1) Ghazal Kaviani

2.1) The loss that should be used for training Standard Autoencoders is the MSE (between input & output).

In some cases, where the input data is between [0, 1], binary cross-entropy loss is also acceptable.

2.2) Vanilla autoencoders don't produce a distribution in the latent/embedding space, rather just a vector, usually without meaningfull entity explanation. Therefore, when using that tipe of autoencoder, one will only get that from said latent representation, over a whole distribution that VAE encoders are able to generate, thus giving sampling capacity.

Nonetheless, if one is <u>not</u> interested in the samples that can be obtained from the latent space, and rather just shrink all the input information in a smaller representation for other uses, like file compression, denoising, feature extraction, etc., then vanilla autoencoders are sufficient (thus, still usefull).

2.3) $D_{KL}(p(x)||q(x)) = E_{x\sim p(x)}[\log \frac{p(x)}{q(x)}]$

I know that the entropy of a distribution is

$$H(P) = -\mathbb{E}_{x\sim p}\left(\log(P(x))\right) \quad \text{(I)}$$

and that the entropy of a joint distribution is

$$H(P,Q) = -\mathbb{E}_{x\sim p}\left(\log(Q(x))\right) \quad \text{(II)}$$

If we measure the distance between those distributions (P, Q) assuming that entropy is a good indicator, we can just take the difference of (I) & (II)

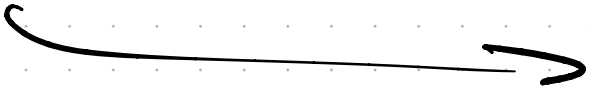
$$H(P,Q) - H(P) = -\mathbb{E}_{x\sim p}\left(\log(Q(x))\right) - \left[-\mathbb{E}_{x\sim p}\left(\log(P(x))\right)\right]$$

$$= -\mathbb{E}_{x\sim p}\left(\log(Q(x))\right) + \left[\mathbb{E}_{x\sim p}\left(\log(P(x))\right)\right]$$

$$= \mathbb{E}_{x\sim p}\left(\log(P(x))\right) - \mathbb{E}_{x\sim p}\left(\log(Q(x))\right)$$

$$= \mathbb{E}_{x\sim p}\left(\log(P(x)) - \log(Q(x))\right) = \boxed{\mathbb{E}_{x\sim p}\left(\log\left(P(x)/Q(x)\right)\right)}$$

which is the KL divergence



2.3 continued)

if we're trying to draw from a normal distribution:

$P = f_\alpha \sim \mathcal{N}(\mu_\alpha, \sigma_\alpha)$; $Q = g_\beta \sim \mathcal{N}(\mu_\beta, \sigma_\beta)$

then $D_{KL}(P(x) \| Q(x)) = D_{KL}(f_\alpha \| g_\beta) =$

$= \mathbb{E}_{x\sim p}\left(\log\left(\frac{f_\alpha(x)}{g_\beta(x)}\right)\right) = \mathbb{E}_{x\sim p}\left(\log\left(f_\alpha(x) - g_\beta(x)\right)\right)$

$= \mathbb{E}_{x\sim p}\left(\log\left(\frac{1}{\sigma_\alpha\sqrt{2\pi}} \cdot e^{-\frac{1}{2}\left(\frac{x-\mu_\alpha}{\sigma_\alpha}\right)^2} - \frac{1}{\sigma_\beta\sqrt{2\pi}} \cdot e^{-\frac{1}{2}\left(\frac{x-\mu_\beta}{\sigma_\beta}\right)^2}\right)\right)$

$= \mathbb{E}_{x\sim p}\left(\log(\sigma_\alpha) - \frac{1}{2}\log(2\pi) - \frac{1}{2}\log\left(\frac{x-\mu_\alpha}{\sigma_\alpha}\right) + \frac{1}{2}\log(x^2)\right.$

$\left. + \log(\sigma_\beta) + \frac{1}{2}\log(2\pi) + \frac{1}{2}\log\left(\frac{x-\mu_\beta}{\sigma_\beta}\right) - \frac{1}{2}\log(x^2)\right)$

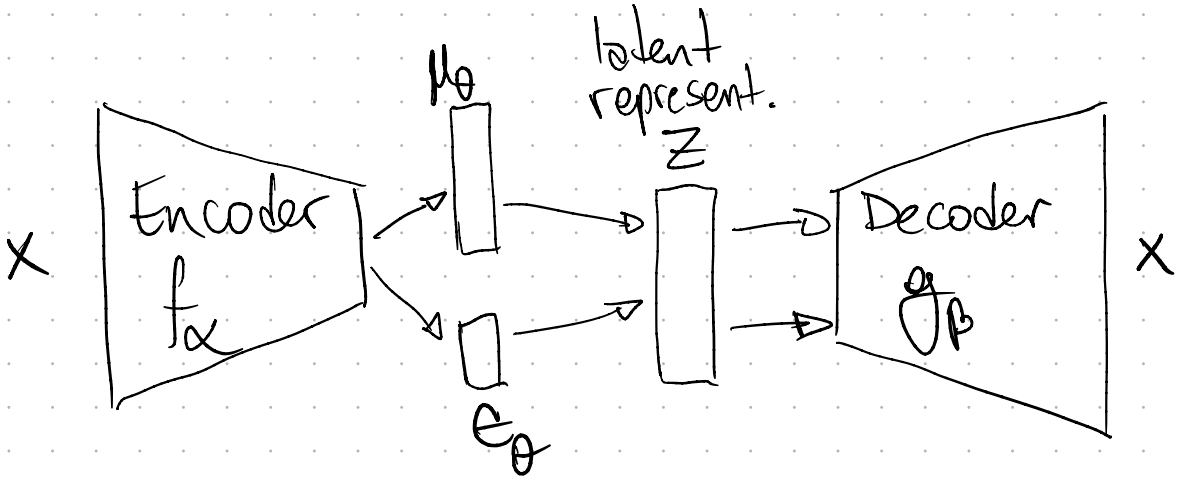
$= \mathbb{E}_{x\sim p}\left(\log(\sigma_\alpha) + \log(\sigma_\beta) - \frac{1}{2}\log\left(\frac{x-\mu_\alpha}{\sigma_\alpha}\right) + \frac{1}{2}\log\left(\frac{x-\mu_\beta}{\sigma_\beta}\right)\right)$

$$= \frac{(\mu_\alpha - \mu_\beta)^2}{\sigma_\alpha^2 \sigma_\beta^2} + \frac{1}{2}\left(\frac{\sigma_\alpha^2}{\sigma_\beta^2} - 1 - \log\left(\sigma_\alpha^2/\sigma_\beta^2\right)\right) = D_{KL}(f_\alpha \| g_\beta)$$

2.4) Gaussian reparametrization trick:

$$q(x_{t-1} \mid x_t, x_0) = \mathcal{N}(x_{t-1}; \mu_q(t), \Sigma_q(t))$$

$$\mu_q = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}} \right) \epsilon_\theta(x_t, t), \quad \Sigma \text{ assumed known}$$



2.5)

$$ELBO = E_{z\sim q_\phi(z|x)}[\log p_\theta(x,z) - \log q_\phi(z|x)]$$

Show that $\log(p_\theta(x)) \geq ELBO$.

For cleanliness, we drop θ and ϕ, since they will be learned and don't play a role here. Therefore:

Expanding ELBO:

$$\mathbb{E}_{z\sim q(z|x)}\left(\log(p(x,z)) - \log(q(z|x))\right)$$

$p(x,z) = p(x|z)\cdot p(z)$ from Bayes

$$= \mathbb{E}_{z\sim q(z|x)} \log\left(\frac{p(x|z)\cdot p(z)}{q(z|x)}\right) \quad (I)$$

Taking $\log(p(x))$ requires knowledge of q:

$$\log(p(x)) = \underbrace{\mathbb{E}_q\left(\log\left(\frac{p(x|z)\cdot p(z)}{q(z|x)}\right)\right)}_{(I) = ELBO} + \underbrace{D_{KL}(q(z|x)\,\|\,p(z|x))}_{\text{positive number or } 0}$$

$\therefore \log(p_\theta(x)) \geq ELBO$ ■

2.6)

$$ELBO = E_{z\sim q_\phi(z|x)}[\log p_\theta(x|z)] - E_{z\sim q_\phi(z|x)}[\log \frac{q_\phi(z|x)}{p_\theta(z)}]$$

(Also dropping θ & ϕ)

From 2.5: $ELBO = \mathbb{E}_{z\sim q(z|x)}\left(\log(p(x,z)) - \log(q(z|x))\right)$

$p(x,z) = p(x|z)\cdot p(z)$ from Bayes

$$\therefore ELBO = \mathbb{E}_{z\sim q(z|x)} \log\left(\frac{p(x|z)\cdot p(z)}{q(z|x)}\right) \quad (I)$$

manipulating (I):

$$\mathbb{E}_{z\sim q(z|x)} = \left(\log\left(\frac{p(x|z)\cdot p(z)}{q(z|x)}\cdot\frac{p(z)}{p(z)}\right)\right) = \mathbb{E}_{z\sim q}\left(\log\left(\frac{p(x|z)\cdot p(z)}{q(z|x)}\right) + \log p(z) - \log p(z)\right)$$

$$= \mathbb{E}_{z\sim q(z|x)}\left(\log(p(x|z)\cdot p(z)) + \log p(z) - \log(q(z|x)) - \log p(z)\right)$$

$$= \mathbb{E}_{z\sim q(z|x)}\left(\log(p(x|z)) + \log p(z) + \underbrace{\log p(z) - \log(q(z|x))}_{= -\log\left(\frac{q(z|x)}{p(z)}\right)} - \log p(z)\right)$$

$$= \mathbb{E}_{z\sim q(z|x)}\left(\log(p(x|z)) + \log p(z) - \log\left(\frac{q(z|x)}{p(z)}\right) - \log p(z)\right)$$

$$= \mathbb{E}_{z\sim q(z|x)}\left(\log(p(x|z))\right) - \mathbb{E}_{z\sim q(z|x)}\left(\log\left(\frac{q(z|x)}{p(z)}\right)\right) \quad \blacksquare$$

3.1) The two-player game played by a GAN is rather simple: The Generator wants to generate samples so good that they could fool the discriminator, which, in its turn, wants to always be able to tell real vs fake apart.

In this case the Nash equilibrium is unclear even locally. Instead, it is searched in the parameter space given some assumptions as presented by Goodfellow, et-al in Generative Adversarial Nets (NeurIPS, 2014) ("the GANs paper").

3.2)

x = reg
y = monet

- The role of:
  - G is to generate regular images that look like Monet paintings
  - D_x is to tell actual real images from images generated by G
  - F is to generate Monet paintings that look like regular images
  - D_y is to tell actual Monet paintings from the ones generated by F
- There are 2 two-player games: One for G vs. D_x and one for F vs. D_y.
- there is no obvious Nash equilibria.

3.3)

- The cycle-consistency loss is needed to ensure that the mapping from a domain to another, and then back (full cycle: out from a domain and back to it after 2 mappings/transfers) is consistent to the input.

that is, $F(G(x)) = x$ & $G(F(x)) = x$.

- If not enforced, the final transformation might lose information and the final output could be significantly different than the input.
- I actually call F & G as F & G in the previous question, but regardless, they are the mappings/transformations from one domain to another (real images and Monet paintings).

CS 4644/7643: Deep Learning
Spring 2024
HW4 Solutions

ARTHUR SCAQUETTI DO NASCIMENTO

Due: 11:59pm, April 4, 2024

4.1.1) (Denoising Diffusion Probabilistic Models)

- **Key contributions:** This work proposes a novel technique for generative modeling. The process is essentially training a denoising autoencoder to perform Markov Chain Monte Carlo sampling, which is completely different from other generative models like GANs and VAEs. To the best of my knowledge this is the paper that introduced diffusion models as an option for generative AI.

- **Strengths:**

  - This paper introduced one of the most powerful techniques for denoising diffudion models. The mere fact that it had beaten all benchmarks for a relatively simple dataset (CIFAR10) is already a strength.
  - This is one of the few works in ML/AI where the authors are very explicit about why they came up with that frameworks, and how they did it backed up by a lot of math. As opposed to papers that achieve a phenomenal performance in whatever task they do, but at the end you don't get why it works they way it does; works where they either over rely on an assumption or over engineer some blackbox.
  - If my thought process is correct, they introduced diffusion models to genAI. This is arguably the highest strenght, given the results we see from Midjourney, DALL-E, etc.

- **Weaknesses:**

  - This work only displays image generation, while it could test and showcase the generation capabilities on different domains. Specifically, I expected/wanted to see policy or trajectory generation, especially given that the most senior author is Pieter Abbeel.
  - A light weakness is the heavy language. Even for a NeurIPS paper, this is a very dense work to read. Even though there is a lot to explain in a limited amount of pages, I think that the authors could have done a better job in moving content around, resizing and rephrasing or even reducing the scope of this work for the NeurIPS submission, and publish the complete, longer version in a journal, with more room for writing.

4.2.1) (Denoising Diffusion Probabilistic Models) Personal takeaways:
The first thing I takeaway from this paper is that I should actually read a paper that has some sort of "hype" regardless of the heavy language and that I take over 15min to understand even the abstract + conclusions. Not a very pleasant takeaway, but that is something I am incorporating in my research style.
Another is that diffusion models are not just something to generate images from text. That said, I have a better understanding of one of my labmate's work, and wonder how this could be incorporated to model-based RL to generate policies for a POMDP. That is something I might be interested on doing in the upcoming months.

4.1.2) (High-Resolution Image Synthesis with Latent Diffusion Models)

- **Key contributions:** In contrast to DDPM, where VAEs where completely set aside, this paper puts together a smart architecture for combining diffusion models with VAEs (taking advantage of the dimensionality reduction in the latent space), which, combined with transformers, was a huge leap to achieve those amazing results from text-to-image (now even text-to-video) generation.

- **Strengths:**

  - Just like the DDPM paper, this is work is a hallmark just by the fact that the authors achieve state-of-the-art for some set of scores in different tasks. While some might not see this as an academic strength, I do and my point for that is that novel research is nothing but small increments of knowledge on major building blocks.
  - The authors introduce a very clever architecture (arguably overengineered, but I don't think this is less of a merit) that drastically decreases computational resource consumption for tasks as complex as generating high quality images.
  - This work introduces the possibility of many modalities of conditioning, including text and semantics. I am not sure if this is the first work that did this.

- **Weaknesses:**

  - Although they did a good job with the LDMs, they did not achieve high quality for details, such as specific edges and texture.
  - This is yet another paper that "just" puts two things together, achieve good results and just show the math that is already in the literature. They don't back their approach with math, as the DDPM paper did. They are basically working with blackboxes that happen to work well for those tasks with enough tunning.

