A Survey of Reinforcement Learning with Reservoir Computing

Quentin Clark, BU ECE, qtcc@bu.edu

Synopsis: This paper surveys existing uses of Reservoir Computing (RC) for Reinforcement Learning (RL), establishing theoretical and empirical benefits of using RC systems for RL function approximation over traditional deep neural networks. Also included is a vanilla implementation of an RC-based DQN model trained on the OpenAI control tasks with open-sourced code available.

Introduction: Function approximation techniques to reduce staggeringly large state and action spaces in RL has well-established itself as the only way to achieve state of the art RL-based performance on a variety of tasks within a non-differentiable MDP environment[1]. These results have come from using neural networks as function approximators of the value function, Q function, or as a parameterized policy for policy-gradient algorithms. These are most frequently trained with *backpropagation,* a technique for finding the gradient of a neural network's parameters with respect to some differentiable objective function $L(y, y')$ combined with some form of *stochastic gradient descent,* a set of first-order optimization algorithms that attempt to minimize the loss function.

Such approaches have empirically worked well for a variety of tasks, including for increasingly complex neural network architecture like recurrent neural networks (RNNs) and Transformers. However, RNNs in particular suffer from many training difficulties including a tendency to suffer from exploding gradients[2] and computation constraints associated with backpropagation through time (BTT)[3]. These are not intractable problems but require careful solutions or training regime considerations to resolve them.

An alternate neural network training paradigm, called Reservoir Computing (RC), does not use gradient information at all but instead treats the hidden layer neurons of a neural network as *non-linear, high-dimensionality representations of the input layer*. Then, a linear readout layer is trained on the values of these neurons, using computationally cheap linear methods like Support Vector Machines or least-squares linear regression[4]. This approach is conceptually similar to Kernel methods in classical machine learning, but with a randomly initialized form of bootstrapping the data into a linearly learnable dimension instead of using a fixed kernel.

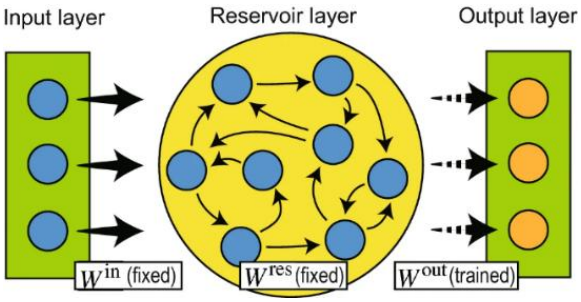

Figure 1: Illustration of the Reservoir Computing setup.[5]

This paradigm can achieve very strong results on tasks with strong recurrent properties, at a significantly cheaper training cost (for example, learning to approximate the behavior of differential equations like the Mackey-Glass equations), and comes with some theoretical guarantees[6].

Using reservoir computing over traditional deep neural networks with gradient descent has some benefits that could be particularly useful for RL:

1: Cheap Recurrent Learning: utilizing recurrence in reinforcement learning is of particular interest

[1] Tesauro 95, Mnih et al 13, Silver et al 16, among others.
[2] Pascanu et al. 13
[3] Gruslys et al 16.
[4] Schrauwen et al. 7.
[5] Sakemi et al 20.
[6] Gonon 21.

because it can allow the neural network to capture more *complete state information* off of an *incomplete observation*. This can help a RL agent learn to incorporate past observations as part of the state without needing to explicitly expand the state space. For example, the now famous Atari DQN paper formulated the state space as the current frame and previous five frames in order to more robustly approximate the true MDP, as a single frame screenshot fails to account for movement of game objects[7]. However, this has many downsides: it massively increases memory requirements because storing previous frames quickly becomes expensive, especially if one is using an experience replay system, and it clearly captures some sort of redundant or non-useful information given that the current observation should have more useful information than previous observations. RNNs naturally capture this idea by having "fading memory" of the network's history, but without needing to store previous frames.

