**layers = []**

**for i in range(len(sizes)-1):**

**layers.append(nn.Linear(sizes[i], sizes[i+1]))**

**layers.append(activation())**

**layers[-1] = output\_activation()**

**mlp\_model = nn.Sequential(\*layers)**

For the MLP model, I add the input layer, the hidden layers, and the output layer into a list in order, as well as the activation function between these layers. And load the list in the nn.Sequential to construct the model.

**torch.distributions.Categorical(logits=self.logits\_net(obs))**

For the \_distribution function of the Actor class, I pass the observation to the MLP network to get the log probs. of the observation, and generate a Categorical distribution using the log probabilities of the observation.

**pi.log\_prob(act)**

For the log prob over the given action, just need to invoke the log\_prob method of the Categorical distribution ‘pi’.

As for the forward method of the Actor class, the implementation is done by invoking the methods above and returning the distribution ‘pi’ and the log prob of the action.

**d\_vals = vals[1:]-vals[:-1]**

**self.tdres\_buf[path\_slice] = discount\_cumsum(rews[:-1]+d\_vals, self.gamma\* self.lam)**

**self.ret\_buf[path\_slice] = discount\_cumsum(rews[:-1], self.gamma)**

About the end trajectory method of the class VPGBuffer, the TD residual is calculated by . So I minus the v[1:-2] by v[2:-1] to get the .

The trajectory's remaining return is obtained by discount cumsum of reward with a discount gamma.

**with torch.no\_grad():**

**pi = self.actor.\_distribution(state)**

**act\_sample = pi.sample()**

**v = self.critic.forward(state)**

**# given obs 'state' and the prob. distri. generate from obs, get the log\_p of the sampled action**

**log\_p\_act = self.actor.\_log\_prob\_from\_distribution(pi, act\_sample)**

**return np.array(act\_sample), np.array(v), np.array(log\_p\_act)**

For the step method of the class of Agent, first I need to get a distribution ‘pi’ and sample an action from it, then calculate the log prob of this action. The estimated value is obtained by the forward method of the Critic class. Transform the variables to ndarray before returning them.

**return self.act(torch.as\_tensor(obs, dtype=torch.float32))**

for the get\_action method, invoke the provided method act to invoke the step method, and transform the observation to tensor before passing it to the method.

**data = buf.get()**

**obs = data['obs']**

**act = data['act']**

**tdres = data['tdres'].squeeze()**

**# Do 1 policy gradient update**

**actor\_optimizer.zero\_grad()**

**# policy gradient - residual of TD multi the log\_p of action**

**log\_p\_act = agent.actor.forward(obs,act)[1]**

**actor\_loss = - tdres @ log\_p\_act**

**actor\_loss.backward()**

**actor\_optimizer.step()**

**# We suggest to do 100 iterations of value function updates**

**ret = data['ret']**

**for \_ in range(100):**

**critic\_optimizer.zero\_grad()**

**val\_estimate = agent.critic.forward(obs)**

**critic\_loss = nn.functional.mse\_loss(val\_estimate, ret)**

**critic\_loss.backward()**

**critic\_optimizer.step()**

The loss of the actor\_optimizer is calculated by the td residual multi the log prob. of action.

The loss of the critic\_optimizer is the MSE loss between the estimated value getting by critic.forward(obs) and the reward of trajectory ‘ret’.