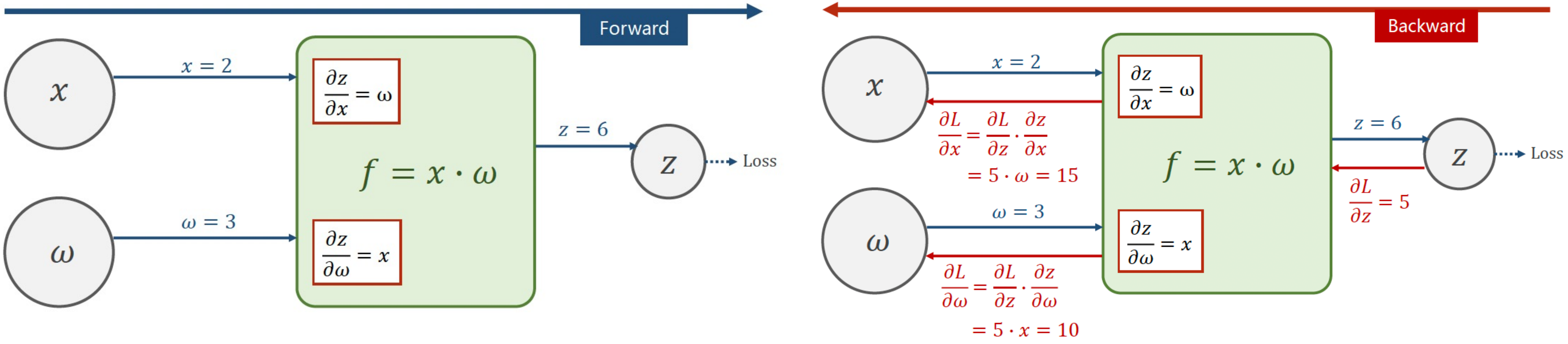
PyTorch Tutorial

朱书琦　zhusq22@mails.tsinghua.edu.cn

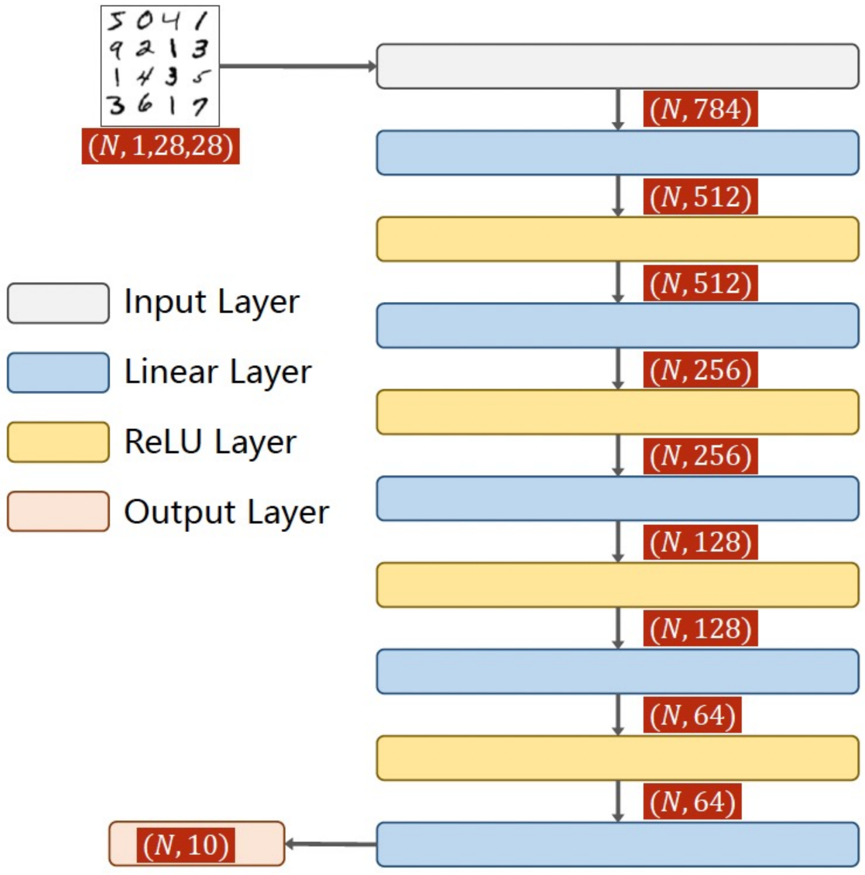
Back Propagation



MLP

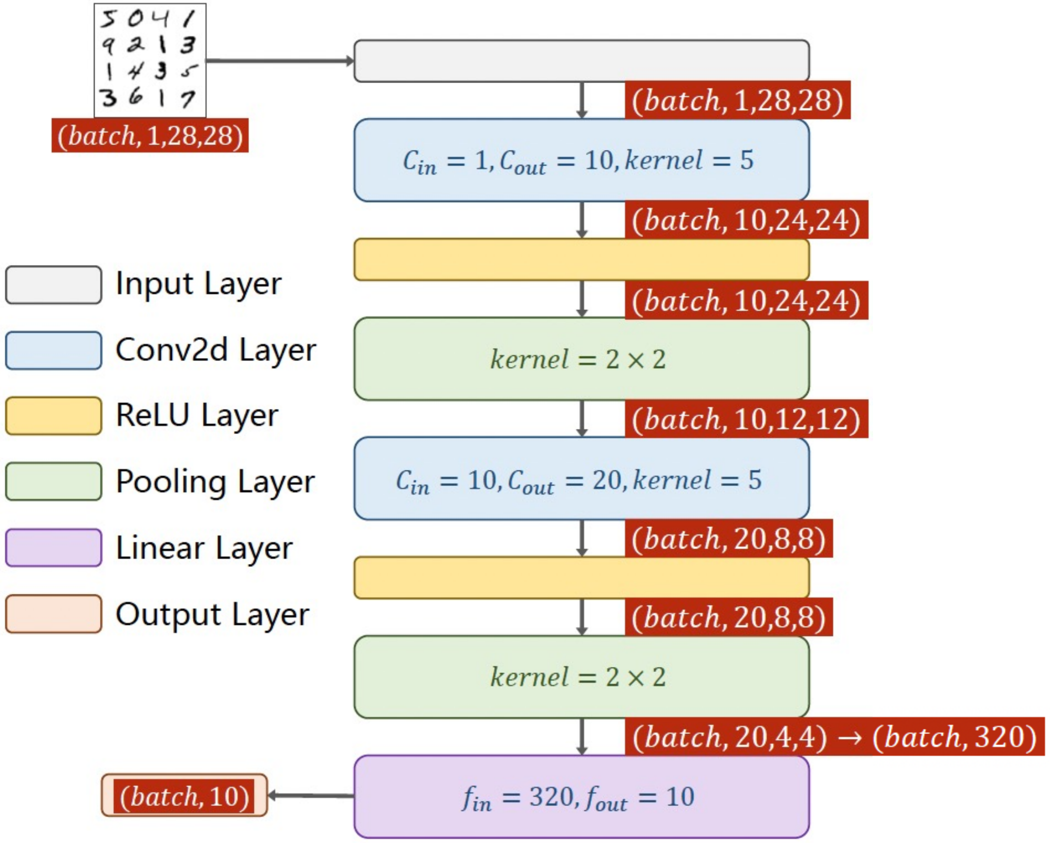
- Construct MLP model
  - choose hyperparameters at will
  - output dimensions(10) for softmax

```
import torch.nn.functional as F
class MLP(torch.nn.Module):
    def __self__(self):
        super(MLP, self).__init__()
        self.l1 = torch.nn.Linear( in_features: 784,  out_features: 512)
        self.l2 = torch.nn.Linear( in_features: 512,  out_features: 256)
        self.l3 = torch.nn.Linear( in_features: 256,  out_features: 128)
        self.l4 = torch.nn.Linear( in_features: 128,  out_features: 64)
        self.l5 = torch.nn.Linear( in_features: 64,  out_features: 10)

    def forward(self, x):
        x = x.view(-1, 784)
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = F.relu(self.l3(x))
        x = F.relu(self.l4(x))
        return self.l5(x)
```



