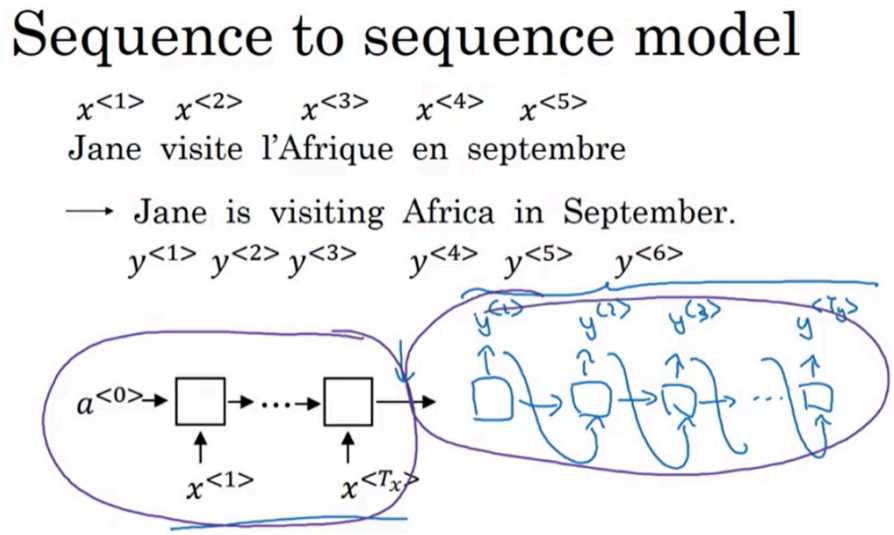
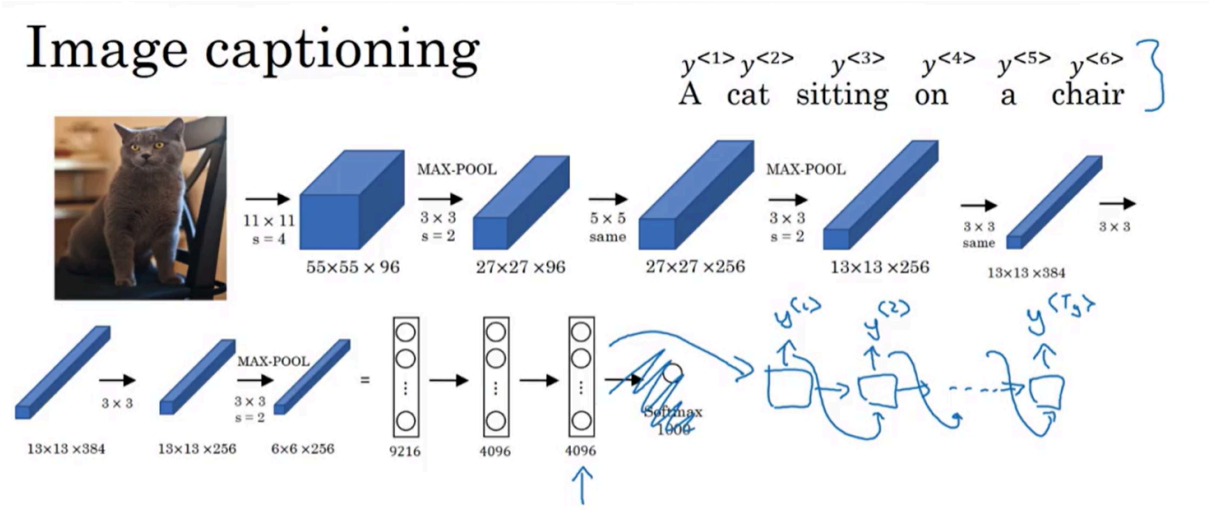
# WEEK 3: Sequence Models & Attention Mechanism

## Basic Models

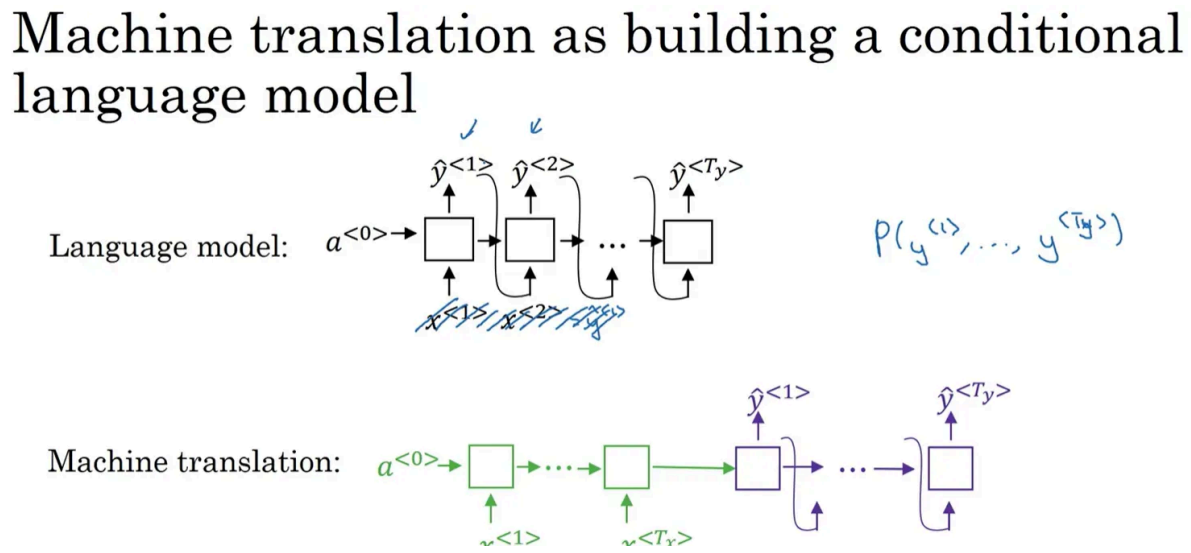



Machine Translation - French to English
encoder-decoder



# Picking the Most Likely Sentence

## 1. Background: Language Models vs. Machine Translation



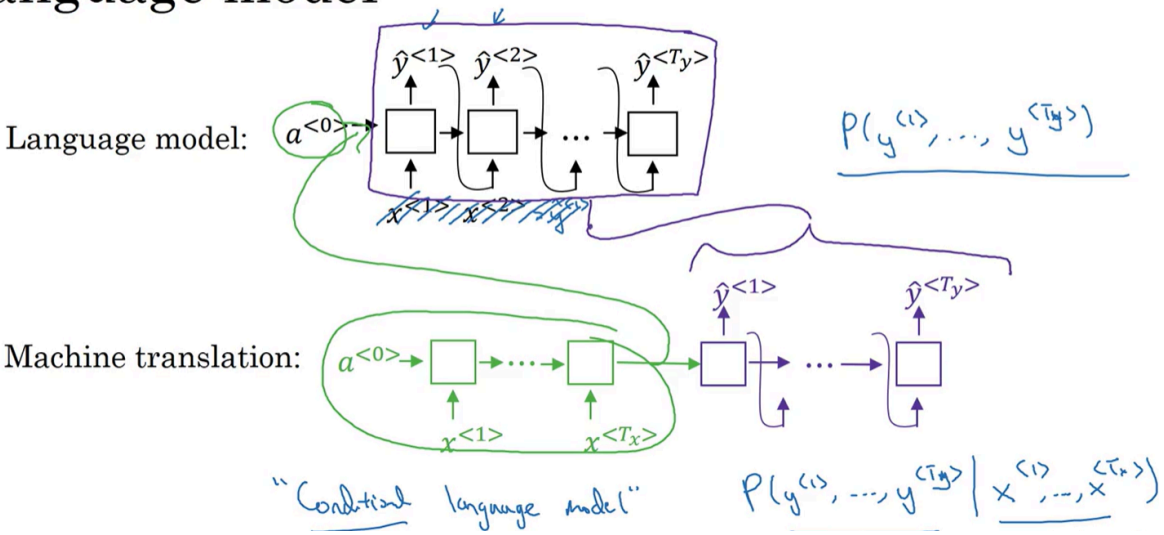

- **Language Model** estimates probability of a sentence:

  $P(y)$, where $y = (y^{<1>}, y^{<2>}, \ldots, y^{<T>})$ is a sequence of words.

- **Machine Translation Model** as a *conditional* language model: models probability of an output sentence given an input (e.g., translating French → English).
  - Notation: $P(y \mid x)$, where x is input sentence and y is output translation.
  - Example: Given $x =$ “Jane visite l’Afrique en septembre”, Model estimates probability for different English translations y.

## 2. Conditional Language Modeling Explained

## Machine translation as building a conditional language model




- **Encoder-Decoder Framework**:
  - Encoder (green): transforms input sentence x into a vector representation.
  - Decoder (purple): generates output sequence, leveraging encoded representation rather than starting from zero vector like basic language models.
  - Enables modeling $P(y \mid x)$: Probability of y conditioned on x.

## Finding the most likely translation

Jane visite l'Afrique en septembre.

$P(y^{<1>}, \ldots, y^{<T_y>} \mid x)$

- ⟶ Jane is visiting Africa in September.
- ⟶ Jane is going to be visiting Africa in September.
- ⟶ In September, Jane will visit Africa.
- ⟶ Her African friend welcomed Jane in September.

$$\arg\max_{y^{<1>},\ldots,y^{<T_y>}} P(y^{<1>}, \ldots, y^{<T_y>} \mid x)$$

# 3. Why Not Sample Randomly?

- **Random Sampling** from $P(y \mid x)$ yields variable translations – some correct, some awkward, some irrelevant.

