第6章 循环神经网络

循环神经网络（Recurrent Neural Network，RNN）是一类具有短期记忆能力的神经网络．在循环神经网络中，神经元不但可以接受其他神经元的信息，也可以接受自身的信息，形成具有环路的网络结构．和前馈神经网络相比，循环神经网络更加符合生物神经网络的结构．目前，循环神经网络已经被广泛应用在语音识别、语言模型以及自然语言生成等任务上．

本章内容基于《神经网络与深度学习》第6章：循环神经网络的相关内容进行设计。在阅读本章之前，建议先了解如图6.1所示的关键知识点，以便更好地理解和掌握相应的理论和实践知识。

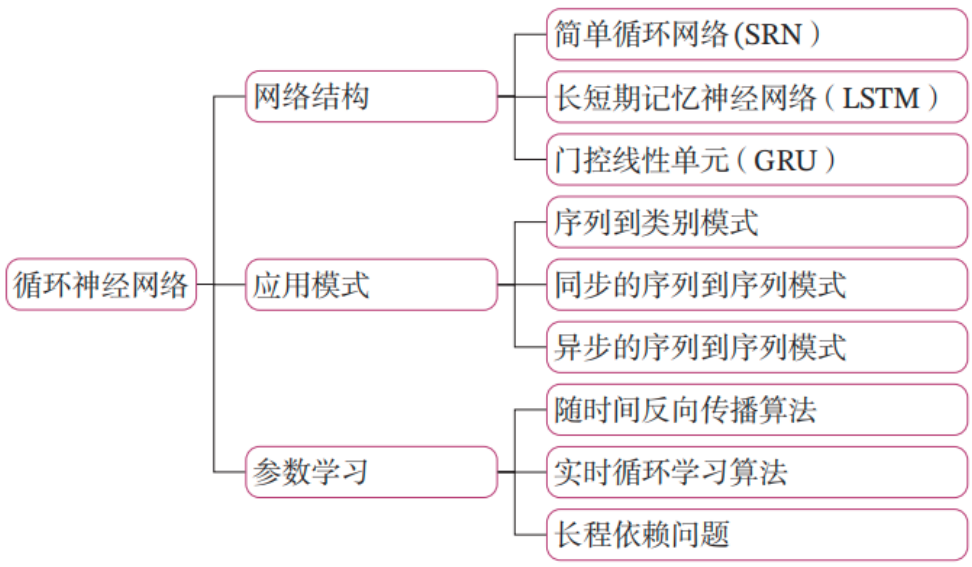


图6.1 《神经网络与深度学习》关键知识点回顾

本章内容主要包含两部分：

- 模型解读：介绍经典循环神经网络原理，为了更好地理解长程依赖问题，我们设计一个简单的数字求和任务来验证简单循环网络的记忆能力。长程依赖问题具体可分为梯度爆炸和梯度消失两种情况。对于梯度爆炸，我们复现简单循环网络的梯度爆炸现象并尝试解决。对于梯度消失，一种有效的方式是改进模型，我们也动手实现一个长短期记忆网络，并观察是否可以缓解长程依赖问题。
- 案例实践：基于双向长短期记忆网络实现文本分类任务．并了解如何进行补齐序列数据，如何将文本数据转为向量表示，如何对补齐位置进行掩蔽等实践知识。

循环神经网络非常擅于处理序列数据，通过使用带自反馈的神经元，能够处理任意长度的序列数据．给定输入序列$[\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2, \dots]$，循环神经网络从左到右扫描该序列，并不断调用一个相同的组合函数$f(\cdot)$来处理时序信息．这个函数也称为循环神经网络单元（RNN Cell）. 在每个时刻t，循环神经网络接受输入信息$\boldsymbol{x}_t \in \mathbb{R}^M$，并与前一时刻的隐状态$\boldsymbol{h}_{t-1} \in \mathbb{R}^D$一起进行计算，输出一个新的当前时刻的隐状态$\boldsymbol{h}_t$.

$$\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t),$$

其中$\boldsymbol{h}_0 = 0$，$f(\cdot)$是一个非线性函数.

循环神经网络的参数可以通过梯度下降法来学习。和前馈神经网络类似，我们可以使用随时间反向传播（BackPropagation Through Time，BPTT）算法高效地手工计算梯度，也可以使用自动微分的方法，通过计算图自动计算梯度。

循环神经网络被认为是图灵完备的，一个完全连接的循环神经网络可以近似解决所有的可计算问题。然而，虽然理论上循环神经网络可以建立长时间间隔的状态之间的依赖关系，但是由于具体的实现方式和参数学习方式会导致梯度爆炸或梯度消失问题，实际上，通常循环神经网络只能学习到短期的依赖关系，很难建模这种长距离的依赖关系，称为长程依赖问题（Long-Term Dependencies Problem）。

6.1 循环神经网络的记忆能力实验

循环神经网络的一种简单实现是简单循环网络（Simple Recurrent Network，SRN）．

令向量$\boldsymbol{x}_t \in \mathbb{R}^M$表示在时刻$t$时网络的输入，$\boldsymbol{h_t} \in \mathbb{R}^D$ 表示隐藏层状态（即隐藏层神经元活性值），则$\boldsymbol{h}_t$不仅和当前时刻的输入$\boldsymbol{x}_t$相关，也和上一个时刻的隐藏层状态$\boldsymbol{h}_{t-1}$相关. 简单循环网络在时刻t的更新公式为

$$\boldsymbol{h}_t = f(\boldsymbol{W}\boldsymbol{x}_t + \boldsymbol{U}\boldsymbol{h}_{t-1} + b),$$

其中$\boldsymbol{h}_t$为隐状态向量，$\boldsymbol{U} \in \mathbb{R}^{D\times D}$为**状态-状态**权重矩阵，$\boldsymbol{W} \in \mathbb{R}^{D\times M}$为**状态-输入**权重矩阵，$\boldsymbol{b} \in \mathbb{R}^D$为偏置向量。

图6.2 展示了一个按时间展开的循环神经网络。

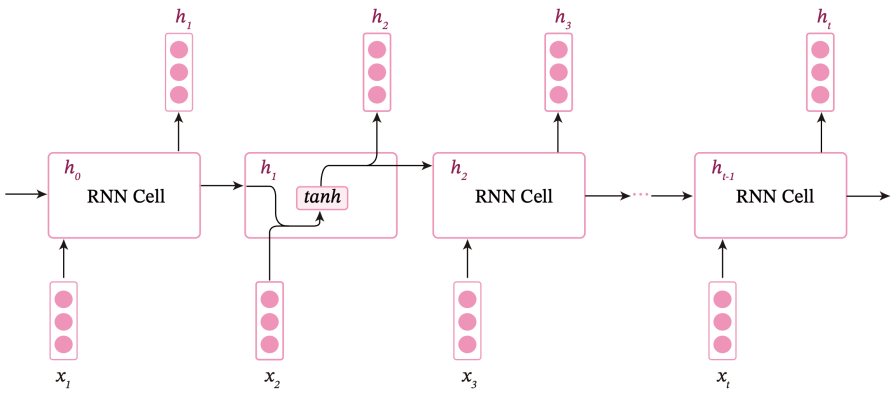


图6.2 循环神经网络结构

简单循环网络在参数学习时存在长程依赖问题，很难建模长时间间隔（Long Range）的状态之间的依赖关系。为了测试简单循环网络的记忆能力，本节构建一个数字求和任务进行实验。

数字求和任务的输入是一串数字，前两个位置的数字为0-9，其余数字随机生成（主要为0），预测目标是输入序列中前两个数字的加和。图6.3展示了长度为10的数字序列.

数字序列										标签
1	2	0	0	0	0	0	0	0	0	3
2	4	0	0	3	0	0	0	0	0	6
3	2	0	0	0	0	1	0	0	0	5
4	1	0	0	0	0	0	0	4	0	5

图6.3 数字求和任务示例

如果序列长度越长，准确率越高，则说明网络的记忆能力越好．因此，我们可以构建不同长度的数据集，通过验证简单循环网络在不同长度的数据集上的表现，从而测试简单循环网络的长程依赖能力.

6.1.1 数据集构建

我们首先构建不同长度的数字预测数据集DigitSum.

6.1.1.1 数据集的构建函数

由于在本任务中，输入序列的前两位数字为 0 − 9，其组合数是固定的，所以可以穷举所有的前两位数字组合，并在后面默认用0填充到固定长度.但考虑到数据的多样性，这里对生成的数字序列中的零位置进行随机采样，并将其随机替换成0-9的数字以增加样本的数量.

我们可以通过设置k的数值来指定一条样本随机生成的数字序列数量.当生成某个指定长度的数据集时，会同时生成训练集、验证集和测试集。当k=3时，生成训练集。当k=1时，生成验证集和测试集. 代码实现如下：

In [1]:
```
import random
import numpy as np

# 固定随机种子
random.seed(0)
np.random.seed(0)

def generate_data(length, k, save_path):
    if length < 3:
        raise ValueError("The length of data should be greater than 2.")
    if k == 0:
        raise ValueError("k should be greater than 0.")
    # 生成100条长度为length的数字序列，除前两个字符外，序列其余数字暂用0填充
    base_examples = []
    for n1 in range(0, 10):
        for n2 in range(0, 10):
            seq = [n1, n2] + [0] * (length - 2)
            label = n1 + n2
            base_examples.append((seq, label))

    examples = []
    # 数据增强：对base_examples中的每条数据，默认生成k条数据，放入examples
    for base_example in base_examples:
        for _ in range(k):
            # 随机生成替换的元素位置和元素
            idx = np.random.randint(2, length)
            val = np.random.randint(0, 10)
            # 对序列中的对应零元素进行替换
            seq = base_example[0].copy()
            label = base_example[1]
            seq[idx] = val
            examples.append((seq, label))

    # 保存增强后的数据
    with open(save_path, "w", encoding="utf-8") as f:
```

```
        for example in examples:
            # 将数据转为字符串类型，方便保存
            seq = [str(e) for e in example[0]]
            label = str(example[1])
            line = " ".join(seq) + "\t" + label + "\n"
            f.write(line)

    print(f"generate data to: {save_path}.")

# 定义生成的数字序列长度
lengths = [5, 10, 15, 20, 25, 30, 35]
for length in lengths:
    # 生成长度为length的训练数据
    save_path = f"./datasets/{length}/train.txt"
    k = 3
    generate_data(length, k, save_path)
    # 生成长度为length的验证数据
    save_path = f"./datasets/{length}/dev.txt"
    k = 1
    generate_data(length, k, save_path)
    # 生成长度为length的测试数据
    save_path = f"./datasets/{length}/test.txt"
    k = 1
    generate_data(length, k, save_path)
```

```
generate data to: ./datasets/5/train.txt.
generate data to: ./datasets/5/dev.txt.
generate data to: ./datasets/5/test.txt.
generate data to: ./datasets/10/train.txt.
generate data to: ./datasets/10/dev.txt.
generate data to: ./datasets/10/test.txt.
generate data to: ./datasets/15/train.txt.
generate data to: ./datasets/15/dev.txt.
generate data to: ./datasets/15/test.txt.
generate data to: ./datasets/20/train.txt.
generate data to: ./datasets/20/dev.txt.
generate data to: ./datasets/20/test.txt.
generate data to: ./datasets/25/train.txt.
generate data to: ./datasets/25/dev.txt.
generate data to: ./datasets/25/test.txt.
generate data to: ./datasets/30/train.txt.
generate data to: ./datasets/30/dev.txt.
generate data to: ./datasets/30/test.txt.
generate data to: ./datasets/35/train.txt.
generate data to: ./datasets/35/dev.txt.
generate data to: ./datasets/35/test.txt.
```

6.1.1.2 加载数据并进行数据划分

为方便使用，本实验提前生成了长度分别为5、10、 15、20、25、30和35的7份数据，存放于“./datasets”目录下，读者可以直接加载使用。代码实现如下：

In [2]:

```
import os
# 加载数据
def load_data(data_path):
    # 加载训练集
    train_examples = []
    train_path = os.path.join(data_path, "train.txt")
    with open(train_path, "r", encoding="utf-8") as f:
        for line in f.readlines():
            # 解析一行数据，将其处理为数字序列seq和标签label
            items = line.strip().split("\t")
            seq = [int(i) for i in items[0].split(" ")]
            label = int(items[1])
            train_examples.append((seq, label))

    # 加载验证集
    dev_examples = []
    dev_path = os.path.join(data_path, "dev.txt")
    with open(dev_path, "r", encoding="utf-8") as f:
        for line in f.readlines():
            # 解析一行数据，将其处理为数字序列seq和标签label
            items = line.strip().split("\t")
            seq = [int(i) for i in items[0].split(" ")]
            label = int(items[1])
            dev_examples.append((seq, label))

    # 加载测试集
    test_examples = []
    test_path = os.path.join(data_path, "test.txt")
    with open(test_path, "r", encoding="utf-8") as f:
        for line in f.readlines():
            # 解析一行数据，将其处理为数字序列seq和标签label
            items = line.strip().split("\t")
            seq = [int(i) for i in items[0].split(" ")]
            label = int(items[1])
            test_examples.append((seq, label))

    return train_examples, dev_examples, test_examples

# 设定加载的数据集的长度
length = 5
```

```
# 该长度的数据集的存放目录
data_path = f"./datasets/{length}"
# 加载该数据集
train_examples, dev_examples, test_examples = load_data(data_path)
print("dev example:", dev_examples[:2])
print("训练集数量：", len(train_examples))
print("验证集数量：", len(dev_examples))
print("测试集数量：", len(test_examples))
```

```
dev example: [([0, 0, 6, 0, 0], 0), ([0, 1, 0, 0, 8], 1)]
训练集数量： 300
验证集数量： 100
测试集数量： 100
```

6.1.1.3 构造Dataset类

为了方便使用梯度下降法进行优化，我们构造了DigitSum数据集的Dataset类，函数 `__getitem__` 负责根据索引读取数据，并将数据转换为张量。代码实现如下：

In [3]:
```
from paddle.io import Dataset

class DigitSumDataset(Dataset):
    def __init__(self, data):
        self.data = data

    def __getitem__(self, idx):
        example = self.data[idx]
        seq = paddle.to_tensor(example[0], dtype="int64")
        label = paddle.to_tensor(example[1], dtype="int64")
        return seq, label

    def __len__(self):
        return len(self.data)
```

6.1.2 模型构建

使用SRN模型进行数字加和任务的模型结构为如图6.4所示.

