Supplemental Material to Dynamic Weights in Multi-Objective Deep Reinforcement Learning

1. Algorithms

In this section of the appendix, we first present some of the recent advances in Deep RL we extended to multi-objective Deep RL and then include the pseudo-code for Conditioned Network (CN), Multi-Network (MN) and Diverse Experience Replay (DER) algorithms.

1.1. Prioritized Sampling

For both replay buffer types, we used proportional prioritized sampling (also referred to as Prioritized Experience Replay (Schaul et al., 2015)). This technique replaces the uniform sampling of experiences for use in training by prioritized sampling, favouring experiences with a high TD-error (i.e., the error between their actual Q-value and their target Q-value). To update each sample's priority in the dynamic weights setting, we observe that TD-errors will typically be weight-dependent. It follows that a priority can be overestimated if the TD-error is large for the weight on which the sample was trained but otherwise low, or it can be underestimated if the TD-error is low for the trained weight but otherwise high. In the first case, the sample is likely to be resampled quickly and its TD-error will be re-evaluated. Hence we consider the overestimation to be reasonably harmless[1]. In contrast, underestimating a sample's TD-error can have a more significant impact because it is unlikely to be resampled (and thus re-evaluated) soon. To alleviate this problem, we used Prioritized experience replay's ε parameter which offsets each error by a positive constant ε; $p(\delta) = (\delta + \varepsilon)^{\alpha}$. This increases the frequency at which low-error experiences are sampled allowing for possibly underestimated experiences to be re-evaluated reasonably often. As a result, on average, experiences that get sampled less often are samples that consistently have low TD-errors for all weight vectors used in training.

The question then remains which TD-error should be used for a given sample. For both the MO baseline and MN we update the priority w.r.t. the TD-error on the active weight vector.

[1]We further note that a given sample can only be overestimated often if it repeatedly has a large TD-error for the weights it is being trained on, in which case it should not be considered as overestimated.

We find this to be insufficient for CN as it trains both on the active weight vector and on randomly sampled past weight vectors. Ideally we would compute a TD-error relative not only to the active weight-vector but also all the past weight vectors. However, it would be too computationally expensive to perform a forward pass of each training sample on all encountered weights, so we only consider the two weight vectors (i.e., $\mathbf{w}_t$ and $\mathbf{w}_j$) it was last trained on. Only using the active weight vector's TD-error to determine the priority would prevent past policies from being maintained, as their TD-error would have no influence on how often experiences are trained on. Conversely, only taking the randomly sampled weight vector in consideration could hurt convergence on the active weight vector's policy. Hence we balance current and past policies by computing the average of both TD-errors and use that value to determine the experience's priority.

1.2. Double DQN

Double DQN (Van Hasselt, Guez, and Silver, 2016) reduces Q-value overestimation by using the online network for action selection in the training phase. I.e., $y_j = r_j + \gamma Q^{-}(argmax_{a'} Q(a', s_{j+1}), s_{j+1})$ instead of $y_j = r_j + \gamma max_{a'} Q^{-}(a', s_{j+1})$. As a result, an action needs to be overestimated by both the target and the online network to cause the feedback loop that would occur in standard DQN. The same technique can be used in multi-objective DQNs. It is especially useful for the Multi-Network algorithm, as overestimated Q-values can have a significant impact on policy selection.

1.3. Conditioned Network Algorithm

The Conditioned Network (CN) algorithm (Algorithm 1) for multi-objective deep reinforcement learning under dynamic weights, handles changes in weights (i.e., the relative importance of each objective) by conditioning a single network on the current weight vector, $\mathbf{w}$. As such, the $\mathbf{Q}$-values outputted by the network depend on which $\mathbf{w}$ is inputted, alongside the state. For an architectural overview of the networks we employed, please refer to Appendix 2.1.

After initializing the network, the agent starts interacting with the environment. At every timestep, first a weight

Algorithm 1 Dynamic Weight-Conditioned Reinforcement Learning

```
 1: Define $a^*_{\mathbf{w},s}$ as shorthand for $argmax_{a \in A}\mathbf{Q}_{CN}(a,s;\mathbf{w}) \cdot \mathbf{w}$
 2: initialize (diverse) replay buffer $\mathcal{D}$ and unique weight history $\mathcal{W}$
 3: $\mathbf{Q}_{CN}, \mathbf{Q}^-_{CN} \leftarrow initializeConditionedModel()$
 4: for steps $t \in \{0...T\}$ do
 5:   $\mathbf{w}_t \leftarrow getWeightVector(t)$
 6:   add $\mathbf{w}_t$ to $\mathcal{W}$
 7:   With probability $\varepsilon$ select a random action $a_t$
 8:   Otherwise $a_t = a^*_{\mathbf{w}_t,s_t}$
 9:   Execute action $a_t$ and observe $\mathbf{r}_t$ and $\mathbf{s}_{t+1}$
10:   Store transition $(s_t, a_t, \mathbf{r}_t, s_{t+1})$ in $\mathcal{D}$
11:   Sample minibatch of transitions from $\mathcal{D}$
12:   for each sampled transition $(s_j, a_j, \mathbf{r}_j, s_{j+1})$ do
13:     $\mathbf{w}_j$ randomly sampled from $\mathcal{U}(\mathcal{W})$
14:     if transition is terminal then
15:       $y_j = y'_j = \mathbf{r}_j$
16:     else
17:       $y_j = \mathbf{r}_j + \gamma\mathbf{Q}^-_{CN}(a^*_{\mathbf{w}_t,s_{j+1}}, s_{j+1}; \mathbf{w}_t)$
18:       $y'_j = \mathbf{r}_j + \gamma\mathbf{Q}^-_{CN}(a^*_{\mathbf{w}_j,s_{j+1}}, s_{j+1}; \mathbf{w}_j)$
19:     end if
20:   end for
21:   perform gradient descent step on
        $\frac{1}{2}\left[|y_j - \mathbf{Q}_{CN}(a_j, s_j; \mathbf{w}_t)| + |y'_j - \mathbf{Q}_{CN}(a_j, s_j; \mathbf{w}_j)|\right]$
22:   Every $N^-$ steps; $\mathbf{Q}^-_{CN} = \mathbf{Q}_{CN}$ {Synchronize target network}
23:   anneal($\varepsilon$)
24: end for
```

