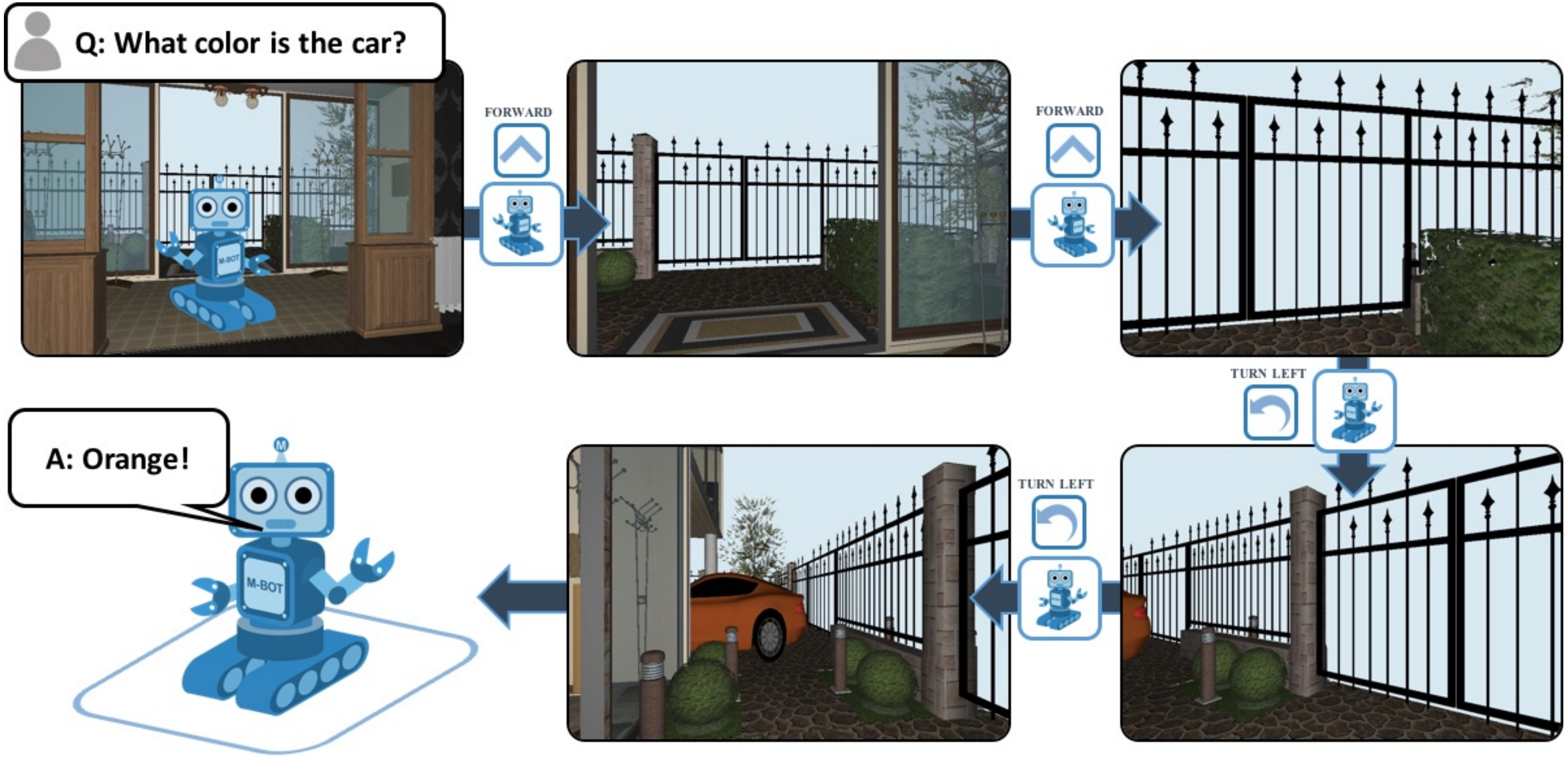
A glance at Reinforcement Learning. A2C.

Workshop by Oleksandr Maksymets

Research Engineer in Facebook AI Research

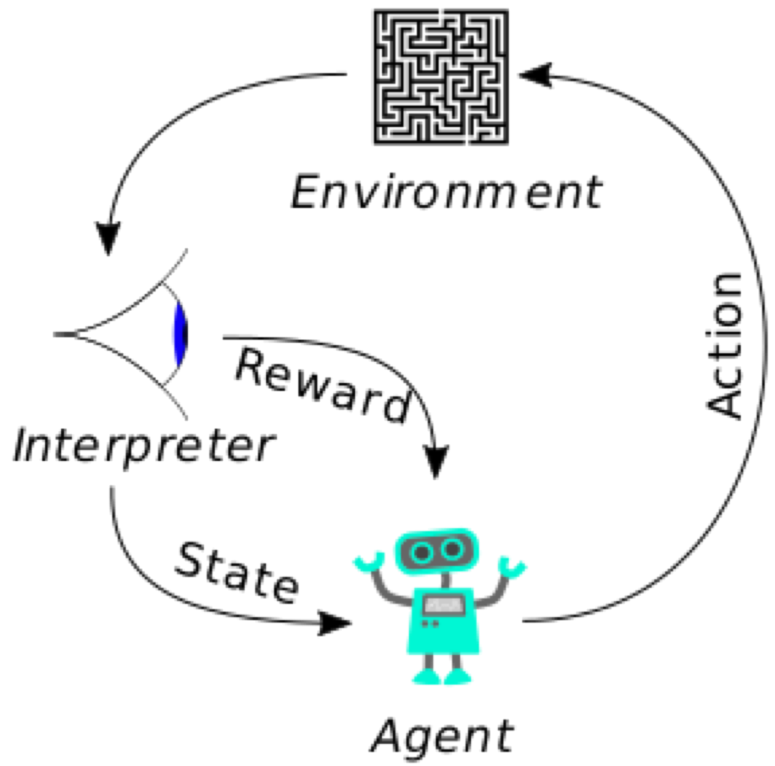
A-STAR team: Agents that See, Talk, Act, and Reason

My focus in FAIR:
Embodied Question Answering



Type of Deep Learning

- Supervised learning
  - classification, regression
- Unsupervised learning
  - clustering
- Reinforcement learning
  - learn from interaction w/ environment to achieve a goal



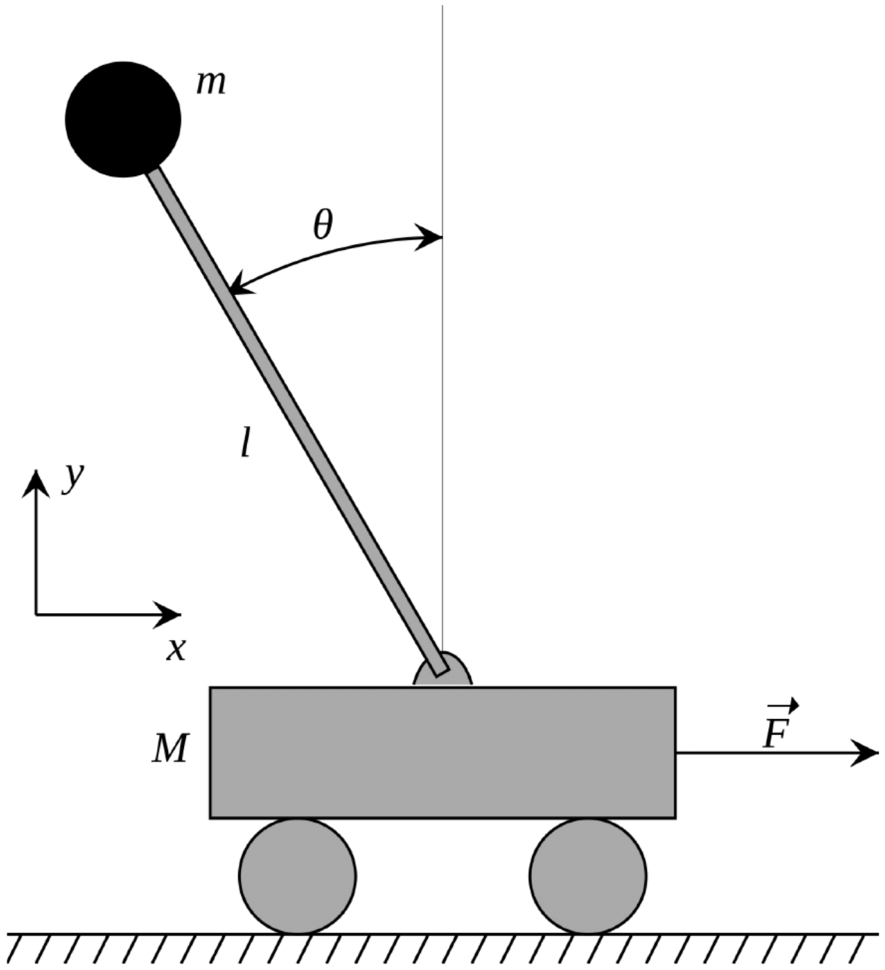
Cart-Pole Problem

Objective: balance a pole on top of a movable cart

State: angle, angular speed, position, horizontal velocity

Action: horizontal force applied on the cart

Reward: 1 at each time step if the pole is upright



Atari games

Objective: win the game

State: raw screen pixels

Action: keypress in the game

Reward: score increase in a the game

This image is CC0 public domain

Robot grasping of objects

Objective: grab an object and move to location

State: raw pixels of RGB camera

Action: engine robot movements

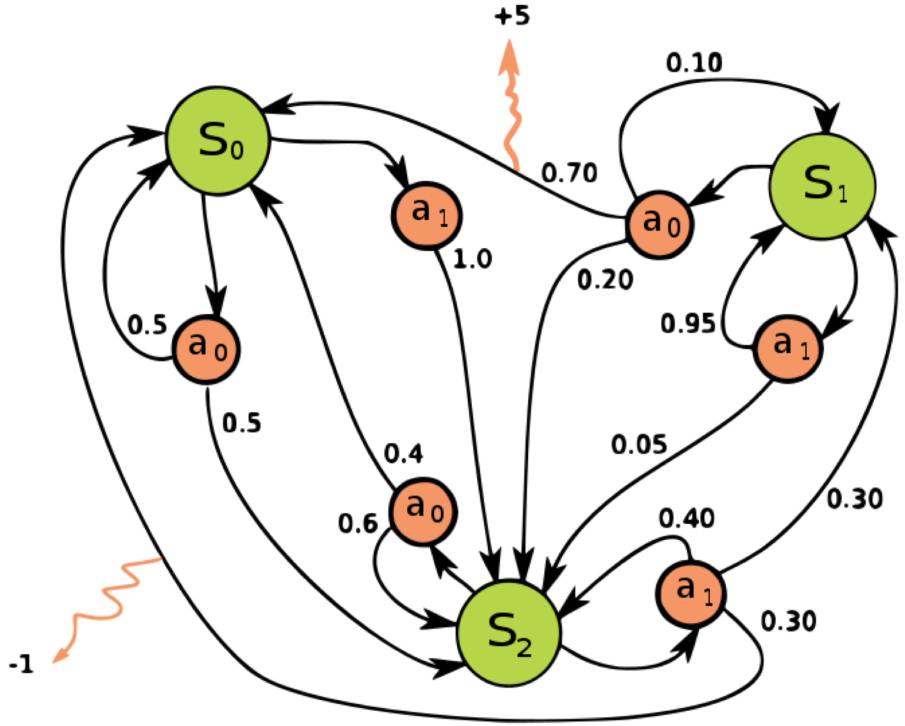
Reward: Positive if object was moved successfully, otherwise negative

Markov Decision Process

Mathematical definition of the RL problem

- set of **states** S, set of **actions** A, **initial state** S_0
- **transition model** P(s,a,s')
- **reward** function R(s, a)
- **goal**: maximize cumulative reward in the long run