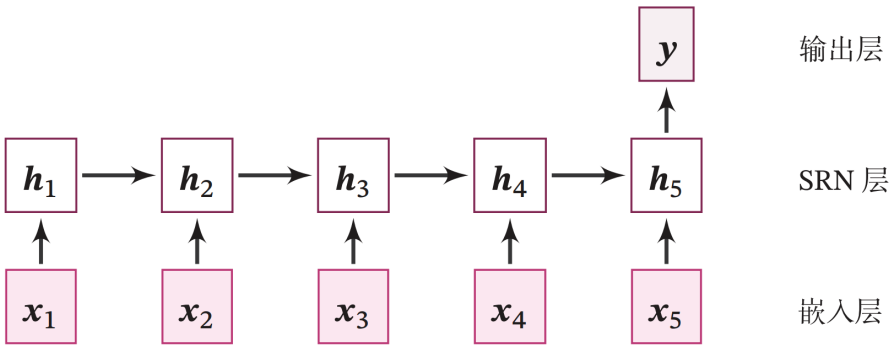


图6.4 基于SRN模型的数字预测

整个模型由以下几个部分组成：

（1） 嵌入层：将输入的数字序列进行向量化，即将每个数字映射为向量；

（2） SRN 层：接收向量序列，更新循环单元，将最后时刻的隐状态作为整个序列的表示；

（3） 输出层：一个线性层，输出分类的结果.

6.1.2.1 嵌入层

本任务输入的样本是数字序列，为了更好地表示数字，需要将数字映射为一个嵌入（Embedding）向量。嵌入向量中的每个维度均能用来刻画该数字本身的某种特性。由于向量能够表达该数字更多的信息，利用向量进行数字求和任务，可以使得模型具有更强的拟合能力。

首先，我们构建一个嵌入矩阵（Embedding Matrix）$\boldsymbol{E} \in \mathbb{R}^{10\times M}$，其中第$i$行对应数字$i$的嵌入向量，每个嵌入向量的维度是$M$。如图6.5所示。给定一个组数字序列$\boldsymbol{S} \in \mathbb{R}^{B\times L}$，其中$B$为批大小，$L$为序列长度，可以通过查表将其映射为嵌入表示$\boldsymbol{X} \in \mathbb{R}^{B\times L\times M}$。

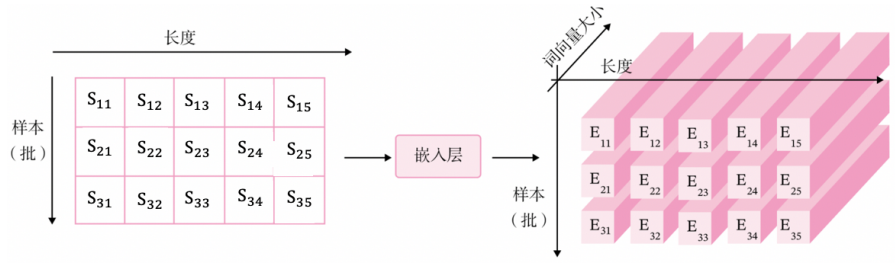


图6.5 嵌入矩阵

> **提醒**：为了和代码的实现保持一致性，这里使用形状为(样本数量 × 序列长度 × 特征维度)的张量来表示一组样本。

或者也可以将每个数字表示为10维的one-hot向量，使用矩阵运算得到嵌入表示：

$$\boldsymbol{X} = \boldsymbol{S}^{'}\boldsymbol{E},$$

其中$\boldsymbol{S}' \in \mathbb{R}^{B\times L\times 10}$是序列$\boldsymbol{S}$对应的one-hot表示。

> **思考**：如果不使用嵌入层，直接将数字作为SRN层输入有什么问题？

基于索引方式的嵌入层的实现如下：

In [4]:
```
import paddle
import paddle.nn as nn

class Embedding(nn.Layer):
    def __init__(self, num_embeddings, embedding_dim, para_attr=paddle.ParamAttr(initializer=nn.initializer.XavierUniform())):
        super(Embedding, self).__init__()
        # 定义嵌入矩阵
        self.W = paddle.create_parameter(shape=[num_embeddings, embedding_dim], dtype="float32", attr=para_attr)

    def forward(self, inputs):
        # 根据索引获取对应词向量
        embs = self.W[inputs]
        return embs

emb_layer = Embedding(10, 5)
inputs = paddle.to_tensor([0,1,2,3])
emb_layer(inputs)
```

```
W0516 19:19:23.716145   101 device_context.cc:447] Please NOTE: device: 0, GPU Compute Capability: 7.0, Driver API Version: 11.2, Runtime API Version: 10.1
W0516 19:19:23.720912   101 device_context.cc:465] device: 0, cuDNN Version: 7.6.
/opt/conda/envs/python35-paddle120-env/lib/python3.7/site-packages/paddle/tensor/creation.py:130: DeprecationWarning: `np.object` is a deprecated alias for the builtin `object`. To silence this warning, use `object` by itself. Doing this will not modify any behavior and is safe.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  if data.dtype == np.object:
```

Out[4]:
```
Tensor(shape=[4, 5], dtype=float32, place=CUDAPlace(0), stop_gradient=False,
       [[-0.38702673,  0.56934923,  0.33333215,  0.60759938, -0.06306949],
        [ 0.21330097, -0.12489522, -0.25023243, -0.33473995, -0.25782728],
        [-0.12060214, -0.46596608, -0.00060794, -0.25233585,  0.59036279],
        [ 0.22183451, -0.56272739,  0.55152571,  0.11593318,  0.24588455]])
```

> **思考**：请同学们思考基于one-hot编码的嵌入层应该如何实现.

6.1.2.2 SRN层

数字序列$\boldsymbol{S} \in \mathbb{R}^{B\times L}$经过嵌入层映射后，转换为$\boldsymbol{X} \in \mathbb{R}^{B\times L\times M}$，其中$B$为批大小，$L$为序列长度，$M$为嵌入维度。

在时刻t，SRN将当前的输入$\boldsymbol{X}_t \in \mathbb{R}^{B\times M}$与隐状态$\boldsymbol{H}_{t-1} \in \mathbb{R}^{B\times D}$进行线性变换和组合，并通过一个非线性激活函数$f(\cdot)$得到新的隐状态，SRN的状态更新函数为:

$$\boldsymbol{H}_t = \text{Tanh}(\boldsymbol{X}_t\boldsymbol{W} + \boldsymbol{H}_{t-1}\boldsymbol{U} + \boldsymbol{b}),$$

其中$\boldsymbol{W} \in \mathbb{R}^{M\times D}, \boldsymbol{U} \in \mathbb{R}^{D\times D}, \boldsymbol{b} \in \mathbb{R}^{1\times D}$是可学习参数，$D$表示隐状态向量的维度。

简单循环网络的代码实现如下:

In [5]:
```
import paddle
import paddle.nn as nn
import paddle.nn.functional as F
paddle.seed(0)

# SRN模型
class SRN(nn.Layer):
    def __init__(self, input_size,  hidden_size, W_attr=None, U_attr=None, b_attr=None):
        super(SRN, self).__init__()
        # 嵌入向量的维度
        self.input_size = input_size
        # 隐状态的维度
        self.hidden_size = hidden_size
        # 定义模型参数W，其shape为 input_size x hidden_size
        self.W = paddle.create_parameter(shape=[input_size, hidden_size], dtype="float32", attr=W_attr)
        # 定义模型参数U，其shape为hidden_size x hidden_size
        self.U = paddle.create_parameter(shape=[hidden_size, hidden_size], dtype="float32",attr=U_attr)
        # 定义模型参数b，其shape为 1 x hidden_size
        self.b = paddle.create_parameter(shape=[1, hidden_size], dtype="float32", attr=b_attr)

    # 初始化向量
    def init_state(self, batch_size):
        hidden_state = paddle.zeros(shape=[batch_size, self.hidden_size], dtype="float32")
        return hidden_state

    # 定义前向计算
    def forward(self, inputs, hidden_state=None):
        # inputs: 输入数据，其shape为batch_size x seq_len x input_size
        batch_size, seq_len, input_size = inputs.shape

        # 初始化起始状态的隐向量，其shape为 batch_size x hidden_size
        if hidden_state is None:
            hidden_state = self.init_state(batch_size)

        # 循环执行RNN计算
        for step in range(seq_len):
```

```
            # 获取当前时刻的输入数据step_input, 其shape为 batch_size x input_size
            step_input = inputs[:, step, :]
            # 获取当前时刻的隐状态向量hidden_state, 其shape为 batch_size x hidden_size
            hidden_state = F.tanh(paddle.matmul(step_input, self.W) + paddle.matmul(hidden_state, self.U) + self.b)
        return hidden_state
```

> 提醒：这里只保留了简单循环网络的最后一个时刻的输出向量。

In [6]:
```
## 初始化参数并运行

W_attr = paddle.ParamAttr(initializer=nn.initializer.Assign([[0.1, 0.2], [0.1,0.2]]))
U_attr = paddle.ParamAttr(initializer=nn.initializer.Assign([[0.0, 0.1], [0.1,0.0]]))
b_attr = paddle.ParamAttr(initializer=nn.initializer.Assign([[0.1, 0.1]]))

srn = SRN(2, 2, W_attr=W_attr, U_attr=U_attr, b_attr=b_attr)

inputs = paddle.to_tensor([[[1, 0],[0, 2]]], dtype="float32")
hidden_state = srn(inputs)
print("hidden_state", hidden_state)
```

```
hidden_state Tensor(shape=[1, 2], dtype=float32, place=CUDAPlace(0), stop_gradient=False,
       [[0.31773996, 0.47749740]])
```

飞桨框架已经内置了SRN的API `paddle.nn.SimpleRNN`，其与自己实现的SRN不同点在于其实现时采用了两个偏置，同时矩阵相乘时参数在输入数据前面，如下公式所示：

$$\boldsymbol{H}_t = \text{Tanh}(\boldsymbol{W}\boldsymbol{X}_t + \boldsymbol{b}_x + \boldsymbol{U}\boldsymbol{H}_{t-1} + \boldsymbol{b}_h),$$

其中$\boldsymbol{W} \in \mathbb{R}^{M\times D}, \boldsymbol{U} \in \mathbb{R}^{D\times D}, \boldsymbol{b}_x \in \mathbb{R}^{1\times D}, \boldsymbol{b}_h \in \mathbb{R}^{1\times D}$是可学习参数，$M$表示嵌入向量的维度，$D$表示隐状态向量的维度。

另外，内置SRN API在执行完前向计算后，会返回两个参数：序列向量和最后时刻的隐状态向量。在飞桨实现时，考虑到了双向和多层SRN的因素，返回的向量附带了这些信息。

其中序列向量outputs是指最后一层SRN的输出向量，其shape为[batch_size, seq_len, num_directions *hidden_size]；最后时刻的隐状态向量shape为[num_layers* num_directions, batch_size, hidden_size]。

这里我们可以将自己实现的SRN和Paddle框架内置的SRN返回的结果进行打印展示，实现代码如下。

In [7]:
```
# 这里创建一个随机数组作为测试数据，数据shape为batch_size x seq_len x input_size
batch_size, seq_len, input_size = 8, 20, 32
inputs = paddle.randn(shape=[batch_size, seq_len, input_size])

# 设置模型的hidden_size
hidden_size = 32
paddle_srn = nn.SimpleRNN(input_size, hidden_size)
self_srn = SRN(input_size, hidden_size)

self_hidden_state = self_srn(inputs)
paddle_outputs, paddle_hidden_state = paddle_srn(inputs)

print("self_srn hidden_state: ", self_hidden_state.shape)
print("paddle_srn outpus:", paddle_outputs.shape)
print("paddle_srn hidden_state:", paddle_hidden_state.shape)
```

```
self_srn hidden_state:  [8, 32]
paddle_srn outpus: [8, 20, 32]
paddle_srn hidden_state: [1, 8, 32]
```

可以看到，自己实现的SRN由于没有考虑多层因素，因此没有层次这个维度，因此其输出shape为[8, 32]。同时由于在以上代码使用Paddle内置API实例化SRN时，默认定义的是1层的单向SRN，因此其shape为[1, 8, 32]，同时隐状态向量为[8,20, 32].

