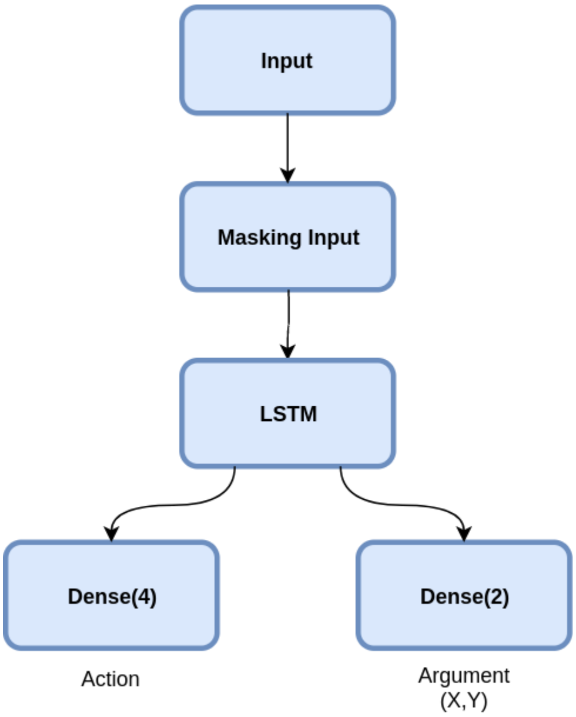
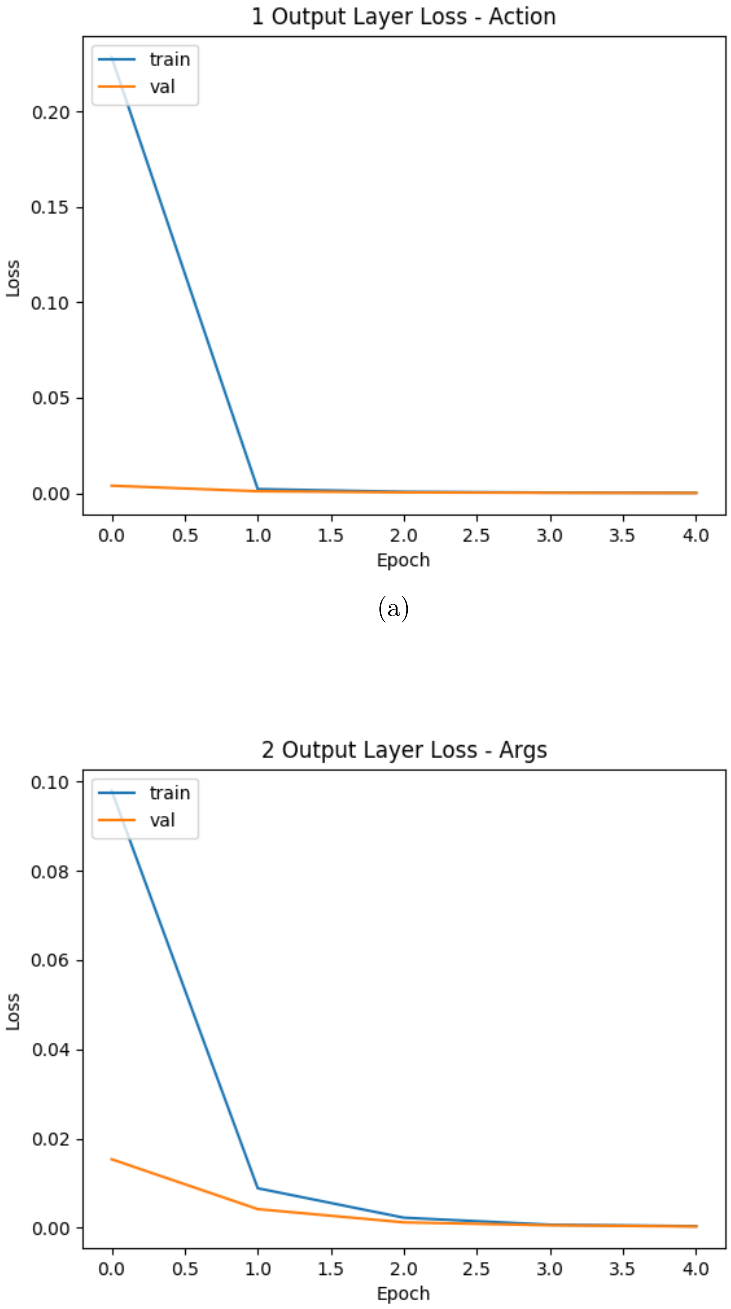


Figure 4.6: The structure of the network with a single output layer for the arguments (X,Y)

substantial increase of the loss.
The selected training algorithm is ADAM [37] with default parameters instead of the classic *Stochastic Gradient Descent* (SGD). This was mainly due to the reason that SGD uses a fixed learning rate while ADAM is able to adaptively change the learning rate.
The selected loss functions are the categorical cross-entropy and the mean squared error for the action and arguments (when using a continuous domain, otherwise cross entropy is used as well) output layers, respectively. Fig. 4.7 depicts the training and validation losses during the training of the network for every output layer.



(a)

(b)

Figure 4.7: Losses during training

The fundamental observation here is that since all the data is randomly generated, the two losses in each diagram would not part ways, meaning that

the network is overfitting. This is because the more training examples are generated the more the training, validation and test sets are statistically the same.
In this first experiment the output layer is basically a fixed sequence of actions and is correctly learned by the network as verified by the tests run.
The arguments output layer is a little bit more complex since it has to remember different dependencies. In particular it has to select the temporary, and original object positions in synchronization with the *place* actions.
From the tests it has been observed that the network is not able to use its memory to store exact values and provide them in output at the correct time. It is able to provide a value, altough approximated, at the correct timestep. If the network is not trained enough, these arguments will be close to the correct position but will be unusable for our purposes. Even with enough training we cannot assume to receive the exact same value as a generated argument. Another issue is that the error between the input and output arguments is not really consistent throughout the range of values.

4.2.3 Object Swapping using error-prone continuous domain

The previous experiment, altough interesting, did not really achieve our purpose because there were no primitive failure handling. In this test a probability that every single action can fail is introduced. Accordingly the generator will take an input error probability to determine with which probability an action will fail.
Introducing a failure probability for the primitives clashes with preconditions and effects. A failure can therefore modify the environment and the preconditions as well. The network has to selected the right actions to perform to restore the correct preconditions before re executing the failed primitive.
Specifically, whenever a *pick* action fails it has to be selected again until it succeds since preconditions should not be affected by this failure. The same goes for the *place* action except that when this action fails the correct preconditions (i.e. the robot need to grasp the object before placing it somewhere) have to be restored. Therefore the object has to be picked up before executing again the place.
These failures aim to reproduce an object incorrectly grasped, for example an object that slipped from the robot gripper or an object placed on a surface on a wrong position.

