Eligibility Traces and Recurrent Neural Network

Pierre Thodoroff
Reasoning and Learning lab
School of Computer Science
McGill University
pierre.thodoroff@mail.mcgill.ca

1 Introduction

Inspired from the dynamical system research we are investigating treating the value function as a system that evolve through time. The previous values should have a strong impact on to the next one unless the behavior of the system is chaotic. In this project we will study the different implications of interpreting the value function as a dynamical system. We will use recurrent neural network as a basis of our analogy. We define a linear recurrent neural network with one hidden unit our value function. From a reinforcement learning perspective this is equivalent to define an augmented state space by adding the previous value function as part of the state. We will now define notations that will be used throughout the paper.

1.1 Notation

- $S_n = \phi_n$ is the state at time step n. This can be the state at time step n or the feature of the state if we use function approximation
- $V(S_n) = V(S_n, \theta_x) = \theta_x \phi_n = \theta_x S_n$ is the traditional value function used in reinforcement learning parametrized by a vector θ_x
- $X_n = \begin{bmatrix} V(X_{n-1}) \\ S_n \end{bmatrix}$ is the augmented state at time step n.
- $V(X_n) = V(X_n, \theta) = X_n \theta = V(S_n) + \mu * V(X_{n-1})$ where $\theta = \begin{bmatrix} \mu \\ \theta_x \end{bmatrix}$

2 Analysis of the momentum method

The first interesting point is that $V(X)$ can be fully expressed using $V(S)$. In all the derivation of this section we assume that the weights do not change during the trajectory. We can gain intuition by making this assumption and simplify the calculations. Most of the intuitions carry on if the weights are changed during the trajectory.

$$V(X_n) = \mu * V(X_{n-1}) + V(S_n) \tag{1}$$
$$= \mu^2 * V(X_{n-2}) + \mu V(S_{n-1}) + V(S_n) \tag{2}$$
$$= \sum_{i=0}^{n} \mu^{n-i} * V(S_i) \tag{3}$$

We can express $V(X_n)$ using a weighted sum of $V(S_i)$ parametrized by μ. If μ is large then the decay will be small and we will average through many previous values. However if μ is small the decay will be large and we will only look at a couple of previous time steps. Intuitively this makes sense if the previous value function is very indicative of the value of my next state (high μ) then it possible to consider a lot of previous step. However if it is not then we should shorten our horizon.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

2.1 n-step return

We now analyze the n-step return induced by this augmented state space:

$$G_t^n = \sum_{i=0}^{n-1} \gamma^i r_{t+i+1} + \gamma^n V(X_{t+n}) \tag{4}$$

$$= \sum_{i=0}^{n-2} \gamma^i r_{t+i+1} + \gamma^{n-1} r_{t+n} + \gamma^n V(X_{t+n}) + \gamma^{n-1} V(X_{t+n-1}) - \gamma^{n-1} V(X_{t+n-1}) \tag{5}$$

$$= \sum_{i=0}^{n-2} \gamma^i r_{t+i+1} + \gamma^{n-1}(r_{t+n} + \gamma V(X_{t+n}) - V(X_{t+n-1})) + \gamma^{t+n-1} V(X_{t+n-1}) \tag{6}$$

$$= \sum_{i=0}^{n-2} \gamma^i r_{t+i+1} + \gamma^{n-1}\delta(X_{t+n}) + \gamma^{n-1} V(X_{t+n-1}) \tag{7}$$

$$= \sum_{i=1}^{n} \gamma^{i-1}\delta(X_{t+i}) + V(X_t) \tag{8}$$

Where we define $\delta(X_i) = r_i + \gamma V(X_i) - V(X_{i-1})$ We expressed the n-step return in function of the usual TD error. We now look to replace V(X) by V(S) in δ

$$\delta(X_n) = r_n + \gamma V(X_n) - V(X_{n-1}) \tag{9}$$

$$= r_n + \gamma V(S_n) + \gamma\mu V(X_{n-1})) - V(X_{n-1}) \tag{10}$$

$$= r_n + \gamma V(S_n) - (1 - \mu\gamma) V(X_{n-1}) \tag{11}$$

$$= r_n + \gamma V(S_n) - (1 - \mu\gamma) \sum_{i=0}^{n-1} \mu^{n-i} * V(S_i) \tag{12}$$

