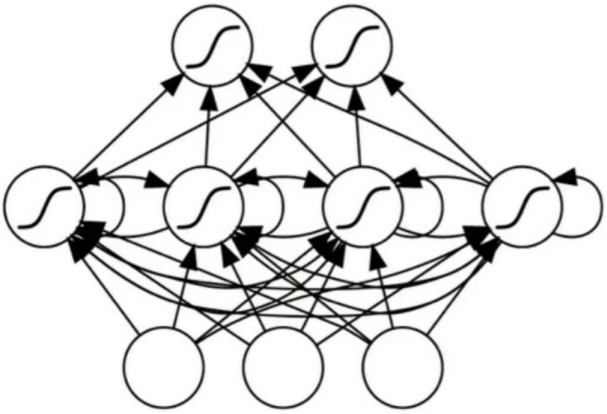
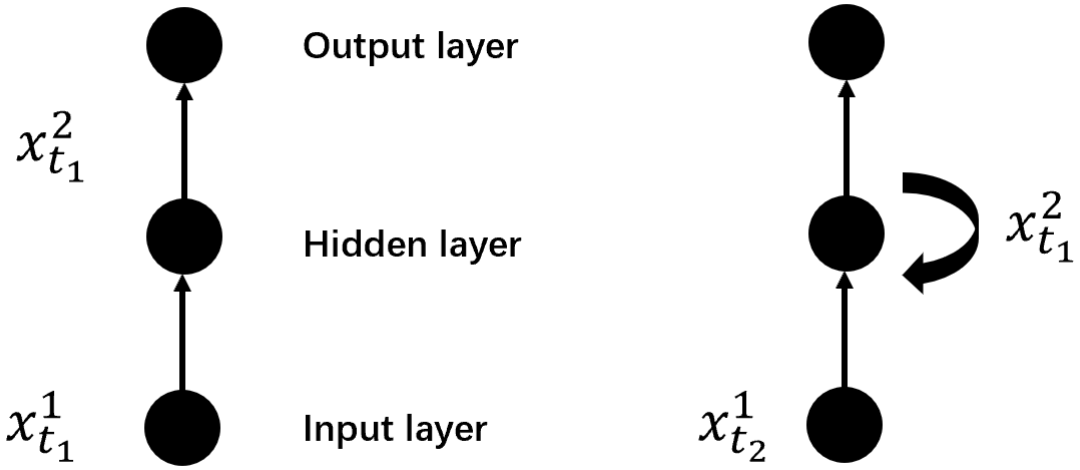
RNN

实现逻辑

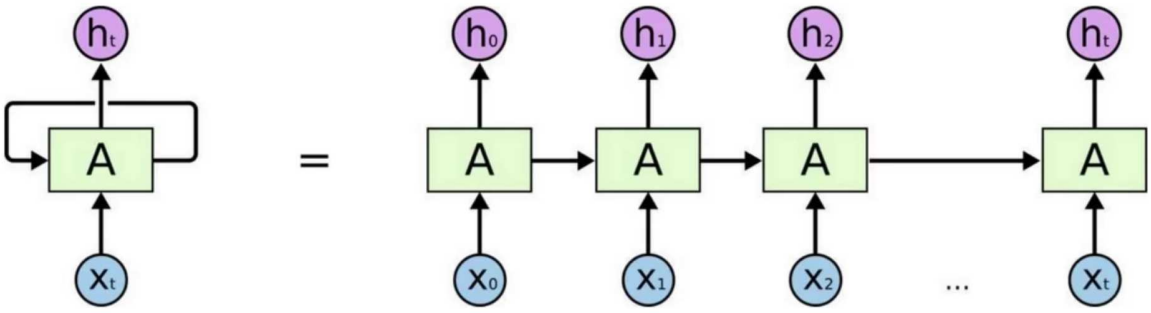
递归神经网络（recursive neural network），与前馈神经网络最大的不同就是，隐藏层神经元之间也会建立权重链接，不仅如此隐藏层的输出也会再次作为输入进行运算，即前一时刻做出的运算会对下一时刻的运算造成影响，那么就在特征上呈现了时间相关性。



在 t_1 时刻，经过隐藏层运算后，结果为 $x_{t_1}^2$ ，而在 t_2 时刻，新的输入 $x_{t_2}^1$ 和 $x_{t_1}^2$ 会组合成新的输入，从而提取到它们之间的相似性特征。



