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

Federico Vendramin¹, Elisa Tosello¹, Nicola Castaman², and Stefano Ghidoni¹

Abstract—This work proposes a Long Short-Term Memory network able to learn robot task planning primitives when solving TAMP problems. Given a task to a robot, the network is able to select the appropriate sequence of actions to be performed, along with their arguments, while facing environment uncertainty and action/motion failures. Proposed tests ask a robot to swap two cubes on a table. Focusing on this task assignment, an expert policy is proposed able to generate an artificial dataset useful for training. Accuracy of action selection up to 97% was obtained on sequences of the length of the training ones; an accuracy up to 91%, instead, was obtained on longer task sequences.

I. INTRODUCTION

This paper aims to solve Task and Motion Planning (TAMP) problems by using Deep Neural Networks (DNNs). Specifically, it focuses on Robot Learning task planning primitives by means of Long Short-Term Memory (LSTM) networks [1].

Traditional TAMP algorithms lack of generality, scalability, and completeness. They are not able to handle environment uncertainty or motion failures when executing primitive actions (e.g., a person moving in the robot workspace or an object slipping from the robot end-effector). The attempt of this work is facing these limitations by introducing Artificial Intelligence (AI). Questions authors are trying to answer follow. Can NNs be exploited to learn a task and produce an online planner by selecting the correct action to be performed? Will this planner be able to react to changes in the environment or unexpected failures without recomputing, each time, the entire plan? Will it be able to learn how to change the task execution according to the encountered motion failures? Based on these open questions, faced challenges and contributions of this work follow.

A. Challenges

• **Online Planning and Execution**: the main goal is to use neural networks to explore the feasibility of an online task planner. The network should be able to reason about the environment and react to unexpected changes;
• **Learning**: since neural networks are trained end to end, authors aim to implement a network able to learn with supervision how to predict the next action necessary to achieve a specific task;
• **Dataset**: a public dataset to use for training the network is lacking. Authors aim to face this lack knowing that data collection could be resource expensive if done in an inefficient way.

B. Contributions

An attempt of agent able to learn how to reason on assigned tasks is proposed. This agent is able to correctly handle environment observations by producing a sequence of actions that fulfill the assigned task. This sequence can be seen as a program that invokes robot motion primitives. Tests focus on a toy example: the swap of two cubes on a table. Experiments were executed using an artificial environment generated by an expert policy. This environment depicts the position of the objects and the gripper status (full or empty).

• **Online Planning and Execution**: the proposed model can detect robotic manipulation failures such as unsuccessful grasps or wrong object placings. It reacts to these action failures by selecting those actions that restore the right action preconditions and complete the task;
• **Learning**: accuracy of up to 97% has been reached when evaluating the network on sequences of the length of the training ones (with maximum sequence length equals to maxlen = 30). Accuracy of up to 91% has been reached while evaluating the network on sequences of higher length (maxlen = 100), even if these sequences have been encountered for the first time;
• **Dataset**: an artificial dataset generator is proposed able to produce the synthetic examples necessary for training.

II. RELATED WORK

Various researchers focus on the resolution of TAMP problems. Initial works investigated on a Hierarchical architecture that solves task and motion planning problems in isolation: the task planner decides what actions to do and the motion planner decides how to realize these actions. An example is the Hierarchical Planning in the Now (HPN) [2] approach: it interleaves planning and execution reducing search depth but requiring reversible actions when backtracking.

The main issue is that the environment is not deterministic: task and motion planners should cooperate so that only geometrically valid plans are sent to the executor. Task and motion planning integration has then been explored by implementing approaches able to search in the hybrid...
task-geometry space and distribute search across the two spaces. [3], for example, extends a hierarchical task planner with geometric primitives, using shared literals that relate task-level symbols with motion-level geometric entities. [4] interfaces off-the-shelf task and motion planners using a heuristic to remove objects that potentially block the robots path. The Robosynth framework [5] uses a Satisfiability Modulo Theories (SMT) solver ([6], [7]) to generate task and motion plans from a static roadmap, employing plan outlines to guide the planning process. FFRobot [8] develops an FF-like [9] task layer heuristic based on a lazily-expanded roadmap. Finally, the Task-Motion Kit [10] proposes a Iteratively Deepened Task and Motion Planning (IDTMP) method that uses a constraint-based task planner to compute different candidate plans. In detail, the Z3 [11] SMT solver is used as task planner: it generates alternative task plans maintaining a stack of constraints and performing repeated satisfiability checks as constraints are pushed onto and popped from the stack. The Open Motion Planning Library (OMPL) [12] is used for motion planning and the Flexible Collision Library (FCL) [13] is used for collision checking.

