# Multi-Armed Bandit

January 10, 2020



Co-funded by the
Erasmus+ Programme
of the European Union


# 1 Multi-Armed Bandit

Consider an agent (a decision maker) faced with a number of slot machines (one-armed bandits). Each time the decision maker decides to play on some slot machine (each time he/she pulls the corresponding leaver), he/she receives certain reward, or has to pay some penalty. The actual amount received or payed is a stochastic quantity, decided by chance, depending also on the selected slot machine, since not all "bandits" are alike, and some are more generous than others. However, since all slot machines initially look alike to the decision make, the only way to decide which one is more favorable is to experiment by pulling different leavers. The goal is to maximize total accumulated reward received over a long sequence of repeated plays.

## 1.1 Preparataion

[1]:
```
# Import and initialize plotting library
import StatsPlots
import StatsPlots: @df, plot, plot!
StatsPlots.Plots.gr();
```

[2]:
```
# Import and initialize random number generators
import Random, Distributions
Random.seed!(round(Int64, time() * 1000));
```

[3]:
```
# Import and initialize data frames
import DataFrames
import DataFrames: DataFrame
```

## 1.2 An Approach to Simulating Automata

Let us consider a decision making problem in more detail. The decision maker can be seen as an automaton which advances its internal state based on rewards received from the environment. At each decision time instant, the agent makes a decision, it takes an action based on its internal state. Likewise, the environment itself is an automaton, which evolves driven by the actions received from the decision maker, and rewards the decision maker accordingly.

**Automaton** is fully specified by a function mapping a pair of its current state and externally given input (x, u) to the pair (x_, y) of the consecutive state and output. Such function is termed **state transition function**.

Considering that the state is to be considered internal to an agent, it is possible to consider its input-output form, in which the state is hidden. The actual implementation returns also the state, but only for debugging and logging purposes. The exposed state is actually a *deep copy* of the agent state: it is completely appropriate to manipulate the returned state, the automaton itself will not be affected.

Consequently, if performance requires, the state transition function need not be pure and may mutate the state in-place. The code will run safely.

[4]:
```
"""
Transforms state-transition-function `stf` into a dynamic system (automaton)
in input-output form, in which the state is evolved internaly starting from
the given initial value `x0`.

The returned system is a function of the form `u -> (y, x)`, where `u` is the␣
 ↪system
input, while `y` and `x` are the corresponding output and the following state.
"""
function toInOut(stf, x0; exposeState=false)
    # Array is used only to allow the inner function
```

```
    # to modify the state.
    x = [x0]
    u ->
    begin
        x_, y = stf(x[1], u)
        x[1] = x_
        # Deep copy is used because the internal state is exposed
        # and this is unsafe.
        if exposeState
            return (y, deepcopy(x_))
        else
            return y
        end
    end
end;
```

Let us now build a generic function used to simulate any automaton. Function `evolution` is essentially an iterator, which can be used to yield output one by one.

[5]:
```
function evolution(automaton, input)
    (automaton(u) for u in input)
end
```

[5]:
```
evolution (generic function with 1 method)
```

### 1.2.1 Example 1: Simple integrator

Let us test the above framework using a simple example, unrelated to the bandit problem: consider a simple integrator.

