CHAPTER 2

Probability Distributions Using PyTorch

Probability and random variables are an integral part of computation in a graph-computing platform like PyTorch. Understanding probability and associated concepts are essential. This chapter covers probability distributions and implementation using PyTorch, and interpreting the results from tests.

In probability and statistics, a random variable is also known as a *stochastic variable*, whose outcome is dependent on a purely stochastic phenomenon, or random phenomenon. There are different types of probability distributions, including normal distribution, binomial distribution, multinomial distribution, and Bernoulli distribution. Each statistical distribution has its own properties.

The torch.distributions module contains probability distributions and sampling functions. Each distribution type has its own importance in a computational graph. The distributions module contains binomial, Bernoulli, beta, categorical, exponential, normal, and Poisson distributions.

P. Mishra, *PyTorch Recipes*, https://doi.org/10.1007/978-1-4842-4258-2_2

Recipe 2-1. Sampling Tensors

Problem

Weight initialization is an important task in training a neural network and any kind of deep learning model, such as a convolutional neural network (CNN), a deep neural network (DNN), and a recurrent neural network (RNN). The question always remains on how to initialize the weights.

Solution

Weight initialization can be done by using various methods, including random weight initialization. Weight initialization based on a distribution is done using uniform distribution, Bernoulli distribution, multinomial distribution, and normal distribution. How to do it using PyTorch is explained next.

How It Works

To execute a neural network, a set of initial weights needs to be passed to the backpropagation layer to compute the loss function (and hence, the accuracy can be calculated). The selection of a method depends on the data type, the task, and the optimization required for the model. Here we are going to look at all types of approaches to initialize weights.

If the use case requires reproducing the same set of results to maintain consistency, then a manual seed needs to be set.

```
In [2]: import torch

In [3]: # how to perform random sampling of the tensors

In [4]: torch.manual_seed(1234)

Out[4]: <torch._C.Generator at 0x2852c8a1b30>

In [5]: torch.manual_seed(1234)
        torch.randn(4,4)

Out[5]: tensor([[-0.1117, -0.4966,  0.1631, -0.8817],
                [ 0.0539,  0.6684, -0.0597, -0.4675],
                [-0.2153,  0.8840, -0.7584, -0.3689],
                [-0.3424, -1.4020,  0.3206, -1.0219]])
```

The seed value can be customized. The random number is generated purely by chance. Random numbers can also be generated from a statistical distribution. The probability density function of the *continuous uniform distribution* is defined by the following formula.

$$f(x)=\begin{cases}\dfrac{1}{b-a} & \text{for } a\le x\le b,\\ 0 & \text{for } x<a \text{ or } x>b\end{cases}$$

The function of x has two points, a and b, in which a is the starting point and b is the end. In a continuous uniform distribution, each number has an equal chance of being selected. In the following example, the start is 0 and the end is 1; between those two digits, all 16 elements are selected randomly.

```
In [8]: torch.Tensor(4, 4).uniform_(0, 1) #random number from uniform distribution

Out[8]: tensor([[0.2837, 0.6567, 0.2388, 0.7313],
                [0.6012, 0.3043, 0.2548, 0.6294],
                [0.9665, 0.7399, 0.4517, 0.4757],
                [0.7842, 0.1525, 0.6662, 0.3343]])
```

In statistics, the *Bernoulli distribution* is considered as the discrete probability distribution, which has two possible outcomes. If the event happens, then the value is 1, and if the event does not happen, then the value is 0.

For *discrete probability distribution*, we calculate probability mass function instead of probability density function. The probability mass function looks like the following formula.

$$\begin{cases} q = (1-p) & \text{for} k = 0 \\ p & \text{for} k = 1 \end{cases}$$

From the Bernoulli distribution, we create sample tensors by considering the uniform distribution of size 4 and 4 in a matrix format, as follows.

```
In [10]: torch.bernoulli(torch.Tensor(4, 4).uniform_(0, 1))

Out[10]: tensor([[1., 1., 1., 1.],
                 [0., 0., 0., 0.],
                 [0., 0., 1., 0.],
                 [1., 1., 0., 0.]])
```