We obtain this formula where instead of subtracting $V(S_{n-1})$ we subtract a weighted average of all the previous $V(S_i)$. The term $(1 - \mu\gamma)$ act as a normalization term such that the weighting sums to one. If μ is high $(1 - \mu\gamma)$ will be small but the summation will decay more slowly. Effectively this will average over more $V(S_i)$ and potentially reduce variance. However if μ is low $(1 - \mu\gamma)$ will be large but the decay will be fast meaning that only a few steps before will be considered. Intuitively if the value function is very unstable then μ should be small and we should not be averaging over a lot of previous value but rather keep a short term horizon. We will now investigate the bias of the proposed method.

$$\delta(X_n) = r_n + \gamma V(X_n - V(X_{n-1}) \tag{13}$$

$$= r_n + \gamma V(S_n) + \gamma\mu V(X_{n-1})) - (\mu V(X_{n-2}) + V(S_{n-1})) \tag{14}$$

$$= r_n + \gamma V(S_n) - V(S_{n-1}) + \mu(\gamma V(X_{n-1}) - V(X_{n-2})) \tag{15}$$

Let's define $\delta(S_i) = r_i + \gamma V(S_i) - V(S_{i-1})$

$$G_t^n = \sum_{i=1}^{n} \gamma^{i-1}\delta(X_{t+i}) + V(X_t) \tag{16}$$

$$= \sum_{i=1}^{n} \gamma^{i-1}(r_{t+i} + \gamma V(S_{t+i}) - V(S_{t+i-1}) + \mu(\gamma V(X_{t+i-1}) - V(X_{t+i-2}))) + V(X_t) \tag{17}$$

$$= \sum_{i=1}^{n} \gamma^{i-1}(\delta(S_{t+i}) + \mu(\gamma V(X_{t+i-1}) - V(X_{t+i-2}))) + V(X_t) \tag{18}$$

$$= \sum_{i=1}^{n} \gamma^{i-1}\delta(S_{t+i}) + \mu \sum_{i=1}^{n} \gamma^{i-1}(\gamma V(X_{t+i-1}) - V(X_{t+i-2})) + V(X_t) \tag{19}$$

$$= \sum_{i=1}^{n} \gamma^{i-1}\delta(S_{t+i}) + V(X_t) + \mu(\gamma^n V(X_{t+n-1}) - V(X_{t-1})) \tag{20}$$

We observe that the method is biased by the term $\mu(\gamma^n V(X_{t+n-1}) - V(X_{t-1}))$. It however has a nice interpretation. If the value function is accurate and "well behaved" through time then $\gamma^n V(X_{t+n-1}) - V(X_{t-1})$ should be close to 0 and we can have a high μ to reduce variance by averaging through more $V(S_i)$. However if $\gamma^n V(X_{t+n-1}) - V(X_{t-1})$ is large then we need to reduce μ in order to reduce the bias and average over less $V(S_i)$. In other words because our value function is not very reliable we need to shorten our horizon.
It is important to note that if we learn μ is it possible to get rid of the bias and obtain an unbiased method that should converge.

2.2 Discussion

The momentum method ends up being a variance trade-off. Our hope is that by introducing bias we can reduce the variance of reinforcement algorithm and speeds up learning. By treating μ as a weight it is also possible to do gradient learning of this parameter. It will however require special treatment as it values should not change too fast and be contained between zero and one. Another way of looking at this would be to consider adding the value function as using a function approximation. This allow to use all the algorithm in reinforcement learning (SARSA, Q-learning ect...). An attempt to prove convergence for TD(λ) with this function approximation was made but one of the assumptions required in [1] is not satisfied. Indeed the proof requires the matrix ϕ of the function approximator to be of full rank. More theoretical analysis can be undertaken by seeing this momentum as a type of function approximation.

3 Experimentation

In this section we evaluate the impact of adding the previous value to the state. Theoretically it is possible to learn μ using the gradient information but as a first step we decided to keep it fixed. All the algorithm developed in this paper are in the function approximation framework as our state becomes non-tabular as soon as we add the previous value function to the state.

3.1 Policy Evaluation

The first algorithm evaluated is semi-gradient TD(0) for policy evaluation. We describe the algorithm below.

