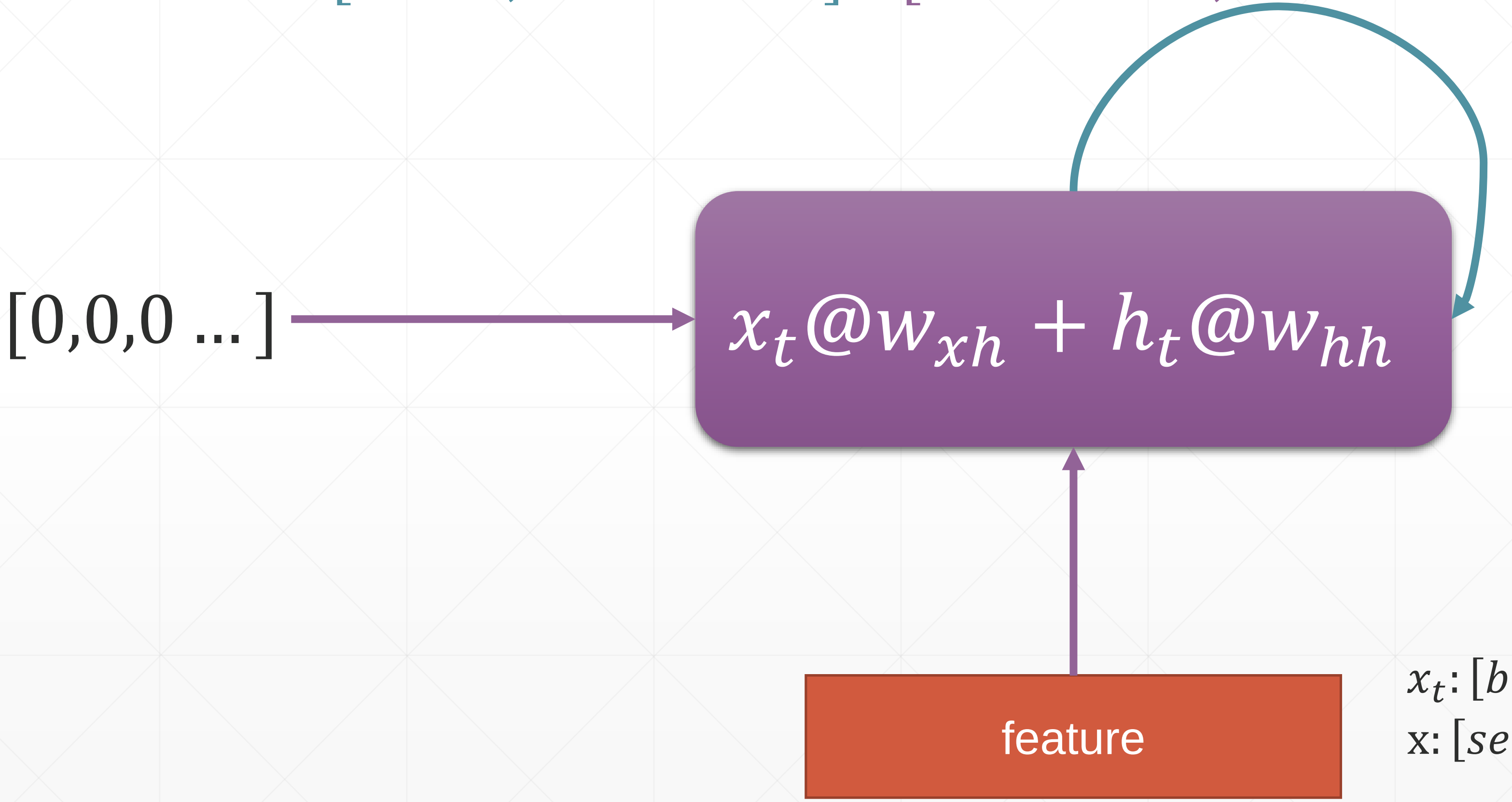
PyTorch

RNN Layer使用

主讲人：龙良曲

Folded model

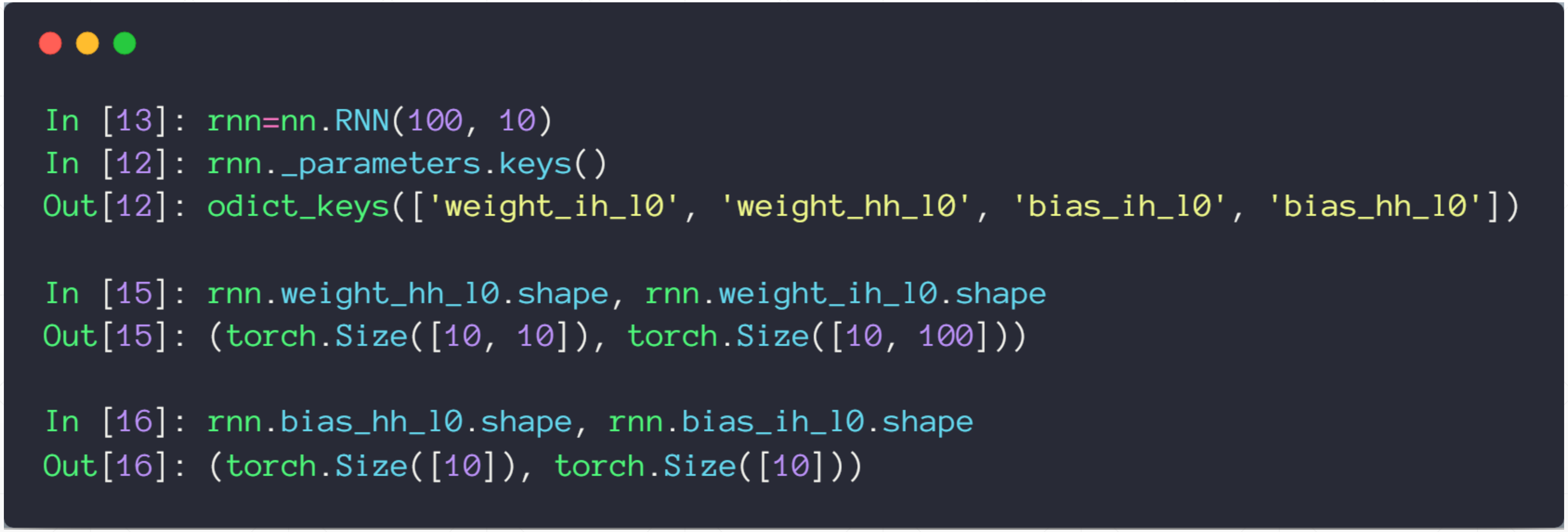
$$[batch, feature\ len]@[hidden\ len, feature\ len]^T + [batch, hidden\ len]@[hidden\ len, hidden\ len]^T$$



$x_t: [batch, feature\ len]$

$x: [seq\ len, batch, feature\ len]$

input dim, hidden dim



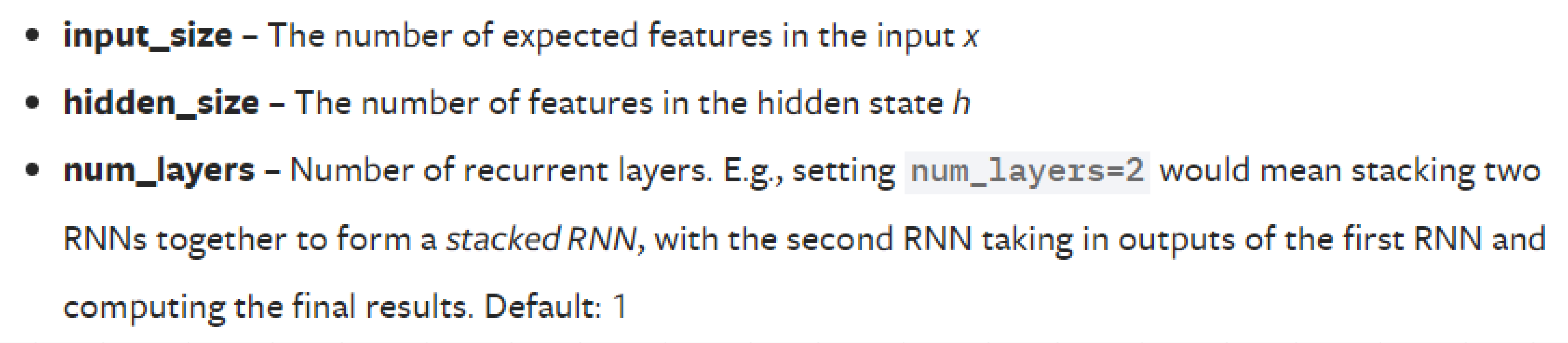
```
In [13]: rnn=nn.RNN(100, 10)
In [12]: rnn._parameters.keys()
Out[12]: odict_keys(['weight_ih_l0', 'weight_hh_l0', 'bias_ih_l0', 'bias_hh_l0'])

In [15]: rnn.weight_hh_l0.shape, rnn.weight_ih_l0.shape
Out[15]: (torch.Size([10, 10]), torch.Size([10, 100]))

In [16]: rnn.bias_hh_l0.shape, rnn.bias_ih_l0.shape
Out[16]: (torch.Size([10]), torch.Size([10]))
```

