教育部产学合作协同育人项目资助
2022年北京交通大学《深度学习》课程

实验4 循环神经网络实验

主讲教师：丛润民
助　　教：张禹墨

目录

1. 序列数据处理

- 基本处理
- 高级处理

2. 循环神经网络

- 基本原理
- 动手实现
- torch.nn.RNN

3. 长短期记忆网络

- 基本原理
- 动手实现
- torch.nn.LSTM

4. 门控循环单元

- 基本原理
- 动手实现
- torch.nn.GRU

序列数据处理

处理目标：将原始序列数据处理为方便模型运算的序列数据

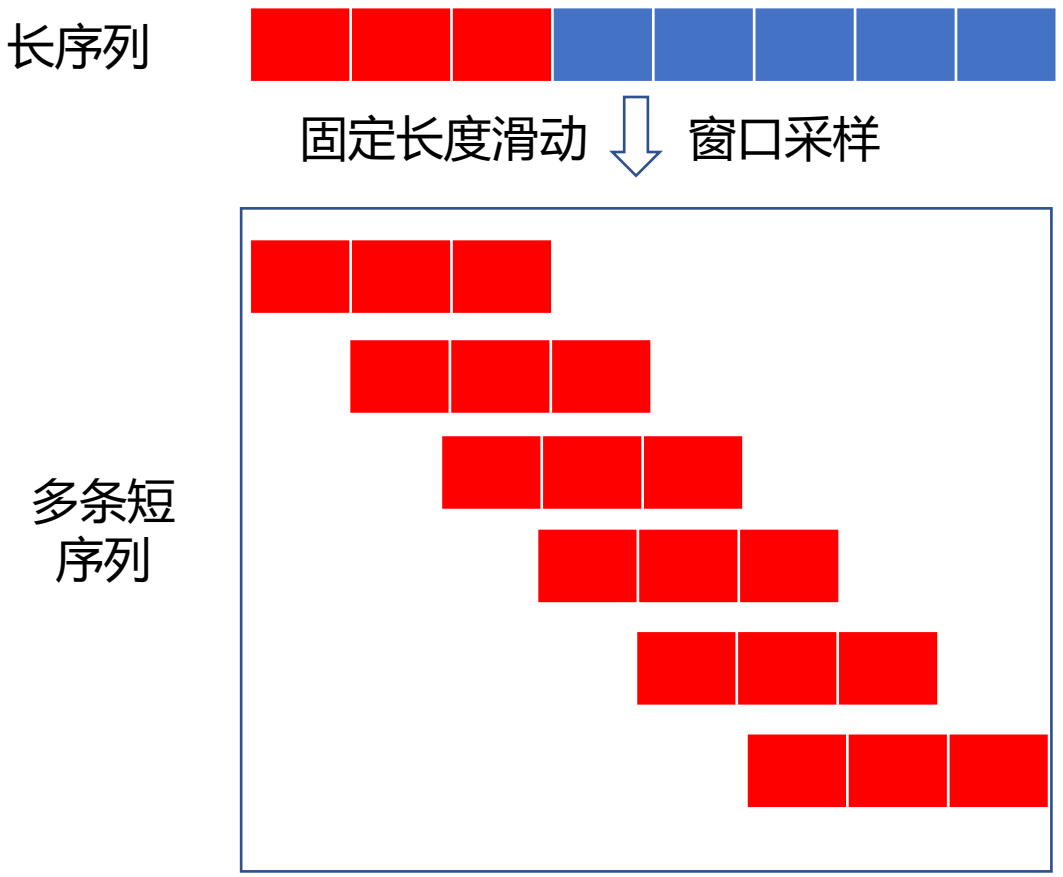
基本处理

- 固定长度滑动窗口
- 数据集划分注意事项

高级处理

- 固定时间跨度滑动窗口
- 不等长序列填充&打包
- 序列重采样

基本数据处理 - 固定长度滑动窗口



固定滑动窗口采样示意图

```
long_seq = raw_df.loc[2, '2017-01']['temperature'].dropna()
long_seq
```

打印长序列long_seq

UTC时间	Temperature
2017-01-01 00:00:00	0.264706
2017-01-01 01:00:00	0.250000
2017-01-01 02:00:00	0.250000
2017-01-01 03:00:00	0.250000
2017-01-01 04:00:00	0.250000
...	
2017-01-31 19:00:00	0.294118
2017-01-31 20:00:00	0.294118
2017-01-31 21:00:00	0.279412
2017-01-31 22:00:00	0.294118
2017-01-31 23:00:00	0.279412

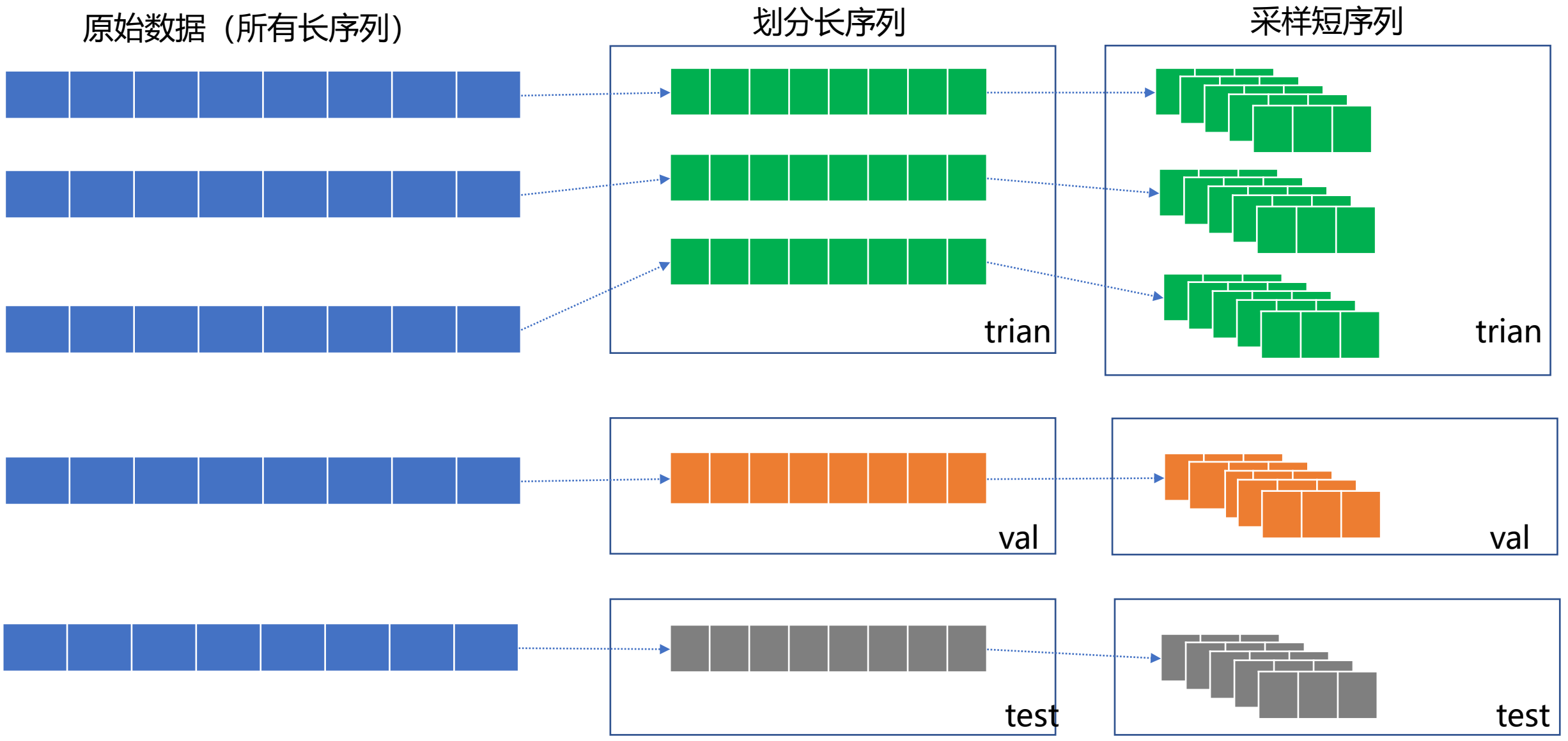
```
window_size = 12
short_seqs = []
for i in range(long_seq.shape[0] - window_size):
    short_seqs.append(long_seq.iloc[i:i+window_size].tolist())
short_seqs = np.array(short_seqs)
print(short_seqs.shape)
```

取长度窗口12，得到多条短序列

	long_seq	window_size	short_seqs
shape	(1, 691)	12	(679, 12)

基本数据处理 - 数据集划分注意事项

先划分原始长序列，再采样短序列



基本数据处理 – 数据集划分示例代码

```
train_set_proportion, val_set_proportion = 0.6, 0.2
total_len = long_seq.shape[0]
train_val_split = int(total_len * train_set_proportion)
val_test_split = int(total_len * (train_set_proportion + val_set_proportion))
train_seq, val_seq, test_seq = long_seq[:train_val_split],\
                               long_seq[train_val_split:val_test_split],\
                               long_seq[val_test_split:]
```

数据划分比例 trian：val：test = 6：2：2

划分长序列

```
train_set = []
for i in range(train_seq.shape[0] - window_size):
    train_set.append(train_seq.iloc[i:i+window_size].tolist())
train_set = np.array(train_set)
print(train_set.shape)
```

采样短序列

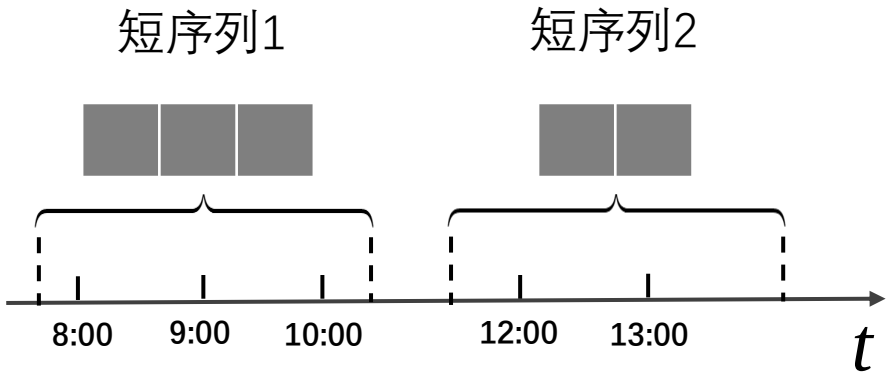
```
(402, 12)
```

注意

✓ 首先对完整、有序的原始长序列按比例划分，再分别进行滑动窗口，采样短序列

✗ 首先用滑动窗口生成多条短序列，再划分短序列。

高级数据处理 – 固定时间窗口滑动



固定时间窗口采样示意图

注：固定时间窗口采样得到的序列的长度不一致，无法直接处理为Tensor，需要进行**填充**

```
long_seq = raw_df.loc[2, '2017-01']['temperature'].dropna()
long_seq
```

UTC时间	Temperature
2017-01-01 00:00:00	0.264706
2017-01-01 01:00:00	0.250000
2017-01-01 02:00:00	0.250000
2017-01-01 03:00:00	0.250000
2017-01-01 04:00:00	0.250000
...	
2017-01-31 19:00:00	0.294118
2017-01-31 20:00:00	0.294118
2017-01-31 21:00:00	0.279412
2017-01-31 22:00:00	0.294118
2017-01-31 23:00:00	0.279412

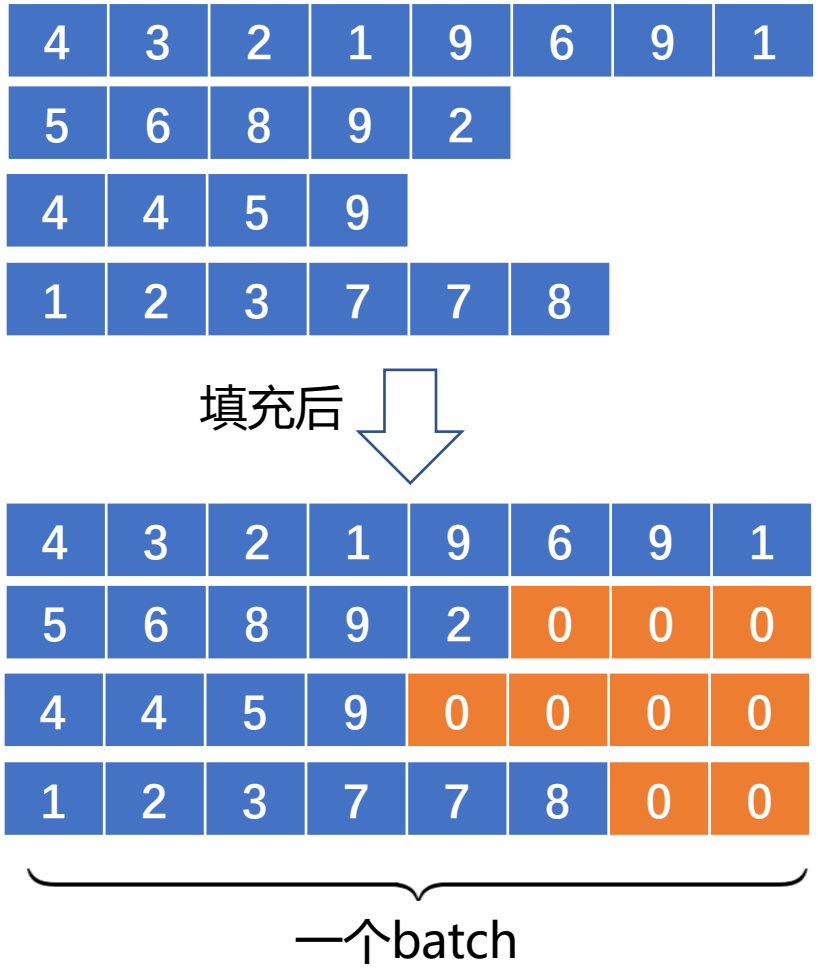
取时间窗口12，得到多条短序列

```
window_size = 12  # (小时)
short_seqs = []
start_time, end_time = long_seq.index.min(), long_seq.index.max() - pd.Timedelta(window_size, 'h')
cur_time = start_time
while cur_time < end_time:
    short_seqs.append(long_seq.loc[cur_time:cur_time + pd.Timedelta(window_size-1, 'h')].tolist())
    cur_time += pd.Timedelta(1, 'h')
```

	long_seq	window_size	short_seqs
shape	(1, 663)	12	(659, 3~12)

