“Andriy's long-awaited sequel in his "The Hundred-Page" series of machine learning textbooks is a masterpiece of concision.”

— **Bob van Luij**t, CEO and Co-Founder of Weaviate

“Andriy has this almost supernatural talent for shrinking epic AI concepts down to bite-sized, ‘Ah, now I get it!’ moments.”

— **Jorge Torres**, CEO at MindsDB

“Andriy paints for us, in 100 marvelous strokes, the journey from linear algebra basics to the implementation of transformers.”

— **Florian Douetteau**, Co-founder and CEO at Dataiku

“Andriy's book is an incredibly concise, clear, and accessible introduction to machine learning.”

— **Andre Zayarni**, Co-founder and CEO at Qdrant

“This is one of the most comprehensive yet concise handbooks out there for truly understanding how LLMs work under the hood.”

— **Jerry Liu**, Co-founder and CEO at LlamaIndex

Featuring a foreword by **Tomáš Mikolov** and back cover text by **Vint Cerf**

The Hundred-Page Language Models Book

Andriy Burkov

ISBN 978-1-7780427-2-0

Publisher: True Positive Inc.

To my family, with love

"Language is the source of misunderstandings."
—Antoine de Saint-Exupéry*, The Little Prince*

"In mathematics you don't understand things. You just get used to them."
—John von Neumann

"Computers are useless. They can only give you answers."
— Pablo Picasso

The book is distributed on the "read first, buy later" principle

Contents

Chapter 3. Recurrent Neural Network

In this chapter, we explore recurrent neural networks, a fundamental architecture that revolutionized sequential data processing. While transformers have become dominant in many applications, understanding RNNs first provides an ideal stepping stone—their elegant design introduces key sequential processing concepts that make Transformer mathematics more intuitive. We'll examine RNNs' structure and applications in language modeling, building essential foundations for more advanced architectures.

3.1. Elman RNN

A **recurrent neural network**, or **RNN**, is a neural network designed for sequential data. Unlike **feedforward neural networks**, RNNs include loops in their connections, enabling information to carry over from one step in the sequence to the next. This makes them well-suited for tasks like time series analysis, natural language processing, and other sequential data problems.

To illustrate the sequential nature of RNNs, let's consider a neural network with a single unit. Consider the input document *"Learning from text is cool."* Ignoring case and punctuation, the matrix representing this document would be as follows:

Word	Embedding vector
learning	$[0.1,0.2,0.6]^\top$
from	$[0.2,0.1,0.4]^\top$
text	$[0.1,0.3,0.3]^\top$
is	$[0.0,0.7,0.1]^\top$
cool	$[0.5,0.2,0.7]^\top$
PAD	$[0.0,0.0,0.0]^\top$

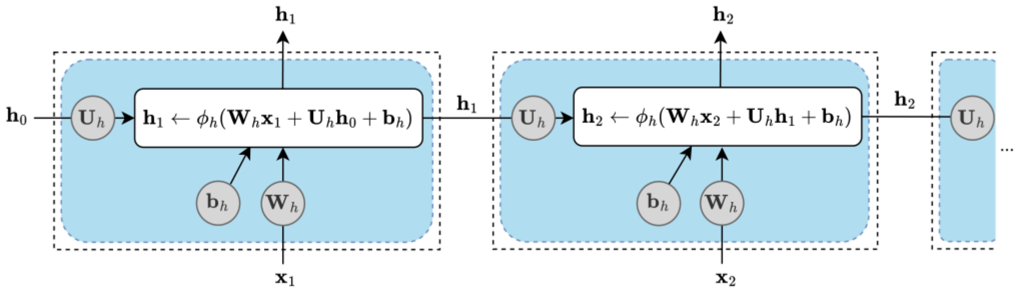
Each row of the matrix represents a word's embedding learned during neural network training. The order of words is preserved. The matrix dimensions are (sequence length, embedding dimensionality). Sequence length specifies the maximum number of words in a document. Shorter documents are padded with padding tokens (like PAD in this example), while longer ones are truncated. **Padding** uses dummy embeddings, usually **zero vectors**.

More formally, the matrix would look like this:

$$\mathbf{X} = \begin{bmatrix} 0.1 & 0.2 & 0.6 \\ 0.2 & 0.1 & 0.4 \\ 0.1 & 0.3 & 0.3 \\ 0.0 & 0.7 & 0.1 \\ 0.5 & 0.2 & 0.7 \\ 0.0 & 0.0 & 0.0 \end{bmatrix}$$

Here, we have five 3D embedding vectors, $\mathbf{x}_1, \dots, \mathbf{x}_5$, representing each word in the document. For instance, $\mathbf{x}_1 = [0.1,0.2,0.6]^\top$, $\mathbf{x}_2 = [0.2,0.1,0.4]^\top$, and so on. The sixth vector is a padding vector.

