Transformer_Captioning

December 3, 2023

[1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs682/assignment3/'
FOLDERNAME = 'cs682/assignment3/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the COCO dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs682/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs682/assignment3/cs682/datasets
/content/drive/My Drive/cs682/assignment3
```

1 Image Captioning with Transformers

You have now implemented a vanilla RNN and for the task of image captioning. In this notebook you will implement key pieces of a transformer decoder to accomplish the same task.

NOTE: This notebook will be primarily written in PyTorch rather than NumPy, unlike the RNN notebook.

[2]:
```
# Setup cell.
import time, os, json
import numpy as np
import matplotlib.pyplot as plt
```

```
from cs682.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs682.transformer_layers import *
from cs682.captioning_solver_transformer import CaptioningSolverTransformer
from cs682.classifiers.transformer import CaptioningTransformer
from cs682.coco_utils import load_coco_data, sample_coco_minibatch,␣
 ↪decode_captions
from cs682.image_utils import image_from_url

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

%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

2 COCO Dataset

As in the previous notebooks, we will use the COCO dataset for captioning.

[3]:
```
# Load COCO data from disk into a dictionary.
data = load_coco_data(pca_features=True)

# Print out all the keys and values from the data dictionary.
for k, v in data.items():
    if type(v) == np.ndarray:
        print(k, type(v), v.shape, v.dtype)
    else:
        print(k, type(v), len(v))
```

```
base dir
/content/drive/MyDrive/cs682/assignment3/cs682/datasets/coco_captioning
train_captions <class 'numpy.ndarray'> (400135, 17) int32
train_image_idxs <class 'numpy.ndarray'> (400135,) int32
val_captions <class 'numpy.ndarray'> (195954, 17) int32
val_image_idxs <class 'numpy.ndarray'> (195954,) int32
train_features <class 'numpy.ndarray'> (82783, 512) float32
val_features <class 'numpy.ndarray'> (40504, 512) float32
idx_to_word <class 'list'> 1004
word_to_idx <class 'dict'> 1004
train_urls <class 'numpy.ndarray'> (82783,) <U63
val_urls <class 'numpy.ndarray'> (40504,) <U63
```

3 Transformer

As you have seen, RNNs are incredibly powerful but often slow to train. Further, RNNs struggle to encode long-range dependencies (though LSTMs are one way of mitigating the issue). In 2017, Vaswani et al introduced the Transformer in their paper "Attention Is All You Need" to a) introduce parallelism and b) allow models to learn long-range dependencies. The paper not only led to famous models like BERT and GPT in the natural language processing community, but also an explosion of interest across fields, including vision. While here we introduce the model in the context of image captioning, the idea of attention itself is much more general.

4 Transformer: Multi-Headed Attention

4.0.1 Dot-Product Attention

Recall that attention can be viewed as an operation on a query $q \in \mathbb{R}^d$, a set of value vectors $\{v_1, ..., v_n\}, v_i \in \mathbb{R}^d$, and a set of key vectors $\{k_1, ..., k_n\}, k_i \in \mathbb{R}^d$, specified as

$$c = \sum_{i=1}^{n} v_i \alpha_i \alpha_i = \frac{\exp(k_i^\top q)}{\sum_{j=1}^{n} \exp(k_j^\top q)} \tag{1}$$

$$\tag{2}$$

where α_i are frequently called the "attention weights", and the output $c \in \mathbb{R}^d$ is a correspondingly weighted average over the value vectors.

4.0.2 Self-Attention

In Transformers, we perform self-attention, which means that the values, keys and query are derived from the input $X \in \mathbb{R}^{\ell \times d}$, where ℓ is our sequence length. Specifically, we learn parameter matrices $V, K, Q \in \mathbb{R}^{d \times d}$ to map our input X as follows:

$$v_i = V x_i \quad i \in \{1, ..., \ell\} \tag{3}$$
$$k_i = K x_i \quad i \in \{1, ..., \ell\} \tag{4}$$
$$q_i = Q x_i \quad i \in \{1, ..., \ell\} \tag{5}$$

4.0.3 Multi-Headed Scaled Dot-Product Attention

In the case of multi-headed attention, we learn a parameter matrix for each head, which gives the model more expressivity to attend to different parts of the input. Let h be number of heads, and Y_i be the attention output of head i. Thus we learn individual matrices Q_i, K_i and V_i. To keep our overall computation the same as the single-headed case, we choose $Q_i \in \mathbb{R}^{d \times d/h}$, $K_i \in \mathbb{R}^{d \times d/h}$ and $V_i \in \mathbb{R}^{d \times d/h}$. Adding in a scaling term $\frac{1}{\sqrt{d/h}}$ to our simple dot-product attention above, we have

$$Y_i = \text{softmax}\left(\frac{(XQ_i)(XK_i)^\top}{\sqrt{d/h}}\right)(XV_i) \tag{6}$$

where $Y_i \in \mathbb{R}^{\ell \times d/h}$, where ℓ is our sequence length.