vector $\mathbf{w}_t$ is perceived, and added to the set of observed weights $\mathcal{W}$ if it is different from $\mathbf{w}_{t-1}$. $\mathcal{W}$ is used to sample historical weights from, so that the network keeps training with regards to both current and previously observed weights. This is necessary in order to make the network generalize over the relevant part of weight simplex. Specifically, for each gradient descent step, the target consists of two equally weighted components; one for the current weight $\mathbf{w}_t$ and one randomly sampled weight from $\mathcal{W}$, $\mathbf{w}_j$.

While not explicitly visible in the algorithm, CN makes use of prioritized experience replay. Please refer to (Schaul et al., 2015) for details. Each timestep, an experience tuple is perceived and added to the replay buffer $\mathcal{D}$. Then, a minibatch of transitions is sampled from $\mathcal{D}$, on which the network is trained. Each experience's priority is updated based on the average TD-error of the two weight vectors it was trained on.

As described in the main paper, CN can have a secondary experience replay buffer for diverse experience replay (DER). For a description of when and which samples are added to the secondary replay buffer, please refer to the main paper.

In this paper, we make use of ε-greedy exploration, with ε, the probability of performing a random action, annealed over time. For Minecart we anneal it from 1 to 0.05 over the first 100k steps, for the easier DST problem we anneal it to 0.01 over 10k steps. However, CN is compatible with any sort of exploration strategy.

1.4. Multi-Network algorithm

The Multi-Network (MN) algorithm (Algorithm 2) for multi-objective deep reinforcement learning under dynamic weights handles changes in weights by gradually building an approximate partial CCS, i.e., a set of policies such that each policy performs near optimality for at least one encountered weight vector.

The algorithm starts with an empty set of policies Π. Then, for each encountered weight vector $\mathbf{w}_t$, it trains a neural network through scalarized deep Q-learning (Mossalam et al., 2016). The differences with standard deep Q-learning are that the DQN's outputs are vector valued and that action selection is done by scalarizing these Q-vector-values w.r.t. the current weight vector $\mathbf{w}_t$ (Lines 2 and 2 of Algorithm 2).

As is the case for CN, experiences are sampled from the replay buffer through prioritized sampling, with priorities being computed on the TD-error for the active weight vector.

When the active weight vector changes at time $t+1$, the policy trained (before the change) for $\mathbf{w}_t$ is stored if it is optimal for at least one past weight vector. To account for approximation errors, a constant κ is subtracted from any past policy's scalarized value. Hence, when two scalarized values are within an error κ of each other, the more recent policy is favoured. Any policy in Π that is made redundant by the inclusion of the new policy trained on $\mathbf{w}_t$ is discarded.

Then, the policy with the maximal scalarized value for the new weight vector $\mathbf{w}_{t+1}$ is used as a starting point for its Q-network $\mathbf{Q}_{\mathbf{w}_{t+1}}$. As in (Mossalam et al., 2016), we considered full re-use, where all parameters of the model Q-network are copied into the new Q-network and partial re-use, in which all but the last dense layer were copied to the new Q-network. We found that the latter performed poorly and therefore only considered full re-use in this paper.

1.5. Diverse Experience Replay

We now present our implementation of Diverse Experience Replay (DER, Algorithm 3).

We maintain both a first-in first-out replay buffer and a diverse replay buffer. Experiences are added to the FIFO buffer as they are observed. When the FIFO buffer is full, the oldest trace τ is removed from it and considered for

