CSCM77 Coursework

Released: 27th February 2024

Due: 22nd March 2024

1. Introduction

The tasks in this assignment are based on your practical work in the lab sessions and understanding of the theories and methods. Thus, through this coursework, you are expected to demonstrate both practical skills and theoretical knowledge of several computer vision techniques.

This work must be completed individually rather than a group work. You must accomplish on your own implementation and submit the required documentation as indicated to be individual identities. You can request helps and hints from the lecturer and lab tutors, but you cannot collaborate with your peers.

This coursework is worth 100 points and will contribute to 20% of your final marks of this module.

Submission deadline is 22nd March 12 PM and you must submit a four-page report on Canvas before the deadline. You don't need to submit your code.

Your will be assessed on both your four-page report (30 points) and your demonstration to lab tutors (70 points). Please refer to the grading rubric.

2. Task description

2.1. Variational Autoencoder

In this task, you will implement a variational autoencoder and a conditional variational autoencoder with slightly different architectures and apply them to the popular MNIST handwritten dataset. Recall from lecture, an autoencoder seeks to learn a latent representation of our training images by using unlabeled data and learning to reconstruct its inputs. The variational autoencoder extends this model by adding a probabilistic spin to the encoder and decoder, allowing us to sample from the learned distribution of the latent space to generate new images at inference time.

2.1.1 Setup Code

Before getting started, we need to run some boilerplate code to set up our environment, same as previous assignments. You'll need to rerun this setup code each time you start the notebook. First, run this cell load the autoreload extension. This allows us to edit .py source files, and re-import them into the notebook for a seamless editing and debugging experience.

```
%load_ext autoreload
```

```
%autoreload 2
```

Google Colab Setup: Next we need to run a few commands to set up our environment on Google Colab. If you are running this notebook on a local machine you can skip this section. Run the following cell to mount your Google Drive. Follow the link, sign in to your Google account (the same account you used to store this notebook!) and copy the authorization code into the text box that appears below.

```
from google.colab import drive
drive.mount('/content/drive')
Mounted at /content/drive
```

Now recall the path in your Google Drive where you uploaded this notebook, fill it in below. If everything is working correctly then running the following cell should print the filenames from the assignment:

```
['cscm77', 'gan.py', 'generative_adversarial_networks.ipynb',
'cscm77_helper.py', 'vae.py',
'variational_autoencoders.ipynb']
```

Once you have successfully mounted your Google Drive and located the path to this assignment, run the following cell to allow us to import from the .py files of this assignment. If it works correctly, it should print the message:

```
Hello from vae.py!
```

```
Hello from cscm77_helper.py!
```

as well as the last edit time for the file *vae.py*.

We will use GPUs to accelerate our computation in this notebook. Run the following to make sure GPUs are enabled:

```
if torch.cuda.is_available():
  print('Good to go!')
else:
  print('Please set GPU via Edit -> Notebook Settings.')
```

```
Good to go!
```

2.1.2. Load the MNIST dataset

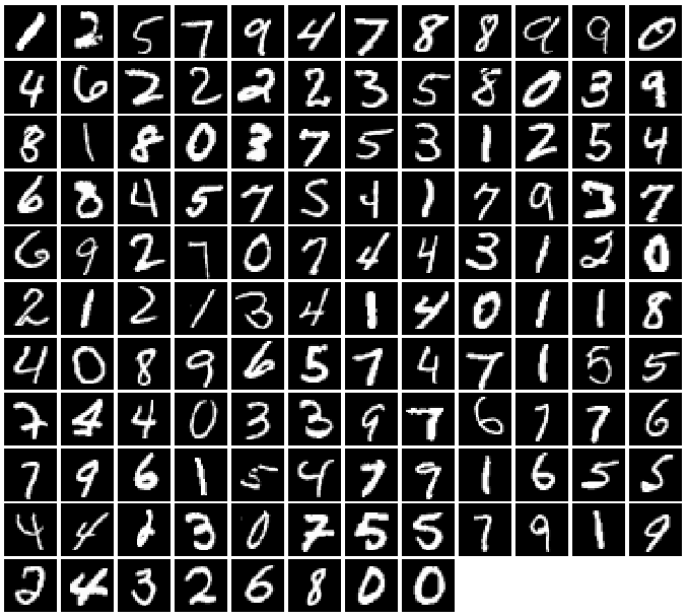
VAEs (and GANs as you'll see in the next notebook) are notoriously finicky with hyperparameters, and also require many training epochs. In order to make this assignment approachable, we will be working on the MNIST dataset, which is 60,000 training and 10,000 test images. Each picture contains a centered image of white digit on black background (0 through 9). This was one of the first datasets used to train

convolutional neural networks and it is fairly easy -- a standard CNN model can easily exceed 99% accuracy. To simplify our code here, we will use the PyTorch MNIST wrapper, which downloads and loads the MNIST dataset. The default parameters will take 5,000 of the training examples and place them into a validation dataset. The data will be saved into a folder called *MNIST_data*.

```
batch_size = 128

mnist_train = dset.MNIST('./MNIST_data', train=True, download=True,
                           transform=T.ToTensor())
loader_train = DataLoader(mnist_train, batch_size=batch_size,
                          shuffle=True, drop_last=True, num_workers=2)
```