In our implementation, we apply dropout to the attention weights (though in practice it could be used at any step):

$$Y_i = \text{dropout}\left(\text{softmax}\left(\frac{(XQ_i)(XK_i)^\top}{\sqrt{d/h}}\right)\right)(XV_i) \tag{7}$$

Finally, then the output of the self-attention is a linear transformation of the concatenation of the heads:

$$Y = [Y_1; \ldots; Y_h]A \tag{8}$$

were $A \in \mathbb{R}^{d \times d}$ and $[Y_1; \ldots; Y_h] \in \mathbb{R}^{\ell \times d}$.

Implement multi-headed scaled dot-product attention in the `MultiHeadAttention` class in the file `cs682/transformer_layers.py`. The code below will check your implementation. The relative error should be less than `e-3`.

[8]:
```
torch.manual_seed(231)

# Choose dimensions such that they are all unique for easier debugging:
# Specifically, the following values correspond to N=1, H=2, T=3, E//H=4, and E=8.
batch_size = 1
sequence_length = 3
embed_dim = 8
attn = MultiHeadAttention(embed_dim, num_heads=2)

# Self-attention.
data = torch.randn(batch_size, sequence_length, embed_dim)
self_attn_output = attn(query=data, key=data, value=data)

# Masked self-attention.
mask = torch.randn(sequence_length, sequence_length) < 0.5
masked_self_attn_output = attn(query=data, key=data, value=data, attn_mask=mask)

# Attention using two inputs.
other_data = torch.randn(batch_size, sequence_length, embed_dim)
attn_output = attn(query=data, key=other_data, value=other_data)

expected_self_attn_output = np.asarray([[
[-0.2494,  0.1396,  0.4323, -0.2411, -0.1547,  0.2329, -0.1936,
         -0.1444],
        [-0.1997,  0.1746,  0.7377, -0.3549, -0.2657,  0.2693, -0.2541,
         -0.2476],
        [-0.0625,  0.1503,  0.7572, -0.3974, -0.1681,  0.2168, -0.2478,
         -0.3038]]])
```

```
expected_masked_self_attn_output = np.asarray([[
[-0.1347,  0.1934,  0.8628, -0.4903, -0.2614,  0.2798, -0.2586,
         -0.3019],
        [-0.1013,  0.3111,  0.5783, -0.3248, -0.3842,  0.1482, -0.3628,
         -0.1496],
        [-0.2071,  0.1669,  0.7097, -0.3152, -0.3136,  0.2520, -0.2774,
         -0.2208]]])

expected_attn_output = np.asarray([[
[-0.1980,  0.4083,  0.1968, -0.3477,  0.0321,  0.4258, -0.8972,
         -0.2744],
        [-0.1603,  0.4155,  0.2295, -0.3485, -0.0341,  0.3929, -0.8248,
         -0.2767],
        [-0.0908,  0.4113,  0.3017, -0.3539, -0.1020,  0.3784, -0.7189,
         -0.2912]]])

print('self_attn_output error: ', rel_error(expected_self_attn_output,␣
 ↪self_attn_output.detach().numpy()))
print('masked_self_attn_output error: ',␣
 ↪rel_error(expected_masked_self_attn_output, masked_self_attn_output.detach().
 ↪numpy()))
print('attn_output error: ', rel_error(expected_attn_output, attn_output.
 ↪detach().numpy()))
```

```
self_attn_output error:  0.0003775124598178026
masked_self_attn_output error:  0.0001526367643724865
attn_output error:  0.0003527921483788199
```

5 Positional Encoding

While transformers are able to easily attend to any part of their input, the attention mechanism has no concept of token order. However, for many tasks (especially natural language processing), relative token order is very important. To recover this, the authors add a positional encoding to the embeddings of individual word tokens.

Let us define a matrix $P \in \mathbb{R}^{l \times d}$, where $P_{ij} = $

$$\begin{cases} \sin\left(i \cdot 10000^{-\frac{j}{d}}\right) & \text{if j is even} \\ \cos\left(i \cdot 10000^{-\frac{(j-1)}{d}}\right) & \text{otherwise} \end{cases}$$

Rather than directly passing an input $X \in \mathbb{R}^{l \times d}$ to our network, we instead pass $X + P$.

Implement this layer in `PositionalEncoding` in `cs682/transformer_layers.py`. Once you are done, run the following to perform a simple test of your implementation. You should see errors on the order of `e-3` or less.

