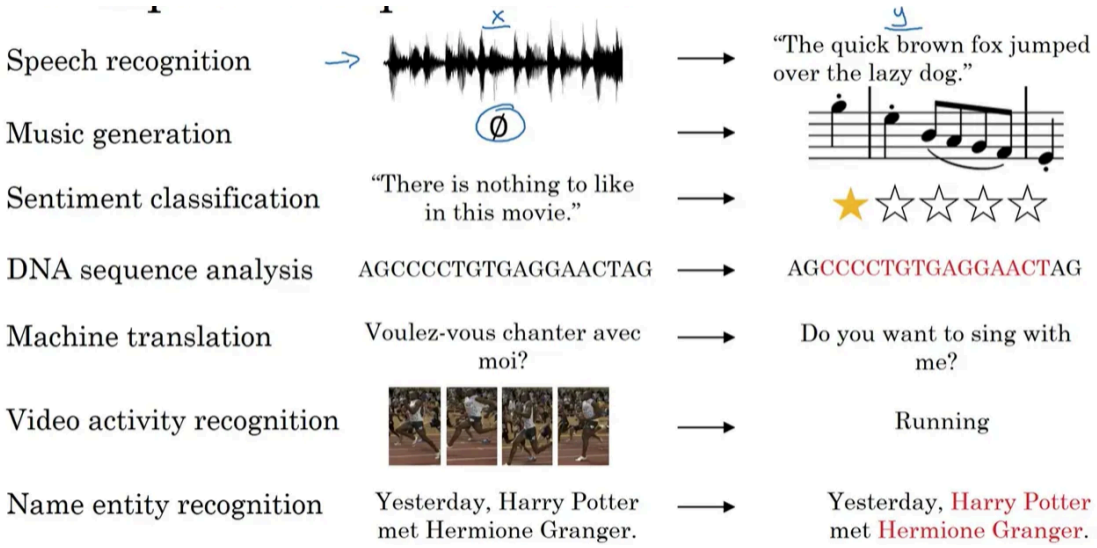
WEEK 1: Recurrent Neural Network (RNN).

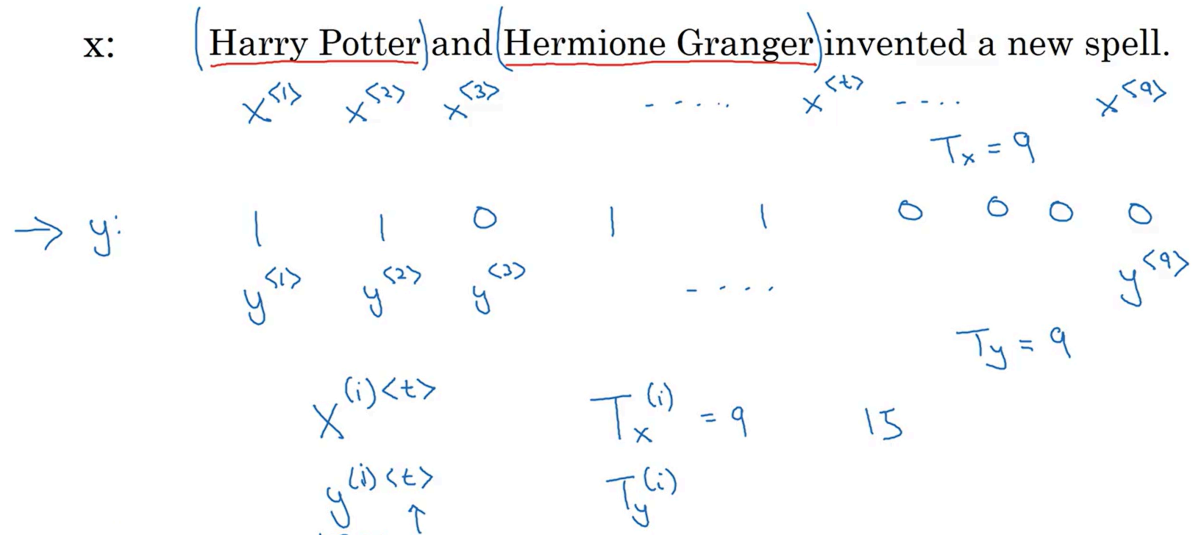
Why Sequence Models?

- **Sequence models** (like RNNs) are designed for data where order matters, such as text, speech, or video.
- They are used in real-life for tasks like **speech recognition**, **language translation**, **music generation**, and **analyzing DNA sequences**.
- Sequence models can handle inputs and outputs of varying lengths (like translating a sentence from English to French).
- Unlike traditional models, they remember information from earlier in the sequence to make better predictions later on.
- Everyday examples: **voice assistants** turning speech into text, Google Translate, or even recommending music based on your listening history.
- Examples:

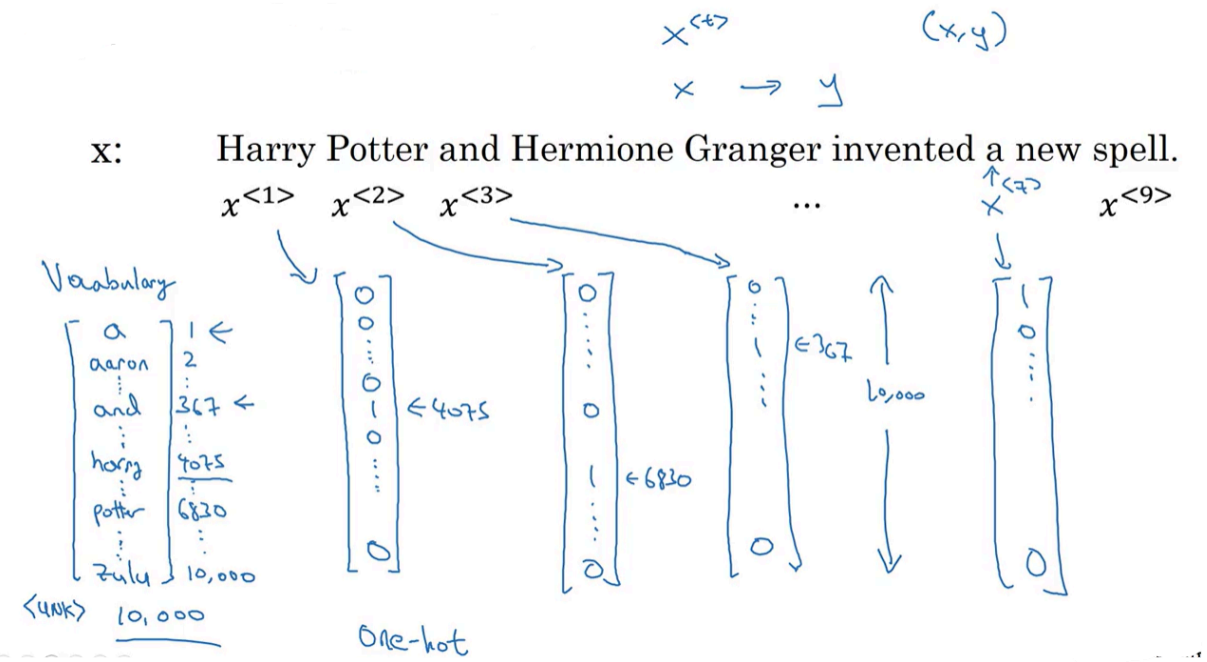


Notation

Example: Named Entity Recognition (NER)



Representing Words:

