LEARNING TASK DECOMPOSITION WITH ORDER-MEMORY POLICY NETWORK

Anonymous authors
Paper under double-blind review

ABSTRACT

Many complex real-world tasks are composed of several levels of sub-tasks. Humans leverage these hierarchical structures to accelerate the learning process and achieve better generalization. To simulate this process, we introduce Ordered Memory Policy Network (OMPN) to discover task decomposition by imitation learning from demonstration. OMPN has an explicit inductive bias to model a hierarchy of sub-tasks. Experiments on Craft world and Dial demonstrate that our model can more accurately recover the task boundaries with behavior cloning under both unsupervised and weakly supervised setting than previous methods. OMPN can also be directly applied to partially observable environments and still achieve high performance. Our visualization further confirms the intuition that OMPN can learn to expand the memory at higher levels when one subtask is close to completion.

1 INTRODUCTION

Learning from Demonstration (LfD) is a popular paradigm for policy learning and has served as a warm-up stage in many successful reinforcement learning applications (Vinyals et al., 2019; Silver et al., 2016). However, beyond simply imitating the experts' behaviors, an intelligent agent's crucial capability is to decompose an expert's behavior into a set of useful skills and discover sub-tasks. The discovered structure from expert demonstrations could be leveraged to re-use previously learned skills in the face of new environments (Sutton et al., 1999; Gupta et al., 2019; Andreas et al., 2017). Since manually labeling sub-task boundaries for each demonstration video is extremely expensive and difficult to scale up, it is essential to learn task decomposition *unsupervisedly*, where the only supervision signal comes from the demonstration itself.

This question of discovering a meaningful segmentation of the demonstration trajectory is the key focus of Hierarchical Imitation Learning (Kipf et al., 2019; Shiarlis et al., 2018; Fox et al., 2017; Achiam et al., 2018) These works can be summarized as finding the optimal behavior hierarchy so that the behavior can be better predicted (Solway et al., 2014). They usually model the sub-task structure as latent variables, and the subtask identifications are extracted from a learnt posterior. In this paper, we propose a novel perspective to solve this challenge: could we design a *smarter* neural network architecture, so that the sub-task structure can emerge during imitation learning? To be specific, we want to design a recurrent policy network such that examining the memory trace at each time step could reveal the underlying subtask structure.

Drawing inspiration from the Hierarchical Abstract Machine (Parr & Russell, 1998), we propose that each subtask can be considered as a finite state machine. A hierarchy of sub-tasks can be represented as different slots inside the memory bank. At each time step, a subtask can be internally updated with the new information, call the next-level subtask, or return the control to the previous level subtask. If our designed architecture maintains a hierarchy of sub-tasks operating in the described manner, then subtask identification can be as easy as monitoring when the low-level subtask returns control to the higher-level subtask, or when the high-level subtask expands to the new lower-level subtask.

We give an illustrative grid-world example in Figure 1. In this example, there are different ingredients like grass for the agent to pickup. There is also a factory where the agent can use the ingredients. Suppose the agent wants to complete the task of building a bridge. This task can be decomposed into a tree-like, multi-level structure, where the root task is divided into $GetMaterial$

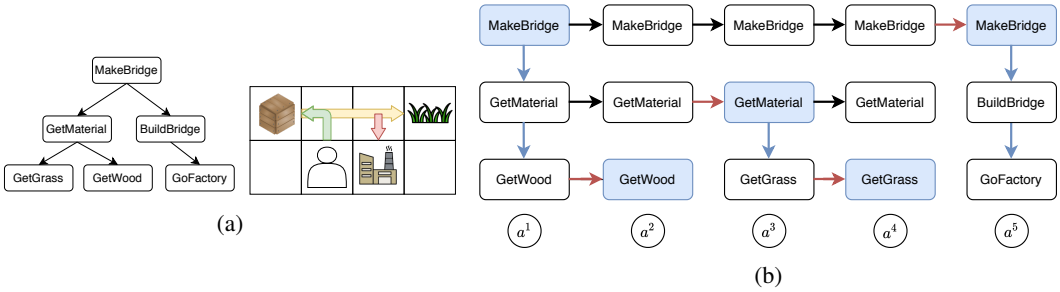


Figure 1: (a) A simple grid world with the task "make bridge", which can be decomposed into multi-level subtask structure. (b) The representation of subtask structure within the agent memory with *horizontal update* and *vertical expansion* at each time step. The black arrow indicates a copy operation. The *expansion position* is the memory slot where the vertical expansion starts and is marked blue.

and $BuildBridge$. $GetMaterial$ can be further divided into $GetGrass$ and $GetWood$. We provide a sketch on how this subtask structure should be represented inside the agent's memory during each time step. The memory would be divided into different levels, corresponding to the subtask structure. When $t = 1$, the model just starts with the root task, $MakeBridge$, and vertically expands into $GetMaterial$, which further vertically expands into $GetWood$. The *vertical expansion* corresponds to planning or calling the next level subtasks. The action is produced from the lowest-level memory. The intermediate $GetMaterial$ is copied for $t < 3$, but horizontally updated at $t = 3$, when $GetWood$ is finished. The *horizontal update* can be thought of as an internal update for each subtask, and the updated $GetMaterial$ vertically expands into a different child $GetGrass$. The root task is always copied until $GetMaterial$ is finished at $t = 4$. As a result, $MakeBridge$ goes through one horizontal update at $t = 5$ and then expands into $BuildBridge$ and $GoFactory$. We can identify the subtask boundaries from this representation by looking at the change of *expansion position*, which is defined to be the memory slot where vertical expansion happens. E.g., from $t = 2$ to $t = 3$, the expansion position goes from the lowest level to the middle level, suggesting the completion of the low-level subtask. From $t = 4$ to $t = 5$, the expansion position goes from the lowest level to the highest level, suggesting the completion of both low-level and mid-level subtasks.

