Solving POMDPs with Neural Network State Update Functions and Hill Climbing

By Jason Whitmore

January 2025

Abstract: This paper expands on the literature of methods to solve POMDPs through the use of state update functions. A new method is introduced that implements state update functions using neural networks. This new method is tested through three experiments. The first two experiments seek to understand the relationship between the method's hyperparameters and performance. The third experiment compares the new method with existing state update functions in terms of performance. These experiments produce both expected and unexpected outcomes.

1. Introduction

Solving Markov Decision Processes (MDPs) requires finding optimal policies that maximize discounted returns. This is the primary problem in reinforcement learning. Even with constraints (e.g. model free), the literature contains many different algorithms and techniques to find optimal policies.

However, very few real world applications fit into the MDP framework. Observations containing little useful information – rather than states – are received by the agent. Partially Observable Markov Decision Processes (POMDPs) describe this setting.

Finding optimal policies for POMDPs (also called "solving" POMDPs) is difficult. In this paper, a simple technique is used to map POMDPs to MDPs and therefore become much easier to solve.

2. Related Work

2.1. POMDPs

In contrast to MDPs, POMDPs are more applicable to real world settings. This is because "In [many interesting cases], potentially important aspects of the environment's state are not directly observable" [1, p. 464].

A minor addition distinguishes POMDPs from MDPs:

> The environment would emit not its states, but only *observations* – signals that depend on its state but, like a robot's sensors, provide only partial information about it [1, p. 464].

With this change in the environment, the interaction loop between the agent and the environment also changes:

> The environmental interaction would then have no explicit states or rewards, but would simply be an alternating sequence of actions [$a_t \in A$ [1]] and observations [$o_t \in O$] [1, p. 464].

It should also be noted that some parts of the literature on POMDPs use the term "perception" rather than "observation", but the terms seem synonymous.

2.2. Using Observations as States

In a simplistic way of solving a POMDP, each observation can be treated as if it were a state in a MDP. However, this approach is limited and is well acknowledged in the literature:

> Perhaps the simplest example of an approximate state is just the latest observation, [$s_t = o_t$]. Of course this approach cannot handle any hidden state information [1, p. 468].
>
> Robotics has seen a surge of interest in reactive robots agents that use their current sensor values to choose their next action but as the tasks get more complex, we are finding that some tasks cannot be solved with perception-to-action mappings alone [2, p. 2].

In other words, the observation does not have enough information in it to even approximate the information contained within the state. Information in the state that is not in the observation could be very important.

These issues make using the observation as if it were a state a poor technique to solving a POMDP.

2.3. History

With using observations as states being ineffective, utilizing the agent's previous observations, rewards, and actions may be more effective. This *history* is defined:

> ...we can recover the idea of state as used in this book from the sequence of [observations, actions, and rewards]. Let us use the word *history*, and the notation [h_t], for an initial portion of the trajectory up to an observation: [$h_t = o_0, a_0, r_0, o_2, ..., o_t$ [2]]. The history represents the most that we can know about the past without looking outside the data stream (because the history is the whole past data stream) [1, p. 465].

However, using the history directly is unrealistic because "the history grows with t and can become large and unwieldy" [1, p. 465]. The history must therefore be transformed before it can become a usable state:

> The idea of state is that of some compact summary of the history that is as useful as the actual history for predicting the future. ... To be a summary of the history, the state must be a function of history, [$s_t = f(h_t)$], and to be as useful for predicting the future as the whole history, it must have what is known as the *Markov property* [1, p. 465].

[1] Notation in section 2 is modified to be consistent with the rest of the paper. In this case, A_t becomes a_t and so on.

[2] In [1, p. 464], an assumption is made that "the reward is a direct, known function of the observation". This paper does not make this assumption, therefore the history equation includes rewards.

2.4. State Update Functions (SUFs)

A general way of mapping an agent's history into a usable state is through a state update function (SUF). The SUF is described as:

> [$s_{t+1} = u(s_t, a_t, o_{t+1})$], for all $t \geq 0$, ... The function u is called the *state-update* function. ... The state-update function is a central part of any agent architecture that handles partial observability. It must be efficiently computable, as no actions or predictions can be made until the state is available [1, p. 466].

Although this description is promising, it does not provide implementation details.

It is important to note that the state that is output by the state update function is different from the environment state. In this paper, *state* will refer to the state that the is given to the agent by the SUF and *hidden state* or *underlying state* refers to the environment state.