$x_{t_1}^2$ 和 $x_{t_2}^1$ 经过计算后得到 $x_{t_1+t_2}^2$ 会与下一时刻的输入 $x_{t_3}^1$ 再次输入到隐藏层，构成一个循环，形成了时间序列。



对于前序得到的暂时的神经元输出，RNN 只是把它们作为中间结果输入给下一个进来的序列，最终只考虑 h_t 的值作为隐藏层的输出，而之前的都只作为中间结果暂存。显然 h_t 就包含了之前所有序列的特征，但实际上在一段话中的第一个字符和最后一个字符并没有太大联系，包含的特征联系过于复杂会导致难以精确训练，所以产生了长短期记忆网络（LSTM, Long Short-Term Memory），让网络能够选择忘记一些特征。

实战

使用LSTM进行情感分析

https://www.bilibili.com/video/BV1Bi4y157xD

数据集是 Sentimention140, Twitter 上的内容，包含160万条记录，0：负面，4：正面

原始数据集处理

```
dataset = pd.read_csv("training.1600000.processed.noemoticon.csv",
engine="python", header=None)

print(dataset.shape) # [160万, 6]
print(dataset.info) # 看到每一列数据的内容

# 0        0  ...  @switchfoot http://twitpic.com/2y1zl - Awww, t...
# 1        0  ...  is upset that he can't update his Facebook by ...
# 2        0  ...  @Kenichan I dived many times for the ball. Man...
# 3        0  ...    my whole body feels itchy and like its on fire
# 4        0  ...  @nationwideclass no, it's not behaving at all....
...      ..  ...                                                ...
```

此时可以查看文本情感类别的统计，

```
print(dataset[0].value_counts()) # 统计各个类别数据占比
# 0    800000
# 4    800000
# Name: 0, dtype: int64
```

标签值为0和1更适合分类任务，

```
dataset['sentiment_category'] = dataset[0].astype('category') # 新建一列，并转换成分类变量
dataset['sentiment'] = dataset['sentiment_category'].cat.codes # 新建一列，分类变量转换成0和1
print(dataset['sentiment'].value_counts()) # 统计各个类别数据占比
# 0    800000
# 1    800000
# Name: sentiment, dtype: int64
print(dataset.info)
# 0        0  1467810369  ...                  0         0
# 1        0  1467810672  ...                  0         0
# 2        0  1467810917  ...                  0         0
# 3        0  1467811184  ...                  0         0
# 4        0  1467811193  ...                  0         0
# ...     ..         ...  ...                ...       ...
# 1599995  4  2193601966  ...                  4         1
```

```
# 1599996  4  2193601969  ...                 4         1
# ...     ..         ...  ...               ...       ...
```

保存为新的文件，

```
dataset.to_csv('training_processed.csv', header=None, index=None)
```

划分数据集

```
from torchtext.legacy import data

label = data.LabelField() # 标签
tweet = data.Field(lower=True) # 内容，都转变成小写

# 设置表头，原始数据表头是0，1，2，3，4，5
fields = [('score', None), ('id', None), ('date', None), ('query', None),
          ('name', None), ('tweet', tweet), ('category', None), ('label',
label)]

# 读取数据
twitterDataset = data.TabularDataset(
    path='training_processed.csv',
    format='CSV',
    fields=fields,
    skip_header=False # 不跳过表头
)

# 分离 train, test, val
train, test, val = twitterDataset.split(split_ratio=
[0.8,0.1,0.1],strata_field='label')
# print(len(train), len(test), len(val))
# print(vars(train[5643])) # 查看其中一个样本
```

构建词汇表

```
vocab_size = 20000 # 常见单词
tweet.build_vocab(train, max_size=vocab_size)
label.build_vocab(train)
# print(len(tweet.vocab)) # 20002, 多出来的两个单词是unk和pad，表示未知和填充单词
```