The generation of sample random values from a *multinomial distribution* is defined by the following script. In a multinomial distribution, we can choose with a replacement or without a replacement. By default, the multinomial function picks up without a replacement and returns the result as an index position for the tensors. If we need to run it with a replacement, then we need to specify that while sampling.

```
In [12]: torch.Tensor([10, 10, 13, 10,34,45,65,67,87,89,87,34])

Out[12]: tensor([10., 10., 13., 10., 34., 45., 65., 67., 87., 89., 87., 34.])

In [13]: torch.multinomial(torch.tensor([10., 10., 13., 10.,
                                         34., 45., 65., 67.,
                                         87., 89., 87., 34.]),
                           3)

Out[13]: tensor([ 8,  7, 11])
```

Sampling from multinomial distribution with a replacement returns the tensors' index values.

```
In [14]: torch.multinomial(torch.tensor([10., 10., 13., 10.,
                                         34., 45., 65., 67.,
                                         87., 89., 87., 34.]),
                           5, replacement=True)
Out[14]: tensor([ 5, 10,  5,  9, 10])
```

The weight initialization from the normal distribution is a method that is used in fitting a neural network, fitting a deep neural network, and CNN and RNN. Let's have a look at the process of creating a set of random weights generated from a normal distribution.

```
In [16]: torch.normal(mean=torch.arange(1., 11.),
                      std=torch.arange(1, 0, -0.1))
Out[16]: tensor([1.2111, 2.3034, 3.5310, 4.7278, 6.1060, 6.3294, 6.9060, 7.9908,
         9.3492,
                 9.9928])

In [17]: torch.normal(mean=0.5,
                      std=torch.arange(1., 6.))
Out[17]: tensor([-1.1794, -2.9019,  2.4459,  7.5613,  5.9058])

In [18]: torch.normal(mean=0.5,
                      std=torch.arange(0.2,0.6))
Out[18]: tensor([0.7487])
```

Recipe 2-2. Variable Tensors

Problem

What is a variable in PyTorch and how is it defined? What is a random variable in PyTorch?

Solution

In PyTorch, the algorithms are represented as a computational graph. A variable is considered as a representation around the tensor object, corresponding gradients, and a reference to the function from where it was created. For simplicity, gradients are considered as slope of the function. The slope of the function can be computed by the derivative of the function with respect to the parameters that are present in the function. For example, in linear regression (Y = W*X + alpha), representation of the variable would look like the one shown in Figure 2-2.

Basically, a PyTorch variable is a node in a computational graph, which stores data and gradients. When training a neural network model, after each iteration, we need to compute the gradient of the loss function with respect to the parameters of the model, such as weights and biases. After that, we usually update the weights using the gradient descent algorithm. Figure 2-1 explains how the linear regression equation is deployed under the hood using a neural network model in the PyTorch framework.

In a computational graph structure, the sequencing and ordering of tasks is very important. The one-dimensional tensors are X, Y, W, and alpha in Figure 2-2. The direction of the arrows change when we implement backpropagation to update the weights to match with Y, so that the error or loss function between Y and predicted Y can be minimized.

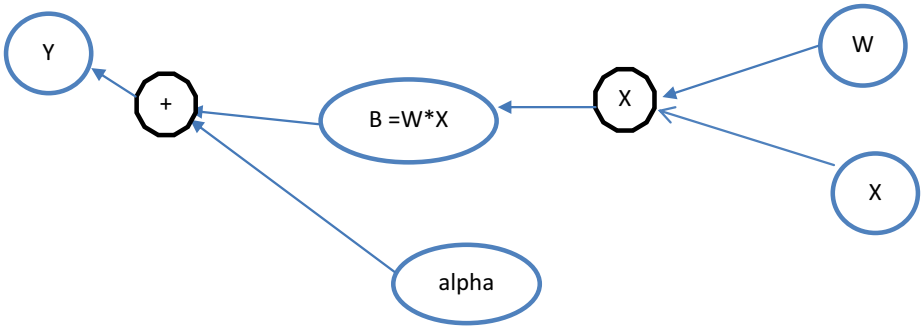

Figure 2-1. *A sample computational graph of a PyTorch implementation*

How It Works

An example of how a variable is used to create a computational graph is displayed in the following script. There are three variable objects around tensors— x1, x2, and x3—with random points generated from $a = 12$ and $b = 23$. The graph computation involves only multiplication and addition, and the final result with the gradient is shown.