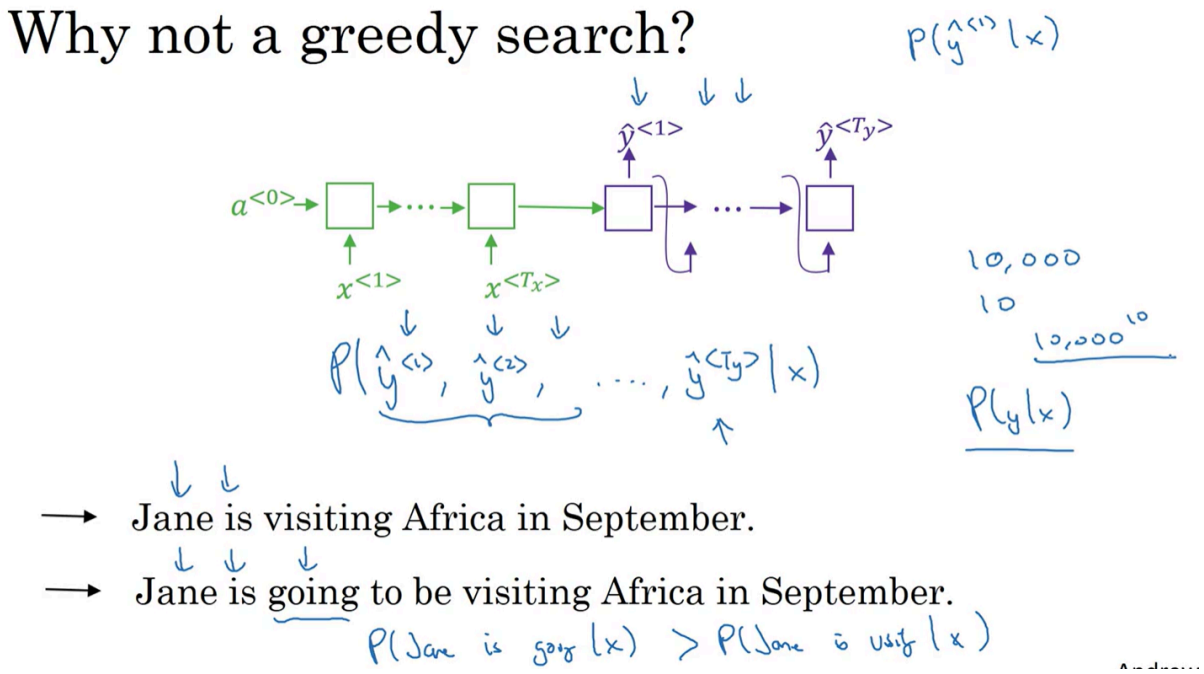
- Example outputs for x above:
  - "Jane is visiting Africa in September." (good)
  - "Jane is going to be visiting Africa in September." (verbosely correct)
  - "Her African friend welcomed Jane in September." (incorrect)

- **Goal:** *Find the translation* $y^*$ *maximizing probability*: $y^* = \arg\max_y P(y|x)$

# 4. Search Algorithms for Picking $y^*$: How to Actually Find It

- **Exhaustive Search** is infeasible:
  - Total possible sentences grows *exponentially* with sentence length ( $V^T$, with V = vocab size, T = length).
  - Example: V=10,000, T=10 → $10^{40}$ sequences.

## a. Greedy Search




- **Procedure:** At each time step, pick word with highest $P(y_t|y_{1:t-1}, x)$.
- **Issue:** May get stuck in sub-optimal translations by only locally maximizing at each step (misses global optimum for whole sentence).

- **Example:**
  - Step 1: Picks "Jane is"
  - Next: Picks "going" (because "going" often follows "is" in English, not considering global translation quality)
  - Output: "Jane is going to be visiting Africa in September." (less optimal)

## b. Next Step: Beam Search

- **Standard solution** for maximizing P(y|x) approximately :
  - Maintains top k partial hypotheses instead of just 1 (greedy search), balances breadth and depth.

## 5. Summary & Key Takeaways

- **Machine Translation** is modeled as conditional language modeling: maximizing $P(y \mid x)$.
- **Greedy search often fails** to find best translation; need search algorithms to navigate huge space efficiently.
- **Beam search** is most popular approximate method.

**Practical Example:**

- Input: "Jane visite l'Afrique en septembre"
- Output candidates:
  1. "Jane is visiting Africa in September" (desired)
  2. "Jane is going to be visiting Africa in September" (less optimal)

## Suggestion for Deeper Reasoning

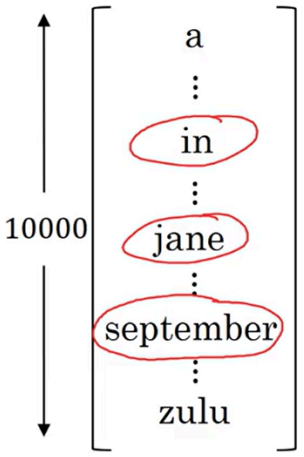
- **Understanding model output requires considering joint probability over entire sequences** — not just local maxima for word choices.
- **Efficient search (e.g., beam search) is crucial** for scaling translation systems to realistic vocab sizes and sentence lengths.

# Beam Search Algorithm

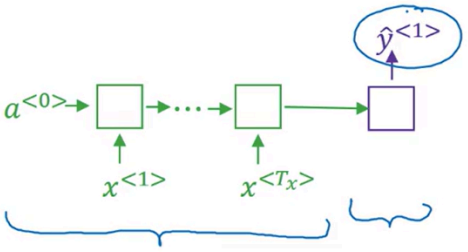
# Beam search algorithm

$B = 3$ (beam width)

