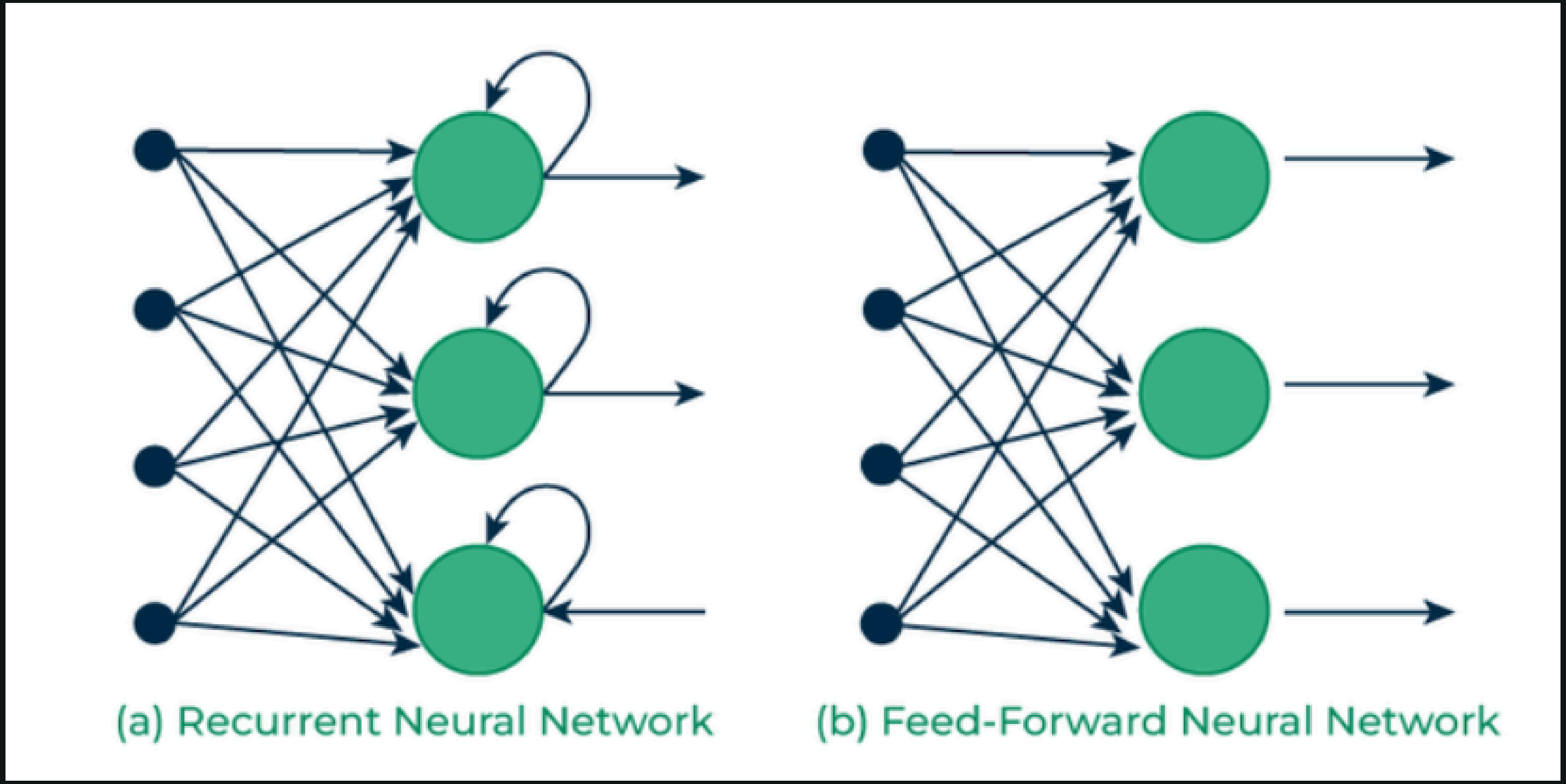
SEQUENTIAL NEURAL NETWORKS

Architecting Intelligence

How RNN Differs from Feedforward Neural Networks?

Feedforward Neural Networks (FNNs) process data in one direction from input to output without retaining information from previous inputs. This makes them suitable for tasks with independent inputs like image classification. However FNNs struggle with sequential data since they lack memory.

Recurrent Neural Networks (RNNs) solve this by incorporating loops that allow information from previous steps to be fed back into the network. This feedback enables RNNs to remember prior inputs making them ideal for tasks where context is important.



Recurrent Vs Feedforward networks

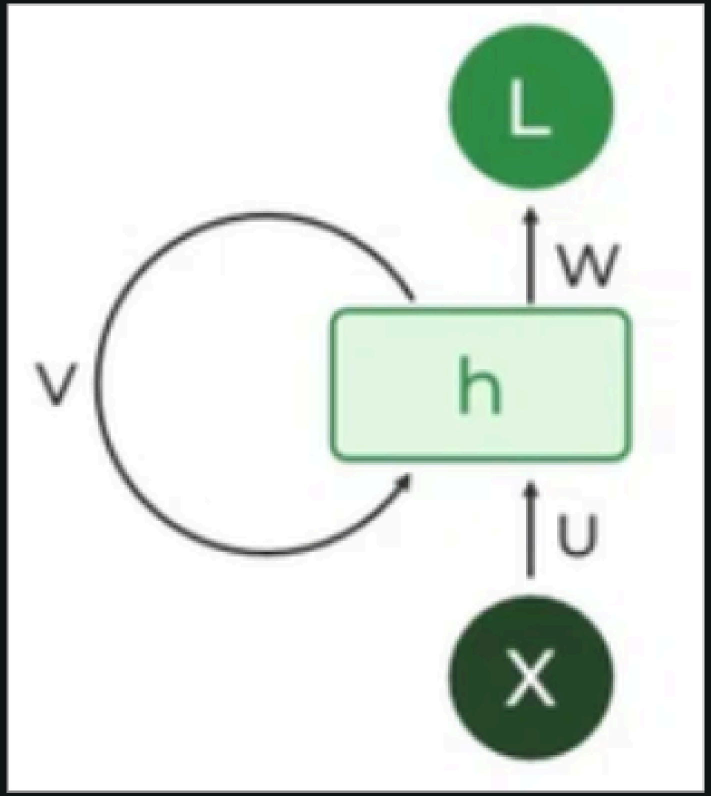
Challenges with ANNs:

1. Text input can be of varying size
2. So, we do zero padding to try solve the issue. Let’s say the max number of words in the sentence is 100 and min is 5, so we do zero padding for 95 inputs in the input layer so it becomes computationally expensive and leads to **Unnecessary computation**
3. If input’s max word length is 100, and the text input for prediction is of 200, so even zero padding would fail because no input was of size=100
4. ANN does not have any memory to retain sequence. Semantic meaning of the text is not captured by ANNs

Recurrent Neural Networks (RNNs) differ from regular neural networks in how they process information. While standard neural networks pass information in one direction i.e. from input to output, RNNs feed information back into the network at each step.

