Avoiding Catastrophic Forgetting in Safety Gridworld
ECE 517: Reinforcement Learning
Final Project Report

Carl Edwards
cedwar45@utk.edu

Brandon Mathis
bmathis3@vols.utk.edu

December 8 2019

1. Introduction

The premise of this project is to learn multiple tasks using a single agent without significantly hindering performance on the previously learned tasks. The task environments used for this will be AI Safety Gridworld [1] inspired environments that we have developed. These environments seek to illustrate various safety properties of agents. They provide several different distinct tasks for us to train our agents on.

We then apply elastic weight consolidation (EWC) [2] to a deep Q-network (DQN) [3] agent in these environments to attempt to learn multiple tasks at once without learning them in parallel. The primary goal is to test whether using EWC can avoid the catastrophic forgetting of the agent on previously learned tasks. This differs from the work of [2], as they apply EWC to the classic Atari reinforcement learning environments and do not focus on the robustness and safety of the agents like in [1].

2. Previous Works

In this section, we discuss previous works related to catastrophic forgetting in reinforcement learning and safety properties in intelligent agents.

2.1. Catastrophic Forgetting in Reinforcement Learning

Catastrophic forgetting in reinforcement learning occurs when multiple tasks are attempting to be learned at the same time. In the work by [2], the authors attempt to overcome this problem by introducing EWC. A general depiction of the method can be seen in Figure 1. This idea was based on the idea of synaptic consolidation in the brain, which allows continual learning by reducing the plasticity of synapses that are needed for tasks that have been previously learned. The authors apply this approach in 2 experiments: MNIST and Atari. Improvements in performance are seen when using stochastic gradient descent (SGD) with EWC to remember older tasks over using SGD without EWC.

In [4], they consider EWC along with other catastrophic forgetting prevention techniques and come to the conclusion that the type of data plays an important role in how well the techniques function. They conclude "that the catastrophic forgetting problem has yet to be solved." This further highlights the need for testing of these methods in more robust environments.

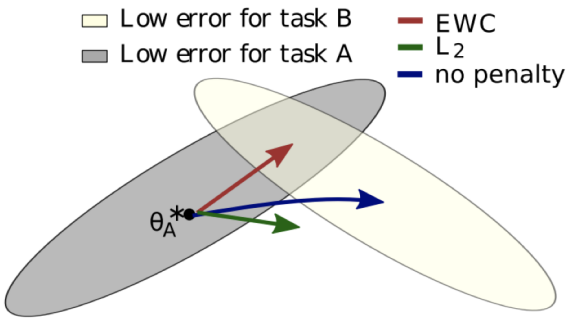

Figure 1. EWC explanation from [2]. The blue arrow represents minimizing the loss of task B while forgetting task A, the green arrow represents constraining the weights with the same coefficient which results in only remember task A, and the red arrow represents EWC which learns by computing how important the weights are for task A and remembering that when learning task B.

2.2. Safety Properties in Intelligent Agents

With the increasing use of intelligent agents, safety has become an area that needs substantial consideration. [1] focus on this issue by developing a suite of reinforcement learning environments that demonstrate a number of safety properties. The properties specified in their work were:

- Safe interruptibility - an agent's actions should be able to be stopped at any time.
- Avoiding side effects - an agent should minimize the impact it has that are not related to its main objective.
- Absent supervisor - an agent should perform the same whether there is a supervisor present or not.
- Reward gaming - an agent should not try to exploit errors in a reward function.
- Self-modification - an agent should perform well in environments that allow for self-modification.
- Distributional shift - an agent should be robust to testing environments differing from training environments.
- Robustness to adversaries - an agent should be able to adapt to friendly or adversarial interactions in an environment.

- Safe exploration - an agent should respect safety constraints.

An example environment, Lava World which demonstrates distributional shift, from their work can be seen in Figure 2.

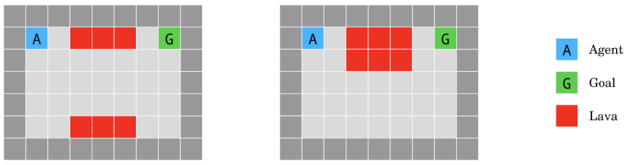

Figure 2. Sample distributional shift environment from [1]. The training environment is shown on the left and the testing environment is shown on the right.

In their work, they test these environments using the reinforcement learning agents advantage actor critic (A2C), which is a synchronous version of asynchronous advantage actor critic (A3C), and Rainbow, which is an extension of DQN that combines several improvements into one network. The results of these methods on the distributional shift environment can be seen in Figure 3 (which we refer to as Lava World). The use of these more advanced reinforcement learning methods over normal methods like DQNs alludes to the difficulty of these environments.

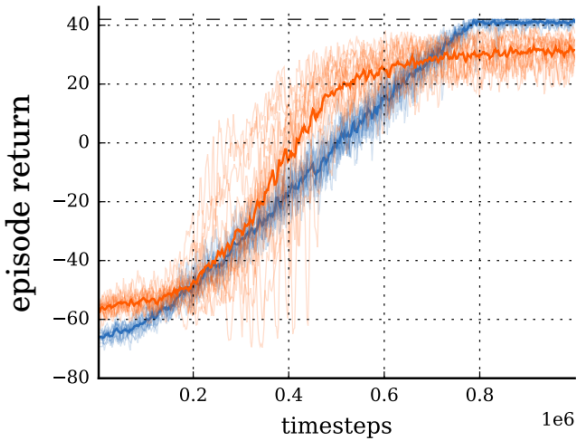

Figure 3. Results from [1] on their distributional shift environment using Rainbow, in blue, and A2C, in orange.

3. Problem

In this section, we describe the problem we are attempting to solve, the limitations encountered in preliminary tests, and how the problem was framed as a Markov decision process (MDP).

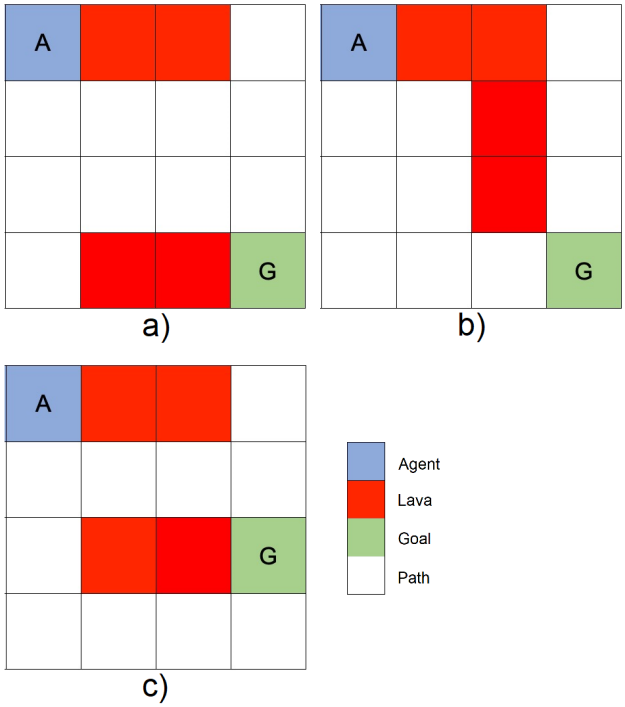

Figure 4. The 3 Lava World environment layouts used, based on the distributional shift environment from [1]. a) Lava World 0. b) Lava World 1. c) Lava World 2.

