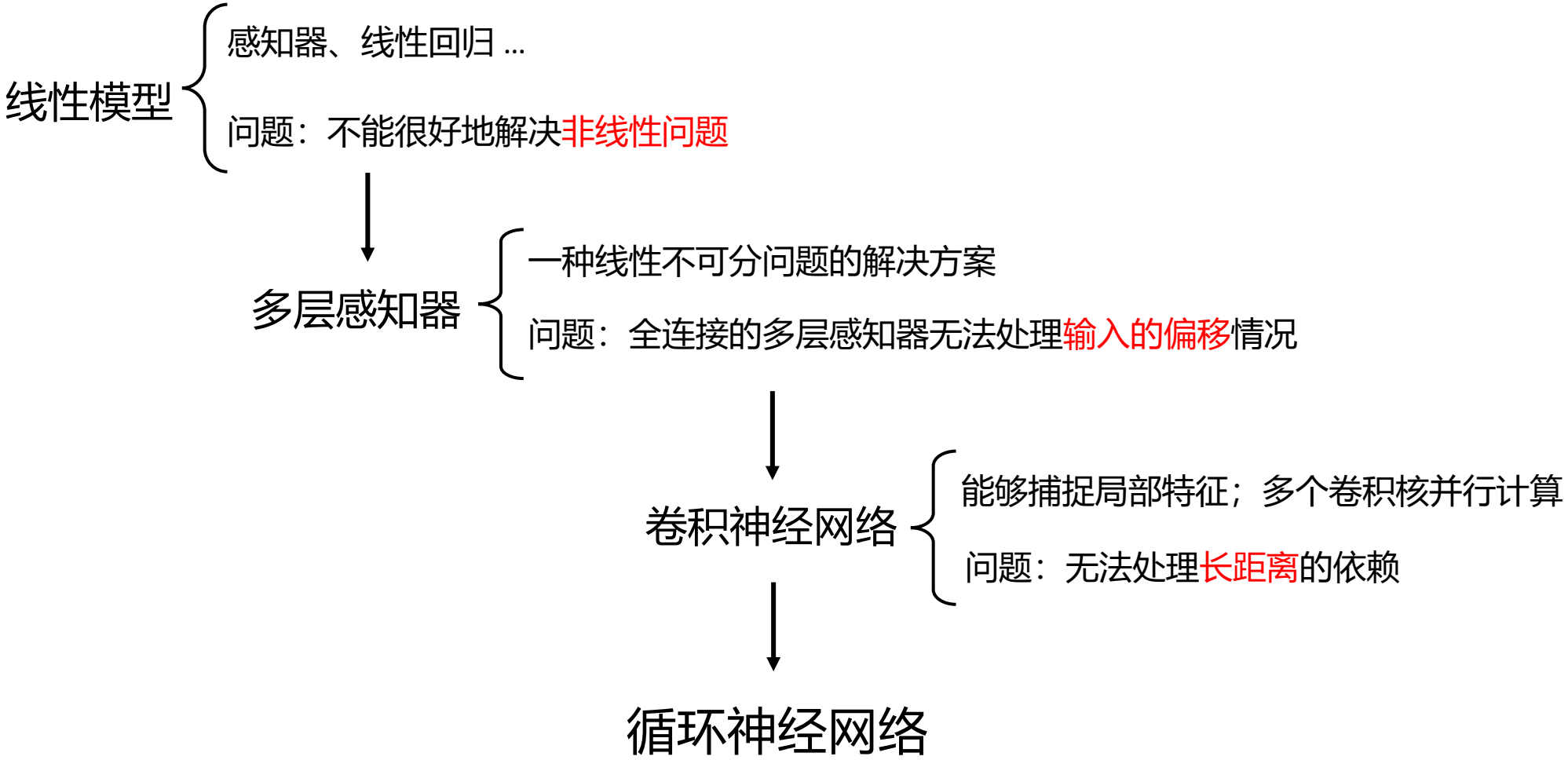
# 循环神经网络

汇报人：程征

# 前言


线性模型
感知器、线性回归 ...
问题：不能很好地解决非线性问题
多层感知器
一种线性不可分问题的解决方案
问题：全连接的多层感知器无法处理输入的偏移情况
卷积神经网络
能够捕捉局部特征；多个卷积核并行计算
问题：无法处理长距离的依赖
循环神经网络

# 模型结构

循环神经网络（Recurrent Neural Network，RNN）指的是网络的隐含层输出又作为其自身的输入，其结构如图1所示。当实际使用循环神经网络时，需要设定一个有限的循环次数，将其展开后相当于堆叠多个共享隐含层参数的前馈神经网络。