It is always a good idea to look at examples from the dataset before working with it. Let's visualize the digits in the MNIST dataset. We have defined the function show_images in *cscm77_helper.py* that we call to visualize the images.

```
from cscm77_helper import show_images

imgs = loader_train.__iter__().next()[0].view(batch_size, 784)
show_images(imgs)
```



2.1.3. Fully-connected VAE

Our first VAE implementation will consist solely of fully connected layers. We'll take the 1 x 28 x 28 shape of our input and flatten the features to create an input dimension size of 784. In this section you'll define the Encoder and Decoder models in the VAE class of *vae.py* and implement the reparametrization trick, forward pass, and loss function to train your first VAE.

FC-VAE Encoder

Now lets start building our fully-connected VAE network. We'll start with the encoder, which will take our images as input (after flattening C,H,W to D shape) and pass them through a three Linear+ReLU layers. We'll use this hidden dimension representation to predict both the posterior mu and posterior log-variance using two separate linear layers (both shape (N,Z)).

Note that we are calling this the 'logvar' layer because we'll use the log-variance (instead of variance or standard deviation) to stabilize training. This will specifically matter more when you compute reparametrization and the loss function later.

Define an encoder, hidden_dim (H), mu_layer, and logvar_layer in the initialization of the VAE class in vae.py. Use nn.Sequential to define the encoder, and separate Linear layers for the mu and logvar layers. In all of these layers, H will be a hidden dimension you set and will be the same across all encoder and decoder layers. Architecture for the encoder is described below:

- Flatten (Hint: nn.Flatten)
- Fully connected layer with input size 784 (input_size) and output size H
- ReLU
- Fully connected layer with input_size H and output size H
- ReLU
- Fully connected layer with input_size H and output size H
- ReLU

FC-VAE Decoder

We'll now define the decoder, which will take the latent space representation and generate a reconstructed image. The architecture is as follows:

- Fully connected layer with input size as the latent size (Z) and output size H
- ReLU
- Fully connected layer with input_size H and output size H
- ReLU
- Fully connected layer with input_size H and output size H
- ReLU
- Fully connected layer with input_size H and output size 784 (input_size)
- Sigmoid
- Unflatten (nn.Unflatten)

Define a decoder in the initialization of the VAE class in vae.py. Like the encoding step, use nn.Sequential

2.1.4. Reparametrization

Now we'll apply a reparametrization trick in order to estimate the posterior during our forward pass, given the μ and σ^2 estimated by the encoder. A simple way to do this could be to simply generate a normal distribution centered at our μ and having a std corresponding to our σ^2. However, we would have to backpropogate through this random sampling that is not differentiable. Instead, we sample initial random data ϵ from a fixed distribution, and compute z as a function of (ϵ,μ, σ^2). Specifically: $z=\mu+\sigma\epsilon$. We can easily find the partial derivatives w.r.t μ and σ^2 and backpropagate through z. If $\epsilon = N(0,1)$, then its easy to verify that the result of our forward pass calculation will be a distribution centered at μ with variance σ^2. Implement reparametrization in vae.py and verify your mean and std error are at or less than 1e-4.

```
reset_seed(0)
from vae import reparametrize
latent_size = 15
size = (1, latent_size)
mu = torch.zeros(size)
logvar = torch.ones(size)

z = reparametrize(mu, logvar)

expected_mean = torch.FloatTensor([-0.4363])
expected_std = torch.FloatTensor([1.6860])
z_mean = torch.mean(z, dim=-1)
z_std = torch.std(z, dim=-1)
assert z.size() == size

print('Mean Error', rel_error(z_mean, expected_mean))
print('Std Error', rel_error(z_std, expected_std))
```

Below is the expected output:

```
Mean Error 5.639056398351415e-05
Std Error 7.1412955526273885e-06
```

2.1.5. FC-VAE forward

Complete the VAE class by writing the forward pass. The forward pass should pass the input image through the encoder to calculate the estimation of mu and logvar, reparametrize to estimate the latent space z, and finally pass z into the decoder to generate an image.