- **policy**: agent behavior, mapping from S to A
  π(s) or π(s,a) (deterministic vs. stochastic)



Rewards

- **Episodic tasks**

  Episode finished after N steps

- **Additive rewards**

  $V(s_0, s_1, \ldots) = r(s_0) + r(s_1) + r(s_2) + \ldots$

- **Discounted rewards**

  $V(s_0, s_1, \ldots) = r(s_0) + \gamma * r(s_1) + \gamma^2 * r(s_2) + \ldots$

Value functions

- State value function: $V^{\pi}(s)$
  - expected return when starting in *s* and following π
- State-action value function: $Q^{\pi}(s, a)$
  - expected return when starting in *s*, performing *a,* and following π
- Bellman equation $$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'} \left[r^{a}_{ss'} + \gamma V^{\pi}(s') \right] = \sum_{a} \pi(s,a) Q^{\pi}(s,a)$$

Q-learning idea

- use any policy to estimate Q
- Q directly approximates Q* (Bellman equation)

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

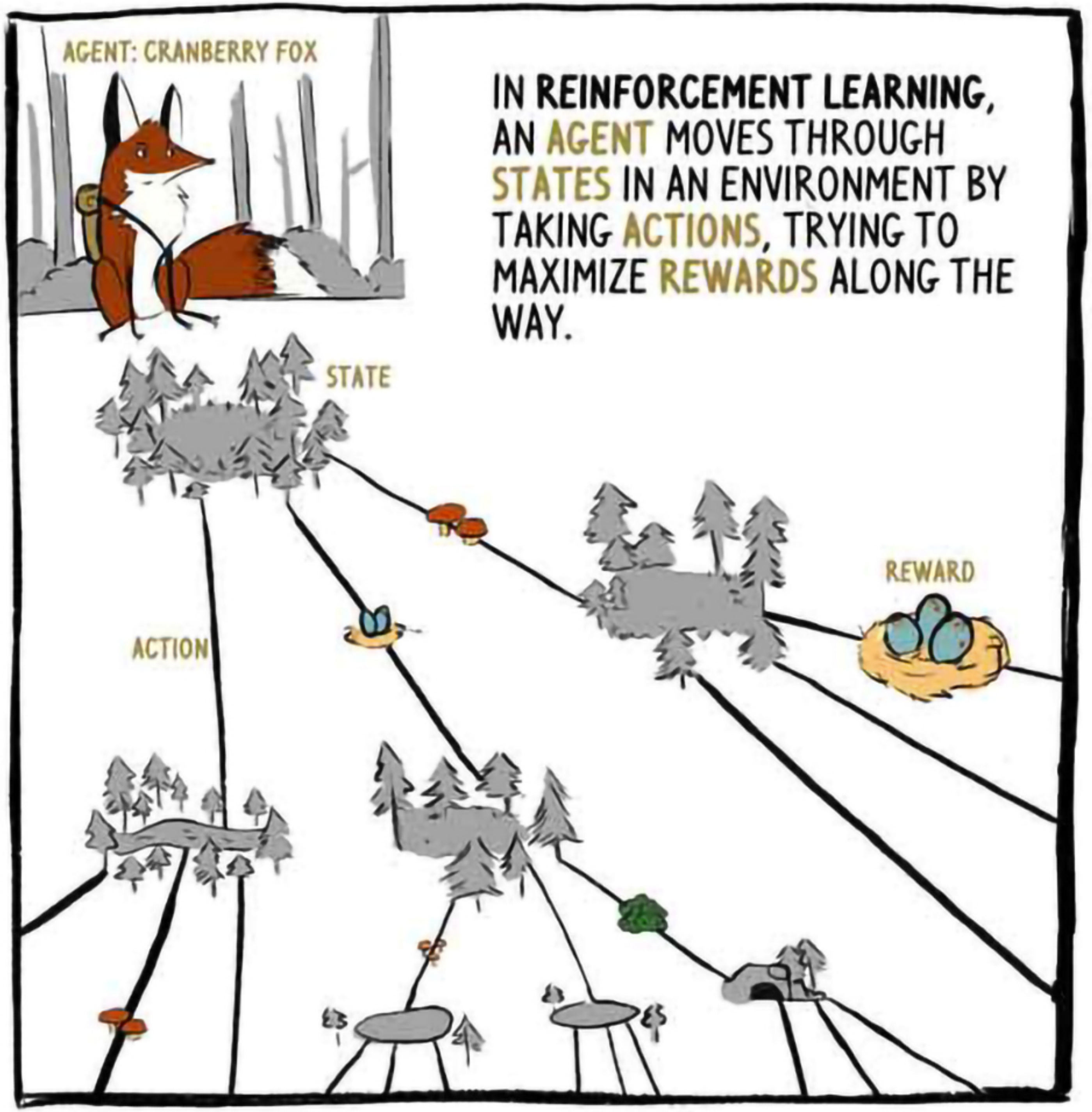
Why advantage actor-critic algorithm

- Considered as strong baseline
- Optimize both Value and Policy that combines benefits of Policy or Value based algorithms
- Has production-ready implementations

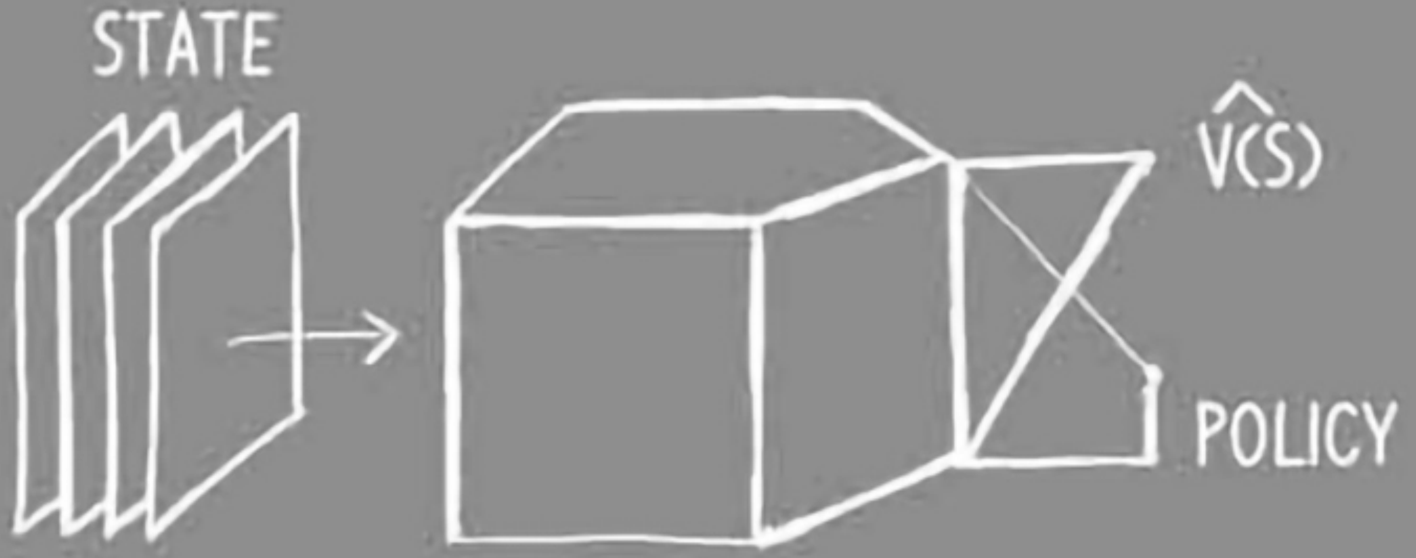
Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t\text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial\left(R - V(s_i;\theta'_v)\right)^2/\partial\theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$


INTUITIVE RL
INTRO TO ADVANTAGE ACTOR CRITIC (A2C)
WRITTEN BY RUDY GILMAN
ART BY KATHERINE WANG


AGENT: CRANBERRY FOX
IN REINFORCEMENT LEARNING, AN AGENT MOVES THROUGH STATES IN AN ENVIRONMENT BY TAKING ACTIONS, TRYING TO MAXIMIZE REWARDS ALONG THE WAY.
STATE
REWARD
ACTION


A2CS TAKE IN A STATE—SENSORY INPUTS IN CRANBERRY'S CASE—AND GENERATE TWO OUTPUTS:
1) AN ESTIMATE OF HOW MANY REWARDS THEY EXPECT TO GET FROM THAT POINT ONWARDS, THE STATE VALUE.
2) A RECOMMENDATION OF WHAT ACTION TO TAKE, THE POLICY
THE "CRITIC"
WOW, WHAT A WONDERFUL GLEN! THIS WILL BE A FRUITFUL DAY OF FORAGING. I BET I'LL GATHER 20 REWARD POINTS BEFORE SUNSET TODAY.
$\hat{V(S)} = 20$
THE "ACTOR"
THOSE FLOWERS LOOK PRETTY, I'M FEELING DRAWN TOWARDS "A"...
A: 80%
B: 10%
C: 10%
A
B
C



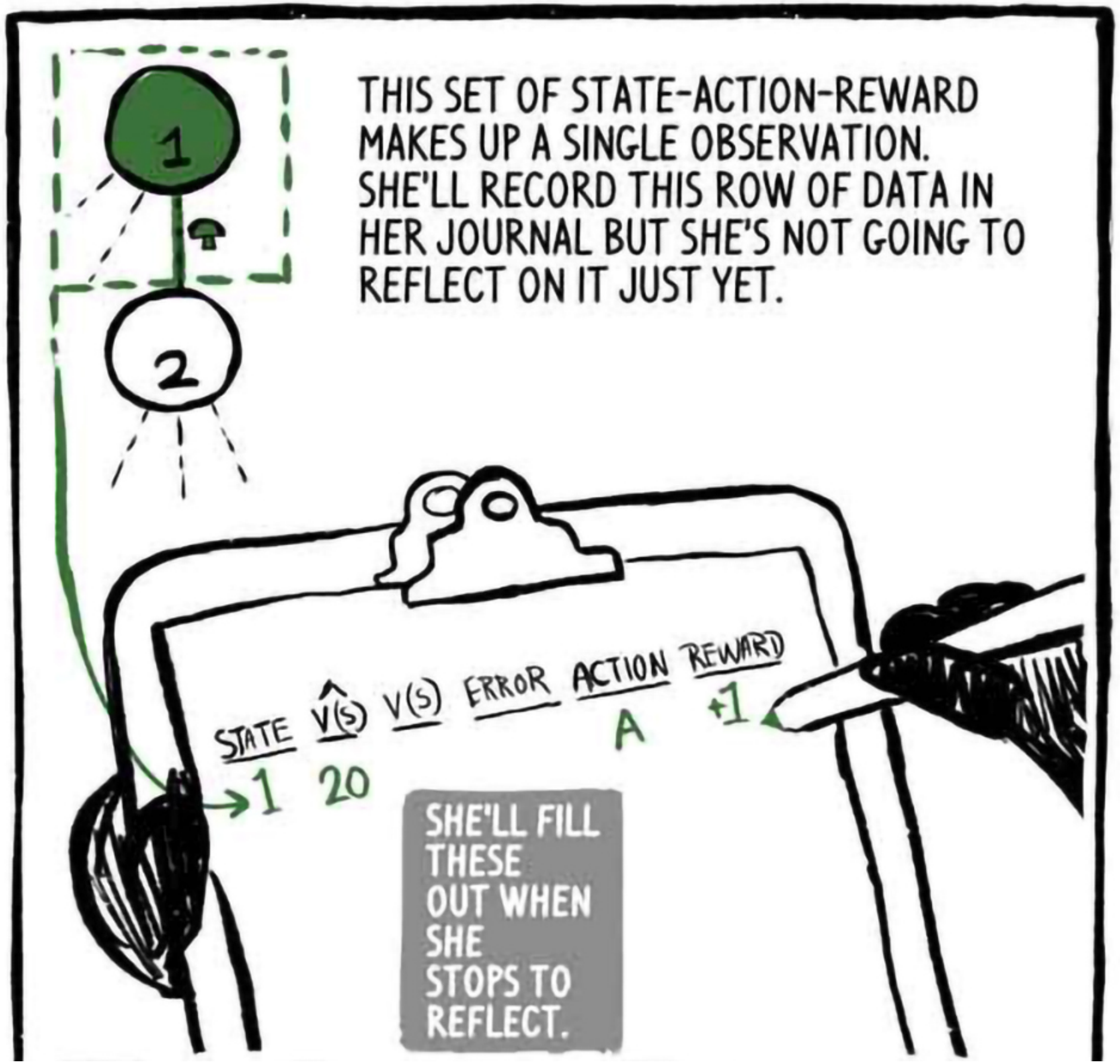
A DEEP RL MODEL IS AN INPUT-OUTPUT MAPPING MACHINE JUST LIKE ANY OTHER NN CLASSIFICATION OR REGRESSION MODEL. INSTEAD OF MAPPING PICTURES OR TEXT TO CATEGORIES, A DEEP RL MODEL MAPS STATES TO ACTIONS AND/OR STATES TO STATE VALUES. AN A2C DOES BOTH!