所有短序列中最短序列长度为3

高级数据处理 – 不等长序列填充&打包



序列填充示意图

序列填充

```
from itertools import zip_longest

padded_seqs = np.array(list(zip_longest(*short_seqs, fillvalue=0))).transpose()
padded_seqs.shape
```

```
(659, 12)
```

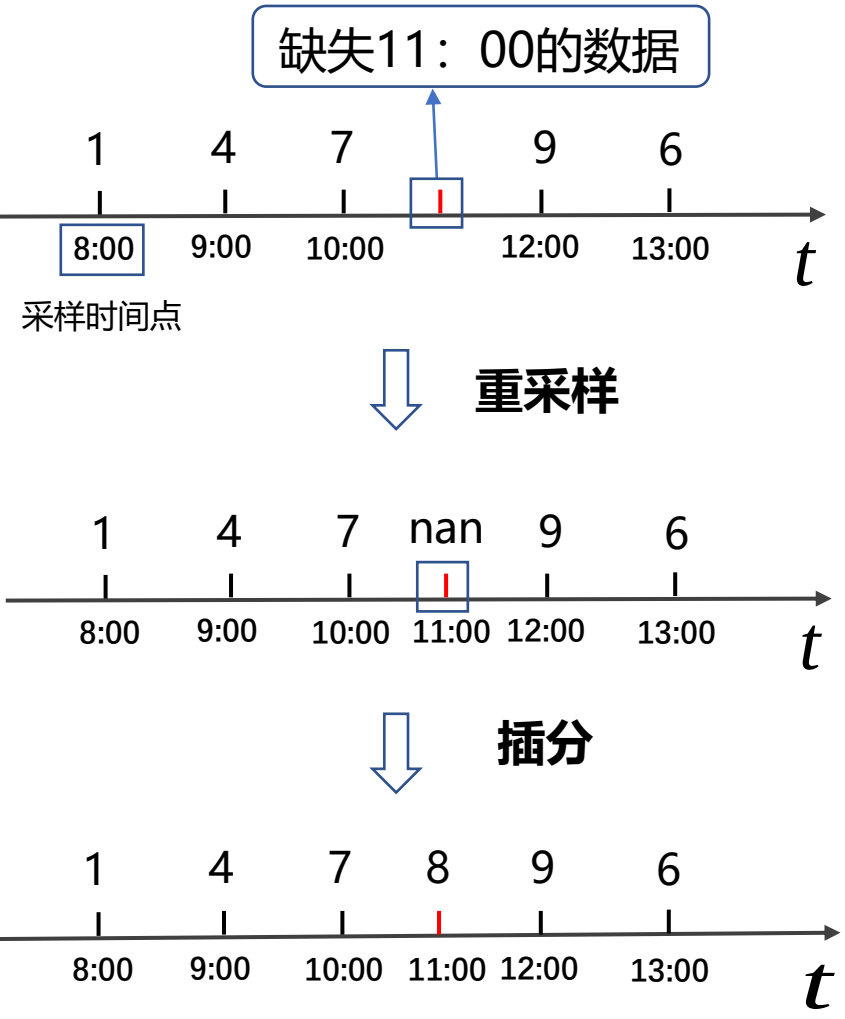
借助itertools中的zip_longest

序列打包

```
import torch
from torch.nn.utils.rnn import pack_padded_sequence

packed_seqs = pack_padded_sequence(torch.tensor(padded_seqs), seq_lengths,
                                   batch_first=True, enforce_sorted=False)
```

需要手动输入每条序列的长度

高级数据处理 – 序列重采样



重采样+插分示意图

重采样

```
start_time, end_time = long_seq.index.min(), long_seq.index.max() - pd.Timedelta(window_size, 'h')
full_index = pd.date_range(start_time, end_time, freq='h')
reindex_seq = long_seq.reindex(full_index)
```

将原始长序列中缺失的时间戳补全，
缺失时间点值被填充为nan

线性插分

```
inter_seq = reindex_seq.interpolate(method='linear', axis=0, limit=2, limit_direction='both')
```

将空缺的值补全，
使用interpolate进行线性插分

目录

1. 序列数据处理

- 基本处理
- 高级处理

2. 循环神经网络

- 基本原理
- 动手实现
- torch.nn.RNN

3. 长短期记忆网络

- 基本原理
- 动手实现
- torch.nn.LSTM

4. 门控循环单元

- 基本原理
- 动手实现
- torch.nn.GRU

循环神经网络 – 基本原理

- 循环神经网络能够处理**任意长度的时序数据**

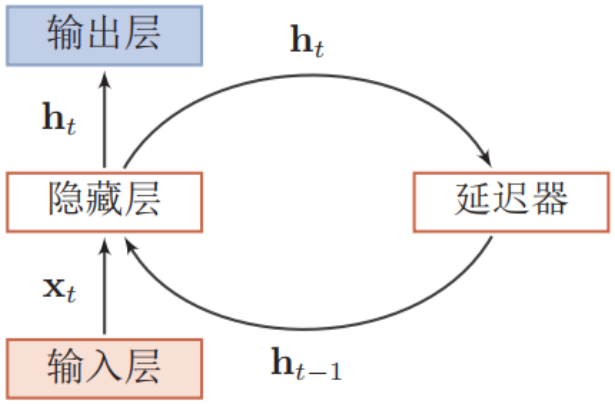
给定一个输入序列，$x_{1:T} = (x_1, x_2, \cdots, x_t, \cdots x_T)$ 其计算公式如下：

$$h_t = f(h_{t-1}, x_t),$$
$$y_t = g(h_t)$$

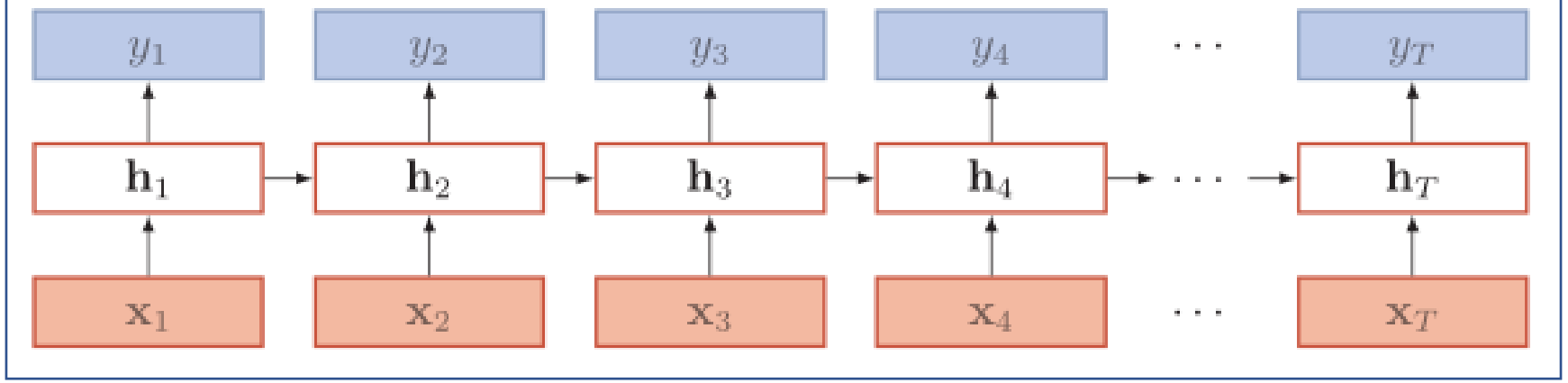
$f(\square), g(\square)$ 实例化

$$h_t = \sigma(W_h x_t + U_h h_{t-1} + b_h),$$
$$y_t = \sigma(W_y h_t + b_y)$$

激活函数可以替换，
如tanh、Relu



画法一



画法二

循环神经网络 – 模型实现 – 初始化参数

```python
class MyRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        """
        :param input_size: 指定输入数据的维度。例如，对于简单的时间序列预测问题，每一步的输入均为一个采样值，因此input_size=1.
        :param hidden_size: 指定隐藏状态的维度。这个值并不受输入和输出控制，但会影响模型的容量。
        :param output_size: 指定输出数据的维度。此值取决于具体的预测要求。例如，对简单的时间序列预测问题，output_size=1.
        """
        super().__init__()
        self.hidden_size = hidden_size

        # 可学习参数的维度设置，可以类比一下全连接网络的实现。其维度取决于输入数据的维度，以及指定的隐藏状态维度。
        self.w_h = nn.Parameter(torch.rand(input_size, hidden_size))
        self.u_h = nn.Parameter(torch.rand(hidden_size, hidden_size))
        self.b_h = nn.Parameter(torch.zeros(hidden_size))

        self.w_y = nn.Parameter(torch.rand(hidden_size, output_size))
        self.b_y = nn.Parameter(torch.zeros(output_size))

        # 准备激活函数。Dropout函数可选。
        self.tanh = nn.Tanh()
        self.leaky_relu = nn.LeakyReLU()

        # 可选：使用性能更好的参数初始化函数
        for param in self.parameters():
            if param.dim() > 1:
                nn.init.xavier_uniform_(param)
```

- 参数维度定义
- 激活函数选取
- 参数初始化

循环神经网络 – 模型实现 – 隐状态更新

```python
def forward(self, x):
    """
    :param x: 输入序列。一般来说，此输入包含三个维度：batch，序列长度，以及每条数据的特征。
    """
    batch_size = x.size(0)
    seq_len = x.size(1)

    # 初始化隐藏状态，一般设为全0。由于是内部新建的变量，需要同步设备位置。
    h = torch.zeros(batch_size, self.hidden_size).to(x.device)
    # RNN实际上只能一步一步处理序列。因此需要用循环迭代。
    y_list = []
    for i in range(seq_len):
        h = self.tanh(torch.matmul(x[:, i, :], self.w_h) +
                      torch.matmul(h, self.u_h) + self.b_h)  # (batch_size, hidden_size)
        y = self.leaky_relu(torch.matmul(h, self.w_y) + self.b_y)  # (batch_size, output_size)
        y_list.append(y)
    # 一般来说，RNN的返回值为最后一步的隐藏状态，以及每一步的输出状态。
    return h, torch.stack(y_list, dim=1)
```

- **隐藏状态初始化**
- **循环迭代更新**
- **注意返回值**

循环神经网络 – 数据处理

数据情况

	sensor_index	UTC time	temperature	humidity	pressure	pm1	pm25	pm10
395	0	2017-01-17 11:00:00	0.323529	0.971508	0.482360	0.068882	0.289673	0.186466
396	0	2017-01-17 12:00:00	0.294118	0.973699	0.484556	0.060778	0.264484	0.174436
397	0	2017-01-17 13:00:00	0.294118	0.973890	0.487776	0.035656	0.156171	0.117293
398	0	2017-01-17 14:00:00	0.279412	0.974271	0.491729	0.038898	0.171285	0.129323
399	0	2017-01-17 15:00:00	0.279412	0.974557	0.497731	0.040519	0.178841	0.133835

选择temperature这一列进行预测

```python
def sliding_window(seq, window_size):
    result = []
    for i in range(len(seq) - window_size):
        result.append(seq[i:i+window_size])
    return result
```

- 固定长度滑动窗口

```python
train_set, test_set = [], []
for sensor_index, group in raw_df.groupby('sensor_index'):
    full_seq = group['temperature'].interpolate(method='linear', limit=3, limit_area='outside')
    full_len = full_seq.shape[0]
    train_seq, test_seq = full_seq.iloc[:int(full_len * 0.8)].to_list(), \
                          full_seq.iloc[int(full_len * 0.8):].to_list()
    train_set += sliding_window(train_seq, window_size=13)
    test_set += sliding_window(test_seq, window_size=13)

# 即使使用了线性插分，依然可能会有缺失值。这里选择直接抛弃。
train_set, test_set = np.array(train_set), np.array(test_set)
train_set, test_set = (item[~np.isnan(item).any(axis=1)] for item in (train_set, test_set))
print(train_set.shape, test_set.shape)
```

- 1）划分长序列
- 2）采样短序列

循环神经网络 – 模型训练

```
device = 'cuda:0'
model = MyRNN(input_size=1, hidden_size=32, output_size=1).to(device)

loss_func = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0005)
```

- 初始化模型、loss函数、优化器

```
def mape(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    non_zero_index = (y_true > 0)
    y_true = y_true[non_zero_index]
    y_pred = y_pred[non_zero_index]

    mape = np.abs((y_true - y_pred) / y_true)
    mape[np.isinf(mape)] = 0
    return np.mean(mape) * 100
```

- 定义和实现指标函数
- 对于回归任务一般选取RMSE, MAE, MAPE
- Scikit-learn中提供了MAE和MSE
- RMSE由MSE求根得到
- MAPE需要自己实现

