RNN

Dealing with Sequences

1. **Autoregressive Models:** These models predict the next element in the sequence based on the past values. Each time step uses the previous data points to make a prediction. For example:
   - Prediction x_t based on x_t, x_{t-1}.
   - Common in time series forecasting or language modeling.

2. **Using a Hidden state (RNN-style models):** Rather than directly using the past inputs, these models summarize past information in a hidden state. This hidden state captures the context of all previous inputs up to time t, and it's then used to predict future values:
   - Each new input updates the hidden state.
   - The hidden state at time t is influenced by both current input x_t and the previous hidden state h_{t-1}.
   - This structure is the foundation of Recurrent Neural Networks (RNN).
   - The core idea that is important in RNNs are hidden state which enables the model to remember the past information.

3. **Different formulation of the problem**
   - One to many: the input is the standard format the output is a sequence (Image captioning)
   - Many to one: the input is a sequence the output is a fixed-size vector (Sentimental analysis)
   - Direct many to many: input and output are both sequence (Video captioning)
   - Delayed many to many: input and output are sequence but prediction starts after some delay (this handles cases where understanding depends on future context as well like translation)

Text data and Embedding

1. Why can't ML handle raw input text?
   - ML models require numerical inputs.
   - Text is symbolic and must first be converted into a numerical representation before it can be processed.
2. One-hot Encoding: is a basic method for converting text to vector:
   - Build a vocabulary (all unique words)
   - Assign each word an index
   - Represent each word as a binary vector
   - Limitations:
     - Very high-dimensional (vector size = vocabulary size)
     - Very sparse (only one non-zero value)
     - No semantic similarity: words like "cat" and "dog" are just as distance as "cat" and "table".
3. Word Embeddings (Dense Vectors)
   - They are learned representation of words in lower-dimensional space.
   - Instead of 1s and 0s, words are mapped to dense vectors of real numbers.
   - These embeddings capture semantic relationships, meaning similar words are close vectors in space.
   - These embeddings are pretrained or learned from the data, unlike fixed nature of one-hot vectors.
   - Good embeddings should map semantically close words to geometrically close vectors.

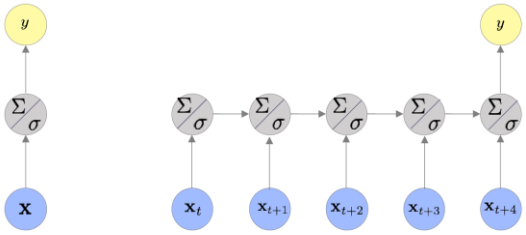
4. One-hot encoding Vs. Word embedding
    - One-hot encoding:
        - Sparse: mostly zeros
        - High-dimensional: As large as vocabulary
        - Rigid: No learning involved, based on arbitrary indexing
    - Word embedding:
        - Dense: most values are non-zero
        - Lower dimensional: Typically size of 50-300 dimensions
        - Learned from data: Reflect semantic similarities

Dealing with Sequences -issue

1. Variable-length sequences:
    - Real-world sequences (sentences, sensor logs) often vary in length
    - Solutions:
        - Padding (with zeros or special tokens)
        - Truncation or splitting
        - Sequence-aware models like RNN can handle this better than standard dense networks.
2. Short-term vs. long-term dependencies
    - Some tasks (like recognizing "walking") only needs recent inputs.
    - Other like making a sandwich requires long context.
    - RNN-based models must capture dependencies over varied time scales, which is non-trivial.
3. Order matters:
    - Unlike sets, the order of elements in a sequence is critical ("dog bites man" ≠ "man bites dog")
    - Sequential models must retain this temporal or positional information.

Modeling Sequences

1. Vanilla (feedforward) Network (left side)
    - A standard one-to-one network processes a single input vector x and gives a single output y.
    - It has no memory of the past, each prediction is made in isolation.



2. Sequential Modeling via Recurrence (right side)
    - Here, the architecture is extended across time using the same hidden unit repeated.
    - At each time step $t, t+1, \ldots, t+4$, the model receives a new input and updates its state.
    - Recurrence is introduced via connections between the hidden layers across time.

Recurrent Neural Networks

1. Recurrence = Memory:
    - Past states influence the current computation.
    - The output at time t is informed y past input via the hidden state.
2. Casual Dependency Modeling:
    - The decision made at time t depends on the results at time $t-1$.
    - This mimics real-world scenarios: what happens now depends on what happened before.

3. Dual input source:
   - Each RNN cell considers both:
     - The current input x_t
     - And the previous state h_{t-1}
   - In other words, RNN incorporates both present and past information when reasoning over a sequence.

Mathematical Formulation of RNN

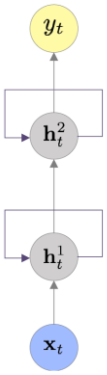
$$h_t = \sigma(W_h h_{t-1} + W_x x_t), \quad y_t = \sigma(W_y h_t)$$

1. Key Symbols:
   - $x_t \in R^m$: input at time t
   - $h_t \in R^p$: hidden state at time t
   - $y_t \in R$: output
   - $W_h \in R^{p \times p}$: weights for past hidden state
   - $W_x \in R^{p \times m}$: weights for current input
   - $W_y \in R^p$: weights for final prediction
2. RNN Equations:
   - The first equation shows the hidden state update:
     - It combines the influence of the past and current input using a nonlinearity σ (tanh)
     - The hidden state size p is a hyperparameter – set manually.
   - The output computation equation (second one):
     - Used in many-to-one setups when you need a final prediction.

