Bandits

Ana Matran-Fernandez
University of Essex

February 11, 2019

Bandits

- We are in effect revisiting some ideas from lecture 2
  - Hypothesis testing
- This is a much easier framework to understand than hypothesis testing
- Bandits are the simplest type of reinforcement learning problem

Not supervised learning

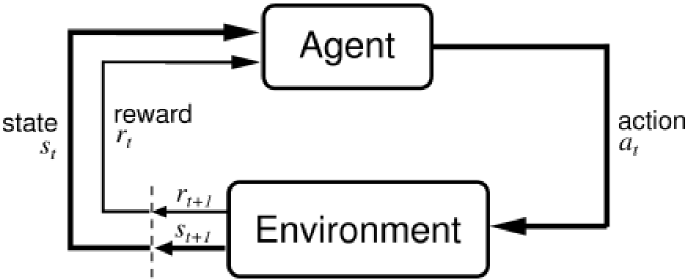
This is **NOT** supervised learning:

- In SL we learn from samples provided by a knowledgeable supervisor (features and labels)
- In RL:
  - There is no supervisor: the agent learns from its own experience as it explores
  - There is no dataset!
- This leads to the exploration vs exploitation dilemma

Exploration vs. Exploitation Dilemma

- Making a decision involves a fundamental choice:
  - Exploitation: Make the best decision given current information
  - Exploration: Gather more information
- The best long-term strategy may involve short-term sacrifices
- Gather enough information to make the best overal decisions
- E.g. Restaurant selection
  - Exploitation: Go to your favourite restaurant
  - Exploration: Try a new restaurant

Reinforcement learning

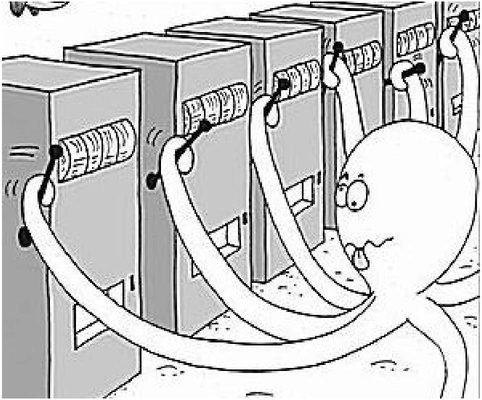


source: `https://github.com/brianfarris/RLtalk/blob/master/RLtalk.ipynb`

Examples

- You send a user an e-mail
  - User clicks on the link you get $r = 1$
  - User fails to click on the link after 3 days $r = 0$
- Playing games
  - What is the next best action to take in Chess?
    - Chess has a sequential element - hence “Reinforcement Learning”
    - But close enough. . .
- Online adverts
  - User clicks on an advert ($r = 1$)
  - User clicks fails to click on an advert ($r = 0$)

Multi-Armed Bandits



source: Microsoft Research

- Agent
- Action
- Reward
- One single state

The multi-armed bandit problem

- A bandit is a tuple $< A, R >$
- Where $a \in A$ is a an action out of a set of actions (or "arms")
- $r \in R$ is a reward from a set of rewards
- $R(a, r) = P(r|a)$
  - The probability of getting a reward r given that I have done action a
  - It's an unknown probability distribution over rewards
- At each step, the agent selects an action
- "You do an action, you get a reward"

The goal

- Find an optimal policy $\pi(a) = P(a)$ that maximises the long term sum of rewards
  - Long term cumulative reward is $\sum_{t=0}^{T} = r_t$
- The *action-value* function is the expected reward for taking action a
  - $Q(a) = E[r|a]$
- The *value* function is $V = E_\pi[r]$
  - The expected reward, given the policy I'm following
  - Optimal $V^* = Q(a^*) = \max_{a \in A} Q(a)$

Example

- Three actions to choose from

```
def action_0():
    return np.random.choice([1, 0], p=[0.5, 0.5])

def action_1():
    return np.random.choice([1, 0], p=[0.6, 0.4])

def action_2():
    return np.random.choice([1, 0], p=[0.2, 0.8])

rewards = [action_0, action_1, action_2]

print(rewards[0]())  # 0
print(rewards[0]())  # 1
print(rewards[0]())  # 0
print(rewards[0]())  # 0
```