接下来，我们可以将自己实现的SRN与Paddle内置的SRN在输出值的精度上进行对比，这里首先根据Paddle内置的SRN实例化模型（为了进行对比，在实例化时只保留一个偏置，将偏置b_x设置为0），然后提取该模型对应的参数，使用该参数去初始化自己实现的SRN，从而保证两者在参数初始化时是一致的。

在进行实验时，首先定义输入数据 `inputs`，然后将该数据分别传入Paddle内置的SRN与自己实现的SRN模型中，最后通过对比两者的隐状态输出向量。代码实现如下：

In [8]:
```
paddle.seed(0)

# 这里创建一个随机数组作为测试数据，数据shape为batch_size x seq_len x input_size
batch_size, seq_len, input_size, hidden_size = 2, 5, 10, 10
inputs = paddle.randn(shape=[batch_size, seq_len, input_size])

# 设置模型的hidden_size
bx_attr = paddle.ParamAttr(initializer=nn.initializer.Assign(paddle.zeros([hidden_size, ])))
paddle_srn = nn.SimpleRNN(input_size, hidden_size, bias_ih_attr=bx_attr)

# 获取paddle_srn中的参数，并设置相应的paramAttr,用于初始化SRN
W_attr = paddle.ParamAttr(initializer=nn.initializer.Assign(paddle_srn.weight_ih_l0.T))
U_attr = paddle.ParamAttr(initializer=nn.initializer.Assign(paddle_srn.weight_hh_l0.T))
b_attr = paddle.ParamAttr(initializer=nn.initializer.Assign(paddle_srn.bias_hh_l0))
self_srn = SRN(input_size, hidden_size, W_attr=W_attr, U_attr=U_attr, b_attr=b_attr)

# 进行前向计算，获取隐状态向量，并打印展示
self_hidden_state = self_srn(inputs)
```

```
paddle_outputs, paddle_hidden_state = paddle_srn(inputs)
print("paddle SRN:\n", paddle_hidden_state.numpy().squeeze(0))
print("self SRN:\n", self_hidden_state.numpy())
```

```
paddle SRN:
 [[ 0.3246606  -0.05465741 -0.3090897  -0.51604617 -0.11149617  0.4267313
   0.47200006 -0.06585315  0.85319966  0.18898569]
 [-0.4299355  -0.6067489  -0.59150505  0.30245274 -0.03939498  0.61462754
   0.4030218   0.49883503  0.02484456 -0.38516262]]
self SRN:
 [[ 0.32466057 -0.05465744 -0.3090897  -0.51604617 -0.11149605  0.4267313
   0.47200006 -0.06585318  0.85319966  0.18898569]
 [-0.42993543 -0.6067488  -0.59150493  0.3024528  -0.03939501  0.61462754
   0.40302184  0.49883503  0.02484456 -0.38516262]]
```

可以看到，两者的输出基本是一致的。另外，还可以进行对比两者在运算速度方面的差异。代码实现如下：

In [9]:
```
import time

# 这里创建一个随机数组作为测试数据，数据shape为batch_size x seq_len x input_size
batch_size, seq_len, input_size, hidden_size = 2, 5, 10, 10
inputs = paddle.randn(shape=[batch_size, seq_len, input_size])

# 实例化模型
self_srn = SRN(input_size, hidden_size)
paddle_srn = nn.SimpleRNN(input_size, hidden_size)

# 计算自己实现的SRN运算速度
model_time = 0
for i in range(100):
    strat_time = time.time()
    out = self_srn(inputs)
    # 预热10次运算，不计入最终速度统计
    if i < 10:
        continue
    end_time = time.time()
    model_time += (end_time - strat_time)
avg_model_time = model_time / 90
print('self_srn speed:', avg_model_time, 's')

# 计算Paddle内置的SRN运算速度
model_time = 0
for i in range(100):
    strat_time = time.time()
    out = paddle_srn(inputs)
    # 预热10次运算，不计入最终速度统计
    if i < 10:
        continue
    end_time = time.time()
    model_time += (end_time - strat_time)
avg_model_time = model_time / 90
print('paddle_srn speed:', avg_model_time, 's')
```

```
self_srn speed: 0.0010721974902682834 s
paddle_srn speed: 0.0004733721415201823 s
```

可以看到，由于Paddle内部相关算子由C++实现，Paddle框架实现的SRN的运行效率显著高于自己实现的SRN效率。

6.1.2.3 线性层

线性层会将最后一个时刻的隐状态向量$\boldsymbol{H}_L \in \mathbb{R}^{B\times D}$进行线性变换，输出分类的对数几率（Logits）为：

$$\boldsymbol{Y} = \boldsymbol{H}_L\boldsymbol{W}_o + \boldsymbol{b}_o,$$

其中$\boldsymbol{W}_o \in \mathbb{R}^{D\times 19}$，$\boldsymbol{b}_o \in \mathbb{R}^{19}$为可学习的权重矩阵和偏置。

> 提醒：在分类问题的实践中，我们通常只需要模型输出分类的对数几率（Logits），而不用输出每个类的概率。这需要损失函数可以直接接收对数几率来损失计算。

线性层直接使用`paddle.nn.Linear`算子。

6.1.2.4 模型汇总

在定义了每一层的算子之后，我们定义一个数字求和模型Model_RNN4SeqClass，该模型会将嵌入层、SRN层和线性层进行组合，以实现数字求和的功能.

具体来讲，Model_RNN4SeqClass会接收一个SRN层实例，用于处理数字序列数据，同时在`__init__`函数中定义一个`Embedding`嵌入层，其会将输入的数字作为索引，输出对应的向量，最后会使用`paddle.nn.Linear`定义一个线性层。

> 提醒：为了方便进行对比实验，我们将SRN层的实例化放在\code{Model_RNN4SeqClass}类外面。通常情况下，模型内部算子的实例化是放在模型里面。

在`forward`函数中，调用上文实现的嵌入层、SRN层和线性层处理数字序列，同时返回最后一个位置的隐状态向量。代码实现如下：

In [10]:
```
# 基于RNN实现数字预测的模型
class Model_RNN4SeqClass(nn.Layer):
    def __init__(self, model, num_digits, input_size, hidden_size, num_classes):
```

```
        super(Model_RNN4SeqClass, self).__init__()
        # 传入实例化的RNN层，例如SRN
        self.rnn_model = model
        # 词典大小
        self.num_digits = num_digits
        # 嵌入向量的维度
        self.input_size = input_size
        # 定义Embedding层
        self.embedding = Embedding(num_digits, input_size)
        # 定义线性层
        self.linear = nn.Linear(hidden_size, num_classes)

    def forward(self, inputs):
        # 将数字序列映射为相应向量
        inputs_emb = self.embedding(inputs)
        # 调用RNN模型
        hidden_state = self.rnn_model(inputs_emb)
        # 使用最后一个时刻的状态进行数字预测
        logits = self.linear(hidden_state)
        return logits

# 实例化一个input_size为4， hidden_size为5的SRN
srn = SRN(4, 5)
# 基于srn实例化一个数字预测模型实例
model = Model_RNN4SeqClass(srn, 10, 4, 5, 19)
# 生成一个shape为 2 x 3 的批次数据
inputs = paddle.to_tensor([[1, 2, 3], [2, 3, 4]])
# 进行模型前向预测
logits = model(inputs)
print(logits)
```

```
Tensor(shape=[2, 19], dtype=float32, place=CUDAPlace(0), stop_gradient=False,
       [[ 0.36087763, -0.03377634, -0.04800312,  0.49252868, -0.45962709,
         -0.44703209,  0.64295375,  0.53624588, -0.19376591,  0.11085325,
         -0.31243768,  0.29747075, -0.31725749, -0.41438878,  0.00990404,
          0.45916951, -0.31540897,  0.57389849, -0.03416194],
        [ 0.01424524, -0.12685573, -0.07969519,  0.56528699, -0.65557188,
         -0.57581109,  0.84303617,  0.07659776,  0.01592400, -0.38144892,
         -0.24371660,  0.38759732, -0.46055052, -0.87889659, -0.16003403,
          0.67612255, -0.36139122,  0.40609291, -0.26660436]])
```

6.1.3 模型训练

6.1.3.1 训练指定长度的数字预测模型

基于RunnerV3类进行训练，只需要指定length便可以加载相应的数据。设置超参数，使用Adam优化器，学习率为 0.001，实例化模型，使用第4.5.4节定义的Accuracy计算准确率。使用Runner进行训练，训练回合数设为500。代码实现如下：

In [11]:
```
import os
import random
import paddle
import numpy as np
from nndl import Accuracy, RunnerV3

# 训练轮次
num_epochs = 500
# 学习率
lr = 0.001
# 输入数字的类别数
num_digits = 10
# 将数字映射为向量的维度
input_size = 32
# 隐状态向量的维度
hidden_size = 32
# 预测数字的类别数
num_classes = 19
# 批大小
batch_size = 8
# 模型保存目录
save_dir = "./checkpoints"

# 通过指定length进行不同长度数据的实验
def train(length):
    print(f"\n====> Training SRN with data of length {length}.")
    # 固定随机种子
    np.random.seed(0)
    random.seed(0)
    paddle.seed(0)

    # 加载长度为length的数据
    data_path = f"./datasets/{length}"
    train_examples, dev_examples, test_examples = load_data(data_path)
    train_set, dev_set, test_set = DigitSumDataset(train_examples), DigitSumDataset(dev_examples), DigitSumDataset(test_examples)
    train_loader = paddle.io.DataLoader(train_set, batch_size=batch_size)
    dev_loader = paddle.io.DataLoader(dev_set, batch_size=batch_size)
    test_loader = paddle.io.DataLoader(test_set, batch_size=batch_size)
    # 实例化模型
    base_model = SRN(input_size, hidden_size)
    model = Model_RNN4SeqClass(base_model, num_digits, input_size, hidden_size, num_classes)
    # 指定优化器
    optimizer = paddle.optimizer.Adam(learning_rate=lr, parameters=model.parameters())
```

```
    # 定义评价指标
    metric = Accuracy()
    # 定义损失函数
    loss_fn = nn.CrossEntropyLoss()

    # 基于以上组件，实例化Runner
    runner = RunnerV3(model, optimizer, loss_fn, metric)

    # 进行模型训练
    model_save_path = os.path.join(save_dir, f"best_srn_model_{length}.pdparams")
    runner.train(train_loader, dev_loader, num_epochs=num_epochs, eval_steps=100, log_steps=100, save_path=model_save_path)

    return runner
```

```
/opt/conda/envs/python35-paddle120-env/lib/python3.7/site-packages/matplotlib/__init__.py:107: DeprecationWarning: Using or importi
ng the ABCs from 'collections' instead of from 'collections.abc' is deprecated, and in 3.8 it will stop working
  from collections import MutableMapping
/opt/conda/envs/python35-paddle120-env/lib/python3.7/site-packages/matplotlib/rcsetup.py:20: DeprecationWarning: Using or importing
the ABCs from 'collections' instead of from 'collections.abc' is deprecated, and in 3.8 it will stop working
  from collections import Iterable, Mapping
/opt/conda/envs/python35-paddle120-env/lib/python3.7/site-packages/matplotlib/colors.py:53: DeprecationWarning: Using or importing
the ABCs from 'collections' instead of from 'collections.abc' is deprecated, and in 3.8 it will stop working
  from collections import Sized
```

6.1.3.2 多组训练

接下来，分别进行数据长度为10, 15, 20, 25, 30, 35的数字预测模型训练实验，训练后的 `runner` 保存至 `runners` 字典中。

In [12]:
```
srn_runners = {}
lengths = [10, 15, 20, 25, 30, 35]
for length in lengths:
    runner = train(length)
    srn_runners[length] = runner
```

```
====> Training SRN with data of length 10.
[Train] epoch: 0/500, step: 0/19000, loss: 3.19570
/opt/conda/envs/python35-paddle120-env/lib/python3.7/site-packages/paddle/tensor/creation.py:130: DeprecationWarning: `np.object` i
s a deprecated alias for the builtin `object`. To silence this warning, use `object` by itself. Doing this will not modify any beha
vior and is safe.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  if data.dtype == np.object:
[Train] epoch: 2/500, step: 100/19000, loss: 2.62474
[Evaluate]  dev score: 0.03000, dev loss: 2.88084
[Evaluate] best accuracy performence has been updated: 0.00000 --> 0.03000
[Train] epoch: 5/500, step: 200/19000, loss: 2.36054
[Evaluate]  dev score: 0.10000, dev loss: 2.84322
[Evaluate] best accuracy performence has been updated: 0.03000 --> 0.10000
[Train] epoch: 7/500, step: 300/19000, loss: 2.52785
[Evaluate]  dev score: 0.06000, dev loss: 2.80278
[Train] epoch: 10/500, step: 400/19000, loss: 2.40403
[Evaluate]  dev score: 0.10000, dev loss: 2.73180
[Train] epoch: 13/500, step: 500/19000, loss: 2.37544
[Evaluate]  dev score: 0.14000, dev loss: 2.60807
[Evaluate] best accuracy performence has been updated: 0.10000 --> 0.14000
[Train] epoch: 15/500, step: 600/19000, loss: 2.64682
[Evaluate]  dev score: 0.16000, dev loss: 2.45143
[Evaluate] best accuracy performence has been updated: 0.14000 --> 0.16000
[Train] epoch: 18/500, step: 700/19000, loss: 2.19377
[Evaluate]  dev score: 0.23000, dev loss: 2.31756
[Evaluate] best accuracy performence has been updated: 0.16000 --> 0.23000
[Train] epoch: 21/500, step: 800/19000, loss: 2.18995
[Evaluate]  dev score: 0.23000, dev loss: 2.21527
[Train] epoch: 23/500, step: 900/19000, loss: 2.43149
[Evaluate]  dev score: 0.28000, dev loss: 2.11419
[Evaluate] best accuracy performence has been updated: 0.23000 --> 0.28000
[Train] epoch: 26/500, step: 1000/19000, loss: 1.88478
[Evaluate]  dev score: 0.31000, dev loss: 2.05610
[Evaluate] best accuracy performence has been updated: 0.28000 --> 0.31000
[Train] epoch: 28/500, step: 1100/19000, loss: 1.62245
[Evaluate]  dev score: 0.34000, dev loss: 2.03233
[Evaluate] best accuracy performence has been updated: 0.31000 --> 0.34000
[Train] epoch: 31/500, step: 1200/19000, loss: 2.15871
[Evaluate]  dev score: 0.39000, dev loss: 1.96400
[Evaluate] best accuracy performence has been updated: 0.34000 --> 0.39000
[Train] epoch: 34/500, step: 1300/19000, loss: 1.49926
[Evaluate]  dev score: 0.37000, dev loss: 1.89185
```

