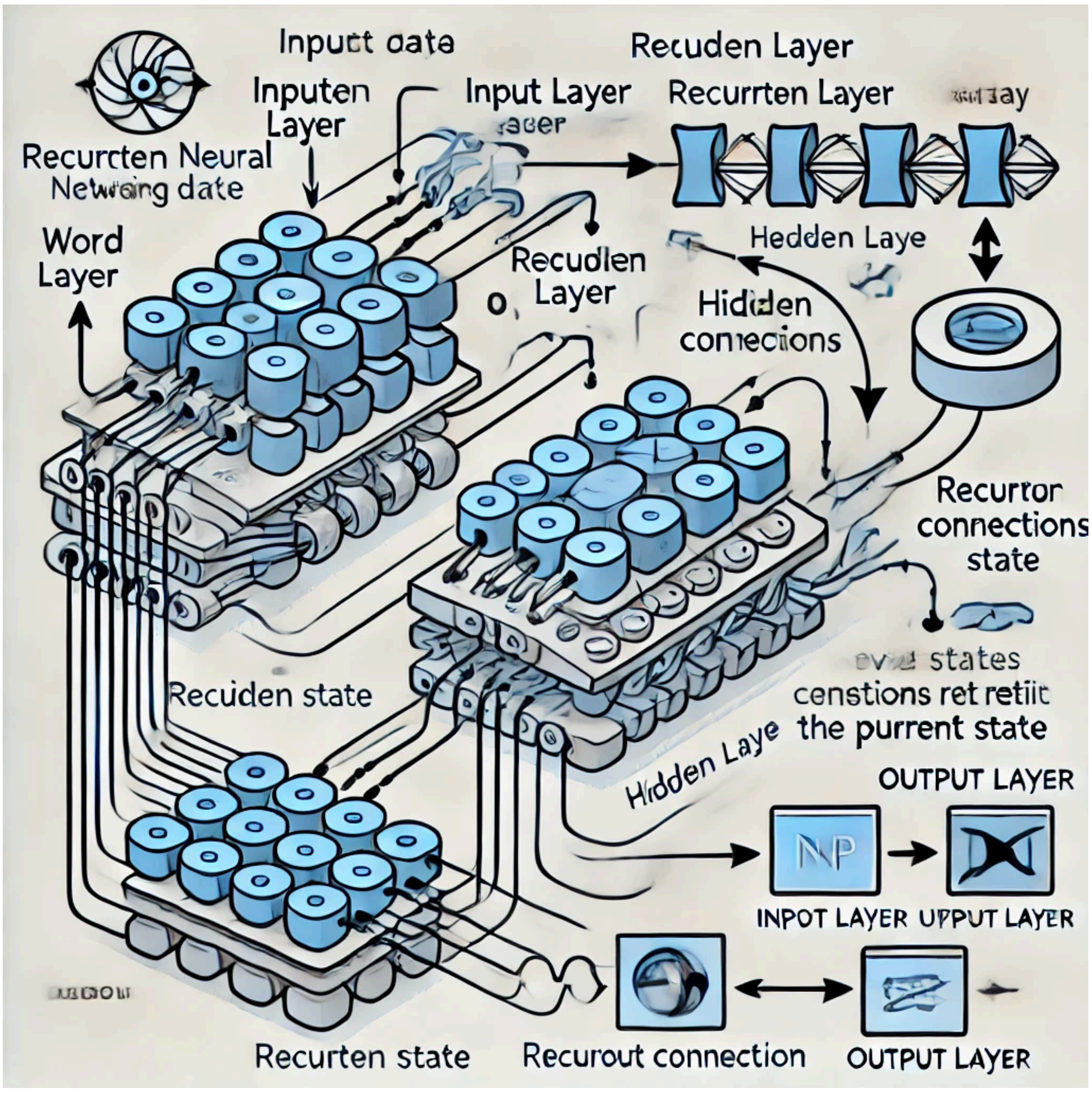
005.2. Recurrent Neural Networks (RNNs)



Recurrent Neural Networks (RNNs) are a type of neural network designed specifically for sequential data, such as time series, speech, or natural language. They are particularly effective when the order or context of data points matters, as they allow information to persist over time.

Recurrent Neural Networks (RNNs) are powerful models for sequential data, capable of maintaining a hidden state or memory over time. Although standard RNNs are limited by issues like the vanishing gradient problem, they form the foundation for more advanced architectures like LSTMs and GRUs, which are now commonly used for tasks requiring long-range dependencies.

