Unsupervised Learning of Temporal Abstractions with Slot-based Transformers

Anonymous Author(s)
Affiliation
Address
email

Abstract

The discovery of reusable sub-routines simplifies decision-making and planning in complex reinforcement learning problems. Previous approaches propose to learn such temporal abstractions in a purely unsupervised fashion through observing state-action trajectories gathered from executing a policy. However, a current limitation is that they process each trajectory in an entirely *sequential* manner, which prevents them from revising earlier decisions about sub-routine boundary points in light of new incoming information. In this work we propose SloTTAr, a fully parallel approach that integrates sequence processing Transformers with a Slot Attention module for learning about sub-routines in an unsupervised fashion. We demonstrate how SloTTAr is capable of outperforming strong baselines in terms of boundary point discovery, while being up to 30x faster on existing benchmarks.

1 Introduction

An intelligent goal-seeking agent situated in the real world should make decisions along various timescales to act and plan efficiently. A natural approach that facilitates this behavior is via a divide-and-conquer strategy, where goals are decomposed along sub-goals, and solutions to such sub-goals (i.e. 'correct' sequences of actions) are stored as reusable 'primitive' *sub-routines* [9, 4, 35]. Viewing complex goal-directed behavior as (novel) compositions of known sub-routines simplifies both decision-making and planning [36, 43], while also benefiting out-of-distribution generalization [32].

The options framework [42] introduced a design strategy for the hierarchical organization of behavior within the context of reinforcement learning. Crucial to its success is the quality of each 'option' (or sub-routine) in terms of the level of modularity and reusability it offers. Consider for example, the task of planning a travel itinerary to go from London to New York. Useful sub-routines in this context may include purchasing flight tickets, navigating to the airport, or boarding the flight. This factorization into sub-routines is desirable as these sub-routines capture mostly self-contained activities. Thus they are far more likely to apply to other travel plans between two different cities, which may again involve taking flights, etc.

Prior approaches propose to address this issue by learning about useful sub-routines directly from data [2, 37, 23, 27]. Of particular interest is the fully unsupervised setting, where the learner is only given access to state-action trajectories from an (expert) policy. Two relevant works can be distinguished: CompILE [23] and OMPN [27]. In CompILE expert trajectories are modeled using an RNN-based latent variable model that infers a pre-determined number of boundary points by iteratively processing the entire trajectory multiple times to recover segments associated with reusable sub-routines. Alternatively, OMPN equips an RNN with a multi-layer hierarchical memory, where information processing at different levels in the memory can be interpreted as belonging to separate segments. While OMPN additionally includes a strong preference for capturing hierarchical relationships between different sub-routines, both methods are limited insofar they process the

Submitted to NeurIPS 2021Workshop on Deep Reinforcement Learning. Do not distribute.

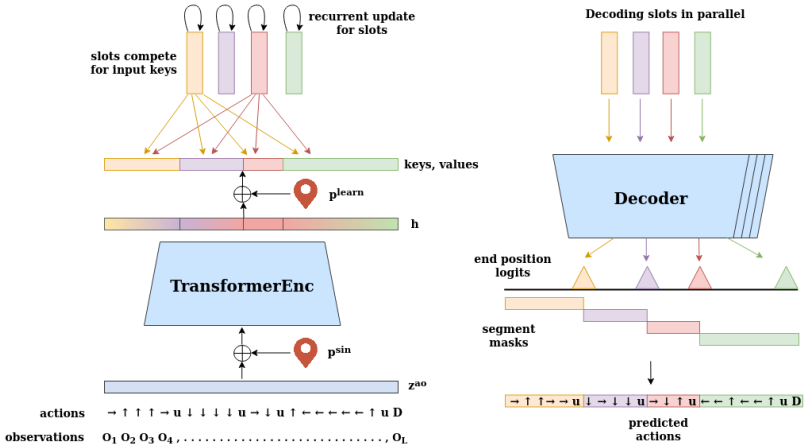

Figure 1: SloTTAr comprises of 3 modules: learning spatio-temporal features of action-observation sequences (*TransformerEnc*), learning a similarity-based grouping of actions to their respective sub-routines through computing slot-based representations (*Slot Attention* [26]), and decoding slots to end positions and action segments of the sub-routine they capture (*Decoder*).

trajectory in an entirely *sequential* manner. This prevents them from revising earlier decisions about boundary points in light of new information that becomes only accessible at a future stage. Moreover, iteratively processing the sequence multiple times (as in CompILE) or interfacing with a deep hierarchical memory (as in OMPN) incurs significant computational costs (Table 2).

