CS323 Project 4: Generative Models

Author: Hasan Abed Al Kader Hammoud, Guocheng Qian, Modar Alfadly, and Shuming Liu

Due Date: 25 April 2023 @ 11:59 PM

Student Name: David Felipe Alvear Goyes

KAUST ID: 187594

Degree: Electrical and Computer Engineering

Major: Robotics and autonomous systems

To setup a conda environment for this project, just run the following commands:

```
source $(conda info --base)/etc/profile.d/conda.sh
conda create -n cs323 python=3.9.2 -y
conda activate cs323

conda install pytorch=1.8.1 torchvision=0.9.1 torchaudio=0.8.1
cudatoolkit=11.1 -c pytorch -c conda-forge -y
conda install jupyter=1.0.0 -y  # to edit this file
conda install matplotlib=3.3.4 -y  # for plotting
conda install tqdm=4.59.0 -y  # for a nice progress bar
conda install tensorboard=2.4.0 -y  # to use tensorboard

pip install jupyter_http_over_ws  # for Google Colab
jupyter serverextension enable --py jupyter_http_over_ws  # Google
Colab
```

In the previous projects, you learned the basics of deep learning; data loading and processing along with model building, training, and evaluation. In the process, you learned how to tackle few computer vision applications; regression, classification, semantic segmentation, object detection, and video classification. However, all the models that we trained so far fall under the discriminative type (predict something given input data). We will introduce you in this project to generative models (learn the underlying data distribution or how to sample).

Before you start this project, you are highly encouraged to watch this introductory video and go through this notebook.

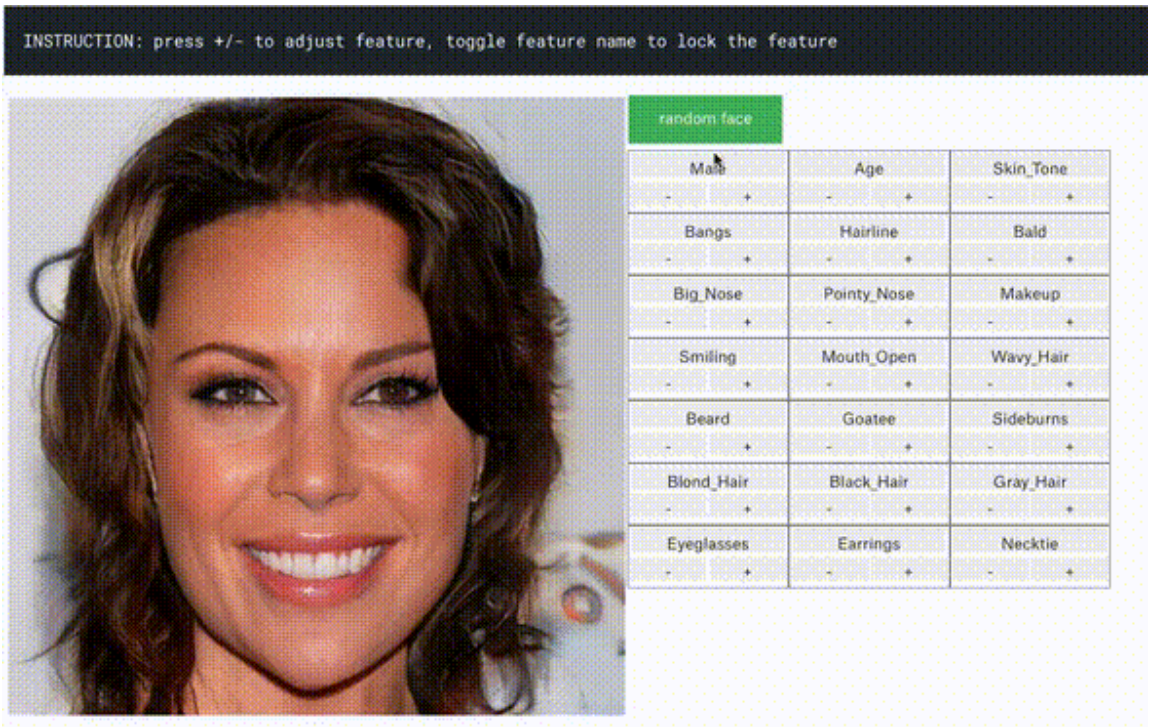
In [2]:
```
from pathlib import Path

from PIL import Image
import matplotlib.pyplot as plt
```

```
import tensorboard
from tqdm.notebook import tqdm

import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
# from torch.utils.tensorboardX import SummaryWriter
from tensorboardX import SummaryWriter

import torchvision
import torchvision.transforms as T
import torchvision.transforms.functional as TF
from torchvision.utils import make_grid
```

What do we mean by data distribution? Let's define it loosely here. The idea is to get an intuition instead of defining it fromally.

Think about all 200×200 RGB colored images. They have 40000 pixels each of which has 3 color channels that can be in the range $[0, 255]$. This means there is $256^{3 \times 200 \times 200} \approx 10^{288988}$ unique images. This number is ridiculously large; the estimated number of atoms in the observable universe is only between 10^{78} and 10^{82}. Now, let's think of all 200×200 RGB colored images that only have cats in them. These images are a subset of all images but they are still virtually infinite; there are hundreds of cat breeds, millions of cats in the world, different possible angles, lighting, and poses to take cat pictures. However, we can still recognize a cat image instantly since we have developed a mental image (an implicit representation) of what cats look like. We have few properties and characteristics in mind that define cats to us like color, breed, age, ..., etc. For example, it is not hard to imagine an orange tabby kitten. We call these properties latent representation.

This is an example of a generative model that was trained on human faces with a latent representation that can control 21 different properties:

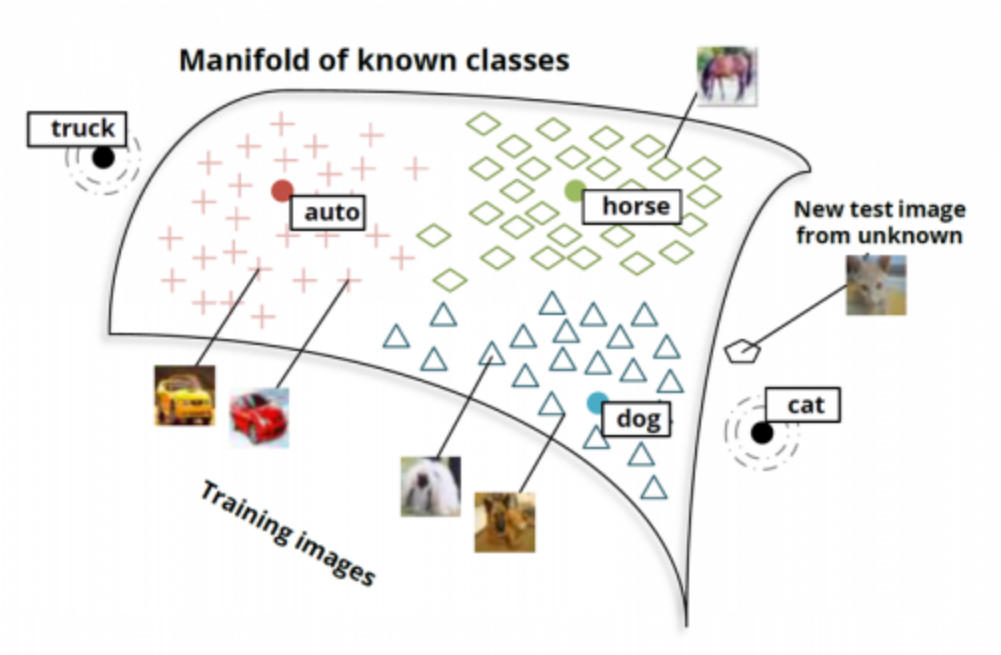


If we looked at the images as vectors (i.e., $\mathbf{x} \in [0,1]^{3\times 200\times 200}$), we will find that cat images are concentrated in few regions in this space. The probability $p(\mathbf{x})$ of finding a cat in any location is defined by the density of cat images in that location. If an image $\mathbf{x}$ is close to or surrounded by many cat images, we would say that the probability that $\mathbf{x}$ is a cat image is high ($p(\mathbf{x})$ is close to 1). Otherwise, the probability should be low ($p(\mathbf{x})$ is close to 0). This way we defined $p(\mathbf{x})$ as the cat distribution. We can also consider a latent vector $\mathbf{z}$ (encoding color, breed, age, ...) where it has a prior distribution $p(\mathbf{z})$. Similar intuition to $p(\mathbf{x})$ still applies to $p(\mathbf{z})$; cats are distributed in the latent space (e.g., different cat breeds live longer than others). It is much more convenient to work with the latent vector $\mathbf{z}$ than the original image $\mathbf{x}$ since all the information is distilled in a samller representation. A latent-based generative model is a model that tries to learn how to decode a latent vector to an image and vice versa (i.e., encode an image as a latent vector). The encoder takes an image $\mathbf{x}$ and gives you a latent vector $\mathbf{z}$ (color, breed, age, ...). The decoder takes a latent vector $\mathbf{z}$ to generate an image $\mathbf{x}$. Basically, we want a way to learn the joint distribution $p(\mathbf{x}, \mathbf{z})$.

$$p(\mathbf{x}, \mathbf{z}) = \overbrace{p(\mathbf{z}|\mathbf{x})}^{\text{encoder}} p(\mathbf{x}) = \overbrace{p(\mathbf{x}|\mathbf{z})}^{\text{decoder}} \underbrace{p(\mathbf{z})}_{\text{prior}}$$

The expansion is by Bayes rule. If we have an encoder and a decoder with the prior, we have modeled the data distribution $p(\mathbf{x})$. If I encoded a cat image and gave you the latent vector only like (a blue cat), you would know that it is unlikely since you know $p(\mathbf{z})$ which doesn't have many blue cats. I can also generate images by sampling (generating) a random latent vector and use the decoder to generate a new image.

To better understand the concepts of prior read the following blog Priors.



In [3]:
```python
class FFHQThumbnails(Dataset):
    """Flickr-Faces-HQ Dataset (FFHQ)

    download thumbnails128x128 from https://github.com/NVlabs/ffhq-dataset
    the dataset is available in Ibex (but the large version only 1024x1024)
    """
    def __init__(self, root_dir=None):
        if root_dir is None:
            root_dir = Path(torch.hub.get_dir()) / 'datasets/FFHQ'
        self.root_dir = Path(root_dir)
        if not (self.root_dir / 'thumbnails128x128').exists():
            msg = f'put the folder thumbnails128x128 in {self.root_dir}'
            raise FileNotFoundError(msg)
        self.images = list(self.root_dir.glob('thumbnails128x128/*.png'))
        assert len(self.images) == 70000, f'found {len(self.images)} images'

    def __getitem__(self, index):
        # there is not target we only need the image
        return TF.to_tensor(Image.open(self.images[index]))

    def __len__(self):
        return len(self.images)


dataset = FFHQThumbnails()
assert len(dataset) == 70000, 'did you download all the files?'
TF.to_pil_image(dataset[torch.randint(len(dataset), ())])
```

Out[3]:

Part 1: Variational Auto-Encoders (7 points)

Task 1: Model Construction

Variational Auto-Encoders (VAEs) are generative models that try to model the data density function (distribution) explicitly. They assume that $p(\mathbf{z})$ follows a normal distribution $\mathcal{N}(\mu, \mathbf{\Sigma})$ where μ is the mean and $\mathbf{\Sigma}$ is the covariance matrix (usually diagonal $\mathbf{\Sigma} = \text{diag}(\sigma)$), i.e the covariance matrix is isotropic. Then, they train a model $e(\mathbf{x}) = \hat{\mathbf{z}}$ to act as the encoder $p(\hat{\mathbf{z}}|\mathbf{x})$ and another model $d(\mathbf{z}) = \hat{\mathbf{x}}$ to act as the decoder $p(\hat{\mathbf{x}}|\mathbf{z})$. Our goal is to train the encoder and the decoder such that the distribution of the generated data $p(\hat{\mathbf{x}})$ gets as close as possible to the true data distribution $p(\mathbf{x})$. To achieve this, we need to minimize the distance between these to distributions. One common distribution distance is the KL-divergence.

$$\arg\min \text{KL}(p(\hat{\mathbf{x}}), p(\mathbf{x})) = \arg\max \mathbb{E}_{p(\mathbf{x})}[\log p(\hat{\mathbf{x}})]$$

where $\log p(\hat{\mathbf{x}})$ is the log-likelihood (evidence) which can be lower bounded by a term known as the **E**vidence **L**ower **BO**und (ELBO).

$$\log p(\hat{\mathbf{x}}) \geq \underbrace{\overbrace{\mathbb{E}_{p(\hat{\mathbf{z}}|\mathbf{x})}[\log p(\hat{\mathbf{x}}|\hat{\mathbf{z}})]}^{\text{reconstruction term}} - \overbrace{\text{KL}(p(\hat{\mathbf{z}}|\mathbf{x}), p(\mathbf{z}))}^{\text{prior regularizer}}}_{\text{ELBO}}$$

The reconstruction term is usually approximated with $-\frac{1}{2}\|\hat{\mathbf{x}} - \mathbf{x}\|_2^2 + \text{constant}$ where $\hat{\mathbf{x}}$ is predicted by the decoder.
The regularization term is estimated with $\text{KL}\left(\hat{\mathbf{z}}, \mathcal{N}(\mathbf{0}, \mathbf{I})\right)$ where $\hat{\mathbf{z}} \sim \mathcal{N}(\mu, \text{diag}(\sigma))$ such that μ and σ are predicted by the encoder.

$$\text{KL}\left(\mathcal{N}(\mu, \text{diag}(\sigma)), \mathcal{N}(\mathbf{0}, \mathbf{I})\right) = \frac{1}{2}\sum_i (\mu^2 + \sigma^2 - \log\sigma^2 - \mathbf{1})_i$$

In [4]:

```python
def kl_divergence_with_standard_normal(mean, std):
    p = torch.distributions.Normal(mean, std)
    q = torch.distributions.Normal(0, 1)
    return torch.distributions.kl_divergence(p, q)
```

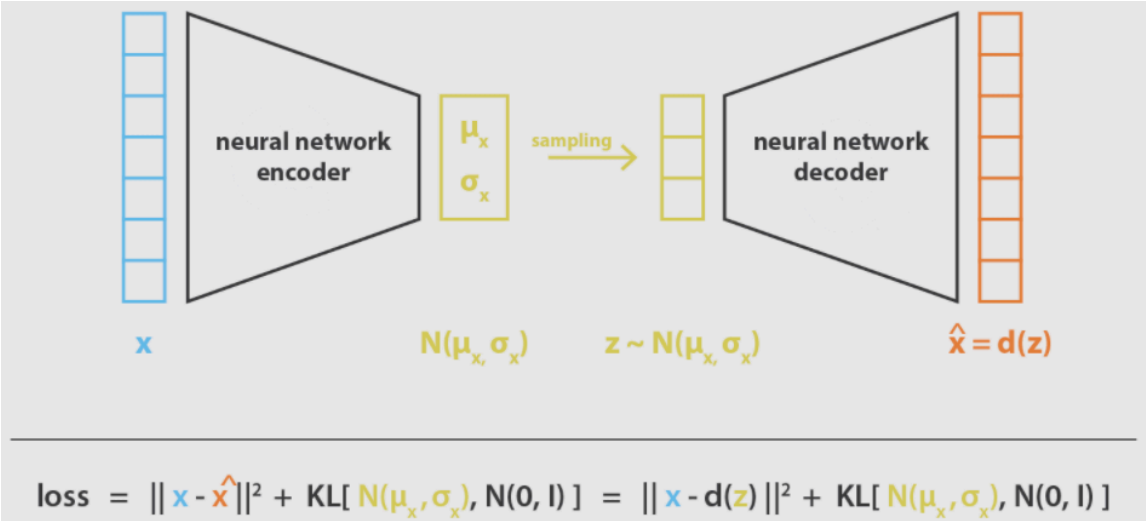
```python
mean, logvar = torch.randn(5), torch.randn(5)
std = (0.5 * logvar).exp()  # logvar = log(std**2)
kld = kl_divergence_with_standard_normal(mean, std)

# TODO: vvvvvvvvvvv (1 points)
# implement the kl_div element-wise (do not sum)
# use only mean and logvar (do not use std directly)
# out = 0.5 * (mean**2 + std**2 - 2 * std.log() - 1)
out = 0.5 * (mean**2 + logvar.exp() - logvar - 1)
# ^^^^^^^^^^^^^^^^^

print('KLD:', kld)
print('OUT:', out)
print('Same?', torch.allclose(kld, out))
```

```
KLD: tensor([0.3904, 2.9264, 1.0192, 2.3070, 2.0715])
OUT: tensor([0.3904, 2.9264, 1.0192, 2.3070, 2.0715])
Same? True
```

The final loss function looks like this (negative the ELBO without the constants):

$$\mathcal{L}_{\beta\text{-VAE}}(\mathbf{x}) = \overbrace{\underbrace{\|\hat{\mathbf{x}} - \mathbf{x}\|_2^2}_{\hat{\mathbf{x}}\to\mathbf{x}}}^{\text{reconstruction term}} + \beta \overbrace{\sum_i (\underbrace{\mu^2}_{\mu\to\mathbf{0}} + \underbrace{\sigma^2 - \log\sigma^2 - \mathbf{1}}_{\sigma^2\to\mathbf{1}})_i}^{\text{regularization term}}$$

where β is the trade-off coefficient between reconstruction and regularization.



Implementing VAEs is very easy. Check this example out. If you are looking for VAE variants, check this and this instead. The encoder is a network that takes an image as inputs and gives two feature vectors as output (μ and $\log\sigma^2$). We use $\log$ for the variance because we know that it is always positive (similar to what we did for the width and height for bounding boxes in YOLOv3). We will use ResNet-18 as the encoder without the last linear layer. The feature size is 512, we will split it in half and use the first 256 features as μ and the second 256 features as $\log\sigma^2$. Then, we need to sample a latent vector from the normal distribution $\hat{\mathbf{z}} \sim \mathcal{N}(\mu, \text{diag}(\sigma))$ using a simple

reparametrization trick $\hat{\mathbf{z}} = \mu + \sigma\varepsilon$ where $\varepsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$. This trick is important in order be able to backpropagate the gradients back to the encoder. Finally, we use a decoder network that generate an image given a latent vector. It is worth noting that VAEs still care about the mean squared error between the generated image and the real one (as in autoencoders), therefore the ideal output of a VAE is the average image over all plausible ones.

In [5]:

```
# TODO: Understand the following implementation and how does this compare to
"""
Compares in:
obj_function, encoder, decoder, reparam trick
"""
class VAE(nn.Module):
    def __init__(self):
        super().__init__()
        resnet = torchvision.models.resnet18()
        resnet.fc = nn.Identity()
        self.encoder = resnet
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(256, 512, 3, stride=2),
            nn.LeakyReLU(inplace=True),
            nn.ConvTranspose2d(512, 256, 3, stride=2),
            nn.LeakyReLU(inplace=True),
            nn.ConvTranspose2d(256, 128, 3, stride=2),
            nn.LeakyReLU(inplace=True),
            nn.ConvTranspose2d(128, 64, 3, stride=2),
            nn.LeakyReLU(inplace=True),
            nn.ConvTranspose2d(64, 32, 3, stride=2),
            nn.LeakyReLU(inplace=True),
            nn.ConvTranspose2d(32, 32, 3, stride=2, output_padding=1),
            nn.LeakyReLU(inplace=True),
            nn.Conv2d(32, 16, 5, padding=2),
            nn.LeakyReLU(inplace=True),
            nn.Conv2d(16, 3, 3, padding=1),
        )

    def forward(self, inputs):
        mean, logvar = self.encode(inputs)
        latent_vector = self.sample_normal(mean, logvar)
#         print(f"latent_vector: {latent_vector.shape}")
        outputs = self.decode(latent_vector)
        return outputs, mean, logvar

    def encode(self, images):
        # TODO: vvvvvvvvvvv
        # encode the image and split the output features
        encoder_output = self.encoder(images) # size [2, 512]
#         print(f"encoder_output: {encoder_output.shape}")
        mean, logvar = torch.chunk(encoder_output, chunks=2, dim=1) # [2, 25
#         print(f"mean: {mean.shape}")
#         print(f"logvar: {logvar.shape}")
        # ^^^^^^^^^^^^^^^^
        return mean, logvar
```

```
    @staticmethod
    def sample_normal(mean, logvar):
        # TODO: vvvvvvvvvvv
        # implement and understand the reparameterization trick
        std = torch.exp(0.5*logvar) # calculate the std using logvar
        epsilon = torch.randn_like(std) # calculate the epsilon
        z_hat = mean + std*epsilon #
        # ^^^^^^^^^^^^^^^^
        return z_hat

    def decode(self, latent_vector):
        # TODO: vvvvvvvvvvv
        # decode the latent_vector to an image
        latent_vector = latent_vector.view(-1, 256, 1, 1) # [B,c,h,w] resize
#         print(f"latent_vector: {latent_vector.shape}")
        x_hat = self.decoder(latent_vector)
#         print(f"x_hat: {x_hat.shape}\n")
        # ^^^^^^^^^^^^^^^^
        return x_hat


model = VAE()
x = torch.rand(2, 3, 128, 128)
outputs, mean, logvar = model(x)
print(mean.shape)
print(logvar.shape)
print(outputs.shape)
```

```
torch.Size([2, 256])
torch.Size([2, 256])
torch.Size([2, 3, 128, 128])
```

Task 2: Training and Evaluation

Let's write our training function but add to it tensorboard logging support.

