**Problem1: Predict the Next Word Using Context**

Let’s say we have the sentence: "I love to eat"

We want the model to **predict the next word** — e.g., "pizza" or "food" — based on the **previous words**.This is a **language modeling** problem: Given a sequence of words, predict the most probable next word.

**Why Use an RNN?**

Because **RNNs (Recurrent Neural Networks)** can:

* Process sequences word by word
* **Remember previous words** (like "I love to...")
* Make predictions **based on what came before**

This is useful for Text generation, Speech recognition, Time series forecasting, Sentiment analysis (if trained on full sequences)

**How RNN Works — A Simple View:** Imagine this:

1. The RNN reads "I" → updates its memory
2. Then it reads "love" → updates memory again
3. Then "to" → it knows so far: "I love to"
4. Then "eat" → its memory now reflects the context
5. It predicts the next word — maybe "pizza"

Each step **passes context forward** using a hidden state h.

**Algorithmically** For each time step t:

Input: word\_t, hidden\_t-1

Output: hidden\_t = tanh(W \* word\_t + U \* hidden\_t-1 + b)

Then Output prediction = softmax(V \* hidden\_t), Where W, U, V are learnable matrices, hidden\_t is the **memory** of the RNN, softmax gives the probability of the next word

**Memory Example** Say the input is: "I love to"**.** Let’s feed it word-by-word into the RNN. After reading "I", "love", and "to", it should learn:

"I love to" → most likely followed by "eat"

So: Input: ["I", "love", "to"]

Output: "eat"

This is what RNNs are good at: using **sequence memory** to make a decision.

**Summary of Our Tiny RNN Task**

| **Step** | **Description** |
| --- | --- |
| **Input** | Short word sequences (like "I love to") |
| **Output** | Predict next word ("eat") |
| **Model** | Simple RNN |
| **Learning** | Adjust weights so the RNN can remember context |

**Problem2: Sentiment Classification from Movie Reviews**

**Task:** Given a **sequence of words in a movie review**, predict whether the **sentiment is positive or negative**. **Example:**

| **Review Text** | **Sentiment** |
| --- | --- |
| "This movie was absolutely great!" | 👍 Positive |
| "It was a total waste of time." | 👎 Negative |

The challenge here is **meaning depends on word order,** "not bad" is positive, but "bad" alone is negative

**Why RNN:** RNN reads the sentence **one word at a time**, keeping a hidden memory of the past. RNNs **remember what came before** — crucial to detect things like "not" before "good", like:

1. "This" → neutral
2. "movie" → still neutral
3. "was" → okay
4. "absolutely" → maybe positive
5. "great" → now we’re sure it’s positive!

So RNN builds a **contextual understanding** before deciding.

**Model Input** is "This movie was absolutely great" **and output** 0.93 → interpreted as **positive** sentiment**.** It works well because it doesn’t just look at individual words — it understands them **in order**.

**Algorithm**

RNN does: h\_t = tanh(Wx \* x\_t + Wh \* h\_t-1 + b) # hidden state update

After all words are read, we pass the final state to a **Dense layer**:

output = sigmoid(V \* h\_T) # binary classification (positive or negative)

**Summary**

| **Aspect** | **Value** |
| --- | --- |
| **Problem** | Binary classification (sentiment) |
| **Input** | Sequence of word tokens |
| **Output** | 0 or 1 (Negative or Positive) |
| **Why RNN?** | Understands word order + context |
| **Prediction Point** | After reading the **whole** sequence |

**Problem: Speech-to-Text (Automatic Speech Recognition)**

Given an audio waveform of someone speaking, the goal is to generate the corresponding text transcript — e.g., "I need to book a flight to Delhi."

**Why is this hard?**

* **Audio is a continuous time series** — not just text.
* One word can stretch across many time frames (variable length).
* The number of audio samples ≠ number of output words.
* Requires **memory of past frames** (context).
* Words can depend on **what was said before** ("I want to" → "eat" vs "sleep").

**Why RNN (esp. LSTM/GRU)?**

RNNs (and later LSTMs/GRUs) are **made for sequential dependencies** — exactly what speech has:

1. Audio is broken into **time frames**
2. Each frame is **a feature vector** (e.g., MFCCs)
3. The RNN processes each frame one by one, **retaining context**

Without memory (as in CNNs or vanilla MLPs), you'd lose the ability to know **what's happening over time**.

**Why RNN is (was) the Go-To Solution**

| **Problem Aspect** | **Why RNN Solves It** |
| --- | --- |
| Variable-length input | RNNs can handle sequences of any length |
| Temporal dependency | They **remember** what happened before |
| Sequence output | RNNs can output sequences (not just one label) |
| Streaming input | Can predict word-by-word as audio streams in |

Without RNNs or their successors (LSTM, GRU, Transformers), solving this was near impossible using classical ML.

**Real-World Use:** Google’s early speech-to-text (e.g., in Android voice typing), Apple Siri, and Amazon Alexa **all started with LSTM-based RNNs** before shifting to Transformer-based models like Whisper or Conformer.

**Specific Problem: *Phoneme-Level Speech-to-Text (Simplified)***

Given a sequence of audio-derived phonemes (sound units), predict the **corresponding word**.

**Example:** Suppose you have audio preprocessed into **phoneme sequences**, like:

Input: ["th", "ih", "s"] → Output: "this"

Input: ["k", "ae", "t"] → Output: "cat"

Input: ["d", "ao", "g"] → Output: "dog"

This simulates a **sequence-to-one mapping**: given time-series **audio segments (phonemes)**, predict the **spoken word**.

**Why RNN?**

* **Order matters**: "k-ae-t" ≠ "t-ae-k"
* **Variable length**: Some words have 2 phonemes, some have 4+
* **Need memory**: The output depends on the whole input, not just the last sound

RNN processes phonemes one-by-one, updates its hidden state, and after the full sequence, it outputs the word class.

**Algorithmic Solution Using RNN -** Let’s break this down:

**1. Data Setup:** Input: sequences of phonemes, each mapped to a vector (like word embeddings)andOutput: a class label (e.g., class 0: "cat", class 1: "dog", etc.)

Input: [ [13], [7], [24] ] - one-hot or index-encoded phonemes and Target: 0, e.g., label for "cat"

**2. Model:** Input: sequence of phoneme embeddings → RNN → Dense → Softmax

**3. Step-by-Step RNN Computation:** Let each phoneme vector be xₜ, and hidden state be hₜ.

For time steps t = 1 to T: hₜ = tanh(Wₓ \* xₜ + Wₕ \* hₜ₋₁ + b)

After the last time step T, the final state h\_T is passed to a Dense + Softmax layer to classify the word: ŷ = softmax(V \* h\_T + c), where:

* Wₓ, Wₕ, V, b, c are learnable weights
* ŷ is a probability distribution over known words
* Loss: **categorical crossentropy** between ŷ and true word class

**Summary of the RNN Speech Word Classifier**

| **Component** | **Role** |
| --- | --- |
| Input | Sequence of phonemes (1-hot or embedding) |
| Model | RNN → Dense → Softmax |
| Output | Single word label |
| Why RNN? | Remembers phoneme sequence order |
| Loss | Categorical crossentropy |
| Output Format | Classification |