Algorithm 1 Semi-gradient TD(0) Policy evaluation with momentum μ

1: Input: The policy π to be evaluated
2: Initialize $\theta = \begin{bmatrix} \mu \\ \theta_x \end{bmatrix}$ randomly
3: **for all** episodes **do**
4: Initialize S
5: $X \leftarrow \begin{bmatrix} 0 \\ S \end{bmatrix}$
6: **for all** step of the episode **do**
7: Choose $A \sim \pi(-|X)$
8: Take action A, observe R,S'
9: $X' \leftarrow \begin{bmatrix} V(X,\theta) \\ S' \end{bmatrix}$
10: $\theta \leftarrow \theta + \alpha(R + \gamma V(X',\theta) - V(X,\theta))\nabla V(X,\theta)$
11: $X \leftarrow \begin{bmatrix} V(X,\theta) \\ S' \end{bmatrix}$
12: **end for**
13: **end for**

An interesting implementation detail to note is that we add the value function calculated with the new parameter to the state after the update to the parameters(line 11). We could add the one calculated originally but it is less accurate. This can be seen as an intermediate methods between teacher forcing and just taking the original one calculated. Intuitively it should improve performance because the new value function will be more accurate. This has been proven empirically (Insert experiment in appendix here). A similar trick is used in TD(0) with V(S). For those experimentation we fixed μ and do not change its value through training. The experiment is done on a random walk with 19 states. We attempt to estimate the value function of the random policy.

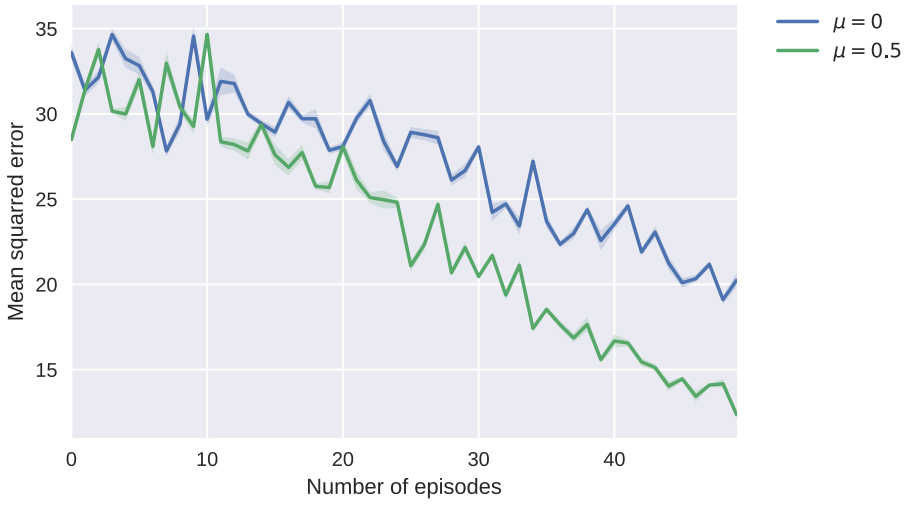

Figure 1: TD(0) with different μ

In this experiment we calculate the mean squared error using trajectories. It is not possible to just query the value of a state with momentum as you need the previous value. It is interesting to see that the method performs slightly better. The important point is that it doesn't not perform worse. The problem is very straightforward so not much improvement can be gain from averaging over the previous value function. More significant improvements should be expected in more complex environment.

3.2 Control

The second algorithm evaluated is SARSA. In this setting again we keep μ fixed.

Algorithm 2 Semi-gradient SARSA with momentum μ

Initialize $\theta = \begin{bmatrix} \mu \\ \theta_x \end{bmatrix}$ randomly
for all episodes **do**
  Initialize S
  $X \leftarrow \begin{bmatrix} 0 \\ S \end{bmatrix}$
  $A \leftarrow get - action(X)$
  for all step of the episode **do**
    Take action A, observe R,S'
    if S' is terminal **then**
      $\theta \leftarrow \theta + \alpha(R - Q(X, A, \theta))\nabla Q(X, A, \theta)$
    end if
    $X' \leftarrow \begin{bmatrix} Q(X, A) \\ S' \end{bmatrix}$
    $A' \leftarrow get - action(X')$
    $\theta \leftarrow \theta + \alpha(R + \gamma Q(X', A', \theta) - Q(X, A, \theta))\nabla Q(X, A, \theta)$
    $X \leftarrow \begin{bmatrix} Q(X, A) \\ S' \end{bmatrix}$
  end for
end for

We evaluate its performance on Cartpole with a simple linear function approximation(binning).