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$$

Root Mean Square Error

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|$$

Mean Absolute Error

$$MAPE = \frac{100\%}{n}\sum_{i=1}^{n}\left|\frac{\hat{y}_i - y_i}{y_i}\right|$$

Mean Absolute Percentage Error

循环神经网络 – 模型训练

训练函数

```
from sklearn.utils import shuffle

loss_log = []
score_log = []
trained_batches = 0
for epoch in range(4):
    for batch in next_batch(shuffle(train_set), batch_size=64):
        batch = torch.from_numpy(batch).float().to(device)   # (b
        # 使用短序列的前12个值作为历史，最后一个值作为预测值。
        x, label = batch[:, :12], batch[:, -1]

        hidden, out = model(batch.unsqueeze(-1))
        prediction = out[:, -1, :].squeeze(-1)   # (batch)

        loss = loss_func(prediction, label)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        loss_log.append(loss.detach().cpu().numpy().tolist())
        trained_batches += 1

        # 每训练一定数量的batch，就在测试集上测试模型效果。
        if trained_batches % 200 == 0:
```

测试函数

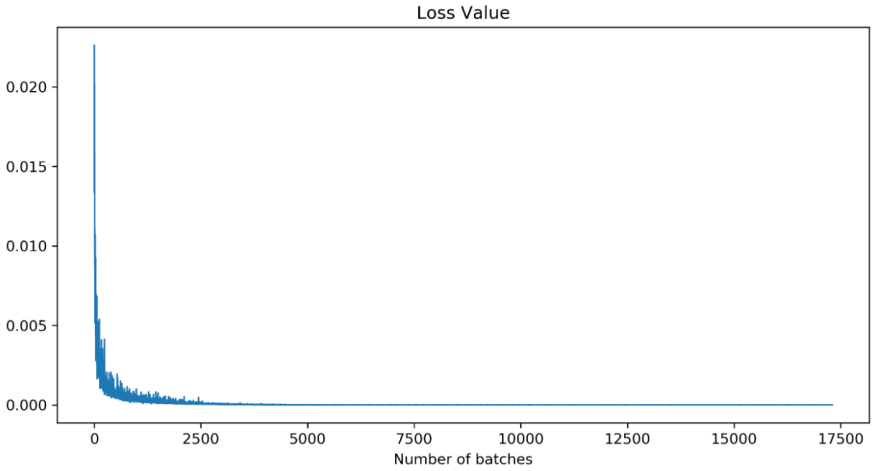
读取batch

```
def next_batch(data, batch_size):
    data_length = len(data)
    num_batches = math.ceil(data_length / batch_size)
    for batch_index in range(num_batches):
        start_index = batch_index * batch_size
        end_index = min((batch_index + 1) * batch_size, data_length)
        yield data[start_index:end_index]
```

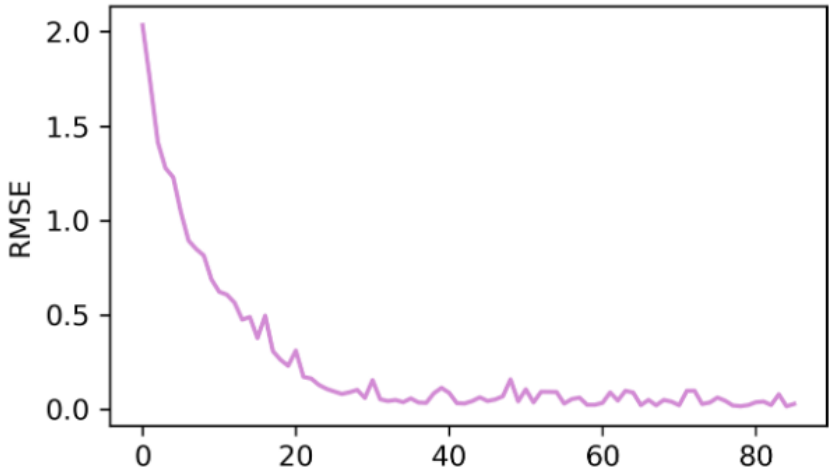
测试函数

```
for batch in next_batch(test_set, batch_size=64):
    batch = torch.from_numpy(batch).float().to(device)
    # (batch, seq_len)
    x, label = batch[:, :12], batch[:, -1]

    hidden, out = model(batch.unsqueeze(-1))
    prediction = out[:, -1, :].squeeze(-1)   # (batch)
    all_prediction.append(prediction.detach().cpu().numpy())
all_prediction = np.concatenate(all_prediction)
all_label = test_set[:, -1]
# 进行反归一化操作。
all_prediction = dataset.denormalize(all_prediction, 'temperature')
all_label = dataset.denormalize(all_label, 'temperature')
# 计算测试指标。
rmse_score = math.sqrt(mse(all_label, all_prediction))
mae_score = mae(all_label, all_prediction)
mape_score = mape(all_label, all_prediction)
score_log.append([rmse_score, mae_score, mape_score])
print('RMSE: %.4f, MAE: %.4f, MAPE: %.4f' %
      (rmse_score, mae_score, mape_score))
```

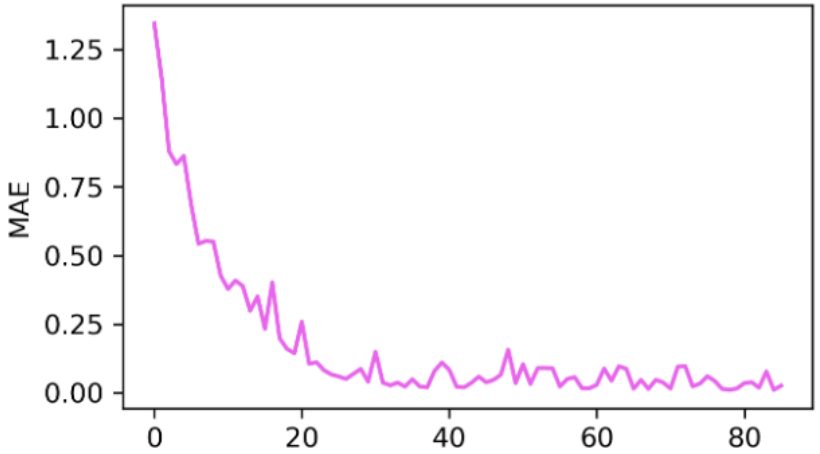
循环神经网络 – 训练结果可视化（使用自己实现的模型）



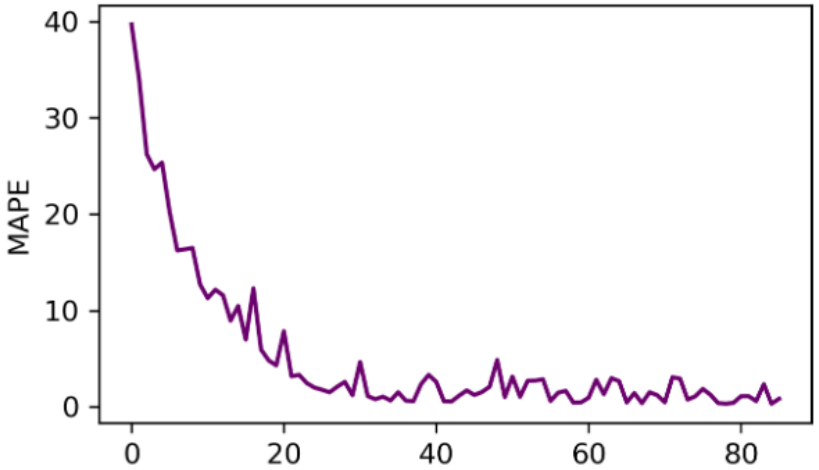
Loss变化



RMSE变化



MAE变化



MAPE变化

循环神经网络 – torch.nn.RNN

输入

Inputs: input, h_0

- **input** of shape (*seq_len, batch, input_size*): tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- **h_0** of shape (*num_layers*num_directions, batch, hidden_size*): tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided. If the RNN is bidirectional, num_directions should be 2, else it should be 1.

输出

Outputs: output, h_n

- **output** of shape (*seq_len, batch, num_directions*hidden_size*): tensor containing the output features (*h_t*) from the last layer of the RNN, for each *t*. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence.
  For the unpacked case, the directions can be separated using `output.view(seq_len, batch, num_directions, hidden_size)`, with forward and backward being direction *0* and *1* respectively. Similarly, the directions can be separated in the packed case.
- **h_n** of shape (*num_layers*num_directions, batch, hidden_size*): tensor containing the hidden state for *t = seq_len*.
  Like *output*, the layers can be separated using `h_n.view(num_layers, num_directions, batch, hidden_size)`.

循环神经网络 – torch.nn.RNN

参数

Parameters

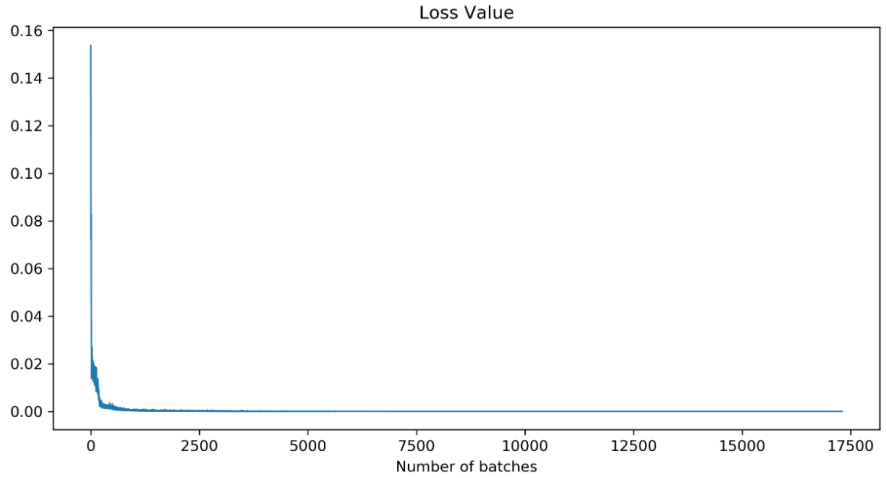
- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
- **nonlinearity** – The non-linearity to use. Can be either `'tanh'` or `'relu'`. Default: `'tanh'`
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as *(batch, seq, feature)*. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each RNN layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional RNN. Default: `False`

例子

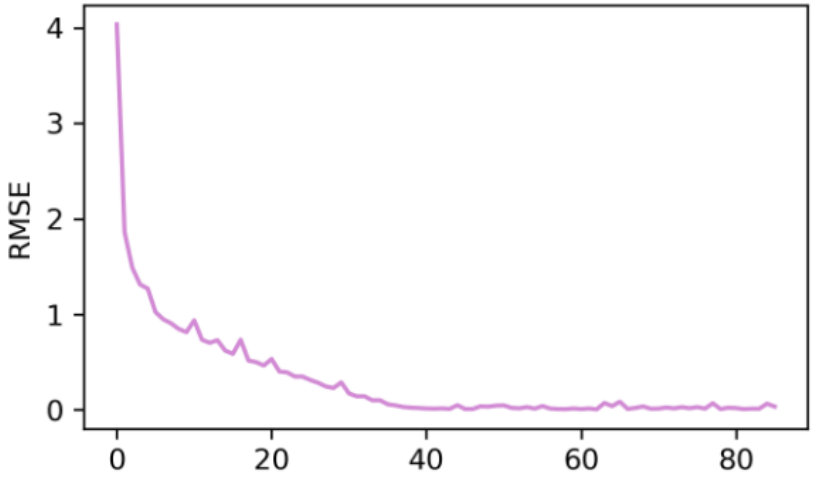
Examples:

```
>>> rnn = nn.RNN(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> output, hn = rnn(input, h0)
```

