# 高级量化交易技术

**闫涛**
科技有限公司
北京 2021.05.08
{yt7589}@qq.com

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

# 第零篇深度学习

# 第 A01 章回归


## Abstract

在本章中我们将讨论回归问题，我们首先讨论用线性回归来处理简单问题，接着我们研究分段线性回归来处理复杂问题，然后我们对线性回归进行扩展，讨论多项式回归。


## 1 深度学习框架概述

在本章中，我们将研究回归问题，就是模型将生成一个数作为结果。

### 1.1 线性回归

在这里我们研究一个简单的线性回归问题，我们研究的问题是匀加速直线运动的速度问题。我们知道匀加速直线运动的速度公式为：

$$v = v_0 + at = 1.5 + 2.3 \times t \tag{1}$$

我们假设我们不知道这个公式，只有如下所示的数据：

表 1: 时间速度观察数据表

| 时间 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 速度 | 3.8 | 6.1 | 8.4 | 10.7 | 13.0 | 15.3 | 17.6 | 19.9 | 22.2 | 24.5 |

# 第 Z01 章深度学习框架


## Abstract

在本章中我们将利用 Numpy 开发一个小型的深度学习框架，实现类似 PyTorch 的功能。


## 2 深度学习框架概述

在本章中，我们将利用 Numpy，开发一个基于动态计算图的深度学习框架。

### 2.1 张量

在深度学习中，最基本的元素是张量（Tensor）。1 维张量就是我们所熟悉的向量，2 维张量就是矩阵，3 维及以上就是通用的张量。张量（Tensor）在 Numpy 中就用多维数组来表示。我们首先定义一个仅支持加法运算，但是支持自动微分的张量原型，在后面的章节在逐渐扩充完善。

#### 2.1.1 最简张量模型

我们首先定义一个最简单的张量，如下所示：

```
import numpy as np

class Tensor(object):
    def __init__(self, data, autograd=False, creators=None, creation_op=
    None, cid=None):
        self.data =np.array(data)
        self.creators = creators
        self.creation_op = creation_op
        self.autograd = autograd
        self.grad = None
        self.children = {}
        if (cid is None):
            cid = np.random.randint(0, 10000000)
        self.cid = cid
        if creators is not None:
            for c in creators:
                if self.cid not in c.children:
                    c.children[self.cid] = 1
                else:
                    c.children[self.cid] += 1

    def all_children_grads_accounted_for(self):
        for cid, cnt in self.children.items():
            if cnt != 0:
                return False
        return True

    def backward(self, grad=None, grad_origin=None):
        if self.autograd:
            if grad_origin is not None:
                if self.children[grad_origin.cid] == 0:
```

```
                    raise Exception('cannot backprop more than once')
                else:
                    self.children[grad_origin.cid] -= 1
            if self.grad is None:
                self.grad = grad
            else:
                self.grad += grad
            if self.creators is not None and (self.
    all_children_grads_accounted_for() or grad_origin is None):
                if self.creation_op == 'add':
                    self.creators[0].backward(self.grad, self)
                    self.creators[1].backward(self.grad, self)

    def __add__(self, other):
        if self.autograd and other.autograd:
            return Tensor(self.data + other.data, autograd=True, creators
    =[self, other], creation_op='add')
        return Tensor(self.data + other.data)

    def __repr__(self):
        return str(self.data.__repr__())

    def __str__(self):
        return str(self.data.__str__())

    def to_string(self):
        ts = 'tensor_{0}:\r\n'.format(self.cid)
        ts = '{0}    data:{1};\r\n'.format(ts, self.data)
        ts = '{0}    autograd: {1};\r\n'.format(ts, self.autograd)
        ts = '{0}    creators: {1};\r\n'.format(ts, self.creators)
        ts = '{0}    creation_op: {1};\r\n'.format(ts, self.creation_op)
        ts = '{0}    cid: {1};\r\n'.format(ts, self.cid)
        ts = '{0}    grad: {1};\r\n'.format(ts, self.grad)
        ts = '{0}    children: {1};\r\n'.format(ts, self.children)
        return ts
```

Listing 1: 策略迭代 (chpZ01/tensor.py)

直接看代码比较难以理解，下面我们先写一个单元测试用例，实现一个前向传播过程，如下所示：

```
class TTensor(unittest.TestCase):
    @classmethod
    def setUp(cls):
        pass

    @classmethod
    def tearDown(cls):
        pass

    def test_init_001(self):
```

```
        a = Tensor([1, 2, 3, 4, 5], autograd=True)
        b = Tensor([10, 10, 10, 10, 10], autograd=True)
        c = Tensor([5, 4, 3, 2, 1], autograd=True)
        d = a + b
        e = b + c
        f = d + e
        print('f: {0};'.format(f.to_string()))
        print('d: {0};'.format(d.to_string()))
        print('e: {0};'.format(e.to_string()))
        print('a: {0};'.format(a.to_string()))
        print('b: {0};'.format(b.to_string()))
        print('c: {0};'.format(c.to_string()))
```

Listing 2: 向量前向传播测试用例 (chpZ01/tensor.py)

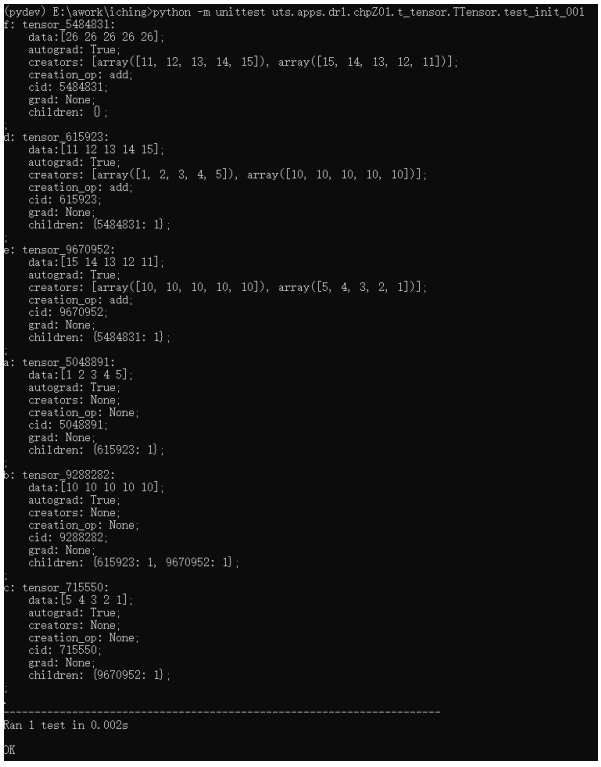
运行上面的测试用例:

```
python -m unittest uts.apps.drl.chpZ01.t_tensor.TTensor.test_init_001
```

Listing 3: 向量前向传播测试用例运行 (chpZ01/tensor.py)

运行结果如下所示: 其将形成如下所示的动态图: 接下来我们看一下反向传播过程, 首先来

图 1: 向量前向传播执行结果

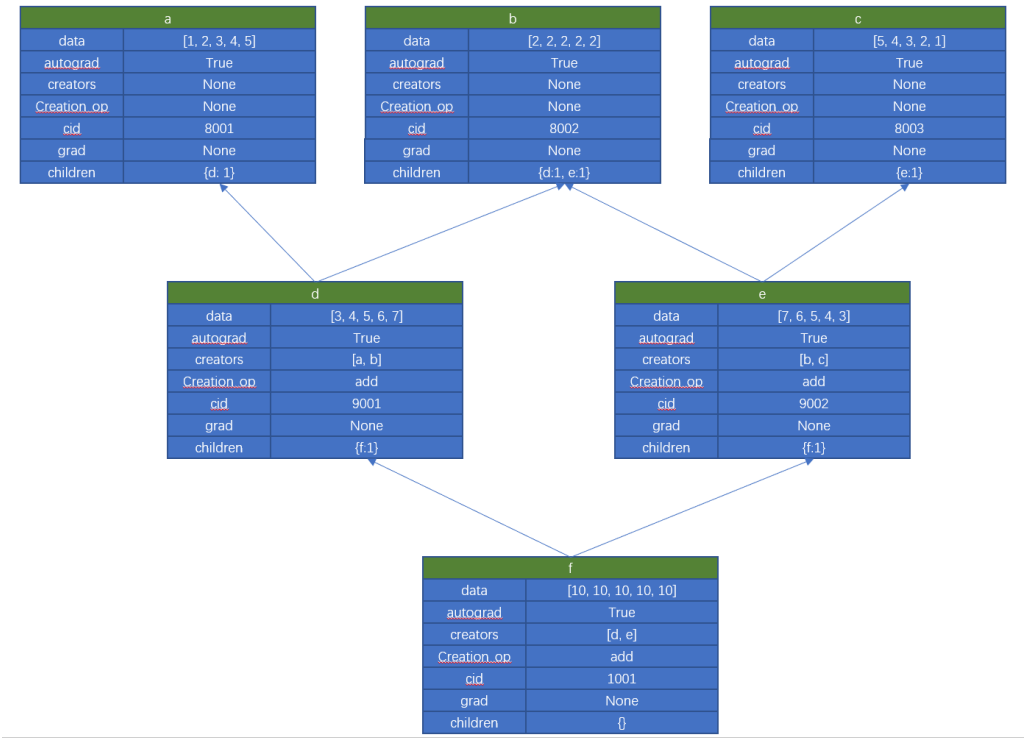



看测试用例:

```
class TTensor(unittest.TestCase):

    def test_add_backward_001(self):
```

图 2: 向量加法动态图示例




```
        a = Tensor([1, 2, 3, 4, 5], autograd=True)
        b = Tensor([10, 10, 10, 10, 10], autograd=True)
        c = Tensor([5, 4, 3, 2, 1], autograd=True)
        d = a + b
        e = b + c
        f = d + e
        f.backward(Tensor([1, 1, 1, 1, 1]))
        print('f: {0};'.format(f.to_string()))
        print('d: {0};'.format(d.to_string()))
        print('e: {0};'.format(e.to_string()))
        print('a: {0};'.format(a.to_string()))
        print('b: {0};'.format(b.to_string()))
        print('c: {0};'.format(c.to_string()))
```

Listing 4: 向量前向传播测试用例 (chpZ01/tensor.py)

我们首先来看理论分析，对于 $f = d + e$，我们先定 $\frac{\partial j}{\partial f} = [1, 1, 1, 1, 1]$，根据链式求导法则，得到 $\frac{\partial j}{\partial \boldsymbol{e}} = \frac{\partial j}{\partial \boldsymbol{f}} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{e}}$，向量对向量微分是得到的是 Jacobian 矩阵，如下所示：

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{e}} = \begin{bmatrix} \frac{\partial f_1}{\partial e_1} & \frac{\partial f_1}{\partial e_2} & \cdots & \frac{\partial f_1}{\partial e_n} \\ \frac{\partial f_2}{\partial e_1} & \frac{\partial f_2}{\partial e_2} & \cdots & \frac{\partial f_2}{\partial e_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_n}{\partial e_1} & \frac{\partial f_n}{\partial e_2} & \cdots & \frac{\partial f_n}{\partial e_n} \end{bmatrix} \in R^{n \times n} \tag{2}$$

其实际执行为 $R^{1 \times n} \cdot R^{n \times n} = R^{1 \times n}$，以上为理论分析，下面我来看程序上面的代码实现。对于向量 $\boldsymbol{f}$，我们直接指定 $\frac{\partial j}{\partial \boldsymbol{f}} = [1, 1, 1, 1, 1]$，向量 $\boldsymbol{f}$ 当前状态为：反向传播程序如下所示：

图 3: 向量 f 原始状态

| f | |
|---|---|
| data | [10, 10, 10, 10, 10] |
| autograd | True |
| creators | [d, e] |
| Creation_op | add |
| cid | 1001 |
| grad | None |
| children | {} |

```
1    def backward(self, grad=None, grad_origin=None):
2        if self.autograd:
3            if grad_origin is not None:
4                if self.children[grad_origin.cid] == 0:
5                    raise Exception('cannot backprop more than once')
6                else:
7                    self.children[grad_origin.cid] -= 1
8            if self.grad is None:
9                self.grad = grad
10           else:
11               self.grad += grad
12           if self.creators is not None and (self.
    all_children_grads_accounted_for() or grad_origin is None):
13               if self.creation_op == 'add':
14                   self.creators[0].backward(self.grad, self)
15                   self.creators[1].backward(self.grad, self)
```

Listing 5: 向量前向传播测试用例 (chpZ01/tensor.py)

代码解读如下所示：

- 第 2 行：因为 autograd 为 True，所以执行下面的代码；
- 第 3~7 行：因为 grad_origin 为 False，所以不执行；
- 第 8~11 行：因为 grad 为 None，因此执行第 9 行使 $grad = [1, 1, 1, 1, 1]$；
- 第 12 行：因为 creators 为 d 和 e，且 all_children_grads_accounted_for 为 True，且 grad_origin 为 None，因此会执行下面的语句；
- 第 13~15 行：由于是 add 运算，以自己的 grad 和自身为参数，分别调用 d 和 e 的后向传播算法；

执行完上述代码后，系统状态如下所示：我们先来看传到 d 节点，这时 grad_origin 为 f，此时 d 的 children 只有一个节点 f，初始时 children[f.cid]=1，运行后 children[f.cid]=0，因为 self.grad 为空，所以 self.grad=[1, 1, 1, 1, 1]，因为 self.creators=[a, b]，又为 add 操作，所以会调用 a 和 b 的反向传播算法。经过 a 和 b 的反向传播算法处理后，结果如下所示：我们再来看传到 e 节点，这时 grad_origin 为 f，此时 d 的 children 只有一个节点 f，初始时 children[f.cid]=1，运行后 children[f.cid]=0，因为 self.grad 为空，所以 self.grad=[1, 1, 1, 1, 1]，因为 self.creators=[a,