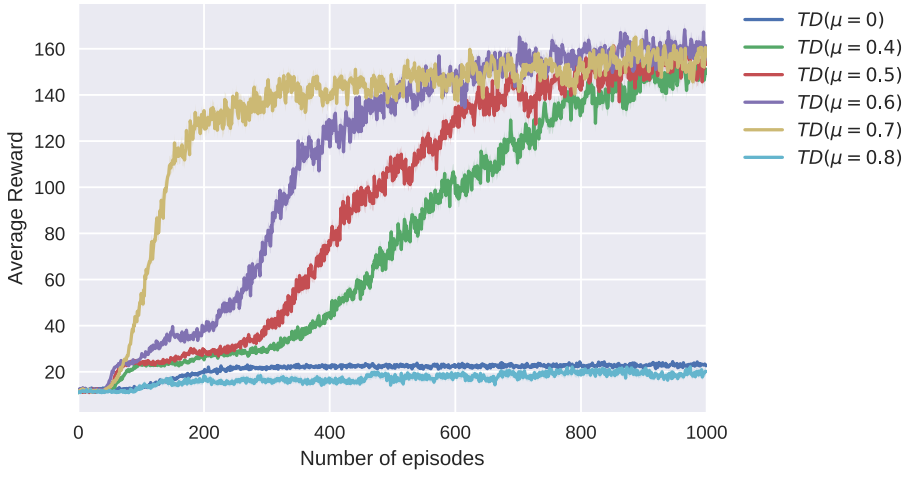

Figure 2: Average reward on Cartpole when changing μ

By varying the momentum we get significant increase in performance. The optimal μ will end up being a bias/variance environment dependent trade off. If μ is too high then the bias is too high and the algorithm won't work. However the higher μ is the more we reduce variance and increase the speed of learning.

Those result have been obtained using a hyper-parameter search on μ and λ. One of the interesting aspect of this graph is combining the use of μ and λ yields the best performance. Using eligibility

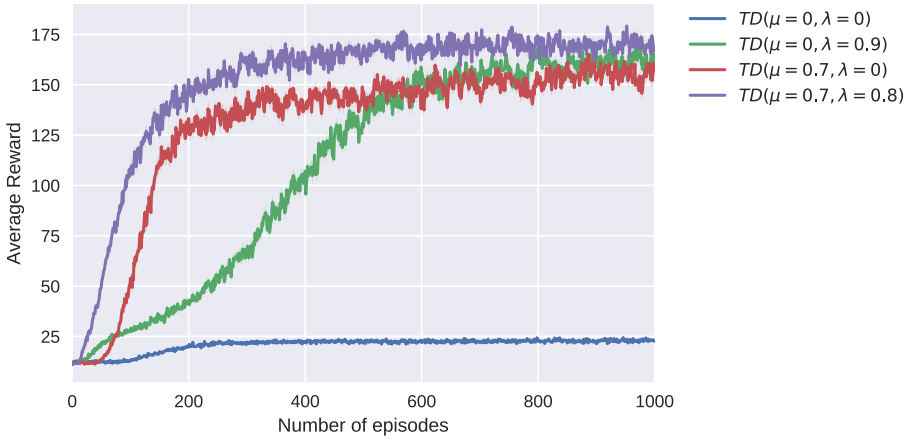

Figure 3: Comparison between momentum and eligibility traces

traces λ increases the performance of the algorithm at the beginning. This shows experimentally that "momentum" and eligibility traces exploits different thing and can be combined to yield optimal performance.

3.3 Gradient Learning

TODO

4 Connection with Eligibility traces

4.1 TD

We first analyze the use of stochastic gradient descent to learn a good value function f. If we were to use stochastic gradient descent on the value function to adjust its weight we would get the following:

$$\begin{aligned} \theta_{t+1} &= \theta_t - \frac{1}{2}\alpha\nabla[V(S_t) - \overline{V}(S_t)]^2 \\ \theta_{t+1} &= \theta_t + \alpha[\overline{V}(S_t) - V(S_t)]\nabla V(S_t) \end{aligned} \tag{21}$$

Where $\overline{V}(S_t)$ is the true value function and $V(S_t)$ is our approximation. However because we do not have the real value of V we replace it with an approximation $U(S_{t+1}) = R_t + \gamma V(S_{t+1})$ yielding the following update :

$$\begin{aligned} \theta_{t+1} &= \theta_t + \alpha[U(S_t) - V(S_t)]\nabla V(S_t) \\ \theta_{t+1} &= \theta_t + \alpha\delta_t\nabla V(S_t) \end{aligned} \tag{22}$$

This inspected further in the chapter 9 of [3].