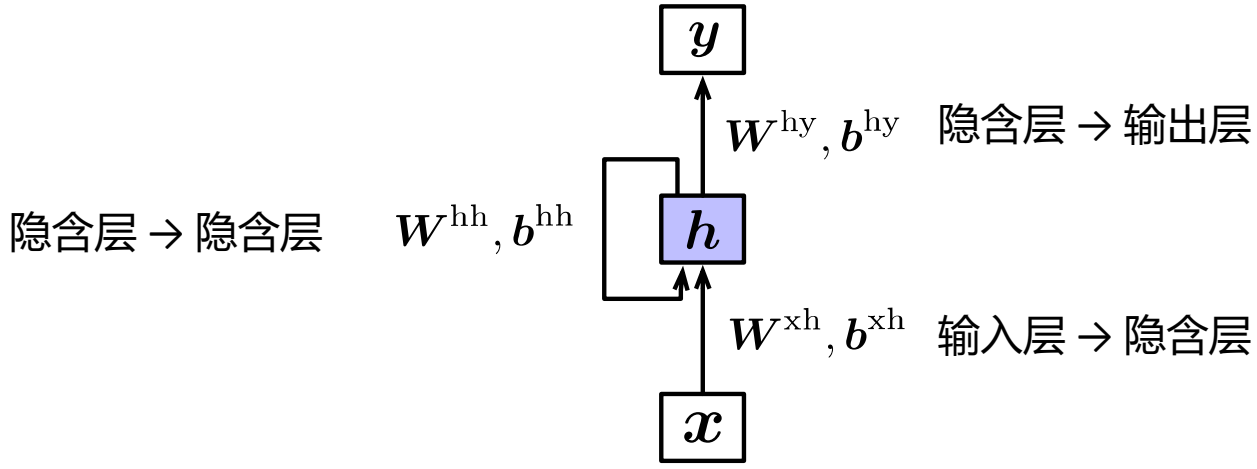



图1 循环神经网络示意图

# 模型结构

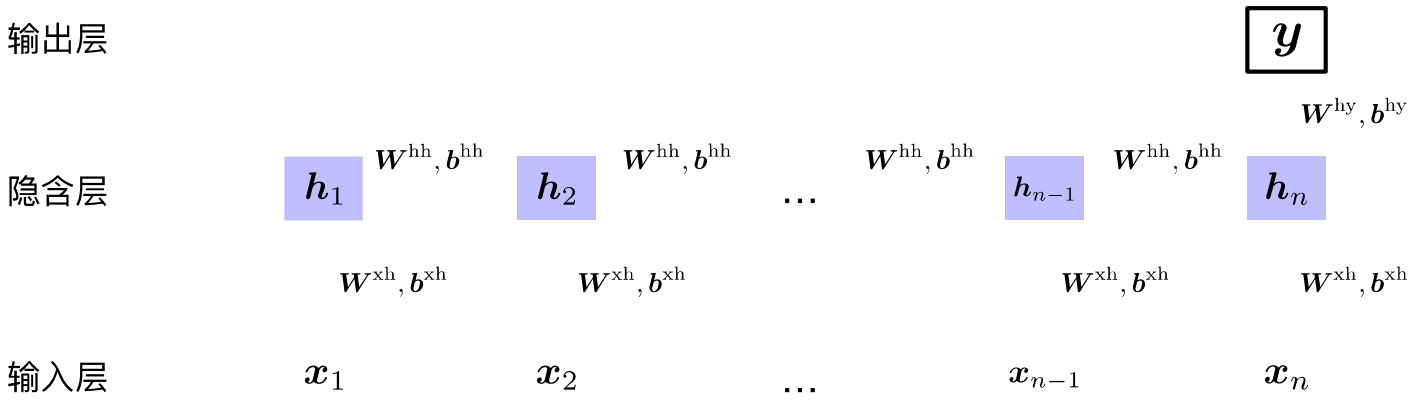



图2 循环神经网络处理序列输入示意图

当使用循环神经网络处理一个序列输入时，需要将循环神经网络按输入时刻展开，然后将序列中的每个输入依次对应到网络不同时刻的输入上，并将当前时刻网络隐含层的输出也作为下一时刻的输入。图2展示了循环神经网络处理序列输入的示意图，其中序列的长度为$n$。按时刻展开的循环神经网络可以使用如下公式描述：

# 模型结构

$$\boldsymbol{h}_t = \tanh(\overbrace{\boldsymbol{W}^{\mathrm{xh}}\boldsymbol{x}_t + \boldsymbol{b}^{\mathrm{xh}}}^{\text{当前时刻的输入}} + \overbrace{\boldsymbol{W}^{\mathrm{hh}}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{\mathrm{hh}}}^{\text{上一时刻的隐含层输出}}) \quad (1)$$

$$\boldsymbol{y} = \mathrm{Softmax}(\boldsymbol{W}^{\mathrm{hy}}\boldsymbol{h}_n + \boldsymbol{b}^{\mathrm{hy}}) \quad (2)$$

