

Recurrent Neural Network

李熙

July 29, 2023

1 理论学习

1.1 循环神经网络 RNN

1. 网络结构

RNN 拥有可以自反馈的结构。在网络中，还存在一个“延时器”来记录隐藏状态得最近一次或几次的活性值。

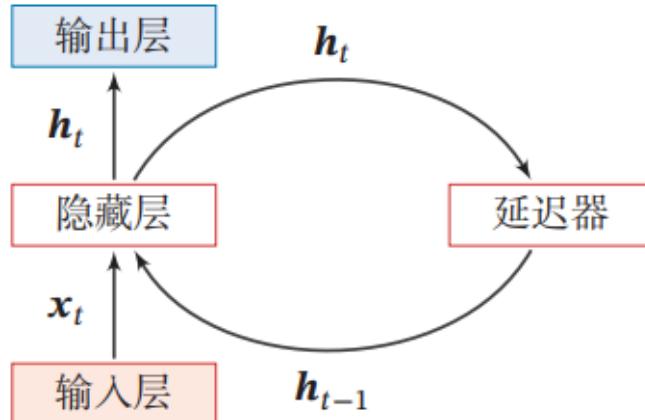


Figure 1: RNN 网络结构

2. RNN 的参数学习

RNN 的参数可以通过梯度下降方法来进行学习。

给定观察序列 (x, y) ，其中 $x_{1:T} = (x_1, \dots, x_T)$ 为长度是 T 的输入序列， $y_{1:T} = (y_1, \dots, y_T)$ 是对应的标签序列。定义时刻 t 的损失函数为

$$\mathcal{L}_t = \mathcal{L}(y_t, g(\mathbf{h}_t)) \quad (1)$$

定义整个序列的损失函数为

$$\mathcal{L} = \sum_{t=1}^T \mathcal{L}_t \quad (2)$$

则损失函数关于参数 U (隐藏状态的参数) 的梯度为

$$\frac{\partial \mathcal{L}}{\partial U} = \sum_{t=1}^T \frac{\partial \mathcal{L}_t}{\partial U} \quad (3)$$

在 RNN 中有两种计算梯度的方式，分别是随时间反向传播算法和实时循环学习法。

(a) 随时间反向传播算法 BPTT

首先每一层的净输入为 $z_k = Uh_{k-1} + Wx_k + b$ ，所以在 t 时刻的损失函数关于参数 u_{ij} 的梯度为：

$$\frac{\partial \mathcal{L}_t}{\partial u_{ij}} = \sum_{k=1}^t \frac{\partial^+ z_k}{\partial u_{ij}} \frac{\partial \mathcal{L}_t}{\partial z_k} \quad (4)$$

定义误差项 $\delta_{t,k} = \frac{\partial \mathcal{L}_t}{\partial z_k}$ ，再对 Eq.4 中做矩阵变换得到：

$$\frac{\partial \mathcal{L}_t}{\partial u_{ij}} = \sum_{k=1}^t [\delta_{t,k}]_i [h_{k-1}]_j \quad (5)$$

最后再写成矩阵形式：

$$\frac{\partial \mathcal{L}_t}{\partial U} = \sum_{k=1}^t \delta_{t,k} h_{k-1} \quad (6)$$

同理， \mathcal{L} 关于权重 W 和偏置 b 的梯度为

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial W} &= \sum_{t=1}^T \sum_{k=1}^t \delta_{t,k} x_k^\top \\ \frac{\partial \mathcal{L}}{\partial b} &= \sum_{t=1}^T \sum_{k=1}^t \delta_{t,k} \end{aligned} \quad (7)$$

BPTT 算法有点像 HMM 算法里的前后向计算步骤。在 BPTT 算法中，参数的梯度需要在一个完整的“前向”计算和“反向”计算后才能得到并进行参数更新。

(b) 实时循环学习法 RTRL

RTRL 算法从第 1 个时刻开始，除了计算循环神经网络的隐状态之外，还依次前向计算偏导数 $\frac{\partial h_1}{\partial u_{ij}}, \frac{\partial h_2}{\partial u_{ij}}, \frac{\partial h_3}{\partial u_{ij}}, \dots$ ，然后同时计算损失函数对 u_{ij} 的偏导数

$$\frac{\partial \mathcal{L}_t}{\partial u_{ij}} = \frac{\partial h_t}{\partial u_{ij}} \frac{\partial \mathcal{L}_t}{\partial h_t} \quad (8)$$

(c) 两种算法比较

在输出维度远低于输入维度的循环神经网络中，BPTT 算法的计算量会更小，但是 BPTT 算法需要保存所有时刻的中间梯度，空间复杂度较高。RTRL 算法不需要梯度回溯，因此更适合用于需要在线学习或无限序列的任务中。

1.2 RNN 的改进方案

RNN 本质上还是通过反向传播算法来学习参数，那么主要存在的问题仍旧是连乘导致的梯度消失（过小的权重连乘）和梯度爆炸（过大的权重连乘），所以很难建模长时间间隔的状态之间

的依赖关系。梯度消失问题可以通过残差连接的思想缓解，即使 h_t 和 h_{t-1} 之间既有线性关系也有非线性关系。非线性关系保留了模型的表达能力，线性关系又有效缓解了梯度消失问题。

$$h_t = h_{t-1} + g(x_t, h_{t-1}; \theta) \quad (9)$$

但是改进后仍然存在梯度爆炸和记忆容量的问题。所以引入门控机制来进一步改进模型。

1.2.1 长短期记忆网络 LSTM

1. LSTM 循环单元结构

LSTM 中除了隐藏状态以外，还引入了新的内部状态（记忆单元） $c_t \in \mathbb{R}^D$ 进行线性的循环信息传递，同时非线性地输出信息给隐藏状态 $h_t \in \mathbb{R}^D$ 。而内部状态 c_t 还依赖于一个通过非线性函数得到的中间参数——候选状态 $\tilde{c}_t \in \mathbb{R}^D$ 。

