Multi-Modal Image Generation for News Stories

Rajwinder Mahal
rsm2207@columbia.edu

Arjun Krishna
ak4471@columbia.edu

Waleed Khan
wk2383@columbia.edu

Abstract

Online news publishers are constantly searching for ways to capture the attention of their readers and stand out from the competition. One way to do this is by using eye-catching images for news stories to boost reader engagement. However, finding and creating these images can be time consuming and expensive, especially for small publishers. To address this issue, we explore the feasibility of using a transformer based model to generate images using news headlines. In this paper, we propose a multi-modal image generation model for news stories. The model is trained on a dataset of news articles and images to generate images that are visually appealing and consistent with the news story.

1. Introduction

In the world of journalism and media, capturing and retaining readers' attention is a top priority, especially when there are hundreds of publishers publishing news stories on the internet. One way to achieve this is by using visually appealing images, high quality cover images for news stories. However, this task can be time consuming and costly as it is not very straightforward to brainstorm, find and create high quality images that are consistent with the news story as well as appealing to grab readers' attention. This is even more challenging for smaller media outlets and individual journalists that have limited resources and lack the financial and personnel resources of larger media outlets.

Existing platforms like Unsplash and Shutterstock offer solutions to these challenges but come with their own limitations. High-quality images can come at a steep cost, and images may not be exclusive to the publisher. Moreover, finding high quality images on these platforms can also be challenging, leaving media outlets struggling to stand out from the crowd.

In addition to the challenges faced by journalists and media outlets in finding and creating images for news stories, recent advances in generative models have presented new opportunities to explore new ways to solve many problems. Models such as DALL-E and ChatGPT have demonstrated remarkable capabilities in generating images and text. These advances in generative models sparked our curiosity about the potential for these models to generate high quality images for news stories. As a result, we propose a multi-modal image generation model for news stories that leverages the power of generative adversarial networks and large language models to generate visually appealing images that are relevant and consistent with the news stories. We conducted experiments using Vector Quantized Generative Adversarial Network (VQGAN) [2] and Bidirectional and Auto-Regressive Transformers (BART) [5]. We trained these models on a dataset of news articles and corresponding images to generate visually appealing images. We think that our model has the potential to help publishers create exclusive, high quality images that boost readers' engagement with the news content.

2. Related Work

In recent years, generative models have demonstrated impressive capabilities in generating high quality images and it is an active area of research. DALL-E [7] is a generative model developed by OpenAI that uses transformer based architecture to generate images from textual descriptions. At a high level, DALL-E works by first encoding the textual input into a high dimensional latent space using a transformer based encoder. This latent representation is then passed through a series of generative convolutional networks to produce a 2D grid of pixels. It also uses generative adversarial networks (GANs) to further refine generated images which helps to improve the visual realism of the output images. Overall, DALL-E has shown to generate highly realistic and complex images.

Another popular model is StyleGAN [4] which uses generative adversarial networks (GANs) based architecture to generate images with high levels of detail and style variation. It is developed by Nvidia and has shown to generate impressive images across a range of domains such as art, fashion, and architecture. Unlike traditional generative models which generate images by sampling from a fixed la-

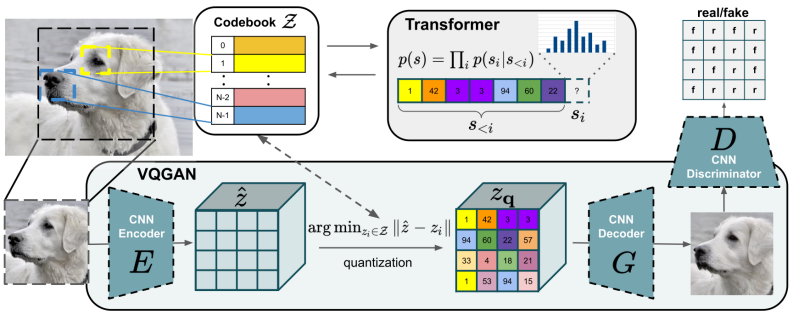


Figure 1. A convolutional VQGAN that leans a codebook of pixels by using a perceptual loss and a GAN generator loss.

tent space, StyleGAN generates images by first generating a set of intermediate feature maps and then using adaptive instance normalization to combine these features with the learned style information. At a high level, it works by first mapping a random noise vector to an intermediate latent space which is then used to generate a set of intermediate feature maps that capture various aspects of the image such as shape, color, and texture. The model then uses adaptive normalization to combine these intermediate feature maps with the style information learned from the training data. Overall, StyleGAN has shown impressive results in image generation and has been used in a wide range of applications such as image generation, style transfer, and image editing.

Furthermore, Diffusion models [3] have recently achieved state-of-the-art results on a variety of tasks such as image generation, text generation, and audio generation. These diffusion models work by gradually adding noise to a latent representation of the data and then learn to reverse the noise process to generate new data. Models such as Imagen [8] and Stable Diffusion have shown state-of-the-art results in image generation and represent a promising new direction in generative modeling.

While models such as DALL-E, StyleGAN, and Imagen have demonstrated impressive capabilities in generating high quality images, they have not been specifically designed for generating images for news stories. Our intuition is that these models can be improved to generate images that are relevant and consistent with a given news story. To test our intuition, we trained our model on a news dataset with corresponding images. Our model builds on the strengths of these existing models while also addressing the unique challenges and requirements of generating images for news stories.

