Building a Conditional GAN for CIFAR-10

Reporting on my work for the first part of my Deep Learning Assignment 2.

Student ID: 2012072
Class: DAAA/FT/2B/01

Oh Tien Cheng

Next Steps

Forenote

This report summarizes multiple experiments I have conducted over the course of multiple notebooks. While this report does contain the most important code snippets, I would still recommend checking out the full source code for a clearer view of the experiments conducted.

Background

Problem Statement

My objective for this project is to attempt to create a GAN model that is able to generate realistic images based on information that is provided to it.

Such a task generally falls under two categories:

1. Conditional Image Generation: Given a class (e.g. a car), generate an image of that class
2. Controllable Image Generation: Given some description of features of the image (e.g. The car should be red in color), generate an image with those features

In this project, I hope to tackle the first category (Conditional Image Generation), with the view of eventually tackling the second category if time and resources permit.

> 💡 While I would preferably want to perform controllable image generation, my research has come into a few roadblocks.
>
> One approach is known as classifier gradients, where a trained classifier is used to identify if an image has certain features (e.g. sunglasses), and gradient ascent is performed on the latent space to get an image with that feature. For this approach, I am limited in a lack of time to find datasets containing labelled features of the image that also

> fit the CIFAR10 dataset, as well as the time necessary to train the extra classifier required.
>
> Another approach, introduced in the GANSpace paper, proposes to perform Principle Component Analysis on the latent space, to identify principle components which correspond to features of the corresponding image. However, the limitations of this approach are that it only works on certain GAN architectures (e.g. StyleGAN), and to be quite frank, I don't fully understand the approach enough to comfortably attempt to replicate this work.

So, what would I consider to be a success? Well, I would preferably hope to be able to reliably generate images conditionally that look real enough, based on

- Eye Power 🦹‍♀️
- Metrics like FID (described later in the report)

By real enough, I mean that I hope to get results that are close enough to some of the top GAN models out there, with minimal artifacts.

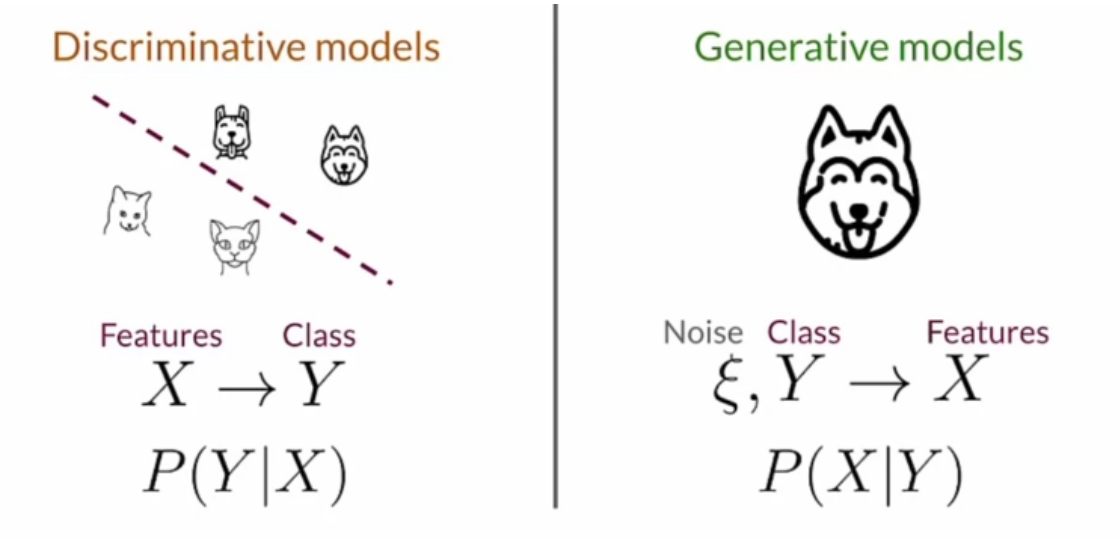
What is a GAN?

A Generative Adversarial Network (GAN) is a type of deep learning algorithm used as generative models. In a GAN, there are two networks: a generator, and a discriminator

The goal of the generator network is to create an output, that fools the discriminator into thinking that the output is real (e.g. a real image). In effect, the generator is attempting to learn the distribution of the data. The discriminator tries to predict between real images and fake images created by the generator. In that way, the two models compete adversarially in a game to try and continuously beat each other:

- the generator must create increasingly realistic data to fool the improving discriminator

- the discriminator needs to improve so that it does not let the generator fool it



Credit: Build Basic Generative Adversarial Networks, deeplearning.ai

Generator

The generator network tries to learn to make a realistic representation of a class. Typically this means it takes in a random input as noise, and possibly some class information, and tries to output a set of features that looks like it belongs to the stated class. That is, the model tries to capture the conditional probability distribution $P(X|Y)$.

> 💡 Random noise is needed to ensure that the model does not generate the same output every time.

Discriminator

The discriminator network tries to predict if a piece of data is real or a fake created by the generator. The primary purpose of the discriminator is during training, as a critic to ensure that the generator improves its output to become more realistic.

Issues with GAN Training

Training a GAN can be difficult, as there needs to be a balance between the generator and discriminator during training. If the generator is too strong, it will fool the discriminator all the time, and thus the discriminator is unable to provide feedback (in the form of weight updates during backpropagation) to the generator, stopping the generator from improving. On the other hand, if the discriminator is too strong, the generator will not be able to even begin at fooling the discriminator, preventing the generator from improving. Thus, a delicate balance between the skill level of the generator and discriminator must be maintained throughout training, or the GAN will fail to converge.

Another thing I need to look out for during training is the issue of mode collapse. A mode in a probability distribution is just an area with a high concentration of observations. In a dataset like CIFAR-10, there could be multiple modes of the data distribution, which are the most common images (in terms of similarity) for each class. Mode collapse occurs when the generator learns to fool the discriminator by producing examples from a mode in the training set (i.e. it learns a mode of the distribution), and hence stops learning as the generator is able to fool the discriminator every time with that single mode.

Use Cases of a GAN

Although the dataset used, CIFAR-10, is somewhat limiting in the image size and contents of the dataset, successfully creating a GAN that can generate images conditioned on class information would show that conditional image generation can be done, and hence the same principles applied to more varied datasets. This would allow for use cases like:

- Art Generation for Video Games

One of my dreams is to eventually build a role playing game (RPG) of my dreams, especially a game with a procedurally generated simulated world. Unfortunately, I'm quite bad at art in general (I

failed art back in Secondary School 😟), and so I would struggle to get the art assets (e.g. portraits) needed for a good game as I would either have to pay someone else to do it for me (expensive!), or get prebuilt art assets (a great way to make your game look like an asset flip scam).

So, if I could design a GAN to generate these art assets for me, I think it would be absolutely swell. And, if I manage to get it such that I can get more control over what the GAN generates for me (the goal of this project), I would even happier.

Data

CIFAR-10

CIFAR-10 is a labelled subset of the 80 million tiny images dataset. It consists of 60000 32x32 color images in 10 classes

- They are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck

There are 6000 images per class. CIFAR-10 splits data into 50000 training images, and 10000 test images. It is a common benchmark dataset, used to evaluate computer vision models. It is also commonly used as a benchmark for GAN training, as GAN training typically takes a long time, and so a smaller dataset like CIFAR with a lower resolution is easier to train, allowing GAN models to be more easily evaluated.

> 💡 CIFAR: Canadian Institute for Advanced Research

Source Code

> 💡 For all the experiments, PyTorch and PyTorch Lightning are used. PyTorch Lightning is an extension to PyTorch, designed to reduce the use of boilerplate code. This helps me save time writing code to do things like log my results, log the hyperparameters used, save model

> checkpoints, et cetera, giving me the time to experiment more with the architectures of the models, and play around with hyperparameters and training techniques.

To make the set up of the dataset easier and consistent for multiple experiments, a PyTorch Lightning (PL) Data Module is defined as follows. A `LightningDataModule` is a custom class in PL, which defines a structure for defining a dataset, such that it can be easily loaded by a PL Trainer. The full source code can be found in `data/dataset.py`.

```
class CIFAR10DataModule(LightningDataModule):
    def __init__(
        self,
        data_dir: str,
        batch_size: int,
        num_workers: int,
        transforms: Optional[list] = None,
    ):
        """Defines the CIFAR10 Dataset

        :param data_dir: Directory to save and load CIFAR10 dataset
        :type data_dir: str
        :param batch_size: Number of images in each batch
        :type batch_size: int
        :param num_workers: For use in parallel processing
        :type num_workers: int
        :param transforms: Preprocessing steps to be applied to data
        :type transforms: Optional[list], optional
        """
        super().__init__()
        self.data_dir = data_dir
        self.batch_size = batch_size
        self.num_workers = num_workers

        self.transform = T.Compose([T.ToTensor()] if transforms is N

        self.num_classes = 10
```

```
    def prepare_data(self):
        """Download dataset into data directory"""
        CIFAR10(self.data_dir, train=True, download=True)
        CIFAR10(self.data_dir, train=False, download=True)

    def setup(self, stage: Optional[str] = None):
        """Set up data splits for training, validation and testing

        :param stage: if set to "fit", only loads training and val s
        :type stage: Optional[str], optional
        """
        if stage == "fit" or stage is None:
            self.cifar_train = CIFAR10(
                self.data_dir, train=True, transform=self.transform
            )
            self.cifar_test = CIFAR10(
                self.data_dir, train=False, transform=self.transforr
            )
        if stage == "test" or stage is None:
            self.cifar_test = CIFAR10(
                self.data_dir, train=False, transform=self.transforr
            )

    def train_dataloader(self):
        return DataLoader(
            self.cifar_train, batch_size=self.batch_size, num_worker
        )

    def val_dataloader(self):
        return DataLoader(
            self.cifar_test, batch_size=self.batch_size, num_workers
        )
```

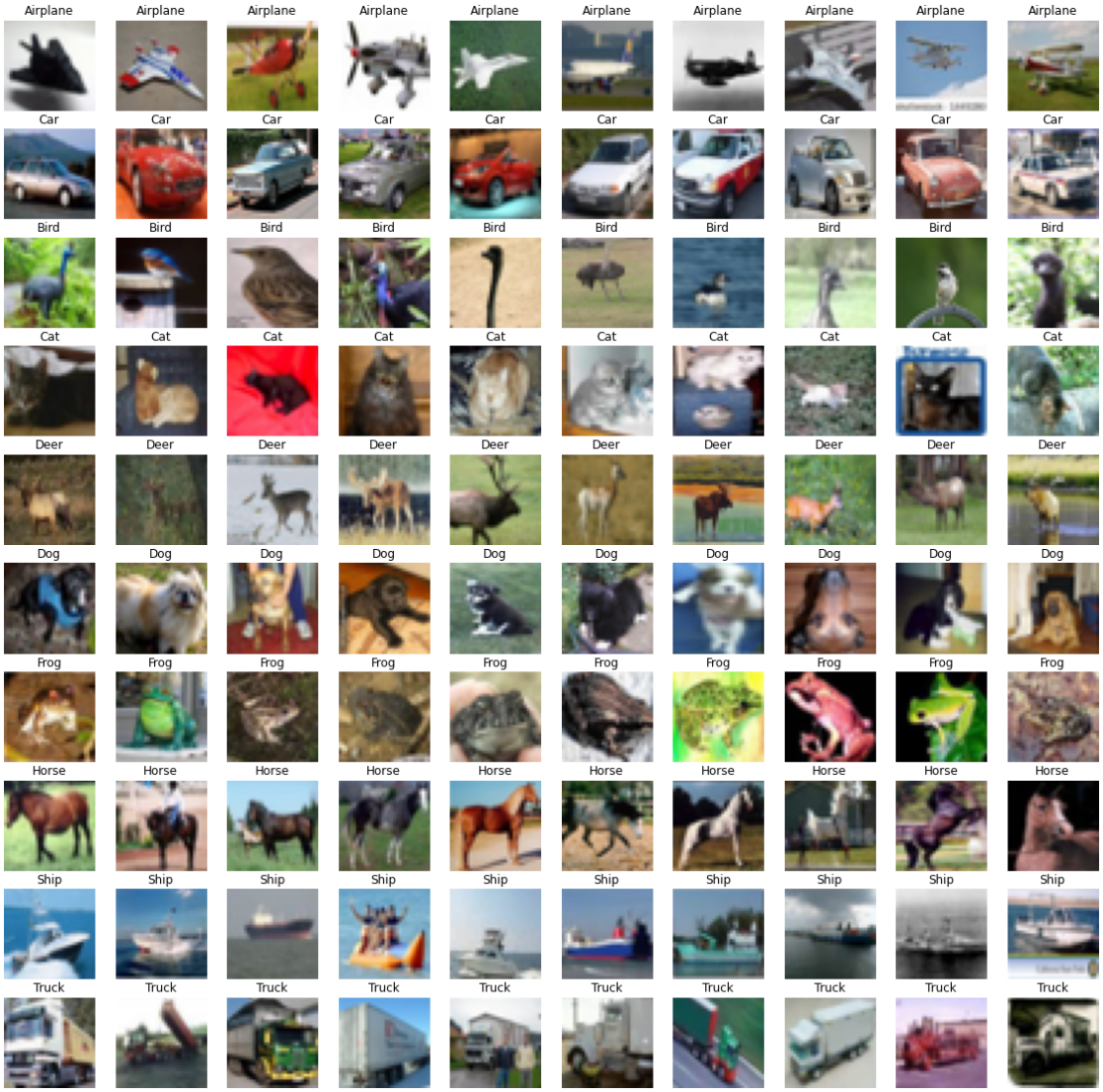
For my experiments, I will only make use of the 50K training data in the training set for the training of the GAN. This is because I want to be able to compare my result with that obtained by others, and the typical experimental setup for CIFAR10 conditional generation is to train only on the training set. If I train on both the training and

testing data, the metrics obtained (e.g. KID, FID) will be biased as the metrics are calculated in comparison to the CIFAR10 testing data.

Exploratory Data Analysis

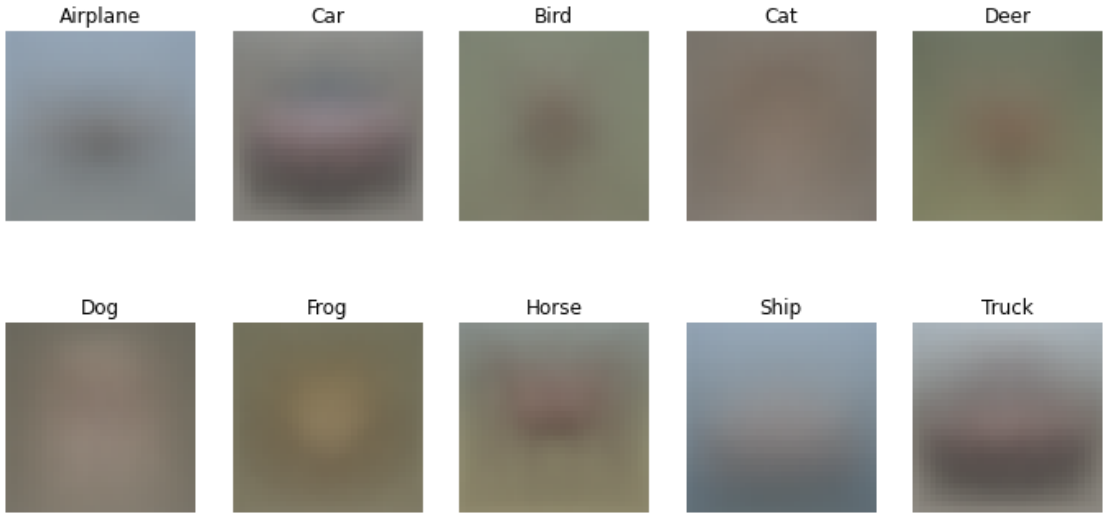
> 💡 The full EDA has been done in an EDA Notebook, which can be found in the repository. All EDA is done on the training set only. In this report, I leave out parts of the EDA notebook which have already been covered prior (e.g dataset size, resolution of each image) to avoid redundancy.

I begin by visualizing 10 images of each class to have a view of what the dataset looks like:



10 Images Per Class

As can be seen, I note that the sources of the images appear to be quite diverse. For example, I can clearly see toy planes alongside actual planes, and some images with text on them. This could make some of these images difficult to generate.



This can be shown by a plot of the "average" image per class, where I can observe that some classes have very blurry images, suggesting that images are more diverse (if images are very similar, the average image should be more clearly shown), with cars being potentially the least diverse class.

I also calculated certain GAN evaluation metrics on the training and validation dataset to get a "ground truth" score, which lets me tell what is the best possible result I might hope for.

Metric	Value
Inception Score	11.24
FID	3.15
KID	-1.49

Data Pre-processing and Augmentation

Data Preprocessing

A common preprocessing step when training a GAN is to normalize the dataset images such that they are within the range of -1 to 1. By

performing feature scaling, I help the model to converge faster by gradient descent.

Data Augmentation

Data augmentation is common applied in deep learning as a technique for preventing overfitting of a model by randomly applying transformations to images to generate variations of it. In this case, we want to reduce overfitting in the discriminator, as an overfitting discriminator, as found in Karras et al., "Training Generative Adversarial Networks with Limited Data." can lead to reduce quality of the generated images.

> The distributions overlap initially but keep drifting apart as the discriminator becomes more and more confident, and the point where FID starts to deteriorate is consistent with the loss of sufficient overlap between distributions. This is a strong indication of overfitting, evidenced further by the drop in accuracy measured for a separate validation set. - Section 2, Overfitting in GANs

When performing data augmentation in GAN training, there are some things to take note of:

- Some augmentations are "leaky", meaning that the generator may learn to generate augmented images as it is unable to separate the augmentation from the data distribution.

As such, as a start, I will avoid the use of data augmentation, but will experiment with it later on.

Hyperparameters

The most important hyperparameter for the data loader is the batch size, which defines how many real images (and thus how many generated fake images) will be fed into the discriminator during training of either the generator or discriminator.

It was found in Brock, Donahue, and Simonyan, "Large Scale GAN Training for High Fidelity Natural Image Synthesis." that larger batch sizes could improve the overall quality of the GAN output, but would lead to more unstable training.

> We begin by increasing the batch size for the baseline model, and immediately find tremendous benefits in doing so. Rows 1-4 of Table 1 show that **simply increasing the batch size by a factor of 8 improves the state-of-the-art IS by 46%**. We conjecture that this is a result of **each batch covering more modes**, providing **better gradients** for both networks. One notable side effect of this scaling is that **our models reach better final performance in fewer iterations, but become unstable and undergo complete training collapse.**

As a result of their findings, I will try and use as large a batch size as possible, within the limitations of the GPU memory.

```
# @title Basic Hyperparameters { run: "auto" }
DATA_DIR = "./data"  # @param {type:"string"}
BATCH_SIZE = 256  # @param {type:"integer"}
NUM_WORKERS = 2  # @param {type:"integer"}
```

💡 `NUM_WORKERS`: The number of processes to run in parallel to load in the data. This hyperparameter is limited by the number of CPU cores available
`DATA_DIR`: The directory to download the CIFAR dataset to

Source Code

```
preprocessing = [T.ToTensor(), T.Normalize((0.5, 0.5, 0.5), (0.5, 0.
basic_aug = [T.RandomHorizontalFlip(p=0.3)]

dm = CIFAR10DataModule(
    data_dir=DATA_DIR,
    batch_size=BATCH_SIZE,
    num_workers=NUM_WORKERS,
    transforms=preprocessing,
    augments=basic_aug,
)
```

💡 The CIFAR10 dataset in `torchvision` is already in the range of 0-1, so by specifying a mean and standard deviation of 0.5, I am subtracting 0.5 from the mean (which is approximately 0.5), such that the mean is 0 (so a

range of -0.5 to 0.5), and dividing by 0.5 (dividing a number by half is the same as multiplying it by two, so the range becomes -1.0 to 1.0).

Experiments

Metrics

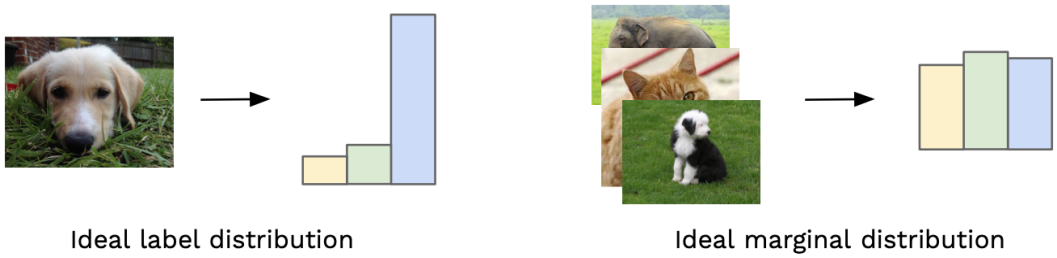
It can be difficult to evaluate a GAN, given that the quality of an image can be highly subjective. However, there are some metrics which allow us to evaluate how similar a generated image is to a real image. Of these metrics, three stand out as being the most popular, being used in most GAN papers in recent years.

Inception Score

The Inception Score (IS) makes use of the Inception V3 image classification model to assess the quality of the GAN output. The intuition behind the Inception Score is as follows. When I feed images into the network, the output will be a softmax probability distribution. If the GAN generated image (e.g. of a dog) is very distinctly of a single class, the softmax distribution should be less uniform. On the other hand, if the image generated is not very clear, the probability distribution is expected to be uniform. So, the class probability distribution can provide information on the quality of the image (so long as the image class is one of the 1000 classes inside the ImageNet dataset).

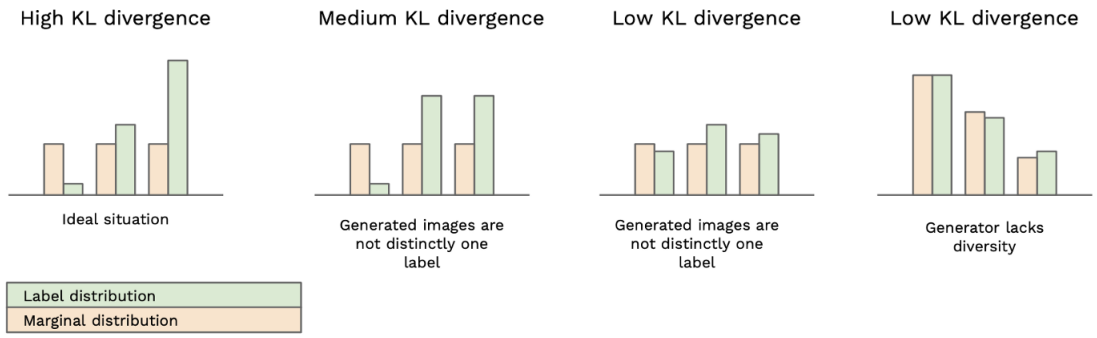
Now, if I added up the class probability distribution for a large number of GAN generated images of various classes, I get something called the marginal probability distribution. If all the images generated were diverse, I would expect that the marginal probability distribution would be fairly uniform, as a less uniform distribution would suggest overall low diversity in the output of the GAN.

So, the ideal individual label distribution for each image is sharp, while the ideal marginal distribution should be more uniform.



Credit: Mack, "A Simple Explanation of the Inception Score."

What the Inception Score measures then is the difference between these two distributions via a statistical distance called Kullback–Leibler divergence. In the optimal case, a high KL divergence would mean that the label and marginal distributions are highly dissimilar, which would be when the images generated are both high in quality and diversity.



Credit: Mack, "A Simple Explanation of the Inception Score."

As found in my earlier EDA (see section above), the ideal IS score for CIFAR10 is around 11.24 (the IS score as calculated on the CIFAR10 validation set).

Fréchet Inception Distance

$$d^2 = ||\mu_1 - \mu_2||^2 + Tr(C_1 + C_2 - 2\sqrt{C_1 * C_2})$$

One flaw with the Inception Score is that it only looks at the distribution of fake images, and does not compare it with the statistics from the actual CIFAR10 dataset to see how similar they are to the real images. To resolve this, the Frechet Inception Distance (FID) was introduced in Heusel et al., "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium." , which instead of using the final output of the Inception V3 model, instead uses the activations of the final pooling layer of the model, which are

then summarized by calculating the mean and covariance. These statistics are collected for the distribution for real and fake images respectively, and then the Fréchet distance (see formula above) between the distributions is calculated.

If this distance is very low, then it suggests that the fake images are highly similar to the real data and thus of high quality. In addition, in order to get a low Fréchet distance, the fake images need to be as similarly diverse as the real dataset.

