Universiteit
Leiden
The Netherlands

Enhancing autonomy and efficiency in goal-conditioned reinforcement learning

Yang, Z.

Citation
Yang, Z. (2025, February 26). *Enhancing autonomy and efficiency in goal-conditioned reinforcement learning*. *SIKS Dissertation Series*. Retrieved from https://hdl.handle.net/1887/4196074

Version:	Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/4196074

Note: To cite this publication please use the final published version (if applicable).

Chapter 4

Continous Episodic Control

Authors: Zhao Yang, Thomas Moerland, Mike Preuss, Aske Plaat
Published in IEEE Conference on Games (CoG), 21-24 August 2023, Boston, USA.

Abstract

Non-parametric episodic memory can be used to quickly latch onto high-rewarded experience in reinforcement learning tasks. In contrast to parametric deep reinforcement learning approaches in which reward signals need to be back-propagated slowly, these methods only need to discover the solution once, and may then repeatedly solve the task. However, episodic control solutions are stored in discrete tables, and this approach has so far only been applied to discrete action space problems. Therefore, this paper introduces **C**ontinuous **E**pisodic **C**ontrol (CEC), a novel non-parametric episodic memory algorithm for sequential decision making in problems with a continuous action space. Results on several sparse-reward continuous control environments show that our proposed method learns faster than state-of-the-art model-free RL and memory-augmented RL algorithms, while maintaining good long-run performance as well. In short, CEC can be a fast approach for learning in continuous control tasks. [1]

[1] Code can be found at https://github.com/yangzhao-666/cec.

4.1 Introduction

Deep reinforcement learning (RL) methods have recently demonstrated superhuman performance on a wide range of tasks, including Gran Turisma (Wurman et al., 2022), StarCraft (Vinyals et al., 2019), Go (Silver et al., 2017), etc. However, in these methods, the weights of neural networks are slowly updated over time to match the target predictions based on the encountered reward signal. Therefore, even if the agent finds the solution to the problem, the learning still can be fairly slow, and the agent may not be able to solve the problem in subsequent episodes. This is one of the reasons state-of-the-art RL methods generally require many interactions with the environment and substantial computational resources (Mnih, Kavukcuoglu, Silver, Rusu, Veness, Bellemare, Graves, Riedmiller, et al., 2015; Silver et al., 2016). Non-parametric episodic memory (EM) is introduced to directly store high-rewarded experiences, enabling the agent to quickly latch onto these experiences in reinforcement learning problems. Compared to parametric deep RL methods, these methods only need to solve the task once, because the solution is stored in memory and can be quickly retrieved. This is especially helpful for tasks where the agent only gets a final reward once it succeeds, i.e. sparse reward problems (Ecoffet et al., 2021). Although parametric RL methods do perform well in the long run (due to its generalizability, optimality and ability of dealing with stochasticity), sometimes we prefer a quick and less computationally expensive solution to the problem (Blundell, Uria, Pritzel, Li, Ruderman, Leibo, Rae, et al., 2016). The ability to quickly latch onto recent successful experiences is also clearly evident in nature, where scrub jays for example store food and directly remember the exact location using episodic memory (Clayton & Dickinson, 1998).

Episodic memory is a term that originates from neuroscience (Tulving, 1972), where it refers to memory that we can quickly recollect. In the context of RL, this concept has generally been implemented as a non-parametric (or semi-parametric) table that can be read from and written into rapidly. Information stored in the memory can then either be used directly for control (for action selection) or to generate training targets for parametric deep RL methods. When used for control (usually called episodic control), episodic memory stores state-action pairs and their corresponding episodic returns in memory and extracts the policy by taking actions with maximum returns. It is therefore conceptually similar to tabular RL methods (C. J. C. H. Watkins, 1989) but with a different update rule, where stored values are directly replaced with larger encountered ones instead of being gradually updated towards targets.

