ES 226R Problem Set 2

Andrew T. Sullivan

11 May 2020

1 Question 1: Reinforcement Learning of a Maze

1.1 Problem setup

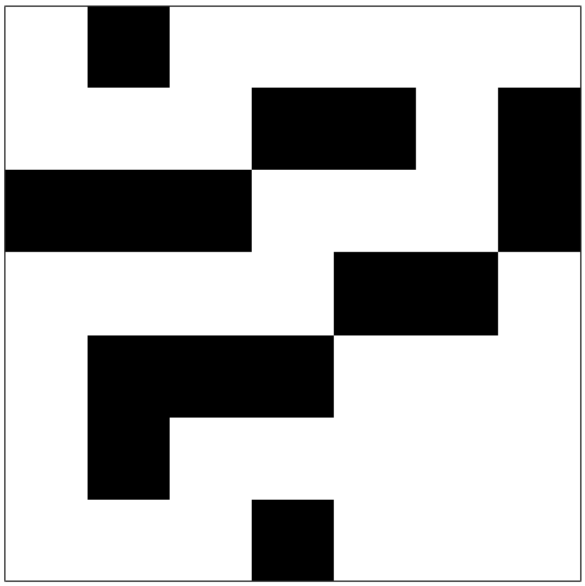


Figure 1: Maze Setup.

Reinforcement learning is the process by which an agent can learn to complete a task in which feedback is provided in the form of rewards (which could have positive or negative value). This is an intermediate between supervised learning, wherein the desired outcome is known and feedback is provided to the agent in the form of an error signal, and unsupervised learning, wherein no feedback is provided and the agent must extract patterns from the available data on its own. A short introduction to Markov Decision Processes (MDP) and the Temporal Difference (TD) learning method is provided below.

In this problem, we seek to train an agent to solve the maze shown in Figure 1. Black boxes represent forbidden states inaccessible to the agent, and white states are accessible. The agent begins in the top-left corner and its objective is to reach the goal state in the lower-right corner in as few steps as possible. In order to accomplish this, we view the maze learning problem as a Markov Decision Process, $M = (S, A, P, R, \gamma)$, where the entire problem is characterized by the available states, S, actions available to the agent at each state, A, transition probability between states, P, rewards associated with each state or state-action pair, R, and discounting factor, γ. The Markov property states that $P[S_{t+1}|S_t] = P[S_{t+1}|S_1...S_t]$. In other words, the history of the agent is entirely described by the current state.

Reinforcement learning algorithms seek to maximize expected cumulative reward. An RL agent can be described by a policy, value function, or model, or some combination of the three. A policy describes the agent's behavior function beyond the simple probability matrix between states described earlier. Deterministic policies are a direct map from state to action, $a = \pi(s)$. The value function captures how "good" the agent believes each state or state-action pair to be, in terms of predicted future reward:

$$V(s) = E[R_{t+1} + R_{t+2} + ...|S_t = s] \tag{1}$$

A model describes the agent's representation of the task environment and predicts what the environment will do next. Here, we do not consider model-based RL. An important factor in each RL algorithm is the balance between exploration and exploitation, i.e. how much the agent explores previously unexplored states to improve its knowledge of the environment vs. how much the agent utilizes its current estimate of the value function to maximize reward. This is explored below in terms of the maze.

The mathematical description of a MDP relies on the definition of the return, $G_t = \Sigma_{k=0}^{\infty}\gamma^k R_{t+k+1}$, i.e. the total discounted future reward, where a discounting factor γ is applied to future rewards to account for uncertainty, the preference for immediate reward, and some mathematical simplicity. The value function, then, is defined by $V(s) = E[G_t|S_t = s]$. Decomposing the return into the above definition and simplifying yields the famous Bellman equation:

$$V(s) = E[R_{t+1} + \gamma V(S_t + 1)|S_t = s] \tag{2}$$

This can be solved directly for small matrices but below is solved iteratively using the temporal difference method. Analogous functional forms for the definition of the state-action value function, $Q_\pi(s, a)$, and for the Bellman expectation equation, are also present for MDPs. The optimal state value and state-action value functions are the maximum values over all policies and are sought by the agent.

The temporal difference method (TD) is a simple and popular training algorithm for MDPs. In short, the current estimate of the value function at the current state, $V(S_t)$, is updated by using the sum of the receieved reward and the discounted current estimate for the value of the next state as an approximation for the expected future reward (i.e. the target value). The value of the current state is then updated according to:

$$V(S_t) \leftarrow (1 - \alpha)V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1})) \tag{3}$$

α is the learning rate. This can be rearranged as:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{4}$$

where the latter term is the temporal difference error, δ_t. It is believed that some form of temporal difference learning may be present in physiological neural networks in the brain, with the error signal being provided by dopaminergic neurons. An analogous algorithm exists for updating the state-action value function. This algorithm is known as SARSA. In either case, an action is chosen from the current state which brings the agent to a new state with an associated reward. The current estimate of the state value or state-action value is then updated according to the observed reward and new state. Below we apply TD(0) learning to the maze MDP described above.

1.2 A description of the algorithm

The algorithm is an implementation of TD learning applied to the maze. The maze is described as binary values, where 1 corresponds to an available state, and 0 corresponds to a forbidden state. The reward is -1 at all states (and thus, all time steps), except the goal state. Thus, the longer the agent spends navigating the maze, the lower its final cumulative reward. The true value function associated with each state is then the number of steps needed to reach the goal directly. This value is 0 at the goal state and -26 at the start state. Two different learning algorithms are explored below, known as V and V_{ctrl}, where the latter is semi-directed and the former is random. These will be described in more detail later. V is initialized with all states equal to 0, while V_{ctrl} is initialized by multiplying the maze matrix by -1, so all accessible states have a value of -1 and all inaccessible states have a value of 0. This is done due to the nature of the directed exploration described later. Three N parameters are used in the algorithm: N specifies the maximum number of iterations allowed for a given training episode; N_{trial} specifies the number of episodes used for training; and N_{max} is analogous to N for the test run of the maze after training. The learning rate, α is set to 0.2, and the discounting factor, γ, is set to 0.9. The importance of the hyperparameter values are explored in the Results section. Training is accomplished by the TD and TDcontrol functions, which implement random and semi-directed TD learning as described above. Inputs are the maze, the reward matrix, the current estimate of the value function, the parameters described above, and the starting and goal locations.

Each function begins by calling two other functions. MazeTransitions returns an n x m x 4 matrix, where i x j specifies the current state, and k specifies the intendent movement (N, E, S, W). A 0 indicates the movement is forbidden, and a 1 indicates it is permitted. Forbidden states include those with a 0 in the maze matrix, as well as movements outside of the borders of the maze. This matrix is then fed into the MazeProb function, which returns a normalized matrix such that the sum along the k direction is 1 for all permitted states and 0 for all forbidden states. Each entry for a permitted state-action has equal probability, e.g. if in an allowed state with three allowed movements to new states, the probability of each is $\frac{1}{3}$. The TD function then loops through for up to N iterations, extracting the 1 x 1 x 4 matrix of transition probabilities, and dividing it through by the minimum non-zero probability to find the smallest integer representation of each entry. Since probabilities are equal for this implementation, this is trivial. The current location of the agent is then optionally plotted. A vector is constructed with 1-4 each repeated the

corresponding number of times in the transition matrix, e.g. if the transition matrix was, hypothetically, [0 0.3333 0.6666 0], it will be converted to [0 1 2 0]. The vector [2 3 3] will then be constructed to represent the available transitions in their correct proportions. A random integer from 1 to the size of this vector is generated, and the corresponding entry defines the new state. The value of the current state is then updated according to the TD error and the learning rate. If the new state is equal to the goal state, the operation is terminated. This training function is iteratively called for N_{trial} times, passing the current estimate of the value function matrix in and storing the new estimate for the next iteration. The average difference between the new and old states is then stored as an estimate for the amount of learning that occurred. It should also be noted that the value of all forbidden states are updated at the end of each iteration by finding the current minimum value in the matrix and subtracting 1. This prevents the agent from accessing the forbidden states during the test run of the maze.

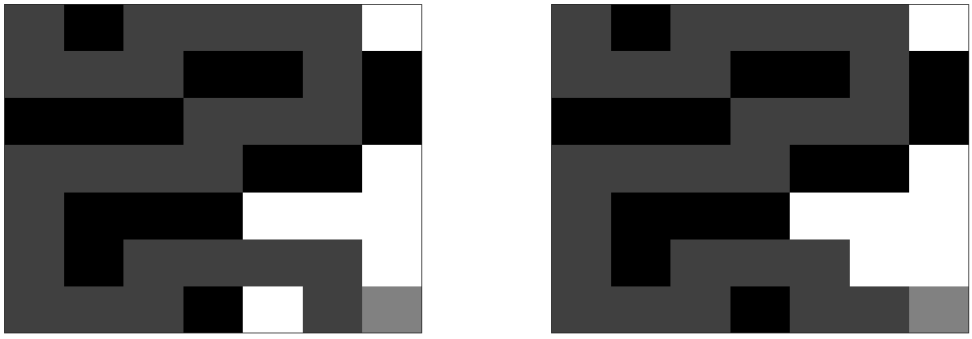
After training, the RunMaze function is called, passing in the maze, final value function estimate, start and goal locations, and the maximum number of iterations allowed. If this number is surpassed before the agent reaches the goal, a 0 is returned, indicating a failure. Otherwise, a 1 is returned, indicating a success. For each iteration, a greedy algorithm is employed, i.e. the agent chooses the next state which has the maximum value function from the available transitions to nearest neighbors. All the visited states are shown as gray boxes, and are a lighter gray if the state was visited more than once (or is the ending state).

The TDcontrol function performs similar computations to the TD function, with one important distinction. In this case, we exploit the current knowledge of the value function to increase the probability that the agent chooses a given next state if it has a higher value. As before, the transition probability matrix is constructed, and is extracted as a 1 x 4 vector for the given current state on each iteration. The valid neighbors are identified (i.e. those that do not correspond to states outside of the borders of the maze), and their corresponding value estimates are obtained. If all the values are 0, the probability transition matrix alone is used as in the random TD case. Otherwise, the TransitionScale function is called, which takes in the transitions vector, the valid transitions, and the values associated with the new states. If only one transition probability is nonzero, the function does not do anything and returns the transitions as-is, since this indicates only one transition is allowed anyway. Otherwise, the transitions are scaled by the following expression:

$$P_i \leftarrow \frac{P_i \Sigma_{j \neq i} V_j}{\Sigma_i P_i \Sigma_{j \neq i} V_j} \tag{5}$$

This ensures that the states with the largest (i.e. least negative) values have the highest probability of being chosen. This new transition vector is then converted to integers by dividing through by the smallest nonzero value as before, but a ceil function is included to ensure the resultant values are integers. The final transition vector is then returned, a random number is generated, and the corresponding value is chosen as in the random TD function. The state and value function are then updated.

