Automatic Policy Decomposition through Abstract State Space Dynamic Specialization

Rene Sturgeon
Mathematics and Computer Science
Royal Military College of Canada
Kingston, Canada
rene.sturgeon@rmc.ca

Francois Rivest
Mathematics and Computer Science
Royal Military College of Canada
Kingston, Canada
francois.rivest@{rmc.ca, mail.mcgill.ca}

***Abstract*—Significant progress has been made recently in deep reinforcement learning in the development of options. This idea consists in learning policies (or macro of actions) for sub-goals. An important bottleneck of this approach is that these options are often available as actions everywhere in the state space, hence, potentially enlarging the action space to search for the optimal policy. In this paper, we propose to use the fact that the state space is rarely fully connected, but instead has regions of highly connected states with fewer links between those regions. Our proposed model extends deep Q-Learning network (DQN) by splitting the top layers into multiple heads each specializing in learning the dynamics of a particular region of the state space as well as the optimal policy for that region. The state prediction quality of each head is used to determine which head is the local expert, rating its contribution to the current state's policy. We show that this approach is able to learn something similar to options and generalized value function, providing a promising alternative to the current approach.**

***Index Terms*—reinforcement learning, deep learning, option, neural networks, model learning, video games**

I. Introduction

Since the success of deep Q-learning (DQN) on Atari video games [1], the deep reinforcement learning (DRL) approach produced key algorithms allowing end-to-end development of AI algorithms with little human expertise (e.g., [1]–[3]). In DRL, the agent attempts to learn an optimal policy to maximize the rewards it will receive achieving tasks or sub-goals. While DQN and similar deep network approaches allow such system to learn a better state representation (or abstraction), some recent work in DRL focused on temporally extended actions, or *options* [4]. These are like sub-policies or macro actions, that can be selected like regular actions, but which cover multiple time steps. Options have only recently been successfully integrated in DRL in a fully automated fashion with the option-critic (OC) architecture [5]. But since these approaches assume an option can be taken from anywhere in the state space, this potentially increases the number of valid actions, and hence, the overall size of the search space. For options to speed up learning, the system must learn and restrict when they should be available in the state space. How to do this remains an open question [5].

This project was supported in part by a CDARP grant to F.R.

In this paper, we build upon DRL, which can be seen as generating an abstract state space in the top layers, and we train the network to learn the environment dynamics similarly to [6], [7]. But instead of using it to regularize or to plan, we use it to divide the abstract state space into slightly overlapping regions thereby loosely partitioning the state space. Thus, at any given time step, the agent will be within only one or two regions of the state space. The top layers of the deep network are split into multiple *heads*, each head specializing for a particular region. A region is defined by the set of states for which the head is a better predictor of the abstract state space dynamic. Although each head has all the primitive actions, the overall policy is more about choosing the right head than the specific primitive action (similar to [5]). The main difference is that each head or sub-policy (resembling an option in a loose sense), is only used when the agent is in a state within its area of expertise. Outside such region, there is no need to learn what that sub-policy (or head) does. Instead of learning which policy or option to follow (as in [5]), our system uses the current level of abstract state prediction of each head to determine their contribution to the global policy, in a way similar to a mixture of experts.

We will first review related DRL work in Section II. Then we will briefly describe DQN and our basic extension for comparison purposes in Section III-A. Section III-B will describe our policy decomposition algorithm. Finally we will compare our new algorithm to our baseline on ATARI games in the Arcade Learning Environment (ALE) [8]. Section IV will show that some of the network heads are specializing for different game situations and thus specialized in a way similar to options, but available only within a limited set of states.

II. Related Work

The work on temporal abstractions (or options) in RL, is now more than 20 years old [4]. It is only recently that it has been fully integrated in its original form in DRL in the OC architecture [5]. The objective in RL is to learn a policy that will maximize the discounted sum of future rewards. An option is defined as a macro-action or sub-policy, which consists of an initialization set (set of states from which it can be selected), the option policy itself, and a termination function (indicating when the option is over, returning the control to

978-1-7281-6926-2/20/$31.00 ©2020 Crown

the main policy). The optimal policy can then be thought of as a combination of primitive actions and options. In [5], the network must learn an optimal policy made solely of options, as well as each option's policy and its exit function. There are three main limitations in this work: 1) It tends to learn very short options (a problem that is solved by penalizing switching between options in [9]). 2) Each option is available at every state, requiring exploration of each option over the whole state space in order to learn each option's optimal policy and exit function. 3) In their setup, the options represent an over-representation of the problem, as a single option would be sufficient to learn the whole task. In contrast, in this paper, we focus on loosely partitioning the state space and the top part of the network such that each head (or option) specializes for different regions of the state space. We are not learning which head to use or when to exit it directly; instead, we are using each head prediction performance to weight their contributions to current policy.

