Course Materials for GEN-AI

Northeastern University

Instructor: Ramin Mohammadi
r.mohammadi@northeastern.edu

Recurrent Neural Networks (RNNs)

Intorduction

Recurrent Neural Networks (RNNs) are a type of neural network designed to handle sequential data, like time series, sentences, or any data where the order of inputs matters. Unlike traditional neural networks, RNNs have connections that loop back on themselves, allowing them to "remember" previous information and use it when processing new inputs.

RNNs are useful in applications like language translation, speech recognition, and time series prediction because they can capture patterns over time.

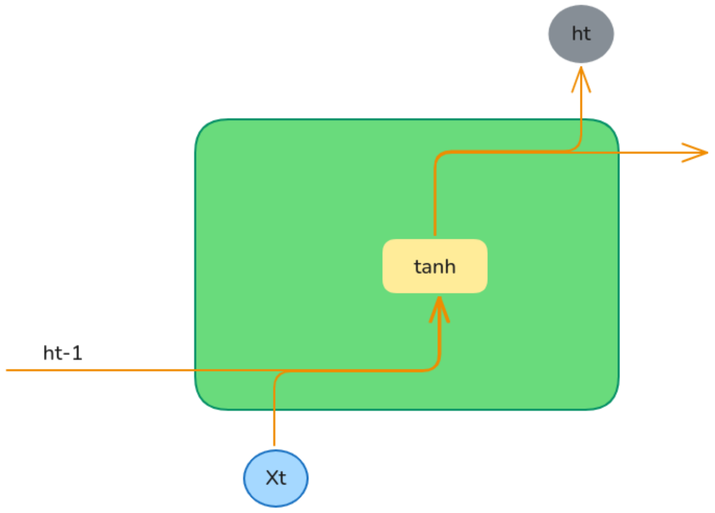


Figure 1: RNN

Parameter sharing in RNN

Unlike traditional neural networks, where each layer may have its own set of weights, RNNs use the same set of parameters (weights and biases) across all time steps in the sequence. An effective approach is to conceptualize the problem as involving multiple networks working in union to handle varying lengths of input data.

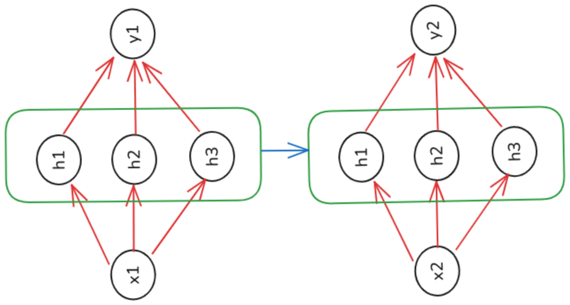


Figure 2: Conceptual view of RNNs

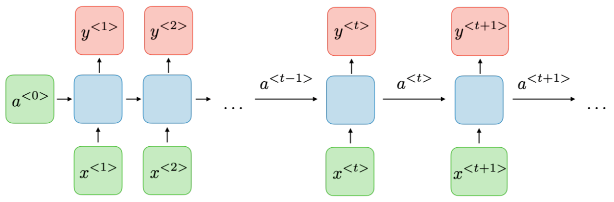
Let's undrstand the motivation behind RNNs using an example: supposed that we have multiple NNs instead of using a single network. One could provide a single word and request a prediction, or provide two words and request another prediction. When two words are provided, the networks puts together two of NNs and process both words together.

Similarly, with a set of 10 words, the networks combine 10 of NNs and process all 10 words simultaneously.
This concept of parameter sharing is crucial because it allows the RNN to generalize patterns across the sequence and reduces the complexity of the model.

How does this model work for RNN?

To "make this model work," there are a couple of key considerations:

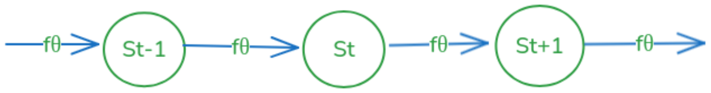
1. Networks do not know about each other: In a sequence, each time step can be thought of as a separate network. These networks need to be connected in such a way that each step has knowledge of the previous one. This is done by sharing the hidden states between time steps. Essentially, the output of one step becomes an input (hidden state) to the next, allowing the RNN to "remember" what happened earlier in the sequence.
2. Identical networks (in terms of weights): RNNs achieve this connection between steps through parameter sharing. The same set of weights is used across all time steps, ensuring that each "network" processes data in a consistent manner.



Dynamic Systems

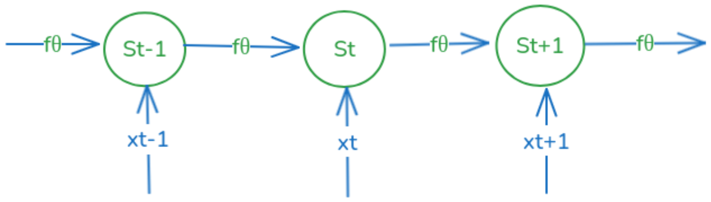
A dynamic system refers to the way the network evolves and processes information over time, with its outputs and hidden states depending not only on the current input but also on the history of previous inputs (like a recursive function). The mathematical representation is:

$$s_t = f(s_{t-1})$$



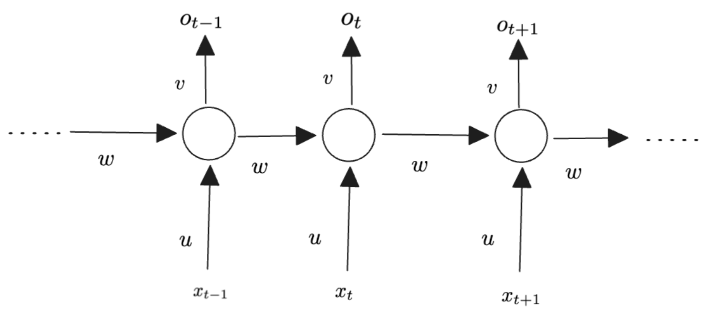
Now consider a dynamic system with an external signal x:

that has Markovian assumption that current, past, and future are independent.



Each state now contains information about the whole past sequence. It fails to generalize well if different function is used at each state.

