RNN_LSTM-report-zh

RNN和LSTM的核心实现

RNN：

我们这里使用Truncated BPTT（截断式的基于时间的反向传播），给定数据是一个一维的正弦波，要求对这个时间序列进行预测。而RNN的结构非常简单，只有一个线性层+tanh，参数的话只有两个矩阵，做一下循环变换就行了。但是显然由于是循环变化，我们的model需要使用多次，这与之前把x丢进model就能产生y的不同。假如我们需要反复利用model倒也行，计算图将会自动建立，而一旦我们从下次开始建图时，假如不把图切断，计算图将会继续扩建！

所以我们要添加一个切断函数：

代码块

```
class Variable:
    ......
    def unchain_backward(self):
        if self.creator is not None:
            funcs = [self.creator]
            while funcs:
                f = funcs.pop()
                for x in f.inputs:
                    if x.creator is not None:
                        funcs.append(x.creator)
                        x.unchain()
```

这个函数与之前的反向传播函数类似，但是不需要考虑拓扑序，全部切断就行！

整体的RNN代码非常简单：

代码块

```
if '__file__' in globals():
    import os, sys
    sys.path.append(os.path.join(os.path.dirname(__file__), '..'))
import numpy as np
import matplotlib.pyplot as plt
import dezero
from dezero import Model
import dezero.functions as F
import dezero.layers as L

```

```python
# Hyperparameters
max_epoch = 100
hidden_size = 100
bptt_length = 30

train_set = dezero.datasets.SinCurve(train=True)
seqlen = len(train_set)


class SimpleRNN(Model):
    def __init__(self, hidden_size, out_size):
        super().__init__()
        self.rnn = L.RNN(hidden_size)
        self.fc = L.Linear(out_size)

    def reset_state(self):
        self.rnn.reset_state()

    def __call__(self, x):
        h = self.rnn(x)
        y = self.fc(h)
        return y


model = SimpleRNN(hidden_size, 1)
optimizer = dezero.optimizers.Adam().setup(model)

# Start training.
for epoch in range(max_epoch):
    model.reset_state()
    loss, count = 0, 0

    for x, t in train_set:
        x = x.reshape(1, 1)
        y = model(x)
        loss += F.mean_squared_error(y, t)
        count += 1

        if count % bptt_length == 0 or count == seqlen:
            model.cleargrads()
            loss.backward()
            loss.unchain_backward()
            optimizer.update()

    avg_loss = float(loss.data) / count
    print('| epoch %d | loss %f' % (epoch + 1, avg_loss))

```

```python
# Plot
xs = np.cos(np.linspace(0, 4 * np.pi, 1000))
model.reset_state()
pred_list = []

with dezero.no_grad():
    for x in xs:
        x = np.array(x).reshape(1, 1)
        y = model(x)
        pred_list.append(float(y.data))

plt.plot(np.arange(len(xs)), xs, label='y=cos(x)')
plt.plot(np.arange(len(xs)), pred_list, label='predict')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

注意，loss是累积的，具体原因不详，但说是可以识别到长度性质。

LSTM

有如下公式，f是遗忘门，i是输入门，o是输出门，h是隐藏层，c是当前层

$$\begin{aligned}
f_t &= \sigma(x_t W_x^{(f)} + h_{t-1} W_h^{(f)} + b^{(f)}) \\
i_t &= \sigma(x_t W_x^{(i)} + h_{t-1} W_h^{(i)} + b^{(i)}) \\
o_t &= \sigma(x_t W_x^{(o)} + h_{t-1} W_h^{(o)} + b^{(o)}) \\
u_t &= \tanh(x_t W_x^{(u)} + h_{t-1} W_h^{(u)} + b^{(u)}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot u_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}$$

然后就不言而喻了，步骤和上面的RNN几乎一致。

代码块

```python
class LSTM(Layer):
    def __init__(self, hidden_size, in_size=None):
        super().__init__()

