EEE 443 - Final Project
Text-to-Image Generation

Onur Ünlü
Electrical & Electronics Engineering
Bilkent University
onur.unlu@ug.bilkent.edu.tr

Yiğit Uz
Electrical & Electronics Engineering
Bilkent University
yigit.uz@ug.bilkent.edu.tr

Ş. Taha Kiremitçi
Electrical & Electronics Engineering
Bilkent University
taha.kiremitci@ug.bilkent.edu.tr

Doğa Diren
Electrical & Electronics Engineering
Bilkent University
doga.diren@ug.bilkent.edu.tr

Abstract

In this project, the task is the classical text-to-image synthesis. We implement several different architectures to achieve a satisfactory text-to-image model. We use CLIP model as a text and image encoder. Implemented models are CLIP-GLASS, VQGAN+CLIP, CLIP Guided Diffusion, Quick CLIP Guided Diffusion, Stable Diffusion, and Imagen. Approaches, and the overviews of each model is provided in the report. Furthermore, development process of each model is explained including the challenges face during the development process. The report ends with the outputs of the models and their comparisons. Possible future improvements are also discussed in the conclusion.

1 Introduction

The ability to visualize and understand the underlying relationship between the visual world and language was deemed to be an ability exclusive to humans. Inspired by how humans can visualize their languages, building a machine learning model with similar capabilities have been a major goal aiming for human-like intelligence. However, until the rise of deep learning such a task seemed intractable and much of the focus was on the reverse task of extracting captions from given images. This changed with the advent of Generative Adversarial Networks (GANs) [1] which made it possible to train generative models for images in a completely unsupervised manner. After the introduction of GANs there was surge of of interest and advanced research efforts in image synthesis tasks. Main idea in a GAN was to frame the image synthesis task an adversarial game of between two networks. A generator network was trained to produce realistic samples whereas the discriminator was trained to distinguish between real and generated images. This approach has led to many successful applications such as high-resolution synthesis of human faces [2], image superresolution [3], image in-painting [4], [5], data augmentation [6], style transfer [7],[8], image-to-image translation [9], [10] and representation learning [11], [12]. Further developments in GAN architectures then lead to conditional image synthesis which pawed the way for successful text-to-image generation models [13]. Omitting many different novel GAN architectures that were regarded as state-of-the-art (SOTA) models in text-to-image generation like DC-GAN[14], MC-GAN[15], StackGAN(stacked generative adversarial network) [16] one could argue that the most prevalent text-to-image model that uses GAN architecture today is the VQGAN+CLIP [17] which will be introduced later in section 2.2.3. In more recent years, GAN models started to get replaced by the surging diffusion models. Diffusion model is a parameterized Markov chain that gradually converts one distribution to another, first proposed in [18]. Improvements in diffusion models like Denoising Diffusion Probabilistic Models (DDPM) has

pawed the way for text-to-image generation models based on diffusion which is started to replace GAN models as the state-of-the-art models. Most recent architectures that are regarded as SOTA include DALL-E 2 [19], Imagen [20], Stable diffusion by Stability AI [21]. Both in diffusion and GAN models, a crucial part of the model is the text/image encoder, the most natural choice for an encoder is OpenAI's CLIP [22] model as it is specially trained to represent images and texts in the same representational space. In this project, we focus on DALL-E 2, CLIP guided diffusion, Stable Diffusion, VQGAN+CLIP, and CLIP-GLASS [23] to implement them in our own settings trying to reach or improve the performance of the open-source models.

Organization. The report is organized as follows. In the §2 we first describe and comment on the given data set in §2.1. In §2.2, we explain the models used in the project. In §2.3, we explain the development process of the models. We conclude the report with the generations from models in §3 and discussions in §4. Appendix includes the codes and some exemplary loss curves.

2 Methods

2.1 Data and Pre-Processing

Provided data set is designed for text-to-image synthesis task. Data set had the following components:

- train/test _cap: Captions for training/testing images. It has 17 indices from the vocabulary for each image. Some of the images have multiple captions.
- train/test_imid: The indices of training images. train/test _imid and train/test_cap has same number of rows. train/test _imid links caption indices to corresponding images. Briefly, it matches captions with images.
- train/test _url: Flickr URLs for training/testing images.
- word_code: A dictionary containing the actual words, which are shown by indices in train/test_cap.

Our observations about the dataset after investigating it:

- Nearly 15% of the URLs are not active at the moment. Thus, while downloading images, we forced a try-except method to get rid of errors and download only available images. This process took long time since dataset is already large (around 80,000 images) and try-except method takes time to process the conditions. Downloading process took around ~ 8 hours.
- Captions have special words such as "x_START", and "x_END" to indicate when does the wording sequence start and end. These captions are important because length of the sequences are not equal, but they are indiced with 17-length array. Thus these special words helps us indicate the length of the sentence. Another special word is "x_UNK". This special word indicates to an unknown word in the caption. We conjecture that the abundance of "x_UNK" words decreases the accuracy of the model.

Since we decided to use CLIP as text-image encoder the dataset was formatted in a manner fit to CLIP model. During our research, we found the format of the dataset which is understandable by CLIP [22]. Data loading process is all done using a custom data loader. Downloading, and preprocessing of the images are in the following order:

- Image URLs are read from the respecting file and each image is downloaded using a try-except methodology to bypass the errors occurring from broken links. Each image is saved as the last part of their URL given in the dataset.
- Caption IDs are mapped from the array-typed file to the dictionary to create a sentence. "x_UNK" keywords are included in the sentences as an indicator of an unknown.
- CLIP works best with a "JSONline" type file for training. Thus, we created the corresponding file which contained a dictionary with keys "URL" and "Captions". Values of respective keys are the image paths and list of captions that describe that image.
- Images vary in size. CLIP handles reshaping and normalization inside its algorithm, thus we did not need to handle it manually. However, some images were in 'CMYK' coding, therefore we converted them to 'RGB' before going on.

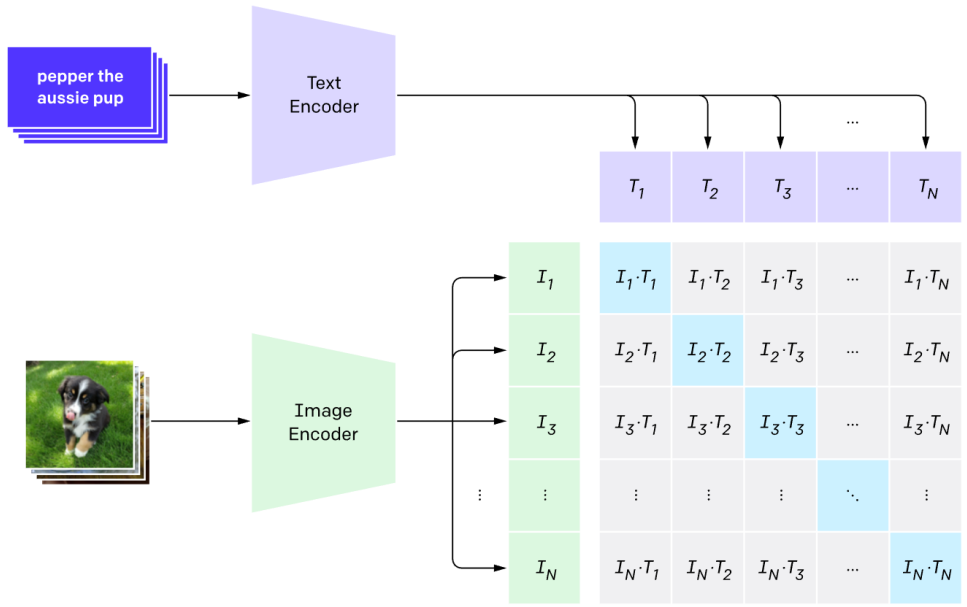


Figure 1: Summary of CLIP's approach. While standard image models jointly train an image feature extractor and a linear classifier to predict some label, CLIP jointly trains an image encoder and a text encoder to predict the correct pairings of a batch of (image, text) training examples [22]

2.2 Models

2.2.1 CLIP

CLIP (Contrastive Language-Image Pre-Training) is an open-source deep learning model that, simply put, aims to learn the relation between natural language and images. It was trained on an abundantly available data: the text paired with images found across the internet. One of the main properties of the model is its promise in zero-shot tasks. The model was especially trained to make it robust against unseen data. CLIP consists of an image and a text encoder and therefore can be used to embed both texts and images. Thus, it is a perfect model to use in a text-to-image generation task. We provide the CLIP's training setting to get an overview of model's function. Contrastive pre-training is performed as follows: model is given an input of N batches of image-text pairs, these inputs are embedded into $[I_1, I_2, ..., I_N], [T_1, T_2, ..., T_N]$ and then the model maximizes the cosine-similarities of the images and texts that are paired while minimizing the similarities of images and texts that are not paired, formally given by:

$$\max \sum_{i=1}^{N} T_i I_i + \min \sum_{i,j}^{N} T_i I_j \tag{1}$$

Note that, there are many CLIP configurations depending on the choice of model for image and text encoder. For the image encoder, Vision Transformers (ViT) [24] are used. On the other hand, the text encoder is a Transformer [25] with the architecture modifications described in [22]. Specifically it uses a 63M-parameter 12 layer 512-wide model with 8 attention heads. The transformer operates on a lower-cased byte pair encoding (BPE) representation of the text with a 49,408 vocab size [26] and the max sequence length is capped at 77. Figure 1 displays the overview of the CLIP model.

2.2.2 CLIP-GLASS

Having a text and image encoder like CLIP, the architecture in [23] is the most intuitive approach. In this model, we treat CLIP as a guide and do a latent space exploration over the latent space of a GAN to find the optimal z from latent space. We can model this as an optimization problem:

$$max_z sim\left(CLIP_I(G(z)), CLIP_T(T)\right) \tag{2}$$

where sim stands for cosine similarity operator, $CLIP_I, CLIP_T$ stands for image and text encoder of clip respectively, z is the vector from the latent space, G(z) is the output of the GAN, and T is the text input. To solve this optimization problem, one can use any optimizer but we follow [23] and utilize a genetic algorithm which is described further in 2.3.2. Another thing to note is the choice of GAN architecture to perform a latent space search on. In this project we use Deepmind's BigGAN architectures. Important aspect of choosing a GAN is to consider the latent space of it, for example Nvidia's StyleGAN trained over human face data set performs poorly in our setting as given text input may not describe a human face in which case even the optimal z will fail to generate a sensible image.

2.2.3 VQGAN+CLIP

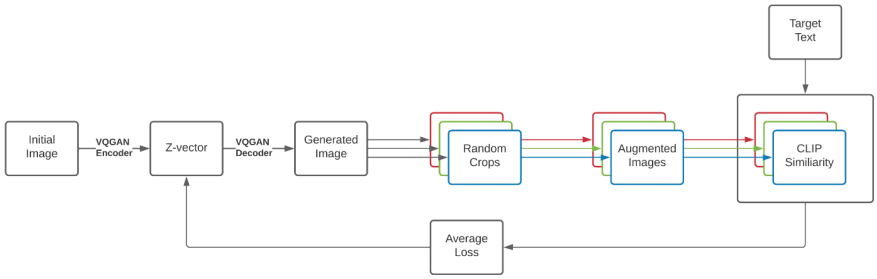


Figure 2: Diagram of optimization loop [17]. Although simplified, this diagram showcases the interaction between CLIP and VQGAN and how we use it for text-to-image generation task.

This model uses VQGAN and clip as pre-trained models. VQGAN-CLIP start training with a text caption and uses a GAN and produces images. Besides the GAN, it also uses CLIP to improve the similarity of generated image towards the actual image. Optimization method in VQGAN-CLIP is the spherical distance between embedding of the candidate and the embedding of the text caption as a loss function. It differentiates through CLIP with respect to GAN's latent vector representation of the image [17].

Image generation process start with a random valued "initial image". The optimization loop repeats itself to alter the image, until the generated image, which is the output image can be captioned by the target text semantically. While common sense is to use a random initial image, one can use an existing image as the "initial image". In this scenario, it is understood that the input image is desired to be edited. The choice of initial image does not change the output performance of the architecture [27].

Vector quantization is used in VQGAN-CLIP model (codebook in Figure 3). This is applied to the dataset using a convolutional encoder and decoder (E and G in Figure 3). First, input is embedded and in the latent space, the spherical distance among captions and images are searched. The embedding tries to optimize the distance between the real image and the fictional generated image (D in Figure 3). Guidance of the GAN is made through CLIP, which is a joint text-image encoder. CLIP embeds

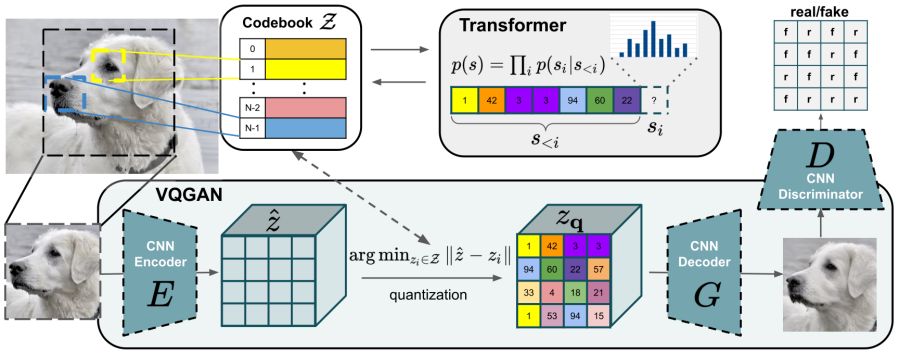


Figure 3: Architecture of VQGAN [28]. We defer the explanation to §2.2.3

captions and images independently and searches for cosine similarity between them. The similarity is reformed as a loss in training process [28].

2.2.4 CLIP Guided Diffusion

In CLIP guided diffusion model, the diffusion model is the Gaussian diffusion models introduced by [18] and improved by [29]; [30]. Given a sample from the data distribution $x_0 \sim q(x_0)$, a Markov chain of latent variables $x_1, ..., x_T$ is produced by progressively adding Gaussian noise to the sample:

$$q(x_t|x_{t-1}) := \mathcal{N}(x_t; \sqrt{\alpha_t}x_{t-1}, (1-\alpha_t)\mathcal{I}) \tag{3}$$

Under the assumption that the added noise at each time step is small and the total noise throughout the chain is large enough, x_T is well approximated by $\mathcal{N}(\prime, \mathcal{I})$. This suggests that $p_\theta(x_{t-1}|x_t)$ can be learned to approximate the true posterior:

$$p_\theta(x_t|x_{t-1}) := \mathcal{N}(\mu_\theta(x_t), \Sigma(x_t)) \tag{4}$$

We leave out the details on how to derive $\mu_\theta(x_t)$ & $\Sigma(x_t)$ and refer the reader to [30]. [31] showed that diffusion models can be improved with classifier guidance, where a class-conditional diffusion model with mean $\mu_\theta(x_t|y)$ and variance $\Sigma_\theta(x_t|y)$ is additively perturbed by the gradient of the log-probability $logp_\phi(y|x_t)$ of a target class y predicted by a classifier. The resulting new perturbed mean $\tilde{\mu}_\theta(x_t|y)$ is given by:

$$\tilde{\mu}_\theta(x_t|y) = \mu_\theta(x_t|y) + s\Sigma_\theta(x_t|y)\nabla_{x_t} logp_\phi(y|x_t) \tag{5}$$

In CLIP guided diffusion model, the idea is to use CLIP as the classifier in the class-conditional diffusion model. Again leaving out the details, the mean becomes following:

$$\tilde{\mu}_\theta(I_t|T) = \mu_\theta(I_t|T) + s\Sigma_\theta(I_t|T)\nabla_{I_t}(CLIP_I(I_t) \cdot CLIP_T(T)) \tag{6}$$

where I_t are the images at step t and T is the text input given to the model. In our project we use its implementation in [32] and the quick version of it developed by Daniel Russel whose github repository is currently not open.

2.2.5 Stable Diffusion

Stable diffusion is a text-to-image latent diffusion model. Its training dataset consists of 512x512 images. Generally, diffusion is trained to get rid of the noise, mainly the Gaussian noise, during their entire training process. At the end of this denoising process, main goal is to achieve the interested sample, which is an image in this case. Mostly, it is regarded as one of the SOTA models of image

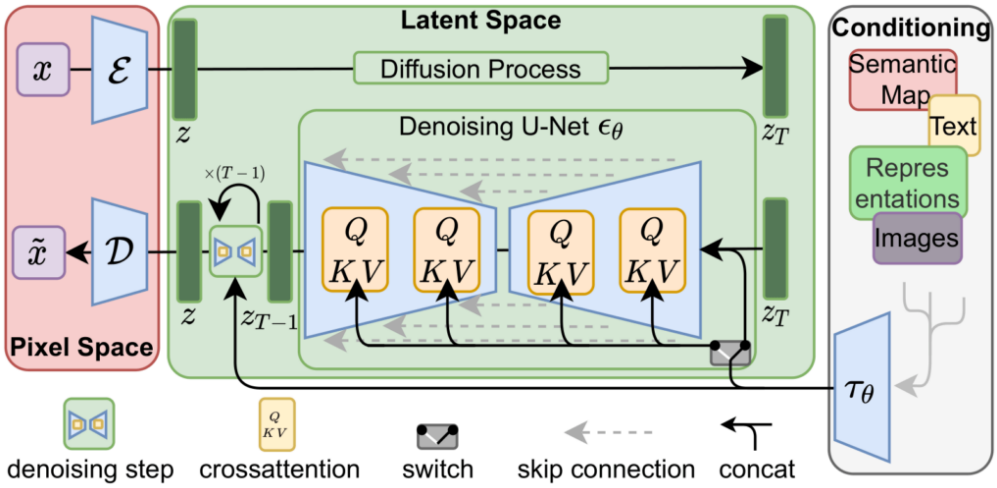


Figure 4: the architecture used in the stable diffusion model [33]. As explained in 2.2.5 text encoder used in our project is CLIP. This diagram displays the architecture consisting of U-Net, autoencoder, and CLIP. One can also see the cross-attention layers and the skipped connections.

generation. However, one downside of stable diffusion is that it is slow because of its repetitive nature. Another downside is that it needs too much memory because it attacks the image in pixel level [34], [35]. On the other hand, latent diffusion model, the diffusion model which our project relies on, is trained to generate a compressed version of the image. Latent diffusion models get rid of the downsides of traditional diffusion models. There are three main parts of latent diffusion models: autoencoder, U-Net, and text-encoder [34]. Autoencoder has two main parts, encoder, and decoder. The encoder part encodes the image to a low dimensional latent space, which will be the input to U-Net model. Decoder will perform a decoding to increase the dimension of the latent space elements and transform it into an image [36].

U-Net does the compression and decompression processes using encoders and decoders built inside it. Main goal of U-Net is to predict the noise residuals to ultimately predict the denoised image. U-Net performs down sampling while doing these steps. To prevent information loss, short-cuts between ResNet blocks (which are responsible from up sampling and down sampling) of U-Net are made. U-Net conditions its outputs using cross-attention layers, on text-embeddings. Cross-attention layers are added on both ends of the U-Net, encoder and the decoder [34]. Text encoder used in this model is the CLIP model. CLIP is used because it provides simultaneous embedding for words and images and project them on same latent space [22].

2.2.6 Imagen

Taking a slightly different approach, we use the most recent SOTA model, Imagen by Google [20]. We refer to it as a different approach solely due to it not using CLIP as text encoder whereas its diffusion based model is the same with other diffusion models. In place of CLIP, it uses T5 as the language model. One significant finding of Imagen was showing that large pre-trained frozen text

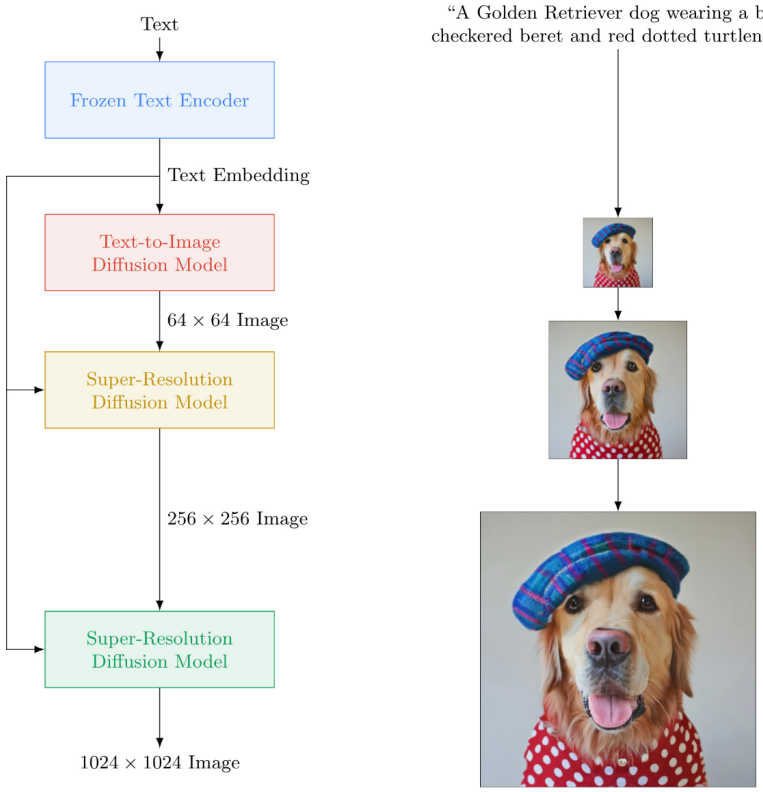


Figure 5: Visualization of Imagen. Imagen uses a large frozen T5-XXL encoder to encode the input text into embeddings. A conditional diffusion model maps the text embedding into a 64×64 image. Imagen further utilizes text-conditional super-resolution diffusion models to upsample the image 64×64→256×256 and 256×256→1024×1024. [20]

encoders are very effective for the text-to-image task. They prove it by comparing the model using multi-modal embedding CLIP model and outperforming it with T5. Omitting the technical details for conciseness, we provide an overview diagram in figure 5. Note that the Imagen model that is regarded as SOTA is using T5-XXL (4.6 billion parameters) as text encoder and rest of the model has 3 billion parameters. Furthermore, the trained model is not open sourced, therefore, we only reach a small portion of its promised performance as we cannot properly train such big models in our own setting.

2.3 Training

In this section, we summarize the development process of each model. Note that there are no loss curves reported in this section and some of the exemplary loss curves are deferred to the appendix for clear reading.

2.3.1 CLIP Training

Inspired by the findings of [20], we decided to use the open-source versions of the CLIP model. However, although CLIP has promising results for zero-shot performance, we fine-tuned it on our image-text data set to achieve better performance. After loading one of the pre-trained models, CLIP model was trained on our dataset using the cosine similarity defined in 1 as the objective. The code that we implemented to fine tune the CLIP model can be found in the appendix. To ensure that the model is in the proximity of the successful pre-trained version, we keep the learning rate low and experiment with freezing some parts of the model. After fine tuning for 20 epochs where each took 2.5 hours on a RTX 2070 graphic card, we observed that the training loss decreased from $\sim$1.5 to $\sim$0.1. However, we also observed that the validation loss had very little changes. To address that, we resorted to using the Kerem Turgutlu's FastAI's finetuning scheme [37] to finetune the CLIP model in order to check whether our implementation of fine-tuning was successful or not. During FastAI's finetuning training loss again decrease from 1.5 to 0.04 but the validation loss still only changed from 1.27 to 1.18, therefore we decided on checking the models further. Upon checking the performances of the fine-tuned models against themselves and the pre-trained model, by giving them randomly chosen prompts and a image for them connect the right image with the right prompt, we see that both fine-tuned models perform indistinguishably and they outperform the pre-trained models. For example for the image in Figure 6, 5 prompts were given to the models: "a black and white dog holding a ball near sea", "a red dog holding a ball", "a white horse running in a grassy field", "a sports car in streets", and "a cat holding a ball". The model fine-tuned by our code gives 96.7% probability to the "a black and white dog holding a ball near sea" prompt whereas the model fine-tuned by FastAI's scheme gives 96.8% outperforming the pre-trained CLIP which outputs 94.2%. For the rest of the project we decided to use the model fine-tuned by FastAI's scheme as it is a cleaner

Figure 6: An image from the test dataset to test the finetuned the CLIP model that is discussed in 2.3.1. It was also the source of inspiration for us to test the generations of models with the prompt "a dog in front of sea"

code-piece. However, we found out that FastAI's implementation for CLIP with "ViT-L14" was faulty and there was no configuration setting for "ViT-B16", therefore we corrected and implemented them ourselves.

2.3.2 CLIP GLASS Training

CLIP Glass model does not require further training. One possibility was to train the Deepmind's BigGAN over our dataset but it was computationally too heavy and upon training it for several epochs (each taking around $\sim$ 3-4 hours) there was no visible performance increase. Therefore we use the pretrained BigGAN 256 model to do a latent space search on. During the development procedure, multiple GAN models were tested to find the best performing one, StyleGAN was the best performing among other alternatives yet its limited scope due to its training being on data sets of faces or churches or cars, outputs were not generalizable. BigGAN's other configurations -128 and 512- were also tested: BigGAN 128's outputs are poor in quality and 512 was computationally heavy for our setting. Therefore, the GAN used for latent space search is BigGAN 256. To increase the performance from [23], we utilize our fine-tuned CLIP model and experimented with the parameters of the genetic algorithm. To implement the genetic algorithm we use the pymoo library and refer the reader to its documentation for details [38]. Some design choices of the genetic algorithm were chosen as following: Initial sampling is done following a truncated normal distribution, crossover technique is Simulated Binary Crossover with each variable participating where eta variable is set to 3, and the mutation type was polynomial mutation. Lastly, and maybe most importantly, the evaluation of the population was done according to the cosine similarity defined in 1. The genetic algorithm was run for 500 generations.

2.3.3 VQGAN+CLIP Training

Prior to training a VQGAN model, there needs to be a pretrained CLIP model [17].To evaluate the performance of our training, besides using different prompts, we decided to compare the one-shot results of the network with the fine-tuned versions [36]. "ViT-B/32" is the default clip model used in built-in VQGAN model [39]. We also have used the same model to fine tune our CLIP model. So, we gave several prompts and observed the outputs of both fine-tuned and one-shot versions of them. We kept track of the loss during training to avoid overfitting and to detect convergence. Performance of the model has improved when fine-tuning has done. Github repository was followed when building and training the both environment and the model [40].