4.2.2) (High-Resolution Image Synthesis with Latent Diffusion Models) Personal takeaways: Even though I listed putting two things and making them work as a weakness of this paper, a personal takeaway is that maybe it is ok to just do that. Maybe the math can be revisited later and more conclusions could be drawn? I am saying this because they did pave the road (at least to some extent) to SORA, so maybe showing two black-boxes together and show a proof of concept is enough for the progress of science as we know it. That is a more philosophical takeaway, but epistemology has been something that has been haunting me.

In [ ]:
```
# from google.colab import drive
# drive.mount('/content/drive')
```

change to whatever your path is to the part1-GANs folder

In [ ]:
```
# %cd "drive/MyDrive/homework4/part1-GANs"
```

In [ ]:
```
%load_ext autoreload
%autoreload 2
```

Homework 4: Part 1 - Generative Adversarial Networks (GANs) [9 pts]

What is a GAN?

In 2014, Goodfellow et al. presented a method for training generative models called Generative Adversarial Networks (GANs for short).

In a GAN, there are two different neural networks. Our first network is a traditional classification network, called the **discriminator**. We will train the discriminator to take images and classify them as being real (belonging to the training set) or fake (not present in the training set). Our other network, called the **generator**, will take random noise as input and transform it using a neural network to produce images. The goal of the generator is to fool the discriminator into thinking the images it produced are real.

We can think of this back and forth process of the generator (G) trying to fool the discriminator (D) and the discriminator trying to correctly classify real vs. fake as a minimax game:

$$\underset{G}{\text{minimize}}\ \underset{D}{\text{maximize}}\ \mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] + \mathbb{E}_{z\sim p(z)}\left[\log(1-D(G(z)))\right]$$

where $z \sim p(z)$ are the random noise samples, $G(z)$ are the generated images using the neural network generator G, and D is the output of the discriminator, specifying the probability of an input being real. In Goodfellow et al., they analyze this minimax game and show how it relates to minimizing the Jensen-Shannon divergence between the training data distribution and the generated samples from G.

To optimize this minimax game, we will alternate between taking gradient *descent* steps on the objective for G and gradient *ascent* steps on the objective for D:

1. update the **generator** (G) to minimize the probability of the **discriminator making the correct choice**.
2. update the **discriminator** (D) to maximize the probability of the **discriminator making the correct choice**.

While these updates are useful for analysis, they do not perform well in practice. Instead, we will use a different objective when we update the generator: maximize the

probability of the **discriminator making the incorrect choice**. This small change helps to allevaiate problems with the generator gradient vanishing when the discriminator is confident. This is the standard update used in most GAN papers and was used in the original paper from Goodfellow et al..

In this assignment, we will alternate the following updates:

1. Update the generator (G) to maximize the probability of the discriminator making the incorrect choice on generated data:

$$\underset{G}{\text{maximize}}\ \mathbb{E}_{z\sim p(z)}\left[\log D(G(z))\right]$$

2. Update the discriminator (D), to maximize the probability of the discriminator making the correct choice on real and generated data:

$$\underset{D}{\text{maximize}}\ \mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] + \mathbb{E}_{z\sim p(z)}\left[\log(1 - D(G(z)))\right]$$

In [ ]:
```
# Setup cell.
import numpy as np
import torch
import torch.nn as nn
from torch.nn import init
import torchvision
import torchvision.transforms as T
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler
import torchvision.datasets as dataset
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

import sys
sys.path.append("part1_GANS")

# # # import os
# # # # home_dir = os.path.dirname(os.path.abspath('hw4-part1.ipynb'))
# # # home_dir = '/home/anascimento7/CS7643/Assignments/hw4_code_student_
# # # print(home_dir)
# # # # gans_dir = os.path.join(home_dir, 'part1-GANs')
# # # sys.path.append(home_dir)
# # # print(home_dir)

from part1_GANs.gan_pytorch import preprocess_img, deprocess_img, rel_err

dtype = torch.cuda.FloatTensor if torch.cuda.is_available() else torch.Fl

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # Set default size of plots.
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'


def show_images(images):
    sqrtn = int(np.ceil(np.sqrt(images.shape[0])))
    fig = plt.figure(figsize=(sqrtn, sqrtn))
    gs = gridspec.GridSpec(sqrtn, sqrtn)
```

```
    gs.update(wspace=0.05, hspace=0.05)

    for i, img in enumerate(images):
        ax = plt.subplot(gs[i])
        plt.axis('off')
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.set_aspect('equal')
        # If the image is in the shape (3, 32, 32), use transpose to chan
        if img.shape[0] == 3:
            img = np.transpose(img, (1, 2, 0))
        plt.imshow(img)
        # plt.show()

    return
```

Dataset

We are going to be using SVHN (Street View and House Number Dataset)

Link: http://ufldl.stanford.edu/housenumbers/

In [ ]:
```
# NUM_TRAIN = 73257
# NUM_VAL = 26032

NOISE_DIM = 96
batch_size = 128
val_batch_size = 128

svhn_train = dataset.SVHN(
    'datasets/svhn',
    split="train",
    download=True,
    transform=T.ToTensor()
)
```

```
loader_train = DataLoader(
    svhn_train,
    batch_size=batch_size,
    sampler=ChunkSampler(len(svhn_train), 0)
)

print(len(svhn_train))

svhn_val = dataset.SVHN(
    'datasets/svhn',
    split="test",
    download=True,
    transform=T.ToTensor()
)

print(len(svhn_val))

loader_val = DataLoader(
    svhn_val,
    batch_size=val_batch_size,
    sampler=ChunkSampler(len(svhn_val), 0)
)

iterator = iter(loader_train)
imgs, labels = next(iterator)
print(imgs.shape)
# imgs = imgs.view(batch_size, 3072).numpy().squeeze()
show_images(imgs[0:99])
```

```
Using downloaded and verified file: datasets/svhn/train_32x32.mat
73257
Using downloaded and verified file: datasets/svhn/test_32x32.mat
26032
torch.Size([128, 3, 32, 32])
```

TODO: Random Noise [1pts]

Generate uniform noise from -1 to 1 with shape `[batch_size, dim]`.

Implement `sample_noise` in `gan_pytorch.py`.

Hint: use `torch.rand`.

Make sure noise is the correct shape and type:

In [ ]:
```
from part1_GANs.gan_pytorch import sample_noise
from part1_GANs.gan_pytorch import Flatten, Unflatten, initialize_weights

def test_sample_noise():
    batch_size = 3
    dim = 4
    torch.manual_seed(6476)
    z = sample_noise(batch_size, dim)
    np_z = z.cpu().numpy()
    assert np_z.shape == (batch_size, dim)
    assert torch.is_tensor(z)
    assert np.all(np_z >= -1.0) and np.all(np_z <= 1.0)
```

```
    assert np.any(np_z < 0.0) and np.any(np_z > 0.0)
    print('All tests passed!')

test_sample_noise()
```

```
All tests passed!
```

TODO: Discriminator [0.5 pts]

GAN is composed of a discriminator and generator. The discriminator differentiates between a real and fake image. Let us create the architecture for it. In `gan_pytorch.py`, fill in the architecture as part of the `nn.Sequential` constructor in the function below. All fully connected layers should include bias terms.

The recommendation for discriminator architecture is to have 1-2 Conv2D blocks, and a fully connected layer after that.

Recall that the Leaky ReLU nonlinearity computes $f(x) = \max(\alpha x, x)$ for some fixed constant α; for the LeakyReLU nonlinearities in the architecture above we set $\alpha = 0.01$.

The output of the discriminator should have shape `[batch_size, 1]`, and contain real numbers corresponding to the scores that each of the `batch_size` inputs is a real image.

In [ ]:
```
from part1_GANs.gan_pytorch import discriminator

def test_discriminator():
    model = discriminator()
    cur_count = count_params(model)
    print('Number of parameters in discriminator: ', cur_count)

test_discriminator()
```

```
Number of parameters in discriminator:  142529
```

TODO: Generator [0.5 pts]

Recommendation for the generator is to have some fully connected layer followed by 1-2 ConvTranspose2D (It is also known as a fractionally-strided convolution). Remeber to use activation functions such as ReLU, Leaky RelU, etc. Finally use nn.Tanh() to finally output between [-1, 1].

The output of a generator should be the image. Therefore the shape will be `[batch_size, 3, 32, 32]`.

In [ ]:
```
from part1_GANs.gan_pytorch import generator

def test_generator():
    model = generator()
    cur_count = count_params(model)
    print('Number of parameters in generator: ', cur_count)
```

```
test_generator()
```

```
Number of parameters in generator:  2119811
```

TODO: Implement the Discriminator and Generator Loss Function [3 pts]

This will be the Vanilla GAN loss function which involves the Binary Cross Entropy Loss. Fill the `bce_loss`, `discriminator_loss`, and `generator_loss`. The errors should be less than 5e-5.

In [ ]:
```
from part1_GANs.gan_pytorch import bce_loss, discriminator_loss, generato

# import os
# print(os.getcwd())

# answers = dict(np.load('../test_resources/gan-checks.npz')) #ORIGINAL
answers = dict(np.load('test_resources/gan-checks.npz'))

def test_discriminator_loss(logits_real, logits_fake, d_loss_true):
    d_loss = discriminator_loss(torch.Tensor(logits_real).type(dtype),
                                torch.Tensor(logits_fake).type(dtype)).cp
    # print(d_loss, d_loss_true)
    assert torch.allclose(torch.Tensor(d_loss_true), torch.Tensor(d_loss)
    print("Maximum error in d_loss: %g"%rel_error(d_loss_true, d_loss.num

def test_generator_loss():
    sample_logits = torch.tensor([0.0, 0.5, 0.4, 0.1])
    g_loss = generator_loss(torch.Tensor(sample_logits).type(dtype)).cpu(
    assert torch.allclose(torch.Tensor([0.581159]), torch.Tensor(g_loss),
    print("Maximum error in d_loss: %g"%rel_error(0.581159, g_loss.item()

def test_bce_loss():
    input = torch.tensor([0.4, 0.1, 0.2, 0.3])
    target = torch.tensor([0.3, 0.25, 0.25, 0.2])
    bc_loss = bce_loss(input, target).item()
    # print('BC LOSS', bc_loss)
    assert torch.allclose(torch.Tensor([0.7637264728546143]), torch.Tenso
    print("Maximum error in d_loss: %g"%rel_error(0.7637264728546143, bc_

test_discriminator_loss(
    answers['logits_real'],
    answers['logits_fake'],
    answers['d_loss_true']
)

test_generator_loss()

test_bce_loss()
```

```
Maximum error in d_loss: 3.97058e-09
Maximum error in d_loss: 3.76114e-09
Maximum error in d_loss: 0
```

In [ ]:
```
## All imports done in the test cells

from part1_GANs.gan_pytorch import sample_noise
```

```
from part1_GANs.gan_pytorch import Flatten, Unflatten, initialize_weights
from part1_GANs.gan_pytorch import discriminator
from part1_GANs.gan_pytorch import generator
from part1_GANs.gan_pytorch import bce_loss, discriminator_loss, generato
```

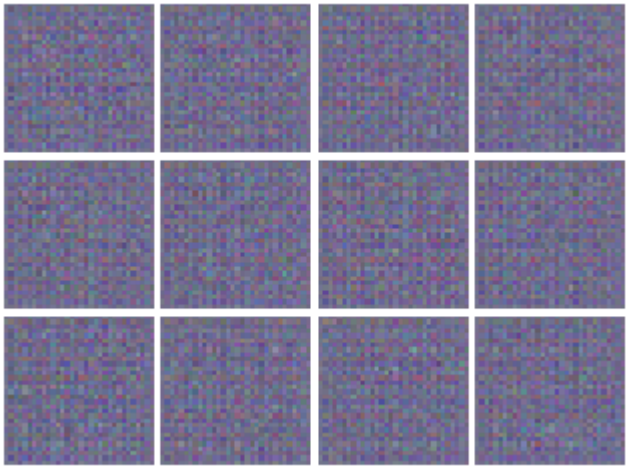
Now let's complete the GAN

Fill the training function `train_gan` and make sure to complete the `get_optimizer`. Adam is a good recommendation for the optimizer.

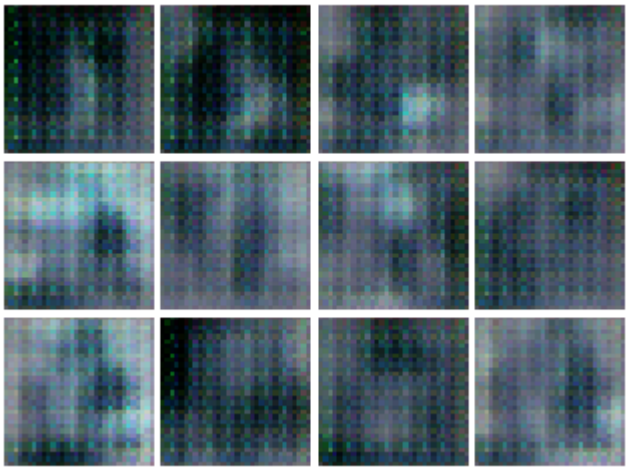
The top batch of images shown are the outputs from your generator. The bottom batch of images are the real inputs that the discriminator recieves along with your generated images.