        H, I = hidden_size, in_size
        self.x2f = Linear(H, in_size=I)
        self.x2i = Linear(H, in_size=I)
        self.x2o = Linear(H, in_size=I)
        self.x2u = Linear(H, in_size=I)
        self.h2f = Linear(H, in_size=H, nobias=True)
        self.h2i = Linear(H, in_size=H, nobias=True)
        self.h2o = Linear(H, in_size=H, nobias=True)
        self.h2u = Linear(H, in_size=H, nobias=True)
        self.reset_state()

    def reset_state(self):
        self.h = None
        self.c = None

    def forward(self, x):
        if self.h is None:
            f = F.sigmoid(self.x2f(x))
            i = F.sigmoid(self.x2i(x))
            o = F.sigmoid(self.x2o(x))
            u = F.tanh(self.x2u(x))
        else:
            f = F.sigmoid(self.x2f(x) + self.h2f(self.h))
            i = F.sigmoid(self.x2i(x) + self.h2i(self.h))
            o = F.sigmoid(self.x2o(x) + self.h2o(self.h))
            u = F.tanh(self.x2u(x) + self.h2u(self.h))

        if self.c is None:
            c_new = (i * u)
        else:
            c_new = (f * self.c) + (i * u)

        h_new = o * F.tanh(c_new)

        self.h, self.c = h_new, c_new
        return h_new
```

然后这里补充一个东西：

SeqDataLoader

它的目的是像1,500,999这样间隔取数，作为一个batch，这样只有30条？不，这30条是并行的。假如jump是500，那么这个数据将会是30 * 500的，然后在1-30的时间步上并行。这个dataloader仅适用于RNN、LSTM这样的序列。

代码块

```python
class SeqDataLoader(DataLoader):
    def __init__(self, dataset, batch_size, gpu=False):
        super().__init__(dataset=dataset, batch_size=batch_size, shuffle=False,
                         gpu=gpu)

    def __next__(self):
        if self.iteration >= self.max_iter:
            self.reset()
            raise StopIteration

        jump = self.data_size // self.batch_size
        batch_index = [(i * jump + self.iteration) % self.data_size for i in
                       range(self.batch_size)]
        batch = [self.dataset[i] for i in batch_index]

        xp = cuda.cupy if self.gpu else np
        x = xp.array([example[0] for example in batch])
        t = xp.array([example[1] for example in batch])

        self.iteration += 1
        return x, t
```

使用方法（数据集还是那个正弦波）

代码块

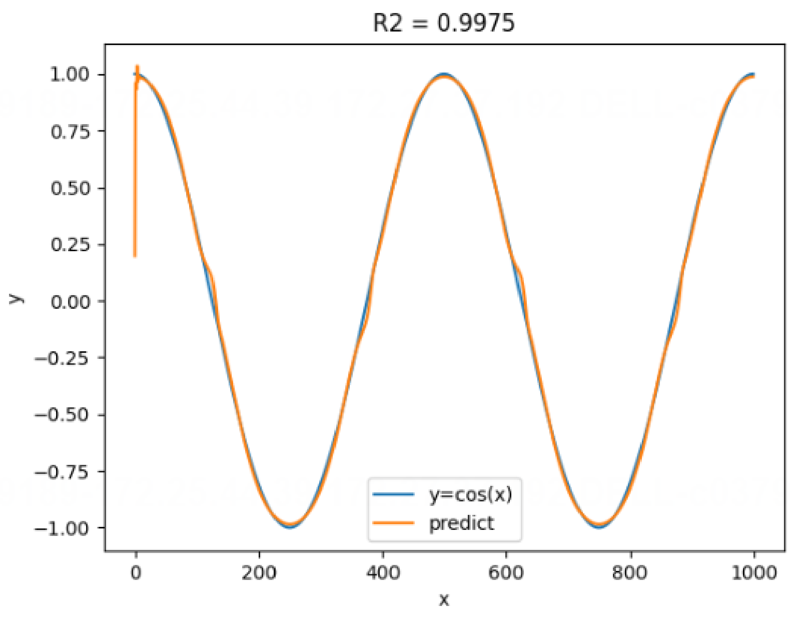
```
if '__file__' in globals():
    import os, sys
    sys.path.append(os.path.join(os.path.dirname(__file__), '..'))
import numpy as np
import matplotlib.pyplot as plt
import dezero
from dezero import Model
from dezero import SeqDataLoader
import dezero.functions as F
import dezero.layers as L


