Sequence Modeling: Recurrent and Recursive Nets

---

1. What is an RNN and Why Do We Need It?

A **Recurrent Neural Network (RNN)** is a type of neural network specially designed to handle **sequential data** — things like text, speech, time-series, or any data where order matters.

Regular (feedforward) neural networks treat every input independently. That's fine for things like classifying a single image, but it breaks down when context matters. For example, the word "bank" means something different in "river bank" vs. "savings bank." A feedforward network can't capture that.

Simple Intuition

Think of reading a sentence:

> “The movie was not good”

To understand the word **“good”**, your brain remembers the earlier word **“not”**.
So, meaning depends on **previous words (context)**.

👉 RNNs work in the same way—they **remember past information** while processing new input.

Unlike normal neural networks (which treat inputs independently), an RNN:

- Takes **one input at a time**
- **Keeps memory** of previous inputs i.e, output from previous hidden unit is given as input to the current hidden unit
- Uses that memory to influence the current output. At the current state, output is calculated based on the current input and output from previous hidden unit .

The **hidden state (h)** is the “memory” of the network.

At each time step:

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

- h_t: current memory
- h_{t-1}: previous memory
- x_t: current input
- f: activation function (like tanh or ReLU)

So, **current state = previous memory + new input**

The key idea behind RNNs: parameter sharing. Instead of learning separate weights for each position in the sequence, RNNs reuse the same weights at every time step. This means:

- The model can handle sequences of different lengths
- It can generalize patterns regardless of where they appear in a sequence
- It learns far more efficiently than a network with separate parameters per position

A simple example: if a model reads "I went to Nepal in 2009" and "In 2009, I went to Nepal", it should recognize 2009 as the important piece of information in both cases. RNNs can do this because they share weights across positions.

Why is it called “Recurrent”? —- It is called “recurrent” because the same process repeats again and again for each step in the sequence.The state at time t depends on the state at time t−1.

How an **RNN processes the sentence?**

Convert words to numbers

RNNs cannot understand words directly, so each word is converted into a vector (embedding):

- The → x_1
- movie → x_2
- was → x_3
- not → x_4
- good → x_5

Process one word at a time

The RNN reads the sentence sequentially (left to right) and keeps updating its memory (hidden state).

Step 1: "The"

- Input: x_1
- Previous memory: none (start with zero)

👉 Memory now stores:

> "Some sentence has started"

Step 2: "movie"

- Input: x_2
- Memory updated using previous state

👉 Memory now:

> "Talking about a movie"

Step 3: "was"

- Input: x_3

👉 Memory:

> "Statement about a movie"

Step 4: "not"

- Input: x4x_4x4

👉 Memory becomes very important here:

> "Something negative is coming"

This word **changes the meaning direction**

Step 5: "good"

- Input: x5x_5x5

👉 Now RNN combines:

- "not" (negative signal)
- "good" (positive word)

Final memory:

> "The movie was NOT good" → overall **negative sentiment**

2. Unfolding Computational Graphs (Section 10.1)

What is a Computational Graph?

A computational graph is just a visual/mathematical way of showing how computations flow — inputs go in, operations happen, outputs come out.

What is Unfolding?

This turns the looping recurrence into a **straight chain** (a directed acyclic graph), which is much easier to work with mathematically. Crucially, the **same parameters θ are reused at every step** — this is parameter sharing in action.

Unfolding Computational Graphs

Unfolding a computational graph of a Recurrent Neural Network (RNN) is a way to clearly see how the network processes a sequence step by step over time. Normally, an RNN looks like a loop because it reuses the same computation again and again for each input in a sequence. But this loop can be confusing to understand, so we "unfold" it, meaning we stretch the loop across time into a straight line. Each step in this unfolded graph represents the RNN at a particular time step, where it takes the current input and the hidden state (memory) from the previous step, and produces a new hidden state and output. Even though it looks like many separate layers, all these steps share the same weights, which is why the model can generalize across sequences of different lengths. This unfolded view makes it easier to understand how information flows from earlier steps to later ones, and it is especially useful when training the network using backpropagation through time (BPTT), where errors are passed backward through all these time steps to update the shared weights.

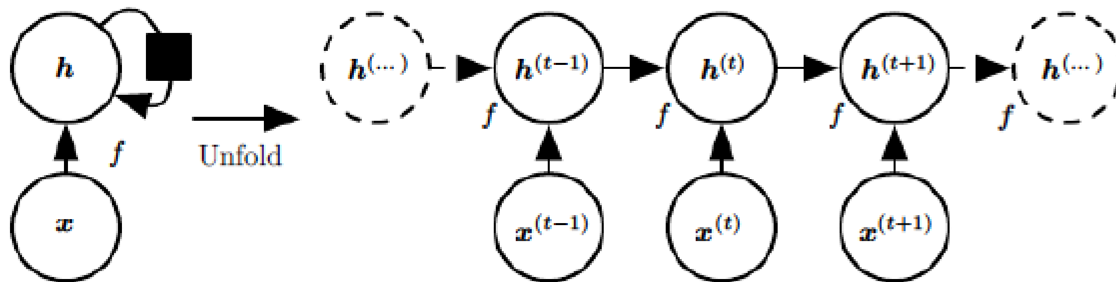


Figure 1

Why Does Unfolding Help?

Two major advantages:

1. The model always sees the same input "size" at each step (a state transitioning to the next state), regardless of how long the sequence actually is.
2. The same function f with the same parameters is applied at every time step — one model for all lengths and all positions.

