# Solution of a Toy Problem by Reinforcement Learning

David MacKay, Iain Murray, and Peter Latham

March 1, 2006

## 1 Problem statement

You get to toss coins, each of which is either of type H, having probabilities favouring heads $\{(1-f), f\}$ or of type T, with probabilities $\{f, (1-f)\}$, where $f$ is a known bias parameter such as $f = 0.1$. At each time step, one may either *continue* tossing the current coin, or one may *declare* a guess for whether it is of type H or T. Declarations are rewarded if accurate and penalized if incorrect. The cost of continuing is zero. The cost of declaring incorrectly is a penalty of $R_-$; declaring correctly yields a reward of $R_+$. To make the problem interesting, we set $R_+ < R_-$. The task is to choose a *strategy* that maximizes the average return per unit time. We include no discounting in the problem. Once you have declared for coin $t$ and been rewarded or penalized, you get to move on to coin $n+1$. There is an infinite supply of coins, all with the same bias $f$. The two types $H$ and $T$ are equiprobable. Declaring takes no time: when you declare, you immediately get to toss the next coin.

## 2 Exact numerical solution

The optimal strategy can be characterized by a single number $\mu$ which is the absolute difference between the number of heads and number of tails at which we declare. How we got to that point is irrelevant. Let $n = F_{\rm H} - F_{\rm T}$ be the difference between the number of heads and the number of tails produced by the current coin. If the absolute value of difference is less than $\mu$, we continue tossing the coin. Given the sufficient statistic $n$, the posterior probability of the theory that the coin is of type H is

$$P({\rm H}\,|\,n) = \frac{1}{1+\exp(-\beta n)} \equiv p_n$$

where $\beta = \log\left[(1-f)/f\right]$. We call $n$ the state. The state starts at $n = 0$. Given $n$, the predictive distribution for the next outcome to be a head, thus increasing $n$ by 1, is

$$T_{\rm up}(n) = p_n(1-f) + (1-p_n)f$$

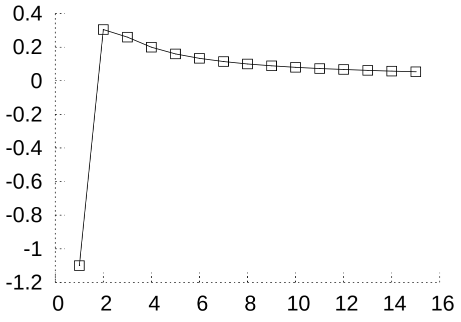
Similarly

$$T_{\rm down}(n) = p_n f + (1-p_n)(1-f)$$

For any candidate value of $\mu$ we can compute two quantities: (a) the expected reward $R(\mu)$; that's easy, it's just

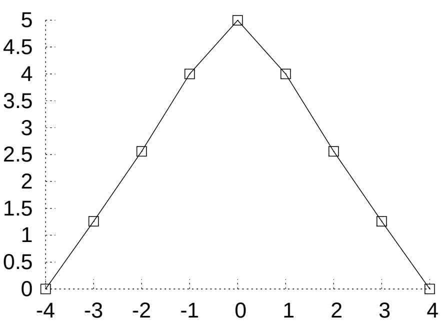
$$R(\mu) = p_\mu R_+ - (1-p_\mu)R_- = -R_- + p_\mu(R_+ + R_-);$$

and (b) the expected time to get that expected reward if we are currently in state $n$ and the policy is $\mu$: $t_\mu(n)$.



| $\mu$ | Rate | Reward | Average time per coin |
|---|---|---|---|
| 1 | −1.100 | −1.100 | 1.000 |
| 2 | 0.305 | 0.744 | 2.439 |
| 3 | 0.260 | 0.971 | 3.739 |
| 4 | 0.199 | 0.997 | 4.998 |
| 5 | 0.160 | 0.9996 | 6.249 |

Figure 1. Expected reward per unit time (vertical axis) as a function of strategy $\mu$ (horizontal axis). The parameters are $f = 0.1$, $R_- = 20$, and $R_+ = 1$. (The first strategy, $\mu = 1$, is easy to check manually.)



| $n$ | Average time to go | $P(\mathrm{H} \mid n)$ |
|---|---|---|
| −4 | 0 | 0.00015 |
| −3 | 1.26 | 0.00136 |
| −2 | 2.5595 | 0.01219 |
| −1 | 3.9985 | 0.10000 |
| 0 | 4.9985 | 0.50000 |
| 1 | 3.9985 | 0.90000 |
| 2 | 2.5595 | 0.98780 |
| 3 | 1.26 | 0.99863 |
| 4 | 0 | 0.99984 |

Figure 2. Expected time to go, from each state $n$, when the strategy is $\mu = 4$.

