Learning to Compose Skills

Farhan Tejani[1]

Undergraduate Thesis

[1] A paper based on this work was presented at the 2017 Neural Information Processing Systems Deep Reinforcement Learning Symposium. Himanshu Sahni, Saurabh Kumar, and Charles Isbell were co-authors on that paper.

Abstract

We present a differentiable framework capable of learning a wide variety of com- positions of simple policies that we call skills. By recursively composing skills with themselves, we can create hierarchies that display complex behavior. Skill networks are trained to generate skill-state embeddings that are provided as inputs to a trainable composition function, which in turn outputs a policy for the overall task. Our experiments on an environment consisting of multiple collect and evade tasks show that this architecture is able to quickly build complex skills from simpler ones. Furthermore, the learned composition function displays some transfer to unseen combinations of skills, allowing for zero-shot generalizations. We also design experiments for learning both skills as well as optimal compositions of those skills from scratch. This allows for the agent to more intelligently search the space of possible skills rather than having a human hand-design the composition functions, which may be limited in scope.

Introduction

A key property of intelligent agents is the ability to learn simple skills throughout their lifetimes and compose them together to solve complicated tasks. Yet, traditional reinforcement learning (RL) agents lack this ability, making it hard to learn in environments with long term dependencies. Recent advances in using deep neural networks as function approximators allow for learning in high dimensional state spaces [16], but do not address this fundamental problem.

In RL, an agent interacts with a dynamic environment and learns to maximize the notion of a long-term reward. The task is typically characterized as a Markov Decision Process (MDP) defined by the tuple $\{S, A, T, R, \gamma\}$ of states, actions, transition function, reward, and discount factor. A policy $\pi(S) \rightarrow P(A)$ maps each state to a probability distribution over actions that the agent should take. The agent must learn to optimize this policy in order to maximize the long-term expected discounted reward that it obtains.

Hierarchical RL [6, 25] offers a solution to the generalization problem by decomposing a single complicated task into a hierarchy of simpler subtasks, often times using intrinsic rewards to motivate underlying learners. A related strategy is to use options [22], a set of policies with fixed, possibly stochastic, initiation and termination criteria, that are made available to the agent along with base environment actions. Both approaches focus on decomposing a difficult problem into a sequence of simpler sub-goals. The motivation behind this work is that solving problems in the real world rarely calls for optimal sequential decompositions of arbitrary tasks; instead, a set of basic skills can be composed in multiple interesting ways to exhibit complex behavior.

A major distinction between our work and recent attempts to learn an optimal sequence of sub-goals [1, 8, 17] is that our framework can learn a much richer set of skill compositions. For example, in the game of Pacman, an agent must learn to collect food pellets while also avoiding enemy ghosts. In the usual view of hierarchical RL, a sub-goal or option, such as "navigate to food pellet A" or "evade enemy ghost B," would be activated one at a time and the agent must learn to alternate between them to complete the overall task of “collect A while avoid B.” A better approach is to learn a policy that composes both sub-goals, i.e. identifies paths to the food pellet that also keep Pacman far away from ghosts. In this work, we consider a subset of compositions defined by Linear Temporal Logic (LTL) [3, 18]. A wide variety of common RL tasks can be specified using the temporal modal operators defined in LTL: next (O), always (#), eventually (♦), and until (*U*), along with the basic logic connectives: negation(¬), disjunction (∨), conjunction (∧) and implication (→) [14]. The Pacman task above can be translated into LTL as ¬G U (♦F1 ∧ ♦F2 ∧ . . . ♦Fn), where G is the proposition that the Pacman occupies the same location as any of the ghosts and F1 through Fn are the corresponding propositions for all the food pellets. Thus, the LTL sentence can be interpreted as "do not get eaten by a ghost until all the food pellets have been collected."

Our main contribution in this work is the expression of these compositions as differentiable functions. Representations of the individual skill policies are fed to this function as inputs and a representation for the composed task policy is produced. Skill policies are learned only once, and a wide variety of compositions can be created after the fact. We show that learning to compose skills is more efficient than learning to sequence those skills as is typically done in hierarchical RL. Moreover, we show how recursive compositions can be used to create rich hierarchies for more complicated behavior.