In [7]:
```
# https://www.tensorflow.org/tensorboard/tensorboard_in_notebooks
log_dir = Path('./runs')
log_dir.mkdir(exist_ok=True)
tensorboard.notebook.start(f'--logdir={log_dir} --bind_all')

# if you don't see tensorboard here, you can open it in a new browser tab
# in rare cases, you might need to consider using ngrok: https://ngrok.com/
tensorboard.notebook.list()
!hostname -I  # use this ip with port above
```

Index of /

Name	Size	Date Modified
.vol/		5/7/24, 10:01:44 AM
Applications/		9/15/24, 9:22:05 AM
bin/		5/7/24, 10:01:44 AM
cores/		3/3/21, 1:24:44 PM
dev/		9/8/24, 9:33:19 AM
etc/		6/26/24, 7:39:58 PM
home/		9/8/24, 9:33:55 AM
Library/		6/26/24, 7:41:24 PM
opt/		5/17/23, 1:51:46 PM
private/		9/8/24, 9:33:35 AM
sbin/		5/7/24, 10:01:44 AM
System/		5/7/24, 10:01:44 AM
tmp/		9/15/24, 10:58:56 AM
Users/		6/26/24, 7:38:33 PM
usr/		5/7/24, 10:01:44 AM
var/		6/26/24, 7:39:48 PM
Volumes/		9/15/24, 10:41:30 AM
.file	0 B	5/7/24, 10:01:44 AM

```
Known TensorBoard instances:
  - port 6006: logdir runs (started 0:00:00 ago; pid 2915678)
172.17.0.4
```

In [8]:
```python
def train(model, loader, optimizer, device, beta=1, epochs=5, show_every=50)
    model.train(True)
    num_batches = len(loader)
    writer = SummaryWriter(log_dir=None)  # use the default log_dir=./runs
    for epoch in range(epochs):
```

```
        loss_sum = count = 0
        for i, images in enumerate(tqdm(loader), 1):
            images = images.to(device)
            outputs, mean, logvar = model(images)
            # assuming Gaussian:
            recon_term = F.mse_loss(outputs.sigmoid(), images)
            # assuming Bernoulli:
            # recon_term = F.binary_cross_entropy_with_logits(outputs, image

            # constants don't affect the gradients (change the learning rate
            reg_term = (mean**2 + logvar.exp() - logvar).mean() - 1

            # TODO: vvvvvvvvvvv (0.5 points)
            # why do we multiply beta by this factor? (check the loss formul
            # factor = latent_dim / image_size
            # normalize reg_term to be at same magnitude order of recon_term
            factor = mean.shape[-1] / images.shape[1:].numel()
            # ^^^^^^^^^^^^^^^^^
            loss = recon_term + (beta * factor) * reg_term

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            count += len(images)
            loss_sum += recon_term.item() * len(images)
            step = epoch * num_batches + i - 1
            writer.add_scalar('loss/train', loss.item(), step)
            writer.add_scalar('recon_term/train', recon_term.item(), step)
            writer.add_scalar('reg_term/train', reg_term.item(), step)
            last_iteration = i == len(loader)
            if i % show_every == 0 or last_iteration:
                fig = plt.figure()
                ax = fig.add_subplot()
                grid = torch.stack([images[:8], outputs.data[:8].sigmoid()])
                grid = grid.transpose(0, 1).flatten(0, 1)
                grid = torchvision.utils.make_grid(grid)
                writer.add_image('images', grid, step)
                ax.imshow(TF.to_pil_image(grid))
                ax.set_title(f'Epoch {epoch}: Loss = {loss_sum / count:.5f}'
                ax.axis('off')
                plt.show(fig)
        # writer.add_graph(model.eval(), images)
        writer.close()
```

In [8]:

```
device = torch.device('cuda:0')
loader = DataLoader(
    dataset,
    batch_size=128,
    shuffle=True,
    num_workers=4,
    pin_memory=device.type == 'cuda',
    drop_last=True,
)
model = VAE().to(device)
```

```
optimizer = torch.optim.Adam(model.parameters())
# train(model, loader, optimizer, device, beta=1, epochs=5, show_every=50)
```

Hopefully, after some tinkering you got the training loss to reach a point where you can see reasonable results. However, is the loss the best metric we can use to evalute our generative model? Well, other metrics exists such as the Frechet distance. The idea behind it is very simple. It uses the Frechet distance between two Multivariate Gaussian distributions. The first distribution is inferred from the dataset while the second is from generated samples of our model. Instead of working on raw images we usually work with extracted features of these images (a common backbone is the inception model; see FID score). We will use ResNet-18 here instead.

In [9]:
```
class FrechetDistance:
    """Frechet's distance between two multi-variate Gaussians
    https://www.sciencedirect.com/science/article/pii/0047259X8290077X
    """
    def __init__(self, double=True, num_iterations=20, eps=1e-12):
        self.eps = eps
        self.double = double
        self.num_iterations = num_iterations

    def __call__(self, normal1, normal2):
        # make sure that both of them have unbiased set the same
        mu1, sigma1 = normal1.mean, normal1.covariance_matrix
        mu2, sigma2 = normal2.mean, normal2.covariance_matrix
        return self.compute(mu1, sigma1, mu2, sigma2)

    def compute(self, mu1, sigma1, mu2, sigma2):
        """Compute Frechet's distance between two multi-variate Gaussians
        https://gist.github.com/ModarTensai/185ca53b35b012c7fe781e4c567378a6
        """
        norm_2 = (mu1 - mu2).norm(2, dim=-1).pow(2)
        trace1 = sigma1.diagonal(0, -1, -2).sum(-1)
        trace2 = sigma2.diagonal(0, -1, -2).sum(-1)
        sigma3 = self.psd_matrix_sqrt(sigma1 @ sigma2)
        trace3 = sigma3.diagonal(0, -1, -2).sum(-1)
        return norm_2 + trace1 + trace2 - 2 * trace3

    def psd_matrix_sqrt(self, matrix):
        """Compute the square root of a PSD matrix using Newton's method
        https://gist.github.com/ModarTensai/7c4aeb3d75bf1e0ab99b24cf2b3b37a3
        """
        dtype = matrix.dtype
        if self.double:
            matrix = matrix.double()
        norm = matrix.norm(dim=[-2, -1], keepdim=True).clamp_min_(self.eps)
        matrix = matrix / norm

        def mul_diag_add(inputs, scale=-0.5, diag=1.5):
            # multiply by a scalar then add a scalar to the diagonal
            inputs.mul_(scale).diagonal(0, -1, -2).add_(diag)
            return inputs
```

```
        other = mul_diag_add(matrix.clone())  # avoid inplace
        matrix = matrix @ other
        for i in range(1, self.num_iterations):
            temp = mul_diag_add(other @ matrix)
            matrix = matrix @ temp
            if i + 1 < self.num_iterations:  # skip last step
                other = temp @ other
        return (matrix * norm.sqrt()).to(dtype)
```

Let's create a class that can accumulate the statistics of a stream of batches; the first two moments (mean μ and covariance matrix $\mathbf{\Sigma}$).

```
gaussian = MultivariateNormal(feature_size=5)

data = torch.randn(2000, gaussian.feature_size)
for batch in data.chunk(100):
    gaussian(batch)  # accumulate stats over the batches

# compute the stats on the entire data
mean = data.mean(0)
covariance_matrix = (data - mean).T.conj() @ (data - mean) /
(len(data) - 1)

# compare the streamed stats to the full stats
print(torch.allclose(gaussian.mean, mean))
print(torch.allclose(gaussian.covariance_matrix,
covariance_matrix))
```

In [10]:

```
class MultivariateNormal(nn.Module):
    """Multivariate normal (also called Gaussian) distribution
    https://gist.github.com/ModarTensai/185ca53b35b012c7fe781e4c567378a6
    """
    def __init__(self, feature_size, unbiased=True):
        super().__init__()
        self.count = 0
        self.unbiased = bool(unbiased)
        self.feature_size = feature_size
        mean = torch.zeros(self.feature_size)
        mass = torch.zeros(self.feature_size, self.feature_size)
        self.register_buffer('mean', mean)
        self.register_buffer('mass', mass)

    @property
    def factor(self):
        """Get the normalization factor"""
        return 1 / (self.count - int(bool(self.unbiased)))

    @property
    def covariance_matrix(self):
        """Get the covariance matrix"""
        return self.mass * self.factor

    @property
```

```
    def variance(self):
        """Get the variance."""
        return self.mass.diag() * self.factor

    def forward(self, batch):
        """Perform the forward pass (only update in training mode)"""
        mean, covariance, count = self.get_stats(batch, self.unbiased)
        if self.training:
            self.stats_update(mean, covariance, count, self.unbiased)
        return mean, covariance

    @staticmethod
    def get_stats(batch, unbiased=True):
        """Compute the statistics of a batch
        https://gist.github.com/ModarTensai/5ab449acba9df1a26c12060240773110
        """
        assert 1 <= batch.ndim <= 2
        if batch.ndim == 1:
            batch.unsqueeze(0)
        count = batch.shape[0]
        mean = batch.mean(0)
        if count == 1:
            covariance = None
        else:
            batch = batch - mean
            factor = 1 / (count - int(bool(unbiased)))
            covariance = factor * batch.t().conj() @ batch
        return mean, covariance, count

    def stats_update(self, mean, covariance, count, unbiased=None):
        """Update the model given batch statistics
        https://gist.github.com/ModarTensai/dc95444faf3624ed979b4d0b2088fdf1
        """
        diff1 = mean - self.mean
        self.mean += diff1 * (count / (self.count + count))
        diff2 = mean - self.mean
        mass = diff1[:, None] @ diff2[None, :]
        if count > 1:
            mass += covariance
            mass *= count
            if unbiased is None:
                unbiased = self.unbiased
            if unbiased:
                mass -= covariance
        self.mass += mass
        self.count += count
```

We will use `MultivariateNormal` to fit a multivariate Gaussian distribution $\mathcal{N}(\mu, \mathbf{\Sigma})$ to our generated images. We will also do the same and fit another multivariate Gaussian distribution to the ground truth dataset. Finally, we will compute the Frechet distance between them.

In [11]:
```
def evaluate(model, loader, device):
    model.train(False)
```

```
    batch_size = loader.batch_size
    latent_dim = model.decoder[0].in_channels

    # get the distance and feature extractor
    frechet_distance = FrechetDistance()
    feature_extractor = torchvision.models.resnet18(pretrained=True)
    feature_extractor.to(device).eval().requires_grad_(False)
    feature_size = feature_extractor.fc.in_features
    feature_extractor.fc = nn.Identity()

    # compute the stats of the dataset
    dataset_distribution = MultivariateNormal(feature_size).to(device)
    # with torch.no_grad():  # is this needed?
    for image_batch in tqdm(loader):
        features = feature_extractor(image_batch.to(device))
        dataset_distribution(features)

    # compute the stats of the model
    model_distribution = MultivariateNormal(feature_size).to(device)
    with torch.no_grad():
        for _ in tqdm(range(len(loader))):
            latent_vector = torch.randn(batch_size, latent_dim, device=devic
            image_batch = model.decode(latent_vector)
            features = feature_extractor(image_batch)
            model_distribution(features)

    return frechet_distance(model_distribution, dataset_distribution)


distance = evaluate(model, loader, device)
print(f'Frechet ResNet18 Distance = {distance:.3f} (lower is better)')
```

```
/home/ubuntu/.local/lib/python3.8/site-packages/torchvision/models/_utils.p
y:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and
may be removed in the future, please use 'weights' instead.
  warnings.warn(
/home/ubuntu/.local/lib/python3.8/site-packages/torchvision/models/_utils.p
y:223: UserWarning: Arguments other than a weight enum or `None` for 'weight
s' are deprecated since 0.13 and may be removed in the future. The current b
ehavior is equivalent to passing `weights=ResNet18_Weights.IMAGENET1K_V1`. Y
ou can also use `weights=ResNet18_Weights.DEFAULT` to get the most up-to-dat
e weights.
  warnings.warn(msg)
  0%|          | 0/546 [00:00<?, ?it/s]
  0%|          | 0/546 [00:00<?, ?it/s]
Frechet ResNet18 Distance = 1047.000 (lower is better)
```

Now, it is your turn to put these utilities in use.

In [12]:
```
# TODO: vvvvvvvvvvv (2 points)
# copy `train()` here and do the following changes:
# - print the Frechet distance (FD) at every epoch similar to `evaluate()`
# - don't use `evaluate()` directly as it is inefficient
#   you don't need to load the feature extractor at each epoch
#   make sure to compute `dataset_distribution` only once (when `epoch == 0`
# - compute `vae_distribution` on the reconstructed images instead
```

```
#   this is for simplicity; in practice, we should use generated images
#   generated_images = decoder(random_latent_vector)
#   reconstructed_images = decoder(sample(encoder(real_image)))
# - be careful to set the models in eval and train mode appropriately if nee
# - avoid gradients if required to prevent memory leaks (out of memory error
def train(model, loader, optimizer, device, beta=1, epochs=5, show_every=50)
    model.train(True)
    num_batches = len(loader)
    writer = SummaryWriter(log_dir=None)  # use the default log_dir=./runs

    # get the distance and feature extractor
    frechet_distance = FrechetDistance()
    feature_extractor = torchvision.models.resnet18(pretrained=True)
    feature_extractor.to(device).eval().requires_grad_(False)
    feature_size = feature_extractor.fc.in_features
    feature_extractor.fc = nn.Identity()

    # compute the stats of the dataset
    print("Compute the dataset distribution")
    dataset_distribution = MultivariateNormal(feature_size).to(device)
    for image_batch in tqdm(loader):
        features = feature_extractor(image_batch.to(device))
        dataset_distribution(features)

    print("Begin training")
    for epoch in range(epochs):
        loss_sum = count = 0
        vae_distribution = MultivariateNormal(feature_size).to(device)
        for i, images in enumerate(tqdm(loader), 1):
            images = images.to(device)
            outputs, mean, logvar = model(images)
            # assuming Gaussian:
            recon_term = F.mse_loss(outputs.sigmoid(), images)
            # assuming Bernoulli:
            # recon_term = F.binary_cross_entropy_with_logits(outputs, image

            # compute the features for update the vae_distribution
            with torch.no_grad():
                features = feature_extractor(outputs.sigmoid())
                vae_distribution(features)

            # constants don't affect the gradients (change the learning rate
            reg_term = (mean**2 + logvar.exp() - logvar).mean() - 1

            # TODO: vvvvvvvvvvv (0.5 points)
            # why do we multiply beta by this factor? (check the loss formul
            # factor = latent_dim / image_size
            # normalize reg_term to be at same magnitude order of recon_term
            factor = mean.shape[-1] / images.shape[1:].numel()
            # ^^^^^^^^^^^^^^^^^
            loss = recon_term + (beta * factor) * reg_term

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

```
            count += len(images)
            loss_sum += recon_term.item() * len(images)
            step = epoch * num_batches + i - 1
            writer.add_scalar('loss/train', loss.item(), step)
            writer.add_scalar('recon_term/train', recon_term.item(), step)
            writer.add_scalar('reg_term/train', reg_term.item(), step)
            last_iteration = i == len(loader)
            if i % show_every == 0 or last_iteration:
                fig = plt.figure()
                ax = fig.add_subplot()
                grid = torch.stack([images[:8], outputs.data[:8].sigmoid()])
                grid = grid.transpose(0, 1).flatten(0, 1)
                grid = torchvision.utils.make_grid(grid)
                writer.add_image('images', grid, step)
                ax.imshow(TF.to_pil_image(grid))
                ax.set_title(f'Epoch {epoch}: Loss = {loss_sum / count:.5f}'
                ax.axis('off')
                plt.show(fig)

        # print frechet distance every epoch
        distance = frechet_distance(vae_distribution, dataset_distribution)
        print(f'Frechet ResNet18 Distance = {distance:.3f} (lower is better)
        # writer.add_graph(model.eval(), images)
        writer.close()
# ^^^^^^^^^^^^^^^^^

train(model, loader, optimizer, device, beta=1, epochs=5, show_every=50)
```

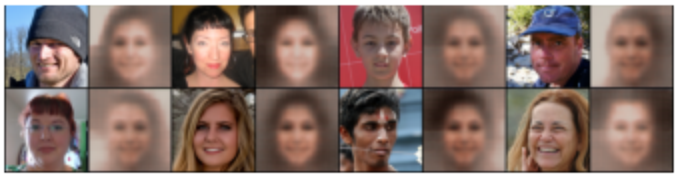
```
Compute the dataset distribution
  0%|          | 0/546 [00:00<?, ?it/s]
Begin training
  0%|          | 0/546 [00:00<?, ?it/s]
```

Epoch 0: Loss = 0.07150

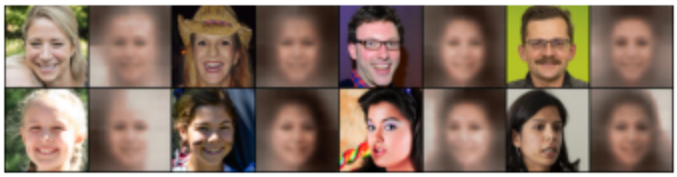
Epoch 0: Loss = 0.06845

Epoch 0: Loss = 0.06129

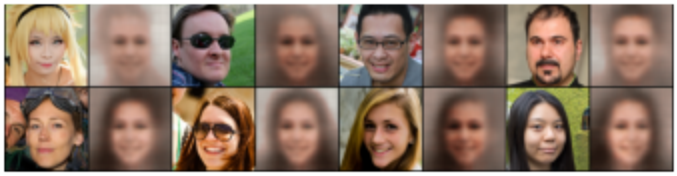
Epoch 0: Loss = 0.05601



Epoch 0: Loss = 0.05221

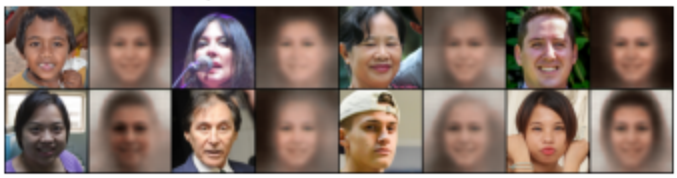


Epoch 0: Loss = 0.04923



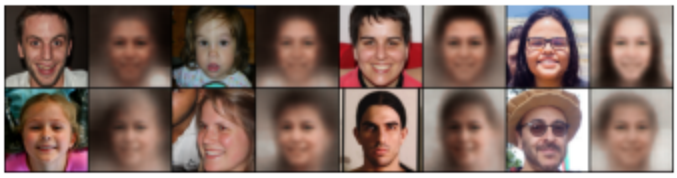
Epoch 0: Loss = 0.04694

Epoch 0: Loss = 0.04515



Epoch 0: Loss = 0.04360

Epoch 0: Loss = 0.04232

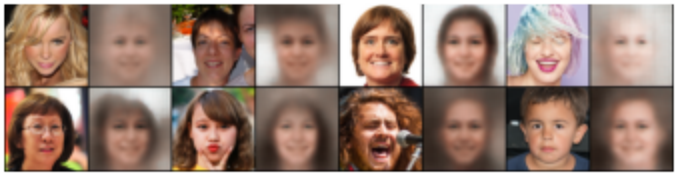


Epoch 0: Loss = 0.04130

Frechet ResNet18 Distance = 763.134 (lower is better)

0%| | 0/546 [00:00<?, ?it/s]

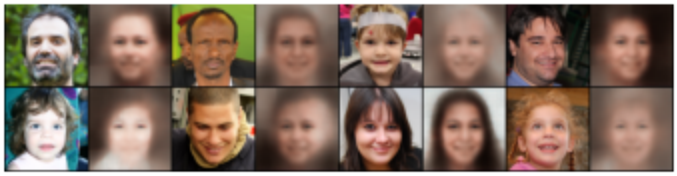
Epoch 1: Loss = 0.02959



Epoch 1: Loss = 0.02946

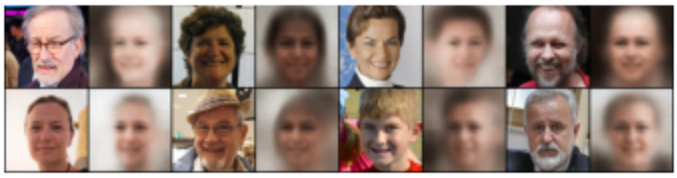
Epoch 1: Loss = 0.02933

Epoch 1: Loss = 0.02914



Epoch 1: Loss = 0.02894

Epoch 1: Loss = 0.02880



Epoch 1: Loss = 0.02865

Epoch 1: Loss = 0.02851

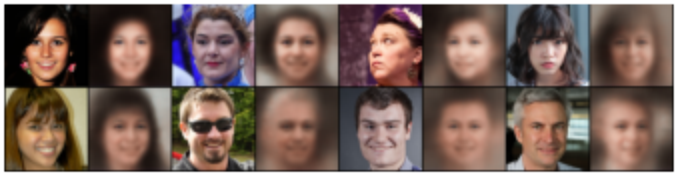
Epoch 1: Loss = 0.02841

Epoch 1: Loss = 0.02832

Epoch 1: Loss = 0.02820

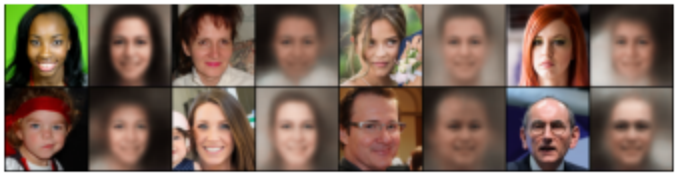
```
Frechet ResNet18 Distance = 613.648 (lower is better)
  0%|          | 0/546 [00:00<?, ?it/s]
```

Epoch 2: Loss = 0.02710



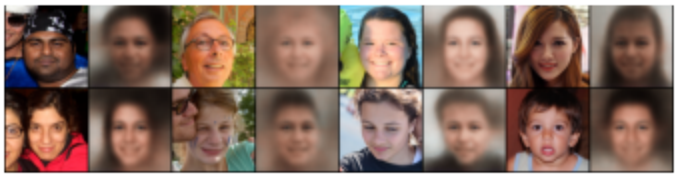
Epoch 2: Loss = 0.02686

Epoch 2: Loss = 0.02675



Epoch 2: Loss = 0.02664

Epoch 2: Loss = 0.02652



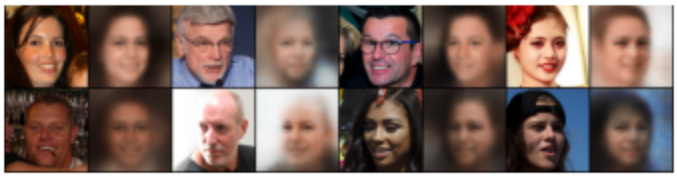
Epoch 2: Loss = 0.02650

Epoch 2: Loss = 0.02640

Epoch 2: Loss = 0.02629

Epoch 2: Loss = 0.02622

Epoch 2: Loss = 0.02614

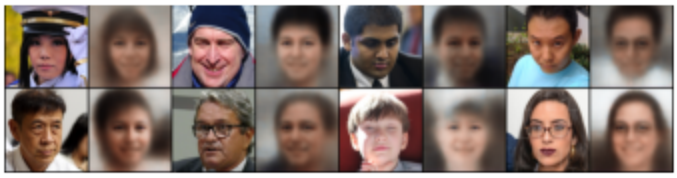


Epoch 2: Loss = 0.02603

Frechet ResNet18 Distance = 579.916 (lower is better)

  0%| | 0/546 [00:00<?, ?it/s]

Epoch 3: Loss = 0.02489



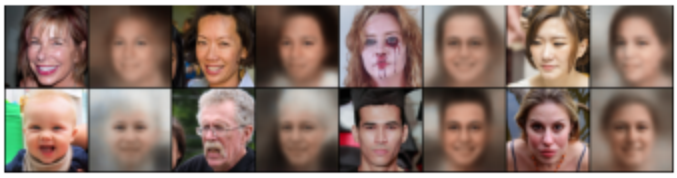
Epoch 3: Loss = 0.02465

Epoch 3: Loss = 0.02474

Epoch 3: Loss = 0.02474

Epoch 3: Loss = 0.02473

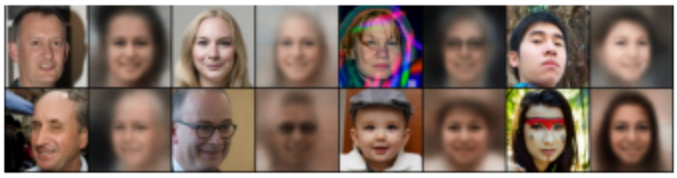
Epoch 3: Loss = 0.02465



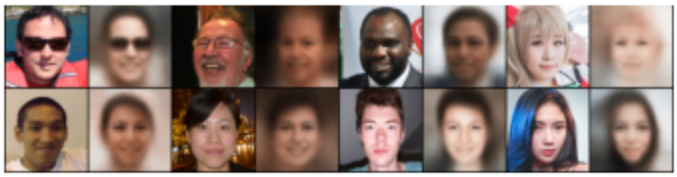
Epoch 3: Loss = 0.02459

Epoch 3: Loss = 0.02453

Epoch 3: Loss = 0.02450



Epoch 3: Loss = 0.02446



Epoch 3: Loss = 0.02441