Let's simulate: $Q(a_i)$

In [32]:
```
def action_0():
    return np.random.choice([1,0], p=[0.5, 0.5])

def action_1():
    return np.random.choice([1,0], p=[0.6, 0.4])

def action_2():
    return np.random.choice([1,0], p=[0.2, 0.8])

rewards = [action_0, action_1, action_2]
```

In [35]:
```
pulls = 100000

action_value = []
#for action in range(len(rewards)):
#    value = [rewards[action]() for _ in range(pulls)]
#    action_value.append(value)
for reward in rewards:
    value = [reward() for _ in range(pulls)]
    action_value.append(value)
```

In [36]:
```
for action, value in enumerate(action_value):
    print("Action %d: Q(a_%d)=%.2f" % (action, action, np.mean(value)))
```

```
Action 0: Q(a_0)=0.50
Action 1: Q(a_1)=0.60
Action 2: Q(a_2)=0.20
```

Let's simulate: V (1)

In [50]:
```
p0, p1, p2 = 0.33, 0.33, 0.34

def policy():
    return np.random.choice([0, 1, 2], p=[p0, p1, p2])
```

In [51]:
```
tot_reward = 0
for pull in range(pulls):
    action = policy()
    tot_reward += rewards[action]()
print("Total reward =", tot_reward)
print("Average reward: V =", tot_reward/pulls)
```

```
Total reward = 43000
Average reward: V = 0.43
```

In [52]:
```
# Manually:
V = np.mean(action_value[0])*p0 + np.mean(action_value[1]) * p1 + np.mean(action_value[2]) * p2
print("V =", V)
```

```
V = 0.4308734
```

In [53]:
```
# With the formula:
V = 0.5 * p0 + 0.6 * p1 + 0.2 * p2
print("V =", V)
```

```
V = 0.431
```

Let's simulate: V (2)

In [54]:
```
p0, p1, p2 = 0.4, 0.5, 0.1

def policy():
    return np.random.choice([0, 1, 2], p=[p0, p1, p2])
```

In [55]:
```
tot_reward = 0
for pull in range(pulls):
    action = policy()
    tot_reward += rewards[action]()
print("Total reward =", tot_reward)
print("Average reward: V =", tot_reward/pulls)
```

```
Total reward = 51840
Average reward: V = 0.5184
```

In [56]:
```
# Manually:
V = np.mean(action_value[0])*p0 + np.mean(action_value[1]) * p1 + np.mean(action_value[2]) * p2
print("V =", V)
```

```
V = 0.519739
```

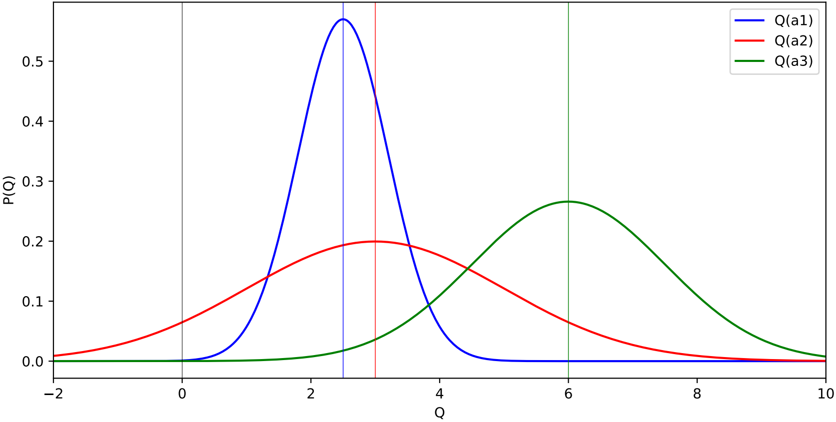
In [57]:
```
# With the formula:
V = 0.5 * p0 + 0.6 * p1 + 0.2 * p2
print("V =", V)
```

```
V = 0.52
```

Goals (1)

- So our goal is to find the best action
- Optimal $V^* = \max_{a \in A} Q(a)$
- But these values can only be found through averages
  - $\hat{Q}(a), \hat{V}$
- We could have done hypothesis testing (recall Lecture 2)
  - But this would entail a random policy
  - Maybe we can do better

