A Literature Review of Deep RL

Noah Feinberg **Raymond Guo**

Abstract

The early 2010s saw a rise in the use of deep neural networks applied to a wide variety of problem domains. The focus of our review is exploring the effect of deep networks on reinforcement learning problems, specifically learning to play games based on sensory input. The seminal paper in this domain was Mnih et al. [2013] which applied convolutional neural networks (CNNs) to learn Q-functions for a variety of Atari 2600 games, a standard testing environment. Further developments to this technique have refined methods by improving all aspects of algorithms.

1 Background

1.1 Reinforcement learning

Reinforcement learning (RL) is a very general field that is interested in learning how agents can learn to make optimal decisions in a dynamic environment to maximize cumulative reward. RL contrasts with two other main paradigms of machine learning, supervised learning and unsupervised learning; in supervised learning a model learns from labeled input/output pairs to predict outputs of new inputs; unsupervised learning tries to learn information about the structure of unlabeled data. Neither of these paradigms fit well to a domain such as playing Atari games, where the best action is not known so it cannot be labeled ahead of time. Instead RL algorithms must explore their environment to gain information about the best actions and then exploit those actions to maximize reward.

RL is typically modeled as a Markov decision process (MDP) which includes the following components

- A collection of environment states $\mathcal{S}$
- A set of actions the agent can take $\mathcal{A}$
- Transition probabilities $P_a(s, s') = \mathbb{P}(s_{t+1} = s' \,|\, s_t = s \wedge A_t = a)$ from state s to s' $(s, s' \in \mathcal{S})$ given action $a \in \mathcal{A}$
- Immediate rewards $R_a(s, s')$ based on action and transition

The goal of reinforcement learning is to then determine an optimal policy

$$\begin{aligned} \pi : \mathcal{A} \otimes \mathcal{S} &\to [0, 1] \\ s, a &\mapsto \mathbb{P}(A_t \,|\, S_t = s) \end{aligned}$$

that maximizes the expected discounted reward

$$V_\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \,\middle|\, S_0 = s \right]$$

where γ represents how much less we value future rewards.

We could also define a function Q_π as

$$Q_\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \,|\, S_0 = s, A_0 = a \right]$$

Preprint. Under review.

where V represents the value of a state under a policy, Q represents the value of state action pair under that policy and there is a natural connection between them

$$V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) Q_\pi(s,a)$$
$$Q_\pi(s,a) = \sum_{s' \in \mathcal{S}} \mathbb{P}(s' \mid s,a)(R_a(s,s') + \gamma V_\pi(s'))$$

1.2 Q-Learning

Now we consider the optimal Q function in the sense of

$$Q(s,a) = \max_\pi Q_\pi(s,a)$$

Now we could follow a deterministic policy of choosing

$$a = \text{argmax}_{a' \in \mathcal{A}} Q(s,a)$$

Under this policy we have the property that $V(s) = \max_{a, \in \mathcal{A}} Q(s,a)$ and so we can simplify the expression for Q in terms of V to

$$\begin{aligned} Q(s,a) &= \sum_{s' \in \mathcal{S}} \mathbb{P}(s' \mid s,a)(R_a(s,s') + \gamma \max_{a' \in \mathcal{A}} Q(s',a')) \\ &= \mathbb{E}_{s'}[R_a(s,s') + \gamma \max_{a' \in \mathcal{A}} Q(s',a')] \end{aligned}$$

which is called the Bellman equation.
The goal of Q-learning is to learn Q values by initializing them to random values, then performing an update rule based on this Bellman equation to improve our guess. Suppose at time t we make action a and transition from s to s', we can use whatever reward we observed as a single term approximation of this expectation

$$R_a(s,s') + \gamma \max_{a' \in \mathcal{A}} Q(s',a')$$

Then we can update our current value of Q by choosing some learning rate $\alpha \in [0,1]$ and making our new value of Q a weighted average

$$Q(s,a) \leftarrow (1-\alpha) Q(s,a) + \alpha(R_a(s,s') + \gamma \max_{a' \in \mathcal{A}} Q(s',a'))$$

In practice we will not use a deterministic greedy algorithm, but rather an *ϵ-greedy algorithm* which behaves the same as the greedy algorithm with probability $1-\epsilon$, and otherwise picks a random action to explore.