```
Frechet ResNet18 Distance = 562.802 (lower is better)
  0%|          | 0/546 [00:00<?, ?it/s]
```

Epoch 4: Loss = 0.02384

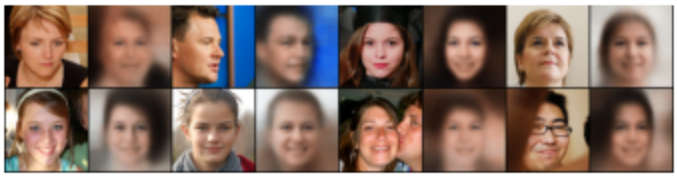
Epoch 4: Loss = 0.02390

Epoch 4: Loss = 0.02375

Epoch 4: Loss = 0.02378

Epoch 4: Loss = 0.02372

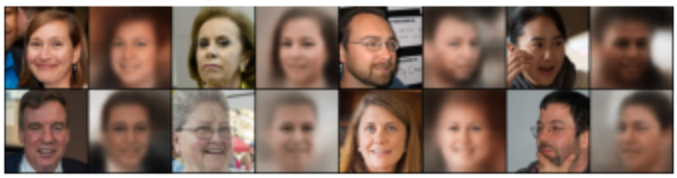
Epoch 4: Loss = 0.02366



Epoch 4: Loss = 0.02362

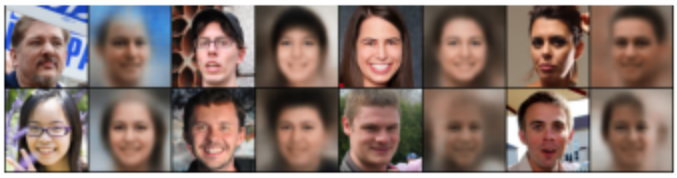
Epoch 4: Loss = 0.02359

Epoch 4: Loss = 0.02356



Epoch 4: Loss = 0.02352

Epoch 4: Loss = 0.02350



```
Frechet ResNet18 Distance = 545.126 (lower is better)
```

In [13]:
```
torch.cuda.empty_cache()  # Clear GPU memory cache
```

Check the state-of-the-art (SOTA) results on a bigger version of this dataset.

Task 3: Visualization

Let's take two images and visualize few interpolation steps between them in the latent space.

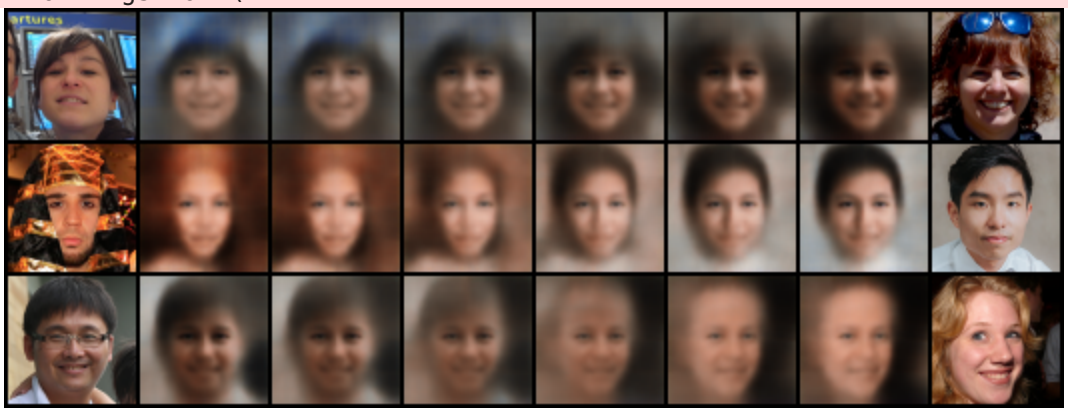
In [14]:
```
@torch.no_grad()
def sweep(model, images, count=8, spherical=False, stitch=True, grid=True):
    """Generate interpolations between images in the latent space"""
    assert len(images) % 2 == 0, 'should be an even number of images'
    model.train(False)
    latent = model.sample_normal(*model.encode(images))

    count -= 2 if stitch else 0
    latent = interpolate(*latent.chunk(2), count, spherical)
    output = model.decode(latent.flatten(0, 1)).sigmoid()
    output = TF.resize(output, output.shape[-1] // 2)
    output = output.view(-1, count, *output.shape[1:])
    count += 2 if stitch else 0

    if stitch:
        images = TF.resize(images, images.shape)
```

In [15]:
```
#@title  { run: "auto", vertical-output: true, display-mode: "both" }
def interpolate(x, y, count, spherical=False):
    """Interpolate between two feature vectors.

    Args:
        x: tensor of shape [N, F]
        y: tensor of shape [N, F]
        count: number of interpolation steps
        spherical: whether to do spherical or linear interpolation

    Returns:
        tensor of shape [N, count, F]
    """
    alphas = torch.linspace(0, 1, count, device=x.device)
    if spherical:
        # https://en.wikipedia.org/wiki/Slerp
        omega = torch.cosine_similarity(x, y, dim=-1).arccos().unsqueeze(-1)
        factor = lambda x: (x * omega).sin()
        out = torch.stack([factor(1 - a) * x + factor(a) * y for a in alphas
        out = out / omega.sin()
    else:
        # https://en.wikipedia.org/wiki/Linear_interpolation
        # TODO: Understand the solve the part below (0.5 points)
        # apply linear interpolation between x and y given alphas
        # you can either do it manually and stack or use torch.lerp
#         out = torch.stack([(1 - a) * x + a * y for a in alphas])
        out = torch.stack([torch.lerp(x, y, a) for a in alphas])
        # ^^^^^^^^^^^^^^^^^
    return out.transpose(0, 1)
```

```
@torch.no_grad()
def sweep(model, images, count=8, spherical=False, stitch=True, grid=True):
    """Generate interpolations between images in the latent space"""
    assert len(images) % 2 == 0, 'should be an even number of images'
    model.train(False)
    latent = model.sample_normal(*model.encode(images))

    count -= 2 if stitch else 0
    latent = interpolate(*latent.chunk(2), count, spherical)
    output = model.decode(latent.flatten(0, 1)).sigmoid()
    output = TF.resize(output, output.shape[-1] // 2)
    output = output.view(-1, count, *output.shape[1:])
    count += 2 if stitch else 0

    if stitch:
        images = TF.resize(images, images.shape[-1] // 2)
        images = images.view(2, -1, *images.shape[1:]).transpose(0, 1)
        start, end = images.chunk(2, dim=1)
        output = torch.cat([start, output, end], dim=1)

    if grid:
        output = torchvision.utils.make_grid(output.flatten(0, 1), count)
    return output


num_rows = 3 #@param {type:"slider", min:1, max:10, step:1}
indices = torch.randint(len(dataset), (2 * num_rows,))
images = torch.stack([dataset[i] for i in indices]).to(device)

num_columns = 8 #@param {type:"slider", min:1, max:10, step:1}
TF.to_pil_image(sweep(model, images, num_columns, spherical=False))
```

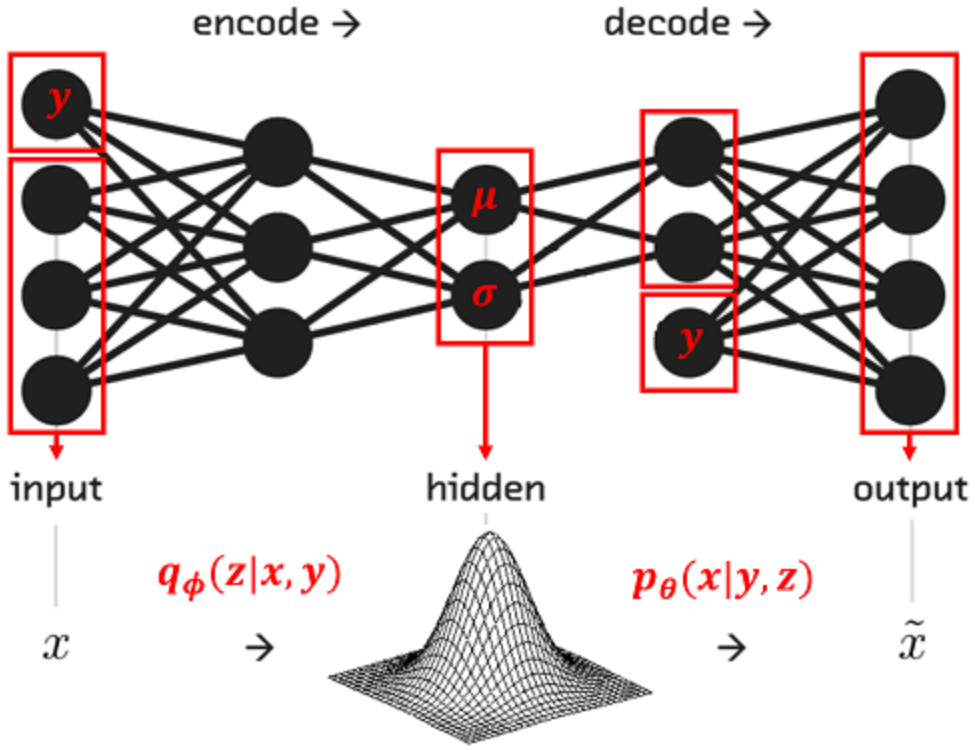
```
/home/ubuntu/.local/lib/python3.8/site-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the antialias parameter of all the resizing transforms (Resize(), RandomResizedCrop(), etc.) will change from None to True in v0.17, in order to be consistent across the PIL and Tensor backends. To suppress this warning, directly pass antialias=True (recommended, future default), antialias=None (current default, which means False for Tensors and True for PIL), or antialias=False (only works on Tensors - PIL will still use antialiasing). This also applies if you are using the inference transforms from the models weights: update the call to weights.transforms(antialias=True).
  warnings.warn(
```

Out[15]:



Conditional VAEs

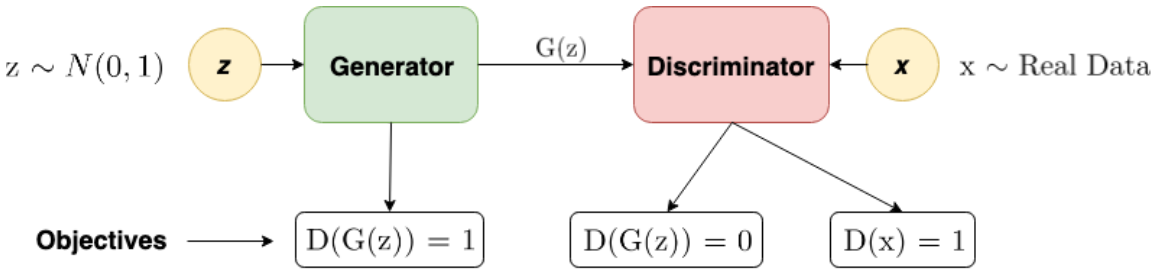
So far, we don't have control over the output and we don't know what every feature in the latent vector mean. We can investigate the latent space by sampling and then increasing/decreasing the value of every feature (out of the 256 features) independently and see how this affects the generated image. But this is after the fact. Can we somehow have a finer control over this? Read up on disentangled representation learning. Another common trick is to modify our model to be **conditional**. Consider passing a secondary condition $\mathbf{y}$ (e.g., label) as input to the encoder and the decoder. We define what this condition represent (e.g., female/male, young/old, ...) but it comes at a cost of extra supervision (we need to know these details about the images). The final model is now called a Conditional Variational Auto-Encoder (CVAE).



Part 2: Generative Adversarial Networks (13 points)

Another type of latent-based generative models is the Generative Adversarial Network (GAN). Unlike VAEs, GANs don't model the data distribution explicitly. They don't have an encoder (image $\rightarrow$ latent vector). Instead, they use a discriminator model $D(\mathbf{x})$ that guide the decoder (here called the generator $G(\mathbf{z})$) to generate samples that are closer to the real distribution. The discriminator is just a binary image classifer with a single scalar output classifying whether an input image is real or fake (generated). So, the generator starts at the begining of training with generating random images and the

discriminator tells it how realistic are they. To train the discriminator to differentiate real from fake, we will use the generated images as fake samples and images from the dataset as real samples then train it with them. This process will repeat iteratively (i.e., train the discriminator with real and fake samples for few iterations then train the generator using the discriminator). One way to look at this is to view them as adversaries (enemies) that try to outsmart each other (hence, the name).



The final objective can be written as a min-max problem:

$$\min_G \max_D \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$$

We can split this into two loss functions; one for the generator and the other for the discriminator:

$$\mathcal{L}_G^{\text{GAN}}(\mathbf{z}) = \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))] \tag{1}$$
$$\mathcal{L}_D^{\text{GAN}}(\mathbf{x}) = -\mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})}[\log D(\mathbf{x})] - \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))] \tag{2}$$

Assuming the output of $D(\mathbf{x})$ is in the range $[0, 1]$ (e.g., uses Sigmoid) and without loss of generality, the generator wants to fool the discriminator (i.e., $D(G(\mathbf{z}))$ should be close to 1) while the discriminator wants to differentiate between real and fake images (i.e., $D(x)$ shoule be close to 1 while $D(G(\mathbf{z}))$ should be close to 0). This is the first time where we needed to modify our training procedure manually.

Study this example to see how they are implemented in practice. There are other GAN variants out there you can check some of them here.

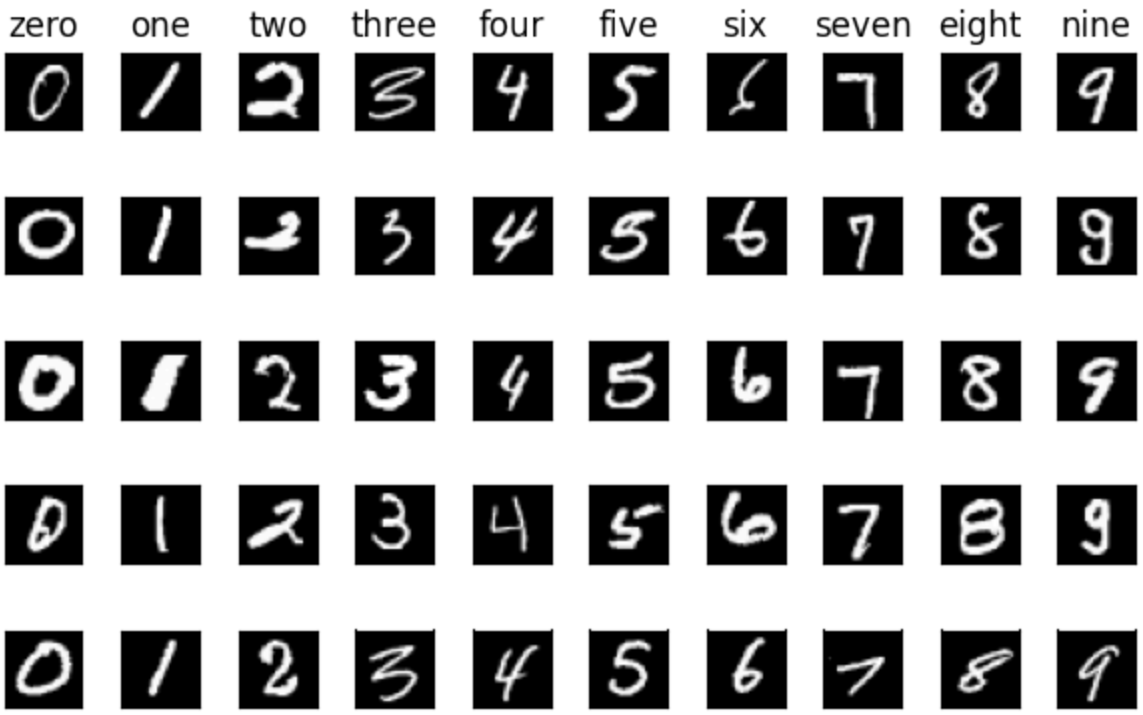
Before you start take a look at the concepts of mode collapse and GAN metrics.

Table 1: Generator and discriminator loss functions. The main difference whether the discriminator outputs a probability (MM GAN, NS GAN, DRAGAN) or its output is unbounded (WGAN, WGAN GP, LS GAN, BEGAN), whether the gradient penalty is present (WGAN GP, DRAGAN) and where is it evaluated.

GAN	Discriminator Loss	Generator Loss
MM GAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{GAN}} = -\mathbb{E}_{x\sim p_d}[\log(D(x))] - \mathbb{E}_{\hat{x}\sim p_g}[\log(1-D(\hat{x}))]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{GAN}} = \mathbb{E}_{\hat{x}\sim p_g}[\log(1-D(\hat{x}))]$
NS GAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{NSGAN}} = -\mathbb{E}_{x\sim p_d}[\log(D(x))] - \mathbb{E}_{\hat{x}\sim p_g}[\log(1-D(\hat{x}))]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{NSGAN}} = -\mathbb{E}_{\hat{x}\sim p_g}[\log(D(\hat{x}))]$
WGAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{WGAN}} = -\mathbb{E}_{x\sim p_d}[D(x)] + \mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{WGAN}} = -\mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})]$
WGAN GP	$\mathcal{L}_{\mathrm{D}}^{\mathrm{WGANGP}} = \mathcal{L}_{\mathrm{D}}^{\mathrm{WGAN}} + \lambda\mathbb{E}_{\hat{x}\sim p_g}[(\lvert\lvert\nabla D(\alpha x + (1-\alpha)\hat{x})\rvert\rvert_2 - 1)^2]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{WGANGP}} = -\mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})]$
LS GAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{LSGAN}} = -\mathbb{E}_{x\sim p_d}[(D(x)-1)^2] + \mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})^2]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{LSGAN}} = -\mathbb{E}_{\hat{x}\sim p_g}[(D(\hat{x}-1))^2]$
DRAGAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{DRAGAN}} = \mathcal{L}_{\mathrm{D}}^{\mathrm{GAN}} + \lambda\mathbb{E}_{\hat{x}\sim p_d+\mathcal{N}(0,c)}[(\lvert\lvert\nabla D(\hat{x})\rvert\rvert_2 - 1)^2]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{DRAGAN}} = \mathbb{E}_{\hat{x}\sim p_g}[\log(1-D(\hat{x}))]$
BEGAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{BEGAN}} = \mathbb{E}_{x\sim p_d}[\lvert\lvert x - \mathrm{AE}(x)\rvert\rvert_1] - k_t\mathbb{E}_{\hat{x}\sim p_g}[\lvert\lvert\hat{x} - \mathrm{AE}(\hat{x})\rvert\rvert_1]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{BEGAN}} = \mathbb{E}_{\hat{x}\sim p_g}[\lvert\lvert\hat{x} - \mathrm{AE}(\hat{x})\rvert\rvert_1]$

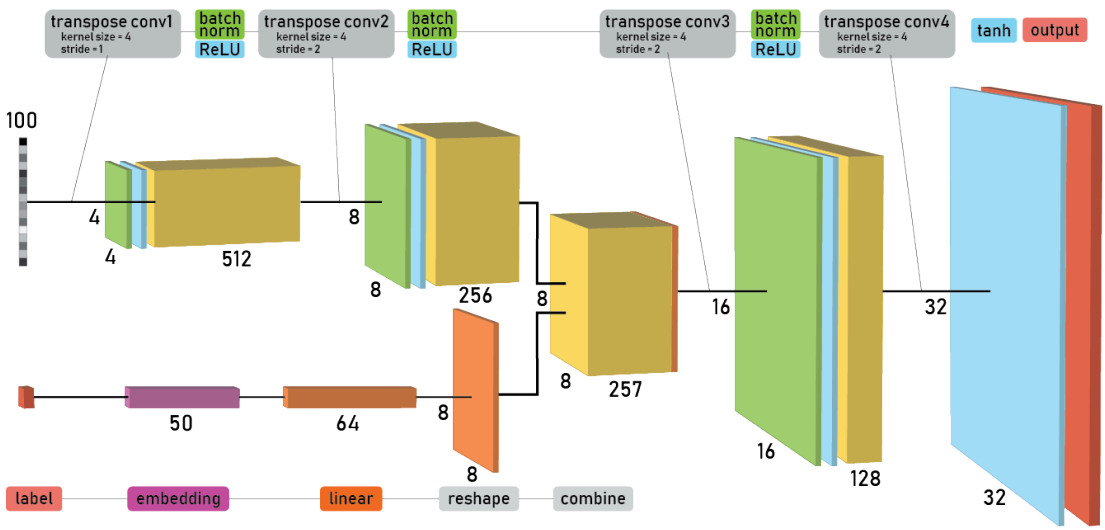
In this part, we are going to implement a conditional version of the Deep Convolutional Generative Adversarial Network (DCGAN) on MNIST since we have the labels. We expect the model to generate hand written digits given input conditions. The outputs of Conditional DCGAN with different conditions (from 0 to 9, in each coloum) should look like the following (each image is an example):

In [16]:
```
#@title  { run: "auto", vertical-output: true, display-mode: "both" }
def visualize_mnist(root_dir, num_examples=5):
    mnist = torchvision.datasets.MNIST(root_dir, download=True)
    labels = [label[4:] for label in mnist.classes]
    index = iter(torch.randperm(len(mnist)))

    rows, cols = num_examples, len(labels)
    fig, axes = plt.subplots(rows, cols, dpi=100)
    for col in range(cols):
        for row in range(rows):
            label = -1
            while label != col:
                # don't worry, we will never run out ;)
                image, label = mnist[next(index)]
            ax = axes[row][col]
            ax.imshow(image, cmap='gray')
            ax.xaxis.set_ticks([])
            ax.yaxis.set_ticks([])
            if row == 0:
                ax.set_title(labels[col])
            # if col == 0:
            #     ax.set_ylabel(f'{row + 1}')
    fig.tight_layout()
    plt.show(fig)


num_examples = 5 #@param {type:"slider", min:2, max:10, step:1}
mnist_root = Path(torch.hub.get_dir()) / 'datasets/MNIST'
visualize_mnist(mnist_root, num_examples)
```



Task 1: Model Construction

Implement the **Conditional DCGAN Generator** for images of size $1 \times 32 \times 32$ (we will upsample MNIST):



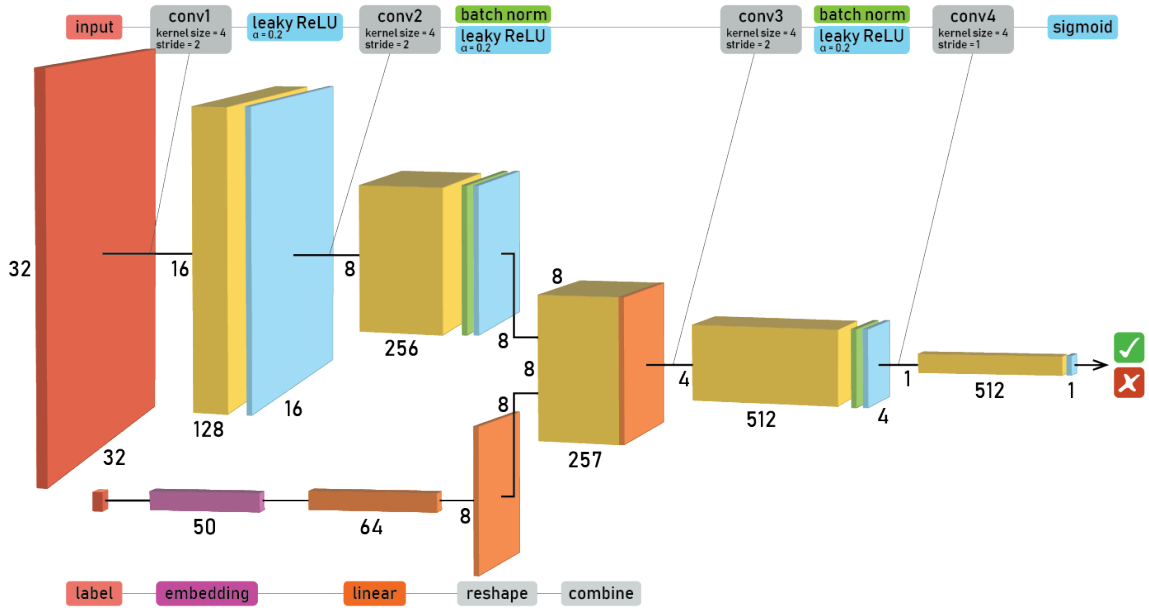
In [17]:
```
# TODO:  vvvvvvvvvvv (3 points)
# implement the generator network for C-DCGAN as in the figure
# generate a random batch to be passed to the generator
# answer the following from the paper and/or the figure:
# - what is the input in the top branch (the vector with size 100)?
# - what is input in the bottom branch?
# - what does the embedding layer do?
# - what does the transpose conv do?
class CondGenerator(nn.Module):
```