2.3.4 CLIP Guided Diffusion Training

Similar to the other models, main novelty of our implementation is to use our own fine-tuned CLIP model. In the earlier stages of the project, we conjectured that we could train a diffusion model from scratch but after trying for weeks as a single epoch took $\sim$ 4-5 hours the performance of the model was far from satisfactory. Therefore, we decided to use a trained diffusion model's checkpoint to initiate the model and run the training for a a little amount of time. After doing that for many different pre-trained models, we decided to test the performances of the available pre-trained models and our fine-tuned ones. Although fine-tuning decreased the error slightly, we observed no visible improvement in the overall performance. Therefore, the fine tuning part was taken out of the project and we decided to use the pre-trained diffusion models due to limited access to computational tools making us unavailable to train a diffusion model on our own. Taking a slight detour, at this point, we want to mention that instead of CLIP guided diffusion model we planned to train and use DALL-E 2 but an epoch of training the decoder architecture of DALL-E 2 took $\sim$ 90 hours on RTX 2070 graphic card, therefore DALL-E 2 was taken out of the project as well since it also has no available pre-trained models that are ready to use. Turning back to the CLIP guided diffusion, we followed the repository [41] to generate images from the diffusion model. A single generation takes about 20 minutes on RTX 2070 GPU. Upon our research, we found out that this generation time can be decreased drastically while preserving the quality of the output, This is done by tweaking some parameters of the model, initializing the process by using a perlin noise, and by gradient checkpointing. The idea behind quick CLIP guided diffusion is credited to Daniel Russel yet we cannot cite him at the time as his repository is closed.

2.3.5 Stable Diffusion Training

Hugging face libraries are used while training the stable diffusion model [42]. To fine tune this model, we transferred our dataset to another format to match with the configuration of hugging face libraries [43]. Then, to use our CLIP model, we modified the source code of the diffusers to load our finetuned model instead of the OpenAI's finetuned one. We trained over the stable diffusion model "stable-diffusion-v1-4" with custom dataset and custom CLIP model. We fine tuned the mentioned stable model and observed the outputs. Thanks to functionality of Hugging face libraries, we accelerated our training process (using bash command "accelerate" with our scripts) and we were able to train our models way faster than expected [42].

2.3.6 Imagen Training

Although listed in training section, there is no actual training in this model. Since it is a huge model requiring excessive computational tools we did not fine-tune of train the model from scratch. Similar to DALL-E 2 case an epoch of training took $\sim$ 90 hours on RTX 2070 GPU, therefore we resort to pre-trained models. Development process of this model was to follow [44] in order to implement the model locally. Note that the text encoder model that we use, namely "T5-3B", is a inferior model to the "T5-XXL" used in actual Imagen and we conjecture that the pre-trained available Unet models are also inferior to the ones used in the actual model.

3 Results

After we were satisfied with each model's development we passed to the generation step. A prompt that we commonly used to assess the models' current performances was "a dog in front of sea", therefore we include the generations from it but it does not have a corresponding image in the test dataset, teherefore we use an image from the test dataset that fits to this prompt. Rest of the prompts are taken from the test dataset. Table 1 displays the outputs of the explained models given different prompts from the test data set. We also include the corresponding image from the test data set for comparison. We defer the discussion of the results to 4.

4 Discussion

The best outputs are given by the stable diffusion model which was expected as the best pre-trained version of stable diffusion model is available for use. Rest of the models' performances vary from image to image and depend heavily on subtle details in the prompt like understanding a brown and black horse as a single horse rather than two horses. Starting from VQGAN, we see that the overall setting of the image fits the prompt but there are random figures, horse is hardly a horse, and human faces are unrecognizable. In CLIP-GLASS only problematic output is the horse, although the grass in the image horse is not recognizable. Another different aspect of the CLIP-GLASS is the dark lighting and the lack of facial features in humans. We conjecture that the quality of the images would increase for higher number of generations in genetic algorithm but in its current setting the outputs are satisfactory. In CLIP guided diffusion, first two images are near perfect but the model struggles with generating a horse and especially struggles with the last two prompts, we conjecture that this is due to the details in prompts as it generates an image of a woman walking her dog imprinted on a sidewalk and a single human that tries to attain both female and male features with a dog as a painting on snow . Quick CLIP guided diffusion, on the other hand, sacrifices some of its quality in dog and room images but is far more successful in the last two prompts, it again fails to generate a recognizable horse even though the city setting is correct. Lastly, Imagen model performs satisfactorily with minor errors in all prompts, in all of the images the setting and the theme of the image is correct, however there are scenes that disrupt its reality like distorted dog face and horse body, and lack of facial features of humans. We again note that our implementation of the Imagen model is far from its original as we are limited in our computation power. Overall, from these prompts, we observe that the models can distinguish animals, can generate the image in the right setting, distinguish between colors, and have an adequate understanding of visualizing the natural language. Therefore, we regard the project as successful since it satisfactorily manages the initial goal of generating images from text input.

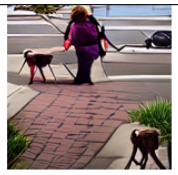
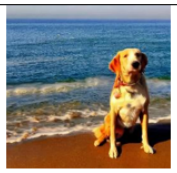
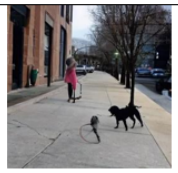
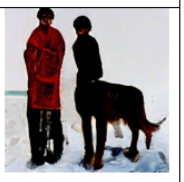
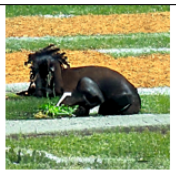
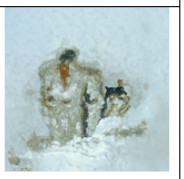
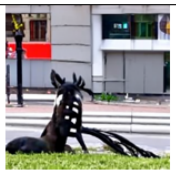
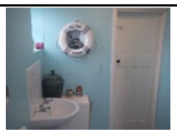
Prompt / Model	a dog in front of sea	a room with blue walls and a white sink and door	a brown and black horse in the middle of the city eating grass	a woman walking her dog on the sidewalk	a woman man and a dog standing in the snow
VQGAN-CLIP					
Stable Diffusion					
CLIP-GLASS					
CLIP Guided Diffusion					
Quick CLIP Guided Diffusion					
Imagen					
Test Data					

Table 1: Table of generated images for different models. Images in the bottom row are the images from the test data set for comparison. It can be seen that all models generate an image that is fit to the given prompt even though they have imperfections and limitations. Stable diffusion model performs best at generating photo-realistic images whereas other models have difficulties for some captions. One major limitation in the results seem to be the ability to convey human expressions or further any living expression without disturbing reality. Results also showcase the importance of the details in prompts, for example, even though the data set has two horses for "a brown and black horse" prompt models often take it as a single horse with brown and black fur. Again due to the prompts, CLIP guided diffusion has difficulties in last 2 prompts as it treats sidewalk and snow as the medium for the images.

4.1 Limitations

The most prevalent limitation in all of the models seems to be the human facial features. None of the models can output a realistic human face which is indeed a challenging task when the generator is not explicitly trained on human face data set. In our expriments, for example, using StyleGAN in CLIP-GLASS produced realistic human faces as it was solely trained on human face dataset. Another limitation of models is their susceptibility to minor changes in the prompt and high variance between generations. Additionally, CLIP-GLASS seems to have a bias towards images with darker lighting.

4.2 Future Work

As the model with best outputs, stable diffusion model does not seem to have much space for improvement that can be done in our computational setting, yet it may be possible to elevate its performance further by fine-tuning. For the CLIP-GLASS model, there are various exciting approaches to try in order to increase its performance. To list a few, first is to increase the population size in order to find the optimal population, second is to train a GAN model that has a better latent space than BigGAN for general purpose text-to-image generation, and third is to use the previous generations for a prompt to initialize the genetic algorithm on a better initial population. This idea of initializing with images of previous generations is also valid for CLIP guided diffusion models as their performances also depend on the initial image. For Imagen model, the most crucial improvement would be to train the diffusion model further and to use a larger T5 model in order to get closer to its reported performance in [20].

References

[1] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Networks," 2014. [Online]. Available: https://arxiv.org/abs/1406.2661

[2] Z. Zhang, M. Li, and J. Yu, "D2PGGAN: Two Discriminators Used in Progressive Growing of GANS," in *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Brighton, United Kingdom: IEEE, May 2019, pp. 3177–3181. [Online]. Available: https://ieeexplore.ieee.org/document/8683262/

[3] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Honolulu, HI: IEEE, Jul. 2017, pp. 105–114. [Online]. Available: http://ieeexplore.ieee.org/document/8099502/

[4] R. A. Yeh, C. Chen, T. Y. Lim, A. G. Schwing, M. Hasegawa-Johnson, and M. N. Do, "Semantic Image Inpainting with Deep Generative Models," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Honolulu, HI: IEEE, Jul. 2017, pp. 6882–6890. [Online]. Available: http://ieeexplore.ieee.org/document/8100211/

[5] J. Yu, Z. Lin, J. Yang, X. Shen, X. Lu, and T. Huang, "Free-Form Image Inpainting With Gated Convolution," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. Seoul, Korea (South): IEEE, Oct. 2019, pp. 4470–4479. [Online]. Available: https://ieeexplore.ieee.org/document/9010689/

[6] M. Frid-Adar, I. Diamant, E. Klang, M. Amitai, J. Goldberger, and H. Greenspan, "GAN-based synthetic medical image augmentation for increased CNN performance in liver lesion classification," *Neurocomputing*, vol. 321, pp. 321–331, Dec. 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0925231218310749

[7] L. A. Gatys, A. S. Ecker, and M. Bethge, "Image Style Transfer Using Convolutional Neural Networks," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA: IEEE, Jun. 2016, pp. 2414–2423. [Online]. Available: http://ieeexplore.ieee.org/document/7780634/

[8] Y. Jing, Y. Yang, Z. Feng, J. Ye, Y. Yu, and M. Song, "Neural Style Transfer: A Review," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 11, pp. 3365–3385, Nov. 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8732370/

[9] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-Image Translation with Conditional Adversarial Networks," Nov. 2018, arXiv:1611.07004 [cs]. [Online]. Available: http://arxiv.org/abs/1611.07004

[10] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks," in *2017 IEEE International Conference on Computer Vision (ICCV)*. Venice: IEEE, Oct. 2017, pp. 2242–2251. [Online]. Available: http://ieeexplore.ieee.org/document/8237506/

[11] J. Donahue and K. Simonyan, "Large Scale Adversarial Representation Learning," Nov. 2019, arXiv:1907.02544 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1907.02544

[12] Y. Bengio, A. Courville, and P. Vincent, "Representation Learning: A Review and New Perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1798–1828, Aug. 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6472238/

[13] M. Mirza and S. Osindero, "Conditional Generative Adversarial Nets," Nov. 2014, arXiv:1411.1784 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1411.1784

[14] S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee, "Generative Adversarial Text to Image Synthesis," Jun. 2016, arXiv:1605.05396 [cs]. [Online]. Available: http://arxiv.org/abs/1605.05396

[15] H. Park, Y. Yoo, and N. Kwak, "MC-GAN: Multi-conditional Generative Adversarial Network for Image Synthesis," Aug. 2018, arXiv:1805.01123 [cs]. [Online]. Available: http://arxiv.org/abs/1805.01123

[16] H. Zhang, T. Xu, H. Li, S. Zhang, X. Wang, X. Huang, and D. Metaxas, "StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks," Aug. 2017, arXiv:1612.03242 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1612.03242

[17] K. Crowson, S. Biderman, D. Kornis, D. Stander, E. Hallahan, L. Castricato, and E. Raff, "VQGAN-CLIP: Open Domain Image Generation and Editing with Natural Language Guidance," Sep. 2022, arXiv:2204.08583 [cs]. [Online]. Available: http://arxiv.org/abs/2204.08583

[18] J. Sohl-Dickstein, E. A. Weiss, N. Maheswaranathan, and S. Ganguli, "Deep Unsupervised Learning using Nonequilibrium Thermodynamics," Nov. 2015, arXiv:1503.03585 [cond-mat, q-bio, stat]. [Online]. Available: http://arxiv.org/abs/1503.03585

[19] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen, "Hierarchical Text-Conditional Image Generation with CLIP Latents," Apr. 2022, arXiv:2204.06125 [cs]. [Online]. Available: http://arxiv.org/abs/2204.06125

[20] C. Saharia, W. Chan, S. Saxena, L. Li, J. Whang, E. Denton, S. K. S. Ghasemipour, B. K. Ayan, S. S. Mahdavi, R. G. Lopes, T. Salimans, J. Ho, D. J. Fleet, and M. Norouzi, "Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding," May 2022, arXiv:2205.11487 [cs]. [Online]. Available: http://arxiv.org/abs/2205.11487

[21] "Stable Diffusion Public Release." [Online]. Available: https://stability.ai/blog/stable-diffusion-public-release

[22] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever, "Learning Transferable Visual Models From Natural Language Supervision," Feb. 2021, arXiv:2103.00020 [cs]. [Online]. Available: http://arxiv.org/abs/2103.00020

[23] F. A. Galatolo, M. G. C. A. Cimino, and G. Vaglini, "Generating images from caption and vice versa via CLIP-Guided Generative Latent Space Search," in *Proceedings of the International Conference on Image Processing and Vision Engineering*, 2021, pp. 166–174, arXiv:2102.01645 [cs]. [Online]. Available: http://arxiv.org/abs/2102.01645

[24] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," Jun. 2021, arXiv:2010.11929 [cs]. [Online]. Available: http://arxiv.org/abs/2010.11929

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," Dec. 2017, arXiv:1706.03762 [cs]. [Online]. Available: http://arxiv.org/abs/1706.03762

[26] R. Sennrich, B. Haddow, and A. Birch, "Neural Machine Translation of Rare Words with Subword Units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, 2016, pp. 1715–1725. [Online]. Available: http://aclweb.org/anthology/P16-1162

[27] A. v. d. Oord, O. Vinyals, and K. Kavukcuoglu, "Neural Discrete Representation Learning," May 2018, arXiv:1711.00937 [cs]. [Online]. Available: http://arxiv.org/abs/1711.00937

[28] P. Esser, R. Rombach, and B. Ommer, "Taming Transformers for High-Resolution Image Synthesis," Jun. 2021, arXiv:2012.09841 [cs]. [Online]. Available: http://arxiv.org/abs/2012.09841

[29] Y. Song and S. Ermon, "Generative Modeling by Estimating Gradients of the Data Distribution," Oct. 2020, arXiv:1907.05600 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1907.05600

[30] J. Ho, A. Jain, and P. Abbeel, "Denoising Diffusion Probabilistic Models," Dec. 2020, arXiv:2006.11239 [cs, stat]. [Online]. Available: http://arxiv.org/abs/2006.11239

[31] P. Dhariwal and A. Nichol, "Diffusion models beat gans on image synthesis," 2021. [Online]. Available: https://arxiv.org/abs/2105.05233

[32] [Online]. Available: https://github.com/crowsonkb/guided-diffusion

[33] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," 2021. [Online]. Available: https://arxiv.org/abs/2112.10752

[34] “Stable Diffusion with Diffusers.” [Online]. Available: https://huggingface.co/blog/stable_diffusion

[35] “Google Colaboratory.” [Online]. Available: https://colab.research.google.com/github/huggingface/notebooks/blob/main/diffusers/diffusers_intro.ipynb

[36] A. Dertat, “Applied Deep Learning - Part 3: Autoencoders,” Oct. 2017. [Online]. Available: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

[37] “CLIP | self_supervised.” [Online]. Available: https://keremturgutlu.github.io/self_supervised/20%20-%20clip.html

[38] “pymoo - GA: Genetic Algorithm.” [Online]. Available: https://pymoo.org/algorithms/soo/ga.html

[39] “sentence-transformers/clip-ViT-B-32 · Hugging Face.” [Online]. Available: https://huggingface.co/sentence-transformers/clip-ViT-B-32

[40] “Vqgan-clip.” [Online]. Available: https://github.com/nerdyrodent/VQGAN-CLIP

[41] “Clip guided diffusion.” [Online]. Available: https://github.com/nerdyrodent/CLIP-Guided-Diffusion

[42] “Stable Diffusion text-to-image fine-tuning.” [Online]. Available: https://huggingface.co/docs/diffusers/training/text2image

[43] “Load image data.” [Online]. Available: https://huggingface.co/docs/datasets/v2.4.0/en/image_load

[44] “Imagen- pytorch.” [Online]. Available: https://github.com/cene555/Imagen-pytorch/tree/20318089b42cf25fea27618319aa4f7a105be2ec

A Data from training

Here is the loss table from the fine tuning of CLIP model: Note that the training loss actually started

epoch	train_loss	valid_loss	time
0	0.255269	1.275214	1:24:28
1	0.204383	1.279200	1:58:25
2	0.167627	1.250787	1:20:20
3	0.146204	1.271605	1:22:42
4	0.116458	1.242378	1:18:53
5	0.093020	1.240321	1:17:48
6	0.079661	1.215634	1:17:51
7	0.064179	1.221287	1:17:46
8	0.060559	1.196998	1:17:48
9	0.046122	1.183175	1:18:11

Figure 7: Training and Validation loss for fastAI CLIP finetuning

from 1.5 and decreased throughout the first epoch. It is better seen in our implementation of fine tune curve as we plot the initial loss as well:

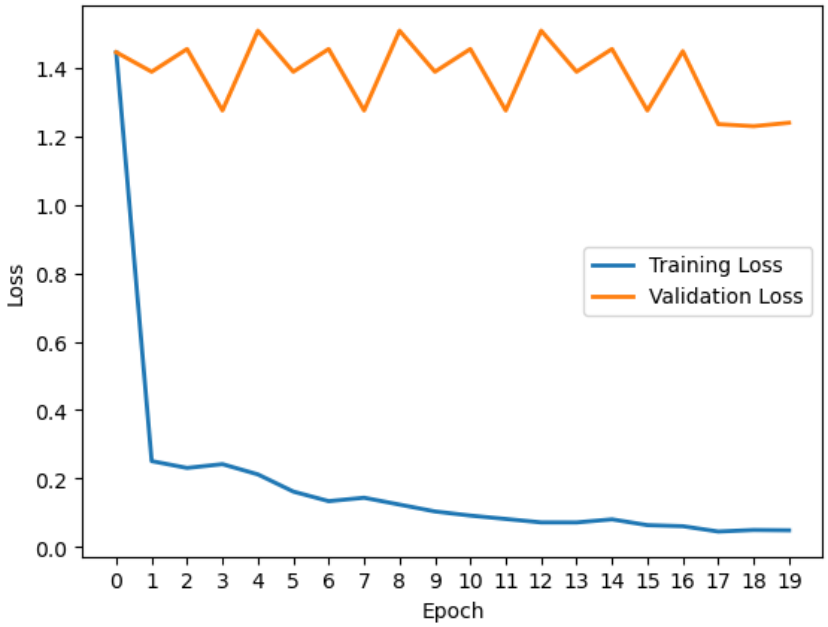


Figure 8: Training and Validation loss for our CLIP finetuning

Here is a loss curve from the quick CLIP guided diffusion model generation:

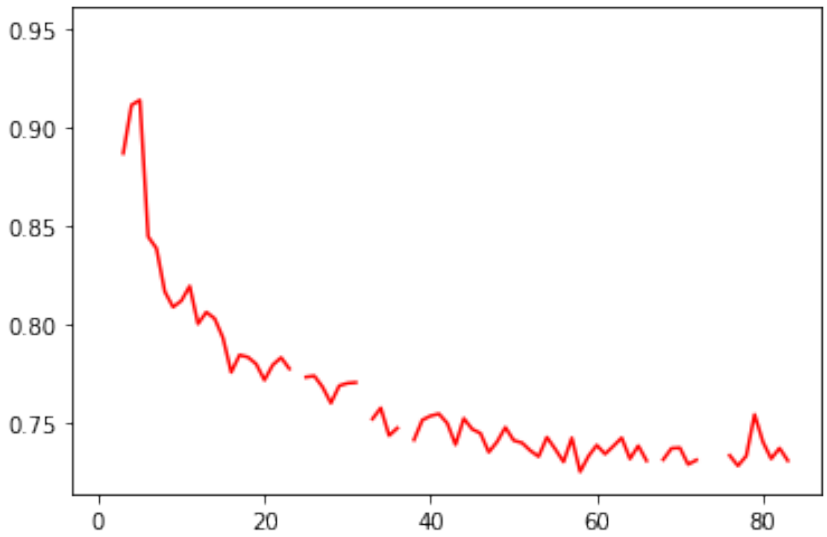


Figure 9: Loss curve during the generation of CLIP guided diffusion model

Here is a loss curve from a short fine-tune run of VQGAN model:

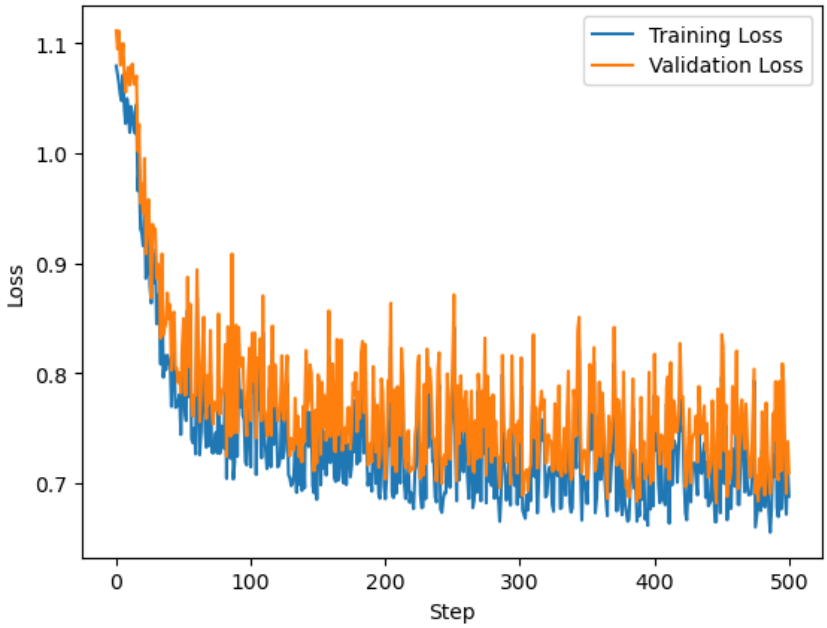


Figure 10: Loss curve during a a short fine-tune run of VQGAN model

B Codes

Note that some of the models were coded in notebook environments, therefore a direct run may not be possible.

B.1 Data Load and Preprocess

Download Data:

```
# -*- coding: utf-8 -*-
"""
Created on Fri Dec 23 17:21:29 2022

@author: gs_sr
"""

import h5py
import numpy as np

ftest = h5py.File('eee443_project_dataset_test.h5', 'r')
ftrain = h5py.File('eee443_project_dataset_train.h5', 'r')

# test_caps = ftest['test_caps']
# test_imid = ftest['test_imid']
# test_ims = ftest['test_ims']
test_url = ftest['test_url']

# train_caps = ftrain['train_cap']
# train_imid = ftrain['train_imid']
# train_ims = ftrain['train_ims']
train_url = ftrain['train_url']
# word_code = ftrain['word_code']

# print('Train Caps',train_caps.shape)
# print('TRain imid',train_imid.shape)
# print('Teain url',train_url[0].decode('UTF-8'))
# print('Word Code',word_code.shape)
# print('Min imid',min(train_imid))

# train_urlNP = np.array(ftrain['train_url'])
# import urllib.request

# urllib.request.urlretrieve(train_url[1].decode('UTF-8'), "/home/
    argegpu/seref/443final/iamge/im1")

# import requests

# #with open('pic1.jpg', 'wb') as handle:
# response = requests.get(train_url[1].decode('UTF-8'), stream=
    True)
#train_urlNP.shape[0]
import time
import urllib.request
seconds=time.time()
for i in range(train_url):
    imgURL = train_url[i].decode('UTF-8')
    img_name = imgURL.split("/")[-1].strip()
    if i%100==0:
        print(i)
        print((time.time()-seconds)/60)
        seconds=time.time()
    try:
        im_dir=f"C:\\Users\\gs_sr\\Desktop\\test_images\\{img_name
            }"
        urllib.request.urlretrieve(imgURL,im_dir )
```

```
    except:
        continue
```

Conversion from CMYK to RGB:

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import sys
import json
import jsonlines
from PIL import Image


# In[6]:


sys.path += ['data/image_dataset/train_images']


# In[14]:


data = jsonlines.open("data/train_lines.jsonl", "r")
for i in data:
    path_to_image = i['URL']
    image = Image.open(f'data/image_dataset/train_images/{
        path_to_image}')
    if image.mode == 'CMYK':
        print('DETECTED CMYK')
        image = image.convert('RGB')
        image.save(f'data/image_dataset/train_images/{
            path_to_image}')
```

Handle the caption data:

```
import numpy as np
import h5py as h5
import urllib
from urllib.request import urlopen
import PIL.Image
import grequests
import validators
import pandas as pd

ftest = h5.File('eee443_project_dataset_test.h5', 'r')
ftrain = h5.File('eee443_project_dataset_train.h5', 'r')

train_cap = ftrain['train_cap']
train_imid = ftrain['train_imid']
train_url = ftrain['train_url']

test_caps = ftest['test_caps']
test_imid = ftest['test_imid']
test_url = ftest['test_url']

word_code = ftrain['word_code']
```

```
words = pd.read_hdf('eee443_project_dataset_train.h5', 'word_code
    ') #same for test
words = words.to_dict('split')
wordDict = dict(zip(words['data'][0], words['columns']))

with h5.File('eee443_project_dataset_train.h5','r') as f: #same
    for test
    names = list(f.keys())
    train_cap = f[names[0]][()] #same for test
    train_imid = np.array(f[names[1]][()]) #same for test
    train_imid -=1 #no need in test
    train_url = np.array(f[names[3]][()]) #same for test

print(train_cap.shape)
print(train_imid.shape)
print(train_imid.max()),
print(train_imid.min())
print(train_url.shape)

string_array = np.empty(len(train_imid), dtype="<U120")
for i in range(len(string_array)):
    if i%50000==0:
        print(i)
    new_string = "";
    for j in train_cap[i]:
        if wordDict[j] == "x_START_":
            pass
        elif wordDict[j] == "x_END_":
            break
        else:
            new_string = new_string + " " + wordDict[j]
    new_string = new_string.strip()
    string_array[i] = new_string
```

Check whether image exists:

```
# -*- coding: utf-8 -*-
"""
Created on Sat Dec 24 19:46:30 2022

@author: gs_sr
"""
import os

ls=os.listdir(f"C:\\Users\\gs_sr\\Desktop\\train_images")
# for i in range(len(ls)):
#     ls[i] = ls[i][:-4]


import h5py
import numpy as np

# ftest = h5py.File('eee443_project_dataset_test.h5', 'r')
ftrain = h5py.File('eee443_project_dataset_train.h5', 'r')

# test_caps = ftest['test_caps']
# test_imid = ftest['test_imid']
# test_ims = ftest['test_ims']
# test_url = ftest['test_url']
```