Step 1



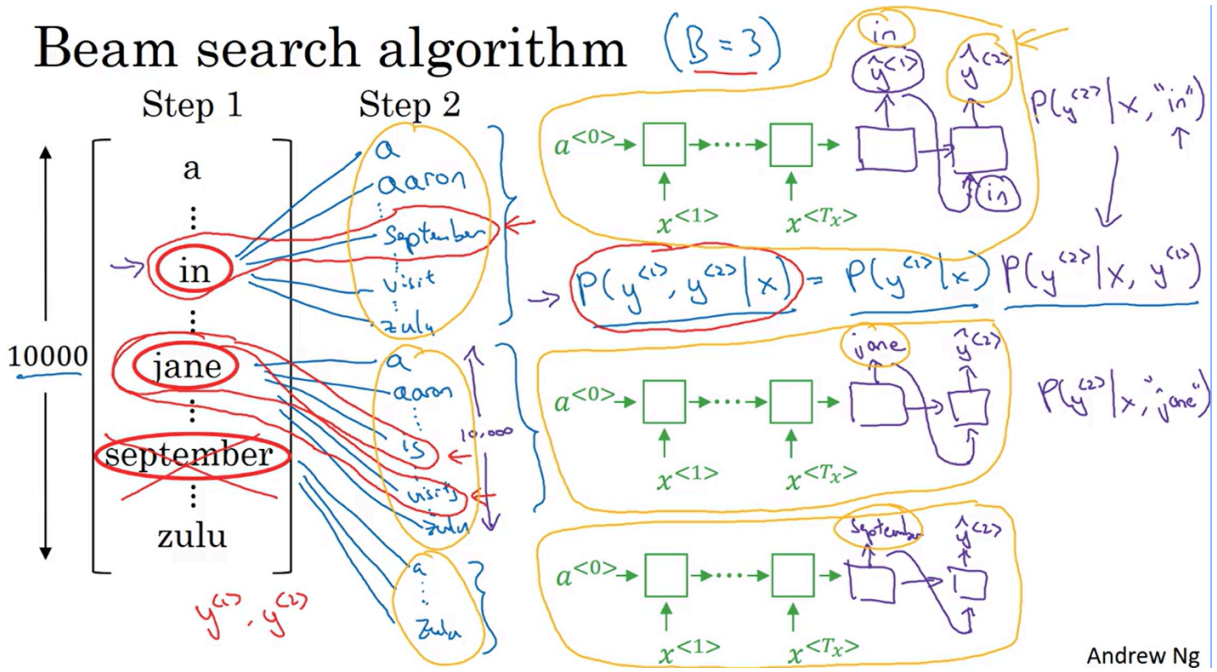

$\rightarrow P(y^{<1>} \mid x)$



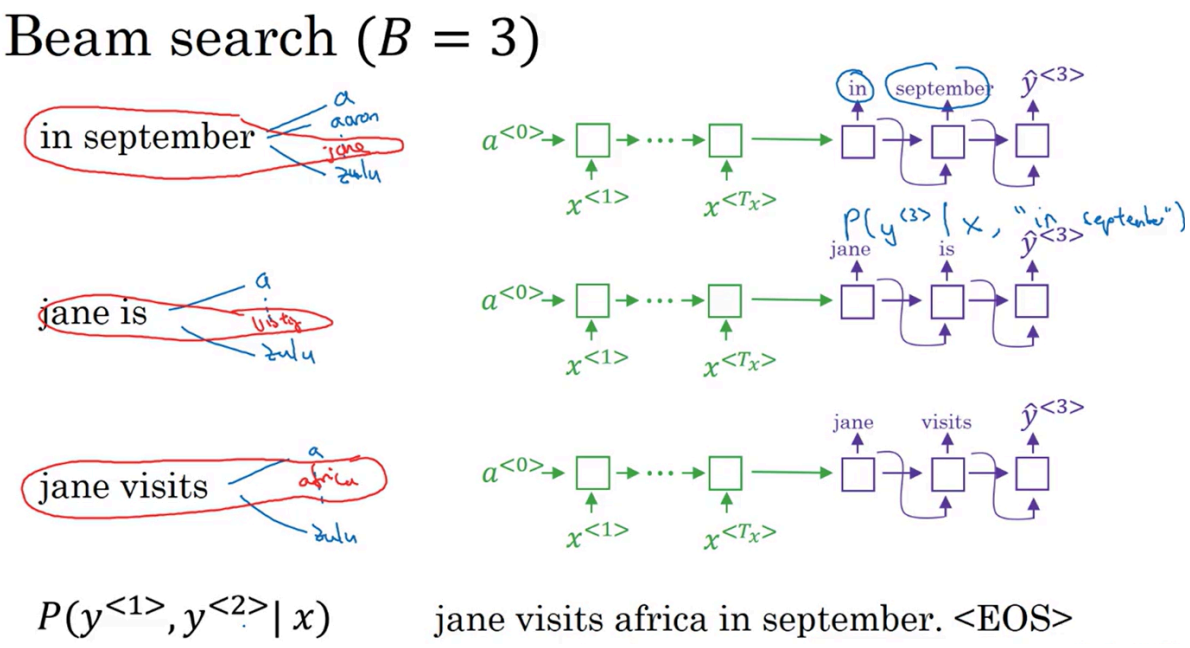

# Beam search algorithm

$(B = 3)$

Step 1 Step 2




Andrew Ng

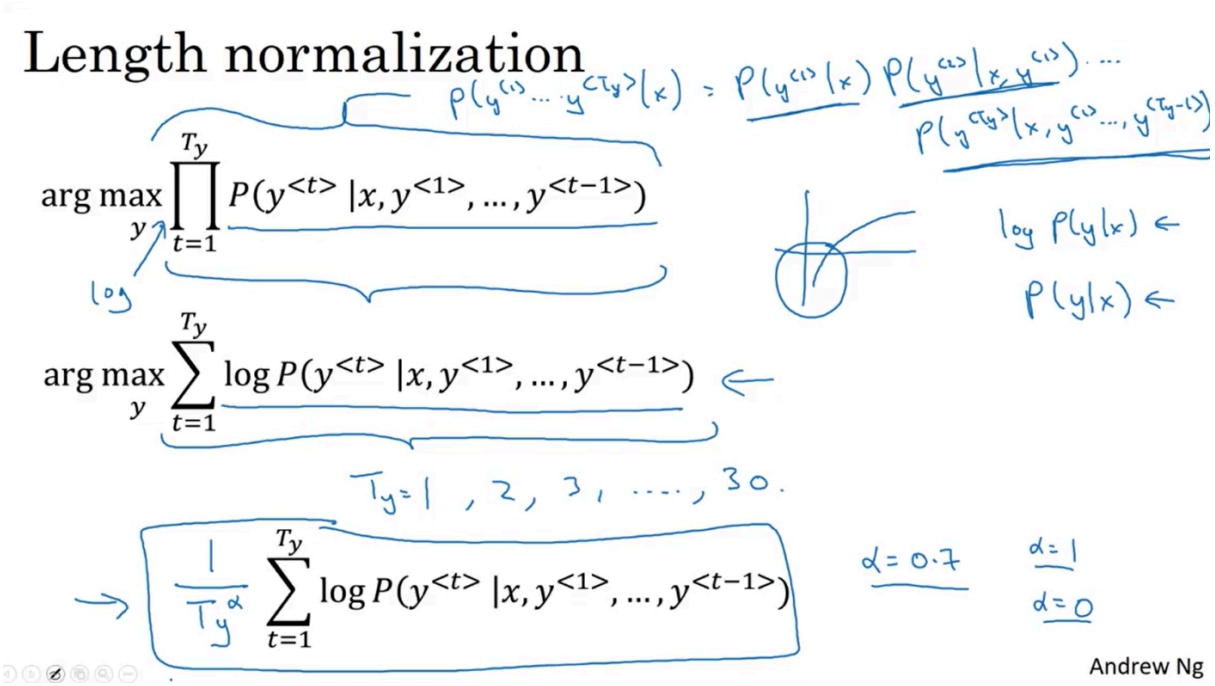


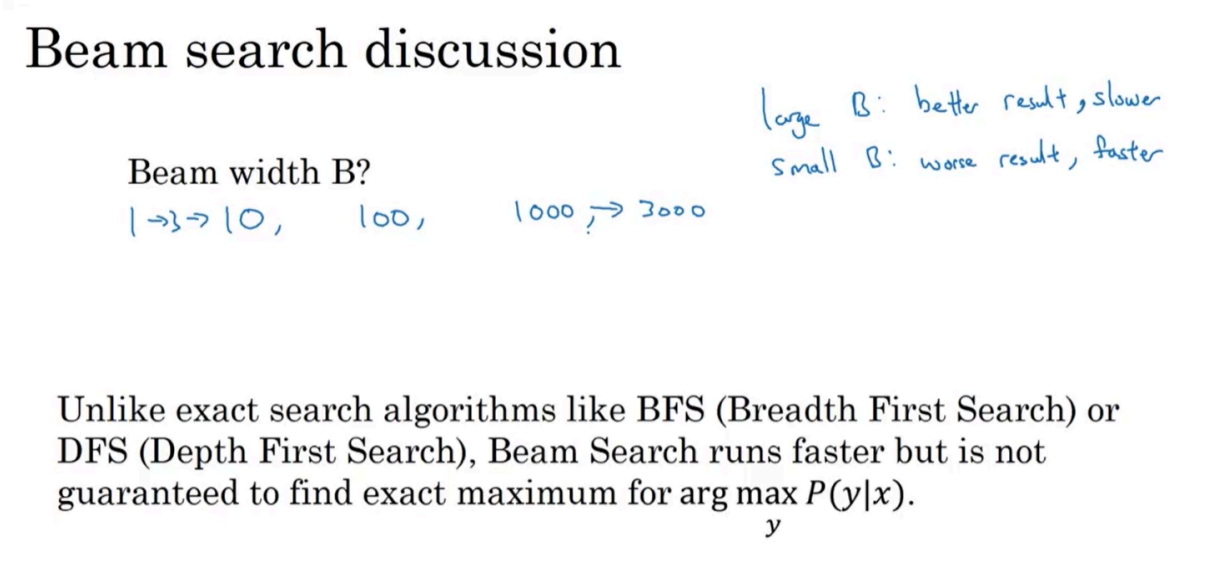

- **Problem Context:**
  - Task: Find the most likely output sequence (e.g., translation, transcript) rather than a random one.
  - Applications: Machine translation, speech recognition.
- **Definition:**
  - **Beam Search** is the most widely used algorithm to generate likely output sequences from models (e.g., neural networks).
- **How it Works:**
  1. **Initialization:**
     - At each step, instead of keeping only the single most likely choice (greedy search), it retains the top B (beam width) candidates.
     - Example: Beam width B=3
  2. **Step-wise Expansion:**
     - For each candidate at a given position, generate possible next steps (words).
     - Calculate probability for each possible sequence fragment up to current step.
     - For every expansion, multiply probabilities along the sequence (chain rule for conditional probabilities).

    - Retain the top B complete or partial sequences.
3. **Iteration:**
    - Repeat for each subsequent word until the end-of-sequence token is produced.
    - Only the top B sequences are kept during each iteration.

- **Efficiency:**
    - At every step, only maintain B options rather than all possible sequences (memory and compute efficient).
    - If vocabulary size is V, at each word position you consider B×V options, but keep only the best B.
- **Comparison:**
    - **Greedy Search** (B=1): Only follows the most probable single path.
    - **Beam Search** (B>1): Explores multiple likely paths, leading to better results in practice.
- **Example Flow:**
    - Translating “Jane, visite l’Afrique en Septembre” (French) to English.
    - Step 1: Find top 3 first words with highest probability.
    - Step 2: For each first word, expand to find most probable second words.
    - Step 3: Continue expanding, always keeping only top 3 partial translation sequences.
    - On completion: One of the outputs could be “Jane visits Africa in September.”
- **Practical Tips:**
    - Larger beam width usually improves accuracy, but increases computation.
    - Beam search works efficiently by instantiating a small number of network copies at each step.
- **Key Takeaways:**
    - **Beam search** balances quality and efficiency for sequence generation tasks.
    - Essential for tasks where the best sequence matters, not just individual best choices.

# Refinements to Beam Search




# 1. Probability Calculation for Sequences

- When translating or generating a sequence, we want to **maximize the conditional probability** of an output sequence given the input:$P(y_1, y_2, \ldots, y_T|x) = \prod_{t=1}^{T} P(y_t|x, y_1, \ldots, y_{t-1})$
  - Each term is usually a number between 0 and 1.

- Multiplying many small numbers yields a very tiny value, causing problems for floating-point representation (numerical underflow).

## 2. Log Probability for Numerical Stability

- Instead of maximizing the product of probabilities, we take the logarithm:

$$\log P(y_1, \ldots, y_T|x) = \sum_{t=1}^{T} \log P(y_t|x, y_1, \ldots, y_{t-1})$$

- Maximizing this sum is equivalent to maximizing the product, but it's much more numerically stable.
- **Implementation tip:** Most code for beam search tracks the sum of log probabilities rather than the product.

## 3. Length Bias & Normalization

- Without adjustment, longer sentences are penalized:
  - More words means more multiplication of numbers less than one, which reduces the overall probability.
- This bias can make beam search output **unnaturally short translations**.
- **Solution:** Normalize by length:

$$\text{Normalized Score} = \frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} \log P(y_t|x, y_1, \ldots, y_{t-1})$$

  - $T_y$: Number of words in the output sentence.
  - α: Hyperparameter (typical range: 0–1; e.g., 0.7).
  - α=1: Fully normalized by length.
  - α=0: No normalization.
- This is a heuristic, not derived from theory, but it works well in practice.