At each time step, the RNN updates its memory by combining the previous hidden state and the current input. This hidden state acts like a compressed summary of everything seen so far, so ***it may lose some details.***

3. Recurrent Neural Networks in Detail

A Recurrent Neural Network (RNN) can be understood as a model that processes sequences step by step by repeatedly applying the same function over time. Instead of treating each input separately, the RNN keeps a hidden state (memory) that gets updated at every time step using both the previous state and the current input. When we unfold the network, it looks like a deep chain where each step depends on the previous one, but importantly, all steps share the same parameters, which helps the model handle sequences of different lengths. In a typical RNN, the input is first combined with the previous hidden state to produce a new hidden state, then this hidden state is used to generate an output at that time step. The model is trained by comparing each output with the correct target and summing the errors over all time steps. To learn, the RNN uses backpropagation through time, where errors are passed backward through the unfolded steps. Overall, an RNN works by gradually building a compressed summary of past inputs in its hidden state and using that summary to make predictions at each step, making it powerful for sequence data like text, speech, or time series.

Figure 2

Forward Propagation Equations

The most standard RNN does the following at each time step t:

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$

Meaning of the above equations:

- Compute pre-activation using previous hidden state and current input
- Apply tanh to get the new hidden state
- Compute the output
- Convert output to probabilities

The weight matrices are:

- **U** — input to hidden
- **W** — hidden to hidden (recurrent connection)
- **V** — hidden to output

The total loss across a full sequence is just the sum of per-step losses: $$L = \sum_t L^{(t)}$$

Three Common RNN Architectures

The book describes three important patterns:

Pattern 1 — Many-to-many with hidden-to-hidden connections: Every time step produces an output. Hidden units are connected to the next step's hidden units. This is the most powerful and most commonly discussed setup. It is Turing complete — it can theoretically compute anything a Turing machine can. The above figure 2 shows this.

Pattern 2 — Many-to-many with output-to-hidden connections: Instead of hidden units feeding into the next hidden state, the *output* at each time step feeds into the next hidden state. This is less powerful

(cannot simulate a Turing machine), but it has a major training advantage: time steps can be trained **in parallel** because they're decoupled from each other.

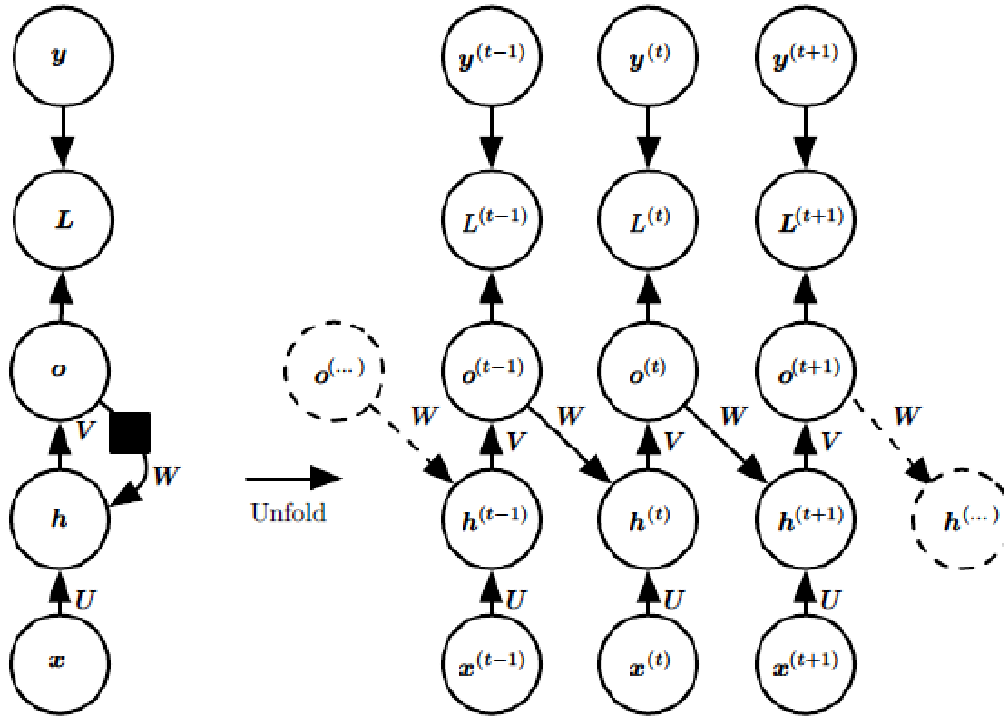
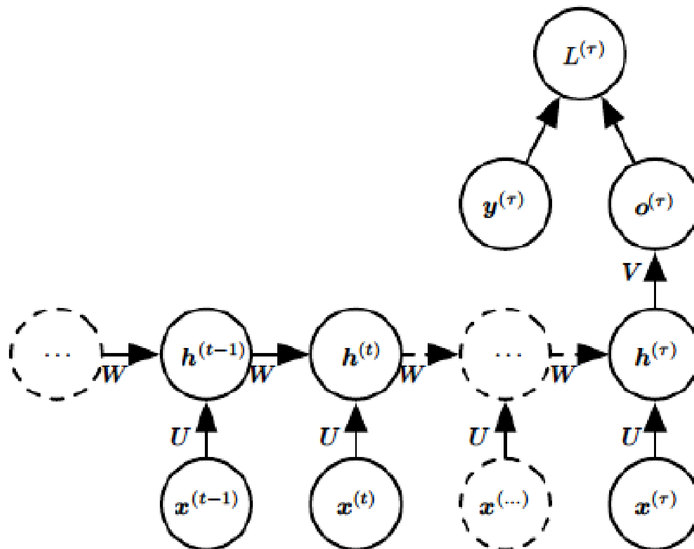


Figure 3

Pattern 3 — Many-to-one : The RNN reads the whole sequence and only produces **one output at the end**. Useful for tasks like sentiment classification of a full sentence.