[10]:
```
torch.manual_seed(231)

batch_size = 1
sequence_length = 2
embed_dim = 6
data = torch.randn(batch_size, sequence_length, embed_dim)

pos_encoder = PositionalEncoding(embed_dim)
output = pos_encoder(data)

expected_pe_output = np.asarray([[[-1.2340,  1.1127,  1.6978, -0.0865, -0.0000,␣
 ↪ 1.2728],
                                  [ 0.9028, -0.4781,  0.5535,  0.8133,  1.2644,␣
 ↪ 1.7034]]])

print('pe_output error: ', rel_error(expected_pe_output, output.detach().
 ↪numpy()))
```

```
pe_output error:  0.00010421011374914356
```

6 Inline Question 1

Several key design decisions were made in designing the scaled dot product attention we introduced above. Explain why the following choices were beneficial: 1. Using multiple attention heads as opposed to one. 2. Dividing by $\sqrt{d/h}$ before applying the softmax function. Recall that d is the feature dimension and h is the number of heads. 3. Adding a linear transformation to the output of the attention operation.

Only one or two sentences per choice is necessary, but be sure to be specific in addressing what would have happened without each given implementation detail, why such a situation would be suboptimal, and how the proposed implementation improves the situation.

Your Answer: 1. Each attention head looks at the data in a unique way helping us capture more diverse patterns. It's like having different experts focusing on different aspects, and combining their insights through concatenation improves overall performance, just like a team of specialists working together. 2. Scaling: When we have lots of information (higher dimension d), the dot product in the equation can become too big, causing issues in the training and making it slow, and causing the softmax to suffer from saturation and causing vanishing gradients. Rescaling with this factor wrt dimension size help us avoid this issue. 3. After attention, we concatenate the results from different heads. Adding a layer here helps the model in learning how much weight to give to each head's perspective. Without this, averaging the attention scores wouldn't capture the unique contributions of each head, making it less effective in utilizing diverse viewpoints.

7 Transformer for Image Captioning

Now that you have implemented the previous layers, you can combine them to build a Transformer-based image captioning model. Open the file `cs682/classifiers/transformer.py` and look at the `CaptioningTransformer` class.

Implement the `forward` function of the class. After doing so, run the following to check your forward pass using a small test case; you should see error on the order of `e-5` or less.

[11]:
```
torch.manual_seed(231)
np.random.seed(231)

N, D, W = 4, 20, 30
word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
V = len(word_to_idx)
T = 3

transformer = CaptioningTransformer(
    word_to_idx,
    input_dim=D,
    wordvec_dim=W,
    num_heads=2,
    num_layers=2,
    max_length=30
)

# Set all model parameters to fixed values
for p in transformer.parameters():
    p.data = torch.tensor(np.linspace(-1.4, 1.3, num=p.numel()).reshape(*p.
 ↪shape))

features = torch.tensor(np.linspace(-1.5, 0.3, num=(N * D)).reshape(N, D))
captions = torch.tensor((np.arange(N * T) % V).reshape(N, T))

scores = transformer(features, captions)
expected_scores = np.asarray([[[-16.9532,   4.8261,  26.6054],
         [-17.1033,   4.6906,  26.4844],
         [-15.0708,   4.1108,  23.2924]],
        [[-17.1767,   4.5897,  26.3562],
         [-15.6017,   4.8693,  25.3403],
         [-15.1028,   4.6905,  24.4839]],
        [[-17.2172,   4.7701,  26.7574],
         [-16.6755,   4.8500,  26.3754],
         [-17.2172,   4.7701,  26.7574]],
        [[-16.3669,   4.1602,  24.6872],
         [-16.7897,   4.3467,  25.4831],
         [-17.0103,   4.7775,  26.5652]]])
print('scores error: ', rel_error(expected_scores, scores.detach().numpy()))
```

```
scores error:  5.056720614439509e-06
```