图 4: 反向传播从 f 到 d 和 e

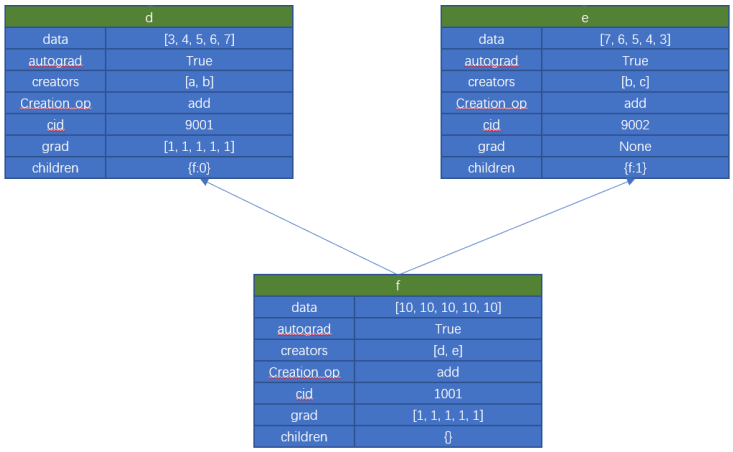
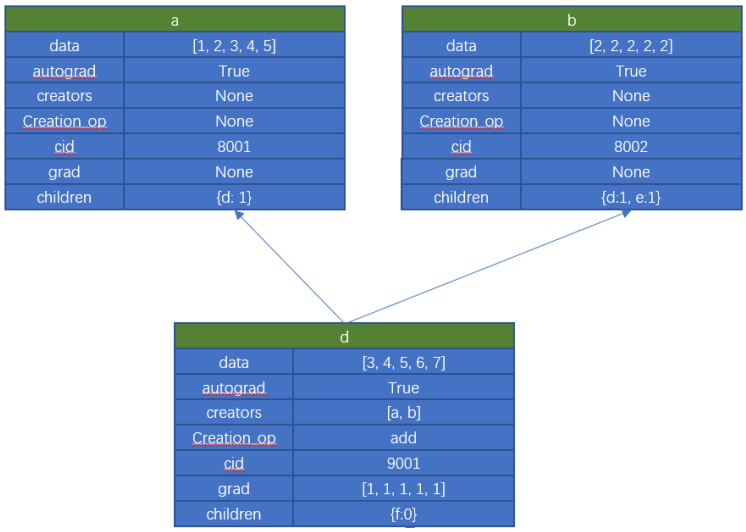



图 5: 反向传播从 d 到 a 和 b




b]，又为 add 操作，所以会调用 b 和 c 的反向传播算法。经过 b 和 c 的反向传播算法处理后，结果如下所示：

### 2.1.2 负号

我们首先定义负号方法：

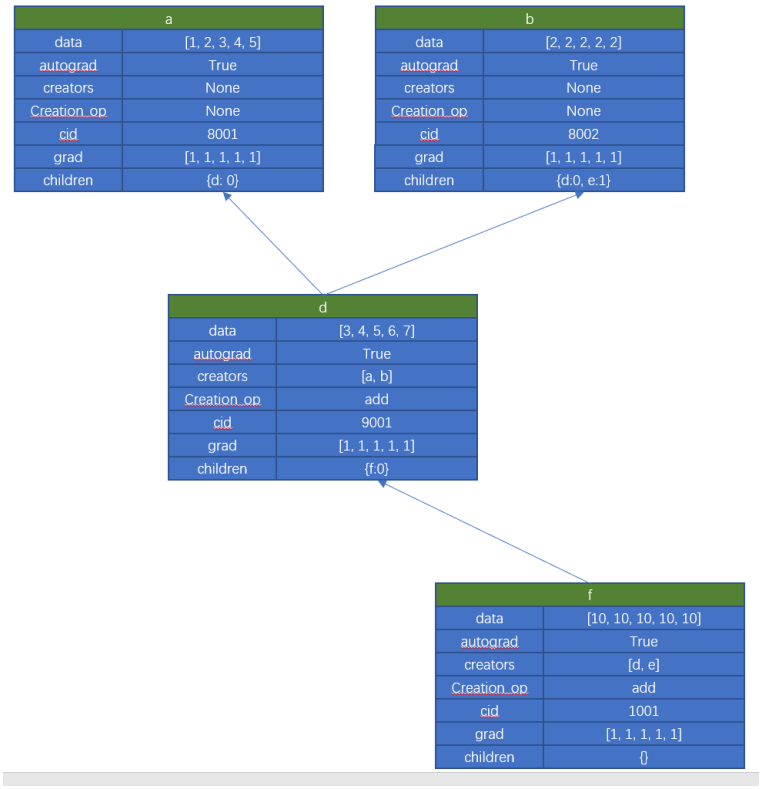
```
1    def __neg__(self):
2        if self.autograd:
3            return Tensor(self.data * -1, autograd=True, creators=[self],
     creation_op='neg')
4        return Tensor(self.data * -1)
```

Listing 6: 张量负号方法 (chpZ01/tensor.py)

其后向传播定义为：

```
1    def backward(self, grad=None, grad_origin=None):
2        if self.autograd:
3            if grad_origin is not None:
```

图 6: 反向传播从 d 到 a 和 b 后的处理结果




```
                if self.children[grad_origin.cid] == 0:
                    raise Exception('cannot backprop more than once')
                else:
                    self.children[grad_origin.cid] -= 1
            if self.grad is None:
                self.grad = grad
            else:
                self.grad += grad
            if self.creators is not None and (self.
    all_children_grads_accounted_for() or grad_origin is None):
                if self.creation_op == 'add':
                    self.creators[0].backward(self.grad, self)
                    self.creators[1].backward(self.grad, self)
                elif self.creation_op == 'neg':
                    self.creators[0].backward(self.grad.__neg__())
```

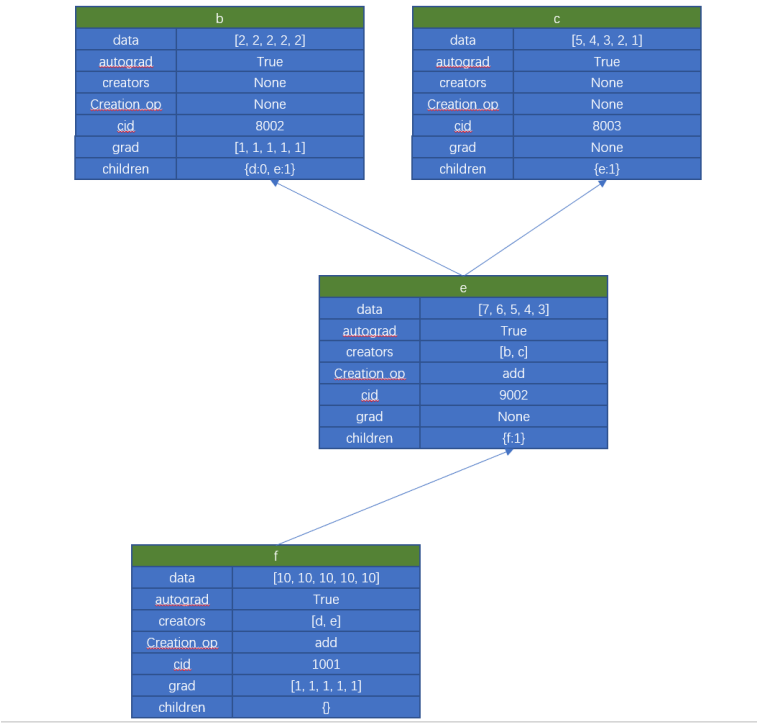
Listing 7: 张量负号反向传播 (chpZ01/tensor.py)

### 2.1.3 减法

我们首先定义减法:

```
    def __sub__(self, other):
        if self.autograd:
            return Tensor(self.data - other.data, autograd=True, creators
    =[self, other], creation_op='sub')
```

图 7: 反向传播从 e 到 b 和 c



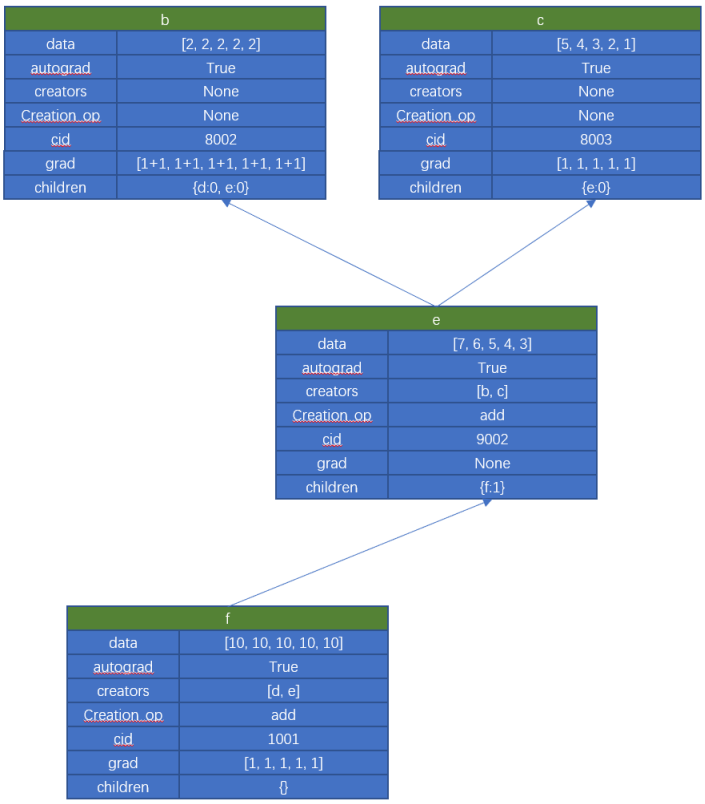

```
4         return Tensor(self.data - other.data)
```

Listing 8: 张量减法 (chpZ01/tensor.py)

减法的后向传播定义为:

```
1     def backward(self, grad=None, grad_origin=None):
2         if self.autograd:
3             if grad_origin is not None:
4                 if self.children[grad_origin.cid] == 0:
5                     raise Exception('cannot backprop more than once')
6                 else:
7                     self.children[grad_origin.cid] -= 1
8             if self.grad is None:
9                 self.grad = grad
10            else:
11                self.grad += grad
12            if self.creators is not None and (self.
    all_children_grads_accounted_for() or grad_origin is None):
13                if self.creation_op == 'add':
14                    self.creators[0].backward(self.grad, self)
15                    self.creators[1].backward(self.grad, self)
16                ......
17                elif self.creation_op == 'sub':
18                    org = Tensor(self.grad.data)
19                    self.creators[0].backward(org, self)
20                    org = Tensor(self.grad.__neg__().data)
```

图 8: 反向传播从 e 到 b 和 c 后的处理结果




```
21                 self.creators[1].backward(org, self)
```

Listing 9: 张量减法后台传播 (chpZ01/tensor.py)

### 2.1.4 乘法

我们首先定义乘法，这里的乘法是指对 * 运算符的重载，就是两个数组对应元素相乘（Elementwise Multiplication），不是我们将在后面介绍的矩阵和向量的乘法：

```
1     def __mul__(self, other):
2         if self.autograd and other.autograd:
3             return Tensor(self.data * other.data, autograd=True, creators
    =[self, other], creation_op='mul')
4         return Tensor(self.data * other.data)
```

Listing 10: 张量乘法 (chpZ01/tensor.py)

乘法反向传播算法：

```
1     def backward(self, grad=None, grad_origin=None):
2         if self.autograd:
3             if grad_origin is not None:
4                 if self.children[grad_origin.cid] == 0:
5                     raise Exception('cannot backprop more than once')
6                 else:
7                     self.children[grad_origin.cid] -= 1
8             if self.grad is None:
```

```
                self.grad = grad
            else:
                self.grad += grad
            if self.creators is not None and (self.
    all_children_grads_accounted_for() or grad_origin is None):
                if self.creation_op == 'add':
                    self.creators[0].backward(self.grad, self)
                    self.creators[1].backward(self.grad, self)
                ......
                elif self.creation_op == 'mul':
                    rst = self.grad * self.creators[1]
                    self.creators[0].backward(rst, self)
                    rst = self.grad * self.creators[0]
                    self.creators[1].backward(rst, self)
```

Listing 11: 张量乘法反向传播 (chpZ01/tensor.py)

从数学概念上来看，两个向量 Elementwise 相乘之后，得到的是同维度的向量，再求出的微分是一个矩阵，如下所示：

$$\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b} = \begin{bmatrix} c_1 & c2 & ... & c_n \end{bmatrix} = \begin{bmatrix} a_1 * b_1 & a_2 * b_2 & ... & an * bn \end{bmatrix} \in R^n \tag{3}$$

我们要求的 $\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{a}}$ 是一个向量对向量的微分，其为一个 $R^{n \times n}$ 的矩阵，如下所示：

$$\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{a}} = \begin{bmatrix} \frac{\partial c_1}{\partial a_1} & \frac{\partial c_1}{\partial a_2} & ... & \frac{\partial c_1}{\partial a_n} \\ \frac{\partial c_2}{\partial a_1} & \frac{\partial c_2}{\partial a_2} & ... & \frac{\partial c_2}{\partial a_n} \\ ... & ... & ... & ... \\ \frac{\partial c_n}{\partial a_1} & \frac{\partial c_n}{\partial a_2} & ... & \frac{\partial c_n}{\partial a_n} \end{bmatrix} = \begin{bmatrix} b_1 & 0 & ... & 0 \\ 0 & b_2 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & b_n \end{bmatrix} \in R^{n \times n} \tag{4}$$

我们上面的结果就是这个矩阵对角线的值。

### 2.1.5 sum 运算

我们首先来看 sum 运算，其有一个参数 dim，表明要对哪维操作（即消除哪维），例如 v.sum(0)，就是消除行，如下所示：

$$sum(v, 0) = sum(\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 6 & 7 & 9 & 10 & 11 \end{bmatrix}, 0) = \begin{bmatrix} 9 & 12 & 16 & 19 & 22 \end{bmatrix} \tag{5}$$

同理 sum(1) 如下所示：

$$sum(v, 0) = sum(\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 6 & 7 & 8 & 9 & 10 \end{bmatrix}, 0) = \begin{bmatrix} 15 \\ 20 \\ 40 \end{bmatrix} \tag{6}$$