For a finite time Markov Decision Process (MDP), Q-learning will converge to an optimal solution. That being said, Q-learning suffers from significant limitations since it has to store Q values for every state action pair, which is not just costly, but often impossible when state space is continuous. Additionally it needs to learn each individual pair of values separately, and can't take advantage of any kind of similarity between states to learn faster.

1.3 Deep Learning

Deep learning (DL) refers to a broad spectrum of algorithms which utilize large networks with many layers to approximate arbitrary functions. The form of DL we are interested in is primarily deep convolutional neural networks (Deep CNNs) which stack many convolutional layers on top of each other to pull features out from input and learn a desired function.

2 Deep Q-learning

2.1 SGD for Q-Learning

The first big breakthrough towards Deep Q-learning was made in Mnih et al. [2013] which made two significant improvements to RL algorithms which saw significant improvements in performance. The

first change was the use of aforementioned Deep Q-Networks, multi layer CNNs applied to the pixels of sensory input to approximate the Q function. Learning a Q function in this way means we don't need to store our values in a table and can take up much less space, additionally continuity of of the function allows it to take advantage of closeness of different inputs; instead of needing to learn Q values separately for each sequence individually we can generalize the learning to our whole input space.

This is accomplished by defining a class of function $Q(s, a : \theta)$ parameterized by θ and updating θ over time using the Bellman equation to converge to the true function $Q^*(s, a)$. Now the model trains by minimizing over a sequence of loss functions $L_i(\theta_i)$ that changes after each iteration of the algorithm. The loss we are minimizing over at each step is the following MSE

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s, a : \theta_i))^2\right]$$

with y_i defined as

$$\mathbb{E}_{s'\in\mathcal{S}}[R_a(s, s') + \gamma \max_{a'\in\mathcal{A}} Q(s', a' : \theta_{i-1}) \mid s, a]$$

that is we use what we observed actually happened combined with the previous networks predictions of Q to set as the label, which will then clearly change as the network learns. The distribution $\rho(s, a)$ is a probability distribution over how the model chooses actions, typically this is an ϵ-greedy algorithm.Such a network is termed a Deep Q-Network (DQN) and we can use standard SGD algorithms to approximate gradients of loss functions to minimize this function at each step.

2.2 Double Deep Q-Learning

While DQN is very effective on its own, it and Q-learning in general have a tendency to overestimate the value of states because of the max operation in updating Q values. This can be problematic since we won't usually overestimate values uniformly, and the error increases as the action space grows, which will lead to learning suboptimal policies. A solution to this problem already existed for standard tabular Q-learning, double Q-learning in which we learn two Q tables, using one to pick the optimal policy, and a second to evaluate the Q function.

This was then applied to DQN by van Hasselt et al. [2015] by learning two parameters θ, θ' so we know estimate the true Q^* by the formula

$$Q^*(s, a) = r_t + \gamma Q(s_{t+1}, \text{argmax}_{a'\in\mathcal{A}}(Q(s_{t+1}, a' : \theta)), \theta')$$

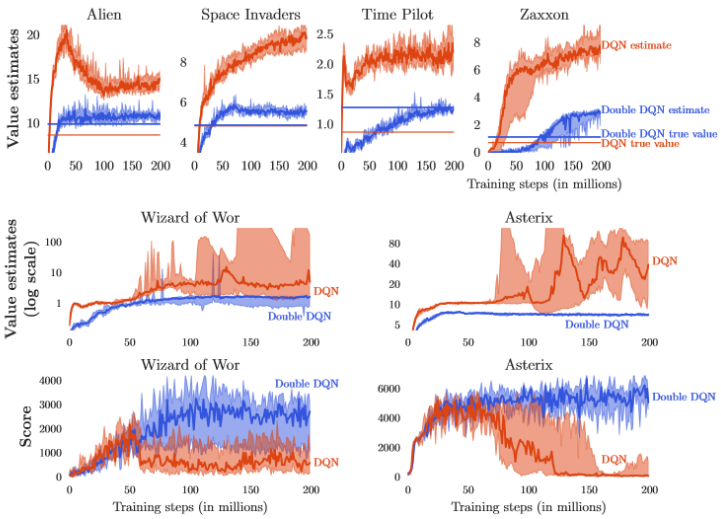
then do SGD like before computing the difference between this and the value

