```python
class ReplayBuffer():
    def __init__(self, max_size, input_shape, n_actions):
        self.mem_size = max_size
        self.mem_cntr = 0
        self.state_memory = np.zeros((self.mem_size, *input_shape))
        self.new_state_memory = np.zeros((self.mem_size, *input_shape))
        self.action_memory = np.zeros((self.mem_size, n_actions))
        self.reward_memory = np.zeros(self.mem_size)
        self.terminal_memory = np.zeros(self.mem_size, dtype=np.bool)

    def store_transition(self, state, action, reward, state_, done):
        index = self.mem_cntr % self.mem_size

        self.state_memory[index] = state
        self.new_state_memory[index] = state_
        self.action_memory[index] = action
        self.reward_memory[index] = reward
        self.terminal_memory[index] = done

        self.mem_cntr += 1

    def sample_buffer(self, batch_size):
        max_mem = min(self.mem_cntr, self.mem_size)

        batch = np.random.choice(max_mem, batch_size)

        states = self.state_memory[batch]
        states_ = self.new_state_memory[batch]
        actions = self.action_memory[batch]
        rewards = self.reward_memory[batch]
        dones = self.terminal_memory[batch]

        return states, actions, rewards, states_, dones
```

بصورت دوتایی وارد می‌شود

```python
class CriticNetwork(nn.Module):
    def __init__(self, beta, input_dims, n_actions, fc1_dims=256, fc2_dims=256,
            name='critic', chkpt_dir='/home/amirm_eb/Desktop/1'):
        super(CriticNetwork, self).__init__()
        self.input_dims = input_dims
        self.fc1_dims = fc1_dims
        self.fc2_dims = fc2_dims
        self.n_actions = n_actions
        self.name = name
        self.checkpoint_dir = chkpt_dir
        self.checkpoint_file = os.path.join(self.checkpoint_dir, name+'_sac')

        # state and action dimensions are concatenated
        self.fc1 = nn.Linear(self.input_dims[0] + n_actions, self.fc1_dims)
        self.fc2 = nn.Linear(self.fc1_dims, self.fc2_dims)
        self.q = nn.Linear(self.fc2_dims, 1)

        self.optimizer = optim.Adam(self.parameters(), lr=beta)
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')

        self.to(self.device)

    def forward(self, state, action):
        action_value = self.fc1(T.cat([state, action], dim=1))
        action_value = F.relu(action_value)
        action_value = self.fc2(action_value)
        action_value = F.relu(action_value)

        q = self.q(action_value)

        return q
```

$$y = r(s,a) + \gamma\left(\min_{i=1,2} Q_i(s',a') - \alpha \log \pi(a'|s')\right)$$

$$SSE = \sum (y - Q(s,a))^2$$

```python
class ActorNetwork(nn.Module):
    def __init__(self, alpha, input_dims, max_action, fc1_dims=256,
            fc2_dims=256, n_actions=2, name='actor', chkpt_dir='/home/amirm_eb/Desktop/1'):
        super(ActorNetwork, self).__init__()
        self.input_dims = input_dims
        self.fc1_dims = fc1_dims
        self.fc2_dims = fc2_dims
        self.n_actions = n_actions
        self.name = name
        self.checkpoint_dir = chkpt_dir
        self.checkpoint_file = os.path.join(self.checkpoint_dir, name+'_sac')
        self.max_action = max_action
        self.reparam_noise = 1e-6

        self.fc1 = nn.Linear(*self.input_dims, self.fc1_dims)
        self.fc2 = nn.Linear(self.fc1_dims, self.fc2_dims)
        self.mu = nn.Linear(self.fc2_dims, self.n_actions)
        self.sigma = nn.Linear(self.fc2_dims, self.n_actions)

        self.optimizer = optim.Adam(self.parameters(), lr=alpha)
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')

        self.to(self.device)

    def forward(self, state):
        prob = self.fc1(state)
        prob = F.relu(prob)
        prob = self.fc2(prob)
        prob = F.relu(prob)

        mu = self.mu(prob)
        sigma = self.sigma(prob)
```

شبکه actor

$\pi(a|s) =$

ورودی که خروجی s می‌دهد

شبکه عصبی

```python
        # We clamp the standard deviation to be within a stable range
        sigma = T.clamp(sigma, min=self.reparam_noise, max=1)

        return mu, sigma

    def sample_normal(self, state, reparameterize=True):
        mu, sigma = self.forward(state)
        probabilities = Normal(mu, sigma)

        if reparameterize:
            actions = probabilities.rsample() # Reparameterization trick
        else:
            actions = probabilities.sample()

        # Apply the squashing function and scale to action space
        action = T.tanh(actions) * T.tensor(self.max_action).to(self.device)

        # Calculate log probabilities, including the correction for the tanh squashing
        log_probs = probabilities.log_prob(actions)
        log_probs -= T.log(1 - action.pow(2) + self.reparam_noise)
        log_probs = log_probs.sum(1, keepdim=True)

        return action, log_probs
```

واریانس را بین 10^{-6} و ۱ می‌کند.