Key Concepts of RNNs:

1. Sequential Data Handling

Unlike feedforward neural networks, RNNs can process sequences of data by maintaining an internal state (or memory) that allows them to retain information from previous time steps. This makes them ideal for tasks like text processing, where word order is crucial.

2. Recurrent Connections

In an RNN, each neuron (or layer of neurons) has connections not only to the next layer but also back to itself (hence "recurrent"). This feedback loop allows information to be passed from one step to the next.

The output of the current step is influenced by the previous step's output, which gives the network a sense of "memory" over time.

The basic RNN cell can be described as:

$$h_t = f(W_{ih}x_t + W_{hh}h_{t-1} + b_h)$$

Where:

- h_t is the hidden state at time step t,
- x_t is the input at time step t,
- h_{t-1} is the hidden state from the previous time step,
- W_{ih} and W_{hh} are the weight matrices,
- b_h is the bias,
- f is a non-linear activation function (commonly *tanh* or *ReLU*).

3. Internal Memory (Hidden State)

The key feature of RNNs is their ability to maintain a **hidden state**, which acts as a memory of the previous inputs in the sequence. This hidden state is updated at every time step as new inputs are received.

The hidden state allows RNNs to maintain context over a sequence of inputs, enabling them to process sentences in NLP or recognize patterns over time in time-series data.

4. Backpropagation Through Time (BPTT)

RNNs are **trained** using a variant of the standard backpropagation algorithm called **Backpropagation Through Time (BPTT)**. In BPTT, gradients are computed for the recurrent connections by "unfolding" the network over time, and errors are propagated backward through the entire sequence.

Vanishing/Exploding Gradient Problem: When dealing with long sequences, RNNs often suffer from the **vanishing gradient problem**, where gradients shrink exponentially during backpropagation, making it difficult to learn long-range dependencies. Conversely, gradients can explode, causing instability.

5. Types of RNNs

There are different variants of RNNs based on the sequence input/output structure:

- **Many-to-One**: A common RNN structure where a sequence of inputs produces a single output, such as in sentiment analysis where an entire sentence results in a classification (positive/negative sentiment).
- **Many-to-Many**: This structure processes a sequence of inputs to produce a sequence of outputs. For example, machine translation systems translate one sentence into another in a different language, word by word.
- **One-to-Many**: A single input can generate a sequence of outputs, such as generating text or music from a single seed input.

6. Limitations:Short-Term Memory, Training Instability

Standard RNNs struggle to retain information over long sequences due to the vanishing gradient problem. They are better suited for short-term dependencies.

Training can be slow and unstable, especially when dealing with long sequences, requiring techniques like gradient clipping to mitigate the exploding gradient problem.

7. Variants of RNNs

To address the limitations of standard RNNs, two key variants have been developed: Long Short-Term Memory Networks (LSTMs) and Gated Recurrent Units (GRUs)

Long Short-Term Memory Networks (LSTMs): LSTMs use special gates (input, forget, and output gates) to regulate the flow of information, allowing them to maintain memory over long sequences more effectively.

Gated Recurrent Units (GRUs): GRUs simplify the LSTM architecture by combining the forget and input gates into a single update gate, making them faster to train while still managing long-term dependencies.

8. Applications of RNNs

RNNs are widely used in various sequential data tasks, including:

- **Natural Language Processing (NLP)**: Tasks like machine translation, text generation, and speech recognition heavily rely on RNNs.
- **Time Series Prediction**: Forecasting stock prices, weather prediction, and other temporal tasks.
- **Speech Recognition**: Understanding spoken language by processing audio as sequential data.
- **Video Analysis**: Recognizing actions or events in videos.

Example in Python

Below is a Python example using **PyTorch** to implement a simple **Recurrent Neural Network (RNN)** for a text classification task. We'll use the **IMDb dataset** for sentiment analysis, classifying reviews as either positive or negative.

The model will output the training and testing loss and accuracy at each epoch, indicating its performance on the IMDb sentiment classification task.

This is a simple RNN-based text classifier. For better performance, you can explore using LSTMs or GRUs, which are better at handling long-range dependencies in sequential data.

