1. 门控循环单元（GRU）

简单思考一下矩阵连续乘积可以导致梯度消失或梯度爆炸的问题在实践中的意义：

1. **早期观测值对预测所有未来观测值具有非常重要的意义。** 考虑一个极端情况，其中第一个观测值包含一个校验和， 目标是在序列的末尾辨别校验和是否正确。 在这种情况下，第一个词元的影响至关重要。 我们希望有某些机制能够在一个记忆元里存储重要的早期信息。 如果没有这样的机制，我们将不得不给这个观测值指定一个非常大的梯度， 因为它会影响所有后续的观测值。
2. **一些词元没有相关的观测值。** 例如，在对网页内容进行情感分析时， 可能有一些辅助HTML代码与网页传达的情绪无关。 我们希望有一些机制来*跳过*隐状态表示中的此类词元。
3. **序列的各个部分之间存在逻辑中断。** 例如，书的章节之间可能会有过渡存在， 或者证券的熊市和牛市之间可能会有过渡存在。 在这种情况下，最好有一种方法来*重置*我们的内部状态表示。

在学术界已经提出了许多方法来解决这类问题。 其中最早的方法是"长短期记忆"（long-short-term memory，LSTM）。 门控循环单元（gated recurrent unit，GRU） 是一个稍微简化的变体，通常能够提供同等的效果， 并且计算的速度明显更快。

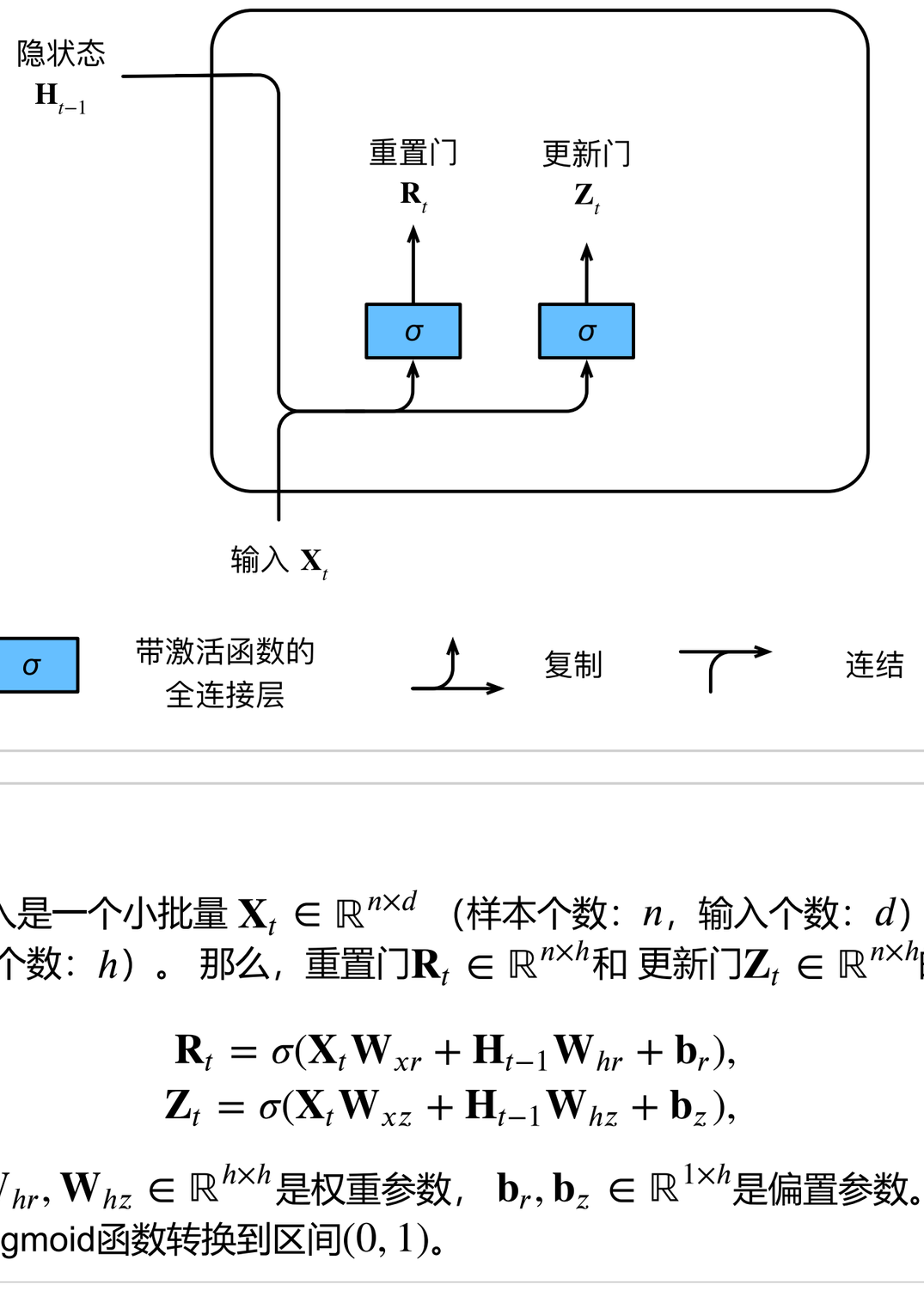
门控隐状态

门控循环单元与普通的循环神经网络之间的关键区别在于： 后者支持隐状态的门控。 这意味着模型有专门的机制来确定应该何时更新隐状态， 以及应该何时重置隐状态。 这些机制是可学习的，并且能够解决了上面列出的问题。 例如，如果第一个词元非常重要， 模型将学会在第一次观测之后不更新隐状态。 同样，模型也可以学会跳过不相关的临时观测。 最后，模型还将学会在需要的时候重置隐状态。

重置门和更新门

首先介绍*重置门*（reset gate）和*更新门*（update gate）。 它们是$(0, 1)$区间中的向量， 可以进行凸组合。 重置门允许我们控制“可能还想记住”的过去状态的数量； 更新门将允许我们控制新状态中有多少个是旧状态的副本。

下图 描述了门控循环单元中的重置门和更新门的输入， 输入是由当前时间步的输入和前一时间步的隐状态给出。 两个门的输出是由使用sigmoid激活函数的两个全连接层给出。



门控循环单元的数学表达。

对于给定的时间步t，假设输入是一个小批量 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ （样本个数：n，输入个数：d）， 上一个时间步的隐状态是 $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ （隐藏单元个数：h）。 那么，重置门$\mathbf{R}_t \in \mathbb{R}^{n \times h}$和 更新门$\mathbf{Z}_t \in \mathbb{R}^{n \times h}$的计算如下所示：

$$\mathbf{R}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r),$$
$$\mathbf{Z}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z),$$

其中$\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d \times h}$ 和$\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h \times h}$是权重参数， $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1 \times h}$是偏置参数。 请注意，在求和过程中会触发广播机制。 输入值通过sigmoid函数转换到区间$(0, 1)$。

候选隐状态

接下来将重置门$\mathbf{R}_t$ 与有隐状态的循环神经网络：

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h).$$

中的常规隐状态更新机制集成， 得到在时间步t的*候选隐状态*（candidate hidden state） $\tilde{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$。