```
# train_caps = ftrain['train_cap']
# train_imid = ftrain['train_imid']
# train_ims = ftrain['train_ims']
train_url = ftrain['train_url']
# word_code = ftrain['word_code']
arra=np.zeros(15000)
k=0
for j,i in enumerate(train_url):
    if any(s in i.decode('UTF-8') for s in ls):
        pass
    else:
        arra[k]=j
        k+=1
```

Create jsonline file:

```
# -*- coding: utf-8 -*-
"""
Created on Sat Dec 24 19:46:30 2022

@author: gs_sr
"""
import os

ls=os.listdir(f"C:\\Users\\gs_sr\\Desktop\\443_final\\dataset\\
    train_images")



import h5py
import numpy as np

# ftest = h5py.File('eee443_project_dataset_test.h5', 'r')
ftrain = h5py.File('eee443_project_dataset_train.h5', 'r')

# test_caps = ftest['test_caps']
# test_imid = ftest['test_imid']
# test_ims = ftest['test_ims']
# test_url = ftest['test_url']

# train_caps = ftrain['train_cap']
# train_imid = ftrain['train_imid']
# train_ims = ftrain['train_ims']
train_url = ftrain['train_url']

train_imid = ftrain['train_imid']
train_imid = np.array(ftrain['train_imid'])-1

# train_ims = ftrain['train_ims']
train_url = ftrain['train_url']

string_array=np.load('string_array.npy')
arr1inds = train_imid.argsort()
sorted_train_imid = train_imid[arr1inds[:]]
sorted_string_array = string_array[arr1inds[:]]
# word_code = ftrain['word_code']
arra=np.empty(82782,dtype="<U100")
#dataset=[{image_dir, captions_list}]
dataset_list=[]
```

```
k=0
prev=-1
for j,i in enumerate(train_url):
    if any(s in i.decode('UTF-8') for s in ls):
        captionList=[]
        whereIm=np.where(sorted_train_imid==j)[0]
        for k in whereIm:
            captionList.append(sorted_string_array[k])
        dictofIm={'URL':i.decode('UTF-8').split("/")[-1].strip(),
            'Captions':captionList}
        dataset_list.append(dictofIm)



import json
with open('output.jsonl','w') as outfile:
    for entry in dataset_list:
        json.dump(entry,outfile)
        outfile.write('\n')
```

B.2 CLIP Tuning

Self tune:

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import logging
import math
import os
import sys
import time
from dataclasses import dataclass, field
from pathlib import Path
from typing import Callable, Optional, List
import json
import jsonlines
import shutil
import numpy as np
import torch
import clip
import matplotlib.pyplot as plt
from tqdm.auto import tqdm
from PIL import Image

#import transformers
'''
from transformers import (
    CONFIG_MAPPING,
    AutoConfig,
    FlaxCLIPModel,
    CLIPModel,
    CLIPProcessor,
    CLIPTokenizer,
    TrainingArguments,
    is_tensorboard_available,
```

```
    IntervalStrategy

)
'''
from torchvision.datasets import VisionDataset
#from transformers.testing_utils import CaptureLogger
#from torchvision.io import ImageReadMode, read_image
'''
from torchvision.transforms import (
    # added for image augmentation
    ToPILImage,
    RandomCrop,
    ColorJitter,
    RandomHorizontalFlip,
    RandomVerticalFlip,
    RandomResizedCrop,
    ToTensor,
    # /added for image augmentation
    CenterCrop,
    ConvertImageDtype,
    Normalize,
    Resize
)
'''
#from torchvision.transforms.functional import InterpolationMode

import jsonlines
from pathlib import Path
from typing import Optional, Callable
from importlib.util import find_spec
import torch.nn as nn
logger = logging.getLogger(__name__)


# In[ ]:




# In[ ]:




# In[2]:


@dataclass
class ModelArguments:
    """
    Arguments pertaining to which model/config/tokenizer we are
        going to fine-tune, or train from scratch.
    """

    model_name_or_path: Optional[str] = field(
        default=None,
        metadata={
```

```
            "help": "The model checkpoint for weights
                initialization."
            "Don't set if you want to train a model from scratch."
        },
    )
    config_name: Optional[str] = field(
        default=None, metadata={"help": "Pretrained config name or
            path if not the same as model_name"}
    )
    tokenizer_name: Optional[str] = field(
        default=None, metadata={"help": "Pretrained tokenizer name
            or path if not the same as model_name"}
    )
    cache_dir: Optional[str] = field(
        default=None, metadata={"help": "Where do you want to
            store the pretrained models downloaded from s3"}
    )
    use_fast_tokenizer: bool = field(
        default=True,
        metadata={"help": "Whether to use one of the fast
            tokenizer (backed by the tokenizers library) or not
            ."},
    )
    dtype: Optional[str] = field(
        default="float32",
        metadata={
            "help": "Floating-point format in which the model
                weights should be initialized and trained. Choose
                one of `[float32, float16, bfloat16]`."
        },
    )
    save_optimizer: Optional[bool] = field(
        default=True,
        metadata={"help": "Whether to store full train state
            including optimizer."},
    )


@dataclass
class DataTrainingArguments:
    """
    Arguments pertaining to what data we are going to input our
        model for training and eval.
    """

    dataset_name: Optional[str] = field(
        default=None, metadata={"help": "The name of the dataset
            to use (via the datasets library)."}
    )
    data_dir: Optional[str] = field(
        default=None, metadata={"help": "Path to local folder
            containing data files."}
    )
    train_file: Optional[str] = field(
        default=None, metadata={"help": "The input training data
            file (a jsonlines file)."}
    )
    validation_file: Optional[str] = field(
        default=None,
```

```
    metadata={"help": "An optional input evaluation data file
        (a jsonlines file)."},
)
train_file: Optional[str] = field(default=None, metadata={"
    help": "The input training data file (a text file)."})
validation_file: Optional[str] = field(
    default=None,
    metadata={"help": "An optional input evaluation data file
        to evaluate the perplexity on (a text file)."},
)
max_train_samples: Optional[int] = field(
    default=None,
    metadata={
        "help": "For debugging purposes or quicker training,
            truncate the number of training examples to this "
        "value if set."
    },
)
max_eval_samples: Optional[int] = field(
    default=None,
    metadata={
        "help": "For debugging purposes or quicker training,
            truncate the number of evaluation examples to this
             "
        "value if set."
    },
)
overwrite_cache: bool = field(
    default=False, metadata={"help": "Overwrite the cached
        training and evaluation sets"}
)
validation_split_percentage: Optional[int] = field(
    default=5,
    metadata={
        "help": "The percentage of the train set used as
            validation set in case there's no validation split
            "
    },
)
block_size: Optional[int] = field(
    default=None,
    metadata={
        "help": "Optional input sequence length after
            tokenization. "
        "The training dataset will be truncated in block of
            this size for training. "
        "Default to the model max input length for single
            sentence inputs (take into account special tokens)
            ."
    },
)
overwrite_cache: bool = field(
    default=False, metadata={"help": "Overwrite the cached
        training and evaluation sets"}
)
preprocessing_num_workers: Optional[int] = field(
    default=None,
    metadata={"help": "The number of processes to use for the
        preprocessing."},
```

```
    )
    text_column_name: Optional[str] = field(
            default='text',
            metadata={"help": "Column containing main text data
                .")},
    )
    augment_images: Optional[bool] = field(
        default=True,
        metadata={ "help": "Augment input training images" }
    )
    augment_captions: Optional[bool] = field(
        default=True,
        metadata={ "help": "Augment input training images" }
    )
    captions_per_image: Optional[int] = field(
        default=5,
        metadata={"help": "Number of captions per image to use
            when creating train dataset."},
    )

    def __post_init__(self):
        if self.dataset_name is None and self.train_file is None
            and self.validation_file is None:
            raise ValueError("Need either a dataset name or a
                training/validation file.")
        else:
            if self.train_file is not None:
                extension = self.train_file.split(".")[-1]
                assert extension in ["csv", "json", "txt", "jsonl
                    "], "`train_file` should be a csv, a json or a
                     txt file."
            if self.validation_file is not None:
                extension = self.validation_file.split(".")[-1]
                assert extension in ["csv", "json", "txt", "jsonl
                    "], "`validation_file` should be a csv, a json
                     or a txt file."


@dataclass
class ImageAugmentationArguments:
    """
    Arguments for image augmentations configuration
    """
    random_horizontal_flip: Optional[float] = field(
        default=0.5,
        metadata={ "help": "Probability of applying random
            horizontal flip" }
    )
    random_vertical_flip: Optional[float] = field(
        default=0.5,
        metadata={ "help": "Probability of applying random
            vartical flip" }
    )


# In[3]:


data_args = DataTrainingArguments(
```

```
    dataset_name = None,
    data_dir = "./data",
    train_file = "data/train_lines.jsonl",
    validation_file = "data/train_lines.jsonl",
    max_train_samples = None,
    max_eval_samples = None,
    overwrite_cache = False,
    validation_split_percentage = 5,
    preprocessing_num_workers = 0,
    augment_images = False, ##Can be changed to augment captions
       and images (NEEDS FUTURE WORK)
    augment_captions = False,
    captions_per_image = 5,
)

augmentation_args = ImageAugmentationArguments()


# In[4]:


class ImageTextDataset(VisionDataset):
    """
    Dtaset for loading image-text data for tasks like CLIP
       training, Image Captioning.

    Args:
        root: (string): The root path where the dataset is stored.
            The expected format is jsonlines where each line is a
               json object containing to keys.
            `filename`: The path to the image.
            `captions`: An `array` of captions.
        split: (string): Dataset split name. Is used for parsing
           jsonl files from `root` folder.
        captions_per_image: (int): number of captions per image to
            use. Defaults to 5.
        augment_captions: (bool): If true the jsonl files with `
           textaug_` prefix are selected from root
            folder.
        transform (callable, optional): A function/transform that
            takes in an PIL image
            and returns a transformed version. E.g, ``transforms.
               ToTensor``
        target_transform (callable, optional): A function/
           transform that takes in the
            target and transforms it.
        transforms (callable, optional): A function/transform that
            takes input sample and its target as entry
            and returns a transformed version.
    """

    def __init__(
        self,
        root: str,
        split: str,
        captions_per_image:int = 5,
    ):
        super().__init__(root, None, None, None)
        self.root = root
```

```
        prefix = ""
        filepaths = list(Path(root).glob(f"{prefix}train*.jsonl"))
        fps_empty_msg = f"""\
        The `filepaths` is empty. Please make sure that `root`
            folder contains jsonl files
        named properly: [textaug_]{split}*.jsonl.
        `textaug_` prefix is expected if `augment_captions` is `
            True`.
        """
        assert len(filepaths) > 0, fps_empty_msg

        self.captions = []
        self.image_paths = []
        for count, filepath in enumerate(filepaths):
            with jsonlines.open(filepath, "r") as reader:
                length = len(list(jsonlines.open(filepath, "r")))
                step =0
                for example in reader:
                    if captions_per_image ==1:
                        captions_per_image_cur = 1
                    else:
                        captions_per_image_cur = len(example["
                            Captions"])

                    if split=='train' and step < 0.85*length:
                        self.captions.extend(clip.tokenize(example
                            ["Captions"][:captions_per_image_cur])
                            )
                        self.image_paths.extend([example["URL"]] *
                             captions_per_image_cur)
                    elif split=='valid' and step > 0.85*length:
                        self.captions.extend(clip.tokenize(example
                            ["Captions"][:captions_per_image_cur])
                            )
                        self.image_paths.extend([example["URL"]] *
                             captions_per_image_cur)
                    step+=1
        print(f"{count+1} input files for {split} split found")

    def _load_image(self, idx: int):
        path = f"{self.root}/image_dataset/train_images/{self.
            image_paths[idx]}"
        return preprocess(Image.open(path))

    def _load_target(self, idx):
        return self.captions[idx]

    def __getitem__(self, index: int):
        path = f"{self.root}/image_dataset/train_images/{self.
            image_paths[index]}"
        image = preprocess(Image.open(path))
        target = self.captions[index]

        if self.transforms is not None:
            image, target = self.transforms(image, target)
        return image, target

    def __len__(self) -> int:
        return len(self.captions)
```

```
# In[7]:


torch.cuda.is_available()
device = "cuda:0" if torch.cuda.is_available() else "cpu" # If
    using GPU then use mixed precision training.


# In[8]:


device


# In[18]:


model, preprocess = clip.load("ViT-B/32",device=device,jit=False)
    #Must set jit=False for training


# In[14]:


train_dataset = ImageTextDataset(
    data_args.data_dir,
    'train',
    captions_per_image=data_args.captions_per_image,
)

eval_dataset = ImageTextDataset(
    data_args.data_dir,
    'valid',
    captions_per_image=1,
)


# In[7]:


model, preprocess = clip.load("ViT-B/32",device=device,jit=False)
    #Must set jit=False for training
optimizer = torch.optim.Adam(model.parameters(), lr=5e-7,betas
    =(0.9,0.98),eps=1e-6,weight_decay=0.2) #Params used from paper
    , the lr is smaller, more safe for fine tuning to new dataset

checkpoint = torch.load("model_checkpoint/model_1_epoch2.pt")
'''
# Use these 3 lines if you use default model setting(not training
    setting) of the clip. For example, if you set context_length
    to 100 since your string is very long during training, then
    assign 100 to checkpoint['model_state_dict']["context_length"]
checkpoint['model_state_dict']["input_resolution"] = model.
    input_resolution #default is 224
checkpoint['model_state_dict']["context_length"] = model.
    context_length # default is 77
checkpoint['model_state_dict']["vocab_size"] = model.vocab_size
```

```
'''
model.load_state_dict(checkpoint['model_state_dict'])


# In[20]:


optimizer.load_state_dict(checkpoint['optimizer_state_dict'])


# In[21]:


optimizer = torch.optim.Adam(model.parameters(), lr=5e-6, betas
    =(0.9,0.98),eps=1e-6,weight_decay=0.2) #Params used from paper
    , the lr is smaller, more safe for fine tuning to new dataset


# In[18]:


import gc


# In[22]:


# use your own data
train_dataloader = torch.utils.data.DataLoader(train_dataset,
    batch_size = 16, shuffle=True) #Define your own dataloader
eval_dataloader = torch.utils.data.DataLoader(eval_dataset,
    batch_size = 16, shuffle=True) #Define your own dataloader
#https://github.com/openai/CLIP/issues/57
def convert_models_to_fp32(model):
    for p in model.parameters():
        p.data = p.data.float()
        p.grad.data = p.grad.data.float()

#optimizer = torch.optim.Adam(model.parameters(), lr=5e-7,betas
    =(0.9,0.98),eps=1e-6,weight_decay=0.2) #Params used from paper
    , the lr is smaller, more safe for fine tuning to new dataset
clip.model.convert_weights(model)
#model.float()
epoch_avg =0
cost_hist = []
eval_epoch_avg =0
eval_cost_hist = []

total_loss =0
loss_img = nn.CrossEntropyLoss()
loss_txt = nn.CrossEntropyLoss()

def cross_entropy(logits, axis):
    logprobs = nn.functional.log_softmax(logits, dim=axis)
    nll = torch.diag(logprobs)
    ce = -torch.mean(nll)
    return ce

def clip_loss(similarity):
```

```
    loss = (cross_entropy(similarity, axis=0) + cross_entropy(
        similarity, axis=1)) / 2
    return loss


epochs = tqdm(range(5))
# add your own code to track the training progress.
for epoch in epochs:
    steps_per_epoch = len(train_dataset) // 16
    epoch_avg =0
    # train
    steps_trained_progress_bar = tqdm(range(steps_per_epoch), desc
        =f"Training. Loss: {total_loss}", position=1, leave=False)
    for step, batch in enumerate(train_dataloader):
        optimizer.zero_grad()
        steps_trained_progress_bar.update(1)
        images,texts = batch

        images= images.to(device)
        texts = texts.to(device)

        logits_per_image, logits_per_text = model(images, texts)
        #logits = model(images, texts)[0]

        ground_truth = torch.arange(len(images),dtype=torch.long,
            device=device)
        #total_loss = clip_loss(logits)
        total_loss = (loss_img(logits_per_image,ground_truth) +
            loss_txt(logits_per_text,ground_truth))/2
        #print(total_loss , alt_loss)
        total_loss.backward()
        if device == "cpu":
            optimizer.step()
        else :
            convert_models_to_fp32(model)
            optimizer.step()
            clip.model.convert_weights(model)

        epoch_avg+= total_loss.item()
        steps_trained_progress_bar.set_description('\rStep: %s,
            Total loss%.4f' % (step, total_loss))


    # EVAL
    eval_steps_per_epoch = len(eval_dataset) // 16
    eval_epoch_avg =0

    eval_steps_trained_progress_bar = tqdm(range(
        eval_steps_per_epoch), desc=f"Eval. Loss: {total_loss}",
        position=1, leave=False)
    for step, batch in enumerate(eval_dataloader):
        eval_steps_trained_progress_bar.update(1)
        images,texts = batch

        images= images.to(device)
        texts = texts.to(device)

        #logits_per_image, logits_per_text = model(images, texts)
        logits = model(images, texts)[0]
```

```
        #ground_truth = torch.arange(len(images),dtype=torch.long,
            device=device)
        total_loss = clip_loss(logits).item()
        eval_epoch_avg+=total_loss
        eval_steps_trained_progress_bar.set_description('\rStep: %
            s, Total loss%.4f' % (step, total_loss))
    epoch_avg /= steps_per_epoch
    eval_epoch_avg /= eval_steps_per_epoch
    eval_cost_hist.append(eval_epoch_avg)
    cost_hist.append(epoch_avg)

    epochs.set_description('\rEpoch: %s, Training Loss: %.4f,
        Validation Loss: %.4f' % (epoch, epoch_avg, eval_epoch_avg
        ))
    torch.save({
        'epoch': epoch+1,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'loss': total_loss,
        }, f"model_checkpoint/model_1_epoch{epoch}.pt") #just
            change to your preferred folder/filename


# In[23]:


cost_hist


# In[32]:


eval_cost_hist


# In[1]:


torch.cuda.empty_cache()


# In[15]:


import gc
gc.collect()


# In[ ]:


Epoch loss: 0.06743557198830087


# In[ ]:


Validation loss: 0.1261
```

```
# In[4]:


device= 'cuda:0'


# In[5]:


model, preprocess = clip.load("ViT-B/32",device=device,jit=False)
    #Must set jit=False for training


# In[6]:


checkpoint = torch.load("model_checkpoint/model_1_epoch20.pt")
'''
# Use these 3 lines if you use default model setting(not training
    setting) of the clip. For example, if you set context_length
    to 100 since your string is very long during training, then
    assign 100 to checkpoint['model_state_dict']["context_length"]
checkpoint['model_state_dict']["input_resolution"] = model.
    input_resolution #default is 224
checkpoint['model_state_dict']["context_length"] = model.
    context_length # default is 77
checkpoint['model_state_dict']["vocab_size"] = model.vocab_size
'''
model.load_state_dict(checkpoint['model_state_dict'])


# In[7]:


image = preprocess(Image.open('data/image_dataset/test_images
    /3625376_1dfc183f7b_z.jpg')).unsqueeze(0).to(device)
text = clip.tokenize(["a black and white dog holding a ball near
    sea"]).to(device)

with torch.no_grad():
    image_features = model.encode_image(image)
    text_features = model.encode_text(text)

    logits_per_image, logits_per_text = model(image, text)
    probs = logits_per_image.softmax(dim=-1).cpu().numpy()
for l, p in zip(["A black dog","a white and black dog ","a black
    cat","a black and white dog near sea","a black and white dog
    holding a ball near sea"], probs[0]):
    print(f"{l:<16} {p:.4f}")


# In[8]:


image_features /= image_features.norm(dim=-1, keepdim=True)
text_features /= text_features.norm(dim=-1, keepdim=True)
similarity = text_features.cpu().numpy() @ image_features.cpu().
    numpy().T
```

```
# In[9]:


similarity


# In[19]:


logit_scale = model_a.logit_scale.exp()


# In[75]:


logit_scale


# In[31]:


plt.imshow(Image.open('data/image_dataset/test_images/3625376
    _1dfc183f7b_z.jpg'))


# In[ ]:




# In[ ]:




# In[70]:


torch.save({
        'epoch': epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'loss': total_loss,
        }, f"model_checkpoint/model_1_epoch{epoch}.pt") #just
            change to your preferred folder/filename
```

FastAI's fine tune:

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import logging
import math
```

```
import os
import sys
import time
from dataclasses import dataclass, field
from pathlib import Path
from typing import Callable, Optional, List
import json
import jsonlines
import shutil
import numpy as np
import torch
import clip
import matplotlib.pyplot as plt
from tqdm.auto import tqdm
from PIL import Image

#import transformers
'''
from transformers import (
    CONFIG_MAPPING,
    AutoConfig,
    FlaxCLIPModel,
    CLIPModel,
    CLIPProcessor,
    CLIPTokenizer,
    TrainingArguments,
    is_tensorboard_available,
    IntervalStrategy

)
'''
from torchvision.datasets import VisionDataset
#from transformers.testing_utils import CaptureLogger
#from torchvision.io import ImageReadMode, read_image
'''
from torchvision.transforms import (
    # added for image augmentation
    ToPILImage,
    RandomCrop,
    ColorJitter,
    RandomHorizontalFlip,
    RandomVerticalFlip,
    RandomResizedCrop,
    ToTensor,
    # /added for image augmentation
    CenterCrop,
    ConvertImageDtype,
    Normalize,
    Resize
)
'''
#from torchvision.transforms.functional import InterpolationMode

import jsonlines
from pathlib import Path
from typing import Optional, Callable
from importlib.util import find_spec
import torch.nn as nn
logger = logging.getLogger(__name__)
```

```
# In[2]:


sys.path.append(r"./data/image_dataset/train_images")


# In[3]:


from fastai.vision.all import *
from fastai.distributed import *


# In[4]:


from self_supervised.multimodal.clip import *
import clip


# In[5]:


@dataclass
class ModelArguments:
    """
    Arguments pertaining to which model/config/tokenizer we are
        going to fine-tune, or train from scratch.
    """

    model_name_or_path: Optional[str] = field(
        default=None,
        metadata={
            "help": "The model checkpoint for weights
                initialization."
            "Don't set if you want to train a model from scratch."
        },
    )
    config_name: Optional[str] = field(
        default=None, metadata={"help": "Pretrained config name or
            path if not the same as model_name"}
    )
    tokenizer_name: Optional[str] = field(
        default=None, metadata={"help": "Pretrained tokenizer name
            or path if not the same as model_name"}
    )
    cache_dir: Optional[str] = field(
        default=None, metadata={"help": "Where do you want to
            store the pretrained models downloaded from s3"}
    )
    use_fast_tokenizer: bool = field(
        default=True,
        metadata={"help": "Whether to use one of the fast
            tokenizer (backed by the tokenizers library) or not
            ."},
    )
    dtype: Optional[str] = field(
```

```
        default="float32",
        metadata={
            "help": "Floating-point format in which the model
                weights should be initialized and trained. Choose
                one of `[float32, float16, bfloat16]`."
        },
    )
    save_optimizer: Optional[bool] = field(
        default=True,
        metadata={"help": "Whether to store full train state
            including optimizer."},
    )


@dataclass
class DataTrainingArguments:
    """
    Arguments pertaining to what data we are going to input our
        model for training and eval.
    """

    dataset_name: Optional[str] = field(
        default=None, metadata={"help": "The name of the dataset
            to use (via the datasets library)."}
    )
    data_dir: Optional[str] = field(
        default=None, metadata={"help": "Path to local folder
            containing data files."}
    )
    train_file: Optional[str] = field(
        default=None, metadata={"help": "The input training data
            file (a jsonlines file)."}
    )
    validation_file: Optional[str] = field(
        default=None,
        metadata={"help": "An optional input evaluation data file
            (a jsonlines file)."},
    )
    train_file: Optional[str] = field(default=None, metadata={"
        help": "The input training data file (a text file)."})
    validation_file: Optional[str] = field(
        default=None,
        metadata={"help": "An optional input evaluation data file
            to evaluate the perplexity on (a text file)."},
    )
    max_train_samples: Optional[int] = field(
        default=None,
        metadata={
            "help": "For debugging purposes or quicker training,
                truncate the number of training examples to this "
            "value if set."
        },
    )
    max_eval_samples: Optional[int] = field(
        default=None,
        metadata={
            "help": "For debugging purposes or quicker training,
                truncate the number of evaluation examples to this
                 "
```

```
            "value if set."
        },
    )
    overwrite_cache: bool = field(
        default=False, metadata={"help": "Overwrite the cached
            training and evaluation sets"}
    )
    validation_split_percentage: Optional[int] = field(
        default=5,
        metadata={
            "help": "The percentage of the train set used as
                validation set in case there's no validation split
                "
        },
    )
    block_size: Optional[int] = field(
        default=None,
        metadata={
            "help": "Optional input sequence length after
                tokenization. "
            "The training dataset will be truncated in block of
                this size for training. "
            "Default to the model max input length for single
                sentence inputs (take into account special tokens)
                ."
        },
    )
    overwrite_cache: bool = field(
        default=False, metadata={"help": "Overwrite the cached
            training and evaluation sets"}
    )
    preprocessing_num_workers: Optional[int] = field(
        default=None,
        metadata={"help": "The number of processes to use for the
            preprocessing."},
    )
    text_column_name: Optional[str] = field(
            default='text',
            metadata={"help": "Column containing main text data
                ."},
    )
    augment_images: Optional[bool] = field(
        default=True,
        metadata={ "help": "Augment input training images" }
    )
    augment_captions: Optional[bool] = field(
        default=True,
        metadata={ "help": "Augment input training images" }
    )
    captions_per_image: Optional[int] = field(
        default=5,
        metadata={"help": "Number of captions per image to use
            when creating train dataset."},
    )

    def __post_init__(self):
        if self.dataset_name is None and self.train_file is None
            and self.validation_file is None:
```

```
            raise ValueError("Need either a dataset name or a
                training/validation file.")
        else:
            if self.train_file is not None:
                extension = self.train_file.split(".")[-1]
                assert extension in ["csv", "json", "txt", "jsonl
                    "], "`train_file` should be a csv, a json or a
                     txt file."
            if self.validation_file is not None:
                extension = self.validation_file.split(".")[-1]
                assert extension in ["csv", "json", "txt", "jsonl
                    "], "`validation_file` should be a csv, a json
                     or a txt file."


@dataclass
class ImageAugmentationArguments:
    """
    Arguments for image augmentations configuration
    """
    random_horizontal_flip: Optional[float] = field(
        default=0.5,
        metadata={ "help": "Probability of applying random
            horizontal flip" }
    )
    random_vertical_flip: Optional[float] = field(
        default=0.5,
        metadata={ "help": "Probability of applying random
            vartical flip" }
    )


# In[6]:


data_args = DataTrainingArguments(
    dataset_name = None,
    data_dir = "./data",
    train_file = "data/train_lines.jsonl",
    validation_file = "data/train_lines.jsonl",
    max_train_samples = None,
    max_eval_samples = None,
    overwrite_cache = False,
    validation_split_percentage = 5,
    preprocessing_num_workers = 0,
    augment_images = False, ##Can be changed to augment captions
        and images (NEEDS FUTURE WORK)
    augment_captions = False,
    captions_per_image = 5,
)

augmentation_args = ImageAugmentationArguments()


# In[7]:


class ImageTextDataset(VisionDataset):
    """
```

```
Dtaset for loading image-text data for tasks like CLIP
    training, Image Captioning.

