```
#####################################################################
#                                   #
# CS435 Generative AI: Security, Ethics and Governance      #
#                                   #
# Instructor: Dr. Adnan Masood                  #
# Contact:   adnanmasood@gmail.com                  #
#                                   #
# Notebook is MIT Licensed                  #
#####################################################################
```

Recurrent Neural Networks (RNNs)

Welcome to this comprehensive notebook on **Recurrent Neural Networks (RNNs)**. We will explore RNNs from multiple perspectives (five distinct levels) and build an intuition of how they work, their history, the math behind them, and how to code a simple RNN from scratch.

Table of Contents

---

1. Building an Intuitive Understanding

Think of RNNs like remembering what happened in the previous sentence when you read a book. They use their "memory" to keep track of information from past steps to understand what should come next.

RNNs are special neural networks that process information one step at a time and remember what they have seen before. This memory lets them handle tasks like predicting the next word in a sentence or classifying text.

RNNs process sequences of data (e.g., text, audio, or time series) by maintaining a hidden state that updates after each new data point. This hidden state is what allows them to keep information about previous elements in the sequence.

RNNs implement a feedback loop in their architecture. At each time step, an RNN cell takes the current input and the hidden state from the previous time step to produce a new hidden state. However, training vanilla RNNs can lead to issues like vanishing or exploding gradients.

More complex variants, such as LSTM (Long Short-Term Memory) networks and GRU (Gated Recurrent Units), address vanishing and exploding gradients by introducing gating mechanisms. These gates control the flow of information to and from the hidden state, enabling the network to "forget" or "remember" selectively.

2. Intuition

If you are reading a sentence in English, the words that came before might change how you interpret the next word. That is exactly the idea of RNNs – they remember what came before. They use a hidden state (like a short-term memory) to keep track of previous inputs and use this memory to help predict the next output.

3. Brief History

RNNs have been around for decades. Early foundational work was done by **John Hopfield** in the 1980s with Hopfield networks. In 1989, **Williams and Zipser** studied gradient-based learning methods for recurrent networks. **Sepp Hochreiter** and **Jürgen Schmidhuber** introduced the

LSTM (Long Short-Term Memory) in 1997 to address the vanishing gradient problem. Later modifications like **GRUs** (by Kyunghyun Cho and others) further optimized the gating mechanisms.

4. Underlying Technology

- **Hidden State**: Each time step's memory
- **Weights**: Matrices that transform input and hidden state
- **Activation Function**: Often a $\tanh$ or ReLU
- **Loss Function**: Determines the error
- **Backpropagation Through Time (BPTT)**: Specialized training method that unrolls the RNN across time steps

RNNs differ from feed-forward networks by introducing feedback loops. This allows them to operate on sequences of varying lengths and maintain a form of memory.

5. Math Behind RNNs

Hidden State Update:

$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

where:

- h_t is the hidden state at time t
- x_t is the input at time t
- W_{xh} and W_{hh} are weight matrices
- b_h is the bias term
- σ is an activation function (often $\tanh$ or ReLU)

Output:

$$y_t = W_{hy}h_t + b_y$$

where:

- y_t is the output at time t
- W_{hy} is the weight matrix for the hidden-to-output transformation
- b_y is the output bias

The key point is that h_t depends on h_{t-1}, so the network can remember information from previous time steps.

6. Example with Intuitive Code

Imagine we have a sequence of letters `HELLO`, and we want to predict the next letter in the sequence. We'll do a simple RNN approach:

1. Convert letters to numbers (e.g., H=0, E=1, L=2, O=3).
2. Pass them through an RNN.
3. Predict the next letter.

We'll use PyTorch to illustrate, but the idea is the same in other frameworks.

Here's a small code snippet to show a tiny RNN in action (no training, just forward pass):

```
import torch
import torch.nn as nn

# Let's define a simple RNN module.
class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleRNN, self).__init__()
        self.hidden_size = hidden_size

        # Weight matrices
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        hidden = torch.tanh(hidden)
        output = self.i2o(combined)
```

```
        output = self.softmax(output)
        return output, hidden

    def init_hidden(self):
        return torch.zeros(1, self.hidden_size)

# Sample usage
input_size = 4  # Suppose we have 4 possible letters
hidden_size = 8
output_size = 4  # Predict among 4 letters

rnn = SimpleRNN(input_size, hidden_size, output_size)
hidden = rnn.init_hidden()

# Let's create a dummy input for 'H' (assuming one-hot encoding)
h_letter = torch.tensor([[1, 0, 0, 0]], dtype=torch.float)

output, hidden = rnn(h_letter, hidden)
print("Output (probabilities):", output)
print("Hidden state:", hidden)
```

```
Output (probabilities): tensor([[-1.6606, -0.9803, -1.5722, -1.4819]], grad_fn=<LogSoftmaxBackward0>)
Hidden state: tensor([[ 0.0999,  0.0120,  0.2132, -0.3944, -0.2753,  0.2929,  0.0459, -0.1475]],
       grad_fn=<TanhBackward0>)
```

