In [1]:
```
# import some dependencies
import torch
from torch.autograd import Variable

import pyro
import pyro.distributions as dist
```

Models in Pyro: From Primitive Distributions to Stochastic Functions

The basic unit of Pyro programs is the *stochastic function*. This is an arbitrary Python callable that combines two ingredients:

- deterministic Python code; and
- primitive stochastic functions

Concretely, a stochastic function can be any Python object with a `__call__()` method, like a function, a method, or a PyTorch `nn.Module`.

Throughout the tutorials and documentation, we will often call stochastic functions *models*, since stochastic functions can be used to represent simplified or abstract descriptions of a process by which data are generated. Expressing models as stochastic functions in Pyro means that models can be composed, reused, imported, and serialized just like regular Python callables.

Without further ado, let's introduce one of our basic building blocks: primitive stochastic functions.

Primitive Stochastic Functions

Primitive stochastic functions, or distributions, are an important class of stochastic functions for which we can explicitly compute the probability of the outputs given the inputs. Pyro includes a standalone library, `pyro.distributions`, of GPU-accelerated multivariate probability distributions built on PyTorch. This comes with

various familiar distributions like the bernoulli and uniform distributions, but users can implement custom distributions by subclassing `pyro.distributions.Distribution`.

Using primitive stochastic functions is easy. For example, to draw a sample `x` from the unit normal distribution $\mathcal{N}(0, 1)$ we do the following:

In [2]:
```
mu = Variable(torch.zeros(1))   # mean zero
sigma = Variable(torch.ones(1)) # unit variance
x = dist.normal(mu, sigma)      # x is a sample from N(0,1)
print(x)
```

```
Variable containing:
 1.6869
[torch.FloatTensor of size 1]
```

Importantly, `dist.normal` is a *function* that takes parameters and `dist.normal(...)` returns a sample. Note that the parameters passed to `dist.normal` are PyTorch `Variables`. This is necessary because we want to make use of PyTorch's fast tensor math and autograd capabilities. To score the sample `x`, i.e. to compute its log probability according to the distribution $\mathcal{N}(0, 1)$, we simply do the following:

In [3]:
```
log_p_x = dist.normal.log_pdf(x, mu, sigma)
print(log_p_x)
```

```
Variable containing:
-2.3417
[torch.FloatTensor of size 1]
```

Note that the arguments to the `log_pdf` method are: first, the value to score (i.e. `x`), followed by the arguments to the distribution (i.e. `mu` and `sigma`).

The `pyro.sample` Primitive

One of the core language primitives in Pyro is the `pyro.sample` statement. Using `pyro.sample` is as simple as calling a primitive stochastic function with one important difference:

In [4]:
```
x = pyro.sample("my_sample", dist.normal, mu, sigma)
print(x)
```

```
Variable containing:
-0.8290
[torch.FloatTensor of size 1]
```

Just like a direct call to `dist.normal`, this returns a sample from the unit normal distribution. The crucial difference is that this sample is *named*. Pyro's backend uses these names to uniquely identify sample statements and *change their behavior at runtime* depending on how the enclosing stochastic function is being used. As we will see, this is how Pyro can implement the various manipulations that underlie inference algorithms.

We've seen how `pyro.sample` interacts with a primitive stochastic function like `dist.normal`. Things are no different for an arbitrary stochastic function `fn()`. For example we might have:

```
x = pyro.sample("my_sample", fn, arg1, arg2)
```

This results in a named sample from `fn` with value `x` and arguments `arg1` and `arg2` passed to `fn`.

A Simple Model

Now that we've introduced `pyro.sample` and `pyro.distributions` we can write a simple model. Since we're ultimately interested in probabilistic programming because we want to model things in the real world, let's choose something concrete.

Let's suppose we have a bunch of data with daily mean temperatures and cloud cover. We want to reason about how temperature interacts with whether it was sunny or cloudy. A simple stochastic function that does that is given by:

In [5]:
```
def weather():
    cloudy = pyro.sample('cloudy', dist.bernoulli,
                         Variable(torch.Tensor([0.3])))
    cloudy = 'cloudy' if cloudy.data[0] == 1.0 else 'sunny'
    mean_temp = {'cloudy': [55.0], 'sunny': [75.0]}[cloudy]
    sigma_temp = {'cloudy': [10.0], 'sunny': [15.0]}[cloudy]
    temp = pyro.sample('temp', dist.normal,
                       Variable(torch.Tensor(mean_temp)),
                       Variable(torch.Tensor(sigma_temp)))
    return cloudy, temp.data[0]

for _ in range(3):
    print(weather())
```

```
('sunny', 62.1190299987793)
('cloudy', 58.74806213378906)
('sunny', 79.73405456542969)
```

Let's go through this line-by-line. First, in lines 2-3 we use `pyro.sample` to define a binary random variable 'cloudy', which is given by a draw from the bernoulli distribution with a parameter of `0.3`. Since the bernoulli distributions returns `0`s or `1`s, in line 4 we convert the value `cloud` to a string so that return values of `weather` are easier to parse. So according to this model 30% of the time it's cloudy and 70% of the time it's sunny.