Loss Function

Before we're able to train our final model, we'll need to define our loss function. As seen below, the loss function for VAEs contains two terms: A reconstruction loss term (left) and KL divergence term (right).

$$-E_{Z\,q_\phi(z|x)}[logp_\theta(x|z)] + D_{KL}(q_\phi(z|x), p(z)))$$

Note that this is the negative of the variational lowerbound shown in lecture--this ensures that when we are minimizing this loss term, we're maximizing the variational lowerbound. The reconstruction loss term can be computed by simply using the binary cross entropy loss between the original input pixels and the output pixels of our decoder (Hint: nn.functional.binary_cross_entropy). The KL divergence term works to force the latent space distribution to be close to a prior distribution (we're using a standard normal gaussian as our prior).

To help you out, we've derived an unvectorized form of the KL divergence term for you. Suppose that $q_\phi(z|x)$ is a Z-dimensional diagonal Gaussian with mean $\mu_{z|x}$ of shape (Z,) and standard deviation $\sigma_{z|x}$ of shape (Z,), and that p(z) is a Z-dimensional

Gaussian with zero mean and unit variance. Then we can write the KL divergence term as:

$$D_{KL}(q_\phi(z|x), p(z))) = -\frac{1}{2}\sum_{j=1}^{J}(1 + log(\sigma^2_{z|x})_j - (\mu_{z|x})_j^2 - (\sigma_{z|x})_j^2)$$

It's up to you to implement a vectorized version of this loss that also operates on minibatches. You should average the loss across samples in the minibatch. Implement loss_function in vae.py and verify your implementation below. Your relative error should be less than or equal to 1e-5.

```
from vae import loss_function
size = (1,15)

image = torch.sigmoid(torch.FloatTensor([[2,5],
[6,7]]).unsqueeze(0).unsqueeze(0))
image_hat = torch.sigmoid(torch.FloatTensor([[1,10],
[9,3]]).unsqueeze(0).unsqueeze(0))

expected_out = torch.tensor(8.5079)
mu, logvar = torch.ones(size), torch.zeros(size)
out = loss_function(image, image_hat, mu, logvar)
print('Loss error', rel_error(expected_out,out))
```

Below is the expected output:

```
Loss error 2.1297676389877955e-06
```

2.1.6. Train a model and visual results

Now that we have our VAE defined and loss function ready, lets train our model! Our training script is provided in cscm77_helper.py, and we have pre-defined an Adam optimizer, learning rate, and # of epochs for you to use. Training for 10 epochs should take ~2 minutes and your loss should be less than 120.

```
num_epochs = 10
latent_size = 15
from vae import VAE
from cscm77_helper import train_vae
input_size = 28*28
device = 'cuda'
vae_model = VAE(input_size, latent_size=latent_size)
vae_model.cuda()
for epoch in range(0, num_epochs):
  train_vae(epoch, vae_model, loader_train)
```

Below is the expected output:

```
Train Epoch: 0 Loss: 163.427429
Train Epoch: 1 Loss: 138.039627
Train Epoch: 2 Loss: 123.292709
Train Epoch: 3 Loss: 121.097710
Train Epoch: 4 Loss: 125.457184
```

```
Train Epoch: 5 Loss: 114.404221
Train Epoch: 6 Loss: 116.560722
Train Epoch: 7 Loss: 119.169411
Train Epoch: 8 Loss: 112.095634
Train Epoch: 9 Loss: 109.473122
```

Visualize your results: After training our VAE network, we're able to take advantage of its power to generate new training examples. This process simply involves the decoder: we intialize some random distribution for our latent spaces z, and generate new examples by passing these latent space into the decoder.

Run the cell below to generate new images! You should be able to visually recognize many of the digits, although some may be a bit blurry or badly formed. Our next model will see improvement in these results.

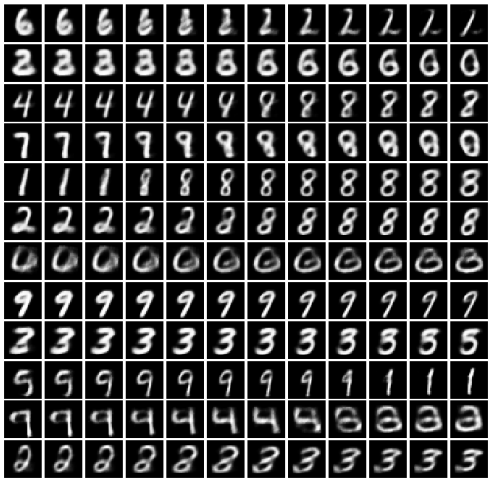
```
z = torch.randn(10, latent_size).to(device='cuda')
import matplotlib.gridspec as gridspec
vae_model.eval()
samples = vae_model.decoder(z).data.cpu().numpy()

fig = plt.figure(figsize=(10, 1))
gspec = gridspec.GridSpec(1, 10)
gspec.update(wspace=0.05, hspace=0.05)
for i, sample in enumerate(samples):
  ax = plt.subplot(gspec[i])
  plt.axis('off')
  ax.set_xticklabels([])
  ax.set_yticklabels([])
  ax.set_aspect('equal')
  plt.imshow(sample.reshape(28,28), cmap='Greys_r')
  plt.savefig(os.path.join(GOOGLE_DRIVE_PATH,'vae_generation.jpg'))
```