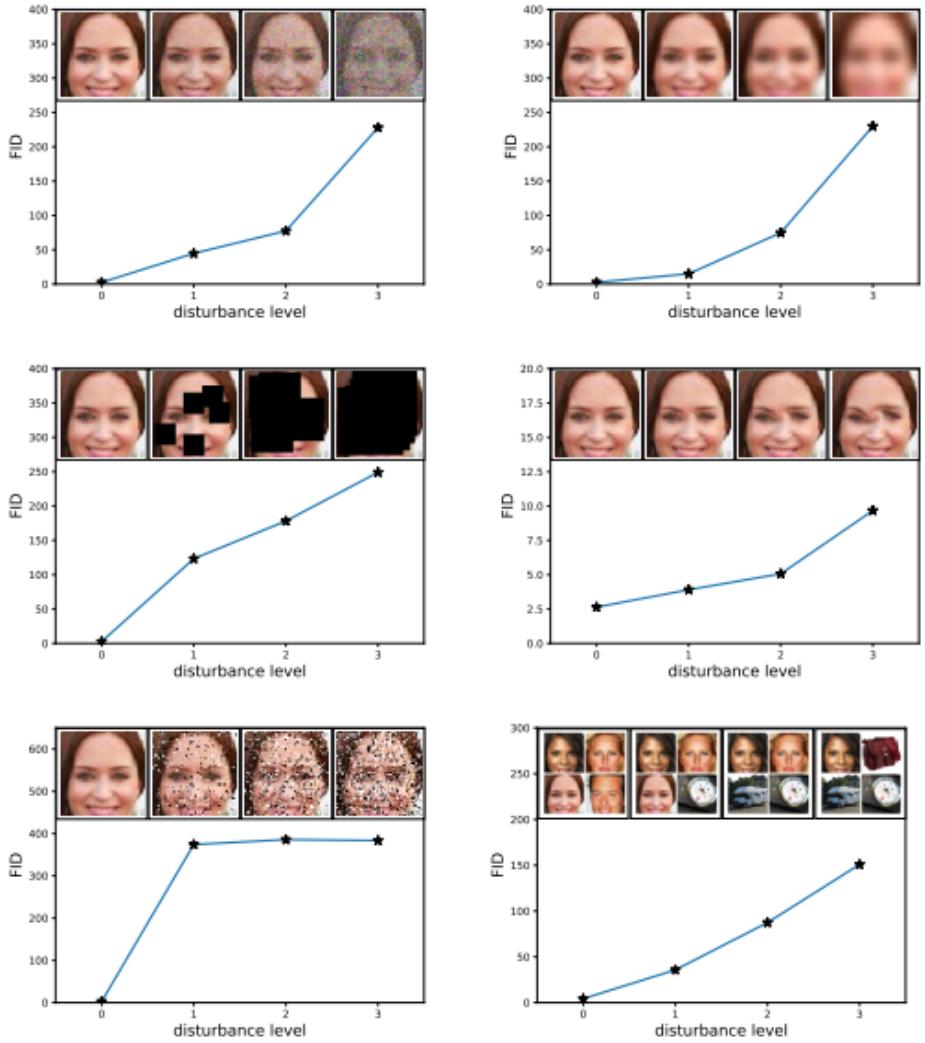


Figure 3: FID is evaluated for Gaussian noise (upper left), Gaussian blur (upper right), implanted black rectangles (middle left), swirled images (middle right), salt and pepper noise (lower left), and CelebA dataset contaminated by ImageNet images (lower right). Left is the smallest disturbance level of zero, which increases to the highest level at right. The FID captures the disturbance level very well by monotonically increasing.

FID score correlates well with image quality (Credit: Heusel et al., "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash

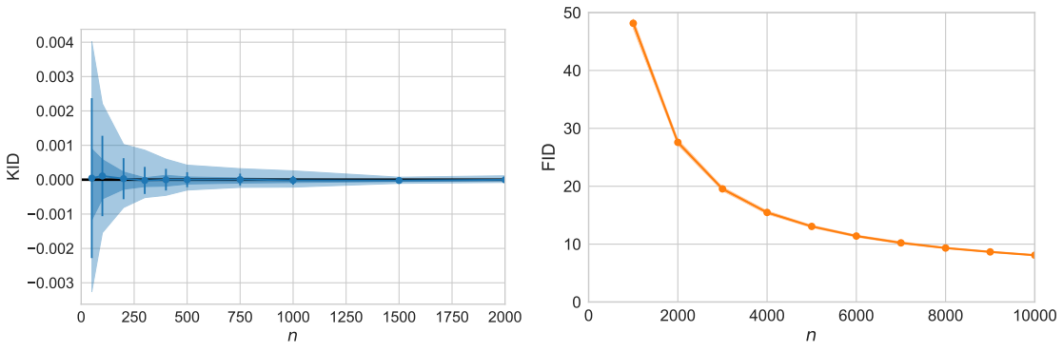
Equilibrium.")

As found in my earlier EDA (see section above), the ideal FID score for CIFAR10 is around 3.15 (the FID between the CIFAR10 training and validation set).

Kernel Inception Distance

While FID is commonly used (and thus is included as it let's me compare the results of my GAN with other papers), it is a biased metric for datasets like CIFAR with a low amount of data (FID assumes a multivariate Gaussian distribution of the activations, and can only be positive)

To resolve this, the Kernel Inception Distance metric is used, which measures the dissimilarity (Maximum Mean Discrepancy) between the fake and real image distribution, similarly to FID. Unlike FID however, KID estimates are less biased even for small sample sizes, allowing me to be more confident in the KID scores reported.



(a) KID estimates are unbiased, and standard deviations shrink quickly even for small n.

(b) FID estimates exhibit strong bias for n even up to 10 000. All standard deviations are less than 0.5.

Estimates of distances between the CIFAR-10 train and test sets. The fact that FID score highly depends on the sample size means that FID scores can be highly misleading at times. (Credit: Demystifying MMD GANs)

As such, the primary metric I will use for deciding between different GANs will be the Kernel Inception Distance, with the FID and IS serving mainly as a point of comparison with results obtained by others (KID is a relatively new metric, so most GAN architectures still report their results using FID and IS)

As found in my earlier EDA (see section above), the ideal KID score for CIFAR10 is around -1.49 (the KID between the CIFAR10 training

and validation set).

> 💡 For KID, the smaller the value, the better, and the value of KID can even be negative.

Source Code

To implement these metrics, I make use of the **Torch Fidelity** library, which provides precise (it has been shown that the scores produced by Torch Fidelity match the original Tensorflow implementations of these metrics to a high precision) and efficient (Torch Fidelity caches the statistics of the dataset, and uses a single Inception network for multiple scores, resulting in a massive speedup compared to the original implementation) implementation of these metrics.

```
metrics = torch_fidelity.calculate_metrics(
        input1=torch_fidelity.GenerativeModelModuleWrapper(
            self.G, self.latent_dim, "normal", self.num_classes
        ),
        input1_model_num_samples=10000,
        input2="cifar10-val",
        isc=True,
        fid=True,
        kid=True
)
self.log("KID", metrics["kernel_inception_distance_mean"], prog_bar=
self.log("FID", metrics["frechet_inception_distance"], prog_bar=True
self.log("IS", metrics["inception_score_mean"], prog_bar=True)
```

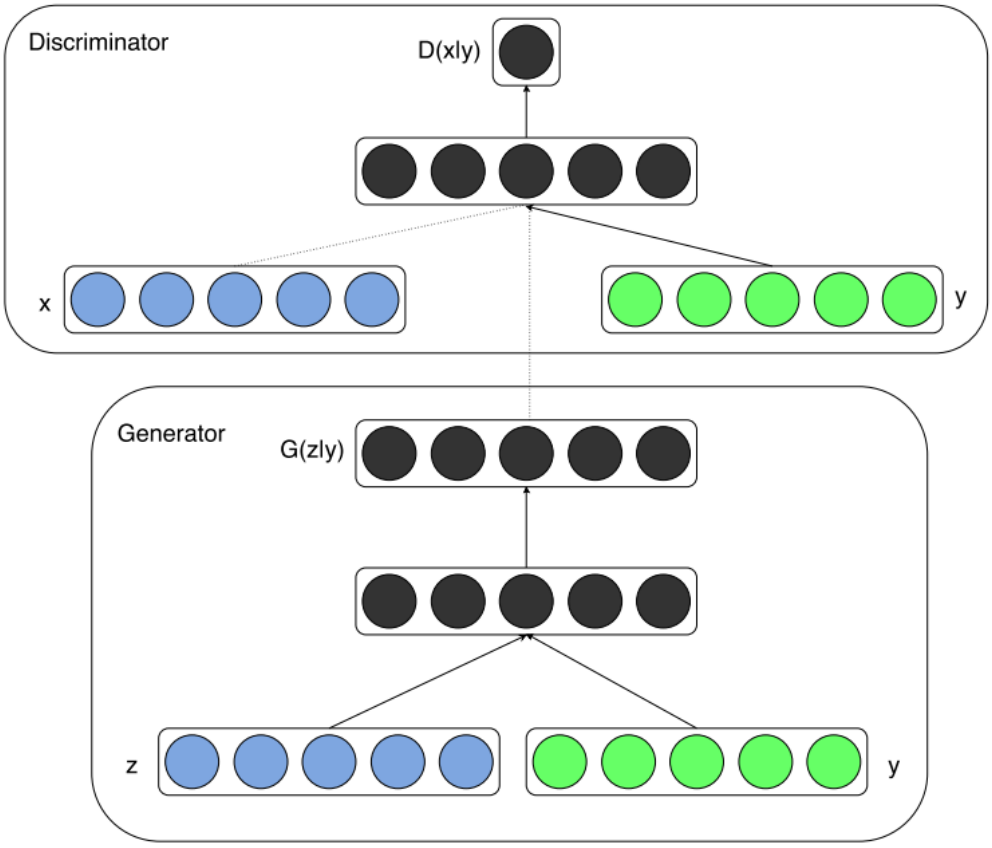
In this implementation of the FID calculation, the score is calculated for 10K fake samples, and the statistics pre-calculated from the CIFAR10 validation set (10K real images)

> 💡 The original paper that introduced the FID score actually recommends 50K samples (meaning to calculate FID based on the CIFAR10 training statistics), but since most papers use 10K samples, and FID depends

highly on the sample size, I have to follow the use of 10K samples I want to be able to directly compare my results with that achieved by others.

Building a Basic Conditional DCGAN

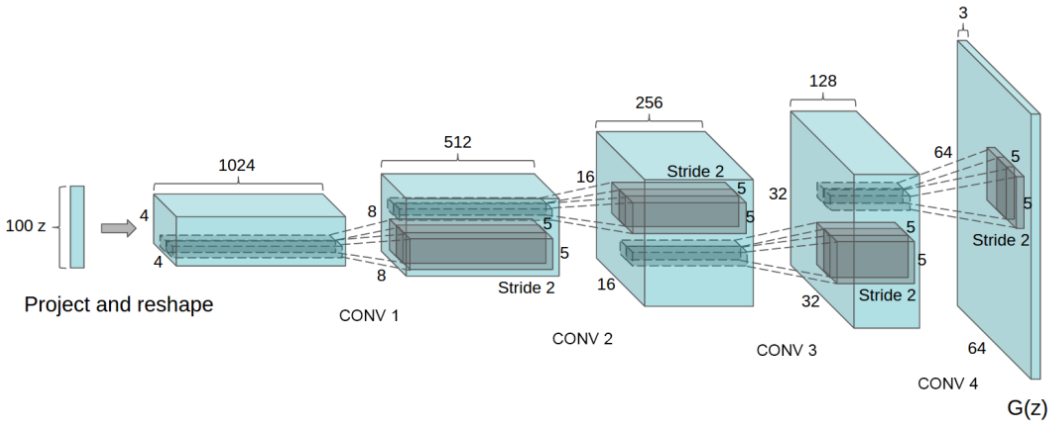
To accomplish my task, I will attempt to construct a Conditional GAN. A conditional GAN is a GAN which is able to perform conditional generation, by embedding the class information into the latent vector $\vec{z}$ that is fed into the generator model, while training the discriminator to classify an image which has also had the class information embedded onto it.



Structure of a Simple Conditional GAN (Credit: Radford, Metz, and Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.")

The specific generator and discriminator architecture used will be taken from Radford, Metz, and Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.", which introduced the Deep Convolutional

Generative Adversarial Network (DCGAN), one of the first CNN architectures to be successfully employed for image generation.



DCGAN Architecture (note that the dimensions here are different, as the visualization is taken for a another dataset)

It employed the following innovations:

- Instead of using fixed pooling operations like Max Pooling in the convolutional blocks, strided convolutions are used, where the convolutional layers can learn their own downsampling operations. Transposed convolutions (also known as fractionally strided convolutions) are used for upsampling.
- Batch normalization which normalizes the inputs to layers to have zero mean and variance of one helps to stabilize learning in deep convolutional layers. However, it was found that applying batch norm to the output layer of the generator and input to the discriminator caused instability in training, so batch norm is omitted for these layers
- Use of ReLU activation in the generator for all layers except the output, which uses the Tanh activation (which restricts the image output to a range of -1 to 1)
- LeakyReLU activation in the discriminator.

While this paper introduced the architecture for unconditional generation, I will adapt it to work for conditional image generation.

Generator

For the generator, I make the following modifications:

- Embed the class information to the input via concatenating the embeddings for the class

The implementation is modified from the DCGAN Tutorial on PyTorch Documentation, modified for conditional generation.

```
class Generator(nn.Module):
    def __init__(
        self,
        latent_dim: int = 128,
        embed_dim: int = 256,
        num_filters: int = 64,
        num_classes: int = 10,
    ):
        super().__init__()
        self.latent_dim = latent_dim
        self.num_filters = num_filters  # Base Number of Filters in
        self.num_classes = num_classes
        self.embed_dim = embed_dim
        self.label_embedding = nn.Embedding(num_classes, embed_dim)
        self.latent = nn.Linear(latent_dim + embed_dim, latent_dim)
        self.main = nn.Sequential(
            nn.ConvTranspose2d(latent_dim, num_filters * 4, 4, 1, 0,
            nn.BatchNorm2d(num_filters * 4),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(num_filters * 4, num_filters * 2, 4,
            nn.BatchNorm2d(num_filters * 2),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d(num_filters * 2, num_filters, 4, 2, 1
            nn.BatchNorm2d(num_filters),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d(num_filters, 3, 4, 2, 1, bias=False),
            nn.Tanh(),
        )

    def forward(self, x: torch.Tensor, y: list = None):
        y_embed = self.label_embedding(y)
        y_embed = y_embed / torch.norm(y_embed, p=2, dim=1, keepdim=
```

```
        conditional_inputs = torch.cat([x, y_embed], dim=1) # Concat
        conditional_inputs = self.latent(conditional_inputs)
        conditional_inputs = conditional_inputs.view(
            conditional_inputs.shape[0], self.latent_dim, 1, 1
        )
        fake = self.main(conditional_inputs)
        if not self.training:
            fake = 255 * (fake.clamp(-1, 1) * 0.5 + 0.5)
            fake = fake.to(torch.uint8)
        return fake
```

There are several ways to incorporate image information in a conditional GAN. The most common way is to concatenate the embedding into the latent vector, but another approach is to multiply the latent vector by the embedding. Through prior experimentation with both methods, I have found that both methods work well, but I will stick with the more traditional concatenation method here.

Discriminator

For the discriminator, I make the following modifications:

- Add a dropout layer to reduce overfitting
- Embed the class information to the input

```
class Discriminator(nn.Module):
    def __init__(self, num_filters: int = 64, num_classes: int = 10)
        super().__init__()
        self.num_filters = num_filters
        self.num_classes = num_classes
        self.label_embedding = nn.Embedding(num_classes, 32 * 32)
        self.main = nn.Sequential(
            nn.Conv2d(4, num_filters, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(num_filters, num_filters * 2, 4, 2, 1, bias=Fa
            nn.BatchNorm2d(num_filters * 2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(num_filters * 2, num_filters * 4, 4, 2, 1, bia
```

```
            nn.BatchNorm2d(num_filters * 4),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout2d(0.2),
            nn.Conv2d(
                num_filters * 4, 1, 4, 1, 0, bias=False
            ),  # Apply a 4x4 Convolution to a 4x4 input, resulting
            nn.Sigmoid(),
        )

    def forward(self, x: torch.Tensor, y: list = None):
        labels = self.label_embedding(y)
        labels = labels.view(labels.shape[0], 1, 32, 32)
        conditional_inputs = torch.cat([x, labels], dim = 1)  # Conc
        return self.main(conditional_inputs)
```

The implementation is modified from the DCGAN Tutorial on PyTorch Documentation, modified for conditional generation.

Loss

The loss function used will be the binary cross entropy loss function, defined as follows:

$$\min_d \max_g -[\mathbb{E}(\log(D(x)) + \mathbb{E}(1 - \log(D(G(z)))]$$

where $\mathbb{E}$ represents the expected value (that is, mean or average value) of the binary cross entropy loss.

In short, what this loss represents is that we want to penalize the discriminator if it decides to classify the real images as real and the fake images as fake, and penalize the generator if the discriminator does not classify its images as real (i.e. a value of 1 from the discriminator).

Luckily, in PyTorch, this loss is easily implemented using `nn.BCELoss()`.

Putting Things Together

```
class ConditionalDCGAN(LightningModule):
```

```python
    def __init__(
        self,
        latent_dim: int = 128,
        embed_dim: int = 128,
        num_classes: int = 10,
        g_lr: float = 0.0002,
        d_lr: float = 0.0002,
        adam_betas: Tuple[float, float] = (0.0, 0.999),
        batch_size: int = 64,
        d_steps: int = 1,
        **kwargs,
    ):
        super().__init__()

        self.latent_dim = latent_dim
        self.embed_dim = embed_dim
        self.num_classes = num_classes
        self.g_lr = g_lr
        self.d_lr = d_lr
        self.betas = adam_betas
        self.batch_size = batch_size
        self.d_steps = d_steps  # Number of Discriminator steps per
        self.save_hyperparameters()

        self.G = Generator(
            latent_dim=self.latent_dim,
            embed_dim=self.embed_dim,
            num_classes=num_classes,
        )

        self.D = Discriminator(num_classes=num_classes)

        self.G.apply(self._weights_init)
        self.D.apply(self._weights_init)

        self.adversarial_loss = nn.BCELoss()  # BCE Loss

        self.viz_z = torch.randn(64, self.latent_dim)
        self.viz_labels = torch.LongTensor(torch.randint(0, self.nu

        self.unnormalize = NormalizeInverse((0.5, 0.5, 0.5), (0.5, 0
```

```
    @staticmethod
    def _weights_init(m):
        classname = m.__class__.__name__
        if classname.find("Conv") != -1:
            torch.nn.init.normal_(m.weight, 0.0, 0.02)
        elif classname.find("BatchNorm") != -1:
            torch.nn.init.normal_(m.weight, 1.0, 0.02)
            torch.nn.init.zeros_(m.bias)

    def forward(self, z, labels):
        return self.G(z, labels)

    # Alternating schedule for optimizer steps (i.e.: GANs)
    def optimizer_step(
        self,
        epoch,
        batch_idx,
        optimizer,
        optimizer_idx,
        optimizer_closure,
        on_tpu,
        using_native_amp,
        using_lbfgs,
    ):
        # update discriminator opt every step
        if optimizer_idx == 1:
            optimizer.step(closure=optimizer_closure)

        # update generator opt every 4 steps
        if optimizer_idx == 0:
            if (batch_idx + 1) % self.d_steps == 0:
                optimizer.step(closure=optimizer_closure)
            else:
                # call the closure by itself to run `training_step`
                optimizer_closure()

    def training_step(self, batch, batch_idx, optimizer_idx):
        imgs, labels = batch  # Get real images and corresponding la

        # Generate Noise Vector z
```

```
        z = torch.randn(imgs.shape[0], self.latent_dim)
        z = z.type_as(imgs)  # Ensure z runs on the same device as t
        self.fake_labels = torch.LongTensor(
            torch.randint(0, self.num_classes, (imgs.shape[0],))
        ).to(self.device)
        # Train Generator
        if optimizer_idx == 0:
            # Generate Images
            self.fakes = self.forward(z, self.fake_labels)

            # Classify Generated Images with Discriminator
            fake_preds = torch.squeeze(self.D(self.fakes, self.fake_

            # We want to penalize the Generator if the Discriminator
            # Hence, set the target as a 1's vector
            target = torch.ones(imgs.shape[0]).type_as(imgs)

            g_loss = self.adversarial_loss(fake_preds, target)

            self.log(
                "train_gen_loss",
                g_loss,
                on_epoch=True,
                on_step=False,
                prog_bar=True,
            )  # Log Generator Loss
            tqdm_dict = {
                "g_loss": g_loss,
            }
            output = OrderedDict(
                {"loss": g_loss, "progress_bar": tqdm_dict, "log": t
            )
            return output

        # Train Discriminator
        if optimizer_idx == 1:
            # Train on Real Data
            real_preds = torch.squeeze(self.D(imgs, labels))
            target = torch.ones(imgs.shape[0]).type_as(imgs)
            d_real_loss = self.adversarial_loss(real_preds, target)
```

```
            # Train on Generated Images
            self.fakes = self.forward(z, self.fake_labels)
            target = torch.zeros(imgs.shape[0]).type_as(imgs)
            fake_preds = torch.squeeze(self.D(self.fakes, self.fake_
            d_fake_loss = self.adversarial_loss(fake_preds, target)
            d_loss = (d_real_loss + d_fake_loss) / 2

            self.log(
                "train_discriminator_loss",
                d_loss,
                on_epoch=True,
                on_step=False,
                prog_bar=True,
            )
            tqdm_dict = {
                "d_loss": d_loss,
            }
            output = OrderedDict(
                {"loss": d_loss, "progress_bar": tqdm_dict, "log": t
            )
            return output

    def training_epoch_end(self, outputs):
        # Log Sampled Images
        sample_imgs = self.unnormalize(self.fakes[:64]).cpu().detach
        sample_labels = self.fake_labels[:64].cpu().detach()
        num_rows = int(np.floor(np.sqrt(len(sample_imgs))))
        fig = visualize(sample_imgs, sample_labels, grid_shape=(num_
        self.logger.log_image(key="generated_images", images=[fig])
        plt.close(fig)
        del sample_imgs
        del sample_labels
        del fig
        gc.collect()

    def validation_step(self, batch, batch_idx):
        pass

    def validation_epoch_end(self, outputs):
        sample_imgs = (
            self.forward(self.viz_z.to(self.device), self.viz_labels
```

```
            .cpu()
            .detach()
        )
        fig = visualize(sample_imgs, self.viz_labels.cpu().detach(),
        metrics = torch_fidelity.calculate_metrics(
            input1=torch_fidelity.GenerativeModelModuleWrapper(
                self.G, self.latent_dim, "normal", self.num_classes
            ),
            input1_model_num_samples=10000,
            input2="cifar10-val",
            isc=True,
            fid=True,
            kid=True,
        )
        self.logger.log_image(key="Validation Images", images=[fig])
        plt.close(fig)
        del sample_imgs
        del fig
        gc.collect()
        self.log("FID", metrics["frechet_inception_distance"], prog_
        self.log("IS", metrics["inception_score_mean"], prog_bar=Tru
        self.log("KID", metrics["kernel_inception_distance_mean"], p

    def configure_optimizers(self):
        """Define the optimizers and schedulers for PyTorch Lightnin

        :return: A tuple of two lists - a list of optimizers and a l
        :rtype: Tuple[List, List]
        """
        opt_G = Adam(
            self.G.parameters(), lr=self.g_lr, betas=self.betas
        )  # optimizer_idx = 0
        opt_D = Adam(
            self.D.parameters(), lr=self.d_lr, betas=self.betas
        )  # optimizer_idx = 1
        return [opt_G, opt_D], []
```

Hyperparameters

For training the GAN, I specify some hyperparameters for the optimization process. The optimizer used is the Adam algorithm.

- Learning Rate: The learning rate for both the generator and discriminator is set as 0.0002
- Betas: The β_1, β_2 decay rate hyperparameters for the Adam optimizer are set as 0 and 0.999 respectively

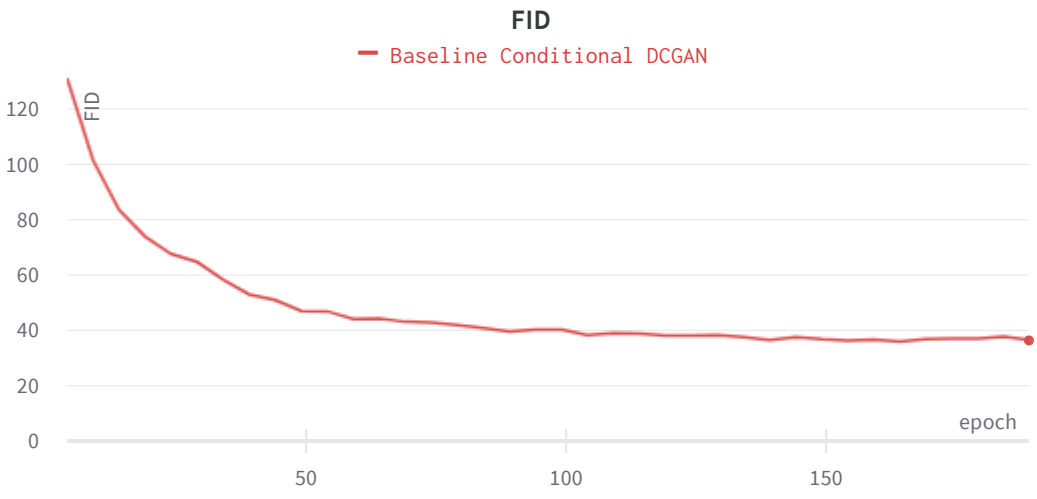
Training

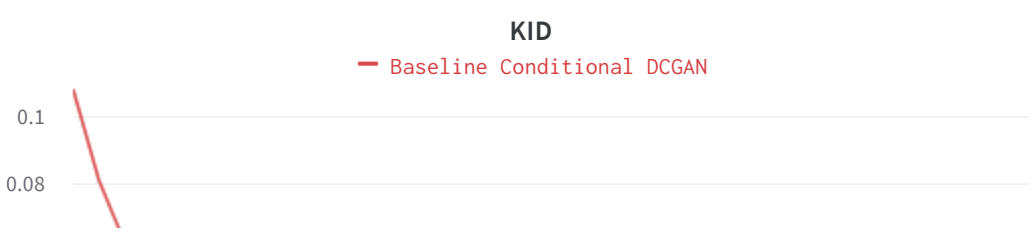
💡 The training notebook for the baseline model is found in `experiments/Baseline Conditional DCGAN.ipynb`

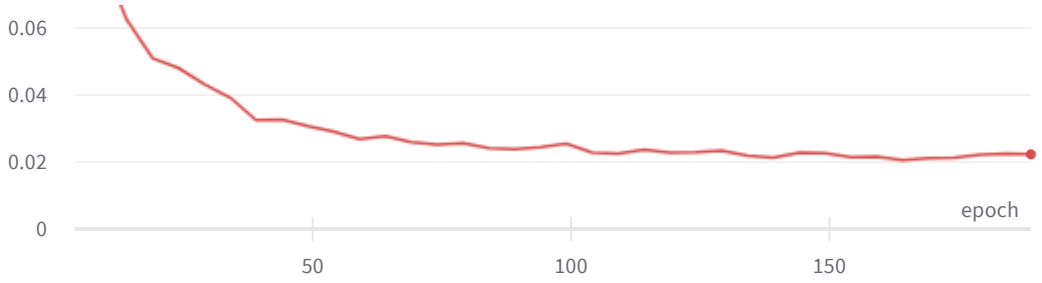
Results

First Run

While the first run had a promising FID score, I observe that the training collapses due to the discriminator getting too strong after around 50 epochs. As a result, I decided to halt the training early on once the FID score begins to stagnate.

