Stable Diffusion

CLIP is used for text encoding.

CLIP algorithm explanation

```
# image_encoder - ResNet or Vision Transformer
# text_encoder  - CBOW or Text Transformer
# I[n, h, w, c] - minibatch of aligned images
# T[n, l]       - minibatch of aligned texts
# W_i[d_i, d_e] - learned proj of image to embed
# W_t[d_t, d_e] - learned proj of text to embed
# t             - learned temperature parameter

# extract feature representations of each modality
I_f = image_encoder(I) #[n, d_i]
T_f = text_encoder(T)  #[n, d_t]

# joint multimodal embedding [n, d_e]
I_e = l2_normalize(np.dot(I_f, W_i), axis=1)
T_e = l2_normalize(np.dot(T_f, W_t), axis=1)

# scaled pairwise cosine similarities [n, n]
logits = np.dot(I_e, T_e.T) * np.exp(t)

# symmetric loss function
labels = np.arange(n)
loss_i = cross_entropy_loss(logits, labels, axis=0)
loss_t = cross_entropy_loss(logits, labels, axis=1)
loss   = (loss_i + loss_t)/2
```

Figure 3. Numpy-like pseudocode for the core of an implementation of CLIP.

This pseudocode is a simplified core of **CLIP** (Contrastive Language–Image Pretraining), where the goal is to **align images and texts** in a shared embedding space.
At a high level:

- **Images** and **Texts** get encoded separately into feature vectors.
- These feature vectors are projected into a **joint embedding space**.
- Then, they are **compared** using **cosine similarity**.
- A **contrastive loss** is applied, trying to **pull together** matching image-text pairs and **push apart** mismatched ones.

Now, let's go step-by-step:

---

1. Inputs

- `I[n, h, w, c]` : a minibatch of **n images** of shape height × width × channels.
- `T[n, l]` : a minibatch of **n texts** (aligned, meaning *i-th image matches i-th text*).
- `W_i[d_i, d_e]` : a learned linear projection from **image features** to **embedding space**.
- `W_t[d_t, d_e]` : a learned linear projection from **text features** to **embedding space**.
- `t` : a **learned temperature parameter** (controls the sharpness of similarities).

---

2. Feature Extraction

python
CopyEdit
```
I_f = image_encoder(I)  # [n, d_i]
T_f = text_encoder(T)   # [n, d_t]
```

- The **image encoder** (like ResNet or ViT) outputs features of dimension `d_i`.
- The **text encoder** (like CBOW or Transformer) outputs features of dimension `d_t`.

---

3. Project into Joint Embedding Space

python
CopyEdit
```
I_e = l2_normalize(np.dot(I_f, W_i), axis=1)
T_e = l2_normalize(np.dot(T_f, W_t), axis=1)
```

- Multiply by learned projections `W_i` and `W_t` to map into a common space of dimension `d_e`.
- **L2 normalization** makes embeddings **unit vectors** (important because cosine similarity becomes simple dot product).

---

4. Compute Similarities

python
CopyEdit
```
logits = np.dot(I_e, T_e.T) * np.exp(t)
```

- Dot product between all **image embeddings** and all **text embeddings**.
- Result is a **[n x n] similarity matrix**: each entry `[i,j]` is the similarity between image *i* and text *j*.
- Multiply by `exp(t)`, where `t` is learnable. Higher `t` → sharper softmax → harder contrastive learning.

---

5. Define the Labels

python
CopyEdit
```
labels = np.arange(n)
```

- This just says: for each sample *i*, the correct match is at position *i* (since images and texts are aligned).

---

6. Loss Computation

Here's the key part:

python
CopyEdit
```
loss_i = cross_entropy_loss(logits, labels, axis=0)
loss_t = cross_entropy_loss(logits, labels, axis=1)
loss   = (loss_i + loss_t)/2
```

- `loss_i`:
  - For each **image**, treat its similarities to all texts as a softmax classification.
  - It should "predict" the correct text among all texts.
- `loss_t`:
  - For each **text**, treat its similarities to all images as a softmax classification.
  - It should "predict" the correct image among all images.

✅ **Take the average of the two**: make sure both image-to-text and text-to-image alignments are good.

Why is this loss effective?

- It **pulls** corresponding image-text pairs closer together in the embedding space.
- It **pushes** non-matching pairs apart.
- It does **both directions**: *image → text* and *text → image*.
- Learning the **temperature** `t` allows it to dynamically adjust the difficulty.