Computing the Gradient — Backpropagation Through Time (BPTT)

To train the network, we compute gradients using **BPTT**. This works by:

1. Running the RNN forward through the sequence (left to right)
2. Computing the loss
3. Running backward through time (right to left) to propagate gradients

The computational cost is **O(τ)** (proportional to sequence length), and it **cannot be parallelized** because each step depends on the previous one. Memory cost is also O(τ) since all intermediate states must be stored.

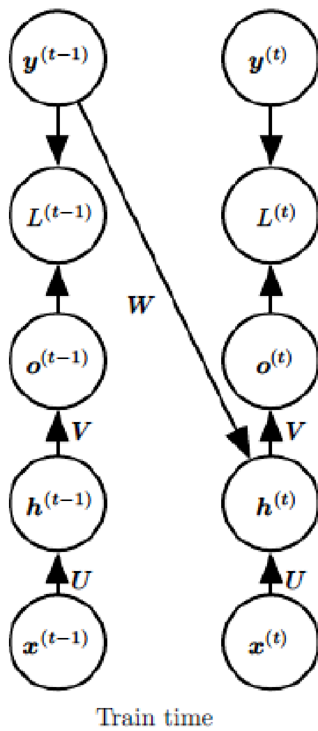
4. Teacher Forcing

The Problem

During training with the "output-to-hidden" architecture (Pattern 2 above), the network feeds its own output back in as the next input. But during early training, those outputs are wrong, which compounds errors over time.

The Solution: Teacher Forcing

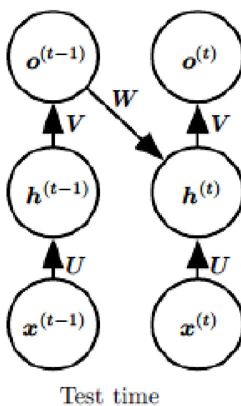
Teacher forcing is a training technique used in Recurrent Neural Networks where, instead of feeding the model's own predicted output back into the next time step, we give it the correct (actual) output from the training data. This helps the model learn faster and more accurately because it always receives the right context while learning. In simple terms, the model is "guided" at each step with the correct answer, so it doesn't get confused by its own mistakes and can focus on learning the correct patterns in the sequence.



Train time

The Downside

During training, we already have the correct outputs (labels) because they come from the dataset, so we can feed them into the model at each step. But during testing or real use, the model does not have access to the correct outputs anymore and must rely on its own predictions. If it makes a small mistake early, that error can carry forward and grow over time, leading to poor results. This mismatch between training (where it sees correct inputs) and testing (where it sees its own possibly wrong outputs) can reduce the model's performance.



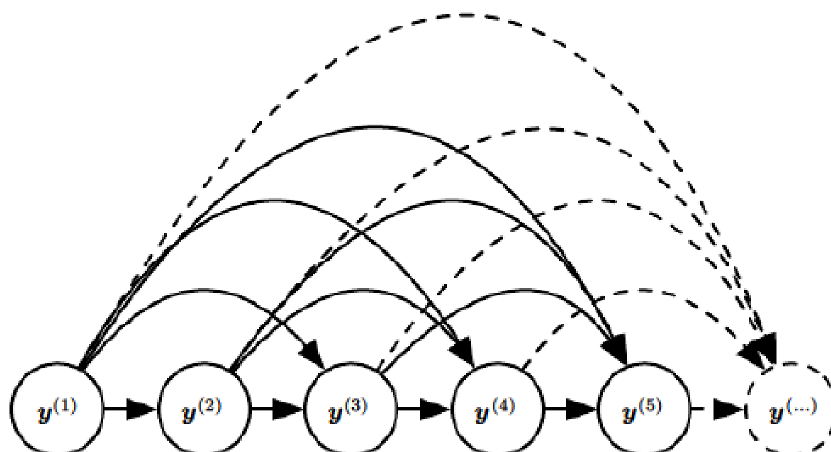
Test time

Computing the Gradient in a Recurrent Neural Network

Computing the gradient in a Recurrent Neural Network (RNN) is done using a method called backpropagation through time (BPTT), which is simply the usual backpropagation applied to the unfolded version of the network. Since the RNN is unrolled across multiple time steps, we first do a forward pass from the beginning of the sequence to the end to compute all hidden states, outputs, and losses. Then, we move backward through time, step by step, calculating how much each part of the network contributed to the final error. At each time step, the gradient flows not only through the current output but also through the hidden state connections that link different time steps, which means each state receives influence from both the present and the future steps. Because the same parameters are used at every time step, we add up (accumulate) the gradients from all time steps to update those shared weights. Overall, this process allows the RNN to learn from the entire sequence, but it can be computationally expensive and requires storing all intermediate states during the forward pass.

5. RNNs as Directed Graphical Models

RNNs can be seen as models that represent how each element in a sequence depends on previous elements. Instead of trying to model the entire sequence at once (which would be very complex and require too many parameters), an RNN breaks it into smaller steps, where each output depends on the past through a hidden state (memory). This hidden state acts like a bridge between the past and the future, carrying important information forward. In this way, the RNN models the probability of each element in the sequence based on what has already been seen, following a step-by-step (chain rule) approach. Because the same function and parameters are reused at every time step, the model remains efficient even for long sequences, unlike traditional models that would grow in size with sequence length. Overall, this view helps us understand that RNNs are not just neural networks, but also powerful probabilistic models that can capture dependencies across a sequence in an efficient and structured way.