Driven by this intuition, we propose the *Ordered Memory Policy Network* (OMPN) as generic network architecture. We propose to use a bottom-up recurrence and a top-down recurrence to implement *horizontal update* and *vertical expansion* respectively. Our proposed memory-update rule further maintains a hierarchy among memories such that the higher-level memory can store longer-term information like root task information, while the lower-level memory can store shorter-term information like leaf subtask information. At each time step, the model would softly decide the expansion position from which to perform vertical expansion based on a differentiable stick-breaking process, so that our model can be trained end-to-end.

We demonstrate the effectiveness of our approach with multi-task behavior cloning, since it is one of the most popular imitation learning paradigms. We perform experiments on a grid-world environment called Craft, as well as a robotic environment called Dial with a continuous action space. We show that in both environments, OMPN is able to perform task decomposition in both an unsupervised and weakly supervised manner, comparing favorably with strong baselines. OMPN is also able to outperform baseline behavior cloning, e.g., LSTM, in terms of sample complexity and returns, especially in partially observable environments. Our ablation study shows the contribution of both the bottom-up and the top-down recurrences on learning task decomposition. We provide visualization to confirm that OMPN learns to use a higher expansion position when the subtask is close to completion.

2 Ordered Memory Policy Network

We describe our policy architecture given the intuition described above. Our model is a recurrent policy network $p(a^t|s^t, M^t)$ where $M \in \mathcal{R}^{n \times m}$ is a block of n memory while each memory has

dimension m. We use M_i to refer to the ith slot of the memory, so $M = [M_1, M_2, ..., M_n]$. The highest-level memory is M_n while the lowest-level memory is M_1. Each memory can be thought of as the representation of a subtask. We use superscript to denote the time step t.

At each time step, our model will first transform the observation $s^t \in \mathcal{S}$ to $x^t \in \mathcal{R}^m$. This can be achieved by a domain-specific observation encoder. Then we have an ordered-memory module $M^t, O^t = OM(x^t, M^{t-1})$ to generate the next memory and the output. The output O^t is sent into a feed-forward neural net to generate the action distribution.

2.1 Ordered Memory Module

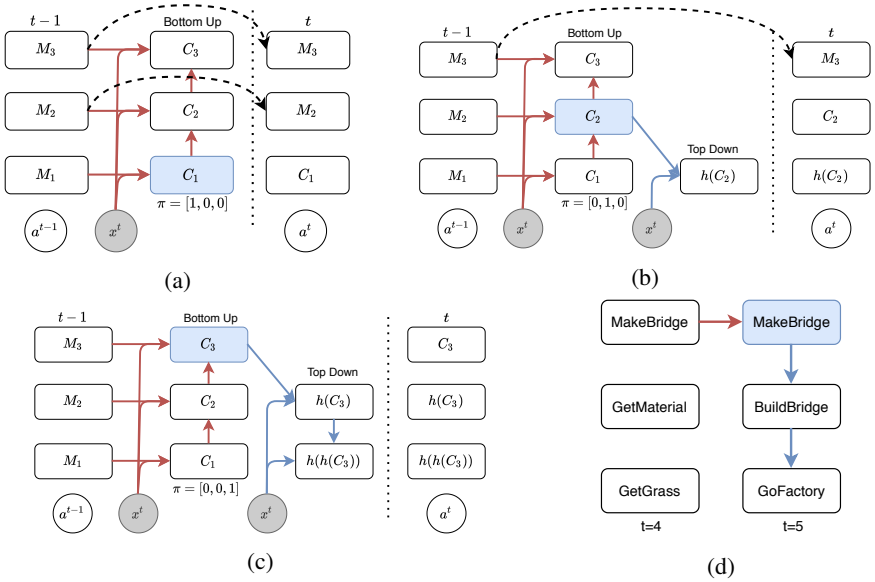


Figure 2: Dataflow of how M^{t-1} will be updated in M^t for three memory slots when the expansion position is at a (a) low, (b) middle, or (c) high position. Blue arrows and red arrows corresponding to the vertical expansions and horizontal updates. (d) is a snapshot of $t = 5$ from the grid-world example Figure 1b. The subtask-update behavior corresponds to the memory-update when the expansion position is at the high position.

The ordered memory module first goes through a bottom-up recurrence. This operation implements the *horizontal update* and updates each memory with the new observation. We define C^t to be the updated memory:

$$C_i^t = \mathcal{F}(C_{i-1}^t, x^t, M_i^{t-1})$$

for $i = 1, ..., n$ where $C_0^t = x^t$ and $\mathcal{F}$ is a cell function. Different from our mental diagram, we make it an recurrent process since the high-level memory might be able to get information from the updated lower-level memory in addition to the observation. For each memory, we also generate a score f_i^t from 0 to 1 with $f_i^t = \mathcal{G}(x^t, C_i^t, M_i^t)$ for $i = 1, ..., n$. The score f_i^t can be interpreted the probability that subtask i is completed at time t.

In order to properly generate the final *expansion position*, we would like to insert the inductive bias that the higher-level subtask is expanded only if the higher-level subtask is not completed while all the lower-level subtasks are completed, as is shown in Figure 1b. As a result we use a stick-breaking process as follows:

$$\hat{\pi}_i^t = \begin{cases} (1 - f_i^t) \prod_{j=1}^{i-1} f_j^t & 1 < i \leq n \\ 1 - f_1^t & i = 1 \end{cases}$$

Finally we have the expansion position $\pi_i^t = \hat{\pi}_i^t / \sum \hat{\pi}^t$ as a properly normalized distribution over n memories. We can also define the ending probability as the probability that every subtask is finished.

$$\pi_{end}^t = \prod_{i=1}^{n} f_i^t \tag{1}$$

Then we use a top-down recurrence on the memory to implement the vertical expansion. Starting from $\hat{M}_n^t = 0$, we have

$$\hat{M}_i^t = h(\overleftarrow{\pi}_{i+1}^t C_{i+1}^t + (1 - \overleftarrow{\pi}_{i+1}^t)\hat{M}_{i+1}^t, x^t),$$

where $\overrightarrow{\pi}_i^t = \sum_{j \geq i} \pi_j^t$, $\overleftarrow{\pi}_i^t = \sum_{j \leq i} \pi_j^t$, and h can be any cell function. Then we update the memory in the following way:

$$M^t = M^{t-1}(1 - \overrightarrow{\pi}^t) + C^t \pi^t + \hat{M}^t(1 - \overleftarrow{\pi}^t) \tag{2}$$

where the output is read from the lowest-level memory $O^t = M_1^t$. For better understanding purpose, we show in Figure 2 how M^{t-1} will be updated into M^t with $n = 3$, when the expansion position is at a high, middle and low position respectively. The memory higher than the expansion position will be preserved, while the memory at and lower than the expansion position will be over-written. We also take the snapshot of $t = 5$ from our the early example in Figure 1b and show that the subtask-update behavior corresponds to our memory-update when the expansion position is at the high position.