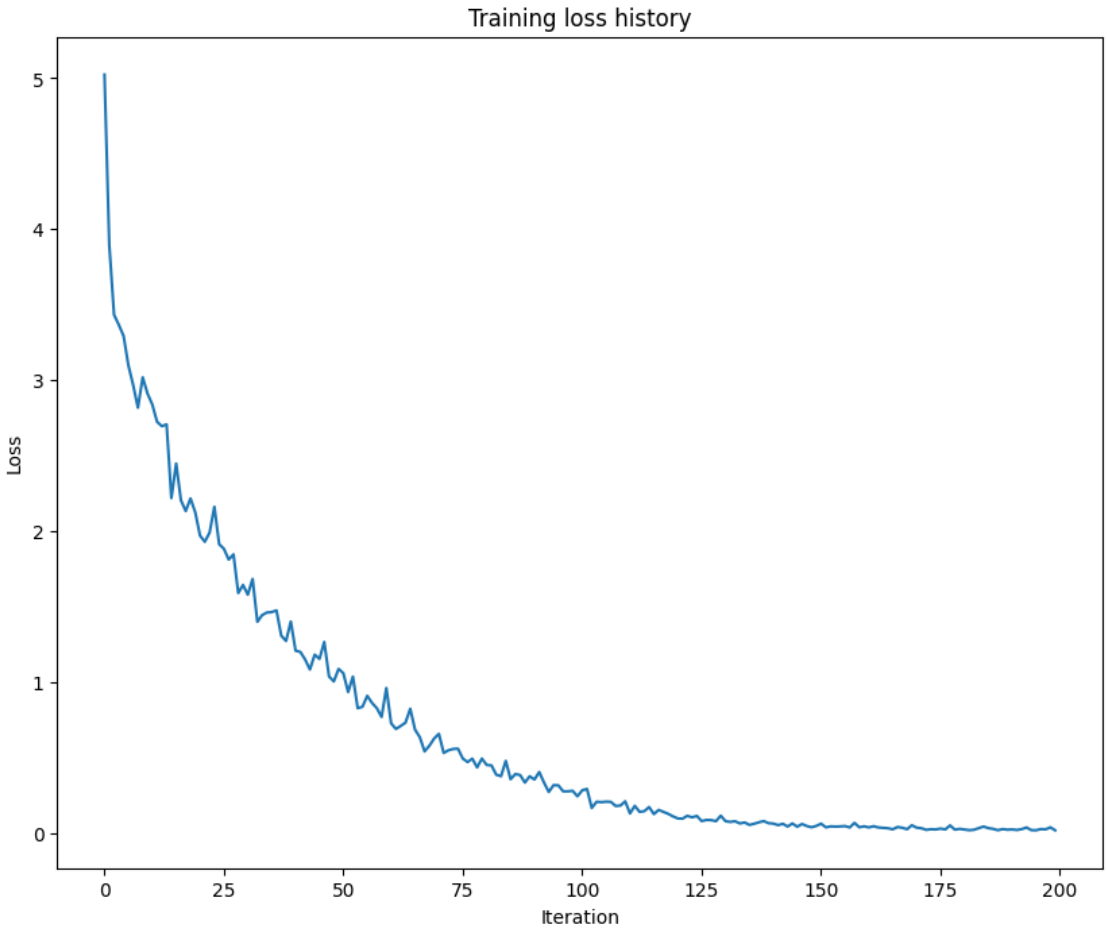
8 Overfit Transformer Captioning Model on Small Data

Run the following to overfit the Transformer-based captioning model on the same small dataset as we used for the RNN previously.

[12]:
```
torch.manual_seed(231)
np.random.seed(231)

data = load_coco_data(max_train=50)

transformer = CaptioningTransformer(
          word_to_idx=data['word_to_idx'],
          input_dim=data['train_features'].shape[1],
          wordvec_dim=256,
          num_heads=2,
          num_layers=2,
          max_length=30
        )


transformer_solver = CaptioningSolverTransformer(transformer, data,␣
 ↪idx_to_word=data['idx_to_word'],
           num_epochs=100,
           batch_size=25,
           learning_rate=0.001,
           verbose=True, print_every=10,
         )

transformer_solver.train()

# Plot the training losses.
plt.plot(transformer_solver.loss_history)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Training loss history')
plt.show()
```

```
base dir
/content/drive/MyDrive/cs682/assignment3/cs682/datasets/coco_captioning
(Iteration 1 / 200) loss: 5.023862
(Iteration 11 / 200) loss: 2.838965
(Iteration 21 / 200) loss: 1.969423
(Iteration 31 / 200) loss: 1.578652
(Iteration 41 / 200) loss: 1.207569
(Iteration 51 / 200) loss: 1.058506
(Iteration 61 / 200) loss: 0.726672
(Iteration 71 / 200) loss: 0.657085
(Iteration 81 / 200) loss: 0.450900
```

```
(Iteration 91 / 200) loss: 0.354287
(Iteration 101 / 200) loss: 0.282963
(Iteration 111 / 200) loss: 0.129757
(Iteration 121 / 200) loss: 0.096503
(Iteration 131 / 200) loss: 0.078078
(Iteration 141 / 200) loss: 0.062510
(Iteration 151 / 200) loss: 0.061801
(Iteration 161 / 200) loss: 0.037236
(Iteration 171 / 200) loss: 0.035470
(Iteration 181 / 200) loss: 0.023350
(Iteration 191 / 200) loss: 0.023562
```



Print final training loss. You should see a final loss of less than 0.03.

[13]:
```
print('Final loss: ', transformer_solver.loss_history[-1])
```

```
Final loss:  0.017880885
```

9 Transformer Sampling at Test Time

The sampling code has been written for you. You can simply run the following to compare with the previous results with the RNN. As before the training results should be much better than the validation set results, given how little data we trained on.