常见的查看词汇表的方法

```
# 查看下最常见的5个单词
print(tweet.vocab.freqs.most_common(5))
# [('i', 597193), ('to', 447685), ('the', 414891), ('a', 300750), ('my',
250075)]
# 查看索引到单词的对应关系
print(tweet.vocab.itos[:10])
# ['<unk>', '<pad>', 'i', 'to', 'the', 'a', 'my', 'and', 'you', 'is']
# 查看单词到索引的对应关系
print(tweet.vocab.stoi)
# {'<unk>': 0, '<pad>': 1, 'i': 2, 'to': 3, 'the': 4, ...}
```

文本的批处理

```
# 文本批处理，分割的同时做包装，一批一批的读取数据
train_iter, val_iter, test_iter = data.BucketIterator.splits((train,val,test),
                                                             batch_size=32,
                                                             device=device,

sort_within_batch=True,
                                                             sort_key=
                                                             lambda
x:len(x.tweet))
# sort_within_batch=True 一个batch内的数据就会按照sort_key的规则降序排列
# sort_key=lambda x:len(x.tweet)    使用tweet的长度，即包含的单词的数量
```

模型构建

```
device = "cuda" if torch.cuda.is_available() else "cpu"

class simple_LSTM(nn.Module): # 继承nn.Module
    def __init__(self, hidden_size, embedding_dim, vocal_size):
        super(simple_LSTM, self).__init__() # 调用父类的构造方法
        self.embedding = nn.Embedding(vocal_size, embedding_dim)
        self.encoder = nn.LSTM(input_size=embedding_dim,
hidden_size=hidden_size,
                               num_layers=1)
        self.predictor = nn.Linear(hidden_size, 2) # 二分类

    def forward(self, seq):
        output, (hidden, cell) = self.encoder(self.embedding(seq))
        # output    torch.size([24, 32, 100])
        # hidden    torch.size([1, 32, 100])
        # cell    torch.size([1, 32, 100])
        # 24:一条评论多少单词，32:batch_size，100:hidden_size
        preds = self.predictor(hidden.squeeze(0))
        # 不需要hidden中的“1”维度
        return preds

# 创建对象
lstm_model = simple_LSTM(100, 300, vocab_size+2)
lstm_model.to(device)

# 优化器
optimizer = optim.Adam(lstm_model.parameters(), lr=0.01)
```

```
# 损失函数
criterion = nn.CrossEntropyLoss()
```

输出准确率

```
def train_val_test(model, optimizer, criterion, train_iter, val_iter, test_iter,
epochs):
    for epoch in range(1, epochs+1):
        train_loss = 0
        val_loss = 0
        model.train() # 声明开始训练
        for indices, batch in enumerate(train_iter):
            optimizer.zero_grad() # 梯度置0
            outputs = model(batch.tweet) # [batch_size, 2]
            loss = criterion(outputs, batch.label)
            loss.backward() # 反向传播
            optimizer.step() # 参数更新
            train_loss += loss.data.item()*batch.tweet.size(0) # 累计每一批的损失值
        train_loss /= len(train_iter) # 计算平均损失
        print("Epoch: {}, Train loss: {:.2f}".format(epoch, train_loss))

        # 声明验证
        model.eval()
        for indices, batch in enumerate(val_iter):
            context = batch.tweet.to(device)
            target = batch.label.to(device)
            pred = model(context)
            loss = criterion(pred, target)
            val_loss += loss.item()*context.size(0) # 累计每一批的损失值
        val_loss /= len(val_iter) # 计算平均损失
        print("Epoch: {}, Val loss: {:.2f}".format(epoch, val_loss))

        # 声明测试
        model.eval()
        correct = 0
        test_loss = 0
        with torch.no_grad(): # 测试时，无需进行梯度计算
            for idx, batch in enumerate(test_iter):
                context = batch.tweet.to(device)
                target = batch.label.to(device)
                outputs = model(context)
                loss = criterion(outputs, target)
                test_loss += loss.item()*context.size(0) # 累计每一批的损失值
                # 获取最大预测值的索引
                preds = outputs.argmax(1)
                correct += preds.eq(target.view_as(preds)).sum().item()
            test_loss /= len(test_iter) # 计算平均损失
        print("Epoch: {}, Test loss: {:.2f}".format(epoch, test_loss))
        print("Accuracy:
{}".format(100*correct/(len(test_iter)*batch.tweet.size(1))))
```

开始训练

```
train_val_test(lstm_model, optimizer, criterion,
               train_iter, val_iter, test_iter, epochs=2)
```

数据增强