6.1.3.3 损失曲线展示

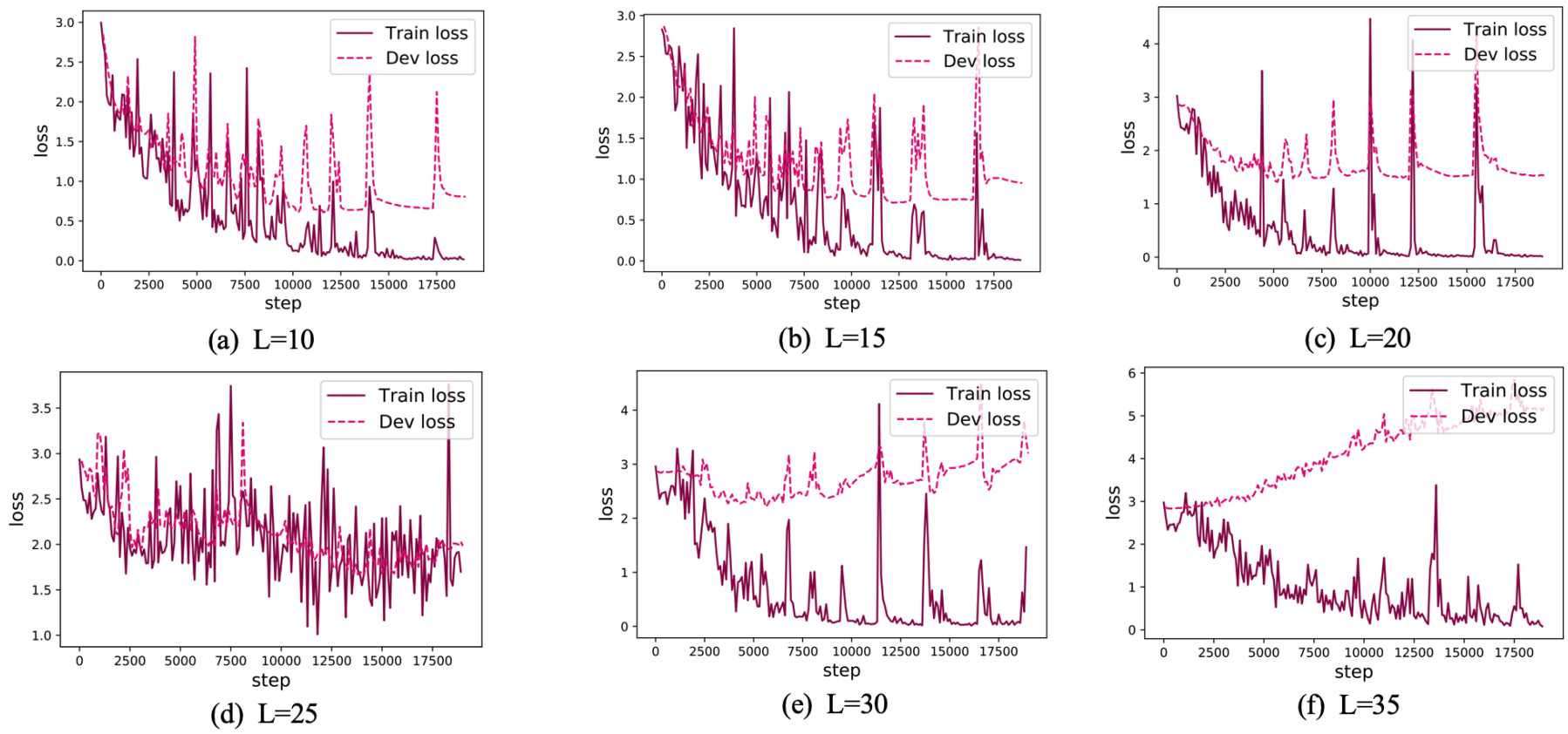
定义 `plot_training_loss` 函数，分别画出各个长度的数字预测模型训练过程中，在训练集和验证集上的损失曲线，实现代码实现如下：

In [24]:
```
import matplotlib.pyplot as plt

def plot_training_loss(runner, fig_name, sample_step):

    plt.figure()
    train_items = runner.train_step_losses[::sample_step]
    train_steps=[x[0] for x in train_items]
    train_losses = [x[1] for x in train_items]
    plt.plot(train_steps, train_losses, color='#8E004D', label="Train loss")
```

```
    dev_steps=[x[0] for x in runner.dev_losses]
    dev_losses = [x[1] for x in runner.dev_losses]
    plt.plot(dev_steps, dev_losses, color='#E20079', linestyle='--', label="Dev loss")

    #绘制坐标轴和图例
    plt.ylabel("loss", fontsize='x-large')
    plt.xlabel("step", fontsize='x-large')
    plt.legend(loc='upper right', fontsize='x-large')

    plt.savefig(fig_name)
    plt.show()
```

In [25]:
```
# 画出训练过程中的损失图
for length in lengths:
    runner = srn_runners[length]
    fig_name = f"./images/6.6_{length}.pdf"
    plot_training_loss(runner, fig_name, sample_step=100)
```

图6.6展示了在6个数据集上的损失变化情况，数据集的长度分别为10、15、20、25、30和35. 从输出结果看，随着数据序列长度的增加，虽然训练集损失逐渐逼近于0，但是验证集损失整体趋向越来越大，这表明当序列变长时，SRN模型保持序列长期依赖能力在逐渐变弱，越来越无法学习到有用的知识.



(a) L=10 (b) L=15 (c) L=20

(d) L=25 (e) L=30 (f) L=35

图6.6 SRN在不同长度数据集训练损失变化图

6.1.4 模型评价

在模型评价时，加载不同长度的效果最好的模型，然后使用测试集对该模型进行评价，观察模型在测试集上预测的准确度. 同时记录一下不同长度模型在训练过程中，在验证集上最好的效果。代码实现如下。

In [17]:
```
srn_dev_scores = []
srn_test_scores = []
for length in lengths:
    print(f"Evaluate SRN with data length {length}.")
    runner = srn_runners[length]
    # 加载训练过程中效果最好的模型
    model_path = os.path.join(save_dir, f"best_srn_model_{length}.pdparams")
    runner.load_model(model_path)

    # 加载长度为length的数据
    data_path = f"./datasets/{length}"
    train_examples, dev_examples, test_examples = load_data(data_path)
    test_set = DigitSumDataset(test_examples)
    test_loader = paddle.io.DataLoader(test_set, batch_size=batch_size)

    # 使用测试集评价模型，获取测试集上的预测准确率
    score, _ = runner.evaluate(test_loader)
    srn_test_scores.append(score)
    srn_dev_scores.append(max(runner.dev_scores))

for length, dev_score, test_score in zip(lengths, srn_dev_scores, srn_test_scores):
    print(f"[SRN] length:{length}, dev_score: {dev_score}, test_score: {test_score: .5f}")
```

接下来，将SRN在不同长度的验证集和测试集数据上的表现，绘制成图片进行观察。

In [ ]:
```
import matplotlib.pyplot as plt

plt.plot(lengths, srn_dev_scores, '-o', color='#8E004D',  label="Dev Accuracy")
plt.plot(lengths, srn_test_scores,'-o', color='#E20079', label="Test Accuracy")

#绘制坐标轴和图例
```

```
plt.ylabel("loss", fontsize='x-large')
plt.xlabel("step", fontsize='x-large')
plt.legend(loc='upper right', fontsize='x-large')

fig_name = "./images/6.7.pdf"
plt.savefig(fig_name)
plt.show()
```

图6.7 展示了SRN模型在不同长度数据训练出来的最好模型在验证集和测试集上的表现。可以看到，随着序列长度的增加，验证集和测试集的准确度整体趋势是降低的，这同样说明SRN模型保持长期依赖的能力在不断降低.

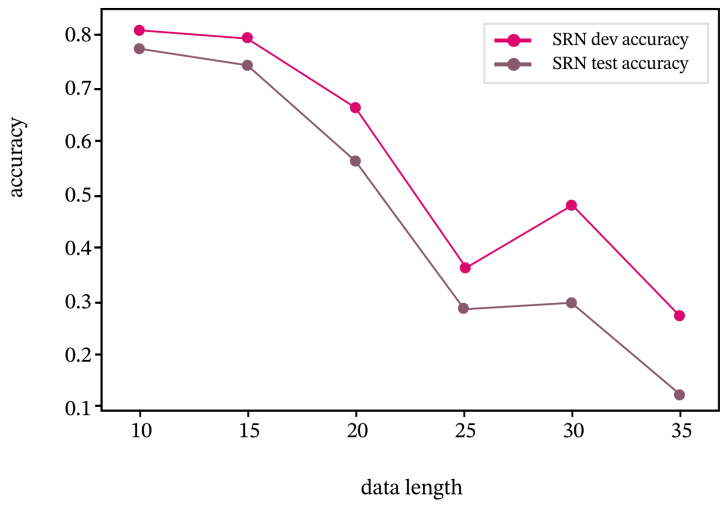


图6.7 SRN在不同长度的验证集和测试集的准确度变化图

> 动手练习 6.1 参考《神经网络与深度学习》中的公式(6.50)，改进SRN的循环单元，加入隐状态之间的残差连接，并重复数字求和实验。观察是否可以缓解长程依赖问题。

6.2 梯度爆炸实验

造成简单循环网络较难建模长程依赖问题的原因有两个：梯度爆炸和梯度消失。一般来讲，循环网络的梯度爆炸问题比较容易解决，一般通过权重衰减或梯度截断可以较好地来避免；对于梯度消失问题，更加有效的方式是改变模型，比如通过长短期记忆网络LSTM来进行缓解。

本节将首先进行复现简单循环网络中的梯度爆炸问题，然后尝试使用梯度截断的方式进行解决。这里采用长度为20的数据集进行实验，训练过程中将进行输出W,U,b的梯度向量的范数，以此来衡量梯度的变化情况。

6.2.1 梯度打印函数

使用 `custom_print_log` 实现了在训练过程中打印梯度的功能， `custom_print_log` 需要接收runner的实例，并通过 `model.named_parameters()` 获取该模型中的参数名和参数值. 这里我们分别定义 `W_list` , `U_list` 和 `b_list` ，用于分别存储训练过程中参数W,U和b的梯度范数。

In [11]:
```
W_list = []
U_list = []
b_list = []
# 计算梯度范数
def custom_print_log(runner):
    model = runner.model
    W_grad_l2, U_grad_l2, b_grad_l2 = 0, 0, 0
    for name, param in model.named_parameters():
        if name == "rnn_model.W":
            W_grad_l2 = paddle.norm(param.grad, p=2).numpy()[0]
        if name == "rnn_model.U":
            U_grad_l2 = paddle.norm(param.grad, p=2).numpy()[0]
        if name == "rnn_model.b":
            b_grad_l2 = paddle.norm(param.grad, p=2).numpy()[0]
    print(f"[Training] W_grad_l2: {W_grad_l2:.5f}, U_grad_l2: {U_grad_l2:.5f}, b_grad_l2: {b_grad_l2:.5f} ")
    W_list.append(W_grad_l2)
    U_list.append(U_grad_l2)
    b_list.append(b_grad_l2)
```

6.2.2 复现梯度爆炸现象

为了更好地复现梯度爆炸问题，使用SGD优化器将批大小和学习率调大，学习率为0.2，同时在计算交叉熵损失时，将reduction设置为sum，表示将损失进行累加。 代码实现如下：

In [ ]:
```
import os
import random
import paddle
import numpy as np

np.random.seed(0)
random.seed(0)
paddle.seed(0)

# 训练轮次
num_epochs = 50
```

```
# 学习率
lr = 0.2
# 输入数字的类别数
num_digits = 10
# 将数字映射为向量的维度
input_size = 32
# 隐状态向量的维度
hidden_size = 32
# 预测数字的类别数
num_classes = 19
# 批大小
batch_size = 64
# 模型保存目录
save_dir = "./checkpoints"


# 可以设置不同的length进行不同长度数据的预测实验
length = 20
print(f"\n====> Training SRN with data of length {length}.")

# 加载长度为length的数据
data_path = f"./datasets/{length}"
train_examples, dev_examples, test_examples = load_data(data_path)
train_set, dev_set, test_set = DigitSumDataset(train_examples), DigitSumDataset(dev_examples),DigitSumDataset(test_examples)
train_loader = paddle.io.DataLoader(train_set, batch_size=batch_size)
dev_loader = paddle.io.DataLoader(dev_set, batch_size=batch_size)
test_loader = paddle.io.DataLoader(test_set, batch_size=batch_size)
# 实例化模型
base_model = SRN(input_size, hidden_size)
model = Model_RNN4SeqClass(base_model, num_digits, input_size, hidden_size, num_classes)
# 指定优化器
optimizer = paddle.optimizer.SGD(learning_rate=lr, parameters=model.parameters())
# 定义评价指标
metric = Accuracy()
# 定义损失函数
loss_fn = nn.CrossEntropyLoss(reduction="sum")

# 基于以上组件，实例化Runner
runner = RunnerV3(model, optimizer, loss_fn, metric)

# 进行模型训练
model_save_path = os.path.join(save_dir, f"srn_explosion_model_{length}.pdparams")
runner.train(train_loader, dev_loader, num_epochs=num_epochs, eval_steps=100, log_steps=1,
             save_path=model_save_path, custom_print_log=custom_print_log)
```

接下来，可以获取训练过程中关于$\boldsymbol{W}$，$\boldsymbol{U}$和$\boldsymbol{b}$参数梯度的L2范数，并将其绘制为图片以便展示，相应代码如下：

In [ ]:
```
import matplotlib.pyplot as plt

def plot_grad(W_list, U_list, b_list, save_path, keep_steps=40):

    # 开始绘制图片
    plt.figure()
    # 默认保留前40步的结果
    steps = list(range(keep_steps))
    plt.plot(steps, W_list[:keep_steps], "r-", color="#8E004D", label="W_grad_l2")
    plt.plot(steps, U_list[:keep_steps], "-.", color="#E20079", label="U_grad_l2")
    plt.plot(steps, b_list[:keep_steps], "--", color="#3D3D3F", label="b_grad_l2")

    plt.xlabel("step")
    plt.ylabel("L2 Norm")
    plt.legend(loc="upper right")
    plt.savefig(save_path)
    print("image has been saved to: ", save_path)

save_path =  f"./images/6.8.pdf"
plot_grad(W_list, U_list, b_list, save_path)
```

图6.8 展示了在训练过程中关于$\boldsymbol{W}$，$\boldsymbol{U}$和$\boldsymbol{b}$参数梯度的L2范数，可以看到经过学习率等方式的调整，梯度范数急剧变大，而后梯度范数几乎为0. 这是因为Tanh为Sigmoid型函数，其饱和区的导数接近于0，由于梯度的急剧变化，参数数值变的较大或较小，容易落入梯度饱和区，导致梯度为0，模型很难继续训练.