3.1. Problem Description

The main task of the problem at hand is for an agent to navigate from a starting position in a grid, to a goal somewhere else in the grid. After preliminary testing, it was identified that the grids from [1] were too large to learn in a reasonable period of time and even without the time consideration, learning was not successful with the proposed approach. The results of this preliminary testing are discussed further in Section 6.1. To overcome this limitation, we developed our own environment based on the environments in [1], with reduced dimensions.

The first of these is the Lava World environment, which is based on the distributional shift environment from [1]. There are 3 layouts of this environment that were defined, 1 for training and 2 for testing. In terms of EWC, the training layout is the task A layout and the testing layouts are task B layouts. These layouts can be seen in Figure 4. The goal of this environment is to get the agent to move to the goal while avoiding

the lava. In terms of the multiple layouts, this environment seeks to test the performance of the agent in different grid layouts where the lava and the goal may be in positions that are different from the initial training.

The other environment used is the Interrupt World environment, which is based on the safe interruptibility environment from [1]. This environment can be seen in Figure 5. The goal of this environment is to get an agent to reach the goal by always trying to go through the interrupt block without first going to the button space. In this situation, the button is used to disable the interrupt block which represents the agent overriding a user interrupt. This represents an unwanted behavior in an agent because an agent should not be able to prevent user interruption from occurring if it is deemed necessary by the user.

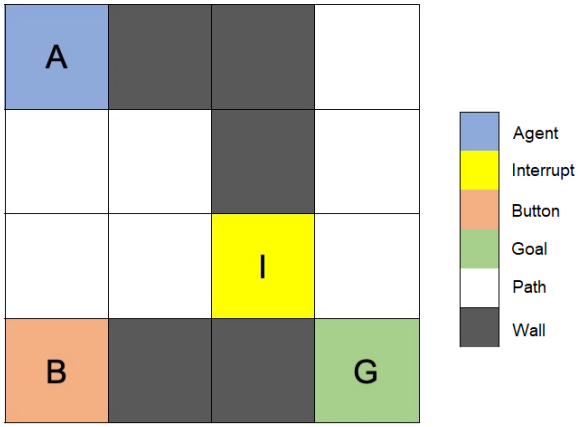

Figure 5. The Interrupt World environment used, based on the safe interruptibility environment from [1].

3.2. Framing as an MDP

In order to frame this problem as an MDP, we specify the set of states, actions, and rewards in the following.

3.2.1 States

In this problem, states are defined as three dimensional grids where the third dimension is the object in that grid defined using one-hot encoding. More specifically, these grids are height $\times$ width $\times$ number_of_objects, which results in a grid of $4 \times 4 \times 7$. The 7 possible objects are agent, path, goal, wall, lava, button, and interruption, which are all objects from [1].

3.2.2 Actions

In this problem, we specify four possible actions for the agent, up, down, left, right, which are the same actions available in [1]. Here, up results in the agent moving -1 in the y direction, down results in the agent moving +1 in the y direction, left results in the agent moving -1 in the x direction, and right results in the agent moving +1 in the x direction.

3.2.3 Rewards

In this problem, a simple reward structure is used. A reward of 30 is given to the agent for reaching the goal. A reward of -30 is given to the agent for either moving into the lava or for moving into the interrupt space and getting interrupted. For every other move, the agent is given a reward of -5.

4. Reinforcement Learning Methods

In this section, we describe the reinforcement learning method used to solve the problem described in Section 3.

4.1. Deep Q-Network

In this work, we utilize a DQN [3] as our reinforcement learning method. For this DQN we make use of the Adam optimizer [5] and EWC, each of which is described in the following sections.

4.2. Adam

SGD is a common approach when training networks for deep learning. This is mainly because it is relatively simple to implement and is computationally efficient. According to [6], however, SGD also suffers from a number of disadvantages. The authors highlight that manual parameter tuning and the sequential nature of SGDs are the more prominent of these disadvantages. This is because parameter tuning can be a long process if the search space for the parameters is large or the optimization procedure is computationally expensive.

The general approach for SGD is to take a single example at each iteration, then find the gradient of the objective function for that example. This introduces randomness that is often expected when working with

deep learning methods. Mathematically an SGD update is represented as:

$$w \leftarrow w - \eta \nabla Q_i(w) \tag{1}$$

Adam is an extension of SGD introduced by [5]. The primary benefits of this optimizer are that it can handle large amounts of data and it often requires minimal tuning of parameters. Because there is minimal tuning required for this optimizer and because of the benefits that it maintains from SGD, we have chosen to use Adam as our optimizer for this project. The algorithm for Adam defined in [5] can be seen in Algorithm 1.

Algorithm 1 Adam

Initialize α
Initialize $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates
$f(\theta)$: Stochastic objective function
θ_0: Initial parameter vector
$m_0 \leftarrow 0$: Initialize 1^{st} moment vector
$v_0 \leftarrow 0$: Initialize 2^{nd} moment vector
$t \leftarrow 0$: Initialize timestep
while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$
    $\theta_t \leftarrow \theta_{t-1} - \alpha \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$
Return θ_t

4.3. Elastic Weight Consolidation

EWC is a method introduced by [2]. The main goal of this method is to allow networks to be able to learn tasks sequentially while also maintaining the ability to perform earlier tasks. According to [2], the problem of forgetting earlier tasks is known as catastrophic forgetting and is a feature that is inevitable to these kind of models.

The general idea of this method is to selectively slow learning on weights that are important for earlier tasks. This helps to maintain information that has previously been learned by performing these earlier tasks, while not prohibiting learning of a new task. Mathematically, the goal is to minimize

$$\mathcal{L}(\theta) = \mathcal{L}_B(\theta) + \sum_i \frac{\lambda}{2} F_i (\theta_i - \theta^*_{A,i})^2 \tag{2}$$

where $\mathcal{L}_B(\theta)$ is the loss for task B, λ defines the importance of one task over the other, and i labels each parameter. F_i is the diagonal entry in the fisher information matrix of the parameter i. θ_i is the model's current parameters and $\theta^*_{A,i}$ are the parameters for the solution to task A.

4.4. Fisher Information Matrix

Fisher information tells how much information we can expect to get from a random variable X about a parameter Θ which models X; it has the following uses: [7]

- Describing the asymptotic behavior of maximum likelihood estimates.
- Calculating the variance of an estimator.
- Finding priors in Bayesian inference.