2.5. k-th Order State Update Functions (KO-SUFs)

k-th order state update functions (KO-SUFs) are one way to implement a SUF. KO-SUFs

> [Use] the last k [observations, actions, and rewards], [$s_t = (o_t, a_{t-1} r_{t-1}, \ldots, r_{t-k})$], for some $k \geq 1$, which can be achieved by a state-update function that just shifts the new data in and the oldest data out. This *kth order history* approach is still very simple, but can greatly increase the agent's capabilities compared to trying to use the single immediate observation directly as the state [1, p. 468].

However, KO-SUFs also have a significant disadvantage:

> [KO-SUFs] are often undesirable. When [k] is more than needed, they exponentially increase the number of agent internal states for which a policy must be stored and learned; when [k] is less than needed, the agent reverts to the disadvantages of undistinguished hidden state. Even if the agent designer understands the task well enough to know its maximal memory requirements, the agent is at a disadvantage with constant [k] because, for most tasks, different amounts of memory are needed at different steps of the task [2, p. 8].

2.6. Recurrent Memory

Although not explicit, recurrent memory methods (such as Elman networks) can be used to implement SUFs:

> Recurrent neural networks, such as Elman networks ... provide a way to construct history features. ... the input layer of an Elman network is divided into two parts: the true input units and the *context units*. The context units hold feedback signals coming from the network state at a previous time step. The context units, which function as [memory], remember an aggregate of previous network states, and so the output of the network depends on the past as well as on the current input [3, p. 4].

3. Proposed Method

3.1. Optimizing a SUF

For simple POMDPs, the SUF can be implemented in a tabular form. Each $(s_{t-1}, a_{t-1}, r_{t-1}, o_t)$ tuple can be mapped to some s_t. Let the environment sets for the states, actions, rewards, and observations be denoted with S, A, R, O, respectively. A tabular SUF would have $|S| * |A| * |R| * |O|$ rows, each with $|S|$ options for each row. Therefore, there are $|S|^{|S|*|A|*|R|*|O|}$ possible SUFs.

Selecting some objective function to maximize (such as the start value), each possible tabular SUF could be iterated through, and the one with the highest objective function value could be discovered and selected.

This algorithm would only be feasible for POMDPs with very small state, action, reward, and observation sets.

3.2. Neural Network State Update Functions (NN-SUFs)

Instead of using a table, SUFs can be implemented as a neural network (NN-SUFs). The input vector to a NN-SUF is a concatenation of the vector representations of the current observation, previous state, previous reward, and previous action. The output vector is the current state vector. This NN-SUF can be described in a similar way to how the abstract SUF is described [1, p. 466]:

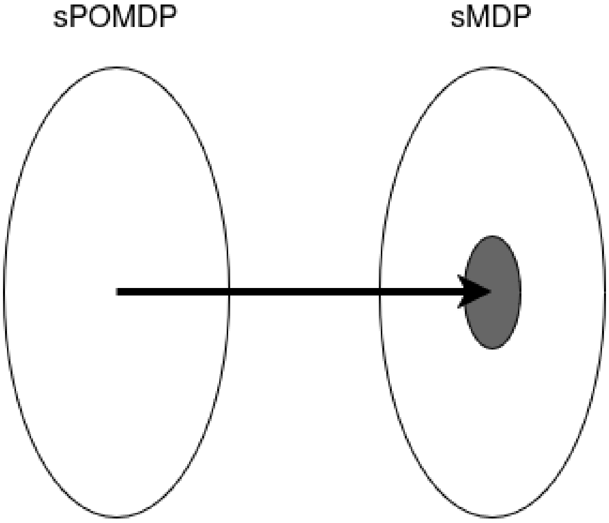
$$\vec{s_t} = u(\vec{s_{t-1}}, \vec{a_{t-1}}, \vec{r_{t-1}}, \vec{o_t}; \vec{\theta})$$

Where $\vec{\theta}$ are the parameters of the neural network.

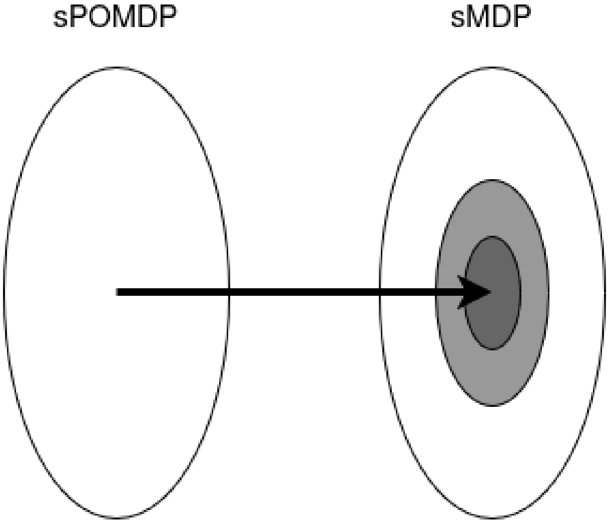
NN-SUFs are very similar to, but not the same as, Elman networks as described in section 2.6. NN-SUFs are normal feedforward neural networks that use the output (the state vector) as memory. Elman networks use a specialized structure.

