Reinforcement Learning for Underwater Communication

Ian Pegg
ipegg@ucsd.edu

Ria Aggarwal
r2aggarw@ucsd.edu

Yuance Li
yul130@ucsd.edu

Abstract

Neural networks suffer from the phenomenon of catastrophic forgetting which hinders continual or lifelong learning and leads to erasure of previously acquired skills. In this paper, we explore whether catastrophic forgetting can be addressed in a dynamic environment within reinforcement learning context. Our work is intended as a proof of concept for underwater communication in which we control the movement of the receiver present underwater to maximize the quality of the received audio signal. In particular, we find that a distributed noisy network can mitigate within task forgetting by learning a more generalizable policy which can quickly adapt to changes in the environment. However, the issue of catastrophic forgetting is still unresolved.

1. Introduction

Underwater communication is a challenging problem given the variability of the medium. Water may be saline or fresh, warm or cool, clear or cloudy, and all of these factors may be slowly changing over time. Because of water's conductive nature, it transmits radio signals poorly, so today's state of the art in underwater communication uses audio signals. These signals are variously attenuated, scattered, Doppeler-shifted, delayed, and refracted by the medium in such a way that long-distance communication at high datarates becomes impossible. [1]

Our work focuses on a specific problem within this domain. We have a transmitter with the ability to aim its signal, and a receiver with the ability to move up and down to get the best signal. Using reinforcement learning techniques, we will train an agent to position a receiver based on the observed environmental conditions and optimize the received signal strength. The nature of reinforcement learning is such that many trials are required before an agent reaches an optimal policy. Thus, the bulk of training must be performed in a simulated environment to be completed in a reasonable amount of time. Due to the similarity with the Atari game *Pong*, a modified version of this game is our starting point for the environment simulator.

Our goal is to deliver a proof of concept agent that can successfully control a *Pong* paddle to defeat an opponent in an environment where the ball may change speed and direction as it traverses the playing area, simulating an audio signal moving through a changing medium.

2. Reinforcement Learning

This section provides a brief introduction to reinforcement learning [2, 3].

2.1. Markov Decision Processes

A Markov decision process is a Markov chain with rewards and controlled transitions defined by a tuple $(\mathcal{X}, \mathcal{U}, p_0, p_f, T, r, q, \gamma)$ or $(\mathcal{S}, \mathcal{A}, p_0, p_f, T, r, q, \gamma)$. The states can be denoted by either x or s and the actions can be denoted by either u or a:

- $\mathcal{S}$ or $\mathcal{X}$: is a discrete/continuous set of states
- $\mathcal{A}$ or $\mathcal{U}$: set of all actions
- p_0 is a prior pmf/pdf defined on $\mathcal{S}$ or $\mathcal{X}$
- $p_f(\cdot|x, u)$ is a conditional pmf/pdf defined on $\mathcal{X}$ for a given $x \in \mathcal{X}$ and $u \in \mathcal{U}$.
- T is a finite/infinite time horizon
- $r(x, u)$ is a function specifying the reward of applying control $u \in \mathcal{U}$ in state $x \in \mathcal{X}$.
- $q(x)$ is a terminal reward of being in state x at time T.
- $\gamma \in [0, 1]$ is a discount factor.

Under this framework, the goal of reinforcement learning is to identify a policy π that maximizes the total reward. A control policy $\pi(x)$ is a function from state x to control $u \in \mathcal{U}$. To this end, we define a *value function*, the cumulative reward of a policy π applied to an MDP with initial state $x \in \mathcal{X}$, at time $t = 0$:

Finite-horizon:

$$V_0^\pi(x) = \mathbb{E}\left[q(x_T) + \sum_{t=0}^{T-1} r(x_t, \pi_t(x_t)) \mid x_0 = x\right]$$

Discounted Infinite-horizon:

$$V^{\pi}(x) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(x_t, \pi(x_t)) \mid x_0 = x\right]$$

As $T \to \infty$, optimal policies become stationary, i.e., $\pi := \pi_0 \equiv \pi_1 \equiv \cdots$, and independent of x_0.

