Reinforcement Learning– Intelligent Weighting of Monte Carlo and Temporal Differences

Martin Christiansson

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6072
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

© 2019 by Martin Christiansson. All rights reserved.
Printed in Sweden by Tryckeriet i E-huset
Lund 2019

1

Abstract

In Reinforcement learning the updating of the value functions determines the information spreading across the state/state-action space which condenses the value-based control policy. It is important to have an information propagation across the value domain in a manner that is effective. Two common ways to update the value function is Monte-Carlo updating and temporal difference updating. They are two extreme cases opposite of another. Monte-Carlo updates in episodic manner where fully played out episodes are used to collect the environment responses and rewards. The value function gets updated at the end of every episode. Monte-Carlo updating needs a large amount of episodes and time steps to converge to an accurate result which is of course a downside. However, the positive is that it will be an unbiased approximation of the value function. In circumstances like simulations and small real world problems it can be applied successfully. However, for larger problems it will cause problems regarding learning time and computer power. On the other hand, by use of temporal difference updating one can in some cases achieve a more effective spreading of information across the value domain. It uses, in contrary to Monte-Carlo update, an incremental update at every time step with the newest information together with an approximation of the expected total discounted accumulated reward for the rest of the episode. In this way the agent learns at every time-step. This leads to a more effective updating of the Q-value function. However the downside is that it introduces biases due to the approximation. Another drawback is that the algorithm only passes information one time step backward in time. By combining Monte-Carlo and Temporal-Difference update the best of the two can be exploited. A popular way to do that is by weighting the importance of the two. The method is called $TD(\lambda)$ where the λ variable is a tuning parameter "how much to trust the long term update vs. the step wise update. $TD(\lambda = 0)$ takes one step in the environment, bootstrapping the rest and updates. $TD(\lambda = 1)$ updates with received rewards and hence it does not make use of any approximation. A value of λ in between is weighting the importance of the two. The optimal choice of λ depends on the specific situation and is dependant on many factors both from the environment and the control problem itself. This thesis proposes an idea to intelligently choose a proper value for λ dynamically together with choosing the values

of other hyper parameters used in the reinforcement learning strategy. The main idea is to use a dropout technique as an inferential prediction for the uncertainty in the system. High inferential uncertainty reflects a less trustworthy Q-value function and tuning parameters can be chosen accordingly. In situations where information has propagated throughout the network and bounds the inferential uncertainty for example a lower value of λ and ε (exploit versus explore parameter) can hopefully be used advantageously.

2

Acknowledgement

Fist and foremost I like to thank my supervisor Fredrik Bagge Carlson and my examiner Anders Robertsson. They have supported me along the way and contributed with guidance and valuable information in order to proceeding in the thesis.

Contents

1. Abstract **3**

2. Acknowledgement **5**

3. Introduction **9**

4. Background **10**

5. Programming language, tools and packages **11**

6. Theory **12**

6.1 Reinforcement learning 12

6.2 Upscaling using Neural networks 15

7. Method **18**

7.1 Limitations 18

7.2 Uncertainties 18

7.3 Normalized variance 19

7.4 How to choose λ 20

7.5 Network structure 21

8. Results **22**

9. Discussion **36**

10. Conclusion **38**

10.1 Future work 38

Bibliography **39**

3

Introduction

Reinforcement learning is a branch within artificial intelligence and machine learning. Its main idea is to learn via trial and error. An example could be a robot, a so called agent, with the ability to sense and act in its environment. In the environment the agent receives rewards based on its actions and a predetermined reward function. The goal is to maximize the total discounted accumulated reward along the agent's trajectory and is achieved by trial and error. The reinforcement learning method can be applied to many problems and in many situations, e.g. self-playing computer programs within the gaming industry and for controlling systems where it is otherwise difficult with other classical control algorithms. However, reinforcement learning faces many difficulties regarding learning time, state space and control space dimensions. In practical scenarios this could be more or less unmanageable. The reinforcement learning technique often utilizes neural networks for modeling the so called Q-value function in the reinforcement learning structure. In order to have an efficient learning algorithm it is important to have fine tuned hyper parameters within the learning algorithm. The optimal choice for these parameters differs for every system and situation and they also depend deeply on the uncertainties within the process and model. Knowledge of system uncertainties is also of great value when to make decisions that could be of higher priorities or importance. Developing a method for modeling uncertainties within the system and based on this information choose optimal hyper parameters is the scope of this thesis.

4

Background

An important factor for all learning is the ability to estimate our own level of skill. Knowing that you may not be the best in for example bicycling, dancing or any other task you may think of, the very first time you do it. This self-awareness leads us in the learning process. It is a guide telling us when to thrust our already gained knowledge and use it, or to try out new things to correct for the mistakes. In artificial intelligence, a sort of guidance in the learning process could be the "awareness" of uncertainties within the system. Noise, randomness, learning time are examples of factors effecting the system uncertainties. Learning algorithms including some sort of estimation mechanism for the system uncertainties would therefore be beneficial. The system in this case is the neural network, in conjunction with the physical system using the reinforcement learning technique. The estimation is done using dropouts for all neural network layers except the last layer. Estimation by feed forwarding the network a number of times using randomized dropouts produces a prediction distribution with a mean value and its associated variance.
A normalized version of the variance is used to tune a weighting factor in the reinforcement learning technique to achieve faster learning.

5

Programming language, tools and packages

All coding is done in Python programming language version: Python 3.5. Programming editor: Pycharm from Jetbrain [7]. Used Python packages:

- tensorflow [8]
- numpy [9]
- OpenAIGym [10]
- matplotlib [11]

Results based from OpenAIGym environment:

- CartPole-v0 [12], see fig. 8.21.

6

Theory

6.1 Reinforcement learning

Reinforcement learning is a branch within artificial intelligence and machine learning. The idea is to learn by trial and error. Usually the so called agent tries different things in its environment, receives rewards either directly or delayed which is a measurement of how good its actions have been. The agent then learns to take the best actions to maximize the rewards. There are many ways to implement an reinforcement algorithm which we shall discuss in this section.

The Bellman equation

The reinforcement learning method is based on solving the so called Bellman equation. eq. (6.1). The basic idea is to have a value function that describes how good it is to be in a specific state and to take a specific action in that state. Such a function could be used as a guide to take the best action in every state. An example could be a robot avoiding an obstacle by taking an appropriate action and receiving a reward for that action. This function, often denoted the Q-value function or in short the Q-function, is solved for in an iterative manner while the agent is interacting in its environment. In order to solve the Q-function in the whole state-action space, the agent must be testing different actions in the environment. This is often done to some degree by letting it randomly acting in the environment. A technique often used is the ε-greedy [1], which is a control policy where the agent is taking random action with a probability of ε, and takes the best action according to the Q-function, for the rest.

Monte Carlo sampling.