max_epoch = 100
batch_size = 30
hidden_size = 100
bptt_length = 30

train_set = dezero.datasets.SinCurve(train=True)
dataloader = SeqDataLoader(train_set, batch_size=batch_size)
seqlen = len(train_set)


class BetterRNN(Model):
    def __init__(self, hidden_size, out_size):
        super().__init__()
        self.rnn = L.LSTM(hidden_size)
        self.fc = L.Linear(out_size)

    def reset_state(self):
        self.rnn.reset_state()

    def __call__(self, x):
        y = self.rnn(x)
        y = self.fc(y)
        return y

model = BetterRNN(hidden_size, 1)
optimizer = dezero.optimizers.Adam().setup(model)

for epoch in range(max_epoch):
    model.reset_state()
    loss, count = 0, 0

    for x, t in dataloader:
        y = model(x)
        loss += F.mean_squared_error(y, t)
        count += 1

        if count % bptt_length == 0 or count == seqlen:
            model.cleargrads()
            loss.backward()
            loss.unchain_backward()
            optimizer.update()
    avg_loss = float(loss.data) / count
    print('| epoch %d | loss %f' % (epoch + 1, avg_loss))

# Plot
xs = np.cos(np.linspace(0, 4 * np.pi, 1000))
model.reset_state()
pred_list = []

with dezero.no_grad():
    for x in xs:
        x = np.array(x).reshape(1, 1)
        y = model(x)
        pred_list.append(float(y.data))

plt.plot(np.arange(len(xs)), xs, label='y=cos(x)')
plt.plot(np.arange(len(xs)), pred_list, label='predict')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

RNN和LSTM的手撕实验

RNN我们采用自制的sin数据集检验效果，代码如下（rnn.py）：

代码块

```python
import numpy as np
import matplotlib.pyplot as plt
import nailorch
from nailorch import Model
from nailorch import SeqDataLoader
import nailorch.functions as F
import nailorch.layers as L
np.random.seed(42)

max_epoch = 100
batch_size = 30
hidden_size = 100
bptt_length = 30

train_set = nailorch.datasets.SinCurve(train=True)
dataloader = SeqDataLoader(train_set, batch_size=batch_size)
seqlen = len(train_set)


class BetterRNN(Model):
    def __init__(self, hidden_size, out_size):
        super().__init__()
        self.rnn = L.RNN(hidden_size)
        self.fc = L.Linear(out_size)

    def reset_state(self):
        self.rnn.reset_state()

    def __call__(self, x):
        y = self.rnn(x)
        y = self.fc(y)
        return y


model = BetterRNN(hidden_size, 1)
optimizer = nailorch.optimizers.Adam().setup(model)

# ======================
# Training
# ======================
for epoch in range(max_epoch):
    model.reset_state()
    loss, count = 0, 0

    for x, t in dataloader:
        y = model(x)
        loss += F.mean_squared_error(y, t)
        count += 1

        if count % bptt_length == 0 or count == seqlen:
            model.cleargrads()
            loss.backward()
            loss.unchain_backward()
            optimizer.update()

    avg_loss = float(loss.data) / count
    print('| epoch %d | loss %f' % (epoch + 1, avg_loss))


# =====================
# Evaluation + Metrics
# =====================
def mse(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

def rmse(y_true, y_pred):
    return np.sqrt(mse(y_true, y_pred))

def mae(y_true, y_pred):
    return np.mean(np.abs(y_true - y_pred))

def r2_score(y_true, y_pred):
    ss_res = np.sum((y_true - y_pred) ** 2)
    ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
    return 1 - ss_res / ss_tot


xs = np.cos(np.linspace(0, 4 * np.pi, 1000))
model.reset_state()

pred_list = []
true_list = []

with nailorch.no_grad():
    for x in xs:
        x = np.array(x).reshape(1, 1)
        y = model(x)
        pred_list.append(float(y.data))
        true_list.append(float(x))

y_true = np.array(true_list)
y_pred = np.array(pred_list)

print("\nEvaluation Metrics:")
print("MSE :", mse(y_true, y_pred))
print("RMSE:", rmse(y_true, y_pred))
print("MAE :", mae(y_true, y_pred))
print("R2  :", r2_score(y_true, y_pred))


# ======================
# Plot
# ======================
plt.plot(np.arange(len(xs)), y_true, label='y=cos(x)')
plt.plot(np.arange(len(xs)), y_pred, label='predict')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title(f'R2 = {r2_score(y_true, y_pred):.4f}')
plt.show()
```