7. Example Calculations

Let's do a small, simplified mock calculation (ignoring actual matrix shapes for simplicity):

1. **Weights** (W_{xh}, W_{hh}) are randomly initialized. Suppose we have a hidden size of 2.
2. **Bias** (b_h) is also randomly initialized.
3. **Input** at time t is x_t. Let's say $x_t = [1, 0, 0, 0]$ for the letter 'H'.
4. **Hidden State** at time $t-1$ is h_{t-1}. Suppose it starts as $[0, 0]$.
5. The new hidden state: $h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$.
6. The output: $y_t = W_{hy}h_t + b_y$ -> we then apply something like softmax.
7. **During training**, we compare the predicted output to the actual label (the next letter) and adjust weights via BPTT.

8. Step-by-Step Example from Scratch

1. **Prepare Data**: Convert text sequence into numeric form. (One-hot or embedded vectors)
2. **Initialize RNN**: Decide input size, hidden size, output size.
3. **Forward Pass**: For each time step:
    - Compute new hidden state with old hidden state and current input.
    - Compute output (prediction).
4. **Loss Calculation**: Compare predictions with target labels.
5. **Backpropagation Through Time (BPTT)**: Unroll the RNN and compute gradients for each time step.
6. **Update Weights**: Adjust using optimizer (e.g. SGD, Adam).
7. **Repeat** until predictions converge or we reach a desired accuracy.

9. Illustrative Problem It Solves

RNNs are great for **sequence prediction**: for example, predicting the next word in a sentence. If your sentence is “I love going to the …,” the RNN will use context from the entire sentence to guess the next word (maybe “park” or “movies”).

10. Real World Problem

A **real world** application is **sentiment analysis** on social media posts. RNNs can read each post word by word, maintain context, and classify if it's positive, negative, or neutral. They can also do **language translation** by reading one language's words and predicting the corresponding words in another language.

11. How to Solve a Real World Problem Using RNN

1. **Data Collection**: Gather text data. Example: thousands of tweets labeled with sentiment.

2. **Data Preprocessing**: Clean text, remove punctuation, convert to tokens.
3. **Vectorization**: Use embeddings or one-hot encoding.
4. **Build/Use RNN**: LSTM or GRU or Vanilla RNN.
5. **Train**: With a labeled dataset, minimizing a loss function.
6. **Evaluate**: Use accuracy, F1-score, etc.
7. **Deploy**: Serve the model via an API.

12. Points to Ponder (Questions)

1. **How do RNNs handle long sequences?**
2. **What is the vanishing gradient problem?**
3. **How do LSTMs and GRUs differ from vanilla RNNs?**
4. **What is BPTT (Backpropagation Through Time)?**
5. **What are possible solutions to exploding gradients?**
6. **Why do we often use embeddings instead of one-hot vectors?**

13. Answers

1. **How do RNNs handle long sequences?**
    - They use hidden states to pass information forward. However, for very long sequences, they often rely on LSTM/GRU to mitigate forgetting.
2. **What is the vanishing gradient problem?**
    - When gradients become extremely small during backprop, updates become negligible, and the network stops learning long-range dependencies.
3. **How do LSTMs and GRUs differ from vanilla RNNs?**
    - They have gating mechanisms (like forget gate, input gate) to better control what is remembered and what is forgotten.
4. **What is BPTT (Backpropagation Through Time)?**
    - It's the method of training RNNs by unfolding them across time steps and computing gradients at each step.
5. **What are possible solutions to exploding gradients?**
    - Gradient clipping, using LSTM or GRU, proper initialization.
6. **Why do we often use embeddings instead of one-hot vectors?**
    - Embeddings capture semantic relationships and reduce dimensionality compared to sparse one-hot vectors.

14. A Sample Exercise

Below is a minimal example of training a small RNN on a toy sequence task (predicting the next digit in a repeating sequence). Please complete the TODO sections.