اگر خیلی کوچک باشد نزدیک صفر ممکن در محاسبات عددی خطا رخ بدهد

اگر زیاد باشد باعث اشباع شدن خواهد شد و انعطاف ... ندارد.

$\pi(a|s)$ نمونه بگیرد

نمونه $= m + \epsilon \cdot \sigma$

برای گرادیان بدست آمدن

یا نمونه عادی

$[-m-a, m-a]$

ریوقان tanh باید بزنیم.

$f(y) = \frac{f(x)}{g(x)}$

توجه داریم action ما یک بردار است برای بین log_probs داریم. (توجه داریم هر action کدام از مولفه‌های) بسته‌تر مستقلند و ماتریس واریانس قطری است.

```python
class Agent():
    def __init__(self, alpha, beta, input_dims,
                 env, gamma, n_actions, max_size,max_action, tau,
                 layer1_size, layer2_size, batch_size):
        self.gamma = gamma
        self.tau = tau
        self.memory = ReplayBuffer(max_size, input_dims, n_actions)
        self.batch_size = batch_size
        self.n_actions = n_actions
        self.max_action = max_action

        # --- Actor and Critic Networks ---
        self.actor = ActorNetwork(alpha, input_dims, n_actions=n_actions,fc1_dims = layer1_size,fc2_dims = layer2_size,
                    name='actor', max_action = self.max_action )

        self.critic_1 = CriticNetwork(beta, input_dims, n_actions=n_actions,fc1_dims = layer1_size,fc2_dims = layer2_size,
                    name='critic_1')
        self.critic_2 = CriticNetwork(beta, input_dims, n_actions=n_actions,fc1_dims = layer1_size,fc2_dims = layer2_size,
                    name='critic_2')

        # *** FIX ***: Removed the separate Value network.
        # --- Target Critic Networks for stable Q-learning targets ---
        self.target_critic_1 = CriticNetwork(beta, input_dims, n_actions=n_actions,fc1_dims = layer1_size,fc2_dims = layer2_size,
                    name='target_critic_1')
        self.target_critic_2 = CriticNetwork(beta, input_dims, n_actions=n_actions,fc1_dims = layer1_size,fc2_dims = layer2_size,
                    name='target_critic_2')
```

discount

برای آپدیت کردن

actor

دو تا

target

```python
# --- Target Critic Networks for stable Q-learning targets ---
self.target_critic_1 = CriticNetwork(beta, input_dims, n_actions=n_actions,fc1_dims = layer1_size,fc2_dims = layer2_size,
            name='target_critic_1')
self.target_critic_2 = CriticNetwork(beta, input_dims, n_actions=n_actions,fc1_dims = layer1_size,fc2_dims = layer2_size,
            name='target_critic_2')

# *** FIX ***: Added automatic temperature (alpha) tuning
self.log_alpha = T.zeros(1, requires_grad=True, device=self.actor.device)
self.alpha_optimizer = optim.Adam([self.log_alpha], lr=alpha)
# Target entropy is a heuristic. A common choice is -|A|, the negative of the action dimension.
self.target_entropy = -T.tensor(n_actions, dtype=T.float).to(self.actor.device)

self.update_network_parameters(tau=1) # Initialize target networks to match main networks
```

دو تا target

پارامتر قابل یادگیری

برای یادگیری پارامتر آلفا

کپی کردن target و اصلی

```python
def choose_action(self, observation):
    state = T.Tensor(np.array([observation])).to(self.actor.device)
    # When acting in the environment, we don't need reparameterization
    actions, _ = self.actor.sample_normal(state, reparameterize=False)

    return actions.cpu().detach().numpy()[0]
def store_transition(self, state, action, reward, new_state, done):
    self.memory.store_transition(state, action, reward, new_state, done)
```

```python
def update_network_parameters(self, tau=None):
    if tau is None:
        tau = self.tau

    # --- Update Target Critic 1 ---
    target_critic_1_params = self.target_critic_1.named_parameters()
    critic_1_params = self.critic_1.named_parameters()
    target_critic_1_state_dict = dict(target_critic_1_params)
    critic_1_state_dict = dict(critic_1_params)
    for name in critic_1_state_dict:
        critic_1_state_dict[name] = tau * critic_1_state_dict[name].clone() + \
            (1 - tau) * target_critic_1_state_dict[name].clone()
    self.target_critic_1.load_state_dict(critic_1_state_dict)

    # --- Update Target Critic 2 ---
    target_critic_2_params = self.target_critic_2.named_parameters()
    critic_2_params = self.critic_2.named_parameters()
    target_critic_2_state_dict = dict(target_critic_2_params)
    critic_2_state_dict = dict(critic_2_params)
    for name in critic_2_state_dict:
        critic_2_state_dict[name] = tau * critic_2_state_dict[name].clone() + \
            (1 - tau) * target_critic_2_state_dict[name].clone()
    self.target_critic_2.load_state_dict(critic_2_state_dict)
```