In reinforcement learning the Bellman equation is solved by letting the agent take a sequence of actions for either a predetermined number of steps or for an undetermined number of time steps. Monte-Carlo sampling [1] or in short MC-sampling, makes use of undetermined number of time steps, or until reached episode termination before update of the Q-function is done. Rewards along the trajectories are

Equation 6.1 Bellman equation for the evolution of the Q-value function, where s,a is the current state and action, respectively, whereas s',a' represent the next state and action, γ is the discount factor, and r is the received reward for action a in states. $\mathbb{E}$ meaning the expectation value of the Q-function.

$$\mathbb{E}\{Q(s,a)\} = \mathbb{E}\{r_{t+1} + \gamma Q(s',a')\} \tag{6.1}$$

summed with a discount factor γ, see eq. (6.2). The G_{MC} is then used for iterative update of the Q-function. see eq. (6.3).

Equation 6.2 Monte-Carlo target.

$$G_{MC(t+1)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^{(end-1)} r_{end} \tag{6.2}$$

Equation 6.3 Q-function at a state-action pair updates with learning rate α every time that state-action pair is being visited.

$$Q(s,a) \leftarrow Q(s,a) + \alpha(G_{MC} - Q(s,a)) \tag{6.3}$$

By solving the Bellman equation using Monte-Carlo sampling for the target, one gets an unbiased, converging estimate for the Q function. However, it suffers from high variance from stochasticity along the many state transitions [1]. A big disadvantage with Monte-Carlo sampling is that it takes a relative large amount of time steps in order to converge to an accurate approximation of the true Q function which can be unfeasible for real-world systems. Monte-Carlo sampling is also restricted to episodic tasks, meaning that the game or task must be terminated before the update process can be carried out, which causes problems, because not all learning processes posses this episodic nature.

Temporal difference approximation.

Another way to compute the target is to take one action and receive a reward for that action, and then approximate the value of the state using Q as an estimate for the expected discounted accumulated reward for rest of the trajectory. The approximation is called bootstrapping and is just the value of the Q-function in the next state after the transition. The value of the Q-function at this state depends on the chosen action at that state. If the action that maximizes the Q-function is used, it is the so called Q-learning algorithm [1], see eq. (6.4).

Equation 6.4 Target for Q-learning algorithm.

$$G = r_{t+1} + max_{a'} Q(s', a') \tag{6.4}$$

$TD(\lambda)$

The $TD(\lambda)$ [1] uses a weighted sum of all n-step returns as a target for updating the Q-values, see eq. (6.5) and eq. (6.6). This gives the advantage of utilizing a mix of Monte-Carlo sampling and Temporal-difference approximation. In some situations, for example where the Q-functions is well known, it could be beneficial to use more of Temporal-difference updating, and likewise for situations where the Q-function is less known, it could be more useful to trust more on the Monte-Carlo strategy. $TD(\lambda)$ gives the possibility of blending the two techniques.

Equation 6.5 n-step return G_n.

$$G_n = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{n-1} r_{t+n} + \gamma^n max_{a'} Q(s', a') \tag{6.5}$$

Equation 6.6 G_λ return. Where G_n is the n-step return, and λ, is the weighting factor that controls the contribution of each n-step return.

$$G_\lambda = (1 - \lambda) \sum_{n}^{\infty} \lambda^{n-1} G_n \tag{6.6}$$

Reinforcement learning using tables

In smaller problems with finite and not too large numbers of state-action pairs one can use a table to represent the Q-value function. Hence the Q-function hence has the dimension [observable states $\times$ number of actions]. Equation (6.3) and eq. (6.7) show how the Q-function is updated. The parameter α is the so called learning rate which determines how much to update in the direction of the return.

Equation 6.7 Q-learning update with learning rate α and discount factor γ

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r_{t+1} + \gamma Q(s', a'_{max}) - Q(s,a)\big) \tag{6.7}$$

6.2 Upscaling using Neural networks

Deep networks

In large problems, use of tables to represent the Q-function will cause problems because these tables will quickly grow to extremely large sizes when used to describe real and practical systems. Neural networks are general function approximators [2] and could therefore be used in reinforcement learning problems to model the complex Q-function where Q-tables otherwise would be of extensive unmanageable sizes. The approach is to have all states needed to represent a Markov process (or at least approximate a Markov process) together with the available actions as inputs to the neural network. The output is in this case a single scalar value to represent the Q-value for that state. This reduces the extremely large Q-tables to a set of fewer parameters namely the weights of the neural network to describe the Q-function.

Optimizers and stochastic training

The learning process of neural network is to update the weights in the direction which minimizes a certain cost function. The simplest way of doing this is using the vanilla Gradient descent optimizer [5] to find the direction of the weight updates. However, the gradient decent method is a rather slow method to find optima and therefore more advanced methods have been developed to make more effective optimizers, like the Adam-optimizer and the RMS-prop optimizer which make use of information from prior steps. They are so called momentum optimizers [4]. Stochastic batch update is yet another technique in the optimizing process and is used to reduce the computations cost and to get more stable weight updates. It works by having the neural network not learn from all data points but instead from a random set of data points at every iteration. Stochastic batch updates converge to at least local minima as well as learning from all data points.

Q-learning algorithm with neural networks

The Q-learning algorithm, see eq. (6.7), uses an approximation of the future discounted accumulated rewards in order to have a value to update the Q-function at every iteration [1]. The update is also off-policy due to the bootstrapping is based on the action that gives maximum expected reward and not by following the control policy itself, for example epsilon-greedy [1]. When modeling the Q-function by a neural network one can not directly update a state-action pair like it is done in the table case. Instead it is updated using the gradient of the loss function with a stored batch of transitions, see eq. (6.8)

Monte-Carlo learning with neural networks

Monte-Carlo update with neural networks uses the same strategy as for Monte-Carlo update with tables, i.e. by having an unbiased target to update against. Monte-Carlo is playing out the whole episode and stores all rewards along the trajectory

Equation 6.8 Loss function L to minimize. ω_n is a set of fix weights that get updated in a predetermined number of iterations. ω are the current weights.

$$L(\omega) = \frac{1}{2}\sum_{batch}(r_{t+1} + \gamma Q(s', a'_{max}, \omega_n) - Q(s, a, \omega))^2 \tag{6.8}$$

and then update without any approximation. The learning is done by batch update using a predetermined number of stored rewards and transitions sampled from the experience memory, see eq. (6.9).

Equation 6.9 Loss function L with target G_{MC}

$$L(\omega) = \frac{1}{2}\sum_{batch}(G_{MC} - Q(s, a, \omega))^2 \tag{6.9}$$

$TD(\lambda)$ algorithm with neural networks

The $TD(\lambda)$ uses a weighted sum of all n-step returns, see eq. (6.5) and eq. (6.6) as a target for updating the Q-values.

Equation 6.10 Loss function L with target G_λ.

$$L(\omega) = \frac{1}{2}\sum_{batch}(G_\lambda - Q(s, a, \omega))^2 \tag{6.10}$$

Eligibility traces

The drawback of using $TD(\lambda)$ is that even if the information is available it will not be used until after the full episode is played out. It would be more far more efficient to make use of the newest information and train the network at every time step. Eligibility traces solve this problem and it can be shown to be approximately mathematically equivalent to the $TD(\lambda)$ algorithm with the crucial difference that it is updating the value function at every time-step [1]. The idea is to update the network at every visited state-action pair in proportion to the so called TD-error for the current state-action pair. This is done using saved gradients at every visited state, and letting them exponentially decay by the factor $\lambda\gamma$, see eqs.(6.11-6.13).