代码块

```
Evaluation Metrics:
MSE : 0.0012268895240442074
RMSE: 0.035026982799610466
MAE : 0.018463027836480055
R2  : 0.9975486673818644
```

在 SinCurve 数据集上对所构建的 RNN 模型进行训练与评估，模型在预测余弦序列时表现出较高的拟合精度。最终评估结果显示，模型在测试序列上的均方误差（MSE）为 0.00123，均方根误差（RMSE）为 0.0350，平均绝对误差（MAE）为 0.0185，说明预测值与真实值之间的偏差整体较小。从拟合优度指标来看，模型的 R^2 达到 0.9975，接近于 1，表明模型能够解释约 99.75% 的数据方差，成功捕捉了余弦函数的周期性变化特征。结合预测曲线与真实曲线的可视化结果可以观察到，两者在整体趋势和相位上高度一致，仅在局部区域存在轻微误差，说明基于 RNN 并结合截断反向传播（BPTT）的序列建模方法在该时间序列预测任务中具有良好的建模能力和泛化性能。

对于长短期记忆网络（Long Short-Term Memory，LSTM），因其能够有效捕捉序列中的长期依赖关系，被广泛应用于环境与气象数据建模中。这里采用空气污染数据集构建基于 LSTM 的回归预测模型，对污染物浓度的时间演化特性进行建模与分析。

在数据预处理阶段，选取污染物浓度（pollution）作为预测目标，同时引入露点（dew）、气温（temp）、气压（press）、风速（wnd_spd）、降雪（snow）和降雨（rain）等 6 个气象相关变量作为输入特征，共形成 7 维特征向量。数据按照时间顺序划分为训练集和测试集，其中训练集占比为 70%。为保持时序结构，采用序列数据加载方式，并结合截断反向传播（BPTT）策略进行模型训练。

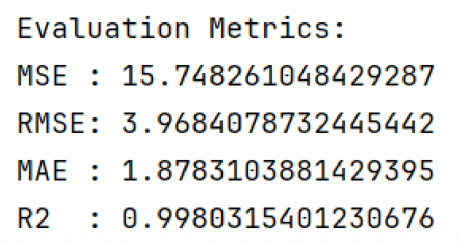
模型结构由一层 LSTM 网络和一层全连接层组成，其中 LSTM 隐藏层维度设为 100，用于提取时间序列中的动态特征；全连接层将隐藏状态映射为单步污染物浓度预测值。模型采用均方误差（MSE）作为损失函数，并使用 Adam 优化算法进行参数更新。在 GPU 环境下进行 3000 个 epoch 的训练，以保证模型充分收敛。