1. Recurrent Neurons

The fundamental processing unit in RNN is a Recurrent Unit. They hold a hidden state that maintains information about previous inputs in a sequence. Recurrent units can "remember" information from prior steps by feeding back their hidden state, allowing them to capture dependencies across time.

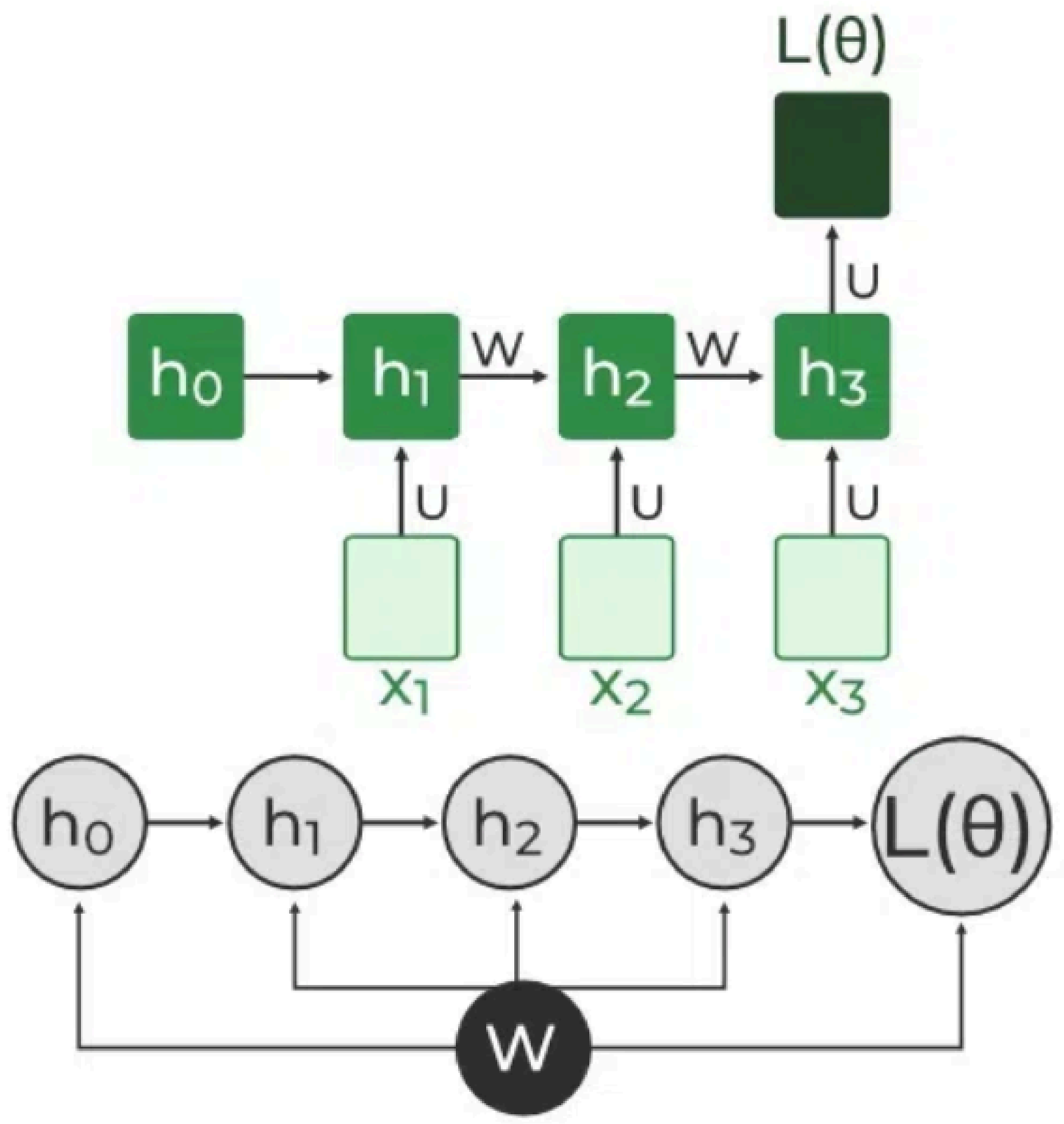


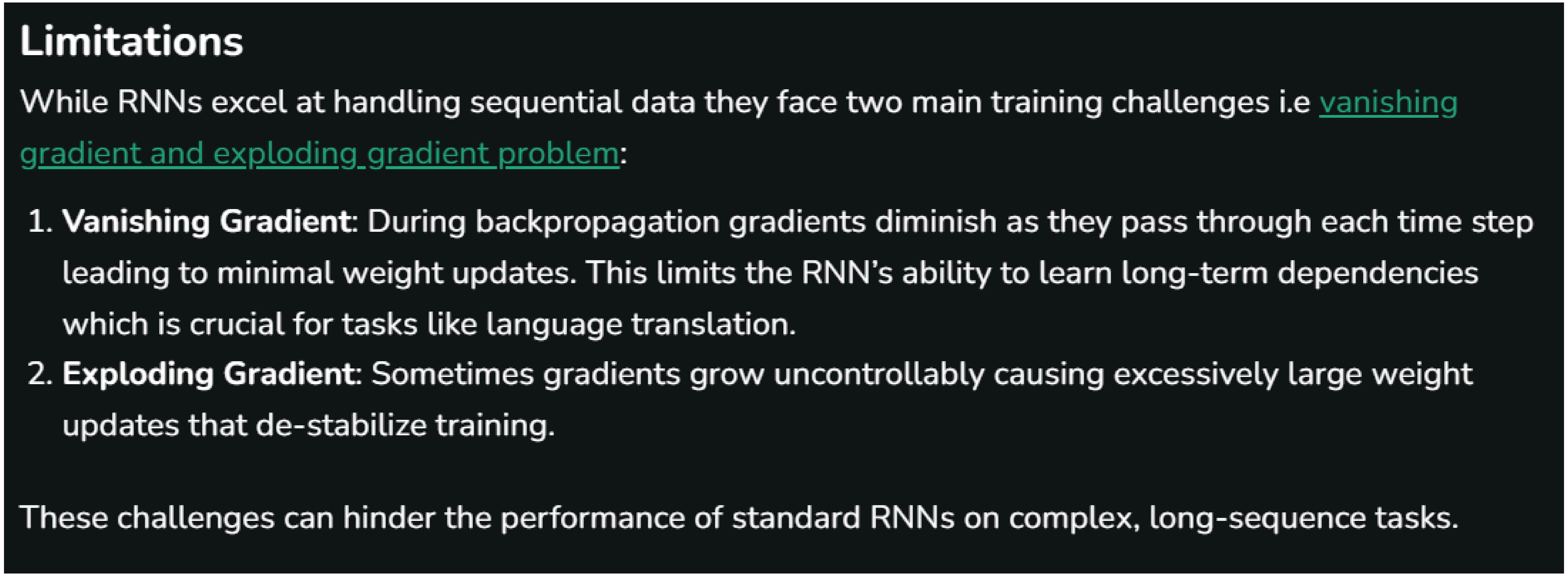
Recurrent Neuron

Backpropagation Through Time (BPTT) in RNNs

Since RNNs process sequential data Backpropagation Through Time (BPTT) is used to update the network's parameters. The loss function L(θ) depends on the final hidden state h_3 and each hidden state relies on preceding ones forming a sequential dependency chain:

h_3 depends on depends on h_2, h_2 depends on h_1, $\ldots$, h_1 depends on h_0.



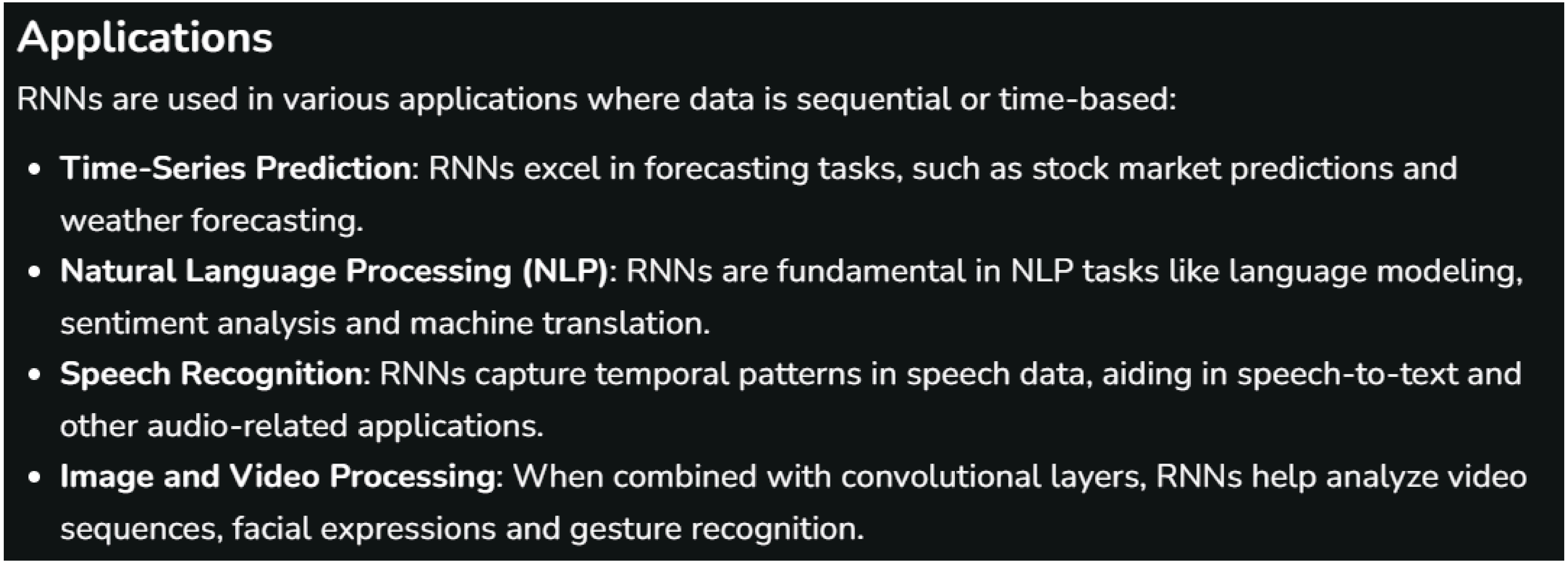


Limitations

While RNNs excel at handling sequential data they face two main training challenges i.e vanishing gradient and exploding gradient problem:

1. **Vanishing Gradient**: During backpropagation gradients diminish as they pass through each time step leading to minimal weight updates. This limits the RNN's ability to learn long-term dependencies which is crucial for tasks like language translation.
2. **Exploding Gradient**: Sometimes gradients grow uncontrollably causing excessively large weight updates that de-stabilize training.