The ability of the network to learn these kind of problem is questioned

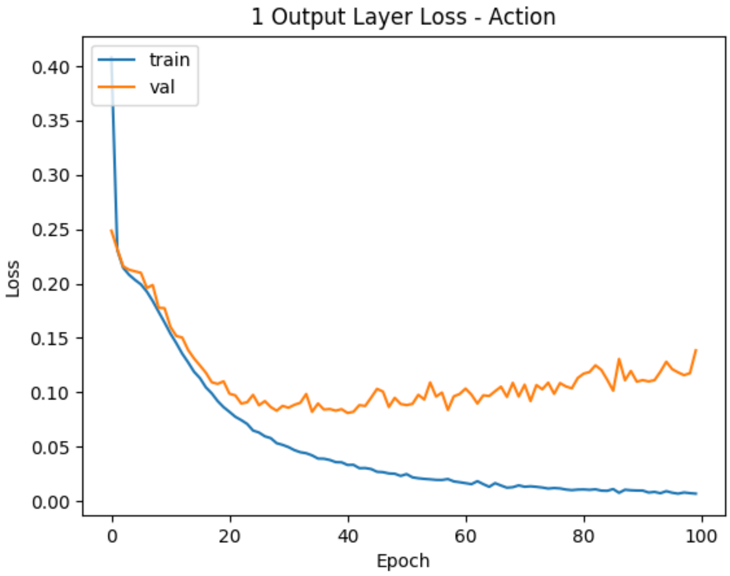
along with its ability to generalize to an unseen, different pattern of failures throughout the execution of the task.
The network was trained using 3000 training samples and introducing a failure probability for each discrete action of 0.2, splitting the data as already mentioned. Batch size of 4, 50 units, 100 epochs and ADAM [37] Optimizer with default parameters were selected for training.
In Fig. 4.8 validation and training losses and accuracies are reported. It can be observed that after 25 epochs the network starts to overfit and the training can be stopped at that point since the validation accuracy is not improving anymore. After 100 epochs the network start to deteriorate.

Results are not as good as those of section 4.2.2. The primitive action selection is learned, and if a cube is not correctly picked up, the pick is selected again. This behavior is influenced by the timesteps advancement. This means that if an action fails n times, the network will re execute that action one, two maybe three times but then it will eventually select the next action to be performed since, statistically, it is what happens in the dataset. In terms of generalization this is not ideal and points toward some kind of hierarchical solution.

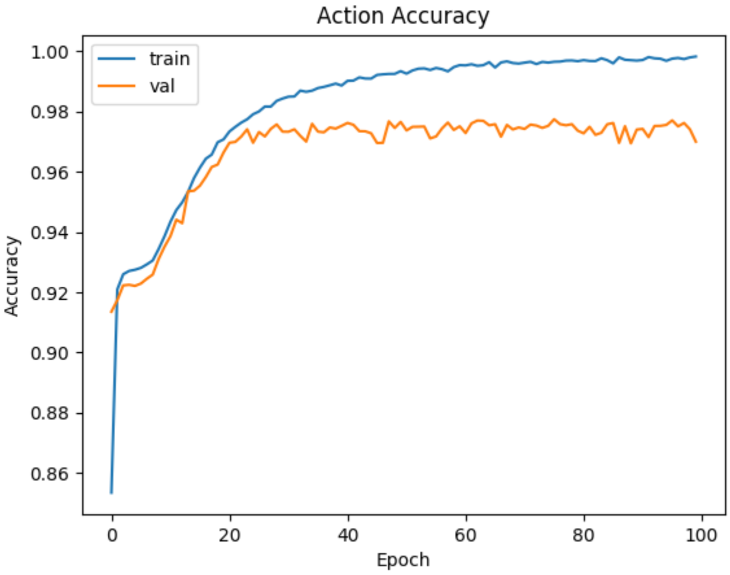
The values obtained can be quite off, meaning that the network is not really able to infer this type of behavior and is not capable of storing exact values inside its memory which instead is used to hold long term dependencies and relations. Moreover stopping the training at 25 epochs when the results for the action layer are not improving anymore could lead to even worse results, where the predicted (X,Y) place positions are wrong.

4.2.4 Object Swapping using error-free discrete domain

Inspired by the NPI paper [19] where the generated arguments are always discrete, experiments previously proposed are executed again. But the problem of arguments generation is formulated as a classification. This means working in a discrete domain instead of in a continuous one like in section refsection:SwapNoErrCont.
The arguments output layer for the (X, Y) coordinates is now splitted into two separate output layers where each coordinate has its own set of output values from 0 to 9 (Fig. 4.10). Specifically there are ten output per layer, where each value represent the cube position in either X or Y. Since the cube can be in only one cell at a time a softmax function is applied, in this way only the maximum value is selected as the predicted output argument.



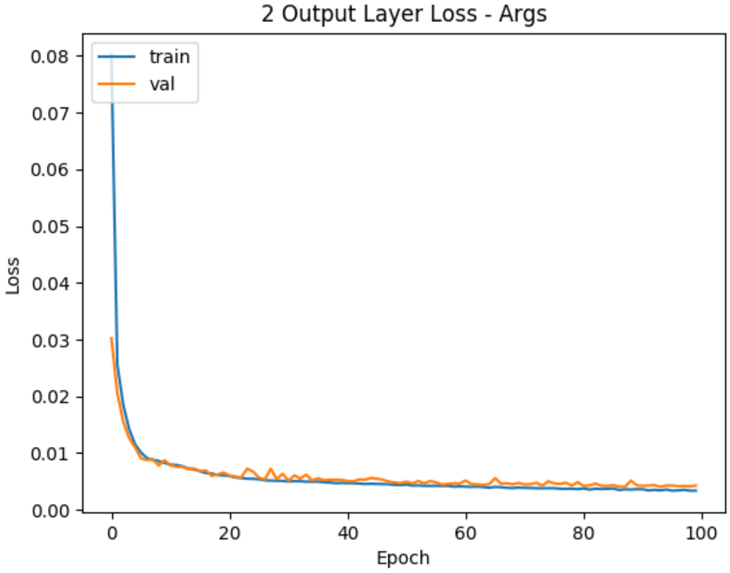
(a)



(b)

Figure 4.8: Action losses and accuracies during training with primitive failures

This can be seen as a discretization of the search space and in reality can



(a)

Figure 4.9: Losses during training with primitive failures

be interpreted as the position of the cubes inside a chessboard.
Training for 15 epochs using 3000 training examples a batch size of 4 samples, we obtain almost 100% of accuracy in each layer.

The loss regarding the action layer in Fig. 4.11a quickly drops as expected since a fixed pattern has to be learned.
The loss reported in Fig. 4.11b is for only one of the argument output layers since they are almost the same. As expected, it converges slower than the action layer, because only the place actions require the arguments that are set to a reserved value while the other primitives are selected.
Switching to a discrete domain also enable us to quantify in a more intuitive way how well the argument generation is performing by using accuracies instead of losses.