Equation 6.11 TD-error

$$\delta = r + \gamma Q(s', a') - Q(s, a) \tag{6.11}$$

Equation 6.12 Eligibility trace, where $\nabla_{\omega} Q(s, a, \omega)$ is the gradient of the Q-function at s, a with respect to the network weights.

$$E_{t+1} = \lambda \gamma E_t + \nabla_{\omega} Q(s, a, \omega) \tag{6.12}$$

Equation 6.13 Weight update with eligibility trace

$$\omega \leftarrow \omega + \alpha \delta E \tag{6.13}$$

7

Method

7.1 Limitations

For this thesis, investigations and testing of how varying λ based on the system variance affects the learning algorithm performance. At every iteration, the weights are updated for the latest visited state, i.e. no batch update is used. All results are based on the OpenAIGym game: CartPole-v0 [12].

7.2 Uncertainties

Noise in the process along with noise and errors related to updating and training the neural network model will introduce errors and uncertainties in the system. It is important to be able to estimate these uncertainties for many reasons. Knowledge about the system uncertainty can make the learning process more efficient. It can also be used to decide how much to explore versus exploit in the learning process. Every system has its own inherent noise due to stochastic mechanisms in the environment and hence the Q-value function is described as an expectation value and has its associated variance.

Modeling uncertainties with dropouts

Dropout is a technique often used to avoid overfitting when training neural networks [3]. This technique will in this case be used as an inference model to approximate the noise in the Q-function [6]. The network is trained in usual manner with dropout. The network is then evaluated by making a predetermined number of predictions of the Q value. Letting the Q-function predict the Q value for the same state many times with random dropouts will be equivalent to do evaluations by different networks with slightly different weights. This evaluation results in a prediction distribution over the values which tells something about the uncertainties in the system. Low variance in the Q value predictions, presumably reflects low level of uncertainties in the system. It also means that the network weights have been updated in such a way that the network is robust to slight changes in the input

space. This occur when training on consistent data with low noise levels. However, a lot of factors affect the Q-value prediction distribution and variance other than just the stochasticity and noise in the training data. In the result section figures reflect the effect of these factors, see list below.

- Weight initialization
- Input value to network
- Magnitude of Q-value to predict
- Number of training iterations
- Denseness of data points
- Noise and stochasticity in rewards
- Dropout percentage
- λ value (Higher variance in the rewards for longer trajectories.)
- Number of evaluations (to get a statistically satisfying approximation)
- Possible others

Understanding the impact of every factor on the variance is crucial in order to make use of the information about the uncertainty. In order to visualize the effects of these factors on the predicted variance for the system, a 2D regression problem is constructed which makes it possible to plot the variance for the states.

7.3 Normalized variance

Using a normalization of the variance reduces the correlation between input value and prediction as well as the correlation between magnitude of Q-value and prediction. This can be seen in figs.(8.5-8.8). However, this will not work for all systems. If the mean value of the predictions is $<< 1$ problems can arise regarding division by a small number close to 0 in the variance computations which causes variance explosion. This problem needs to be addressed in order use the normalized variance as a measure of the Q-function quality.

Equation 7.1 Normalized variance. x is predicted value, μ is the expectation value.

$$\sigma^2_{normalized} = \frac{1}{n}\sum((x-\mu)/\mu)^2 \tag{7.1}$$

The variance explosion problem is in this case addressed by replacing the division by μ in eq. (7.1) by 1 if $|\mu|$ is less than 1. There are other ways of doing this, i.e. normalizing the variance using $\sqrt{1+\mu^2}$. The first one is chosen for simplicity.

7.4 How to choose λ

If we predict high uncertainties in the Q-value function i.e., the dropout evaluation method infers high variance, the algorithm weights the importance of the unbiased Monte-Carlo updating method higher while for lower variance the algorithm utilizes to greater extent to the temporal difference method. The idea of this approach is, that for systems with high uncertainties, the best guess for the Q-value would be of less biased character. Even if this choice is suffering from high variance for the reward return, its mean return is presumably closer to the true value than if a biased Temporal-difference approximation is used. When the Q-function begins to represent the true reward returns more accurately, the update relies more on the nearby Q-values as an approximation for the rest of the trajectory, in order to tune the Q-function. In this example of the inverted pendulum, the λ value is fitted for the specific typical variance for the system which is in this case measured beforehand. This means that a low typical system variance yields λ values close to zero while high typical system variance results in λ values closer to 1. A linear mapping for λ is done for system variances in between. Also a moving average filter for λ is used to decrease the effects of fluctuations in the variance approximation and to avoid updating cutoffs that can occur if λ rapidly switches from 1 to 0 and cutting off the gradients to be carried throughout the eligibility trace.
The eligibility trace method inherently correlates λ to weight update step length. This is due to the trace built up from previous steps leading to greater weight updates with greater values of λ. Therefore a normalized version of the eligibility trace is used at every update step along with the TD error and learning rate α, see Algorithm 1 below. This will decouple the dependence between λ and step length and it will also reduce the risk of overshooting minima in the loss function. This is further reduced by letting the learning rate α decrease exponentially.

Algorithm 1

---

initialization:
while *episode not terminated* **do**
  Take action a according to ε-greedy
  compute: $\nabla_{\omega} Q(s,a,\omega)$, δ, λ, v (normalized variance)
  $E = \lambda \gamma E + \nabla_{\omega} Q(s,a,\omega)$
  $E^{'} = E/\mathrm{norm}(E)$
  $\omega \leftarrow \omega + \alpha \delta E^{'}$
  $\varepsilon \leftarrow \varepsilon_{decay} \times \varepsilon$
  $t \leftarrow t + 1$
end

7.5 Network structure

Two different network structures are chosen: one for the regression problem and one for the reinforcement learning problem.
The regression network has one input neuron and five fully connected hidden layers with size of 100 neurons with ReLu activation functions, where the Q-function network has five input neurons and three fully connected hidden with size 100 neurons with ReLu activation functions. The output layers have the dimension of a scalar for prediction of the y-coordinate value for the regression problem and the Q-value for the reinforcement learning problem. Both networks are trained using dropouts in each hidden layer.

8

Results

Results are presented as figures together with explanatory text. In the testing procedure analysis was made for how the factors in section 7.2 affect the variance. For demonstration and visualization a regression problem was implemented. A 2-D visualization can be seen in figs.(8.1-8.2). For the evaluation of the reinforcement learning algorithms, the environment CartPole-v0 [12] was used, see fig. 8.21.

In the process of testing the different reinforcement-learning algorithms outliers were present in every method. In these cases the performance went down to a level compared to not learning at all. This occurred approximately 20% of all evaluating runs. They were removed as outliers and are not part of the presented results. For every run of a learning algorithm its corresponding network weights were randomly initialized. This caused slightly different predictions of the Q values and hence cause different initial variance. This can be seen in fig. 8.3 and fig. 8.4.