$$Q(s_t, a_t, \theta)$$

for multiple time steps to get some update. This however, only updates the parameter θ, to update θ' we periodically set it equal to a weighted average between it and θ

$$\theta' \leftarrow \lambda\theta + (1 - \lambda)\theta'$$

for a hyperparameter $0 < \lambda \ll 1$. Decoupling the evaluation and selection does in fact greatly decrease overestimation and improve performance



Taken from van Hasselt et al. [2015]

2.3 Value Advantage Decomposition

Using the value and Q functions already defined, we could define an advantage function as

$$A_\pi(s,a) = V_\pi(s,a) - Q_\pi(s,a)$$

then because of how Q_π is defined we see that

$$E_\pi[A_\pi(s,a)] = 0$$

So we can think of V representing a baseline of how good a state s is, and then A represents how good or bad every action is in that state s is, which together we can use to find Q.

The insight of Wang et al. [2015] is that we can compute V, A individually with separate DNN, then recombine them at the end to gain an estimate of Q; a major advantage of this is that the algorithm they propose is architecture agnostic, that is the DNNs used to compute both parts can be changed, only keeping the part of the network that recombines them. This allows for improvements to search and model architecture to take advantage of this structure for even better performance. Empirical results have shown that this separation allows the model to train faster and generalize better.

To describe this we will now train two models parameterized by 3 values,

$$Q(s,a:\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha)$$

so both networks share some parameter θ before splitting apart. One issue with this strategy is that the equation $Q = V + A$ is under-determined, there is some "true" Q value but there are infinitely many solutions to this equation of the form $Q = (V + c) + (A - c)$ that a network could potentially learn.

In order to solve this Wang et al. [2015] added a module to the end of the network that recombines the values as

$$Q(s,a;\theta,\alpha,\beta) = V(s,a:\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s,a';\theta,\alpha) \right)$$

under this formulation we force the modified A values to be mean 0 which is similar to what we had when we derived the formula for the theoretical version. Importantly this step is part of the network so gradient computations are done here and it requires no modification of specific optimization algorithms.

The authors note that this algorithm sees large improvements over the single network architecture when there are many actions with similar rewards present.

2.4 Replay Memory

In standard Q-learning, every time we take a new action we make updates. One significant problem of this is succeeding states will be highly correlated with the one you were just in, and the updates will be in similar directions. This correlation can mean the steps we take wont be representative of the true gradient, so we may not converge well.

The other significant contribution of Mnih et al. [2013] was applying *replay memory* to Q-learning. Replay memory is a strategy to both decrease this correlation, and allow for reuse of previously seen states. As our agent plays, we record experience tuples

$$e_t = (s_t, a_t, r_t, s_{t+1})$$

which we store in a data-set $\mathcal{D}$. Now we can perform batch SGD on $\mathcal{D}$ to get a collection of uncorrelated experiences.

In their paper, they would limit the size of the memory to a fixed site N and use a FIFO to replace new memories as the agent continues to play and learn, then they would sample memories from this uniformly for their SGD. This raises two related questions, how can we more effectively choose which memories to keep, i.e. going from a queue to a priority queue, and how can we sample memories that are more useful. Both these questions are related to some notion of "importance" for samples so it motivates trying to quantify how useful a given sample is.

