Chapter 15 Report: Modeling Sequential Data Using Recurrent Neural Networks

Quan Pham

June 23, 2025

Contents

1 Key concepts

1.1 Sequential data

Sequential data, also known as sequence data or sequences, is a type of data that its elements appear in a certain order, with interdependent relationships over time or position. The order of these elements is crucial, as it provide context and structural information that cannot be ignore during analysis. Changing the order can alter or even eliminate the meaning of the entire dataset.

Sequential data can have different lengths, from short sequences to very long ones. These are some typical examples: **Time series:** Daily stock price, sensor data (temperature, humidity), heart rate. **Natural language:** Words in a sentence, sentences in a paragraph. **Audio:** Voice waveforms, music sequences. **Video:** Sequences of frames **Biological sequence:** DNA sequences, protein sequences.

1.2 Categories of sequence modeling

If either the input or output is a sequence, the modeling task likely falls into one of these categories:

- Many-to-one: The input data is a sequence, but the output is a fixed-size vector or scalar, not a sequence.
- One-to-many: The input data is in standard format and not a sequence, but the output is a sequence.
- Many-to-many: Both the input and output arrays are sequences. This category can be further divided based on whether the input and output are synchronized.

1.3 Recurrent Neural Networks

A recurrent neural network (RNN) is any network that contains a cycle within its network connections, meaning that the value of some unit is directly, or indirectly, dependent on its own earlier outputs as an input

Computing activation in an RNN is very similar to standard multilayer peerceptrons and other types of feedforward NNs. For the hidden layer, the net input (preactivation), is computed as follows:

$$\boldsymbol{z}_h^{(t)} = \boldsymbol{W}_{xh}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h$$

Then, the activations of the hidden units at the time step, t, are calculated as follows:

$$\boldsymbol{h}^{(t)} = \sigma_h(\boldsymbol{z}_h^{(t)}) = \sigma_h(\boldsymbol{W}_{xh}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h)$$

Once the activations of the hidden units at the current time step are computed, then the activations of the hidden units at the time step are computed, then the activations of the output units will be computed as follows:

$$\boldsymbol{o}^{(t)} = \sigma_o(\boldsymbol{W}_{ho}\boldsymbol{h}^{(t)} + \boldsymbol{b}_o)$$

1.4 Challenges of learning long-range interactions

Recurrent neural networks (RNNs) are theoretically capable of capturing dependencies across arbitrary sequence lengths. However, in practice, learning long-range interactions is challenging due to issues such as vanishing and exploding gradients during training. When backpropagating errors through many time steps, gradients can shrink exponentially (vanishing) or grow uncontrollably (exploding), making it difficult for the network to learn dependencies that span long intervals.

The vanishing gradient problem causes the network to "forget" information from earlier time steps, limiting its ability to model long-term dependencies. Conversely, exploding gradients can lead to unstable updates and numerical issues.

2 Algorithms and Models Studied

2.1 Converting raw text into sequence data

- remove HTML markups, punctuation and other non-letter characters
- extract emoticons
- split text into words
- encoding each unique token into intergers
- padding sequences for each batch

2.2 Backpropagation Through Time

$$L = \sum_{t=1}^{T} L^{(t)}$$

$$\frac{\partial L^{(t)}}{\partial \boldsymbol{W}_{hh}} = \frac{\partial L^{(t)}}{\partial \boldsymbol{o}^{(t)}} \times \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \times (\sum_{k=1}^{t} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} \times \frac{\partial \boldsymbol{h}^{(k)}}{\partial \boldsymbol{W}_{hh}})$$

$$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t)}} = \prod_{i=k+1}^{t} \frac{\partial \boldsymbol{h}^{(i)}}{\partial \boldsymbol{h}^{(i-1)}}$$

2.3 Truncated Backpropagation Through Time

Using gradient clipping, we specify a cut-off or threshold value for the gradients, and we assign this cut-off value to gradient values that exceed this value. In contrast, TBPTT simply limits the number of time steps that the signal can backpropagate after each forward pass. For example, even if the sequence has 100 elements or steps, we may only backpropagate the most recent 20 time steps.

2.4 Long-short Term Memory

LSTMs resemble standard recurrent neural networks but here each ordinary recurrent node is replaced by a memory cell. Each memory cell contains an internal state, i.e., a node with a self-connected recurrent edge of fixed weight 1, ensuring that the gradient can pass across many time steps without vanishing or exploding.

2.5 Gated Recurrent Units

GRUs have a simpler architecture than LSTMs; therefore, they are computationally more efficient, while their performance in some tasks, such as polyphonic music modeling, is comparable to LSTMs.