式中，$tanh$是激活函数，$t$是输入序列的当前时刻，其隐含层$h_t$不但与当前的输入$x_t$有关，而且与上一时刻的隐含层$h_{t-1}$有关，这实际上是一种递归形式的定义。每个时刻的输入经过层层递归，对最终的输出产生一定的影响，每个时刻的隐含层$h_t$承载了$1\sim t$时刻的全部输入信息，因此循环神经网络的隐含层也称作记忆（Memory）单元。

# 模型结构

以上循环神经网络在最后时刻产生输出结果，此时适用于文本分类等问题。除此之外，如图3所示，还可以在每个时刻产生一个输出结果，这种结构适用于处理自然语言处理中常见的序列标注（Sequence Labeling）问题，如词性标注、命名实体识别，甚至分词等。

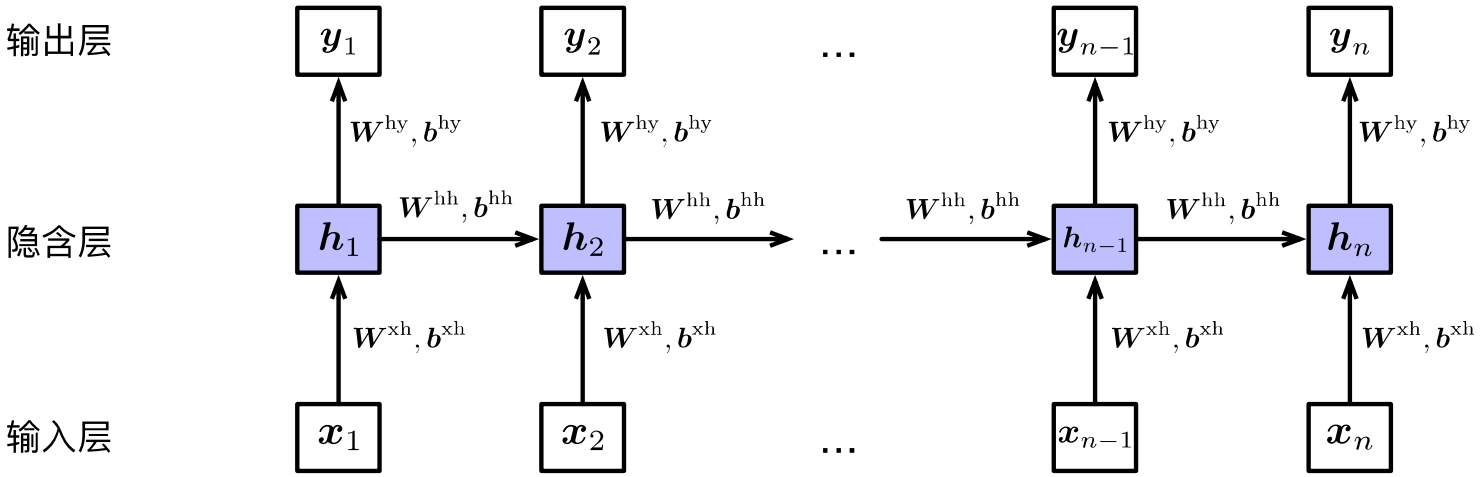



图3 循环神经网络用于处理序列标注问题的示意图

# 长短时记忆网络

在原始的循环神经网络中，信息是通过多个隐含层逐层传递到输出层的。直观上，这会导致信息的损失；更本质地，这会使得网络参数难以优化。长短时记忆网络（Long Short Term Memory，LSTM）可以较好地解决这些问题。

目前优化神经网络的方法都是基于BP，即根据损失函数计算的误差通过梯度反向传播的方式，指导深度网络权值的更新优化。其中将误差从末层往前传递的过程需要**链式法则（Chain Rule）**的帮助，而链式法则是一个**连乘的形式**，所以当层数越深的时候，梯度将以指数形式传播。梯度消失问题和梯度爆炸问题一般随着网络层数的增加会变得越来越明显。在根据损失函数计算的误差通过**梯度反向传播**的方式对深度网络权值进行更新时，得到的**梯度值接近0或特别大**，也就是梯度消失或爆炸。梯度消失或梯度爆炸在本质原理上其实是一样的。

# 长短时记忆网络

长短时记忆网络首先将隐含层的更新方式修改为：

$$\boldsymbol{u}_t = \tanh(\boldsymbol{W}^{\mathrm{xh}}\boldsymbol{x}_t + \boldsymbol{b}^{\mathrm{xh}} + \boldsymbol{W}^{\mathrm{hh}}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{\mathrm{hh}}) \tag{3}$$

$$\boldsymbol{h}_t = \boldsymbol{h}_{t-1} + \boldsymbol{u}_t \tag{4}$$

long term　　short term

相当于直接将 $\boldsymbol{h}_k$ 与 $\boldsymbol{h}_t$ （$k < t$）进行了跨层连接，跨过了中间的$t-k$层，从而减小了网络的层数，使得网络更容易被优化。

$$\begin{aligned}\boldsymbol{h}_t &= \boldsymbol{h}_{t-1} + \boldsymbol{u}_t \\ &= \boldsymbol{h}_{t-2} + \boldsymbol{u}_{t-1} + \boldsymbol{u}_t \\ &= \boldsymbol{h}_k + \boldsymbol{u}_{k+1} + \boldsymbol{u}_{k+2} + \cdots + \boldsymbol{u}_t\end{aligned}$$