Args:
    root: (string): The root path where the dataset is stored.
        The expected format is jsonlines where each line is a
            json object containing to keys.
        `filename`: The path to the image.
        `captions`: An `array` of captions.
    split: (string): Dataset split name. Is used for parsing
        jsonl files from `root` folder.
    captions_per_image: (int): number of captions per image to
         use. Defaults to 5.
    augment_captions: (bool): If true the jsonl files with `
        textaug_` prefix are selected from root
         folder.
    transform (callable, optional): A function/transform that
         takes in an PIL image
        and returns a transformed version. E.g, ``transforms.
            ToTensor``
    target_transform (callable, optional): A function/
        transform that takes in the
         target and transforms it.
    transforms (callable, optional): A function/transform that
         takes input sample and its target as entry
        and returns a transformed version.
"""

def __init__(
    self,
    root: str,
    split: str,
    captions_per_image:int = 5,
):
    super().__init__(root, None, None, None)
    self.root = root
    prefix = ""
    filepaths = list(Path(root).glob(f"{prefix}train*.jsonl"))
    fps_empty_msg = f"""\
    The `filepaths` is empty. Please make sure that `root`
        folder contains jsonl files
    named properly: [textaug_]{split}*.jsonl.
    `textaug_` prefix is expected if `augment_captions` is `
        True`.
    """
    assert len(filepaths) > 0, fps_empty_msg
    self.captions_dict = {}
    self.captions = []
    self.image_paths = []
    for count, filepath in enumerate(filepaths):
        with jsonlines.open(filepath, "r") as reader:
            length = len(list(jsonlines.open(filepath, "r")))
            step =0
            for example in reader:
                if captions_per_image ==1:
                    captions_per_image_cur = 1
                else:
                    captions_per_image_cur = len(example["
                        Captions"])
```

```
                    if split=='train' and step < 0.85*length:
                        self.captions.extend(example["Captions"][:
                            captions_per_image_cur])
                        tmp_dict = {example["URL"]:example["
                            Captions"][:captions_per_image_cur] }
                        self.image_paths.extend([example["URL"]] *
                             captions_per_image_cur)
                        self.captions_dict.update(tmp_dict)
                    elif split=='valid' and step > 0.85*length:
                        self.captions.extend(example["Captions"][:
                            captions_per_image_cur])
                        self.image_paths.extend([example["URL"]] *
                             captions_per_image_cur)
                    step+=1
        print(f"{count+1} input files for {split} split found")

    def _load_image(self, idx: int):
        path = f"{self.root}/image_dataset/train_images/{self.
            image_paths[idx]}"
        return preprocess(PILImage.create(path))
        #return preprocess(Image.open(path))

    def _load_target(self, idx):
        return self.captions[idx]

    def __getitem__(self, index: int):
        path = f"{self.root}/image_dataset/train_images/{self.
            image_paths[index]}"
        image = preprocess(Image.open(path))
        target = self.captions[index]

        if self.transforms is not None:
            image, target = self.transforms(image, target)
        return image, target

    def __len__(self) -> int:
        return len(self.captions)


# In[8]:


torch.cuda.is_available()
device = "cuda:0" if torch.cuda.is_available() else "cpu" # If
    using GPU then use mixed precision training.


# In[9]:


device


# In[10]:


train_dataset = ImageTextDataset(
    data_args.data_dir,
```

```
    'train',
    captions_per_image=data_args.captions_per_image,
)


eval_dataset = ImageTextDataset(
    data_args.data_dir,
    'valid',
    captions_per_image=1,
)


# In[11]:


train_paths = train_dataset.image_paths
eval_paths = eval_dataset.image_paths


# In[12]:


train_captions = train_dataset.captions
eval_captions = eval_dataset.captions


# In[13]:


def read_image(fn): return PILImage.create(fn)
def read_text(fn): return full_captions[fn.name][np.random.randint
    (len(full_captions[fn.name]))]
def dummy_targ(o): return 0


# In[14]:


def read_image_train(idx): return PILImage.create(f"./data/
    image_dataset/train_images/{train_paths[idx]}")
def read_text_train(idx): return train_captions[idx]
def dummy_targ(o): return 0


# In[15]:


def read_image_valid(idx): return PILImage.create(f"./data/
    image_dataset/train_images/{eval_paths[idx]}")
def read_text_valid(idx): return eval_captions[idx]


# In[16]:


train_images = get_image_files(f"./data/image_dataset/train_images
    /")
#valid_images = get_image_files(validpath)
```

```
# In[17]:


clip_stats = ([0.48145466, 0.4578275, 0.40821073], [0.26862954,
    0.26130258, 0.27577711])
clip_tokenizer = ClipTokenizer()
size,bs = 224, 32


item_tfms = [RandomResizedCrop(size, min_scale=0.9),
    clip_tokenizer, ToTensor()]

dsets_train = Datasets(np.arange(len(train_paths)), tfms=[
    read_image_train, read_text_train, dummy_targ],n_inp=2)
dsets_valid = Datasets(np.arange(len(eval_paths)), tfms=[
    read_image_valid, read_text_valid, dummy_targ],n_inp=2)

'''
n=int(0.85*len(train_images))
dsets_train = Datasets(train_images[:n], tfms=[read_image,
    read_text, dummy_targ],n_inp=2, get_idxs=None)
dsets_valid = Datasets(train_images[n:], tfms=[read_image,
    read_text, dummy_targ],n_inp=2, get_idxs=None)
'''


batch_tfms = [IntToFloatTensor, Normalize.from_stats(*clip_stats)]
train_dl = TfmdDL(dsets_train, shuffle=True, bs=bs, after_item=
    item_tfms, after_batch=batch_tfms, drop_last=True)

valid_dl = TfmdDL(dsets_valid, shuffle=False, bs=bs*2, after_item=
    item_tfms, after_batch=batch_tfms)

dls = DataLoaders(train_dl,valid_dl, device=default_device())


# In[18]:


for i in range(5): dls.train_ds[i][0].show(title=dls.train_ds[i
    ][1])


# In[19]:


clip_trainer_cb = CLIPTrainer()
cbs = [clip_trainer_cb]
opt_func = ranger

arch = 'vitb16'
do_finetune = True
use_grad_check = True
grad_check_nchunks = 2
finetune_modelname = 'ViT-B/16'
#finetune_modelname = 'ViT-B/32'

#Custom CONFIG  for ViT-B/16 Not present in Turgutlu's
    implementation!!!
```

```
def vitb16_config(input_res, context_length, vocab_size):
    "ViT-B/16 configuration, uses 16x16 patches"
    return dict(embed_dim=512,
                image_resolution=input_res,
                vision_layers=12,
                vision_width=768,
                vision_patch_size=16,
                context_length=context_length,
                vocab_size=vocab_size,
                transformer_width=512,
                transformer_heads=8,
                transformer_layers=12)

#This portion was wrong in Turgutlu's implementation!!!
def vitl14_config(input_res, context_length, vocab_size):
    "ViT-L/14 configuration, uses 14x14 patches"
    return dict(embed_dim=768,
                image_resolution=input_res,
                vision_layers=24,
                vision_width=1024,
                vision_patch_size=14,
                context_length=context_length,
                vocab_size=vocab_size,
                transformer_width=768,
                transformer_heads=8,
                transformer_layers=12)

vitb32_config_dict = vitb16_config(size, clip_tokenizer.
   context_length, clip_tokenizer.vocab_size)
#vitb32_config_dict = vitb32_config(size, clip_tokenizer.
   context_length, clip_tokenizer.vocab_size)

clip_model = CLIP(**vitb32_config_dict, checkpoint=use_grad_check,
    checkpoint_nchunks=grad_check_nchunks)


# In[20]:


if do_finetune:
    print("Loading pretrained model..")
    clip_pretrained_model, _ = clip.load(finetune_modelname, jit=
       False)
    clip_model.load_state_dict(clip_pretrained_model.state_dict())


# In[21]:


learner = Learner(dls, clip_model, loss_func=noop, cbs=cbs,
   opt_func=opt_func)
learner.to_fp16();


# In[22]:


learner.fit_flat_cos(10, 1e-5, pct_start=0.25)
```

```
# In[ ]:


torch.save({
        'model_state_dict': clip_model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'loss': total_loss,
        }, f"model_checkpoint/model_1_epoch{epoch}.pt") #just
            change to your preferred folder/filename


# In[57]:


learner.save(file ="./model_fastai", with_opt=True,
    pickle_protocol=2)


# In[59]:


checkpoint = torch.load('./models/model_fastai.pth')


# In[63]:


clip_pretrained_model.load_state_dict(checkpoint['model'])


# In[1]:


image = _(Image.open('data/image_dataset/test_images/3625376
    _1dfc183f7b_z.jpg')).unsqueeze(0).to(device)
text = clip.tokenize(["a black and white dog holding a ball near
    sea"]).to(device)

with torch.no_grad():
    image_features = clip_pretrained_model.encode_image(image)
    text_features = clip_pretrained_model.encode_text(text)

    logits_per_image, logits_per_text = clip_pretrained_model(
        image, text)
    probs = logits_per_image.softmax(dim=-1).cpu().numpy()
for l, p in zip(["A black dog","a white and black dog ","a black
    cat","a black and white dog near sea","a black and white dog
    holding a ball near sea"], probs[0]):
    print(f"{l:<16} {p:.4f}")


# In[67]:


image_features /= image_features.norm(dim=-1, keepdim=True)
text_features /= text_features.norm(dim=-1, keepdim=True)
similarity = text_features @ image_features.T
```

```
# In[ ]:
```

B.3 CLIP GLASS

CLIP GLASS model:

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import os
import argparse
import sys
import torch
from pytorch_pretrained_biggan import BigGAN as DMBigGAN
from pytorch_pretrained_biggan import  truncated_noise_sample,
    one_hot_from_names, display_in_terminal, save_as_images
from matplotlib import pyplot as plt

from pytorch_pretrained_biggan import BigGAN
import clip
import kornia
from PIL import Image
from torchvision.utils import save_image
import torchvision

import numpy as np
import pickle
from pymoo.optimize import minimize
from pymoo.algorithms.so_genetic_algorithm import GA #
    so_genetic_algorithm
from pymoo.factory import get_algorithm, get_decision_making,
    get_decomposition
from pymoo.visualization.scatter import Scatter

from scipy.stats import truncnorm

from pymoo.factory import get_sampling, get_crossover,
    get_mutation
from pymoo.operators.mixed_variable_operator import
    MixedVariableSampling, MixedVariableMutation,
    MixedVariableCrossover
from pymoo.model.sampling import Sampling

from pymoo.model.problem import Problem


# In[2]:


class TruncatedNormalRandomSampling(Sampling):
    def __init__(self, var_type=np.float64):
        super().__init__()
        self.var_type = var_type

    def _do(self, problem, n_samples, **kwargs):
```

```
        return truncnorm.rvs(-2, 2, size=(n_samples, problem.n_var
            )).astype(np.float64)

class NormalRandomSampling(Sampling):
    def __init__(self, mu=0, std=1, var_type=np.float64):
        super().__init__()
        self.mu = mu
        self.std = std
        self.var_type = var_type

    def _do(self, problem, n_samples, **kwargs):
        return np.random.normal(self.mu, self.std, size=(n_samples
            , problem.n_var))

class BinaryRandomSampling(Sampling):
    def __init__(self, prob=0.5):
        super().__init__()
        self.prob = prob

    def _do(self, problem, n_samples, **kwargs):
        val = np.random.random((n_samples, problem.n_var))
        return (val < self.prob).astype(np.bool_)

def get_operators(config):
    if config["config"] == "DeepMindBigGAN256" or config["config"]
        == "DeepMindBigGAN512":
        mask = ["real"]*config["dim_z"] + ["bool"]*config["
            num_classes"]

        real_sampling = None
        if config["config"] == "DeepMindBigGAN256" or config["
            config"] == "DeepMindBigGAN512":
            real_sampling = TruncatedNormalRandomSampling()

        sampling = MixedVariableSampling(mask, {
            "real": real_sampling,
            "bool": BinaryRandomSampling(prob=5/1000)
        })

        crossover = MixedVariableCrossover(mask, {
            "real": get_crossover("real_sbx", prob=1.0, eta=3.0),
            "bool": get_crossover("bin_hux", prob=0.2)
        })

        mutation = MixedVariableMutation(mask, {
            "real": get_mutation("real_pm", prob=0.5, eta=3.0),
            "bool": get_mutation("bin_bitflip", prob=10/1000)
        })

        return dict(
            sampling=sampling,
            crossover=crossover,
            mutation=mutation
        )


# In[3]:
```

```
def save_grid(images, path):
    grid = torchvision.utils.make_grid(images)
    torchvision.utils.save_image(grid, path)

def show_grid(images):
    grid = torchvision.utils.make_grid(images)
    plt.imshow(grid.permute(1, 2, 0).cpu().detach().numpy())
    plt.show()

def biggan_norm(images):
    images = (images + 1) / 2.0
    images = images.clip(0, 1)
    return images

def biggan_denorm(images):
    images = images*2 - 1
    return images


def freeze_model(model):
    for param in model.parameters():
        param.requires_grad = False


# In[4]:


class DeepMindBigGAN(torch.nn.Module):
    def __init__(self, config):
        super(DeepMindBigGAN, self).__init__()
        self.config = config
        self.G = DMBigGAN.from_pretrained(config["weights"])
        self.D = None

    def has_discriminator(self):
        return False

    def generate(self, z, class_labels, minibatch = None):
        if minibatch is None:
            return self.G(z, class_labels, self.config["truncation
                "])
        else:
            assert z.shape[0] % minibatch == 0
            gen_images = []
            for i in range(0, z.shape[0] // minibatch):
                z_minibatch = z[i*minibatch:(i+1)*minibatch, :]
                cl_minibatch = class_labels[i*minibatch:(i+1)*
                    minibatch, :]
                gen_images.append(self.G(z_minibatch, cl_minibatch
                    , self.config["truncation"]))
            gen_images = torch.cat(gen_images)
            return gen_images


# In[5]:


class GenerationProblem(Problem):
    def __init__(self, config):
```

```
        self.generator = Generator(config)
        self.config = config

        super().__init__(**self.config["problem_args"])

    def _evaluate(self, x, out, *args, **kwargs):
        ls = self.config["latent"](self.config)
        ls.set_from_population(x)

        with torch.no_grad():
            generated = self.generator.generate(ls, minibatch=self
                .config["batch_size"])
            sim = self.generator.clip_similarity(generated).cpu().
                numpy()
            if self.config["problem_args"]["n_obj"] == 2 and self.
                config["use_discriminator"]:
                dis = self.generator.discriminate(generated,
                    minibatch=self.config["batch_size"])
                hinge = torch.relu(1 - dis)
                hinge = hinge.squeeze(1).cpu().numpy()
                out["F"] = np.column_stack((-sim, hinge))
            else:
                out["F"] = -sim

            out["G"] = np.zeros((x.shape[0]))


# In[6]:


class DeepMindBigGANLatentSpace(torch.nn.Module):
    def __init__(self, config):
        super(DeepMindBigGANLatentSpace, self).__init__()
        self.config = config

        self.z = torch.nn.Parameter(torch.tensor(
            truncated_noise_sample(self.config["batch_size"])).to(
            self.config["device"]))
        self.class_labels = torch.nn.Parameter(torch.rand(self.
            config["batch_size"], self.config["num_classes"]).to(
            self.config["device"]))

    def set_values(self, z, class_labels):
        self.z.data = z
        self.class_labels.data = class_labels

    def set_from_population(self, x):
        self.z.data = torch.tensor(x[:,:self.config["dim_z"]].
            astype(float)).float().to(self.config["device"])
        self.class_labels.data = torch.tensor(x[:,self.config["
            dim_z"]:].astype(float)).float().to(self.config["
            device"])

    def forward(self):
        z = torch.clip(self.z, -2, 2)
        class_labels = torch.softmax(self.class_labels, dim=1)

        return z, class_labels
```

```
# In[7]:


configs = dict(
    DeepMindBigGAN256 = dict(
        task = "txt2img",
        dim_z = 128,
        num_classes = 1000,
        latent = DeepMindBigGANLatentSpace,
        model = DeepMindBigGAN,
        weights = "biggan-deep-256",
        use_discriminator = False,
        algorithm = "ga",
        norm = biggan_norm,
        denorm = biggan_denorm,
        truncation = 1.0,
        pop_size = 64,
        batch_size = 8,
        problem_args = dict(
            n_var = 128 + 1000,
            n_obj = 1,
            n_constr = 128,
            xl = -2,
            xu = 2
        )
    ),
    DeepMindBigGAN512 = dict(
        task = "txt2img",
        dim_z = 128,
        num_classes = 1000,
        latent = DeepMindBigGANLatentSpace,
        model = DeepMindBigGAN,
        weights = "biggan-deep-512",
        use_discriminator = False,
        algorithm = "ga",
        norm = biggan_norm,
        denorm = biggan_denorm,
        truncation = 1.0, #1.0
        pop_size = 32, #32
        batch_size = 1, #8
        problem_args = dict(
            n_var = 128 + 1000,
            n_obj = 1,
            n_constr = 128,
            xl = -2,
            xu = 2
        )
    )
)



def get_config(name):
    return configs[name]


# In[8]:
```

```
class Generator:
    def __init__(self, config):
        self.config = config
        self.augmentation = None

        self.CLIP, clip_preprocess = clip.load("ViT-B/32", device=
            self.config["device"], jit=False)

        #Load the fine tuned clip model
        checkpoint = torch.load('./models/model_fastai.pth')
        self.CLIP.load_state_dict(checkpoint['model'])

        self.CLIP = self.CLIP.eval()
        freeze_model(self.CLIP)
        self.model = self.config["model"](config).to(self.config["
            device"]).eval()
        freeze_model(self.model)

        if config["task"] == "txt2img":
            self.tokens = clip.tokenize([self.config["target"]]).
                to(self.config["device"])
            self.text_features = self.CLIP.encode_text(self.tokens
                ).detach()
        if config["task"] == "img2txt":
            image = clip_preprocess(Image.open(self.config["target
                "])).unsqueeze(0).to(self.config["device"])
            self.image_features = self.CLIP.encode_image(image)

    def generate(self, ls, minibatch=None):
        z = ls()
        result = self.model.generate(*z, minibatch=minibatch)
        if hasattr(self.config, "norm"):
            result = self.config["norm"](result)
        return result

    def discriminate(self, images, minibatch=None):
        images = self.config["denorm"](images)
        return self.model.discriminate(images, minibatch)

    def has_discriminator(self):
        return self.model.has_discriminator()

    def clip_similarity(self, input):
        if self.config["task"] == "txt2img":
            image = kornia.resize(input, (224, 224))
            if self.augmentation is not None:
                image = self.augmentation(image)

            image_features = self.CLIP.encode_image(image)

            sim = torch.cosine_similarity(image_features, self.
                text_features)
        elif self.config["task"] == "img2txt":
            try:
                text_tokens = clip.tokenize(input).to(self.config
                    ["device"])
            except:
                return torch.zeros(len(input))
```

```
            text_features = self.CLIP.encode_text(text_tokens)

            sim = torch.cosine_similarity(text_features, self.
                image_features)
        return sim


    def save(self, input, path):
        if self.config["task"] == "txt2img":
            if input.shape[0] > 1:
                save_grid(input.detach().cpu(), path)
            else:
                save_image(input[0], path)
        elif self.config["task"] == "img2txt":
            f = open(path, "w")
            f.write("\n".join(input))
            f.close()


# In[35]:


config = dict(
        device = "cuda",
        config = "DeepMindBigGAN256",
        generations = 550,
        save_each = 5,
        tmp_folder = "./tmp",
        target = "a woman man and a dog standing in the snow",
)


# In[36]:


config.update(get_config(config["config"]))


# In[37]:


config


# In[ ]:




# In[38]:


iteration = 0
def save_callback(algorithm):
    global iteration
    global config

    iteration += 1
```

```
    if iteration % config["save_each"] == 0 or iteration == config
        ["generations"]:
        if config["problem_args"]["n_obj"] == 1:
            sortedpop = sorted(algorithm.pop, key=lambda p: p.F)
            X = np.stack([p.X for p in sortedpop])
        else:
            X = algorithm.pop.get("X")

        ls = config["latent"](config)
        ls.set_from_population(X)

        with torch.no_grad():
            generated = algorithm.problem.generator.generate(ls,
                minibatch=config["batch_size"])
            if config["task"] == "txt2img":
                ext = "jpg"
            elif config["task"] == "img2txt":
                ext = "txt"
            name = "genetic-it-%d.%s" % (iteration, ext) if
                iteration < config["generations"] else "genetic-it
                -final.%s" % (ext, )
            algorithm.problem.generator.save(generated, os.path.
                join(config["tmp_folder"], name))


problem = GenerationProblem(config)
operators = get_operators(config)

if not os.path.exists(config["tmp_folder"]): os.mkdir(config["
    tmp_folder"])

algorithm = get_algorithm(
    config["algorithm"],
    pop_size=config["pop_size"],
    sampling=operators["sampling"],
    crossover=operators["crossover"],
    mutation=operators["mutation"],
    eliminate_duplicates=True,
    callback=save_callback,
    **(config["algorithm_args"][config["algorithm"]] if "
        algorithm_args" in config and config["algorithm"] in
        config["algorithm_args"] else dict())
)

res = minimize(
    problem,
    algorithm,
    ("n_gen", config["generations"]),
    save_history=False,
    verbose=True,
)


pickle.dump(dict(
    X = res.X,
    F = res.F,
    G = res.G,
    CV = res.CV,
```

```
), open(os.path.join(config["tmp_folder"], "genetic_result"), "wb
    "))

if config["problem_args"]["n_obj"] == 2:
    plot = Scatter(labels=["similarity", "discriminator",])
    plot.add(res.F, color="red")
    plot.save(os.path.join(config["tmp_folder"], "F.jpg"))


if config["problem_args"]["n_obj"] == 1:
    sortedpop = sorted(res.pop, key=lambda p: p.F)
    X = np.stack([p.X for p in sortedpop])
else:
    X = res.pop.get("X")

ls = config["latent"](config)
ls.set_from_population(X)

torch.save(ls.state_dict(), os.path.join(config["tmp_folder"], "
    ls_result"))

if config["problem_args"]["n_obj"] == 1:
    X = np.atleast_2d(res.X)
else:
    try:
        result = get_decision_making("pseudo-weights", [0, 1]).do(
            res.F)
    except:
        print("Warning: cant use pseudo-weights")
        result = get_decomposition("asf").do(res.F, [0, 1]).argmin
            ()

    X = res.X[result]
    X = np.atleast_2d(X)

ls.set_from_population(X)

with torch.no_grad():
    generated = problem.generator.generate(ls)

if config["task"] == "txt2img":
    ext = "jpg"
elif config["task"] == "img2txt":
    ext = "txt"

problem.generator.save(generated, os.path.join(config["tmp_folder
    "], "output.%s" % (ext)))


import gc


# In[ ]:


torch.cuda.empty_cache()
gc.collect()
```