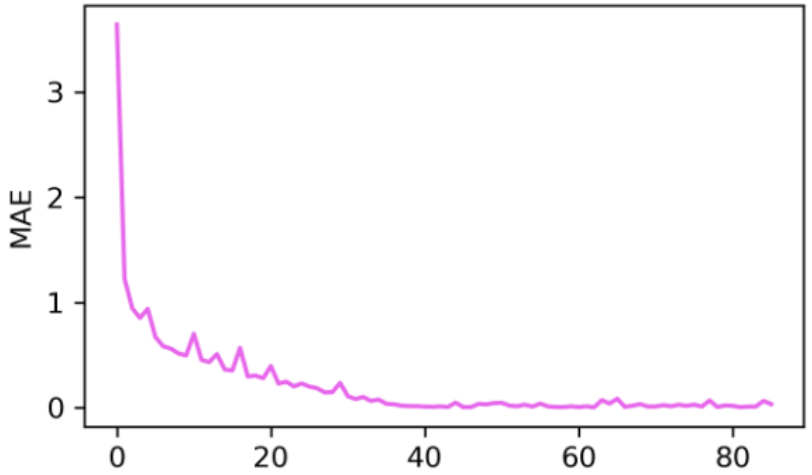
循环神经网络 – 训练结果可视化（使用Pytorch自带的模型）



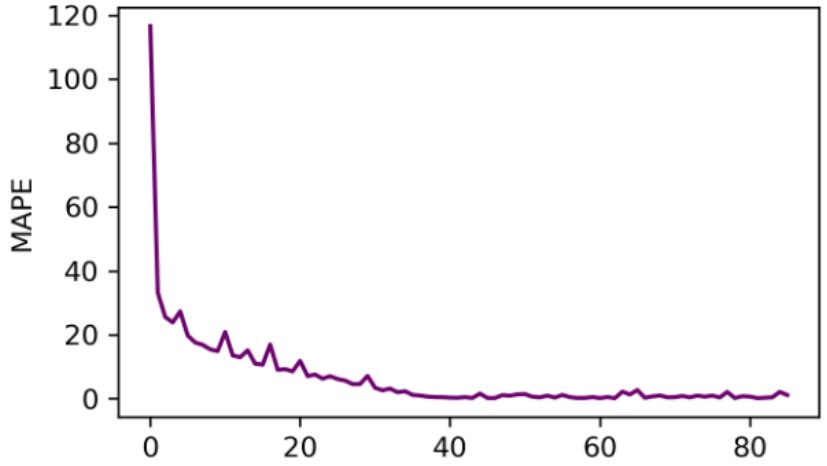
Loss变化



RMSE变化



MAE变化



MAPE变化

目录

1. 序列数据处理

- 基本处理
- 高级处理

2. 循环神经网络

- 基本原理
- 动手实现
- torch.nn.rnn

3. 长短期记忆网络

- 基本原理
- 动手实现
- torch.nn.LSTM

4. 门控循环单元

- 基本原理
- 动手实现
- torch.nn.GRU

长短期记忆网络LSTM – 基本原理

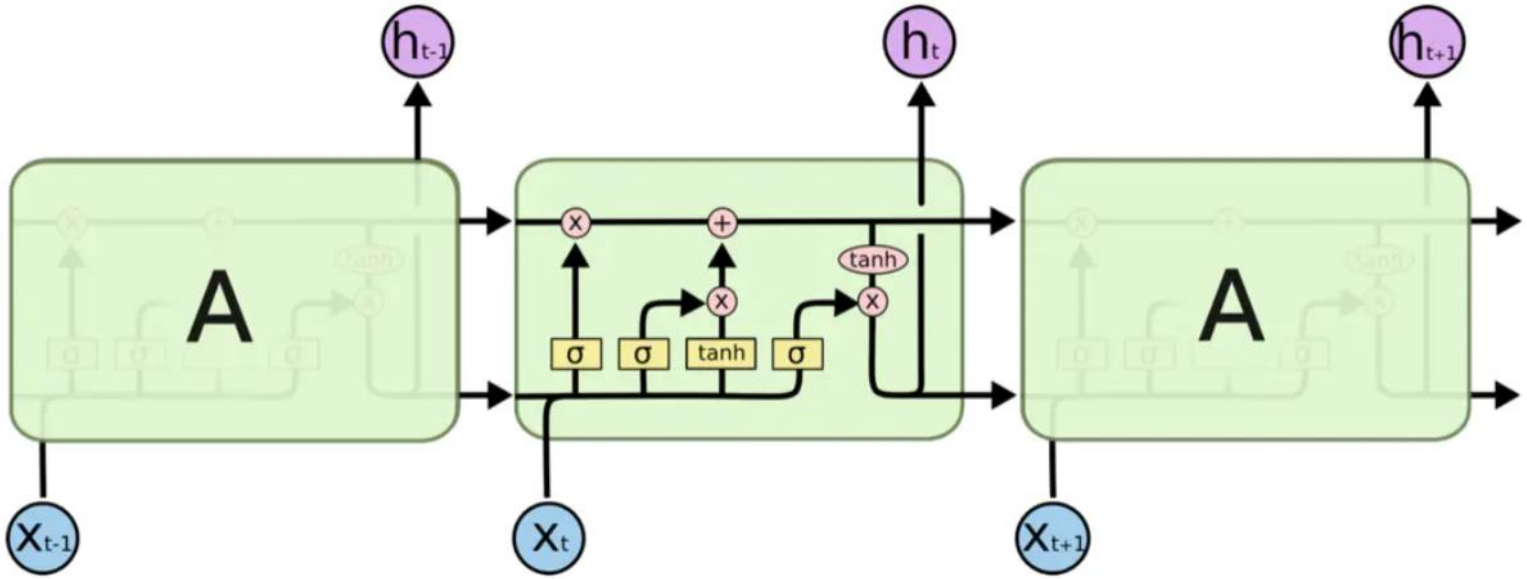
LSTM：Long Short Term Memory networks

- 一种特殊形式的RNN
- 解决长程依赖问题

短程依赖，普通的RNN能够解决

长程依赖，借助LSTM解决

长短期记忆网络 – 基本原理

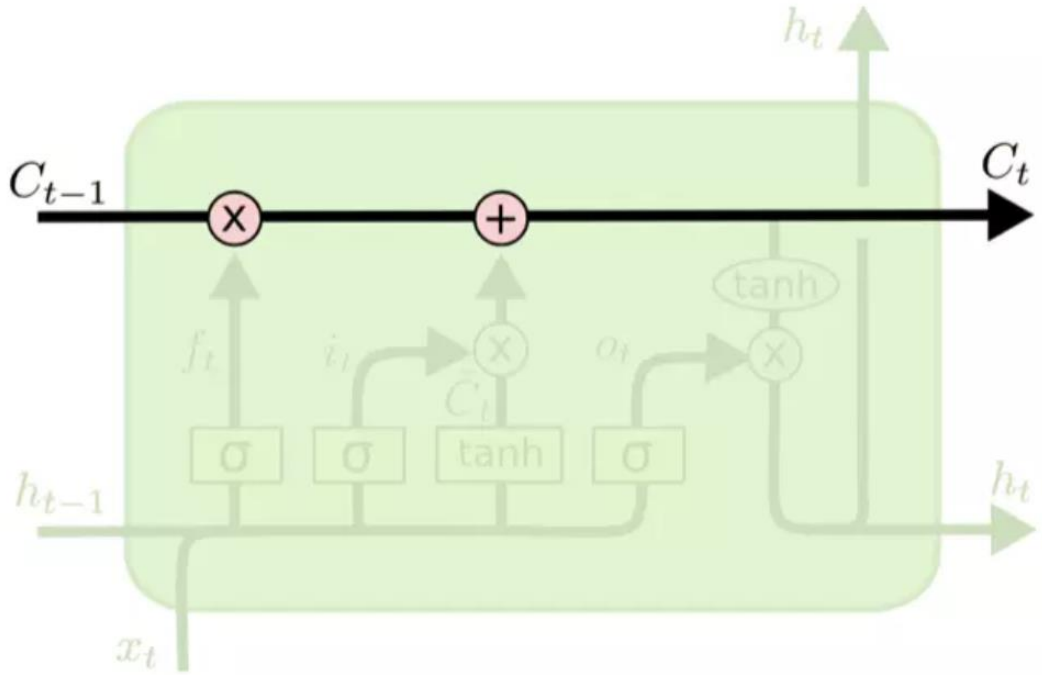


LSTM结构

$$f_t = \sigma_g\left(W_f x_t + U_f h_{t-1} + b_f\right)$$
$$i_t = \sigma_g\left(W_i x_t + U_i h_{t-1} + b_i\right)$$
$$o_t = \sigma_g\left(W_o x_t + U_o h_{t-1} + b_o\right)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c\left(W_c x_t + U_c h_{t-1} + b_c\right)$$
$$h_t = o_t \circ \sigma_h\left(c_t\right)$$