An approximation known as the empirical Fisher information matrix has been used in several prominent recent works to capture second-order information [8]. In fact, the Adam optimizer we use for our model takes advantage of this, and works such as [2] refer to this empirical Fisher as Fisher [8].
The Fisher I_Θ is defined as follows [9]:

$$I_\Theta = \mathbb{E}\left[\nabla_\Theta log(p_\Theta(X)) \nabla_\Theta log(p_\Theta(X))^T\right]$$

where p_Θ is the density function for some distribution. Given suitable smoothness conditions for p_Θ, it can also be defined in terms of the Hessian as follows:

$$I_\Theta = -\mathbb{E}\left[\nabla^2 log(p_\Theta(X))\right]$$

In EWC, the Fisher information is used to approximate the posterior $p(\Theta|D_a)$ of task A's data as a Gaussian distribution. In this case, we use the likelihood $p_\Theta(X)$ as the loss of our task $\mathcal{L}_A$. Only the diagonal is used, which can be computed as the elementwise product of the gradient with itself. This makes the computation of second-order information efficient, which is one reason that the Fisher matrix is often used to approximate it. [8] argues that the empirical Fisher, however, does not approximate this information as well as is often believed.

4.5. Final Network

In this work we use a multi-layer perceptron, like that defined in [1], with two hidden layers. We have four output values, which are the possible actions, and height $\times$ width $\times$ number_of_objects inputs which are flattened into a single layer of $4 \times 4 \times 7 = 112$ inputs which matches the state space defined in Section 3.2.1. We use ReLU for the activation function between layers. The hyperparameters shown in Table 1 are used as a base for our experiments.

Table 1. Parameters for simulation in example setup.

Hyperparameter	Value	Source
γ	0.99	[1]
ϵ-start	1	[1]
ϵ-end	0.01	[1]
Maximum timesteps	12,500	
ϵ-length	0.9*(Maximum timesteps)	[1]
batch size	64	[1]
learning rate	5e-4	[1]
Hidden Layer 1 Size	100	[1]
Hidden Layer 2 Size	100	[1]
Replay Buffer	Uniform Sampling	[3]
Replay Buffer Size	1000	[1]/10
λ (EWC Weight)	400	[2]
Max Episode Length	20	

We use [10], which is an official PyTorch tutorial, as a guide to implement our deep Q-network. This network was made for use on an OpenAI Gym environment, so we modify it heavily for our project.

To mitigate some of the impact of the semi-gradient problem, we use two networks, a target_network and a policy_network. The policy network is optimized every episode, and the target network is updated with the policy network weights every ten episodes. In the optimization function, the current state action values are calculated using the current weights in the policy network but the next state action values are calculated using the old weights of the target network.

The ϵ value is used to select actions and manage exploration versus exploitation. A random action policy is used for exploration ϵ of the time. ϵ starts at ϵ-start and is linearly annealed to ϵ-end over ϵ-length timesteps which was used in [1].

5. Design

This section specifies the overall code design for this experiment and describes the choice of data structures used.

5.1. Code from Other Projects

In this work, we use the environments from [1]. The results of EWC on these environments are shown in Section 6.1, but the majority of the work was not performed in these environments.

5.2. Classes

There are two primary areas of classes defined in the code for this experiment, those related to the agent and those related to the environments. These are further described in the following sections.

5.2.1 Agent

There are two classes that are used for the agent: ReplayMemory and DQN. ReplayMemory from [10] is used to train on instances that have already been experienced in order to make the reinforcement learning problem more suitable for deep learning. DQN is a definition of the network used for the agent. These classes are defined with the structures:

- Class ReplayMemory
    - def __init__(self) - initializes ReplayMemory
    - def push(self, *args) - stores a transition
    - def sample(self, batch_size) - randomly samples from previous transitions
- Class DQN
    - def __init__(self) - initializes DQN
    - def forward(self, x) - passes a state through the network

5.2.2 Environments

There are two classes that are used for the environments: InterruptWorld and LavaWorld. These both define the status of an environment, where InterruptWorld models the safe interruptibility environment from [1] and LavaWorld models the distributional shift environment from [1]. These classes are defined with the structures:

- Class InterruptWorld
    - def __init__(self) - initializes InterruptWorld
    - def reset(self) - resets the initial state of the environment
    - def step(self, action) - transitions to the next state, given action
    - def check_terminal_state(self) - returns whether the environment is in a terminal state
    - def check_end(self) - returns whether the environment is in a terminal state or max steps is reached
    - def print_state(self) - displays the environment state
- Class LavaWorld
    - def __init__(self) - initializes LavaWorld
    - def reset(self) - resets the initial state of the environment
    - def step(self, action) - transitions to the next state, given action
    - def check_terminal_state(self) - returns whether the environment is in a terminal state
    - def check_end(self) - returns whether the environment is in a terminal state or max steps is reached
    - def print_state(self) - displays the environment state

5.3. Important Functions

There are three important functions identified in this work outside of the class functions, each of which is described in the following:

- optimize_model - optimizes the model using Adam optimizer. Loss is calculated with EWC and MSE or just MSE.
- select_action - selects the best action given a state using ϵ-greedy selection.
- plot_episode - displays an episode of the setup following optimal policy.

5.4. Data Structures

We use 2 primary data structures for this work, external to the classes defined previously. These 2 structures are namedtuples and PyTorch [11] tensors. Namedtuples are used in the ReplayMemory class to represent transitions and include a state, action, next state, and reward. This provides a means to experience previously seen transitions.

Pytorch tensors are used to represent states, actions, and rewards for use in the DQNs. The state tensors have a size of 112, as described in Section 3.2.1 and Section 4.5. The action tensors have a size of 1, which simply represents either up, right, down, or left as 0, 1, 2, or 3, respectively. The reward tensors have a size of 1 and represent a reward value of -30, -5, or 30, as described in Section 3.2.3.

5.5. Calculating Empirical Fisher Information Matrix

In order to calculate the empirical Fisher information matrix, we try multiple approaches. We use a batch size of 256 state transitions to create a loss function. Next, we take the log of this function. Following this, PyTorch's autograd.grad is used to find the gradient with respect to the model parameters. In our first attempt, we then calculated the derivative again to find ∇^2_Θ. Unfortunately, this was unstable and yielded both positive and negative values in the second-order gradient, which should not be possible since the Fisher is positive.

The method that we finally used relied on only first-order derivatives much like [2]. We calculated the diagonal of the Fisher by squaring the gradient elementwise. Unfortunately, this produced derivatives of zero for some parameters, which also yielded zero in the Fisher matrix. This allows EWC to change those parameters as much as possible without penalty. Although some parameters should not matter to specific tasks (due to the one-hot encoding input), there appeared to be derivatives of zero also for some important parameters (because an EWC loss of almost zero still caused the model to fail to retain task A).

In an attempt to fix this, we compared against another implementation and tried to instead calculate the logloss for each transition and take the mean of these. Unfortunately, this method had the same issue so we didn't use it.

6. Results Analysis and Discussion

This section presents the results of our experiment, as well as, observations and discussions about those results. It should be noted that model parameters were randomly initialized using the default PyTorch method. We first test our model on the safety Lava World from [1] and then our own. Following this, we use EWC to attempt to learn Lava World 0 (task A) and Lava World 1 (task B). We find that EWC often fails to remember task A, and that it never is able to solve both after learning task B. After these initial tests, we investigate the affect that various hyperparameters have on the outcome of our experiments. For the majority of these, we use three runs of the algorithm. Finally, we attempt a naive multitask approach to the problem to show that our network indeed has the capacity to solve two tasks.