The final layer of the NN-SUF has a $tanh$ activation function. This constrains the output vector components to the range $(-1, 1)$, which allows training to be stable.

Let $sPOMDP$ and $sMDP$ be the set of all POMDPs and MDPs, respectively. Any SUF will form a mapping from $sPOMDP$ to $sMDP$. By changing the NN-SUF's parameter values, the specific mapping changes. The set of all possible MDPs that a NN-SUF can map to through parameter changes is a subset of $sMDP$ (represented as the gray area in the figure):



An increase in the number of parameters (and therefore an increase in the possible combinations of parameters) can increase the number of possible mappings (represented by the lighter gray area):



The ratio of the lengths of a NN-SUF's input to output vector can be described as the compression ratio (CR). The CR describes how much of the history is removed from the agent state. A high CR suggests that a low amount of the history is available to the agent and a low CR suggests a high amount of the history is available to the agent. Stated mathematically, CR is defined as:

$$\text{Compression Ratio (CR)} = \frac{|\vec{s}| + |\vec{a}| + |\vec{o}| + 1}{|\vec{s}|}$$

Note that $|\vec{s}|$ refers to the length of the state vector, *not* the size of the state set, S. The same applies to the other variables in the equation. The 1 refers to the single component reserved to the reward signal.

Finding the best combination of parameters requires an optimization algorithm.

3.3. Optimizing a NN-SUF

Like any kind of optimization algorithm, an objective function, often denoted as $J(\theta)$, must be selected. In the context of reinforcement learning, there are many possibilities, but the most common for episodic environments is the start value, $V_\pi(s_0)$. This start value can either be calculated in a Monte Carlo manner or as a separate trained function $(V(s_0; \theta_\pi))$.

The variables to optimize over, θ, are simply the NN-SUF's parameters. In other words, optimizing a NN-SUF involves finding parameters that maximize the selected objective function.

Stated mathematically:

$$\theta^* = argmax(J(\theta))$$

A simple hill climbing algorithm is shown below:

Algorithm 1: NN-SUF Optimization (hill climbing)

```
Data: $\epsilon \geq 0, I \in \{0, 1, 2, 3, ...\}$
Result: $\vec{\theta^*}$
Let $\vec{\theta}$ be the NN-SUF parameters
$\vec{\theta^*} \leftarrow \vec{\theta}$
$J^* \leftarrow J(\vec{\theta})$
for $i$ = 0, 1, 2, ..., $I$ do
    Let $\vec{r}$ be a vector of length $|\vec{\theta}|$ with each component uniformly sampled in
      $[-\epsilon, \epsilon]$
    $\vec{\theta} \leftarrow \vec{\theta^*} + \vec{r}$
    Solve the environment using the NN-SUF with parameters $\vec{\theta}$
    if $J(\vec{\theta}) > J(\vec{\theta^*})$ then
        $J^* \leftarrow J(\vec{\theta})$
        $\vec{\theta^*} \leftarrow \vec{\theta}$
    end
end
return $\vec{\theta^*}$
```

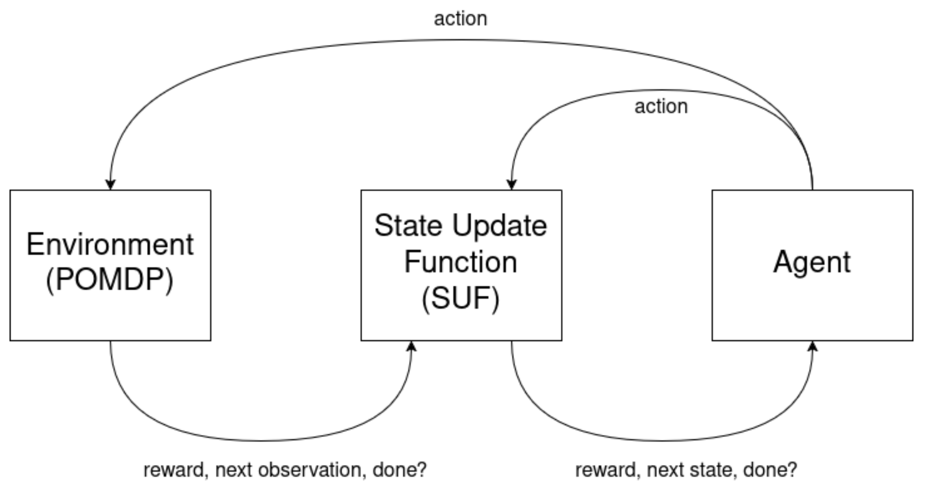
This algorithm can also be modified to optimize continuously by removing the loop limit, I, and not returning the optimal parameters, $\vec{\theta^*}$

3.4. SUF Implementation

Although not necessary for the proposed method of this paper, a brief note on the implementation of SUFs may be valuable.