Network input and the magnitude of the value to predict is correlated to the variance in the network predictions which can be seen in figs.(8.5-8.8). In order to decouple or reduce the impact of input and magnitude on the variance, a normalized variance is used, see eq. (7.1).

The variance and normalized variance is dependant on number of training steps performed on the network which can be seen in fig. 8.9, and fig. 8.10.

The amount of data the network is trained on is of great importance and therefore the denseness of data points has an impact on the normalize variance, see fig. 8.11.

If the network trains with noisy data the normalized prediction variance increases in most cases. There are situations where noisy data can cause less variance which are in those situations where the Q-function is complex. The result is a smoothness effect which leads to less variance, see fig. 8.12.

A factor that highly effects the variance is the dropout percentage in the layers of the network, see fig. 8.13.

In figs.(8.14-8.15) the learning curve is shown for the inverted pendulum, using Eligibility Traces with $\lambda = 0$, and $\lambda = 1$ respectively. In fig. 8.16 the performance of the normalized-variance based λ is presented. The variation of the λ during the learning can be seen in fig. 8.17. The variation of λ is correlated to abrupt changes

in the state space, this is shown in fig. 8.18. Comparison between fix λ-values and the variance-based λ method is presented in fig. 8.19.

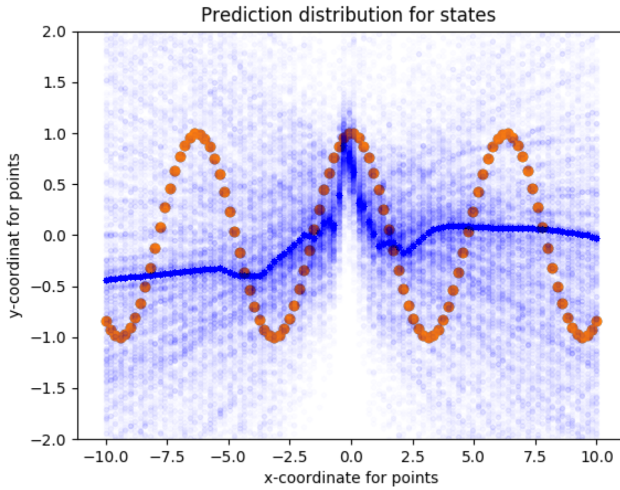

Figure 8.1 Network prediction distribution of a cosine function. Network has trained for 100 training steps.

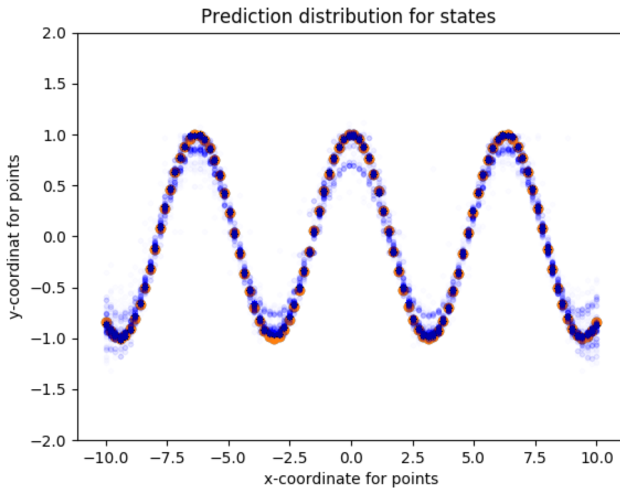

Figure 8.2 Network prediction distribution of a cosine function. Network has trained for 10000 training steps.

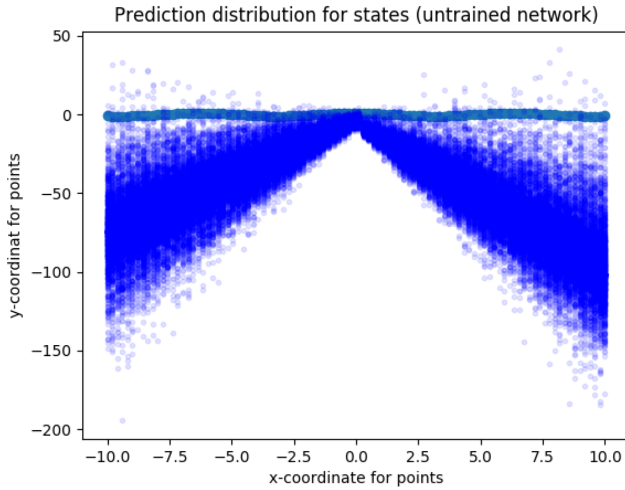

Figure 8.3 Weight initialization. Compare fig.(8.4) for slightly different prediction distribution. The thick and slightly curled line is here the cosine curve overlayed as a reference.

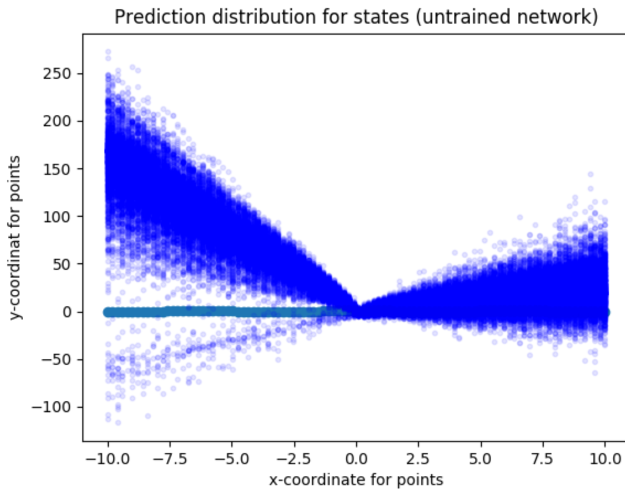

Figure 8.4 Weight initialization. Compare fig.(8.3) for slightly different prediction distribution. The thick and slightly curled line is here the cosine curve overlayed as a reference.

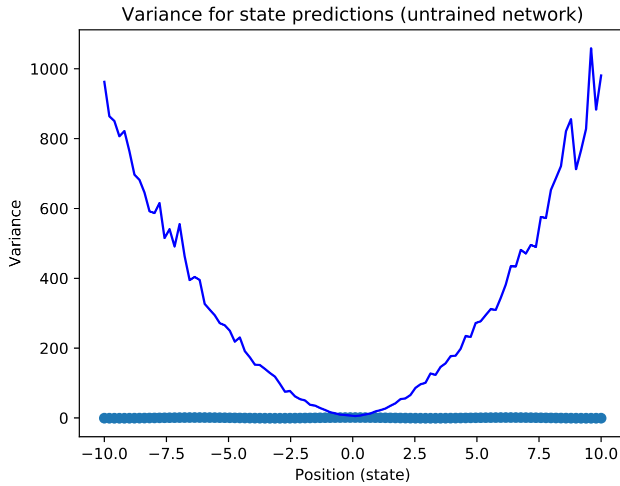

Figure 8.5 Variance for state predictions for a regression problem. Given x-coordinates the y-value is to be predicted for a cosine curve. The network is untrained for the problem. Value of input and prediction is strongly correlated. The thick and slightly curled line is here the cosine curve overlayed as a reference.