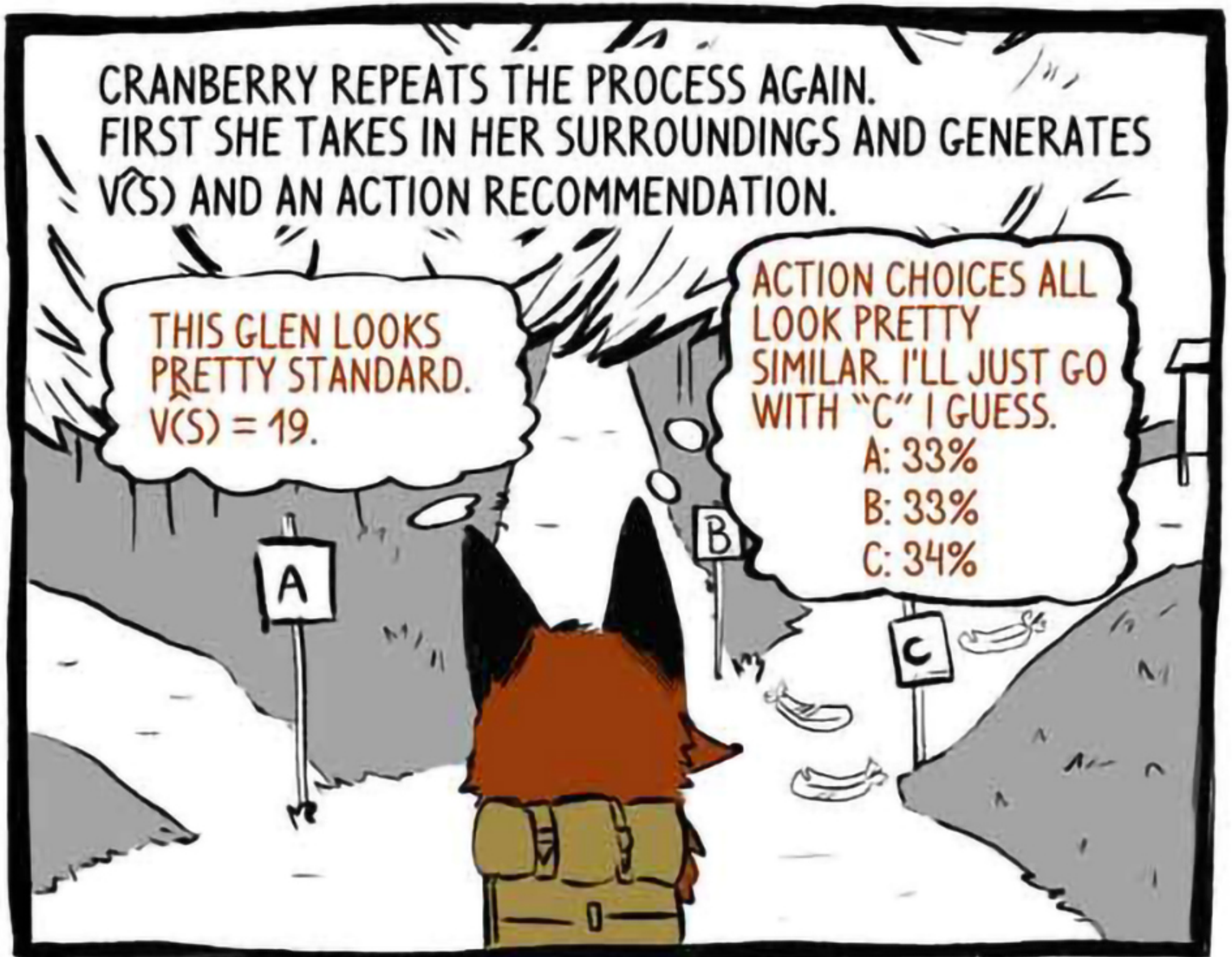
Model definition in Pytorch

```
class ActorCritic(nn.Module):
    def __init__(self):
        super(ActorCritic, self).__init__()
        self.linear1 = nn.Linear(N_INPUTS, 64)
        self.linear2 = nn.Linear(64, 128)
        self.linear3 = nn.Linear(128, 64)

        self.actor = nn.Linear(64, N_ACTIONS)
        self.critic = nn.Linear(64, 1)

    # In a PyTorch model, you only have to defi
    def forward(self, x):
        x = self.linear1(x)
        x = F.relu(x)
        x = self.linear2(x)
        x = F.relu(x)
        x = self.linear3(x)
        x = F.relu(x)
        return x
```

```
# Only the Actor head
def get_action_probs(self, x):
    x = self(x)
    action_probs = F.softmax(self.actor(x))
    return action_probs

# Only the Critic head
def get_state_value(self, x):
    x = self(x)
    state_value = self.critic(x)
    return state_value

# Both heads
def evaluate_actions(self, x):
    x = self(x)
    action_probs = F.softmax(self.actor(x))
    state_values = self.critic(x)
    return action_probs, state_values
```

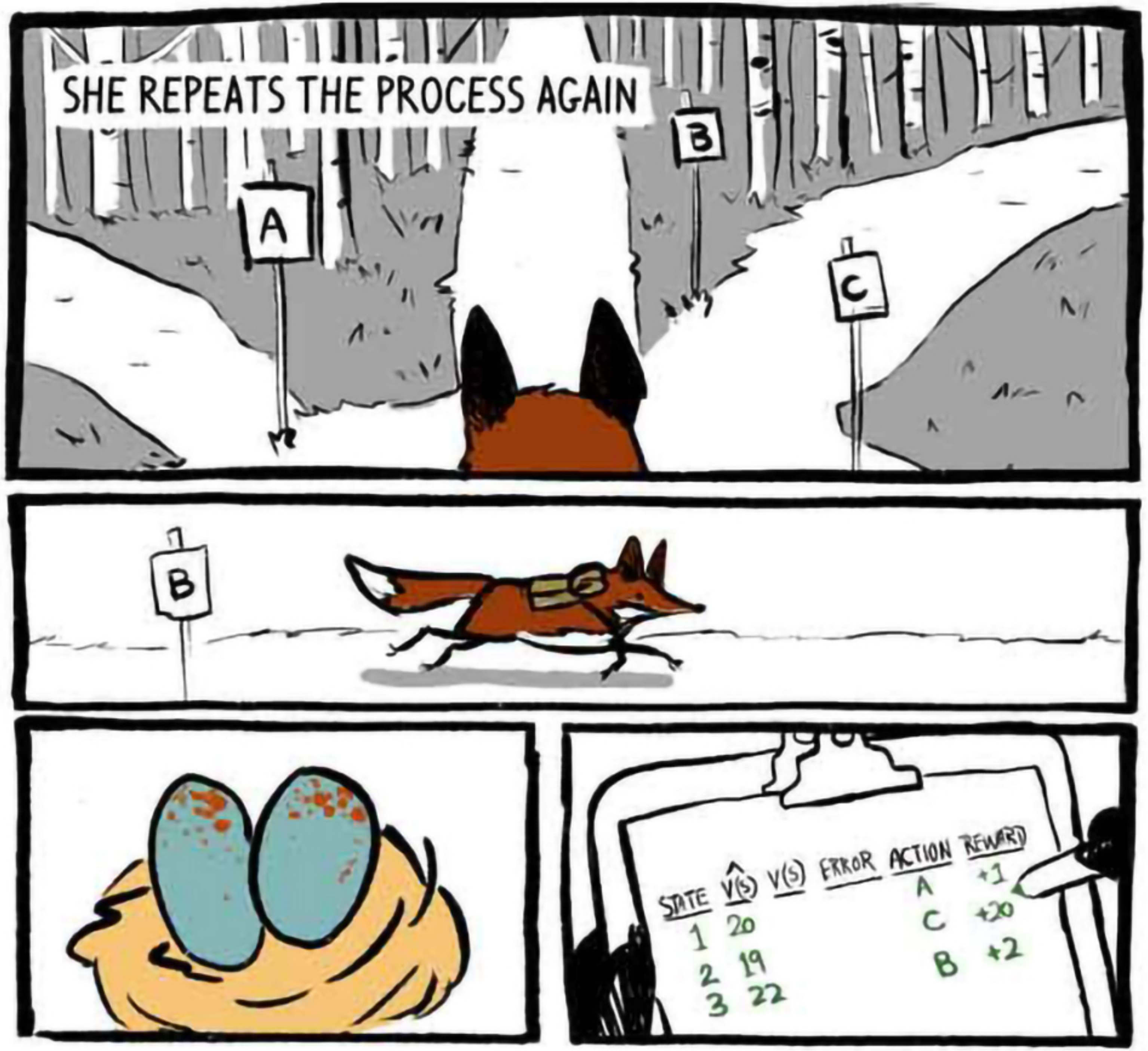

A
+1
SCORE!


THIS SET OF STATE-ACTION-REWARD MAKES UP A SINGLE OBSERVATION. SHE'LL RECORD THIS ROW OF DATA IN HER JOURNAL BUT SHE'S NOT GOING TO REFLECT ON IT JUST YET.
1
2
STATE
$\hat{V}(S)$
V(S)
ERROR
ACTION
REWARD
1
20
A
+1
SHE'LL FILL THESE OUT WHEN SHE STOPS TO REFLECT.


CRANBERRY REPEATS THE PROCESS AGAIN.
FIRST SHE TAKES IN HER SURROUNDINGS AND GENERATES $\hat{V}(S)$ AND AN ACTION RECOMMENDATION.
THIS GLEN LOOKS PRETTY STANDARD. $\hat{V}(S) = 19$.
ACTION CHOICES ALL LOOK PRETTY SIMILAR. I'LL JUST GO WITH "C" I GUESS.
A: 33%
B: 33%
C: 34%
A
B
C


THEN SHE TAKES AN ACTION
C


RECEIVES A REWARD
+20!!!
STATE V̂(S) V(S) ERROR ACTION REWARD
A +1
1 20 C +20
2 19
AND RECORDS THE OBSERVATION.