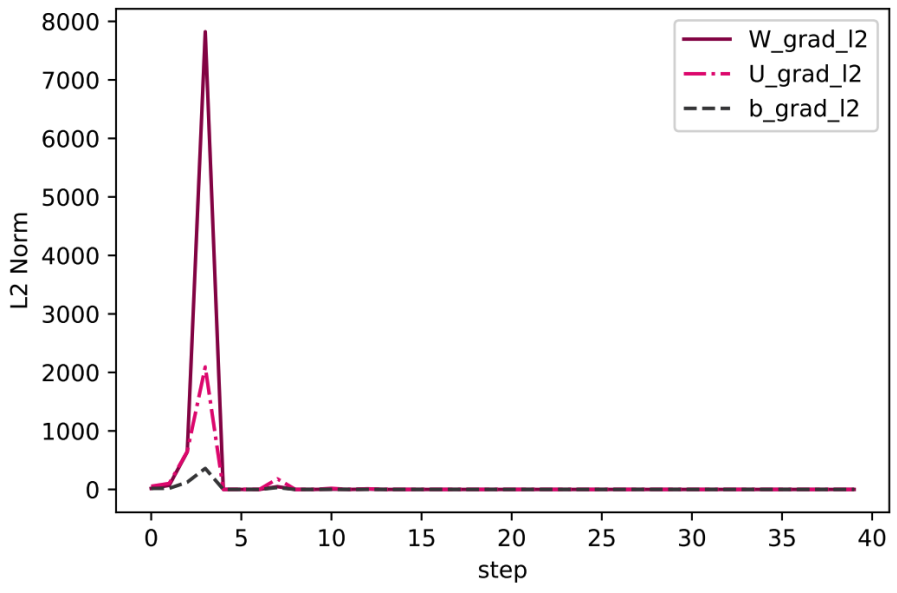


图6.8 梯度变化图

接下来，使用该模型在测试集上进行测试。

In [ ]:
```
print(f"Evaluate SRN with data length {length}.")
# 加载训练过程中效果最好的模型
model_path = os.path.join(save_dir, f"srn_explosion_model_{length}.pdparams")
runner.load_model(model_path)

# 使用测试集评价模型，获取测试集上的预测准确率
score, _ = runner.evaluate(test_loader)
print(f"[SRN] length:{length}, Score: {score: .5f}")
```

6.2.3 使用梯度截断解决梯度爆炸问题

梯度截断是一种可以有效解决梯度爆炸问题的启发式方法，当梯度的模大于一定阈值时，就将它截断成为一个较小的数。一般有两种截断方式：按值截断和按模截断．本实验使用按模截断的方式解决梯度爆炸问题。

按模截断是按照梯度向量$\boldsymbol{g}$的模进行截断，保证梯度向量的模值不大于阈值b，裁剪后的梯度为:

$$\boldsymbol{g} = \begin{cases} \boldsymbol{g}, & ||\boldsymbol{g}|| \leq b \\ \frac{b}{||\boldsymbol{g}||} * \boldsymbol{g}, & ||\boldsymbol{g}|| > b \end{cases}.$$

当梯度向量$\boldsymbol{g}$的模不大于阈值b时，$\boldsymbol{g}$数值不变，否则对$\boldsymbol{g}$进行数值缩放。

> 在飞桨中，可以使用paddle.nn.ClipGradByNorm进行按模截断. 在代码实现时，将ClipGradByNorm传入优化器，优化器在反向迭代过程中，每次梯度更新时默认可以对所有梯度裁剪。

在引入梯度截断之后，将重新观察模型的训练情况。这里我们重新实例化一下：模型和优化器，然后组装runner，进行训练。代码实现如下：

In [ ]:
```
# 清空梯度列表
W_list.clear()
U_list.clear()
b_list.clear()
# 实例化模型
base_model = SRN(input_size, hidden_size)
model = Model_RNN4SeqClass(base_model, num_digits, input_size, hidden_size, num_classes)

# 定义clip，并实例化优化器
clip = nn.ClipGradByNorm(clip_norm=5.0)
optimizer = paddle.optimizer.SGD(learning_rate=lr, parameters=model.parameters(), grad_clip=clip)
# 定义评价指标
metric = Accuracy()
# 定义损失函数
loss_fn = nn.CrossEntropyLoss(reduction="sum")

# 实例化Runner
runner = RunnerV3(model, optimizer, loss_fn, metric)

# 训练模型
model_save_path = os.path.join(save_dir, f"srn_fix_explosion_model_{length}.pdparams")
runner.train(train_loader, dev_loader, num_epochs=num_epochs, eval_steps=100, log_steps=1, save_path=model_save_path, custom_print
```

在引入梯度截断后，获取训练过程中关于$\boldsymbol{W}$，$\boldsymbol{U}$和$\boldsymbol{b}$参数梯度的L2范数，并将其绘制为图片以便展示，相应代码如下：

In [ ]:
```
save_path =  f"./images/6.9.pdf"
plot_grad(W_list, U_list, b_list, save_path, keep_steps=100)
```

图6.9 展示了引入按模截断的策略之后，模型训练时参数梯度的变化情况。可以看到，随着迭代步骤的进行，梯度始终保持在一个有值的状态，表明按模截断能够很好地解决梯度爆炸的问题.

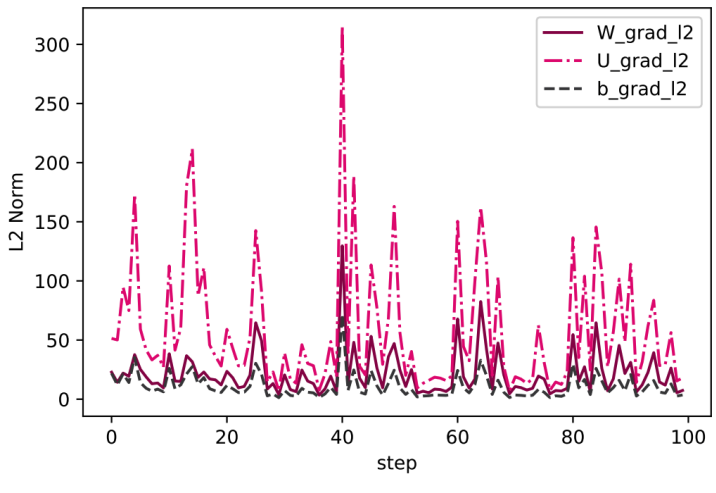


图6.9 增加梯度截断策略后，SRN参数梯度L2范数变化趋势

接下来，使用梯度截断策略的模型在测试集上进行测试。

In [24]:
```
print(f"Evaluate SRN with data length {length}.")
```

```
# 加载训练过程中效果最好的模型
model_path = os.path.join(save_dir, f"srn_fix_explosion_model_{length}.pdparams")
runner.load_model(model_path)

# 使用测试集评价模型，获取测试集上的预测准确率
score, _ = runner.evaluate(test_loader)
print(f"[SRN] length:{length}, Score: {score: .5f}")
```

由于为复现梯度爆炸现象，改变了学习率，优化器等，因此准确率相对比较低。但由于采用梯度截断策略后，在后续训练过程中，模型参数能够被更新优化，因此准确率有一定的提升。

6.3 LSTM的记忆能力实验

长短期记忆网络（Long Short-Term Memory Network，LSTM）是一种可以有效缓解长程依赖问题的循环神经网络．LSTM 的特点是引入了一个新的内部状态（Internal State）$c \in \mathbb{R}^D$ 和门控机制（Gating Mechanism）．不同时刻的内部状态以近似线性的方式进行传递，从而缓解梯度消失或梯度爆炸问题．同时门控机制进行信息筛选，可以有效地增加记忆能力．例如，输入门可以让网络忽略无关紧要的输入信息，遗忘门可以使得网络保留有用的历史信息．在上一节的数字求和任务中，如果模型能够记住前两个非零数字，同时忽略掉一些不重要的干扰信息，那么即时序列很长，模型也有效地进行预测.

LSTM 模型在第 t 步时，循环单元的内部结构如图6.10所示．

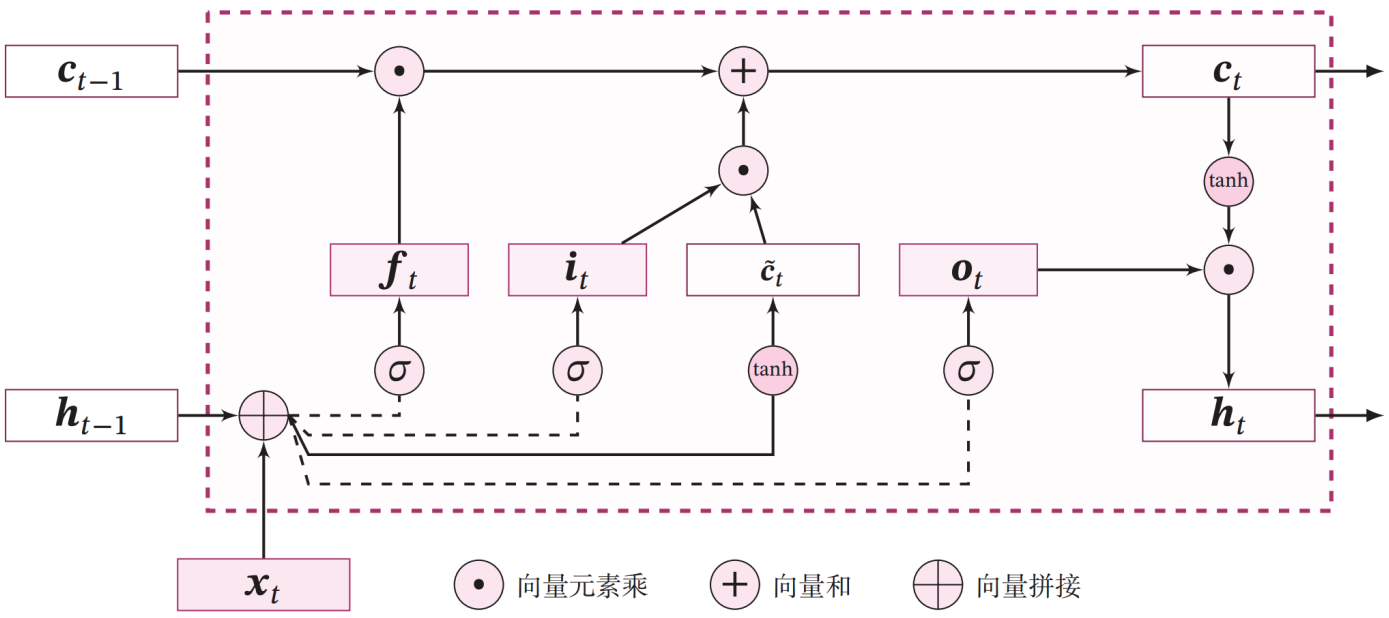


图6.10 LSTM网络的循环单元结构

> **提醒**：为了和代码的实现保存一致性，这里使用形状为 (样本数量 × 序列长度 × 特征维度) 的张量来表示一组样本.

假设一组输入序列为$\boldsymbol{X} \in \mathbb{R}^{B\times L\times M}$，其中$B$为批大小，$L$为序列长度，$M$为输入特征维度，LSTM从从左到右依次扫描序列，并通过循环单元计算更新每一时刻的状态内部状态$\boldsymbol{C}_t \in \mathbb{R}^{B\times D}$和输出状态$\boldsymbol{H}_t \in \mathbb{R}^{B\times D}$。

具体计算分为三步：

（1）计算三个“门”

在时刻t，LSTM的循环单元将当前时刻的输入$\boldsymbol{X}_t \in \mathbb{R}^{B\times M}$与上一时刻的输出状态$\boldsymbol{H}_{t-1} \in \mathbb{R}^{B\times D}$，计算一组输入门$\boldsymbol{I}_t$、遗忘门$\boldsymbol{F}_t$和输出门$\boldsymbol{O}_t$，其计算公式为

$$\begin{aligned}
\boldsymbol{I}_t &= \sigma(\boldsymbol{X}_t\boldsymbol{W}_i + \boldsymbol{H}_{t-1}\boldsymbol{U}_i + \boldsymbol{b}_i) \in \mathbb{R}^{B\times D},\\
\boldsymbol{F}_t &= \sigma(\boldsymbol{X}_t\boldsymbol{W}_f + \boldsymbol{H}_{t-1}\boldsymbol{U}_f + \boldsymbol{b}_f) \in \mathbb{R}^{B\times D},\\
\boldsymbol{O}_t &= \sigma(\boldsymbol{X}_t\boldsymbol{W}_o + \boldsymbol{H}_{t-1}\boldsymbol{U}_o + \boldsymbol{b}_o) \in \mathbb{R}^{B\times D},
\end{aligned}$$

其中$\boldsymbol{W}_* \in \mathbb{R}^{M\times D}, \boldsymbol{U}_* \in \mathbb{R}^{D\times D}, \boldsymbol{b}_* \in \mathbb{R}^D$为可学习的参数，$\sigma$表示Logistic函数，将“门”的取值控制在$(0,1)$区间。这里的“门”都是$B$个样本组成的矩阵，每一行为一个样本的“门”向量。

（2）计算内部状态

首先计算候选内部状态：

$$\tilde{\boldsymbol{C}}_t = \tanh(\boldsymbol{X}_t\boldsymbol{W}_c + \boldsymbol{H}_{t-1}\boldsymbol{U}_c + \boldsymbol{b}_c) \in \mathbb{R}^{B\times D},$$

其中$\boldsymbol{W}_c \in \mathbb{R}^{M\times D}, \boldsymbol{U}_c \in \mathbb{R}^{D\times D}, \boldsymbol{b}_c \in \mathbb{R}^D$为可学习的参数。

使用遗忘门和输入门，计算时刻t的内部状态：

$$\boldsymbol{C}_t = \boldsymbol{F}_t \odot \boldsymbol{C}_{t-1} + \boldsymbol{I}_t \odot \tilde{\boldsymbol{C}}_t,$$