کپی کردن target از اصلی با استفاده از سر (تاو)

```python
def learn(self):
    if self.memory.mem_cntr < self.batch_size:
        return

    state, action, reward, new_state, done = \
        self.memory.sample_buffer(self.batch_size)

    # Convert numpy arrays to tensors
    reward = T.tensor(reward, dtype=T.float).to(self.actor.device)

    # *** FIX ***: Cast the 'done' tensor to float to allow for arithmetic operations
    done = T.tensor(done, dtype=T.float).to(self.actor.device)

    state_ = T.tensor(new_state, dtype=T.float).to(self.actor.device)
    state = T.tensor(state, dtype=T.float).to(self.actor.device)
    action = T.tensor(action, dtype=T.float).to(self.actor.device)

    with T.no_grad():
        next_actions, next_log_probs = self.actor.sample_normal(state_, reparameterize=True)

        target_q1_values = self.target_critic_1.forward(state_, next_actions)
        target_q2_values = self.target_critic_2.forward(state_, next_actions)
        target_q_values = T.min(target_q1_values, target_q2_values).view(-1)

        alpha = self.log_alpha.exp()
        # The (1 - done) term will now work correctly
        next_target = reward + self.gamma * (1 - done) * (target_q_values - alpha * next_log_probs.view(-1))

    # --- Update Critic Networks ---
    q1_values = self.critic_1.forward(state, action).view(-1)
    q2_values = self.critic_2.forward(state, action).view(-1)
```

```python
q1_values = self.critic_1.forward(state, action).view(-1)
q2_values = self.critic_2.forward(state, action).view(-1)

critic_1_loss = 0.5 * F.mse_loss(q1_values, next_target)
critic_2_loss = 0.5 * F.mse_loss(q2_values, next_target)
critic_loss = critic_1_loss + critic_2_loss

self.critic_1.optimizer.zero_grad()
self.critic_2.optimizer.zero_grad()
critic_loss.backward()
self.critic_1.optimizer.step()
self.critic_2.optimizer.step()
```

```python
actions, log_probs = self.actor.sample_normal(state, reparameterize=True)

q1_new_policy = self.critic_1.forward(state, actions)
q2_new_policy = self.critic_2.forward(state, actions)
min_q_new_policy = T.min(q1_new_policy, q2_new_policy).view(-1)

actor_loss = (alpha.detach() * log_probs.view(-1) - min_q_new_policy).mean()

self.actor.optimizer.zero_grad()
actor_loss.backward()
self.actor.optimizer.step()

# --- Alpha (temperature) loss ---
alpha_loss = -(self.log_alpha * (log_probs.detach() + self.target_entropy)).mean()

self.alpha_optimizer.zero_grad()
alpha_loss.backward()
self.alpha_optimizer.step()

# --- Soft update target networks ---
self.update_network_parameters()
```

تابع خطای actor
E(___)

کپی با تأخیر از اصلی به target

```python
env = gym.make(env_name, render_mode="rgb_array")
dir_path = os.path.join(dir, env_name)
os.makedirs(dir_path, exist_ok=True)

if record_video:
    env = RecordVideo(env, video_folder=os.path.join(dir_path, 'videos'),
                      episode_trigger=lambda ep: ep == n_games - 1)

# *** FIX ***: Instantiate agent with the correct (modern) parameters
agent = Agent(alpha=alpha_lr, beta=beta_lr, input_dims=env.observation_space.shape,
              env=env,gamma = gamma, n_actions=env.action_space.shape[0],max_size = memory_size,
              max_action = float(env.action_space.high[0]),tau =tau,
              layer1_size =fc1_dim,layer2_size = fc2_dim, batch_size=batch_size)

scores = []
best_score = -np.inf

# Add a warmup phase
num_steps = 0
```

ساخت فیلم برای آخری

ساخت agent

برای این score ماکزیمم بشود.

```python
for game in trange(n_games):
    observation, info = env.reset()
    score = 0
    done = False

    while not (done):
        if num_steps < warmup:
            # Take random actions during the warmup phase to fill the buffer
            action = env.action_space.sample()
        else:
            action = agent.choose_action(observation)

        next_observation, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated
        score += reward
        agent.store_transition(observation, action, reward, next_observation, done)

        if num_steps >= warmup:
            agent.learn()

        observation = next_observation
        num_steps += 1

    scores.append(score)
    avg_score = np.mean(scores[-100:])

    if avg_score > best_score:
        best_score = avg_score
        agent.save_models()
```