代码：

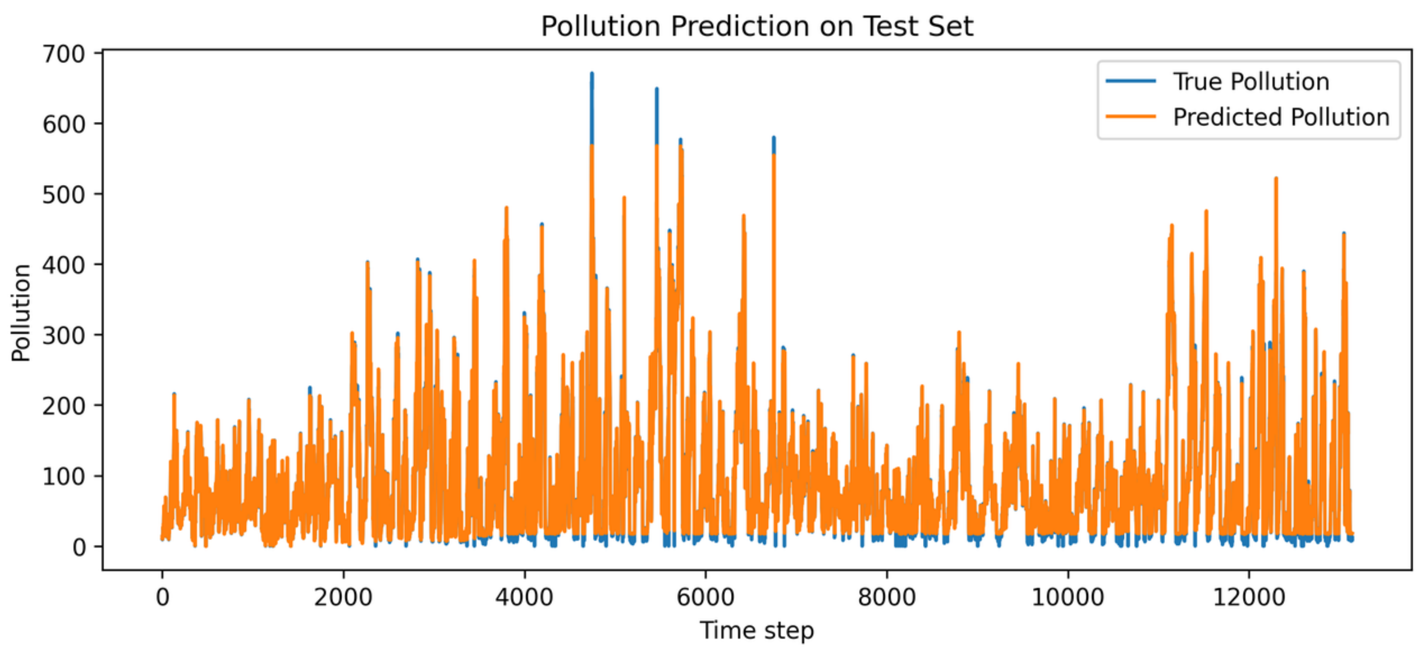
代码块

```python
import numpy as np
import matplotlib.pyplot as plt
import nailorch
from nailorch import Model
from nailorch import SeqDataLoader
import nailorch.functions as F
import nailorch.layers as L
from nailorch.datasets import Dataset
import pandas as pd
import cupy as cp

np.random.seed(42)

max_epoch = 3000
batch_size = 128
hidden_size = 100
bptt_length = 30

class Pollution(Dataset):
    def __init__(self, file_path, train=True, split=0.7):
        super().__init__()
        self.train = train
        data = pd.read_csv(file_path)[
            ["pollution", "dew", "temp", "press", "wnd_spd", "snow", "rain"]
        ].to_numpy().astype(np.float32)

        label = pd.read_csv(file_path)["pollution"].to_numpy().astype(np.float32).reshape(-1, 1)
        num_split = int(split * data.shape[0])
        if self.train:
            self.data = data[:num_split]
            self.label = label[:num_split]
        else:
            self.data = data[num_split:]
            self.label = label[num_split:]

train_set = Pollution(file_path="pollution_train.csv")
dataloader = SeqDataLoader(train_set, batch_size=batch_size, gpu=True)
seqlen = len(train_set)

class BetterRNN(Model):
    def __init__(self, hidden_size, out_size):
        super().__init__()
        self.rnn = L.LSTM(hidden_size, in_size=7)
        self.fc = L.Linear(out_size, in_size=hidden_size)

    def reset_state(self):
        self.rnn.reset_state()

    def __call__(self, x):
        y = self.rnn(x)
        y = self.fc(y)
        return y

model = BetterRNN(hidden_size, 1)
model.to_gpu()
optimizer = nailorch.optimizers.Adam().setup(model)

# ================
# Training
# ================
loss_history = []  # 记录每个 epoch 的平均 loss（基于每个 mini-batch 的标量 loss）
for epoch in range(max_epoch):
    model.reset_state()
    loss, count = 0, 0

    total_loss = 0.0  # 用来累积每个 mini-batch 的标量 loss（用于统计/绘图）
    for x, t in dataloader:
        y = model(x)
        l = F.mean_squared_error(y, t)
        loss += l
        count += 1

        # --- 累积标量用于统计（不影响反传） ---
        total_loss += float(l.data)

        if count % bptt_length == 0 or count == seqlen:
            model.cleargrads()
            loss.backward()
            loss.unchain_backward()
            optimizer.update()
    # 记录并打印（保持你原来的 avg_loss 输出但同时记录更可靠的 avg）
    avg_loss_report = float(loss.data) / count
    epoch_avg_loss = total_loss / count if count > 0 else float('nan')
    loss_history.append(epoch_avg_loss)