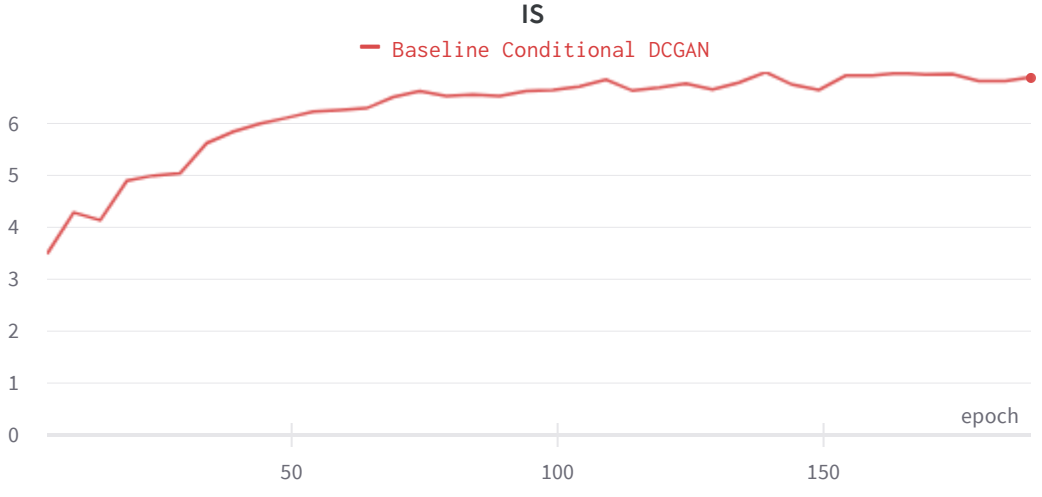


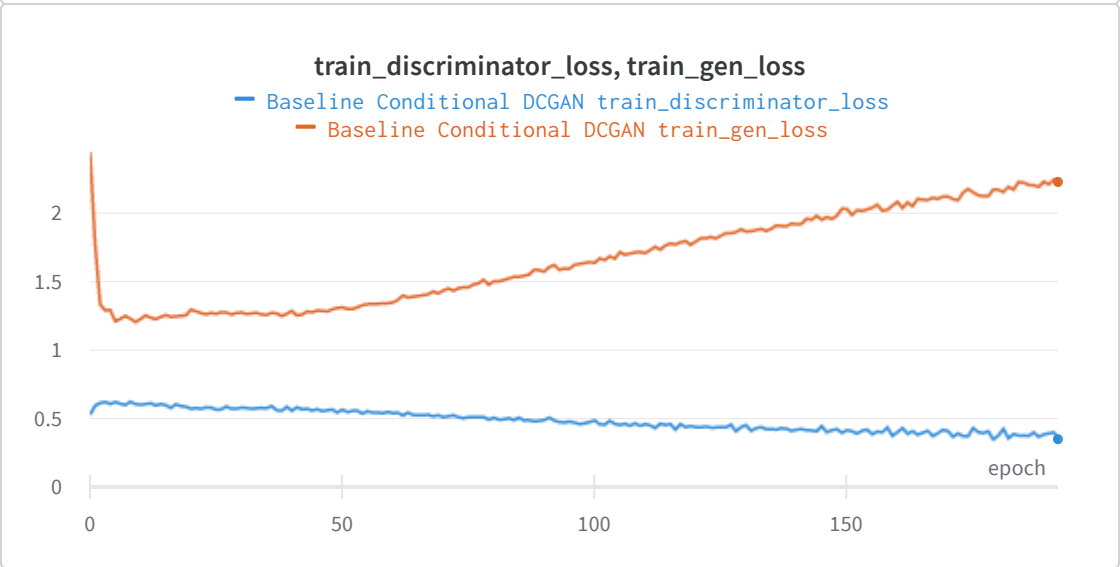


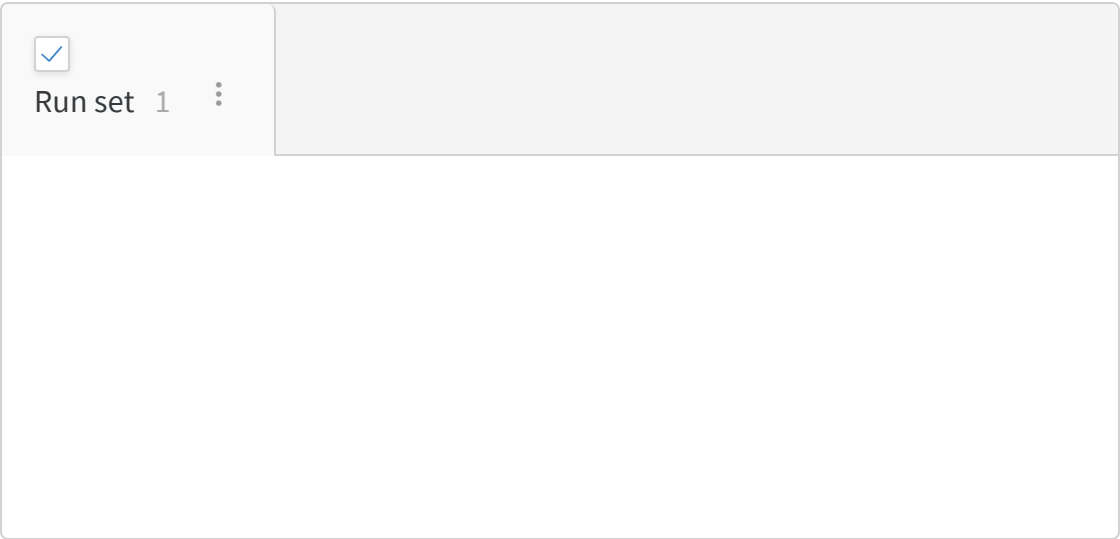


Validation Images

Step 453







For our first run the training eventually collapses, giving a baseline FID of around 36. As can be seen from the losses, there clearly is a need to regularize the discriminator to stop it from becoming too strong.

There are a few avenues of approach here

- I could improve the optimization process through techniques like label smoothing and data augmentation
- I could change up the loss function in hopes of reducing training instability
- I could modify the learning rates

One Sided Label Smoothing

One of the ways to improve the training of the GAN, as recommended by Goodfellow, "NIPS 2016 Tutorial." is to replace the positive classification target of the discriminator when working with real images (a value of 1, corresponding to a real image), with a smoothed value like 0.9. This prevents the discriminator from being too confident in it's predictions, reducing vulnerability to adversarial examples created by the generator.

In this case, I set the label smoothing hyperparameter as `0.1`.

```
real_preds = torch.squeeze(self.D(imgs, labels))
target = torch.ones(imgs.shape[0]).type_as(imgs) - self.label_smooth
d_real_loss = self.adversarial_loss(real_preds, target)
```

> 💡 In this case, Ian Goodfellow recommends only performing label smoothing when training the discriminator, so the ones target when training the generator should not be smoothed.

Adding R1 Regularization

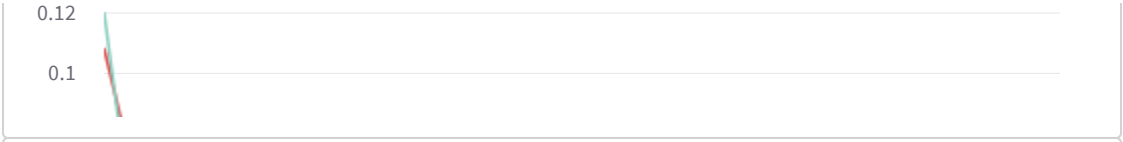
One of the ways to regularize the discriminator is to perform R1 regularization. R1 regularization tries to enforce the Nash Equilibrium (the optimal solution when training a GAN when both the generator and discriminator are no longer able to improve), by penalizing the gradient on real data alone. If the generator performs well (discovering the actual data distribution), we want to ensure that the discriminator does not create a gradient that will cause the generator to create something outside that distribution.

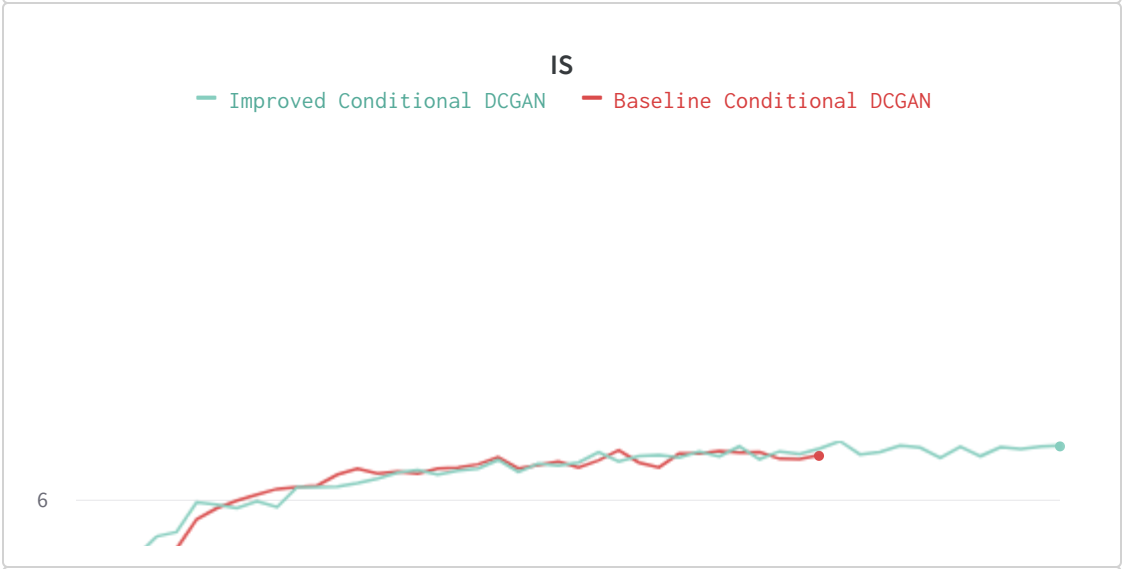
$$R_1(\psi) = \frac{\gamma}{2}\mathbb{E}_{p_{D(x)}}[||\nabla D_\psi(x)||^2]$$

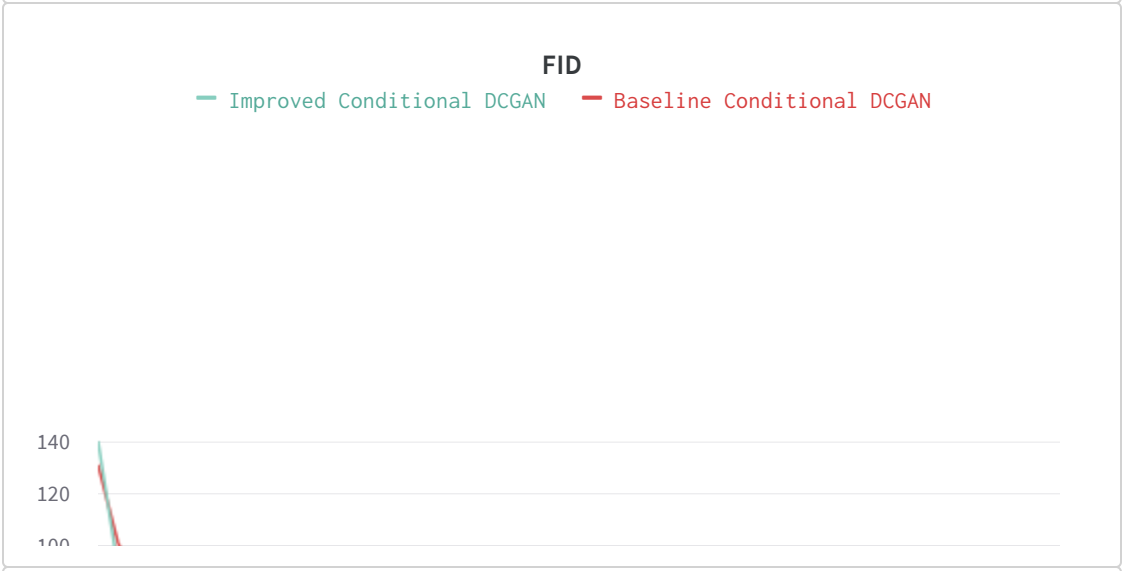
The implementation of the R1 Regularization loss is taken from the original implementation, as implemented in DIRAC GAN with some minor modifications by myself.

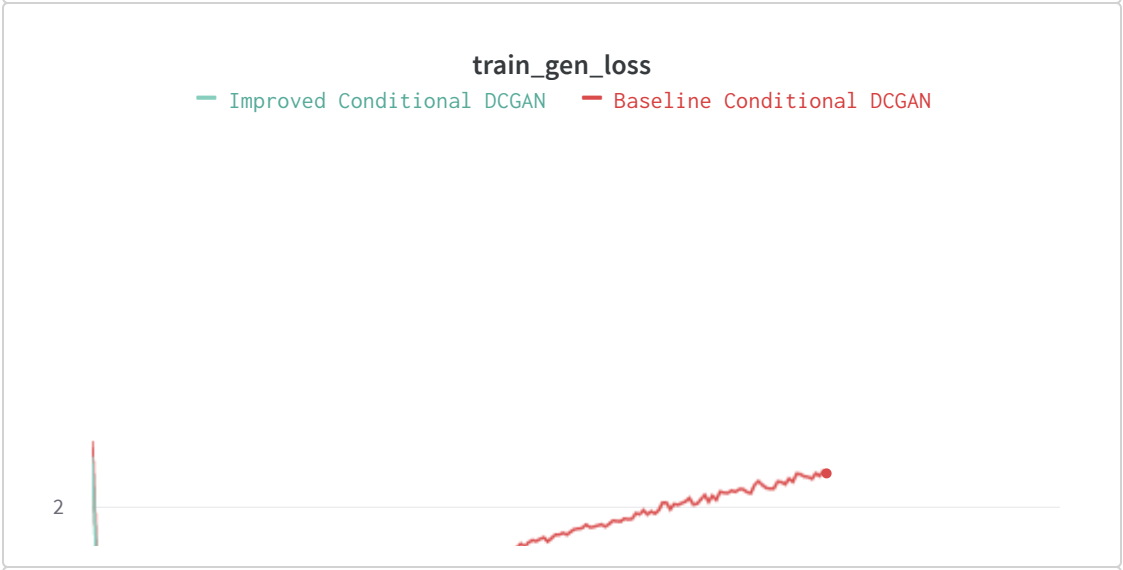
```
class R1(nn.Module):
    """
    Implementation of the R1 GAN regularization, taken from official
```

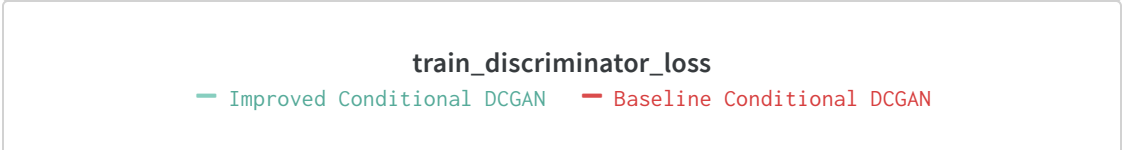
```
    """
    def __init__(self, gamma):
        """
        Constructor method
        """
        # Call super constructor
        super(R1, self).__init__()
        self.gamma = gamma

    def forward(
        self, prediction_real: torch.Tensor, real_sample: torch.Tens
    ) -> torch.Tensor:
        """
        Forward pass to compute the regularization
        :param prediction_real: (torch.Tensor) Prediction of the dis
        :param real_sample: (torch.Tensor) Batch of the correspondir
        :return: (torch.Tensor) Loss value
        """
        # Calc gradient
        grad_real = torch.autograd.grad(
            outputs=prediction_real.sum(),
            inputs=real_sample,
            create_graph=True,
        )[0]
        # Calc regularization
        regularization_loss: torch.Tensor = (
            self.gamma * grad_real.pow(2).view(grad_real.shape[0],
        )
        return regularization_loss
```


0.12
0.1


IS
Improved Conditional DCGAN
Baseline Conditional DCGAN
6


FID
Improved Conditional DCGAN
Baseline Conditional DCGAN
140
120


train_gen_loss
Improved Conditional DCGAN
Baseline Conditional DCGAN
2


train_discriminator_loss
Improved Conditional DCGAN
Baseline Conditional DCGAN

Validation Images

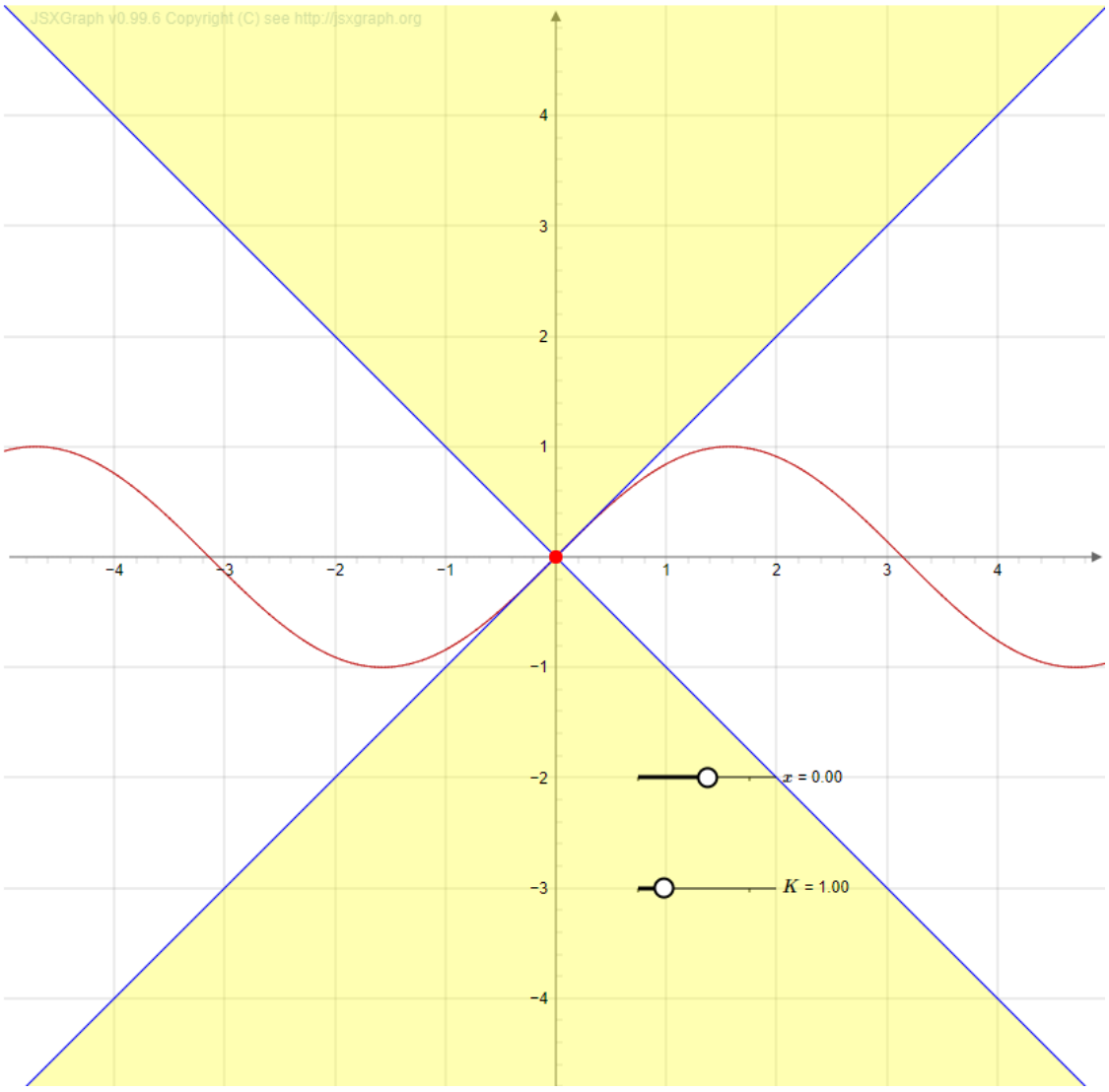
Step 597

Run set 2

As can be seen from the loss curves and improved KID score, the added regularization and label smoothing does help a bit, but the training is still unstable as the discriminator loss and generator loss still diverge after a while.

Spectral Normalization

To further stabilize the training of the discriminator network, I attempt to introduce spectral normalization. Spectral normalization is a weight normalization method that attempts to enforce Lipschitz continuity in the network. Lipschitz continuity means that for a given function, the magnitude of its gradient should be at most K for every point.

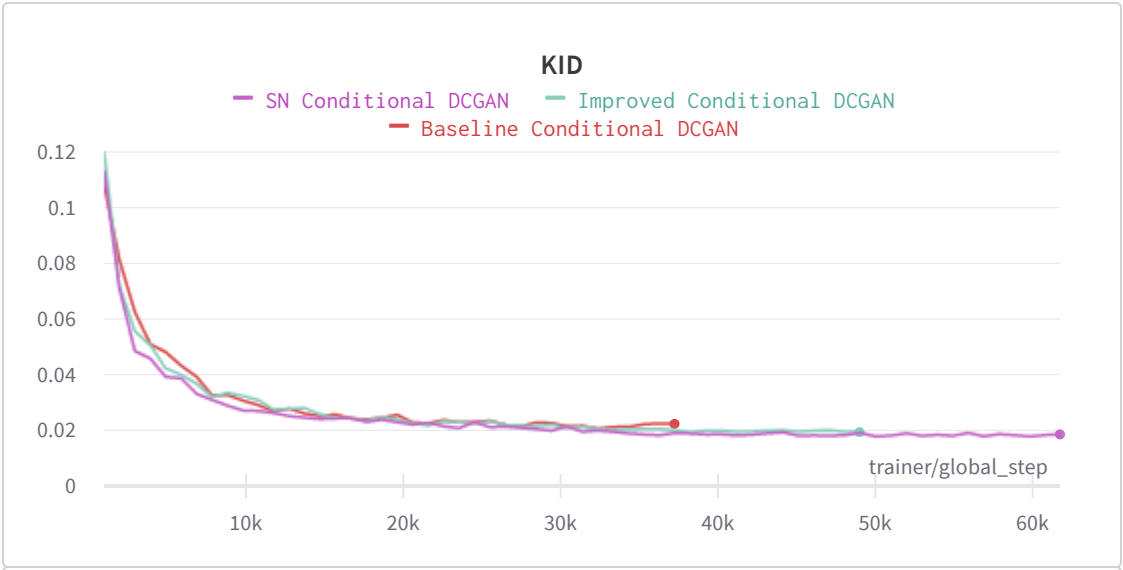


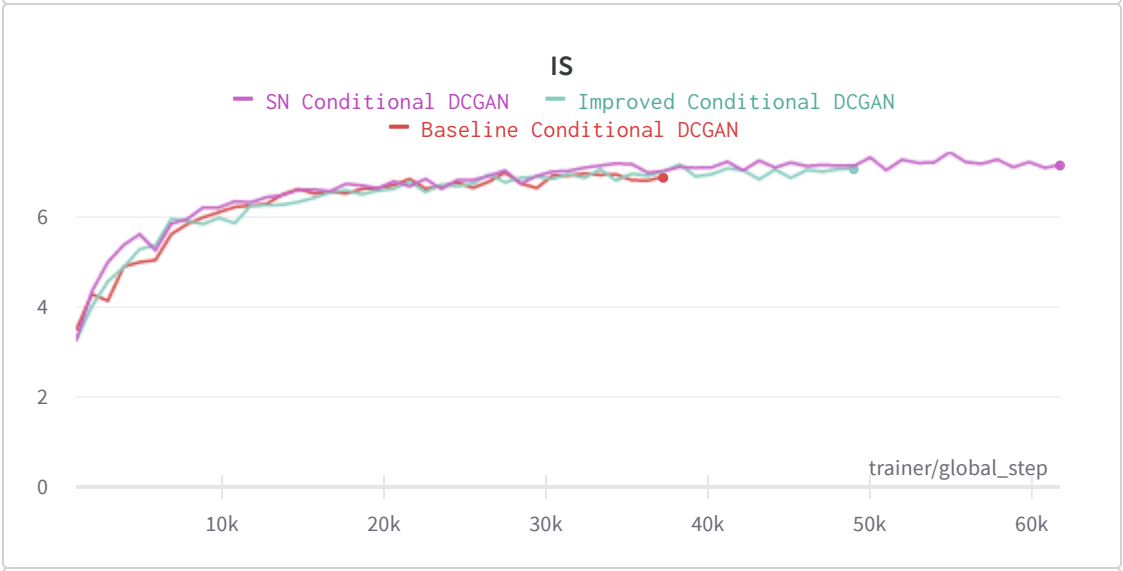
sin(x) is a 1-Lipschitz continuous function - there exists a double cone such that wherever it is placed on the line, the function always remains entirely outside of the cone (Credit: Spectral Normalization Explained)

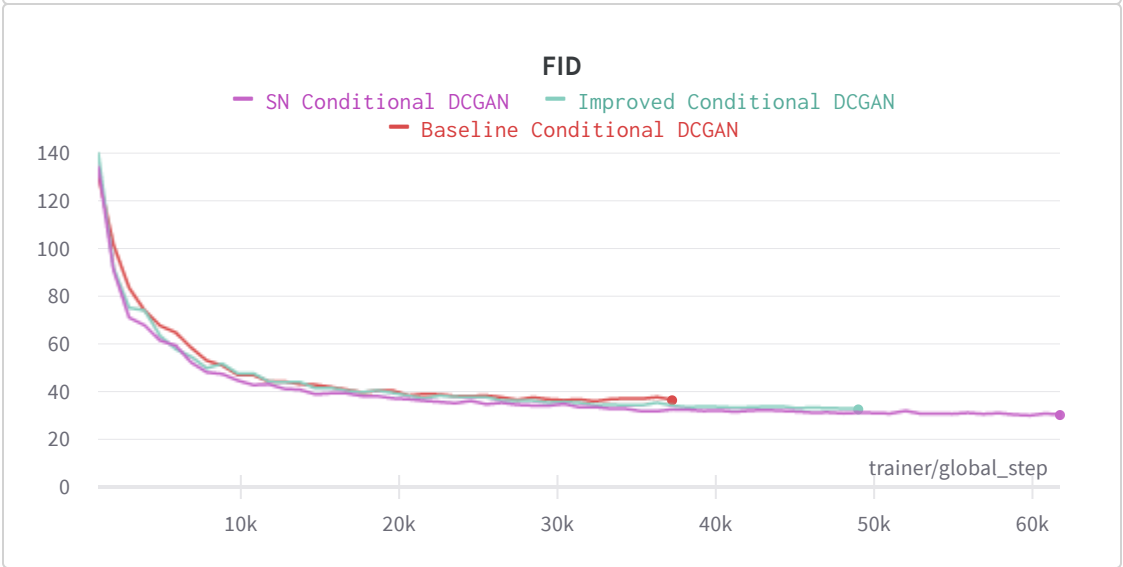
The actual result of the network having Lipschitz continuity is that it constrains the gradients in the network (by enforcing an upper bound to the gradients). As a result spectral normalization reduces the risk of exploding gradients during training. In addition, spectral normalization controls the variance of the weights in the network, helping to prevent the vanishing gradient problem. (Lin, Sekar, and Fanti, "Why Spectral Normalization Stabilizes GANs.")