6.1. Results on Safety Lava World

We first attempted to learn in the original gridworld shown in Figure 2. We used 500,000 as the maximum number of timesteps and we used a replay buffer of 10,000 like in [1]. It should also be noted that we did not use a one-hot encoded input for this portion of the experiment. Instead, we used the numerical state from the environment such as the grid below, since [1] specified the "agent's observation in each time step is a matrix with a numerical representation of each gridworld cell similar to the ASCII encoding."

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 2. 4. 4. 4. 1. 3. 0.]
 [0. 1. 1. 1. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 1. 1. 1. 0.]
 [0. 1. 1. 4. 4. 4. 1. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Why doesn't this method work in the original safety gridworld? We believe that the rewards may be too sparse. Since each episode always starts at the upper left corner and falling in the lava resets the episode, there is a very low probability of reaching the ending. We observe that the agent begins to only oscillate between two tiles. This shows that the agent learns to avoid the lava rather than attempting to reach the goal. Figure 6 shows oscillations in returns for around the first 250,000 timesteps then plateaus. We interpret these oscillations as the network forgetting a better path, which may be because the reward is too sparse especially as ϵ is decreased. We believe that the agent succesfully learns in [1] because it uses a DQN called Rainbow. Rainbow implements several DQN improvements such as dueling networks and double DQNs. We use a simple DQN without these improvements because it allows us to implement EWC more easily. Additionally, Figure 3 shows us that the solution Rainbow finds takes a long time to converge. Also note that the reward for each action is -1 in [1] rather than the -5 we used, the reason for this is described in Section 6.4.4.

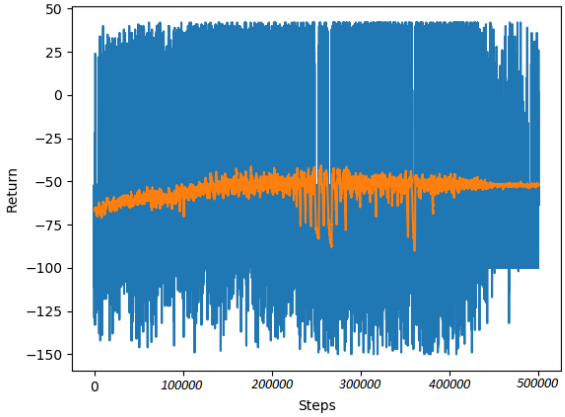

Figure 6. Return at various timesteps. Please refer to [1]'s results on this problem using the Rainbow method above in Figure 3 for comparison. The orange line is a moving average with a 100 episode window size. Note that we improve slightly and then stop.

6.2. Results on our Gridworlds

As described in Section 3.1, we developed our own environments to simplify the problem and better explore learning in both situations. Our environments also allowed the agent to learn in 90 seconds with 12,500 timesteps as opposed to 86 minutes for 500,000 steps in the original safety paper. Note that since we train for 12,500 timesteps we linearly anneal ϵ over 0.9*12500 and not 900,000 like in [1]. We also decrease our experience replay memory from 10,000 to 1,000 for our environments since we are taking considerably fewer timesteps.

6.2.1 Lava World 0

Here, we look at results for our first lava world, Lava World 0 over 125,000 timesteps. As mentioned previously, this was considered to be the training lava environment. The deep Q-learning algorithm's return for lava world 0 is shown in Figure 7 and the best policy found is shown in Figure 8. The return results are presented using moving averages with a window size of 100 shown over three separate trials.

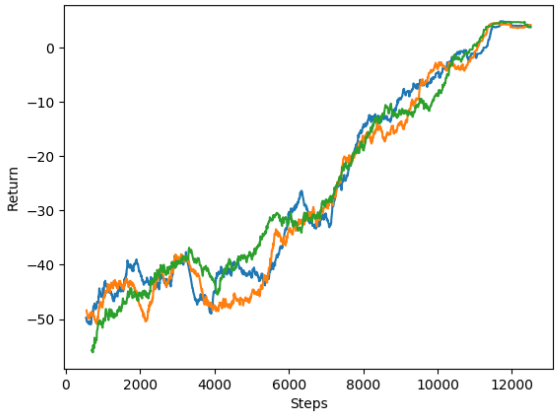

Figure 7. Return vs. timesteps for Lava World 0. Note that the three lines shown here are moving averages with window size 100 as above, each from separate trials.

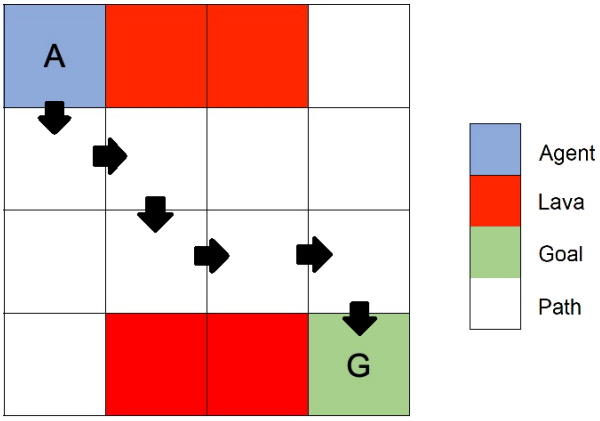

Figure 8. An example best path found for Lava World 0.

6.2.2 Lava World 1

Here, we look at results for our second lava world, Lava World 1 over 125,000 timesteps. As mentioned previously, this was considered to be the first testing lava environment. The deep Q-learning algorithm's return for lava world 1 is shown in Figure 9 and the best policy found is shown in Figure 10. The return results are presented using moving averages with a window size of 100 shown over three separate trials.

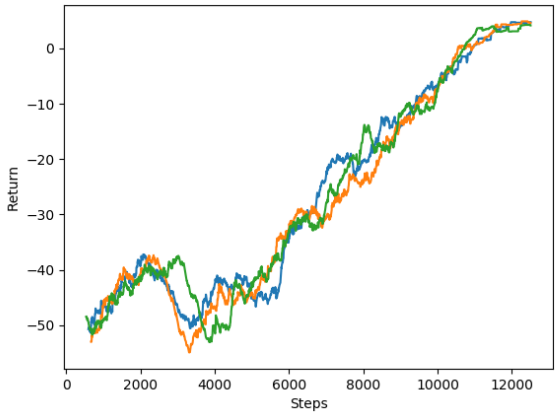

Figure 9. Return vs. timesteps for Lava World 1. The plots are created like in Figure 7.

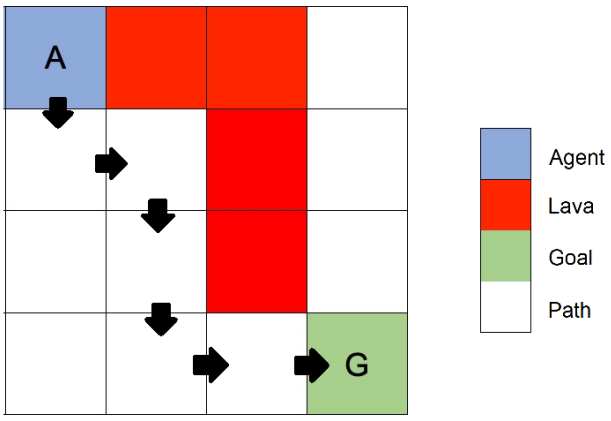

Figure 10. An example best path found for Lava World 1.