In this paper we propose a novel architecture to learning sub-routines that addresses these shortcomings, which we call *Slot-based Transformer for Temporal Abstraction (SloTTAr)*. Central to our approach is the similarity between the *spatial grouping* of pixels into visual objects and the *temporal grouping* of state-action pairs into self-contained sub-routines. This motivates us to combine a parallel Transformer encoder with a Slot Attention module [26] (developed for learning object representations [16]), to group learned features at different temporal positions and recover a modular factorization into 'slots' (sub-routines) of the inputs. A parallel Transformer decoder reconstructs the action sequence from each slot and outputs an unnormalized distribution over endpoints. From these, the segment belonging to each sub-routine and a standard reconstruction objective for unsupervised training can be derived. We demonstrate the efficacy of our approach on Craft [27] and MiniGrid [6] where we typically observe significant improvements over CompILE and OMPN in terms of recovering 'ground-truth' sub-routines. In this way, we show how general principles of similarity-based grouping used to segment visual inputs into objects [40, 25, 24, 16] are also relevant for grouping other input modalities [33].

2 Method

Given an input sequence of actions $\boldsymbol{a} = [a_1, a_2, ..., a_L] : a_l \in \{1, .., A\}$ where A is the size of the action space and observations $\boldsymbol{o} = [\boldsymbol{o}_1, \boldsymbol{o}_2, ..., \boldsymbol{o}_L] : \boldsymbol{o}_l \in \mathbb{R}^{D_{\text{obs}}}$ of length L. Our goal is to infer the unique constituent latent *sub-routine* to which a pair of $(a_l, \boldsymbol{o}_l)$ belongs and learn its associated representation ($\texttt{slot_k} \in \mathbb{R}^{D_{\text{slots}}}$). Similar to CompILE [23], we consider an unsupervised reconstruction objective for training, yet here we strive for a fully parallel approach that employs more general-purpose modules for grouping inputs based on their internal predictive structure.

Our model, SloTTAr, consists of three modules, as shown in Figure 1. First, a Transformer encoder [44] learns suitable spatio-temporal features from the input sequences $\boldsymbol{a}$, $\boldsymbol{o}$. This encoder benefits from *global* access to the whole input sequence to learn suitable context features at each location, unlike prior purely sequential RNN-based approaches [23, 27]. Subsequently, we use a Slot Attention module [26] to iteratively group the features at each temporal location in parallel based on their internal predictive structure and obtain K slot representations `slot_k`. The slot representations are decoded in parallel by a decoder that outputs both a predicted actions $\hat{\boldsymbol{a}}^{(k)}$ and an unnormalized distribution over the end position of the sub-routine modelled by the respective slot (`end_logits_k`).

Algorithm 1 Our implementation of the Slot Attention update [26].

Inputs: `inputs` $\in \mathbb{R}^{L \times D_{\text{in}}}$, `slots where slot_k` $\sim \mathcal{N}(\boldsymbol{\mu}_k, diag(\sigma_k)) \in \mathbb{R}^{D_{\text{slots}}}$ $\forall k \in K$
Params: Linear projections for attention: $key, query, value$; `GRU; MLP; LayerNorm x2`

for $t = 0, 1, ..., T$ **do**
  `slots_prev = slots`
  `slots = LayerNorm(slots)`
  `attn = Softmax(`$\frac{1}{\sqrt{D_{\text{slots}}}}$ key`(inputs)`$\cdot$ $query$`(slots)`^{T}`, axis='slots')` ▷ norm.
  `updates := WeightedSum(weights=attn, values=`$value$`(inputs))` ▷ aggregate
  `slots = GRU(state=slots_prev, inputs=updates)` ▷ GRU update (per-slot)
  `slots += MLP(LayerNorm(slots))` ▷ residual update (per-slot)
end for
return `slots`

Sub-routine segment masks are generated from the unnormalized distributions to compute the aggregated action sequence logits $\hat{\boldsymbol{a}}$. Finally, a reconstruction objective between $\hat{\boldsymbol{a}}$ and $\boldsymbol{a}$ can be defined.

Encoder. We process action tokens $\boldsymbol{a}$ and observations $\boldsymbol{o}$ by `Embedding` and `Linear` layers respectively to acquire separately learned distributed representations for each. Next, we learn a joint representation of the action and observation representations at each position using an `MLP`, after which a sinusoidal positional encoding $p_l^{\text{sin}} \in \mathbb{R}^{D_{\text{enc}}}$ is added [44]. Finally, we apply a number of standard Transformer encoder layers to learn suitable spatio-temporal features based on the content of the *entire* sequence and add a learned positional encoding $p^{\text{learn}} \in \mathbb{R}^{D_{\text{enc}}}$ for follow-up processing:

$$\boldsymbol{z}_l^a,\ \boldsymbol{z}_l^o = \texttt{Embedding}(a_l),\ \texttt{Linear}(\boldsymbol{o}_l) \qquad \boldsymbol{z}_l^{ao} = \texttt{MLP}(\texttt{concat}(\boldsymbol{z}_l^a, \boldsymbol{z}_l^o)) \qquad \forall\ l \in L$$
$$\boldsymbol{h} = \texttt{TransformerEnc}(\boldsymbol{z}^{ao} + \boldsymbol{p}^{\text{sin}}) + \boldsymbol{p}^{\text{learn}}$$

Slot Attention. We use Slot Attention [26] to group these spatio-temporal features according to their constituent sub-routines and learn associated representations given by the slots. Slot Attention was previously only applied to the visual setting where pixels are grouped according to constituent visual objects to model images. Here we hypothesize that sub-routines take on a similar role as modular and reusable primitives when modeling action sequences, suggesting that they similarly can be inferred using a grouping mechanism that focuses on their internal predictive structure (modularity) [16].