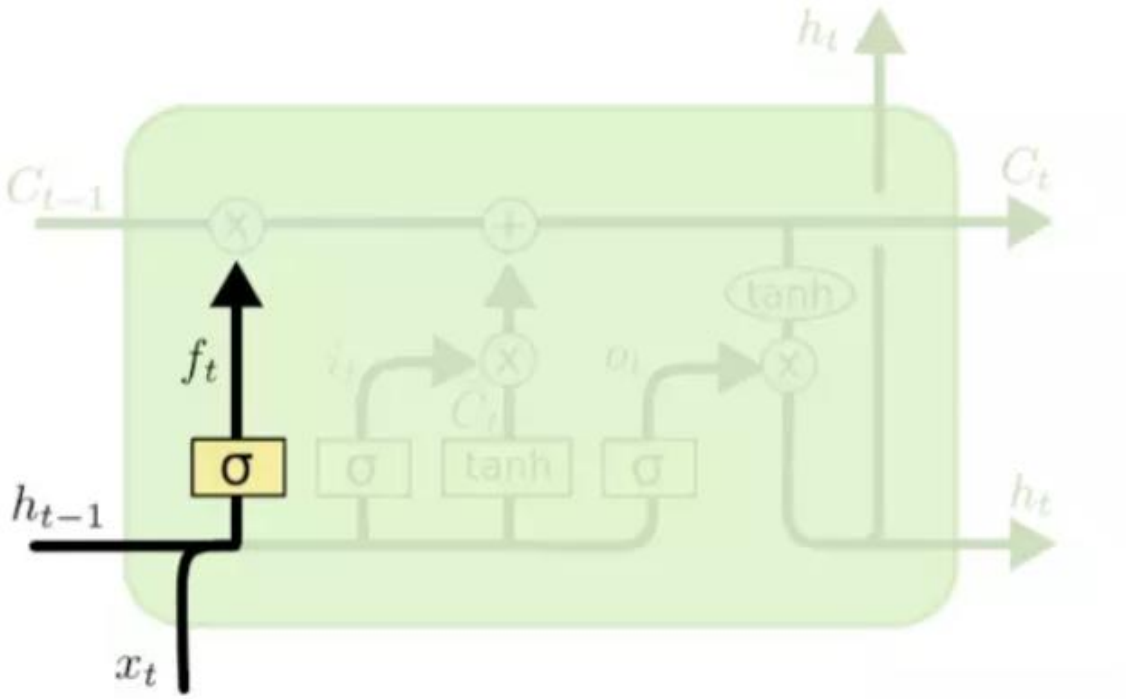
LSTM公式

长短期记忆网络 – 基本原理



- LSTM的核心是细胞状态，用贯穿细胞的水平线表示

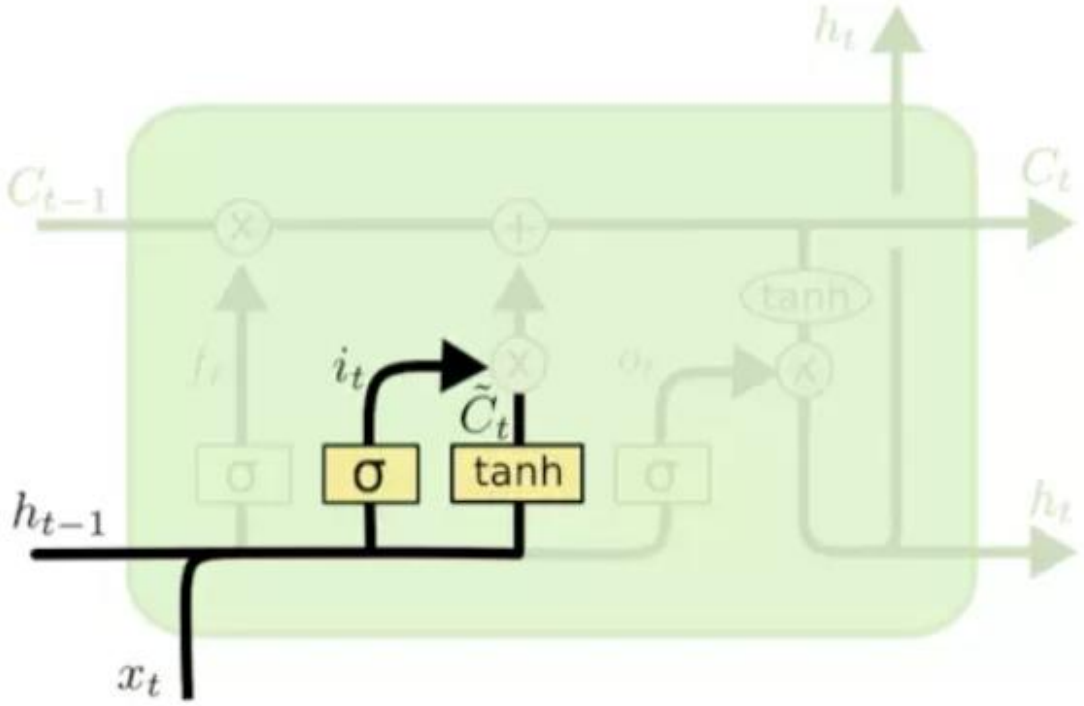
长短期记忆网络 – 基本原理



- 1）计算遗忘门，决定细胞状态需要舍弃哪部分无用信息

$$f_t = \sigma_g\left(W_f x_t + U_f h_{t-1} + b_f\right)$$

长短期记忆网络 – 基本原理



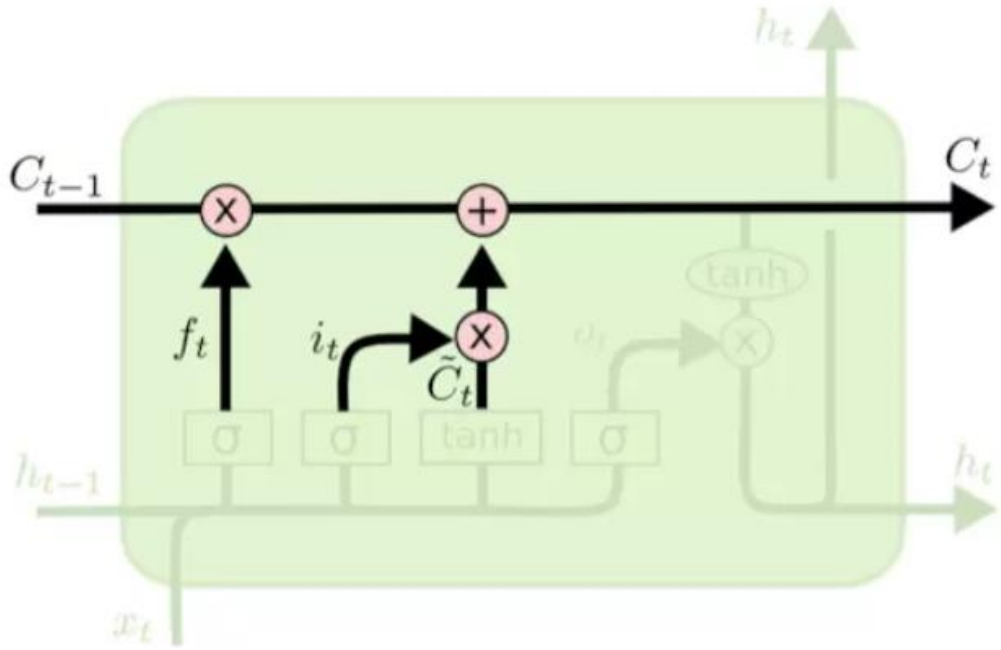
- 2）计算输入门，决定细胞状态需要添加哪些有用信息

$$i_t = \sigma_g\left(W_i x_t + U_i h_{t-1} + b_i\right)$$

- 3）计算候选细胞状态

$$\tilde{c}_t = \sigma_c\left(W_c x_t + U_c h_{t-1} + b_c\right)$$

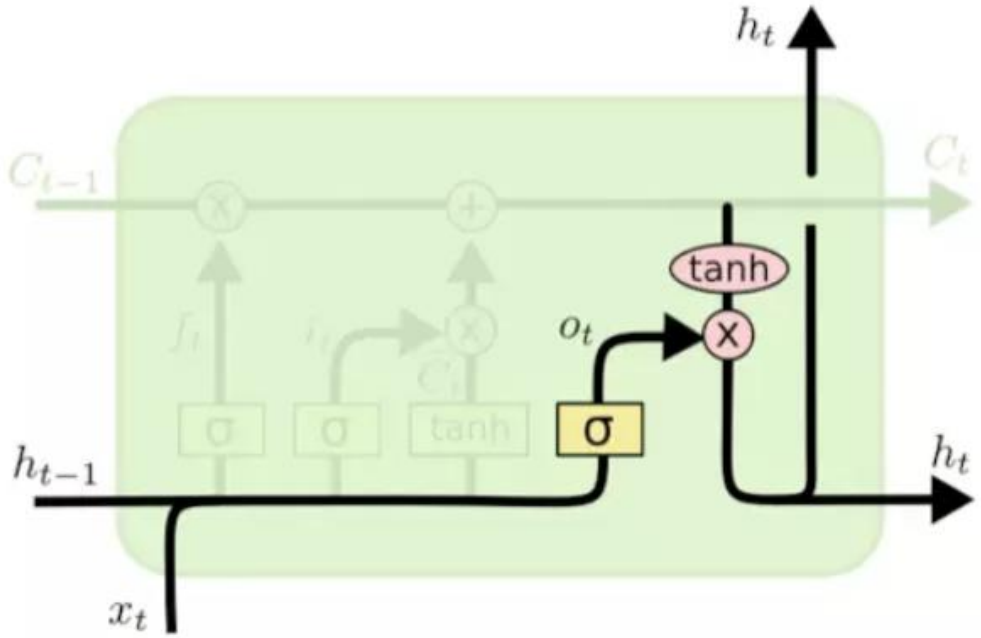
长短期记忆网络 – 基本原理



- 4）更新细胞状态

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$

长短期记忆网络 – 基本原理



- 5）计算输出门，控制细胞状态中哪些信息被输出：

$$o_t = \sigma_g\left(W_o x_t + U_o h_{t-1} + b_o\right)$$

- 6）计算输出隐状态：

$$h_t = o_t \circ \sigma_h\left(c_t\right)$$

长短期记忆网络 – 模型实现 – 初始化参数

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

        self.w_f = nn.Parameter(torch.rand(input_size, hidden_size))
        self.u_f = nn.Parameter(torch.rand(hidden_size, hidden_size))
        self.b_f = nn.Parameter(torch.zeros(hidden_size))

        self.w_i = nn.Parameter(torch.rand(input_size, hidden_size))
        self.u_i = nn.Parameter(torch.rand(hidden_size, hidden_size))
        self.b_i = nn.Parameter(torch.zeros(hidden_size))

        self.w_o = nn.Parameter(torch.rand(input_size, hidden_size))
        self.u_o = nn.Parameter(torch.rand(hidden_size, hidden_size))
        self.b_o = nn.Parameter(torch.zeros(hidden_size))

        self.w_c = nn.Parameter(torch.rand(input_size, hidden_size))
        self.u_c = nn.Parameter(torch.rand(hidden_size, hidden_size))
        self.b_c = nn.Parameter(torch.zeros(hidden_size))

        self.sigmoid = nn.Sigmoid()
        self.tanh = nn.Tanh()

        for param in self.parameters():
            if param.dim() > 1:
                nn.init.xavier_uniform_(param)
```

$$f_t = \sigma_g\left(W_f x_t + U_f h_{t-1} + b_f\right)$$

$$i_t = \sigma_g\left(W_i x_t + U_i h_{t-1} + b_i\right)$$

$$o_t = \sigma_g\left(W_o x_t + U_o h_{t-1} + b_o\right)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c\left(W_c x_t + U_c h_{t-1} + b_c\right)$$

$$h_t = o_t \circ \sigma_h\left(c_t\right)$$

长短期记忆网络 – 模型实现

```
def forward(self, x):
    batch_size = x.size(0)
    seq_len = x.size(1)

    # 需要初始化隐藏状态和细胞状态
    h = torch.zeros(batch_size, self.hidden_size).to(x.device)
    c = torch.zeros(batch_size, self.hidden_size).to(x.device)
    y_list = []
    for i in range(seq_len):
        forget_gate = self.sigmoid(torch.matmul(x[:, i, :], self.w_f) +
                                   torch.matmul(h, self.u_f) + self.b_f)
        # (batch_size, hidden_size)
        input_gate = self.sigmoid(torch.matmul(x[:, i, :], self.w_i) +
                                  torch.matmul(h, self.u_i) + self.b_i)
        output_gate = self.sigmoid(torch.matmul(x[:, i, :], self.w_o) +
                                   torch.matmul(h, self.u_o) + self.b_o)
        # 这里可以看到各个门的运作方式。
        # 三个门均通过hadamard积作用在每一个维度上。
        c = forget_gate * c +
        input_gate * self.tanh(torch.matmul(x[:, i, :], self.w_c) +
                               torch.matmul(h, self.u_c) +
                                            self.b_c)
        h = output_gate * self.tanh(c)
        y_list.append(h)
    return torch.stack(y_list, dim=1), (h, c)
```

$$f_t = \sigma_g\left(W_f x_t + U_f h_{t-1} + b_f\right)$$

$$i_t = \sigma_g\left(W_i x_t + U_i h_{t-1} + b_i\right)$$

$$o_t = \sigma_g\left(W_o x_t + U_o h_{t-1} + b_o\right)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c\left(W_c x_t + U_c h_{t-1} + b_c\right)$$

$$h_t = o_t \circ \sigma_h\left(c_t\right)$$

- 初始化隐藏状态hidden_state 为全0向量
- 按照公式循环迭代更新hidden_state，计算输出
- 返回最后一步的隐藏状态和每一步的输出
- 注意不同地方的激活函数的选取

长短期记忆网络 – 优化模型实现思路

核心思想：合并矩阵运算，提高并行性

1）各门控计算优化，以遗忘门为例：

$$f_t = \sigma_g\left(W_f x_t + U_f h_{t-1} + b_f\right) = \sigma_g\left(W_f'\left(x_t \| h_{t-1}\right) + b_f\right)$$

遗忘门中的两个矩阵运算实际上可以合并为一个，$W_f' \in \square^{\text{(input_size+hidden_size), hidden_size}}$

相当于拼合 $W_f \in \square^{\text{input_size, hidden_size}}$ 和 $U_f \in \square^{\text{hidden_size, hidden_size}}$，此时，门的计算和全连接网络一致，可使用nn.Linear替代：

```
self.forget_gate = nn.Linear(input_size+hidden_size, hidden_size)
f_g = self.sigmoid(self.hidden_gate(torch.cat([x, h], dim=-1)))
```

长短期记忆网络 – 优化模型实现思路

核心思想：合并矩阵运算，提高并行性

2）所有门控一起计算：

- 三个门控单元和 $\tilde{C}_t$ 的计算公式高度一致

一步实现所有门的计算，再将结果拆分后进行激活

```
self.gates = nn.Linear(input_size+hidden_size, hidden_size * 4)
f_g, i_g, o_g, c_g = self.gates(torch.cat([x, h], dim=-1)).chunk(chunks=4, dim=-1)
```

长短期记忆网络 – 优化模型实现

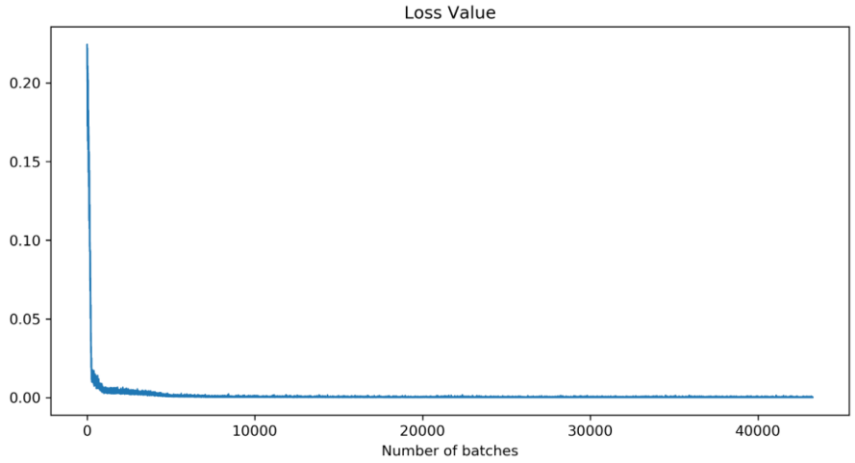
```python
class MyLSTM(nn.Module):
    def __init__(self, input_size, hidden_size):
        super().__init__()
        self.hidden_size = hidden_size

        self.gates = nn.Linear(input_size+hidden_size, hidden_size * 4)
        self.sigmoid = nn.Sigmoid()
        self.tanh = nn.Tanh()

        for param in self.parameters():
            if param.dim() > 1:
                nn.init.xavier_uniform_(param)

    def forward(self, x):
        batch_size = x.size(0)
        seq_len = x.size(1)