接下来我们来看 expand 操作，其有两个操作，第 1 个参数是扩展哪维，第 2 个参数是重复几份，如下所示：

```
org: [
    [1, 2, 3],
    [4, 5, 6]
]
```

```
expand(0, 4):
[
    [
        [1, 2, 3],
        [4, 5, 6]
    ],
    [
        [1, 2, 3],
        [4, 5, 6]
    ],
    [
        [1, 2, 3],
        [4, 5, 6]
    ],
    [
        [1, 2, 3],
        [4, 5, 6]
    ]
]
expand(1, 4):
[
    [
        [1, 1, 1, 1],
        [2, 2, 2, 2],
        [3, 3, 3, 3]
    ],
    [
        [4, 4, 4, 4],
        [5, 5, 5, 5],
        [6, 6, 6, 6]
    ]
]
```

Listing 12: 张量 expand 示例 (chpZ01/tensor.py)

下面我们来看具体的代码实现:

```
    def sum(self, dim):
        if self.autograd:
            return Tensor(self.data.sum(dim), autograd=True, creators=[
    self], creation_op='sum_' + str(dim))
        return Tensor(self.data.sum(dim))

    def expand(self, dim, copies):
        trans_cmd = list(range(0, len(self.data.shape)))
        trans_cmd.insert(dim, len(self.data.shape))
        new_shape = list(self.data.shape) + [copies]
        new_data = self.data.repeat(copies).reshape(new_shape)
        new_data = new_data.transpose(trans_cmd)
        if self.autograd:
```

```
13            return Tensor(new_data, autograd=True, creators=[self],
    creation_op='expand_'+str(dim))
14        return Tensor(new_data)
```

Listing 13: 张量 expand 示例 (chpZ01/tensor.py)

代码解读如下所示：

- 第 1~4 行：定义 sum 运算，其直接调用 numpy 的 sum 函数，比较简单；
- 第 6 行：定义 expand 运算，我们以上例中 $v = [[1,2,3],[4,5,6]] \in R^{2\times 3}$ 为例进行讲解，我们先来看 $dim = 0, copies = 4$ 的情况；
- 第 7 行：$trans_cmd$ 为 $[0,1]$；
- 第 8 行：$trans_cmd$ 为 $[2,0,1]$，因为 $len(self.data.shape)$ 的值为 2，当 $dim = 0$ 时，在列表最前面插入；
- 第 9 行：$new_shape$ 为 $[2,3,4]$；
- 第 10 行：$self.data.repeat(4)$ 后为 $R^{24}$ 的数组 $[111122223333444455556666]$，将其重新定义为 $R^{2\times 3\times 4}$，其值如下所示：

```
1   [
2       [
3           [1  1  1  1]
4           [2  2  2  2]
5           [3  3  3  3]
6       ]
7       [
8           [4  4  4  4]
9           [5  5  5  5]
10          [6  6  6  6]
11      ]
12  ]
13
```

Listing 14: 张量 expand 代码解读 1(chpZ01/tensor.py)

- 第 11 行：将最后 1 维放到前面，如下所示：

```
1   [
2       [
3           [1  2  3]
4           [4  5  6]
5       ]
6       [
7           [1  2  3]
8           [4  5  6]
9       ]
10      [
11          [1  2  3]
12          [4  5  6]
13      ]
14      [
```

```
            [1  2  3]
            [4  5  6]
        ]
    ]

```

Listing 15: 张量 expand 代码解读 1(chpZ01/tensor.py)

- 第 6 行：定义 expand 运算，我们以上例中 $v = [[1, 2, 3], [4, 5, 6]] \in R^{2\times3}$ 为例进行讲解，我们先来看 $dim = 1, copies = 4$ 的情况；
- 第 7 行：$trans_cmd$ 为 $[0, 1]$；
- 第 8 行：$trans_cmd$ 为 $[0, 2, 1]$，因为 $len(self.data.shape)$ 的值为 2，当 $dim = 0$ 时，在列表最前面插入；
- 第 9 行：$new_shape$ 为 $[2, 3, 4]$；
- 第 10 行：$self.data.repeat(4)$ 后为 $R^{24}$ 的数组 $[111122223333444455556666]$，将其重新定义为 $R^{2\times3\times4}$，其值如下所示：

```
    [
        [
            [1  1  1  1]
            [2  2  2  2]
            [3  3  3  3]
        ]
        [
            [4  4  4  4]
            [5  5  5  5]
            [6  6  6  6]
        ]
    ]

```

Listing 16: 张量 expand 代码解读 1(chpZ01/tensor.py)

- 第 11 行：将最后 1 维放到中间 $R^{2\times4\times3}$，如下所示：

```
    [
        [
            [1  2  3]
            [1  2  3]
            [1  2  3]
            [1  2  3]
        ]
        [
            [4  5  6]
            [4  5  6]
            [4  5  6]
            [4  5  6]
        ]
    ]

```

Listing 17: 张量 expand 代码解读 1(chpZ01/tensor.py)

我们用符号来表示这一过程 $u = sum(\boldsymbol{v}, 0)$ 为例：

$$u = v_1 + v_2 + v_3 + v4 + v5 = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix} \tag{7}$$

注意：上面是行向量，只是一行写不下而折行显示。我们来看 $\frac{u}{\boldsymbol{v}}$，根据定义其是一个行向量，形状为 $R^{1\times n}$，如下所示：

$$\begin{aligned} \frac{\partial u}{\partial \boldsymbol{v}} &= \begin{bmatrix} \frac{\partial u}{\partial v_1} & \frac{\partial u}{\partial v_2} & \frac{\partial u}{\partial v_3} & \frac{\partial u}{\partial v_4} & \frac{\partial u}{\partial v_5} \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \end{bmatrix} \end{aligned} \tag{8}$$

我们来看更一般的情况：

$$\boldsymbol{u} = \begin{bmatrix} v_{1,1} + v_{1,2} + v_{1,3} \\ v_{2,1} + v_{2,2} + v_{2,3} \end{bmatrix} \tag{9}$$

对其进行求解：

$$\frac{\partial \boldsymbol{u}}{\partial V} = \begin{bmatrix} \frac{\boldsymbol{u}}{V_{1,1}} & \frac{\boldsymbol{u}}{V_{1,2}} & \frac{\boldsymbol{u}}{V_{1,1}} \\ \frac{\boldsymbol{u}}{V_{2,1}} & \frac{\boldsymbol{u}}{V_{2,2}} & \frac{\boldsymbol{u}}{V_{2,3}} \end{bmatrix} \tag{10}$$

对于其中的每一项来说，是向量对标量的导数，其仍为一个向量，如下所示：

$$\frac{\partial \boldsymbol{u}}{\partial V_{1,1}} = \begin{bmatrix} \frac{\partial u_1}{\partial V_{1,1}} \\ \frac{\partial u_2}{\partial V_{1,1}} \end{bmatrix} = \begin{bmatrix} \frac{\partial v_{1,1}+v_{1,2}+v_{1,3}}{\partial V_{1,1}} \\ \frac{\partial v_{2,1}+v_{2,2}+v_{2,3}}{\partial V_{1,1}} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{11}$$

我们为了简化计算的需要，通常将式**??**视为全 1 的矩阵，但是实际上，其是一个每个元素均为只有一个元素为 1 其余元素为 0 的向量。求导程序如下所示：

```
    def backward(self, grad=None, grad_origin=None):
        if self.autograd:
            if grad_origin is not None:
                if self.children[grad_origin.cid] == 0:
                    raise Exception('cannot backprop more than once')
                else:
                    self.children[grad_origin.cid] -= 1
            if self.grad is None:
                self.grad = grad
            else:
                self.grad += grad
            if self.creators is not None and (self.
    all_children_grads_accounted_for() or grad_origin is None):
                if self.creation_op == 'add':
                    self.creators[0].backward(self.grad, self)
                    self.creators[1].backward(self.grad, self)
                ......
                elif 'sum' in self.creation_op:
                    dim = int(self.creation_op.split('_')[1])
                    ds = self.creators[0].data.shape[dim]
                    self.creators[0].backward(self.grad.expand(dim, ds))
```

```
                elif 'expand' in self.creation_op:
                    dim = int(self.creation_op.split('_')[1])
                    self.creators[0].backward(self.grad.sum(dim))
# testing program
def test_sum_grad_001(self):
    v = Tensor(np.array([
        [1, 2, 3],
        [4, 5, 6]
    ]), autograd=True)
    u = v.sum(0)
    u.backward(Tensor(np.array([1, 1, 1])))
    print('grad: {0};'.format(v.to_string()))
```

Listing 18: 张量 expand 示例 (chpZ01/tensor.py)

运行结果如下所示：

```
grad: tensor_2961896:
    data:[[1 2 3]
 [4 5 6]];
    autograd: True;
    creators: None;
    creation_op: None;
    cid: 2961896;
    grad: [[1 1 1]
 [1 1 1]];
    children: {9624403: 1};
```

Listing 19: 张量 expand 示例 (chpZ01/tensor.py)

### 2.1.6 转置运算

下面我们来看转置运算：

```
    def transpose(self):
        if self.autograd:
            return Tensor(self.data.transpose(), autograd=True, creators
    =[self], creation_op='transpose')
        return Tensor(self.data.transpose())

        def backward(self, grad=None, grad_origin=None):
        if self.autograd:
            if grad_origin is not None:
                if self.children[grad_origin.cid] == 0:
                    raise Exception('cannot backprop more than once')
                else:
                    self.children[grad_origin.cid] -= 1
            if self.grad is None:
                self.grad = grad
            else:
                self.grad += grad
            if self.creators is not None and (self.
    all_children_grads_accounted_for() or grad_origin is None):
```

```
                if self.creation_op == 'add':
                    self.creators[0].backward(self.grad, self)
                    self.creators[1].backward(self.grad, self)
                ......
                elif 'transpose' in self.creation_op:
                    self.creators[0].backward(self.grad.transpose())

def test_transpose_001(self):
    v = Tensor(np.array([
        [1, 2, 3],
        [4, 5, 6]
    ]), autograd=True)
    v_t = v.transpose()
    print('v_t: \r\n{0};'.format(v_t))
    v_t.backward(Tensor(np.array([[1, 1], [1, 1], [1, 1]])))
    print('grad v: \r\n{0};'.format(v.grad))
```

Listing 20: 张量 expand 示例 (chpZ01/tensor.py)

运行结果如下所示：

```
v_t:
[[1  4]
    [2  5]
    [3  6]];
grad v:
[[1  1  1]
    [1  1  1]];
```

Listing 21: 张量 expand 示例 (chpZ01/tensor.py)

### 2.1.7 矩阵乘法

下面我们来看矩阵乘法，在深度学习中，矩阵乘法最大的应用场景是第 $l-1$ 层的激活值 $\boldsymbol{a}^{l-1}$ 与第 $l$ 层的连接权值 $W \in R^{N_l \times N_{l-1}}$ 相乘得到第 $l$ 层的输入值 $\boldsymbol{z}^l$，如下所示：

$$\boldsymbol{z}^l = W \cdot \boldsymbol{a}^{l-1} \tag{12}$$

下面我们来举一个具体的例子，我们假设第一层的输出 $\boldsymbol{a}^1 \in R^3$，第 2 层的神经元数量为 4，我们用 $W^2 \in R^{4\times 3}$ 来表示第 1 层到第 2 层的连接权值，其中 $W_{i,j}$ 代表第 1 层的第 $j$ 个神经元指向第 2 层第 $i$ 个神经元的连接权值，我们用 $\boldsymbol{z} \in R^4$ 来表示第 2 层的输入信号，具体数据如下所示：

$$\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \\ W_{4,1} & W_{4,2} & W_{4,3} \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 11.0 & 21.0 & 31.0 \\ 12.0 & 22.0 & 32.0 \\ 13.0 & 23.0 & 33.0 \\ 14.0 & 24.0 & 34.0 \end{bmatrix} \cdot \begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \end{bmatrix} = \begin{bmatrix} 146.0 \\ 152.0 \\ 158.0 \\ 164.0 \end{bmatrix} \tag{13}$$

我们来看 $\frac{\partial \boldsymbol{z}^2}{\partial \boldsymbol{a}^1}$ 是向量对向量的微分，其值为一个 Jacobian 矩阵，如下所示：

$$\frac{\partial \boldsymbol{z}^2}{\partial \boldsymbol{a}^1} = \begin{bmatrix} \frac{\partial z_1^2}{\partial a_1^1} & \frac{\partial z_1^2}{\partial a_2^1} & \frac{\partial z_1^2}{\partial a_3^1} \\ \frac{\partial z_2^2}{\partial a_1^1} & \frac{\partial z_2^2}{\partial a_2^1} & \frac{\partial z_2^2}{\partial a_3^1} \\ \frac{\partial z_3^2}{\partial a_1^1} & \frac{\partial z_3^2}{\partial a_2^1} & \frac{\partial z_3^2}{\partial a_3^1} \\ \frac{\partial z_4^2}{\partial a_1^1} & \frac{\partial z_4^2}{\partial a_2^1} & \frac{\partial z_4^2}{\partial a_3^1} \end{bmatrix} = \begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \\ W_{4,1} & W_{4,2} & W_{4,3} \end{bmatrix} \tag{14}$$

我们假设第 2 层即为输出层，并假设其微分为：

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^2} = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \tag{15}$$

综上可得：

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^1} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^2} \frac{\partial \boldsymbol{z}^2}{\partial \boldsymbol{a}^1} = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 11.0 & 21.0 & 31.0 \\ 12.0 & 22.0 & 32.0 \\ 13.0 & 23.0 & 33.0 \\ 14.0 & 24.0 & 34.0 \end{bmatrix} = \begin{bmatrix} 50 & 90 & 130 \end{bmatrix} \tag{16}$$

我们再来看 $\frac{\partial \boldsymbol{z}^2}{\partial W^2}$，这是一个向量对矩阵的微分，根据我们的微分规则，微分结果是分子的维数乘以分母的维数，在本例中 $\boldsymbol{z}^2 \in R^4$ 且 $W^2 \in R^{4\times 3}$，则 $\frac{\partial \boldsymbol{z}^2}{\partial W^2} \in R^{4\times 4\times 3}$，既其主体为一个有 4 个元素的行向量，而该行向量中的元素为 $R^{4\times 3}$ 的矩阵，下面我们以第 1 个元素 $z_1^2$ 为例，我们首先根据前向传播过程，得到该元素的表达式：

$$z_1^2 = W_{1,:}^2 \cdot \boldsymbol{a}^1 = W_{1,1}^2 a^1 1 + W_{1,2}^2 a^1 2 + W_{1,3}^2 a^1 3 \tag{17}$$