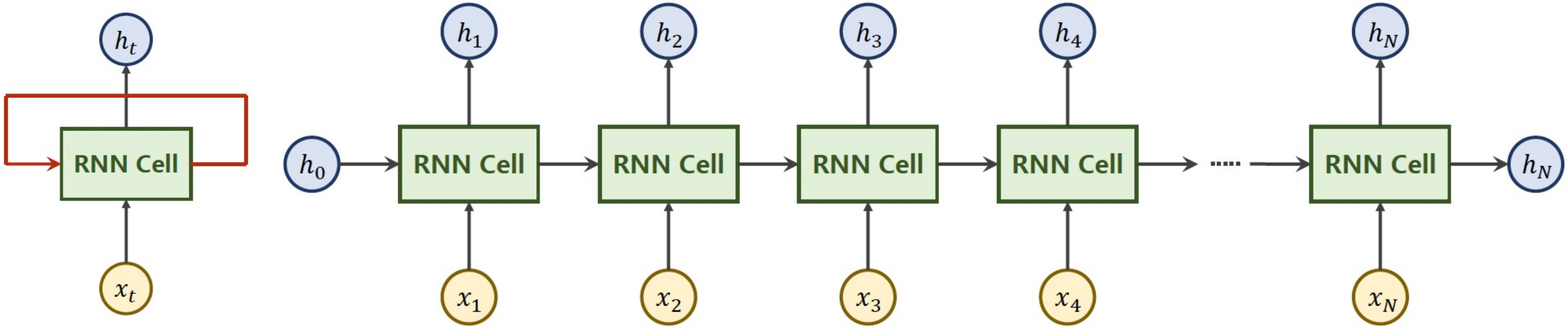
CNN

```python
class CNN(torch.nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = torch.nn.Conv2d( in_channels: 1,  out_channels: 10, kernel_size=5)
        self.conv2 = torch.nn.Conv2d( in_channels: 10,  out_channels: 20, kernel_size=5)
        self.pooling = torch.nn.MaxPool2d(2)
        self.fc = torch.nn.Linear( in_features: 320,  out_features: 10)

    def forward(self, x):
        # Flatten data from (batch_size, 1, 28, 28) to (batch_size, 784)
        batch_size = x.size(0)
        x = F.relu(self.pooling(self.conv1(x)))
        x = F.relu(self.pooling(self.conv2(x)))
        x = x.view(batch_size, -1)  # flatten
        x = self.fc(x)
        return x
```