3. Method

In this paper, we conducted experiments to generate images using Vector Quantized Generative Adversarial Network (VQGAN) and Bidirectional and Auto-Regressive Transformers (BART). In this section, we describe how we used these models to create a simple multi-modal architecture that is trained using limited resources and data, yet produces impressive results. Our model architecture approach is inspired by DALL-E mini, which is a much smaller model as compared to the original DALL-E, yet it has shown impressive results in image generation.

3.1. VQGAN

Vector Quantized Generative Adversarial Network (VQGAN) is a generative model that combines elements of both generative adversarial networks (GANs) and vector quantization to generate high quality images. VQGAN uses a codebook of embedding vectors to quantize the feature maps generated by the generator network. This allows VQGAN to generate images that are semantically meaningful and diverse. For our use case, we used VQGAN to encode images into a sequence of discrete tokens that can be used in a transformer model. Our approach used a pre-trained convolutional VQGAN model that is trained on ImageNet. In figure 1, a convolutional VQGAN learns a codebook of pixels by using a perceptual loss and a GAN generator loss. It uses Variational Autoencoder (VAE) to compress the input data into a lower dimensional latent space, i.e. a codebook. This codebook enables strong compression while retaining high perceptual quality. For example, representing a 256x256 image requires 65,536 discrete tokens. But by using the codebook, this 256x256 image can be divided into 16 sub-images with a 4x4 grid which can be represented using only 256 discrete tokens. For our use case, we used the encoder part of VQGAN in figure 1 to represent our 256x256 images using 256 discrete tokens from a vocabulary of size 16,384.

3.2. BART

Bidirectional and Auto-Regressive Transformers (BART) is a sequence-to-sequence transformer model developed by Facebook AI Research (FAIR). It is based on the transformer architecture introduced in the original paper by Vaswani et al. (2017), and uses a combination of

denoising auto-encoding and back-translation techniques. During pre-training, it uses a noise function to corrupt the text data so that the model can learn to reconstruct the original text. It utilizes a standard sequence-to-sequence transformer architecture except that it uses GELU activation functions instead of ReLU. The base model has 6 layers in the encoder and decoder. Each decoder layer performs cross attention over the final hidden layer of the encoder. In figure 2, the encoder inputs are corrupted using mask symbols which are then encoded using the bidirectional encoder. Then the likelihood of original inputs is calculated with an autoregressive decoder. In our case, we used a bidirectional encoder to encode captions and headlines and trained an autoregressive decoder to output image tokens (details in the next section).

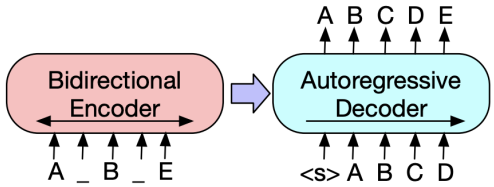


Figure 2. BART encoder and decoder.

3.3. Our architecture

Our implementation uses multi-modal approach, inspired by DALLE-Mini [1], where we use VQGAN and BART to generate images. We used a pre-trained VQGAN model which is trained on ImageNet. Based on our research, we found that the pre-trained model doesn't perform well on faces, so we fine-tuned it on our news dataset. We also used a pre-trained large BART model. We first separated the encoder and decoder from the model and used them in our custom PyTorch Lightning module. We used the pre-trained BART encoder as it is. For the BART decoder, we first set its weights to random values and then modified its configuration to output 257 tokens (256 images tokens + BOS token). We also modified its vocab size to 16,388 (16,384 VQGAN codebook size + BOS + EOS + Padding + Decoder Start token). Our custom module manually handles the forward pass between the encoder and decoder. It has a language modeling head that takes decoder output to calculate the logits. We use softmax cross-entropy loss between the model prediction logits and the original image encodings from the VQGAN encoder.

In Figure 3, the inputs are image caption/headline with corresponding image. We use BART tokenizer to tokenize text into input ids. To convert image into tokens, we use VQGAN encoder to represent image as 256 discrete tokens. Then the encoded text is fed to the BART encoder to capture the contextual information and semantic meaning of the input. This encoder output is then fed to the BART decoder along with encoded image tokens from VQGAN encoder. The goal of the decoder is to learn how to output the next image token in the given sequence. The decoder output then can be fed to VQGAN decoder to reconstruct the generated image.

3.4. Metrics

Our main goal is to produce images that could be reasonably judged to be intelligible and relevant to the given inputs (headlines/captions). For fine-tuning VQGAN, we used Learned Perceptual Image Patch Similarity (LPIPS) [9] loss as the primary metric to optimize the model. LPIPS measures the perceptual distance between the generated image, in our case a reconstructed image, and the ground truth image in terms of their perceptual similarity. For evaluating different VQGAN experiments and final image reconstructions, we used Peak Signal-to-Noise Ratio (PSNR), Structural Similarity Index (SSIM) and LPIPS. PSNR measures the difference between the original and reconstructed images in terms of their signal-to-noise ratio. SSIM measures the structural similarity between the two images. By using multiple metrics, we gain a more comprehensive understanding of the performance of the VQGAN experiments.

For training the BART decoder, we used the softmax cross-entropy loss as the primary metric to optimize the model. The softmax cross-entropy measures the difference between the predicted and actual probability distributions of the target tokens in the output sequence. By minimizing the loss, the decoder learns to generate sequences that are more similar to the ground truth.

