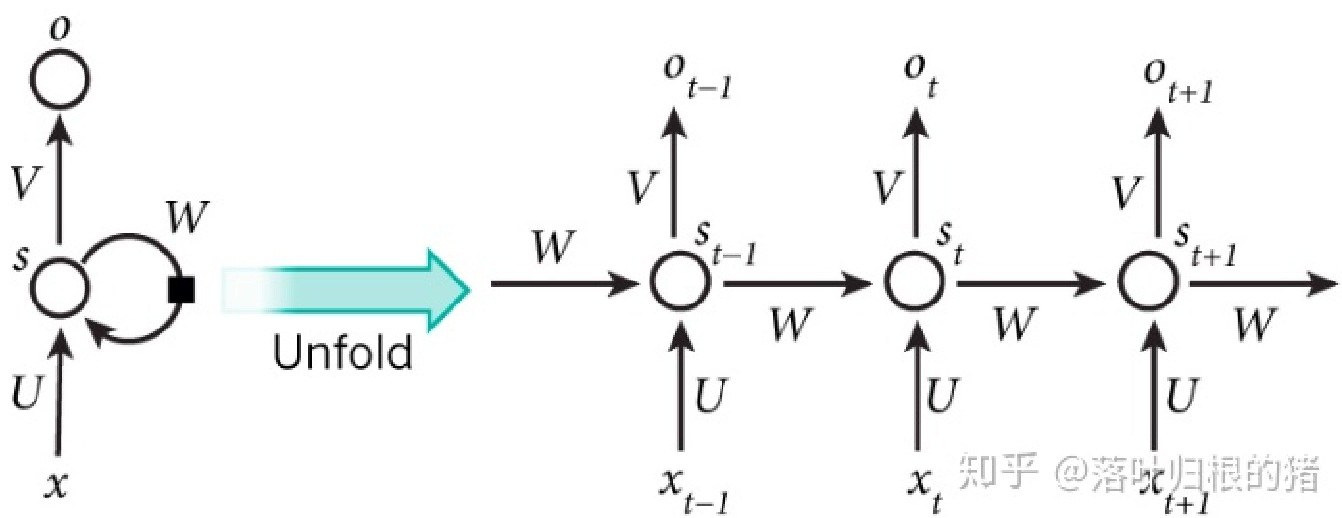
RNN

能用GRU或LSTM（长文本长度最好不要超过100， 太长用bert，transformer等） 尽量不要用RNN
一般不会使用很深的RNN，两层三层

一、知识点-》书上

1.1参数学习

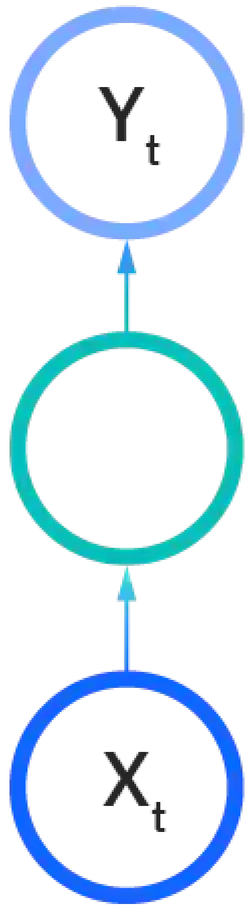
- 计算梯度方式
  - 随时间反向传播算法
  - 实时循环学习算法



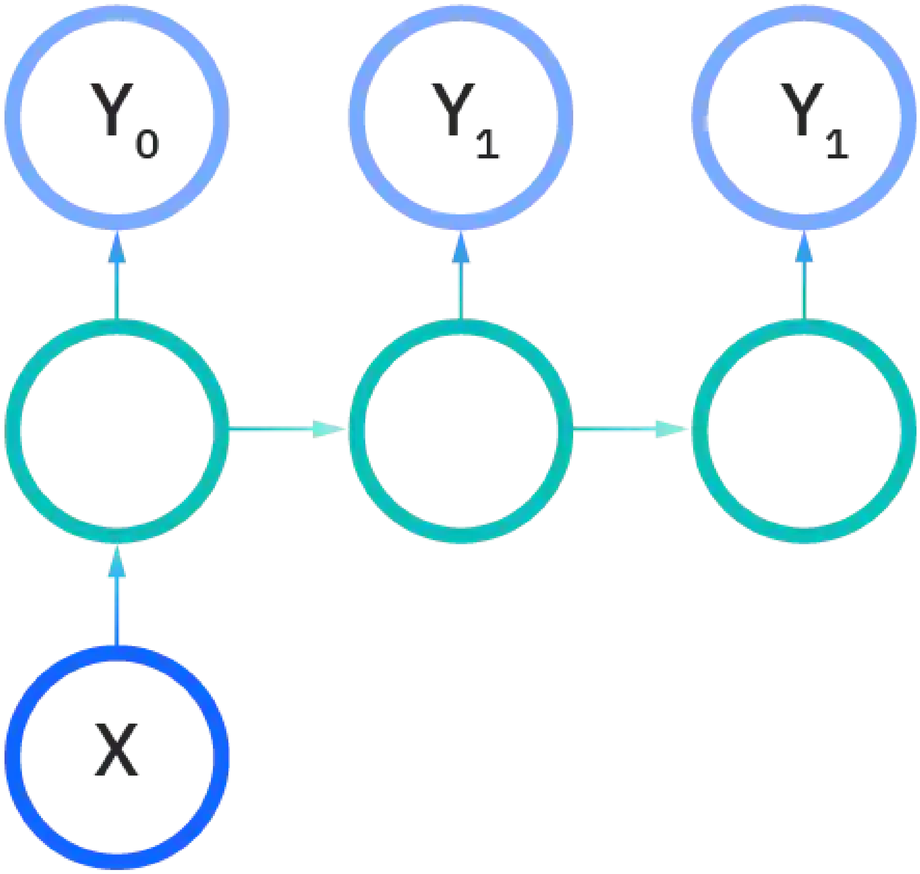
每一个词x(t) 经过的运算(RNNCell)是同一个，多层RNN是多个RNNCell