2.1.7. Latent space interpolation

As a final visual test of our trained VAE model, we can perform interpolation in latent space. We generate random latent vectors z_0 and z_1, and linearly interpolate between them; we run each interpolated vector through the trained generator to produce an image. Each row of the figure below interpolates between two random vectors. For the most part the model should exhibit smooth transitions along each row, demonstrating that the model has learned something nontrivial about the underlying spatial structure of the digits it is modeling.

```
S = 12
latent_size = 15
device = 'cuda'
```

```
z0 = torch.randn(S,latent_size , device=device)
z1 = torch.randn(S, latent_size, device=device)
w = torch.linspace(0, 1, S, device=device).view(S, 1, 1)
z = (w * z0 + (1 - w) * z1).transpose(0, 1).reshape(S * S, latent_size)
x = vae_model.decoder(z)
show_images(x.data.cpu())
```



2.1.8. Conditional FC-VAE

The second model you'll develop will be very similar to the FC-VAE, but with a slight conditional twist to it. We'll use what we know about the labels of each MNIST image, and *condition* our latent space and image generation on the specific class. Instead of $q_\phi(z|x)$ and $p_\phi(x|z)$ we have $q_\phi(z|x,c)$ and $p_\phi(x|z,c)$. This will allow us to do some powerful conditional generation at inference time. We can specifically choose to generate more 1s, 2s, 9s, etc. instead of simply generating new digits randomly.

Define Network with class input

Our CVAE architecture will be the same as our FC-VAE architecture, except we'll now add a one-hot label vector to both the x input (in our case, the flattened image dimensions) and the z latent space. If our one-hot vector is called `c`, then `c[label] = 1` and `c = 0` elsewhere.

For the `CVAE` class in `vae.py` use the same FC-VAE architecture implemented in the last network with the following modifications:

1. Modify the first linear layer of your `encoder` to take in not only the flattened input image, but also the one-hot label vector `c`
2. Modify the first layer of your `decoder` to project the latent space + one-hot vector to the `hidden_dim`
3. Lastly, implement the forward pass to combine the flattened input image with the one-hot vectors (`torch.cat`) before passing them to the encoder and combine the latent space with the one-hot vectors (`torch.cat`) before passing them to the `decoder`

Train the model

Using the same training script, let's now train our CVAE! Training for 10 epochs should take ~2 minutes and your loss should be less than 120.

```
from vae import CVAE
num_epochs = 10
latent_size = 15
from cscm77_helper import train_vae
input_size = 28*28
device = 'cuda'

cvae = CVAE(input_size, latent_size=latent_size)
cvae.cuda()
for epoch in range(0, num_epochs):
  train_vae(epoch, cvae, loader_train, cond=True)
```

Below is the expected output:

```
Train Epoch: 0  Loss: 143.905441
Train Epoch: 1  Loss: 131.232330
Train Epoch: 2  Loss: 126.080849
Train Epoch: 3  Loss: 119.384857
Train Epoch: 4  Loss: 111.767593
Train Epoch: 5  Loss: 111.876450
Train Epoch: 6  Loss: 119.063759
Train Epoch: 7  Loss: 110.267746
Train Epoch: 8  Loss: 109.231987
Train Epoch: 9  Loss: 102.071381
```

Visualize Results

We've trained our CVAE, now lets conditionally generate some new data! This time, we can specify the class we want to generate by adding our one hot matrix of class labels. We use `torch.eye` to create an identity matrix, gives effectively gives us one label for each digit. When you run the cell below, you should get one example per digit. Each digit should be reasonably distinguishable (it is ok to run this cell a few times to save your best results).

```
z = torch.randn(10, latent_size)
c = torch.eye(10, 10) # [one hot labels for 0-9]
import matplotlib.gridspec as gridspec
z = torch.cat((z,c), dim=-1).to(device='cuda')
cvae.eval()
samples = cvae.decoder(z).data.cpu().numpy()

fig = plt.figure(figsize=(10, 1))
gspec = gridspec.GridSpec(1, 10)
gspec.update(wspace=0.05, hspace=0.05)
for i, sample in enumerate(samples):
  ax = plt.subplot(gspec[i])
  plt.axis('off')
  ax.set_xticklabels([])
  ax.set_yticklabels([])
  ax.set_aspect('equal')
  plt.imshow(sample.reshape(28, 28), cmap='Greys_r')
```

```
plt.savefig(os.path.join(GOOGLE_DRIVE_PATH,'conditional_vae_generation.jpg'
))
```