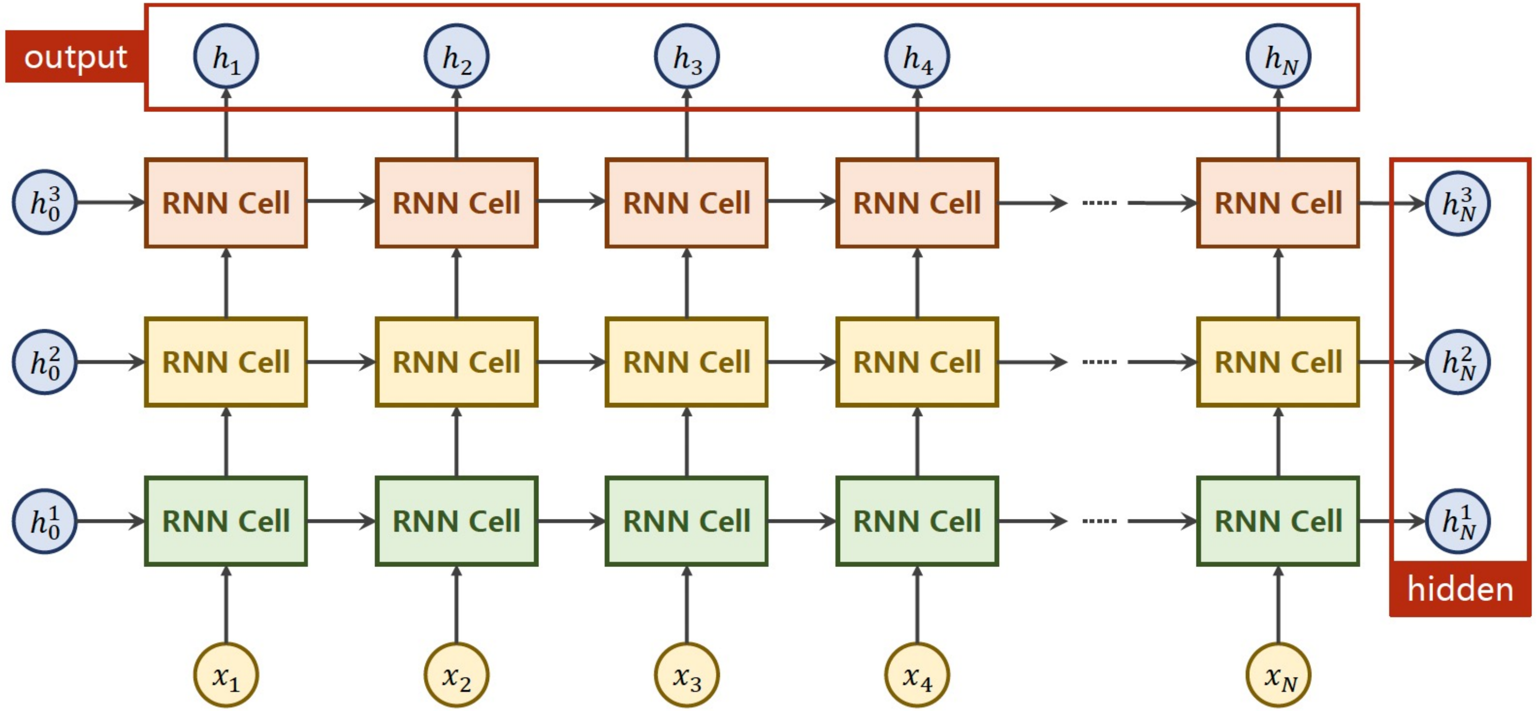


RNN

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



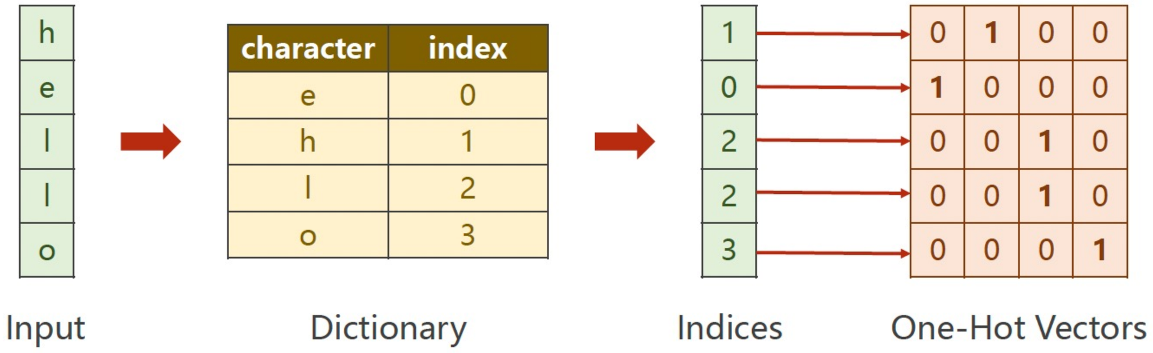
RNN

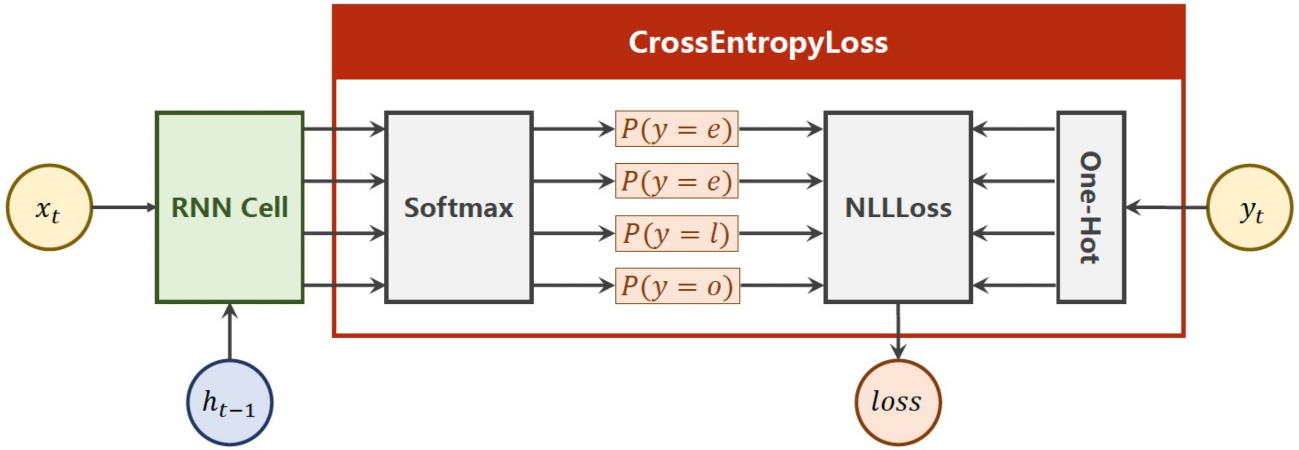


- $input.shape = (seqLen, batchSize, inputSize)$
- $h\_0.shape = (numLayers, batchSize, hiddenSize)$
- $output.shape = (seqLen, batchSize, hiddenSize)$
- $h\_n.shape = (numLayers, batchSize, hiddenSize)$

RNN

- **Task:** Train a model to transfer “hello” to “ohlol”.
- How to present a character?