Although we show only the discrete case for illustration, the vector π^t is actually continuous. As a result, the whole process is fully differentiable and can be trained end-to-end. More details can be found in the appendix A.

2.2 Unsupervised Task Decomposition with Behavior Cloning

We describe how to train our model using behavior cloning, and how to perform task decomposition. Suppose we have an action space $\mathcal{A}$. We first augment this action space with $\mathcal{A}' = \mathcal{A} \cup \{done\}$, where $done$ is a special action. Then we can modify the action distribution accordingly:

$$p'(a^t|s^t) = \begin{cases} p(a^t|s^t)(1 - \pi_{end}^t) & a^t \in \mathcal{A} \\ \pi_{end}^t & a^t = done \end{cases}$$

Then for each demonstration trajectory $\tau = \{s^t, a^t\}_{t=1}^T$, we transformed it into $\tau' = \tau \cup \{s^{T+1}, a^{T+1} = done\}$, which is essentially telling the model to output $done$ only after the end of the trajectory. This process can be achieved on both discrete and continuous action space without heavy human involvement described in Appendix A. Then we will maximize $\sum_{t=1}^{T+1} \log p'(a^t|s^t)$ on τ'. We find that including π_{end}^t into the loss is crucial to prevent our model degenerating into only using the lowest-level memory, since it provides the signal to raise the expansion position at the end of the trajectory.

As is illustrated in Figure 2c, we expect the expansion position to be high if the low-level subtasks are completed. Hence we can achieve unsupervised task decomposition by monitoring the behavior of π^t. To be specific, we define $\pi_{avg}^t = \sum_{i=1}^n i\pi_i^t$ as the expected expansion position. Given π_{avg}, we consider different methods to recover the subtask boundaries.

Top-K In this method we choose the time steps of K largest π_{avg} to detect the boundary, where K is the desired number of sub-tasks. We find that this method is suitable for the discrete action space, where there is a very clear boundary between subtasks.

Thresholding In this method we standardize the π_{avg} into $\hat{\pi}_{avg}$ from 0 to 1, and then we compute a Boolean array $\mathbb{1}(\pi_{avg} > thres)$, where $thres$ is from 0 to 1. We retrieve the subtask boundaries from the ending of each $True$ segments. We find this method is suitable for continuous control settubgs, where the subtask boundaries are more ambiguous and smoothed out accross time steps.

3 Related Work

Our work is related to option discovery and hierarchical imitation learning. The existing option discovery works have focused on building a probabilistic graphical model on the trajectory, with options as latent variables. DDO (Fox et al., 2017) proposes an iterative EM-like algorithm to discover multiple level of options from the demonstration. DDO was later applied in the continuous action space (Krishnan et al., 2017) and program modelling (Fox et al., 2018). Recent works like compILE (Kipf et al., 2019) and VALOR (Achiam et al., 2018) also extend this idea by incorporating

more powerful inference methods like VAE (Kingma & Welling, 2013). Lee (2020) also explore unsupervise task decompostion via imitation, but their method is not fully end-to-end, requires an auxiliary self-supervision loss, and does not support multi-level structure. Our work focuses on the role of neural network inductive bias in discovering re-usable options or subtasks from demonstration. We do not have an explicit "inference" stage in our training algorithm to infer the option/task ID from the observations. Instead, this inference "stage" is implicitly designed into our model architecture via the stick-breaking process and expansion position. Based on these considerations, we choose compILE as the representative baseline for this field of work.

Our work is also related to Hierarchical RL (Vezhnevets et al., 2017; Nachum et al., 2018; Bacon et al., 2017). These works usually propose an architecture that has a high-level controller to output a goal, while the low-level architecture takes the goal and outputs the primitive actions. However, these works mainly deal with the control problem, and do not focus on learning task decomposition from the demonstration. The most related work in this field is Relay Policy Learning (Gupta et al., 2019), which proposes to perform hierarchical imitation learning first as pretraining, and hierarchical RL as finetuning. Their hierarchical IL algorithm takes the advantage of the fact that their goals and states are within the same space, so that they are able to re-label the unstructured demonstrations to train a hierarchical goal-conditioned policy. We argue that this method is only restricted for the goal-conditioned policies. Our approach does not have such constraints. In addition to the option framework, our work is closely related to Hierarchical Abstract Machine (HAM) (El Hihi & Bengio, 1996). Our concept of subtask is similar to the finite state machine (FSM). The horizontal update corresponds to the internal update of the FSM, while the vertical expansion corresponds to calling the next level of the FSM. Our stick-breaking process is also a continuous realization of the idea that low-level FSM transfers control back to high-level FSM at completion.

Recent work (Andreas et al., 2017) introduces the modular policy networks for reinforcement learning so that it can be used to decompose a complex task into several simple subtasks. In this setting, the agent is provided a sequence of subtasks, called *sketch*, at the beginning. Shiarlis et al. (2018) propose TACO to jointly learn sketch alignment with action sequence, as well as imitating the trajectory. This work can only be applied in the "weakly supervised" setting, where they have some information like the sub-task sequence. Nevertheless, we also choose TACO (Shiarlis et al., 2018) as one of our baselines.

