1、自然语言处理

适用与序列模型，比如语音，歌曲，文字等,适用于小数据集，低算力的问题有效。

类型：

one to one:固定得到输入到输出，如图像分类

one to many:固定的输入到序列输出，如图像的文字描述

many to one:序列输入到输出，如情感分析，分类正面负面情绪

many to many:序列输入到序列的输出，如机器翻译，称之为编解码网络

同步多对多：同步序列输入到同步输出，如文本生成，视频每一帧的分类，也称之为序列生成。

输入：词的表示：用one-hot编码（就是会建立所有词的词库）

输出：用softmax激活函数表示概率（输出词的概率）

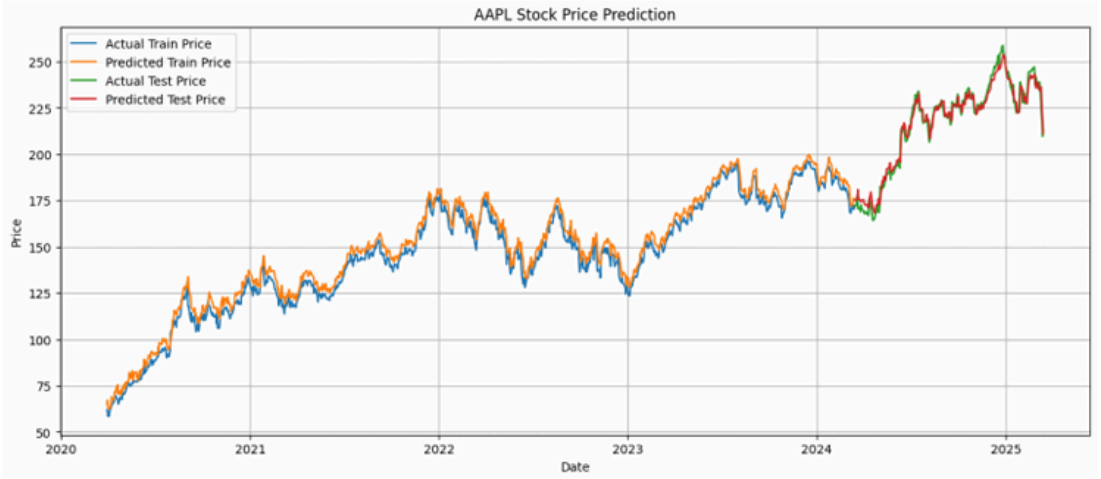
缺点：有限记忆

时间序列表示方法：sequence representation:[个数，维度]

```
 import numpy as np
 import pandas as pd
 import yfinance as yf
 from sklearn.preprocessing import MinMaxScaler
 from tensorflow.keras.models import Sequential
 from tensorflow.keras.layers import Dense, SimpleRNN
 import matplotlib.pyplot as plt
 # 1. 获取股票数据
def get_stock_data(ticker, start_date, end_date):
 stock_data = yf.download(ticker, start=start_date, end=end_date)
 return stock_data
 # 2. 数据预处理
def prepare_data(data, look_back=60):
 # 只使用收盘价
close_prices = data['Close'].values
 # 归一化数据
scaler = MinMaxScaler(feature_range=(0, 1))
 scaled_data = scaler.fit_transform(close_prices.reshape(-1, 1))
 # 创建训练数据
X, y = [], []
 for i in range(look_back, len(scaled_data)):
 X.append(scaled_data[i-look_back:i, 0])
 y.append(scaled_data[i, 0])
 X = np.array(X)
 y = np.array(y)
 # 重塑数据为RNN输入格式 [样本数, 时间步长, 特征数]
 X = np.reshape(X, (X.shape[0], X.shape[1], 1))
    return X, y, scaler
 # 3. 创建RNN模型
def create_rnn_model(look_back):
    model = Sequential()
    model.add(SimpleRNN(units=50, return_sequences=True, input_shape=(look_back,
1)))
```

```
    model.add(SimpleRNN(units=50))
    model.add(Dense(units=1))
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model
 # 4. 主函数
def main():
    # 参数设置
    ticker = 'AAPL'  # 股票代码
    start_date = '2020-01-01'
    end_date = '2025-03-16'