Algorithm 2 Dynamic Multi-Network Reinforcement Learning

```
1: initialize (diverse) replay buffer $\mathcal{D}$ and unique weight history $\mathcal{W}$
2: $\kappa \leftarrow$ Improvement constant
3: $\Pi \leftarrow$ empty set of $(\mathbf{Q_w}, \mathbf{w}, \mathbf{V_w})$ tuples {With $\mathbf{w}$ a weight vector, $\mathbf{Q_w}$ a policy for that weight vector (i.e., a multi-objective Q-network), $\mathbf{V_w}$ the stateless value vector of the policy}
4: $\mathbf{w}_0 \leftarrow getWeightVector(0)$
5: add $\mathbf{w}_0$ to $\mathcal{W}$
6: $\mathbf{Q}_{\mathbf{w}_0}, \mathbf{Q}^-_{\mathbf{w}_0} \leftarrow initializeFirstModel()$
7: for steps $t \in \{0...T\}$ do
8:     With probability $\varepsilon$ select a random action $a_t$
9:     Otherwise $a_t = argmax_{a \in A} \mathbf{Q}_{\mathbf{w}_t}(a, s) \cdot \mathbf{w}_t$
10:    Execute action $a_t$ and observe $\mathbf{r}_t$ and $\mathbf{s}_{t+1}$
11:    Store transition $(s_t, a_t, \mathbf{r}_t, s_{t+1})$ in $\mathcal{D}$
12:    Sample minibatch of transitions from $\mathcal{D}$
13:    for each sampled transition $(s_j, a_j, \mathbf{r}_j, s_{j+1})$ do
14:        if transition is terminal then
15:            $y_j = \mathbf{r}_j$
16:        else
17:            $a'_j = \text{argmax}_{a'} \mathbf{Q}_{\mathbf{w}_t}(a', s_{j+1}) \cdot \mathbf{w}_t$
18:            $y_j = \mathbf{r}_j + \gamma \mathbf{Q}^-_{\mathbf{w}_t}(a'_j, s_{j+1})$
19:        end if
20:    end for
21:    perform gradient descent step on
           $$[|y_j - \mathbf{Q}_{\mathbf{w}_t}(a_j, s_j)|]$$
22:    Every $N^-$ steps; $\mathbf{Q}^-_{\mathbf{w}_t} = \mathbf{Q}_{\mathbf{w}_t}$ {Synchronize target network}
23:    anneal($\varepsilon$)
24:    $\mathbf{w}_{t+1} \leftarrow getWeightVector(t)$
25:    if $\mathbf{w}_t \neq \mathbf{w}_{t+1}$ then
26:        if $\exists \mathbf{w} \in \mathcal{W} : \mathbf{V}_t \cdot \mathbf{w} > \max_{\mathbf{V}' \in \Pi} \mathbf{V}' \cdot \mathbf{w} - \kappa$ then
27:            add $(\mathbf{Q}_{\mathbf{w}_t}, \mathbf{w}_t, \mathbf{V_t})$ to $\Pi$
28:            remove policies made redundant by $\mathbf{Q}_{\mathbf{w}_t}$
29:        end if
30:        $\mathbf{Q}_{\mathbf{w}'}, \mathbf{w}', \mathbf{V}_{\mathbf{w}'} \leftarrow argmax_{(\mathbf{Q}_{\mathbf{w}'}, \mathbf{w}', \mathbf{V}_{\mathbf{w}'}) \in \Pi} \mathbf{w} \cdot \mathbf{V}_{\mathbf{w}'}$ {pick a policy to continue learning from}
31:        $\mathbf{Q}_{\mathbf{w}_{t+1}}, \mathbf{Q}^-_{\mathbf{w}_{t+1}} \leftarrow copyModel(\mathbf{Q}_{\mathbf{w}'})$ {Partial or full re-use}
32:        add $\mathbf{w}_{t+1}$ to $\mathcal{W}$
33:    else
34:        $\mathbf{Q}_{\mathbf{w}_{t+1}}, \mathbf{Q}^-_{\mathbf{w}_{t+1}} \leftarrow \mathbf{Q}_{\mathbf{w}_t}, \mathbf{Q}^-_{\mathbf{w}_t}$ {continue training same policy}
35:    end if
36: end for
```

memorization into the secondary diverse replay buffer.

The trace τ is only added to the secondary buffer if it increases the replay buffer diversity. To determine this, we first compute a signature for each trace up for consideration (i.e., τ and all traces already present in the diverse replay buffer $\mathcal{D}'$). Note that this signature can typically be computed in advance. Next, a diversity function d computes the relative diversity of each signature w.r.t. all other considered signatures (Algorithm 3 Line 3). If τ's relative diversity is lower than the minimal relative diversity already present in the secondary buffer $\mathcal{D}'$, it is discarded. Otherwise, the trace that contributes least to the buffer's diversity is removed from $\mathcal{D}'$ to make place for τ.

This process is repeated until there is enough space for τ in the diverse buffer or τ has a lower diversity than the lowest diversity trace in $\mathcal{D}'$, in which case τ is discarded and the traces that were removed during the current selection process are re-added.

For our experiments, we used a trace's return vector $\sum_{t=0}^{|\tau|} \gamma^t \mathbf{r}_t$ as its signature and the crowding distance (Deb et al., 2002) as the diversity function.

When using DER, half of the buffer size is used by the diverse replay buffer. When sampling from DER, no distinction is made between diverse and main replay buffers.