Quick Intuition

If you have an image of a dog and a caption "a cute dog":

- The similarity between (dog image, "a cute dog") should be high.
- The similarity between (dog image, "a bowl of ramen") should be low.
- And vice-versa from text → image.

What is a Fixed Markov Chain

📖 First, what is a Markov chain?

- A **Markov chain** is a sequence of random steps where:
  - The next state depends **only** on the **current** state (not the whole past).
- **Mathematically**:
  $P(xt+1 \mid x0,x1,\ldots,xt)=P(xt+1 \mid xt)P(xt+1 \mid x0,x1,\ldots,xt)=P(xt+1 \mid xt)$
- This is called the **Markov property**: *"memoryless" process*.

You can imagine it as **moving through a set of states** step by step, according to some **fixed transition probabilities**.

🧩 Now, what is a fixed Markov chain with T steps?

Fixed:

- The **transition rule** (the probabilities of moving from one state to another) **does not change** over time.

T steps:

- You run the chain for **T time steps** (then stop).

🎯 In plain English:

> A **fixed Markov chain with T steps** is a random process where you move between states according to the same fixed rules at every step, for exactly **T moves**.

At each time t=0,1,2,...,T−1t=0,1,2,...,T−1, you apply the same transition behavior, starting from some initial state.

After T steps, you stop.

📈 Tiny Example

Imagine a 2-state Markov chain:

States:

- 0 = "Rainy"
- 1 = "Sunny"

Transition probabilities:

From / To	Rainy (0)	Sunny (1)

Rainy	0.7	0.3
Sunny	0.4	0.6

- If it's rainy today, 70% chance rainy tomorrow.
- If it's sunny today, 60% chance sunny tomorrow.

Suppose T = 3.
You start Rainy (state 0):

- Step 1: pick next state based on Rainy row `[0.7, 0.3]`
- Step 2: pick next state based on new state's row
- Step 3: pick next state again

That's **T = 3 steps**, all using the same table.

✅ The **transition rule is fixed** across steps.

🔥 Why does this matter?

- Many models (like **diffusion models** in machine learning) use **fixed Markov chains** to gradually corrupt data (forward process) or denoise it (reverse process).
- It's mathematically clean to **reason about the evolution** of distributions when transitions are fixed.
- If T is large, you can sometimes reach a **stationary distribution** (the system forgets its starting point).

📜 Summary

Term	Meaning
Markov chain	Process where next state depends only on current
Fixed	Transition probabilities do not change over time

T steps	Process runs for exactly T time steps, no more

Prompt Integration with U-net

Let's go through **how Stable Diffusion uses cross-attention to integrate text into the U-Net** carefully.

🧠 Big Idea

In **Stable Diffusion**, the **U-Net** doesn't just denoise blindly.

Instead, **at every step** of denoising, it is **guided by the text prompt**.
This happens via **Cross-Attention** inside the U-Net.

✅ Cross-Attention allows the U-Net to **"look at" the text embeddings** while deciding **how to denoise**.

🎯 Where exactly is Cross-Attention used?

- Inside the **middle and up-sampling blocks** of the U-Net.
- After certain convolution layers, there are **self-attention blocks replaced or enhanced with cross-attention**blocks.

So the denoising is **conditioned** on the text **throughout the process**, not just at the start.

🛠️ How Cross-Attention Works in the U-Net

At a high level:

Component	What it is
Query (Q)	Comes from the current U-Net feature maps (the noisy image features)
Key (K)	Comes from the **text embeddings** (from the text encoder)
Value (V)	Comes from the **text embeddings** (same as keys)

The cross-attention computes:

Attention(Q,K,V)=softmax(QKTdk)VAttention(Q,K,V)=softmax(dkQKT)V

✅ So, the **image features "query" the text embeddings** to decide what to pay attention to!

✨ Why Cross-Attention Matters

Imagine you're trying to **denoise an image towards "a cat wearing sunglasses"**:

- If the U-Net only looks at noise, it might not know *which direction* to denoise.
- But with cross-attention, it can **look at** the text "cat" and "sunglasses" words, **attend to them**, and guide the denoising **toward that meaning**.

Thus, cross-attention **injects semantic meaning into the generation process at every level**.