From a software design perspective, it is often desirable to separate many of the components from each other and limit interactions to a strict interface. The primary reason for this is that each individual component can be replaced by another component and the software will still function as long as the interface is the same.

The SUF can interact with the environment and the agent in this way:


action
action
Environment (POMDP)
State Update Function (SUF)
Agent
reward, next observation, done?
reward, next state, done?

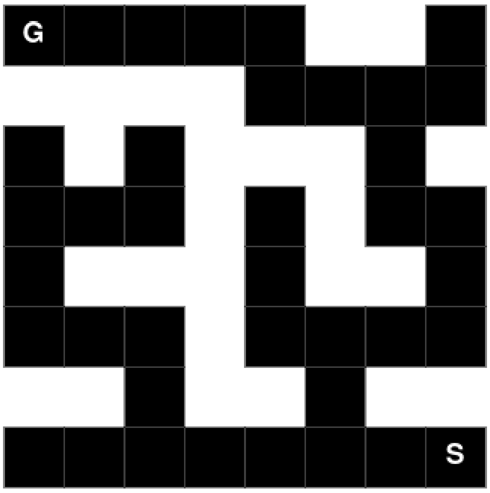
This interaction loop is similar to the MDP interaction loop. The primary difference is that the environment gives information to directly to the SUF rather than to the agent.

4. Experimental Setup

4.1. The `Maze` environment

Many of the common POMDP environments in the literature are extremely complicated to solve. For example, the ATARI environments included with OpenAI's gym package has visual observations that, when flattened, produce observation vectors that have thousands of components in them. These environments are simply not feasible for testing new methods on home computing hardware. Instead, a simpler POMDP environment is needed.

`Maze` is a custom POMDP environment that is intended to be fast, simple, and easy to debug. The overall structure of the environment is a maze in a 10 by 10 grid.



4.1.1. States

The basic state description only contains three pieces of information: the row of the agent, the column of the agent, and the current timestep. This basic state description makes this environment well suited for experiments as the agent must learn important information from a history of observations and actions that give incomplete information.

The state transition function uses the structure of the maze as described in section 4.1 to determine what the next state is.

At the start of an episode, the agent is placed at the location denoted with ”S”. The goal is located at the location denoted with ”G”.

4.1.2. Actions

The agent has four actions available: move up, down, left, or right. If the agent moves towards a wall, the location remains unchanged.

4.1.3. Rewards

The rewards in this environment are sparse. All rewards are equal to 0 except for when the agent moves to the goal location before the timestep limit is reached and the agent receives a reward of 1.

This sparse reward function makes solving the environment more difficult, but also reduces human bias in designing a reward function.

With a discount factor (γ) equal to 0.99, the optimal start value ($V_{\pi^*}(s_0)$) is equal to approximately 0.82.

4.1.4. Observations

The observations given by the environment to the agent are simply the descriptions of the grid spaces immediately above, below, left, and right of the agent. Each of these grid spaces could either be open, closed (wall), or the reward grid space.

Observations are implemented as a vector of length 12 (for 3 possible types of grid spaces multiplied by 4 nearby grid spaces). Each nearby grid space has 3 elements of the observation vector where the grid type is one hot encoded.

A single observation contains very little information about the hidden state, but the agent's history contains enough information to infer choose the optimal action. Therefore, a properly configured/trained SUF is required to solve the environment.

4.2. The MDP Algorithm

When solving the MDP produced by the state update function, the REINFORCE algorithm has been chosen. This algorithm is not often used because of the high variance [1, p. 329]. Although this is true, there are a few other properties of the algorithm that were beneficial.

First, the REINFORCE algorithm does not have a value function that needs to be trained. This eliminates at least two hyperparameters (the value function learning rate and the value function hidden layer size).

Second, the experiment can have more timesteps per second because there is only one neural network that has forward and backward passes (the policy function).

Third, the primary criticism that the REINFORCE algorithm has a high variance appears to be overstated when looking at the experiment results in section 5.