    look_back = 60  # 回看天数

    # 获取数据
    stock_data = get_stock_data(ticker, start_date, end_date)

    # 准备数据
    X, y, scaler = prepare_data(stock_data, look_back)

    # 分割训练集和测试集
    train_size = int(len(X) * 0.8)
    X_train, X_test = X[:train_size], X[train_size:]
    y_train, y_test = y[:train_size], y[train_size:]

    # 创建并训练模型
    model = create_rnn_model(look_back)
    history = model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_split=0.1, verbose=1)

    # 进行预测
    train_predict = model.predict(X_train)
    test_predict = model.predict(X_test)

    # 反归一化预测结果
    train_predict = scaler.inverse_transform(train_predict)
    y_train_inv = scaler.inverse_transform([y_train])
    test_predict = scaler.inverse_transform(test_predict)
    y_test_inv = scaler.inverse_transform([y_test])

    # 可视化结果
    plt.figure(figsize=(15, 6))

    # 训练数据
    plt.plot(stock_data.index[look_back:train_size+look_back], y_train_inv[0],
             label='Actual Train Price')
    plt.plot(stock_data.index[look_back:train_size+look_back], train_predict,
             label='Predicted Train Price')

    # 测试数据
test_index = stock_data.index[train_size+look_back:]
 plt.plot(test_index, y_test_inv[0], label='Actual Test Price')
 plt.plot(test_index, test_predict, label='Predicted Test Price')
 plt.title(f'{ticker} Stock Price Prediction')
 plt.xlabel('Date')
 plt.ylabel('Price')
 plt.legend()
 plt.grid(True)
```

```
 plt.show()
 # 计算预测误差
train_rmse = np.sqrt(np.mean((train_predict - y_train_inv[0])**2))
 test_rmse = np.sqrt(np.mean((test_predict - y_test_inv[0])**2))
 print(f'Train RMSE: {train_rmse:.2f}')
 print(f'Test RMSE: {test_rmse:.2f}')
 if __name__ == '__main__':
 main()
```