Why RNNs Are Efficient Graphical Models

A naive graphical model over a long sequence would need **$O(k^\tau)$** parameters (exponential in length). Because RNNs share parameters across time steps, they need only **O(1)** parameters as a function of sequence length. The hidden state $h^{(t)}$ acts as a compact "bridge" between past and future, allowing long-range dependencies without storing everything explicitly.

Determining Sequence Length

During generation, the RNN needs to know when to stop. Three strategies:

1. Add a special **end-of-sequence token** to the vocabulary
2. Add a **binary output** (sigmoid unit) that predicts whether to continue or stop at each step
3. **Predict the total length τ** directly as a separate output

6. Modeling with Context: Conditional RNNs

The idea is to use an RNN not just to model a sequence by itself, but to generate or predict a sequence **based on some extra information (context)**. This context can be another input, like a single vector (for example, an image or a feature vector) or even another sequence. The RNN uses this context along with its hidden state to guide what it outputs at each time step. For example, in tasks like image captioning, the image is given as context, and the RNN generates a sequence of words describing it. The context can be given to the model in different ways, such as feeding it at every time step or using it to initialize the hidden state at the beginning. Because of this, the RNN learns to produce outputs that are not just based on past sequence elements, but also influenced by the given context. Overall, this makes RNNs very powerful for tasks where the output depends on some external information, like translation, caption generation, or question answering.

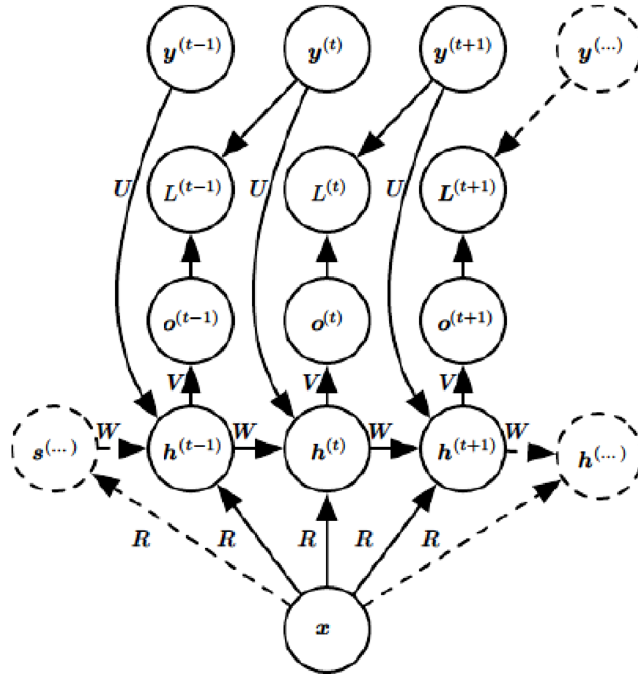


Figure 10.9: An RNN that maps a fixed-length vector $\boldsymbol{x}$ into a distribution over sequences $\mathbf{Y}$. This RNN is appropriate for tasks such as image captioning, where a single image is used as input to a model that then produces a sequence of words describing the image. Each element $\boldsymbol{y}^{(t)}$ of the observed output sequence serves both as input (for the current time step) and, during training, as target (for the previous time step).

If the context is a fixed-size vector x, common approaches include:

- Providing x as **extra input at every time step**
- Using x as the **initial hidden state h^(0)**
- **Both** of the above

When a separate weight matrix R is introduced for the context, we can think of x^T R as a learned bias term that shifts the hidden unit behavior based on the input context.

7. Bidirectional RNNs

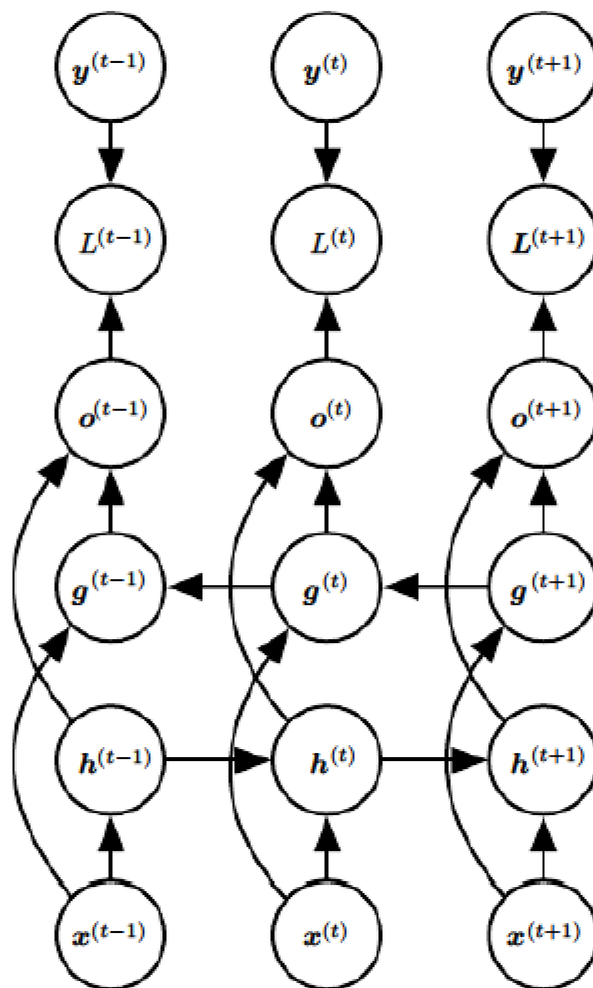
Bidirectional RNNs are introduced as an improvement over normal RNNs that only look at past information. In a standard RNN, at any time step, the model can only use the inputs it has already seen (past), but in many real-world tasks, the future context is also important to understand the current input.