其是一个标量，我们接下来要求一个标量对矩阵的微分，根据我们的微分规则，其形状为 $R^{3\times 4}$，如下所示：

$$\frac{\partial z_1^2}{\partial W^2} = \begin{bmatrix} \frac{\partial z_1^2}{\partial W_{1,1}^2} & \frac{\partial z_1^2}{\partial W_{2,1}^2} & \frac{\partial z_1^2}{\partial W_{3,1}^2} & \frac{\partial z_1^2}{\partial W_{4,1}^2} \\ \frac{\partial z_1^2}{\partial W_{1,2}^2} & \frac{\partial z_1^2}{\partial W_{2,2}^2} & \frac{\partial z_1^2}{\partial W_{3,2}^2} & \frac{\partial z_1^2}{\partial W_{4,2}^2} \\ \frac{\partial z_1^2}{\partial W_{1,3}^2} & \frac{\partial z_1^2}{\partial W_{2,3}^2} & \frac{\partial z_1^2}{\partial W_{3,3}^2} & \frac{\partial z_1^2}{\partial W_{4,3}^2} \end{bmatrix} = \begin{bmatrix} a_1^1 & 0 & 0 & 0 \\ a_2^1 & 0 & 0 & 0 \\ a_3^1 & 0 & 0 & 0 \end{bmatrix} \tag{18}$$

同理可得第 2 个元素为：

$$\frac{\partial z_1^2}{\partial W^2} = \begin{bmatrix} \frac{\partial z_1^2}{\partial W_{1,1}^2} & \frac{\partial z_1^2}{\partial W_{2,1}^2} & \frac{\partial z_1^2}{\partial W_{3,1}^2} & \frac{\partial z_1^2}{\partial W_{4,1}^2} \\ \frac{\partial z_1^2}{\partial W_{1,2}^2} & \frac{\partial z_1^2}{\partial W_{2,2}^2} & \frac{\partial z_1^2}{\partial W_{3,2}^2} & \frac{\partial z_1^2}{\partial W_{4,2}^2} \\ \frac{\partial z_1^2}{\partial W_{1,3}^2} & \frac{\partial z_1^2}{\partial W_{2,3}^2} & \frac{\partial z_1^2}{\partial W_{3,3}^2} & \frac{\partial z_1^2}{\partial W_{4,3}^2} \end{bmatrix} = \begin{bmatrix} 0 & a_1^1 & 0 & 0 \\ 0 & a_2^1 & 0 & 0 \\ 0 & a_3^1 & 0 & 0 \end{bmatrix} \tag{19}$$

我们可以以此类推下去。我们要求 $\frac{\partial \mathcal{L}}{\partial W^2}$ 为：

$$\frac{\partial \mathcal{L}}{\partial W^2} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^2} \frac{\partial \boldsymbol{z}^2}{\partial W^2} \in R^{1\times 3\times 4} \tag{20}$$

我们可以先做 $R^{1\times 4} \cdot R^{4\times 1}$ 的点击，其中 $R^{4\times 1}$ 的每个元素是一个 $R^{3\times 4}$，就是先做标量乘矩阵，然后再做矩阵相加，最终得到 $R^{1\times 3\times 4}$ 的结果：

$$\frac{\partial \mathcal{L}}{\partial W^2} = \begin{bmatrix} \begin{bmatrix} a_1^1 & a_1^1 & a_1^1 & a_1^1 \\ a_2^1 & a_2^1 & a_2^1 & a_2^1 \\ a_3^1 & a_3^1 & a_3^1 & a_3^1 \end{bmatrix} \end{bmatrix} \tag{21}$$

需要注意的是，我们上述安排，主要是为了便于进行链式微分运算，得到的微分结果，需要对第 1 维求和求平均去掉第 1 维后，然后进行一下转置，才能作为梯度调整来使用。有了上述的理论知识，我们来看具体的代码实现：

```
    def backward(self,grad=None, grad_origin=None):
        if(self.autograd):
            if(grad is None):
                grad = Tensor(np.ones_like(self.data))
            if(grad_origin is not None):
                if(self.children[grad_origin.id] == 0):
                    raise Exception("cannot backprop more than once")
                else:
                    self.children[grad_origin.id] -= 1
            if(self.grad is None):
                self.grad = grad
            else:
                self.grad += grad
            # grads must not have grads of their own
            assert grad.autograd == False
            # only continue backpropping if there's something to
            # backprop into and if all gradients (from children)
            # are accounted for override waiting for children if
            # "backprop" was called on this variable directly
            if(self.creators is not None and
               (self.all_children_grads_accounted_for() or
                grad_origin is None)):

                if(self.creation_op == "add"):
                    self.creators[0].backward(self.grad, self)
                    self.creators[1].backward(self.grad, self)
                ......
                if(self.creation_op == "mm"):
                    c0 = self.creators[0]
                    c1 = self.creators[1]
                    new = self.grad.mm(c1.transpose())
                    c0.backward(new)
                    new = self.grad.transpose().mm(c0).transpose()
                    c1.backward(new)


    def mm(self, x):
        if(self.autograd):
            return Tensor(self.data.dot(x.data),
                          autograd=True,
                          creators=[self,x],
                          creation_op="mm")
        return Tensor(self.data.dot(x.data))

    # testcase
    def test_mm_001(self):
```

```
        a1 = Tensor(np.array([[1.0], [2.0], [3.0]]), autograd=True)
        w_2 = Tensor(np.array([
            [11.0, 21.0, 31.0],
            [12.0, 22.0, 32.0],
            [13.0, 23.0, 33.0],
            [14.0, 24.0, 34.0]
        ]), autograd=True)
        z2 = w_2.mm(a1)
        print('z2: {0};\r\n{1}'.format(z2.data.shape, z2))
        z2.backward(Tensor(np.ones_like(z2.data)))
        print('a1.grad: {0};'.format(a1.grad))
        print('w_2.grad: {0};'.format(w_2.grad))
```

Listing 22: 张量间乘法示例 (chpZ01/tensor.py)

运行测试用例的结果如下所示：

```
z2: (4, 1);
[[146.]
 [152.]
 [158.]
 [164.]]
a1.grad: [[ 50.]
 [ 90.]
 [130.]];
w_2.grad: [[1. 2. 3.]
 [1. 2. 3.]
 [1. 2. 3.]
 [1. 2. 3.]];
```

Listing 23: 张量间乘法运行结果示例 (chpZ01/tensor.py)

### 2.1.8 激活函数

激活函数是非线性函数

# 第一篇深度学习

# 第 1 章强化学习概述

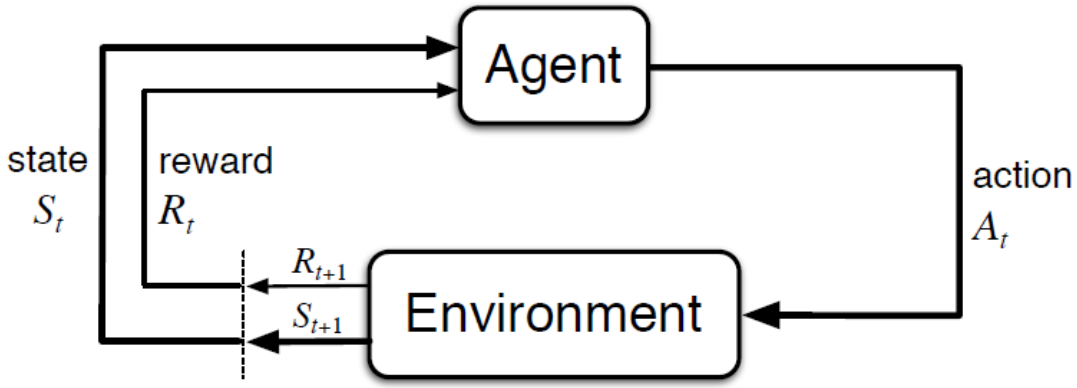
## Abstract

在本章中我们将讨论强化学习中的环境、Agent、状态、Action 和奖励，并重点讨论 MDP 相关内容。

## 3 MDP 概述

一个典型的强化学习系统结构如下所示：

图 9: 典型强化学习系统架构图




如图所示：

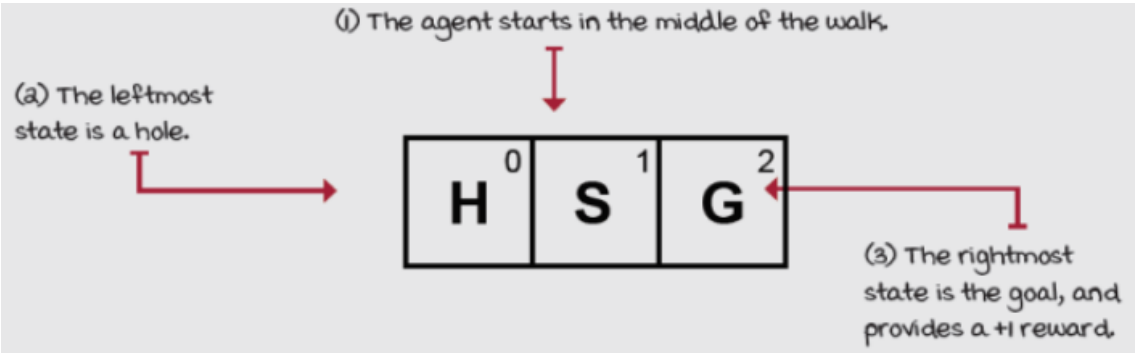
1. 在 $t$ 时刻 Agent 观察到环境状态 $S_t$，并得到上一时刻所采取的行动 $A_{t-1}$（在图中未画出）所得到的奖励 $r_t$；
2. Agent 根据环境状态 $S_t$，根据某种策略 $\pi$，选择行动 $A_t$；
3. 环境接收到 Agent 的行动 $A_t$ 后，根据环境的动态特性，转移到新的状态 $S_{t+1}$，并产生 $R_t$ 的奖励信号；

### 3.1 典型环境

#### 3.1.1 Bandit Walk 环境

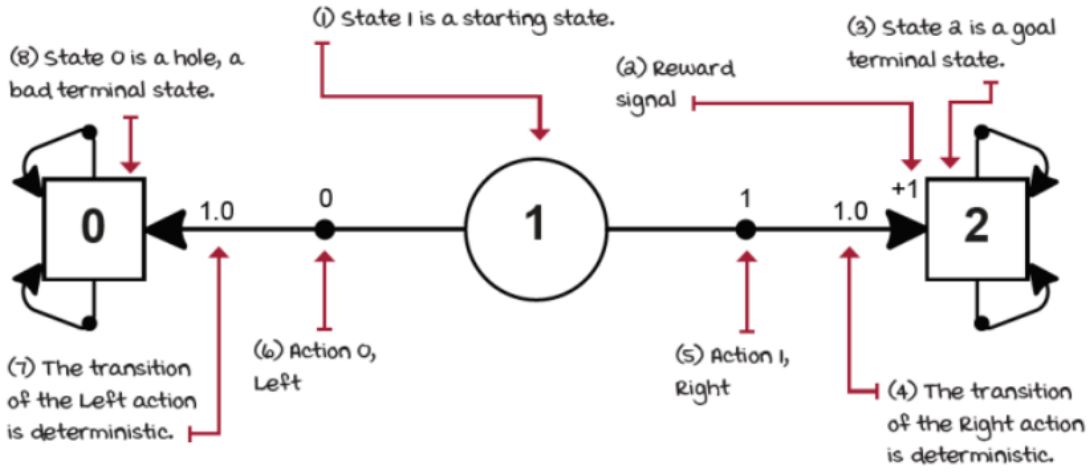
下面我们来研究一个最简单的强化学习环境，叫 Bandit Walk，如下所示：

图 10: Bandit Walk 环境图



如图所示，Agent 初始时位于中间的 S 格，状态编号为 $S_0$，其可以采取向左、向右两个动作，向左则进入状态 $H$，其是一个洞，就会掉到洞里，过程就会结束，此时得到的奖励为 0；当 Agent 采取向右行动时，就会进入 G 状态，此时会获得奖励 +1，由此可见其是一个确定性的环境，就是说当 Agent 采取向右行动时，会 100% 确定进行 G 状态。我们可以通过如下的图来表示上述过程：

图 11: Bandit Walk 环境 MDP 图




如图所示：

- 在初始状态 $S_0$ 时，有两个可选行动，分别表示为向左、向右的直线；
- 当采取向右行动时，就会到达小黑点位置，然后由环境决定将转到哪个状态，以及转到这个状态的概率，以本例为例，其就是以 100% 的概率转到 G 状态 $S_2$，其中小黑点上面的 1 代表行动编号，向右简头上面的 1.0 代表 100% 的概率，向右简头处的 1 代表奖励为 +1；

我们首先安装所需要的库：

```
pip install -i https://pypi.tuna.tsinghua.edu.cn/simple gym
```

Listing 24: 安装 gmy 库

下面我们用 Python 对象来表示这一过程：

```
P = {
    0: {
        0: [(1.0, 0, 0.0, True)],
        1: [(1.0, 0, 0.0, True)]
    },
    1: {
        0: [(1.0, 0, 0.0, True)],
        1: [(1.0, 2, 1.0, True)]
    },
    2: {
        0: [(1.0, 2, 0.0, True)],
```

```
        1: [(1.0, 2, 0.0, True)]
    }
}
print(P)
```