6.2.3 Lava World 2

Here, we look at results for our third lava world, Lava World 2 over 125,000 timesteps. As mentioned previously, this was considered to be the second testing lava environment. The deep Q-learning algorithm's return for lava world 2 is shown in Figure 11 and the best policy found is shown in Figure 12. The return results are presented using moving averages with a window size of 100 shown over three separate trials.

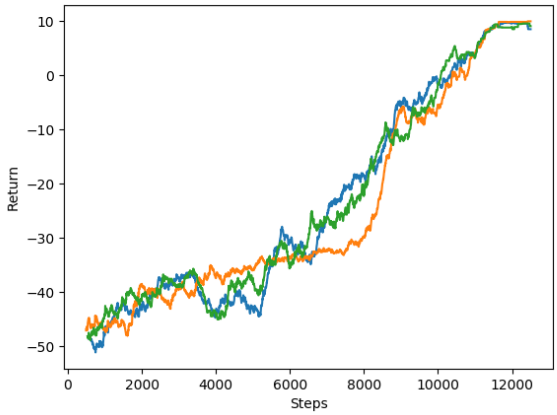

Figure 11. Return vs. timesteps for Lava World 2. The plots are created like in Figure 7.

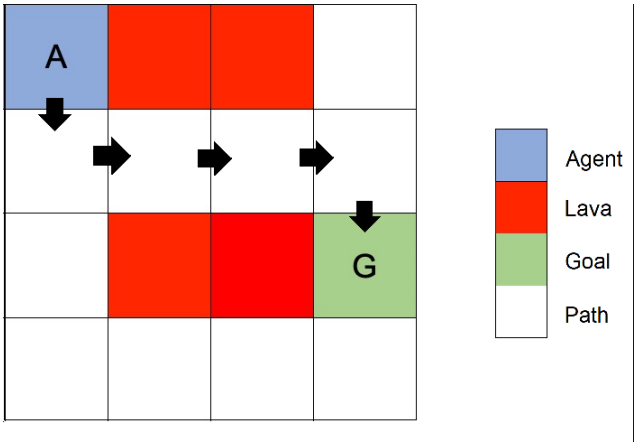

Figure 12. The best path found for Lava World 2.

6.2.4 Interrupt World

The deep Q-learning algorithm's return for Interrupt World is shown in Figure 13 and the best policy is in Figure 14. It is notable that the algorithm never discovered that the button would guarantee that the interrupt was safe, even when the interrupt was set to almost always cause a problem. Since the DQN failed to learn the best path here, we decided not to use this when trying to learn multiple tasks.

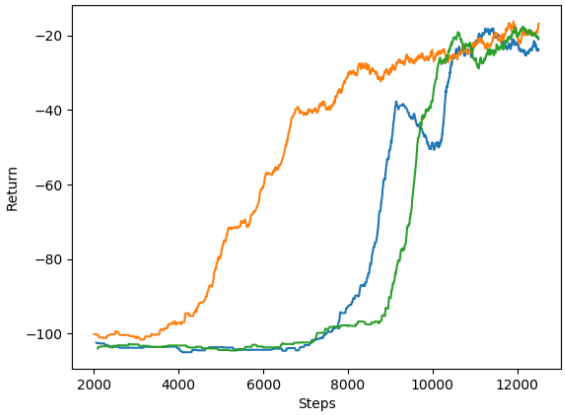

Figure 13. Return vs. timesteps for the Interrupt World. The plots are created like in Figure 7.

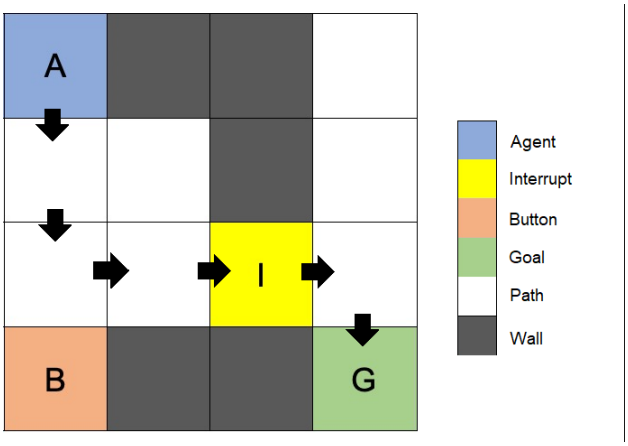

Figure 14. The best path found for Interrupt World. Notice how the optimal path would go to the button first (since the interrupt has a chance of causing the episode to end otherwise).

6.3. EWC Results

In order to test EWC, we first train on a task A and save the parameters. We then compute the Fisher Information Matrix as described above with a batch of 256 samples from our replay memory. Next, we attempt to train on task B using EWC as described in [2]. We load the model parameters from task A as a starting point and use the loss defined in equation 2. As mentioned previously, we choose to use Lava World 0 as task A and Lava World 1 as task B. This is because the same solution path won't work for both. The result is shown in Figures 15, 16, and 17. Additionally, the time for each run increases from 90 seconds to 110.

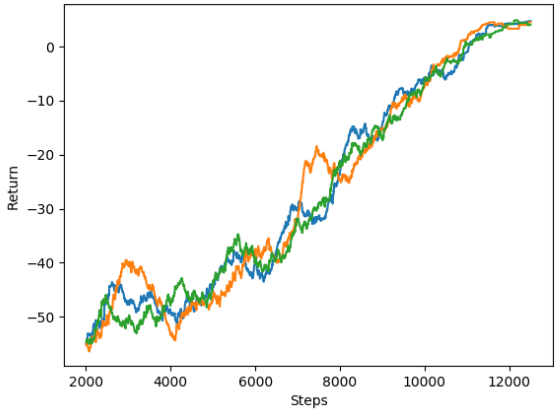

Figure 15. Moving average of return vs. timesteps for three trials of EWC as it learns Task B. From the figure, we can see that it successfully learns Task B near the end.

Although the model succesfully learns task B, it forgets task A and makes a mistake which is shown in Figure 17. So, why doesn't EWC work? The solution above has task B solved and then on task A it walks partially to task A's goal and jumps in the lava. Why? If we look at task B's world, it was avoiding the lava in row 3, column 3 (which didn't exist in task A).

The EWC loss displays an interesting pattern. Figure 16 shows how it starts at 0 when the parameters of the model are the same as the solution to task A. As the model begins to learn task B, the EWC loss goes up as the task B loss, $\mathcal{L}_B(\theta)$, is quickly decreased. After the task B loss has been minimized, the EWC term comes into play and is minimized towards zero.