除了内部状态，还引入了门控机制，分别为输入门 i_t 、遗忘门 f_t 和输出门 o_t 。这三个门的作用是：

- (a) 遗忘门 f_t 控制上一个时刻的内部状态 c_{t-1} 需要遗忘多少信息
- (b) 输入门 i_t 控制当前时刻的候选状态 \tilde{c}_t 有多少信息需要保存
- (c) 输出门 o_t 控制当前时刻的内部状态 c_t 有多少信息需要输出给隐状态 h_t

由以上新引入的变量构成的循环单元结构为：

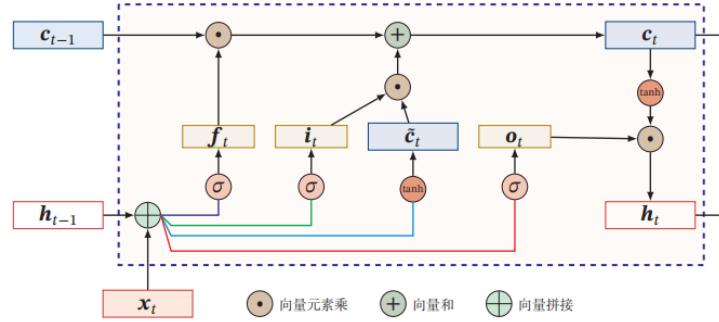


Figure 2: LSTM 循环单元结构

2. LSTM 数学表达

LSTM 中关于这些新引进的变量之间的关系为：

$$\begin{aligned} i_t &= \sigma(\mathbf{W}_i x_t + \mathbf{U}_i h_{t-1} + \mathbf{b}_i) \\ f_t &= \sigma(\mathbf{W}_f x_t + \mathbf{U}_f h_{t-1} + \mathbf{b}_f) \\ o_t &= \sigma(\mathbf{W}_o x_t + \mathbf{U}_o h_{t-1} + \mathbf{b}_o) \\ \tilde{c}_t &= \tanh(\mathbf{W}_c x_t + \mathbf{U}_c h_{t-1} + \mathbf{b}_c) \\ c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned} \quad (10)$$

其中 $\sigma(\cdot)$ 为 Logistic 函数，其输出区间为 $(0, 1)$

3. 长短期记忆

- (a) 短期记忆：每个时刻都会被重写的隐状态
- (b) 长期记忆：网络参数，隐含了从训练数据中学到的经验，其更新周期要远远长于短期记忆
- (c) 长短期记忆：在 LSTM 网络中，记忆单元 c 可以在某个时刻捕捉到某个关键信息，并有能力将此关键信息保存一定的时间间隔。记忆单元 c 中保存信息的生命周期要长于短期记忆 h ，但又远远短于长期记忆。所以“长短期记忆”是指指长的“短期记忆”

1.2.2 门控循环单元网络 GRU

1. GRU 循环单元结构

GRU 不引入额外的记忆单元，只引入了更新门 $z_t \in [0, 1]^D$ 和重置门 $r_t \in [0, 1]^D$ 和隐状态的候选状态 \tilde{h}_t ，通过线性变换来控制当前状态需要从历史状态中保留多少信息，以及需要从候选状态中接受多少新信息。因此，GRU 的表达要比 LSTM 简单很多。

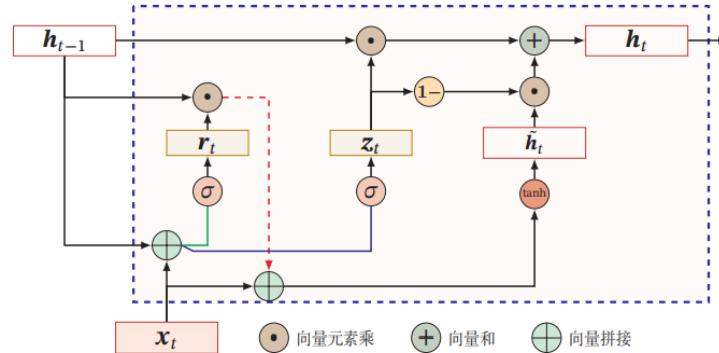


Figure 3: GRU 循环单元结构

2. GRU 数学表达

$$\begin{aligned}
 z_t &= \sigma(\mathbf{W}_z \mathbf{x}_t + \mathbf{U}_z \mathbf{h}_{t-1} + \mathbf{b}_z) \\
 r_t &= \sigma(\mathbf{W}_r \mathbf{x}_t + \mathbf{U}_r \mathbf{h}_{t-1} + \mathbf{b}_r) \\
 \tilde{h}_t &= \tanh(\mathbf{W}_h \mathbf{x}_t + \mathbf{U}_h (r_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h) \\
 h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t
 \end{aligned} \tag{11}$$

1.3 堆叠循环神经网络 SRNN

堆叠循环神经网络 (Stacked Recurrent Neural Network, SRNN) 可以简单地理解为将多个循环网络堆叠起来，也可以理解为增加隐状态的层数，如 Fig4 所示

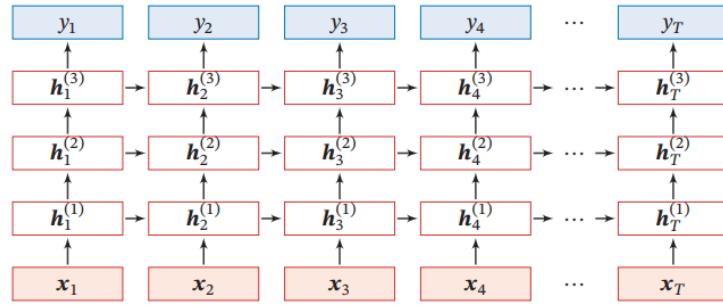


Figure 4: 堆叠神经网络

1.4 RNN 的应用

1.4.1 Many-to-One

Many-to-One，也就是“序列到类别模式”，主要用于序列数据的分类问题，比如情感分类。主要有两种模式，一种是将最终的隐藏状态 h_T 看成是序列的最终表示（特征），如 Fig5(a) 所示；另外一种是将所有隐藏状态进行平均作为整个序列的表示，如 Fig5(b) 所示。