3.5. Training Data

Our initial plan was to scrape news articles with corresponding images from popular news publishers such as TechCrunch and the New York Times. We implemented a simple web crawler to scrape news articles from a given list of target domains. As we started to scrape articles, our crawler performed very well on scraping the article content including headlines. However, we find it very difficult to extract the primary image from the page as each page has multiple images including images related to ads. Moreover, most online news publishers only list image credits as the image captions rather than a more descriptive caption. So, we decided to use an existing dataset. We found three datasets that fit our criteria: VisualNews [6], NYTimes800k, and GoodNews. However, these datasets were only available on request. So, we requested access to these datasets and we first got access to VisualNews dataset. It contains a large collection of news stories from The Guardian, BBC, USA Today, and The Washington Post.

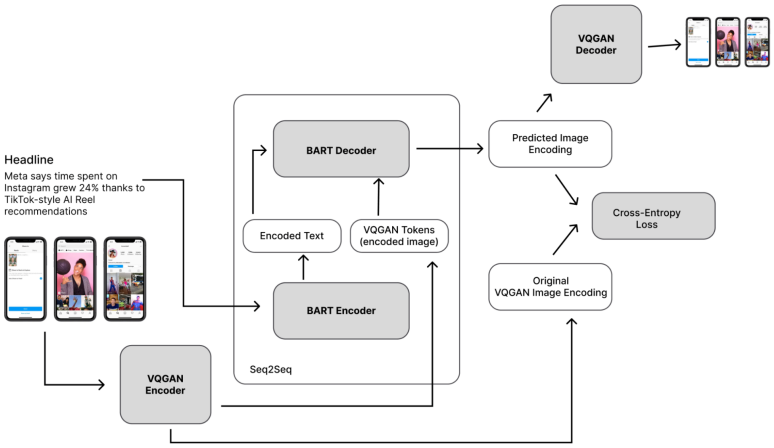


Figure 3. Training pipeline for multi-modal image generation for news stories, inspired by DALL-E mini architecture.

It is composed of over 1.2 million images along with associated news articles, image captions, author information and other metadata.

One of the main data preprocessing we performed was to separate the images that have only captions from images that have both captions and headlines. We also split the data into train, validation, and test splits. We saved 10% of the original data for testing and used 10% of the remaining 90% data for the validation.

As the dataset is very large, we also pre-encoded images using fine-tuned VQGAN encoder to speed up the training. We also pre-encoded headlines and captions using BART tokenizer to avoid doing it during BART decoder training. Doing so, we avoided encoding images and text on-the-fly during training. Moreover, we resized all original images to 256x256 and took the center crop before encoding the images using VQGAN encoder.

4. Fine-Tuning VQGAN

In this section, we describe in more details our process for fine-tuning VQGAN and various experiments we conducted, along with evaluation results using metrics such as PSNR, SSIM, and LPIPS.

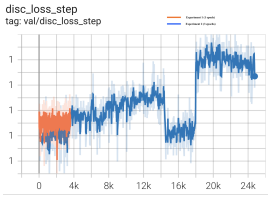
4.1. Experiment Setup

In our experiments, we used the original VQGAN implementation by Esser et al. Figure 4 shows that the VQGAN has about 90 million total parameters with about 76 million trainable parameters. Most of the architecture related code is adapted from the Taming Transformers repository, however, we end up making some adjustments since the original code was dependent on multiple shared components in the Taming Transformers repository. We decoupled

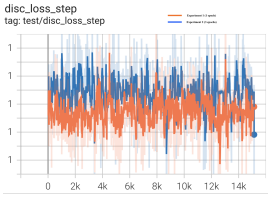
```
==========================================================
Layer (type:depth-idx)                   Param #
==========================================================
├─Encoder: 1-1                           --
|    └─Conv2d: 2-1                       3,584
|    └─ModuleList: 2-2                   --
|    |    └─Module: 3-1                  738,944
|    |    └─Module: 3-2                  738,944
|    |    └─Module: 3-3                  2,690,304
|    |    └─Module: 3-4                  2,952,448
|    |    └─Module: 3-5                  10,498,048
|    └─Module: 2-3                       --
|    |    └─ResnetBlock: 3-6             4,721,664
|    |    └─AttnBlock: 3-7               1,051,648
|    |    └─ResnetBlock: 3-8             4,721,664
|    └─GroupNorm: 2-4                    1,024
|    └─Conv2d: 2-5                       1,179,904
├─Decoder: 1-2                           --
|    └─Conv2d: 2-6                       1,180,160
|    └─Module: 2-7                       --
|    |    └─ResnetBlock: 3-9             4,721,664
|    |    └─AttnBlock: 3-10              1,051,648
|    |    └─ResnetBlock: 3-11            4,721,664
|    └─ModuleList: 2-8                   --
|    |    └─Module: 3-12                 887,040
|    |    └─Module: 3-13                 1,215,232
|    |    └─Module: 3-14                 4,133,632
|    |    └─Module: 3-15                 4,855,296
|    |    └─Module: 3-16                 19,679,744
|    └─GroupNorm: 2-9                    256
|    └─Conv2d: 2-10                      3,459
├─VQLPIPSWithDiscriminator: 1-3          --
|    └─LPIPS: 2-11                       --
|    |    └─ScalingLayer: 3-17           --
|    |    └─vgg16: 3-18                  (14,714,688)
|    |    └─NetLinLayer: 3-19            (64)
|    |    └─NetLinLayer: 3-20            (128)
|    |    └─NetLinLayer: 3-21            (256)
|    |    └─NetLinLayer: 3-22            (512)
|    |    └─NetLinLayer: 3-23            (512)
|    └─NLayerDiscriminator: 2-12         --
|    |    └─Sequential: 3-24             663,361
├─VectorQuantizer2: 1-4                  --
|    └─Embedding: 2-13                   4,194,304
├─Conv2d: 1-5                            65,792
├─Conv2d: 1-6                            65,792
==========================================================
Total params: 91,453,380
Trainable params: 76,737,220
Non-trainable params: 14,716,160
==========================================================
```