B.4 Stable diffusion

Stable diffusion fine-tuning:

```
import argparse
import copy
import logging
import math
import os
import random
from pathlib import Path
from typing import Iterable, Optional

import numpy as np
import torch
import torch.nn.functional as F
import torch.utils.checkpoint

import datasets
import diffusers
import transformers
from accelerate import Accelerator
from accelerate.logging import get_logger
from accelerate.utils import set_seed
from datasets import load_dataset
from diffusers import AutoencoderKL, DDPMScheduler,
    StableDiffusionPipeline, UNet2DConditionModel
from diffusers.optimization import get_scheduler
from diffusers.utils import check_min_version
from diffusers.utils.import_utils import is_xformers_available
from huggingface_hub import HfFolder, Repository, whoami
from torchvision import transforms
from tqdm.auto import tqdm
from transformers import CLIPTextModel, CLIPTokenizer


# Will error if the minimal version of diffusers is not installed.
    Remove at your own risks.
check_min_version("0.10.0.dev0")

logger = get_logger(__name__, log_level="INFO")


def parse_args():
    parser = argparse.ArgumentParser(description="Simple example
        of a training script.")
    parser.add_argument(
        "--pretrained_model_name_or_path",
        type=str,
        default=None,
        required=True,
        help="Path to pretrained model or model identifier from
            huggingface.co/models.",
    )
    parser.add_argument(
        "--revision",
        type=str,
        default=None,
        required=False,
```

```
    help="Revision of pretrained model identifier from
        huggingface.co/models.",
)
parser.add_argument(
    "--dataset_name",
    type=str,
    default=None,
    help=(
        "The name of the Dataset (from the HuggingFace hub) to
            train on (could be your own, possibly private,"
        " dataset). It can also be a path pointing to a local
            copy of a dataset in your filesystem,"
        " or to a folder containing files that Datasets can
            understand."
    ),
)
parser.add_argument(
    "--dataset_config_name",
    type=str,
    default=None,
    help="The config of the Dataset, leave as None if there's
        only one config.",
)
parser.add_argument(
    "--train_data_dir",
    type=str,
    default=None,
    help=(
        "A folder containing the training data. Folder
            contents must follow the structure described in"
        " https://huggingface.co/docs/datasets/image_dataset#
            imagefolder. In particular, a `metadata.jsonl`
            file"
        " must exist to provide the captions for the images.
            Ignored if `dataset_name` is specified."
    ),
)
parser.add_argument(
    "--image_column", type=str, default="image", help="The
        column of the dataset containing an image."
)
parser.add_argument(
    "--caption_column",
    type=str,
    default="text",
    help="The column of the dataset containing a caption or a
        list of captions.",
)
parser.add_argument(
    "--max_train_samples",
    type=int,
    default=None,
    help=(
        "For debugging purposes or quicker training, truncate
            the number of training examples to this "
        "value if set."
    ),
)
parser.add_argument(
```

```
    "--output_dir",
    type=str,
    default="sd-model-finetuned",
    help="The output directory where the model predictions and
        checkpoints will be written.",
)
parser.add_argument(
    "--cache_dir",
    type=str,
    default=None,
    help="The directory where the downloaded models and
       datasets will be stored.",
)
parser.add_argument("--seed", type=int, default=None, help="A
   seed for reproducible training.")
parser.add_argument(
    "--resolution",
    type=int,
    default=512,
    help=(
        "The resolution for input images, all the images in
           the train/validation dataset will be resized to
           this"
        " resolution"
    ),
)
parser.add_argument(
    "--center_crop",
    action="store_true",
    help="Whether to center crop images before resizing to
       resolution (if not set, random crop will be used)",
)
parser.add_argument(
    "--random_flip",
    action="store_true",
    help="whether to randomly flip images horizontally",
)
parser.add_argument(
    "--train_batch_size", type=int, default=16, help="Batch
       size (per device) for the training dataloader."
)
parser.add_argument("--num_train_epochs", type=int, default
   =100)
parser.add_argument(
    "--max_train_steps",
    type=int,
    default=None,
    help="Total number of training steps to perform.  If
       provided, overrides num_train_epochs.",
)
parser.add_argument(
    "--gradient_accumulation_steps",
    type=int,
    default=1,
    help="Number of updates steps to accumulate before
       performing a backward/update pass.",
)
parser.add_argument(
    "--gradient_checkpointing",
```

```
    action="store_true",
    help="Whether or not to use gradient checkpointing to save
        memory at the expense of slower backward pass.",
)
parser.add_argument(
    "--learning_rate",
    type=float,
    default=1e-4,
    help="Initial learning rate (after the potential warmup
       period) to use.",
)
parser.add_argument(
    "--scale_lr",
    action="store_true",
    default=False,
    help="Scale the learning rate by the number of GPUs,
       gradient accumulation steps, and batch size.",
)
parser.add_argument(
    "--lr_scheduler",
    type=str,
    default="constant",
    help=(
        'The scheduler type to use. Choose between ["linear",
           "cosine", "cosine_with_restarts", "polynomial",'
        ' "constant", "constant_with_warmup"]'
    ),
)
parser.add_argument(
    "--lr_warmup_steps", type=int, default=500, help="Number
       of steps for the warmup in the lr scheduler."
)
parser.add_argument(
    "--use_8bit_adam", action="store_true", help="Whether or
       not to use 8-bit Adam from bitsandbytes."
)
parser.add_argument(
    "--allow_tf32",
    action="store_true",
    help=(
        "Whether or not to allow TF32 on Ampere GPUs. Can be
           used to speed up training. For more information,
           see"
        " https://pytorch.org/docs/stable/notes/cuda.html#
           tensorfloat-32-tf32-on-ampere-devices"
    ),
)
parser.add_argument("--use_ema", action="store_true", help="
   Whether to use EMA model.")
parser.add_argument(
    "--non_ema_revision",
    type=str,
    default=None,
    required=False,
    help=(
        "Revision of pretrained non-ema model identifier. Must
            be a branch, tag or git identifier of the local
           or"
```

```
        " remote repository specified with --
            pretrained_model_name_or_path."
    ),
)
parser.add_argument("--adam_beta1", type=float, default=0.9,
    help="The beta1 parameter for the Adam optimizer.")
parser.add_argument("--adam_beta2", type=float, default=0.999,
     help="The beta2 parameter for the Adam optimizer.")
parser.add_argument("--adam_weight_decay", type=float, default
    =1e-2, help="Weight decay to use.")
parser.add_argument("--adam_epsilon", type=float, default=1e
    -08, help="Epsilon value for the Adam optimizer")
parser.add_argument("--max_grad_norm", default=1.0, type=float
    , help="Max gradient norm.")
parser.add_argument("--push_to_hub", action="store_true", help
    ="Whether or not to push the model to the Hub.")
parser.add_argument("--hub_token", type=str, default=None,
    help="The token to use to push to the Model Hub.")
parser.add_argument(
    "--hub_model_id",
    type=str,
    default=None,
    help="The name of the repository to keep in sync with the
        local `output_dir`.",
)
parser.add_argument(
    "--logging_dir",
    type=str,
    default="logs",
    help=(
        "[TensorBoard](https://www.tensorflow.org/tensorboard)
            log directory. Will default to"
        " *output_dir/runs/**CURRENT_DATETIME_HOSTNAME***."
    ),
)
parser.add_argument(
    "--mixed_precision",
    type=str,
    default=None,
    choices=["no", "fp16", "bf16"],
    help=(
        "Whether to use mixed precision. Choose between fp16
            and bf16 (bfloat16). Bf16 requires PyTorch >="
        " 1.10.and an Nvidia Ampere GPU.  Default to the value
            of accelerate config of the current system or the
            "
        " flag passed with the `accelerate.launch` command.
            Use this argument to override the accelerate
            config."
    ),
)
parser.add_argument(
    "--report_to",
    type=str,
    default="tensorboard",
    help=(
        'The integration to report the results and logs to.
            Supported platforms are `"tensorboard"`'
```

```
            ' (default), `"wandb"` and `"comet_ml"`. Use `"all"`
                to report to all integrations.'
        ),
    )
    parser.add_argument("--local_rank", type=int, default=-1, help
        ="For distributed training: local_rank")
    parser.add_argument(
        "--checkpointing_steps",
        type=int,
        default=500,
        help=(
            "Save a checkpoint of the training state every X
                updates. These checkpoints are only suitable for
                resuming"
            " training using `--resume_from_checkpoint`."
        ),
    )
    parser.add_argument(
        "--resume_from_checkpoint",
        type=str,
        default=None,
        help=(
            "Whether training should be resumed from a previous
                checkpoint. Use a path saved by"
            ' `--checkpointing_steps`, or `"latest"` to
                automatically select the last available checkpoint
                .'
        ),
    )
    parser.add_argument(
        "--enable_xformers_memory_efficient_attention", action="
            store_true", help="Whether or not to use xformers."
    )

    args = parser.parse_args()
    env_local_rank = int(os.environ.get("LOCAL_RANK", -1))
    if env_local_rank != -1 and env_local_rank != args.local_rank:
        args.local_rank = env_local_rank

    # Sanity checks
    if args.dataset_name is None and args.train_data_dir is None:
        raise ValueError("Need either a dataset name or a training
             folder.")

    # default to using the same revision for the non-ema model if
        not specified
    if args.non_ema_revision is None:
        args.non_ema_revision = args.revision

    return args


def get_full_repo_name(model_id: str, organization: Optional[str]
    = None, token: Optional[str] = None):
    if token is None:
        token = HfFolder.get_token()
    if organization is None:
        username = whoami(token)["name"]
        return f"{username}/{model_id}"
```

```
    else:
        return f"{organization}/{model_id}"


dataset_name_mapping = {
    "lambdalabs/pokemon-blip-captions": ("image", "text"),
}


# Adapted from torch-ema https://github.com/fadel/pytorch_ema/blob
    /master/torch_ema/ema.py#L14
class EMAModel:
    """
    Exponential Moving Average of models weights
    """

    def __init__(self, parameters: Iterable[torch.nn.Parameter],
        decay=0.9999):
        parameters = list(parameters)
        self.shadow_params = [p.clone().detach() for p in
            parameters]

        self.collected_params = None

        self.decay = decay
        self.optimization_step = 0

    @torch.no_grad()
    def step(self, parameters):
        parameters = list(parameters)

        self.optimization_step += 1

        # Compute the decay factor for the exponential moving
            average.
        value = (1 + self.optimization_step) / (10 + self.
            optimization_step)
        one_minus_decay = 1 - min(self.decay, value)

        for s_param, param in zip(self.shadow_params, parameters):
            if param.requires_grad:
                s_param.sub_(one_minus_decay * (s_param - param))
            else:
                s_param.copy_(param)

        torch.cuda.empty_cache()

    def copy_to(self, parameters: Iterable[torch.nn.Parameter]) ->
        None:
        """
        Copy current averaged parameters into given collection of
            parameters.

        Args:
            parameters: Iterable of `torch.nn.Parameter`; the
                parameters to be
                updated with the stored moving averages. If `None
                    `, the
```

```
            parameters with which this `
                ExponentialMovingAverage` was
            initialized will be used.
        """
        parameters = list(parameters)
        for s_param, param in zip(self.shadow_params, parameters):
            param.data.copy_(s_param.data)

    def to(self, device=None, dtype=None) -> None:
        r"""Move internal buffers of the ExponentialMovingAverage
            to `device`.

        Args:
            device: like `device` argument to `torch.Tensor.to`
        """
        # .to() on the tensors handles None correctly
        self.shadow_params = [
            p.to(device=device, dtype=dtype) if p.
                is_floating_point() else p.to(device=device)
            for p in self.shadow_params
        ]

    def state_dict(self) -> dict:
        r"""
        Returns the state of the ExponentialMovingAverage as a
            dict.
        This method is used by accelerate during checkpointing to
            save the ema state dict.
        """
        # Following PyTorch conventions, references to tensors are
            returned:
        # "returns a reference to the state and not its copy!" -
        # https://pytorch.org/tutorials/beginner/
            saving_loading_models.html#what-is-a-state-dict
        return {
            "decay": self.decay,
            "optimization_step": self.optimization_step,
            "shadow_params": self.shadow_params,
            "collected_params": self.collected_params,
        }

    def load_state_dict(self, state_dict: dict) -> None:
        r"""
        Loads the ExponentialMovingAverage state.
        This method is used by accelerate during checkpointing to
            save the ema state dict.
        Args:
            state_dict (dict): EMA state. Should be an object
                returned
                from a call to :meth:`state_dict`.
        """
        # deepcopy, to be consistent with module API
        state_dict = copy.deepcopy(state_dict)

        self.decay = state_dict["decay"]
        if self.decay < 0.0 or self.decay > 1.0:
            raise ValueError("Decay must be between 0 and 1")

        self.optimization_step = state_dict["optimization_step"]
```

```
        if not isinstance(self.optimization_step, int):
            raise ValueError("Invalid optimization_step")

        self.shadow_params = state_dict["shadow_params"]
        if not isinstance(self.shadow_params, list):
            raise ValueError("shadow_params must be a list")
        if not all(isinstance(p, torch.Tensor) for p in self.
            shadow_params):
            raise ValueError("shadow_params must all be Tensors")

        self.collected_params = state_dict["collected_params"]
        if self.collected_params is not None:
            if not isinstance(self.collected_params, list):
                raise ValueError("collected_params must be a list
                    ")
            if not all(isinstance(p, torch.Tensor) for p in self.
                collected_params):
                raise ValueError("collected_params must all be
                    Tensors")
            if len(self.collected_params) != len(self.
                shadow_params):
                raise ValueError("collected_params and
                    shadow_params must have the same length")


def main():
    args = parse_args()
    logging_dir = os.path.join(args.output_dir, args.logging_dir)

    accelerator = Accelerator(
        gradient_accumulation_steps=args.
            gradient_accumulation_steps,
        mixed_precision=args.mixed_precision,
        log_with=args.report_to,
        logging_dir=logging_dir,
    )

    # Make one log on every process with the configuration for
        debugging.
    logging.basicConfig(
        format="%(asctime)s - %(levelname)s - %(name)s - %(message
            )s",
        datefmt="%m/%d/%Y %H:%M:%S",
        level=logging.INFO,
    )
    logger.info(accelerator.state, main_process_only=False)
    if accelerator.is_local_main_process:
        datasets.utils.logging.set_verbosity_warning()
        transformers.utils.logging.set_verbosity_warning()
        diffusers.utils.logging.set_verbosity_info()
    else:
        datasets.utils.logging.set_verbosity_error()
        transformers.utils.logging.set_verbosity_error()
        diffusers.utils.logging.set_verbosity_error()

    # If passed along, set the training seed now.
    if args.seed is not None:
        set_seed(args.seed)
```

```
# Handle the repository creation
if accelerator.is_main_process:
    if args.push_to_hub:
        if args.hub_model_id is None:
            repo_name = get_full_repo_name(Path(args.
                output_dir).name, token=args.hub_token)
        else:
            repo_name = args.hub_model_id
        repo = Repository(args.output_dir, clone_from=
            repo_name)

        with open(os.path.join(args.output_dir, ".gitignore"),
            "w+") as gitignore:
            if "step_*" not in gitignore:
                gitignore.write("step_*\n")
            if "epoch_*" not in gitignore:
                gitignore.write("epoch_*\n")
    elif args.output_dir is not None:
        os.makedirs(args.output_dir, exist_ok=True)

# Load scheduler, tokenizer and models.
noise_scheduler = DDPMScheduler.from_pretrained(args.
    pretrained_model_name_or_path, subfolder="scheduler")
tokenizer = CLIPTokenizer.from_pretrained(
    args.pretrained_model_name_or_path, subfolder="tokenizer",
        revision=args.revision
)
text_encoder = CLIPTextModel.from_pretrained(
    args.pretrained_model_name_or_path, subfolder="
        text_encoder", revision=args.revision
)
vae = AutoencoderKL.from_pretrained(args.
    pretrained_model_name_or_path, subfolder="vae", revision=
    args.revision)
unet = UNet2DConditionModel.from_pretrained(
    args.pretrained_model_name_or_path, subfolder="unet",
        revision=args.non_ema_revision
)

# Freeze vae and text_encoder
vae.requires_grad_(False)
text_encoder.requires_grad_(False)

# Create EMA for the unet.
if args.use_ema:
    ema_unet = UNet2DConditionModel.from_pretrained(
        args.pretrained_model_name_or_path, subfolder="unet",
            revision=args.revision
    )
    ema_unet = EMAModel(ema_unet.parameters())

if args.enable_xformers_memory_efficient_attention:
    if is_xformers_available():
        unet.enable_xformers_memory_efficient_attention()
    else:
        raise ValueError("xformers is not available. Make sure
            it is installed correctly")

if args.gradient_checkpointing:
```

```
    unet.enable_gradient_checkpointing()

# Enable TF32 for faster training on Ampere GPUs,
# cf https://pytorch.org/docs/stable/notes/cuda.html#
    tensorfloat-32-tf32-on-ampere-devices
if args.allow_tf32:
    torch.backends.cuda.matmul.allow_tf32 = True

if args.scale_lr:
    args.learning_rate = (
        args.learning_rate * args.gradient_accumulation_steps
            * args.train_batch_size * accelerator.
            num_processes
    )

# Initialize the optimizer
if args.use_8bit_adam:
    try:
        import bitsandbytes as bnb
    except ImportError:
        raise ImportError(
            "Please install bitsandbytes to use 8-bit Adam.
                You can do so by running `pip install
                bitsandbytes`"
        )

    optimizer_cls = bnb.optim.AdamW8bit
else:
    optimizer_cls = torch.optim.AdamW

optimizer = optimizer_cls(
    unet.parameters(),
    lr=args.learning_rate,
    betas=(args.adam_beta1, args.adam_beta2),
    weight_decay=args.adam_weight_decay,
    eps=args.adam_epsilon,
)

# Get the datasets: you can either provide your own training
    and evaluation files (see below)
# or specify a Dataset from the hub (the dataset will be
    downloaded automatically from the datasets Hub).



# In distributed training, the load_dataset function
    guarantees that only one local process can concurrently
# download the dataset.
if args.dataset_name is not None:
    # Downloading and loading a dataset from the hub.
    # dataset = load_dataset("imagefolder", data_dir="C:\\
        Users\\Yi it Uz\\Downloads\\EEE443_StableDiffusion\\
        diffusers\\examples\\text_to_image", split="train")
    dataset = load_dataset(
        args.dataset_name,
        args.dataset_config_name,
        cache_dir=args.cache_dir,
    )
else:
```

```
    data_files = {}
    if args.train_data_dir is not None:
        data_files["train"] = os.path.join(args.train_data_dir
            , "**")
    # dataset = load_dataset("imagefolder", data_dir="C:\\
        Users\\Yi it Uz\\Downloads\\EEE443_StableDiffusion\\
        diffusers\\examples\\text_to_image", split="train")
    dataset = load_dataset(
        "imagefolder",
        data_files=data_files,
        cache_dir=args.cache_dir,
    )
    # See more about loading custom images at
    # https://huggingface.co/docs/datasets/v2.4.0/en/
        image_load#imagefolder

# Preprocessing the datasets.
# We need to tokenize inputs and targets.
# dataset = load_dataset("imagefolder", data_dir="C:\\Users\\
    Yi it Uz\\Downloads\\EEE443_StableDiffusion\\diffusers\\
    examples\\text_to_image", split="train")
# print("\n", dataset, "\n")
column_names = dataset["train"].column_names
# column_names = ["file_name", "text"]


# 6. Get the column names for input/target.
dataset_columns = dataset_name_mapping.get(args.dataset_name,
    None)
if args.image_column is None:
    image_column = dataset_columns[0] if dataset_columns is
        not None else column_names[0]
else:
    image_column = args.image_column
    if image_column not in column_names:
        raise ValueError(
            f"--image_column' value '{args.image_column}'
                needs to be one of: {', '.join(column_names)}"
        )
if args.caption_column is None:
    caption_column = dataset_columns[1] if dataset_columns is
        not None else column_names[1]
else:
    caption_column = args.caption_column
    if caption_column not in column_names:
        raise ValueError(
            f"--caption_column' value '{args.caption_column}'
                needs to be one of: {', '.join(column_names)}"
        )

# Preprocessing the datasets.
# We need to tokenize input captions and transform the images.
def tokenize_captions(examples, is_train=True):
    captions = []
    for caption in examples[caption_column]:
        if isinstance(caption, str):
            captions.append(caption)
        elif isinstance(caption, (list, np.ndarray)):
            # take a random caption if there are multiple
```

```
            captions.append(random.choice(caption) if is_train
                else caption[0])
        else:
            raise ValueError(
                f"Caption column `{caption_column}` should
                    contain either strings or lists of strings
                    ."
            )
    inputs = tokenizer(
        captions, max_length=tokenizer.model_max_length,
            padding="max_length", truncation=True,
            return_tensors="pt"
    )
    return inputs.input_ids

# Preprocessing the datasets.
train_transforms = transforms.Compose(
    [
        transforms.Resize(args.resolution, interpolation=
            transforms.InterpolationMode.BILINEAR),
        transforms.CenterCrop(args.resolution) if args.
            center_crop else transforms.RandomCrop(args.
            resolution),
        transforms.RandomHorizontalFlip() if args.random_flip
            else transforms.Lambda(lambda x: x),
        transforms.ToTensor(),
        transforms.Normalize([0.5], [0.5]),
    ]
)

def preprocess_train(examples):
    images = [image.convert("RGB") for image in examples[
        image_column]]
    examples["pixel_values"] = [train_transforms(image) for
        image in images]
    examples["input_ids"] = tokenize_captions(examples)
    return examples

with accelerator.main_process_first():
    if args.max_train_samples is not None:
        dataset["train"] = dataset["train"].shuffle(seed=args.
            seed).select(range(args.max_train_samples))
    # Set the training transforms
    train_dataset = dataset["train"].with_transform(
        preprocess_train)

def collate_fn(examples):
    pixel_values = torch.stack([example["pixel_values"] for
        example in examples])
    pixel_values = pixel_values.to(memory_format=torch.
        contiguous_format).float()
    input_ids = torch.stack([example["input_ids"] for example
        in examples])
    return {"pixel_values": pixel_values, "input_ids":
        input_ids}

# DataLoaders creation:
train_dataloader = torch.utils.data.DataLoader(
```

```
    train_dataset, shuffle=True, collate_fn=collate_fn,
        batch_size=args.train_batch_size
)

# Scheduler and math around the number of training steps.
overrode_max_train_steps = False
num_update_steps_per_epoch = math.ceil(len(train_dataloader) /
    args.gradient_accumulation_steps)
if args.max_train_steps is None:
    args.max_train_steps = args.num_train_epochs *
        num_update_steps_per_epoch
    overrode_max_train_steps = True

lr_scheduler = get_scheduler(
    args.lr_scheduler,
    optimizer=optimizer,
    num_warmup_steps=args.lr_warmup_steps * args.
        gradient_accumulation_steps,
    num_training_steps=args.max_train_steps * args.
        gradient_accumulation_steps,
)

# Prepare everything with our `accelerator`.
unet, optimizer, train_dataloader, lr_scheduler = accelerator.
    prepare(
    unet, optimizer, train_dataloader, lr_scheduler
)
if args.use_ema:
    accelerator.register_for_checkpointing(ema_unet)

# For mixed precision training we cast the text_encoder and
    vae weights to half-precision
# as these models are only used for inference, keeping weights
    in full precision is not required.
weight_dtype = torch.float32
if accelerator.mixed_precision == "fp16":
    weight_dtype = torch.float16
elif accelerator.mixed_precision == "bf16":
    weight_dtype = torch.bfloat16

# Move text_encode and vae to gpu and cast to weight_dtype
text_encoder.to(accelerator.device, dtype=weight_dtype)
vae.to(accelerator.device, dtype=weight_dtype)
if args.use_ema:
    ema_unet.to(accelerator.device)

# We need to recalculate our total training steps as the size
    of the training dataloader may have changed.
num_update_steps_per_epoch = math.ceil(len(train_dataloader) /
    args.gradient_accumulation_steps)
if overrode_max_train_steps:
    args.max_train_steps = args.num_train_epochs *
        num_update_steps_per_epoch
# Afterwards we recalculate our number of training epochs
args.num_train_epochs = math.ceil(args.max_train_steps /
    num_update_steps_per_epoch)

# We need to initialize the trackers we use, and also store
    our configuration.
```

```
# The trackers initializes automatically on the main process.
if accelerator.is_main_process:
    accelerator.init_trackers("text2image-fine-tune", config=
        vars(args))

# Train!
total_batch_size = args.train_batch_size * accelerator.
    num_processes * args.gradient_accumulation_steps

logger.info("***** Running training *****")
logger.info(f"  Num examples = {len(train_dataset)}")
logger.info(f"  Num Epochs = {args.num_train_epochs}")
logger.info(f"  Instantaneous batch size per device = {args.
    train_batch_size}")
logger.info(f"  Total train batch size (w. parallel,
    distributed & accumulation) = {total_batch_size}")
logger.info(f"  Gradient Accumulation steps = {args.
    gradient_accumulation_steps}")
logger.info(f"  Total optimization steps = {args.
    max_train_steps}")
global_step = 0
first_epoch = 0

# Potentially load in the weights and states from a previous
    save
if args.resume_from_checkpoint:
    if args.resume_from_checkpoint != "latest":
        path = os.path.basename(args.resume_from_checkpoint)
    else:
        # Get the most recent checkpoint
        dirs = os.listdir(args.output_dir)
        dirs = [d for d in dirs if d.startswith("checkpoint")]
        dirs = sorted(dirs, key=lambda x: int(x.split("-")[1])
            )
        path = dirs[-1]
    accelerator.print(f"Resuming from checkpoint {path}")
    accelerator.load_state(os.path.join(args.output_dir, path)
        )
    global_step = int(path.split("-")[1])

    first_epoch = global_step // num_update_steps_per_epoch
    resume_step = global_step % num_update_steps_per_epoch

# Only show the progress bar once on each machine.
progress_bar = tqdm(range(global_step, args.max_train_steps),
    disable=not accelerator.is_local_main_process)
progress_bar.set_description("Steps")

for epoch in range(first_epoch, args.num_train_epochs):
    unet.train()
    train_loss = 0.0
    for step, batch in enumerate(train_dataloader):
        # Skip steps until we reach the resumed step
        if args.resume_from_checkpoint and epoch ==
            first_epoch and step < resume_step:
            if step % args.gradient_accumulation_steps == 0:
                progress_bar.update(1)
            continue
```

```
with accelerator.accumulate(unet):
    # Convert images to latent space
    latents = vae.encode(batch["pixel_values"].to(
        weight_dtype)).latent_dist.sample()
    latents = latents * 0.18215

    # Sample noise that we'll add to the latents
    noise = torch.randn_like(latents)
    bsz = latents.shape[0]
    # Sample a random timestep for each image
    timesteps = torch.randint(0, noise_scheduler.
        num_train_timesteps, (bsz,), device=latents.
        device)
    timesteps = timesteps.long()

    # Add noise to the latents according to the noise
        magnitude at each timestep
    # (this is the forward diffusion process)
    noisy_latents = noise_scheduler.add_noise(latents,
         noise, timesteps)

    # Get the text embedding for conditioning
    encoder_hidden_states = text_encoder(batch["
        input_ids"])[0]

    # Get the target for loss depending on the
        prediction type
    if noise_scheduler.config.prediction_type == "
        epsilon":
         target = noise
    elif noise_scheduler.config.prediction_type == "
        v_prediction":
         target = noise_scheduler.get_velocity(latents,
             noise, timesteps)
    else:
         raise ValueError(f"Unknown prediction type {
            noise_scheduler.config.prediction_type}")

    # Predict the noise residual and compute loss
    model_pred = unet(noisy_latents, timesteps,
        encoder_hidden_states).sample
    loss = F.mse_loss(model_pred.float(), target.float
        (), reduction="mean")

    # Gather the losses across all processes for
        logging (if we use distributed training).
    avg_loss = accelerator.gather(loss.repeat(args.
        train_batch_size)).mean()
    train_loss += avg_loss.item() / args.
        gradient_accumulation_steps

    # Backpropagate
    accelerator.backward(loss)
    if accelerator.sync_gradients:
         accelerator.clip_grad_norm_(unet.parameters(),
             args.max_grad_norm)
    optimizer.step()
    lr_scheduler.step()
    optimizer.zero_grad()
```

```
            # Checks if the accelerator has performed an
                optimization step behind the scenes
            if accelerator.sync_gradients:
                if args.use_ema:
                    ema_unet.step(unet.parameters())
                progress_bar.update(1)
                global_step += 1
                accelerator.log({"train_loss": train_loss}, step=
                    global_step)
                train_loss = 0.0

                if global_step % args.checkpointing_steps == 0:
                    if accelerator.is_main_process:
                        save_path = os.path.join(args.output_dir,
                            f"checkpoint-{global_step}")
                        accelerator.save_state(save_path)
                        logger.info(f"Saved state to {save_path}")

            logs = {"step_loss": loss.detach().item(), "lr":
                lr_scheduler.get_last_lr()[0]}
            progress_bar.set_postfix(**logs)

            if global_step >= args.max_train_steps:
                break

    # Create the pipeline using the trained modules and save it.
    accelerator.wait_for_everyone()
    if accelerator.is_main_process:
        unet = accelerator.unwrap_model(unet)
        if args.use_ema:
            ema_unet.copy_to(unet.parameters())

        pipeline = StableDiffusionPipeline.from_pretrained(
            args.pretrained_model_name_or_path,
            text_encoder=text_encoder,
            vae=vae,
            unet=unet,
            revision=args.revision,
        )
        pipeline.save_pretrained(args.output_dir)

        if args.push_to_hub:
            repo.push_to_hub(commit_message="End of training",
                blocking=False, auto_lfs_prune=True)

    accelerator.end_training()


if __name__ == "__main__":
    main()
```

Run this in terminal to fine tune

```
accelerate launch train_text_to_image.py \
  --pretrained_model_name_or_path="CompVis/stable-diffusion-v1-4"
      \
  --train_data_dir="C:\Users\Yi it Uz\Downloads\LAST\diffusers" \
  --use_ema \
  --resolution=512 --center_crop --random_flip \
```

```
  --train_batch_size=1 \
  --gradient_accumulation_steps=4 \
  --gradient_checkpointing \
  --mixed_precision="fp16" \
  --max_train_steps=15000 \
  --learning_rate=1e-05 \
  --max_grad_norm=1 \
  --lr_scheduler="constant" --lr_warmup_steps=0 \
  --output_dir="my-model"
```