In [ ]:
```
from part1_GANs.gan_pytorch import get_optimizer, train_gan

# Make the discriminator
D = discriminator().type(dtype)

# Make the generator
G = generator().type(dtype)

# Use the function you wrote earlier to get optimizers for the Discrimina
D_solver = get_optimizer(D)
G_solver = get_optimizer(G)

batch_size = 256

loader_train = DataLoader(
    svhn_train,
    batch_size=batch_size,
    sampler=ChunkSampler(len(svhn_train), 0)
)

# Run it!
images = train_gan(
    D,
    G,
    D_solver,
    G_solver,
    discriminator_loss,
    generator_loss,
    loader_train,
    batch_size = batch_size,
    noise_size=96,
    num_epochs=100
)
```

```
Iter: 0, D: 1.406, G:0.6583
```

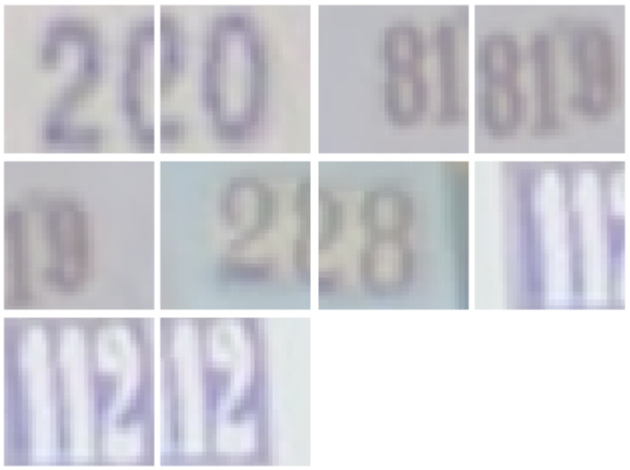


Iter: 250, D: 1.179, G:1.959



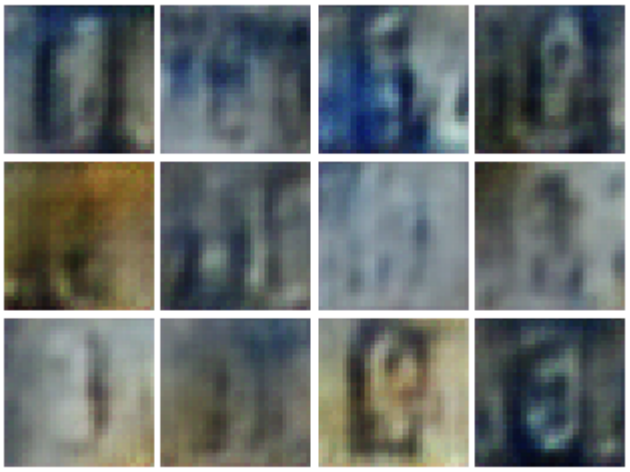
Iter: 500, D: 1.414, G:0.6468

Iter: 750, D: 1.377, G:0.807

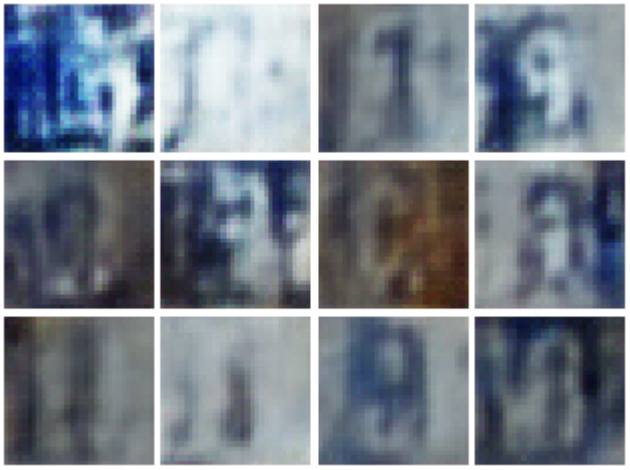


Iter: 1000, D: 1.349, G:0.7401

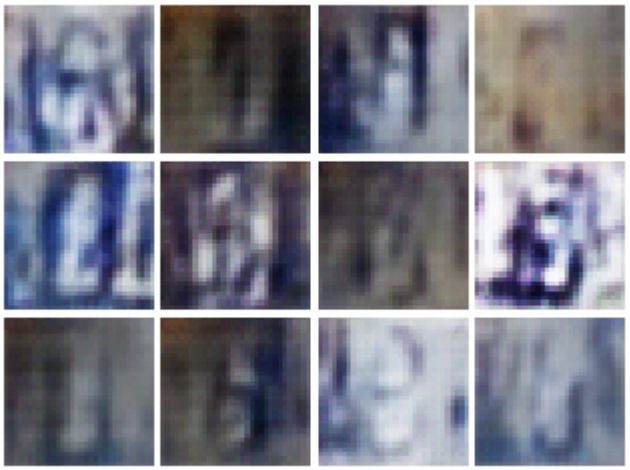
Iter: 1250, D: 1.363, G:0.6993



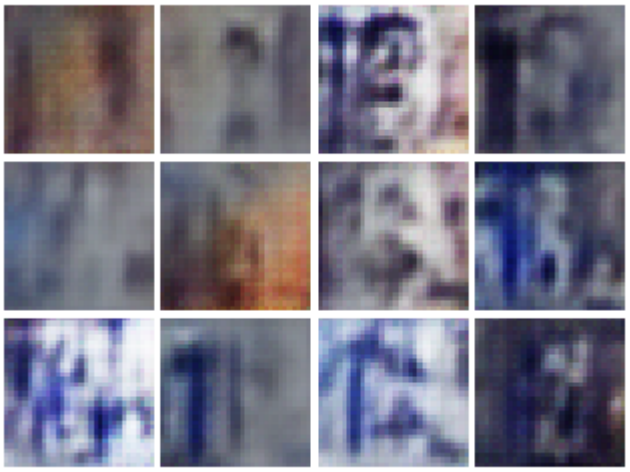
Iter: 1500, D: 1.376, G:0.6824



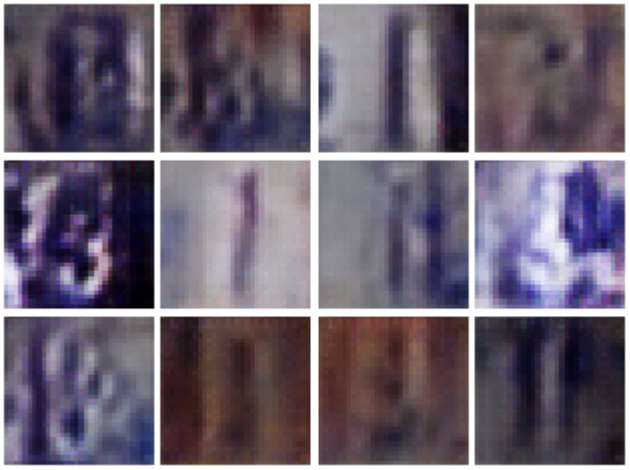
Iter: 1750, D: 1.355, G:0.7176



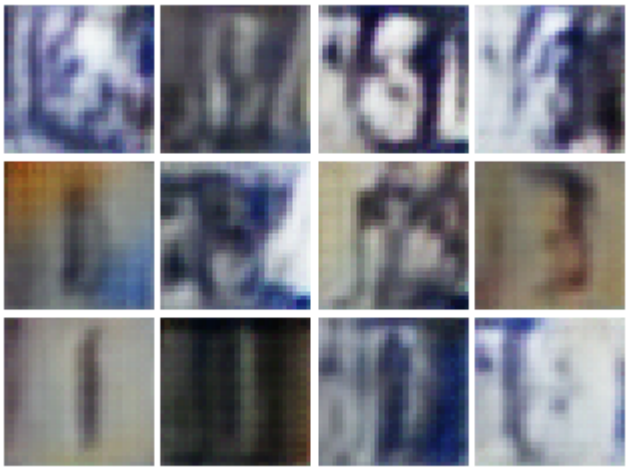
Iter: 2000, D: 1.378, G:0.7773



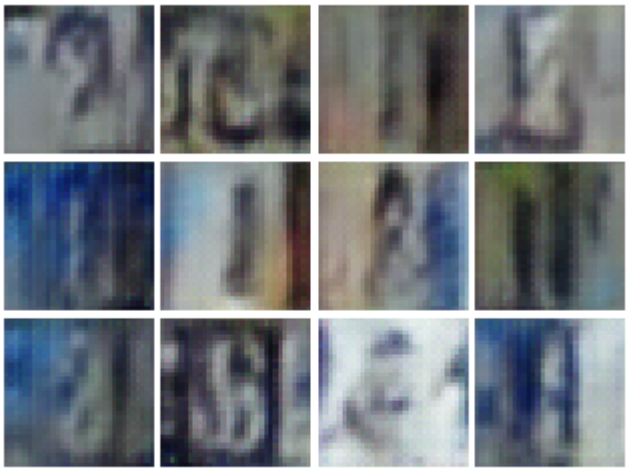
Iter: 2250, D: 1.371, G:0.8752



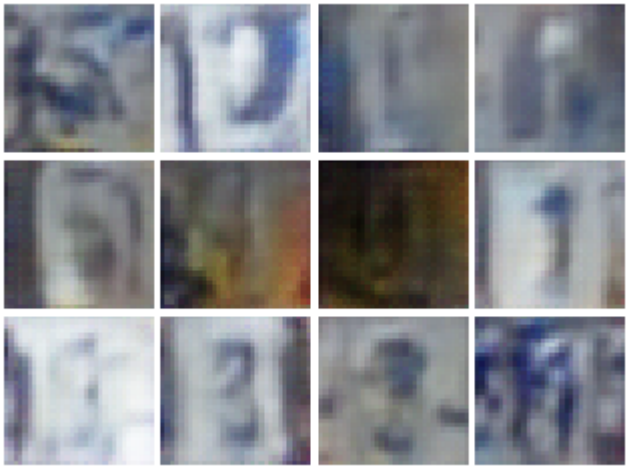
Iter: 2500, D: 1.327, G:0.8104



Iter: 2750, D: 1.399, G:0.8146

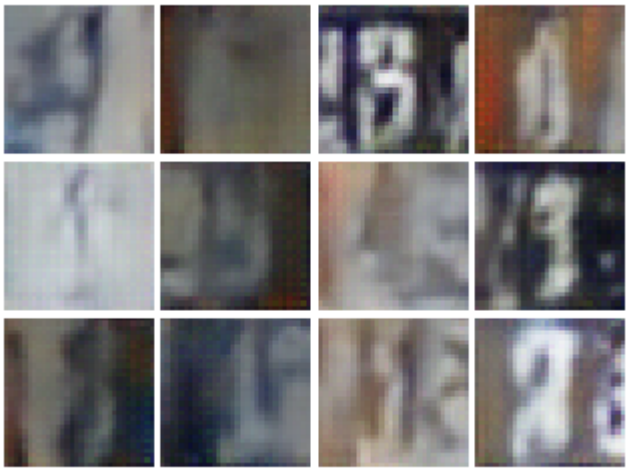


Iter: 3000, D: 1.361, G:0.7172



Iter: 3250, D: 1.344, G:0.8159

Iter: 3500, D: 1.338, G:0.6952

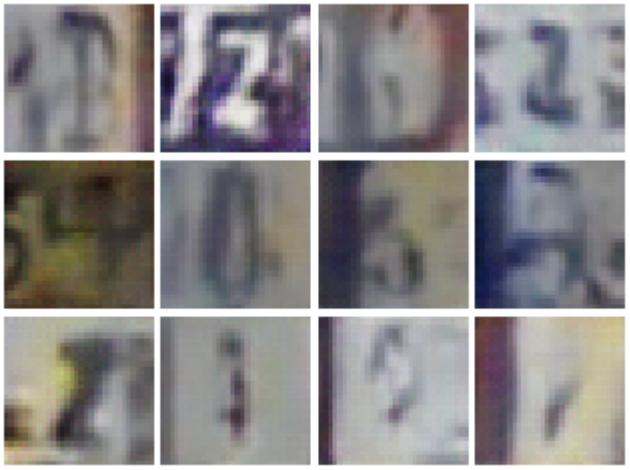


Iter: 3750, D: 1.286, G:0.8664

Iter: 4000, D: 1.324, G:0.8503

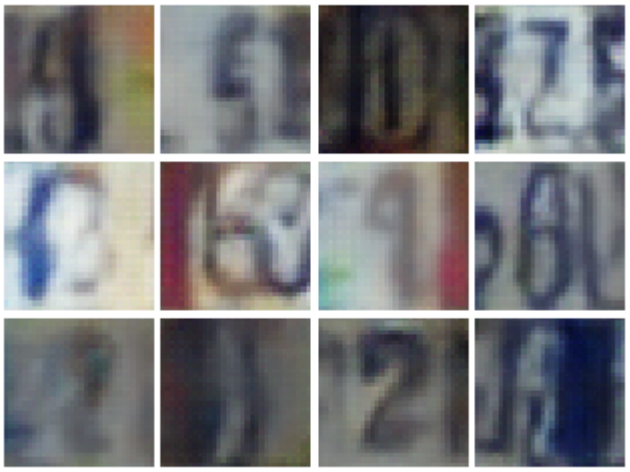
Iter: 4250, D: 1.377, G:0.8462

Iter: 4500, D: 1.312, G:0.85



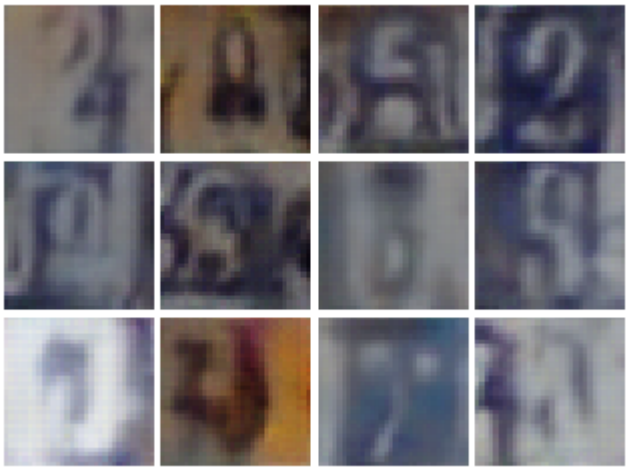