3. Important design Detail: weights are shared across time
   - This parameter sharing makes training efficient and enables generalization across sequences of varying lengths.
   - There are no W_x^t or W_h^t only W_x and W_h

RNNs with Multiple Hidden Layers (Deep RNNs)

1. Standard RNN vs Deep RNN:
   - A simple RNN has a single hidden layer per time stamp
   - A deep RNN stacks multiple hidden layers vertically (not just across time)
   - This enables model to capture more complex patterns, similar to how deep feed forward networks capture hierarchical features.
   - Layering lets each level learn a different abstraction the bottom might capture short-term patterns and higher levels capture long-term patterns or abstract meanings.



2. Equations:
   - First layer update:

$$\boldsymbol{h_t^1} = \sigma(W_h^1 \boldsymbol{h_{t-1}^1} + W_x \boldsymbol{x_t})$$

   - Second layer update:

$$\boldsymbol{h_t^2} = \sigma(W_h^2 \boldsymbol{h_{t-1}^2} + W_{12} \boldsymbol{h_t^1})$$

Backpropagation Through Time

1. Unfolded RNN and Loss Definition
   - RNN is unrolled across time steps $(t, t+1, t+2)$.
   - The loss is defined over the output at the final time step.

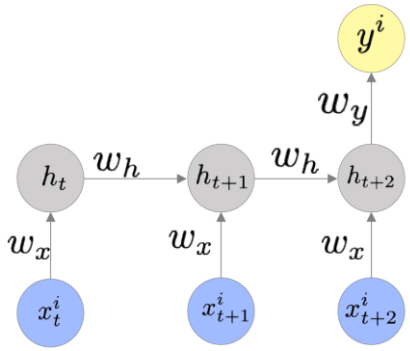
$$\hat{y^i} = f_\sigma(w_y h_{t+2})$$

$$\mathcal{J}^i(w) = \left(y^i - \hat{y^i}\right)^2$$

   - Hidden state is recursively computed using:

$$h_t = f_\sigma(w_h h_{t-1} + w_x x_t^i)$$

   - Final output is computed using h_{t+2}



2. Backpropagation (core idea)
   - Compute gradient of loss with respect to parameters like w_x via chain rule.
   - This involves how each weight influenced the output through all time steps.

3. Backpropagation equations: they capture how past time steps influence the current prediction, and how parameter updates are accumulated over time.

$$\frac{\partial \mathcal{J}^i(w)}{\partial w_x} = \frac{\partial \mathcal{J}^i}{\partial \hat{y^i}} \cdot \frac{\partial \hat{y^i}}{\partial h_{t+2}} \cdot \sum_{k=t}^{t+2} \frac{\partial h_{t+2}}{\partial h_k} \cdot \frac{\partial h_k}{\partial w_x}$$

4. Computational challenges
   - Gradient accumulation formula:

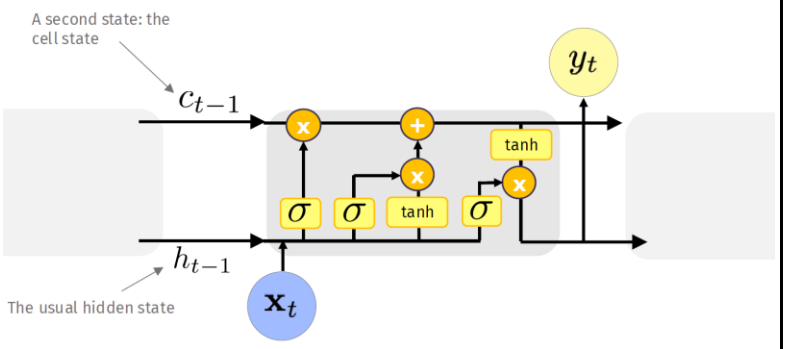
$$\frac{\partial h_{t+2}}{\partial h_k} = \prod_{j=k+1}^{t+2} \frac{\partial h_j}{\partial h_{j-1}}$$

   - This formula highlights the recursive dependency chain, where gradient at each level must be propagated backwards through time using the chain rule.

5. Challenges: Training RNNs involves computing gradients over time via repeated multiplication (chain rule).
   - Long dependency chains:
     - Vanishing gradients (products of small values : near zero)
     - This causes earlier layers (or time steps) to have negligible influence.
     - Solution: Gated Cells that have control mechanism that let the network retain important long-term signals and discard irrelevant information.
     - Exploiting gradients (products of large values: very large)
     - Solution: Detect when a gradient exceeds a threshold and rescale it:
   - Scalability:
     - Multiple hidden layers add complexity
     - Sequences of varying length complicate gradient tracking.
   - Sequence output:
     - What if we need to predict outputs at each time step?