The partial derivative of the loss function with respect to the weights and biases in a neural network model is achieved in PyTorch using the Autograd module. Variables are specifically designed to hold the changed values while running a backpropagation in a neural network model when the parameters of the model change. The variable type is just a wrapper around the tensor. It has three properties: data, grad, and function.

```
In [45]: from torch.autograd import Variable

In [47]: Variable(torch.ones(2,2),requires_grad=True)

Out[47]: tensor([[1., 1.],
                 [1., 1.]], requires_grad=True)

In [53]: a, b = 12,23
         x1 = Variable(torch.randn(a,b),
                       requires_grad=True)
         x2 = Variable(torch.randn(a,b),
                       requires_grad=True)
         x3 =Variable(torch.randn(a,b),
                       requires_grad=True)

In [66]: c = x1 * x2
         d = a + x3
         e = torch.sum(d)

         e.backward()

         print(e)

         tensor(3326.2632, grad_fn=<SumBackward0>)
```

Recipe 2-3. Basic Statistics

Problem

How do we compute basic statistics, such as mean, median, mode, and so forth, from a Torch tensor?

Solution

Computation of basic statistics using PyTorch enables the user to apply probability distributions and statistical tests to make inferences from data. Though the Torch functionality is like that of Numpy, Torch functions have GPU acceleration. Let's have a look at the functions to create basic statistics.

How It Works

The mean computation is simple to write for a 1D tensor; however, for a 2D tensor, an extra argument needs to be passed as a mean, median, or mode computation, across which the dimension needs to be specified.

```
In [19]: #computing the descriptive statistics: mean
         torch.mean(torch.tensor([10., 10., 13., 10., 34.,
                                  45., 65., 67., 87., 89., 87., 34.]))
Out[19]: tensor(45.9167)

In [20]: # mean across rows and across columns
         d = torch.randn(4, 5)
         d
Out[20]: tensor([[-0.2632, -0.5432, -1.6406,  0.9295, -1.2777],
                 [-0.7428,  0.9711,  0.3551,  0.8562, -0.3635],
                 [-0.1552, -1.2282, -0.8039, -0.4530, -0.2217],
                 [-2.0901, -1.2658, -1.8761, -0.6066,  0.7470]])

In [21]: torch.mean(d,dim=0)
Out[21]: tensor([-0.8128, -0.5165, -0.9914,  0.1815, -0.2789])

In [22]: torch.mean(d,dim=1)
Out[22]: tensor([-0.5590,  0.2152, -0.5724, -1.0183])
```

Median, mode, and standard deviation computation can be written in the same way.

```
In [23]: #compute median
         torch.median(d,dim=0)

Out[23]: (tensor([-0.7428, -1.2282, -1.6406, -0.4530, -0.3635]),
          tensor([1, 2, 0, 2, 1]))

In [24]: torch.median(d,dim=1)

Out[24]: (tensor([-0.5432,  0.3551, -0.4530, -1.2658]), tensor([1, 2, 3, 1]))

In [25]: # compute the mode
         torch.mode(d)

Out[25]: (tensor([-1.6406, -0.7428, -1.2282, -2.0901]), tensor([2, 0, 1, 0]))

In [26]: torch.mode(d,dim=0)

Out[26]: (tensor([-2.0901, -1.2658, -1.8761, -0.6066, -1.2777]),
          tensor([3, 3, 3, 3, 0]))

In [27]: torch.mode(d,dim=1)

Out[27]: (tensor([-1.6406, -0.7428, -1.2282, -2.0901]), tensor([2, 0, 1, 0]))
```