RNN

```python
input_size = 4
hidden_size = 4
batch_size = 1
num_layers = 1
idx2char = ['e', 'h', 'l', 'o']
x_data = [1, 0, 2, 2, 3]  # hello
y_data = [3, 1, 2, 3, 2]  # ohlol
one_hot_lookup = [[1, 0, 0, 0],
                  [0, 1, 0, 0],
                  [0, 0, 1, 0],
                  [0, 0, 0, 1]]
```

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

RNN

```python
if __name__ == '__main__':
    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)

    net = RNN(input_size, hidden_size, batch_size, num_layers)
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=0.05)

    for epoch in range(100):
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        _, idx = outputs.max(dim=1)
        print(f"Epoch {epoch}/99, Loss {loss.item():.4f}: {''.join([idx2char[x] for x in idx])}")
```

```
Epoch 93/15, Loss 0.3479: ohlol
Epoch 94/15, Loss 0.3478: ohlol
Epoch 95/15, Loss 0.3477: ohlol
Epoch 96/15, Loss 0.3476: ohlol
Epoch 97/15, Loss 0.3475: ohlol
Epoch 98/15, Loss 0.3474: ohlol
Epoch 99/15, Loss 0.3473: ohlol
```

RNN

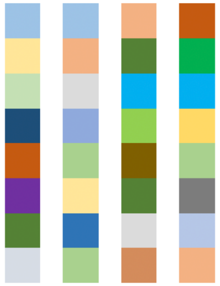
- How to present a character?
- word2vec, torch.nn.Embedding, BERT, …

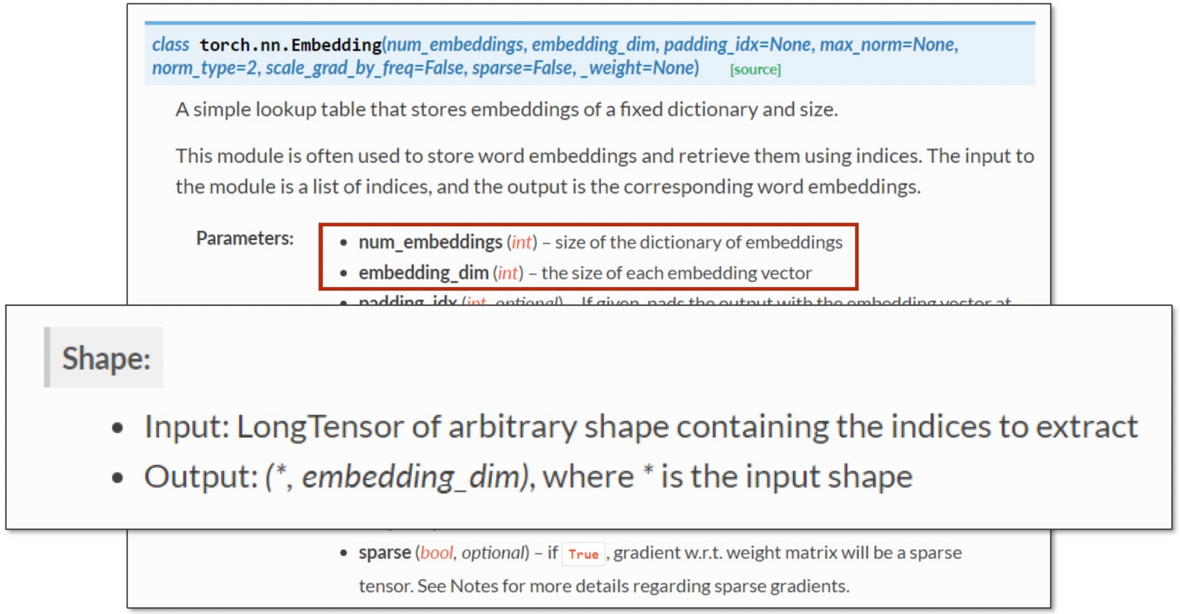
One-hot vectors:

- High-dimension
- Sparse
- Hardcoded

Embedding vectors:

- Lower-dimension
- Dense
- Learned from data





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.