Long-short Term Memory

1. Gated Cells (LSTM building blocks): the gating mechanism regulates memory and allow the model what to forget and what to remember, solving gradient degradation issues.
   - Main idea: LSTM cells use gates to control the flow of information over time. Each gate is a learned filter that decides:
     - What to keep
     - What to forget
     - What to output



   - The diagram shows an LSTM cell as a composite of several operations involving:
   - Sigmoid activations (σ): acts as a gate values between 0 and 1.
   - Tanh activations: transform candidate content.
   - Element-wise operations: multiplication (x), addition(+).
   - Horizontal line: cell state, flows with minor changes (key to long term memory)
   - Vertical lines: hidden state flow, interact with current input and output.
2. High level structure: At each level t the LSTM receives:
   - The current input vector X_t
   - The previous hidden state h_{t-1}
   - The previous cell state c_{t-1}
3. The output will be:
   - A new hidden state h_t (used for prediction)
   - A new cell state c_t (the internal memory)

LSTM phases

1. Forget phase: LSTM decides what information discard from the cell state.
   - The forget gate determines how much of the previous cell state c_{t-1} to keep:

$$f_t = \sigma\big(W_f \cdot [h_{t-1}, x_t] + b_f\big)$$

   - Then, the previous cell state is updated elementwise:

$$\widetilde{c_t^{(\text{forget})}} = f_t \odot c_{t-1}$$

   - This is a crucial phase for long-term memory control: it can erase irrelevant memory.
2. Input phase: LSTM decides what new information to store
   - This phase determines what new information from X_t and h_{t-1} should be added to the cell state. It uses two gates :
     - Input gate: $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$
     - Candidate values (new content to store): $\widetilde{c_t} = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$
     - The actual new content to add:

$$\widetilde{c_t^{(\text{add})}} = i_t \odot \widetilde{c_t}$$

3. Cell state update: combine forget and input
   - New merge the two branches:

$$c_t = f_t \odot c_{t-1} + i_t \odot \widetilde{c_t}$$

- This is the core memory update step, selectively forgetting old content and inserting new.
- Cell state is preserved across time, with controlled changes.

4. Output phase: what to emit as hidden state
   - The output gate decides what part of cell state should affect the output hidden state:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

   - Then, the hidden state will be:

$$h_t = o_t \odot \tanh(c_t)$$

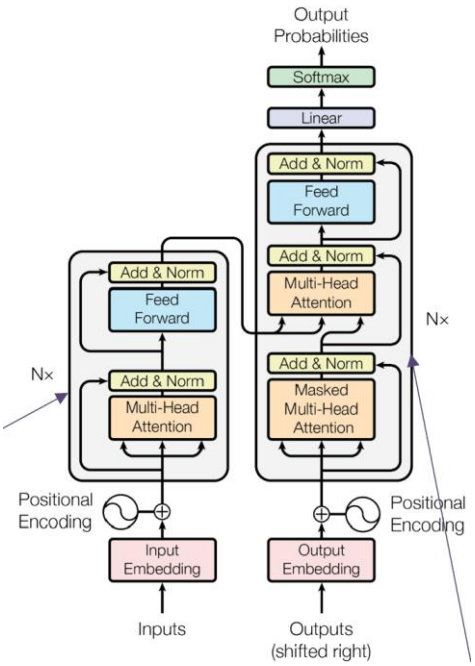
   - This h_t can be used for output prediction y_t or passed on to the next time step.

Word position and positional encoding

1. Why positional encoding matters:
   - Transformers lack any sequence ordering, unlike RNNs. So we need to guide the model to understand the word order.
2. How positional encoding is done: each world gets two components
   - Word embedding: encode semantic meaning
   - Positional encoding: encode each word's position in the sequence.
   - These two vectors are added together.
3. Types of positional encodings:
   - Fixed (sinusoidal): encodes positions using sine and cosine functions.
   - Learned: learnable vectors optimized during the training.

Transformer's Main Structure

1. The transformers consist of two main parts:
   - Encoder: processes the input sequence.
   - Decoder: generates the output sequence (used in the task like summarization or translation)
2. Encoder Block: each of the N encoder layers have:
   - Multi-head self-attention
   - Feed-forward neural networks.
   - Residual connection and layer normalization after each sub-layer
3. Decoder Block: each of the N decoder layers have:
   - Masked multi-head self-attention (to avoid using feature tokens).
   - Encoder-decoder attention (queries from decoder, keys and values from encoder).
   - Fead-forward neural network.
   - Residual and normalization layers throughout.



Self-attention: Core mechanism

Purpose: helps the model to focus on relevant words across the sentence while processing each word.

1. Analogy
   - Query(Q): What we're looking for
   - Key(K): what we have
   - Value(V): What we give if the key matches the query.
2. Steps:
   - Projects input into Q, K, V using learnable weight matrices (linear layers)
   - Compute dot product $Q \times K^T$
   - Scale the scores by $\sqrt{d}$ (dimension of embeddings).
   - Apply softmax to get attention weights.
   - Multiply these weights with V: $\text{softmax}(QK^T/\sqrt{d}) \times V$

Masked Self-Attention (in Decoder)

Why needed: At training time, we have full sequences. But in the inference, we generate output one token at a time. So mast the future:

- For position I, Allow only positions $\leq i$
- Implemented using a look-ahead mask
- Ensures the model doesn't cheat by using future tokens.

Multi-Head Self Attention

A single attention head might capture only one type of relation so we use multi-head.

- Each head has its own, Q, K, V matrices.
- They run in parallel to capture different relationships.
- Outputs of all heads are concatenated and linearly projected.

Benefits:

- Provide diverse representations.
- Enhance model's ability to capture complex features.

Residual Connections and Layer Normalization

Each sublayer (attention or feed-forward) is followed by:

- Residual connection: Adds input back to output
- LayerNorm: Stabilize training
- This allows gradient to flow easily and stabilizes deep networks

Feed-Forward Networks

Each encoder-decoder block contains a simple 2-layer fully connected Networks:

- Applies the same transformation to each position independently.
- Often includes ReLU or GeLU activation.

