Reinforcement Learning & Neural Networks

Alexander Kjeserud

December 2, 2016

Preface

The reason why I chose Reinforcement Learning as a subject is, the concept of machines learning from experience. That machines, through simulation can surpass human experts at tasks designed for humans. My initial goal in this course was to use Reinforcement Learning to teach an agent to play Connect-4 on an average human level. I changed path after my adviser suggested I should do a more theoretical study. From there, neural nets was the main focus, until I came across David Silver's lectures on YouTube. Which led my research back to reinforcement learning. It has been educational. I have read my first academic article (eight in total). I have learned about old and new algorithm types, statistics, maths, computer science history, etc. Studying Reinforcement Learning has proven to be more diverse than expected. I would like to thank Roland Olsson for his time, and David Silver for his lectures.

Abstract This article covers the basics of Reinforcement Learning and Neural Networks. It explains common terms and concepts in these fields. What type of problems RL applies to, and RL techniques use to solve those problems.

1 Reinforcement Learning

1.1 What is Reinforcement Learning?

Reinforcement Learning(RL) in the broadest sense is a framework, consisting of an agent and its environment. An agent is considered that which through action has the ability to interact with its environment. The environment is everything the agent does not have direct access to, i.e. other agents, sensors which an agent receives input from, even the algorithm generating the reinforcement signal. When completing a goal, the agent is rewarded according to a reward function. The total reward received by the agent is called the return. An essential part of RL is to find a balance between exploitation and exploration. Exploitation being always selecting the action known to give the best reward based on current knowledge of the environment, also known as greedy action selection. Exploration is when the agent deviates from the greedy action policy by either selecting an action at random, based on how strong the evidence for a lesser action is, or some other definition. Reinforcement problems may be divided into several subcategories:

- trail or instant reward -based tasks: A trial is an environmental timestep. The agent gets an input from the environment. The agent does an action. The environment rewards/punishes the agent, e.g. roulette, optimizing fuel to oxygen ratio in a combustion engine.
- episodic tasks: These are recurring problems that has one or more start and termination states, e.g. Chess, train route.
- continuous tasks: are tasks that may run for an infinite amount of time, e.g. the stock market.
- a combination of the three

Ambiguity is largely represented in everyday life, RL handles this well. If multiple actions seems to be equally good in a situation, the agent will

depending on its implementation, either oscillate between the two actions or keep to one.

1.1.1 Agent

The role of the agent is to find an optimal action to take in a given situation, the situation is a representation of the environment's state. It does this by exploring the actions available to the agent at different states, favoring the actions granting the agent the highest reward. This state to action mapping is called the agent's Policy, it is the probability of taking an action in a state. When the agent is given a reward, it updates its policies so that the actions taken in the future will earn the agent more reward. An agent's input signal should have the Markov property, or be as close as possible to it. When the input signal is Markov, an action can be made from earlier evidence, regardless of how the agent got to a state or at what time it got there.

1.1.2 Environment

The environment is where the agent exists and interacts. The goal of a problem is often to reach a certain environmental state. The state of the environment may change at any timestep, due to interaction from an agent or some other influence.

1.2 Usages of Reinforcement Learning

1.2.1 Robotics

RL is used to teach robots to do certain tasks like autopiloting helicopters, teaching bipedal drones to walk, balancing of objects. RL has even been used to teach a robot arm to use a bullwhip with the precision to hit the flame of a candle.

1.2.2 Games

Gameplay is much used in RL research and real world applications alike. Many games have some feature that can resemble, or even tries to be as realistic as possible to real world tasks. This makes game ideal as either simulators or as model environments for an agent. Some games can be used as an interface to test agents against real human players. In the RL community

it is important to make as general agent as possible. That makes games a perfect benchmark tool. A good algorithm should solve multiple problems with as little modification as possible.

1.3 Reinforcement Learning Today

Although there has been made progress in the field of RL the last few years, algorithms developed in the 70-90's are still widely used. Today these algorithms are either augmented and/or combined with other techniques to solve more complex problems. Google's DeepMind managed to combine RL and Supervised Learning to train an agent using data sets from expert player games, and then improving on that agent using RL. This solution to GO used several "old" RL algorithms like Monte Carlo Tree Search for the agent's knowledge graph and the REINFORCE algorithm to update RL policy network.

2 Markov Decision Process(MDP)

For a RL problem to be modeled as a MDP it has to satisfy the Markov property. To satisfy the Markov property the agent's transition probability from any state needs to be dependent on that state alone. For example, the probability for going from $state_t$ to $state_{t+1}$ is dependent only on $state_t$, not previous states or previous actions.