## 4. Beam Width Effects

- **Beam width** controls the number of candidates tracked per step:

- **Large beam width:**
  - Considers more options, possibly better results.
  - Slower and uses more memory.
- **Small beam width:**
  - Faster, less resource-heavy.
  - May miss the best results.

- **Typical values:**
  - Beam width of 3 is small (used for demos).
  - Around 10 is common in production.
  - Hundreds or thousands may be used in research, but with diminishing returns.

## 5. Relation to Other Search Algorithms

- **Beam search** is an *approximate* search:
  - Faster than exhaustive methods (BFS, DFS), but isn't guaranteed to find the exact best sequence (global optimum).
  - Trades off completeness (finding the best) for efficiency.

## Notions You Should Understand

- **Logarithm function is monotonic** (preserves order; maximizing log-probability gives same best result as raw probability).
- **Implementation:**
  - Track the sum of log-probabilities for each candidate.
  - Normalize scores by length to avoid short-output bias.
  - Experiment with α and beam width to get best results for your application.

# Error Analysis in Beam Search

# Example

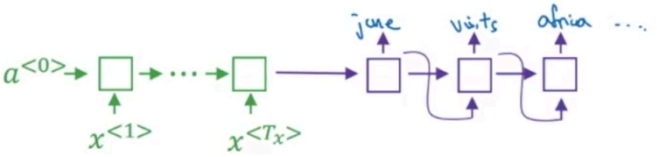
→ RNN
→ Beam Search
BT

Jane visite l'Afrique en septembre.

Human: Jane visits Africa in September. $(y^*)$

Algorithm: Jane visited Africa last September. $(\hat{y})$ ⇐

RNN computes $P(y^*|x) < P(\hat{y}|x)$




# Error analysis on beam search

$P(y^*|x)$
$P(\hat{y}|x)$

Human: Jane visits Africa in September. $(y^*)$

Algorithm: Jane visited Africa last September. $(\hat{y})$

Case 1: $P(y^*|x) > P(\hat{y}|x)$ ⇐

$\arg\max_y P(y|x)$

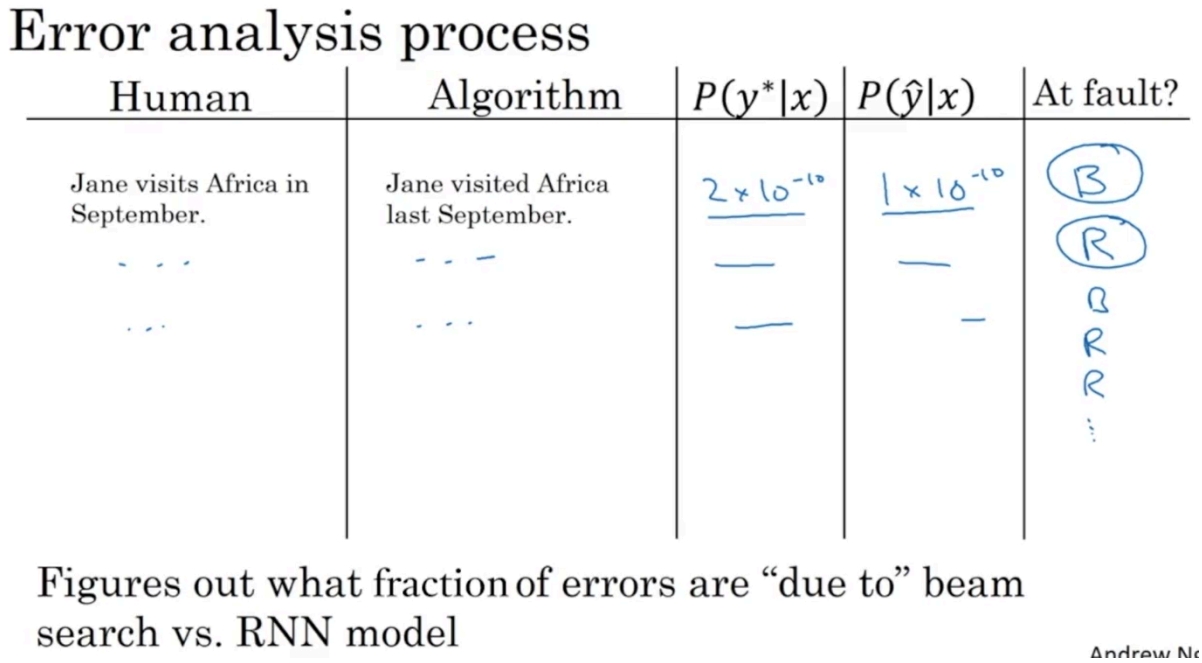
Beam search chose $\hat{y}$. But $y^*$ attains higher $P(y|x)$.

Conclusion: Beam search is at fault.

Case 2: $P(y^*|x) \leq P(\hat{y}|x)$ ⇐

$y^*$ is a better translation than $\hat{y}$. But RNN predicted $P(y^*|x) \leq P(\hat{y}|x)$.

Conclusion: RNN model is at fault.



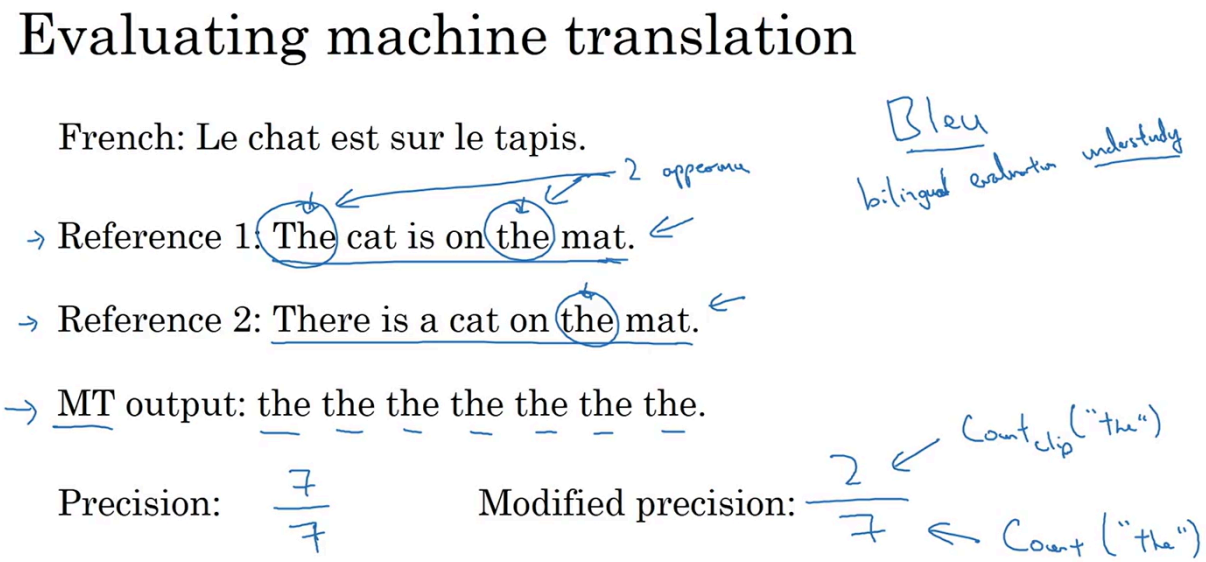

**Error Analysis in Beam Search**

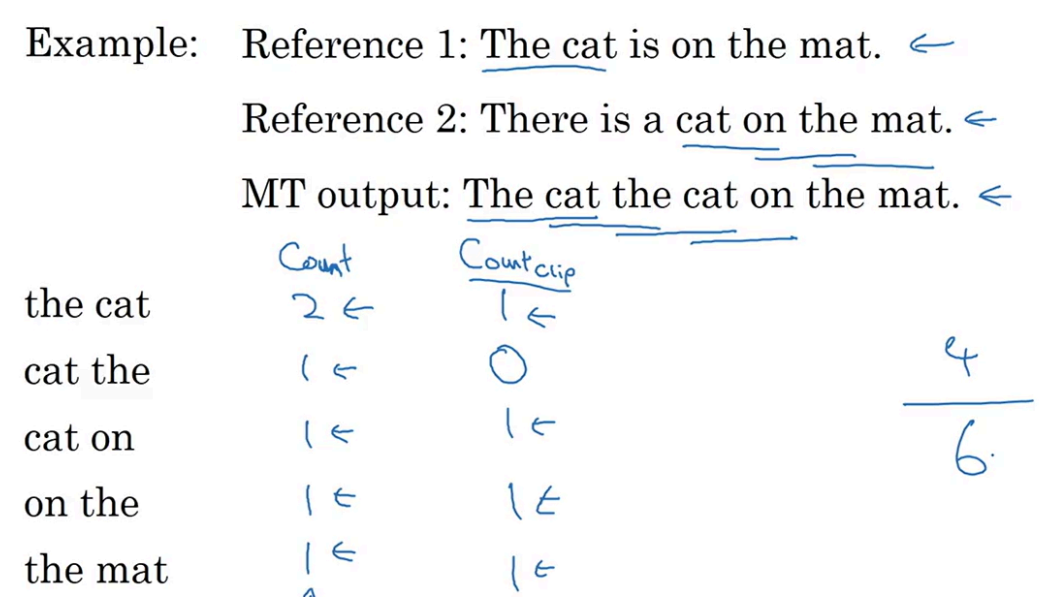
- **Beam Search: What & Why**
  - Beam search is an *approximate (heuristic) search algorithm* for sequence models (e.g., translation, text generation).
  - It keeps only the top B candidate output sequences at each step (common B values: 3, 10, 100).
  - Because it's approximate, beam search doesn't always find the most likely sentence.
- **Problem: Where Do Errors Come From?**
  - Sometimes, the output isn't the best translation or result.
  - Question: Is the error because of beam search itself, or is the underlying RNN (neural network) model bad?
  - If you know which is at fault, you know where your troubleshooting time is best spent.
- **Example Scenario**
  - Input: French sentence "Jane visite l'Afrique en septembre."
  - Human/provided translation ($y^*$): "Jane visits Africa in September."