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h),$$

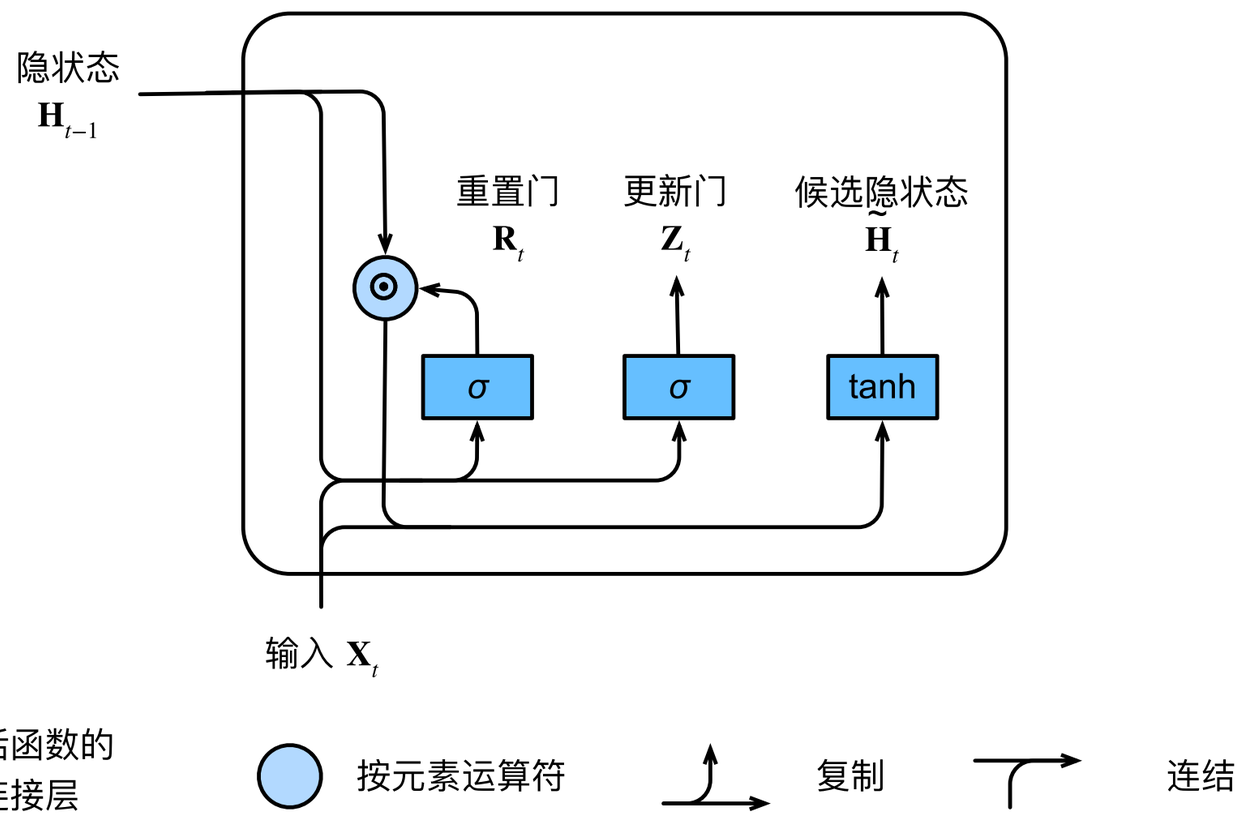
其中$\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ 和$\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$是权重参数， $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$是偏置项， 符号⊙是Hadamard积（按元素乘积）运算符。 在这里使用tanh非线性激活函数来确保候选隐状态中的值保持在区间$(-1, 1)$中。

与有隐状态的循环神经网络相比，

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h),$$

中的$\mathbf{R}_t$和$\mathbf{H}_{t-1}$ 的元素相乘可以减少以往状态的影响。 每当重置门$\mathbf{R}_t$中的项接近1时， 我们恢复一个如有隐状态循环神经网络中的普通的循环神经网络。 对于重置门$\mathbf{R}_t$中所有接近0的项， 候选隐状态是以$\mathbf{X}_t$作为输入的多层感知机的结果。 因此，任何预先存在的隐状态都会被*重置*为默认值。

下图说明了应用重置门之后的计算流程。



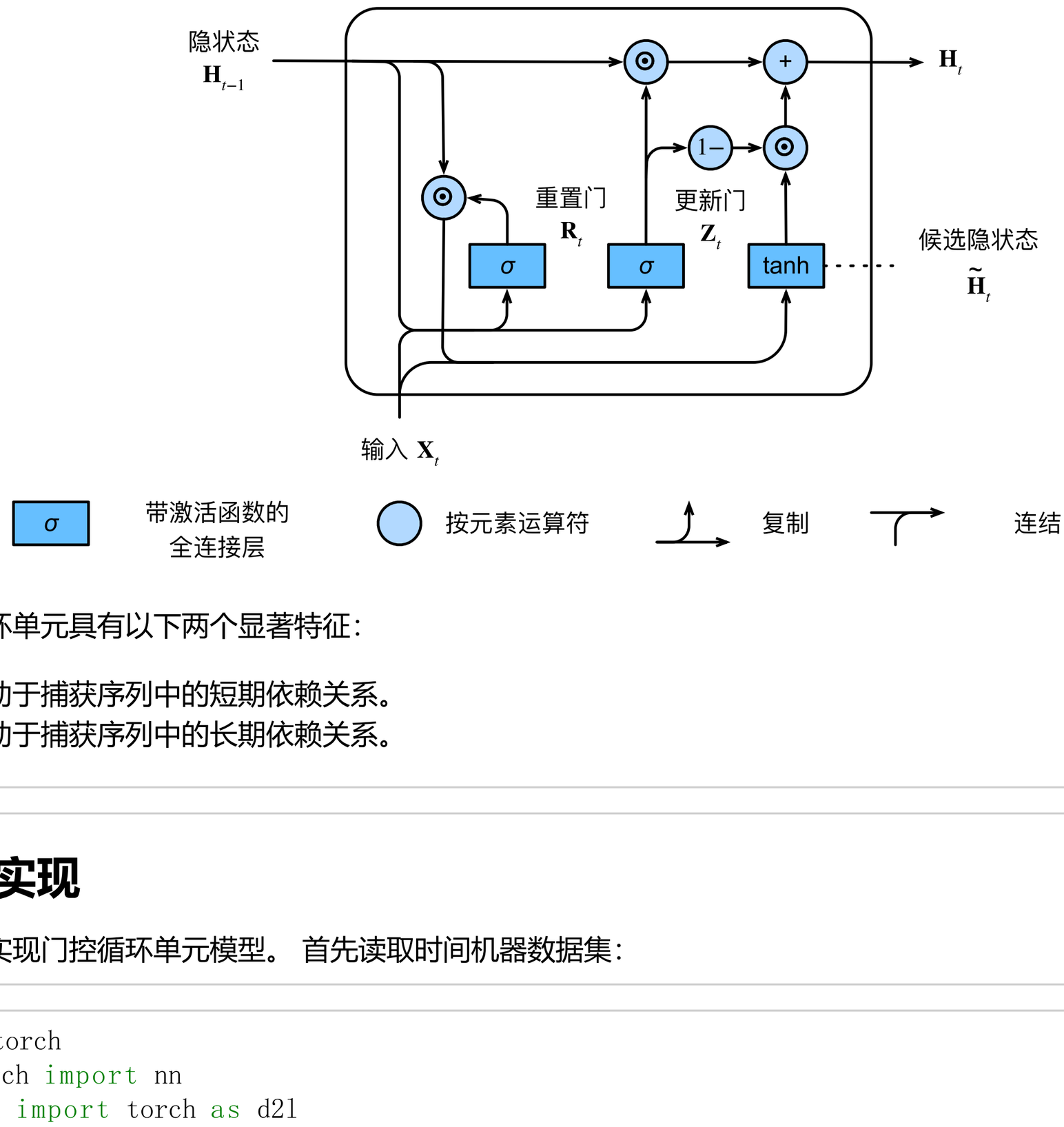
隐状态

上述的计算结果只是候选隐状态，仍然需要结合更新门$\mathbf{Z}_t$的效果。 这一步确定新的隐状态$\mathbf{H}_t \in \mathbb{R}^{n \times h}$ 在多大程度上来自旧的状态$\mathbf{H}_{t-1}$和 新的候选状态$\tilde{\mathbf{H}}_t$。 更新门$\mathbf{Z}_t$仅需要在 $\mathbf{H}_{t-1}$和$\tilde{\mathbf{H}}_t$ 之间进行按元素的凸组合就可以实现这个目标。 这就得出了门控循环单元的最终更新公式：