Luckily, PyTorch already includes an implementation of spectral normalization, making it easy to add it to the discriminator.

```
from torch.nn.utils import spectral_norm
class Discriminator(nn.Module):
    def __init__(self, num_filters: int = 64, num_classes: int = 10)
        super().__init__()
        self.num_filters = num_filters
        self.num_classes = num_classes
        self.label_embedding = nn.Embedding(num_classes, 32 * 32)
        self.main = nn.Sequential(
            spectral_norm(nn.Conv2d(4, num_filters, 4, 2, 1, bias=Fa
            nn.LeakyReLU(0.2, inplace=True),
            spectral_norm(nn.Conv2d(num_filters, num_filters * 2, 4,
            nn.BatchNorm2d(num_filters * 2),
            nn.LeakyReLU(0.2, inplace=True),
            spectral_norm(nn.Conv2d(num_filters * 2, num_filters * 4
            nn.BatchNorm2d(num_filters * 4),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout2d(0.2),
            spectral_norm(nn.Conv2d(
                num_filters * 4, 1, 4, 1, 0, bias=False
            )),  # Apply a 4x4 Convolution to a 4x4 input, resulting
            nn.Sigmoid(),
        )

    def forward(self, x: torch.Tensor, y: list = None):
        labels = self.label_embedding(y)
        labels = labels / torch.norm(labels, p=2, dim=1, keepdim=Tru
        labels = labels.view(labels.shape[0], 1, 32, 32)
        conditional_inputs = torch.cat([x, labels], dim=1)  # Concat
        return self.main(conditional_inputs)
```

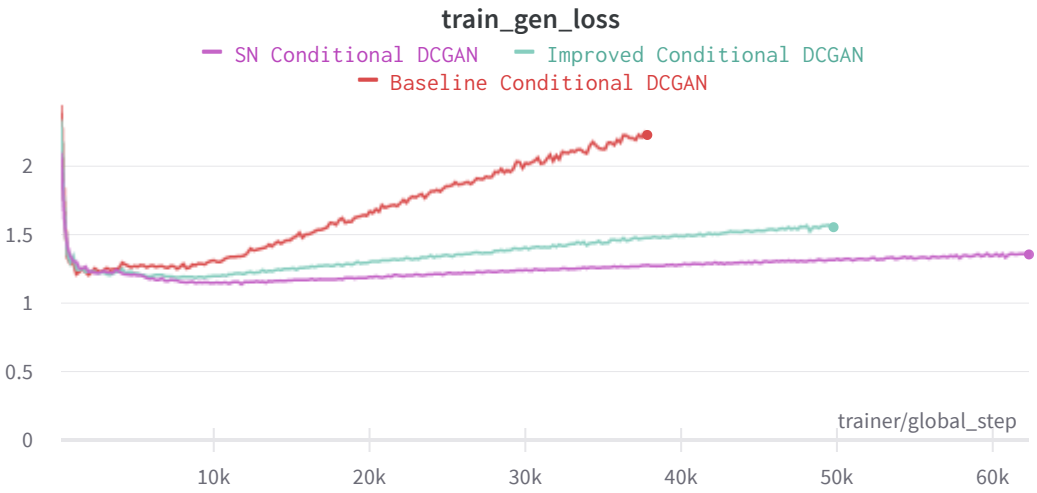


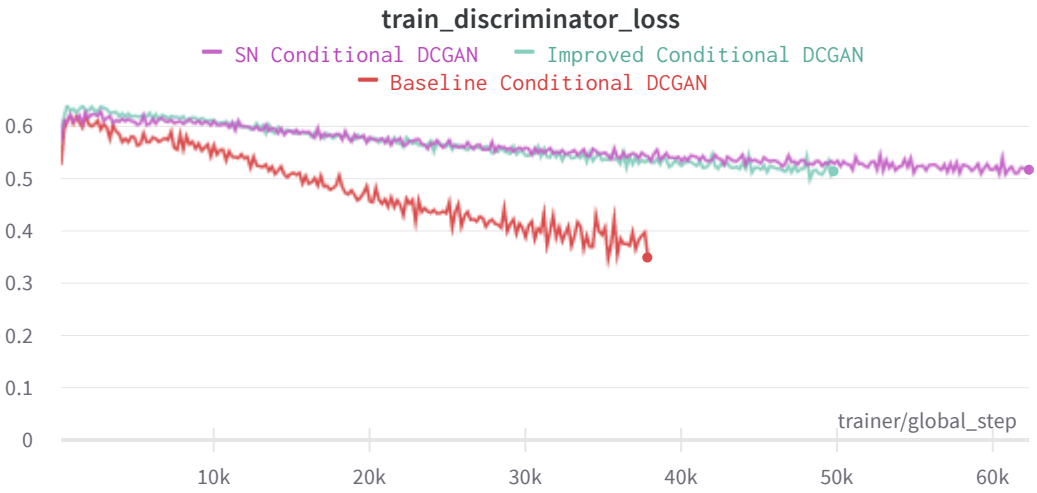




Validation Images

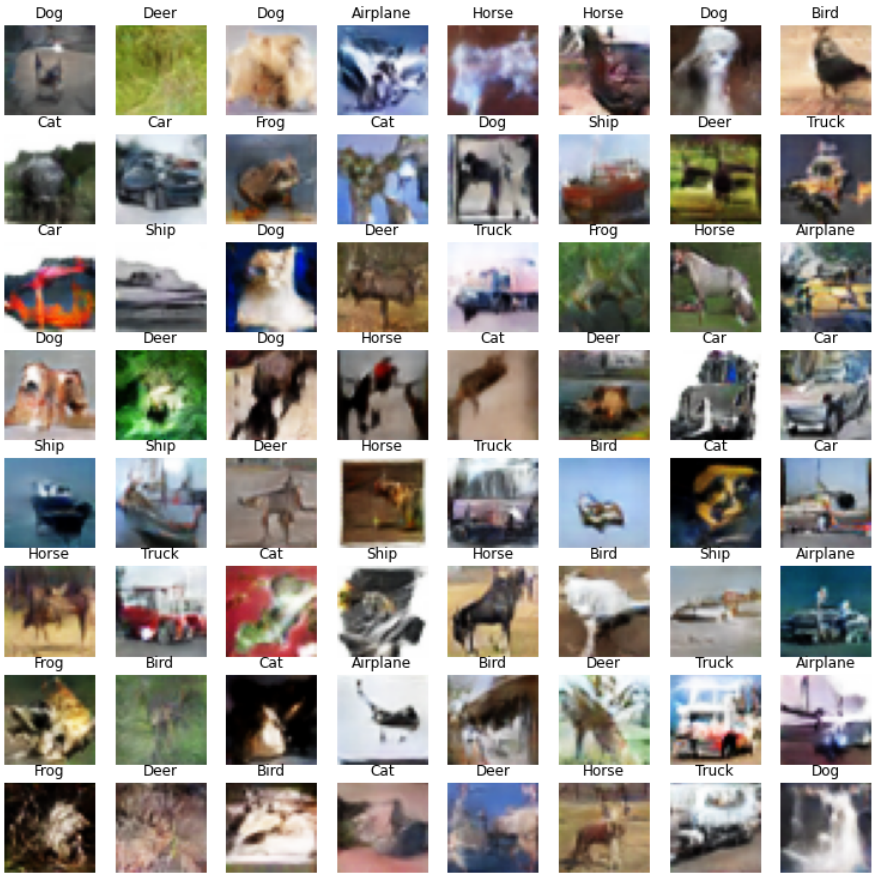
Step 753





Run set 3

As can be observed from the loss curve, spectral normalization does appear to allow for more stable training, but the overall impact on the GAN is still limited, suggesting that there may be a limit to the GAN architecture used.



Images Generated by the SN-DCGAN

SNGAN with Projection Improving the Generator and Discriminator

To improve the overall GAN architecture, I decided to try out the SNGAN with Projection Discriminator, from Miyato and Koyama, "CGANs with Projection Discriminator.". The following sections will provide a brief description of the changes I have to make to my current GAN.

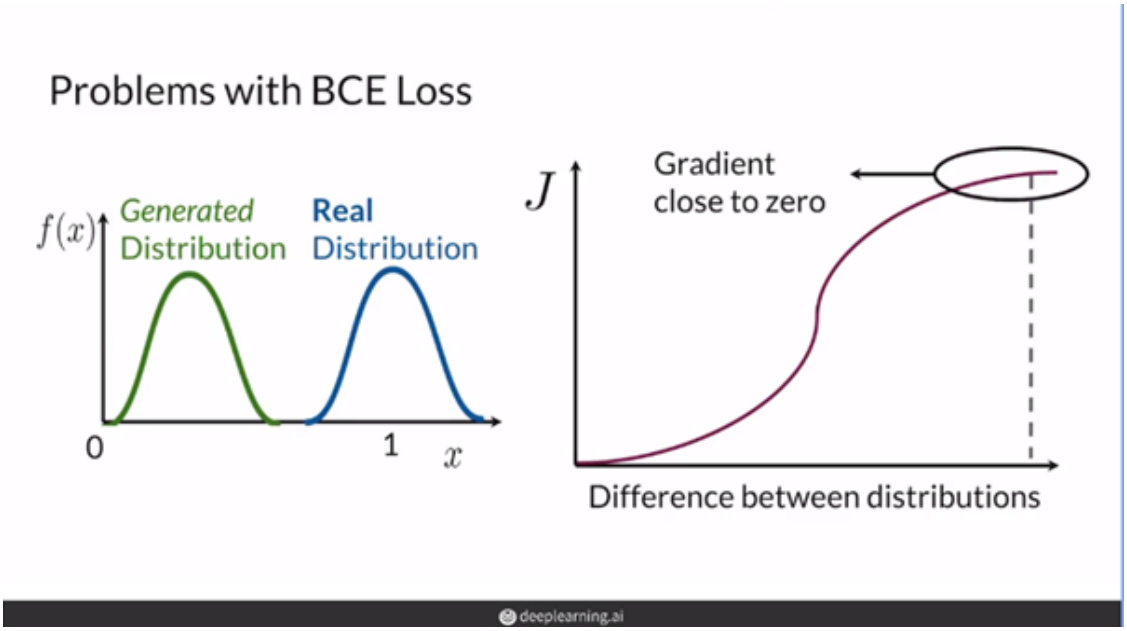
Changing the Loss Function

What's wrong with a BCE Loss?

In the beginning, our GAN made use of binary cross entropy loss between the output from the discriminator and a ones vector (penalizing the generator if the discriminator predicted its outputs as fake images).

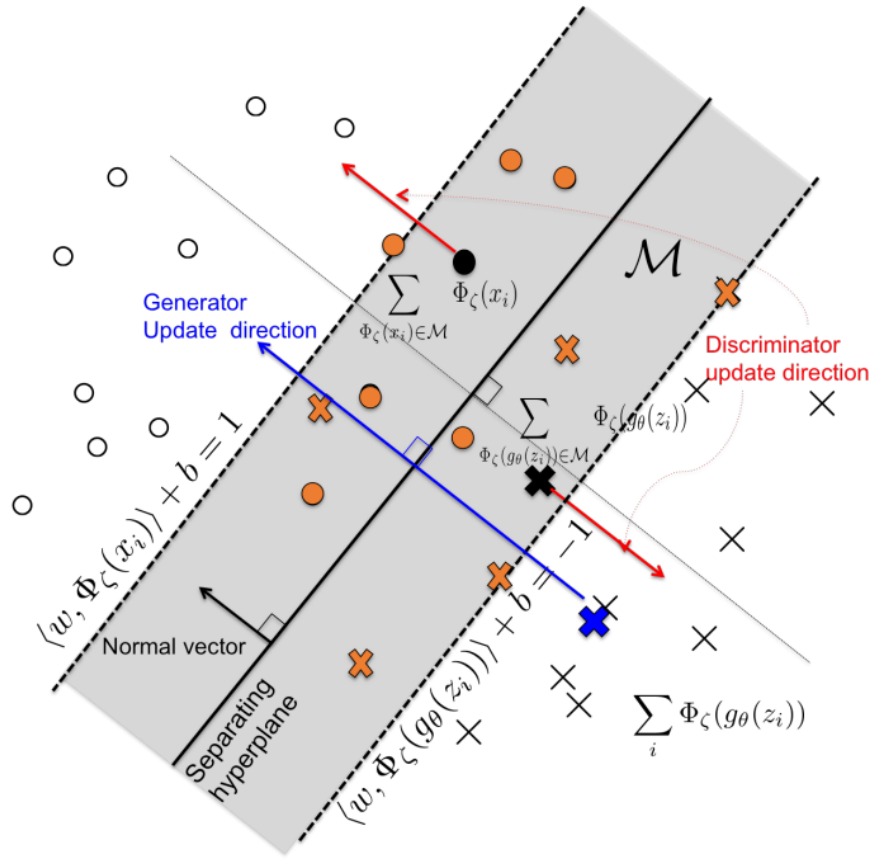
However, the issue with this is that the generator loss may not be correlated with image quality as the generator is evaluated against the predictions of the current discriminator, which itself constantly improves during training. This means that when the discriminator improves faster than the generator, the loss can actually increase even if the generator is also improving. This has several consequences:

- Using a binary cross entropy loss is uninformative during training.
- If the discriminator is too strong, the loss function will have flat regions as there is less overlap between the generated distribution and the target data distribution, resulting in vanishing gradients. (and thus preventing the generator from improving)



Vanishing Gradients Can Occur When The Discriminator Gets Too Strong (Credit: Build Basic Generative Adversarial Networks, deeplearning.ai)

Hinge Loss



Hinge Loss (Credit: Geometric GAN)

To replace the BCE Loss, I make use of the Hinge Loss as proposed in Lim, Jae Hyun, and Jong Chul Ye. “Geometric GAN.”

The idea is inspired by support vector classifiers: that the final classifier head of the discriminator will learn a linear boundary between the real and fake images such that the the real and fake images are maximally separately away from the boundary.

As such, the generator will try to generate samples such that they are closer to the boundary while the discriminator tries to classify images such that real and fake images are further away from the boundary.

To accomplish this, the Hinge Loss is introduced. Experimental results show that it results in more stable training and reduced risk of mode collapse, possibly due to the geometric intuition behind the loss.

The loss for the generator is calculated as follows:

$$L_G = -\mathbb{E}(D(G(z), y)$$

```
-fake_preds.mean()
```

This penalizes the generator if the discriminator predicts the fake samples as being further away from the separating hyperplane. (since the aim is to minimize the loss)

The loss for the discriminator is calculated as follows:

$$L_D = -\mathbb{E}[\min(0, D(x, y) - 1] - \mathbb{E}[\min(0, -D(G(z), y) - 1)]$$

```
-torch.minimum(
        torch.tensor(
                0.0, dtype=torch.float, device=real_preds.device
        ),
        real_preds - 1.0,
).mean()
- torch.minimum(
        torch.tensor(
                0.0, dtype=torch.float, device=fake_preds.device
        ),
        -fake_preds - 1.0,
).mean()
```

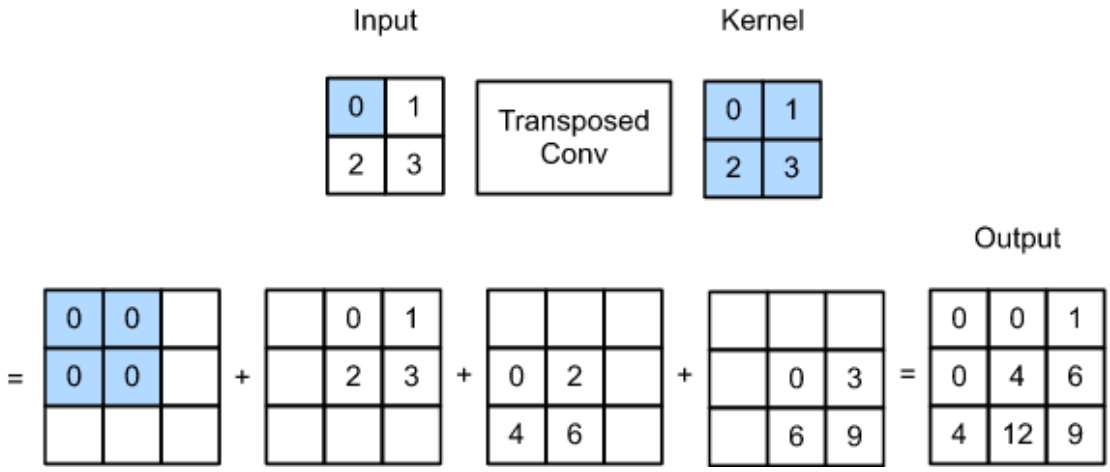
The discriminator is rewarded if the real samples are predicted as >1, and the fake samples predicted as <-1, such that the margin between the real and fake samples is maximized.

What's wrong with Transposed Convolutions?

One of the changes made in their generator architecture is to replace transposed convolutions (used in DCGANs), with fixed upsampling layers followed by convolutional layers.

Previously, I made use of transposed (aka fractionally strided) convolutions to perform upsampling.

Transposed convolutions work in the following way:



How a transposed convolution works (Credit: Dive into Deep Learning)

1. It starts by applying padding on the output feature map.
2. For each pixel in the input, we broadcast the pixel value and multiply it with each of the values in the kernel
3. The resulting matrix is pasted onto the output feature map

- When there is an overlap, values will be added together
- Any values in the padded region will be ignored

However, there is an issue with this as it can lead to checkboard artifacts. This is especially when the amount of overlap of the pixels being upscaled with the kernel is uneven when the kernel size is not divisible by the stride. However, even in the case where this is not

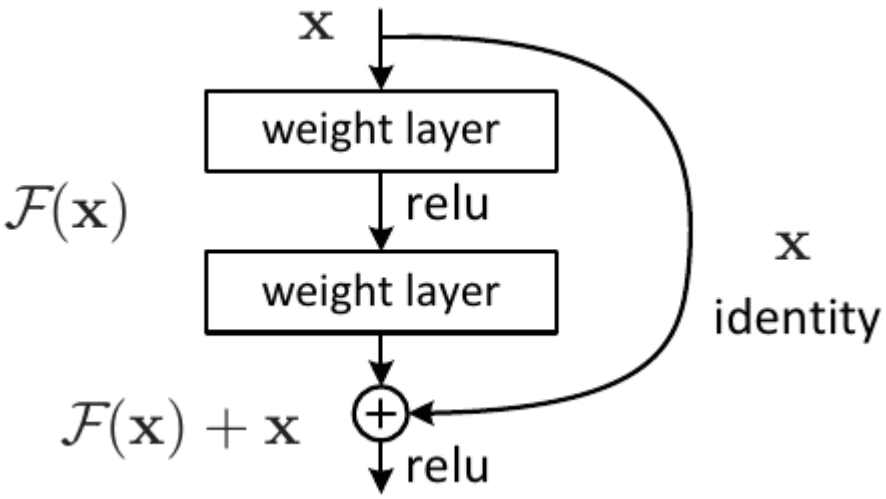
true (in my generator, the kernel size is divisible by the stride), artifacts can still be generated as transpose convolutions appear to be prone to creating artifacts as it is still easy for the model to learn kernels that cause artifacts. (Odena, Dumoulin, and Olah, "Deconvolution and Checkerboard Artifacts.")

To resolve this, it has been found that separating the upsampling process from the convolution layer helps to reduce artifacts. This works by using an upscaling algorithm (e.g. nearest neighbor interpolation) to first upsize the image, then use a normal convolutional layer to process the upscaled image.

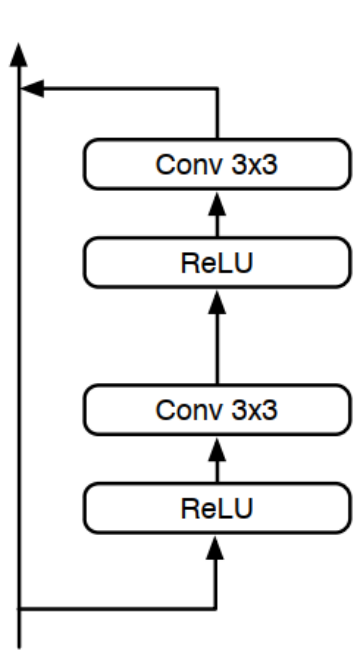
Credit: Odena, Dumoulin, and Olah, "Deconvolution and Checkerboard Artifacts."

Residual Connections

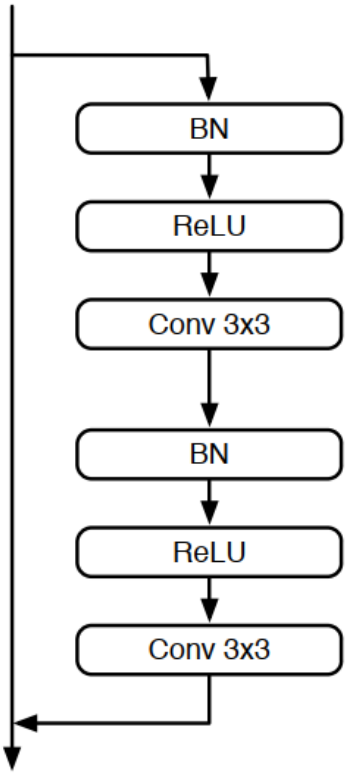
Another major change is to the generator and discriminator, where residual connections are added to the networks. The idea of a residual network was pioneered in He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition." to allow for the construction of deeper neural networks.



A Residual Block



(a) ResBlock architecture for the discriminator. Spectral normalization (Miyato et al., 2018) was applied to each *conv* layer.



(b) ResBlock for the generator.

Conditional Batch Normalisation

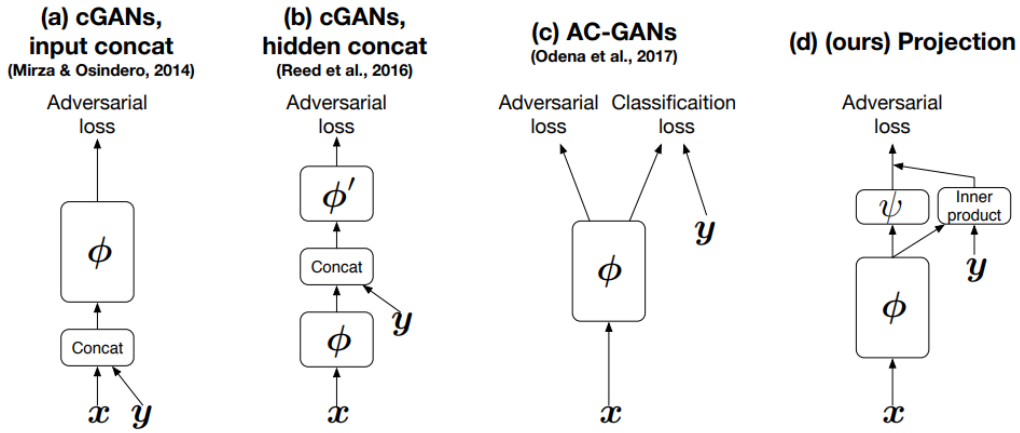
Instead of directly concatenating the class information into the input of the model, the class information is fed into the Batch Normalisation layer, where the BN layer learns class specific parameters.