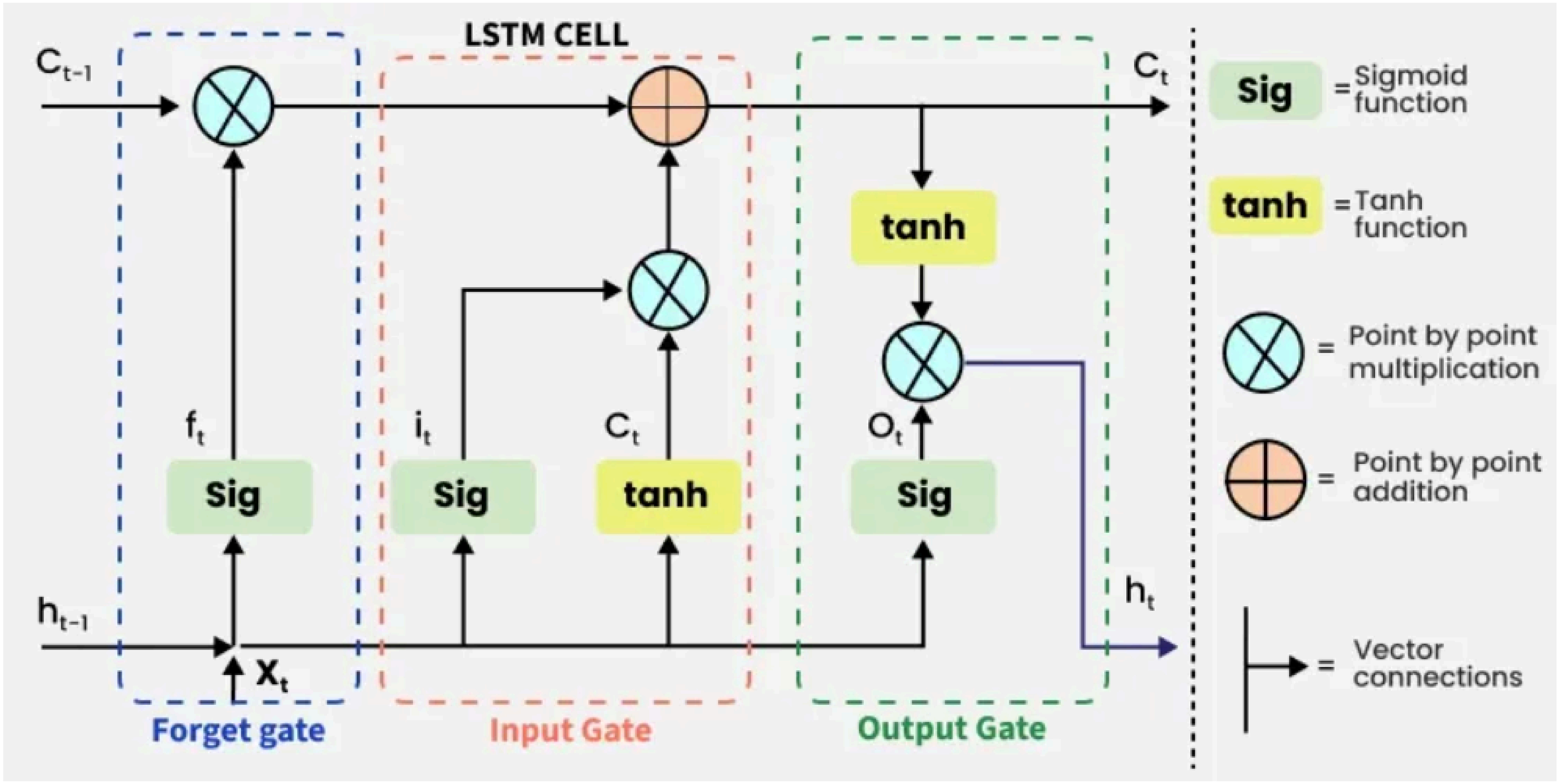
These challenges can hinder the performance of standard RNNs on complex, long-sequence tasks.



Applications

RNNs are used in various applications where data is sequential or time-based:

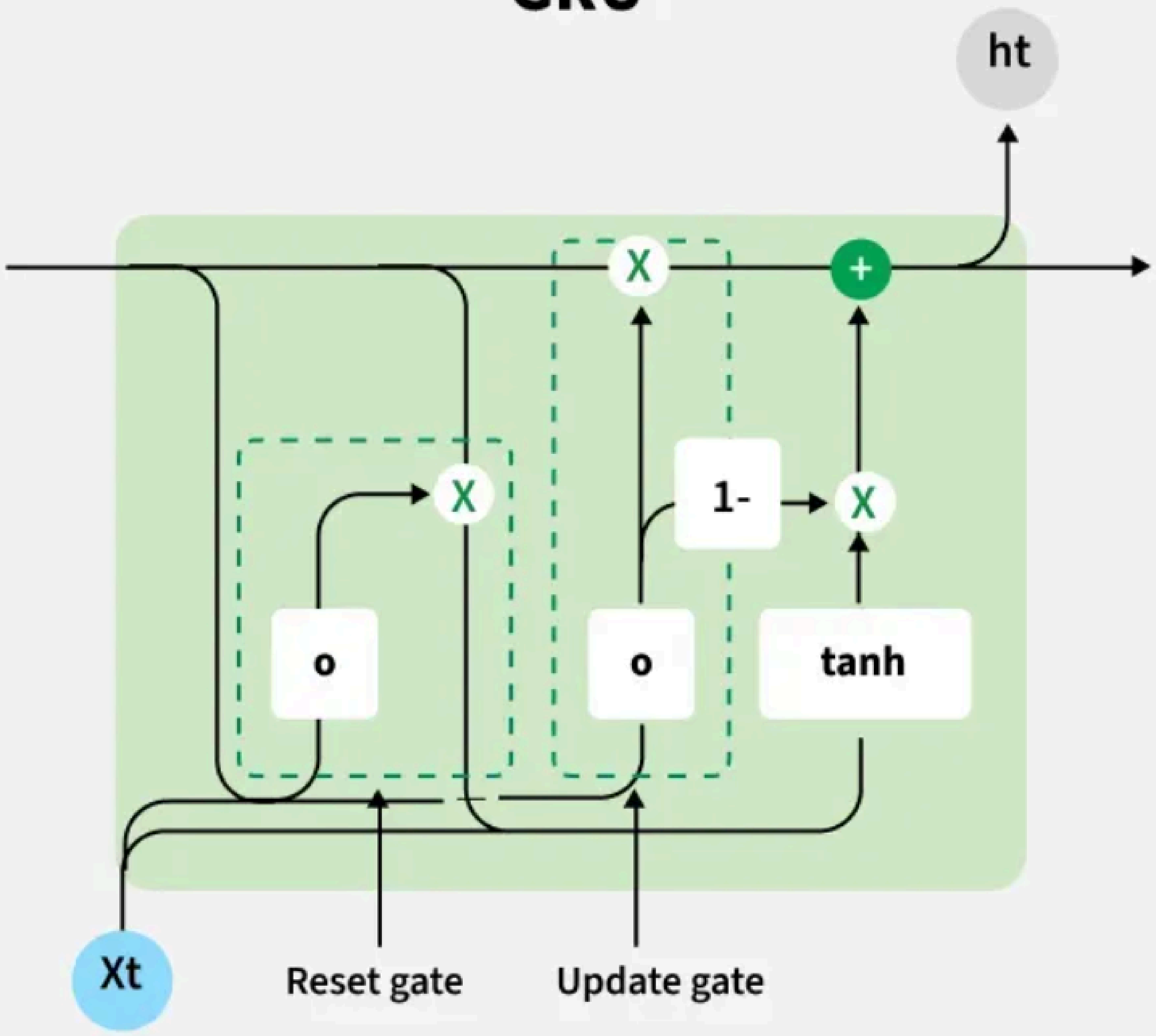
- **Time-Series Prediction**: RNNs excel in forecasting tasks, such as stock market predictions and weather forecasting.
- **Natural Language Processing (NLP)**: RNNs are fundamental in NLP tasks like language modeling, sentiment analysis and machine translation.
- **Speech Recognition**: RNNs capture temporal patterns in speech data, aiding in speech-to-text and other audio-related applications.
- **Image and Video Processing:** When combined with convolutional layers, RNNs help analyze video sequences, facial expressions and gesture recognition.