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t.$$

每当更新门$\mathbf{Z}_t$接近1时，模型就倾向只保留旧状态。 此时，来自$\mathbf{X}_t$的信息基本上被忽略， 从而有效地跳过了依赖链条中的时间步t。 相反，当$\mathbf{Z}_t$接近0时， 新的隐状态$\mathbf{H}_t$就会接近候选隐状态$\tilde{\mathbf{H}}_t$。 这些设计可以帮助我们处理循环神经网络中的梯度消失问题， 并更好地捕获时间步距离很长的序列的依赖关系。 例如，如果整个子序列的所有时间步的更新门都接近于1， 则无论序列的长度如何，在序列起始时间步的旧隐状态都将很容易保留并传递到序列结束。

下图说明了更新门起作用后的计算流。



总之，门控循环单元具有以下两个显著特征：

- 重置门有助于捕获序列中的短期依赖关系。
- 更新门有助于捕获序列中的长期依赖关系。

从零开始实现

现在从零开始实现门控循环单元模型。 首先读取时间机器数据集：

In [1]:
```
import torch
from torch import nn
from d2l import torch as d2l

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

初始化模型参数

下一步是初始化模型参数。 从标准差为0.01的高斯分布中提取权重， 并将偏置项设为0，超参数 `num_hiddens` 定义隐藏单元的数量， 实例化与更新门、重置门、候选隐状态和输出层相关的所有权重和偏置。

In [2]:
```
def get_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return torch.randn(size=shape, device=device)*0.01

    def three():
        return (normal((num_inputs, num_hiddens)),
                normal((num_hiddens, num_hiddens)),
                torch.zeros(num_hiddens, device=device))

