Character-Level Recurrent Neural Networks

March 11, 2025

1 Introduction

A **Recurrent Neural Network (RNN)** is a type of neural network designed for sequential data. Unlike traditional feedforward networks, RNNs maintain a hidden state that allows them to process inputs over time. Here we will analyze an RNN implemented in NumPy and compare it with a PyTorch version.

2 Code Analysis

Analyze the NumPy implementation of the RNN and answer the following questions:

2.1 Understanding the Model

- What does each matrix (W_{xh}, W_{hh}, W_{hy}) represent?
- Why do we use the `tanh` activation function in the hidden state update?
- How is the hidden state initialized, and why?

2.2 Training Process

- What loss function is used, and why?
- How does the model update its weights?
- What is the purpose of gradient clipping (`np.clip`)?

2.3 Text Generation

- How does the model generate text?
- How does sampling work in the `sample` function?
- What happens if you modify the sampling temperature?

3 Modifications and Experiments

Task 1: Add Dropout Modify the PyTorch implementation to include dropout. Try different dropout rates and compare the results. What is dropout and why would we use it? (see `https://towardsdatascience.com/dropout-in-neural-networks-47a162d621d9/` or Deep Learning textbooks)

Task 2: Change Activation Function

Modify the activation function in the hidden state update. Compare `tanh`, `ReLU`, and `sigmoid`. How does that impact performance? Try this again after Task 3.

Task 3: Implement an LSTM Replace the vanilla RNN with an LSTM in PyTorch. What changes in terms of output and performance?

Task 4: Experiment with Hyperparameters Modify:

- Number of hidden units
- Sequence length
- Learning rate

Report how these changes affect training loss and generated text.

Task 5: Pick a different input text, see `https://www.gutenberg.org/` Enjoy playing with different languages or writing styles and seeing how the model performs. Report what differences you observe.

Task 6: LSTMs in your research area Find a journal article in a field of interest to use that uses an LSTM model or a GRU model (they probably didn't use a vanilla RNN). Summarize the article (give the reference and DOI), why they used this type of model, and what they learned from it. This can be any variation, e.g. it can use both CNN and LSTM, and so on.

4 Submission

Each student should submit:

- Written answers to the analysis questions.
- Modified code with their experiments.
- A brief report (1-2 pages) summarizing their findings.

5 Resources

- Andrej Karpathy's blog post on RNNs: `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`
- PyTorch documentation: `https://pytorch.org/docs/stable/nn.html`
- Text sources: `https://www.gutenberg.org/`

A Appendix: Code for Dropout and LSTM

A.1 Adding Dropout in PyTorch

Dropout is a technique to prevent overfitting by randomly setting some activations to zero during training. The following code modifies the PyTorch RNN to include dropout:

Listing 1: Adding Dropout to PyTorch RNN

```
import torch.nn as nn

class CharRNN(nn.Module):
    def __init__(self, vocab_size, hidden_size, dropout_rate=0.3):
        super(CharRNN, self).__init__()
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(vocab_size, hidden_size, batch_first=True)
        self.dropout = nn.Dropout(dropout_rate)  # Dropout layer
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, hidden):
        out, hidden = self.rnn(x, hidden)
        out = self.dropout(out)  # Apply dropout after RNN output
        out = self.fc(out)
        return out, hidden
```

Students should experiment with different dropout rates (e.g., 0.1, 0.5) and observe the effect on training.

A.2 Converting the RNN to an LSTM

To modify the PyTorch model to use an LSTM instead of an RNN, replace the `nn.RNN` layer with `nn.LSTM`. Additionally, update the hidden state handling, since LSTMs maintain both a hidden state and a cell state:

Listing 2: Changing PyTorch RNN to LSTM

```
class CharLSTM(nn.Module):
    def __init__(self, vocab_size, hidden_size, num_layers=1, dropout_rate=0.3):
        super(CharLSTM, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
```

```
        self.lstm = nn.LSTM(vocab_size, hidden_size, num_layers, \
            batch_first=True, dropout=dropout_rate)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, hidden):
        out, hidden = self.lstm(x, hidden)  # LSTM replaces RNN
        out = self.fc(out)
        return out, hidden

    def init_hidden(self, batch_size):
        return (torch.zeros(self.num_layers, batch_size, self.hidden_size),
                torch.zeros(self.num_layers, batch_size, self.hidden_size))
```

A.3 Updating the Training Loop for LSTM

The training loop needs to be updated to handle the new hidden state structure:

Listing 3: Modifying Training Loop for LSTM

```
model = CharLSTM(vocab_size, hidden_size, num_layers=2, dropout_rate=0.3)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
criterion = nn.CrossEntropyLoss()

for epoch in range(num_epochs):
    model.train()
    hidden = model.init_hidden(batch_size=1)  # Initialize LSTM hidden state

    for i in range(0, len(data) - seq_length, seq_length):
        inputs = encode_text(data[i:i+seq_length])
        targets = encode_text(data[i+1:i+seq_length+1])

        input_tensor = torch.zeros(1, seq_length, vocab_size)
        target_tensor = torch.tensor(targets, dtype=torch.long).unsqueeze(0)

        for t, char in enumerate(inputs):
            input_tensor[0, t, char] = 1

        optimizer.zero_grad()
        hidden = tuple(h.detach() for h in hidden)
        output, hidden = model(input_tensor, hidden)
        loss = criterion(output.view(-1, vocab_size), target_tensor.view(-1))
        loss.backward()
        optimizer.step()

    if epoch % 100 == 0:
        print(f'Epoch {epoch}, Loss: {loss.item()}')
```