Summary

Chapter 3 of the sources, titled "Recurrent Neural Network," introduces recurrent neural networks (RNNs) as a foundational architecture for sequential data processing. While Transformers have become dominant, understanding RNNs is crucial as they lay the groundwork for sequential processing concepts that make Transformer mathematics more intuitive.

Here's a detailed summary of Chapter 3:

- **3.1 Elman RNN**
  - **Recurrent Neural Networks (RNNs)** are neural networks designed specifically for sequential data. Unlike feedforward neural networks (FNNs), RNNs incorporate **loops in their connections**, which allows information to be carried over from one step in a sequence to the next. This makes them well-suited for tasks involving time series analysis, natural language processing, and other sequential data problems.
  - The chapter uses an example document, "Learning from text is cool," to illustrate the sequential nature. Each word in the document is represented by an **embedding vector**, forming a matrix where rows represent word embeddings and the order of words is preserved. The dimensions of this matrix are (sequence length, embedding dimensionality). Shorter documents are often padded with dummy embeddings, typically zero vectors.
  - 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.
  - At each time step t, the current input embedding $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$ are combined. This combination involves multiplying them with **trainable weight matrices** $\mathbf{W}_h$ and $\mathbf{U}_h$, and adding a **bias vector** $\mathbf{b}_h$, to produce the updated hidden state $\mathbf{h}_t$.
  - Unlike multilayer perceptron (MLP) units, which output scalars, an RNN unit outputs vectors and functions as an entire layer. The **initial hidden state** $\mathbf{h}_0$ is typically a zero vector.
  - A **hidden state** serves as a **memory vector** that captures information from previous steps in a sequence, allowing the neural network to use context from earlier words for predictions.
  - To create a deeper network, multiple RNN layers can be stacked, where the outputs of the first layer ($\mathbf{h}_t$) become inputs to the second, and so on.

- **3.2 Mini-Batch Gradient Descent**
  - For training large models and datasets, **mini-batch gradient descent** is a widely used method. It calculates derivatives over smaller subsets of data, which **speeds up learning** and **reduces memory usage**.
  - In mini-batch gradient descent, the data is organized with the shape (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.
  - The training process involves selecting a mini-batch, passing it through the network, computing the loss, calculating gradients, updating parameters, and repeating. This method often leads to faster convergence and efficiently utilizes parallel processing capabilities of modern hardware. PyTorch models require the first dimension of input data to be the batch dimension.
- **3.3 Programming an RNN**
  - The `ElmanRNNUnit` class implements a single Elman RNN unit. Its constructor initializes trainable weight matrices `self.Uh` and `self.Wh` with random values, and the bias vector `self.b` to zero.
  - The `forward` method computes the new hidden state using the current input `x` and the previous hidden state `h`, both of shape (`batch_size`, `emb_dim`), applying the **tanh activation function**: `torch.tanh(x @ self.Wh + h @ self.Uh + self.b)`. The `@` symbol denotes matrix multiplication in PyTorch.
  - The `ElmanRNN` class builds a two-layer Elman RNN by creating a `ModuleList` of `ElmanRNNUnit` instances. This ensures proper registration of parameters.
  - Its `forward` method extracts `batch_size`, `seq_len`, and `emb_dim` from the input tensor `x`. It initializes hidden states for all layers with zero tensors. It then iterates over time steps (`t`) and layers (`l`), computing and updating hidden states:
    - `h_new = rnn_unit(input_t, h_prev[l])`
    - `h_prev[l] = h_new`
    - `input_t = h_new`
  - Finally, it stacks the collected outputs into a tensor of shape (`batch_size`, `seq_len`, `emb_dim`).
- **3.4 RNN as a Language Model**
  - The `RecurrentLanguageModel` class integrates three main components: an **embedding layer**, the `ElmanRNN`, and a final **linear layer**.
  - The embedding layer (`nn.Embedding`) transforms input token indices into dense vectors, with `padding_idx` ensuring padding tokens are mapped to zero vectors.