二、循环神经网络的类型

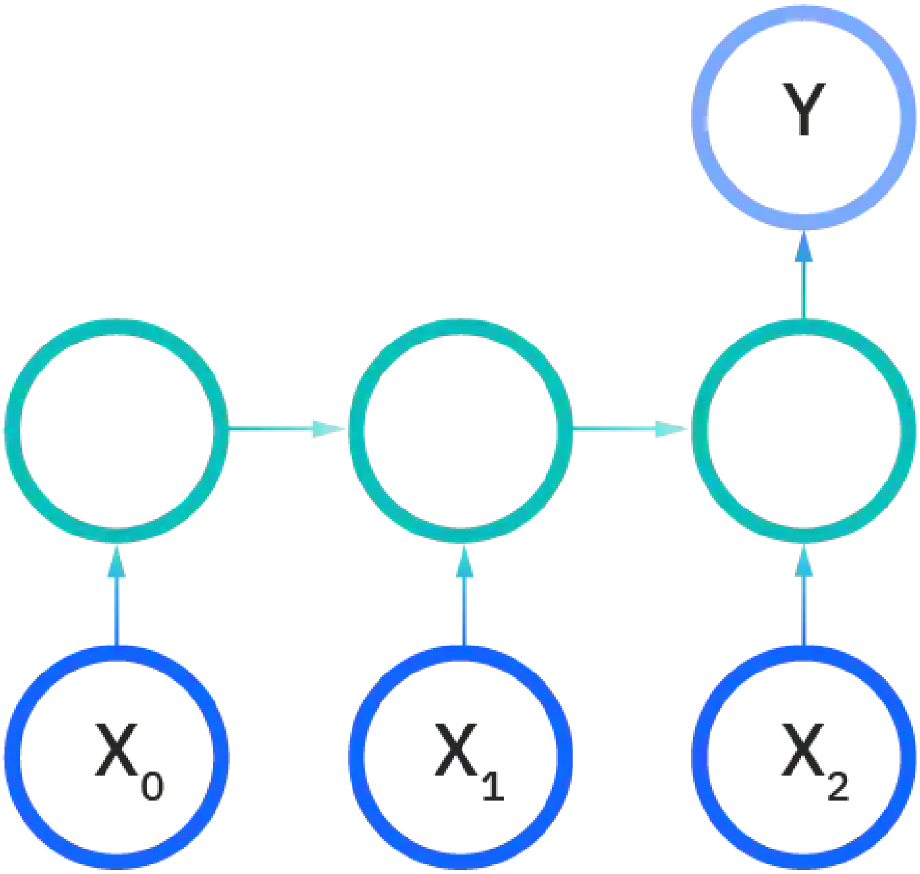
一对一



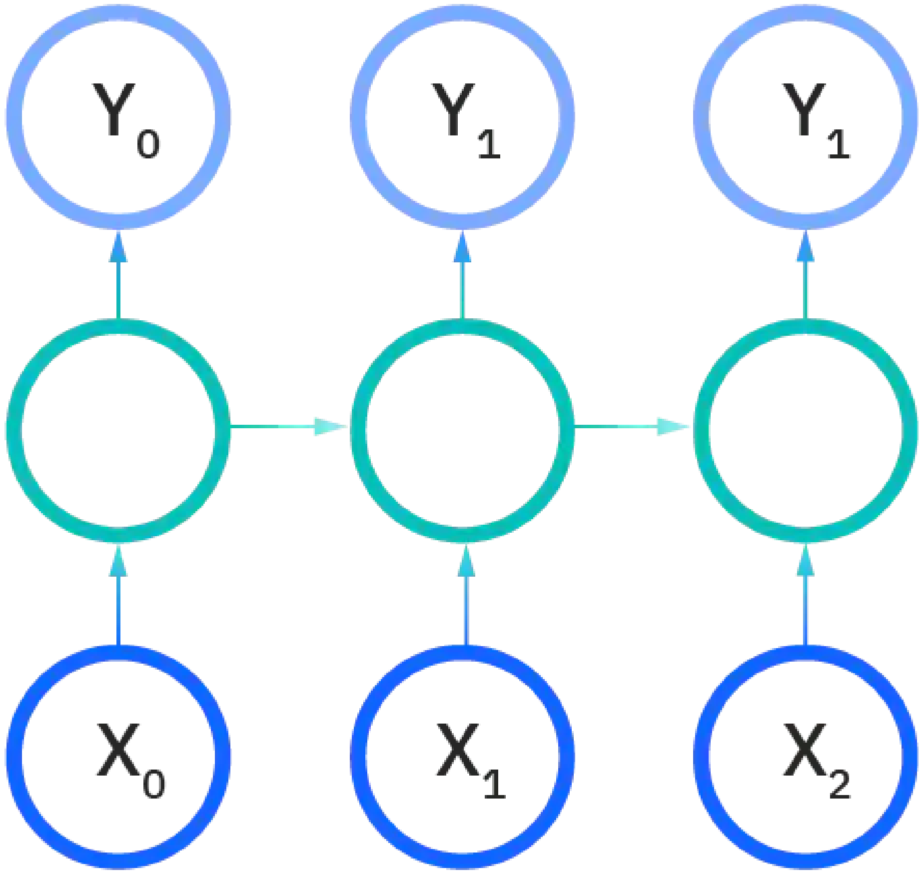
一对多



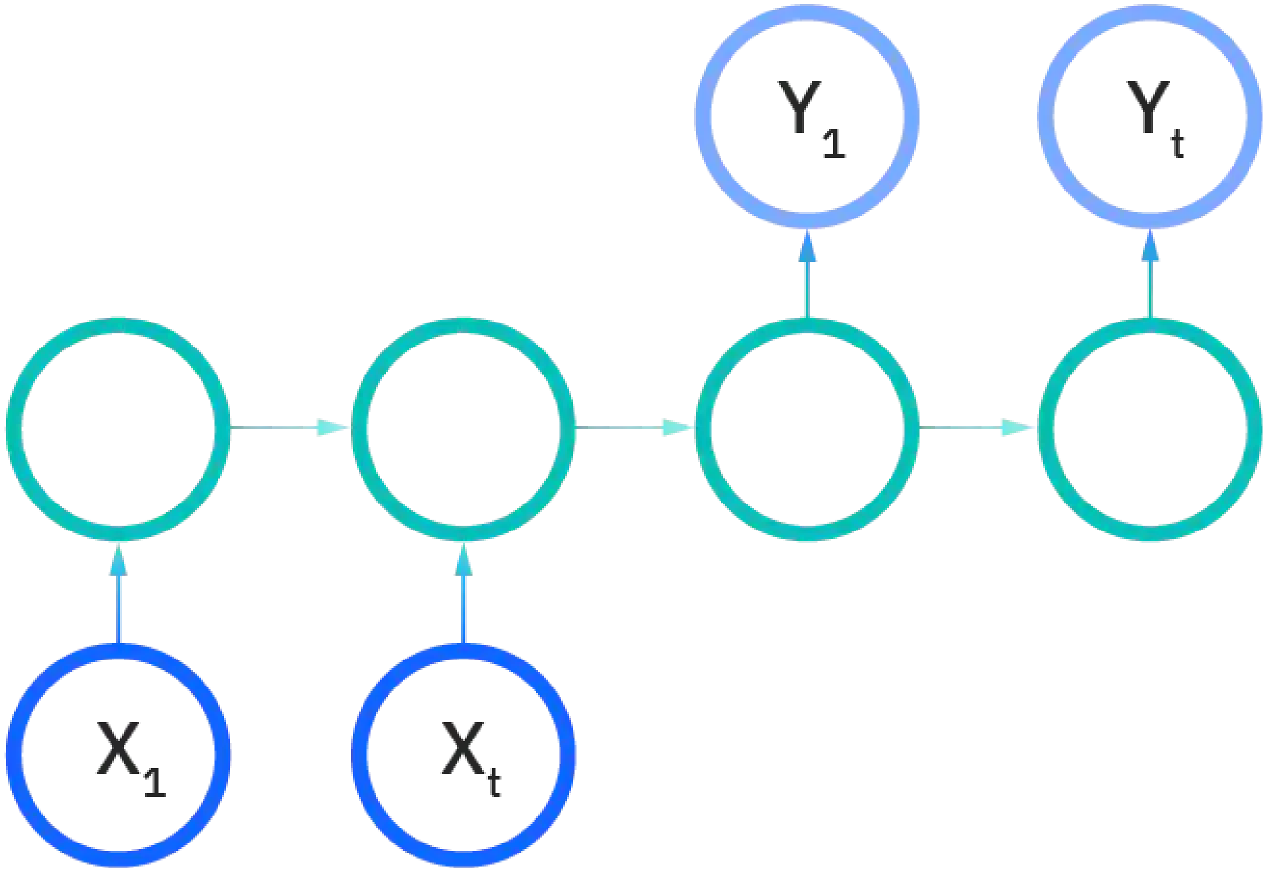
多对一



多对多



Y0
Y1
Y1
X0
X1
X2



三、变体 RNN 架构

双向循环神经网络 (BRNN)：这些是 RNN 的变体网络架构。虽然单向 RNN 只能从先前的输入中提取以预测当前状态，但双向 RNN 会提取未来数据以提高其准确性。