2.2. Discounted Reward

In an infinite horizon problem (such as a game of *Pong* where each game can last forever), each reward will be multiplied by a discount factor. Considering a *Pong* game, such a discount factor has the effect of increasing the value of scoring sooner. Similarly, it reduces the impact of being scored on far in the future. Therefore, even against a perfect opponent, the value function will be maximized by missing the ball as little as possible.

$$v_{\pi}(s_k) = \mathbb{E}\left[\sum_{t=k}^{T} \gamma^{t-k} r_t(s_t, \pi(s_t))\right] \tag{1}$$

In equation 1, as $\gamma \to 1$, future rewards become more important. As $\gamma \to 0$ the agent becomes *myopic* such that only the immediate reward matters.

2.3. Action-Value (Q) Function

In practice, the action-value function in equation 2 is often more convenient than the state-value function v. The input to the action-value function is the state-action pair.

$$q_{\pi}(s_k, a_k) = r(s_k, a_k) + \mathbb{E}\left[\sum_{t=k+1}^{\infty} \gamma^{t-k} r(s_t, \pi(s_t)) \mid s_k\right] \tag{2}$$

2.4. Partially-Observable MDP

In most cases assumption **3** for an MDP will be violated. If **3** is violated, we have a partially-observable MDP (POMDP). In our modified *Pong* game, while the agent will be able to observe the position of the paddles and the trajectory of the ball, the agent will have no knowledge of the medium through which the ball travels. It will only have the behavior of the ball to infer the true state of the environment.

A POMDP is defined by the tuple $(\mathcal{S}, \mathcal{A}, p, r, \Omega, \mathcal{O})$:

- Ω: the set of observations
- $\mathcal{O}$: the set of observation probabilities.

In a POMDP, an agent at state s selects action a based on its belief about the current state, $b(s)$. After taking action a, it observes reward r and observation o. Based on the observation and its belief about the current state, the agent can then update its belief about the new state:

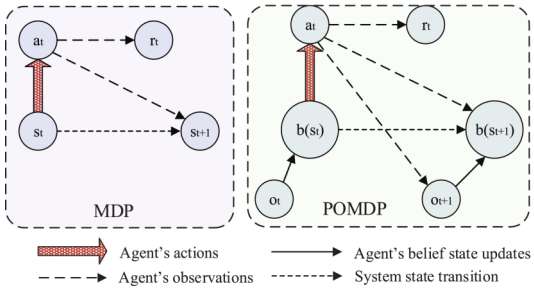

Figure 1: POMDP versus MDP [3]

$$b(s') = \frac{\mathcal{O}(o|s,a,s') \sum_{s \in \mathcal{S}} p(s'|s,a) b(s)}{\sum_{s' \in \mathcal{S}} \mathcal{O}(o|s,a,s') \sum_{s \in \mathcal{S}} p(s'|s,a) b(s)} \tag{3}$$

where $\mathcal{O}(o|s,a,s')$ is the probability of receiving observation o after taking action a in state s leading to state s'. Figure 1 illustrates the differences between a MDP and POMDP.

In the remainder of this paper, we will assume that the *Pong* game is a true MDP. In the future, we will be able to refine our model by considering a full POMDP.

2.5. Bellman Equations

The previous sections provided a brief introduction to MDPs and POMDPs. Here we will introduce the first step to solving these processes. If we consider the classic dynamic programming approach to solving MDPs, we need a recursive form of the value or action-value functions. We can in fact derive both.

$$v_{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s\right] \tag{4}$$

$$= \mathbb{E}_{\pi}\left[r_0 + \gamma \sum_{t=1}^{\infty} \gamma^{t-1} r_t | s_0 = s\right] \tag{5}$$

$$= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left(r + \gamma \mathbb{E}_{\pi}\left[\sum_{t=1}^{\infty} \gamma^{t-1} r_t | s_1 = s'\right]\right) \tag{6}$$

$$= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left(r + \gamma v_{\pi}(s')\right) \tag{7}$$

Equation 7, is Bellman equation for the state-value function, with which we can compute $v_{\pi}(s)$ using only the policy function, the reward space, the state-transition probability function, and the value function of all states s'.