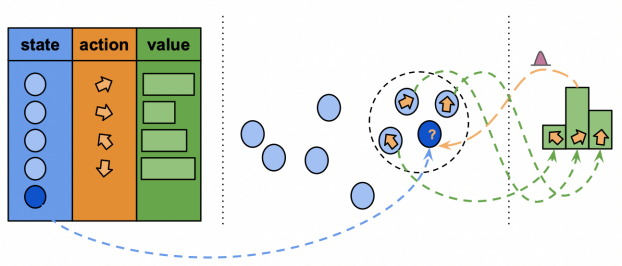

Figure 4.1: Conceptual illustration of the Continuous Episodic Control (CEC) algorithm. Left: We store a non-parametric table/buffer of state-action-value estimates obtained from previous episodes. Middle & Right: Action selection for a novel query state (dark blue circle) is performed by collecting its nearest neighbours in the buffer (dashed circle), and then sampling one of the actions of these neighbours proportional to their value (right). After possibly injecting exploration noise the selected action for the query state is returned.

Since current episodic control methods maintain a value estimate for each state-action pair (Blundell, Uria, Pritzel, Li, Ruderman, Leibo, Rae, et al., 2016), they cannot naturally deal with continuous action spaces. One solution to this problem is to combine episodic memory with deep RL methods, where the episodic memory part generates training targets for a value network that is used to act (value-based approaches (Mnih, Kavukcuoglu, Silver, Rusu, Veness, Bellemare, Graves, Riedmiller, et al., 2015)) or to guide act (actor-critic approaches (Lillicrap et al., 2015)) in the environment. The drawback of this approach is that we not only have to maintain extra memory, but again rely on the relatively slow deep RL agent for action selection and thereby performance is affected. The natural question that arises is: **can we directly use episodic memory for action selection in continuous control tasks, without relying on parametric RL methods?**

This work therefore introduces **C**ontinuous **E**pisodic **C**ontrol (CEC), an algorithm that uses episodic memory directly for action selection in tasks with a continuous action space. The main idea is based on the principle of generalisation: similar states generally require similar actions to obtain high returns. However, given a new query state, we 1) may not have that exact state in our buffer, and 2) we definitely only have value estimates for a few points in the continuous space of possible actions. We solve these issues by 1) specifying the nearest neighbour search in state space and 2) selecting actions proportionally to their value estimate, injected with Gaussian noise per action. The overall process of CEC is illustrated in Figure 4.1.

Experiments on a range of continuous control tasks show that our method outperforms state-of-the-art model-free and memory-augmented RL methods in various

continuous control problems. We also visualize how CEC manages to capture state-action values quickly and reliably in the beginning of the training, which results in steeper learning curves. In short, CEC seems to be a promising approach to learn quickly in continuous control tasks.

4.2 Background

In reinforcement learning (Sutton & Barto, 2018), an agent interacts with the environment. This process is formed as a Markov Decision Process (MDP) defined as the tuple $M = \langle S, A, P, R, \gamma \rangle$, where S is a set of states, A is a set of actions the agent can take, P specifies the transition dynamic of the environment, R is the reward function, and γ is the discount factor. At timestep t, the agent observes a state $s_t \in S$, and selects an action $a_t \in A$. The environment returns a next state $s_{t+1} \sim P(\cdot|s_t, a_t)$ and an associated reward $r_t = R(s_t, a_t, s_{t+1})$ after the action is executed. The agent selects actions according to a policy $\pi(\cdot|s_t)$ that maps a state to a distribution over actions. The goal is to learn a policy that can maximize the expected return: $\mathbb{E}_{s_{t+1}\sim P(\cdot|s_t,a_t),a_t\sim\pi(\cdot|s_t),r_t\sim R(\cdot|s_t,a_t)}[\sum_{t=0}^{T}\gamma^t \cdot r_t]$, where $\gamma \in [0, 1]$.

Define the state-action value $Q(s_t, a_t)$ as the expected return:

$$Q(s_t, a_t) = \mathbb{E}_{P,\pi,R}[\sum_{k=t}^{T}\gamma^k \cdot r_k],$$

when the agent starts from state s_t and action a_t at time-step t. Q-learning learns the optimal state-action value function by updating the current estimated value towards a target value (TD-target) following:

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot [r_t + \gamma \cdot \max_{a' \in A} \hat{Q}(s_{t+1}, a') - \hat{Q}(s_t, a_t)]\,,$$

where α is the learning rate, $\hat{Q}(s_t, a_t)$ is estimated state-action value.

