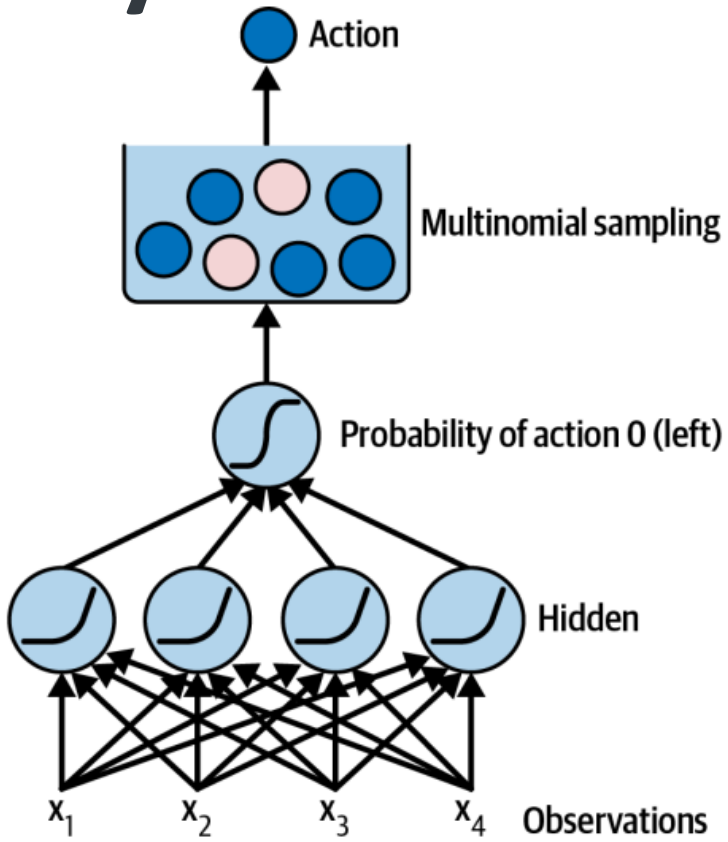
3.
Neural Network Policies

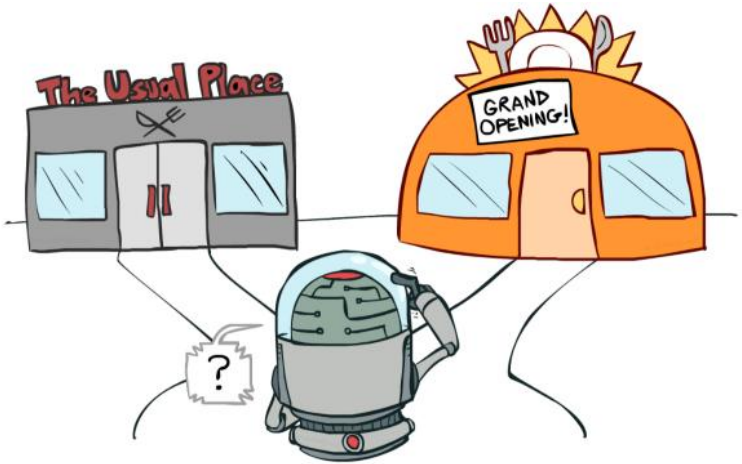
Neural Network Policy

- Input: observations
- Output: the action to be executed
  - The neural network will estimate a probability for each action, and then select an action randomly, according to these probabilities.
- For CartPole environment, there are just two actions (left or right), so we only need probability p of action 0 (left), and the probability of action 1 (right) will be $1 - p$.

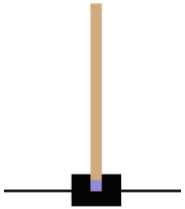


Exploration vs. Exploitation

- Why are we picking a random action based on the probabilities given by the network, rather than picking the action with the highest score?
  - This approach lets the agent find the right balance between *exploring* new actions and *exploiting* the actions that are known to work well.
- The *exploration/exploitation dilemma* is central in RL.



Hidden State



- In CartPole environment, the past actions and observations can be ignored, since each observation contains the environment's full state.
- If there were some hidden state, then you might need to consider past actions and observations as well.
  - Example: if the environment only reveal the position of the cart but not its velocity, you have to consider not only the current observation but also the previous observation in order to estimate the current velocity.
- Also if the observations are noisy, you generally want to use the past few observations to estimate the most likely current state.