The expected time satisfies

$$t_\mu(\mu) = 0; \quad t_\mu(-\mu) = 0; \quad t_\mu(n) = 1 + T_{\text{up}}(n)t_\mu(n{+}1) + T_{\text{down}}(n)t_\mu(n{-}1).$$

We can solve for $\mathbf{t}_\mu$ by matrix inversion.
Dividing $R(\mu)$ by $t_\mu(0)$, we find the average reward per unit time.

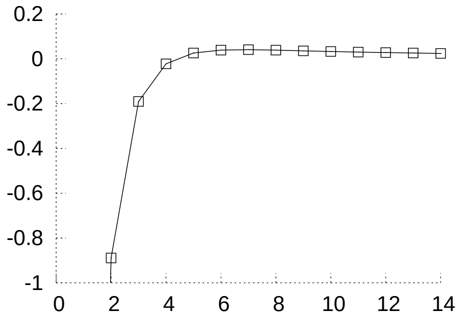
### 2.1 Illustration

Let $R_- = 20$ and $R_+ = 1$ (so that declarations are only worth making when you have greater than 95% confidence). Let $f = 0.1$.
The expected reward per unit time (the 'Rate') is shown in figure 1. The optimum strategy is $\mu = 2$. Source code in `octave` is given in appendix A.
Figure 3 shows the reward rate as a function of strategy $\mu$ for parameters $f = 1/3$, $R_- = 20$, and $R_+ = 1$. In this case the optimal strategy is $\mu = 7$.

## 3 Alternative problem statement by I.A.M.

Aim: a slightly different approach that might bridge between the direct solution and some RL algorithms.

As before: you get to toss coins, each of which is either of type H, having probabilities favoring heads $\{(1-f), f\}$ or of type T, with probabilities $\{f, (1-f)\}$, where $f$ is a known bias parameter



| $\mu$ | Rate | Reward | Average time per coin |
|---|---|---|---|
| 1 | -6.000 | -6.000 | 1.00 |
| 2 | -0.888 | -3.200 | 3.60 |
| 3 | -0.190 | -1.333 | 7.00 |
| 4 | -0.022 | -0.235 | 10.58 |
| 5 | 0.025 | 0.363 | 14.09 |
| 6 | 0.038 | 0.676 | 17.44 |
| 7 | 0.040 | 0.837 | 20.67 |
| 8 | 0.038 | 0.918 | 23.81 |
| 9 | 0.035 | 0.959 | 26.89 |
| 10 | 0.032 | 0.979 | 29.94 |

Figure 3. Expected reward per unit time (vertical axis) as a function of strategy $\mu$ (horizontal axis). The parameters are $f = 1/3$, $R_- = 20$, and $R_+ = 1$.

such as $f = 0.1$. At each time step, one may either *continue* tossing the current coin, or one may *declare* a guess for whether it is of type H or T. Declarations are rewarded if accurate, with reward $R_+$, and penalized if incorrect with penalty $R_-$. It is assumed $R_+ < R_-$. Penalties are negative rewards.

Now we assume this is a one shot game, but our time is still valuable, so we associate a penalty $c > 0$ with each request to see another toss. Our aim is to come up with a *strategy* that maximizes the expected net total reward. There is still no discounting in this problem.

## 4 Solution to the alternative problem

As in the previous problem, all that matters is $n = F_{\mathrm{H}} - F_{\mathrm{T}}$, the difference between the number of heads and tails. As before we let $p_n$ be the posterior probability of the coin being of type H given the sufficient statistic $n$. First we pick a particular policy determined by a particular value of $\mu$, the absolute value of $n$ at which we declare the most probable identity of the coin. Let $V_n$ be the *value* of being in state $n$, giving the total expected future reward when using policy $\mu$. The policy giving the largest $V_0$ is the one that we should use.

When in states $n = \pm\mu$ we will declare a guess, the game finishes and so the total expected future reward is equal to the expected reward for the final action:

$$V_\mu = V_{-\mu} = p_\mu R_+ - (1 - p_\mu)R_-. \tag{1}$$

In other states we'll toss the coin again and stay in the game. The value (total expected future reward) of these states is the expected value of the next state minus the costs associated with the action.

$$V_n = V_{n+1}\left(p_n(1-f) + (1-p_n)f\right) + V_{n-1}\left(p_n f + (1-p_n)(1-f)\right) - c \quad \forall\ |n| < \mu \tag{2}$$

I believe these could all be called Bellman equations and various general RL algorithms apply. Or we could spot that it's just a system of linear equations and solve using matrix division.