```
    def __init__(self, latent_dim, num_channels, num_classes):
        super().__init__()
        self.latent_dim = latent_dim
        self.num_classes = num_classes
        self.num_channels = num_channels

        # prior distribution branch to decode
        self.prior_dist_decoder = nn.Sequential(
            nn.ConvTranspose2d(latent_dim, 512, 4, stride=1, padding=0),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(512, 256, 4, stride=2, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
        )

        # conditional branch
        self.conditional_decoder = nn.Sequential(
            nn.Embedding(num_classes, 50),
            nn.Linear(50,64),
        )

        # main branch
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(257, 128, 4, stride=2,  padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(128, num_channels, 4, stride=2,  padding=1),
            nn.Tanh(),
        )

    def forward(self, latent_vector, label):
        # pass the latent vector to the prior distribution decoder
#         print(f'latent_vector: {latent_vector.shape}')
        latent_tensor = self.prior_dist_decoder(latent_vector.view(-1, self.
        # pass the label to the conditional decoder
        label_tensor = self.conditional_decoder(label)
        # combine to pass to the main decoder
        comb_tensor = torch.cat([latent_tensor, label_tensor.view(-1, 1, 8,
#         print(f'comb_tensor: {comb_tensor.shape}')
        # pass to main decoder
        generated_tensor = self.decoder(comb_tensor)
        return generated_tensor

    def random_batch(self, batch_size):
        device = next(generator.parameters()).device
        latent = torch.randn(batch_size, self.latent_dim, device=device)
        label = torch.randint(0, self.num_classes, (batch_size,), device=dev
        return latent, label
# ^^^^^^^^^^^^^^^^^


batch_size = 2
latent_dim = 100
num_channels = 1
num_classes = 10
```

```
device = torch.device('cuda:0')
generator = CondGenerator(latent_dim, num_channels, num_classes).to(device)

# test your implementation by doing a single forward pass using random input
latent_vector, labels = generator.random_batch(batch_size)
print(f'latent vector: {latent_vector.shape}, labels:{labels}')
print(generator(latent_vector, labels).shape)
```

```
latent vector: torch.Size([2, 100]), labels:tensor([8, 4], device='cuda:0')
torch.Size([2, 1, 32, 32])
```

Implement the **Conditional DCGAN Discriminator** as the following:



In [18]:

```
# TODO: vvvvvvvvvvv (2 points)
# implement the discriminator network for C-DCGAN as in the figure
# however, don't use sigmoid here (leave it to the training function)
# answer the following from the paper and/or the figure:
# 1. what is the usage of the discriminator?
# 2. what is the output of the discriminator?
class CondDiscriminator(nn.Module):
    def __init__(self, num_channels, num_classes):
        super().__init__()
        self.num_classes = num_classes
        self.num_channels = num_channels

        # image encoder
        self.image_encoder = nn.Sequential(
            nn.Conv2d(num_channels, 128, 4, stride=2, padding=1),
            nn.LeakyReLU(negative_slope=0.2, inplace=True),
            nn.Conv2d(128, 256, 4, stride=2, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(negative_slope=0.2, inplace=True),
        )

        # conditional branch
        self.conditional_decoder = nn.Sequential(
            nn.Embedding(num_classes, 50),
```

```
            nn.Linear(50,64),
        )

        # main branch
        self.encoder = nn.Sequential(
            nn.Conv2d(257, 512, 4, stride=2, padding=1),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(negative_slope=0.2, inplace=True),
            nn.Conv2d(512, 512, 4, stride=1, padding=0),
            nn.Flatten(),
            nn.Linear(512, 1)
        )


    def forward(self, image, label):
        # pass the image to the image encoder
        latent_tensor = self.image_encoder(image)
#         print(f'latent_tensor: {latent_tensor.shape}')
        # pass the label to the conditional decoder
        label_tensor = self.conditional_decoder(label)
        # combine to pass to the main encoder
        comb_tensor = torch.cat([latent_tensor, label_tensor.view(-1, 1, 8,
#         print(f'comb_tensor: {comb_tensor.shape}')
        # pass to the main encoder
        tensor = self.encoder(comb_tensor)#.sigmoid()
        return tensor

# ^^^^^^^^^^^^^^^^^


discriminator = CondDiscriminator(num_channels, num_classes).to(device)
# test your implementation by doing a single forward pass using random input
latent_vector, label = generator.random_batch(batch_size)
image = generator(latent_vector, labels)
print(f'image: {image.shape}')
print(discriminator(image, label).shape)
# ^^^^^^^^^^^^^^^^^
```

```
image: torch.Size([2, 1, 32, 32])
torch.Size([2, 1])
```

Once we have both the generator and the discriminator, let's initialize their weights as described in the paper (Section 4).

In [19]:
```
def weight_initialization(module):
    if isinstance(module, (nn.Conv2d, nn.ConvTranspose2d)):
        # TODO: vvvvvvvvvvv (0.5 points)
        # use the same initialization as mentioned in the paper
        nn.init.normal_(module.weight, 0.0, 0.02)
        nn.init.zeros_(module.bias)
        # ^^^^^^^^^^^^^^^^^
    elif isinstance(module, nn.BatchNorm2d):
        nn.init.normal_(module.weight, 1.0, 0.02)
        nn.init.zeros_(module.bias)
```

```
generator.apply(weight_initialization)
discriminator.apply(weight_initialization)
print('Done!')
```

```
Done!
```

Task 2: Training and Evaluation

In [20]:

```
def train(generator,
          discriminator,
          loader,
          generator_optimizer,
          discriminator_optimizer,
          device,
          epochs=5,
          show_every=50):
    generator.train(True)
    discriminator.train(True)

    criterion = F.binary_cross_entropy_with_logits
    as_real = lambda x: criterion(x, torch.ones_like(x))
    as_fake = lambda x: criterion(x, torch.zeros_like(x))

    num_classes = generator.num_classes
    num_batches = len(loader)
    writer = SummaryWriter(log_dir=None)  # use the default log_dir=./runs
    for epoch in range(epochs):
        loss_sum = count = 0
        for i, (real, real_labels) in enumerate(tqdm(loader), 1):
            real, real_labels = real.to(device), real_labels.to(device)

            # generate fake images
            latent, fake_labels = generator.random_batch(len(real))
            fake = generator(latent, fake_labels)

            # TODO: vvvvvvvvvvv (4 points)
            # Understand and solve the following
            # compute the loss and update the discriminator **only**
            # maximize log(D(x)) + log(1 - D(G(z)))
            # understand how well the discriminator should perform at the gl

            d_loss = 0.5*(as_real(discriminator(real, real_labels)) + as_fak
            discriminator_optimizer.zero_grad()
            d_loss.backward(retain_graph=True)
            discriminator_optimizer.step()

            # compute the loss and update the generator **only**
            # maximize log(D(G(z)))

            g_loss = as_real(discriminator(fake, fake_labels))
            generator.zero_grad()
            g_loss.backward(retain_graph=False)
            generator_optimizer.step()
            # ^^^^^^^^^^^^^^^^^
```

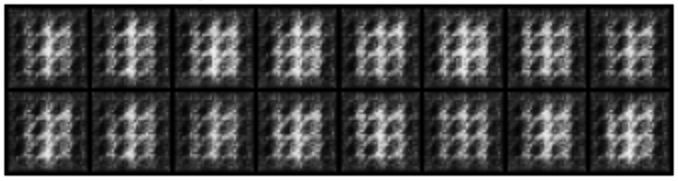
```
            count += len(fake)
            loss_sum += g_loss.item() * len(fake)

            step = epoch * num_batches + i - 1
            writer.add_scalar('D_loss/train', d_loss.item(), step)
            writer.add_scalar('G_loss/train', g_loss.item(), step)

            last_iteration = i == len(loader)
            if i % show_every == 0 or last_iteration:
                fig = plt.figure()
                ax = fig.add_subplot()
                grid = (fake.data[:16] + 1) * 0.5
                grid = torchvision.utils.make_grid(grid)
                writer.add_image('images', grid, step)
                ax.imshow(TF.to_pil_image(grid))
                ax.set_title(f'Epoch {epoch}: Loss = {loss_sum / count:.5f}'
                ax.axis('off')
                plt.show(fig)
        writer.close()
```

In [21]:

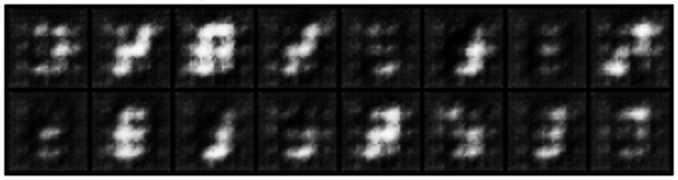
```
# TODO: vvvvvvvvvvv (0.5 points)
# load and prepare the MNIST dataset to be ready for training
# use the recommended setup as in Section 4 in the paper
def scale_image_to_tanh_range(image):
    return image * 2 - 1

img_size = 32
batch_size = 128
learning_rate = 0.0002
betas = (0.5, 0.999)
transform = T.Compose([T.Resize(img_size), T.ToTensor(), T.Normalize([0.5],
# transform = T.Compose([T.Resize(img_size), T.ToTensor(), T.Lambda(scale_im
dataset = torchvision.datasets.MNIST(root=mnist_root, train=True, download=T
loader = torch.utils.data.DataLoader(dataset, batch_size, shuffle=True)
generator_optimizer = torch.optim.Adam(generator.parameters(), lr=learning_r
discriminator_optimizer = torch.optim.Adam(discriminator.parameters(), lr=le
train(generator, discriminator, loader, generator_optimizer,
      discriminator_optimizer, device, epochs=20, show_every=50)
# ^^^^^^^^^^^^^^^^^
```