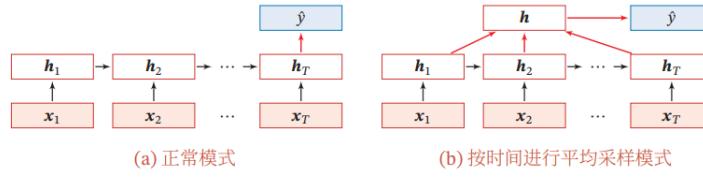


Figure 5: Many-to-One

1.4.2 Many-to-Many

Many-to-Many，也就是“同步的序列到序列模式”，主要用于序列标注任务，比如词性标注。其模式如 Fig6所示。

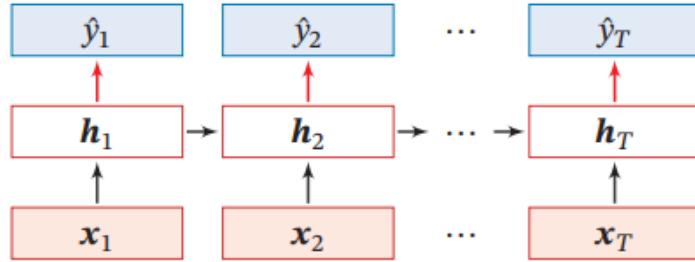


Figure 6: Many-to-Many

1.4.3 Seq2Seq

1. Encoder-Decoder 机制

一般的深度学习模型都可以归类为 Encoder-Decoder 机制：

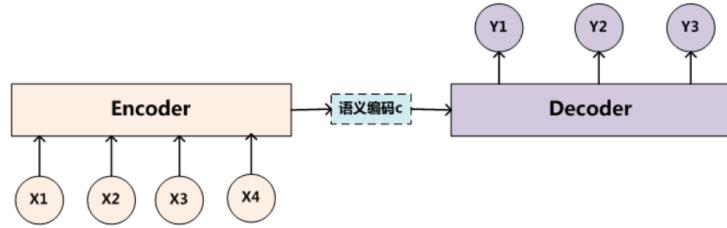


Figure 7: Encoder-Decoder

语义编码 C 可以理解为中间参数

2. 异步的序列到序列模型

Seq2Seq 模型可以用于机器翻译、语音识别、文本摘要、图像标题生成等序列到序列的生成任务。

RNN 应用到 Seq2Seq 模型就是“异步的序列到序列模型”，如 Fig8 所示。在模型中， h_{T+1} 可以看作是中间参数，也就是语义编码 C 。编码器是生成 h_{T+1} 的部分，解码器是产生序列 \hat{y}_t 的部分。

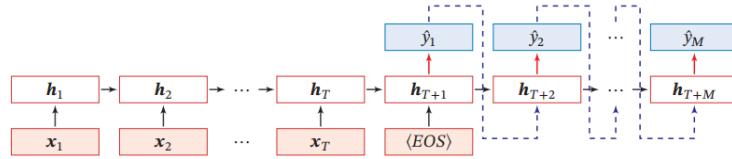


Figure 8: Seq2Seq

其数学表达为：

$$\begin{aligned}
 h_t &= f_1(h_{t-1}, x_t), \forall t \in [1, T] \\
 h_{T+t} &= f_2(h_{T+t-1}, \hat{y}_{t-1}), \forall t \in [1, M] \\
 \hat{y}_t &= g(h_{T+t}), \forall t \in [1, M]
 \end{aligned} \tag{12}$$

1.5 Attention 机制

具体参见论文阅读记录。

1.6 关于 RNN 理论学习的总结

一个简单的 RNN 结构可以看成是一个简单的全连接网络的“循环”，正如 Fig1，其循环的含义是前一个隐藏状态也要作为后一个隐藏状态的输入。我认为更清楚一点的理解是“依赖”，就是时序序列之间的依赖关系。

对于 RNN 的改进是基于门控机制。

2 实验部分

2.1 股市预测

在这个实验开始之前，记录一些股市专用的术语。

1. open(开盘价): 是指当日开盘后该股票的第一笔交易成交的价格。如果开市后 30 分钟内无成交价，则以前日的收盘价作为开盘价
2. close(收盘价): 指每天成交中最后一笔股票的价格
3. volume(交易量): 反映成交的数量多少和参与买卖人的多少。一般可用成交股数和成交金额两项指标来衡量
4. high(最高价): 是指当日所成交的价格中的最高价位
5. low(最低价): 是指当日所成交的价格中的最低价位
6. p-change(涨跌幅): 计算方式为 (当天的 close-昨天的

2.1.1 代码实现

实现的网络结构如下：

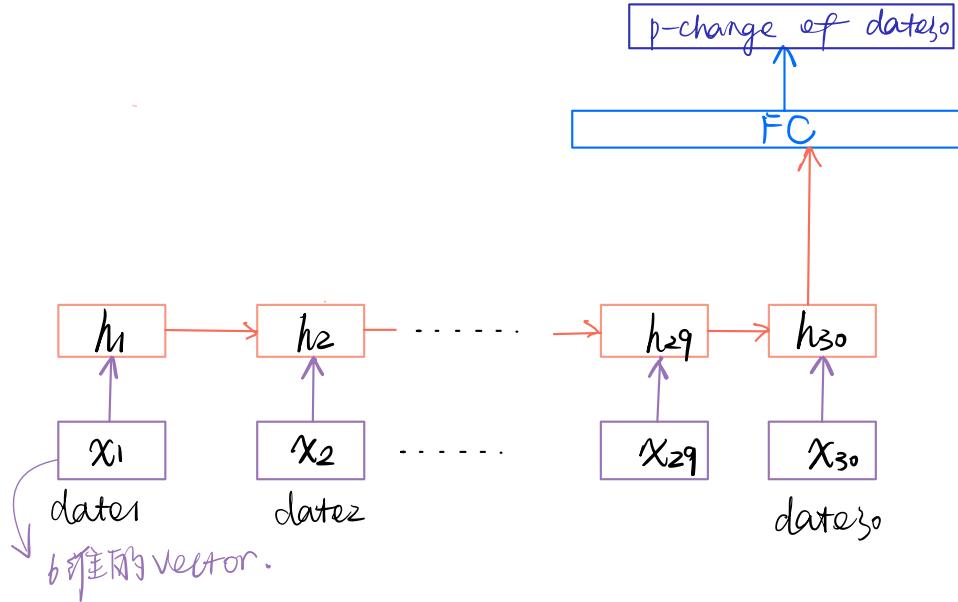


Figure 9: 股市预测网络结构

1. 数据处理

数据来源见参考资料。下载的数据需要做以下处理包括：升序排列，计算 p-change 和必要的字符串处理（把货币符去掉）。数据集大小为 250，数据集每一条数据包括日期、open、close、