## 5 Back to the original problem

In the original problem there was no explicit cost $c$ for tossing the coin. The problem was expressed as wishing to maximize the average reward per unit time. We can get the same effect by setting

$c$ equal to the average reward rate (excluding negative rewards due to $c$). That is the worth of a unit of time if there are no other games available. Time could be more valuable if there are better games we could play.
If we have a fixed policy and the average reward rate is computed under the same policy then the net expected value of the game has been fixed to zero, i.e.

$$V_0 = 0. \tag{3}$$

This gives the additional equation needed to solve for an additional unknown. The equations in $(\{V_n\}, c)$ are still linear, so we can still solve directly for $c$. These average reward rates for each policy agree with the David's solutions. As before we would pick the policy with the best reward rate.

## 6 Solution with $TD(\lambda)$ and $Q$–learning

Peter can write this bit? :-)

## A Source code for exact solution

coin0.m

```
## Works out the expected reward and the expected time to get that
## reward.                    DJCM                  Mon 27/2/06

## required inputs:
## f  (in (0,1)      bias of both coins   (f<0.5)
## mu (integer)      policy defining maximum value of n
## Rplus, Rminus     Reward and Penalty for correct and incorrect guesses

States = 2*mu-1 ; # The number of states required for matrix inversion
Zero   = mu     ; # make a note of the Zero state.
Mu     = States+1 ;
Go = ones(States,1) ;
T = zeros(States,States); ## transition matrix (to be)
beta = log((1-f)/f) ;     ## log odds per outcome
for s = 1:States+1
  n = s-Zero   ;    ## obtain the preferred index that I would use in a
                    ## sensible programming language.
  po(s) = 1.0/(1+exp(- beta * n )) ; ## posterior probability of Heads bias.
  T_up(s) = po(s) * (1-f) + (1-po(s)) * f ;
endfor
for s = 1:States
  if (s<States)
    T( s, s+1 ) = T_up(s);
  endif
  if (s>1)
    T( s, s-1 ) = 1.0-T_up(s);
  endif
endfor
## Solve the equation   TimeToGo = Go + T * TimeToGo
##                          i.e. (1-T) * TimeToGo  = Go
TimeToGo = (eye(States)-T) \ Go
Reward = po(Mu) * (Rplus+Rminus)   - Rminus
Rate = Reward / TimeToGo(Zero)
```

---

## B Iain's source code

solvev.m

```
% This code should work in recent versions of Octave or Matlab

f=0.1;                          % bias of coins (<0.5)
mu=3;                           % (#heads - #tails) at which we declare
Rplus=1;                        % reward for being correct
Rminus=20;                      % cost=-reward for being wrong

ns=-mu:mu;                      % possible states
beta=log((1-f)/f);
p_ns=1./(1+exp(-beta*ns));
p_mu=p_ns(end);

% Solve:
% V_-mu = p_mu R+ - (1-p_mu) R-
% for n=(-mu+1):(mu-1)
%   V_n = V_{n+1} (p_n(1-f)+(1-p_n)f) + V_{n-1} (p_n f + (1-p_n)(1-f)) - c
% V_mu = p_mu R+ - (1-p_mu) R-
% V_0 = 0
% let vars be (V_-mu, ..., V_mu, c)'
% and rewrite as A*vars = RHS

A=eye(length(ns)+1);
RHS=zeros(length(ns)+1,1);

% V_-mu = p_mu R+ - (1-p_mu) R-
RHS(1)= p_mu*Rplus - (1-p_mu)*Rminus;

% V_n = -c + V_{n+1} (p_n(1-f)+(1-p_n)f) + V_{n-1} (p_n f + (1-p_n)(1-f))
A(2:2*mu,end)=1;
for n=(-mu+1):(mu-1)
    idx=n+mu+1;
    A(idx,idx+1)=-(p_ns(idx)*(1-f)+(1-p_ns(idx))*f);
    A(idx,idx-1)=-(p_ns(idx)*f+(1-p_ns(idx))*(1-f));
end

% V_mu = p_mu R+ - (1-p_mu) R-
RHS(2*mu+1)= p_mu*Rplus - (1-p_mu)*Rminus;

% V_0 = 0
A(end,end)=0;
A(end,mu+1)=1;
```

```
vars = A\RHS;
c=vars(end);

average_reward_unit_time=c
```