Encoder-Decoder Attention

In decoder, after mask self-attention:

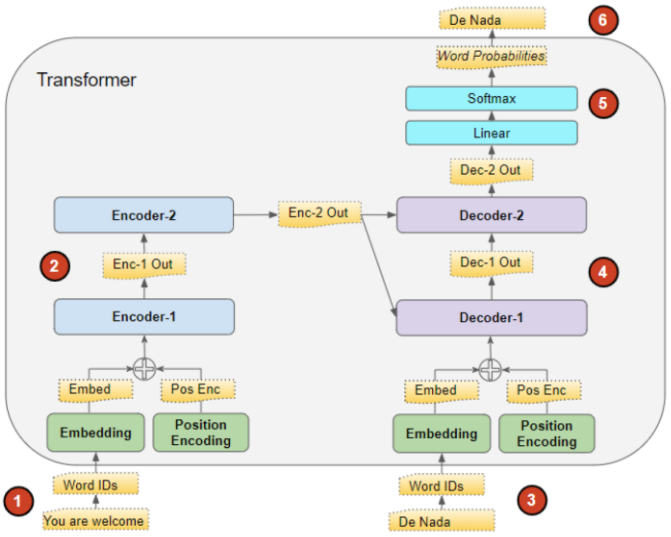
- A second attention connects decoder outputs to encoder outputs.
- Q from decoder, K and V from encoder.
- Enables decoder to focus on relevant parts of the input sequence during generation.

Inference Process

During Inference

- The model generates one token at a time
- Uses previously generated tokens as input for the next step
- Starts with special $<$ start $>$ token and ends with an $<$ *end* $>$ token

Training a Transformer



1. Input Preparation: Source Sentence Tokenization and Embedding
   - Input You are welcome
   - The source sentence is tokenized into Word IDs (numerical indices from a vocabulary)
   - Each token goes through two components
     - Embedding layer: Converts token IDs into dense vectors capturing semantic information.
     - Positional Encoding: Injects information about word order (import because attention alone is order-agnostic)
   - These two vectors are added together and sent to encoder stack.
2. Encoder Stack: Contextual Representation learning
   - This stage consists of multiple identical encoder layers (in this case encoder-1 and encoder-2)
   - Each encoder block includes:
     - Multi-head Self-attention: Allows the model to consider relationships between all the words in the input sentences.
     - Feed-forward network: Adds non-linearity and richness.
     - Residual Connections + LayerNorm: Helps with stability and gradient flow
   - Output: A set of context-rich embeddings (Enc-2 Out) one for each input token.
3. Decoder Input Initialization: Starting the Generation
   - The decoder starts with a special token: $< START >$
     - Embedded into a vector.
     - Combined with positional encoding.
   - This forms the initial decoder input
4. Decoder Stack: Generating the next word
   - Decoder-1 and Decoder-2 are transformer decoder blocks.
   - Each decoder block includes:
     - Masked self-attention: Only allows attending to previous tokens, not future ones (no cheating)
     - Encoder-decoder attention: Queries from decoder attend to encoder outputs (context from input sentence)
     - Feed-forward Network
     - Residual and LayerNorm
5. Linear + Softmax: predict the next word
   - The output from last decoder block (Dec-2 out) goes through:
     - Linear layer: Projects embedding to vocabulary size.
     - Softmax: Converts logits to probabilities over vocabulary.
   - The highest probability word is selected.

6. Auto Regressive Loop: Feeding prediction Back
    - The predicted word “De” is appended to the sequence: “[<START>],[De]”
    - This sequence is reprocessed in the decoder, and the process repeats until <END> token is generated

Understanding the Attention Mechanism (Google)

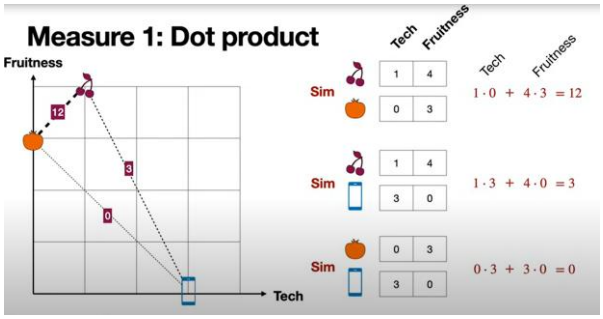
- Context: The attention mechanism is a foundational concept powering transformers and large language models (LLMs) like ChatGPT.
- Purpose: To explain how attention improves traditional sequence-to-sequence models used in tasks like translation

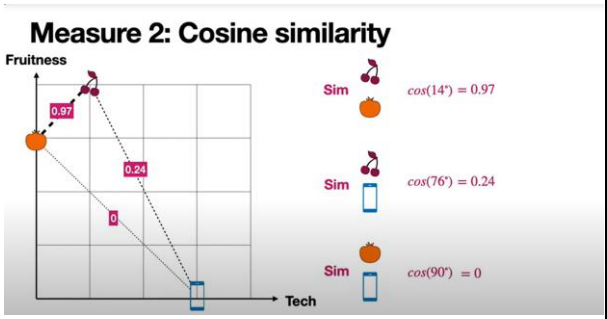
1. Why attention is needed in sequence models: Traditional encoder-decoder limits
    - Processes word by word, passing only the final hidden state to the decoder.
    - Problem: some languages have different word order, making alignment tricky.
    - Example: “Black cat ate the mouse” -> First French word is “chat” (cat), not “black”.
2. What is attention mechanism?
    - Definition: a method that allows a model dynamically focus on most relevant parts of the input sequence while generating output.
    - Mechanism: Assigns weights (attention scores) to all input tokens higher for relevant ones, lower for others.
3. How attention works in encoder-decoder model
    - Key differences from traditional models:
        - All hidden states from the encoder are passed to the encoder (not just the last one).
        - Decoder performs an additional attention step at each time step.
    - Attention process step-by-step:
        - Input: all encoder hidden states, each linked to a specific word.
        - Scoring: Each hidden state is assigned a score (importance).
        - Weighting: each state is multiplied by a softmax score: relevant states are amplified.
        - Context vector: A weighting sum creates a context vector.
        - Concatenation: Combine current decoder state with context vector.
        - Prediction: Feed the combined vector into a feed-forward neural network to predict the next word.
        - Repeat until an <end> token is generated.