In deep Q-learning (Mnih et al., 2013), state-action value function $Q(s_t, a_t)$ is approximated by a neural network denoted by $Q_\phi(s_t, a_t)$. Then the function is updated by minimizing the mean squared error $L(\phi, D)$ between the TD-target and the estimated value computed using samples from D:

$$L(\phi, D) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim D}\Big(r_t + \gamma \cdot \max_{a' \in A} \hat{Q}_\phi(s_{t+1}, a') - \hat{Q}_\phi(s_t, a_t)\Big)^2$$

After the optimal state-action value function is learned, the optimal policy can be exacted by greedily taking actions that have highest state-action values:

$$a_t = \operatorname*{argmax}_{a_t} Q(s_t, a_t). \tag{4.1}$$

4.3 Related Work

In this section, we will discuss RL methods that use or are augmented with episodic memory. There are mainly two directions, one that directly uses episodic memory for control ('episodic control'), and another which uses data in the episodic memory to guide the learning update of parametric RL agents ('memory-augmented RL'). We will discuss both.

Episodic control Model-free episodic control (MFEC) (Blundell, Uria, Pritzel, Li, Ruderman, Leibo, Rae, et al., 2016) uses a non-parametric table to store episodic return $G_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$ for each state-action pair and only overwrites it when a higher return of the state-action pair is encountered:

$$\hat{Q}_{em}(s_t, a_t) \leftarrow \max[\hat{Q}_{em}(s_t, a_t), G_t]$$

The optimal policy is extracted by greedily taking actions with the greatest values stored in the EM table, similar with Equation (4.1). It outperforms state-of-the-art model-free RL methods on some Atari games in the short run. Neural episodic control (Pritzel et al., 2017b) proposed to use a semi-tabular approach called differentiable neural dictionary (DND) which can store trainable representations which are updated during the training process. NEC selects action in the same way as MFEC. The slowly changing representations and quickly updating values lead to better performance than MFEC. To the best of our knowledge, no previous method has studied the use of episodic memory for action selection in continuous action spaces.

Memory-augmented RL Instead of using non-parametric models directly for action selection, information stored can also be used to enhance learning of a deep RL agent. During updates of value-based deep RL methods (Mnih, Kavukcuoglu, Silver, Rusu, Veness, Bellemare, Graves, Riedmiller, et al., 2015) or critic parts of actor-critic RL methods (Lillicrap et al., 2015), the neural network is updated by minimizing the error between a TD target and the currently estimated Q value. Therefore, values from the episodic memory can be used to form a new update target alongside the original TD target:

$$L = \alpha \cdot (\hat{Q} - \hat{Q}_{original})^2 + \beta \cdot (\hat{Q} - \hat{Q}_{em})^2, \tag{4.2}$$

where L is the loss/error to minimize, α and β are hyper-parameters to balance the contribution of the original TD target $\hat{Q}_{original}$ and the episodic target $\hat{Q}_{em}$, and $\hat{Q}$ is

the current Q value estimation. Episodic Memory Deep Q-networks (EMDQN) (Lin et al., 2018), Episodic Memory Actor-Critic (EMAC) (Kuznetsov & Filchenkov, 2021), Model-based Episodic Control (MBEC) (Le et al., 2021) and Generalizable Episodic Memory (GEM) (H. Hu et al., 2021) all use this approach, but in different ways. EMDQN combines values from the EM with 1-step TD target in deep Q learning using fixed α and β. EMAC combines values from the EM with 1-step TD target in DDPG for solving tasks with continuous action space. MBEC brings episodic memory into the model-based RL settings, and combines values from the EM with 1-step TD target using a learned weight instead of fixed α and β. GEM uses EM for implicit planning and performs the update with an n-step TD target (Sutton & Barto, 2018), first taking n steps based on the EM then bootstrapping from a Q-network. Aforementioned methods are all trying to distill information stored in EMs into parametric models. Although by doing so the performance of parametric models is enhanced, meanwhile it also loses the underlying benefit of EM which is fast reading and writing.

Table 4.1: Overview of related memory-based RL approaches. None of these previous works uses episodic memory for action selection in problems with a continuous action space.