```
  0%|          | 0/469 [00:00<?, ?it/s]
```

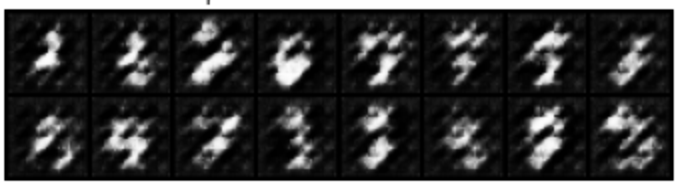
Epoch 0: Loss = 12.25798



Epoch 0: Loss = 11.35719



Epoch 0: Loss = 9.28249



Epoch 0: Loss = 8.27057

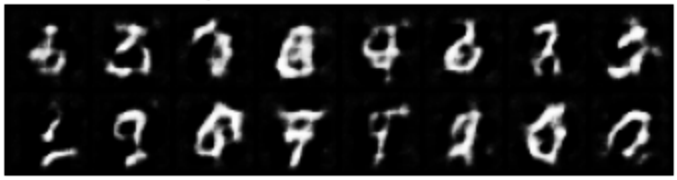
Epoch 0: Loss = 7.45154

Epoch 0: Loss = 6.86577

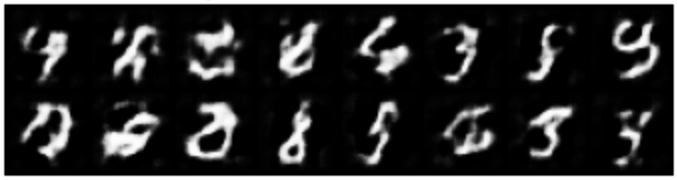
Epoch 0: Loss = 6.41891

Epoch 0: Loss = 6.06063

Epoch 0: Loss = 5.78402



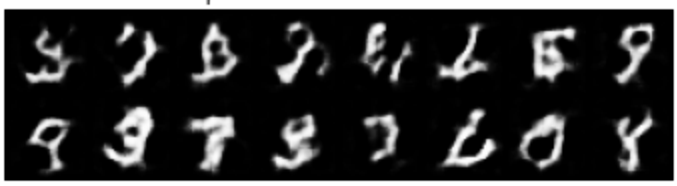
Epoch 0: Loss = 5.67263



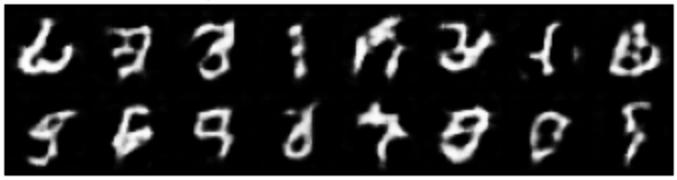
0%| | 0/469 [00:00<?, ?it/s]

Epoch 1: Loss = 3.30022

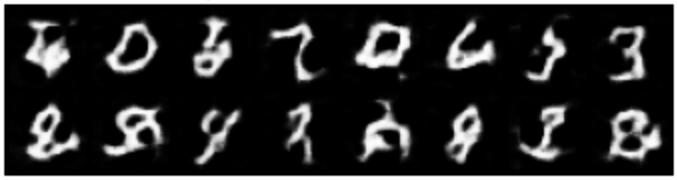
Epoch 1: Loss = 3.15808



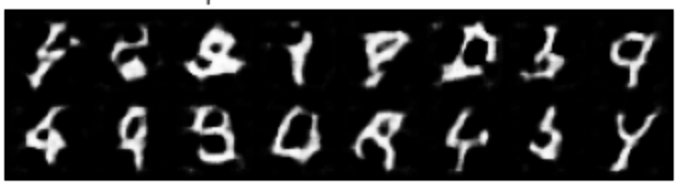
Epoch 1: Loss = 3.10739



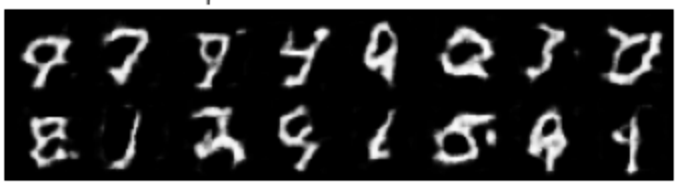
Epoch 1: Loss = 3.18857



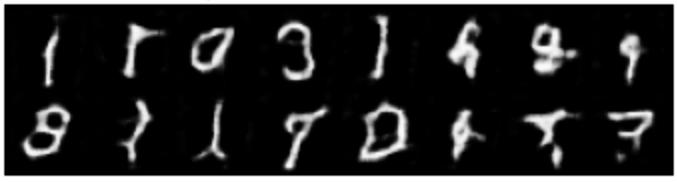
Epoch 1: Loss = 3.19734



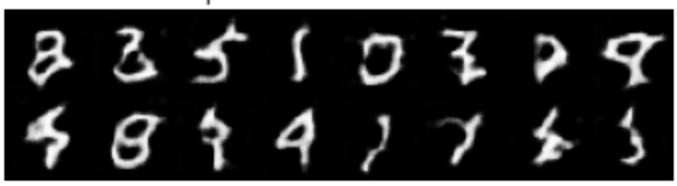
Epoch 1: Loss = 3.22679



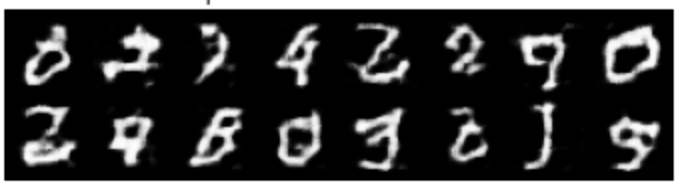
Epoch 1: Loss = 3.25195



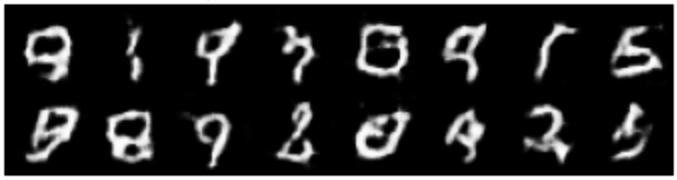
Epoch 1: Loss = 3.33312



Epoch 1: Loss = 3.41129

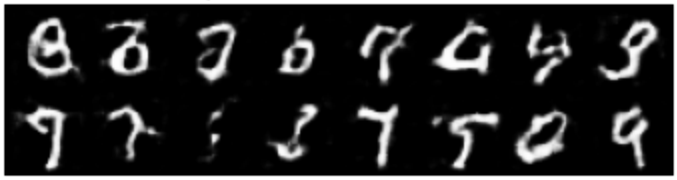


Epoch 1: Loss = 3.36905

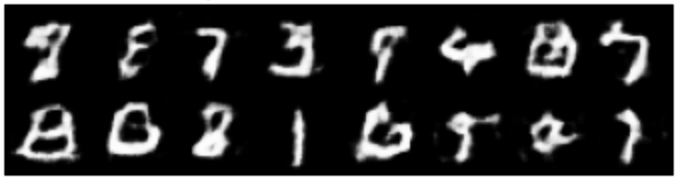


0%| | 0/469 [00:00<?, ?it/s]

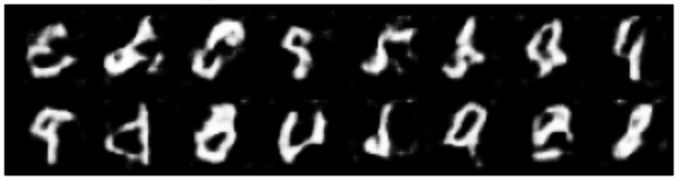
Epoch 2: Loss = 2.79872



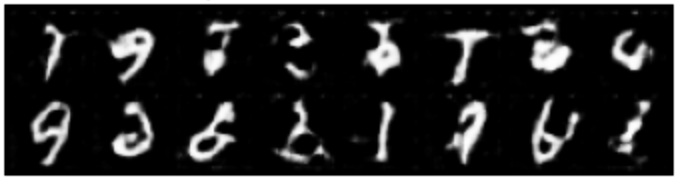
Epoch 2: Loss = 3.11700



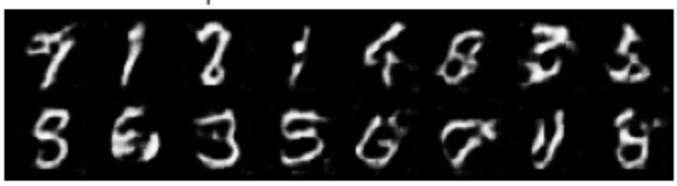
Epoch 2: Loss = 3.35902



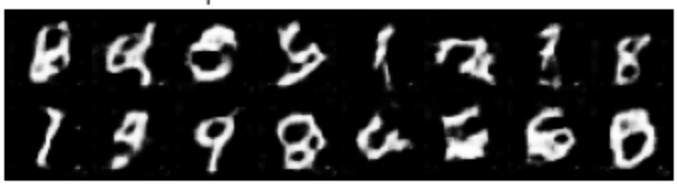
Epoch 2: Loss = 3.61054



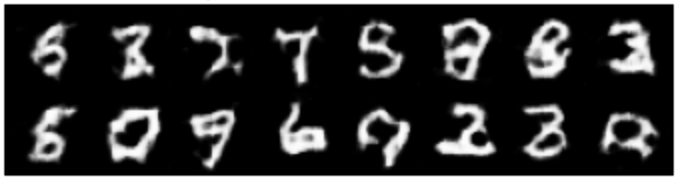
Epoch 2: Loss = 3.77634



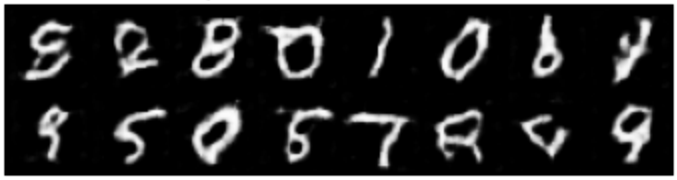
Epoch 2: Loss = 3.71099



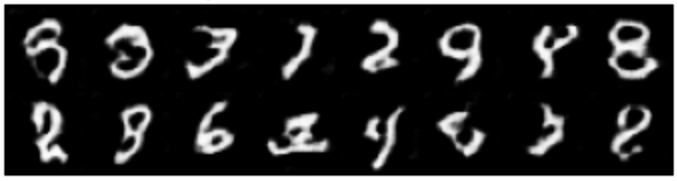
Epoch 2: Loss = 3.79233



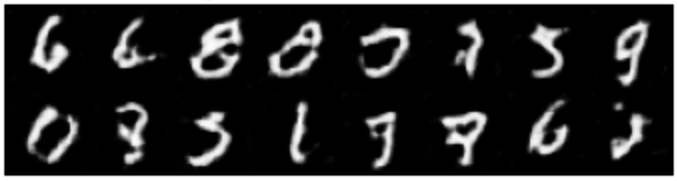
Epoch 2: Loss = 3.78472



Epoch 2: Loss = 3.85542

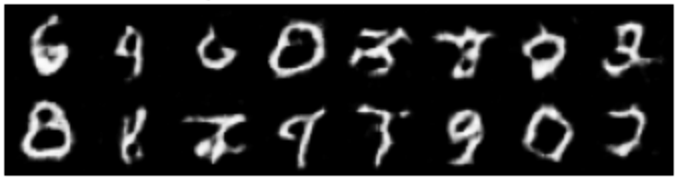


Epoch 2: Loss = 3.88304

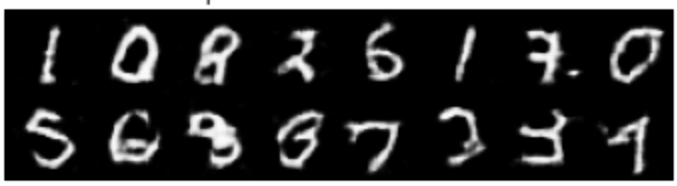


0%| | 0/469 [00:00<?, ?it/s]

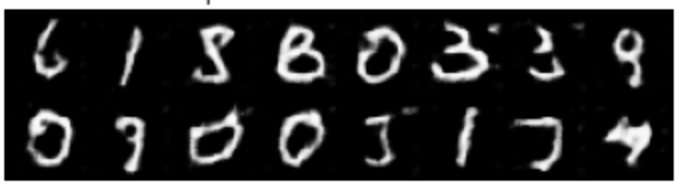
Epoch 3: Loss = 3.57193



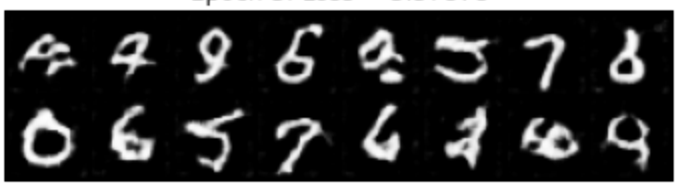
Epoch 3: Loss = 3.29220



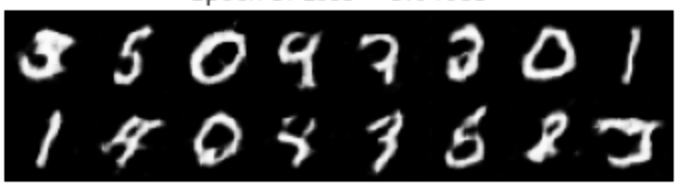
Epoch 3: Loss = 3.35379



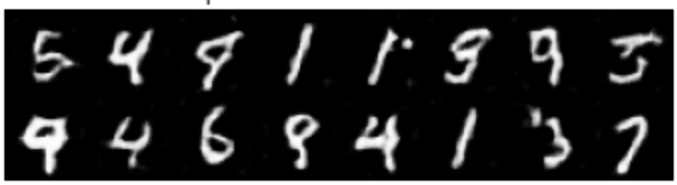
Epoch 3: Loss = 3.57878



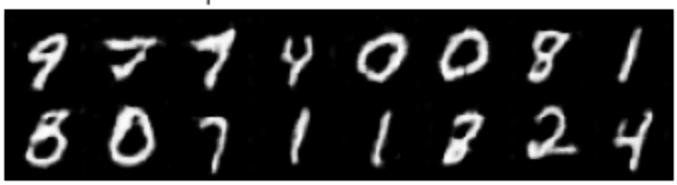
Epoch 3: Loss = 3.64685



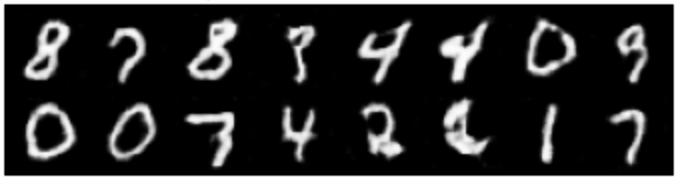
Epoch 3: Loss = 3.69874



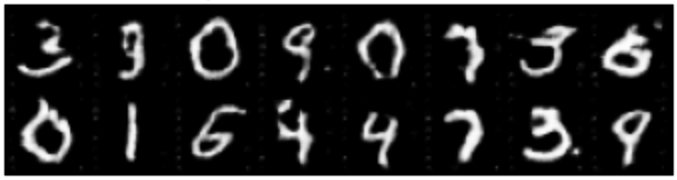
Epoch 3: Loss = 3.70861



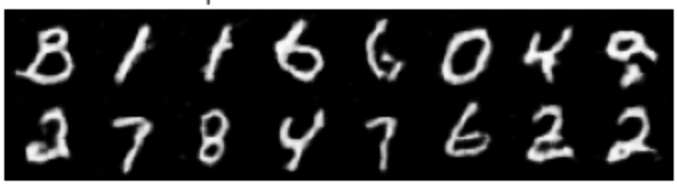
Epoch 3: Loss = 3.74211



Epoch 3: Loss = 3.70820

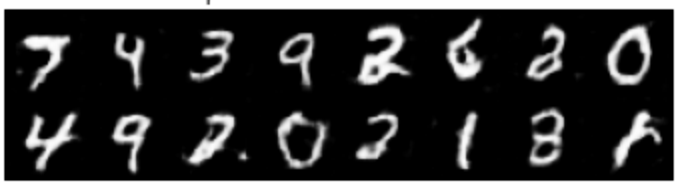


Epoch 3: Loss = 3.71836

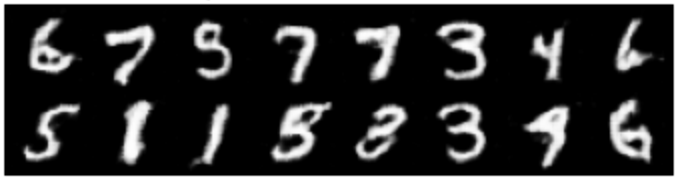


0%| | 0/469 [00:00<?, ?it/s]

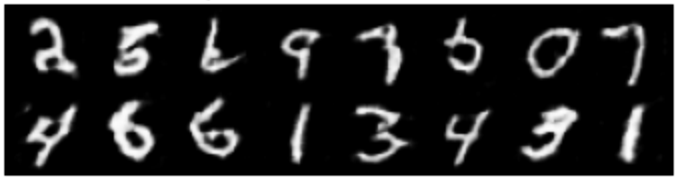
Epoch 4: Loss = 3.91816



Epoch 4: Loss = 4.02564

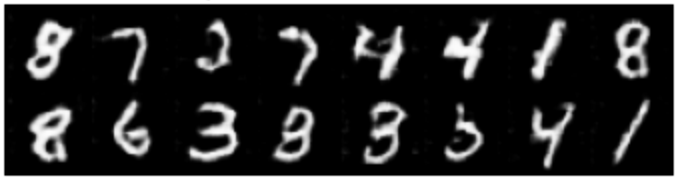


Epoch 4: Loss = 3.81663

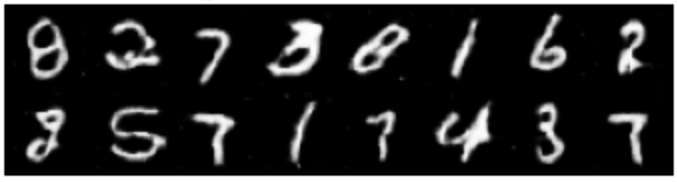


Epoch 4: Loss = 3.80756

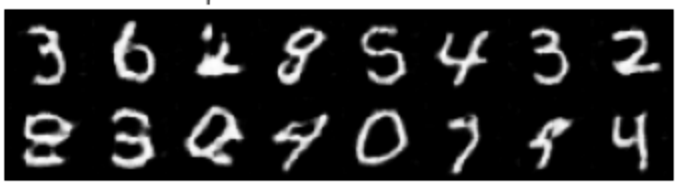
Epoch 4: Loss = 3.70492



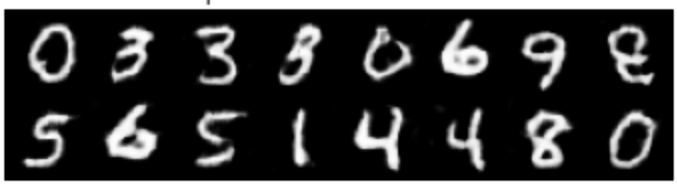
Epoch 4: Loss = 3.80125



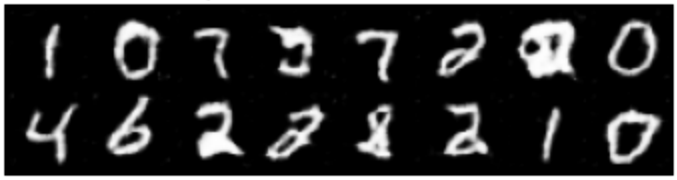
Epoch 4: Loss = 3.79909



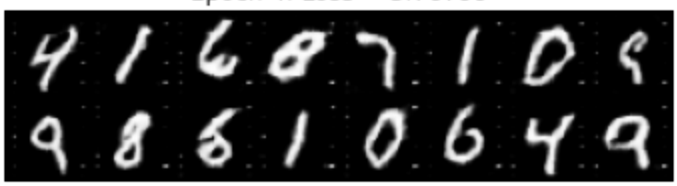
Epoch 4: Loss = 3.73584



Epoch 4: Loss = 3.74218

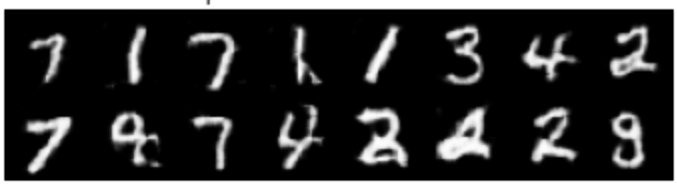


Epoch 4: Loss = 3.76799

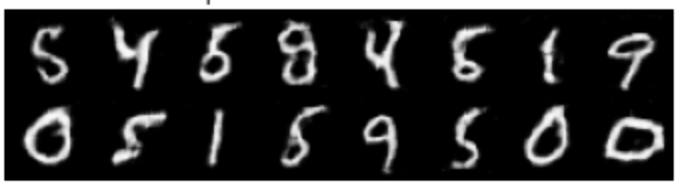


0%| | 0/469 [00:00<?, ?it/s]

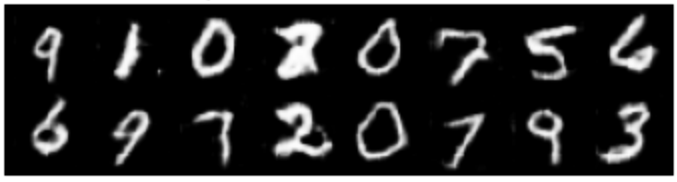
Epoch 5: Loss = 4.71058



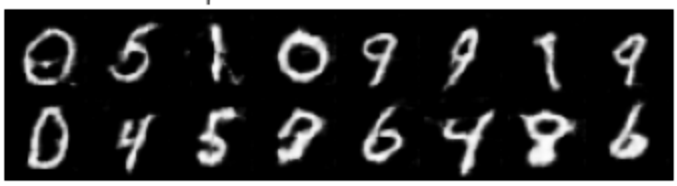
Epoch 5: Loss = 3.60029



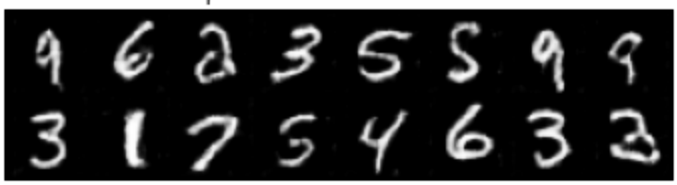
Epoch 5: Loss = 3.44909



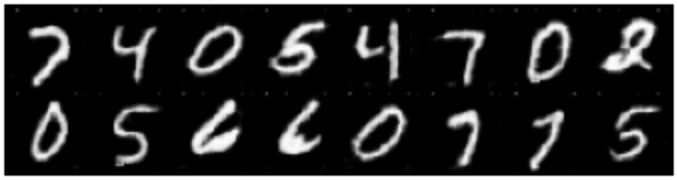
Epoch 5: Loss = 3.49106



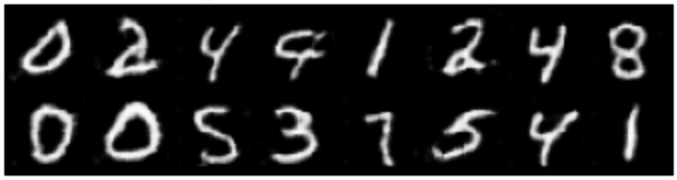
Epoch 5: Loss = 3.53366



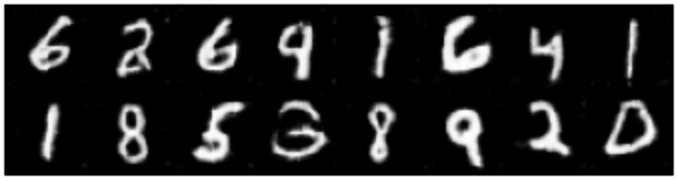
Epoch 5: Loss = 3.70289



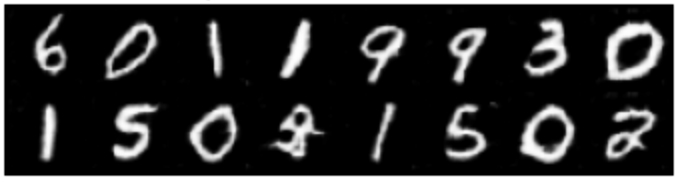
Epoch 5: Loss = 3.82135



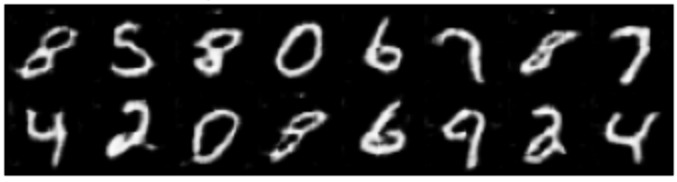
Epoch 5: Loss = 3.81428



Epoch 5: Loss = 3.89927

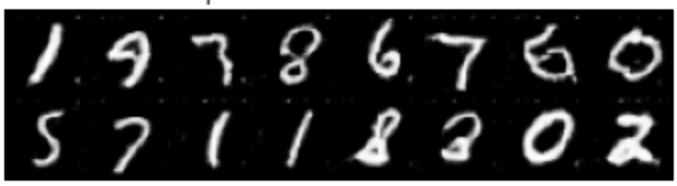


Epoch 5: Loss = 3.91274

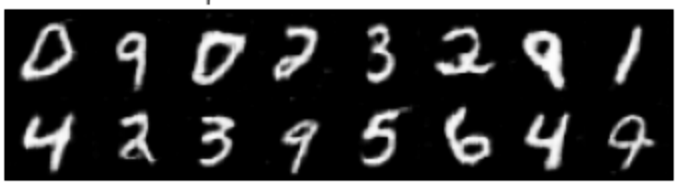


0%| | 0/469 [00:00<?, ?it/s]

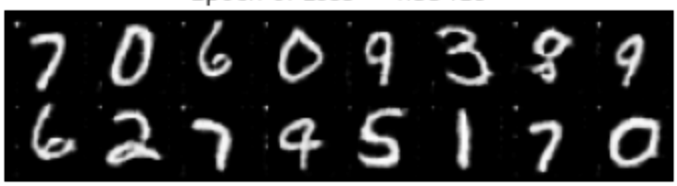
Epoch 6: Loss = 3.52965



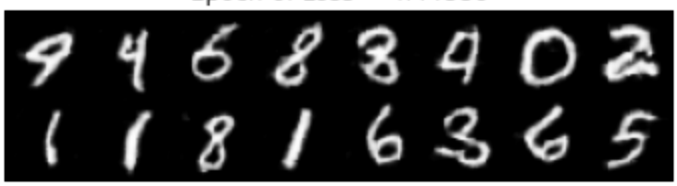
Epoch 6: Loss = 3.94361



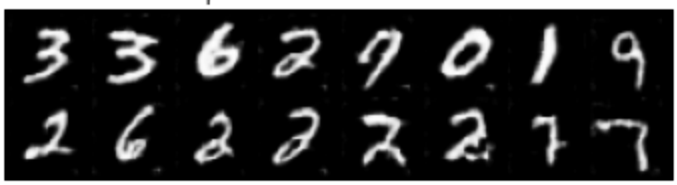
Epoch 6: Loss = 4.33419



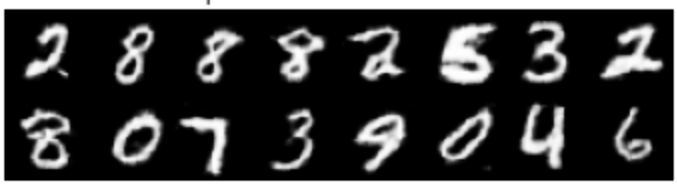
Epoch 6: Loss = 4.44886



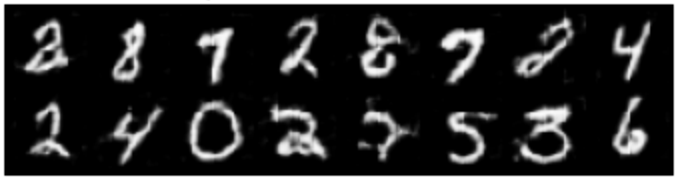
Epoch 6: Loss = 4.52075



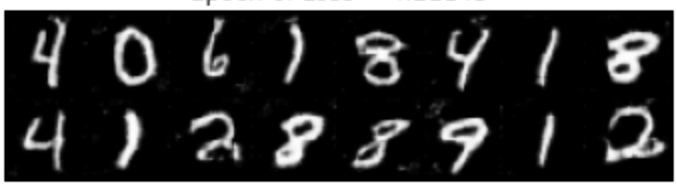
Epoch 6: Loss = 4.50498



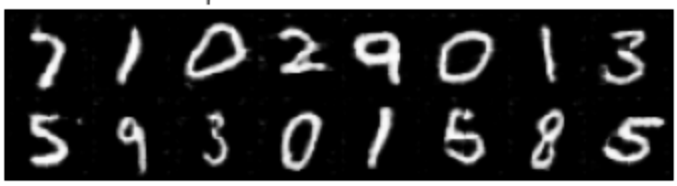
Epoch 6: Loss = 4.28169



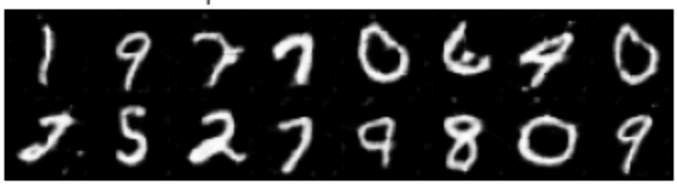
Epoch 6: Loss = 4.21143



Epoch 6: Loss = 4.19181

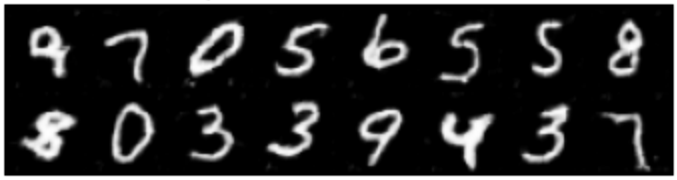


Epoch 6: Loss = 4.22064

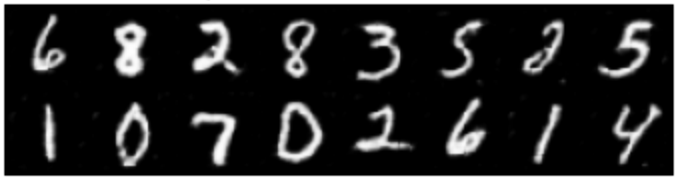


0%| | 0/469 [00:00<?, ?it/s]

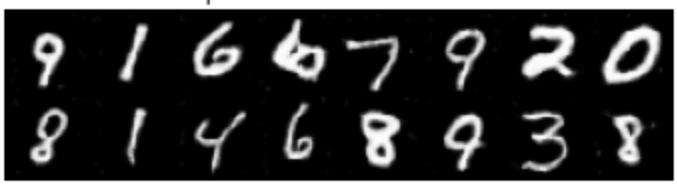
Epoch 7: Loss = 5.13500



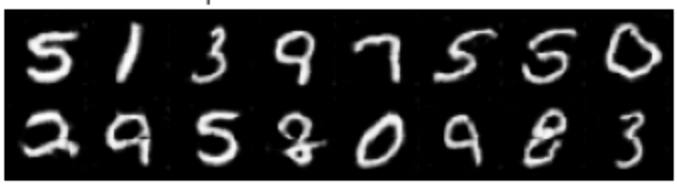
Epoch 7: Loss = 5.35157



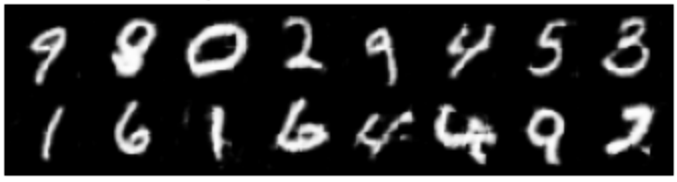
Epoch 7: Loss = 5.58259



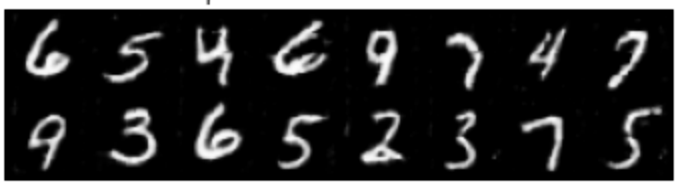
Epoch 7: Loss = 5.03069



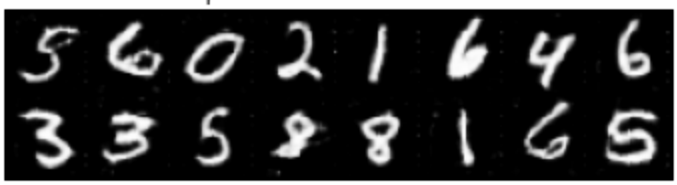
Epoch 7: Loss = 4.84636



