**Text-to-Image Generator**

**Using PyTorch**

**Text-to-Encoder Part**

* **At first importing important Libraries**
* **Creating a MultiHeadAttention Class :- The Multiheadattention class encapsulates the mechanism where the model can focus on different parts of the input sequence (Q, K, V) simultaneously, leveraging multiple attention heads (num\_heads). This approach enhances the model's ability to capture complex dependencies and patterns in data, especially useful in tasks requiring understanding of sequential data like natural language processing.**
* **Creating a positionwWiseFeedForward Class :**- **In Transformer models, PositionWiseFeedForward is typically applied independently to each position in the sequence, enhancing the model's ability to learn intricate relationships between tokens or words. This layer contributes to the overall expressive power of Transformer architectures in processing sequential data.**
* **Creating a function for Positional Encoding :**- **The purpose of PositionalEncoding is to inject information about the relative or absolute position of the tokens in the sequence into the embeddings. This is crucial because Transformer models don't have a built-in understanding of sequence order like recurrent neural networks (RNNs) do. By adding these positional encodings to the input embeddings, the Transformer model can differentiate between tokens based on their positions in the sequence**.

**Creating an instance of Encoder Layer :- The EncoderLayer class encapsulates a single layer of a Transformer encoder, which sequentially applies multi-head self-attention and position-wise feed-forward networks. It utilizes residual connections, layer normalization, and dropout to improve learning stability and generalization. This architecture is crucial for processing and encoding information in a Transformer-based model, commonly used in natural language processing tasks like machine translation and text generation.**

**Creating an instance of Transformer The Transformer class is a fundamental implementation of a Transformer model for sequence-to-sequence tasks in NLP. It leverages embedding layers, positional encodings, multiple layers of self-attention and feed-forward networks, and dropout regularization to encode input sequences effectively. This structure facilitates learning contextual relationships between tokens and is widely used in tasks such as machine translation, text generation, and language understanding.**

Creating an Encoder output function by putting it all together to give an dense vector representation of the input\_text provided

StyleGan part

Creating an Instance of ImageLabelDataset :- class that I have built is designed to create a dataset for paired images and their corresponding labels, where the labels are stored as text in separate files and need to be encoded into dense vector representations. It provides a structured way to load paired image and label data, facilitating training or evaluation of models that require both types of data for learning. It handles file reading, image loading, label encoding, and transformation application, making it versatile for various deep learning applications.

Then Defining get\_loader function to handle the Dataset or to shuffle them

Then created a function called check\_loader function where I have checked to make sure that get\_loader function works properly.

**Defined a variable named Factor which gets multiplied to the IN\_CHANNELS to have the number of channels that we want in each resolution**

**Noise Mapping Network**

**The noise mapping network takes Z and puts it through eight fully connected layers separated by some activation.**

**In the init part we send in\_features and out\_channels. Create a linear layer, then we define a scale that will be equal to the square root of 2 divided by in\_features, we copy the bias of the current column layer into a variable because we don't want the bias of the linear layer to be scaled, then we remove it, Finally, we initialize linear layer.**

**In the forward part, we send x and all that we are going to do is multiplicate x with scale and add the bias after reshaping it.**

**then create the MappingNetwork class.**

**In the init part we send z\_dim and w\_din, and we define the network mapping that first normalizes z\_dim, followed by eight of WSLInear and ReLU as activation functions.**

**In the forward part, we return the network mapping**

**Adaptive Instance Normalization (AdaIN)**

**Then created AdaIN class**

**In the init part we send channels, w\_dim, and we initialize instance\_norm which will be the instance normalization part, and we initialize style\_scale and style\_bias which will be the adaptive parts with WSLinear that maps the Noise Mapping Network W into channels.**

**In the forward pass, we send x, apply instance normalization for it, and return style\_sclate \* x + style\_bias.**

**Inject Noise**

**then create the class InjectNoise to inject the noise into the generator**

**In the init part we sent channels and we initialize weight from a random normal distribution and we use nn.Parameter so that these weights can be optimized**

**In the forward part, we send an image x and we return it with random noise added.**

**Then the class WSConv2d (weighted scaled convolutional layer) is created to Equalized Learning Rate for the conv layers.**

**the class PixelNorm is made to normalize Z before the Noise Mapping Network.**

**Create ConvBlock Class :- the** defines a convolutional block module using Weight Standardization (WSConv2d) and Leaky ReLU activation. This kind of block is commonly used in various neural network architectures, including in GANs (Generative Adversarial Networks) for both the generator and discriminator networks

And now finally we will make an Instance of Discriminator :- The Discriminator class is crucial in the progressive growing GAN framework for distinguishing real images from generated ones across different resolutions. It employs progressive blocks of convolutional layers (ConvBlock) and RGB layers (WSConv2d) to handle varying input sizes. The fade\_in method facilitates smooth transitions between different resolution levels, while minibatch\_std enhances the discriminator's ability to learn batch-level statistics. By incorporating these components, the Discriminator effectively operates in tandem with the Generator to produce high-quality synthetic images. Adjust the in\_channels, img\_channels, and factors parameters to suit your specific model requirements and dataset characteristics.