volume、high、low、p-change。数据集前 200 个数据作为训练集，整个数据集即 250 个数据作为测试集。

```
# 数据处理
def prepare_data(dir, N):
    aapl = pd.read_csv(dir)
    X = []
    y = []
    date = []
    for i in range(30, len(aapl) - 1):
        y.append(aapl.iloc[i + 1, 6])
        date.append(aapl.iloc[i, 0])

    aapl['Close'] = aapl['Close'].apply(lambda x: x[1:])
    aapl['Open'] = aapl['Open'].apply(lambda x: x[1:])
    aapl['High'] = aapl['Low'].apply(lambda x: x[1:])
    aapl['Low'] = aapl['Low'].apply(lambda x: x[1:])

    mms = MinMaxScaler()
    # 训练集归一化
    mms.fit(aapl.iloc[:N][['Close', 'Volume', 'Open', 'High', 'Low']])
    aapl[['Close', 'Volume', 'Open', 'High', 'Low']] = mms.transform(aapl[['Close', 'Volume',
                                                                     'Open', 'High', 'Low']])
    # print(aapl.head())
    for i in range(31, len(aapl)):
        list = aapl.iloc[i - 30:i, 1:7].values.tolist()
        X.append(list)
    X = torch.tensor(X, dtype=torch.float)
    y = torch.tensor(y, dtype=torch.float).view(-1, 1)
    X_train, X_test = X[:N], X
    y_train, y_test = y[:N], y
    trainloader = DataLoader(TensorDataset(X_train, y_train), 64, True)
    return trainloader, date, X_test, y_test
```

2. 网络搭建

RNN 的实现，利用 pytorch 自带的工具库比较简单。以 RNN 的实现为例，平均池化层和 squeeze 的目的是压缩一维数据使数据符合线性层的维度要求。

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        self.gru = nn.GRU(input_size=input_size, hidden_size=hidden_size, batch_first=True)
        self.lstm = nn.LSTM(input_size=input_size, hidden_size=hidden_size, num_layers=2,
                           batch_first=True)
        self.rnn = nn.RNN(input_size=input_size, hidden_size=hidden_size, num_layers=1,
                          batch_first=True)
        self.activation = nn.ReLU()
        self.output = nn.Linear(hidden_size, 1)

    def forward(self, x):
        # ----rnn----
        x, _ = self.rnn(x)
        # x=[batch_size, days, hidden_size]
```