        h, c = (torch.zeros(batch_size, self.hidden_size).to(x.device) for _ in range(2))
        y_list = []
        for i in range(seq_len):
            forget_gate, input_gate, \
            output_gate, candidate_cell = self.gates(torch.cat([x[:, i, :], h], dim=-1)).chunk(4, -1)
            forget_gate, input_gate, output_gate = (self.sigmoid(g)
                                                    for g in (forget_gate, input_gate, output_gate))
            c = forget_gate * c + input_gate * self.tanh(candidate_cell)
            h = output_gate * self.tanh(c)
            y_list.append(h)
        return torch.stack(y_list, dim=1), (h, c)
```

核心思想：

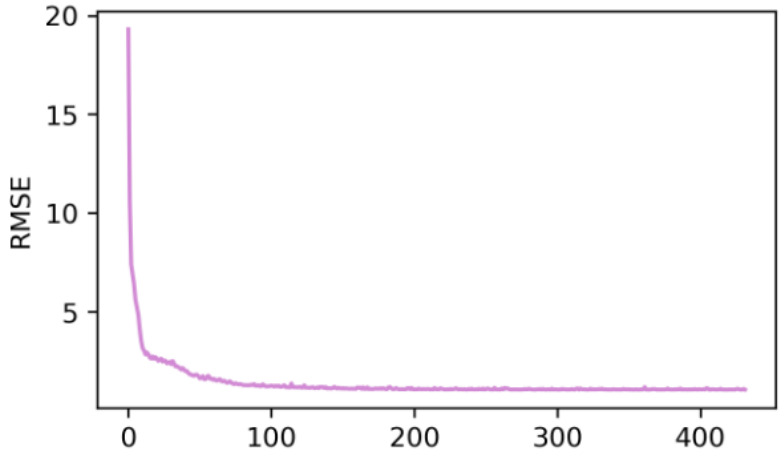
提高并行性

合并矩阵运算

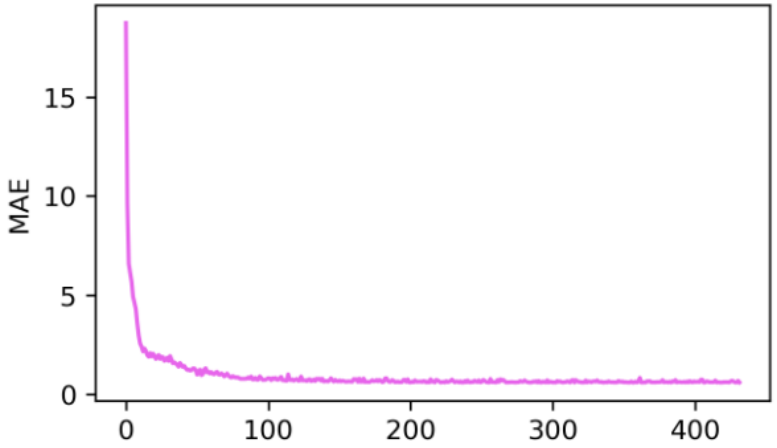
长短期记忆网络 – 训练结果可视化（使用自己实现的模型）



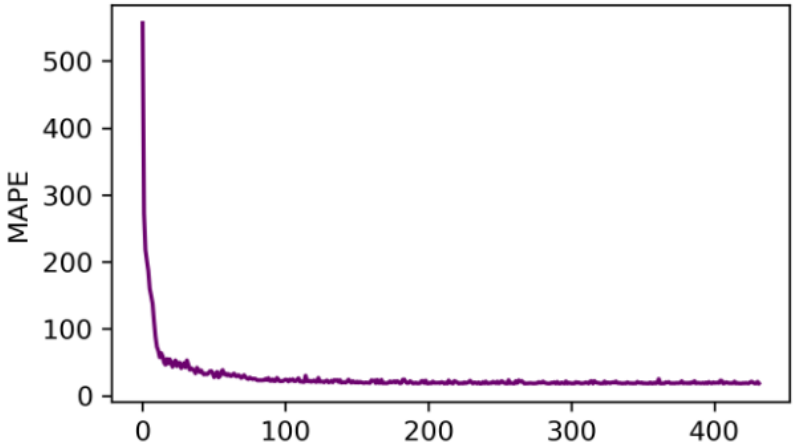
Loss变化



RMSE变化



MAE变化



MAPE变化

长短期记忆网络 – torch.nn.LSTM

输入

Inputs: input, (h_0, c_0)

- **input** of shape (*seq_len, batch, input_size*): tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- **h_0** of shape (*num_layers*num_directions, batch, hidden_size*): tensor containing the initial hidden state for each element in the batch. If the LSTM is bidirectional, num_directions should be 2, else it should be 1.
- **c_0** of shape (*num_layers*num_directions, batch, hidden_size*): tensor containing the initial cell state for each element in the batch.

  If (*h_0, c_0*) is not provided, both **h_0** and **c_0** default to zero.

输出

Outputs: output, (h_n, c_n)

- **output** of shape (*seq_len, batch, num_directions*hidden_size*): tensor containing the output features (*h_t*) from the last layer of the LSTM, for each *t*. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence.

  For the unpacked case, the directions can be separated using `output.view(seq_len, batch, num_directions, hidden_size)`, with forward and backward being direction *0* and *1* respectively. Similarly, the directions can be separated in the packed case.
- **h_n** of shape (*num_layers*num_directions, batch, hidden_size*): tensor containing the hidden state for *t = seq_len*.

  Like *output*, the layers can be separated using `h_n.view(num_layers, num_directions, batch, hidden_size)` and similarly for *c_n*.
- **c_n** of shape (*num_layers*num_directions, batch, hidden_size*): tensor containing the cell state for *t = seq_len*.

长短期记忆网络 – torch.nn.LSTM

参数

Parameters

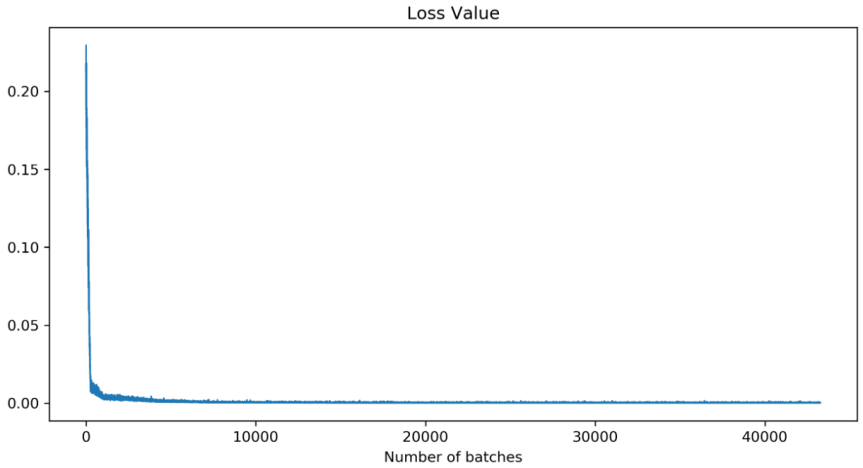
- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (batch, seq, feature). Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional LSTM. Default: `False`

例子

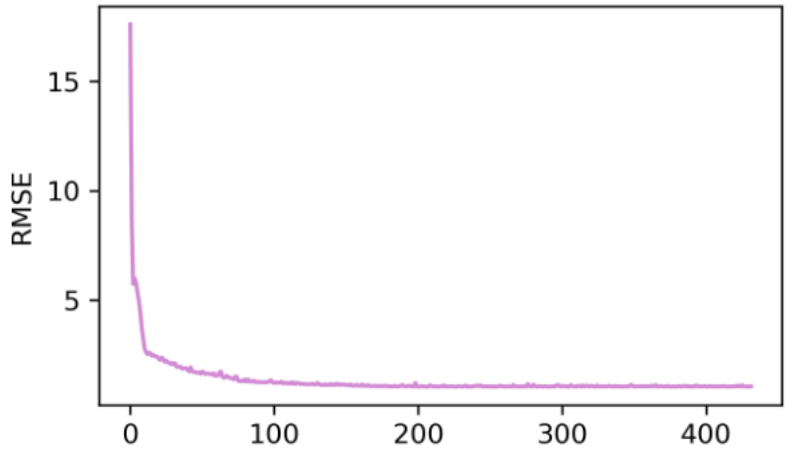
Examples:

```
>>> rnn = nn.LSTM(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> c0 = torch.randn(2, 3, 20)
>>> output, (hn, cn) = rnn(input, (h0, c0))
```

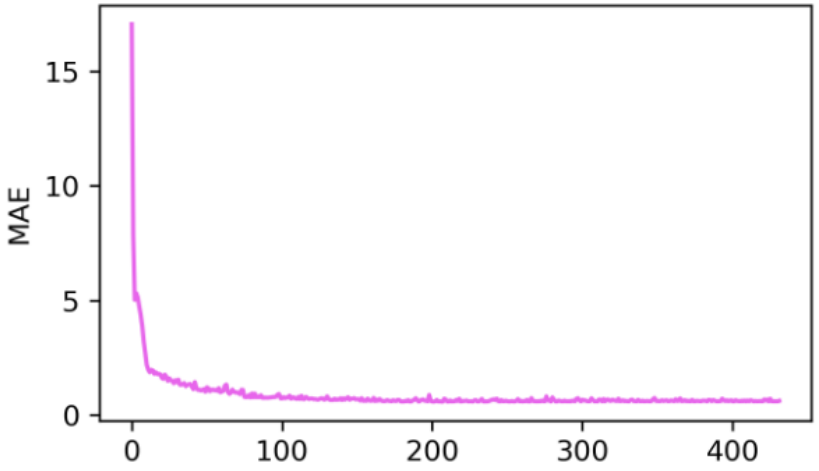
长短期记忆网络 – 训练结果可视化（使用Pytorch自带模型）



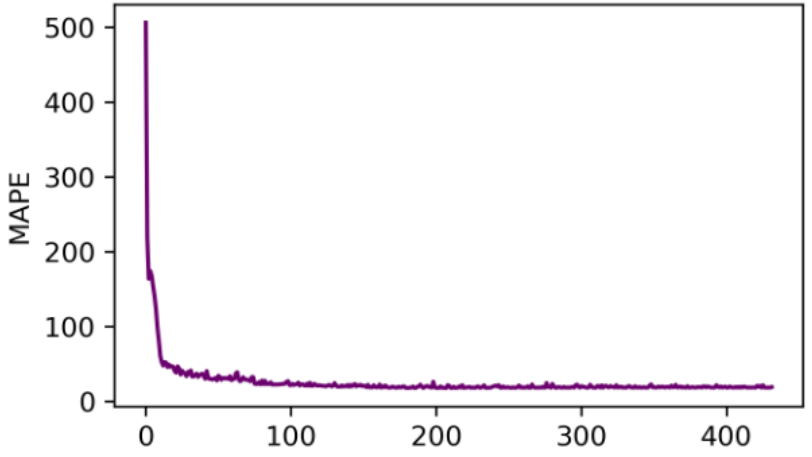
Loss变化



RMSE变化



MAE变化



MAPE变化

目录

1. 序列数据处理

- 基本处理
- 高级处理

2. 循环神经网络

- 基本原理
- 动手实现
- torch.nn.RNN

3. 长短期记忆网络

- 基本原理
- 动手实现
- torch.nn.LSTM

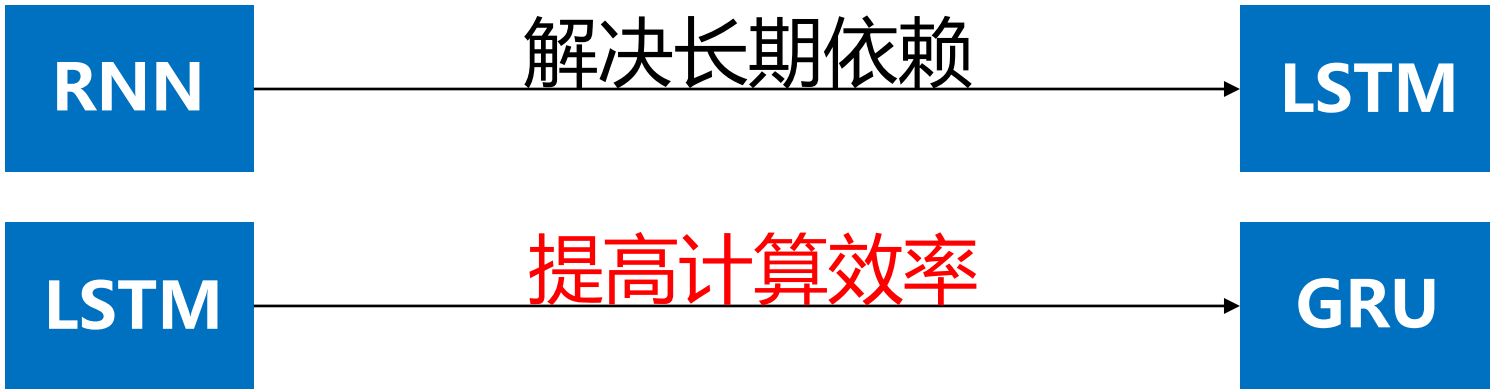
4. 门控循环单元

- 基本原理
- 动手实现
- torch.nn.GRU

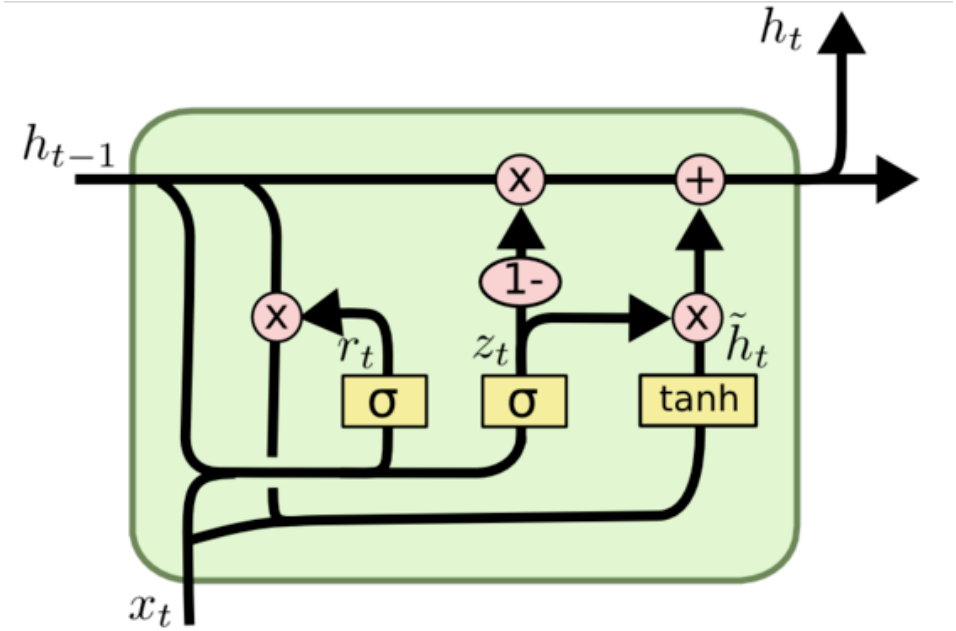
门控循环单元 – 基本原理

GRU：Gate Recurrent Unit

- 一种特殊形式的RNN
- 相比LSTM，简化门控机制，提高计算效率
- 门控：
  - 重置门 r_t ：控制遗忘多少之前时刻的信息
  - 更新门 z_t ：控制保留多少当前时刻的信息



门控循环单元 – 基本原理



GRU结构

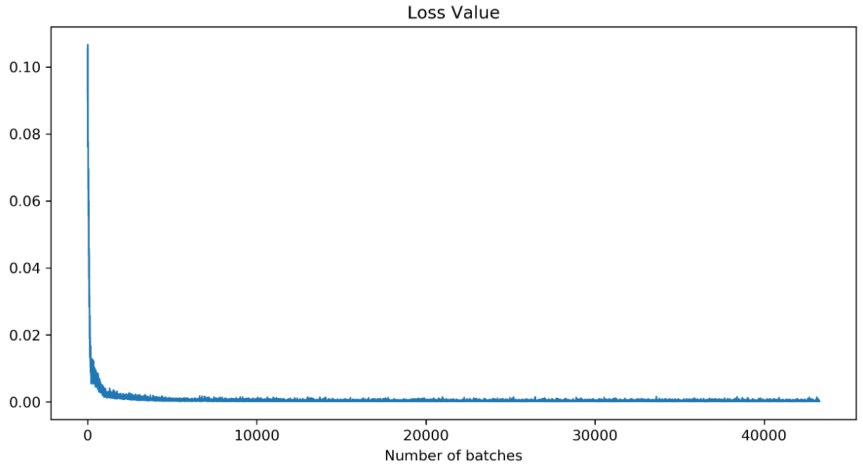
$$z_t = \sigma_g(W_z x_t + U_z h_{t-1} + b_z)$$
$$r_t = \sigma_g(W_r x_t + U_r h_{t-1} + b_r)$$
$$\hat{h}_t = \phi_h(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t$$

GRU公式

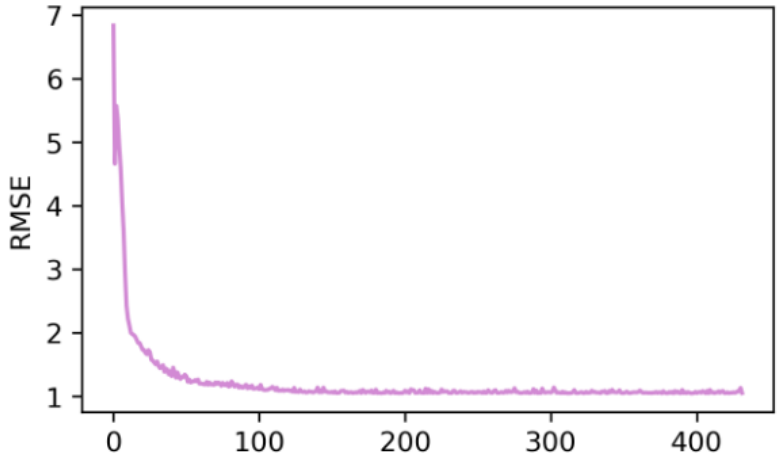
门控循环单元 – 模型实现（优化后）

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

        self.gates = nn.Linear(input_size+hidden_size, hidden_size*2)
        # 用于计算candidate hidden state
        self.hidden_transform = nn.Linear(input_size+hidden_size, hidden_size)

        self.sigmoid = nn.Sigmoid()
        self.tanh = nn.Tanh()

        for param in self.parameters():
            if param.dim() > 1:
                nn.init.xavier_uniform_(param)

    def forward(self, x):
        batch_size = x.size(0)
        seq_len = x.size(1)

        h = torch.zeros(batch_size, self.hidden_size).to(x.device)
        y_list = []
        for i in range(seq_len):
            update_gate, reset_gate = self.gates(torch.cat([x[:, i, :], h], dim=-1)).chunk(2, -1)
            update_gate, reset_gate = (self.sigmoid(gate) for gate in (update_gate, reset_gate))
            candidate_hidden = self.tanh(self.hidden_transform(torch.cat([x[:, i, :], reset_gate * h], dim=-1)))
            h = (1-update_gate) * h + update_gate * candidate_hidden
            y_list.append(h)
        return torch.stack(y_list, dim=1), h
```