2.2. Generative Adversarial Networks

In this assignment, you will learn to build generative models using neural networks. Specifically, you will learn how to build models which generate novel images that resemble a set of training images. In 2014, Goodfellow et al. presented a method for training generative models called Generative Adversarial Networks (GANs for short). In a GAN, we build two different neural networks. Our first network is a traditional classification network, called the **discriminator**. We will train the discriminator to take images, and classify them as being real (belonging to the training set) or fake (not present in the training set). Our other network, called the **generator**, will take random noise as input and transform it using a neural network to produce images. The goal of the generator is to fool the discriminator into thinking the images it produced are real.

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

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

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

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

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

While these updates are useful for analysis, they do not perform well in practice. Instead, we will use a different objective when we update the generator: maximize the probability of the discriminator making the incorrect choice. This small change helps to

alleviate problems with the generator gradient vanishing when the discriminator is confident. This is the standard update used in most GAN papers, and was used in the original paper from Goodfellow et al.

In this assignment, we will alternate the following updates:

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

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

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

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

[1] Generative Adversarial Networks. Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio.

Setup code

Before getting started, we need to run some boilerplate code to set up our environment, same as previous assignments. You'll need to rerun this setup code each time you start the assignment.

Google Colab Setup

Please refer to the previous VAE sector.

Note that if CUDA is not enabled, torch.cuda.is_available() will return False and this work sheet will fallback to CPU mode. The global variables dtype and device will control the data types throughout this assignment. We will be using torch.float = torch.float32 for all operations.

Dataset

We will be using MNIST dataset. To download and setup the dataset, see the hints in VAE implementation for details. The default parameters will take 5,000 of the training examples and place them into a validation dataset. The data will be saved into a folder called MNIST_data.

2.2.1. Random noise, Discriminator, Generator

Random noise: The first step is to generate uniform noise from -1 to 1 with shape [batch_size, noise_dim]. Hint: use torch.rand. Implement sample_noise and verify all tests pass below.

```
from gan import sample_noise
reset_seed(0)


batch_size = 3
noise_dim = 4

z = sample_noise(batch_size, noise_dim)
assert z.shape == (batch_size, noise_dim)
assert torch.is_tensor(z)
assert torch.all(z >= -1.0) and torch.all(z <= 1.0)
assert torch.any(z < 0.0) and torch.any(z > 0.0)
print('All tests passed!')
```

Discriminator: Our first step is to build a discriminator. All fully connected layers should include bias terms. The architecture is:

- Fully connected layer with input size 784 and output size 256
- LeakyReLU with alpha 0.01
- Fully connected layer with input_size 256 and output size 256
- LeakyReLU with alpha 0.01
- Fully connected layer with input size 256 and output size 1

Recall that the Leaky ReLU nonlinearity computes $f(x)=\max(\alpha x,x)$ for some fixed constant α; for the LeakyReLU nonlinearities in the architecture above we set $\alpha=0.001$. The output of the discriminator should have shape [batch_size, 1], and contain real numbers corresponding to the scores that each of the batch_size inputs is a real image. Implement discriminator in gan.py and test your solution by running the cell below.

```
from gan import discriminator

def test_discriminator(true_count=267009):
  model = discriminator()
  cur_count = count_params(model)
  print(cur_count)
  if cur_count != true_count:
    print('Incorrect number of parameters in discriminator. Check your
achitecture.')
  else:
    print('Correct number of parameters in discriminator.')

test_discriminator()
```

```
Below is the expected output:
267009
Correct number of parameters in discriminator.
```

Generator: Now to build the generator network:

- Fully connected layer from noise_dim to 1024
- ReLU
- Fully connected layer with size 1024

- ReLU
- Fully connected layer with size 784
- TanH (to clip the image to be in the range of [-1,1])

Implement `generator` in `gan.py` and test your solution by running the cell below.

```
from gan import generator

def test_generator(true_count=1858320):
  model = generator(4)
  cur_count = count_params(model)
  print(cur_count)
  if cur_count != true_count:
    print('Incorrect number of parameters in generator. Check your
achitecture.')
  else:
    print('Correct number of parameters in generator.')

test_generator()
```

```
Below is the expected output:
1858320
Correct number of parameters in generator.
```

2.2.2. GAN loss

Compute the generator and discriminator loss. The generator loss is:

$$\ell_G = -\mathbb{E}_{z\sim p(z)}\left[\log D(G(z))\right]$$

And the discriminator loss is

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

Note that these are negated from the equations presented earlier as we will be minimizing these losses.

