Minor in AI

RNN & Deep RNN

A Practical Implementation

March 20, 2025

1 Introduction

Recurrent Neural Networks (RNNs) are powerful for sequential data processing, such as time series and natural language processing (NLP). This document explores:

- Implementing a simple RNN forward pass using NumPy.
- Understanding hidden state updates.
- Implementing a deep RNN with PyTorch.
- Tokenizing text and creating input sequences for training.

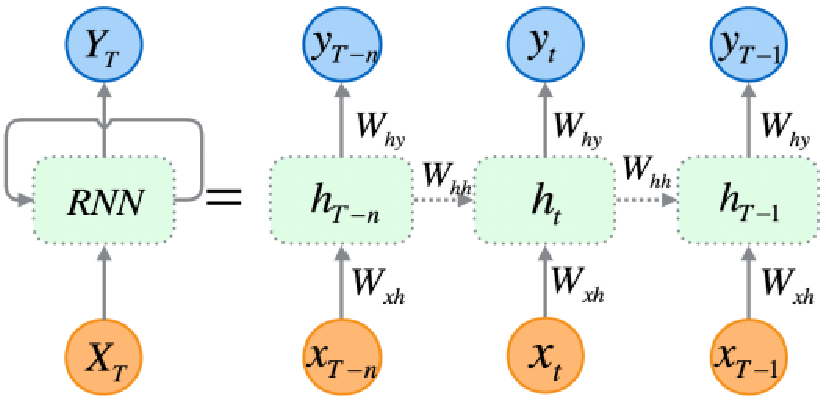

Figure 1: RNN PC : ResearchGate

2 Implementing an RNN Forward Pass in NumPy

First, we define an RNN forward pass, where we compute the next hidden state and the output prediction at each time step.

2.1 Initializing Data and Parameters

```python
import numpy as np

np.random.seed(182)

# Input data: shape (5000, 5, 10) -> (features, batch size, time steps)
x_t = np.random.randn(5000, 5, 10)
# Initial hidden state: shape (231,5) -> (hidden size, batch size)
a_prev = np.random.randn(231, 5)

# Initializing parameters
parameters = {}
parameters["w1"] = np.random.randn(231,5000)  # Input to hidden weights
parameters["w2"] = np.random.randn(231,231)  # Hidden to hidden weights
parameters["b1"] = np.random.randn(231,1)     # Bias for hidden state
parameters["b2"] = np.random.randn(5000,1)    # Bias for output
parameters["w3"] = np.random.randn(5000,231) # Hidden to output weights
```

2.2 Defining Softmax Function

Softmax is used for output activation in classification tasks.

```python
def softmax(x):
    return np.exp(x) / np.sum(np.exp(x), axis=0)
```

2.3 Implementing the RNN Cell Forward Pass

An RNN cell computes the next hidden state and the output prediction at a single time step.

```python
def rnn_cell_forward(xt, a_prev, parameters):
    w1, w2, b1, b2, w3 = parameters["w1"], parameters["w2"], parameters[
    "b1"], parameters["b2"], parameters["w3"]

    a_next = np.tanh(np.dot(w1, xt) + np.dot(w2, a_prev) + b1)
    yt_pred = softmax(np.dot(w3, a_next) + b2)

    return a_next, yt_pred
```

2.4 Running Forward Pass for All Time Steps

We iterate over all time steps to compute the hidden states and predictions.

```python
def run_forward(x, a0, parameters):
    n_x, m, t_x = x.shape  # Extract dimensions
    n_y, n_a = parameters["w3"].shape

    a = np.zeros((n_a, m, t_x))
    y_pred = np.zeros((n_y, m, t_x))
    a_next = a0

    for i in range(t_x):
        xt = x[:, :, i]
        a_next, yt_pred = rnn_cell_forward(xt, a_next, parameters)

    return a_next, y_pred

# Running forward pass
a_tmp, y_pred = run_forward(x_t, a_prev, parameters)
```

```
(5000, 5)
(231, 5)
(5000, 5)
(231, 5)
(5000, 5)
(231, 5)
(5000, 5)
(231, 5)
(5000, 5)
(231, 5)
(5000, 5)
(231, 5)
```

```
(5000, 5)
(231, 5)
(5000, 5)
(231, 5)
(5000, 5)
(231, 5)
(5000, 5)
(231, 5)
```

3 Building a Deep RNN Using PyTorch

Now, we implement a multi-layer RNN using PyTorch.