    - The `ElmanRNN` processes these embeddings.
    - A fully connected layer (`nn.Linear`) converts the RNN's output into vocabulary-sized logits for each token in the sequence.
    - The `forward` method passes input `x` through the embedding layer, then through the `ElmanRNN` (producing `rnn_output` of shape (`batch_size`, `seq_len`, `emb_dim`)), and finally applies the fully connected layer to get `logits` of shape (`batch_size`, `seq_len`, `vocab_size`).
- **3.5 Embedding Layer**
    - An **embedding layer** (e.g., `nn.Embedding` in PyTorch) maps token indices from a vocabulary to dense, fixed-size vectors. It functions as a **learnable lookup table**, where each token is assigned a unique embedding vector adjusted during training to capture meaningful numerical representations.
    - The layer creates a matrix where each row represents the embedding vector for a specific token. When given `token_indices` (e.g., `torch.tensor()`), it retrieves the corresponding rows.
    - It also manages **padding tokens** by mapping them to a zero vector that remains unchanged during training. For instance, `padding_idx=0` makes the embedding for token 0 always ``.
    - Modern language models can have vocabularies with hundreds of thousands of tokens and embedding dimensions in the thousands, making the embedding matrix a significant part of the model.
- **3.6 Training an RNN Language Model**
    - **Reproducibility** is enforced by using `set_seed` to set random seeds for Python, PyTorch CPU, and CUDA (for GPUs), and disabling CUDA's auto-tuner and enforcing deterministic algorithms.
    - The training setup involves:
        - Detecting available CUDA devices.
        - Initializing a tokenizer (e.g., `AutoTokenizer.from_pretrained("microsoft/Phi-3.5-mini-instruct")`) and setting its padding token.
        - Defining hyperparameters (e.g., `emb_dim`, `num_layers`, `batch_size`, `learning_rate`, `num_epochs`).
        - Downloading and preparing the dataset (`train_loader`, `test_loader`).
        - Instantiating `RecurrentLanguageModel` and initializing its weights (e.g., Xavier initialization).
        - Moving the model to the appropriate device (`model.to(device)`).
        - Defining the loss function, typically `nn.CrossEntropyLoss`, with `ignore_index` set to `tokenizer.pad_token_id` to prevent loss calculation for padding tokens.

    - Setting up the optimizer, such as `torch.optim.AdamW`.
- The **training loop** proceeds as follows:
    - Iterate over `num_epochs`.
    - Set the model to training mode (`model.train()`).
    - Iterate over `batch` from `train_loader`.
    - Move `input_seq` and `target_seq` to the device (`.to(device)`).
    - Zero gradients (`optimizer.zero_grad()`).
    - Perform a **forward pass** (`output = model(input_seq)`).
    - Reshape the model's output (`output.reshape(batch_size_current * seq_len, vocab_size)`) and flatten the target (`target.reshape(batch_size_current * seq_len)`) to be compatible with the loss function.
    - Calculate the loss (`loss = criterion(output, target)`).
    - Perform **backpropagation** (`loss.backward()`) to compute gradients.
    - Update model parameters (`optimizer.step()`).
- **3.7 Dataset and DataLoader**
    - The `Dataset` class serves as an **interface to the actual data source**, allowing access to individual examples and providing the dataset size.
    - The `DataLoader` class is used with a `Dataset` to efficiently manage tasks such as **batching, shuffling, and parallel data loading**. It iterates over the dataset in batches.
    - Setting `shuffle=True` shuffles data before batching in each epoch, preventing the model from learning the order of training data. `num_workers` enables parallel data loading.
- **3.8 Training Data and Loss Computation**
    - For neural language models, the text corpus is split into **overlapping input and target sequences**, where the target sequence is shifted forward by one token relative to the input. This setup trains the model to predict the next token at each position.
    - During training, the RNN processes one token at a time, updating its hidden states layer by layer. At each step, it generates logits to predict the next token, which are converted to probabilities using softmax.
    - The **loss is computed for each position** (e.g., using Equation 2.1 for cross-entropy loss: $\text{loss}(\hat{\mathbf{y}}, \mathbf{y}) = -\log(\hat{y}_{(c)})$) and then averaged across all tokens in a training example and all examples in the batch. This average loss is then used in backpropagation to update parameters.
    - Historically, early RNNs were limited to short sequences due to hardware. By 2014, advances allowed training on hundreds of words.