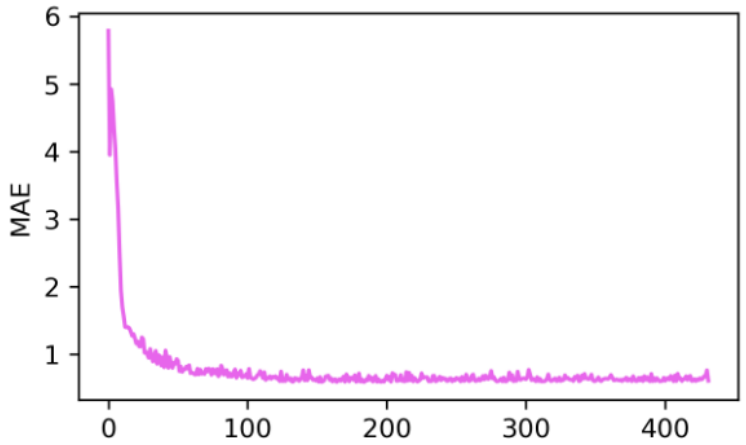
门控循环单元– 训练结果可视化（使用自己实现的GRU）



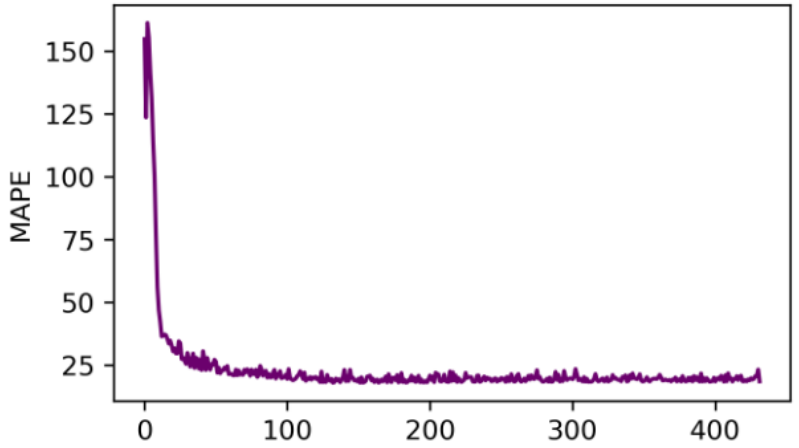
Loss变化



RMSE变化



MAE变化



MAPE变化

门控循环单元 – torch.nn.GRU

输入

Inputs: input, h_0

- **input** of shape (*seq_len, batch, input_size*): tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` for details.
- **h_0** of shape (*num_layers*num_directions, batch, hidden_size*): tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided. If the RNN is bidirectional, num_directions should be 2, else it should be 1.

输出

Outputs: output, h_n

- **output** of shape (*seq_len, batch, num_directions*hidden_size*): tensor containing the output features h_t from the last layer of the GRU, for each *t*. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence. For the unpacked case, the directions can be separated using `output.view(seq_len, batch, num_directions, hidden_size)`, with forward and backward being direction *0* and *1* respectively.
  Similarly, the directions can be separated in the packed case.
- **h_n** of shape (*num_layers*num_directions, batch, hidden_size*): tensor containing the hidden state for *t = seq_len*
  Like *output*, the layers can be separated using `h_n.view(num_layers, num_directions, batch, hidden_size)`.

门控循环单元 – torch.nn.GRU

参数

Parameters

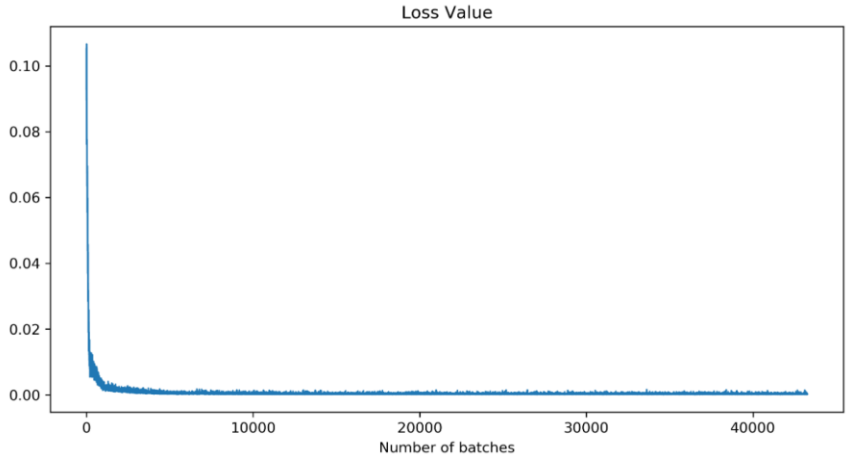
- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two GRUs together to form a *stacked GRU*, with the second GRU taking in outputs of the first GRU and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (batch, seq, feature). Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each GRU layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional GRU. Default: `False`

例子

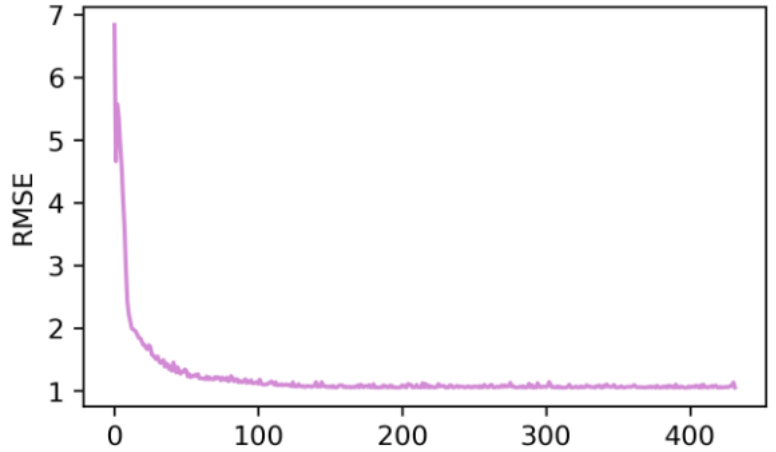
Examples:

```
>>> rnn = nn.GRU(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> output, hn = rnn(input, h0)
```