The **Elman RNN**, introduced by Jeffrey Locke Elman in 1990 as the **simple recurrent neural network**, processes a sequence of embedding vectors one at a time, as illustrated below:

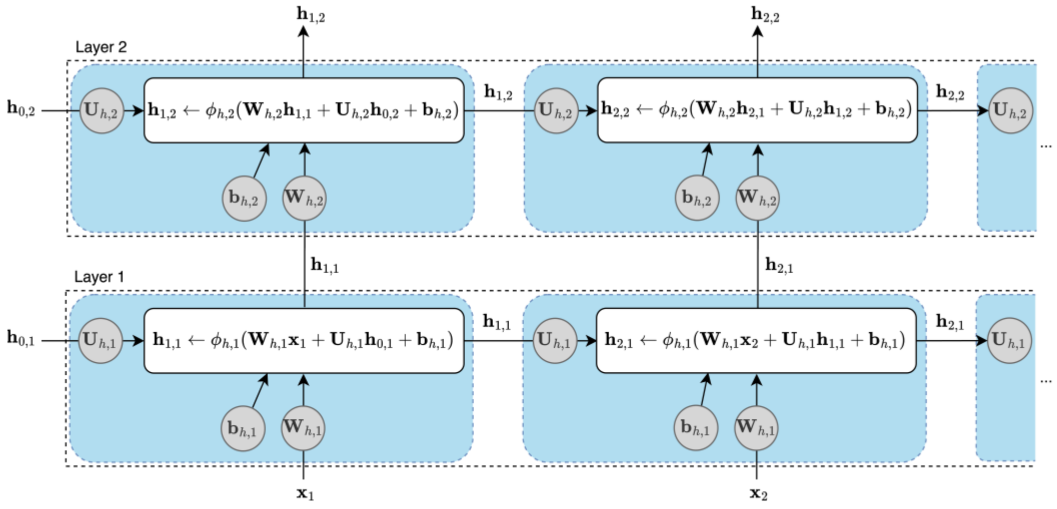


$\mathbf{h}_1$
$\mathbf{h}_2$
$\mathbf{h}_0$
$\mathbf{U}_h$
$\mathbf{h}_1 \leftarrow \phi_h(\mathbf{W}_h\mathbf{x}_1 + \mathbf{U}_h\mathbf{h}_0 + \mathbf{b}_h)$
$\mathbf{h}_2 \leftarrow \phi_h(\mathbf{W}_h\mathbf{x}_2 + \mathbf{U}_h\mathbf{h}_1 + \mathbf{b}_h)$
$\mathbf{b}_h$
$\mathbf{W}_h$
$\mathbf{x}_1$
$\mathbf{x}_2$
...

At each time step t, the current input embedding $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$ are combined by multiplying them with trainable weight matrices $\mathbf{W}_h$ and $\mathbf{U}_h$, adding a bias vector $\mathbf{b}_h$, and producing the updated hidden state $\mathbf{h}_t$. Unlike MLP units, which output scalars, an RNN unit outputs vectors and acts as an entire layer. The initial hidden state $\mathbf{h}_0$ is usually a **zero vector**.

A **hidden state** is a memory vector that captures information from previous steps in a sequence. Updated at each step using current input and past state, it helps neural networks use context from earlier words to predict the next word in a sentence.

To deepen the network, we add a second RNN layer. The first layer's outputs, $\mathbf{h}_t$, become inputs to the second, whose outputs are the network's final outputs:



Layer 2
Layer 1
$\mathbf{h}_{1,2}$
$\mathbf{h}_{2,2}$
$\mathbf{h}_{0,2}$
$\mathbf{U}_{h,2}$
$\mathbf{h}_{1,2} \leftarrow \phi_{h,2}(\mathbf{W}_{h,2}\mathbf{h}_{1,1} + \mathbf{U}_{h,2}\mathbf{h}_{0,2} + \mathbf{b}_{h,2})$
$\mathbf{h}_{2,2} \leftarrow \phi_{h,2}(\mathbf{W}_{h,2}\mathbf{h}_{2,1} + \mathbf{U}_{h,2}\mathbf{h}_{1,2} + \mathbf{b}_{h,2})$
$\mathbf{b}_{h,2}$
$\mathbf{W}_{h,2}$
$\mathbf{h}_{1,1}$
$\mathbf{h}_{2,1}$
$\mathbf{h}_{0,1}$
$\mathbf{U}_{h,1}$
$\mathbf{h}_{1,1} \leftarrow \phi_{h,1}(\mathbf{W}_{h,1}\mathbf{x}_1 + \mathbf{U}_{h,1}\mathbf{h}_{0,1} + \mathbf{b}_{h,1})$
$\mathbf{h}_{2,1} \leftarrow \phi_{h,1}(\mathbf{W}_{h,1}\mathbf{x}_2 + \mathbf{U}_{h,1}\mathbf{h}_{1,1} + \mathbf{b}_{h,1})$
$\mathbf{b}_{h,1}$
$\mathbf{W}_{h,1}$
$\mathbf{x}_1$
$\mathbf{x}_2$
...

Figure 3.1: A two-layer Elman RNN. The first layer's outputs serve as inputs to the second layer.

3.2. Mini-Batch Gradient Descent

Before coding the RNN model, we need to discuss the shape of the input data. In Section 1.7, we used the entire dataset for each gradient descent step. Here, and for training all future models, we'll adopt **mini-batch gradient descent**, a widely used method for large models and datasets. Mini-batch gradient descent calculates derivatives over smaller data subsets, which speeds up learning and reduces memory usage.

With mini-batch gradient descent, the data shape is organized as (batch size, sequence length, embedding dimensionality). This structure divides the training set into fixed-size mini-batches, each containing sequences of embeddings with consistent lengths. (From this point on, “batch” and “mini-batch” will be used interchangeably.)

For example, if the batch size is 2, the sequence length is 4, and the embedding dimensionality is 3, the mini-batch can be represented as:

$$\text{batch}_1 = \begin{bmatrix} \text{seq}_{1,1} & \text{seq}_{1,2} & \text{seq}_{1,3} & \text{seq}_{1,4} \\ \text{seq}_{2,1} & \text{seq}_{2,2} & \text{seq}_{2,3} & \text{seq}_{2,4} \end{bmatrix}$$

Here, $\text{seq}_{i,j}$, for $i \in \{1,2\}$ and $j \in \{1, \ldots, 4\}$ is an embedding vector.

Let’s have the following embeddings for each sequence:

$$\text{seq}_1: \begin{bmatrix} [0.1,0.2,0.3] \\ [0.4,0.5,0.6] \\ [0.7,0.8,0.9] \\ [1.0,1.1,1.2] \end{bmatrix}$$

$$\text{seq}_2: \begin{bmatrix} [1.3,1.4,1.5] \\ [1.6,1.7,1.8] \\ [1.9,2.0,2.1] \\ [2.2,2.3,2.4] \end{bmatrix}$$

The mini-batch will look like this:

$$\text{batch}_1 = \begin{bmatrix} [0.1,0.2,0.3] & [0.4,0.5,0.6] & [0.7,0.8,0.9] & [1.0,1.1,1.2] \\ [1.3,1.4,1.5] & [1.6,1.7,1.8] & [1.9,2.0,2.1] & [2.2,2.3,2.4] \end{bmatrix}$$

During each step of gradient descent, we:

1. Select a mini-batch from the training set,
2. Pass it through the neural network,
3. Compute the loss,
4. Calculate gradients,
5. Update model parameters,
6. Repeat from step 1.

Mini-batch gradient descent often achieves faster **convergence** compared to using the entire training set per step. It efficiently handles large models and datasets by using modern hardware’s parallel processing capabilities. In PyTorch, models require the first dimension of the input data to be the batch dimension, even if there’s only one example in the batch.

3.3. Programming an RNN

Let’s implement an Elman RNN unit:

```
import torch
import torch.nn as nn

class ElmanRNNUnit(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.Uh = nn.Parameter(torch.randn(emb_dim, emb_dim)) ❶
        self.Wh = nn.Parameter(torch.randn(emb_dim, emb_dim)) ❷
        self.b = nn.Parameter(torch.zeros(emb_dim)) ❸

    def forward(self, x, h):
        return torch.tanh(x @ self.Wh + h @ self.Uh + self.b) ❹
```

In the constructor:

- Lines ❶ and ❷ initialize `self.Uh` and `self.Wh`, the weight matrices for the hidden state and input vector, with random values.
- Line ❸ sets `self.b`, the bias vector, to zero.