其中$\odot$为逐元素积。

3）计算输出状态 当前LSTM单元状态（候选状态）的计算公式为: LSTM单元状态向量$\boldsymbol{C}_t$和$\boldsymbol{H}_t$的计算公式为

$$\boldsymbol{C}_t = \boldsymbol{F}_t \odot \boldsymbol{C}_{t-1} + \boldsymbol{I}_t \odot \tilde{\boldsymbol{C}}_t,$$
$$\boldsymbol{H}_t = \boldsymbol{O}_t \odot \tanh(\boldsymbol{C}_t).$$

LSTM循环单元结构的输入是$t-1$时刻内部状态向量$\boldsymbol{C}_{t-1} \in \mathbb{R}^{B\times D}$和隐状态向量$\boldsymbol{H}_{t-1} \in \mathbb{R}^{B\times D}$，输出是当前时刻$t$的状态向量$\boldsymbol{C}_t \in \mathbb{R}^{B\times D}$和隐状态向量$\boldsymbol{H}_t \in \mathbb{R}^{B\times D}$。通过LSTM循环单元，整个网络可以建立较长距离的时序依赖关系。

通过学习这些门的设置，LSTM可以选择性地忽略或者强化当前的记忆或是输入信息，帮助网络更好地学习长句子的语义信息。

在本节中，我们使用LSTM模型重新进行数字求和实验，验证LSTM模型的长程依赖能力。

6.3.1 模型构建

在本实验中，我们将使用第6.1.2.4节中定义Model_RNN4SeqClass模型，并构建 LSTM 算子．只需要实例化 LSTM 算，并传入Model_RNN4SeqClass模型，就可以用 LSTM 进行数字求和实验

6.3.1.1 LSTM层

LSTM层的代码与SRN层结构相似，只是在SRN层的基础上增加了内部状态、输入门、遗忘门和输出门的定义和计算。这里LSTM层的输出也依然为序列的最后一个位置的隐状态向量。代码实现如下：

In [22]:
```
import paddle.nn.functional as F
# 声明LSTM和相关参数
class LSTM(nn.Layer):
    def __init__(self, input_size, hidden_size, Wi_attr=None, Wf_attr=None, Wo_attr=None, Wc_attr=None,
                 Ui_attr=None, Uf_attr=None, Uo_attr=None, Uc_attr=None, bi_attr=None, bf_attr=None,
                 bo_attr=None, bc_attr=None):
        super(LSTM, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        # 初始化模型参数
        self.W_i = paddle.create_parameter(shape=[input_size, hidden_size], dtype="float32", attr=Wi_attr)
        self.W_f = paddle.create_parameter(shape=[input_size, hidden_size], dtype="float32", attr=Wf_attr)
        self.W_o = paddle.create_parameter(shape=[input_size, hidden_size], dtype="float32", attr=Wo_attr)
        self.W_c = paddle.create_parameter(shape=[input_size, hidden_size], dtype="float32", attr=Wc_attr)
        self.U_i = paddle.create_parameter(shape=[hidden_size, hidden_size], dtype="float32", attr=Ui_attr)
        self.U_f = paddle.create_parameter(shape=[hidden_size, hidden_size], dtype="float32", attr=Uf_attr)
        self.U_o = paddle.create_parameter(shape=[hidden_size, hidden_size], dtype="float32", attr=Uo_attr)
        self.U_c = paddle.create_parameter(shape=[hidden_size, hidden_size], dtype="float32", attr=Uc_attr)
        self.b_i = paddle.create_parameter(shape=[1, hidden_size], dtype="float32", attr=bi_attr)
        self.b_f = paddle.create_parameter(shape=[1, hidden_size], dtype="float32", attr=bf_attr)
        self.b_o = paddle.create_parameter(shape=[1, hidden_size], dtype="float32", attr=bo_attr)
        self.b_c = paddle.create_parameter(shape=[1, hidden_size], dtype="float32", attr=bc_attr)

    # 初始化状态向量和隐状态向量
    def init_state(self, batch_size):
        hidden_state = paddle.zeros(shape=[batch_size, self.hidden_size], dtype="float32")
        cell_state = paddle.zeros(shape=[batch_size, self.hidden_size], dtype="float32")
        return hidden_state, cell_state

    # 定义前向计算
    def forward(self, inputs, states=None):
        # inputs: 输入数据，其shape为batch_size x seq_len x input_size
        batch_size, seq_len, input_size = inputs.shape

        # 初始化起始的单元状态和隐状态向量，其shape为batch_size x hidden_size
        if states is None:
            states = self.init_state(batch_size)
        hidden_state, cell_state = states

        # 执行LSTM计算，包括：输入门、遗忘门和输出门、候选内部状态、内部状态和隐状态向量
        for step in range(seq_len):
            # 获取当前时刻的输入数据step_input: 其shape为batch_size x input_size
            step_input = inputs[:, step, :]
            # 计算输入门，遗忘门和输出门，其shape为：batch_size x hidden_size
            I_gate = F.sigmoid(paddle.matmul(step_input, self.W_i) + paddle.matmul(hidden_state, self.U_i) + self.b_i)
            F_gate = F.sigmoid(paddle.matmul(step_input, self.W_f) + paddle.matmul(hidden_state, self.U_f) + self.b_f)
            O_gate = F.sigmoid(paddle.matmul(step_input, self.W_o) + paddle.matmul(hidden_state, self.U_o) + self.b_o)
            # 计算候选状态向量，其shape为：batch_size x hidden_size
            C_tilde = F.tanh(paddle.matmul(step_input, self.W_c) + paddle.matmul(hidden_state, self.U_c) + self.b_c)
            # 计算单元状态向量，其shape为：batch_size x hidden_size
            cell_state = F_gate * cell_state + I_gate * C_tilde
            # 计算隐状态向量，其shape为：batch_size x hidden_size
            hidden_state = O_gate * F.tanh(cell_state)

        return hidden_state
```

In [10]:
```
Wi_attr = paddle.ParamAttr(initializer=nn.initializer.Assign([[0.1, 0.2], [0.1, 0.2]]))
Wf_attr = paddle.ParamAttr(initializer=nn.initializer.Assign([[0.1, 0.2], [0.1, 0.2]]))
Wo_attr = paddle.ParamAttr(initializer=nn.initializer.Assign([[0.1, 0.2], [0.1, 0.2]]))
Wc_attr = paddle.ParamAttr(initializer=nn.initializer.Assign([[0.1, 0.2], [0.1, 0.2]]))
Ui_attr = paddle.ParamAttr(initializer=nn.initializer.Assign([[0.0, 0.1], [0.1, 0.0]]))
Uf_attr = paddle.ParamAttr(initializer=nn.initializer.Assign([[0.0, 0.1], [0.1, 0.0]]))
Uo_attr = paddle.ParamAttr(initializer=nn.initializer.Assign([[0.0, 0.1], [0.1, 0.0]]))
Uc_attr = paddle.ParamAttr(initializer=nn.initializer.Assign([[0.0, 0.1], [0.1, 0.0]]))
bi_attr = paddle.ParamAttr(initializer=nn.initializer.Assign([[0.1, 0.1]]))
```

```
bf_attr = paddle.ParamAttr(initializer=nn.initializer.Assign([[0.1, 0.1]]))
bo_attr = paddle.ParamAttr(initializer=nn.initializer.Assign([[0.1, 0.1]]))
bc_attr = paddle.ParamAttr(initializer=nn.initializer.Assign([[0.1, 0.1]]))

lstm = LSTM(2, 2, Wi_attr=Wi_attr, Wf_attr=Wf_attr, Wo_attr=Wo_attr, Wc_attr=Wc_attr,
                  Ui_attr=Ui_attr, Uf_attr=Uf_attr, Uo_attr=Uo_attr, Uc_attr=Uc_attr,
                  bi_attr=bi_attr, bf_attr=bf_attr, bo_attr=bo_attr, bc_attr=bc_attr)

inputs = paddle.to_tensor([[[1, 0]]], dtype="float32")
hidden_state = lstm(inputs)
print(hidden_state)
```

飞桨框架已经内置了LSTM的API `paddle.nn.LSTM`，其与自己实现的SRN不同点在于其实现时采用了两个偏置，同时矩阵相乘时参数在输入数据前面，如下公式所示：。

$$
\begin{aligned}
\boldsymbol{I}_t &= \sigma(\boldsymbol{W}_{ii}\boldsymbol{X}_t + \boldsymbol{b}_{ii} + \boldsymbol{U}_{hi}\boldsymbol{H}_{t-1} + \boldsymbol{b}_{hi}) \\
\boldsymbol{F}_t &= \sigma(\boldsymbol{W}_{if}\boldsymbol{X}_t + \boldsymbol{b}_{if} + \boldsymbol{U}_{hf}\boldsymbol{H}_{t-1} + \boldsymbol{b}_{hf}) \\
\boldsymbol{O}_t &= \sigma(\boldsymbol{W}_{io}\boldsymbol{X}_t + \boldsymbol{b}_{io} + \boldsymbol{U}_{ho}\boldsymbol{H}_{t-1} + \boldsymbol{b}_{ho}), \\
\tilde{\boldsymbol{C}}_t &= \tanh(\boldsymbol{W}_{ic}\boldsymbol{X}_t + \boldsymbol{b}_{ic} + \boldsymbol{U}_{hc}\boldsymbol{H}_{t-1} + \boldsymbol{b}_{hc}), \\
\boldsymbol{C}_t &= \boldsymbol{F}_t \cdot \boldsymbol{C}_{t-1} + \boldsymbol{I}_t \cdot \tilde{\boldsymbol{C}}_t, \\
\boldsymbol{H}_t &= \boldsymbol{O}_t \cdot \tanh(\boldsymbol{C}_t).
\end{aligned}
$$

其中$\boldsymbol{W}_* \in \mathbb{R}^{M\times D}, \boldsymbol{U}_* \in \mathbb{R}^{D\times D}, \boldsymbol{b}_{i*} \in \mathbb{R}^{1\times D}, \boldsymbol{b}_{h*} \in \mathbb{R}^{1\times D}$是可学习参数。

另外，在Paddle内置LSTM实现时，对于参数$\boldsymbol{W}_{ii}, \boldsymbol{W}_{if}, \boldsymbol{W}_{io}, \boldsymbol{W}_{ic}$，并不是分别申请这些矩阵，而是申请了一个大的矩阵$\boldsymbol{W}_{ih}$，将这个大的矩阵分割为4份，便可以得到$\boldsymbol{W}_{ii}, \boldsymbol{W}_{if}, \boldsymbol{W}_{ic}, \boldsymbol{W}_{io}$。同理，将会得到$\boldsymbol{W}_{hh}, \boldsymbol{b}_{ih}$和$\boldsymbol{b}_{hh}$.

最后，Paddle内置LSTM API将会返回参数序列向量outputs和最后时刻的状态向量，其中序列向量outputs是指最后一层SRN的输出向量，其shape为[batch_size, seq_len, num_directions *hidden_size]；最后时刻的状态向量是个元组，其包含了两个向量，分别是隐状态向量和单元状态向量，其shape均为[num_layers* num_directions, batch_size, hidden_size]。

这里我们可以将自己实现的SRN和Paddle框架内置的SRN返回的结果进行打印展示，实现代码如下。

In [11]:
```
# 这里创建一个随机数组作为测试数据，数据shape为batch_size x seq_len x input_size
batch_size, seq_len, input_size = 8, 20, 32
inputs = paddle.randn(shape=[batch_size, seq_len, input_size])

# 设置模型的hidden_size
hidden_size = 32
paddle_lstm = nn.LSTM(input_size, hidden_size)
self_lstm = LSTM(input_size, hidden_size)

self_hidden_state = self_lstm(inputs)
paddle_outputs, (paddle_hidden_state, paddle_cell_state) = paddle_lstm(inputs)

print("self_lstm hidden_state: ", self_hidden_state.shape)
print("paddle_lstm outpus:", paddle_outputs.shape)
print("paddle_lstm hidden_state:", paddle_hidden_state.shape)
print("paddle_lstm cell_state:", paddle_cell_state.shape)
```

可以看到，自己实现的LSTM由于没有考虑多层因素，因此没有层次这个维度，因此其输出shape为[8, 32]。同时由于在以上代码使用Paddle内置API实例化LSTM时，默认定义的是1层的单向SRN，因此其shape为[1, 8, 32]，同时隐状态向量为[8,20, 32].