The pseudocode for this algorithm is below:

```
Algorithm 2: REINFORCE
  Data: Learning rate $\alpha > 0$, Number of episodes $E \in \{0, 1, 2, 3, ...\}$
  Result: $\pi(a|s; \vec{\theta})$. Policy optimality dependent on input data
  Let $\vec{\theta}$ be the policy parameters.
  for $e = 0, 1, ..., E$ do
      Generate an episode $s_0, a_0, r_0, ...s_T, a_T, r_T$, following $\pi(a|s; \vec{\theta})$
      $\vec{temp} \leftarrow \vec{0}$
      $G \leftarrow 0$
      for $t = T, T - 1, ..., 0$ do
          $G \leftarrow G + r_t$
          $\vec{temp} \leftarrow \vec{temp} + \alpha\gamma^t G \nabla ln\pi(a_t|s_t; \vec{\theta})$
      end
      $\vec{\theta} \leftarrow \vec{\theta} + \vec{temp}$
  end
  return $\pi(a|s; \vec{\theta})$
```

Note that this pseudocode is not the same pseudocode in [1, p. 328]. This is because the pseudocode in the book is most likely incorrect (see Appendix B for an explanation).

4.3. Measuring Performance

Measuring the performance of this method will be done in two ways: The start value at the end of training and learning efficiency (LE).

Since the primary goal of reinforcement learning is to find a policy that maximizes the associated value function at every state, the start value at the end of training is a simple measurement of performance. This start value is collected in a Monte Carlo manner and is denoted as *MCSV*.

Although finding the optimal policy is important, reinforcement learning tends to be very slow. To consider the time to find a good policy, the change in start value over training is divided by the wall time elapsed in seconds. This is defined as the learning efficiency:

$$\text{Learning efficiency (LE)} = \frac{\text{End MCSV - Start MCSV}}{\text{Time elapsed (s)}}$$

Where *End MCSV* and *Start MCSV* are the MCSV calculated at the end and start of training, respectively.

This fraction expresses the average increase in the start value for every second in training.

The end MCSV is the primary performance measurement. However, if two methods produce the same end MCSV, the one with the greatest learning efficiency will be considered better. In other words, the learning efficiency is a tiebreaker.

4.4. Experiment 1: State Size and Performance

This method is somewhat unique to reinforcement learning because the state vector size is an experimental parameter, rather than an immutable characteristic of the environment. This experiment seeks to determine the relationship between the state vector size and performance.

The basic intuition is that as state size increases, more useful information can be stored in any state. More useful information available to the agent results in policies with higher discounted returns. Therefore, there should be a positive relationship between state size and start value. To test this prediction, state vector sizes in the set $\{1, 2, 3, 4, 5\}$ will be evaluated for performance. ϵ is set to 10^{-3} in this experiment. Each iteration of the NN-SUF optimization algorithm has the agent train for 10,000,000 timesteps.

4.5. Experiment 2: Epsilon and Performance

The epsilon parameter of the converter is like a learning rate in a gradient ascent/descent algorithm. An epsilon that is set too high may overshoot local improvements, leading to poor performance. A low epsilon may easily find local improvements, but these improvements are likely to be small, also leading to poor performance. Therefore, the relationship between epsilon and performance should have one maxima. Outside of this maxima, the performance should decline.

To test this prediction, epsilon values in $\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$ will be evaluated for performance.

The state vector size is set to 3 in this experiment. Each iteration of the NN-SUF optimization algorithm has the agent train for 10,000,000 timesteps.

4.6. Experiment 3: Comparison with KO-SUFs

In addition to seeing how hyperparameters effect the performance of NN-SUFs, a comparison with KO-SUFs will be performed.

In this experiment, the best NN-SUF performance results will be compared with observation stacking with $k \in \{1, 2, 3\}$.

For the maze environment that is used in this paper, it is noteworthy to consider the extreme case where k is equal to the timestep limit. In this case, the entire history is known, and the agent can easily infer important information about the underlying state from this agent state. However, this is not feasible, since the agent state would be a vector with over a thousand components in length. This long of a vector would slow down computation speed and be inefficient.

The prediction for the results of this experiment is that the k has a positive relationship with end MCSV. This is because as k increases, more useful information is available to the agent, which helps the agent choose better actions. This positive relationship may be weak, as even the increase of k from 1 to 2 may not provide much more useful information to the agent at all.

Additionally, it is predicted that NN-SUFs will outperform all the KO-SUFs. This is because the neural network state update function has many parameters that can be adjusted to obtain good results, while the KO-SUFs only have one parameter, k that is actually a hyperparameter that is not trained during the experiment.

Each run of this experiment will have the agent train for 150,000,000 timesteps.