Dynamic Systems to RNN



$$x \in R^d \tag{1}$$
$$s \in R^p \tag{2}$$
$$o \in R^q \tag{3}$$
$$w \in R^{P \times P} \tag{4}$$
$$u \in R^{P \times d} \tag{5}$$

$$v \in R^{P \times q} \tag{6}$$

$$a_t = b + ws_{t-1} + ux_t \tag{7}$$

$$s_t = \sigma(a_t) - \textbf{Using Sigmoid} \tag{8}$$
$$s_t = \tanh(a_t) - \textbf{Using Tanh} \tag{9}$$

$$o_t = C + vs_t \tag{10}$$
$$P_t = \text{ softmax } (o_t) - \textbf{In case of probability} \tag{11}$$

where:

- b is the bias term for internal steps
- C is the bias term of the output

Computing Gradient in RNN

Using the generalized backpropagation algorithm, we can derive the ”Back Propagation Through Time (BPTT)” algorithm.

$$L = \sum_t L_t \tag{12}$$

The loss function for the entire sequence, where at each time step a prediction is made.

$$\frac{\partial L}{\partial o_t} = \frac{\partial L}{\partial L_t} \cdot \frac{\partial L_t}{\partial o_t} \tag{13}$$

where $\frac{\partial L}{\partial L_t} = 1$ and $\frac{\partial L_t}{\partial o_t}$ depends on the cost function chosen. The output at time t does not impact the output at other time steps.

$$\frac{\partial L}{\partial s_T} = \frac{\partial L}{\partial o_T} \cdot \frac{\partial o_T}{\partial s_T} = \frac{\partial L}{\partial o_T} \cdot v \tag{14}$$

T is the last time step or end of the sequence. It's important to note that a change in s_t will impact o_t and any s_t in the future.

$$\partial_t = \frac{\partial L}{\partial s_t} = \frac{\partial L}{\partial o_t} \cdot \frac{\partial o_t}{\partial s_t} + \frac{\partial L}{\partial s_{t+1}} \cdot \frac{\partial s_{t+1}}{\partial s_t} \tag{15}$$

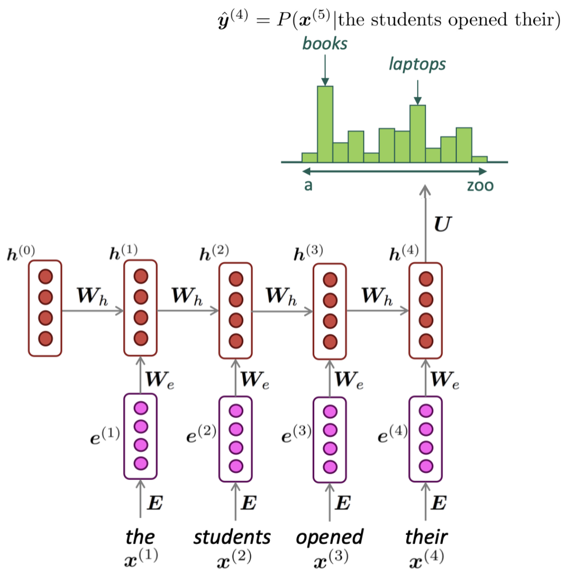
$$\partial_t = \frac{\partial L}{\partial o_t} \cdot v + \partial_{t+1} \cdot w \cdot \left(1 - \tanh^2(b + ws_{t-1} + ux_t)\right) \tag{16}$$

$$\frac{\partial L}{\partial v} = \sum_t \frac{\partial L}{\partial o_t} \cdot \frac{\partial o_t}{\partial v} = \sum_t \frac{\partial L}{\partial o_t} \cdot s_t \tag{17}$$

$$\frac{\partial L}{\partial w} = \sum_t \frac{\partial L}{\partial s_t} \cdot \frac{\partial s_t}{\partial w} = \sum_t \partial_t \cdot \left(1 - \tanh^2(s_t)\right) \cdot s_{t-1} \tag{18}$$

$$\frac{\partial L}{\partial u} = \sum_t \frac{\partial L}{\partial s_t} \cdot \frac{\partial s_t}{\partial u} = \sum_t \partial_t \cdot \left(1 - \tanh^2(s_t)\right) \cdot x_t \tag{19}$$

RNN Advantages:



- Can process **any length** input.
- Computation for step t can (in theory) use information from **many steps back**.
- **Model size doesn't increase** for longer input context.
- Same weights applied on every timestep, so there is **symmetry** in how inputs are processed.

RNN Disadvantages:

- Recurrent computation is **slow**.
- In practice, difficult to access information from **many steps back**.

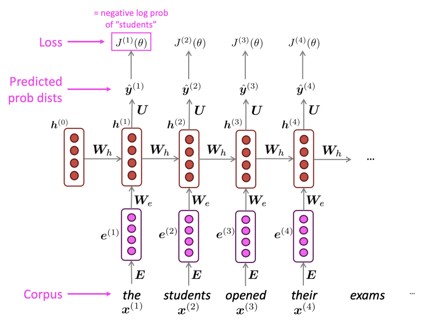
Training an RNN Language Model

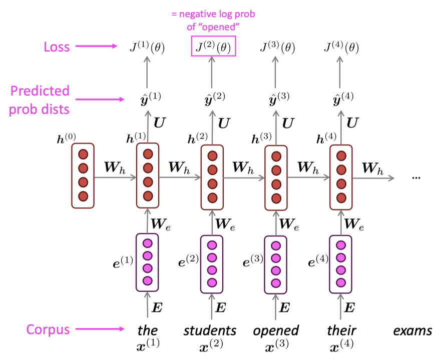
- Get a **big corpus of text** which is a sequence of words $x^{(1)}, \ldots, x^{(T)}$.
- Feed into RNN-LM; compute output distribution $\hat{y}^{(t)}$ **for every step t**.
  - i.e., predict probability distribution of **every word**, given words so far.
- **Loss function** on step t is **cross-entropy** between predicted probability distribution $\hat{y}^{(t)}$ and the true next word $y^{(t)}$ (one-hot for $x^{(t+1)}$):