1.3 Results



(a) Trained with Random Movement. (b) Trained with Semi-directed Movement.

Figure 2: Trained Agents Navigating Maze.

Figure 2 shows example maze runs for agents trained with random movement and semi-directed movement. Up to 1000 iterations per training event were permitted, with the event terminating if the agent reached the goal beforehand. 100 training episodes were performed for each agent, and the agent was given up to 1000 iterations to reach the goal during the test. White squares are locations that were not visited by the agent; black squares are forbidden states; dark gray squares are locations visited by the agent; the light gray square is the goal. If the agent had visited a state more than once, it would also appear as light gray. Thus, we can see from the above that both random and semi-directed training were successful in that the agent goes directly to the goal with as few steps as possible. As a proxy for learning, the average change in values between each trial number was recorded for random

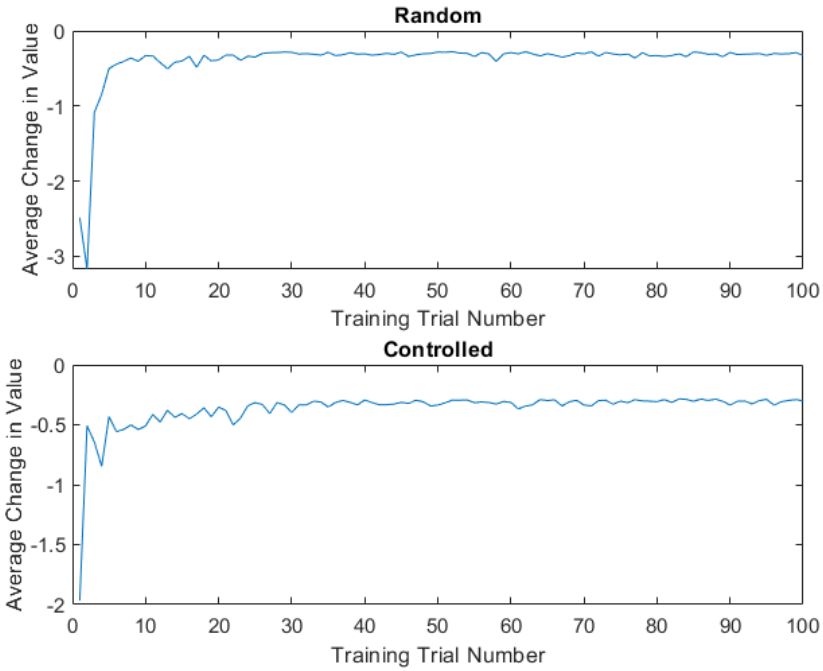
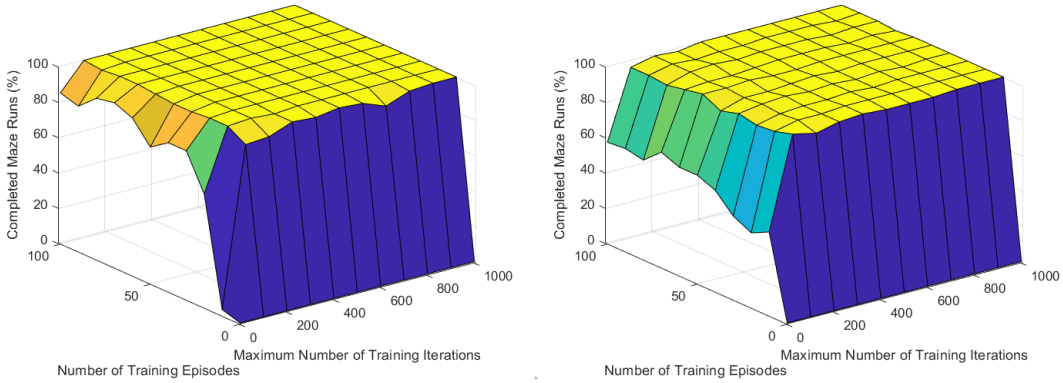


Figure 3: Change in Values over Training.

and semi-directed learning over 100 iterations, as shown in Figure 3. In either case, the agent learns quite quickly using this estimate, with average absolute values below 1 after 5 and 2 iterations for the random and semi-directed learning, respectively. Of course, this is merely an average, so it is possible that positive and negative changes in value would cancel out, but we do not expect any positive changes to value, since all rewards are non-positive.



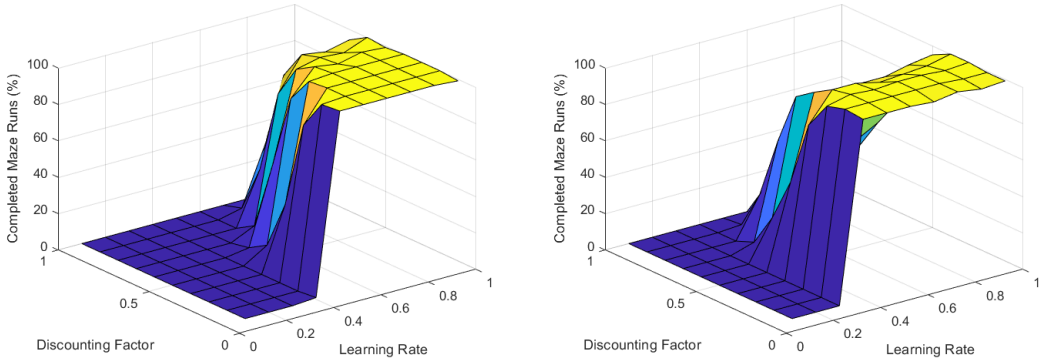
(a) Trained with Random Movement.

(b) Trained with Semi-directed Movement.

Figure 4: Effect of Training Parameters on Number of Completed Test Runs.

Figure 4 explores the effect of different training parameters on the number of completed test runs (out of 100). For each combination of the maximum number of iterations permitted per training episode and the total number

of training episodes, 100 tests were performed, where each test consisted of a retraining of the values and recording if the agent successfully reached the goal in the specified number of iterations (1000). Failures occurred when the value for a given state, typically near the beginning of the maze, was slightly higher (less negative) than that of the next state, causing the agent to simply jump back and forth between these two states forever. This likely occurred if the agent did not visit these states enough during training. This procedure was performed for both the random and semi-directed algorithms. The results are qualitatively similar: as long as approximately 10-20 runs were included for training and the agent was given at least 50 iterations to reach the goal, either training method functioned with an accuracy above 90 percent. Interestingly, the surface for the semi-directed learning appears less consistently successful than the random case. Aside from a potential error in the code, there is a possible reason for this based on the size of the problem. Random movement is a useful method for ensuring that the state space is explored completely, while semi-directed learning does not guarantee such complete exploration. For small state spaces, it is therefore possible that random motion is preferable. For large, high-dimensional spaces, however, it would take an unrealistic number of training iterations to completely explore the state space. It is in these scenarios where semi-directed learning is more suitable. Of course, it is also worth noting that only one semi-directed movement algorithm is employed here. A wide variety of such algorithsm exist, including the commonly employed ϵ-greedy algorithm, to balance exploration and exploitation. Variations of these algorithms also exist which change the relative amounts of these two factors over time, favoring exploration of the state space early and exploitation of known values later.



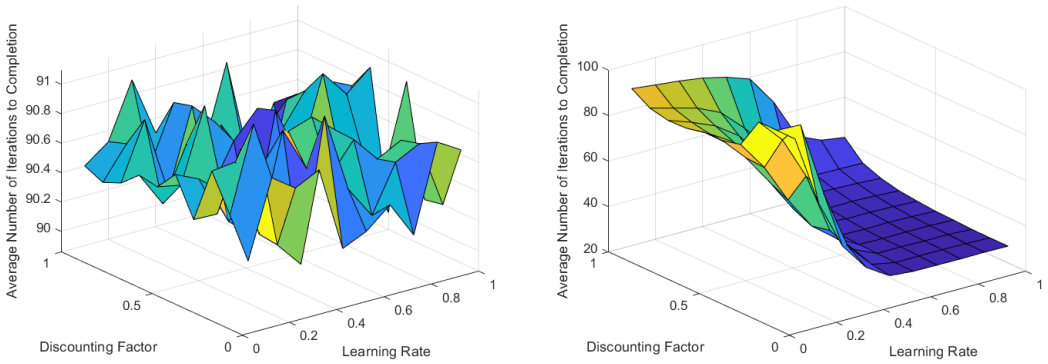
(a) Trained with Random Movement.

(b) Trained with Semi-directed Movement.

Figure 5: Effect of Hyperparameters on Number of Completed Test Runs.

As a final investigation of this RL problem, we explore the influence of the learning parameters α and γ, the learning rate and discounting factor, respectively. Figure 5 shows the results from an analogous testing procedure to that used above for different combinations of values for these two parameters on number of successful test runs after training. Again, both algorithms perform similarly. However, these results are somewhat unexpected, as they indicate the algorithms perform more successfully for high learning rate values and low discounting factors. Commonly, the reverse trend is employed during training, favoring slow training and relatively large discounting factors to prevent overshoot and increase the influence of expected future reward. It is again possible that these somewhat contradictory results again relate to the nature of this problem. Given that there are no positive rewards in the maze, the influence of expected future reward is likely less important than if the goal had a large positive reward value, for example, favoring a myopic form of learning. It is possible, given the number of training iterations allowed, that small learning rates did not give the agent sufficient time to update its value function effectively over the duration of the permitted trial period, favoring larger learning rates.

In order to explore this latter point, the average number of iterations to reach the goal location for each training episode were also recorded, and are plotted in Figure 6. The randomly trained algorithm shows no discernable pattern, indicating that these parameters do not influence how quickly the algorithm reaches the goal during training. This is expected, since the current value estimates have no role in directing training. For the semi-directed case, however, the same combination of learning rate and discounting factor which showed high success rate shows low numbers of iterations to reach the goal. Higher average numbers to reach the goal are found for combinations with low success rate in the previous experiment, suggesting that it is possible that, for low learning rates, not enough iterations were permitted for the agent to reach the goal. Given the semi-directed nature of exploration for this agent, it is also plausible that it was more likely to get stuck moving between two states for long periods of time, leading to some unsuccessful learning attempts. This can be remedied by altering the algorithm to favor exploration more than



(a) Trained with Random Movement.

(b) Trained with Semi-directed Movement.

Figure 6: Effect of Hyperparameters on Speed of Completion During Training.

exploitation, and possibly by increasing the number of permitted iterations during learning to allow the agent to escape these stuck states (these tests were run with 300 permitted iterations per training episode).

In conclusion, this simple RL maze algorithm suggests that both random and semi-directed learning can lead to successfully trained agents. For a small task such as this, fast learning rates, small discounting factors, and random exploration lead to the most successful learning. It is expected, however, that these trends will change significantly for larger state spaces and wider ranges of rewards available to the agent, favoring some degree of exploitation.