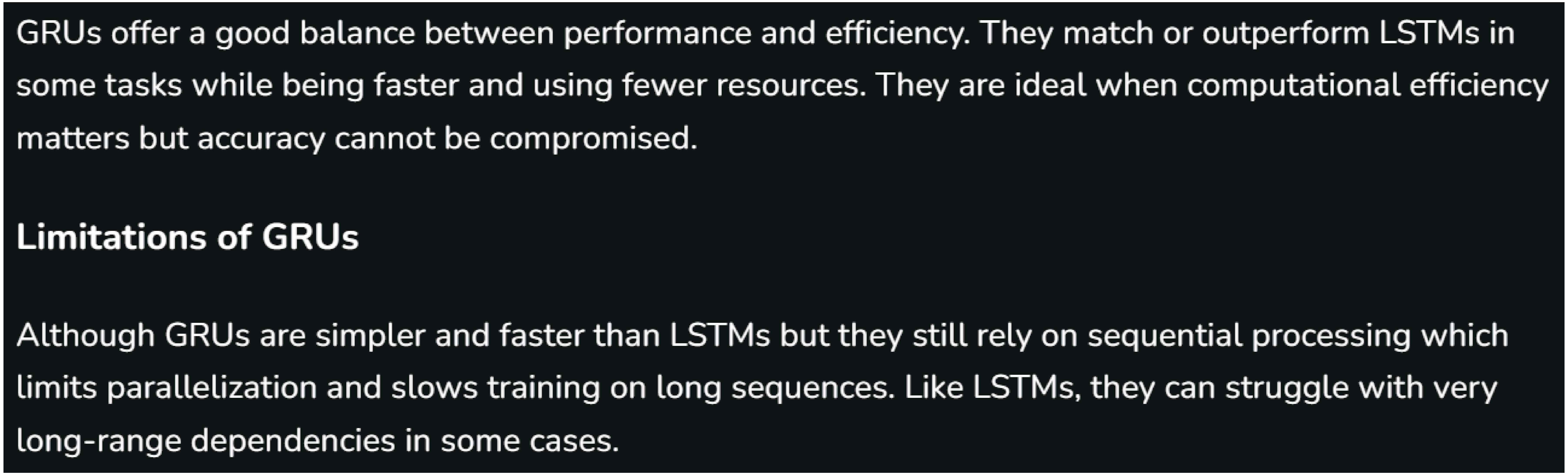

LSTM CELL
C_{t-1}
C_t
h_{t-1}
X_t
f_t
i_t
C_t
O_t
h_t
Sig
Sig
tanh
tanh
Sig
Forget gate
Input Gate
Output Gate
Sig = Sigmoid function
tanh = Tanh function
= Point by point multiplication
= Point by point addition
= Vector connections

Limitations of LSTMs

They are more complex than RNNs which makes them slower to train and demands more memory. Despite handling longer sequences better they still face challenges with very long-range dependencies. Their sequential nature also limits the ability to process data in parallel which slows down training.

GRU





GRUs offer a good balance between performance and efficiency. They match or outperform LSTMs in some tasks while being faster and using fewer resources. They are ideal when computational efficiency matters but accuracy cannot be compromised.

Limitations of GRUs

Although GRUs are simpler and faster than LSTMs but they still rely on sequential processing which limits parallelization and slows training on long sequences. Like LSTMs, they can struggle with very long-range dependencies in some cases.