TD(λ) is one of the most widely used algorithm in reinforcement learning. It bridges the gap between monte-carlo and TD(0). Eligibility traces provides a very efficient way of incorporating n-step return in our algorithm. We define the updates of TD(λ) as follows:

$$\begin{aligned} e_0 &= 0 \\ e_t &= \nabla V_t(X_t) + \gamma\lambda e_{t-1} \\ \theta_{t+1} &= \theta_t + \alpha\delta_t e_t \\ \delta_t &= R_t + \gamma V(X_{t+1}) - V(X_t) \end{aligned} \tag{23}$$

If the updates are offline one can establish the equivalency between the backward view (eligibility traces) and the forward view(lambda-return). However if the updates are done during the trajectory then the eligibility trace do not account for that. A solution has been proposed in true online TD [3]. Eligibility traces can be seen as a vector representing the credits to be assigned to each weights. If λ is small then the decay is large and we will not assign credit for states that happened a long time ago.

4.2 Recurrent neural network

In this section we introduce recurrent neural network and 2 different methods to train them. . Let's consider the problem of training the parameters θ of a dynamical system over a variable h subjected to the evolution equation:

$$h(t+1) = f(h(t), x(t), \theta) \tag{24}$$

h(t) correspond to the value of the hidden state at time t, x(t) the value of the input at time t and f can be a linear or non-linear function. In the deep learning community f is often a Relu or some kind of non-linear function. The goal is to minimize a loss function

$$\sum_t l_t(h(t), y(t)) \tag{25}$$

An important difference to note is that in the more recent RNN literature, another function takes the hidden state and create an output. In the older RNN literature especially the one talking about Real Time Recurrent Learning(the algorithm of interest) a common setup is to have label of the hidden state. The value of your hidden state is your prediction for that time step t.
Using the chain rule one can modify all the parameters of the network according to the gradient $\frac{\partial l_t}{\partial h}$ by propagating the error through all the time steps (backpropagation through time). This is an exact gradient method however it requires to wait until the end of the sequence to update the parameters.

However the continual strategy knows as real-time recurrent learning(RTRL) does something different, it maintains the full gradient of the current state with respect to the parameters:

$$G(t) = \frac{\partial h(t)}{\partial \theta} \tag{26}$$

This matrix G satisfy the following evolution equation :

$$G(t+1) = \frac{\partial f(h(t), x(t), \theta)}{\partial h} G(t) + \frac{\partial f(h(t), x(t), \theta)}{\partial \theta} \tag{27}$$

The interesting part of is that using G(t) it is possible to update the parameters without back propagating through the sequence:

$$\theta = \theta - \alpha \frac{\partial l_t}{\partial \theta} \tag{28}$$

where

$$\frac{\partial l_t}{\partial \theta} = \frac{\partial l_t}{\partial h} G(t) \tag{29}$$

This yields the following update :

$$\theta = \theta - \alpha \frac{\partial f(h(t), x(t), \theta)}{\partial h} G(t) \tag{30}$$

The problem however is that G(t) is a very large matrix of size $[h \times \theta]$. For a reasonably sized network it is impossible to store/evaluate it. That is why people study approximation of this matrix [3]. It is important to note that this is an approximate gradient method. Indeed it does not account the impact of changing the weight through the trajectory.

4.3 Eligibility traces and real time recurrent learning

We define a linear recurrent neural network with one hidden unit (the value function). The error function is $[V(S_t) - \overline{V}(S_t)]^2$ but because we do not have access to the real value function we use $U(S_t)$. The gradient update for this system is :

$$\begin{aligned} \theta_{t+1} &= \theta_t - \alpha \frac{1}{2} \frac{\partial l_t}{\partial \theta} = \theta_t - \alpha \frac{1}{2} \frac{\partial l_t}{\partial h(t)} G(t) \\ \theta_{t+1} &= \theta_t + \alpha [U(S_t) - V(S_t)] \nabla V(S_t) \end{aligned} \tag{31}$$

Using equations (21) we get the following update :

$$\begin{aligned} \theta_{t+1} &= \theta_t + \alpha [U(S_t) - V(S_t)] \nabla V(S_t) \\ \theta_{t+1} &= \theta_t + \alpha \delta_t \nabla V(S_t) \end{aligned} \tag{32}$$

where

$$\begin{aligned} \frac{\partial l_t}{\partial h(t)} &= -\delta_t \\ \frac{\partial h(t)}{\partial \theta} &= G(t) = \nabla V(S_t) \end{aligned} \tag{33}$$