4.2.5 Object Swapping using error prone discrete domain

A failure probability of the 20% for each action is introduced to the experiment of section 4.2.3. Results regarding the parameters are definitely better. The best results were obtained by stacking 3 LSTM layers (Fig. 4.12) of 50

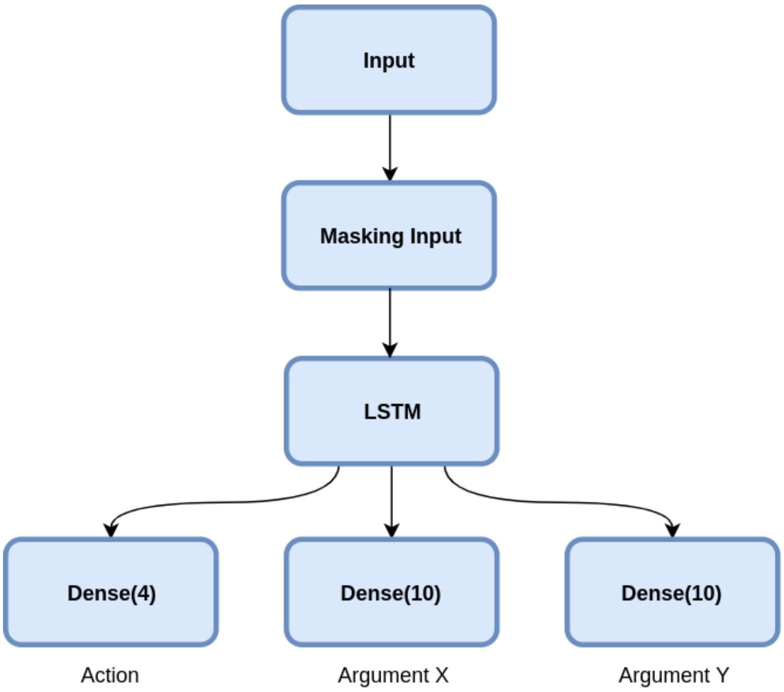
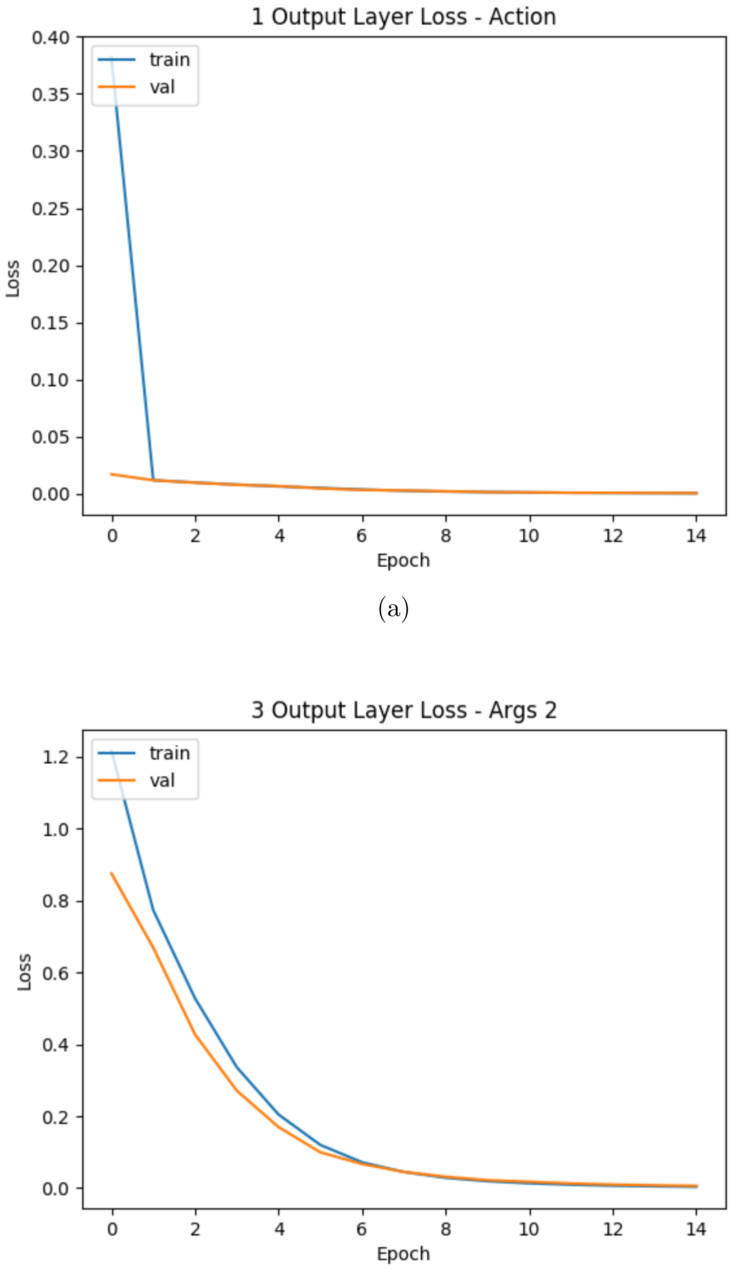


Figure 4.10: The structure of the network with two dense layer of 10 output for each argument

units each. The same number of training examples, splits and epochs of the previous experiments were used.
Accuracies of 97.3% and 99.5% are reached on the validation sets for the action layer and the argument layers, respectively (Fig. 4.13). These are good results.

In Fig. 4.14a the training and validation losses of the action layer are reported. It can be observed that the validation loss starts to deviate from the training loss meaning that the model is beginning to overfit. Anyway by looking at the validation accuracy it does not get worse. This is because even though the prediction is a little bit further from the target value, by applying the softmax that selects the maximum output, the output is still correct.
The spike that can be observed in the action loss (Fig. 4.14a) is probably due to a certain configuration of data since a mini batch size of 4 is relatively small and also due to the changes to the learning rate by ADAM, using a larger batch size would indeed smooth the losses.
The training of the parameters for the argument layers in Fig. 4.14b instead seems to be quite smooth. Unfortunately the learning of the actions collapse



(a)

(b)

Figure 4.11: Losses during training using an error-free discrete domain

with 5 stacked layers or when the number of LSTM units are further improved.

