From RNNs to LLMs: A Concise Overview of Deep Learning in NLP

Anjan Goswami

1 Introduction

Natural Language Processing (NLP) has witnessed profound advancements in recent years, fundamentally reshaping how machines comprehend and interact with human language. The primary aim of this tutorial is to offer readers an insight into these pivotal developments, particularly emphasizing the revolutionary strides made in the realm of language modeling. By journeying through this tutorial, one would garner a holistic understanding of the cutting-edge techniques and architectures that are currently pushing the boundaries of what machines can achieve with human languages.

2 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are specifically designed to tackle sequence modeling by preserving a memory of previous inputs. Unlike traditional n-gram-based language models, which act as finite response systems and operate as n-order Markov models, RNNs function as infinite response systems. This means that, theoretically, they can produce an output of indefinite length from a limited input. The mechanics of RNNs involve a recurrence equation, which at every time step t, combines the current input with the hidden state vector from the previous time step $(t-1)$. To kick-start this recursive process, an initial state is defined. The subsequent mathematical representation delves into the specifics of this mechanism:

2.1 Mathematical Definition

The RNN can be mathematically described as the following recursive equation:

$$\mathbf{h}_t = \sigma(\mathbf{W} \cdot \mathbf{h}_{t-1} + \mathbf{U} \cdot \mathbf{x}_t + \mathbf{b}_h)$$
$$\mathbf{o}_t = \mathbf{V} \cdot \mathbf{h}_t + \mathbf{b}_o$$

The base case is:

$$\mathbf{h}_0 = \text{initialization (often zeros)}$$

where:

- $\mathbf{x}_t \in \mathbb{R}^{d_x \times 1}$: Input vector at time step t
- $\mathbf{h}_t \in \mathbb{R}^{d_h \times 1}$: Hidden state vector at time step t
- $\mathbf{o}_t \in \mathbb{R}^{d_o \times 1}$: Output vector at time step t
- $\mathbf{U} \in \mathbb{R}^{d_h \times d_x}$: Weight matrix for input
- $\mathbf{W} \in \mathbb{R}^{d_h \times d_h}$: Weight matrix for previous hidden state
- $\mathbf{V} \in \mathbb{R}^{d_o \times d_h}$: Weight matrix for output
- $\mathbf{b}_h \in \mathbb{R}^{d_h \times 1}$: Bias vector for hidden state
- $\mathbf{b}_o \in \mathbb{R}^{d_o \times 1}$: Bias vector for output
- σ: activation function (e.g., sigmoid or tanh).

3 Limitations of RNNs

3.1 Vanishing and Exploding Gradients

In the context of RNNs, when computing gradients through backpropagation through time (BPTT), the gradients can either vanish or explode. This is because the gradient is the product of several terms. Let's consider an RNN with the activation function f. The gradient of the loss L with respect to a particular weight W for a time step t is given by:

$$\frac{\partial L}{\partial W} \propto \prod_{k=1}^{t} \frac{\partial f(z_k)}{\partial z_k}$$

When using an activation function like the sigmoid or tanh, the derivatives can be very small, causing the gradient to vanish for larger values of t. On the other hand, if the product becomes very large, the gradient can explode.

3.2 Limited Memory Capacity

The hidden state h_t of an RNN at time step t is given by:

$$h_t = f(W_h h_{t-1} + W_x x_t + b_h)$$

As t increases, the influence of initial states and inputs tends to diminish due to multiplicative interactions. This can make it difficult for RNNs to retain long-term dependencies.

3.3 Parallelization

Due to sequential dependencies, computing h_t relies on the completed computation of h_{t-1}, prohibiting parallelization.

4 Mitigations by LSTM and GRU

4.1 LSTM

LSTM introduces gating mechanisms that allow for controlled memory:

$$
\begin{aligned}
f_t &= \sigma(W_f[h_{t-1}, x_t] + b_f) && \text{– Forget Gate} \\
i_t &= \sigma(W_i[h_{t-1}, x_t] + b_i) && \text{– Input Gate} \\
\tilde{C}_t &= \tanh(W_C[h_{t-1}, x_t] + b_C) \\
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\
o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o) && \text{– Output Gate} \\
h_t &= o_t \odot \tanh(C_t)
\end{aligned}
$$

Through element-wise multiplication and additive interactions, LSTMs can effectively retain or forget information. The cell state C_t can carry long-term information due to these additive updates.