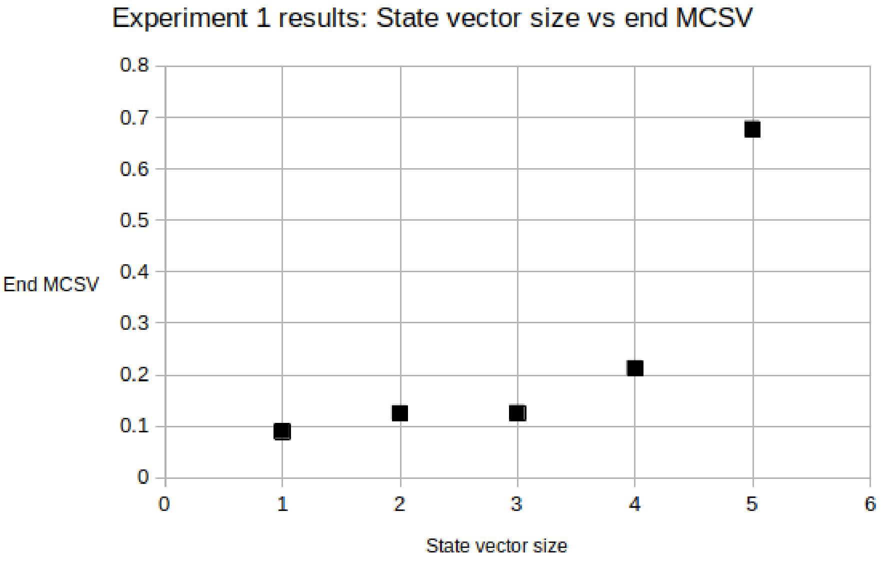
5. Results and Discussion

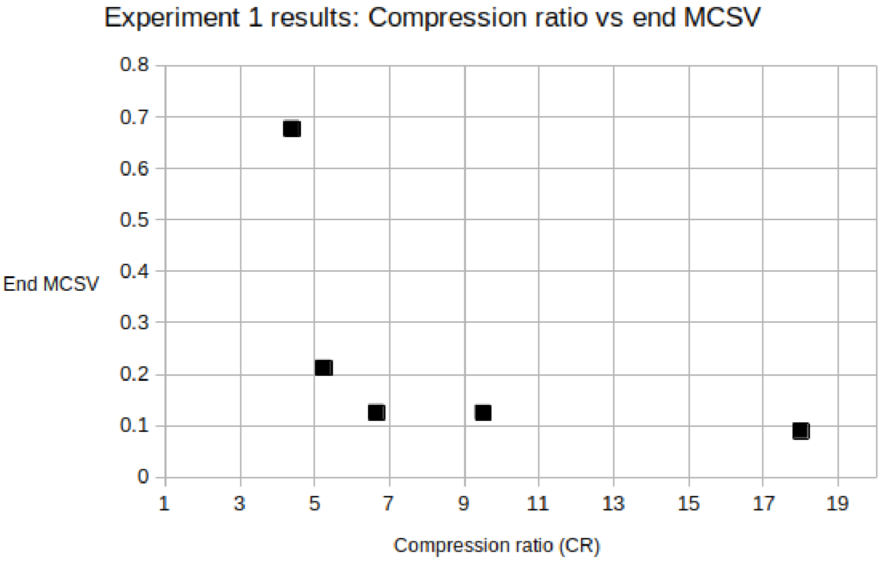
5.1. Experiment 1 Results

Experiment results as a table:

$\lvert\vec{s}\rvert$	CR	End MCSV	Learning efficiency
1	18	0.0899	$1.4396 * 10^{-6}$
2	9.5	0.1253	$2.2496 * 10^{-6}$
3	6.66...	0.1257	$2.1862 * 10^{-6}$
4	5.25	0.2125	$3.888 * 10^{-6}$
5	4.4	0.6769	$1.3165 * 10^{-5}$

Experiment results as scatter plots:





These results were expected: there is a positive relationship between state vector size and end MCSV.