Definition of Markov Property

$$Pr[S_{t+1}|S_t] = Pr[S_{t+1}|S_t, S_{t-1}, ..., S_1]$$

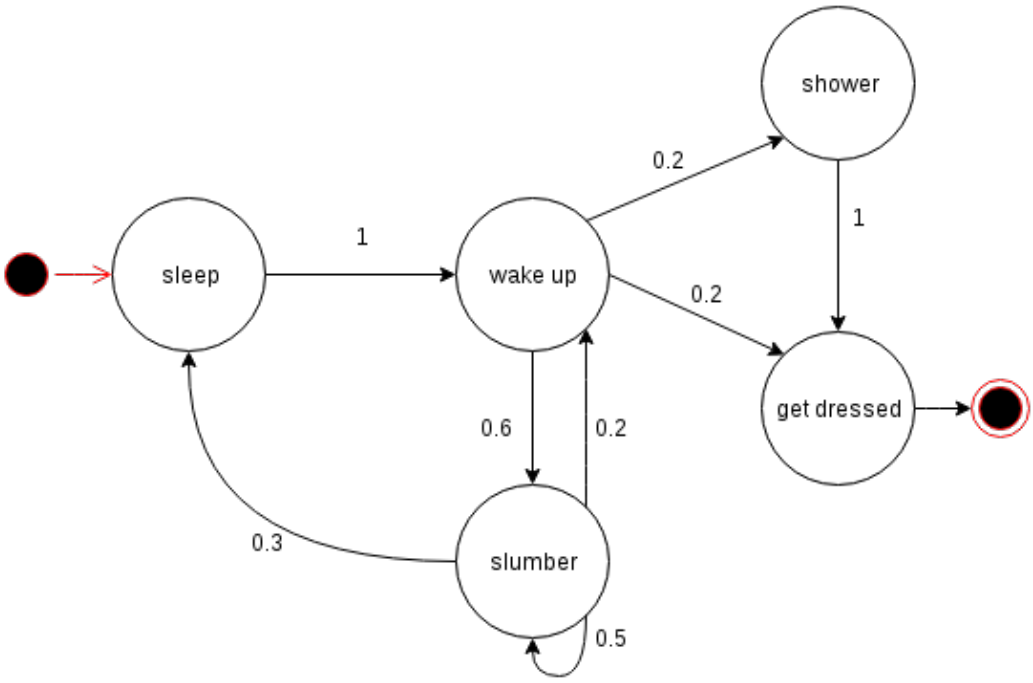

Figure 1: Example of a Finite Markov Process. The circles represents states, and the weighted arrows are transition probabilities. This model states that slumbering has the same probability of happening, regardless of how many times you've slumbered in the past. To get a more accurate model we could add more states, "wake up + t", where t is the amount of times we have woken up. The downside is that we would have to add new "wake up" states until the probability of getting out of the slumber cycle is 1.

The figure above can be represented as a matrix, where each transition is given by $P_{ss'} = Pr[S_{t+1} = s'|S_t = s]$. This matrix contains the probability of ending up in every state from every state.

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0.6 & 0.2 & 0.2 \\ 0.3 & 0.2 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 2: The state transition matrix of figure 1, where the rows are sleep, wake up, slumber,shower and get dressed respectively.

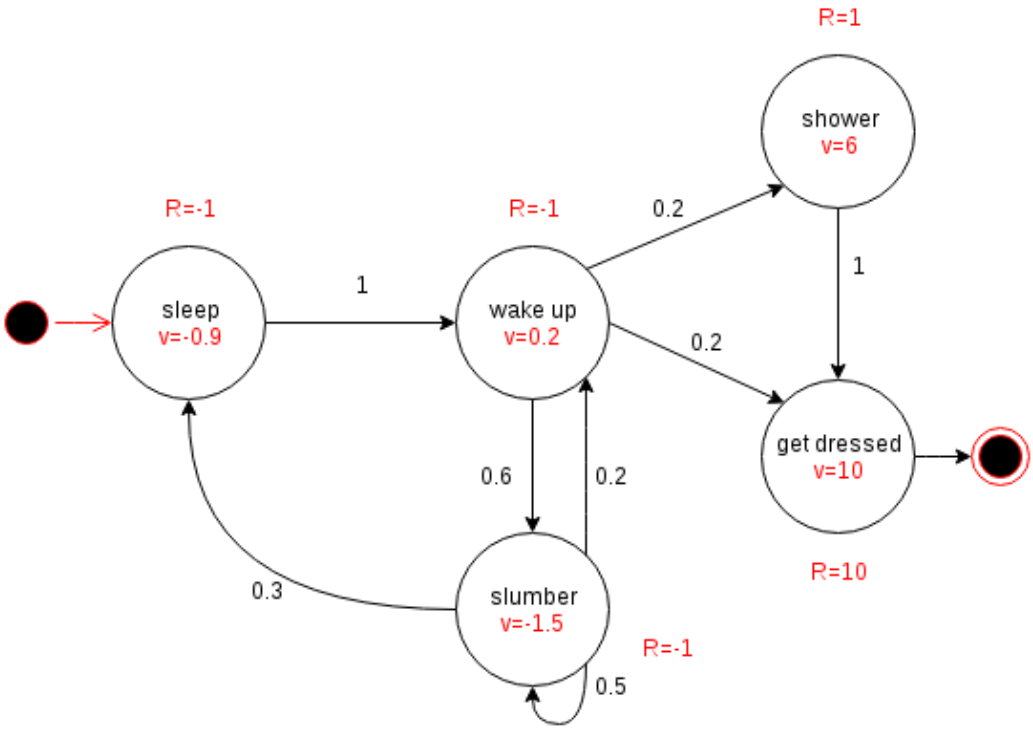

Figure 3: Markov Reward Process. Some units of reward has been added to our model, one unit of reward for showering, 10 for getting dressed and -1 for all other transitions. We have also added the value of each state given $\gamma = 0.5$

Now that we have a basic concept of the Markov Process we can introduce rewards to our model, making it a Markov Reward Model. The goal in RL is to maximize the accumulated reward, called the return of the MDP. Defined below is the discounted return function where $\gamma \in [0, 1]$ is how much we should consider future reward at time t. τ represent the number of timesteps.

Definition of Discounted Return

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{\tau-1} R_\tau$$

When $\gamma = 0$ The agent will only consider the reward it got by ending up in its current state (immediate reward), such agents are called myopic.

When $\gamma = 1$ we get a farsighted agent, the return will not be discounted at all, making the return the sum of all rewards. Notice in figure 3 that we are punished for being short sighted. By showering followed by getting dressed, we would get a return of 11. While by just getting dressed we end up with a return of 10. So, why do we discount the reward? The MDP is stochastic so, the reward given to us far into the future may not be guaranteed. It also help us avoid overflow for long chains.