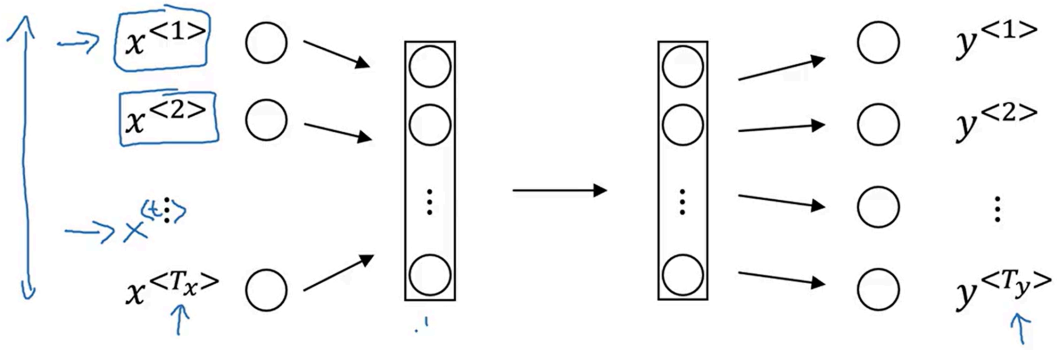


- **Input Sequence Notation**
  - $x^{\langle t \rangle}$: The input at position t in the sequence (for general indexing of words in a sentence)
  - $x^{\langle 1 \rangle}, x^{\langle 2 \rangle}, \ldots, x^{\langle T_x \rangle}$: Indexed input words, and T_x is the length of the input sequence
- **Output Sequence Notation**

    - $y^{\langle t \rangle}$: The output at position t in the sequence
    - $y^{\langle 1 \rangle}, y^{\langle 2 \rangle}, \ldots, y^{\langle T_y \rangle}$: Indexed output values, and T_y is the length of the output sequence
- **Input/Output Sequence Length**
    - T_x: Number of words in the input sequence (length of input)
    - T_y: Number of words/tokens in the output sequence (length of output)
    - Convention: T_x and T_y may be different, but in some problems (like NER), they are usually equal
- **Training Example Indexing**
    - $x^{(i)}$: The $i-th$ training example input sequence
    - $x^{(i)\langle t \rangle}$: The t-th element of the i-th training example's input sequence
    - $T_x^{(i)}$: The length of the input sequence in the i-th training example
    - $y^{(i)\langle t \rangle}$: The t-th output in the output sequence for the i-th example
    - $T_y^{(i)}$: The output sequence length for the i-th training example
- **Vocabulary and One-Hot Encoding**
    - Use V for vocabulary size, e.g., V=10,000
    - Each word represented as a one-hot vector of length V
        - If "Harry" is word index 4075, then one-hot vector is all zeros except for a one at position 4075
    - For unknown words: Special token **UNK** (stands for "unknown word") for words not in vocabulary

Recurrent Neural Network Model

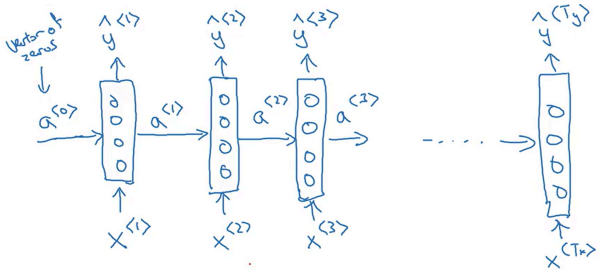
Why not a standard network?



Problems:

- Requires large number of input features for each word (one hot encode) computational cost is high
- Inputs, outputs can be different lengths in different examples
- Doesn't share features learned across different positions of text

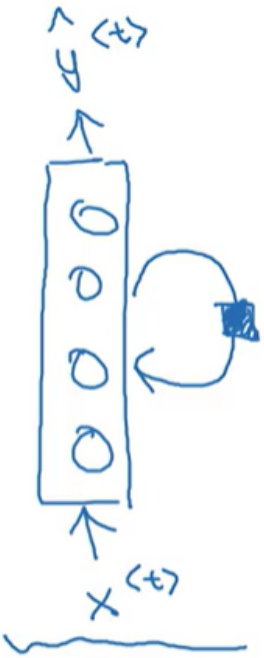
RNN model:



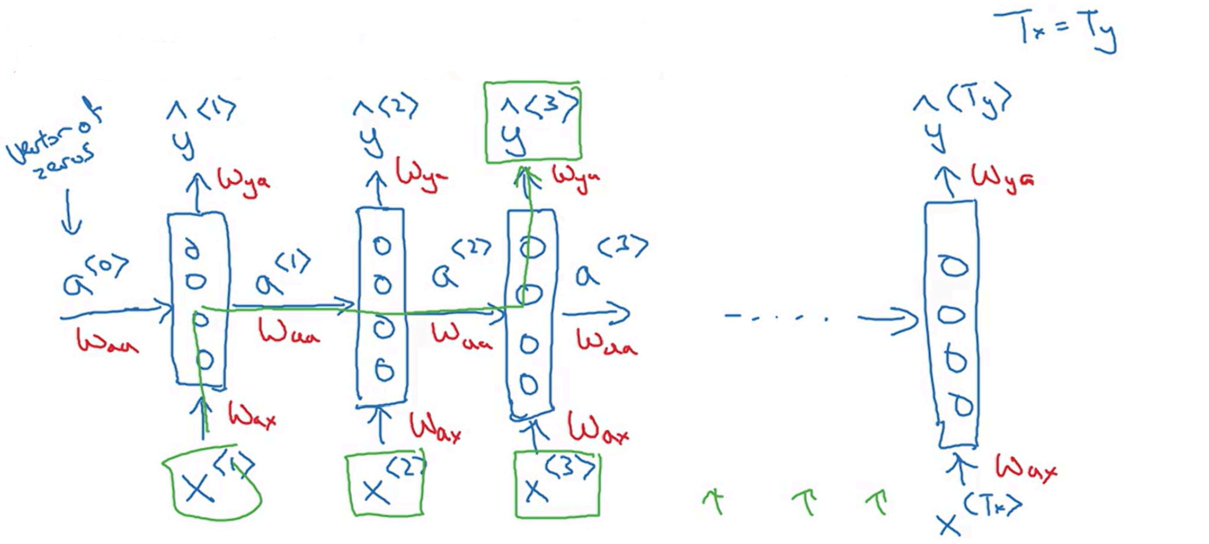
usually takes input from previous layer and give it to the next layer here activations are shared among each layer

where $T_x = T_y$ here

$a^{<0>}$ usually zero vector, some cases the author may give random inputs for start