SHE REPEATS THE PROCESS AGAIN
A
B
C
B
STATE $\hat{V}(S)$ V(S) ERROR ACTION REWARD
1 20 A +1
2 19 C +20
3 22 B +2

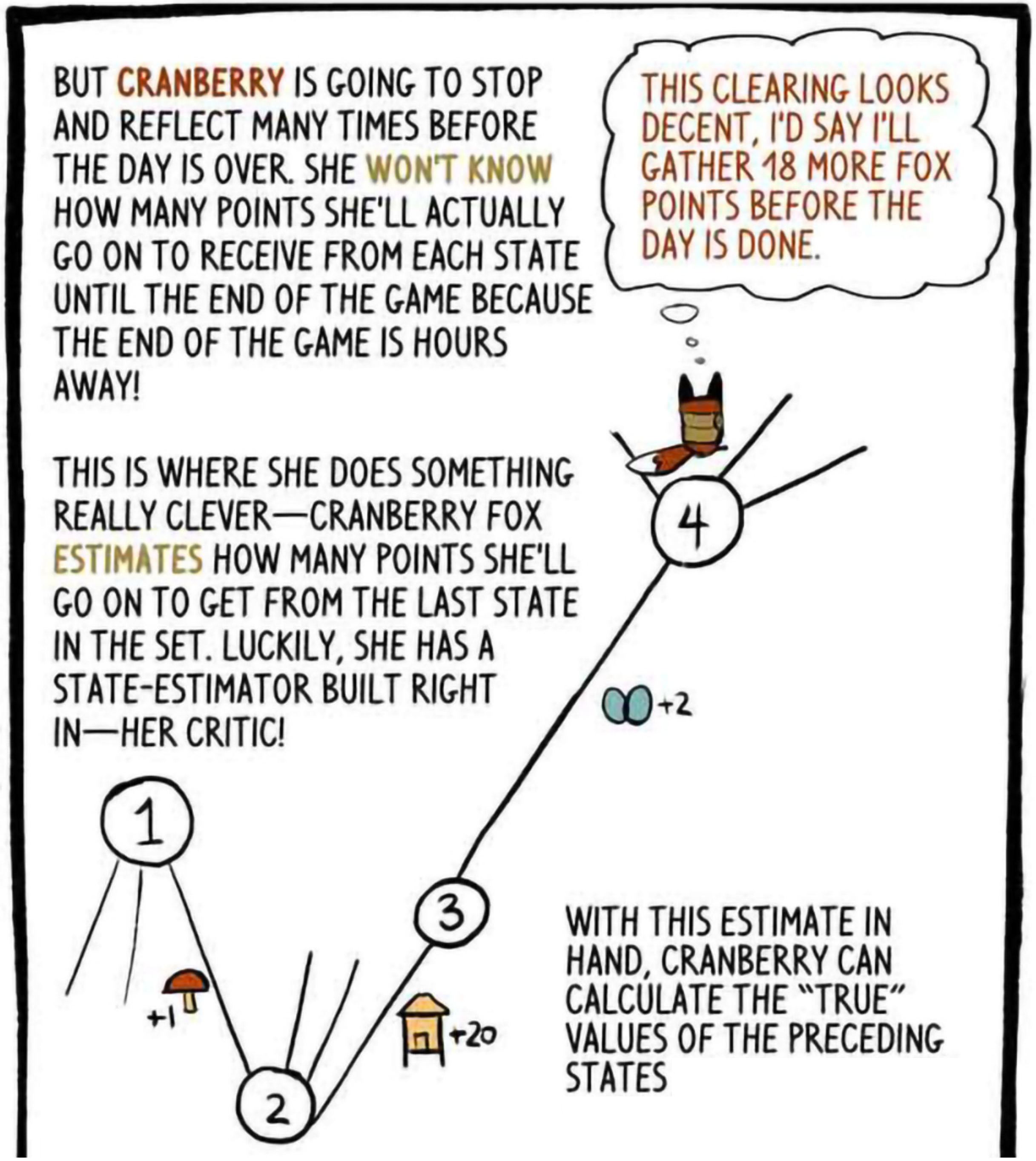
Loop of observation collection

```
state = env.reset()
finished_games = 0

while finished_games < N_GAMES:
    states, actions, rewards, dones = [], [], [], []

    # Gather training data
    for i in range(N_STEPS):
        s = Variable(torch.from_numpy(state).float().unsqueeze(0))

        action_probs = model.get_action_probs(s)
        action = action_probs.multinomial(num_samples=1).data[0][0].item()
        next_state, reward, done, _ = env.step(action)
        states.append(state); actions.append(action); rewards.append(reward); dones.append(done)

        if done: state = env.reset(); finished_games += 1
        else: state = next_state
```

AFTER COLLECTING THREE OBSERVATIONS, CRANBERRY STOPS TO REFLECT.

OTHER FAMILIES OF MODEL WAIT UNTIL THE VERY END OF THE DAY BEFORE REFLECTING (MONTE CARLO) WHILE OTHERS REFLECT AFTER EVERY STEP (ONE-STEPS).

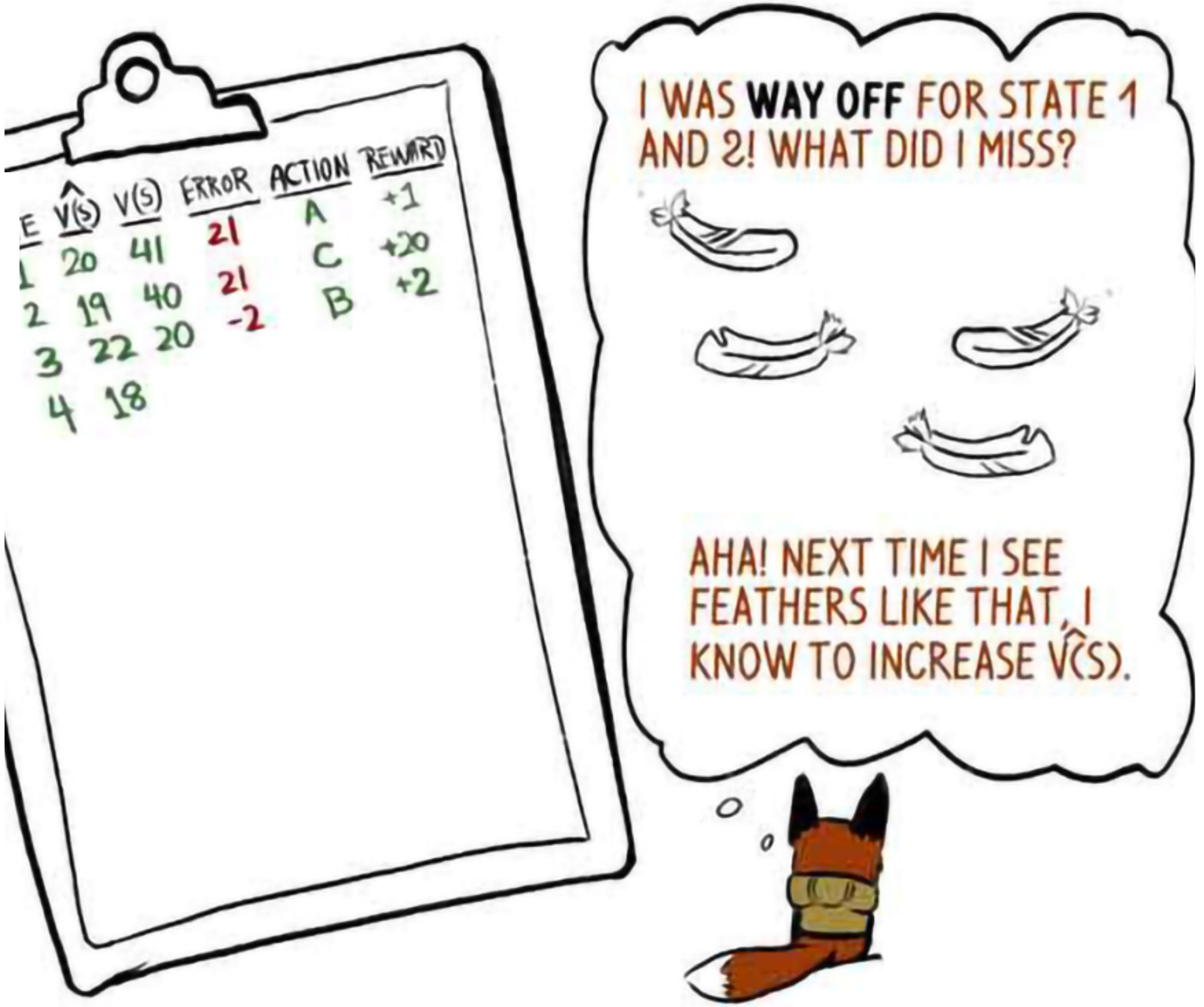
BEFORE SHE CAN TUNE HER INNER CRITIC, CRANBERRY NEEDS TO CALCULATE HOW MANY POINTS SHE WOULD ACTUALLY GO ON TO RECEIVE FROM EACH GIVEN STATE.


BUT CRANBERRY IS GOING TO STOP AND REFLECT MANY TIMES BEFORE THE DAY IS OVER. SHE WON'T KNOW HOW MANY POINTS SHE'LL ACTUALLY GO ON TO RECEIVE FROM EACH STATE UNTIL THE END OF THE GAME BECAUSE THE END OF THE GAME IS HOURS AWAY!
THIS CLEARING LOOKS DECENT, I'D SAY I'LL GATHER 18 MORE FOX POINTS BEFORE THE DAY IS DONE.
THIS IS WHERE SHE DOES SOMETHING REALLY CLEVER—CRANBERRY FOX ESTIMATES HOW MANY POINTS SHE'LL GO ON TO GET FROM THE LAST STATE IN THE SET. LUCKILY, SHE HAS A STATE-ESTIMATOR BUILT RIGHT IN—HER CRITIC!
1
+1
2
+20
3
+2
4
WITH THIS ESTIMATE IN HAND, CRANBERRY CAN CALCULATE THE "TRUE" VALUES OF THE PRECEDING STATES

REWARDS ARE OFTEN DISCOUNTED TO REFLECT THE FACT THAT A REWARD NOW IS BETTER THAN A REWARD IN THE FUTURE. TO KEEP THINGS SIMPLE, CRANBERRY ISN'T DISCOUNTING HER REWARDS.

NOW CRANBERRY CAN GO THROUGH EACH ROW OF DATA AND COMPARE HER STATE-VALUE PREDICTIONS TO THEIR ACTUAL VALUES. SHE USES THE DIFFERENCE BETWEEN THESE NUMBERS TO HONE HER PREDICTION SKILLS. EVEN THREE STEPS INTO THE DAY, CRANBERRY HAS GATHERED VALUABLE EXPERIENCES WORTH REFLECTING ON.