Other work in hierarchical DRL also looked at how to create temporal abstractions. For example, [10] has two levels of controllers (similarly to OC [5]). The top (or meta) controller receives the external rewards and learns a policy in which the actions can be seen as selecting sub-goals or desired states. For each sub-goal, there is a sub-controller intrinsically rewarded when reaching the sub-goal state. Another approach is the hybrid reward function [11]. Like in our approach, they break the top layer of the network into distinct components of the value function, each associated to different types of rewards (or goals). In both of these approaches, goal states or reward types must be manually determined. How to do this automatically remains an open question. In contrast, we propose an end-to-end approach that specializes its heads toward state space regions of similar dynamics rather than per goal states or reward types.

The idea that learning the environment dynamics could help learning the value function has also been studied by [6] and was useful in learning the optimal policy. In [7], they learn the abstract state space dynamics to allow the system to plan in abstract space. Here, we will divide the network abstract space representation and value function into multiple heads, where each head will learn its own abstract representation dynamic. The heads' ability to predict their own abstract state space dynamics will be use to automatically determine their region of expertise and to weight their final contribution to the value function.

The only work we are aware that focuses strictly on reducing options' initialization set is the work of [12]. In this work, they focus on separating the state space using adaptive hyper-planes and associating specialized skills (or policies) to each so-define sub-regions of the state-space. In this paper, we are not restricting the regions of expertise of each head to be defined by sets of hyper-planes. Instead, we are separating region of the abstract state space by their internal dynamics. Those sharing the same dynamics gets associated to the same head. When a head is good at predicting its own abstract space dynamic, it means it is in its region of expertise and should play a role locally in the policy. Similarly. when it cannot predict its own dynamic, then it should not contribute to the output value function of the network.

III. Model

The model is based on a variation of DQN [1], and double-DQN [13] which we adapted for smaller memory. However, the proposed method is not restricted to DQN DRL architecture. The first subsection describes the baseline regularized to learn the abstract state space dynamic without specialized heads, followed by the description of the full model.

A. Baseline

In RL, the objective is to find a policy $\pi : S \rightarrow A$ mapping each state $s \in S$ to an action $a \in A$ that maximizes the sum of discounted future rewards $r \in \mathbb{R}$. DQN uses Q-Learning which works by learning an approximation of $Q : S \times A \rightarrow \mathbb{R}$ of the sum of future rewards from state s given action a. The final policy consists of always selecting the action a with the largest Q-value, given state s.

Similarly to [1], our Q-Value network receives the state as input and has one output per primitive action. For Atari games, it has three convolution layers followed by two fully connected layers, each followed by a rectified linear activation unit (relu), with the exception of the output layer, as follows:

- Convolution layer 1: 32 8x8 filters with stride of 4
- Convolution layer 2: 64 4x4 filters with stride of 2
- Convolution layer 3: 64 3x3 filters with stride of 1
- Fully connected hidden layer 1: 512 units
- Fully connected output layer 2: Number of actions

To remain close to DQN architecture we did not use dropouts, pooling, or batch normalization. To maintain good decorrelation of samples with a smaller replay memory, and inspired by the A3C algorithm [14], we used four worker threads to play games and generate training samples and one thread to train the network. We also implement double DQN [13], which consists in using a stable target network in the training process, and which was shown to avoid overestimation of Q-values.

There are therefore multiple copies of the network at different training stages. Let Q_{var} be the network under training, Q_{work} be the one generating samples $(s_t, a_t, r_{t+1}, s_{t+1})$ for the replay memory, and Q_{tar} be the target network used to generate target values for training.

The network weights are randomly initialized using Xavier initialisation [15] and then copied to the other networks. Then the worker network is used to generate samples which are saved to replay memory while playing games using an ϵ-Greedy policy on their Q-Values:

$$a_t = \begin{cases} \underset{a \in A}{\arg\max}\{Q_{work}(s_t, a)\} & \text{with prob } 1 - \epsilon \\ \text{random action } a \in A & \text{with prob } \epsilon \end{cases} \quad (1)$$

Once enough samples are in the replay memory, the training of the network begins using mini-batches of 32 samples from the replay memory per update, using loss (7) and Adam

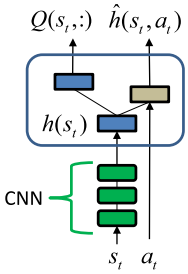

Fig. 1. Baseline Network. As in DQN, the network receives the input state s_t (an image), and produces a Q-value $Q(s_t, a)$ for each possible action a. Once an action is selected, that action a_t, combined with the top hidden layer activity $h(s_t)$, generates a prediction $\hat{h}(s_t, a_t)$ of the hidden layer activity (abstract state) $h(s_{t+1})$ at the next time step.