My implementation from this is modified from the implementation in BigGAN, with the modification being to include the embedding layer directly into each cBN module instead of relying on a shared class embedding.

```
class ConditionalBatchNorm2d(nn.Module):
    def __init__(
        self,
        num_classes: int = 10,
        num_features: int = 256,
        eps: float = 1e-5,
        momentum: float = 0.1,
        norm_style: str = "bn",
    ):
        """https://github.com/ajbrock/BigGAN-PyTorch/blob/98459431a5

        :param output_size: [description]
        :type output_size: [type]
        :param input_size: [description]
        :type input_size: [type]
        :param eps: [description], defaults to 1e-5
        :type eps: [type], optional
        :param momentum: [description], defaults to 0.1
        :type momentum: float, optional
        :param norm_style: [description], defaults to "bn"
        :type norm_style: str, optional
        """
        super().__init__()
        self.num_features = num_features
        self.embed = spectral_norm(nn.Embedding(num_classes, num_fea
        # Prepare gain and bias layers
        self.gain = spectral_norm(nn.Linear(num_features, num_featur
        self.bias = spectral_norm(nn.Linear(num_features, num_featur
        # epsilon to avoid dividing by 0
        self.eps = eps
        # Momentum
        self.momentum = momentum
        # Norm style?
        self.norm_style = norm_style
```

```
        if self.norm_style in ["bn", "in"]:
            self.register_buffer("stored_mean", torch.zeros(num_feat
            self.register_buffer("stored_var", torch.ones(num_featur

    def forward(self, x, y):
        # Calculate class-conditional gains and biases
        y_embed = self.embed(y)
        gain = (1 + self.gain(y_embed)).view(y_embed.size(0), -1, 1,
        bias = self.bias(y_embed).view(y.size(0), -1, 1, 1)
        if self.norm_style == "bn":
            out = F.batch_norm(
                x,
                self.stored_mean,
                self.stored_var,
                None,
                None,
                self.training,
                0.1,
                self.eps,
            )
        elif self.norm_style == "in":
            out = F.instance_norm(
                x,
                self.stored_mean,
                self.stored_var,
                None,
                None,
                self.training,
                0.1,
                self.eps,
            )
        elif self.norm_style == "nonorm":
            out = x
        return out * gain + bias
```

Projection GAN



Discriminator models for conditional GANs (Credit: Miyato and Koyama, "CGANs with Projection Discriminator.")

> By construction, **any assumption about the form of the distribution would act as a regularization on the choice of the discriminator**. In this paper, we propose a specific form of the discriminator, a form motivated by a probabilistic model in which the **distribution of the conditional variable y given x is discrete or uni-modal continuous distributions.** This model assumption is in fact common in many real world applications, including class-conditional image generation and super-resolution. ...adhering to this assumption will **give rise to a structure of the discriminator that requires us to take an inner product between the embedded condition vector y and the feature vector**

The architecture used for the discriminator is a projection discriminator, where class information is provided via a dot product of the class embeddings and the final feature vector. As explained by Miyato and Koyama, this style of providing the class information to the discriminator acts as a form of regularization, providing greater stability during training.

```
class ResNetDiscriminator(nn.Module):
    def __init__(
        self,
        num_filters: int = 128,
        num_classes: int = 10,
        activation: callable = F.relu,
    ):
        """Implementation inspired by Projection Discriminator: http
```

```
        :param num_filters: [description], defaults to 128
        :type num_filters: int, optional
        :param num_classes: [description], defaults to 10
        :type num_classes: int, optional
        :param activation: [description], defaults to F.relu
        :type activation: callable, optional
        """
        super().__init__()
        self.num_filters = num_filters
        self.num_classes = num_classes
        self.activation = activation
        self.blocks = nn.Sequential(
            ResidualBlockDiscriminatorHead(3, num_filters, activatio
            ResidualBlockDiscriminator(
                num_filters, num_filters, activation=activation, dow
            ),
            ResidualBlockDiscriminator(
                num_filters, num_filters, activation=activation, dow
            ),
            ResidualBlockDiscriminator(
                num_filters, num_filters, activation=activation, dow
            ),
        )
        self.classifier = spectral_norm(nn.Linear(num_filters, 1, bi
        self.embed = spectral_norm(nn.Embedding(num_classes, num_fil

    def forward(self, x, y):
        h = self.blocks(x)
        h = self.activation(h)
        h = h.mean([2, 3])  # Global Avg Pooling
        out = self.classifier(h)
        out = out + torch.sum(self.embed(y) * h, axis=1, keepdims=Tr
        return out
```

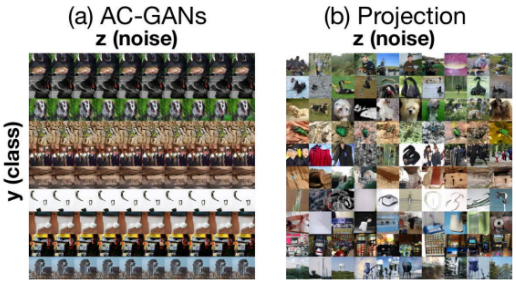


Figure 5: comparison of the images generated by (a) AC-GANs and (b) *projection*.

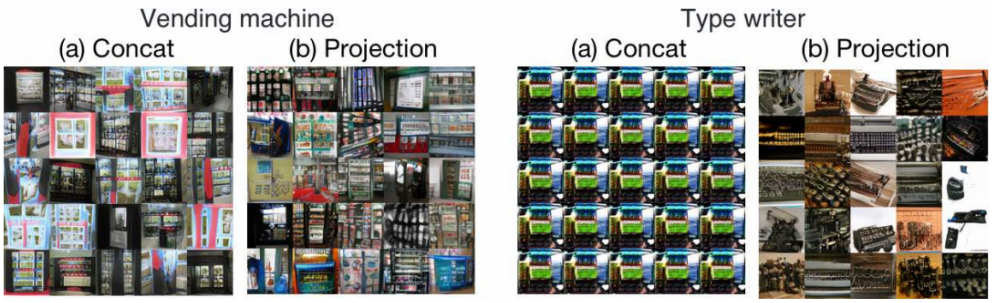


Figure 6: Collapsed images on the *concat* model.

Compared to other methods of injecting class information, a projection discriminator is able to generate more diverse images. Credit: Miyato and Koyama, "CGANs with Projection Discriminator."

> 💡 Compared to AC-GANs, a Projection Discriminator can produce more diverse results as the classification loss of an AC-GAN encourages the ACGAN to produce images that are easily classifiable, which often results in similar images being produced for each class. Thus, the benefit of the Projection method is that it reduces the risk of mode collapse during training as compared to both a normal cGAN and AC-GAN.

Putting Everything Together

```
class SNGAN(LightningModule):
    def __init__(
        self,
        latent_dim: int = 128,
        num_classes: int = 10,
        g_lr: float = 0.0002,
        d_lr: float = 0.0002,
        adam_betas: Tuple[float, float] = (0.0, 0.999),
        d_steps: int = 1,
        r1_gamma: Optional[float] = 0.2,
```

```
        w_init_policy: str = "ortho",
        **kwargs,
    ):
        super().__init__()

        self.latent_dim = latent_dim
        self.num_classes = num_classes
        self.g_lr = g_lr
        self.d_lr = d_lr
        self.betas = adam_betas
        self.d_steps = d_steps  # Number of Discriminator steps per
        self.r1_gamma = r1_gamma
        self.w_init_policy = w_init_policy
        self.save_hyperparameters()

        self.G = ResNetGenerator(
            latent_dim=latent_dim,
            num_classes=num_classes,
        )

        self.D = ResNetDiscriminator(num_classes=num_classes)

        if w_init_policy == "normal":
            init_func = self._normal_weights_init
        elif w_init_policy == "ortho":
            init_func = self._ortho_weights_init
        else:
            raise ValueError("Unknown Weight Init Policy")
        self.G.apply(init_func)
        self.D.apply(init_func)

        self.generator_loss = HingeGANLossGenerator()
        self.discriminator_loss = HingeGANLossDiscriminator()
        self.regularization_loss = R1(r1_gamma)

        self.viz_z = torch.randn(64, self.latent_dim)
        self.viz_labels = torch.LongTensor(torch.randint(0, self.nur

        self.unnormalize = NormalizeInverse((0.5, 0.5, 0.5), (0.5, 0

    @staticmethod
```

```
    def _normal_weights_init(m):
        if (
            isinstance(m, nn.Linear)
            or isinstance(m, nn.Conv2d)
            or isinstance(m, nn.Embedding)
        ):
            torch.nn.init.normal_(m.weight, 0.0, 0.02)
        elif isinstance(m, nn.BatchNorm2d):
            torch.nn.init.normal_(m.weight, 1.0, 0.02)
            torch.nn.init.zeros_(m.bias)

    @staticmethod
    def _ortho_weights_init(m):
        if (
            isinstance(m, nn.Linear)
            or isinstance(m, nn.Conv2d)
            or isinstance(m, nn.Embedding)
        ):
            torch.nn.init.orthogonal_(m.weight)

    def forward(self, z, labels):
        return self.G(z, labels)

    # Alternating schedule for optimizer steps (i.e.: GANs)
    def optimizer_step(
        self,
        epoch,
        batch_idx,
        optimizer,
        optimizer_idx,
        optimizer_closure,
        on_tpu,
        using_native_amp,
        using_lbfgs,
    ):
        # update discriminator opt every step
        if optimizer_idx == 1:
            optimizer.step(closure=optimizer_closure)

        # update generator opt every 4 steps
        if optimizer_idx == 0:
```

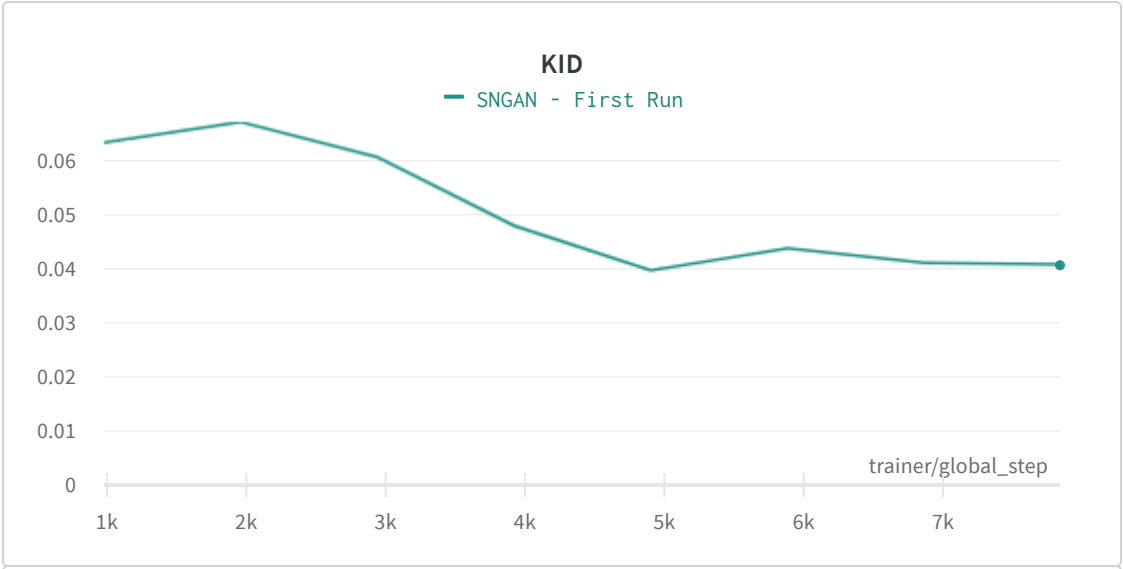
```
            if (batch_idx + 1) % self.d_steps == 0:
                optimizer.step(closure=optimizer_closure)
            else:
                # call the closure by itself to run `training_step`
                optimizer_closure()

    def training_step(self, batch, batch_idx, optimizer_idx):
        imgs, labels = batch  # Get real images and corresponding la

        # Generate Noise Vector z
        z = torch.randn(imgs.shape[0], self.latent_dim)
        z = z.type_as(imgs)  # Ensure z runs on the same device as t
        self.fake_labels = torch.LongTensor(
            torch.randint(0, self.num_classes, (imgs.shape[0],))
        ).to(self.device)
        # Train Generator
        if optimizer_idx == 0:
            # Generate Images
            self.fakes = self.forward(z, self.fake_labels)

            # Classify Generated Images with Discriminator
            fake_preds = torch.squeeze(self.D(self.fakes, self.fake_

            # We want to penalize the Generator if the Discriminator
            # Hence, set the target as a 1's vector
            g_loss = self.generator_loss(fake_preds)

            self.log(
                "train_gen_loss",
                g_loss,
                on_epoch=True,
                on_step=False,
                prog_bar=True,
            )  # Log Generator Loss
            tqdm_dict = {
                "g_loss": g_loss,
            }
            output = OrderedDict(
                {"loss": g_loss, "progress_bar": tqdm_dict, "log": t
            )
            return output
```

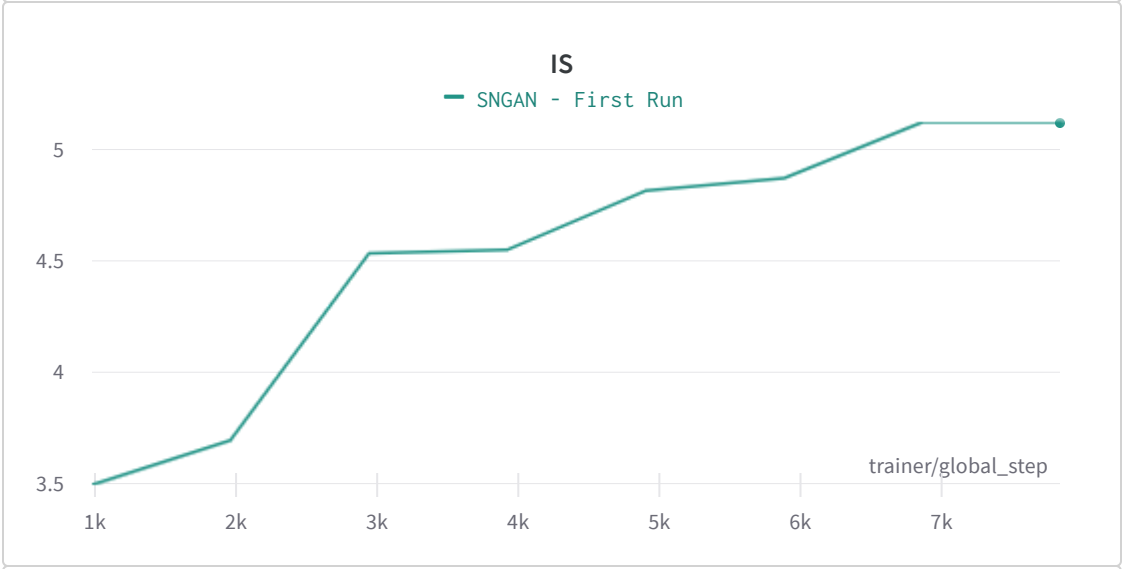
```
        # Train Discriminator
        if optimizer_idx == 1:
            # Train on Real Data
            imgs.requires_grad = True
            real_preds = torch.squeeze(self.D(imgs, labels))
            # Train on Generated Images
            self.fakes = self.forward(z, self.fake_labels)
            fake_preds = torch.squeeze(self.D(self.fakes, self.fake_
            d_loss = self.discriminator_loss(real_preds, fake_preds)
            if self.r1_gamma is not None:
                d_loss = d_loss + self.regularization_loss(real_pred
            self.log(
                "train_discriminator_loss",
                d_loss,
                on_epoch=True,
                on_step=False,
                prog_bar=True,
            )
            tqdm_dict = {
                "d_loss": d_loss,
            }
            output = OrderedDict(
                {"loss": d_loss, "progress_bar": tqdm_dict, "log": t
            )
            return output

    def training_epoch_end(self, outputs):
        # Log Sampled Images
        sample_imgs = self.unnormalize(self.fakes[:64].detach()).cpu
        sample_labels = self.fake_labels[:64].detach().cpu()
        num_rows = int(np.floor(np.sqrt(len(sample_imgs))))
        fig = visualize(sample_imgs, sample_labels, grid_shape=(num_
        self.logger.log_image(key="generated_images", images=[fig])
        plt.close(fig)
        del sample_imgs
        del sample_labels
        del fig
        gc.collect()

    def validation_step(self, batch, batch_idx):
```

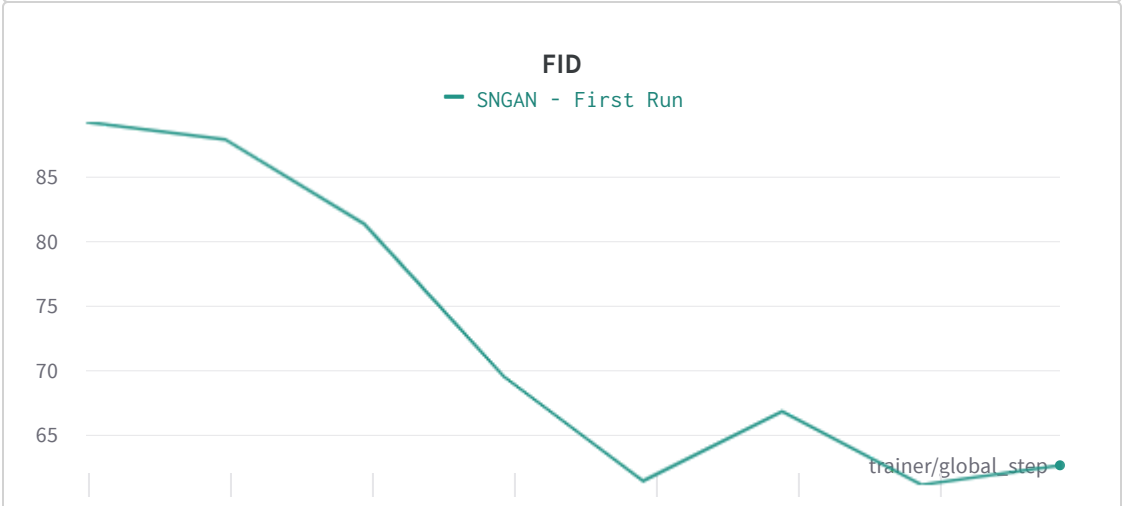
```
        pass

    def validation_epoch_end(self, outputs):
        sample_imgs = (
            self.forward(self.viz_z.to(self.device), self.viz_labels
            .cpu()
            .detach()
        )
        fig = visualize(sample_imgs, self.viz_labels.cpu().detach(),
        metrics = torch_fidelity.calculate_metrics(
            input1=torch_fidelity.GenerativeModelModuleWrapper(
                self.G, self.latent_dim, "normal", self.num_classes
            ),
            input1_model_num_samples=10000,
            input2="cifar10-val",
            isc=True,
            fid=True,
            kid=True,
        )
        self.logger.log_image(key="Validation Images", images=[fig])
        plt.close(fig)
        del sample_imgs
        del fig
        gc.collect()
        self.log("FID", metrics["frechet_inception_distance"], prog_
        self.log("IS", metrics["inception_score_mean"], prog_bar=Tru
        self.log("KID", metrics["kernel_inception_distance_mean"], p

    def configure_optimizers(self):
        """Define the optimizers and schedulers for PyTorch Lightnin

        :return: A tuple of two lists - a list of optimizers and a l
        :rtype: Tuple[List, List]
        """
        opt_G = Adam(
            self.G.parameters(), lr=self.g_lr, betas=self.betas
        )  # optimizer_idx = 0
        opt_D = Adam(
            self.D.parameters(), lr=self.d_lr, betas=self.betas
        )  # optimizer_idx = 1
        return [opt_G, opt_D], []
```

Training

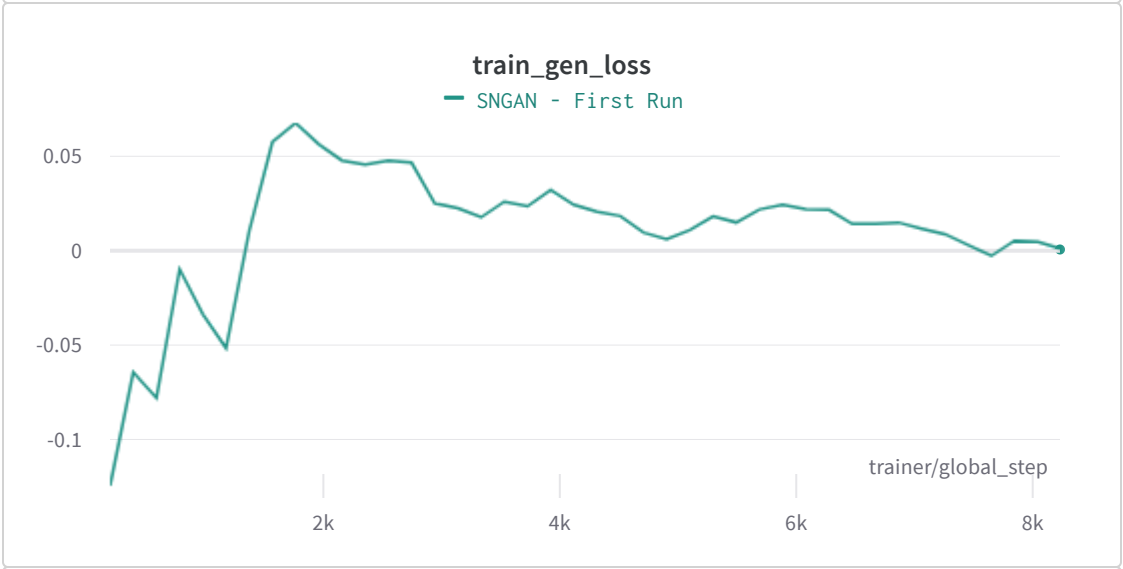
I first began training the SNGAN with the same hyperparameters as the previous GANs.