Figure 4. VQGAN model summary

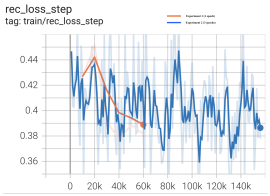
all the needed components and implemented our own training script. We fine-tuned VQGAN on over 800k images which proved to be time consuming when done using a single GPU. To tackle this issue, we modified the model to use



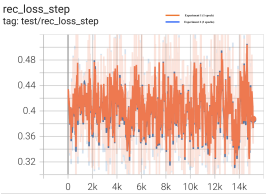
(a) Discriminator validation loss



(b) Discriminator testing loss



(c) Image reconstruction training loss



(d) Image reconstruction testing loss

Figure 5. Loss graphs from the VQGAN Fine-Tuning. Experiment 1 is in blue, and Experiment 2 is in orange.

Test metric	DataLoader 0
test/ae_loss_epoch	0.4414067566394806
test/d_weight_epoch	0.0
test/disc_factor_epoch	1.0
test/disc_loss_epoch	1.0000025033950806
test/g_loss_epoch	-0.5763425827026367
test/logits_fake_epoch	0.5763425827026367
test/logits_real_epoch	0.5762884616851807
test/nll_loss_epoch	0.4014285206794739
test/p_loss_epoch	0.2730328440666199
test/quant_loss_epoch	0.03997941315174103
test/rec_loss_epoch	0.4014285206794739

(a) Experiment 1 (1 epoch) test set results

Test metric	DataLoader 0
test/ae_loss_epoch	0.4393095076084137
test/d_weight_epoch	0.0
test/disc_factor_epoch	1.0
test/disc_loss_epoch	1.000008463859558
test/g_loss_epoch	-0.774586021900177
test/logits_fake_epoch	0.774586021900177
test/logits_real_epoch	0.7744798064231873
test/nll_loss_epoch	0.39948326349258423
test/p_loss_epoch	0.26881957054138184
test/quant_loss_epoch	0.03982929140329361
test/rec_loss_epoch	0.39948326349258423

(b) Experiment 2 (3 epochs) test test results

Figure 6. VQGAN fine-tuning results for Experiments 1 and 2

PyTorch Lightning. We used four Tesla V100 GPUs. It is common to set the number of data loading workers to the number of CPU cores. In our case, we used 48 data loading workers which significantly reduced GPU idle time. To maximize the GPU memory use, we used a batch size of 4 per GPU with an effective batch size 16 (4 GPUs). We set the learning rate to be the same as the one used in the original pre-trained checkpoint, which is 4.5e-6. We also used the distributed sampler for data loaders for efficient multi-GPU training.

By making these modifications and leveraging the benefits of PyTorch Lightning, we were able to fine-tune our model in about 5 hours for 1 epoch, 4x reduction in training time.

4.2. Experiments

We tested three different versions of the VQGAN. The first one ("Original"), used the pre-trained checkpoint that was provided, without any fine-turning. The pre-trained model was trained on ImageNet for 12 epochs. The second version ("Experiment 1") was trained on our VisualNews dataset for 1 epoch (5 hours training time), and the third version ("Experiment 2") was trained for 3 epochs (15 hours training time).

In GAN architecture, the goal of the generator, in our case reconstructor, is to generate samples that are similar to the real data so that the discriminator can't distinguish between the real and fake samples. As shown in Figure 5a and 5b, the discriminator loss was unstable for both experiments, but it was higher on average for Experiment 2, meaning that the generator/reconstructor was able to generate higher quality images that the discriminator could not distinguish.

Figure 5c and 5d show the reconstruction loss for both experiments. Although the loss is very unstable throughout the training process, it tends to go downward. We can see in figures 6a and 6b that the reconstruction loss in Experiment 2 is slightly lower than that in Experiment 1. However, there is a big tradeoff between the training time and the slight improvement in the reconstruction loss. For example, Experiment 1 took about 5 hours to train and Experiment 2 took about 15 hours, almost 3 times more time. However, there is only a slight improvement in the reconstruction loss between the both experiments. The test reconstruction loss in Experiment 1 is around 0.40 while in Experiment 2, it is around 0.399.

4.3. Evaluation

As shown in Table 1 below, compared to the pre-trained checkpoint, Experiment 1 achieved better results for PSNR and SSIM, but a worse score for LPIPS.

	PSNR	SSIM	LPIPS
Pre-trained checkpoint	20.24	0.44	0.001809
Experiment 1	20.49	0.47	0.001874
Experiment 2	20.27	0.46	0.001758

Table 1. Evaluation of the quality of reconstructed images using various metrics.

