📚 Learning Objectives

- Understand the architecture of **RNNs**, **LSTMs**, and **GRUs**
- Implement **RNN-based models for NLP tasks**

🧠 1. RNN Architecture: Basic RNN

🔄 Why RNN?

RNNs are designed to **process sequential data**. For example: sentences, time series, music, etc.

They have a memory of **previous inputs**, unlike regular feed-forward networks.

🧱 Basic RNN Cell Architecture

Each time step has:

- Input xtx_t
- Hidden state hth_t (which stores memory)
- Output yty_t

Mathematical Formulation:

Let:

- xtx_t: input at time step tt
- ht−1h_{t-1}: hidden state from previous time step
- Wxh,Whh,bhW_{xh}, W_{hh}, b_h: weights and bias
- tanh\tanh: activation function

Then:

ht=tanh(Wxhxt+Whhht−1+bh)h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h) yt=Whyht+byy_t = W_{hy}h_t + b_y

---

🧪 Example in Code (Basic RNN using PyTorch)

```
import torch

import torch.nn as nn


# Sample input (sequence_length=5, batch_size=1, input_size=10)
```

```
x = torch.randn(5, 1, 10)

rnn = nn.RNN(input_size=10, hidden_size=20, num_layers=1, batch_first=False)

# Initial hidden state (num_layers, batch, hidden_size)
h0 = torch.zeros(1, 1, 20)

out, hn = rnn(x, h0)

print("Output shape:", out.shape)
print("Final hidden state:", hn.shape)
```

🧠 2. LSTM (Long Short-Term Memory)

🧩 Why LSTM?

RNNs struggle with **long-term dependencies** due to **vanishing gradients**. LSTMs fix this using **gates** that control memory.

🧱 LSTM Cell Architecture

LSTM has:

- Cell state CtC_t (long-term memory)
- Hidden state hth_t (short-term)
- Gates: forget ftf_t, input iti_t, output oto_t

Mathematics:

Let:

- σ\sigma: sigmoid
- tanh\tanh: tanh
- ⊙\odot: element-wise multiplication

ft=σ(Wf·[ht−1,xt]+bf)f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) it=σ(Wi·[ht−1,xt]+bi)i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) C~t=tanh(WC·[ht−1,xt]+bC)\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)

$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$ $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$ $h_t = o_t \odot \tanh(C_t)$

🧪 Code Example: LSTM

```
lstm = nn.LSTM(input_size=10, hidden_size=20, num_layers=1)


x = torch.randn(5, 1, 10)

h0 = torch.zeros(1, 1, 20)

c0 = torch.zeros(1, 1, 20)


out, (hn, cn) = lstm(x, (h0, c0))


print("LSTM output:", out.shape)
```

🧠 3. GRU (Gated Recurrent Unit)

🔧 Why GRU?

GRU is a **simplified version of LSTM** — it has **fewer gates** and **faster training**, but similar performance.

🧱 GRU Cell Architecture

Has:

- Update gate z_t
- Reset gate r_t
- Candidate memory $\tilde{h}_t$

Mathematics:

$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$ $r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$ $\tilde{h}_t = \tanh(W_h \cdot [r_t \odot h_{t-1}, x_t])$ $h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$

🧪 Code Example: GRU

```
gru = nn.GRU(input_size=10, hidden_size=20, num_layers=1)


x = torch.randn(5, 1, 10)

h0 = torch.zeros(1, 1, 20)


out, hn = gru(x, h0)


print("GRU output:", out.shape)
```

📝 4. Implementing RNN for NLP Task (Text Classification)

We'll build a simple **sentiment classifier** using an LSTM on IMDB dataset.

🧪 Code Example (Text Classification with LSTM)

```
import torch

import torch.nn as nn

from torchtext.datasets import IMDB

from torchtext.data.utils import get_tokenizer

from torchtext.vocab import build_vocab_from_iterator

from torch.utils.data import DataLoader

from torch.nn.utils.rnn import pad_sequence


# Tokenizer and vocab

tokenizer = get_tokenizer("basic_english")


def yield_tokens(data_iter):

    for label, line in data_iter:

        yield tokenizer(line)


train_iter = IMDB(split='train')
```

```
vocab = build_vocab_from_iterator(yield_tokens(train_iter), specials=["<pad>", "<unk>"])
vocab.set_default_index(vocab["<unk>"])

# Pipeline
def text_pipeline(x): return vocab(tokenizer(x))
def label_pipeline(x): return 1 if x == 'pos' else 0

# Collate function
def collate_batch(batch):
    text_list, label_list = [], []
    for label, text in batch:
        text_tensor = torch.tensor(text_pipeline(text), dtype=torch.int64)
        text_list.append(text_tensor)
        label_list.append(torch.tensor(label_pipeline(label), dtype=torch.int64))
    text_batch = pad_sequence(text_list, padding_value=0)
    return text_batch, torch.tensor(label_list)

train_iter = IMDB(split='train')
dataloader = DataLoader(list(train_iter)[:1000], batch_size=8, collate_fn=collate_batch)

# LSTM Model
class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, output_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.lstm = nn.LSTM(embed_dim, hidden_dim)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
```

```python
        embedded = self.embedding(x)
        output, (hn, cn) = self.lstm(embedded)
        return self.fc(hn[-1])

model = LSTMClassifier(len(vocab), 100, 128, 2)

# Training loop (simplified)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

for epoch in range(3):
    total_loss = 0
    for text, label in dataloader:
        optimizer.zero_grad()
        output = model(text)
        loss = criterion(output, label)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {total_loss:.4f}")
```

📌 **Summary**

Model	Pros	Cons
RNN	Simple, good for short sequences	Vanishing gradient, poor long-term memory
LSTM	Good long-term memory	More parameters, slower
GRU	Faster than LSTM, good performance	No explicit cell state

---