Incorporating varying time-scale for each neuron to capture hierarchy is not a new idea (Chung et al., 2016; El Hihi & Bengio, 1996; Koutnik et al., 2014). However, these works do not focus on recovering the structure after training, which makes these methods less interpretable. Shen et al. (2018) introduce Ordered Neurons and show that they can induce syntactic structure by examining the hidden states after language modelling. However ONLSTM does not provides mechanism to achieve the top-down and bottom-up recurrence. Our model is mainly inspired by the Ordered Memory (Shen et al., 2019). However, unlike previous work our model is a decoder expanding from root task to subtasks, while the Ordered Memory is an encoder composing constituents into sentences. Recently Mittal et al. (2020) propose to combine top-down and bottom-up process. However their main motivation is to handle uncertainty in the sequential prediction and they do not maintain a hierarchy of memories with different update frequencies.

4 EXPERIMENT

We would like to evaluate whether OMPN is able to jointly learning task decomposition during behavior cloning. We perform experiments with a discrete action space on a grid world as well as a continuous action space on a 3D robot arm. We experiment in both unsupervised and weakly supervised settings. For the unsupervised setting, we did not provide any task information. For the weakly supervised setting, we provide the subtask sequence, or sketch, in addition to the observation. We use a one-layer GRU to encode the sketch and concatenate it with the observation. This is similar to the setting in Shiarlis et al. (2018). We use $noenv$ and $sketch$ to denote these two settings respectively. We choose compILE to be our unsupervised baseline while TACO to be our weakly supervised baseline.

We report two metrics for task decomposition: F1 scores and alignment accuracy. The F1 scores is computed by comparing the predicted subtask boundaries with the ground-truth subtask boundaries, and are computed with different levels of tolerance, in a similar fashion as Kipf et al. (2019).

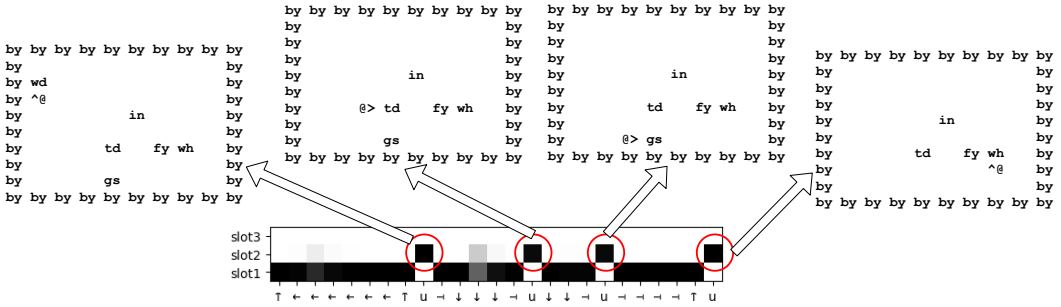
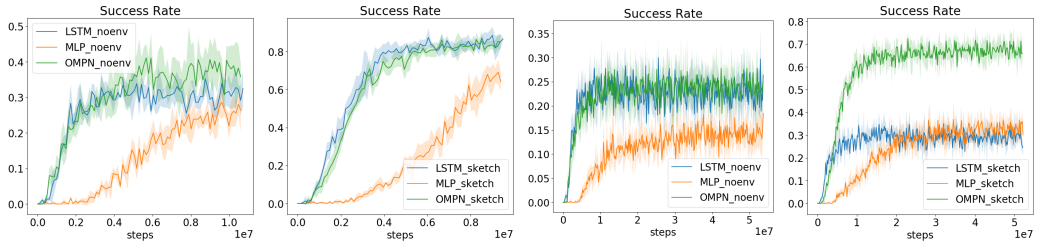


Figure 3: Unsupervised task decomposition results on Craft World. We show the expansion position π and the action sequence for the demonstration of $BuildBed$. There are four subtasks: $GetWood$, $GoToolshed$, $GetGrass$ and $GoFactory$. "by" is Boundary, "in" is Iron, "td" is Toolshed, "fy" is Factory, "wh" is Workbench, "gs" is Grass, and "wd" is Wood. In the action sequence, "u" means USE which is either picking up/using the object. At the end of each subtask, the model learns to switch to a higher expansion position.

Alignment accuracy is computed by assigning each time step a subtask and comparing the predicted alignment with the ground-truth. This is the main metric used in TACO (Shiarlis et al., 2018). More details can be found in the appendix B.

4.1 Craft World



(a) (b) (c) (d)

Figure 4: Multi-task Behavior Cloning Results on Craft. (a) and (b) are on fully observable environments while (c) and (d) are on partially observable environments. This is the average of 5 runs and the shaded error is the standard deviation. X-axis is the number of frames.

The Craft is adapted from Andreas et al. (2017)[1]. At the beginning of each episode, an agent is equipped with a task along with the sketch, e.g. $makecloth = (getgrass, gofactory)$. The original environment is fully observable. To further test our model, we make it also support partial observation by providing a self-centric window and masking out the observation outside the window.

We first demonstrate the behavior cloning results of our policy network in Figure 4. We demonstrate the effectiveness of OMPN by comparing it to an LSTM or simply an MLP. We find that sequential models are better than memory-less MLP models in these environments with a higher returns and success rates. OMPN has similar performance as the LSTM in the fully observable cases. However, it has significantly better performance than the LSTM in the partially observable environment, where success is more dependent on model's ability to store longer-term information, e.g., remember the object locations to pick it up later. We also find that providing more environment information leads to better returns, which is expected.

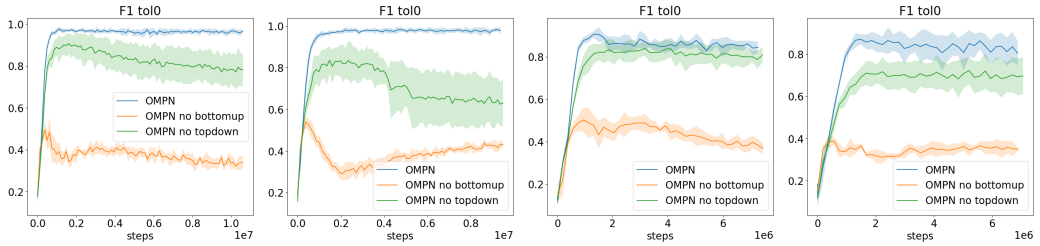
We demonstrate our model's ability to learn task decomposition in Table 1. Our results show that OMPN is able to outperform both baselines in both unsupervised and weakly-supervised settings with a higher F1(tol=0) and alignment accuracy. More details about the implementations of the baselines and hyper-parameter search are in the appendix D and E.