Listing 25: Bandit Walk python 程序

代码解读如下所示：

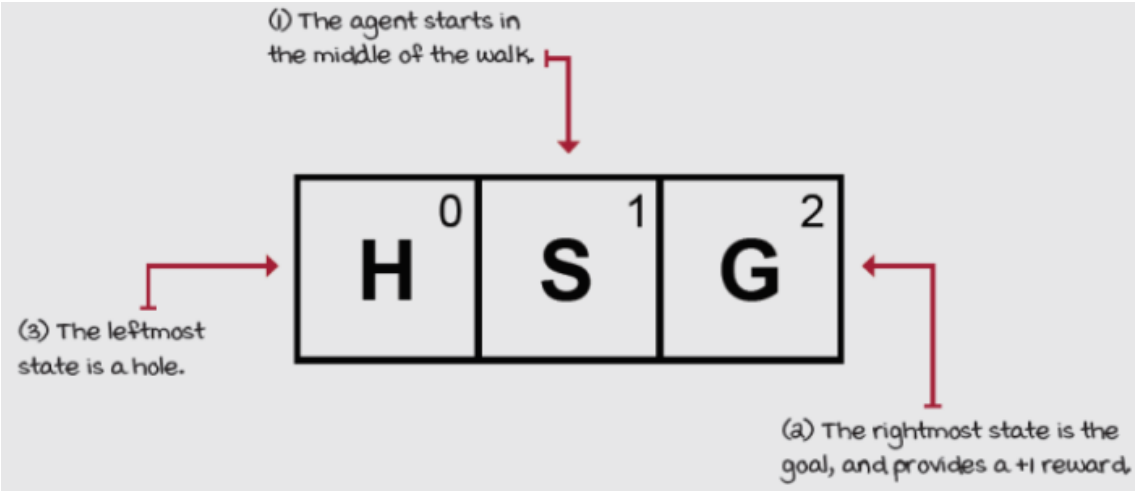
- P 为一个字典对象，其键值 0、1、2 代表三个状态；
- P 的键值 0：其同样是一个字典对象，键值代表可以采取的行动，0 代表向右，1 代表向右；
- P 的键值 0 下键值 0：即在状态 0 下面采取行动 0，其值为一个数组，代表由环境决定要转到哪个状态，转到每个状态为一个 Turple，含义为：（概率, 目的状态，获得奖励，新状态是否为终止状态），注意：我们规定在终止状态采取任何行动都会回到自身；

### 3.1.2 Bandit Slippery Walk 环境

上面我们仅举了一个例子，其他状态读者可以自己解析出来。

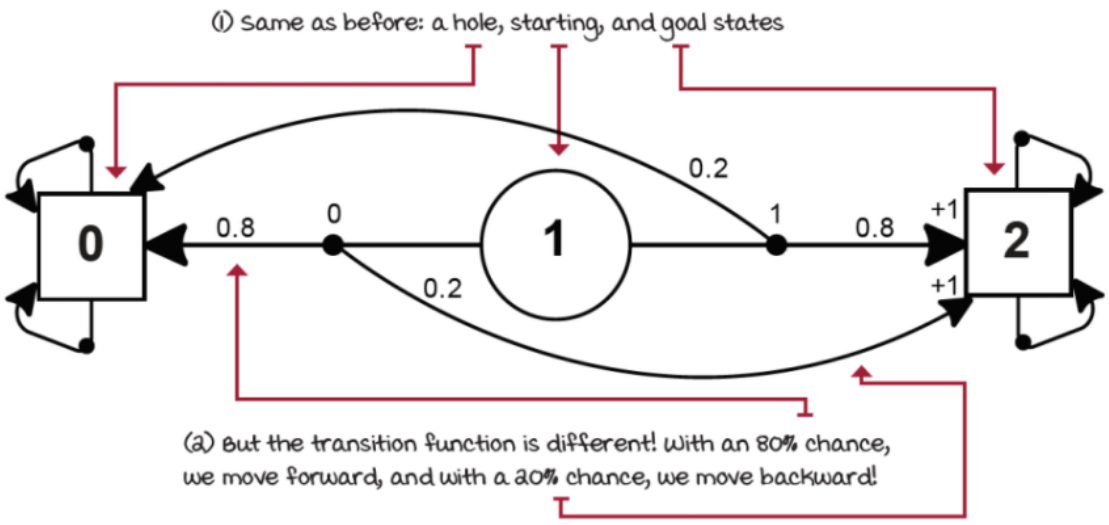
在上面的环境中，我们向左移动，环境会确定地向左移动。但是在本节中，当我们向左移动时，环境在 80% 的情况下会向左移动，20% 的情况会向右移动。如下图所示：

图 12: Bandit Slippery Walk 环境图




除了环境的随机性之外，环境与上一节相同。其 MDP 图如下所示：

图 13: Bandit Slippery Walk 环境 MDP 图




如上图所示，在开始时 Agent 位于状态 S，其可以采取的行动为向左编号为 0 或向右编号为 1，我们以向右为例，当 Agent 采取向右行动时，其达到状态 S 右侧的小黑点，上面的 1 代表是编号为 1 的行动，此时环境将以 80% 的概率转变为状态 G，得到 +1 的奖励，如图中的右箭头所示，同时环境还可能将以 20% 的概率变为状态 H，其所获得的奖励为 0，如图中向左的曲线箭头所示。读者可以按照上面的描述，自己补充出其他状态变化情况。由前面的讨论可以看出，在这个例子中，当 Agent 采取向右行动 Action 时，环境仅以 80% 的概率完成该 Action，同时还可能以 20% 的概率向相反的方向变化，既环境具有一定的随机性。我们可以通过如下的 Python 代码来表示这一过程：

```
    def bandit_slippery_walk(self):
        P = {
            0: {
                0: [(1.0, 0, 0.0, True)],
                1: [(1.0, 0, 0.0, True)]
            },
            1: {
                0: [(0.8, 0, 0.0, True), (0.2, 2, 1.0, True)],
                1: [(0.8, 2, 1.0, True), (0.2, 0, 0.0, True)]
            },
            2: {
                0: [(1.0, 2, 0.0, True)],
                1: [(1.0, 2, 0.0, True)]
            }
        }
        print(P)
```

Listing 26: Bandit Slippery Walk python 程序

如上所示，在状态 S 时，如果采取编号为 0 的向左行动，则有 80% 的概率会进入到状态 H，奖励为 0.0，并且是终止状态，当采用编号为 1 的向右行动时，将进入状态 G，获得奖励为 1.0，并且为终止状态，采用这种方式我们就表示了环境的随机性。

### 3.2 典型交互

Agent 与环境的交互分为分段的或连续的，由一系列时间步聚组成，在时间 $t$ 时刻：

- Agent 得到环境给的奖励信号 $R_t$，其由 Agent 在上一时刻 $S_{t-1}$ 采取行动 $A_{t-1}$ 时所获得的，并且 Agent 观察到环境状态 $S_t$；
- Agent 根据所观察到的环境状态 $S_t$，选择采取行动 $A_t$；
- 环境接收到行动 $A_t$ 后，会转移到新的状态 $S_{t+1}$，并且会给 Agent 奖励 $R_{t+1}$；
- 依次循环.....

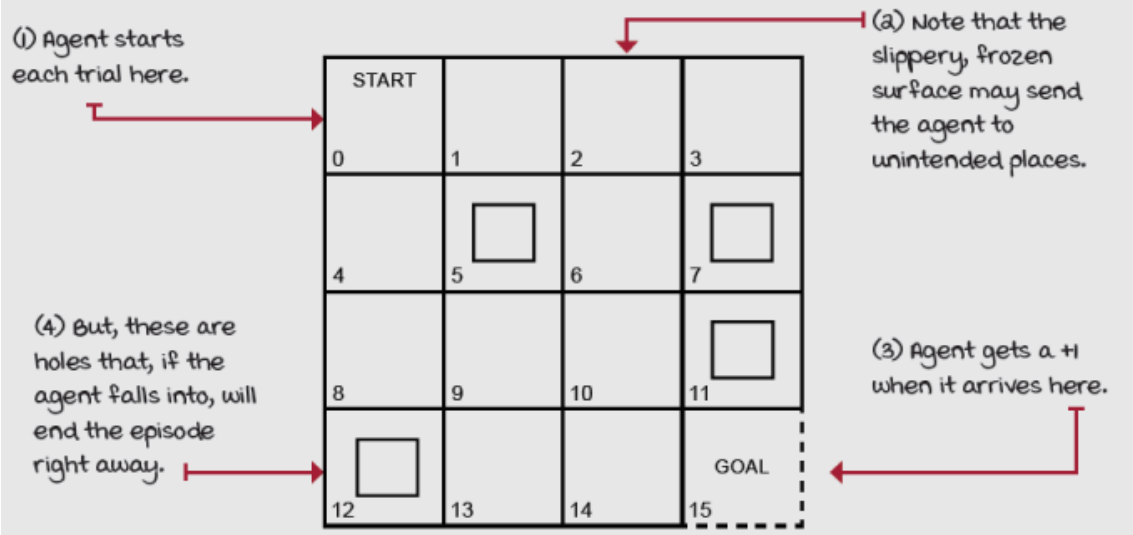
上述过程可以表示为：

$$(R_0, S_0, A_0), (R_1, S_1, A_1), (R_2, S_2, A_2), ..., (R_t, S_t, A_t), ..., (R_T, S_T, A_T) \tag{22}$$

### 3.3 MDP 定义

我们以 Frozen Lake 为例来定义 MDP 过程。该环境如下所示：

图 14: Frozen Lake 环境图




如图所示：

- Agent 从状态 Start 开始；
- 在每个状态，Agent 可以采取向左、向上、向下、向右行动，当在边缘状态时，走出环境的行动会 100% 使 Agent 留在原状态；
- 由于是冻冰的湖面，例如当 Agent 选择向下行动时，其有 33.3% 的概率向下运动，还有 66.7% 的概率会向垂直的方向运动，既以 33.3% 的概率向左运动，33.3% 的概率向右运动；
- 当 Agent 到达有洞的状态时，过程立即结束；
- 当 Agent 到达最终节点时，可以获得 +1 的奖励；

### 3.3.1 环境状态建模

时刻 $t$ 环境状态的状态表示为 $S_t$，环境所有可能的状态用集合 $\mathcal{S}$ 表示，通常我们用 $n$ 维向量来表示一个状态：

$$S_t = \boldsymbol{s} = \begin{bmatrix} s_1 \\ s_2 \\ \dots \\ s_n \end{bmatrix} \in R^n \tag{23}$$

对于我们当前研究的这个问题，环境状态只需要表示 Agent 处于哪个状态即可，我们采用 0∼15 来对状态进行编号，因此状态可以用 0∼15 来表示：

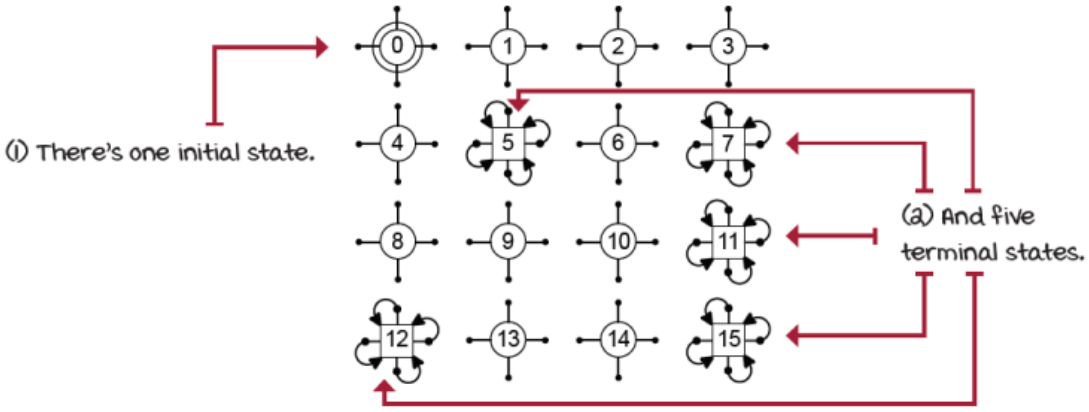
$$S_t = \boldsymbol{s} = \begin{bmatrix} i \end{bmatrix} \in R^1, i \in \{0, 1, 2, 3, ..., 15\} \tag{24}$$

我们规定环境只与当前状态有关，而与过去的历史无关，这就是马可夫特性，即我们研究的过程是无记忆的。乍一看，这是一个非常严重的限制条件，但是在实际应用中，我们通常可以通过设计合适的状态，使所研究的问题变为无记忆的。用数学语言可以表示为：

$$P(S_{t+1}|S_t, A_t) = P(S_{t+1}|S_t, A_t, S_{t-1}, A_{t-1}, ...) \tag{25}$$

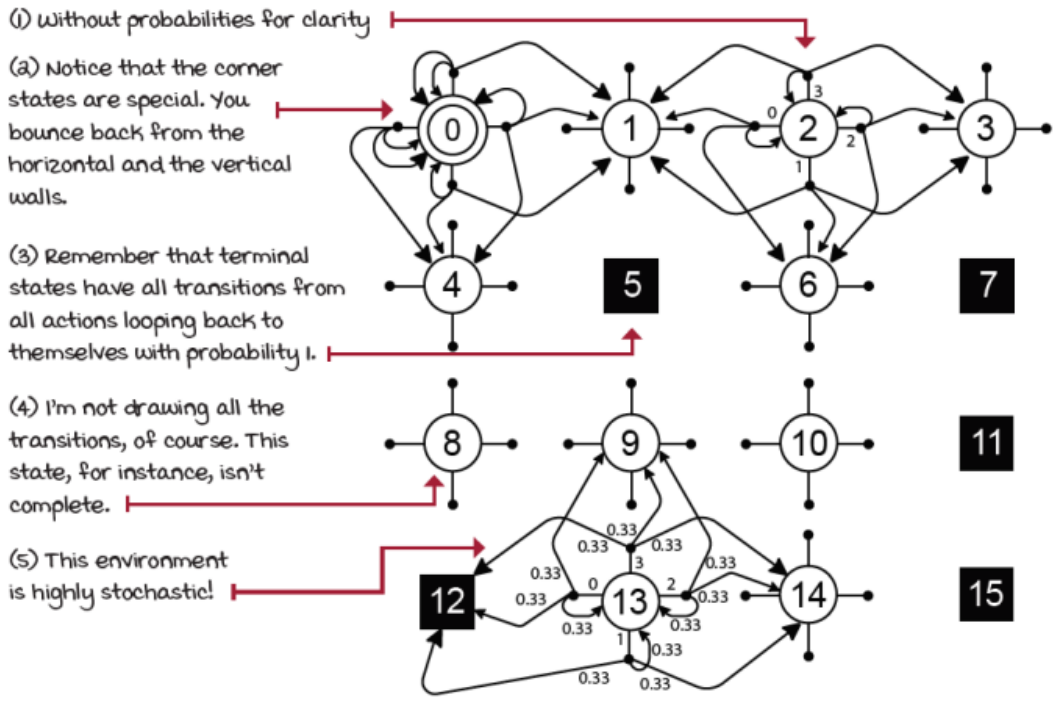
以 Frozen Lake 为例，其每个状态和在状态上可以采取的行动如下所示：

图 15: Frozen Lake 状态和行动图




当 Ageng 采取行动后，环境会根据自身的动态特性，转移到下一个状态，我们称之为转移函数，如下图所示：

图 16: Frozen Lake 状态转移图




在状态 13 时，共有向左、向下、向右、向上编号分别为 0、1、2、3 的四种行动，当 Agent 采取行动 0 向左时，将到达左侧的小黑点，

- 行动 0（向左）：到达左侧小黑点，由于冻冰原因，其有如下三种可能性：
  - 33.3%：向左，进入状态 12，获取奖励 0.0，并且为终止状态，用 (0.333, 12, 0.0, True) 表示；
  - 33.3%：向下，由于是边缘节点，其仍然在状态 13，获取奖励 0.0，不为终止状态，用 (0.333, 13, 0.0, False) 表示；
  - 33.3%：向上，进入状态 9，获取奖励为 0.0，不是终止状态，用 (0.33, 9, 0.0, False) 表示；
- 行动 1（向下）：到达下面小黑点，有如下三种可能性：
  - 33.3%（向下）：由于是边缘节点，其仍然在状态 13，获得奖励为 0.0，不是终止状态，用 (0.333, 13, 0.0, False) 表示；
  - 33.3%（向左）：进入状态 12，获得奖励 0.0，并且为终止状态，用 (0.333, 12, 0.0, True) 表示；
  - 33.3%（向右）：进入状态 14，获得奖励 0.0，不是终止状态，用 (0.333, 14, 0.0, False) 表示；