    print('| epoch %d | loss_report %f | epoch_avg_loss %f' % (epoch + 1, avg_loss_report, epoch_avg_loss))

model.save_weights("LSTM-3000.npz")

model.load_weights("LSTM-3000.npz")
model.to_gpu()

# ===== 测试集可视化 =====
test_set = Pollution(file_path="pollution_train.csv", train=False)
test_loader = SeqDataLoader(test_set, batch_size=1, gpu=True)

model.reset_state()
preds = []
trues = []

with nailorch.no_grad():
    for x, t in test_loader:
        y = model(x)
        preds.append(float(cp.asnumpy(y.data).ravel()[0]))
        trues.append(float(cp.asnumpy(t).ravel()[0]))

# ========================
# 计算额外评价指标（MSE, RMSE, MAE, R2）
# ========================
def mse(y_true, y_pred):
    y_true = np.array(y_true)
    y_pred = np.array(y_pred)
    return np.mean((y_true - y_pred) ** 2)

def rmse(y_true, y_pred):
    return np.sqrt(mse(y_true, y_pred))

def mae(y_true, y_pred):
    y_true = np.array(y_true)
    y_pred = np.array(y_pred)
    return np.mean(np.abs(y_true - y_pred))

def r2_score(y_true, y_pred):
    y_true = np.array(y_true)
    y_pred = np.array(y_pred)
    ss_res = np.sum((y_true - y_pred) ** 2)
    ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
    return 1 - ss_res / ss_tot

print("\nEvaluation Metrics:")
print("MSE :", mse(trues, preds))
print("RMSE:", rmse(trues, preds))
print("MAE :", mae(trues, preds))
print("R2  :", r2_score(trues, preds))

# ======================
# Plot prediction (原有) 并保存
# ======================
plt.figure(figsize=(10, 4))
plt.plot(trues, label="True Pollution")
plt.plot(preds, label="Predicted Pollution")
plt.xlabel("Time step")
plt.ylabel("Pollution")
plt.legend()
plt.title("Pollution Prediction on Test Set")
plt.savefig("pollution_prediction.png", dpi=300, bbox_inches="tight")
plt.show()

# ======================
# Plot & save training loss 曲线 (每 epoch 的平均 mini-batch loss)
# ======================
plt.figure(figsize=(8, 4))
plt.plot(np.arange(1, len(loss_history) + 1), loss_history, label="Epoch Avg Loss")
plt.xlabel("Epoch")
plt.ylabel("MSE (per-mini-batch average)")
plt.title("Training Loss Curve")
plt.legend()
plt.grid(True)
plt.savefig("training_loss.png", dpi=300, bbox_inches="tight")
plt.show()
```

实验结果：



```
Evaluation Metrics:
MSE : 15.748261048429287
RMSE: 3.9684078732445442
MAE : 1.8783103881429395
R2  : 0.9980315401230676
```



在测试集上的实验结果表明，该 LSTM 模型具有较高的预测精度。评价指标显示，模型在测试集上的 MSE 为 15.75，RMSE 为 3.97，MAE 为 1.88，决定系数 R2 达到 0.998，说明预测结果与真实污染物浓度变化趋势高度一致。预测曲线与真实曲线在整体走势和局部波动上均表现出良好的拟合效果，验证了 LSTM 模型在空气污染时间序列预测任务中的有效性与稳定性。