Algorithm 3 Diverse Replay Buffer

```
1: {With $s$ a signature function, $d$ a diversity function, $\mathcal{D}$ the main memory, $\mathcal{D}'$ the secondary memory and $e$ an experience to memorize}
2: if main memory $\mathcal{D}$ is full then
3:     extract oldest trace $\tau$ from $\mathcal{D}$
4:     add $e$ to $\mathcal{D}$
5:     while $\mathcal{D}'$ does not have enough space for $\tau$ do
6:         $\mathcal{F} \leftarrow d(\{s(\tau_i) | \tau_i \in \mathcal{D}' \cup \{\tau\}\})$
7:         select trace $\tau_j$ with lowest diversity $f_j \in \mathcal{F}$
8:         if $\tau_j \neq \tau$ then
9:             remove $\tau_j$ from $\mathcal{D}'$
10:        else
11:            Discard $\tau$
12:            undo deletions
13:            return
14:        end if
15:    end while
16:    add $\tau$ to $\mathcal{D}'$
17: end if
```

2. Implementation details

We now present our implementation details. More specifically, we give a table of hyperparameters (Table 1) and a full description of the network architecture.

2.1. Network Architecture

Figure 1 gives a schematic representation of the architecture we used in our experiments.

Our network contains more dense layers than single-objective Dueling DQN, we justify this by the need to output multi-objective Q-values and to either (1) output precise Q-values (in the case of MN knowing one action is

Table 1: Hyperparameters

General parameters (Minecart)	
Exploration rate	$1 \rightarrow 0.05$ over 100k steps
Buffer size	100.000
Frame skip	4
Discount factor	0.98
General parameters (DST)	
Exploration rate	$0.1 \rightarrow 0.01$ over 10k steps
Buffer size	10.000
Frame skip	1
Discount factor	0.95
Optimization parameters	
Batch size	64 (Minecart), 16 (DST)
Optimizer	SGD
Learning rate	0.02
Momentum	0.90
Nesterov Momentum	true
N^-	150
Prioritized sampling parameters	
ε	0.01
α	2.0

better than another is not sufficient), or (2) learn multiple weight-conditioned policies (in the case of CN).

The input to the network consists of two 48x48 frames (scaled down from the original 480x480 dimensions). The first convolution layer consists of 32 6x6 filters with stride 2. The second convolution layer consists of 48 5x5 filters with stride 2. Each convolution is followed by a maxpooling layer. A dense layer of 512 units is then connected to each temporal dimension of the convolution.

Following a multi-objective generalization of the *Dueling DQN* architecture (Wang, de Freitas, and Lanctot, 2015), this layer's outputs are then fed into the advantage and value streams, consisting of dense layers of size 512. The advantage stream is then fed into a layer of $|A| \times N$ dense units, while the value stream is fed into a dense layer of N units. These $|A| + 1 \times N$ outputs are then combined into $|A| \times N$ outputs by a multi-objective generalization of Dueling DQN's module.

$$\mathbf{Q}(s,a;\theta,\alpha,\beta) = \mathbf{V}(s;\theta,\beta) + \Big(\mathbf{A}(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'}\mathbf{A}(s,a';\theta,\alpha)\Big) \quad (1)$$

α and β denote the parameters of the advantage stream and of the value stream and θ denotes the parameters of all preceding layers. For the Conditioned Network, an additional parameter $\mathbf{w}$ is added to each function (corresponding to

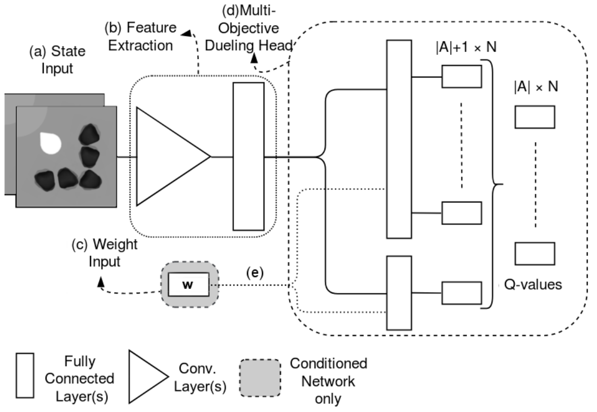


Figure 1: Features are extracted from the raw input by convolutional layers followed by a fully connected layer. The extracted features (output of (b)) are fed into a Multi-Objective Dueling DQN head (d). The conditioned architecture feeds a weight input (c) into the Q-value head (link (e)).

the weight input, link (e) in Figure 1);

$$\mathbf{Q}(s,a,\mathbf{w};\theta,\alpha,\beta) = \mathbf{V}(s,\mathbf{w};\theta,\beta) + \Big(\mathbf{A}(s,a,\mathbf{w};\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'}\mathbf{A}(s,a',\mathbf{w};\theta,\alpha)\Big) \quad (2)$$

The hyperparameters used for optimization are given in Table 1.

3. Test Problems

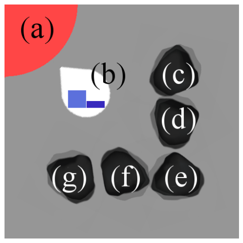
In this section, we present the test problems used in our experimental evaluation in greater detail.

3.1. Minecart Problem

The Minecart problem models the challenges of resource collection, has a continuous state space, stochastic transitions and delayed rewards.