Depending on how many steps we backpropagate the information we get different reinforcement learning algorithm. If we just propagate the information at 1 time step we get TD(0). If we backpropagate all the way up to the beginning we recover TD(1). If we keep a running estimate of $G(t) = \frac{\partial h(t)}{\partial \theta}$ using real time recurrent learning we recover exactly eligibility traces ! Indeed the "gradient" update rule for real time recurrent learning is:

$$G(t+1) = \mu G(t) + \begin{bmatrix} \phi_x \\ V(X_{t-1}) \end{bmatrix} \tag{34}$$

It is identical to the one used for eligibility traces :

$$e(t+1) = \lambda e(t) + \begin{bmatrix} \phi_x \\ V(X_{t-1}) \end{bmatrix} \tag{35}$$

where $\mu = \lambda\gamma$.

5 Discussion

5.1 Eligibility traces <=> real time recurrent learning

Following the connection established in the previous section we can now view eligibility traces as a real time gradient used to update the parameter at each time step. Eligibility traces are an approximate gradient if used during the trajectory the same way RTRL is. This suggest that the fix proposed in [4] could be used to get an exact real time gradient in linear recurrent neural network. It is interesting to see that the same parameter is used (μ) to determine how to weigh the previous value function and propagate the gradient back in time. Intuitively this makes sense. This parameter represent the horizon at which the algorithm want to look back in time. If the value function is robust then it is able to look further in time (high μ). This also mean that solving the problem of learning long term dependencies is very similar to the credit assignment problem. Indeed if the value function is really bad then μ will be low and the information won't propagate far in time making it hard to learn long term dependencies. If the value function is robust, it is possible to look further in time to attribute credits to the responsible state.

5.2 Bias-variance trade-off

We can also analyze the trade-off above from a bias-variance perspective. Indeed if μ is high the the bias induced by the momentum is high but the variance is reduced as we average over more values. However from the eligibility traces point of views this will increase variance and reduce the bias (look further in time). Having a low μ means that we do not trust a lot the previous value function which from the momentum's perspective introduce variance and reduce bias. From the eligibility traces perspective it introduce bias and reduce variance however. We have opposite effect. In general we believe it is more efficient to have a μ as high as possible as it implies we trust the value function, can average over a lot of value and simplify the credit assignment problem.

5.3 Gradient

The first point to note is that by being able to learn μ with the gradient we also developed a method to do λ gradient based learning. Indeed as we saw earlier their value are tied. The first experimental result suggest that the optimal way of setting those 2 parameter is when they have the same value supporting the theory we developed earlier. By viewing eligibility traces as a gradient we can consider the literature developed to learn long term dependencies in recurrent neural network. Long short term memory cells enables learning long term dependencies by gating the hidden unit and restricting its modification. This yield μ equals to 1 most of the time. The interesting thing is that the gating process is based on the input and value of the previous value function. From a reinforcement learning perspective this means that we would have a state-dependent lambda like in emphatic TD [5].

6 Questions

- Is the connection in 4.3 clear and correct ?
- Experiment idea for λ gradient learning ?
- I call the proposed method momentum because it makes me think of the momentum in optimization but because it might not be the same thing I am not sure this a good name. Any thoughts/suggestion ?
- is the use of μ as my parameter okay ?

References

[1] Tsitsiklis, John N., and Benjamin Van Roy. "An analysis of temporal-difference learning with function approximation." IEEE transactions on automatic control 42.5 (1997): 674-690.

[2] Williams, Ronald J., and David Zipser. "Gradient-based learning algorithms for recurrent networks and their computational complexity." Backpropagation: Theory, architectures, and applications 1 (1995): 433-486.

[3] Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. Vol. 1. No. 1. Cambridge: MIT press, 1998.

[4] Ollivier, Yann, Corentin Tallec, and Guillaume Charpiat. "Training recurrent networks online without backtracking." arXiv preprint arXiv:1507.07680 (2015).

[5] Van Seijen, Harm, et al. "True online temporal-difference learning." Journal of Machine Learning Research 17.145 (2016): 1-40.

[6] Mahmood, A. Rupam, et al. "Emphatic temporal-difference learning." arXiv preprint arXiv:1507.01569 (2015). APA