📜 More Formally

Suppose:

- Current U-Net feature map: $x \in \mathbb{R}^{n \times d_x}$ (n tokens, d_x dim)
- Text embeddings: $c \in \mathbb{R}^{m \times d_c}$ (m tokens, d_c dim)

Steps:

1. Project x into queries $Q = xW_Q$
2. Project c into keys and values: $K = cW_K, \quad V = cW_V$
3. Compute attention scores $A = \text{softmax}(QK^T / \sqrt{d_k})$
4. Output: $A \times V$

This output is then added back into the U-Net feature maps, influencing the next steps of processing.

✅ **Result**: Denoising is text-guided **at every spatial location**.

🖼️ Tiny Flow Chart

Inside U-Net Block:

mathematica
CopyEdit

```
Noisy Feature Maps (Query)   --->  Cross Attention  ---> Updated
Feature Maps
Text Embedding (Key & Value)
```

🚀 TL;DR

- U-Net feature maps act as **queries**.
- Text embeddings act as **keys** and **values**.
- **Cross-attention** lets the U-Net *peek* at the text and *decide* how to denoise better.
- This is **how text conditioning happens** in Stable Diffusion during image generation!

Stable Diffusion Use-cases and Architectural Adjustments

1. 📈 Image Super-Resolution

(Make a blurry or low-res image sharper and higher-res)

✅ **Main idea:**
Instead of starting from random noise, you **condition the U-Net on a low-resolution image**.

Architecture Changes:

- **Input:** Instead of pure noise xTxT, you start with a **low-res noisy version** of the image.
- **Conditioning:** Provide the **low-resolution image as an extra input** along with noise and timestep.
- Sometimes the low-res image is **concatenated** with the noisy latent as extra channels, or passed via cross-attention.
- The U-Net learns to "refine" the blurry image progressively.

In short:
Low-res image is treated **like a guiding prompt**, similar to how text was used.

---

2. 🎨 Style Transfer

(Apply the style of one image onto another)

✅ **Main idea:**
You condition the diffusion model **on two things**:

- The **content image** (what you want to preserve)
- The **style image** (what artistic style you want to apply)

Architecture Changes:

- **Conditioning:** Both the content and style information are encoded.
    - Content image could be injected into U-Net feature maps.
    - Style information could influence cross-attention layers.
- **Objective:** Generate an image that **matches content features** of the content image and **style features** of the style image.

- Sometimes **two encoders** are used:
    - One for content, one for style.

In short:
You combine **dual conditioning** inside the U-Net: content + style.

---

3. 🎯 Inpainting

(Fill missing parts of an image, e.g., remove object and refill background)

✅ **Main idea:**
You give the model:

- A **partially masked image**
- A **mask** that tells which regions are missing

Architecture Changes:

- **Input:** Instead of only random noise, the input includes:
    - Noisy image + mask
- **Conditioning:**
    - Masked regions are where model needs to "invent" data.
    - Unmasked regions (known parts) are copied from the input.
- **Training:** During training, you randomly mask patches in images so the model learns to inpaint missing regions.

Typical implementation:

- **Concatenate** the image + mask as input channels.
- Model predicts denoised latents conditioned on visible context.

In short:
Mask + visible parts tell the U-Net *where* to "hallucinate" new pixels.

---

4. 🌌 Outpainting

(Extend an image beyond its original boundaries)

✅ **Main idea:**
Similar to inpainting, but **now you add new empty regions around the image**, and the model **hallucinates outward**.

Architecture Changes:

- **Input:** Image placed in the center of a larger canvas, with surrounding regions as masked (empty).
- **Conditioning:**
    - Original image guides the newly generated parts.
    - Empty surrounding regions are where the U-Net needs to generate new content.
- Sometimes the **text prompt** is also still used, to control what the extended scene should contain.

In short:
Exactly like inpainting, but the **mask** is applied around the original image instead of within.

🧠 Common Pattern Across All These Tasks

Aspect	How it Changes
Input	Not just random noise — it's noise + guiding context (text, images, masks)
Conditioning	U-Net is modified to accept extra conditions, either via concatenation or cross-attention
Training	Needs slightly modified tasks (e.g., masked training for inpainting)
Loss	Often still predict noise (like in basic diffusion), but may add extra perceptual or style losses

✅ **The U-Net remains the core engine**,
✅ **but conditioning changes depending on the task**.

🚀 TL;DR Table

Use Case	How U-Net Differs
Super-Resolution	Condition on low-res image

Style Transfer	Condition on content + style encodings
Inpainting	Condition on masked image + mask
Outpainting	Same as inpainting but mask surrounds the image