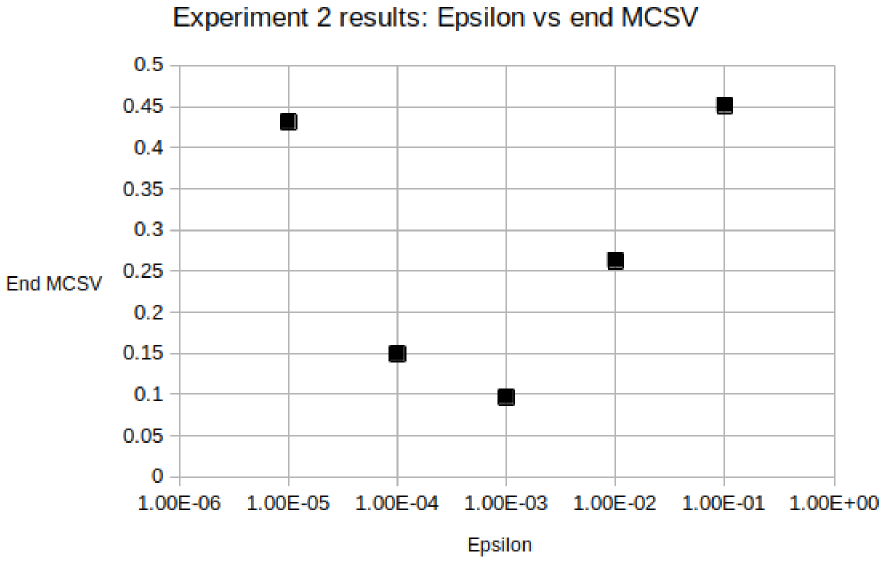
The most unexpected result from this experiment is the large increase in performance between state vector sizes of 4 and 5. It is not clear whether this large increase is due to random chance or accurately shows an unusual part of the relationship between the state vector size and the end MCSV.

5.2. Experiment 2 Results

Experiment results as a table:

ϵ	End MCSV	Learning efficiency
10^{-5}	0.4313	$8.6242 * 10^{-6}$
10^{-4}	0.1493	$2.627 * 10^{-6}$
10^{-3}	0.097	$1.5142 * 10^{-6}$
10^{-2}	0.2628	$4.9762 * 10^{-6}$
10^{-1}	0.4514	$9.0095 * 10^{-6}$

Experiment results as a scatter plot:



These results were unexpected. Rather than seeing a relationship with one maxima, these results show a relationship with one minima. It is not clear whether these unexpected results is due to random chance or actually shows an unusual part of the relationship between epsilon and the end MCSV.

If the results from the experiment are accurate, then there may be several maxima in the relationship.

5.3. Experiment 3 Results

Experiment results as a table:

SUF	End MCSV	Learning efficiency
Best NN-SUF	0.6769	$1.3165 * 10^{-5}$
KO-SUF ($k = 1$)	0.7451	$5.7487 * 10^{-5}$
KO-SUF ($k = 2$)	0.7775	$6.073 * 10^{-5}$
KO-SUF ($k = 3$)	0.7615	$5.8362 * 10^{-5}$

These results were unexpected.

First, the KO-SUFs show a very weak positive relationship between k and end MCSV rather than a strong positive relationship.

Second, it appears that even with a small amount of the most recent history (small k), the agent is able to infer enough information about the state to choose the optimal action. This low amount of information does not enable the agent to determine the current timestep of the MDP. However, for this environment, an optimal policy only needs to know the agent position (not the current timestep) to determine the optimal action.

Third, it is unexpected that all KO-SUFs have exceeded the performance of the best NN-SUF in this paper. The likely explanation for this is that the environment is too simple for NN-SUFs to surpass the performance of KO-SUFs. In other words, NN-SUFs are too complex of a solution for a simple problem.

6. Limitations

The optimization algorithm described in section 3.3 is the biggest limitation of this method. The algorithm is intended to only evaluate the objective function once the MDP produced by the NN-SUF has been solved by the agent. This results in a very slow optimization process.

Experiment 3 provides empirical evidence for how slow the NN-SUF optimization algorithm is in the environment. All the KO-SUFs outperformed the NN-SUF in terms of learning efficiency and MCSV

Additionally, Experiment 2 provides empirical evidence that the optimal ϵ hyperparameter is difficult to find.

7. Future Work

It is not clear from this paper how well a NN-SUF would compare with other POMDP techniques. Experiment 3 could be expanded to include other techniques, such as using RNNs and LSTMs. These experiments could also be expanded to include more challenging environments for evaluating performance. For example, the *Montezuma's Revenge* ATARI environment is extremely challenging to solve efficiently, even for state of the art techniques. However, a properly configured NN-SUF could achieve better performance compared to other methods.

A promising, but also difficult extension of the technique presented in this paper is to derive the gradient of the objective function with respect to the policy and NN-SUF parameters mathematically. Once this gradient is expressed in terms of o_{t+1}, r_t and the done signal, a model free policy gradient algorithm can be designed, which would make the optimization algorithm in section 3.3 obsolete. This new policy and NN-SUF optimization algorithm would probably learn much faster than the technique presented in this paper.

References

[1] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, second edition. Cambridge, MA: The MIT Press, 2018.