# 长短时记忆网络

不过简单地将旧状态$h_{t-1}$和新状态$u_t$进行相加，这种更新方式过于粗糙，并没有考虑两种状态对$h_t$贡献的大小。为了解决这一问题，可以通过前一时刻的隐含层和当前输入计算一个系数，并以此系数对两个状态加权求和，具体公式为：

$$\boldsymbol{f}_t = \sigma(\boldsymbol{W}^{\mathrm{f,xh}}\boldsymbol{x}_t + \boldsymbol{b}^{\mathrm{f,xh}} + \boldsymbol{W}^{\mathrm{f,hh}}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{\mathrm{f,hh}}) \tag{5}$$

$$\boldsymbol{h}_t = \boldsymbol{f}_t \odot \boldsymbol{h}_{t-1} + (1 - \boldsymbol{f}_t) \odot \boldsymbol{u}_t \tag{6}$$

式中，$\sigma$表示Sigmoid函数，其输出恰好介于0-1之间，可作为加权求和的系数；$\odot$表示Hardamard乘积，即按张量对应元素进行相乘；此时$f_t$被称作遗忘门（Forget gate），因为如果其较小时，旧状态$h_{t-1}$对当前状态的贡献也较小，也就是将过去的信息遗忘了。

# 长短时记忆网络

然而，这种加权的方式有一个问题，就是旧状态$h_{t-1}$和新状态$u_t$的贡献是互斥的，也就是如果$f_t$较小，则$1-f_t$就会较大，反之亦然。但是，这两种状态对当前状态的贡献有可能都比较大或者比较小，因此需要使用独立的系数分别控制。因此，引入新的系数以及新的加权方式，即：

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}^{\mathrm{i,xh}}\boldsymbol{x}_t + \boldsymbol{b}^{\mathrm{i,xh}} + \boldsymbol{W}^{\mathrm{i,hh}}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{\mathrm{i,hh}}) \tag{7}$$

$$\boldsymbol{h}_t = \boldsymbol{f}_t \odot \boldsymbol{h}_{t-1} + \boldsymbol{i}_t \odot \boldsymbol{u}_t \tag{8}$$

式中，新的系数$i_t$用于控制输入状态$u_t$对当前状态的贡献，因此又被称作输入门（Input gate）

# 长短时记忆网络

类似地，还可以对输出增加门控机制，即输出门（Output gate）：

$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}^{\mathrm{o,xh}}\boldsymbol{x}_t + \boldsymbol{b}^{\mathrm{o,xh}} + \boldsymbol{W}^{\mathrm{o,hh}}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{\mathrm{o,hh}}) \tag{9}$$

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \boldsymbol{u}_t \tag{10}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t) \tag{11}$$

式中，$c_t$又被称作记忆细胞（Memory cell），即存储（记忆）了截至当前时刻的重要信息。与原始的循环神经网络一样，既可以使用$h_n$预测最终的输出结果，又可以使用$h_t$预测每个时刻的输出结果。

# 长短时记忆网络

无论是传统的循环神经网络还是LSTM，信息流动都是单向的，在一些应用中，这并不合适，如词性标注任务，一个词的词性不但与前面的单词及其自身有关，还与后面的单词有关，但是传统的循环神经网络并不能利用某一时刻后面的信息。

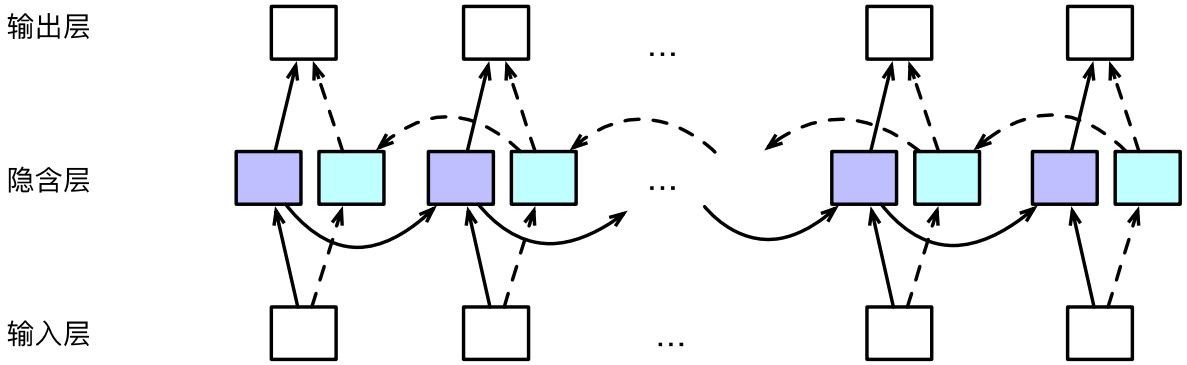



图4 双向循环神经网络结构

为了解决该问题，可以使用双向循环神经网络或双向LSTM，简称Bi-RNN或Bi-LSTM，其中Bi代表Bidirectional。其思想是将同一个输入序列分别接入向前或向后两个循环神经网络中，然后再将两个循环神经网络的隐含层拼接在一起，共同接入输出层进行预测。