我们这里仅举了两个例子，其余内容读者可以自己补全。环境的状态转移函数如下所示：

$$
\begin{aligned}
p(s'|s,a) &= P(S_t = s'|S_{t-1} = s, A_{t-1} = a) \\
\sum_{s' \in S} p(s'|s,a) &= 1, \forall s \in S, \forall a \in A(s)
\end{aligned}
\tag{26}
$$

上式表明在任意时刻，环境状态为 $S_{t-1}=s$，Agnet 采取行动为 $A_{t-1}=a$ 时，环境由于具有随机性，以一个确定的概率分布进入新状态 $S_t=s'$，并且如果我们将所有可能到达的新状态的概率相加，其值为 1。当 Agent 根据自己的策略，在任意时刻采取行动后，系统会给 Agent 一个奖励 Reward，其是一个标量，越大代表该行动决策越好，越小代表越差，甚至可以为负值，代表需要尽力避免的情况。需要注意的是，Agent 不仅要关注当前获得的奖励，还要关注最终获得的累积的奖励，Agent 的目标就是使最终获得的累积奖励最大。环境的奖励函数如下表示：

$$r(s,a)=E\bigg(R_t|S_{t-1}=s,A_{t-1}=a\bigg) \tag{27}$$

上式表明在 $t-1$ 时刻，环境状态为 $S_{t-1}=s$，Agent 采取行动 $A_{t-1}=a$，环境在 $t$ 时刻给出奖励 $R_t$，由于环境具有随机性，环境可能进入不同的状态，从而获得不同的奖励，而且即使是进入同一个状态，获得的奖励也有可能不同，因此在这种情况，下的奖励就是所有这种情况下获得奖励的期望值。在 $t-1$ 时刻，环境状态为 $S_{t-1}=s$，Agent 采取行动 $A_{t-1}=a$，环境进入 $S_t=s'$ 时，获得的奖励为：

$$r(s,a,s')=E\bigg(R_t|S_{t-1}=s,A_{t-1}=a,S_t=s'\bigg) \tag{28}$$

从上式就可以看出，即使是转移到同一个状态，也可能获得不同的奖励，所以我们将奖励定义所有这些值的期望。在上面我们定义在任意时刻，Agent 通过与环境交互，获得的奖励为 $R_t$，同时我们知道，Agent 的目标是使整个过程，所有时刻所获得奖励的累加值最大，我们将其定义为回报 $G_t$。但是由于未来具有更大的不确定性，因此距离当前时间点越近，获得的奖励就越有价值，越远则价值越小，因此我们引入折扣的概念，如下所示：

$$\begin{aligned} G_t &= R_{t+1}+\gamma R_{t+2}+\gamma^2 R_{t+3}+...+\gamma^{k-1}R_{t+k}+...+R_T \\ &= \sum_{k=0}^{\infty}\gamma^k R_{t+k+1} \\ &= R_{t+1}+\gamma G_{t+1} \end{aligned} \tag{29}$$

### 3.4 状态价值函数

我们假定 Agent 的策略为 $\pi$，我们定义当 Agent 在某个状态可以获得的累积奖励的期望值为该状态的值函数，如下所示：

$$v_\pi(s)=E_\pi(G_t|S_t=s)=E_\pi(R_{t+1}+\gamma G_{t+1}|S_t=s) \tag{30}$$

由于上式是求期望值，根据期望值定义，可以得到：

$$v_\pi(s)=\sum_a \pi(a|s)\sum_{s',r}p(s',r|s,a)(r+\gamma v_\pi(s')) \tag{31}$$

这个公式在本质上是一个递归形式的公式，我们将用递归的方法来求解。

### 3.5 行动价值函数

我们还需要知道在某个状态下，采取某个行动到底有多好，这就是行动价值函数：

$$q_\pi(s,a)=E_\pi(G_t|S_t=s,A_t=a)=E_\pi(R_{t+1}+\gamma G_{t+1}|S_t=s,A_t=a) \tag{32}$$

根据期望的定义可得：

$$q_\pi(s,a)=\sum_{s',r}p(s',r|s,a)(r+\gamma v_\pi(s')) \tag{33}$$

这就是所谓的 Q 函数，需要注意的是这个公式的含义，这个公式表示在状态 $S_t = s$ 时 Agent 不按照策略 $\pi$ 情况下采取 $A_t = a$，然后 Agent 会一直采用策略 $\pi$，这种情况下所取得的累积奖励的期望值。

### 3.6 优势函数

当在状态 $S_t = s$ 时 Agent 不按照策略 $\pi$ 情况下采取 $A_t = a$，与在状态 $S_t = s$ 时 Agent 按照策略 $\pi$ 相比，所取得的累积奖励的期望值的变化量定义为优势函数（Advantage Function）：

$$a_\pi(s,a) = q_\pi(s,a) - v_\pi(s) \tag{34}$$

### 3.7 优化

我们的目的是要找到最优策略，使得在每个状态下的状态价值函数可以最大，每个行动价值函数也达到最大，需要注意的是，最优策略可能不止一种，但是对于每个状态的状态价值函数值却是唯一的，同时每个状态采取行动的行动价值函数值也是唯一的。我们用 $\pi^*$” 来表示最优策略，定义如下所示：

$$v_*(s) = \max_\pi v_\pi(s), \forall s \in S \tag{35}$$

我们可以将 $v_\pi$ 的计算公式代入可得：

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)\Big(r + \gamma v_*(s')\Big) \tag{36}$$

同样对于行动价值函数来说，最优策略可以定义为：

$$q_*(s,a) = \max_\pi q_\pi(s,a), \forall s \in S, \forall a \in A \tag{37}$$

代入具体的计算公式可行：

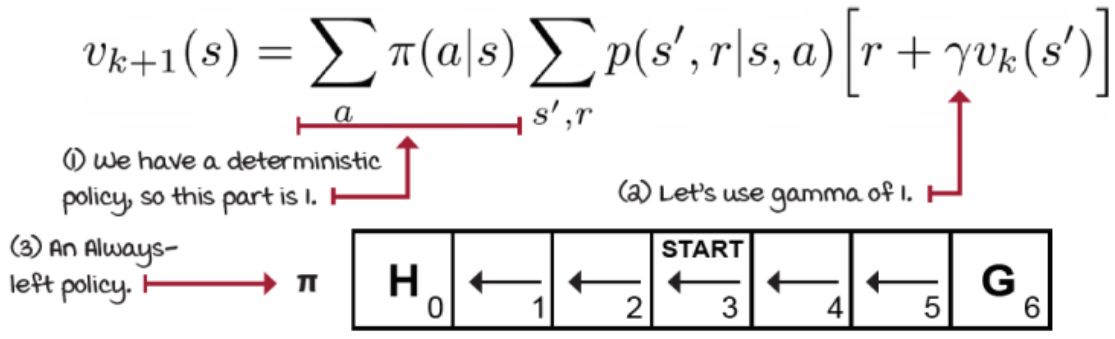
$$q_*(s,a) = \ sum_{s',r} p(s',r|s,a)\Big(r + \gamma \max_{a'} q_*(s',a')\Big) \tag{38}$$

在有了最佳策略定义之后，我们就需要来评估策略，我们首先用状态价值函数来评估策略，我们称之为预测问题：

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\Big(r + \gamma vk(s')\Big) \tag{39}$$

在上式中，下标 $k$ 代表是第几次迭代，通过迭代，我们可以求出每个状态的状态价值函数的值。我们以 Slippery Wall Floor 为例，来看上面公式的具体使用：

图 1.7 Slippery Walk Floor 环境示意图



如上图所示，我们的策略是在每个状态均采取向左的行动，我们假设现在我们在状态 5 处，根据当前策略，我们只有一个行动既向左行动，同时由于环境具有随机性，使我们可能进入到状态 4（50% 概率）、状态 5（33.3% 概率）和状态 6（16.6% 概率），初始时，我们设所有状态的状态价值函数的值为 0，如下所示：

$$
\begin{aligned}
v_1^\pi(5) &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\Big(r + \gamma v_0^\pi(s')\Big) \\
&= \sum_{s',r} p(s',r|s,a)\Big(r + \gamma v_0^\pi(s')\Big) && 1 \\
&= p(s'=4, r=0|s=5, a=Left)(r=0+1.0*v_0^\pi(4)) && 2 \\
&+ p(s'=5, r=0|s=5, a=Left)(r=0+1.0*v_0^\pi(5)) && 3 \\
&+ p(s'=6, r=1)(r=1+1.0*v_0^\pi(6)) && 4 \\
&= 0.5*(0+0) + 0.33*(0+0) + 0.16*(1+0) = 0.16
\end{aligned}
\tag{40}
$$

公式解读如下所示：

- 第 1 行：由于本策略只采取向左行动，因此前面的 $\sum_a \pi(a|s)$ 的概率值为 1，所以可以去掉；
- 第 2 行：转移到状态 4 时的情况；
- 第 3 行：转移到状态 5 时的情况；
- 第 4 行：转移到状态 6 时的情况；

通过上面的计算过程可以看出，我们将初始状态时所有状态的状态价值函数值设置为 0，然后通过上面的迭代算法，就可以算出各个状态的状态价值函数的真实值，并且收敛的速度还是比较快的。我们首先初始化强化学习环境，我们先安装所需的库：

```
cd /e/awork/ext
git clone https://github.com/mimoralea/gym-walk.git
cd gym-walk
pip install .
```

Listing 27: 安装依赖库

环境初始化代码如下所示：

```
def startup(self):
    env = gym.make('SlipperyWalkFive-v0')
    P = env.env.P
    init_state = env.reset()
    goal_state = 6
    LEFT, RIGHT = range(2)
    pi = lambda s: {
        0:LEFT, 1:LEFT, 2:LEFT, 3:LEFT, 4:LEFT, 5:LEFT, 6:LEFT
    }[s]
    self.print_policy(pi, P, action_symbols=('<', '>'), n_cols=7)

def print_policy(self, pi, P, action_symbols=('<', 'v', '>', '^'),
                 n_cols=4, title='Policy:'):
    print(title)
```

```
15          arrs = {k:v for k,v in enumerate(action_symbols)}
16          for s in range(len(P)):
17              a = pi(s)
18              print("| ", end="")
19              if np.all([done for action in P[s].values()
20                         for _, _, _, done in action]):
21                  print("".rjust(9), end=" ")
22              else:
23                  print(str(s).zfill(2), arrs[a].rjust(6), end=" ")
24              if (s + 1) % n_cols == 0: print("|")
```

Listing 28: SWF 环境初始化 (chp001/exp001_002.py)

代码解读如下所示：

- 第 2 行：生成强化学习环境，使用的是 gmy_walk 包中定义的环境；
- 第 3 行：P 为我们在前面讨论的结构：

```
1 {
2     si:{
3         ai: [(p, s', r, False)]
4     }
5 }
```

Listing 29: SWF 环境初始化 (chp001/exp001_002.py)

  第一层为以状态为键的字典，每个状态是以行动为键的字典，值为一个列表，代表在当前状态下采取行动后，环境会以多大的概率，转移动新状态，获得的奖励以及新状态是否为终止状态；

- 第 4 行：重置环境状态，并获取初始状态；
- 第 5 行：将状态 6 设置为目标状态；
- 第 6 行：设置行动 LEFT 为 0，RIGHT 为 1；
- 第 7~9 行：定义策略，其为 lambda 表达式，参数为 s，定义了一个字典，策略为取出该字典中以参数 s 为键的元素，其中 s 代表状态编号，策略的类型为函数；
- 第 10 行：打印策略；
- 第 12、13 行：定义打印策略函数：，
  - pi 为策略类型为函数，参数为状态编号，返回值为所采取的行动；
  - P 为状态、行动的转移矩阵，即环境的 MDP 特性；
  - action_symbols 行动的符号表示；
  - n_cols 共有几个状态，这里有 7 个状态，0 和 6 代表终止状态；
  - title 标题；
- 第 15 行：arrs 的形式为：0: '<', 1: '>'；
- 第 16 行：循环处理每个状态，s 为状态编号；
- 第 27 行：以状态编号为参数，调用策略函数，求出该状态下采取的行动编号（在其他复杂问题中，可能是采取一系列行动的概率分布）；
- 第 19、20 行：通过如下代表：

```
def test001(self, P, state):
    v1 = [action for action in P[state].values()]
    print('v1: {0};'.format(v1))
    v2 = [done for action in P[state].values() for _, _, _, done
    in action]
    print('v2: {0};'.format(v2))
    v3 = np.all(v2)
    print('v3: {0};'.format(v3))
```

