Published at the Workshop on Machine Learning Methods for High-Level Cognitive Capabilities in Robotics held in conjunction with the International Conference on Intelligent Robots and Systems (IROS), 2016.

Learning Vector Symbolic Architectures for Reactive Robot Behaviours

Peer Neubert, Stefan Schubert and Peter Protzel

***Abstract*— Vector Symbolic Architectures (VSA) combine a hypervector space and a set of operations on these vectors. Hypervectors provide powerful and noise-robust representations and VSAs are associated with promising theoretical properties for approaching high-level cognitive tasks. However, a major drawback of VSAs is the lack of opportunities to learn them from training data. Their power is merely an effect of good (and elaborate) design rather than learning. We exploit high-level knowledge about the structure of reactive robot problems to learn a VSA based on training data. We demonstrate preliminary results on a simple navigation task. Given a successful demonstration of a navigation run by pairs of sensor input and actuator output, the system learns a single hypervector that encodes this reactive behaviour. When executing (and combining) such VSA-based behaviours, the advantages of hypervectors (i.e. the representational power and robustness to noise) are preserved. Moreover, a particular beauty of this approach is that it can learn encodings for behaviours that have exactly the same form (a hypervector) no matter how complex the sensor input or the behaviours are.**

I. INTRODUCTION

A Vector Symbolic Architecture (VSA) [1] combines a hypervector space with a set of operations on these vectors. Hypervectors are dense or sparse vectors with very high numbers of dimensions (e.g. 10,000). Hypervectors have great representational power and facilitate high robustness to noise, even if the vectors are binary and sparse [2] (although they may also be dense and real-valued). Several approaches have been proposed to combine vector spaces with appropriate operations to define a VSA, e.g. [3], [4], [5], [6]. VSAs are known to have promising theoretical properties and have been argued to be an approach to general artificial intelligence [7]. They can encode role/filler pairs [3] and address Jackendoff's challenges for cognitive neuroscience [1]. Hypervectors and VSAs are used for example at Numenta's Hierarchical Temporal Memory [8], for modelling associative long short-term memory [9] and for reactive robot control [10].

In [1, p.6], Gayler points out a major issue with VSAs: "Typical connectionist architectures rely on training procedures to achieve their effectiveness. However, VSAs provide no opportunity for training to substitute for architectural effectiveness. That is, good performance depends on good design rather than automated training, and this is a harder research task.".

In this paper we address this issue and present a methodology to learn VSAs that encode and execute reactive behaviours. In particular, we demonstrate the approach by learning a VSA to solve the robot navigation task described in [10] (which is there solved by a manually designed VSA). We exploit high level knowledge about the task (similar to a robotics prior [11]) to learn a navigation policy from demonstration of successful navigation runs.

The authors are with Technische Universität Chemnitz, Germany firstname.lastname@etit.tu-chemnitz.de

II. THE VECTOR SYMBOLIC ARCHITECTURE

We build upon the Multiply-Add-Permute (MAP) architecture of [6]. Our VSA consists of the following elements:

- Each **hypervector** is an element of $[-1, 1]^d$ with d is set to 4,000 in our experiments.
- **Distances** of hypervectors are computed using the cosine-distance.
- The **bind()** operator $\otimes$ is the element-wise product.
- The **bundle()** operator $\oplus$ is the element-wise sum. We normalize the values by limiting each vector entry independently to range $[-1, 1]$.
- Permutation of vector dimensions can be used to quote information [6]. We define a **protect()** operator $\odot$, which applies a circular right shift on the vector elements. The width of the shift is computed from the vector itself (i.e. as average index of elements > 0) and its hypervector representation is superposed with the shifted input vector using the bundle() operator. This allows to invert the permutation by extracting and decoding the width of the shift from this superposition.
- **Encoding/decoding**: In our preliminary experiments, we use a very simple coding of sensor and actuator values: We encode scalar values by assigning random hypervectors. These assignments are stored in a lookup table (LUT). This allows to encode similar input values with the same hypervector and to later decode the vectors.
- **Clean-up memory**: A similar LUT is used to implement the clean-up memory that is required to denoise hypervectors. Again this is a tribute to the preliminary state of this work, for larger problems, this could be implemented, e.g., in form of recurrent associative networks.

For details on the properties of these operations (which are essential for this work), we refer the reader to [12].