A challenge with trainable compositions is that skill policies must be represented in a differentiable manner so that they can be utilized inside the composition function. In most modern RL domains, policies are represented as deep neural networks, with the outputs normalized to form a probability distribution over actions. The action distribution alone, however, may not encode enough information on the importance of a sub-goal in the current state to arbitrate between competing sub-goals. On the other hand, the entire policy network may contain multiple layers and thousands of weights. Trying to learn a composition function over that would be very challenging. Therefore, we use a special architecture for training skill policies which allows us to embed information on the skill and the state in a single layer of the network. We call these *skill-state embeddings*. Each embedding is then fed into a composition layer which learns to solve the overall task. The cost of acquiring skills is cheap and training the composition function is faster than learning the overall task from scratch. More importantly, the skills can be reused for different compositions. Finally, we show that the composition function itself shows some transfer to unseen tasks, allowing for zero-shot task generalization.

Literature Review

Our work is related to a family of hierarchical RL methods [4, 7, 10, 11, 12, 13]. Approaches in hierarchical RL typically learn the sub-goal policies and a meta-policy simultaneously, using intrinsic rewards for completion of sub-goals [13] or by tying parameters across different modules [1] or by adopting a meta-learning approach [8]. A fundamental difference in our approach is that instead of learning optimal decompositions for a given complex task, we take the view of learning optimal compositions given a set of base tasks. The act of learning to grill a pancake [9] does not require us learn an optimal, sequential decomposition of cooking by interacting with a wide variety of recipes. Instead, it is much easier if a base set of skills, such as whisking eggs, measuring flour, heating the pan etc. can be composed to occur simultaneously, in sequence, optionally or held true until another sub-goal is satisfied. The advantage of the ComposeNet architecture is that the overall tasks can be constructed post-hoc and pre-learned policies for skills can be quickly composed together to solve unseen tasks. This achieves much greater re-use of skills and quicker transfer to composed tasks as the skill networks are frozen after training once. Oh et al. [17] describe a framework to optimally sequence skills that can be learned in isolation from the main task. But they limit their discussion to sequence of sub-goals, like program instructions, with occasional interruptibility for a higher priority task.

Our work is also similar to a related framework in hierarchical RL called options [2, 22]. The key difference here is that the skills are not provided to the agent as augmentations of its action set. Instead, our model learns skill-state embeddings, which are provided to a composition function which then learns to aggregate them and output an embedding for the overall task. Typically, multiple options cannot be activated in parallel. At a given state, an agent may activate a legal option and chose an action according to the policy prescribed by the option. After choosing the option, it must follow it for at least one step before activating another option or a taking a primitive action. In contrast, by composing skill-state embeddings, the agent is able to arbitrate between multiple sub-policies simultaneously to form optimal behavior according to the composed task.

In this sense, modular reinforcement learning is a closer analogue to our approach [1, 5, 19, 23]. The skills can be regarded as sub-modules and the compositional layer as an aggregator that combines each skill's suggestion into a policy for the overall task. A crucial distinction in our work is that the skill modules do not provide direct policy recommendations to the aggregator and nor does the aggregator output a policy. Instead, they both learn to create skill-state embeddings for their particular skills or tasks. A final layer transforms embeddings in this space into policy actions. Representing submodule policies with embeddings allows us to create a richer description of the state, conditioned on each skill, in a way that allows us to create a trainable composition function.

Also related is work in multi-task RL, such as by van Seijen et al. [24], who use a Hybrid Reward Architecture agent to solve the game of Ms. Pacman. Our work can be seen as lying between multi-task and hierarchical RL as our framework is capable of solving simultaneous goals, sequential goals and also optional goals, goals that must be held true until other goals are satisfied, etc.

Materials and Methods

The ComposeNet architecture allows learning of skill-state embeddings which can be used inside a differentiable composition function. Each skill has its own network trunk but the final layer, called the policy layer, is shared across all the skills (Figure 1a). Each trunk is trained for its particular skill in isolation but gradients from all the skills are applied to the policy layer. The trunks are therefore forced to encode information about their particular skill as well as the agent state in their topmost layer. The policy layer is learning to take embeddings from any skill trunk and output a policy corresponding to that skill. This can be seen as a reversal of many multi-task learning architectures where a common input trunk is used with branches at the top for different tasks [24]. In that case, a common embedding is learned for all tasks. Our goal is the opposite, i.e. to learn unique embeddings for each task and a common layer that can take any embedding and output the corresponding policy.