Experiment 2 also achieved slightly better results for

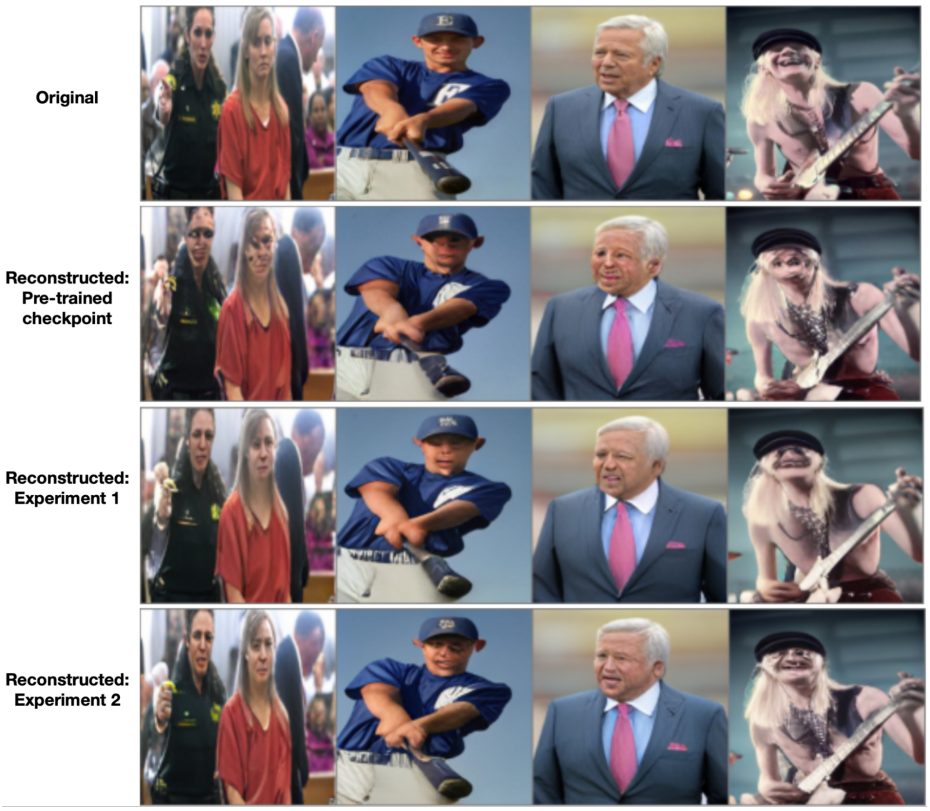


Figure 7. Image reconstruction using the different VQGAN checkpoints.

PSNR and SSIM, as well as a better score for LPIPS. The PSNR and SSIM metrics correspond more directly to the quantitative difference between the original and reconstructed images, whereas LPIPS corresponds to the perceptual similarity between the two images, measured by their differences in the lower-level representations/features learned by another neural network (VGG-19). This means that Experiment 2's outputs are generally more accurate in terms of human perception and this can be clearly seen in figure 7. As we can see in figure 7, all of the models capture the core features of each of the images, but do not reconstruct specific details, specifically the faces. But the fine-tuned models do reconstruct the faces slightly more accurately, and the second experiment in particular does a better job at rendering small details such as hands.

Overall, we found that there was a fairly large impact for fine-tuning the model on our VisualNews dataset, but it is arguable whether the minor improvements in Experiment 2 were worth the tradeoff of needing 3x more training time.

5. Training BART Decoder

In this section, we describe in more details our process for training the BART decoder and various experiments we conducted, along with the final evaluation results using metrics such as PSNR, SSIM, and LPIPS. During the training of the BART decoder, we used softmax cross-entropy loss.

5.1. Experiment Setup

In our experiments, we used the original BART encoder and decoder from the pre-trained BART Large from the Hugging Face. We first separated the encoder and decoder from the pre-trained model. We used the encoder as it is with frozen weights. For the decoder, we set its weights to random and made some configuration changes based on our requirements such as vocab size, input/output sequence length, etc. To combine the encoder and decoder together, we wrote our own module based on PyTorch Lightning and manually handled forward pass including the decoder inputs and calculating logits. Figure 8 shows that our model has about 439 million total parameters but only half of it (235 million) are trainable. This is due to our approach to freeze the encoder weights to speed up the training. Since our dataset is very large, we used V100 GPUs with multi-GPU training using PyTorch Lightning. We set the number of data loading workers to the number of available CPU cores, which is 32 in our case.

```
==========================================================================
Layer (type:depth-idx)                                  Param #
==========================================================================
├─BartEncoder: 1-1                                      --
|    └─Embedding: 2-1                                   (51,471,360)
|    └─BartLearnedPositionalEmbedding: 2-2              (1,050,624)
|    └─ModuleList: 2-3                                  --
|    |    └─BartEncoderLayer: 3-1                       (12,596,224)
|    |    └─BartEncoderLayer: 3-2                       (12,596,224)
|    |    └─BartEncoderLayer: 3-3                       (12,596,224)
|    |    └─BartEncoderLayer: 3-4                       (12,596,224)
|    |    └─BartEncoderLayer: 3-5                       (12,596,224)
|    |    └─BartEncoderLayer: 3-6                       (12,596,224)
|    |    └─BartEncoderLayer: 3-7                       (12,596,224)
|    |    └─BartEncoderLayer: 3-8                       (12,596,224)
|    |    └─BartEncoderLayer: 3-9                       (12,596,224)
|    |    └─BartEncoderLayer: 3-10                      (12,596,224)
|    |    └─BartEncoderLayer: 3-11                      (12,596,224)
|    |    └─BartEncoderLayer: 3-12                      (12,596,224)
|    └─LayerNorm: 2-4                                   (2,048)
├─BartDecoder: 1-2                                      --
|    └─Embedding: 2-5                                   16,781,312
|    └─BartLearnedPositionalEmbedding: 2-6              265,216
|    └─ModuleList: 2-7                                  --
|    |    └─BartDecoderLayer: 3-13                      16,796,672
|    |    └─BartDecoderLayer: 3-14                      16,796,672
|    |    └─BartDecoderLayer: 3-15                      16,796,672
|    |    └─BartDecoderLayer: 3-16                      16,796,672
|    |    └─BartDecoderLayer: 3-17                      16,796,672
|    |    └─BartDecoderLayer: 3-18                      16,796,672
|    |    └─BartDecoderLayer: 3-19                      16,796,672
|    |    └─BartDecoderLayer: 3-20                      16,796,672
|    |    └─BartDecoderLayer: 3-21                      16,796,672
|    |    └─BartDecoderLayer: 3-22                      16,796,672
|    |    └─BartDecoderLayer: 3-23                      16,796,672
|    |    └─BartDecoderLayer: 3-24                      16,796,672
|    └─LayerNorm: 2-8                                   2,048
├─Linear: 1-3                                           16,781,312
├─CrossEntropyLoss: 1-4                                 --
==========================================================================
Total params: 439,068,672
Trainable params: 235,389,952
Non-trainable params: 203,678,720
==========================================================================
```