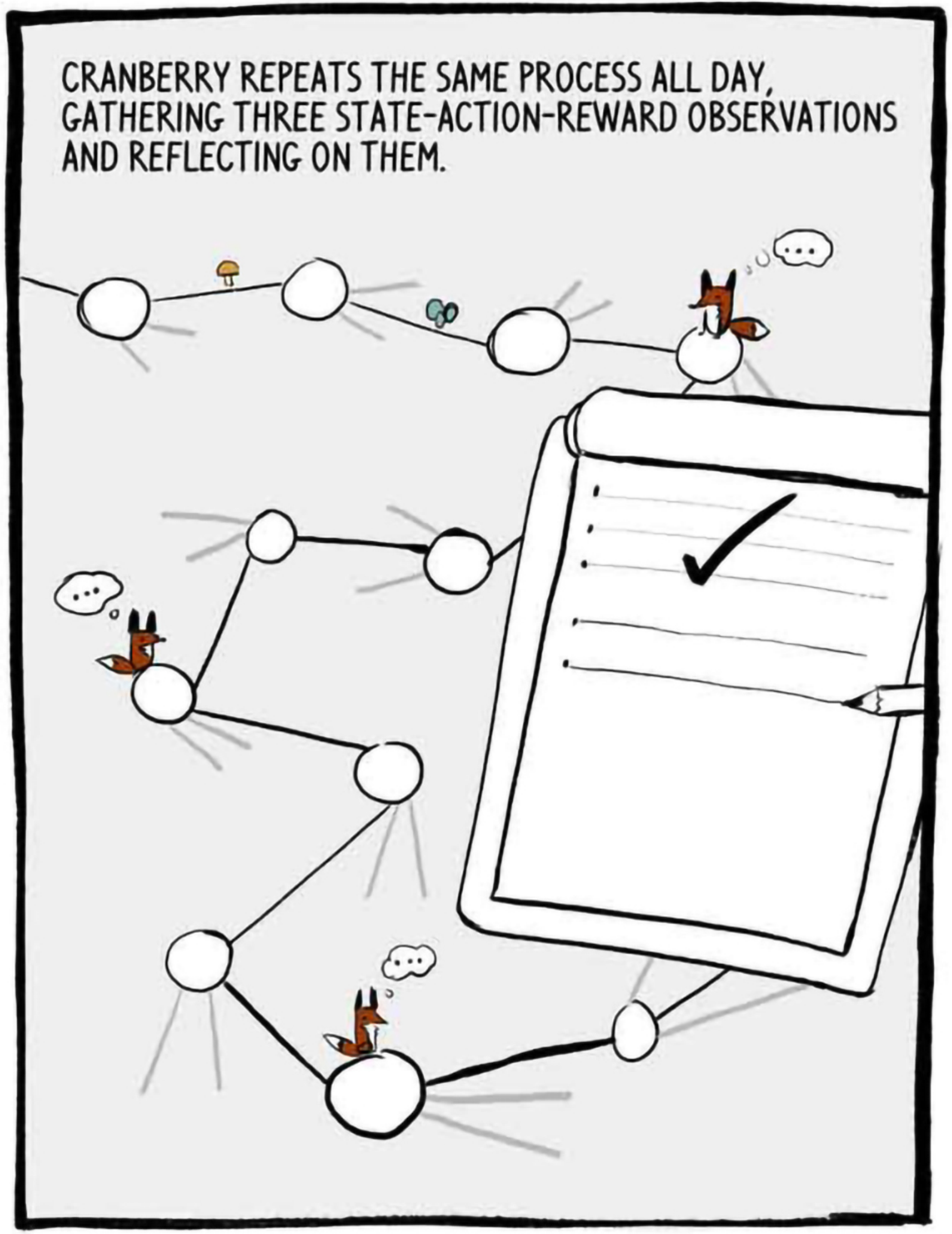
IT MAY SEEM CRAZY THAT CRANBERRY IS ABLE TO USE HER ESTIMATE OF V(S) AS THE GROUND TRUTH TO COMPARE HER OTHER PREDICTIONS AGAINST. BUT ANIMALS (US INCLUDED) DO THIS ALL THE TIME! IF YOU FEEL LIKE THINGS ARE GOING WELL, NO NEED TO WAIT TO REINFORCE THE ACTIONS THAT LED YOU THERE.



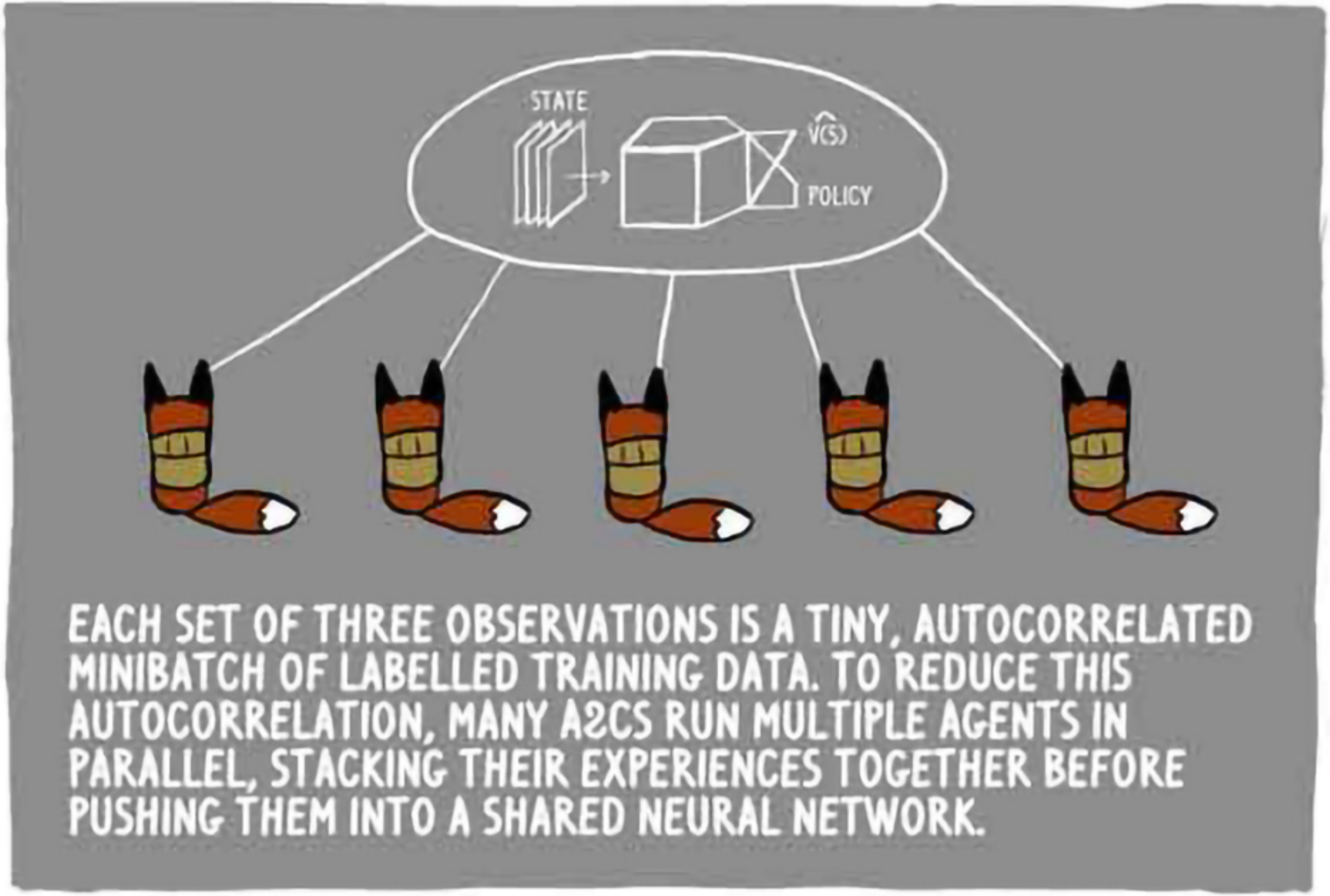
Calculation of actual rewards and reflect/train implementations

```python
def calc_actual_state_values(rewards, dones):
    R = []
    rewards.reverse()

    # If we happen to end the set on a terminal state, set next return to zero
    if dones[-1] == True: next_return = 0

    # If not terminal state, bootstrap v(s) using our critic
    else:
        s = torch.from_numpy(states[-1]).float().unsqueeze(0)
        next_return = model.get_state_value(Variable(s)).data[0][0]

    # Backup from last state to calculate "true" returns for each state in the set
    R.append(next_return)
    dones.reverse()
    for r in range(1, len(rewards)):
        if not dones[r]: this_return = rewards[r] + next_return * GAMMA
        else: this_return = 0
        R.append(this_return)
        next_return = this_return

    R.reverse()
    state_values_true = Variable(torch.FloatTensor(R)).unsqueeze(1)

    return state_values_true
```

```python
def reflect(states, actions, rewards, dones):

    # Calculating the ground truth "labels" as described above
    state_values_true = calc_actual_state_values(rewards, dones)

    s = Variable(torch.FloatTensor(states))
    action_probs, state_values_est = model.evaluate_actions(s)
    action_log_probs = action_probs.log()

    a = Variable(torch.LongTensor(actions).view(-1, 1))
    chosen_action_log_probs = action_log_probs.gather(1, a)

    advantages = state_values_true - state_values_est

    entropy = (action_probs * action_log_probs).sum(1).mean()
    action_gain = (chosen_action_log_probs * advantages).mean()
    value_loss = advantages.pow(2).mean()
    total_loss = value_loss - action_gain - 0.0001 * entropy

    optimizer.zero_grad()
    total_loss.backward()
    nn.utils.clip_grad_norm(model.parameters(), 0.5)
    optimizer.step()
```


CRANBERRY REPEATS THE SAME PROCESS ALL DAY, GATHERING THREE STATE-ACTION-REWARD OBSERVATIONS AND REFLECTING ON THEM.
...
...
...

From A2C to A3C

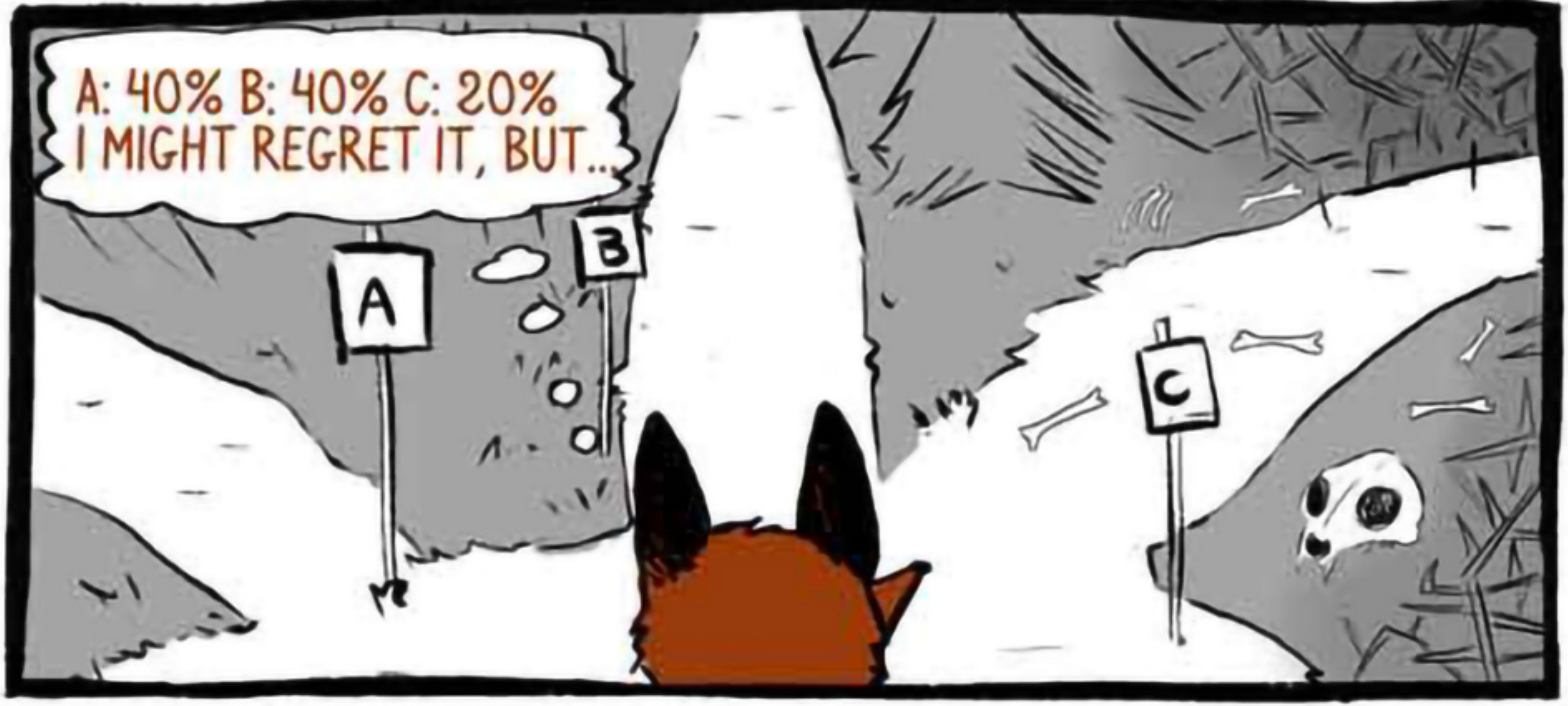


THE DAY IS ALMOST OVER. ONLY TWO STEPS TO GO.