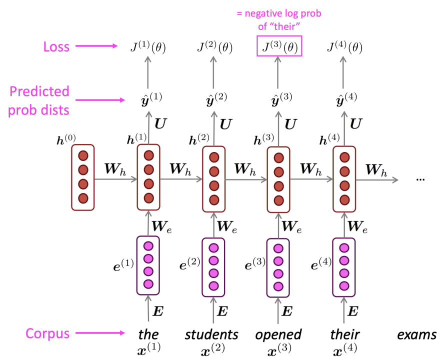
$$J^{(t)}(\theta) = CE(y^{(t)}, \hat{y}^{(t)}) = -\sum_{w \in V} y_w^{(t)} \log \hat{y}_w^{(t)} = -\log \hat{y}_{x_{t+1}}^{(t)}$$

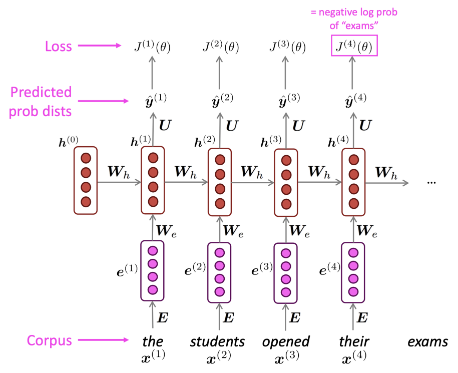
- Average this to get **overall loss** for the entire training set:

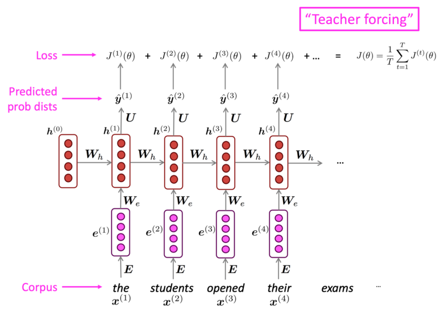
$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T}\sum_{t=1}^{T} -\log \hat{y}_{x_{t+1}}^{(t)}$$











- However: Computing loss and gradients across **entire corpus** $x^{(1)}, \ldots, x^{(T)}$ at once is **too expensive** (memory-wise)!

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta)$$

- In practice, consider $x^{(1)}, \ldots, x^{(T)}$ as a **sentence** (or a **document**).
- **Recall: Stochastic Gradient Descent** allows us to compute loss and gradients for small chunk of data, and update.
- Compute loss $J(\theta)$ for a sentence (actually, a batch of sentences), compute gradients and update weights. Repeat on a new batch of sentences.

Problems with RNN

When we multiply many derivatives together across time in RNNs during backpropagation through time (BPTT), the following two problems can occur:
Exploding Gradients: If the gradients (derivatives) are large, multiplying them over many time steps can cause the gradients to grow exponentially, leading to extremely large values. This results in instability in the learning process, with the weights being updated too drastically, potentially causing the model to diverge.
Vanishing Gradients: On the other hand, if the gradients are small, their multiplication over many time steps causes them to shrink exponentially. This leads to very small gradient values, effectively preventing the model from learning long-term dependencies, as the weight updates become negligible.
This is due to the fact that $f = f_T \cdot f_{T-1} \cdot \dots \cdot f_2 \cdot f_1$ hence its Jacobian will be:

$$f' = f'_T f'_{T-1} \dots f'_2 f'_1 \tag{20}$$

$$\begin{aligned}&\text{where:}\\ &f' = \frac{\partial f(x)}{\partial x}\\ &f'_t = \frac{\partial f_t(a_t)}{\partial a_t} \quad \text{and} \quad a_t = f_{t-1}\left(f_{t-2}\left(\dots f_2\left(f_1(x)\right)\right)\right)\Big)\end{aligned} \tag{21}$$

- If derivatives are > 1, then the multiplication can go to infinity (explosion). This means that the eigen values of the Jacobian should be > 1 which is less common.
- If derivatives are < 1, then the multiplication can go to zero (vanishing). This means that the eigen values of the Jacobian should be < 1 which is really common.

In exploding gradients situation, the search space has a gradient like the image below, it's very non-linear and flat in some areas, then all of a sudden becomes quite large (similar to a cliff). As the process approaches the minima, a sudden large gradient causes a significant deviation from the minima, resulting in slower convergence.

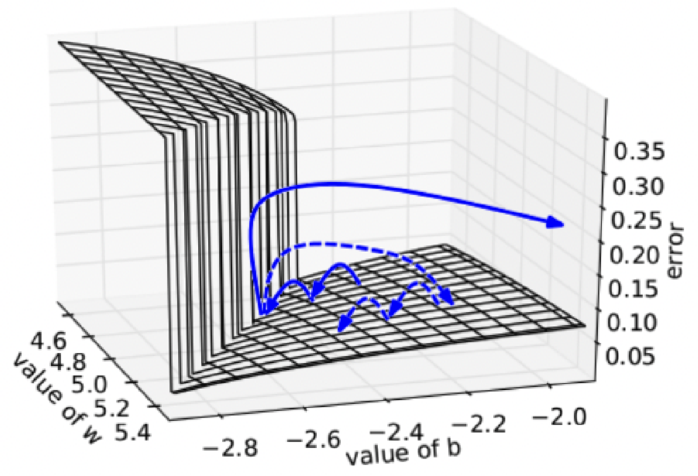


Figure 3: Exploding Gradients

How to solve these issues?

Gradient Clipping

Gradient clipping is a technique used in training neural networks to prevent a problem called exploding gradients. To prevent this, gradient clipping sets a limit on how large the gradients can get. If a gradient value is higher than a certain threshold, it's "clipped" or reduced to that maximum value. This ensures that the updates to the model's weights are more controlled, preventing the training from going off track.

Residual Connection

Residual connections help mitigate the exploding gradient issue by providing a direct path for information and gradients to flow, bypassing some layers. This stabilizes the weight updates and prevents gradients from becoming too large. While residual connections are more commonly used to address vanishing gradients, they also reduce the likelihood of gradients exploding in deep networks by ensuring smoother information flow.

Echo State Networks

Echo State Networks (ESNs) help solve the gradient problems in RNNs, particularly the vanishing gradient problem, by using a fixed recurrent weight matrix with specific properties. In an ESN, only the output weights are learned, while the input and recurrent weights are randomly initialized and remain unchanged.
The key idea is to set the eigenvalues of the recurrent weight matrix (the Jacobian) to be less than 1 but close to it. This ensures that the activations decay slowly, allowing information to persist over time without vanishing too quickly. This controlled decay helps maintain stability while still allowing the network to learn temporal dependencies. By focusing on learning only the output weights, ESNs avoid the issues of exploding or vanishing gradients during backpropagation.

