Reinforcement Learning **Spring Semester, 2018/9**

Lecture 9: May 15, 2019

Lecturer: Yishay Mansour *Scribe: ym*

DISCLAIMER: Based on `Learning and Planning in Dynamical Systems` by Shie Mannor©, all rights reserved.

This lecture continue looking at the case where the MDP models are large. In the previous lecture we looked at approximating the value function. In this lecture we will consider learning directly a policy and optimizing it.

The main advantages and disadvantages of policy optimization (rather than value function approximation) are the following:

1. Continuous action space. In such a case the policy optimization method is the most natural and the one which is used in practice (for applications such as robotics).

2. Convergence: While the policy optimization would normally converge, it will most likely converge to a local optimum.

3. High dimensional spaces: It can be fairly effective in selecting the actions (in contrast to learning accurate values).

4. Evaluation time would typically be longer (compared to value function approaches).

5. Stochastic Policy: allows naturally to have a stochastic policy.

9.1 Stochastic policies

We have seen that the optimal policy in an MDP is deterministic, so what benefit can be in considering a stochastic policy?

The main benefit is in situations where there is a miss-specification of the model. The main issue is that the state encoding might create a system which is not Markovian anymore, by coalescing certain states, which have identical encoding. We will give two example of this phenomena.

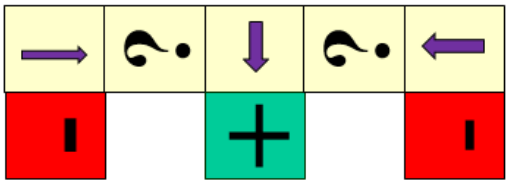

Figure 9.1: Grid-world example

Aliased Grid-world

Consider the example in Figure 9.1. The green state is the good goal and the red ones are the bad. The encoding of each state is the location of the walls. In each state we need to choose a direction. The problem is that we have two states which are indistinguishable (marked by question mark).

It is not hard to see that any deterministic policy would fail from some start state (either the left or the right one). Alternatively, we can use a randomized policy in those states,with probability half go right and probability half go left. For such a policy we have a rather short time to reach the green goal state (and avoid the red states).

The issue here was that two different states had the same encoding, and thus violated the Markovian assumption.

Zero-sum games

The MDP model was not design for interactive zero-sum games, however, in many of the application we saw how to learn a policy to play a board game (such as backgammon). The main issue with a more general game, is the the optimal policy might be stochastic.

Consider a penny-matching game, which each player select a bit $\{0, 1\}$. If the two selected bits are identical the first player wins and if they differ the second player wins. The best policy for each player is stochastic (selecting each bit with probability half).

An important observation is that if one of the player play deterministically (or almost deterministically), then the other player can win (or almost always win). For this reason, any ϵ-greedy would have a poor performance.

Here we violated the assumption that the rewards depend only on the state. In this example they depend indirectly on the policy selected.

9.2 Policy optimization

We will assume that each state s has an encoding $\phi(s) \in \mathbb{R}^{d_1}$. The policy will have a parametrization $\theta \in \mathbb{R}^{d_2}$. The policy will be based on the two encodings, and we have $\pi(a|s,\theta)$, which is the probability of selecting action a when observing state s (or, equivalently, $\phi(s)$), and having a policy parametrization θ.

The optimization problem would be:

$$\theta^* = \arg\max_{\theta} J(\theta)$$

where $J(\theta)$ is the expected return of the policy $\pi(\cdot|\cdot,\theta)$.

This maximization problem can be solved in multiple ways. We will mainly look at gradient based methods.

We start be giving a few example on how to parameterize the policy. The first is a *log linear policy*. We will assume an encoding of the state and action pairs, i.e., $\phi(s,a)$. The linear part will compute $\mu(s,a) = \phi(s,a)^\top\theta$. Given the values of $\mu(s,a)$ for each $a \in A$, the policy select action a with probability proportional to $e^{\mu(s,a)}$. Namely,

$$\pi(a|s,\theta) = \frac{e^{\mu(s,a)}}{\sum_{b\in A} e^{\mu(s,b)}}$$

The second example is a *Gaussian policy*, where the action space is a real number, i.e., $A = \mathbb{R}$. The encoding is of states, and the actions are any real number. Given a state s we compute $\mu(s) = \phi(s)^\top\theta$. We select an action a from the normal distribution with mean $\mu(s)$ and variance σ^2, i.e., $N(\mu(s),\sigma^2)$. (The policy has an additional parameter σ.)

