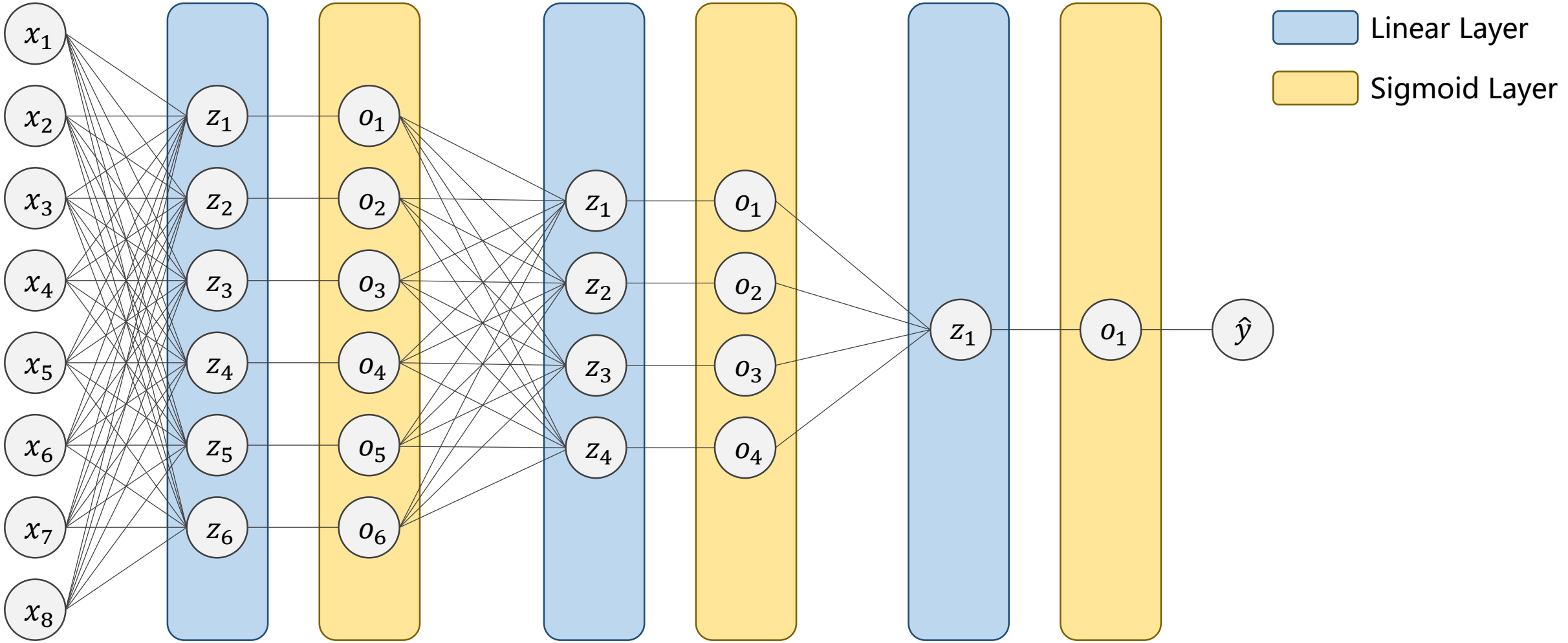
PyTorch Tutorial

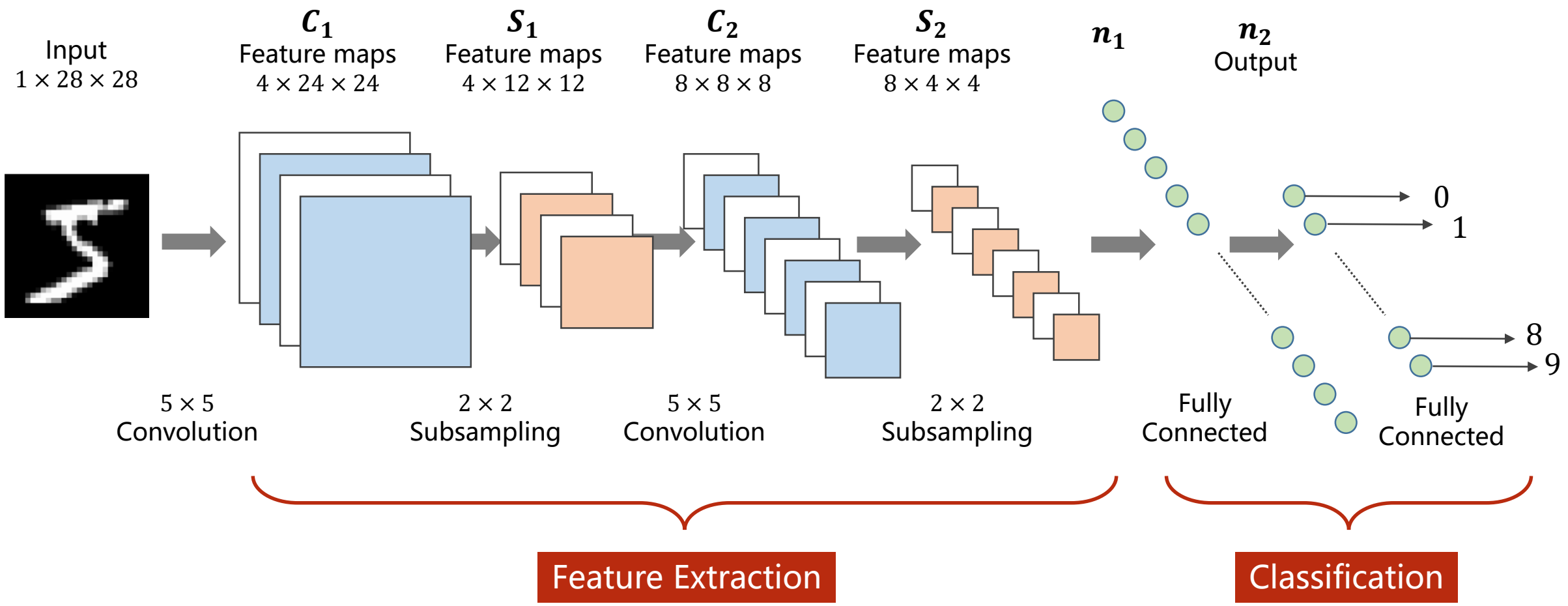
12. Basic RNN

循环神经网络：适合具有序列特征的输入数据

Revision: DNN



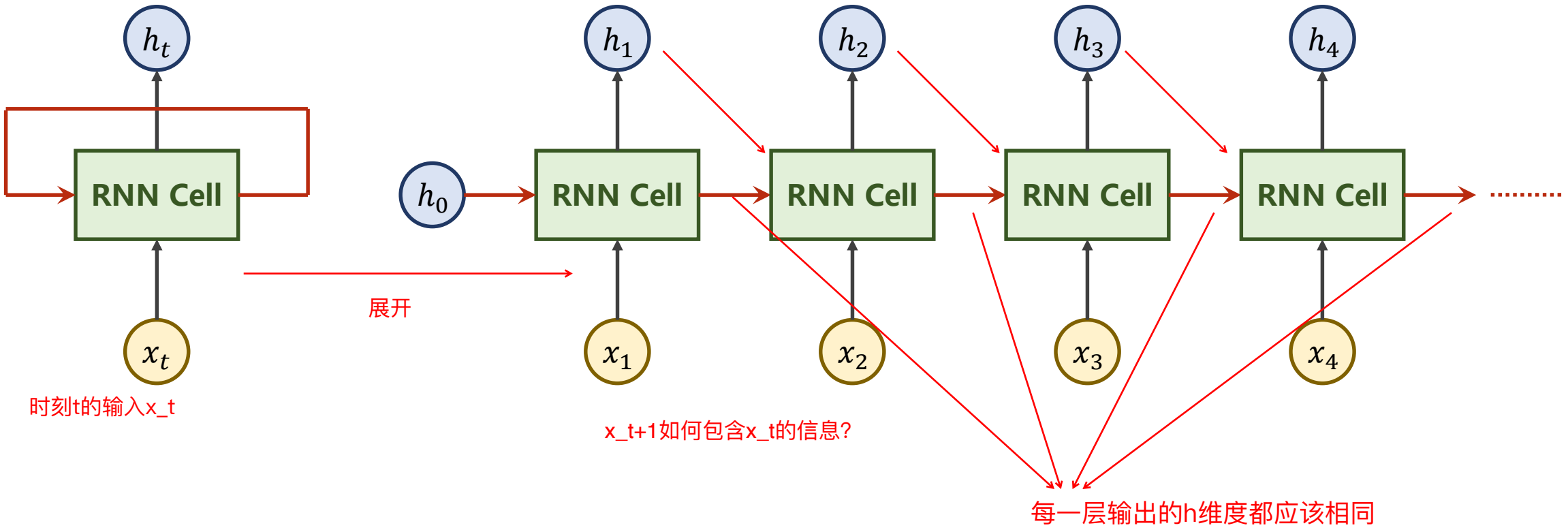
Revision: CNN



What is RNNs?

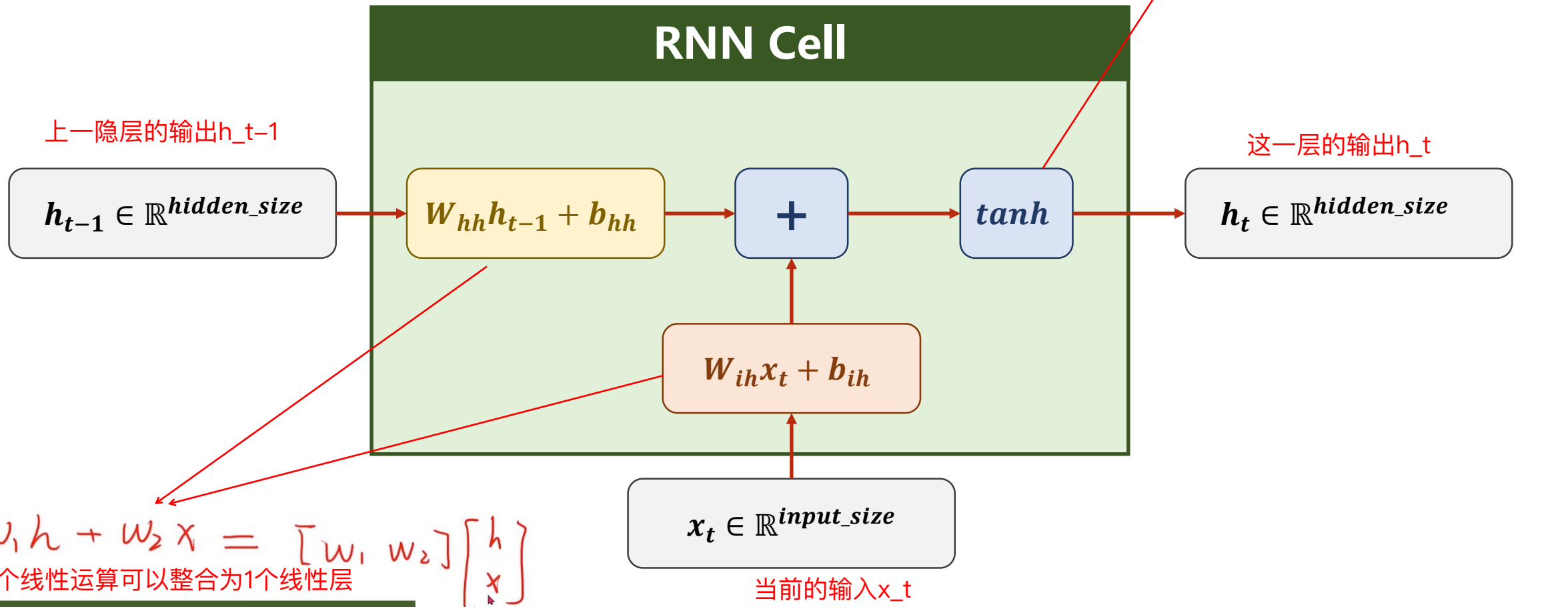
这里的RNN Cell是循环共用的，其本质是一个线性层，但是其权重受序列整体的影响。

h = hidden



What is RNN?

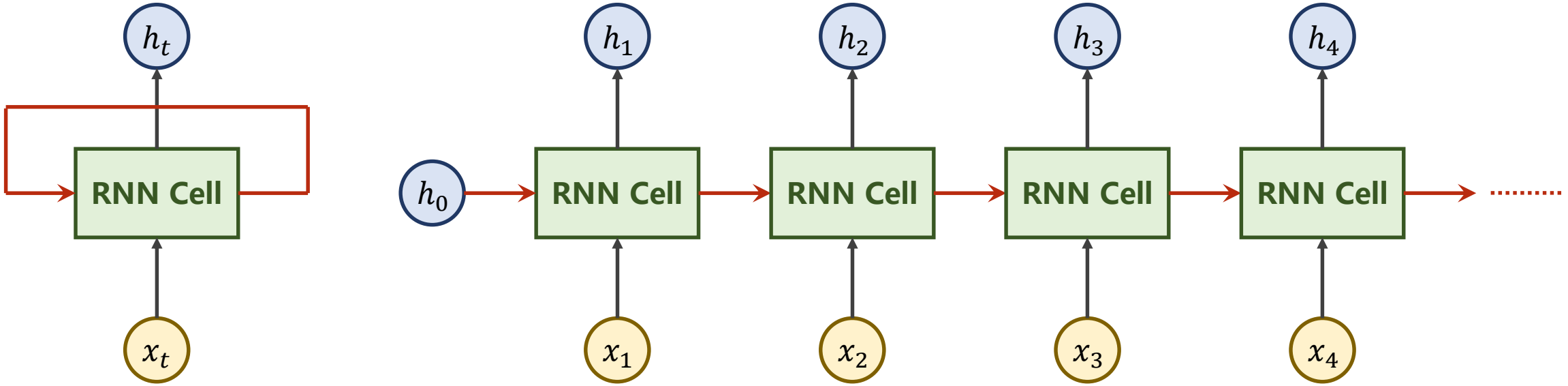
$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1} + b_{hh})$$



$$w_1 h + w_2 x = \begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} h \\ x \end{bmatrix}$$