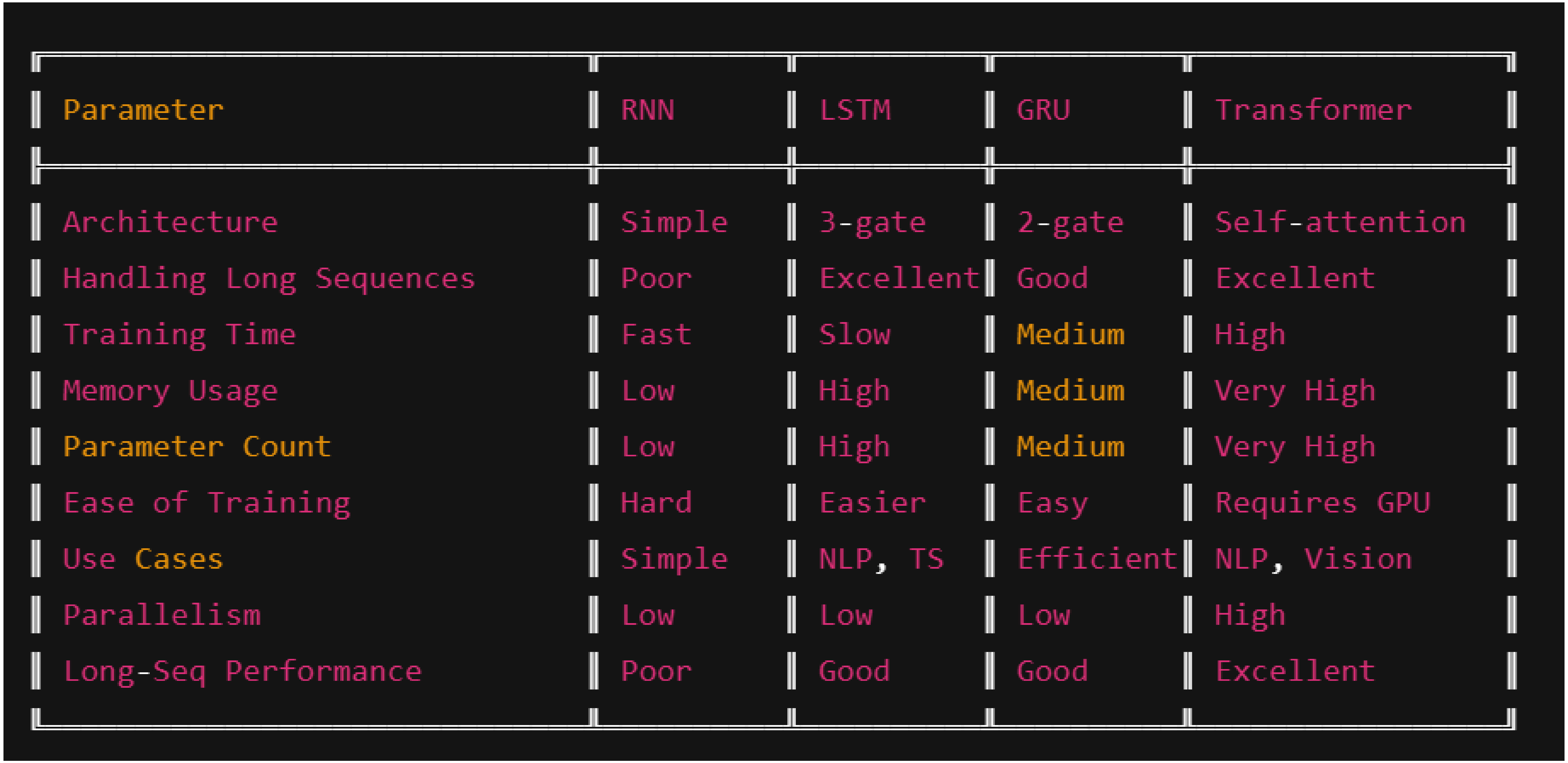
Comparison

1. Number of Gates:
LSTM: Has three gates — input (or update) gate, forget gate, and output gate.
GRU: Has two gates — reset gate and update gate.
2. Memory Units:
LSTM: Uses two separate states — the cell state (c_t) and the hidden state (h_t). The cell state acts as an “internal memory” and is crucial for carrying long-term dependencies.
GRU: Simplifies this by using a single hidden state (h_t) to both capture and output the memory.
3. Parameter Count:
LSTM: Generally has more parameters than a GRU because of its additional gate and separate cell state.
For an input size of d and a hidden size of h, the LSTM has
$4 \times ((d \times h) + (h \times h) + h)$ parameters.
GRU: Has fewer parameters.
For the same sizes, the GRU has
$3 \times ((d \times h) + (h \times h) + h)$ parameters.
4. Computational Complexity:
LSTM: Due to the extra gate and cell state, LSTMs are typically more computationally intensive than GRUs.
GRU: Is simpler and can be faster to compute, especially on smaller datasets or when computational resources are limited.
5. Empirical Performance:
LSTM: In many tasks, especially more complex ones, LSTMs have been observed to perform slightly better than GRUs.
GRU: Can perform comparably to LSTMs on certain tasks, especially when data is limited or tasks are simpler. They can also train faster due to fewer parameters.

Why RNNs, LSTMs and GRUs Failed Leading to the Rise of Transformers?

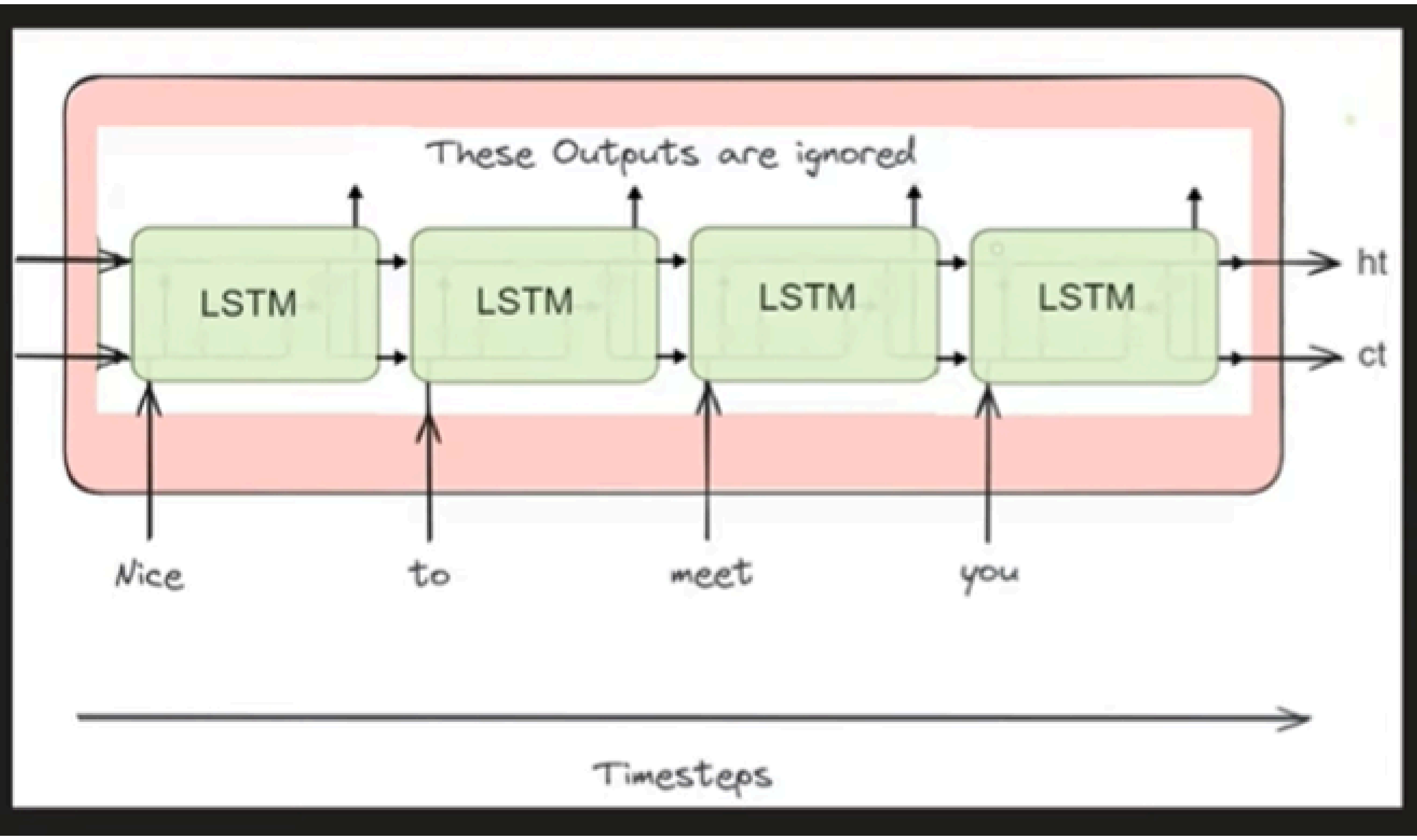
While LSTMs and GRUs improved on basic RNNs, they still had major drawbacks. Their step-by-step sequential processing made it difficult to handle very long sequences and complex dependencies efficiently. This sequential nature also limited parallelization which causes slow and costly training.