Generation from stable diffusion model

```
from diffusers import StableDiffusionPipeline

pipe = StableDiffusionPipeline.from_pretrained("runwayml/stable-
    diffusion-v1-5")

# disable the following line if you run on CPU
#pipe = pipe.to("cuda")

prompt = "a woman man and a dog standing in the snow"
image = pipe(prompt).images[0]

image.save("hugging_face_4th.png")
```

B.5 VQGAN+CLIP

VQGAN fine-tune and generate:

```
import argparse
import math
import random
# from email.policy import default
from urllib.request import urlopen
from tqdm import tqdm
import sys
import os

import matplotlib.pyplot as plt

# pip install taming-transformers doesn't work with Gumbel, but
    does not yet work with coco etc
# appending the path does work with Gumbel, but gives
    ModuleNotFoundError: No module named 'transformers' for coco
    etc
sys.path.append('taming-transformers')

from omegaconf import OmegaConf
from taming.models import cond_transformer, vqgan
#import taming.modules

import torch
from torch import nn, optim
from torch.nn import functional as F
from torchvision import transforms
from torchvision.transforms import functional as TF
from torch.cuda import get_device_properties
torch.backends.cudnn.benchmark = False          # NR: True is a
    bit faster, but can lead to OOM. False is more deterministic.
```

```
#torch.use_deterministic_algorithms(True)        # NR:
    grid_sampler_2d_backward_cuda does not have a deterministic
    implementation

from torch_optimizer import DiffGrad, AdamP

from CLIP import clip
import kornia.augmentation as K
import numpy as np
import imageio

from PIL import ImageFile, Image, PngImagePlugin, ImageChops
ImageFile.LOAD_TRUNCATED_IMAGES = True

from subprocess import Popen, PIPE
import re

# Supress warnings
import warnings
warnings.filterwarnings('ignore')


# Check for GPU and reduce the default image size if low VRAM
default_image_size = 256 #512  # >8GB VRAM
if not torch.cuda.is_available():
    default_image_size = 256  # no GPU found
elif get_device_properties(0).total_memory <= 2 ** 33:  # 2 ** 33
    = 8,589,934,592 bytes = 8 GB
    default_image_size = 256 #304  # <8GB VRAM

# Create the parser
vq_parser = argparse.ArgumentParser(description='Image generation
    using VQGAN+CLIP')

# Add the arguments
vq_parser.add_argument("-p",    "--prompts", type=str, help="Text
    prompts", default=None, dest='prompts')
vq_parser.add_argument("-ip",   "--image_prompts", type=str, help
    ="Image prompts / target image", default=[], dest='
    image_prompts')
vq_parser.add_argument("-i",    "--iterations", type=int, help="
    Number of iterations", default=500, dest='max_iterations')
vq_parser.add_argument("-se",   "--save_every", type=int, help="
    Save image iterations", default=50, dest='display_freq')
vq_parser.add_argument("-s",    "--size", nargs=2, type=int, help
    ="Image size (width height) (default: %(default)s)", default=[
    default_image_size,default_image_size], dest='size')
vq_parser.add_argument("-ii",   "--init_image", type=str, help="
    Initial image", default=None, dest='init_image')
vq_parser.add_argument("-in",   "--init_noise", type=str, help="
    Initial noise image (pixels or gradient)", default=None, dest
    ='init_noise')
vq_parser.add_argument("-iw",   "--init_weight", type=float, help
    ="Initial weight", default=0., dest='init_weight')
vq_parser.add_argument("-m",    "--clip_model", type=str, help="
    CLIP model (e.g. ViT-B/32, ViT-B/16)", default='ViT-B/32',
    dest='clip_model')
```

```
vq_parser.add_argument("-conf", "--vqgan_config", type=str, help="
    VQGAN config", default=f'checkpoints/vqgan_imagenet_f16_16384.
    yaml', dest='vqgan_config')
vq_parser.add_argument("-ckpt", "--vqgan_checkpoint", type=str,
    help="VQGAN checkpoint", default=f'checkpoints/
    vqgan_imagenet_f16_16384.ckpt', dest='vqgan_checkpoint')
vq_parser.add_argument("-nps", "--noise_prompt_seeds", nargs="*",
     type=int, help="Noise prompt seeds", default=[], dest='
    noise_prompt_seeds')
vq_parser.add_argument("-npw", "--noise_prompt_weights", nargs
    ="*", type=float, help="Noise prompt weights", default=[],
    dest='noise_prompt_weights')
vq_parser.add_argument("-lr", "--learning_rate", type=float,
    help="Learning rate", default=0.1, dest='step_size')
vq_parser.add_argument("-cutm", "--cut_method", type=str, help="
    Cut method", choices=['original','updated','nrupdated','
    updatedpooling','latest'], default='latest', dest='cut_method
    ')
vq_parser.add_argument("-cuts", "--num_cuts", type=int, help="
    Number of cuts", default=32, dest='cutn')
vq_parser.add_argument("-cutp", "--cut_power", type=float, help="
    Cut power", default=1., dest='cut_pow')
vq_parser.add_argument("-sd", "--seed", type=int, help="Seed",
    default=None, dest='seed')
vq_parser.add_argument("-opt", "--optimiser", type=str, help="
    Optimiser", choices=['Adam','AdamW','Adagrad','Adamax','
    DiffGrad','AdamP','RAdam','RMSprop'], default='Adam', dest='
    optimiser')
vq_parser.add_argument("-o", "--output", type=str, help="Output
     image filename", default="output.png", dest='output')
vq_parser.add_argument("-vid", "--video", action='store_true',
    help="Create video frames?", dest='make_video')
vq_parser.add_argument("-zvid", "--zoom_video", action='store_true
    ', help="Create zoom video?", dest='make_zoom_video')
vq_parser.add_argument("-zs", "--zoom_start", type=int, help="
    Zoom start iteration", default=0, dest='zoom_start')
vq_parser.add_argument("-zse", "--zoom_save_every", type=int,
    help="Save zoom image iterations", default=10, dest='
    zoom_frequency')
vq_parser.add_argument("-zsc", "--zoom_scale", type=float, help="
    Zoom scale %%", default=0.99, dest='zoom_scale')
vq_parser.add_argument("-zsx", "--zoom_shift_x", type=int, help="
    Zoom shift x (left/right) amount in pixels", default=0, dest='
    zoom_shift_x')
vq_parser.add_argument("-zsy", "--zoom_shift_y", type=int, help="
    Zoom shift y (up/down) amount in pixels", default=0, dest='
    zoom_shift_y')
vq_parser.add_argument("-cpe", "--change_prompt_every", type=int,
     help="Prompt change frequency", default=0, dest='
    prompt_frequency')
vq_parser.add_argument("-vl", "--video_length", type=float, help
    ="Video length in seconds (not interpolated)", default=10,
    dest='video_length')
vq_parser.add_argument("-ofps", "--output_video_fps", type=float,
    help="Create an interpolated video (Nvidia GPU only) with this
     fps (min 10. best set to 30 or 60)", default=0, dest='
    output_video_fps')
vq_parser.add_argument("-ifps", "--input_video_fps", type=float,
    help="When creating an interpolated video, use this as the
```

```
    input fps to interpolate from (>0 & <ofps)", default=15, dest
    ='input_video_fps')
vq_parser.add_argument("-d",    "--deterministic", action='
    store_true', help="Enable cudnn.deterministic?", dest='
    cudnn_determinism')
vq_parser.add_argument("-aug",  "--augments", nargs='+', action='
    append', type=str, choices=['Ji','Sh','Gn','Pe','Ro','Af','Et
    ','Ts','Cr','Er','Re'], help="Enabled augments (latest vut
    method only)", default=[], dest='augments')
vq_parser.add_argument("-vsd",  "--video_style_dir", type=str,
    help="Directory with video frames to style", default=None,
    dest='video_style_dir')
vq_parser.add_argument("-cd",   "--cuda_device", type=str, help="
    Cuda device to use", default="cuda:0", dest='cuda_device')




# Execute the parse_args() method
args = vq_parser.parse_args()

if not args.prompts and not args.image_prompts:
    args.prompts = "A cute, smiling, Nerdy Rodent"

if args.cudnn_determinism:
   torch.backends.cudnn.deterministic = True

if not args.augments:
   args.augments = [['Af', 'Pe', 'Ji', 'Er']]

# Split text prompts using the pipe character (weights are split
    later)
if args.prompts:
    # For stories, there will be many phrases
    story_phrases = [phrase.strip() for phrase in args.prompts.
        split("^")]

    # Make a list of all phrases
    all_phrases = []
    for phrase in story_phrases:
        all_phrases.append(phrase.split("|"))

    # First phrase
    args.prompts = all_phrases[0]

# Split target images using the pipe character (weights are split
    later)
if args.image_prompts:
    args.image_prompts = args.image_prompts.split("|")
    args.image_prompts = [image.strip() for image in args.
        image_prompts]

if args.make_video and args.make_zoom_video:
    print("Warning: Make video and make zoom video are mutually
        exclusive.")
    args.make_video = False

# Make video steps directory
```

```
if args.make_video or args.make_zoom_video:
    if not os.path.exists('steps'):
        os.mkdir('steps')

# Fallback to CPU if CUDA is not found and make sure GPU video
    rendering is also disabled
# NB. May not work for AMD cards?
if not args.cuda_device == 'cpu' and not torch.cuda.is_available()
    :
    args.cuda_device = 'cpu'
    args.video_fps = 0
    print("Warning: No GPU found! Using the CPU instead. The
        iterations will be slow.")
    print("Perhaps CUDA/ROCm or the right pytorch version is not
        properly installed?")

# If a video_style_dir has been, then create a list of all the
    images
if args.video_style_dir:
    print("Locating video frames...")
    video_frame_list = []
    for entry in os.scandir(args.video_style_dir):
        if (entry.path.endswith(".jpg")
                or entry.path.endswith(".png")) and entry.is_file
                    ():
            video_frame_list.append(entry.path)

    # Reset a few options - same filename, different directory
    if not os.path.exists('steps'):
        os.mkdir('steps')

    args.init_image = video_frame_list[0]
    filename = os.path.basename(args.init_image)
    cwd = os.getcwd()
    args.output = os.path.join(cwd, "steps", filename)
    num_video_frames = len(video_frame_list) # for video styling


# Various functions and classes
def sinc(x):
    return torch.where(x != 0, torch.sin(math.pi * x) / (math.pi *
        x), x.new_ones([]))


def lanczos(x, a):
    cond = torch.logical_and(-a < x, x < a)
    out = torch.where(cond, sinc(x) * sinc(x/a), x.new_zeros([]))
    return out / out.sum()


def ramp(ratio, width):
    n = math.ceil(width / ratio + 1)
    out = torch.empty([n])
    cur = 0
    for i in range(out.shape[0]):
        out[i] = cur
        cur += ratio
    return torch.cat([-out[1:].flip([0]), out])[1:-1]
```

```
# For zoom video
def zoom_at(img, x, y, zoom):
    w, h = img.size
    zoom2 = zoom * 2
    img = img.crop((x - w / zoom2, y - h / zoom2,
                    x + w / zoom2, y + h / zoom2))
    return img.resize((w, h), Image.LANCZOS)


# NR: Testing with different intital images
def random_noise_image(w,h):
    random_image = Image.fromarray(np.random.randint(0,255,(w,h,3)
        ,dtype=np.dtype('uint8')))
    return random_image


# create initial gradient image
def gradient_2d(start, stop, width, height, is_horizontal):
    if is_horizontal:
        return np.tile(np.linspace(start, stop, width), (height,
            1))
    else:
        return np.tile(np.linspace(start, stop, height), (width,
            1)).T


def gradient_3d(width, height, start_list, stop_list,
    is_horizontal_list):
    result = np.zeros((height, width, len(start_list)), dtype=
        float)

    for i, (start, stop, is_horizontal) in enumerate(zip(
        start_list, stop_list, is_horizontal_list)):
        result[:, :, i] = gradient_2d(start, stop, width, height,
            is_horizontal)

    return result


def random_gradient_image(w,h):
    array = gradient_3d(w, h, (0, 0, np.random.randint(0,255)), (
        np.random.randint(1,255), np.random.randint(2,255), np.
        random.randint(3,128)), (True, False, False))
    random_image = Image.fromarray(np.uint8(array))
    return random_image


# Used in older MakeCutouts
def resample(input, size, align_corners=True):
    n, c, h, w = input.shape
    dh, dw = size

    input = input.view([n * c, 1, h, w])

    if dh < h:
        kernel_h = lanczos(ramp(dh / h, 2), 2).to(input.device,
            input.dtype)
        pad_h = (kernel_h.shape[0] - 1) // 2
```

```
        input = F.pad(input, (0, 0, pad_h, pad_h), 'reflect')
        input = F.conv2d(input, kernel_h[None, None, :, None])

    if dw < w:
        kernel_w = lanczos(ramp(dw / w, 2), 2).to(input.device,
            input.dtype)
        pad_w = (kernel_w.shape[0] - 1) // 2
        input = F.pad(input, (pad_w, pad_w, 0, 0), 'reflect')
        input = F.conv2d(input, kernel_w[None, None, None, :])

    input = input.view([n, c, h, w])
    return F.interpolate(input, size, mode='bicubic',
        align_corners=align_corners)


class ReplaceGrad(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x_forward, x_backward):
        ctx.shape = x_backward.shape
        return x_forward

    @staticmethod
    def backward(ctx, grad_in):
        return None, grad_in.sum_to_size(ctx.shape)

replace_grad = ReplaceGrad.apply


class ClampWithGrad(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, min, max):
        ctx.min = min
        ctx.max = max
        ctx.save_for_backward(input)
        return input.clamp(min, max)

    @staticmethod
    def backward(ctx, grad_in):
        input, = ctx.saved_tensors
        return grad_in * (grad_in * (input - input.clamp(ctx.min,
            ctx.max)) >= 0), None, None

clamp_with_grad = ClampWithGrad.apply


def vector_quantize(x, codebook):
    d = x.pow(2).sum(dim=-1, keepdim=True) + codebook.pow(2).sum(
        dim=1) - 2 * x @ codebook.T
    indices = d.argmin(-1)
    x_q = F.one_hot(indices, codebook.shape[0]).to(d.dtype) @
        codebook
    return replace_grad(x_q, x)


class Prompt(nn.Module):
    def __init__(self, embed, weight=1., stop=float('-inf')):
        super().__init__()
        self.register_buffer('embed', embed)
        self.register_buffer('weight', torch.as_tensor(weight))
```

```
        self.register_buffer('stop', torch.as_tensor(stop))

    def forward(self, input):
        input_normed = F.normalize(input.unsqueeze(1), dim=2)
        embed_normed = F.normalize(self.embed.unsqueeze(0), dim=2)
        dists = input_normed.sub(embed_normed).norm(dim=2).div(2).
            arcsin().pow(2).mul(2)
        dists = dists * self.weight.sign()
        return self.weight.abs() * replace_grad(dists, torch.
            maximum(dists, self.stop)).mean()


#NR: Split prompts and weights
def split_prompt(prompt):
    vals = prompt.rsplit(':', 2)
    vals = vals + ['', '1', '-inf'][len(vals):]
    return vals[0], float(vals[1]), float(vals[2])


class MakeCutouts(nn.Module):
    def __init__(self, cut_size, cutn, cut_pow=1.):
        super().__init__()
        self.cut_size = cut_size
        self.cutn = cutn
        self.cut_pow = cut_pow # not used with pooling

        # Pick your own augments & their order
        augment_list = []
        for item in args.augments[0]:
            if item == 'Ji':
                augment_list.append(K.ColorJitter(brightness=0.1,
                    contrast=0.1, saturation=0.1, hue=0.1, p=0.7))
            elif item == 'Sh':
                augment_list.append(K.RandomSharpness(sharpness
                    =0.3, p=0.5))
            elif item == 'Gn':
                augment_list.append(K.RandomGaussianNoise(mean
                    =0.0, std=1., p=0.5))
            elif item == 'Pe':
                augment_list.append(K.RandomPerspective(
                    distortion_scale=0.7, p=0.7))
            elif item == 'Ro':
                augment_list.append(K.RandomRotation(degrees=15, p
                    =0.7))
            elif item == 'Af':
                augment_list.append(K.RandomAffine(degrees=15,
                    translate=0.1, shear=5, p=0.7, padding_mode='
                    zeros', keepdim=True)) # border, reflection,
                    zeros
            elif item == 'Et':
                augment_list.append(K.RandomElasticTransform(p
                    =0.7))
            elif item == 'Ts':
                augment_list.append(K.RandomThinPlateSpline(scale
                    =0.8, same_on_batch=True, p=0.7))
            elif item == 'Cr':
                augment_list.append(K.RandomCrop(size=(self.
                    cut_size,self.cut_size), pad_if_needed=True,
                    padding_mode='reflect', p=0.5))
```

```
            elif item == 'Er':
                augment_list.append(K.RandomErasing(scale=(.1, .4)
                    , ratio=(.3, 1/.3), same_on_batch=True, p=0.7)
                    )
            elif item == 'Re':
                augment_list.append(K.RandomResizedCrop(size=(self
                    .cut_size,self.cut_size), scale=(0.1,1),
                    ratio=(0.75,1.333), cropping_mode='resample',
                    p=0.5))

        self.augs = nn.Sequential(*augment_list)
        self.noise_fac = 0.1
        # self.noise_fac = False

        # Uncomment if you like seeing the list ;)
        # print(augment_list)

        # Pooling
        self.av_pool = nn.AdaptiveAvgPool2d((self.cut_size, self.
            cut_size))
        self.max_pool = nn.AdaptiveMaxPool2d((self.cut_size, self.
            cut_size))

    def forward(self, input):
        cutouts = []

        for _ in range(self.cutn):
            # Use Pooling
            cutout = (self.av_pool(input) + self.max_pool(input))
                /2
            cutouts.append(cutout)

        batch = self.augs(torch.cat(cutouts, dim=0))

        if self.noise_fac:
            facs = batch.new_empty([self.cutn, 1, 1, 1]).uniform_
                (0, self.noise_fac)
            batch = batch + facs * torch.randn_like(batch)
        return batch


# An updated version with Kornia augments and pooling (where my
    version started):
class MakeCutoutsPoolingUpdate(nn.Module):
    def __init__(self, cut_size, cutn, cut_pow=1.):
        super().__init__()
        self.cut_size = cut_size
        self.cutn = cutn
        self.cut_pow = cut_pow # Not used with pooling

        self.augs = nn.Sequential(
            K.RandomAffine(degrees=15, translate=0.1, p=0.7,
                padding_mode='border'),
            K.RandomPerspective(0.7,p=0.7),
            K.ColorJitter(hue=0.1, saturation=0.1, p=0.7),
            K.RandomErasing((.1, .4), (.3, 1/.3), same_on_batch=
                True, p=0.7),
        )
```

```
        self.noise_fac = 0.1
        self.av_pool = nn.AdaptiveAvgPool2d((self.cut_size, self.
            cut_size))
        self.max_pool = nn.AdaptiveMaxPool2d((self.cut_size, self.
            cut_size))

    def forward(self, input):
        sideY, sideX = input.shape[2:4]
        max_size = min(sideX, sideY)
        min_size = min(sideX, sideY, self.cut_size)
        cutouts = []

        for _ in range(self.cutn):
            cutout = (self.av_pool(input) + self.max_pool(input))
                /2
            cutouts.append(cutout)

        batch = self.augs(torch.cat(cutouts, dim=0))

        if self.noise_fac:
            facs = batch.new_empty([self.cutn, 1, 1, 1]).uniform_
                (0, self.noise_fac)
            batch = batch + facs * torch.randn_like(batch)
        return batch


# An Nerdy updated version with selectable Kornia augments, but no
    pooling:
class MakeCutoutsNRUpdate(nn.Module):
    def __init__(self, cut_size, cutn, cut_pow=1.):
        super().__init__()
        self.cut_size = cut_size
        self.cutn = cutn
        self.cut_pow = cut_pow
        self.noise_fac = 0.1

        # Pick your own augments & their order
        augment_list = []
        for item in args.augments[0]:
            if item == 'Ji':
                augment_list.append(K.ColorJitter(brightness=0.1,
                    contrast=0.1, saturation=0.1, hue=0.1, p=0.7))
            elif item == 'Sh':
                augment_list.append(K.RandomSharpness(sharpness
                    =0.3, p=0.5))
            elif item == 'Gn':
                augment_list.append(K.RandomGaussianNoise(mean
                    =0.0, std=1., p=0.5))
            elif item == 'Pe':
                augment_list.append(K.RandomPerspective(
                    distortion_scale=0.5, p=0.7))
            elif item == 'Ro':
                augment_list.append(K.RandomRotation(degrees=15, p
                    =0.7))
            elif item == 'Af':
                augment_list.append(K.RandomAffine(degrees=30,
                    translate=0.1, shear=5, p=0.7, padding_mode='
                    zeros', keepdim=True)) # border, reflection,
                    zeros
```

```
            elif item == 'Et':
                augment_list.append(K.RandomElasticTransform(p
                    =0.7))
            elif item == 'Ts':
                augment_list.append(K.RandomThinPlateSpline(scale
                    =0.8, same_on_batch=True, p=0.7))
            elif item == 'Cr':
                augment_list.append(K.RandomCrop(size=(self.
                    cut_size,self.cut_size), pad_if_needed=True,
                    padding_mode='reflect', p=0.5))
            elif item == 'Er':
                augment_list.append(K.RandomErasing(scale=(.1, .4)
                    , ratio=(.3, 1/.3), same_on_batch=True, p=0.7)
                    )
            elif item == 'Re':
                augment_list.append(K.RandomResizedCrop(size=(self
                    .cut_size,self.cut_size), scale=(0.1,1),
                    ratio=(0.75,1.333), cropping_mode='resample',
                    p=0.5))

        self.augs = nn.Sequential(*augment_list)


    def forward(self, input):
        sideY, sideX = input.shape[2:4]
        max_size = min(sideX, sideY)
        min_size = min(sideX, sideY, self.cut_size)
        cutouts = []
        for _ in range(self.cutn):
            size = int(torch.rand([])**self.cut_pow * (max_size -
                min_size) + min_size)
            offsetx = torch.randint(0, sideX - size + 1, ())
            offsety = torch.randint(0, sideY - size + 1, ())
            cutout = input[:, :, offsety:offsety + size, offsetx:
                offsetx + size]
            cutouts.append(resample(cutout, (self.cut_size, self.
                cut_size)))
        batch = self.augs(torch.cat(cutouts, dim=0))
        if self.noise_fac:
            facs = batch.new_empty([self.cutn, 1, 1, 1]).uniform_
                (0, self.noise_fac)
            batch = batch + facs * torch.randn_like(batch)
        return batch


# An updated version with Kornia augments, but no pooling:
class MakeCutoutsUpdate(nn.Module):
    def __init__(self, cut_size, cutn, cut_pow=1.):
        super().__init__()
        self.cut_size = cut_size
        self.cutn = cutn
        self.cut_pow = cut_pow
        self.augs = nn.Sequential(
            K.RandomHorizontalFlip(p=0.5),
            K.ColorJitter(hue=0.01, saturation=0.01, p=0.7),
            # K.RandomSolarize(0.01, 0.01, p=0.7),
            K.RandomSharpness(0.3,p=0.4),
            K.RandomAffine(degrees=30, translate=0.1, p=0.8,
                padding_mode='border'),
```

```
            K.RandomPerspective(0.2,p=0.4),)
        self.noise_fac = 0.1


    def forward(self, input):
        sideY, sideX = input.shape[2:4]
        max_size = min(sideX, sideY)
        min_size = min(sideX, sideY, self.cut_size)
        cutouts = []
        for _ in range(self.cutn):
            size = int(torch.rand([])**self.cut_pow * (max_size -
                min_size) + min_size)
            offsetx = torch.randint(0, sideX - size + 1, ())
            offsety = torch.randint(0, sideY - size + 1, ())
            cutout = input[:, :, offsety:offsety + size, offsetx:
                offsetx + size]
            cutouts.append(resample(cutout, (self.cut_size, self.
                cut_size)))
        batch = self.augs(torch.cat(cutouts, dim=0))
        if self.noise_fac:
            facs = batch.new_empty([self.cutn, 1, 1, 1]).uniform_
                (0, self.noise_fac)
            batch = batch + facs * torch.randn_like(batch)
        return batch


# This is the original version (No pooling)
class MakeCutoutsOrig(nn.Module):
    def __init__(self, cut_size, cutn, cut_pow=1.):
        super().__init__()
        self.cut_size = cut_size
        self.cutn = cutn
        self.cut_pow = cut_pow

    def forward(self, input):
        sideY, sideX = input.shape[2:4]
        max_size = min(sideX, sideY)
        min_size = min(sideX, sideY, self.cut_size)
        cutouts = []
        for _ in range(self.cutn):
            size = int(torch.rand([])**self.cut_pow * (max_size -
                min_size) + min_size)
            offsetx = torch.randint(0, sideX - size + 1, ())
            offsety = torch.randint(0, sideY - size + 1, ())
            cutout = input[:, :, offsety:offsety + size, offsetx:
                offsetx + size]
            cutouts.append(resample(cutout, (self.cut_size, self.
                cut_size)))
        return clamp_with_grad(torch.cat(cutouts, dim=0), 0, 1)


def load_vqgan_model(config_path, checkpoint_path):
    global gumbel
    gumbel = False
    config = OmegaConf.load(config_path)
    if config.model.target == 'taming.models.vqgan.VQModel':
        model = vqgan.VQModel(**config.model.params)
        model.eval().requires_grad_(False)
        model.init_from_ckpt(checkpoint_path)
```

```
    elif config.model.target == 'taming.models.vqgan.GumbelVQ':
        model = vqgan.GumbelVQ(**config.model.params)
        model.eval().requires_grad_(False)
        model.init_from_ckpt(checkpoint_path)
        gumbel = True
    elif config.model.target == 'taming.models.cond_transformer.
       Net2NetTransformer':
        parent_model = cond_transformer.Net2NetTransformer(**
           config.model.params)
        parent_model.eval().requires_grad_(False)
        parent_model.init_from_ckpt(checkpoint_path)
        model = parent_model.first_stage_model
    else:
        raise ValueError(f'unknown model type: {config.model.
           target}')
    del model.loss
    return model


def resize_image(image, out_size):
    ratio = image.size[0] / image.size[1]
    area = min(image.size[0] * image.size[1], out_size[0] *
       out_size[1])
    size = round((area * ratio)**0.5), round((area / ratio)**0.5)
    return image.resize(size, Image.LANCZOS)


# Do it
device = torch.device(args.cuda_device)
model = load_vqgan_model(args.vqgan_config, args.vqgan_checkpoint)
    .to(device)
jit = True if "1.7.1" in torch.__version__ else False
perceptor = clip.load(args.clip_model, jit=jit)[0].eval().
    requires_grad_(False).to(device)
check = torch.load('./model_fastai.pth', map_location='cpu')
perceptor.load_state_dict(check['model'])

# clock=deepcopy(perceptor.visual.positional_embedding.data)
# perceptor.visual.positional_embedding.data = clock/clock.max()
# perceptor.visual.positional_embedding.data=clamp_with_grad(clock
    ,0,1)

cut_size = perceptor.visual.input_resolution
f = 2**(model.decoder.num_resolutions - 1)

# Cutout class options:
# 'latest','original','updated' or 'updatedpooling'
if args.cut_method == 'latest':
    make_cutouts = MakeCutouts(cut_size, args.cutn, cut_pow=args.
       cut_pow)
elif args.cut_method == 'original':
    make_cutouts = MakeCutoutsOrig(cut_size, args.cutn, cut_pow=
       args.cut_pow)
elif args.cut_method == 'updated':
    make_cutouts = MakeCutoutsUpdate(cut_size, args.cutn, cut_pow=
       args.cut_pow)
elif args.cut_method == 'nrupdated':
    make_cutouts = MakeCutoutsNRUpdate(cut_size, args.cutn,
       cut_pow=args.cut_pow)
```

```
else:
    make_cutouts = MakeCutoutsPoolingUpdate(cut_size, args.cutn,
        cut_pow=args.cut_pow)

toksX, toksY = args.size[0] // f, args.size[1] // f
sideX, sideY = toksX * f, toksY * f

# Gumbel or not?
if gumbel:
    e_dim = 256
    n_toks = model.quantize.n_embed
    z_min = model.quantize.embed.weight.min(dim=0).values[None, :,
        None, None]
    z_max = model.quantize.embed.weight.max(dim=0).values[None, :,
        None, None]
else:
    e_dim = model.quantize.e_dim
    n_toks = model.quantize.n_e
    z_min = model.quantize.embedding.weight.min(dim=0).values[None
        , :, None, None]
    z_max = model.quantize.embedding.weight.max(dim=0).values[None
        , :, None, None]


if args.init_image:
    if 'http' in args.init_image:
      img = Image.open(urlopen(args.init_image))
    else:
      img = Image.open(args.init_image)
    pil_image = img.convert('RGB')
    pil_image = pil_image.resize((sideX, sideY), Image.LANCZOS)
    pil_tensor = TF.to_tensor(pil_image)
    z, *_ = model.encode(pil_tensor.to(device).unsqueeze(0) * 2 -
        1)
elif args.init_noise == 'pixels':
    img = random_noise_image(args.size[0], args.size[1])
    pil_image = img.convert('RGB')
    pil_image = pil_image.resize((sideX, sideY), Image.LANCZOS)
    pil_tensor = TF.to_tensor(pil_image)
    z, *_ = model.encode(pil_tensor.to(device).unsqueeze(0) * 2 -
        1)
elif args.init_noise == 'gradient':
    img = random_gradient_image(args.size[0], args.size[1])
    pil_image = img.convert('RGB')
    pil_image = pil_image.resize((sideX, sideY), Image.LANCZOS)
    pil_tensor = TF.to_tensor(pil_image)
    z, *_ = model.encode(pil_tensor.to(device).unsqueeze(0) * 2 -
        1)
else:
    one_hot = F.one_hot(torch.randint(n_toks, [toksY * toksX],
        device=device), n_toks).float()
    # z = one_hot @ model.quantize.embedding.weight
    if gumbel:
        z = one_hot @ model.quantize.embed.weight
    else:
        z = one_hot @ model.quantize.embedding.weight

    z = z.view([-1, toksY, toksX, e_dim]).permute(0, 3, 1, 2)
```

```
    #z = torch.rand_like(z)*2
                # NR: check

z_orig = z.clone()
z.requires_grad_(True)

pMs = []
normalize = transforms.Normalize(mean=[0.48145466, 0.4578275,
    0.40821073],
                                  std=[0.26862954, 0.26130258,
                                     0.27577711])

# From imagenet - Which is better?
#normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
#                                 std=[0.229, 0.224, 0.225])

# CLIP tokenize/encode
if args.prompts:
    for prompt in args.prompts:
        txt, weight, stop = split_prompt(prompt)
        embed = perceptor.encode_text(clip.tokenize(txt).to(device
            )).float()
        pMs.append(Prompt(embed, weight, stop).to(device))

for prompt in args.image_prompts:
    path, weight, stop = split_prompt(prompt)
    img = Image.open(path)
    pil_image = img.convert('RGB')
    img = resize_image(pil_image, (sideX, sideY))
    batch = make_cutouts(TF.to_tensor(img).unsqueeze(0).to(device)
        )
    embed = perceptor.encode_image(normalize(batch)).float()
    pMs.append(Prompt(embed, weight, stop).to(device))

for seed, weight in zip(args.noise_prompt_seeds, args.
    noise_prompt_weights):
    gen = torch.Generator().manual_seed(seed)
    embed = torch.empty([1, perceptor.visual.output_dim]).normal_(
        generator=gen)
    pMs.append(Prompt(embed, weight).to(device))


# Set the optimiser
def get_opt(opt_name, opt_lr):
    if opt_name == "Adam":
        opt = optim.Adam([z], lr=opt_lr)        # LR=0.1 (Default)
    elif opt_name == "AdamW":
        opt = optim.AdamW([z], lr=opt_lr)
    elif opt_name == "Adagrad":
        opt = optim.Adagrad([z], lr=opt_lr)
    elif opt_name == "Adamax":
        opt = optim.Adamax([z], lr=opt_lr)
    elif opt_name == "DiffGrad":
        opt = DiffGrad([z], lr=opt_lr, eps=1e-9, weight_decay=1e
            -9) # NR: Playing for reasons
    elif opt_name == "AdamP":
        opt = AdamP([z], lr=opt_lr)
    elif opt_name == "RAdam":
        opt = optim.RAdam([z], lr=opt_lr)
```

```
    elif opt_name == "RMSprop":
        opt = optim.RMSprop([z], lr=opt_lr)
    else:
        print("Unknown optimiser. Are choices broken?")
        opt = optim.Adam([z], lr=opt_lr)
    return opt

opt = get_opt(args.optimiser, args.step_size)


# Output for the user
print('Using device:', device)
print('Optimising using:', args.optimiser)

if args.prompts:
    print('Using text prompts:', args.prompts)
if args.image_prompts:
    print('Using image prompts:', args.image_prompts)
if args.init_image:
    print('Using initial image:', args.init_image)
if args.noise_prompt_weights:
    print('Noise prompt weights:', args.noise_prompt_weights)


if args.seed is None:
    seed = torch.seed()
else:
    seed = args.seed
torch.manual_seed(seed)
print('Using seed:', seed)


# Vector quantize
def synth(z):
    if gumbel:
        z_q = vector_quantize(z.movedim(1, 3), model.quantize.
            embed.weight).movedim(3, 1)
    else:
        z_q = vector_quantize(z.movedim(1, 3), model.quantize.
            embedding.weight).movedim(3, 1)
    return clamp_with_grad(model.decode(z_q).add(1).div(2), 0, 1)


#@torch.no_grad()
@torch.inference_mode()
def checkin(i, losses):
    losses_str = ', '.join(f'{loss.item():g}' for loss in losses)
    tqdm.write(f'i: {i}, loss: {sum(losses).item():g}, losses: {
        losses_str}')
    out = synth(z)
    info = PngImagePlugin.PngInfo()
    info.add_text('comment', f'{args.prompts}')
    TF.to_pil_image(out[0].cpu()).save(args.output, pnginfo=info)



def ascend_txt():
    global i
    out = synth(z)
```

```
    iii = perceptor.encode_image(normalize(make_cutouts(out))).
        float()

    result = []

    if args.init_weight:
        # result.append(F.mse_loss(z, z_orig) * args.init_weight /
            2)
        result.append(F.mse_loss(z, torch.zeros_like(z_orig)) *
            ((1/torch.tensor(i*2 + 1))*args.init_weight) / 2)

    for prompt in pMs:
        result.append(prompt(iii))

    if args.make_video:
        img = np.array(out.mul(255).clamp(0, 255)[0].cpu().detach
            ().numpy().astype(np.uint8))[:,:,:]
        img = np.transpose(img, (1, 2, 0))
        imageio.imwrite('./steps/' + str(i) + '.png', np.array(img
            ))

    return result # return loss


def train(i):
    global loss_list

    opt.zero_grad(set_to_none=True)
    lossAll = ascend_txt()

    if i % args.display_freq == 0:
        checkin(i, lossAll)

    loss = sum(lossAll)
    loss_list.append(loss.item())
    loss.backward()
    opt.step()

    #with torch.no_grad():
    with torch.inference_mode():
        z.copy_(z.maximum(z_min).minimum(z_max))



i = 0 # Iteration counter
j = 0 # Zoom video frame counter
p = 1 # Phrase counter
smoother = 0 # Smoother counter
this_video_frame = 0 # for video styling

# Messing with learning rate / optimisers
#variable_lr = args.step_size
#optimiser_list = [['Adam',0.075],['AdamW',0.125],['Adagrad
    ',0.2],['Adamax',0.125],['DiffGrad',0.075],['RAdam',0.125],['
    RMSprop',0.02]]

# Do it
try:
    loss_list = []
```

```
with tqdm() as pbar:
    while True:
        # Change generated image
        if args.make_zoom_video:
            if i % args.zoom_frequency == 0:
                out = synth(z)

                # Save image
                img = np.array(out.mul(255).clamp(0, 255)[0].
                    cpu().detach().numpy().astype(np.uint8))
                    [:,:,:]
                img = np.transpose(img, (1, 2, 0))
                imageio.imwrite('./steps/' + str(j) + '.png',
                    np.array(img))

                # Time to start zooming?
                if args.zoom_start <= i:
                    # Convert z back into a Pil image
                    #pil_image = TF.to_pil_image(out[0].cpu())

                    # Convert NP to Pil image
                    pil_image = Image.fromarray(np.array(img).
                        astype('uint8'), 'RGB')

                    # Zoom
                    if args.zoom_scale != 1:
                        pil_image_zoom = zoom_at(pil_image,
                            sideX/2, sideY/2, args.zoom_scale)
                    else:
                        pil_image_zoom = pil_image

                    # Shift - https://pillow.readthedocs.io/en
                        /latest/reference/ImageChops.html
                    if args.zoom_shift_x or args.zoom_shift_y:
                        # This one wraps the image
                        pil_image_zoom = ImageChops.offset(
                            pil_image_zoom, args.zoom_shift_x,
                             args.zoom_shift_y)

                    # Convert image back to a tensor again
                    pil_tensor = TF.to_tensor(pil_image_zoom)

                    # Re-encode
                    z, *_ = model.encode(pil_tensor.to(device)
                        .unsqueeze(0) * 2 - 1)
                    z_orig = z.clone()
                    z.requires_grad_(True)

                    # Re-create optimiser
                    opt = get_opt(args.optimiser, args.
                        step_size)

                # Next
                j += 1

        # Change text prompt
        if args.prompt_frequency > 0:
            if i % args.prompt_frequency == 0 and i > 0:
```

```
            # In case there aren't enough phrases, just
                loop
            if p >= len(all_phrases):
                p = 0

            pMs = []
            args.prompts = all_phrases[p]

            # Show user we're changing prompt
            print(args.prompts)

            for prompt in args.prompts:
                txt, weight, stop = split_prompt(prompt)
                embed = perceptor.encode_text(clip.
                    tokenize(txt).to(device)).float()
                pMs.append(Prompt(embed, weight, stop).to(
                    device))

            '''
            # Smooth test
            smoother = args.zoom_frequency * 15 #
                smoothing over x frames
            variable_lr = args.step_size * 0.25
            opt = get_opt(args.optimiser, variable_lr)
            '''

            p += 1

    '''
    if smoother > 0:
        if smoother == 1:
            opt = get_opt(args.optimiser, args.step_size)
        smoother -= 1
    '''

    '''
    # Messing with learning rate / optimisers
    if i % 225 == 0 and i > 0:
        variable_optimiser_item = random.choice(
            optimiser_list)
        variable_optimiser = variable_optimiser_item[0]
        variable_lr = variable_optimiser_item[1]

        opt = get_opt(variable_optimiser, variable_lr)
        print("New opt: %s, lr= %f" %(variable_optimiser,
            variable_lr))
    '''


    # Training time
    train(i)

    # Ready to stop yet?
    if i == args.max_iterations:
        if not args.video_style_dir:
            np.save("vqgan_loss.npy", np.array(loss_list))
            # plt.plot(np.array(loss_list))
            # plt.xlabel("Epoch Num")
            # plt.ylabel("Loss")
```

```
                    # plt.title("Loss Curve of VQGAN-CLIP")
                    # plt.savefig("Loss_Curve_VQGAN.png")
                    # we're done
                    break
                else:
                    if this_video_frame == (num_video_frames - 1):
                        # we're done
                        make_styled_video = True
                        break
                    else:
                        # Next video frame
                        this_video_frame += 1

                        # Reset the iteration count
                        i = -1
                        pbar.reset()

                        # Load the next frame, reset a few options
                            - same filename, different directory
                        args.init_image = video_frame_list[
                            this_video_frame]
                        print("Next frame: ", args.init_image)

                        if args.seed is None:
                            seed = torch.seed()
                        else:
                            seed = args.seed
                        torch.manual_seed(seed)
                        print("Seed: ", seed)

                        filename = os.path.basename(args.
                            init_image)
                        args.output = os.path.join(cwd, "steps",
                            filename)

                        # Load and resize image
                        img = Image.open(args.init_image)
                        pil_image = img.convert('RGB')
                        pil_image = pil_image.resize((sideX, sideY
                            ), Image.LANCZOS)
                        pil_tensor = TF.to_tensor(pil_image)

                        # Re-encode
                        z, *_ = model.encode(pil_tensor.to(device)
                            .unsqueeze(0) * 2 - 1)
                        z_orig = z.clone()
                        z.requires_grad_(True)

                        # Re-create optimiser
                        opt = get_opt(args.optimiser, args.
                            step_size)

            i += 1
            pbar.update()
except KeyboardInterrupt:
    pass

# All done :)
```

```
# Video generation
if args.make_video or args.make_zoom_video:
    init_frame = 1      # Initial video frame
    if args.make_zoom_video:
        last_frame = j
    else:
        last_frame = i  # This will raise an error if that number
            of frames does not exist.

    length = args.video_length # Desired time of the video in
        seconds

    min_fps = 10
    max_fps = 60

    total_frames = last_frame-init_frame

    frames = []
    tqdm.write('Generating video...')
    for i in range(init_frame,last_frame):
        temp = Image.open("./steps/"+ str(i) +'.png')
        keep = temp.copy()
        frames.append(keep)
        temp.close()

    if args.output_video_fps > 9:
        # Hardware encoding and video frame interpolation
        print("Creating interpolated frames...")
        ffmpeg_filter = f"minterpolate='mi_mode=mci:me=hexbs:
            me_mode=bidir:mc_mode=aobmc:vsbmc=1:mb_size=8:
            search_param=32:fps={args.output_video_fps}'"
        output_file = re.compile('\.png$').sub('.mp4', args.output
            )
        try:
            p = Popen(['ffmpeg',
                       '-y',
                       '-f', 'image2pipe',
                       '-vcodec', 'png',
                       '-r', str(args.input_video_fps),
                       '-i',
                       '-',
                       '-b:v', '10M',
                       '-vcodec', 'h264_nvenc',
                       '-pix_fmt', 'yuv420p',
                       '-strict', '-2',
                       '-filter:v', f'{ffmpeg_filter}',
                       '-metadata', f'comment={args.prompts}',
                   output_file], stdin=PIPE)
        except FileNotFoundError:
            print("ffmpeg command failed - check your installation
                ")
        for im in tqdm(frames):
            im.save(p.stdin, 'PNG')
        p.stdin.close()
        p.wait()
    else:
        # CPU
        fps = np.clip(total_frames/length,min_fps,max_fps)
```

```
    output_file = re.compile('\.png$').sub('.mp4', args.output
        )
    try:
        p = Popen(['ffmpeg',
                   '-y',
                   '-f', 'image2pipe',
                   '-vcodec', 'png',
                   '-r', str(fps),
                   '-i',
                   '-',
                   '-vcodec', 'libx264',
                   '-r', str(fps),
                   '-pix_fmt', 'yuv420p',
                   '-crf', '17',
                   '-preset', 'veryslow',
                   '-metadata', f'comment={args.prompts}',
                   output_file], stdin=PIPE)
    except FileNotFoundError:
        print("ffmpeg command failed - check your installation
            ")
    for im in tqdm(frames):
        im.save(p.stdin, 'PNG')
    p.stdin.close()
    p.wait()
```