For example, in speech recognition, the correct phoneme for the current sound might depend on the next few words.example: I love apple as it is good for health and I love apple because its features are good.

Bidirectional RNNs solve this by using two RNNs: one processes the sequence from left to right (past to future), and the other processes it from right to left (future to past).

Bidirectional RNNs run **two RNNs simultaneously**:

- h^(t): reads the sequence **forward** (left to right)
- g^(t): reads the sequence **backward** (right to left)



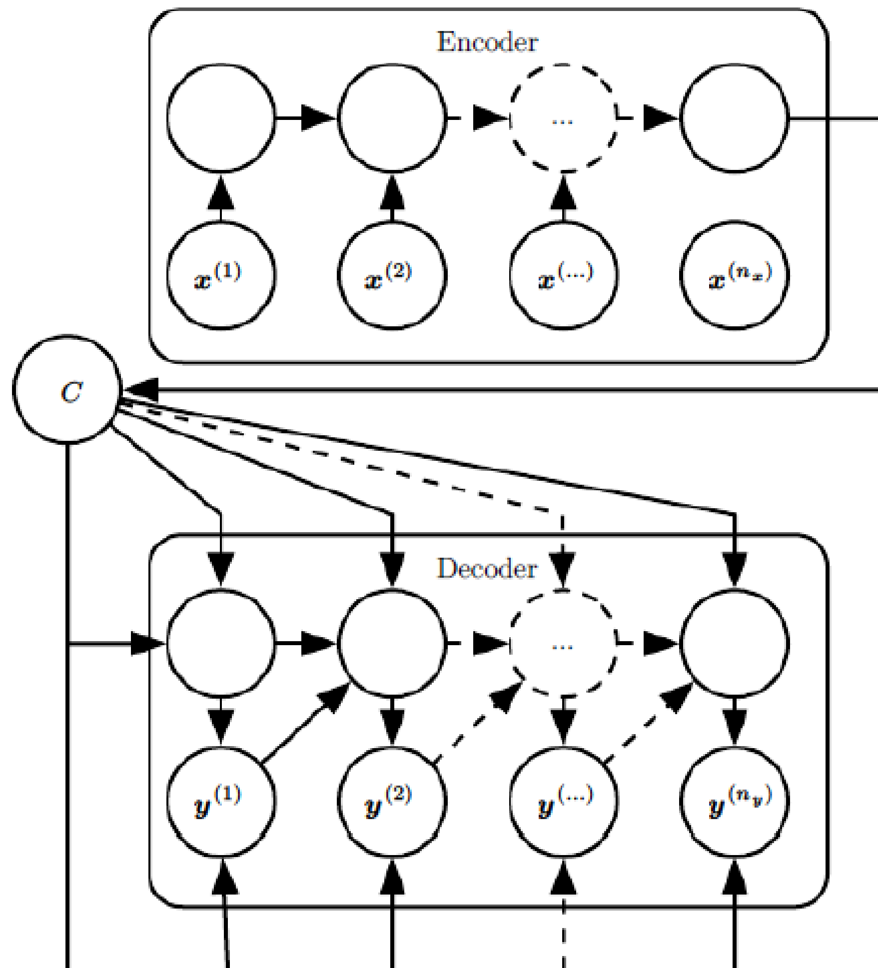
The output at each time step is then computed from both h^(t) and g^(t), giving access to context from both past and future without needing a fixed window size. This has been highly successful in handwriting recognition, speech recognition, and bioinformatics.

8. Encoder-Decoder (Sequence-to-Sequence) Architecture

All previous architectures required input and output sequences to be the **same length**. That's impractical for, say, translation ("Hello" in English → "Hola" in Spanish is fine, but longer sentences are different lengths).

The **encoder-decoder** architecture solves this:

1. **Encoder RNN** reads the entire input sequence x^(1), ..., x^(nx) and produces a context vector **C** (typically the final hidden state h^(nx))
2. **Decoder RNN** takes C as input and generates the output sequence y^(1), ..., y^(ny) of potentially different length