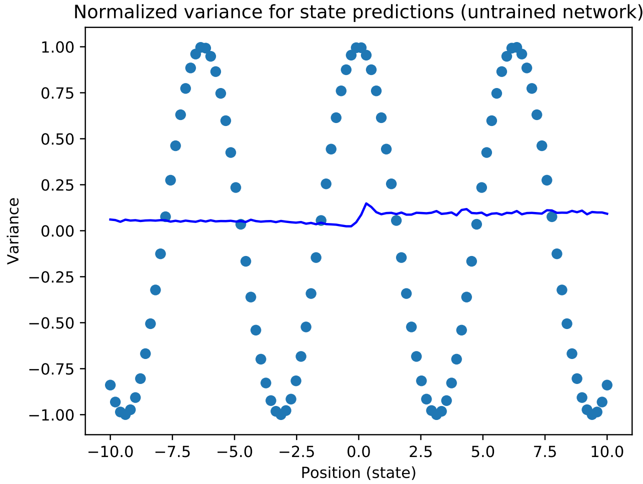

Figure 8.6 Normalized variance for state predictions for a regression problem. Given x-coordinates the y-value is to be predicted for a cosine curve which is here overlayed as a reference. The network is untrained for the problem. No correlation between input and prediction can be seen.

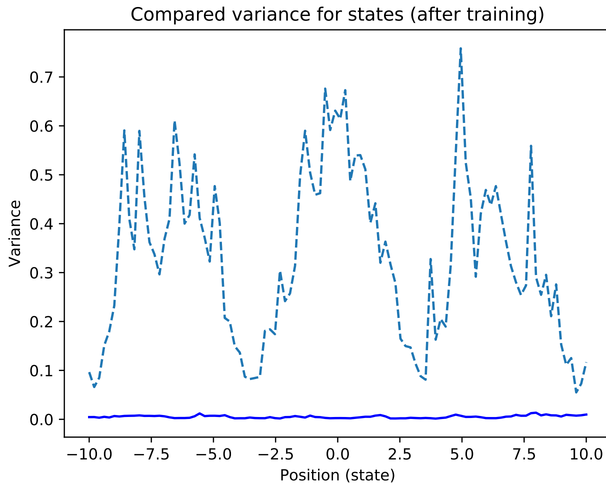

Figure 8.7 Compared prediction variance for $\cos x + 2$ (solid line), $10 * (\cos x + 2)$ (dashed line). This shows that the magnitude of the value to predict is strongly correlated to the variance.

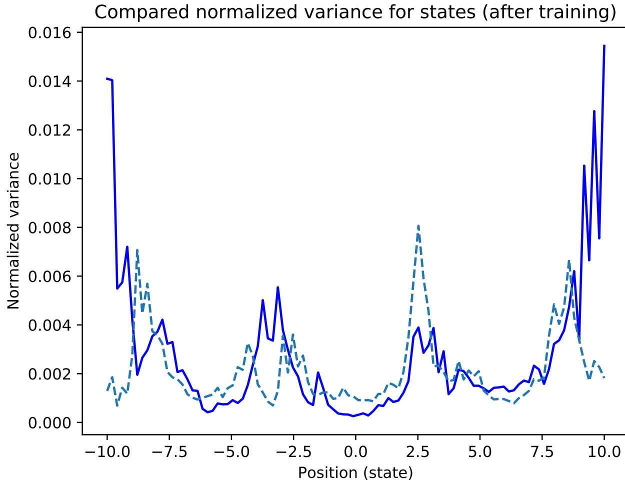

Figure 8.8 Compared normalized prediction variance for $\cos x + 2$ (solid line), $10 * (\cos x + 2)$ (dashed line). This shows that the normalized variance is not correlated to the magnitude of value to predict. The periodic nature of the normalized variance reflects the network's uneven performance in predicting the function.

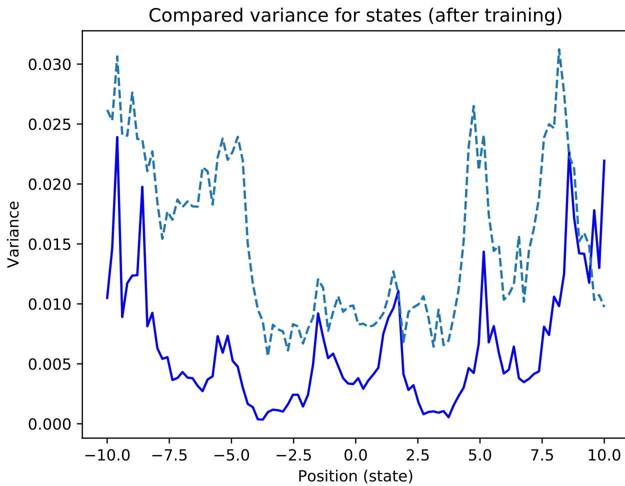

Figure 8.9 Compared variance for prediction of $\cos x + 2$. Network trained 1000 times (dashed line), network trained 10000 times (solid line).

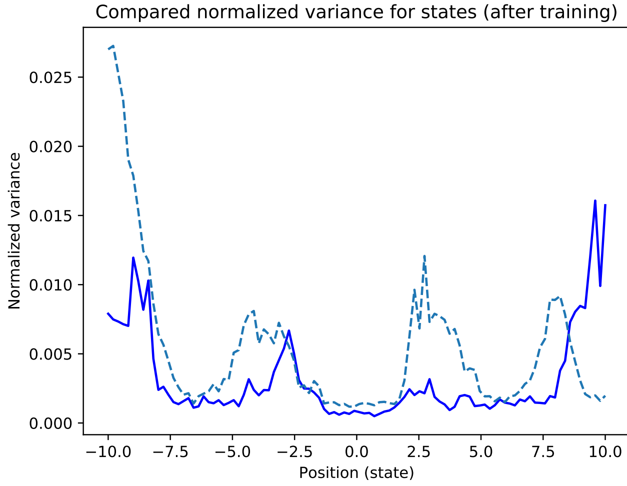

Figure 8.10 Compared n variance for prediction of $\cos x + 2$. Network trained 1000 times (dashed line), network trained 10000 times (solid line).