# 长短时记忆网络

另一类对循环神经网络的改进方式是将多个网络堆叠起来，形成堆叠循环神经网络（Stacked RNN），如图5所示。此外，还可以在堆叠循环神经网络的每一层加入一个反向循环神经网络，构成更复杂的堆叠双向循环神经网络。

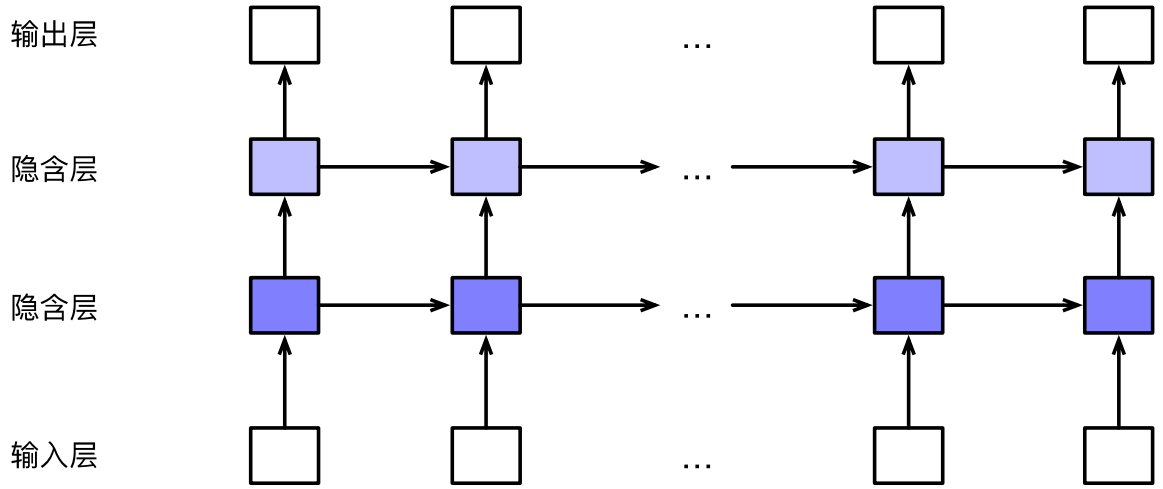



图5 堆叠循环神经网络示意图

# 模型实现

```python
from torch.nn import RNN

# 定义一个RNN，每个时刻输入大小为4，隐含层大小为5

rnn = RNN(input_size=4,   # 每个时刻输入的大小
          hidden_size=5,   # 隐含层的大小
          batch_first=True,   # 根据习惯，通常设置为Tru
          bidirectional=False,   # 是否使用双向RNN，默
          num_layers=1   # 堆叠的循环神经网络层数，默认为
         )

# 输入 输入数据的形状为（barch, seq_len, input_size）
#输入数据批次大小为2，即有两个序列，每个序列的长度为3，每个时
inputs = torch.rand(2,3,4)

# 输出 输出数据有两个,分别为隐含层序列和最后一个时刻的隐含层
# 形状分别为（batch, seq_len, hidden_size）和（1, bat
outputs,hn = rnn(inputs)
```

```python
# 隐含层序列
outputs, outputs.shape
# 最后一个时刻的隐含层
hn, hn.shape
```

```
(tensor([[[ 0.1714,  0.1262, -0.0883,  0.3072, -0.3320],
          [-0.2307,  0.2539,  0.0974,  0.2822,  0.1645],
          [-0.2500,  0.1128,  0.4913, -0.0663, -0.2225]],

         [[-0.1063,  0.3026,  0.0704,  0.2954,  0.0970],
          [ 0.2079,  0.0405,  0.1882,  0.0798, -0.5700],
          [-0.2511,  0.1317, -0.0941,  0.3082,  0.0565]]],
        grad_fn=<TransposeBackward1>),
 torch.Size([2, 3, 5]))
(tensor([[[-0.2500,  0.1128,  0.4913, -0.0663, -0.2225],
          [-0.2511,  0.1317, -0.0941,  0.3082,  0.0565]]],
        grad_fn=<StackBackward0>),
 torch.Size([1, 2, 5]))
```

