The code is here:
https://github.com/tsmatz/reinforcement-learning-tutorials/blob/master/03-actor-critic.ipynb

Policy-based algorithms find the optimal policy by changing the policy, whereas value-based algorithms find the optimal value function, which is supposed to lead to an optimal policy too. Value-based models are usually less accurate but more effective and stable. This code is an implementation for Actor-Critic algorithm. Actor-Critic combines policy-based and value-based methods. The agent, which is an actor, learns a policy for making decisions and a value function, which is a critic, evaluates the actions taken by the actor. In this code, they implement this algorithm to solve one of the classic control environments called CartPole (Barto et al., 1983). In this problem, there is a pole attached to a cart by an unactuated joint. The cart moves on a frictionless track. A pendulum is placed upright of the cart and the aim is to balance the pole by moving the cart with applying a force. The action space has two actions: pushing the cart to the right or left. We can observe the cart's position, velocity, angle and angular velocity of the pole. The simulation ends after 500 steps, and at each step 1 reward is given. To be successful, we need to keep the pole balanced for 500 steps, that leads to a total reward of 500.

Details of the code

Actor network

```
class ActorNet(nn.Module):
    def __init__(self, hidden_dim=16):
        super().__init__()

        """
        This is the neural network for the actor. It takes in the state and outputs the logits of the action.
        There are four dimensions in each state: cart position, cart velocity, pole angle, pole angular velocity.
        We have two actions, push cart to the left or to the right.
        """
        self.hidden = nn.Linear(4, hidden_dim)
        self.output = nn.Linear(hidden_dim, 2)

    def forward(self, s):
        outs = self.hidden(s)
        outs = F.relu(outs)
        logits = self.output(outs)
        return logits
```

Critic Network

```
class ValueNet(nn.Module):
    def __init__(self, hidden_dim=16):
        super().__init__()

        """
        This is the neural network for the critic. It takes in the state and outputs estimated value of the state.
        There are four dimensions in each state: cart position, cart velocity, pole angle, pole angular velocity.
        """

        self.hidden = nn.Linear(4, hidden_dim)
        self.output = nn.Linear(hidden_dim, 1)

    def forward(self, s):
        outs = self.hidden(s)
        outs = F.relu(outs)
        value = self.output(outs)
        return value
```

Sampling Function

```
"""
This function is to sample an action given a state. It takes in the state and returns the action.
"""
def pick_sample(s):
    with torch.no_grad():
        s_batch = np.expand_dims(s, axis=0) # adjust the size
        s_batch = torch.tensor(s_batch, dtype=torch.float)

        logits = actor_func(s_batch) # get the action logits from the actor model
        logits = logits.squeeze(dim=0) # adjust the shape
        probs = F.softmax(logits, dim=-1) # convert action logits to probabilities
        a = torch.multinomial(probs, num_samples=1) # sample an action
        return a.tolist()[0]
```

One episode iteration and cumulative reward calculation.

```
s, _ = env.reset()
#This is one episode, they complete when the pole falls or 500 steps are done
while not done:
    states.append(s.tolist())
    a = pick_sample(s) # sample an action
    s, r, term, trunc, _ = env.step(a) # take the action, update the state etc.
    done = term or trunc
    actions.append(a)
    rewards.append(r)

cum_rewards = np.zeros_like(rewards)
reward_len = len(rewards)
# They calculate the cumulative rewards, they use 0.99 as gamma by multplying the reward at the previous step.
for j in reversed(range(reward_len)):
    cum_rewards[j] = rewards[j] + (cum_rewards[j+1]*0.99 if j+1 < reward_len else 0)
```

Critic Model optimization

```
opt1.zero_grad() # delete the gradients from previous steps
states = torch.tensor(states, dtype=torch.float)
cum_rewards = torch.tensor(cum_rewards, dtype=torch.float)
values = value_func(states) # get the estimated values of the states
values = values.squeeze(dim=1) # adjust the size
vf_loss = F.mse_loss( # calculate the MSE loss between the estimated values and the cumulative rewards
    values,
    cum_rewards,
    reduction="none")
vf_loss.sum().backward() # backpropagate the loss
opt1.step() # update the weights of the critic
```

Actor model optimization

```
# get the estimated values of the states, witthout gradients this time because we will update the actor
with torch.no_grad():
    values = value_func(states)
opt2.zero_grad() # delete the gradients from previous steps
actions = torch.tensor(actions, dtype=torch.int64)
advantages = cum_rewards - values # calculate the advantages, basically additional values compared to baseline
logits = actor_func(states) # get the action logits from the actor model
log_probs = -F.cross_entropy(logits, actions, reduction="none") # calculate the log probabilities of the actions
pi_loss = -log_probs * advantages # calculate the policy loss
pi_loss.sum().backward() # backpropagate the loss
opt2.step() # update the weights of the actor
```

References

- Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5), 834-846.