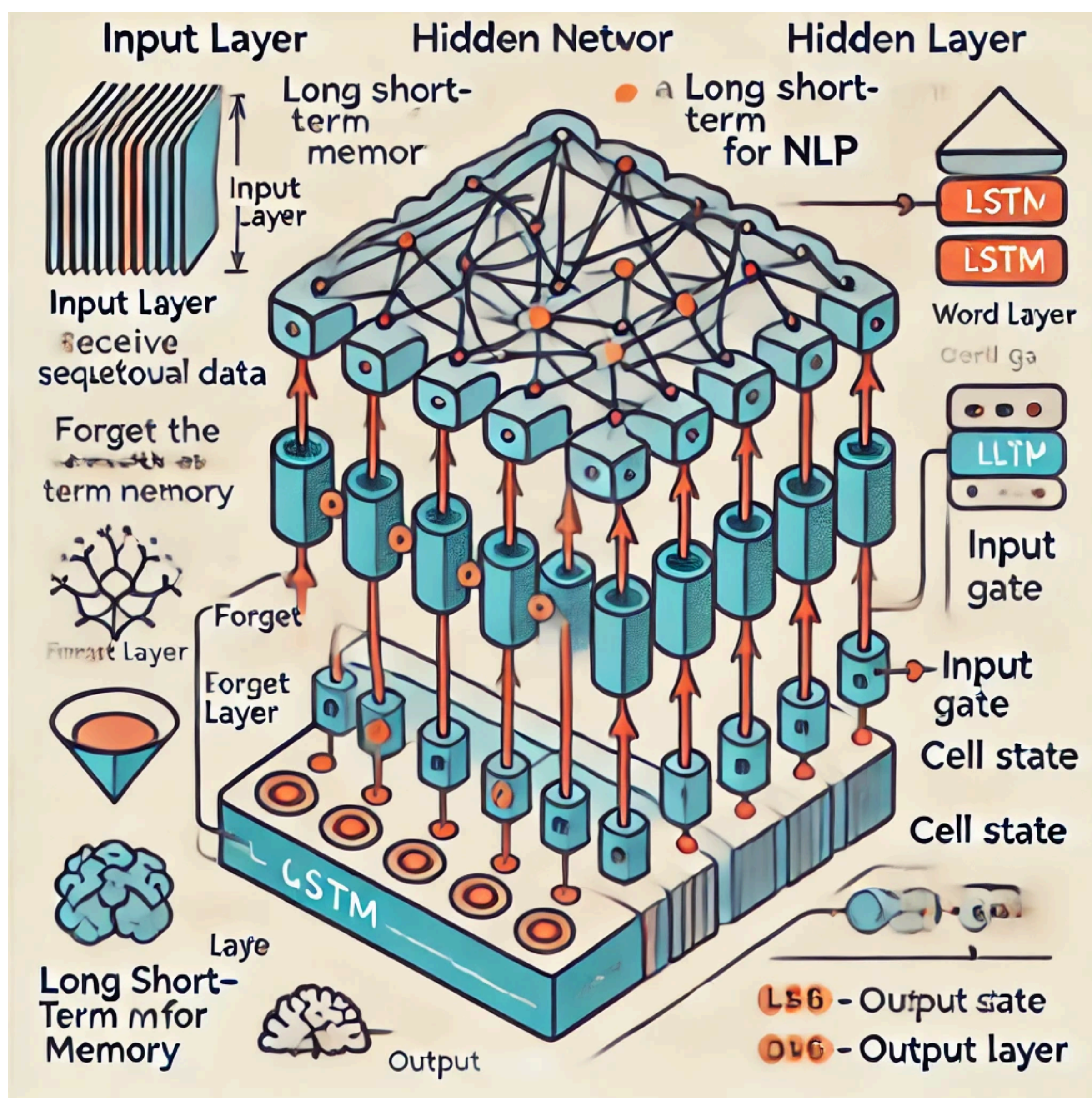
005.3 LSTM



Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) designed to overcome the limitations of traditional RNNs, particularly the **vanishing gradient problem**, which makes it difficult for RNNs to learn long-range dependencies. LSTMs are widely used in tasks that involve sequential data, such as natural language processing, speech recognition, and time-series prediction, where maintaining context over longer sequences is crucial.