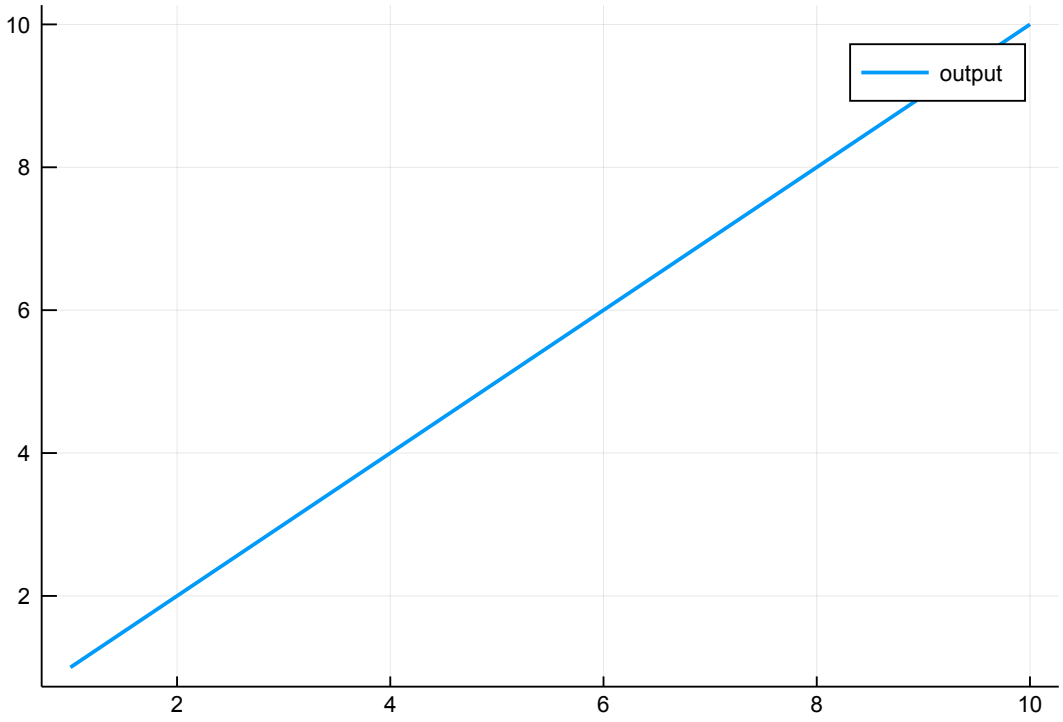
[6]:
```
integrator = (x, u) -> (x+u, x+u);
```

Unit step response of the integrator will be simulated. We will feed the integrator with constant stream of 1, and we will expect the output to grow as unit ramp.

[7]:
```
integrator! = toInOut(integrator, 0)
step_response = evolution(integrator!, (1 for _ in 1:10));
output = [y for y in step_response];
```

[8]:
```
plot(output, linewidth=2, label="output")
```

[8]:




### 1.2.2 Agent-Environment Interaction

Interaction between the agent and its environment can also be described in generic terms. We assume that the agent is invoked first. That its output (action) is fed into the environment, which produces the appropriate reward, and feeds it back to the agent. This implies that the reward of the first iteration is not well defined, and that the proper implementation of agent need to take this into consideration.

Note, however, that the topology of the interaction is quite generic, and deserves implementation of its own.

[9]:
```
function sequentialLoopStf(stf1, stf2)
    ((x1, x2, u1), u) ->
    begin
        x1_, y1 = stf1(x1, u1)
        x2_, y2 = stf2(x2, y1)
        return ((x1_, x2_, y2), (y1, y2))
    end
end;
```

First note that that compound system is completely autonomous: the input is completely disregarded. Note also that the state of the compound system is

three-fold: it contains of the state of the first system, the state of the second systems, and of the output of the second system from the last iteration. This value will be used as the input to the first system in the following iteration.

[10]:
```
"""
Simulates `iters_count` successive iterations of interaction between an agent␣
 ↪and
its environment. Both agent and the environment are assumed to be in the␣
 ↪input-output
form (hense `!` after argument names: each iteration modifies the internal␣
 ↪state of both).

The return value is a tuples containing arrays of: actions, rewards, agent's␣
 ↪states and
environment's states.
"""
function interact(agent, environment, x0agent, x0environment, iters_cnt,␣
 ↪init_input)
    x0 = (x0agent, x0environment, init_input)
    loop = toInOut(sequentialLoopStf(agent, environment), x0; exposeState=true)
    results = evolution(loop, nothing for _ in 1:iters_cnt)
    rewards = []
    actions = []
    agent_states = []
    environment_states = []
    for ((action, reward), (x_agent, x_env, _)) in results
        push!(rewards, reward)
        push!(actions, action)
        push!(agent_states, x_agent)
        push!(environment_states, x_env)
    end
    return actions, rewards, agent_states, environment_states
end;
```

Note also that due to deep copying in the implementation of toInOut the automaton need no be implemented as a pure function! It can actually mutate state. The state is captured within toInOut and only its deep copy is exposed outside (for logging and debugging).