    W_xz, W_hz, b_z = three()  # 更新门参数
    W_xr, W_hr, b_r = three()  # 重置门参数
    W_xh, W_hh, b_h = three()  # 候选隐状态参数
    # 输出层参数
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs, device=device)
    # 附加梯度
    params = [W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.requires_grad_(True)
    return params
```

定义模型

现在定义隐状态的初始化函数 `init_gru_state` 。 此函数返回一个形状为（批量大小，隐藏单元个数）的张量，张量的值全部为零。

In [3]:
```
def init_gru_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device), )
```

现在准备定义门控循环单元模型， 模型的架构与基本的循环神经网络单元是相同的， 只是权重更新公式更为复杂。

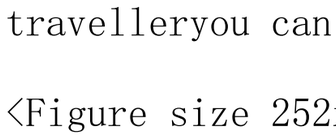
In [4]:
```
def gru(inputs, state, params):
    W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    for X in inputs:
        Z = torch.sigmoid((X @ W_xz) + (H @ W_hz) + b_z)
        R = torch.sigmoid((X @ W_xr) + (H @ W_hr) + b_r)
        H_tilda = torch.tanh((X @ W_xh) + ((R * H) @ W_hh) + b_h)
        H = Z * H + (1 - Z) * H_tilda
        Y = H @ W_hq + b_q
        outputs.append(Y)
    return torch.cat(outputs, dim=0), (H,)
```

训练与预测

训练结束后分别打印输出训练集的困惑度， 以及前缀“time traveler”和“traveler”的预测序列上的困惑度。

In [5]:
```
vocab_size, num_hiddens, device = len(vocab), 256, d2l.try_gpu()
num_epochs, lr = 500, 1
model = d2l.RNNModelScratch(len(vocab), num_hiddens, device, get_params,
                            init_gru_state, gru)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 27622.1 tokens/sec on cuda:0
time traveller for so it will be convenient to speak of himwas e
traveller bether and sevedta excention slight of whick for
```



简洁实现

高级API包含了前文介绍的所有配置细节， 可以直接实例化门控循环单元模型。 这段代码的运行速度要快得多， 因为它使用的是编译好的运算符而不是Python来处理之前阐述的许多细节。

In [6]:
```
num_inputs = vocab_size
gru_layer = nn.GRU(num_inputs, num_hiddens)
model = d2l.RNNModel(gru_layer, len(vocab))
model = model.to(device)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.0, 278571.5 tokens/sec on cuda:0
time travelleryou can show black is white by argument said filby
travelleryou can show black is white by argument said filby
```


2. 长短期记忆网络（LSTM）

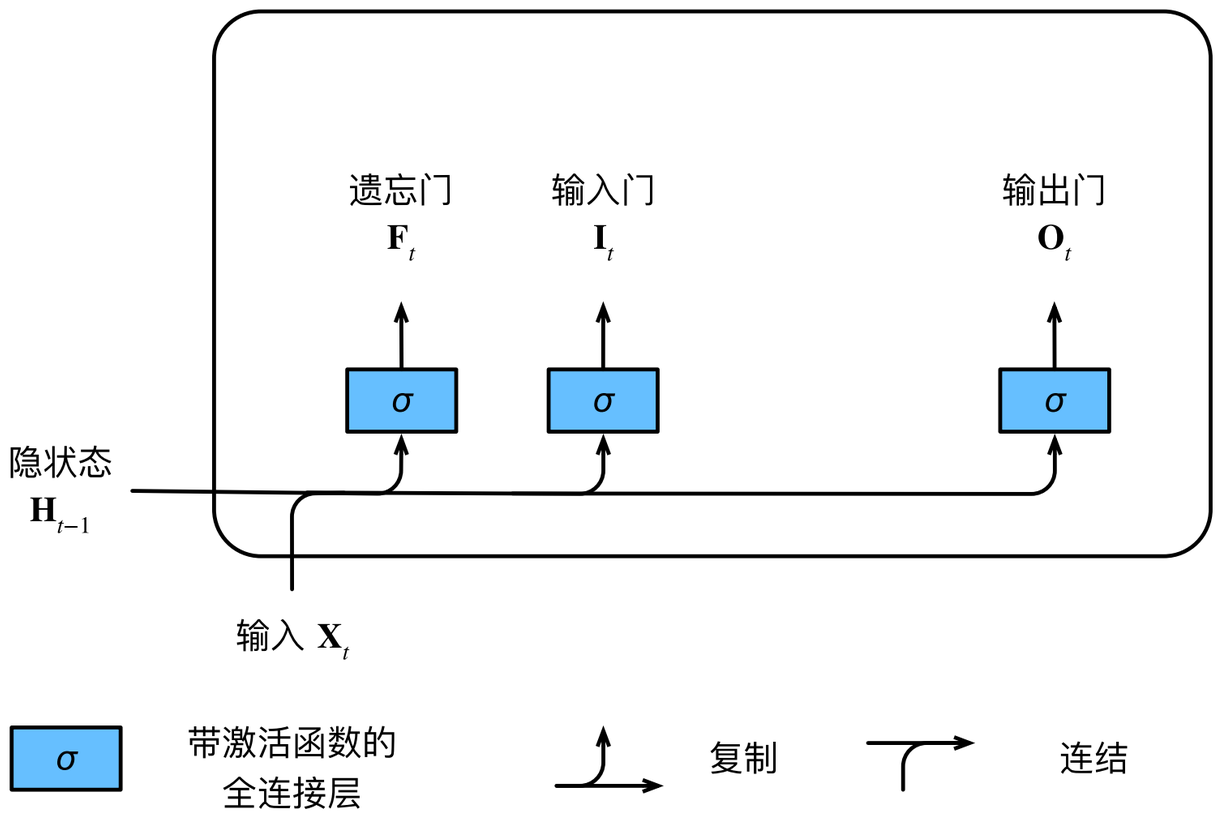
长期以来，隐变量模型存在着长期信息保存和短期输入缺失的问题。 解决这一问题的最早方法之一是长短期存储器（long short-term memory，LSTM）。 它有许多与门控循环单元一样的属性。 有趣的是，长短期记忆网络的设计比门控循环单元稍微复杂一些， 却比门控循环单元早诞生了近20年。

门控记忆元

可以说，长短期记忆网络的设计灵感来自于计算机的逻辑门。 长短期记忆网络引入了*记忆元*（memory cell），或简称为*单元*（cell）。 有些文献认为记忆元是隐状态的一种特殊类型， 它们与隐状态具有相同的形状，其设计目的是用于记录附加的信息。 为了控制记忆元，我们需要许多门。 其中一个门用来从单元中输出条目，我们将其称为*输出门*（output gate）。 另外一个门用来决定何时将数据读入单元，我们将其称为*输入门*（input gate）。 我们还需要一种机制来重置单元的内容，由*遗忘门*（forget gate）来管理， 这种设计的动机与门控循环单元相同， 能够通过专用机制决定什么时候记忆或忽略隐状态中的输入。 让我们看看这在实践中是如何运作的。