```

print(x.size())
out = F.avg_pool2d(x, (x.shape[1], 1)).squeeze()
return self.output(out)

```

2.1.2 结果分析

大致的结果是，RNN 网络对于测试集前 200 个数据可以拟合地很不错，但是对于后 50 个数据差强人意。下面比较不同的参数的影响。

1. RNN、GRU、LSTM 的比较

参数：训练轮数 1000，隐藏神经元 32，隐藏层数 1，优化器 Adam。

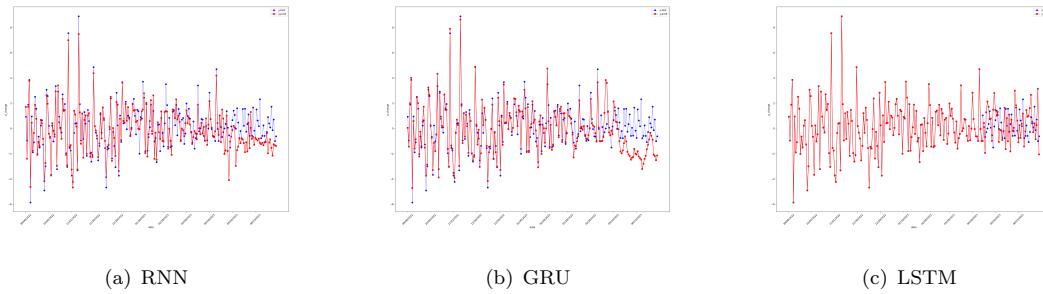


Figure 10: RNN、GRU、LSTM 的比较

在相同的参数条件下，LSTM 结构训练效果是最好的。虽然仍然存在最后 50 个数据预测不准确的情况，但是对于是涨还是跌的情况预测地还是比较好的，对于股市来说，有一定参考性。

2. 隐藏层神经元个数的影响

参数：LSTM，训练轮数 1000，隐藏层数 1，优化器 Adam。

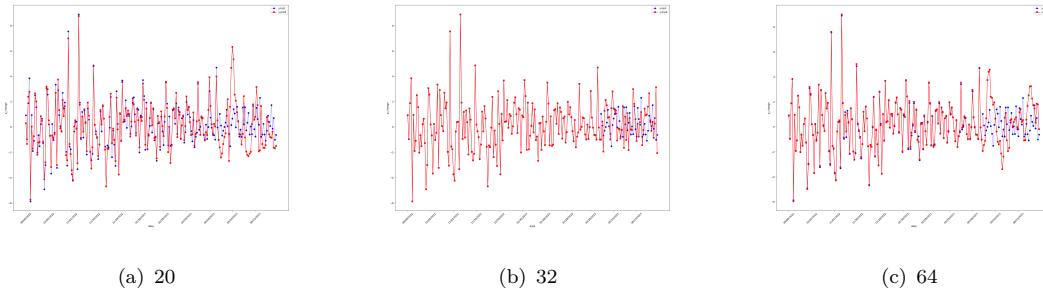


Figure 11: 隐藏层神经元个数的影响

对于 LSTM 结构来说，32 个隐藏神经元效果是比较好的，20 个是欠拟合，而 64 个是过拟合。

3. Stack 结构的影响

参数：训练轮数 1000，优化器 Adam。RNN-20-1 的意思是使用 RNN 结构，隐藏状态数 20，隐藏层为 1。

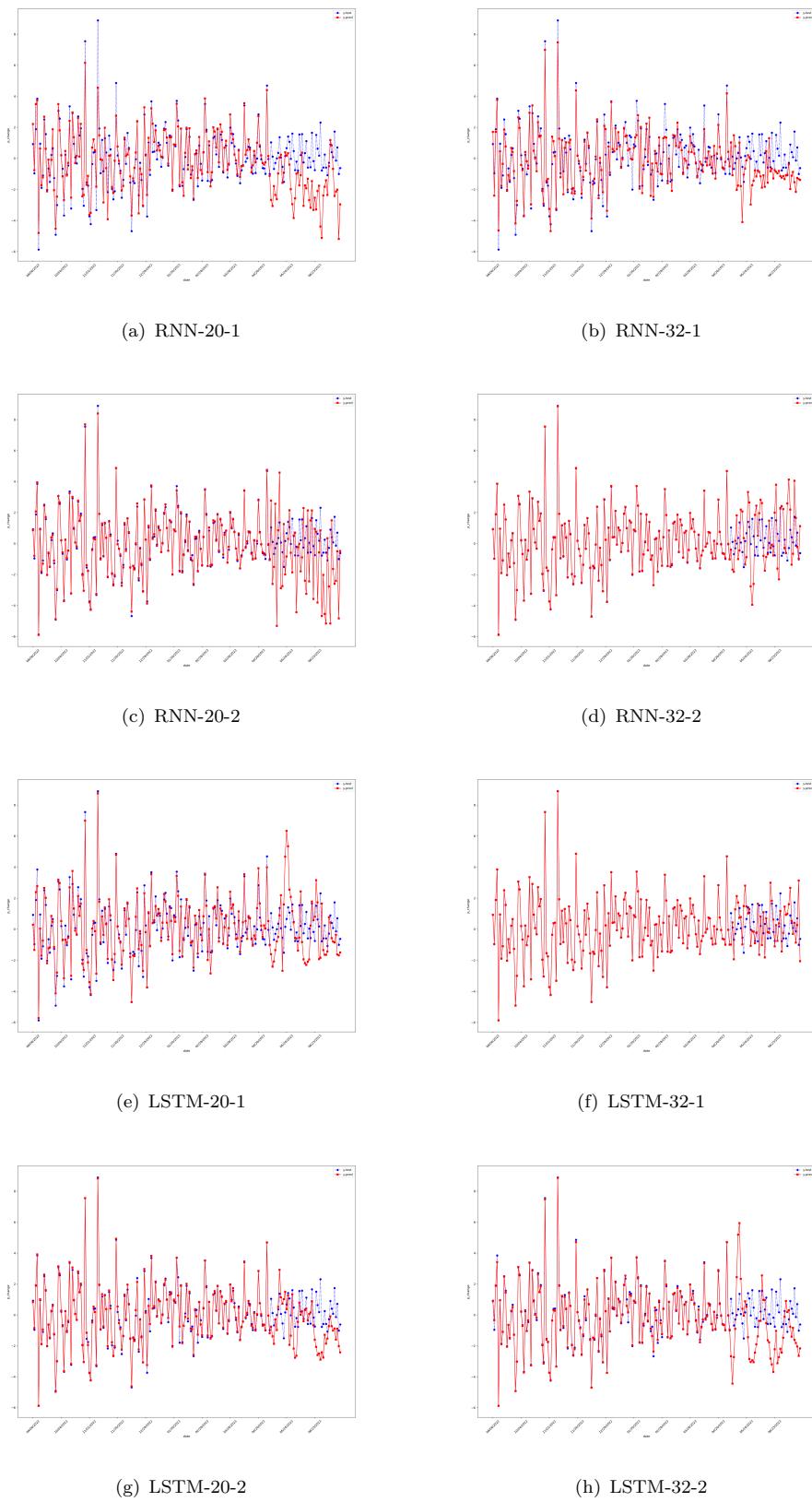


Figure 12: Stack 结构的影响

多层隐藏层一般要比单层隐藏层效果要好，但是隐藏层数多了的话，容易过拟合。

4. 优化算法的影响

参数：RNN，训练轮数 1000，隐藏层状态数 32，隐藏层数 1。

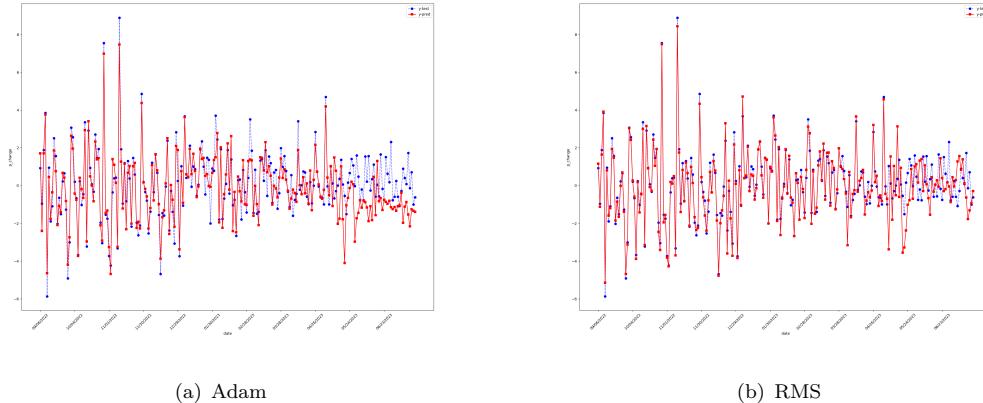


Figure 13: 优化算法的影响

RMS 优化算法明显要较优于 Adam 算法。

2.2 情感分类

2.2.1 代码实现

代码实现的难点主要在于数据集的处理。RNN 的实现大同小异，主要是增加了 embedding 层。其网络结构如下（双层双向 LSTM 结构为例）：

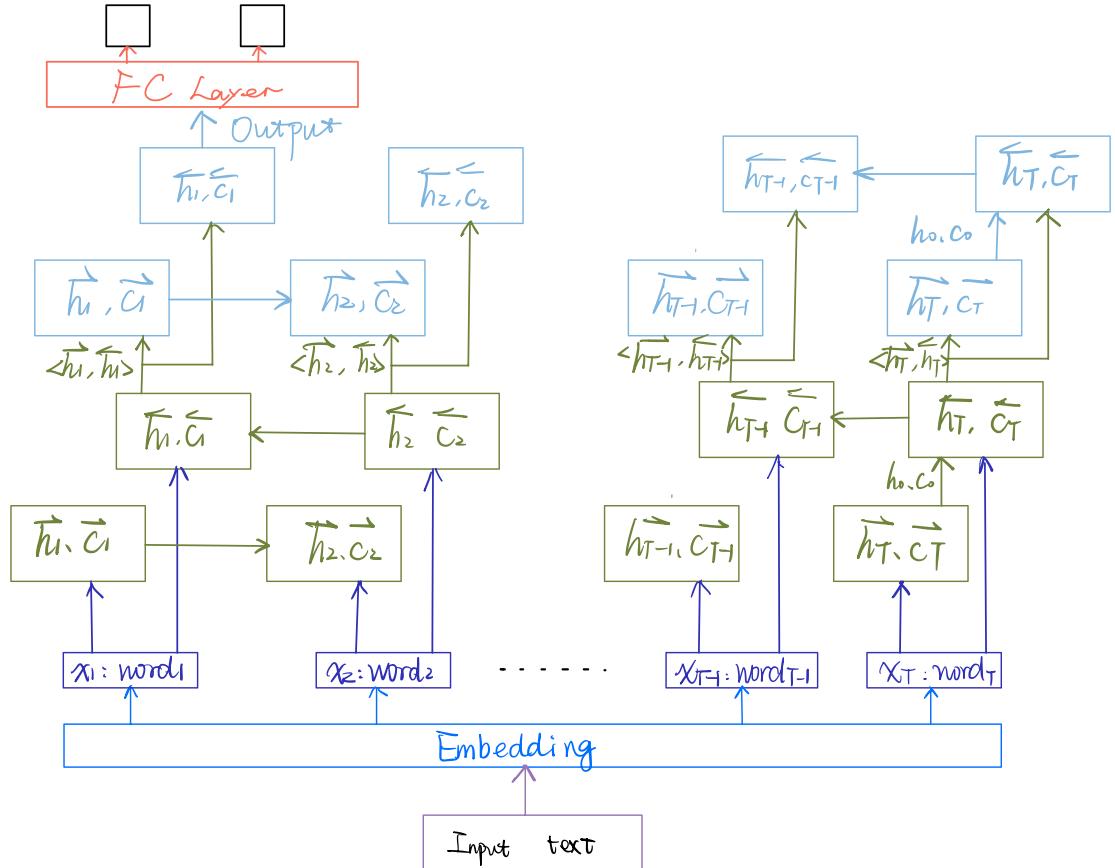


Figure 14: 情感分类网络结构

1. 数据处理

数据处理用了 keras 自带的 idmb 数据集转为 dataloader 形式。