$\hat{Q}(a)$



Goals (2)

- We would like to find the best action using the minimum amount of trials possible
- Keep focusing on the best action
  - While also checking making sure that other actions are sufficiently explored
- This is known as the "exploration/exploitation" dilemma

Regret

- *Regret* is the opportunity loss for one step
  - i.e., the difference between the actual payoff and the one you would have if you had played the best option
  - $I_t = E\left[(V^* - Q(a_t))\right]$
- *Total regret* is the total opportunity loss
  - $L_t = E\left[\sum_{t=0}^{T}(V^* - Q(a_t))\right] = E\left[\sum_{t=0}^{T}\left(\max_{a \in A} Q(a) - Q(a_t)\right)\right]$
- It helps us understand how well an algorithm could possibly do, independently of the scale of the rewards

Let's simulate: Regret

In [72]:

```
# Regret
V_star = max([np.mean(value) for value in action_value]) # 0.6

tot_regret = 0
for pull in range(pulls):
    tot_regret += (V_star - rewards[policy()]())
print("Regret: I_t = %.2f" % (tot_regret/pulls))
```

```
Regret: I_t = 0.08
```

In [73]:

```
I = (V_star - 0.5) * p0 + (V_star - 0.6) * p1 + (V_star - 0.2) * p2
print("I_t = %.2f" % I)
```

```
I_t = 0.08
```

In [74]:

```
print("Total regret: L_t = %.2f" % tot_regret)  # also called cumulative regret
```

```
Total regret: L_t = 7896.00
```

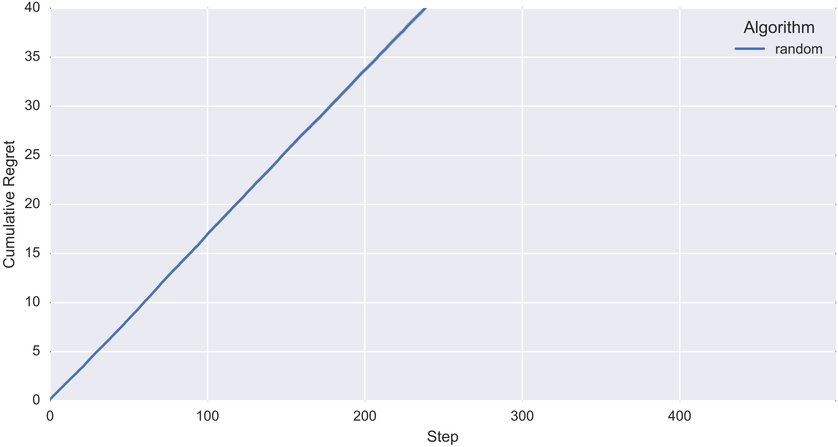
Counting regret

- The *count* $N_t(a)$ is the number of times we took action a until time t
- The *gap* Δ_a is the difference between the value of the optimal action and that of the action taken, $\Delta_a = V^* - Q(a)$
- It turns out that regret can be written in terms of gaps and counts:
  - $L_t = \sum_{a \in A} \left(E\left[N_t(a) \Delta_a \right] \right)$
- A good algorithm ensures small counts for large gaps
- But we have no clue what the gaps are…

Pure exploration

- Somewhat similiar to the A/B case
- You send more or less the equal number of e-mails
- Very simple setup
- Link: When to Run Bandit Tests Instead of A/B Tests
- Link: Split testing vs Multi-Armed Bandits