III. LEARNING REACTIVE BEHAVIOURS

It is widely accepted that a key for successful application of machine learning techniques is exploitation of inductive bias [13]. In the area of mobile robotics, this may be structural knowledge of the problem to solve, the robot capabilities and the properties of the world the robot is acting in, e.g incorporated in form of robotics-priors [11]. We exploit the fact that reactive behaviours can be formalized

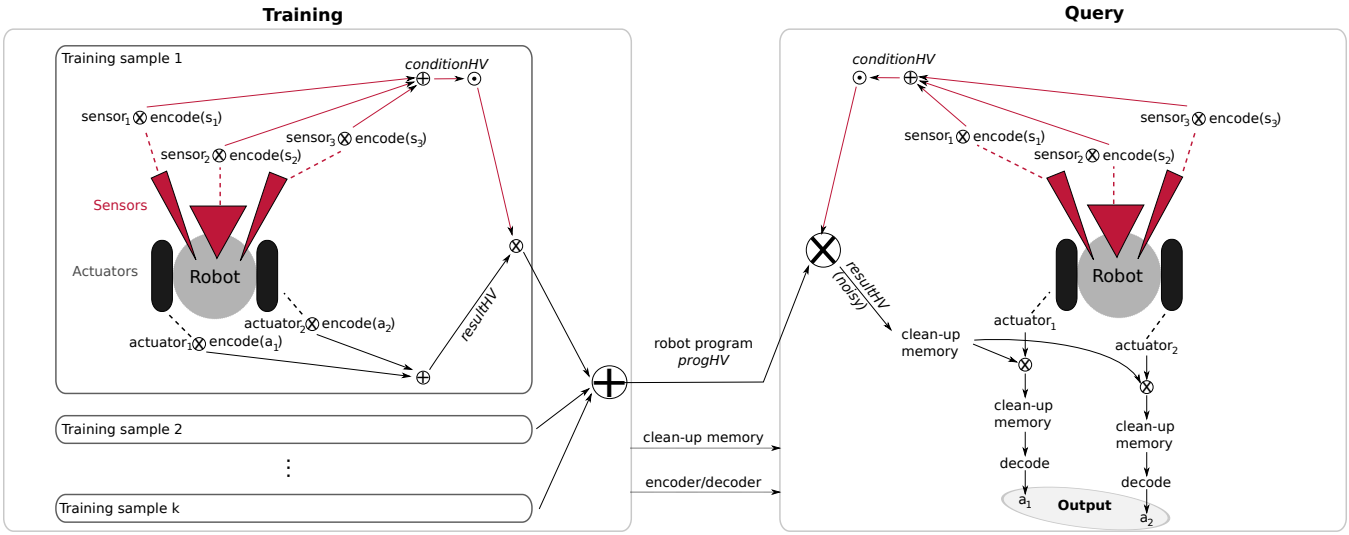

Fig. 1. Schematic overview of data flow during learning and query phase for the simulated robot used in the experiments. The robot sensors and their data flow are shown in red. In the training phase, the sensor and actuator combinations that are presented in the training samples are combined to a single hypervector representation *progHV*. This hypervector connects the training stage to the execution phase. During the latter, the hypervector is queried for the current sensor representation and the corresponding set of actuator commands (a_1, a_2) is returned.

as combination of *(condition, result)* or *(input, output)* pairs, respectively. Encoding a reactive behaviour means encoding these pairs (hopefully with good interpolation properties and robustness to noise) and executing such a behaviour means querying this encoding with the current *condition* (e.g. the vector of sensor inputs) and obtaining the according *result* (e.g. the motor velocities). Based on this knowledge, we can design a VSA that can be trained to resemble reactive behaviours from a training set. The overall system is illustrated in Fig. 1.

The training procedure is outlined in Algorithm 1. The training data is a set of $j = 1 : k$ pairs of sensor inputs $S_j = s_1, ..., s_n$ and actuator outputs $A_j = a_1, ..., a_m$ for the n sensors and the m actuators. Based on the VSA and encoder of section II, a single hypervector $progHV$ is computed that encodes the behaviour shown in the training sequence. Line 1 computes a hypervector for each sensor and actor that is used to encode this role in lines 3 and 4 (a hypervector "name" of this sensor/actuator). For each training sample, the condition hypervector is computed as bundle of bindings of sensors and their values (line 3). The result hypervector is computed similarly in line 4. The query procedure of line 5 evaluates whether the current conditions are already known. This computation build upon the hypervector arithmetic property that each member of a bundle of hypervectors is similar to this bundle, while other (random) hypervectors have a probability close to 1 of being dissimilar. If these conditions are unknown, the program is extended with the binding of the current condition and results (line 6) and the conditions are bundled with the known conditions (line 7). Lines 8-11 add the result hypervector and its parts to the clean-up memory to allow later reconstruction during the execution of the behaviour.