nn.RNN

- __init__

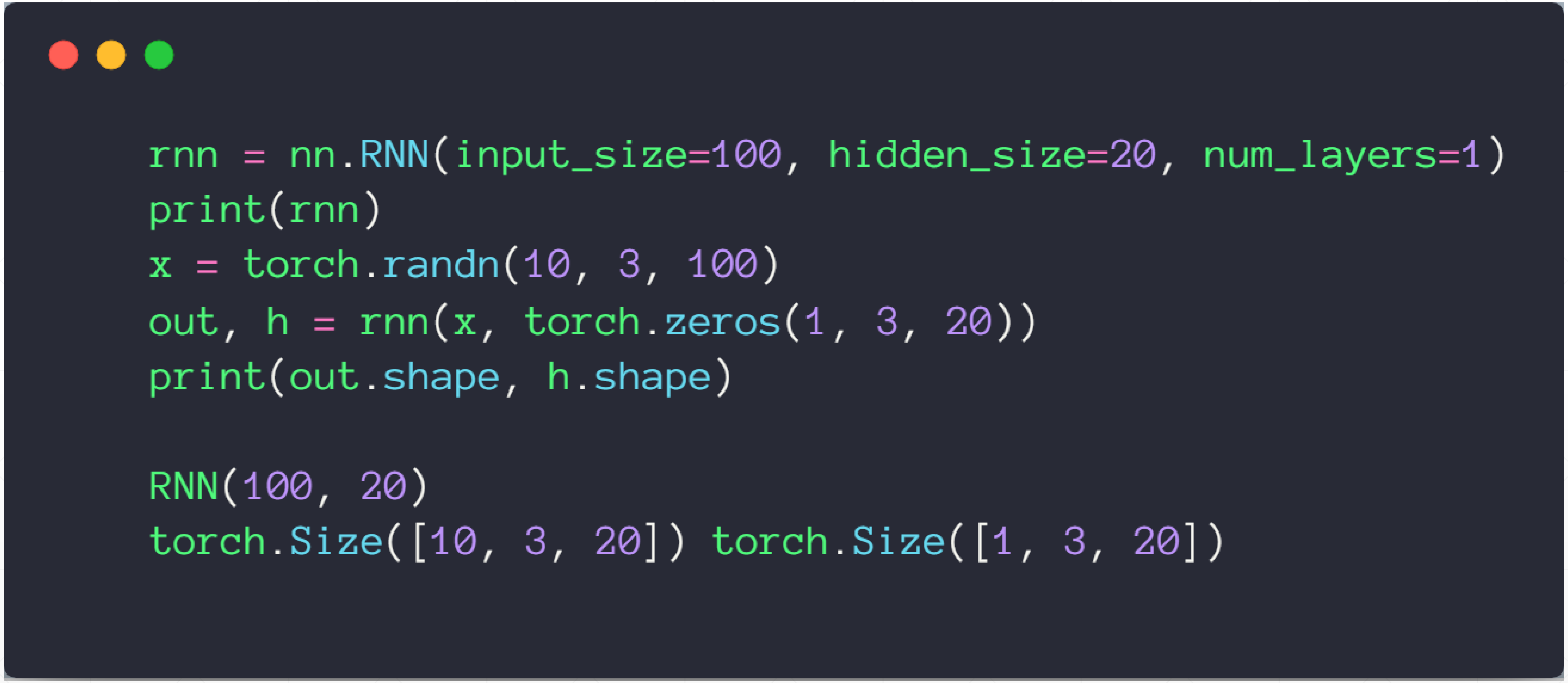


- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1

nn.RNN

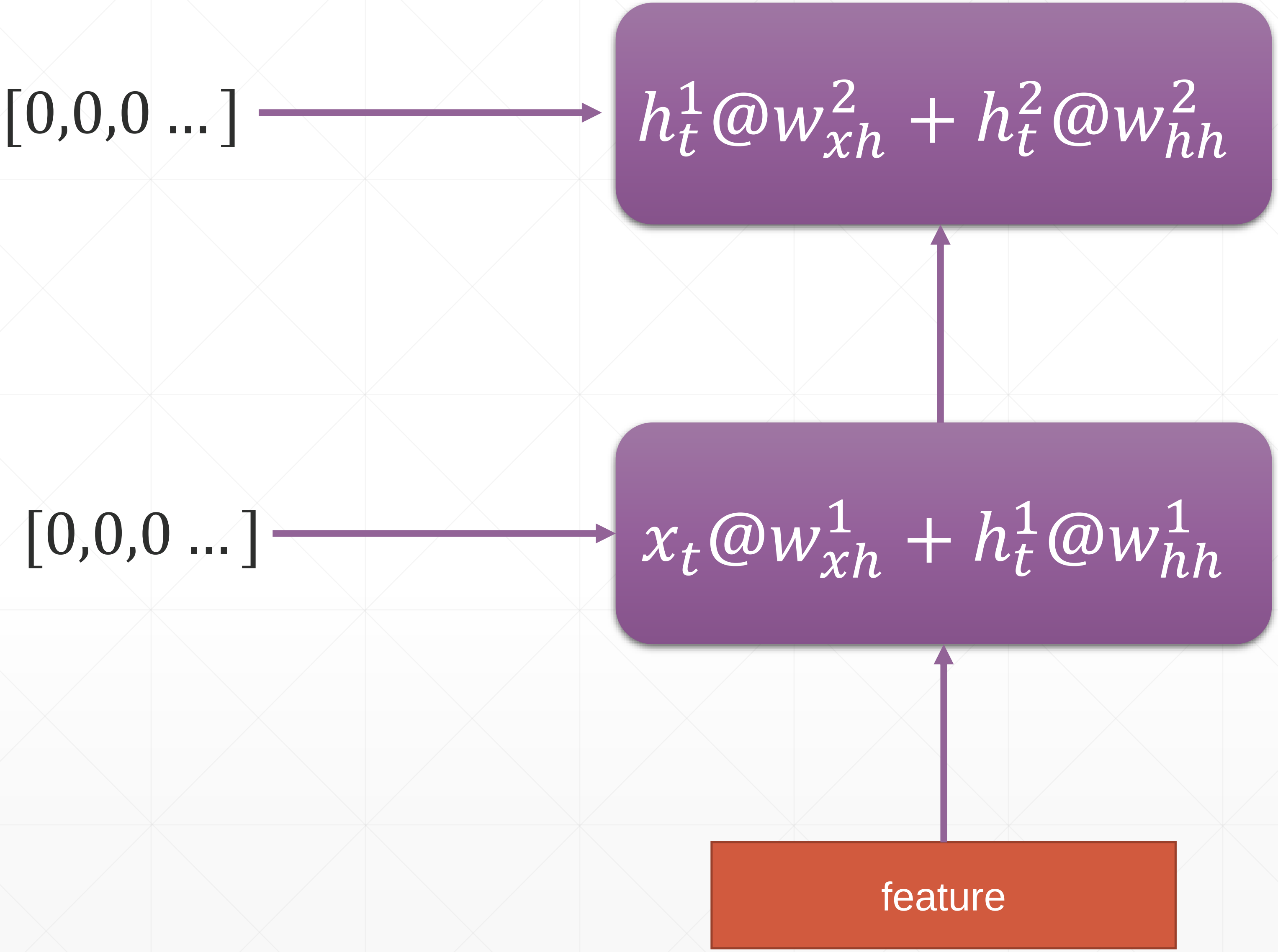
- out, ht = forward(x, h0)
  - x: [seq len, b, word vec]
  - h0/ht: [num layers, b, h dim]
  - out: [seq len, b, h dim]

Single layer RNN



```
rnn = nn.RNN(input_size=100, hidden_size=20, num_layers=1)
print(rnn)
x = torch.randn(10, 3, 100)
out, h = rnn(x, torch.zeros(1, 3, 20))
print(out.shape, h.shape)

RNN(100, 20)
torch.Size([10, 3, 20]) torch.Size([1, 3, 20])
```



[0,0,0 ...]