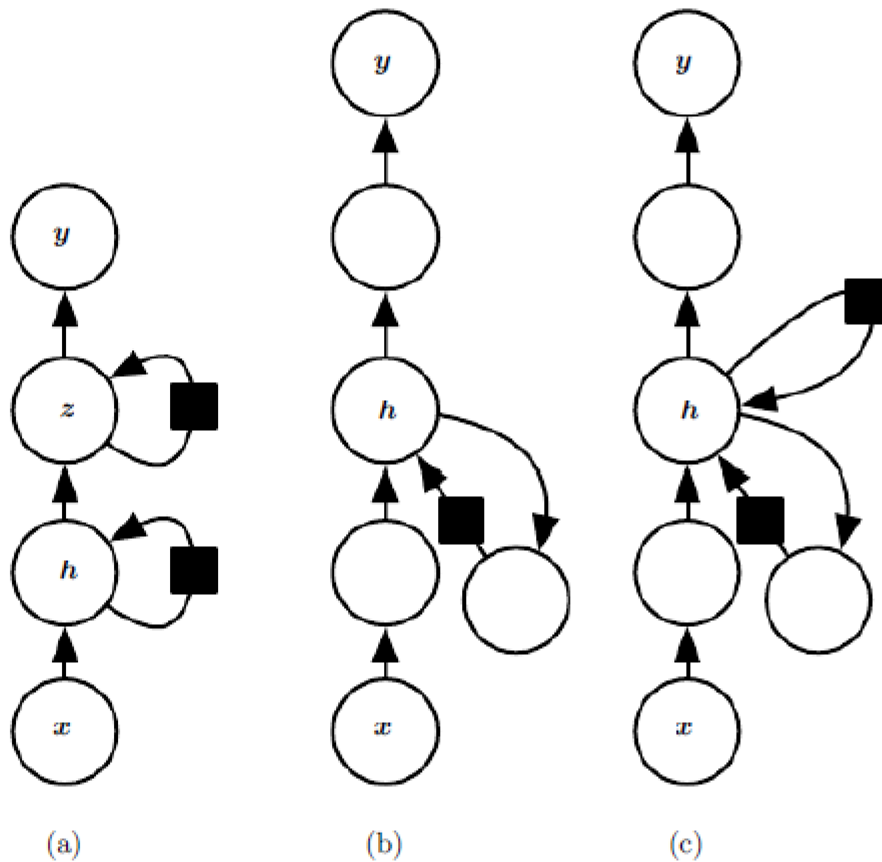


These architectures explain how RNNs can handle tasks where the input and output sequences can have different lengths, like language translation or question answering. The idea is simple: one RNN, called the encoder, reads the entire input sequence and converts it into a fixed-size summary called a context vector, which captures the overall meaning of the input. Then another RNN, called the decoder, uses this context to generate the output sequence step by step. The decoder predicts each output based on the context and the previous outputs, gradually building the full sequence. This approach is powerful because it separates understanding (encoding) from generation (decoding), but it also has a limitation—if the input sequence is very long, compressing everything into a single fixed-size vector may lose some important information.

Limitation: If the input sequence is very long, forcing all of it into a fixed-size C loses information. This motivated **attention mechanisms**, where C becomes a variable-length sequence and the decoder learns to focus on relevant parts of the input at each step.

9. Deep Recurrent Networks

Deep Recurrent Networks extend the basic RNN idea by adding more layers or depth to improve the model's ability to learn complex patterns. In a simple RNN, there are three main parts: input to hidden, hidden to hidden (across time), and hidden to output, and each of these is usually just a single transformation. In deep RNNs, we make these parts more powerful by stacking multiple layers or using deeper transformations (like small neural networks inside each step). This helps the model learn richer and more detailed representations of the data, similar to how deep neural networks work in other areas. However, adding depth also makes training harder because the information has to pass through more layers and time steps, which can slow learning and make optimization difficult. So, while deep RNNs are more powerful, they require careful design to balance complexity and training efficiency.



Three approaches to making RNNs deep:

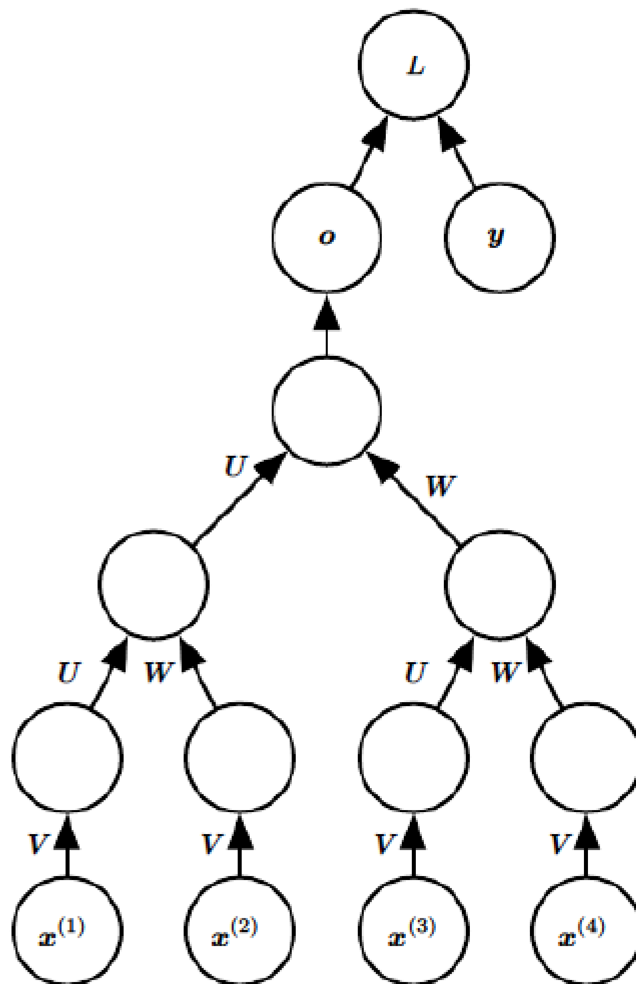
1. **Hierarchical hidden states** — break the hidden state into multiple layers stacked on top of each other
2. **Deep transition functions** — use a small MLP for each of the three connection types (makes paths between time steps longer, harder to optimize)
3. **Skip connections** — add shortcut connections in the hidden-to-hidden path to mitigate the path-lengthening problem

---

10. Recursive Neural Networks (Section 10.6)

While RNNs process sequences as **chains**, **recursive neural networks** process data as **trees**.