Combining task and motion planning in this way still do not guarantee generality, completeness, and scalability. The aforementioned algorithms, for example, are not able to face the environment uncertainty. One way to solve these limitations involves the introduction of AI. An example is the Neural Programmer-Interpreter (NPI) [14] network: a recurrent and compositional neural network that learns how to represent and execute programs. It has three learnable components: a task-agnostic recurrent core, a persistent key-value program memory, and a domain specific encoder that enables a single NPI to operate in multiple perceptually different environments with distinct affordances. The core is a LSTM network that at each time step selects the next program to run conditioned on the current observation of the environment. Another example is [15], which selects sub-goals using Deep Learning in Minecraft. It interleaves perception, goal, reasoning, and acting with the visual observations of the environment captured by the in-game characters camera. Basic motion primitives compose the action plan with respect to four sub-goals: walking forward, creating stairs, removing obstacles, and bridging obstacles. The sub-goals selection is done by a CNN trained on the environment observations. Finally, [16] 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.

To guarantee the greatest possible generalization, Meta-Learning can be introduced. It allows to train the model by exploiting very few significant samples of a multitude of different tasks. An example is Neural Task Programming (NTP) [17]: a robot learning framework based on NPI. This tool has been tested on robotic manipulators for tasks like blocks stacking and sorting. NTP interprets a task specification and instantiates a hierarchical policy as a neural program, where the lowest level of the hierarchy is made of primitive actions executed by the robot API.

III. LONG SHORT-TERM MEMORY NETWORKS

Authors decided to implement the Neural Network by means of Long Short-TERM Memory (LSTM) networks [1]. LSTMs, in fact, solve the vanishing and exploding gradient problem of other NNs: the output of a LSTM depends only on the memory status and on the current input. Moreover, LSTMs can remember long-term interactions by maintaining an internal state which depends on previous inputs. Focusing on task planning, this means that LSTMs let users work on temporal sequences, letting a cognitive choice of the actions to be performed.

A LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell is responsible for “remembering” values over arbitrary time intervals; hence the word “memory” in LSTM. Each of the three gates can be though of as a “conventional” artificial neuron that computes an activation (using an activation function) of a weighted sum. Hence the notation of “gate”: every gate controls the flow of values that goes through the connections of the LSTM. In this way, LSTMs learn how to control the flow of information inside the cell and choose what to remember and what to forget.

Formally, the state $s_i$ of the $i$th cell at time $t$ is equals to the sum of the input gate and the forget net. The input gate $g_i$ can be defined as:

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

where $x_j^t$ is the input vector and $h_{j-1}^t$ is the hidden state vector at the previous step. In order to compute $g_i$, Gating is performed on both $x_j^t$ and $h_{j-1}^t$: $x_j^t$ is multiplied by $U^g$, the matrix of input weights; $h_{j-1}^t$ is multiplied by $W^g$, the matrix of recurrent weights, $g_i^t$ is the sigmoid of the sum of these amounts and the bias $b_i^g$. Similarly, the forget gate $f_i$ is defined as:

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

The internal state can now be defined as:

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

Authors highlight that no multiple multiplications by the same operation $W^f$ is carried out: the recurrent update is given by the sum of only two components.