2: Hardware Neural Network Compatibility: Implementing neural networks directly in hardware instead of emulating their behavior in software has the promise of allowing for radically more space and energy efficient neural networks. Traditional backpropagation techniques are incompatible with training hardware neural networks directly because one does not have access to the gradients for the network, and making a backpropagation layer in hardware would be difficult and very sensitive to perturbations. Reservoir computing allows for training because one only needs access to each neurons' values and then a linear training step, which can be done in hardware (such as via an FPGA, although other solutions exist) or in software on a separate computer. This is possible because the linear learning step is simple and capable of being manually programmed into hardware in a way backpropagation cannot be. There are several possible hardware implementations: photonics, liquid, mechanical, and biological, among others[8].

3: Sample Efficiency: a main drawback of RL compared to classical control methods and traditional supervised ML is that effective deep RL requires a truly staggering amount of sample to work properly[9]. RC cannot solve problems with exploration or sample distribution, but there are strong empirical results showing that RC learns from limited samples far faster than SOTA neural networks for sequence tasks. Although reasonable-sized RCs have a lower capacity than modern deep neural nets, they near-universally have far greater sample efficiency[10].

Algorithms:

Fundamentally, any existing deep RL algorithm that uses a traditional neural network as a value function approximator (Q-Learning, Value Iteration, TD Learning, SARSA, etc.) can be translated to an RC context easily. Instead of performing a gradient descent step, one updates the linear output layer of the RC with an online learning rule. Also included are modifications to add an *input bias* to the linear output weights, which is empirically found to substantially increase performance.

Algorithm 1: DQN with Reservoir Computing
Initialize output w_1 randomly with input bias b_0
Let $Q(s_i, a_i, : w_i)$ be our Q-Value approximator, constructed of a fixed-weight RC $\psi(s_i) \to x_i \in R^n$ and learned layer $w_i \in R^{(|A|, n+1)}$ s.t. $Q(s_i : w_i) = w_i * \psi(s_i)$, $Q(s_i, a_i, : w_i) = Q(s_i : w_i)[a_i]$
for episode $= 1, M$ **do**
  initialize environment at $s_1 \in S$
  for $t = 1, T$ do
    With $P[\epsilon]$, select random action $a_t \in A$
    Otherwise, $a_t = \max_a Q(s_t, a; w_n)$
    Take environment step, receive r_t
    Set $\hat{y_t} = r_t + \gamma \max_a Q(s_{t+1}, a; w_t)$
    Set $y_t = Q(s_t, a_t : w_t)$
    $w_{t+1}[a_t] := w_t[a_t] + \alpha[\hat{y_t} - y_t]\psi(s_t)$
  end for
end for
Red indicates optional input bias[11].

[7] Mnih et al 13.
[8] Tanaka et al 19.
[9] Irpan 18.
[10] Cisneros 22.
[11] As suggested in Kanno and Uchida 22.

If wanted, one could add other bells and whistles like a separate estimation/update network (Double DQN), dueling networks, experience replay, etc.

One could also replace the update step with some other algorithm. Broadly, the assumption made using the basic Least Mean Squares (LMS) update step is that are trying to minimize the MSE $L_{MSE}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$ between your predictions and sample data, which leads cleanly to the gradient $\nabla L_{MSE}(\hat{y}, y) = (\hat{y} - y)$. We can see this running update replicates Stochastic Gradient Descent. But you could borrow techniques from online learning/stochastic optimization theory like per-sample learning rate adaptation, momentum, storing past changes and incorporating those into the algorithm somehow, etc. (you obviously cannot use second-order optimization algorithms). Of course, that would require making an informed decision about your algorithm[12] but I would hazard to guess that both in the deep RL and RC RL case there is room to improve on simply using SGD[13].

Policy Gradients using RCs are much more difficult because limiting the parametrization of a policy $\pi_\theta(s)$ needed to calculate $\nabla J_\pi(\theta)$ to the linear gradient of the output layer would intuitively substantially neuter performance. In practice, existing literature either uses PPO like bounds for separate actor-critic networks[14] or use approximations of the network's gradient and do classical backpropagation through time[15].

Existing Literature:

There is already a relatively robust literature surrounding using reservoir computing reinforcement learning, although smaller than the literature for other modern deep learning approaches or other forms of deep reinforcement learning.