Method	EM usage	Action space
MFEC	action selection	discrete
NEC	action selection	discrete
EMAC	target update	continuous
EMDQN	target update	discrete
GEM	target update	continuous
MBEC	target update	discrete
Neural Map	feature storage	discrete
CEC (ours)	**action selection**	**continuous**

Finally, note that there are various other methods, such as Neural Map (Parisotto & Salakhutdinov, 2018), that learn to interact with an external (tabular) memory module to deal with partial observability, but these approaches do not store any value or policy estimates in the table, like episodic memory. A summary of related work is provided in Table 4.1.

4.4 Continuous Episodic Control (CEC)

The principle of CEC is that actions the agent takes in similar states should also be similar. It is implemented by maintaining a table that contains a state-action-value

tuple for each row. Each entry stores the cumulative discounted reward the agent received in an episode after taking the specific action in the specific state. To interact with the memory, CEC mainly has two functions: an **update** function that adds new encountered data into the memory or updates old data after an episode is terminated, and an **action selection** function that is used during the episode to select actions.

Update We update the episodic memory after each collected episode. Intuitively, we want our episodic memory to have good coverage of the visited state space, for which we use the following mechanism. We first collect state-action-value pairs (s, a, v) from episodes, where each value

$$v = \hat{Q}_{em}(s, a) = \sum_{i=0}^{T} (\gamma)^i \cdot r_{t+i} | s_t = s, a_t = a$$

is the cumulative discounted reward obtained after taking action a in state s (for readability we often omit the dependence of v on s and a when it is clear from the context). For each collected (s, a, v), we then find its closest state neighbour s^c in the current memory. The new state-action-value tuple is directly added into the CEC memory if the state is far away from its closest neighbor. Thereby, if the CEC memory does not contain any information about the area near the incoming state, we directly add it into the memory.

When its closest neighbor is within a certain distance threshold d, and its corresponding value is smaller than the value of the incoming state, we replace the previous state-action-value tuple with the new one. The intuition is that when the closest neighbor is close enough, we assume that the CEC memory has information about the area near the new state. However, if the stored value is worse than the one of the new state-action-value tuple, we do overwrite it since we now have information about a better action in the specific state region. The update rule can be expressed as:

$$\text{EM}(s^c, a^c, v^c) \leftarrow \begin{cases} (s, a, v) & \text{if } f(s, s^c) < d \\ & \text{and } v > v^c, \\ (s^c, a^c, v^c) & \text{otherwise} \end{cases} \tag{4.3}$$

Here, superscript c indicates the closest neighbour in the buffer, superscript l indicates the least recently updated entry in the buffer, EM() refers to the slot of the specific entry in the memory buffer, $f()$ specifies a distance measure and d specifies a distance threshold. Once the buffer has filled up, when a new coming state is out of the distance

threshold of its closest state, the least updated tuple will be thrown away and the new coming tuple will be added, otherwise, it will follow the aforementioned update rule:

$$\mathrm{EM}(s^l, a^l, v^l) \leftarrow \begin{cases} (s, a, v) & \text{if } f(s, s^c) > d, \\ (s^l, a^l, v^l) & \text{otherwise} \end{cases} \tag{4.4}$$

Here, superscript l indicates the least recently updated entry in the buffer.

Action selection When a state s is encountered, a corresponding action a is selected based on information stored in the CEC memory. We first find the k nearest-neighbors of state s, denoted by S_k, then filter out neighbors which are too far away. Especially when the memory is sparse, the closest neighbors could still be very far away and we do not consider these faraway neighbors as 'similar' states. Next we let values stored for state-action pairs of the filtered neighbors S_k decide which action should be selected. Higher values indicate better actions, but for exploration purposes we still want to give other actions a chance to be chosen. Therefore, we sample an action from the set S_k proportional to their value estimates, by taking a softmax:

$$p(a_s|S_k) = \frac{e^{v_s/\tau}}{\sum_{s' \in S_k} e^{v_{s'}/\tau}}, s \in S_k \tag{4.5}$$

where subscripts s and s' indicate the state the specific action and value belong to in the buffer, and τ is a temperature factor to scale exploration. We now have a sample from the discrete set of a_i points in the buffer, but our true action space is of course continuous. We therefore transform our sample to a continuous distribution by adding Gaussian noise to the sampled action with probability ϵ. During the evaluation, all exploration is turned off and actions are selected by greedily taking the action of the closest neighbour.