Long Delay Networks

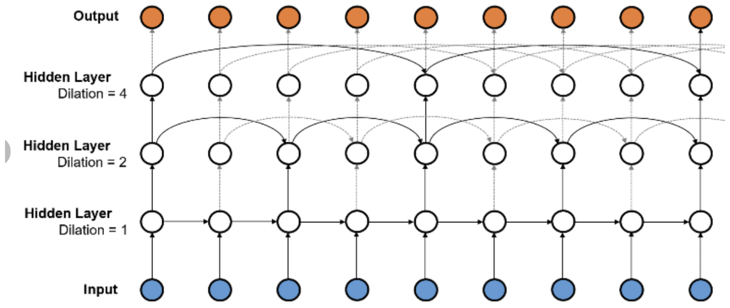


Figure 4: Long delay networks

Long Delay Networks (LDNs) help mitigate the vanishing gradient problem by introducing explicit delays in the network. These delays preserve the influence of earlier inputs over longer periods, ensuring that the gradients from distant time steps don't vanish. This allows the network to effectively capture and learn from long-term dependencies in sequential data.

Leaky Units

Leaky units in RNNs help retain information from previous time steps by mixing a portion of the current input with the previous hidden state, allowing information to "leak" over time. This helps mitigate the vanishing gradient problem by preserving memory for longer, enabling the network to handle long-term dependencies more effectively. Leaky units also provide smoother transitions between states, leading to more stable learning in sequential tasks.

Recall that:

$$S_t = \sigma(WS_{t-1} + Ux_t)$$

Consider:

$$S_{t,i} = \left(1 - \frac{1}{T_i}\right) S_{t-1} + \frac{1}{T_i}\sigma(WS_{t-1} + Ux_t)$$

$$1 < T_i < \infty$$

- $T_i = 1$, ordinary RNN.
- $T_i > 1$, gradients propagate more easily.
- $T_i \gg 1$, the state changes very slowly (extreme case), $S_{t,i} \approx S_{t-1}$.

The idea is to be more flexible by selecting T_i at each step; at some steps, we might forget, and at some, we might keep more.

Gated RNNs

Gated RNNs are a type of Recurrent Neural Network (RNN) that include mechanisms, called gates, to control the flow of information through the network. These gates help address the vanishing gradient problem and allow the network to better capture long-term dependencies in sequential data. The gates selectively allow information to be "forgotten" or "retained" at different time steps, making them more powerful for tasks involving long sequences.
For example, in *abb abab ab* sequence, to count how many "a" is in the sequence, we don't need to keep the whole past. We can count and forget at each time stamp.

Long Short-Term Memory (LSTM):

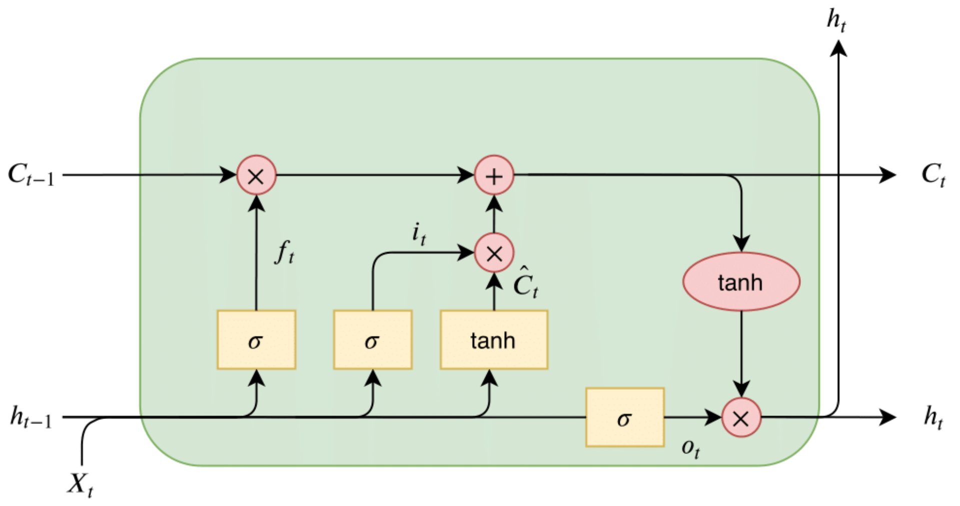


Figure 5: LSTM

LSTMs use three gates—**input gate**, **forget gate**, and **output gate**—to control the flow of information.

- The **input gate** i_t decides how much of the new input to let into the cell state.
- The **forget gate** f_t determines how much of the previous cell state to retain.
- The **output gate** o_t controls the final output at each time step.

The LSTM architecture helps maintain important information over long sequences and avoids the vanishing gradient problem by controlling the updates to the hidden state.

Allow each time step to modify:

LSTM Equations

The LSTM unit uses gates to control the flow of information through the cell. Here are the detailed equations for each gate and the final cell state update:

$$
\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\
\tilde{C}_t &= \tanh(W_C x_t + U_C h_{t-1} + b_C) \\
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\
h_t &= o_t \odot \tanh(C_t)
\end{aligned}
$$

Where:

- f_t: Forget gate activation at time step t
- i_t: Input gate activation at time step t
- $\tilde{C}_t$: Candidate cell state at time step t
- C_t: Cell state at time step t
- o_t: Output gate activation at time step t
- h_t: Hidden state (final output) at time step t
- x_t: Input at time step t
- h_{t-1}: Hidden state from the previous time step
- W_f, W_i, W_C, W_o: Weight matrices for the input x_t
- U_f, U_i, U_C, U_o: Weight matrices for the hidden state h_{t-1}
- b_f, b_i, b_C, b_o: Bias terms
- $\odot$: Element-wise (Hadamard) product
- σ: Sigmoid activation function
- tanh: Hyperbolic tangent activation function