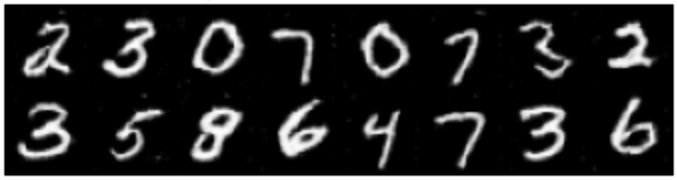
Epoch 7: Loss = 4.76922



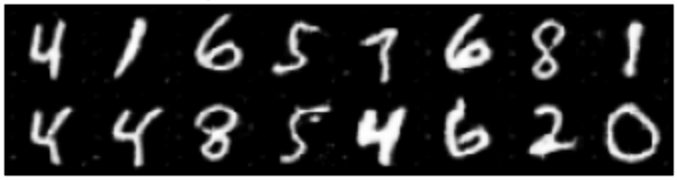
Epoch 7: Loss = 4.82758



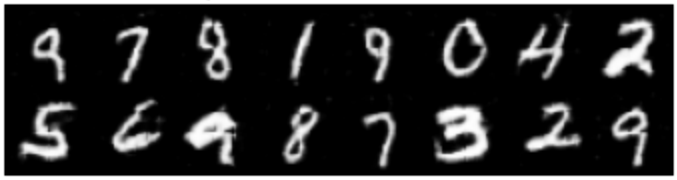
Epoch 7: Loss = 4.91206



Epoch 7: Loss = 5.03765

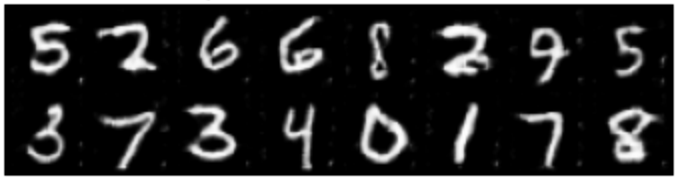


Epoch 7: Loss = 5.02088

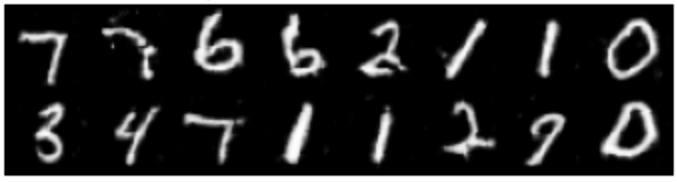


0%| | 0/469 [00:00<?, ?it/s]

Epoch 8: Loss = 3.44333

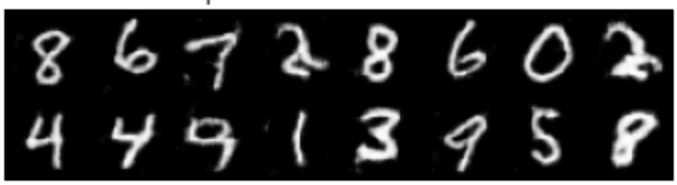


Epoch 8: Loss = 3.92865

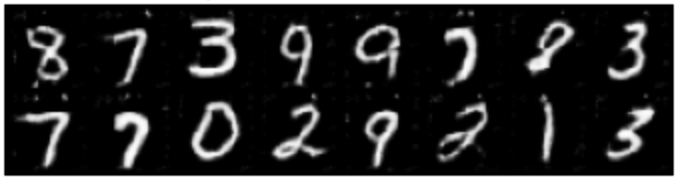


Epoch 8: Loss = 4.21525

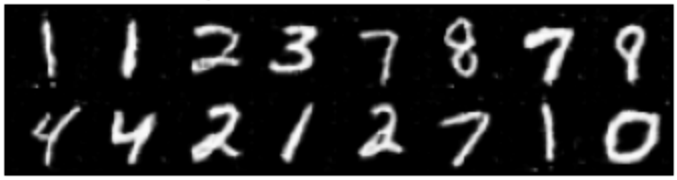
Epoch 8: Loss = 4.27611



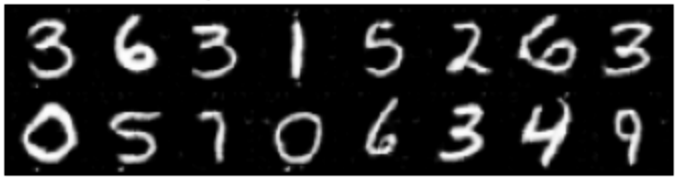
Epoch 8: Loss = 4.32052



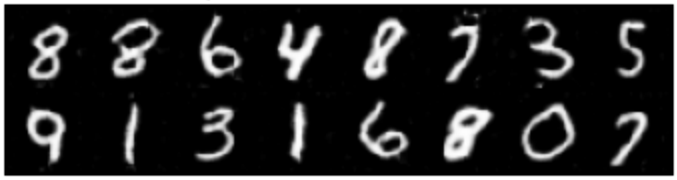
Epoch 8: Loss = 4.51921



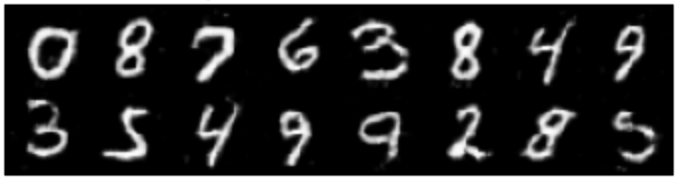
Epoch 8: Loss = 4.70760



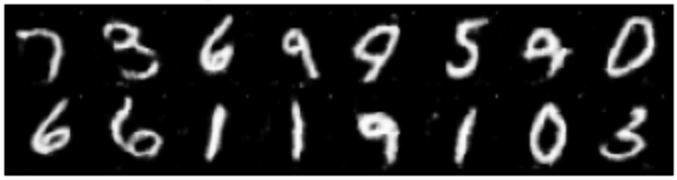
Epoch 8: Loss = 4.85230



Epoch 8: Loss = 4.94320

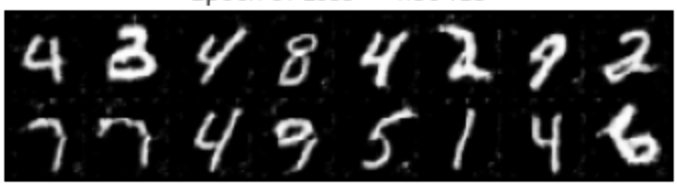


Epoch 8: Loss = 4.88958

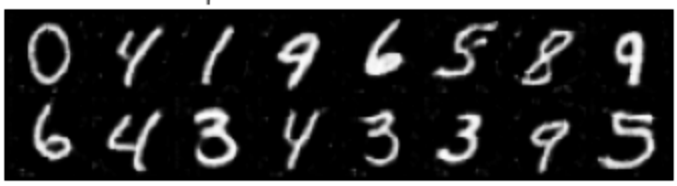


0%| | 0/469 [00:00<?, ?it/s]

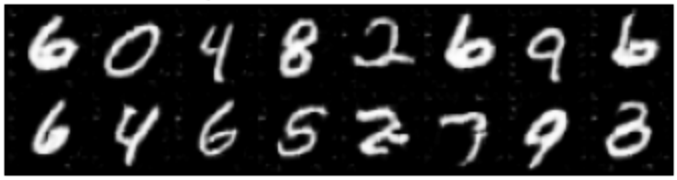
Epoch 9: Loss = 4.59415



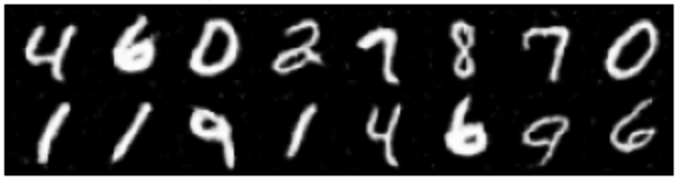
Epoch 9: Loss = 4.95272



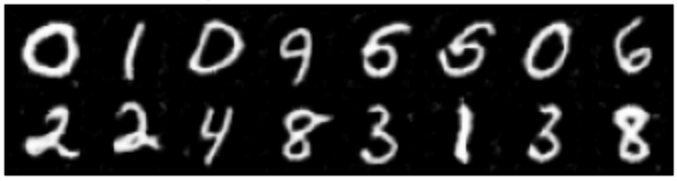
Epoch 9: Loss = 5.26003



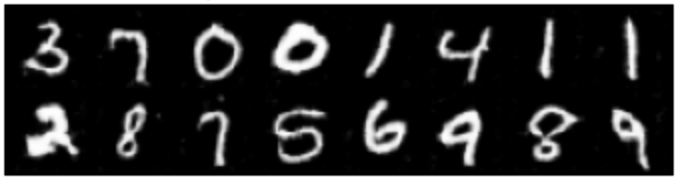
Epoch 9: Loss = 5.50086



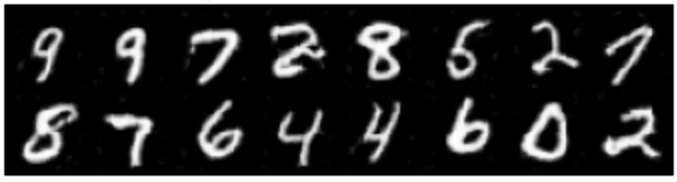
Epoch 9: Loss = 5.68408



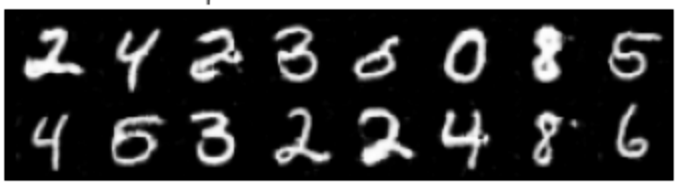
Epoch 9: Loss = 5.84657



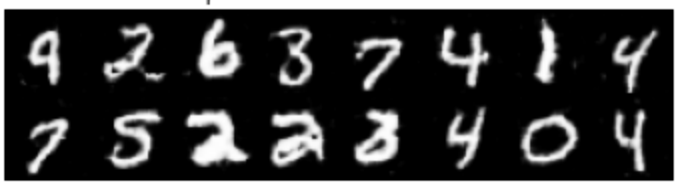
Epoch 9: Loss = 5.97287



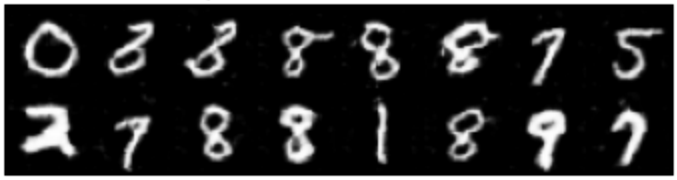
Epoch 9: Loss = 6.06606



Epoch 9: Loss = 6.06426

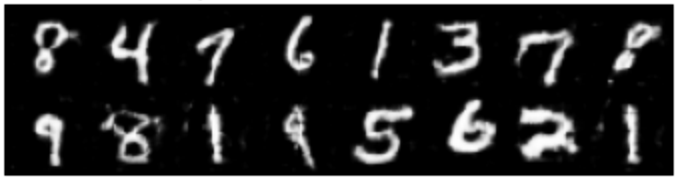


Epoch 9: Loss = 5.91306

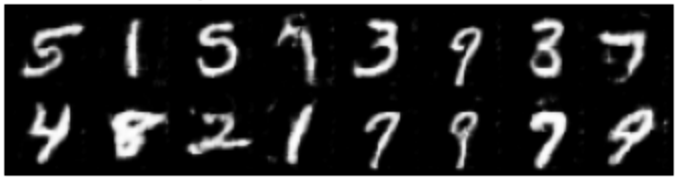


0%| | 0/469 [00:00<?, ?it/s]

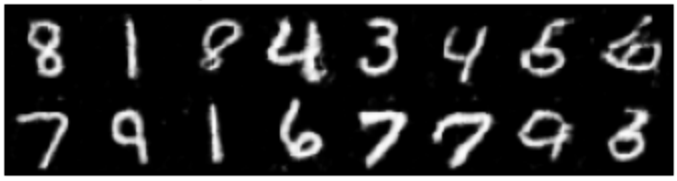
Epoch 10: Loss = 2.72728



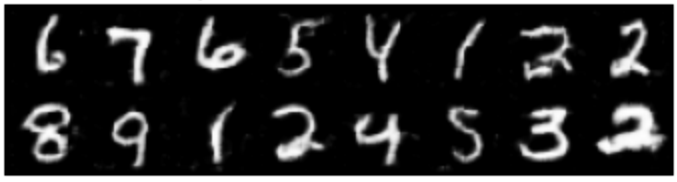
Epoch 10: Loss = 2.95963



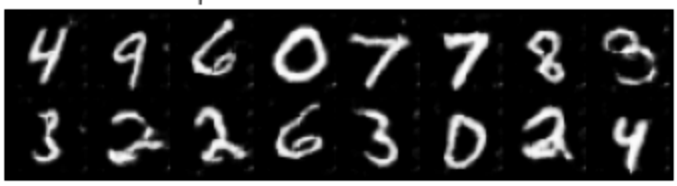
Epoch 10: Loss = 3.13843



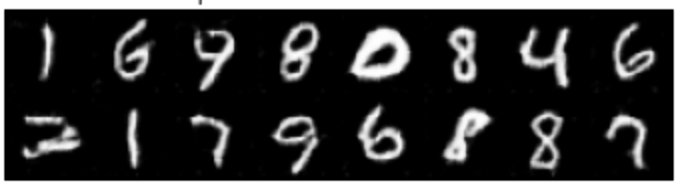
Epoch 10: Loss = 3.29107



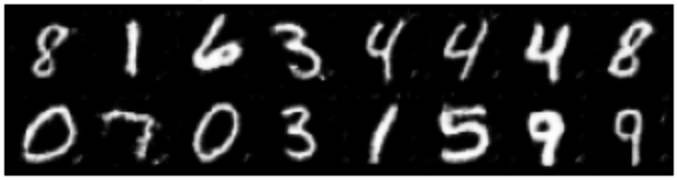
Epoch 10: Loss = 3.48332



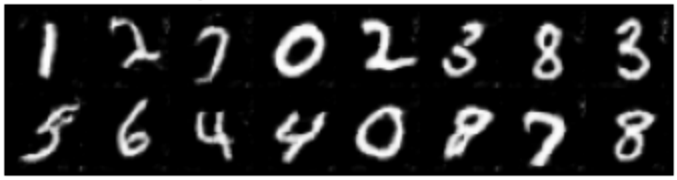
Epoch 10: Loss = 3.54670



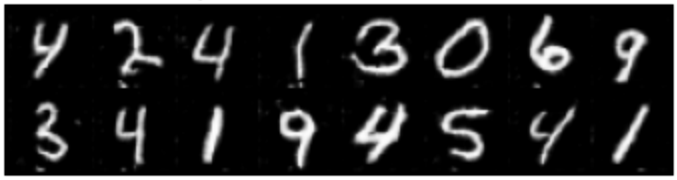
Epoch 10: Loss = 3.63793



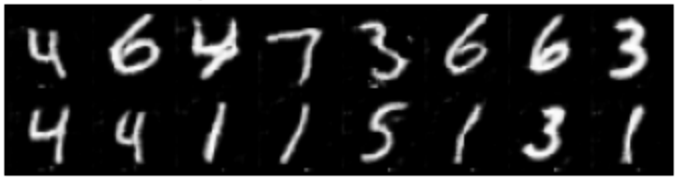
Epoch 10: Loss = 3.78963



Epoch 10: Loss = 3.95760

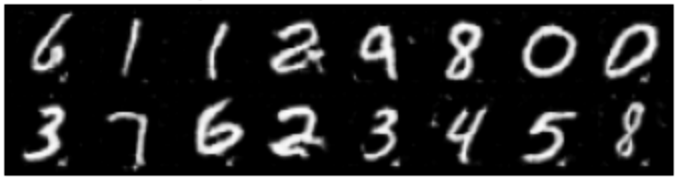


Epoch 10: Loss = 4.02545

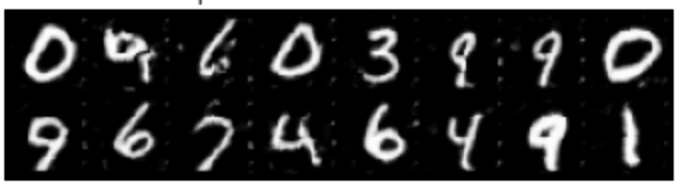


0%| | 0/469 [00:00<?, ?it/s]

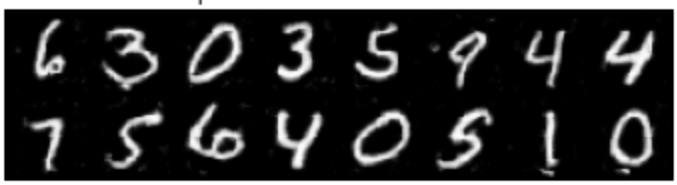
Epoch 11: Loss = 5.78171



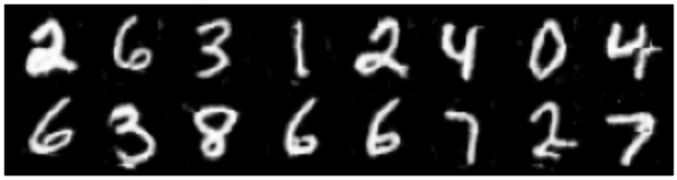
Epoch 11: Loss = 5.98729



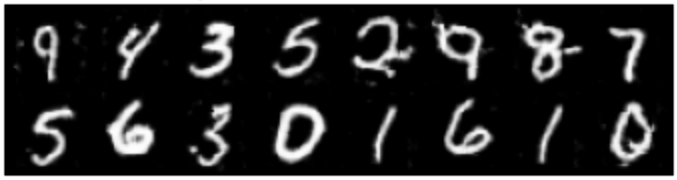
Epoch 11: Loss = 6.11750



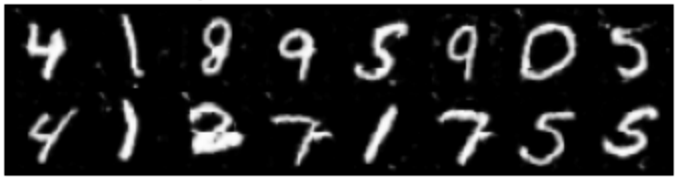
Epoch 11: Loss = 6.24981



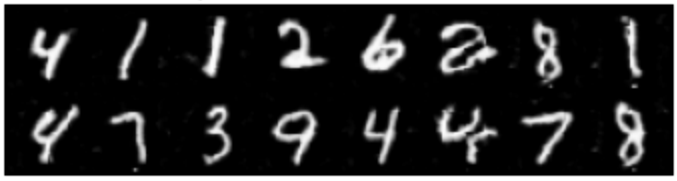
Epoch 11: Loss = 6.34998



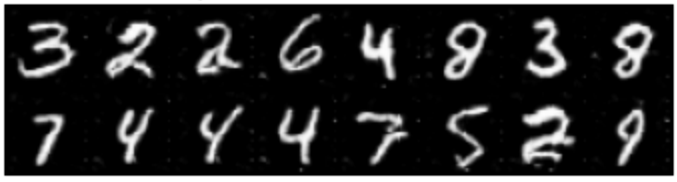
Epoch 11: Loss = 6.46799



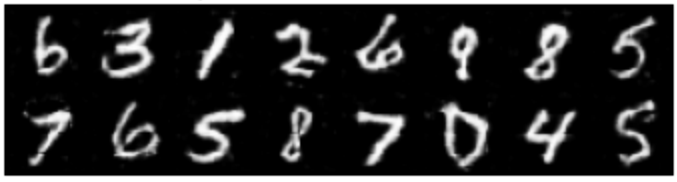
Epoch 11: Loss = 6.56095



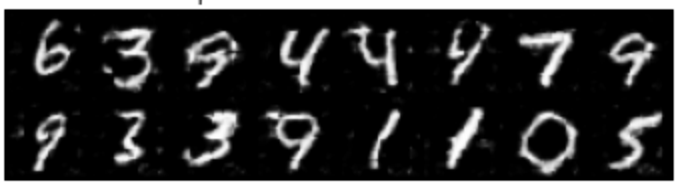
Epoch 11: Loss = 6.64429



Epoch 11: Loss = 6.58391

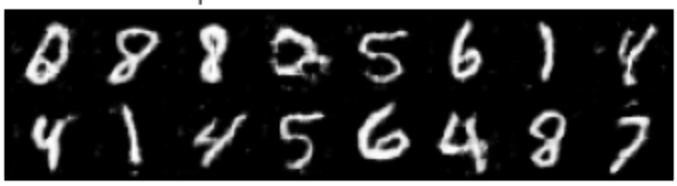


Epoch 11: Loss = 6.42817



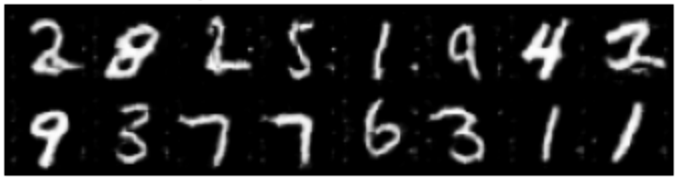
0%| | 0/469 [00:00<?, ?it/s]

Epoch 12: Loss = 3.33261

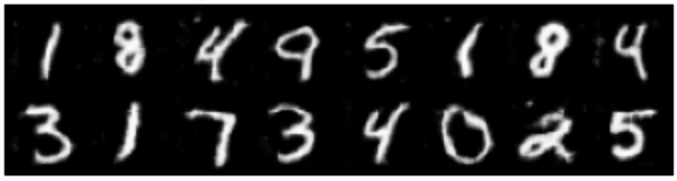


Epoch 12: Loss = 3.58011

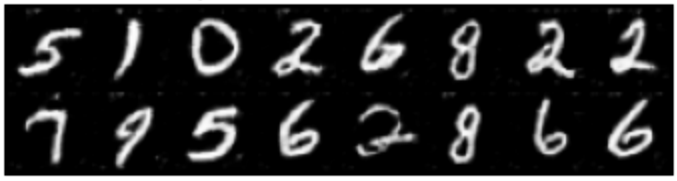
Epoch 12: Loss = 3.69803



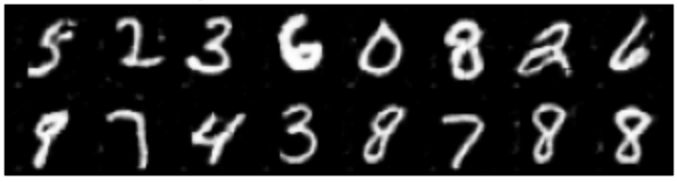
Epoch 12: Loss = 3.94378



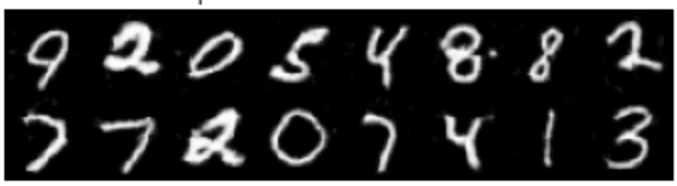
Epoch 12: Loss = 4.07749



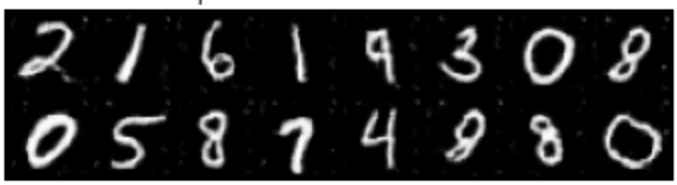
Epoch 12: Loss = 4.25112



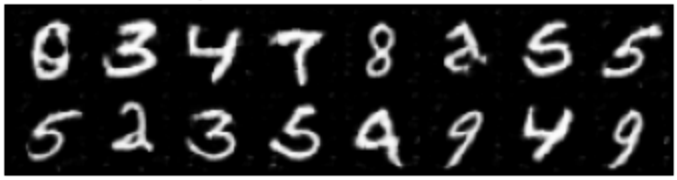
Epoch 12: Loss = 4.36909



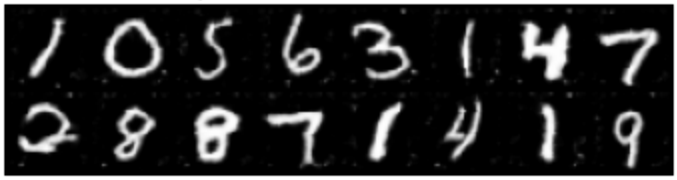
Epoch 12: Loss = 4.41489



Epoch 12: Loss = 4.52099

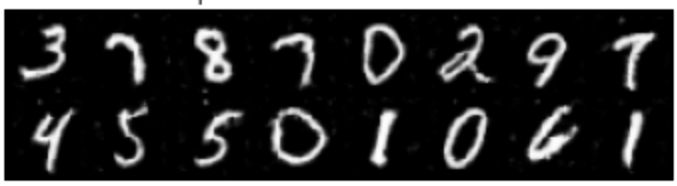


Epoch 12: Loss = 4.57055

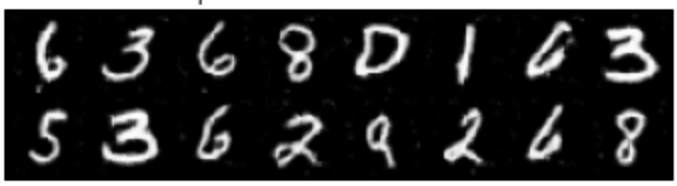


0%| | 0/469 [00:00<?, ?it/s]

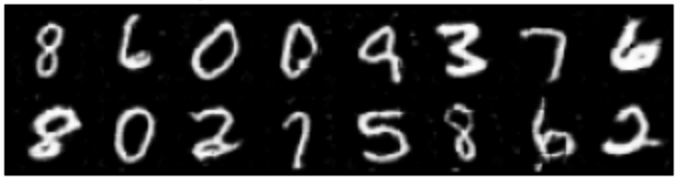
Epoch 13: Loss = 5.98980



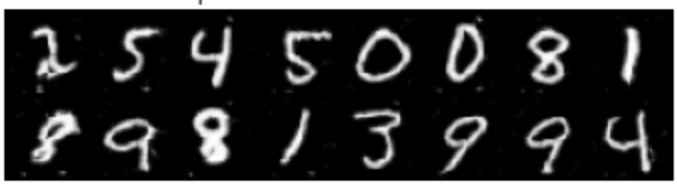
Epoch 13: Loss = 6.18786



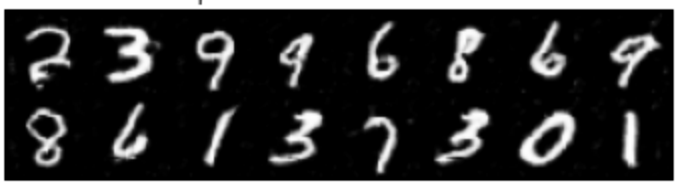
Epoch 13: Loss = 6.32871



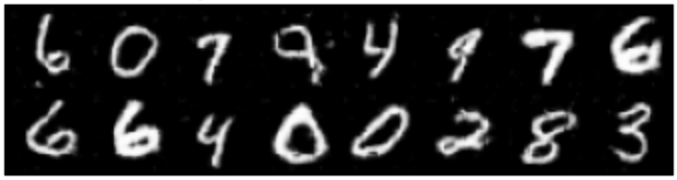
Epoch 13: Loss = 6.41793



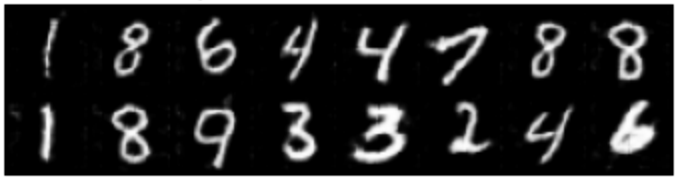
Epoch 13: Loss = 6.49501



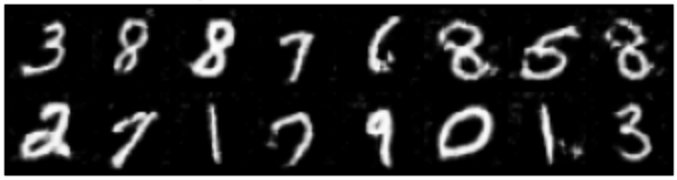
Epoch 13: Loss = 6.25947



Epoch 13: Loss = 5.84442

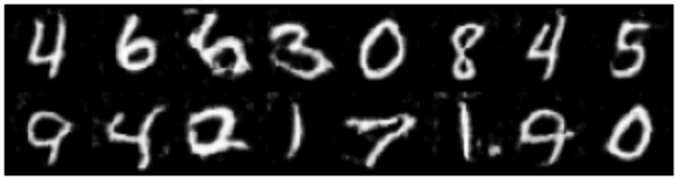


Epoch 13: Loss = 5.65816



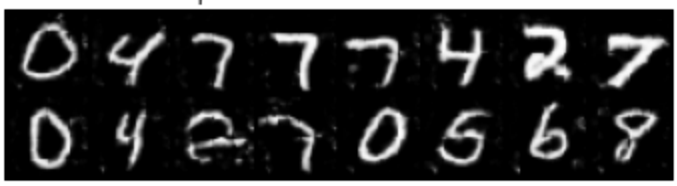
Epoch 13: Loss = 5.59449

Epoch 13: Loss = 5.59589

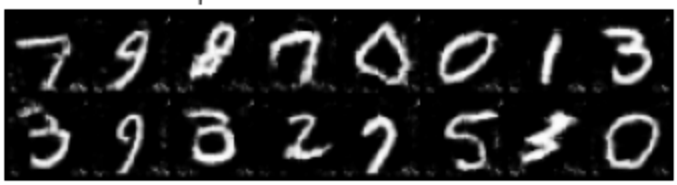


0%| | 0/469 [00:00<?, ?it/s]

Epoch 14: Loss = 5.91879

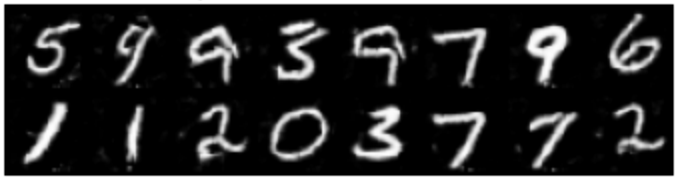


Epoch 14: Loss = 6.15691

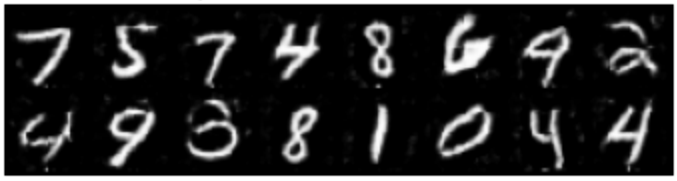


Epoch 14: Loss = 6.31298

Epoch 14: Loss = 6.46536

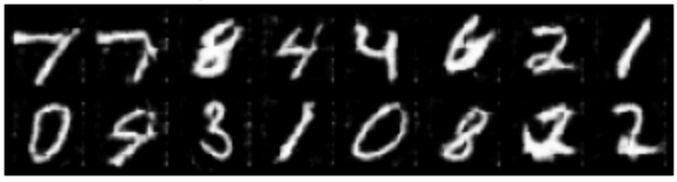


Epoch 14: Loss = 6.56448

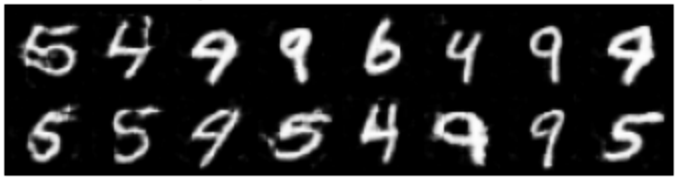


Epoch 14: Loss = 6.67118

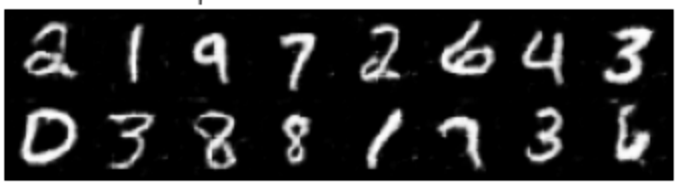
Epoch 14: Loss = 6.30544



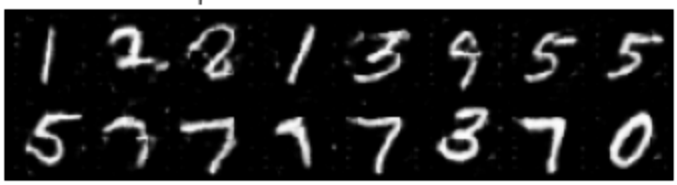
Epoch 14: Loss = 5.92462



Epoch 14: Loss = 5.68248

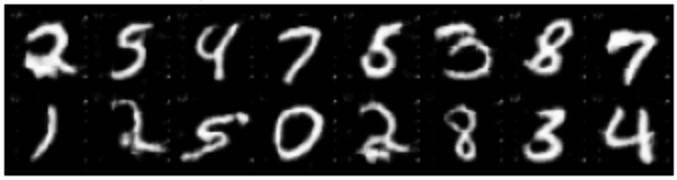