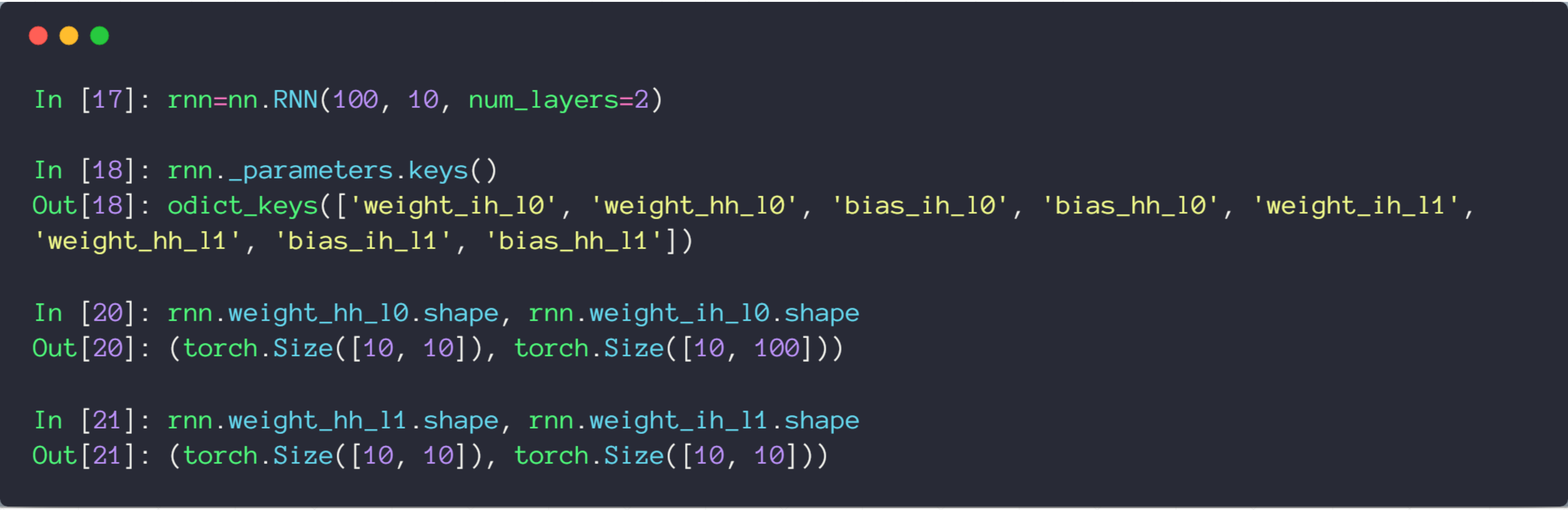
$h_t^1@w_{xh}^2 + h_t^2@w_{hh}^2$

[0,0,0 ...]