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# Toy dataset: A repeating sequence of [0,1,2,3], predict next in sequence
# We'll create a small dataset of sequences of length 3, label is the 4th.

sequence = [0, 1, 2, 3]
data = []
labels = []

#TODO - There is an error in the following code, can you figure out the problem and fix it?
# HINT - It's an IndexError: list index out of range. This means you're trying to access an element of
#        the sequence list using an index that's larger than the list's size.
# for i in range(len(sequence) - 1):
#     data.append(sequence[i:i+2])  # e.g. [0,1], [1,2], [2,3]
#     labels.append(sequence[i+2])  # e.g. next number

for i in range(len(sequence) - 2):
    data.append(sequence[i:i+2])  # e.g. [0,1], [1,2], [2,3]
    labels.append(sequence[i+2])  # e.g. next number
```

```
    labels.append(sequence[-1])  # e.g. next number

# Convert to tensors
data_tensors = torch.tensor(data, dtype=torch.long)
labels_tensors = torch.tensor(labels, dtype=torch.long)

class ToyRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(ToyRNN, self).__init__()
        self.hidden_size = hidden_size

        # We will use an embedding for input
        self.embedding = nn.Embedding(input_size, input_size)

        # RNN cell
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)

        # Final output layer
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # x shape: (batch_size, seq_length)
        embedded = self.embedding(x)
        # shape: (batch_size, seq_length, input_size)

        # TODO: Initialize hidden state (size: 1 x batch_size x hidden_size)
        # hidden = ???
        hidden = torch.zeros(1, embedded.size(0), self.hidden_size)

        # RNN forward
        out, hidden = self.rnn(embedded, hidden)
        # out shape: (batch_size, seq_length, hidden_size)

        # We only want the last time step's output
        # TODO: Extract the last step's output from 'out'.
        # last_out = ??? # shape: (batch_size, hidden_size)
        last_out = out[:, -1, :]

        # Pass through fully connected layer
        logits = self.fc(last_out)
        return logits

# Hyperparameters
input_size = 4  # digits 0,1,2,3
hidden_size = 8
output_size = 4
lr = 0.01
epochs = 50

# Model, loss, optimizer
model = ToyRNN(input_size, hidden_size, output_size)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=lr)

for epoch in range(epochs):
    # Forward
    logits = model(data_tensors)
    loss = criterion(logits, labels_tensors)

    # Backward
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch+1) % 10 == 0:
        print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}")

# Test: let's predict the sequence [2,3] -> what is next?
test_input = torch.tensor([[2, 3]], dtype=torch.long)
with torch.no_grad():
    prediction = model(test_input)
    pred_label = torch.argmax(prediction, dim=1)
    print("Test input: [2,3], Predicted next number:", pred_label.item())
```

```
Epoch 10/50, Loss: 0.3404
Epoch 20/50, Loss: 0.1173
Epoch 30/50, Loss: 0.0528
Epoch 40/50, Loss: 0.0305
Epoch 50/50, Loss: 0.0211
Test input: [2,3], Predicted next number: 2
```

Explanation:

- **Embedding**: We turn the integer inputs (0,1,2,3) into embedding vectors.
- **RNN**: Processes each sequence step by step.
- **Hidden State**: Re-initialized for each sample.
- **Fully Connected Layer**: Maps final hidden state to an output (one of 0,1,2,3).

Fill the `TODO` comments (we already gave the typical solutions, but keep them to practice).

15. Glossary

RNN (Recurrent Neural Network): A type of neural network designed for sequential data processing.

Hidden State: A vector that stores information from previous time steps.

Weight Matrices (W_{xh}, W_{hh}, W_{hy}): Parameters that transform input and hidden state.

Bias (b_h, b_y): Additional parameter added to the transformation for shifting.

Activation Function (σ): A function (e.g. $\tanh$, ReLU) applied after the linear transformation.

Backpropagation Through Time (BPTT): The process of training RNNs by unrolling them over time steps.

Vanishing Gradient: A problem where gradients become extremely small, hindering learning.

Exploding Gradient: A problem where gradients become extremely large, causing unstable updates.

Embedding: A dense representation of input tokens (words, digits, etc.) in a lower-dimensional space.

End of Notebook

```
import os, sys, platform, datetime, uuid, socket

def signoff(name="Anonymous"):
    colab_check = "Yes" if 'google.colab' in sys.modules else "No"
    mac_addr = ':'.join(format((uuid.getnode() >> i) & 0xff, '02x') for i in reversed(range(0, 48, 8)))
    print("+++ Acknowledgement +++")
    print(f"Executed on: {datetime.datetime.now()}")
    print(f"In Google Colab: {colab_check}")
    print(f"System info: {platform.system()} {platform.release()}")
    print(f"Node name: {platform.node()}")
    print(f"MAC address: {mac_addr}")
    try:
        print(f"IP address: {socket.gethostbyname(socket.gethostname())}")
    except:
        print("IP address: Unknown")
    print(f"Signing off, {name}")

signoff("Ali Muhammad Asad")
```

```
+++ Acknowledgement +++
Executed on: 2025-01-28 18:11:44.760650
In Google Colab: No
System info: Linux 6.8.0-51-generic
Node name: alimuhammad-Inspiron-7559
MAC address: 20:47:47:74:94:05
IP address: 127.0.1.1
Signing off, Ali Muhammad Asad
```

Start coding or generate with AI.