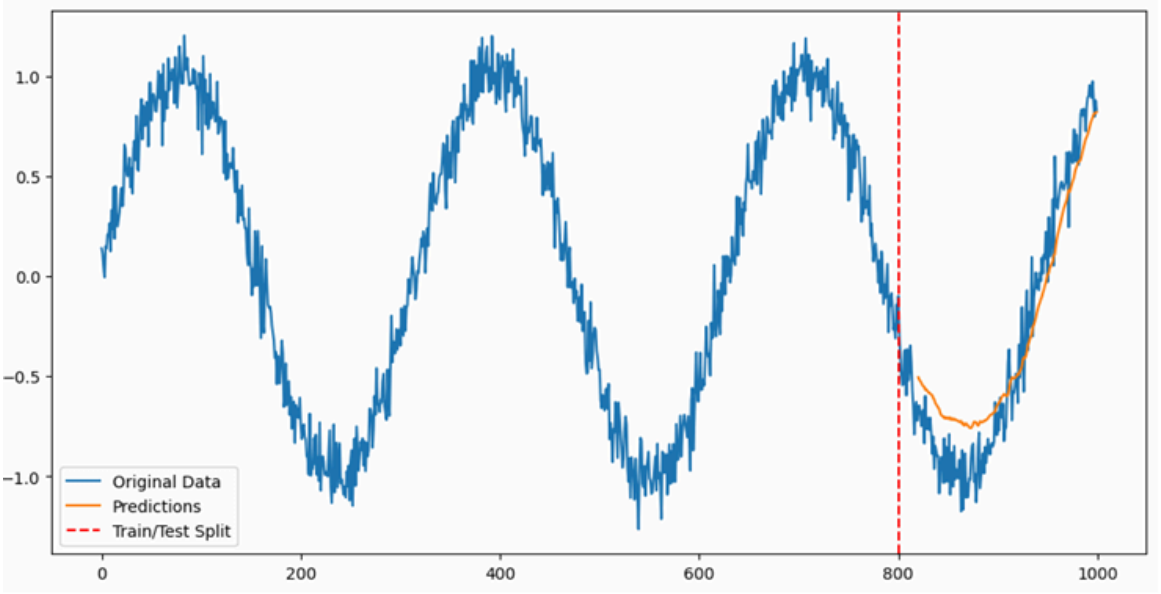


LSTM: Long short term memory（长短时记忆函数）

增加记忆细胞，可以传递前部远处部位信息

```
import torch
 import torch.nn as nn
 import numpy as np
 import matplotlib.pyplot as plt
 # 生成合成数据（正弦波 + 噪声）
def generate_data(seq_length=1000):
 time = np.arange(0, seq_length, 1)
 data = np.sin(0.02 * time) + 0.1 * np.random.randn(seq_length)
 return data
 # 数据预处理
def create_dataset(data, look_back=20):
    X, y = [], []
    for i in range(len(data) - look_back):
        X.append(data[i:(i + look_back)])
        y.append(data[i + look_back])
    return torch.FloatTensor(X).unsqueeze(-1), torch.FloatTensor(y).unsqueeze(-1)
 # 定义LSTM模型
class LSTMModel(nn.Module):
    def __init__(self, input_size=1, hidden_size=64, output_size=1):
        super().__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
    def forward(self, x):
        out, _ = self.lstm(x)  # LSTM层输出
        out = out[:, -1, :]  # 取最后一个时间步
        out = self.fc(out)
        return out
 # 训练配置
def main():
```

```
    # 参数设置
    look_back = 20
    epochs = 50
    learning_rate = 0.001
    # 生成数据
    data = generate_data()
    train_size = int(len(data) * 0.8)
    train_data = data[:train_size]
    test_data = data[train_size:]
    # 创建数据集
    X_train, y_train = create_dataset(train_data, look_back)
    X_test, y_test = create_dataset(test_data, look_back)
    # 初始化模型
    model = LSTMModel()
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    # 训练循环
    for epoch in range(epochs):
        outputs = model(X_train)
        loss = criterion(outputs, y_train)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if (epoch + 1) % 5 == 0:
            print(f'Epoch [{epoch + 1}/{epochs}], Loss: {loss.item():.4f}')
    # 测试预测
with torch.no_grad():
 test_pred = model(X_test)
 test_loss = criterion(test_pred, y_test)
 print(f'Test Loss: {test_loss.item():.4f}')
 # 可视化结果
plt.figure(figsize=(12, 6))
 plt.plot(data, label='Original Data')
 plt.plot(range(train_size + look_back, len(data)),
test_pred.squeeze().numpy(),
 label='Predictions')
 plt.axvline(x=train_size, color='r', linestyle='--', label='Train/Test
Split')
 plt.legend()
 plt.show()
 if __name__ == "__main__":
 main()
```



GRU（门控循环单元）：是一种用于处理序列数据的神经网络模型，它通过两个门来管理信息流，适合自然语言处理等任务。以下是其门结构的详细说明：

GRU的两个门

重置门（Reset Gate）：决定在计算新候选隐藏状态时，忽略多少之前的隐藏状态信息。 比如，如果重置门的值接近0，意味着网络会更多地基于当前输入，而忽略过去的信息。

更新门（Update Gate）：决定保留多少之前的隐藏状态，以及使用多少新计算的候选状态来更新 当前隐藏状态。 如果更新门的值接近1，网络会更多地接受新信息；如果接近0，则更多地保留旧信息。

工作流程

1. 计算重置门：根据当前输入和之前的隐藏状态，决定忽略多少过去信息。
2. 计算更新门：决定新旧信息的混合比例。
3. 生成候选隐藏状态：结合重置门的结果，计算一个新的潜在隐藏状态。
4. 更新隐藏状态：根据更新门，融合之前的隐藏状态和新候选状态，得到最终的隐藏状态。