The output vector $h_i^t$ is consequently obtained by:

$$h_i^t = \tanh(s_i^t q_i)$$

$$q_i^t = \sigma(b_i^q + \sum_j U_{i,j}^q x_j^t + \sum_j W_{i,j}^q h_{j-1}^t)$$
IV. OUR USE CASE AND NEURAL NETWORK

A. Use Case

A robot is asked to swap two cubes, $A$ and $B$, on a table (see Figure 1): it should pick $A$ from its start position $(x_A, y_A)$, place it on a temporary location $(x_{\text{TEMP}}, y_{\text{TEMP}})$, pick $B$ from $(x_B, y_B)$, place it on $(x_A, y_A)$, pick $A$ again and place it on $(x_B, y_B)$ (see Figure 2). The table is discretized as a chessboard where cubes can be placed in the center of every chessboard cell. Learning this task means: learning the sequence of actions to be performed in order to achieve the assigned goal (see Listing 1) and learning how to recover from failures. Indeed, an action can fail, e.g., the grasped object can slip from the robot’s gripper or can be wrongly placed. This implies that a recovery routine has to be performed in order to restore the right action preconditions (see Listing 2).

Listing 1: The actions sequence of the swapping task when no failure is perceived.

```
begin
PickA ();
Place (x_{\text{TEMP}}, y_{\text{TEMP}});
PickB ();
Place (x_A, y_A);
PickA ();
Place (x_B, y_B);
end ;
```

Listing 2: The actions sequence of the swapping task when the robot fails to pick and place cubes: a rerun has to be performed in order to recover the correct scene.

```
begin
PickA (); // Failed!
PickA (); // Rerun
Place (x_{\text{TEMP}}, y_{\text{TEMP}});
PickB ();
Place (x_A, y_A); // Failed!
PickB (); // Pick B again, but this fails too!
PickB ();
Place (x_A, y_A);
PickA ();
Place (x_B, y_B);
end ;
```

B. Dataset

Neural Networks need a dataset of examples to learn from. In the case in analysis, acquiring this data could be challenging. It is why authors propose a Dataset Generator with an expert policy that generates useful synthetic data composed of examples in which the assigned task is completed, managing any failures.

In detail, the expert policy generates a sequence of Input-Output pairs called Environment and Output, respectively. Environment is the observed high level encoding of the environment; Output is the desired output of the network. The description of the environment is composed by the coordinates $(x_A, y_A)$ and $(x_B, y_B)$ of the two cubes, the status of the gripper (1 if full, 0 if empty), and the temporary swap position $(x_{\text{TEMP}}, y_{\text{TEMP}})$. Each coordinate is a single input to the network. In a continuous domain, this would correspond to the Input Environment of Figure 3, where every coordinate is in the range 0 - 1. Performed tests demonstrated that the network is more performing in a discrete domain (the chessboard of Section IV-A). In this case, each coordinate is represented by 10 inputs (10 for the $x$ coordinates and 10 for the $y$ ones) that are either 1 or 0 in order to reflect if the corresponding cell of the chessboard is or is not occupied. The output is composed of the sequence of Actions to be performed together with their Arguments, i.e., the temporary swap position.

The generator is able to introduce an error probability that simulates the failure of an action. E.g., the action of picking a cube can fail, meaning that the cube falls from the robot gripper. In this case the robot needs to pick up the object again. This action should be introduced in the sequence.
Fig. 3: A continuous training example. Input: \((x_A, y_A, x_B, y_B, \text{gripper status})\), and the temporary swap position \((x_{TEMP}, y_{TEMP})\). Output: the action to be performed together with the \((x, y)\) arguments of the place primitive.

Therefore, authors generated the dataset by introducing a 20% failure probability for each action.

C. Training

Once generated the dataset, it is used to train the network. This means that every \(i\)-th training example has the form

\[
\begin{align*}
\{\text{In}; \text{Out}\}_i, & \quad \text{In}_i = \{in_1, in_2, ..., in_n\}, \\
\text{Out}_i = \{out_1, out_2, ..., out_o\}, & \quad \forall t \in [0, \text{maxlen}] 
\end{align*}
\]

where \(\{\text{In}; \text{Out}\}_i\) is a couple of two sequences: \(\text{In}\) is the sequence of \(n\) features representing the environment observations; \(\text{Out}\) is the sequence of \(o\) features describing the target output. Every sequence has a \(\text{maxlen}\) maximum length.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Examples</td>
<td>3000</td>
</tr>
<tr>
<td>Batch Size</td>
<td>4</td>
</tr>
<tr>
<td>Epochs</td>
<td>Up to 150</td>
</tr>
<tr>
<td>Optimizer</td>
<td>ADAM</td>
</tr>
<tr>
<td>Loss Function</td>
<td>Crossentropy</td>
</tr>
</tbody>
</table>