Epoch 14: Loss = 5.61575



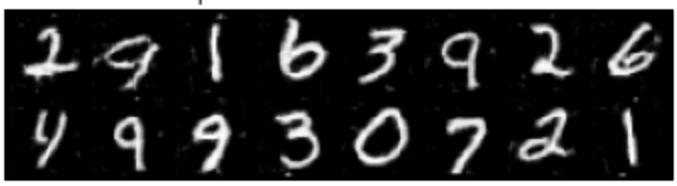
0%| | 0/469 [00:00<?, ?it/s]

Epoch 15: Loss = 4.36331

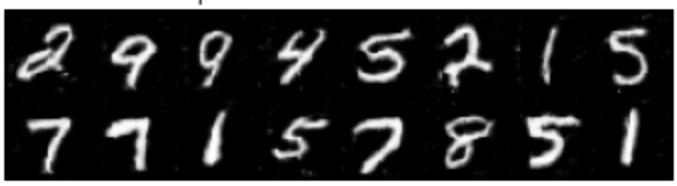


Epoch 15: Loss = 4.58231

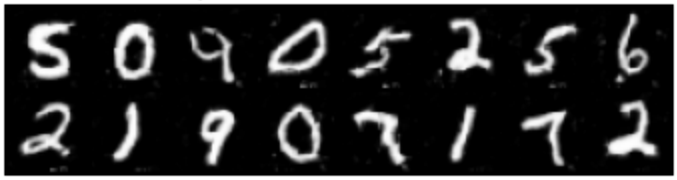
Epoch 15: Loss = 4.84482



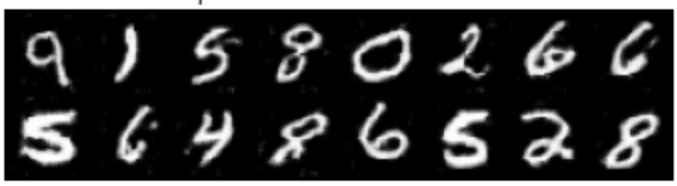
Epoch 15: Loss = 5.06786



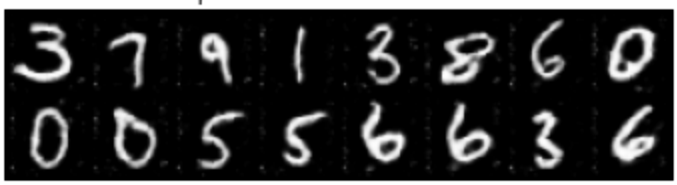
Epoch 15: Loss = 5.26227



Epoch 15: Loss = 5.32963

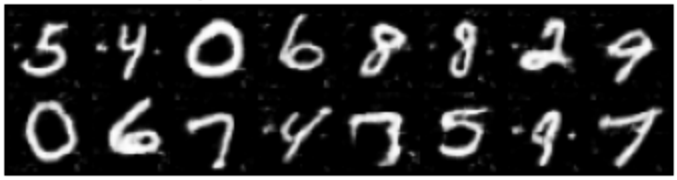


Epoch 15: Loss = 5.12476

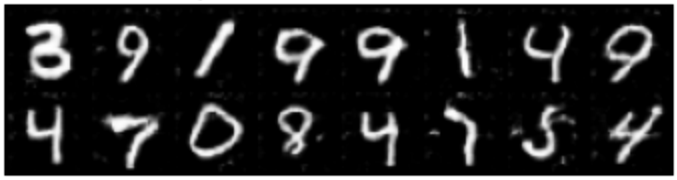


Epoch 15: Loss = 5.08663

Epoch 15: Loss = 5.13184



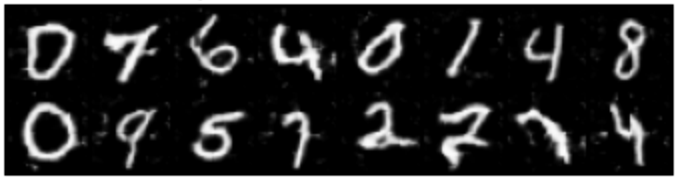
Epoch 15: Loss = 5.16069



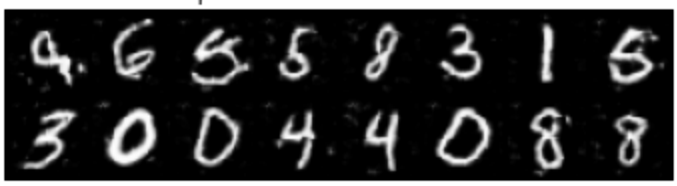
0%| | 0/469 [00:00<?, ?it/s]

Epoch 16: Loss = 6.01639

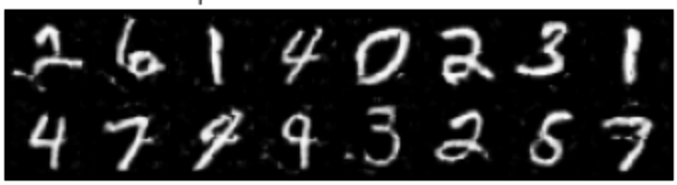
Epoch 16: Loss = 6.20361



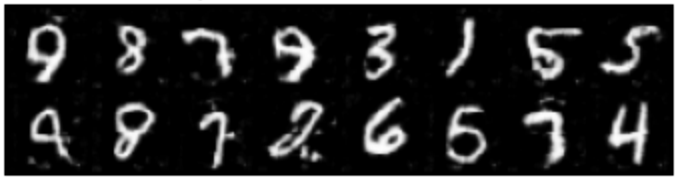
Epoch 16: Loss = 6.32838



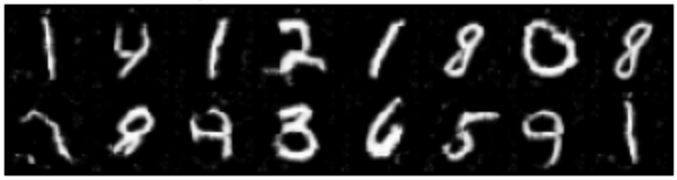
Epoch 16: Loss = 6.47006



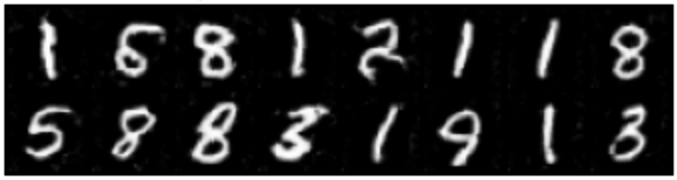
Epoch 16: Loss = 6.61945



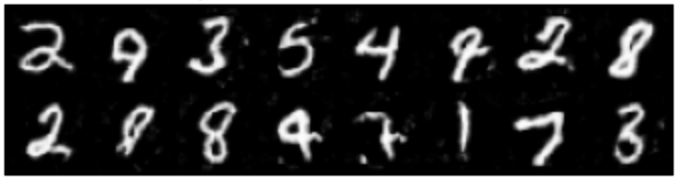
Epoch 16: Loss = 6.70373



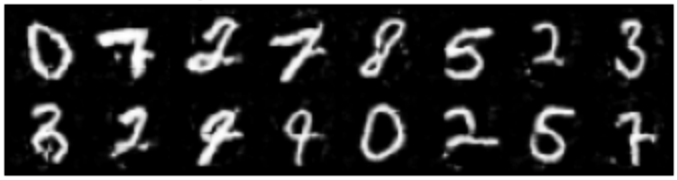
Epoch 16: Loss = 6.77560



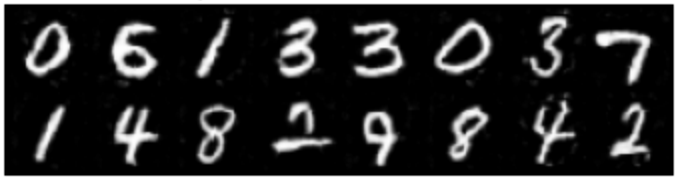
Epoch 16: Loss = 6.84193



Epoch 16: Loss = 6.89401

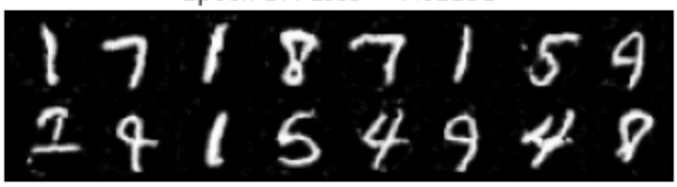


Epoch 16: Loss = 6.92119

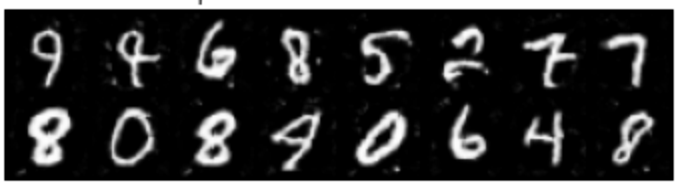


0%| | 0/469 [00:00<?, ?it/s]

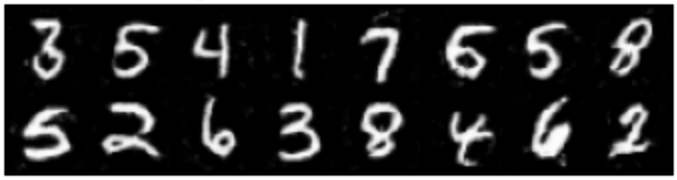
Epoch 17: Loss = 7.62231



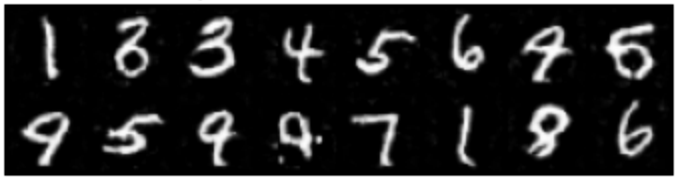
Epoch 17: Loss = 7.59800



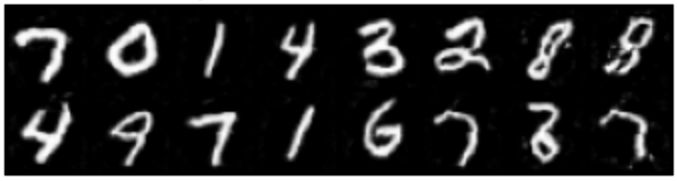
Epoch 17: Loss = 7.58846



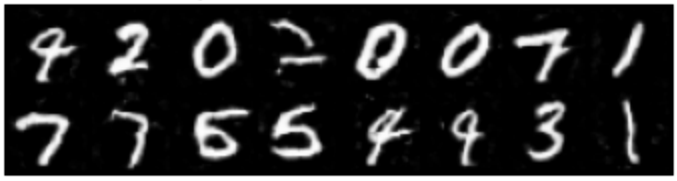
Epoch 17: Loss = 7.59889



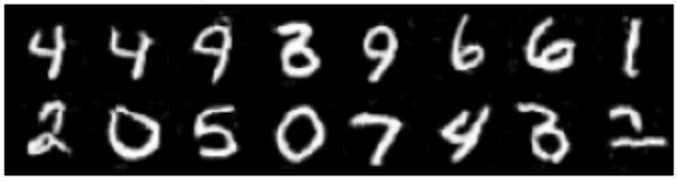
Epoch 17: Loss = 7.69169



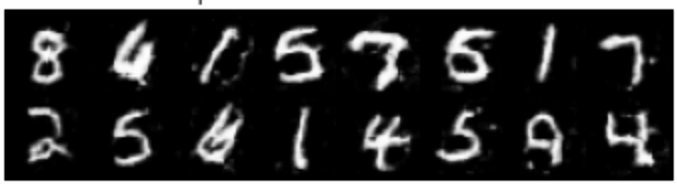
Epoch 17: Loss = 7.71218



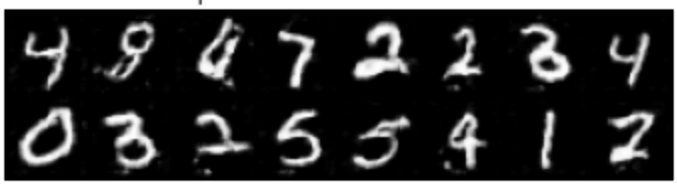
Epoch 17: Loss = 7.79390



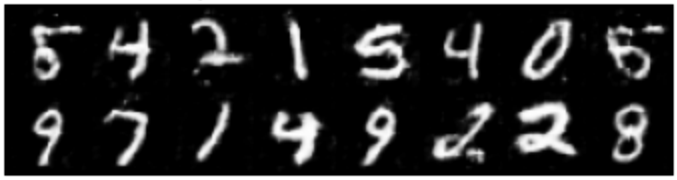
Epoch 17: Loss = 7.08218



Epoch 17: Loss = 6.66052

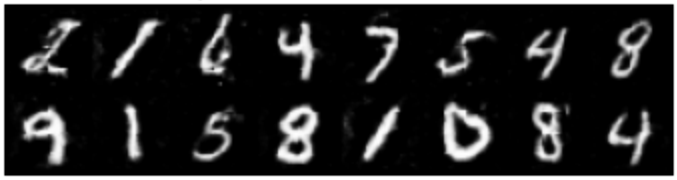


Epoch 17: Loss = 6.54043

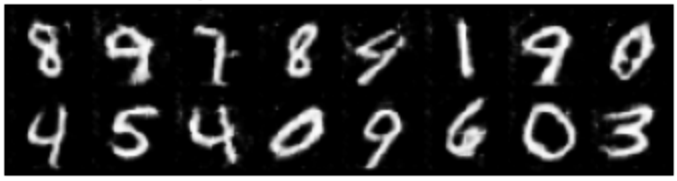


0%| | 0/469 [00:00<?, ?it/s]

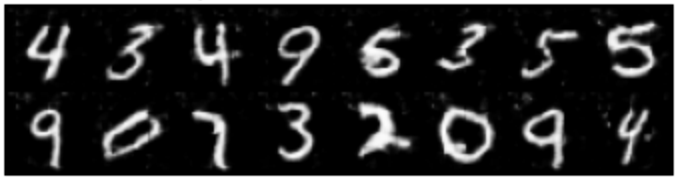
Epoch 18: Loss = 3.94918



Epoch 18: Loss = 4.40318



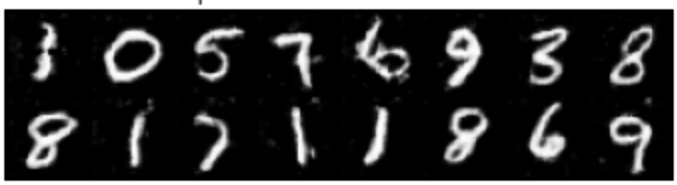
Epoch 18: Loss = 4.70653



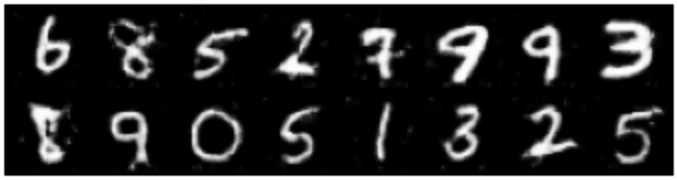
Epoch 18: Loss = 4.86408

Epoch 18: Loss = 4.50674

Epoch 18: Loss = 4.45914

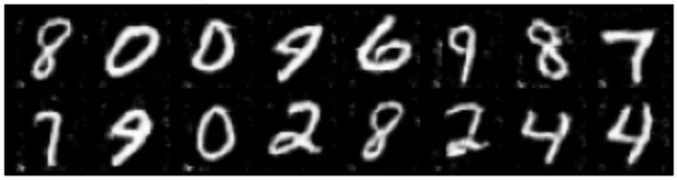


Epoch 18: Loss = 4.45806



Epoch 18: Loss = 4.52035

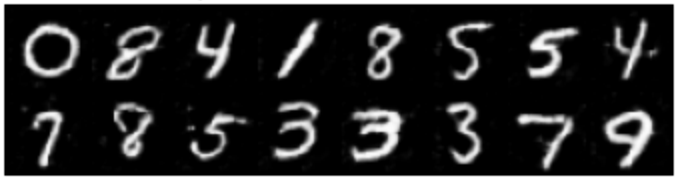
Epoch 18: Loss = 4.62317



Epoch 18: Loss = 4.67258

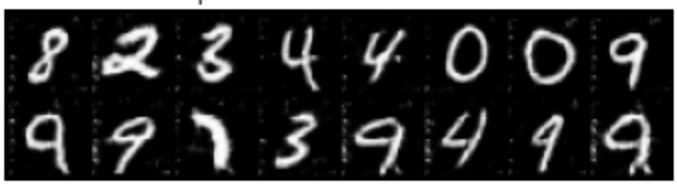
0%| | 0/469 [00:00<?, ?it/s]

Epoch 19: Loss = 3.88058

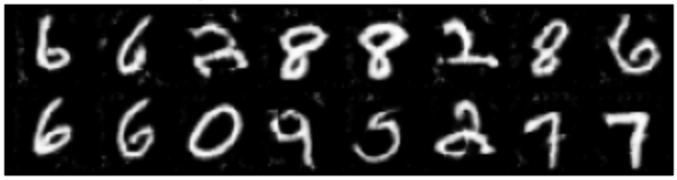


Epoch 19: Loss = 4.12246

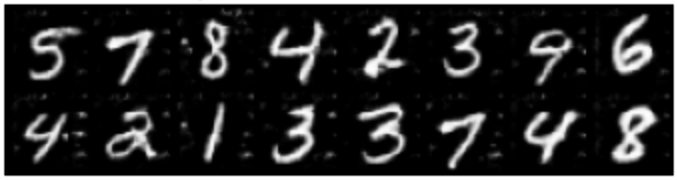
Epoch 19: Loss = 4.53609



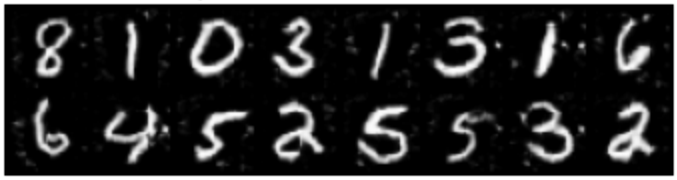
Epoch 19: Loss = 4.89567



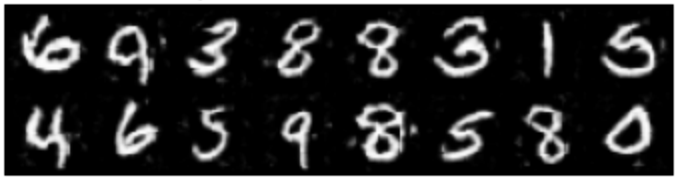
Epoch 19: Loss = 5.18552



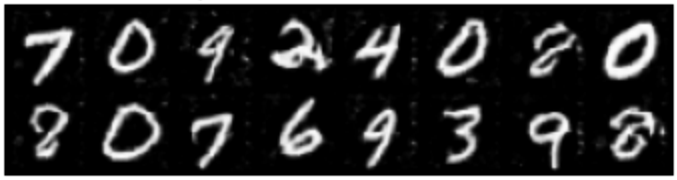
Epoch 19: Loss = 5.41350



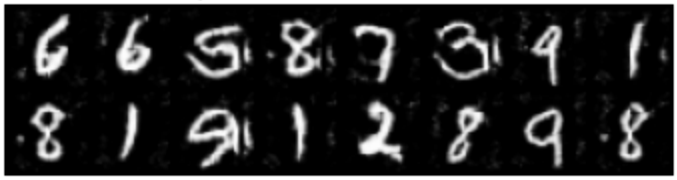
Epoch 19: Loss = 5.62481

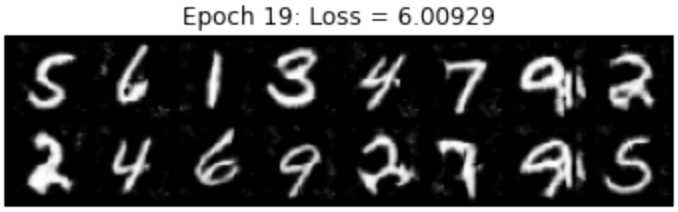


Epoch 19: Loss = 5.79830



Epoch 19: Loss = 5.94754





Let's visualize an example for each digit (conditionally).

In [22]:
```
generator.train(False)
with torch.no_grad():
    # TODO: vvvvvvvvvvv (1 points)
    # create a label for each digit with an accompanying latent vector
    latent = torch.randn(10, generator.latent_dim, device=device)
    label = torch.tensor(range(10), device=device, dtype=torch.long)
    # ^^^^^^^^^^^^^^^^^
    image = torchvision.utils.make_grid(generator(latent, label), 10)
TF.to_pil_image(TF.resize((image + 1) * 0.5, 64))
```

Out[22]:

Task 3: Network Inversion

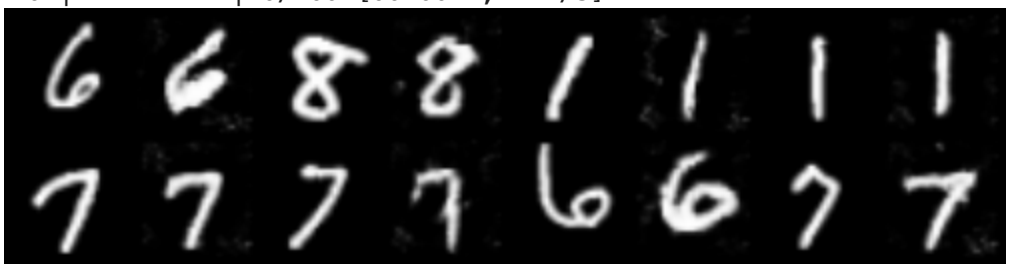
Since GANs don't have an encoder, let's invert the generator (i.e., find the latent vector that generate the closet output to a given image). The idea is very simple, do gradient descent on the latent vector. Start with a randomly generated vector $\mathbf{z}$. Then, compute a reconstruction loss (e.g., MSE) to the given image. Compute the gradient of the loss with respect to $\mathbf{z}$ using backpropagation (`torch.autograd.backward()` or `torch.autograd.grad()`). The only issue with this, is that you have to provide the label as well because it is not differentiable. However, if you insist, we can use the embedding instead then choose the label with the closest embedding to the final solution.

In [23]:
```
def invert(model, images, labels, lr, steps):
    model.train(False)
    latent_dim = model.latent_dim
    latent = torch.randn(len(images), latent_dim, device=images.device)
    latent.requires_grad_(True)
    for _ in tqdm(range(steps)):
        outputs = model(latent, labels)
        # TODO: vvvvvvvvvvv (1 points)
        # compute the needed gradient
        MSELoss = nn.MSELoss()
        loss = MSELoss(outputs, images)
        loss.backward()
        grad = latent.grad
        # ^^^^^^^^^^^^^^^^^
```