Requirements

You'll need the following libraries installed:

```
pip install torch torchtext
```

Example Code: Recurrent Neural Network for Text Classification

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchtext.legacy import data, datasets
import random

# Set the random seeds for reproducibility
SEED = 1234
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

# Define fields for the text and labels
TEXT = data.Field(tokenize='spacy',
tokenizer_language='en_core_web_sm', include_lengths=True)
LABEL = data.LabelField(dtype=torch.float)

# Load IMDb dataset
train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)

# Build the vocabulary using pre-trained word embeddings (e.g.,
GloVe)
TEXT.build_vocab(train_data, max_size=25_000,
vectors="glove.6B.100d", unk_init=torch.Tensor.normal_)
LABEL.build_vocab(train_data)

# Create iterators for batching the dataset
BATCH_SIZE = 64
```

```
train_iterator, test_iterator = data.BucketIterator.splits(
      (train_data, test_data),
      batch_size=BATCH_SIZE,
      sort_within_batch=True, # necessary for using RNN
                              # with packed sequences
      device=torch.device(
            'cuda' if torch.cuda.is_available() else 'cpu'
      )
)

# Define the Recurrent Neural Network (RNN) model
class RNN(nn.Module):
      def __init__(
            self, vocab_size, embedding_dim, hidden_dim, output_dim
      ):
            super(RNN, self).__init__()

            # Embedding layer
            self.embedding = nn.Embedding(vocab_size, embedding_dim)

            # RNN layer
            self.rnn = nn.RNN(embedding_dim, hidden_dim)

            # Fully connected output layer
            self.fc = nn.Linear(hidden_dim, output_dim)

      def forward(self, text, text_lengths):
            # text: (sentence_length, batch_size)
            embedded = self.embedding(text)

            # Pack the sequence for efficient processing
            packed_embedded = nn.utils.rnn.pack_padded_sequence(
                  embedded, text_lengths.to('cpu')
            )

            # Pass through the RNN
            packed_output, hidden = self.rnn(packed_embedded)

            # Use the final hidden state for classification
            return self.fc(hidden.squeeze(0))

# Hyperparameters
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100  # Must match the size of the GloVe vectors
used
HIDDEN_DIM = 256
OUTPUT_DIM = 1
```

```
# Initialize the model
model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)

# Load the pre-trained GloVe embeddings
pretrained_embeddings = TEXT.vocab.vectors
model.embedding.weight.data.copy_(pretrained_embeddings)

# Training setup
optimizer = optim.Adam(model.parameters())
criterion = nn.BCEWithLogitsLoss()

# Move the model and criterion to the GPU if available
device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
model = model.to(device)
criterion = criterion.to(device)

# Function to calculate accuracy
def binary_accuracy(preds, y):
    rounded_preds = torch.round(torch.sigmoid(preds))
    correct = (rounded_preds == y).float()
    return correct.sum() / len(correct)

# Function to train the model
def train(model, iterator, optimizer, criterion):
    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for batch in iterator:
        optimizer.zero_grad()
        text, text_lengths = batch.text
        predictions = model(text, text_lengths).squeeze(1)
        loss = criterion(predictions, batch.label)
        acc = binary_accuracy(predictions, batch.label)
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)

# Function to evaluate the model
def evaluate(model, iterator, criterion):
    epoch_loss = 0
    epoch_acc = 0
    model.eval()

    with torch.no_grad():
    for batch in iterator:
        text, text_lengths = batch.text
        predictions = model(text, text_lengths).squeeze(1)
        loss = criterion(predictions, batch.label)
        acc = binary_accuracy(predictions, batch.label)

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)

# Training loop
N_EPOCHS = 5

for epoch in range(N_EPOCHS):
    train_loss, train_acc = train(model, train_iterator,
optimizer, criterion)
    test_loss, test_acc = evaluate(model, test_iterator,
criterion)

    print(f'Epoch {epoch+1}')
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc:
{train_acc*100:.2f}%')
    print(f'\tTest Loss: {test_loss:.3f} | Test Acc:
{test_acc*100:.2f}%')

# Save the model
torch.save(model.state_dict(), 'rnn_model.pth')
```

Key Components of the Code:

1. **Data Preprocessing**: The IMDb dataset is tokenized using `spacy`. We use the `Field` and `LabelField` to define how text and labels should be processed.
2. **Embedding Layer**: The embedding layer converts input words into dense vectors (using pre-trained GloVe embeddings).
3. **RNN Layer**: The RNN layer processes the sequence of word embeddings. It outputs the hidden state, which is then used to classify the input sentence.
4. **Packing Sequences**: `pack_padded_sequence` is used to handle variable-length sentences efficiently during training.
5. **Training and Evaluation**: The model is trained using binary cross-entropy loss (`BCEWithLogitsLoss`) and evaluated after every epoch.