Regret of pure exploration

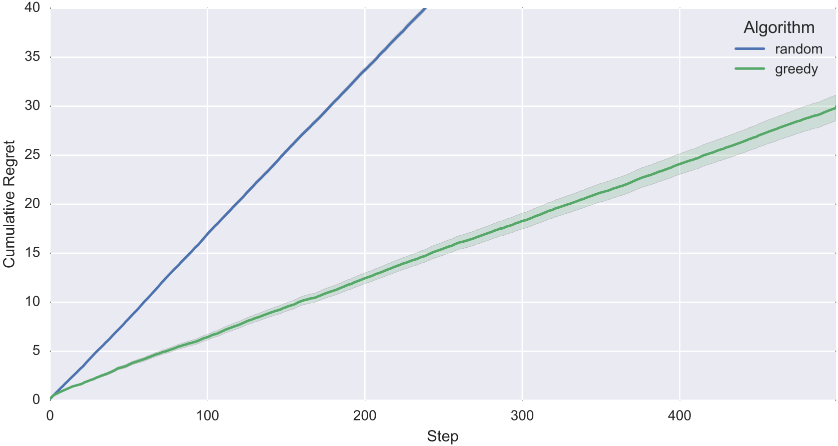


- If an algorithm always explores, it will have linear total regret

Greedy

- Pure exploitation
- You always choose the action with the highest $\hat{Q}(a)$
- Can you see a problem with this?
- Let's try it out

Regret of greedy

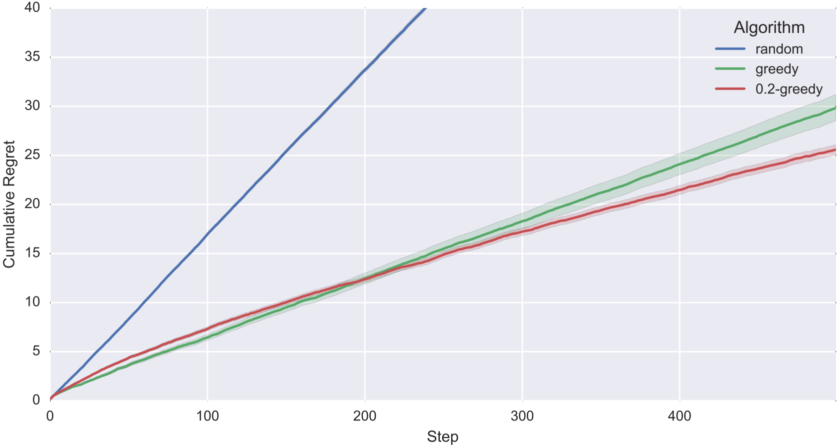


- If an algorithm always explores, it will have linear total regret
- If an algorithm never explores, it will have linear total regret

ϵ-GREEDY

- You set a small probability ϵ with which you act randomly
- The rest of the time $(1 - \epsilon)$ you choose the best action
- This is a very common (but inefficient) setup
- What is the optimal ϵ?

Regret of ϵ-greedy



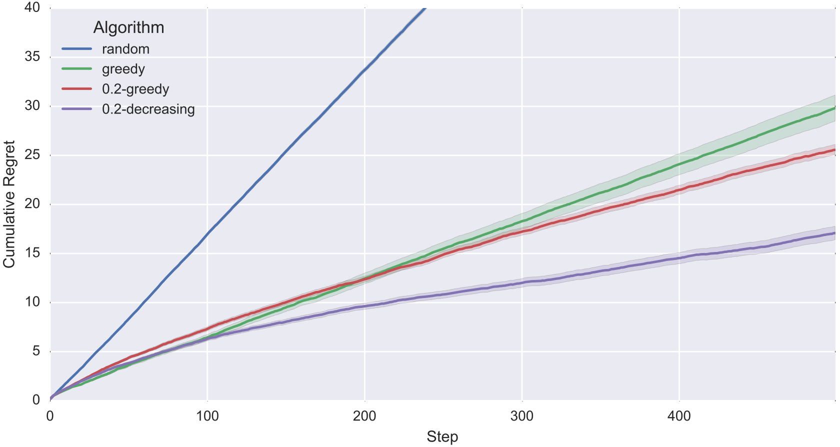
- If ϵ is constant, ϵ-greedy has asymptotic linear total regret