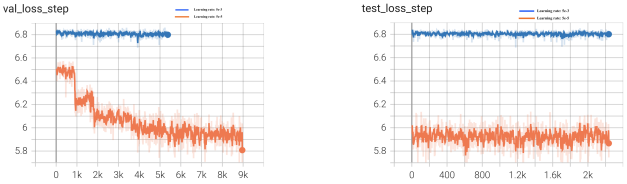
Figure 8. BART encoder-decoder model summary.

5.2. Batch Size

The batch size is a hyperparamter that determines the number of training samples used in one iteration of gradient calculation. A larger batch size typically results in faster convergence but requires more memory and computation resources. In our case, we used a batch size of 8 per GPU to maximize GPU memory utilization. We tried to go higher but that resulted in a memory error. In total we used 4 GPUs, resulting in an effective batch size of 32.

5.3. Learning Rate

We experimented with two different base learning rates, 5e-3 and 5e-5. We used a linear learning rate with a warmup scheduler from the transformers library. It is a popular practice to increase the learning rate during the initial phase of training, followed by a linear decrease in the learning rate as training progresses. We used it to improve the model convergence and prevent the model from getting stuck in the local minima. By gradually increasing the learning rate during the initial warmup phase, the model is able to explore a larger portion of the parameter space which helps in faster convergence and improved generalization. We initialized the learning rate to a small non-zero value and then gradually increased it to maximum value of 5e-3 or 5e-3 depending on the experiment during the first 10% of the total training steps (the warmup phase). By doing so, we aimed to prevent the model from learning patterns too quickly, which could potentially require unlearning later on. This is especially relevant when fine-tuning a model on a new dataset, as rapid parameter changes may occur during initial training.



(a) Decoder validation loss (b) Decoder testing loss

Figure 9. BART decoder test loss with different learning rates. α = 5e-3 is in blue, and α =5e-5 is in orange.

We experimented with 2 different base learning rates. In figure 9a and 9b, it is clear that the learning rate of 5e-5 performs much better compared to a base learning rate of 5e-3. Due to limited resources, we didn't perform many experiments to find the optimal learning rate but based on these two experiments, we decided to use a base learning rate of 5e-5.

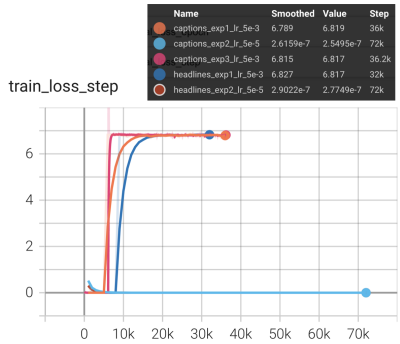
5.4. Experiments

In this section, we provide more details on different experiments conducted while training the BART decoder. We conducted experiments using different datasets created using the larger VisualNews dataset: a dataset of about 250k headlines with corresponding images, a dataset of about 250k captions with corresponding images, and another dataset of about 850k captions with corresponding images.

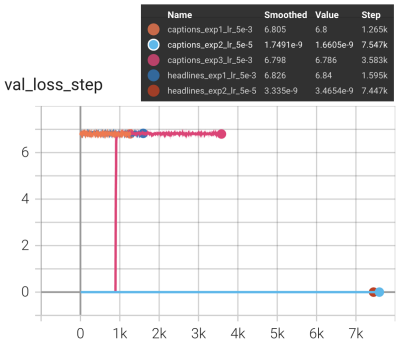
5.4.1 Initial Experiments

Initially, we conducted two parallel experiments, namely captions_exp1 and headlines_exp1. Unfortunately, both experiments resulted in high training and validation loss with no signs of improvement in model performance, as shown in figure 10a and 10b. To address this issue, we decided to use a linear learning rate scheduler with a warmup, using a base learning rate of 5e-5. As we conducted two more experiments, captions_exp2 and headlines_exp2, we observed a much lower loss, almost close to zero. As we couldn't figure out the main cause of this significantly lower loss, we conducted another experiment, captions_exp3, on the larger captions dataset with over 850k images. Our intuition was that our model might perform better if trained on a much larger dataset. However, this also resulted in similar problems where the model's performance failed to improve over time.