4.2 GRU

GRUs simplify the gating mechanism but still provide a mechanism to capture long-term dependencies:

$$
\begin{aligned}
z_t &= \sigma(W_z[h_{t-1}, x_t] + b_z) && \text{– Update Gate} \\
r_t &= \sigma(W_r[h_{t-1}, x_t] + b_r) && \text{– Reset Gate} \\
\tilde{h}_t &= \tanh(W[r_t \odot h_{t-1}, x_t] + b) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t
\end{aligned}
$$

The update and reset gates in GRU help in capturing dependencies over different time scales.

4.3 Why LSTM and GRUs have better memory than regular RNNs:

1. **Gating Mechanisms:** The primary innovation in LSTMs and GRUs is the introduction of gating mechanisms. These gates decide what information gets passed through and what gets discarded or modified. This allows them to capture and retain long-term dependencies effectively.

2. **Additive Interactions:** In standard RNNs, the hidden state is updated through multiplicative interactions, which can cause rapid decay or growth of values. LSTMs, on the other hand, have additive interactions (specifically in the cell state update mechanism), which allow values to be more steadily passed along over longer sequences.

4.4 Limitations:

1. **Still prone to vanishing or exploding gradient, albeit to a lesser extent:** The gating mechanisms in LSTMs and GRUs are designed to mitigate the vanishing gradient problem, but they don't eliminate it entirely. For very long sequences, even LSTMs and GRUs can struggle to capture all the information. However, the problem arises at much greater sequence lengths compared to simple RNNs.

2. **Complexity:** LSTMs and GRUs introduce multiple weight matrices and gating mechanisms, significantly increasing the computational and memory requirements. For instance, an LSTM has three gates (input, forget, and output) and an intermediate cell update. Each of these involves its own weight matrices and operations. This makes LSTMs (and, to a slightly lesser extent, GRUs) more computationally intensive than standard RNNs.

4.5 Complexity Increase:

1. **Parameter-wise:** LSTMs have more parameters than a standard RNN for the same hidden layer size because of the multiple gates and cell state. For a given input size n and hidden size h, a vanilla RNN has parameters of the order $O(nh + h^2)$. In contrast, an LSTM, with its three gates and cell state, increases this to $O(4nh + 4h^2)$.

2. **Computationally:** The forward pass in LSTMs and GRUs involves more matrix multiplications and element-wise operations due to the additional gates. This increases the computational burden, especially for larger models or longer sequences.

3. **Memory:** Because of the added gates and additional cell state (in the case of LSTM), these models require more memory both in terms of model parameters and intermediate computations during training.

In conclusion, while LSTMs and GRUs have been designed to address some fundamental challenges in RNNs, they are not without their limitations. Their increased complexity can make them slower to train and deploy. However, their ability to capture longer-term dependencies has made them the preferred choice for many sequence modeling tasks. It's worth noting that newer architectures like Transformers have been introduced, which offer alternative solutions to the challenges of sequence modeling.

5 Transformer Architecture

The transformer model, introduced by Vaswani et al. in their 2017 paper "Attention is All You Need," has since become the foundation for many state-of-the-art NLP models by addressing several limitations of traditional recurrent models, like Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU).

Central to transformers is the concept of attention, which allows them to weigh the significance of different parts of input data differently. Early transformer-based large language models (GPT and similar initial variants) are primarily built upon two components: the encoder and the decoder. Both of these parts utilize the self-attention mechanism, which allows the model to weigh input tokens differently based on their relevance.

5.1 Encoder

The encoder's main job is to process the input data and compress this information into a context or a sequence of vectors. These vectors then serve as the input for the decoder.

5.1.1 Multi-Head Self Attention

This mechanism allows the encoder to focus on different parts of the input for different tasks or reasons. Instead of having one set of attention weights, it has multiple sets (heads), which lets it capture various aspects of the input.

5.1.2 Position-wise Feed-Forward Networks

After the multi-head attention layer, the transformer uses feed-forward networks, which are applied to each position separately and identically.

5.1.3 Residual Connection

Each sub-layer (like multi-head attention or feed-forward neural network) in the encoder is followed by a residual connection and a layer normalization.

5.2 Decoder

The decoder's role is to produce the output data from the context provided by the encoder.

5.2.1 Masked Multi-Head Self Attention

In the decoder, the self-attention mechanism works slightly differently. It ensures that the prediction for a particular word doesn't depend on future words in the sequence, which is achieved using masking.

5.2.2 Multi-Head Attention over Encoder's Output

This layer helps the decoder focus on relevant parts of the input sentence, similar to the way attention mechanisms work in seq2seq models with LSTMs.

5.2.3 Position-wise Feed-Forward Networks and Residual Connection

Just as in the encoder, the decoder also contains feed-forward networks and residual connections.