Basic Neural Network Policy

- Building a basic neural network policy using Keras:

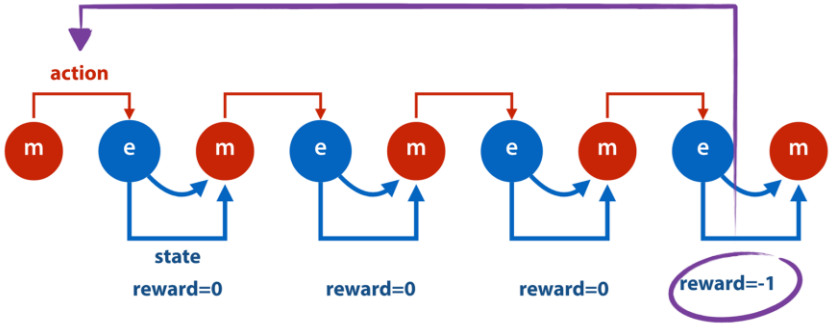
```
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Dense(5, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid"),
])
```

- The number of inputs is the size of the observation space, which in the case of CartPole is 4.

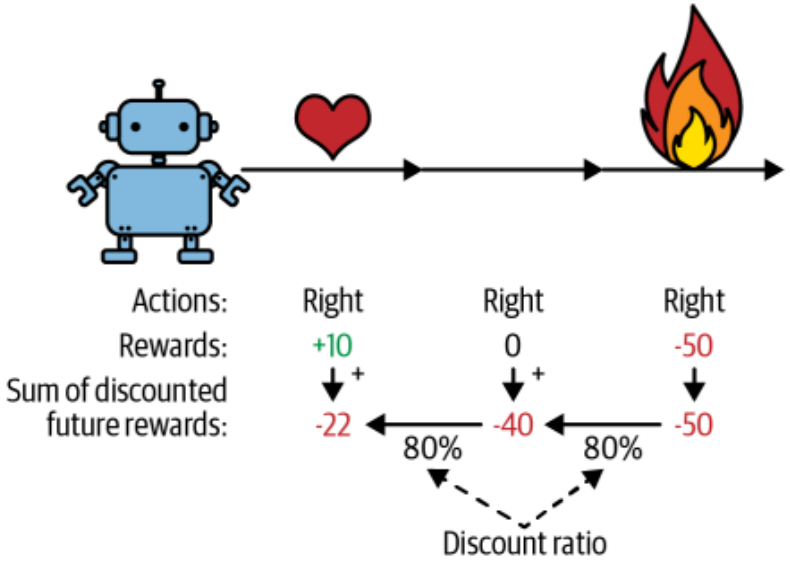
Evaluating Actions

- If we knew the best action at each step, we could train the neural network as regular supervised learning, by minimizing the cross entropy between the *estimated* and *target* probability distributions.
- However, in RL the only guidance the agent gets is through rewards, and rewards are typically sparse and delayed.
- *Credit assignment problem* (CAP): when the agent gets a reward, it is hard for it to know which actions should get credited (or blamed) for it.



Evaluating Actions

- A common strategy: evaluate an action based on the sum of all the rewards that come after it, applying a *discount factor* γ, at each step.
- This sum of discounted rewards is called the action's *return*.



Action Advantage

- A good action may be followed by several bad actions that cause the pole to fall quickly, resulting in the good action getting a low return.
- However, if we play the game enough times, on average good actions will get a higher return than bad ones.
- *Action advantage*: how much better or worse an action is, compared to the other possible actions, on average.
  - We must run many episodes and normalize all the action returns, by subtracting the mean and dividing by the standard deviation.
- We can reasonably assume that actions with a negative advantage were bad while actions with a positive advantage were good.

4.
Policy Gradient

REINFORCE Algorithm

1. Let the neural network policy play the game several times, and at each step, compute the gradients that would make the chosen action even more likely—but don't apply these gradients yet.
2. Once you have run several episodes, compute each action's advantage.
3. Multiply each gradient vector by the corresponding action's advantage:
    a. positive action advantage: the action is probably good, and you want to apply the gradients computed earlier to make the action even more likely to be chosen in the future.
    b. negative action advantage: the action is probably bad, and you want to apply the opposite gradients to make this action slightly less likely in the future.
4. Finally, compute the mean of all the resulting gradient vectors, and use it to perform a gradient descent step.