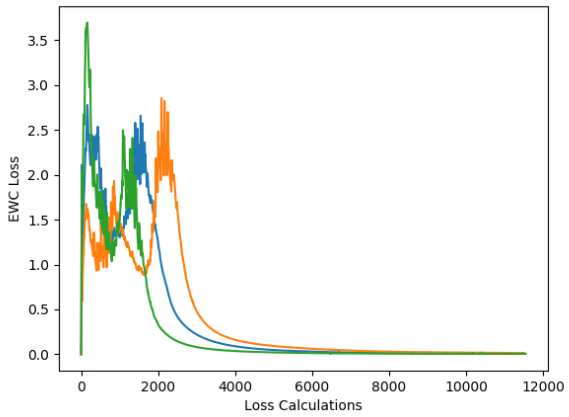

Figure 16. The EWC loss vs. the number of times the loss has been computed (this happens roughly every timestep). This corresponds to Figure 15. Note that EWC loss refers to the summation term on the right-hand side of Equation 2.

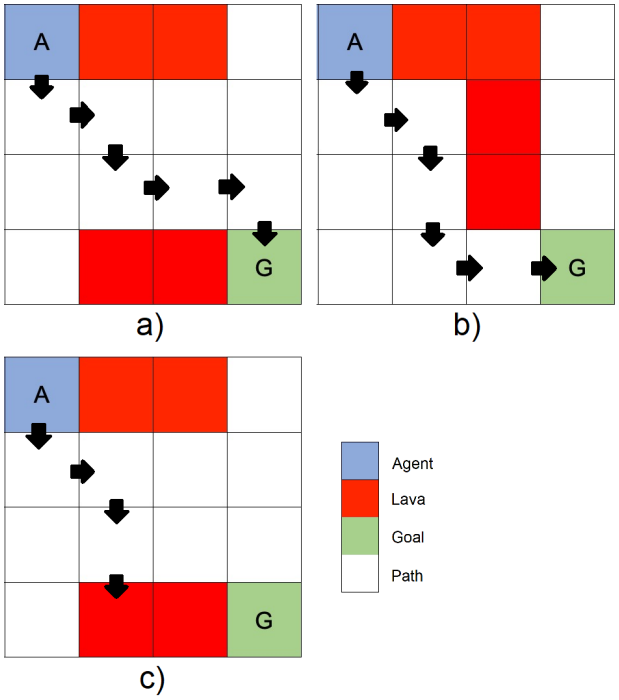

Figure 17. The best paths found using EWC. a) The best path found for task A. b) The best path found for task B. c) Path taken on task A after learning task B.

In the original catastrophic forgetting paper [2], the solution quality on task A eventually degrades. This is clear because the quality of the solution for task A decreases by some percentage. In a discrete gridworld, one move off the path breaks the identified solution. In something like ATARI or MNIST that relies on image data, the network and policy is more receptive to small perturbations, and can recover more easily. For example, in breakout if the paddle moves the wrong way on one frame, then the ball will move and it might then move the correct way on the next few frames. This doesn't apply in gridworld, where if you get stuck you never reach the goal. Since these gridworld solutions end up with a deterministic policy that produces one solution, if we wander off the path at only one point the agent fails.

Another critical reason for failure is that the gradient of the model parameters is zero for some values. This causes the empirical Fisher Information Matrix diagonal to have some zero values in it. This makes some sense, since we use a one-hot encoding as input so there are unused weights. However, this phenomenon seems to occur even on weights which were used in task A. Since F_i is 0, those weights to be changed without being regularized in task B even if they might matter to task A. Looking at Equation 2, we can see that this will cause the penalty for that parameter to be zero no matter how much the weight changes from the original solution.

6.4. EWC Investigation

Several changes were attempted to get EWC to work. Unfortunately, none had a significant impact on performance. The results of this investigation are shown in the following sections.

6.4.1 Changing Network Capacity

We attempted to change the capacity of the network to allow freedom to hold more solutions. The results of this can be seen in Figure 18 and Table 2. These results establish that network capacity is unlikely to be the cause of failing to retain task A. We can also see that shrinking the hidden layers too much can cause them to fail to learn at all (for 10,10). We continue to use (100, 100) since it is used in [1] and provides additional capacity for retaining more patterns than the smaller models.

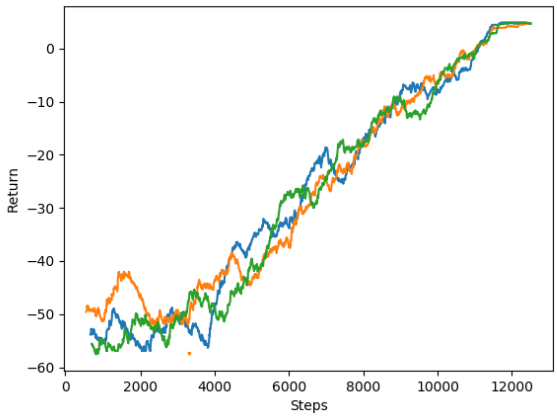

Figure 18. Moving average of return vs. timesteps on task B for three trials of EWC with each hidden layer having 200 hidden nodes.

Table 2. Table of attempted networks

Hidden layer 1 size	Hidden layer 2 size	Outcome
100	100	Failure to retain A
200	200	Failure to retain A
10	10	Failure to learn A/B
20	10	Failure to retain A
20	20	Failure to retain A

6.4.2 Different Batch Sizes for Calculating Fisher

We next tried to increase the batch size for calculating the empirical fisher information matrix. We tried increasing it from 256 to 512 and 1,000, but this produced no noticeable results that we could measure.

6.4.3 Change the Learning Rate

We noticed that EWC loss could be jumpy, so we next tried to reduce the learning rate on task B from 5e-4 to 5e-5. Unfortunately, this also didn't work. Since the learning rate was reduced, we tried to learn for 50,000 timesteps instead, as presented in Figure 19, which also failed to retain task A. The loss is successfully smoothed, but the model still fails to retain task A. It's also interesting to note the bimodal distribution that appears to form in all three distinct trials. The reason for this might be related to the optimizer we used,

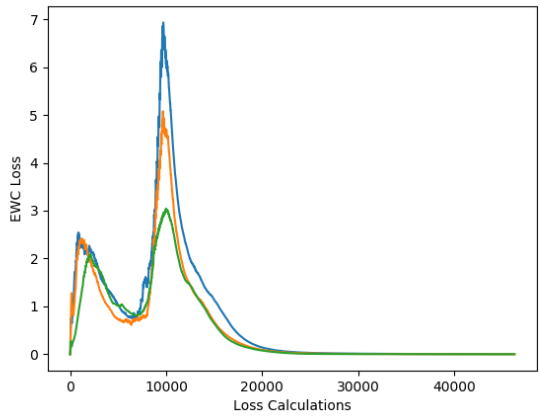

Figure 19. EWC curve of learning rate at 5e-5. Note that we used 50,000 timesteps here instead of 12,500.

Adam. The model might settle into one basin of attraction (minima) and start minimizing the EWC loss in that but then escape to minima with which temporarily increases the EWC loss again. It isn't clear why this tends to happen only twice.