2.6 Bidirectional RNNs

A Bidirectional Recurrent Neural Network (BRNN) is a type of recurrent neural network (RNN) that processes sequential data by analyzing it in both forward and backward directions. Unlike standard, or unidirectional, RNNs which process sequences chronologically (from past to future), a BRNN uses two separate hidden layers:

- Forward Layer: Processes the input sequence from start to end.
- Backward Layer: Processes the input sequence from end to start (in reverse).

The outputs from both of these layers are then combined (e.g., by concatenation, summation, or averaging) at each time step to produce the final output. This dual-directional approach allows the model to capture context from both past and future elements in the sequence, providing a more comprehensive understanding.

3 Follow tutorial code in the book

3.1 Project one - predicting the sentiment of IMDb reviews

3.1.1 IMDb reviews dataset

The tutorial code loads the IMDb reviews datset using `torchtext` package. However, `torchtext` uses `torchdata` to load data, which doesn't support new python versions. I have to download the dataset from hugging face using pandas.readparquet().

```
splits = {'train': 'plain_text/train-00000-of-00001.parquet', 'test':
↪  'plain_text/test-00000-of-00001.parquet', 'unsupervised':
↪  'plain_text/unsupervised-00000-of-00001.parquet'}
df = pd.read_parquet("hf://datasets/stanfordnlp/imdb/" + splits["train"])

test_dataset = pd.read_parquet("hf://datasets/stanfordnlp/imdb/" +
↪  splits["test"])
```

Split train dataset to train set and validation set.

```
from sklearn.model_selection import train_test_split
train_dataset, valid_dataset = train_test_split(df, train_size=20000,
 ↪  test_size=5000, random_state=1)
```

3.1.2 Processing the text data

Tokenization

```
token_counts = Counter()

def tokenizer(text):
    text = re.sub(r'<[^>]*>', '', text)
    emoticons = re.findall(r'(?::|;|=)(?:-)?(?:\)|\(|D|P)', text.lower())
    text = re.sub(r'[\W]+', ' ', text.lower()) +\
        ' '.join(emoticons).replace('-', '')
    tokenized = text.split()
    return tokenized

for i, (text, label) in train_dataset.iterrows():
    tokens = tokenizer(text)
    token_counts.update(tokens)

print('Vocab-size:', len(token_counts))
```

```
Vocab-size: 69353
```

Vocab

```
from torchtext.vocab import vocab
sorted_by_freq_tuples = sorted(token_counts.items(), key=lambda x: x[1],
 ↪  reverse=True)
ordered_dict = OrderedDict(sorted_by_freq_tuples)
vocab = vocab(ordered_dict)
vocab.insert_token("<pad>", 0)
vocab.insert_token("<unk>", 1)
vocab.set_default_index(1)

print([vocab[token] for token in ['this', 'is', 'an', 'example']])
```

collate batch function.

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

def collate_batch(batch):
```

```
    label_list, text_list, lengths = [], [], []
    for _text, _label in batch:
        label_list.append(label_pipeine(_label))
        processed_text = torch.tensor(text_pipeline(_text),
        ↪  dtype=torch.int64)
        text_list.append(processed_text)
        lengths.append(processed_text.size(0))
    label_list = torch.tensor(label_list)
    lengths = torch.tensor(lengths)
    padded_text_list = nn.utils.rnn.pad_sequence(text_list,
    ↪  batch_first=True)
    return padded_text_list.to(device), label_list.to(device),
    ↪  lengths.to(device)
```

3.1.3 Loading data

Define TabularDataset class, which can return item from dataframe.

```
class TabularDataset(Dataset):
    def __init__(self, dataframe):
        self.dataframe = dataframe

    def __len__(self):
        return len(self.dataframe)

    def __getitem__(self, idx):
        return self.dataframe.iloc[idx]['text'],
        ↪  self.dataframe.iloc[idx]['label']
```

prepare data loaders.

```
batch_size = 32
train_set = TabularDataset(train_dataset)
valid_set = TabularDataset(valid_dataset)
test_set = TabularDataset(test_dataset)

train_dl = DataLoader(train_set, batch_size=batch_size, shuffle=True,
↪  collate_fn=collate_batch)
valid_dl = DataLoader(valid_set, batch_size=batch_size, shuffle=False,
↪  collate_fn=collate_batch)
test_dl = DataLoader(test_set, batch_size=batch_size, shuffle=False,
↪  collate_fn=collate_batch)
```

3.1.4 Train and evaluate model