Now that we have a way to embed skill and state information in a single vector, we can combine two or more embeddings to create a new embedding for a composition of those skills. A composition, then, is a mapping from embeddings of all relevant skills to an embedding of the composed task.

Note that the policy function is agnostic to where the embeddings are coming from. This means that the same function must learn to map embeddings of all skills and any of their compositions to a policy. This property allows us to do recursive compositions of composed embeddings with other skills and create hierarchies of behavior. If both functions are differentiable, gradients with respect to the parameters of the composition function, C, can be formed using gradient based RL methods.

In practice, two skill-state embeddings are concatenated end-to-end and fed to a fully connected layer, or the composition layer, which acts as the composition function C. The output of the composition layer is the same dimensionality as the skill-state embeddings and is fed into the pretrained policy layer whose output is now treated as a policy for the composed task (Figure 1b). Hence, the composition layer must learn to take two skill-state embeddings and output an embedding for the composed task. It is assumed that the correct skills for the task are provided to the agent and the form of the composition is known. This can be seen as a semi-supervised way of representing the task.

ComposeNet is trained as follows. First, skill trunks and the shared policy layer (without the composition layer) are trained simultaneously using asynchronous actor critic (A3C) [15]. Once converged, the weights of the skill trunks and the policy layer are frozen. Now a task consisting of a composition of two or more skills is chosen and only the composition layer is trained on samples from it.

Environment and Skills

To test our approach, we devised a domain similar to Pacman, where an agent must collect or evade colored objects, Red, Green and Blue. The objects to collect remain stationary but the enemies chase the agent along the shortest path. Once an object is collected, it disappears from the map. The agent can teleport across the map if it goes out of bounds, but the objects cannot. An example task in this environment is "collect object Red while evading object Blue" ($\neg b \ U \ r$).

The agent’s state is a 15x15 pixel grayscale image of the game grid. There are six skills in this environment: collecting and evading the three objects respectively. The skills are trained separately in environments with reward functions only relevant to that skill.

We consider four types of compositions in this environment:
1. ¬p U q, collect object q while evading enemy p;
2. ♦p ∨ ♦q, collect object p or q;
3. #¬p ∧ #¬q, always evade enemy p and enemy q; and
4. ♦(p ∧ ♦q), collect object q then object p.

Results

We first train all six skills networks for about 3 million steps total (i.e. 500,000 steps on average per skill). After this, skill networks and the policy layer are frozen. This initial cost is fixed and amortized over all possible compositions.

In the graphs, we compare performance of our method (ComposeNet) to two baselines: (1) training a single network from scratch, and (2) a meta-controller approach where a network picks from relevant trained skills every step. We also experimented with using skills as options by augmenting the agent's action space. That performed worse than training from scratch on all problems, likely due to the increased number of actions. We have omitted those results for clarity.

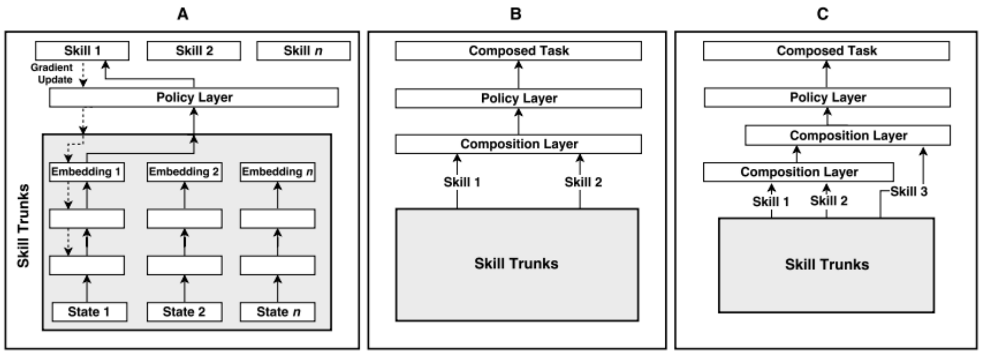
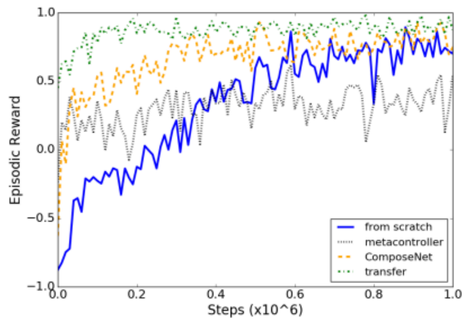