    - Beam search output ($\hat{y}$): “Jane visited Africa last September.” (bad translation, changed meaning)
    - Two components:
        - RNN model (encoder-decoder), gives scores to sentences (P(y | x).
        - Beam search algorithm, searches for highest scoring sentences.
- **How To Attribute the Error: The Main Logic**
    - Compute both:
        - P($y^*|x$): Probability (by model) of correct (human) translation.
        - P$(\hat{y}|x)$: Probability (by model) of beam search output.
    - Ask: Which is higher?
- **Two Cases for Error Attribution**
    - **Case 1:** $P(y^*|x) > P(\hat{y}|x)$
        - The model says the human translation is more likely than what beam search found.
        - **Conclusion:** Beam search failed to find the best answer. Error is with beam search.
    - **Case 2:** $P(y^*|x) \leq P(\hat{y}|x)$
        - Model prefers beam search output (even if worse translation).
        - **Conclusion:** Model itself is flawed—needs to be improved (training, architecture, regularization).
- **Practical Error Analysis Steps**
    - For errors on your development set:
        1. For each problem, compute the above probabilities.
        2. Attribute each error to *Search (B)* or *Model (R)*.
        3. Tally up—see which source is responsible for more mistakes.
        4. Focus your improvement efforts where they matter most (e.g., wider beam, better model).
    - If using *length normalization* for scoring (longer outputs penalized), compare *normalized* scores.

- **Key Takeaway**
  - This technique helps you fix what's really broken—don't just guess!
  - Works for any approximate algorithm (not just beam search, e.g., greedy search).

# Bleu Score (Bilingual Evaluation Under Study)



# Bleu score on unigrams

Example: Reference 1: The cat is on the mat.

Reference 2: There is a cat on the mat.

MT output: The cat the cat on the mat. ($\hat{y}$)

$$p_1 = \frac{\sum_{\text{unigram} \in \hat{y}} \text{Count}_{\text{clip}}(\text{unigram})}{\sum_{\text{unigram} \in \hat{y}} \text{Count}(\text{unigram})}$$

unigram

n-gram

$$p_n = \frac{\sum_{\text{n-gram} \in \hat{y}} \text{Count}_{\text{clip}}(\text{n-gram})}{\sum_{\text{n-gram} \in \hat{y}} \text{Count}(\text{n-gram})}$$

# Bleu details

$p_n$ = Bleu score on n-grams only $\quad p_1, p_2, p_3, p_4$

Combined Bleu score: $\quad BP \exp\left(\frac{1}{4}\sum_{n=1}^{4} p_n\right)$

BP = brevity penalty

$$BP = \begin{cases} 1 & \text{if MT\_output\_length} > \text{reference\_output\_length} \\ \exp(1 - \text{reference\_output\_length}/\text{MT\_output\_length}) & \text{otherwise} \end{cases}$$

## 1. Why BLEU Score?

- In **machine translation**, there can be many correct translations for a sentence (unlike image recognition, which has one right answer).
- BLEU (Bilingual Evaluation Understudy) is an **automatic metric** to evaluate how close a machine-generated translation is to human references.
- It is used as a **substitute for human evaluation** to speed up development and comparison of translation systems.

## 2. How BLEU Score Works – Intuition

- Given a machine translation and one or more human reference translations, BLEU checks if the words and phrases in the machine output appear in any reference.
- The more overlap (in words and phrases), the higher the BLEU score.

## 3. Precision and Modified Precision

- **Precision**: Fraction of words in the machine output that appear in the references.
  - Example: If output is "the the the the the the the", and references contain "the" multiple times, all words match, so precision = 7/7 = 1.
- **Problem**: This can be gamed by repeating common words.
- **Modified Precision**: Each word in the machine output only gets credit **up to the maximum number of times it appears in any reference**.
  - Example: If "the" appears at most 2 times in references, and output has 7 "the", only 2 are counted. Modified precision = 2/7.

## 4. N-grams: Unigrams, Bigrams, Trigrams, etc.

- BLEU doesn't just look at single words (**unigrams**), but also at sequences of words:
  - **Bigrams**: pairs of words
  - **Trigrams**: triples of words
  - ...up to n-grams (usually up to 4-grams)
- For each n-gram, count how many times it appears in the output and in the references (again, clip to the max in any reference).

## Example: Bigrams

- Output: "the cat the cat on the mat"
- Bigrams: "the cat", "cat the", "cat on", "on the", "the mat"
- For each bigram, count how many times it appears in references (clip if needed).
- Modified bigram precision = (sum of clipped counts) / (total bigrams in output)

## 5. Mathematical Formula for Modified Precision

- For n-grams:
  - P1P_1P1: unigram precision
  - P2P_2P2: bigram precision
  - ...and so on

$$P_n = \frac{\sum_{\text{n-grams in output}} \text{clipped count}}{\sum_{\text{n-grams in output}} \text{count}}$$

# 6. Combining N-gram Precisions

- BLEU combines the modified precisions for 1-gram to 4-gram (by default).
- Instead of a simple average, BLEU uses the **geometric mean**:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\frac{1}{N}\sum_{n=1}^{N} \log P_n\right)$$

  - N is usually 4 (for up to 4-grams)
  - $P_n$ is the modified precision for n-grams
  - exp and log are used for numerical stability (geometric mean)

# 7. Brevity Penalty (BP)

- **Why?** Short translations can get high precision by matching a few words, but may miss important content.
- **BP** penalizes outputs that are **shorter** than the reference:
  - If output length >>> reference length: BP=1
  - If output length ≤ reference length:

    $$\text{BP} = \exp\left(1 - \frac{\text{reference length}}{\text{output length}}\right)$$
- This discourages very short, incomplete translations.

# 8. Summary & Usage

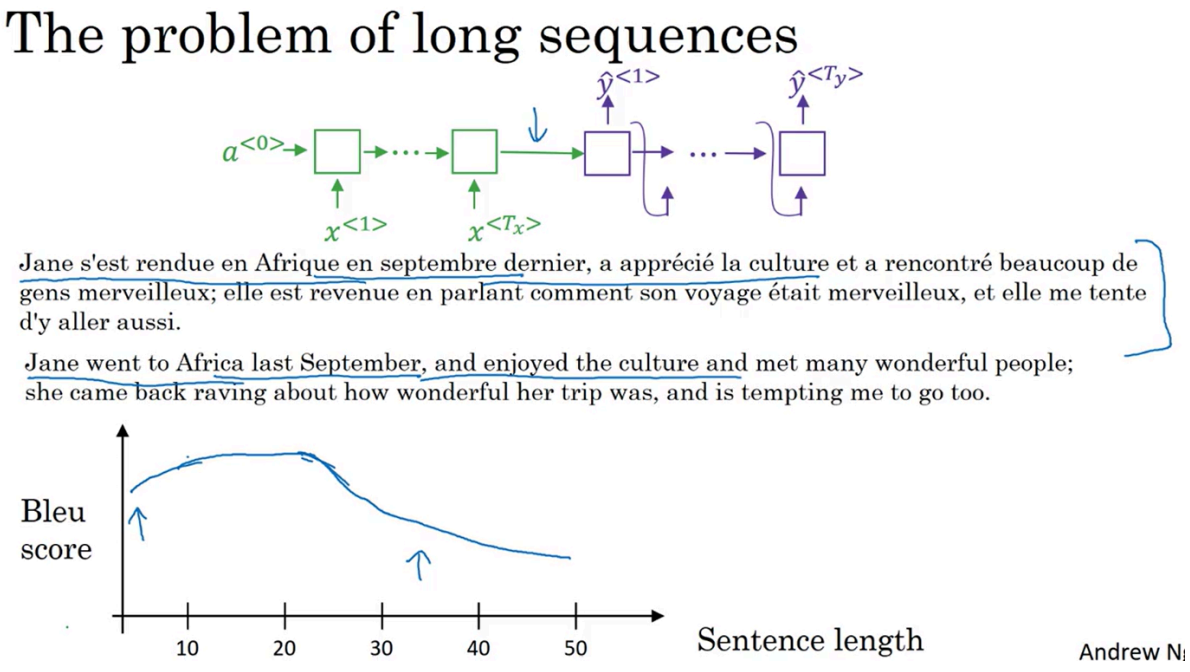
- BLEU gives a **single score** (0 to 1, often shown as 0 to 100) to compare translation systems.

- Widely used for **machine translation** and **image captioning** (where multiple good outputs exist).
- Not used for tasks with only one correct answer (like speech recognition).
- Most people use open-source BLEU implementations; rarely coded from scratch.

## 9. Key Takeaways

- BLEU measures **overlap** between machine output and human references using n-grams and a brevity penalty.
- **Higher BLEU = closer to human translation** (but not perfect; doesn't capture meaning or grammar fully).
- Useful for **automatic, fast evaluation** of text generation systems.