Definition of Expected Return

$$v(s) = E[G_t|S_t = s] = E[R_{t+1} + \gamma v(s')]$$

The return let us give each state a value that represents how "good" it is to be in a given state. More explicitly it gives us an expected value of the return we can get, being in that state. We calculate this value using the Bellman Expectation Equation with one step look ahead. This can also be done more concisely using matrices.

$$\begin{bmatrix} v(s_1) \\ . \\ . \\ . \\ v(s_n) \end{bmatrix} = \begin{bmatrix} R_1 \\ . \\ . \\ . \\ R_n \end{bmatrix} + \gamma \begin{bmatrix} P_{11} & ... & P_{1n} \\ . & & \\ . & & \\ . & & \\ P_{n1} & ... & P_{nn} \end{bmatrix} \begin{bmatrix} v(s_1) \\ . \\ . \\ . \\ v(s_n) \end{bmatrix}$$

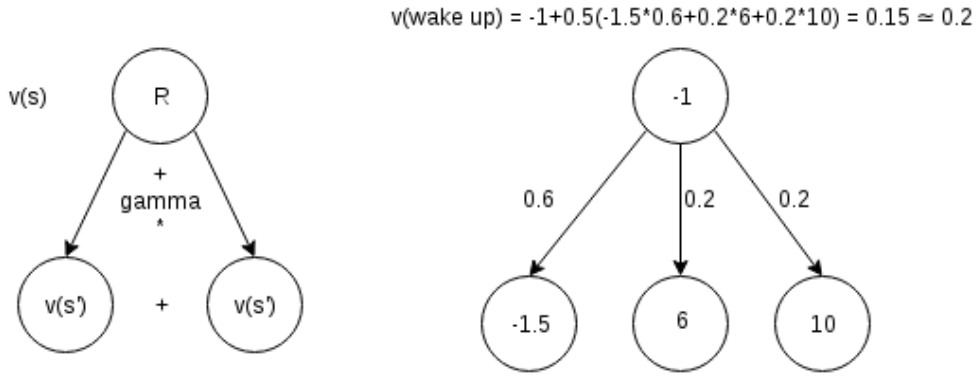

Figure 4: To the left we see a backup diagram, to better understand how the Bellman Expectation work. The top circle represents the state we are to calculate the reward of, and the bottom circles are our current state's successor states, the states which our current state's value directly depend upon, given that $\gamma \neq 0$. To the right we have calculated the value of the wake up state. We move from the bottom up averaging the successor state values over their probability of happening times the discount rate summed with the reward of the state that we are in.

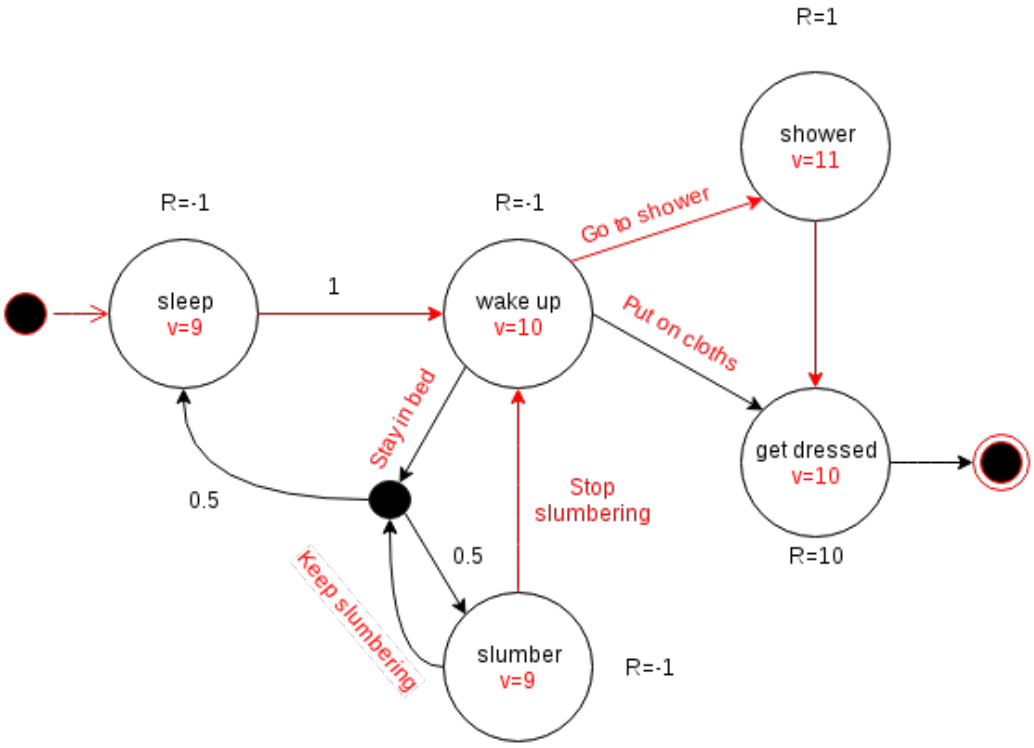

Figure 5: Markov Decision Process. Some of the random probabilities are now replaced with actions. Action arrows that goes directly between two sates has a probability of 1 to end in the state they point to. The "turn off alarm" action on the other hand has a stochastic outcome. The state values are also updated. The current values have a discount factor of $\gamma = 1$ and follows the optimal policy q_* marked by the red arrows.

We now have actions in our model and are no longer passengers unable to affect the flow through the environment. With our new actions we need a policy. The policy is a distribution over actions given a state.

Definition of Policy

$$\pi(s|a) = Pr[A_t = a|S_t = s]$$

The expected return of the MDP is dependent upon our policy, so is the transition probability matrix and the expected return. Our actions let us measure each actions value in any given state for all policy π. Just like state

values, action values gives us expected return by preforming action a in state s. We calculate the action value in the same way that we calculate the state value, using the Bellman Expectation Equation.

Definition of $BellmanExpectationEquation for Q^{\pi}$

$$q_{\pi} = R_s^a + \gamma \sum_{s=S'} P_{ss'}^a v_{\pi}(s')$$

For all MDPs there are at least one optimal policy which is equal or better than all other policies in all states. The optimal policy is denoted q_*.