LSTMs are a powerful type of RNN designed to handle long-term dependencies in sequential data. They achieve this by using gates to control the flow of information, allowing them to selectively remember or forget information over time. LSTMs are widely used in a variety of fields, especially NLP, time-series forecasting, and speech recognition, where understanding context and sequence is essential.

Key Concepts of LSTMs:

1. LSTM Cell Structure

The primary innovation of LSTMs is their unique cell structure, which includes mechanisms for regulating the flow of information. Each LSTM cell has three key components:

- **Cell State**: The "memory" of the cell, which runs through the entire sequence with minimal modifications, allowing the network to store information over long periods.
- **Gates**: These are responsible for regulating the flow of information into, out of, and within the LSTM cell. There are three types of gates:
    - **Forget Gate**: Decides what information from the cell state should be discarded.
    - **Input Gate**: Decides what new information should be stored in the cell state.
    - **Output Gate**: Decides what part of the cell state should be output as the hidden state for the next time step.

2. LSTM Cell Operation

Here's a breakdown of how an LSTM cell operates:

- **Forget Gate**: The forget gate controls how much of the previous cell state C_{t-1} should be "forgotten" or discarded. It uses a sigmoid function to produce a value between 0 and 1, where 1 means "keep everything" and 0 means "forget everything".

  $$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

- **Input Gate**: The input gate controls how much of the new information should be added to the cell state. It has two steps: a sigmoid layer decides which values will be updated, and a tanh layer creates a vector of new candidate values to add to the state.

  $$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

  $$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

- **Cell State Update**: The cell state C_t is updated by combining the old cell state C_{t-1} (after multiplying by the forget gate's output) and the new candidate values $\tilde{C}_t$ (after multiplying by the input gate's output).

  $$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- **Output Gate**: The output gate controls what information is passed to the next hidden state. It first uses a sigmoid function to decide which parts of the cell state should be output, and then multiplies the cell state by the output of the sigmoid gate.

  $$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

  $$h_t = o_t * \tanh(C_t)$$

3. Key Advantages of LSTMs

- **Long-Range Dependency**: LSTMs are capable of learning and remembering information over long sequences, addressing the vanishing gradient problem that affects traditional RNNs.
- **Selective Memory**: By controlling what information is stored, updated, or forgotten via the gates, LSTMs can selectively "remember" important parts of a sequence and "forget" irrelevant ones, improving learning efficiency.
- **Bidirectional LSTMs**: In many applications, it's useful to know both the previous and future context. Bidirectional LSTMs run two LSTMs on the input, one from the start to the end, and another from the end to the start. This can improve performance in tasks like speech recognition and text classification.

4. Training LSTMs

- **Backpropagation Through Time (BPTT)**: LSTMs, like standard RNNs, are trained using a variant of the backpropagation algorithm known as Backpropagation Through Time (BPTT). During BPTT, errors are propagated backward through each time step in the sequence.
- **Vanishing/Exploding Gradient Mitigation**: The internal gating mechanisms of LSTMs help mitigate the vanishing gradient problem, which allows them to capture dependencies over longer sequences more effectively than standard RNNs.

5. Variants of LSTM

- **Peephole LSTMs**: These LSTMs allow the gates to also look at the cell state, which can sometimes improve performance by providing the gates with more information.
- **Stacked LSTMs**: It's common to stack multiple LSTM layers on top of each other to create a deep LSTM network, which can capture even more complex patterns in the data.
- **Bidirectional LSTMs**: These use two LSTMs, one processing the input sequence forward and one processing it backward, to capture dependencies in both directions.