B.6 CLIP Guided Diffusion

CLIP guided diffusion model:

```
#!/usr/bin/env python
# coding: utf-8

# In[11]:


# Imports

import math
import io
import sys

from IPython import display
from PIL import Image
import requests
import torch
from torch import nn
from torch.nn import functional as F
from torchvision import transforms
from torchvision.transforms import functional as TF
from tqdm.notebook import tqdm
import gc
sys.path.append('./CLIP')
sys.path.append('./guided-diffusion')
import clip
from guided_diffusion.script_util import
    create_model_and_diffusion, model_and_diffusion_defaults


# In[1]:
```

```
import lpips


# In[3]:


sys.path.append('./models')


# In[17]:


# Define necessary functions

def fetch(url_or_path):
    if str(url_or_path).startswith('http://') or str(url_or_path).
       startswith('https://'):
        r = requests.get(url_or_path)
        r.raise_for_status()
        fd = io.BytesIO()
        fd.write(r.content)
        fd.seek(0)
        return fd
    return open(url_or_path, 'rb')


def parse_prompt(prompt):
    if prompt.startswith('http://') or prompt.startswith('https
       ://'):
        vals = prompt.rsplit(':', 2)
        vals = [vals[0] + ':' + vals[1], *vals[2:]]
    else:
        vals = prompt.rsplit(':', 1)
    vals = vals + ['', '1'][len(vals):]
    return vals[0], float(vals[1])


class MakeCutouts(nn.Module):
    def __init__(self, cut_size, cutn, cut_pow=1.):
        super().__init__()
        self.cut_size = cut_size
        self.cutn = cutn
        self.cut_pow = cut_pow

    def forward(self, input):
        sideY, sideX = input.shape[2:4]
        max_size = min(sideX, sideY)
        min_size = min(sideX, sideY, self.cut_size)
        cutouts = []
        for _ in range(self.cutn):
            size = int(torch.rand([])**self.cut_pow * (max_size -
               min_size) + min_size)
            offsetx = torch.randint(0, sideX - size + 1, ())
            offsety = torch.randint(0, sideY - size + 1, ())
            cutout = input[:, :, offsety:offsety + size, offsetx:
               offsetx + size]
```

```
            cutouts.append(F.adaptive_avg_pool2d(cutout, self.
                cut_size))
        return torch.cat(cutouts)


def spherical_dist_loss(x, y):
    x = F.normalize(x, dim=-1)
    y = F.normalize(y, dim=-1)
    return (x - y).norm(dim=-1).div(2).arcsin().pow(2).mul(2)


def tv_loss(input):
    """L2 total variation loss, as in Mahendran et al."""
    input = F.pad(input, (0, 1, 0, 1), 'replicate')
    x_diff = input[..., :-1, 1:] - input[..., :-1, :-1]
    y_diff = input[..., 1:, :-1] - input[..., :-1, :-1]
    return (x_diff**2 + y_diff**2).mean([1, 2, 3])


def range_loss(input):
    return (input - input.clamp(-1, 1)).pow(2).mean([1, 2, 3])


# In[4]:


# Define necessary functions

def fetch(url_or_path):
    if str(url_or_path).startswith('http://') or str(url_or_path).
        startswith('https://'):
        r = requests.get(url_or_path)
        r.raise_for_status()
        fd = io.BytesIO()
        fd.write(r.content)
        fd.seek(0)
        return fd
    return open(url_or_path, 'rb')

class MakeCutouts(nn.Module):
    def __init__(self, cut_size, cutn, cut_pow=1.):
        super().__init__()
        self.cut_size = cut_size
        self.cutn = cutn
        self.cut_pow = cut_pow

    def forward(self, input):
        sideY, sideX = input.shape[2:4]
        max_size = min(sideX, sideY)
        min_size = min(sideX, sideY, self.cut_size)
        cutouts = []
        for _ in range(self.cutn):
            size = int(torch.rand([])**self.cut_pow * (max_size -
                min_size) + min_size)
            offsetx = torch.randint(0, sideX - size + 1, ())
            offsety = torch.randint(0, sideY - size + 1, ())
            cutout = input[:, :, offsety:offsety + size, offsetx:
                offsetx + size]
```

```
            cutouts.append(F.adaptive_avg_pool2d(cutout, self.
                cut_size))
        return torch.cat(cutouts)


def spherical_dist_loss(x, y):
    x = F.normalize(x, dim=-1)
    y = F.normalize(y, dim=-1)
    return (x - y).norm(dim=-1).div(2).arcsin().pow(2).mul(2)


def tv_loss(input):
    """L2 total variation loss, as in Mahendran et al."""
    input = F.pad(input, (0, 1, 0, 1), 'replicate')
    x_diff = input[..., :-1, 1:] - input[..., :-1, :-1]
    y_diff = input[..., 1:, :-1] - input[..., :-1, :-1]
    return (x_diff**2 + y_diff**2).mean([1, 2, 3])


# In[5]:


# Model settings

model_config = model_and_diffusion_defaults()
model_config.update({
    'attention_resolutions': '32, 16, 8',
    'class_cond': True,
    'diffusion_steps': 1000,
    'rescale_timesteps': True,
    'timestep_respacing': '1000',  # Modify this value to decrease
        the number of
                                   # timesteps.
    'image_size': 512,
    'learn_sigma': True,
    'noise_schedule': 'linear',
    'num_channels': 256,
    'num_head_channels': 64,
    'num_res_blocks': 2,
    'resblock_updown': True,
    'use_fp16': True,
    'use_scale_shift_norm': True,
})


# In[10]:


# Load models

device = torch.device('cuda:0' if torch.cuda.is_available() else '
    cpu')
print('Using device:', device)

model, diffusion = create_model_and_diffusion(**model_config)
model.load_state_dict(torch.load('./models/diff_512.pt',
    map_location='cpu'))
model.requires_grad_(False).eval().to(device)
for name, param in model.named_parameters():
```

```
    if 'qkv' in name or 'norm' in name or 'proj' in name:
        param.requires_grad_()
if model_config['use_fp16']:
    model.convert_to_fp16()

clip_model = clip.load('ViT-B/16', jit=False)[0].eval().
    requires_grad_(False).to(device)
check = torch.load('./model_fastai_b16.pth', map_location='cpu')
clip_model.load_state_dict(check['model'])
clip_size = clip_model.visual.input_resolution
normalize = transforms.Normalize(mean=[0.48145466, 0.4578275,
    0.40821073],
                                 std=[0.26862954, 0.26130258,
                                     0.27577711])


# In[6]:


# Model settings

model_config = model_and_diffusion_defaults()
model_config.update({
    'attention_resolutions': '32, 16, 8',
    'class_cond': False,
    'diffusion_steps': 1000,
    'rescale_timesteps': True,
    'timestep_respacing': '1000',  # Modify this value to decrease
        the number of
                                   # timesteps.
    'image_size': 256,
    'learn_sigma': True,
    'noise_schedule': 'linear',
    'num_channels': 256,
    'num_head_channels': 64,
    'num_res_blocks': 2,
    'resblock_updown': True,
    'use_checkpoint': False,
    'use_fp16': True,
    'use_scale_shift_norm': True,
})


# In[7]:


# Load models

device = torch.device('cuda:0' if torch.cuda.is_available() else '
    cpu')
print('Using device:', device)

model, diffusion = create_model_and_diffusion(**model_config)
model.load_state_dict(torch.load('./models/diff_256.pt',
    map_location='cpu'))
model.requires_grad_(False).eval().to(device)
if model_config['use_fp16']:
    model.convert_to_fp16()
```

```
clip_model = clip.load('ViT-B/16', jit=False)[0].eval().
    requires_grad_(False).to(device)
check = torch.load('./model_fastai_b16.pth', map_location='cpu')
clip_model.load_state_dict(check['model'])
clip_size = clip_model.visual.input_resolution
normalize = transforms.Normalize(mean=[0.48145466, 0.4578275,
    0.40821073],
                                 std=[0.26862954, 0.26130258,
                                    0.27577711])
lpips_model = lpips.LPIPS(net='vgg').to(device)


# In[13]:


prompts = 'A dog in front of sea'
batch_size = 1
clip_guidance_scale = 1000  # Controls how much the image should
    look like the prompt.
tv_scale = 150              # Controls the smoothness of the final
     output.
cutn = 40
cut_pow = 0.5
n_batches = 1
init_image = None   # This can be an URL or local path and must be
     in quotes.
skip_timesteps = 0  # This needs to be between approx. 200 and 500
     when using an init image.
                    # Higher values make the output look more like
                         the init.
seed = 0


# In[19]:


prompts = ['A giraffe among zebras']
image_prompts = []
batch_size = 1
clip_guidance_scale = 1000  # Controls how much the image should
    look like the prompt.
tv_scale = 150              # Controls the smoothness of the final
     output.
range_scale = 50            # Controls how far out of range RGB
    values are allowed to be.
cutn = 16
n_batches = 1
init_image = None   # This can be an URL or local path and must be
     in quotes.
skip_timesteps = 0  # This needs to be between approx. 200 and 500
     when using an init image.
                    # Higher values make the output look more like
                         the init.
init_scale = 0      # This enhances the effect of the init image,
    a good value is 1000.
seed = 0


# In[20]:
```

```
def do_run():
    if seed is not None:
        torch.manual_seed(seed)

    make_cutouts = MakeCutouts(clip_size, cutn)
    side_x = side_y = model_config['image_size']

    target_embeds, weights = [], []

    for prompt in prompts:
        txt, weight = parse_prompt(prompt)
        target_embeds.append(clip_model.encode_text(clip.tokenize(
            txt).to(device)).float())
        weights.append(weight)

    for prompt in image_prompts:
        path, weight = parse_prompt(prompt)
        img = Image.open(fetch(path)).convert('RGB')
        img = TF.resize(img, min(side_x, side_y, *img.size),
            transforms.InterpolationMode.LANCZOS)
        batch = make_cutouts(TF.to_tensor(img).unsqueeze(0).to(
            device))
        embed = clip_model.encode_image(normalize(batch)).float()
        target_embeds.append(embed)
        weights.extend([weight / cutn] * cutn)

    target_embeds = torch.cat(target_embeds)
    weights = torch.tensor(weights, device=device)
    if weights.sum().abs() < 1e-3:
        raise RuntimeError('The weights must not sum to 0.')
    weights /= weights.sum().abs()

    init = None
    if init_image is not None:
        init = Image.open(fetch(init_image)).convert('RGB')
        init = init.resize((side_x, side_y), Image.LANCZOS)
        init = TF.to_tensor(init).to(device).unsqueeze(0).mul(2).
            sub(1)

    cur_t = None

    def cond_fn(x, t, out, y=None):
        n = x.shape[0]
        fac = diffusion.sqrt_one_minus_alphas_cumprod[cur_t]
        x_in = out['pred_xstart'] * fac + x * (1 - fac)
        clip_in = normalize(make_cutouts(x_in.add(1).div(2)))
        image_embeds = clip_model.encode_image(clip_in).float()
        dists = spherical_dist_loss(image_embeds.unsqueeze(1),
            target_embeds.unsqueeze(0))
        dists = dists.view([cutn, n, -1])
        losses = dists.mul(weights).sum(2).mean(0)
        tv_losses = tv_loss(x_in)
        range_losses = range_loss(out['pred_xstart'])
        loss = losses.sum() * clip_guidance_scale + tv_losses.sum
            () * tv_scale + range_losses.sum() * range_scale
        if init is not None and init_scale:
            init_losses = lpips_model(x_in, init)
```

```
            loss = loss + init_losses.sum() * init_scale
        return -torch.autograd.grad(loss, x)[0]

    if model_config['timestep_respacing'].startswith('ddim'):
        sample_fn = diffusion.ddim_sample_loop_progressive
    else:
        sample_fn = diffusion.p_sample_loop_progressive

    for i in range(n_batches):
        cur_t = diffusion.num_timesteps - skip_timesteps - 1

        samples = sample_fn(
            model,
            (batch_size, 3, side_y, side_x),
            clip_denoised=False,
            model_kwargs={},
            cond_fn=cond_fn,
            progress=True,
            skip_timesteps=skip_timesteps,
            init_image=init,
            randomize_class=True,
            cond_fn_with_grad=True,
        )

        for j, sample in enumerate(samples):
            cur_t -= 1
            if j % 100 == 0 or cur_t == -1:
                print()
                for k, image in enumerate(sample['pred_xstart']):
                    filename = f'progress_{i * batch_size + k:05}.
                        png'
                    TF.to_pil_image(image.add(1).div(2).clamp(0,
                        1)).save(filename)
                    tqdm.write(f'Batch {i}, step {j}, output {k
                        }:')
                    display.display(display.Image(filename))

gc.collect()
do_run()


# In[ ]:




# In[9]:


def do_run():
    if seed is not None:
        torch.manual_seed(seed)

    text_embed = clip_model.encode_text(clip.tokenize(prompt).to(
        device)).float()

    init = None
    if init_image is not None:
```

```
    init = Image.open(fetch(init_image)).convert('RGB')
    init = init.resize((model_config['image_size'],
        model_config['image_size']), Image.LANCZOS)
    init = TF.to_tensor(init).to(device).unsqueeze(0).mul(2).
        sub(1)

make_cutouts = MakeCutouts(clip_size, cutn, cut_pow)

cur_t = None

def cond_fn(x, t, y=None):
    with torch.enable_grad():
        x = x.detach().requires_grad_()
        n = x.shape[0]
        my_t = torch.ones([n], device=device, dtype=torch.long
            ) * cur_t
        out = diffusion.p_mean_variance(model, x, my_t,
            clip_denoised=False, model_kwargs={'y': y})
        fac = diffusion.sqrt_one_minus_alphas_cumprod[cur_t]
        x_in = out['pred_xstart'] * fac + x * (1 - fac)
        clip_in = normalize(make_cutouts(x_in.add(1).div(2)))
        image_embeds = clip_model.encode_image(clip_in).float
            ().view([cutn, n, -1])
        dists = spherical_dist_loss(image_embeds, text_embed.
            unsqueeze(0))
        losses = dists.mean(0)
        tv_losses = tv_loss(x_in)
        loss = losses.sum() * clip_guidance_scale + tv_losses.
            sum() * tv_scale
        return -torch.autograd.grad(loss, x)[0]

if model_config['timestep_respacing'].startswith('ddim'):
    sample_fn = diffusion.ddim_sample_loop_progressive
else:
    sample_fn = diffusion.p_sample_loop_progressive

for i in range(n_batches):
    cur_t = diffusion.num_timesteps - skip_timesteps - 1

    samples = sample_fn(
        model,
        (batch_size, 3, model_config['image_size'],
            model_config['image_size']),
        clip_denoised=False,
        model_kwargs={'y': torch.zeros([batch_size], device=
            device, dtype=torch.long)},
        cond_fn=cond_fn,
        progress=True,
        skip_timesteps=skip_timesteps,
        init_image=init,
        randomize_class=True,
    )

    for j, sample in enumerate(samples):
        cur_t -= 1
        if j % 100 == 0 or cur_t == -1:
            print()
            for k, image in enumerate(sample['pred_xstart']):
```

```
                    filename = f'progress_{i * batch_size + k:05}.
                        png'
                    TF.to_pil_image(image.add(1).div(2).clamp(0,
                        1)).save(filename)
                    tqdm.write(f'Batch {i}, step {j}, output {k
                        }:')
                    display.display(display.Image(filename))

do_run()
```