3 Evaluation methods

3.1 Dynamic programming

In reinforcement learning Dynamic Programming may be used for planning. For Dynamic Programming to be applicable for our problem, our problem need to have optimal substructure. What this means, is that our problem can be broken down into smaller problems. The solutions for these subproblems will contribute to solving the main problem. The main problem is maximizing return from start to finish, and the subproblems are maximizing return from any state. When a subproblem is solved, we cache the solution, so we don't have to recompute it in the future. In RL our value function can serve as this memorization. This requires that we know the MDP for the problem (states, action, transition probabilities, rewards and the discount factor). When we have the MDP we can calculate the optimal value function v_* and in turn find the optimal policy π_*. If we also have a policy we can use DP to predict its return.

3.1.1 Iterative Policy Evaluation

Definition of Iterative Policy Evaluation

$$v_{k+1}(s) = \sum_{a \in A} \pi(a|s)(R_s^a + \gamma \sum_{s' \in S} Pr_{ss'}^a v_k(s'))$$
$$\mathbf{v}_{k+1} = \mathbf{R}^{\pi} + \mathbf{Pr}^{\pi} \mathbf{v}_k$$

In the definition above v_{k+1} is the value of state s at the k+1 iteration, R_s^a is the immediate reward of taking action a in state s, $Pr_{ss'}^a$ is the probability of ending up in s' when taking action a in state s and $v_k(s')$ is the value at a predecessor state of s at out last iteration. For $k = 0$ we can use an arbitrary value function. The values in the MDP example in the last section were computed starting with $v(s) = 0$ and an equiprobable policy for all s. Making these iterative improvements to the value function will eventually converge to the true v_π for enough iterations. This method is also called a full backup, because we compute $v(s)$ using all possible preceding states.

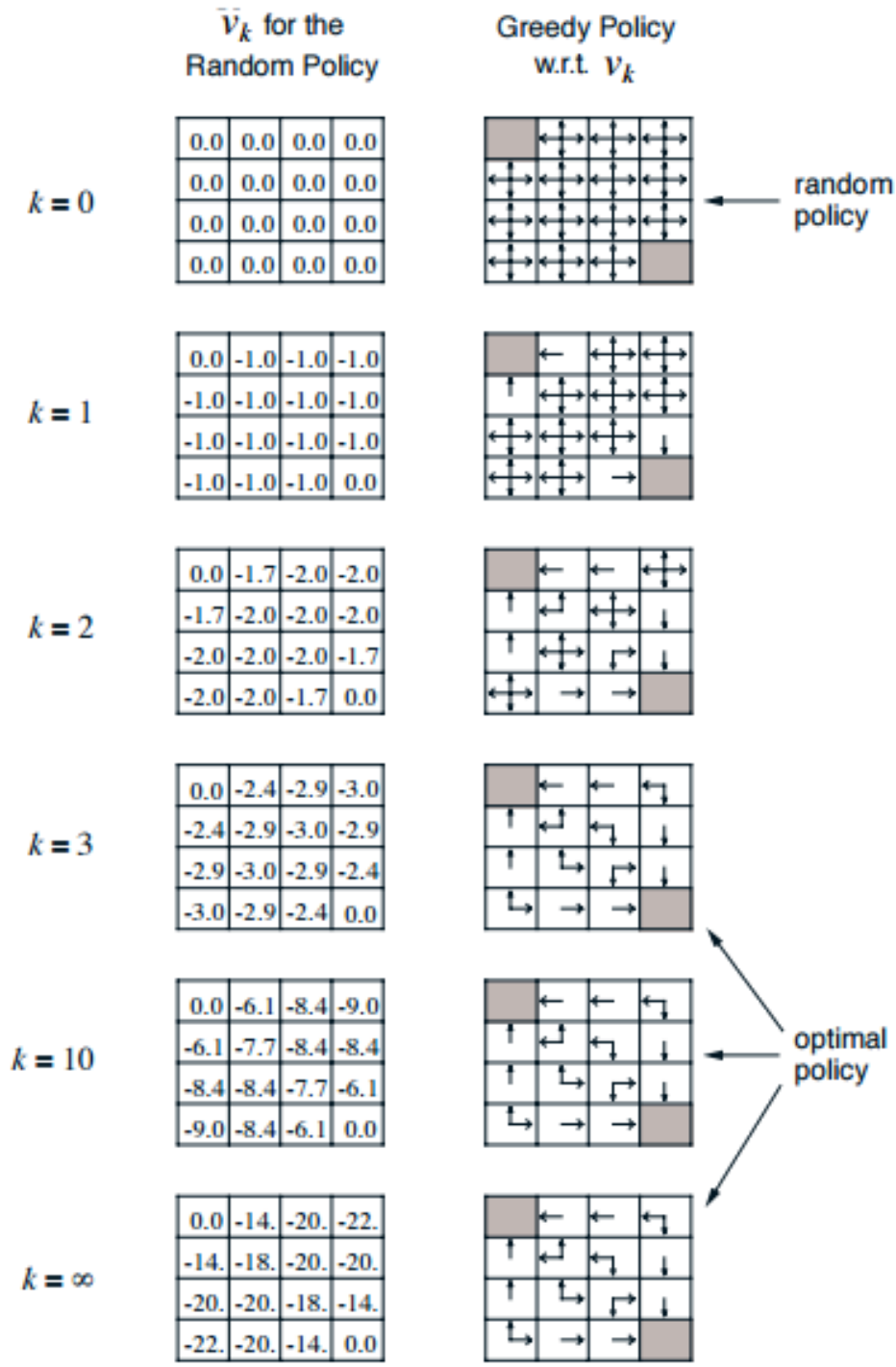

Figure 6: Sutton & Barto 2016. Here we can see the a gridworld example of the iterative policy evaluation for an equiprobable policy. The gray fields in the grid are termination states. All other states has four actions, left, right, up, down. Each state transition has a reward of -1.

3.1.2 Policy Improvement and Iteration

Definition of Improved Policy

$$\pi'(s) = \underset{a}{\operatorname{argmax}}\, q_\pi(s, a)$$

If we have a policy π, we can can generate an equal or better policy π' by acting greedy with respect to v_π. Sutton and Barto proves that the new policy $\pi'$is guaranteed to be better than π in finite MDPs as long as π is not an optimal policy. We now evaluate the new policy and in turn improve it. Continuing this iteration will eventually converge to the optimal policy.