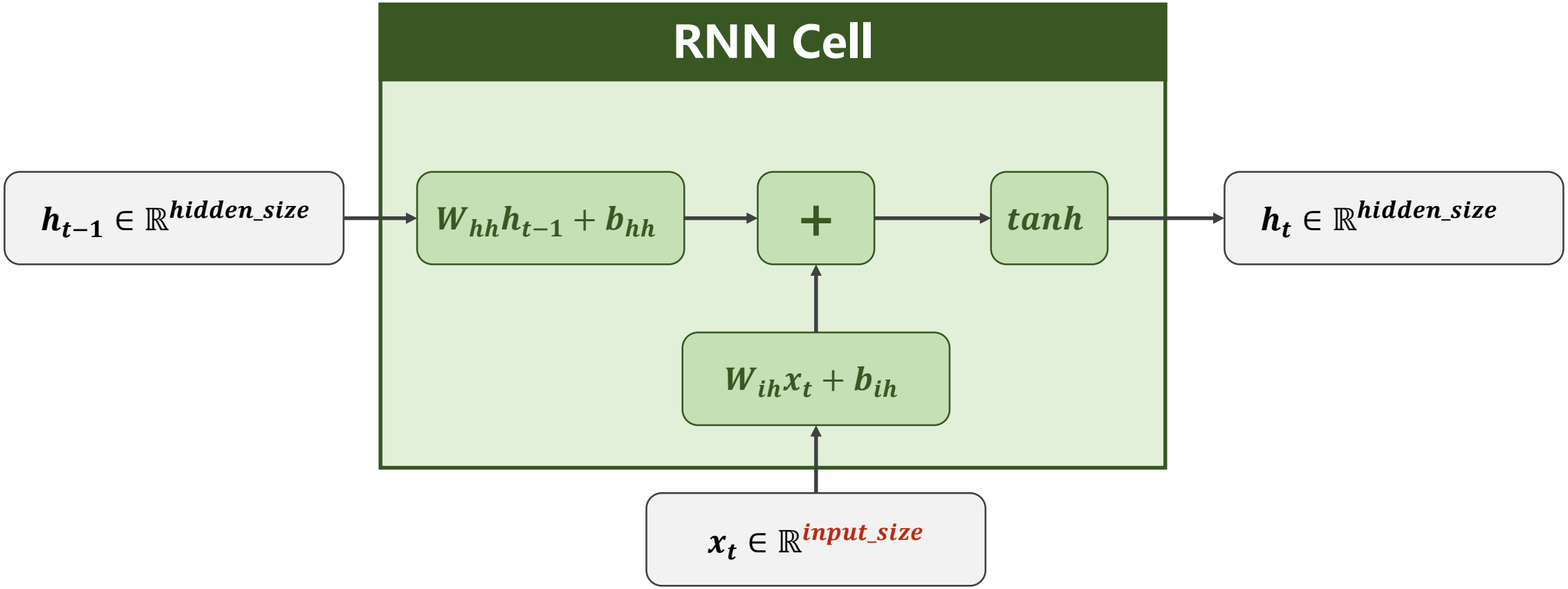
两个线性运算可以整合为1个线性层

What is RNNs?


h_t
RNN Cell
x_t
h_0
h_1
h_2
h_3
h_4
RNN Cell
RNN Cell
RNN Cell
RNN Cell
x_1
x_2
x_3
x_4

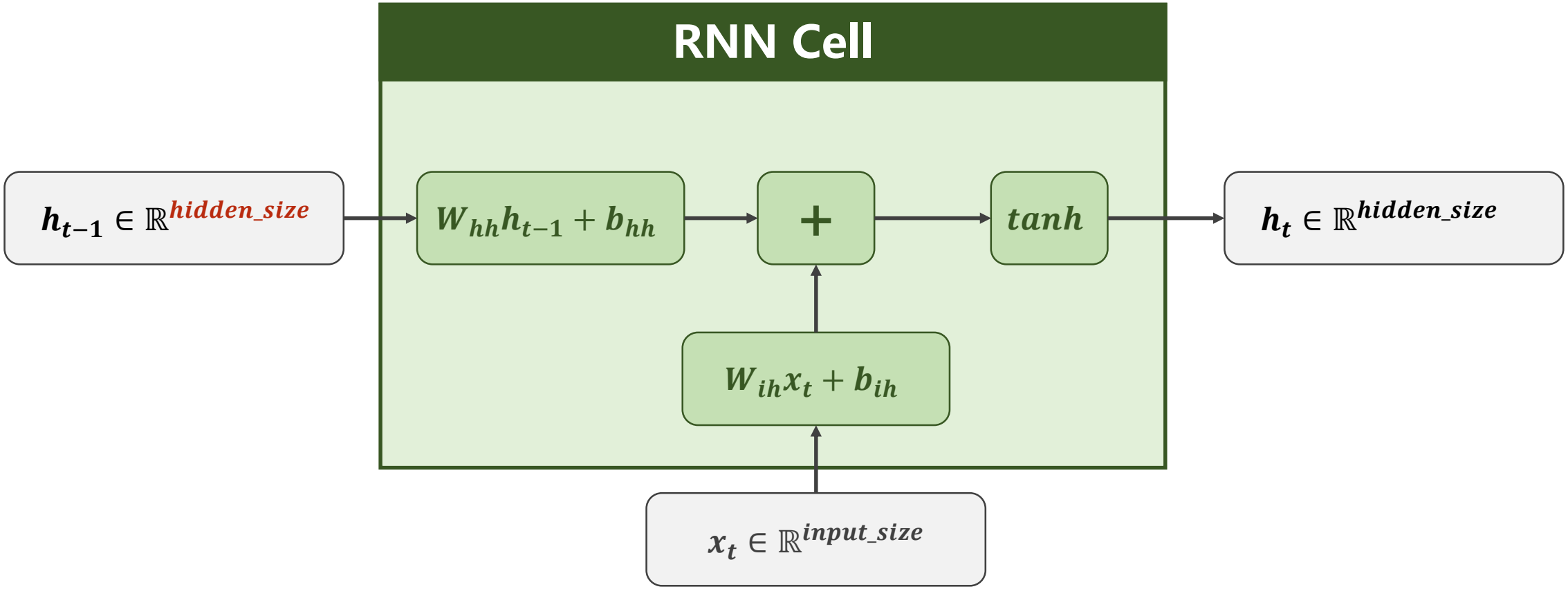
RNN Cell in PyTorch

```
cell = torch.nn.RNNCell(input_size=input_size, hidden_size=hidden_size)
```



RNN Cell in PyTorch

```
cell = torch.nn.RNNCell(input_size=input_size, hidden_size=hidden_size)
```



RNN Cell in PyTorch

```
cell = torch.nn.RNNCell(input_size=input_size, hidden_size=hidden_size)
```

```
hidden = cell(input, hidden)
```

input

input of shape $(batch, input\_size)$

hidden of shape $(batch, hidden\_size)$

output

hidden of shape $(batch, hidden\_size)$

How to use RNNCell

- Suppose we have sequence with below properties:
  - $batchSize = 1$
  - $seqLen = 3$
  - $inputSize = 4$
  - $hiddenSize = 2$

How to use RNNCell

- Suppose we have sequence with below properties:
  - $\boldsymbol{batchSize} = 1$
  - $seqLen = 3$
  - $\boldsymbol{inputSize} = 4$
  - $hiddenSize = 2$
- So the shape of inputs and outputs of RNNCell:
  - $\boldsymbol{input}.shape = (\boldsymbol{batchSize}, \boldsymbol{inputSize})$
  - $output.shape = (batchSize, hiddenSize)$

How to use RNNCell

- Suppose we have sequence with below properties:
  - $\boldsymbol{batchSize} = 1$
  - $seqLen = 3$
  - $inputSize = 4$
  - $\boldsymbol{hiddenSize} = 2$
- So the shape of inputs and outputs of RNNCell:
  - $input.shape = (batchSize, inputSize)$
  - $\boldsymbol{output}.shape = (\boldsymbol{batchSize}, \boldsymbol{hiddenSize})$

How to use RNNCell

- Suppose we have sequence with below properties:
  - $\boldsymbol{batchSize} = 1$
  - $\boldsymbol{seqLen} = 3$
  - $\boldsymbol{inputSize} = 4$
  - $hiddenSize = 2$
- So the shape of inputs and outputs of RNNCell:
  - $input.shape = (batchSize, inputSize)$
  - $output.shape = (batchSize, hiddenSize)$
- The sequence can be warped in one Tensor with shape:
  - $dataset.shape = (\boldsymbol{seqLen}, \boldsymbol{batchSize}, \boldsymbol{inputSize})$

序列长度 N

How to use RNNCell

```
import torch

batch_size = 1
seq_len = 3
input_size = 4
hidden_size = 2

cell = torch.nn.RNNCell(input_size=input_size, hidden_size=hidden_size)

# (seq, batch, features)
dataset = torch.randn(seq_len, batch_size, input_size)
hidden = torch.zeros(batch_size, hidden_size)

for idx, input in enumerate(dataset):
    print('=' * 20, idx, '=' * 20)
    print('Input size: ', input.shape)

    hidden = cell(input, hidden)

    print('outputs size: ', hidden.shape)
    print(hidden)
```

Parameters

How to use RNNCell

```
import torch

batch_size = 1
seq_len = 3
input_size = 4
hidden_size = 2

cell = torch.nn.RNNCell(input_size=input_size, hidden_size=hidden_size)

# (seq, batch, features)
dataset = torch.randn(seq_len, batch_size, input_size)
hidden = torch.zeros(batch_size, hidden_size)

for idx, input in enumerate(dataset):
    print('=' * 20, idx, '=' * 20)
    print('Input size: ', input.shape)

    hidden = cell(input, hidden)

    print('outputs size: ', hidden.shape)
    print(hidden)
```

Construction of RNNCell

How to use RNNCell

```
import torch

batch_size = 1
seq_len = 3
input_size = 4
hidden_size = 2

cell = torch.nn.RNNCell(input_size=input_size, hidden_size=hidden_size)

# (seq, batch, features)
dataset = torch.randn(seq_len, batch_size, input_size)
hidden = torch.zeros(batch_size, hidden_size)

for idx, input in enumerate(dataset):
    print('=' * 20, idx, '=' * 20)
    print('Input size: ', input.shape)

    hidden = cell(input, hidden)

    print('outputs size: ', hidden.shape)
    print(hidden)
```

Wrapping the sequence into:

$(seqLen, batchSize, inputSize)$

How to use RNNCell

```
import torch

batch_size = 1
seq_len = 3
input_size = 4
hidden_size = 2

cell = torch.nn.RNNCell(input_size=input_size, hidden_size=hidden_size)

# (seq, batch, features)
dataset = torch.randn(seq_len, batch_size, input_size)
hidden = torch.zeros(batch_size, hidden_size)

for idx, input in enumerate(dataset):
    print('=' * 20, idx, '=' * 20)
    print('Input size: ', input.shape)

    hidden = cell(input, hidden)

    print('outputs size: ', hidden.shape)
    print(hidden)
```

Initializing the ***hidden*** to **zero**

How to use RNNCell

```
import torch

batch_size = 1
seq_len = 3
input_size = 4
hidden_size = 2

cell = torch.nn.RNNCell(input_size=input_size, hidden_size=hidden_size)

# (seq, batch, features)
dataset = torch.randn(seq_len, batch_size, input_size)
hidden = torch.zeros(batch_size, hidden_size)

for idx, input in enumerate(dataset):
    print('=' * 20, idx, '=' * 20)
    print('Input size: ', input.shape)

    hidden = cell(input, hidden)

    print('outputs size: ', hidden.shape)
    print(hidden)
```

The shape of input is:

$(batchSize, inputSize)$

```
Input size:  torch.Size([1, 4])
```

How to use RNNCell

```
import torch

batch_size = 1
seq_len = 3
input_size = 4
hidden_size = 2

cell = torch.nn.RNNCell(input_size=input_size, hidden_size=hidden_size)

# (seq, batch, features)
dataset = torch.randn(seq_len, batch_size, input_size)
hidden = torch.zeros(batch_size, hidden_size)

for idx, input in enumerate(dataset):
    print('=' * 20, idx, '=' * 20)
    print('Input size: ', input.shape)

    hidden = cell(input, hidden)

    print('hidden size: ', hidden.shape)
    print(hidden)
```

The shape of *hidden* is:

$(batchSize, hiddenSize)$

```
hidden size:  torch.Size([1, 2])
```

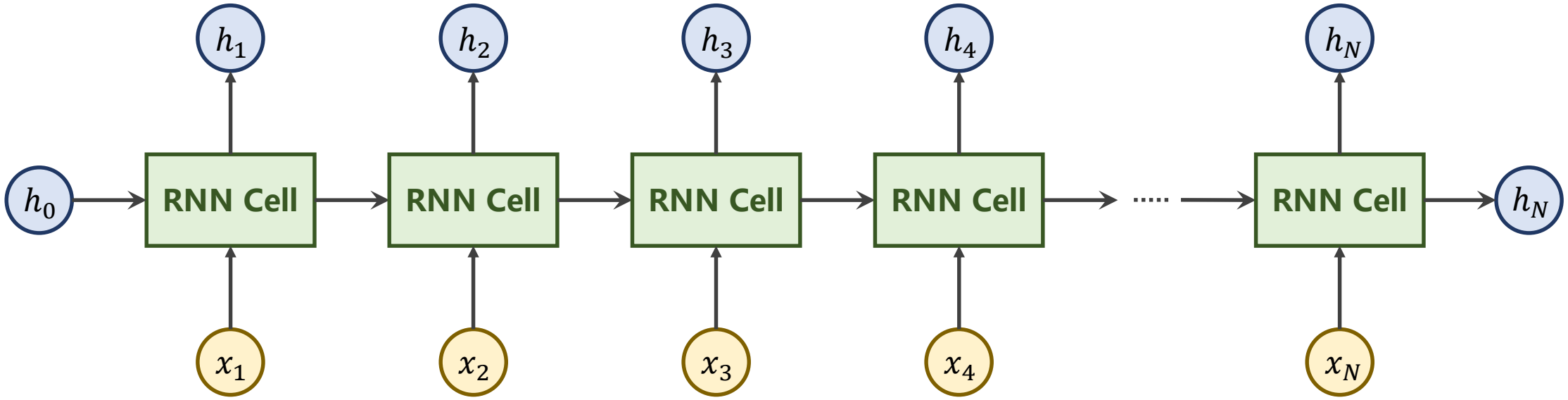
How to use RNNCell

```
import torch

batch_size = 1
seq_len = 3
input_size = 4
hidden_size = 2

cell = torch.nn.RNNCell(input_size=input_size, hidden_size=hidden_size)

# (seq, batch, features)
dataset = torch.randn(seq_len, batch_size, input_size)
hidden = torch.zeros(batch_size, hidden_size)

for idx, input in enumerate(dataset):
    print('=' * 20, idx, '=' * 20)
    print('Input size: ', input.shape)

    hidden = cell(input, hidden)

    print('outputs size: ', hidden.shape)
    print(hidden)
```

```
========== 0 ==========
Input size:  torch.Size([1, 4])
outputs size:  torch.Size([1, 2])
tensor([[-0.1579,  0.5140]])
========== 1 ==========
Input size:  torch.Size([1, 4])
outputs size:  torch.Size([1, 2])
tensor([[-0.9577,  0.6502]])
========== 2 ==========
Input size:  torch.Size([1, 4])
outputs size:  torch.Size([1, 2])
tensor([[-0.7661, -0.9128]])
```