- The Elman RNN model, even with comparable parameters to a simple Transformer, achieved a perplexity of 72.41, which is better than the count-based model's 299.06 but still significantly higher than modern LLMs. This performance gap is attributed to the Elman RNN's tendency for its hidden state to "forget" information from earlier tokens, the small model size, and a relatively short context size (e.g., 30 tokens).
- Long Short-Term Memory (LSTM) networks improved upon RNNs, but still faced challenges with very long sequences. Transformers later surpassed both architectures due to better handling of long contexts and improved parallel computation. However, newer RNN architectures like minLSTM and xLSTM are showing competitive performance to Transformers, suggesting that no model type is permanently obsolete.

Formulas

Here is an overview of the key mathematical expressions used and explained in the context of Chapter 3:

- **1. Elman RNN Hidden State Update** The core of the Elman RNN is how it updates its **hidden state** $\mathbf{h}_t$ at each time step t. This hidden state acts as a **memory vector** that encapsulates information from previous steps in the sequence. The formula for updating the hidden state combines the current input embedding $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$, applying trainable weights and a bias, then passing the result through an activation function. The formula for the updated hidden state is:

$$\mathbf{h}_t = \tanh(\mathbf{x}_t \mathbf{W}_h + \mathbf{h}_{t-1} \mathbf{U}_h + \mathbf{b}_h)$$

  - $\mathbf{h}_t$: The **new hidden state vector** at time step t. This is the output of the RNN unit for the current step.
  - $\mathbf{x}_t$: The **current input embedding vector** at time step t. This represents the current word or token being processed.
  - $\mathbf{h}_{t-1}$: The **hidden state vector from the previous time step** $(t-1)$. For the first step ($t=0$), the initial hidden state $\mathbf{h}_0$ is typically a zero vector.
  - $\mathbf{W}_h$: A **trainable weight matrix** that is multiplied with the current input $\mathbf{x}_t$. In the code, this is `self.Wh`.
  - $\mathbf{U}_h$: A **trainable weight matrix** that is multiplied with the previous hidden state $\mathbf{h}_{t-1}$. In the code, this is `self.Uh`.
  - $\mathbf{b}_h$: A **trainable bias vector** that is added to the weighted sum. In the code, this is `self.b`.
  - $\tanh$: The **hyperbolic tangent activation function**. This non-linear function is applied element-wise to the result of the linear combination, outputting values

between -1 and 1. Its purpose is to introduce non-linearity, enabling the model to learn complex patterns.

- **2. RNN Language Model Output (Logits)** After the RNN processes the sequence of embeddings and produces its `rnn_output` (the final hidden states for each position in the sequence, typically from the last RNN layer), a **linear layer** (`nn.Linear`) is applied. This layer transforms the RNN's output into **logits**, which are raw scores, for each token in the vocabulary. These logits represent the unnormalized probabilities that each word in the vocabulary is the next token in the sequence. The formula for generating logits is implicitly a linear transformation:

$$\mathbf{L} = RNN_{output}\mathbf{W}_{fc} + \mathbf{b}_{fc}$$

  - $\mathbf{L}$: The **output vector of logits** for each position in the sequence, with dimensions (`batch_size`, `seq_len`, `vocab_size`).
  - RNN_{output}: The **output from the final layer of the Elman RNN**, with dimensions (`batch_size`, `seq_len`, `emb_dim`).
  - $\mathbf{W}_{fc}$: The **trainable weight matrix** of the final fully connected linear layer, with dimensions (`emb_dim`, `vocab_size`).
  - $\mathbf{b}_{fc}$: The **trainable bias vector** of the final fully connected linear layer.
- **3. Softmax Activation Function** Although primarily defined and detailed in Chapter 2, the **softmax function** is crucial for language models like the RNN because it converts the raw logits (output of the final linear layer) into a **discrete probability distribution** over the vocabulary. This is necessary for predicting the next token and for calculating the cross-entropy loss. The formula for softmax is:

$$\text{softmax}(\mathbf{z}, k) = \frac{e^{z_{(k)}}}{\sum_{j=1}^{D} e^{z_{(j)}}}$$

  - $\text{softmax}(\mathbf{z}, k)$: The **probability** assigned to token k from a vector of logits $\mathbf{z}$.
  - $\mathbf{z}$: A **vector of logits** (raw outputs from the neural network).
  - $z_{(k)}$: The **logit for the k-th token** in the vocabulary.
  - D: The **dimensionality of the logit vector**, which is equivalent to the size of the vocabulary.
  - e: **Euler's number** (approximately 2.72).
  - $\sum_{j=1}^{D} e^{z_{(j)}}$: The **sum of exponentials of all logits** in the vector, which normalizes the probabilities so they sum to 1. **Explanation**: Softmax ensures that each token receives a non-negative probability, and all probabilities for the next token in the vocabulary sum to 1.
- **4. Cross-Entropy Loss Function (for a single example)** The **cross-entropy loss** is the standard loss function used to train language models, including the RNN, as it measures how well the predicted probabilities match the true

distribution. In Chapter 3, the training process explicitly states that loss is computed using "Equation 2.1". For a single training example where the target label is one-hot encoded (meaning only the correct class has a value of 1, and others are 0), the simplified cross-entropy loss is:

$$\text{loss}(\hat{\mathbf{y}}, \mathbf{y}) = -\log(\hat{y}_{(c)})$$

- $\text{loss}(\hat{\mathbf{y}}, \mathbf{y})$: The **cross-entropy loss** for a given example, where $\hat{\mathbf{y}}$ is the vector of predicted probabilities and $\mathbf{y}$ is the one-hot encoded true label.
- $\hat{y}_{(c)}$: The **predicted probability** assigned by the model to the **correct class** c.
- $\log$: The **natural logarithm**. **Explanation**: This formula shows that the loss is the negative logarithm of the probability assigned to the true (correct) token. A lower loss means the model assigned a higher probability to the correct token, indicating a better prediction. If the model predicts the correct token with a probability close to 1, the loss approaches 0. If it assigns a probability close to 0 to the correct token, the loss approaches infinity, penalizing incorrect predictions heavily.

- **5. Average Cross-Entropy Loss (for a batch of N examples)** During training, especially with **mini-batch gradient descent**, the loss is calculated across multiple predictions within a sequence and then averaged over all examples in a batch. This average loss is then used during backpropagation to update the model's parameters. The formula for the average loss over N examples (or tokens in a batch) is:

$$\text{loss} = -\frac{1}{N}\sum_{i=1}^{N}\log(\hat{y}_{i(c_i)})$$

  - loss: The **average loss** computed across N examples in a batch.
  - N: The **total number of examples** (or tokens) in the batch.
  - i: An **index** iterating through each example/token in the batch.
  - $\hat{y}_{i(c_i)}$: The **predicted probability** for the correct class c_i of the i-th example/token.
  - $\sum$: The **summation operator**, summing up the negative log-probabilities for all examples in the batch. **Explanation**: By averaging the loss, each example contributes equally to the overall loss, regardless of the total number of examples, providing a stable target for the optimizer during training.