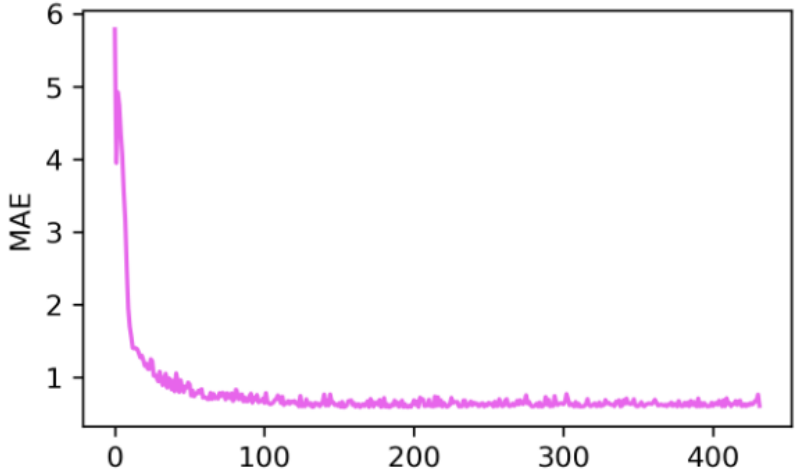
门控循环单元– 训练结果可视化（使用Pytorch自带GRU）



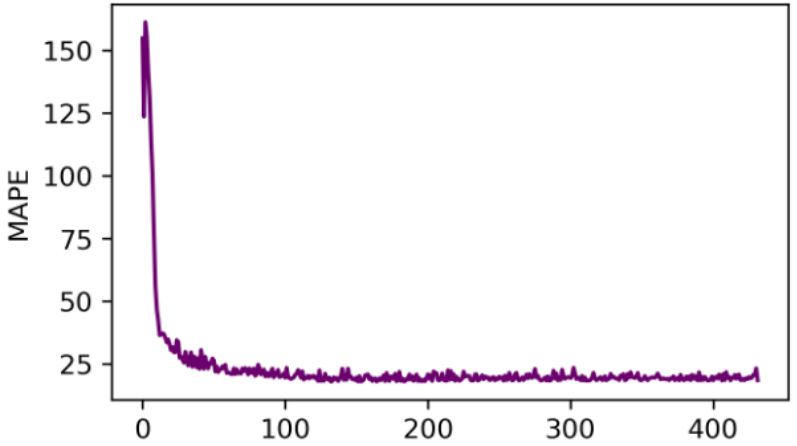
Loss变化



RMSE变化



MAE变化

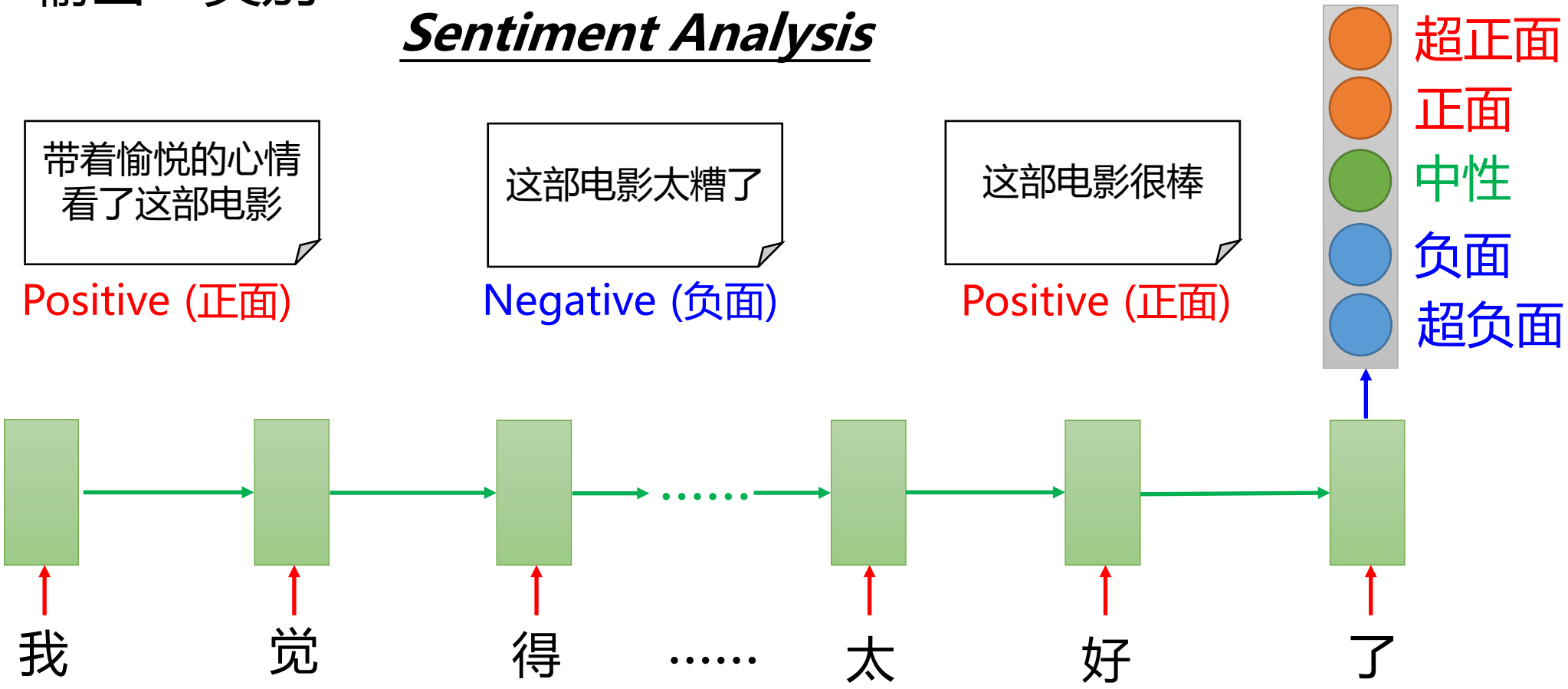


MAPE变化

RNN的一些应用-序列到类别

来源：李宏毅《1天搞懂深度学习》

- 输入：序列
- 输出：类别

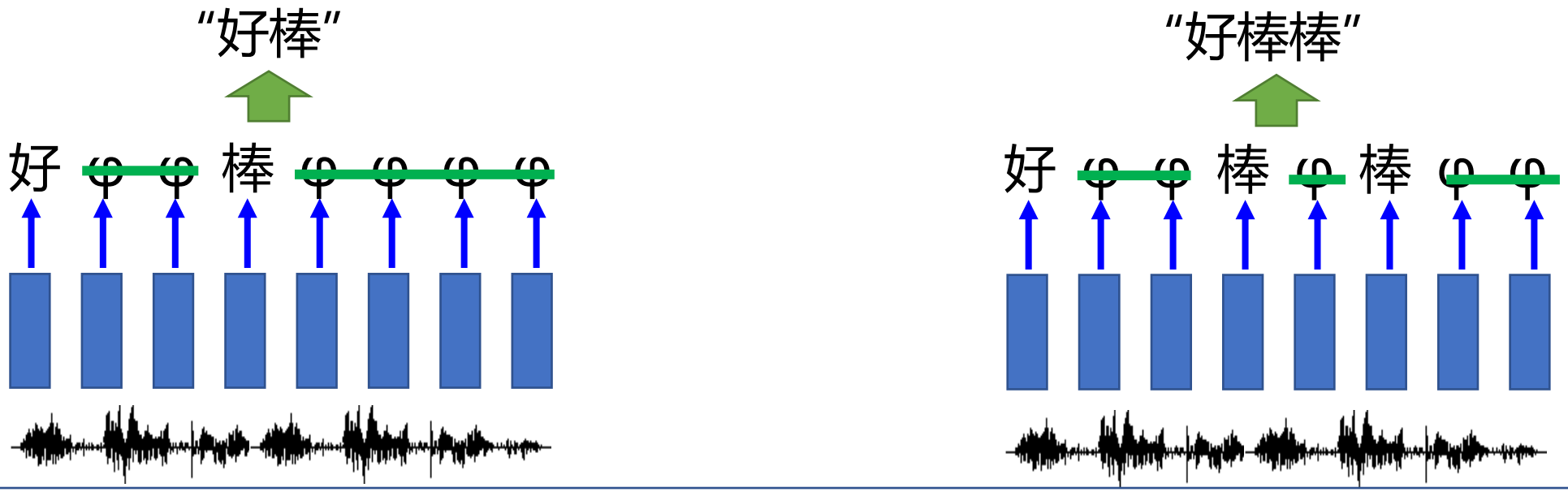


RNN的一些应用-序列到序列

来源：李宏毅《1天搞懂深度学习》

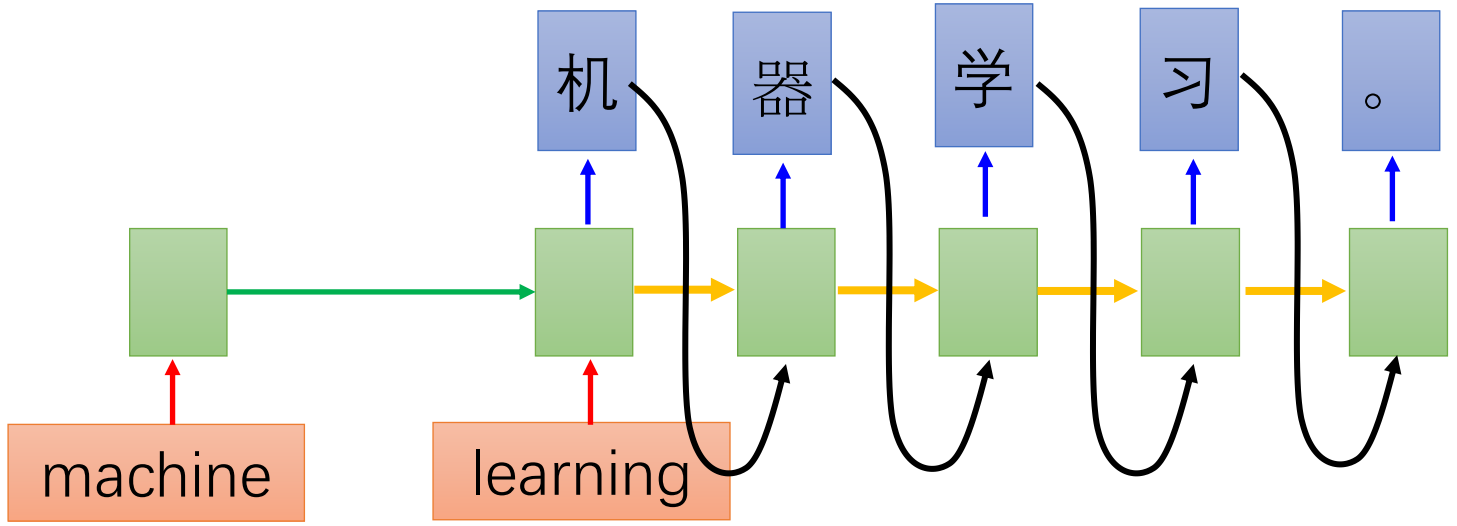
语音识别

同步的序列到序列

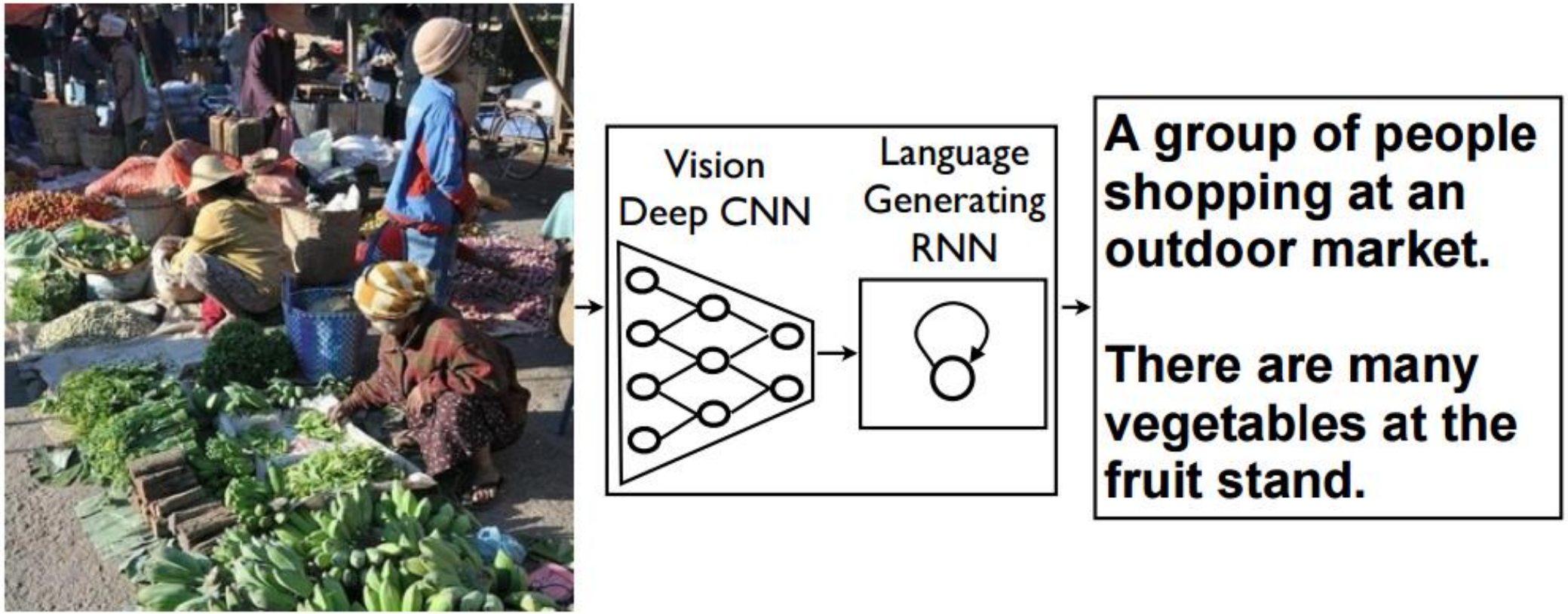


机器翻译

异步的序列到序列

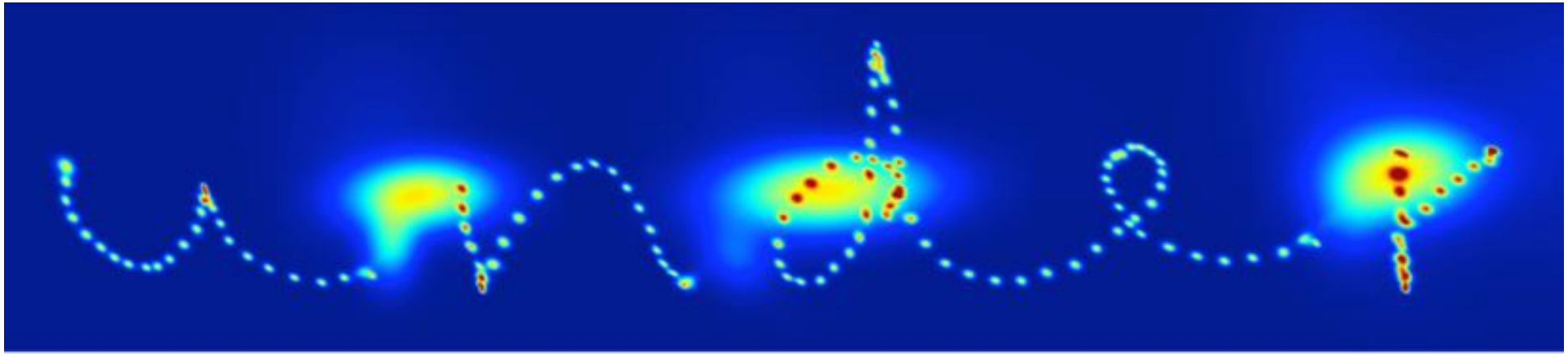


RNN的一些应用 – 看图说话



RNN的一些应用 – 写字、作诗

写字



作诗

白鹭窥鱼立, Egrets stood, peeping fishes. 青山照水开. Water was still, reflecting mountains. 夜来风不动, The wind went down by nightfall, 明月见楼台. as the moon came up by the tower.	满怀风月一枝春, Budding branches are full of romance. 未见梅花亦可人. Plum blossoms are invisible but adorable. 不为东风无此客, With the east wind comes Spring. 世间何处是前身. Where on earth do I come from?

实验要求

数据集介绍1

- 高速公路车流量数据
- PeMS是美国加利福尼亚州高速公路的实时车流量数据。
- 数据由铺设在道路上的检测线圈采集。
- 本实验中包含04和07两个地区的数据，分别储存在PEMS04.npz和PEMS07.npz两个文件中。
- 原始数据使用numpy二进制文件存储，可以使用numpy.load函数读取。
- 数据中的三个特征维度：车流量、拥挤程度和车速

列名	含义
userId	用户ID，不同用户的唯一标识符
venueId	地点ID，不同地点的唯一标识符
venueCategoryId	地点类别ID
venueCategory	地点类别的名称
latitude	地点的纬度
longitude	地点的经度
timezoneOffset	所在时区相对于格林威治时间的时差(分钟)
utcTimestamp	格林威治标准时间

实验要求

数据集介绍2

- 用户签到数据
- FourSquare是一个地点推荐网站，类似于国内的大众点评。
- 当用户到达某个地点时，可以通过手机App进行“签到”(check-in)。
- 将一个用户所有的签到记录按照时间顺序排序，就能得到此用户的行动轨迹
- 本实验中使用的数据包含纽约和东京两个城市的用户签到数据，分别存储在FS_NYC.csv和FS_TKY.csv两个文件中。

列名	含义
userId	用户ID，不同用户的唯一标识符
venueId	地点ID，不同地点的唯一标识符
venueCategoryId	地点类别ID
venueCategory	地点类别的名称
latitude	地点的纬度
longitude	地点的经度
timezoneOffset	所在时区相对于格林威治时间的时差(分钟)
utcTimestamp	格林威治标准时间

循环神经网络实验

- 手动实现循环神经网络RNN，并在至少一个数据集上进行实验，从训练时间、预测精度、Loss变化等角度分析实验结果（最好使用图表展示）
- 使用torch.nn.rnn实现循环神经网络，并在至少一个数据集上进行实验，从训练时间、预测精度、Loss变化等角度分析实验结果（最好使用图表展示）
- 不同超参数的对比分析（包括hidden_size、batchsize、lr等）选其中至少1-2个进行分析

循环神经网络实验

- 使用PyTorch实现LSTM和GRU并在至少一个数据集进行试验分析 （平台课同学选做，专业课同学必做）

- 设计实验，对比分析LSTM和GRU在相同数据集上的结果。