The performance achievable with RCRL is currently much more limited than with deep RL generally, similar to RC's underperformance against DL in the supervised learning context but much less aggressively. RCRL has been successfully applied to the now standard gamut of RL tasks: control problems, robotics problems, and Atari 2600 games. I will not go over every paper but will briefly discuss three that I believe are informative of the potential power of RCRL when compared to traditional deep RL.

This work[16] is perhaps the most comprehensive of basic tasks possible with RCRL and goes in depth on the hyperparameters one must tune to make RCRL work. It uses Liquid State Machines (LSMs, a kind of RC with spiking neurons) on OpenAI's Cartpole task, Pacman, and some Atari games (Boxing, Freeway, Gopher, Krull). To ensure desired stable spiking response behavior, they fed their LSM random inputs and then tuned connectivity between neurons accordingly by analyzing the eigenvalue spectrum of the recurrent connectivity matrix. Experiments are run both giving the LSM complete and incomplete state information (to see if the LSM can, through memory, capture velocity properties instead of simply being given them). It fails to solve Cartpole (receiving 200 reward or higher, indicating the model can orient the pole upright and keep it there) but achieves a reasonable result. The partial observation network performed more poorly than the full state observation network, but stronger than non-recurrent networks given the same partial observation. Performance on Pac-Man is strong, and reasonable on Atari: stronger

[12] Orbana 10.
[13] Broadly, it seems to me that using SGD in deep RL is not particularly theoretically motivated seeing as you are not updating your network to a ground truth value for a sample, but an online prediction of your Q-Function. The excellent Fan et al. 20. provides some theoretical guarantees for the convergence of a particular variation of DQN, but with so many assumptions and modifications required to fix the aforementioned problem it scarcely resembles in-use algorithms. This is undoubtedly a topic one could spend years on.
[14] Tieck et al. 18
[15] Mohammadi et al. 22.
[16] Ponghiran et al. 19

than human performance for two of the games, but weaker than DQN at all[17].

So far, I have discussed RC systems with reservoirs resembling simple multi-layer perceptrons with some recurrence. One might ask why it would not be possible to use more modern architecture. Such a task is possible, as shown by this paper[18] which applies convolutional neural networks (CNN) to reservoir computing in a Q-Learning context for the CarRacing and DoomTakeCover Gym tasks. They did not generate the convolutional weights totally randomly but used canonical correlation analysis on a random sampling of game images for each task. This intuitively helps resolve some of the inherent anxiety that the enterprise of RC is fundamentally misguided because the Aristotelian final cause of Deep Learning is in some sense to extract more and more representative features of the input data as network depth increases. Randomly initializing weights fails to do that, but by still utilizing empirical relationships in the data to generate features in a pre-processing step you still capture some of the intuitive power of deep learning. This RCRC (as they call it) performs quite well at both tasks, surpassing DQN performance and A3C performance on both, and being competitive with then SOTA models.

However, all of these papers (and many more not included) fail to take advantage of the second potential benefit of RCRL outlined above: hardware compatibility. These results all used RC in software on Von Neuman machines, with the associated energy and performance losses compared to a specialized hardware implementation. This paper[19] begins to rectify that by implementing a photonic RC RL system to solve the OpenAI Gym CartPole and MountainCar control tasks. This paper claims to be the first to do RC RL in hardware[20], and to great success, solving both tasks very efficiently both in computer simulated RC and hardware RC. The RC was implemented in hardware, but the weights were learned in software, with the RC connecting to the environment in software and communicating the reservoir weights to the computer via hardware modules. One thing that I am surprised it did not do was attempt to transfer weights from the RC trained in simulation to the physical RC. Similar to the ongoing work on effective strategies for transferring RL agents learned in simulation to physical agents, it seems like taking advantage of the ability to train RL models using high-performance computing resources and then transferring the learned output layer weights to physical models for end-product or application use has potential, but there is also the potential for the transfer learning approach to run into issues regarding the simulation not matching the physical system closely enough for effective transfer.

This work also revealed many of the difficulties with training a physical reservoir online. Careful attention was taken to the hardware modules used and parameters (like polling rates) had to be decided deliberately.

RC_RL: An Implementation of RC DQN:

For pedagogical purposes, I implemented DQN with RC in Python to benchmark on the OpenAI tasks[21]. I could only find one other implementation of RC based RL on the internet, so I hope this can serve as a useful baseline for future exploration in this area.