6 Mathematics of Transformers

Let's denote the input matrix as X. The first step involves computing the attention scores. For simplicity, we'll consider the scaled dot-product attention.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

Where:

- Q is the query matrix
- K is the key matrix
- V is the value matrix
- d_k is the dimension of the keys

6.1 Advantages over RNNs

Unlike RNNs that process data sequentially, transformers leverage attention to process data in parallel. This architectural distinction addresses several RNN limitations:

- Parallelism: Transformers can process all data simultaneously. Due to parallelization, they are more expansive for context capturing and deep stacking.
- Vanishing and Exploding Gradients: The architecture is less susceptible due to its non-recurrent nature.
- Memory: The attention mechanism allows for more extended memory retention.

6.2 Architecture Variations

Encoder-Decoder: Two separate stacks for source and target.
Encoder-Only: Focuses on encoding input for tasks like embeddings.
Decoder-Only: Used primarily in LLMs for generative tasks.

6.3 Why Decoder-Only in LLMs?

Most recently many successful large language models have a decoder-only architecture primarily because these are efficient for generative tasks, and reduce complexity.

7 Zero-Shot and Few-Shot Learning in LLMs

In the realm of machine learning, the terms "zero-shot," "one-shot," and "few-shot" learning refer to the ability of a model to understand and perform tasks that it hasn't been explicitly trained on. For Large Language Models (LLMs), this capability is particularly fascinating.

7.1 Zero-Shot Learning

Zero-shot learning pertains to the model's capability to handle tasks without seeing any explicit example during training. Given the right prompt, LLMs can generate relevant outputs for tasks they haven't been fine-tuned on.

7.2 Few-Shot Learning

In contrast, few-shot learning involves providing the model with a handful of examples to help it grasp the task better. By showcasing the input-output pattern through these examples, LLMs can effectively generalize the pattern and apply it to novel inputs.

8 Prompt Engineering

Prompt engineering has emerged as an essential practice when working with LLMs. Crafting the right prompt can drastically alter the output of the model. While it seems simple, prompt engineering often requires an intricate understanding of how the model thinks and can be as much of an art as it is a science. This topic's depth and breadth, exploring various strategies and techniques, would warrant an entirely separate discussion.

9 Fine-Tuning LLMs

Fine-tuning is a strategy used to tailor a pre-trained model (like an LLM) to a specific task or domain by continuing training on a smaller, task-specific dataset. This process can significantly boost performance in a specific task without requiring training a model from scratch.

9.1 Common Fine-Tuning Techniques

9.1.1 Gradient-Based Fine-Tuning

This is the most straightforward fine-tuning method. It simply involves continuing the training process on the new dataset and adjusting the model weights using gradient descent.

9.1.2 Layer Freezing

In this technique, only a subset of the model's layers is trained during the fine-tuning phase. The remaining layers are kept frozen, preserving the knowledge they have. This method is particularly useful when the fine-tuning dataset is small, reducing the risk of overfitting.

9.1.3 Adaptive Learning Rates

Different learning rates might be used for different layers or parts of the model. For instance, layers closer to the input might be fine-tuned with a slower learning rate compared to the top layers.

9.1.4 Adapter Architecture

Adapter architectures add small, task-specific feed-forward networks (adapters) between the layers of the original model. During fine-tuning, only these adapter layers are trained, while the original model's weights remain frozen. This approach allows for task-specific adaptations without altering the pre-trained parameters, ensuring efficient and modular fine-tuning.

9.2 Nontriviality of Fine-Tuning

Choosing the right fine-tuning strategy can significantly impact the performance of the model on the target task. It's essential to consider the size and quality of the fine-tuning dataset. Using non-representative or low-quality data can lead to suboptimal or even harmful model behaviors. Moreover, there's a risk of overfitting to the fine-tuning data due to its typically smaller size compared to the initial training data.

10 The Role of High-Quality Data

Recent research indicates that with high-quality data, one can train smaller yet more accurate models. We will later publish a tutorial on data and LLMs.

11 Limitations

Despite their success, transformers and LLMs are not without their challenges. Issues like hallucinations, expensive computational costs, and unpredictability still persist.

12 Conclusion

The transformative role of transformers in the field of NLP is undeniable. By harnessing their unique ability to discern and attend to salient pieces of infor-

mation, these architectures have instigated a seismic shift in our understanding and application of NLP techniques. However, as we stride forward, it is imperative that research efforts also concentrate on enhancing model reliability, scalability, and controllability. It's worth noting that this tutorial offers but a glimpse into the vast expanse of NLP advancements. Many pertinent topics and details have been reserved for more exhaustive discussions, underscoring the depth and dynamism of this ever-evolving field.