How to use RNN

```
cell = torch.nn.RNN(input_size=input_size, hidden_size=hidden_size,
                    num_layers=num_layers)
```



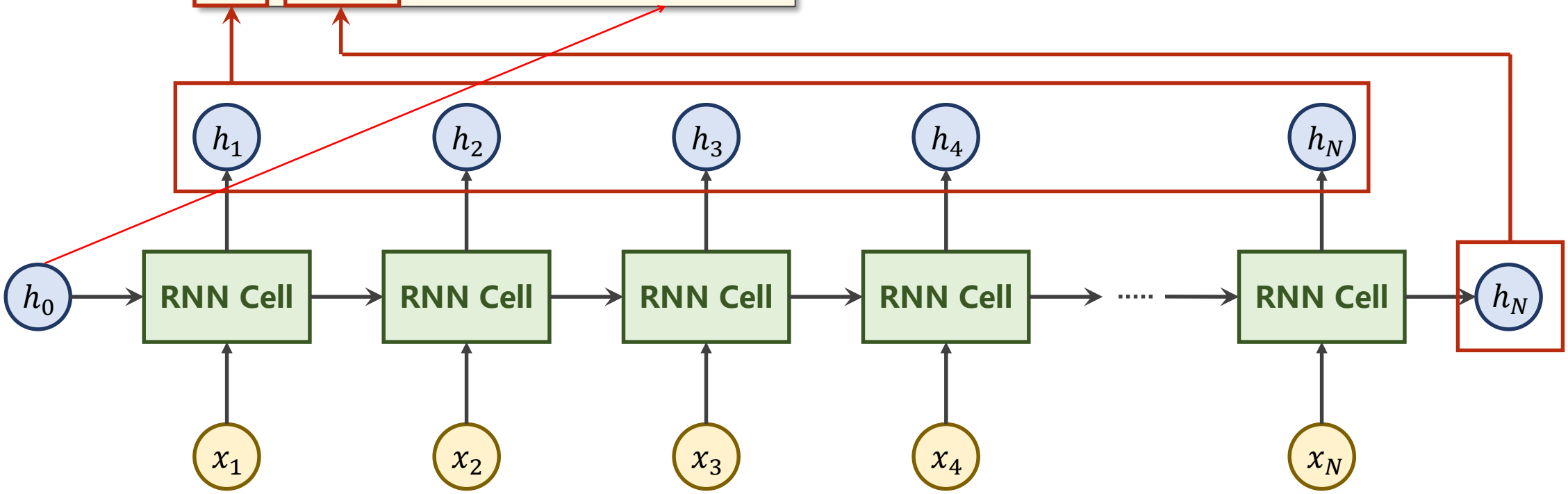
How to use RNN

```
cell = torch.nn.RNN(input_size=input_size, hidden_size=hidden_size,
                    num_layers=num_layers)
```

RNN的层数

整个输入序列

```
out, hidden = cell(inputs, hidden)
```



How to use RNN

```
cell = torch.nn.RNN(input_size=input_size, hidden_size=hidden_size,
                    num_layers=num_layers)
```

```
out, hidden = cell(inputs, hidden)
```

input

input of shape $(seqSize, batch, input\_size)$

hidden of shape $(numLayers, batch, hidden\_size)$

output

Output of shape $(seqLen, batch, hidden\_size)$

hidden of shape $(numLayers, batch, hidden\_size)$

How to use RNN

- Suppose we have sequence with below properties:
  - $batchSize$
  - $seqLen$
  - $inputSize, hiddenSize,$
  - $numLayers$

How to use RNN

- Suppose we have sequence with below properties:
  - $\boldsymbol{batchSize}$
  - $\boldsymbol{seqLen}$
  - $\boldsymbol{inputSize}, hiddenSize,$
  - $numLayers$
- The shape of ***input*** and ***h_0*** of RNN:
  - $input.shape = (\boldsymbol{seqLen}, \boldsymbol{batchSize}, \boldsymbol{inputSize})$
  - $h\_0.shape = (numLayers, batchSize, hiddenSize)$

How to use RNN

- Suppose we have sequence with below properties:
  - $\boldsymbol{batchSize}$
  - $seqLen$
  - $inputSize$, $\boldsymbol{hiddenSize}$,
  - $\boldsymbol{numLayers}$
- The shape of ***input*** and ***h_0*** of RNN:
  - $input.shape = (seqLen, batchSize, inputSize)$
  - $h\_0.shape = (\boldsymbol{numLayers}, \boldsymbol{batchSize}, \boldsymbol{hiddenSize})$

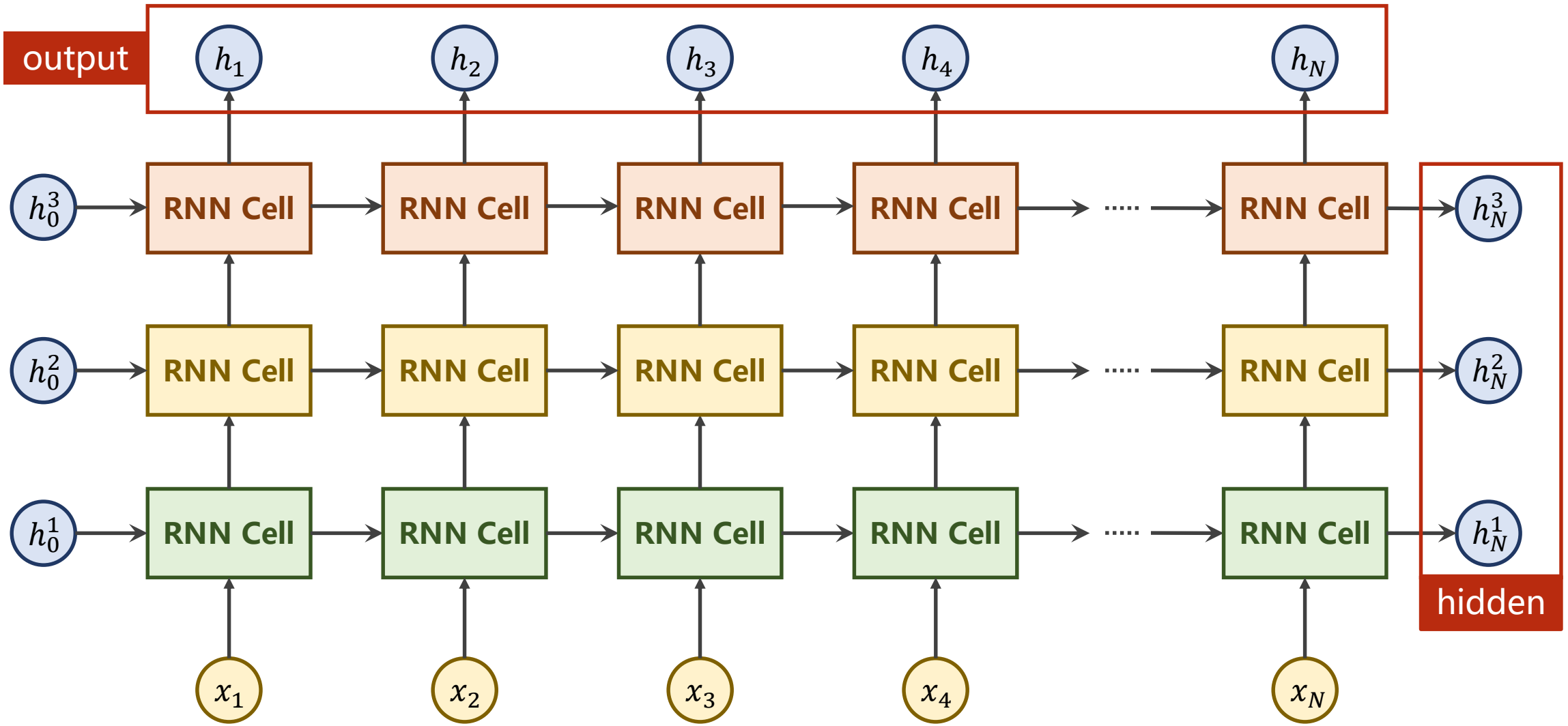
How to use RNN

- Suppose we have sequence with below properties:
  - $\boldsymbol{batchSize}$
  - $\boldsymbol{seqLen}$
  - $inputSize$, $\boldsymbol{hiddenSize}$,
  - $numLayers$
- The shape of ***input*** and ***h_0*** of RNN:
  - $input.shape = (seqLen, batchSize, inputSize)$
  - $h\_0.shape = (numLayers, batchSize, hiddenSize)$
- The shape of ***output*** and ***h_n*** of RNN:
  - $output.shape = (\boldsymbol{seqLen}, \boldsymbol{batchSize}, \boldsymbol{hiddenSize})$
  - $h\_n.shape = (numLayers, batchSize, hiddenSize)$

How to use RNN

- Suppose we have sequence with below properties:
  - $\boldsymbol{batchSize}$
  - $seqLen$
  - $inputSize$, $\boldsymbol{hiddenSize}$,
  - $\boldsymbol{numLayers}$
- The shape of ***input*** and ***h_0*** of RNN:
  - $input.shape = (seqLen, batchSize, inputSize)$
  - $h\_0.shape = (numLayers, batchSize, hiddenSize)$
- The shape of ***output*** and ***h_n*** of RNN:
  - $output.shape = (seqLen, batchSize, hiddenSize)$
  - $h\_n.shape = (\boldsymbol{numLayers}, \boldsymbol{batchSize}, \boldsymbol{hiddenSize})$

How to use RNN - numLayers



How to use RNN

```
import torch

batch_size = 1
seq_len = 3
input_size = 4
hidden_size = 2
num_layers = 1

cell = torch.nn.RNN(input_size=input_size, hidden_size=hidden_size,
                    num_layers=num_layers)

# (seqLen, batchSize, inputSize)
inputs = torch.randn(seq_len, batch_size, input_size)
hidden = torch.zeros(num_layers, batch_size, hidden_size)

out, hidden = cell(inputs, hidden)

print('Output size:', out.shape)
print('Output:', out)
print('Hidden size: ', hidden.shape)
print('Hidden: ', hidden)
```

Parameters

How to use RNN

```
import torch

batch_size = 1
seq_len = 3
input_size = 4
hidden_size = 2
num_layers = 1

cell = torch.nn.RNN(input_size=input_size, hidden_size=hidden_size,
                    num_layers=num_layers)

# (seqLen, batchSize, inputSize)
inputs = torch.randn(seq_len, batch_size, input_size)
hidden = torch.zeros(num_layers, batch_size, hidden_size)

out, hidden = cell(inputs, hidden)

print('Output size:', out.shape)
print('Output:', out)
print('Hidden size: ', hidden.shape)
print('Hidden: ', hidden)
```

Construction of RNN

How to use RNN

```
import torch

batch_size = 1
seq_len = 3
input_size = 4
hidden_size = 2
num_layers = 1

cell = torch.nn.RNN(input_size=input_size, hidden_size=hidden_size,
                    num_layers=num_layers)

# (seqLen, batchSize, inputSize)
inputs = torch.randn(seq_len, batch_size, input_size)
hidden = torch.zeros(num_layers, batch_size, hidden_size)

out, hidden = cell(inputs, hidden)

print('Output size:', out.shape)
print('Output:', out)
print('Hidden size: ', hidden.shape)
print('Hidden: ', hidden)
```

Wrapping the sequence into:

$(seqLen, batchSize, inputSize)$

How to use RNN

```
import torch

batch_size = 1
seq_len = 3
input_size = 4
hidden_size = 2
num_layers = 1

cell = torch.nn.RNN(input_size=input_size, hidden_size=hidden_size,
                    num_layers=num_layers)

# (seqLen, batchSize, inputSize)
inputs = torch.randn(seq_len, batch_size, input_size)
hidden = torch.zeros(num_layers, batch_size, hidden_size)

out, hidden = cell(inputs, hidden)

print('Output size:', out.shape)
print('Output:', out)
print('Hidden size: ', hidden.shape)
print('Hidden: ', hidden)
```

Initializing the ***hidden*** to **zero**

How to use RNN

```
import torch

batch_size = 1
seq_len = 3
input_size = 4
hidden_size = 2
num_layers = 1

cell = torch.nn.RNN(input_size=input_size, hidden_size=hidden_size,
                    num_layers=num_layers)

# (seqLen, batchSize, inputSize)
inputs = torch.randn(seq_len, batch_size, input_size)
hidden = torch.zeros(num_layers, batch_size, hidden_size)

out, hidden = cell(inputs, hidden)

print('Output size:', out.shape)
print('Output:', out)
print('Hidden size: ', hidden.shape)
print('Hidden: ', hidden)
```

The shape of *output* is:

$(seqSize, batchSize, hiddenSize)$

How to use RNN

```
import torch

batch_size = 1
seq_len = 3
input_size = 4
hidden_size = 2
num_layers = 1

cell = torch.nn.RNN(input_size=input_size, hidden_size=hidden_size,
                    num_layers=num_layers)

# (seqLen, batchSize, inputSize)
inputs = torch.randn(seq_len, batch_size, input_size)
hidden = torch.zeros(num_layers, batch_size, hidden_size)

out, hidden = cell(inputs, hidden)

print('Output size:', out.shape)
print('Output:', out)
print('Hidden size: ', hidden.shape)
print('Hidden: ', hidden)
```

The shape of *hidden* is:

$(numLayers, batchSize, hiddenSize)$

How to use RNN

```
import torch

batch_size = 1
seq_len = 3
input_size = 4
hidden_size = 2
num_layers = 1

cell = torch.nn.RNN(input_size=input_size, hidden_size=hidden_size,
                    num_layers=num_layers)

# (seqLen, batchSize, inputSize)
inputs = torch.randn(seq_len, batch_size, input_size)
hidden = torch.zeros(num_layers, batch_size, hidden_size)

out, hidden = cell(inputs, hidden)

print('Output size:', out.shape)
print('Output:', out)
print('Hidden size: ', hidden.shape)
print('Hidden: ', hidden)
```

```
Output size: torch.Size([3, 1, 2])
Output: tensor([
        [[ 0.2360, -0.5550]],
        [[ 0.2685,  0.0106]],
        [[ 0.1172, -0.2262]]])
Hidden size:  torch.Size([1, 1, 2])
Hidden:  tensor([[[ 0.1172, -0.2262]]])
```