In the `forward` method, line ❹ handles the computation for each time step. It processes the current input `x` and the previous hidden state `h`, both shaped (`batch_size`, `emb_dim`), combines them with the weight matrices and bias, and applies the **tanh** activation. The output is the new hidden state, also of shape (`batch_size`, `emb_dim`).

The `@` character is the **matrix multiplication** operator in PyTorch. We use `x @ self.Wh` rather than `self.Wh @ x` because of the way PyTorch handles batch dimensions in matrix multiplication. When working with batched inputs, `x` has a shape of (`batch_size`, `emb_dim`), while `self.Wh` has a shape of (`emb_dim`, `emb_dim`). Remember from Section 1.6 that for two matrices to be multipliable, the number of columns in the left matrix must be the same as the number of rows in the right matrix. This is satisfied in `x @ self.Wh`.

Now, let's define the class `ElmanRNN`, which implements a two-layer Elman RNN using `ElmanRNNUnit` as its core building block:

```
class ElmanRNN(nn.Module):
    def __init__(self, emb_dim, num_layers):
        super().__init__()
        self.emb_dim = emb_dim
        self.num_layers = num_layers
        self.rnn_units = nn.ModuleList(
            [ElmanRNNUnit(emb_dim) for _ in range(num_layers)]
        ) ❶

    def forward(self, x):
        batch_size, seq_len, emb_dim = x.shape ❷
        h_prev = [
            torch.zeros(batch_size, emb_dim, device=x.device) ❸
            for _ in range(self.num_layers)
        ]
        outputs = []
        for t in range(seq_len): ❹
            input_t = x[:, t]
            for l, rnn_unit in enumerate(self.rnn_units):
                h_new = rnn_unit(input_t, h_prev[l])
                h_prev[l] = h_new    # Update hidden state
                input_t = h_new      # Input for next layer
            outputs.append(input_t)  # Collect outputs
        return torch.stack(outputs, dim=1) ❺
```

In line ❶ of the constructor, we initialize the RNN layers by creating a `ModuleList` containing `ElmanRNNUnit` instances—one per layer. Using `ModuleList` instead of a regular Python list ensures the parent module (`ElmanRNN`) properly registers all RNN unit parameters. This guarantees that calling `.parameters()` or `.to(device)` on the parent module includes parameters from all modules in the `ModuleList`.

In the `forward` method:

- Line ❷ extracts `batch_size`, `seq_len`, and `emb_dim` from the input tensor `x`.
- Line ❸ initializes the hidden states `h_prev` for all layers with zero tensors. Each hidden state in the list has the shape (`batch_size`, `emb_dim`).

> We store hidden states for each layer in a list instead of a multidimensional tensor because we need to modify them during processing. In-place modifications of tensors can disrupt PyTorch's automatic differentiation system, which might result in incorrect gradient calculations.

- Line ❹ iterates over time steps `t` in the input sequence. For each `t`:
    - Extract the input at time `t`: `input_t = x[:, t]`.
    - For each layer `l`:
        - Compute the new hidden state `h_new` from `input_t` and `h_prev[l]`.
        - Update the hidden state: `h_prev[l] = h_new` (updates in place).
        - Set `input_t = h_new` to pass to the next layer.
    - Append the output of the last layer: `outputs.append(input_t)`.
- Once all time steps are processed, line ❺ converts the `outputs` list into a tensor by stacking it along the time dimension. The resulting tensor has the shape (`batch_size`, `seq_len`, `emb_dim`).

3.4. RNN as a Language Model

An RNN-based language model uses `ElmanRNN` as its building block:

```
class RecurrentLanguageModel(nn.Module):
    def __init__(self, vocab_size, emb_dim, num_layers, pad_idx):
        super().__init__()
        self.embedding = nn.Embedding(
            vocab_size,
            emb_dim,
            padding_idx=pad_idx
        ) ❶
        self.rnn = ElmanRNN(emb_dim, num_layers)
        self.fc = nn.Linear(emb_dim, vocab_size)

    def forward(self, x):
        embeddings = self.embedding(x)
        rnn_output = self.rnn(embeddings)
        logits = self.fc(rnn_output)
        return logits
```

The `RecurrentLanguageModel` class integrates three components: an embedding layer, the `ElmanRNN` defined earlier, and a final linear layer.

In the constructor, line ❶ defines the embedding layer. This layer transforms input token indices into dense vectors. The `padding_idx` parameter ensures that padding tokens are represented by zero vectors. (We'll cover the embedding layer in the next section.)