Iter: 4750, D: 1.344, G:0.7998

Iter: 5000, D: 1.361, G:0.915



Iter: 5250, D: 1.368, G:0.8965

Iter: 5500, D: 1.291, G:0.8306



Iter: 5750, D: 1.313, G:0.7721

Iter: 6000, D: 1.369, G:0.7824

Iter: 6250, D: 1.357, G:0.6901

Iter: 6500, D: 1.258, G:0.907

Iter: 6750, D: 1.332, G:0.8968

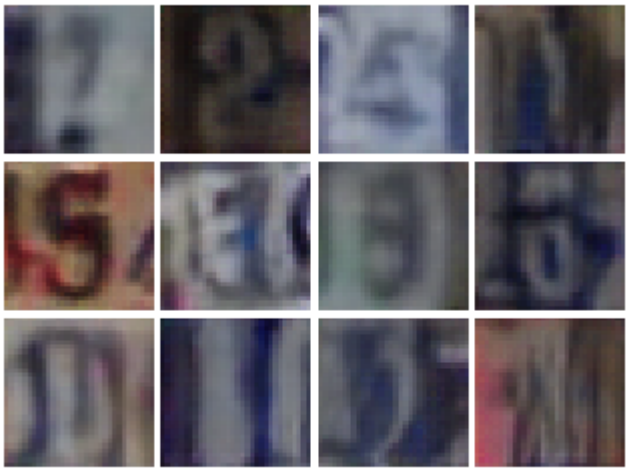
Iter: 7000, D: 1.293, G:0.8181

Iter: 7250, D: 1.264, G:0.8648

Iter: 7500, D: 1.275, G:0.7945

Iter: 7750, D: 1.27, G:0.7771

Iter: 8000, D: 1.271, G:0.8519



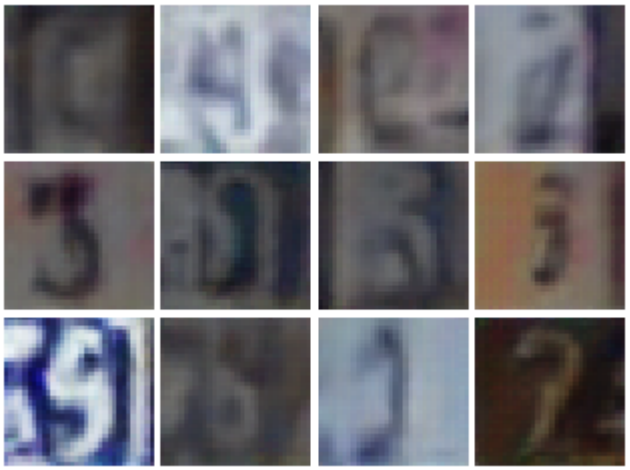
Iter: 8250, D: 1.318, G:0.799

Iter: 8500, D: 1.226, G:1.007

Iter: 8750, D: 1.298, G:0.81

Iter: 9000, D: 1.279, G:0.8775

Iter: 9250, D: 1.312, G:0.7987



Iter: 9500, D: 1.187, G:0.8791

Iter: 9750, D: 1.34, G:0.8476

Iter: 10000, D: 1.188, G:1.092

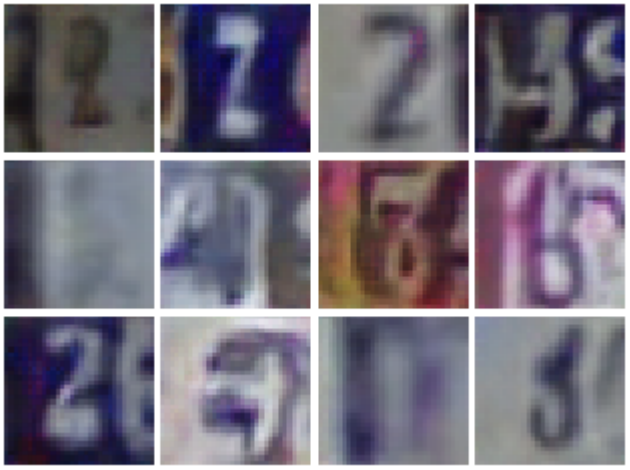
Iter: 10250, D: 1.329, G:0.7354

Iter: 10500, D: 1.148, G:0.9445

Iter: 10750, D: 1.235, G:0.7695

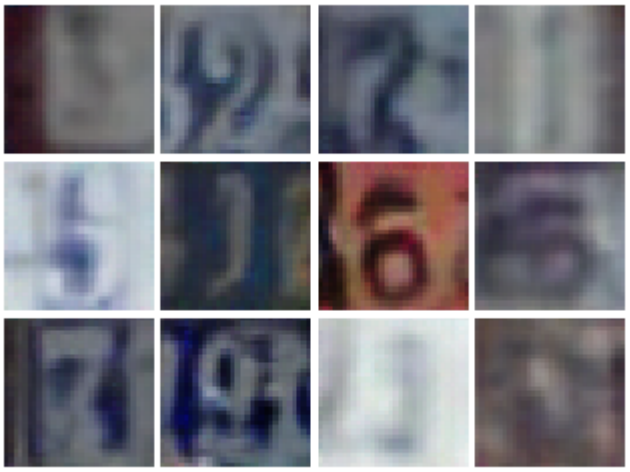
Iter: 11000, D: 1.216, G:1.365

Iter: 11250, D: 1.234, G:0.8879



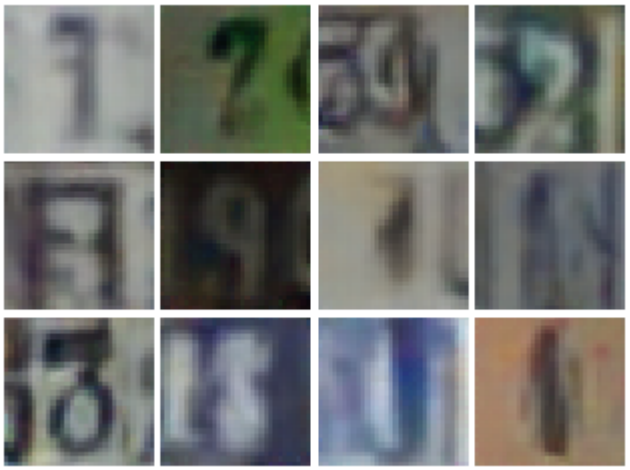
Iter: 11500, D: 1.167, G:0.9297

Iter: 11750, D: 1.243, G:0.9439



Iter: 12000, D: 1.269, G:0.9028

Iter: 12250, D: 1.241, G:0.8992

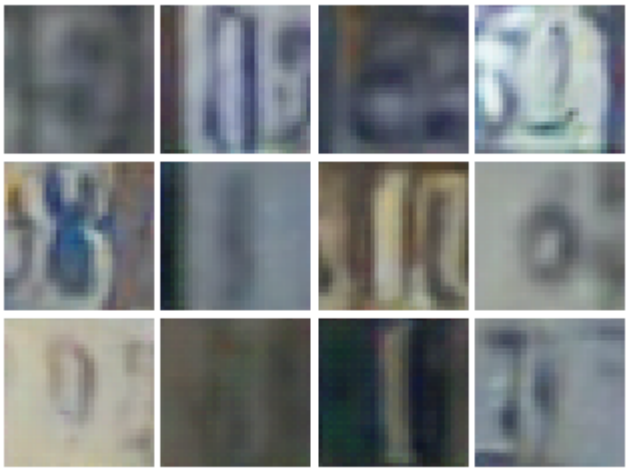


Iter: 12500, D: 1.177, G:1.036

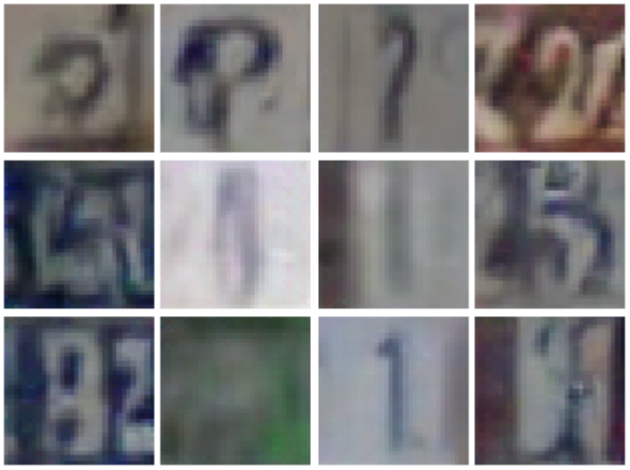
Iter: 12750, D: 1.282, G:0.9237

Iter: 13000, D: 1.316, G:0.8535

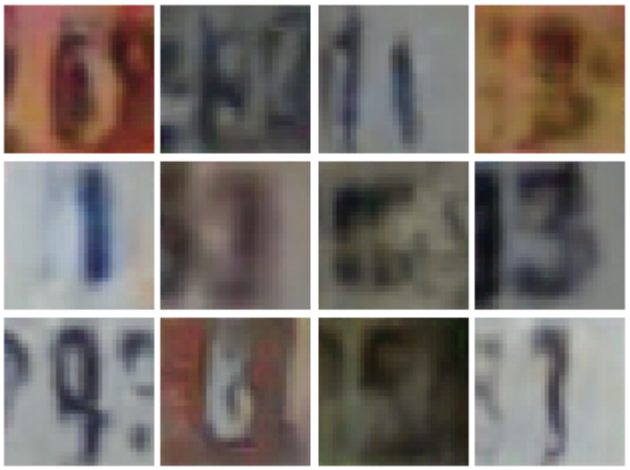
Iter: 13250, D: 1.317, G:0.9666



Iter: 13500, D: 1.279, G:0.879

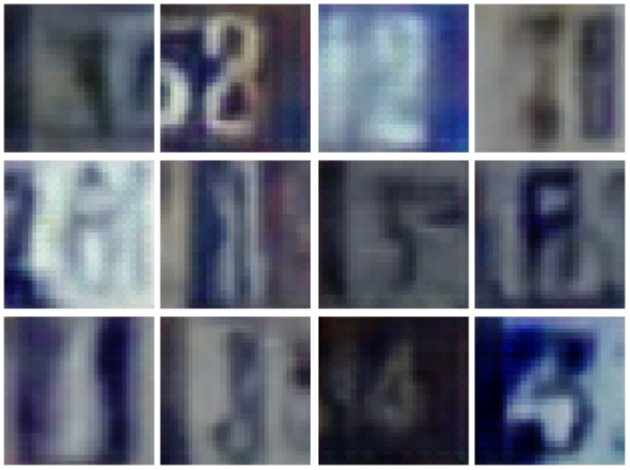


Iter: 13750, D: 1.234, G:0.8873



Iter: 14000, D: 1.16, G:0.9272

Iter: 14250, D: 1.438, G:0.8887



Iter: 14500, D: 1.233, G:0.9248

Iter: 14750, D: 1.19, G:0.8669

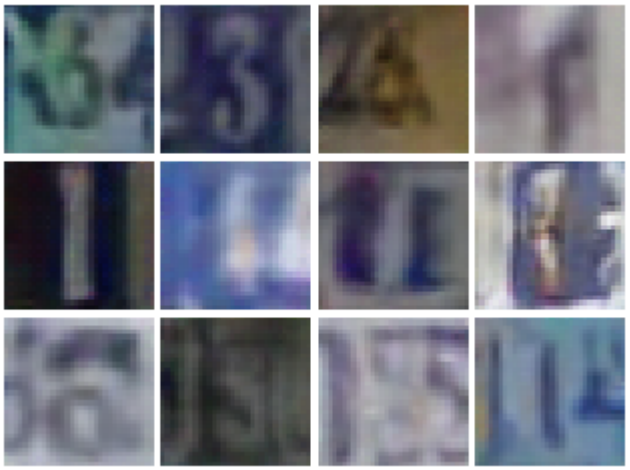
Iter: 15000, D: 1.304, G:1.002

```
Iter: 15250, D: 1.255, G:0.905
```