Recursive Neural Networks are introduced as a variation of recurrent models that work on **tree-like structures instead of simple sequences**. While regular RNNs process data step by step in a chain (like words in a sentence), recursive networks combine inputs in a hierarchical way, building a structure similar to a tree. For example, in language processing, they can combine words into phrases, and then phrases into a full sentence, capturing deeper relationships between parts of the input. At each node of the tree, the same set of weights is reused, which makes the model efficient and consistent. One advantage of this approach is that it can represent long-range relationships more effectively with fewer steps compared to standard RNNs. Overall, recursive neural networks are useful when the data naturally has a hierarchical structure, such as sentences, parse trees, or certain types of structured data.



Instead of $h^{(t)}$ being computed from $h^{(t-1)}$ one step at a time, a recursive net computes node representations by combining children nodes in a tree structure. The same weight matrices (U, V, W) are used at every node.

Key advantage: For a sequence of length τ, an RNN has depth τ, but a balanced binary tree has depth only **$O(\log \tau)$**. This dramatically reduces the vanishing/exploding gradient problem for long sequences.

Used successfully in natural language processing (parsing, sentiment analysis) and computer vision. The tree structure can be fixed (e.g., a parse tree from a language parser) or in theory learned by the model itself.

11. The Challenge of Long-Term Dependencies

The main challenge is how RNNs struggle to learn relationships between events that are far apart in a sequence, called long-term dependencies. The problem happens during training when gradients are passed backward through many time steps using backpropagation through time. As this happens, the gradients can become very small (vanishing gradients eg: at time state t, if the weight is 0.2, at time state t+1 it is 0.2*0.2=0.04,at t+2 it is 0.04*0.2=0.008 and so on) or very large (exploding gradients eg: at time state t, if the weight is 2, at time state t+1 it is 2*2=4,at t+2 it is 4*2=8 and so on), making learning unstable or ineffective. Because of this, the network tends to focus more on recent inputs and may "forget" important information from the distant past. This makes it difficult for basic RNNs to capture long-range patterns, like remembering a word that appeared much earlier in a sentence. Overall, while RNNs are designed to handle sequences, this limitation makes it hard for them to learn long-term relationships unless special techniques or architectures (like LSTM or GRU) are used.

14. LSTM — Long Short-Term Memory (Section 10.10)

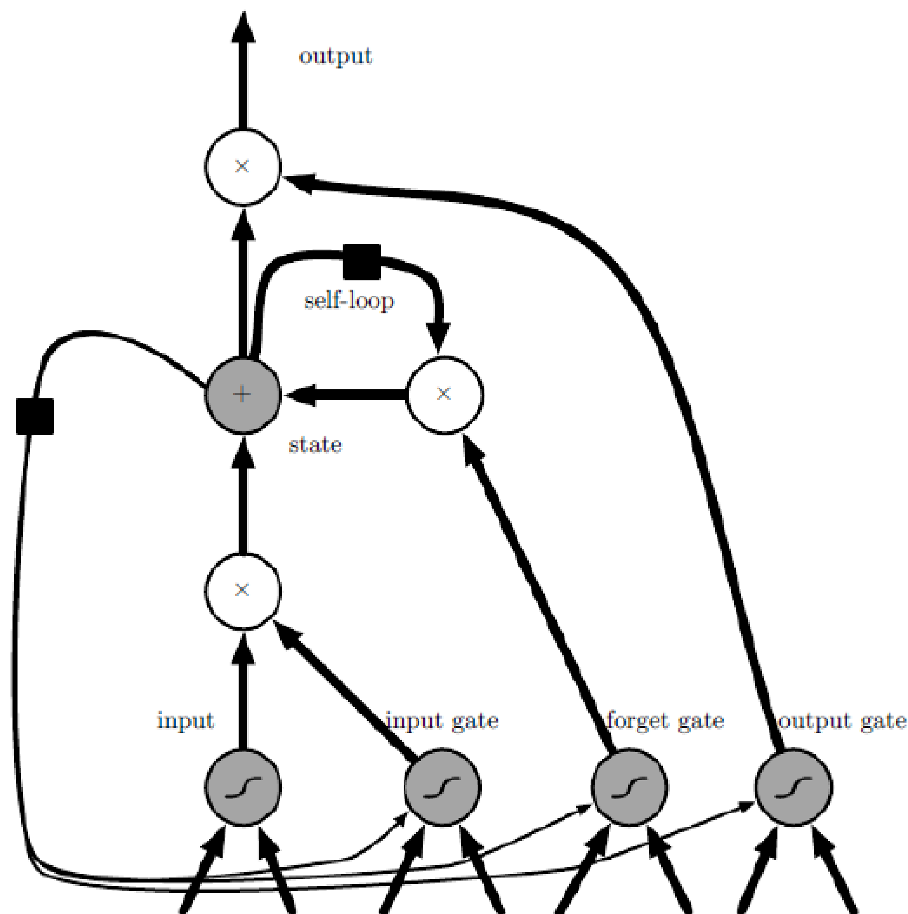
Why LSTM?

LSTMs were designed specifically to solve the vanishing gradient problem by creating **self-loops with learned, adaptive gating**. The key insight: the weight of the self-loop is not fixed — it is controlled by a learned gate, allowing the network to decide dynamically how long to retain information.