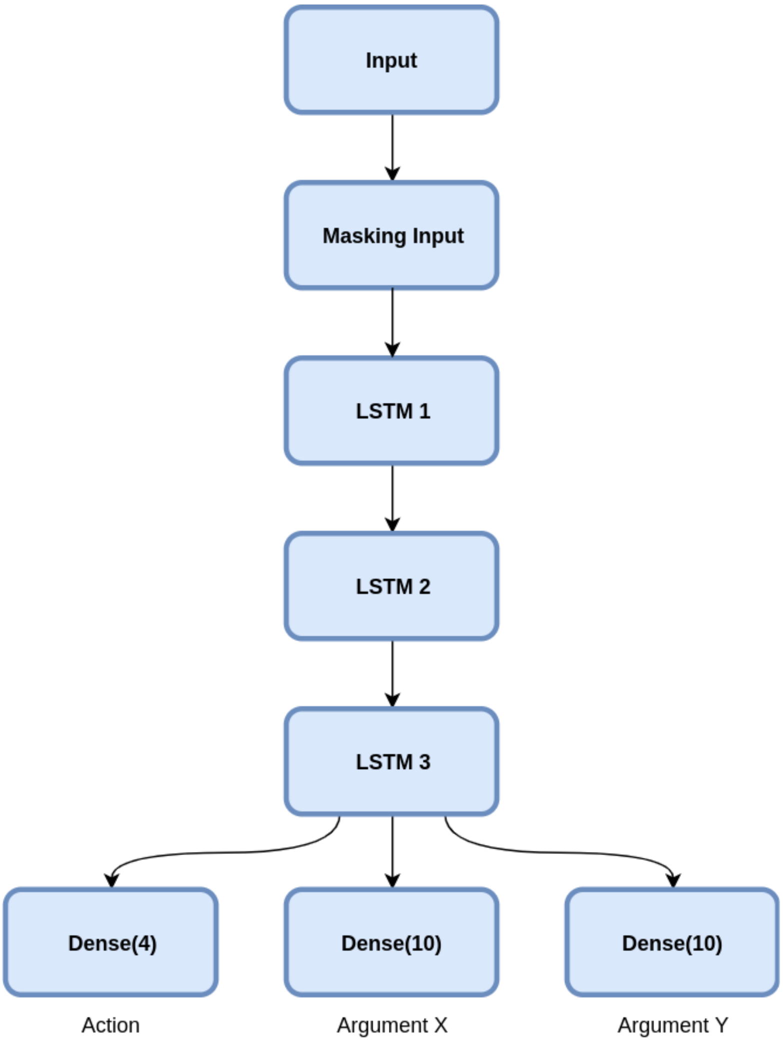
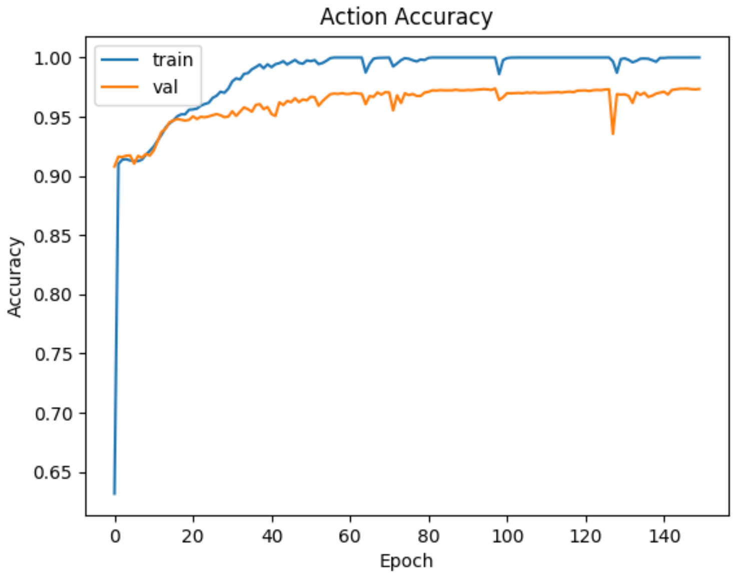


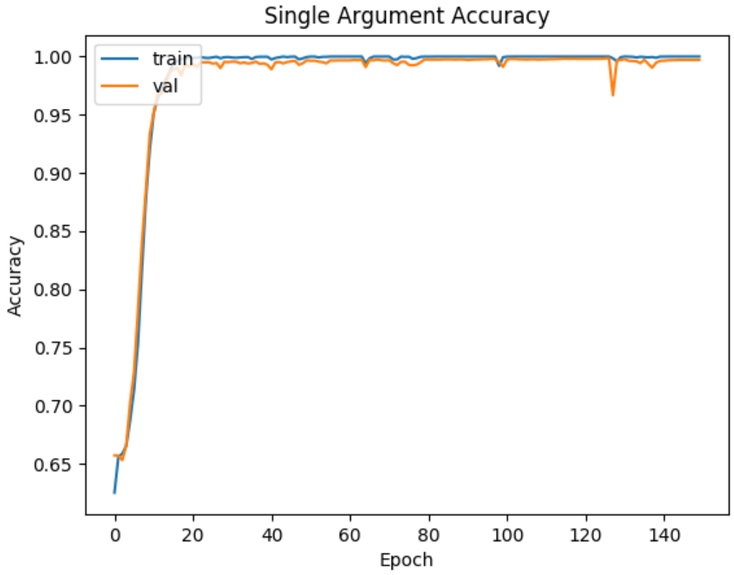
Figure 4.12: The structure of the network with 3 stacked LSTM layers

When stacking 5 LSTM layers, one of the two argument output layer does not correctly learn and diverges.

Using a continuous domain the action selection seems to behave quite good at first, by selecting the correct action for the first few timesteps. The fact that the predicted value has a dropping trend for example from 1.0, 0.7, 0.5, 0.3 from the same, repeated, action means that the network will eventually make a wrong prediction. Intuitively this is due to the fact that the network learns that after a few timesteps the next action is expected.
From this intuition we decided to introduce some limit cases inside the



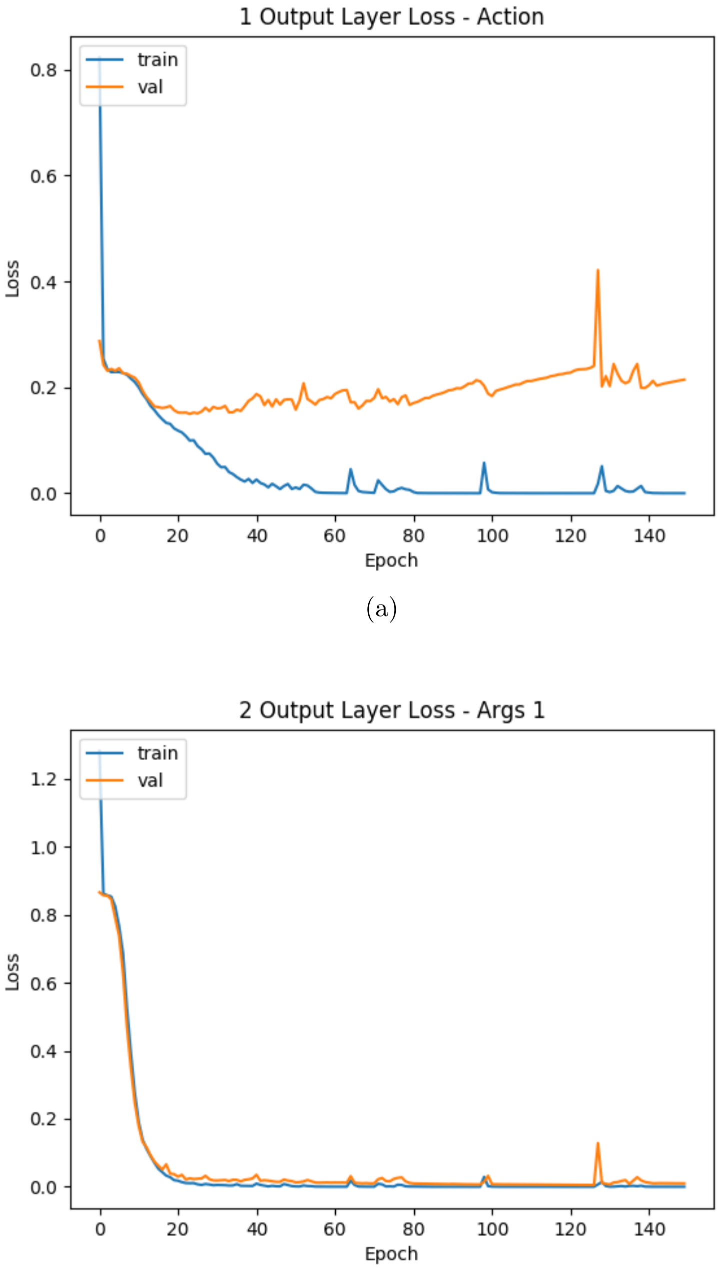
(a)