```
def prepare_data():
    (x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=MAX_WORDS)
    x_train = pad_sequences(x_train, maxlen=MAX_LEN, padding='post', truncating='post')
    x_test = pad_sequences(x_test, maxlen=MAX_LEN, padding='post', truncating='post')
    # 将数据转为 tensorDataset
    train_data = TensorDataset(torch.LongTensor(x_train), torch.LongTensor(y_train))
    test_data = TensorDataset(torch.LongTensor(x_test), torch.LongTensor(y_test))
    # 将数据放入 dataloader
    train_sampler = RandomSampler(train_data)
    trainloader = DataLoader(train_data, sampler=train_sampler, batch_size=BATCH_SIZE)
    test_sampler = SequentialSampler(test_data)
    testloader = DataLoader(test_data, sampler=test_sampler, batch_size=BATCH_SIZE)
    return trainloader, testloader
```

2. 网络搭建

和股市预测相比，RNN 网络只添加了 embedding 层。对于 LSTM 网络，添加了更多的参数，搭建了 Bi-LSTM 网络。同样地，增加了平均池化层和 squeeze。因为在 rnn 层后得到的输出的维度为 [batch_size,sentence_length(MAX_LEN),embedding_size]，而线性层需要的输入维度需要将 batch_size 这一维放在 embedding_size 维度之前，所以在这里选择将第 1 维，也就是 sentence_length 这一维直接删除。

```
class LSTM(nn.Module):
    def __init__(self, input_size, embedding_size, hidden_size):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.bidirectional = False
        self.embedding_size = embedding_size
        self.bi_num = 2 if self.bidirectional else 1
        self.num_layer = 2

        self.embedding = nn.Embedding(input_size, embedding_size)

        self.lstm = nn.LSTM(self.embedding_size, self.hidden_size,
                           self.num_layer, bidirectional=self.bidirectional, batch_first=True)
        self.output = nn.Linear(self.hidden_size * self.bi_num, 2)