输入门、忘记门和输出门

就如在门控循环单元中一样， 当前时间步的输入和前一个时间步的隐状态 作为数据送入长短期记忆网络的门中， 如下图所示。 它们由三个具有sigmoid激活函数的全连接层处理， 以计算输入门、遗忘门和输出门的值。 因此，这三个门的值都在$(0,1)$的范围内。



下面细化一下长短期记忆网络的数学表达。

假设有h个隐藏单元，批量大小为n，输入数为d。 因此，输入为$\mathbf{X}_t \in \mathbb{R}^{n \times d}$， 前一时间步的隐状态为$\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$。 相应地，时间步$t$的门被定义如下： 输入门是$\mathbf{I}_t \in \mathbb{R}^{n \times h}$， 遗忘门是$\mathbf{F}_t \in \mathbb{R}^{n \times h}$， 输出门是$\mathbf{O}_t \in \mathbb{R}^{n \times h}$。 它们的计算方法如下：

$$\begin{aligned}\mathbf{I}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i),\\ \mathbf{F}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f),\\ \mathbf{O}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),\end{aligned}$$

其中$\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d \times h}$ 和$\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h \times h}$是权重参数， $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1 \times h}$是偏置参数。

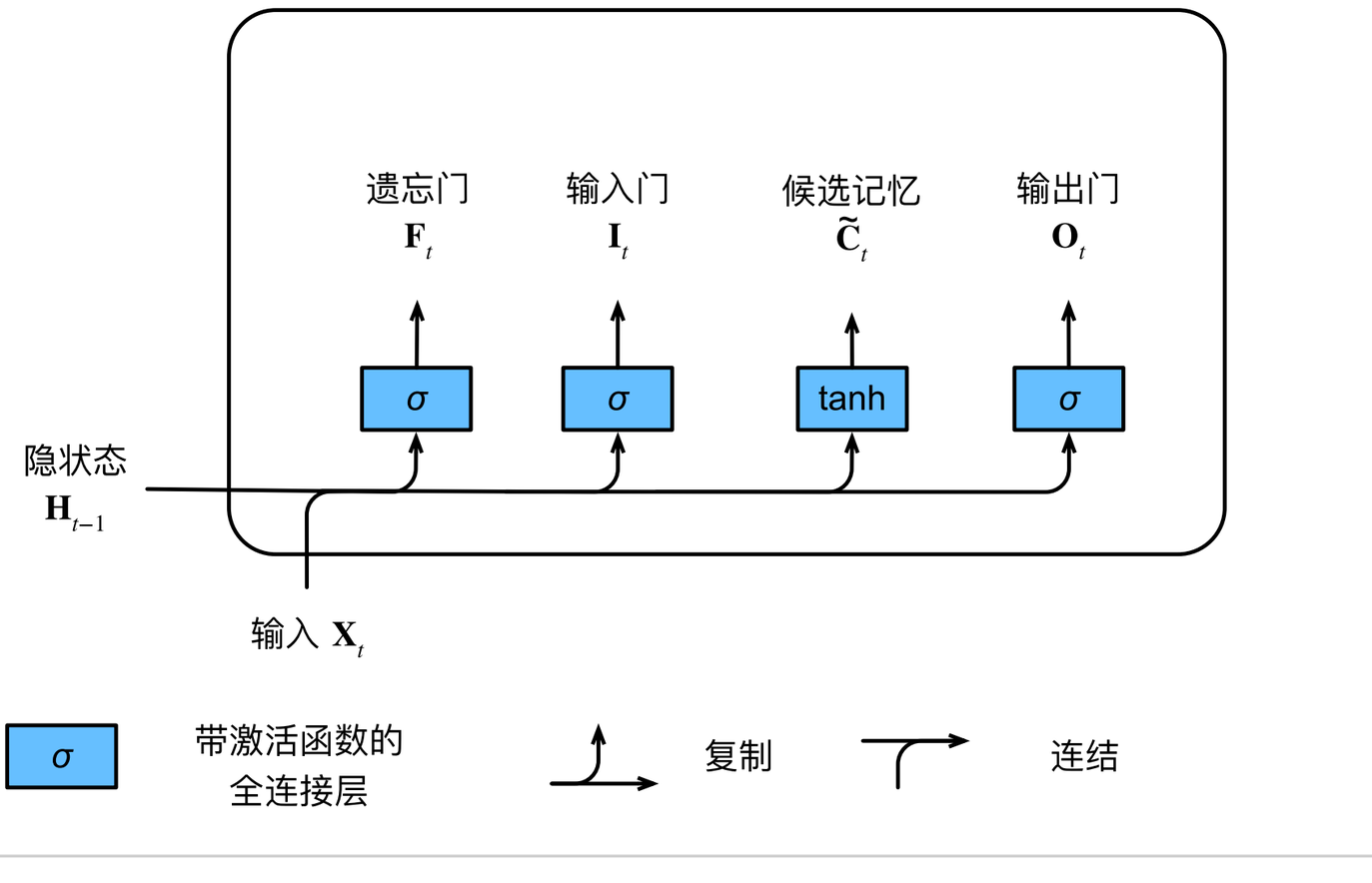
候选记忆元

由于还没有指定各种门的操作，所以先介绍*候选记忆元*（candidate memory cell） $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n \times h}$。 它的计算与上面描述的三个门的计算类似， 但是使用tanh函数作为激活函数，函数的值范围为$(-1,1)$。 下面导出在时间步t处的方程：

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c),$$

其中$\mathbf{W}_{xc} \in \mathbb{R}^{d \times h}$和 $\mathbf{W}_{hc} \in \mathbb{R}^{h \times h}$是权重参数， $\mathbf{b}_c \in \mathbb{R}^{1 \times h}$是偏置参数。

候选记忆元:



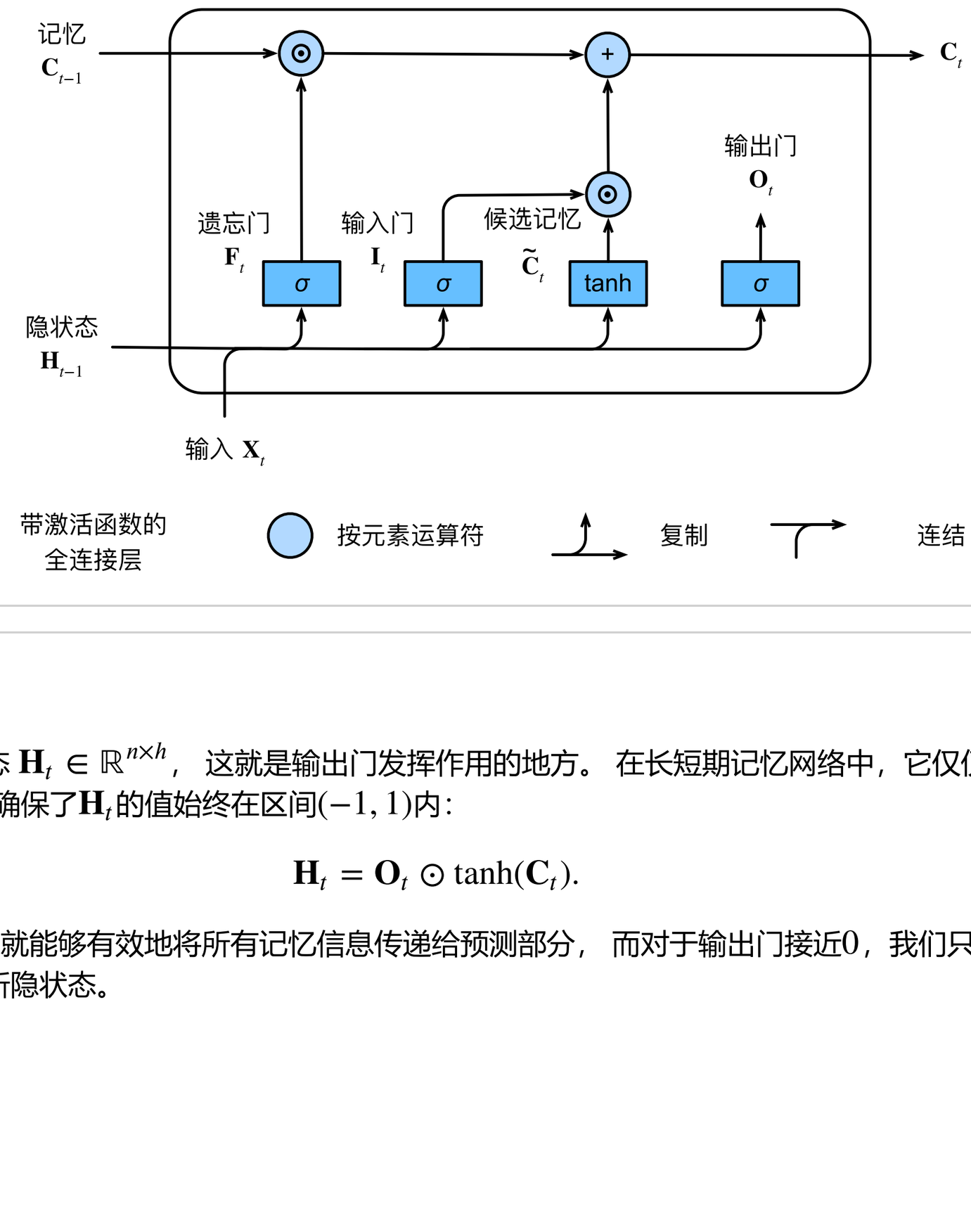
记忆元

在门控循环单元中，有一种机制来控制输入和遗忘（或跳过）。 类似地，在长短期记忆网络中，也有两个门用于这样的目的： 输入门$\mathbf{I}_t$控制采用多少来自$\tilde{\mathbf{C}}_t$的新数据， 而遗忘门$\mathbf{F}_t$控制保留多少过去的 记忆元$\mathbf{C}_{t-1} \in \mathbb{R}^{n \times h}$的内容。 使用按元素乘法，得出：

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t.$$

如果遗忘门始终为1且输入门始终为0， 则过去的记忆元$\mathbf{C}_{t-1}$ 将随时间被保存并传递到当前时间步。 引入这种设计是为了缓解梯度消失问题， 并更好地捕获序列中的长距离依赖关系。

这样我们就得到了计算记忆元的流程图，如:



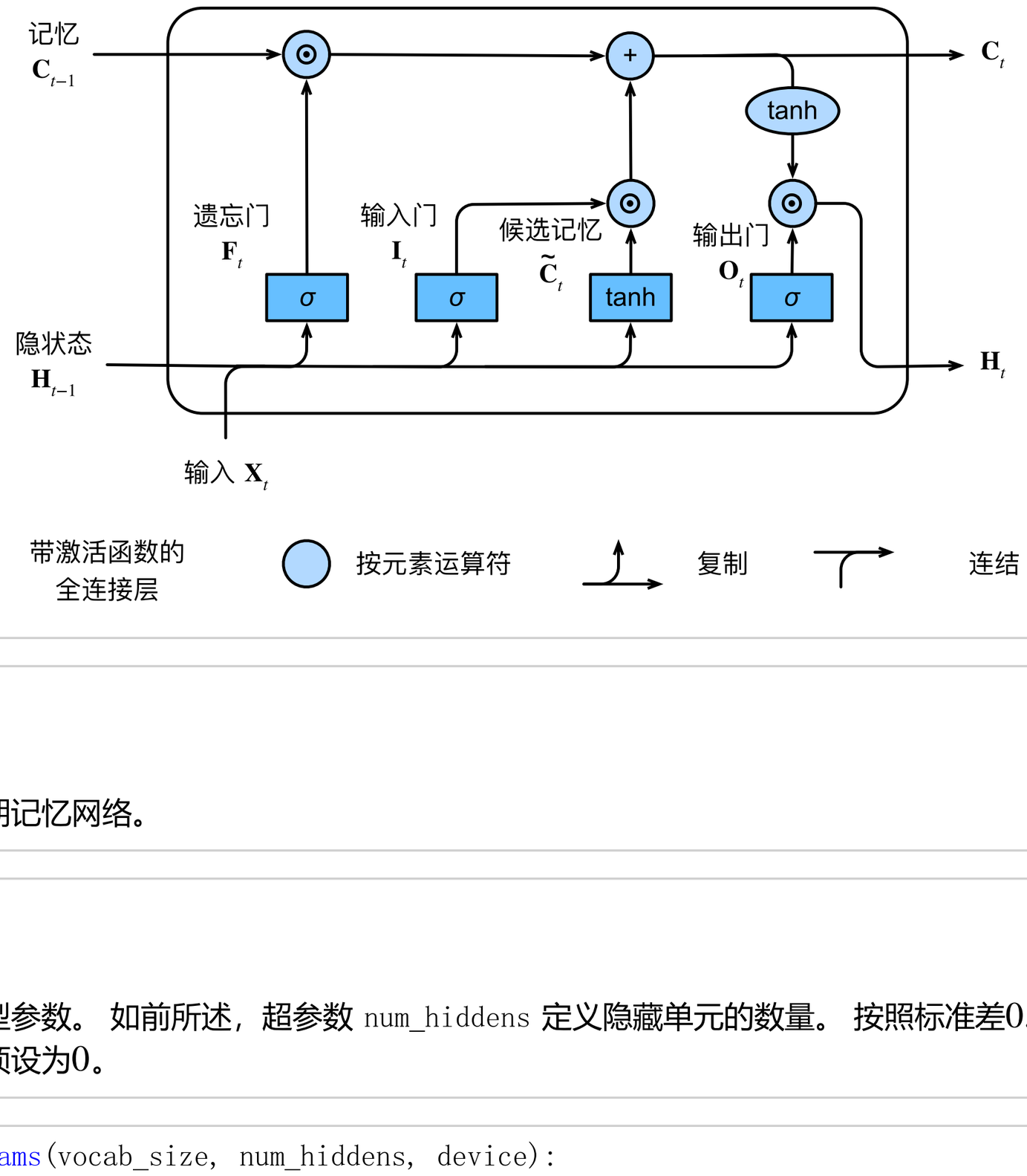
隐状态

最后定义如何计算隐状态$\mathbf{H}_t \in \mathbb{R}^{n \times h}$， 这就是输出门发挥作用的地方。 在长短期记忆网络中，它仅仅是记忆元的tanh的门控版本。 这就确保了$\mathbf{H}_t$的值始终在区间$(-1, 1)$内：

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t).$$

只要输出门接近1，我们就能够有效地将所有记忆信息传递给预测部分， 而对于输出门接近0，我们只保留记忆元内的所有信息，而不需要更新隐状态。

数据流:



从零开始实现

现在从零开始实现长短期记忆网络。

初始化模型参数

接下来定义和初始化模型参数。 如前所述，超参数 `num_hiddens` 定义隐藏单元的数量。 按照标准差0.01的高斯分布初始化权重，并将偏置项设为0。

In [7]:
```
def get_lstm_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return torch.randn(size=shape, device=device)*0.01