For the purpose of these equations, we assume that the output from the discriminator is a real number in the range 0<D(x)<1, which results from squashing the raw score from the discriminator through a sigmoid function. However, for a cleaner and more numerically stable implementation, we have not included the sigmoid in the discriminator architecture above -- instead we will implement the sigmoid as part of the loss function.

HINTS: You can use the function `torch.nn.functional.binary_cross_entropy_with_logits` to compute these losses in a numerically stable manner.

Given a score sϵR and a label yϵ{0,1} , the binary cross entropy loss (with logits) is defined as:

$$bce(s, y) = -y * \log(\sigma(s)) - (1 - y) * \log(1 - \sigma(s))$$

Where σ(s)=1/(1+exp(-s)) is the sigmoid function. A naive implementation of this formula can be numerically unstable, so you should prefer to use the built-in PyTorch implementation. You will also need to compute labels corresponding to real or fake and use the logit arguments to determine their size. Make sure you cast these labels to the correct data type using the global dtype variable, for example:

true_labels = torch.ones(size, device=device)

Instead of computing the expectation of log D(G(z)), log D(x) and log(1-D(G(z))), we will be averaging over elements of the minibatch, so make sure to combine the loss by averaging instead of summing. Implement discriminator_loss and generator_loss in gan.py and test your solution by running the cell below. You should see errors < 1e-7.

```
from gan import discriminator_loss

answers['logits_fake'] = torch.tensor(
  [-1.80865868,  0.09030055, -0.4428902 , -0.07879368, -0.37655044,
    0.32084742, -0.28590837,  1.01376281,  0.99241439,  0.39394346],
  dtype=dtype, device=device)
answers['d_loss_true'] = torch.tensor(1.8423983904443109, dtype=dtype,
device=device)
answers['logits_real'] = torch.tensor(
  [ 0.93487311, -1.01698916, -0.57304769, -0.88162704, -1.40129389,
   -1.45395693, -1.54239755, -0.57273325,  0.98584429,  0.13312152],
  dtype=dtype, device=device)

def test_discriminator_loss(logits_real, logits_fake, d_loss_true):
  d_loss = discriminator_loss(logits_real, logits_fake)
  print("Maximum error in d_loss: %g"%rel_error(d_loss_true, d_loss))
test_discriminator_loss(answers['logits_real'], answers['logits_fake'],
                        answers['d_loss_true'])
```

```
Below is the expected output:
Maximum error in d_loss: 0
```

```
from gan import generator_loss

answers['g_loss_true'] = torch.tensor(0.771286196423346, dtype=dtype,
device=device)

def test_generator_loss(logits_fake, g_loss_true):
  g_loss = generator_loss(logits_fake)
  print("Maximum error in g_loss: %g"%rel_error(g_loss_true, g_loss))

test_generator_loss(answers['logits_fake'], answers['g_loss_true'])
```

```
Below is the expected output:
Maximum error in g_loss: 0
```

Optimize our loss: Next, you'll define a function that returns an optim.Adam optimizer for the given model with a 1e-3 learning rate, beta1=0.5, beta2=0.999. We'll use this to construct optimizers for the generators and discriminators for the rest of the notebook in get_optimizer. Implement get_optimizer in gan.py before moving forward.

2.2.3. Train a GAN

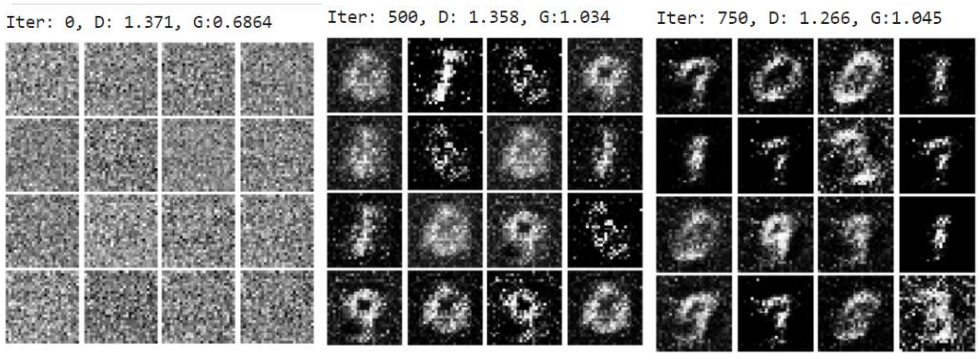
We provide you the main training loop... you won't need to change this function, but we encourage you to read through and understand it.