3.1.3 Value Iteration

Definition of Value Iteration

$$v_{k+1} = \max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s')$$

Much like the Policy improvement iteration we iteratively evaluate the value function, but instead of computing the value in respect to some policy, we are only interested in following the best action in any state. Once the rate of change for $v(S)$ slows down we can stop the iteration process.

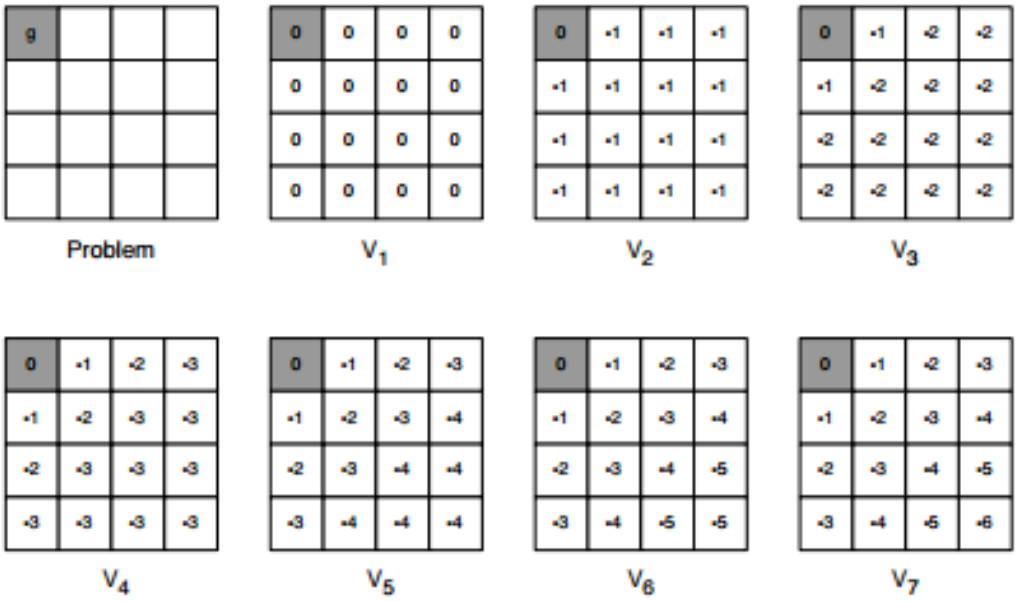

Figure 7: David Silver. Here we can see how the value propagates each iteration outwards from the termination state.

The algorithms described above are synchronous, meaning that all states are updated at each iteration. It does not have to be like that, we can for example given that we have a sample through our MDP, compute the value for the states encountered in the sample, and the states around them.

3.2 Mote Carlo

Unlike Dynamic Programming, Monte Carlo algorithms is model free. You might have noticed that all DP algorithms mentioned in the last section includes the state transition probability and reward. In model free learning we do not need this information and in many RL problems the MDP dynamics are not known to us.

3.2.1 Policy Evaluation

What we use instead in MC algorithms to find the value function of an environment, is an average over episode samples using an arbitrary policy, and use those data to compute our value function. This will converge to

the true value function by the law of large numbers. A downside with these methods, is that the only work properly on episodic problems.

First-Visit Policy Evaluation For every state we encounter we keep a visit counter $N(s)$ and a accumulated return $S(s)$. As the name suggests, every time we arrive at a new state for the first time in an episode, we increment $N(s) = N(s) + 1$ and $S(s) = S(s) + G_t$. The value function is then $V(s) = \frac{S(s)}{N(s)}$.

Every-Visit Policy Evaluation Just like in the First-Visit algorithm, but instead of just incrementing accumulated sum and visit counter the first time we visit a state per episode, we do it at every time step.

Definition of Incremental Monte Carlo Update

$$V(s_t) = V(s_t) + \frac{1}{N(s_t)}(G_t - V(s_t))$$

3.3 Temporal Difference

Temporal Difference is a combination of Monte Carlo and Dynamic Programming. Like MC it uses a model free approach through sampling. Like DP it uses the known value functions to estimate future values with out an episode having to terminate. This means that we can use it on-line on a single play, not just on a sample after an entire episode.

Definition of TD(0)

$$V(s_t) = V(s_t) + \alpha(R_{t+1} + \gamma V(s_{t+1}) - V(s_t))$$

After a play we back up the transition, updating the value for previous state. $R_{t+1} + \gamma + V(s_{t+1})$ is called the TD target. R_{t+1} is the actual received reward from play t, and $V(s_{t+1})$ is our estimated value of the state we ended up in. $R_{t+1} + \gamma + V(s_{t+1}) - V(s_t)$ is called the TD error. It is the difference between the actual value of s_t and our predicted value of s_t. So we update our estimate of Vs_t in the direction of the error times a learning factor α. In TD(0) we back up the single plays only. In TDλ) we back up λ) plays where λ in an integer. Where future TD targets are discounted to limit potential noise.

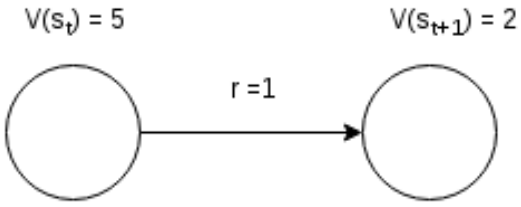

Figure 8: This is an example of a TD(0) backup. We use $\alpha, \gamma = 1$. $V(s_t) = 5 + 1 * (1 + 1 * 2) - 5 = 3$. Our new approximation for $V(s_t)$ is now 3.

