Reinforcement Learning of Multiple Tasks Using Parametric Bias

Leszek Rybicki, Yuuya Sugita and Jun Tani

Abstract—We propose a reinforcement learning system designed to learn multiple different continuous state-action-space tasks. The system has been tested on a family of space-searching task akin to Morris water maze, but with obstacles. While exploring a task, the agent builds its internal model of the environment and approximates a state value function. For learning multiple tasks, we use a parametric bias switching mechanism in which the value of the parametric bias layer identifies the task for the agent. Each task has a specific parametric bias vector, and during training the vectors self-organize to reflect the structure of relationships between tasks in the task set. This mapping of the task set to parametric bias space can later be used to generate novel behaviors of the agent.

I. INTRODUCTION

This paper describes the results of an experiment involving learning multiple continuous state space delayed reinforcement problems akin to the Morris water maze with obstacles. The agent was expected to find the target, while avoiding an obstacle. The position of the target and obstacle varied from task to task, but neither was explicitly given to the agent, requiring it to develop a strategy by methods of trial-and-error. The agent is aware which of the tasks it is supposed to attempt.

While the agent explores the environment separately for the tasks, we propose that it use a single learning model for value function approximation, state prediction and policy generation. In effect, the obtained information about target and obstacle position can easily be re-used for many tasks, thus requiring less exploration than would be required in case of multiple model reinforcement learning regulated by gating [10]. This can further be improved by building an internal model of the parametrized set of tasks, reflecting their similarities and using that model for regulating the transfer of skills: tasks regarded as similar in the internal model can be solved using similar policies. Since the model reflects the structure in many degrees, information can be re-used selectively - obstacle avoidance skill should be shared among tasks with the same obstacle placement while value function approximation should be shared between tasks with the same target position. Certain skills, like state change prediction where obstacles are not present, can be shared among all tasks. We find it important to stress that by multiple tasks we don’t mean hierarchical reinforcement learning, but a family of tasks not necessarily organized in a vertical hierarchy.

We propose using the parametric bias method for task switching, self-organizing the internal model of the task set and generalizing experience and behavior among tasks.

Introduced in [1], a parametric bias is a method for training a single network with multiple data sets. In principle, parametric bias is a set of input units, the values of which identify a data set. To train a network on multiple data sets, each data set is assigned a parametric bias vector, random at first, and the network is trained to reproduce the data set, given its pb-vector. The parametric bias vectors are updated during training of the network by means of error back-propagation. While training a data set only its corresponding pb-vector is updated. Details of training the parametric bias can be found in section III-B.

In our method, we use three feed-forward neural networks for approximating the state value function, generating actions, based on the current state and predicting the results of actions for all the tasks. The three networks share a common parametric bias layer. Cached exploration data is task-dependent, as well as a discretized approximation of the value function, which is iteratively updated in a TD(0)-like scheme and used to train the value approximating network. A detailed description of the method, its components and training is presented in section III. After training, setting the values of the parametric bias units to a vector corresponding to a task caused the value function network to approximate the value function of that task, the world model network predicted the positions of obstacles in the environment of the task and the policy generator produced a policy for the task in question.

II. EXPERIMENT DESCRIPTION

In the experiments, the agent was placed in a simulated square-shaped environment. The environment contained a goal zone, where the reinforcement was non-zero and a variable set of obstacles: walls, which the agent could not penetrate. The agent was the only moving object in the environment and the reinforcement signal was dependent only on the state, not the agent’s action. In no case the position of the goal was given to the agent, neither absolute,
nor relative to its own position, making the setup similar to the Morris water maze, in which an animal must use its spatial orientation and long-term memory to find a hidden (transparent) island in a pool of water.

The state of the environment used in the experiment was the agent’s two coordinate position in space between $-1$ and $1$. This encoding of state was chosen as simple enough to test the proposed method, while keeping the environment Markovian and minimizing dimensionality of the state space for elements of the model that use lookup tables. In fact, any continuous state space should suffice, but the $x$ and $y$ position of the agent significantly facilitated visualization and analysis.