[14]:
```
# If you get an error, the URL just no longer exists, so don't worry!
# You can re-sample as many times as you want.
for split in ['train', 'val']:
    minibatch = sample_coco_minibatch(data, split=split, batch_size=2)
    gt_captions, features, urls = minibatch
    gt_captions = decode_captions(gt_captions, data['idx_to_word'])

    sample_captions = transformer.sample(features, max_length=30)
    sample_captions = decode_captions(sample_captions, data['idx_to_word'])

    for gt_caption, sample_caption, url in zip(gt_captions, sample_captions,␣
 ↪urls):
        img = image_from_url(url)
        # Skip missing URLs.
        if img is None: continue
        plt.imshow(img)
        plt.title('%s\n%s\nGT:%s' % (split, sample_caption, gt_caption))
        plt.axis('off')
        plt.show()
```

```
URL Error:  Gone http://farm1.staticflickr.com/202/487987371_489a65d670_z.jpg
```

train
a <UNK> decorated living room with a big tv in it <END>
GT:<START> a <UNK> decorated living room with a big tv in it <END>

val
a open refrigerator with a stuffed and red pizza <END>
GT:<START> a bedroom with a bed desk and <UNK> <UNK> <END>

val
a man is with a bite in his food in a hand <END>
GT:<START> a group of people <UNK> outside by a wall <END>

[ ]:

Generative_Adversarial_Networks

December 3, 2023

[4]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs682/assignments/assignment3/'
FOLDERNAME = 'cs682/assignment3'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the COCO dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs682/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs682/assignment3/cs682/datasets
/content/drive/My Drive/cs682/assignment3
```

[5]:
```
!bash collectSubmission.sh
```

```
collectSubmission.sh: line 27: C_R: unbound variable
```

0.1 Using GPU

Go to `Runtime > Change runtime type` and set `Hardware accelerator` to `GPU`. This will reset Colab. **Rerun the top cell to mount your Drive again.**

1 Generative Adversarial Networks (GANs)

So far in CS 231N, all the applications of neural networks that we have explored have been **discriminative models** that take an input and are trained to produce a labeled output. This has

ranged from straightforward classification of image categories to sentence generation (which was still phrased as a classification problem, our labels were in vocabulary space and we had learned a recurrence to capture multi-word labels). In this notebook, we will expand our repetoire, and build **generative models** using neural networks. Specifically, we will learn how to build models which generate novel images that resemble a set of training images.

1.0.1 What is a GAN?

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

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

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

To optimize this minimax game, we will alternate between taking gradient *descent* steps on the objective for G and gradient *ascent* steps on the objective for D: 1. update the **generator** (G) to minimize the probability of the **discriminator making the correct choice**. 2. update the **discriminator** (D) to maximize the probability of the **discriminator making the correct choice**.

While these updates are useful for analysis, they do not perform well in practice. Instead, we will use a different objective when we update the generator: maximize the probability of the **discriminator making the incorrect choice**. This small change helps to allevaiate problems with the generator gradient vanishing when the discriminator is confident. This is the standard update used in most GAN papers and was used in the original paper from Goodfellow et al..

In this assignment, we will alternate the following updates: 1. Update the generator (G) to maximize the probability of the discriminator making the incorrect choice on generated data:

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

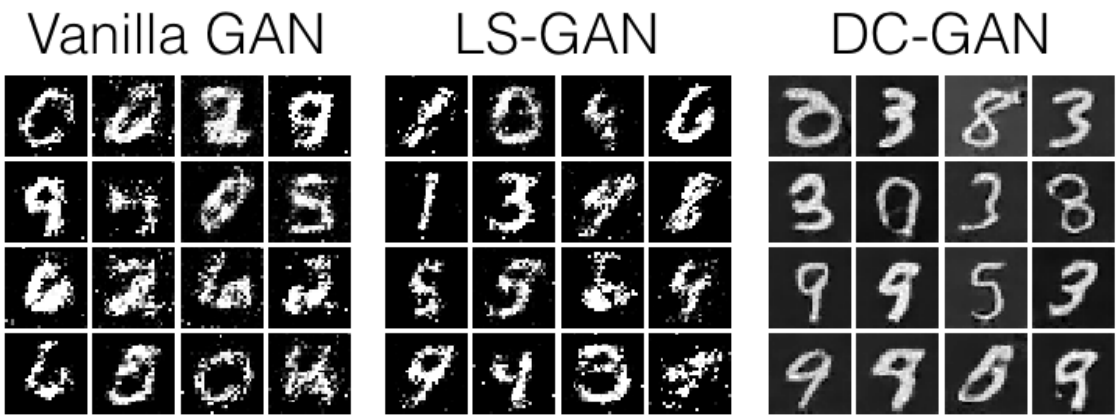
2. Update the discriminator (D), to maximize the probability of the discriminator making the correct choice on real and generated data:

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

Here's an example of what your outputs from the 3 different models you're going to train should look like. Note that GANs are sometimes finicky, so your outputs might not look exactly like this. This is just meant to be a *rough* guideline of the kind of quality you can expect:

[ ]:
```
# Run this cell to see sample outputs.
from IPython.display import Image
Image('images/gan_outputs_pytorch.png')
```

[ ]:



[ ]:
```
# Setup cell.
import numpy as np
import torch
import torch.nn as nn
from torch.nn import init
import torchvision
import torchvision.transforms as T
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler
import torchvision.datasets as dset
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from cs682.gan_pytorch import preprocess_img, deprocess_img, rel_error,␣
 ↪count_params, ChunkSampler