The math behind Attention: Keys, Queries, and Values matrices

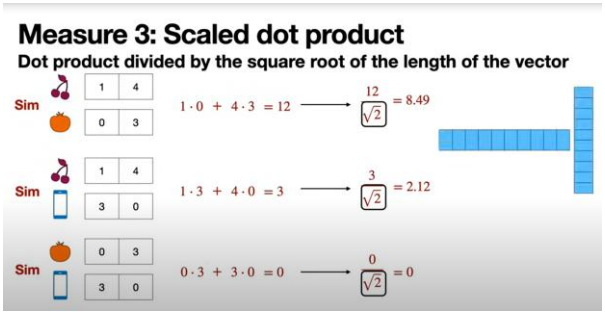
Core concept of Attention mechanism

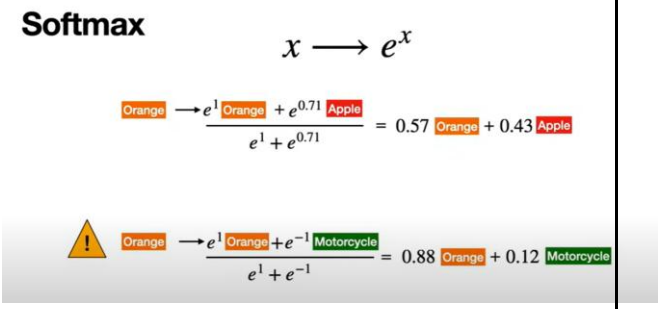
1. Word embedding and Contextual Meaning
   - Embedding puts words in a high-dimensional space where semantic similarity = proximity.
   - Example: Word "Apple" is ambiguous (fruit or brand) its meaning is derived from its neighboring words ("orange", "phone")
2. Similarity Between Words
   - Similarity is used to move embedding closet together when relevant.
   - Three types of similarity:
     - Dot product: measures alignment of vectors
     - Cosine Similarity: Measures angles between vectors
     - Scaled dot product: Dot product divided by $\sqrt{d}$ (dimension size) to avoid overly large values in high dimensions.





3. Softmax for Normalization
   - Converts similarity scores into positive weights that sum to 1.
   - Prevents exploding values and handle negatives.
   - This process creates a weighted average of all other word vectors.



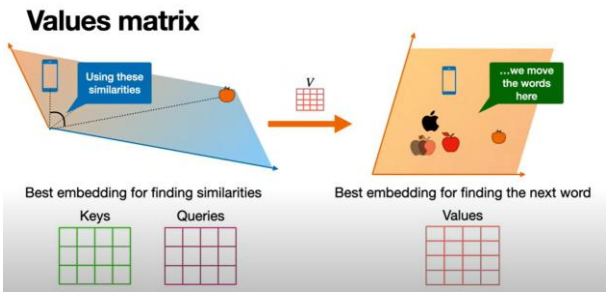


Key mechanism: how attention works

1. Input embeddings are given.
2. Similarity matrix is computed using cosine or scaled dot product.
3. Apply softmax to get normalized attention weights.
4. Use these weights to blend word vectors each word is replaced with a contextualized version, a weighted sum of all words.
5. This process is repeated over multiple layers.

Key, Query, and Value (KQV) Matrices

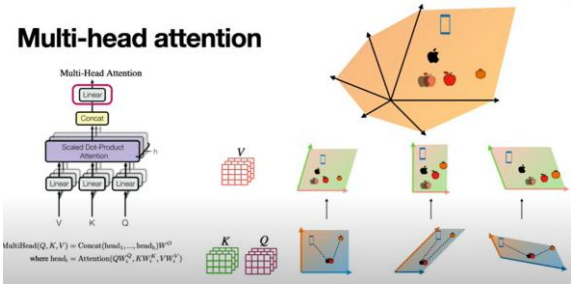
1. Queries and Keys:
   - Query is a vector that represents a specific token from input sequence.
   - Key is a vector that corresponds to the input sequence.
   - Value is a vector with key that constructs the output from attention score.
   - When a query and key match well, their attention score is high, the corresponding value is emphasized in the output
   - Used to compute similarity between words
   - Embeddings are transformed into a new space via linear transformations.
   - These helps distinguish ambiguous words better by manipulating how closeness is measured.



2. Values:
   - While Keys and Queries determine how much attention each word gets.
   - Values determine what is being passed to the next layer.
   - They are also a linear transformation of the original embedding but focus on the output content.

Multi-Head Attention

- Instead of one attention calculation, multiple sets of KQV matrices are used ("Heads")
- Each head learns to focus on different aspect (syntax, semantic)
- Outputs are concatenated and passed through another linear transformation.
- A final linear layer compresses high-dimensional info to manageable size.