4 Model Free Control

4.1 Generalized Policy Iteration(GPI)

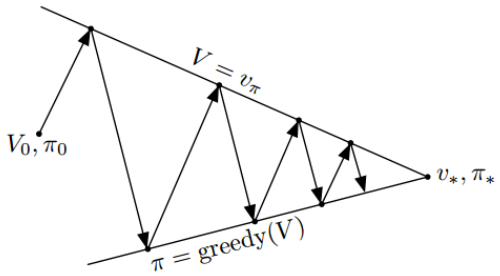

Figure 9: Sutton & Barto. A graphical representation of GPI. We start off with a value function and a policy. We improve our value function by evaluating our policy and in turn generate a new policy by acting greedy with respect to our value function. In other words, we first find the true v_π. When we have v_π, we know if there are sub optimal actions taken in our policy. We then switch the sub optimal actions with the optimal once. This is our new policy. We continue this process until our policy stops improving.

We can apply the Value Iteration and Policy Improvement algorithm we used to improve our policy by Dynamic Programming. This idea, where we evaluate a policy and update the policy in respect to the new evaluation, is known as Generalized Policy Iteration. For model free problems, instead of evaluating the policy using DP(as the require the MDP), we can use MC

or TD. To be able to use GPI model free, we can no longer use the state values. Generating a policy based on state values, we will need to know the values of a state's successor states, and without the model we don't know the successor states. We instead use action values.

4.2 Monte Carlo

4.2.1 On-Policy Methods

On-Policy methods are when the agent learn directly from episodic sample data it generates it self.

Monte Carlo with Exploring Start(ES) This is the simplest MC algorithm. This algorithm requires an exploring start. An exploring start requires that all state-action pairs have a probability of being the initial state-action pair in an episode. Guaranteeing that all state-action values reach their true value by the law of large numbers, as number of episodes sampled approaches infinity. Each iteration we pick a state to start in and an action to take from that start state. After this we complete the episode following our current policy. When we have the sample, we store the return for each state-action pair for the first time it occurred in the episode. The Value function is updated by averaging the return of all state-action pairs. Finally we generate a new policy by acting greedy in respect to our value function. This is repeated until the change in value approaches zero, at which point the policy has also stopped changing. The algorithm has now converged to an optimal policy.

Monte Carlo ES (Exploring Starts)

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \leftarrow$ arbitrary
    $\pi(s) \leftarrow$ arbitrary
    $Returns(s,a) \leftarrow$ empty list

Repeat forever:
    Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ s.t. all pairs have probability > 0
    Generate an episode starting from S_0, A_0, following π
    For each pair s, a appearing in the episode:
        $G \leftarrow$ return following the first occurrence of s, a
        Append G to $Returns(s,a)$
        $Q(s,a) \leftarrow$ average$(Returns(s,a))$
    For each s in the episode:
        $\pi(s) \leftarrow \arg\max_a Q(s,a)$

Figure 10: Sutton & Barto. Monte Carlo Exploring Start as explained above.

Monte Carlo without Exploring Start As expected, Exploring Start is not applicable to most RL problems. An alternative solution to this, is to have some stochasticity to each policy causing all state-action pairs to be sampled from. This is where we introduce a new hyperparameter commonly used in RL, ϵ. ϵ is a real number between zero and one. Where $\epsilon = 0$ has a zero probability of picking a random action, and $\epsilon = 1$ is an equiprobable policy. In other words, ϵ is the rate at which the agent selects a random action in stead of the greedy one. While this randomness helps us explore the whole state-action space, so our agent does not get stuck in a local maximum. It is also a problem, seeing as there is always at least one best action to take. So as our algorithm converges to an optimal policy, our agent will still pick randomly with a probability of ϵ. At a point near convergence this sate-action selection will always be as good as our policy or worse. In stationary problems this can easily be fixed by reducing ϵ as we reach convergence.

4.2.2 Off-Policy Methods

In contrast with on-policy methods these are based on the agent learning by using episode data generated by other sources than our agent, e.g. other agents or humans. A common strategy is to have one on-policy which always acts greedy in respect to its value function. And an off-policy which has some

randomness, allowing all state-action pairs to be visited. The samples form the off-policy are used to improve the value function of the on-policy.

4.3 Temporal Difference

With TD we no longer have to wait for the sampling before we update the value function. We can do it both online while we are sampling an episode, or on an off-policy sample. TD has proven to converge to an optimal policy faster do to its bootstrapping. The value function is based on preceding values. Where as in MC, an actions value is based solely an average over the actual returns from that state-value pair.

4.3.1 SARSA: On-Policy TD Control

SARSA is given its name by its update rule $Q(s,a) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A))$. An state-action pairs value $Q(S,A)$ is updated using the reward R and the preceding state-action pair value $Q(S',A')$.

Sarsa: An on-policy TD control algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figure 11: Sutton & Barto. A SARSA algorithm. This algorithm will converge to an optimal policy as states are sampled an infinite number of times.

4.3.2 Q-learning: Off-Policy TD Control

Q-learning: An off-policy TD control algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 12: Sutton & Barto. The Q-learning algorithm. It's much like the SARSA algorithm, but instead of the value function being updated using a policy, we update our value assuming that we are going to select the action in the preceding state that we think will grants us most return.