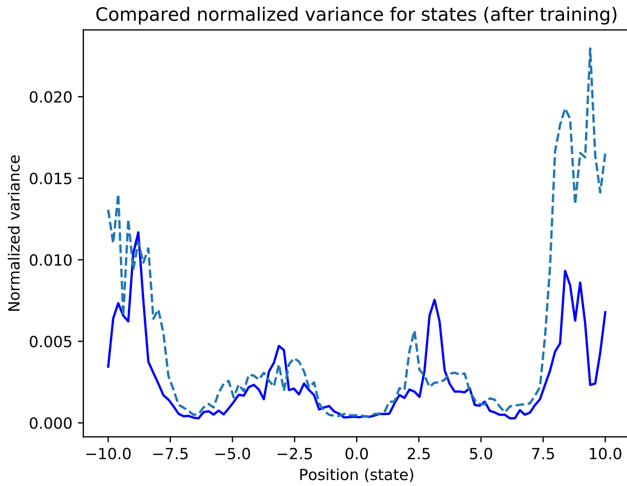

Figure 8.11 Compared normalized variance, network trained with denser points (solid line), network trained with 50 percent denseness (dashed line). Generally denser points yield less variance except for in some cases where the function is complex and fewer points smooth out the complexity.

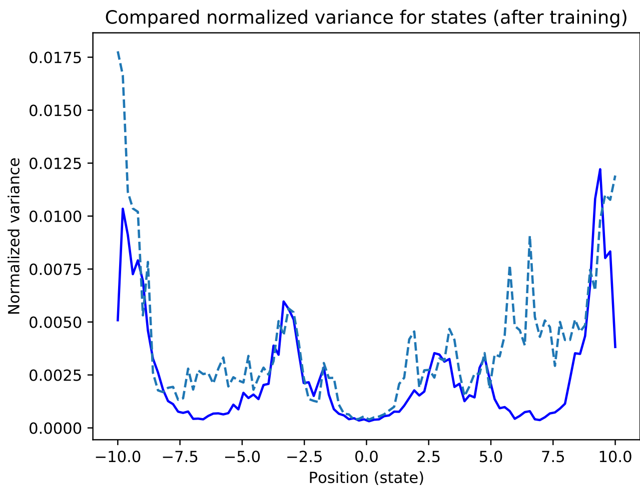

Figure 8.12 Compared normalized variance. Network trained with noise twice the magnitude (dashed line) of the solid line.

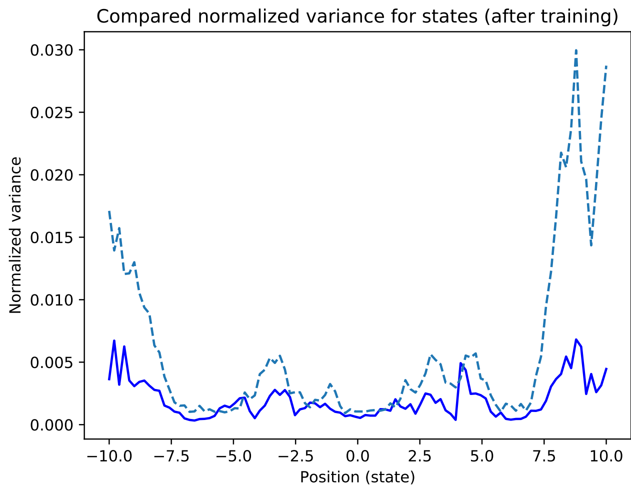

Figure 8.13 Compared normalized variance, network trained with dropout rate of 0.01 (solid line), network trained with dropout rate of 0.02 (dashed line)

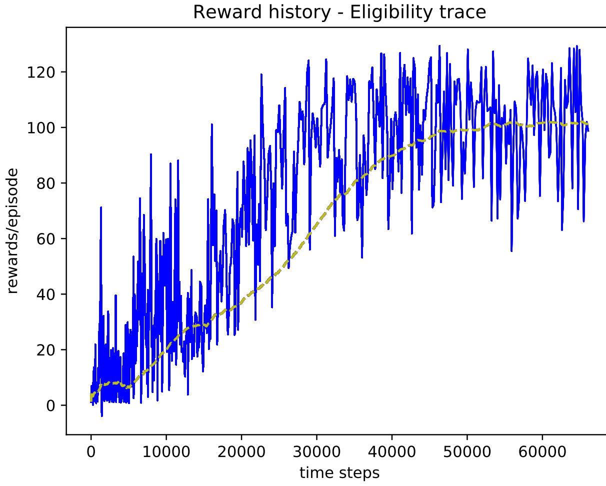

Figure 8.14 Reward history for eligibility trace using $\lambda = 0$. Dashed line is moving average using the rewards from the 100 latest episodes.

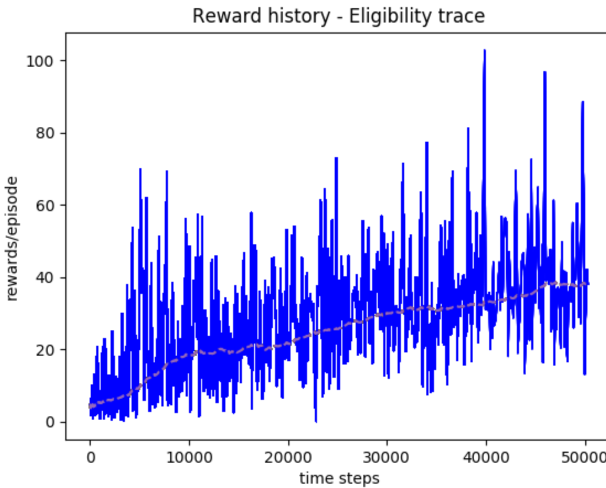

Figure 8.15 Reward history for eligibility trace using $\lambda = 1$. Dashed line is moving average using the rewards from the 100 latest episodes.

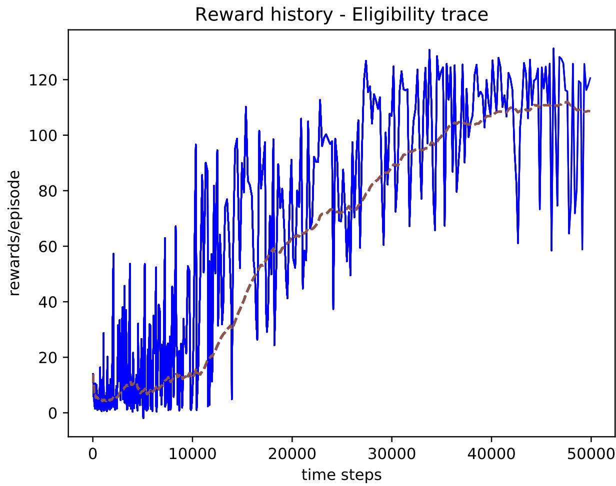

Figure 8.16 Reward history for eligibility trace using normalized variance-based λ. Dashed line is moving average using the rewards from the 100 latest episode.s

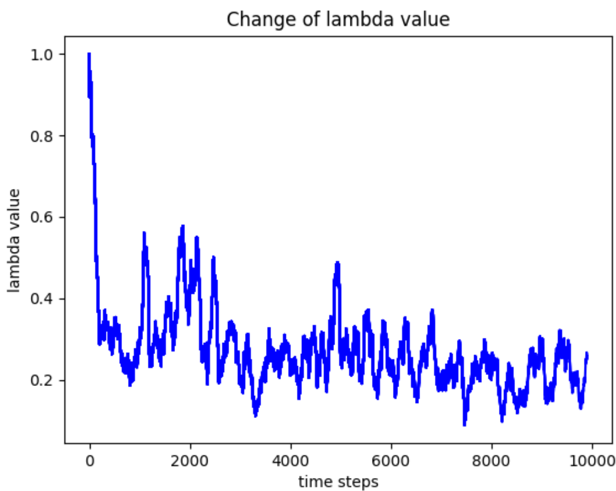

Figure 8.17 Variation of λ, using the variance-based λ learning algorithm.