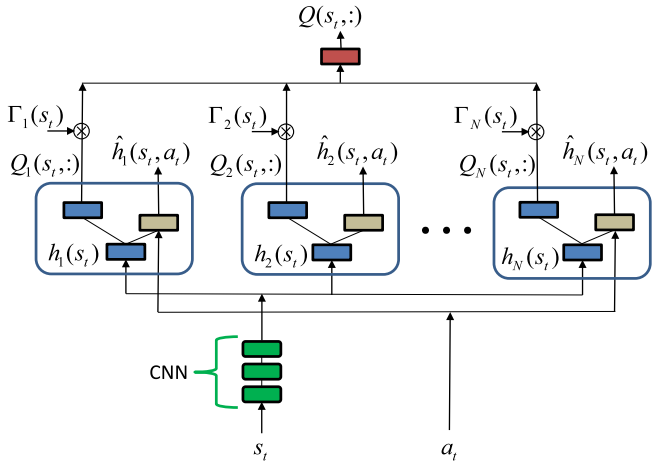

Fig. 2. Abstract State Space Dynamic Specialization Network. The hidden layer from Fig. 1 is split into N distinct *heads*, each feeding its own Q-values as well as a prediction of its own next state. The Q-values are then weighted by each head's current performance Γ at predicting its own next state ($\hat{h}$). These Q-values are then added together to form the global policy. Note that the number of weights between the top hidden layer h and the output Q-values is the same as in the baseline network (or DQN).

optimizer [16]. Its weights are copied back to the the worker threads every 100 updates, and to the target network every 2,500 updates. The complete list of hyper-parameters is given in Table I.

As in previous work, gradient and rewards are clipped. Given a sample (s, a, r, s'), the constant target value is

$$target = \begin{cases} \text{sign}(r) \text{ if end of episode, otherwise} \\ \text{sign}(r) + \gamma Q_{tar}(s', \arg\max_{a' \in A}\{Q_{var}(s', a')\}) \end{cases} \quad (2)$$

where γ is the discount factor and the loss on Q is

$$L_Q = \text{huber}(Q_{var}(s, a) - target), \quad (3)$$

and where the Huber loss is used to clipped the gradient:

$$\text{huber}(error) = \begin{cases} 0.5 \times error^2 & \text{if } |error| \leq 1 \\ |error| - 0.5 & \text{otherwise.} \end{cases} \quad (4)$$

In order to make it comparable to our proposed architecture, and similarly to [6] and [7], we added an extra output layer to the network, parallel to the Q-value output layer, and of the same size as the top hidden layer (see Fig. 1). This layer also receives the selected action and generates a prediction $\hat{h}(s_t)$ of the hidden layer activity $h(s_{t+1})$ at the next time step.

Given a sample (s, a, r, s'), the loss function on this additional output layer is simply the mean Huber loss of each component

$$L_P = \frac{1}{n} \sum_{j=1}^{n} \text{huber}(\hat{h}_{var}(s, a)_j - h_{tar}(s')_j), \quad (5)$$

where n is the number of units of the hidden layer. The Huber loss is used instead of the squared error to clip the gradient.

To ensure that the magnitude of the predictive error remains within a reasonable scale and that the hidden state representation does not collapse, we further constrain the hidden layer to an hypersphere of radius 1 by adding the regularization term

$$L_R = \text{huber}(\|h_{var}(s)\|_2 - 1). \quad (6)$$

The final network loss is therefore given by

$$L_{QPR} = L_Q + \zeta_1 L_P + \zeta_2 L_R, \quad (7)$$

where $\zeta_1 = 0.5$ and $\zeta_2 = 0.25$ are the weight factors given to the predictive loss L_P and the regularization loss L_R respectively.

B. Abstract State Space Dynamic Specialization

In this paper, we use the fact that in practice, Markov environments are often not fully connected. Instead, they may have more connected regions with specific dynamics, with some connections linking one highly connected region to another. We further assume that in some situations, different regions may share similar dynamics along some dimensions, therefore, potentially sharing similar good policies.

First, the top hidden layer h and its two output layers Q and $\hat{h}$ are split into N equal-size *heads* as showed in Fig. 2. For example, in Atari, the DQN top hidden layer h has $n = 512$ units. If we create $N = 16$ heads, each will have a hidden layer h and a hidden state prediction output layer $\hat{h}$ of size $512/16 = 32$. Although each head has now its own policy Q, since these will be added together to form the global policy, the number of weights between h and Q remains the same. On the other hand, there are fewer weights between h and $\hat{h}$, i.e. n^2/N instead of n^2. This is an important difference with OC which has more weights than its non-option equivalent.