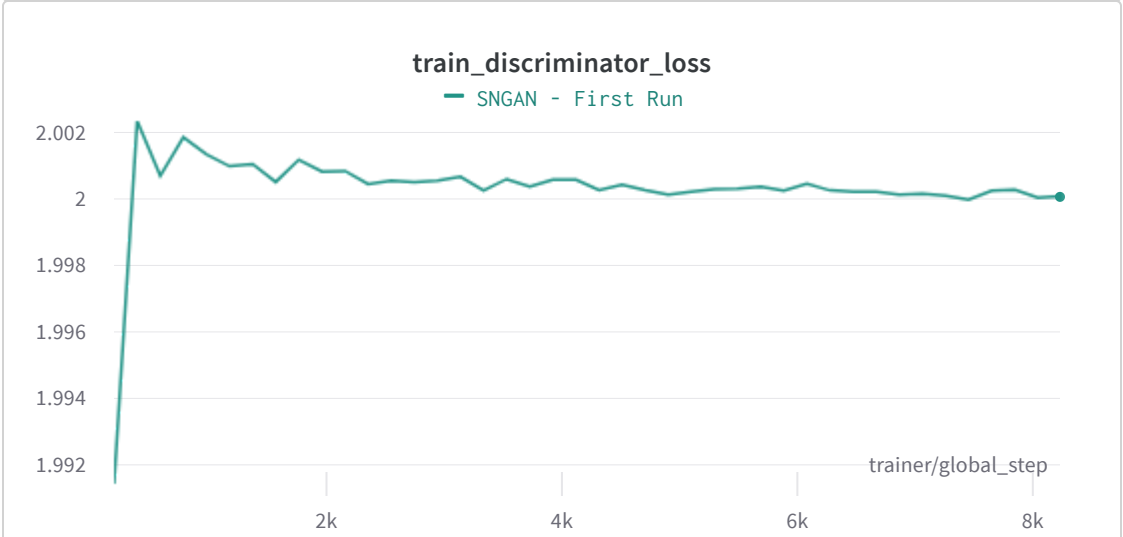


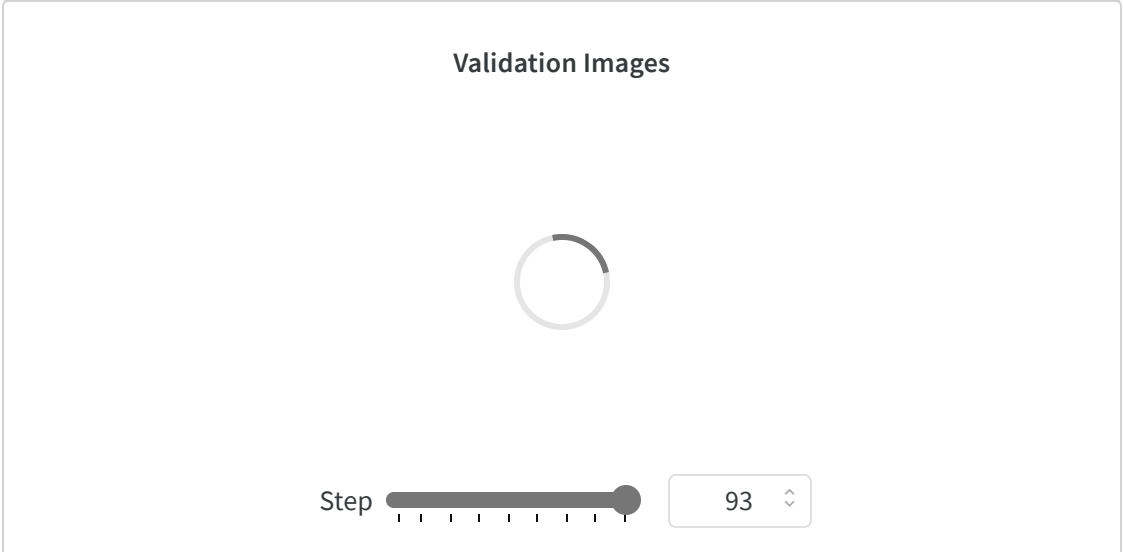


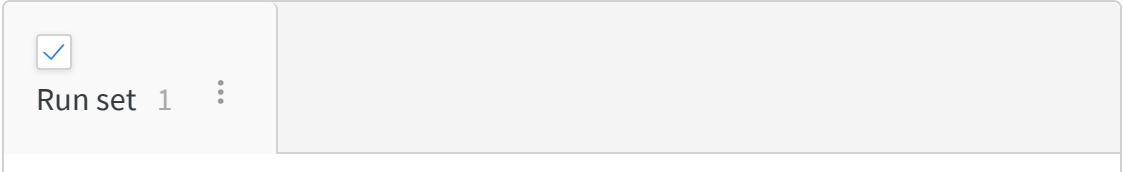



1k
2k
3k
4k
5k
6k
7k


train_gen_loss
SNGAN - First Run
0.05
0
-0.05
-0.1
trainer/global_step
2k
4k
6k
8k


train_discriminator_loss
SNGAN - First Run
2.002
2
1.998
1.996
1.994
1.992
trainer/global_step
2k
4k
6k
8k


Validation Images
Step
93


Run set 1

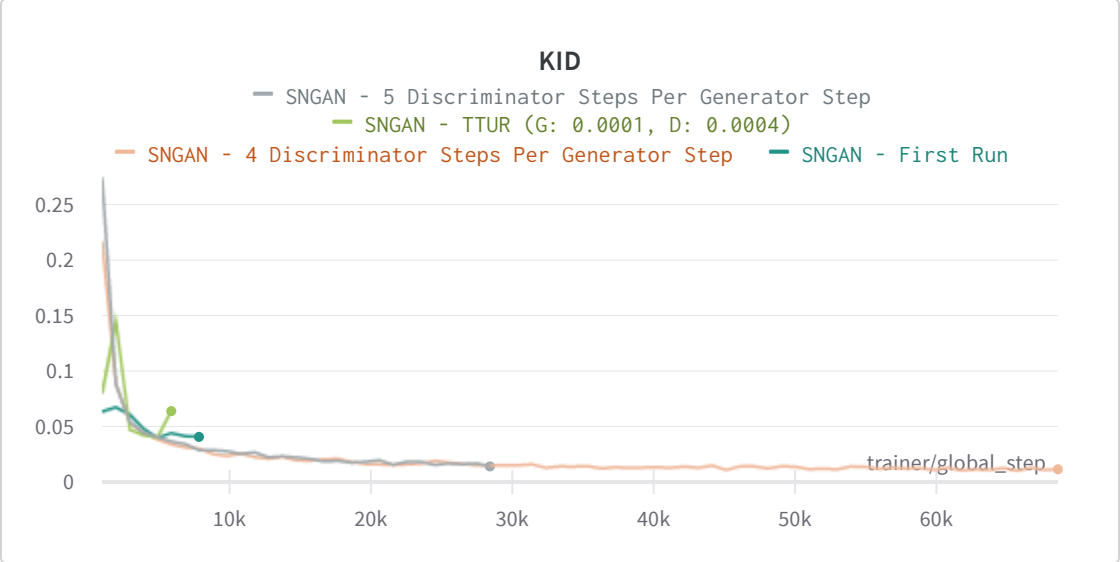
The first attempt at training the SNGAN failed, as it suffered from mode collapse. Looking at the discriminator loss, which remained extremely high from the start, it seemed that the discriminator was unable to keep up from the start, and thus it became easy for the generator to find modes of the distribution that would fool the discriminator.

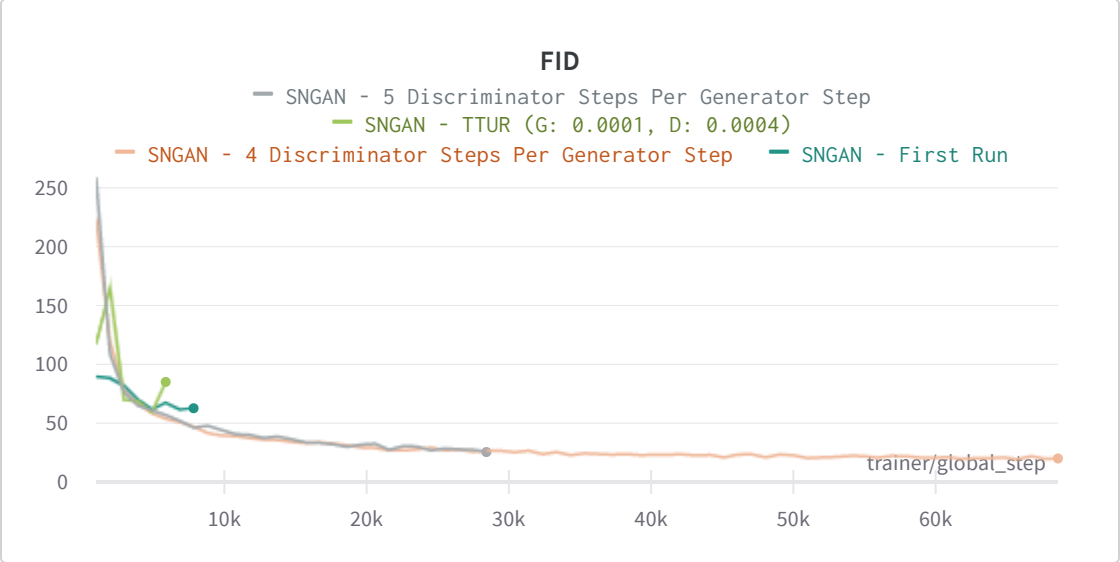
Training Collapse Early On - Epoch 41

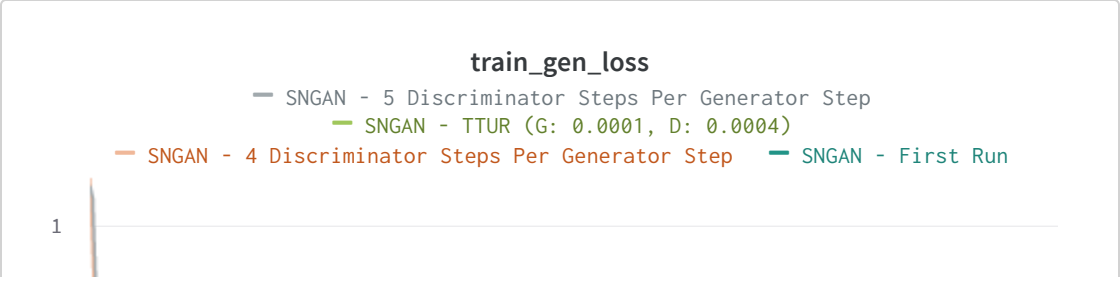
Experimenting with TTUR and Unbalanced Discriminator Updates

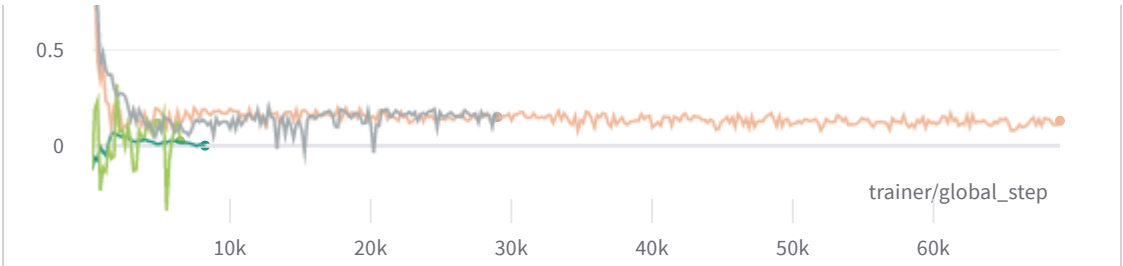
To attempt to prevent mode collapse, I now move my attention to tuning the learning rates for both the generator and discriminator. I will attempt a few strategies:

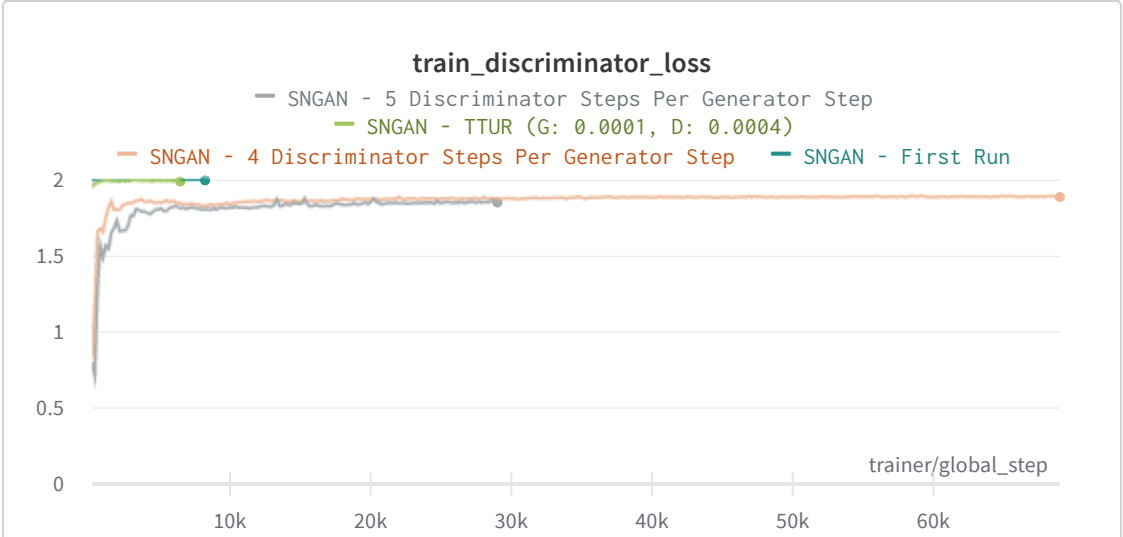
- Keep the same learning rate for the generator and discriminator, but increase the number of discriminator updates per generator update
- Apply the two time-scale update rule (TTUR) by having a higher learning rate for the discriminator as compared to the generator





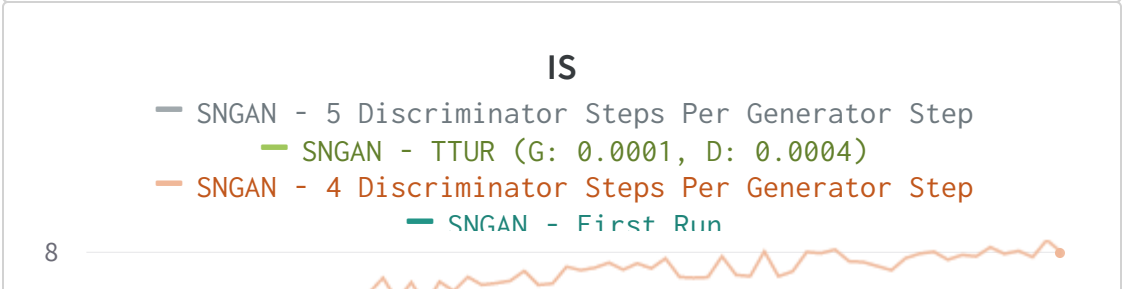


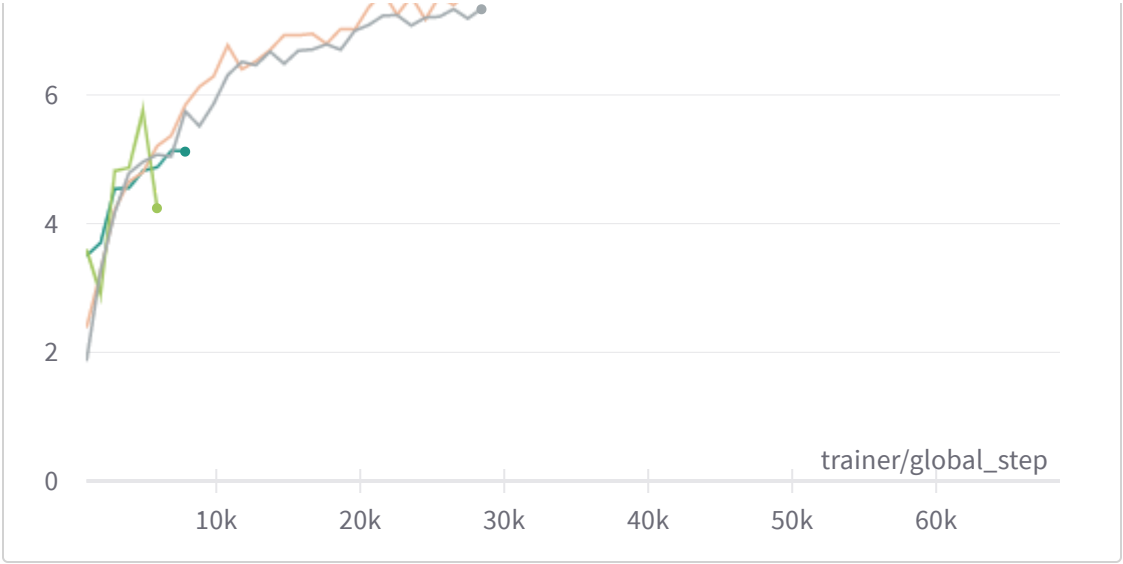




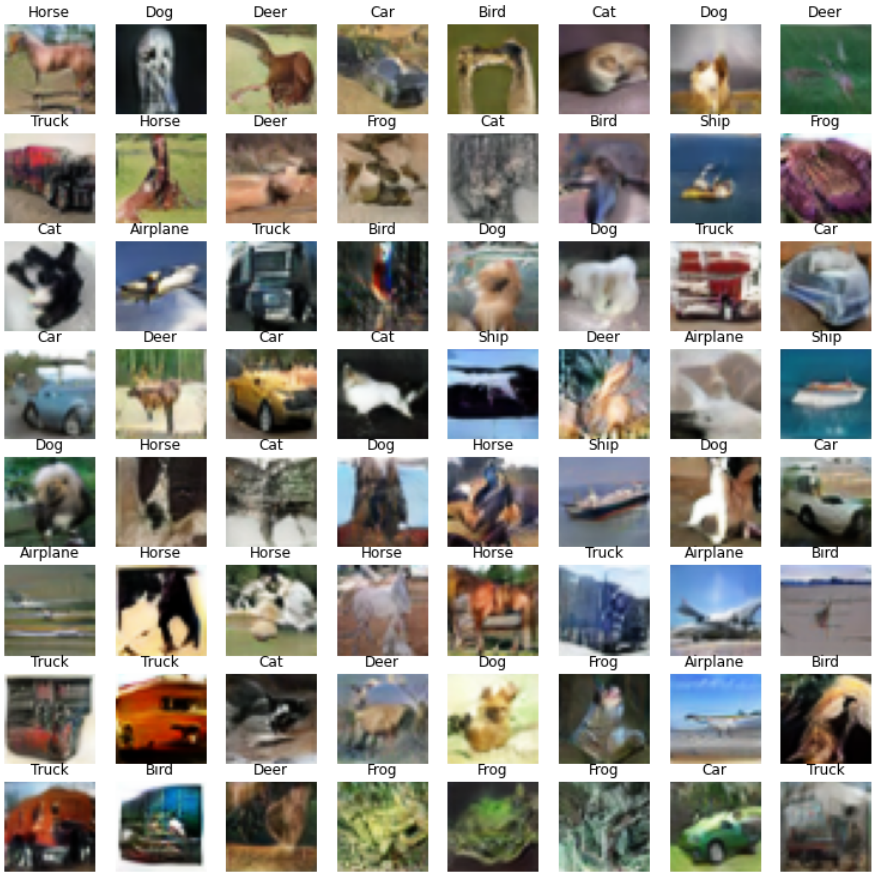
Validation Images

Step 837



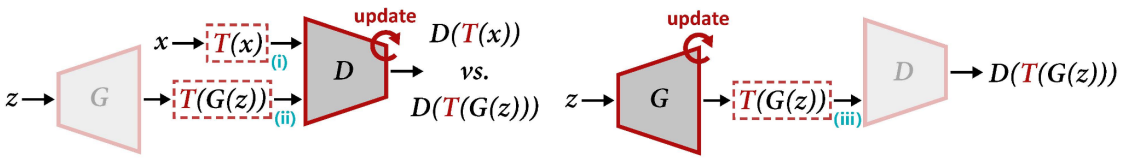


Run set 4



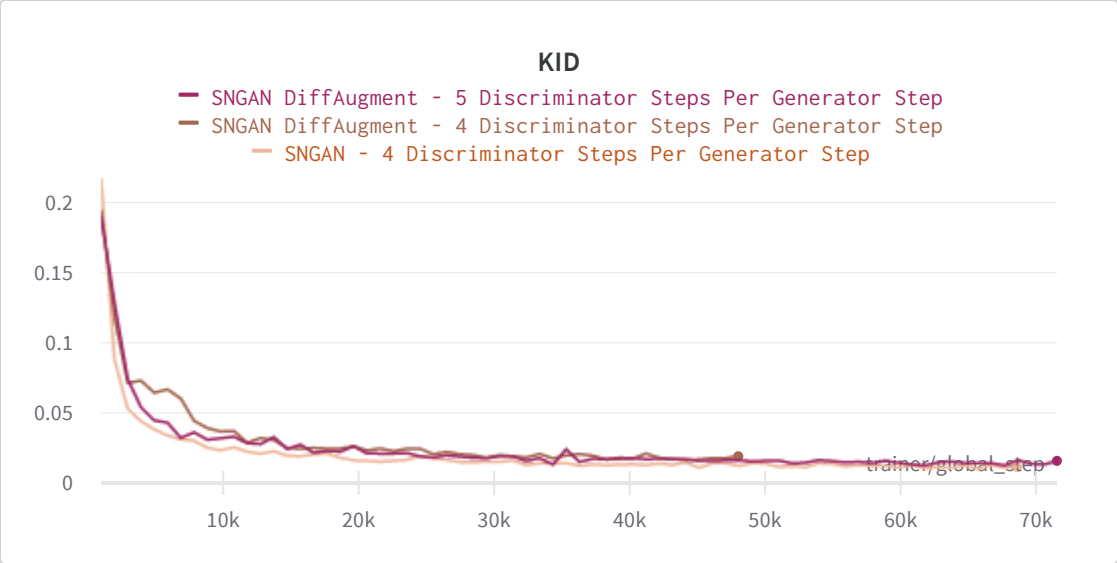
As can be observed, the training becomes much more stable after tuning of the ratio of discriminator updates to generator updates and the learning rate. Of the various experiments, it seems that setting a ratio of 4 discriminator optimization steps per generator optimization steps works out the best, with a final KID of 0.01144 (or an FID of 19.988).

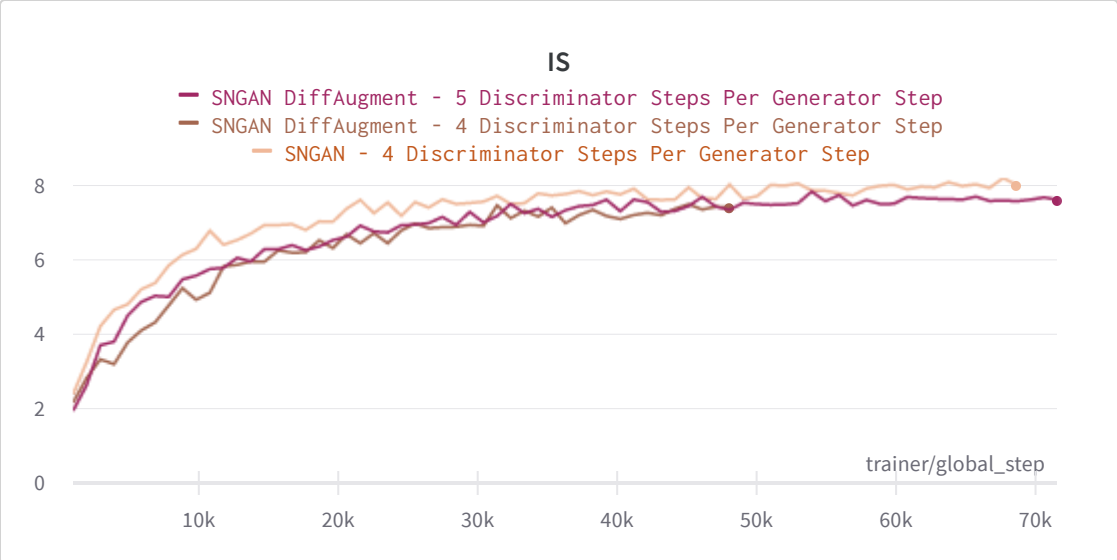
DiffAugment

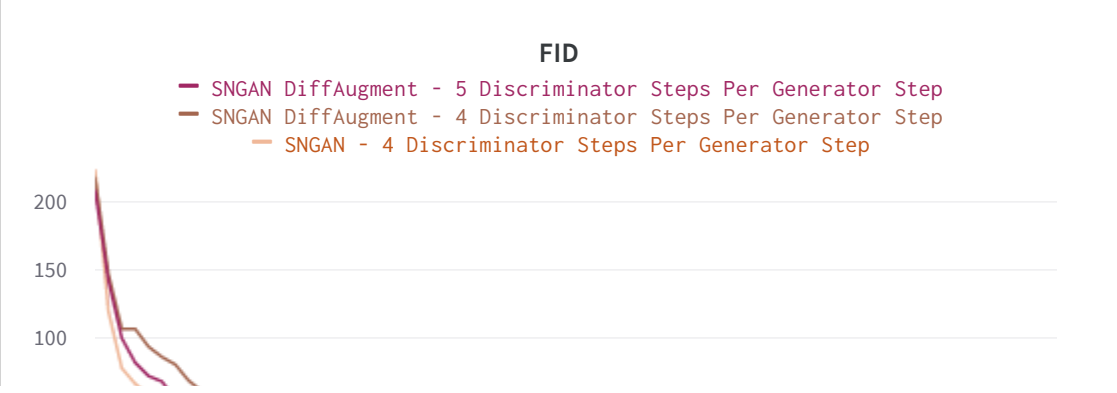


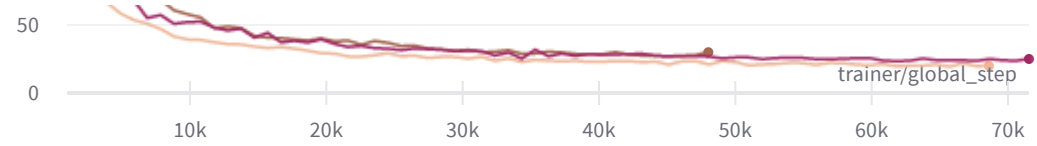
Credit: Zhao et al., "Differentiable Augmentation for Data-Efficient GAN Training."

To improve the performance of my GAN, I attempted to introduce data augmentation in the form of DiffAugment. DiffAugment applies data augmentation to both the real and fake images. Unlike typical data augmentations, these data augmentations are differentiable, allowing gradients to be propagated through the augmentation to the generator, allowing data augmentation to thus be applied when training the generator as well, maintaining the balance between the generator and discriminator during training.