I build off of the DQN module framework here[22] which uses PyTorch for the neural network implementation and Gym for the environment. For the RC components I used ReservoirPy[23], which

[17] Note: this paper used memory information on the Atari system (or rather, the system in emulation) as the state information, and not screenshots like Minh. et al.
[18] Chang and Futagami 20.
[19] Kanno and Uchida et al 22.
[20] I could find one paper that claimed to do it earlier in 2018, with a mirror device that even implemented weight updates in hardware, but it was actually learning the Mackey-Glass time series, so I consider this a supervised learning task and not RL. See Bueno 18.
[21] https://github.com/qtcc-bu/RC_RL.
[22] https://github.com/ritakurban/Practical-Data-Science/
[23] https://github.com/reservoirpy/reservoirpy. I also tried an implementation in PyRCN https://github.com/TUD-

provides easy to use out of box implementations of Echo State Networks, which can be easily adapted into an Extreme Learning Machine by removing all recurrence. My solution includes support for both ESN and ELMs and can optionally use experience replay.

By far the largest takeaway from my experiments was the observation that hyperparameter tuning is very difficult with online RC training. I have non-negligible experience with supervised ML, some experience with RL, and have experienced the various difficulties with finding a model architecture and learning hyperparameters for both (tuning the learning rate, testing optimizers, tuning model depths, choosing exploration strategies and annealing rates for RL, etc.) but RCRL was a unique challenge. For instance, I found that the learning rate α would only converge for learning rates within an order of magnitude around 5e-4.

Ultimately, only using naïve manual hyperparameter tuning on an ELM I was able to achieve performance superior to DQN on Cartpole for a similar wall clock training time, but unable to converge for the other Control tasks. The ELM was able to "solve" Cartpole (usually defined as consistent >200 reward), which the Ponghiran et al. paper was notably unable to do. I was not able to achieve acceptable performance using an ESN.

Two things stood out to me: one, that the RC appeared to realize some of the empirical results about sample efficiency found elsewhere, with it training faster than the 64-neuron large hidden layer NN and at a similar rate to the larger 200-neuron NN. However, the ELM experienced a clear case of catastrophic forgetting, aggressively losing all performance around the episode 800 mark.

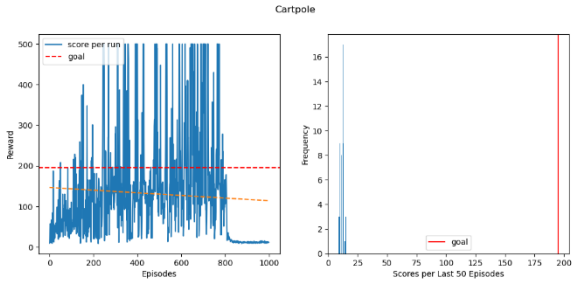

Figure 2: RC RL Cartpole, 1000 Reservoir Neurons

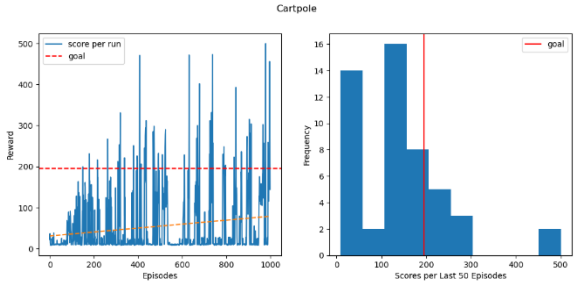

Figure 3: DQN Cartpole, 64 Hidden Neurons

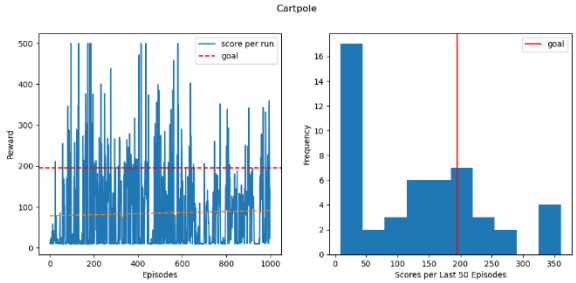

Figure 4: DQN Cartpole, 200 Hidden Neurons

I only have a weak heuristic understanding of why the ELM was susceptible to forgetting and not the DQN, but I suspect it is because the online LMS update will naturally replace all the previously learned weight information. While this is also theoretically true for the DQN's weights, I suspect something about the behavior of those gradients makes very small Q-value updates practically vanish the gradients such that the network forgets less information. This still doesn't explain why the ELM weights, given that they should grow closer to the MDP's true Q-values over time,