The important factor is to determine which head best understands the region of the state space. The head that is better at predicting its next state in a given region of the environment is more likely to be able to learn a good policy for it. Therefore, the head sub-policy should be weighted more, locally, in the overall policy. To do this, we devised a measure of the quality of each head's predictions of the hidden state transitions over the last few time steps.

Let $g_i(s_t)$ be the moving average of the prediction error of head i at time t such that

$$g_i(s_t) = (1-\omega)g_i(s_{t-1}) + \omega e^{-\beta\|\hat{h}_i(s_{t-1}) - h_i(s_t)\|_2^2/\bar{h}(\tau)}, \quad (8)$$

where β plays the role of a temperature parameter allowing to control the level of separation between the heads based on their error, and ω determines the temporal window of interest, and $\bar{h}(\tau)$ is a global moving average of workers' heads prediction error. Smaller values of ω will encourage specializing over longer temporal windows, but will reduce the ability to react quickly to changes in the environment. The terms are then normalized to give the *specialization factors*

$$\Gamma_i(s_t) = \frac{g_i(s_t)}{\sum_{j=1}^{N} g_j(s_t)}. \quad (9)$$

Note that the $\sum_{i=1}^{N}\Gamma_i(s_t) = 1$ and that $0 < \Gamma_i(s_t) < 1\ \forall i$.

For the trainer to compute $\Gamma_i(s')$, the replay memory must be extended to include $g(s_t) = (g_1(s), \ldots, g_N(s_t))$. It will also save the expert i_t^* with the highest weighted Q-value

$$i_t^* = \underset{i\in\{1,\ldots,N\}}{\arg\max}\left\{\max_{a\in A}\{\Gamma_i(s_t)Q_{work,i}(s,a)\}\right\}. \quad (10)$$

Thus, the worker now saves in replay memory, the tuples $(s_t, g(s_t), i_t^*, a_t, r_{t+1}, s_{t+1})$.

Given replay memory samples (s, g, i^*, a, r, s'), the trainer can then compute $\Gamma_i(s')$ using (8) and (9) for each head.

The loss on Q is computed individually for each head as

$$L_{Q_i} = \text{huber}(Q_{var,i}(s,a) - target_i), \quad (11)$$

where the target Q-value for each head is defined by

$$target_i = \begin{cases} \text{sign}(r) \text{ if end of episode, otherwise} \\ \text{sign}(r) + \gamma Q_{tar,i}(s', a') \text{ if } i \neq i^* \\ \text{sign}(r) + \gamma \max\limits_{j\in\{1,\ldots,N\}}\{Q_{tar,j}(s', a')\} \text{ if } i = i^* \end{cases} \quad (12)$$

and where $a' = \arg\max_{a''\in A}\{\sum Q_{var,i}(s', a'')\}$.

Each head also has its own prediction loss

$$L_{P_i} = \frac{1}{m}\sum_{j=1}^{m}\text{huber}(\hat{h}_{var,i}(s,a)_j - h_{tar,i}(s')_j) \quad (13)$$

where $m = n/N$.

Finally, in (6) we regularized $h(s)$ on a hypersphere of radius 1. In order to keep the same hyper-parameters across the two models, we need to maintain the loss scales. But hyper-spheres of fewer dimensions have different component scales when keeping the radius constant. Therefore, we need to find the radius y that will keep the component scale and such that

$$\sum_{i=1}^{N}(\|h_i\|_2 - y)^2 \approx (\|h\|_2 - 1)^2. \quad (14)$$

Let Δh be the expected value of one component of the vector in both models. Expanding (14) we get

$$\begin{gathered} \sum_N\left(\sqrt{\sum_m \Delta h^2} - y\right)^2 \approx \left(\sqrt{\sum_n \Delta h^2} - 1\right)^2 \\ \sum_N\left(\sum_m \Delta h^2 - 2y\sqrt{\sum_m \Delta h^2} + y^2\right) \approx \sum_n \Delta h^2 - 2\sqrt{\sum_n \Delta h^2} + 1. \end{gathered} \quad (15)$$