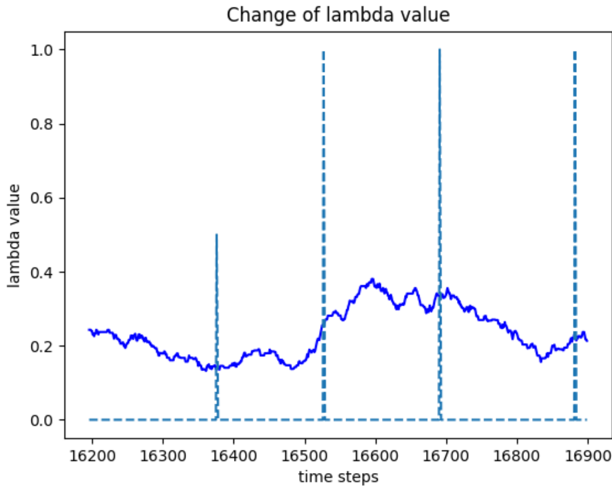

Figure 8.18 Variation of λ value. Spikes with value of 1 where the episode ends by pendulum falling down, and spikes with value of 0.5 when the episode end by reaching the maximum duration. When the episodes ends with the pendulum falling down it seems that predicted normalized variance reaches a local maximum, and for episodes ending by reaching maximum duration the variance seems to be unchanged. The oscillating pattern is linked to abrupt changes in state-space, which occur when the CartPole-v0 [12] game terminates by the pendulum falling down, resetting the game and therefore changing the pendulum state from down position with high angular velocity to top position with no angular velocity. The cause is presumable that training on data in a subset of the state space causes a drift in the accuracy of the Q-value predictions in the rest of the state space.

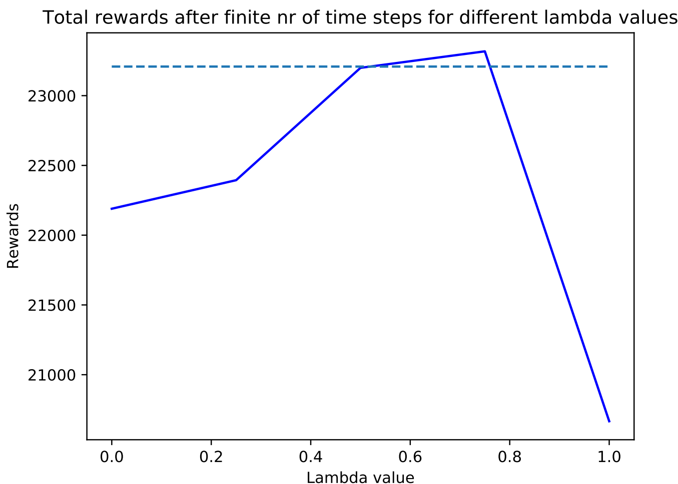

Figure 8.19 Total rewards after a finite number of time steps for each method. Dashed line represents the normalized variance-based λ method. Result presented is the average of 10 method evaluations.

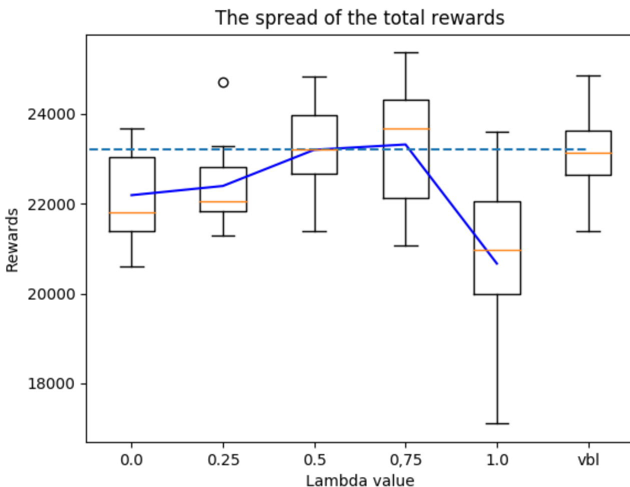

Figure 8.20 The box diagram shows how the total rewards are spread out for the different methods. The fixed-λ method generally seems to have greater spread than the variance-based λ method, resulting it to be a more robust method than the others.

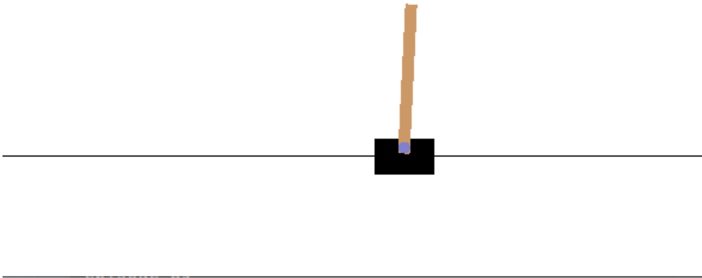

Figure 8.21 OpenAIGym game CartPole-v0 [12]. Control signal acts on the cart (the base) which for this game environment could be either 0 or 1, representing a discrete force on the cart in the leftmost direction and rightmost direction, respectively. The measured states are cart position, cart velocity, angle and angle velocity between the pendulum and the cart.

9

Discussion

From the figures in the result section, correlation between variance and the learning process is concluded. There are in most cases a direct relation between variance and poorly trained networks. Using this methodology a variance-based λ method is condensed. From figs.(8.3-8.13) conclusions can be made for how the variance reflects a poorly trained network. Normalized variance seems to be higher in cases where the network has been trained on noisy data, fewer and sparser data points and low number of training iterations. This information is used as a guidance for the tuning of λ. When comparing the normalized variance-based λ method to fixed-λ methods, it seems that the performance of it is equivalent to the best tuned λ value, see fig. 8.19. The reason for this could be that this method utilizes a better guess for the Q-value when there are high uncertainties for its true value, while shifting to an updating strategy that relies more on local Q-values yielding less variance, when the Q-function is assumed to be a better approximation of the true nature of the system. The learning curves for fixed-λ methods with $\lambda = 0$ and $\lambda = 1$, respectively, are shown in figs.(8.14-8.16), in comparison to the learning curve for the normalized variance-based λ method, see fig. 8.16.