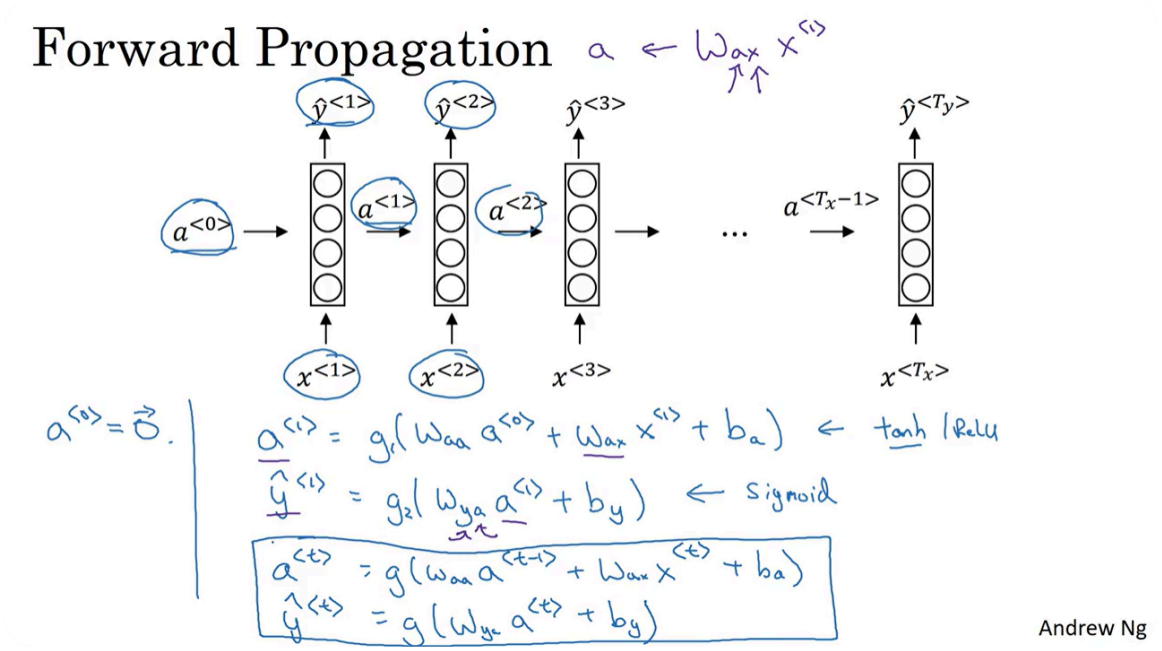
Roled RNN

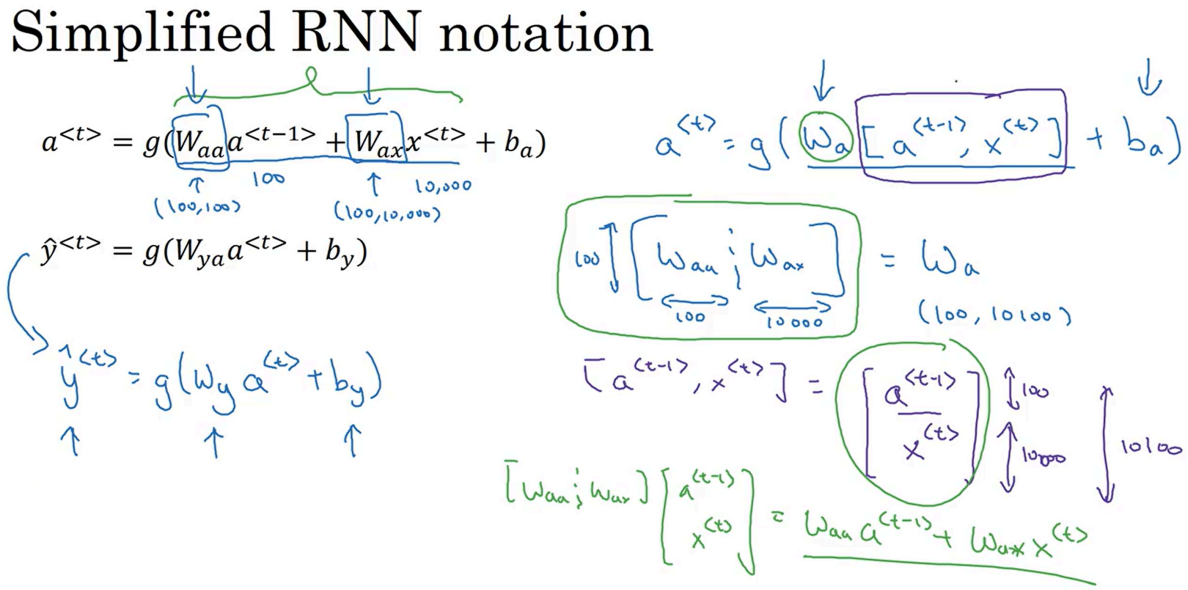


He said, “Teddy Roosevelt was a great President.”

He said, “Teddy bears are on sale!”

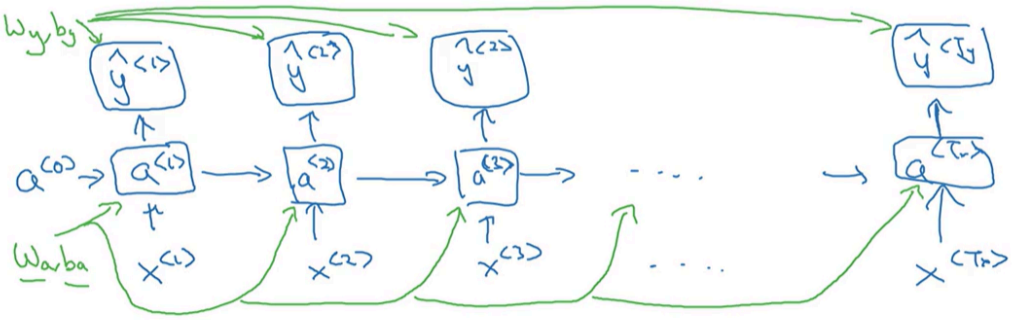
simple unidirectional RNN it may not generalize well because for the above examples it may not get correct output (we require bidirectional RNN)





Backpropagation Through Time

Forward Propagation:

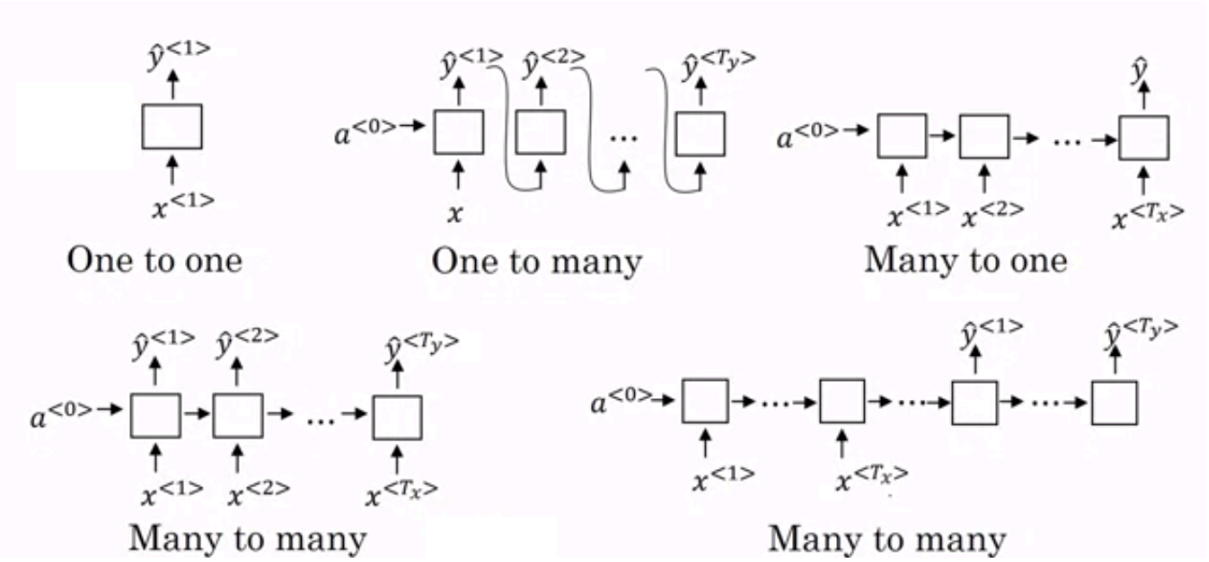


Loss Function:

$$\mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -y^{(t)} \log \hat{y}^{(t)} - (1-y^{<t>}) \log (1-\hat{y}^{(t)})$$

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

Different Types of RNNs



1. One-to-One

- **Diagram meaning:** Single input → Single output
- **Example scenarios:**

- **Image Classification:** Input = Image, Output = Label (e.g., cat, dog).
- **Spam Detection:** Input = One email text, Output = Spam/Not Spam.
- **Sentiment Analysis (binary):** Input = One review, Output = Positive/Negative.

2. One-to-Many

- **Diagram meaning:** Single input → Sequence output
- **Example scenarios:**
  - **Image Captioning:** Input = Image, Output = Caption (a sentence describing the image).
  - **Music Generation from a Theme:** Input = Seed note, Output = Music sequence.
  - **Text-to-Speech (simplified):** Input = Word, Output = Sequence of speech signals.