Iter: 15500, D: 1.204, G:1.011

Iter: 15750, D: 1.212, G:0.9114

Iter: 16000, D: 1.152, G:0.9947

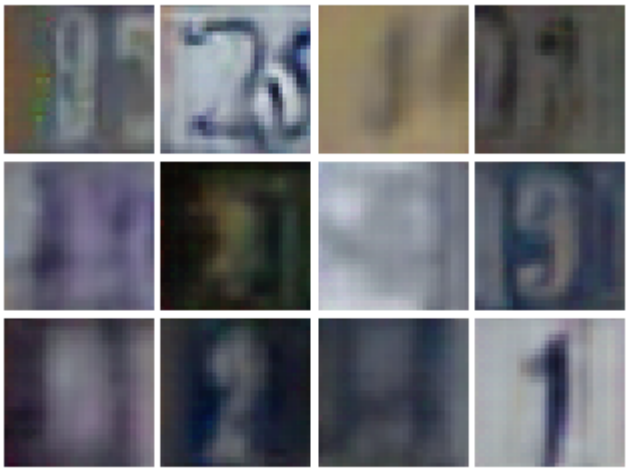


Iter: 16250, D: 1.211, G:1.014

Iter: 16500, D: 1.2, G:0.9297

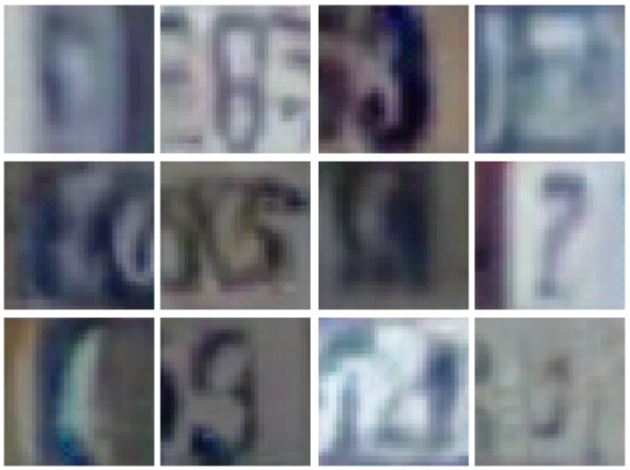
Iter: 16750, D: 1.21, G:0.9168

Iter: 17000, D: 1.255, G:0.9419

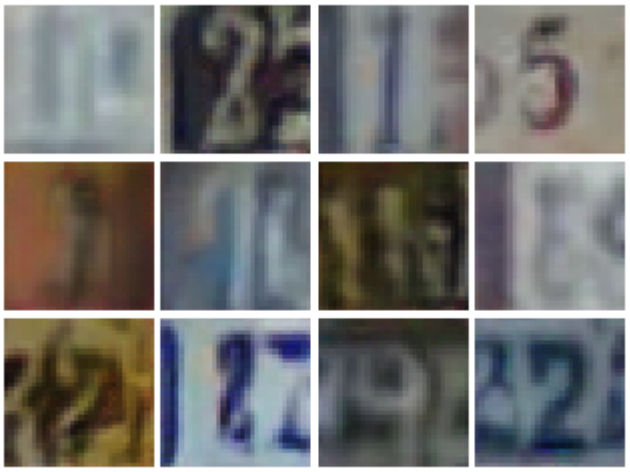


Iter: 17250, D: 1.279, G:0.8876

Iter: 17500, D: 1.141, G:1.02

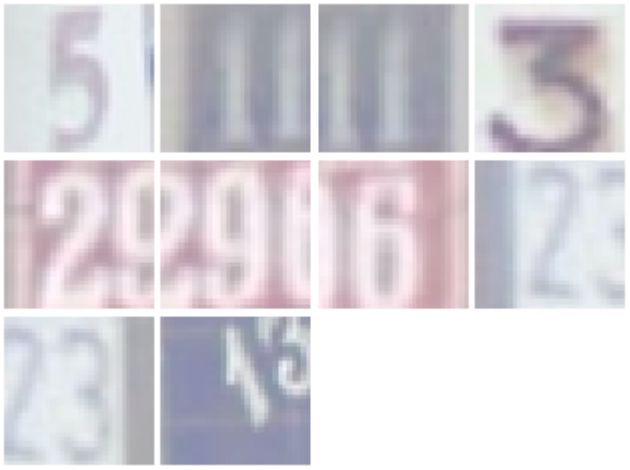


Iter: 17750, D: 1.208, G:0.9403



Iter: 18000, D: 1.183, G:0.9539

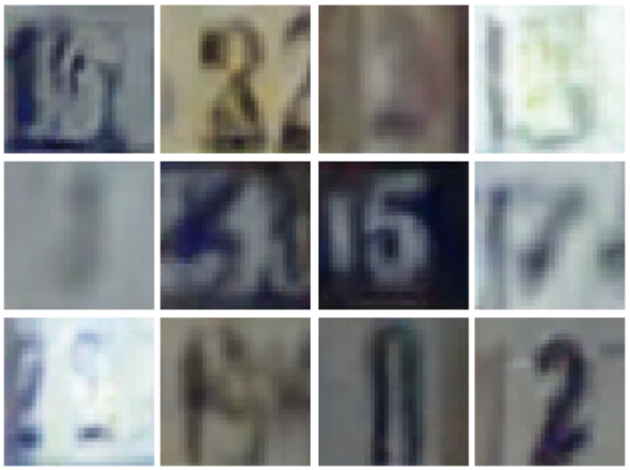
Iter: 18250, D: 1.166, G:2.503



Iter: 18500, D: 1.139, G:0.9401

Iter: 18750, D: 1.168, G:0.9265

Iter: 19000, D: 1.128, G:0.8735



Iter: 19250, D: 0.997, G:1.035

Iter: 19500, D: 0.9675, G:1.131

Iter: 19750, D: 0.8818, G:1.341

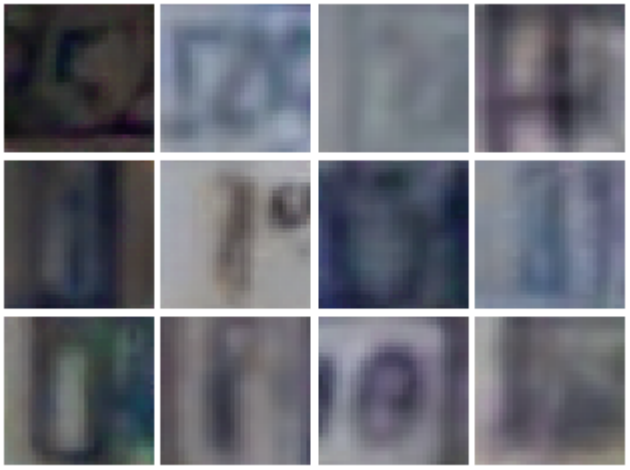
Iter: 20000, D: 1.206, G:0.9386

Iter: 20250, D: 1.198, G:1.721

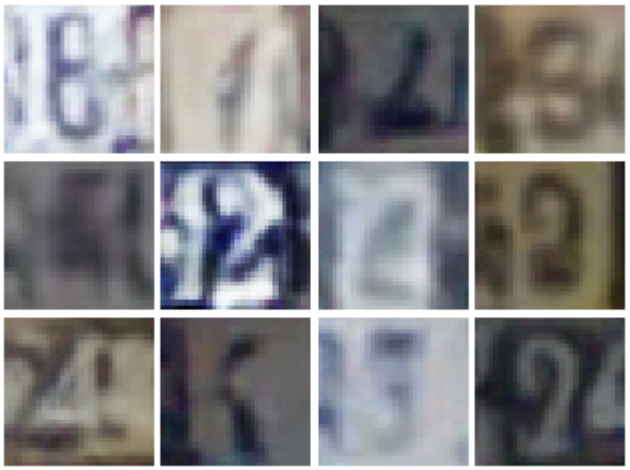
Iter: 20500, D: 1.255, G:1.083

Iter: 20750, D: 1.002, G:1.044

Iter: 21000, D: 1.079, G:1.219



Iter: 21250, D: 1.13, G:1.031

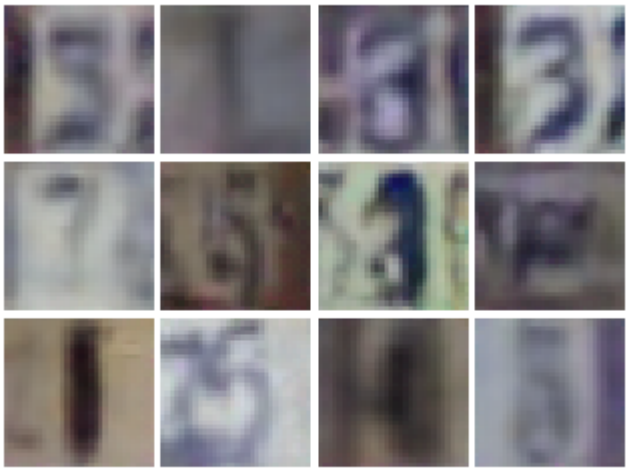


Iter: 21500, D: 1.066, G:1.352

Iter: 21750, D: 1.201, G:1.02

Iter: 22000, D: 1.163, G:1.014

Iter: 22250, D: 1.234, G:0.9257



Iter: 22500, D: 1.186, G:0.9466

```
Iter: 22750, D: 1.117, G:0.9053
```

Iter: 23000, D: 1.189, G:1.017

Iter: 23250, D: 1.186, G:0.997

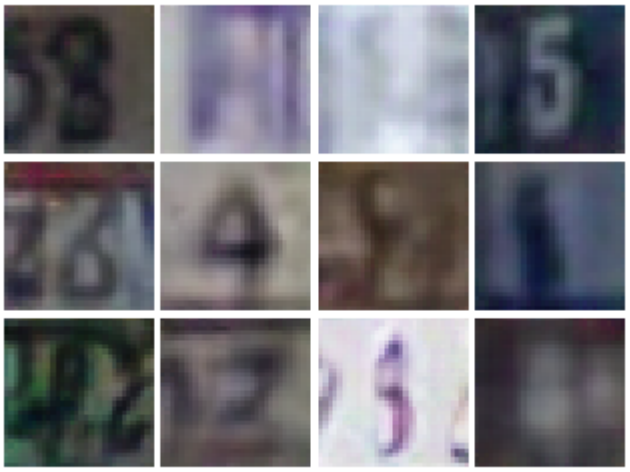
Iter: 23500, D: 0.9303, G:1.638

Iter: 23750, D: 1.174, G:0.9272

Iter: 24000, D: 0.925, G:1.148

Iter: 24250, D: 1.213, G:0.9479

Iter: 24500, D: 1.201, G:1.031

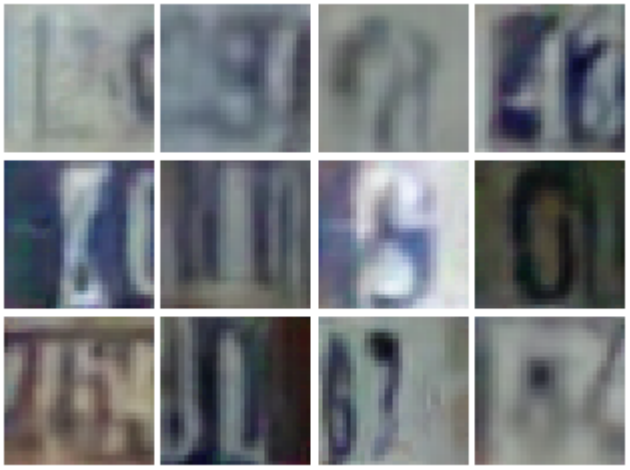


Iter: 24750, D: 1.043, G:1.229

Iter: 25000, D: 1.01, G:1.16

Iter: 25250, D: 1.162, G:0.9737

Iter: 25500, D: 1.119, G:1.046



Iter: 25750, D: 0.9775, G:1.329

Iter: 26000, D: 1.979, G:2.785

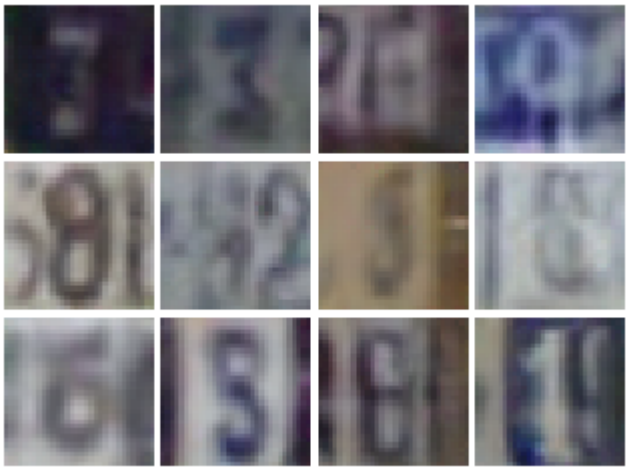
Iter: 26250, D: 0.8196, G:1.45

Iter: 26500, D: 0.8369, G:1.927

Iter: 26750, D: 1.13, G:1.076

Iter: 27000, D: 1.019, G:1.118

Iter: 27250, D: 0.8785, G:1.159



Iter: 27500, D: 0.9562, G:1.489

Iter: 27750, D: 1.068, G:1.339

Iter: 28000, D: 1.295, G:0.905

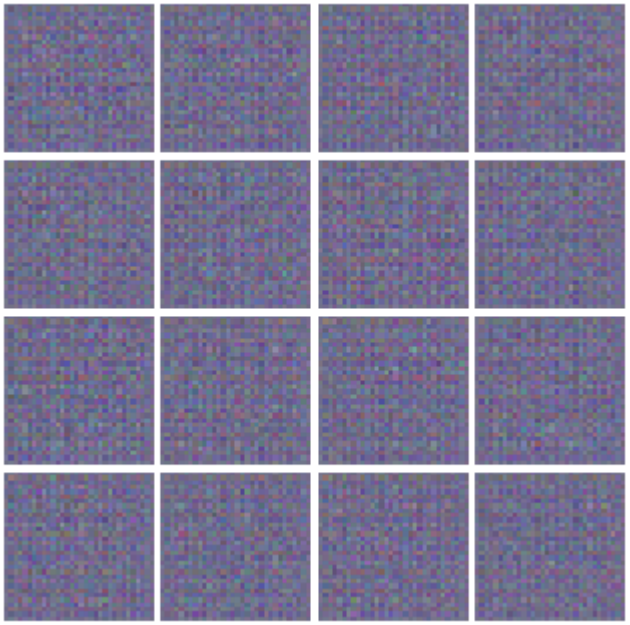
Iter: 28250, D: 0.8853, G:1.188

Iter: 28500, D: 1.189, G:1.071

You do not have to save the above training images for your final report, please just save the images generated by the cell below

In [ ]:
```
numIter = 0
if len(images) > 8:
    step = len(images) // 8
else:
    step = 1

for i in range(0, len(images), step):
    img = images[i]
    numIter = 250 * i * step
    print("Iter: {}".format(numIter))
    img = (img)/2 + 0.5
    show_images(img.reshape(-1, 3, 32, 32))
    plt.show()
```

```
Iter: 0
```



Iter: 49000

Iter: 98000

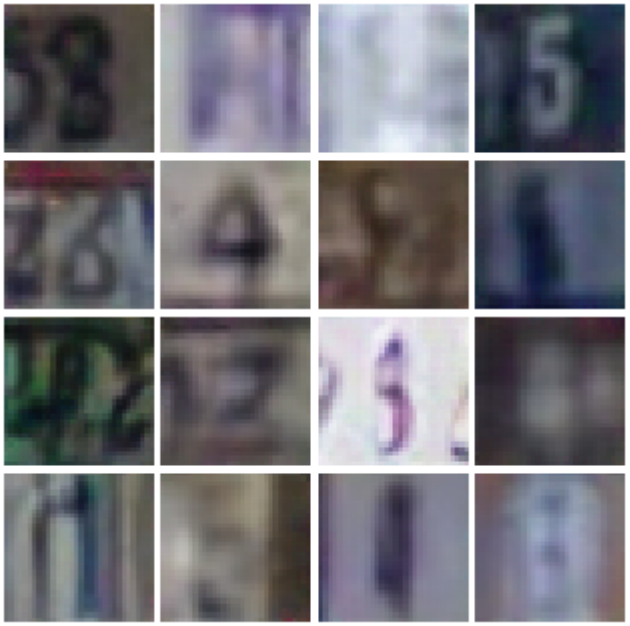
Iter: 147000

Iter: 196000

Iter: 245000

Iter: 294000

Iter: 343000



Iter: 392000

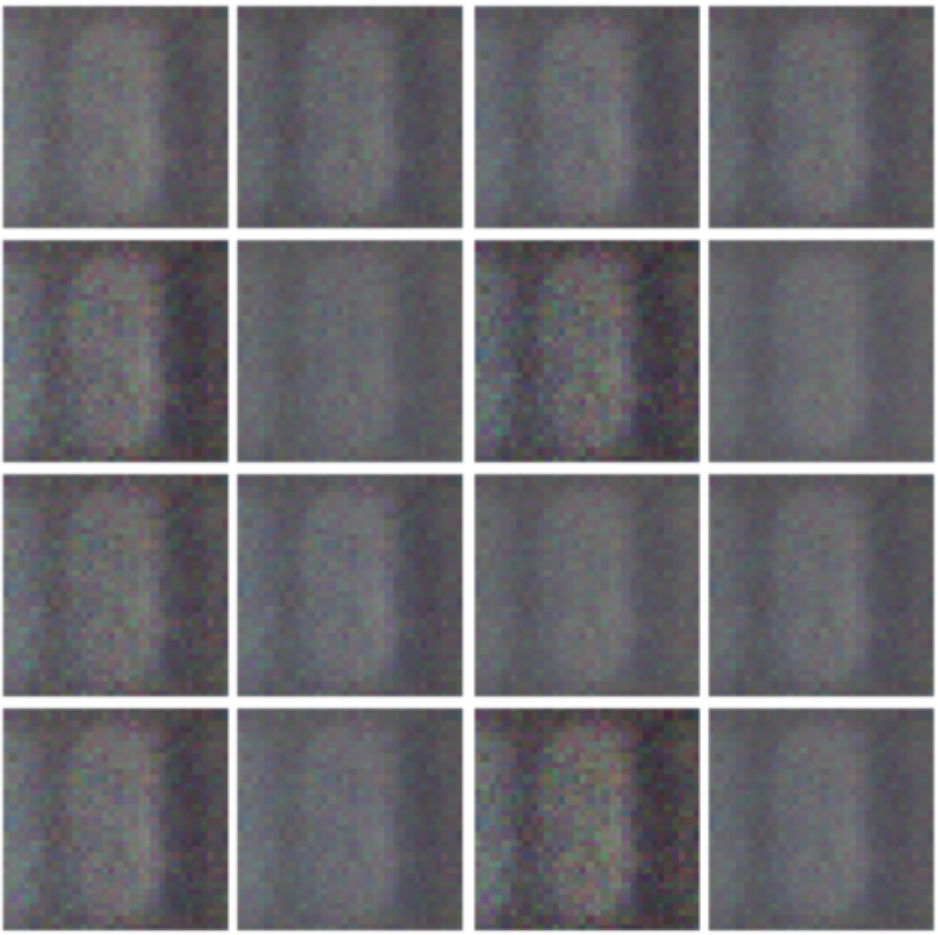
[TODO] GAN Output Visual Evaluation [2pts (Report)]

The main deliverable for this notebook is the final GAN generations! Run the cells in this section to show your final output. If you are unhappy with the generation, feel free to train for longer. Keep in mind we are not expecting perfect outputs, especially considering your limited computation resources. Below we have shown a few example images of what acceptable and unacceptable outputs are. Make sure your final image displays and is included in the report.