With an analogous derivation, we can arrive at the Bellman equation for the action-value function

$$q_\pi(s,a) = \sum_{s'}\sum_{r} p(s',r|s,a)\left(r+\gamma\sum_{a'}\pi(a'|s')q_\pi(s',a')\right) \quad (8)$$

2.6. Temporal Difference Learning

Equations 7 and 8 hint at an iterative approach to approximating the value function. Given an initial guess for $v_\pi(s')$ or $q_\pi(s',a')$, we can step through the MDP and iteratively update our value function estimates.

This leads us to an our first reinforcement learning technique, temporal difference (TD) learning. TD showed early promise in TD-Gammon [4], a reinforcement learning agent that learned to play backgammon at a master level.

$$\mathcal{V}(S_t) \leftarrow V(S_t) + \alpha\left[R_{t+1} + \gamma\mathcal{V}(S_{t+1} - V(S_t)\right] \quad (9)$$

$$\mathcal{Q}(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma\mathcal{Q}(S_{t+1},A_{t+1}) - \mathcal{Q}(S_t,A_t)] \quad (10)$$

where α is the learning rate. Equations 9 and 10 are the TD update equations for estimating the state value function and action value function respectively. The power of these equations is that we have entirely done away with the environment model p and the policy π, instead using Monte-Carlo methods to extract information from those functions without estimating them directly.

Considering equation 9, we can interpret this as updating the value function with the weighted difference between the observed value $R_{t+1} + \gamma\mathcal{V}(S_{t+1})$ and target value $\mathcal{V}(S_t)$.

In the following sections, we will describe some of the reinforcement learning techniques derived from TD learning.

2.6.1 SARSA

The name "SARSA" comes from the tuple that defines SARSA learning: $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$. Here, we learn the action-value function by following some policy with the exploration method of our choice, updating the function using equation 10. This is called *on-policy* learning.

2.6.2 Q-learning

Q-learning is subtly different from SARSA [5]. Instead of updating the action-value function based on the policy, the update is made using the action that maximizes the action-value function at state S_{t+1}. The update equation is given by 11.

$$\mathcal{Q}(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma\max_a \mathcal{Q}(S_{t+1},a) - \mathcal{Q}(S_t,A_t)] \quad (11)$$

In this way, we directly learn the action-value function given the optimal policy, and our learning is de-coupled from the policy that is actually followed. This is called *off-policy* learning.

3. Related Work

In underwater communication, no work has attempted to resolve the problem we proposed in the introduction using reinforcement learning. However, we do have sources to draw from. Other authors have applied reinforcement learning to underwater communication in other applications, and famously, Google DeepMind has applied reinforcement learning to various Atari games including *Pong*.

Reinforcemnt learning has been applied in an attempt to resolve propagation delay issues in communications with an autonomous underwater vehicle (AUV) [6]. The Dyna-Q learning algorithm (the authors' variant of Q-learning) was used to determine modulation parameters where the effective signal to noise (ESNR) ratio was used as the state and the discounted bit error rate was used as the reward. ϵ-greedy exploration was used, and learning was achieved through a version of experience replay. After some fine-tuning of the replay memory length, adaptive modulation using Dyna-Q was able to out-perform a static fine-tuned modulation scheme.

In a slightly different paradigm, a model-based reinforcement learning was applied to the retrieval of sensor data from an AUV [7]. In this approach, the reward was latency, plus a large penalty for dropped packets, the state was the number of packet re-tries, and ϵ-greedy exploration was used. The action space allowed the transmitter to choose the next node in the network. The adaptive protocol outperformed all baseline protocols.

In the field of wireless communication in general, reinforcement learning has seen wider application. In a review of reinforcement learning in communications [3], the main topics that were addressed were jamming, multi-modal communication, network routing, and adaptive transmission in time-varying channels. These studies use more advanced reinforcement learning techniques like dueling deep Q-learning, LSTM-guided exploration [8], and online learning [9].