Next, we initialize the custom `ElmanRNN`, specifying the embedding dimensionality and the number of layers. Finally, we add a fully connected layer, which converts the RNN's output into vocabulary-sized logits for each token in the sequence.

In the `forward` method:

- We pass the input `x` through the embedding layer. Input `x` has shape (`batch_size`, `seq_len`), and the output `embeddings` have shape (`batch_size`, `seq_len`, `emb_dim`).
- We then pass the embedded input through our `ElmanRNN`, obtaining `rnn_output` with shape (`batch_size`, `seq_len`, `emb_dim`).
- Finally, we apply the fully connected layer to the RNN output, producing logits for each token in the vocabulary at each position in the sequence. The output logits have shape (`batch_size`, `seq_len`, `vocab_size`).

3.5. Embedding Layer

An **embedding layer**, implemented as `nn.Embedding` in PyTorch, maps token indices from a vocabulary to dense, fixed-size vectors. It acts as a learnable lookup table, where each token is assigned a unique embedding vector. During training, these vectors are adjusted to capture meaningful numerical representations of the tokens.

Let's see how an embedding layer works. Imagine a vocabulary with five tokens, indexed from 0 to 4. We want each token to have a 3D embedding vector. To begin, we create an embedding layer:

```
import torch
import torch.nn as nn

vocab_size = 5  # Number of unique tokens
emb_dim = 3     # Size of each embedding vector
emb_layer = nn.Embedding(vocab_size, emb_dim)
```

The embedding layer initializes the embedding matrix **E** with random values. In this case, the matrix has 5 rows (one for each token) and 3 columns (the embedding dimensionality):

$$\mathbf{E} = \begin{bmatrix} 0.2 & -0.4 & 0.1 \\ -0.3 & 0.8 & -0.5 \\ 0.7 & 0.1 & -0.2 \\ -0.6 & 0.5 & 0.4 \\ 0.9 & -0.7 & 0.3 \end{bmatrix}$$

Each row in **E** represents the embedding vector for a specific token in the vocabulary.

Now, let's input a sequence of token indices:

```
token_indices = torch.tensor([0, 2, 4])
```

The embedding layer retrieves the rows of **E** corresponding to the input indices:

$$\text{Embeddings} = \begin{bmatrix} 0.2 & -0.4 & 0.1 \\ 0.7 & 0.1 & -0.2 \\ 0.9 & -0.7 & 0.3 \end{bmatrix}$$

This output is a matrix whose number of rows equals the input sequence length and whose number of columns equals the embedding dimensionality:

```
embeddings = embedding_layer(token_indices)
print(embeddings)
```

The output might look like this:

```
tensor([[ 0.2, -0.4,  0.1],
        [ 0.7,  0.1, -0.2],
        [ 0.9, -0.7,  0.3]])
```

The embedding layer can manage padding tokens as well. Padding ensures sequences in a mini-batch have the same length. To prevent the model from updating embeddings for padding tokens during training, the layer maps them to a zero vector that remains unchanged. For example, we can define the padding index as follows:

```
emb_layer = nn.Embedding(vocab_size, emb_dim, padding_idx=0)
```

With this configuration, the embedding for token 0 (padding token) is always $[0,0,0]^\top$.

Given the input:

```
token_indices = torch.tensor([0, 2, 4])
embeddings = emb_layer(token_indices)
print(embeddings)
```

The result would be:

```
tensor([[ 0.0,  0.0,  0.0],  # Padding token
        [ 0.7,  0.1, -0.2],  # Token 2 embedding
        [ 0.9, -0.7,  0.3]]) # Token 4 embedding
```

With modern language models, vocabularies often include hundreds of thousands of tokens, and embedding dimensions are typically several thousands. This makes the embedding matrix a significant part of the model, sometimes containing up to 2 billion parameters.

3.6. Training an RNN Language Model

Start by importing libraries and defining utility functions:

```
import torch, torch.nn as nn

def set_seed(seed):
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed) ❶
    torch.backends.cudnn.deterministic = True ❷
    torch.backends.cudnn.benchmark = False ❸
```

The `set_seed` function enforces **reproducibility** by setting the Python random seed, the PyTorch CPU seed, and, in line ❶, the CUDA seed for all GPUs (Graphics Processing Units). CUDA is NVIDIA's parallel computing platform and API that enables significant performance improvements in computing by leveraging the power of GPUs. Using `torch.cuda.manual_seed_all` ensures consistent GPU-based random behavior, while lines ❷ and ❸ disable CUDA's auto-tuner and enforce deterministic algorithms, guaranteeing identical results across different GPU models.