2 Question 2: The Mountain Car Problem

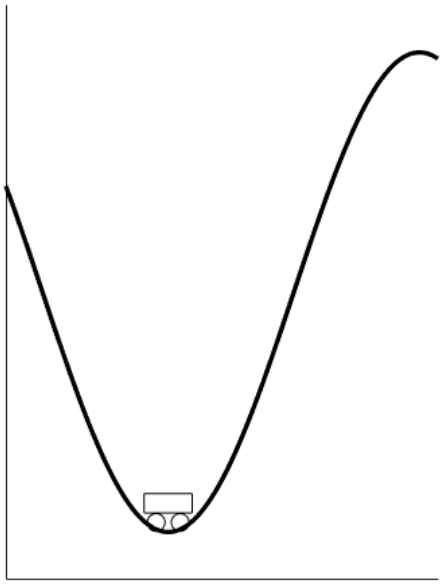


Figure 7: Mountain Car Setup.

2.1 Background

The mountain car problem is a classic example of a reinforcement learning task. A car is driving on a hill and gets stuck at the bottom (Figure 7). The car does not have enough accelerating power to simply drive up the hill, and thus must reverse up the previous hill and oscillate back and forth in order to obtain enough momentum to reach the top. The RL agent, in the form of the car, does not know the goal *a priori* and thus must learn this method. A number of different implementations exist to solve this problem, and it is a common teaching and testing method for new algorithms.

2.2 Implementation

The implementation for this problem follows the constraints imposed by Barto and Sutton [1]. In summary, the state space, **S**, is characterized by the position and velocity at each point, and there are three possible accelerations available to the car. We consider a sinusoidal hill profile with slope factor $s = 3$, i.e. $sin(3x)$. The state-action space is summarized as $-1.2 \leq x_t \leq 0.6$, $-0.07 \leq v_t \leq 0.07$, and $a_t = [-1, 0, 1]$. At each time point, the state is updated according to the update equations:

$$x_{t+1} = x_t + v_t \tag{6}$$

$$v_{t+1} = v_t + Aa_t + Bcos(sx_t) \tag{7}$$

Here, we use $A = 0.001$ and $B = -0.0025$ as in [1]. Though this example constrains the velocity to be set to 0 at the left edge of the valley, i.e. there is an inelastic wall at $x = -1.2$, results from the following implementation seemed to get stuck at this position once it was reached, not moving for the duration of the trial. Thus, we instead consider a slightly elastic wall by setting $v(x = -1.2) = 0.01$. Position was randomly initialized from the choices -0.6, -0.5, and -0.4. Velocity and acceleration were both set to zero. Given that the above state space is defined continuously, it is necessary to discretize it for the purpose of applying reinforcement learning update steps, as discussed next. For this purpose, we discretize the position and velocity in steps of 0.1 and 0.01, respectively. Note that the true values are not rounded since this leads to the car not moving from its current position, but reinforcement learning update points in the state-action space are adjusted discretely.

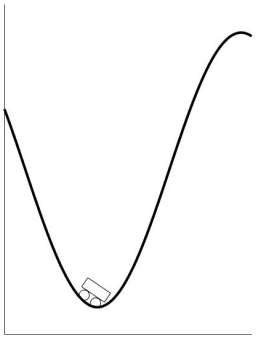
While the above maze problem employed a value update equation based on the Bellman equation, here we consider a full state-action value function, $Q(x, v, a)$, as the car must learn to make the correct acceleration decision based on its current position and velocity. For update purposes, we use the famous SARSA algorithm, where the current state-action value is updated after taking an action and observing the obtained reward, the next state, and the following action. This update is summarized by the following equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \tag{8}$$

As in the value equation for the maze example, α is the learning rate and γ is the discounting factor. Rewards are -1 for all x below the top of the hill, i.e. for all $x < 0.5$. Rewards above this point are 0. When updating the state-action value function, the current position and velocity are rounded to find the nearest value in the discretized grid. The final piece of the simulation is how the agent makes decisions during learning. For this, we apply an ϵ-greedy algorithm, a simple and ubiquitous algorithm known for its applicability to the multi-armed bandit problem. Briefly, a random number between 0 and 1 is generated. If the value is above the threshold $1 - \epsilon$, the current action is chosen which brings the agent to the state with the current highest value (i.e. chooses the maximal $Q(S, A)$ at the current state). If below this threshold, the action is randomly chosen from the available list. This encourages a balance of exploration and exploitation. For testing, a greedy algorithm is employed at each state, i.e. the agent chooses the state-action with the highest value.

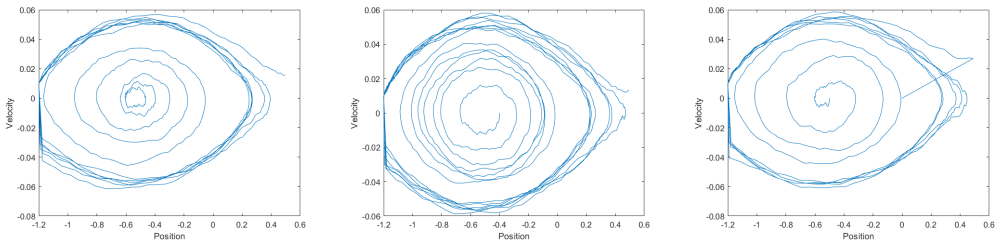
At the beginning of the simulation, the state-action value function is set to all zeros, and the position, velocity, and acceleration are initialized as above. For each of $Ntrial$ episodes, the agent has N iterations to reach the goal point by semi-randomly exploring the state space. The ϵ-greedy strategy is employed to select the next action, and the state is updated accordingly. The SARSA algorithm is then used to update the state-action value. If the new position is beyond the threshold set at 0.5, the episode is terminated.

2.3 Results



Animation 1: Successful Trial of Mountain Car Problem with Minimal Acceleration.

We perform simulations with a discounting factor of 0.9, learning rate of 0.15-0.2, ϵ of 0.10, and varying acceleration power. This latter determinant seems to be the main factor which sets the performance of the model, along with requirements for other parameters. Given the relatively large parameter space, $(N, Ntrial, \alpha, \gamma, A_{max}, A_{min})$, it is difficult to perform simulations which effectively identify the best parameters. We restrict ourselves to acceleration conditions where $A_{max} = -A_{min}$, and do not consider cars which can accelerate variably, i.e. the car can only fully accelerate forward, backward or not at all, as this would further expand the dimensionality of the state-action value matrix. While Barto and Sutton divided the state space into 8-by-8 grids, our division sets the final dimensionality to 19 x 15, approximately 4.5 times larger. As will be seen later, this necessitates much higher numbers of episodes and iterations per episode for sufficient training. Accelerations above values of approximately 10 are trivial as the car can get to the top of the hill without generating momentum by oscillating. For an acceleration of 5, training occurs relatively quickly, consistent with values found in Barto and Sutton. Reliable training can occur in as few as 200 episodes consisting of a couple of hundred iterations each. As accelerating power is decreased, however, these numbers increase exponentially. This is to be expected, however, given the nature of the problem. As some states are only accessible following the correct state-action sequences from other states, it becomes increasingly less likely that states at higher elevations (i.e. the goal state) will be randomly accessed as accelerating power is decreased. For acceleration power of 2, at least 1000 episodes with 15,000 iterations per episode are necessary to sufficiently



(a) Initial Episode. (b) Middle Episode. (c) Final Episode.

Figure 8: State Space Trajectories over Training.

characterize the state-action space. Below 2, the problem becomes even more challening. Figure 8 shows example trajectories for accelerating power of 1 at the first, middle, and last episode. The goal state was reached at iteration 993, 1260, and 827, respectively. It appears that the car learns to rapidly increase its speed by moving up as far as possible on either side as the hill as training progresses. It also appears that the car is utilizing the semi-elastic collision with the wall in all trajectories, making this slightly less realistic. Over the 6000 trials used to train this version of the algorithm, the ending position was reached on trial 1087 with a standard deviation of 570 and a maximum of 5690. This is in stark contrast to the case where maximal acceleration is 5, where the end is reached at iteration 305 on average with a standard deviation of 208, and a maximum of 1872, or an average of 179 with standard deviation of 145 for acceleration of 10. Interestingly, even with these large acceleration values, it still takes the car approximately 100-300 iterations to reach the top during the test. Figure 9 shows the trajectory during the

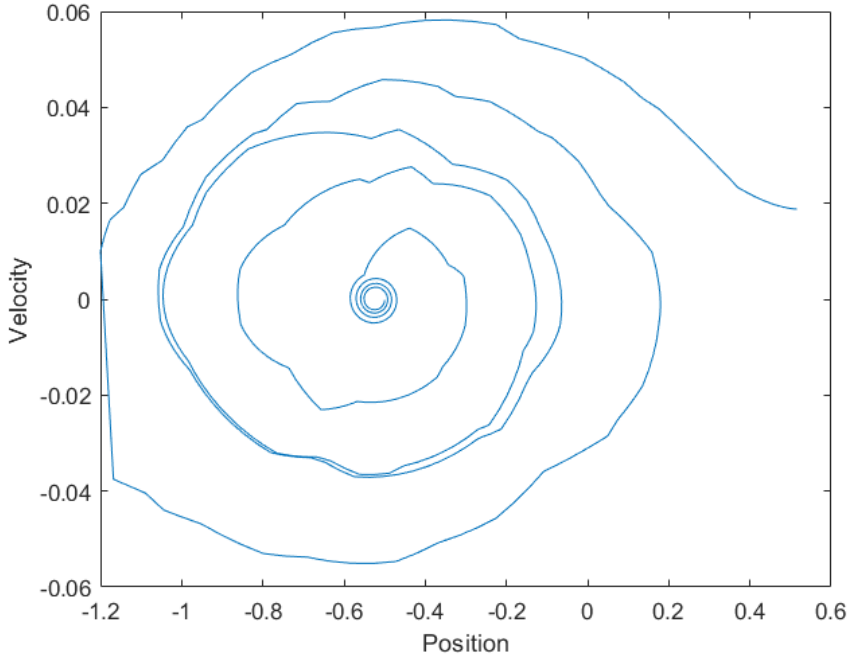


Figure 9: Trajectory Sampled During Test with Minimal Accelerating Power.

following test after training for the same training sequence as in Figure 8, where the agent uses a full greedy algorithm to maximize value at each state. This is the same test shown in Animation 1 (note: clicking on the animation should cause it to play when using a PDF viewer compatible with the LaTeX animate package, e.g. Adobe Reader or Foxit). Clearly, the agent has learned to oscillate to reach the peak of the hill in as few trials as possible. Here, it reaches the top after 1140 iterations.