Since the state is not needed ``in production'', being the internal quantity of agents, it is possible to make a version of the toInOut function which would not expose state at all. The corresponding modified interact function would also not expose state evolution of neither state agent nor the environment.

### 1.2.3 Example 2: Simple integrator in closed loop

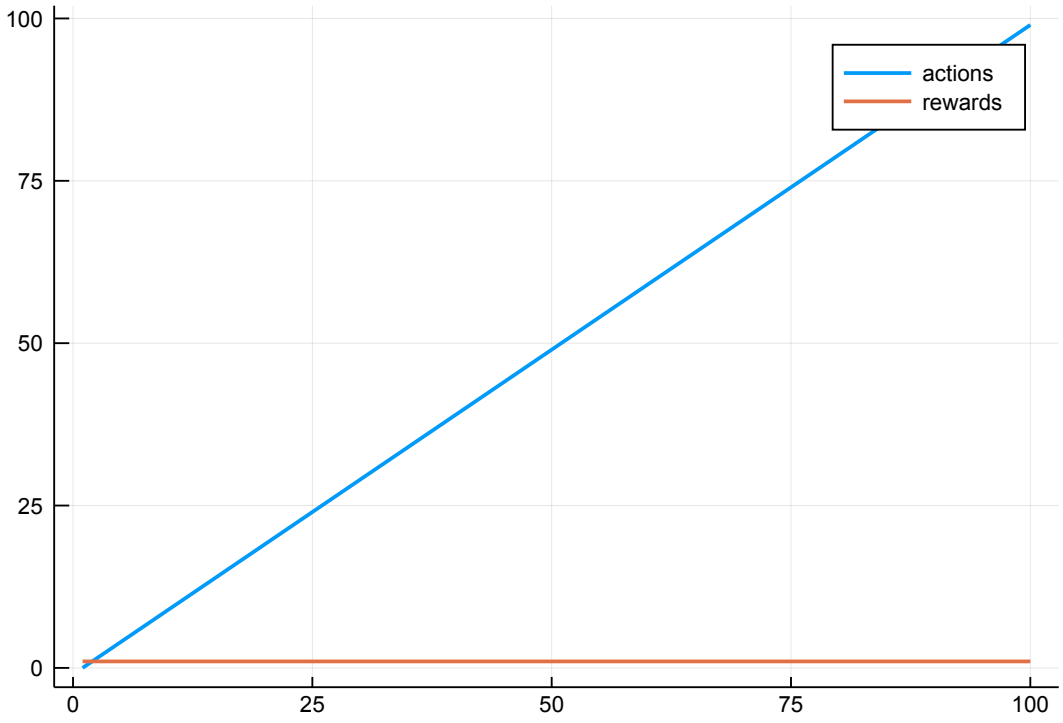
Consider a simple integrating agent, whose output at any time instant is just the sum of all rewards received in the past. Let us embed this agent in a simple indifferent environment which rewards all actions with the same amount (which we set to 1).

[11]:
```
environment = (x, u) -> (x, 1.0);
agent = (x, u) -> (x+u, x+u);
```

[12]:
```
actions, rewards, integrator_states, environment_states = interact(agent,␣
 ↪environment, 0, 0, 100, 0);
```

[13]:
```
plot(actions, label="actions", linewidth=2)
plot!(rewards, label="rewards", linewidth=2)
```

[13]:




## 1.3 Multi-Armed Bandits

### 1.3.1 Problem Statement

Let us model each ``bandit'' by the corresponding reward probability distribution. Without loss of generality, let us assume that the reward given by each bandit is normally distributed, with given mean and variance different for

each agent. Means and variances of these distributions will be selected randomly, uniformly within appropriate ranges: all means will be selected between 0 and 1, while all standard deviations will be selected in range between 0 and 0.25.