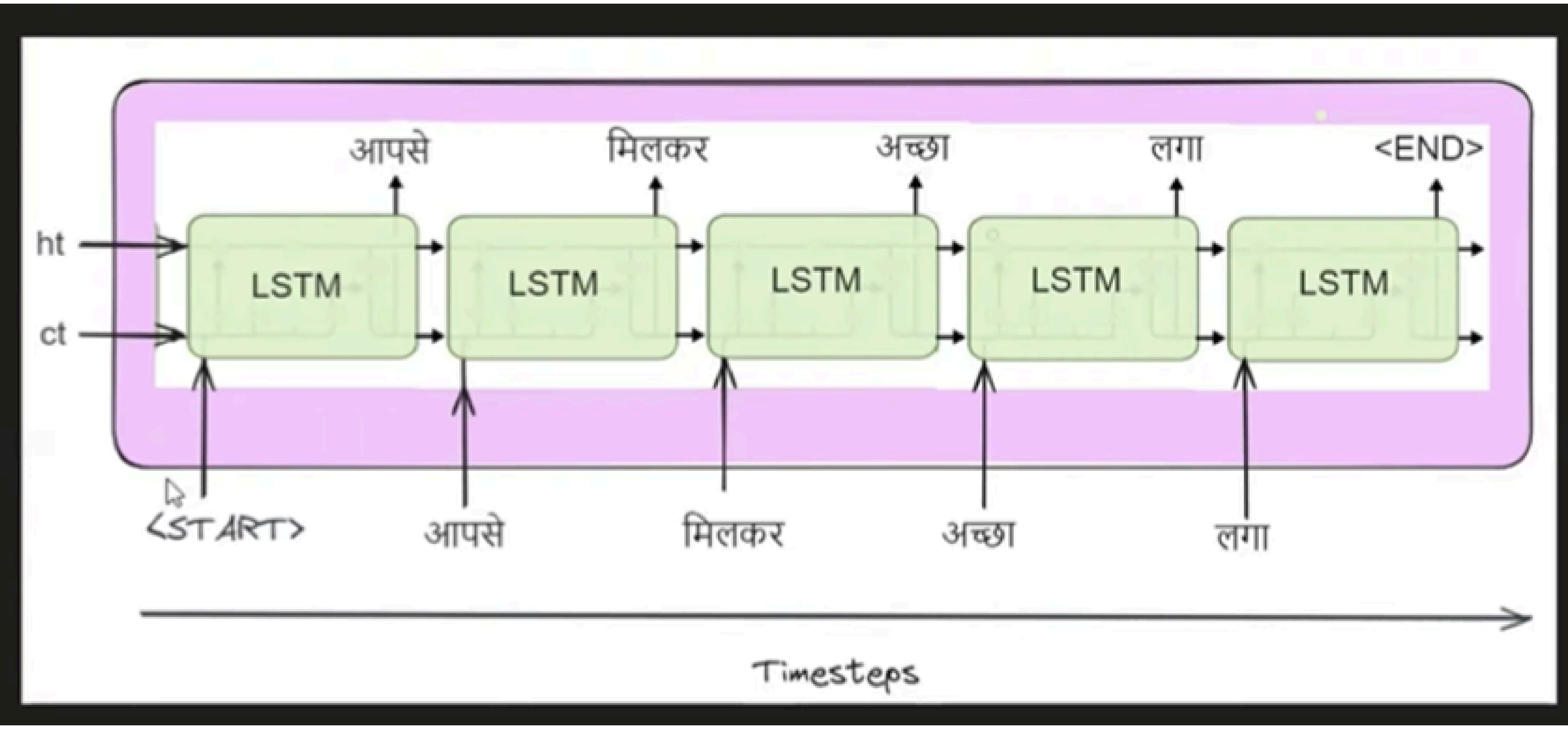
Transformers solved these problems by using self-attention which processes the entire sequence at once. This allows transformers to capture long-range dependencies more effectively and train much faster. Unlike RNN-based models, transformers do not rely on sequential steps helps in making them highly scalable and suitable for larger datasets and more complex tasks.