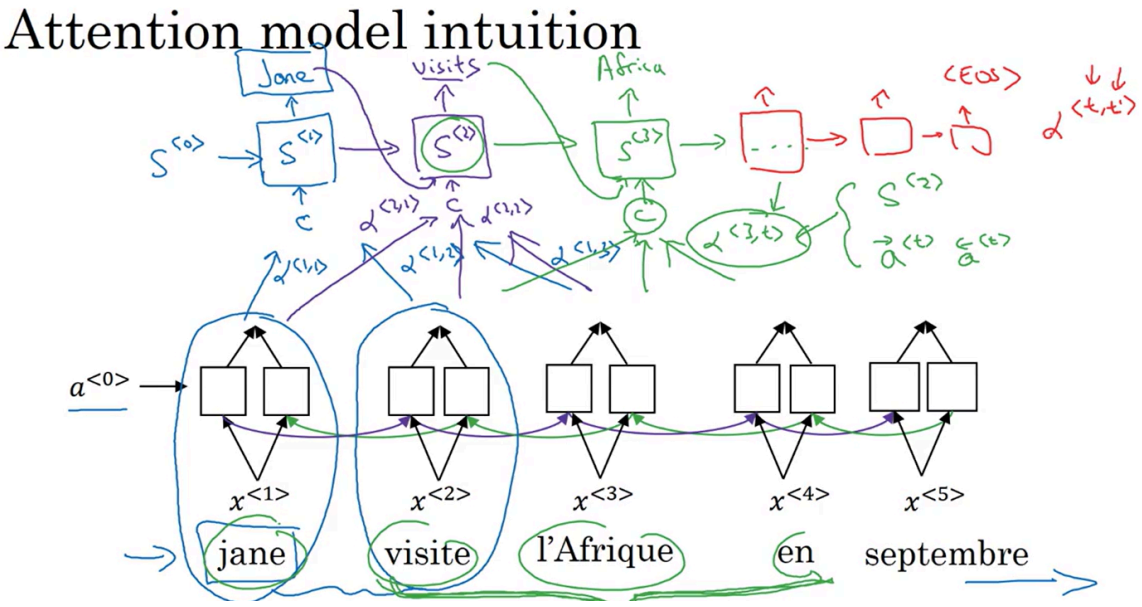
# Attention Model Intuition




- **Encoder-Decoder Limitation**

- Standard seq2seq models use Encoder-Decoder architectures (usually RNNs) for tasks like machine translation.
- Encoder reads the entire input sentence, stores it in a fixed-size vector (bottleneck), and passes it to the decoder.
- **Problem:** For long sentences, it's difficult for the encoder to memorize the entire sequence, leading to degraded translation performance for lengthy inputs.

- **Human Analogy**
  - Human translators don't memorize the whole sentence before translating. They translate segment by segment, referring to relevant input chunks for each output word.
- **Attention Mechanism Motivation**
  - Encoder-Decoder works well for short sentences but BLEU score (translation quality metric) drops sharply for long sentences.
  - Attention lets the model mimic human translation strategies: focus on parts of the input relevant to each output word.
- **Basic Idea of Attention**

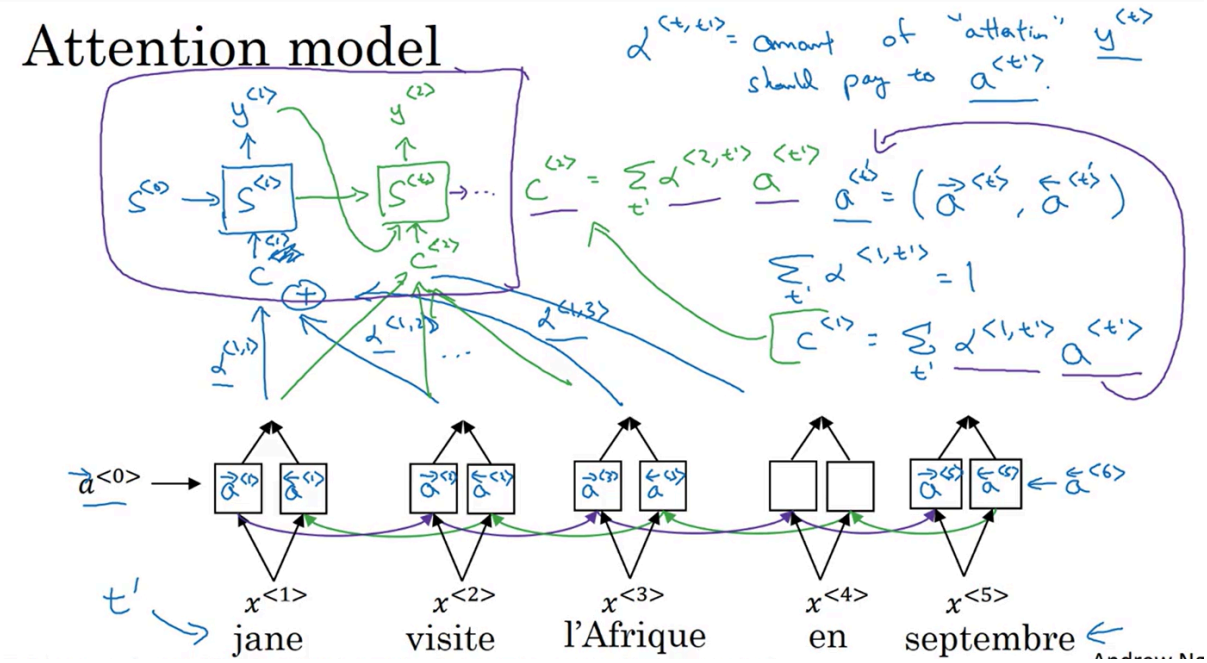


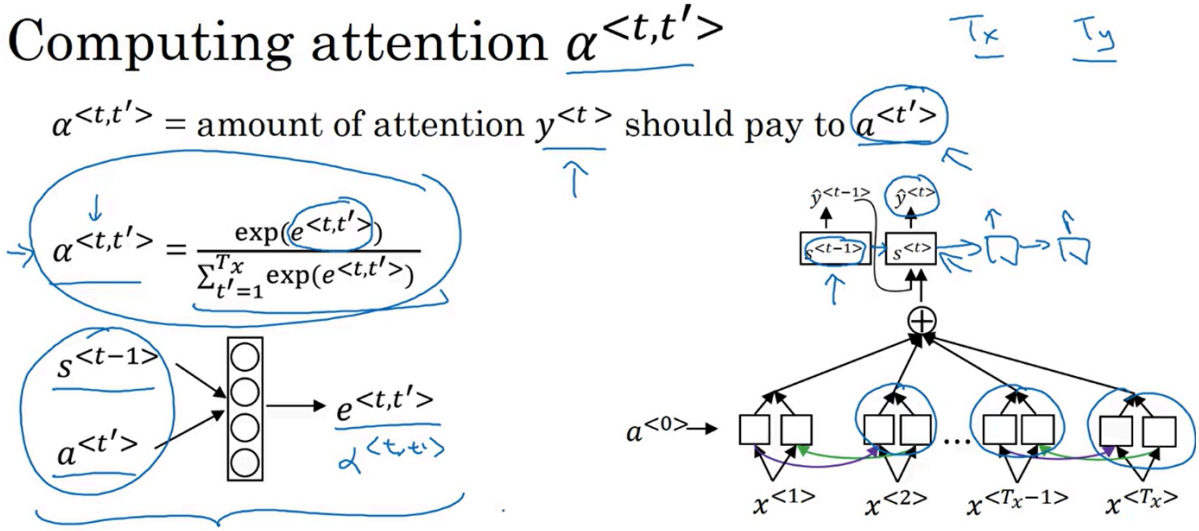

- Instead of relying on a single context vector, the decoder computes a *weighted combination* of encoder hidden states at each output step.
- These *weights* (called attention weights, denoted as α) determine how much focus is given to each word in the input sentence while generating each word in the output.

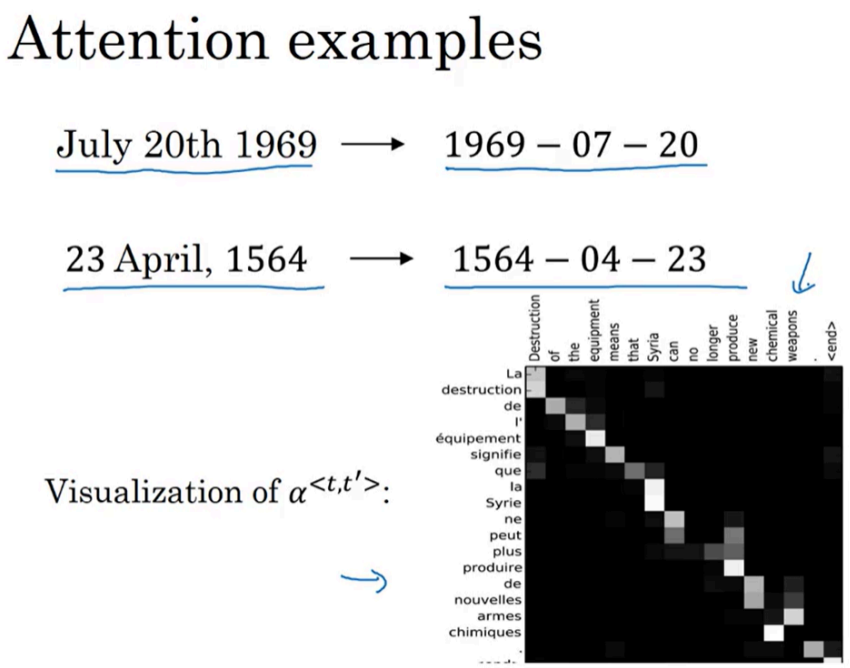
- **Bidirectional RNN Features**
  - The encoder is often bidirectional, producing rich representations for every input word (captures both left and right context in sequence).
- **Generating Output with Attention**
  - For output word t:
    - Compute attention weights $\alpha_{t,i}$ for each input word i.
    - Combine encoder states using these weights to compute a context vector $C_t$.
    - Context vector + previous output/state is used by the decoder to generate the next word.
- **Technical Flow**
  - Decoder's hidden state for step t is $S_t$.
  - At each decoding step:
    - Attention weights ($\alpha_{t,i}$) are computed for each encoder output based on previous state and current input.
    - Context $C_t$ formed as the weighted sum of encodings.
    - Decoder uses $S_t, C_t$ and previous output for generating the next word.
- **Advantages of Attention**
  - Solves the fixed-size bottleneck by dynamically focusing on relevant source parts.
  - Improves accuracy for long sentences; avoids sharp BLEU score drop.
  - More interpretable: can visualize which source words are attended while generating each output token.
- **Impact & Extensions**
  - First formalized in Bahdanau et al. (2015) for machine translation.
  - Foundation for many subsequent deep learning innovations (Transformers, modern NLP models).
  - Attention generalized far beyond translation; used in text, vision, and more.
- **Key Takeaway**
  - Attention enables neural sequence models to process input in a *context-dependent* way, improving translation quality and scalability.