Some common features of all of these studies are important to note. In those focusing on underwater communications, the focus has only been on channel parameters, and

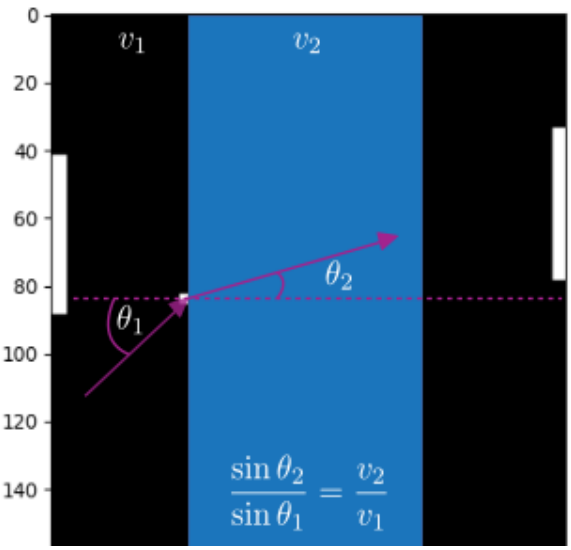

Figure 2: Pong game with refractive layer

deep-learning has not been applied. In the communications papers, the authors derive their algorithms from the work of Google's DeepMind Atari games project, whose latest agent, Agent57 [10], has outperformed humans on all 57 Atari games.

4. Method

The following formally defines the state space, action space, and the reward of our MDP problem. The **state space** $\mathcal{X}$ of the Pong game is a stack of 4 frames at time t. The **action** space $\mathcal{U}$ is discrete and has three actions which are to do nothing (0), move up (1), and move down (2). The total **reward** for successfully hitting the ball past the opponent is +1. Conversely, if the opponents scores a goal the reward is -1. For all the other intermediate states the reward is 0. Each episode culminates when either one of the players score a total of 20 points. Therefore, agent can receive a total reward ranging from -20 and +20 at the end of each episode.

4.1. Modified Pong Game

We created our own *Pong* environment upon which we could add our own modifications using the OpenAI Gym utilities. This environment was then implemented in pure Python using Numpy for rendering with a basic refracting layer in the middle as explained in figure 2.

4.2. Deep Q-Leaning

Given a state, $x \in \mathcal{X}$, the objective is to find a policy $\pi(x)$ that maximises the total expected future rewards given by $Q(x, u)$:

$$Q^*(x,u) = \max_{\pi} Q^{\pi}(x,u) = r(x,u) + \gamma \mathbb{E}_{x' \sim p_f(\cdot|x,u)} \left[\max_{u' \in \mathcal{U}} Q^*(x',u') \right] \quad (12)$$

Since we don't know the motion model $p_f(x'|x,u)$ and the reward function $r(x,u)$ but we have access to samples of the system transitions and incurred rewards, the model free reinforcement learning approach will be used to solve the MDP problem. We can employ Temporal Difference methods to approximate the expected long-term reward by a sample average over a single system transition and an estimate of the expected long-term reward at the new state (bootstrapping). There are two steps which are involved in arriving at the optimal Q value iteratively which are policy evaluation and policy improvement. Any number of policy evaluation steps can be performed before doing the policy improvement step which makes the algorithm highly generalizable. Moreover, convergence to $Q^*(x,u)$ is guaranteed only for $0 \leq \gamma < 1$.

Policy Evaluation: updates the Q value estimate $Q^{\pi}(x_t, u_t)$ towards an estimated long-term reward:

$$Q^{\pi}(x_t,u_t) \leftarrow Q^{\pi}(x_t,u_t) + \alpha(r(x_t,u_t) + \gamma Q^{\pi}(x_{t+1},u_{t+1}) - Q^{\pi}(x_t,u_t)) \quad (13)$$