Listing 30: SWF 环境初始化 (chp001/exp001_002.py)

状态 0 时:

```
    v1: [
            [
                (0.5000000000000001, 0, 0.0, True),
                (0.3333333333333333, 0, 0.0, True),
                (0.16666666666666666, 0, 0.0, True)
            ],
            [
                (0.5000000000000001, 0, 0.0, True),
                (0.3333333333333333, 0, 0.0, True),
                (0.16666666666666666, 0, 0.0, True)
            ]
        ];
    v2: [True, True, True, True, True, True];
    v3: True;
```

Listing 31: SWF 环境初始化 (chp001/exp001_002.py)

状态 1 时:

```
    v1: [
            [
                (0.5000000000000001, 0, 0.0, True),
                (0.3333333333333333, 1, 0.0, False),
                (0.16666666666666666, 2, 0.0, False)
            ],
            [
                (0.5000000000000001, 2, 0.0, False),
                (0.3333333333333333, 1, 0.0, False),
                (0.16666666666666666, 0, 0.0, True)
            ]
        ];
    v2: [True, False, False, False, False, True];
    v3: False;
```

Listing 32: SWF 环境初始化 (chp001/exp001_002.py)

状态 2 时:

```
    v1: [
            [
                (0.5000000000000001, 1, 0.0, False),
```

```
                (0.3333333333333333, 2, 0.0, False),
                (0.16666666666666666, 3, 0.0, False)
            ],
            [
                (0.5000000000000001, 3, 0.0, False),
                (0.3333333333333333, 2, 0.0, False),
                (0.16666666666666666, 1, 0.0, False)
            ]
        ];
    v2: [False, False, False, False, False, False];
    v3: False;
```

Listing 33: SWF 环境初始化 (chp001/exp001_002.py)

上面的代码就是确定某个状态是否是终止状态；

- 第 21 行：当为终止状态时打印空；
- 第 23 行：当不为终止状态时，打印状态编号和行动的符号；

下面我们来看我们取得胜利的概率，如下所示：

```
def probability_success(self, env, pi, goal_state, n_episodes=100,
max_steps=200):
    random.seed(123); np.random.seed(123) ; env.seed(123)
    results = []
    for epoch in range(n_episodes):
        state, done, steps = env.reset(), False, 0
        while not done and steps < max_steps:
            state, _, done, h = env.step(pi(state))
            steps += 1
        results.append(state == goal_state)
    return np.sum(results)/len(results)
```

Listing 34: 求获得胜利既到达状态 6 的概率 (chp001/exp001_002.py)

代码解读如下所示：

- 第 4 行：循环指定 epoch 数；
- 第 5 行：初始化环境，我们假定开始时 Agent 在状态 3；
- 第 6 行：完整执行一个 epoch，并且如果超过指定步数后，强制终止此 epoch；
- 第 7 行：在当前状态 state 下，根据策略 pi，采取一个行协 pi(state)，调用环境的 evn.step 方法，转移到下一个状态，其返回值为：下一状态、奖励、是否完结、附加信息，其中新状态和奖励是根据我们状态转移 P 来决定的；
- 第 8 行：统计当前 epoch 中的步数；
- 第 9 行：当完成一个 epoch 后，如果终止状态为 Goal 状态，则将 1 加入到 results 列表中，否则将 0 加入到 results 列表中；
- 第 10 行：results 列表中为 1 的项数除以总项数即为获胜的概率；

接下来我们看一下在该策略下，我们可以获取平均回报：

```
    def mean_return(self, env, pi, n_episodes=100, max_steps=200):
        random.seed(123); np.random.seed(123) ; env.seed(123)
        results = []
        for _ in range(n_episodes):
            state, done, steps = env.reset(), False, 0
            results.append(0.0)
            while not done and steps < max_steps:
                state, reward, done, _ = env.step(pi(state))
                results[-1] += reward
                steps += 1
        return np.mean(results)
```

Listing 35: 求平均回报 (chp001/exp001_002.py)

这段代码的逻辑与**??**类似，只不过是每个 epoch 开始前，向 results 列表中加入一个 0.0 元素，在该 epock 中的每一步，都会将所获得奖励 reward 叠加到该值上，这样可以求出每个 epoch 的累积奖励，最后我们返回这些累积奖励值的平均值。接下来我们对当前策略进行评估，利用迭代法求出所状态价值函数的值：

```
    def policy_evaluation(self, pi, P, gamma=1.0, theta=1e-10):
        prev_V = np.zeros(len(P), dtype=np.float64)
        while True:
            V = np.zeros(len(P), dtype=np.float64)
            for s in range(len(P)):
                for prob, next_state, reward, done in P[s][pi(s)]:
                    V[s] += prob * (reward + gamma * prev_V[next_state] *
    (not done))
            if np.max(np.abs(prev_V - V)) < theta:
                break
            prev_V = V.copy()
        return V
```

Listing 36: 策略评估 (chp001/exp001_002.py)

代码解读如下所示：

- 第 2 行：初始时将所有状态的状态价值函数的值设置为 0，将其作为上一迭代的值；
- 第 4 行：进行无限次迭代循环；
- 第 5 行：当前状态的状态价值函烽的初始值置为 0；
- 第 6 行：对每个状态进行循环；
- 第 7 行：对每个状态，在当前策略下采取的行动，根据转移函数 P，得到概率 prob，下一个状态 next_state 和奖励 reward，以及是否结束标志 done；
- 第 8 行：利用公式 $v_{k+1} = \sum_a \pi(a|s) \sum_{s',r} (r + \gamma v_k)$ 求出新的状态价值函数的值；
- 第 8、9 行：当求完所有状态的状态价值函数值时，看两次迭代之间状态价值函数的差值是否小于阈值，如果小于则退出最外层的无限循环；
- 第 10 行：如果二者的差值大于阈值，则将当前值设置为上一次迭代值，重复进行循环；

上面程序的最终结果就是求出所有状态的真实的状态价值函数的值，接下来我们打印状态价值函数的值：

```
    def print_state_value_function(self, V, P, n_cols=4, prec=3, title='State-value function:'):
        print(title)
        for s in range(len(P)):
            v = V[s]
            print("| ", end="")
            if np.all([done for action in P[s].values() for _, _, _, done in action]):
                print("".rjust(9), end=" ")
            else:
                print(str(s).zfill(2), '{}'.format(np.round(v, prec)).rjust(6), end=" ")
            if (s + 1) % n_cols == 0: print("|")

    def startup(self):
        env = gym.make('SlipperyWalkFive-v0')
        P = env.env.P
        init_state = env.reset()
        goal_state = 6
        LEFT, RIGHT = range(2)
        pi = lambda s: {
            0:LEFT, 1:LEFT, 2:LEFT, 3:LEFT, 4:LEFT, 5:LEFT, 6:LEFT
        }[s]
        self.print_policy(pi, P, action_symbols=('<', '>'), n_cols=7)
        prob = self.probability_success(env, pi, goal_state)
        print('win prob:{0};'.format(prob))
        g_mean = self.mean_return(env, pi)
        print('mean return:{0};'.format(g_mean))
        V = self.policy_evaluation(pi, P)
        self.print_state_value_function(V, P, n_cols=7, prec=5)
        improved_pi = self.policy_improvement(V, P)
        self.print_policy(improved_pi, P, action_symbols=('<', '>'), n_cols=7)
```

Listing 37: 打印状态价值函数的值 (chp001/exp001_002.py)

这个函数比较简单，我们就不做介绍了。通过对策略的评价，我们得到了每个状态的状态价值函数的值，接下来我们讨论怎样根据这些内容来优化我们的策略。这里我们需要用到行动函数，即在每个状态，我们从所有行动中，找出可以使我们转到的新状态，可以获得最高的状态价值函数，并以此为新的策略：

$$\pi'(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s')) \tag{41}$$

上式中的 $\pi'(s)$ 就是改进后的策略。策略改进代码如下所示：

```
    def policy_improvement(V, P, gamma=1.0):
        Q = np.zeros((len(P), len(P[0])), dtype=np.float64)
        for s in range(len(P)):
```

```
4             for a in range(len(P[s])):
5                 for prob, next_state, reward, done in P[s][a]:
6                     Q[s][a] += prob * (reward +
7                                 gamma * V[next_state] * (not done))
8         new_pi = lambda s: {s:a for s, a
9             in enumerate(np.argmax(Q, axis=1))
10        }[s]
11        return new_pi
```

Listing 38: 策略改进 (chp001/exp001_002.py)

代码解读如下所示：

- 第 2 行：Q 为一个二维数组，第一维是所有的状态，第二维是在某个状态下所有的行动选项，其值为该行动的行动价值函数，初始时设置为 0；
- 第 3 行：循环处理所有状态：
- 第 4 行：循环处理每个状态下可以采取的所有行动：
- 第 5 行：根据环境的状态转移函数，在该状态下采取该行动，可以得到：转移到新状态的概率、新状态、奖励、是否为终止状态；
- 第 6、7 行行：根据公式 $q_\pi(s,a)=\sum_{s',r}p(s',r|s,a)(r+\gamma v_\pi(s'))$；
- 第 8~10 行：求出新的优化过的策略：

$$\begin{bmatrix} q(s_0,a_0) & q(s_0,a_1) & \dots & q(s_0,a_{k_0}) \\ q(s_1,a_0) & q(s_1,a_1) & \dots & q(s_1,a_{k_1}) \\ \dots & \dots & \dots & \dots \\ q(s_l,a_0) & q(s_l,a_1) & \dots & q(s_l,a_{k_l}) \end{bmatrix} \tag{42}$$

  上式中每一行代表一个状态，每一列代表在该状态下采取某个行动可以获取到的累积回报，注意每行的数量可能并不相同。np.argmax(,axis=1) 代表把第 2 维去掉，即求每一行的最大值，也就是求出某个状态采取什么行动可以获得最大的回报。最终形厉每个状态应该采取的行动编号，利用 lambda 表达式形成策略函数。；
- 第 11 行：返回这个新生成的策略函数；

运行结果如下所示：

图 18: Slippery Walk Floor 策略优化一次结果

```
(pydev) E:\awork\iching>python app_main.py
易经量化交易系统 v0.0.1
MDP应用
Policy:
|          | 01     < | 02     < | 03     < | 04     < | 05     < |          |
获胜概率：0.07;
平均回报：0.07;
State-value function:
|          | 01 0.00275 | 02 0.01099 | 03 0.03571 | 04 0.10989 | 05 0.33242 |          |
Policy:
|          | 01     > | 02     > | 03     > | 04     > | 05     > |          |
```

如上图可以看出，仅经过一次策略优化，我们就可以得到正确的策略。实际上我们对策略优化之后，我们可以重新进行策略评估，然后再进行优化，形成所谓的策略迭代，程序如下所示：

```
1     def policy_iteration(self, P, gamma=1.0, theta=1e-10):
2         random_actions = np.random.choice(tuple(P[0].keys()), len(P))
3         pi = lambda s: {s:a for s, a in enumerate(random_actions)}[s]
4         while True:
5             old_pi = {s:pi(s) for s in range(len(P))}
6             V = self.policy_evaluation(pi, P, gamma, theta)
7             pi = self.policy_improvement(V, P, gamma)
8             if old_pi == {s:pi(s) for s in range(len(P))}:
9                 break
10        return V, pi
11
12    def startup(self):
13        env = gym.make('SlipperyWalkFive-v0')
14        P = env.env.P
15        init_state = env.reset()
16        goal_state = 6
17        LEFT, RIGHT = range(2)
18        pi = lambda s: {
19            0:LEFT, 1:LEFT, 2:LEFT, 3:LEFT, 4:LEFT, 5:LEFT, 6:LEFT
20        }[s]
21        self.print_policy(pi, P, action_symbols=('<', '>'), n_cols=7)
22        prob = self.probability_success(env, pi, goal_state)
23        print('win prob:{0};'.format(prob))
24        g_mean = self.mean_return(env, pi)
25        print('mean return:{0};'.format(g_mean))
26        V = self.policy_evaluation(pi, P)
27        self.print_state_value_function(V, P, n_cols=7, prec=5)
28        improved_pi = self.policy_improvement(V, P)
29        self.print_policy(improved_pi, P, action_symbols=('<', '>'),
    n_cols=7)
30        # policy iteration
31        optimal_V, optimal_pi = self.policy_iteration(P)
32        self.print_policy(optimal_pi, P, action_symbols=('<', '>'),
    n_cols=7)
33        self.print_state_value_function(optimal_V, P, n_cols=7, prec=5)
```

Listing 39: 策略迭代 (chp001/exp001_002.py)

代码解读如下所示：

- 第 2 行：其中 P[0]=0:[(0.5, 0, 0.0, True), ...], 1:[...]，所以 tuple(P[0].keys()) 为 (0, 1)，因此 np.random.choice 为生成长度为 len(P) 既所有状态数的数组，数组的每一位由 (0,1) 中随机抽取；
- 第 3 行：生成策略的 lambda 表达式，参数为状态编号；
- 第 4 行：进入无限循环：
- 第 5 行：将当前策略保存为原始策略，类型为字典，格式为：状态: 行动；
- 第 6 行：求出在当前策略下的状态价值函数；
- 第 7 行：根据状态价值函数得到优化后的策略；

- 第 8、9 行：将优化后策略也转为格式为状态: 行动的字典，与原来的策略形成的字典进行比较，如果相等则意味着找到了最佳策略，则退出无限循环，否则继续循环优化；

运行结果如下所示：

图 19: Slippery Walk Floor 策略迭代结果

```
(pydev) E:\awork\iching>python app_main.py
易经量化交易系统 v0.0.1
MDP应用
Policy:
|          | 01     < | 02     < | 03     < | 04     < | 05     < |          |
获胜概率：0.07;
平均回报：0.07;
State-value function:
|          | 01 0.00275 | 02 0.01099 | 03 0.03571 | 04 0.10989 | 05 0.33242 |          |
Policy:
|          | 01     > | 02     > | 03     > | 04     > | 05     > |          |
Policy:
|          | 01     > | 02     > | 03     > | 04     > | 05     > |          |
State-value function:
|          | 01 0.66758 | 02 0.89011 | 03 0.96429 | 04 0.98901 | 05 0.99725 |          |
```