The Minecart environment consists of a rectangular image, depicting a base, mines and the minecart controlled by the agent. A typical frame of the Minecart environment is given in Figure 2 (left). Each episode starts with the agent on top of the base. Through the *accelerate*, *brake*, *turn left*, *turn right*, *mine*, or *do nothing* (useful to preserve momentum) actions, the agent should reach a mine, collect resources from it and return to the base to sell the collected resources.

The reward vectors are N-dimensional: $\mathbf{r} = (r_1, ..., r_N)$. The first $N-1$ elements correspond to the amount of each of the $N-1$ resources the agent gathered, the last element is the consumed fuel. Particular challenges of this environment are the sparsity of the first $N-1$ components of the



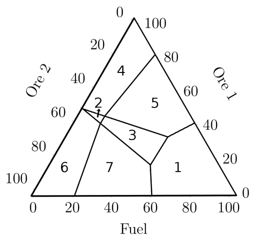


Figure 2: (left) Instance of the Minecart environment with 5 mines ((c) to (g)) containing varying amounts of 2 ores. The 2 bars on the minecart (b) indicate how much of each ore is present in the cart. Ores are sold on the base (a). (right) Weight vectors in the same region share the same optimal policy. Axes are the relative importance in % of each objective. We distinguish (1) collecting no resources if the fuel cost is too high, (6,7) privileging ore 2, (4,5) privileging ore 1, and (2,3) privileging the quick collection of either ore. Differences between each pair lies in the higher fuel cost, in which case it is optimal to accelerate less.

reward vector, as well as the delay between actions (e.g., mining) and resulting reward. The resources an agent collects by mining are generated from the mine's random distribution, resulting in a stochastic transition function. All other actions result in deterministic transitions. The weight vector $\mathbf{w}$ expresses the relative importance of each objective, i.e., the price per resource.

The underlying state consists of the minecart's position, its velocity and its content, the position of the mines and their respective ore distribution. While the implementation makes these available for non-deep MORL research, these properties were not used in our experiments. In the deep setting the agent should learn to extract them from the visual representation of the state.

Figure 2 shows the visual cues the agent should exploit to extract appropriate features of the state. First and most obviously, the position of the mines (black) and the home position (area top left). Second, indicators about the minecart's content are represented by vertical bars on the cart, one for each ore type. Each bar is the size of the cart's capacity. When the cart has reached its maximal capacity C, and mining will have no effect on the cart's content but still incur the normal mining penalty p_m in terms of fuel consumption. At that point the agent should return back to its home position. Additionally, the minecart's orientation is given by the cone's direction. Accelerating incurs a penalty of p_a in terms of fuel consumption. In addition, every time-step the agent receives a penalty in the fuel objective p_i representing the cost of keeping the engine running.

The default configuration of the minecart environment we used in our experiments is given in Table 2. The setting contains 5 mines, with distribution means for ores 1 and 2 given in Table 3, and a standard deviation fixed at $\sigma = 0.05$.

Table 2: Minecart configuration

General Minecart Configuration	
Cart capacity	1.5
Acceleration	0.0075
Ores	2
Rotation angle	10 degrees
Fuel component rewards	
Idle cost p_i	-0.005
Mining cost p_m	-0.05
Acceleration cost p_a	-0.025

Table 3: Ore distribution per mine, if either ore is more valuable, mining from (d) to (f) results in wasted capacity on the less valuable ore. Hence, while the average content collected from these mines is higher, they are not always optimal because of the limited cart capacity.

Mine	(c)	(d)	(e)	(f)	(g)
μ_{ore_1}	0.2	0.15	0.2	0.1	0.
μ_{ore_2}	0.	0.1	0.2	0.15	0.2

Optimal Policies For $\gamma = 0.98$ used in our experiments, this configuration divides the weight-space into 7 regions according to their optimal policies as shown in Figure 2. The 7 policies are;

1. do not collect any resources
2. go to mine (e) quickly and mine until full,
3. go to mine (e) slowly and mine until full,
4. go to mine (c) rapidly and mine until full,
5. go to mine (c) slowly and mine until full,
6. go to mine (g) quickly and mine until full,
7. go to mine (g) slowly and mine until full,

3.2. Deep Sea Treasure

In the Deep Sea Treasure (DST) problem (Vamplew et al., 2011), a submarine must dive to collect a treasure. The further the treasure is, the higher its value. The agent can move *left*, *right*, *up*, and *down* which will move him to the corresponding neighboring cell unless that cell is outside of the map or a sea bottom cell (black cells). The reward signal an agent perceives consists of a treasure component and a time component. The submarine collects a penalty of -1 for its second objective at every step. When it reaches a treasure, the treasure's value is collected as a reward for the first objective, and the episode ends. As the original DST problem has only two policies in its convex coverage set, we used a modified version of the DST map – given in Figure 3 – in our experiments.

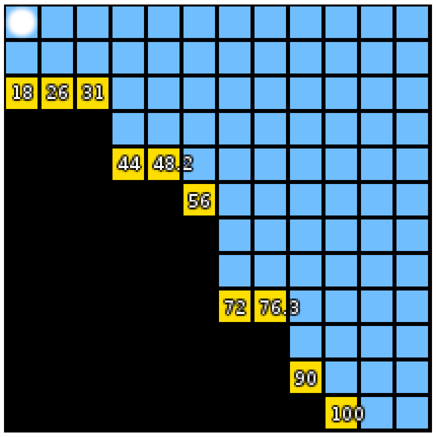


Figure 3: DST map, yellow squares indicate treasures and their value, the agent is marked by a white circle. Black areas are the ocean floor, blue areas are the ocean.