Policy Improvement: The policy improvement step can be implemented model-free, i.e., can compute $\max_u Q^{\pi}(x,u)$ without knowing the motion model p_f or the state reward:

$$\pi'(x) = \arg\max_{u \in \mathcal{U}} Q^{\pi}(x,u) \quad (14)$$

The fact that Q^{π} is an approximation to the true Q-function and not the actual true Q-function still causes problems because picking the best control according to the current estimate of the Q-function might not be the actual best control. If we commit to a deterministic policy early on than we might not visit all the states in the state space. Hence, estimating Q^{π} will not be possible at those never-visited states and controls. To ensure that we do not commit to the wrong controls too early and continue exploring the state and control spaces we should employ ϵ-greedy policy. A stochastic policy that picks the best control according to Q(x, u) in the policy improvement step but ensures that all other controls are selected with a small (non-zero) probability:

$$\pi(u|x) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{U}|} & \text{if } u = \arg\max_{u' \in \mathcal{U}} Q(x,u') \\ \frac{\epsilon}{|\mathcal{U}|} & \text{otherwise} \end{cases} \quad (15)$$

Q-learning is an off-policy TD optimal control method that does not use importance sampling. The learned Q function eventually approximates Q* regardless of the policy being followed to generate the transitions. The Q-values are updated as follows:

$$Q(x_t, u_t) \leftarrow Q(x_t, u_t) + \alpha \left[r(x_t, u_t) + \gamma \max_{u \in \mathcal{U}(x_{t+1})} Q(x_{t+1}, u_{t+1}) - Q(x_t, u_t) \right] \quad (16)$$

The state space is continuous and large therefore we will experiment with function approximators. The idea is to represent the Q-value function using function approximation with parameters θ.

$$\hat{Q}(x, u; \theta) \approx Q^{\pi}(x, u)$$

Applying tabular methods to continuous space is intractable. Hence, we try to use a function approximator that generalizes from seen to unseen states. A differentiable function approximator is preferable to allow parameter updates and for this purpose we will use a Neural Network as our non linear function approximator.

4.3. Neural Network

The Deep Q-Network (DQN) has two parts, an encoder and a classifier. The encoder is Convolutional Neural network with three convolutional layers. The first convolutional layer has 3 in-channels and 32 out-channels with a stride of 4 and kernel size 8. The second convolutional layer has 64 out-channels with a stride of 2 and kernel size 4. The third convolutional layer has 64 out-channels with a stride of 1 and kernel size 3. The classifier has 2 fully connected layers with 3136 and 512 hidden units respectively. The input is a 400 $\times$ 300 dimensional image and the output is 3 dimensional vector which represents the Q value associated with every action for a given state (image). In the Q-learning algorithm we pick the action associated with highest Q-value by taking a max over the output of the DQN.

4.4. Exploration vs Exploitation

We use an ϵ-greedy policy to strike a balance between exploitation and exploration. This is done because if we commit to a deterministic policy early on we might never explore some of the states and control strategies which may lead to better long term rewards. In order to overcome this problem we start with a initial exploration probability of $\epsilon_0 = 1.0$ and set the exploration probability decay parameter as k = 1000k and the exploration probability value is updated after every step t as follows:

$$\epsilon = \epsilon_0 \, e^{-\frac{t}{k}}$$

As the number of episodes increase our ϵ value approaches 0 which is desired.

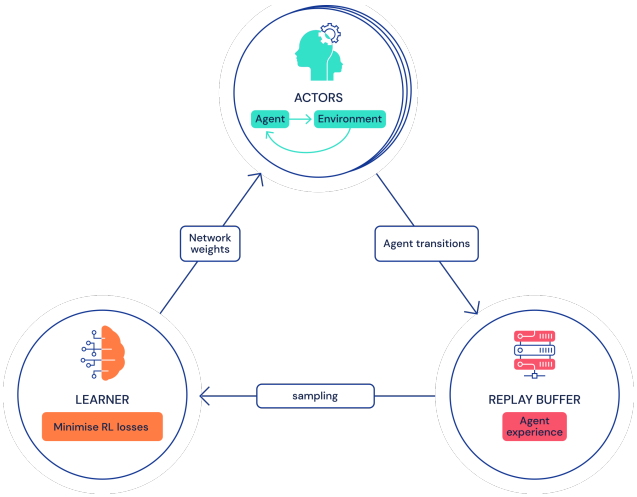

Figure 3: Components of distributed learning [10]