一个意想不到的细节是，GRU没有像LSTM那样的单独细胞状态，隐藏状态同时作为记忆和输出，这使其 比LSTM更简单，但可能在某些复杂任务中表现稍逊。

2、Transformer架构

1、整体结构

输入：一个序列（例如句子）经过嵌入（Embedding）和位置编码（Positional Encoding）后输 入。

编码器：由N个相同的层组成，每层包含多头自注意力机制（Multi-Head Self-Attention）和前馈神经网络（Feed-Forward Neural Network, FFN）。

解码器：同样由N个相同的层组成，每层包含两个注意力机制（自注意力和编码器-解码器注意力） 以及一个FFN。

输出：解码器的输出经过线性层和Softmax层，生成目标序列。 以下是Transformer架构的直观图（想象一个从左到右的流程）：

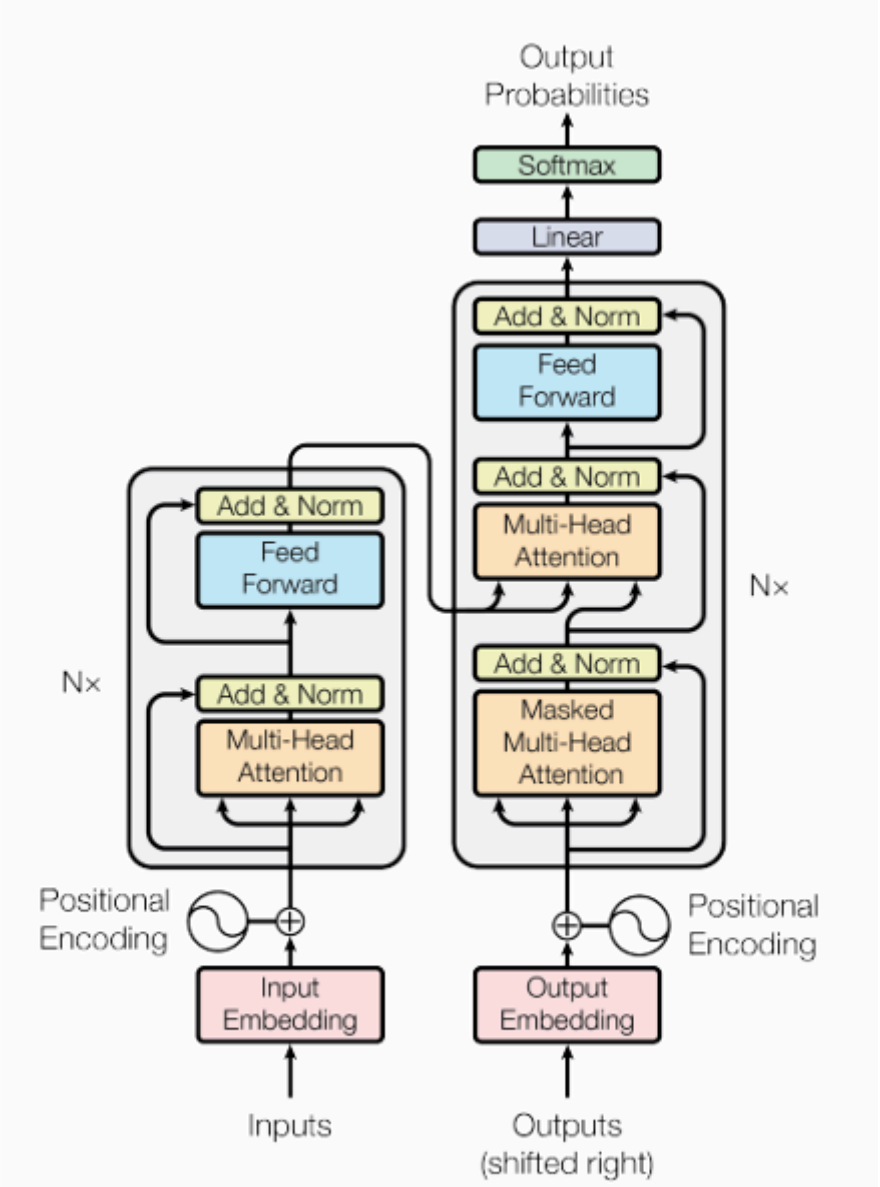


Figure 1: The Transformer - model architecture.