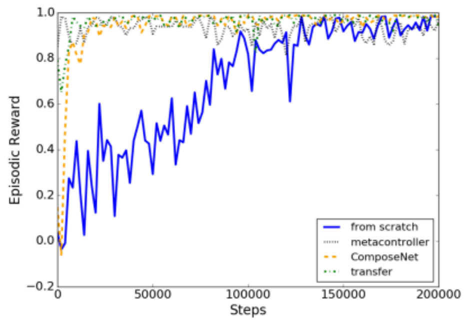
Figure 1: The ComposeNet architecture. (a) Training of the base skills. Each skill has its own trunk but shares a final layer, the policy layer, with all other skills. (b) Once the skill trunks and policy layer have been trained, skill-state embeddings are concatenated and fed into the composition layer. (c) By composing compositions of skills with other skills, one can create complex hierarchies.

Single Compositions and Zero-Shot Generalizations

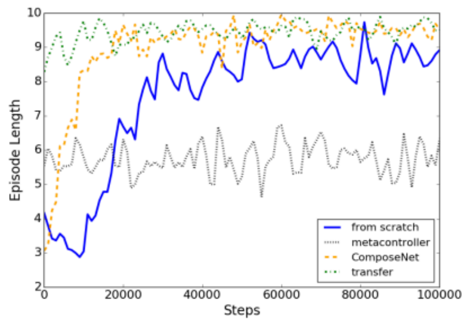
Figure 2 shows performance of ComposeNet compared to our baselines on a sample task for each type of composition. Overall, the results show that individual skills can be successfully composed with the ComposeNet architecture to near optimality and are learned faster than either of the baselines. For the "while" and "then" compositions (Figures 2a and 2d), the meta-controller initially achieves a somewhat good reward but then learning slows down significantly. This is because the meta-controller quickly learns that the skill "collect blue" may lead to a high reward. But improving the reward requires it to learn to alternate between reaching blue and evading green. Similarly, with "collect red then green", the agent may reach red then randomly stumble into green or follow the "collect green" skill only and collect red along the way accidentally. But activating them in sequence with correct timing is harder to learn. The exception is "collect red or blue", as in this case the meta-controller can select any skill at random and ensure a high reward. ComposeNet quickly learns to achieve high reward for all types of compositions. For the $p \wedge q$ composition, the meta-controller strategy completely fails to learn. Evading both objects is a hard task and actions must be chosen to evade both at the same time. Activating only one, say 'avoid red', may lead the agent towards danger, towards green. Our learned composition function ensures both are evaded simultaneously. This is an example of why the ComposeNet architecture is better suited to a wide variety of compositions than traditional hierarchical RL approaches.