Initially, we suspected that there might be issues with our training process or model implementation. However, we later discovered that we made a small mistake in our



(a) Decoder training loss



(b) Decoder validation loss

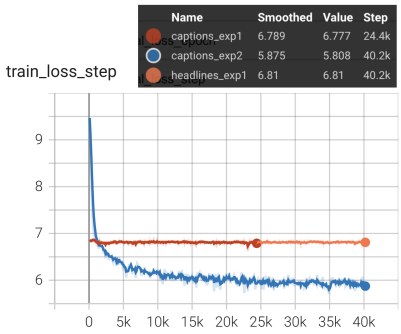
Figure 10. BART Decoder losses (before right shifting decoder inputs)

decoder forward pass where we neglected to right shift the decoder inputs. This resulted in the decoder being given the entire input it needs to output instead of learning to predict the next token in the sequence, and hence impeding the learning process. After spending a lot of time and GCP credits, we finally resolved with issue with the help of Prof. Austin Reiter.

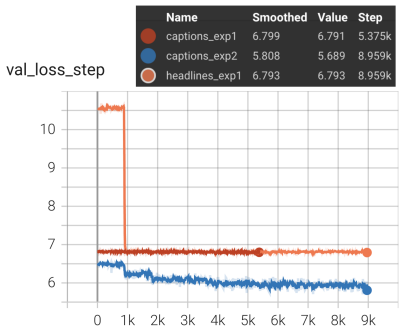
After fixing the issue, our model's performance improved significantly in subsequent experiments conducted in section 5.4.2. Looking back, we believe that this problem could have been avoided if we had used the BartForConditionalGeneration implementation from the transformers library, with some modifications, instead of manually handling the forward pass, decoder inputs and calculating the logits. Nonetheless, this was a valuable learning experience that highlights the importance of careful implementation and debugging during the development process.

5.4.2 Experiments after fixing decoder inputs (right shifting)

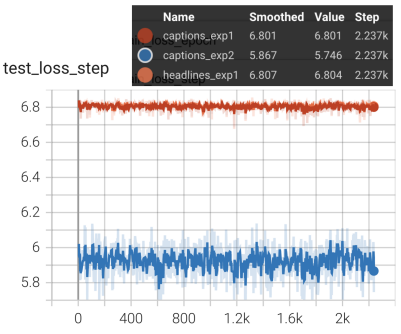
As we had limited time to test models after fixing the decoder inputs, we chose 3 models to test: headlines_exp1, captions_exp1, and captions_exp2. The headline model used headlines data as the input for training and the captions



(a) Decoder train loss



(b) Decoder validation loss



(c) Decoder test loss

Figure 11. BART Decoder losses (after right shifting decoder inputs)

models used captions data as the input. The captions_exp1 model used a learning rate of 5e-3, and the other two models used a learning rate of 5e-5. For all three experiments, we used a linear learning rate scheduler with a warmup during the first 10% of the total training steps.

Both of the exp1 models quickly dropped their loss values, but they then plateaued and did not improve further over the course of training. On the other hand, the exp2 model was able to continue learning over the course of its epochs, though its loss was beginning to plateau as well near the end of the training. We think that the model in exp2 can be improved if we train it on larger dataset (captions dataset of 850k images), however, due to limited resources left at the end, we couldn't experiment any further. Also, table

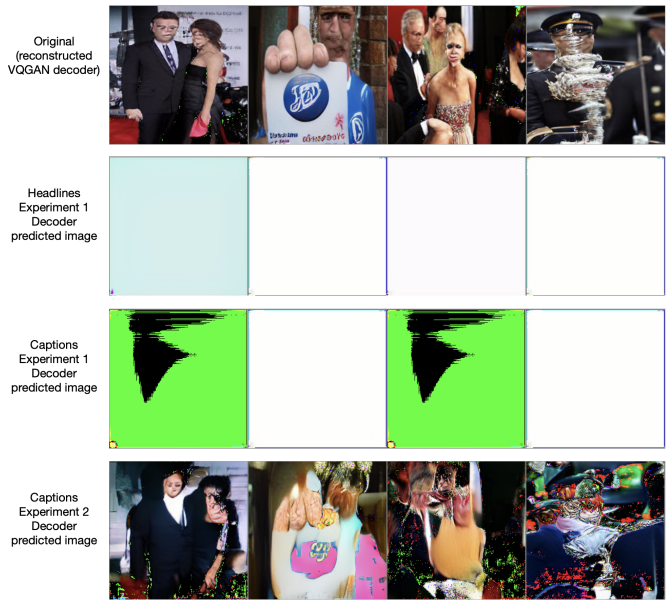


Figure 12. BART Decoder image generation during training, using different inputs and learning rates

2 shows that the test loss for all 3 models was pretty well correlated with the training loss. As shown before, both the captions_exp1 and headlines_exp1 models ended up at a significantly higher loss than the captions_exp2 model.

5.5. Evaluation

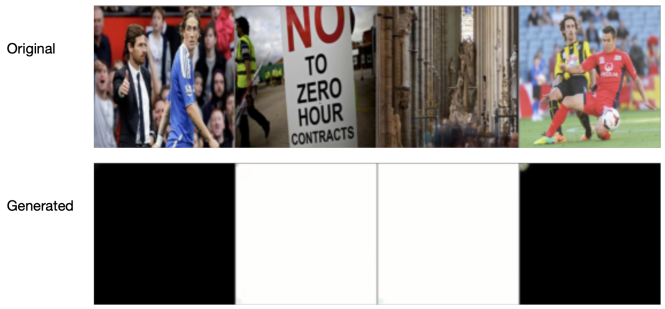


Figure 13. BART Decoder image generation during testing (model captions_exp2)

As shown in Figure 12, only the captions_exp2 model was able to begin generating images from the input tokens, while both of the exp1 models were not able to generate anything meaningful. The captions_exp2 model is beginning to learn the general features present in the original images, but the results are still very distorted and not realistic. We think that the performance of captions_exp2 model can be further improved if we train it on a larger dataset as we have seen earlier that its loss plateaued towards the end. It is also possible that experimenting with different learning rates would produce better results.