Play One Step

- We will pretend for now that whatever action it takes is the right one so that we can compute the loss and its gradients.
  - These gradients will just be saved for a while, and we will modify them later depending on how good or bad the action turned out to be.

```
def play_one_step(env, obs, model, loss_fn):
    with tf.GradientTape() as tape:
        left_proba = model(obs[np.newaxis])
        action = (tf.random.uniform([1, 1]) > left_proba)
        y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)
        loss = tf.reduce_mean(loss_fn(y_target, left_proba))

    grads = tape.gradient(loss, model.trainable_variables)
    obs, reward, done, truncated, info = env.step(int(action))
    return obs, reward, done, truncated, grads
```

Play Multiple Episodes

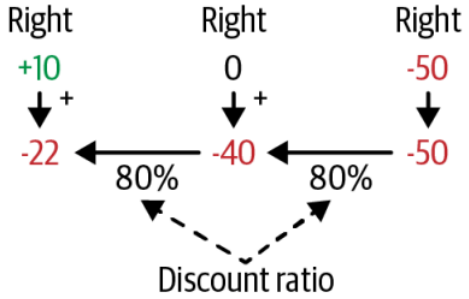
```
def play_multiple_episodes(env, n_episodes, n_max_steps, model, loss_fn):
    all_rewards = []
    all_grads = []
    for episode in range(n_episodes):
        current_rewards = []
        current_grads = []
        obs, info = env.reset()
        for step in range(n_max_steps):
            obs, reward, done, truncated, grads = play_one_step(
                env, obs, model, loss_fn)
            current_rewards.append(reward)
            current_grads.append(grads)
            if done or truncated:
                break

        all_rewards.append(current_rewards)
        all_grads.append(current_grads)

    return all_rewards, all_grads
```

Compute Discounted Rewards

```
def discount_rewards(rewards, discount_factor):
    discounted = np.array(rewards)
    for step in range(len(rewards) - 2, -1, -1):
        discounted[step] += discounted[step + 1] * discount_factor
    return discounted
```



- Say there were 3 actions, and after each action there was a reward: first 10, then 0, then -50:

```
discount_rewards([10, 0, -50], discount_factor=0.8)
```

```
array([-22, -40, -50])
```

Compute Discounted Rewards

- To normalize all discounted rewards across all episodes:

```
def discount_and_normalize_rewards(all_rewards, discount_factor):
    all_discounted_rewards = [discount_rewards(rewards, discount_factor)
                              for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean) / reward_std
            for discounted_rewards in all_discounted_rewards]
```

- Example:

```
discount_and_normalize_rewards([[10, 0, -50], [10, 20]], discount_factor=0.8)
```

```
[array([-0.28435071, -0.86597718, -1.18910299]),
 array([1.26665318, 1.0727777 ])]
```

Defining Hyperparameters

- Define the hyperparameters:

```
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_factor = 0.95
```

- Determine the optimizer and the loss function:

```
optimizer = tf.keras.optimizers.Nadam(learning_rate=0.01)
loss_fn = tf.keras.losses.binary_crossentropy
```

Build and Run the Training Loop

```
for iteration in range(n_iterations):
    all_rewards, all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, model, loss_fn)

    all_final_rewards = discount_and_normalize_rewards(all_rewards,
                                                       discount_factor)
    all_mean_grads = []
    for var_index in range(len(model.trainable_variables)):
        mean_grads = tf.reduce_mean(
            [final_reward * all_grads[episode_index][step][var_index]
             for episode_index, final_rewards in enumerate(all_final_rewards)
                 for step, final_reward in enumerate(final_rewards)], axis=0)
        all_mean_grads.append(mean_grads)

    optimizer.apply_gradients(zip(all_mean_grads, model.trainable_variables))
```

Limitations and Challenges

- This simple policy gradients algorithm solves the CartPole task, but it would not scale well to larger and more complex tasks.
  - It is highly *sample inefficient*: it needs to explore the game for a very long time before it can make significant progress.
- However, it is the foundation of more powerful algorithms, such as *actor-critic* algorithms.
- Researchers try to find algorithms that work well even when the agent initially knows nothing about the environment.
  - However, you should not hesitate to inject prior knowledge into the agent, as it will speed up training dramatically.