```
def train(dataloader, optimizer, loss_fn):
    model.train()
    total_acc, total_loss = 0, 0
    for text_batch, label_batch, lengths in tqdm(dataloader):
        optimizer.zero_grad()
        pred = model(text_batch, lengths)[:, 0]
        loss = loss_fn(pred, label_batch)
        loss.backward()
        optimizer.step()
        total_acc += ((pred >= 0.5).float() ==
        ↪  label_batch).float().sum().item()
        total_loss += loss.item() * label_batch.size(0)

    return total_acc / len(dataloader.dataset), total_loss /
    ↪  len(dataloader.dataset)

def evaluate(dataloader, loss_fn):
    model.eval()
    total_acc, total_loss = 0, 0

    with torch.no_grad():
        for text_batch, label_batch, lengths in tqdm(dataloader):
            pred = model(text_batch, lengths)[:, 0]
            loss = loss_fn(pred, label_batch)
            total_acc += ((pred >= 0.5).float() ==
            ↪  label_batch).float().sum().item()
            total_loss += loss.item() * label_batch.size(0)

    return total_acc / len(dataloader.dataset), total_loss /
    ↪  len(dataloader.dataset)
```

```
loss_fn = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

num_epochs = 10
torch.manual_seed(1)
print(f'Train model on device {torch.cuda.get_device_name(device)}')
for epoch in range(num_epochs):
    print(f'Epoch {epoch+1}:')
    acc_train, loss_train = train(train_dl, optimizer, loss_fn)
    acc_valid, loss_valid = evaluate(valid_dl, loss_fn)
    print(f'accuracy: {acc_train:.4f} val_accuracy: {acc_valid:.4f}')
```

```
accuracy: 0.9488 val_accuracy: 0.8466
```

```
acc_test, _ = evaluate(test_dl, loss_fn)
print(f'test_accuracy: {acc_test:.4f}')
```

```
test_accuracy: 0.8515
```

3.2 Project two - character-level language modeling with Pytorch

```
vocab_size = len(char_array)
embed_dim = 256
rnn_hidden_size = 512
torch.manual_seed(1)
model = RNN(vocab_size, embed_dim, rnn_hidden_size).to(device)
model
```

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

num_epochs = 10000
torch.manual_seed(1)

for epoch in range(num_epochs):
    hidden, cell = model.init_hidden(batch_size)
    seq_batch, target_batch = next(iter(seq_dl))
    seq_batch = seq_batch.to(device)
    target_batch = target_batch.to(device)
    optimizer.zero_grad()
    loss = 0
    for c in range(seq_length):
        pred, hidden, cell = model(seq_batch[:, c], hidden, cell)
        loss += loss_fn(pred, target_batch[:, c])
    loss.backward()
    optimizer.step()
    loss = loss.item()/seq_length
    if epoch % 500 == 0:
        print(f'Epoch {epoch} loss: {loss:.4f}')
```

```
def sample(model, starting_str, len_generated_text=500, scale_factor=1.0):
    encoded_input = torch.tensor([char2int[s] for s in starting_str])
    encoded_input = torch.reshape(encoded_input, (1, -1)).to(device)
    generated_str = starting_str

    model.eval()
```

```
    hidden, cell = model.init_hidden(1)
    for c in range(len(starting_str)-1):
        _, hidden, cell = model(encoded_input[:, c].view(1), hidden, cell)

    last_char = encoded_input[:, -1]
    for i in range(len_generated_text):
        logits, hidden, cell = model(last_char.view(1), hidden, cell)
        logits = torch.squeeze(logits, 0)
        scaled_logits = logits * scale_factor
        m = Categorical(logits=scaled_logits)
        last_char = m.sample()
        generated_str += str(char_array[last_char])

    return generated_str
```

```
torch.manual_seed(4)
print(sample(model, starting_str='But what was', scale_factor=2.0))
```

Text generated by the model:
"""
But what was a complete sure of the mystery which had been so estabiquely. Pencroft, with intending to hoist emotion to all.

Towards six, Cyrus Harding, “whered are the precious verous time to Cyrus Harding’s inhabited; there even intended to survey the cost of the corral.

The six hard later stopped, and then returned to the bay, which were watercourses of our new feet nor their feasts were gazing at the southern part of the ladder–a right by the fine season whose branches was habitable to remain power
"""

4 Pratical exploration with RNNs

4.1 Custom dataset class

Define MysteriousIsland class (utils/data.py):

```
class MysteriousIsland(Dataset):
    def __init__(self, path, chunk_size, start=None, end=None,
    ↪ keep_text=True, keep_text_encoded=True):
        super().__init__()
        with open(path, 'r', encoding='utf8') as fp:
            text = fp.read()

        if start and end:
            start_idx = text.find(start)
```

```
            end_idx = text.find(end)
            text = text[start_idx:end_idx]

        chars = sorted(set(text))
        self.char2int = {ch:i for i, ch in enumerate(chars)}
        self.int2char = np.array(chars)

        text_encoded = np.array([self.char2int[ch] for ch in text])
        self.chunks = torch.tensor(np.array([text_encoded[i:i+chunk_size]
        ↪  for i in range(len(text_encoded) - chunk_size)]))

        if keep_text:
            self.text = text

        if keep_text_encoded:
            self.text_encoded = text_encoded

    def __len__(self):
        return len(self.chunks)

    def __getitem__(self, idx):
        chunk = self.chunks[idx]
        return chunk[:-1].long(), chunk[1:].long()

    def get_encoder_decoder(self):
        encoder = lambda text: np.array([self.char2int[ch] for ch in
        ↪  text])
        decoder = lambda seq: ''.join(self.int2char[seq])
        return encoder, decoder
```