The iterative grouping mechanism in Slot Attention (reproduced in Algorithm 1) is implemented via key-value attention and a stateful slot update rule using a recurrent neural network (`GRU` [7], see also earlier work [12]). The input features are projected to keys and values using linear layers $key(\cdot)$, $value(\cdot)$, queries are computed from slots using a linear layer $query(\cdot)$, and dot-product attention between keys and queries is used to distribute value vectors among the slots. Initial representations for each slot (`slot_k`) are sampled from a Gaussian $\mathcal{N}(\boldsymbol{\mu}_k, diag(\sigma_k))$. Importantly, slots *compete* to represent parts of the input sequence via a `Softmax` over slots, thereby encouraging a decomposition of the input into modular parts that can be processed separately and represented efficiently. Unlike the original Slot Attention formulation, we use separate shift and scaling parameters ($\boldsymbol{\mu}_k$ and σ_k) for each slot, which allows for ordering them as later needed by the decoder. Finally, we replaced the `WeightedMean` (line 9 in Algorithm 3 in Appendix A.3), which we found to lead to more stable training and better performance.

Decoder. To decode the slot representations and obtain a correspondence of actions in $\mathbf{a}$ to their constituent sub-routine segments, we adapt the spatial broadcast decoder architecture [45] to the sequential setting using Transformers. Each slot representation is decoded in parallel to obtain the predicted action logits $\hat{\mathbf{a}}^{(k)}$ as well as an unnormalized distribution over the end position of the sub-routine in the sequence (`end_logits_k`). The latter is used to generate the segment masks corresponding to each sub-routine as described in Algorithm 2. We enforce temporal contiguity of the segment modeled by each slot and use a fixed ordering to compose them, which is different from the standard mixture formulation adopted by prior work on discovering visual objects [15, 14, 26]. This has a similar effect to the sequential decoding in CompILE [23] and provides a useful inductive bias for treating sub-tasks as contiguous chunks in time that can be ordered along the temporal axis. The entire system is trained end-to-end to minimize the cross-entropy loss between the aggregated

Table 1: F1 scores and alignment accuracies on Craft [27] (fully & partially observable) and on the partially observable *Doorkey-8x8* environment from MiniGrid [6]. F1 scores are computed with tolerance=1 consistent with [27]. * We observed that 4 of the 5 seeds for the OMPN-2 failed to train stably and loss resulted in NaNs despite only making changes the memory hierarchy depth.

	Craft (fully)		Craft (partial)		DoorKey-8x8 (partial)	
	F1 score	Align. acc.	F1 score	Align. acc.	F1 score	Align. acc.
CompILE	83.03 (3.06)	86.60 (0.46)	71.99 (13.38)	69.27 (15.48)	50.58 (4.01)	72.88 (2.58)
OMPN	96.80 (2.10)	95.98 (2.37)	**92.10 (2.81)**	**89.10 (3.05)**	41.10 (12.13)	56.61 (5.15)
OMPN-2	96.34 (1.12)	95.26 (2.85)	90.41 (3.23)	84.18 (3.37)	30.24*	52.81*
SloTTAr	**99.58 (0.55)**	**99.47 (0.49)**	80.41 (3.17)	82.11 (3.01)	**94.77 (3.88)**	**94.51 (2.84)**

predicted action logits $\sum_k \texttt{mask_k} \cdot \hat{\mathbf{a}}^{(k)}$ and the ground-truth $\mathbf{a}$. For additional details on the model architecture please refer to Appendix A.3.

Algorithm 2 Mask Generation

Inputs: `end_logits` $\in \mathbb{R}^{K \times L}$, `masks = []`, `end_cdf_k = 1`, `mask_upto_km1 = 0`

for $k = 1, ..., K$ **do**
  `end_dist_k = Softmax(end_logits_k, axis='L')` ▷ norm. over sequence length
  `end_cdf_k = CumSum(end_dist_k, axis='L')` ▷ end position CDF
  `mask_k = mask_upto_k * (1 - mask_upto_km1)` ▷ compute kth mask.
  `masks = Append(masks, mask_k)` ▷ append kth mask
end for
return `masks`

3 Additional Related Work

The Craft environment was originally introduced to evaluate the method of Andreas et al. [2], however they rely on annotations of sub-routine sequences ('policy sketches') as additional supervision to the model. TACO [37] was also proposed to address this setting, but views it as a sequence alignment and classification problem using an LSTM [18] trained with a CTC loss [13]. In other recent work [1] the primitives are learned directly from offline data for continuous control. However, these methods [1] learn a continuous (low-dimensional) space of primitives whereas our method represents primitives as a discrete set.