Figure 10 shows the learned state values after training for the same simulation trajectory shown above. The state value is obtained from the state-action value by summing across all possible actions at each state. Areas around the circumference near the minimal velocity and near the maximal position have the least negative values, aside from

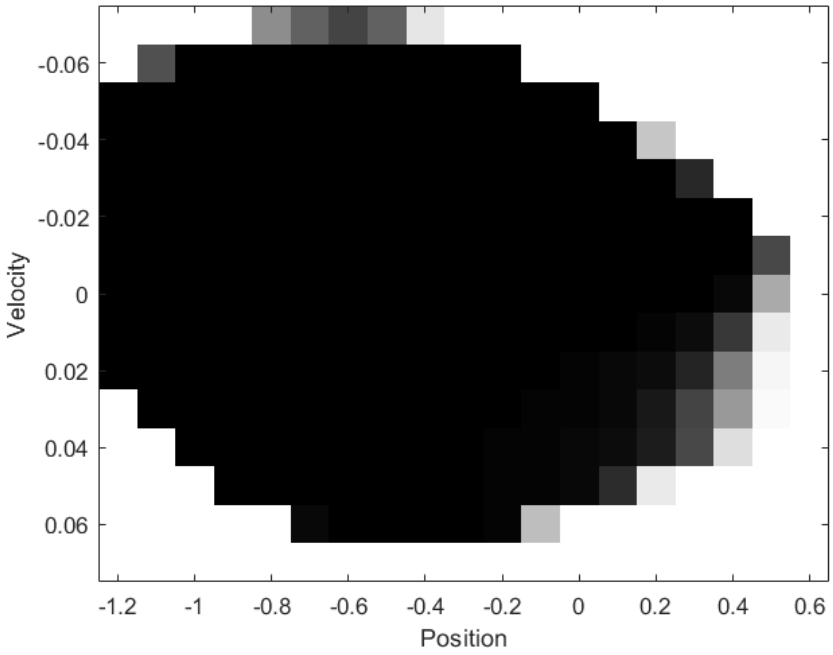


Figure 10: State-Action Value Function Learned Using SARSA.

those states which were not visited or where the goal was located, which have a value of 0. Unsurprisingly, the agent learns to oscillate to reach the circumference of this value function.

In conclusion, this SARSA RL algorithm can learn over many trials to reach the top of a hill by gaining momentum through oscillations for a variety of accelerating powers. There are a number of further experiments and improvements that can be made to this algorithm, however. The number of iterations required to reach the top of the hill is orders of magnitude larger than that shown in Barto and Sutton. One factor which likely contributes to this discrepancy is the higher dimensionality of the state space, as discussed previously due to the finer discretization. Another experiment which would be more easily performed with larger computational power is quantification of accuracy. While it is apparent that the algorithm can be successfully trained for multiple values of accelerating power, this is not always successful, and the algorithm must be re-trained. How frequently this occurs would be important in optimizing training, and specifically, for identifying proper hyperparameter values, which was not done for this algorithm. Smaller learning rate for larger number of iterations is a safer strategy to prevent the algorithm from getting stuck, at the expense of time. Discounting factor is also important, as the value of future states is essential in encouraging the agent to visit states near the left bound of position. Finally, the elasticity of this left bound is still problematic, and different parameter values should be investigated to ensure the agent does not get stuck at this position if its velocity is set to 0.

3 Question 3: Implementation of a Model for Memory Consolidation

3.1 Introduction

In 1994, Pablo Alvarez and Larry R. Squire published "Memory consolidation and the medial temporal lobe: a simple network model" in PNAS [2]. This paper looked at current evidence for the phenomenon of consolidation, in which a memory trace for a given event is converted to some form that is independent of the medial temporal lobe (MTL, including hippocampus, parahippocampal cortex, entorhinal cortex, and other structures). Evidence from retrograde amnesia studies suggest that older memories are less sensitive to MTL damage as memories are passed to the neocortex. The model proposed by the paper considers the MTL memory system as a temporary memory store which serves to link together the relevant higher-order cortical areas needed to process the sensory information associated with the memory. Concurrent activation of these areas serves to strengthen their connections via Hebbian learning, a common learning rule which is implemented neurophysiologically in the form of long-term potentiation of synapses. In short, when a synapse is activated and the postsynaptic cell then fires within some short time window, the strength of the synapse is increased. This is mediated molecularly by the influx of calcium ions through NMDA glutamate receptors. Calcium transduction pathways can then control trafficking of AMPA receptors to the synaptic membrane and gene transcription through the CREB transcription factor pathway for longer-term effects. Thus, over time, the connections between concurrently active neocortical structures will be strengthened. In order to prevent the rewriting of old memories, or storing of irrelevant information, the changes of neocortical-neocortical synaptic weights are viewed as being much slower than those of MTL-neocortical connections. This permits the rapid storage of memory in the MTL-neocortical connections, and the gradual consolidation of these memories to be entirely independent of MTL once neocortical-neocortical connections are strong enough. The MTL, however, has a limited memory storage capacity, while that of the neocortex is quite large.

3.2 Model Description

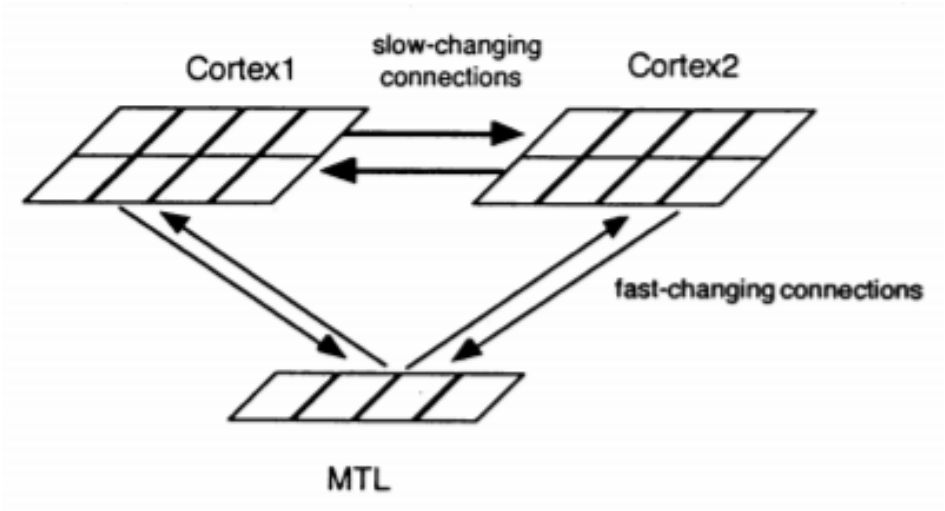


Figure 11: Model Setup from [2].

The model described above was implemented computationally in the Alvarez and Squire paper (Figure 11). To permit rapid training and testing, the number of neurons is quite small, but this merely served as an indication that the above model could provide evidence for phenomena observed in living systems. The model considers two neocortical areas (Cortex1 and Cortex2), each consisting of 8 neurons. The MTL consists of four neurons. All neurons are organized into non-overlapping groups of four. Thus, each cortical area has two groups, and the MTL has one, consistent with the smaller capacity of the MTL. There are reciprocal connections between all neurons in different areas (i.e. all neurons are reciprocally connected between Cortex1 and Cortex2, between Cortex1 and MTL, and between Cortex2 and MTL), but no explicit connections exist within a given area. Neurons are represented by their activity, a, which is updated at each time step according to:

$$a_i = \delta a_i + \Sigma a_j w_{i,j} + \epsilon \tag{9}$$

δ serves as a decay term, $w_{i,j}$ as the synaptic weight connecting neuron j to neuron i, and ϵ is a uniformly distributed noise term between -0.05 and 0.05. All activites and weights are constrained between 0 and 1. All activities are updated synchronously in all neocortical areas, and are then updated synchronously in the MTL neurons. Winner-take-all lateral inhibition is also present within each group of four, so during the update step, only the neuron with the largest activity is left active, and the activity of the other three is set to 0. Weights are updated after activities based on a modified Hebbian learning rule and exponential forgetting:

$$\Delta w_{i,j} = \lambda a_i (a_j - \bar{a}) - \rho w_{i,j} \tag{10}$$

Here, λ is the learning rate, ρ is the forgetting rate, and $\bar{a}$ is the mean activation level of all units projecting to unit i. The learning and forgetting rates for the neocortical-neocortical connections are orders of magnitude smaller than the MTL for consistency with the above description.

The task of the network was to reconstruct a pattern of activity, i.e. a memory, after an incomplete cue was provided. A memory pattern was defined as two active units in each cortical area (one per group of four). Two orthogonal activity patterns were used to train the model. Each pattern was presented twice to the model by setting the activity of the four cortical neurons to 1. Activations are then updated in cortex and MTL and connection strengths are then updated for each time step. Three time steps per presentation were run. After training, the network was tested by presenting half of the pattern (setting the two neurons to 1 in one of the cortical areas and not the other), and allowing activity to cycle for three time steps without weight updates. The final pattern of activity was then compared in the cortical areas to the correct pattern and the sum of squared errors was calculated. The total error was defined as the sum of these errors for all four possible half-patterns, presented in random order.

To account for consolidation, which takes place over much longer time scales than the three cycles used above, random activity is generated in the MTL by setting one of the four units to 1 and allowing activity to cycle for three time steps. This is repeated for a specified number of consolidation time steps before the test half-patterns are presented. Lesioning is modeled by inactivating the MTL, i.e. by setting all its synaptic weights projecting to and from other neurons to 0.

3.3 Implementation

Learning and forgetting parameters are initialized using the values specified in the publication: $\lambda_{MTL-Cor} = 0.1$, $\lambda_{Cor-Cor} = 0.002$, $\rho_{MTL-Cor} = 0.04$, and $\rho_{Cor-Cor} = 0.0008$. Consistent with the model, these rate constants are higher for MTL-neocortex connections to permit more rapid learning. The activity decay parameter $\delta = 0.7$. The InitializeWeights function randomly initializes a 20 x 20 matrix of weights, where $w_{i,j}$ denotes the weight connecting neuron j to neuron i. Neurons 1-8 are in the first cortical section, neurons 9-16 in the second, and neurons 17-20 are in the MTL. The function also utilizes this information to set all self-connections to zero (i.e. the connections between all neurons in the same area). The identity of these connections are stored and returned in a separate 20 x 20 matrix, $wfixed$, which contains a 1 if the neurons are connected and a 0 otherwise. Each simulation consists of learning and test phases. Two non-overlapping patterns are randomly initialized consisting of the activity of one neuron being set to one in each layer of each cortical area (Figure 12). A random permutation is used to find the order of pattern presentation, with each pattern being presented twice. An example pattern is displayed below, with neurons 4, 5, 9, and 16 active, and all others inactive.