- Each output step selectively attends to different parts of the input, using dynamically computed attention weights, leading to human-like granular translation.

# Attention Model






# 1. Motivation: Why Attention?

- **Problem with vanilla encoder-decoder:**
  - Standard seq2seq models encode the entire input sentence into a single fixed-size vector (the "context vector").
  - For long sentences, this bottleneck makes it hard for the model to remember all details, leading to poor translations.
- **Human analogy:**
  - Human translators don't memorize the whole sentence at once; they focus on relevant parts as they translate each word.
- **Attention mechanism:**
  - Allows the model to "attend" to different parts of the input for each output word, mimicking human translation strategies.

# 2. Architecture Overview

- **Encoder:**
  - Typically a bidirectional RNN (often LSTM or GRU).

    - For each input word at position t′, produces a feature vector $a_{t'}$ (concatenation of forward and backward hidden states).
- **Decoder:**
    - A unidirectional RNN (LSTM/GRU) generates the output sequence one word at a time.
    - At each output time step t, the decoder uses a context vector $c_t$ computed via attention.

# 3. Attention Mechanism: Step-by-Step

# Step 1: Compute Encoder Features

- For each input word (indexed by t′), the encoder outputs $a_{t'}$.
- $a_{t'}$ captures information about the input word and its context (both left and right, due to bidirectionality).

# Step 2: Compute Attention Weights

- For each output time step t, compute attention weights $\alpha_{t,t'}$ for each input position t′.
- **Intuition:** $\alpha_{t,t'}$ measures how much the output word at position t should "pay attention" to the input word at position t′.
- **Properties:**
    - $\alpha_{t,t'} \geq 0$ (non-negative)
    - $\alpha_{t,t'}$ = 1 (sum to 1 for each output step)

# Step 3: Compute Context Vector

- The context vector for output step t is a weighted sum of encoder features:

  $c_t = \sum_{t'} \alpha_{t,t'} \cdot a_{t'}$
- **Intuition:** $c_t$ summarizes the relevant parts of the input for generating the t-th output word.

# Step 4: How are Attention Weights Computed?

- For each output step t and input position t′:

1. Compute a score $e_{t,t'}$ using a small neural network:
    - Inputs: previous decoder hidden state $s_{t-1}$ and encoder feature $a_{t'}$.
    - $e_{t,t'} = \text{score}(s_{t-1}, a_{t'})$
2. Apply softmax over all t′ to get normalized attention weights:
   $\alpha_{t,t'} = \frac{\exp(e_{t,t'})}{\sum_k \exp(e_{t,k})}$

- **Why this works:**
    - The neural network learns to assign higher scores to input positions that are more relevant for the current output word.
    - Softmax ensures the weights are positive and sum to 1.

# 4. Putting It All Together: Decoding with Attention

- At each output time step t:
    1. Use previous decoder state $s_{t-1}$and all encoder features $a_{t'}$ to compute $e_{t,t'}$ and $\alpha_{t,t'}$.
    2. Compute context vector $c_t$ as weighted sum of $a_{t'}$.
    3. Use $c_t$, previous output, and $s_{t-1}$ to compute new decoder state $s_t$ and generate output word $y_t$.
    4. Repeat for each output word.

# 5. Mathematical Summary

- **Encoder output:** $a_{t'}$ for each input position t′.
- **Attention score:**$e_{t,t'} = \text{score}(s_{t-1}, a_{t'})$
- **Attention weights:**$\alpha_{t,t'} = \frac{\exp(e_{t,t'})}{\sum_k \exp(e_{t,k})}$
- **Context vector:**$c_t = \sum_{t'} \alpha_{t,t'} \cdot a_{t'}$
- **Decoder update:**
    - Use $c_t$, $s_{t-1}$, and previous output to compute $s_t$ and $y_t$.

# 6. Intuition and Insights

- **Why use previous decoder state $s_{t-1}$?**

    - It encodes what has been generated so far, helping the model decide which input parts are relevant for the next word.
- **Why a small neural network for $e_{t,t'}$?**
    - The function mapping $(s_{t-1}, a_{t'})$ to a score is complex and best learned from data.
- **Why softmax?**
    - Ensures a valid probability distribution over input positions for each output word.

## 7. Computational Cost

- For input length $T_x$ and output length $T_y$, total number of attention weights is $T_x \times T_y$.
- **Quadratic cost** in sentence lengths, but usually manageable for typical NLP tasks.

## 8. Applications Beyond Translation

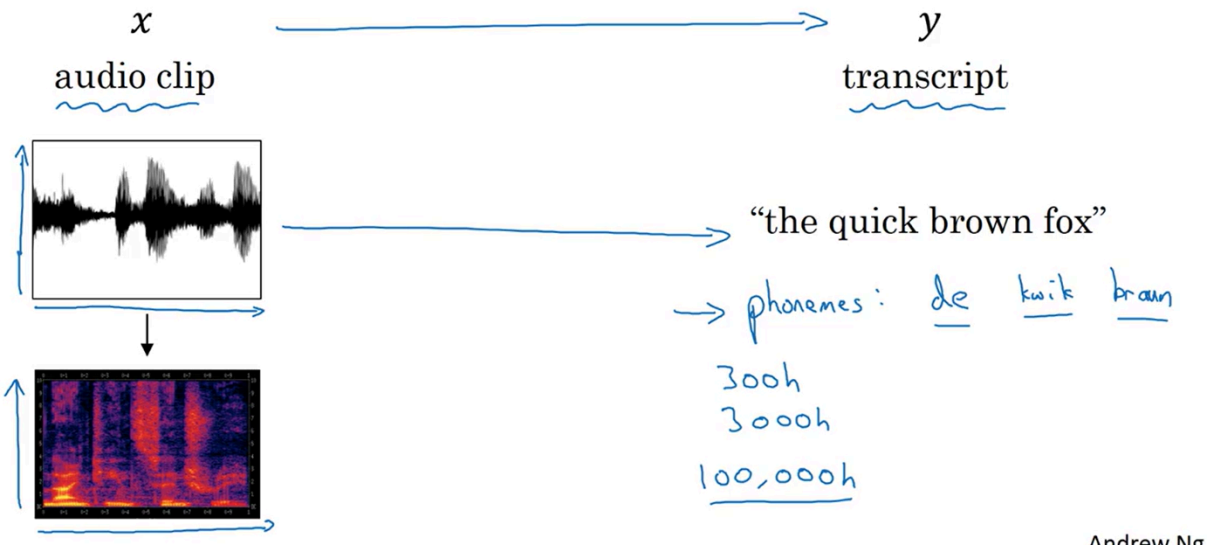
- **Image captioning:**
    - Attention can focus on different image regions for each word in the caption.
- **Date normalization:**
    - Model can attend to relevant parts of a date string to output a normalized format.
- **Visualization:**
    - Attention weights can be visualized to interpret which input parts the model focuses on for each output word.

## 9. Key Takeaways

- **Attention** allows sequence models to dynamically focus on relevant input parts for each output step.
- **Improves performance** on long sequences and makes models more interpretable.
- **Foundation** for modern architectures like Transformers.