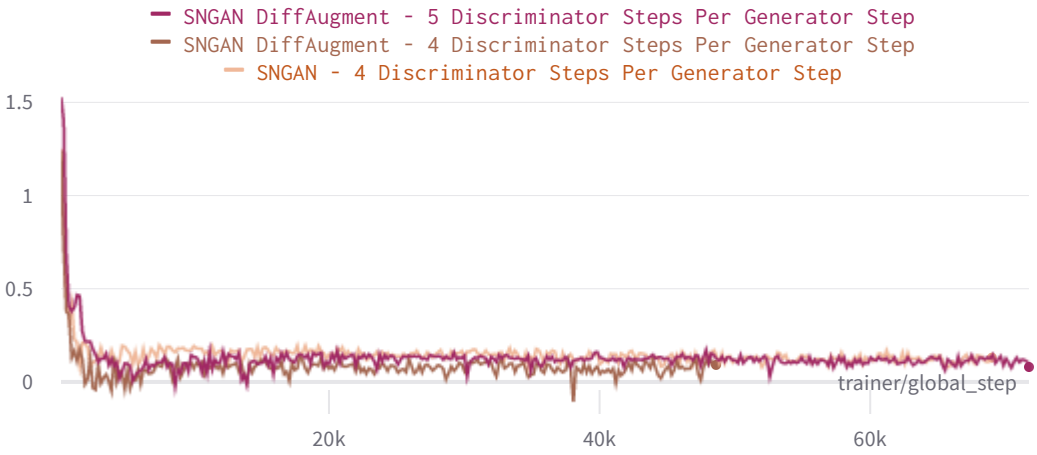




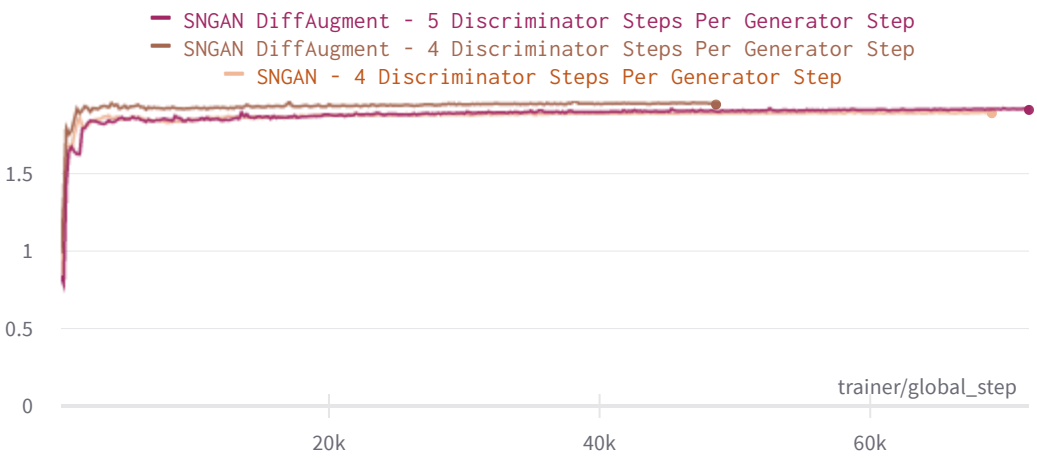
Validation Images

Step 873

train_gen_loss



train_discriminator_loss



Run set 3

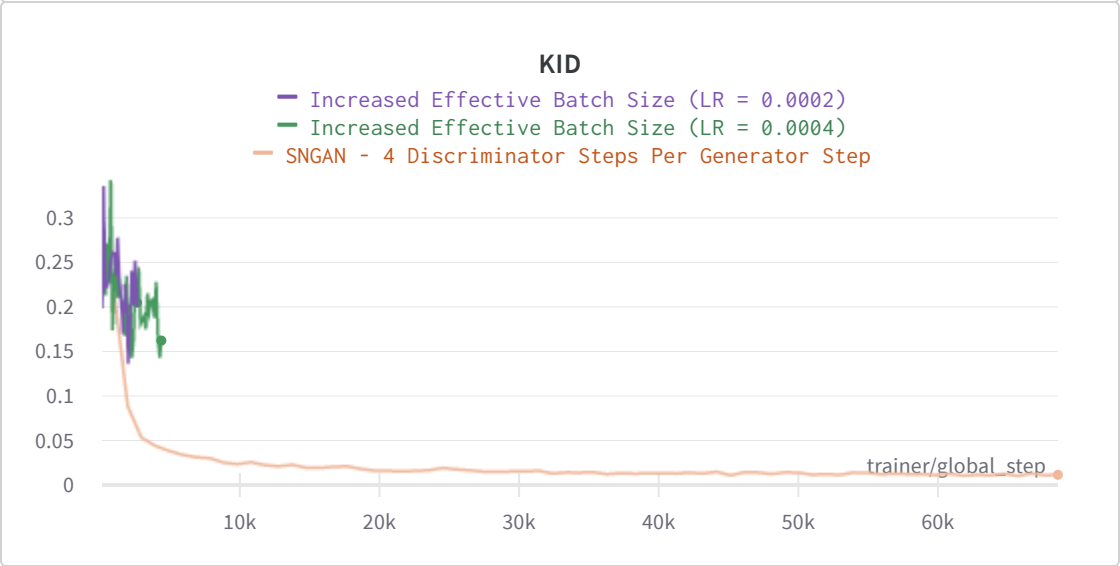
Applying DiffAugment does not appear to help in training as while training appears to be stable, it seems that the GAN metrics are not an improvement over the baseline without any data augmentation.

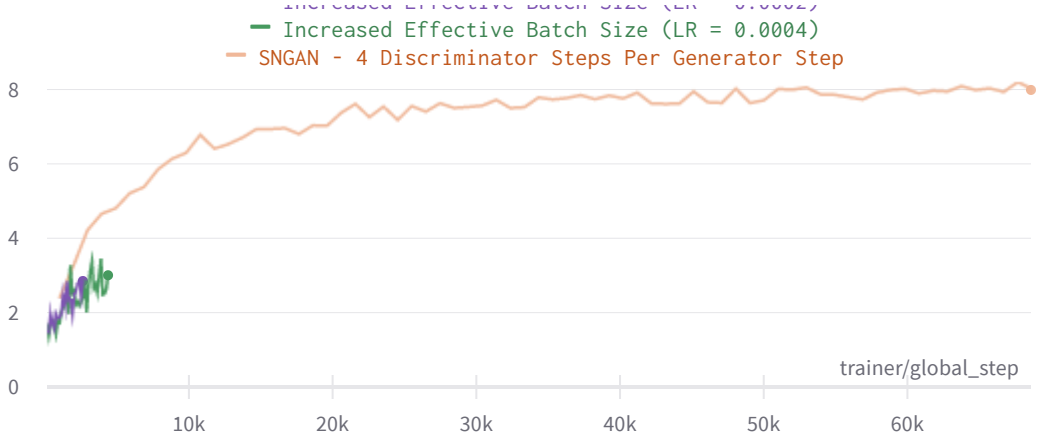
Increasing Batch Size

As mentioned in the earlier section on hyperparameters, an increase in batch size has been shown to lead to an improvement in FID. However, given the limited GPU memory available, there is usually a limit to the batch size. However, I can increase the effective batch size by performing gradient accumulation. With this technique, I increase my effective batch size by 8 times, getting an effective batch size of 2048.

```
trainer = Trainer(
    check_val_every_n_epoch=5,
    logger=wandb_logger,
    max_epochs=1000,
    callbacks=[
        ModelSummary(3),
    ],
    gpus=1,
    accumulate_grad_batches=8 # 8 * 256 = 2048
)
```

When increasing the batch size, the learning rate also needs to be adjusted, so I attempt two learning rates: one which is the same as before, and the other scaled by a factor of 2.

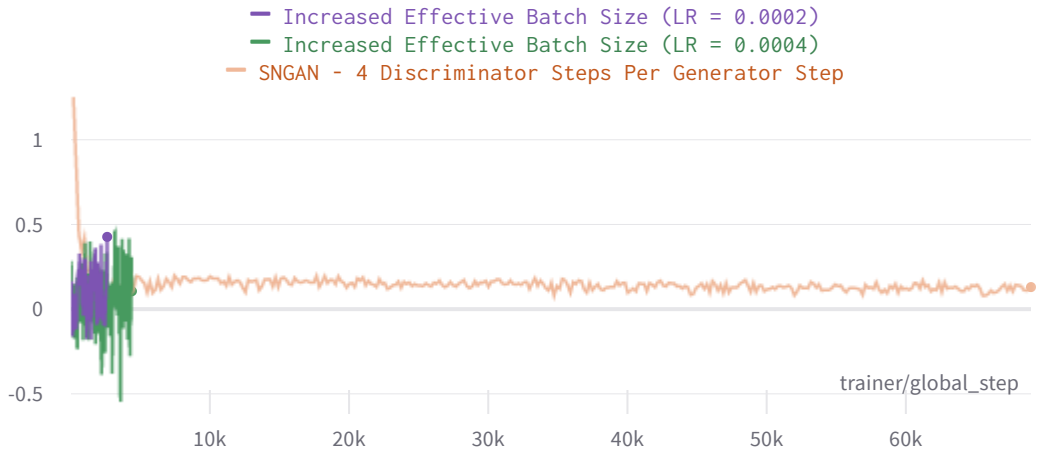




Validation Images

Step 837

train_gen_loss



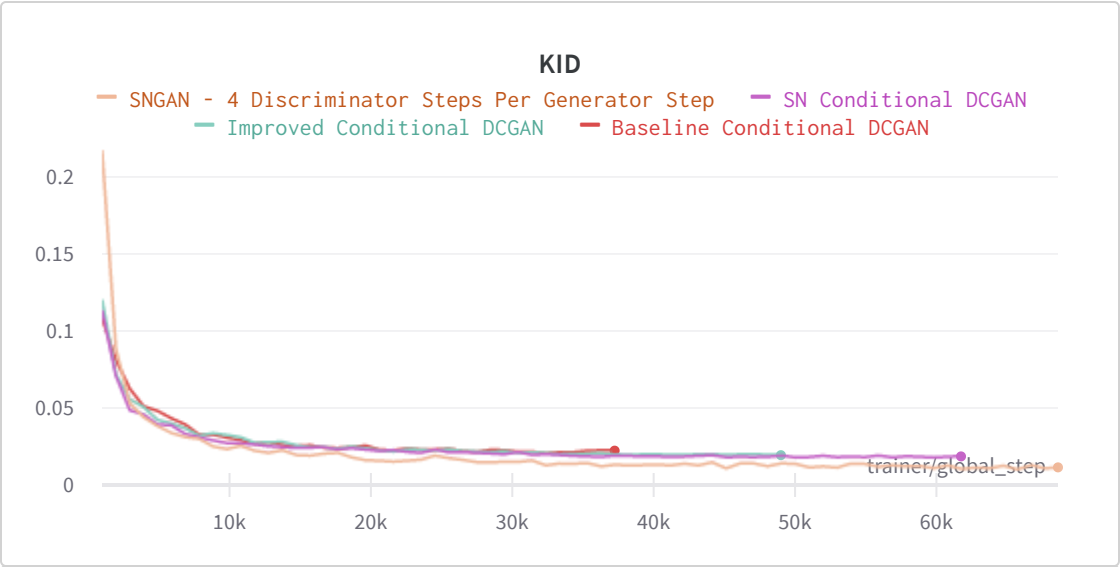
Run set 3

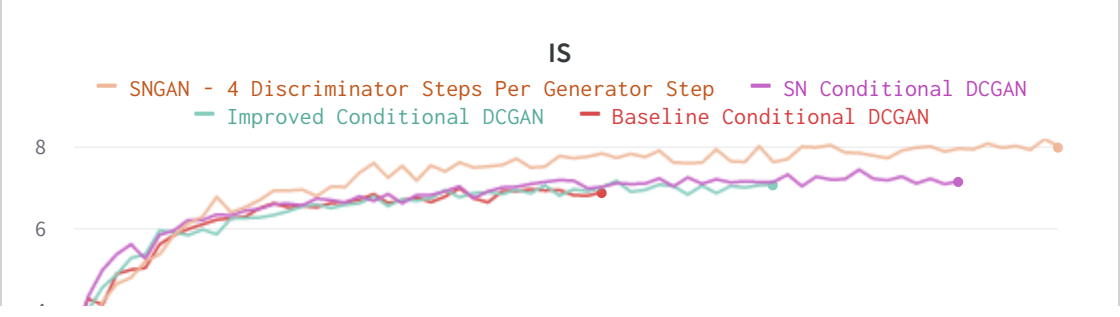
Unfortunately in this case, it appears that a larger batch size causes the training to become extremely unstable, and thus a failure in convergence.

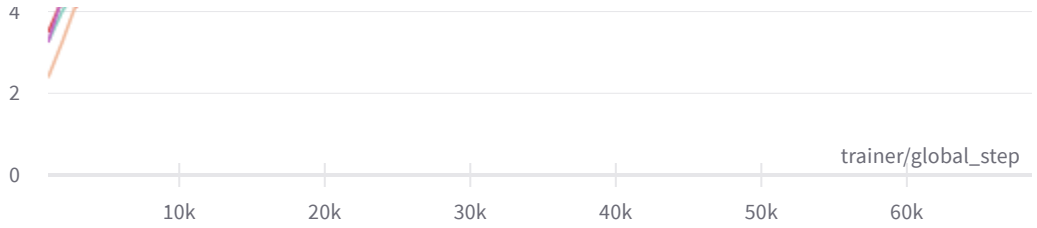
> 💡 I suspect that given more time, I would be able to find a learning rate that can result in stable convergence.

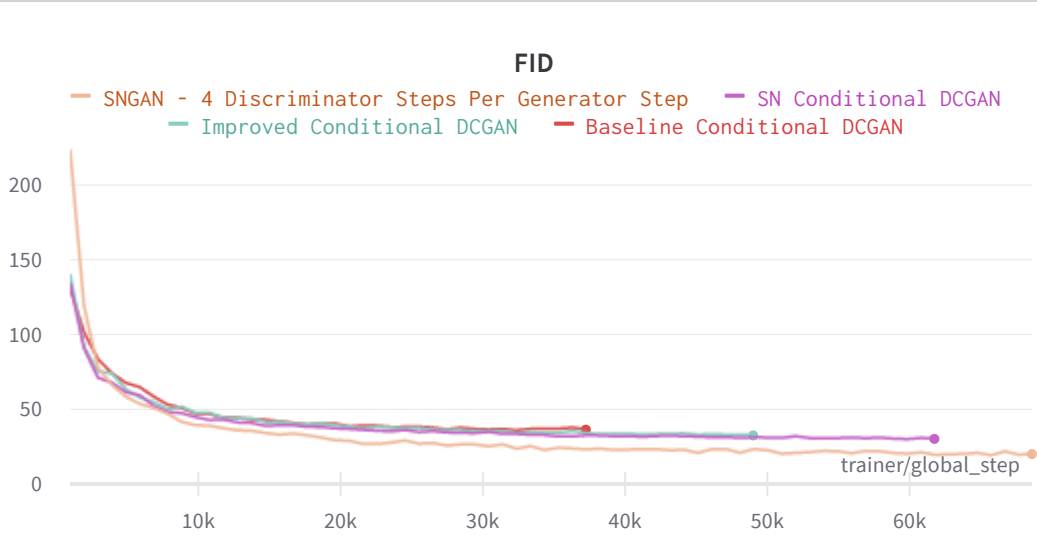
Results

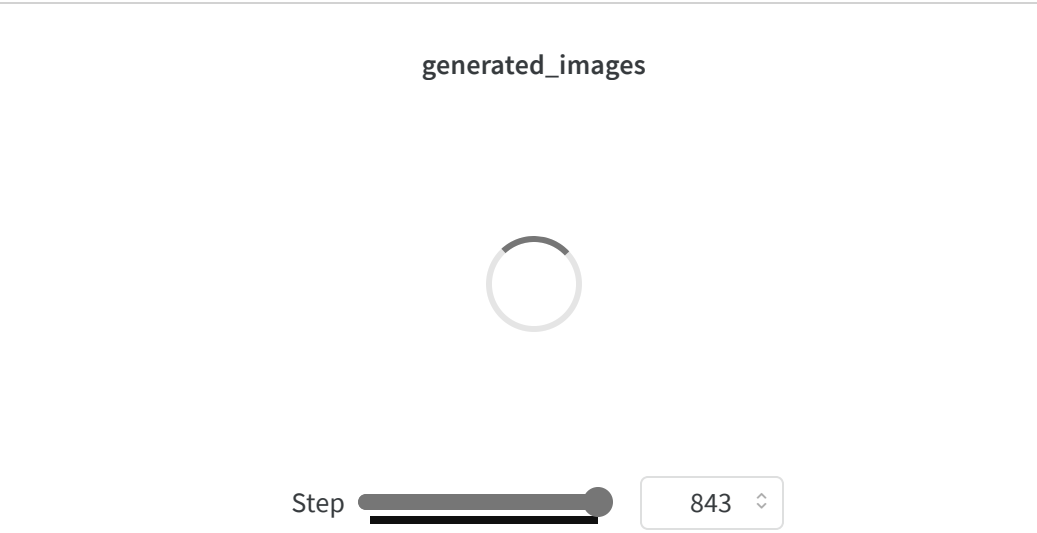
Summary of Training

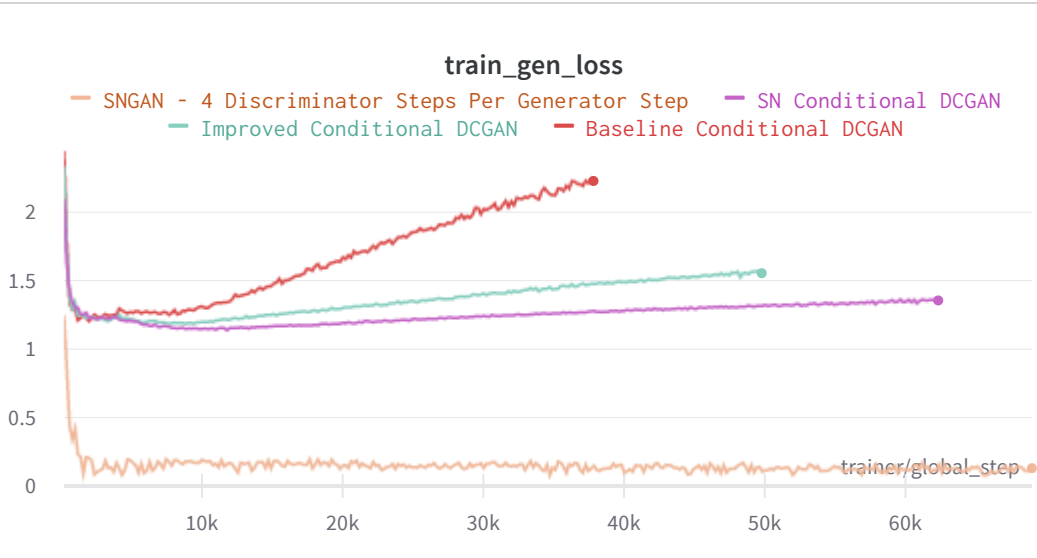


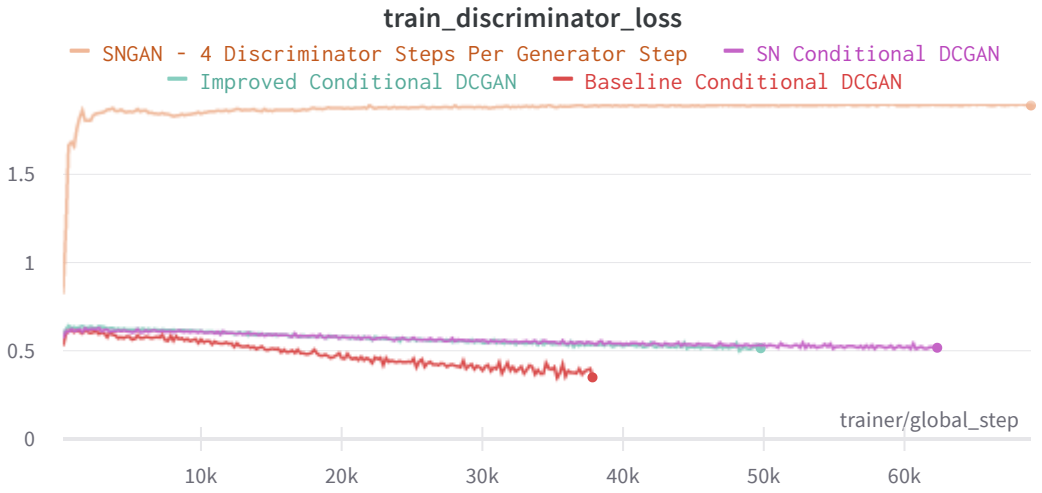



4
2
0
trainer/global_step
10k
20k
30k
40k
50k
60k


FID
SNGAN - 4 Discriminator Steps Per Generator Step
SN Conditional DCGAN
Improved Conditional DCGAN
Baseline Conditional DCGAN
200
150
100
50
0
trainer/global_step
10k
20k
30k
40k
50k
60k


generated_images
Step
843


train_gen_loss
SNGAN - 4 Discriminator Steps Per Generator Step
SN Conditional DCGAN
Improved Conditional DCGAN
Baseline Conditional DCGAN
2
1.5
1
0.5
0
trainer/global_step
10k
20k
30k
40k
50k
60k



Validation Images

Step 837

Run set 4

Based on the overall training results, I decide to select the SNGAN trained with a ratio of 4 discriminator updates per generator update as the final model, as it attained the best KID score.

Generating 1000 Images

To see the results of the GAN, I generated 100 images per class in CIFAR-10.

Source Code

```
run = wandb.init()
artifact = run.use_artifact('tiencheng/DELE_CA2_GAN/model-19kepjip:v
artifact_dir = artifact.download()
state_dict = torch.load("/content/dele-generative-adversarial-networ
model.load_state_dict(state_dict["state_dict"])
model.cuda().eval()
BATCH_SIZE = 100
with torch.no_grad():
  for batch in range(10):
    labels = torch.LongTensor(torch.randint(0, 10, (100, ))).cuda()
    z = torch.Tensor(truncated_z_sample(BATCH_SIZE, 128, truncation=
    images = model(z, labels).cpu()
    fig = visualize(images, labels.cpu(), grid_shape=(10, 10), figsi
    plt.savefig(f"{batch}.png")
```

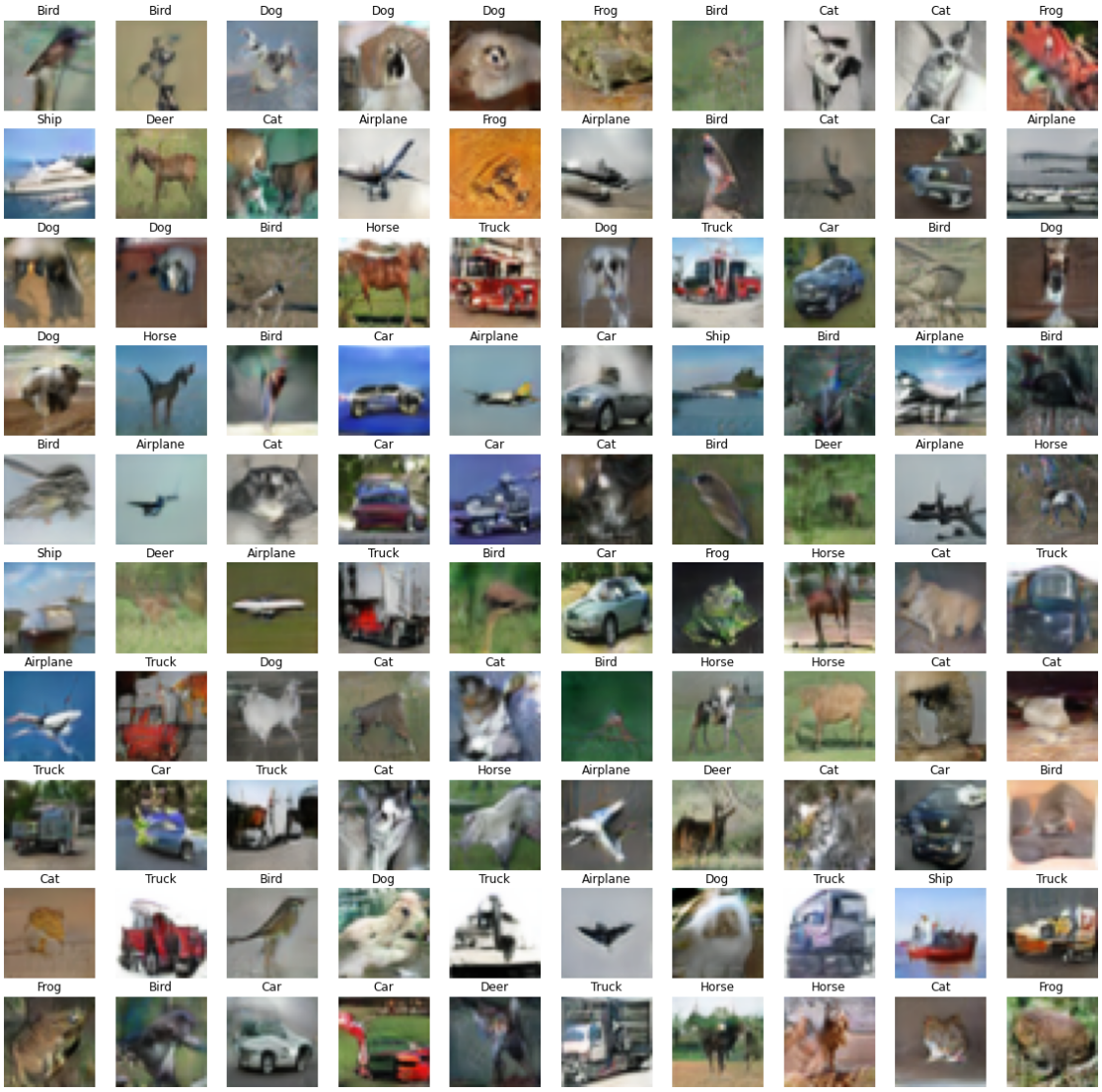
Experimenting with Truncation Trick

To improve the quality of the generated samples further, at the cost of some diversity, I employ the truncation trick. When generating images, the GAN takes in a noise vector, which is usually sampled from a normal distribution. As such, the model typically sees values that are closer to the mean of the distribution as the noise. So, by truncating the values in the noise vector, I can force the values in the noise vector to be closer to the mean of the distribution, resulting in a higher quality output as we are now sampling from points in the latent space that the model is more familiar with. The drawback is that truncating too much can result in the GAN producing images that are very similar.

Figure 2: (a) The effects of increasing truncation. From left to right, the threshold is set to 2, 1, 0.5, 0.04. (b) Saturation artifacts from applying truncation to a poorly conditioned model.