For each pattern presentation, the activity of the active neurons is set to 1, and all others are set to 0. This activity cycles through the network for three time steps, and the weights are updated after each. The UpdateActivity function updates activity in all cortical areas synchronously, as specified in the paper, by using linear algebra:

$$\mathbf{a_{1:16}} \leftarrow \delta \mathbf{a_{1:16}} + \mathbf{a_{1:20}} \mathbf{w^T_{1:16,1:20}} + \epsilon \tag{11}$$

Here, the activites in the cortical neurons are updated by decaying the current values and adding on a 1 x 16 vector of random noise selected from -0.05 to 0.05 as specified in the paper. Also added on is the matrix product of the activites of all neurons and the matrix connecting those neurons to the cortical neurons. This expression is equivalent to $(\mathbf{wa^T})^\mathbf{T}$ by matrix algebra. This term will produce the sum of the activies of all neurons multiplied by their weights connecting to cortical neuron i. Since all weights connecting neurons within a given section of cortex are 0, the only nonzero terms connect neurons from other areas to the cortical area of interest. Lateral inhibition is then applied by taking each group of four neurons, finding the neuron with the highest activity, and setting all others within that layer to zero. Activities are then constrained to be between 0 and 1, as required. The procedure is then repeated for the MTL neurons, based on the updated cortical activities.

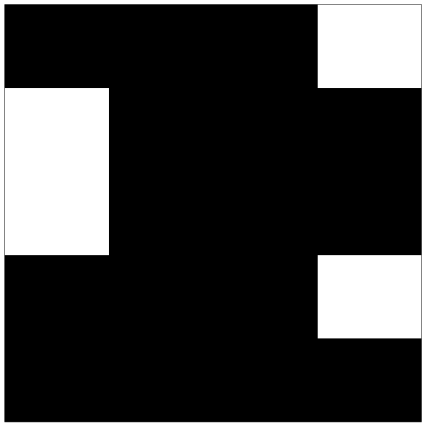


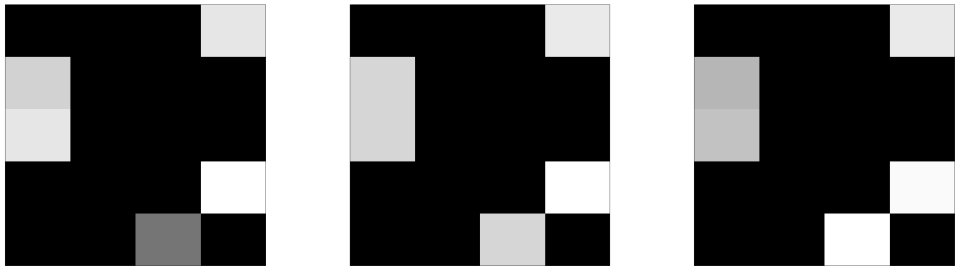
Figure 12: Example Pattern for Training and Testing.

The UpdateWeight function is then used to update the 20 x 20 weight matrix based on the activities just calculated. This is done layer-by-layer. For each cortical layer, $wfixed_{i:i+3,:}$ is extracted, i.e. the fixed connection vector of all connections to the layer of interest. The activity of the neurons of interest are extracted, and the $\bar{a}$ is calculated as the average activity of all neurons connecting to the layer, i.e. the average activity of all neurons with an entry of 1 in the connection vector. It should be noted that this average value is the same for all of the neurons in the cortical area, since all neurons in a given area receive projections from the other two areas. The difference of the activity of each neuron and the average activity is then calculated, and is multiplied by the connection vector so it is zero for neurons which do not connect to the layer of interest. Two row vectors are constructed with the correct λ and ρ parameters, i.e. sixteen for the corticocortical connections and four for the MTL-cortex connections. The weights projecting to the layer of interest are updated as:

$$\mathbf{w_{i:i+3,:}} \leftarrow \mathbf{w_{i:i+3,:}} + \lambda \circ \mathbf{a^T_{i:i+3}} \circ (\mathbf{wfixed_{i:i+3,:}} \circ (\mathbf{a} - \bar{a})) - \rho \circ \mathbf{w_{i:i+3,:}} \tag{12}$$

Here, $\circ$ denotes the Hadamard product, i.e. element-wise multiplication. Thus, each weight is updated by subtracting the decay term and adding the central term to the previous value. The central term is the product of the activation of the neuron i, the difference in activation of neuron j and the average, and the λ value corresponding to the anatomical nature of the connection (corticocortical or MTL-cortical). The procedure is then repeated for connections to the MTL with the important correction that the λ and ρ vectors consist entirely of the cortical-MTL values.

3.4 Results

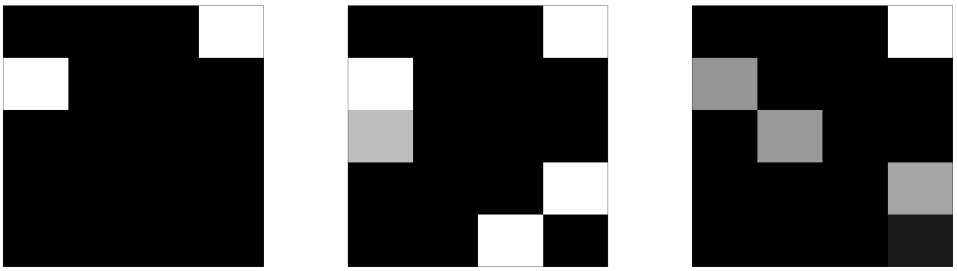


(a) $t = 1$. (b) $t = 2$. (c) $t = 3$.

Figure 13: Activity Cycling Through Network.

Figure 13 shows an example training from the same pattern as presented above. Each subfigure shows the cycling of activity one time step later. As evidenced by these images, activity diffuses into the MTL, strongly activating

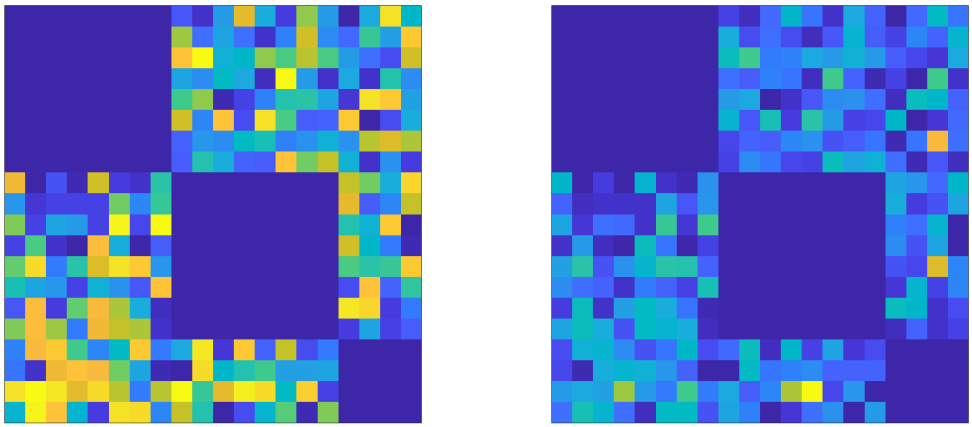
a single neuron. After training, the consolidation phase occurs. A counter is reset and is incremented up to the specified number of consolidation steps. For each step, one neuron is randomly selected from the four MTL cells. Its activity is set to 1 and all other cells are set to 0. The activity and weights are then updated three times. Following consolidation, a random permutation is selected for presentation of testing patterns. Each pattern is split into two half-patterns corresponding to the activity in one of the two cortical areas. For each entry in this permutation, the corresponding half-pattern is presented to the trained network. The test function is called, which takes in the trained weight matrix, the half-pattern, the decay factor, the number of cycling time steps, and a parameter "lesion." If lesion is true (1), all weights to and from MTL neurons are set to 0. The UpdateActivity function is then called for the specified number of times without updating weights, and the final activity pattern is returned to the script. For each of the four half-pattern presentations, the sum of squared errors is calculated between the total desired pattern and the returned activity of the sixteen cortical neurons. This error is summed across all four presentations.



(a) Half-Pattern. (b) Normal. (c) Lesion.

Figure 14: Testing by Presentation of Half-Pattern.

An example reconstruction of the pattern above based on the half-pattern in the first subfigure with no consolidation time is shown in Figure 14. It is clear from these results that the normally trained network is capable of reconstructing activity in the desired neurons based on the activation of the MTL cell active during training. Without activation of this cell and without sufficient consolidation time, the lesioned network is unable to properly reconstruct the memory.



(a) After Initialization. (b) After Training.

Figure 15: Weight Matrix.

Figure 15 shows an example of a randomly initialized and trained weight matrix. Values are randomly initialized between 0.0 and 0.2 as specified in the paper. Note the squares of no connection along the diagonal, consistent with no local excitatory connections within a given area (each cortex region or the MTL). After training, most weights are decreased in value, except for a few specific weights with high value. Those along the second-to-last row project

to the MTL, and those along the second-to-last column project back to cortical areas. Thus, these weights are those which stored the memory in cell 19, as seen in the pattern reconstruction.

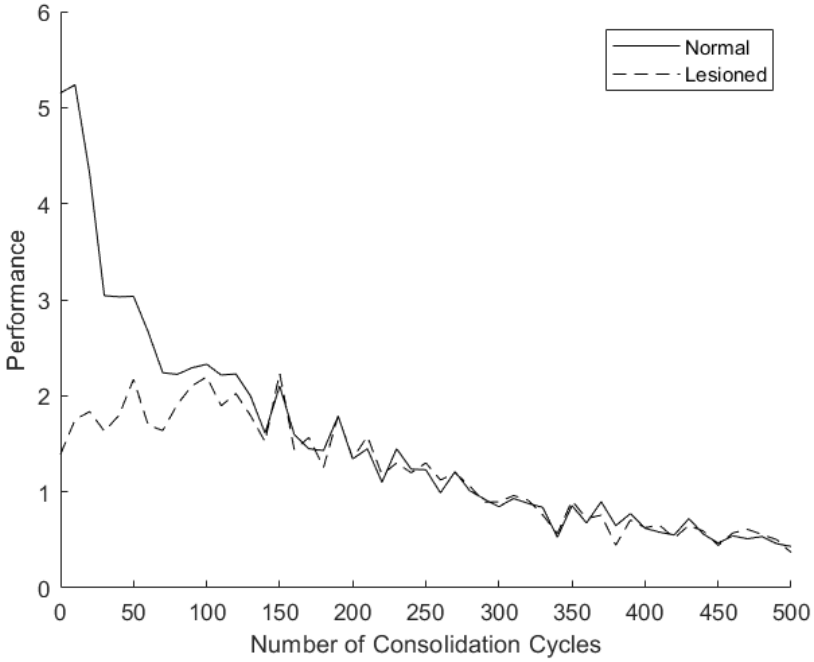


Figure 16: Effect of Lesioning and Consolidation Time.