Standard deviation shows the deviation from the measures of central tendency, which indicates the consistency of the data/variable. It shows whether there is enough fluctuation in data or not.

```
In [28]: #compute the standard deviation
         torch.std(d)
Out[28]: tensor(0.9237)

In [29]: torch.std(d,dim=0)
Out[29]: tensor([0.8890, 1.0459, 1.0087, 0.8243, 0.8287])

In [30]: torch.std(d,dim=1)
Out[30]: tensor([0.9987, 0.7507, 0.4458, 1.1436])

In [31]: #compute variance
         torch.var(d)
Out[31]: tensor(0.8532)

In [32]: torch.var(d,dim=0)
Out[32]: tensor([0.7903, 1.0939, 1.0175, 0.6795, 0.6868])

In [33]: torch.var(d,dim=1)
Out[33]: tensor([0.9974, 0.5636, 0.1987, 1.3079])
```

Recipe 2-4. Gradient Computation

Problem

How do we compute basic gradients from the sample tensors using PyTorch?

Solution

We are going to consider a sample datase0074, where two variables (x and y) are present. With the initial weight given, can we computationally get the gradients after each iteration? Let's take a look at the example.

How It Works

`x_data` and `y_data` both are lists. To compute the gradient of the two data lists requires computation of a loss function, a forward pass, and running the stuff in a loop.

The forward function computes the matrix multiplication of the weight tensor with the input tensor.

```
In [50]: # Using forward pass
         def forward(x):
             return x * w
```

```
In [76]: import torch
         from torch.autograd import Variable

         x_data = [11.0, 22.0, 33.0]
         y_data = [21.0, 14.0, 64.0]

         w = Variable(torch.Tensor([1.0]),  requires_grad=True)  # Any random value

         # Before training
         print("predict (before training)",  4, forward(4).data[0])
```

```
predict (before training) 4 tensor(4.)
```

```
In [77]: # Using forward pass
         def forward(x):
             return x * w
```

```
In [78]: # define the Loss function
         def loss(x, y):
             y_pred = forward(x)
             return (y_pred - y) * (y_pred - y)
```

```
In [79]: # Run the Training Loop
         for epoch in range(10):
             for x_val, y_val in zip(x_data, y_data):
                 l = loss(x_val, y_val)
                 l.backward()
                 print("\tgrad: ", x_val, y_val, w.grad.data[0])
                 w.data = w.data - 0.01 * w.grad.data

                 # Manually set the gradients to zero after updating weights
                 w.grad.data.zero_()

             print("progress:", epoch, l.data[0])
```

```
        grad:  11.0 21.0 tensor(-220.)
        grad:  22.0 14.0 tensor(2481.6001)
        grad:  33.0 64.0 tensor(-51303.6484)
progress: 0 tensor(604238.8125)
        grad:  11.0 21.0 tensor(118461.7578)
        grad:  22.0 14.0 tensor(-671630.6875)
        grad:  33.0 64.0 tensor(13114108.)
progress: 1 tensor(39481139200.)
        grad:  11.0 21.0 tensor(-30279010.)
        grad:  22.0 14.0 tensor(171986000.)
        grad:  33.0 64.0 tensor(-3358889472.)
progress: 2 tensor(2590022582665216.)
        grad:  11.0 21.0 tensor(7755301376.)
        grad:  22.0 14.0 tensor(-44050112512.)
        grad:  33.0 64.0 tensor(860298674176.)
progress: 3 tensor(169906757784039325696.)
        grad:  11.0 21.0 tensor(-1986333900800.)
        grad:  22.0 14.0 tensor(11282376818688.)
        grad:  33.0 64.0 tensor(-220344807849984.)
progress: 4 tensor(11145967797036089329319936.)
        grad:  11.0 21.0 tensor(508751660449792.)
        grad:  22.0 14.0 tensor(-2889709562888192.)
        grad:  33.0 64.0 tensor(56436029183229952.)
progress: 5 tensor(731181229735636676902291243008.)
        grad:  11.0 21.0 tensor(-130304505987203072.)
        grad:  22.0 14.0 tensor(740129586448171008.)
```

In [80]:
```
# After training
print("predict (after training)",  4, forward(4).data[0])
```

```
predict (after training) 4 tensor(-92686910753578617489326080.)
```

The following program shows how to compute the gradients from a loss function using the variable method on the tensor.