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

%load_ext autoreload
%autoreload 2

def show_images(images):
    images = np.reshape(images, [images.shape[0], -1]) # Images reshape to␣
 ↪(batch_size, D).
    sqrtn = int(np.ceil(np.sqrt(images.shape[0])))
```

```
    sqrtimg = int(np.ceil(np.sqrt(images.shape[1])))

    fig = plt.figure(figsize=(sqrtn, sqrtn))
    gs = gridspec.GridSpec(sqrtn, sqrtn)
    gs.update(wspace=0.05, hspace=0.05)

    for i, img in enumerate(images):
        ax = plt.subplot(gs[i])
        plt.axis('off')
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.set_aspect('equal')
        plt.imshow(img.reshape([sqrtimg,sqrtimg]))
    return

answers = dict(np.load('gan-checks.npz'))
dtype = torch.cuda.FloatTensor if torch.cuda.is_available() else torch.
 ↪FloatTensor
```

1.1 Dataset

GANs 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.

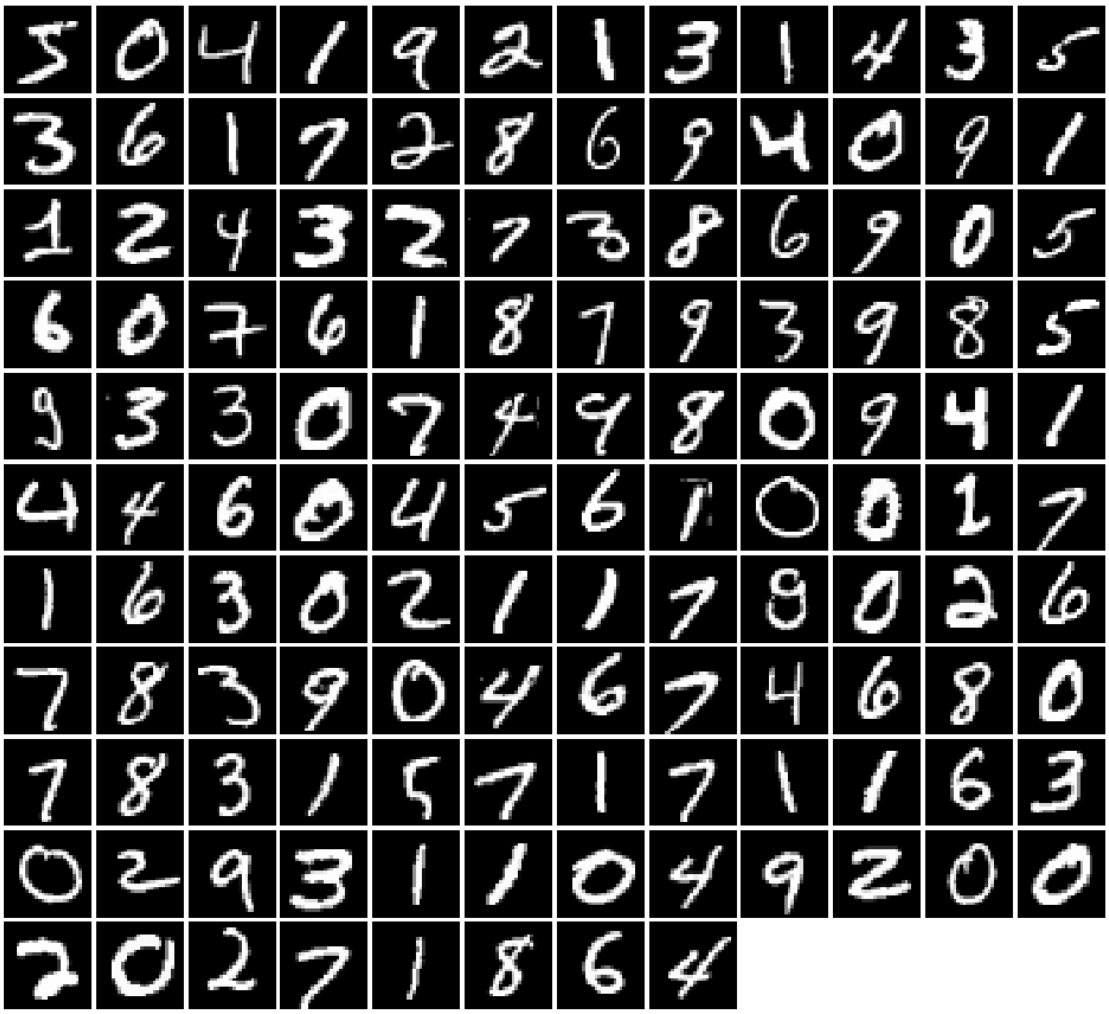
To simplify our code here, we will use the PyTorch MNIST wrapper, which downloads and loads the MNIST dataset. See the documentation for more information about the interface. The default parameters will take 5,000 of the training examples and place them into a validation dataset. The data will be saved into a folder called `MNIST_data`.

[ ]:
```
NUM_TRAIN = 50000
NUM_VAL = 5000

