Learn RNN by Coding

Table of Contents

1. Mathematical Concept for RNN 1
1.1. Task 1
1.2. What is RNN? 1
2. Code Implementation of RNN 2
2.1. Task 2
2.2. Model Implementation 3
2.3. Training 5
3. Interesting Parts 5
3.1. Rolling Prediction 5
3.2. Gradient Explosion and Gradient Vanishing 6

1. Mathematical Concept for RNN

1.1. Task

Given a sequence of data, we want to predict the next value in the sequence. RNN is a type of model that is used to capture the temporal information in the sequence.

Example 1.1 Given a sequence of points: $\{(x_i, \sin(x_i))\}_{i=1}^{N}$, we want to predict the next value in the sequence, especially the next value of $\sin(x_{N+1})$.

Though the sampel seems to be a regression problem, but we could see it as a sin signal data and we want to predict the next value of the signal. Then we need to use RNN to model the distribution of the signal or just find the dependency inside the data and predict the next value.

1.2. What is RNN?

For a RNN, we have two parts: hidden state update and output generation.

For the first part, in RNN, we have many hidden states that contain the information of the last (few) processing timestep. And in each timestep, we would give the input of current timestep and the hidden state of last timestep, and obtain a new hidden state.

Definition 1.2 In RNN, we have hidden states $\{h_t\}_{i=1}^{T}$. For timestep s, our input is x_s and the hidden state of last timestep h_{s-1}, and we would update the hidden state as follows:

$$h_s = f_h(W_i x_s + W_h h_{s-1} + b_h)$$

where f_h is the activation function, usually the tanh function.

By introducing the last timestep's hidden state, we could capture the temporal information of the sequence.

For the second part, output generation, we use the hidden state to generate the output.

Definition 1.3 In RNN, we have output $\{o_t\}_{i=1}^{T}$. For timestep s, our hidden state is h_s, and we would generate the output as follows:

$$o_s = f_o(W_o h_s + b_o)$$

where f_o is the activation function, usually the softmax function, or Sigmoid for binary classification.

1.2.1. Multiple Layers RNN

In the last subsection, we introduce the RNN model with a single hidden layer. In practice, we often want to use multiple hidden layers to capture the complex patterns in the data.

What is the same and what is the different?

- The same: Each timestep, we only look at a slice of the input data and update the whole hidden state.
- The different: In the multi-layer RNN, we update the hidden state of each layer sequentially.

A single layer RNN is just a special case of the multi-layer RNN when the number of layers is 1. And we could actually write the hidden state of single layer RNN as $h_t^{(0)}$

We only need to know one more update rule for the multi-layer RNN:

Definition 1.4

In the multi-layer RNN, we have hidden states $\left\{h_t^{(l)}\right\}_{l=1}^{T}$ at each timestep t. For timestep s, our input is x_s, and we would update the hidden state as follows:

$$h_s^{(1)} = f_h\left(W_h^{(1)} h_{s-1}^{(0)} + W_i^{(1)} x_s + b_h^{(1)}\right)$$

$$h_s^{(l)} = f_h\left(W_h^{(l)} h_{s-1}^{(l-1)} + W_i^{(l)} h_s^{(l-1)} + b_h^{(l)}\right)$$

where f_h is the activation function, usually the tanh function.

It is worth noticing that for each layer, hidden state take the hiddent state of the last timestep as information and the input of current timestep acts like the real input.

2. Code Implementation of RNN

2.1. Task

As mentioned in the previous section, here we use RNN to solve a task of sin signal prediction. We use synthetic data to train the RNN model. The data is generated by the following code:

```python
# script for synthetic data generation
import numpy as np
import torch
```

```

def sine(x):
    return np.sin(x)

def cosine(x):
    return np.cos(x)

# generate a sine wave
def generate_data(num_samples=200, method = None):
    # numerically generate the data
    x = np.linspace(0, 10*np.pi, num_samples)
    y = method(x)

    # add noise to the data
    np.random.seed(42)
    y = y + np.random.normal(0, 0.01, num_samples)

    # format the data into tensor of shape (batch_size, seq_len, input_size)
    y_tensor = torch.from_numpy(y).float().reshape(1, num_samples, 1)

    X_seq = y_tensor[:,:-1,:]
    Y_seq = y_tensor[:,1:,:]

    return X_seq, Y_seq
```

Here `X_seq` is the input sequence and `Y_seq` is the target sequence. **We only do one step prediction.**

2.2. Model Implementation

Since we have the mathematical concept of RNN, we could use `torch.nn.module` to build a RNN from scratch. Here are the core parts.

```python
def __init__(self,input_size, hidden_size, output_size):
        super(MyRNN, self).__init__()

        self.hidden_size = hidden_size

        # input layer
        self.i2h = nn.Linear(input_size, hidden_size)

        # hidden layer
        self.h2h = nn.Linear(hidden_size, hidden_size)
```

```
11
12          # output layer
13          self.h2o = nn.Linear(hidden_size, output_size)
14
15          # activation function
16          self.tanh = nn.Tanh()
```