    def three():
        return (normal((num_inputs, num_hiddens)),
                normal((num_hiddens, num_hiddens)),
                torch.zeros(num_hiddens, device=device))

    W_xi, W_hi, b_i = three()  # 输入门参数
    W_xf, W_hf, b_f = three()  # 遗忘门参数
    W_xo, W_ho, b_o = three()  # 输出门参数
    W_xc, W_hc, b_c = three()  # 候选记忆元参数
    # 输出层参数
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs, device=device)
    # 附加梯度
    params = [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc,
              b_c, W_hq, b_q]
    for param in params:
        param.requires_grad_(True)
    return params
```

定义模型

在**初始化函数**中， 长短期记忆网络的隐状态需要返回一个*额外*的记忆元， 单元的值为0，形状为（批量大小，隐藏单元数）。 因此，我们得到以下的状态初始化。

In [8]:
```
def init_lstm_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device),
            torch.zeros((batch_size, num_hiddens), device=device))
```

实际模型的定义与前面讨论的一样： 提供三个门和一个额外的记忆元。 请注意，只有隐状态才会传递到输出层， 而记忆元$\mathbf{C}_t$不直接参与输出计算。

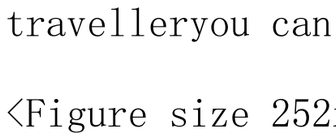
In [9]:
```
def lstm(inputs, state, params):
    [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc, b_c,
     W_hq, b_q] = params
    (H, C) = state
    outputs = []
    for X in inputs:
        I = torch.sigmoid((X @ W_xi) + (H @ W_hi) + b_i)
        F = torch.sigmoid((X @ W_xf) + (H @ W_hf) + b_f)
        O = torch.sigmoid((X @ W_xo) + (H @ W_ho) + b_o)
        C_tilda = torch.tanh((X @ W_xc) + (H @ W_hc) + b_c)
        C = F * C + I * C_tilda
        H = O * torch.tanh(C)
        Y = (H @ W_hq) + b_q
        outputs.append(Y)
    return torch.cat(outputs, dim=0), (H, C)
```

训练和预测

让我们通过实例化 RNN 中 引入的 RNNModelScratch 类来训练一个长短期记忆网络。

In [10]:
```
vocab_size, num_hiddens, device = len(vocab), 256, d2l.try_gpu()
num_epochs, lr = 500, 1
model = d2l.RNNModelScratch(len(vocab), num_hiddens, device, get_lstm_params,
                            init_lstm_state, lstm)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 24560.0 tokens/sec on cuda:0
time traveller for so it will be convenient to speak of himwas e
traveller why thrug has oullig st cenceningming wi ce not u
```



简洁实现

使用高级API可以直接实例化 LSTM 模型。 高级API封装了前文介绍的所有配置细节。

同样， 这段代码的运行速度要快得多， 因为它使用的是编译好的运算符而不是Python来处理之前阐述的许多细节。

In [11]:
```
num_inputs = vocab_size
lstm_layer = nn.LSTM(num_inputs, num_hiddens)
model = d2l.RNNModel(lstm_layer, len(vocab))
model = model.to(device)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

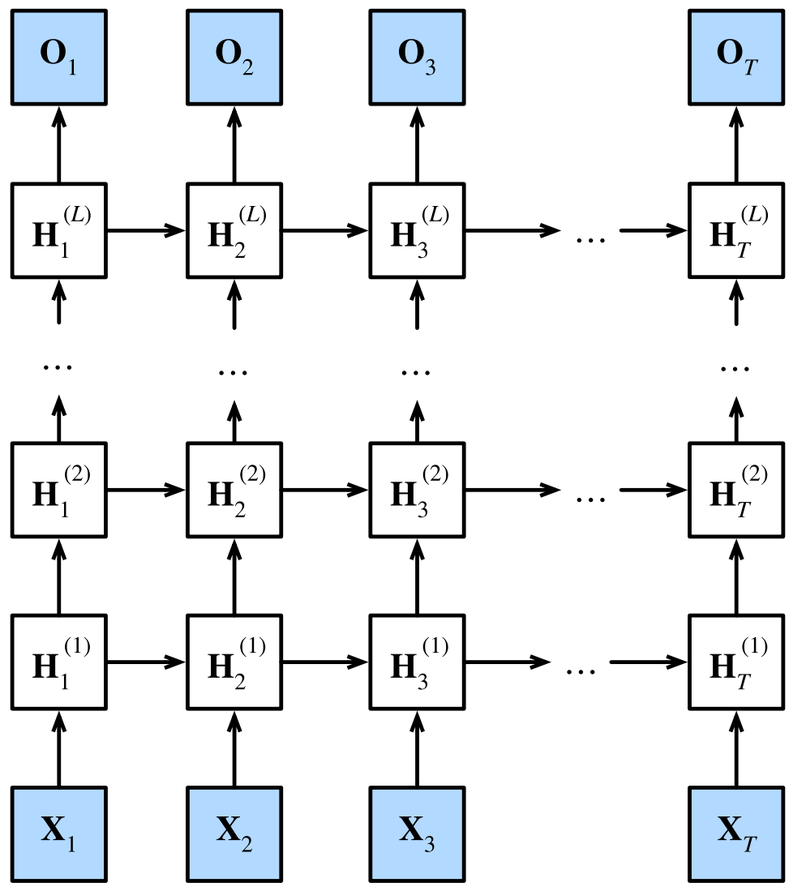
```
perplexity 1.1, 266182.4 tokens/sec on cuda:0
time traveller for so it will be convenient to speak of himwas e
travelleryou can show black is white by argument said filby
```


3. 深度循环神经网络

目前为止只讨论了具有一个单向隐藏层的循环神经网络。 其中，隐变量和观测值与具体的函数形式的交互方式是相当随意的。 只要交互类型建模具有足够的灵活性，这就不是一个大问题。 然而，对于一个单层来说，这可能具有相当的挑战性。 之前在线性模型中，我们通过添加更多的层来解决这个问题。 而在循环神经网络中，我们首先需要确定如何添加更多的层， 以及在哪里添加额外的非线性，因此这个问题有点棘手。

可以将多层循环神经网络堆叠在一起， 通过对几个简单层的组合，产生了一个灵活的机制。 特别是，数据可能与不同层的堆叠有关。 例如，我们可能希望保持有关金融市场状况 （熊市或牛市）的宏观数据可用， 而微观数据只记录较短期的时间动态。

下图描述了一个具有L个隐藏层的深度循环神经网络， 每个隐状态都连续地传递到当前层的下一个时间步和下一层的当前时间步。



函数依赖关系

深度架构中的函数依赖关系可以形式化， 这个架构是由上图中描述了L个隐藏层构成。 后续的讨论主要集中在经典的循环神经网络模型上， 但是这些讨论也适应于其他序列模型。

假设在时间步t有一个小批量的输入数据 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ （样本数：n，每个样本中的输入数：d）。 同时，将l^{th}隐藏层（$l = 1, \ldots, L$） 的隐状态设为$\mathbf{H}_t^{(l)} \in \mathbb{R}^{n \times h}$ （隐藏单元数：h）， 输出层变量设为$\mathbf{O}_t \in \mathbb{R}^{n \times q}$ （输出数：q）。 设置$\mathbf{H}_t^{(0)} = \mathbf{X}_t$， 第$l$个隐藏层的隐状态使用激活函数$\phi_l$，则：

$$\mathbf{H}_t^{(l)} = \phi_l(\mathbf{H}_t^{(l-1)} \mathbf{W}_{xh}^{(l)} + \mathbf{H}_{t-1}^{(l)} \mathbf{W}_{hh}^{(l)} + \mathbf{b}_h^{(l)}),$$

其中，权重$\mathbf{W}_{xh}^{(l)} \in \mathbb{R}^{h \times h}$， $\mathbf{W}_{hh}^{(l)} \in \mathbb{R}^{h \times h}$和 偏置$\mathbf{b}_h^{(l)} \in \mathbb{R}^{1 \times h}$ 都是第l个隐藏层的模型参数。

最后，输出层的计算仅基于第l个隐藏层最终的隐状态：

$$\mathbf{O}_t = \mathbf{H}_t^{(L)} \mathbf{W}_{hq} + \mathbf{b}_q,$$

其中，权重$\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$和偏置$\mathbf{b}_q \in \mathbb{R}^{1 \times q}$都是输出层的模型参数。