$x_t@w_{xh}^1 + h_t^1@w_{hh}^1$

feature

2 layer RNN



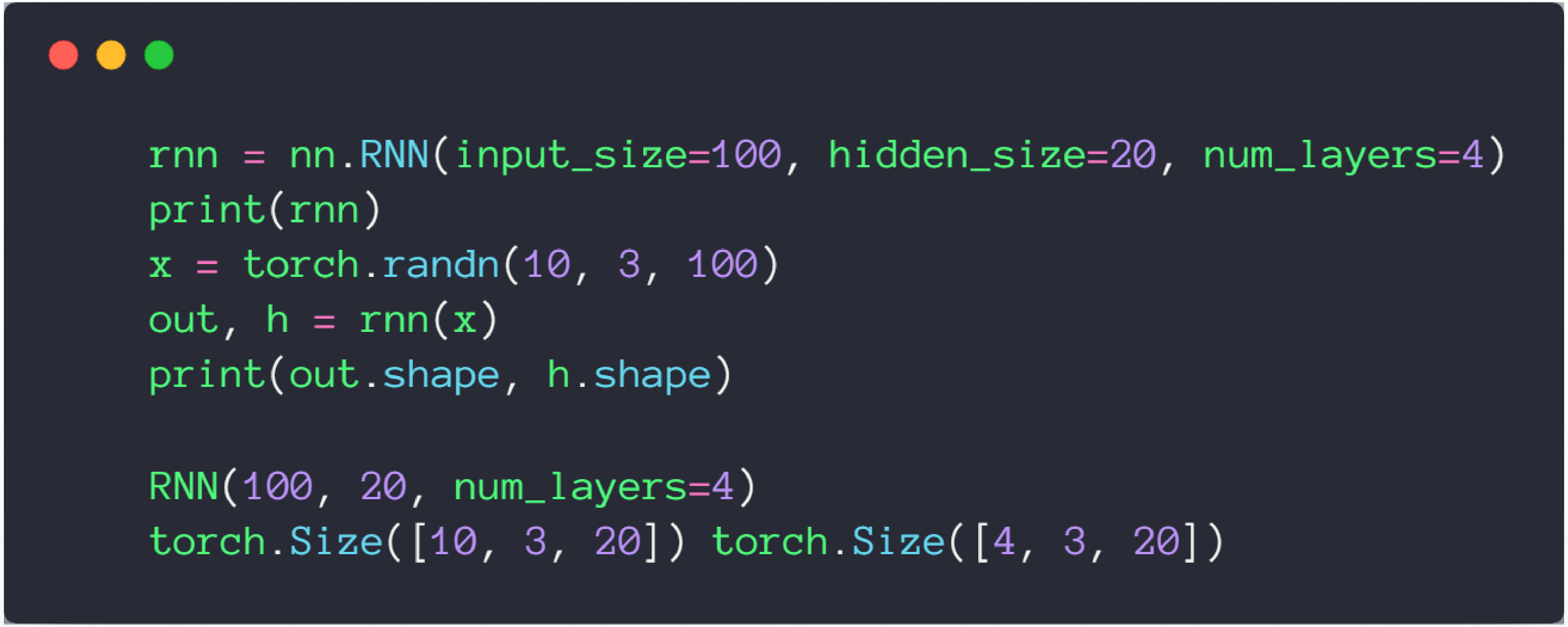
```
In [17]: rnn=nn.RNN(100, 10, num_layers=2)

In [18]: rnn._parameters.keys()
Out[18]: odict_keys(['weight_ih_l0', 'weight_hh_l0', 'bias_ih_l0', 'bias_hh_l0', 'weight_ih_l1',
'weight_hh_l1', 'bias_ih_l1', 'bias_hh_l1'])

In [20]: rnn.weight_hh_l0.shape, rnn.weight_ih_l0.shape
Out[20]: (torch.Size([10, 10]), torch.Size([10, 100]))

In [21]: rnn.weight_hh_l1.shape, rnn.weight_ih_l1.shape
Out[21]: (torch.Size([10, 10]), torch.Size([10, 10]))
```

[T, b, h_dim], [layers, b, h_dim]

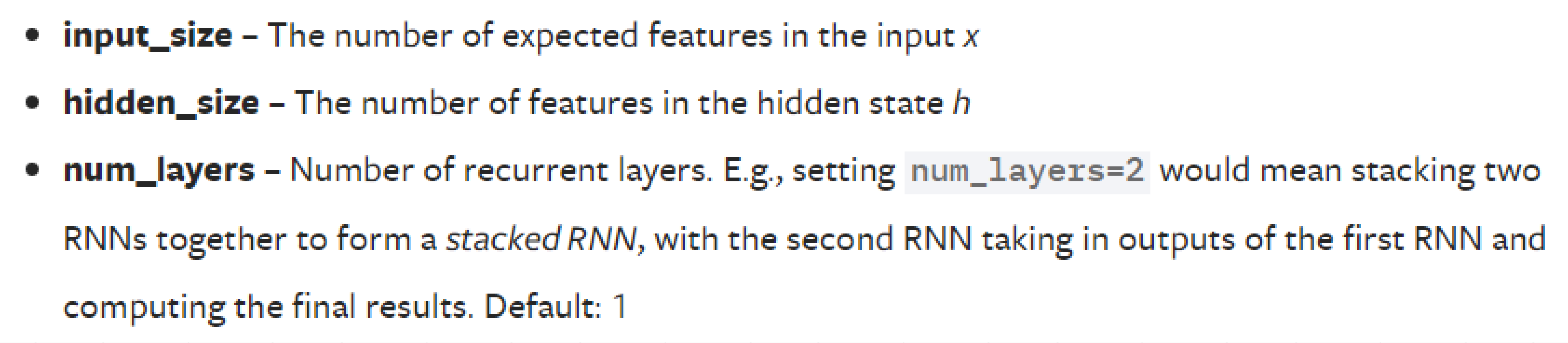


```
rnn = nn.RNN(input_size=100, hidden_size=20, num_layers=4)
print(rnn)
x = torch.randn(10, 3, 100)
out, h = rnn(x)
print(out.shape, h.shape)

RNN(100, 20, num_layers=4)
torch.Size([10, 3, 20]) torch.Size([4, 3, 20])
```

