# 实验：RNN 的简单应用

## 一、 实验内容

借助 Torchtext 实现一个简单的 RNN 网络,并在斯坦福 IMDB 数据集上进行 训练。

## 二、 实验目标

（1） 掌握 RNN 基本网络结构和其工作原理。

（2） 了解 PyTorch 深度学习环境搭建。

（3） 运用 RNN 对数据集进行进行简单的训练，完成文本分类任务。

（4） 通过 IMDB 数据集的数据预处理过程，了解 Torchtext的设计思路和细节。

## 三、 实验内容

#### 3.1简单RNN实验

##### 3.1.1.数据集准备

使用torchtext.datasets 下载数据集

遇到个问题就是spacy 超时无法下载，经过排查 通过下述命令可解决

spacy 无法下载 解决办法

pip --default-timeout=10000 install https://github.com/explosion/spacy-models/releases/download/en\_core\_web\_sm-2.3.0/en\_core\_web\_sm-2.3.0.tar.gz

|  |
| --- |
| # 分词器 SEED = 1234 torch.manual\_seed(SEED) torch.backends.cudnn.deterministic = True TEXT = data.Field(tokenize='spacy', tokenizer\_language='en\_core\_web\_sm') LABEL = data.LabelField(dtype=torch.float)  # 切分数据集 train\_data, test\_data = datasets.IMDB.splits(TEXT, LABEL)  # 构建词向量 MAX\_VOCAB\_SIZE = 25\_000 TEXT.build\_vocab(train\_data, max\_size=MAX\_VOCAB\_SIZE) LABEL.build\_vocab(train\_data)  # 整理数据集 BATCH\_SIZE = 64 device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu') print(device) train\_iterator, test\_iterator = data.BucketIterator.splits((train\_data, test\_data), batch\_size=BATCH\_SIZE, device=device) |

##### 3.1.2RNN网络定义

定义RNN

|  |
| --- |
| **class** RNN(nn**.**Module):  **def** \_\_init\_\_(self, input\_dim, embedding\_dim, hidden\_dim, output\_dim):  super()**.**\_\_init\_\_()  self**.**embedding **=** nn**.**Embedding(input\_dim, embedding\_dim)  self**.**rnn **=** nn**.**RNN(embedding\_dim, hidden\_dim)  self**.**fc **=** nn**.**Linear(hidden\_dim, output\_dim)    **def** forward(self, text):  embedded **=** self**.**embedding(text)  output, hidden **=** self**.**rnn(embedded)  **assert** torch**.**equal(output[**-**1,:,:], hidden**.**squeeze(0))  **return** self**.**fc(hidden**.**squeeze(0)) |

##### 3.1.3定义维度损失函数/优化器

采用二元交叉熵损失函数

使用实验要求中的SGD优化器

|  |
| --- |
| # 定义维度 INPUT\_DIM = len(TEXT.vocab) EMBEDDING\_DIM = 100 HIDDEN\_DIM = 256 OUTPUT\_DIM = 1  model = RNN(INPUT\_DIM, EMBEDDING\_DIM, HIDDEN\_DIM, OUTPUT\_DIM) # model = nn.RNN(INPUT\_DIM, EMBEDDING\_DIM, HIDDEN\_DIM, OUTPUT\_DIM) model = model.to(device)  # 计算二元交叉熵 criterion = nn.BCEWithLogitsLoss() criterion = criterion.to(device)  # 优化器 optimizer = optim.SGD(model.parameters(), lr=1e-3, momentum=0.9) |

##### 3.1.4定义训练验证函数

|  |
| --- |
| # 训练方法 def train(iterator):  epoch\_loss = 0  model.train()   for batch in iterator:  optimizer.zero\_grad()  # batch.text 就是上面forward函数的参数text，压缩维度是为了和batch.label维度一致  predictions = model(batch.text).squeeze(1)  loss = criterion(predictions, batch.label)  loss.backward()  optimizer.step()   epoch\_loss += loss.item()   return epoch\_loss / len(iterator) |

##### 3.1.5定义评估函数

|  |
| --- |
| def evaluate(iterator):  epoch\_acc = 0  epoch\_loss = 0  model.eval()  with torch.no\_grad():  for batch in iterator:  predictions = model(batch.text).squeeze(1)  loss = criterion(predictions, batch.label)  acc = binary\_accuracy(predictions, batch.label)  epoch\_acc += acc.item()  epoch\_loss += loss.item()  return epoch\_acc / len(iterator),epoch\_loss / len(iterator) |

##### 3.1.6定义工具函数

包含计算准确率及消耗时间函数

|  |
| --- |
| # 计算 accuracy def binary\_accuracy(preds, y):  rounded\_preds = torch.round(torch.sigmoid(preds))  correct = (rounded\_preds == y).float() # convert into float for division  acc = correct.sum() / len(correct)  return acc  # 计算消耗时间 def epoch\_time(start\_time, end\_time):  elapsed\_time = end\_time - start\_time  elapsed\_mins = int(elapsed\_time / 60)  elapsed\_secs = int(elapsed\_time - (elapsed\_mins \* 60))  return elapsed\_mins, elapsed\_secs |