Long Short-Term Memory (LSTM):

LSTM is a special type of RNN designed to solve the problem of long-term dependencies. Instead of

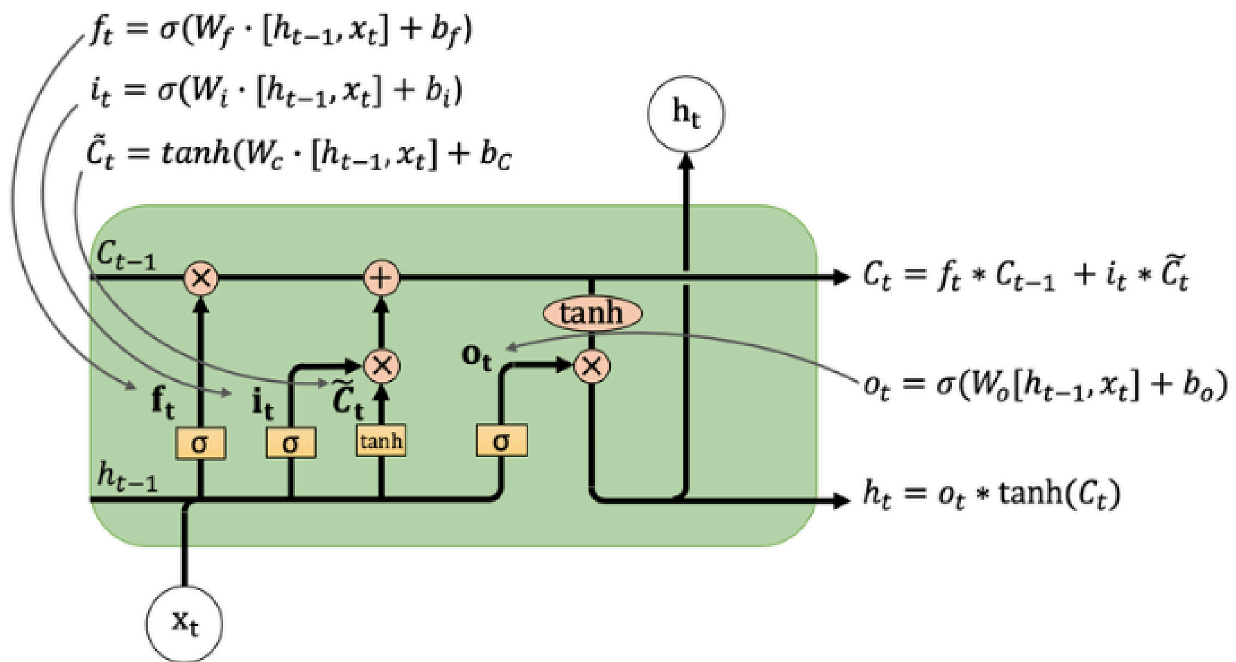
using a simple hidden state, it introduces a memory cell that can store information for a long time and control what to keep or forget using gates. These gates act like regulators that decide how information flows through the network, helping it remember important details and ignore unnecessary ones. Because of this controlled memory mechanism, LSTMs can learn patterns over long sequences much better than basic RNNs.



The LSTM Cell

Each LSTM cell contains:

- **Forget gate:** decides what information from the previous memory should be removed
- **Input gate:** decides what new information should be added to the memory
- **Output gate:** decides what part of the memory should be used to produce the current output



All gates use **sigmoid** activations (output between 0 and 1). A sigmoid output of 1 = "open the gate fully"; 0 = "close it completely."

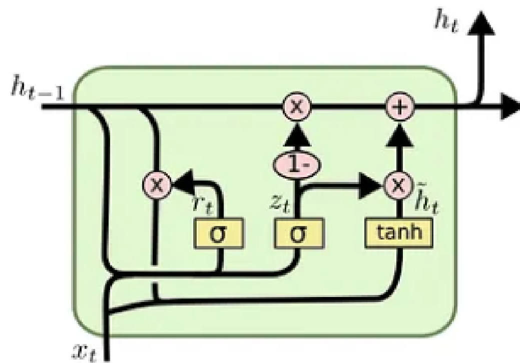
Applications

LSTM works very well and has been used successfully in tasks like

- speech recognition
- handwriting recognition
- machine translation
- image captioning
- language parsing.

Gated RNN (like GRU):

Gated RNNs, especially the Gated Recurrent Unit (GRU), are a simpler version of LSTM that also use gates to control information flow but with fewer components. Instead of having a separate memory cell, GRU directly updates its hidden state using gates. This makes it easier to train and faster while still handling long-term dependencies better than a basic RNN. Even though it is simpler, GRU often performs as well as LSTM in many tasks.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Main gates in GRU:

- **Update gate:** decides how much of the past information to keep
- **Reset gate:** decides how much of the past information to forget when adding new input

Summary Table of All Key Concepts

Topic	Key Idea
RNN	Neural network for sequences; shares weights across time steps
Unfolding	Turns recurrence into a DAG; enables BPTT
Hidden state h^(t)	Lossy compressed summary of the past
BPTT	Backprop through the unrolled graph; O(τ) cost, sequential
Teacher Forcing	Feed true targets during training instead of model outputs
Bidirectional RNN	Two RNNs reading forward and backward; uses past and future context
Encoder-Decoder	Handles variable-length input→output (e.g., translation)

Deep RNN	Multiple layers in the recurrent transitions; more expressive
Recursive Net	Tree-structured graph; depth $O(\log \tau)$ instead of $O(\tau)$
Vanishing/Exploding Gradient	Core challenge; gradients shrink or grow exponentially over time
LSTM	Gated cell state; forget, input, output gates; solves vanishing gradient
GRU	Simpler LSTM variant with update and reset gates