LSTM Node

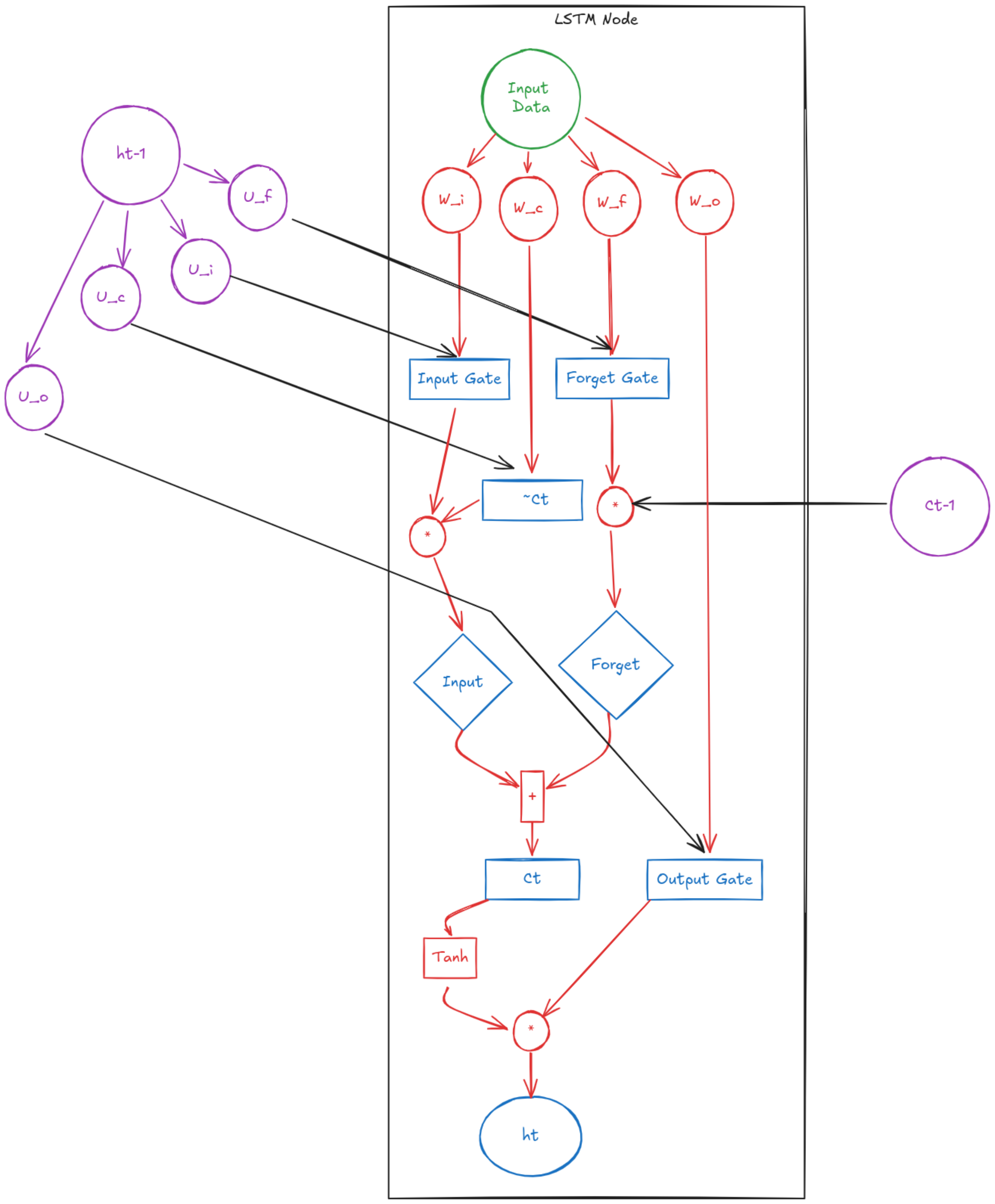


Figure 6: Breakdown of a LSTM Node

LSTM Layer

We will stack series of LSTM nodes together to form a layer:

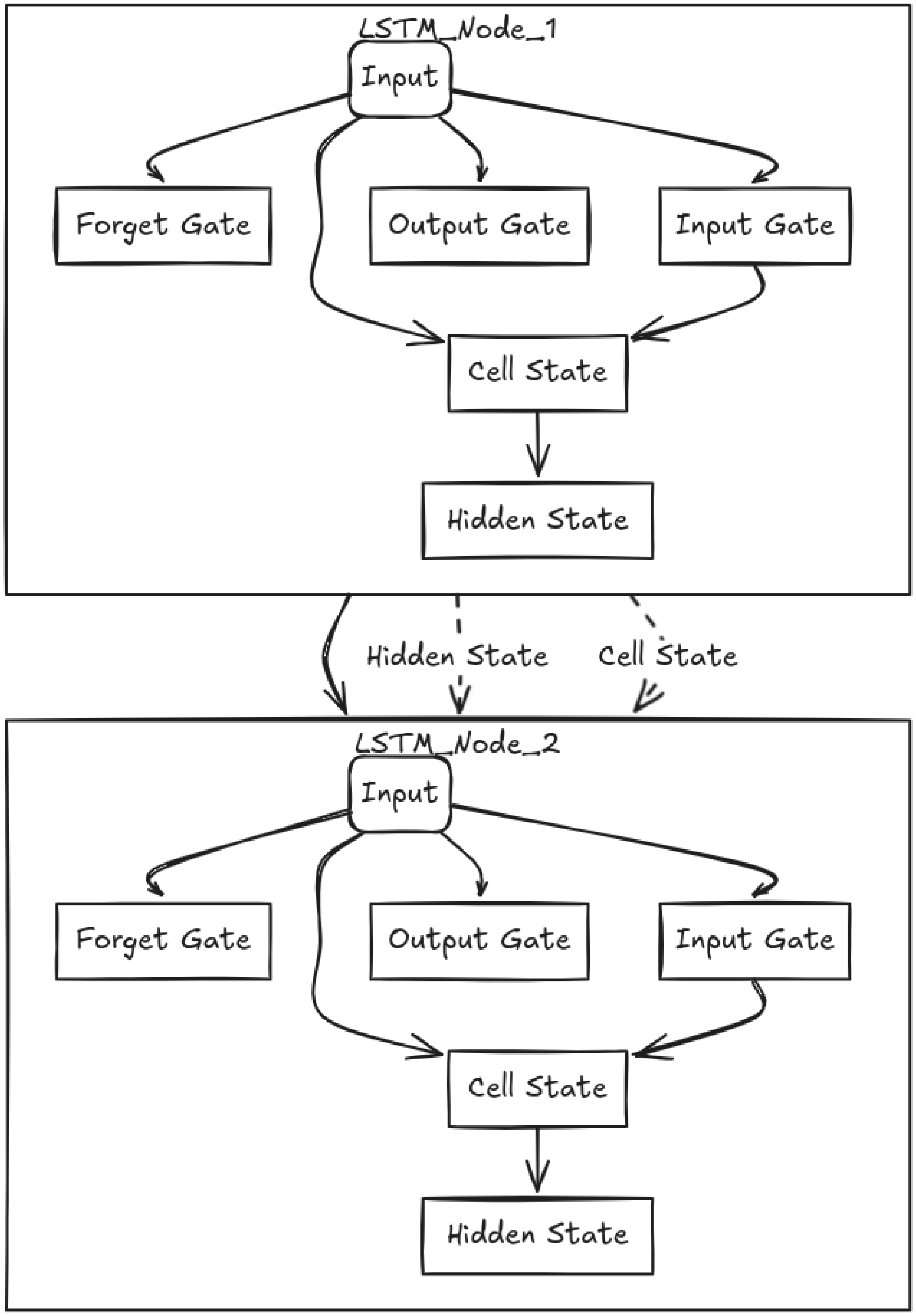


Figure 7: Example of an LSTM layer

Gated Recurrent Unit (GRU):

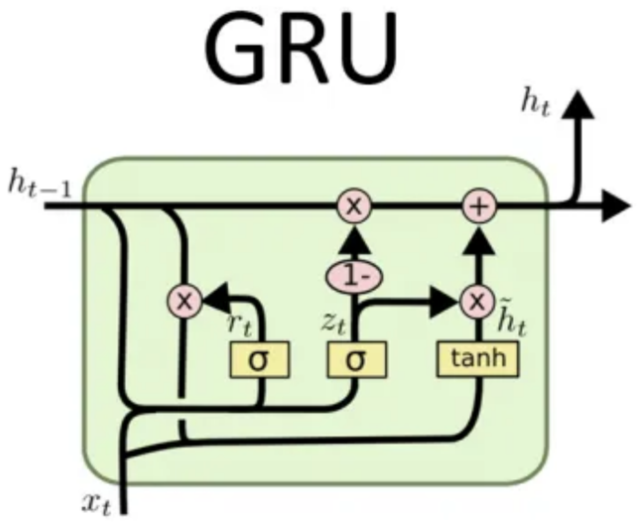


Figure 8: GRU

GRUs are a simplified version of LSTMs and have only two gates—**update gate** and **reset gate**.

- The **update gate** controls how much of the previous memory to retain.
- The **reset gate** determines how much of the past information to forget.

GRUs are more computationally efficient than LSTMs because they have fewer gates and perform well on tasks with moderate sequence lengths.
Standard RNNs:

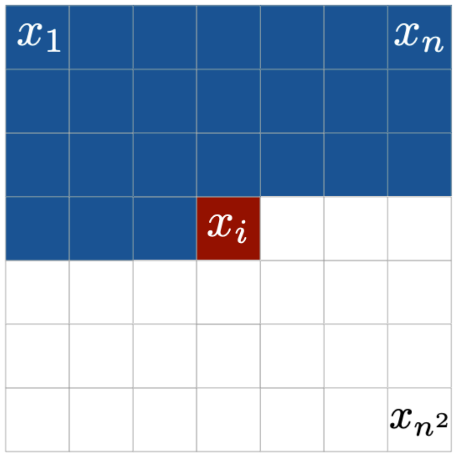
$$h_t = f(Wh_{t-1} + Ux_t)$$

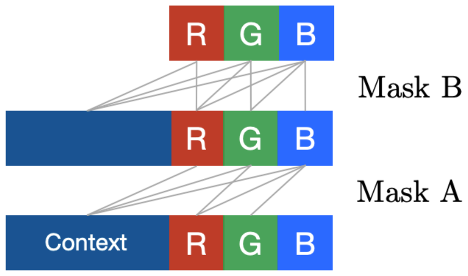
In GRU:

- **Update gate:**
$$z_t = \sigma(W^z x_t + U^z h_{t-1})$$
- **Reset gate:**
$$r_t = \sigma(W^r x_t + U^r h_{t-1})$$
- **New memory:**
$$\tilde{h}_t = \tanh(Wx_t + r_t \cdot Uh_{t-1})$$
(If reset gate is 0, then it forgets the past and only keeps the new data.)
- **Final memory:**
$$h_t = z_t \odot \tilde{h}_t + (1 - z_t) \cdot h_{t-1}$$
(Combines current and previous memory.)

Basically, similar to leaky units, by pairwise multiplication of r_t, z_t, and h_{t-1}, we make a decision on which node in h_{t-1} we carry information and which we forget.

Pixel RNN

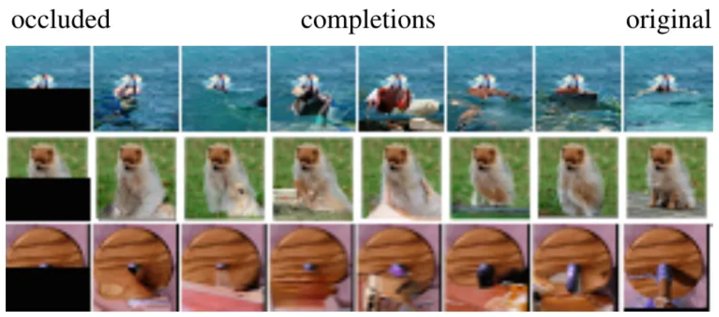




1. Model images pixel by pixel using raster scan order.
2. Each pixel conditional $p(x_t \mid x_{1:t-1})$ needs to specify 3 colors:

$$p(x_t \mid x_{1:t-1}) = p(x_t^{\text{red}} \mid x_{1:t-1})p(x_t^{\text{green}} \mid x_{1:t-1}, x_t^{\text{red}})p(x_t^{\text{blue}} \mid x_{1:t-1}, x_t^{\text{red}}, x_t^{\text{green}})$$

   and each conditional is a categorical random variable with 256 possible values.
3. Conditionals modeled using RNN variants (autoregressive approach). LSTMs + masking (like MADE).