如果我们回到本文前面的“feeling under the weather”的例子，如果模型知道序列中的最后一个词是“weather”，它可以更好地预测该短语中的第二个词是“under”。

长短期记忆（LSTM）：这是一种流行的 RNN 架构，由 Sepp Hochreiter 和 Juergen Schmidhuber 引入，作为梯度消失问题的解决方案。在他们的论文(PDF, 388 KB)（链接位于 IBM 外部）中，他们致力于解决长期依赖问题。也就是说，如果影响当前预测的先前状态不是最近的过去，则 RNN 模型可能无法准确预测当前状态。举个例子，假设我们想预测下面的斜体词，“爱丽丝对坚果过敏。她不能吃*花生酱*。”坚果过敏的背景可以帮助我们预测不能食用的食物中含有坚果。但是，如果该上下文是之前的几句话，那么 RNN 将很难甚至不可能连接信息。为了解决这个问题，LSTM 在神经网络的隐藏层中有“细胞”，它们有三个门——一个输入门、一个输出门和一个遗忘门。这些门控制预测网络输出所需的信息流。例如，如果性别代词（例如“she”）在前面的句子中重复多次，您可以将其从单元格状态中排除。

门控循环单元(GRU)：这种 RNN 变体类似于 LSTM，因为它也可以解决 RNN 模型的短期记忆问题。它没有使用“单元状态”来调节信息，而是使用隐藏状态，并且它不是三个门，而是两个——一个重置门和一个更新门。与 LSTM 中的门类似，重置和更新门控制要保留多少信息和哪些信息。