[1] https://github.com/jacobandreas/psketch

We visualize the unsupervised task decomposition results in Figure 3. We investigate $BuildBed$, which has four subtasks: $GetWood$, $GoToolshed$, $GetGrass$ and $GoFactory$. We find that at the end of each subtask, the model learns to switch to a higher expansion position, while within each subtask, the model learns to maintain the lower expansion position. This suggests that by adding the ordered memory inductive bias, the model use the long-term information in the higher-level memory at the end of each subtask, while the model use the short-term information in the low-level memory within the execution of each subtask. We also show the observations at each high expansion positions. The visualization suggests that high expansion positions correspond to facing the target object/location for each subtask.

	Full Obs			Partial Obs		
	F1(tol=0)	F1(tol=1)	Align. Acc.	F1(tol=0)	F1(tol=1)	Align. Acc.
OMPN + noenv	96(0.8)	97(0.8)	95(1)	84(2.6)	85(2)	88(1.7)
compILE	36(5.6)	97(0.6)	86(1.4)	42(7.2)	57(4.8)	54(1.4)
OMPN + sketch	98(0.9)	99(0.8)	98(1.4)	87(2.3)	88(2.5)	86(3.7)
TACO	-	-	85(3.6)	-	-	66(2.2)

Table 1: Task decomposition results on Craft World. For OMPN and TACO, the results are read from the time steps of highest returns. For compILE, the results are read from time steps of highest reconstruction accuracy. The numbers in the parenthesis are the standard deviation.



(a) Full + NoEnv (b) Full + Sketch (c) Full + NoEnv (d) Full + Sketch

Figure 5: Ablation study on different settings with F1 score ($tol = 0$). After removing either the top-down recurrence and bottom-up recurrence, the structure learning ability would be decreased. The x-axis is number of frames.

In order to validate the design of our model architecture, we perform an ablation study by removing either the bottom-up recurrence or the top-down recurrence. The F1 score results can be found in Figure 5. We find that after removing each components, the model's ability to learn the structure is decreased. We find that, in all cases, removing the bottom-up recurrence hurts more than removing the top-down recurrence. We hypothesize that it's because the bottom-up recurrence incorporates new observation into each subtask, so that the stick-breaking process can better decide whether the subtask is finished or not.

4.2 DIAL

In this experiment, we have a robotic setting similar in Shiarlis et al. (2018) where a JACO 6DoF manipulator interact with a large number pad[2]. For each episode, the sketch is a sequence of numbers to be pressed. We generate 1400 trajectories for imitation learning with each sketch being 4 digits. We use the thresholding method to detect the subtask boundaries.

Our task decomposition results can be found in Table 2. We find that in the unsupervised setting, compILE failed to learn anything meaningful on this environment, while OMPN is able to have a reasonable task alignment accuracy.

[2] https://github.com/KyriacosShiarli/taco/tree/master/taco/jac

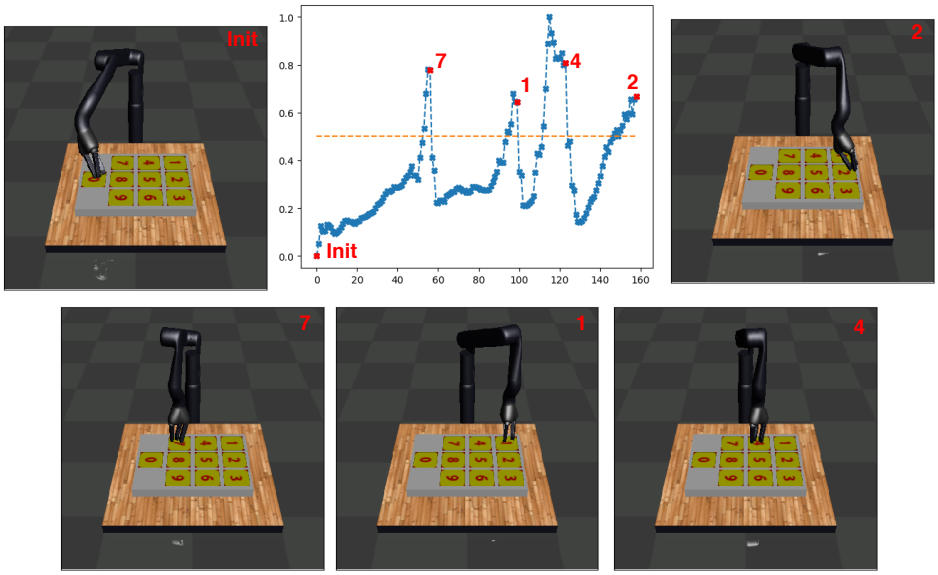


Figure 6: Unsupervised task decomposition of Dial domain. The task is to press the number sequence $[7, 1, 4, 2]$. We plot the $\hat{\pi}_{avg}$ for every time steps as well as plotting the threshold 0.5. The task boundary is detected as the last time step of a segment above the threshold. The frames at the detected task boundary show that the robot just finishes each subtask.

Our performance in the weakly-supervised setting is similar to the unsupervised setting, however we find that TACO achieves almost perfect alignment accuracy. We argue that this might be caused by the fact that our model has a very trivial way of incorporating sketch information. In the Dial domain, the trajectory length is much longer, and as a result, a more effective way would be using attention over the sketch sequence to properly feed the task information into the model, instead of using one constant vector over time as we did. We leave it to future work.

