

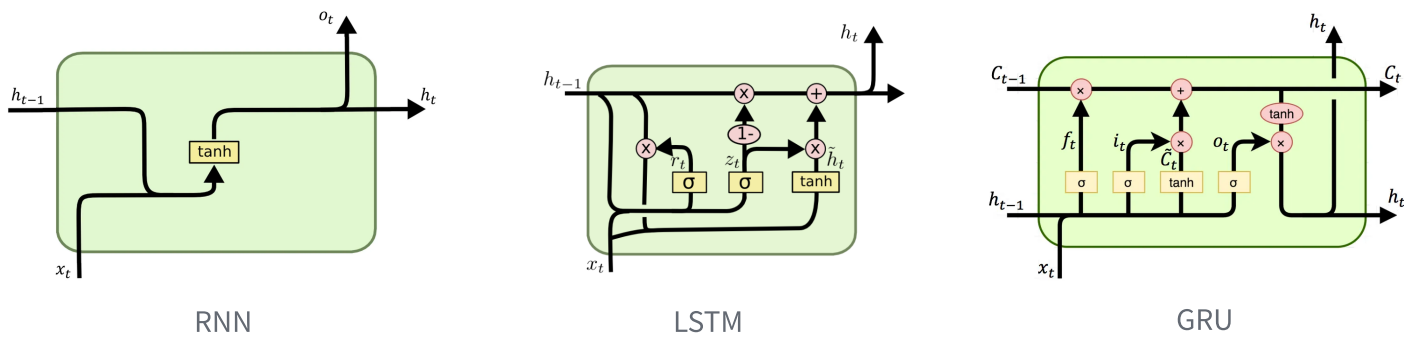
5.3循环神经网络-GRU

GRU结构

GRU是在2014年提出来的，而LSTM是1997年。

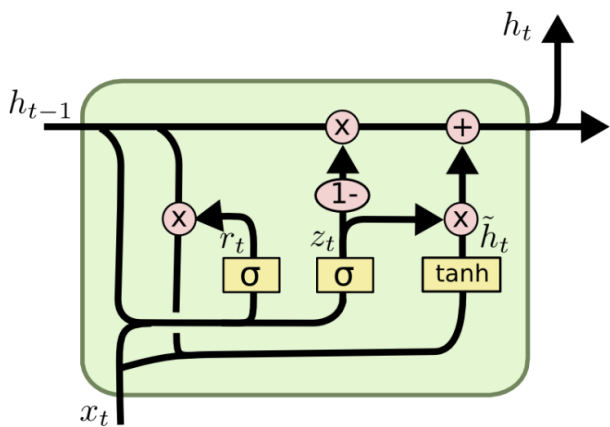
GRU类似LSTM，也是为了解决RNN长期记忆的梯度消失问题

LSTM有三个不同的门，参数较多，训练困难。GRU只含有两个门控结构，调优后相比LSTM效果相差无几，且结构简单，更容易训练，所以很多时候会更倾向于使用GRU。



GRU在LSTM的基础上主要做出了两点改变：

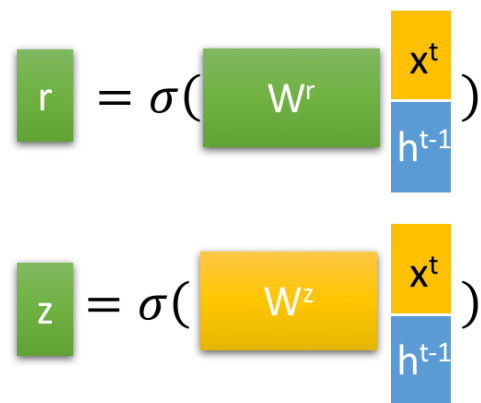
- (1) GRU只有两个门。GRU将LSTM中的输入门和遗忘门合二为一，称为更新门（update gate），控制前边记忆信息能够继续保留到当前时刻的数据量；另一个门称为重置门（reset gate），控制要遗忘多少过去的信息。
- (2) 取消进行线性自更新的记忆单元（memory cell），而是直接在隐藏单元中利用门控直接进行线性自更新。GRU的逻辑图如图所示：



$$\begin{aligned} z_t &= \sigma(W_z \cdot [h_{t-1}, x_t]) \\ r_t &= \sigma(W_r \cdot [h_{t-1}, x_t]) \\ \tilde{h}_t &= \tanh(W \cdot [r_t * h_{t-1}, x_t]) \\ h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t \end{aligned}$$