The agent’s action set is the set of velocity vectors with a limit value of $0.1$ - meaning that the agent could move at most $5\%$ of the length of the side of the square.

State change depended on the agent’s desired motion vector and vicinity of obstacles. Denoting the transformation for limiting speed as $g(u(t))$, the actual state change was:

$$x(t + 1) = \begin{cases} x(t) + g(u(x)) & \text{if } x(t) + g(u(x)) \text{ is valid} \\ x(t) & \text{otherwise} \end{cases}$$

(1)

Further on, we make references to “policy vector field”, by which we mean the vector field of actual state change (see fig. 2). If the policy vector field is the gradient of the value function, the policy can be referred to as greedy in regard to the value function.

Vector fields shown in fig. 2 and subsequent figures present actual state change, not action vectors.

Tasks presented to the agent differed in two aspects: by the placement of the target, resulting in a different distribution of reinforcement signal in the state space, or by the placement of obstacles, changing the environment properties. Both modifications were employed in the task set, see fig. 1. The goal of the agent was to reach the target in the shortest path from a random starting point. In some cases the shortest path involved avoiding obstacles.
Fig. 3. The structure of the learning system, incorporating an approximator of the state value function, a predictor of the next state (world model), a policy generator and a parametric bias layer to identify multiple tasks. A repository of previously visited states is used for training the value function estimator and the world model, which in turn provide data for training the policy generator.

Fig. 4. Average value of sensor input from the distance sensors overlaid on the experiment state space: the higher the average, the smaller the perceived distance to the obstacle. Note that edges of the state space are also obstacles.

III. MODEL DESCRIPTION

The learning model consists of a policy generator, an internal world model, a value function approximator, and a parametric bias module identifying the tasks. A discretized repository of training data, separate for every task, contains observed reinforcement in previously visited states and results of actions taken. The value function approximator and world model, together with a repository of previously visited states are used only to generate data for the policy generator. After training, the policy network alone can be used to control the agent. Figure 3 illustrates the relationship between the parts of the system. The rectangular elements represent the neural networks: the policy generator, the world model (for predicting state change), the value function approximating network and the parametric bias layer shared by the neural networks. We used three-layer feed-forward neural networks \( \text{tanh} \) transfer function. We considered networks with recurrent connections and training of whole sequences, but decided against them for reasons that will become apparent in the next sections.

The repositories storing previous experience of the agent are shown as the diamond shape.

Elements independent of the training system - the environment and the simulated distance sensors are shown as circles. The agent’s action vector is used for interacting with the environment, resulting in state change, sensory data and the reinforcement signal, which are in turn sent to the agent.

The policy generator network takes the current state (position in space) as input, together with sensory data and the task’s parametric bias and produces an action vector.

The world model network takes the current state, an action vector and the parametric bias and predicts the state resulting from taking the action in the given state \( \tilde{x}_{t+1} = f(x_t, u) \).

It’s role is most important in the parts of the environment, where the resulting state is least predictable - near obstacles. In order to provide a prediction of the next state, it has to take the placement of obstacles into account. The placement of obstacles is discovered by this element from sensory data and state change prediction.

The value function approximator takes a state vector as input and evaluates it, according to the current policy. We denote the approximation of the value function as \( \tilde{V}(x) \).

While all elements of the model are used during training, a trained agent can use just the policy generator network for interacting with the environment.

A. Sensory data

Sensory data consists of the proximity to the nearest obstacle in four directions: 4 units in total. The proximity sensor has a range of 0.5 (1/4 of the length of the space). Averaged readings from all the proximity sensors can be seen in figure 4. Sensory data was not incorporated into state space itself to keep the dimensionality of the state space to a minimum, but was added as extra input for all networks and the world model network was trained to predict not only position change, but to predict sensory input in the future position as well, which was used as input for the value function approximation network.