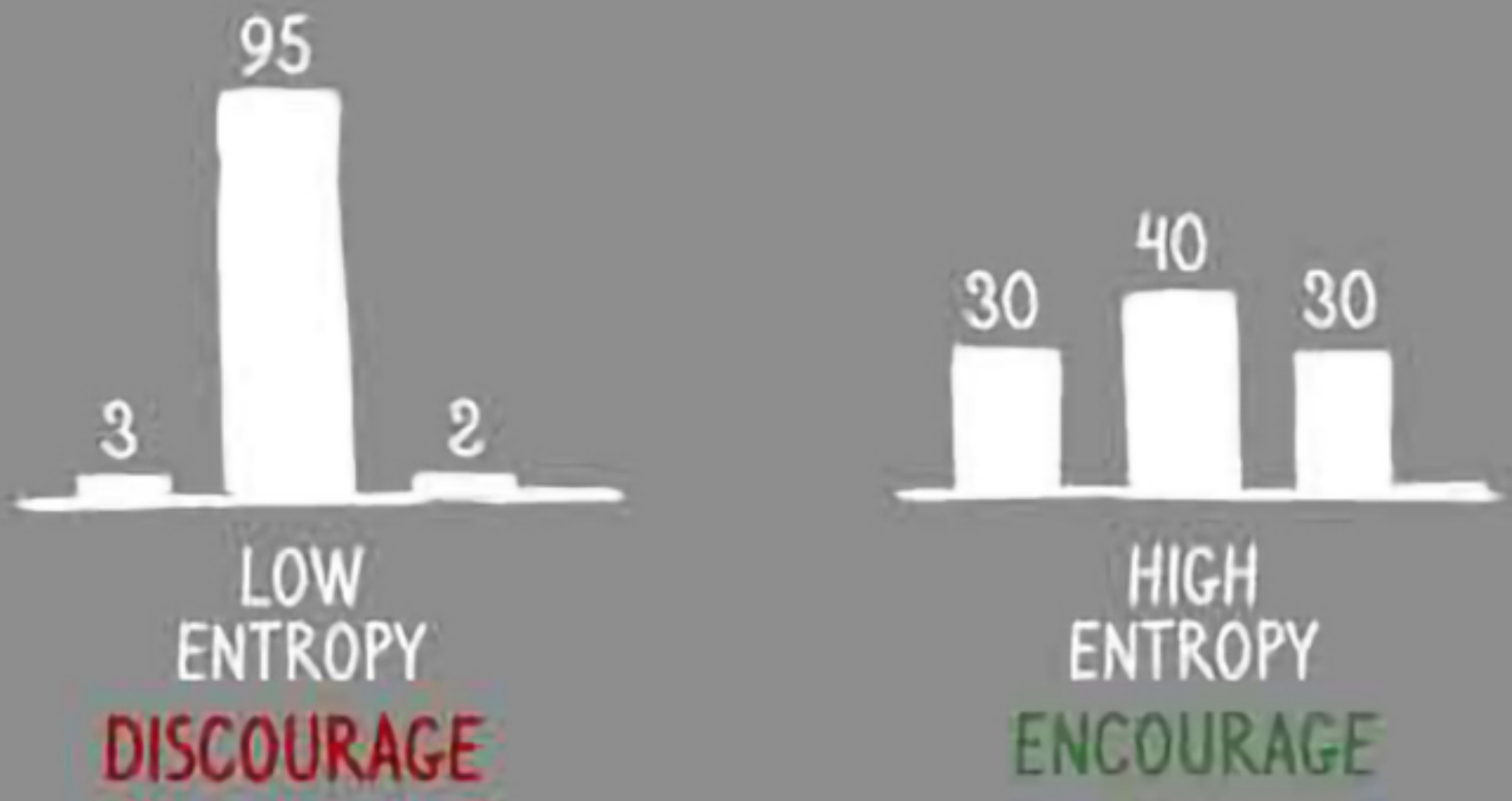
AS WE SAW EARLIER, CRANBERRY'S ACTION RECOMMENDATIONS ARE EXPRESSED AS PERCENTAGE CONFIDENCES ABOUT HER OPTIONS. INSTEAD OF SIMPLY TAKING THE HIGHEST-CONFIDENCE CHOICE, CRANBERRY SAMPLES FROM THIS ACTION DISTRIBUTION. THIS ENSURES SHE DOESN'T ALWAYS SETTLE FOR SAFE BUT POTENTIALLY MEDIOCRE ACTIONS.

Get sampled action from model for current state:

```
action_probs = model.get_action_probs(s)
action = action_probs.multinomial(num_samples=1).data[0][0].item()
```


A: 40% B: 40% C: 20%
I MIGHT REGRET IT, BUT...
A
B
C


SOMETIMES EXPLORING UNKNOWN THINGS
CAN LEAD TO EXCITING NEW DISCOVERIES...
C



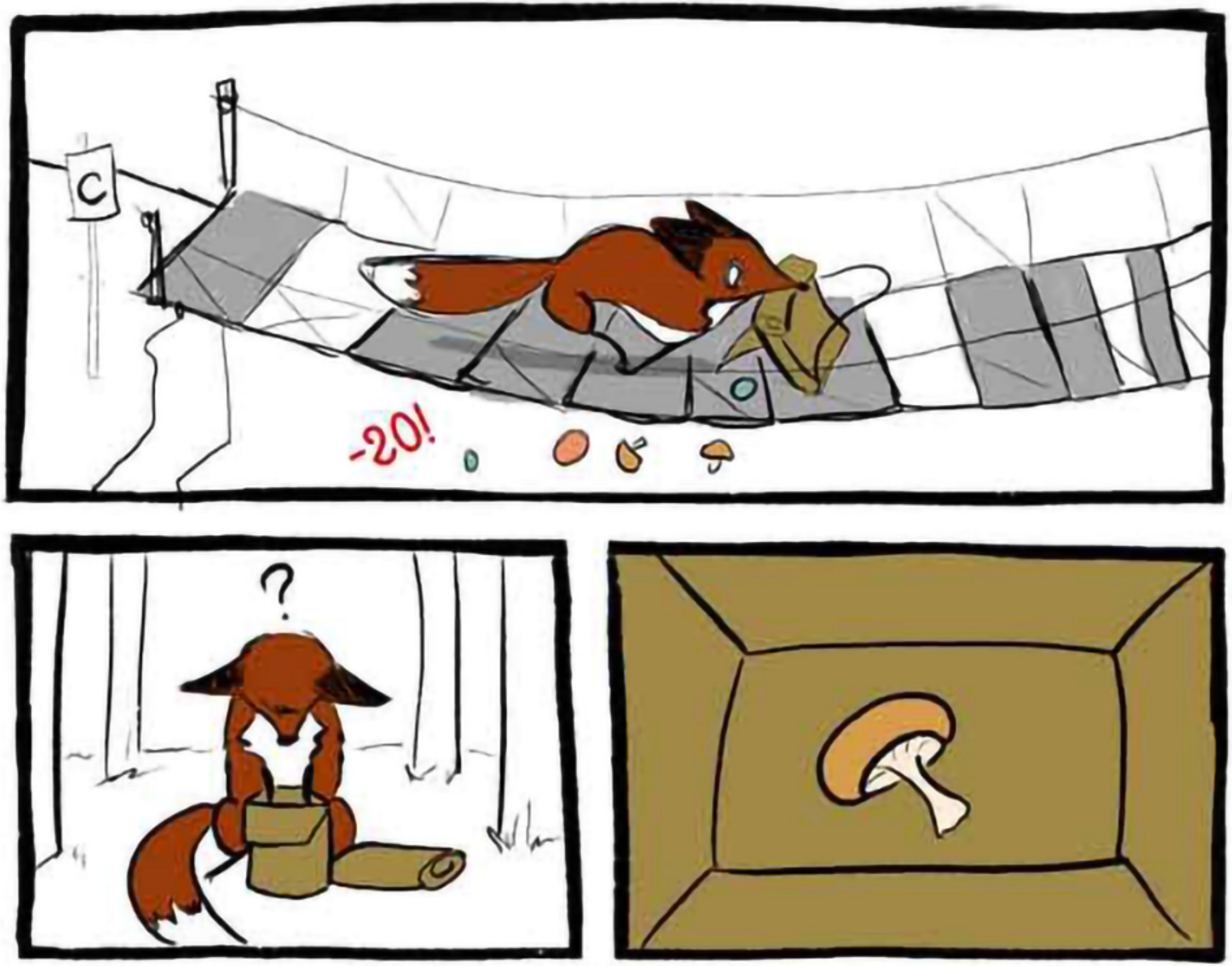
TO FURTHER ENCOURAGE EXPLORATION, A VALUE CALLED ENTROPY IS SUBTRACTED FROM THE LOSS FUNCTION. ENTROPY REFERS TO THE "SPREAD" OF THE ACTION DISTRIBUTION.

Total loss encourages exploration

```
entropy = (action_probs * action_log_probs).sum(1).mean()
action_gain = (chosen_action_log_probs * advantages).mean()
value_loss = advantages.pow(2).mean()
total_loss = value_loss - action_gain - 0.0001 * entropy
```

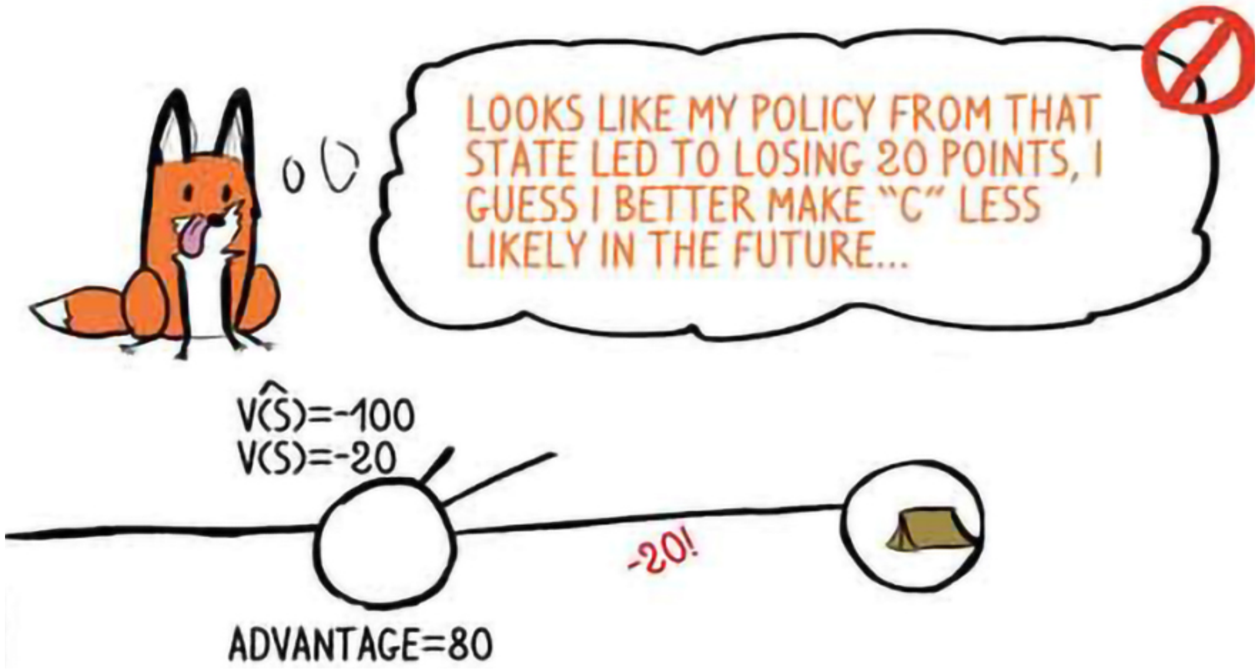

+5!
LOOKS LIKE THE GAMBLE PAID OFF!
OR DID IT?