We would like to use the policy gradient to optimize the expected return of the policy. Let $J(\theta)$ be the expected return of the policy $\pi(\cdot|\cdot,\theta)$. We will compute the gradient of $J(\theta)$, i.e., $\nabla_\theta J(\theta)$. The update of the policy parameter is

$$\theta_{t+1} = \theta_t + \alpha\nabla_{\theta_t} J(\theta_t)$$

One challenge we will have to address is to relate $\nabla_\theta J(\theta)$ to the gradients $\nabla\pi(a|s,\theta)$.

9.2.1 Finite differences methods

This methods can be used even when we do not have a representation of the gradient of the policy. This can be done by introducing perturbations.

The simplest case is component-wise gradient estimates. Let e_i be a unit vector, i.e., has in the i-entry a value 1 and a value 0 in all the other entries. The perturbation that we will add is δe_i for some $\delta > 0$. We will use the following approximation:

$$\frac{\partial}{\partial \theta_i} J(\theta) \approx \frac{\hat{J}(\theta + \delta e_i) - \hat{J}(\theta)}{\delta}$$

where $\hat{J}(\theta)$ is unbiased estimator of $J(\theta)$. A more symmetric approximation is sometimes better,

$$\frac{\partial}{\partial \theta_i} J(\theta) \approx \frac{\hat{J}(\theta + \delta e_i) - \hat{J}(\theta - \delta e_i)}{2\delta}$$

The problem is that we need to average many samples of $\hat{J}(\theta \pm \delta e_i)$ to overcome the noise. Another weakness is that we need to do the computation per dimension. In addition, the selection of δ is also critical. A small δ might have a large noise rate that we need to overcome (by using many samples). A large δ run the risk of facing the non-linearity of J.

Rather then doing the computation per dimension, we can do a more global approach and use a least squares estimation of the gradient. Consider a random vector u_i, then we have

$$J(\theta + \delta u_i) \approx J(\theta) + \delta (u_i)^\top \nabla J(\theta)$$

We can define the following least square problem,

$$G = \arg\min_x \sum_i (J(\theta + \delta u_i) - J(\theta) - \delta (u_i)^\top x)^2,$$

where G is our estimate for $\nabla J(\theta)$.

We can move to matrix notation and define $\Delta J^{(i)} = J(\theta + \delta u_i) - J(\theta)$ and $\Delta J = [\cdots, \Delta J^{(i)}, \cdots]^\top$. We define $\Delta \theta^{(i)} = \delta u_i$, and the matrix $[\Delta \Theta] = [\cdots \Delta \theta^{(i)}, \cdots]^\top$, where the i-th row is $\Delta \theta^{(i)}$.

We would like to solve for the gradient, i.e,

$$\Delta J \approx [\Delta \Theta] x$$

This is a standard least square problem and the solution is

$$G = ([\Delta \Theta]^\top [\Delta \Theta])^{-1} [\Delta \Theta]^\top \Delta J$$

One issue that we neglected is that we actually do not a have the value of $J(\theta)$. The solution is to solve also for the value of $J(\theta)$.

We can define a matrix $M = [1, [\Delta \Theta]]$ vector of unknowns $x = [J(\theta), \nabla J(\theta)]$ and have the target be $z = [\cdots, J(\theta + \delta u_i), \cdots]$. We can solve for $z \approx Mx$.

9.3 Policy Gradient Theorem

The policy gradient theorem will relate the gradient of the expected return $\nabla J(\theta)$ and the gradients of the policy $\nabla \pi(a|s, \theta)$.