我们先通过上一个传输下来的状态 h^{t-1} 和当前节点的输入 x_t 来获取两个门控状态。如下图2-2所示，其中 r 控制重置的门控（reset gate），z 为控制更新的门控（update gate）。

σ 为sigmoid函数，通过这个函数可以将数据变换为0-1范围内的数值，从而来充当门控信号。


$$r = \sigma(W^r \begin{bmatrix} x^t \\ h^{t-1} \end{bmatrix})$$
$$z = \sigma(W^z \begin{bmatrix} x^t \\ h^{t-1} \end{bmatrix})$$

GRU与LSTM的异同

1. 共同点

- 克服长程依赖问题：通过引入门控机制来控制信息流动。
- 适用于序列数据：如时间序列、自然语言处理、音频数据等。
- 网络结构：都包含隐藏状态，能够将序列信息编码并传递到后续时间步。

2. 不同点

特性	LSTM	GRU
门的数量	3个门（输入门、遗忘门、输出门）	2个门（重置门、更新门）
单元状态	包含隐藏状态和单元状态（cell state），双状态结构	只有隐藏状态，单状态结构
参数量	参数较多，计算开销较大	参数较少，计算开销较小
记忆能力	能够更精细地控制信息的遗忘和记忆	结构较简单，可能会表现出较快的训练和预测
表达能力	更灵活，能处理更复杂的序列模式	在某些情况下，效果接近甚至优于LSTM，但计算效率高

在不同场景的选择

1. LSTM适用场景

- 需要捕获复杂长程依赖的任务：如机器翻译、长时间依赖的时间序列预测。

- **对模型表现要求较高**：特别是在**小数据集**上训练时，LSTM的复杂性可能有助于更好地拟合数据。
- **需要更精确的记忆控制**：如医疗时间序列分析（复杂的动态模式）。

2. GRU适用场景

- **计算资源有限**：由于GRU参数较少，训练和推理速度更快，适合资源受限的场景。
- **数据集规模较大**：大数据集上，GRU的简单结构能更快地收敛，且性能与LSTM相当。
- **应用场景对长期依赖的要求不太苛刻**：如简单的时间序列预测、情感分析等。

3. 实验驱动的选择

对于具体问题，通常可以尝试以下步骤：

1. **从GRU开始**：由于其计算效率高，适合快速迭代。
2. **尝试LSTM**：如果GRU表现不佳，可以尝试LSTM，特别是在需要更强记忆能力时。
3. **性能对比**：根据模型在验证集或测试集上的表现（如损失、准确率、F1分数等）来决定最终使用哪种模型。

动手实践GRU

从0实现

```
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4
5 # 数据示例
6 torch.manual_seed(0)
7 x_train = torch.randn(10, 4, 3) # 10个样本，每个样本有4个时间步，每个时间步3个特征
8 y_train = torch.randn(10, 1)   # 10个目标值，回归任务
9
10 # 自定义GRU模型
11 class CustomGRU(nn.Module):
12     def __init__(self, input_size, hidden_size):
13         super(CustomGRU, self).__init__()
14         self.input_size = input_size
15         self.hidden_size = hidden_size
16
17         # Update gate parameters
18         self.W_z = nn.Linear(input_size + hidden_size, hidden_size)
19
20         # Reset gate parameters
21         self.W_r = nn.Linear(input_size + hidden_size, hidden_size)
```

```

22
23     # Candidate hidden state parameters
24     self.W_h = nn.Linear(input_size + hidden_size, hidden_size)
25
26     # Output layer
27     self.fc = nn.Linear(hidden_size, 1) # 用于回归输出
28
29     def forward(self, x, hidden):
30         for t in range(x.size(1)): # 遍历每个时间步
31             combined = torch.cat((x[:, t, :], hidden), dim=1)
32
33             # Update gate
34             z_t = torch.sigmoid(self.W_z(combined))
35
36             # Reset gate
37             r_t = torch.sigmoid(self.W_r(combined))
38
39             # Candidate hidden state
40             combined_candidate = torch.cat((x[:, t, :], r_t * hidden), dim=1)
41             h_tilde = torch.tanh(self.W_h(combined_candidate))
42
43             # Final hidden state
44             hidden = (1 - z_t) * hidden + z_t * h_tilde
45
46             # 输出最后时间步的隐藏状态，通过全连接层
47             output = self.fc(hidden)
48             return output, hidden
49
50 # 超参数
51 input_size = 3
52 hidden_size = 5
53 num_epochs = 100
54 learning_rate = 0.01
55
56 # 模型、损失函数和优化器
57 model = CustomGRU(input_size, hidden_size)
58 criterion = nn.MSELoss()
59 optimizer = optim.Adam(model.parameters(), lr=learning_rate)
60
61 # 训练
62 for epoch in range(num_epochs):
63     hidden = torch.zeros(x_train.size(0), hidden_size) # 初始化隐藏状态
64     outputs, hidden = model(x_train, hidden)
65
66     loss = criterion(outputs, y_train)
67
68     optimizer.zero_grad()