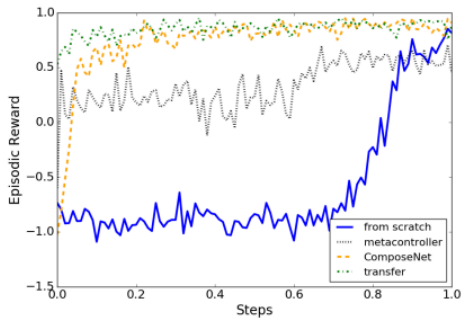
(a) $\neg g \,\mathcal{U}\, b$



(b) $\Diamond r \vee \Diamond b$



(c) $\Box \neg r \wedge \Box \neg g$



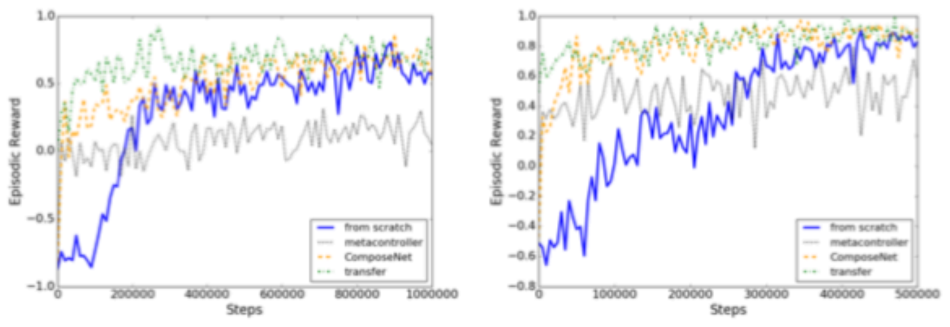
(d) $\Diamond(r \wedge \Diamond g)$

Figure 2: Performance of our method (ComposeNet) compared with baselines of metacontroller strategy and learning from scratch. In the 'transfer' condition, the composition layer is initialized with weights learned over similar tasks. (a) **Collect blue while evade green** zero-shot reward: 0.45. (b) **Collect red or blue** zero-shot reward: 0.79. (c) **Evade red and green** zero-shot episode length: 8.28 (d) **Collect red then green** zero-shot reward: 0.53.

We also tested zero-shot task generalization by training the composition layer on other tasks containing the same compositions. For example, we trained the same composition layer on all five tasks of the type ¬p U q, except ¬g U b. The learned weights were then used as initialization for the composition layer of the held-out task. Our results show that there is significant zero-shot generalization to compositions of the same type. Further training on the held-out task quickly produces near optimal rewards. For the "collect red or blue" and "evade red and green" tasks we transferred from only two other tasks, as the order of the objects does not matter in these compositions.

Hierarchies of Compositions

The ComposeNet architecture is versatile enough that the composition layer can accept itself as an input, leading to more complex hierarchies. Figure 3 shows results on two composed tasks, 'collect red while evade green and blue', (¬g ∧ ¬b) U r, and 'collect red or green while evade blue', ¬b U (r ∨ g). The networks are formed by first composing the literals in parentheses, and then composing the resulting embedding with the embedding for the third literal. For example, in



(a) $(\neg g \wedge \neg b)\,\mathcal{U}\,r$ (b) $\neg b\,\mathcal{U}\,(r \vee g)$

Figure 3: Hierarchies of compositions allow for even more complex tasks to be learned quickly. (a) Zero-shot reward: 0.01. The task of evading two enemies while collecting the third object is challenging and success rate is low for all policies. ComposeNet reaches a good rate of success fairly quickly. (b) Zero-shot reward: 0.49. In this case, the task was of collecting either one of two objects while evading the third.

Figure 3a, the embeddings for 'evade green' and 'evade blue' are composed first. The output is fed into another composition layer, along with the embedding for 'collect red'. The output of this layer is then fed to the policy layer. In the 'transfer' condition for this task, the first composition layer was initialized with weights trained on all 'evade this and that' tasks. These weights have been trained to compose two evade policies into a single policy that successfully evades both objects. The second composition layer is initialized with weights trained on all 'collect this while evade that' tasks. This layer takes as input 'collect red' embeddings and the composed embedding from the first compositional layer and produces an embedding for the complete task. Similarly, for the transfer treatment in the second task, weights from training on all 'collect this or that' tasks and all 'collect this while evade that' tasks were used to initialize the two composition layers.

The results for zero-shot generalization show that some transfer occurs to such hierarchically composed tasks, even when the training set is comprised solely of flat compositions of two literals. The task in Figure 3a is fairly challenging, so the zero-shot policy is able to collect reward only about half the time, resulting in an average reward close to zero. With a few samples from the composed task, it quickly learns a high-reward policy. In Figure 3b, the transferred policy starts with a decent zero-shot reward of close to 0.5 and also converges quickly. ComposeNet allows for this mix-and-match composition capability and reuse of learned skills, even in complicated hierarchies.

Discussion

We have presented a framework called ComposeNet which allows an agent to compose simple skills into a hierarchy to solve complicated tasks. The skills are learned separately and can be reused for multiple compositions. Key in the framework are skill-state embeddings and a trainable composition function, backed by our ablative studies. Moreover, when testing on composed tasks it has never seen before, ComposeNet shows some zero-shot generalization capability, and quickly converges with few environment samples. This suggests that the ability to compose skills in this domain may be transferable.

This research aligns with the current trajectory of option learning in the field of RL. Recent work from McGill's Reasoning and Learning Laboratory [2] aims to learn options, which are temporally extended actions such as "go to pellet A" from scratch. However, a key difference is that with this framework, an agent will naturally learn which options are necessary for solving the overall task – these options (or skills) do not need to be pre-defined as in our work. It would be interesting to combine these ideas to learn base skills as well as composition functions.