We will state the policy gradient theorem for the episodic return, but it also holds for the discounted return and average reward return. Recall that the episodic return is $\sum_{t=1}^{T} r_t$ and $V^{\pi}(s) = E[\sum_{t=1}^{T} r_t | s_1 = s]$.

Theorem 9.1 (Policy Gradient Theorem)**.** *For any policy $\pi(\cdot|\cdot;\theta)$, we have*

$$\nabla J(\theta) \propto \sum_{s \in S} \mu(s) \sum_{a \in A} Q^{\pi}(s,a) \nabla_{\theta} \pi(a|s;\theta)$$

where $\mu(s) = \nu(s)/T$ and $\nu(s) = \sum_{t=1}^{T} \Pr[s_t = s | s_1, \theta]$.

Proof. For each state s we have

$$\begin{aligned}
\nabla V^{\pi}(s) &= \nabla \sum_{a} \pi(a|s) Q^{\pi}(s,a) \\
&= \sum_{a} \nabla \pi(a|s) Q^{\pi}(s,a) + \pi(a|s) \nabla Q^{\pi}(s,a) \\
&= \sum_{a} \nabla \pi(a|s) Q^{\pi}(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) \nabla V^{\pi}(s') \\
&= \sum_{x \in S} \sum_{t=1}^{T} \Pr[s_t = x | s_1 = s, \pi] \sum_{a} \nabla \pi(a|x) Q^{\pi}(x,a)
\end{aligned}$$

where the first identity follows since by averaging $Q^{\pi}(s,a)$ over the actions a with the probabilities induce by $\pi(a|s)$ we have right expectation of the immediate reward. The next state is distributed correctly, and therefore the identity holds. The second equality follows from the gradient of a multiplication. The third follows since $\nabla Q^{\pi}(s,a) = \nabla[r(s,a) + \gamma \sum_{s'} p(s'|s,a) V^{\pi}(s'|s,a)]$. The last identity follows from unrolling s' to s'', etc. and then reorganizing the terms.

Using this we have

$$\begin{aligned}\nabla J(\theta) &= \nabla V^{\pi}(s_1) \\ &= \sum_s \left(\sum_{t=1}^{T} \Pr[s_t = s | s_1, \pi] \right) \sum_a \nabla \pi(a|s) Q^{\pi}(s,a) \\ &= \sum_s \nu(s) \sum_a \nabla \pi(a|s) Q^{\pi}(s,a) \\ &= T \sum_s \mu(s) \sum_a \nabla \pi(a|s) Q^{\pi}(s,a)\end{aligned}$$

□

The Policy Gradient Theorem gives us a way to compute the gradient. We can sample states from the distribution $\mu(s)$ using the policy π. We still need to resolve the sampling of the state.

We can use the following simple identity,

$$\nabla f(x) = f(x) \frac{\nabla f(x)}{f(x)} = f(x) \nabla \log f(x)$$

This implies that we can restate the Policy Gradient Theorem as,

$$\nabla J(\theta) \propto \sum_{s \in S} \mu(s) \sum_{a \in A} \pi(a|s;\theta) Q^{\pi}(s,a) \nabla_\theta \log \pi(a|s;\theta) = E^{\pi}[Q^{\pi}(s,a) \nabla_\theta \log \pi(a|s;\theta)]$$

We can now see what does the theorem says about some simple policy class. For the log-linear policy we have $\log \pi(a|s) \propto \phi(s,a)^{\top} \theta$, and then

$$\nabla \log \pi(a|s) = \phi(s,a) - \sum_b \pi(b|s;\theta) \phi(s,b)$$

and the update is $\Delta\theta \propto \alpha U \phi(s,a)$ where $E[U] = Q^{\pi}(s,a)$.

For Gaussian we have $\nabla \log \pi(a|s;\theta) \propto (a - \mu(s))\phi(s)/\sigma^2$ and update is $\Delta\theta \propto \alpha U (a - \mu(s))\phi(s)/\sigma^2$ where $E[U] = Q^{\pi}(s,a)$.