And here is how to use:

```
start = 'THE MYSTERIOUS ISLAND'
end = '\n\n*** END OF THE PROJECT GUTENBERG'
mys_dataset = utils.data.MysteriousIsland('../data/1268-0.txt', 41, start,
↪  end)
```

4.2 Split dataset function

This is the function I write (utils/data.py):

```
def split_dataset(dataset, train_size=None, test_size=None,
↪  random_state=None):
    ...
```

```
    indices = torch.randperm(len(dataset))
    train_indices = indices[:n_train]
    test_indices = indices[n_train:n_train+n_test]
    return Subset(dataset, train_indices), Subset(dataset, test_indices)
```

4.3 Adjust hyperparameter to explore new models

```
vocab_size = 79
model_v1 = utils.model.RNN_v1(vocab_size)
model_v1.load_state_dict(torch.load('../models/gen-v1.pth'))
model_v1 = model_v1.to(device)
model_v1
```

```
RNN_v1(
  (embedding): Embedding(79, 256)
  (rnn): LSTM(256, 512, batch_first=True)
  (fc): Linear(in_features=512, out_features=79, bias=True)
)
```

```
# utils.model.py
class RNN_v2(nn.Module):
    def __init__(self, vocab_size, embed_dim=128, rnn_hidden_size=512,
    ↪  num_layers=1):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.rnn_hidden_size = rnn_hidden_size
        self.rnn = nn.LSTM(embed_dim, rnn_hidden_size,
        ↪  num_layers=num_layers, bidirectional=True, batch_first=True)
        self.layer_norm = nn.LayerNorm(2*rnn_hidden_size)
        self.fc = nn.Linear(2*rnn_hidden_size, vocab_size)

    def forward(self, x, hidden, cell):
        out = self.embedding(x).unsqueeze(1)
        out, (hidden, cell) = self.rnn(out, (hidden, cell))
        out = self.layer_norm(out)
        out = self.fc(out).reshape(out.size(0), -1)
        return out, hidden, cell

    def init_hidden(self, batch_size):
        hidden = torch.zeros(2, batch_size, self.rnn_hidden_size)
        cell = torch.zeros(2, batch_size, self.rnn_hidden_size)
        return hidden, cell
```

```
model_v2 = utils.model.RNN_v2(vocab_size)
model_v2.load_state_dict(torch.load('../models/gen-v2.2.pth'))
```

```
model_v2 = model_v2.to(device)
model_v2
```

```
RNN_v2(
  (embedding): Embedding(79, 128)
  (rnn): LSTM(128, 512, batch_first=True, bidirectional=True)
  (layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
  (fc): Linear(in_features=1024, out_features=79, bias=True)
)
```

```
print(sample(model_v1, "the mysterious", 500, 1.0))
```

This is the output:

the mysterious voice,-pity of spine from the sails. It shoulders. But the enemys of the enormous acid and twenty feet from the return to the brig, delighted ear to accompanied ourselves in a few works narrowed towards him a sort of flames were scarcely eaten, noticed the lexter on an insuffactured. Thus the stranger might happen a pale of step thus then, to life, at this place in the unfortunate man who does not so ching towards the best felt. He strongly attracted fifty degrees us. I could be a little band,

```
print(sample(model_v2, "the mysterious", 500, 1.0))
```

This is the output:

the mysterious places! Why should one!"

Cyrus Harding then thought of took the question of four" rich, he does not careful to penetrate intourable condition. He expected to come caway or that of publicious. Besides, Top arrived at the struggle and crew of eggs.

"Ah!y like a wild with pyrites presented no marker.

Presently, the settlers immediately ran began by astonished great results. After having done no less exciting as he was accustomed to return, and arrived at the tenarts of Granite House.

It was a

5 Learnings and Challenges

5.1 Key Takeaways

- Properties of sequences that make them different from other types of data such as structured data or images.
- Foundations of RNNs for sequence modeling: how a basic RNN model works and its limitations with regard to capturing long-term dependencies in sequence data.
- LSTM cells consist of a gating mechanism to reduce the effect of exploding and vanishing gradient problems, which are common in basic RNN models.
- implemnt RNNs with Pytorch.

5.2 Challenges Encountered

- It's difficult to find a unified definition for each concept.
- I fell into the perfectionism trap.
- Writing report is an open-ended task.

5.3 Opening Questions

- embedding in character level models?
- what after RNNs?