In [102]:
```
from torch import FloatTensor
from torch.autograd import Variable

a = Variable(FloatTensor([5]))

weights = [Variable(FloatTensor([i]), requires_grad=True) for i in (12, 53, 91, 73)]

w1, w2, w3, w4 = weights

b = w1 * a
c = w2 * a
d = w3 * b + w4 * c
Loss = (10 - d)

Loss.backward()

for index, weight in enumerate(weights, start=1):
    gradient, *_ = weight.grad.data
    print(f"Gradient of w{index} w.r.t to Loss: {gradient}")
```

```
Gradient of w1 w.r.t to Loss: -455.0
Gradient of w2 w.r.t to Loss: -365.0
Gradient of w3 w.r.t to Loss: -60.0
Gradient of w4 w.r.t to Loss: -265.0
```

Recipe 2-5. Tensor Operations

Problem

How do we compute or perform operations based on variables such as matrix multiplication?

Solution

Tensors are wrapped within the variable, which has three properties: grad, volatile, and gradient.

How It Works

Let's create a variable and extract the properties of the variable. This is required to weight update process requires gradient computation. By using the mm module, we can perform matrix multiplication.

```
In [88]: x = Variable(torch.Tensor(4, 4).uniform_(-4, 5))
         y = Variable(torch.Tensor(4, 4).uniform_(-3, 2))
         # matrix multiplication
         z = torch.mm(x, y)
         print(z.size())

         torch.Size([4, 4])
```

The following program shows the properties of the variable, which is a wrapper around the tensor.

```
In [85]: z = Variable(torch.Tensor(4, 4).uniform_(-5, 5))
         print(z)

         tensor([[-2.6830,  2.0509, -2.6185, -3.6709],
                 [-4.9271,  3.6834,  4.0502,  3.4515],
                 [ 3.4576,  4.5814,  0.0632, -4.6377],
                 [-3.1634,  2.9827, -3.9532, -0.6395]])

In [86]: print('Requires Gradient : %s ' % (z.requires_grad))
         print('Volatile : %s ' % (z.volatile))
         print('Gradient : %s ' % (z.grad))
         print(z.data)

         Requires Gradient : False
         Volatile : False
         Gradient : None
         tensor([[-2.6830,  2.0509, -2.6185, -3.6709],
                 [-4.9271,  3.6834,  4.0502,  3.4515],
                 [ 3.4576,  4.5814,  0.0632, -4.6377],
                 [-3.1634,  2.9827, -3.9532, -0.6395]])
```

Recipe 2-6. Tensor Operations

Problem

How do we compute or perform operations based on variables such as matrix-vector computation, and matrix-matrix and vector-vector calculation?

Solution

One of the necessary conditions for the success of matrix-based operations is that the length of the tensor needs to match or be compatible for the execution of algebraic expressions.