How to use RNN

- **batch_first**: if True, the input and output tensors are provided as:
  - $(batchSize, seqLen, input\_size)$

```
cell = torch.nn.RNN(input_size=input_size, hidden_size=hidden_size,
                    num_layers=num_layers, batch_first=True)
```

```
inputs = torch.randn(seq_len, batch_size, input_size)
```

```
inputs = torch.randn(batch_size, seq_len, input_size)
```

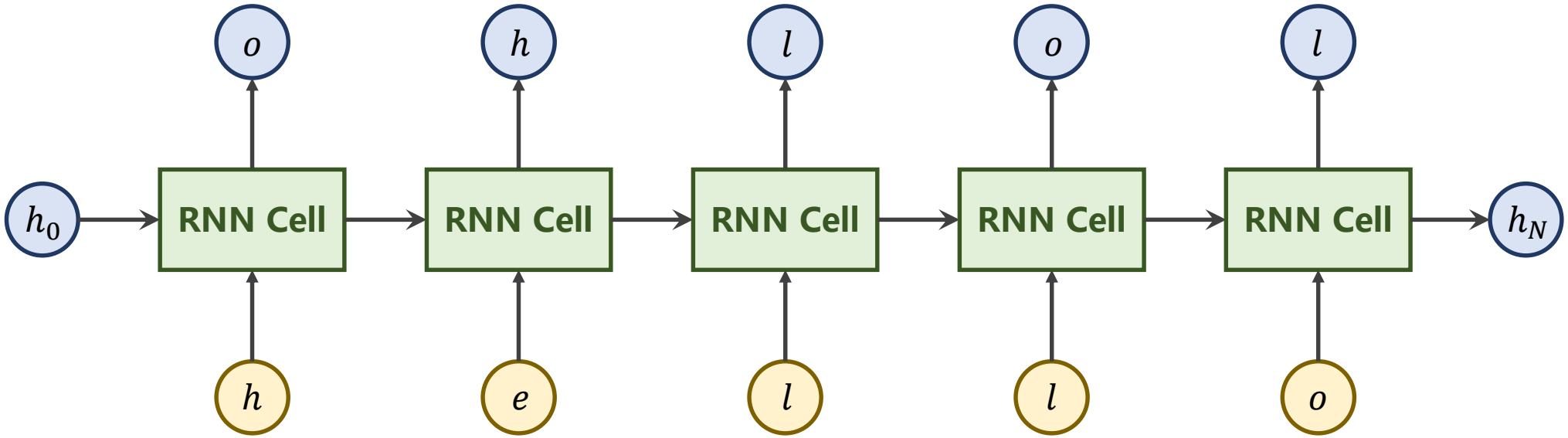
How to use RNN

```
import torch

batch_size = 1
seq_len = 3
input_size = 4
hidden_size = 2
num_layers = 1

cell = torch.nn.RNN(input_size=input_size, hidden_size=hidden_size,
                    num_layers=num_layers, batch_first=True)

# (seqLen, batchSize, inputSize)
inputs = torch.randn(batch_size, seq_len, input_size)
hidden = torch.zeros(num_layers, batch_size, hidden_size)

out, hidden = cell(inputs, hidden)

print('Output size:', out.shape)
print('Output:', out)
print('Hidden size: ', hidden.shape)
print('Hidden: ', hidden)
```

```
Output size: torch.Size([1, 3, 2])
Output: tensor([[
        [ 0.6582, -0.7281],
        [ 0.5021, -0.8324],
        [-0.5225, -0.5621]]])
Hidden size:  torch.Size([1, 1, 2])
Hidden:  tensor([[[-0.5225, -0.5621]]])
```

Example 12-1: Using RNNCell

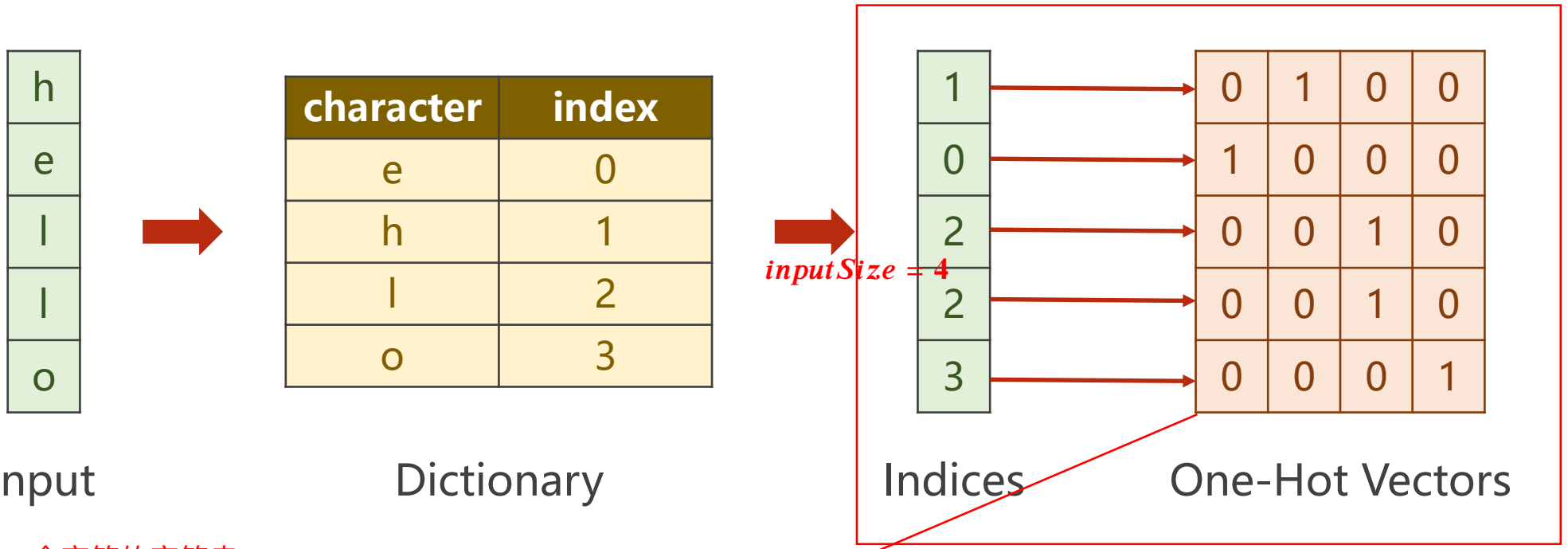
- Train a model to learn:
  - “hello” -> “ohlol” 训练RNN网络来实现序列的转换



Example 12-1: Using RNNCell

- The inputs of RNN Cell should be vectors of numbers.

RNN的输入不能是字符串，应该是向量。首先构造词典，然后将字符串使用one-hot编码为向量。

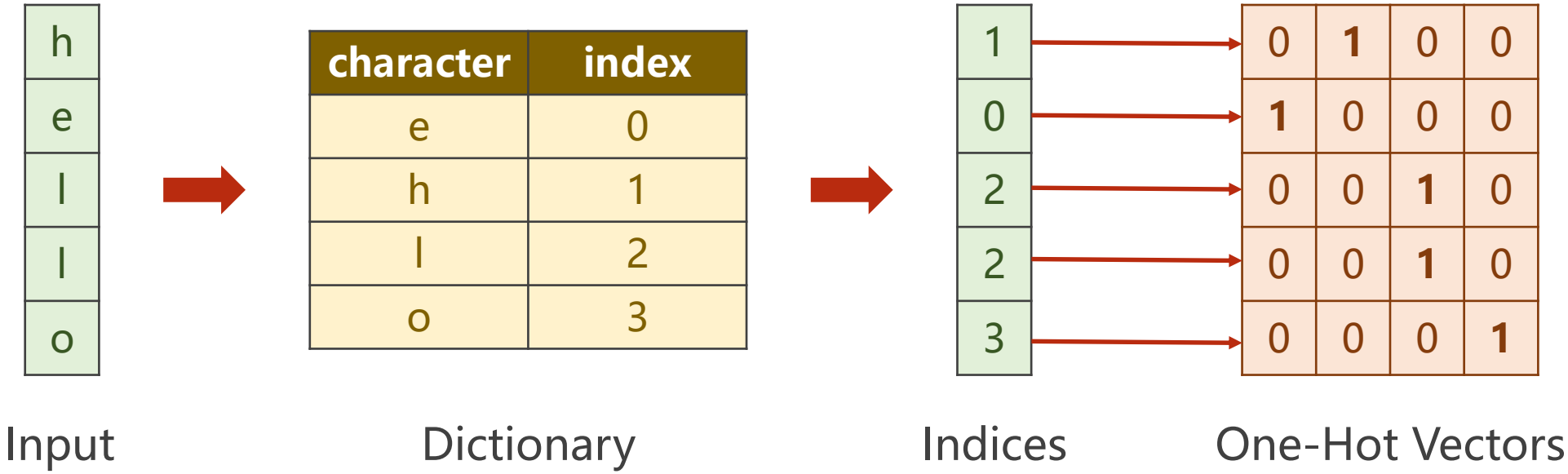


长度为n的包含m个字符的字符串，
经过独热编码之后的向量大小为n*m

One-Hot Vectors的宽度与词典的长度一致
One-Hot Vectors中只有0和1

Example 12-1: Using RNNCell

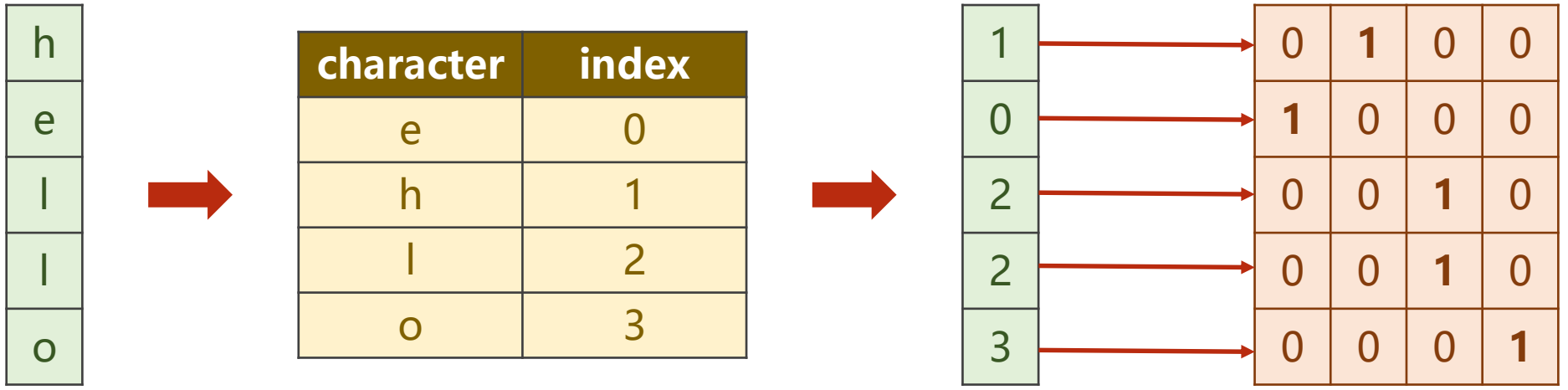
- The inputs of RNN Cell should be vectors of numbers.



$$inputSize = ?$$

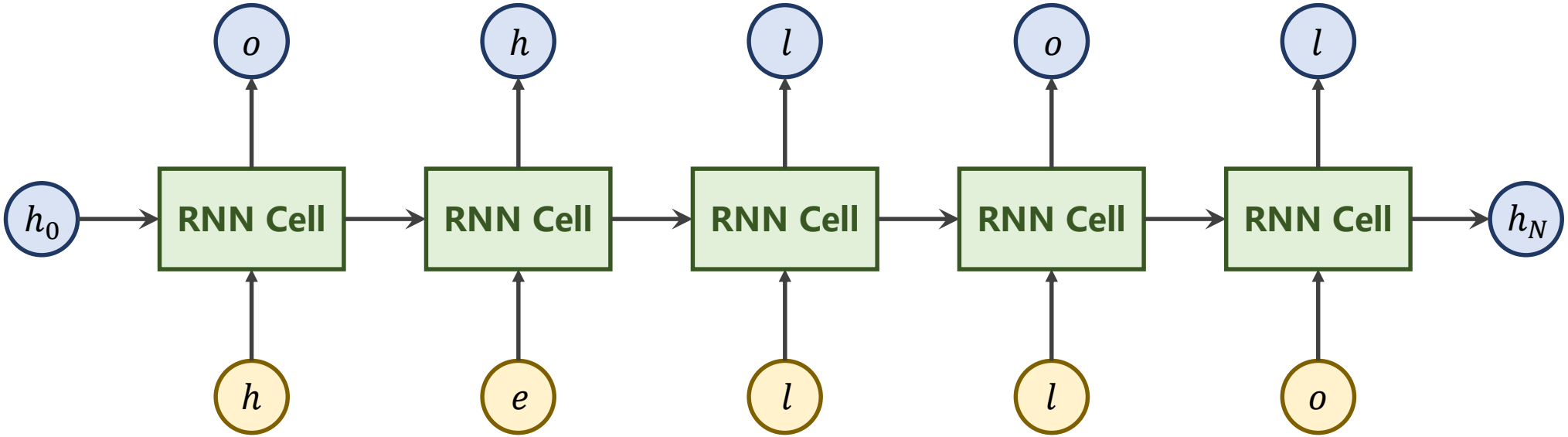
Example 12-1: Using RNNCell

- The inputs of RNN Cell should be vectors of numbers.