接下来，我们可以将自己实现的LSTM与Paddle内置的LSTM在输出值的精度上进行对比，这里首先根据Paddle内置的LSTM实例化模型（为了进行对比，在实例化时只保留一个偏置，将偏置b_{ih}设置为0），然后提取该模型对应的参数，进行参数分割后，使用相应参数去初始化自己实现的LSTM，从而保证两者在参数初始化时是一致的。

在进行实验时，首先定义输入数据 `inputs`，然后将该数据分别传入Paddle内置的LSTM与自己实现的LSTM模型中，最后通过对比两者的隐状态输出向量。代码实现如下：

In [12]:
```
import paddle
paddle.seed(0)

# 这里创建一个随机数组作为测试数据，数据shape为batch_size x seq_len x input_size
batch_size, seq_len, input_size, hidden_size = 2, 5, 10, 10
inputs = paddle.randn(shape=[batch_size, seq_len, input_size])

# 设置模型的hidden_size
bih_attr = paddle.ParamAttr(initializer=nn.initializer.Assign(paddle.zeros([4*hidden_size, ])))
paddle_lstm = nn.LSTM(input_size, hidden_size, bias_ih_attr=bih_attr)

# 获取paddle_lstm中的参数，并设置相应的paramAttr,用于初始化lstm
print(paddle_lstm.weight_ih_l0.T.shape)
chunked_W = paddle.split(paddle_lstm.weight_ih_l0.T, num_or_sections=4, axis=-1)
chunked_U = paddle.split(paddle_lstm.weight_hh_l0.T, num_or_sections=4, axis=-1)
chunked_b = paddle.split(paddle_lstm.bias_hh_l0.T, num_or_sections=4, axis=-1)

Wi_attr = paddle.ParamAttr(initializer=nn.initializer.Assign(chunked_W[0]))
Wf_attr = paddle.ParamAttr(initializer=nn.initializer.Assign(chunked_W[1]))
Wc_attr = paddle.ParamAttr(initializer=nn.initializer.Assign(chunked_W[2]))
Wo_attr = paddle.ParamAttr(initializer=nn.initializer.Assign(chunked_W[3]))
Ui_attr = paddle.ParamAttr(initializer=nn.initializer.Assign(chunked_U[0]))
Uf_attr = paddle.ParamAttr(initializer=nn.initializer.Assign(chunked_U[1]))
```

```
Uc_attr = paddle.ParamAttr(initializer=nn.initializer.Assign(chunked_U[2]))
Uo_attr = paddle.ParamAttr(initializer=nn.initializer.Assign(chunked_U[3]))
bi_attr = paddle.ParamAttr(initializer=nn.initializer.Assign(chunked_b[0]))
bf_attr = paddle.ParamAttr(initializer=nn.initializer.Assign(chunked_b[1]))
bc_attr = paddle.ParamAttr(initializer=nn.initializer.Assign(chunked_b[2]))
bo_attr = paddle.ParamAttr(initializer=nn.initializer.Assign(chunked_b[3]))
self_lstm = LSTM(input_size, hidden_size, Wi_attr=Wi_attr, Wf_attr=Wf_attr, Wo_attr=Wo_attr, Wc_attr=Wc_attr,
                 Ui_attr=Ui_attr, Uf_attr=Uf_attr, Uo_attr=Uo_attr, Uc_attr=Uc_attr,
                 bi_attr=bi_attr, bf_attr=bf_attr, bo_attr=bo_attr, bc_attr=bc_attr)

# 进行前向计算，获取隐状态向量，并打印展示
self_hidden_state = self_lstm(inputs)
paddle_outputs, (paddle_hidden_state, _) = paddle_lstm(inputs)
print("paddle SRN:\n", paddle_hidden_state.numpy().squeeze(0))
print("self SRN:\n", self_hidden_state.numpy())
```

可以看到，两者的输出基本是一致的。另外，还可以进行对比两者在运算速度方面的差异。代码实现如下：

In [23]:
```
import time

# 这里创建一个随机数组作为测试数据，数据shape为batch_size x seq_len x input_size
batch_size, seq_len, input_size = 8, 20, 32
inputs = paddle.randn(shape=[batch_size, seq_len, input_size])

# 设置模型的hidden_size
hidden_size = 32
self_lstm = LSTM(input_size, hidden_size)
paddle_lstm = nn.LSTM(input_size, hidden_size)

# 计算自己实现的SRN运算速度
model_time = 0
for i in range(100):
    strat_time = time.time()
    hidden_state = self_lstm(inputs)
    # 预热10次运算，不计入最终速度统计
    if i < 10:
        continue
    end_time = time.time()
    model_time += (end_time - strat_time)
avg_model_time = model_time / 90
print('self_lstm speed:', avg_model_time, 's')

# 计算Paddle内置的SRN运算速度
model_time = 0
for i in range(100):
    strat_time = time.time()
    outputs, (hidden_state, cell_state) = paddle_lstm(inputs)
    # 预热10次运算，不计入最终速度统计
    if i < 10:
        continue
    end_time = time.time()
    model_time += (end_time - strat_time)
avg_model_time = model_time / 90
print('paddle_lstm speed:', avg_model_time, 's')
```

可以看到，由于Paddle框架的LSTM底层采用了C++实现并进行优化，Paddle框架内置的LSTM运行效率远远高于自己实现的LSTM。

6.3.1.2 模型汇总

在本节实验中，我们将使用6.1.2.4的Model_RNN4SeqClass作为预测模型，不同在于在实例化时将传入实例化的LSTM层。

> 动手联系6.2 在我们手动实现的LSTM算子中，是逐步计算每个时刻的隐状态。请思考如何实现更加高效的LSTM算子。

6.3.2 模型训练

6.3.2.1 训练指定长度的数字预测模型

本节将基于RunnerV3类进行训练，首先定义模型训练的超参数，并保证和简单循环网络的超参数一致. 然后定义一个 `train` 函数，其可以通过指定长度的数据集，并进行训练. 在 `train` 函数中，首先加载长度为 `length` 的数据，然后实例化各项组件并创建对应的Runner，然后训练该Runner。同时在本节将使用4.5.4节定义的准确度（Accuracy）作为评估指标，代码实现如下：

In [18]:
```
import os
import random
import paddle
import numpy as np
from nndl import RunnerV3

# 训练轮次
num_epochs = 500
# 学习率
lr = 0.001
# 输入数字的类别数
num_digits = 10
# 将数字映射为向量的维度
input_size = 32
# 隐状态向量的维度
hidden_size = 32
```

```
# 预测数字的类别数
num_classes = 19
# 批大小
batch_size = 8
# 模型保存目录
save_dir = "./checkpoints"

# 可以设置不同的length进行不同长度数据的预测实验
def train(length):
    print(f"\n====> Training LSTM with data of length {length}.")
    np.random.seed(0)
    random.seed(0)
    paddle.seed(0)

    # 加载长度为length的数据
    data_path = f"./datasets/{length}"
    train_examples, dev_examples, test_examples = load_data(data_path)
    train_set, dev_set, test_set = DigitSumDataset(train_examples), DigitSumDataset(dev_examples), DigitSumDataset(test_examples)
    train_loader = paddle.io.DataLoader(train_set, batch_size=batch_size)
    dev_loader = paddle.io.DataLoader(dev_set, batch_size=batch_size)
    test_loader = paddle.io.DataLoader(test_set, batch_size=batch_size)
    # 实例化模型
    base_model = LSTM(input_size, hidden_size)
    model = Model_RNN4SeqClass(base_model, num_digits, input_size, hidden_size, num_classes)
    # 指定优化器
    optimizer = paddle.optimizer.Adam(learning_rate=lr, parameters=model.parameters())
    # 定义评价指标
    metric = Accuracy()
    # 定义损失函数
    loss_fn = paddle.nn.CrossEntropyLoss()
    # 基于以上组件，实例化Runner
    runner = RunnerV3(model, optimizer, loss_fn, metric)

    # 进行模型训练
    model_save_path = os.path.join(save_dir, f"best_lstm_model_{length}.pdparams")
    runner.train(train_loader, dev_loader, num_epochs=num_epochs, eval_steps=100, log_steps=100, save_path=model_save_path)

    return runner
```

6.3.2.2 多组训练

接下来，分别进行数据长度为10, 15, 20, 25, 30, 35的数字预测模型训练实验，训练后的 `runner` 保存至 `runners` 字典中。

In [19]:
```
lstm_runners = {}

lengths = [10, 15, 20, 25, 30, 35]
for length in lengths:
    runner = train(length)
    lstm_runners[length] = runner
```

6.3.2.3 损失曲线展示

分别画出基于LSTM的各个长度的数字预测模型训练过程中，在训练集和验证集上的损失曲线，代码实现如下：

In [20]:
```
# 画出训练过程中的损失图
for length in lengths:
    runner = lstm_runners[length]
    fig_name = f"./images/6.11_{length}.pdf"
    plot_training_loss(runner, fig_name, sample_step=100)
```

图6.11展示了LSTM模型在不同长度数据集上进行训练后的损失变化，同SRN模型一样，随着序列长度的增加，训练集上的损失逐渐不稳定，验证集上的损失整体趋向于变大，这说明当序列长度增加时，保持长期依赖的能力同样在逐渐变弱. 同图6.5相比，LSTM模型在序列长度增加时，收敛情况比SRN模型更好。

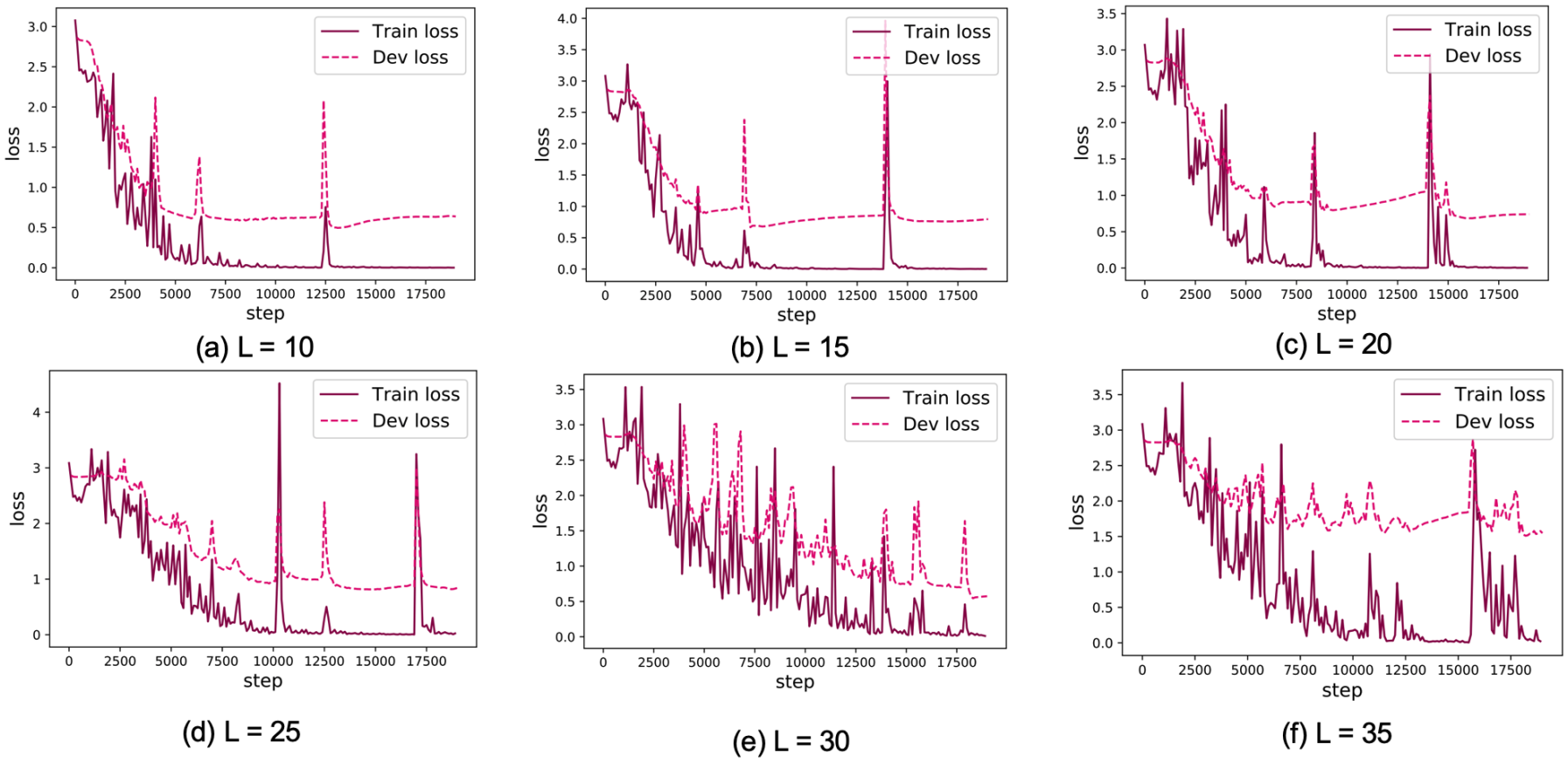


图6.11 LSTM在不同长度数据集训练损失变化图

> **动手练习6.3**： 改进第6.3.1.1节中的 LSTM 算子，使其可以支持双向 LSTM 模型的计算

6.3.3 模型评价

6.3.3.1 在测试集上进行模型评价

使用测试数据对在训练过程中保存的最好模型进行评价，观察模型在测试集上的准确率. 同时获取模型在训练过程中在验证集上最好的准确率，实现代码如下:

In [21]:
```
lstm_dev_scores = []
lstm_test_scores = []
for length in lengths:
    print(f"Evaluate LSTM with data length {length}.")
    runner = lstm_runners[length]
    # 加载训练过程中效果最好的模型
    model_path = os.path.join(save_dir, f"best_lstm_model_{length}.pdparams")
    runner.load_model(model_path)

    # 加载长度为length的数据
    data_path = f"./datasets/{length}"
    train_examples, dev_examples, test_examples = load_data(data_path)
    test_set = DigitSumDataset(test_examples)
    test_loader = paddle.io.DataLoader(test_set, batch_size=batch_size)

    # 使用测试集评价模型，获取测试集上的预测准确率
    score, _ = runner.evaluate(test_loader)
    lstm_test_scores.append(score)
    lstm_dev_scores.append(max(runner.dev_scores))

for length, dev_score, test_score in zip(lengths, lstm_dev_scores, lstm_test_scores):
    print(f"[LSTM] length:{length}, dev_score: {dev_score}, test_score: {test_score: .5f}")
```

6.3.3.2 模型在不同长度的数据集上的准确率变化图

接下来，将SRN和LSTM在不同长度的验证集和测试集数据上的准确率绘制成图片，以方面观察。

In [ ]:
```
import matplotlib.pyplot as plt

plt.plot(lengths, srn_dev_scores, '-o', color='#E20079',  label="SRN Dev Accuracy")
plt.plot(lengths, srn_test_scores,'-o', color='#946279', label="SRN Test Accuracy")
plt.plot(lengths, lstm_dev_scores, '-o', color='#8E004D',  label="LSTM Dev Accuracy")
plt.plot(lengths, lstm_test_scores,'-o', color='#3D3D3F', label="LSTM Test Accuracy")

#绘制坐标轴和图例
plt.ylabel("loss", fontsize='x-large')
plt.xlabel("step", fontsize='x-large')
plt.legend(loc='lower left')

fig_name = "./images/6.12.pdf"
plt.savefig(fig_name)
plt.show()
```

图6.12 展示了LSTM模型与SRN模型在不同长度数据集上的准确度对比。随着数据集长度的增加，LSTM模型在验证集和测试集上的准确率整体也趋向于降低；同时LSTM模型的准确率显著高于SRN模型，表明LSTM模型保持长期依赖的能力要优于SRN模型.