与多层感知机一样，隐藏层数目L和隐藏单元数目h都是超参数。 也就是说，它们可以由我们调整的。 另外，用门控循环单元或长短期记忆网络的隐状态 来代替

$$\mathbf{H}_t^{(l)} = \phi_l(\mathbf{H}_t^{(l-1)} \mathbf{W}_{xh}^{(l)} + \mathbf{H}_{t-1}^{(l)} \mathbf{W}_{hh}^{(l)} + \mathbf{b}_h^{(l)}),$$

中的隐状态进行计算， 可以很容易地得到深度门控循环神经网络或深度长短期记忆神经网络。

简洁实现

实现多层循环神经网络所需的许多逻辑细节在高级API中都是现成的。

简单起见，仅示范使用此类内置函数的实现方式。 以长短期记忆网络模型为例， 该代码与之前在 LSTM 中使用的代码非常相似， 实际上唯一的区别是我们指定了层的数量， 而不是使用单一层这个默认值。

像选择超参数这类架构决策也跟 LSTM 中的决策非常相似。 因为有不同的词元，所以输入和输出都选择相同数量，即 `vocab_size` 。 隐藏单元的数量仍然是256。 唯一的区别是，现在**通过** `num_layers` **的值来设定隐藏层数。**

In [12]:
```
vocab_size, num_hiddens, num_layers = len(vocab), 256, 2
num_inputs = vocab_size
device = d2l.try_gpu()
lstm_layer = nn.LSTM(num_inputs, num_hiddens, num_layers)
model = d2l.RNNModel(lstm_layer, len(vocab))
model = model.to(device)
```

训练与预测

由于使用了长短期记忆网络模型来实例化两个层，因此训练速度被大大降低了。

In [13]:
```
num_epochs, lr = 500, 2
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.0, 183966.1 tokens/sec on cuda:0
time travelleryou can show black is white by argument said filby
travelleryou can show black is white by argument said filby

<Figure size 252x180 with 1 Axes>
```

4. 双向循环神经网络

在序列学习中，我们以往假设的目标是： 在给定观测的情况下 （例如，在时间序列的上下文中或在语言模型的上下文中）， 对下一个输出进行建模。 虽然这是一个典型情景，但不是唯一的。 还可能发生什么其它的情况呢？ 考虑以下三个在文本序列中填空的任务：

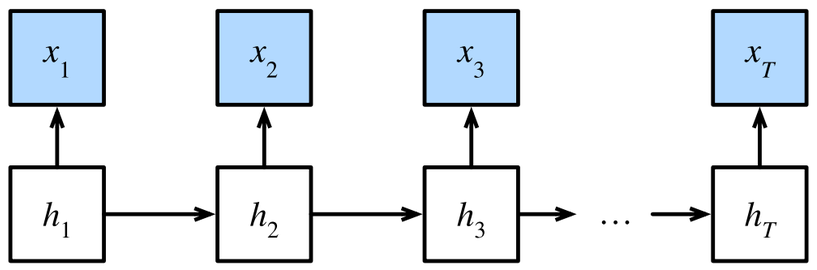
- 我 ___ 。
- 我 ___ 饿了。
- 我 ___ 饿了，我可以吃半头猪。

根据可获得的信息量，我们可以用不同的词填空， 如“很高兴”（"happy"）、“不”（"not"）和“非常”（"very"）。 很明显，每个短语的“下文”传达了重要信息（如果有的话）， 而这些信息关乎到选择哪个词来填空， 所以无法利用这一点的序列模型将在相关任务上表现不佳。 例如，如果要做好命名实体识别 （例如，识别“Green”指的是“格林先生”还是绿色）， 不同长度的上下文范围重要性是相同的。 为了获得一些解决问题的灵感，让我们先迂回到概率图模型。

隐马尔可夫模型中的动态规划

这一小节是用来说明动态规划问题的， 具体的技术细节对于理解深度学习模型并不重要， 但它有助于思考为什么要使用深度学习， 以及为什么要选择特定的架构。

如果想用概率图模型来解决这个问题， 可以设计一个隐变量模型： 在任意时间步t，假设存在某个隐变量h_t， 通过概率$P(x_t \mid h_t)$控制我们观测到的x_t。 此外，任何$h_t \rightarrow h_{t+1}$转移 都是由一些状态转移概率$P(h_{t+1} \mid h_t)$给出。 这个概率图模型就是一个*隐马尔可夫模型*（hidden Markov model，HMM），如：



因此，对于有T个观测值的序列， 我们在观测状态和隐状态上具有以下联合概率分布：

$$P(x_1, \ldots, x_T, h_1, \ldots, h_T) = \prod_{t=1}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t), \text{ where } P(h_1 \mid h_0) = P(h_1).$$

现在，假设观测到所有的x_i，除了x_j， 并且目标是计算$P(x_j \mid x_{-j})$， 其中$x_{-j} = (x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_T)$。 由于$P(x_j \mid x_{-j})$中没有隐变量， 因此考虑对$h_1, \ldots, h_T$选择构成的 所有可能的组合进行求和。 如果任何$h_i$可以接受$k$个不同的值（有限的状态数）， 这意味着需要对$k^T$个项求和， 这个任务显然难于登天。 幸运的是，有个巧妙的解决方案：*动态规划*（dynamic programming）。

要了解动态规划的工作方式， 考虑对隐变量$h_1, \ldots, h_T$的依次求和。 根据

$$P(x_1, \ldots, x_T, h_1, \ldots, h_T) = \prod_{t=1}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t), \text{ where } P(h_1 \mid h_0) = P(h_1).$$

，将得出：

$$
\begin{aligned}
&P(x_1, \ldots, x_T) \\
&= \sum_{h_1, \ldots, h_T} P(x_1, \ldots, x_T, h_1, \ldots, h_T) \\
&= \sum_{h_1, \ldots, h_T} \prod_{t=1}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \\
&= \sum_{h_2, \ldots, h_T} \underbrace{\left[\sum_{h_1} P(h_1) P(x_1 \mid h_1) P(h_2 \mid h_1)\right]}_{\pi_2(h_2) \stackrel{\text{def}}{=}} P(x_2 \mid h_2) \prod_{t=3}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \\
&= \sum_{h_3, \ldots, h_T} \underbrace{\left[\sum_{h_2} \pi_2(h_2) P(x_2 \mid h_2) P(h_3 \mid h_2)\right]}_{\pi_3(h_3) \stackrel{\text{def}}{=}} P(x_3 \mid h_3) \prod_{t=4}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \\
&= \ldots \\
&= \sum_{h_T} \pi_T(h_T) P(x_T \mid h_T).
\end{aligned}
$$

通常将“前向递归”（forward recursion）写为：

$$
\pi_{t+1}(h_{t+1}) = \sum_{h_t} \pi_t(h_t) P(x_t \mid h_t) P(h_{t+1} \mid h_t).
$$

递归被初始化为$\pi_1(h_1) = P(h_1)$。 符号简化，也可以写成$\pi_{t+1} = f(\pi_t, x_t)$， 其中$f$是一些可学习的函数。 这看起来就像循环神经网络中的隐变量模型中的更新方程。

与前向递归一样，我们也可以使用后向递归对同一组隐变量求和。这将得到：

$$
\begin{aligned}
&P(x_1, \ldots, x_T) \\
&= \sum_{h_1, \ldots, h_T} P(x_1, \ldots, x_T, h_1, \ldots, h_T) \\
&= \sum_{h_1, \ldots, h_T} \prod_{t=1}^{T-1} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \cdot P(h_T \mid h_{T-1}) P(x_T \mid h_T) \\
&= \sum_{h_1, \ldots, h_{T-1}} \prod_{t=1}^{T-1} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \cdot \underbrace{\left[\sum_{h_T} P(h_T \mid h_{T-1}) P(x_T \mid h_T)\right]}_{\rho_{T-1}(h_{T-1}) \stackrel{\text{def}}{=}} \\
&= \sum_{h_1, \ldots, h_{T-2}} \prod_{t=1}^{T-2} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \cdot \underbrace{\left[\sum_{h_{T-1}} P(h_{T-1} \mid h_{T-2}) P(x_{T-1} \mid h_{T-1}) \rho_{T-1}(h_{T-1})\right]}_{\rho_{T-2}(h_{T-2}) \stackrel{\text{def}}{=}} \\
&= \ldots \\
&= \sum_{h_1} P(h_1) P(x_1 \mid h_1) \rho_1(h_1).
\end{aligned}
$$