上面讲的是策略迭代算法（PI：Policy Iteration），其核心思想是先根据策略求出状态价值函数，然后根据状态价值函数，在每个状态下选择能够达到最大状态价值函数状态的行动，形成优化后的策略，然后循环执行上述过程。但是这种方法通常收敛速度较慢，我们接下来介绍值迭代算法（VI：Value Iteration）。

## 3.8 值迭代（VI）

值迭代（VI：Value Iteration）的核心思想是先令所有状态的状态价值函数值为 0.0，在每个状态下求出所采取的行动的回报，找到得到最大回报的行动，根据下面的公式确定新的状态价值函数值：

$$v_{k+1} = \max_a \sum_{s',r} p(s', r|s, a)\Big(r + \gamma v_k(s')\Big) \tag{43}$$

代码实现如下所示：

```
    def value_iteration(self, P, gamma=1.0, theta=1e-10):
        V = np.zeros(len(P), dtype=np.float64)
        while True:
            Q = np.zeros((len(P), len(P[0])), dtype=np.float64)
            for s in range(len(P)):
                for a in range(len(P[s])):
                    for prob, next_state, reward, done in P[s][a]:
                        Q[s][a] += prob * (reward + gamma * V[next_state]
    * (not done))
            if np.max(np.abs(V - np.max(Q, axis=1))) < theta:
                break
            V = np.max(Q, axis=1)
        pi = lambda s: {s:a for s, a in enumerate(np.argmax(Q, axis=1))}[
    s]
        return V, pi

    def startup(self):
```

```
16          env = gym.make('SlipperyWalkFive-v0')
17          P = env.env.P
18          init_state = env.reset()
19          goal_state = 6
20          LEFT, RIGHT = range(2)
21          pi = lambda s: {
22              0:LEFT, 1:LEFT, 2:LEFT, 3:LEFT, 4:LEFT, 5:LEFT, 6:LEFT
23          }[s]
24          self.print_policy(pi, P, action_symbols=('<', '>'), n_cols=7)
25          prob = self.probability_success(env, pi, goal_state)
26          print('win prob: {0};'.format(prob))
27          g_mean = self.mean_return(env, pi)
28          print('mean return: {0};'.format(g_mean))
29          V = self.policy_evaluation(pi, P)
30          self.print_state_value_function(V, P, n_cols=7, prec=5)
31          improved_pi = self.policy_improvement(V, P)
32          self.print_policy(improved_pi, P, action_symbols=('<', '>'),
    n_cols=7)
33          # policy iteration
34          print('PI: Policy Iteration')
35          optimal_V, optimal_pi = self.policy_iteration(P)
36          self.print_policy(optimal_pi, P, action_symbols=('<', '>'),
    n_cols=7)
37          self.print_state_value_function(optimal_V, P, n_cols=7, prec=5)
38          print('VI: Value Iteration')
39          V2, pi2 = self.value_iteration(P)
40          self.print_policy(optimal_pi, P, action_symbols=('<', '>'),
    n_cols=7)
41          self.print_state_value_function(optimal_V, P, n_cols=7, prec=5)
```

Listing 40: 策略迭代 (chp001/exp001_002.py)

代码解读如下所示：

- 第 2 行：初始状态下状态价值函数值为 0.0；
- 第 3 行：无限循环求解状态价值函数和策略：
- 第 4 行：初始化 $q(s,a)$ 函数，第 0 维是状态，第 1 维是行动，初始值为 0.0；
- 第 5、6 行：循环处理每个 $(s,a)$ 组合：
- 第 7 行：当在状态 $s$ 采取行动 $a$ 后，环境会转移到新状态 $s'$ 并给出奖励 $r$，用环境 P 中的 $(prob, s', r, isTerminal)$ 来表示，其中 prob 为转移到新状态 $s'$ 并获取奖励 $r$ 的概率，最后一项表示 $s'$ 是否是终止状态；
- 第 8 行：利用公式 $q(s,a) = \sum_{s',r} p(s',r|s,a)(r + \gamma v_\pi(s'))$，求出 Q 函数的值；
- 第 9、10 行：如果上一步得到状态价值函数值，与 Q 函数值对每个状态采取行动可以获得最大 Q 函数值所组成的新状态价值函数值之间的差距小于阈值时终止最外层无限循环；
- 第 11 行：对于每个状态，用采取所有行动中取得最大 Q 值作为状态价值函数的值；
- 第 12 行：当结束无限循环之后，将策略定为在每个状态，取可以取得最大 Q 值的行动作为应该选取的行动；

# 第 2 章 Bandit 问题

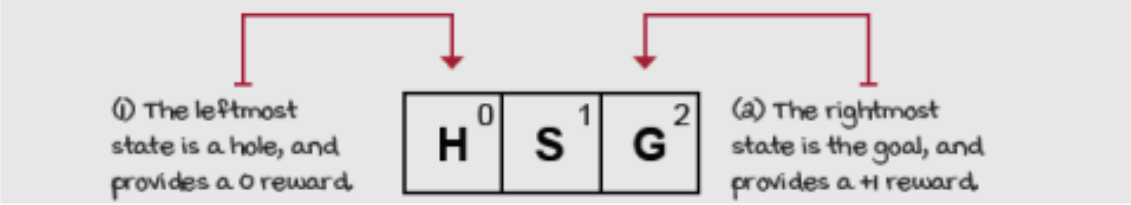
## Abstract

在本章中我们将介绍 Bandit 问题，就是 Agent 并不知道 MDP 环境（即第 1 章中的 P），但是 Agent 可以通过适当的策略来获得最优策略。

## 4 Bandit 问题概述

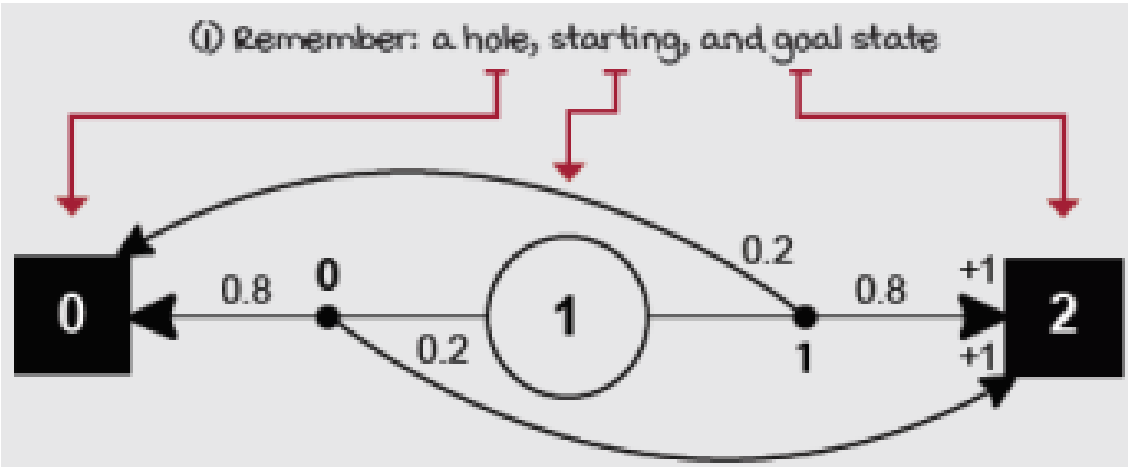
我们以 SBW（Slippery Bandit Walk）为例，如下图所示：

图 20: Slippery Bandit Walk 环境




初始时，Agent 位于状态 S，左侧状态 H 为一个洞，是终止状态，获得奖励为 0.0；右侧状态 G 为目标，获得奖励为 +1.0，是终止状态。其 MDP 环境如下所示：

图 21: Slippery Bandit Walk 环境之 MDP




如上图所示：

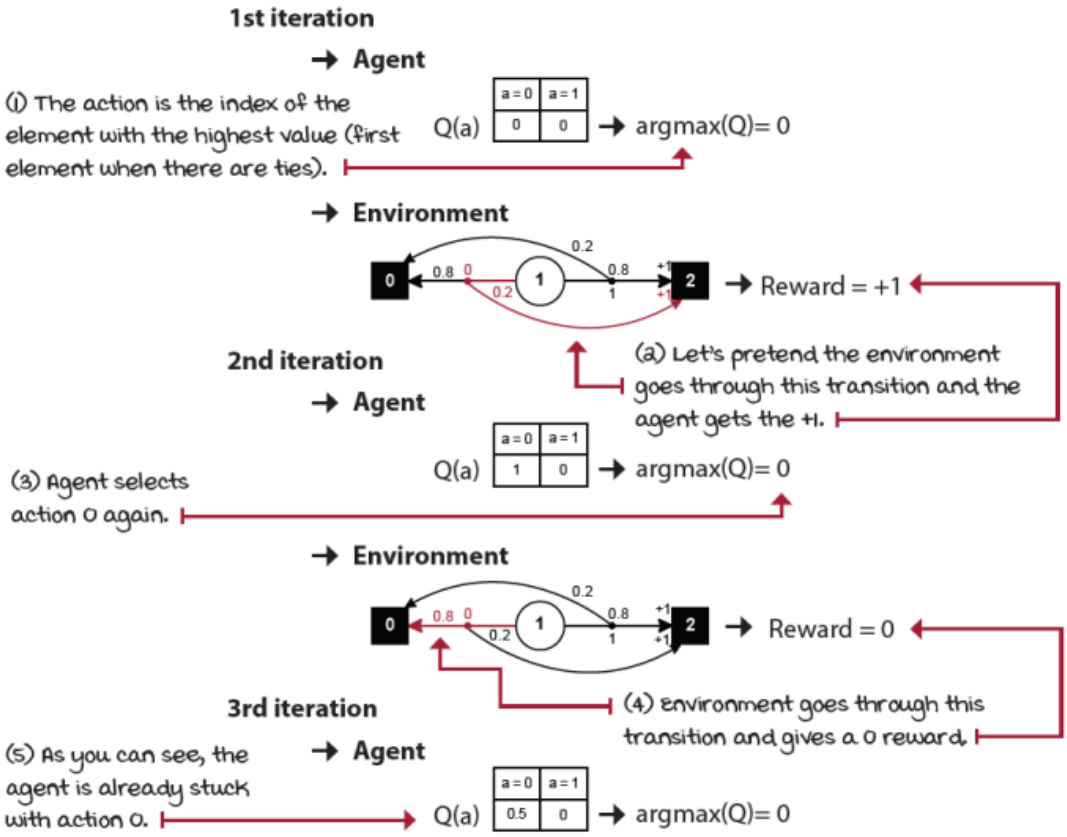
- Agent 在状态 S 时，选择行动 0（向左），进入左侧小黑点：
  - 以 80% 概率进入状态 H，获得奖励 0.0 并终止；
  - 以 20% 概率进入状态 G，获得奖励 +1.0 并终止；
- Agent 在状态 S 时，选择行动 1（向右），进入右侧小黑点：
  - 以 80% 概率进入状态 G，获得奖励 +1.0 并终止；
  - 以 20% 概率进入状态 H，获得奖励 0.0 并终止；

但是与第 1 章不同，我们这里假设我们并不知道这个转移函数。我们要研究在这种情况下，Agent 怎样找到最佳策略。

### 4.1 贪婪策略

Agent 会首先随机选择一个行动，如果该行动产生正的奖励，之后就一直坚持采用该行动，以 SBW 环境为例，如图所示：

图 22: SBW 环境之 Greedy Policy




我们将在状态 S 能采取的所有行动的 Q 值设为 0，既 $\{0:0.0, 1:0.0\}$，假设我们初始时在状态 S 选择行动 0（向左），进入左侧实心小点，此时环境随机进入状态 G，因此我们得到了 +1.0 的奖励，更新 Q 值为 $\{0:1.0, 1:0.0\}$，因为我们看到采取行动 0 可以获得 1.0 的奖励，因此我们会一直选择行动 0。当第 2 次时，我们仍然选择行动 0，此时环境随机地转到状态 H，此时我们获得的奖励为 0.0，我们重新计算 Q 值：$q = \frac{1.0+0.0}{2} = 0.5$，式中分子表示我们分别获得了 +1.0 和 0.0 的奖励，分母为我们进行试验的次数。当第 3 次时，我们仍然会选择行动 0，此时环境随机地转到状态 H，我们获得的奖励为 0.0，我们重新计算 Q 值：$q = \frac{1.0+0.0+0.0}{3} = 0.333333$，分子表示历次试验获得的奖励，分子为试验次数。最终我们策略的 Q 值为 0.2 左右。

# 第二篇时序信号分析

# 第章 Prophet 使用


## Abstract

在本章中，我们将讲解利用 Facebook 的 Prophet 来进行时序信号预测的方法。


## 5 Prophet 概述

### 5.1 配置 Prophet 开发环境

在本节中，我们将在 anaconda 环境下，安装 prophet 包，并利用一个小例子，简单讲解一下 Prophet 的使用方法。

# 第三篇量化交易平台

# 第四篇期权量化交易

# 第 401 章股指期权


## Abstract

在本章中，我们将以上证 50ETF 为例，讲解股指期权基本知识。


## 6 股指期权概述

# 第五篇基金管理平台

# 第六篇常用工具

# 第 601 章强化学习环境


## Abstract

采用创新方式运行 gym。


## 7 强化学习环境概述

我们以 SBW（Slippery Bandit Walk）为例，如下图所示：

### 7.1 Docker 模式

如下所示：

```
# install docker
https://docs.docker.com/engine/install/ubuntu/
# install nvidia-docker
https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html#docker
# download image
sudo nvidia-docker pull mimoralea/gdrl:v0.14
# startup image
sudo nvidia-docker run -it --rm -p 8888:8888 -v /home/ps/yantao/:/mnt/yantao/ mimoralea/gdrl:v0.14 -d /bin/bash
# jupity notebook access
http://192.168.2.66:8888
# list image list
sudo nvidia-docker ps
# enter specific image: mimoralea/gdrl:v0.14
sudo nvidia-docker exec -it image-container-id /bin/bash
```

Listing 41: 张量 expand 示例 (chpZ01/tensor.py)

注意：由于已经安装了 Docker，所以需要忽略第一个命令。

# 8 附录 X