# 模型实现

```python
from torch.nn import LSTM

# LSTM 输入数据与RNN相同，不同之处在于其输出数据除了最后一个时刻的隐含层hn，
# 还输出了最后一个时刻的记忆细胞cn
lstm = LSTM(input_size=4,hidden_size=5,batch_first=True)
inputs = torch.rand(2,3,4)
outputs,(hn,cn) = lstm(inputs)

# 隐含层序列
outputs, outputs.shape
# 最后一个时刻的隐含层
hn, hn.shape
# 最后一个时刻的记忆细胞
cn, cn.shape
```

```
(tensor([[[-0.1108, -0.0481, -0.0612, -0.1043,  0.1479],
          [-0.1000,  0.0107, -0.0975, -0.1870,  0.1898],
          [-0.1201,  0.0903, -0.1166, -0.2071,  0.2260]],

         [[-0.1182, -0.0611, -0.0833,  0.0139,  0.0599],
          [-0.1840,  0.1898, -0.0853, -0.0879,  0.1154],
          [-0.1120,  0.0737, -0.0971, -0.2451,  0.2480]]],
        grad_fn=<TransposeBackward0>),
 torch.Size([2, 3, 5]))
(tensor([[[-0.1201,  0.0903, -0.1166, -0.2071,  0.2260],
          [-0.1120,  0.0737, -0.0971, -0.2451,  0.2480]]],
        grad_fn=<StackBackward0>),
 torch.Size([1, 2, 5]))
(tensor([[[-0.3403,  0.1331, -0.2023, -0.4868,  0.3749],
          [-0.2871,  0.1273, -0.1851, -0.4817,  0.3854]]],
        grad_fn=<StackBackward0>),
 torch.Size([1, 2, 5]))
```

# 基于LSTM的文本分类

```python
#数据整理
from torch.nn.utils.rnn import pad_sequence

def collate_fn_lstm(examples):
    #获得每个序列的长度
    lengths = torch.tensor([len(ex[0]) for ex in examples])
    inputs = [torch.tensor(ex[0]) for ex in examples]
    targets = torch.tensor([ex[1] for ex in examples],dtype=torch.long)
    #对批次内的样本进行补齐，使其具有相同的长度
    inputs = pad_sequence(inputs,batch_first=True)
    return inputs,lengths,targets
```

# 基于LSTM的文本分类

```python
from torch.nn.utils.rnn import pack_padded_sequence, pad_sequence

class LSTM(nn.Module):
    def __init__(self,vocab_size,embedding_dim,hidden_dim,num_class):
        super(LSTM,self).__init__()
        self.embeddings = nn.Embedding(vocab_size,embedding_dim)
        self.lstm = nn.LSTM(embedding_dim,hidden_dim,batch_first=True, bidirectional=True)
        self.output = nn.Linear(hidden_dim,num_class)
    def forward(self,inputs,lengths):
        embeddings = self.embeddings(inputs)
        #使用pack_padded_sequence函数将变长序列打包
        x_pack = pack_padded_sequence(embeddings,lengths,batch_first=True,enforce_sorted=False)
        hidden,(hn,cn) = self.lstm(x_pack)
        outputs = self.output(hn[-1])
        return outputs
```

# pack_padded_sequence

```python
pack_padded_sequence(sample,   # 输入使用pad_sequence补齐后的样本序列
                     lengths,  # 每条样本序列的长度
                     batch_first=True,   # batch是否在第一位
                     enforce_sorted=False  # 如果是True，则输入应该是按长度降序排序的序列。
                    )                      # 如果是 False ，会在函数内部进行排序。默认值为 True
```

lengths.cpu()

```
Args:
    input (Tensor): padded batch of variable length sequences.
    lengths (Tensor or list(int)): list of sequence lengths of each batch
        element (must be on the CPU if provided as a tensor).
    batch_first (bool, optional): if ``True``, the input is expected in ``B x T x *``
        format.
    enforce_sorted (bool, optional): if ``True``, the input is expected to
        contain sequences sorted by length in a decreasing order. If
        ``False``, the input will get sorted unconditionally. Default: ``True``.
```

# pack_padded_sequence

举例说明

```
# 定义四个样本序列
x0 = torch.tensor([1,2], dtype=torch.float)
x1 = torch.tensor([1,2,3], dtype=torch.float)
x2 = torch.tensor([1,2,3,4], dtype=torch.float)
x3 = torch.tensor([1,2,3,4,5], dtype=torch.float)
sample = [x0, x1, x2, x3]
sample

# 记录序列的长度
lengths = torch.tensor(
lengths
```

```
[53]: [tensor([1., 2.]),
       tensor([1., 2., 3.]),
       tensor([1., 2., 3., 4.]),
       tensor([1., 2., 3., 4., 5.])
[53]: tensor([2, 3, 4, 5])
```

```
# 使用pad_sequence将序列补齐
sample = pad_sequence(sample, batch_first=True)
sample
```

```
[50]: tensor([[1., 2., 0., 0., 0.],
              [1., 2., 3., 0., 0.],
              [1., 2., 3., 4., 0.],
              [1., 2., 3., 4., 5.]])
```