This map was designed such that, for a discount factor of 0.95, each treasure is the goal of an optimal policy in the CCS (Figure 4). And in addition, each policy in the CCS has approximately the same proportion of weights for which it is optimal ($\sim 10\%$ of weight vectors for each policy, Figure 5). The full results obtained for the image version DST are given in Table 4.

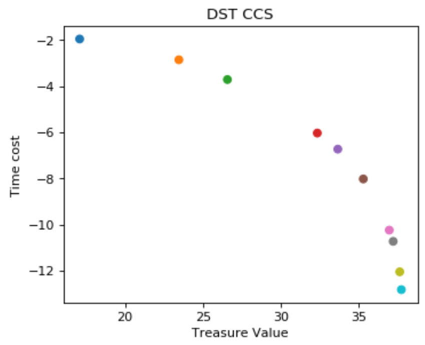


Figure 4: Convex Coverage Set for the given DST map and a discount factor of $\gamma = 0.95$.

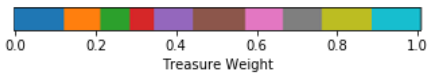


Figure 5: Optimal policy colormap, each color corresponds to one of the optimal policies in Figure 4. Time cost weight is $1-$ treasure weight.

4. Additional Results

In this section we give the complete results table for DST (Table 4), and we experimentally compare selective experience replay (Isele and Cosgun, 2018) and Exploration-based selection (De Bruin et al., 2018) to DER on the Minecart problem. We also provide results for a naive baseline (NAIVE) suggest by (Liu, Xu, and Hu, 2015).

4.1. Naive algorithm

The naive algorithm suggested by (Liu, Xu, and Hu, 2015) learns optimal Q-values for each objective then selects actions by scalarizing these multiple single-objective Q-values can learn to perform well for edge weight vectors (i.e., weight vectors for which one objective is much more important than the others). However, when a trade-off is required between objectives it would be unable to perform optimally.

4.2. Exploration-based Selection

De Bruin et al. (2018) propose an alternative to FIFO memorization based on how exploratory a transition's action a_t is. The distance between the Q-value of the optimal action a_t^* in state s_t and the taken action a_t are used as a diversity metric. Hence actions that differ strongly from the optimal action are more likely to be preserved in the replay buffer. The main obstacle to this approach working in this setting is that the exploratory nature of an action is likely to be dependent on the weight vector. An action that would be exploratory for a weight vector $\mathbf{w}$ could be optimal for another weight vector $\mathbf{w}'$. We found that while this metric can be useful for a single weight vector, it proves unreliable when used across different weight vectors. In addition, we identified the long-term dependence there can be between experiences. In complex problems, a reward is typically the result of a long sequence of actions. Hence, while exploration-based selection might permanently store an interesting experience, it is unlikely to store all the experiences leading to that experience. In contrast, our approach handles trajectories as atomic units. Hence, if a rewarding experience is stored, the actions leading to that reward will be stored too.

4.3. Selective Experience Replay

Selective experience replay (Isele and Cosgun, 2018) was recently proposed to prevent catastrophic forgetting in single-objective multi-task lifelong learning. In this setting, an agent must learn to perform well on a sequence of tasks and maintain that performance while learning new tasks. As a result the replay buffer can be biased towards the most recent task. From there, a parallel can be drawn with the multiple policies that need to be learned for different

Table 4: Average episodic regret (Δ) and improvement over MO with Std. ER baseline ($>$) for both weight change scenarios (lower is better) for DST. We distinguish between overall performance, and performance over the last 25k steps

		Overall				Last 25k steps			
		Standard ER		DER		Standard ER		DER	
	Algorithm	Δ	$>$	Δ	$>$	Δ	$>$	Δ	$>$
Sparse Weight Changes	NAIVE	0.061	+64.86%	0.06	+62.16%	0.064	+137.04%	0.084	+211.11%
	MO	0.037	-0.0%	0.031	-16.22%	0.027	-0.0%	0.022	-18.52%
	MN	0.031	-16.22%	0.03	-18.92%	0.02	-25.93%	0.019	-29.63%
	CN	0.024	-35.14%	**0.021**	**-43.24%**	0.012	-55.56%	**0.009**	**-66.67%**
	CN-UVFA	0.025	-32.43%	0.023	-37.84%	0.015	-44.44%	**0.009**	**-66.67%**
	CN-ACTIVE	0.032	-13.51%	0.028	-24.32%	0.021	-22.22%	0.016	-40.74%
	UVFA	0.034	-8.11%	0.03	-18.92%	0.023	-14.81%	0.017	-37.04%
Regular Weight Changes	NAIVE	0.093	+97.87%	0.095	+102.13%	0.1	+122.22%	0.114	+153.33%
	MO	0.047	-0.0%	0.052	+10.64%	0.045	-0.0%	0.05	+11.11%
	MN	0.113	+140.43%	0.111	+136.17%	0.126	+180.0%	0.104	+131.11%
	CN	0.029	-38.3%	**0.025**	**-46.81%**	0.02	-55.56%	**0.014**	**-68.89%**
	CN-UVFA	0.029	-38.3%	0.028	-40.43%	0.018	-60.0%	0.017	-62.22%
	CN-ACTIVE	0.04	-14.89%	0.042	-10.64%	0.03	-33.33%	0.032	-28.89%
	UVFA	0.057	+21.28%	0.051	+8.51%	0.053	+17.78%	0.046	+2.22%