Decaying ϵ-greedy

- Same as ϵ-greedy, but now you decrease ϵ as you choose actions
- E.g.: We do

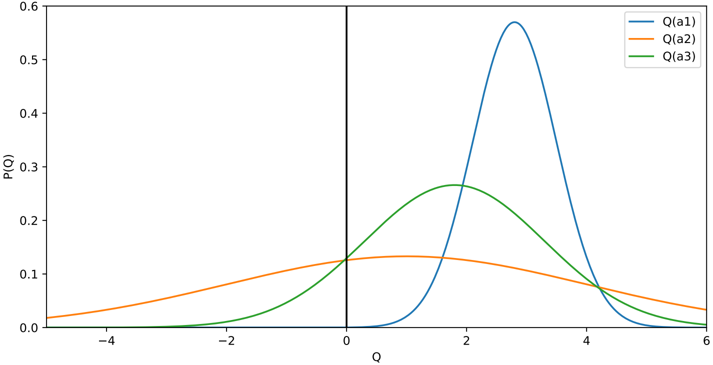
```
e *= 0.99
```

Regret of decaying ϵ-greedy



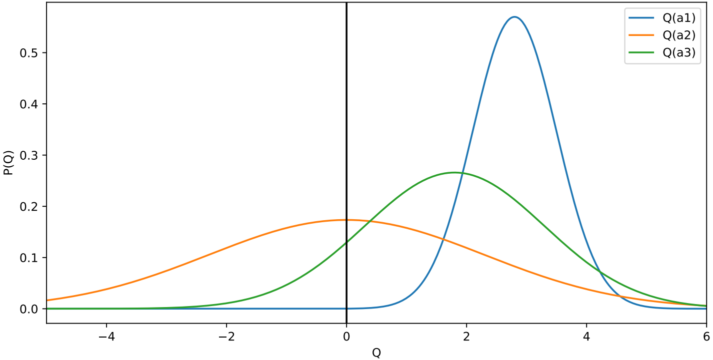
- Decaying ϵ-greedy has logarithmic asymptotic total regret.

Optimism in the face of uncertainty (1)



- You should try actions with highly uncertain outcomes
  - You believe the best action is the one you haven't explored enough
  - Choose the one that has the most potential to be best

Optimism in the face of uncertainty (2)



- We're more certain about a_2
- More likely to pick another action
- Until we find the best one, a^*

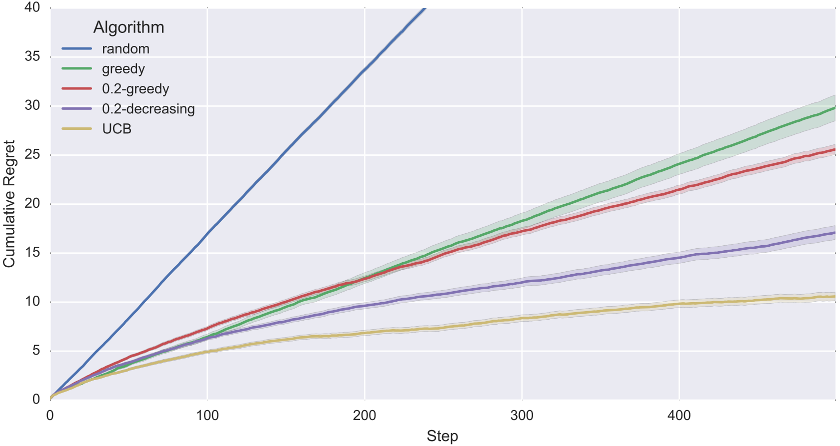
Upper Confidence Bounds (1)

- So far we've estimated $\hat{Q}(a)$, the expected reward for a given action
- We will now add something that characterises how big the tail of the distribution is, $U(a)$, so that $Q(a) \leq \hat{Q}(a) + U(a)$
- Then, we can pick the action with the highest UCB
  - $a^* = \arg\max_{a \in A} \left(\hat{Q}(a) + U(a)\right)$

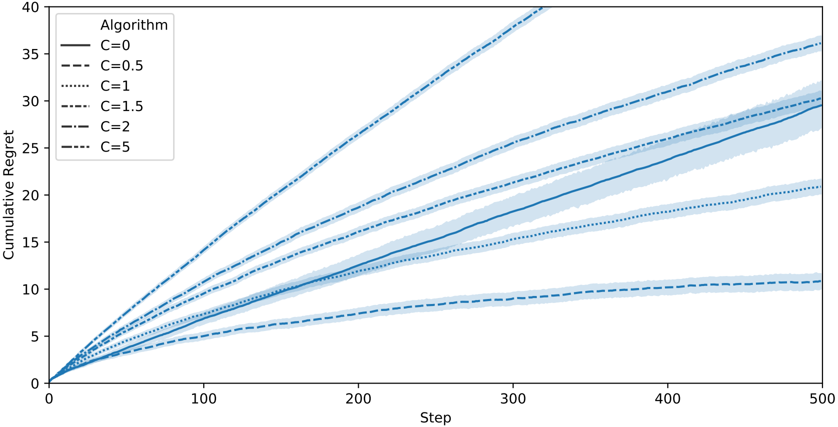
Upper Confidence Bounds (2)