[46]:
```
function generate_distributions(bandits_no)
    means = [rand(Distributions.Uniform(0, 1)) for _ in 1:bandits_no]
    stds = [rand(Distributions.Uniform(0, 0.25)) for _ in 1:bandits_no]
    dist = [Distributions.Normal( ,  ) for ( ,  ) in zip(means, stds)]
    return dist, means, stds
end;
```

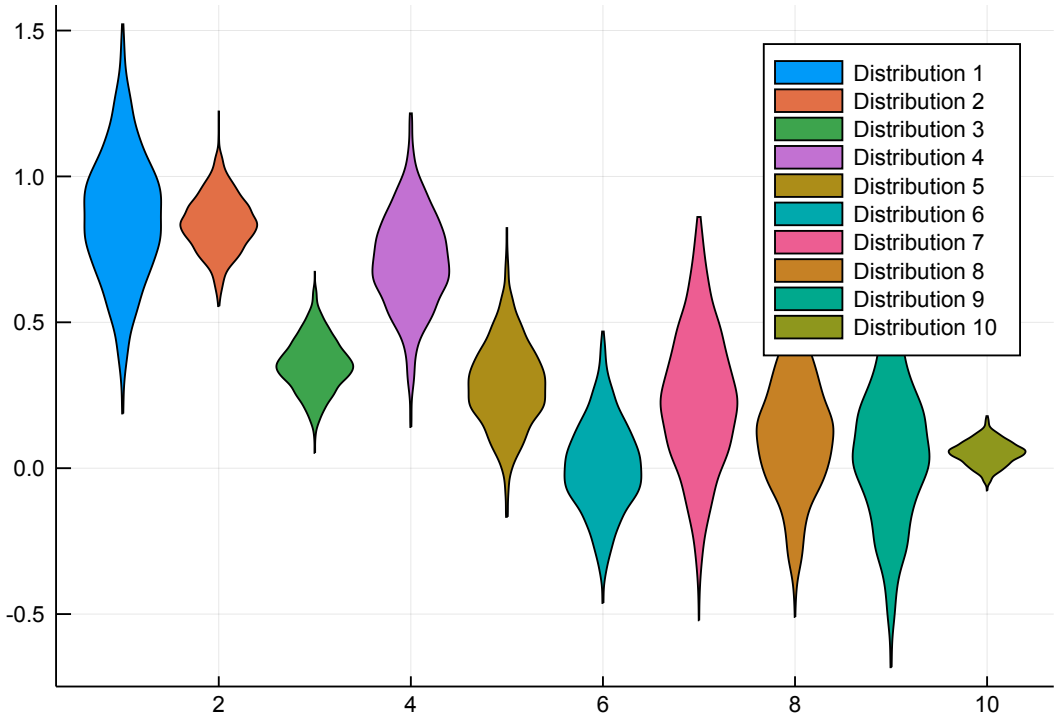
[47]:
```
distributions, means, stds = generate_distributions(10);
```

Let us now sample and visualize the generated distributions.

[48]:
```
function sample_distributions(distributions)
    samples = DataFrames.DataFrame()
    for (index, d) in enumerate(distributions)
        name = Symbol("Distribution $index")
        samples[name] = rand(d, 1000)
    end
    return samples
end;
```

[49]:
```
samples = sample_distributions(distributions)
@df samples StatsPlots.violin(cols())
```

[49]:



Let us now define the environment. In multi-armed bandit problem, the environment is stateless: past actions of the decision makers and past rewards do not affect the current reward. The current reward is a stochastic quantity, but it depends solely on the last action of the agent.

[50]:
```
function bandits(distributions)
    (state, action) ->
    begin
        reward = 0
        if action > 0 && action <= length(distributions)
            reward = rand(distributions[action])
        end
        return state, reward
    end
end;
```

### 1.3.2 Solution: $\varepsilon$-greedy agent

[51]:
```
function  Greedy( , bandits_no)
    dist = Distributions.Uniform(0, 1)
    inner = (state, reward) ->
    begin
        lastAction, accumulators, counters = state
        if !(lastAction == 0)
            accumulators[lastAction] += reward
            counters[lastAction] += 1
        end
        # We do not need to take extra precautions here, because
        # Julia handles NaN, Inf, etc. gracefully.
        if rand(dist) <
            action = argmin(counters)
        else
            action = argmax([a/c for (a, c) in zip(accumulators, counters)])
        end
        return ((action, accumulators, counters), action)
    end
    return inner, (0, [0.0 for i in 1:bandits_no], [0 for i in 1:bandits_no])
end
```