The first paper addressing this limitation was Schaul et al. [2016] which used temporal-difference (TD) error

$$\delta_t = \left| Q(s_t, a_t) - (r_t + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a')) \right|$$

to weight a probability distribution over the stored memories. The two proposed distributions they gave were

$$p(t) \propto (\delta_t + \epsilon)^\alpha$$

for some hyperparameter $\epsilon, \alpha \geq 0$, and a rank based one

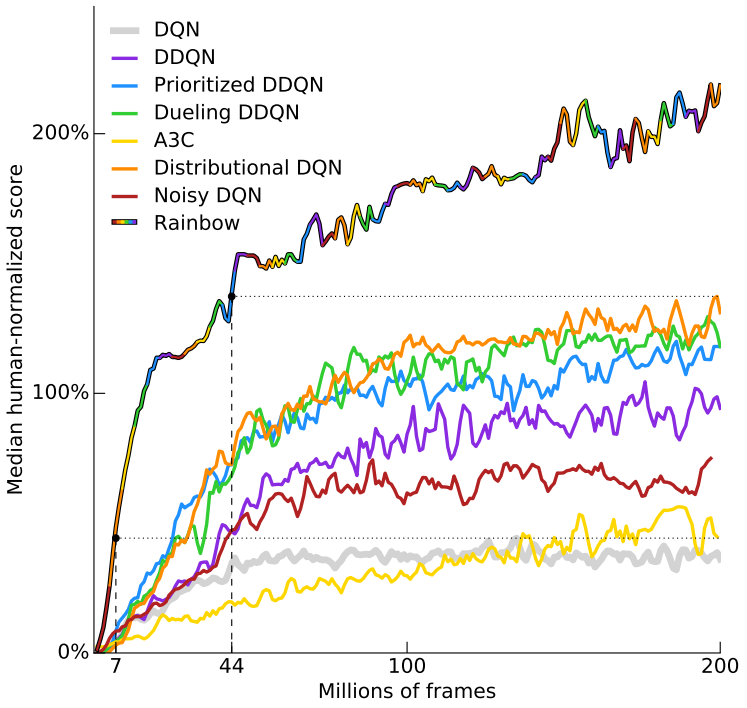
$$p(t) \propto \text{rank}(\delta_t)^{-\alpha}$$

For both of these distributions we note that $\alpha = 0$ reduces to the uniform distribution, and as α gets larger we more aggressively prioritize the largest errors. Prioritized sampling was later adapted to distributional RL (maybe add a section about this) by Hessel et al. [2017] which instead adopted a loss proportional to KL divergence of our target distribution and predicted one.

An important caveat is that all practical implementations of these priority based methods only update the probabilities when we sample that point in our SGD. The Q-learning algorithm will already compute δ_t in the standard step so there is no cost to updating it when we sample the point, but updating all the probabilities after every step would be prohibitively expensive.

2.5 Combining Improvements

After Mnih et al. [2013] was published, many papers were published after which improved the performance of their algorithm by improving different elements of it, some of which we have discussed. Some work had been done combining multiple elements together, but Hessel et al. [2017] went further and combined work from 6 different papers to achieve state of the art performance on Atari 2600 games



Taken fromHessel et al. [2017]

This suggests that the various improvements made over this period were addressing largely independent issues with the standard DQN algorithm and showed surprising modularity between many of these different approaches.

3 Exploration

In order to avoid being stuck in local minima, Reinforcement Learning algorithms must have some incentive to explore, rather than always greedily choosing the optimal action they've learned so far. The naive way to implement this, as discussed near the bottom of section 1.2, is to employ ϵ-greedy algorithms, which usually make the greedy choice, but instead choose a random action with small probability ϵ.

An alternative choice for prioritizing exploration is simply to count how many times each state has been visited, and add an incentive to explore states that have been seen less. As seen in Strehl and Littman [2008], this can be enforced by adding a term proportional to $N(x,a)^{-\frac{1}{2}}$ to the value of each state-action pair (x,a), where $N(x,a)$ is simply the amount of times (x,a) has already been visited.