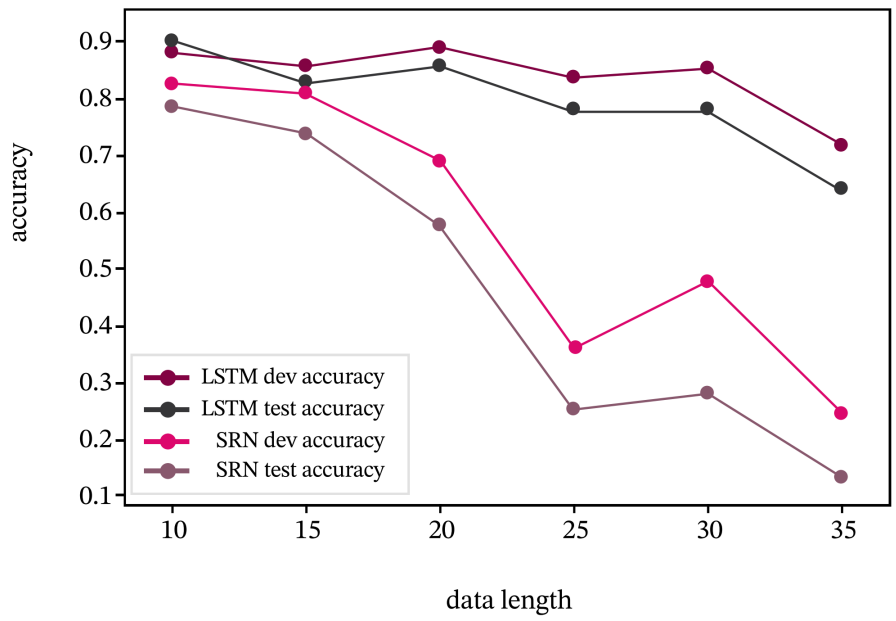


图6.12 LSTM与SRN网络在不同长度数据集上的准确度对比图

> **动手练习6.4**： 请实现 GRU 算子，完成上面实验，并对比 GRU 和 LSTM 的实验效果

6.3.3.3 LSTM模型门状态和单元状态的变化

LSTM模型通过门控机制控制信息的单元状态的更新，这里可以观察当LSTM在处理一条数字序列的时候，相应门和单元状态是如何变化的。首先需要对以上LSTM模型实现代码中，定义相应列表进行存储这些门和单元状态在每个时刻的向量。

In [ ]:
```
import paddle.nn.functional as F
# 声明LSTM和相关参数
class LSTM(nn.Layer):
    def __init__(self, input_size, hidden_size, para_attr=paddle.ParamAttr(initializer=nn.initializer.XavierUniform())):
        super(LSTM, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        # 初始化模型参数
        self.W_i = paddle.create_parameter(shape=[input_size, hidden_size], dtype="float32", attr=para_attr)
        self.W_f = paddle.create_parameter(shape=[input_size, hidden_size], dtype="float32", attr=para_attr)
        self.W_o = paddle.create_parameter(shape=[input_size, hidden_size], dtype="float32", attr=para_attr)
        self.W_a = paddle.create_parameter(shape=[input_size, hidden_size], dtype="float32", attr=para_attr)
        self.U_i = paddle.create_parameter(shape=[hidden_size, hidden_size], dtype="float32", attr=para_attr)
        self.U_f = paddle.create_parameter(shape=[hidden_size, hidden_size], dtype="float32", attr=para_attr)
        self.U_o = paddle.create_parameter(shape=[hidden_size, hidden_size], dtype="float32", attr=para_attr)
        self.U_a = paddle.create_parameter(shape=[hidden_size, hidden_size], dtype="float32", attr=para_attr)
        self.b_i = paddle.create_parameter(shape=[1, hidden_size], dtype="float32", attr=para_attr)
        self.b_f = paddle.create_parameter(shape=[1, hidden_size], dtype="float32", attr=para_attr)
        self.b_o = paddle.create_parameter(shape=[1, hidden_size], dtype="float32", attr=para_attr)
        self.b_a = paddle.create_parameter(shape=[1, hidden_size], dtype="float32", attr=para_attr)

    # 初始化状态向量和隐状态向量
    def init_state(self, batch_size):
        hidden_state = paddle.zeros(shape=[batch_size, self.hidden_size], dtype="float32")
        cell_state = paddle.zeros(shape=[batch_size, self.hidden_size], dtype="float32")
        return hidden_state, cell_state

    # 定义前向计算
    def forward(self, inputs, states=None):
        batch_size, seq_len, input_size = inputs.shape  # inputs batch_size x seq_len x input_size

        if states is None:
            states = self.init_state(batch_size)
        hidden_state, cell_state = states

        # 定义相应的门状态和单元状态向量列表
        self.Is = []
        self.Fs = []
        self.Os = []
        self.Cs = []
        # 初始化状态向量和隐状态向量
        cell_state = paddle.zeros(shape=[batch_size, self.hidden_size], dtype="float32")
        hidden_state = paddle.zeros(shape=[batch_size, self.hidden_size], dtype="float32")

        # 执行LSTM计算，包括：隐藏门、输入门、遗忘门、候选状态向量、状态向量和隐状态向量
        for step in range(seq_len):
            input_step = inputs[:, step, :]
            I_gate = F.sigmoid(paddle.matmul(input_step, self.W_i) + paddle.matmul(hidden_state, self.U_i) + self.b_i)
            F_gate = F.sigmoid(paddle.matmul(input_step, self.W_f) + paddle.matmul(hidden_state, self.U_f) + self.b_f)
            O_gate = F.sigmoid(paddle.matmul(input_step, self.W_o) + paddle.matmul(hidden_state, self.U_o) + self.b_o)
            C_tilde = F.tanh(paddle.matmul(input_step, self.W_a) + paddle.matmul(hidden_state, self.U_a) + self.b_a)
            cell_state = F_gate * cell_state + I_gate * C_tilde
            hidden_state = O_gate * F.tanh(cell_state)
```

```
            # 存储门状态向量和单元状态向量
            self.Is.append(I_gate.numpy().copy())
            self.Fs.append(F_gate.numpy().copy())
            self.Os.append(O_gate.numpy().copy())
            self.Cs.append(cell_state.numpy().copy())
        return hidden_state
```

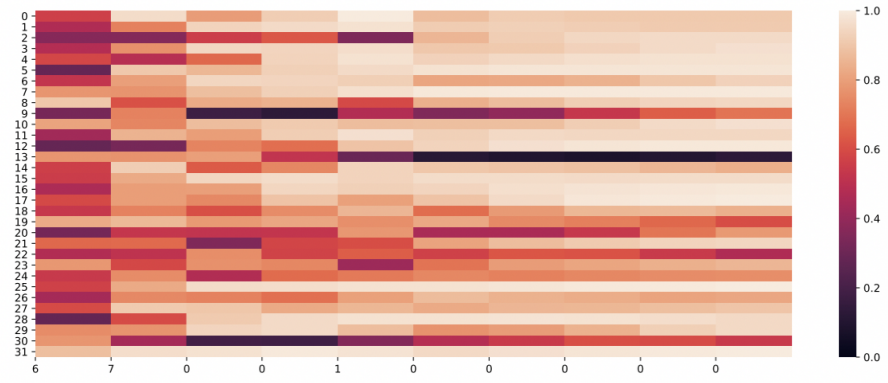
接下来，需要使用新的LSTM模型，重新实例化一个runner，本节使用序列长度为10的模型进行此项实验，因此需要加载序列长度为10的模型。

In [ ]:
```
# 实例化模型
base_model = LSTM(input_size, hidden_size)
model = Model_RNN4SeqClass(base_model, num_digits, input_size, hidden_size, num_classes)
# 指定优化器
optimizer = paddle.optimizer.Adam(learning_rate=learning_rate, parameters=model.parameters())
# 定义评价指标
metric = Accuracy()
# 定义损失函数
loss_fn = paddle.nn.CrossEntropyLoss()
# 基于以上组件，重新实例化Runner
runner = RunnerV3(model, optimizer, loss_fn, metric)

length = 10
# 加载训练过程中效果最好的模型
model_path = os.path.join(save_dir, f"best_lstm_model_{length}.pdparams")
runner.load_model(model_path)
```

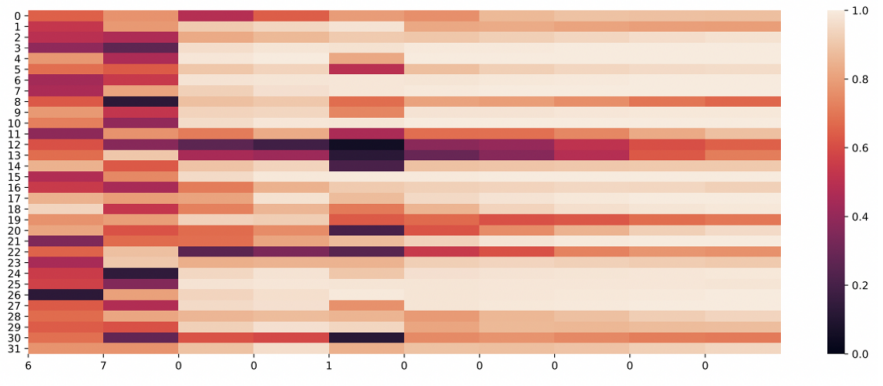
接下来，给定一条数字序列，并使用数字预测模型进行数字预测，这样便会将相应的门状态和单元状态向量保存至模型中.然后分别从模型中取出这些向量，并将这些向量进行绘制展示。代码实现如下：

In [23]:
```
import seaborn as sns

def plot_tensor(inputs, tensor,  save_path, vmin=0, vmax=1):
    tensor = np.stack(tensor, axis=0)
    tensor = np.squeeze(tensor, 1).T

    plt.figure(figsize=(16,6))
    # vmin, vmax定义了色彩图的上下界
    ax = sns.heatmap(tensor, vmin=vmin, vmax=vmax)
    ax.set_xticklabels(inputs)
    ax.figure.savefig(save_path)


# 定义模型输入
inputs = [6, 7, 0, 0, 1, 0, 0, 0, 0, 0]
X = paddle.to_tensor(inputs.copy())
X = X.unsqueeze(0)
# 进行模型预测，并获取相应的预测结果
logits = runner.predict(X)
predict_label = paddle.argmax(logits, axis=-1)
print(f"predict result: {predict_label.numpy()[0]}")

# 输入门
Is = runner.model.model.Is
plot_tensor(inputs, Is, save_path="./images/6.13_I.pdf")
# 遗忘门
Fs = runner.model.model.Fs
plot_tensor(inputs, Fs, save_path="./images/6.13_F.pdf")
# 输出门
Os = runner.model.model.Os
plot_tensor(inputs, Os, save_path="./images/6.13_O.pdf")
# 单元状态
Cs = runner.model.model.Cs
plot_tensor(inputs, Cs, save_path="./images/6.13_C.pdf", vmin=-5, vmax=5)
```

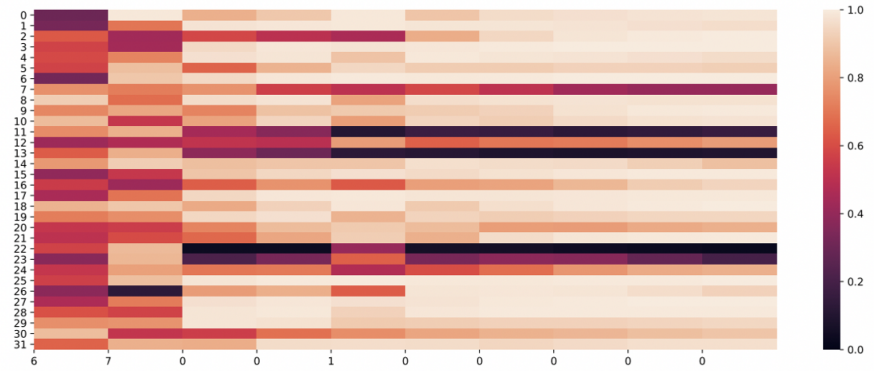
图6.13 当LSTM处理序列数据[6, 7, 0, 0, 1, 0, 0, 0, 0, 0]的过程中单元状态和门数值的变化图，其中横坐标为输入数字，纵坐标为相应门或单元状态向量的维度，颜色的深浅代表数值的大小。可以看到，当输入门遇到不同位置的数字0时，保持了相对一致的数值大小，表明对于0元素保持相同的门控过滤机制，避免输入信息的变化给当前模型带来困扰；当遗忘门遇到数字1后，遗忘门数值在一些维度上变小，表明对某些信息进行了遗忘；随着序列的输入，输出门和单元状态在某些维度上数值变小，在某些维度上数值变大，表明输出门在根据信息的重要性选择信息进行输出，同时单元状态也在保持着对文本预测重要的一些信息.



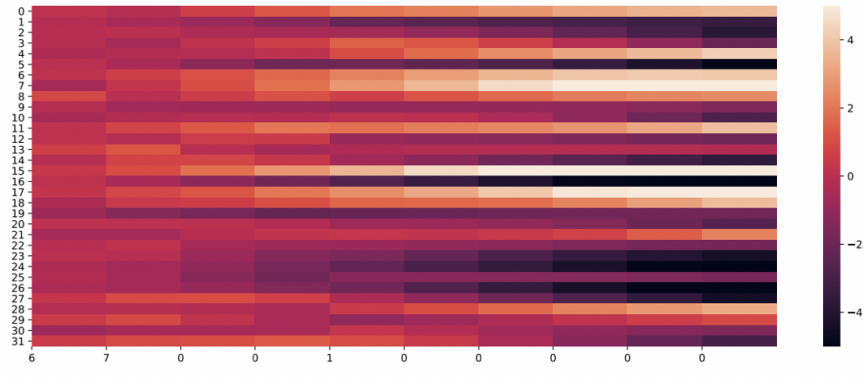
(a) 输入门



(b) 遗忘门



(c) 输出门



(d) 单元状态

图6.13 LSTM中单元状态和门数值的变化图