6. Applications of LSTMs

LSTMs have been successfully applied in many sequential data tasks, including:

- **Natural Language Processing (NLP)**:
  - Machine translation.
  - Text summarization.
  - Question answering.
  - Named entity recognition (NER).
- **Speech Recognition**: LSTMs are effective at processing audio signals and recognizing spoken language.
- **Time-Series Forecasting**: Predicting future values based on past time series data (e.g., stock prices, weather patterns).
- **Video Processing**: Understanding actions in video sequences.

Example in Python

Here is a Python example using **PyTorch** to implement a simple **Long Short-Term Memory (LSTM)** network for a text classification task. We'll use the **IMDb dataset** for sentiment analysis, where the LSTM will classify movie reviews as positive or negative.

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

You can further experiment by changing the number of layers, hidden units, dropout rates, or using unidirectional LSTMs for comparison.

Requirements

You'll need the following libraries installed:

```
pip install torch torchtext
```

Example Code: LSTM 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 seed 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
BATCH_SIZE = 64
```

```
train_iterator, test_iterator = data.BucketIterator.splits(
      (train_data, test_data),
      batch_size=BATCH_SIZE,
      sort_within_batch=True,
      device=torch.device(
            'cuda' if torch.cuda.is_available() else 'cpu'
      )
)

# Define the LSTM model
class LSTM(nn.Module):
      def __init__(
            self, vocab_size, embedding_dim,
            hidden_dim, output_dim, n_layers,
            bidirectional, dropout
      ):
            super(LSTM, self).__init__()

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

            # LSTM layer
            self.lstm = nn.LSTM(
                  embedding_dim, hidden_dim,
                  num_layers=n_layers,
                  bidirectional=bidirectional, dropout=dropout
            )

            # Fully connected layer
            self.fc = nn.Linear(
                  hidden_dim * 2 if bidirectional else hidden_dim,
                  output_dim
            )

            # Dropout layer
            self.dropout = nn.Dropout(dropout)

      def forward(self, text, text_lengths):
            # text: (sentence_length, batch_size)
            embedded = self.dropout(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 LSTM
```

```
        packed_output, (hidden, cell) = self.lstm(
            packed_embedded
        )

        # If the model is bidirectional, we need to
        # concatenate the forward and backward hidden states
        if self.lstm.bidirectional:
            hidden = torch.cat(
                (hidden[-2,:,:], hidden[-1,:,:]), dim=1)
        else:
            hidden = hidden[-1,:,:]

        # Pass through the fully connected layer
        return self.fc(self.dropout(hidden))

# Hyperparameters
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100  # Must match the GloVe vector size
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 2
BIDIRECTIONAL = True
DROPOUT = 0.5

# Initialize the model
model = LSTM(
    INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM,
    N_LAYERS, BIDIRECTIONAL, DROPOUT
)

# Load 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 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 binary accuracy
def binary_accuracy(preds, y):
    rounded_preds = torch.round(torch.sigmoid(preds))
    correct = (rounded_preds == y).float()
```

```
    return correct.sum() / len(correct)

# Training function
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)

# Evaluation function
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} '
              f'| Train Acc: {train_acc*100:.2f}%')
        print(f'\tTest Loss: {test_loss:.3f} '
              f'| Test Acc: {test_acc*100:.2f}%')

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

Key Components of the Code:

1. **Data Preprocessing**: The IMDb dataset is loaded and tokenized using `spacy`. The `Field` and `LabelField` from `torchtext` are used to preprocess the text and labels.
2. **Embedding Layer**: Pre-trained GloVe embeddings are loaded into the embedding layer, which converts input words into dense vectors.
3. **LSTM Layer**: The LSTM layer processes the sequence of word embeddings. It can be unidirectional or bidirectional, and you can stack multiple layers.
4. **Packed Sequences**: We use `pack_padded_sequence` to efficiently handle variable-length sequences, which is important for working with RNNs like LSTMs.
5. **Dropout and Fully Connected Layer**: Dropout is used to prevent overfitting, and the fully connected layer provides the final output.
6. **Training and Evaluation**: The model is trained using the `Adam` optimizer and evaluated with binary cross-entropy loss (`BCEWithLogitsLoss`), suitable for binary classification.