```

```

69     loss.backward()
70     optimizer.step()
71
72     if (epoch+1) % 10 == 0:
73         print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
74
75 # 测试输出
76 print("\nFinal output after training:", outputs)
77

```

- 输出示例

```

1 Epoch [10/100], Loss: 0.8517
2 Epoch [20/100], Loss: 0.6867
3 Epoch [30/100], Loss: 0.4487
4 Epoch [40/100], Loss: 0.2006
5 Epoch [50/100], Loss: 0.0579
6 Epoch [60/100], Loss: 0.0098
7 Epoch [70/100], Loss: 0.0067
8 Epoch [80/100], Loss: 0.0049
9 Epoch [90/100], Loss: 0.0029
10 Epoch [100/100], Loss: 0.0015
11 Final output after training: tensor([[ 1.2980],
12         [ 1.2390],
13         [ 0.4441],
14         [-1.7230],
15         [-1.3356],
16         [ 0.8752],
17         [ 0.7882],
18         [ 0.0545],
19         [ 0.2110],
20         [ 0.5518]], grad_fn=<AddmmBackward0>)

```

简易实现（直接使用torch内部GRU）

```

1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4
5 # 模拟数据
6 torch.manual_seed(0)
7 x_train = torch.randn(10, 4, 3) # 10个样本，每个样本有4个时间步，每个时间步3个特征
8 y_train = torch.randn(10, 1)   # 10个目标值

```

```

9
10 # 定义一个简单的GRU模型
11 class SimpleGRU(nn.Module):
12     def __init__(self, input_size, hidden_size, output_size):
13         super(SimpleGRU, self).__init__()
14         self.gru = nn.GRU(input_size, hidden_size, num_layers=1,
15 batch_first=True)
16         self.fc = nn.Linear(hidden_size, output_size) # 用最后时间步的隐藏状态预测
17 # 输出
18
19     def forward(self, x):
20         # 初始化隐藏状态
21         h0 = torch.zeros(1, x.size(0), hidden_size) # num_layers=1, (1,
22 batch_size, hidden_size)
23
24         # GRU前向传播
25         out, hn = self.gru(x, h0) # out包含所有时间步的输出, hn是最后时间步的隐藏状
26 # 态
27
28         # 使用最后时间步的隐藏状态进行预测
29         out = self.fc(hn[-1]) # hn[-1] 取最后一层的隐藏状态
30         return out
31
32 # 超参数
33 input_size = 3 # 输入特征维度
34 hidden_size = 5 # GRU隐藏层大小
35 output_size = 1 # 输出维度
36 num_epochs = 100 # 训练轮数
37 learning_rate = 0.01
38
39 # 实例化模型、损失函数和优化器
40 model = SimpleGRU(input_size, hidden_size, output_size)
41 criterion = nn.MSELoss()
42 optimizer = optim.Adam(model.parameters(), lr=learning_rate)
43
44 # 训练模型
45 for epoch in range(num_epochs):
46     outputs = model(x_train)
47     loss = criterion(outputs, y_train)
48
49     optimizer.zero_grad()
50     loss.backward()
51     optimizer.step()
52
53     if (epoch+1) % 10 == 0:
54         print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
55
56

```

```
52 # 测试模型
53 print("\nFinal output after training:", outputs)
54
```

- 输出示例

```
1 Epoch [10/100], Loss: 0.8123
2 Epoch [20/100], Loss: 0.6035
3 Epoch [30/100], Loss: 0.4432
4 ...
5 Epoch [100/100], Loss: 0.0256
6
7 Final output after training: tensor([[ 0.1573],
8                                     [-0.2045],
9                                     [ 0.4526],
10                                    [ 0.0789],
11                                    ...
12                                    [ 0.1023]]), grad_fn=<AddmmBackward0>)
13
```