4.5. Experience Replay

A technique called experience replay is adopted to improve learning stability. It simply uses a queue called replay memory to preserve a fixed number of recent historical state transitions in our case it is 100,000 historical transitions, coming from multiple episodes. Every time we randomly sample a number of state transitions from the replay memory, pass them as a batch into the neural network for training. This also improves training efficiency in practice because we are reusing our experience generated form previous policies. Training a model using batch data is much faster than one-by-one training when using a modern deep learning frameworks like PyTorch.

4.6. Distributed Learning

With experience replay, we now have a framework where the training of the network is offline and can be decoupled from from the process of experience generation. In the distributed learning framework, multiple actors, each in their own environment, generate experiences that are pushed to a shared memory pool [11]. Then, just as in DQN with experience replay, this memory is sampled to train the policy network. The actors' policy networks are periodically updated from the learner's policy network. See figure 3.

The benefits of distributed learning are twofold:

1. The learner operates asynchronously so it can process experiences faster, leading to faster convergence. In our case, using ten actors, convergence was about six times faster.

2. The set of experiences is more diverse, leading to improved results.

4.7. Noisy Network

Noisy networks (hereafter referred to as noisy-net) are a state-action space exploration technique. In contrast to dithering methods like ϵ-greedy [12] and intrinsic motivation methods like entropy regularization [13], noisy-net *learns* the degree of exploration and is unbiased by any intrinsic motivation. The parameters of noisy-net [14] are stochastic, and both the mean and variance of each parameter is trained. For exploration, we simply choose the greedy action generated by noisy-net.

Specifically, we replace the fully-connected layers of our network, given by the equation

$$y = wx + b,\ w \in \mathbb{R}^{p\times q},\ x \in \mathbb{R}^q,\ b \in \mathbb{R}^p \tag{17}$$

with a noisy layer:

$$y = \underbrace{(\mu^w + \sigma^w \odot \varepsilon^w)}_{w} x + \underbrace{\mu^b + \sigma^b \odot \varepsilon^b}_{b} \tag{18}$$

where $\odot$ is element-wise multiplication. μ and σ are trained by stochastic gradient descent and ε are normally-distributed random variables.

The downside of this network is that it requires twice as many parameters as DQN and requires the generation of $p + q$ or $pq + q$ random variables, depending on the chosen randomization scheme (see [14]).

5. Results

Here, we will compare the performance of our baseline DQN algorithm with noisy, distributed, and distributed-noisy algorithms. In general, the noisy-net *vastly* outperforms DQN in speed of convergence, and distributed training significantly outperforms sequential training in terms of final rewards.

5.1. Baseline Comparisons

To begin, we will compare the performance of different algorithms in an unchanging environment.

Figure 4 shows the performance of our four networks on the baseline environment, with no refraction. Noisy-net achieves high performance in just a handful of episodes, and converges in about 300. If one focuses on the plot of steps per episode, it can be seen that the noisy agent achieves convergence by first learning to return the ball *without* learning how to score on the opponent (of which high steps per episode is an indicator). Noisy-net also significantly outperforms DQN in terms of the final rewards.

Figure 5 shows the performance of our four networks on an environment with a refractive layer set to half the speed of the default. This is an environment in which our stochastic opponent excels, and the DQN baseline does quite poorly. Here, we can see *significant* improvement for