3.1 Reading and Preprocessing Text

We use `War and Peace` as input data, preprocessing it into tokens.

```python
import torch
import torch.nn as nn
import re
from collections import Counter

file_path = "War_and_Peace.txt"
with open(file_path, "r", encoding="utf-8") as f:
    lines = f.read()

print("sample text ->",lines[:100])
```

```
sample text -> The Project Gutenberg eBook of War and Peace

This ebook is for the use of anyone anywhere in th
```

```python
# Preprocessing text
def pre_process_data(text):
    text = text.lower()
    text = re.sub(r"[^a-z\s]", "", text)
    return text.split()

tokens = pre_process_data(lines)
print("sample Tokens ->",tokens[:10])
```

```
sample Tokens -> ['the', 'project', 'gutenberg', 'ebook', 'of', 'war',
'and', 'peace', 'this', 'ebook']
```

```python
word_count = Counter(tokens)
vocab = sorted(word_count.keys())
word_to_index = {word: i for i, word in enumerate(vocab)}
index_to_word = {i: word for i, word in enumerate(vocab)}
```

```
sample word to index -> [('a', 0), ('aah', 1), ('ab', 2), ('aback', 3),
('abacus', 4), ('abandon', 5), ('abandoned', 6), ('abandoning', 7),
('abandonment', 8), ('abandons', 9)]
```

```python
for i in range(10): # First 10 words of vocab
  word = vocab[i]

  # Printing the corresponding words and their counts
  print(word,word_count[word])
```

```
a 10494
aah 1
ab 1
aback 3
abacus 1
abandon 25
abandoned 54
abandoning 26
abandonment 14
abandons 1
```

3.2 Creating Training Sequences

We define input sequences of length 5 and their corresponding targets.

```python
seq_length = 5

def create_sequence(tokens, seq_length):
    inputs, targets = [], []
    for i in range(len(tokens) - seq_length):
        seq_in = tokens[i:i + seq_length]
        seq_out = tokens[i + seq_length]
        inputs.append([word_to_index[word] for word in seq_in])
        targets.append(word_to_index[seq_out])
    return inputs, targets

inputs, targets = create_sequence(tokens, 5)

print(inputs[0])
print(targets[0])
```

```
[17809, 13774, 7973, 5554, 12207]
19543
```

3.3 Defining the Deep RNN Model

A Deep Recurrent Neural Network (Deep RNN) is a type of neural network designed to handle sequential data by stacking multiple RNN layers. This increases the network's ability to learn complex temporal patterns compared to a single-layer RNN.

Deep RNNs can capture long-term dependencies better and are widely used in tasks such as:

- Natural Language Processing (NLP)
- Time Series Forecasting
- Speech Recognition

- Text Generation

The architecture consists of multiple recurrent layers stacked on top of each other. Each layer processes sequential data and passes it to the next layer. The final layer produces an output, typically a probability distribution over a vocabulary in NLP tasks.

```python
# Define the Deep RNN class
class DeepRnn(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers):
        super(DeepRnn, self).__init__()
        # Embedding layer to convert words into vector representations
        self.embeddings = nn.Embedding(vocab_size, embed_dim)
        # Multi-layer RNN
        self.rnn = nn.RNN(embed_dim, hidden_dim, num_layers, batch_first
=True)
        # Fully connected layer to map hidden states to vocabulary size
        self.fc = nn.Linear(hidden_dim, vocab_size)

    def forward(self, x):
        # Convert word indices into dense vectors
        embeddings = self.embeddings(x)
        # Pass embeddings through RNN layers
        output, hidden = self.rnn(embeddings)
        # Final output layer considers only the last time step output
        output = self.fc(output[:, -1, :])
        return output
```

The above implementation initializes a deep RNN with:

- An embedding layer to map words to dense vectors.
- A multi-layer RNN to capture temporal dependencies.
- A fully connected output layer that predicts the next word.

This model will be trained on a dataset such as 'War and Peace', preprocessed into sequences of words. The training phase involves optimizing the network using gradient descent.

4 Key Takeaways

1. **Deep RNNs improve sequential learning:** Stacking multiple RNN layers enhances the model's ability to capture complex patterns in sequential data.
2. **Embedding layers help with word representation:** Words are mapped to dense vectors before being processed by the RNN layers.
3. **Multi-layer RNNs capture long-term dependencies:** The deep architecture allows better learning of relationships over extended sequences.
4. **Training involves optimizing through backpropagation:** Gradient descent updates weights to minimize prediction errors over time.
5. **Deep RNNs have applications in NLP and beyond:** They are widely used in text generation, speech recognition, and time series forecasting.

Google Collab Link: Code Here!!!