In the context of the options framework, several methods have been proposed for option and/or sub-goal discovery [3, 28, 29, 41, 8, 39, 38]. However, these methods are not easily extendable to the case with highly nonlinear function approximation. Alternatively, other methods using nonlinear function approximation seek to maximize coverage (diversity) of learned skills by maximizing the mutual information between options and terminal states achieved by their execution [17, 11].

Many recent works have applied Transformers to modalities beyond natural language, such as images [10, 46], and other high-dimensional input domains [21, 20]. The application of Transformers to reinforcement learning has also recently received a lot of interest [31, 34, 22, 5, 19].

4 Experiments

We compare SLoTTAr to CompILE [23] and OMPN [27] on the partially and fully observable versions of Craft environment [27] and on a partially observable environment in MiniGrid [6]. Here our focus is on the fully unsupervised setting without the use of task sketches as an auxiliary supervision [2, 37]. We use 3 tasks in the Craft environment namely, MakeAxe, MakeBed, MakeShears consistent with prior work [27]. In MiniGrid we use the DoorKey-8x8 environment. Episodes in this environment are procedurally generated leading to far greater variability for actions that constitute a sub-routine compared to Craft. Further in the DoorKey-8x8 environment, two actions (*'PICKUP'* or *'TOGGLE'*) are used to determine sub-routine boundaries as opposed to in Craft where this is marked by only a single *'USE'* action.

Table 2: Number of tokens (in thousand) processed per-second during training and testing on a single Nvidia GTX 1080Ti GPU card. These numbers are computed on the fully observable Craft task using a batch size of 128 and a sequence length of 65 tokens. The number of segments is 4. OMPN and OMPN-2 use 3 and 2 levels of memory hierarchy respectively.

	Train	Test
CompILE	14	36
OMPN	3	7
OMPN-2	4	10
SloTTAr	**92**	**226**

Evaluation. To quantitatively measure the quality of the action sequence decomposition, we use the F1 score (with a tolerance=1) based on the accuracy of the boundary index predictions with respect to ground-truth. Further, we also report the alignment accuracy between the predicted and ground-truth sub-routines consistent with prior work [27]. Please refer to Appendix A.1 for further details on how ground truth sub-routine boundaries are computed.

Hyperparameter Search. Below we report results after conducting an extensive hyperparameter search (up to 200 configurations) for each method. We include training parameters like batch size and learning rate, capacity parameters like layer sizes, and model-specific parameters such as the number of Slot Attention iterations in case of SloTTAr. Results are reported for the best configurations with mean and std. dev. computed over 5 seeds. Details about the search parameters for each method, including details about the best configurations can be found in Appendix A.3.

Results. Table 1 shows the results of comparing SloTTAr to both baselines on all three environments. On the fully observable Craft tasks, it can be seen how SLoTTAr outperforms both CompILE and OMPN. Similarly, on the partially observable settings in Craft, it can be seen how SloTTAr outperforms CompILE, although this time it performs worse compared to OMPN. We speculate how the strong hierarchical inductive bias in OMPN gives it an edge in this setting, which closely reflects the hierarchical sub-task structure in Craft. Indeed, when we only consider OMPN configurations having two levels of memory (OMPN-2) it can be observed how the difference in alignment accuracy to the best performing configuration reduces[1]. Finally, on the harder DoorKey-8x8 partially observable environment, it can observed how SloTTAr significantly outperforms both CompILE and OMPN in terms of both metrics. These results demonstrate the strengths of our approach and suggest that a fully parallel approach to computing sub-routine boundary points is highly beneficial.

Another added advantage of using a parallel architecture is the potential gain in computational complexity. Table 2 reports the number of tokens processed per-second during training (forward and backward passes) and at testing time (forward pass) for each model. It can be seen how SloTTAr is about 7x faster compared to CompILE and about 30x faster compared to the OMPN model, suggesting that it also may scale better to more sophisticated environments. Reducing the memory depth in case of OMPN only yields a marginal improvement.

5 Conclusion

We have proposed a novel approach to learning action segments belonging to modular sub-routines (suitable as 'options' [42]) in a purely unsupervised fashion. Our approach draws insight from prior literature on learning about visual objects [26] and Transformers [44] to improve over existing purely sequential approaches. It was shown how SloTTAr outperforms CompILE and OMPN in terms of recovering ground-truth sub-routine segments and provides a massive speed-up by leveraging parallel computation over the time axis. To that extent we demonstrated how general principles of similarity-based grouping used to segment visual inputs are also relevant for grouping other input modalities, suggesting many additional avenues for future research.

[1]The best configuration for OMPN in Table 1 was obtained when using 3 levels of hierarchy depth (Table 9).

References

[1] Anurag Ajay, Aviral Kumar, Pulkit Agrawal, Sergey Levine, and Ofir Nachum. OPAL: Offline primitive discovery for accelerating offline reinforcement learning. In *Int. Conf. on Learning Representations (ICLR)*, May 2021.