3. Many-to-One

- **Diagram meaning:** Sequence input → Single output
- **Example scenarios:**
  - **Sentiment Analysis (on long text):** Input = Review sentence(s), Output = Sentiment label.
  - **Document Classification:** Input = Sequence of words in document, Output = Category (sports, politics, etc.).
  - **Fraud Detection:** Input = Transaction sequence, Output = Fraud/Not Fraud.

4. Many-to-Many (synchronous)

- **Diagram meaning:** Sequence input → Sequence output (same length)
- **Example scenarios:**
  - **Named Entity Recognition (NER):** Input = Sentence words, Output = Labels for each word (e.g., Person, Location, etc.).
  - **Part-of-Speech Tagging:** Input = Sentence, Output = POS tag for each word.

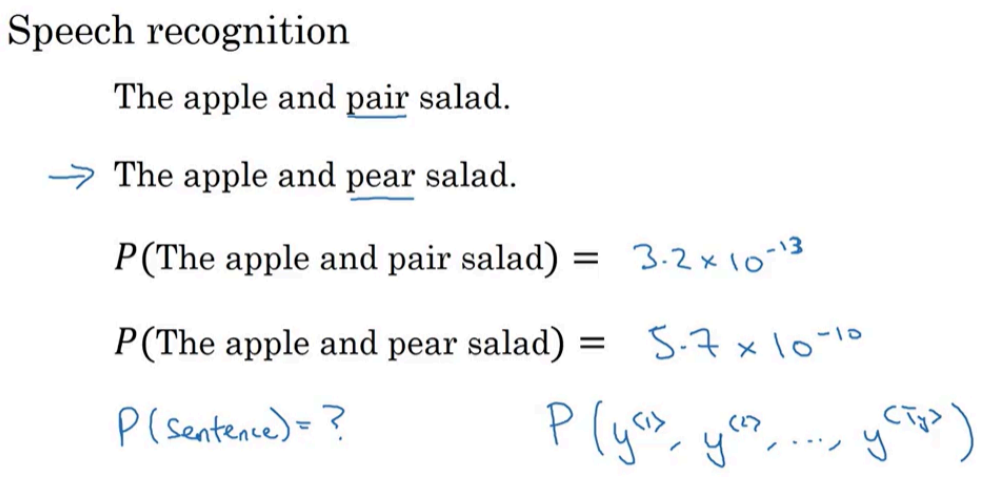
- **Video Frame Labeling:** Input = Frames, Output = Label per frame.

5. Many-to-Many (asynchronous / seq2seq)

- **Diagram meaning:** Sequence input → Sequence output (different lengths)
- **Example scenarios:**
  - **Machine Translation:** Input = Sentence in English, Output = Sentence in French.
  - **Chatbots / Q&A:** Input = User's query, Output = Response.
  - **Speech Recognition:** Input = Audio sequence, Output = Text sequence.

Language Model and Sequence Generation

The job of Language model is to predict the probability of the sentence

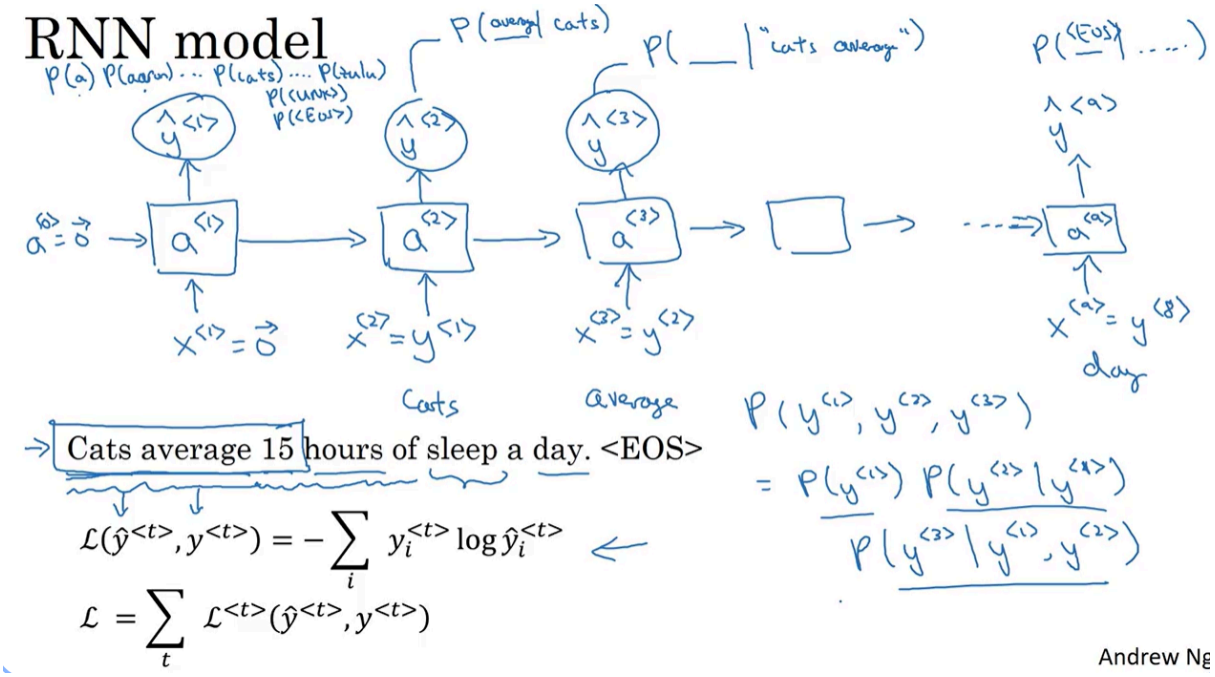


Definition & Purpose

- **Language Model**: Estimates probability P(sentence) for any given sentence
- **Applications**: Speech recognition, machine translation systems
- **Example**: "apple and pear salad" vs "apple and pair salad" → model assigns higher probability to grammatically correct version

Training Data & Preprocessing

- **Corpus**: Large body of text (tens of thousands of sentences)
- **Tokenization Steps**:
  - Map words to vocabulary indices/one-hot vectors
  - Add **EOS** (End of Sentence) token
  - Handle **UNK** (Unknown) tokens for out-of-vocabulary words
  - Typical vocabulary: ~10,000 most common words



Input-Output Mapping:

- $x^{<t>} = y^{<t-1>}$ (previous word becomes current input)
- $y^{<t>}$ = target word at time t

Training Process

- **Softmax Output**: Predicts probability distribution over entire vocabulary
- **Loss Function**: $L(\hat{y}^{<t>}, y^{<t}) = -\sum_{i=1} y_i^t \log \hat{y}_i^{<t>})$
- **Total Loss**: $L = \sum_{t=1}^{T_y} L_t$

Probability Calculation

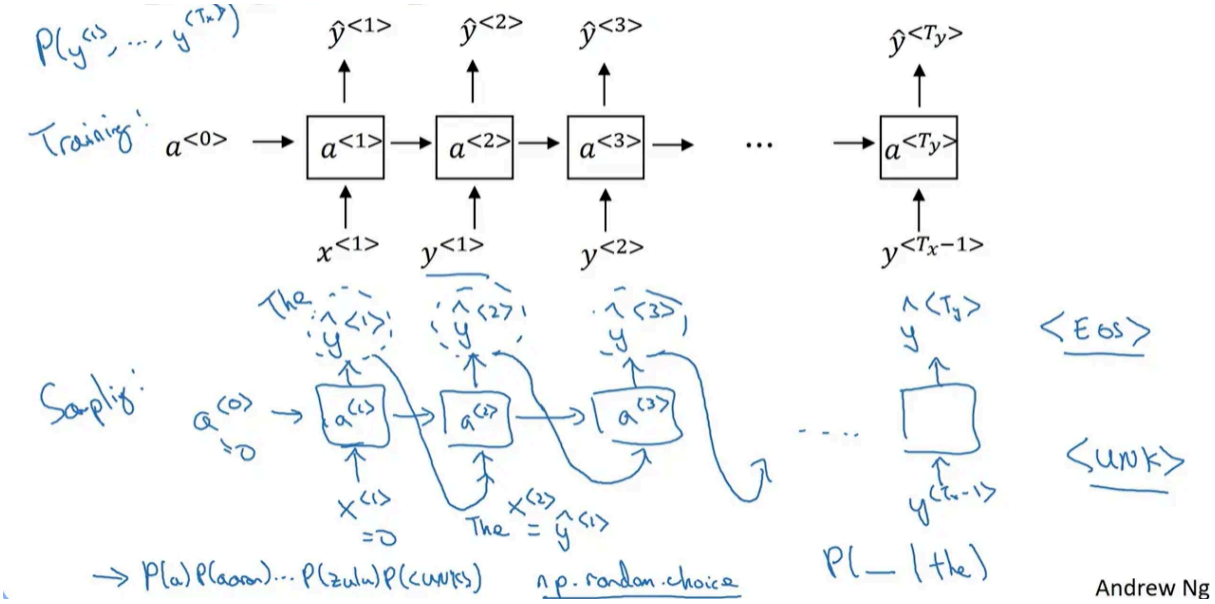
For sentence y^1, y^2, y^3:

$$P(y^1, y^2, y^3) = P(y^1) \times P(y^2|y^1) \times P(y^3|y^1, y^2)$$

Key Insights

- RNN learns conditional probabilities: P(word | previous words)
- Each timestep uses ground truth previous words (teacher forcing)
- Model can predict next word given any sequence prefix
- Foundation for text generation and sequence sampling

Sampling Novel Sequences



1. Purpose of Sampling

- After training a sequence model (like an RNN), you can generate new, original text by sampling from the model.
- This helps you see what the model has learned and how well it can mimic the style of the training data.

2. **How Sampling Works: Step-by-Step**

Step 1: Initialize

- Start with special initial inputs: usually x1=0x_1 = 0x1=0 (start token) and a0=0a_0 = 0a0=0 (initial hidden state).

Step 2: Predict Probabilities

- The model outputs a probability distribution (via softmax) over all possible words in the vocabulary for the next word.

Step 3: Sample the Next Word

- Use a function like `np.random.choice` to randomly select a word, but with probabilities given by the model's output.
- This means more likely words are chosen more often, but less likely words can still appear, adding variety.

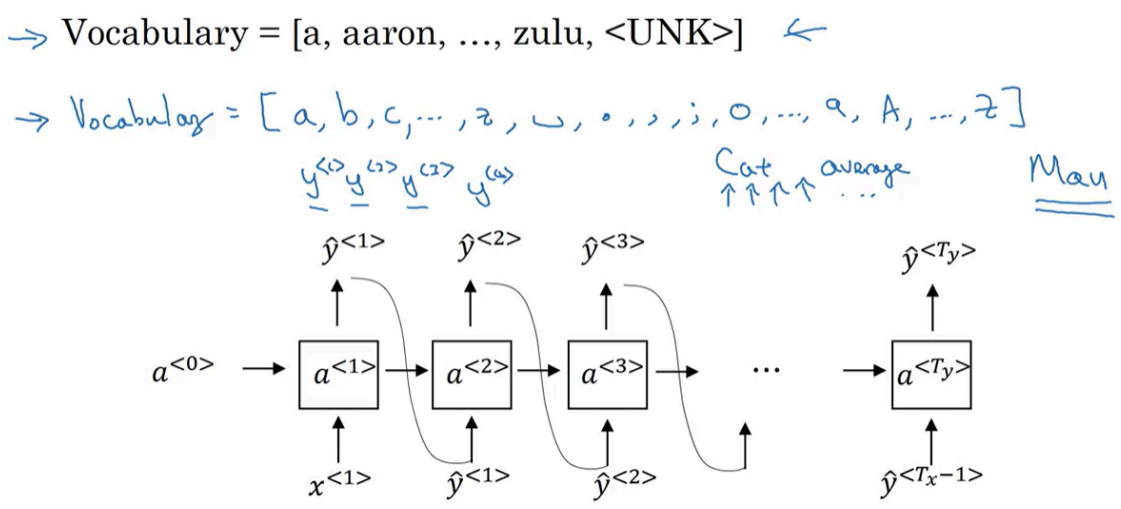
Step 4: Feed the Sampled Word Forward

- The word you just picked becomes the input for the next timestep.
- Repeat: the model predicts the next word's probabilities, and you sample again.

Step 5: Continue Until Stopping Condition

- Keep sampling and feeding forward until:
  - You generate an End-of-Sentence (EOS) token (if your vocabulary has one), or
  - You reach a set number of words (e.g., 20 or 100).
- If you sample an unknown word token (<UNK>), you can either keep it or resample to avoid it.

3. **Word-Level vs. Character-Level Models**



- **Word-level models:** Each token is a word. Most common in practice.

- **Character-level models:** Each token is a character (a-z, punctuation, etc.).
  - **Pros:** No unknown word problem; can generate any sequence.
  - **Cons:** Sequences are much longer; harder to capture long-range dependencies; more computationally expensive.
- Word-level models are still more widely used, but character-level models are useful for special cases.

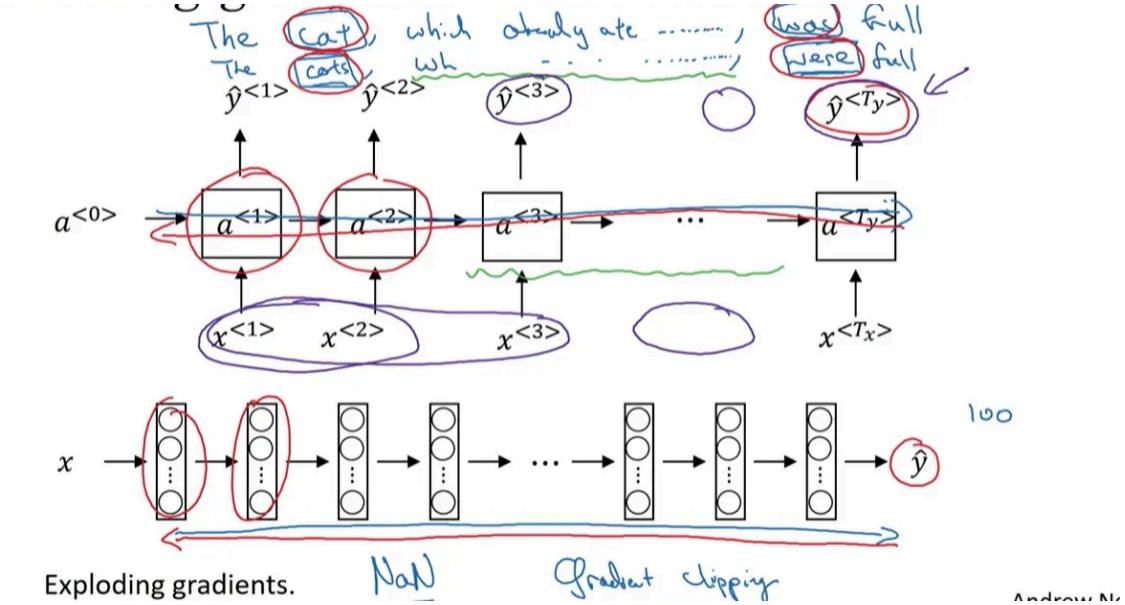
4. **Examples of Generated Text**

- If trained on news articles: "concussion epidemic to be examined"
- If trained on Shakespeare: "The mortal moon hath her eclipse in love"

5. **Key Takeaways**

- Sampling with the model's probabilities creates new, realistic text that reflects the training data's style.
- Always picking the most likely word (greedy decoding) makes text repetitive and dull.
- Purely random sampling (ignoring probabilities) produces nonsense.
- The sampling process is essential for making language models creative and useful for text generation tasks.