Assuming that all Δh are approximately equal then

$$\begin{gathered} \sum_N\left(m\Delta h^2 - 2y\sqrt{m\Delta h^2} + y^2\right) \approx n\Delta h^2 - 2\sqrt{n\Delta h^2} + 1 \\ Nm\Delta h^2 - 2Ny\sqrt{m\Delta h^2} + Ny^2 \approx n\Delta h^2 - 2\sqrt{n\Delta h^2} + 1 \\ -2Ny\sqrt{m\Delta h^2} + Ny^2 \approx -2\sqrt{n\Delta h^2} + 1 \\ -2\sqrt{N}y\sqrt{Nm\Delta h^2} + \left(\sqrt{N}y\right)^2 \approx -2\sqrt{n\Delta h^2} + 1 \end{gathered} \quad (16)$$

which gives $y = 1/\sqrt{N}$. Therefore the new regularization loss for each head becomes

$$L_{R_i} = N\,\text{huber}(\|h_i(s)\|_2 - 1/\sqrt{N}). \quad (17)$$

With the three losses, L_{Q_i}, L_{P_i}, and L_{R_i} for each head i, and having the specialization factor $\Gamma_i(s)$ for each head, we can now calculate the total loss L_{QPR}

$$L_{QPR} = \sum_{i=1}^{N}\Gamma_i(s)[L_{Q_i} + \zeta_1 L_{P_i} + \zeta_2 L_{R_i}], \quad (18)$$

where ζ_1 and ζ_2 can remain as in the baseline model. Note that $\Gamma_i(s)$ can be seen as a weighted learning rate for each head, being higher for the head that has little error on its hidden state prediction, and lower for the irrelevant heads.

IV. Experiments and Results

A. Methods

To test our algorithm, we use the Arcade Learning Environment (ALE) [8] as in [1], that simulates numerous Atari 2600 video games. The variety of games present different problems to a general agent such as sparse rewards requiring a large amount of exploration or a wide variance in the rewards within the game. Its design makes it simple to interface with algorithms. The video output is 160x210 2-d array of pixels (128 color pallet) which is read one frame at a time and reduce to 84x84 black and white images. The input to the network is made of 4 consecutive frames, and each action is repeated four times. The output also includes a reward signal, that correspond to the score in the game, an end game signal, and a loss of life signal. The input to a game has a maximum of 18 actions (game dependent) and a reset game command. On current hardware, the games can be simulated many times faster than real time allowing for quicker training.

Our model was written in Python and Tensorflow, a CPU/GPU neural networks API. Most of our simulations were done on a 20-core Xeon processor with 16GB of memory and a Nvidia GeForce 1080Ti graphics card. Some of our nodes had exactly twice that capacity, running two simulations concurrently.