因此可以将“后向递归”（backward recursion）写为：

$$
\rho_{t-1}(h_{t-1}) = \sum_{h_t} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \rho_t(h_t),
$$

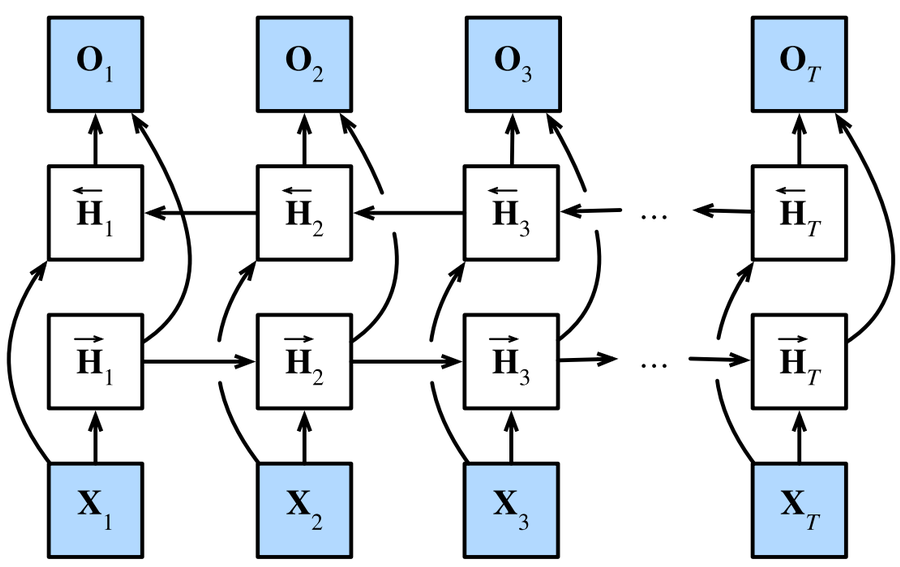
初始化$\rho_T(h_T) = 1$。 前向和后向递归都允许我们对T个隐变量在$\mathcal{O}(kT)$ （线性而不是指数）时间内对$(h_1, \ldots, h_T)$的所有值求和。 这是使用图模型进行概率推理的巨大好处之一。 它也是通用消息传递算法的一个非常特殊的例子。 结合前向和后向递归，我们能够计算

$$P(x_j \mid x_{-j}) \propto \sum_{h_j} \pi_j(h_j)\rho_j(h_j)P(x_j \mid h_j).$$

因为符号简化的需要，后向递归也可以写为$\rho_{t-1} = g(\rho_t, x_t)$， 其中$g$是一个可以学习的函数。 同样，这看起来非常像一个更新方程， 只是不像我们在循环神经网络中看到的那样前向运算，而是后向计算。 事实上，知道未来数据何时可用对隐马尔可夫模型是有益的。 信号处理学家将是否知道未来观测这两种情况区分为内插和外推。

双向模型

如果希望在循环神经网络中拥有一种机制， 使之能够提供与隐马尔可夫模型类似的前瞻能力， 就需要修改循环神经网络的设计。 幸运的是，这在概念上很容易， 只需要增加一个“从最后一个词元开始从后向前运行”的循环神经网络， 而不是只有一个在前向模式下“从第一个词元开始运行”的循环神经网络。 *双向循环神经网络*（bidirectional RNNs） 添加了反向传递信息的隐藏层，以便更灵活地处理此类信息。 下图描述了具有单个隐藏层的双向循环神经网络的架构。



事实上，这与隐马尔可夫模型中的动态规划的前向和后向递归没有太大区别。 其主要区别是，在隐马尔可夫模型中的方程具有特定的统计意义。 双向循环神经网络没有这样容易理解的解释， 我们只能把它们当作通用的、可学习的函数。 这种转变集中体现了现代深度网络的设计原则： 首先使用经典统计模型的函数依赖类型，然后将其参数化为通用形式。

定义

双向循环神经网络是由 Schuster.Paliwal 提出的。 下面看看这样一个网络的细节。

对于任意时间步t，给定一个小批量的输入数据 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ （样本数：n，每个示例中的输入数：d）， 并且令隐藏层激活函数为ϕ。 在双向架构中，我们设该时间步的前向和反向隐状态分别为 $\overrightarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$和 $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$， 其中$h$是隐藏单元的数目。 前向和反向隐状态的更新如下：

$$\begin{aligned}\overrightarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}),\\ \overleftarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}),\end{aligned}$$

其中，权重$\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{d \times h}, \mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h \times h}, \mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d \times h}, \mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h \times h}$ 和偏置$\mathbf{b}_h^{(f)} \in \mathbb{R}^{1 \times h}, \mathbf{b}_h^{(b)} \in \mathbb{R}^{1 \times h}$都是模型参数。

接下来，将前向隐状态$\overrightarrow{\mathbf{H}}_t$ 和反向隐状态$\overleftarrow{\mathbf{H}}_t$连接起来， 获得需要送入输出层的隐状态$\mathbf{H}_t \in \mathbb{R}^{n \times 2h}$。 在具有多个隐藏层的深度双向循环神经网络中， 该信息作为输入传递到下一个双向层。 最后，输出层计算得到的输出为 $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ （q是输出单元的数目）：

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q.$$

这里，权重矩阵$\mathbf{W}_{hq} \in \mathbb{R}^{2h \times q}$ 和偏置$\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ 是输出层的模型参数。 事实上，这两个方向可以拥有不同数量的隐藏单元。

模型的计算代价及其应用

双向循环神经网络的一个关键特性是：使用来自序列两端的信息来估计输出。 也就是说，我们使用来自过去和未来的观测信息来预测当前的观测。 但是在对下一个词元进行预测的情况中，这样的模型并不是我们所需的。 因为在预测下一个词元时无法知道下一个词元的下文是什么， 所以将不会得到很好的精度。 具体地说，在训练期间能够利用过去和未来的数据来估计现在空缺的词； 而在测试期间，只有过去的数据，因此精度将会很差。 下面的实验将说明这一点。

另一个严重问题是，双向循环神经网络的计算速度非常慢。 其主要原因是网络的前向传播需要在双向层中进行前向和后向递归， 并且网络的反向传播还依赖于前向传播的结果。 因此，梯度求解将有一个非常长的链。

双向层的使用在实践中非常少，并且仅仅应用于部分场合。 例如，填充缺失的单词、词元注释（例如，用于命名实体识别） 以及作为序列处理流水线中的一个步骤对序列进行编码（例如，用于机器翻译）。

双向循环神经网络的错误应用

由于双向循环神经网络使用了过去的和未来的数据， 所以不能盲目地将这一语言模型应用于任何预测任务。 尽管模型产出的困惑度是合理的， 该模型预测未来词元的能力却可能存在严重缺陷。 以下面的示例代码引以为戒，以防在错误的环境中使用它们。

In [15]:
```
# 通过设置“bidirective=True”来定义双向LSTM模型
vocab_size, num_hiddens, num_layers = len(vocab), 256, 2
num_inputs = vocab_size
lstm_layer = nn.LSTM(num_inputs, num_hiddens, num_layers, bidirectional=True)
model = d2l.RNNModel(lstm_layer, len(vocab))
model = model.to(device)
# 训练模型
num_epochs, lr = 500, 1
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 118288.1 tokens/sec on cuda:0
time travellererererererererererererererererererererererererererer
travellererererererererererererererererererererererererererer

<Figure size 252x180 with 1 Axes>
```

上述结果显然令人瞠目结舌。