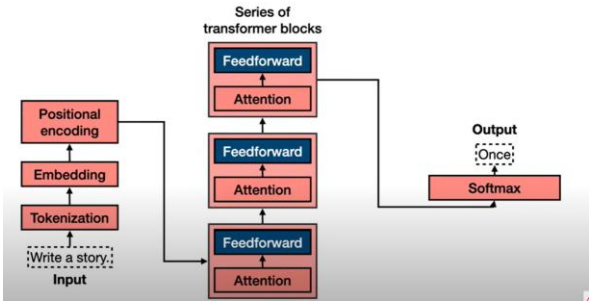


Training and Optimization

- All matrices (K, Q, V) are learned during training.
- The figures out which heads matter and how to weigh them through backpropagation.

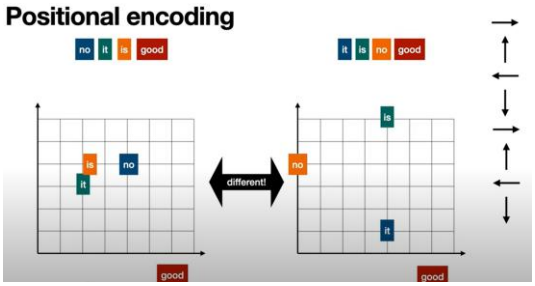
What are Transformer Models and how do they work?

1. Preprocessing Pipeline
   - Tokenization:
     - Splits sentences into tokens (often words or subwords).
     - Handles punctuation and common suffixes/prefixes.
   - Example: “Doesn’t” → “Does” + “n’t”



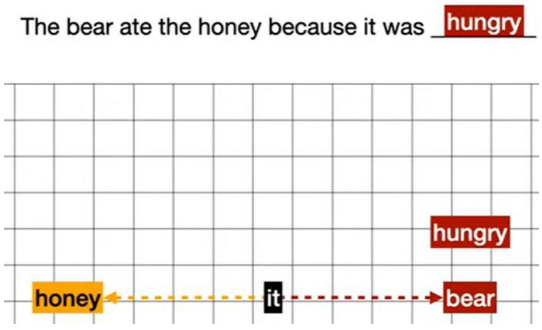
2. Embeddings
   - Converts tokens into high-dimensional vectors.
   - Embeddings are semantic representation meaning similar words are closer in vector space.
   - Can be generated though models like word2vec or trained directly via neural network.

3. Positional Encoding
   - Neural networks alone don’t understand word order. Traditional sequential models like RNNs process words one after another, naturally understanding order. But Transformers process all words simultaneously through self-attention. Imagine shuffling the words in "The cat sat on the mat" - without positional information, the model might see the same thing as "mat the on cat The sat". That's problematic!

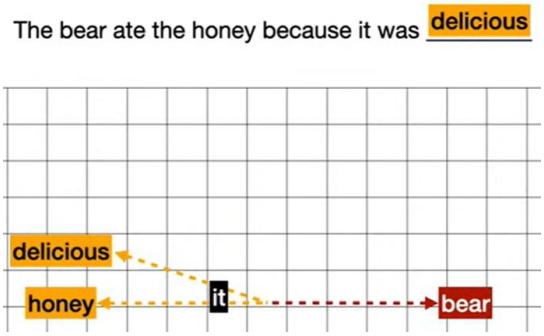


   - Solution: Add position vectors to embeddings to encode order. it encodes positions using sine and cosine functions of different frequencies. Think of it like giving each position a unique mathematical "fingerprint" that the model can recognize.
     Each position gets a completely unique encoding vector. No two positions will ever have identical encodings, no matter how long your sequence is. The sine and cosine waves at different frequencies create patterns that never repeat exactly.

4. Attention Mechanism:
   - Context-aware modification of embeddings using gravitational pull from other relevant words.
   - Examples:
     - “Apple” in “buy an apple and orange” → pulls toward fruit.
     - “Apple” in “Apple released a new phone” → pulls toward tech.
   - Multiple words (like 'orange' or 'phone') influence the current word depending on context.



- Sentence: “The bear ate the honey because it was ___.”
  - “hungry” → “it” refers to bear.
  - “delicious” → “it” refers to honey.
- Attention helps shift the meaning of "it" depending on surrounding words.



What is self-Attention?

Self-attention is a mechanism for capturing the relation between words/tokens of an input. Not just considering each word in isolation but understanding how each word relates to others in a sentence. This is critical in tasks like translation, text classification, and language modeling.

Step by step breakdown

1. Word embedding as input: each word in a sentence is first embedded into a numerical vector:
   let X ∈ $R^{T\times N}$, where:
   - T: number of tokens (words)
   - N: size of word embedding
   - Think of each row of T as the embedding of a word

2. Linear Projections: Create Q, K, V: we compute three projections from X
   - Query (Q): what we’re looking for
   - Key (K): what we have
   - Value (V): what we’ll return if there’s match
   - There are linear projections using weight matrices
   $$Q = XW_Q, K = XW_K, V = XW_V$$
   - $W_Q, W_K, W_V \in R^{N\times D}$
   - $Q, K, V \in R^{T\times D}$

3. Computing Attention Score
   - We compute how well each query matches each key using dot products.
   $$\text{Scores} = QK^T \in R^{T\times T}$$
   - This gives a matrix of attention scores between all word pairs.
   - Each row i contain the relevance of all tokens j to token i.