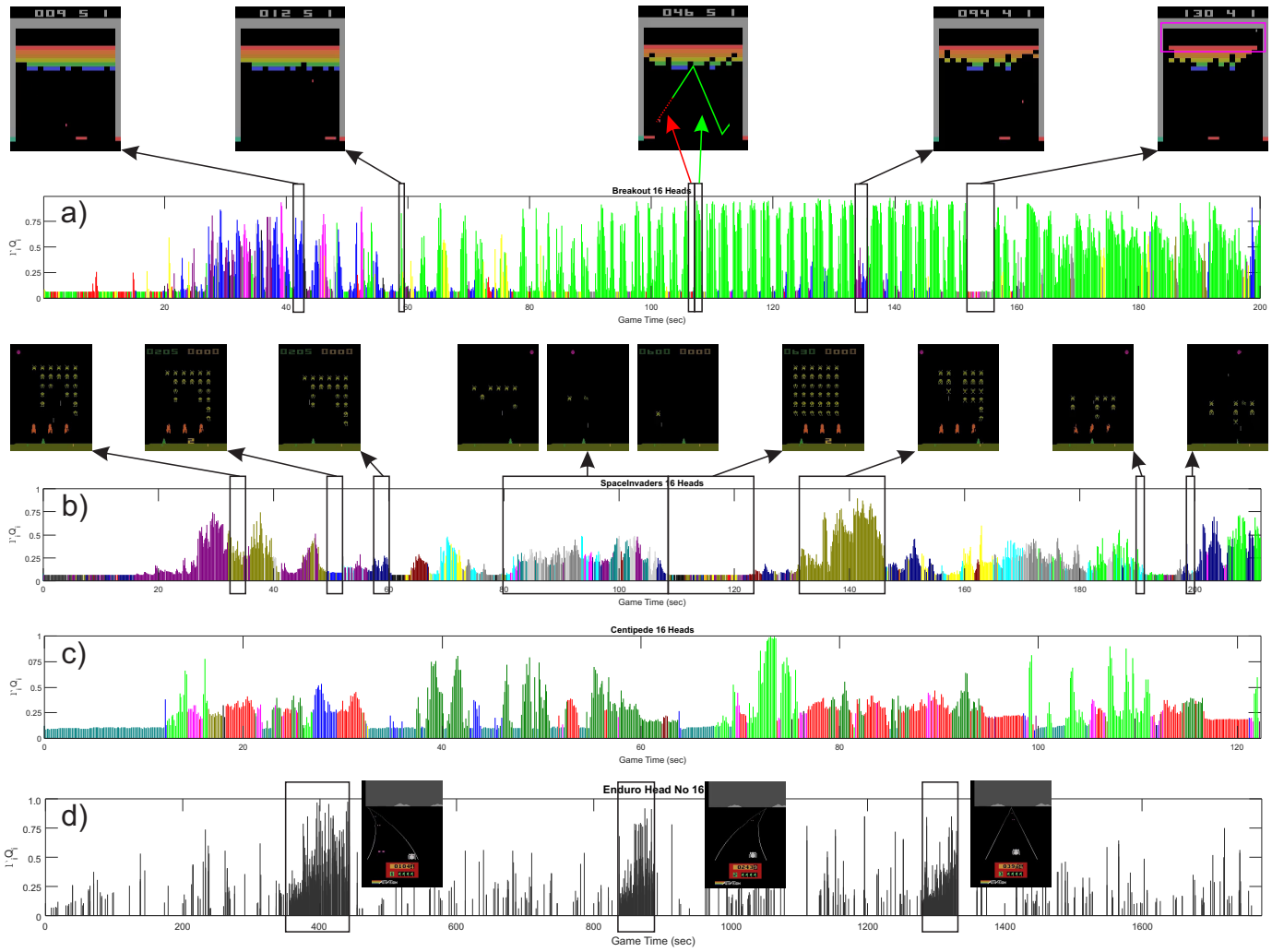

Fig. 3. Plots of *leading head*'s scores $S_i(s_t, a_t) = \Gamma_i(s_t)Q_i(s_t, a_t)$ colored by leading head during four different ATARI games. The leading head is defined as the head with highest score selecting action a_t at time step t. A low value on this graph usually means that all heads are almost equally contributing to the selected action. Black rectangles surround areas of interest from which screenshots were taken and showed above. See text for details. a) Breakout: Third screenshot, we added approximate colored path representing leading edges (in red and green). Last screenshot, we surrounded in purple the top area of interest. b) SpaceInvaders: There are 3 screenshots related to the fourth rectangles. c) Centipedes: Screenshots elements were to small to be visible. d) Enduro: Only head #16 is displayed. Screenshots are right after the corresponding rectangles.

B. Results

Table II shows the performance of DQN, our baseline, our multi-head model, and OC. The number of heads (h) or options (o) used is given in parenthesis. The number of training steps is 10M across all papers. This is the same number of steps we used in Breakout, Centipede, Enduro, and SpaceInvaders. To see if there was any significant difference in longer training we used 14M steps for Asterix, MsPacman, Seaquest, and Zaxxon. But there was no significant changes. The objective here is not to beat the state of the art in terms of asymptotic performance, but simply to show that our algorithm is at least reaching similar performances. The important contribution is the ability to have various parts of the network specializing themselves in different aspects of the games.

In order to analyze if the specialization was successful, we extracted the scores $S_i(s_t, a_t) = \Gamma_i(s_t)Q_i(s_t, a_t)$ and extracted the maximal head and its score for each time step. The head with the highest score on the selected action is considered to be the *leading head* for that time step. Fig. 3 shows the leading head score and color plot from four different games. A low value on this graph usually means that all heads are almost equally contributing to the action selection.

In **Breakout** (Fig. 3, plot a), we can see a change in leading heads around the second screenshot. This is where the ball's speed increases after the first 12 hits. Then, there is a long period of oscillation with one head leading repetitively (in green). As showed in the third screenshot, right after hitting the ball, there is not much to do or to predict (red arrow), but quickly after the leading head takes over until it hits the ball again (green arrow). The fourth screenshot occurs right after a loss of life and the ball speed returns to its slow speed. The ball quickly returns to its fast speed because this occurs when

TABLE I
LIST OF HYPERPARAMETERS

Hyperparameter	Value	Description
γ	0.99	Discount factor for future rewards
α	0.000625	Overall learning rate in the optimizer
Mini-batch size	32	Size of the mini-batches
Initial ϵ	1.00	Starting value of the ϵ-Greedy exploration
Final ϵ	0.10	Final value of the ϵ-Greedy exploration
Decay steps	1×10^6	Number of steps to decay ϵ to its final value
Evaluator ϵ	0.05	Fixed exploration value used by the evaluator for all evaluations
No op steps	30	Maximum number of no-operation steps that agent will do at the beginning of game
Replay memory size	256,000	Number of samples that can be stored in memory
Initial replay size	32,000	Number of samples to be stored before the trainer begins its updates.
Frame dimensions	84x84	Size of the frame dimensions input to the network
Evaluator net update	50,000	Number of updates before the trainer transfers current network parameters to the evaluator network
Worker net update	100	Number of updates before the trainer transfers current network parameters to the worker networks
Target net update	2,500	Number of updates before the trainer transfers current network parameters to the target network
ζ_1	0.50	Weight factor for predictive loss
ζ_2	0.25	Weight factor for regularization loss
ω	0.9 or 0.5	Moving average weight factor in $g_i()$
β	0-30	Temperature parameter in $g_i()$, annealed up to final value over 3M updates (few had $\beta = 1$ constant)

