Table of Contents

Probability distributions - torch.distributions

The `distributions` package contains parameterizable probability distributions and sampling functions. This allows the construction of stochastic computation graphs and stochastic gradient estimators for optimization. This package generally follows the design of the TensorFlow Distributions package.

It is not possible to directly backpropagate through random samples. However, there are two main methods for creating surrogate functions that can be backpropagated through. These are the score function estimator/likelihood ratio estimator/REINFORCE and the pathwise derivative estimator. REINFORCE is commonly seen as the basis for policy gradient methods in reinforcement learning, and the pathwise derivative estimator is commonly seen in the reparameterization trick in variational autoencoders. Whilst the score function only requires the value of samples $f(x)$, the pathwise derivative requires the derivative $f'(x)$. The next sections discuss these two in a reinforcement learning example. For more details see Gradient Estimation Using Stochastic Computation Graphs .

Score function

When the probability density function is differentiable with respect to its parameters, we only need `sample()` and `log_prob()` to implement REINFORCE:

$$\Delta\theta = \alpha r \frac{\partial \log p(a|\pi^\theta(s))}{\partial \theta}$$

where θ are the parameters, α is the learning rate, r is the reward and $p(a|\pi^\theta(s))$ is the probability of taking action a in state s given policy π^θ.

In practice we would sample an action from the output of a network, apply this action in an environment, and then use `log_prob` to construct an equivalent loss function. Note that we use a negative because optimizers use gradient descent, whilst the rule above assumes gradient ascent. With a categorical policy, the code for implementing REINFORCE would be as follows:

```
probs = policy_network(state)
# Note that this is equivalent to what used to be called multinomial
m = Categorical(probs)
action = m.sample()
next_state, reward = env.step(action)
loss = -m.log_prob(action) * reward
loss.backward()
```

```
torch.manual_seed(0)
probs=torch.rand(4)probs.requires_grad=True)
m = torch.distributions.Categorical(probs)
action= m.sample()
next_state, reward = -1, 0.8 # 应该为env.step(action)
loss = -m.log_prob(action) * reward
loss.backward(retain_graph=True)
loss.grad # 梯度为1
action.grad # 无梯度
probs.grad # 有梯度，tensor([ 0.5387，-0.5026，0.5387，0.5387])
```

可以看到loss的梯度最终从loss->cagegorical->probs->policy_network

Pathwise derivative

The other way to implement these stochastic/policy gradients would be to use the reparameterization trick from the `rsample()` method, where the parameterized random variable can be constructed via a parameterized deterministic function of a parameter-free random variable. The reparameterized sample therefore becomes differentiable. The code for implementing the pathwise derivative would be as follows:

```
params = policy_network(state)
m = Normal(*params) 此处的params一般为mean与std, 梯度会传到mean与std
# Any distribution with .has_rsample == True could work based on the application
action = m.rsample()
next_state, reward = env.step(action)  # Assuming that reward is differentiable
loss = -reward
loss.backward()
```

```
torch.manual_seed(0)
n=4
mean=torch.zeros([n], requires_grad=True)
std = torch.ones([n], requires_grad=True)
m = torch.distributions.Normal(mean, std)
action=m.rsample()next_state, reward = -1, 0.8*action # 实际上是从env获得env.step(action)
loss = -reward.sum()loss.backward(retain_graph=True)

loss.grad # 梯度为1
action.grad # 有梯度，
probs.grad # 有梯度，tensor([ 0.5387，-0.5026，0.5387，0.5387])
```

可以看到loss的梯度最终从loss->reward->action->(mean, std)->policy_network

Distribution

CLASS torch.distributions.distribution.Distribution(*batch_shape=torch.Size([])*, *event_shape=torch.Size([])*, *validate_args=None*) [SOURCE]

Bases: `object`

Distribution is the abstract base class for probability distributions.

PROPERTY arg_constraints: DICT[STR, CONSTRAINT]

Returns a dictionary from argument names to `Constraint` objects that should be satisfied by each argument of this distribution. Args that are not tensors need not appear in this dict.

PROPERTY batch_shape: SIZE

Returns the shape over which parameters are batched.

cdf(*value*) [SOURCE]

Returns the cumulative density/mass function evaluated at *value*.

Parameters

value (*Tensor*) –

Return type

Tensor