$\mathbf{w}_t$ in our setting, and the resulting bias. While some of the challenges of both settings are comparable, we found that selective experience replay performs poorly on our dynamic weights problem. We hypothesize that this is due to two major differences in our approach. First, the transition-based selection presents the same problem we observe for Exploration-based Selection (see above). Second, their best working variant of selective experience replay, called *distribution matching* does not promote diverse experiences, instead it attempts to match the distribution of experiences across all tasks. If this distribution is not diverse, rare interesting experiences obtained through random exploration are likely to be overridden by more common experiences.

4.4. Results

These factors contribute to the poor performance of exploration-based selection and selective experience replay (which we label respectively as EXP and SEL in Figure 6 and Tables 5,6,7 and 8). For all algorithms in the sparse weight change scenario, selective experience replay performs worse than DER. However, we found that SEL generally improved performance over standard experience replay. In contrast, EXP has a consistently damaging effect on performance. As for DER, we find that the influence of SEL on the regular weight change scenario is insignificant. EXP however still has a significant negative impact on performance. Regardless of the weight change scenario or experience replay type, the naive algorithm fails to learn any kind of tradeoff and as a result it performs poorly across our experiments.

References

De Bruin, T.; Kober, J.; Tuyls, K.; and Babuška, R. 2018. Experience selection in deep reinforcement learning for control. *The Journal of Machine Learning Research* 19(1):347–402.

Deb, K.; Pratap, A.; Agarwal, S.; and A. M. T. Meyarivan, T. 2002. A fast and elitist multiobjective genetic algorithm: Nsga-ii. 6:182 – 197.

Isele, D., and Cosgun, A. 2018. Selective experience replay for lifelong learning. *CoRR* abs/1802.10269.

Liu, C.; Xu, X.; and Hu, D. 2015. Multiobjective reinforcement learning: A comprehensive overview. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 45(3):385–398.

Mossalam, H.; Assael, Y. M.; Roijers, D. M.; and Whiteson, S. 2016. Multi-objective deep reinforcement learning. *CoRR* abs/1610.02707.

Schaul, T.; Quan, J.; Antonoglou, I.; and Silver, D. 2015. Prioritized experience replay. *CoRR* abs/1511.05952.

Vamplew, P.; Dazeley, R.; Berry, A.; Issabekov, R.; and Dekker, E. 2011. Empirical evaluation methods for multiobjective reinforcement learning algorithms. *Machine Learning* 84(1):51–80.

Van Hasselt, H.; Guez, A.; and Silver, D. 2016. Deep reinforcement learning with double q-learning. In *AAAI*, 2094–2100.

Wang, Z.; de Freitas, N.; and Lanctot, M. 2015. Dueling network architectures for deep reinforcement learning. *CoRR* abs/1511.06581.

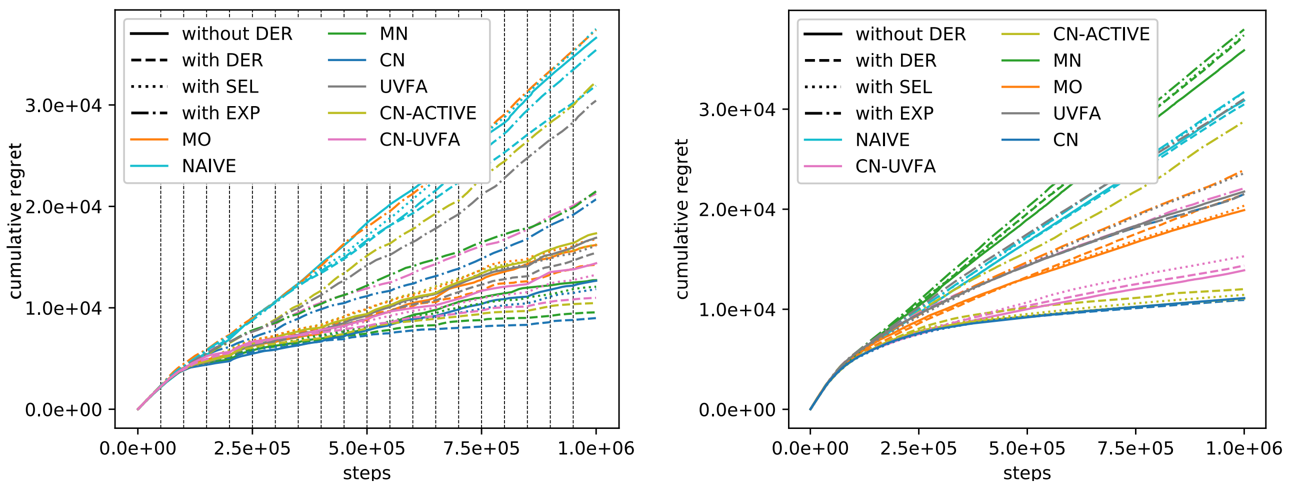


Figure 6: **Left:** Cumulative regret for the Minecart problem when weights change every 50k steps (vertical lines), MN+DER and OnUVFA+DER overlap each other, CN+SEL and MN+SEL overlap each other. **Right:** Cumulative regret for the Minecart problem when weights change over the span of 10 episodes, CN, CN+DER, CN+SEL and CNC overlap to form the lowest curve.