# pack_padded_sequence

以列的形式进行打包

```
pack_padded_sequence(sample, lengths, batch_first=True, enforce_sorted=False)
```

```
[57]: PackedSequence(data=tensor([1., 1., 1., 1., 2., 2., 2., 2., 3., 3., 3., 4., 4., 5.]), batch_sizes
=tensor([4, 4, 3, 2, 1]), sorted_indices=tensor([3, 2, 1, 0]), unsorted_indices=tensor([3, 2, 1,
0]))
```

由大到小排序后序列的索引号

打包后的序列可以直接被self.lstm对象直接调用

```
# 使用pad_sequence将序列补齐
sample = pad_sequence(sample, batch_first=True)
sample
```

```
[50]: tensor([[1., 2., 0., 0., 0.],
              [1., 2., 3., 0., 0.],
              [1., 2., 3., 4., 0.],
              [1., 2., 3., 4., 5.]])
```

# 基于循环神经网络的序列到序列模型

除了能够处理分类问题和序列标注问题，循环神经网络另一个强大的功能是能够处理序列到序列的理解和生成问题，相应的模型被称为序列到序列模型（Sequence-to-Sequence，Seq2seq），也被称为编码器-解码器模型。序列到序列模型指的是首先对一个序列（如一个自然语言句子）编码，然后再对其解码，即生成一个新的序列。很多自然语言处理问题都可以看作是序列到序列问题，如机器翻译，即首先对源语言的句子编码，然后生成相应的目标语言翻译。

# 基于循环神经网络的序列到序列模型

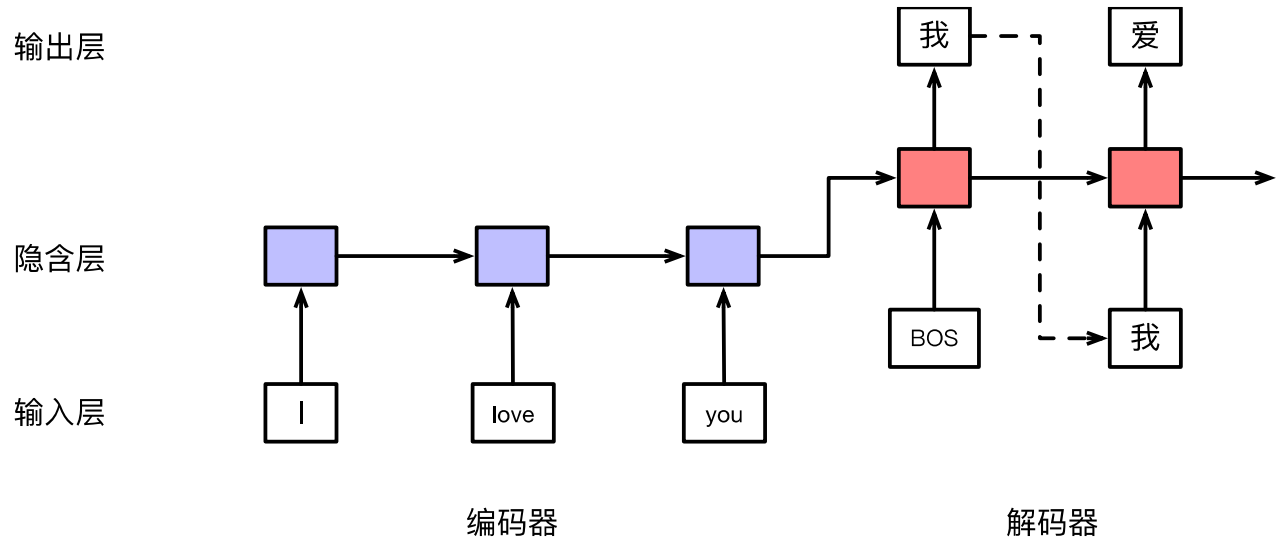



图6 序列到序列模型

图6展示了一个基于序列到序列模型进行机器翻译的示例。首先对编码器使用循环神经网络对源语言句子编码，然后以最后一个单词对应的隐含层作为初始，再调用解码器（另一个循环神经网络）逐词生成目标语言的句子，图中的BOS表示句子起始标记。

# 基于循环神经网络的序列到序列模型

基于循环神经网络的序列到序列模型有一个基本假设，就是原始序列的最后一个隐含状态（一个向量）包含了该序列的全部信息。然而，该假设显然不合理，尤其是当序列比较长时，要做到这一点就更困难。为了解决该问题，**注意力模型**应运而生。

# 注意力模型

为了解决序列到序列模型记忆长序列能力不足的问题，一个非常直观的想法是，当要生成一个目标语言单词时，不光要考虑前一个时刻的状态和已经生成的单词，还要考虑**当前生成的单词和源语言句子中的哪些单词更相关**，即更关注源语言的哪些词，这种做法叫作注意力机制（Attention mechanism）。