In lines 5-6 we define the parameters we're going to use to sample the temperature in lines 7-9. These parameters depend on the particular value of `cloudy` we sampled in line 2. For example, the mean temperature is 55 degrees (Fahrenheit) on cloudy days and 75 degrees on sunny days. Finally we return the two values `cloudy` and `temp` in line 10.

Procedurally, `weather()` is a non-deterministic Python callable that returns two random samples. Because the randomness is invoked with `pyro.sample`, however, it is much more than that. In particular `weather()` specifies a joint probability distribution over two named random variables: `cloudy` and `temp`. As such, it defines a probabilistic model that we can reason about using the techniques of probability theory. For example we might ask: if I observe a temperature of 70 degrees, how likely is it to be cloudy? How to formulate and answer these kinds of questions will be the subject of the next tutorial.

We've now seen how to define a simple model. Building off of it is easy. For example:

In [6]:
```
def ice_cream_sales():
    cloudy, temp = weather()
    expected_sales = [200] if cloudy == 'sunny' and temp > 80.0 else
[50]
    ice_cream = pyro.sample('ice_cream', dist.normal,
                            Variable(torch.Tensor(expected_sales)),
                            Variable(torch.Tensor([10.0])))
    return ice_cream
```

This kind of modularity, familiar to any programmer, is obviously very powerful. But is it powerful enough to encompass all the different kinds of models we'd like to express?

Universality: Stochastic Recursion, Higher-order Stochastic Functions, and Random Control Flow

Because Pyro is embedded in Python, stochastic functions can contain arbitrarily complex deterministic Python and randomness can freely affect control flow. For example, we can construct recursive functions that terminate their recursion nondeterministically, provided we take care to pass `pyro.sample` unique sample names whenever it's called. For example we can define a geometric distribution like so:

In [7]:
```
def geometric(p, t=None):
    if t is None:
        t = 0
    x = pyro.sample("x_{}".format(t), dist.bernoulli, p)
    if torch.equal(x.data, torch.zeros(1)):
        return x
    else:
        return x + geometric(p, t+1)

print(geometric(Variable(torch.Tensor([0.5]))))
```

```
Variable containing:
 0
[torch.FloatTensor of size 1]
```

Note that the names `x_0`, `x_1`, etc., in `geometric()` are generated dynamically and that different executions can have different numbers of named random variables.

We are also free to define stochastic functions that accept as input or produce as output other stochastic functions:

In [8]:
```
def normal_product(mu, sigma):
    z1 = pyro.sample("z1", dist.normal, mu, sigma)
    z2 = pyro.sample("z2", dist.normal, mu, sigma)
    y = z1 * z2
    return y

def make_normal_normal():
    mu_latent = pyro.sample("mu_latent", dist.normal,
                            Variable(torch.zeros(1)),
                            Variable(torch.ones(1)))
    fn = lambda sigma: normal_product(mu_latent, sigma)
    return fn

print(make_normal_normal()(Variable(torch.ones(1))))
```

```
Variable containing:
 0.8229
[torch.FloatTensor of size 1]
```

Here `make_normal_normal()` is a stochastic function that takes one argument and which, upon execution, generates three named random variables.

The fact that Pyro supports arbitrary Python code like this—iteration, recursion, higher-order functions, etc.—in conjuction with random control flow means that Pyro stochastic functions are *universal*, i.e. they can be used to represent any computable probability distribution. As we will see in subsequent tutorials, this is incredibly powerful.

It is worth emphasizing that this is one reason why Pyro is built on top of PyTorch: dynamic computational graphs are an important ingredient in allowing for universal models that can benefit from GPU-accelerated tensor math.

Next Steps

We've shown how we can use stochastic functions and primitive distributions to represent models in Pyro. In order to learn models from data and reason about them we need to be able to do inference. This is the subject of the next tutorial.