Table 5: Overall average episodic regret (Δ) and improvement over MO with Std. ER baseline ($>$) for the *sparse weight change* scenario (lower is better) for *Minecart*.

	Overall							
	Standard ER		DER		SEL		EXP	
Algorithm	Δ	$>$	Δ	$>$	Δ	$>$	Δ	$>$
NAIVE	0.732	+125.93%	0.638	+96.91%	0.75	+131.48%	0.709	+118.83%
MO	0.324	–	0.285	-12.04%	0.336	+3.7%	0.748	+130.86%
MN	0.255	-21.3%	0.191	-41.05%	0.242	-25.31%	0.43	+32.72%
CN	0.253	-21.91%	**0.18**	**-44.44%**	0.237	-26.85%	0.414	+27.78%
CN-UVFA	0.288	-11.11%	0.22	-32.1%	0.265	-18.21%	0.425	+31.17%
CN-ACTIVE	0.347	+7.1%	0.21	-35.19%	0.325	+0.31%	0.645	+99.07%
UVFA	0.338	+4.32%	0.308	-4.94%	0.322	-0.62%	0.609	+87.96%

Table 6: Average episodic regret (Δ) and improvement over MO with Std. ER baseline ($>$) for the *sparse weight change* scenario (lower is better) for *Minecart* over the last 250k steps.

	Last 250k steps							
	Standard ER		DER		SEL		EXP	
Algorithm	Δ	$>$	Δ	$>$	Δ	$>$	Δ	$>$
NAIVE	0.791	+187.64%	0.651	+136.73%	0.851	+209.45%	0.802	+191.64%
MO	0.275	–	0.207	-24.73%	0.241	-12.36%	0.818	+197.45%
MN	0.139	-49.45%	**0.063**	**-77.09%**	0.133	-51.64%	0.403	+46.55%
CN	0.184	-33.09%	0.068	-75.27%	0.155	-43.64%	0.467	+69.82%
CN-UVFA	0.218	-20.73%	0.102	-62.91%	0.187	-32.0%	0.414	+50.55%
CN-ACTIVE	0.316	+14.91%	0.088	-68.0%	0.202	-26.55%	0.754	+174.18%
UVFA	0.302	+9.82%	0.253	-8.0%	0.213	-22.55%	0.743	+170.18%

Table 7: Overall Average episodic regret (Δ) and improvement over MO with Std. ER baseline ($>$) for the *regular weight change* scenario (lower is better) for Minecart.

	Overall							
	Standard ER		DER		SEL		EXP	
Algorithm	Δ	$>$	Δ	$>$	Δ	$>$	Δ	$>$
NAIVE	0.617	+55.03%	0.61	+53.27%	0.634	+59.3%	0.635	+59.55%
MO	0.398	–	0.43	+8.04%	0.407	+2.26%	0.478	+20.1%
MN	0.718	+80.4%	0.746	+87.44%	0.748	+87.94%	0.76	+90.95%
CN	0.222	-44.22%	**0.219**	**-44.97%**	0.222	-44.22%	0.43	+8.04%
CN-UVFA	0.278	-30.15%	0.287	-27.89%	0.306	-23.12%	0.442	+11.06%
CN-ACTIVE	0.221	-44.47%	0.24	-39.7%	0.229	-42.46%	0.576	+44.72%
UVFA	0.435	+9.3%	0.43	+8.04%	0.472	+18.59%	0.62	+55.78%

Table 8: Average episodic regret (Δ) and improvement over MO with Std. ER baseline ($>$) for the *regular weight change* scenario (lower is better) for Minecart over the last 250k steps.

	Last 250k steps							
	Standard ER		DER		SEL		EXP	
Algorithm	Δ	$>$	Δ	$>$	Δ	$>$	Δ	$>$
NAIVE	0.56	+117.05%	0.551	+113.57%	0.588	+127.91%	0.581	+125.19%
MO	0.258	–	0.319	+23.64%	0.251	-2.71%	0.36	+39.53%
MN	0.67	+159.69%	0.709	+174.81%	0.72	+179.07%	0.712	+175.97%
CN	0.069	-73.26%	**0.064**	**-75.19%**	0.066	-74.42%	0.267	+3.49%
CN-UVFA	0.149	-42.25%	0.149	-42.25%	0.165	-36.05%	0.299	+15.89%
CN-ACTIVE	0.065	-74.81%	0.071	-72.48%	0.069	-73.26%	0.56	+117.05%
UVFA	0.273	+5.81%	0.267	+3.49%	0.346	+34.11%	0.538	+108.53%