6.4.4 Changing Step Reward

One notable difference between the safety gridworlds paper and our gridworlds is that we use a reward of -5 instead of -1 for each step. There is a clear reason behind this. We initially started with -1, and then we tested -3 and -5. Our model attempts to approximate the value for each action, this means that the difference between states will only be -1. An action that moves into the goal will have an approximation of 30. The action that moves into the state next to the goal will have an approximation of 29. If we use -5 instead, the second action will instead have a value of 25. The aim of increasing the penalty for an action here is to widen the distance between these action values. Ideally this would make the network more secure against perturbations, since it takes more changes to a weight to cause it to shift a value by 5 rather than 1. Each state has four action Q-values, and we pick the highest expected return. We had hoped that if these four values were farther apart then changing the weights wouldn't change which one is the highest Q-value. If the highest Q-value changes, then the agent moves in the wrong direction so it can no longer reach the goal of task A. We hoped that this would help prevent actions in task A states from being changed. Since we either fail or succeed at reaching the goal (and we didn't succeed at reaching it for task A after learning B), we cannot measure whether doing this helped.

6.4.5 Change the EWC Weight

The λ hyperparamter for EWC controls how well we should remember task A. In [2] they empirically determined a value of 400. We also used this value for most of our experiments.

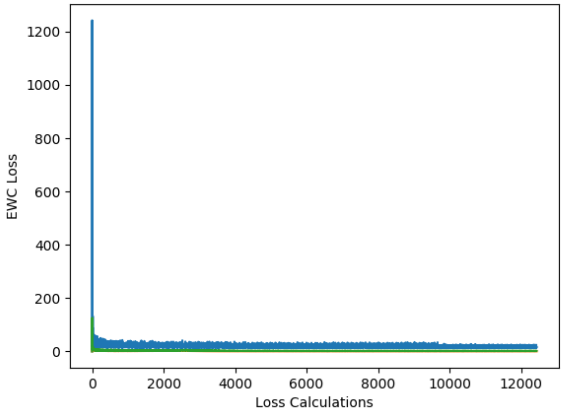

Figure 20. EWC curve with weight $\lambda = 1000$. The figure is difficult to decipher since larger λ value causes the function to resemble a corner, but it resembles the other EWC curves above.

Since we were failing to retain task A, we tried to increase the value of λ to 1,000 (shown in Figure 20) and then 10,000. These both failed to retain A and learn B. The zero gradient observed previously means that changing the weight λ just doesn't matter for some parameters.

6.4.6 Use Minimum Value in Fisher Information Matrix

In order to address the problem we found with the zero gradient values, we implement the following equation to modify the fisher information matrix:

$$F_i = max(F_i, \alpha) \; \forall \text{ parameters } i \tag{3}$$

We tested this with a combination of α and λ values, without any significant luck on any combination as can be seen in Table 3 and Figures 21 - 26. Values of α

Table 3. Trials using minimum fisher value

λ	α	Result
400	1	failed A failed B
1,000	1	failed A failed B
10,000	1	retained A failed B
10,000	1	retained A failed B
100	.01	failed A learned B
150	.01	failed A learned B
200	.01	failed A failed B
300	.01	failed A learned B
5000	.01	failed A failed B
7500	.01	failed A failed B
10,000	.01	failed A failed B
100,000	.01	failed A failed B

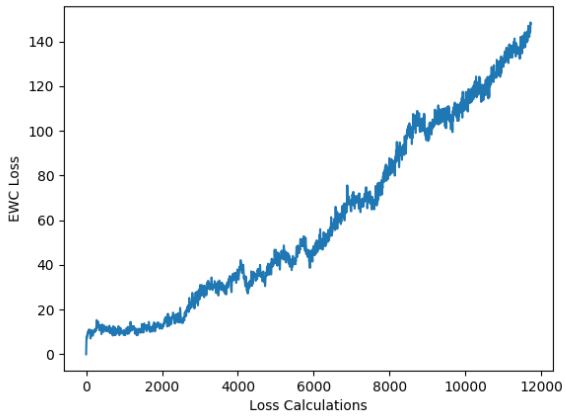

Figure 22. EWC vs. number of optimizations done corresponding to Figure 21.

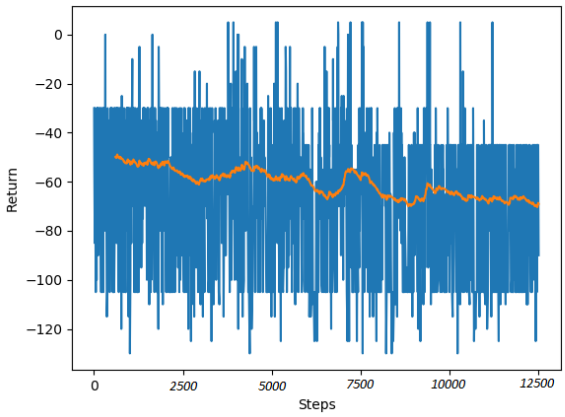

Figure 21. This shows the the return on task B for the first hyperparameter combination (λ = 400 and α = 1). It fails to learn task B and forgets A. Since we only tried each of these hyperparameter combinations once due to time constraints, we also show the return (blue) along with the moving average (orange).

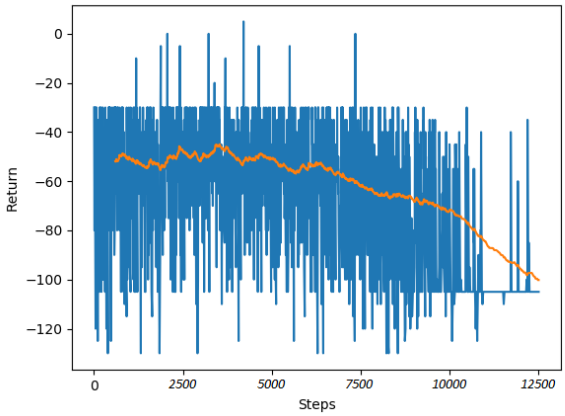

Figure 23. This shows the the return on task B for the only hyperparameter combination (λ = 10,000 and α = 1) which doesn't forget task A. Notice that the returns get worse as the $epsilon$ value decreases since we aren't taking as many random actions.

and λ are multiplied together in Equation 2, so they both have very similar affects on the loss (larger values try to retain task A more). This can be seen in Figure 23 since the average return actually decreases. The EWC loss values in the corresponding Figure 24 are remarkably low given the hyperparameter combination, and the jitter is likely because the learning rate is too high for the loss function with such large hyperparameters. Another interesting trial is in Figure 25 and 26. This trial was run for 12,500 more timesteps without changing ϵ, and it shows interesting periodic behavior as well as showing that more time doesn't yield improved performance. It fails to have learned task B at the end, although the model appears to learn it and then forget, which indicates that the solution may not be completely stable.

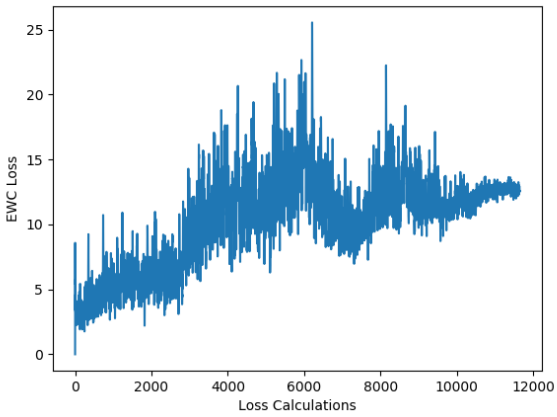

Figure 24. EWC vs. number of optimizations done corresponding to Figure 23.