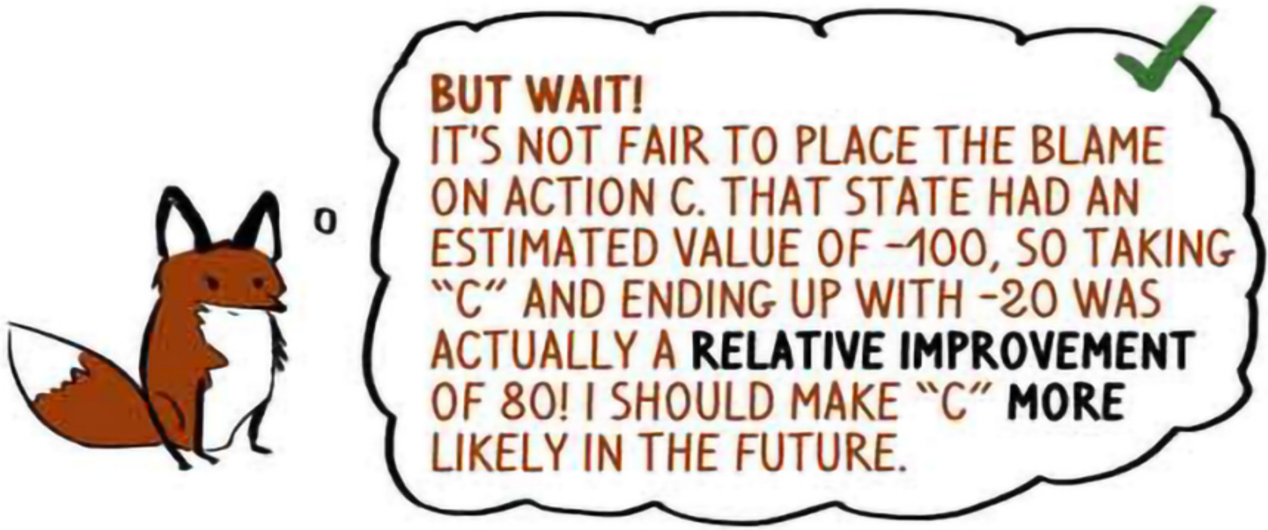
SOMETIMES AN AGENT WILL FIND ITSELF IN A STATE WHERE ALL ACTIONS LEAD TO NEGATIVE OUTCOMES. A2CS, HOWEVER, ARE EXCELLENT AT MAKING THE MOST OF BAD SITUATIONS.


C
-20!
?

WE TALKED ABOUT HOW CRANBERRY TUNES HER INNER CRITIC. BUT HOW DOES SHE ADJUST HER INNER ACTOR? HOW DOES SHE LEARN TO MAKE SUCH REFINED CHOICES?

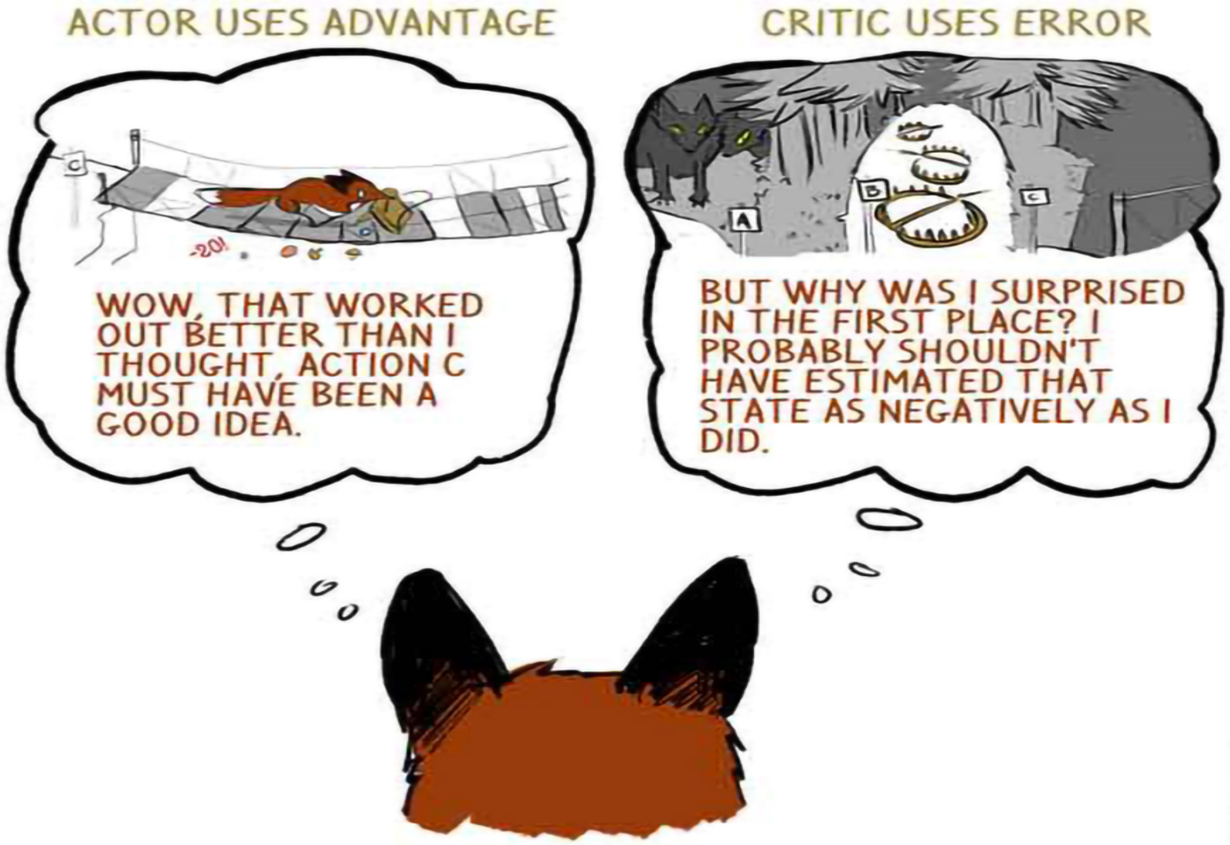
A SIMPLE POLICY GRADIENT FOX WOULD LOOK AT THE ACTUAL RETURNS FOLLOWING AN ACTION AND TUNE HER POLICY TO MAKE GOOD RETURNS MORE LIKELY.





INSTEAD OF TUNING HER POLICY IN RESPONSE TO THE TOTAL RETURNS SHE GOT BY TAKING ACTION C, SHE TUNES HER ACTIONS TO THE RELATIVE RETURNS OF TAKING ACTION C. THIS IS CALLED THE "ADVANTAGE".

WHAT WE CALLED THE ADVANTAGE IS JUST THE ERROR. AS THE ADVANTAGE, CRANBERRY USES IT TO MAKE ACTIONS THAT WERE SURPRISINGLY GOOD MORE LIKELY. AS THE ERROR, SHE USES THE SAME QUANTITY TO NUDGE HER INNER CRITIC TO MAKE BETTER ESTIMATIONS OF STATE VALUES.



NOW WE CAN SHOW HOW TOTAL LOSS IS COMPUTED—THIS IS THE FUNCTION WE MINIMIZE TO IMPROVE OUR MODEL.

TOTAL LOSS = ACTION LOSS + VALUE LOSS – ENTROPY.

NOTICE WE'RE SHOVING GRADIENTS OF THREE QUALITATIVELY DIFFERENT TYPES BACK THROUGH A SINGLE NN. THIS IS EFFICIENT BUT IT CAN MAKE CONVERGENCE MORE DIFFICULT.

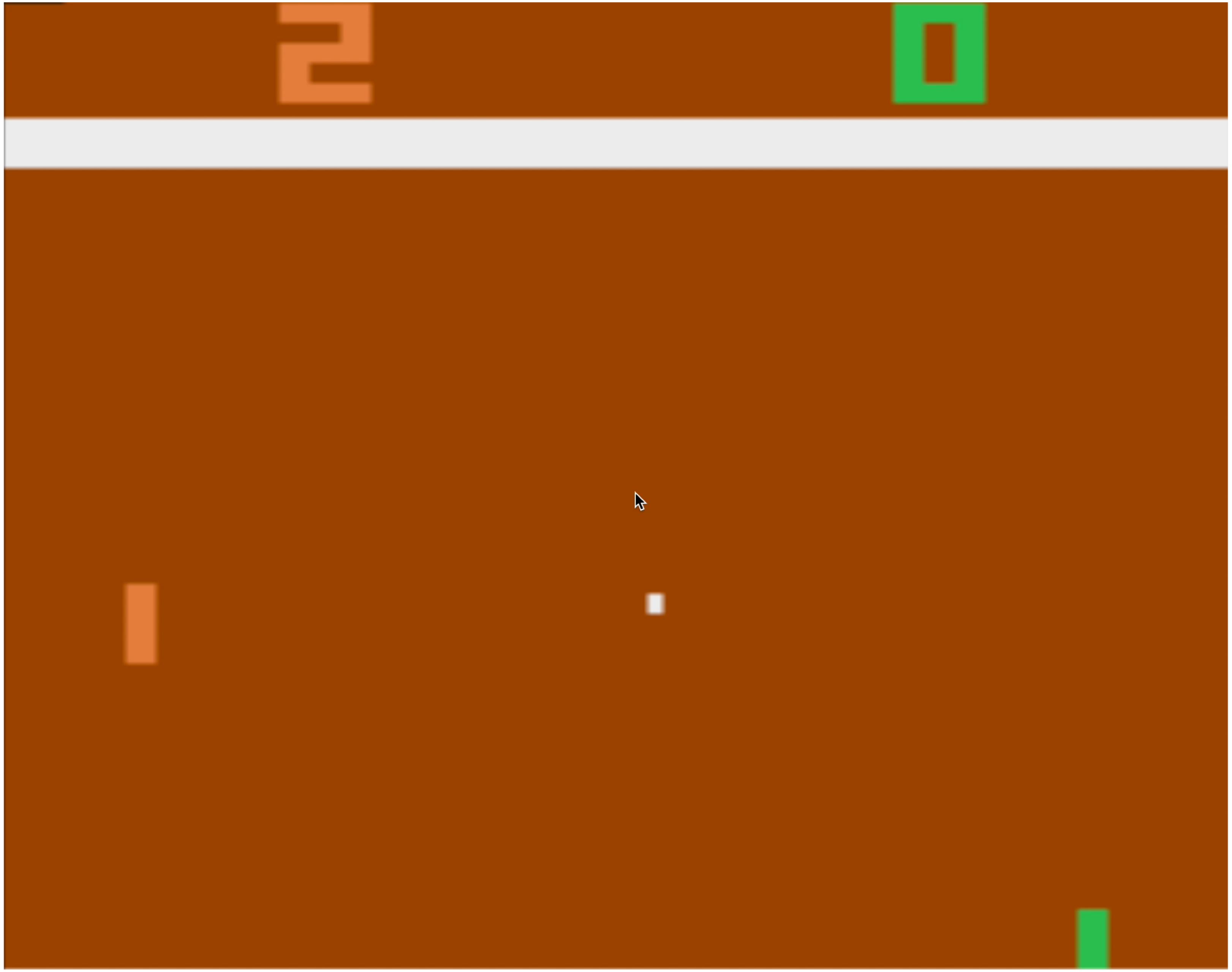
LIKE ALL ANIMALS, AS CRANBERRY MATURES SHE'LL HONE HER ABILITY TO PREDICT STATE VALUES, GAIN INCREASING CONFIDENCE IN HER ACTION CHOICES AND FIND HERSELF SURPRISED LESS OFTEN BY REWARDS.