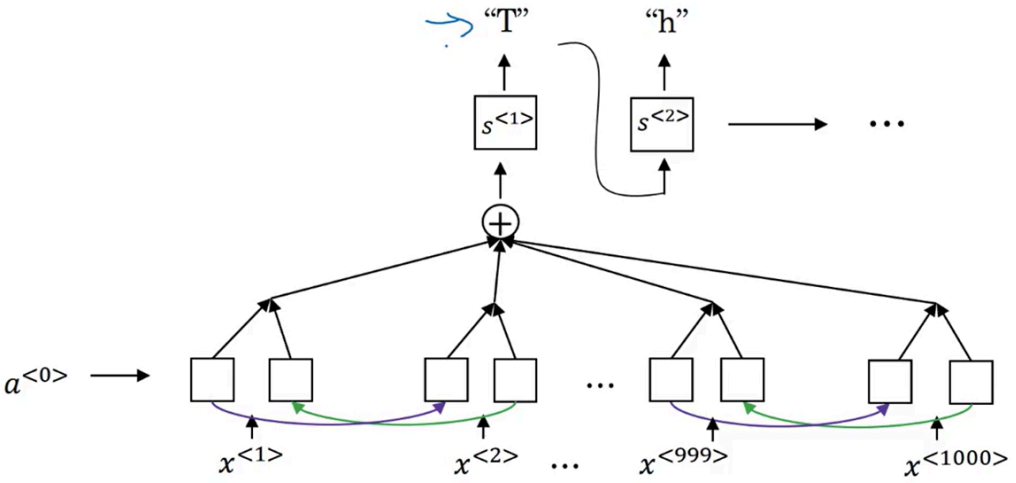
# Speech Recognition

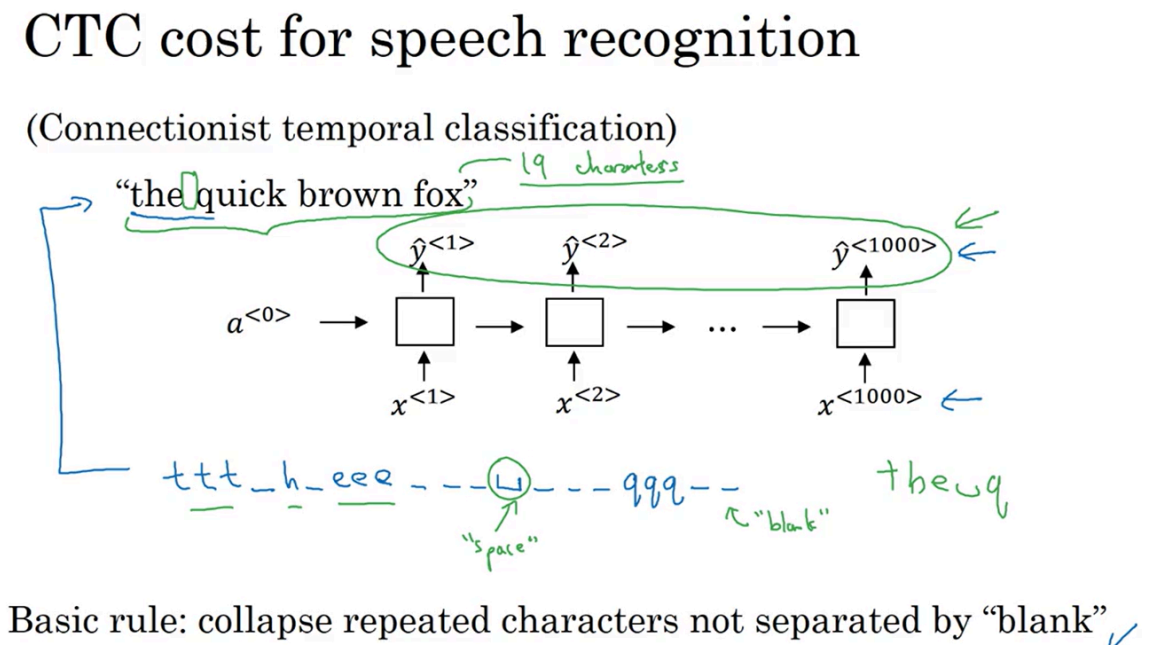
# Speech recognition problem




Andrew Ng

# Attention model for speech recognition






- **Speech Recognition Problem**: Convert an input audio clip x into its text transcript y. The audio signal is a time series measuring air pressure changes.
- **Preprocessing**: Audio is often converted to a *spectrogram*, which plots time (x-axis) vs frequency (y-axis), with color indicating energy (loudness). Spectrograms are similar to how the human ear processes frequencies.
- **Phoneme-based Systems** (Traditional): Earlier speech recognition relied on hand-engineered linguistic units (phonemes), like breaking "the quick brown fox" into basic sounds. These required significant manual effort.
- **End-to-End Deep Learning**: Modern systems bypass phoneme engineering; they directly map audio to transcript using deep neural networks.
- **Data Requirements**:
  - Academic datasets: Several hundred to a few thousand hours.
  - Commercial systems: 10,000 to 100,000+ hours of transcribed audio.
  - Large labeled datasets + deep learning = major advances.
- **Speech Recognition Architectures**:
  - **Attention Models**: Apply attention over audio time frames to produce output transcript.
  - **CTC (Connectionist Temporal Classification)**:

        - Handles input/output length mismatch (more input frames than output letters).
        - Model produces a large output sequence (with possible repetition and "blank" symbols).
        - Collapses repeated characters and removes blanks to yield transcript.
        - Used in large-scale systems (e.g., Baidu Deep Speech).
- **Practical Considerations**:
    - Building production-grade systems demands large datasets.
    - Simpler tasks (e.g., trigger word detection) require less data and will be discussed next.

**Key Formula – CTC Collapsing Rule**:

- Repeated characters (not separated by 'blank') collapse into a single symbol.
    - "blank" (_) allows many outputs to compress to a short transcript.

**Applications**:

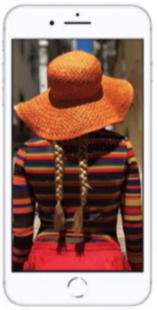
- End-to-end models learn directly from audio to transcript with massive datasets, outperforming older phoneme-based pipelines.
- CTC and attention models are the foundation of modern speech recognition systems.

# Trigger Word Detection

# What is trigger word detection?

Amazon Echo
(Alexa)

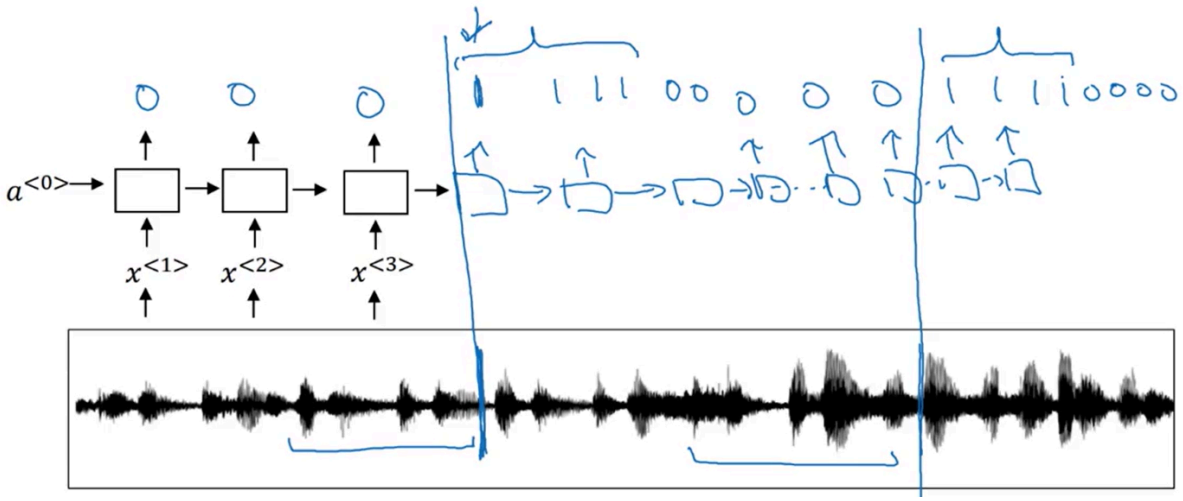

Baidu DuerOS
(xiaodunihao)



Apple Siri
(Hey Siri)

Google Home
(Okay Google)

# Trigger word detection algorithm




**Trigger Word Detection**

- **Trigger Word Detection:**
  - System listens for specific “wake up” words/phrases (e.g., “Alexa”, “Hey Siri”).
  - On detecting the trigger word, device activates/responds.
- **Model Architecture:**
  - Input is an audio clip (e.g., microphone stream).

    - Audio features are extracted (typically as spectrograms): $x_1, x_2, x_3, \ldots$
    - These sequential features are fed into an RNN (like LSTM or GRU).
- **Notations:**
    - $x_1, x_2, x_3, .$**:** Sequence of input audio features (e.g., frames of the spectrogram).
    - **RNN:** Processes the sequence of features to model temporal patterns.
    - **Output/Targets (y)**: Supervised targets, same length as input sequence.
        - For all time steps *before* the trigger word is spoken: y=0
        - Immediately *after* the trigger word: set y=1
        - After some frames, if trigger word reoccurs: output 1 again after that position.
        - For most of the timeline: y=0, resulting in an imbalanced dataset (mostly zeros).
- **Labeling "Hack" for Training:**
    - To reduce imbalance:
        - Instead of just a single "1" right after the trigger word, mark several consecutive output labels as 1 (for a small window after the trigger word).
        - Makes ratio of 1s to 0s a bit more balanced, which may help training but is considered a "hack".
- **Goal:**
    - RNN should output "1" right after the trigger word is said (or for a small window).
    - In practice: you'll play with these labels further in programming exercises.
- **Related Concepts Recap:**
    - You've learned about RNNs (LSTM, GRU), word embeddings, and attention mechanisms.
    - All are used for processing sequence data (text and audio).

**Summary Table of Notation:**

| Symbol | Meaning |
|---|---|
| $x_1, x_2, \ldots$ | Audio features over time (input sequence) |
| RNN | Recurrent Neural Network (LSTM/GRU) |
| y | Output/label sequence |
| y=1 | Trigger word detected at this timestep |

| Symbol | Meaning |
| --- | --- |
| y=0 | No trigger word detected |

**Bottom line:**

For trigger word detection, you convert your audio to a sequence of features ($x_1, x_2, \ldots$), feed this into an RNN, and train it with labels (y) that mark “1” right after the trigger word occurs and “0” everywhere else (sometimes using a small window of 1’s for better balance). Use RNNs for sequential data and consider label balancing tricks to improve learning.