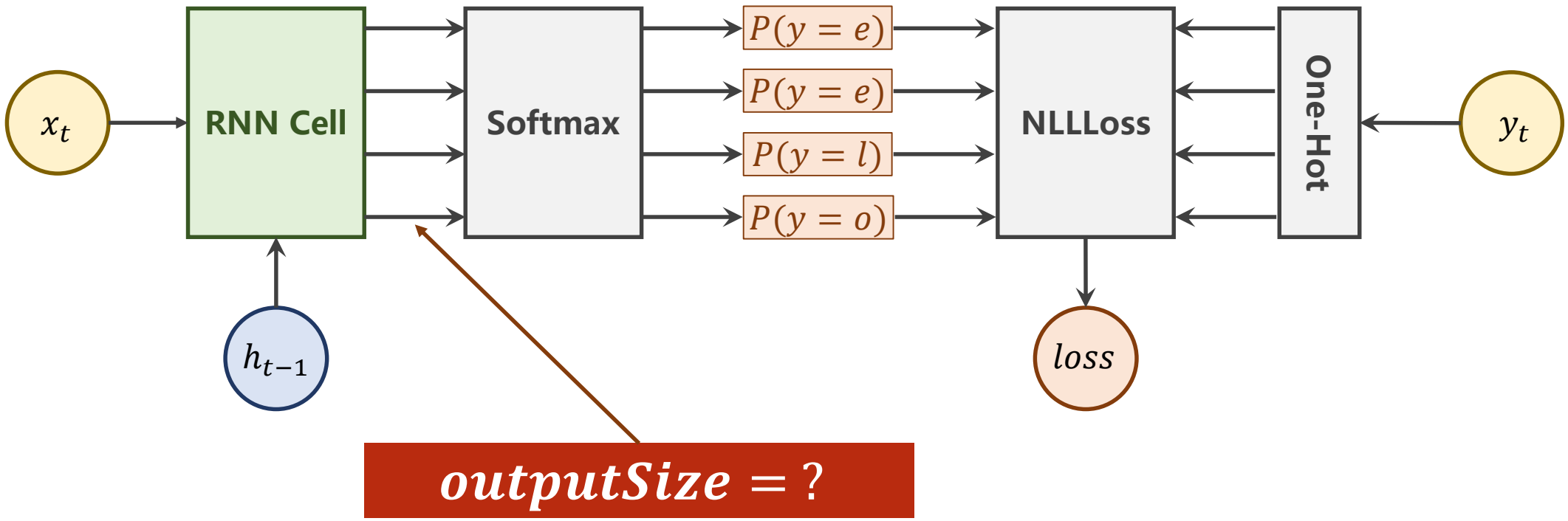
$$inputSize = 4$$

Example 12-1: Using RNNCell



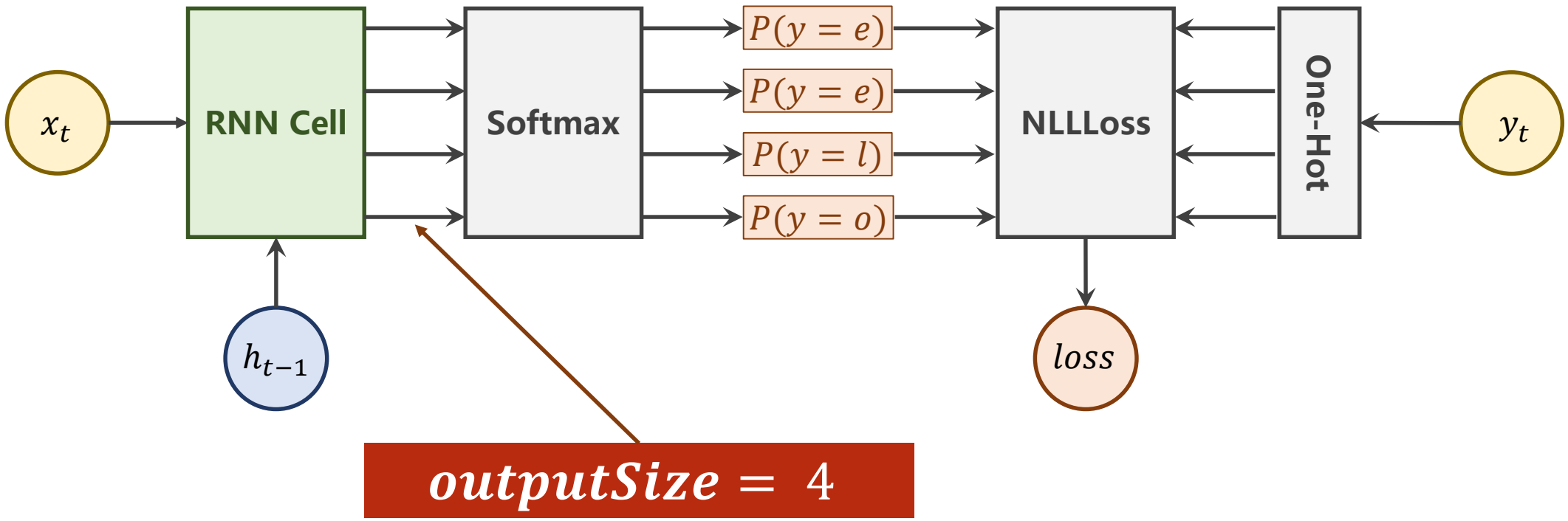
Example 12-1: Using RNNCell

- The outputs of RNN Cell should be vectors of prediction.



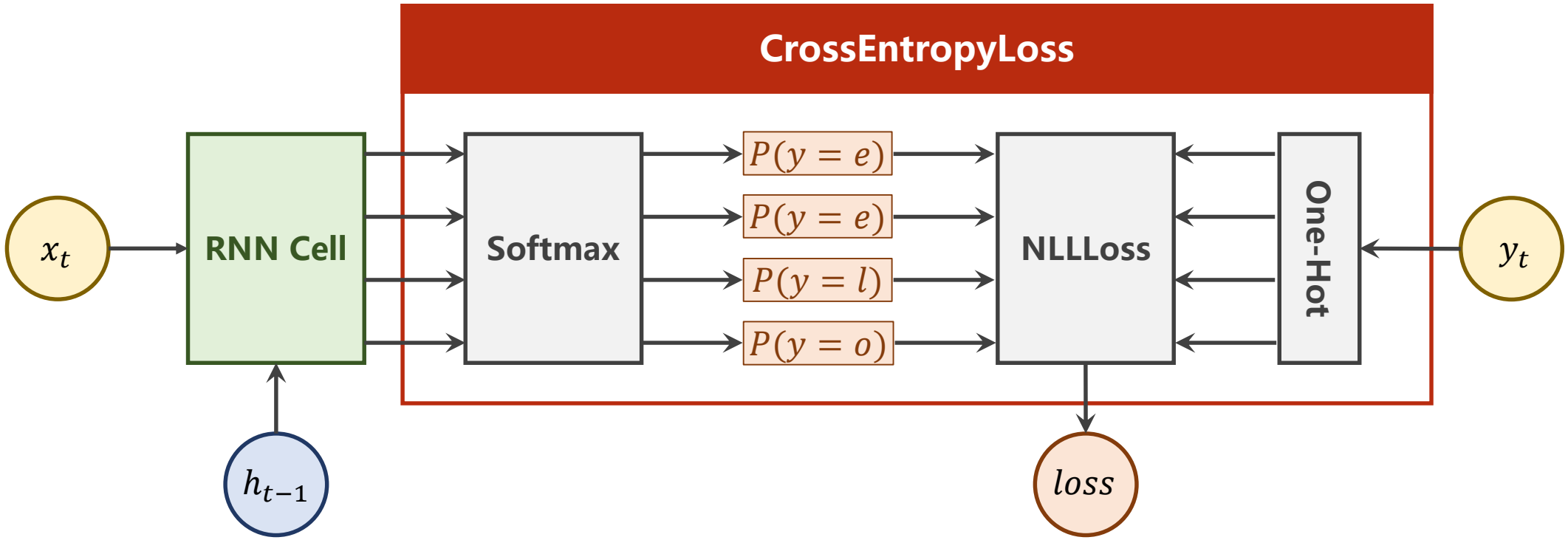
Example 12-1: Using RNNCell

- The outputs of RNN Cell should be vectors of prediction.



Example 12-1: Using RNNCell

- The outputs of RNN Cell should be vectors of prediction.



$$outputSize = 4$$

Example 12-1: Code – Parameters

```
import torch

input_size = 4
hidden_size = 4
batch_size = 1
```

Example 12-1: Code – Prepare Data

```
idx2char = ['e', 'h', 'l', 'o']
x_data = [1, 0, 2, 2, 3]
y_data = [3, 1, 2, 3, 2]

one_hot_lookup = [[1, 0, 0, 0],
                  [0, 1, 0, 0],
                  [0, 0, 1, 0],
                  [0, 0, 0, 1]]
x_one_hot = [one_hot_lookup[x] for x in x_data]

inputs = torch.Tensor(x_one_hot).view(-1, batch_size, input_size)
labels = torch.LongTensor(y_data).view(-1, 1)
```

The dictionary

Example 12-1: Code – Prepare Data

```
idx2char = ['e', 'h', 'l', 'o']
x_data = [1, 0, 2, 2, 3]
y_data = [3, 1, 2, 3, 2]

one_hot_lookup = [[1, 0, 0, 0],
                  [0, 1, 0, 0],
                  [0, 0, 1, 0],
                  [0, 0, 0, 1]]
x_one_hot = [one_hot_lookup[x] for x in x_data]

inputs = torch.Tensor(x_one_hot).view(-1, batch_size, input_size)
labels = torch.LongTensor(y_data).view(-1, 1)
```

The input sequence is 'hello'

Example 12-1: Code – Prepare Data

```
idx2char = ['e', 'h', 'l', 'o']
x_data = [1, 0, 2, 2, 3]
y_data = [3, 1, 2, 3, 2]

one_hot_lookup = [[1, 0, 0, 0],
                  [0, 1, 0, 0],
                  [0, 0, 1, 0],
                  [0, 0, 0, 1]]
x_one_hot = [one_hot_lookup[x] for x in x_data]

inputs = torch.Tensor(x_one_hot).view(-1, batch_size, input_size)
labels = torch.LongTensor(y_data).view(-1, 1)
```

The output sequence is 'ohlol'

Example 12-1: Code – Prepare Data

```
idx2char = ['e', 'h', 'l', 'o']
x_data = [1, 0, 2, 2, 3]
y_data = [3, 1, 2, 3, 2]

one_hot_lookup = [[1, 0, 0, 0],
                  [0, 1, 0, 0],
                  [0, 0, 1, 0],
                  [0, 0, 0, 1]]
x_one_hot = [one_hot_lookup[x] for x in x_data]

inputs = torch.Tensor(x_one_hot).view(-1, batch_size, input_size)
labels = torch.LongTensor(y_data).view(-1, 1)
```

Convert indices into one-hot vector

Example 12-1: Code – Prepare Data

```
idx2char = ['e', 'h', 'l', 'o']
x_data = [1, 0, 2, 2, 3]
y_data = [3, 1, 2, 3, 2]

one_hot_lookup = [[1, 0, 0, 0],
                  [0, 1, 0, 0],
                  [0, 0, 1, 0],
                  [0, 0, 0, 1]]
x_one_hot = [one_hot_lookup[x] for x in x_data]

inputs = torch.Tensor(x_one_hot).view(-1, batch_size, input_size)
labels = torch.LongTensor(y_data).view(-1, 1)
```

Reshape the inputs to

$(seqLen, batchSize, inputSize)$

Example 12-1: Code – Prepare Data

```
idx2char = ['e', 'h', 'l', 'o']
x_data = [1, 0, 2, 2, 3]
y_data = [3, 1, 2, 3, 2]

one_hot_lookup = [[1, 0, 0, 0],
                  [0, 1, 0, 0],
                  [0, 0, 1, 0],
                  [0, 0, 0, 1]]
x_one_hot = [one_hot_lookup[x] for x in x_data]

inputs = torch.Tensor(x_one_hot).view(-1, batch_size, input_size)
labels = torch.LongTensor(y_data).view(-1, 1)
```

Reshape the labels to

$(seqLen, 1)$

Example 12-1: Code – Design Model

```
class Model(torch.nn.Module):
    def __init__(self, input_size, hidden_size, batch_size):
        super(Model, self).__init
        self.batch_size = batch_size
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.rnncell = torch.nn.RNNCell(input_size=self.input_size,
                                        hidden_size=self.hidden_size)

    def forward(self, input, hidden):
        hidden = self.rnncell(input, hidden)
        return hidden

    def init_hidden(self):
        return torch.zeros(self.batch_size, self.hidden_size)

net = Model(input_size, hidden_size, batch_size)
```

Initial the parameters

Example 12-1: Code – Design Model

```
class Model(torch.nn.Module):
    def __init__(self, input_size, hidden_size, batch_size):
        super(Model, self).__init__()
        # self.num_layers = num_layers
        self.batch_size = batch_size
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.rnncell = torch.nn.RNNCell(input_size=self.input_size,
                                        hidden_size=self.hidden_size)

    def forward(self, input, hidden):
        hidden = self.rnncell(input, hidden)
        return hidden

    def init_hidden(self):
        return torch.zeros(self.batch_size, self.hidden_size)

net = Model(input_size, hidden_size, batch_size)
```

Shape of inputs :

$(batchSize, inputSize)$

Shape of hidden:

$(batchSize, hiddenSize)$

Example 12-1: Code – Design Model

```
class Model(torch.nn.Module):
    def __init__(self, input_size, hidden_size, batch_size):
        super(Model, self).__init__()
        # self.num_layers = num_layers
        self.batch_size = batch_size
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.rnncell = torch.nn.RNNCell(input_size=self.input_size,
                                        hidden_size=self.hidden_size)

    def forward(self, input, hidden):
        hidden = self.rnncell(input, hidden)
        return hidden

    def init_hidden(self):
        return torch.zeros(self.batch_size, self.hidden_size)

net = Model(input_size, hidden_size, batch_size)
```

Provide initial hidden

Example 12-1: Code – Loss and Optimizer

```
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=0.1)
```

Example 12-1: Code – Training Cycle

```
for epoch in range(15):
    loss = 0
    optimizer.zero_grad()
    hidden = net.init_hidden()
    print('Predicted string: ', end='')
    for input, label in zip(inputs, labels):
        hidden = net(input, hidden)
        loss += criterion(hidden, label)
        _, idx = hidden.max(dim=1)
        print(idx2char[idx.item()], end='')
    loss.backward()
    optimizer.step()
    print(', Epoch [%d/15] loss=%.4f' % (epoch+1, loss.item()))
```

Training steps

Example 12-1: Code – Training Cycle

```
for epoch in range(15):
    loss = 0
    optimizer.zero_grad()
    hidden = net.init_hidden()
    print('Predicted string: ', end='')
    for input, label in zip(inputs, labels):
        hidden = net(input, hidden)
        loss += criterion(hidden, label)
        _, idx = hidden.max(dim=1)
        print(idx2char[idx.item()], end='')
    loss.backward()
    optimizer.step()
    print(', Epoch [%d/15] loss=%.4f' % (epoch+1, loss.item()))
```

Shape of *inputs*:

$(seqLen, batchSize, inputSize)$

Shape of *input*:

$(batchSize, hiddenSize)$

Example 12-1: Code – Training Cycle

```
for epoch in range(15):
    loss = 0
    optimizer.zero_grad()
    hidden = net.init_hidden()
    print('Predicted string: ', end='')
    for input, label in zip(inputs, labels):
        hidden = net(input, hidden)
        loss += criterion(hidden, label)
        _, idx = hidden.max(dim=1)
        print(idx2char[idx.item()], end='')
    loss.backward()
    optimizer.step()
    print(', Epoch [%d/15] loss=%.4f' % (epoch+1, loss.item()))
```