    def forward(self, x):
        embedded = self.embedding(x)
        x, (_, _) = self.lstm(embedded)
        # x=[batch_size,sen_size(MAX_LEN),embedding_size]
        out = F.avg_pool2d(x, (x.shape[1], 1)).squeeze()
        # out=[batch_size,embedding_size]
        out = self.output(out)
        return out
```

2.2.2 结果分析

由于 idmb 数据集比较大，网络训练得比较慢，所以没有进行很全面的对比分析。

1. Stack 结构的影响

参数：RNN，训练轮数 5，优化器 AdamW，隐藏层神经元 100。

[Epoch1] Loss:0.75452 Test acc:0.505	[Epoch1] Loss:0.54662 Test acc:0.825	[Epoch1] Loss:0.30322 Test acc:0.828
[Epoch2] Loss:0.69389 Test acc:0.503	[Epoch2] Loss:0.29224 Test acc:0.843	[Epoch2] Loss:0.21403 Test acc:0.831
[Epoch3] Loss:0.69170 Test acc:0.500	[Epoch3] Loss:0.20652 Test acc:0.842	[Epoch3] Loss:0.20513 Test acc:0.827
[Epoch4] Loss:0.68434 Test acc:0.507	[Epoch4] Loss:0.15799 Test acc:0.830	[Epoch4] Loss:0.19301 Test acc:0.827
[Epoch5] Loss:0.67208 Test acc:0.509	[Epoch5] Loss:0.11075 Test acc:0.831	[Epoch5] Loss:0.15605 Test acc:0.827

(a) 1

(b) 2

(c) 3

Figure 15: Stack 结构的影响

对于 RNN 网络来说，一层隐藏层很难收敛，而对于 2 层和 3 层来说，收敛地比较快，并且能达到还不错的准确率。相比较来说，两层的结构效果略好。

2. Bi-directional 结构的影响

参数: LSTM, 训练轮数 5, 优化器 AdamW, 隐藏层神经元 100, 隐藏层数 2。

[Epoch1] Loss:0.45188 Test acc:0.865	[Epoch1] Loss:0.66970 Test acc:0.742
[Epoch2] Loss:0.21366 Test acc:0.863	[Epoch2] Loss:0.48821 Test acc:0.801
[Epoch3] Loss:0.10534 Test acc:0.860	[Epoch3] Loss:0.34018 Test acc:0.809
[Epoch4] Loss:0.05400 Test acc:0.852	[Epoch4] Loss:0.27306 Test acc:0.812
[Epoch5] Loss:0.02862 Test acc:0.851	[Epoch5] Loss:0.17847 Test acc:0.831

(a) 单向结构

(b) 双向结构

Figure 16: Bi-directional 结构的影响

相比来说, 单向 LSTM 结构要更好一点, 但是因为轮数的限制, 双向结构还在收敛的过程中, 其效果应该要比单向 LSTM 结构要更好。

2.3 词性标注

2.3.1 代码实现

1. 数据处理

数据处理部分主要是将 UDPOS 数据集制作成 dataloader 形式。从 torchtext 中直接加载的数据集分为 text 和 udtags 和 postags 三部分，实验中只用了 udtags。数据处理的部分主要是处理文本，统计词频，然后创建词汇表。在本实验的处理当中，使用了预训练的词向量来初始化词汇表中的单词向量，也就是为词汇表中的每一个单词分配了一个已经设计好的词向量，这可以更好地表示词汇之间的特征。

```
def prepare_data():
    TEXT = data.Field(lower=True)
    UD_TAGS = data.Field(unk_token=None)
    fields = (("text", TEXT), ("udtags", UD_TAGS), (None, None))
    # 加载UDPOS数据集，只加载UD标签
    train_data, valid_data, test_data = datasets.UDPOS.splits(fields)
    # 构建词汇表，统计并排序词频、为每个单词分配向量标识符
    TEXT.build_vocab(train_data, min_freq=2, vectors="glove.6B.100d", unk_init=torch.Tensor.
                    normal_)
    UD_TAGS.build_vocab(train_data)
    trainloader, validloader, testloader = data.BucketIterator.splits((train_data,
                                                                      valid_data, test_data), batch_size=
                                                                      BATCH_SIZE, device=device)
    return trainloader, validloader, testloader, len(TEXT.vocab), len(UD_TAGS.vocab),
           TEXT.vocab.stoi[TEXT.pad_token], UD_TAGS.vocab.stoi[UD_TAGS.pad_token]
```

2. 网络搭建

基本上没有什么其他的变化。不同点大概在于在词性标注任务中不需要用平均池化层和 squeeze 函数来压缩维度。因为在词性标注任务中，文本中的每一个单词都需要对应一个标签。也就是说 sentence_length 这一维是不需要压缩的。也因此，在输入的时候，这一维被处理成了第 0 维。

2.3.2 结果分析

对于该任务，经过多次实验对比分析，选择最基础的 RNN 结构和参数就可以达到比较不错的效果，参数的改变并没有质的提升。如果想要进一步提高，可能需要添加新的结构和方法。

1. 隐藏层大小的影响

参数：RNN，训练轮数 5，优化器 AdamW，隐层 1。

<pre>[Epoch1] Loss:0.97581 Train acc:0.701 Val acc:0.778 [Epoch2] Loss:0.41189 Train acc:0.861 Val acc:0.803 [Epoch3] Loss:0.34117 Train acc:0.878 Val acc:0.806 [Epoch4] Loss:0.32669 Train acc:0.882 Val acc:0.808 [Epoch5] Loss:0.31119 Train acc:0.883 Val acc:0.808 Test acc: 0.814</pre>	<pre>[Epoch1] Loss:0.87429 Train acc:0.722 Val acc:0.784 [Epoch2] Loss:0.39395 Train acc:0.864 Val acc:0.804 [Epoch3] Loss:0.33207 Train acc:0.878 Val acc:0.804 [Epoch4] Loss:0.31484 Train acc:0.880 Val acc:0.806 [Epoch5] Loss:0.30698 Train acc:0.882 Val acc:0.808 Test acc: 0.812</pre>	<pre>[Epoch1] Loss:0.82952 Train acc:0.730 Val acc:0.779 [Epoch2] Loss:0.39397 Train acc:0.862 Val acc:0.802 [Epoch3] Loss:0.33388 Train acc:0.877 Val acc:0.802 [Epoch4] Loss:0.31776 Train acc:0.880 Val acc:0.810 [Epoch5] Loss:0.31124 Train acc:0.880 Val acc:0.826 Test acc: 0.824</pre>
--	--	--

(a) 32

(b) 64

(c) 128

Figure 17: 隐藏层大小的影响

相比较而言，选择稍微大一点的隐藏层效果更好。

2. RNN、LSTM、GRU 的比较

参数：训练轮数 5，优化器 AdamW，隐层 1，隐状态 128。