Then the moment for building Generator :-

**Generator**

**In the generator architecture, we have some patterns that repeat so first create a class to make our code as clean as possible, By naming the class GenBlock which will be inherited from nn.Module.**

**In the init part we send in\_channels, out\_channels, and w\_dim, then we initialize conv1 by WSConv2d which maps in\_channels to out\_channels, conv2 by WSConv2d which maps out\_channels to out\_channels, leaky by Leaky ReLU with a slope of 0.2 as they use in the paper, inject\_noise1, inject\_noise2 by the InjectNoise, adain1, and adain2 by AdaIN**

**In the forward part, we send x, and we pass it to conv1 then to inject\_noise1 with leaky, then we normalize it with adain1, and again we pass that into conv2 then to inject\_noise2 with leaky and we normalize it with adain2. And finally, we return x.**

**in the init part let's initialize 'starting\_constant' by constant 4 x 4 tensor which is put through an iteration of the generator, map by 'MappingNetwork', initial\_adain1, initial\_adain2 by AdaIN, initial\_noise1, initial\_noise2 by InjectNoise, initial\_conv by a conv layer that map in\_channels to itself, leaky by Leaky ReLU with a slope of 0.2, initial\_rgb by WSConv2d that maps in\_channels to img\_channels wi=hich is 3 for RGB, prog\_blocks by ModuleList() that will contain all the progressive blocks, and rgb\_blocks by ModuleList() that will contain all the RGB blocks.**

**To fade in new layers (an origin component of ProGAN), we add the fade\_in part, which we send alpha, scaled, and generated . The reason why we use tanh is that will be the output(the generated image) and we want the pixels to be range between 1 and -1.**

**In the forward part, we send the noise (Z\_dim), the alpha value which is going to fade in slowly during training (alpha is between 0 and 1), and steps which is the number of the current resolution that we are working with, we pass x into the map to get the intermediate noise vector W, we pass starting\_constant to initial\_noise1, apply for it and for W initial\_adain1, then we passe it into initial\_conv, and again we add initial\_noise2 for it with leaky as activation function, and apply for it and W initial\_adain2. Then we check if steps = 0 if it is, then all we want to do is run it through the initial RGB and we have done, otherwise, we loop over the number of steps, and in each loop we upscaling(upscaled) and we run through the progressive block that corresponds to that resolution(out). In the end, we return fade\_in that takes alpha, final\_out, and final\_upscaled after mapping it to RGB.**

**In the code next you can find the generate\_examples function that takes the generator gen, the number of steps to identify the current resolution, and a number n=100.**

**Next we create gradient penalty :-**

**he gradient\_penalty function calculates the gradient penalty used in the Wasserstein GAN (WGAN) to enforce the Lipschitz constraint on the critic (discriminator).**

**critic is the discriminator network, real and fake are batches of real and generated images respectively, alpha is the interpolation parameter for fade-in (typically used in progressive growing GANs), train\_step is the current training step (useful for progressive growing GANs), and device specifies whether to run computations on CPU or GPU. The gradient\_penalty function helps stabilize training by penalizing gradients that deviate from the desired Lipschitz constraint, thereby improving the overall quality of generated images.**

**And last we Train our Network :-**

**For the train function, we send critic (which is the discriminator), gen(generator), loader, dataset, step, alpha, and optimizer for the generator and for the critic.We start by looping over all the mini-batch sizes that we create with the DataLoader, and we take just the images because we don't need a label.**

**Then we set up the training for the discriminator\Critic when we want to maximize E(critic(real)) - E(critic(fake)). This equation means how much the critic can distinguish between real and fake images.**

**After that, we set up the training for the generator when we want to maximize E(critic(fake)).**

**Finally, we update the loop and the alpha value for fade\_in and ensure that it is between 0 and 1, and we return it.**

**Now since we have everything let's put them together to train our StyleGAN.**

**We start by initializing the generator, the discriminator/critic, and optimizers, then convert the generator and the critic into train mode, then loop over PROGRESSIVE\_EPOCHS, and in each loop, we call the train function number of epoch times, then we generate some fake images and save them, as a result, using generate\_examples function, and finally, we progress to the next image resolution**

**And at last we define a function called “generate\_image\_from\_text” which will take input any text provided by the user and then it will generate the corresponding image based on the text.**

**THANK YOU!!**