##### 3.1.7执行函数

定义epoch为30的测试函数，并保存模型

|  |
| --- |
| # 测试 for epoch in range(30):  start\_time = time.time()  train\_loss = train(train\_iterator)   end\_time = time.time()   epoch\_mins, epoch\_secs = epoch\_time(start\_time, end\_time)  torch.save(model.state\_dict(), 'jn\_rnn')   print(  'Epoch: %d |train loss: %.3f |cost: %d m %d s' % (  epoch + 1, train\_loss, epoch\_mins, epoch\_secs))   test\_acc,test\_loss = evaluate(test\_iterator)  print(  'Epoch: %d |evaluate loss: %.2f |evaluate accuracy: %.2f' % (epoch + 1, test\_loss, test\_acc \* 100)) |

##### 3.1.6RNN执行效果

从执行结果可以看到，一直没有什么具体效果变化

|  |
| --- |
| cuda  Epoch: 1 |train loss: 0.695 |cost: 0 m 20 s  Epoch: 1 |evaluate accuracy: 63.94  Epoch: 2 |train loss: 0.695 |cost: 0 m 20 s  Epoch: 2 |evaluate accuracy: 45.04  Epoch: 3 |train loss: 0.695 |cost: 0 m 20 s  Epoch: 3 |evaluate accuracy: 65.63  …  Epoch: 32 |train loss: 1.876 |cost: 0 m 20 s  Epoch: 32 |evaluate accuracy: 50.06  …  Epoch: 40 |train loss: 1.994 |cost: 0 m 20 s  Epoch: 40 |evaluate accuracy: 50.35  …  Epoch: 66 |train loss: 1.820 |cost: 0 m 20 s  Epoch: 66 |evaluate accuracy: 50.37  …  Epoch: 312 |train loss: 1.973 |cost: 0 m 20 s  Epoch: 312 |evaluate accuracy: 51.64  Epoch: 313 |train loss: 1.937 |cost: 0 m 20 s  Epoch: 313 |evaluate accuracy: 51.71 |

##### 3.1.7修改网络为LSTM

|  |
| --- |
| # 定义RNN class RNN(nn.Module):  def \_\_init\_\_(self, imput\_dim, embedding\_dim, hidden\_dim, output\_dim):  super(RNN, self).\_\_init\_\_()   self.embedding = nn.Embedding(imput\_dim, embedding\_dim)  self.rnn = nn.LSTM(embedding\_dim, hidden\_dim, num\_layers=2,  bidirectional=True, dropout=0.5)  self.fc = nn.Linear(hidden\_dim \* 2, output\_dim)  self.dropout = nn.Dropout(0.5)   def forward(self, x):  embedding = self.dropout(self.embedding(x))   output, (hidden, cell) = self.rnn(embedding)  hidden = torch.cat([hidden[-2], hidden[-1]], dim=1)  hidden = self.dropout(hidden)  out = self.fc(hidden)   return out |

##### 3.1.8LSTM执行效果

|  |
| --- |
| cuda  Epoch: 1 |train loss: 0.693 |cost: 3 m 7 s  Epoch: 1 |evaluate loss: 0.69 |evaluate accuracy: 52.37  Epoch: 2 |train loss: 0.692 |cost: 3 m 6 s  Epoch: 2 |evaluate loss: 0.69 |evaluate accuracy: 50.19  Epoch: 3 |train loss: 0.691 |cost: 3 m 8 s  Epoch: 3 |evaluate loss: 0.69 |evaluate accuracy: 53.50  Epoch: 4 |train loss: 0.689 |cost: 3 m 9 s  Epoch: 4 |evaluate loss: 0.69 |evaluate accuracy: 54.44  Epoch: 5 |train loss: 0.687 |cost: 3 m 10 s  Epoch: 5 |evaluate loss: 0.68 |evaluate accuracy: 55.98  Epoch: 6 |train loss: 0.685 |cost: 3 m 8 s  Epoch: 6 |evaluate loss: 0.68 |evaluate accuracy: 56.50  Epoch: 7 |train loss: 0.684 |cost: 3 m 8 s  Epoch: 7 |evaluate loss: 0.68 |evaluate accuracy: 56.88  Epoch: 8 |train loss: 0.683 |cost: 3 m 7 s  Epoch: 8 |evaluate loss: 0.68 |evaluate accuracy: 57.90  Epoch: 9 |train loss: 0.680 |cost: 3 m 7 s  Epoch: 9 |evaluate loss: 0.67 |evaluate accuracy: 57.67  Epoch: 10 |train loss: 0.677 |cost: 3 m 8 s  Epoch: 10 |evaluate loss: 0.67 |evaluate accuracy: 58.84  Epoch: 11 |train loss: 0.675 |cost: 3 m 11 s  Epoch: 11 |evaluate loss: 0.67 |evaluate accuracy: 58.43  Epoch: 12 |train loss: 0.665 |cost: 3 m 9 s  Epoch: 12 |evaluate loss: 0.66 |evaluate accuracy: 60.48  Epoch: 13 |train loss: 0.670 |cost: 3 m 6 s  Epoch: 13 |evaluate loss: 0.72 |evaluate accuracy: 51.78  Epoch: 14 |train loss: 0.679 |cost: 3 m 8 s  Epoch: 14 |evaluate loss: 0.63 |evaluate accuracy: 64.85  Epoch: 15 |train loss: 0.670 |cost: 3 m 9 s  Epoch: 15 |evaluate loss: 0.64 |evaluate accuracy: 63.84  Epoch: 16 |train loss: 0.685 |cost: 3 m 5 s  Epoch: 16 |evaluate loss: 0.68 |evaluate accuracy: 55.58  Epoch: 17 |train loss: 0.687 |cost: 3 m 1 s  Epoch: 17 |evaluate loss: 0.68 |evaluate accuracy: 56.04  Epoch: 18 |train loss: 0.680 |cost: 3 m 3 s  Epoch: 18 |evaluate loss: 0.67 |evaluate accuracy: 58.81  Epoch: 19 |train loss: 0.674 |cost: 3 m 3 s  Epoch: 19 |evaluate loss: 0.69 |evaluate accuracy: 55.71  Epoch: 20 |train loss: 0.671 |cost: 3 m 0 s  Epoch: 20 |evaluate loss: 0.68 |evaluate accuracy: 57.55 |