Bellemare et al. [2016] took note of the fact that many practical implementations of RL used variations on ϵ-greedy methods in spite of the theory surrounding count-based methods. This is because count-based methods are impractical in many settings (such as playing Atari games, which serve as common benchmarks for RL algorithms), where the number of state-action pairs is so enormous that the vast majority of states are never visited, and it's nearly impossible for the same state to be visited twice. In such cases, in any experiments with reasonable length, count-based methods provide almost no information, and thus no incentive to explore. Bellemare et al. [2016] sought to address this issue by abstracting the idea of a count.

3.1 Pseudo-Counts

The paper defined a learned function that mimics desirable properties of a count. It defined a density model as a function that assigns sequences of states $x_{1:n}$ to probability distributions on states. The probability of a state x under a sequence of states $x_{1:n}$ is then denoted $\rho(x;x_{1:n})$. We intuitively want to think of these functions as determining how probable it is that we see a state x given that we've already seen the sequence of states $x_{1:n}$. We also define $\rho'(x;x_{1:n}) := \rho(x;x_{1:n}x)$, named

the *recoding probability* of x, where $x_{1:n}x$ refers to appending x to the end of the sequence. Since intuitively, when we see x more, we should expect it to occur with higher probability, we seek density models that are *learning positive*, which is the requirement that

$$\rho'(x; x_{1:n}) \geq \rho(x; x_{1:n})$$

for all x and all sequences $x_{1:n}$.

One such natural density model is

$$\rho(x; x_{1:n}) = \frac{N(x, x_{1:n})}{n}$$

where $N(x, x_{1:n})$ is the number of times x appears in the sequence $x_{1:n}$, named the *empirical count*. This is the function we try to abstract with pseudocounts, noting that as discussed above, the empirical count itself is useless in many practical implementations because it is almost always 0 or 1. When ρ is defined as above,

$$\rho(x; x_{1:n}) = \frac{N(x, x_{1:n})}{n}, \rho(x'; x_{1:n}) = \frac{N(x, x_{1:n}) + 1}{n + 1}$$

which are obvious by the definitions. From these equations, we may solve for N and n to yield

$$N(x; x_{1:n}) = \frac{\rho(x; x_{1:n})(1 - \rho'(x; x_{1:n}))}{\rho'(x; x_{1:n}) - \rho(x : x_{1:n})} = n\rho(x; x_{1:n})$$

Bellemare et al. [2016] uses this equality as motivation to construct a pseudo-count using any learning-positive ρ. The paper defines the associated pseudo-count function $\hat{N}$ and pseudo-count total function $\hat{n}$ for a density model ρ by

$$\hat{N}(x; x_{1:n}) = \frac{\rho(x; x_{1:n})(1 - \rho'(x; x_{1:n}))}{\rho'(x; x_{1:n}) - \rho(x : x_{1:n})} = \hat{n}(x; x_{1:n})\rho(x; x_{1:n})$$

Specifically, the paper sought to use the Context-Tree Switching (CTS) density model, which decomposes the probability of seeing a frame as the product of the probabilities of seeing individual patches in the frame, each of which is in turn computed with the namesake Context-Tree Switch algorithm. They examined how the associated pseudo-count $\hat{N}$ changed through simulation, and made the following empirical observations which support the hypothesis that this pseudo-count $\hat{N}$ behaves like the empirical count in desirable ways:

1. Pseudo-counts of events that have not been seen remain near zero
2. The pseudo-count remains reasonably similar in magnitude with the empirical count
3. States that appear more frequently have higher pseudo-counts
4. On average, pseudo-counts grow approximately proportionally to the empirical counts
5. Pseudo-counts adapt quickly when the policy, and thus the underlying distribution of how often states appear, undergo changes.

The practicality of the pseudo-count was tested against Atari benchmarks by adding a bonus proportional to $(\hat{N}(x) + 1)^{-\frac{1}{2}}$ to the value of each state in order to encourage exploration of novel states. This algorithm achieved significant success against standard DQNs on what is thought to be one of the hardest Atari games, Montezuma's Revenge. This game is thought to be difficult because it offers a sparse reward function, so standard DQNs often never find rewards and don't understand how to explore.

In contrast, by consistently being incentivized to explore new states, the algorithm using the pseudo-count bonus achieves much better results, consistently exploring more of the game's rooms and achieving an unprecedentedly high score.