Feature embedding While dealing with a high-dimensional state space, storing the original state and finding its neighbors is expensive. We therefore utilize random projections to project the original state space into a smaller space. We first sample a random matrix $A \in \mathbb{R}^{D_{ori} \times D_{new}}$ for a standard Gaussian distribution, where D_{ori} and D_{new} are the dimensions of the original and projected state space, respectively. We then (matrix) multiply the original state by A to get an embedded state, i.e. $s_{new} = A \cdot s_{ori}$, where s_{new} and s_{ori} are the embedded and original state, respectively.

These transformations will still preserve the relative distance in the original state space. We only use the random projection for **FetchReach** and **Safexp-CarGoal** environments, while for other environments the original state is directly stored and queried.

The overall algorithm can be found in Algorithm 1. $\texttt{knn}(k, s, n, d)$ is a function that first finds the k nearest neighbors of the state s, then filters out states whose distances are larger than n times of the distance threshold d.

Algorithm 1 Continuous Episodic Control (CEC)

Initialize: CEC Memory M, environment `Env`, episode memory M_{eps}, distance threshold d, filter factor n, k for k-nearest-neighbors, discount factor γ, noise standard deviation σ, exploration factor ϵ.
while training budget left **do**
  $s \leftarrow$ `Env.reset()` {Reset the environment.}
  $done$ = false
  while not $done$ **do**
    // SELECT ACTION
    $S_k = \texttt{knn}(k, s, n, d)$ {Find k nearest-neighbors of the state s within threshold $n \times d$.}
    $a \sim p(a_s|S_k)$ {Select actions based on Equation (4.5)}.
    if $\Delta \sim U(0, 1) < \epsilon$ **then**
      $\psi \sim \mathcal{N}(0, \sigma^2)$
      $a \leftarrow a + \psi$
    end if
    $s', r, done \leftarrow \texttt{Env.step}(a)$
    Append $[s, a, r]$ to M_{eps}
    $s \leftarrow s'$
  end while
  // UPDATE
  $v = 0$
  while M_{eps} **not** empty **do**
    $s, a, r \leftarrow M_{eps}.\texttt{pop()}$
    $v \leftarrow r + \gamma \cdot v$
    Update M using (s, a, v). {Update CEC memory based on (Equations (4.3) and (4.4))}.
  end while
end while

4.5 Experiments

In this section, we will first give some insights of CEC by using simple toy examples. Then we scale up to more complex continuous control tasks and compare with state-of-the-art RL and memory-augmented RL methods. Dimensions of environments we used for this work are shown in Table 4.2. Although **Safexp-PointGoal** and **Safexp-CarGoal** have the same action space (two actuators, one for turning and one for moving), in **Safexp-CarGoal** both turning and moving require coordinating both of the actuators, which is more complex. Every experiment is averaged over 5 independent runs and standard errors are plotted as well.

Table 4.2: Dimensions of state spaces and action spaces of different environments.

Env	State Space	Action Space
GrowingTree	1	1
MountainCarContinuous	2	1
PointUMaze	4	2
Point4Rooms	4	2
Safexp-PointGoal	24	2
Safexp-CarGoal	36	2
FetchReach	13	4

4.5.1 Toy Examples

In order to first get an overview and more insights into the method, we create a toy example called GrowingTree. In this environment, the agent needs to 'grow' the 'tree' to the given target height. More details can be found in Tab. **??**. The continuous action space ranges from -0.1 to 0.1, which means the agent can chose to 'cut' (< 0) the 'tree' or 'grow' (> 0) the 'tree'. The observation for the agent is the height of the current 'tree' and the height changes every step by simply adding the action to the previous observation. The optimal policy for the agent to get the final reward is to take 0.1 as the action every step whatever the height of the tree is, which causes the 'tree' to grow linearly.