NOISE_DIM = 96
batch_size = 128

mnist_train = dset.MNIST(
    './cs682/datasets/MNIST_data',
    train=True,
    download=True,
    transform=T.ToTensor()
)
loader_train = DataLoader(
    mnist_train,
    batch_size=batch_size,
    sampler=ChunkSampler(NUM_TRAIN, 0)
```

```
)

mnist_val = dset.MNIST(
    './cs682/datasets/MNIST_data',
    train=True,
    download=True,
    transform=T.ToTensor()
)
loader_val = DataLoader(
    mnist_val,
    batch_size=batch_size,
    sampler=ChunkSampler(NUM_VAL, NUM_TRAIN)
)

iterator = iter(loader_train)
imgs, labels = next(iterator)
imgs = imgs.view(batch_size, 784).numpy().squeeze()
show_images(imgs)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Using downloaded and verified file: ./cs682/datasets/MNIST_data/MNIST/raw/train-
images-idx3-ubyte.gz
Extracting ./cs682/datasets/MNIST_data/MNIST/raw/train-images-idx3-ubyte.gz to
./cs682/datasets/MNIST_data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
./cs682/datasets/MNIST_data/MNIST/raw/train-labels-idx1-ubyte.gz

100%|      | 28881/28881 [00:00<00:00, 186937799.11it/s]

Extracting ./cs682/datasets/MNIST_data/MNIST/raw/train-labels-idx1-ubyte.gz to
./cs682/datasets/MNIST_data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to
./cs682/datasets/MNIST_data/MNIST/raw/t10k-images-idx3-ubyte.gz

100%|      | 1648877/1648877 [00:00<00:00, 32839458.29it/s]

Extracting ./cs682/datasets/MNIST_data/MNIST/raw/t10k-images-idx3-ubyte.gz to
./cs682/datasets/MNIST_data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to
./cs682/datasets/MNIST_data/MNIST/raw/t10k-labels-idx1-ubyte.gz

100%|      | 4542/4542 [00:00<00:00, 8113513.10it/s]

Extracting ./cs682/datasets/MNIST_data/MNIST/raw/t10k-labels-idx1-ubyte.gz to
```

./cs682/datasets/MNIST_data/MNIST/raw



1.2 Random Noise

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

Implement `sample_noise` in `cs682/gan_pytorch.py`.

Hint: use `torch.rand`.

Make sure noise is the correct shape and type:

[ ]:
```
from cs682.gan_pytorch import sample_noise

def test_sample_noise():
    batch_size = 3
    dim = 4
```

```
    torch.manual_seed(231)
    z = sample_noise(batch_size, dim)
    np_z = z.cpu().numpy()
    assert np_z.shape == (batch_size, dim)
    assert torch.is_tensor(z)
    assert np.all(np_z >= -1.0) and np.all(np_z <= 1.0)
    assert np.any(np_z < 0.0) and np.any(np_z > 0.0)
    print('All tests passed!')

test_sample_noise()
```

```
All tests passed!
```

1.3 Flatten

Recall our Flatten operation from previous notebooks… this time we also provide an Unflatten, which you might want to use when implementing the convolutional generator. We also provide a weight initializer (and call it for you) that uses Xavier initialization instead of PyTorch's uniform default.

[ ]:
```
from cs682.gan_pytorch import Flatten, Unflatten, initialize_weights
```

2 Discriminator

Our first step is to build a discriminator. Fill in the architecture as part of the `nn.Sequential` constructor in the function below. 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 $\alpha = 0.01$.

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

Implement `discriminator` in `cs682/gan_pytorch.py`

Test to make sure the number of parameters in the discriminator is correct:

[ ]:
```
from cs682.gan_pytorch import discriminator

def test_discriminator(true_count=267009):
    model = discriminator()
    cur_count = count_params(model)
    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()
```

```
Correct number of parameters in discriminator.
```

3 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 `cs682/gan_pytorch.py`

Test to make sure the number of parameters in the generator is correct:

[ ]:
```
from cs682.gan_pytorch 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()
```