Vanishing Gradients with RNNs

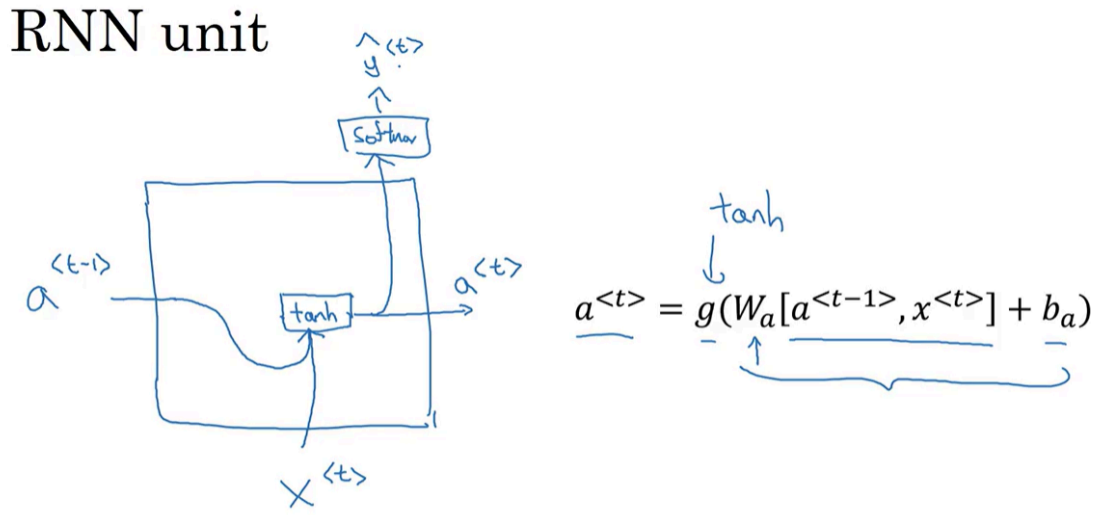


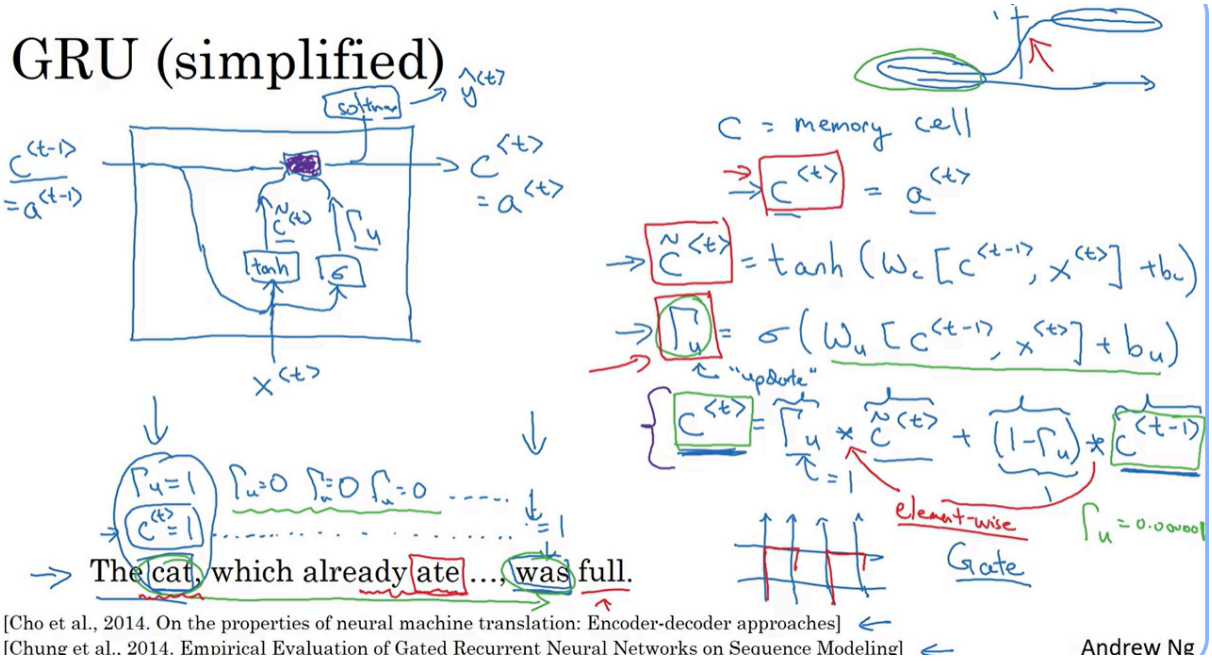
- **RNN Recap**: RNNs process sequences and can be used for tasks like language modeling and named entity recognition.
- **Backpropagation in RNNs**: Training involves propagating errors backward through time (Backpropagation Through Time).
- **Problem: Vanishing Gradients**:
  - In deep networks (and RNNs over long sequences), gradients can shrink exponentially as they're backpropagated.
  - Early layers/timesteps receive extremely small gradients, making it difficult to learn long-term dependencies.
  - Example: In a language model, remembering if the subject was singular/plural gets harder as ***the gap between verb and subject increases.***
- **Local Influence**:
  - RNN outputs are heavily influenced by values close by in the sequence.
  - It's difficult for an output at time t to be heavily influenced by inputs from much earlier in the sequence.
  - Output errors don't effectively update computations from earlier timesteps due to vanishing gradients.
- **Consequence**: Basic RNNs struggle with long-range dependencies.
- **Exploding Gradients**:

- Gradients can also grow exponentially, leading to extremely large parameter updates.
- Symptoms: Model parameters diverge, may see NaNs (numerical overflows).
- **Solution for Exploding Gradients**: Use gradient clipping (limit the size of gradients to keep values within a reasonable range).

- **Summary**:
  - RNNs over long sequences are like very deep neural networks (~1000 layers = 1000 timesteps).
  - Both vanishing and exploding gradients occur, but vanishing gradients are harder to solve (especially in RNNs).
  - Exploding gradients: easy fix via **clipping**. Vanishing gradients: need structural changes to the RNN.

Gated Recurrent Unit (GRU)





1. GRU Basics

- **GRU = Gated Recurrent Unit** → a simplified version of LSTM.
- Designed to solve **vanishing gradient problem** and capture **long-term dependencies**.
- Maintains a **memory cell** (like LSTM, but simpler).
- At each time step (t):
  - Input: ($x^{<t>}$)
  - Previous hidden state (memory): ($c^{<t-1>}$)
  - Output: ($y^{<t>}$), New memory: ($c^{<t>}$)

2. Main Equations

- Candidate memory:

  $\tilde{c}^{<t>} = \tanh(W_c[c^{<t-1>}, x^{<t>}] + b_c)$

- Update gate:

  $\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$

- New memory:

  $c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$

Key:

- (Γ_u) (update gate) decides **how much new info to keep vs. how much old memory to retain**.
- Operations are **element-wise** (denoted by ⊙ in some texts).

3. Interpretation

- If ($\Gamma_u \approx 1$): mostly keep the new candidate value ($\tilde{c}^{<t>}$).
- If ($\Gamma_u \approx 0$): mostly keep the old memory ($c^{<t-1>}$).
- The model learns when to **remember** and when to **update**.

4. Example (Sentence Memory)

Sentence:

“The cat, which already ate, was full.”

- At the word **“cat”** → GRU stores info (($\Gamma_u = 1$)).
- At **intermediate words (“which already ate”)** → update gate ~ 0, memory doesn’t change.
- At **“was full”** → memory updates again.