```
[Epoch1] Loss:0.82952 | Train acc:0.730 | Val acc:0.779 [Epoch1] Loss:0.85007 | Train acc:0.725 | Val acc:0.787  
[Epoch2] Loss:0.39397 | Train acc:0.862 | Val acc:0.802 [Epoch2] Loss:0.38481 | Train acc:0.865 | Val acc:0.803  
[Epoch3] Loss:0.33388 | Train acc:0.877 | Val acc:0.802 [Epoch3] Loss:0.32834 | Train acc:0.879 | Val acc:0.807  
[Epoch4] Loss:0.31776 | Train acc:0.880 | Val acc:0.810 [Epoch4] Loss:0.31202 | Train acc:0.881 | Val acc:0.808  
[Epoch5] Loss:0.31124 | Train acc:0.880 | Val acc:0.826 [Epoch5] Loss:0.30504 | Train acc:0.883 | Val acc:0.813  
Test acc: 0.824 Test acc: 0.815
```

(a) RNN

(b) LSTM

(c) GRU

Figure 18: RNN、LSTM、GRU 的比较

三者的效果是差不多的，没有什么特别大的区别。

3. StackRNN 的影响

参数：RNN，训练轮数 5，优化器 AdamW，隐藏层神经元 128。

```
[Epoch1] Loss:0.82952 | Train acc:0.730 | Val acc:0.779 [Epoch1] Loss:0.86733 | Train acc:0.716 | Val acc:0.776  
[Epoch2] Loss:0.39397 | Train acc:0.862 | Val acc:0.802 [Epoch2] Loss:0.41719 | Train acc:0.857 | Val acc:0.799  
[Epoch3] Loss:0.33388 | Train acc:0.877 | Val acc:0.802 [Epoch3] Loss:0.35045 | Train acc:0.874 | Val acc:0.801  
[Epoch4] Loss:0.31776 | Train acc:0.880 | Val acc:0.810 [Epoch4] Loss:0.32796 | Train acc:0.878 | Val acc:0.797  
[Epoch5] Loss:0.31124 | Train acc:0.880 | Val acc:0.826 [Epoch5] Loss:0.31891 | Train acc:0.879 | Val acc:0.811  
Test acc: 0.824 Test acc: 0.815
```

(a) 一层

(b) 两层

Figure 19: Stack 结构的影响

两者比较，其实也没有什么特别大的区别。

4. Bi-directional 结构的影响

参数：LSTM，训练轮数 5，优化器 AdamW，隐藏层神经元 128，隐藏层数 1。

```
[Epoch1] Loss:0.85007 | Train acc:0.725 | Val acc:0.787 [Epoch1] Loss:0.82188 | Train acc:0.733 | Val acc:0.791  
[Epoch2] Loss:0.38481 | Train acc:0.865 | Val acc:0.803 [Epoch2] Loss:0.37597 | Train acc:0.867 | Val acc:0.804  
[Epoch3] Loss:0.32834 | Train acc:0.879 | Val acc:0.807 [Epoch3] Loss:0.32595 | Train acc:0.879 | Val acc:0.805  
[Epoch4] Loss:0.31202 | Train acc:0.881 | Val acc:0.808 [Epoch4] Loss:0.31104 | Train acc:0.881 | Val acc:0.806  
[Epoch5] Loss:0.30504 | Train acc:0.883 | Val acc:0.813 [Epoch5] Loss:0.30481 | Train acc:0.882 | Val acc:0.819  
Test acc: 0.815 Test acc: 0.822
```

(a) 单向结构

(b) 双向结构

Figure 20: Bi-directional 结构的影响

相比来说，双向 LSTM 结构要更好一点。

5. 优化器的影响

参数：RNN，训练轮数 5，隐藏层神经元 128，隐藏层数 1。

RMS 优化器收敛地比较慢，所以训练了 15 轮，取其饱和的状态进行比较。

```
[Epoch1] Loss:0.82987 | Train acc:0.730 | Val acc:0.779  
[Epoch2] Loss:0.39457 | Train acc:0.861 | Val acc:0.881  
[Epoch3] Loss:0.33418 | Train acc:0.877 | Val acc:0.882  
[Epoch4] Loss:0.31764 | Train acc:0.880 | Val acc:0.810  
[Epoch5] Loss:0.31050 | Train acc:0.880 | Val acc:0.823  
Test acc: 0.823
```

```
[Epoch1] Loss:0.82952 | Train acc:0.730 | Val acc:0.779  
[Epoch2] Loss:0.39397 | Train acc:0.862 | Val acc:0.882  
[Epoch3] Loss:0.33388 | Train acc:0.877 | Val acc:0.882  
[Epoch4] Loss:0.31776 | Train acc:0.880 | Val acc:0.810  
[Epoch5] Loss:0.31124 | Train acc:0.880 | Val acc:0.826  
Test acc: 0.824
```

```
[Epoch1] Loss:0.37040 | Train acc:0.868 | Val acc:0.882  
[Epoch2] Loss:0.34550 | Train acc:0.875 | Val acc:0.888  
[Epoch3] Loss:0.33510 | Train acc:0.877 | Val acc:0.888  
[Epoch4] Loss:0.32722 | Train acc:0.879 | Val acc:0.815  
[Epoch5] Loss:0.32117 | Train acc:0.880 | Val acc:0.815  
Test acc: 0.818
```

(a) Adam

(b) AdamW

(c) RMS

Figure 21: 优化器的影响

最后可以达到的效果是差不多的。相比来说 Adam 优化器效果最好，但是 AdamW 优化器要更快。RMS 优化器训练得比较慢。

3 参考资料

邱锡鹏《神经网络与深度学习》P129-P150, P355-P381
<https://zhuanlan.zhihu.com/p/91839581>
<https://zhuanlan.zhihu.com/p/513553380>
<https://www.nasdaq.com/zh/market-activity/stocks/aapl/historical>
<https://www.cnblogs.com/gcjr/p/14693829.html>