##### 3.19修改优化器后执行效果

修改SGD优化器为 adam

optimizer = optim.Adam(model.parameters(), lr=1e-3)

效果有明显提升

|  |
| --- |
| cuda  Epoch: 1 |train loss: 0.696 |cost: 3 m 3 s  Epoch: 1 |evaluate loss: 0.69 |evaluate accuracy: 50.17  Epoch: 2 |train loss: 0.691 |cost: 3 m 2 s  Epoch: 2 |evaluate loss: 0.70 |evaluate accuracy: 49.52  Epoch: 3 |train loss: 0.683 |cost: 3 m 3 s  Epoch: 3 |evaluate loss: 0.67 |evaluate accuracy: 59.92  Epoch: 4 |train loss: 0.637 |cost: 3 m 3 s  Epoch: 4 |evaluate loss: 0.51 |evaluate accuracy: 75.38  Epoch: 5 |train loss: 0.521 |cost: 3 m 6 s  Epoch: 5 |evaluate loss: 0.46 |evaluate accuracy: 78.22  Epoch: 6 |train loss: 0.412 |cost: 3 m 5 s  Epoch: 6 |evaluate loss: 0.39 |evaluate accuracy: 83.14  Epoch: 7 |train loss: 0.348 |cost: 3 m 3 s  Epoch: 7 |evaluate loss: 0.34 |evaluate accuracy: 86.11  Epoch: 8 |train loss: 0.314 |cost: 3 m 3 s  Epoch: 8 |evaluate loss: 0.36 |evaluate accuracy: 84.65  Epoch: 9 |train loss: 0.283 |cost: 3 m 3 s  Epoch: 9 |evaluate loss: 0.30 |evaluate accuracy: 87.59  Epoch: 10 |train loss: 0.255 |cost: 3 m 3 s  Epoch: 10 |evaluate loss: 0.31 |evaluate accuracy: 88.39  Epoch: 11 |train loss: 0.241 |cost: 3 m 4 s  Epoch: 11 |evaluate loss: 0.30 |evaluate accuracy: 88.21  Epoch: 12 |train loss: 0.220 |cost: 3 m 4 s  Epoch: 12 |evaluate loss: 0.28 |evaluate accuracy: 89.26  Epoch: 13 |train loss: 0.209 |cost: 3 m 3 s  Epoch: 13 |evaluate loss: 0.34 |evaluate accuracy: 88.18  Epoch: 14 |train loss: 0.198 |cost: 3 m 4 s  Epoch: 14 |evaluate loss: 0.29 |evaluate accuracy: 89.48  Epoch: 15 |train loss: 0.179 |cost: 3 m 4 s  Epoch: 15 |evaluate loss: 0.30 |evaluate accuracy: 89.69  Epoch: 16 |train loss: 0.174 |cost: 3 m 2 s  Epoch: 16 |evaluate loss: 0.31 |evaluate accuracy: 88.85  Epoch: 17 |train loss: 0.164 |cost: 3 m 2 s  Epoch: 17 |evaluate loss: 0.32 |evaluate accuracy: 89.35  Epoch: 18 |train loss: 0.157 |cost: 3 m 3 s  Epoch: 18 |evaluate loss: 0.33 |evaluate accuracy: 89.43  Epoch: 19 |train loss: 0.142 |cost: 3 m 4 s  Epoch: 19 |evaluate loss: 0.34 |evaluate accuracy: 89.43  Epoch: 20 |train loss: 0.133 |cost: 3 m 4 s  Epoch: 20 |evaluate loss: 0.36 |evaluate accuracy: 89.16 |