Given no sensory data, the system was able to learn and correctly map tasks with no obstacles, but failed to recognize the placement of obstacles, which was encoded only as the different rule of state change. Without sensory input, the state prediction error caused the discretized value function in the training data to propagate through obstacles, which caused any policy that followed its gradient to ignore obstacles completely.
The addition of sensory information as input for the policy network and the state prediction network (the two elements of the system that were supposed to react to obstacles correctly) decreased the local error of state prediction, but only after the state prediction network was also trained to predict the sensory data and the value approximator was given this prediction as additional input did the system correctly respond to obstacles.

B. Parametric bias

The parametric bias units were updated during training of the networks. Each unit has an individual \( \delta_i \) error value, which is back-propagated from the \( \delta_j \) of the output layer of a connected network. The internal \( \rho_i \) value collects the error for the particular task and the unit’s \( pb_i \) is used as input for the hidden layer. The values are updated according to the following update rule:

\[
\Delta \rho_i = \mu \cdot \Delta \rho_i' + \eta \cdot (1 - \mu) \cdot \delta_i \\
\Delta \rho_i' = \Delta \rho_i \\
\rho_i = \rho_i - \Delta \rho_i \\
pb_i = \tanh(\rho \eta_i)
\]

where \( \eta \) is the unit’s training rate and \( \mu \) is its momentum.

With each task in training having its own parametric bias vector, the differences between data sets, or tasks in this case, cause the vectors to move away from each other. The bigger the difference between two tasks, the stronger the force that pulls the PB-vectors apart. This way, parametric bias vectors self-organize in such a way that the distance between them reflects the relative difference between the data sets.

In our case, the parametric bias consisting of three units is shared by three networks. This imposes a limitation on the training method - since training one network effectively changes the shared PB input for the other networks, all networks have to be trained in parallel. Also, to prevent overtraining of one task, the tasks need to be switched often. For these reasons, we introduced the repositories of observed training data, actions and their consequences for each of the tasks and trained the networks on random selection of samples.

IV. TRAINING

The training of the model consists of three major phases:

1) exploration,
2) updating of the training data,
3) training of the networks.

These three phases are repeated to iteratively improve the approximation of the value function, world model and optimize the generated policy.

A. Exploration

The exploration phase is a single attempt at the task, starting from a random position and using the current policy with uniform noise added to the networks output (amplitude: 0.015). The attempt is capped at 200 steps, which is enough to explore a small portion of the environment. The data collected during the attempt is stored in repositories: one containing the observed reinforcement signal, indexed by state discretized by a factor of 16 and the other containing the resulting state (not discretized), indexed by a state-action vector, discretized also by a factor of 16. The only significance of the discretization factor was to reflect the complexity of the environment in the repositories - a coarse discretization may not include thin obstacles, a fine discretization is wasteful as the reinforcement signal was rather uncomplicated and the prediction of the future state straightforward in most of the state space.

B. Updating value function training data

A third repository contained the up-to-date approximation of the value function, indexed by discretized state. These values were updated to minimize the Bellman error:

\[
V(x) \leftarrow r(x) + \gamma \tilde{V}(\tilde{f}(x, u(x)))
\]

where \( r(x) \) is cached during exploration and \( \tilde{V}(\tilde{f}(x, u(x))) \) is the approximated value \( \tilde{V} \) of the predicted next state \( \tilde{f} \), assuming the default policy action for that state is taken with no noise added \( u(x) \). The \( \gamma \) parameter is the discounting parameter used in reinforcement learning. In this case, low \( \gamma \) causes the training data to strongly resemble the reinforcement signal data, while high \( \gamma \) causes the data to follow the curvature of the value function approximation, but with a shift in the direction of the vector field of the current policy as shown in figure 5.

The action, which was taken in the exploration phase is not taken into account here, enabling the use of a higher exploration ratio in that phase, but a lower in the learning phase.
The updates of the value function training data repository are dependent on the current approximation. This circular dependency is the reason why the training data update and network training phases are alternate.