四、代码

RNN模块实现

PYTHON

```python
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)

# import torch
class MyRNN(torch.nn.Module):
    def __init__(self,input_size ,batch_size , num_layers, embedding_s
        super(MyRNN, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.batch_size = batch_size
        self.embedding_size = embedding_size
        self.num_layers = num_layers
        self.embed = torch.nn.Embedding(self.input_size , self.embeddi
        self.RNN = torch.nn.RNN(input_size = self.embedding_size , hid
        self.fc = torch.nn.Linear(hidden_size,input_size)

    def forward(self , x):
        # xs sel_len batch_size x_len
        x = self.embed(x)
        # 可以使用F.one_hot()
        hidden = torch.zeros(self.num_layers , x.size(0) , self.hidden
        x , hn = self.RNN(x,hidden)
        x = self.fc(x)
        return x
```

```python
input_size = len(vocab)
batch_size = 32
hidden_size = 10
num_layers = 3

input = torch.ones( batch_size , 5,dtype=torch.int)
net = MyRNN(input_size ,batch_size , num_layers,10 ,  hidden_size)
outs = net(input)
# print(out.shape)
lossfn = torch.nn.CrossEntropyLoss()
optim = torch.optim.Adam(net.parameters(),lr=0.05)

epochs = 5
for epoch in range(epochs):
    net.train()
    train_loss = 0
 step = 0
 for x,y in train_iter:
        outs = net(x)
        loss = lossfn(outs.reshape(-1,input_size) , y.reshape(-1))
        optim.zero_grad()
        loss.backward()
        optim.step()
        train_loss += loss
        step += 1
 print(f'epoch {epoch + 1} loss:{train_loss / step}')
```

RNNCell模块实现

PYTHON

```python
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

```
batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)

# import torch

class MyRNNCell(torch.nn.Module):
    def __init__(self , input_size ,batch_size ,  hidden_size , embedd
        super(MyRNNCell, self).__init__()
        self.input_size = input_size
        self.batch_size = batch_size
        self.hidden_size = hidden_size
        self.embedding_size = embedding_size
        self.sel_len = sel_len
        self.embed = torch.nn.Embedding(self.input_size , self.embeddi
        self.RnnCell = torch.nn.RNNCell(input_size=self.embedding_size
        self.fc = torch.nn.Linear(hidden_size, input_size)

    # def forward(self,x):
 #     x = self.embed(x) #     hidden = torch.zeros(self.batch_size ,
        x = self.embed(x)
        hidden = torch.zeros(self.batch_size , self.hidden_size)
        outs = torch.zeros(batch_size , self.sel_len , self.hidden_siz
        for i in range(self.sel_len):
            xtemp = x[:,i,:]
            hidden = self.RnnCell(xtemp , hidden)
            outs[:,i,:] = hidden
        x = outs
        return self.fc(x)



input_size = len(vocab)
batch_size = 32
hidden_size = 10
num_layers = 3

input = torch.zeros(batch_size, num_steps, dtype=torch.long)
net = MyRNNCell(input_size, batch_size, hidden_size , 10 ,  num_steps
outs = net(input)
```

```python
    # print(out.shape)
lossfn = torch.nn.CrossEntropyLoss()
optim = torch.optim.Adam(net.parameters(), lr=0.05)

epochs = 5
for epoch in range(epochs):
    net.train()
    train_loss = 0
    step = 0
    for x,y in train_iter:
        outs = net(x)
        loss = lossfn(outs.reshape(-1,input_size) , y.reshape(-1))
        optim.zero_grad()
        loss.backward()
        optim.step()
        train_loss += loss
        step += 1
    print(f'epoch {epoch + 1} loss:{train_loss / step}')
```

五、问题探讨

pytorch 的LSTM batch_first=True 和 False的性能对比

pytorch 的LSTM batch_first=True 和 False的性能略有区别，不过区别不大。

下面这篇文章试验结论是batch_first= True要比batch_first = False更快。但是我自己跑结论却是相反，batch_first = False更快。

运行多次的结果：

2.3414649963378906　2.0364670753479004

2.188401699066162　2.2298429012298584

2.2532322406768　2.202291488647461

2.2564923763275146　2.1362855434417725

2.3355021476745605　2.1648573875427246

2.367983818054199　2.4390225410461426

2.3107049465179443　2.3457281589508057

2.261659622192383　2.1843318939208984

2.2949719429016113　2.1492083072662354

看到大部分情况后者更快（batch_first = False更快）。