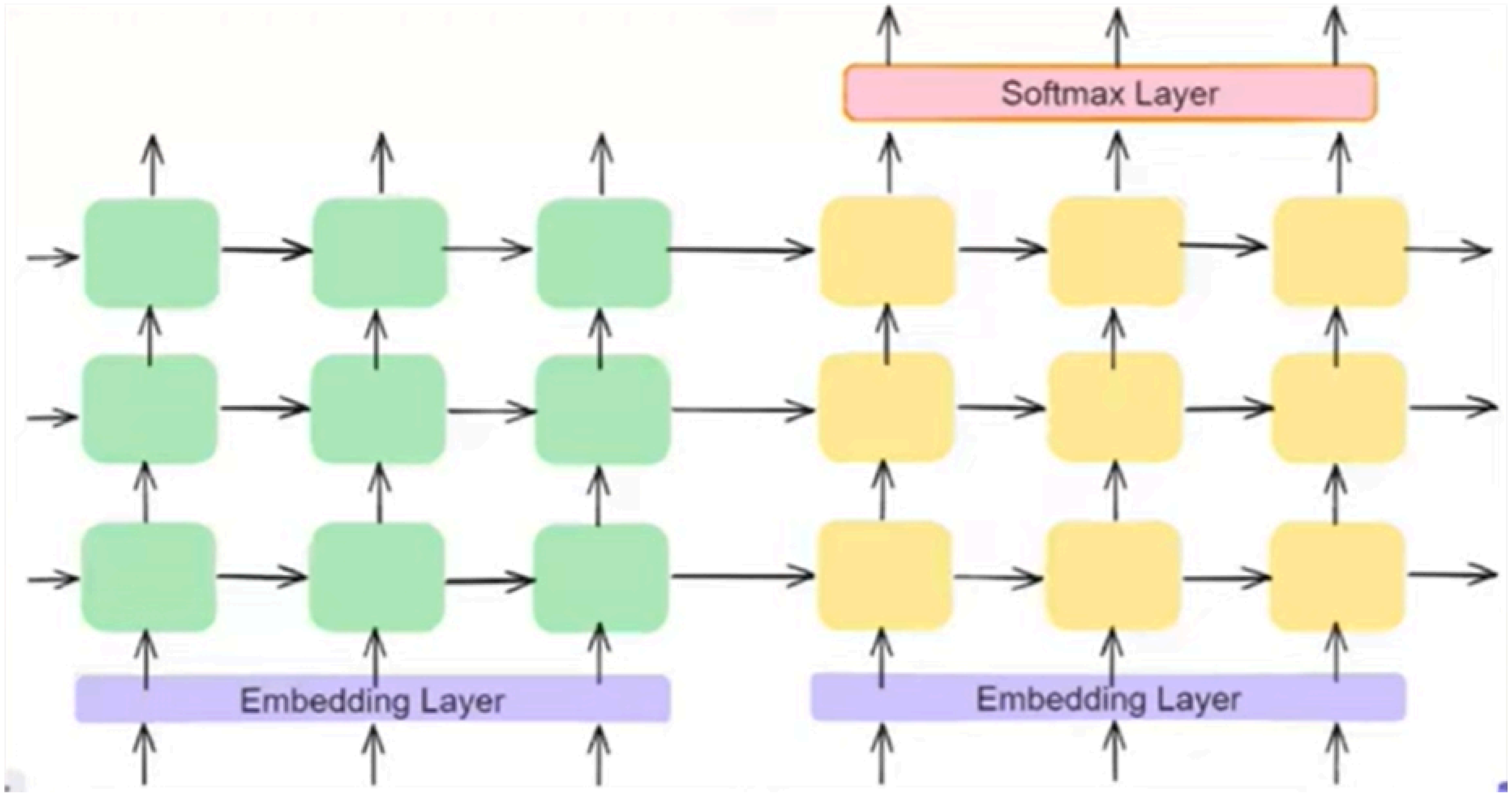


Parameter	RNN	LSTM	GRU	Transformer
Architecture	Simple	3-gate	2-gate	Self-attention
Handling Long Sequences	Poor	Excellent	Good	Excellent
Training Time	Fast	Slow	Medium	High
Memory Usage	Low	High	Medium	Very High
Parameter Count	Low	High	Medium	Very High
Ease of Training	Hard	Easier	Easy	Requires GPU
Use Cases	Simple	NLP, TS	Efficient	NLP, Vision
Parallelism	Low	Low	Low	High
Long-Seq Performance	Poor	Good	Good	Excellent

Encoder-Decoder Architecture






Softmax Layer
Embedding Layer
Embedding Layer

Sequence to Sequence Learning with Neural Networks

Ilya Sutskever
Google
ilyasu@google.com

Oriol Vinyals
Google
vinyals@google.com

Quoc V. Le
Google
qvl@google.com

Abstract

Deep Neural Networks (DNNs) are powerful models that have achieved excellent performance on difficult learning tasks. Although DNNs work well whenever large labeled training sets are available, they cannot be used to map sequences to sequences. In this paper, we present a general end-to-end approach to sequence learning that makes minimal assumptions on the sequence structure. Our method uses a multilayered Long Short-Term Memory (LSTM) to map the input sequence to a vector of a fixed dimensionality, and then another deep LSTM to decode the target sequence from the vector. Our main result is that on an English to French translation task from the WMT'14 dataset, the translations produced by the LSTM achieve a BLEU score of 34.8 on the entire test set, where the LSTM's BLEU score was penalized on out-of-vocabulary words. Additionally, the LSTM did not have difficulty on long sentences. For comparison, a phrase-based SMT system achieves a BLEU score of 33.3 on the same dataset. When we used the LSTM to rerank the 1000 hypotheses produced by the aforementioned SMT system, its BLEU score increases to 36.5, which is close to the previous best result on this task. The LSTM also learned sensible phrase and sentence representations that are sensitive to word order and are relatively invariant to the active and the passive voice. Finally, we found that reversing the order of the words in all source sentences (but not target sentences) improved the LSTM's performance markedly, because doing so introduced many short term dependencies between the source and the target sentence which made the optimization problem easier.

3.4 Training details

We found that the LSTM models are fairly easy to train. We used deep LSTMs with 4 layers, with 1000 cells at each layer and 1000 dimensional word embeddings, with an input vocabulary of 160,000 and an output vocabulary of 80,000. Thus the deep LSTM uses 8000 real numbers to represent a sentence. We found deep LSTMs to significantly outperform shallow LSTMs, where each additional layer reduced perplexity by nearly 10%, possibly due to their much larger hidden state. We used a naive softmax over 80,000 words at each output. The resulting LSTM has 384M parameters of which 64M are pure recurrent connections (32M for the "encoder" LSTM and 32M for the "decoder" LSTM). The complete training details are given below:

- We initialized all of the LSTM's parameters with the uniform distribution between -0.08 and 0.08
- We used stochastic gradient descent without momentum, with a fixed learning rate of 0.7. After 5 epochs, we begun halving the learning rate every half epoch. We trained our models for a total of 7.5 epochs.
- We used batches of 128 sequences for the gradient and divided it the size of the batch (namely, 128).
- Although LSTMs tend to not suffer from the vanishing gradient problem, they can have exploding gradients. Thus we enforced a hard constraint on the norm of the gradient [10, 25] by scaling it when its norm exceeded a threshold. For each training batch, we compute $s = \|g\|_2$, where g is the gradient divided by 128. If $s > 5$, we set $g = \frac{5g}{s}$.
- Different sentences have different lengths. Most sentences are short (e.g., length 20-30) but some sentences are long (e.g., length > 100), so a minibatch of 128 randomly chosen training sentences will have many short sentences and few long sentences, and as a result, much of the computation in the minibatch is wasted. To address this problem, we made sure that all sentences in a minibatch are roughly of the same length, yielding a 2x speedup.