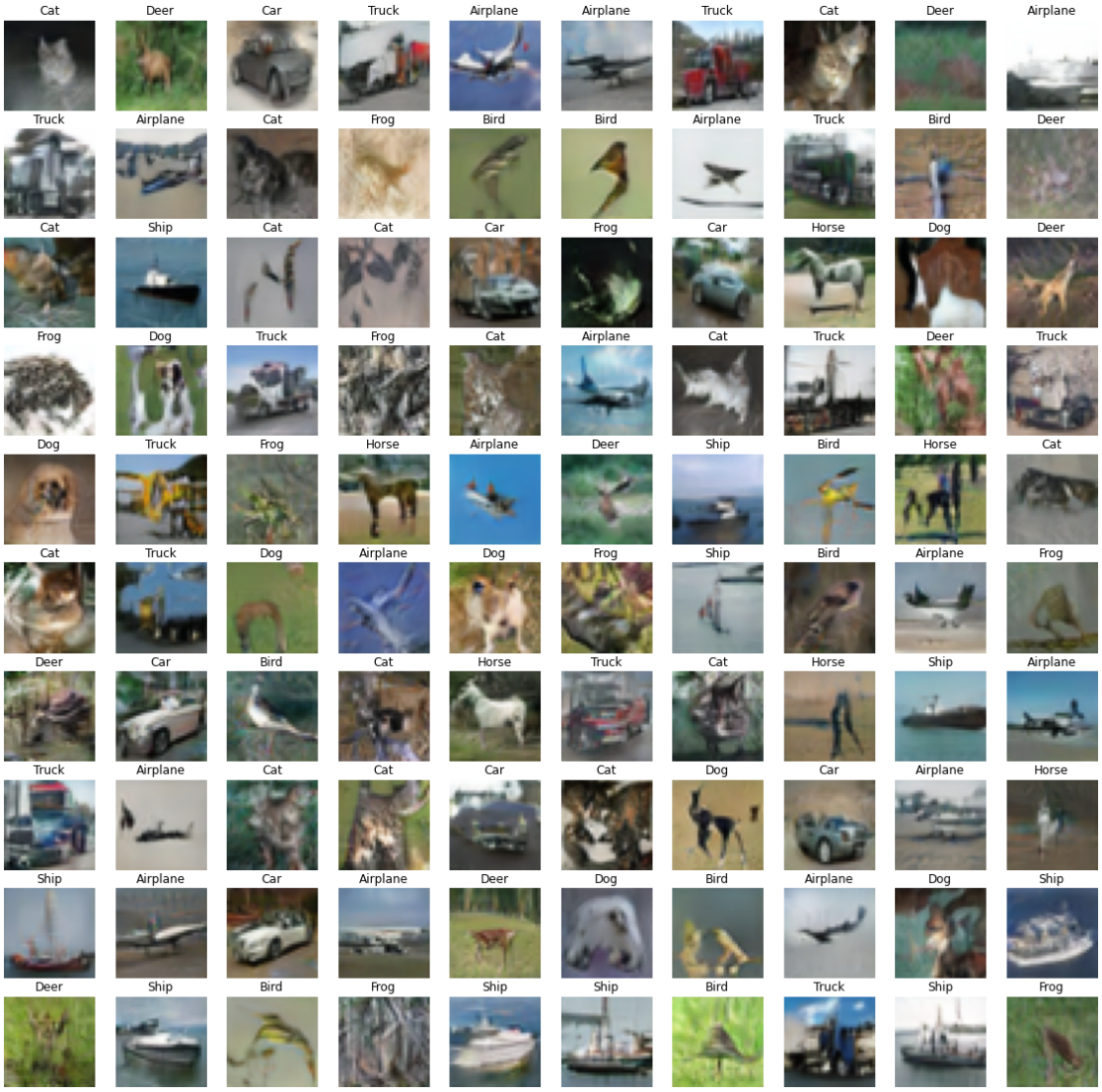
Example of the truncation trick (Credit: Brock, Donahue, and Simonyan, "Large Scale GAN Training for High Fidelity Natural Image Synthesis.")

```
import numpy as np
from scipy.stats import truncnorm
# https://github.com/ajbrock/BigGAN-PyTorch/blob/7b65e82d058bfe035fc4e
def truncated_z_sample(batch_size, z_dim, truncation=0.5, seed=None)
    state = None if seed is None else np.random.RandomState(seed)
    values = truncnorm.rvs(-2, 2, size=(batch_size, z_dim), random_s
    return truncation * values
```



Truncation = 0.9; Images remain very diverse, but are less realistic

Truncation = 0.5; Images are much clearer, but as can be seen, the colors are less vibrant, signifying a loss in diversity.



Truncation = 0.8; More colorful images but less realistic images

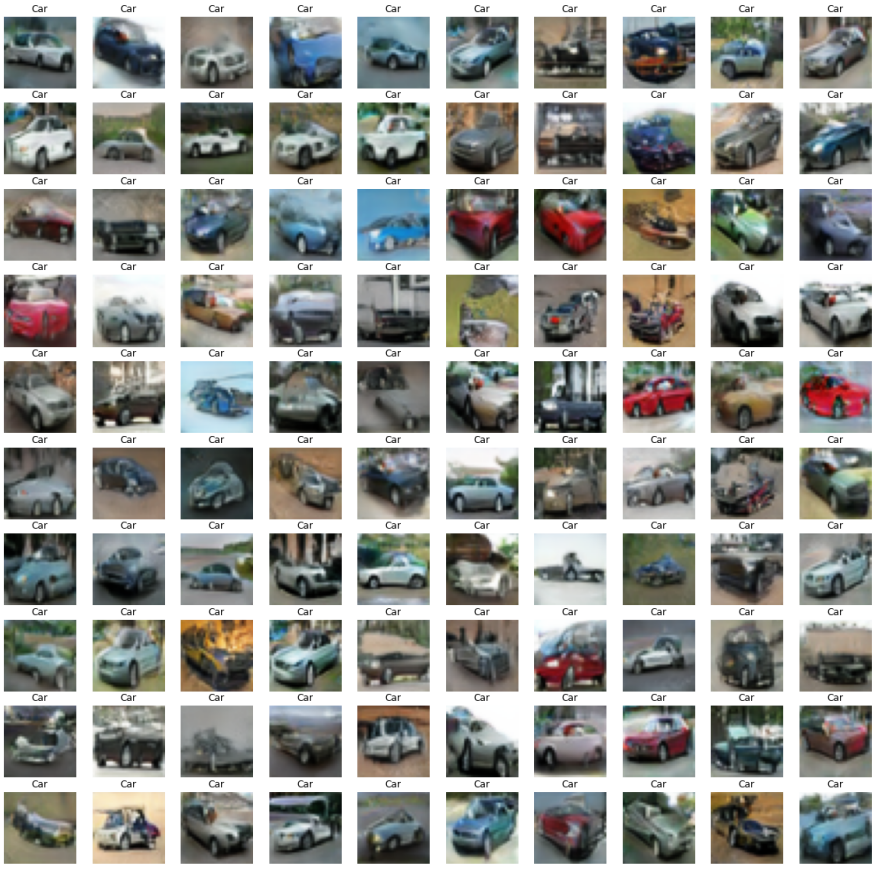
Ultimately, I find that choosing a truncation value between 0.8 and 0.9 results in more realistic looking images, whilst still retaining image diversity.

Final Set of Images Per Class

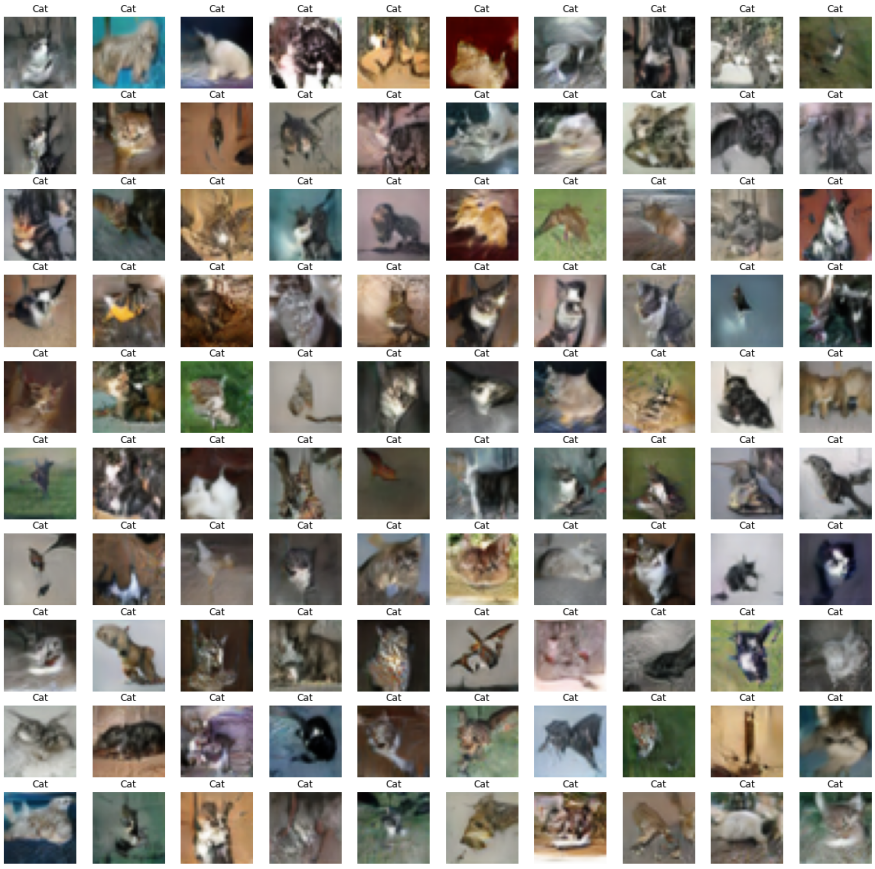
The following images were taken with truncation set as 0.85. The model used is the SNGAN with Projection and R1 Regularization, with 4 Discriminator Steps Per Generator Step.

Airplane

Bird

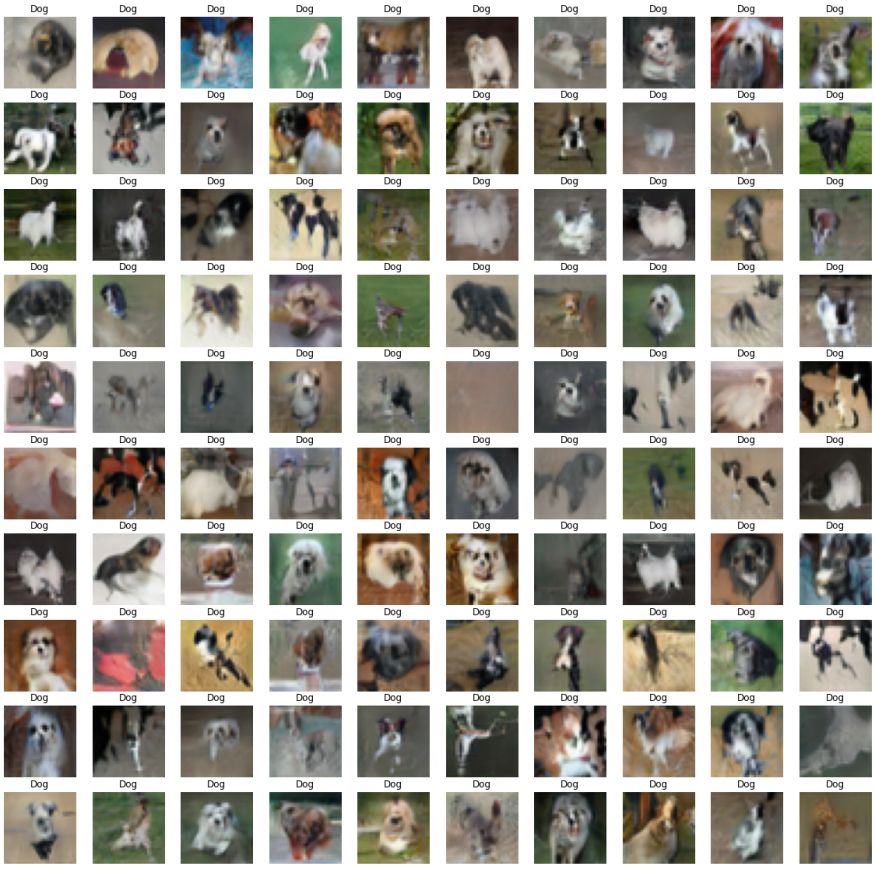


Car

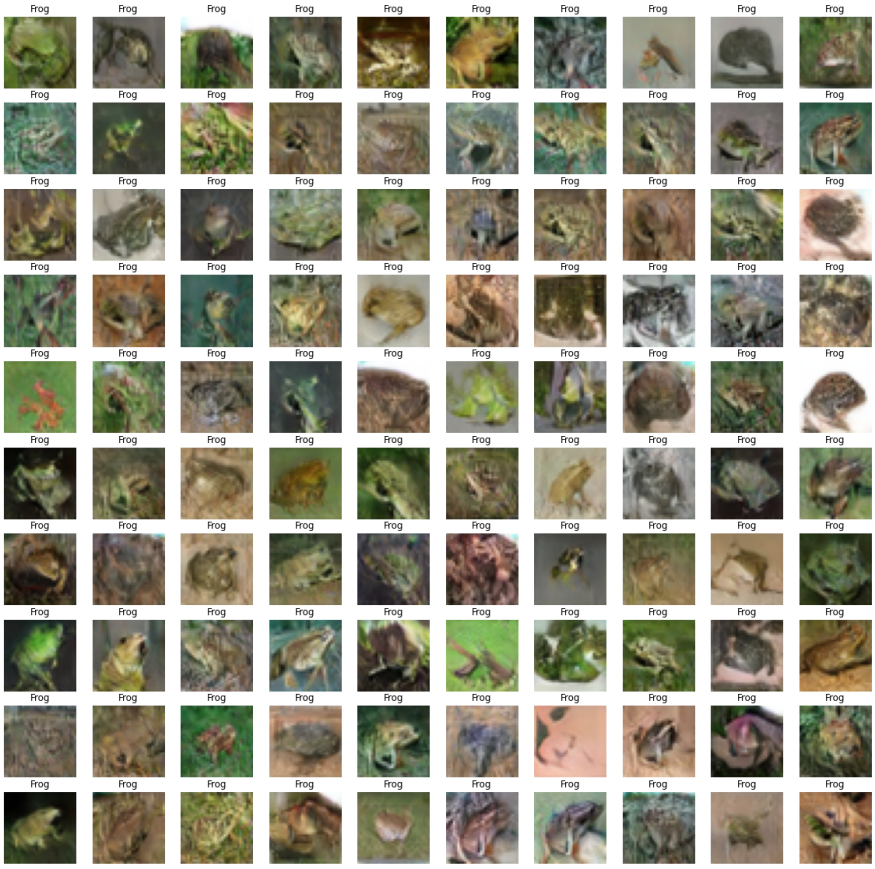


Cat

Deer

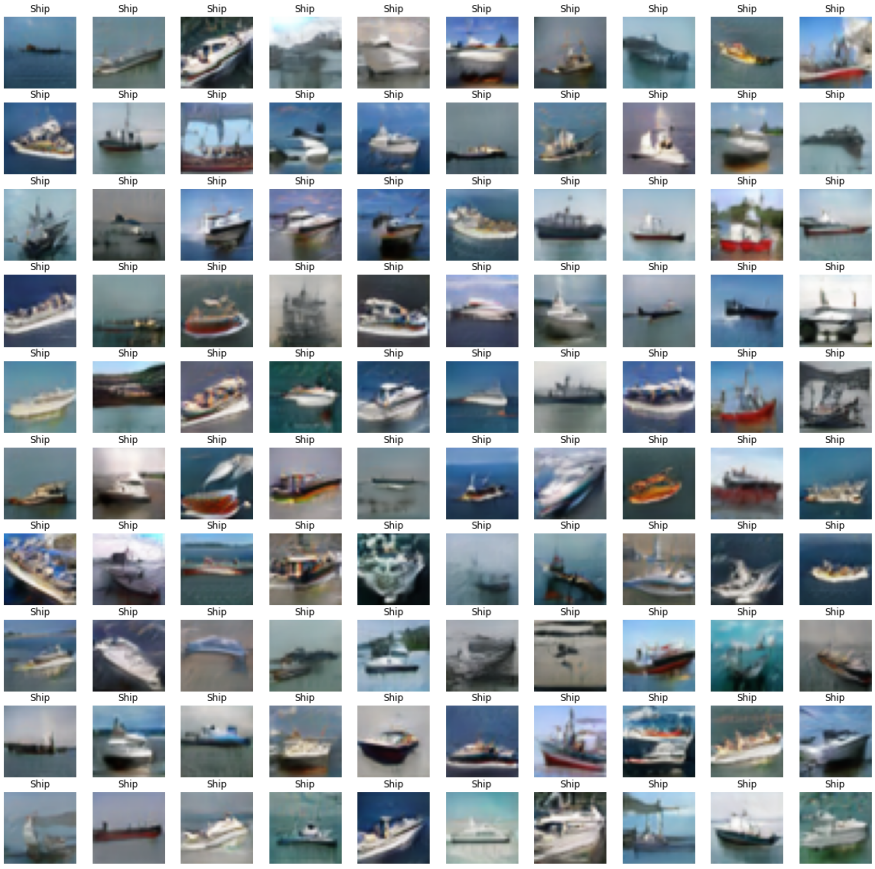


Dog

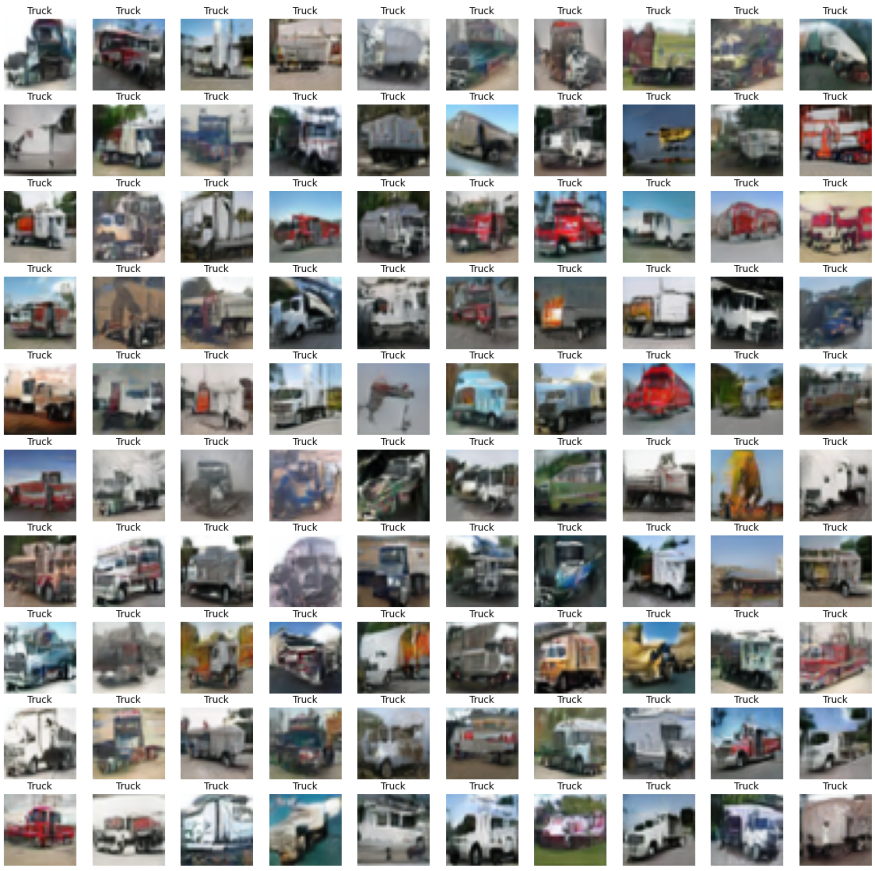


Frog

Horse



Ship



Truck

How Successful is the GAN?

How is the Image Quality?

The final metrics of the SNGAN are as follows:

Metric	Score
Inception Score	7.988
Frechet Inception Distance	19.988
Kernel Inception Distance	0.01144

Analyzing the performance of the model visually I make the following observations:

- Most images look fairly realistic, but some images have distorted features
    - For instance, some cars have very warped wheels, and the legs of some animals like horses are very thin
- The model output for some classes like those related to cats, birds, and frogs look a bit strange as the GAN seems to have learnt the texture of these animals, but has trouble forming the parts of the animals
- The model has a much easier time with vehicles.

In comparison with the optimal scores calculated during the EDA, the scores the model got are still far way, indicating there is definitely room for improvement in terms of image quality. Nevertheless, I am still satisfied by what the GAN has been able to output.

Does the GAN Suffer from Mode Collapse?

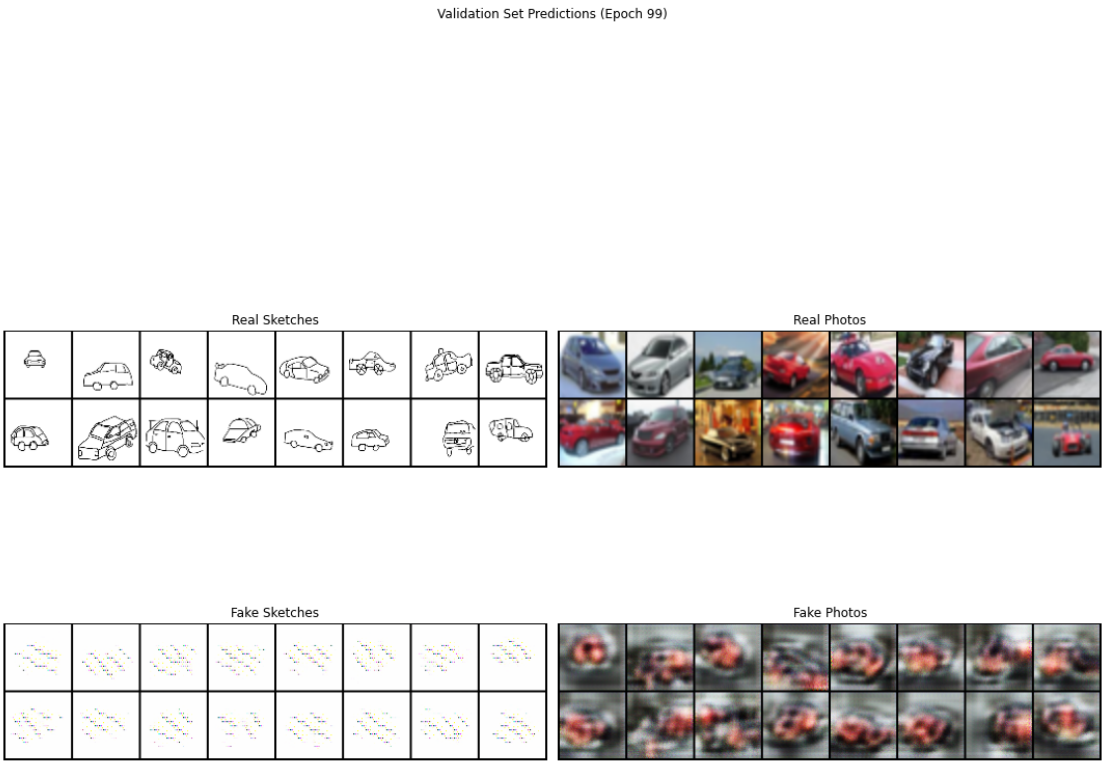
From analyzing all the images, I note that the model appears to be able to generate diverse images for each class. Nevertheless, there does appear to be some partial mode collapse for certain classes as some of the images do seem similar (albeit with slight differences). For instance, the car class contains several red cars in a similar orientation with some slight differences in the brightness of the image.

Two very similar looking cars generated by the model

Learning Points

Start Simple

Probably one of the biggest mistakes I made in the beginning was to be too ambitious in the beginning. I originally wanted to achieve my aim of a controllable GAN via using image translation GANs to convert sketches into images. I attempted to accomplish this via a CycleGAN architecture, and two datasets: a car sketch dataset and a CIFAR10 dataset.



Failed Attempt at a CycleGAN for Sketch to Image Generation

But being this ambitious early on turned out to be a bad idea as the task I was trying to do was simply too difficult, especially given the

limited data I had available. Later on, with research, I discovered that such a task was quite difficult, and so I moved on to trying to control my GAN output some other way via conditional generation.

Be Detail Oriented

Another mistake I made early on was not paying close enough to certain experimental details, such as the reporting of metrics, which resulted in me having to redo some of my experiments as a result. For example, my initial implementation of the FID score metric turned out to be flawed, resulting in me needing to reimplement it and rerun all my experiments. As a result, there were some experiments I wanted to try that could not be done due to a lack of time.

Next Steps

Moving on from this experiment, there are some parts I wish to improve upon in future experiments, that I am unable to do right now due to a lack of time.

Improving the GAN Architecture

While the GAN architecture I tried out is relatively recent, more recent improvements have come out. For instance, as mentioned in the evaluation of the images, it seems that the GAN would benefit from the addition of Self Attention modules, ala, SAGAN or BigGAN, as the location of some features in the images can be oddly placed.

Experimenting with Adaptive Discriminator Augmentation

Experimenting with DiffAugment did not improve the performance of my GAN. One reason might be that the data augmentation was simply too strong in the beginning, hampering the stability of training. Given more time, I would have liked to experiment with Adaptive Discriminator Augmentation, where the strength of the data augmentation is modified to ensure that the data augmentation is never too strong nor too weak.

Making the GAN Controllable

As mentioned in my initial problem statement, my goal is to eventually make the GAN fully controllable. As of right now, I could already attempt to control some aspect of the GAN output via latent space algebra, but I hope to eventually try other techniques to be able to directly influence the features of the final GAN (e.g. color of the car, etc)

Created with ❤️ on Weights & Biases.

https://wandb.ai/tiencheng/DELE_CA2_GAN/reports/Building-a-Conditional-GAN-for-CIFAR-10--VmlldzoxNTIyODEy