D. The implemented Neural Network

Let \(e_t \in E\) be the environment observation at time \(t\) and let \(s_t \in \mathcal{R}^D\) be its fixed-length state encoding extracted by a domain specific encoder \(f_{\text{enc}} : E \rightarrow \mathcal{R}^D\). The network takes the environment representation \(s_t\) as input, possibly along with other inputs like motion planner failures, and computes hidden states \(h_t \in \mathcal{R}^M\), which sum up the progression of the task in the current environment. Three LSTM hidden layers of 50 units each compose the Neural Network (see Figure 4). Three are the output layers: - the next action to be performed, computed as \(f_{\text{Act}} : \mathcal{R}^M \rightarrow A\), with \(A\) the finite set of actions; - the temporary position \(x_{TEMP}\); - the temporary position \(y_{TEMP}\). \(f_{\text{Arg}} : \mathcal{R}^M \rightarrow \mathcal{R}^k\) is used to compute the \((x_{TEMP}, y_{TEMP})\) arguments in the selected discrete domain, with \(\mathcal{R}^k\) a set of \(k\) real numbers (in a discrete domain, the problem in analysis is reduced to a classification problem). After the computation, the Softmax activation function is used to obtain a \(k\)-dimensional vector with values in \([0,1]\) whose sum is equal to 1. The argument with maximum value is selected as output.

E. Results

The network is able to efficiently recover from failure (see Figure 5).

Accuracies of 97.3% and 99.5% are reached on the validation sets for the action layer and the argument layers, respectively (see Figure 6a and 6b).

In Figure 7a the training and validation losses of the action layer are reported. Immediately after the first 15 epochs, the validation loss starts to deviate from the training loss.
meaning that the model is beginning to overfit. Anyway, the validation accuracy does not get worse. This is because even though the prediction is a little bit further from the target value, Softmax selects the maximum output turning on an output that is still correct. The spike observed after 120 epochs is due to the small size of the batch. It is also due to the changes of the learning rate introduced by ADAM. Using a larger batch would smooth the losses. The training of the parameters for the argument layers, instead, is smooth (see Figure 7b).

**F. Platform Specifications**

Experiments have been implemented in Keras [19]. Tensorflow [20] is used as back-end. They are performed on a machine with 8GB DDR3 memory and an Intel i5 2500k at 4.2GHz CPU with Ubuntu 16.04 Operating System. Tensorflow was compiled from source to take advantage of the SSE instructions and consequently obtain a performance improvement. In this way, the training times range from few minutes for those cases with hundreds of parameters, up to 2 or 3 hours if thousands of parameters are exploited. Tests showed a performance deterioration when using Tensorflow with the GPU support (nVidia GTX 970). This is due to the low batch size and the low cardinality of the input set: they do not lead to a speedup factor sufficient to improve the training time. Thus, CPU is preferred.

Experiments have been performed in simulation; Gazebo [21] is used as simulator and the implementation is based on the Robot Operating System [22] (Kinetic version).

**V. Conclusion**

This paper proposed a three layers LSTM Neural Network able to learn a manipulation task, identify failures on action execution, and recover from failures so to complete this task. An artificial dataset, equipped with synthetic data, has been generated and presented to train the network. As future work, authors aim to improve the generalization capabilities of the proposed network by integrating it with the generalization via recursion approach described in [23]: this approach subdivides complex tasks into simpler ones and applies recursion so to improve the generalization. Another idea is that of creating a Domain Specific Encoder: an encoder able to represent the main features through a finite vector of elements. Indeed, in this work a fixed high-level representation of the environment is provided as input to the NN: a table with two cubes on its top where cubes identification and pose estimation can be computed by using computer vision algorithms. To make this representation more general, objects in the scene should be, for example, of different type, shape, and number. This can be achieved by using a neural network as encoder that can learn to represent
In this way, users avoid retraining the network each time a different task is assigned, and the network can efficiently generalize on new, first-time seen tasks.

**REFERENCES**


[19] F. Chollet et al., “Keras,” [https://keras.io](https://keras.io), 2015.