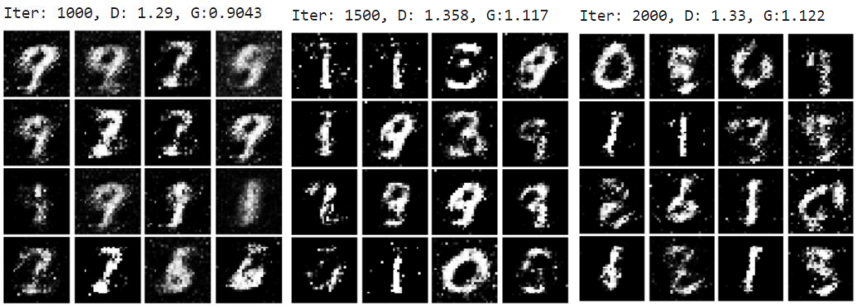
```
def run_a_gan(D, G, D_solver, G_solver, discriminator_loss, generator_loss,
save_filename, show_every=250,
              batch_size=128, noise_size=96, num_epochs=10):
  """
  Train a GAN!

  Inputs:
  - D, G: PyTorch models for the discriminator and generator
  - D_solver, G_solver: torch.optim Optimizers to use for training the
    discriminator and generator.
  - discriminator_loss, generator_loss: Functions to use for computing the
generator and
    discriminator loss, respectively.
  - show_every: Show samples after every show_every iterations.
  - batch_size: Batch size to use for training.
  - noise_size: Dimension of the noise to use as input to the generator.
  - num_epochs: Number of epochs over the training dataset to use for
training.
  """
  iter_count = 0
  for epoch in range(num_epochs):
    for x, _ in loader_train:
      if len(x) != batch_size:
        continue
      D_solver.zero_grad()
      real_data = x.view(-1, 784).to(device)
      logits_real = D(2* (real_data - 0.5))

      g_fake_seed = sample_noise(batch_size, noise_size,
dtype=real_data.dtype, device=real_data.device)
      fake_images = G(g_fake_seed).detach()
      logits_fake = D(fake_images)

      d_total_error = discriminator_loss(logits_real, logits_fake)
      d_total_error.backward()
      D_solver.step()

      G_solver.zero_grad()
      g_fake_seed = sample_noise(batch_size, noise_size,
dtype=real_data.dtype, device=real_data.device)
      fake_images = G(g_fake_seed)

      gen_logits_fake = D(fake_images)
      g_error = generator_loss(gen_logits_fake)
      g_error.backward()
```

```
      G_solver.step()

      if (iter_count % show_every == 0):
        print('Iter: {}, D: {:.4},
G:{:.4}'.format(iter_count,d_total_error.item(),g_error.item()))
        imgs_numpy = fake_images.data.cpu()#.numpy()
        show_images(imgs_numpy[0:16])
        plt.show()
        print()
      iter_count += 1
    if epoch == num_epochs - 1:
      show_images(imgs_numpy[0:16])
      plt.savefig(os.path.join(GOOGLE_DRIVE_PATH,save_filename))
```

Now run the cell below to train your first GAN! Your last epoch results will be stored in fc_gan_results.jpg for you to submit to the autograder.

```
from gan import get_optimizer
reset_seed(0)

# Make the discriminator
D = discriminator().to(device)

# Make the generator
G = generator().to(device)

# Use the function you wrote earlier to get optimizers for the
Discriminator and the Generator
D_solver = get_optimizer(D)
G_solver = get_optimizer(G)
# Run it!
run_a_gan(D, G, D_solver, G_solver, discriminator_loss, generator_loss,
'fc_gan_results.jpg')
```





Well that wasn't so hard, was it? In the iterations in the low 100s you should see black backgrounds, fuzzy shapes as you approach iteration 1000, and decent shapes, about half of which will be sharp and clearly recognizable as we pass 3000.

2.2.4. Deeply convolutional GAN

In the first part of the assignment, we implemented an almost direct copy of the original GAN network from Ian Goodfellow. However, this network architecture allows no real spatial reasoning. It is unable to reason about things like "sharp edges" in general because it lacks any convolutional layers. Thus, in this section, we will implement some of the ideas from DCGAN, where we use convolutional networks.

Discriminator

We will use a discriminator inspired by the TensorFlow MNIST classification tutorial, which is able to get above 99% accuracy on the MNIST dataset fairly quickly.