9.4 REINFORCE: Monte-Carlo updates

The REINFORCE algorithm uses a Monte-Carlo updates. Given a episode $(s_1, a_1, r_1, \ldots, s_T, a_T, r_T)$ for each $t \in [1, T]$ updates,

$$\theta \leftarrow \theta + \alpha R_{t:T} \nabla \log \pi(a_t|s_t;\theta)$$

where $R_{t:T} = \sum_{i=t}^{T} r_i$.

We can extend the REINFORCE to add a baseline function. The baseline function $b(s)$ can depend in an arbitrary way on the state, but does not depend on the action. The main observation would be that we can add or subtract any such function from our estimate U, and it will still be unbiased. This follows since

$$\sum_a b(s)\nabla\pi(a|s;\theta) = b(s)\nabla\sum_a \pi(a|s;\theta) = b(s)\nabla 1 = 0$$

Given this, we can restate the Policy Gradient Theorem as,

$$\nabla J(\theta) \propto \sum_{s\in S}\mu(s)\sum_{a\in A}(Q^\pi(s,a) - b(s))\nabla_\theta\pi(a|s;\theta)$$

This gives us a degree of freedom to select $b(s)$. Note that by setting $b(s) = 0$ we get the old theorem. In many cases it is reasonable to use for $b(s)$ the value of the state, i.e., $b(s) = V^\pi(s)$. The motivation for this is to reduce the variance of the estimator. If we assume that the magnitude f gradients $\|\nabla\pi(a|s;\theta)\|$ is similar for all action $a \in A$, we are left with $E^\pi[(Q^\pi(s,a) - b(s))^2]$ which is minimized by $b(s) = E^\pi[Q^\pi(s,a)] = V^\pi(s)$.

We are left with the challenge of approximating $V^\pi(s)$. On the one hand this is part of the learning. On the other hand we have developed tools to address this in the previous lecture on value function approximation. We can use $V^\pi(s) \approx V(s;w) = b(s)$. The good news is that any $b(s)$ will keep the estimator unbiased, so we do not depend on $V(s;w)$ to be unbiased.

We can now describe the REINFORCE algorithm with baseline function. We will use a Monte-Carlo sampling to estimate $V^\pi(s)$ and this will define our function $b(s)$. We will update using $U - b(s)$ where $E[U] = Q^\pi(s,a)$. More specifically, our algorithm will have a class of value approximation function $V(\cdot;w)$ and a parameterized policy $\pi(\cdot|\cdot;\theta)$, and in addition to step size parameters, $\alpha, \beta > 0$. Given an episode $(s_1, a_1, r_1, \ldots, s_T, a_T, r_T)$ for each $t \in [1,T]$ we compute the $R_{t:T}$ during the times $[t,T]$, i.e., $R_{t:T} = \sum_{i=t}^{T} r_i$. The error in time t is $\Gamma_t = R_{t:T} - V(s_t;w)$. The updates are

$$\Delta w = \alpha\Gamma_t\nabla V(s_t;w)$$

and

$$\Delta\theta = \beta\Gamma_t\nabla\log\pi(a_t|s_t;\theta)$$

We can extend this to handle also TD updates. We will use an actor-critic algorithm. We will use a Q-value updates for this (but can be done similarly with V-values).

The critic maintains an approximate Q function $Q(s, a; w)$. For each time t it defines the TD error to be $\Gamma_t = r_t + Q(s_{t+1}, a_{t+1}; w) - Q(s_t, a_t; w)$. The update will be $\Delta w = \alpha \Gamma \nabla Q(s_t, a_t; w)$. The critic send the actor the TD error Γ.

The actor maintains a policy π which is parameterized by θ. Given a TD error Γ it updates $\Delta\theta = \beta\Gamma\nabla \log \pi(a_t|s_t; \theta)$. Then it selects $a_{t+1} \sim \pi(\cdot|s_{t+1}; \theta)$.

We need to be careful in the way we select the function approximation $Q(\cdot; w)$ since it might introduce a bias. The following theorem gives a guarantee that we will not have such a bias.