nn.RNNCell

- __init__

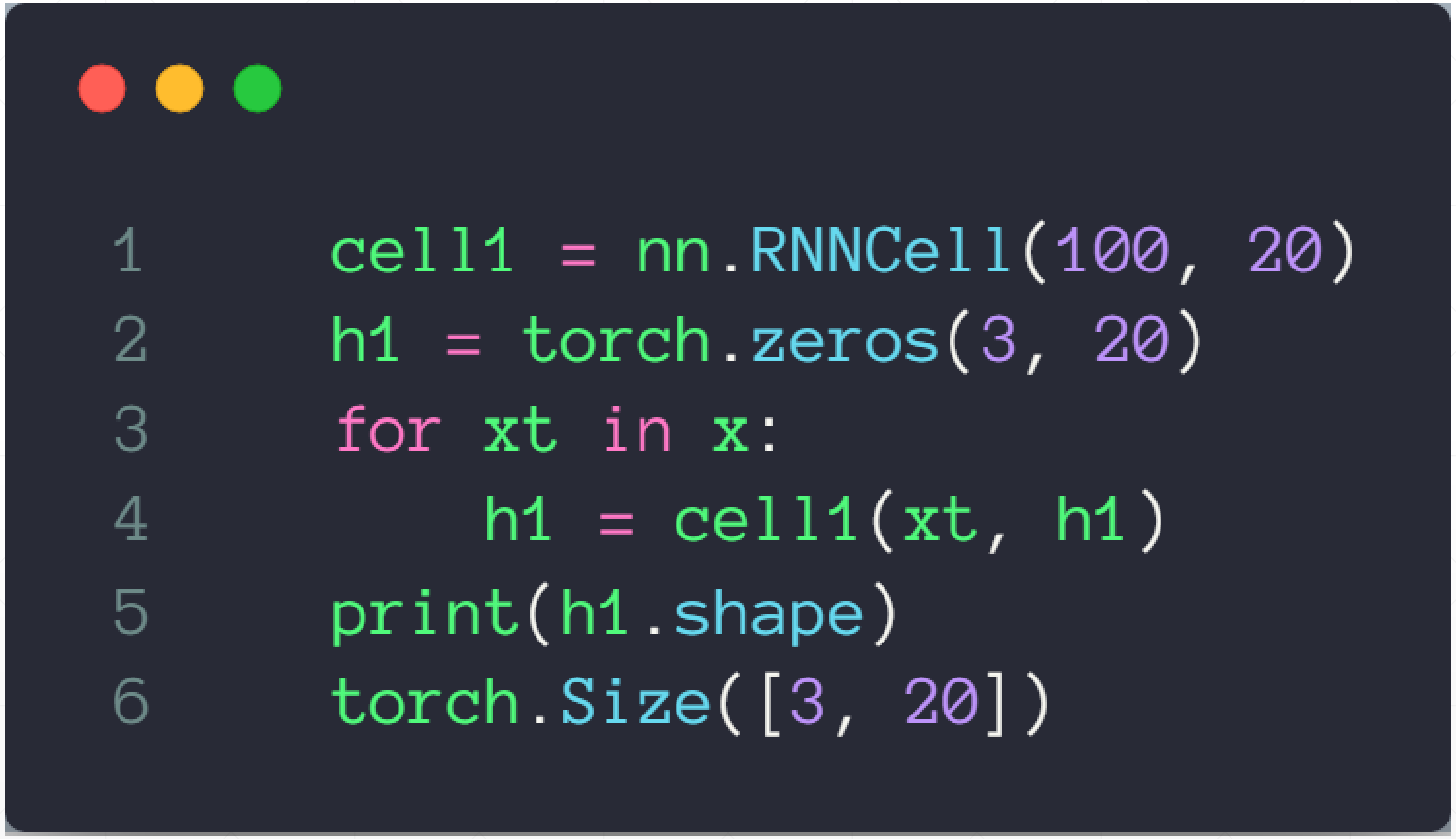


- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1

nn.RNNCell

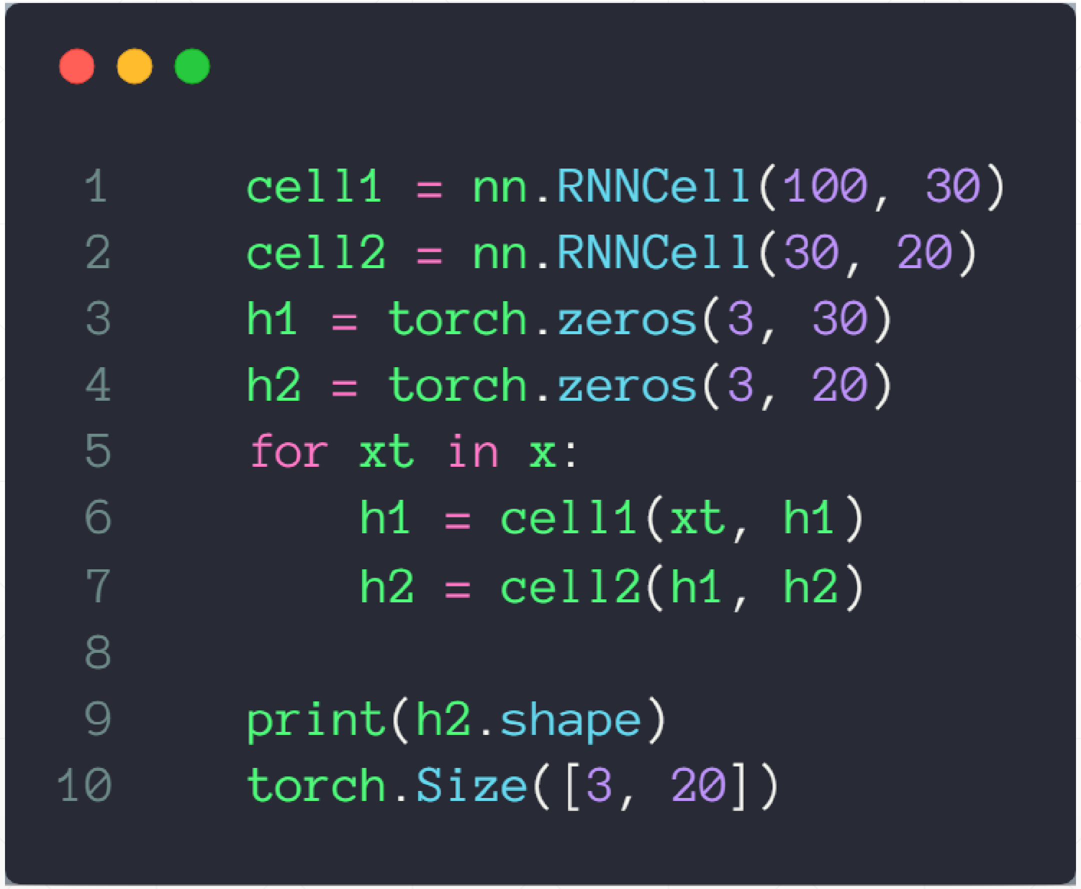
- ht = rnncell(xt, ht_1)
  - xt: [b, word vec]
  - ht_1/ht: [num layers, b, h dim]
  - out = torch.stack([h1, h2,…, ht])

Functional



```
cell1 = nn.RNNCell(100, 20)
h1 = torch.zeros(3, 20)
for xt in x:
    h1 = cell1(xt, h1)
print(h1.shape)
torch.Size([3, 20])
```

Functional



```
cell1 = nn.RNNCell(100, 30)
cell2 = nn.RNNCell(30, 20)
h1 = torch.zeros(3, 30)
h2 = torch.zeros(3, 20)
for xt in x:
    h1 = cell1(xt, h1)
    h2 = cell2(h1, h2)

print(h2.shape)
torch.Size([3, 20])
```

下一课时

时间序列预测

Thank You.