- Reshape into image tensor (Use nn.Unflatten!)
- Conv2D: 32 Filters, 5x5, Stride 1
- Leaky ReLU(alpha=0.01)
- Max Pool 2x2, Stride 2
- Conv2D: 64 Filters, 5x5, Stride 1
- Leaky ReLU(alpha=0.01)
- Max Pool 2x2, Stride 2
- Flatten
- Fully Connected with output size 4 x 4 x 64
- Leaky ReLU(alpha=0.01)
- Fully Connected with output size 1

```
from gan import build_dc_classifier

data = next(enumerate(loader_train))[-1][0].to(dtype=dtype,
device=device)
batch_size = data.size(0)
b = build_dc_classifier().to(device)
data = data.view(-1, 784)
out = b(data)
print(out.size())
torch.Size([128, 1])
```

- Check the number of parameters in your classifier as a sanity check:

```
def test_dc_classifer(true_count=1102721):
  model = build_dc_classifier()
  cur_count = count_params(model)
  print(cur_count)
  if cur_count != true_count:
    print('Incorrect number of parameters in generator. Check your achitecture.')
  else:
    print('Correct number of parameters in generator.')

test_dc_classifer()
1102721
Correct number of parameters in generator.
```

Generator

For the generator, we will copy the architecture exactly from the InfoGAN paper. See Appendix C.1 MNIST. See the documentation for nn.ConvTranspose2d. We are always "training" in GAN mode.

- Fully connected with output size 1024
- ReLU
- BatchNorm
- Fully connected with output size 7 x 7 x 128
- ReLU
- BatchNorm
- Reshape into Image Tensor of shape 7 x 7 x 128
- Conv2D^T (Transpose): 64 filters of 4x4, stride 2, 'same' padding (use padding=1)
- ReLU
- BatchNorm
- Conv2D^T (Transpose): 1 filter of 4x4, stride 2, 'same' padding (use padding=1)
- TanH
- Should have a 28 x 28 x 1 image, reshape back into 784 vector

```
from gan import build_dc_generator

test_g_gan = build_dc_generator().to(device)
test_g_gan.apply(initialize_weights)

fake_seed = torch.randn(batch_size, NOISE_DIM, dtype=dtype, device=device)
fake_images = test_g_gan.forward(fake_seed)
fake_images.size()
```

- Out[32]:

```
torch.Size([128, 784])
```

- Check the number of parameters in your generator as a sanity check:
- In [33]:

```
def test_dc_generator(true_count=6580801):
  model = build_dc_generator(4)
  cur_count = count_params(model)
  print(cur_count)
```

```
    if cur_count != true_count:
      print('Incorrect number of parameters in generator. Check your
achitecture.')
    else:
      print('Correct number of parameters in generator.')

test_dc_generator()
6580801
Correct number of parameters in generator.
```

Now, let's train our DC-GAN!

```
reset_seed(0)

D_DC = build_dc_classifier().to(device)
D_DC.apply(initialize_weights)
G_DC = build_dc_generator().to(device)
G_DC.apply(initialize_weights)

D_DC_solver = get_optimizer(D_DC)
G_DC_solver = get_optimizer(G_DC)

run_a_gan(D_DC, G_DC, D_DC_solver, G_DC_solver, discriminator_loss,
generator_loss, 'dc_gan_results.jpg', num_epochs=5)
```

3. Implementation

The starter code of this coursework is provided on Canvas.

You need to write a four-page report on your work and submit to Canvas (the acceptable format is PDF only). The report must contain the following structure:

Introduction (10%). Contextualise the machine-learning problem and introduce the task and the hypothesis. Make sure to include a few references to previous published work in the field. You should demonstrate an awareness of the research-area.

Methodology (50%). The model(s) you trained to undertake the task. Any decisions on hyperparameters must be stated here, including motivation for your choices where

applicable. If the basis of your decision is experimentation with a number of parameters, then state this.

Results (30%). Describe, compare and contrast the results you obtained on your model(s). Any relationships in the data should be outlined and pointed out here. Only the most important conclusions should be mentioned in the text. By using tables and confusion-matrices to support the section, you can avoid describing the results fully.

Discussion and Conclusion (10%). Restate the task and hypothesis concisely. Reiterate the methods used. Describe the outcome of the experiment and the conclusion that you can draw from these results in respect of the hypothesis.

4. Submission and grading criteria

Your report work will be assessed on its structure, content, and presentation. We expect it to be read as an academic paper, with the explanation appropriately divided as per the structure described in the Task Description above. The full marks of your report is 30 points.

We are running a viva on the lab session of **22nd March Friday 11 AM-12 PM on PC lab 102**. Please mark the date and time on your calendar and come along to demonstrate your framework. You should demonstrate your knowledge of the field, along with any conclusions you can draw from your results. We will mark you on the spot. The full marks of your demonstration is 70 points.