Shape of *labels*:

$(seqSize, 1)$

Shape of *label*:

(1)

Example 12-1: Code – Training Cycle

```
for epoch in range(15):
    loss = 0
    optimizer.zero_grad()
    hidden = net.init_hidden()
    print('Predicted string: ', end='')
    for input, label in zip(inputs, labels):
        hidden = net(input, hidden)
        loss += criterion(hidden, label)
        _, idx = hidden.max(dim=1)
        print(idx2char[idx.item()], end='')
    loss.backward()
    optimizer.step()
    print(', Epoch [%d/15] loss=%.4f' % (epoch+1, loss.item()))
```

RNN Cell

Example 12-1: Code – Training Cycle

```
for epoch in range(15):
    loss = 0
    optimizer.zero_grad()
    hidden = net.init_hidden()
    print('Predicted string: ', end='')
    for input, label in zip(inputs, labels):
        hidden = net(input, hidden)
        loss += criterion(hidden, label)
        _, idx = hidden.max(dim=1)
        print(idx2char[idx.item()], end='')
    loss.backward()
    optimizer.step()
    print(', Epoch [%d/15] loss=%.4f' % (epoch+1, loss.item()))
```

Output prediction

Example 12-1: Code – Result

```
Predicted string: loeeh, Epoch [1/15] loss=8.0117
Predicted string: ooeeh, Epoch [2/15] loss=7.2082
Predicted string: ooooh, Epoch [3/15] loss=6.6208
Predicted string: ooooh, Epoch [4/15] loss=6.1802
Predicted string: ooooh, Epoch [5/15] loss=5.8060
Predicted string: ooooh, Epoch [6/15] loss=5.4739
Predicted string: ooooh, Epoch [7/15] loss=5.1593
Predicted string: ooool, Epoch [8/15] loss=4.8593
Predicted string: ooool, Epoch [9/15] loss=4.5819
Predicted string: ohool, Epoch [10/15] loss=4.3287
Predicted string: ohlol, Epoch [11/15] loss=4.0909
Predicted string: ohlol, Epoch [12/15] loss=3.8573
Predicted string: ohlol, Epoch [13/15] loss=3.6246
Predicted string: ohlol, Epoch [14/15] loss=3.4007
Predicted string: ohlol, Epoch [15/15] loss=3.2005
```

Example 12-2 Using RNN Module

```
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=0.05)

for epoch in range(15):
    optimizer.zero_grad()
    outputs = net(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

    _, idx = outputs.max(dim=1)
    idx = idx.data.numpy()
    print('Predicted: ', ''.join([idx2char[x] for x in idx]), end='')
    print(', Epoch [%d/15] loss = %.3f' % (epoch + 1, loss.item()))
```

Training steps

Example 12-2 Change Model

```
class Model(torch.nn.Module):
    def __init__(self, input_size, hidden_size, batch_size, num_layers=1):
        super(Model, self).__init__()
        self.num_layers = num_layers
        self.batch_size = batch_size
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.rnn = torch.nn.RNN(input_size=self.input_size,
                                hidden_size=self.hidden_size,
                                num_layers=num_layers)

    def forward(self, input):
        hidden = torch.zeros(self.num_layers,
                             self.batch_size,
                             self.hidden_size)
        out, _ = self.rnn(input, hidden)
        return out.view(-1, self.hidden_size)

net = Model(input_size, hidden_size, batch_size, num_layers)
```

```
input_size = 4
hidden_size = 4
num_layers = 1
batch_size = 1
seq_len = 5
```

Example 12-2 Change Model

```
class Model(torch.nn.Module):
    def __init__(self, input_size, hidden_size, batch_size, num_layers=1):
        super(Model, self).__init__()
        self.num_layers = num_layers
        self.batch_size = batch_size
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.rnn = torch.nn.RNN(input_size=self.input_size,
                                hidden_size=self.hidden_size,
                                num_layers=num_layers)

    def forward(self, input):
        hidden = torch.zeros(self.num_layers,
                             self.batch_size,
                             self.hidden_size)
        out, _ = self.rnn(input, hidden)
        return out.view(-1, self.hidden_size)

net = Model(input_size, hidden_size, batch_size, num_layers)
```

```
input_size = 4
hidden_size = 4
num_layers = 1
batch_size = 1
seq_len = 5
```

Example 12-2 Change Model

```
class Model(torch.nn.Module):
    def __init__(self, input_size, hidden_size, batch_size, num_layers=1):
        super(Model, self).__init__()
        self.num_layers = num_layers
        self.batch_size = batch_size
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.rnn = torch.nn.RNN(input_size=self.input_size,
                                hidden_size=self.hidden_size,
                                num_layers=num_layers)

    def forward(self, input):
        hidden = torch.zeros(self.num_layers,
                             self.batch_size,
                             self.hidden_size)
        out, _ = self.rnn(input, hidden)
        return out.view(-1, self.hidden_size)

net = Model(input_size, hidden_size, batch_size, num_layers)
```

```
input_size = 4
hidden_size = 4
num_layers = 1
batch_size = 1
seq_len = 5
```

Shape of *hidden* :

$(numLayers, batchSize, hiddenSize)$

Example 12-2 Change Model

```
class Model(torch.nn.Module):
    def __init__(self, input_size, hidden_size, batch_size, num_layers=1):
        super(Model, self).__init__()
        self.num_layers = num_layers
        self.batch_size = batch_size
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.rnn = torch.nn.RNN(input_size=self.input_size,
                                hidden_size=self.hidden_size,
                                num_layers=num_layers)

    def forward(self, input):
        hidden = torch.zeros(self.num_layers,
                             self.batch_size,
                             self.hidden_size)
        out, _ = self.rnn(input, hidden)
        return out.view(-1, self.hidden_size)

net = Model(input_size, hidden_size, batch_size, num_layers)
```

Reshape *out* to:

$(seqLen \times batchSize, hiddenSize)$

Example 12-2 Change Data

```
idx2char = ['e', 'h', 'l', 'o']
x_data = [1, 0, 2, 2, 3]
y_data = [3, 1, 2, 3, 2]

one_hot_lookup = [[1, 0, 0, 0],
                  [0, 1, 0, 0],
                  [0, 0, 1, 0],
                  [0, 0, 0, 1]]
x_one_hot = [one_hot_lookup[x] for x in x_data]

inputs = torch.Tensor(x_one_hot).view(seq_len, batch_size, input_size)
labels = torch.LongTensor(y_data)
```

```
input_size = 4
hidden_size = 4
num_layers = 1
batch_size = 1
seq_len = 5
```

Shape of *inputs*:

$(seqLen, batchSize, hiddenSize)$

Example 12-2 Change Data

```
idx2char = ['e', 'h', 'l', 'o']
x_data = [1, 0, 2, 2, 3]
y_data = [3, 1, 2, 3, 2]

one_hot_lookup = [[1, 0, 0, 0],
                  [0, 1, 0, 0],
                  [0, 0, 1, 0],
                  [0, 0, 0, 1]]
x_one_hot = [one_hot_lookup[x] for x in x_data]

inputs = torch.Tensor(x_one_hot).view(seq_len, batch_size, input_size)
labels = torch.LongTensor(y_data)
```

```
input_size = 4
hidden_size = 4
num_layers = 1
batch_size = 1
seq_len = 5
```

Shape of *labels*:

$$(seqLen \times batchSize, 1)$$

Example 12-2 Result

```
Predicted:  lhhhh, Epoch [1/15] loss = 1.325
Predicted:  lohhh, Epoch [2/15] loss = 1.190
Predicted:  lolol, Epoch [3/15] loss = 1.090
Predicted:  lolol, Epoch [4/15] loss = 1.017
Predicted:  oolol, Epoch [5/15] loss = 0.958
Predicted:  oolol, Epoch [6/15] loss = 0.904
Predicted:  oolol, Epoch [7/15] loss = 0.852
Predicted:  oolol, Epoch [8/15] loss = 0.804
Predicted:  oolol, Epoch [9/15] loss = 0.759
Predicted:  ohlol, Epoch [10/15] loss = 0.716
Predicted:  ohlol, Epoch [11/15] loss = 0.678
Predicted:  ohlol, Epoch [12/15] loss = 0.643
Predicted:  ohlol, Epoch [13/15] loss = 0.613
Predicted:  ohlol, Epoch [14/15] loss = 0.588
Predicted:  ohlol, Epoch [15/15] loss = 0.568
```

Associate a vector with a word/character

- One-hot encoding of words and characters
  - The one-hot vectors are **high-dimension**. 独热向量维度太高
  - The one-hot vectors are **sparse**. 过于稀疏
  - The one-hot vectors are **hardcoded**. 硬编码

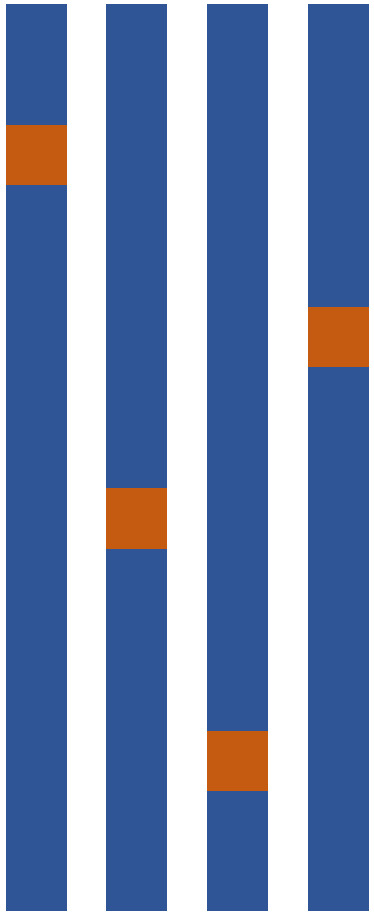
Associate a vector with a word/character

- One-hot encoding of words and characters
  - The one-hot vectors are high-dimension.
  - The one-hot vectors are sparse.
  - The one-hot vectors are hardcoded.
- Do we have a way to associate a vector with a word/character with following specification:
  - **Lower-dimension**
  - **Dense**
  - **Learned from data**
- A popular and powerful way is called EMBEDDING.

Associate a vector with a word/character

- One-hot encoding of words and characters
  - The one-hot vectors are high-dimension.
  - The one-hot vectors are sparse.
  - The one-hot vectors are hardcoded.
- Do we have a way to associate a vector with a word/character with following specification:
  - Lower-dimension
  - Dense
  - Learned from data
- A popular and powerful way is called **EMBEDDING**.

One-hot vs Embedding



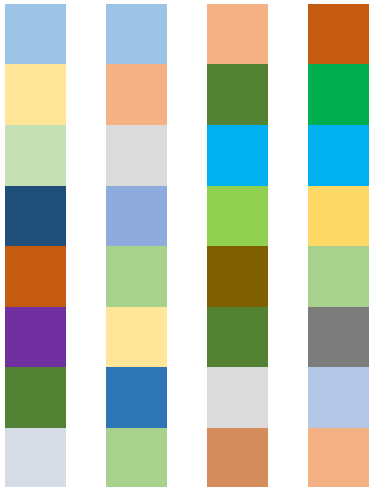
将高维的稀疏数据降维到低维的稠密数据

One-hot vectors:

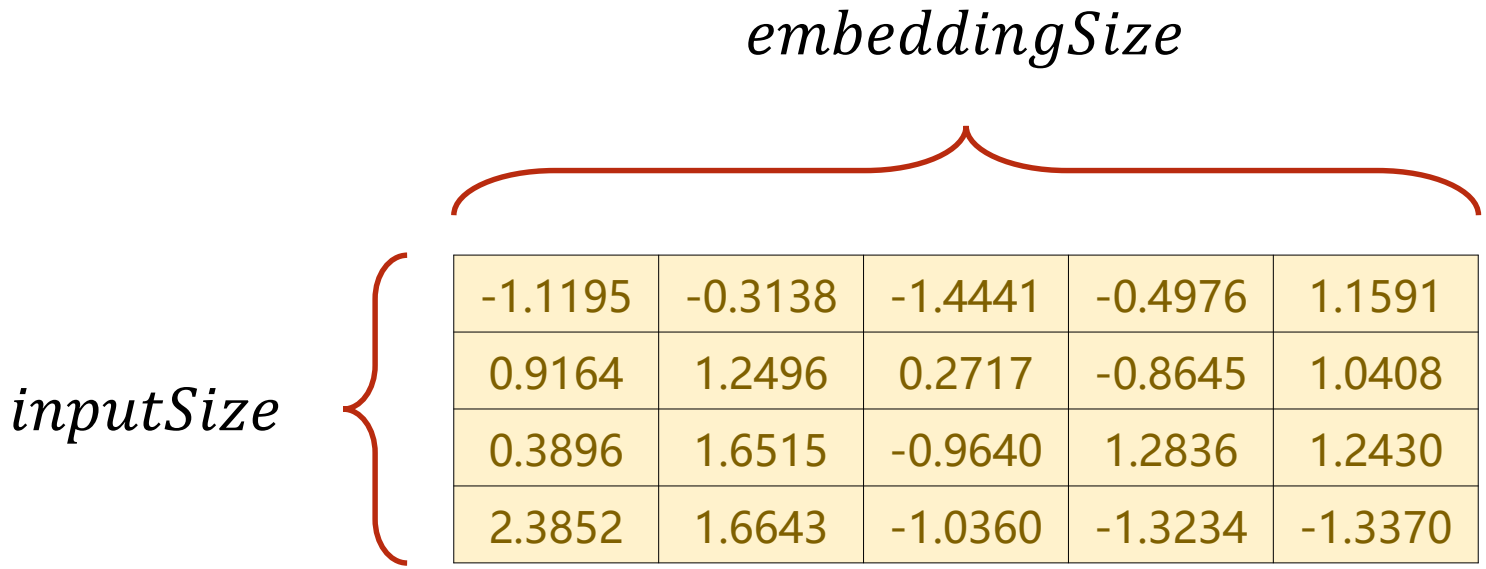
- High-dimension
- Sparse
- Hardcoded

Embedding vectors:

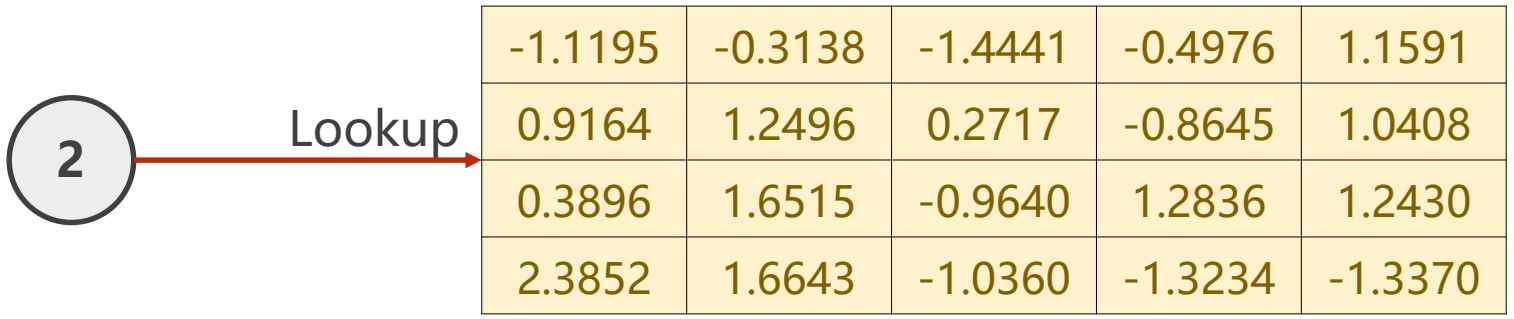
- Lower-dimension
- Dense
- Learned from data



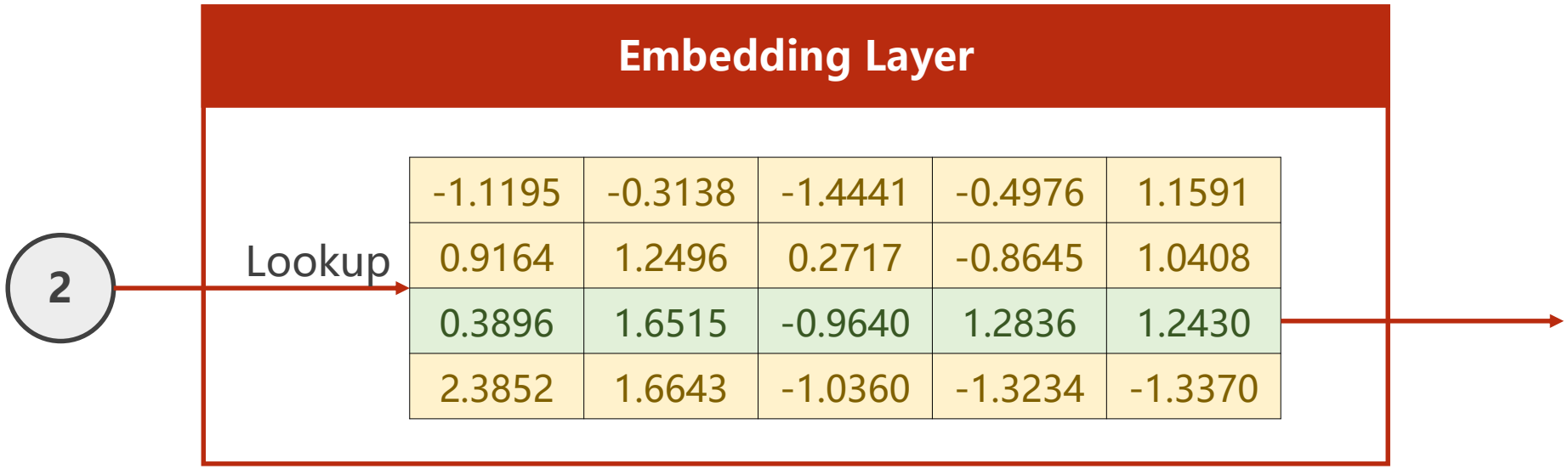
Embedding in PyTorch



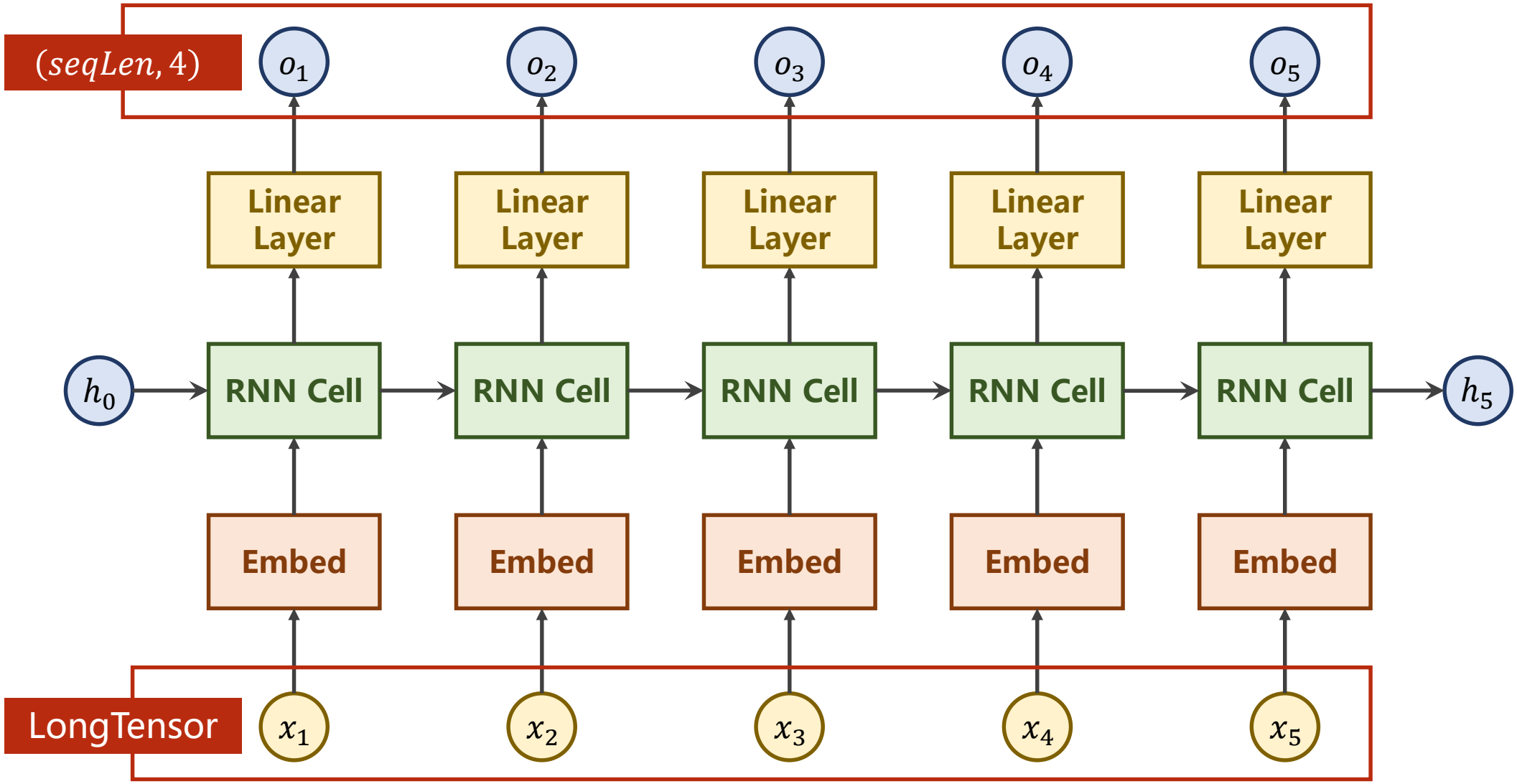
Embedding in PyTorch



Embedding in PyTorch



Example 12-3 Using embedding and linear layer



Example 12-3 Using embedding and linear layer

class **torch.nn.Embedding**(*num_embeddings, embedding_dim, padding_idx=None, max_norm=None, norm_type=2, scale_grad_by_freq=False, sparse=False, _weight=None*) [source]

A simple lookup table that stores embeddings of a fixed dictionary and size.

This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

Parameters:

- **num_embeddings** (*int*) – size of the dictionary of embeddings
- **embedding_dim** (*int*) – the size of each embedding vector
- **padding_idx** (*int, optional*) – If given, pads the output with the embedding vector at

Shape:

- Input: LongTensor of arbitrary shape containing the indices to extract
- Output: *(*, embedding_dim)*, where * is the input shape

- **sparse** (*bool, optional*) – if `True`, gradient w.r.t. weight matrix will be a sparse tensor. See Notes for more details regarding sparse gradients.

Example 12-3 Using embedding and linear layer

class **torch.nn.Linear**(*in_features, out_features, bias=True*) [source]

Applies a linear transformation to the incoming data: $y = Ax + b$

Parameters:

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to False, the layer will not learn an additive bias. Default: `True`

Shape:

- Input: $(N, *, in\_features)$ where $*$ means any number of additional dimensions
- Output: $(N, *, out\_features)$ where all but the last dimension are the same shape as the input.

Example 12-3 Using embedding and linear layer

class `torch.nn.` **`CrossEntropyLoss`** *(weight=None, size_average=True, ignore_index=-100, reduce=True)*
[source]

This criterion combines `nn.LogSoftmax()` and `nn.NLLLoss()` in one single class.

Shape:

- Input: (N, C) where C = *number of classes*, or

  $(N, C, d_1, d_2, \ldots, d_K)$ with $K \geq 2$ in the case of K-dimensional loss.

- Target: (N) where each value is $0 \leq \text{targets}[i] \leq C - 1$, or

  $(N, d_1, d_2, \ldots, d_K)$ with $K \geq 2$ in the case of K-dimensional loss.

- Output: scalar. If reduce is `False`, then the same size

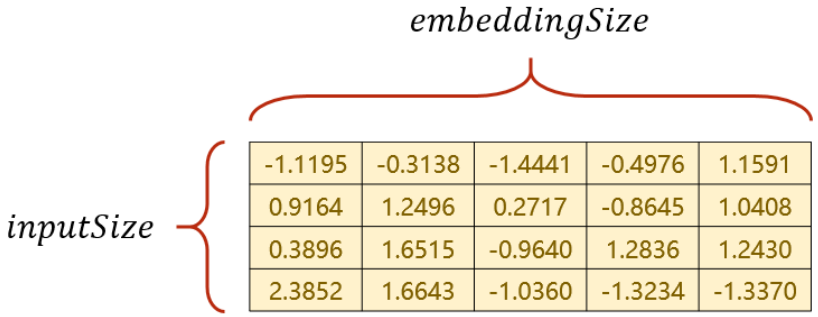
  as the target: (N), or $(N, d_1, d_2, \ldots, d_K)$ with $K \geq 2$ in the case of K-dimensional loss.

Example 12-3 Using embedding and linear layer

```
class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.emb = torch.nn.Embedding(input_size, embedding_size)
        self.rnn = torch.nn.RNN(input_size=embedding_size,
                                hidden_size=hidden_size,
                                num_layers=num_layers,
                                batch_first=True)
        self.fc = torch.nn.Linear(hidden_size, num_class)

    def forward(self, x):
        hidden = torch.zeros(num_layers, x.size(0), hidden_size)
        x = self.emb(x) # (batch, seqLen, embeddingSize)
        x, _ = self.rnn(x, hidden)
        x = self.fc(x)
        return x.view(-1, num_class)
```

Lookup matrix of Embedding:

$(inputSize, embeddingSize)$



Example 12-3 Using embedding and linear layer

```
class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.emb = torch.nn.Embedding(input_size, embedding_size)
        self.rnn = torch.nn.RNN(input_size=embedding_size,
                                hidden_size=hidden_size,
                                num_layers=num_layers,
                                batch_first=True)
        self.fc = torch.nn.Linear(hidden_size, num_class)

    def forward(self, x):
        hidden = torch.zeros(num_layers, x.size(0), hidden_size)
        x = self.emb(x) # (batch, seqLen, embeddingSize)
        x, _ = self.rnn(x, hidden)
        x = self.fc(x)
        return x.view(-1, num_class)
```

Input should be LongTensor:

$(batchSize, seqLen)$

Output with shape:

$(batchSize, seqLen, embeddingSize)$

Notice: batch FIRST

Example 12-3 Using embedding and linear layer

```
class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.emb = torch.nn.Embedding(input_size, embedding_size)
        self.rnn = torch.nn.RNN(input_size=embedding_size,
                                hidden_size=hidden_size,
                                num_layers=num_layers,
                                batch_first=True)
        self.fc = torch.nn.Linear(hidden_size, num_class)

    def forward(self, x):
        hidden = torch.zeros(num_layers, x.size(0), hidden_size)
        x = self.emb(x) # (batch, seqLen, embeddingSize)
        x, _ = self.rnn(x, hidden)
        x = self.fc(x)
        return x.view(-1, num_class)
```

Input of RNN:

$(batchSize, seqLen, embeddingSize)$

Output of RNN:

$(batchSize, seqLen, hiddenSize)$

Example 12-3 Using embedding and linear layer

```
class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.emb = torch.nn.Embedding(input_size, embedding_size)
        self.rnn = torch.nn.RNN(input_size=embedding_size,
                                hidden_size=hidden_size,
                                num_layers=num_layers,
                                batch_first=True)
        self.fc = torch.nn.Linear(hidden_size, num_class)

    def forward(self, x):
        hidden = torch.zeros(num_layers, x.size(0), hidden_size)
        x = self.emb(x) # (batch, seqLen, embeddingSize)
        x, _ = self.rnn(x, hidden)
        x = self.fc(x)
        return x.view(-1, num_class)
```

Shape:

- Input: $(N, *, in\_features)$ where
- Output: $(N, *, out\_features)$ wh

the input.