5 Value Function Approximators

Previously in this paper, we have for the dynamic programming part assumed that we have a model for the entire state space, and for the model free solutions assumed that we store all our state values in a lookup table. While we still need a model to do dynamic programming, we can substitute the value table with a value function approximator, we can use any function approximator we want as long as it can interpret the state signal. There are two large benefits of using neural nets as a value approximator. The first being the agents memory consumption. With table lookup the memory cost of an agent is proportionate with the state space, and if the state space is continuous then the memory of the agent would have to be infinite. With neural nets however, the memory usage of the agent will be limited to the size of the function approximator's parameter space. The other benefit is generalization. For example, lets say we have a problem: getting to work, and the current day is a part of the state signal. It is Monday and you are driving to work. You drop your hot coffee in your lap. The day after you are once again driving to work. For an agent using table look up these two episodes use two disjunct subsets of the state space, so it does not have any

value for poring coffee in its lap on a Tuesday. An agent that uses a neural net value approximator will see the correlation between the coffee in the lap and the high negative reward. This again has two benefits. It reduces the amount of training data we need, and when we visit a previously unvisited state. When using table lookup we will have to set an arbitrary default value to unseen states, but with a neural net we can give it the state signal of a new state, and it will return the value of states with similar features. Fist of all we need a feature vector that we can feed into the value approximator. This is usually our state signal. These features tells us about the state that we are in e.g. the velocity of a train, the weather or the day of the week as used in the example above. All these environment properties are features, and all these features may affect the value of a state in varying degree. This is how we get generalization. In comparison, an agent using table lookup will be almost the same as using the the feature vector as a key for a dictionary. We call the feature vector s. So now that we have our features, we need an approximator function. We call this $\hat{v}(s, w)$ where w is our weights. At first, with randomized initiation weights, these approximations will be wrong. We will need a cost function to tell us exactly how wrong our estimates are we use MSE, $J = (v(s) - \hat{v}(s, w))^2$. Which gives us the weight update rule $w = w + \alpha(v(s) - \hat{v}(s, w))\nabla_w \hat{v}(s, w)$. Where α is the step size(how much we are going adjust the weights).

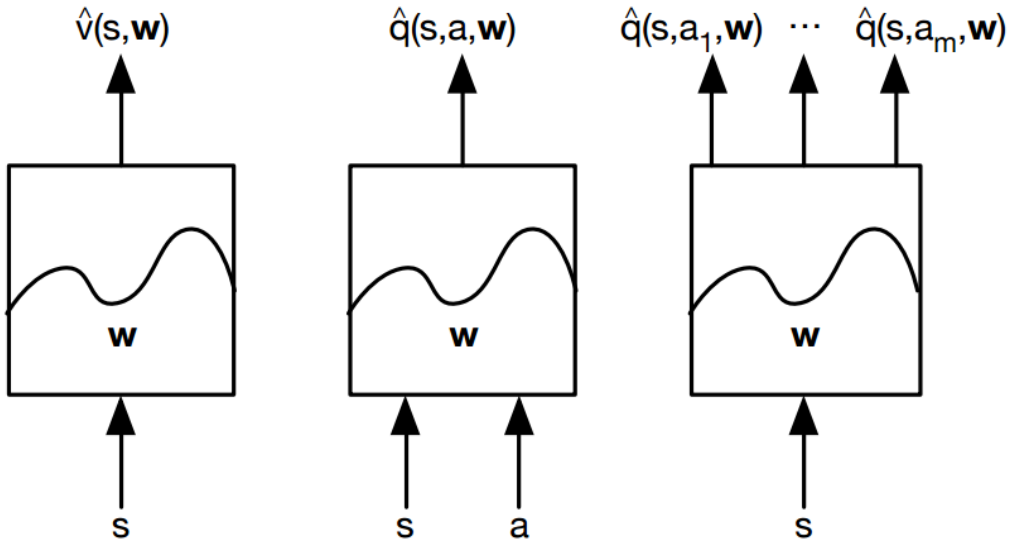

Figure 13: Silver. Three different architectures. The left figure uses a feature vector consisting only of the state signal and returns the states value. The middle model uses state signal and action number as feature vector, getting the action value for the state-action pair. The right figure uses just the state signal and gives the state-action value for all actions in state s.

5.1 Incremental Methods

What is meant with incremental methods, is that we update our value approximator incrementally as we sample our environment. In the previous gradient decent example we assumed that we have the true value function $v(s)$ which we tried to fit. In RL we don't usually have this supervising function. If we did, there would be no need to solve the problem as $v(s)$ is the solution.

5.1.1 Monte Carlo

In Monte Carlo instead of using $v(s)$ as the target, we use the return of our sample G. By taking samples we can generate a data set to train our approximator in a supervised fashion, e.g. (S_t, G_t)

Definition of Monte Carlo weight update

$$w = w + \alpha(G_t - \hat{v}(s_t, w))\nabla_w \hat{v}(s_t, w)$$

5.1.2 Temporal Difference

In TD learning we use the TD-target instead of $v(s)$. To refresh the TD-target is the immediate return of a play r plus what we think the value of the state we ended up in is. Just like with the Monte Carlo solution we can generate data sets from our episode samples $(s_t, r_{t+1} + \gamma\hat{v}(s_{t+1}, w))$.

Definition of Temporal Difference weight update

$$w = w + \alpha((r + \gamma\hat{v}(s', w) - \hat{v}(s_t, w))\nabla_w \hat{v}(s_t, w)$$

6 Neural Networks

6.1 Traditional ANNs

Artificial Neural Network (ANN) is a model for computation loosely based on how the biological brain works. The biological brain has approximately 100 billion interconnected neurons. These neurons sends signals through axons to other neurons throughout the brain. Approximately a neuron is on average connected to 10000 other neurons [9]. In computer science the neuron is represented as an entity with a set of input weights and an output signal. The neuron in fed a signal(s) from either an input source from outside the network, or other neurons. The neuron has a weight for each input signal. The input signals are multiplied by their respective weights. These weighted inputs are summed together with the neuron's bias, if it has one. What happens inside the neuron depends on the networks implementation. Some of the different neurons used are:

- The sigmoidal neuron is maybe the most commonly used neuron used because of its nonlinear properties, allowing complex input/output mapping. It passes the weighted input through a sigmoid function which is the output of the neuron e.g. a real number between 0 and 1.
- Binary neurons outputs a binary value based on weighted input e.g. outputs 1 if the weighted input exceeds a threshold value and 0 otherwise.

- Binary stochastic neuron is much like the binary neuron but instead of outputting 1 when a threshold is exceeded it uses the weighted input as the probability of outputting 1.

A network may have a uniform or combination of several different neurons (Williams [8] suggests using a stochastic output layer in a deterministic network to promote exploration in reinforcement problems). When a network is trained, it can take input data it has never seen before and generalize it to previously learned features.