To fully explore the influence of consolidation, training and testing were iterated from 0 to 500 consolidation cycles in steps of 10. For each step, a network was trained for three cycles for two presentations of each of the two patterns, delayed for the specified number of consolidation steps, and then tested both with and without lesioning on each half-pattern. Activity was summed for lesioned and non-lesioned case. Figure 16 shows the results averaged over 50 simulations, as in Figure 2 of the publication. These results are qualitatively similar- the normal network starts out with high performance which decays rapidly (approximately exponentially) over around 100 consolidation steps, after which the decay is approximately linear. Lesioning prevents this initial high performance, which relies on activity of the MTL. There are differences between these results and the paper, however. The effect of lesioning is not as pronounced here as in Figure 2. In the paper, the performance starts at near zero and rises to a peak before it meets the decay of the normal model. The performance of the normal model also starts higher. However, it should be noted that the formula used to calculate performance in the paper is not the same one used here. Alvarez and Squire use $Performance = 8 - SSE$, whereas here $11 - SSE$ is used, since the former produced negative performance values. Lastly, the trend here is much noisier. It is possible that reducing the spacing between successive consolidation values may reduce the fluctuations, as well as increasing the number of simulations for averaging. It is unclear, however, why the magnitude of the SSEs are off from the paper. In any case, the trends of the model match what is expected from the model, recapitulating the conclusions drawn by Alvarez and Squire concerning the ability of a rapidly changing, low-capacity network such as the medial temporal lobe in facilitating the gradual consolidation of memories in a slowly changing, high-capacity network such as the neocortex.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018.

[2] P. Alvarez and L. R. Squire, "Memory consolidation and the medial temporal lobe: a simple network model," *Proceedings of the national academy of sciences*, vol. 91, no. 15, pp. 7041–7045, 1994.

4 Appendix

Script to run the training and testing procedure for the TD maze learning simulation. Learning is recorded by subtracting the old value estimates from the new value estimates at each iteration, and the mean over the entire value matrix is a proxy for the amount of learning which occurred. An example maze is run after training. The procedure is repeated for the semi-directed TD algorithm. The influence of the trial number parameters N and Ntrial is then examined, where the former denotes the maximum number of iterations allowed in a given training episode, and the latter indicates the number of training episodes. A nested loop over all possible combinations is run for random and semi-directed learning. The amount of successes in the following testing procedure is recorded. Each N/Nmax combination consists of Ntest = 100 testing events, and for each testing event the value estimates are trained Ntrial times. An analogous paradigm is performed for different combinations of the learning rate and discounting factor, with both the amount of successes and average number of iterations needed to reach the goal during training recorded.

```
%% Initialization
maze = [1 0 1 1 1 1 1;1 1 1 0 0 1 0; 0 0 0 1 1 1 0; 1 1 1 1 0 0 1; 1 0 0 0 1 1
    1; 1 0 1 1 1 1 1;1 1 1 0 1 1 1];
imagesc(maze);colormap(gray); pbaspect([1 1 1]);
set(gca,'XTick',[], 'YTick', []);
rewards = -1.*maze;
Vtrue = -1*[26 -1 22 21 20 19 20;25 24 23 -1 -1 18 -1;-1 -1 -1 15 16 17 -1; 11
    12 13 14 -1 -1 3;10 -1 -1 -1 4 3 2;9 -1 5 4 3 2 1;8 7 6 -1 2 1 0];

N = 1000; %maximum number of training iterations
Ntrial = 100; %number of trials
Nmax = 1000; %max number of test iterations
V = zeros(size(maze)); %value
Vcontrol = -1*maze;

alpha = 0.2; %learning rate
gamma = 0.9; %discounting factor
start = [1 1]; %starting position
goal = 'end';

%% Train Random Search RL
k = zeros(1,Ntrial);
Learning = k;
for i = 1:Ntrial
    Vold = V;
    [V, k(i)] = TD(maze,rewards,V,N,alpha,gamma,start,goal);
    Learning(i) = mean(mean(V-Vold));
end
%V = V/sqrt(Ntrial);
imagesc(V);colormap(gray);
RunMaze(maze, V, start, goal, Nmax);
pause;
%% Train Controlled RL
kctrl = zeros(1,Ntrial);
Learningcontrol = kctrl;

for i = 1:Ntrial
    %keyboard;
    Vcontrolold = Vcontrol;
    [Vcontrol,kctrl(i)] = TDControl(maze,rewards,Vcontrol,N,alpha,gamma,start,
        goal);
    Learningcontrol(i) = mean(mean(Vcontrol-Vcontrolold));
end
imagesc(Vcontrol);colormap(gray);
```

```
RunMaze(maze, Vcontrol, start, goal, Nmax);
pause;
subplot(2,1,1);
plot(Learning);
xlabel('Training Trial Number');
ylabel('Average Change in Value');
title('Random');
subplot(2,1,2);
plot(Learningcontrol);
xlabel('Training Trial Number');
ylabel('Average Change in Value');
title('Controlled');


%% Test influence of Trial Number Parameters
Ns = 10:100:1010;
Ntrials = 1:10:101;
kconverge = zeros(length(Ns),length(Ntrials));
kconvergecontrol = kconverge;
Ntest = 100;
for i = 1:length(Ns) %maximum number of training iterations
    for j = 1:length(Ntrials) %number of trials
        V = zeros(size(maze)); %value
        Vcontrol = -1*maze;
        NumSuccess = 0;
        NumSuccessControl = 0;
        Ntrial = Ntrials(j);
        N = Ns(i);

        for k = 1:Ntest
            for l = 1:Ntrial
                [V, ~] = TD(maze,rewards,V,N,alpha,gamma,start,goal);
            end
            Succeed = RunMaze(maze, V, start, goal, Nmax);
            kconverge(i,j) = kconverge(i,j)+Succeed;
        end

        for k = 1:Ntest
            for l = 1:Ntrial
                [Vcontrol,~] = TDControl(maze,rewards,Vcontrol,N,alpha,gamma,
                   start,goal);
            end
            Succeed = RunMaze(maze, Vcontrol, start, goal, Nmax);
            kconvergecontrol(i,j) = kconvergecontrol(i,j)+Succeed;
        end
        disp(N);
        disp(Ntrial);
    end
end

figure;
surf(Ns, Ntrials, kconverge);
xlabel('Maximum Number of Training Iterations');
ylabel('Number of Training Episodes');
zlabel('Completed Maze Runs (%)');
figure;
surf(Ns, Ntrials, kconvergecontrol);
```

```
xlabel('Maximum Number of Training Iterations');
ylabel('Number of Training Episodes');
zlabel('Completed Maze Runs (%)');


%% Test influence of Learning Parameters
Ntrial = 50;
N = 300;
alphas = 0.1:0.1:1.0;
gammas = 0.1:0.1:1.0;
hyperparams = zeros(length(alphas),length(gammas),2);
hyperparamscontrol = hyperparams;

for i = 1:length(alphas)
    for j = 1:length(gammas)
        alpha = alphas(i);
        gamma = gammas(j);
        trials = 0;
        trialscontrol = 0;
        for k = 1:Ntest
            for l = 1:Ntrial
                [V, t] = TD(maze,rewards,V,N,alpha,gamma,start,goal);
                trials = trials + t;
            end
            hyperparams(i,j,2) = trials/Ntrial;
            Succeed = RunMaze(maze, V, start, goal, Nmax);
            hyperparams(i,j,1) = hyperparams(i,j,1)+Succeed;
        end

        for k = 1:Ntest
            for l = 1:Ntrial
                [Vcontrol,tc] = TDControl(maze,rewards,Vcontrol,N,alpha,gamma,
                   start,goal);
                trialscontrol = trialscontrol + tc;
            end
            hyperparamscontrol(i,j,2) = trialscontrol/Ntrial;

            Succeed = RunMaze(maze, Vcontrol, start, goal, Nmax);
            hyperparamscontrol(i,j,1) = hyperparamscontrol(i,j,1)+Succeed;
        end
        disp(alpha);
        disp(gamma);
    end
end

figure;
surf(alphas, gammas, hyperparams(:,:,1));
xlabel('Learning Rate');
ylabel('Discounting Factor');
zlabel('Completed Maze Runs (%)');
figure;
surf(alphas, gammas, hyperparamscontrol(:,:,1));
xlabel('Learning Rate');
ylabel('Discounting Factor');
zlabel('Completed Maze Runs (%)');

figure;
```

```
surf(alphas, gammas, hyperparams(:,:,2)/N);
xlabel('Learning Rate');
ylabel('Discounting Factor');
zlabel('Average Number of Iterations to Completion');
figure;
surf(alphas, gammas, hyperparamscontrol(:,:,2)/N);
xlabel('Learning Rate');
ylabel('Discounting Factor');
zlabel('Average Number of Iterations to Completion');
```

Function to determine permitted transitions at any point in the maze. Forbidden transitions are those which try to move the agent outside of the borders of the maze. A n x m x 4 matrix is returned for nearest-neighbor transitions where the first n x m matrix is the upward movement, the second is rightward, the third is downward, and the fourth is leftward. A 1 indicates the movement from that state (i,j) in the specified direction is permitted, and a 0 indicates that it is forbidden. Forbidden transitions are also those that lead the agent into a wall within the maze (corresponding to a value in the binary maze matrix of 0).

```
function trans = MazeTransitions(maze)

s = size (maze);
s1 = s(1);
s2 = s(2);
trans = zeros(s1,s2,4); %NESW
for i = 2:s(1)-1
    for j = 2:s(2)-1
        if maze(i,j) == 0
            continue;
        end
        trans(i,j,1) = maze(i-1,j);
        trans(i,j,2) = maze(i,j+1);
        trans(i,j,3) = maze(i+1,j);
        trans(i,j,4) = maze(i,j-1);
    end
end
%% Edge cases
trans(1,2:s2-1,2) = maze(1,3:s2);
trans(1,2:s2-1,3) = maze(2,2:s2-1);
trans(1,2:s2-1,4) = maze(1,1:s2-2);

trans(s1,2:s2-1,1) = maze(s1-1,2:s2-1);
trans(s1,2:s2-1,2) = maze(s1,3:s2);
trans(s1,2:s2-1,4) = maze(s1,1:s2-2);

trans(2:s1-1,1,1) = maze(1:s1-2,1);
trans(2:s1-1,1,2) = maze(2:s1-1,2);
trans(2:s1-1,1,3) = maze(3:s1,1);

trans(2:s1-1,s2,1) = maze(1:s1-2,s2);
trans(2:s1-1,s2,3) = maze(3:s1,s2);
trans(2:s1-1,s2,4) = maze(2:s1-1,s2-1);

%% Corners
trans(1,1,2) = maze(1,2);
trans(1,1,3) = maze(2,1);

trans(1,s2,3) = maze(2,s2);
trans(1,s2,4) = maze(1,s2-1);
```

```
trans(s1,1,1) = maze(s1-1,1);
trans(s1,1,2) = maze(s1,2);

trans(s1,s2,1) = maze(s1-1,s2);
trans(s1,s2,4) = maze(s1,s2-1);

for i = 1:s1
    for j = 1:s2
        if maze(i,j) == 0
            trans(i,j,:) = 0;
        end
    end
end
```

Function to normalize transition matrices so the sum of the probabilities of all possible transitions at a given state is 1.

```
function ptrans = MazeProb(trans)
s = size(trans);
ptrans = zeros(s);
for i = 1:s(1)
    for j = 1:s(2)
        if sum(trans(i,j,:)) == 0
            continue;
        end
        ptrans(i,j,:) = trans(i,j,:)./sum(trans(i,j,:));
    end
end
```