We train the CEC agent on the toy example, with results shown in Figure 4.2. Every row of the figure shows an evaluation of the CEC agent after another $10k$ training steps, from top to bottom. The n^{th} row therefore represents the evaluation results after $n \times 10k$ training steps. The left part of the figure shows the heights of the tree during the evaluation episode. If the optimal policy is learned by the agent,

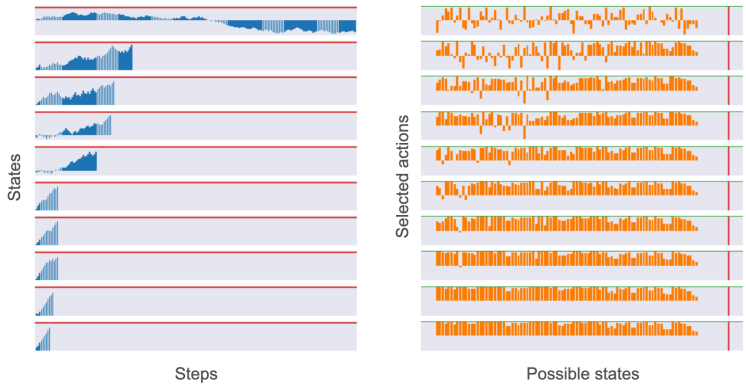

Figure 4.2: Evaluation on the toy example. We train the CEC agent on the toy example for 100 k steps and evaluate it every 10k steps. Training are from top to bottom. **Left**: the performance during the evaluation. It gradually takes shorter episode to reach the goal. x-axis is taken steps during the evaluation episode and y-axis is the state (the height of the current 'tree'). Red horizontal lines are targets we set. **Right**: actions selected by the trained CEC agent. x-axis represents different states while y-axis represents actions. Red horizontal lines are final target states and green vertical lines are optimal actions (0.1).

Table 4.3: Details about GrowingTree environment. The agent only gets a final reward when it successfully reaches the given goal (with a distance tolerance).

state space	[-2, 2]
action space	[-0.1, 0.1]
maximum steps	200
goal	1.0
final reward	1.0
distance tolerance	0.1
start height	0

the height of the tree will increase quickly with the number of steps (x-axis) increases. The right figure shows the actions the CEC agent takes for every possible state (we only evaluate states that lie within [0,1]). The optimal action in every state is (near) 0.1.

In the top rows, the agent still chooses to 'cut' the 'tree' sometimes, but the overall trend of the 'tree' is still growing. Although the solution is not the optimal, the agent is able to solve the problem successfully. In the middle rows, the agent can solve the task fairly quickly. However, if we look at the middle rows of the right figure, the chosen action for every possible state is still far from the optimal and a few of them are still smaller than 0. Gradually, selected actions of all possible states are getting

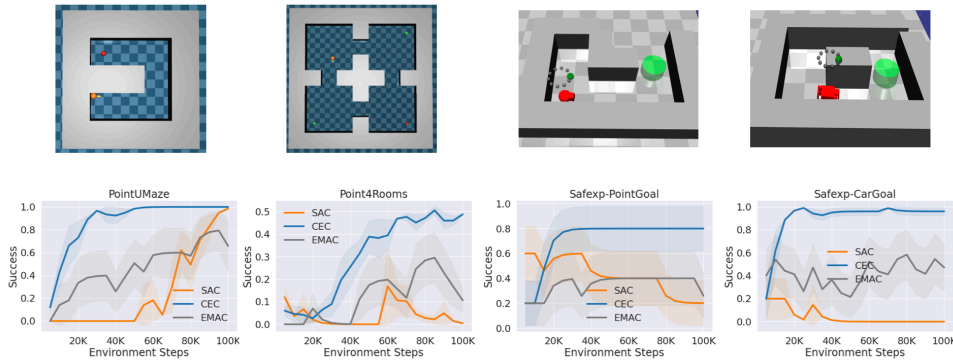

Figure 4.3: Four continuous navigation environments we used in this work and their corresponding results. From left to right, they are 'PointUMaze', 'Point4Rooms', 'Safexp-PointGoal' and 'Safexp-CarGoal', respectively. We see the proposed method CEC (blue lines) outperform SAC (orange lines) and EMAC (grey lines) in all these four tasks. Results are averaged over five independent runs and the shaded area represents standard errors.

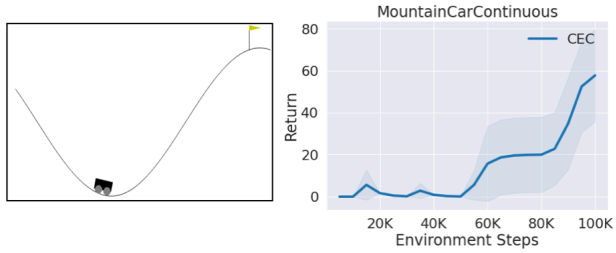

Figure 4.4: MountainCarContinuous environment and learning curve of the CEC agent on it. The agent gradually learns to solve the problem.

closer and closer to the optimal values (bottom rows).