A value function is *compatible* if,

$$\nabla_w Q(s, a; w) = \nabla_\theta \pi(a|s; \theta)$$

The expected square error of w is

$$SE(w) = E^\pi[(Q^\pi(s, a) - Q(s, a; w))^2]$$

Theorem 9.2. *Assume that Q is compatible and w minimizes $SE(w)$, then,*

$$\nabla_\theta J(\theta) = E^\pi[\nabla \log \pi(a|s; \theta) Q(s, a; w)]$$

Proof. Since w minimizes $SE(w)$ we have

$$\begin{aligned} 0 &= \nabla_w SE(w) \\ &= \nabla_w E^\pi[(Q^\pi(s, a) - Q(s, a; w))^2] \\ &= 2E^\pi[(Q^\pi(s, a) - Q(s, a; w))\nabla_w Q(s, a; w)] \end{aligned}$$

Since Q is compatible, we have $\nabla_w Q(s, a; w) = \nabla_\theta \pi(a|s; \theta)$ which implies,

$$0 = E^\pi[(Q^\pi(s, a) - Q(s, a; w))\nabla_\theta \log \pi(a|s; \theta)]$$

and have

$$E^\pi[Q^\pi(s, a)\nabla_\theta \log \pi(a|s; \theta)] = E^\pi[Q(s, a; w)\nabla_\theta \log \pi(a|s; \theta)]$$

This implies that by substituting this in the policy gradient theorem we have

$$\nabla_\theta J(\theta) = E^\pi[\nabla \log \pi(a|s; \theta) Q(s, a; w)]$$

□

We can summarize the various updates for the policy gradient as follows:

- REINFORCE (which is a Monte-Carlo estimate) uses $E^{\pi}[\nabla \log \pi(a|s;\theta) R_t]$.
- Q-function with actor-critic uses $E^{\pi}[\nabla \log \pi(a|s;\theta) Q(a_t|s_t;w)]$.
- A-function with actor-critic uses $E^{\pi}[\nabla \log \pi(a|s;\theta) A(a_t|s_t;w)]$m where $A(a|s;w) = Q(s,a;w) - V(s;w)$.
- TD with actor-critic uses $E^{\pi}[\nabla \log \pi(a|s;\theta)\Delta]$, where Δ is the TD error.

9.5 Application

9.5.1 RoboSoccer: training Aibo

Aibo is a small robot that was used in the Robo-Soccer competitions. One of the main tasks was to get the robot to walk fast and stable. For this the UT Austin team used reinforcement learning, and specifically policy gradient.

The robot is controlled through 12 parameters which include: (1) For the front and rear legs tree parameters: height, x-pos., y-pos. (2) For the locus: length/skew multiplier. (3) height of the body both front and rear. (4) Time (per foot) to go through locus, and (5) time (per foot) on ground or in the air.

The training was done in episodes. The Aibo was trained to walk between two landmarks. (See Figure 9.2.) The optimization used a Policy gradient improvement using Finite difference method. Note that since the actual policy is unknown, there are not that many alternatives.

Given a set of parameters θ, there was a perturbation introduced in the following way. The value of θ_i was modified to $\theta_i + z_i$ where z_i is selected at random from $\{+\epsilon_i, 0, -\epsilon_i\}$. The values of the ϵ_i we selected to be small compared to the value of θ_i. After the perturbation we have a new vector θ'. Many such vectors are sampled. For each vector the policy is evaluated (the return is the time to walk a certain distance, and we like to minimize the time).

After running many such polices, the update is done as follows. For each attribute $i \in [1, 12]$, we split the policies to three subsets, according to the value of z_i, and compute the average return in each subset. If the best outcome is for $z_i = 0$ then we set $A_i = 0$. Otherwise, we set $A_i = avg_i(+\epsilon_i) - avg_i(-\epsilon_i)$. The parameter vector is set to

$$\theta \leftarrow \theta + \alpha \frac{A}{\|A\|}$$

(See also Figure 9.3.)

More figures and data are in the slides.

