

RNNs, LSTMs, and Sequence Modeling

Lecture Overview

This lecture covers recurrent neural networks for sequence modeling, including vanilla RNNs, LSTMs, and their applications.

Topics Covered:

- RNN architecture and basics
- Vanishing gradient problem
- LSTM gating mechanisms
- Bidirectional RNNs
- Sequence classification

1. Recurrent Neural Networks

1.1 RNN Basics

```
import torch
import torch.nn as nn

print("Recurrent Neural Networks (RNNs)")
print("="*70)

print("Problem: How to process sequences of variable length?")

print("\nRNN idea: Maintain hidden state across time steps")
print("  h_t = tanh(W_{hh} * h_{t-1} + W_{xh} * x_t + b)")
print("  y_t = W_{hy} * h_t + b_y")

print("\nKey features:")
print("  - Same weights at each time step (parameter sharing)")
print("  - Hidden state captures sequence history")
print("  - Processes one token at a time")

class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.W_xh = nn.Linear(input_size, hidden_size)
        self.W_hh = nn.Linear(hidden_size, hidden_size)

    def forward(self, x, h=None):
        batch_size, seq_len, _ = x.shape
        if h is None:
            h = torch.zeros(batch_size, self.hidden_size)

        outputs = []
        for t in range(seq_len):
            h = torch.tanh(self.W_xh(x[:, t]) + self.W_hh(h))
            outputs.append(h)

        return torch.stack(outputs, dim=1), h

rnn = SimpleRNN(input_size=32, hidden_size=64)
x = torch.randn(4, 10, 32) # batch=4, seq=10, features=32
out, h_final = rnn(x)
print(f"\nInput: {x.shape}")
print(f"Output: {out.shape}")
print(f"Final hidden: {h_final.shape}")

Recurrent Neural Networks (RNNs)
=====
Problem: How to process sequences of variable length?

RNN idea: Maintain hidden state across time steps
  h_t = tanh(W_{hh} * h_{t-1} + W_{xh} * x_t + b)
  y_t = W_{hy} * h_t + b_y

Key features:
  - Same weights at each time step (parameter sharing)
  - Hidden state captures sequence history
  - Processes one token at a time

Input: torch.Size([4, 10, 32])
Output: torch.Size([4, 10, 64])
Final hidden: torch.Size([4, 64])
```

1.2 PyTorch RNN

```
print("PyTorch RNN")
print("="*70)

rnn = nn.RNN(
    input_size=32,
    hidden_size=64,
    num_layers=2,
```

```

batch_first=True,
bidirectional=False
)

print("nn.RNN parameters:")
print(f"  input_size: 32")
print(f"  hidden_size: 64")
print(f"  num_layers: 2")
print(f"  batch_first: True")

x = torch.randn(4, 10, 32)
h0 = torch.zeros(2, 4, 64)  # (num_layers, batch, hidden)

out, h_n = rnn(x, h0)
print(f"\nInput: {x.shape}")
print(f"Output: {out.shape}")
print(f"h_n: {h_n.shape}")

print("\nRNN Problem: Vanishing Gradients")
print("  - Long sequences: gradients vanish/explode")
print("  - Hard to capture long-range dependencies")
print("  - Solution: LSTM and GRU")

PyTorch RNN
=====
nn.RNN parameters:
  input_size: 32
  hidden_size: 64
  num_layers: 2
  batch_first: True

Input: torch.Size([4, 10, 32])
Output: torch.Size([4, 10, 64])
h_n: torch.Size([2, 4, 64])

RNN Problem: Vanishing Gradients
  - Long sequences: gradients vanish/explode
  - Hard to capture long-range dependencies
  - Solution: LSTM and GRU

```

2. LSTM

2.1 LSTM Architecture

```
print("Long Short-Term Memory (LSTM)")  
print("=*70)  
  
print("LSTM adds gating mechanisms to control information flow")  
  
print("\nThree gates:")  
print("  Forget gate: What to forget from cell state")  
print("    f_t = sigmoid(W_f · [h_{t-1}, x_t] + b_f)")  
  
print("\n  Input gate: What new info to store")  
print("    i_t = sigmoid(W_i · [h_{t-1}, x_t] + b_i)")  
print("    c_tilde = tanh(W_c · [h_{t-1}, x_t] + b_c)")  
  
print("\n  Output gate: What to output")  
print("    o_t = sigmoid(W_o · [h_{t-1}, x_t] + b_o)")  
  
print("\nCell state update:")  
print("    c_t = f_t * c_{t-1} + i_t * c_tilde")  
print("    h_t = o_t * tanh(c_t)")  
  
print("\nWhy gates help:")  
print("  - Forget gate can be 1: perfect memory")  
print("  - Gradients flow through cell state")  
print("  - Learns what to remember/forget")  
  
Long Short-Term Memory (LSTM)  
=====
```

