ISTO COMEÇA NA PARTE 2, NÃO HÁ PARTE 1

Part 2

1 - Word2Vec Architecture and Calculation

a)

1. **Input Layer:** One-hot encoded vectors for the words "AI" (x_{AI}) and "very" (x_{very}).
2. **Projection (Hidden) Layer:** Projection layer where the input vectors are mapped to the embedding dimension $N = 2$ using W_{in} and aggregated (I will assume they are averaged, but it might be different, sei lá pá, às vezes não são averaged).
3. **Output Layer:** The hidden vector is multiplied by W_{out} to get the logits, followed by a Softmax activation to predict the probability of the target "is".

$$\{x_{AI}, x_{very}\} \xrightarrow{W_{in}} \{v_{AI}, v_{very}\} \xrightarrow{Average} h \xrightarrow{W_{out}} z \xrightarrow{Softmax} \hat{y}$$

b)

In the CBOW model, the hidden layer vector h is typically the **average** (or sometimes sum) of the input context embeddings.

1. Get Embeddings: Look up the rows in W_{in} corresponding to indices 0 ("AI") and 2 ("very").
   - $v_{AI} = [0.5, 0.2]$
   - $v_{very} = [0.0, -0.5]$
2. Aggregate:

$$h = \frac{[0.5, -0.3]}{2} = [0.25, -0.15]$$

c)

The logits (raw scores) are calculated by multiplying the hidden vector h by the output matrix W_{out}.

$$z = [0.25, -0.15] \cdot \begin{bmatrix} 1.0 & 0.0 & -0.5 & 0.5 \\ 0.0 & 1.0 & 0.5 & -1.0 \end{bmatrix}$$

$$z = [0.25, -0.15, -0.20, 0.275]$$

d)

The Softmax scores are given by

$$P(y_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

We need the probability for the target "is" (Index 1), so we calculate the exponential for z_1 and the sum of all exponentials.

1. **Calculate Exponentials (e^z):**
   - $e^{0.25} \approx 1.2840$
   - $e^{-0.15} \approx 0.8607$ (IS!!!!!!)
   - $e^{-0.20} \approx 0.8187$
   - $e^{0.275} \approx 1.3165$
2. Sum of Exponentials:

$$\sum = 1.2840 + 0.8607 + 0.8187 + 1.3165 = 4.2799$$

3. Calculate Probability:

$$P(\text{“is”}) = \frac{0.8607}{4.2799} \approx 0.2011$$

2 - Static vs. Dynamic Embeddings
The fundamental limitation of static embeddings is that they assign a single vector representation to each word, regardless of the context.

- Polysemy Problem: Can't handle polysemy (words with multiple meanings). The resulting vector is just a weighted average of every meaning.
- Dynamic Solution: These are contextualized embeddings, they are generated on the fly. The representation of a word is a function of the entire sentence. These models can capture the specific meaning intended in that instance.
  - Example with "bank": "I deposited money at the bank" vs "I am standing on the river bank".

3 - FastText vs Word2Vec
In these languages, words have a lot of variations due to prefixes, suffixes and conjugations (e.g., programar, programamos, programando).

- Word2Vec would treat each word as a distinct, atomic unit. It cannot see that "programar" and "programamos" share the same root.
- FastText represents words as bag of character n-grams (subwords). For example, "programar" might be represented by the sum of vectors for "<pr", "ogr", "ama", "r>". This makes it so information is shared across words with similar roots, learning better representations even for rare word forms.
  FastText can generate embeddings for words it never say (OOV) as long as it's made of characters seen in the training data.
- When FastText encounters an OOV word, it breaks it down into its character n-grams, retreives the vectors for the n-grams and sums or averages them to make a new word.
- Word2Vec usually just assigns a generic <UNK> token or a random vector to OOV words.

4 - Manual Seq2Seq Calculation (The "Reversal" Task)
a)

Add the input vector to the previous state ($h_t = h_{t-1} + x_t$), starting with $h_0 = [0, 0]$.

- Step 1 (Input "A"):
  $x_1 = [1, 0]$
  $h_1 = [0, 0] + [1, 0] = [1, 0]$
- Step 2 (Input "B"):
  $x_2 = [0, 1]$
  $h_2 = [1, 0] + [0, 1] = [1, 1]$
- Final Context:
  $c = h_2 = [1, 1]$

b)

- **Input:** $< BOS >= [0, 0]$.
- Decoder State s_1:
  $s_1 = s_0 + e_1 = c + [0, 0]$
  $s_1 = [1, 1]$
- Logits $y^{(1)} = W_o \cdot s_1$:
  Using W_o (rows 2, 3, and 4 correspond to A, B, C):
  - **A:** $[1, 0] \cdot [1, 1] = 1$
  - **B:** $[0, 1] \cdot [1, 1] = 1$
  - **C:** $[0.5, 0.5] \cdot [1, 1] = 1$
- **Prediction:** Tie between A, B, and C (all logits = 1).
- The target is "B". It failed to CONFIDENTLY predict the correct target "B".

c)

- Input e_2: With Teacher Forcing, we use the embedding of the true previous word ("B"), not the predicted one.
  $e_2 = [0, 1]$.
- Decoder State s_2:
  $s_2 = s_1 + e_2 = [1, 1] + [0, 1] = [1, 2]$
- **Logits** $y^{(2)} = W_o \cdot s_2$**:**
  - **A:** $[1, 0] \cdot [1, 2] = 1$
  - **B:** $[0, 1] \cdot [1, 2] = 2$
  - **C:** $[0.5, 0.5] \cdot [1, 2] = 1.5$
- **Result:** The highest logit is **2** (Word "B").

5 - Exposure Bias and Sampling

a)

This is a problem that happens when, during training, the model is trained using teacher forcing, where it is fed the ground truth as input for the current step, even if its own prediction was wrong. The model is never **-> EXPOSED <-** to its own mistakes, creating **-> BIAS <-** during inference, leading to compounding errors.

b)
Scheduled sampling is a special move that mitigates the exposure bias problem by gradually transitioning the model from teacher forcing to using its own predictions during training.
The probability ϵ_t is the probability of using the ground truth token as input at the current time step, at the start of training $\epsilon_t = 1$, but it decays as training progresses. By increasingly feeding the model its own predictions during training, it learns to recover from its errors, making it more robust.

6 - Attention Mechanism (Bahdanau)

a)
The dynamic context vector c_t is the weighted sum of the encoder hidden states:

$$c_t = \sum_{i=1}^{T_x} \alpha_{t,i} h_i$$

where c_t is the context vector at time step t, h_i is the encoder's hidden state for the i-th word in the source sequence (contains information about that word and its surrounding context) and $\alpha_{t,i}$ is the attention weight representing how much importante the decoder places on the source annotation h_i while generating the target word at step t.

b)
It represents the alignment probability or "relevance", mais ou menos. It quantifies how relevant the i-th source word is for generating the t-th target word. It tells the model "where" to look in the source sentence. For example, when generating the word "programação", the model assigns a high α value to the source word "programming".
This removes the bottleneck of trying to compress the entire sequence into a single static vector, allowing the model to focus on different parts of the input sequence at every time step.

Part 3

1 - Manual Calculation of Self-Attention

a)

$$q_1 = x_1 \cdot W^Q = [1, 0] \cdot \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = [1, 0]$$

b)

$$k_1 = x_1 \cdot W^K = [1, 0] \cdot \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = [1, 0]$$

$$k_2 = x_2 \cdot W^K = [0, 1] \cdot \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = [1, 1]$$

c)

$$v_1 = x_1 \cdot W^V = [1, 0] \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = [0, 1]$$

$$v_2 = x_2 \cdot W^V = [0, 1] \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = [1, 0]$$

d)

$$score_{1,1} = q_1 \cdot k_1 = [1, 0] \cdot [1, 0] = 1$$

$$score_{1,2} = q_1 \cdot k_2 = [1, 0] \cdot [1, 1] = 1$$

e)
The scores are $[1, 1]$.

$$e^1 \approx 2.718, \quad \sum = 2.718 + 2.718 \approx 5.436$$

$$\alpha_{1,1} = \frac{2.718}{5.436} = 0.5$$

$$\alpha_{1,2} = \frac{2.718}{5.436} = 0.5$$

f)

$$z_1 = \alpha_{1,1} v_1 + \alpha_{1,2} v_2$$

$$z_1 = 0.5 \cdot [0, 1] + 0.5 \cdot [1, 0]$$

$$z_1 = [0, 0.5] + [0.5, 0] = [0.5, 0.5]$$

2 - Positional Encodings

a)
RNNs process data sequentially, the order of steps is implicitly captured, the hidden state h_t depends on h_{t-1}.
Transformers process the entire sequence in parallel. The self-attention mechanism is permutation invariant, it just calculates the relationships between words regardless of their distance or order. Explicit positional encodings makes it possible for the attention mechanism to take that into account.

b)
It would treat them as the same sentence 🤣🫵.

3 - BERT vs. GPT Architecture

a)
BERT is encoder-only. It uses bidirectional attention. The attention mask allows eeeeeeeeevery token to attend to aaaaaall other tokens in the sequence (both to the left and right simultaneously).
GPT is decoder-only. It uses causal (masked) attention. The attention mask is triangular (upper-triangular elements are set to $-\infty$). So, every token at position t can only attend to previous positions, preventing it from "seeing" future words.

b)
BERT (Masked Language Modeling):

- Randomly masks some input tokens with the `[MASK]` token.
- The model uses surrounding context (both from left and right) to predict the original masked word. Forces the model to understand bidirectional context.
  GPT (Causal Language Modeling)
- Standard ass autoregressive prediction.
- Given a sequence of tokens $x_1, \ldots, x_{t-1}$, the model just tries to predict the next token x_t. It learns to model the probability distribution of the next word based on the history.
  - Agora pensam assim: esta bodega de ver tokens futuras não parece ajudar nada. E eu respondo assim:
    - The model's goal during training is to predict future tokens, if it could see them, it could cheat. The attention mechanism would simply copy the next token from the input rather than predict it based on context. The loss would drop to zero immediately.

4 - Large Language Models (LLMs) RLHF
a)
The reward model acts as a learned proxy for human preferences. It allows the RL algorithm (e.g., PPO) to optimize the LLM without needing a human to rate every output in real time.
Input: Prompt and generated response.
Output: Score indicating the quality of that response.

b)
In-Context Learning involves providing the model examples of the task (e.g., Input: "I love programming", Output: "Adoro programar") directly inside the prompt (INSIDE CONTEXT) before asking it to complete a new input (Input: "I hate programming", Output: ?).
No, this does not update the weights of the model. It just adapts its output.

Part 4

5 - Manual Calculation of GCN Message Passing
a)
Nodes 1, 2 and 3

b)
The degree $c_u v = 3$, because the graph is not directed, and node 1 has 2 edges and 1 self-loop.

- Sum:

$$h_{sum} = h_1 + h_2 + h_3$$

$$h_{sum} = [1, 0] + [0, 1] + [1, 1] = [2, 2]$$

  - Normalize (Mean):

$$h_{agg} = \frac{1}{3} \cdot [2, 2] = \left[\frac{2}{3}, \frac{2}{3}\right]$$

c)

$$z = I \cdot h_{agg} = \left[\frac{2}{3}, \frac{2}{3}\right]$$

$$h_1^{(1)} = \mathrm{ReLU}\,(z) = \left[\frac{2}{3}, \frac{2}{3}\right]$$

6 - Permutation Invariance

a)

CNNs are design for fixed and grid-structured data (like images) where there is a meaningful spatial order.

Graphs have no natural ordering of nodes. An adjacency matrix represents the graph structure, but if we reorder the rows and columns, the adjacency matrix changes even though the graph topology is identical. A CNN is sensitive to this ordering, if applied to the adjacency matrix, it would learn patterns specific to one arbitrary node ordering!

b)

It means that its output does not change when the input nodes are reordered.

This is important because the indexing of nodes in a graph is arbitrary. The model should produce consistent predictions regardless of how data is stored. (It doesn't matter if a person is node 1 or 100 in a social network, we just care about topology).

7 - Spectral vs. Spatial GNNs

a)

Usually, an adjacency matrix has 0s on the diagonal, meaning a node is not considered a neighbor of itself.

By adding I, we add self loops, during matrix multiplication, the nodes include their own features in the update steps! Without this, the new representation would be based only on neighbors, discarding the node's previous state.

b)

$\tilde{D}$ is the degree matrix of the graph. Each value $\tilde{D}_{ii}$ represents the degree of node i.

Because we are adding $\tilde{A}$ (self-loops), the degree includes the node itself.

$$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$$

Example for a 3 node graph:

$$\tilde{D} = \begin{bmatrix} \deg(1) & 0 & 0 \\ 0 & \deg(2) & 0 \\ 0 & 0 & \deg(3) \end{bmatrix}$$

$\tilde{D}^{-1/2}$ Is the inverse square root of the degree matrix. Since the matrix is diagonal, it's easy to calculate:

$$\tilde{D}^{-1/2} = \begin{bmatrix} \frac{1}{\sqrt{\text{deg}}} & 0 & 0 \\ 0 & \frac{1}{\sqrt{\text{deg}(2)}} & 0 \\ 0 & 0 & \frac{1}{\sqrt{\text{deg}(3)}} \end{bmatrix}$$

We can then use it for normalization. In the formula $\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$, we are sandwiching the adjacency matrix between these degree matrices. Basically, this divides the signal coming from neighbor j to node i by $\sqrt{\text{deg}(i) \cdot \text{deg}(j)}$.

- Without it, nodes with high degree would gather huge feature values just because they have more connections, while isolated nodes would have tiny values. That would make training unstable.

Part 4

1 - Markov Decision Processes (MDP) & Bellman Calculation

a)

Since the action is deterministic, the formula is:

$$Q(s,a) = R + \gamma V(s')$$

- $R = 1$
- $\gamma = 0.9$
- $V(s_{next}) = 5$

$$Q(s_t, a_L) = 1 + 0.9(5) = 1 + 4.5 = 5.5$$

b)

Since the action is stochastic, we calculate the expected value (weighted sum of probabilities):

$$Q(s,a) = \sum P(s'|s,a)[R + \gamma V(s')]$$

1. Win Path (0.8):

$$0.8 \times [10 + 0.9(20)] = 0.8 \times [10 + 18] = 0.8 \times 28 = 22.4$$

2. Fail Path (0.2):

$$0.2 \times [-5 + 0.9(0)] = 0.2 \times [-5] = -1.0$$

3. Total Q-value:

$$Q(s_t, a_R) = 22.4 + (-1.0) = 21.4$$

c)

If the agent acts greedily, it selects the action with the highest Q-value.

$$V(s_t) = \max(Q(s_t, a_L), Q(s_t, a_R))$$

$$V(s_t) = \max(5.5, 21.4) = 21.4$$

2 - Manual Q-Learning Iteration

The Q-Learning update rule is:

$$Q_{new}(S_t, A_t) = Q_{old}(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q_{old}(S_t, A_t) \right]$$

Find best Q-value for the next state (B):

$$\max_{a'} Q(B, a') = 5.0$$

Calculate the temporal difference target:

$$Target = R_{t+1} + \gamma \max Q(S_{t+1}, a')$$

$$Target = -2 + 4.5 = 2.5$$

Update Q-value:

$$Q_{new}(A, \text{Run}) = 2.5 + 0.1[2.5 - 2.5]$$

$$Q_{new}(A, \text{Run}) = 2.5 + 0.1[0]$$

$$Q_{new}(A, \text{Run}) = 2.5$$

The TD error was zero, so the prediction was already aligned with the observed target.

3 - Exploration vs. Exploitation

a)

In a ϵ-greedy policy:

- Exploitation $(1 - \epsilon)$: We pick the greedy action
- Exploration (ϵ): We pick a random action uniformly from all available actions.
  So, the probability of intentionally exploiting the "best" action is $1 - \epsilon = 0.8$, but the best action can also be chosen "accidentally" when trying to pick a random action. The probability of picking each action at random is $\frac{\epsilon}{|A|} = \frac{0.2}{4} = 0.05$.

So, the total probability of selecting the action with value 15 is $0.8 + 0.05 = 0.85$.

b)

$$P(\text{Random Action}) = \frac{\epsilon}{|A|} = 0.05$$

4 - Deep Q-Networks (DQN)

a) Experience Replay

Instead of updating the neural network immediately after every step with the latest experience tuple (s, a, r, s'), the agent stores these transitions in a memory buffer ("replay buffer"). During training, the agent samples a random mini-batch of experiences from this buffer to perform the weight update.

It is necessary because it breaks correlation. Sequential data in RL is highly correlated (what happens at step t is very similar to $t+1$). Training a network on correlated data leads to overfitting and instability (the model forgets everything else and only optimizes for the current specific trajectory). Random sampling breaks this, it makes the data look more i.i.d, that is required for stable gradient descent.

b) Target Networks
We maintain two separate networks:

1. **Main Network (θ):** Used to select actions and is updated at every step or every few steps.
2. **Target Network (θ^-):** A copy of the main network, but it's frozen and only updated periodically.
   The "Moving Target" Problem:

- In the loss function $L(\theta) = (y_i - Q(s, a; \theta))^2$, the target y_i depends on the network's own estimates ($y_i = R + \gamma \max Q(s', a'; \theta)$).
- Without a frozen network, updating θ to increase $Q(s, a)$ also tends to increase $Q(s', a')$ (because the states are similar). This means that the target y_i also moves up. The network ends up chasing a constantly **-> MOVING TARGET <-**, leading to oscillation or divergence.
- To solve this, we use θ^-, which fixes the target values for a while, providing a stable objective for the main network to learn against.

By combining both experience replay and target networks, training DQNs is more stable. The training loop becomes:

1. **Interaction:**
   - At every single timestep t:
     - The agent looks at the current state S_t.
     - It picks an action A_t, using the main network (using ϵ-greedy).
     - It executes an action, gets reward R_{t+1} and next state S_{t+1}.
     - It saves this tuple $(S_t, A_t, R_{t+1}, S_{t+1})$ into the experience replay buffer.
       - WE DO NOT UPDATE WEIGHTS YET.
2. **Training the Main Network:**
   - This happens frequently (e.g., every single timestep, or every 4 timesteps):
     - Sample a random batch of 32 (or 64, 128, ...) experiences from the replay buffer.
     - Calculate the loss using the target network for the "ground truth" guess:

$$\text{Target} = R + \gamma \max_{a'} Q(S', a'; \theta_-)$$

     - And the prediction using the main network:

$$\text{Prediction} = Q(S, A; \theta)$$

    - Update the main network weights (θ) via gradient descent to minimize the difference.
3. **Updating the Target Network:**
    - This happens rarely (e.g., every 1000 or 10000 steps).
        - Simply copy the current weights from the Main Network over the Target Network.

$$\theta^- \leftarrow \theta$$

        - This resets the frozen target to match the current best learner, then stays frozen again.

5 - Credit Assignment Problem
The credit assignment problem consists in determining which past action (or actions) is responsible for a current outcome (reward/punishment).

Sparse rewards, such as in a game like chess, an agent makes a lot of oves but only receives one signal (reward +1 or -1) at the end. For example, if the agent wins, it's difficult to know which specific moves helped winning.
Also, with sparse rewards (zeros everywhere exepct the end), the gradient for earlier states is weak/non-existent, so the agent can't learn a policy for earlier stages (e.g., opening, midgame).

Part 5

1 - Vision Transformers (ViT) - Manual Calculation
a)
Patches along height: $\frac{48}{16} = 3$
Patches along width: $\frac{48}{16} = 3$
Total patches: $3 \times 3 = 9$

b)
Before the linear projection, we flatten each patch to a 1D vector containing all its raw pixel values. $P \times P \times C = 16 \times 16 \times 3 = 768$

c)
We have 9 patches and 1 class token, each one is projected down to the Transformer's hidden dimension D. The final shape is $(10, 128)$.

2 - LoRA (Low-Rank Adaptation) - Parameter Efficiency
a)
For standard fine-tuning, we need to update every weight in the matrix.
$d_{in} \times d_{out} = 1000 \times 1000 = 10^6$ parameters.

b)
For LoRA, we freeze the original matrix and only train two smaller low-rank matrices A and B.

- A needs $8 \times 1000 = 8000$ parameters and B needs $1000 \times 8 = 8000$ parameters. We need 16000 in total.

c)
$\frac{10^6}{16000} = 62.5$

3 - Diffusion Models
a)
The forward process (aka diffusion process) is a fixed Markov chain that gradually adds Gaussian noise to the data. Starting with a clean image x_0, at each step t, a small amount of noise is added according to a schedule. The final state x_T (typically $T = 1000$), the original structure of the image is destroyed, the final state is just Gaussian noise.

b)
A network is trained to reverse this process. It does not predict the fully denoised image x_0 directly at each step. The standard objective is for the network to predict the noise component ϵ that was added to the image at the current timestep t. By subtracting this predicted noise, the model moves one step backward towards a slightly cleaner image x_{t-1}.

4 - CLIP (Contrastive Language-Image Pre-training)
a)
The loss function tries to maximize the values on the diagonal of the $N \times N$ similarity matrix. These are the entries that correspond to the dot products $I_i \cdot T_i$, the images and their correct captions.

b)
It tries to minimize off-diagonal values $I_i \cdot T_j$ (where $i \neq j$), which are incorrect pairings.

c)
Because CLIP learns a shared embedding space between images and text, it can turn classification into a retrieval task.

- Instead of training a new final layer for specific classes (like "cat" or "dog"), you can feed the text prompt "a photo of a cat" and "a photo of a dog" into the text encoder.
- We then compare these texts embeddings to the image embedding.

5 - RAG (Retrieval Augmented Generation)
a)
Standard LLMs rely on parametric memory. This knowledge is frozen at the time of training, so it becomes outdated and the model might hallucinate. RA retrieves relevant information from an external knowledge base before generating a response.
This retrieved text is injected into the LLM's context window (prompt). The model is then asked to answer the user's question using only the provided content to ground itself in factual data.

b)
To perform RAG, we need to find text that is semantically relevant to the query, not just text that shares the same keywords. Vector databases store vector representations of text. The database performs a similarity search (cosine similarity or dot product) to find the nearest neighbors. It compares the vector of the user's query against billions of document vectors to retrieve the top k chunks of text that are most similar in meaning.

Part 6

a)
1 - Data Representation

- Images are static spatial data (2d grids of pixels with c channels). Audio is represented as a temporal sequential data (1d sequence of amplitude values over time).
- Raw amplitude sequences are highly variable and high-dimensional. Spectrograms reveal frequency patterns and allow standard computer vision models to be applied to audio.

2 - The Distributional Hypothesis

- A word's meaning is defined by its context. In Word2Vec, the model does not understand definitions, it just learns to assign similar vector representations to words that appear in similar contexts.

3 - One-Hot Encoding Limitations

- Curse of Dimensionality: One-Hot vectors have a dimension equal to the vocabulary size. If $|V|$ is large, every input is also a large vector. This leads to massive memory usage and inefficiency (vocabulary explosion).
- Morphologically rich languages would assign a unique entry in the vocabulary to each variation of a word. $|V|$ would explode even further.

4 - GloVe Objective Function

- The weighting function scales the loss based on co-occurence frequency X_{ij}.
- It reduces the impact of extremely frequent stop words ("the", "and") so they don't dominate the training.
- It filters out noise from rare co-occurrences that might be accidental.

5 - Evaluation Methods

- Intrinsic methods evaluate the quality of the embeddings directly (geometric properties like distance or analogy) independent of a specific task.
    - Example of intrinsic task: "king is to a man as queen is to ?"
- Extrinsic methods evaluate the embeddings by using them as inputs for a real-world downstream task (like sentiment analysis) and measuring the final model's accuracy.

- A model might score high on analogies (intrinsic) but fail on sentiment analysis (extrinsic) because "good" and "bad" often appear in identical contexts ("the movie was good/bad"). This makes their vectors similar (good for analogy), which is terrible for a sentiment classifier that needs to distinguish them.~

6 - Decoding Strategies: Beam Search

- Greedy decoding: at each step, the model selects the token with highest probability and never looks back, an early stage can never be corrected.
- Beam Search: tracks the top k most probable sequences (hypotheses) simultaneously at each step. It explores multiple potential paths rather than just one.
- If beam width $k = 1$, it's identical to greedy because it only keeps the single "best" path at each step.
- Beam search is slower because it needs to calculate and maintain k branches at every timestep, but improves the quality by reducing the risk of getting stuck in a suboptimal path.

7 - Reversing the Source Sentence
Reversing the source "ABC" to "CBA" introduces some short-term dependencies between the source and the target. The last word of the source becomes the first word read by the encoder, so it's "physically" closer to the start of the decoding process. This reduces the path that the gradient has to travel during backpropagation, so the optimization is easier for early parts of the sentence.

8 - Character-Level Encoding.
Advantage: we can handle OOV words since the vocabulary is just a set of characters.
Disadvantage: the sequence length explodes. This increases the computational cost because the attention mechanism and RNN steps scale with sequence length.

9 - Multi-Head Attention Mechanism
After computing the attention outputs for all h heads independently, the results are concatenated to form a single long vector.
This concatenated vector is then multiplied by an output weight matrix W^O (linear transformation) to project it back to the original dimension d_{mdel}.

10 - Feed-Forward Networks in Transformers:
ReLU in the original transformer paper and GELU in models like BERT or GPT.

11 - Layer Normalization
Batch normalization computes statistics (mean and variance) across the batch dimension. It normalizes a specific feature channel across all samples in a batch.

Layer normalization computes statistics across the feature dimension for a single sample. It normalizes all features for a specific token/word, independent of other samples in the batch.

LN is preferred for variable length sequences such as NLP because BN is difficult to apply to variable lengths (different padding) and relies on stable batch statistics, which are hard to maintain with small batches or sequential dependencies (RNNs). LN works on each sample individually, making it invariant to batch size and sequence lengths (ideal for text).

12 - Types of Graph Tasks
Node classification: Predicting a property of a specific node within a graph (classify whether a user in a social network is a bot or a human based on connections).

Link prediction: predicting whether an edge exists (or should exist) between two nodes (suggesting a friend recommendation).

Graph classification: predicting a property of the entire graph structure (predicting whether a chemical molecule is toxic or non-toxic).

13 - Adjacency Matrix & Self-Loops
The node would discard its own features and its representation would be only based on the neighbors.

14 - Oversmoothing
In deep GCNs with too many layers, you repeat the local averaging (smoothing) process too many times. Eventually, the node representations for all connected nodes converge to the same value (the global average). The nodes become indistinguishable from one another.

15 - On-Policy vs. Off-Policy
Off-policy means that the algorithm learns with the value of the optimal policy (what it should do) independently of the agent's actual actions (what it is doing). It can learn from data generated by a completely different behavior policy.

- Q-Learning (off-policy): updates the Q-value using the maximum possible reward of the next state ($\max_{a'} Q(s', a')$). It assumes the agent will act greedily in the future, even if it's just exploring.
- SARSA (on-policy): updates the Q-value using the actual action a' that the agent selected for the next step: $Q(s', a')$. It learns the value of the policy being followed, including its exploration steps.

16 - The Makov Property
The Markov Property states that the future is independent of the past given the present. In an MDP, this means that the current state S_t contains all necessary information to decide the future and that knowing the history $S_{t-1}, S_{t-2}, \ldots$ adds no new value.

It is essential for S_t to capture the history because the agent makes decisions solely based on S_t. If S_t is incomplete (e.g., a single frame of a moving ball without velocity info), the environment appears random or non-stationary, and the agent cannot learn an optimal policy.

17 - Reward Hypothesis
All of what we mean by goals can be thought of as maximizing the expected value of the cumulative sum of a received scalar signal (reward).

"All goals can be described by the maximization of expected cumulative reward".

18 - Zero-Shot vs. Few-Shot Prompting

- Zero-shot: Give a model a description of the task without specific examples on how to solve it. The model relies on pre-training knowledge.
- Few-Shot Prompting: Providing a few examples of input-output pairs in the prompt before the actual query. This primes the model for the specific format or task.
    - There are no weight updates.

19 - Audio Models (Whisper/Wav2Vec)

- Learn powerful speech representations directly from raw audio waveform, which can be fine-tuned for tasks like speech recognition.
- Self-supervised learning: It learns by masking. The model encodes raw audio into latent representations, masks certain time steps (similar to BERT's MLM) and then tries to predict the correct quantized representation for the masked sections.

20 - CLIP's Training Data

- Difference from supervised learning: traditional supervised learning trains a model to predict a fixed set of integers/classes, these models learn "class 0 looks like this" but don't understand the meaning of the class itself.
- CLIP learns to map images and texts into a shared semantic space. Because it learns the relationship between visual features and language (text description), it can recognize any named object. It doesn't need a specific "apple" output neuron. It just needs to know what the word "apple" means in the embedding space, allowing it to match it to a round red fruit in the image.