Moreover, the image generation quality of the captions_exp2 model did not carry over from training to testing. During training time, we gave the model encodings for both the caption text as well as the image, and had the model learn to predict the next token after we right shifted the decoder inputs. During evaluation, we give model random captions and tried to generate images. However, the model was only able to generate completely black or white images, as shown in Figure 13.

Model Name	Text Input	Learning Rate	Test Loss
captions_exp1	captions	5e-3	6.80
captions_exp2	captions	5e-5	5.92
headlines_exp1	headlines	5e-5	6.81

Table 2. Test loss for all 3 models

6. Challenges

During the initial phases of the project, we encountered a steep learning curve that posed a significant challenge. In the early stages, we found it difficult to gather information on the DALL-E mini implementation to understand its architecture. We found several repositories with JAX implementation of DALL-E mini, however, these repositories presented us with not only technical difficulties but also created a lot of confusion. Most of these repositories either handled only the inference part or were doing

much more than the simple implementation of the original DALL-E mini architecture. However, we finally decided to look into the transformers source code for the BART model, specifically BartForConditionalGeneration. This helped us understand how Bart can be used to generate any sequence, in our case image tokens, given an input sequence.

We also encountered a major challenge while fine-tuning the VQGAN. The goal of fine-tuning is to enable the model to learn codebooks and representations that were better suited to our dataset. However, during our experimentation, we discovered that fine-tuning didn't result in significant improvements in performance. Our hypothesis is that this might be due to the fact the original VQGAN model is already pre-trained on ImageNet which is a large dataset with diverse images. So, fine-tuning on our dataset didn't result in significant improvements in performance but we do see a slight improvement when reconstructing images with faces.

As discussed in section 5.4.1, resolving the significantly lower loss of BART decoder was one of the major challenges that consumed most of our time and GCP credits. Even though the loss seemed to be going down at a proper rate (as shown by graphs), the images generated were nearly identical to the inputted images, even though the model was supposed to be generating new images from the headline text alone. We ended up realizing that we were directly passing the image tokens both as inputs and labels, causing the model to try to predict the same image. Instead, we had to shift the labels to the right by 1, so that the model would try to probabilistically generate the next token in the series. Overall, it is a valuable learning experience that highlights the importance of careful implementation and debugging during the development process.

7. Conclusion and Future Work

In this paper, we conducted experiments to generate images using Vector Quantized Generative Adversarial Network (VQGAN) and Bidirectional and Auto-Regressive Transformers (BART). We implemented a simple multimodal architecture, however, the model is unable to produce reasonable results. We think that it can be improved as we have seen in some of later experiments. We have seen that after fixing the right shifting decoder inputs, our model tends to generate reasonable results but at the same time, it starts to converge after some time. We think that if we pre-train it on the large captions dataset of 850k images and then fine-tune it on the headlines dataset, the model can produce better results. However, we couldn't experiment this as we used most of our GCP credits on fixing the issue with decoder inputs. Moreover, it is possible that there's a problem with our generate method, like we made a mistake in the decoder inputs, that is preventing the model to generate reasonable results.

Moreover, we used a simple greedy searcher to generate images. A greedy searcher produces a token one by one starting with the decoder start token as the input to the decoder. Although this is a simple approach, it will always produce similar results for the same input. So, a better approach would be to use beam search to produce multiple outputs and then use CLIP to select the best generated output. We also need to implement a user-friendly interface and we can explore setting up HuggingFace Spaces in the future to deploy our model for demonstration.

8. Source Code

Our code is available at https://github.com/mahalrs/newsgen.

Majority of the code is written from scratch except the VQGAN model implementation. We adapted VQGAN model code from the Taming Transformers repository, however, we made some adjustments since the original code was dependent on multiple shared components in the Taming Transformers repository. So, we decoupled all the needed components and implemented our own training script. We also converted the model to use our own custom PyTorch Lightning module.

References

[1] Boris Dayma, Suraj Patil, Pedro Cuenca, Khalid Saifullah, Tanishq Abraham, Phúc Lê Khc, Luke Melas, and Ritobrata Ghosh. Dall·e mini, 7 2021. 3

[2] Patrick Esser, Robin Rombach, and Björn Ommer. Taming transformers for high-resolution image synthesis, 2021. 1

[3] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models, 2020. 2

[4] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks, 2019. 1

[5] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension, 2019. 1

[6] Fuxiao Liu, Yinghan Wang, Tianlu Wang, and Vicente Ordonez. Visualnews : Benchmark and challenges in entity-aware image captioning, 2020. 3

[7] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation, 2021. 1

[8] Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily Denton, Seyed Kamyar Seyed Ghasemipour, Burcu Karagol Ayan, S. Sara Mahdavi, Rapha Gontijo Lopes,

Tim Salimans, Jonathan Ho, David J Fleet, and Mohammad Norouzi. Photorealistic text-to-image diffusion models with deep language understanding, 2022. 2

[9] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric, 2018. 3