Figure 9.2: The Aibo training environment

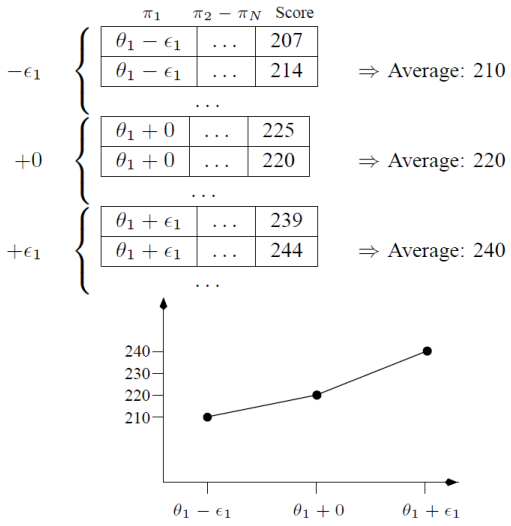

Figure 9.3: The policy updates

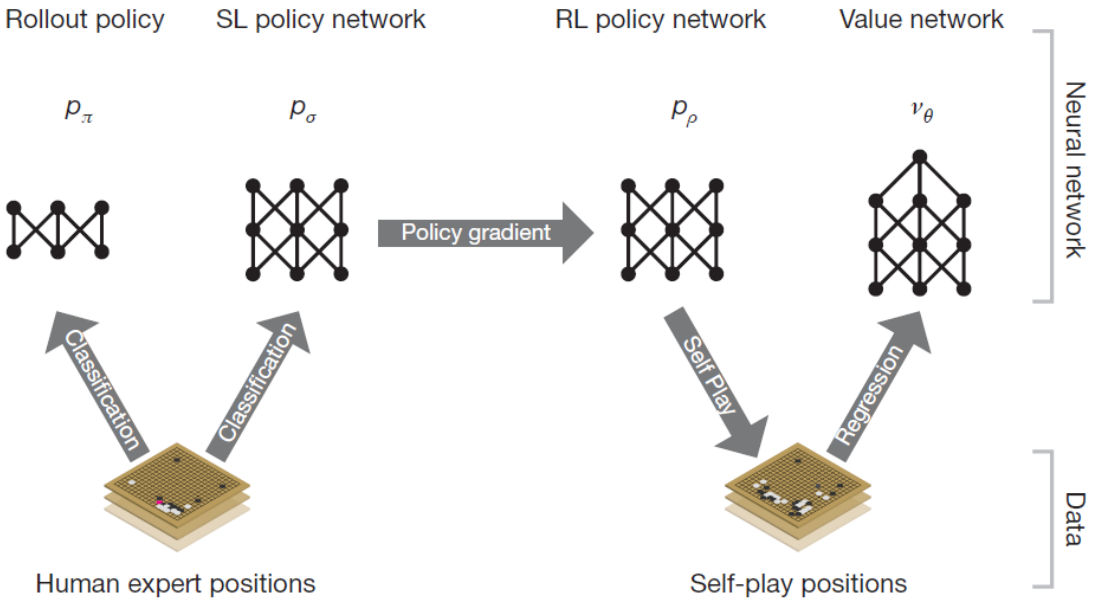

Figure 9.4: The AlphaGo training process

9.5.2 AlphaGo

AlphaGo and its successor AlphaGo Zero are the most advanced computer Go players, and the first to beat the best professional players. We will concentrate on AlphaGo. AlphaGo Zero has an additional benefit that it does not have any prior information about the game (mainly in the form of games between human experts).

The training has roughly three phases. The first phase learns from from human games (played by experts). Those games are used to derive a network SL (Supervised Learning) to predict next move of the human, and a Rollout which is a simple but very fast prediction (3 orders faster than SL).

During phase two an RL network is trained. It is initialized with SL weights, and is optimized on self-play against itself.

In phase three, a value network is trained to predict the winner of the game (trained on the RL network self-play games). (See Figure 9.4.)

Phase 1

The SL network is trained to predict the human moves. It is a deep network (13 layers), and the goal is to output $p(a|s;\sigma)$ where σ are the parameters of the network.