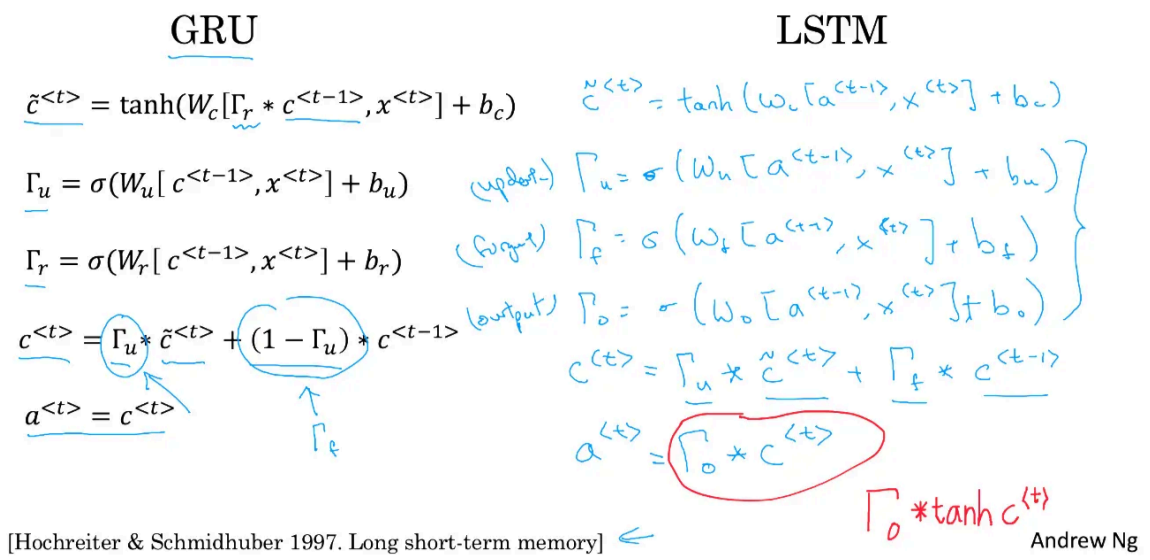
This shows GRU’s ability to **skip irrelevant words** while keeping **long-term context**.

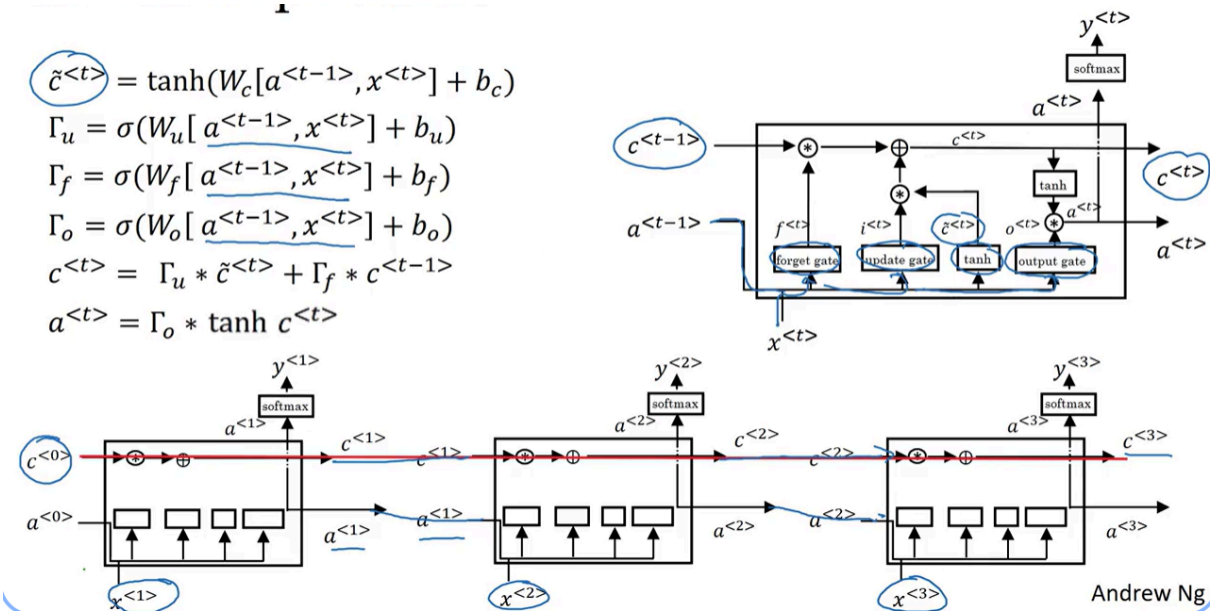
6. Visualization Insights

- Green = memory cell updates.
- Red = equations for update.
- Arrows = how past info flows forward.
- Shows **long-term dependency handling** using gates.

In short: **GRU is a lighter LSTM variant with fewer gates, yet effective in remembering long-term dependencies while being computationally efficient.**

Long Short Term Memory (LSTM)





Equations:

- Candidate memory:

  $$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$$

- Gates:
  - Update gate (a.k.a input gate):

    $$\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u)$$

  - Forget gate:

    $$\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$$

  - Output gate:

$$\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$$

- Memory cell update:
  $c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$
- Hidden state (activation):
  $a^{<t>} = \Gamma_o * \tanh(c^{<t>})$

Key Points:

- Has **3 gates**:
  1. **Update gate** ((Γ_u))
  2. **Forget gate** ((Γ_f))
  3. **Output gate** ((Γ_o))
- Maintains **separate memory cell** ($(c^{<t>})$) and hidden state ($(a^{<t>})$).
- More expressive but computationally heavier.

GRU vs LSTM

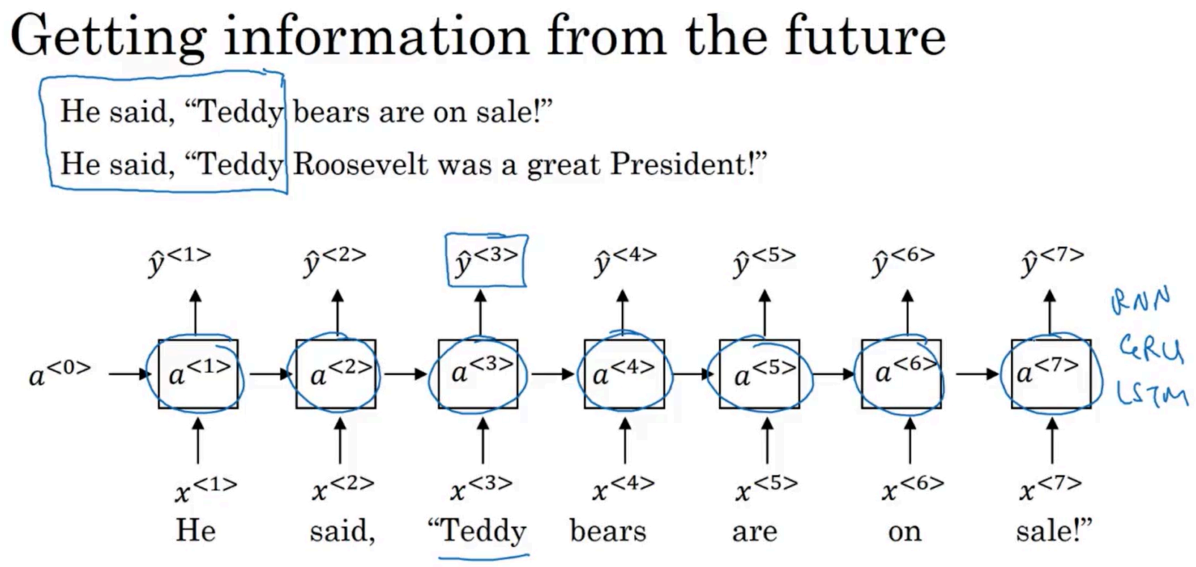
Feature	GRU	LSTM
Gates	2 (Update, Reset)	3 (Update/Input, Forget, Output)
Memory Cell	No separate cell (uses $(c^{<t>} = a^{<t>})$)	Has explicit memory cell ($c^{<t>}$)
Complexity	Simpler, faster training	More complex, slower
Performance	Often similar to LSTM	Sometimes slightly better for very long dependencies
Usage	When efficiency is key	When maximum accuracy is needed