LSTM adds gating mechanisms to control information flow

Three gates:

Forget gate: What to forget from cell state
 $f_t = \text{sigmoid}(W_f \cdot [h_{t-1}, x_t] + b_f)$

Input gate: What new info to store
 $i_t = \text{sigmoid}(W_i \cdot [h_{t-1}, x_t] + b_i)$
 $c_{\text{tilde}} = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$

Output gate: What to output
 $o_t = \text{sigmoid}(W_o \cdot [h_{t-1}, x_t] + b_o)$

Cell state update:
 $c_t = f_t * c_{t-1} + i_t * c_{\text{tilde}}$
 $h_t = o_t * \tanh(c_t)$

Why gates help:

- Forget gate can be 1: perfect memory
- Gradients flow through cell state
- Learns what to remember/forget

2.2 PyTorch LSTM

```
print("PyTorch LSTM")  
print("=*70)  
  
lstm = nn.LSTM(  
    input_size=32,  
    hidden_size=64,  
    num_layers=2,  
    batch_first=True,  
    bidirectional=True  
)  
  
print("Bidirectional LSTM:")  
print("  - Process sequence forward and backward")  
print("  - Concatenate both hidden states")  
print("  - Output size: 2 * hidden_size")  
  
x = torch.randn(4, 10, 32)  
# (num_layers * num_directions, batch, hidden)
```

```
h0 = torch.zeros(4, 4, 64) # 2 layers * 2 directions
c0 = torch.zeros(4, 4, 64)

out, (h_n, c_n) = lstm(x, (h0, c0))
print(f"\nInput: {x.shape}")
print(f"Output: {out.shape} # 2*64=128 due to bidirectional")
print(f"h_n: {h_n.shape}")
print(f"c_n: {c_n.shape}")

PyTorch LSTM
=====
Bidirectional LSTM:
- Process sequence forward and backward
- Concatenate both hidden states
- Output size: 2 * hidden_size

Input: torch.Size([4, 10, 32])
Output: torch.Size([4, 10, 128]) # 2*64=128 due to bidirectional
h_n: torch.Size([4, 4, 64])
c_n: torch.Size([4, 4, 64])
```

3. Sequence Classification

3.1 LSTM Text Classifier

```
print("LSTM Text Classifier")
print("*" * 70)

class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_classes):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=0)
        self.lstm = nn.LSTM(embed_dim, hidden_dim, batch_first=True,
                           bidirectional=True)
        self.fc = nn.Linear(hidden_dim * 2, num_classes)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        # x: (batch, seq_len)
        embedded = self.dropout(self.embedding(x))
        lstm_out, (h_n, c_n) = self.lstm(embedded)

        # Concatenate final hidden states from both directions
        # h_n: (2, batch, hidden) for bidirectional
        hidden = torch.cat([h_n[-2], h_n[-1]], dim=1)

        return self.fc(self.dropout(hidden))

model = LSTMClassifier(
    vocab_size=10000,
    embed_dim=128,
    hidden_dim=256,
    num_classes=2
)

x = torch.randint(0, 10000, (32, 100))
out = model(x)
print(f"Input: {x.shape}")
print(f"Output: {out.shape}")

params = sum(p.numel() for p in model.parameters())
print(f"Parameters: {params:,}")

print("\nComparison: RNN vs LSTM vs GRU")
print("RNN: Simple, fast, vanishing gradients")
print("LSTM: Best for long sequences, more params")
print("GRU: Simpler than LSTM, similar performance")

LSTM Text Classifier
=====
Input: torch.Size([32, 100])
Output: torch.Size([32, 2])
Parameters: 2,111,234

Comparison: RNN vs LSTM vs GRU
RNN: Simple, fast, vanishing gradients
LSTM: Best for long sequences, more params
GRU: Simpler than LSTM, similar performance
```

Summary

Key Takeaways:

- RNNs process sequences by maintaining hidden state
- Vanilla RNNs suffer from vanishing gradients
- LSTMs use gates to control information flow
- Cell state allows long-range dependencies
- Bidirectional processes sequence both ways

- GRU is simpler alternative to LSTM

Practice Exercises:

1. Implement vanilla RNN from scratch
2. Compare RNN vs LSTM on long sequences
3. Build LSTM sentiment classifier
4. Implement GRU cell
5. Visualize LSTM gate activations