	Align. Acc.
OMPN + noenv	90.6(1.4)
compILE	45(5.21)
OMPN + sketch	91.3(0.8)
TACO	98.5(0.01)

Table 2: Task alignment accuracy in Dial. For OMPN we use $thres = 0.5$ for $noenv$ and $thres = 0.4$ for $sketch$. More results with different threshold in the appendix C

We visualize the unsupervised task decomposition result in Figure 6. We plot the $\hat{\pi}_{avg}$ with the number sequence $[7, 1, 4, 2]$. We find that, instead of having a clear subtask boundary, the boundary between subtasks is more ambiguous which results in a more gradual change in the expansion position. Nevertheless we can still find that there is always a segment of high expansion position near the end of each subtask. The visualization shows that the detected subtask boundaries correspond to the frames where the robotic arm reaches the correct numbers.

5 Conclusion

In this work, we investigate the problem of unsupervised task decomposition from the demonstration trajectory. We propose a novel Ordered Memory Policy Network (OMPN) that can represent the subtask structure. Our experiments show that in behavior cloning setting, OMPN learns to recover the ground truth subtask boundary in both unsupervised and weakly supervised settings. In the future, we plan to investigate OMPN with other imitation algorithms like GAIL (Ho & Ermon, 2016) as well as developing a novel control algorithm based on the inductive bias for faster adaptation to compositional combinations of the subtasks.

REFERENCES

Joshua Achiam, Harrison Edwards, Dario Amodei, and Pieter Abbeel. Variational option discovery algorithms. *arXiv preprint arXiv:1807.10299*, 2018. 1, 4

Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In *International Conference on Machine Learning*, pp. 166–175, 2017. 1, 5, 6

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017. 5

Junyoung Chung, Sungjin Ahn, and Yoshua Bengio. Hierarchical multiscale recurrent neural networks. *arXiv preprint arXiv:1609.01704*, 2016. 5

Salah El Hihi and Yoshua Bengio. Hierarchical recurrent neural networks for long-term dependencies. In *Advances in neural information processing systems*, pp. 493–499, 1996. 5

Roy Fox, Sanjay Krishnan, Ion Stoica, and Ken Goldberg. Multi-level discovery of deep options. *arXiv preprint arXiv:1703.08294*, 2017. 1, 4

Roy Fox, Richard Shin, Sanjay Krishnan, Ken Goldberg, Dawn Song, and Ion Stoica. Parametrized hierarchical procedures for neural programming. *ICLR 2018*, 2018. 4

Abhishek Gupta, Vikash Kumar, Corey Lynch, Sergey Levine, and Karol Hausman. Relay policy learning: Solving long-horizon tasks via imitation and reinforcement learning. *arXiv preprint arXiv:1910.11956*, 2019. 1, 5

Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *Advances in neural information processing systems*, pp. 4565–4573, 2016. 8

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013. 5

Thomas Kipf, Yujia Li, Hanjun Dai, Vinicius Zambaldi, Alvaro Sanchez-Gonzalez, Edward Grefenstette, Pushmeet Kohli, and Peter Battaglia. Compile: Compositional imitation learning and execution. In *International Conference on Machine Learning*, pp. 3418–3428. PMLR, 2019. 1, 4, 5

Jan Koutnik, Klaus Greff, Faustino Gomez, and Juergen Schmidhuber. A clockwork rnn. In *International Conference on Machine Learning*, pp. 1863–1871, 2014. 5

Sanjay Krishnan, Roy Fox, Ion Stoica, and Ken Goldberg. Ddco: Discovery of deep continuous options for robot learning from demonstrations. *arXiv preprint arXiv:1710.05421*, 2017. 4

Sang-Hyun Lee. Learning compound tasks without task-specific knowledge via imitation and self-supervised learning. In *International Conference on Machine Learning*, 2020. 5

Sarthak Mittal, Alex Lamb, Anirudh Goyal, Vikram Voleti, Murray Shanahan, Guillaume Lajoie, Michael Mozer, and Yoshua Bengio. Learning to combine top-down and bottom-up signals in recurrent neural networks with attention over modules. *arXiv preprint arXiv:2006.16981*, 2020. 5

Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 3303–3313, 2018. 5

Ronald Parr and Stuart J Russell. Reinforcement learning with hierarchies of machines. In *Advances in neural information processing systems*, pp. 1043–1049, 1998. 1

Yikang Shen, Shawn Tan, Alessandro Sordoni, and Aaron Courville. Ordered neurons: Integrating tree structures into recurrent neural networks. In *International Conference on Learning Representations*, 2018. 5

Yikang Shen, Shawn Tan, Arian Hosseini, Zhouhan Lin, Alessandro Sordoni, and Aaron C Courville. Ordered memory. In *Advances in Neural Information Processing Systems*, pp. 5037–5048, 2019. 5, 11

Kyriacos Shiarlis, Markus Wulfmeier, Sasha Salter, Shimon Whiteson, and Ingmar Posner. Taco: Learning task decomposition via temporal alignment for control. In *International Conference on Machine Learning*, pp. 4654–4663, 2018. 1, 5, 6, 7, 12

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016. 1

Alec Solway, Carlos Diuk, Natalia Córdova, Debbie Yee, Andrew G Barto, Yael Niv, and Matthew M Botvinick. Optimal behavioral hierarchy. *PLOS Comput Biol*, 10(8):e1003779, 2014. 1

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. 1

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. *arXiv preprint arXiv:1703.01161*, 2017. 5

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019. 1

A OMPN ARCHITECTURE DETAILS

We use the gated recursive cell function from Shen et al. (2019) in the top-down and bottom up recurrence. We use a two-layer MLP to compute the score f_i for the stick-breaking process. For the initial memory M^0, we send the environment information into the highest slot while keep the rest of the slots to be zeros. If unsupervised setting, then the every slot is initialized as zero. At the first time step, we also skip the bottom-up process and hard code π^1 such that the memory expands from the highest level. This is used to make sure that at first time step, we could propagate our memory with the expanded subtasks from root task. In our experiment, our cell functions does not share the parameters. We find that to be better than shared-parameter.

We set the number of slots to be 3 in Craft and 2 in Dial, and each memory has dimension 128. We use Adam optimizer to train our model with $\beta_1 = 0.9, \beta_2 = 0.999$. The learning rate is 0.001 in Craft and 0.0005 in Dial. We set the length of BPTT to be 64 in both experiments. We clip the gradients with L2 norm 0.2. The observation has dimension 1076 in Craft and 39 in Dial. We use a linear layer to encode the observation. After reading the output O^t, we concatenate it with the observation x^t and send them to a linear layer to produce the action.