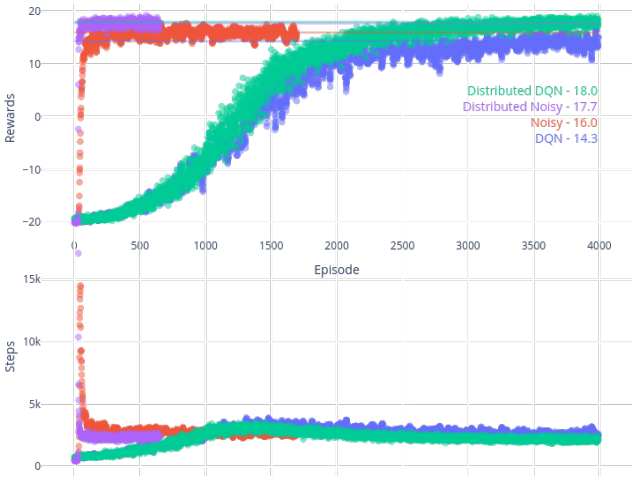

Figure 4: Comparison of algorithms in a refraction-free environment. Rewards per episode (top) and time-steps per episode (bottom) for four different algorithms. Values are smoothed with a 10-episode moving average. Horizontal lines and values beside each network name indicate the final average reward. Noisy-net converges faster, so was trained for fewer episodes than DQN.

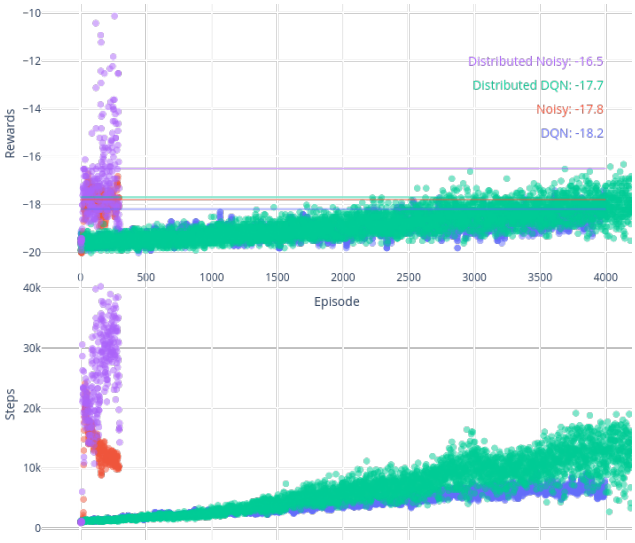

Figure 5: Comparison between distributed, noisy, and baseline DQN models.

noisy-net, especially distributed noisy-net, against the baseline. In this environment, it is only the distributed noisy network that seems to have the potential for a breakout performance.

5.2. Comparisons in a Changing Environment

In an experiment where the speed of the refractive layer changed at each time-step, noisy-net vastly outperformed DQN; see figure 6.

The experiment presented in figure 6 would be difficult

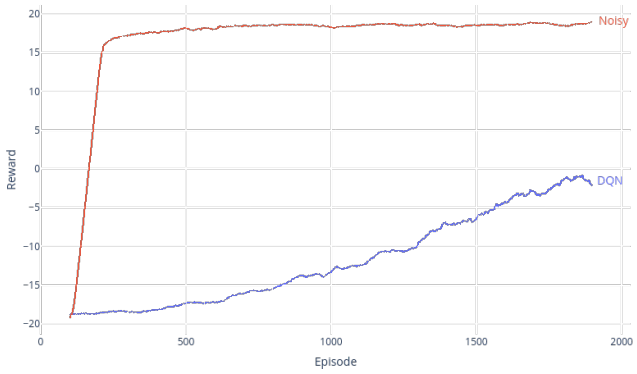

Figure 6: Comparison of DQN and Noisy-net in a changing environment. The ball speed in the refractive layer was changed randomly at each time-step with the change drawn from a normal distribution. The speed of the layer was clipped to between half and double the original speed. Lines are smoothed with a 100-episode moving average.