Results in ImageNet downsampled. Very slow: sequential likelihood evaluation. For example, your prompt could be the image with missing pixels and the autoregressive model generate the next pixel and then continues.

Extension of Residual Networks to RNNs

Residual Networks (ResNets), originally developed for Convolutional Neural Networks (CNNs), tackle the vanishing gradient problem, which is also prevalent in Recurrent Neural Networks (RNNs).

Core Concept:
ResNets introduce ”skip connections” that allow gradients to bypass one or more layers, helping to maintain a strong gradient flow even in deep networks. This concept, while crucial for CNNs, is equally beneficial for RNNs, especially when modeling long sequences where vanishing gradients can be a significant issue.

Adaptation in RNNs:
In RNNs, similar skip connections can be utilized to enhance learning in deep recurrent architectures. This adaptation helps RNNs to better capture long-term dependencies without the degradation in performance typically associated with increased depth.

Mathematical Model:

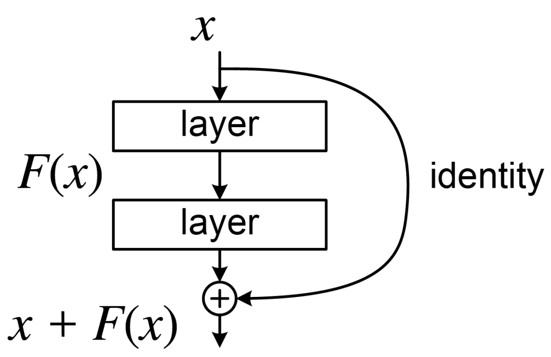


Figure 10: Typical Residual Block

The output of a residual block in an RNN can be expressed as:

$$y = F(x) + x,$$

where $F(x)$ represents the transformations (e.g., recurrent transformations) applied to the input x, and the addition of x ensures that the identity mapping is preserved, enhancing gradient flow and overall network training efficacy.

Impact:
Implementing residual connections in RNNs can lead to more robust models that are capable of learning effectively from long input sequences without performance degradation, thereby extending the powerful benefits of ResNets to the domain of sequential data processing.

1 Motivation

When we use a Recurrent Neural Network (RNN) to process sequences, each hidden state captures information only about the left context (e.g., words that come before). For example, in the sentence "terribly exciting! the movie was," the word "terribly" has its meaning affected by the word "exciting" which is in the right context. This can change the sentiment of the word "terribly" from negative to positive. In this case, we want to use both the left and right contexts to fully understand the meaning of each word in the sentence.

1.1 Task: Sentiment Classification

A sentence like "terribly exciting! the movie was" presents a good example where we need to capture both the left and right context of the word "terribly" to accurately classify the sentiment.

2 Bidirectional RNNs

Bidirectional RNNs solve the problem of missing right context by employing two RNNs:

- One RNN processes the sequence in the forward direction (left to right).
- The other RNN processes the sequence in the backward direction (right to left).

The hidden states of these two RNNs are concatenated, resulting in a representation that contains both left and right contexts. This allows us to understand how "terribly" is influenced by "exciting" even though "exciting" comes later in the sequence.

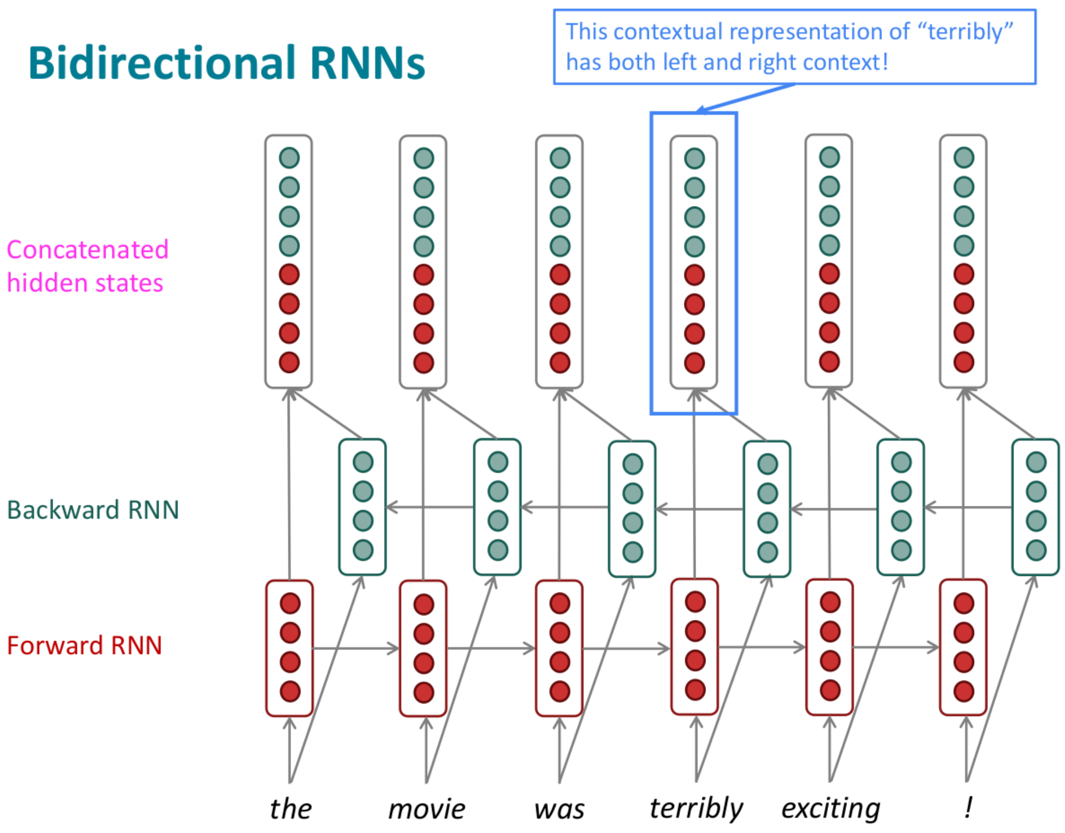

Bidirectional RNNs
This contextual representation of "terribly" has both left and right context!
Concatenated hidden states
Backward RNN
Forward RNN
the
movie
was
terribly
exciting
!