- $UCB1(a) = \hat{Q}(a) + C\sqrt{\frac{log(t)}{N_t(a)}}$
- $N_t(a)$ is the times action a was executed
- t is the current timepoint/time
- $C \in [0, \inf]$ is a constant — I set it to 0.5 for the plots below
  - Can you guess what the effect of C is?

Regret of upper confidence bounds



Influence of C in the Regret of UCB algorithm



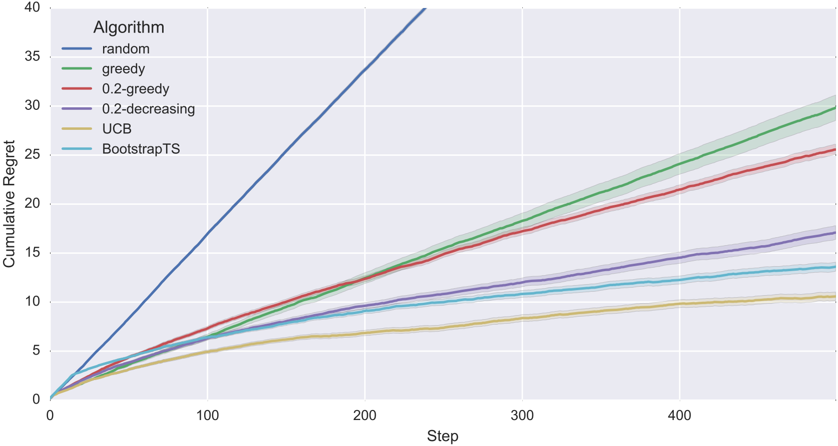
Bootstrap Thompson Sampling

- What if you could take bootstrap samples of the action rewards that we have collected?
- You would have incorporated the uncertainty within your bootstrap samples
- If you have a large number of bootstrap samples, you have a distribution over possible $\hat{Q}(a)$
- Sample from this distribution
- This is a version of **probability matching** (i.e., selecting an action according to the probability that it is the optimal action)
  - $\pi(a) = P[\hat{Q}(a) > \hat{Q}(a'), \quad \forall a' \in A]$

Priors

- You can get stuck here as well in a sub-optimal action (like greedy)
- Add some pseudo-rewards
- Or act randomly a bit

Regret of Bootstrap Thomson Sampling



The Switching Bandit problem

- What if the rewards change?
- Because people are bored of your e-mails
  - They talk to each other
  - Out of fashion
- You might want to have continuous adaptation
- Keeping all values and finding $\hat{Q}(a)$ is expensive
  - What happens in e-mail 1000? And e-mail 100K?

Incremental calculation of the mean

$\mathrm{v_t}$ can be the reward or the sum of rewards you got at different steps

$$\hat{Q}_{t+1}(a) = \hat{Q}_t(a) + \frac{\overbrace{\mathrm{v_t} - \hat{Q}_t(a)}^{\textbf{Error}}}{t}$$

$$\hat{Q}_{t+1}(a) = \hat{Q}_t(a) + \frac{1}{t}\overbrace{\left[\mathrm{v_t} - \hat{Q}_t(a)\right]}^{\textbf{Error}}$$

$$\hat{Q}_{t+1}(a) = \hat{Q}_t(a) + \alpha\left[\mathrm{v_t} - \hat{Q}_t(a)\right]$$

Incremental bootstrap

Oza, Nikunj C., and Russell, Stuart, “Online bagging and boosting.” 2005 IEEE International Conference on Systems, Man and Cybernetics, Vol. 3, 2005.