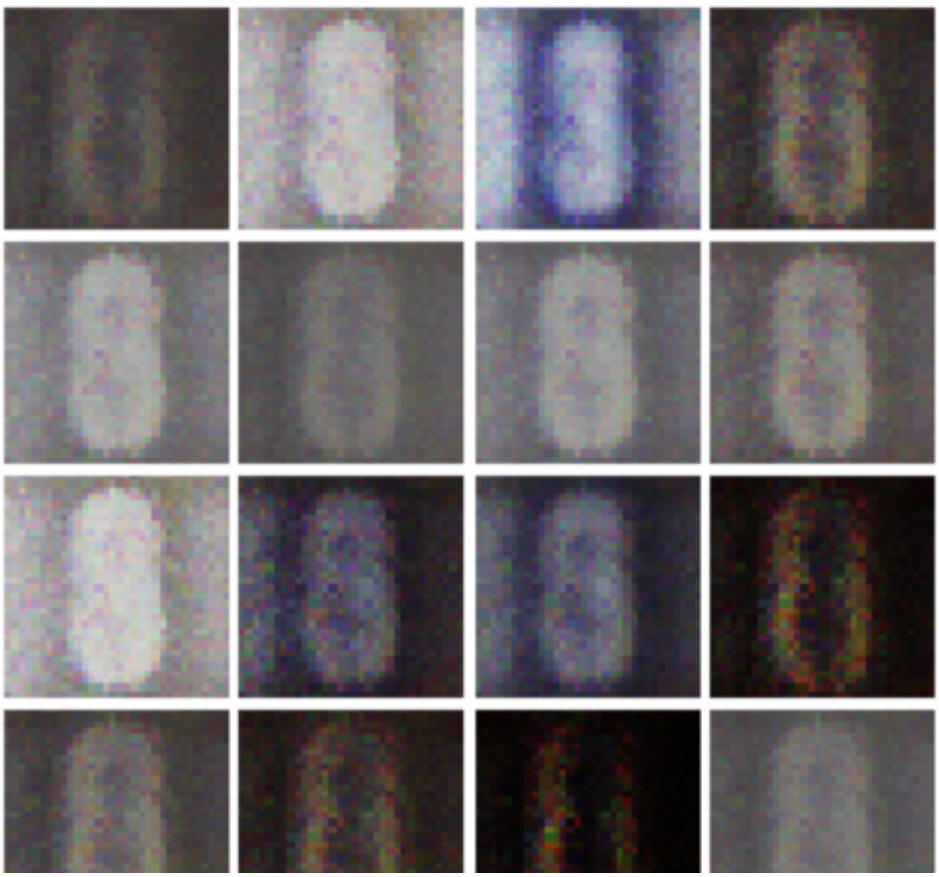
Acceptable Output:

Unreasonable Output:

Iter: 500



Iter: 750



In [ ]:
```
# This output is your answer - please include in the final report
print("Vanilla GAN final image:")
fin = (images[-1] - images[-1].min()) / (images[-1].max() - images[-1].mi
show_images(fin.reshape(-1, 3, 32, 32))
plt.show()
```

```
Vanilla GAN final image:
```

Reflections (2 Points)

[TODO]: Reflect on the GAN Training Process. What were some of the difficulties? How good were the generated outputs?

After understanding the structure of how the training should happen, the difficulties were a matter of tuning (although I relied on the suggested hyperparameters in the notebook, the network's architecture was still flexible).

For my first try, I just followed the suggestions for both discriminator and generator, and after knowing that my code was running smooth I experimented with the addition of batchnorm layers (following hte guidelines of https://arxiv.org/pdf/1511.06434.pdf).

The outputs were not perfect, but not bad. In a qualitative look, some generated images are very similar to the ones from the training set, and could even fool me.

[TODO] What is a mode collapse in GAN training, can you find any examples in your training?

Mode collapse in GAN training is when the outputs of the generator is not reflective of the actual dataset in the sense of not being as diverse. The symptoms are not being able to generate all the available "labels", or only being able to output that is very close to the training set. It basically means that the generator was not able to learn the "meaning" of all of the data.

Yes. Despite not having quantitative results, I can see that the generator tends to output certain numbers more than others. For example, the numbers 7 and 9 are barely seen. Moreover, on different runs I saw this phenomena with other numbers (almost always in pairs).

[TODO] Talk about the trends that you see in the losses for the generator and the discriminator. Explain why those values might make sense given the quality of the images generated after a given number of itterations.

I see a cyclic trend that the losses follow: in an iteration where the discriminator loss goes down, the generator loss goes up; when the generator loss goes down, the discriminator loss goes up. It makes sense because they are two competing networks. The way I see it, as the discriminator gets better, the generator has a harder time to fool it, and when it does, then the discriminator will fight against that.

Because of that it makes sense that the losses don't just go down as they usually do for common classification tasks, but instead they bounce around some number (in my case between 0.7 and 1.6).

Saving Models

In [ ]:
```
# You must save these models to run the next cells
torch.save(D.state_dict(), "d_smart.pth")
torch.save(G.state_dict(), "g_smart.pth")
```

Evaluating GANS

Evaluating via FID Score

It is hard to evaluate GANs effectiveness quantiatively. One way we can do this is by calculating FID (Frechet-Inception-Distance).

Website: https://wandb.ai/ayush-thakur/gan-evaluation/reports/How-to-Evaluate-GANs-using-Frechet-Inception-Distance-FID---Vmlldzo0MTAxOTI

But for this assignment, we will visually evaluate the GAN outputs.

TODO: Evaluating Discriminator

Not graded but for fun see how well your discriminator does against a sample generator.

In [ ]:
```
from part1_GANs.gan_pytorch import get_optimizer, eval_gan, eval_discrimi
from part1_GANs.gan_pytorch import generator, discriminator, discriminato

D = discriminator().type(dtype)
D.load_state_dict(torch.load("d_smart.pth"))
D.eval()

# oracle_g = torch.jit.load("../test_resources/g_smart_o.pt")
oracle_g = torch.jit.load("test_resources/g_smart_o.pt")
oracle_g.eval()

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

accuracy, p, r = eval_discriminator(D, oracle_g, loader_val, device)

# Test accuracy is greater than 0.55
assert accuracy > 0.55
```

```
Accuracy: 0.7161510555926917
Precision: 0.8934291885141465
Recall: 0.47673332899051074
```

Preparing for Submission

Run the following cell to collect `hw4_submission_part1.zip`. You will submit this to HW4: Part 2 - GANs on gradescope. Make sure to also export a PDF of this jupyter notebook and attach that to the end of your theory section. This PDF must show your answers to all the questions in the document, please include the final photos that you generate as well. Do not include all of your training images! You will not be given credit for anything that is not visible to us in this PDF.

In [ ]:
```
!sh collect_submission_part1.sh
```

```
sh: 0: cannot open collect_submission_part1.sh: No such file
```

Contributers

- Manav Agrawal (Lead)
- Matthew Bronars
- Mihir Bafna

In [ ]:
```
# # installations
# !pip install --upgrade pip

# !python --version
# !pip3 install jedi==0.16

# !pip3 uninstall cvxpy -y > /dev/null
# !pip3 install setuptools==65.5.0 pip==22.2 > /dev/null
# # hack for gym==0.21.0 https://github.com/openai/gym/issues/3176
# !pip3 install torch==1.13.1 torchvision==0.14.1 diffusers==0.11.1 \
# scikit-image==0.19.3 scikit-video==1.1.11 zarr==2.12.0 numcodecs==0.10.
# pygame==2.1.2 pymunk==6.2.1 gym==0.21.0 shapely==1.8.4 dnn \
# &> /dev/null # mute output
# !pip install --upgrade "jax[cuda12_pip]"==0.4.23 -f https://storage.goo
# # This will take a while, and do not worry if you get some warnings

## Issues: numcodecs==0.10.2, pygame==2.1.2, gym==0.21.0, upgrade jax num
```

In [ ]:
```
# from google.colab import drive
# drive.mount('/content/drive')

%load_ext autoreload
%autoreload 2

import os
#change to desired path to the part2-DiffusionModels folder
# os.chdir("./drive/MyDrive/hw4/DiffusionModels")
os.chdir("./part2_DiffusionModels")
%pwd
```

Out[ ]:
```
'/home/anascimento7/CS7643/Assignments/hw4_code_student_version/part2_Di
ffusionModels'
```

In [ ]:
```
from part2_DiffusionModels.tests import TestCases
unit_tester = TestCases()
```

Homework 3: Part 2 - Diffusion Models [21 points]

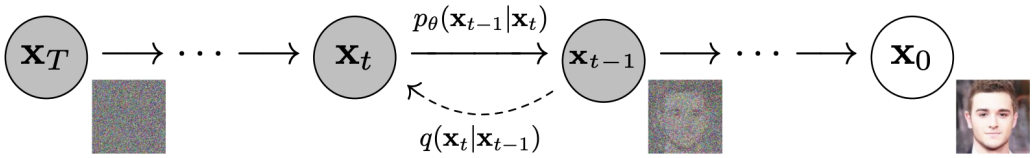
Denoising Diffusion Probabilistic Models (DDPM) - https://arxiv.org/abs/2006.11239/
Classifier Free Guidance (CFG) - https://arxiv.org/abs/2207.12598

In this part of the assignment you will be implementing a diffusion model and applying it in an image generation task. Denoising Diffusion Probabilistic Models (DDPMs), also known as Diffusion Models, are state of the art generative models that train a network to itteratively denoise random gaussian noise. If you have not done so already, please read through the DDPM paper linked above in order to get an understanding of the math behind DDPMs. The code for this part of the assignment is not particularly difficult, but you will have a difficult time if you do not understand the theory behind diffusion models. You will also be implementing classifier free guidance (CFG). While the CFG paper does not have to be read as thoroughly as the DDPM paper, it is definitely a useful read if you are not familiar with CFG.

Note: (you "may" have to revisit this part several times)

- Algorithm1 and Algorithm2 in the paper use 1 indexing for timesteps. We are using 0 indexing.
- In the paper they talk about using either β or $\tilde{\beta}$ for variance. We will always use β, even for the thresholding section
- For thresholding, when t = 0, just set $\bar{\alpha}_{t-1} = 1$
- Use noise_pred_net(sample = x, timestep = t, global_cond = c) to generate an output from the noise prediction network (alternatively you can just call noise_pred_net(x, t, c))
- For CFG, please use zeros_like

Creating the DDPM



For the sake of this assignment, we are breaking the Diffusion Model into two parts. A noise scheduler and a noise prediction net. The noise scheduler handles the noise addition during the forward process and the noise removal during the reverse process. The noise prediction net is a neural network that predicts the amount of noise that should be removed at each denoising step. Transformers, like the one you implemented in the last assignment, can be used as this noise prediction network. For this assignment we are using a U-Net as opposed to a transformer, but you will not be implementing this model, just training it.

IMPORTANT NOTE FOR IMPLEMENTATION!

You will be generating random numbers in various sections of this project. To keep the outputs deterministic for unit test and auto grader purposes, you must use our prebuilt randomizer function. This randomizer is defined in `DiffusionModels/randomizer.py`. Please take a look at that function before you start coding. The function that we define is sufficient for generating any random numbers that you will need for this project.

IMPORTANT NOTE 2

The equations in the DDPM paper describe an unconditional model, you will be implementing a conditional model. This does not significantly change the equations, you just need to pass in the conditioning term along with x and t. Make sure to watch the first lecture on generative models if you are running into confusion about this or other aspects of Diffusion Models.

IMPORTANT NOTE 3

Algorithm 1 - Training, and Algorithm 2 - Sampling in the DDPM paper use 1 indexing for timesteps. However, you should use 0 indexing (index start from 0).

Part 2.1 - Noise Scheduler [6 points]

2.1.1 - Forward Process (Adding Noise) [3 points] [.5 writeup points]

During the forward process, we take an input and gradually add Gaussian noise to the data according to a variance schedule. This is described in section 2, background, of the DDPM paper. You must read that section to complete this part of the assignment.

TODO: Complete initialization function in `DiffusionModels/noise_scheduler.py`

TODO: Implement `add_noise` function in `part2-DiffusionModel/noise_scheduler.py`

Hint: Pay attention to equation 4

Question: what does $q(x_t|x_o)$ represent for a diffusion model? (read the paper)

Answer: the (Gaussian) noise generation process for any t conditioned on x_0. That is the distribution used for the noising (forward) process.

In [ ]:
```
# test initialization
unit_tester.test_noise_scheduler_init()
# test add noise
unit_tester.test_add_noise()
```

```
NoiseScheduler() initialization test case passed!
add_noise() test case passed!
```

2.1.2 - Reverse Process (Removing Noise) [2.5 point]

When taking a step during the backwards process, we have access to a partially noised sample x_t, the denoising timestep t, and our model's noise prediction. We remove the predicted noise to get a slightly less noisy sample x_{t-1}. The completely uncorrupted sample is recovered at x_0. This process is described in detail in section 3 of the DDPM paper, please read this section before completing this part of the assignment.

TODO: Implement `denoise_step` function in `DiffusionModels/noise_scheduler.py` (do not implement the thresholding part yet)

Hint: Pay attention to algorithm 2 - Sampling

In [ ]:
```
# test denoise step
unit_tester.test_denoise_step()
```

```
denoise_step() test case passed!
```

Part 2.2 - Constructing the Diffusion Model [7

Points]

We have provided you with a noise prediction network, and you have just written a noise scheduler. With these two parts you are able to put together a completed diffusion model.

2.2.1 Compute Loss for Training [2 points] [.5 writeup points]

For this part of the assignment, you will be implementing the `compute_loss_on_batch` function of the diffusion model. We have implemented the rest of the training logic for you. Keep in mind, the only part of the diffusion model that needs to be trained is the noise prediction network. Your loss function should capture how well the noise_pred_net estimates the added noise. We expect you to implement the simplified variant of the variational lower bound as described in section 3.4 of the DDPM paper.

TODO: Implement loss computation section of `compute_loss_on_batch` function in `DiffusionModels/diffusion_model.py`

Hint: Use your noise scheduler, refer to equation 14 and Algorithm 1 - Training
Hint: For each item in the batch, you should be selecting a random timestep t to compute the loss

Question: What is $\epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)$?

Answer: It is the learned noise, which we are optimizing for (i.e., updating θ so that we can correctly predict ϵ).

In a more detailed manner:

- ϵ is the ground truth that we have access to
- $\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}$ is the process of jumping to timestep t conditioned on x_0, which yields some noisy data
- That noisy data is ran though the neural net with the timestep t, which will predict $\epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)$ itself

In [ ]:
```
# test loss computation
# print(os.getcwd())
unit_tester.test_compute_loss_on_batch()
```

```
compute_loss_on_batch() test case passed!
```

2.2.2 Classifier Free Guidance [1 Point]

We want our diffusion model to make use of classifier free guidance. If you are not familiar with classifier free guidance, please take a look at the CFG paper now. Classifier free guidance allows us to variably combine noise predictions from a conditional and an unconditional noise prediction network at inference time. This has the effect of enhancing class specific features and is critical for high quality image

generation. In practice, we train the unconditional noise prediction net at the same time as the conditional noise prediction net. Section 3.2 of the CFG paper describes this process.