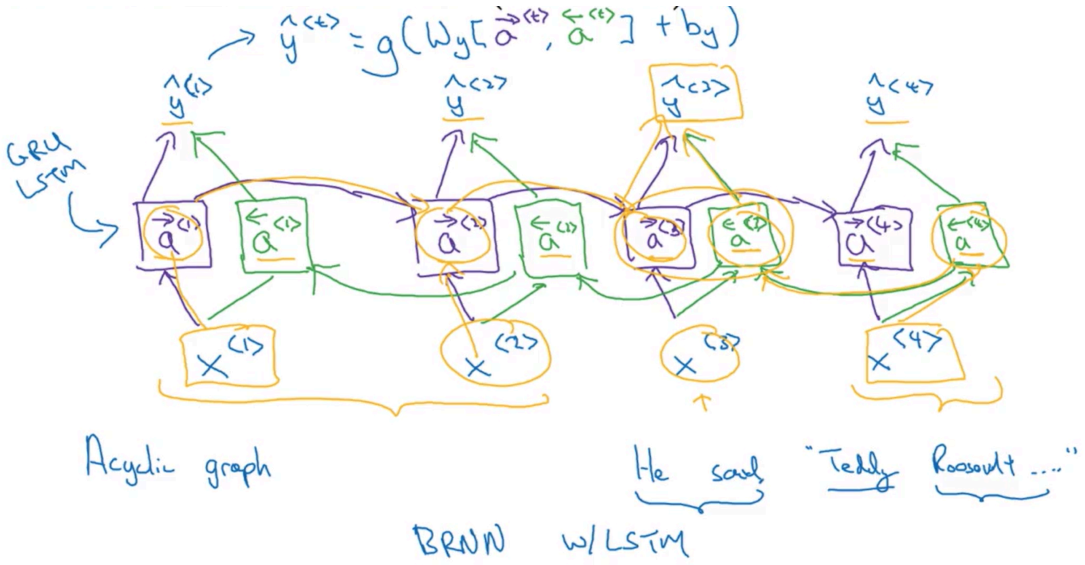
In short:

- **LSTM = more powerful, more parameters.**
- **GRU = simpler, efficient, often just as good.**

Bidirectional RNN

Forward Direction Example:

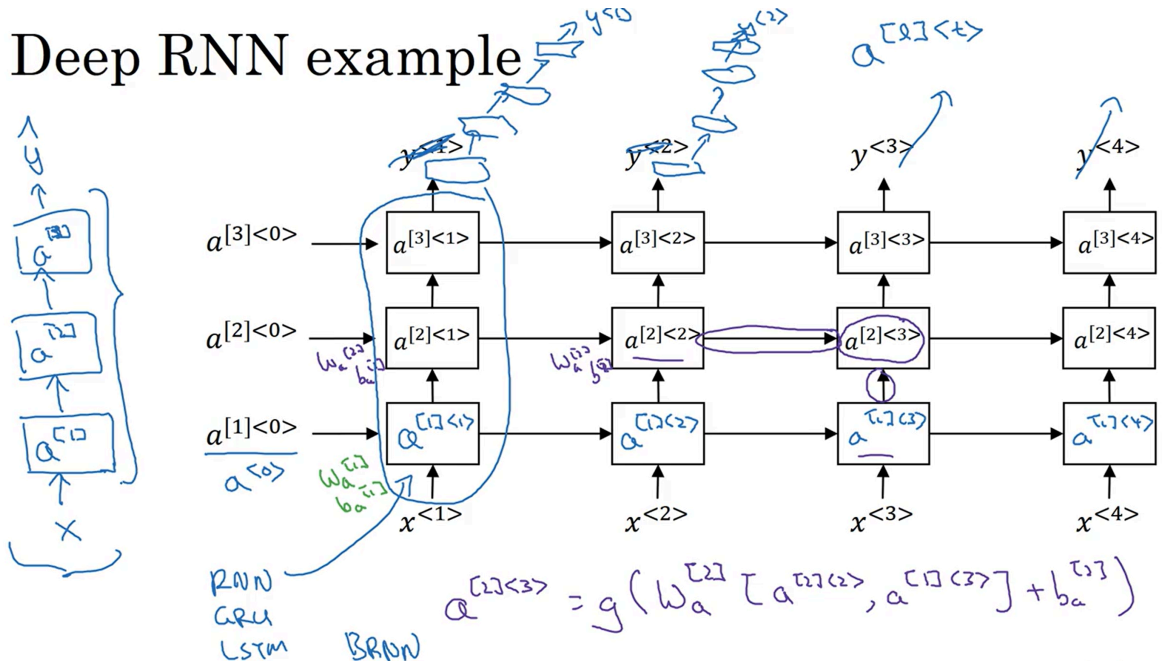




- **Problem Motivation:**
  - Standard (unidirectional) RNNs use only past context provided by earlier words to make predictions for each time step. This can be limiting: e.g., to check if "Teddy" is a person's name, future context (“Roosevelt”) is important.
- **Bidirectional RNN (BRNN) Architecture:**
  - BRNNs have two hidden states per time step:

        - **Forward hidden state:** moves left-to-right over the sequence (past → future)
        - **Backward hidden state:** moves right-to-left (future → past)
    - For each sequence input X_1, X_2, X_3, X_4, the forward pass computes activations from start to end; the backward pass computes from end to start.
- **Combining Hidden States:**
    - Output at time t uses both the forward and backward hidden states:
        - $\hat{y}_t = f(W_y[h_t^f, h_t^b])$
        - This allows the prediction at any time t to use information from both the past and the future.
- **Blocks Used:**
    - The sequence blocks can be standard RNNs, GRUs, or LSTMs.
    - Common practice in NLP: **Bidirectional LSTM** (BiLSTM).
- **Advantages:**
    - Allows the network to use full context of the sentence, both previous and future words.
    - Especially beneficial for tasks like Named Entity Recognition and text labeling, where full context improves accuracy.
- **Limitation:**
    - BRNNs require the entire sequence up front; cannot operate on sequences where future data is unavailable in real time (e.g., live speech recognition).
    - For real-time tasks, more complex architectures are needed.
- **Summary:**
    - BRNN is recommended for many NLP tasks where complete sentences are available.

Deep RNNs



- **Standard RNNs:**
  - Prior RNNs (simple, GRU, LSTM, bidirectional) can already model complex sequences well.
  - For even more complex functions, deeper architectures (multiple stacked layers) are helpful.
- **Deep RNN Architecture:**
  - A deep RNN stacks several recurrent layers vertically.
    - Each layer processes activations from the previous layer and from the corresponding time step.
    - Notation: $a^{[l]<t>}$, where l is the layer and t is the timestep.
  - For each layer l:
    - Activation at time t: function of previous layer's output and previous timestep's value for same layer.
    - Example computation:
      $a^{[2]<3>} = g(W_a^{[2]}[a^{[2]<2>}, a^{[1]<3>}] + b^{[2]})$
- **Stacked Recurrent Layers:**
  - Typical deep networks (feed-forward): can have 100+ layers.
  - Deep RNNs: 2-3 recurrent layers are already considered "deep" due to temporal complexity. Rarely stacked as deep as feed-forward networks.

- **Alternate Architectures:**
  - Stacked RNN layers, followed by standard DNN layers for prediction at each time step.
  - Common to use GRU or LSTM as blocks within these stacks.
  - You can also build deep versions of bidirectional RNNs.
- **Training Considerations:**
  - Deep RNNs are computationally expensive due to both vertical and temporal depth.
  - Large temporal extent means even “shallow” stacks become complex.
- **Summary:**
  - Deep RNN = multi-layer recurrent neural network.
  - Extends basic RNN, GRU, LSTM, and bidirectional RNN models for high learning capacity.
  - Useful in advanced sequence modeling tasks but used cautiously due to computational costs.