Input of FC Layer:

$(batchSize, seqLen, hiddenSize)$

Output of FC Layer:

$(batchSize, seqLen, numClass)$

Example 12-3 Using embedding and linear layer

```
class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.emb = torch.nn.Embedding(input_size, embedding_size)
        self.rnn = torch.nn.RNN(input_size=embedding_size,
                                hidden_size=hidden_size,
                                num_layers=num_layers,
                                batch_first=True)
        self.fc = torch.nn.Linear(hidden_size, num_class)

    def forward(self, x):
        hidden = torch.zeros(num_layers, x.size(0), hidden_size)
        x = self.emb(x) # (batch, seqLen, embeddingSize)
        x, _ = self.rnn(x, hidden)
        x = self.fc(x)
        return x.view(-1, num_class)
```

Shape:

- Input: (N, C) wh

  $(N, C, d_1, d_2$

- Target: (N) wher

  $(N, d_1, d_2, .$

Reshape result to use Cross Entropy Loss:

$(batchSize \times seqLen, numClass)$

Example 12-3 Using embedding and linear layer

```
# parameters
num_class = 4
input_size = 4
hidden_size = 8
embedding_size = 10
num_layers = 2
batch_size = 1
seq_len = 5
```

Example 12-3 Using embedding and linear layer

```
idx2char = ['e', 'h', 'l', 'o']
x_data = [[1, 0, 2, 2, 3]] # (batch, seq_len)
y_data = [3, 1, 2, 3, 2] # (batch * seq_len)

inputs = torch.LongTensor(x_data)
labels = torch.LongTensor(y_data)
```

Input should be LongTensor:

$(batchSize, seqLen)$

Target should be LongTensor:

$(batchSize \times seqLen)$

```
# parameters
num_class = 4
input_size = 4
hidden_size = 8
embedding_size = 10
num_layers = 2
batch_size = 1
seq_len = 5
```

Example 12-3 Using embedding and linear layer

```
net = Model()

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=0.05)
```

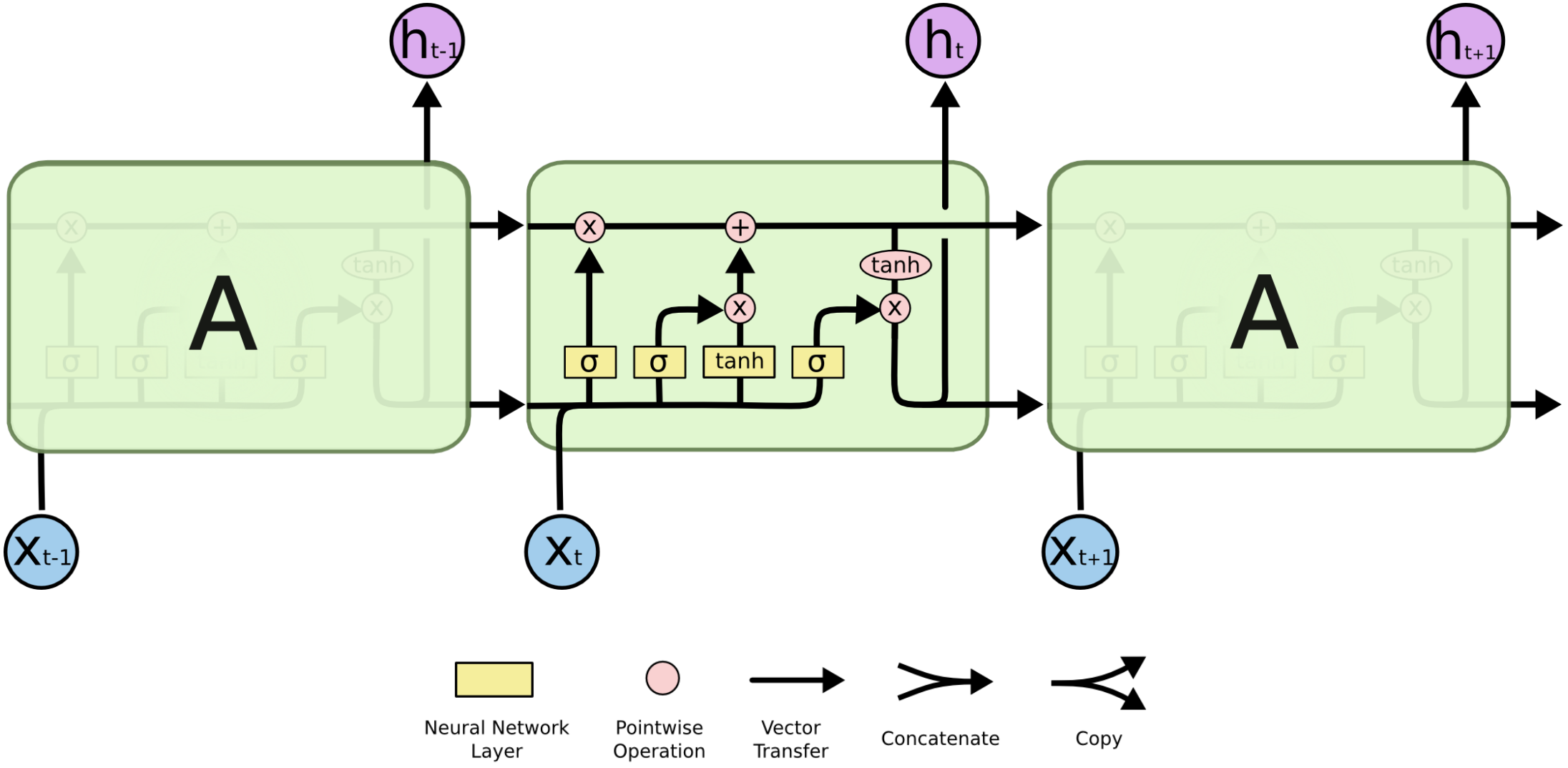
Example 12-3 Using embedding and linear layer

```
for epoch in range(15):
    optimizer.zero_grad()
    outputs = net(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

    _, idx = outputs.max(dim=1)
    idx = idx.data.numpy()
    print('Predicted: ', ''.join([idx2char[x] for x in idx]), end='')
    print(', Epoch [%d/15] loss = %.3f' % (epoch + 1, loss.item()))
```

Example 12-3 Using embedding and linear layer

```
Predicted:  lllll, Epoch [1/15] loss = 1.412
Predicted:  lllll, Epoch [2/15] loss = 1.038
Predicted:  lhlll, Epoch [3/15] loss = 0.799
Predicted:  ohlol, Epoch [4/15] loss = 0.652
Predicted:  ohloo, Epoch [5/15] loss = 0.548
Predicted:  ohlol, Epoch [6/15] loss = 0.459
Predicted:  ohlol, Epoch [7/15] loss = 0.381
Predicted:  ohlol, Epoch [8/15] loss = 0.313
Predicted:  ohlol, Epoch [9/15] loss = 0.248
Predicted:  ohlol, Epoch [10/15] loss = 0.188
Predicted:  ohlol, Epoch [11/15] loss = 0.136
Predicted:  ohlol, Epoch [12/15] loss = 0.098
Predicted:  ohlol, Epoch [13/15] loss = 0.072
Predicted:  ohlol, Epoch [14/15] loss = 0.055
Predicted:  ohlol, Epoch [15/15] loss = 0.043
```

Exercise 12 – 1 Using LSTM



Exercise 12 – 1 Using LSTM

class **torch.nn.LSTM**(**args*, ***kwargs*) [source]

Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

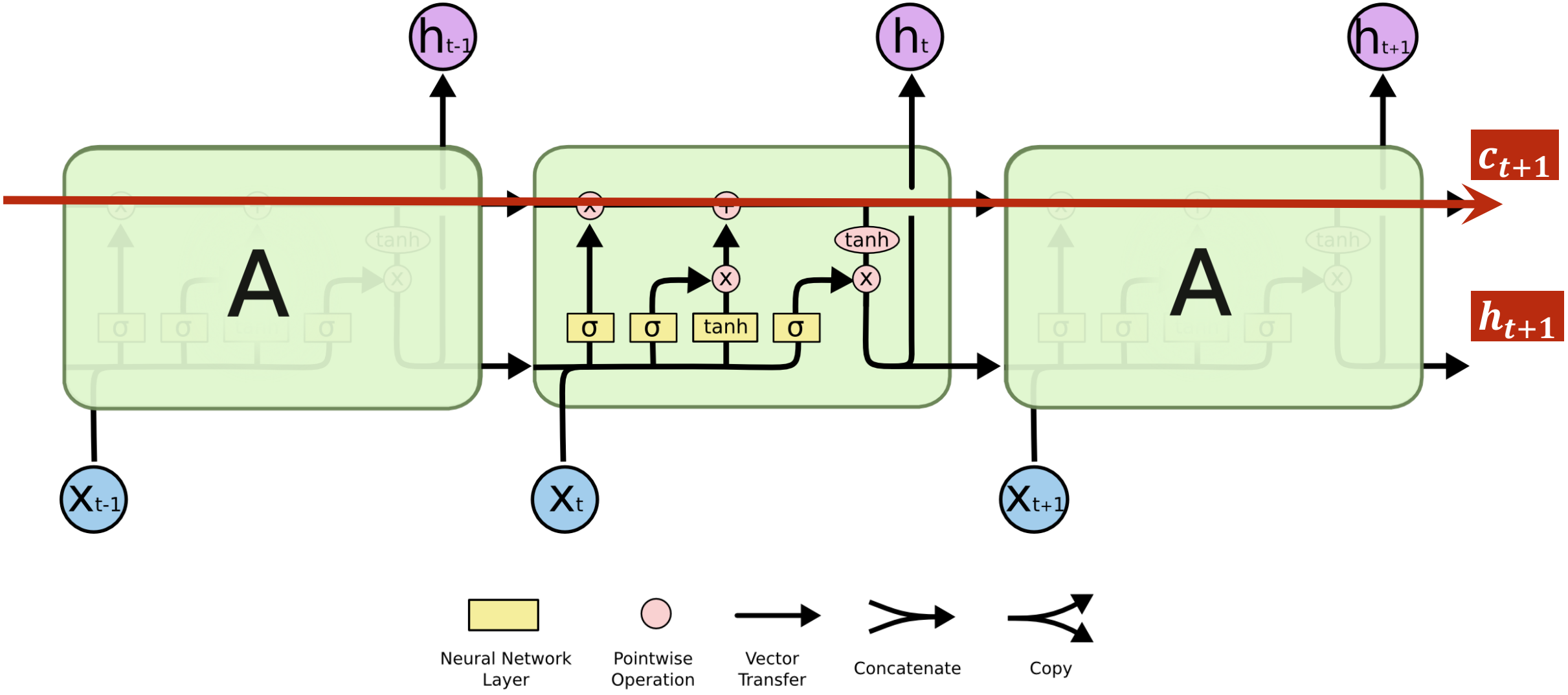
For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \\
c_t &= f_t c_{(t-1)} + i_t g_t \\
h_t &= o_t \tanh(c_t)
\end{aligned}
$$

where h_t is the hidden state at time *t*, c_t is the cell state at time *t*, x_t is the input at time *t*, $h_{(t-1)}$ is the hidden state of the previous layer at time *t-1* or the initial hidden state at time *0*, and i_t, f_t, g_t, o_t are the input, forget, cell, and output gates, respectively. σ is the sigmoid function.

https://pytorch.org/docs/stable/nn.html#lstm

Exercise 12 – 1 Using LSTM



Exercise 12 – 1 Using LSTM

Inputs: input, (h_0, c_0)

- **input** of shape *(seq_len, batch, input_size)*: tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- **h_0** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the initial hidden state for each element in the batch.
- **c_0** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the initial cell state for each element in the batch.

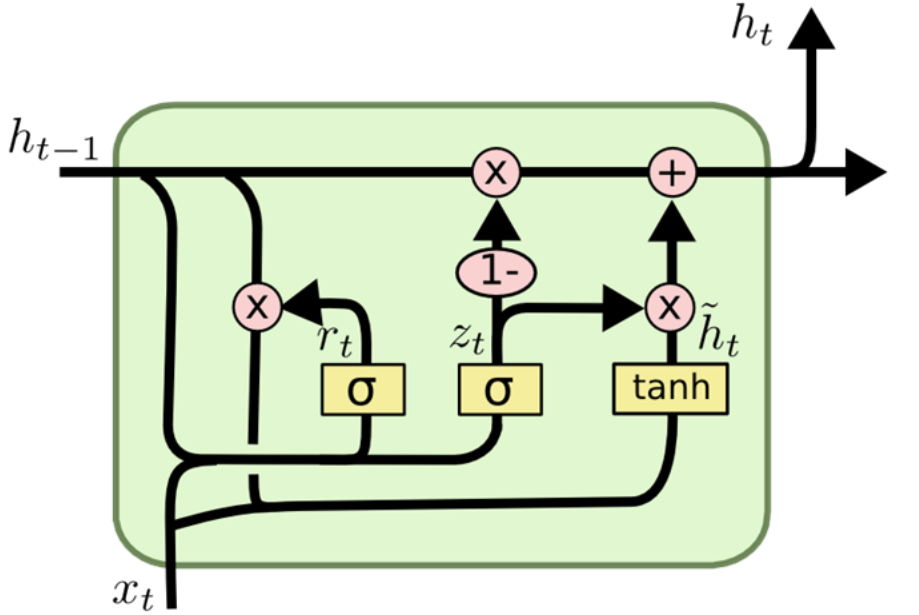
  If *(h_0, c_0)* is not provided, both **h_0** and **c_0** default to zero.

Outputs: output, (h_n, c_n)

- **output** of shape *(seq_len, batch, hidden_size * num_directions)*: tensor containing the output features *(h_t)* from the last layer of the LSTM, for each t. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence.
- **h_n** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the hidden state for *t = seq_len*
- **c_n** (num_layers * num_directions, batch, hidden_size): tensor containing the cell state for *t = seq_len*

https://pytorch.org/docs/stable/nn.html#lstm

Exercise 12 – 2 Using GRU

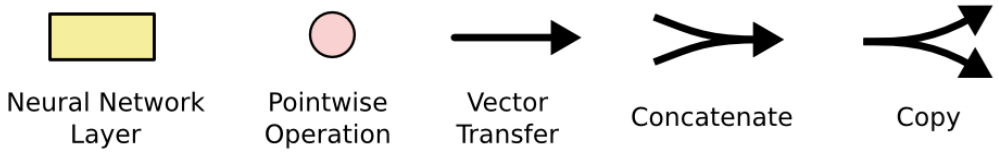


$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$



Exercise 12 – 2 Using GRU

class **torch.nn.GRU**(**args*, ***kwargs*) [source]

Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \\
z_t &= \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \\
n_t &= \tanh(W_{in}x_t + b_{in} + r_t(W_{hn}h_{(t-1)} + b_{hn})) \\
h_t &= (1 - z_t)n_t + z_t h_{(t-1)}
\end{aligned}
$$

where h_t is the hidden state at time *t*, x_t is the input at time *t*, $h_{(t-1)}$ is the hidden state of the previous layer at time *t-1* or the initial hidden state at time 0, and r_t, z_t, n_t are the reset, update, and new gates, respectively. σ is the sigmoid function.

https://pytorch.org/docs/stable/nn.html#gru

Exercise 12 – 2 Using GRU

Inputs: input, h_0

- **input** of shape *(seq_len, batch, input_size)*: tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` for details.
- **h_0** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided.

Outputs: output, h_n

- **output** of shape *(seq_len, batch, hidden_size * num_directions)*: tensor containing the output features h_t from the last layer of the GRU, for each t. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence.
- **h_n** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the hidden state for *t = seq_len*

https://pytorch.org/docs/stable/nn.html#gru

PyTorch Tutorial

12. Basic RNN