```
        latent.data -= lr * grad
    latent.requires_grad_(False)
    grid = torch.stack([images, outputs.data])
    grid = grid.transpose(0, 1).flatten(0, 1)
    grid = torchvision.utils.make_grid((grid + 1) * 0.5)
    return latent, grid


indices = torch.randint(len(dataset), (8,))
image, label = zip(*[dataset[i] for i in indices])
image, label = torch.stack(image).to(device), torch.tensor(label).to(device)
# TODO: vvvvvvvvvv (1 points)
# adjust the hyper parameters until you get reasonable results
latent, output = invert(generator, image, label, lr=1e-2, steps=100)
TF.to_pil_image(TF.resize(output, 128))
# ^^^^^^^^^^^^^^^^^
```

```
 0%|          | 0/100 [00:00<?, ?it/s]
```

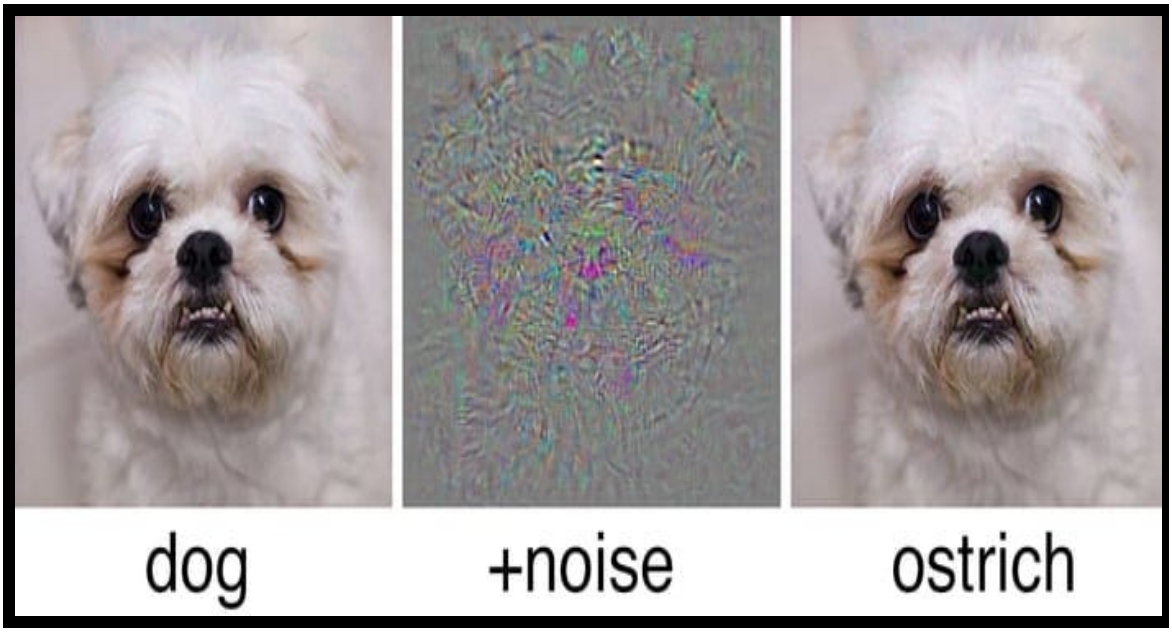
Out[23]:



Part 3: Adversarial Attacks (bonus: 6 points)

> This part is the bonus. If you finish this part correctly, you will get extra points and the maximum points of whole project is 20 points.

Another application of network inversion is adversarial attacks which was first uncovered in "Intriguing Properties of Neural Networks". The paper showed that adding some engineered noise which we refer to as an "adversarial perturbation" could cause the network to misclassify the input. Adversarial attacks could either be targeted where we want to flip the classifier's prediction into a particular class label t or untargeted where we simply want the classifier to misclassify the input sample as any other class. Below we show an example of a dog image which is misclassified as an ostrich upon adding the adversarial perturbation.



Now the main question is: How do we generate these adversarial perturbations? Optimization is the answer! For a targeted attack, we want to minimize the loss function (cross entropy) between the network's prediction and the target label

$$\min_{\delta} \mathcal{L}(f_\theta(x+\delta), t)$$

For the untargeted scenario, we want to maximize the loss function between the network's prediction and the ground truth label, that is we want the network to be penalized if it predicts the input as the correct label

$$\max_{\delta} \mathcal{L}(f_\theta(x+\delta), y)$$

In both scenarios, the attacker wants their perturbation to be impercetible under human inspection; otherwise, the attack can easily be detected! How do we ensure that? By having a norm constraint for the optimization problem! We can bound the adversarial perturbation δ into an ℓ_p ball by imposing the following constraint:

$$\Delta = \{\delta \in \mathbb{R}^d : \|\delta\|_p \leq \epsilon\}, \epsilon > 0$$

A very simple solution to this problem is offered by Fast Sign Gradient Method (FGSM) (1, 2):

$$\tilde{x} = x + \epsilon * sign(\nabla_x \mathcal{L}(\theta, x, y))$$

where $\tilde{x}$ is our attacked image, ϵ is our norm budget (radius of our ℓ-ball) and sign is +1 for a non-negative value and -1 otherwise.

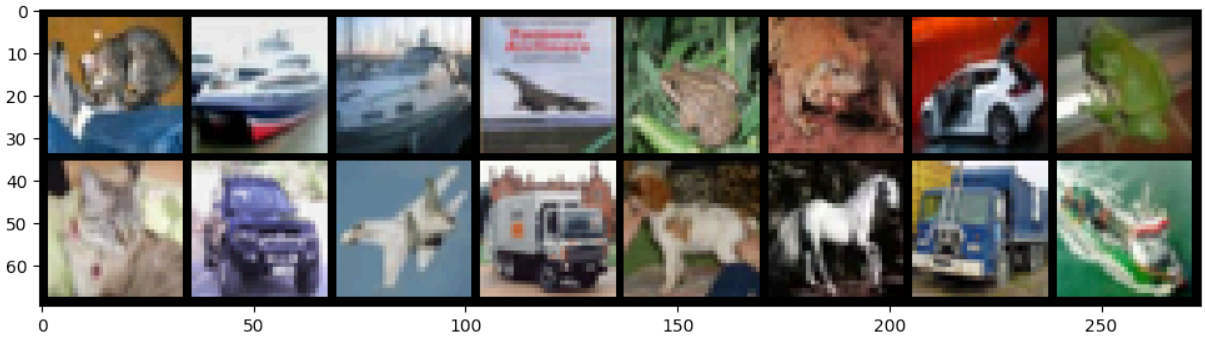
Single Image - Untargeted Attack

In [24]:
```
# We will implement a simple adversarial attack on a pretrained ResNet18 on
# We first load the model in our model in memory/GPU depending on the availa

net = torch.load("./ResNet18.pth").eval()
if torch.cuda.is_available():
    device = "cuda"
else:
    device = "cpu"

# We load CIFAR10 data and transform the images from PIL to tensor.
transform_test = T.Compose([
    T.ToTensor(),
])

testset = torchvision.datasets.CIFAR10(
    root='./data', train=False, download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(
    testset, batch_size=16, shuffle=False, num_workers=2)

# Names of classes in CIFAR10
classes = ('plane', 'car', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Files already downloaded and verified
```

In [25]:
```
# Let us visualize some of CIFAR10 images
images, labels = next(iter(testloader))
grid_img = torchvision.utils.make_grid(images, nrow=8)
fig = plt.figure(figsize=(12,12),dpi=100)
plt.imshow(grid_img.permute(1,2,0))
```

Out[25]:
```
<matplotlib.image.AxesImage at 0x7fef4a1d5520>
```



In [26]:
```
# Clone a copy of the first image in the batch
# TODO: Understand why we have to clone the image

x_pert = images[0].clone()
epsilon = torch.zeros(3,32,32)

# To optimize for epsilon we have to turn on the gradients for this variable
print("Does Epsilon Require Gradient Computation?", epsilon.requires_grad)

####
#TODO: Activate gradient computation for epsilon
# Understand why we have to turn on the gradients for epsilon
```

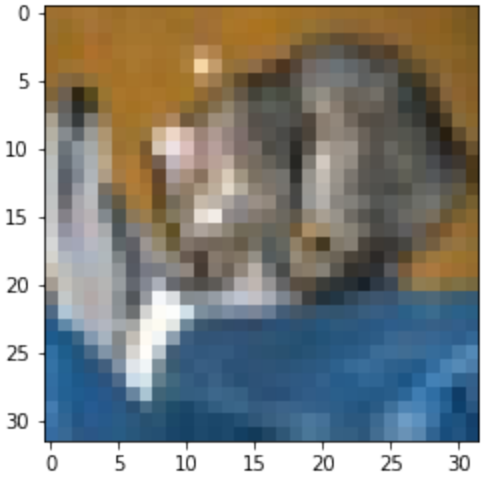
```
epsilon.requires_grad_(True)

####

print("Does Epsilon Require Gradient Computation?", epsilon.requires_grad)

# Let us check the class of the input image
print("Input Image Class:", classes[labels[0]])
```

```
Does Epsilon Require Gradient Computation? False
Does Epsilon Require Gradient Computation? True
Input Image Class: cat
```

In [27]:
```
# We will now implement a very simplistic attack on a single image
# We will use an optimizer and control the step size by the learning rate lr
# To implement a simple version of FGSM we take a single step of size = lr

#TODO: Pass the correct arguments to the optimizer parameters
# Control lr to find an adversary and attain an imperceptible image

optimizer = torch.optim.Adam(params = [epsilon], lr=1e-2)

# We remember we always reset our gradients to 0
optimizer.zero_grad()

# Forward pass
prediction = net((x_pert+epsilon).unsqueeze(0).to(device))

# Backprop and update epsilon
criterion = torch.nn.CrossEntropyLoss()
loss = -criterion(prediction, labels[0].unsqueeze(0).to(device))

loss.backward()
optimizer.step()

# Take the perturbed image and forward pass
attackedimage = x_pert+epsilon
prediction = net(attackedimage.unsqueeze(0).to(device))

# Check the class and visualize the new image
print("New Prediction:", classes[prediction.argmax(1)])
print("ell-2 Norm of the Attack:" ,torch.norm(attackedimage-x_pert,2).item()
plt.imshow(x_pert.permute(1,2,0).detach().cpu())
```

```
New Prediction: dog
ell-2 Norm of the Attack: 0.5540717840194702
```

Out[27]:
```
<matplotlib.image.AxesImage at 0x7fef4a7d6b50>
```



Single Image - Targeted Attack

We now repeat the earlier process but instead of carrying out an untargeted attack (where we maximize the loss between the network's prediction and the class label) we now need to do a targeted attack.

We first repeat the data processing carried out in the earlier cell

In [28]:
```
# Clone a copy of the first image in the batch
# TODO: Understand why we have to clone the image

x_pert = images[0].clone()
epsilon = torch.zeros(3,32,32)

# To optimize for epsilon we have to turn on the gradients for this variable
print("Does Epsilon Require Gradient Computation?", epsilon.requires_grad)

####
#TODO: Activate gradient computation for epsilon
# Understand why we have to turn on the gradients for epsilon
epsilon.requires_grad_(True)

####

print("Does Epsilon Require Gradient Computation?", epsilon.requires_grad)

# Let us check the class of the input image
print("Input Image Class:", classes[labels[0]])
```

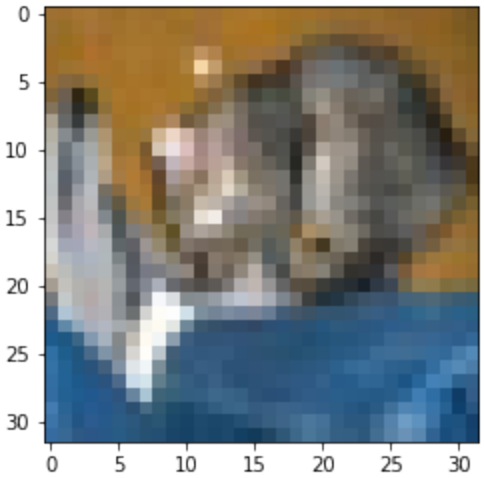
```
Does Epsilon Require Gradient Computation? False
Does Epsilon Require Gradient Computation? True
Input Image Class: cat
```

In [35]:
```
# We will now implement a very simplistic attack on a single image
# We will use an optimizer and control the step size by the learning rate lr
# To implement a simple version of FGSM we take a single step of size = lr
# Let us set lr = 0.05

#TODO: Pass the correct arguments to the optimizer parameters
```

```
# Control lr to find an adversary and attain an imperceptible image

optimizer = torch.optim.Adam(params = [epsilon], lr=0.05)

optimizer.zero_grad()

prediction = net((x_pert+epsilon).unsqueeze(0).cuda())
criterion = torch.nn.CrossEntropyLoss()

#TODO: Fill in the loss for targeted attack. Let us predict the cat image as
# Please do experiment with multiple classes and multiple input images to be

target_class = 9
loss = criterion(prediction, torch.tensor([target_class], device=device))

loss.backward()
optimizer.step()

attackedimage = x_pert+epsilon
prediction = net(attackedimage.unsqueeze(0).cuda())

print("New Prediction:", classes[prediction.argmax(1)])
print("Target Prediction:", classes[target_class])
print("ell-2 Norm of the Attack:" ,torch.norm(attackedimage-x_pert,2).item()
plt.imshow(x_pert.permute(1,2,0).detach().cpu())
```

```
New Prediction: truck
Target Prediction: truck
ell-2 Norm of the Attack: 6.813961505889893
```

Out[35]: <matplotlib.image.AxesImage at 0x7fef4a504100>



Batch Adversarial Attacks

In [57]:
```
#TODO: Extend the earlier implementations of untargeted adversarial attacks
# attack batches of images rather than single images.
# Use the same format as the earlier cells in implementing your attack.

epsilon = torch.zeros(16,3,32,32,device=device) # batch size = 16
epsilon.requires_grad_(True)
```

```
optimizer = torch.optim.Adam(params = [epsilon], lr=0.05)
criterion = torch.nn.CrossEntropyLoss()
target_class = 9

# for batch in testloader:
for i, (images, labels) in enumerate(tqdm(testloader), 1):
    images, labels = images.to(device), labels.to(device)

    # untargeted attack
    optimizer.zero_grad()
    prediction = net(images+epsilon)
    loss = -criterion(prediction, labels)
    loss.backward()
    optimizer.step()

    # Take the perturbed image and forward pass
    attackedimage = images+epsilon
    prediction = net(attackedimage)

    print("Original Prediction:", [classes[l] for l in labels])
    print("\nUntargeted New Prediction:", [classes[p] for p in prediction.ar

    # targeted attack
    optimizer.zero_grad()
    prediction = net(images+epsilon)
    loss = criterion(prediction, torch.tensor([target_class] * 16, device=de
    loss.backward()
    optimizer.step()
    attackedimage = images+epsilon
    prediction = net(attackedimage)

    print("\nTargeted New Prediction:", [classes[p] for p in prediction.argm

    break
###
```

```
  0%|          | 0/625 [00:00<?, ?it/s]
Original Prediction: ['cat', 'ship', 'ship', 'plane', 'frog', 'frog', 'car',
'frog', 'cat', 'car', 'plane', 'truck', 'dog', 'horse', 'truck', 'ship']

Untargeted New Prediction: ['dog', 'frog', 'frog', 'bird', 'deer', 'deer',
'truck', 'bird', 'bird', 'truck', 'bird', 'truck', 'frog', 'bird', 'bird',
'frog']

Targeted New Prediction: ['truck', 'truck', 'bird', 'frog', 'truck', 'frog',
'truck', 'frog', 'frog', 'truck', 'frog', 'truck', 'truck', 'horse', 'truc
k', 'bird']
```