In section 2, we describe that we augment the action space into $\mathcal{A} \cup \{done\}$ and we append the trajectory $\tau = \{s_t, a_t\}_{t=1}^T$ with one last step, which is $\tau \cup \{s_{t+1}, done\}$. This can be easily done if the data is generated by letting an expert agent interact with the environment as in Algorithm 1. If you do not have the luxury of environment interaction, then you can simply let $s_{T+1} = s_T, a_{T+1} = done$. We find that augmenting the trajectory in this way does not change the performance in our Dial experiment, since the task boundary is smoothed out across time steps for continuous action space, but it hurts the performance for Craft, since the final action of craft is usually USE, which can change the state a lot.

Algorithm 1: Data Collection with Gym API

```
env = gym.make(name)
done = False
obs = env.reset()
traj = []
repeat
    action = bot.act(obs)
    nextobs, reward, done = env.step(action)
    traj.append((obs, action, reward))
    obs = nextobs
until done is True;
traj.append((obs, done_action))
```

B TASK DECOMPOSITION METRIC

B.1 F1 SCORES WITH TOLERANCE

For each trajectory, we are given a set of ground truth task boundary gt of length L which is the number of subtasks. The algorithm also produce L task boundary predictions. This can be done in OMPN by setting the correct K in $topK$ boundary detection. For compILE, we set the number of segments to be equal to N. Nevertheless, our definition of F1 can be extended to arbitaray number of predictions.

$$precision = \frac{\sum_{i,j} match(preds_i, gt_j, tol)}{\#predictions)}$$

$$precision = \frac{\sum_{i,j} match(gt_i, preds_j, tol)}{\#ground\ truth}$$

where the $match$ is defined as

$$match(x, y, tol) = [y - tol \leq x \leq y + tol]$$

where [] is the Iverson bracket. The tolerance

B.2 TASK ALIGNMENT ACCURACY

This metric is taken from Shiarlis et al. (2018). Suppose we have a sketch of 4 subtasks $b = [b1, b2, b3, b4]$ and we have the ground truth assignment $\xi_{true} = \{\xi^t_{true}\}^T_{t=1}$. Similar we have the predicted alignment ξ_{pred}. The alignment accuracy is simply

$$\sum_t [\xi^t_{pred} == \xi^t_{true}]$$

For OMPN and compILE, we obtain the task boundary first and construct the alignment as a result. For TACO, we follow the original paper to obtain the alignment.

C MORE ON OMPN TASK DECOMPOSION

C.1 CRAFT

We show the full table of Craft in Table 3. We also display more visuzliation of task decomposition results from OMPN in Figure 7.

Full Obs						
	Unsupervise (noenv)			Weakly Supervise (sketch)		
	F1(tol=0)	F1(tol=1)	Align. Acc.	F1(tol=0)	F1(tol=1)	Align. Acc.
OMPN	96(0.8)	97(0.8)	95(1)	98(0.9)	99(0.8)	98(1.4)
compile	36(5.6)	97(0.6)	86(1.4)	25(0.1)	98(0.8)	86(1)
TACO	-	-	-	-	-	85(3.6)
Partial Obs						
	Unsupervise (noenv)			Weakly Supervise (sketch)		
	F1(tol=0)	F1(tol=1)	Align. Acc.	F1(tol=0)	F1(tol=1)	Align. Acc.
OMPN	84(2.6)	85(2)	88(1.7)	87(2.3)	88(2.5)	86(3.7)
compILE	42(7.2)	57(4.8))	54(1.4)	42(1.9)	53(3.1))	51(0.4)
TACO	-	-	-	-	-	66(2.2)

Table 3: Task decomposition results on Craft World. For OMPN and TACO, the results are read from the time steps of highest returns. For compILE, the results are read from time steps of highest reconstruction accuracy. The numbers in the parenthesis are the standard deviation. We extend compILE to weakly supervised setting by concatenate the input with the task sketch encoding. The results is similar to the unsupervised setting.

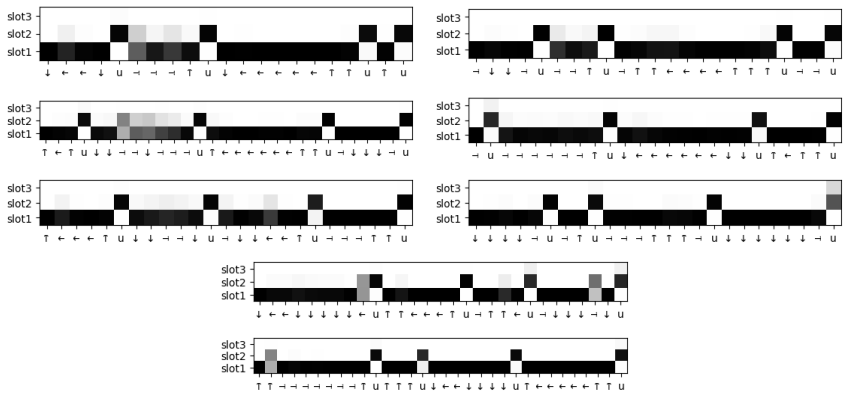


Figure 7: More results on π in Craft. The model is able to robustly switch to a higher expanding position at the end of subtasks.