Ostrovski et al. [2017] furthered these results by showing the impacts of a better density model in producing the pseudo-counts. The CTS density model was a fairly simple model with the advantages of faster performance and easier implementation. After swapping it out for the more sophisticated PixelCNN, the agent improved significantly, as seen in the following graph:

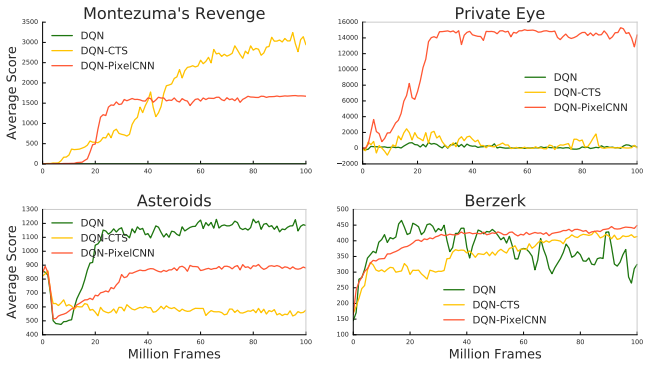


Figure 5. DQN, DQN-CTS and DQN-PixelCNN on hard exploration games (**top**) and easier ones (**bottom**).

Taken from Ostrovski et al. [2017]

This examines the performance of a standard DQN, a DQN augmented with pseudo-counts from CTS, and a DQN augmented with pseudo-counts from PixelCNN on four Atari games. In Montezuma's Revenge and Private Eye, two games that RL models historically have difficulties exploring, both the PixelCNN and CTS pseudo-count algorithms vastly outperform the standard DQN. The CTS algorithm performs better than PixelCNN on Montezuma's Revenge, but PixelCNN dominates both other models in Private Eye.

PixelCNN is also seen to perform better than CTS and the standard DQN in Asteroids and Bezerk, two games where the rewards are much less sparse, so the standard DQN outperforms both pseudo-count methods. Overall, Ostrovski et al. [2017] found that PixelCNN outperformed CTS in 52 of the 57 games when comparing maximum scores, and 51 out of the 57 games when comparing the area under the curves. This provides strong empirical evidence that the effectiveness of pseudo-counts is heavily dependent on the choice of density model.

At the same time, Ostrovski et al. [2017] found the following surprising results when adjusting the relative weights of the environment's reward and the exploration bonus:

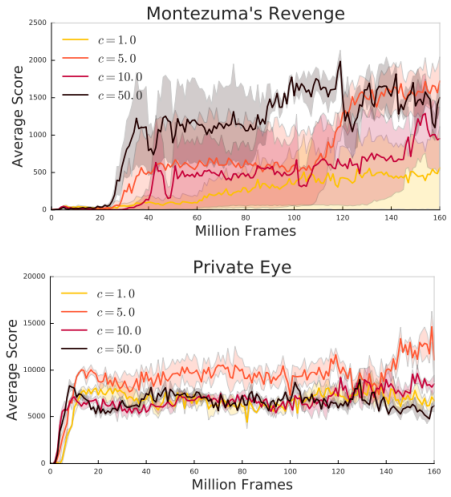


Figure 13. DQN-PixelCNN trained from **intrinsic reward only** (3 seeds for each configuration).

Taken from Ostrovski et al. [2017]

Here, c is a parameter that governs the magnitude of the bonus provided by the pseudocounts. Intrinsic reward refers to this bonus, so these are graphs of agents that are *only receive the exploration bonus as reward*. In spite of this, these agents continuously improve their score (which is not a factor at all in their reward function). This seems to suggest that in many scenarios when rewards are sparse, exploration is significantly more important than even the explicit rewards from the environment.