Through this simple example, we see that our principle "similar states should have similar actions" does work and can indeed quickly solve the problem with a non-optimal solution, while it may also discover a near-optimal solution given more time.

We next test our method in a simple classical RL task, **MountainCarContinuous**. We use the sparse reward version of the task, where the agent only gets a final reward (+100) once it brings the car successfully to the flag on the top of the mountain. Since the initial position of the car is randomly generated near the bottom of the mountain, the agent also needs to deal with proper generalization. The learning curve of CEC on this task is shown in Figure 4.4. The CEC agent gradually learns to solve the task, which gives a first indication that our method can indeed be used for basic RL

problems that require generalization.

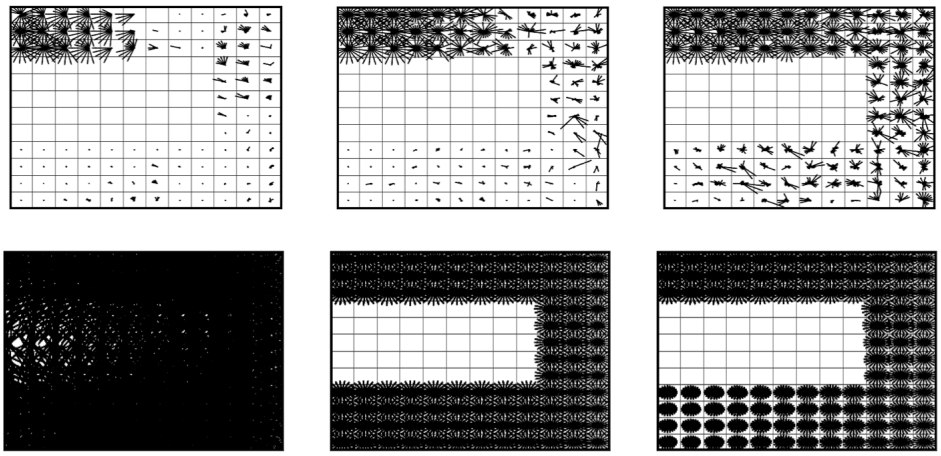

Figure 4.5: State values produced by CEC (top) and SAC (bottom) during the training on PointUMaze. Figures from left to right are after 5k, 50k, and 100k steps of training, respectively. Since the state space is continuous, we discretize the location information into 12×12 grids and the orientation information into 20 directions (represent as different directed arrows). Longer arrows indicate larger state-action values, and SAC outputs very large values at the beginning which is not rational while CEC has larger values for the area near the goal.

4.5.2 Continuous Navigation Tasks

We next move to more complicated test environments, in which we will also compare CEC to two baselines: the model-free deep RL method SAC (Haarnoja, Zhou, Abbeel, & Levine, 2018) and a memory-augmented deep RL method EMAC (Kuznetsov & Filchenkov, 2021) which uses EM for learning targets. We test these in four continuous Mujoco navigation tasks, shown in Figure 4.3. They are PointUMaze, Point4Rooms, Safexp-PointGoal, and Safexp-CarGoal, respectively. These tasks are all sparse-reward problems, in which the agent only gets a final reward when the task is successfully completed. Locations of both goals and agents are initialized with small randomness every episode. Hyper-parameters we are using for CEC on these tasks can be found in Table 4.4.

Results are shown in Figure 4.3. CEC agents can quickly solve the task and outperform SAC and EMAC agents in all environments. In these four navigation tasks, there are explicit bottlenecks and CEC can quickly remember the discovered solution by only discovering it once. In contrast, the deep RL methods need to discover

Table 4.4: Hyper-parameters of CEC we are using for continuous navigation tasks. Notations can be found in Algorithm 1.

Env	k	τ	σ	n	d
PointUMaze	5	0.1	0.3	1	0.1
Point4Rooms	5	1	0.3	3	0.1
PointGoal	5	0.1	0.1	1	0.1
CarGoal	1	10	0.1	3	0.1

the solution multiple times in order to back-propagate reward signals for learning, and this rediscovery is time-consuming due to the bottlenecks in the tasks.