Future Work

Future work in this area includes trying ComposeNet on more complicated domains such as Minecraft. We have demonstrated that the operators: (1) U (collect this while evade that), (2) $\wedge$ (evade this and that), (3) $\vee$ (collect this or that), and (4) ♦($\wedge$ ♦) (collect this then that), can be learned quickly with pre-trained base skills. Work on learning other types of compositions is ongoing. Additionally, we are currently working on a mathematical derivation for learning options in reinforcement learning end-to-end simply by specifying the number of desired options. This algorithm is being tested on simulated environments, such as Atari games like Pacman.

References

[1] J. Andreas, D. Klein, and S. Levine. Modular multitask reinforcement learning with policy sketches. arXiv preprint arXiv:1611.01796, 2016.

[2] P.-L. Bacon, J. Harb, and D. Precup. The option-critic architecture. In AAAI, pages 1726–1734, 2017.

[3] C. Baier, J.-P. Katoen, and K. G. Larsen. Principles of model checking. MIT press, 2008.

[4] A. G. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. Discrete Event Dynamic Systems, 13(4):341–379, 2003.

[5] S. Bhat, C. L. Isbell, and M. Mateas. On the difficulty of modular reinforcement learning for real-world partial programming. In Proceedings of the National Conference on Artificial Intelligence, volume 21, page 318. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[6] T. G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. J. Artif. Intell. Res.(JAIR), 13(1):227–303, Nov. 2000.

[7] T. G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. J. Artif. Intell. Res.(JAIR), 13:227–303, 2000.

[8] K. Frans, J. Ho, X. Chen, P. Abbeel, and J. Schulman. Meta learning shared hierarchies. arXiv preprint arXiv:1710.09767, 2017.

[9] P. C. P. A. Kevin Frans, Jonathan Ho and J. Schulman. Learning a hierarchy. https://blog.openai.com/learning-a-hierarchy/, 2017.

[10] G. Konidaris. Constructing abstraction hierarchies using a skill-symbol loop. In IJCAI: proceedings of the conference, volume 2016, page 1648. NIH Public Access, 2016.

[11] G. Konidaris and A. G. Barto. Building portable options: Skill transfer in reinforcement learning. In IJCAI, volume 7, pages 895–900, 2007.

[12] G. Konidaris, S. Kuindersma, R. Grupen, and A. Barto. Robot learning from demonstration by constructing skill trees. The International Journal of Robotics Research, 31(3):360–375, 2012.

[13] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In Advances in Neural Information Processing Systems, pages 3675–3683, 2016.

[14] M. L. Littman, U. Topcu, J. Fu, C. Isbell, M. Wen, and J. MacGlashan. Environment-independent task specifications via gltl. arXiv preprint arXiv:1704.04341, 2017.

[15] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In International Conference on Machine Learning, pages 1928–1937, 2016.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. Nature, 518(7540):529–533, 2015.

[17] J. Oh, S. Singh, H. Lee, and P. Kohli. Zero-shot task generalization with multi-task deep reinforcement learning. arXiv preprint arXiv:1706.05064, 2017.

[18] A. Pnueli and Z. Manna. The temporal logic of reactive and concurrent systems. Springer, 16:12, 1992.

[19] K. Samejima, K. Doya, and M. Kawato. Inter-module credit assignment in modular reinforcement learning. Neural Networks, 16(7):985 – 994, 2003.

[20] R. S. Sutton and A. G. Barto. Introduction to reinforcement learning, volume 135. MIT Press Cambridge, 1998.

[21] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In Advances in neural information processing systems, pages 1057–1063, 2000.

[22] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. Artificial Intelligence, 112(1):181 – 211, 1999.

[23] E. Uchibe, M. Asada, and K. Hosoda. Behavior coordination for a mobile robot using modular reinforcement learning. In Intelligent Robots and Systems' 96, IROS 96, Proceedings of the 1996 IEEE/RSJ International Conference on, volume 3, pages 1329–1336. IEEE, 1996.

[24] H. van Seijen, M. Fatemi, J. Romoff, R. Laroche, T. Barnes, and J. Tsang. Hybrid reward architecture for reinforcement learning. arXiv preprint arXiv:1706.04208, 2017.

[25] A. S. Vezhnevets, S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. arXiv preprint arXiv:1703.01161, 2017.