Temporal difference training function. Transitions between states are randomly chosen from available states with equal probability, and the Bellman equation is used to update the estimate for the value function of the current state. If the goal state is reached, the operation terminates and the number of iterations to reach the goal state is recorded.

```
function [Vend,k] = TD(maze,rewards,V,N,alpha,gamma,start,goal)
St = start;
if strcmp(goal, 'end')
    goal = size(V);
end

trans = MazeTransitions(maze);
ptrans = MazeProb(trans);
%figure;
for k = 1:N
    transitions = ptrans(St(1),St(2),:);
    transitions = transitions./min(transitions(find(transitions > 0)));
    mazecurrent = maze;
    mazecurrent(St(1),St(2)) = 0.5;
    %imagesc(mazecurrent); colormap(gray);drawnow;
    transset = [repelem(1,transitions(1)),repelem(2,transitions(2)), repelem(3,
       transitions(3)), repelem(4,transitions(4))];
    newstate = transset(randi(length(transset)));
    %keyboard;
    switch newstate
        case 1
            St1 = [St(1)-1 St(2)];
        case 2
            St1 = [St(1) St(2)+1];
        case 3
```

```
            St1 = [St(1)+1 St(2)];
        case 4
            St1 = [St(1) St(2)-1];
    end
    V(St(1),St(2)) = V(St(1),St(2)) + alpha*(rewards(St(1),St(2))+gamma*V(St1(1)
        ,St1(2))-V(St(1),St(2)));
    if all(St1 == goal)
        break;
    end
    St = St1;
end
% disp('Finished');
% disp('N =')
% disp(k);
Vend = V;
Vend(find(maze == 0)) = min(min(Vend))-1;

end
```

Semi-directed temporal difference training function. The TransitionScale function is called to adjust the transition probabilities before choosing the next state based on current value estimates for candidate states.

```
function [Vend,k] = TDControl(maze,rewards,V,N,alpha,gamma,start,goal)
St = start;
if strcmp(goal, 'end')
    goal = size(V);
end

sz = size(maze);
trans = MazeTransitions(maze);
ptrans = MazeProb(trans);
%figure;
for k = 1:N
    transitions = reshape(ptrans(St(1),St(2),:),[4,1,1]);

    Neighbors = [St(1)-1 St(2);St(1) St(2)+1; St(1)+1 St(2); St(1) St(2)-1];
    Valid1 = intersect(find(Neighbors(:,1)~=0),find(Neighbors(:,2)~=0));
    Valid2 = intersect(find(Neighbors(:,1)<=sz(1)),find(Neighbors(:,2)<=sz(2)));
    Valid = intersect(Valid1,Valid2);
    row = Neighbors(Valid',1);
    col = Neighbors(Valid',2);
    NVals = V(sub2ind(sz,row,col));

    if any(NVals ~= 0)
        transitions = TransitionScale(transitions,Valid,NVals);
    else
        transitions = transitions./min(transitions(find(transitions > 0)));
    end
    mazecurrent = maze;
    mazecurrent(St(1),St(2)) = 0.5;
    %imagesc(mazecurrent); colormap(gray);drawnow;
    %keyboard;
    transset = [repelem(1,transitions(1)),repelem(2,transitions(2)), repelem(3,
       transitions(3)), repelem(4,transitions(4))];
    newstate = transset(randi(length(transset)));
    switch newstate
        case 1
            St1 = [St(1)-1 St(2)];
```

```
        case 2
            St1 = [St(1) St(2)+1];
        case 3
            St1 = [St(1)+1 St(2)];
        case 4
            St1 = [St(1) St(2)-1];
    end
    %keyboard;
    V(St(1),St(2)) = V(St(1),St(2)) + alpha*(rewards(St(1),St(2))+gamma*V(St1(1)
        ,St1(2))-V(St(1),St(2)));
    if all(St1 == goal)
        break;
    end
    St = St1;
end
% disp('Finished');
% disp('N =')
% disp(k);
Vend = V;
Vend(find(maze == 0)) = min(min(Vend))-1;

end
```

Function to scale transition probabilities based on current value estimates, as described above.

```
function transitions = TransitionScale(transitions, Valid, NVals)

        transitionsvalid = transitions(Valid);
        if nnz(transitionsvalid) == 1

        else
            for i = 1:length(transitionsvalid)
                transscale(i) = transitionsvalid(i)*sum(NVals(setdiff(1:length(
                    transitionsvalid),i)));
            end
            transnorm = transscale/sum(transscale);
                    %keyboard;
            transint = ceil(transnorm./min(transnorm(find(transnorm > 0))));

            transitions(Valid) = transint;
        end
end
```

Function to run the maze after training RL agent. Success is 1 if the agent reaches the goal within the allotted number of iterations (Nmax), and is 0 otherwise.

```
function Success = RunMaze(maze, Value,start,goal,Nmax)
St = start;
sz = size(maze);
if strcmp(goal, 'end')
    goal = size(Value);
end
i = 0;
Success = 1;
mazecurrent = maze;
while any(St ~= goal)
    i = i+1;
    mazecurrent(St(1),St(2)) = (1-mazecurrent(St(1),St(2)))/2+0.25;
    %imagesc(mazecurrent); colormap(gray);drawnow;
```

```
    Neighbors = [St(1)-1 St(2);St(1) St(2)+1; St(1)+1 St(2); St(1) St(2)-1];
    Valid1 = intersect(find(Neighbors(:,1)~=0),find(Neighbors(:,2)~=0));
    Valid2 = intersect(find(Neighbors(:,1)<=sz(1)),find(Neighbors(:,2)<=sz(2)));
    Valid = intersect(Valid1,Valid2);
    row = Neighbors(Valid',1);
    col = Neighbors(Valid',2);
    NVals = Value(sub2ind(sz,row,col));
    St = Neighbors(Valid(find(NVals == max(NVals))),:);
    %keyboard;
    if i>Nmax
        Success = 0;
        break;
    end
end
mazecurrent(St(1),St(2)) = 0.5;
%imagesc(mazecurrent); colormap(gray);drawnow;
set(gca,'XTick',[], 'YTick', []);
%keyboard;
end
```

Script to perform mountain car simulation. Parameters are initialized, and a loop trains the agent for Ntrial episodes. In each episode, an epsilon-greedy algorithm is used to choose the action, and the state is updated based on the chosen action. From the information at the new state, the state-action value is updated, and the episode is terminated if the goal is reached. After training has completed, the agent has 3N iterations to reach the goal by using a greedy choice algorithm.

```
%state is characterized by velcoity and position
%action is characterized by left, right, or nothing
alpha = 0.15;             %learning rate
gamma = 0.9;             %discounting factor
xmin = -1.2;             %bounds on position
xmax = 0.6;
vmin = -0.07;            %bounds on velocity
vmax = 0.07;
Amin = -1;               %bounds on acceleration
Amax = 1;
x = round([xmin:0.1:xmax],1);
v = round([vmin:0.01:vmax],2);
A = [Amin,0,Amax];
R = -1*ones(length(x),1);
R(end-1) = 0;
R(end) = 0;
Q = zeros(length(x),length(v),length(A));
s = 3;                   %slope factor for sinusoid
N = 30000;                 %number of training trials
aconst = 0.001;
vconst = -0.0025;
e = 0.1;
Ntrial = 6000;

trajectory = zeros(Ntrial,N,3);

for l = 1:Ntrial
vt = 0;
xt = -.1*(randi(3)+3);
At = 2;
    for n = 1:N
        trajectory(l,n,:) = [xt,vt,A(At)];
```

```
        Aold = At;
        At = eGreedy(Q(find(x == round(xt,1)),find(v == round(vt,2)),:),e);
        xold = round(xt,1);
        vold = round(vt,2);
        %keyboard;

        [xt,vt] = StateUpdate(xt,vt,A(At),s,x,v,aconst,vconst);
        Q(find(x == xold),find(v == vold),Aold) = SARSA(Q(find(x == xold),find(v
            == vold),Aold),Q(find(x == round(xt,1)),find(v == round(vt,2)),At),R
           (find(x == round(xt,1))),alpha,gamma);
        %keyboard;
        %DrawCar([x(1),x(end)],[-1.2 1.2],xt,s);drawnow;
        if xt >= x(end)-0.1
            count(l) = n;
            break;
        end
    end
end

vt = 0;
xt = -.1*(randi(3)+3);
At = 2;
for n = 1:3*N
    Qs = Q(find(x == round(xt,1)),find(v == round(vt,2)),:);
    At = find(Qs == max(Qs));
    if length(At) > 1
       At = randi(length(At),1);
    end
    [xt,vt] = StateUpdate(xt,vt,A(At),s,x,v,aconst,vconst);
    xtest(n) = xt;
    vtest(n) = vt;
    %saveas(gcf,join(['Fig',string(n)]));
    %DrawCar([x(1),x(end)],[-1.2 1.2],xt,s);drawnow;
    if xt >= x(end)-0.1
        disp('finished');
        disp(n);
        break;
    end
end
```

Function to draw current state of car and hill.

```
function [] = DrawCar(xrange,yrange,carloc,s)
Offsetx = 0.05;
Offsety = 0.04;
CarOffsety = 0.08;
clf;
hold on;
x = xrange(1):0.001:xrange(2);
plot(x,sin(s*x),'k', 'LineWidth',2);

Wheel1 = RotateOffset(carloc,[carloc-Offsetx,sin(s*carloc)+Offsety],s);
Wheel2 = RotateOffset(carloc,[carloc+Offsetx,sin(s*carloc)+Offsety],s);
plot(Wheel1(1),Wheel1(2),'o','MarkerSize',8, 'MarkerEdgeColor','k');
plot(Wheel2(1),Wheel2(2),'o','MarkerSize',8, 'MarkerEdgeColor','k');

Box = RotateOffset(carloc,[carloc-2*Offsetx,sin(s*carloc)+CarOffsety;carloc+2*
```

```
    Offsetx,sin(s*carloc)+CarOffsety;carloc-2*Offsetx,sin(s*carloc)+2*CarOffsety;
    carloc+2*Offsetx,sin(s*carloc)+2*CarOffsety],s);
%keyboard;
plot([Box(1,1) Box(1,2)],[Box(2,1) Box(2,2)],'k');
plot([Box(1,3) Box(1,4)],[Box(2,3) Box(2,4)],'k');
plot([Box(1,1) Box(1,3)],[Box(2,1) Box(2,3)],'k');
plot([Box(1,2) Box(1,4)],[Box(2,2) Box(2,4)],'k');


pbaspect([(xrange(2)-xrange(1)) (yrange(2)-yrange(1)) 1]);
ylim([yrange]);
xlim([xrange(1) xrange(2)]);
set(gca,'XTick',[], 'YTick', []);
hold off;


end
```