The parameters are update using a gradient step,

$$\Delta\sigma \propto \frac{\partial \log p(a|s;\sigma)}{\partial\sigma}$$

Overall, this network increased the prediction accuracy (compared to previous computer programs) from 44% to 55%. We should note that the increase in accuracy translated to a much more significant improvement in the performance.

In addition to SL, another very fast classifier Rollout was build. It uses a linear function approximation and predicts using a soft-max. The accuracy is only 24% but it is much faster ($2\mu s$ compared to $3ms$, a factor of 1500).

Phase 2

We learn an RL network that outputs $p(a|s;\rho)$ where ρ are the parameters of the network. The network structure is identical to that of SL, and the RL is initialized to the weights of SL, i.e., σ.

The RL is trained using self-play. Rather then playing against the most recent network, the opponent is selected at random between the recent RL configurations. This is done to avoid overfitting. The rewards of the game are given only at the end (win or lose).

The training is done using SGD with policy gradient

$$\Delta\rho \propto \frac{\partial \log p(a|s;\rho)}{\partial\rho}$$

The learned RL network outperformed SL (wins 80%) of the games. However, SL is better in predicting expert human behavior.

Phase 3

We learn a value function $V(s;\theta)$. The goal is to predict the probability that RL will win, and it is trained on the self-play data of RL. It uses a SGD update

$$\Delta\theta \propto \frac{\partial V(s;\theta)}{\partial\theta}$$

When the training was done on complete games, there was a serious problem of over-fitting. (The training error was 0.19 while the test error was 0.37). For this reason, the training is limited to only one position per game. The error rate is about 0.22.

The system uses a Monte-Carlo Search Trees (MCST) to evaluate the positions. The tree is used to perform a lookahead. It uses the Rollout policy to generate

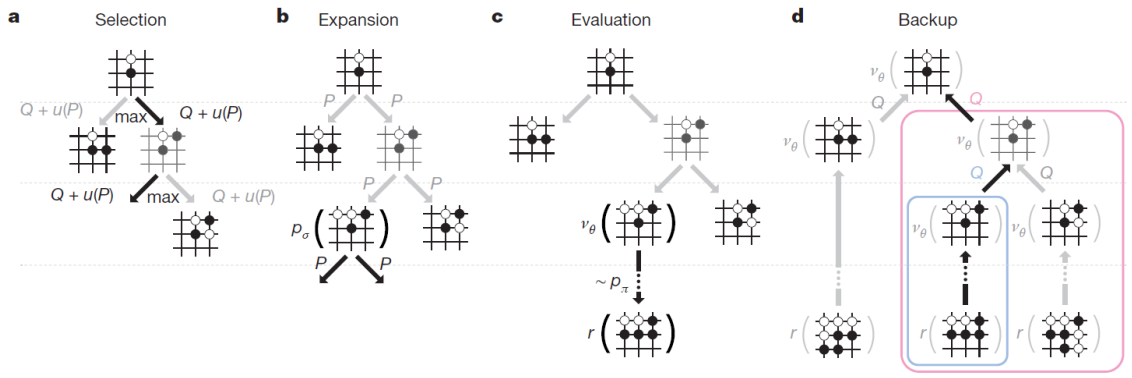

Figure 9.5: AlphaGo use of Monte-Carlo trees

trajectory from the given position. The prediction is an average of the value network prediction $V(s;\theta)$ and the outcome of the Rollout policy. To see a run of the Monte-Carlo Search Tree see Figure 9.5.

9.6 Bibliography Remarks

The training of Aibo for RoboSoccer is by [1].

The work on AlphaGo is by [2].

Part of the outline borrows from David Silver class notes and the the book of Sutton and Barto [3].

Bibliography

[1] Nate Kohl and Peter Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *Proceedings of the 2004 IEEE International Conference on Robotics and Automation, ICRA 2004, April 26 - May 1, 2004, New Orleans, LA, USA*, pages 2619–2624, 2004.

[2] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998.