The initialization part is quite straightforward. For the matrices of parameters to maintain, we have the input-to-hidden, hidden-to-hidden, and hidden-to-output connections. And the activation function is the tanh function.

```python
def forward(self, x):
        """
        Args:
            x (tensor): input tensor, shape: (batch_size, seq_len,
            input_size)

        Returns:
            output (tensor): output tensor, shape: (batch_size,
            output_size)
        """
        batch_size, seq_len, input_size = x.size()

        # initialize hidden state
        hidden = torch.zeros(batch_size, self.hidden_size)

        outputs = []

        for i in range(seq_len):
            x_t = x[:, i, :]
            combined = self.i2h(x_t) + self.h2h(hidden)
            hidden = self.tanh(combined)

            output = self.h2o(hidden)
            outputs.append(output.unsqueeze(1))

        return torch.cat(outputs, dim=1), hidden
```

Simply, we just iterate over the sequence and update the hidden state for each timestep. And we get the output for each timestep. At the end, we cat them by the first dimension, which is the sequence length, to obtain the output tensor. **We also return the last hidden state**, because base on this, we could add more features to the model.

2.3. Training

The training part is quite straightforward. We just need to define the loss function and the optimizer. Here we use the mean squared error loss and the Adam optimizer.

```python
def fit(self, x_train, y_train, epochs=100, lr = 1e-3):
    # define the loss function
    criterion = nn.MSELoss()
    # define the optimizer
    optimizer = optim.Adam(self.parameters(), lr=lr)
    ...
```

`...` stands for the iteration part of the training.

3. Interesting Parts

3.1. Rolling Prediction

For the task above, we only predict the next value in the sequence. That is very much alike to the training process since we have all the true values before the last prediction.

But in practice, we often want to predict the next few values in the sequence. So we need to use the rolling prediction. We need to use the last predicted value as the input of the next prediction. If the model is bad, it would accumulate the error and the prediction would be very bad.

However, we just try to write a code for so-called **rolling prediction**. We maintain two list: `predictions` and `current_input`. We update the `current_input` by appending the last predicted value to the end of the list and pop the first element. And we update the `predictions` by appending the last predicted value to the end of the list.

```python
def rolling_predict(self, initial_input, steps=10):
        self.eval()

        current_input =  initial_input.clone()
        predictions = []

        with torch.no_grad():
            for _ in range(steps):
                # generate predicted sequence throught forward pass
                output, _ = self(current_input)
                last_prediction = output[:,-1,:].unsqueeze(1)

                # append the predicted sequence to the predictions list
                predictions.append(last_prediction)

                # update the current input
```

```
17                  current_input = torch.cat([current_input[:,1:,:],
                    last_prediction], dim=1)
18
19          return torch.cat(predictions, dim=1)
```

3.2. Gradient Explosion and Gradient Vanishing

Since it is very similar to simple neural network, the gradient explosion and gradient vanishing problem is also present in RNN.

The reason for the gradient problem:

Example 3.1 Let's see the operation of BPTT at timestep s, we have the loss function L and we want to update the parameters W_h, which is the weight matrix of the hidden-to-hidden connection.

$$\frac{\partial L}{\partial W_h} = \sum_{t=1}^{s} \frac{\partial L}{\partial h_s} \frac{\partial h_s}{\partial h_t} \frac{\partial h_t}{\partial W_h}$$

The key term is $\frac{\partial h_s}{\partial h_t}$, which is the gradient of the hidden state at timestep s with respect to the hidden state at timestep t. It flows through the timesteps.

$$\frac{\partial h_s}{\partial h_t} = \prod_{k=t+1}^{s} \frac{\partial h_k}{\partial h_{k-1}} = \prod_{k=t+1}^{s} \frac{\partial f_h(W_h h_{k-1} + W_i x_k + b_h)}{\partial h_{k-1}}$$

Let $W_h h_{k-1} + W_i x_k + b_h = z_k$, then we have:

$$\frac{\partial h_s}{\partial h_t} = \prod_{k=t+1}^{s} \frac{\partial f_h}{\partial z_k} \frac{\partial z_k}{\partial h_{k-1}} = \prod_{k=t+1}^{s} [\mathrm{diag}(f_h'(z_k)) \cdot W_h]$$

So if the term in the product is very large, the gradient will explode. If the term is very small, the gradient will vanish. And due to the property of the activation function, the diagonal matrix ranges from 0 to 1. So the gradient will probably vanish.

We could use gradient clipping or normalization to solve the problem. Also, we could use the LSTM to solve the problem.