# 注意力模型

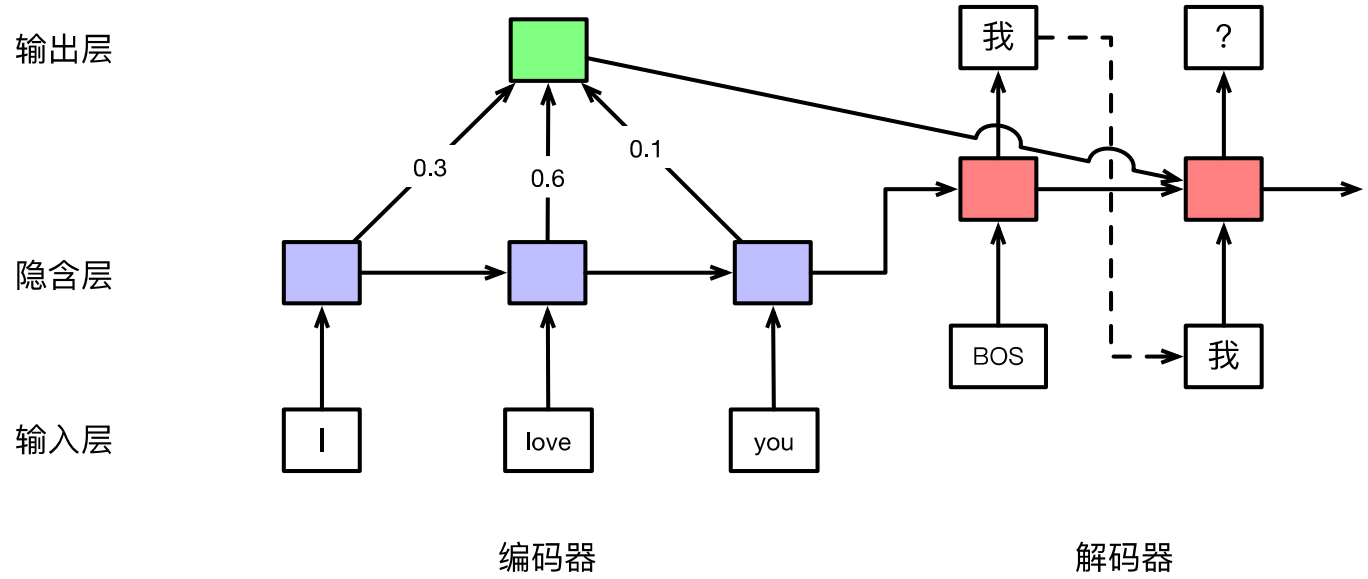



图7 基于注意力机制的序列到序列模型实例

图7给出了一个示例，假设模型已经生成单词“我”后，要生成下一个单词，显然和源语言句子中的“love”关系最大，因此将源语言句子中的“love”对应的状态乘以一个较大的权重，如0.6，而其余词的权重则较小，最终将源语言句子中的每个单词对应的状态进行加权求和，并用作新状态更新的一个额外输入。

# 注意力模型

注意力权重的计算公式为：

$$\hat{\alpha}_s = \mathrm{attn}(\boldsymbol{h}_s, \boldsymbol{h}_{t-1}) \tag{12}$$

$$\alpha_s = \mathrm{Softmax}(\hat{\boldsymbol{\alpha}})_s \tag{13}$$

式中，$h_s$表示源序列中$s$时刻的状态；$h_{t-1}$表示目标序列中前一个时刻的状态；$attn$是注意力的计算公式，即通过两个输入状态的向量，计算一个源序列$s$时刻的注意力分数$\hat{\alpha}_s$;$\hat{\alpha} = [\hat{\alpha}_1, \hat{\alpha}_2, \ldots, \hat{\alpha}_L]$，其中$L$为源序列的长度；最后对整个源序列每个时刻的注意力分数使用$Softmax$函数进行归一化，获得最终的注意力权重$\hat{\alpha}_s$。

# 注意力模型

注意力公式$attn$的计算方式有多种，如：

$$\text{attn}(\boldsymbol{q},\boldsymbol{k}) = \begin{cases} \boldsymbol{w}^\top \tanh(\boldsymbol{W}[\boldsymbol{q};\boldsymbol{k}]) & \text{多层感知器} \\ \boldsymbol{q}^\top \boldsymbol{W}\boldsymbol{k} & \text{双线性} \\ \boldsymbol{q}^\top \boldsymbol{k} & \text{点积} \\ \frac{\boldsymbol{q}^\top \boldsymbol{k}}{\sqrt{d}} & \text{避免因为向量维度}d\text{过大导致点积结果过大} \end{cases}$$

通过引入注意力机制，使得基于循环神经网络的序列到序列模型的准确率有了大幅度的提高。

祝各位学的开心！