2、Self-attention（自注意力）

机制主要用于捕捉序列中不同位置元素之间的关联，让模型能够关注到序列中各个部分的信息，以下详细描述其计算过程：

1. 输入准备

假设我们有一个输入序列，比如一个句子中的单词序列。每个单词会被表示成一个向量，整个序列可以表示成一个矩阵 X，矩阵的每一行对应序列中一个元素的向量表示。 接下来，我们要 为每个元素创建查询（Query）、键（Key）和值（Value）向量。为了得到这些向量，我们会使用三个不同的权重矩阵W_Q、W_K 、W_V 通过将输入矩阵 X分别与这三个权重矩阵相乘，就能得到查询矩阵 Q、键矩阵 K 和值矩阵 V。具体公式如下：

$$Q = XWq$$

$$K = XWk$$

$$V = XWv$$

2.计算注意力分数我们需要计算每个位置的元素对于其他所有位置元素的注意力分数。这 一步通过计算查询矩阵Q 和键矩阵 K 的转置K^T 的点积来实现。得到的结果是一个矩阵，矩阵中的每个元素表示一个位置的查询向量与另一个位置的键向量的相似度，公式为:

$$AttentionScores = QK^T$$

3.缩放操作 由于点积的结果可能会变得很大，导致梯度在反向传播过程中不稳定，所以 需要对注意力分数进行缩放。我们将注意力分数除以键向量维度 d_k 的平方根，公式如下:

$$AttentionScores = \frac{AttentionScores}{\sqrt{d_k}}$$

4.应用掩码（可选） 在某些场景下，比如在解码器中，为了避免模型看到未来的信 息，我们需要对缩放后的注意力分数应用掩码。掩码通常是一个与注意力分数矩阵 形状相同的矩阵，其中某些位置会被设置为负无穷大（在实际计算中，通常用一个 非常大的负数近似代替），这样在后续的 softmax 操作中，这些位置的权重就会趋 近于 0。 5. 计算注意力权重 对缩放后的注意力分数应用 softmax 函数，将分数转换为概率分 布，得到注意力权重矩阵。softmax 函数会使得每个查询向量对应的所有注意力分 数之和为 1，公式如下:

$$AttentionScores = Attentionweights * V$$

这个输出矩阵包含了序列中每个元素结合了其他元素信息后的新表示，能够更好地 反映序列中元素之间的关联和上下文信息。

3、位置编码

在 Transformer 模型中，位置编码（Positional Encoding） 用于向模型注入序列中各元素的相对或绝对位置信息。由于 Transformer 完全基于注意力机制，没有循环或卷积结构，无法自动捕捉序列顺序，因此需要显式添加位置信息。

以下是其机理的详细说明： Transformer 采用正弦和余弦函数的不同频率生成位置编码

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$
$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

pos：序列中的位置（从 0 开始）。

i：编码向量的维度索引（从 0 到d_model−1）。

d_model：模型维度（如 512）。

关键点： 每个维度对应一个正弦或余弦函数，波长从2π到10000×2π呈几何级数递增。 偶数维度用正弦，奇数维度用余弦，确保不同位置的编码唯一。 比如：翻译 I love you。 ---------->I love you。 先把I love you按独热编码方式写成一组向量 you的位置是1,有四位位置编码： 那就分别计算PE（1，0）、PE（1，1）、PE（1，2）、PE(1,3) 形成它的位置编码矩阵 然后加上 单词的词向量矩阵。