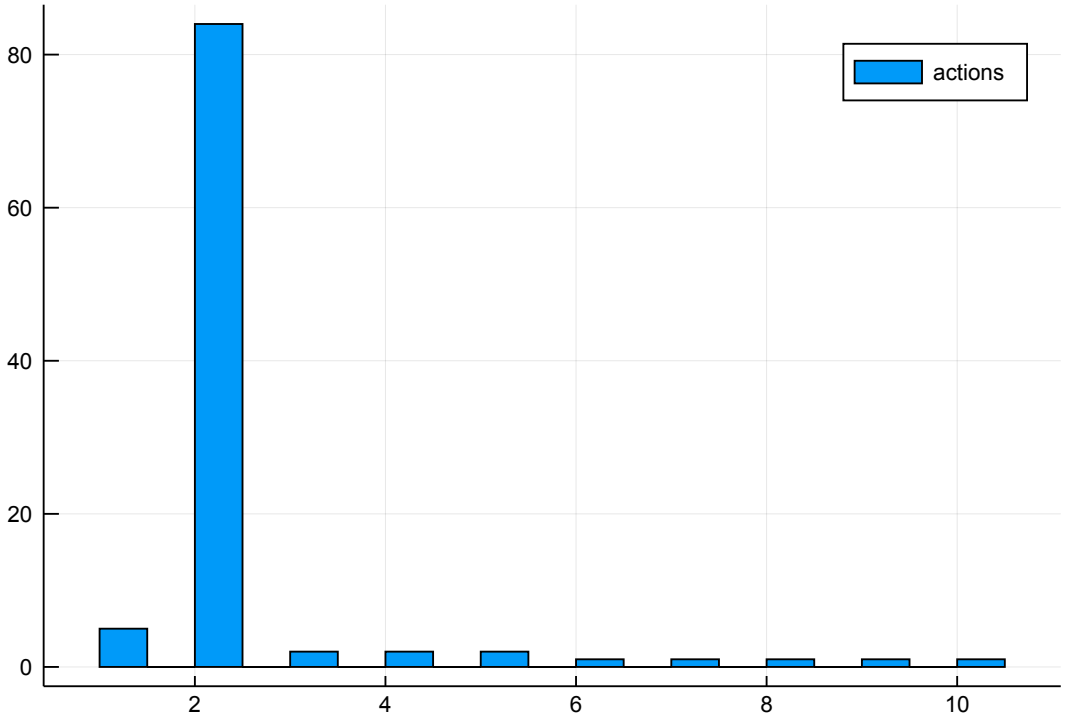
[51]:
```
 Greedy (generic function with 1 method)
```

[52]:
```
bandits_stf = bandits(distributions)
 Greedy_stf,  Greedy_x0 =  Greedy(0.05, 10);
```

```
actions, rewards, agent_states, _ = interact( Greedy_stf, bandits_stf,␣
 ↪ Greedy_x0, 0, 100, 0.0);
```

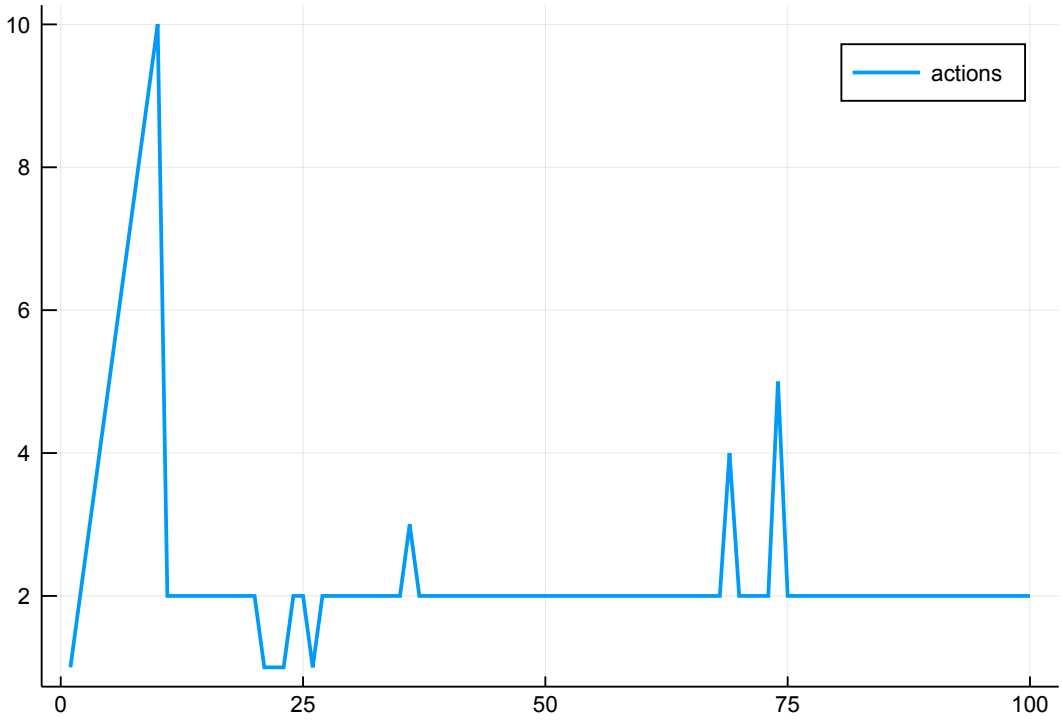
[53]:
```
StatsPlots.histogram(actions, label="actions")
```