(b)

Figure 4.13: Accuracies during training and validation using error prone discrete domain

dataset to explore how the network prediction changes after seeing these



(a)

(b)

Figure 4.14: Losses during training using error prone discrete domain

special cases where, for example, only a specific action primitive fails just in order to fill the maximum sequence length.

4.2.6 Dataset Augmentation

In this section the **Dataset Augmentation** motivated at the end of section 4.2.5 is presented. The first experiment executed consists of the introduction inside the dataset of a specific example a certain number of times, up to 5% of the dataset.

Specifically what we did was inserting a specific case where the first pick would always fail and succeed just in time for the sequence to achieve the task using up to maxlen timesteps.

This does not seem to bring a really appreciable improvement in the overall prediction performances, that reaches around 97% of accuracy as before but on the same test dataset without the enrichment.

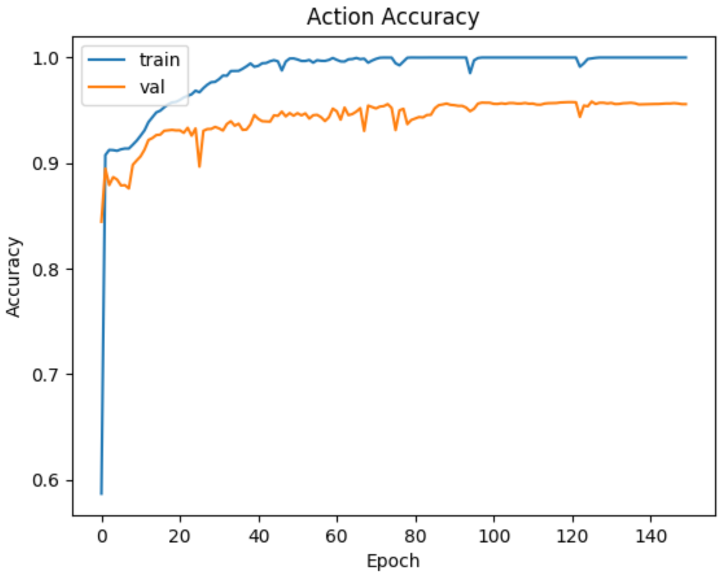
Then all of the other limit cases are inserted for the training. With all the limit cases we refer to those with first-level depth, so the limit cases in fact generated and inserted are those where either the first A pick, the first A place, the first B pick and so on. In other words the nested actions, like the pick needed to be re executed after a failed place, will not fail.

After retraining using this enriched dataset, the network was able to predict correctly examples similar to the newly inserted, i.e., where the first pick would always fails, until the primitive would be correctly executed. The real issue happens when, after a number n of failures of that specific action, the next primitive to execute is predicted correctly but fails again. The network expects it to succeed nevertheless making the next prediction wrong since the limit cases did not show this type of behavior.

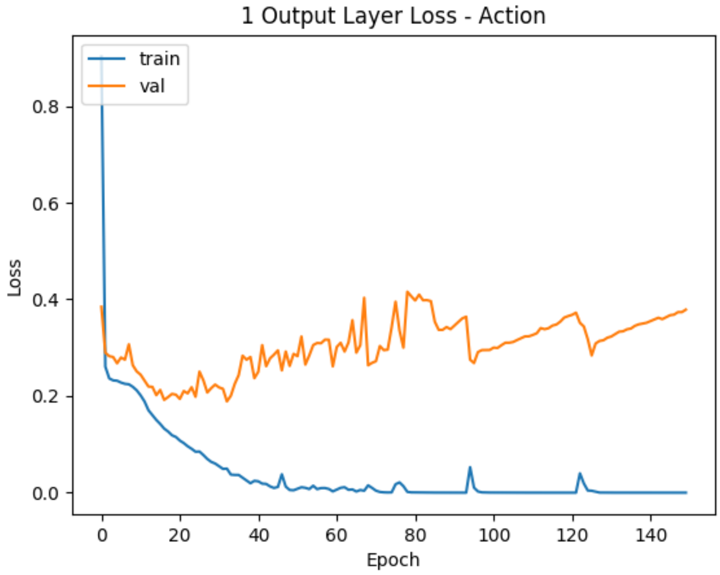
This is not really different from the problem discussed towards the end of section 4.2.3 and points out a limit of the current architecture. Possible solutions would be to either provide the network with a even richer dataset with even more limit cases or a different model.

The first option is not likely to be a good way to solve this problem, in fact what we wanted to achieve was good generalization. Therefore the objective is efficient learning by providing just the minimum number of training examples to make the network learn the task correctly.

The other option would be to use a different architecture that can make use of the recursion abstraction. In this way, the network should be able to decompose the problem into smaller pieces until base cases are reached. In fact, the proposed model suffers from errors due to undesirable dependencies that the model learns inside its hidden state. To avoid this and implement recursion, the network should receive as input its previous prediction (recursion) and the hidden state should be saved on a stack, just like with function calls. After a new prediction is made and before resetting the hidden state



(a)



(b)

Figure 4.15: Accuracies and losses during training using augmented dataset

the context need to be saved in the stac. This can be seen just like function invocation. Moreover, this allows the network to restrict the attention to the currently relevant recursive call, ignoring irrelevant details about other contexts. This could solve the issue encountered of the network "expecting" the next action in the sequence after a certain number of failures.

4.2.7 Generalization over longer sequences

To test the network's ability to generalize to longer sequences, i.e., with higher primitives error probabilities, an additional test dataset is generated using error probability of 0.45 for each primitive. As before, sequences can reach *maxlen* that is set to 100 for these samples. If the sequence is shorter than that it is padded using 0s up to *maxlen*.

Results obtained are fairly good: the network composed of a single LSTM layer with 50 units was able to achieve up to 85% of accuracy while the network composed of three stacked LSTM layer was able to achieve up to 91% instead. The augmentation of the dataset did not bring significant improvement over the normal one. This is probably because the longer sequences used for these tests probably doesn't have limit cases like the ones we manually introduced. An improvement in the accuracy is expected if these kind of examples are introduced, but they were not introduced on purpose to see if with more general samples the accuracy would increase, which did not.
Of course the aim is to avoid filling the dataset with all possible cases and looking towards a more efficient learning where less examples but more intelligence from the model is needed.

Chapter 5

Conclusions

In this work a review of Task Motion Planning and Neural network state of the art has been reported in chapter 2. An in depth analysis of the main frameworks used as starting point for this thesis has been given too.
Then an overview of Deep Learning and its core topics like Gradient Descent, Back propagation as well as an overview of Recurrent Neural Networks along with Backpropagation through time and problems related to the vanishing and exploding gradient have been given.

LSTM have been selected as the main building blocks for creating a task planner that is able to predict the next action to perform based on past history and the current environment observation. Preliminary experiments have been tried and are reported in Appendix A.1.
The main experiments are reported in Chapter 4 where we tried to make the network learn how to swap two cubes and how to handle unexpected action failures. We also reported the insights that brought to the development of the proposed architecture. The first part of the experiments was approaching argument generation as a regression problem, poor results led to treating the problem as a classification: in this way better results were obtained.
The hyperparameters used for training the network have been reported, as well with the selection criteria used.
Potential and limits of this solution has also been questioned and investigated.

Since a public dataset for these type of tasks is not available, a Dataset Generator to produce an artificial dataset has been proposed. The artificial data is generated for each of the experiments and is used to train the network.

5.0.1 Key aspects and limitations

The proposed model is capable of predicting the next action to perform with up to 97% accuracy on sequences of the same length used for training, which is 30 timesteps. 91% of accuracy can be obtained instead for sequences of up to 100 timesteps. The model doesn't need to compute multiple alternate plans and the idea can be easily extended with different feedback from the environment. For example, the feedback from an off the shelf motion planner could be integrated as input to the network, as discussed in the future works. However, the network does not provide strong generalization regarding task length, meaning that is able to achieve good results for a certain length of unseen task length but after that it start to quickly deteriorate. This would probably be the main focus for future works.
An artificial dataset generator has been proposed to generate the training samples to train the different networks tried in the experiments. It is possible to specify an input error probability to generate data with primitive failures.

Major difficulties came from the dataset being artificially generated. This was a necessity since a public dataset that suited our type of tasks was not available and collecting this kind of data from robots is really resources intensive, time and money wise. The use of an artificial dataset can lead to difficulties about interpreting training and validation losses. For example, losses can be really close to each other because they are statistically the same. It is sometimes hard to determine if the model is overfitting the data and makes it hard to achieve better results.
The ability to generalize to task length has been evaluated using a test set with longer sequences. The ability to generalize to different task topology has been tested by generating a lower amount of input data instead. In this way, the actual (X,Y) values fed into the network during training time are actually different from the one used at test time.

Another key difficulty was the scarcity of state of the art regarding this specific topic: Deep Learning is usually applied on different type of data like images, sounds or words. The proposed work uses a high level representation of the environment and therefore requires to hand-engineer the features. Future work could be that of using directly images and let the network learn by itself what are the important features of the environment.

5.0.2 Future Works

There are a lot of starting points that are worth investigating and spending time on to build upon the results obtained in this work. The most important and promising ones are reported here:

- **Generalizing with Recursion**: The idea is to try to integrate the generalization via the recursion approach described in [38]. The model proposed in this work can be seen as a *sequence to sequence* that produces as output a list of instructions without recursion. Experiments lead to believe that a somewhat hierarchical and compositional structure of the data could really improve the generalization capabilities of the model.
  To make each action independent from the actual advancement of the sequence is needed with this approach. In this way, the action prediction results obtained in this work could be further improved, possibly reaching a stronger generalization.
  In order to do this, the network has to take as next input, along with the environment observation, the previous predicted action. This prediction can be either a primitive action or a high level action with its generated arguments. In this way, there is a decomposition of higher level tasks into finer tasks, until a primitive is reached.
  To implement this, a stack is needed to store the context of the network each time a new action is predicted (or invoked) and restore it each time it is completed, exiting its scope.
  This idea should also be combined with the probability of a primitive action failing and eventually with other types of feedback, for example from the motion planner.

- **Domain Specific Encoder**: In this work we provided as input a fixed high level representation of the environment that can be achieved using regular computer vision algorithms for object identification and pose estimation. This is in fact a limit to our generalization purpose because we would like to have flexibility regarding the number and types of objects in the scene.
  This can be achieved by using a neural network as *encoder* that can learn to represent the environment into a fixed size vector of features.
  This is also a prerequisite for the next point: meta-learning.

- **Meta-Learning**: As discussed in the state of the art analysis there is a trend in research that aims at achieving a more efficient learning.

This means that, rather than providing a lot of training examples of the same task, we can provide few training examples, even one for each task. This could avoid retraining the network each time a different task is selected by providing the task as input.
The training is done using a large number of different tasks but using few demonstrations of each one. The network will be able to generalize to entirely different and unseen *tasks* by just providing few or even one example as input.
On top of this, there is the real problem of collecting enough training data from real robots since a lot of resources (Robots, Time, GPUs) are needed.

- **Dataset Improvement and Statistical Analysis**: Different datasets could be artificially created for different, more complex tasks. Moreover the generation of the proposed dataset could be tweaked by generating training examples that are less similar between each other. This lets the network see a broader range of samples.

- **Better integration with Conditional Task Motion Planning**: CTMP was discussed as a starting point for this work. On other hand, the only motion related part of this work is the one about primitive failures. The proposed work could be further developed by a TMP point of view by choosing a more significant use case. For example, the feedback from the motion planner could be integrated as input to signal the network that a solution could not be found and expect the next prediction to be weighted also on this knowledge.
Moreover, a robot with two arms could be used to make the network predict which arm to use based its decisions on collision, shortest path and so on.

- **Deep Reinforcement Learning**: this point is more a possible different approach to the problem rather than an improvement. It would be useful to see which is the potential and limitations of this approach. A clear advantage would be the absence of a dataset, but at the same time the need to specify an adequate reward function could be a major problem.

Appendices

Appendix A

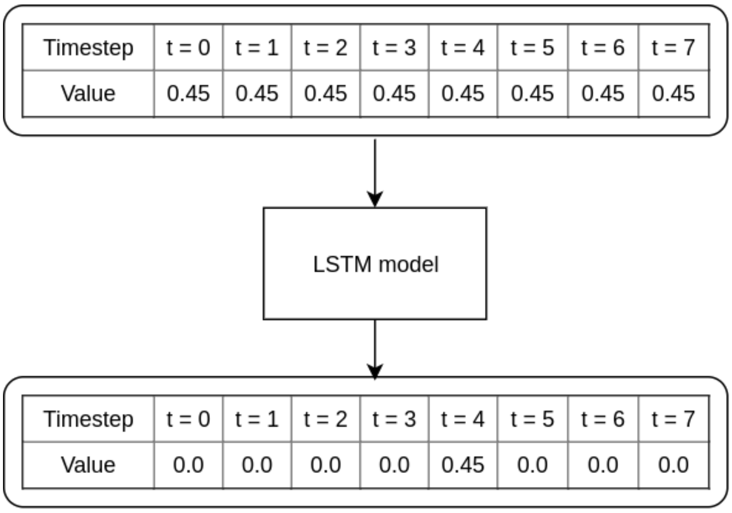
Preliminary Experiments

A.1 Simple Experiments

We chose to use LSTMs as our main building block to search if an implementation of a task planner could be possible. This was mainly influenced because of their ability to work with sequences and long term dependencies but also due to their growing popularity and consequent tools availability.
Since this is rather a difficult problem we had to investigate the networks behaviors in very minimal problems, like returning an input value after a certain number of timesteps, and trying to build up from there. These same experiments were also tested using GRUs networks that are really similar to LSTMs, but with less parameters. Unfortunately they did not provide any improvement and so LSTMs are presented in these experiments and were also selected for all of the work.

A.1.1 Echo - Fixed Input-Output Delay

In this experiment we try to make the network learn a simple delay, or *Echo*, of a continuous real number between 0 and 1 from input to output. The generated training set basically shows many sample sequences of the same length and with the same timestep delay but with different input values. Figure A.1 depicts a training example used for training.
This may seem a trivial experiment but one that cannot be taken for granted. The obtained results seem quite good: the network is able to echo the input number exactly after the learned timestep with two significant figures. The only problem arises in the low values of the parameter, i.e. between 0.00 and about 0.08 where the network tends to overshoot. This could be due to a low number of examples in that range.



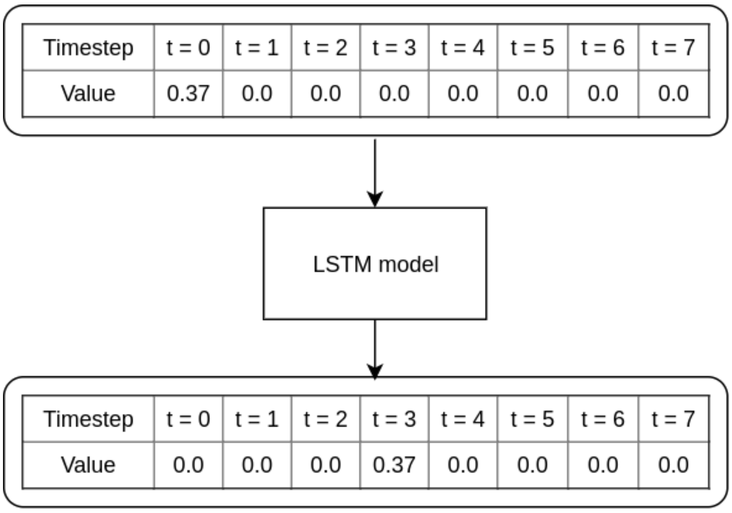
Timestep	t = 0	t = 1	t = 2	t = 3	t = 4	t = 5	t = 6	t = 7
Value	0.45	0.45	0.45	0.45	0.45	0.45	0.45	0.45

Timestep	t = 0	t = 1	t = 2	t = 3	t = 4	t = 5	t = 6	t = 7
Value	0.0	0.0	0.0	0.0	0.45	0.0	0.0	0.0

Figure A.1: A single training example for the Input - Output delay test repeating the input number at each timestep

During training, both the training and validation loss get low values in less than five epochs using 3000 training examples with the 60-20-20 split. The two loss values are quite similar because of the artificial nature of the dataset. Therefore, from a statistical point of view, the higher the number in the training set and validation set the higher the similarity.

A little bit harder variation of the previous experiments has been tried where the key difference is that the input number is fed to the network only once during the first timestep (Fig. A.2). Even though this variation introduces a little bit of complexity, it does not seem to affect the learning and the obtained results are comparable with the original experiment.
The biggest concerns with these results are the actual accuracies of the outputs, are they precise enough for our purposes? Since the numbers generated by the regression should be used as the target position for the robot end effector, we require to obtain an almost exact copy of the input. Even though the network learns the echo sequence, it introduces a significant error.



Timestep	t = 0	t = 1	t = 2	t = 3	t = 4	t = 5	t = 6	t = 7
Value	0.37	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Timestep	t = 0	t = 1	t = 2	t = 3	t = 4	t = 5	t = 6	t = 7
Value	0.0	0.0	0.0	0.37	0.0	0.0	0.0	0.0

Figure A.2: A single training example for the Input - Output delay test without repeating the input number

These experiments showed that the outputs almost always predict the first significant digit correctly and can often predict up to two or even three digits. Even though this seems great a lesser precision but higher consistency of the results would have been better for our purposes.

The main observation is that LSTMs, altough they do have memory, cannot be used to store a floating point number. The memory used by this kind of networks is more of a finite approximation of the previous weighted inputs and is the result of different interactions. An external memory cell would suit the this better.

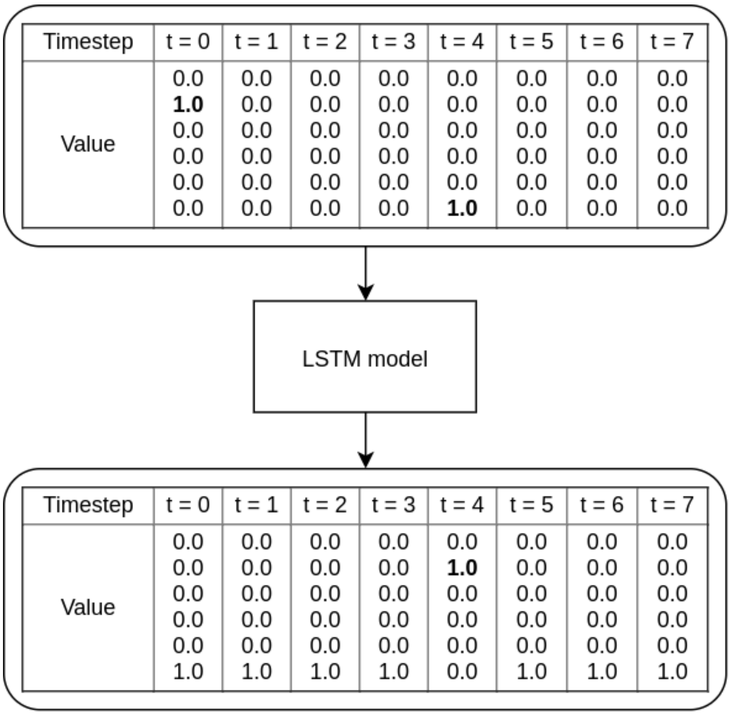
A.1.2 Discrete Echo - Fixed Input-Output Delay

We repeated the previous tests after switching to a discrete form for the output layer, using a softmax activation function. This was because the network proved that it is incapable of providing as output a value with high precision, hence a classification appeared to be more suitable to this kind of experiments.