With the model class ready, we'll train our neural language model. First, we install the `transformers` package—an open-source library providing APIs and tools to easily download, train and use pretrained models from the **Hugging Face Hub**:

```
$ pip3 install transformers
```

The package offers a Python API for training that works with both **PyTorch** and **TensorFlow**. For now, we only need it to get a tokenizer.

Now we import `transformers`, set the tokenizer, define the hyperparameter values, prepare the data, and instantiate the model, loss function, and optimizer objects:

```
from transformers import AutoTokenizer

device = torch.device("cuda" if torch.cuda.is_available() else "cpu") ❶
tokenizer = AutoTokenizer.from_pretrained(
    "microsoft/Phi-3.5-mini-instruct"
) ❷
vocab_size = len(tokenizer) ❸

emb_dim, num_layers, batch_size, learning_rate, num_epochs = get_hyperparamet
ers()

data_url = "https://www.thelmbook.com/data/news"
train_loader, test_loader = download_and_prepare_data(
    data_url, batch_size, tokenizer) ❹

model = RecurrentLanguageModel(
    vocab_size, emb_dim, num_layers, tokenizer.pad_token_id
)
```

```
initialize_weights(model) ❺
model.to(device)

criterion = nn.CrossEntropyLoss(ignore_index=tokenizer.pad_token_id) ❻
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
```

Line ❶ detects a CUDA device if it's available. Otherwise, it defaults to CPU.

> CUDA is not the only GPU acceleration framework available for training neural networks—PyTorch also provides native support for checking availability of MPS (Apple Metal) through its `is_available()` method. In this book, though, we will use CUDA as it remains the most widely used platform for machine learning acceleration.

Most models on the Hugging Face Hub include the tokenizer that was used to train them. Line ❷ initializes the **Phi 3.5 mini** tokenizer. It was trained on a large text corpus using the **byte-pair encoding** algorithm and has a vocabulary size of 32,064.

Line ❸ retrieves the tokenizer's vocabulary size. Line ❹ downloads and prepares the dataset—a collection of news sentences from online articles—tokenizing them and creating `DataLoader` objects. We'll explore `DataLoader` shortly. For now, think of them as iterators over batches.

Line ❺ initializes the model parameters. Initial parameter values can greatly influence the training process. They can affect how quickly training progresses and the final loss value. Certain initialization techniques, like **Xavier initialization**, have shown good results in practice. The `initialize_weights` function, implementing this method, is defined in the notebook.

Line ❻ creates the loss function with the `ignore_index` parameter. This ensures the loss is not calculated for padding tokens.

Now, let's look at the training loop:

```
for epoch in range(num_epochs): ❶
    model.train() ❷
    for batch in train_loader: ❸
        input_seq, target_seq = batch
        input_seq = input_seq.to(device) ❹
        target_seq = target_seq.to(device) ❺
        batch_size_current, seq_len = input_seq.shape ❻
        optimizer.zero_grad()
        output = model(input_seq)
        output = output.reshape(batch_size_current * seq_len, vocab_size) ❼
        target = target_seq.reshape(batch_size_current * seq_len) ❽
        loss = criterion(output, target) ❾
        loss.backward()
        optimizer.step()
```

Line ➊ iterates over epochs. An **epoch** is a single pass through the entire dataset. Training for multiple epochs can improve the model, especially with limited training data. The number of epochs is a **hyperparameter** that you adjust based on the model's performance on the test set.

Line ➋ calls `model.train()` at the start of each epoch to set the model in training mode. This is important for models that have layers behaving differently during training vs. **evaluation**.

> Although our RNN model doesn't use such layers, calling `model.train()` ensures the model is properly configured for training. This avoids unexpected behavior and keeps consistency, especially if future changes add layers dependent on the mode.

Line ➌ iterates over batches. Each batch is a tuple: one tensor contains input sequences, and the other contains target sequences. Lines ➍ and ➎ move these tensors to the same device as the model. If the model and data are on different devices, PyTorch raises an error.

Line ➏ retrieves the batch size and sequence length from `input_seq` (`target_seq` has the same shape). These dimensions are needed to reshape the model's output tensor (`batch_size_current`, `seq_len`, `vocab_size`) and target tensor (`batch_size_current`, `seq_len`) into compatible shapes for the **cross-entropy** loss function. In line ➐, the output is reshaped to (`batch_size_current * seq_len`, `vocab_size`), and in line ➑, the target is flattened to `batch_size_current * seq_len`, allowing the loss calculation in line ➒ to process all tokens in the batch simultaneously and return the average loss per token.