In fig. 8.17. changes in λ during the learning are shown. It seems like λ is changing periodically which reflect the specific nature of the pendulum. In the case of the OpenAIGym Cartpole-v0 [12] an episode termination is represented either by the pendulum falling down, or if it reaches maximum time duration. Variance seems to be unaffected if the episode terminates by reaching the maximum time duration, while reaching local maxim when episodes are terminated by falling down, see fig. 8.18. The reason for this could be that when training on data that are consistent, (pendulum in vicinity of the top position for most of the episode) together with consistent Q-values the variance decreases, where for episode terminations where the pendulum is falling down the abrupt changes in the state yield less consistent data. The mechanism could be that the network is trained for a period of time on data that lies is a subspace of the state space domain, (pendulum in top position with low angular velocity) which causes the weights in the network to update in the manner to minimize the loss function for those states while in some sense the weights get altered for the other input values of the network, which results in a higher variance

when the pendulum enters the region of states where it is less trained recently. If this is the case, stochastic training could be used to break the asymmetric training.

For this setup, that is, the game environment, network structure, dropout rate and system parameters, this method didn't produce a more efficient learning process, However, the performance of the method is equivalent to $TD(\lambda)$ with optimal choice of λ, which is reflected in fig. 8.19. The variance-based λ method seems to have less spread in the total rewards and is therefore a more robust method than for fixed-λ methods, see fig. 8.20.

The nature of the reinforcement learning process is its convergence to the true value function after a sufficient amount of iterations. This is extrapolated to that the variance itself is a measure of the correctness of the value function. This however may not generally be true and could be the reason that the method did not yield better performance for this specific setup.

10

Conclusion

This thesis proposes an idea to an intelligent way of weighting the importance and tradeoffs between variance and biases in reinforcement learning using eligibility traces. By approximating system variance with dropouts in the neural network a measure of the quality of the Q-value function is determined.
The Quality of the neural network is a variable to choose the λ value in the update process. Results show correlations between approximated system variance and different factors listed in the method section. In order to avoid the influence of certain factors, a normalized variance is used. Testing and evaluation show that this proximate measure of uncertainty in a system, and by use of this measure in the way it is done at least for this specific setup is not leading to a more efficient learning algorithm. However, it is more robust than the other methods and the performance is equivalent to $TD(\lambda)$ with optimal choice of λ.

10.1 Future work

Further investigations could be useful to study, such as implementing algorithms that use experience replay [1]. Hopefully this could be a solution to small fluctuations in λ as mentioned in the discussion chapter. Various types of network structures and dropout rates could be important to investigate in order to understand the impact that those factors have on the variance estimation. This thesis is limited to tuning λ, whereas other hyper-parameters, ε, α, γ is not included. They are likely to be just as important to do further research upon.

Bibliography

[1] David Silver, "RL Course by David Silver", Deep Mind - Youtube, May 13, 2015.
https://www.youtube.com/watch?v=2pWv7GOvuf0&list=PL7-jPKtc4r78-wCZcQn5IqyuWhBZ8fOxT
Acceded date: 2018-01-10

[2] Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015
http://neuralnetworksanddeeplearning.com/
Accessed date: 2018-11-28

[3] Nitish Srivastava Geoffrey Hinton Alex Krizhevsky Ilya Sutskever Ruslan Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", Journal of Machine Learning Research 15 (2014) 1929-1958, Submitted 11/13; Published 6/14.
http://jmlr.org/papers/volume15/srivastava14a.old/srivastava14a.pdf
Accessed date: 2018-11-28

[4] MSProp (C2W2L07), Deeplearning.ai, Youtube, Published on Aug 25, 2017
https://www.youtube.com/watch?v=_e-LFe_igno
Acceded date: 2018-01-08

[5] Gradient Descent For Neural Networks (C1W3L09), Deeplearning.ai, Youtube, Published on Aug 25, 2017
https://www.youtube.com/watch?v=7bLEWDZng_M
Accessed date: 2018-01-10

[6] Yarin Gal, Zoubin Ghahraman
"Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning", University of Cambridge,
4 Oct 2016
https://arxiv.org/pdf/1506.02142.pdf
Accessed date: 2018-11-28

[7] "Pycharm IDE"
https://www.jetbrains.com/pycharm/
Accessed date: 2018-01-07

[8] "Tensorflow, software package for python"
https://www.tensorflow.org/
Accessed date: 2018-01-07

[9] "numpy, software package for python"
http://www.numpy.org/
Accessed date: 2018-01-07

[10] "OpenAIGym enviroment"
https://gym.openai.com/
Accessed date: 2018-01-07

[11] "matplotlib, software package for python"
https://www.google.com/search?client=ubuntu&channel=fs&q=mathplotli&ie=utf-8&oe=utf-8
Accessed date: 2018-01-07

[12] Cartpole-V0"
https://gym.openai.com/envs/CartPole-v0/, (updated to Cartpole-v1)
Accessed date: 2018-01-07

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER'S THESIS

Date of issue
February 2019

Document Number
TFRT-6072

Author(s)
Martin Christiansson

Supervisor
Fredrik Bagge Carlson, Dept. of Automatic Control, Lund University, Sweden
Anders Robertsson, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle
Control Reinforcement Learning – Intelligent Weighting of Monte Carlo and Temporal Differences

Abstract
In Reinforcement learning the updating of the value functions determines the information spreading across the state/state-action space which condenses the valuebased control policy. It is important to have an information propagation across the value domain in a manner that is effective. Two common ways to update the value function is Monte-Carlo updating and temporal difference updating. They are two extreme cases opposite of another. Monte-Carlo updates in episodic manner where fully played out episodes are used to collect the environment responses and rewards. The value function gets updated at the end of every episode. Monte-Carlo updating needs a large amount of episodes and time steps to converge to an accurate result which is of course a downside. However, the positive is that it will be an unbiased approximation of the value function. In circumstances like simulations and small real world problems it can be applied successfully. However, for larger problems it will cause problems regarding learning time and computer power. On the other hand, by use of temporal difference updating one can in some cases achieve a more effective spreading of information across the value domain. It uses, in contrary to Monte-Carlo update, an incremental update at every time step with the newest information together with an approximation of the expected total discounted accumulated reward for the rest of the episode. In this way the agent learns at every time-step. This leads to a more effective updating of the Q-value function. However the downside is that it introduces biases due to the approximation. Another drawback is that the algorithm only passes information one time step backward in time. By combining Monte-Carlo and Temporal-Difference update the best of the two can be exploited. A popular way to do that is by weighting the importance of the two. The method is called TD(λ) where the λ variable is a tuning parameter "how much to trust the long term update vs. the step wise update. TD($\lambda = 0$) takes one step in the environment, bootstrapping the rest and updates. TD($\lambda = 1$) updates with received rewards and hence it does not make use of any approximation. A value of λ in between is weighting the importance of the two. The optimal choice of λ depends on the specific situation and is dependant on many factors both from the environment and the control problem itself. This thesis proposes an idea to intelligently choose a proper value for λ dynamically together with choosing the values of other hyper parameters used in the reinforcement learning strategy. The main idea is to use a dropout technique as an inferential prediction for the uncertainty in the system. High inferential uncertainty reflects a less trustworthy Q-value function and tuning parameters can be chosen accordingly. In situations where information has propagated throughout the network and bounds the inferential uncertainty for example a lower value of λ and ε (exploit versus explore parameter) can hopefully be used advantageously.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-40

Recipient's notes

Security classification

http://www.control.lth.se/publications/