C.2 DIAL

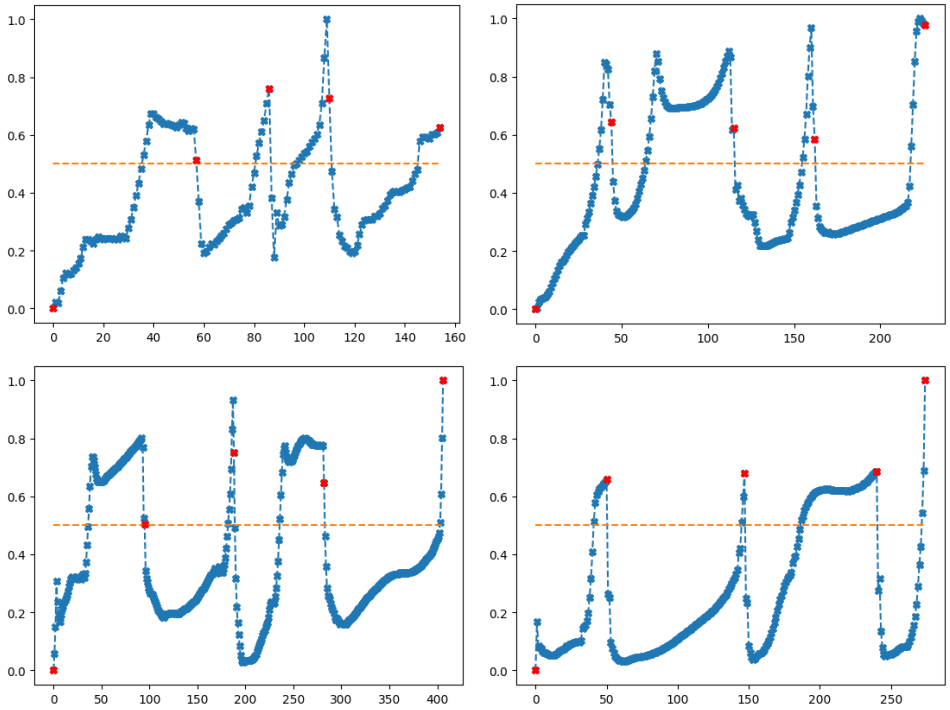
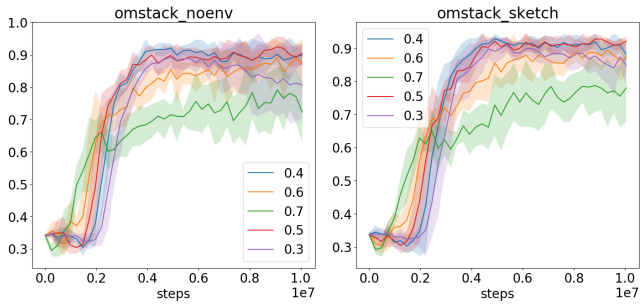


Figure 8: More task decomposition visualization in Dial

	Align. Acc. at different threshold						
	0.2	0.3	0.4	0.5	0.6	0.7	0.8
OMPN + noenv	66.9(8.7)	81.3(4.6)	88.3(3.1)	90.6(1.4)	84.4(5.7)	69.6(9.7)	53.9(8.1)
OMPN + sketch	75.2(10.1)	86.4(3.1)	91.3(0.8)	89.5(2.9)	80.5(7.4)	67.8(8.2)	53.8(8.9)

Table 4: Task alignment accuracy for different threshold in Dial.



(a) (b)

Figure 9: The learning curves of task alignment accuracy for different threshold under (a)unsupervised and (b)weakly supervised setting

We show the result of different threshold in table 4 and Figure 9. More visualizations can be found in Figure 8.

D compILE Details

latent	[concrete, gaussian]
prior	[0.3, 0.5,0.7]
kl_b	[0.05, 0.1, 0.2]
kl_z	[0.05, 0.1, 0.2]

Table 5: compILE hyperparameter search.

Our implementation of compILE is taken from the author github[3]. However, their released code only work for a toy digit sequence example. As a result we modify the encoder and decoder respectively for our environments. During our experiment, we perform the following hyper-parameter sweep on the baseline in Table 5. Although the authors use latent to be concrete during their paper, we find that gaussian perform better in our case. We find that Gaussian with $prior = 0.5$ performs the best in Craft. For Dial, these configurations perform equally bad.

We show the task alignments of compILE for Craft in Figure 10. It seems that compILE learn the task boundary one-off, which explains in Table 3 that the F1(tol=0) is low but F1(tol=1) is high. However, since the subtask ground truth can be ad hoc, this brings the question how should we decide whether our model is learning structure that makes sense or not? Further investigation in building a better benchmark/metric is required.

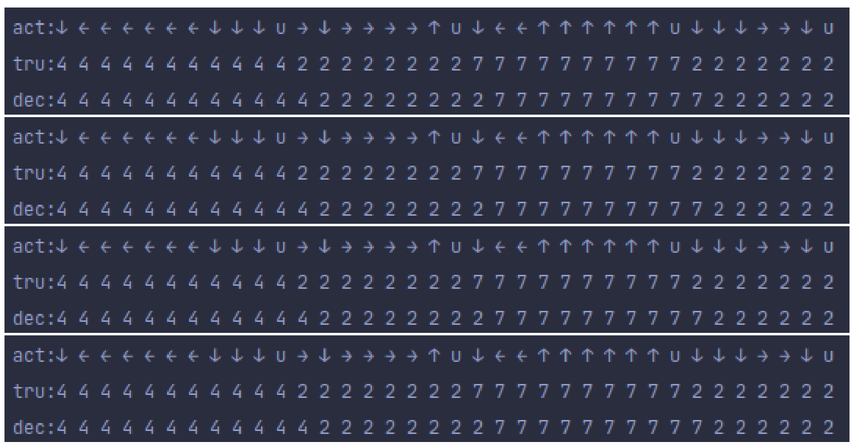


Figure 10: Task alignment results of compILE on Craft.

E TACO Details

dropout	[0.2, 0.4, 0.6, 0.8]
decay	[0.2, 0.4, 0.6, 0.8]

Table 6: TACO hyperparameter search.

We use the implementation from author github[4] and modifiy it into pytorch. Although the author also conduct experiment on Craft and Dial, they did not release the demonstration dataset they use. As a result, we cannot directly use their numbers from the paper. We also apply dropout on the prediction of $STOP$ and apply a linear decaying schedule during training. The hyperparameter search is in table 6. We find the best hyperparameter to be $0.4, 0.4$ for Craft. For Dial, the result is not sensitive to the hyperparameters.

[3] https://github.com/tkipf/compile
[4] https://github.com/KyriacosShiarli/taco