TABLE II
SCORE COMPARISON SUMMARY

Game	DQN[a]	Baseline	Multi-head	OC[b]
Asterix	6,012	n/a	7,900 (8h)	8,000 (8o)
Breakout	401	350	385 (16h)	n/a
Centipede	8,309	1,900	3,140 (16h)	n/a
Enduro	1,002	800	541 (16h)	n/a
MsPacman	2,311	2,800	2,500 (8h)	2,100 (8o)
Seaquest	5,286	n/a	6,000 (8h)	8,000 (8o)
SpaceInvaders	1,976	2,200	1,100 (16h)	n/a-
Zaxxon	4,977	n/a	8,000 (2h)	6,100 (8o)

[a]DQN results are reported from [1].
[b]OC results are estimated from graphs in [5].

the ball hits any brick in the top three rows. Immediately we see the agent return to the previously leading head (in green). The last screenshot shows that when the ball gets behind the seen (purple rectangle), no head is leading. During this period, the controller has no impact on the score and all actions have the same value.

In **SpaceInvaders** (Fig. 3, plot b), we can see by the coloring that different heads are leading at different stage of the game. The beginning of the plot and box 5 corresponds to the beginning of a new level. Both regions are followed by a region with the same leading head (boxes 1 and 6). Box 2 is when the player dies. Boxes 3 and 8 corresponds to the moment the red walls disappears. Box 4 (with three screenshots) shows a section where the game dynamic is faster.

In **Centipede** (Fig. 3, c), again, we can see by the coloring that different heads are leading at different stage of the game.

In **Enduro** (Fig. 3, plot d), we are showing that one head has specialized in the end of the one day race period when four green flags are displayed and the player can log more miles as the obstacles are not as difficult to avoid. In the game shown the agent has successfully completed three days of racing and did not finish the fourth.

The percentage of *lead time* for each head in a game is showed in Table III. In each game, different heads can be leading at different points in the game. But not all the heads may end-up leading. For example, in Breakout, 8 heads are leading between 5% to 25% of the time each, while 4 heads are leading for less than 1% of the time each. Similarly, in Centipedes, 6 heads are leading most of the time, while 7 others basically never lead. In contrast, in Enduro, all heads seems to be equally leading. Note that as opposed to options, multiple heads can be equally contributing at some points to the action selection process. For example, if one head is specialized in predicting obstacles dynamic, while another is specialized at predicting left curve dynamic, then the two could be very well *sharing the lead* when there is an obstacle in a left curve. It is not clear how to analyze those situations, but by definition, is not something options can do. You are either following one option, or the other, but you cannot follow two options at the same time. In contrast, our multi-head approach allows mixture of sub-policies based on head's local specialization.

Finally, Table IV and Table V report the averaged leading duration and maximal leading duration for each head in a game. This is roughly similar to option-length in number of time steps. A big problem in the original OC architecture was that options tended to be short (one time step) [5], a problem they later resolved by penalizing option switching [9]. In contrast, our algorithm does not learn which head to choose or when to switch. Instead, it weights each head using the specialization factor that continuously evaluates how each head understands or predicts the current (local) dynamics. While the average *leading duration* may seem short in Table IV, it must be understood that there are regions where the heads do not specialize much, as at the beginning of the levels, or when the controller is useless (for example, when the ball is stuck behind in Breakout, Fig. 3, plot a, last screenshot). These regions significantly lower the average. Table V the longest *leading duration* in time steps, showing many options leading sometimes for 10 to 30 time steps.