Rotates car based on current location.

```
function Offset = RotateOffset(loc,x,s)
    tangent = s*cos(s*loc);
    th = atan(tangent);

    x = x-[loc,sin(s*loc)];
    R = [cos(th) -sin(th);sin(th) cos(th)];
    Offset = R*x';
    Offset = Offset+[loc,sin(s*loc)]';
    %keyboard;
end
```

Function for updating Q value of current state-action pair based on observed reward and next state-action pair.

```
function Q = SARSA(Q1,Q2,R,alpha,gamma)
    Q = Q1 + alpha*(R+gamma*Q2-Q1);
    %keyboard;
end
```

Epsilon greedy algorithm for choosing next action.

```
function A = eGreedy(Q,e)
    if rand(1) >= 1-e
        A = find(Q == max(Q));
        if length(A) > 1
            A = randi(length(A),1);
        end
    else
        A = randi(3);
    end
end
```

Function to evaluate next state based on current state and chosen action.

```
function [xtnew,vtnew] = StateUpdate(xt,vt,a,s,x,v,aconst,vconst)
    vtnew = max(min(vt+aconst*a+vconst*cos(s*xt),v(end)),v(1));
    xtnew = max(min(xt + vt,x(end)),x(1));
    if xtnew <= x(1)
        vtnew = 0.01;
    end
    %keyboard;
end
```

Training and testing script for Alvarez-Squire Model. Nsim simulations are carried out and the results are averaged together. Each simulation consists of Numconsol events, where an event consists of weight initialization, training, consolidation delay with random activity, and testing. Errors are recorded and plotted as a function of consolidation time for tests with and without lesioning to probe the influence of the MTL.

```
%% Parameters
Nsim = 50;                                  %number of simulations for averaging
lambdaMTL = 0.1;
lambdaCor = 0.002;
lambda = [lambdaMTL lambdaCor];             %learning rates
rhoMTL = 0.04;
rhoCor = 0.0008;
rho = [rhoMTL rhoCor];                      %forgetting rates
delta = 0.7;                                %activity decay rate
sCor1 = [2 4];
NCor1 = sCor1(1)*sCor1(2);
sCor2 = [2 4];
NCor2 = sCor2(1)*sCor2(2);
sMTL = [1 4];
NMTL = sMTL(1)*sMTL(2);
numsteps = 3;
Ntot = NMTL+NCor1+NCor2;                    %total number of cells

Nconsolidation = 0:10:500;                  %different consolidation times
Numconsol = length(Nconsolidation);
error = zeros(2,Numconsol,Nsim);            %Sum of squared errors

%% Initialize weights
[w,wfixed] = InitializeWeights;


%% Training Procedure
for sim = 1:Nsim
    dt = 3; %cycling time steps
    p1 = zeros(1,20);
    p2 = zeros(1,20);

    %initialize two different random patterns with one active cell in
    %each cortical group of four
    rand1 = randperm(4,2);
    rand2 = 4+randperm(4,2);
    rand3 = 8+randperm(4,2);
    rand4 = 12+randperm(4,2);

    p1(rand1(1)) = 1;
    p1(rand2(1)) = 1;
    p1(rand3(1)) = 1;
    p1(rand4(1)) = 1;

    p2(rand1(2)) = 1;
    p2(rand2(2)) = 1;
    p2(rand3(2)) = 1;
    p2(rand4(2)) = 1;

    % Iterate over consolidation time
    for count = 1:Numconsol
        Nconsol = Nconsolidation(count);
        [w,wfixed] = InitializeWeights;
```

```
order = mod(randperm(4),2)+1;                %present patterns in random
    order
for i = 1:4
    if order(i) == 1                         %initialize activity for
        pattern
        a = p1;
    else
        a = p2;
    end
    %imagesc(reshape(a,4,5)');colormap(gray);drawnow;
    %keyboard;
    for n = 1:dt                             %cycle through activity and
        weight updates
        a = UpdateActivity(a,w,delta);
        w = UpdateWeights(a,w,wfixed,lambda,rho);
        %imagesc(reshape(a,4,5)');colormap(gray);drawnow;
        %keyboard;
    end
end

consolcount = 0;
while consolcount < Nconsol              %cycle through consolidation
    steps
    consolcount = consolcount + 1;
    arand = randi(4);                    %choose one random MTL neuron to
        be active
    a = zeros(1,20);
    a(16+arand) = 1;
    for n = 1:dt
        a = UpdateActivity(a,w,delta);
        w = UpdateWeights(a,w,wfixed,lambda,rho);
    end
end

testperm = randperm(4);                      %choose random presentation
    order
for testiter = 1:4
    testpat = testperm(testiter);
    astart = zeros(1,20);
    switch testpat                           %four different half-
        patterns for presentation
        case 1
            astart(rand1(1)) = 1;
            astart(rand2(1)) = 1;
            patt = p1;
        case 2
            astart(rand3(1)) = 1;
            astart(rand4(1)) = 1;
            patt = p1;
        case 3
            astart(rand1(2)) = 1;
            astart(rand2(2)) = 1;
            patt = p2;
        case 4
            astart(rand3(2)) = 1;
            astart(rand4(2)) = 1;
```

```
                patt = p2;
            end
            %imagesc(reshape(patt,4,5)');colormap(gray);drawnow;
            %keyboard;
            %imagesc(reshape(astart,4,5)');colormap(gray);drawnow;
            %keyboard;
            aoutnorm(testiter,:) = test(w,astart,delta,dt,0);           %
               evaluate with non-lesioned system
            %imagesc(reshape(aoutnorm(testiter,:),4,5)');colormap(gray);drawnow;
            %keyboard;
            aoutlesion(testiter,:) = test(w,astart,delta,dt,1);         %
               evaluate with lesioned system
            %imagesc(reshape(aoutlesion(testiter,:),4,5)');colormap(gray);
               drawnow;
            %keyboard;

            SSEnorm(testiter) = sum((aoutnorm(testiter,1:16)-patt(1:16)).^2);
               %sum of squared errors for each trial
            SSElesion(testiter) = sum((aoutlesion(testiter,1:16)-patt(1:16)).^2)
               ;
        end
        %keyboard;
        error(1,count,sim) = sum(SSEnorm);                              %total
           SSE over all four presentations
        error(2,count,sim) = sum(SSElesion);
    end
end

figure;
hold on;
plot(Nconsolidation,11-mean(error(1,:,:),3),'k');
plot(Nconsolidation,11-mean(error(2,:,:),3),'--k');
xlabel('Number of Consolidation Cycles');
ylabel('Performance');
legend({'Normal','Lesioned'});
hold off;
```

Weight initialization function for Alvarez-Squire Model.

```
function [w,wfixed] = InitializeWeights()
%% Initialize weights
%weights initialized between 0.0 and 0.2
w = rand(20)/5;
% w(:,j) are all connections projecting from neuron j
% w(i,:) are all connections projecting to neuron i
wfixed = ones(20);                  %ones if neurons are connected, zeros if not

connect = cell(1,5); %which layers are connected to which neurons
connect{1} = reshape((4*([3,4,5]'-1)+[1:4])',1,12);
connect{2} = reshape((4*([3,4,5]'-1)+[1:4])',1,12);
connect{3} = reshape((4*([1,2,5]'-1)+[1:4])',1,12);
connect{4} = reshape((4*([1,2,5]'-1)+[1:4])',1,12);
connect{5} = reshape((4*([1,2,3,4]'-1)+[1:4])',1,16);
for j = 1:5
    connected_neurons = connect{j};
    all_neurons = 1:20;
    disconnected_neurons = setdiff(all_neurons,connected_neurons);
    w(disconnected_neurons,4*(j-1)+[1:4]) = 0;
```

```
    wfixed(disconnected_neurons,4*(j-1)+[1:4]) = 0;
end
```

Activity update function for Alvarez-Squire Model.

```
%% Update activity
function a = UpdateActivity(a,w,d)
% update cortex
    neurons = 1:16;
    a(neurons) = d*a(neurons)+a*w(neurons,:)'+(rand(1,16)-0.5)/10;
    for i = 1:4:length(a)-4             %lateral inhibition in groups of 4
        neurons = i:i+3;
        act = a(neurons);               %activity of neurons in the layer
        m_ind = find(act==max(act))+i-1; %neuron with maximal activity
        a(setdiff(neurons,m_ind)) = 0;
    end

    a(find(a > 1)) = 1;
    a(find(a < 0)) = 0;
% update MTL
    neurons_in_layer = 17:20;           %repeat for MTL
    a(neurons_in_layer) = d*a(neurons_in_layer)+a*w(neurons_in_layer,:)'+(rand
       (1,4)-0.5)/10;
    m_ind = 16+find(a(neurons_in_layer) == max(a(neurons_in_layer)));
    a(setdiff(neurons_in_layer,m_ind)) = 0;
    a(find(a > 1)) = 1;
    a(find(a < 0)) = 0;
end
```

Weight update function for Alvarez-Squire Model.

```
%% Update weights
function w = UpdateWeights(a,w,wfixed,lam,rho)
    lamMTL = lam(1);
    lamCor = lam(2);
    rhoMTL = rho(1);
    rhoCor = rho(2);
    %update connections
    for i = 1:4:16
        neurons = i:i+3;                                %neurons in the layer
        wf = wfixed(neurons,:);                         %connections
        acti = a(neurons);                              %activity of the neurons
            in the layer
        abar = mean(a(find(wf(1,:)==1)));               %average activity of all
            neurons connected to the layer
        ajminusabar = wf.*(a-abar);                     %difference of each
           projection neuron and the average
        l = [repelem(lamCor,16),repelem(lamMTL,4)];
        r = [repelem(rhoCor,16),repelem(rhoMTL,4)];
        w(neurons,:) = w(neurons,:)+l.*acti'.*ajminusabar-r.*w(neurons,:);  %
           update weights
        %keyboard;
    end

    neurons = 17:20;
    wf = wfixed(neurons,:);
    acti = a(neurons);
    abar = mean(a(find(wf(1,:)==1)));
    ajminusabar = wf.*(a-abar);
```

```
    l = [repelem(lamMTL,20)];
    r = [repelem(rhoMTL,20)];
    w(neurons,:) = w(neurons,:)+l.*acti'.*ajminusabar-r.*w(neurons,:);  %update
       weights

    w=w.*wfixed;                                          %make sure all neurons
       that were unconnected remain so
    w(find(w > 1)) = 1;
    w(find(w < 0)) = 0;
end
```

Testing function for Alvarez-Squire model.

```
function a = test(w,pattern,d,dt,lesion)
    a = pattern;                                  %half-pattern for testing
    if lesion                                     %lesion by setting all
       connections to and from MTL neurons to 0
        w(:,17:20) = 0;
        w(17:20,:) = 0;
    end
    %keyboard;
    for t = 1:dt                                  %cycle through activity for
       three iterations
        a = UpdateActivity(a,w,d);
    end
    %keyboard;
end
```