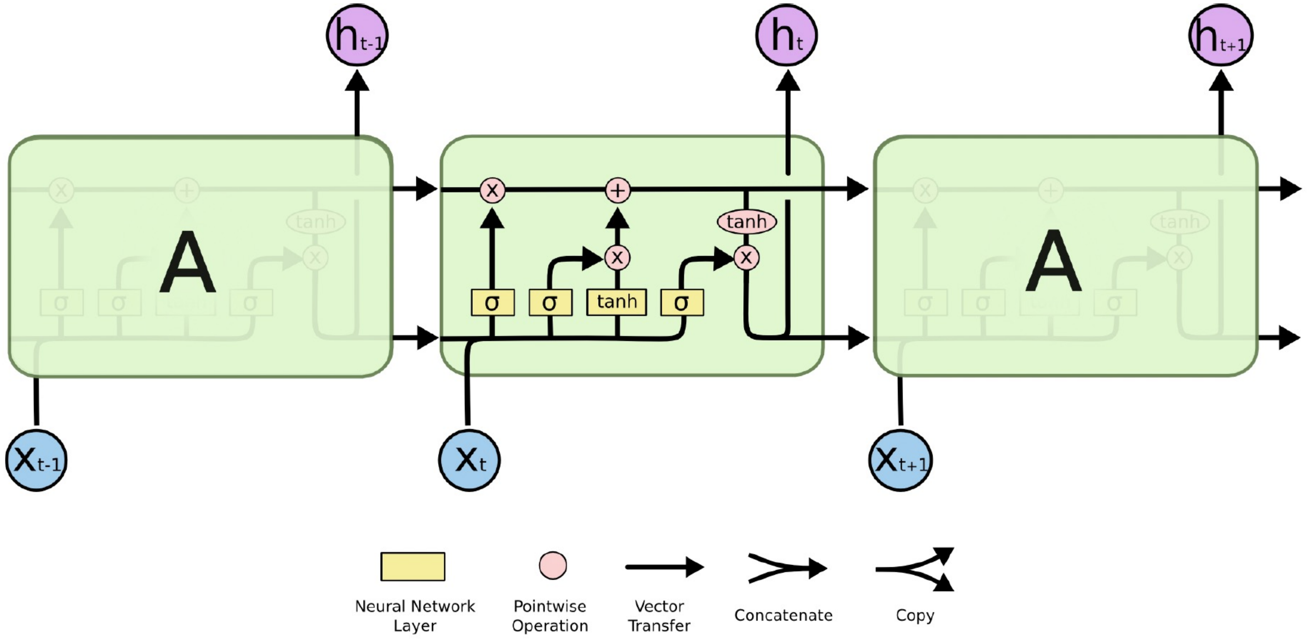
This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

Parameters:
- **num_embeddings** (*int*) – size of the dictionary of embeddings
- **embedding_dim** (*int*) – the size of each embedding vector
- **sparse** (*bool, optional*) – if `True`, gradient w.r.t. weight matrix will be a sparse tensor. See Notes for more details regarding sparse gradients.

Shape:

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

LSTM



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

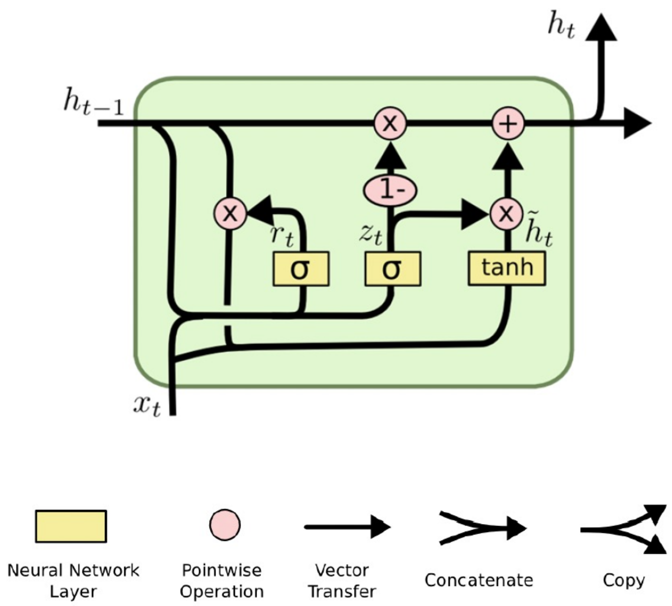
Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

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.

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.

Thanks