We expect you to use 0s as the null tokens.

TODO: Implement classifier free guidance section of `compute_loss_on_batch` function in `DiffusionModels/diffusion_model.py`

Hint: This should only be a few lines, use self.p_uncond

In [ ]:
```
# test classifier free guidance training
unit_tester.test_compute_loss_with_cfg()
```

```
compute_loss_with_cfg() test case passed!
```

2.2.3 Sample Generation (with classifier free guidance) [3.5 points]

Before we can actually use our diffusion model, we need to write the code for generating a sample. To generate an output, we draw a sample x_T from random gaussian noise. We then itteratively denoise the random output for T denoising timesteps until we get our uncorrupted output x_0. You will be implementing sampling with classifier-free guidance. Do not worry about replicating algorithm 2 of the CFG paper, we only expect you to calculate the updated noise prediction from equation 6 of the CFG paper:

$$\bar{\epsilon_\theta}(x_t, t, c) = (1 + w) * \epsilon_\theta(x_t, t, c) - w * \epsilon_\theta(x_t, t, \text{empty})$$

where ϵ_θ is the predicted noise from your noise prediction network and w is the guidance weight

TODO: Implement `generate_sample` function in `DiffusionModels/diffusion_model.py`

Hint: Use your noise scheduler and refer to Algorithm 2 - Sampling of the DDPM paper
Hint: Dont forget that this is a conditional model and to include classifier free guidance

In [ ]:
```
# test generate sample
unit_tester.test_generate_sample()
```

```
generate_sample_step() test case passed!
generate_sample_step() with guidance test case passed!
```

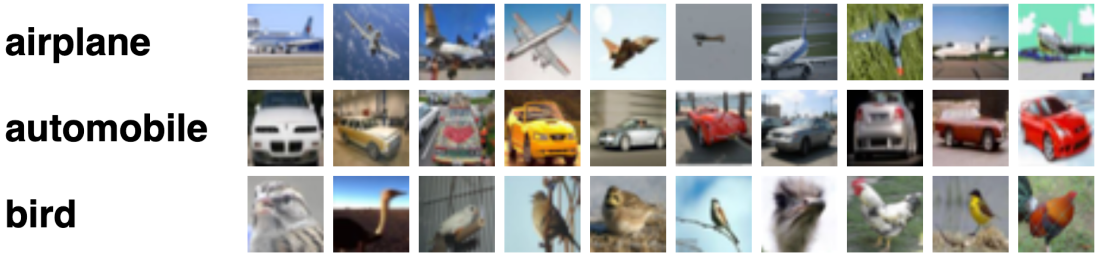
In [ ]:
```
## Trying to test threshold
unit_tester.test_threshold_denoise_step()
```

```
threshold_denoise_step() test case passed!
```

Part 2.3 - Diffusion Model Applications (Computer Vision) [8 points]

Diffusion models are mostly known in popular culture because of image generation

tools such as Dall-e and Midjourney. These models are able to take in text based prompts and output extremely realistic photos. For the sake of this assignment we will not be implementing anything that is nearly as visually impressive, but we hope to shed some insight on a few critical aspects to state of the art diffusion models. Namely classifier free guidance and thresholding. You will be working with a subset of the CIFAR-10 dataset. The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class. We will be working with the first three classes of this dataset - airplanes, automobiles, and birds.



In [ ]:
```
# Imports and global function definitions
from part2_DiffusionModels.computer_vision.image_gen import ImageGenerato
from matplotlib import pyplot as plt

def plot_image(output):
    B, H, W, C = output.shape
    # batch size divided by number of classes
    num_cols = B // 3
    assert B % 3 == 0
    fig, axes = plt.subplots(3, num_cols, figsize=(12, 8))
    # Loop through the rows
    for i in range(3):
        # Loop through the columns
        for j in range(B // 3):
            image = output[i * (B // 3) + j]
            if num_cols == 1:
                axes[i].imshow(image)
                axes[i].axis('off')
            else:
                axes[i, j].imshow(image)
                axes[i, j].axis('off')

    # label the first row "airplane", the second row "automobile", and th
    axes[0, 0].set_title('airplane')
    axes[1, 0].set_title('automobile')
    axes[2, 0].set_title('bird')

    plt.show()
```

In [ ]:
```
# # Run this cell to download the dataset, only needs to be done once eve
# %cd computer_vision
# !wget https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
# !tar -xvzf cifar-10-python.tar.gz
# !rm cifar-10-python.tar.gz
# %cd ..
```

2.3.1 Initialization and Data Augmentation [2 writeup points]

In parts 2.1 and 2.2 you implemented diffusion model training and sample generation, so the majority of the work for generating images is done! The computer_vision/image_gen.py file handles the rest of the logic for image generation. Your first task is to complete the initialization function, this is only a few lines of code.

TODO: Complete initialization function in `DiffusionModels/computer_vision/image_gen.py`

In [ ]:
```
generator = ImageGenerator()
```

After initializing the image generator, take a look through the rest of the file to understand how the data is loaded and how the images are generated. Your next task is to perform data augmentation on the CIFAR dataset (for the sake of training time we do not require you to actually add augmenations and increase the size of your dataset, but go through this exersize with us anyway). This is a common technique in the computer vision world as it allows us to increase the size of our dataset and train the model to be invariant to certain perturbations. You may have done this before when training a classification model, but now we are training a generator. The types of augmentations that we want our generator to be invariant to are not necessariliy the same as for a classifier.

Question: *Why might we want a classifier to be invariant to different augmentations then a generator? Name 2 augmentations that we might perform on a classification dataset that we don't want to perform on a dataset for a generative model. Name 2 augmentations that you want to perform on a dataset for a generative model (its okay if you want also want to do the same augmentations on a classification dataset).*

Your Answer Here: We want classifiers need to be robust to minor variations (out-of-distribution data, but somewhat bounded) and perform the class classifcation correctly; hence, be invariant to those augmentations. On the other hand, we want generators to be more "creative" and give us diverse outputs.

- Two augmentations that we might perform on a classification dataset that we don't want to perform on a dataset for a generative model:
  - Addition of blurr only in the background, not the objects (class specific occlusion)
  - Addition of noise that preserve only the labels (label preserving noise injection)
- Two augmentations that you want to perform on a dataset for a generative model:
  - Addition of blurr (classifiers might want)
  - Addition of flips and rotations (classifiers might want)

OPTIONAL: Implement two augmentations in the `load_dataset` function of `DiffusionModels/computer_vision/image_gen.py` , (these should be the augmentations you described above). Increasing the dataset size will slow down your training, so we do not require you to actually implement these augmentations.

However, we have left room for you to add the code for augmentation if you have the time or compute.
OPTIONAL: Display an image from the CIFAR dataset and show it after the augmentations (3 images total: original, augmentation 1, augmentation 2)

In [ ]:
```
# Feel free to add more cells or generate the images separately and displ
```

2.3.2 Image Generation and Classifier Free Guidance [1 writeup point]

Now it is time to train the model - if this fails to run, go and check over your previous work. We are implementing a small model, but even still, this will take a while to train! Please use google colab if you do not have access to a GPU. The remainder of this assignment will be infeasable for you if you do not have a GPU. That being said, we understand that you have time constraints and we do not expect you to let this model train for hours (although your results would look much better if you did).

We ask that you budget 30 minutes for the training of this image generation model, you can run it for longer if you prefer though. Do not stress too much about the specific images generated, we know the outputs will be poor given your limited computation budget. If your loss isnt decreasing (should get below 0.1 after ~7 epochs of training and below 0.05 by the end of training) or you are generating random noise, then you may have a serious issue. But if your airplane is a smudge on a blue-ish background, that is a great output. The goal is for you to learn, we will be paying much more attention to your analysis than your images.

In [ ]:
```
generator.load_dataset(dataset_paths=[
    'computer_vision/cifar-10-batches-py/data_batch_1',
    'computer_vision/cifar-10-batches-py/data_batch_2',
    'computer_vision/cifar-10-batches-py/data_batch_3',
    'computer_vision/cifar-10-batches-py/data_batch_4',
    'computer_vision/cifar-10-batches-py/data_batch_5',
    ], batch_size = 64
     )
```

In [ ]:
```
# You can just rerun this cell to keep training the model

# Keep the number of epochs at 25 for now (if you need it to be smaller t
# You will have a chance to increase the training time and generate your
generator.train_policy(epochs=25)
generator.policy.save_weights('model_pths/image_weights.pth')

# Load the model from the checkpoint
# generator.policy.load_weights('model_pths/image_weights.pth')
# generator.load_dataset(dataset_paths=[
#     'computer_vision/cifar-10-batches-py/data_batch_1',
#     'computer_vision/cifar-10-batches-py/data_batch_2',
#     'computer_vision/cifar-10-batches-py/data_batch_3',
#     'computer_vision/cifar-10-batches-py/data_batch_4',
#     'computer_vision/cifar-10-batches-py/data_batch_5',
#     ], batch_size = 64
#      )
```

```
Batch:   0%|          | 0/235 [00:00<?, ?it/s]
```

```
Batch:   0%|          | 0/235 [00:00<?, ?it/s]
Training Progress:   0%|          | 0/25 [00:00<?, ?it/s]
```

```
---------------------------------------------------------------------------
-
RuntimeError                              Traceback (most recent call las
t)
Cell In[17], line 5
      1 # You can just rerun this cell to keep training the model
      2
      3 # Keep the number of epochs at 25 for now (if you need it to be sm
aller that is okay as well)
      4 # You will have a chance to increase the training time and generat
e your best images at the end of this section
----> 5 generator.train_policy(epochs=25)
      6 generator.policy.save_weights('model_pths/image_weights.pth')
      8 # Load the model from the checkpoint
      9 # generator.policy.load_weights('model_pths/image_weights.pth')
     10 # generator.load_dataset(dataset_paths=[
   (...)
     16 #     ], batch_size = 64
     17 #      )

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/c
omputer_vision/image_gen.py:90, in ImageGenerator.train_policy(self, epoch
s)
     89 def train policy(self, epochs = 10):
---> 90     self.policy.train(train_epochs = epochs, data_loader=self.data
_loader)

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/d
iffusion_model.py:94, in DiffusionModel.train(self, data_loader, train_epo
chs)
     92 data = data.to(self.device)
     93 cond = cond.to(self.device)
---> 94 loss = self.compute_loss_on_batch(data, cond)
     95 loss.backward()
     96 self.optimizer.step()

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/d
iffusion_model.py:179, in DiffusionModel.compute_loss_on_batch(self, data,
cond, seed)
    173 noisy_data = self.noise_scheduler.add_noise(data, epsilon, t) # x_
t
    174 # print('noisy_data:', noisy_data.shape)
    175
    176 # # loss = (torch.norm(noise_GT - noisy_data))^2
    177
    178 ## From the notebook: use noise_pred_net(sample = x, timestep = t,
global_cond = c) to generate an output from the noise prediction network
(alternatively you can just call noise pred net(x, t, c))
--> 179 ep_theta = self.noise_pred_net(noisy_data.to(self.device), t.to(se
lf.device), cond.to(self.device))
    181 loss = torch.nn.functional.mse_loss(epsilon.to(self.device), ep_th
eta.to(self.device))
    184 ########################################
    185 #             END OF YOUR CODE         #
    186 ########################################

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/module.py:1511, in Module._wrapped_call_impl(self, *args, **kwargs)
   1509     return self._compiled_call_impl(*args, **kwargs)  # type: igno
re[misc]
```

```
   1510 else:
-> 1511     return self._call_impl(*args, **kwargs)

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/module.py:1520, in Module._call_impl(self, *args, **kwargs)
   1515 # If we don't have any hooks, we want to skip the rest of the logi
c in
   1516 # this function, and just call forward.
   1517 if not (self._backward_hooks or self._backward_pre_hooks or self._
forward_hooks or self._forward_pre_hooks
   1518         or _global_backward_pre_hooks or _global_backward_hooks
   1519         or  global forward hooks or  global_forward_pre_hooks):
-> 1520     return forward_call(*args, **kwargs)
   1522 try:
   1523     result = None

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/n
oise_prediction_net.py:623, in ConditionalUnet2D.forward(self, x, t, y)
    621     y_emb = self.label_emb(y)
    622     t = t + y emb
--> 623 return self.unet_forwad(x, t)

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/n
oise_prediction_net.py:581, in UNet.unet_forwad(self, x, t)
    580 def unet_forwad(self, x, t):
--> 581     x1 = self.inc(x)
    582     x2 = self.down1(x1, t)
    583     x2 = self.sa1(x2)

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/module.py:1511, in Module._wrapped_call_impl(self, *args, **kwargs)
   1509     return self._compiled_call_impl(*args, **kwargs)  # type: igno
re[misc]
   1510 else:
-> 1511     return self._call_impl(*args, **kwargs)

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/module.py:1520, in Module._call_impl(self, *args, **kwargs)
   1515 # If we don't have any hooks, we want to skip the rest of the logi
c in
   1516 # this function, and just call forward.
   1517 if not (self._backward_hooks or self._backward_pre_hooks or self._
forward_hooks or self._forward_pre_hooks
   1518         or _global_backward_pre_hooks or _global_backward_hooks
   1519         or  global forward hooks or  global_forward_pre_hooks):
-> 1520     return forward_call(*args, **kwargs)
   1522 try:
   1523     result = None

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/n
oise_prediction_net.py:486, in DoubleConv.forward(self, x)
    484     return F.gelu(x + self.double_conv(x))
    485 else:
--> 486     return self.double_conv(x)

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/module.py:1511, in Module._wrapped_call_impl(self, *args, **kwargs)
   1509     return self._compiled_call_impl(*args, **kwargs)  # type: igno
re[misc]
   1510 else:
```

```
-> 1511     return self._call_impl(*args, **kwargs)

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/module.py:1520, in Module._call_impl(self, *args, **kwargs)
   1515 # If we don't have any hooks, we want to skip the rest of the logi
c in
   1516 # this function, and just call forward.
   1517 if not (self._backward_hooks or self._backward_pre_hooks or self._
forward_hooks or self._forward_pre_hooks
   1518         or _global_backward_pre_hooks or _global_backward_hooks
   1519         or  global forward hooks or  global_forward_pre_hooks):
-> 1520     return forward_call(*args, **kwargs)
   1522 try:
   1523     result = None

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/container.py:217, in Sequential.forward(self, input)
    215 def forward(self, input):
    216     for module in self:
--> 217         input = module(input)
    218     return input

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/module.py:1511, in Module._wrapped_call_impl(self, *args, **kwargs)
   1509     return self._compiled_call_impl(*args, **kwargs)  # type: igno
re[misc]
   1510 else:
-> 1511     return self._call_impl(*args, **kwargs)

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/module.py:1520, in Module._call_impl(self, *args, **kwargs)
   1515 # If we don't have any hooks, we want to skip the rest of the logi
c in
   1516 # this function, and just call forward.
   1517 if not (self._backward_hooks or self._backward_pre_hooks or self._
forward_hooks or self._forward_pre_hooks
   1518         or _global_backward_pre_hooks or _global_backward_hooks
   1519         or  global forward hooks or  global_forward_pre_hooks):
-> 1520     return forward_call(*args, **kwargs)
   1522 try:
   1523     result = None

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/conv.py:460, in Conv2d.forward(self, input)
    459 def forward(self, input: Tensor) -> Tensor:
--> 460     return self._conv_forward(input, self.weight, self.bias)

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/conv.py:456, in Conv2d._conv_forward(self, input, weight, bias)
    452 if self.padding_mode != 'zeros':
    453     return F.conv2d(F.pad(input, self._reversed_padding_repeated_t
wice, mode=self.padding_mode),
    454                     weight, bias, self.stride,
    455                     pair(0), self.dilation, self.groups)
--> 456 return F.conv2d(input, weight, bias, self.stride,
    457                 self.padding, self.dilation, self.groups)

RuntimeError: cuDNN error: CUDNN_STATUS_INTERNAL_ERROR
```

First, you will evaluate the output of unguided image generation. The generate_images

will produce num_samples generations of each class.