This concludes the training loop implementation. The full RNN language model training implementation is in the thelmbook.com/nb/3.1 notebook. Now, let's examine the DataLoader and Dataset classes that make this batch processing possible.

3.7. Dataset and DataLoader

As mentioned earlier, the `download_and_prepare_data` function returns two *loader* objects: `train_loader` and `test_loader`. I asked you to think of them as iterators over batches of data. But what are they, exactly?

These classes were designed to manage data efficiently during training. While this book doesn't focus on data loading and manipulation, a brief explanation is important for clarity.

The `Dataset` class serves as an interface to your actual data source. By implementing its `__len__` method, you can get the size of the dataset. By defining `__getitem__`, you can access individual examples. These examples can come from many "physical" sources: files, databases, or even data generated on the fly.

Let's look at an example. Assume we have a **JSONL** file called `data.jsonl`, where each line is a **JSON** object containing two input features and a label. Here's how a couple of lines might look:

```
{"feature1": 1.0, "feature2": 2.0, "label": 3.0}
{"feature1": 4.0, "feature2": 5.0, "label": 9.0}
...
```

Here's how you can create a custom `Dataset` to read this file:

```
import json
import torch
from torch.utils.data import Dataset

class JSONDataset(Dataset):
    def __init__(self, file_path):
        self.data = []
        with open(file_path, 'r') as f:
            for line in f:
                item = json.loads(line)
                features = [item['feature1'], item['feature2']]
                label = item['label']
                self.data.append((features, label))

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        features, label = self.data[idx]
        features = torch.tensor(features, dtype=torch.float32)
        label = torch.tensor(label, dtype=torch.long)
        return features, label
```

In this example:

- `__init__` reads the file and stores the data in memory,
- `__len__` returns the total number of examples,
- `__getitem__` retrieves a single example and converts it to tensors.

We can access individual examples like this:

```
dataset = JSONDataset('data.jsonl')
features, label = dataset[0]
```

A `DataLoader` is used with a `Dataset` to manage tasks like batching, shuffling, and loading data in parallel. For example:

```
from torch.utils.data import DataLoader

dataset = JSONLDataset('data.jsonl') ❶

data_loader = DataLoader(
    dataset,
    batch_size=32, # Number of examples per batch
    shuffle=True,  # Shuffle data at every epoch
    num_workers=0  # Number of subprocesses for data loading
) ❷

num_epochs = 5
```

```
for epoch in range(num_epochs):
    for batch_features, batch_labels in data_loader: ❸
        print(f"Batch features shape: {batch_features.shape}")
        print(f"Batch labels shape: {batch_labels.shape}")
        # Feed batch_features and batch_labels into your model
```

Line ❶ creates a `Dataset` instance. Line ❷ then wraps the dataset in a `DataLoader`. Finally, line ❸ iterates over the `DataLoader` for five epochs. With `shuffle=True`, the data is shuffled before batching in each epoch. This prevents the model from learning the order of the training data.

With `num_workers=0`, data loading happens in the main process. This simple setup may not be the most efficient, especially for large datasets. Using a positive value for `num_workers` makes PyTorch spawn that many worker processes, enabling parallel data loading. This can significantly speed up training by preventing data loading from becoming a bottleneck.

Output:

```
Batch features shape: torch.Size([32, 2])
Batch labels shape: torch.Size([32])
```

By using a well-designed `Dataset` with a `DataLoader`, you can scale your training pipeline to handle large datasets, optimize data loading with parallel workers, and experiment with different batching strategies. This approach streamlines the training process, letting you concentrate on model design and optimization.

3.8. Training Data and Loss Computation

When studying neural language models, a key aspect is understanding the structure of a training example. The text corpus is split into overlapping input and target sequences. Each input sequence aligns with a target sequence shifted by one token. This setup trains the model to predict the next word at each position in the sequence.

For instance, take the sentence *"We train a recurrent neural network as a language model."* After tokenizing it with the Phi 3.5 mini tokenizer, we get:

```
["_We", "_train", "_a", "_rec", "urrent", "_neural", "_network", "_as", "_a",
"_language", "_model", "."]
```

To create one training example, we convert the sentence into input and target sequences by shifting tokens forward by one position:

```
Input: ["_We", "_train", "_a", "_rec", "urrent", "_neural", "_network", "_as"
, "_a", "_language", "_model"]
Target: ["_train", "_a", "_rec", "urrent", "_neural", "_network", "_as", "_a"
, "_language", "_model", "."]
```

A training example doesn't need to be a complete sentence. Modern language models process sequences up to their **context window** length—the maximum tokens (like 8192) they can handle at once. The window limits how far apart the model can connect relationships in text. Training splits text into window-sized chunks, each target sequence shifted one token forward from its input.