The training data for the state prediction network are non-processed values cached during exploration. Since at first the policy generated by the network is close to standing still (low weights) and the approximated value function is close to zero, the value function training data consists of values resembling the actual reinforcement signals. As the networks learn, the values in the repository resemble the shape of the current value function approximation, shifted up or down according to the reinforcement signal and smoothed in the direction of the policy vector field, as calculated by the world model.

The environment is predictable and doesn’t change, so there could be just one long exploration phase with a high noise ratio, but we found that exploring small portions of the environment is more effective. Probably because a successful trial before the whole state space is explored kick-starts the learning of the value function more so than a collection of all non-zero reinforcement signals somewhere in the space. Such a kick-start for one trial causes the policy generator to try and solve the other tasks the same way, often resulting in successful attempts at similar tasks.

In the learning phase, the policy was improved for around 50 randomly selected states with learning rate 0.01, momentum 0.4 and the two other networks were trained with 50 randomly selected cached states assigned to a randomly selected task with a learning rate of 0.1, momentum 0.8. The learning phase was repeated 20 times to assure frequent switching of the tasks for all networks. The system showed a high degree of sensitivity to the aforementioned parameters, especially the exploration to learning ratio, the learning rate and momentum and the amounts of samples selected to train the networks - some configurations resulted in the parametric bias vectors converging to zero, some caused the approximations to destabilize. The values used in the experiment were chosen by trial and error.

C. Policy improvement

In training of the policy generator, both the value approximation and world model are used.

Since the agent’s aim is to maximize the value function (in the continuous state space), the natural policy seems to be to follow the gradient of the value function uphill in hopes of finding the global maximum. While this premise is good, it has some room for improvement.

The gradient of the approximated value function can be approximated using back-propagation.

For a given state we want to find the action that results in a state with the highest value function. We use the current state and a null action (with some noise added for exploration) as input for the world model network, resulting in the prediction of the next state (see diagram 6). This prediction shouldn’t be very different from the starting point, because the action was null, but the error back-propagation algorithm requires that the activations of all layers be set.

Next, in order to maximize the value function, we set the maximum possible value of the value function (which is limited by 1 since it’s approximated by a neural network) as the desired value, or \(1 - \hat{V}(\tilde{x}_{t+1})\) as the error of the value function approximator and back-propagate this error to its input layer. See diagram 7.

Part of the input of the world model network is the
action vector. The error vector in this part is the error of the suggested action (the noisy null action). The difference between the action input and this error is the action vector that causes the agent to go up the gradient of the value function - a desired behavior for the policy generator.

Because the approximated value function is flat in most of the state space for most of the training time, or contains ripples and local minima and maxima generated by the neural network, we use only those parts of the state space, in which the calculated gradient is significant by filtering out the suggested actions that are below the threshold of noise added to the null action.

This truncation of the training data causes the policy network to generalize the input stimuli to the rest of the state space (see fig. 8). For a partially developed value function, which has a significant gradient only in some small distance from the target, the resulting policy vector field will converge in the center of the target. This way, even a single successful attempt at the task can cause the agent to head in that direction next time.

The world model network and value function network are not trained while generating data for the policy network.

V. RESULTS

The system was tested on eight tasks, differing in environment configurations and target positions. The task set was designed to have three degrees of organization - two degrees for the position of target and two different environment configurations, with an obstacle on one side or the other, as presented in figure 1.

After 30000 repetitions of the training loop the agent is able to reach the target for all eight tasks from any starting position. The generated trajectories are smooth rather than short and clearly follow the gradient of the value functions acquired through learning of the tasks presented in fig. 9. The obstacle position clearly influences the maximum of the value function. The fact that all tasks are solved by the same set of networks is the cause of the similar shape of the value functions for tasks with the same target position (compare left and right column in fig 9).