[2]R. A. McCallum, "Hidden state and reinforcement learning with instance-based state identification," in *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 3, pp. 464-473, June 1996

[3] L.J. Lin and T.M Mitchell, "Memory approaches to reinforcement learning in non-Markovian domains," *Carnegie-Mellon University. Department of Computer Science* (1992)

[4] Anonymous, "Update REINFORCE algorithm: step-wise or episode-wise?," October, 2019. [Online]. Available: https://www.reddit.com/r/reinforcementlearning/comments/don9ux/update_reinforce_algorithm_stepwise_or_episodewise/ [Accessed Jan. 27, 2025]

[5] M. Andrychowicz, et al. "What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study" *arXiv preprint* (2020).

Appendix

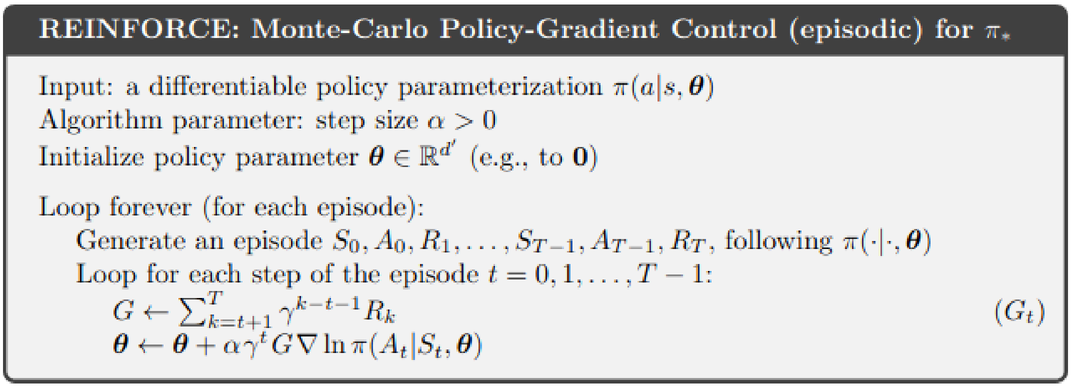
A. Common Hyperparameters

Hyperparameters that are common in all experiments are shown in the table below:

Hyperparameter	Value
Discount factor (γ)	0.99
Hidden layer size (policy)	24
Hidden layer size (NN-SUF)	30
Learning rate (policy)	10^{-4}
Number of NN-SUF optimization iterations	50
MCSV calculation samples (for calculating results)	10,000
MCSV calculation samples (for optimizing NN-SUF)	100
Start value threshold for terminating training	0.8
Hidden layer activation function (NN-SUF and policy)	tanh

B. Modifying the REINFORCE Algorithm

In [1, p. 328], the REINFORCE algorithm is shown:



```
REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for $\pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$                    $(G_t)$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$
```

There are two issues with this algorithm.

First, the line to calculate G_t is inefficient. At each timestep, it takes $O(T)$ time to calculate G_t where T is the number of timesteps in the episode sampled. Calculating G_t for all t will therefore take $O(T^2)$ time. Instead, if the loop variable t decrements rather than increments, most G_t can be calculated using the recurrent formula $G_t \leftarrow r_t + \gamma G_{t+1}$. This shortens the calculation of each G_t to $O(1)$ time and therefore shortens the calculation of all G_t to $O(T)$.

Second, the line which updates the parameters is incorrect. In this pseudocode, once the parameters are updated for the first time after an episode is complete, the collected episode most likely no longer matches what the updated policy would produce. Therefore, for the rest of the loop, updates to the policy parameters would likely not be an approximation of gradient ascent that the algorithm intends to accomplish.

Instead, at each timestep, the changes to the parameters should be added to a temporary variable. After all parameter changes have been added, then the temporary variable can be added to the policy parameters.

Both of these changes have been utilized into the pseudocode in section 4.2. Acknowledgments are due to [4] for confirming the error.

C. Dummy Values

A special case for SUFs exist on the first timestep in an episode where there are no previous states, actions, observations, or rewards. In the implementation of the code, dummy values are used for this special case.

For the NN-SUF, the dummy state vector is populated with -1, the dummy action vector is all 0, and the dummy reward component is set to 0.

For the KO-SUF, all dummy values are set to -1.

D. Weight Initialization

All weights in both the policy and NN-SUF neural networks use uniform Xavier Initialization. The only exception to this is that the last layer of the policy neural network uses uniform Xavier initialization with each value multiplied by 0.01 as recommended by [5, p. 5].

E. Running the Code

The code used in this project can be found at https://github.com/Jason-Whitmore/NN-SUF-HillClimbing. The main file is `Experiment.py`. This file contains the hyperparameters and their associated comments that describe what values to set them to. The experiment is run with from the command line with `python3 Experiment.py`

Acknowledgements are due to the creators of the `NumPy` and `gym` packages for their usefulness in the implementation of this method and the experiments that tested it.