[2] Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 166–175, Sydney, Australia, August 2017.

[3] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Proc. AAAI Conf. on Artificial Intelligence*, pages 1726–1734, San Francisco, CA, USA, February 2017.

[4] Bram Bakker and Jürgen Schmidhuber. Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. In *Proc. Conf. on Intelligent Autonomous Systems*, pages 438–445, Amsterdam, Netherlands, March 2004.

[5] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *Preprint arXiv:2106.01345*, 2021.

[6] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. Minimalistic gridworld environment for openai gym. `https://github.com/maximecb/gym-minigrid`, 2018.

[7] Kyunghyun Cho, Çağlar Gülçehre, Bart van Merriënboer, Dzmitry Bahdanau, Fethi Bougares Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014.

[8] Özgür Şimşek and Andrew G. Barto. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 95, New York, NY, USA, 2004. Association for Computing Machinery.

[9] Peter Dayan and Geoffrey E. Hinton. Feudal reinforcement learning. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pages 271–278, Denver, CO, USA, December 1992.

[10] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

[11] Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is all you need: Learning skills without a reward function. In *Int. Conf. on Learning Representations (ICLR)*, New Orleans, LA, USA, May 2019.

[12] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[13] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 369–376, Pittsburgh, PA, USA, June 2006.

[14] Klaus Greff, Raphaël Lopez Kaufman, Rishabh Kabra, Nick Watters, Christopher Burgess, Daniel Zoran, Loic Matthey, Matthew Botvinick, and Alexander Lerchner. Multi-object representation learning with iterative variational inference. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 2424–2433, Long Beach, CA, USA, June 2019.

[15] Klaus Greff, Sjoerd van Steenkiste, and Jürgen Schmidhuber. Neural expectation maximization. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pages 6691–6701, Long Beach, CA, USA, December 2017.

[16] Klaus Greff, Sjoerd van Steenkiste, and Jürgen Schmidhuber. On the binding problem in artificial neural networks. *arXiv preprint arXiv:2012.05208*, 2020.

[17] Karol Gregor, Danilo Jimenez Rezende, and Daan Wierstra. Variational intrinsic control. In *ICLR Workshop track*, Toulon, France, April 2017.

[18] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, November 1997.

[19] Kazuki Irie, Imanol Schlag, Róbert Csordás, and Jürgen Schmidhuber. Going beyond linear transformers with recurrent fast weight programmers. *Preprint arXiv:2106.06295*, 2021.

[20] Andrew Jaegle, Sebastian Borgeaud, Jean-Baptiste Alayrac, Carl Doersch, Catalin Ionescu, David Ding, Skanda Koppula, Daniel Zoran, Andrew Brock, Evan Shelhamer, et al. Perceiver IO: A general architecture for structured inputs & outputs. *Preprint arXiv:2107.14795*, 2021.

[21] Andrew Jaegle, Felix Gimeno, Andy Brock, Oriol Vinyals, Andrew Zisserman, and João Carreira. Perceiver: General perception with iterative attention. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 4651–4664, Virtual only, July 2021.

[22] Michael Janner, Qiyang Li, and Sergey Levine. Reinforcement learning as one big sequence modeling problem. *Preprint arXiv:2106.02039*, 2021.

[23] Thomas Kipf, Yujia Li, Hanjun Dai, Vinícius Flores Zambaldi, Alvaro Sanchez-Gonzalez, Edward Grefenstette, Pushmeet Kohli, and Peter W. Battaglia. CompILE: Compositional imitation learning and execution. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 3418–3428, Long Beach, California, USA, June 2019.

[24] K Koffka. Principles of gestalt psychology. 1935.

[25] W Köhler. Gestalt psychology. 1929.

[26] Francesco Locatello, Dirk Weissenborn, Thomas Unterthiner, Aravindh Mahendran, Georg Heigold, Jakob Uszkoreit, Alexey Dosovitskiy, and Thomas Kipf. Object-centric learning with slot attention. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Virtual only, December 2020.

[27] Yuchen Lu, Yikang Shen, Siyuan Zhou, Aaron Courville, Joshua B. Tenenbaum, and Chuang Gan. Learning task decomposition with ordered memory policy network. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

[28] Marlos C. Machado, Marc G. Bellemare, and Michael Bowling. A Laplacian framework for option discovery in reinforcement learning. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 2295–2304, Sydney, Australia, August 2017.

[29] Amy McGovern and Andrew G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, page 361–368, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[30] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 1928–1937, New York City, NY, USA, June 2016.

[31] Emilio Parisotto, Francis Song, Jack Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant Jayakumar, Max Jaderberg, Raphael Lopez Kaufman, Aidan Clark, Seb Noury, Matthew M. Botvinick, Nicolas Heess, and Raia Hadsell. Stabilizing Transformers for reinforcement learning. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 7487–7498, Virtual only, July 2020.