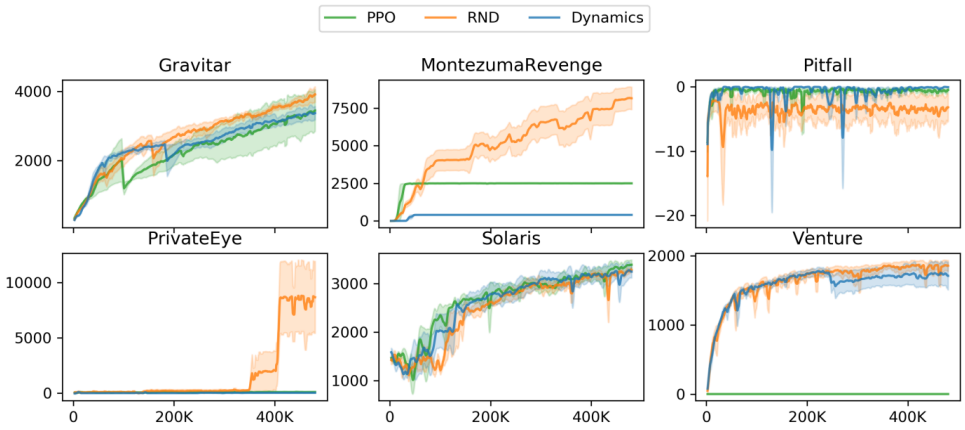
3.2 Random Network Distillation

Burda et al. [2018] also adds an intrinsic reward to the environment's explicit reward in order to encourage exploration, but does so in an entirely different fashion. The method is based off the fact that models are more accurate when provided with inputs similar to those they have already seen. With this in mind, one might attempt to design an algorithm that computes an exploration parameter based off the error of an agent's past predictions of reward. If the agent is poor at predicting the reward of a particular state, we could conclude that the agent hasn't seen similar states, and should strive to explore them.

The problem with such an approach is that these errors in prediction could be caused by either unfamiliarity with data or genuine probabilistic variance in the reward for a particular state. This is called the "noisy TV" problem, where agents trained to explore in this fashion will continuously watch a TV displaying static because it is impossible to predict.

To counteract this problem, Burda et al. [2018] adds two neural networks. The first is a *target* neural network f which never gets trained and has randomly initialized weights fixed for the entirety of the training process. Its purpose is to serve as a deterministic function of the state space. The other is a *predictor* neural network $\hat{f}$ which tries to approximate f, minimizing the MSE loss $\|\hat{f}(x) - f(x)\|^2$ through gradient descent. When $\hat{f}$'s error is higher, we consider the state to be more novel and give the agent more intrinsic reward for exploring it. This solves the "noisy TV" issue, as instead of predicting random reward, we measure the error of a model on a deterministic function: the fixed target network.

As seen in the graph below, Random Network Distillation (RND) performs at a similar level to and on some games significantly improves upon previously state of the art models. It also carries the benefit of being highly efficient, only requiring the forward passes of the target and predictor networks to generate the intrinsic reward.



Taken from Burda et al. [2018]

References

M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos. Unifying count-based exploration and intrinsic motivation. *CoRR*, abs/1606.01868, 2016. URL http://arxiv.org/abs/1606.01868.

Y. Burda, H. Edwards, A. J. Storkey, and O. Klimov. Exploration by random network distillation. *CoRR*, abs/1810.12894, 2018. URL http://arxiv.org/abs/1810.12894.

M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017. URL http://arxiv.org/abs/1710.02298.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

G. Ostrovski, M. G. Bellemare, A. van den Oord, and R. Munos. Count-based exploration with neural density models. *CoRR*, abs/1703.01310, 2017. URL http://arxiv.org/abs/1703.01310.

T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay, 2016.

A. L. Strehl and M. L. Littman. An analysis of model-based interval estimation for markov decision processes. *Journal of Computer and System Sciences*, 74(8):1309–1331, 2008. ISSN 0022-0000. doi: https://doi.org/10.1016/j.jcss.2007.08.009. URL https://www.sciencedirect.com/science/article/pii/S0022000008000767. Learning Theory 2005.

H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL http://arxiv.org/abs/1509.06461.

Z. Wang, N. de Freitas, and M. Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015. URL http://arxiv.org/abs/1511.06581.