This intuition was correct and as expected the network improved giving good results, moreover we were able to use the accuracy as a metrics that allowed us to know how many times the prediction is correct or not while the loss function gave us some hint on the performance of training. The network was able to achieve 100% of accuracy after very few epochs.

A.1.3 Discrete Echo - Variable Input-Output Delay

After the good results using a discrete output layer, in this case we try to make another step forward: the delay is not fixed and learned from the dataset anymore. The echo has to wait until an input trigger is given in input and then provide as output the memorized value. In Fig.A.3 a visual, more intuitive, representation of the experiment is presented.



Timestep	t = 0	t = 1	t = 2	t = 3	t = 4	t = 5	t = 6	t = 7
Value	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	0.0	0.0	0.0	0.0	**1.0**	0.0	0.0	0.0

Timestep	t = 0	t = 1	t = 2	t = 3	t = 4	t = 5	t = 6	t = 7
Value	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	0.0	0.0	0.0	0.0	**1.0**	0.0	0.0	0.0
	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	1.0	1.0	1.0	1.0	0.0	1.0	1.0	1.0

Figure A.3: A training sample for the variable echo experiment

This test has shown that LSTMs can reach 100% of accuracy after few epochs of training in this type of task.

Bibliography

[1] N. Dantam, S. Chaudhuri, and L. E. Kavraki, “The task motion kit,” 2016.

[2] L. M. de Moura and N. Bjorner, “Z3: An efficient smt solver,” in *TACAS*, 2008.

[3] I. A. Sucan, M. Moll, and L. E. Kavraki, “The open motion planning library,” *IEEE Robotics and Automation Magazine*, vol. 19, pp. 72–82, 2012.

[4] R. Fikes and N. J. Nilsson, “Strips: A new approach to the application of theorem proving to problem solving,” *Artif. Intell.*, vol. 2, pp. 189–208, 1971.

[5] M. Fox and D. Long, “Pddl2.1: An extension to pddl for expressing temporal planning domains,” *J. Artif. Intell. Res.*, vol. 20, pp. 61–124, 2003.

[6] J. Hoffmann and B. Nebel, “The ff planning system: Fast plan generation through heuristic search,” *J. Artif. Intell. Res.*, vol. 14, pp. 253–302, 2001.

[7] H. A. Kautz and B. Selman, “Unifying sat-based and graph-based planning,” in *IJCAI*, 1999.

[8] B. Siciliano and O. Khatib, *Springer Handbook of Robotics.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[9] F. Lagriffoul and B. Andres, “Combining task and motion planning: A culprit detection problem,” *I. J. Robotics Res.*, vol. 35, pp. 890–927, 2016.

[10] L. P. Kaelbling and T. Lozano-Pérez, “Integrated task and motion planning in belief space,” *I. J. Robotics Res.*, vol. 32, pp. 1194–1227, 2013.

[11] M. Gharbi, R. Lallement, and R. Alami, "Combining symbolic and geometric planning to synthesize human-aware plans: toward more efficient combined search.," *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 6360–6365, 2015.

[12] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

[13] T. Lozano-Perez and L. P. Kaelbling, "A constraint-based method for solving sequential manipulation planning problems," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014.

[14] S. Nedunuri, S. Prabhu, M. Moll, S. Chaudhuri, and L. E. Kavraki, "Smt-based synthesis of integrated task and motion plans from plan outlines," *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 655–662, 2014.

[15] C. R. Garrett, T. Lozano-Perez, and L. P. Kaelbling, "Ffrob: An efficient heuristic for task and motion planning," in *International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2014.

[16] L. M. de Moura and N. Bjorner, "Satisfiability modulo theories: introduction and applications," *Commun. ACM*, vol. 54, pp. 69–77, 2011.

[17] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, 2009.

[18] J. Pan, S. Chitta, and D. Manocha, "Fcl: A general purpose library for collision and proximity queries," *2012 IEEE International Conference on Robotics and Automation*, pp. 3859–3866, 2012.

[19] S. E. Reed and N. de Freitas, "Neural programmer-interpreters," *CoRR*, vol. abs/1511.06279, 2015.

[20] D. Bonanno, M. Roberts, L. Smith, and D. W. Aha, "Selecting subgoals using deep learning in minecraft: A preliminary report," 2016.

[21] C. Paxton, V. Raman, G. D. Hager, and M. Kobilarov, "Combining neural networks and tree search for task and motion planning in challenging environments," *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 6059–6066, 2017.

[22] M. Andrychowicz, M. Denil, S. G. Colmenarejo, M. W. Hoffman, D. Pfau, T. Schaul, and N. de Freitas, "Learning to learn by gradient descent by gradient descent," in *NIPS*, 2016.

[23] S. Hochreiter, A. S. Younger, and P. R. Conwell, "Learning to learn using gradient descent," in *ICANN*, 2001.

[24] D. Xu, S. Nair, Y. Zhu, J. Gao, A. Garg, L. Fei-Fei, and S. Savarese, "Neural task programming: Learning to generalize across hierarchical tasks," *CoRR*, vol. abs/1710.01813, 2017.

[25] Y. Duan, M. Andrychowicz, B. C. Stadie, J. Ho, J. Schneider, I. Sutskever, P. Abbeel, and W. Zaremba, "One-shot imitation learning," in *NIPS*, 2017.

[26] I. J. Goodfellow, Y. Bengio, A. C. Courville, and G. E. Hinton, "Deep learning," *Nature*, vol. 521 7553, pp. 436–44, 2015.

[27] W. S. Mcculloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity. 1943.," *Bulletin of mathematical biology*, vol. 52 1-2, pp. 99–115; discussion 73–97, 1990.

[28] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65 6, pp. 386–408, 1958.

[29] B. Widrow, "An adaptive adaline neuron using chemical memristors," 1960.

[30] P. J. Werbos, "Backpropagation through time: What it does and how to do it," 1990.

[31] F. A. Gers, J. Schmidhuber, and F. A. Cummins, "Learning to forget: Continual prediction with lstm," *Neural computation*, vol. 12 10, pp. 2451–71, 2000.

[32] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2015.

[33] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zhang, "Tensorflow: A system for large-scale machine learning," in *OSDI*, 2016.

[34] F. Chollet *et al.*, "Keras." `https://github.com/keras-team/keras`, 2015.

[35] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), pp. 2135–2135, ACM, 2016.

[36] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016.

[37] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[38] J. Cai, R. Shin, and D. X. Song, "Making neural programming architectures generalize via recursion," *CoRR*, vol. abs/1704.06611, 2017.