Evaluation

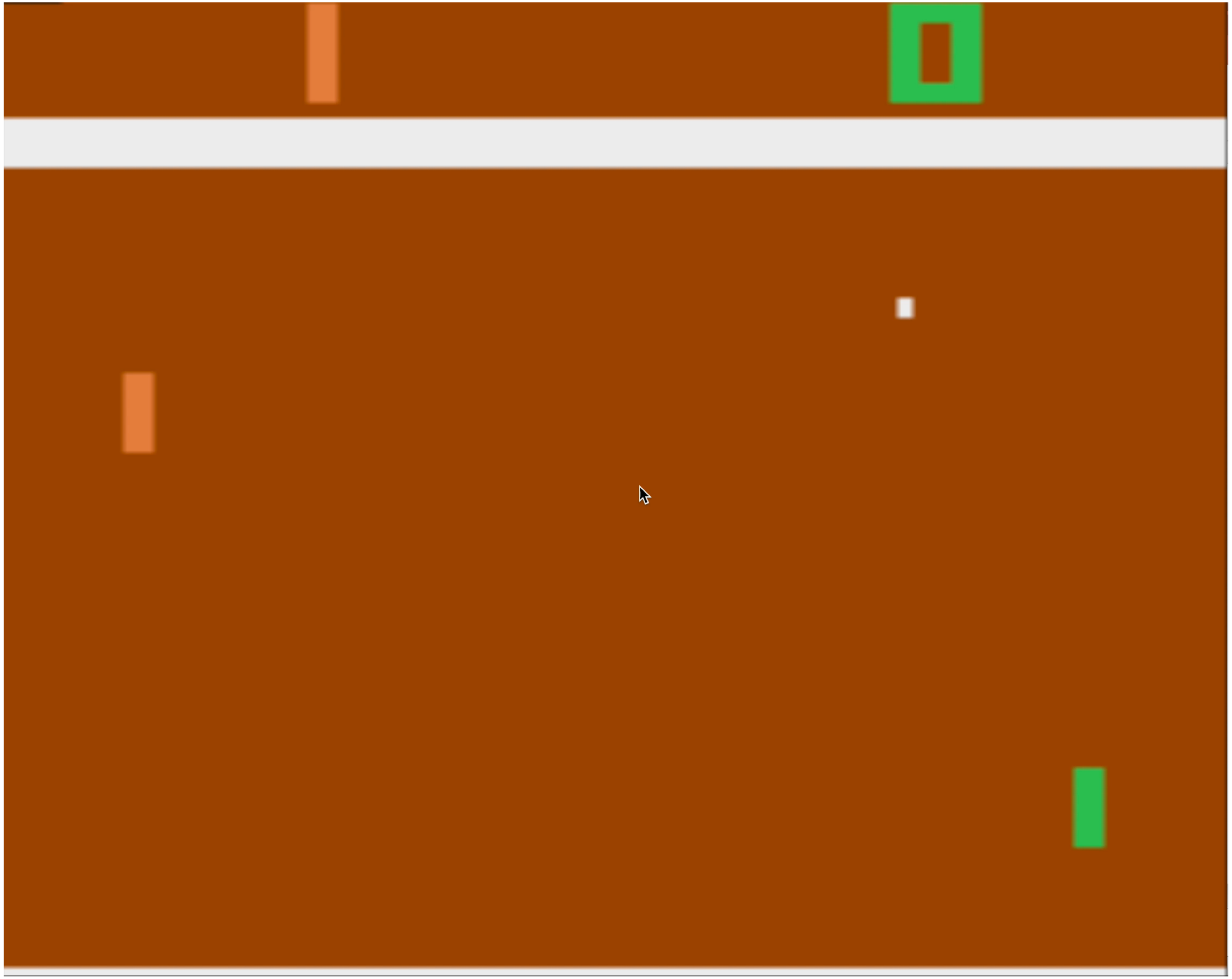
```
def test_model(model):
    score = 0
    done = False
    env = gym.make('CartPole-v0')
    state = env.reset()
    global action_probs
    while not done:
        score += 1
        s = torch.from_numpy(state).float().unsqueeze(0)

        action_probs = model.get_action_probs(Variable(s))

        _, action_index = action_probs.max(1)
        action = action_index.item()
        next_state, reward, done, thing = env.step(action)
        state = next_state
    return score
```

What is different compare to experience collection?

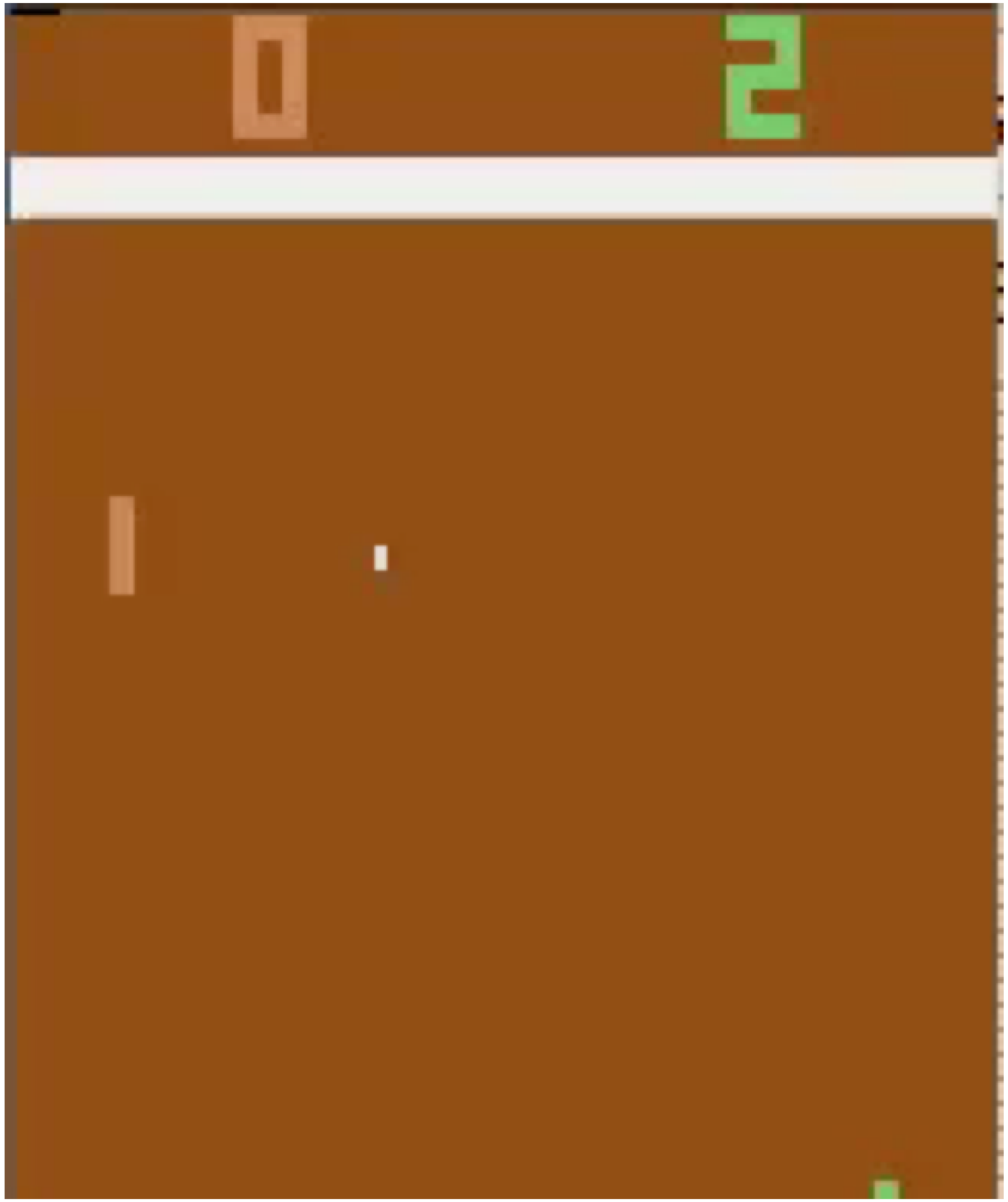
1 day of training



2 days of training



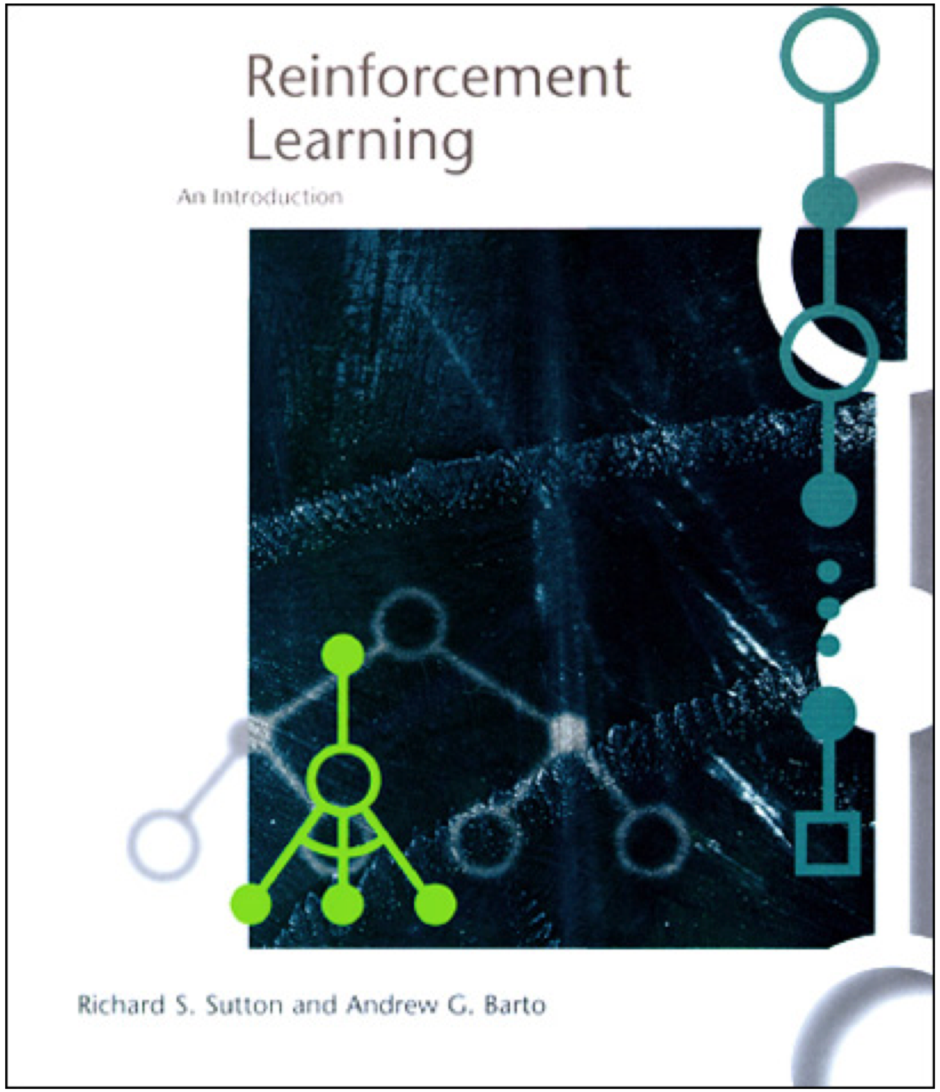
Final result for A3C



RL in real world

- Traffic Light Control
- Robotics
- Web Systems
- Chemistry
- Personalized Recommendations

The RL Intro book



Richard Sutton, Andrew Barto
Reinforcement Learning,
An Introduction

http://www.cs.ualberta.ca/
~sutton/book/the-book.html


THE END

Sources and kudos:

- Richard Sutton, Andrew Barto Reinforcement Learning, An Introduction.
- Rudy Gilman, Kathrine Wang Intuitive RL intro to advantage actor critic (A2C)
- Mnih Badia “Asynchronous Methods for Deep Reinforcement Learning”
- Peter Bodík RAD Lab, UC Berkeley Reinforcement Learning Tutorial