TABLE III
PERCENTAGE OF TIME EACH HEAD IS THE LEADING HEAD IN ONE GAME

Game	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10	H11	H12	H13	H14	H15	H16
Breakout	2.2	15.5	24.8	5.1	6.0	2.6	11.2	1.2	6.4	2.6	7.2	0.0	0.3	13.9	0.8	0.1
Centipede	3.3	9.3	9.9	0.0	21.4	14.3	0.5	0.0	0.0	2.7	2.2	0.5	17.0	18.7	0.0	0.0
Enduro	4.8	7.0	7.7	3.4	9.9	5.8	5.1	5.8	5.8	5.0	6.2	4.9	8.2	7.5	6.0	6.9
SpaceInvaders	3.3	1.7	6.7	14.0	1.2	2.1	8.6	7.6	14.3	7.4	11.6	2.9	3.1	0.5	11.9	3.3

TABLE IV
AVERAGE NUMBER OF CONSECUTIVE ACTIONS SELECTED FROM LEADING HEAD

Game	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10	H11	H12	H13	H14	H15	H16
Breakout	1.4	3.6	6.7	2.1	2.3	2.4	1.9	1.7	1.7	2.9	2.4	0.0	1.0	2.2	2.3	3.0
Centipede	1.2	2.8	7.4	0.0	6.0	2.1	1.0	0.0	0.0	3.2	1.0	5.0	6.9	5.9	0.0	0.0
Enduro	1.5	1.5	1.5	1.6	1.7	1.6	1.5	1.5	1.6	1.5	1.7	1.4	1.6	1.6	1.6	1.9
SpaceInvaders	3.2	3.1	3.3	2.4	2.0	2.3	2.8	2.9	3.2	7.7	5.4	4.7	2.7	1.0	4.4	3.6

TABLE V
MAXIMUM CONSECUTIVE ACTIONS SELECTED FROM LEADING HEAD

Game	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10	H11	H12	H13	H14	H15	H16
Breakout	4	13	32	9	13	6	5	3	6	8	9	0	1	8	5	3
Centipede	2	18	22	0	34	8	1	0	0	11	1	5	91	32	0	0
Enduro	7	8	7	6	11	9	6	7	7	7	8	6	8	8	8	30
SpaceInvaders	15	8	17	11	4	5	17	11	24	117	123	21	7	1	29	30

V. CONCLUSION

In this paper, we developed a new DRL technique to allow a deep network to automatically decompose its policy based on the environment dynamic components. When compared to options, the proposed algorithm solves the initialization set problem and the termination function problem. By automatically evaluating each head's level of expertise for the local environment dynamics, each head (or sub-policy) may significantly contribute to the action selection process only for states where it is relevant. Unlike options, multiple heads can contribute together at anytime in the action selection process. Moreover, two sub-policies don't have to begin and terminate in a synchronous manner either. Finally, compared to sub-goals and generalized value functions algorithms, the proposed algorithm learns fully end-to-end without requiring human assigned sub-goals.

ACKNOWLEDGMENT

A significant portion of the results in this paper comes from R.S. PhD thesis [17].

REFERENCES

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, and Sa, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] H. A. Pierson and M. S. Gashler, "Deep learning in robotics: a review of recent research," *Advanced Robotics*, vol. 31, no. 16, pp. 821–835, 2017.

[3] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving," *Electronic Imaging*, vol. 2017, no. 19, pp. 70–76, 2017.

[4] R. Sutton, D. Precup, and S. Singh, "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, vol. 112, no. 1–2, pp. 181–211, 1999.

[5] P.-L. Bacon, J. Harb, and D. Precup, "The option-critic architecture," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[6] T. D. Bruin, J. Kober, K. Tuyls, and R. Babuška, "Integrating state representation learning into deep reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 1394–1401, 2018.

[7] V. François-Lavet, Y. Bengio, D. Precup, and J. Pineau, "Combined reinforcement learning via abstract representations," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 3582–3589.

[8] M. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[9] J. Harb, P.-L. Bacon, M. Klissarov, and D. Precup, "When waiting is not an option: Learning options with a deliberation cost," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[10] T. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum, *Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation*. Curran Associates, Inc., 2016, pp. 3675–3683.

[11] H. Van Seijen, M. Fatemi, J. Romoff, R. Laroche, T. Barnes, and J. Tsang, "Hybrid reward architecture for reinforcement learning," in *Advances in NIPS*, 2017, pp. 5392–5402.

[12] D. Mankowitz, T. Mann, and S. Mannor, *Adaptive Skills Adaptive Partitions (ASAP)*. Curran Associates, Inc., 2016, pp. 1588–1596.

[13] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *Proceedings of The 33rd International Conference on Machine Learning*, M. Balcan and K. Weinberger, Eds. PMLR, 2016, pp. 1995–2003.

[14] V. Mnih, A. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of The 33rd International Conference on Machine Learning*, M. Balcan and K. Weinberger, Eds. PMLR, 2016, pp. 1928–1937.

[15] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[16] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 12 2014.

[17] R. Sturgeon, "Automatic option discovery within non-stationary environments," PhD, Royal Military College of Canada, 2018.