We also visualize state-action values of CEC and SAC during the training for **PointUMaze**, which is shown in Figure 4.5. The agent always starts from the bottom-left corner and the goal is always located at the top-left corner. Both of these locations are initialized with small randomness. Since the state space of the environment is 3 dimensional (x location of the agent, y location of the agent, orientation of the agent) and continuous, we discretize x, y locations as a grid world (12×12) and orientations as a group of different directed arrows (20 different directions). The length of the arrow represents the state-action value of the state that is jointly indicated by the arrow and the position of the cell and the action for that specific state (only one action is maintained for each state). The longer the arrow is, the larger the state-action value is. From Figure 4.5, we see that in the beginning the SAC agent strongly overestimates all state-actions values, due to the random initialization(arrows went across the whole state space, meaning all state-action values are extremely large). On the contrary, the CEC agent obtains reasonable estimates more quickly, also identifying larger value estimates near the goal region. While training progresses, both agents gradually improve their estimates to better propagate high-value estimates towards the start region (i.e., identifying a good policy). Both agents eventually solve the task, but CEC was much faster, as can be seen in Figure 4.3 as well.

4.5.3 Robotics Control Task

We finally investigate our method on a more complicated robotics control task: **FetchReach**. The goal of the task is to control the robotics arm to reach the red point, which is reset to a randomly sampled position every episode. The environment and its associated learning curve are shown in Figure 4.6. We use $k = 5, \tau = 1, \sigma = 0.1, n = 3, d = 0.5$ for CEC. We see that CEC still outperforms the state-of-the-art baseline methods, although the difference is less pronounced compared to the previous navigation tasks.

We attribute this to the lack of bottlenecks in this task. CEC is probably most useful when a task has some narrow passages, which are challenging from an exploration point of view, and in which CEC can quickly latch onto a few successful trials. Nevertheless, even in a robotics task with no bottleneck characteristics, CEC performs on par with current state-of-the-art model-free baselines.

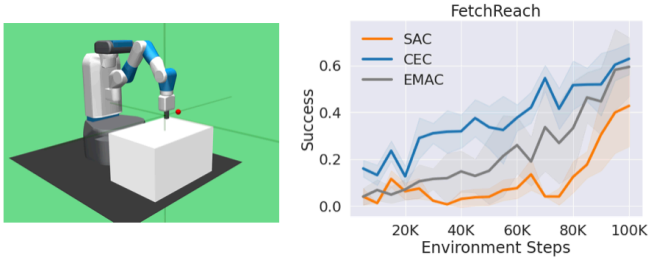

Figure 4.6: FetchReach environment and performance of different agents. CEC still outperforms other RL agents but the benefits are not as large as the ones in previous navigation tasks.

4.6 Conclusion and Future Work

In this work, we propose a new episodic control method called continuous episodic control (CEC) which can handle tasks with a continuous action space. It is the first work that uses episodic memory for continuous action selection. By following the principle that "similar states should have similar actions", CEC agents outperform the state-of-the-art RL method SAC and memory-augmented RL method EMAC in various sparse-reward continuous control environments. Although RL methods might outperform episodic control in the long run, sometimes we do prefer a quick and cheap solution to the problem. Several experiments show CEC has strong performance on continuous navigation tasks, especially when there are bottlenecks in the environment.

Although non-parametric methods (like episodic control) require increasing memory with new data, this can be eliminated by employing a throw-away mechanism that only stores important data. In fact, experimental results show that episodic control is able to scale up to Atari games (Silver et al., 2016), such as Ms.Pac-Man, Space Invaders, and Frostbite, where pixel images are used as observations. However, episodic control also has its limitations. For example, using Monte-Carlo returns as its state-action values has high variance, and also it will not be able to learn the optimal solution in stochastic environments by definition. But we believe sometimes we do

prefer a quick (might be sub-optimal) solution.

In the future, it would be very interesting to investigate better feature embedding methods (such as using latent features of a pre-trained VAE (Kingma & Welling, 2013)) and exploration strategies that work well with CEC. In addition, we could also look at methods to improve the nearest neighbour search, for example, based on clustering. Finally, we believe there is potential to combine episodic control and parametric reinforcement learning methods to distill strengths from both sides and then end up with a more powerful agent.