How It Works

The tensor definition of a scalar is just one number. A 1D tensor is a vector, and a 2D tensor is a matrix. When it extends to an *n* dimensional level, it can be generalized to only tensors. When performing algebraic computations in PyTorch, the dimension of a matrix and a vector or scalar should be compatible.

```
In [103]: #tensor operations

In [109]: mat1 = torch.FloatTensor(4,4).uniform_(0,1)
          mat1

Out[109]: tensor([[0.7748, 0.5287, 0.8794, 0.0527],
                  [0.2824, 0.0036, 0.4439, 0.1425],
                  [0.7889, 0.8024, 0.2917, 0.9638],
                  [0.1534, 0.1216, 0.4023, 0.6097]])


In [110]: mat2 = torch.FloatTensor(5,4).uniform_(0,1)
          mat2

Out[110]: tensor([[0.9493, 0.1600, 0.8915, 0.7011],
                  [0.2557, 0.1726, 0.6665, 0.4602],
                  [0.9003, 0.5600, 0.0388, 0.3431],
                  [0.3064, 0.2131, 0.4333, 0.7120],
                  [0.1883, 0.4072, 0.2834, 0.8132]])


In [111]: vec1 = torch.FloatTensor(4).uniform_(0,1)
          vec1

Out[111]: tensor([0.8854, 0.9122, 0.9897, 0.2618])

In [112]: # scalar addition

In [113]: mat1 + 10.5

Out[113]: tensor([[11.2748, 11.0287, 11.3794, 10.5527],
                  [10.7824, 10.5036, 10.9439, 10.6425],
                  [11.2889, 11.3024, 10.7917, 11.4638],
                  [10.6534, 10.6216, 10.9023, 11.1097]])
```

```
In [114]: # scalar subtraction

In [115]: mat2 - 0.20

Out[115]: tensor([[ 0.7493, -0.0400,  0.6915,  0.5011],
                  [ 0.0557, -0.0274,  0.4665,  0.2602],
                  [ 0.7003,  0.3600, -0.1612,  0.1431],
                  [ 0.1064,  0.0131,  0.2333,  0.5120],
                  [-0.0117,  0.2072,  0.0834,  0.6132]])

In [116]: # vector and matrix addition

In [117]: mat1 + vec1

Out[117]: tensor([[1.6603, 1.4409, 1.8691, 0.3145],
                  [1.1678, 0.9158, 1.4336, 0.4042],
                  [1.6743, 1.7145, 1.2814, 1.2255],
                  [1.0388, 1.0337, 1.3920, 0.8715]])

In [118]: mat2 + vec1

Out[118]: tensor([[1.8348, 1.0722, 1.8812, 0.9629],
                  [1.1412, 1.0848, 1.6562, 0.7220],
                  [1.7857, 1.4721, 1.0285, 0.6049],
                  [1.1918, 1.1252, 1.4230, 0.9738],
                  [1.0738, 1.3193, 1.2731, 1.0750]])
```

Since the mat1 and the mat2 dimensions are different, they are not compatible for matrix addition or multiplication. If the dimension remains the same, we can multiply them. In the following script, the matrix addition throws an error when we multiply similar dimensions—mat1 with mat1. We get relevant results.

```
In [119]: # matrix-matrix addition

In [123]: mat1 + mat2

          ---------------------------------------------------------------------------
          --
          RuntimeError                              Traceback (most recent call las
          t)
          <ipython-input-123-bb4bbf5b2f84> in <module>()
          ----> 1 mat1 + mat2

          RuntimeError: The size of tensor a (4) must match the size of tensor b
           (5) at non-singleton dimension 0


In [131]: mat1 * mat1

Out[131]: tensor([[0.6004, 0.2795, 0.7733, 0.0028],
                  [0.0797, 0.0000, 0.1970, 0.0203],
                  [0.6224, 0.6438, 0.0851, 0.9288],
                  [0.0235, 0.0148, 0.1618, 0.3717]])
```

Recipe 2-7. Distributions

Problem

Knowledge of statistical distributions is essential for weight normalization, weight initialization, and computation of gradients in neural network–based operations using PyTorch. How do we know which distributions to use and when to use them?

Solution

Each statistical distribution follows a pre-established mathematical formula. We are going to use the most commonly used statistical distributions, their arguments in scenarios of problems.

How It Works

Bernoulli distribution is a special case of *binomial distribution*, in which the number of trials can be more than one; but in a Bernoulli distribution, the number of experiment or trial remains one. It is a discrete probability distribution of a random variable, which takes a value of 1 when there is probability that an event is a success, and takes a value of 0 when there is probability that an event is a failure. A perfect example of this is tossing a coin, where 1 is heads and 0 is tails. Let's look at the program.

```
In [ ]: # about Bernoulli distribution

In [138]: from torch.distributions.bernoulli import Bernoulli

In [139]: dist = Bernoulli(torch.tensor([0.3,0.6,0.9]))

In [140]: dist.sample() #sample is binary, it takes 1 with p and 0 with 1-p
Out[140]: tensor([1., 1., 1.])

In [132]: #Creates a Bernoulli distribution parameterized by probs
          #Samples are binary (0 or 1). They take the value 1 with probability p
          #and 0 with probability 1 - p.
```

