

Sequence Models

Collected by Alexandra Agranovich



2025

# Contents

[Sequence Models 3](#_Toc204532485)

[Examples 3](#_Toc204532486)

[RNNs 5](#_Toc204532487)

[RNN General Architecture 5](#_Toc204532488)

[Forward Propagation 5](#_Toc204532489)

[Loss Function and Overall Loss Calculation 6](#_Toc204532490)

[Backpropagation Through Time 6](#_Toc204532491)

[RNN Architectures 7](#_Toc204532492)

[Language Model and Sequence Generation 8](#_Toc204532493)

[Definitions 8](#_Toc204532494)

[Understanding Language Models 8](#_Toc204532495)

[Building a Language Model with RNNs 8](#_Toc204532496)

[RNN Architecture and Training 8](#_Toc204532497)

[Sampling Novel Sequences 8](#_Toc204532498)

[Vanishing Gradients with RNNs 9](#_Toc204532499)

[Bidirectional RNN 13](#_Toc204532500)

[Deep RNNs 13](#_Toc204532501)

[Application of DL concepts in NLP through word embeddings. 14](#_Toc204532502)

[Word Representation 14](#_Toc204532503)

[Word Embeddings 16](#_Toc204532504)

[Learning Word Embeddings 17](#_Toc204532505)

[Applications Using Word Embeddings 25](#_Toc204532506)

[Various Sequence To Sequence Architectures 29](#_Toc204532507)

[Basic Models 29](#_Toc204532508)

[Picking the Most Likely Sentence 30](#_Toc204532509)

[Beam Search 31](#_Toc204532510)

[Error Analysis in Beam Search 33](#_Toc204532511)

[Bleu Score 34](#_Toc204532512)

[Attention Model 35](#_Toc204532513)

[Speech Recognition 43](#_Toc204532514)

[Trigger Word Detection 44](#_Toc204532515)

[Transformers 49](#_Toc204532516)

[Transformer Network Intuition 49](#_Toc204532517)

[Self-Attention Mechanism 49](#_Toc204532518)

[Multi-Head Attention 51](#_Toc204532519)

[Transformer Network 53](#_Toc204532520)

# Sequence Models

**Paper**: <https://proceedings.neurips.cc/paper_files/paper/2014/file/5a18e133cbf9f257297f410bb7eca942-Paper.pdf>

### Examples

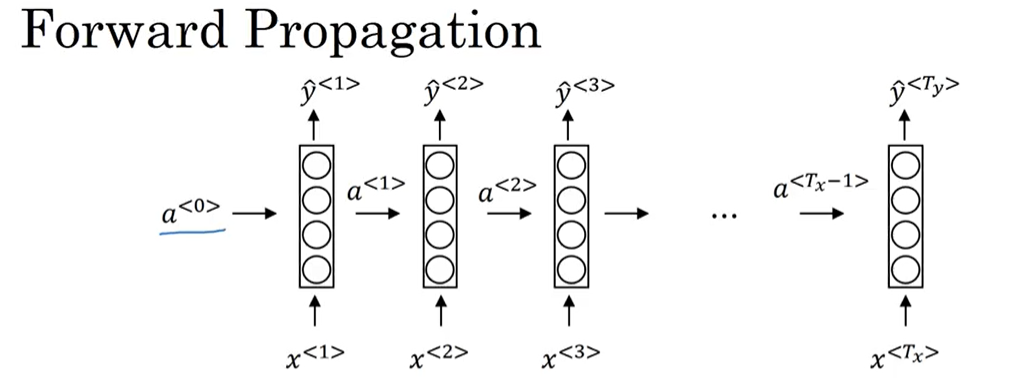
1. **Time Series Data**:
   * **Description**: This type of data consists of observations collected at regular time intervals.
   * **Example**: Stock prices recorded daily over a month. Each price point is a sequence that reflects the stock's performance over time.
2. **DNA Sequences**:
   * **Description**: DNA is represented by sequences of nucleotides, which are the building blocks of genetic material.
   * **Example**: A DNA sequence like "ACGTAGCTAG" consists of the four nucleotides (A, C, G, T) arranged in a specific order, which can be analyzed for various biological functions.
3. **Text Data**:
   * **Description**: Text data consists of sequences of words or characters, often used in natural language processing tasks.
   * **Example**: A sentence like "The cat sat on the mat" is a sequence of words. Each word can be analyzed for sentiment, meaning, or context in tasks like sentiment analysis or machine translation.
4. **Video Frames**:
   * **Description**: Video data is composed of a sequence of frames captured over time, which can be analyzed for activities or events.
   * **Example**: A video of a person walking consists of a series of frames (images) that represent the motion over time. Each frame can be processed to recognize the activity being performed.
5. **Audio Signals**:
   * **Description**: Audio data consists of sound waves represented as a sequence of amplitude values over time.
   * **Example**: A recording of a speech can be represented as a sequence of audio samples, where each sample captures the sound wave's amplitude at a specific moment. This data is crucial for tasks like speech recognition.
6. **User Activity Logs**:
   * **Description**: These logs track user interactions with a system or application over time, forming a sequence of actions.
   * **Example**: A user's navigation path on a website, such as "Home → Products → Cart → Checkout," is a sequence of actions that can be analyzed to understand user behavior and improve user experience.
7. **Sensor Data:**
   * **Description:** Data collected from sensors over time, often used in monitoring and control systems.
   * **Example**: Temperature readings from a weather station taken every hour form a sequence, such as [22°C, 21.5°C, 20°C, 19.5°C]. This data can be analyzed for trends or anomalies.
8. **Game Moves:**
   * **Description**: In gaming, the sequence of moves made by a player can be recorded to analyze strategies or outcomes.
   * **Example:** In a chess game, the sequence of moves like "e4, e5, Nf3, Nc6" represents the actions taken by players, which can be studied for patterns or to improve gameplay.

## RNNs

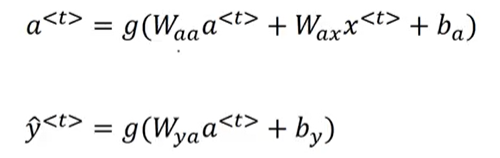
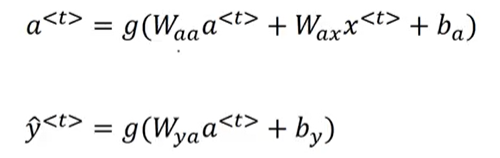
### RNN General Architecture

* The RNN processes input sequentially, passing activation values from one time step to the next.
* Each time step uses information from previous inputs to make predictions, but does not consider future inputs.

### Forward Propagation



**(tanh/ReLU)**



**(sigmoid/softmax)**

Forward propagation involves computing activations from the input sequence step-by-step, using parameters for each timestep.

### Loss Function and Overall Loss Calculation

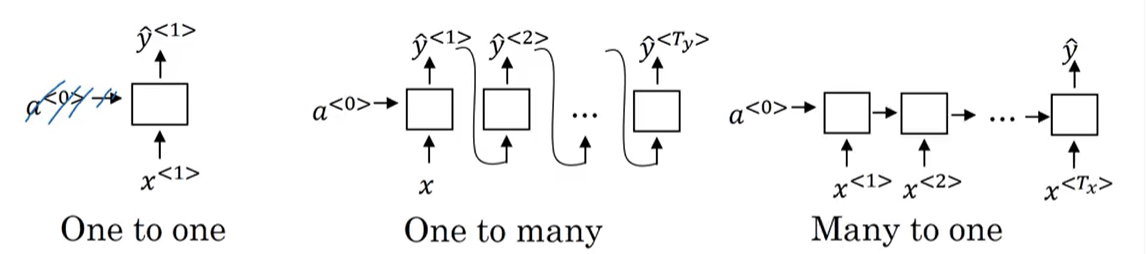
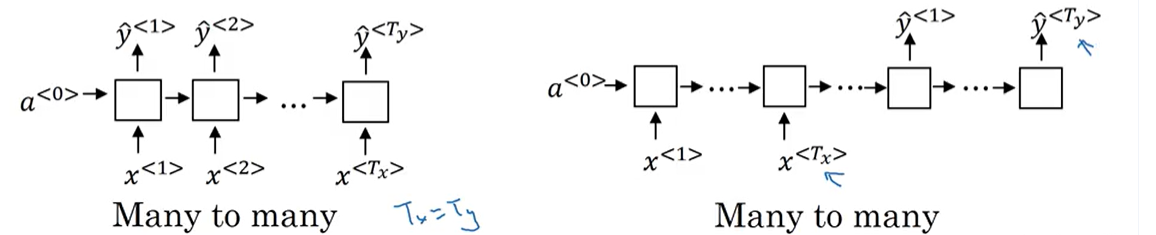
A loss function, such as cross-entropy loss, is defined to measure the difference between predicted and actual outputs.

The overall loss for the entire sequence is calculated by summing the individual losses for each timestep.

### Backpropagation Through Time

* Backpropagation occurs in the opposite direction, where calculations are carried back through the network to update parameters.
* Backpropagation in RNNs involves calculating gradients in the reverse order of the forward pass. This means starting from the last timestep and moving backward to the first.
* The gradients are computed for each weight based on the loss, allowing the network to learn from its errors.
* The calculated gradients are used to update the weights using an optimization algorithm like gradient descent.
* The term "backpropagation through time" describes the process of backpropagation in RNNs, emphasizing the reverse flow of information.
* This method allows for the adjustment of parameters using gradient descent based on the computed losses.
* This process is often referred to as **backpropagation through time (BPTT)**, highlighting the temporal aspect of RNNs where the network learns from sequences of data.

### RNN Architectures

****

**Many-to-Many (Different Lengths)**:

* **Description**: the input and output sequences can have different lengths. The model processes the input sequence and generates an output sequence that may vary in size.
* **Example**: Machine translation, where a sentence in one language (input) is translated into a sentence in another language (output), which may have a different number of words.

**Many-to-Many (Same Length)**:

* **Description**: the input and output sequences have the same length. Each input element corresponds to an output element.
* **Example**: Name entity recognition, where a sequence of words (input) is processed to identify and label entities (output) in the same sequence.

The input is a sequence (e.g., a movie review), and the output is a single value (e.g., a sentiment score)

Both the input and output are single values. Used for tasks where a single input directly maps to a single output.

An example is a simple binary classification task where a single input feature is used to predict a single output label.

The input is a single value (e.g., a genre of music), and the output is a sequence (e.g., generated music notes)

## Language Model and Sequence Generation

### Definitions

* A **corpus** is a large collection of written or spoken texts used for linguistic analysis.
* **Tokenization**is the process of breaking down a text into smaller units called tokens. Tokens can be words, phrases, or even characters, depending on the level of granularity needed. For instance, the sentence "Cats average 15 hours of sleep a day." can be tokenized into the following tokens: ["Cats", "average", "15", "hours", "of", "sleep", "a", "day", "."].

### Understanding Language Models

* A language model estimates the probability of a sequence of words, helping systems like speech recognition determine the most likely sentence.
* For example, it can differentiate between similar-sounding sentences by calculating their probabilities.

### Building a Language Model with RNNs

* To create a language model, you need a large corpus of text, which is tokenized into a vocabulary.
* An end-of-sentence (EOS) token can be added to help the model recognize sentenceboundaries.

### RNN Architecture and Training

* The RNN processes input words sequentially, predicting the next word based on previous words.
* The training involves defining a cost function to minimize the difference between predicted and actual words, allowing the model to learn from a large dataset.

### Sampling Novel Sequences

The process of generating sequences in RNNs involves several key steps:

1. **Initial Input:** Start with an initial input, typically represented as (e.g., a special start token).
2. **Sampling the First Word:**
   * Use the RNN to predict the probability distribution of the next word using a softmax function.
   * Sample the first word y1 from this distribution using a method like np.random.choice.
3. **Iterative Prediction:**
   * For each subsequent time step t:
     + Use the previously sampled word as the input for the next time step .
     + Predict the next word based on the current input and the RNN's hidden state.
4. **Stopping Criteria:**
   * Continue generating words until a stopping condition is met:
     + If an end-of-sentence (EOS) token is included in the vocabulary, stop when this token is generated.
     + Alternatively, set a maximum number of words to generate.
5. **Handling Unknown Tokens:** If the model generates an unknown word token, you can either reject it and resample or include it in the output.

This process allows the RNN to generate coherent sequences based on the learned patterns from the training data.

### Vanishing Gradients with RNNs

**Understanding RNNs and Long-Term Dependencies**

* RNNs are used for tasks like language modeling, but struggle with long-term dependencies in sequences.
* The vanishing gradient problem makes it difficult for RNNs to learn from earlier inputs when processing long sequences.

**Vanishing and Exploding Gradients**

* Vanishing gradients occur when the gradient diminishes as it backpropagates through many layers, hindering learning.
* Exploding gradients, while less common, can cause numerical overflow and are easier to detect.

**Solutions and Next Steps**

* Gradient clipping can help manage exploding gradients by rescaling large gradient vectors.
* Gated Recurrent Units (GRUs) is a solution to the vanishing gradient problem, enabling better handling of long-range dependencies.

#### Gated Recurrent Unit (GRU)

##### Definition

**The Gated Recurrent Unit (GRU)** is a type of RNN designed to capture long-range dependencies in sequential data while addressing the vanishing gradient problem.

##### Key Components

1. **Memory Cell (c):** 
   * The memory cell retains information over time, allowing the model to remember important context, such as singular or plural subjects in a sentence**.**
2. **Output Activation (a):**
   * The output activation at time  t  is equal to the memory cell value:
3. **Update Gate ():**
   * The update gate determines how much of the past information to keep and how much to update. It is computed using a sigmoid activation function:

is the weight matrix

   is the previous hidden state

  is the current input

  is the bias.

1. **Candidate Memory Cell ():**
   * The candidate value for the memory cell is computed as

is the weight matrix for the candidate

is the corresponding bias.

##### Memory Cell Update

The actual memory cell value is updated using the update gate:

This equation allows the model to either update the memory cell with the candidate value or retain the previous value based on the gate's output.

##### Advantages of GRU

* Long-Range Dependencies: The GRU effectively maintains information over long sequences, which is crucial for tasks like language modeling.
* Reduced Vanishing Gradient Problem: The gating mechanism helps mitigate the vanishing gradient issue, allowing for better training of deep networks.

#### Long Short Term Memory (LSTM)

##### Definition

**LSTM (Long Short Term Memory) in Simple Terms:**

* **Purpose**: LSTMs are a type of neural network designed to remember information for long periods, which is important for tasks like language processing or time series prediction.
* **Structure**: They have a special structure that includes three gates:
  + **Forget Gate**: Decides what information to discard from the memory.
  + **Input Gate**: Determines what new information to add to the memory.
  + **Output Gate**: Controls what information to output based on the current memory.

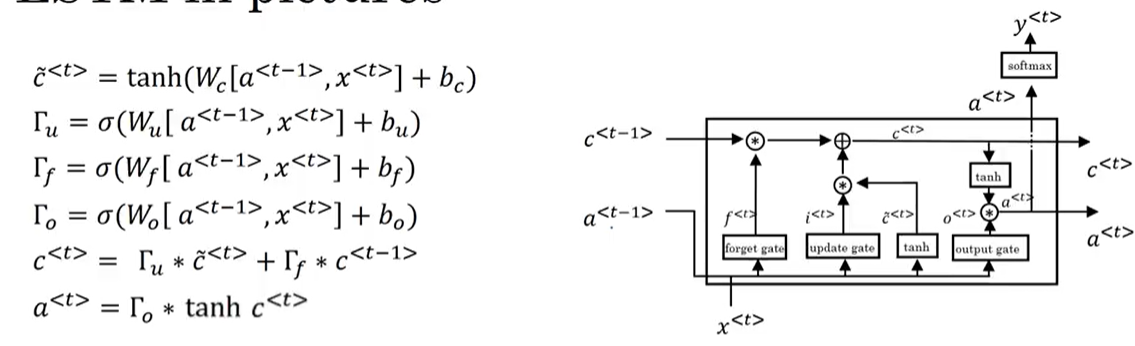
##### How It Works:

1. The forget gate looks at the previous memory and current input to decide what to forget.

2. The input gate checks the current input and previous memory to decide what to add.

3. The memory is updated based on these decisions.

4. Finally, the output gate decides what to send to the next step.



**Candidate Memory Cell**

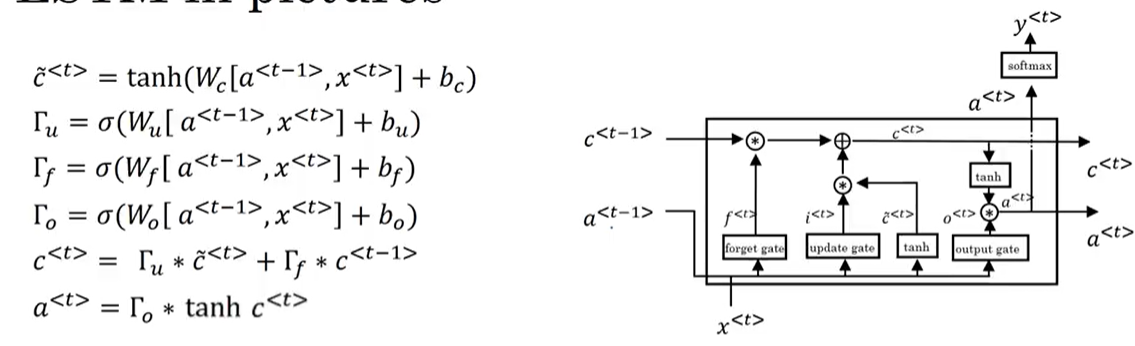
**Update (Input) Gate**

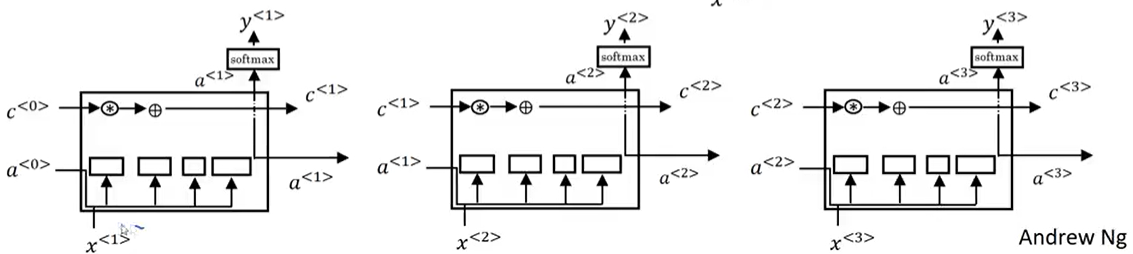
**Forget Gate**

**Output Gate**

**Memory Cell Update**

**Hidden State Update**

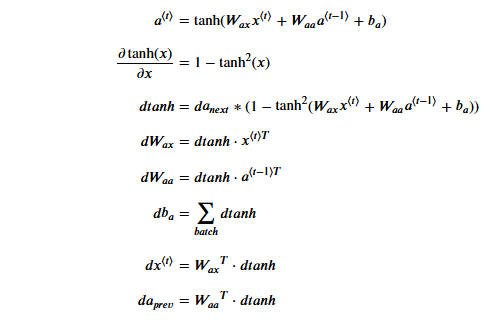


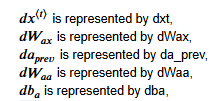


##### Important to remember:

* An LSTM is similar to an RNN in that they both use hidden states to pass along information, but an LSTM also uses a cell state, which is like a long-term memory, to help deal with the issue of vanishing gradients
* An LSTM cell consists of a cell state, or long-term memory, a hidden state, or short-term memory, along with 3 gates that constantly update the relevancy of its inputs:
  + A **forget** gate, which decides which input units should be remembered and passed along. It's a tensor with values between 0 and 1.
    - If a unit has a value close to 0, the LSTM will "forget" the stored state in the previous cell state.
    - If it has a value close to 1, the LSTM will mostly remember the corresponding value.
  + An **update** gate, again a tensor containing values between 0 and 1. It decides on what information to throw away, and what new information to add.
    - When a unit in the update gate is close to 1, the value of its candidate is passed on to the hidden state.
    - When a unit in the update gate is close to 0, it's prevented from being passed onto the hidden state.
  + And an **output** gate, which decides what gets sent as the output of the time step

##### Backward Propagation:





## Bidirectional RNN

 This one let’s you at the point in time to take information from both earlier and later in the sequence.

## Deep RNNs

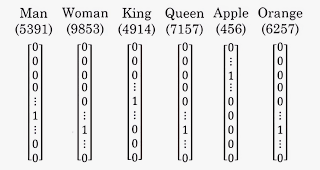
For learning very complex functions sometimes is useful to stack multiple layers of RNNs together to build even deeper versions of these models.

## Application of DL concepts in NLP through word embeddings.

### Word Representation

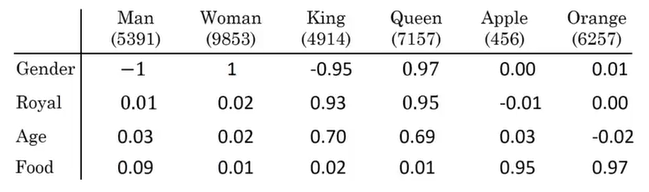
**Understanding Word Representations**

* Traditional word representation uses one-hot vectors, which treat each word independently and do not capture relationships between words.
* This method limits the ability of algorithms to generalize across similar words, making it difficult to predict related terms.



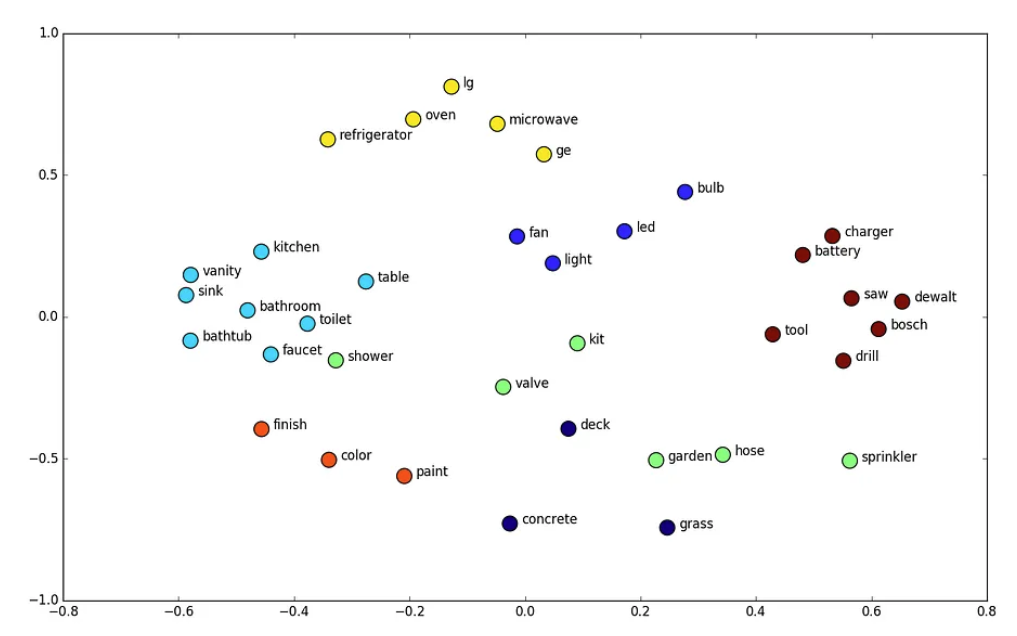
**Introduction to Word Embeddings**

* Word embeddings provide a featurized representation of words, allowing for a more nuanced understanding of relationships (e.g., gender, royalty, age).
* Each word can be represented as a high-dimensional vector, improving the algorithm's ability to recognize similarities between words.
* Word embeddings are dense vector representations of words that capture semantic relationships.
* They allow algorithms to generalize better by recognizing similarities between words, such as "orange" and "apple."



**Visualization and Applications**

* Techniques like t-SNE can visualize these high-dimensional embeddings in lower dimensions, showing how similar words cluster together.
* Word embeddings are crucial for building effective NLP applications, enabling algorithms to learn and generalize better across different contexts.



**Transfer Learning in NLP**

* Transfer learning enables the use of knowledge gained from large, unlabeled text corpora to improve performance on tasks with smaller labeled datasets.
* For example, recognizing "Robert Lin is a durian cultivator" can be inferred from previous knowledge of "orange farmer."

**Training and Fine-Tuning**

* Word embeddings can be learned from extensive text data or obtained from pre-trained models.
* Fine-tuning embeddings during training is beneficial when the labeled dataset is sufficiently large; otherwise, it may not be necessary.

**Applications and Limitations**

* Word embeddings are effective for various NLP tasks like named entity recognition, text summarization, and parsing.
* They are less useful for tasks like language modeling and machine translation when ample task-specific data is available.

**Comparison with Face Recognition**

* The concept of embeddings in NLP is similar to face encoding in recognition tasks, with differences in how they are applied based on the nature of the data.

### Word Embeddings

#### Properties of Word Embeddings

**Understanding Word Embeddings**

* Word embeddings are vector representations of words that capture semantic relationships.
* They can help in analogy reasoning, such as solving "man is to woman as king is to what?"

**Analogy Reasoning with Vectors**

* The difference between vectors (e.g., ) can be similar to other pairs (e.g., ) where **e** refers to the embedding vector associated with a word.
* An algorithm can find a word that fits the analogy by maximizing similarity between vectors.

( Find word what is most similar for a given word by any metric)

**Mostly used kind of similarity: Cosine Similarity**

* Cosine similarity is a common method to measure the similarity between two vectors.
* It calculates the cosine of the angle between vectors, providing a value between -1 and 1, where 1 indicates perfect similarity.

**Why Cosine Similarity is better than Euclidean distance:**

* Euclidean distance is sensitive to the magnitude of the vectors.
* Euclidean distance measures the straight-line distance between two points in space. In contrast, cosine similarity measures the angle between the vectors, focusing on their orientation rather than their distance.
* In summary, using Euclidean distance instead of cosine similarity can lead to misleading interpretations of similarity in NLP tasks, particularly when dealing with high-dimensional word embeddings.

#### Embedding Matrix

**Understanding the Embedding Matrix**

* An embedding matrix ( E ) is created for a vocabulary of 10,000 words, resulting in a 300-dimensional by 10,000-dimensional matrix.
* Each column of the matrix represents the embedding for a specific word in the vocabulary.

**One-Hot Encoding and Matrix Multiplication**

* A one-hot vector is used to represent words, where only one position is set to 1, corresponding to the word's index in the vocabulary.
* Multiplying the embedding matrix  E  by the one-hot vector retrieves the corresponding 300-dimensional embedding vector for that word.

### Learning Word Embeddings

Algorithms for learning word embeddings, highlighting the evolution from complex to simpler methods that yield effective results.

**Understanding Neural Language Models**

* Neural networks can be used to predict the next word in a sequence, which helps in learning word embeddings.
* The process involves creating one-hot vectors for words and using a parameter matrix to obtain embedding vectors.

**Context Types**

1. **Previous Words Context**:
   * Uses a fixed number of preceding words (e.g., the last four words) to predict the next word.
   * Example: Given "I want a glass of," predict "orange."
2. **Surrounding Words Context**:
   * Considers words on both sides of the target word.
   * Example: For the target word "juice," use "a glass of" (left) and "to go along with" (right) as context.
3. **Single Previous Word Context**:
   * Uses only the last word as context.
   * Example: Given "orange," predict the next word.
4. **Nearby Word Context**:
   * Uses a word that is close to the target word.
   * Example: Given "glass," predict a nearby word.

**Target Word Prediction**

* The ***target word*** is the word that the model aims to predict based on the provided context.
* Different contexts can lead to meaningful embeddings, depending on the learning goal (e.g., building a language model vs. learning word embeddings).

#### Word2Vec algorithm

**Word2Vec** is a popular algorithm used to create word embeddings, which are dense vector representations of words in a continuous vector space. Developed by a team led by Tomas Mikolov at Google, Word2Vec captures the semantic meaning of words based on their context in large text corpora. Here are the key aspects of Word2Vec:

**Key Features of Word2Vec:**

1. **Word Embeddings**:
   * Word2Vec transforms words into vectors of fixed dimensions (e.g., 100, 200, or 300 dimensions), allowing for mathematical operations on words.
2. **Contextual Learning**:
   * The algorithm learns word representations based on the context in which words appear. Words that share similar contexts are positioned closer together in the vector space.
3. **Training Models**:
   * Word2Vec primarily uses two architectures:
     + **Skip-Gram Model**: Predicts target words based on a given context word. It is effective for smaller datasets and captures rare words well.
     + **Continuous Bag of Words (CBOW)**: Predicts a target word based on surrounding context words. It is faster and works well with larger datasets.
4. **Applications**:
   * Word2Vec embeddings are widely used in various natural language processing (NLP) tasks, such as sentiment analysis, machine translation, and information retrieval.
5. **Semantic Relationships**:
   * The embeddings can capture semantic relationships, allowing for operations like vector arithmetic. For example, the relationship "king - man + woman" results in a vector close to "queen."

**Summary:**

Word2Vec is a powerful tool for generating word embeddings that represent the meanings of words based on their context in text. It enables various NLP applications by providing a way to understand and manipulate word relationships in a mathematical form.

#### Skip-gram model

The **skip-gram model** in Word2Vec is a technique used to learn word embeddings by predicting target words based on a given context word. Here’s a concise breakdown of how it works:

* **Key Features of the Skip-Gram Model:**
  + **Objective**: The model aims to predict surrounding words (target words) given a specific word (context word) within a defined window size.
* **Training Process (Supervised Learning Setup)**:
  + For each context word, the model randomly selects a target word from a specified window (e.g., ±5 words).
  + It sets up a supervised learning problem where the context word is the input, and the target word is the output.
* **Representation(One-Hot Vector Representation)**:
  + The context word is represented as a **one-hot vector**, which is then transformed into an embedding vector using an embedding matrix:
* **Softmax Function**:
  + A softmax function is used to calculate the probabilities of all possible target words based on the context word's embedding.
  + The probability of a target word given the context word is calculated using the softmax function:

where  - the parameter associated with the target word  t

V - the vocabulary size.

* **Loss Function**:
* The model uses a loss function based on negative log likelihood to optimize the embeddings during training.

where   - the one-hot representation of the target word

 - the predicted probability vector from the softmax output.

* **Hierarchical Softmax**:

To improve computational efficiency (to speed up the computation of the softmax function), hierarchical softmax is introduced, which organizes words into a binary tree structure, allowing for faster classification. Especially when dealing with large vocabularies

**Key Concepts of Hierarchical Softmax:**

1. **Tree Structure**:
   * Instead of treating the vocabulary as a flat list, hierarchical softmax organizes the words into a binary tree structure.
   * Each leaf node of the tree represents a word in the vocabulary, and the internal nodes represent decisions that help classify the words.
2. **Binary Classification**:
   * To determine if a target word is the correct output, the model traverses the tree from the root to the leaf node corresponding to the target word.
   * At each internal node, a binary classifier (often a logistic regression model) is used to decide whether to go left or right in the tree.
3. **Probability Calculation**:
   * The probability of a target word t  given a context word c is calculated as the product of the probabilities at each node along the path from the root to the leaf node for t

where - the probability of taking the left or right branch at node n

1. **Node Probability**:
   * The probability of taking a left or right branch at each node can be computed using the logistic function:

where     - the sigmoid function,

  - the parameter vector for node n,

   - the embedding vector for the context word.

**Advantages of Hierarchical Softmax:**

* **Efficiency**: The computational complexity is reduced from O(V)  (where  V  is the vocabulary size) to O(log V)  since the model only needs to traverse the tree rather than compute probabilities for all words.
* **Scalability**: This method allows the model to handle larger vocabularies more effectively, making it suitable for real-world applications in natural language processing.

#### Negative Sampling

Negative sampling is an efficient alternative to the Skip-Gram model.

**Negative Sampling Overview**

* Negative sampling modifies the learning problem to predict whether a pair of words (context and target) is a valid context-target pair.
* Positive examples are generated by sampling context and target words that appear together, while negative examples are created by randomly selecting words from the dictionary.

**Training the Model**

* The model uses logistic regression to predict the probability of a context-target pair being valid, with a k:1 ratio of negative to positive examples.
* Instead of a large SoftMax classifier, the approach simplifies the problem into multiple binary classification tasks, reducing computational costs.

**Choosing Negative Examples**

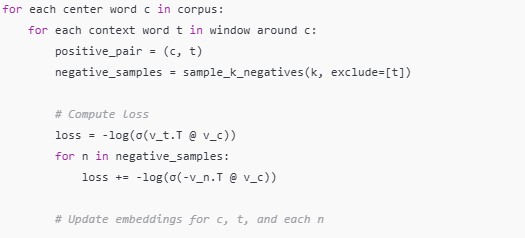
* Negative examples can be sampled based on word frequency, but a balance is needed to avoid bias towards common words.
* A heuristic approach is suggested, sampling words proportional to their frequency raised to the power of 3/4, which has shown effective results in practice:

where:

  - the probability of selecting the word  as a negative example

The frequency of = how often that word appears in the training corpus.

**Pseudocode**



#### GloVe algorithm

**Algorithm Overview**

**GloVe (Global Vectors for Word Representation)** is an algorithm for learning word embeddings based on global co-occurrence statistics from a text corpus.

**Algorithm Overview**

* GloVe constructs a **co-occurrence matrix** ​, where each entry represents how often word j appears in the context of word i.
* It aims to learn embedding of word i that is and embedding of context word j that is such that their dot product approximates the **logarithm** of the observed co-occurrence:

​

where , - bias terms

**Optimization Process**

* The model minimizes the difference between the predicted relationship of words and using a **weighted least squares objective**:

where V - vocabulary size, - weighting function (see below),

* The parameters and e are learned using **stochastic gradient descent (SGD)** or its variants.

**Weighting and Symmetry**

* The **weighting function** ensures that very frequent word pairs do not dominate training, and rare pairs do not introduce noise:

Common choices: α=0.75,

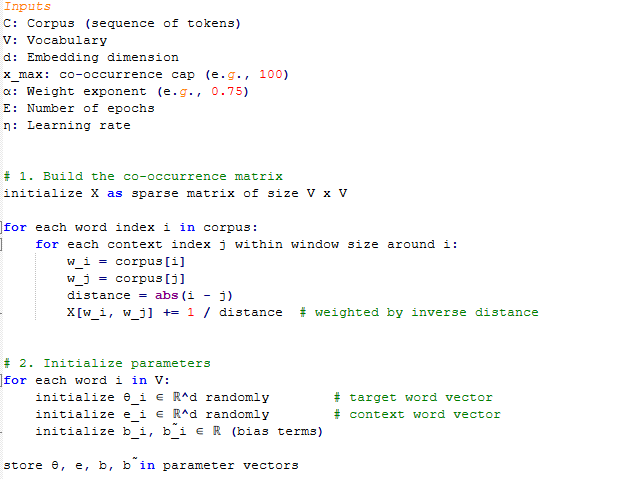
* The roles of and e are symmetric. After training, the final word embedding can be obtained by **averaging**:​

**Interpretability of Embeddings**

* The **individual dimensions** of the learned vectors and are not directly interpretable.
* However, the vectors encode rich **semantic and syntactic relationships** and can support tasks such as:
  + **Analogy solving**: e.g.,

“king”−“man”+“woman”≈“queen”

* **Clustering similar words**
  + **Word similarity evaluation**

**Pseudocode**

### Applications Using Word Embeddings

#### Sentiment Classification

Sentiment classification is a key task in NLP that involves determining whether a piece of text expresses a positive or negative sentiment.

**Understanding Sentiment Classification**

* Sentiment classification aims to predict the sentiment of text, such as star ratings in reviews (e.g., "The dessert is excellent" = 4 stars).
* It can be applied to monitor public sentiment about businesses through social media comments.

**Challenges and Solutions**

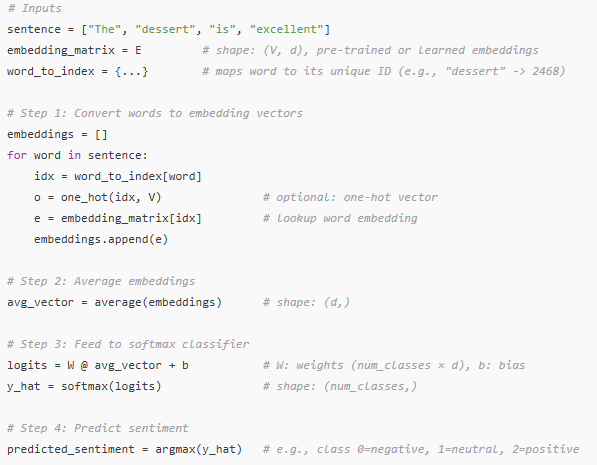
* A common challenge is the lack of large labeled training datasets, which can range from 10,000 to 100,000 words.
* Word embeddings can enhance sentiment classification, allowing effective models even with smaller datasets.

**Modeling Approaches**

* A simple model uses word embeddings to create a feature vector by averaging or summing word vectors, but it may ignore word order.
* A more advanced approach employs Recurrent Neural Networks (RNNs) to consider word sequences, improving sentiment detection by recognizing context and negation.

**Pseudocodes**

1. Simple Sentiment Classification (Bag-of-Embeddings Model)

This model averages word embeddings and passes them to a softmax classifier

1. Sentiment Classification using RNN (Many-to-One)

This version models word sequences using an RNN and takes the final hidden state to classify

sentiment.



*Notes*

|  |  |  |
| --- | --- | --- |
| ***Feature*** | ***Simple Model*** | ***RNN Model*** |
| *Word Order* | ❌ Ignored (avg) | ✅ Preserved |
| *Context Sensitivity* | ❌ Weak | ✅ Strong |
| *Complexity* | 🟢 Simple | 🔴 More complex (requires RNN training) |
| *Input* | Word embeddings | Word embeddings + sequential RNN |

#### Debiasing Word Embeddings

**Understanding Bias in Word Embeddings**

* **Bias** in this context refers to **undesirable associations** learned from real-world text data that reinforce **harmful stereotypes** (e.g., gender, racial, occupational).
* For example, word embeddings trained on large corpora may encode patterns like:

*man−woman ≈ programmer−homemaker*

even though these associations reflect **cultural bias**, not semantic truth.

* Such biases can propagate into **downstream applications**, such as résumé filtering or chatbots, causing discriminatory behavior.

**Methods to Reduce Bias**

*(Following the algorithm introduced by Bolukbasi et al., 2016)*

***1. Identify the Bias Direction***

* First, define the **bias subspace**, typically using **difference vectors**:

*he−she, man−woman, father−mother*

* Use PCA or averaging to determine the **bias direction** ​ in the embedding space (e.g., gender direction).

***2. Neutralization Step***

* For **gender-neutral** words (e.g., *doctor*, *nurse*, *scientist*), remove their projection onto the bias direction:
* This ensures the word embedding no longer contains gender-related information.

***3. Equalization Step***

* For **word pairs** that should be equidistant from neutral concepts (e.g., *grandmother* and *grandfather*), adjust their embeddings such that:
  + Their **midpoint** lies on the neutral subspace
  + Their **distance from the bias axis is equal**
* This avoids unintentionally skewing male/female forms closer to biased terms.

**Classifier Support**

* A binary classifier (e.g., logistic regression) can help:
  + Separate gender-specific definitional words (*woman*, *man*, *mother*)
  + From words that should be neutralized (*engineer*, *nurse*, *leader*)

**Importance of Addressing Bias**

* Bias in embeddings is not just an academic concern—it influences **real-world outcomes**, including:
  + *Hiring algorithms*
  + *Search engines*
  + *Translation systems*
  + *Judicial and medical tools*
* Failing to address bias can **amplify societal inequalities**, while proper debiasing improves

fairness, transparency, and trust in AI.

**Ongoing Research Directions**

* Current limitations:
  + Bias exists beyond binary gender: race, age, religion, etc.
  + Contextualized embeddings (e.g., BERT, GPT) exhibit **dynamic biases**, which require more complex debiasing techniques.
* Areas of active exploration:
  + *Causal debiasing*
  + *Bias auditing tools*
  + *Representation learning with fairness constraints*

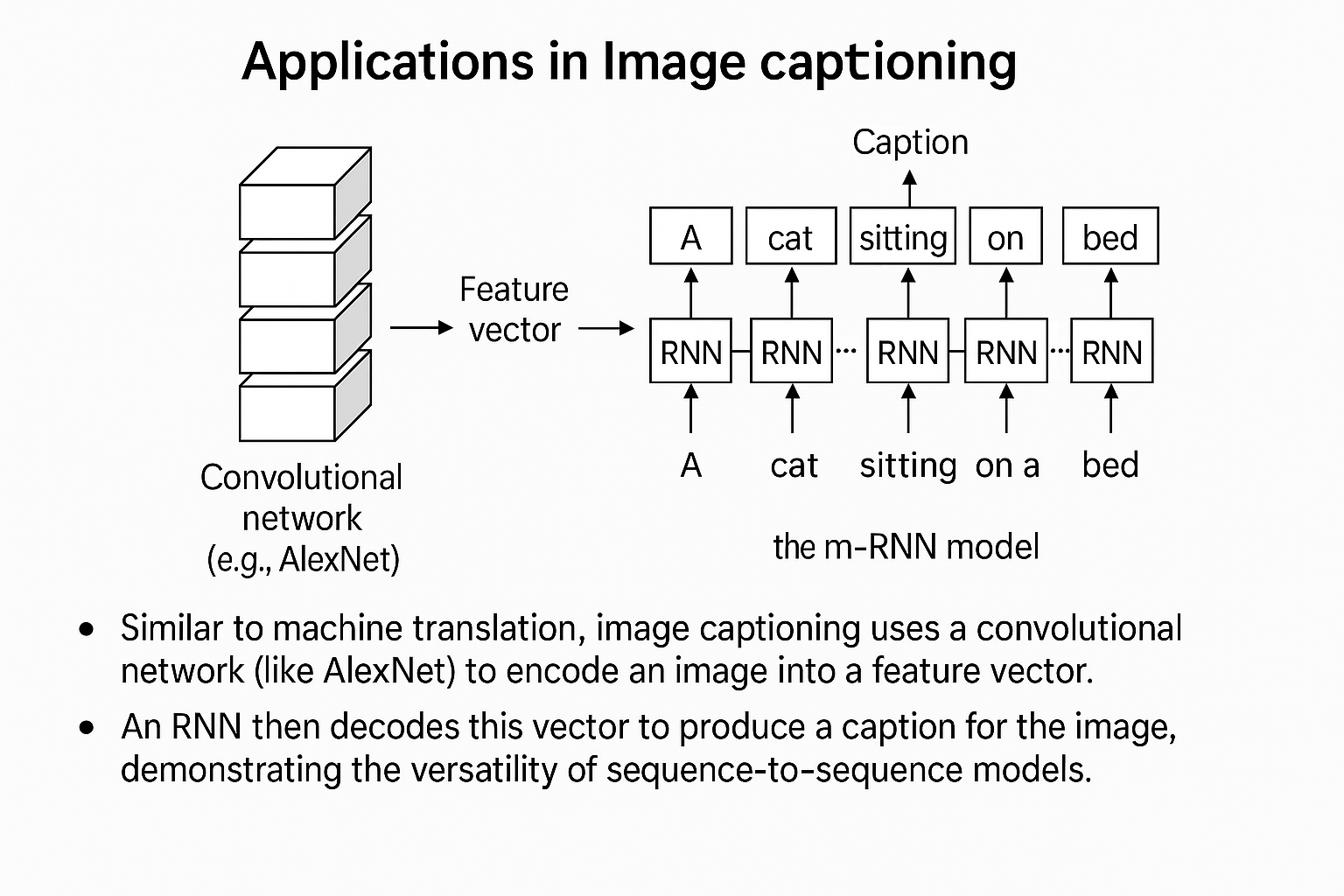
## Various Sequence To Sequence Architectures

Sequence-to-sequence models are essential for tasks like machine translation and speech recognition.

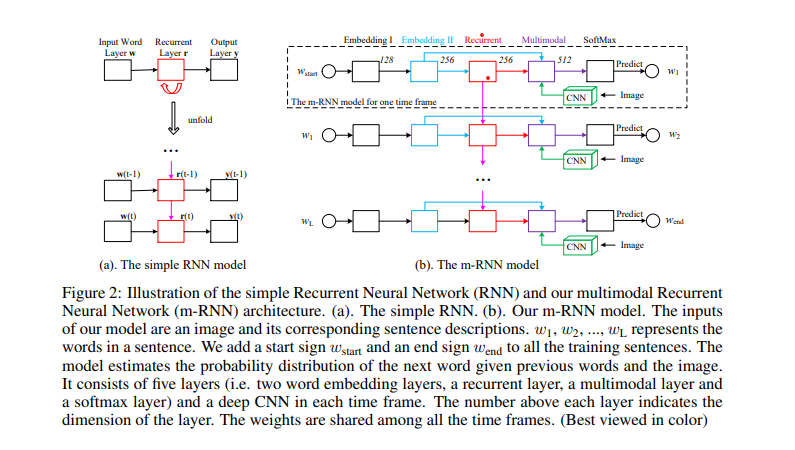
### Basic Models

**Sequence-to-Sequence Models**

* These models consist of an encoder and a decoder network. The encoder processes the input sequence (e.g., a French sentence) and outputs a vector representation.
* The decoder takes this vector and generates the output sequence (e.g., the corresponding English translation) one word at a time.

**Applications in Image Captioning**

* Similar to machine translation, image captioning uses a convolutional network (like AlexNet) to encode an image into a feature vector.
* An RNN then decodes this vector to produce a caption for the image, demonstrating the versatility of sequence-to-sequence models.



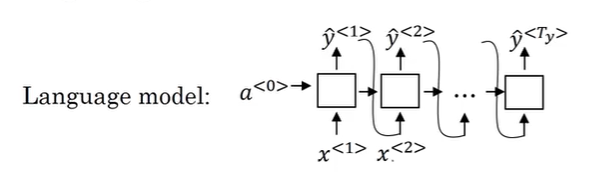
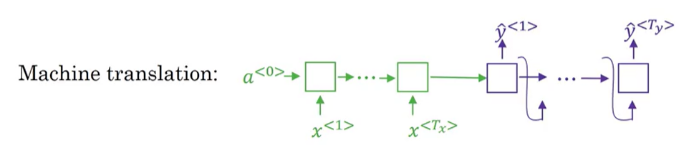
(b) **m-RNN model (**[**https://www.cs.jhu.edu/~ayuille/Pubs15/JunhuaMaoDeepICLR2015.pdf**](https://www.cs.jhu.edu/~ayuille/Pubs15/JunhuaMaoDeepICLR2015.pdf)**)**.

The inputs of model are an image and its corresponding sentence descriptions. *w₁, w₂, ..., w\_L* represents the words in a sentence. We add a start sign *w\_start* and an end sign *w\_end* to all the training sentences. The model estimates the probability distribution of the next word given previous words and the image. It consists of five layers (i.e. two word embedding layers, a recurrent layer, a multimodal layer and a softmax layer) and a deep CNN in each time frame. The number above each layer indicates the dimension of the layer. The weights are shared among all the time frames.

### Picking the Most Likely Sentence

**Machine Translation as Conditional Language Modeling**

* Machine translation can be viewed as a conditional language model, estimating the probability of an output sentence *y* based on an input sentence *x:  P(y|x)*.  This represents the likelihood of (for instance) the English translation conditioned on the French input.



vs

**Finding Optimal Translations**

* The goal is to find the English sentence *y* that maximizes the conditional probability given a French input  *x* :  or more precisely
* The model does not sample outputs randomly; instead, it aims to find the most likely translation: the model is designed to generate translations based on probabilities rather than making random choices.

Here’s a breakdown of this concept:

Key Points:

1. **Probability Distribution**:
   * When the model receives an input sentence (e.g., in French), it calculates a probability distribution over all possible translations (e.g., in English). This distribution indicates how likely each potential translation is given the input.
2. **Maximizing Likelihood**:
   * Instead of randomly selecting a translation from this distribution, the model aims to find the translation that has the highest probability. This is often referred to as finding the **maximum likelihood estimate**.
3. **Deterministic Output**:
   * By focusing on the most likely translation, the model produces a more coherent and contextually appropriate output. Random sampling could lead to nonsensical or irrelevant translations, while maximizing likelihood helps ensure that the translation is meaningful and accurate.
4. **Example**:
   * For an input sentence like "Jane visite l'Afrique en septembre," the model might calculate probabilities for various English translations:
     + "Jane is visiting Africa in September" (high probability)
     + "Jane is going to be visiting Africa in September" (lower probability)
     + "Her African friend welcomed Jane in September" (very low probability)
   * Instead of randomly choosing one of these options, the model selects the translation with the highest probability.

**Greedy Search vs. Optimal Search**

* Greedy search selects the most likely word at each step, which can lead to suboptimal translations. For example, if the first two words are "Jane is," the next word might be chosen based on immediate probability rather than the overall best sequence.
* The number of possible sentences grows exponentially with the length of the sentence. For instance, with a vocabulary of 10,000 words and sentences up to 10 words long, the combinations are:   This highlights the impracticality of evaluating all possible sentences.

**Approximate Search Algorithms**

* Due to the vast search space, approximate search algorithms are employed to find a sentence *y* that maximizes the conditional probability, even if it doesn't guarantee the absolute best solution.
* While approximate algorithms may not always find the optimal solution, they typically perform well enough for practical applications.

### Beam Search

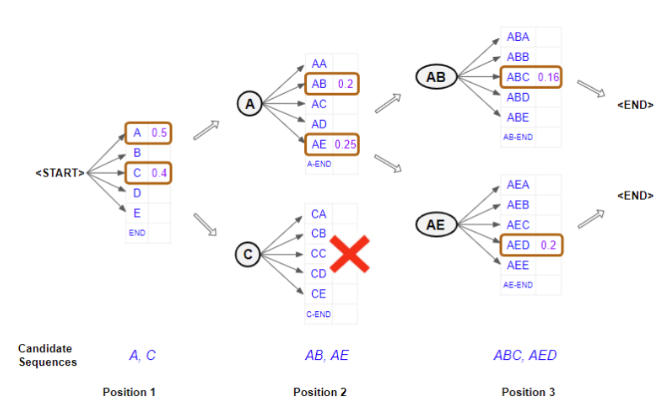
Beam search algorithm is essential for generating the most likely translations in tasks like machine translation and speech recognition.

**Beam Search Algorithm Overview**

* Beam search aims to find the best translation by considering multiple word options at each step, rather than just the most likely one.
* The algorithm uses a parameter called beam width (B) to determine how many alternatives to consider at each step.

**Step-by-Step Process**

* In the first step, beam search evaluates the probabilities of the first word in the translation, keeping track of the top three choices based on their likelihood.
* In the second step, for each of the selected first words, beam search evaluates the probabilities of possible second words, again retaining the top three combinations.

**Iterative Refinement**

* This process continues iteratively, with beam search evaluating the probabilities of subsequent words based on the previously selected words, maintaining the beam width throughout.
* The algorithm ultimately aims to generate a complete and coherent translation by considering multiple possibilities at each stage, leading to better outcomes compared to greedy search methods.

**Refinements to Beam Search**

**Length Normalization**

* Length normalization is introduced to improve the beam search results by addressing the issue of longer sentences having lower probabilities due to the multiplication of many small numbers.

The probability of a sentence P(y) is calculated as:

* By normalizing the probabilities based on the number of words in the output, the algorithm reduces the bias towards shorter translations.

**Logarithmic Transformation**

One can use logarithms to transform the product of probabilities into a sum, enhancing numerical stability and reducing rounding errors:  
  
Maximizing the log probabilities yields the same results as maximizing the original probabilities, making the algorithm more robust.

* A normalized log-likelihood objective:

where

* 𝛼 is another hyperparameter
  + - 𝛼=0 no normalizing
    - 𝛼=1 full normalization

**Beam Width Considerations**

* The choice of beam width affects the performance of the beam search; a larger beam width considers more possibilities but requires more computational resources.
* There is a trade-off between speed and accuracy for large/small beam width.
  + Better results and comprehensive search versus slower performance and higher memory usage for large beam width
  + Faster results and lower memory requirements versus worse results and limited exploration for small beam width

### Error Analysis in Beam Search

* Beam search is a heuristic search algorithm that approximates the most likely sentence but may not always produce the best output.
* The goal is to determine whether errors in translation are due to the beam search algorithm or the RNN model.

**Evaluating Translation Outputs**

* Compare the probabilities of the human-provided translation *y\** and the beam search output using the RNN model: if P(y\*) > P(), beam search is likely at fault; if P(y\*) ≤ P(), the RNN model may be responsible.

**Conducting Error Analysis**

* Analyze multiple examples from the development set to identify which component (beam search or RNN) is causing more errors.
* This process helps decide whether to focus on improving the beam search algorithm or the RNN model for better performance.

Overall, this approach aids in efficiently diagnosing issues in machine translation systems.

### Bleu Score

*Paper:*[*BLEU: a Method for Automatic Evaluation of Machine Translation*](https://www.aclweb.org/anthology/P02-1040.pdf)*by Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu.*

**Understanding the BLEU Score**

The BLEU score evaluates how closely a machine-generated translation aligns with human-generated reference translations. It focuses on the presence of words and phrases (n-grams) in the machine output compared to the references.

**Precision and Modified Precision**

* *Basic Precision*
* *Modified Precision*

Adjusts the count of each word to its maximum occurrence in the references:  
  
whereis the clipped count of word w.

**N-grams and BLEU Calculation**

The BLEU score considers: Unigrams (single words), Bigrams (pairs of words), Trigrams (triples of words), Four-grams (quadruples of words).

The modified precision for n-grams is defined as:

**Final BLEU Score Calculation**

The final BLEU score is calculated by averaging the modified precision scores for unigrams, bigrams, trigrams, and four-grams:  
where BP is the Brevity Penalty:**Overall Importance**

The BLEU score provides a single numerical evaluation metric that has significantly advanced the field of machine translation, allowing for easier comparison and improvement of translation systems.

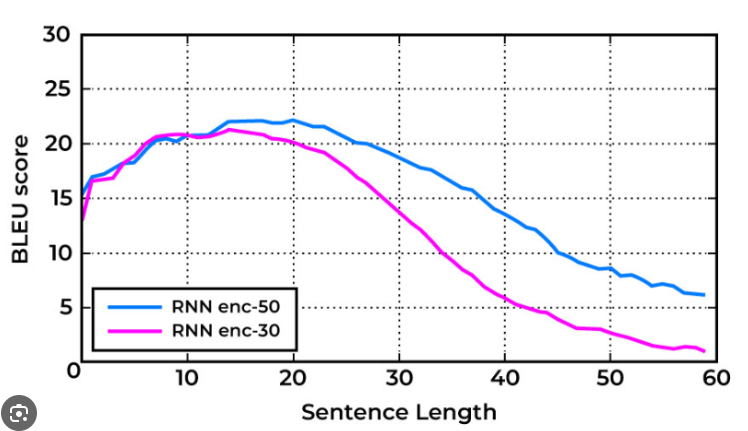
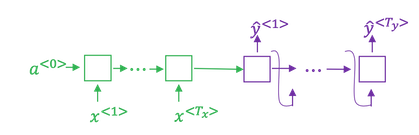
### Attention Model

#### Attention Model Intuition

The Attention Model improves translation by allowing the system to focus on specific parts of a sentence rather than memorizing the entire input.

**Encoder-Decoder Architecture**

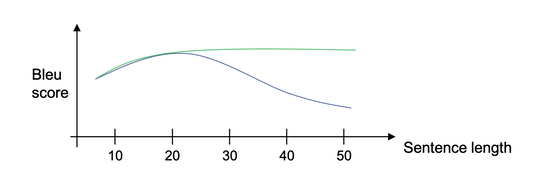
* The Encoder reads the entire input sentence and stores it in activations, while the Decoder generates the output sentence.
* This architecture performs well for short sentences but struggles with longer ones due to memory limitations.



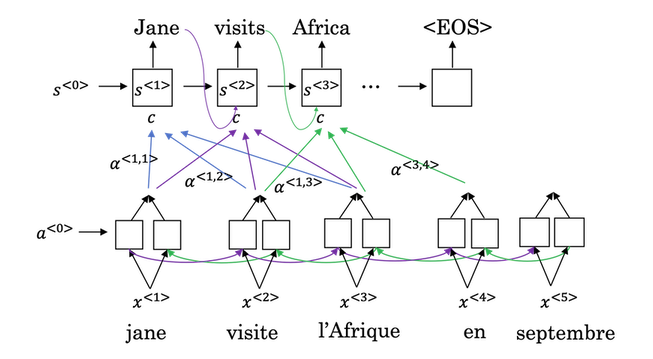
**Mechanism of Attention**

* The model computes attention weights to determine which parts of the input sentence to focus on when generating each word in the output.
* This allows the system to translate one word at a time, improving performance on longer sentences by reducing the need for complete memorization.

The Attention model which translates maybe a bit more like humans looking at part of the sentence at a time. With an Attention model, machine translation systems performance can look like the green line:



#### Overview of the Architecture



Suppose we are translating a source sentence into a target sentence

1. **Encoder (Bottom row) : Create Hidden States**

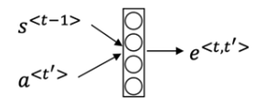
* Processes source sentence mapped to input vectors
* Each token is embedded and processed by a bidirectional RNN: is the **encoder hidden state** (concatenated forward/backward)
* These become **inputs to the attention mechanism**.

**2. Decoder (Top row)**

* The decoder generates words: , ...
* : **decoder hidden state** at time step *t*, computed using:
  + Previous hidden state
  + Previous predicted word
  + Context vector

**3. Attention Mechanism**

At each decoder time step *t*, attention computes:

1. **Alignment Scores (energies)**

Using the **additive score function** (Bahdanau):  
**Parameters involved:**

* ​: weights for decoder state
* ​: weights for encoder hidden state
* ​: scoring vector

**b. Attention Weights**

* Tells how much attention to pay to input word  when generating output word (i.e., this gives a distribution over source words for each target word)
  1. **Context Vector**

* Weighted sum of encoder hidden states
* Summarizes what the decoder should focus on when generating the next word (i.e., summarizes the **most relevant parts** of the input sentence when generating word )
* This context vector is then passed into the decoder RNN to help generate the *t*-th output word.

**d. Generate Decoder Hidden State**

Compute the current decoder hidden state using an RNN (usually a GRU or LSTM):

* Input: previous target token embedding previous state , and context
* Output: updated decoder hidden state

**e. Compute Output Probabilities**

Use the hidden state and context vector to produce the output distribution over the vocabulary:

**f. Sample or take the argmax to generate**

After the decoder computes the **output probability distribution** over the vocabulary:

this gives a vector of probabilities for every word in vocabulary.

Now there are **two common options** for choosing the actual output word :

**1. Take the argmax (greedy decoding)**

* Pick the word with the **highest probability**:
* Deterministic: Always chooses the same token given the same inputs.
* Fast and simple, but may lead to suboptimal overall sequences (e.g., early decisions limit future options).

**2. Sample from the distribution**

* Randomly pick a word according to the **probability distribution**:

(for instance, *y\_t = np.random.choice(vocab, p=softmax(logits))* )

* Introduces **stochasticity** — the same input might lead to different outputs on different runs.
* Used during training (especially in teacher forcing or scheduled sampling), or for **diverse text generation**.

**Analogy:**

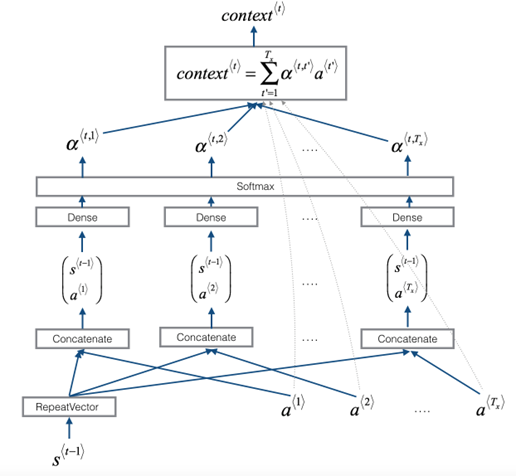
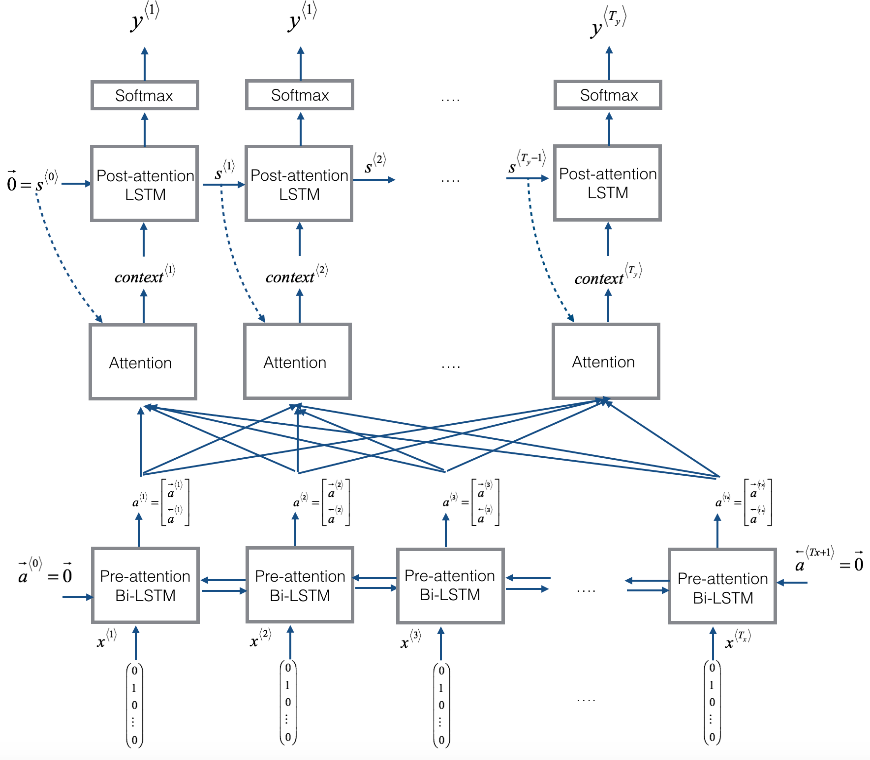
Imagine you're at a vending machine:

* *argmax*: Choose the snack with the highest score.
* *sampling*: Roll a biased die where each snack has a probability, and pick based on that.

The process repeats for each output word until the model generates the end-of-sequence token *⟨EOS⟩.*

**Summary of Parameters**

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Description** | **Learnable?** |
|  | Input embedding of source word | Yes (trainable unless frozen) |
|  | Encoder hidden state | Yes (via encoder RNN) |
|  | Decoder hidden state | Yes (via decoder RNN) |
|  | Weight matrix for decoder state in scoring | ✅ trainable weights inside the attention scoring function. |
|  | Weight matrix for encoder state in scoring | ✅ trainable weights inside the attention scoring function. |
|  | Attention vector (dot product scorer) | ✅ trainable weights inside the attention scoring function. |
|  | Attention weights (softmax outputs) | No (computed dynamically = computed on-the-fly during each forward pass) |
|  | Context vector (weighted sum of encoder states) | No (computed dynamically = computed on-the-fly during each forward pass) |
|  | Combines context and decoder state | ✅ It's a trainable matrix used to linearly transform the concatenated . It is updated during training through backpropagation. |
| *​* | Maps to vocabulary logits | ✅ |
|  | Bias for output logits | ✅ |



*Figure: Neural machine translation with attention*

**Pre-attention and Post-attention LSTMs on both sides of the attention mechanism**

* There are two separate LSTMs in this model (see diagram on the left): pre-attention and post-attention LSTMs.
* *Pre-attention* Bi-LSTM is the one at the bottom of the picture is a Bi-directional LSTM and comes *before* the attention mechanism.
  + The attention mechanism is shown in the middle of the left-hand diagram.
  + The pre-attention Bi-LSTM goes through 𝑇𝑥 time steps
* *Post-attention* LSTM: at the top of the diagram comes *after* the attention mechanism.
  + The post-attention LSTM goes through 𝑇𝑦 time steps.
* The post-attention LSTM passes the hidden state 𝑠⟨𝑡⟩ and cell state 𝑐⟨𝑡⟩ from one time step to the next.

**An LSTM has both a hidden state and cell state**

* In the lecture videos, we were using only a basic RNN for the post-attention sequence model
  + This means that the state captured by the RNN was outputting only the hidden state 𝑠⟨𝑡⟩.
* In this assignment, we are using an LSTM instead of a basic RNN.
  + So the LSTM has both the hidden state 𝑠⟨𝑡⟩ and the cell state 𝑐⟨𝑡⟩.

**Each time step does not use predictions from the previous time step**

* Unlike previous text generation examples earlier in the course, in this model, the post-attention LSTM at time 𝑡 does not take the previous time step's prediction 𝑦⟨𝑡−1⟩ as input.
* The post-attention LSTM at time 't' only takes the hidden state 𝑠⟨𝑡⟩ and cell state 𝑐⟨𝑡⟩ as input.
* We have designed the model this way because unlike language generation (where adjacent characters are highly correlated) there isn't as strong a dependency between the previous character and the next character in a YYYY-MM-DD date.

**Concatenation of hidden states from the forward and backward pre-attention LSTMs**

* 𝑎→⟨𝑡⟩: hidden state of the forward-direction, pre-attention LSTM.
* 𝑎←⟨𝑡⟩: hidden state of the backward-direction, pre-attention LSTM.
* 𝑎⟨𝑡⟩=[𝑎→⟨𝑡⟩,𝑎←⟨𝑡⟩]: the concatenation of the activations of both the forward-direction 𝑎→⟨𝑡⟩ and backward-directions 𝑎←⟨𝑡⟩ of the pre-attention Bi-LSTM.

**Computing "energies"**𝑒⟨𝑡,𝑡′⟩**as a function of**𝑠⟨𝑡−1⟩**and**𝑎⟨𝑡′⟩

* Recall in the lesson videos "Attention Model", at time 6:45 to 8:16, the definition of "e" as a function of 𝑠⟨𝑡−1⟩ and 𝑎⟨𝑡⟩.
  + "e" is called the "energies" variable.
  + 𝑠⟨𝑡−1⟩ is the hidden state of the post-attention LSTM
  + 𝑎⟨𝑡′⟩ is the hidden state of the pre-attention LSTM.
  + 𝑠⟨𝑡−1⟩ and 𝑎⟨𝑡⟩ are fed into a simple neural network, which learns the function to output 𝑒⟨𝑡,𝑡′⟩.
  + 𝑒⟨𝑡,𝑡′⟩ is then used when computing the attention 𝛼⟨𝑡,𝑡′⟩ that 𝑦⟨𝑡⟩ should pay to 𝑎⟨𝑡′⟩.
* The diagram on the right of figure 1 uses a RepeatVector node to copy 𝑠⟨𝑡−1⟩'s value 𝑇𝑥 times.
* Then it uses Concatenation to concatenate 𝑠⟨𝑡−1⟩ and 𝑎⟨𝑡⟩.
* The concatenation of 𝑠⟨𝑡−1⟩ and 𝑎⟨𝑡⟩ is fed into a "Dense" layer, which computes 𝑒⟨𝑡,𝑡′⟩.
* 𝑒⟨𝑡,𝑡′⟩ is then passed through a softmax to compute 𝛼⟨𝑡,𝑡′⟩.
* Note that the diagram doesn't explicitly show variable 𝑒⟨𝑡,𝑡′⟩, but 𝑒⟨𝑡,𝑡′⟩ is above the Dense layer and below the Softmax layer in the diagram in the right half of figure 1.

#### visualize-alphaVisualization of the attention weights

* Each column is a target word (i.e., for decoder time step *t*).
* Each row is a source word (encoder time step t′).
* Bright cells = high attention weight *α(t,t′)* — that encoder word is strongly influencing the output word.

#### Applications and Challenges

* The attention mechanism is applicable in various tasks, such as machine translation and image captioning.
* The algorithm has a quadratic time complexity:  
    
   is the number of input words and is the number of output words.

#### Additional Clarifications

##### Step-by-Step: Encoder in Bahdanau Attention

The encoder processes the entire source sentence and produces a sequence of hidden states that represent the semantic information for each input word.

**1. Input: Source Sentence Tokens**

Let the input/source sentence be a sequence of tokens:

Each is a word (or sub word), which is embedded into a dense vector via an embedding layer:

**2. Encoder RNN (Usually Bidirectional)**

The embedded input sequence is fed into a bidirectional RNN, typically a Bi-LSTM or Bi-GRU.

Each token is encoded into a hidden state that contains:

* Forward context: from beginning up to
* Backward context: from to end

So for each position *t′*, the encoder produces:

These are concatenated:

Result:

Encoder outputs:

Each represents a rich contextual embedding of the *t′* -th source word.

**3. All Encoder Hidden States Are Stored**

All hidden states are stored and passed to the attention mechanism in the decoder.

These hidden states are:

* The “memory” the decoder looks at when producing each target word.
* Used to compute alignment scores for each decoder time step *t*.

**Summary of Encoder Step**

|  |  |
| --- | --- |
| **Component** | **Description** |
| Input | Sequence of source tokens |
| Embedding Layer | Converts each token to dense vector |
| Bi-RNN | Processes embeddings in both directions |
| Hidden States | , one per source token |
| Output | Sequence of encoder hidden states passed to attention module |

### Speech Recognition

**Understanding Speech Recognition**

* The speech recognition problem involves converting an audio clip into a text transcript.
* Audio clips are represented as changes in air pressure over time, which can be visualized as waveforms.

**Pre-processing Audio Data**

* A common pre-processing step is generating a spectrogram, which displays audio intensity across different frequencies over time.
* This step mimics how the human ear processes sound, allowing for better input to recognition algorithms.

**Evolution of Speech Recognition Systems**

* Traditional systems relied on **phonemes**, which are basic units of sound, but modern deep learning approaches can directly convert audio to text without these representations.
* The effectiveness of these systems has improved significantly with larger datasets, with commercial systems now trained on over 10,000 hours of audio.

**Building Speech Recognition Systems**

* Two methods for building these systems include using attention models and the Connectionist Temporal Classification (CTC) cost function:
  + ***An attention-based model for speech recognition*** can be designed by mapping time frames of the audio input along the horizontal axis and using the attention mechanism to generate the corresponding transcript.
  + ***CTC*** **cost for speech recognition** (*Paper:*[*Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks*](https://www.cs.toronto.edu/~graves/icml_2006.pdf)*by Alex Graves, Santiago Fernandes, Faustino Gomez, and Jürgen Schmidhuber*) allows flexible output by collapsing repeated characters and inserting blank characters, enabling the network to handle varying input and output lengths.

In practice, models for speech recognition typically use **bidirectional LSTMs or GRUs**, and often have **multiple layers** (i.e., are deeper). The number of time steps in the input sequence is usually **very large**. In speech recognition, the number of **input time steps** is typically **much greater** than the number of **output time steps**.

For example, if you have a 10-second audio clip sampled at 100 Hz (i.e., 100 feature vectors per second), you’ll end up with **1,000 input time steps**. However, your transcript likely contains far fewer characters—certainly not 1,000.

To handle this mismatch, the **CTC (Connectionist Temporal Classification)** loss function is used. CTC allows the RNN to output sequences like:

ttt\_h\_eee\_\_\_[]\_\_\_qqq\_\_,

where \_ represents a “blank” symbol, and [] represents a space.

The key rule in CTC is: **collapse repeated characters unless they are separated by a blank.**

So the sequence above would decode to: **"the q"**

### Trigger Word Detection

**Trigger word detection systems** are widely used in voice-activated assistants like **Amazon Echo ("Alexa")**, **Apple Siri ("Hey Siri")**, and **Google Home ("OK Google")**. These systems enable users to interact with devices using specific phrases, enhancing the hands-free user experience.

* **How Trigger Word Detection Works**

At a high level, the goal is to detect when a particular word (like “activate” or “Alexa”) has just been spoken in an audio clip. This is typically done using a **RNN**, often with *spectrogram features* extracted from the raw audio.

**Feature Extraction:**

* Audio is converted into features (e.g., spectrogram or MFCC).
* These features are passed through an RNN (or a more advanced architecture like a GRU/LSTM network).
* **Target Labeling and Training Strategy**

During training, we need to assign target labels to each time step:

* **Background audio** (no trigger word):
  + **Trigger word present (e.g., “activate”):**

* + **Problem: Class Imbalance**

Since the trigger word only occurs briefly in long audio sequences, this results in:

* **Highly imbalanced training data** (many more 0s than 1s).
* This imbalance can make the model harder to train effectively.
  + **Practical Hack: Label Multiple Positive Time Steps**

To address this, we apply a common technique:

Instead of setting  **for only one time step,** we label **multiple consecutive steps** after the trigger word finishes (e.g., 50 time steps) with 1s.

This has two benefits:

* Improves **class balance** in the training set.
* Helps the model learn to better **associate the audio pattern** with the trigger word.
  + **Example Use Case**

Suppose you insert the word “activate” into a background clip:

* You update the labels for the next 50-time steps after the word ends.
* This means the model doesn't have to hit a single exact frame to succeed — it gets a **"detection window."**

|  |  |
| --- | --- |
| **Component** | **Description** |
| **Goal** | Detect when a trigger word like "Alexa" is spoken |
| **Model** | RNN (often with spectrogram inputs) |
| **Input Features** | Time steps from audio |
| **Labels** | 0 for silence/background, 1 when trigger word ends |
| **Training Hack** | Label multiple 1s after word to mitigate imbalance |
| **Use Case** | Voice-activated assistants and smart devices |

* **Summary**

#### Additional clarifications: spectrogram and MFCC

##### Spectrogram

A **spectrogram** is a **visual and numerical representation of sound**, showing how the **frequency content** of an audio signal changes over time.

**It’s essentially**

A 2D image where:

* The **horizontal axis (x)** represents **time**
* The **vertical axis (y)** represents **frequency**
* The **color or intensity** at each point shows the **amplitude (energy)** of that frequency at that moment in time

**Why Is It Useful?**

Raw audio is just a waveform (amplitude vs. time), which isn’t very informative for models.  
A **spectrogram reveals the structure of the sound**, like:

* Spoken phonemes
* Background noise
* Musical notes
* Trigger words like “Alexa”

**How It’s Computed:**

The spectrogram is typically generated using the **Short-Time Fourier Transform (STFT)**:

1. **Slice** the audio into overlapping short time windows (e.g., 25ms).
2. For each window, compute the **Fourier Transform** to get the frequency spectrum.
3. Stack these spectra over time → get a matrix:

**Visual Example (simplified):**

Time →

Freq ↓ ┌──────────────┐

│ ••••• │ ← low freqs (e.g. vowels)

│ ••••••• │ ← mid freqs

│ ••• │ ← high freqs (e.g. 's' or 't' sounds)

└──────────────┘

↑ Energy (color/brightness)

**In Deep Learning:**

* Spectrograms turn audio into **images** that can be processed by **CNNs**, **RNNs**, or **transformers**.
* Variants:
  + **Mel spectrogram**: frequencies mapped to the **Mel scale** (how humans perceive pitch)
  + **MFCC** (Mel-frequency cepstral coefficients): popular for speech features

##### MFCCs (Mel-Frequency Cepstral Coefficients)

**MFCCs** are a type of feature extracted from audio, especially popular in **speech recognition** and **speaker identification**.

They summarize the **short-term power spectrum** of a sound and mimic how **humans perceive audio** — focusing more on how we hear than what a machine “sees” in raw frequencies.

**Intuition**

* Humans **don’t perceive pitch linearly**: we’re more sensitive to frequency differences at **lower frequencies** than higher ones.
* MFCCs simulate this by transforming audio to the **Mel scale**, which is **nonlinear and perceptually motivated**.

**How MFCCs Are Computed (Step-by-Step):**

Given an audio signal x[n], where n indexes time-domain samples.

**1. Pre-emphasis (optional)**

Boosts high frequencies

(typically α≈0.95)

**2. Frame the signal**

Slice into short overlapping windows (e.g., 25ms each).

* Frame length: *N*
* Overlap: usually 50%

Each frame:

* *m*: frame index
* *H*: hop size (e.g., *H = N/2*)

**3. Windowing**

Apply a Hamming window to reduce edge effects (e.g., multiply each frame by a window function).

where

**4. Discrete Fourier Transform (DFT)**

Convert each frame from time to frequency domain:

()

Compute the **power spectrum**:

**5. Mel Filter Bank**

Pass the power spectrum through a set of triangular filters **spaced on the Mel scale**. Each filter emphasizes certain frequencies.

Apply the filters to the power spectrum:

**6. Logarithm**

* Take log of the filter bank energies to compress dynamic range (simulates loudness perception):

logS[m]=log(S[m])

**7. Discrete Cosine Transform (DCT)**

* Decorrelate the log energies and keep only the **first N coefficients** (usually 12–13):

##### MFCC vs. Spectrogram

|  |  |  |
| --- | --- | --- |
| **Feature** | **Spectrogram** | **MFCC** |
| Based on | Raw frequency spectrum | Mel-filtered, compressed version |
| Perceptual? | No | Yes (Mel scale + log) |
| Dimensionality | High (many frequencies) | Lower (e.g., 13–20 coefficients) |
| Use Case | CNN input, general audio tasks | Speech tasks, compact features |

## Transformers

### Transformer Network Intuition

**Transformer Architecture Overview**

* The transformer network allows for parallel processing of entire sequences, unlike previous models that processed one token at a time.
* It addresses issues like vanishing gradients found in RNNs, enabling better handling of long-range dependencies.

**Key Innovations**

* The architecture combines attention-based representations with a convolutional neural network (CNN) style of processing.
* Self-attention computes rich representations for each word in a sentence simultaneously, enhancing the model's understanding of context.

**Attention Mechanisms**

* Self-attention generates multiple representations for words in a sentence, improving the richness of the data.
* Multi-headed attention extends this concept, allowing for diverse representations that can be effectively used in various NLP tasks, such as machine translation.

### Self-Attention Mechanism

**Purpose**

Computes attention-based representations for each word in a sentence, allowing the model to understand context.

Self-attention enables a model to **compute context-aware representations** of each word in a sentence by **relating it to every other word**, regardless of their positions. It captures **semantic dependencies** and **long-range interactions**, which are crucial for understanding meaning in language.

**Key Components**

For each input word ​, the model learns three vectors:

* **Query (​)**: Represents the question about the word (encodes the current word’s **semantic role** or "question" it asks about its context).
* **Key ()**: Represents the context of other words (encodes other words’ **contextual roles** or "what they offer" to the query).
* **Value ()**: Represents the information to be aggregated (contains the actual **content/information** that may be transferred to the query word).

These vectors are computed as linear projections of the input embedding:

, ,

where are learnable weight matrices.

**Calculating Attention Representations**

1. **Score Calculation (Inner Product)**: For a word , compute the attention score with other words (a **compatibility score** between each query and every key ​ using a dot product):

This measures **how much attention** word should pay to word ​.

1. **Softmax Normalization**: Convert the scores into **attention weights** that sum to 1 (normalize the scores to get attention weights):

This ensures that the model focuses more on **relevant words** and less on irrelevant ones.

1. **Weighted Sum**: Compute the **output representation** for word by aggregating the values weighted by attention (compute the attention representation for word ):

This new vector ​ is a contextualized embedding of , informed by the entire sentence.

**Contextual Representation**

Self-attention helps the model understand the meaning of each word **based on the words around it**. For example, the word **"l’Afrique"** can mean different things depending on the sentence:

* In *“l’Afrique de l’Ouest”* (West Africa), it refers to a **region**.
* In *“l’Afrique a souffert”* (Africa has suffered), it refers to the **continent as a whole**.

The self-attention mechanism lets the model **adjust the meaning** of “l’Afrique” depending on the **context**, just like humans do when reading.

This is very helpful in long sentences, because it allows the model to connect related words **even if they’re far apart** — something older models like RNNs or CNNs aren't good at.

**Why It Matters**

This elegant mathematical mechanism forms the backbone of Transformer models, such as BERT and GPT. It allows for:

* Parallel computation (unlike RNNs)
* Global dependency modeling
* Rich context understanding

Together, these make self-attention a foundational component of modern NLP.

### Multi-Head Attention

**Overview**

**Multi-head attention** extends the self-attention mechanism by allowing the model to apply attention **multiple times in parallel**, using different **learned projection matrices (**learned weight matrices**)**.  
Each "head" learns to focus on different aspects of the input, enabling the model to capture **richer contextual relationships**.

**Computation Process**

1. **Input Vectors**: For each word in the input sequence, we compute three vectors

* Query: Q
* Key: K
* Value: V

1. **Learned Weight Matrices per Head**: For each attention head *i* , the model learns

, ,

These project the input into different subspaces for computing attention.

1. **Scaled Dot-Product Attention (per head)**: For each head *i*, we compute self-attention using the scaled dot-product formula  
   where is the dimension of the key vectors.

This produces a context-aware output for each head.

1. **Concatenation and Output Projection**: If there are *h* heads, we concatenate the attention outputs from all heads and compute  
     
    is a learned matrix that projects the concatenated vector back into the model’s embedding space (the final weight matrix).

**Final Representation**

The final output of multi-head attention is

This output is passed to subsequent layers (e.g., feedforward layers or residual blocks in the Transformer architecture).

**Why It Matters**

By running multiple attention mechanisms in parallel, **each head learns to focus on different relationships** within the sequence — such as syntactic roles, long-range dependencies, or semantic relevance.

This gives the model a **more nuanced and comprehensive understanding** of the input, which is crucial in tasks like translation, question answering, and summarization.

### Transformer Network

**Paper**: https://arxiv.org/pdf/1706.03762

**Transformer Architecture Overview**

**1. Input Embeddings**

* **Word Embeddings**: Each word in the input sentence is converted into a vector representation using embeddings. Useful are
  + **pre-trained** word embeddings - Word2Vec, GloVe, FastText

or

* + **learned embeddings** - in the context of transformers, embeddings can also be learned directly during the training of the model. Each word in the vocabulary is associated with a randomly initialized vector, which is updated through backpropagation as the model learns.

This captures semantic meaning.

* **Special Tokens**:
  + **Start of Sentence (SOS)**: A token that indicates the beginning of the translation process.
  + **End of Sentence (EOS)**: A token that signifies the end of the translation.

**2. Encoder Block**

* **Multi-Head Attention**:
  + The encoder consists of multiple layers, each containing a multi-head attention mechanism.
  + **Query (Q)**, **Key (K)**, and **Value (V)** matrices are computed from the input embeddings:
    - **Q**: Represents the current word's context.
    - **K**: Represents the context of all words in the input.
    - **V**: Contains the actual values (word embeddings) to be used in the attention calculation.
  + The attention scores are calculated using the dot product of Q and K, followed by a softmax function to obtain weights. These weights are then applied to V to produce the output of the attention layer.
* **Feed-Forward Neural Network**:
  + The output from the multi-head attention is passed through a feed-forward neural network, which consists of two linear transformations with a ReLU activation in between
  + For each token embedding *x*, the FFN applies:

This is applied independently to each position (token) in the sequence

* + This network helps to learn complex features from the attention output.
* **Layer Normalization and Residual Connections**:
  + Each sub-layer (multi-head attention and feed-forward network) is followed by a residual connection (skip connection) and layer normalization. This helps stabilize training and allows gradients to flow better through the network.
  + Reminder: residual connection idea - instead of just passing data through a layer, add the input back to the output. Let’s say you have an input vector *x*, and your model wants to learn a function *H(x)* that transforms it. Without a residual connection, the model must learn *H(x)* from scratch.But with a residual connection, the model instead learns

In other words*: F(x)* is the **residual** — the difference between the desired output *H(x)* and the input *x*. The model’s job is just to learn how to tweak *x*, not replace it entirely.

* **Stacking Encoder Layers**: The encoder block is typically repeated **n times** (commonly six), allowing the model to learn increasingly abstract representations of the input.

**3. Decoder Block**

* **Input to Decoder**:
  + The decoder receives the output from the encoder and starts with the SOS token.
* **Masked Multi-Head Attention**:
  + The first attention layer in the decoder is masked to prevent the model from peeking at future words in the sequence during training. This ensures that predictions are made based only on previously generated words.
* **Multi-Head Attention with Encoder Output**:
  + The second attention layer in the decoder uses the encoder's output as context. It computes attention scores between the current decoder input and the encoder's output to determine which parts of the input are relevant for generating the next word.
* **Feed-Forward Neural Network**:
  + Similar to the encoder, the output from the attention layers is passed through a feed-forward neural network.
* **Layer Normalization and Residual Connections**:
  + Each sub-layer in the decoder also includes residual connections and layer normalization.

**4. Positional Encoding**

* **Importance of Position**

The self-attention mechanism does not inherently consider the order of words. Positional encoding is introduced to provide information about the position of each word in the sequence.

* **Sine and Cosine Functions**:
  + Positional encodings are generated using sine and cosine functions of different frequencies. For a word at position *pos* in a vector of dimension *d*:

For even indices: For odd indices:

* This results in unique positional vectors that are added to the word embeddings, allowing the model to differentiate between words based on their positions.

1. **Output Layer**

After the decoder processes the input and applies self-attention and feed-forward layers, the result is a sequence of **context-rich hidden vectors**, one for each output position.

To convert these vectors into actual **word predictions**, we use two final steps: a **linear transformation** and a **softmax function**.

* **Linear Transformation (Projection to Vocabulary Space)**:

The final output from the decoder is passed through a linear layer that projects the output to the size of the vocabulary.

Each hidden vector from the decoder has a fixed dimension (e.g., 512 or 768), but we need to map it to the size of the vocabulary (e.g., 30,000 tokens).

So we apply a **linear layer**:

* ​: the decoder’s hidden state at position *t*
* : learnable weight matrix
* : bias term
* *V*: vocabulary size
* Result: a vector of **logits** — one unnormalized score for each word in the vocabulary
* **Softmax Layer (Convert to Probabilities)**:

A softmax function is applied to convert the logits into probabilities for each word in the vocabulary, allowing the model to predict the next word in the sequence:

* This gives a **probability distribution** over all words in the vocabulary
* The model can now either:
  + - **Select the most likely word** (argmax)
    - **Sample** from the distribution for diversity

**Why This Matters**

This two-step output layer:

1. **Projects internal representations** into a space the model can "speak" in (i.e., the vocabulary)
2. **Generates interpretable probabilities** for the next word — essential for generating fluent and coherent sequences.

**6. Training with Masked Multi-Head Attention**

During training, the Transformer decoder has access to the entire correct output sequence (also called the **target sequence**). However, we want the model to learn **how to generate the output one word at a time**, just like it will do during inference.

To simulate this behavior, the model uses **masked multi-head self-attention** in the decoder.

**What Does "Masked" Mean?**

Masking prevents the decoder from “cheating” by looking at **future words** in the sequence.

For example, when predicting the 3rd word, it should not be able to see the 4th, 5th, or later words — only the first two.

This is done by applying a **triangular attention mask** that sets attention scores to −∞ (or very negative values) for positions beyond the current token before softmax is applied.

**Why This Matters**

* **Simulates autoregressive prediction** (modeling approach where the model generates each output one step at a time, and each step depends on the outputs it has already generated) during training.
* Helps the model learn to generate text **step by step**, just as it will during real inference (where it won't know the future).
* Prevents **data leakage** from future tokens.

**In Context of Multi-Head Attention**

Each attention head within the masked multi-head attention block uses the same masking mechanism. This ensures **every head** focuses only on the past and current tokens when producing the next word’s representation.

**Conclusion**

The transformer architecture is a powerful model for sequence-to-sequence tasks, such as machine translation. Its ability to process input data in parallel, combined with the attention mechanisms and positional encodings, allows it to capture complex relationships in the data. Understanding these components is crucial for effectively applying transformers in various natural language processing applications.