During training, the RNN processes one token at a time, updating its hidden states layer by layer. At each step, it generates logits aimed at predicting the next token in the sequence. Each logit corresponds to a vocabulary token and is converted into probabilities using **softmax.** These probabilities are then used to compute the loss.

Each training example results in multiple predictions and losses. For example, the model first processes "_We" and tries to predict "_train" by assigning probabilities to all vocabulary tokens. The loss is computed using the probability of "_train," as defined in Equation 2.1. Next, the model processes "_train" to predict "_a," generating another loss. This continues for every token in the input sequence. For the above example, the model makes 11 predictions and calculates 11 losses.

The losses are averaged across the tokens in a training example and all examples in the batch. The average loss expression is then used in backpropagation to update the model's parameters.

Let's break down the loss calculation for each position with some made-up numbers:

- **Position 1**:
  - Target token: "_train"
  - Logit for "_train": -0.5
  - After applying softmax to the logits, suppose the probability of "_train" is 0.1
  - Contribution to the total loss by Equation 2.1 is $-\log(0.1) = 2.30$
- **Position 2**:
  - Target token: "_a"
  - Logit for "_a": 3.2
  - After softmax, the probability for "_a": 0.05
  - Contribution to loss: $-\log(0.05) = 2.99$
- **Position 3**:
  - The probability for "_rec": 0.02
  - Contribution to loss: $-\log(0.02) = 3.91$
- **Position 4**:
  - The probability for "urrent": 0.34
  - Contribution to loss: $-\log(0.34) = 1.08$

We continue until calculating the loss contribution for the final token, the period:

- **Position 11**:
  - Target token: "."
  - Logit for ".": -1.2
  - After softmax, the probability for ".": 0.11
  - Contribution to loss: $-\log(0.11) = 2.21$

The final loss is calculated by taking the average of these values:

$$\frac{(2.30 + 2.99 + 3.91 + 1.08 + \cdots + 2.21)}{11} = 2.11 \text{ (hypothetically)}$$

During training, the objective is to minimize this loss. This involves improving the model so that it assigns higher probabilities to the correct target tokens at each position.

The full code for training the RNN-based language model can be found in thelmbook.com/nb/3.1. I used the following hyperparameter values: `emb_dim = 128`, `num_layers = 2`, `batch_size = 128`, `learning_rate = 0.001`, and `num_epochs = 1`.

Here are three continuations for the prompt *"The President"* generated at later training steps:

```
The President refused to comment on the best news in the five on BBC .
The President has been a `` very serious '' and `` unacceptable '' .
The President 's office is not the first time to be able to take the lead .
```

> When Elman introduced RNNs in 1990, his experiments used sequences averaging 3.92 words, limited by the hardware of the time. By 2014, advances in computing and improved activation functions made it possible to train RNNs on sequences hundreds of words long, turning them from an academic idea into a practical tool.

At training start, our model produced nearly random tokens, but gradually improved, reaching a perplexity of 72.41—better than the count-based model's 299.06 but far behind GPT-2's 20 and modern LLMs' sub-5 scores.

Three key factors explain this performance gap:

1. The model is small, with just 8,292,619 parameters, mostly in the embedding layer.
2. The context size we used was relatively short—30 tokens.
3. The Elman RNN's hidden state gradually "forgets" information from earlier tokens.

Long short-term memory (**LSTM**) networks improved upon RNNs but still struggled with very long sequences. Transformers later superseded both architectures, becoming dominant in natural language processing by 2023 through better handling of long contexts and improved parallel computation enabling larger models.

> Interest in RNNs was reignited in 2024 with the invention of the **minLSTM** and **xLSTM** architectures, which achieve performance comparable to Transformer-based models. This resurgence reflects a broader trend in AI research: no model type is ever permanently obsolete. Researchers often revisit and refine older ideas, adapting them to address modern challenges and leverage current hardware capabilities.

With this, we've completed our study of recurrent neural networks and their applications in language modeling. In the remainder of the book, we'll examine transformer neural networks and language modeling based on them. We'll investigate their approach to tasks like question answering, document classification, and other practical applications.