Executing the learned behaviour means querying the $progHV$ for a result given the current conditions. In hypervector arithmetic this can be done by unbinding the conditions from the program. This results in the hypervector of the corresponding result with some noise. For details see [12]. To remove the noise, the result is run through the clean-up memory:

$$resultHV := VSA.cleanUp(\odot(conditionHV) \otimes progHV) \quad (1)$$

Similarly, the individual actuator values can be obtained by unbinding the actuator names (computed in line 1 in Algorithm 2) from the resultHV. Details are given in Algorithm 2.

Algorithm 1: VSA learning

Data: k training samples $[S, A]_{1:k}$ of sensor and actuator values, a VSA, an encoder, an empty program progHV and an empty hypervector knownCondHV of known conditions
Result: progHV - a hypervector representation of the behaviour

```
// get hypervector representations for each
   sensor and actor
1  [sensor, actuator] = VSA.assignRandomHypervectors()
// for each training sample [S, A]
2  foreach pair [S = (s_1, ..., s_n), A = (a_1, ..., a_m)] do
       // encode values, bind to device and bundle
          condition/result
3      conditionHV := ⊕_{i=1}^{n} (sensor_i ⊗ encode(s_i))
4      resultHV := ⊕_{i=1}^{m} (actuator_i ⊗ encode(a_i))
5      if isDissimilar(knownCondHV, ⊙(conditionHV)) then
           // protect the condition and append
              (bundle) to the program
6          progHV := progHV ⊕ (⊙(conditionHV) ⊗ resultHV)
           // also append (bundle) the condition to
              the set of known conditions
7          knownCondHV :=
           knownCondHV ⊕ (⊙(conditionHV))
           // insert the result and the actuator
              encoding to the clean-up memory
8          VSA.addToCUM(resultHV)
9          foreach actuator_i do
10             VSA.addToCUM(actuator_i ⊗ encode(a_i))
11         end
12     end
13 end
```

Algorithm 2: VSA query

Data: $progHV$ - the output of the learning procedure Alg. 1, the VSA and encoder/decoder used in Alg. 1, the query sensor inputs S
Result: output actuator commands A

// encode values, bind to device and bundle condition
$conditionHV := \oplus_{i=1}^{n}(sensor_i \otimes encode(s_i))$
// query program to get a noisy version of the resultHV
$resultHVNoisy := \odot(conditionHV) \otimes progHV$
// remove noise
$resultHV := vsa.queryCUM(resultHVNoisy)$
// for each actuator , extract the command from the result hypervector
foreach $actuator_i$ **do**
  // unbind a noisy version from the result hypervector
  $commandHVNoisy := actuator_i \otimes resultHV$
  // remove noise
  $commandHV := vsa.queryCUM(commandHVNoisy)$
  // decode the command value from the hypervector
  $a_i := decode(commandHV)$
end

IV. Proof-of-Concept Experiments and Discussion

We implemented the VSA of section II and the described training procedure in Matlab and ran a set of proof-of-concept experiments using a V-REP simulation [14] of the textbook corral-escape task of [10]. Fig. 2 shows the simulated environment with the robot. The robot has a differential drive and is equipped with two light sensors (left/right) and a front-facing distance sensor. Initially, the robot is placed inside the corral and its task is to find the light source (the yellow disk in Fig. 2) and stay there. There are several solutions for this task, e.g., the following set of if-else rules can be used [10]:

```
if senseLightLeft and senseLightRight:
   leftMotor, rightMotor = +1,+1 // stay in light
else if senseLightLeft: // turn left
   leftMotor, rightMotor = -1,+1
else if senseLightRight:
   leftMotor, rightMotor = +1,-1 // turn right
else if senseObstacle: // turn left
   leftMotor, rightMotor = -1,+1
else: // cruise
   leftMotor, rightMotor = +1,+1
```

Levy et al. [10] manually designed a VSA that implements this set of rules. In contrast, we used this hard coded if-else behaviour to generate training sequences to *learn* a VSA using Algorithm 1. While the above listed if-else rules only use binary values, the sensor inputs of the training sequences produced by the simulation environment are real valued scalars (i.e. the distance to the light source or an obstacle) and are handled as such in Algorithms 1 and 2. In our experiments, the learned single hypervector $progHV$ could (together with the encoder and the clean-up memory) successfully resemble the results of the training behaviour.

We believe that learning VSAs is a worthwhile direction to exploit their promising theoretical high-level properties. However, the here presented approach is only a first step. Currently, the generalization and smoothing of training data is just based on the similarity properties of hypervectors and their encoding. How can this be better supported by the learning procedure? To what extend is this VSA implementation robust to partially observable states and how can this be improved? In this particular setting, what is the representation capacity of this single hypervector behaviour? What are good policies to combine several of these behaviours?

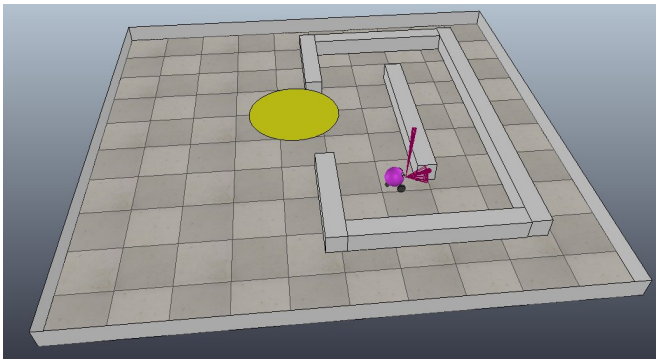

Fig. 2. The V-REP simulation environment. The robot has two light sensors and can additionally detect obstacles in front of the robot. The goal of the task is to navigate to the yellow light source and stay there.

References

[1] R. W. Gayler, "Vector symbolic architectures answer jackendoff's challenges for cognitive neuroscience," in *Proc. of ICCS/ASCS Int. Conf. on Cognitive Science*, Sydney, Australia, 2003, pp. 133–138.
[2] S. Ahmad and J. Hawkins, "Properties of sparse distributed representations and their application to hierarchical temporal memory," *CoRR*, vol. abs/1503.07469, 2015. [Online]. Available: http://arxiv.org/abs/1503.07469
[3] P. Smolensky, "Tensor product variable binding and the representation of symbolic structures in connectionist systems," *Artif. Intell.*, vol. 46, no. 1-2, pp. 159–216, Nov. 1990.
[4] P. Kanerva, "Fully distributed representation," in *Proc. of Real World Computing Symposium*, Tokyo, Japan, 1997, pp. 358–365.
[5] T. A. Plate, "Distributed representations and nested compositional structure," Ph.D. dissertation, Toronto, Ont., Canada, Canada, 1994.
[6] R. W. Gayler, "Multiplicative binding, representation operators, and analogy," in *Advances in analogy research: Integr. of theory and data from the cogn., comp., and neural sciences*, Bulgaria, 1998.
[7] S. D. Levy and R. Gayler, "Vector symbolic architectures: A new building material for artificial general intelligence," in *Proc. of Conference on Artificial General Intelligence*. Amsterdam, The Netherlands: IOS Press, 2008, pp. 414–418.
[8] J. Hawkins, S. Ahmad, and D. Dubinsky, "Cortical learning algorithm and hierarchical temporal memory," pp. 1–68, 2011, Numenta Whitepaper.
[9] I. Danihelka, G. Wayne, B. Uria, N. Kalchbrenner, and A. Graves, "Associative long short-term memory," *CoRR*, vol. abs/1602.03032, 2016. [Online]. Available: http://arxiv.org/abs/1602.03032
[10] S. D. Levy, S. Bajracharya, and R. W. Gayler, "Learning behavior hierarchies via high-dimensional sensor projection," in *Proc. of AAAI Conference on Learning Rich Representations from Low-Level Sensors*, ser. AAAIWS'13-12, 2013, pp. 25–27.
[11] R. Jonschkowski and O. Brock, "Learning state representations with robotic priors," *Autonomous Robots*, vol. 39, no. 3, pp. 407–428, 2015.
[12] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors." *Cognitive Computation*, vol. 1, no. 2, pp. 139–159, 2009.
[13] T. M. Mitchell, "The need for biases in learning generalizations," Rutgers University, New Brunswick, NJ, Tech. Rep., 1980.
[14] [Online]. Available: http://www.coppeliarobotics.com/