Figure 11: Bidirectional RNN example: The concatenated hidden states capture information from both the left and right contexts.

2.1 General Notation

- **Forward RNN**: Processes the sequence from the first word to the last.
- **Backward RNN**: Processes the sequence from the last word to the first.
- **Concatenated hidden states**: The final hidden state for each word is a combination of its forward and backward representations.

2.2 simplified Diagram: Bidirectional RNNs

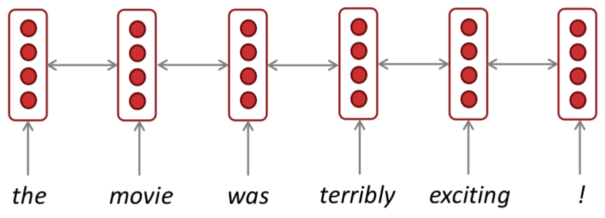


Figure 12: A simplified diagram of a Bidirectional RNN. The two-way arrows indicate bidirectionality, and the depicted hidden states are assumed to be the concatenated forwards+backwards states.

2.3 Limitations of Bidirectional RNNs

Bidirectional RNNs are only applicable when you have access to the entire input sequence. They are not suitable for tasks like Language Modeling (LM), where you only have access to the left context (since the future words are unknown). However, for tasks like encoding where you have the entire sequence, bidirectionality is powerful and should be used by default. For example, BERT (Bidirectional Encoder Representations from Transformers) is a powerful pretrained contextual representation system that relies on bidirectionality. You will learn more about transformers and BERT in a couple of weeks.

3 Multi-layer RNNs

RNNs are already ”deep” in one dimension as they unroll over many timesteps. However, we can make them deep in another dimension by stacking multiple RNNs on top of each other. This creates a **Multi-layer RNN**.

- This allows the network to compute more complex representations.
- Lower RNN layers should compute lower-level features, while higher RNN layers compute higher-level features.
- Multi-layer RNNs are also called *stacked RNNs*.

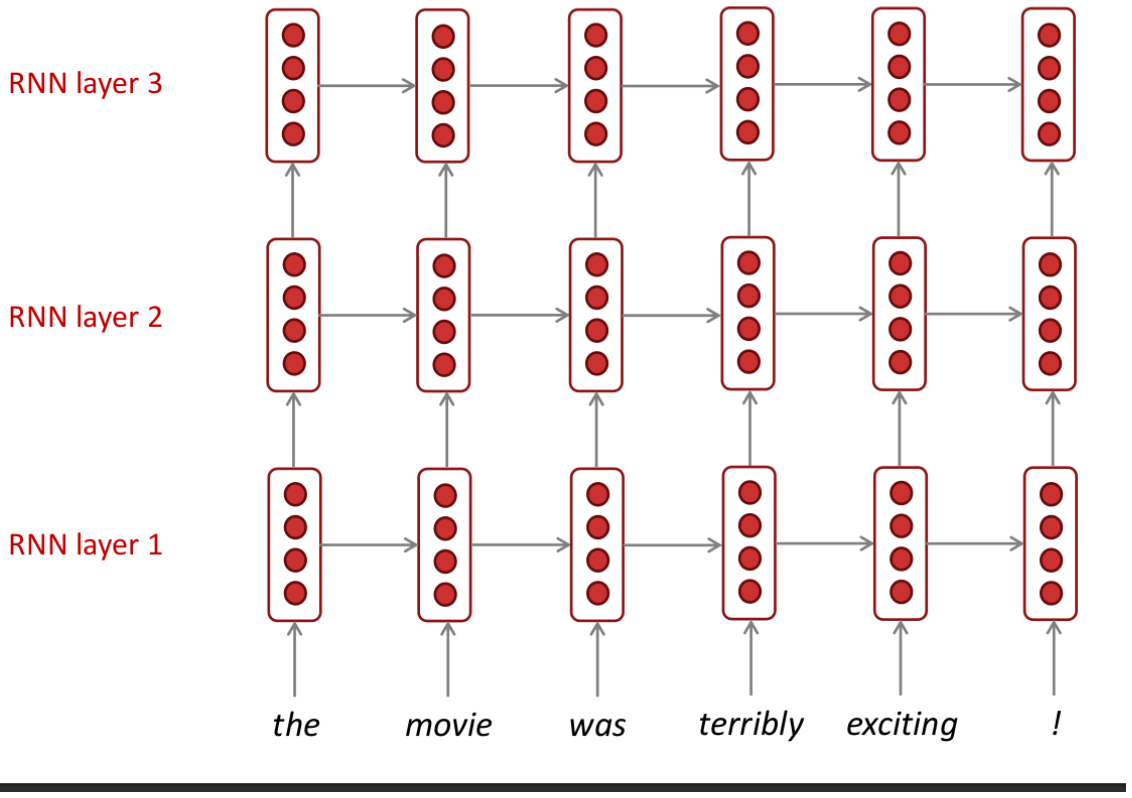


Figure 13: A simplified diagram of a Multi-layer RNN. Each RNN layer processes the sequence and passes the hidden states as input to the next layer.

3.1 Multi-layer RNNs in Practice

Multi-layer or stacked RNNs allow the network to compute more complex representations, which often leads to better performance compared to a single layer of high-dimensional encodings. For example:

- Lower layers compute lower-level features.

- Higher layers compute higher-level features.
- High-performing RNNs are usually multi-layered but not as deep as convolutional or feed-forward networks.

Example: In a 2017 paper, Britz et al. found that for Neural Machine Translation (NMT), 2 to 4 layers were best for the encoder RNN, and 4 layers were best for the decoder RNN. In practice:

- Often, 2 layers are significantly better than 1.
- 3 layers may offer slight improvements over 2.

However, as the number of layers increases, **skip-connections** (or dense connections) are typically required to ensure the model can be trained efficiently. For instance, deeper RNNs (e.g., with 8 layers) need such connections. Transformer-based networks, such as BERT, often contain many more layers (e.g., 12 or 24 layers) and incorporate numerous skip-connections.

- Transformer networks will be covered later, but they rely on many skip-connections to handle deeper architectures.

References

[1] Simon J.D. Prince. *Understanding Deep Learning*. MIT Press, 2023.

[2] Stanford NLP - cs224 - RNN