STKS/PyRCN which appears to be more feature complete, but found difficulties getting the implementation to perform well.

suddenly update in a maligned way deep into training, but it is not exactly unexpected behavior given how unstable RL training can be.

Conclusion:

This paper introduced Reservoir Computing (RC) and discussed its application to Reinforcement Learning (RL), providing some reasons why RC might have uses as a function approximator in RL algorithms that traditional backpropagation trained deep neural networks do not have. Recent advancements in the field were presented and discussed, and a vanilla implementation of Q-Learning using RC for the OpenAI Control tasks was included, and lessons learned presented.

References:

Bueno, Julian, et al. "Reinforcement learning in a large-scale photonic recurrent neural network." Optica 5.6 (2018): 756-760.

Chang, Hanten, and Katsuya Futagami. "Reinforcement learning with convolutional reservoir computing." Applied Intelligence 50 (2020): 2400-2410.

Cisneros, Hugo, Tomas Mikolov, and Josef Sivic. "Benchmarking Learning Efficiency in Deep Reservoir Computing." Conference on Lifelong Learning Agents. PMLR, 2022.

Fan, Jianqing, et al. "A theoretical analysis of deep Q-learning." Learning for Dynamics and Control. PMLR, 2020.

Gonon, Lukas, and Juan-Pablo Ortega. "Fading memory echo state networks are universal." Neural Networks 138 (2021): 10-13.

Gruslys, Audrunas, et al. "Memory-efficient backpropagation through time." Advances in neural information processing systems 29 (2016).

Irpan, Alex. "Deep Reinforcement Learning Doesn't Work Yet." 2018, www.alexirpan.com/2018/02/14/rl-hard.html.

Kanno, Kazutaka, and Atsushi Uchida. "Photonic reinforcement learning based on optoelectronic reservoir computing." Scientific Reports 12.1 (2022): 3720.

Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).

Mohammadi, Nima, Lingjia Liu, and Yang Yi. "Policy-based fully spiking reservoir computing for Multi-Agent distributed dynamic spectrum access." ICC 2022-IEEE International Conference on Communications. IEEE, 2022.

Orbana, Francesco. "Neural Networks (Maybe) Evolved to Make Adam The Best Optimizer." Parameter-free Learning and Optimization Algorithms, 2010, parameterfree.com/2020/12/06/neural-network-maybe-evolved-to-make-adam-the-best-optimizer.

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks." International conference on machine learning. Pmlr, 2013.

Ponghiran, Wachirawit, Gopalakrishnan Srinivasan, and Kaushik Roy. "Reinforcement learning with low-complexity liquid state machines." Frontiers in Neuroscience 13 (2019): 883.

Sakemi, Yusuke, et al. "Model-size reduction for reservoir computing by concatenating internal states through time." Scientific reports 10.1 (2020): 21794.

Schrauwen, Benjamin, David Verstraeten, and Jan Van Campenhout. "An overview of reservoir computing: theory, applications and implementations." Proceedings of the 15th european symposium on artificial neural networks. p. 471-482 2007. 2007.

Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." nature 529.7587 (2016): 484-489.

Tanaka, Gouhei, et al. "Recent advances in physical reservoir computing: A review." Neural Networks 115 (2019): 100-123.

Tesauro, Gerald. "Temporal difference learning and TD-Gammon." Communications of the ACM 38.3 (1995): 58-68.

Tieck, Juan Camilo Vasquez, et al. "Learning continuous muscle control for a multi-joint arm by extending proximal policy optimization with a liquid state machine." Artificial Neural Networks and Machine Learning–ICANN 2018: 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4-7, 2018, Proceedings, Part I 27. Springer International Publishing, 2018.