The *beta distribution* is a family of continuous random variables defined in the range of 0 and 1. This distribution is typically used for Bayesian inference analysis.

```
In [133]: from torch.distributions.beta import Beta

In [141]: dist = Beta(torch.tensor([0.5]), torch.tensor([0.5]))
          dist
Out[141]: Beta()

In [142]: dist.sample()
Out[142]: tensor([0.3067])
```

The binomial distribution is applicable when the outcome is twofold and the experiment is repetitive. It belongs to the family of discrete probability distribution, where the probability of success is defined as 1 and the probability of failure is 0. The binomial distribution is used to model the number of successful events over many trials.

```
In [143]: from torch.distributions.binomial import Binomial

In [144]: dist = Binomial(100, torch.tensor([0 , .2, .8, 1]))

In [147]: dist.sample()
Out[147]: tensor([  0.,  29.,  85., 100.])

In [148]: # 100- count of trials
          # 0, 0.2, 0.8 and 1 are event probabilities
```

In probability and statistics, a categorical distribution can be defined as a generalized Bernoulli distribution, which is a discrete probability distribution that explains the possible results of any random variable that may take on one of the possible categories, with the probability of each category exclusively specified in the tensor.

```
In [174]: from torch.distributions.categorical import Categorical

In [175]: dist = Categorical(torch.tensor([ 0.20, 0.20, 0.20, 0.20, 0.20 ]))
          dist
Out[175]: Categorical()

In [176]: dist.sample()
Out[176]: tensor(4)

In [177]: # 0.20, 0.20, 0.20, 0.20,0.20 event probabilities
```

A *Laplacian distribution* is a continuous probability distribution function that is otherwise known as a *double exponential distribution*. A Laplacian distribution is used in speech recognition systems to understand prior probabilities. It is also useful in Bayesian regression for deciding prior probabilities.

```
In [155]: # Laplace distribution parameterized by loc and 'scale'.

In [157]: from torch.distributions.laplace import Laplace

In [161]: dist = Laplace(torch.tensor([10.0]), torch.tensor([0.990]))
          dist
Out[161]: Laplace()

In [162]: dist.sample()
Out[162]: tensor([10.2167])
```

A *normal distribution* is very useful because of the property of central limit theorem. It is defined by mean and standard deviations. If we know the mean and standard deviation of the distribution, we can estimate the event probabilities.

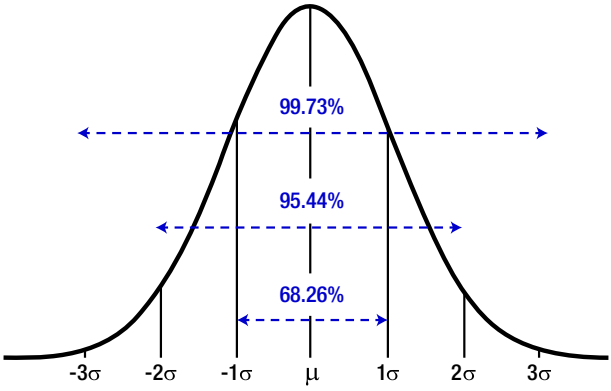

Figure 2-2. *Normal probability distribution*

```
In [163]: #Normal (Gaussian) distribution parameterized by loc and 'scale'.

In [165]: from torch.distributions.normal import Normal

In [166]: dist = Normal(torch.tensor([100.0]), torch.tensor([10.0]))
          dist
Out[166]: Normal()

In [167]: dist.sample()
Out[167]: tensor([95.2452])
```

Conclusion

This chapter discussed sampling distribution and generating random numbers from distributions. Neural networks are the primary focus in tensor-based operations. Any sort of machine learning or deep learning model implementation requires gradient computation, updating weight, computing bias, and continuously updating the bias.

This chapter also discussed the statistical distributions supported by PyTorch and the situations where each type of distribution can be applied.

The next chapter discusses deep learning models in detail. Those deep learning models include convolutional neural networks, recurrent neural networks, deep neural networks, and autoencoder models.