4. Scaling and Softmax: To stabilize gradient we scale scores by $\sqrt{D}$:
   $$\widetilde{W} = \text{softmax}\left(\frac{QK^T}{\sqrt{D}}\right)$$
   - This softmax normalize raw scores into normalized attention weights

5. Weight sum over values: we use the attention weights to combine values:

$$Attention\ outputs = \widetilde{W}V$$

- Shape: $R^{T\times D}$
- Each row represents the new representation of a word, influenced by the context

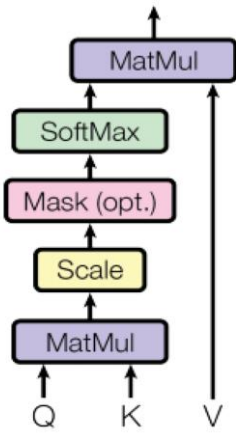
A Web Search analogy

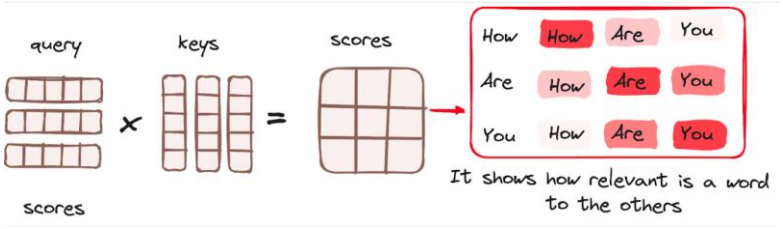
- Query(Q): is the search text you type in the search engine bar. This is the token for which you want to find more information.
- Key(K): is the title of each webpage in the search result window. It represents the possible tokens the query can attend to.
- Value(V): is the actual content of web pages shown.
- Once we match the appropriate search term(Query) with the relevant results(key), we want to get the content (Value) of the most relevant pages.

Multi-Head Attention (Nothing but a scaled dot product)

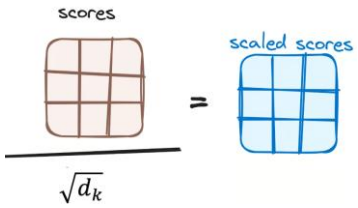
Instead of just one Q-K-V triplet, we compute multiple sets in parallel:

1. Matrix Multiplication (MatMul) – Dot product of Keys and Queries
   - Once the query, key, and value vectors are passed through a linear layer, a dot product matrix multiplication is performed between queries and keys, resulting in creating of score matrix.
   - The score matrix stablished the degree of emphasis of each word should place on other words. Therefore, each word is assigned a score in relation to other words within the same time step. A higher score means greater focus.
   - This process effectively maps the Queries to their corresponding keys.

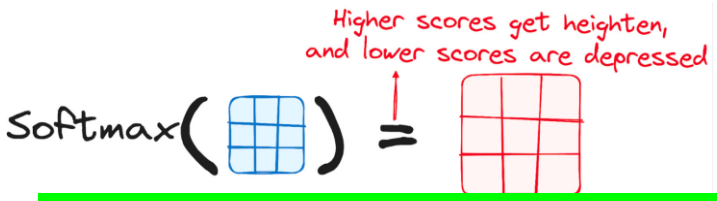




2. Reducing the magnitude of attention scores:
   - The scores are then scaled down by dividing them by the square root of the dimension of the query and key vectors.
   - This step is implemented to ensure more stable gradients, as the multiplication of values can lead to excessively large effects.



3. Applying a softmax to the adjusted scores:



- A softmax function is applied to the adjusted scores to obtain the attention weights.
- This results in probability values ranging from 0 to 1.
- The softmax function emphasizes higher scores while diminishing lower scores, thereby enhancing model's ability to effectively determining which word should receive more attention.

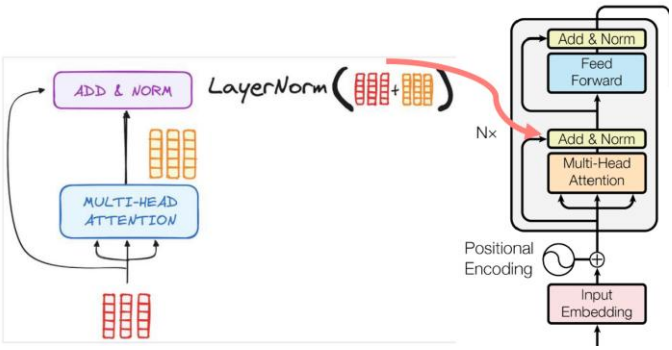
4. Combining softmax results with the value vector:
   - Weights derived from softmax function, are multiplied by the value vector resulting in output vector.
   - In this process, only the words that that present high softmax scores are preserved.

Why it is called multi-head attention?

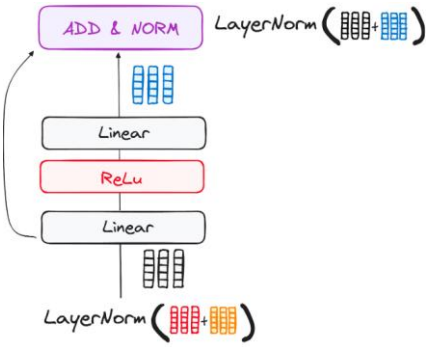
- Before all the process starts, we break our key, values, and queries h times. This process, known as self-attention, happens separately in each of these smaller stages or heads.
- This ensemble passes through a final linear layer, much like a filter that fine-tuned their collective performance. The beauty here lies in the diversity of learning across each head, enriching the encoder model with a robust and multifaced understanding.

Going back to the overall architecture

1. Normalization and Residual Connections:
   - Each sub-layer in an encoder layer is followed by a normalization step.
   - Each sub-layer output is added to its input (residual connection) to help mitigate the vanishing gradient problem, allowing a deeper model.
   - This process is repeated after the Feed-forward Neural Network too.



2. Feed-Forward Neural Network:
    - Picture this network as a due of linear layers, with a ReLU activation nested between them, acting as a bridge.
    - Once the output is processed, it loops back and merges with the input of the pointwise feed-forward network.
    - This merge is followed by another round of normalization, ensuing everything is well-adjusted and sync for the next steps.

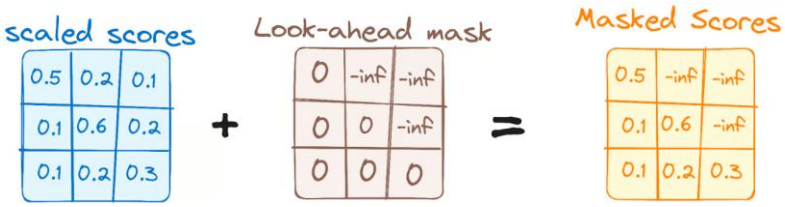


3. The output of the Encoder:
    - The output of the final encoder layer is a set of vectors, each representing sequence with a rich contextual understanding.
    - This output is then used for the input of the decoder in a Transformer model.
    - This encoding guides the model to pay attention to the right words in the input when it's time to decode.
    - Think of building a tower where you can stack N layers of encoder. Each layer of this stack gets a chance to explore and learn different facets of attention, much like layers of knowledge.

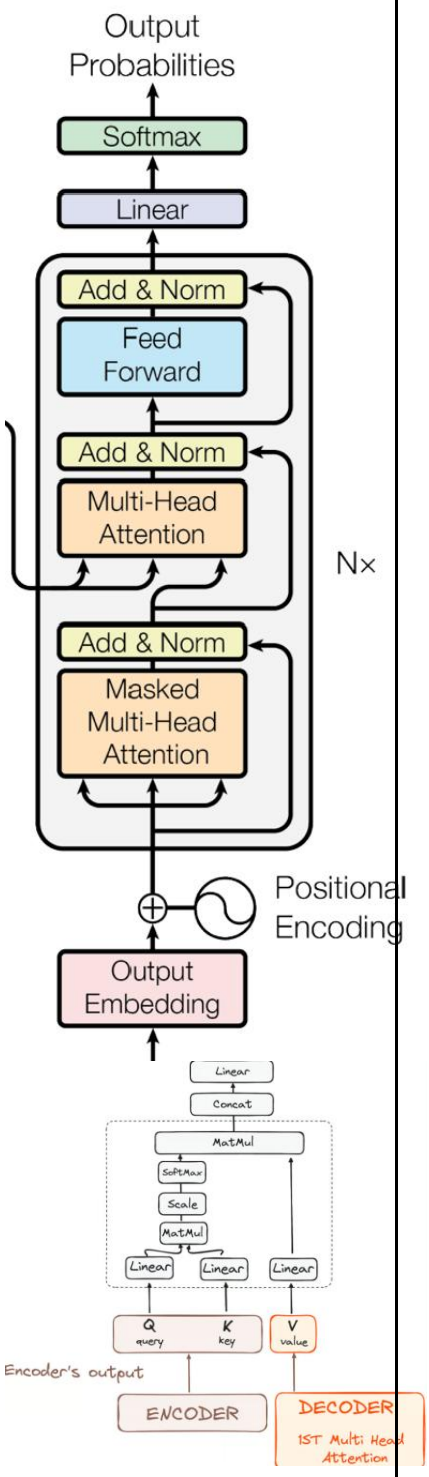
The Decoder Work Flow

- The decoder's role centers on crafting text sequences.
- It's equipped with similar set of sub-layers, two multi-head attention layers, a pointwise feed-forward layer, and incorporates both residual connections and layer normalization after each sub-layer.

1. Output embeddings: At the starting line, the process mirrors that of the encoder. Here, the input first passes through an embedding layer.
2. Positional Encoding: just like encoder followed by embedding, the input passed through positional encoding layer.
3. Stack of decoder layer: the decoder consists of a stack of identical layers each layer has a three main sub-components:
    - Masked Self-Attention Mechanism: this is similar to the self-attention in the encoder but with a crucial difference: it prevents positions from attending to subsequent positions, which means each word in the sequence isn't influenced by future tokens.
    - For instance when the attention scores for the word "are" are being computed, it's important that "are" doesn't get a peek at "you", which is a subsequent word in the sequence.



- The masking ensures that the prediction for a particular position can only depend on known outputs at positions before it. (model can't cheat)
- Encoder decoder multi-head attention (cross attention):
    - In the second multi-head attention of the decoder, the outputs from the encoder take on the role of both queries and keys, while the outputs from the first multi-head attention layer of decoder serve as value
    - This setup effectively aligns encoder's input with decoder's, empowering decoder to identify and emphasizes the most relevant parts of the encoder's input.
    - In this sub-layer, the queries come from the previous decoder layer, and the keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence, effectively integrating information from the encoder with the information in the decoder.
- Feed-Forward Neural Network: Similar to the encoder, each decoder layer includes a fully connected feed-forward network, applied to each position separately and identically.



4. Linear Classifier and Softmax for Generating Output Probabilities
    - The size of this classifier corresponds to the total number of classes involved (number of words contained in the vocabulary). For instance, in a scenario with 1000 distinct classes representing 1000 different words, the classifier's output will be an array with 1000 elements.
    - This output is then introduced to a softmax layer, which transforms it into a range of probability scores, each lying between 0 and 1. The highest of these probability scores is key,its corresponding index directly points to the word that the model predicts as the next in the sequence.