Quick CLIP guided diffusion model:

```
#!/usr/bin/env python
# coding: utf-8

# In[11]:


# Imports

import math
import io
import sys

from IPython import display
from PIL import Image
import requests
import torch
from torch import nn
from torch.nn import functional as F
from torchvision import transforms
from torchvision.transforms import functional as TF
from tqdm.notebook import tqdm
import gc
sys.path.append('./CLIP')
sys.path.append('./guided-diffusion')
import clip
from guided_diffusion.script_util import
    create_model_and_diffusion, model_and_diffusion_defaults


# In[1]:


import lpips


# In[3]:


sys.path.append('./models')


# In[17]:


# Define necessary functions

def fetch(url_or_path):
```

```
    if str(url_or_path).startswith('http://') or str(url_or_path).
        startswith('https://'):
        r = requests.get(url_or_path)
        r.raise_for_status()
        fd = io.BytesIO()
        fd.write(r.content)
        fd.seek(0)
        return fd
    return open(url_or_path, 'rb')


def parse_prompt(prompt):
    if prompt.startswith('http://') or prompt.startswith('https
        ://'):
        vals = prompt.rsplit(':', 2)
        vals = [vals[0] + ':' + vals[1], *vals[2:]]
    else:
        vals = prompt.rsplit(':', 1)
    vals = vals + ['', '1'][len(vals):]
    return vals[0], float(vals[1])


class MakeCutouts(nn.Module):
    def __init__(self, cut_size, cutn, cut_pow=1.):
        super().__init__()
        self.cut_size = cut_size
        self.cutn = cutn
        self.cut_pow = cut_pow

    def forward(self, input):
        sideY, sideX = input.shape[2:4]
        max_size = min(sideX, sideY)
        min_size = min(sideX, sideY, self.cut_size)
        cutouts = []
        for _ in range(self.cutn):
            size = int(torch.rand([])**self.cut_pow * (max_size -
                min_size) + min_size)
            offsetx = torch.randint(0, sideX - size + 1, ())
            offsety = torch.randint(0, sideY - size + 1, ())
            cutout = input[:, :, offsety:offsety + size, offsetx:
                offsetx + size]
            cutouts.append(F.adaptive_avg_pool2d(cutout, self.
                cut_size))
        return torch.cat(cutouts)


def spherical_dist_loss(x, y):
    x = F.normalize(x, dim=-1)
    y = F.normalize(y, dim=-1)
    return (x - y).norm(dim=-1).div(2).arcsin().pow(2).mul(2)


def tv_loss(input):
    """L2 total variation loss, as in Mahendran et al."""
    input = F.pad(input, (0, 1, 0, 1), 'replicate')
    x_diff = input[..., :-1, 1:] - input[..., :-1, :-1]
    y_diff = input[..., 1:, :-1] - input[..., :-1, :-1]
    return (x_diff**2 + y_diff**2).mean([1, 2, 3])
```

```
def range_loss(input):
    return (input - input.clamp(-1, 1)).pow(2).mean([1, 2, 3])


# In[4]:


# Define necessary functions

def fetch(url_or_path):
    if str(url_or_path).startswith('http://') or str(url_or_path).
       startswith('https://'):
        r = requests.get(url_or_path)
        r.raise_for_status()
        fd = io.BytesIO()
        fd.write(r.content)
        fd.seek(0)
        return fd
    return open(url_or_path, 'rb')

class MakeCutouts(nn.Module):
    def __init__(self, cut_size, cutn, cut_pow=1.):
        super().__init__()
        self.cut_size = cut_size
        self.cutn = cutn
        self.cut_pow = cut_pow

    def forward(self, input):
        sideY, sideX = input.shape[2:4]
        max_size = min(sideX, sideY)
        min_size = min(sideX, sideY, self.cut_size)
        cutouts = []
        for _ in range(self.cutn):
            size = int(torch.rand([])**self.cut_pow * (max_size -
                min_size) + min_size)
            offsetx = torch.randint(0, sideX - size + 1, ())
            offsety = torch.randint(0, sideY - size + 1, ())
            cutout = input[:, :, offsety:offsety + size, offsetx:
                offsetx + size]
            cutouts.append(F.adaptive_avg_pool2d(cutout, self.
                cut_size))
        return torch.cat(cutouts)


def spherical_dist_loss(x, y):
    x = F.normalize(x, dim=-1)
    y = F.normalize(y, dim=-1)
    return (x - y).norm(dim=-1).div(2).arcsin().pow(2).mul(2)


def tv_loss(input):
    """L2 total variation loss, as in Mahendran et al."""
    input = F.pad(input, (0, 1, 0, 1), 'replicate')
    x_diff = input[..., :-1, 1:] - input[..., :-1, :-1]
    y_diff = input[..., 1:, :-1] - input[..., :-1, :-1]
    return (x_diff**2 + y_diff**2).mean([1, 2, 3])
```

```
# In[5]:


# Model settings

model_config = model_and_diffusion_defaults()
model_config.update({
    'attention_resolutions': '32, 16, 8',
    'class_cond': True,
    'diffusion_steps': 1000,
    'rescale_timesteps': True,
    'timestep_respacing': '1000',  # Modify this value to decrease
        the number of
                                   # timesteps.
    'image_size': 512,
    'learn_sigma': True,
    'noise_schedule': 'linear',
    'num_channels': 256,
    'num_head_channels': 64,
    'num_res_blocks': 2,
    'resblock_updown': True,
    'use_fp16': True,
    'use_scale_shift_norm': True,
})


# In[10]:


# Load models

device = torch.device('cuda:0' if torch.cuda.is_available() else '
    cpu')
print('Using device:', device)

model, diffusion = create_model_and_diffusion(**model_config)
model.load_state_dict(torch.load('./models/diff_256.pt',
    map_location='cpu'))
model.requires_grad_(False).eval().to(device)
for name, param in model.named_parameters():
    if 'qkv' in name or 'norm' in name or 'proj' in name:
        param.requires_grad_()
if model_config['use_fp16']:
    model.convert_to_fp16()

clip_model = clip.load('ViT-B/16', jit=False)[0].eval().
    requires_grad_(False).to(device)
check = torch.load('./model_fastai_b16.pth', map_location='cpu')
clip_model.load_state_dict(check['model'])
clip_size = clip_model.visual.input_resolution
normalize = transforms.Normalize(mean=[0.48145466, 0.4578275,
    0.40821073],
                                 std=[0.26862954, 0.26130258,
                                     0.27577711])


# In[6]:
```

```
# Model settings

model_config = model_and_diffusion_defaults()
model_config.update({
    'attention_resolutions': '32, 16, 8',
    'class_cond': False,
    'diffusion_steps': 1000,
    'rescale_timesteps': True,
    'timestep_respacing': '1000',  # Modify this value to decrease
        the number of
                                   # timesteps.
    'image_size': 256,
    'learn_sigma': True,
    'noise_schedule': 'linear',
    'num_channels': 256,
    'num_head_channels': 64,
    'num_res_blocks': 2,
    'resblock_updown': True,
    'use_checkpoint': False,
    'use_fp16': True,
    'use_scale_shift_norm': True,
})


# In[7]:


# Load models

device = torch.device('cuda:0' if torch.cuda.is_available() else '
    cpu')
print('Using device:', device)

model, diffusion = create_model_and_diffusion(**model_config)
model.load_state_dict(torch.load('./models/diff_256.pt',
    map_location='cpu'))
model.requires_grad_(False).eval().to(device)
if model_config['use_fp16']:
    model.convert_to_fp16()

clip_model = clip.load('ViT-B/16', jit=False)[0].eval().
    requires_grad_(False).to(device)
check = torch.load('./model_fastai_b16.pth', map_location='cpu')
clip_model.load_state_dict(check['model'])
clip_size = clip_model.visual.input_resolution
normalize = transforms.Normalize(mean=[0.48145466, 0.4578275,
    0.40821073],
                                 std=[0.26862954, 0.26130258,
                                    0.27577711])
lpips_model = lpips.LPIPS(net='vgg').to(device)


# In[13]:


prompts = 'A dog in front of sea'
batch_size = 1
clip_guidance_scale = 1000  # Controls how much the image should
    look like the prompt.
```

```
tv_scale = 150              # Controls the smoothness of the final
    output.
cutn = 40
cut_pow = 0.5
n_batches = 1
init_image = None   # This can be an URL or local path and must be
    in quotes.
skip_timesteps = 0  # This needs to be between approx. 200 and 500
    when using an init image.
                    # Higher values make the output look more like
                        the init.
seed = 0


# In[19]:


prompts = ['A giraffe among zebras']
image_prompts = []
batch_size = 1
clip_guidance_scale = 1000  # Controls how much the image should
   look like the prompt.
tv_scale = 150              # Controls the smoothness of the final
    output.
range_scale = 50            # Controls how far out of range RGB
   values are allowed to be.
cutn = 16
n_batches = 1
init_image = None   # This can be an URL or local path and must be
    in quotes.
skip_timesteps = 0  # This needs to be between approx. 200 and 500
    when using an init image.
                    # Higher values make the output look more like
                        the init.
init_scale = 0      # This enhances the effect of the init image,
   a good value is 1000.
seed = 0


# In[20]:


def do_run():
    if seed is not None:
        torch.manual_seed(seed)

    make_cutouts = MakeCutouts(clip_size, cutn)
    side_x = side_y = model_config['image_size']

    target_embeds, weights = [], []

    for prompt in prompts:
        txt, weight = parse_prompt(prompt)
        target_embeds.append(clip_model.encode_text(clip.tokenize(
            txt).to(device)).float())
        weights.append(weight)

    for prompt in image_prompts:
        path, weight = parse_prompt(prompt)
```

```
    img = Image.open(fetch(path)).convert('RGB')
    img = TF.resize(img, min(side_x, side_y, *img.size),
        transforms.InterpolationMode.LANCZOS)
    batch = make_cutouts(TF.to_tensor(img).unsqueeze(0).to(
        device))
    embed = clip_model.encode_image(normalize(batch)).float()
    target_embeds.append(embed)
    weights.extend([weight / cutn] * cutn)

target_embeds = torch.cat(target_embeds)
weights = torch.tensor(weights, device=device)
if weights.sum().abs() < 1e-3:
    raise RuntimeError('The weights must not sum to 0.')
weights /= weights.sum().abs()

init = None
if init_image is not None:
    init = Image.open(fetch(init_image)).convert('RGB')
    init = init.resize((side_x, side_y), Image.LANCZOS)
    init = TF.to_tensor(init).to(device).unsqueeze(0).mul(2).
        sub(1)

cur_t = None

def cond_fn(x, t, out, y=None):
    n = x.shape[0]
    fac = diffusion.sqrt_one_minus_alphas_cumprod[cur_t]
    x_in = out['pred_xstart'] * fac + x * (1 - fac)
    clip_in = normalize(make_cutouts(x_in.add(1).div(2)))
    image_embeds = clip_model.encode_image(clip_in).float()
    dists = spherical_dist_loss(image_embeds.unsqueeze(1),
        target_embeds.unsqueeze(0))
    dists = dists.view([cutn, n, -1])
    losses = dists.mul(weights).sum(2).mean(0)
    tv_losses = tv_loss(x_in)
    range_losses = range_loss(out['pred_xstart'])
    loss = losses.sum() * clip_guidance_scale + tv_losses.sum
        () * tv_scale + range_losses.sum() * range_scale
    if init is not None and init_scale:
        init_losses = lpips_model(x_in, init)
        loss = loss + init_losses.sum() * init_scale
    return -torch.autograd.grad(loss, x)[0]

if model_config['timestep_respacing'].startswith('ddim'):
    sample_fn = diffusion.ddim_sample_loop_progressive
else:
    sample_fn = diffusion.p_sample_loop_progressive

for i in range(n_batches):
    cur_t = diffusion.num_timesteps - skip_timesteps - 1

    samples = sample_fn(
        model,
        (batch_size, 3, side_y, side_x),
        clip_denoised=False,
        model_kwargs={},
        cond_fn=cond_fn,
        progress=True,
        skip_timesteps=skip_timesteps,
```

```
            init_image=init,
            randomize_class=True,
            cond_fn_with_grad=True,
        )

        for j, sample in enumerate(samples):
            cur_t -= 1
            if j % 100 == 0 or cur_t == -1:
                print()
                for k, image in enumerate(sample['pred_xstart']):
                    filename = f'progress_{i * batch_size + k:05}.
                        png'
                    TF.to_pil_image(image.add(1).div(2).clamp(0,
                        1)).save(filename)
                    tqdm.write(f'Batch {i}, step {j}, output {k
                        }:')
                    display.display(display.Image(filename))

gc.collect()
do_run()


# In[ ]:




# In[9]:


def do_run():
    if seed is not None:
        torch.manual_seed(seed)

    text_embed = clip_model.encode_text(clip.tokenize(prompt).to(
        device)).float()

    init = None
    if init_image is not None:
        init = Image.open(fetch(init_image)).convert('RGB')
        init = init.resize((model_config['image_size'],
            model_config['image_size']), Image.LANCZOS)
        init = TF.to_tensor(init).to(device).unsqueeze(0).mul(2).
            sub(1)

    make_cutouts = MakeCutouts(clip_size, cutn, cut_pow)

    cur_t = None

    def cond_fn(x, t, y=None):
        with torch.enable_grad():
            x = x.detach().requires_grad_()
            n = x.shape[0]
            my_t = torch.ones([n], device=device, dtype=torch.long
                ) * cur_t
            out = diffusion.p_mean_variance(model, x, my_t,
                clip_denoised=False, model_kwargs={'y': y})
            fac = diffusion.sqrt_one_minus_alphas_cumprod[cur_t]
```

```
            x_in = out['pred_xstart'] * fac + x * (1 - fac)
            clip_in = normalize(make_cutouts(x_in.add(1).div(2)))
            image_embeds = clip_model.encode_image(clip_in).float
                ().view([cutn, n, -1])
            dists = spherical_dist_loss(image_embeds, text_embed.
                unsqueeze(0))
            losses = dists.mean(0)
            tv_losses = tv_loss(x_in)
            loss = losses.sum() * clip_guidance_scale + tv_losses.
                sum() * tv_scale
            return -torch.autograd.grad(loss, x)[0]

    if model_config['timestep_respacing'].startswith('ddim'):
        sample_fn = diffusion.ddim_sample_loop_progressive
    else:
        sample_fn = diffusion.p_sample_loop_progressive

    for i in range(n_batches):
        cur_t = diffusion.num_timesteps - skip_timesteps - 1

        samples = sample_fn(
            model,
            (batch_size, 3, model_config['image_size'],
                model_config['image_size']),
            clip_denoised=False,
            model_kwargs={'y': torch.zeros([batch_size], device=
                device, dtype=torch.long)},
            cond_fn=cond_fn,
            progress=True,
            skip_timesteps=skip_timesteps,
            init_image=init,
            randomize_class=True,
        )

        for j, sample in enumerate(samples):
            cur_t -= 1
            if j % 100 == 0 or cur_t == -1:
                print()
                for k, image in enumerate(sample['pred_xstart']):
                    filename = f'progress_{i * batch_size + k:05}.
                        png'
                    TF.to_pil_image(image.add(1).div(2).clamp(0,
                        1)).save(filename)
                    tqdm.write(f'Batch {i}, step {j}, output {k
                        }:')
                    display.display(display.Image(filename))

do_run()
```

B.7 Imagen

Imagen model:

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:
```

```
from PIL import Image
from IPython.display import display
import torch as th
from imagen_pytorch.model_creation import
    create_model_and_diffusion as
    create_model_and_diffusion_dalle2
from imagen_pytorch.model_creation import
    model_and_diffusion_defaults as
    model_and_diffusion_defaults_dalle2
from transformers import AutoTokenizer
import cv2

import glob
import os
from basicsr.archs.rrdbnet_arch import RRDBNet
from realesrgan import RealESRGANer
from realesrgan.archs.srvgg_arch import SRVGGNetCompact
from gfpgan import GFPGANer
#import gfpgan

has_cuda = th.cuda.is_available()
device = th.device('cpu' if not has_cuda else 'cuda')


# In[2]:


device = th.device('cpu')


# In[2]:


has_cuda


# ## Setting Up

# In[3]:


#Device config to adjust code for whether using CUDA or not
device_config = {
    "map_location": "cpu" if not has_cuda else None,
    "use_half": True if has_cuda else False,
    "device_name": "cuda" if has_cuda else "cpu"
}


# In[4]:


def model_fn(x_t, ts, **kwargs):
    guidance_scale = 5
    half = x_t[: len(x_t) // 2]
    combined = th.cat([half, half], dim=0)
    model_out = model(combined, ts, **kwargs)
    eps, rest = model_out[:, :3], model_out[:, 3:]
    cond_eps, uncond_eps = th.split(eps, len(eps) // 2, dim=0)
```

```
    half_eps = uncond_eps + guidance_scale * (cond_eps -
        uncond_eps)
    eps = th.cat([half_eps, half_eps], dim=0)
    return th.cat([eps, rest], dim=1)


# In[5]:


def show_images(batch: th.Tensor):
    """ Display a batch of images inline."""
    scaled = ((batch + 1)*127.5).round().clamp(0,255).to(th.uint8)
        .cpu()
    reshaped = scaled.permute(2, 0, 3, 1).reshape([batch.shape[2],
         -1, 3])
    display(Image.fromarray(reshaped.numpy()))


# In[6]:


def get_numpy_img(img):
    scaled = ((img + 1)*127.5).round().clamp(0,255).to(th.uint8).
        cpu()
    reshaped = scaled.permute(2, 0, 3, 1).reshape([img.shape[2],
        -1, 3])
    return cv2.cvtColor(reshaped.numpy(), cv2.COLOR_BGR2RGB)


# In[7]:


def _fix_path(path):
    d = th.load(path, map_location=device_config["map_location"])
    checkpoint = {}
    for key in d.keys():
        checkpoint[key.replace('module.','')] = d[key]
    return checkpoint


# In[ ]:




# In[8]:


options = model_and_diffusion_defaults_dalle2()
options['use_fp16'] = False
options['diffusion_steps'] = 200
options['num_res_blocks'] = 3
options['t5_name'] = 't5-3b'
options['cache_text_emb'] = True
model, diffusion = create_model_and_diffusion_dalle2(**options)

model.eval()
```

```
#if has_cuda:
#     model.convert_to_fp16()

model.to(device)


# In[12]:


model.load_state_dict(_fix_path(r"C:\Users\onuru\EEE443\
    Final_project\model.pt"))
print('total base parameters', sum(x.numel() for x in model.
    parameters()))


# In[ ]:


num_params = sum(param.numel() for param in model.parameters())
num_params


# In[ ]:


realesrgan_model = RRDBNet(num_in_ch=3, num_out_ch=3, num_feat=64,
                           num_block=23, num_grow_ch=32, scale=4)


# In[ ]:


netscale = 4


# In[ ]:


upsampler = RealESRGANer(
    scale=netscale,
    model_path=r"C:\Users\onuru\EEE443\Final_project\Real-
        ESRGANRealESRGAN_x4plus.pth",
    model=realesrgan_model,
    tile=0,
    tile_pad=10,
    pre_pad=0,
    half=device_config["use_half"]
)


# In[ ]:


face_enhancer = GFPGANer(
    model_path='https://github.com/TencentARC/GFPGAN/releases/
        download/v1.3.0/GFPGANv1.3.pth',
    upscale=4,
    arch='clean',
    channel_multiplier=2,
```

```
    bg_upsampler=upsampler
)


# In[ ]:


tokenizer = AutoTokenizer.from_pretrained(options['t5_name'])


# In[ ]:


prompt = 'a pear on a chair'


# In[ ]:


text_encoding = tokenizer(
    prompt,
    max_length=128,
    padding="max_length",
    truncation=True,
    return_attention_mask=True,
    add_special_tokens=True,
    return_tensors="pt"
)

uncond_text_encoding = tokenizer(
    '',
    max_length=128,
    padding="max_length",
    truncation=True,
    return_attention_mask=True,
    add_special_tokens=True,
    return_tensors="pt"
)


# In[ ]:


import numpy as np
batch_size = 1
cond_tokens = th.from_numpy(np.array([text_encoding['input_ids
    '][0].numpy() for i in range(batch_size)]))
uncond_tokens = th.from_numpy(np.array([uncond_text_encoding['
    input_ids'][0].numpy() for i in range(batch_size)]))
cond_attention_mask = th.from_numpy(np.array([text_encoding['
    attention_mask'][0].numpy() for i in range(batch_size)]))
uncond_attention_mask = th.from_numpy(np.array([
    uncond_text_encoding['attention_mask'][0].numpy() for i in
    range(batch_size)]))
model_kwargs = {}
model_kwargs["tokens"] = th.cat((cond_tokens,
                                 uncond_tokens)).to(device)
model_kwargs["mask"] = th.cat((cond_attention_mask,
                               uncond_attention_mask)).to(device)
```

```
# ## Generation

# In[ ]:


model.del_cache()

sample = diffusion.p_sample_loop(
    model_fn,
    (batch_size * 2, 3, 64, 64),
    clip_denoised=True,
    model_kwargs=model_kwargs,
    device=device_config["device_name"],
    progress=True,
)[:batch_size]

model.del_cache()


# In[ ]:


show_images(sample)


# In[ ]:


for i in sample:
    show_images(i.unsqueeze(0))


# In[ ]:


new_img = get_numpy_img(sample)


# In[ ]:


#%%time
for j in range(batch_size):
    new_img = get_numpy_img(sample[j].unsqueeze(0))

    if not has_cuda:
        new_img = new_img.astype(np.float32)
    for i in range(1):
        _, _, new_img = face_enhancer.enhance(new_img, has_aligned
            =False,
                                              only_center_face=
                                                  False,
                                                  paste_back=True)
        cv2.imwrite(r'./generations/test_out{j}.jpg', new_img)
```