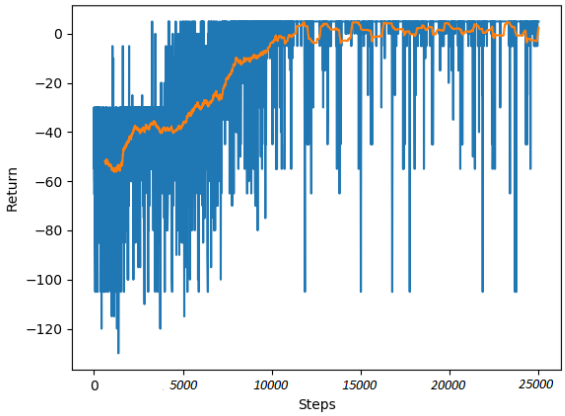

Figure 25. This shows the the return on task B for the hyperparameter combination ($\lambda = 200$ and $\alpha = 0.01$). It is run for 12,500 more timesteps with $\epsilon = \epsilon$-end.

6.5. Naive Multitask Approach

As a proof that the network we use is capable of learning to solve two tasks, we implement what we call a naive multitask approach to the problem. We randomly selected one of two environments with equal probability for each episode and learned otherwise normally. We successfully learned both problems on Lava World 0 and 1. Although this is not the same as most multitask approaches (which often have separate loss functions for each task), it is similar in the fact that it tries to learn the two tasks at once. The optimal

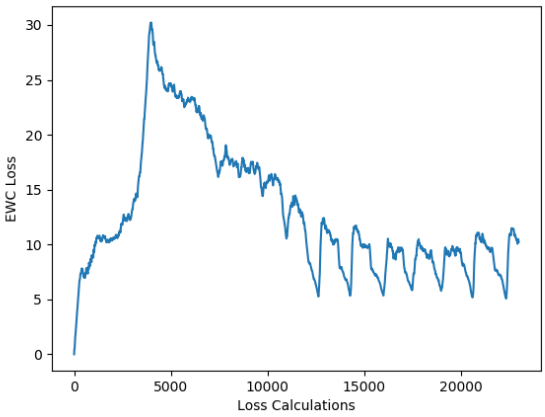

Figure 26. EWC vs. number of optimizations done corresponding to Figure 25.

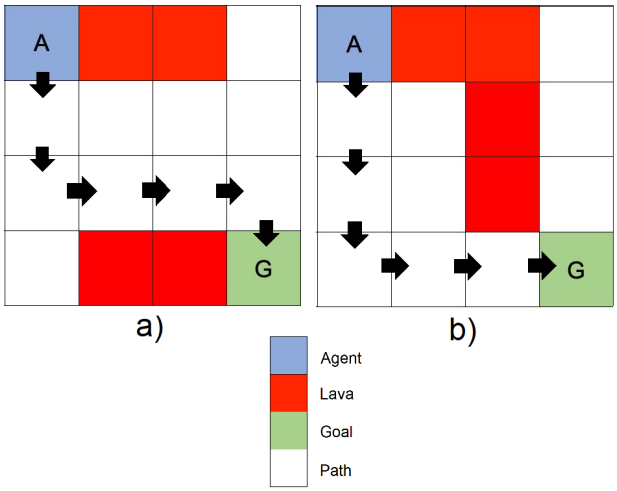

Figure 27. Example of optimal paths found using the naive multitask approach. a) Optimal path found for task A. b) Optimal path found for task B.

paths found for each world can be seen in Figure 27. This occurred in the same number of timesteps we used for training previously, which indicates that finding the best path for one task might help learn the other task. Figure 28 shows the average return for five trials which all successfully learn both tasks. This shows that our network does in fact have the capacity to hold solutions for multiple tasks.

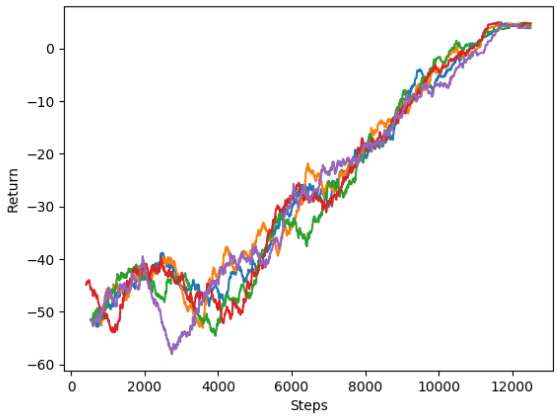

Figure 28. Average return vs. timestep for 5 distinct trials of the naive multitask approach.

7. Conclusions

In this project we implemented DQN with EWC in an attempt to solve 2 AI Safety Gridworld environments using a single network. Knowing that these environments were going to be difficult to learn in, we were unsure of how realistic a robust solution would be. Although this project was overall unsuccessful, several things were learned.

First, we discover that grid world may not be well-suited for deep Q-learning (nontabular) approaches due to being very discrete. One-hot inputs may actually make this worse. In the safety paper, it is clear that even Rainbow struggles somewhat, and doesn't reach the optimal solution. This is amplified in our problem using an unimproved DQN. Additionally, small perterbations in the network can easily break the solution, since even taking one action wrong causes problems (e.g. we jump into lava). Tabular approaches are likely to be much better at solving gridworld problems. However, they cannot be used for learning multiple tasks with EWC, which is why we didn't explore their use.

Second, we discover that the naive approach is able to learn both tasks. Although this learning is not done in a manner that matches our problem goal, this shows that EWC is able to maintain relevant information necessary to perform both tasks. With further exploration into the various parameters or revised task sampling order, EWC may thus potentially be a feasible solution. Due to our time constraints, however, we were not able to further explore this.

8. Appendix

Carl Edwards was primarily responsible for leading software development and parameter exploration / problem definition. Brandon Mathis was primarily responsible for leading development of supporting software and documentation.

References

[1] J. Leike, M. Martic, V. Krakovna, P. A. Ortega, T. Everitt, A. Lefrancq, L. Orseau, and S. Legg, "Ai safety gridworlds," 2017.

[2] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[4] R. Kemker, M. McClure, A. Abitino, T. L. Hayes, and C. Kanan, "Measuring catastrophic forgetting in neural networks," in *Thirty-second AAAI conference on artificial intelligence*, 2018.

[5] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[6] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng, "On optimization methods for deep learning," in *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ser. ICML'11. USA: Omnipress, 2011, pp. 265–272. [Online]. Available: http://dl.acm.org/citation.cfm?id=3104482.3104516

[7] Stephanie, "Fisher information / expected information: Definition," Sep 2018. [Online]. Available: https://www.statisticshowto.datasciencecentral.com/fisher-information/

[8] F. Kunstner, L. Balles, and P. Hennig, "Limitations of the empirical fisher approximation," *arXiv preprint arXiv:1905.12558*, 2019.

[9] J. Duchi. [Online]. Available: https://web.stanford.edu/class/stats311/

[10] A. Pazke, "Reinforcement learning (dqn) tutorial." [Online]. Available: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

[11] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.