[53]:




[54]:
```
plot(actions, label="actions", linewidth=2)
```

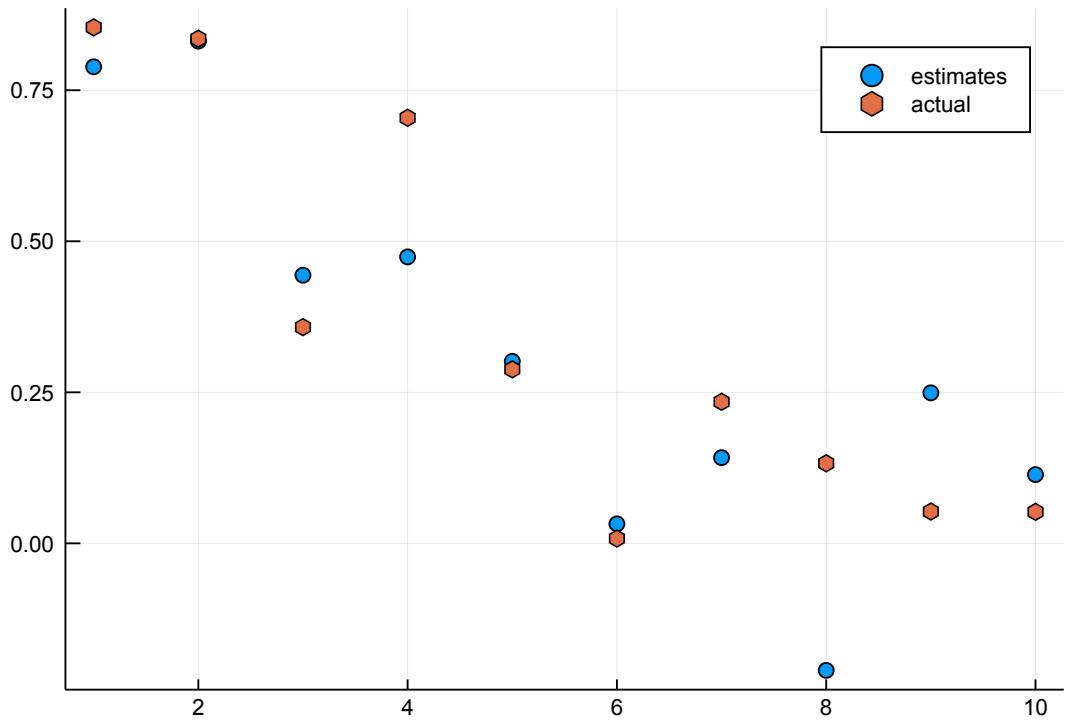
[54]:




[55]:
```
lastState = agent_states[end]
estimates = [a/c for (a, c) in zip(lastState[2], lastState[3])];
```

[56]:
```
plot(estimates, seriestype=:scatter, label="estimates")
plot!(means, seriestype=:scatter, markershape = :hexagon, label="actual")
```

[56]:


estimates
actual
0.75
0.50
0.25
0.00
2
4
6
8
10

[ ]: