Multimodal model: DALL-E

https://mlberkeley.substack.com/p/dalle2

DALL-E consists of two main components.

- Discrete variational autoencoder (dVAE) learns to accurately represent images in a compressed latent space.
    - DALL-E uses a slightly different method to learn it's discrete representations; they call it dVAE. While the exact techniques are a bit different, the core goal remains the same. Nonetheless, I will briefly explain the differences at the beginning of this post. The rest of this post will focus on DALL-E's transformer.
    - **Discrete Latent Space via Vector Quantization:** The **vector quantization (VQ)** module prevents continuous interpolation that could lead to unnatural, morphed images:
    - **Codebook Discreteness:**
      Images are compressed into discrete indices from a learned codebook. Each latent token represents a fixed, meaningful visual pattern, avoiding arbitrary continuous representations. The codebook enforces discreteness in the latent space.

- The transformer is arguably the meat of DALL-E; it is what **allows the model to generate new images that accurately fit with a given text prompt**. **It learns how language and images fit together**, so that when the model is asked to generate images of "an armchair in the shape of an avocado", it's able to spit out some super creative designs for Avocado chairs that have probably never been thought of before
- Transformer which learns the correlations between language and this discrete image representation.

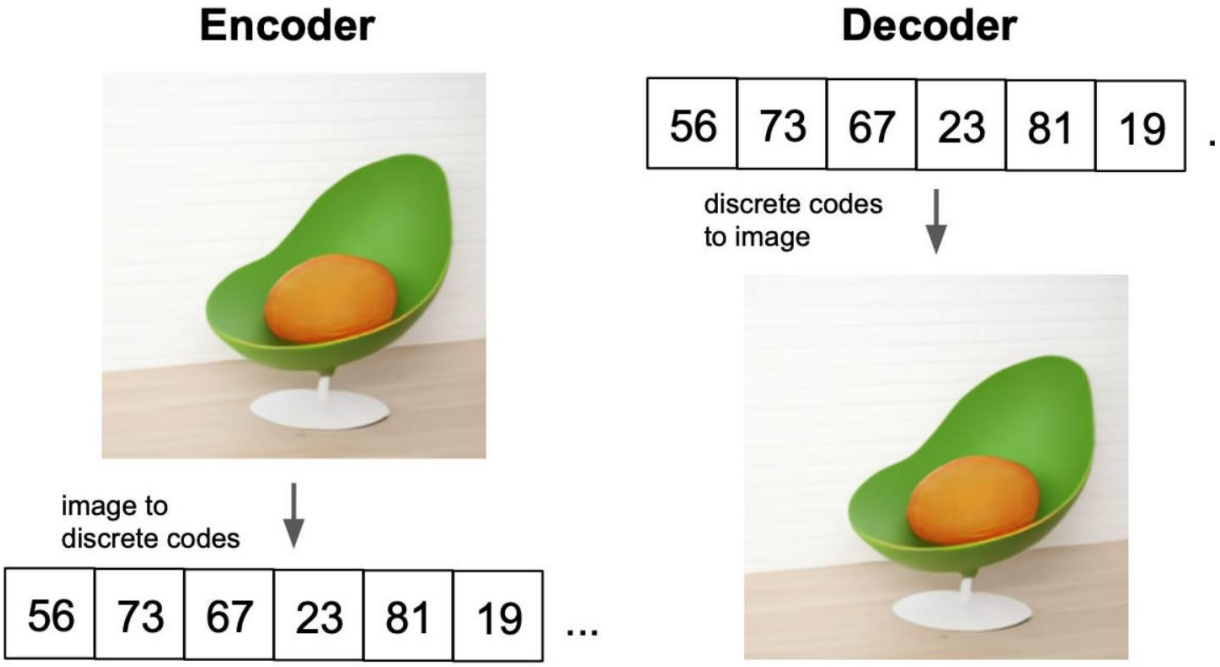
TEXT PROMPT

an armchair in the shape of an avocado [...]

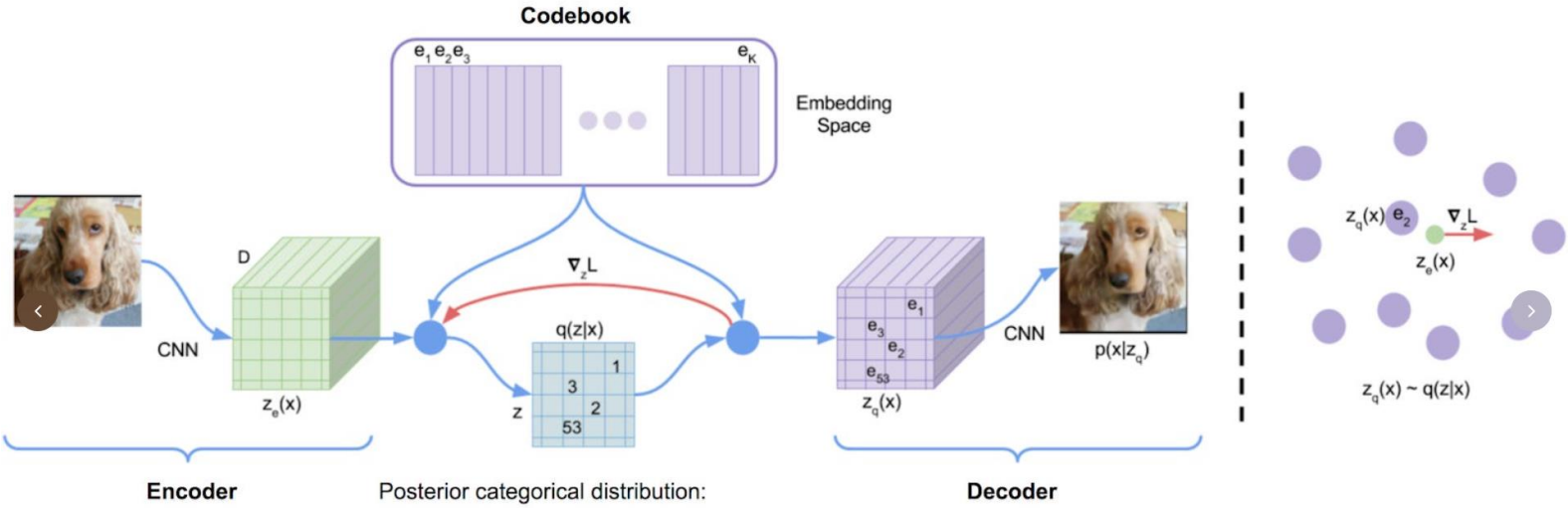
AI-GENERATED IMAGES

dVAE explained

Ultimately, the high level goal of dVAE is the same: it attempts to learn a discrete representation for images. The specifics of how exactly it does this are just a bit different from VQ-VAE.

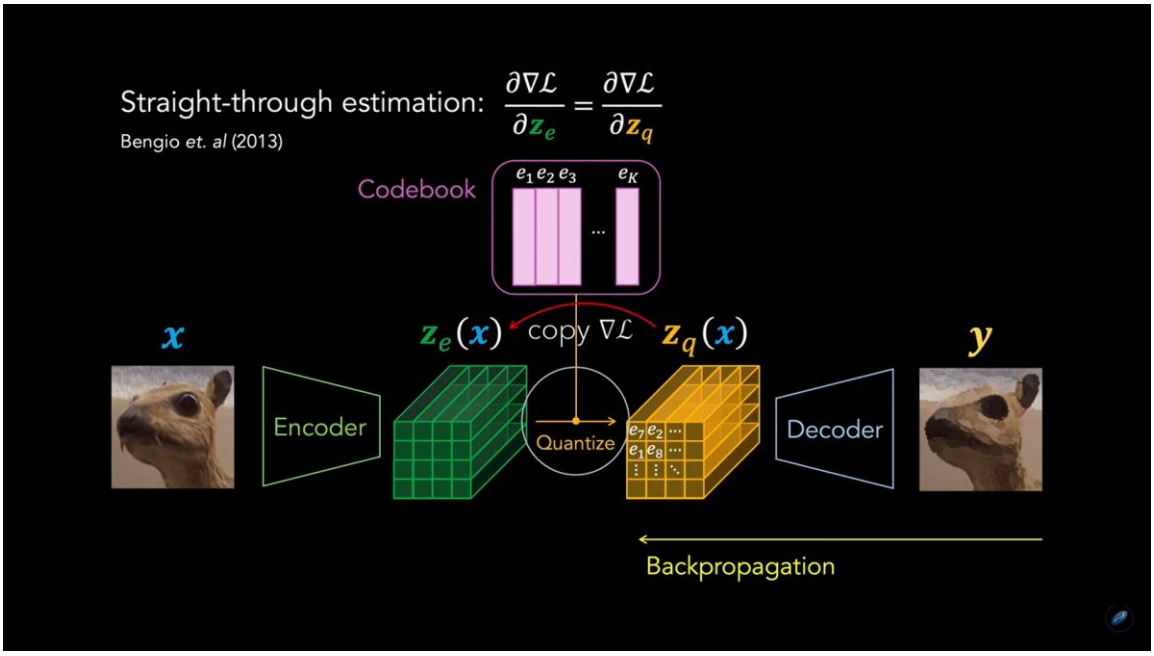


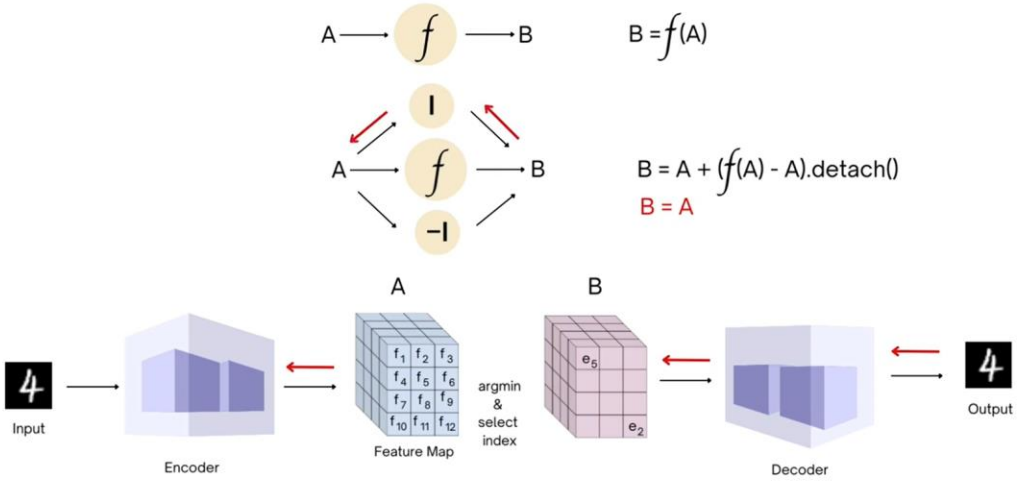
VQ-VAE Review:



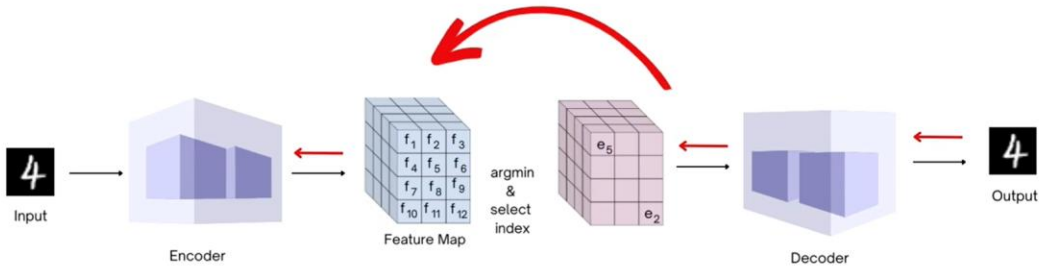
$$q(\mathbf{z} = \mathbf{e}_k|\mathbf{x}) = \begin{cases} 1 & \text{if } k = \arg\min_i \|\mathbf{z}_e(\mathbf{x}) - \mathbf{e}_i\|_2 \\ 0 & \text{otherwise.} \end{cases}$$

dVAE:


Straight-through estimation: $\frac{\partial \nabla \mathcal{L}}{\partial z_e} = \frac{\partial \nabla \mathcal{L}}{\partial z_q}$
Bengio et. al (2013)
Codebook
$e_1 e_2 e_3$
e_K
x
$z_e(x)$
copy $\nabla \mathcal{L}$
$z_q(x)$
y
Encoder
Quantize
Decoder
Backpropagation

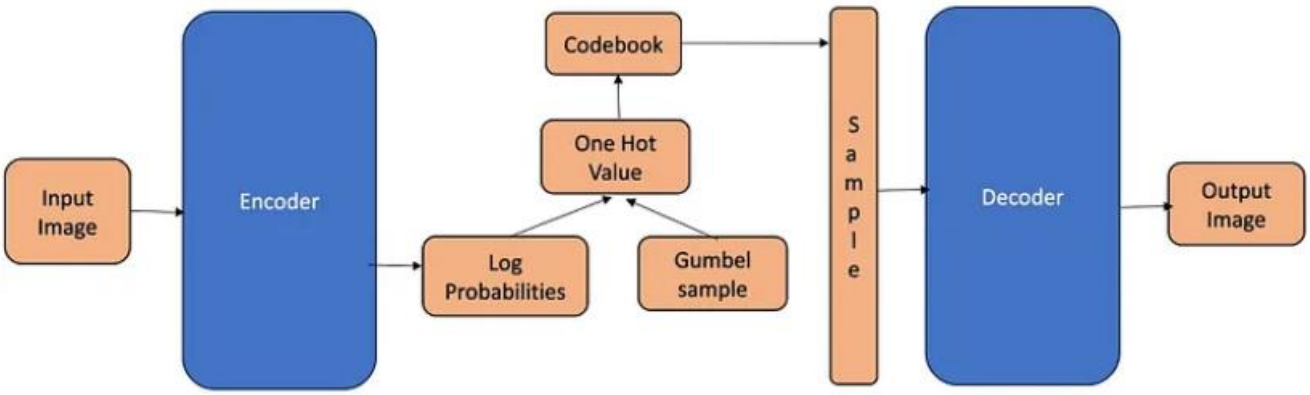

A
f
B
B = f(A)
I
A
f
B
−I
B = A + (f(A) - A).detach()
B = A
A
B
Input
Encoder
Feature Map
argmin & select index
Decoder
Output

Architecture



quantized = inputs + (quantized - inputs).detach()

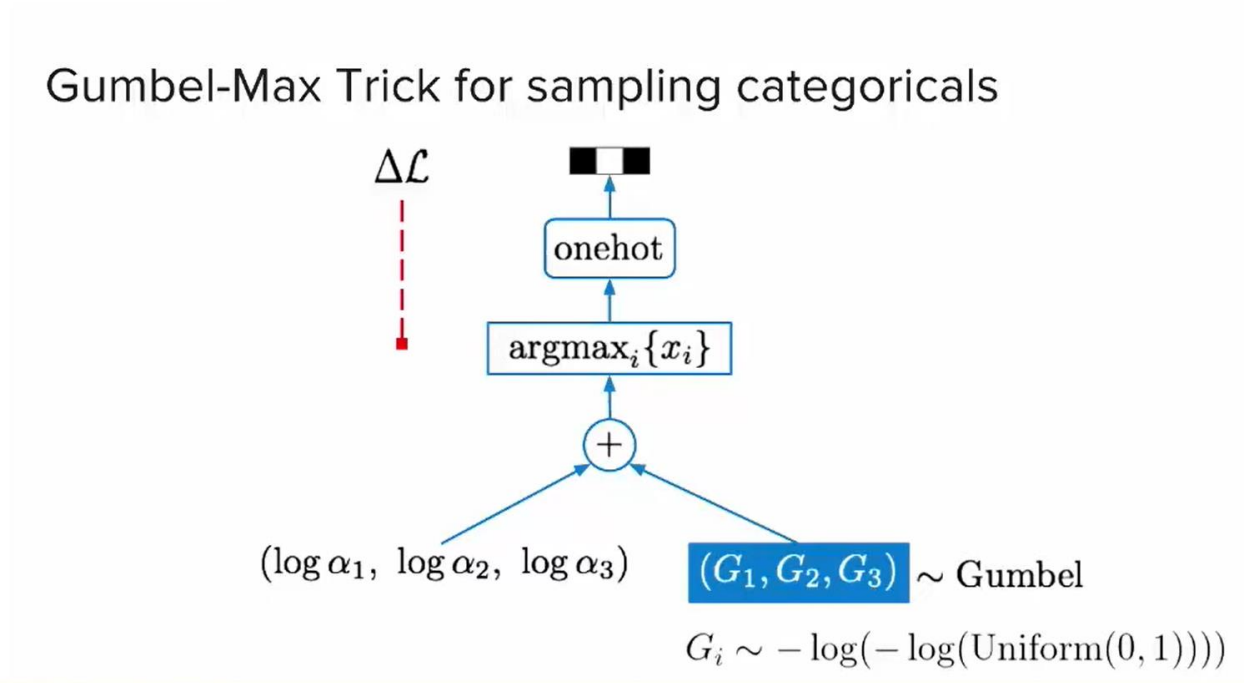
https://jaswanth04.medium.com/discrete-variational-auto-encoder-explained-41493ebe294d



Discrete Variational Auto Encoder with Gumbel Sampling

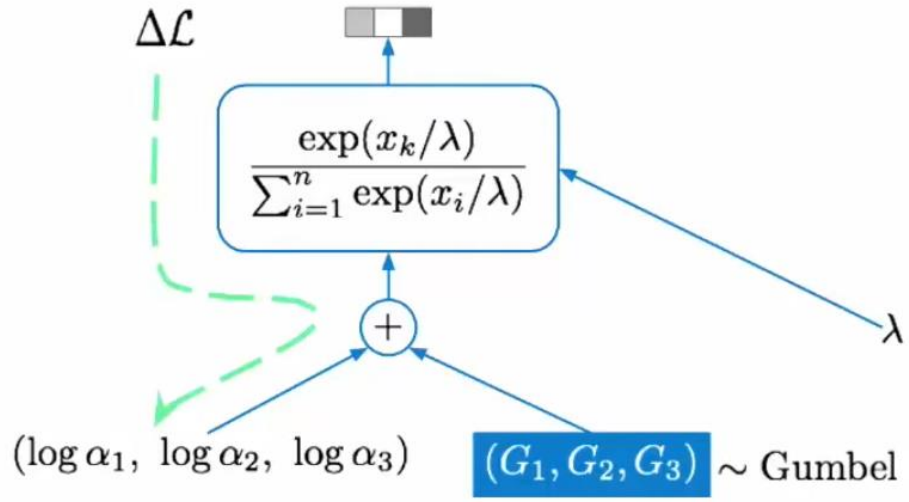
Comibine smooth relaxion and reparameterization trick: Gambal max rick

Ask a question at **goo.gl/slides/nkqkm9**



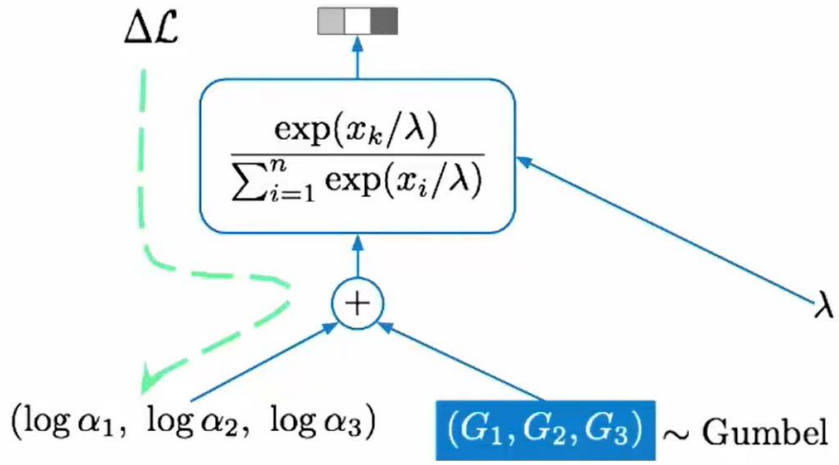
Ask a question at **goo.gl/slides/nkqkm9**

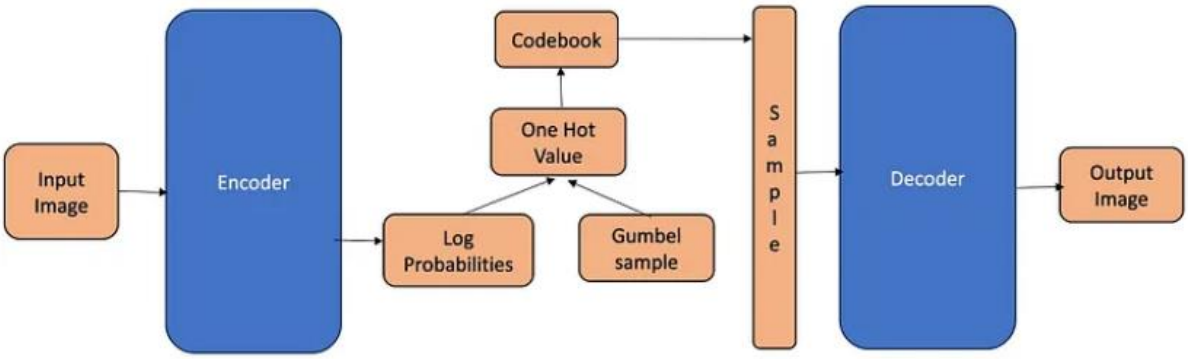
Gumbel-Softmax / Concrete Distribution



Ask a question at **goo.gl/slides/nkqkm9**

Gumbel-Softmax / Concrete Distribution





Discrete Variational Auto Encoder with Gumbel Sampling

$$y_i = \frac{e^{\frac{g_i + log(q(e_i|x))}{\tau}}}{\sum_{j=1}^{k} e^{\frac{g_j + log(q(e_j|x))}{\tau}}}$$

the sampled latent vector is then just a weighted sum of these codebook vectors:

$$z = \sum_{j=1}^{k} y_j e_j$$

Success! We can now differentiate through the sampling operation with respect to the encoder outputs, $q(e_i/x)$. This is because:

Toy Example Setup

1. **Latent Grid**:
   - 2×2 grid.
2. **Codebook**:
   - 3 vectors (e_0, e_1, e_2):
     - $e_0 = [0.1, 0.2, 0.3]$
     - $e_1 = [0.4, 0.5, 0.6]$
     - $e_2 = [0.7, 0.8, 0.9]$
3. **Encoder Outputs (Logits)**:
   - For each grid position, the encoder outputs logits representing the likelihood of each codebook vector:
     - Position $(1, 1)$: $[1.0, 2.0, 0.5]$
     - Position $(1, 2)$: $[0.5, 1.5, 1.0]$
     - Position $(2, 1)$: $[2.5, 0.5, 1.5]$
     - Position $(2, 2)$: $[1.2, 0.8, 2.0]$

Soft Sampling

In **Soft Sampling**, a "soft" combination of the codebook vectors is computed. This allows gradients to flow during training. It uses the **Softmax function** or the **Gumbel-Softmax distribution**.

Step-by-Step Process

1. **Compute Probabilities (Softmax)**:
    - Convert logits into probabilities using the softmax function:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Position (1,1):

- Logits: $[1.0, 2.0, 0.5]$
- Probabilities:

$$\text{Softmax}([1.0, 2.0, 0.5]) = [0.231, 0.628, 0.141]$$

Position (1,2):

- Logits: $[0.5, 1.5, 1.0]$
- Probabilities:

$$\text{Softmax}([0.5, 1.5, 1.0]) = [0.211, 0.576, 0.213]$$

Similarly, compute for other positions.

2. **Weighted Average (Soft Combination):**
    - For each grid position, compute the **weighted sum** of the codebook vectors, using the softmax probabilities as weights.

$$\text{Soft Vector} = \sum_{i=0}^{K-1} \text{probability}_i \cdot e_i$$

Position (1,1):

- Probabilities: $[0.231, 0.628, 0.141]$
- Weighted sum:

$$\text{Soft Vector} = (0.231 \cdot [0.1, 0.2, 0.3]) + (0.628 \cdot [0.4, 0.5, 0.6]) + (0.141 \cdot [0.7, 0.8, 0.9])$$

$$= [0.367, 0.472, 0.578]$$

Similarly, compute for other positions.

Soft Sampling (sampling of codebook vector)

2. Gumbel-Softmax Sampling

The **Gumbel-Softmax distribution** introduces random noise to logits to allow sampling, while maintaining differentiability.

Steps

1. **Add Gumbel Noise**:
   - For each logit xi, add Gumbel noise:

$$g_i = -\log(-\log(U)), \quad U \sim \text{Uniform}(0,1)$$

$$z_i = x_i + g_i$$

2. **Apply Softmax with Temperature**:
   - The Gumbel-softmax output y_i is computed as:

$$y_i = \frac{\exp(z_i/\tau)}{\sum_j \exp(z_j/\tau)}$$

where $\tau > 0$ is the **temperature**:

- High τ: Softer probabilities (closer to uniform).
- Low τ: Sharper probabilities (closer to one-hot).

Computation for Position (1,1)

1. **Logits**: [1.0,2.0,0.5]
2. **Generate Gumbel Noise**:

1. **Logits**: $[1.0, 2.0, 0.5]$
2. **Generate Gumbel Noise**:
   - Draw 3 random samples $U_i \sim \text{Uniform}(0, 1)$: $U = [0.61, 0.42, 0.79]$.
   - Compute Gumbel noise:

$$g_i = -\log(-\log(U_i))$$

     - $g_0 = -\log(-\log(0.61)) = 0.48$
     - $g_1 = -\log(-\log(0.42)) = 0.79$
     - $g_2 = -\log(-\log(0.79)) = 0.23$
   - Gumbel noise: $[0.48, 0.79, 0.23]$
3. **Add Noise to Logits**:

$$z_i = x_i + g_i$$

   - $z_0 = 1.0 + 0.48 = 1.48$
   - $z_1 = 2.0 + 0.79 = 2.79$
   - $z_2 = 0.5 + 0.23 = 0.73$
   - Noisy logits: $[1.48, 2.79, 0.73]$
4. **Apply Softmax with Temperature** ($\tau = 0.5$):

$$y_i = \frac{\exp(z_i/\tau)}{\sum_j \exp(z_j/\tau)}$$

- Compute:
  - $\exp(1.48/0.5) = \exp(2.96) \approx 19.32$
  - $\exp(2.79/0.5) = \exp(5.58) \approx 266.97$
  - $\exp(0.73/0.5) = \exp(1.46) \approx 4.30$
- Denominator:

$$\sum_j \exp(z_j/\tau) = 19.32 + 266.97 + 4.30 = 290.59$$

- Probabilities:
  - $y_0 = \frac{19.32}{290.59} \approx 0.0665$
  - $y_1 = \frac{266.97}{290.59} \approx 0.9186$
  - $y_2 = \frac{4.30}{290.59} \approx 0.0148$
- Gumbel-Softmax Output: $[0.0665, 0.9186, 0.0148]$

5. **Weighted Average (Soft Combination)**:
   - Combine codebook vectors:

$$\text{Soft Vector} = (0.0665 \cdot e_0) + (0.9186 \cdot e_1) + (0.0148 \cdot e_2)$$

   - $(0.0665 \cdot [0.1, 0.2, 0.3]) = [0.00665, 0.0133, 0.01995]$
   - $(0.9186 \cdot [0.4, 0.5, 0.6]) = [0.36744, 0.4593, 0.55116]$
   - $(0.0148 \cdot [0.7, 0.8, 0.9]) = [0.01036, 0.01184, 0.01332]$
   - Final Soft Vector:

$$[0.384, 0.484, 0.584]$$

Repeat for Other Positions

Perform the same steps for other positions (1,2), (2,1), and(2,2). The process is identical, using their respective logits.

Final Outputs

1. **Soft-Sampled Codebook Vectors (Gumbel-Softmax Outputs):**

    1. Position (1,1): [0.384,0.484,0.584]
    2. Position (1,2): (calculated similarly)
    3. Position (2,1): (calculated similarly)
    4. Position (2,2): (calculated similarly)
2. **Resulting Vectors**: These are passed to the decoder as the latent representation.

Key Notes

- The Gumbel noise introduces randomness but retains differentiability.
- The temperature τ controls the "softness" of the sampling:
    - $\tau \to 0$: Approximates hard sampling (argmax-like behavior).
    - $\tau \to \infty$: Approaches uniform probabilities.

Hard Sampling

In **Hard Sampling**, a discrete vector is selected by taking the **argmax** of the logits, representing the most likely codebook vector. No weighted combination is used.

Step-by-Step Process

1. **Argmax**:

   - For each grid position, take the argmax of the logits to select the **most likely codebook vector**.

     Position (1,1):

     - Logits: $[1.0, 2.0, 0.5]$
     - Argmax: 1 (corresponding to e_1)
     - Selected Codebook Vector: $e_1 = [0.4, 0.5, 0.6]$

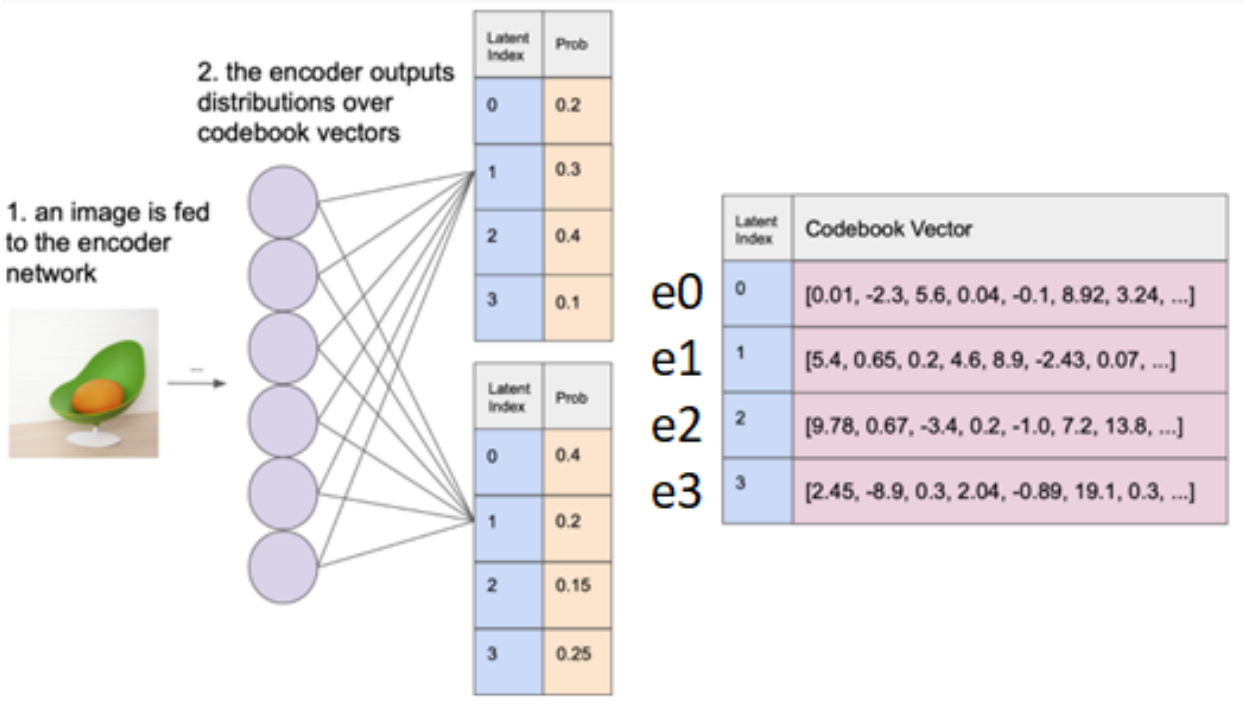
     Position (1,2):

     - Logits: $[0.5, 1.5, 1.0]$
     - Argmax: 1 (corresponding to e_1)
     - Selected Codebook Vector: $e_1 = [0.4, 0.5, 0.6]$

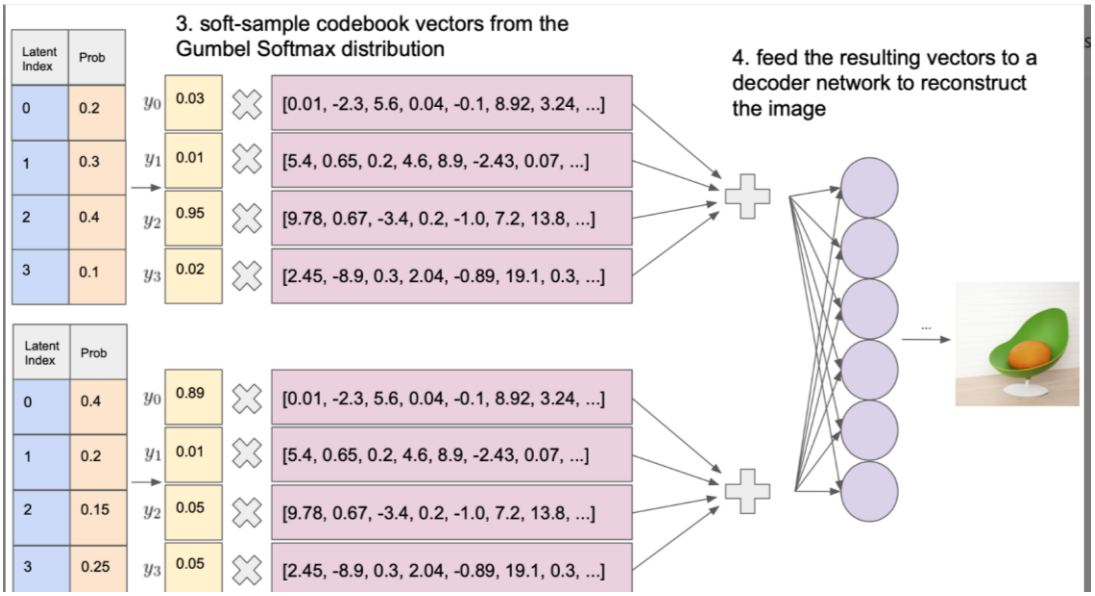
     Similarly, compute for other positions.

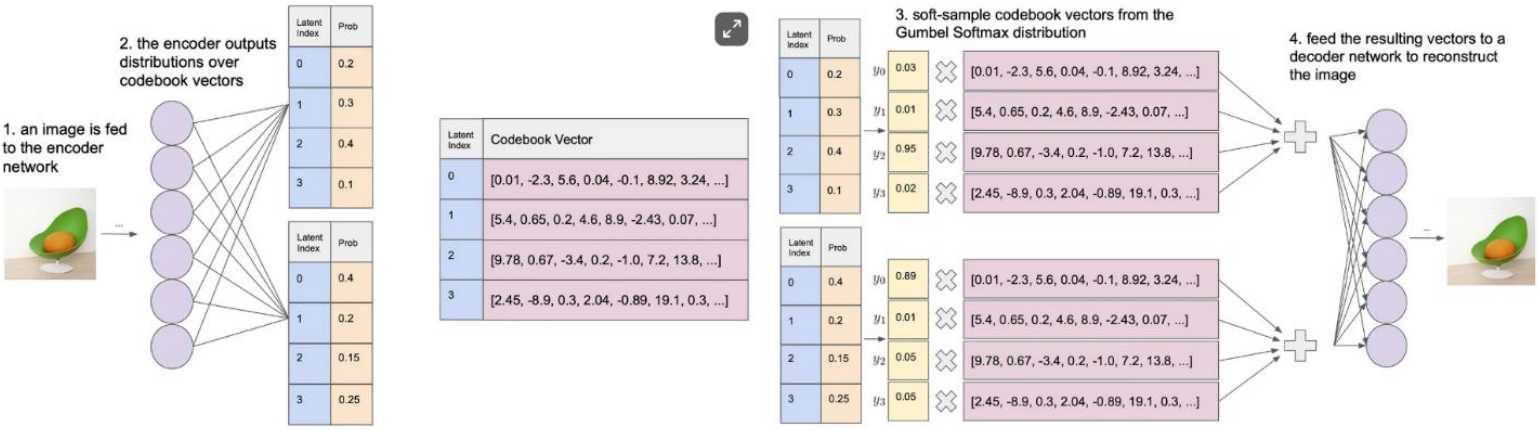
2. **Output**:

   - The final representation is a grid of selected **codebook vectors**, one for each position.



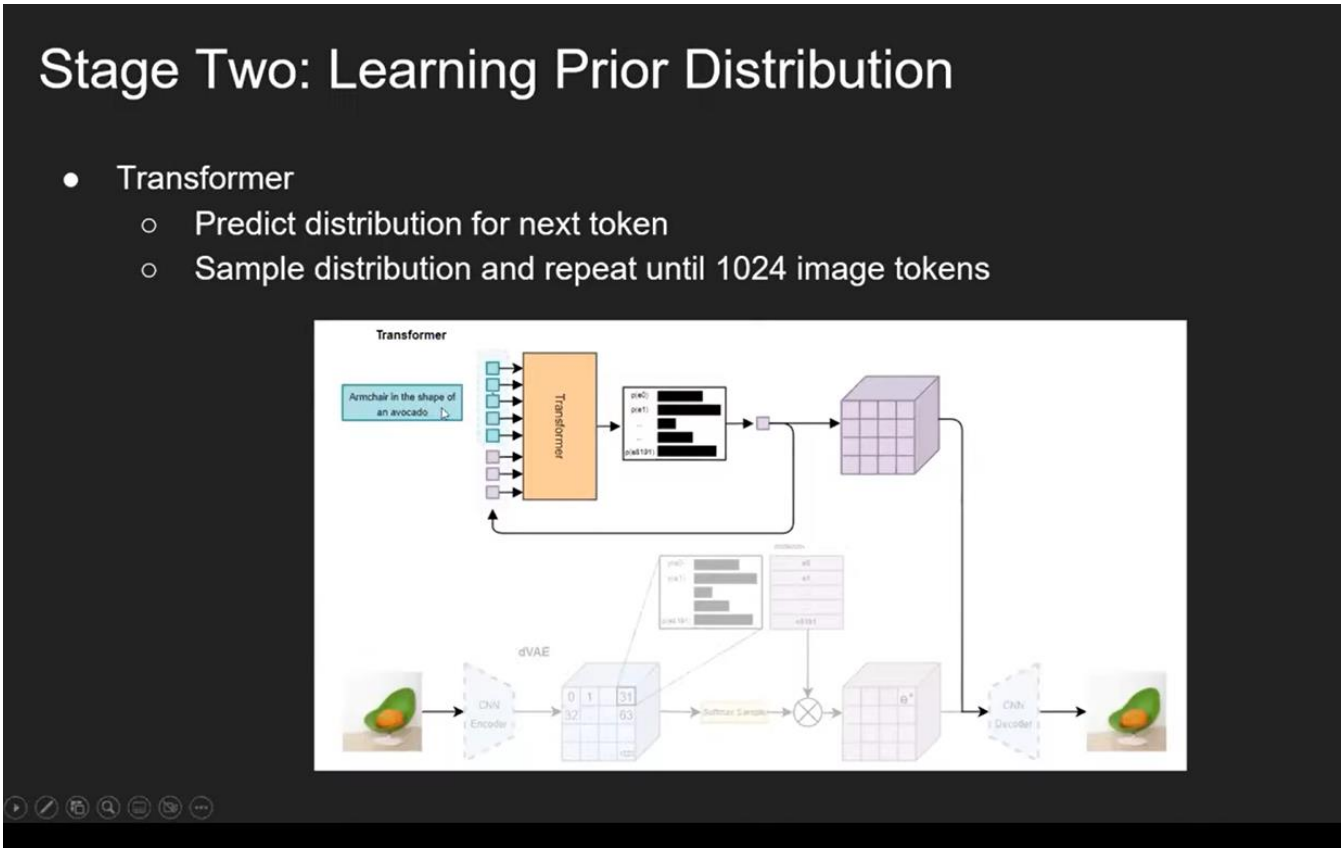
dVAE takes in an image and outputs categorical distributions over the set of codebook vectors for each latent

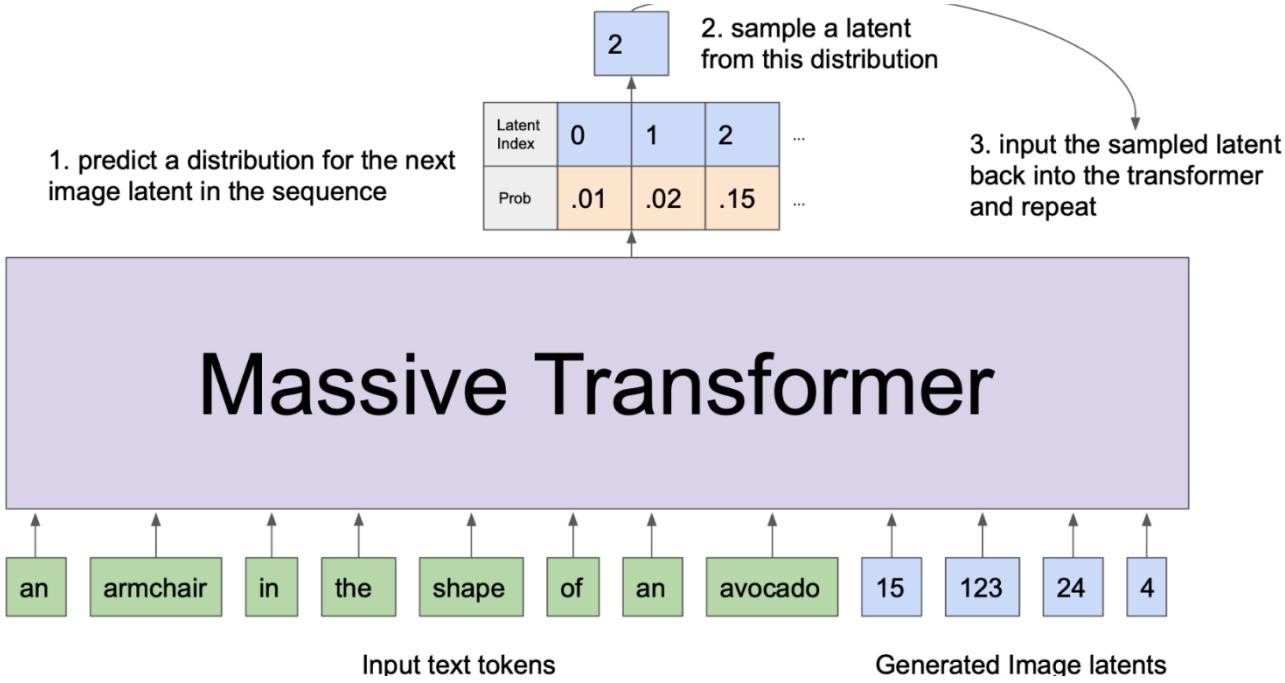


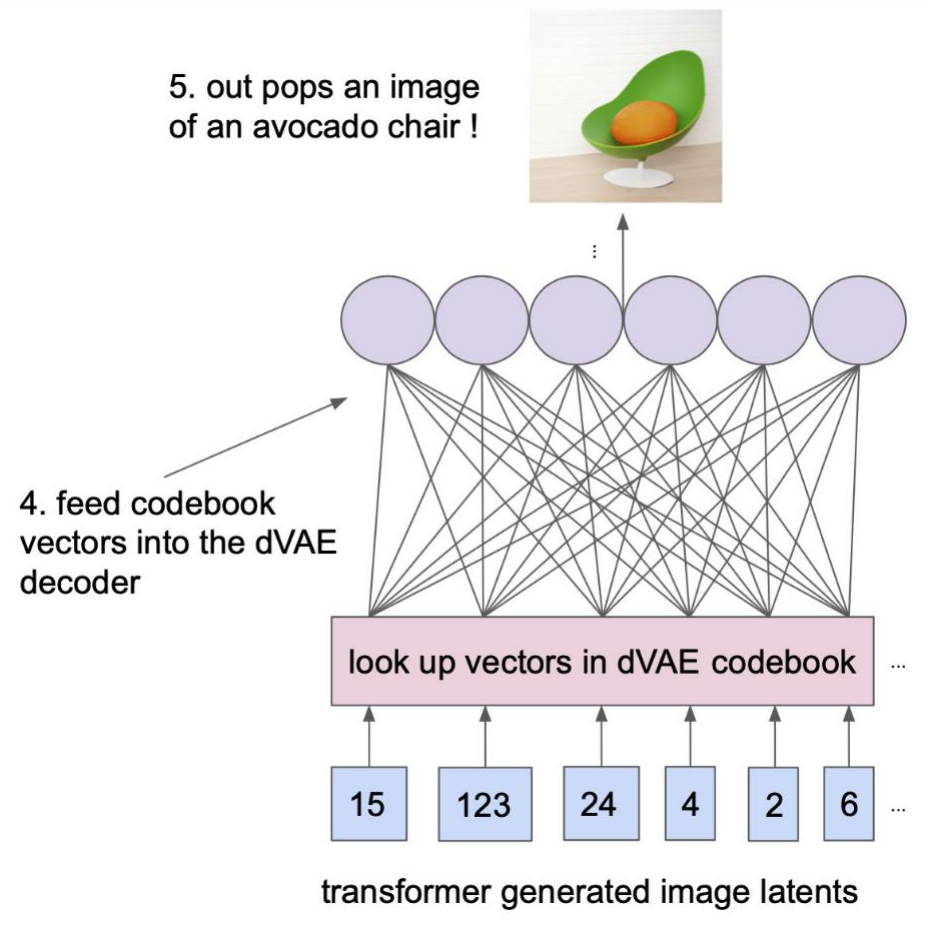


sampling codebook vectors from the Gumbel softmax distribution and then passing them to the decoder to reconstruct the original, encoded image

Stage 2:


Stage Two: Learning Prior Distribution
Transformer
Predict distribution for next token
Sample distribution and repeat until 1024 image tokens
Transformer
Armchair in the shape of an avocado
Transformer
dVAE
CNN Encoder
CNN Decoder


2
2. sample a latent from this distribution
Latent Index
0
1
2
...
Prob
.01
.02
.15
...
1. predict a distribution for the next image latent in the sequence
3. input the sampled latent back into the transformer and repeat
Massive Transformer
an
armchair
in
the
shape
of
an
avocado
15
123
24
4
Input text tokens
Generated Image latents


5. out pops an image of an avocado chair !
4. feed codebook vectors into the dVAE decoder
look up vectors in dVAE codebook
...
15
123
24
4
2
6
...
transformer generated image latents

Setup

Input

1. **Text Prompt (Input to Transformer)**:
   - Example: `"A blue cat sitting on a red chair"`
   - Tokenized into **text tokens**:
     `["A", "blue", "cat", "sitting", "on", "a", "red", "chair"]`
     Represented as indices: `[10, 201, 523, 78, 34, 10, 312, 442]`.
2. **Discrete Latent Tokens (Target from dVAE)**:
   - The image corresponding to the prompt is encoded by the **dVAE encoder** into a sequence of **latent tokens**.
   - Example latent tokens for a 4×4 grid:
     `[1002, 503, 202, 88, 762, 304, 612, 400, 230, 703, 944, 129, 876, 321, 402, 511]`.
3. **Vocabulary/Codebook Size**:
   - Codebook size: 8192 (possible indices for latent tokens).

Step-by-Step Example

Step 1: Combine Input Tokens

- The text tokens `[10, 201, 523, 78, 34, 10, 312, 442]` and latent tokens `[1002, 503, 202, ...]` are concatenated to form the **input sequence** for the transformer.
- Input sequence: `[10, 201, 523, 78, 34, 10, 312, 442, <latent_start>]`
  - `<latent_start>` is a special token to indicate the start of latent token generation.

Step 2: Transformer Outputs

- The transformer is trained to **predict each latent token** autoregressively, given the previous tokens in the sequence.
- For each position in the latent sequence, the transformer outputs a **probability distribution** over the entire vocabulary (8192 latent tokens).

Example output (for the first latent token position):

Codebook Index	1002	503	202	88	...	8191
Predicted Prob	0.4	0.2	0.1	0.05	...	0.0001

Step 3: Categorical Cross-Entropy Loss

For each latent token position, the **categorical cross-entropy loss** is computed:

$$\text{Loss} = -\log(\hat{y}_{\text{true}}),$$

- Example for the first latent token (`1002`):
  - True token index: 1002.
  - Predicted probability for 1002: 0.4.
  - Loss:

$$\text{Loss}_{1002} = -\log(0.4) = 0.916.$$

- The process repeats for all 16 latent tokens, and the losses are summed/averaged.

Step 4: Final Latent Tokens

During inference (generation), the transformer generates latent tokens sequentially:

- Start with text tokens + `<latent_start>` .
- Predict the first latent token: `1002` .
- Feed it back and predict the next token: `503` , and so on.

The final sequence of latent tokens corresponds to the **reconstructed image**.

Summary of Output

1. **Predicted Latent Tokens:**
   - Example sequence: `[1002, 503, 202, 88, 762, 304, 612, 400, ...]`.
2. **Loss:**
   - Computed for each latent token position based on predicted probabilities and true tokens.
3. **Reconstructed Image:**
   - The sequence of tokens is fed into the **dVAE decoder** to reconstruct the image.

- ...

;;;;;;;#pseudocode

STAGE2:

Here's an explanation of the Stage 2 training pseudo code for a model like DALL-E, broken into detailed steps:

Objective

In Stage 2, the goal is to train a transformer model to autoregressively generate the sequence of latent tokens for an image based on input text tokens. The training involves:

1. **Learning the prior distribution of latent tokens conditioned on text.**
2. **Using teacher forcing (feeding ground truth) during training to guide learning.**

;;;;;;;#pseudocode

STAGE2:

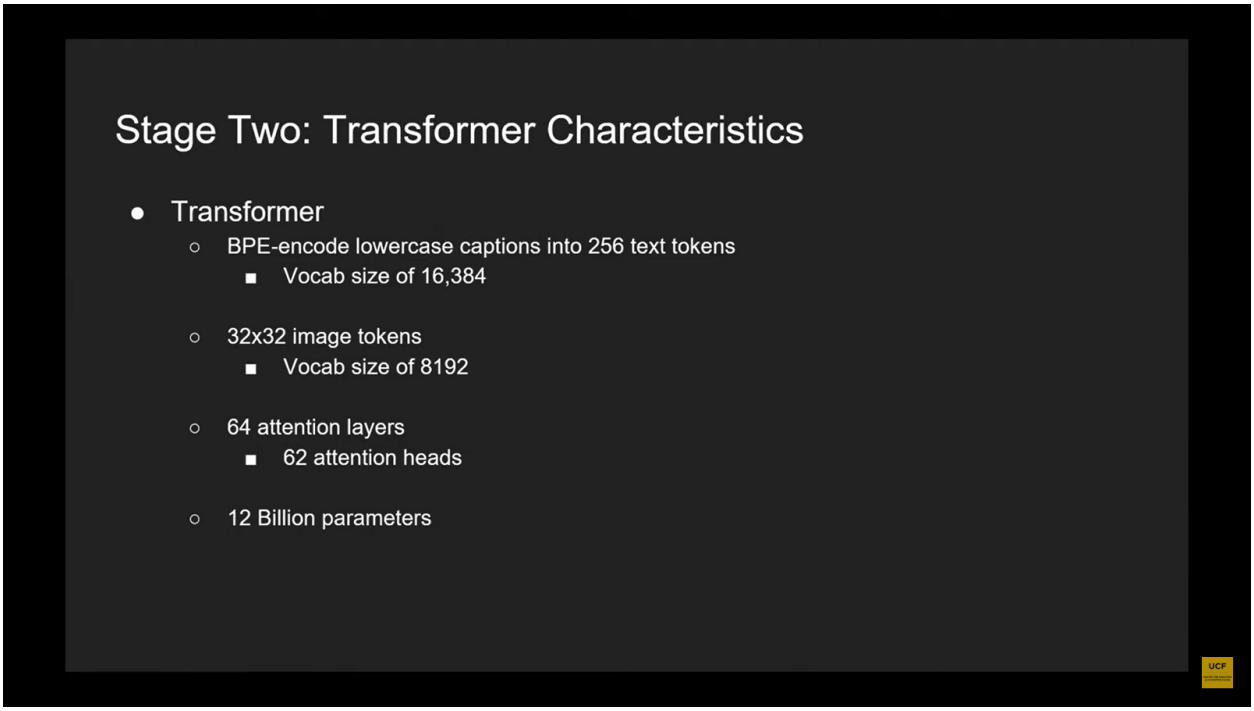
```
# Step 1: Initialize inputs
text_tokens = [10, 201, 523, 78, 34]   # Example text tokens
latent_tokens = [1002, 503, 202, 88, 1245]  # dVAE latent indices
latent_start_token = '<latent_start>'  # Special token indicating start of latent sequence

# Append the start token to text tokens
input_sequence = text_tokens + [latent_start_token]

# Prepare the ground truth (target latent sequence)
target_sequence = latent_tokens

# Initialize cumulative loss
cumulative_loss = 0

# Step 2: Autoregressive training loop
for step in range(len(latent_tokens)):
    # Step 2.1: Pass the input sequence through the Transformer
    predicted_logits = transformer_model(input_sequence)

    # Step 2.2: Extract the logits for the current prediction step // find the index for max prob
    # Predicted_logits shape: [sequence_length, vocab_size]
    # We only need the logits for the next token in the sequence
    current_logits = predicted_logits[len(input_sequence) - 1]  # Get logits for current step

    # Step 2.3: Compute cross-entropy loss
    ground_truth_token = latent_tokens[step]  # Actual token for this step
    loss = loss_function(current_logits, ground_truth_token)

    # Step 2.4: Accumulate loss
    cumulative_loss += loss

    # Step 2.5: Update input sequence with the current ground truth token
    input_sequence.append(ground_truth_token)  # Use ground truth during training (teacher forcing)

# Step 3: Backpropagation
cumulative_loss.backward()  # Compute gradients
optimizer.step()            # Update Transformer weights
```

NOTE:

 Stage 1 (Training dVAE): Gumbel-Softmax is used for smooth, differentiable learning.

 Stage 1 (Post-Training): Hard quantization maps encoder outputs to discrete codebook indices.

 Stage 2 (Transformer): These discrete codebook indices are used as latent tokens for autoregressive training.



- Text captions provided as input are encoded using **Byte Pair Encoding (BPE)**
- Each text caption is encoded into a fixed sequence of **256 tokens**
- The vocabulary size of the BPE tokenizer is **16,384** unique tokens. This ensures a rich representation of natural language while maintaining computational efficiency.

 32×32 image tokens":

- Each image is processed into a latent space representation by the **dVAE encoder** from Stage 1. The result is a **32×32 grid of latent tokens**, representing an abstract and compact description of the image.

- These latent tokens act as a sequence that the Transformer model uses during training or image generation.

 "Vocab size of 8192":

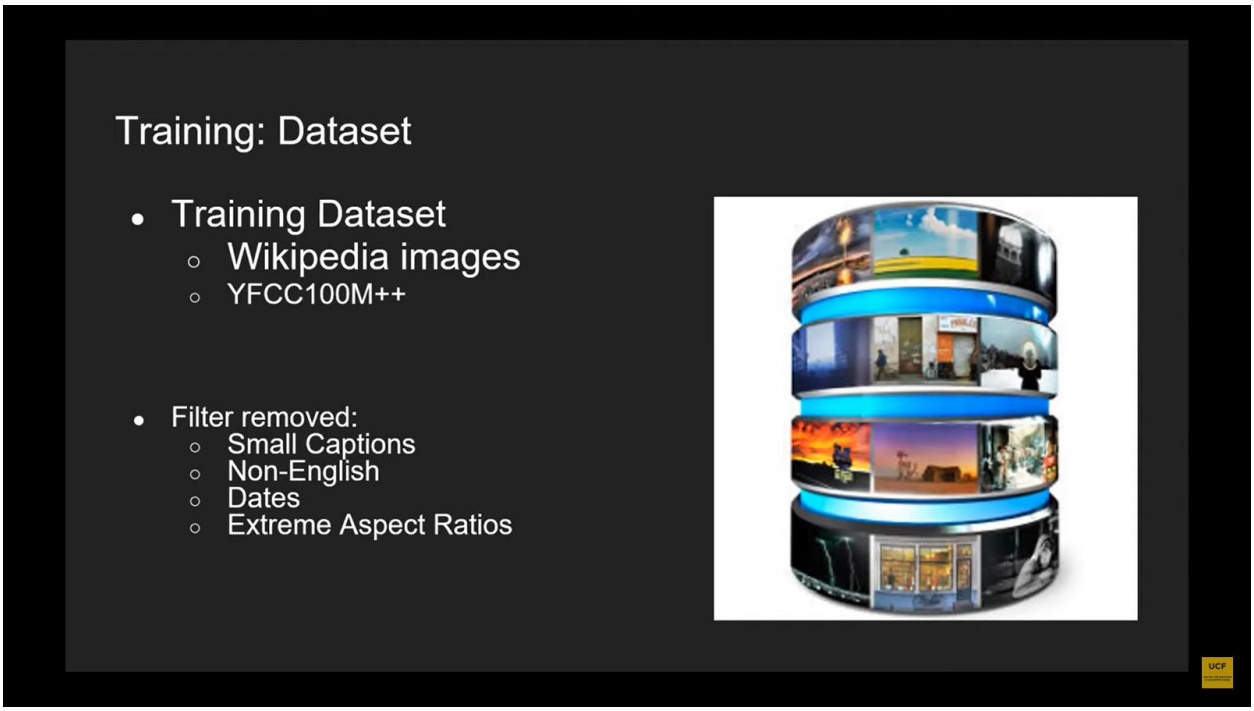
- The latent space is discretized into 8,192 distinct **codebook vectors** (as learned by the dVAE in Stage 1). Each latent token is represented as an index within this vocabulary.

Transformer Architecture:

- **"64 attention layers":**
    - The Transformer model is a deep architecture with **64 layers** of **self-attention and feed-forward networks**. This enables the model to learn complex interactions between text and image tokens over long sequences.
- **"62 attention heads":**
    - Each attention layer uses **62 parallel attention heads** to capture diverse relationships within and across text and image token sequences.

4. Model Size:

- **"12 Billion parameters":**
    - The model has a total of **12 billion learnable parameters**, which include weights for attention layers, feed-forward networks, embedding layers, and more.
    - This massive parameter count enables the Transformer to learn a rich and nuanced mapping between text descriptions and image representations, but it also demands significant computational resources during training and inference



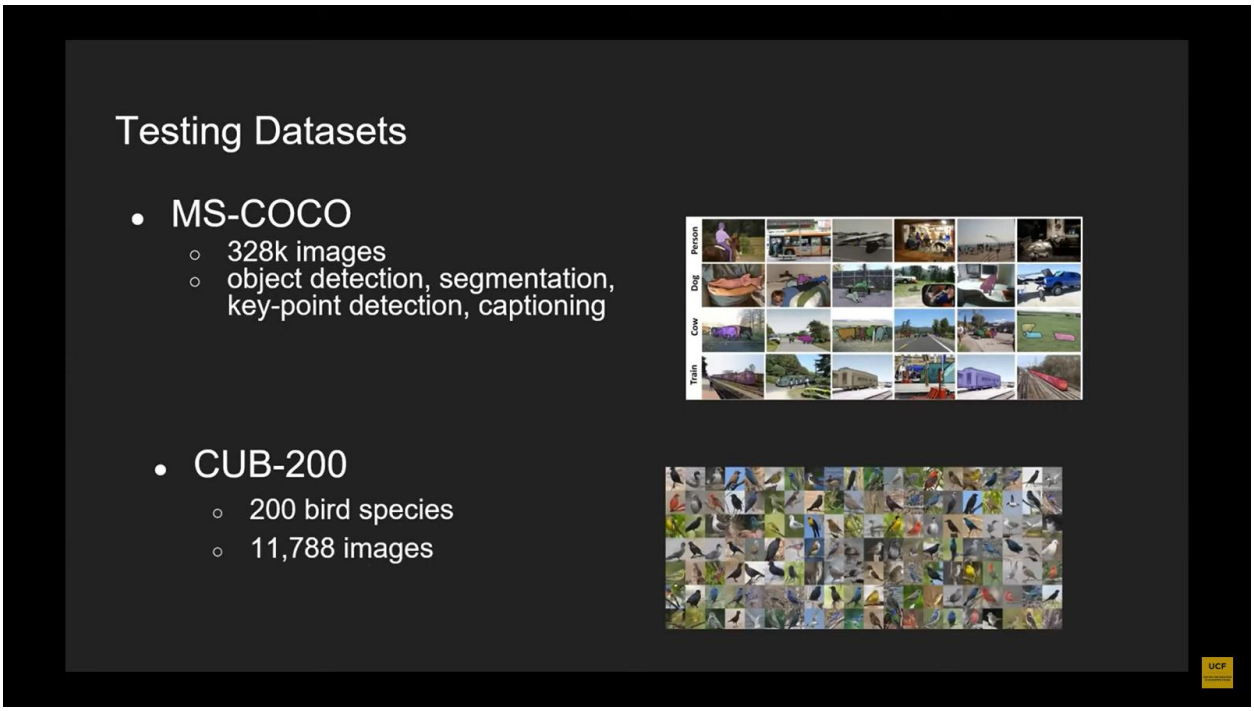
Training Dataset

- The datasets used for training include:
    - **Wikipedia images**: Images collected from Wikipedia, which are usually paired with rich textual information (captions or articles).
    - **YFCC100M++**: This refers to an extended or filtered version of the Yahoo Flickr Creative Commons 100M dataset, a large-scale dataset containing images and videos with associated metadata (such as descriptions, tags, and timestamps).