The sequential case

- What if you are to take a series of actions?
- Surely your current action depends on your future (and your past) actions
- Hence there is going to be a change in the distribution of rewards
  - Induced by the experimenter

Example e-mail campaign

- First e-mail
  - "Please buy this product"
- Second e-mail
  - "Will you buy the add-on?"
- Third e-mail
  - "Let us service your product"
- You want to maximise your rewards
- Creates a tree of possible actions

Tree

- Let's draw the tree of the above example
  - Three different actions for each "state"
- What do you observe?

Introducing states

- $s \in S$ can be used to differentiate between different "states", conditioning π, V and Q values on states
- $\pi(s, a)$, $V(s)$, $Q(s, a)$
- e.g., in the example above, we have $Q(\mathit{firstemail}, \mathit{emailtypeA})$
- Let's write the rest of the states, the policies, V and Q-Values

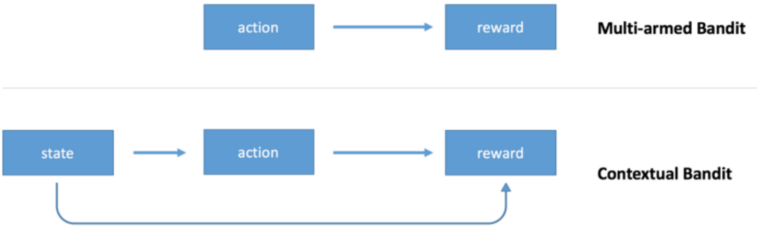
Equilibria

- We will discuss (very) briefly the notion of equilibria
  - Imagine you are putting up large advert banners on your website
  - They hide content
  - User can click on the top right corner and quit the banner
- Where should you put the banner?
- How often should the banner pop-up?

Adversarial bandits

- Most bandits we discussed until now assume the environment is indifferent
- i.e. the user will click in the link if she thinks it is interesting for her to click
- But quite often, people are annoyed by your efforts - so they will try to "adapt" around you
  - Close the advert super-fast without thinking
  - or use an ad blocker!
- Solution - put the advert in random places
  - Mixed policies
- Exp3 - but not now

Contextual Bandits



Source: `https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0`

The contextual bandit extends the MAB model by making the decision conditional on the state of the environment.

The contextual bandit is a tuple $< A, S, R >$

The Contextual Bandit Problem

- Repeat:
  1. Learner presented with context
  2. Learner chooses an action
  3. Learner observes reward for the chosen action
- Goal: Learn a policy that maximises our rewards
- Issues:
  - Classic exploitation vs explore dilemma
  - Plus we also need to learn to use the context effectively
    - Many actions available
    - May not see the same context twice — need to generalise
  - Selection bias: explore while exploiting (i.e., trying to maximise reward), so our data will be skewed

Rethinking states

- We've seen states as black boxes
  - They can only be enumerated (i.e. s_0, $s_1 \ldots$)
- What if a state could be decomposed into a set of features? (i.e., *context*)
  - *Context* is information about the user
  - $sex, age, married, job$...
- Learner needs a policy to select the best action for the given context
- Highly reminiscent of supervised learning
  - We are given features, we would like to predict an outcome
  - **What is our outcome?**

Combining states and actions

- So you now have features that you can encode
- Various encoding strategies
  - One regressor per action
  - A single regressor with encoded actions
- What could be a problem if you don't have separate regressors for each action?

Example

- Our policy maps our context into an action
- Note that in the MAB we had *policy()*
- Before learning, we need to assume a form of policy (e.g., decision tree)

E.g.,

```
def policy(sex, age):
    ''' Returns an action for the given context '''
    if sex == 'male':
        return 1
    elif age >= 45:
        return 0
    else:
        return 2


action = policy('female', 30)
print("Reward", rewards[action]())
```

ϵ-greedy and ϵ-decreasing

- Set ϵ to some small value
- Keep decreasing...
- Very popular because of its simplicity
- You need to be smart about your decreasing schedule
    - Possibly set some lower bound

Bootstrap Thomson Sampling

- Get a bootstrap sample of all your data
- Learn a regressor
- Act greedily using the regressor you learned
- Repeat

Conclusion

- First hit on bandits
- Super-exciting research area
- Used quite a bit on website optimisation and recommender systems
- We will delve deeper in the adversarial case in the future
- Again, the bootstrap saves the day