[32] Xue Bin Peng, Michael Chang, Grace Zhang, Pieter Abbeel, and Sergey Levine. MCP: Learning composable hierarchical control with multiplicative compositional policies. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 3681–3692, Vancouver, Canada, December 2019.

[33] Gabriel A Radvansky and Jeffrey M Zacks. *Event cognition*. Oxford University Press, 2014.

[34] Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, Chloe Hillier, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, April 2020.

[35] Jürgen Schmidhuber. Towards compositional learning in dynamic networks. *Technical Report FKI-129-90*, 1990.

[36] Jürgen Schmidhuber and Reiner Wahnsiedler. Planning simple trajectories using neural subgoal generators. In *Proc. Int. Conf. on From Animals to Animats 2: Simulation of Adaptive Behavior*, page 196–202, August 1993.

[37] Kyriacos Shiarlis, Markus Wulfmeier, Sasha Salter, Shimon Whiteson, and Ingmar Posner. TACO: Learning task decomposition via temporal alignment for control. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 4654–4663, Stockholm, Sweden, July 2018.

[38] Ozgur Simsek and Andre S Barreto. Skill characterization based on betweenness. In *Advances in neural information processing systems*, pages 1497–1504, 2008.

[39] Özgür Şimşek, Alicia P Wolfe, and Andrew G Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd international conference on Machine learning*, pages 816–823, 2005.

[40] Elizabeth S. Spelke. Principles of object perception. *Cognitive Science*, 14(1):29–56, 1990.

[41] Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pages 212–223. Springer, 2002.

[42] Richard S Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[43] Prasad Tadepalli and Thomas G Dietterich. Hierarchical explanation-based reinforcement learning. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 358–366, Nashville, TN, USA, July 1997.

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pages 5998–6008, Long Beach, CA, USA, December 2017.

[45] Nicholas Watters, Loic Matthey, Christopher P Burgess, and Alexander Lerchner. Spatial broadcast decoder: A simple architecture for learning disentangled representations in VAEs. In *Learning from Limited Labeled Data (LLD) Workshop, ICLR*, New Orleans, LA, USA, May 2019.

[46] Xizhou Zhu, Weijie Su, Lewei Lu, Bin Li, Xiaogang Wang, and Jifeng Dai. Deformable DETR: Deformable transformers for end-to-end object detection. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

A Experimental Details

A.1 Datasets

Craft The craft environment was initially introduced and re-implemented as a gym environment [2, 27]. Demonstration data is collected from the rollouts of the heuristic bots [27]. We use rollouts from each of the 3 tasks MakeAxe, MakeBed and MakeShears for both the fully observable and partial observable versions of these environments [27] to report the results in Table 1. We use $10\,000$ trajectories for training and another separate 1000 trajectories each for creating validation and test splits. These are used for hyperparameter tuning and reporting evaluation metrics respectively.

MiniGrid We use the DoorKey-8x8 environment from MiniGrid [6] as an additional dataset. This environment uses procedural generation for each episode ensuring more variability in the action sequences that make up a sub-routine in comparison to Craft. We collect demonstration data by training an A2C agent [30] on this environment until the policy receives an episodic return of ≥ 0.9, which is close to optimal. We use $10\,000$ rollouts collected from this "expert" A2C agent [30] as the training set and 1000 rollouts each for the validation and test splits used to tune hyperparameters and report evaluation metrics respectively.

Ground-truth segment generation For the Craft environment, the *'USE'* action serves as a delimiter that marks the end of sub-routines. In DoorKey-8x8 environment, the *'PICKUP'* and *'TOGGLE'* actions serve as delimiters. We extract the ground-truth boundaries of sub-routines using the indices of these action tokens in the sequences as markers. Ground-truth sub-routine indices for every sequence are in ascending order from left-to-right where delimiting points are specified using the heuristics described above. Table 3 shows the ground-truth sub-routines for each of the environments.

Table 3: Ground-truth sub-routines for Craft [27] (fully & partially observable) and Doorkey-8x8 environment from MiniGrid [6].

MakeAxe	get wood, make at workbench, get iron, make at toolshed
MakeBed	get wood, make at toolshed, get grass, make at workbench
MakeShears	get wood, make at workbench, get iron, make at workbench
DoorKey-8x8	pickup key, open door, go to green goal

A.2 Evaluation Metrics

Evaluation metrics used to quantitatively measure the segmentation performance of models are the alignment accuracy of sub-routine prediction and F1 score of boundary index prediction and is consistent with prior work [27, 23].

Alignment Accuracy The expression to compute alignment accuracy is shown below:

$$\text{Align Acc.} = \frac{1}{L * N} \sum_{l}^{L} \sum_{n}^{N} \mathbb{1}(\hat{s}_l^n = s_l^n)$$

where L is the sequence length (possibly different for each sequence), N is the number of sequences, $\hat{s}_l$ is the predicted sub-routine and s_l is the ground-truth sub-routine of action a_l in the nth sequence.

F1 Score The F1 score on predicted boundary indices is computed as shown below:

$$\text{F1 score} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

where precision is computed as,

$$\text{precision} = \frac{\text{\# matches of boundary predictions with ground-truth}}{\text{total \# boundary predictions}}$$

and recall is computed as,

$$\text{recall} = \frac{\text{\# ground-truth matches with predictions}}{\text{total \# ground-truth boundaries}}$$

Please refer to the Appendix D.1. in prior work [23] for further details.

Algorithm 3 Original Slot Attention update [26].

Inputs: `inputs` $\in \mathbb{R}^{L \times D_{\text{in}}}$, `slots where slot_k` $\sim \mathcal{N}(\boldsymbol{\mu}, diag(\sigma)) \in \mathbb{R}^{D_{\text{slots}}}\ \forall k \in K$
Params: Linear projections for attention: $key, query, value$; `GRU; MLP; LayerNorm x2`

for $t = 0, 1, ..., T$ **do**
  `slots_prev = slots`
  `slots = LayerNorm(slots)`
  `attn = Softmax(`$\frac{1}{\sqrt{D_{\text{slots}}}}$ key`(inputs)`$\cdot$ $query$`(slots)`^{T}`, axis='slots')` ▷ norm.
  `updates := WeightedMean(weights=attn, values=`$value$`(inputs))` ▷ re-normalizing `attn` values over L
  `slots = GRU(state=slots_prev, inputs=updates)` ▷ GRU update (per-slot)
  `slots += MLP(LayerNorm(slots))` ▷ residual update (per-slot)
end for
return `slots`

A.3 Architecture and Training Details

Input Processing The `Embedding` layer used (Section 2) maps action tokens a_l to D_{enc} dimensions (denoted by hidden size in Table 4 and Table 5). The `Linear` layer used (Section 2) maps observations $\boldsymbol{o}_l$ to D_{enc} dimensions. These D_{enc} dimensional action and observation features are then concatenated and processed by a 1-layer MLP with D_{enc} units and *ReLU* activation to learn a joint action-observation embedding space.

Encoder The `Linear` layers used in self-attention have D_{enc} units in the `TransformerEnc` block. The pointwise feedforward network used in the `TransformerEnc` uses 2x the number of units as hidden size. We add the standard sinusoidal positional encoding $\boldsymbol{p}_{\text{sin}}$ [44] to the joint action-observation features $\boldsymbol{z_{ao}}$. Further, we use a `Linear` layer to generate the learned positional encoding $\boldsymbol{p}_{\text{learn}}$ that is added to the outputs $\boldsymbol{h}$ from the `TransformerEnc` module. The final and sweep configuration(s) for each of these hyperparameters are shown in Table 4 and Table 5 respectively. The overall `TransformerEnc` block uses the same architectural template as the Transformer Encoder [44].

Slot Attention The slot attention module uses `Linear` layers with D_{slots} units to generate keys and values. The recurrent update function is implemented using a GRU [7] (see also earlier work [12]) with D_{slots} units (denoted by slot size in Table 5 and Table 4). The MLP network used for the residual update is implemented using a 2-layer network with D_{slots} units and *ReLU* and *Linear* activation for the hidden layer and output layers respectively. Futher, the initial query vector for `slot_k` is sampled from a Gaussian distribution with learnable mean μ_k and standard-deviation σ_k.

For reference Algorithm 3 describes the original Slot Attention update [26].

Decoder The decoder architecture adapts the spatial broadcast decoder [45] in a suitable manner to decode all slots into their corresponding action segments. We broadcast the slot representations along the sequence length L and pass the observations $\boldsymbol{o_l}$ as inputs at each timestep to the Decoder. The Decoder module has the exact same architecture as the Transformer Encoder with the exception that an additional output `Linear` layer of $A + 1$ (where A is the size of action space) dimensions is used to decode the slot representations into predicted action logits and the end position logits. Segment masks are computed given end position logits as shown in Algorithm 2. Then we composite segment masks and predicted action logits to generate the full predicted action sequence.

We perform a random search of 200 hyperparameter configurations from all possible configurations in the hyperparameter sweeps shown in Table 4 on 3 seeds. We train our model for 50 epochs on Craft (fully-observable) and 100 epochs for both the Craft (partially-observable) and DoorKey-8x8 environments. We observed that our models start overfitting on the training set beyond this point. Then, we picked the best performing configuration(s) for each of the 3 environments for our model based on the evaluation metrics on the validation set and run them again on 5 seeds to report final scores (shown in Table 1).

The final configurations used for our model are shown in Table 5.

Table 4: Hyperparameter sweep configurations for SloTTAr for Craft [27] (fully & partially observable) and Doorkey-8x8 environment from MiniGrid [6].

SloTTAr	Craft (fully)	Craft (partial)	MiniGrid (DoorKey-8x8)
batch size	[64, 128, 256]	[64, 128, 256]	[64, 128, 256]
hidden size	[32, 64, 128]	[32, 64, 128]	[32, 64, 128]
number of heads	[4, 8, 16]	[4, 8, 16]	[4, 8, 16]
slot size	[32, 64, 128]	[32, 64, 128]	[32, 64, 128]
slot std_dev	[1.0, 2.0]	[1.0, 2.0]	[1.0, 2.0]
number of iterations	[1, 2]	[1, 2]	[1, 2]
learning rate	[2.5e-4, 5e-4 1e-3]	[2.5e-4, 5e-4 1e-3]	[2.5e-4, 5e-4 1e-3]

Table 5: Final hyperparameter configurations for SloTTAr for Craft [27] (fully & partially observable) and Doorkey-8x8 environment from MiniGrid [6].

SloTTAr	Craft (fully)	Craft (partial)	MiniGrid (DoorKey-8x8)
batch size	128	128	128
hidden size	64	128	64
number of heads	8	16	16
number of layers	1	1	1
slot size	64	64	64
slot std_dev	1.0	2.0	1.0
number of iterations	1	1	1
number of slots	4	4	3
optimizer	Adam	Adam	Adam
learning rate	0.0005	0.00025	0.00025

A.4 Baseline Models and Training Details

We developed our re-implementations of CompILE [2] and OMPN [3] by adapting the authors' implementations available on GitHub. We perform a random search of 200 hyperparameter configurations from all possible configurations in the hyperparameter sweeps shown in Table 7 and Table 8 for each of the 3 environments and both baseline models (CompILE and OMPN) with 3 seeds. Then, we picked the best performing configuration(s) based on their evaluation metrics on the validation set from this sweep. We run these best performing configuration(s) on 5 seeds to obtain all results shown in Table 1. We train CompILE and OMPN models for 30 and 20 epochs respectively as we observed that these models start overfitting on the training set beyond this point. We checkpoint models during the course of training based on their evaluation metric scores on the validation set. We use the best checkpoints to report the final scores shown in Table 1 for all the baseline models.

The final configurations used for the baseline models on the 3 environments are shown in Table 6 and Table 9.

[2] https://github.com/tkipf/compile

[3] https://github.com/Ordered-Memory-RL/ompn_craft

Table 6: Final hyperparameter configurations for CompILE for Craft [27] (fully & partially observable) and DoorKey-8x8 environment from MiniGrid [6].

CompILE	Craft (fully)	Craft (partial)	MiniGrid (DoorKey-8x8)
batch size	128	128	64
beta_b	0.1	0.1	0.001
beta_z	0.1	0.1	0.001
prior rate	3	3	2
hidden size	128	128	128
latent dist.	"gaussian"	"gaussian"	"gaussian"
latent size	128	128	128
number of segments	4	4	3
optimizer	Adam	Adam	Adam
learning rate	0.00025	0.00025	0.00025

Table 7: Hyperparameter sweep configurations for CompILE for Craft [27] (fully & partially observable) and Doorkey-8x8 environment from MiniGrid [6].

CompILE	Craft (fully)	Craft (partial)	MiniGrid (DoorKey-8x8)
batch size	[64, 128, 256]	[64, 128, 256]	[64, 128, 256]
beta_b = beta_z	[0.001, 0.01, 0.1]	[0.001, 0.01, 0.1]	[0.001, 0.01, 0.1]
hidden size	[64, 128]	[64, 128]	[64, 128]
latent dist.	["gaussian", "concrete"]	["gaussian", "concrete"]	["gaussian", "concrete"]
latent size	[64, 128]	[64, 128]	[64, 128]
learning rate	[5e-5, 2.5e-4 1e-3]	[5e-5, 2.5e-4 1e-3]	[5e-5, 2.5e-4 1e-3]
prior rate	[2, 3, 4]	[2, 3, 4]	[2, 3, 4]

Table 8: Hyperparameter sweep configurations for OMPN for Craft [27] (fully & partially observable) and Doorkey-8x8 environment from MiniGrid [6].

OMPN	Craft (fully)	Craft (partial)	MiniGrid (DoorKey-8x8)
batch size	[64, 128, 256]	[64, 128, 256]	[64, 128, 256]
hidden size	[64, 128]	[64, 128]	[64, 128]
learning rate	[2e-4, 5e-4 1e-3]	[2e-4, 5e-4 1e-3]	[2e-4, 5e-4 1e-3]
number of slots	[2, 3]	[2, 3]	[2, 3]
max. gradient norm	[0.5, 1.0, 2.0]	[0.5, 1.0, 2.0]	[0.5, 1.0, 2.0]

Table 9: Final hyperparameter configurations for OMPN for Craft [27] (fully & partially observable) and Doorkey-8x8 environment from MiniGrid [6].

OMPN	Craft (fully)	Craft (partial)	MiniGrid (DoorKey-8x8)
batch size	128	128	128
hidden size	128	128	128
number of slots	3	3	3
number of segments	4	4	3
max. gradient norm	0.5	0.5	1.0
optimizer	Adam	Adam	Adam
learning rate	0.0002	0.0002	0.0002