2. Data Filtering

Certain types of data were removed to ensure high-quality and meaningful training samples. Filters include:

- **Small Captions**: Text captions with very few words were excluded, as they may lack context or relevance to the image.
- **Non-English**: Non-English captions were removed to ensure language consistency for training.
- **Dates**: Captions or metadata that primarily consisted of dates (e.g., "2023-01-01") were filtered out, as they may not provide meaningful semantic content.
- **Extreme Aspect Ratios**: Images with very unusual aspect ratios (e.g., too wide or too tall) were excluded, as these could distort the learning process or reduce model performance.



1. MS-COCO (Microsoft Common Objects in Context)

- **Dataset Size**: Contains **328k images**, which makes it a large-scale dataset for diverse visual tasks.
- **Tasks**:
    - **Object Detection**: Identifying and classifying multiple objects in an image with bounding box annotations.
    - **Segmentation**: Fine-grained pixel-level labeling of objects (e.g., separating objects from the background).
    - **Key-point Detection**: Detecting key body points for pose estimation (e.g., human joints).
    - **Captioning**: Generating descriptive captions for images, linking vision with natural language.
- **Image Example (Top Right)**: Displays samples from the dataset, including categories like "train," "dog," "person," and more, illustrating its diversity and richness for real-world applications.

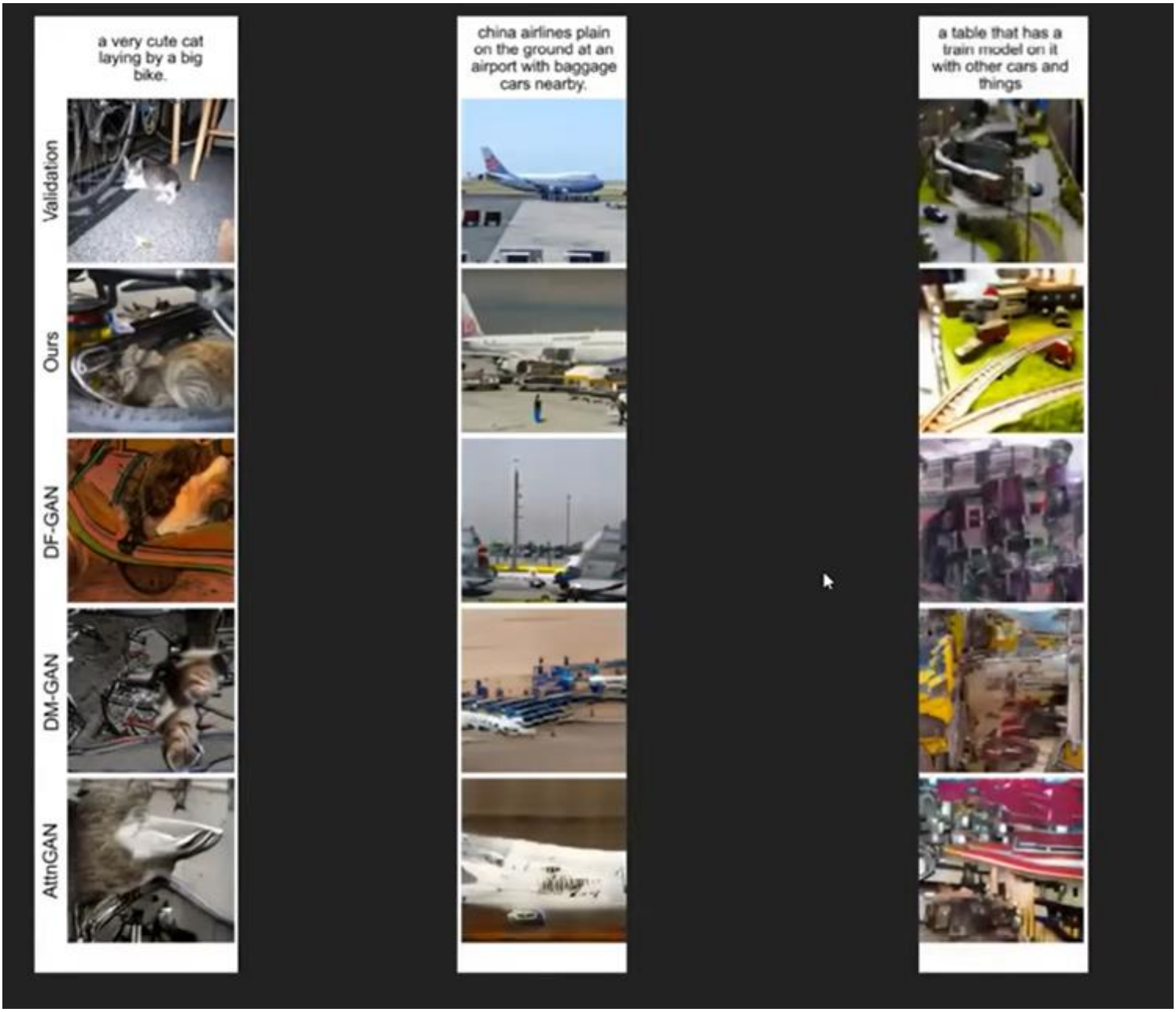
---

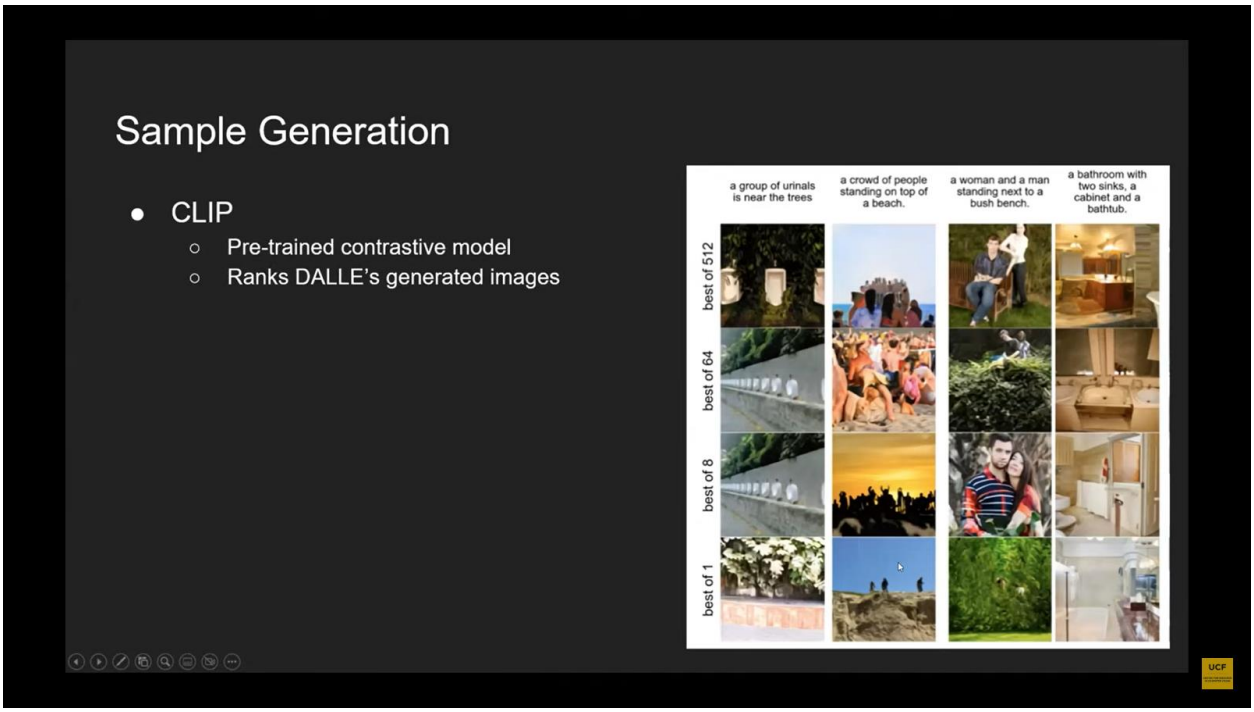
2. CUB-200 (Caltech-UCSD Birds 200)

- **Focus**: A specialized dataset focused on fine-grained bird species classification.
- **Dataset Size**:
    - **200 Bird Species**: Includes detailed annotations for different bird species.

    - **11,788 Images**: Comprehensively covers variations within species (e.g., different poses, lighting conditions).

- **Task**: Primarily used for fine-grained image recognition and classification tasks, where models are evaluated based on their ability to differentiate subtle differences between similar-looking classes (e.g., different bird species).
- **Image Example (Bottom Right)**: Shows a grid of bird images illustrating the dataset's focus on diverse species and scenarios.





Clip Is the matric that is used for evaluation of the models

Implementation: https://github.com/borisdayma/dalle-mini?utm_source=chatgpt.com