6.1.1 ANN's Applications

ANNs greatest strength is its applicability. It has been used successfully in supervised, unsupervised and reinforcement learning. We use ANNs in our everyday lives, maybe without even noticing. They are used in customer recommendations, spam filters, image recognition, voice recognition, ecommerce, weather estimations, self driving cars, etc. In Reinforcement Learning ANNs can be used as value approximators, instead of keeping an average value for all actions in all states.

6.1.2 ANN Architectures

The feedforward ANN is an architecture where input is passed forward through the layers in the network without cycles (in contrast with Recurrent Neural Networks). The vector of input variables is considered the input layer, the last layer is the output layer. Every single layer in between is called a hidden layer. They are considered "hidden" because their activity is hidden from the outside function. ANNs can range from just input output layers to an arbitrary amount of hidden layers. ANNs with more than one hidden layer are termed Deep Neural Networks. As ANNs grow in depth their feature detections also increases [4]. The tradeoff is that they take much more training data to make good approximations [7].

6.1.3 Network Training

Essentially an ANN can be considered a function of a set of variables [4]. The purpose of training is to make an estimation based on input values. E.g. in supervised learning, the network is trained using a training set where each entry contains input and the desired or correct output. The input is fed

through the net and an error is calculated i.e. the difference between actual output and desired output. Then the error is propagated backwards through the layers in the net to find the error in respect to each individual weight, and then update the weights accordingly. This is called the backpropagation algorithm.

Backpropagation. Without any training the input/output mapping of a Neural Network is as good as random. However, we want a specific output given the input. To do this we need a way to measure the error of the input/output mapping. This is done (in supervised learning using BP) by comparing the network's output with the desired output, provided in the training data using a cost function. An example of a cost function is the quadratic cost aka squared error function: $C = \frac{1}{2}\sum_i(\hat{y}_i - y_i)^2$, where i is the number of neurons in the output layer, $\hat{y}$ is the network's output and y is the actual output. When we train a network, we want the error to be as small as possible (without overfitting). This is done by finding the cost function's rate of change with respect to each weight, then moving the weight in the opposite direction. For this to work, there need to exist a first order derivative of the cost function. By dividing the training set into mini-batches, we can do BP iteratively on the mean error of a mini-batch. While following the gradient of a single training sample, will reduce the loss of that input, following the mean gradient of a set will be more stable towards a collective minimum. Downsides of first order BP, is that it tends to get stuck when the rate of change is zero. This can happen in states of optima, local minima, or saddle points.

6.2 Recurrent Neural Nets

RNNs are a lot like feed forward nets, in the sense that they are trying to map some input x to output y. RNNs may map a sequence of x to a sequence of y. While feed forward nets treats every input as a separate case e.g. approximating a probability distribution or captioning an image, RNNs use context of previous sequential input to determine the current input. In the case of captioning a set of images where the images makes up a short film. While the feed forward network would process the images separately, captioning only the content of the image. The RNN can process the images as a temporal sequence, captioning events happening over a period of time.

RNNs excel at tasks where the problem breaks into sequences, like language translation, where the next word in a translated paragraph may depend on previous and future sequences of words.

6.2.1 Architecture

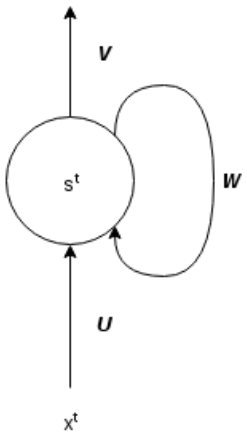

Figure 14: The architecture of a simple Recurrent Neural Net

In figure 14 we see a diagram of a simple RNN where x_t is the network's input at timestep t. U is the weight matrix from the input layer to the hidden layer. W is the weight matrix from the hidden layer to itself. V is the weight matrix from hidden to output layer. st is the hidden state at time t, this can be thought of as the network's memory. All weight matrices are shared across all time steps, allowing sequence length to be theoretically arbitrarily long. The output of s at time t is $f(Ux_t + Ws_{t-1})$, where f is the activation function. We can also turn the recursion direction by making $f(Ux_t + Ws_{t+1})$. If we combine both directions making $V(f(Ux_t + Ws_{t-1}) + f(Ux_t + Ws_{t+1}))$ we get a bidirectional RNN. A problem with too long sequences is the vanishing gradient problem. More complex RNNs like the Long Short Term Memory and Gated Recurrent circumvents this[3].

7 Conclusion

RL can be used to solve complex problems, and improve on already well established control solutions as proved by Gerald Tesauro[7]. We can use

Neural Nets to improve the effectiveness of RL algorithms substantially. Machine Learning algorithms other than Neural Networks can also be used. Because of RL brute forcing nature, its effectiveness is held back by lack of computation power, and the ability to generalize sensory input. RL is an important filed in computer science. Definitively worth more research.

8 Citations

1. Richard S. Sutton and Andrew G. Barto 2016, Reinforcement Learning: An Introduction (Second edition, in progress), https://webdocs.cs.ualberta.ca/ sutton/book/b

2. David Silver 2015, RL Course: Lecture 2: Markov Decision Process, https://www.youtube.com/watch?v=lfHX2hHRMVQ

3. Ian Goodfellow, Yoshua Bengio, Aaron Courville 2016, Deep Learning, http://www.deeplearningbook.org/

4. Michael Nielsen 2016, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com/

5. Stephen Welch 2014, Neural Networks Demystified [Part 4: Backpropagation], https://www.youtube.com/watch?v=GlcnxUlrtek

6. Denny Britz 2015, Introduction to RNNs, http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial -part-1-introduction-to-rnns/

7. Gerald Tesauro 1995, Temporal Difference Learning and TD-Gammon http://www.bkgm.com/articles/tesauro/tdl.html

8. RJ Williams 1992, Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning.

9. Tom Mitchell 1997, Machine Learning