The position of the target is not known to the agent in any way other than from the reinforcement signal, making the problem hard: until the target is found, the agent is unaware of its position. The only way the agent measures the distance to the target is by the internally developed state value function, the shape of which depends on the placement of obstacles, which also is discovered by exploring the environment and approximated using a neural network.

Despite that, not only does the agent learn all eight tasks, but it also forms an internal model of the task set within the parametric bias space. After training, the parametric bias vectors form a shape that reflects the internal structure of the task set - two congruent quadrangles, corresponding to two set of tasks with the same obstacle configuration, but different target placement, as shown in fig. 10.

![Fig. 8. A suboptimal approximation of the value function (top), the calculated gradient (middle) and the generalized policy (bottom).](image)
Fig. 9. Value functions developed for the eight tasks after around 30000 repetitions of the training loop. Parametric bias vectors corresponding to these tasks are shown in fig. 10.

field that converges to a point between two of the trained targets corresponding to the vectors.

This generalization of response to previously untrained target positions is attained solely by changing the value of the parametric bias, no other parameter of the system is modified. The policy generating network uses the new, untrained value of the parametric bias to generate a policy that makes the agent move toward an imaginary target, the position of which can be identified from the position of the parametric bias value relative to the parametric bias values of the trained tasks. In other words, the agent recognizes the target position from the value of the parametric bias.

We attempted leaving out one of the tasks when training the networks, but letting its parametric bias develop nonetheless. The parametric bias of that task assumes its correct

position in the parametric bias space relative to the trained tasks.

VI. DISCUSSION

The method described in this paper is capable of learning multiple continuous state space reinforcement learning tasks and generates good approximations of the optimal value function using very limited exploration data. The training is robust thanks the use of cached exploration data and stimulus generalization from training data for the policy truncated to include only relevant samples. The use of feed-forward neural networks allows for calculation of the gradient of the approximated value function and smoothing of its shape.

The agent was trained on a structured set of tasks. We claim that the development of an internal representation of the task set is not only possible, but also beneficiary for the training: it can be seen during training that the information about obstacles in one task is promptly transferred to three other tasks. The same holds true to the position of the target: we observed skill transfer between tasks having the same target position, as long as the similarity of tasks is properly encoded in the parametric bias space.

Viewed from another perspective, the three-dimensional structure of the task set can be seen not as relationship between different environments, but as a map of one, but changing environment. By observing the environment in several states, the agent is able to generate reactions to previously unseen states. For a trained agent, setting the parametric bias value in the internal map to follow a trajectory corresponds to a moving target, which the agent pursues, despite being trained on static targets only. It can be argued that the position of the target is given as input in the form
of parametric bias in this case, but it is only due to the self-organization of the PB-vectors that the parametric bias space has any correspondence to the positions in the state space.

It is worth noting that the system uses three major machine learning paradigms: supervised learning, reinforcement learning and unsupervised learning postulated in [8] as used by biological organisms, albeit here they are interwoven on different levels. Reinforcement-tagged data is collected and transformed to training data for the networks and supervised training of the networks triggers the self-organization of the parametric bias vectors. It is impossible to clearly distinguish parts that use one paradigm, but not another.

VII. Conclusion and Further Studies

The experiment described in this paper shows that parametric bias can be used for structure discovery when learning a set of skills, or reinforcement learning problems. Observed effects include self-organization of an internal model of the task set, generalization from exploration of only part of state space, re-using of exploration data for solving other tasks, using similar policies for similar tasks (skill transfer) and generalization of behavior when given previously untrained input. The result confirms previous research on the properties of the parametric bias ([1], [7]), expanding it to problems for which correct output is not explicitly given.

The result leaves many possible directions for improvement and further experimentation. The method should be tested on standard benchmark reinforcement learning problems - pendulum up-swing, cart-pole balancing and the valley car driving problem. The state space search problem was chosen as it facilitated creating multiple related tasks. It is high on the agenda of the authors to test the method on a physical robot as a "sanity check" of the simulation.

REFERENCES