```
import numpy as np
import torch
import torch.nn.functional as F
import torch.optim as optim
import random
```

REINFORCE from Sutton&Barto

```
class ReinforceAgent():
    def __init__(self, input_shape, action_size, seed, device, gamma, lr, policy):
        """Initialize an Agent object.
        Params
        ======
            input_shape (tuple): dimension of each state (C, H, W)
            action_size (int): dimension of each action
            seed (int): random seed
            device(string): Use Gpu or CPU
            gamma (float): discount factor
            lr (float): Learning rate
            policy(Model): Pytorch Policy Model
        """
        self.input_shape = input_shape
        self.action_size = action_size
        self.seed = random.seed(seed)
        self.device = device
        self.lr = lr
        self.gamma = gamma
```

single neural network for policy

```
        # Actor-Network
        self.policy_net = policy(input_shape, action_size).to(self.device)
        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=self.lr)

        # Memory
        self.log_probs = []
        self.rewards   = []
        self.masks     = []
```

log probabilities, or $\log \pi(a|s;\theta)$

direct mapping $\pi: s \to a$

```
    def step(self, log_prob, reward, done):

        # Save experience in  memory
        self.log_probs.append(log_prob)
        self.rewards.append(torch.from_numpy(np.array([reward])).to(self.device))
        self.masks.append(torch.from_numpy(np.array([1 - done])).to(self.device))


    def act(self, state):
        """Returns action, log_prob for given state as per current policy."""

        state = torch.from_numpy(state).unsqueeze(0).to(self.device)
        action_probs = self.policy_net(state)
```

available in torch for sampling from softmax

```
        action = action_probs.sample()
        log_prob = action_probs.log_prob(action)
```

and for getting $\log \pi(a|s;\theta)$

```
        return action.item(), log_prob

    def learn(self):

        returns = self.compute_returns(0, self.gamma)

        log_probs = torch.cat(self.log_probs)
        returns   = torch.cat(returns).detach()

        loss  = -(log_probs * returns).mean()
```

Basic PG without baseline computes sample average as approximation of $E[\log \pi(a|s,\theta) \cdot G_t]$

```
        # Minimize the loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

```
        self.reset_memory()

    def reset_memory(self):
        del self.log_probs[:]
        del self.rewards[:]
        del self.masks[:]

    def compute_returns(self, next_value, gamma=0.99):
        R = next_value
        returns = []
        for step in reversed(range(len(self.rewards))):
            R = self.rewards[step] + gamma * R * self.masks[step]
            returns.insert(0, R)
        return returns
```