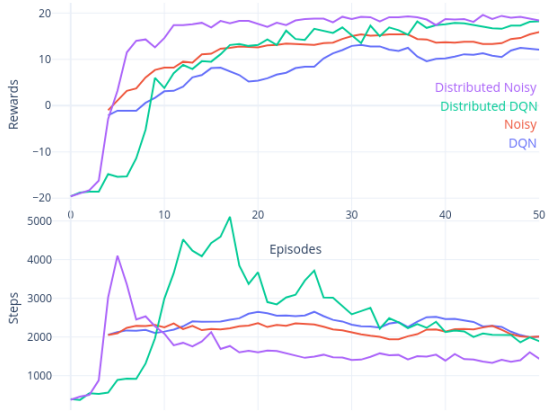

Figure 7: Comparison of algorithms trained on the default environment, then fine-tuned on an environment with the refractive layer speed set to twice the default speed. Lines are smoothed over a 10-episode moving average.

to execute in a distributed environment because the parameters of each of the environments would have to be synchronized to get meaningful results. Therefore, we also investigated how well a pre-trained model can adapt when exposed to a new environment.

Figure 7 shows that when all models are trained on the baseline environment, they initially perform very poorly on the new environment, consistently missing the ball in the first few episodes. When fine-tuning all models converge in about 15 to 30 episodes, with noisy and distributed algorithms converging fastest. This indicates that all of our models are highly adaptable, but noisy-net exploration has significant benefits, even when using a pre-trained model.

6. Conclusion

We have demonstrated two techniques that improve upon our baseline DQN algorithm.

First, noisy-net significantly outperforms DQN in speed of convergence significantly and lead to higher rewards. More interestingly, noisy-net performs significantly better than DQN in changing environments. This may be explained by noisy-net's intelligent stochasticity; noisy-net learns an exploration strategy, whereas the exploration in DQN is purely random.

Second, distributed learning significantly outperforms sequential training in terms of both final rewards and training time. This multi-actor algorithm runs several parallel environments and generates a decorrellated and diverse set of experiences, thus avoiding over-fitting. It also accelerates training at the cost of computational resources.

Last, we demonstrated that both distributed learning and noisy-net can be integrated into a single algorithm which combines the fast convergence of noisy nets with high rewards given by distributed learning. It absorbs the advantages of both methods and achieves superior performance in both changing environment and fine-tuning experiments.

References

[1] M. Chitre et al. "Recent advances in underwater acoustic communications networking". In: *OCEANS 2008*. 2008, pp. 1–10.

[2] Richard Sutton. *Reinforcement learning : an introduction*. Cambridge, Massachusetts London, England: The MIT Press, 2018. ISBN: 978-0262039246.

[3] Nguyen Cong Luong et al. "Applications of deep reinforcement learning in communications and networking: A survey". In: *IEEE Communications Surveys & Tutorials* 21.4 (2019), pp. 3133–3174.

[4] Gerald Tesauro. "TD-Gammon, a self-teaching backgammon program, achieves master-level play". In: *Neural computation* 6.2 (1994), pp. 215–219.

[5] Christopher John Cornish Hellaby Watkins. "Learning from delayed rewards". In: (1989).

[6] Qiang Fu and Aijun Song. "Adaptive modulation for underwater acoustic communications based on reinforcement learning". In: *OCEANS 2018 MTS/IEEE Charleston*. IEEE. 2018, pp. 1–8.

[7] S. Basagni et al. “Finding MARLIN: Exploiting multi-modal communications for reliable and low-latency underwater networking”. In: *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 2017, pp. 1–9.

[8] Oshri Naparstek and Kobi Cohen. “Deep multi-user reinforcement learning for distributed dynamic spectrum access”. In: *IEEE Transactions on Wireless Communications* 18.1 (2018), pp. 310–323.

[9] Paulo Victor Rodrigues Ferreira et al. “Multiobjective reinforcement learning for cognitive satellite communications using deep neural network ensembles”. In: *IEEE Journal on Selected Areas in Communications* 36.5 (2018), pp. 1030–1041.

[10] Adrià Puigdomènech Badia et al. “Agent57: Outperforming the atari human benchmark”. In: *arXiv preprint arXiv:2003.13350* (2020).

[11] Steven Kapturowski et al. “Recurrent experience replay in distributed reinforcement learning”. In: *International conference on learning representations*. 2018.

[12] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*. Vol. 135. MIT press Cambridge, 1998.

[13] Ronald J Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[14] Meire Fortunato et al. *Noisy Networks for Exploration*. 2017. arXiv: `1706.10295 [cs.LG]`.