In [ ]:
```
image = generator.generate_images(guidance=0, num_samples=8)
plot_image(image)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[16], line 1
----> 1 image = generator.generate_images(guidance=0, num_samples=8)
      2 plot_image(image)

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/computer_vision/image_gen.py:119, in ImageGenerator.generate_images(self, guidance, num_samples, threshold, class_label)
    113     all_labels = [class_label] * num_samples * 3
    115 # The size of this conditioning tensor is different than what you would expect given the function signature of generate_sample
    116 # If you are looking at this before implementing generate_sample, the first dimension in this cond tensor is still the batch size
    117 # The generate_sample function signature reflects the shape of the conditioning tensor for the 1D prediction network (part 2.4)
    118 # If none of this makes sense to you, that is fine, you do not need to worry about any of this
--> 119 cond = torch.tensor(all_labels).to(self.policy.device)
    121 # generate the image
    122 image = self.policy.generate_sample(cond, guidance_weight = guidance, threshold=threshold)

RuntimeError: CUDA error: device-side assert triggered
CUDA kernel errors might be asynchronously reported at some other API call, so the stacktrace below might be incorrect.
For debugging consider passing CUDA_LAUNCH_BLOCKING=1.
Compile with `TORCH_USE_CUDA_DSA` to enable device-side assertions.
```

Question: What do you notice about the quality of unguided image generation? How effectively is the model learning each class? This is very early in the training process, but do you notice any features that the model is learning for each class?

Your Answer Here: Unfortunatelly, I am running into CUDA errors, the most frequent being `RuntimeError: cuDNN error: CUDNN_STATUS_INTERNAL_ERROR`, therefore I can not see the images nor draw conclusions about it. Below are some of the things I have tried, but it is also important to note that I did try many different numbers for `self.condition_dim`, both in the int format and tuple of ints.

I am running locally, in VSCode and already tried restarting the kernel, closing and reopening the program, and even restarting my machine. Unfortunatelly I don't have time to dig any further in the web or talk about this in OH. I made a post on piazza about this, apparently some people had the same issue.

Now, we will evaluate image generation with different guidance weights. To do this, we dont actually need to retrain the model at all, this guidance can be done purely at inference time. Great! Explain why we dont have to retrain for image guidance.

Your Answer Here: Unfortunatelly, I am running into CUDA errors, the most frequent being `RuntimeError: cuDNN error: CUDNN_STATUS_INTERNAL_ERROR`, therefore I can not see the images nor draw conclusions about it. Below are some of the things I have tried, but it is also important to note that I did try many different numbers for `self.condition_dim`, both in the int format and tuple of ints.

I am running locally, in VSCode and already tried restarting the kernel, closing and reopening the program, and even restarting my machine. Unfortunatelly I don't have time to dig any further in the web or talk about this in OH. I made a post on piazza about this, apparently some people had the same issue.

In [ ]:
```
image = generator.generate_images(guidance=0.5, num_samples=8)
plot_image(image)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[18], line 1
----> 1 image = generator.generate_images(guidance=0.5, num_samples=8)
      2 plot_image(image)

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/computer_vision/image_gen.py:119, in ImageGenerator.generate_images(self, guidance, num_samples, threshold, class_label)
    113     all_labels = [class_label] * num_samples * 3
    115 # The size of this conditioning tensor is different than what you would expect given the function signature of generate_sample
    116 # If you are looking at this before implementing generate_sample, the first dimension in this cond tensor is still the batch size
    117 # The generate_sample function signature reflects the shape of the conditioning tensor for the 1D prediction network (part 2.4)
    118 # If none of this makes sense to you, that is fine, you do not need to worry about any of this
--> 119 cond = torch.tensor(all_labels).to(self.policy.device)
    121 # generate the image
    122 image = self.policy.generate_sample(cond, guidance_weight = guidance, threshold=threshold)

RuntimeError: CUDA error: device-side assert triggered
CUDA kernel errors might be asynchronously reported at some other API call, so the stacktrace below might be incorrect.
For debugging consider passing CUDA_LAUNCH_BLOCKING=1.
Compile with `TORCH_USE_CUDA_DSA` to enable device-side assertions.
```

In [ ]:
```
image = generator.generate_images(guidance=1, num_samples=8)
plot_image(image)
```

```
---------------------------------------------------------------------------
-
RuntimeError                              Traceback (most recent call las
t)
Cell In[19], line 1
----> 1 image = generator.generate_images(guidance=1, num_samples=8)
      2 plot_image(image)

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/c
omputer_vision/image_gen.py:119, in ImageGenerator.generate_images(self, g
uidance, num_samples, threshold, class_label)
    113     all_labels = [class_label] * num_samples * 3
    115 # The size of this conditioning tensor is different than what you
would expect given the function signature of generate_sample
    116 # If you are looking at this before implementing generate_sample,
the first dimension in this cond tensor is still the batch size
    117 # The generate_sample function signature reflects the shape of the
conditioning tensor for the 1D prediction network (part 2.4)
    118 # If none of this makes sense to you, that is fine, you do not nee
d to worry about any of this
--> 119 cond = torch.tensor(all_labels).to(self.policy.device)
    121 # generate the image
    122 image = self.policy.generate_sample(cond, guidance_weight = guidan
ce, threshold=threshold)

RuntimeError: CUDA error: device-side assert triggered
CUDA kernel errors might be asynchronously reported at some other API call
, so the stacktrace below might be incorrect.
For debugging consider passing CUDA_LAUNCH_BLOCKING=1.
Compile with `TORCH_USE_CUDA_DSA` to enable device-side assertions.
```

In [ ]:
```
image = generator.generate_images(guidance=2, num_samples=8)
plot_image(image)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[20], line 1
----> 1 image = generator.generate_images(guidance=2, num_samples=8)
      2 plot_image(image)

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/computer_vision/image_gen.py:119, in ImageGenerator.generate_images(self, guidance, num_samples, threshold, class_label)
    113     all_labels = [class_label] * num_samples * 3
    115 # The size of this conditioning tensor is different than what you would expect given the function signature of generate_sample
    116 # If you are looking at this before implementing generate_sample, the first dimension in this cond tensor is still the batch size
    117 # The generate_sample function signature reflects the shape of the conditioning tensor for the 1D prediction network (part 2.4)
    118 # If none of this makes sense to you, that is fine, you do not need to worry about any of this
--> 119 cond = torch.tensor(all_labels).to(self.policy.device)
    121 # generate the image
    122 image = self.policy.generate_sample(cond, guidance_weight = guidance, threshold=threshold)

RuntimeError: CUDA error: device-side assert triggered
CUDA kernel errors might be asynchronously reported at some other API call, so the stacktrace below might be incorrect.
For debugging consider passing CUDA_LAUNCH_BLOCKING=1.
Compile with `TORCH_USE_CUDA_DSA` to enable device-side assertions.
```

In [ ]:
```
image = generator.generate_images(guidance=5, num_samples=8)
plot_image(image)
```

```
---------------------------------------------------------------------------
-
RuntimeError                              Traceback (most recent call las
t)
Cell In[21], line 1
----> 1 image = generator.generate_images(guidance=5, num_samples=8)
      2 plot_image(image)

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/c
omputer_vision/image_gen.py:119, in ImageGenerator.generate_images(self, g
uidance, num_samples, threshold, class_label)
    113     all_labels = [class_label] * num_samples * 3
    115 # The size of this conditioning tensor is different than what you
would expect given the function signature of generate_sample
    116 # If you are looking at this before implementing generate_sample,
the first dimension in this cond tensor is still the batch size
    117 # The generate_sample function signature reflects the shape of the
conditioning tensor for the 1D prediction network (part 2.4)
    118 # If none of this makes sense to you, that is fine, you do not nee
d to worry about any of this
--> 119 cond = torch.tensor(all_labels).to(self.policy.device)
    121 # generate the image
    122 image = self.policy.generate_sample(cond, guidance_weight = guidan
ce, threshold=threshold)

RuntimeError: CUDA error: device-side assert triggered
CUDA kernel errors might be asynchronously reported at some other API call
, so the stacktrace below might be incorrect.
For debugging consider passing CUDA_LAUNCH_BLOCKING=1.
Compile with `TORCH_USE_CUDA_DSA` to enable device-side assertions.
```

Question : What do you notice about the generations with classifier free guidance? What features does the model pick up as being most indicattive of each class? Does performance degrade as guidance increases? If so, how?

Your Answer Here : Unfortunatelly, I am running into CUDA errors, the most frequent being `RuntimeError: cuDNN error: CUDNN_STATUS_INTERNAL_ERROR`, therefore I can not see the images nor draw conclusions about it. Below are some of the things I have tried, but it is also important to note that I did try many different numbers for `self.condition_dim`, both in the int format and tuple of ints.

I am running locally, in VSCode and already tried restarting the kernel, closing and reopening the program, and even restarting my machine. Unfortunatelly I don't have time to dig any further in the web or talk about this in OH. I made a post on piazza about this, apparently some people had the same issue.

2.3.3 Thresholding [2 points] + [3 writeup points]

You may have noticed that with high guidance weights, the images tend to oversaturate. This is because the model is predicting values outside of the range [-1, 1] (our input is normalized to this range before being passed to the model). To help mitigate this, it is common to threshold sample predictions at each denoising step. As our denoising step is currently implemented, we can't actually use this technique. This is because we directly estimate x_{t-1}. Why shouldn't we just threshold x_{t-1} to be within

[-1, 1]? Additionally, why is thresholding particularly useful when dealing with large guidance weights? Will thresholding have an effect even if we don't have a large guidance weight?

Your Answer Here: For the coding answer, you can see that I run this above, and it works (I'll try to link that page in Gradescope).

For high quality, we need to play with the values of the bounds for thresholding, so I see it as hyperparameters: it might have a good or a bad effect depending on the values you choose. If we do constrain it to [-1, 1] though, we might make the noise prediction outside what is within the true distribution that we are trying to learn, and maybe also lose data (whatever is outside that interval) in the process.

Although, when dealing with large guidance weights, thresholding will do some clipping to outputs that are probably not wanted, which is a heuristic to deal with outputs that are non-representative of the distribution we are learning.

Even if we don't have a large guidance weight, threshold can still have an effect of preventing the generation of pixel values that are off (too dark or too bright), and/or outliers of the image distribution ("artifacts").

Disclosure: Unfortunatelly, I am running into CUDA errors, the most frequent being `RuntimeError: cuDNN error: CUDNN_STATUS_INTERNAL_ERROR` , therefore I can not see the images nor draw conclusions about it. Below are some of the things I have tried, but it is also important to note that I did try many different numbers for `self.condition_dim` , both in the int format and tuple of ints.

I am running locally, in VSCode and already tried restarting the kernel, closing and reopening the program, and even restarting my machine. Unfortunatelly I don't have time to dig any further in the web or talk about this in OH. I made a post on piazza about this, apparently some people had the same issue.

The answer above is what makes sense to me + some research that I did.

So we need to compute x_{t-1} such that we estimate x_0 as an intermediate step. This way we can threshold our estimate of x_0.
This is shown in equations (6) and (7) of the DDPM paper. The equation for computing an estimate of x_0 is shown in equation (15). Make sure to threshold the prediction of x_0!

TODO: Implement denoising with thresholding in the `denoise_step` function in `DiffusionModels/noise_scheduler.py`

In [ ]:
```
# test denoise step with thresholding
unit_tester.test_threshold_denoise_step()
```

Now that you have completed the thresholding, evaluate how this changes image generation with different guidance weights.

In [ ]:
```
image = generator.generate_images(guidance=0, num_samples=8, threshold=Tr
plot_image(image)
```

In [ ]:
```
image = generator.generate_images(guidance=0.5, num_samples=8, threshold=
plot_image(image)
```

In [ ]:
```
image = generator.generate_images(guidance=1, num_samples=8, threshold=Tr
plot_image(image)
```

In [ ]:
```
image = generator.generate_images(guidance=2, num_samples=8, threshold=Tr
plot_image(image)
```

In [ ]:
```
image = generator.generate_images(guidance=5, num_samples=8, threshold=Tr
plot_image(image)
```

Question: How do the results after thresholding compare to the results before thresholding? With thresholding and high guidance weight, which class specific features are most prominant? Which set of parameters gave you the best output?

Your Answer Here: I guess with thresholding it would perform better.

Unfortunatelly, I am running into CUDA errors, the most frequent being `RuntimeError: cuDNN error: CUDNN_STATUS_INTERNAL_ERROR`, therefore I can not see the images nor draw conclusions about it. Below are some of the things I have tried, but it is also important to note that I did try many different numbers for `self.condition_dim`, both in the int format and tuple of ints.

I am running locally, in VSCode and already tried restarting the kernel, closing and reopening the program, and even restarting my machine. Unfortunatelly I don't have time to dig any further in the web or talk about this in OH. I made a post on piazza about this, apparently some people had the same issue.

TODO: Experiment with different training times and guidance weights! Try and get the best image possible (or show something else interesting). You need at least two more generations, feel free to do more are welcome though.

In [ ]:
```
# room for experimentation
```

Collect Submission

Run the following cell to collect `assignment3_part2_submission.zip`. You will submit this to HW3 Code - Part 2 on gradescope. Make sure to also export a PDF of this jupyter notebook and attach that to the end of your theory section. This PDF must show your answers to all the questions in the document, please leave in all the photos that you generate as well. You will not be given credit for anything that is not visible to us in this PDF.

In [ ]:
```
%%bash collect_submission.sh
```

Contributers

- Matthew Bronars (Lead)
- Manav Agrawal
- Mihir Bafna