```
Correct number of parameters in generator.
```

4 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.

HINTS: You should use the `bce_loss` function defined below to compute the binary cross entropy loss which is needed to compute the log probability of the true label given the logits output from the discriminator. Given a score $s \in \mathbb{R}$ and a label $y \in \{0, 1\}$, the binary cross entropy loss is

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

A naive implementation of this formula can be numerically unstable, so we have provided a numerically stable implementation that relies on PyTorch's `nn.BCEWithLogitsLoss`.

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).type(dtype)`

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. This is taken care of in `bce_loss` which combines the loss by averaging.

Implement `discriminator_loss` and `generator_loss` in `cs682/gan_pytorch.py`

Test your generator and discriminator loss. You should see errors < 1e-7.

[ ]:
```
from cs682.gan_pytorch import bce_loss, discriminator_loss, generator_loss

def test_discriminator_loss(logits_real, logits_fake, d_loss_true):
    d_loss = discriminator_loss(torch.Tensor(logits_real).type(dtype),
                                torch.Tensor(logits_fake).type(dtype)).cpu().
 ↪numpy()
    print("Maximum error in d_loss: %g"%rel_error(d_loss_true, d_loss))
    # print(d_loss_true)
test_discriminator_loss(
    answers['logits_real'],
    answers['logits_fake'],
    answers['d_loss_true']
)
```

```
Maximum error in d_loss: 2.83811e-08
```

[ ]:
```
def test_generator_loss(logits_fake, g_loss_true):
    g_loss = generator_loss(torch.Tensor(logits_fake).type(dtype)).cpu().numpy()
    print("Maximum error in g_loss: %g"%rel_error(g_loss_true, g_loss))

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

```
Maximum error in g_loss: 4.4518e-09
```

5 Optimizing our Loss

Make a function that returns an `optim.Adam` optimizer for the given model with a 1e-3 learning rate, beta1=0.5, beta2=0.999. You'll use this to construct optimizers for the generators and discriminators for the rest of the notebook.

Implement `get_optimizer` in `cs682/gan_pytorch.py`

6 Training a GAN!

We provide you the main training loop. You won't need to change `run_a_gan` in `cs682/gan_pytorch.py`, but we encourage you to read through it for your own understanding.

[ ]:
```
from cs682.gan_pytorch import get_optimizer, run_a_gan

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

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

# Use the function you wrote earlier to get optimizers for the Discriminator␣
 ↪and the Generator
D_solver = get_optimizer(D)
G_solver = get_optimizer(G)

# Run it!
images = run_a_gan(
    D,
    G,
    D_solver,
    G_solver,
    discriminator_loss,
    generator_loss,
    loader_train
)
```

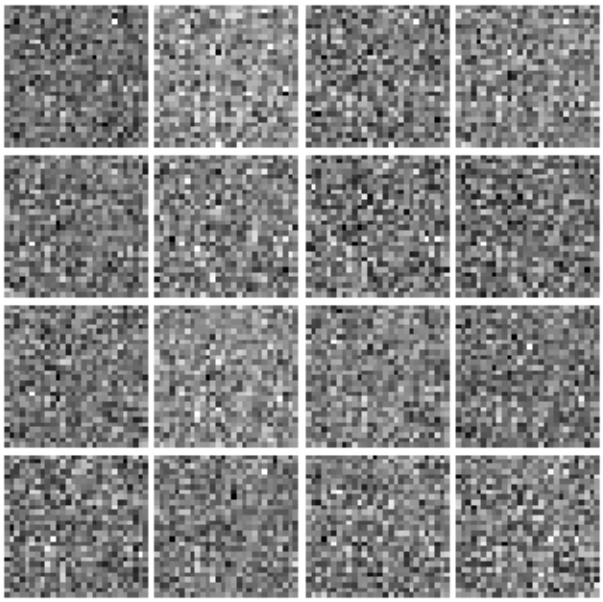
```
Iter: 0, D: 1.421, G:0.6958
Iter: 250, D: 1.325, G:0.9796
Iter: 500, D: 1.08, G:1.021
Iter: 750, D: 1.081, G:1.305
Iter: 1000, D: 1.3, G:0.8411
Iter: 1250, D: 1.227, G:1.001
Iter: 1500, D: 1.361, G:0.8695
Iter: 1750, D: 1.127, G:0.9388
Iter: 2000, D: 1.36, G:0.9755
Iter: 2250, D: 1.279, G:0.9393
Iter: 2500, D: 1.32, G:0.8744
```

```
Iter: 2750, D: 1.287, G:0.6918
Iter: 3000, D: 1.357, G:1.021
Iter: 3250, D: 1.361, G:0.813
Iter: 3500, D: 1.297, G:0.7552
Iter: 3750, D: 1.411, G:0.7174
```

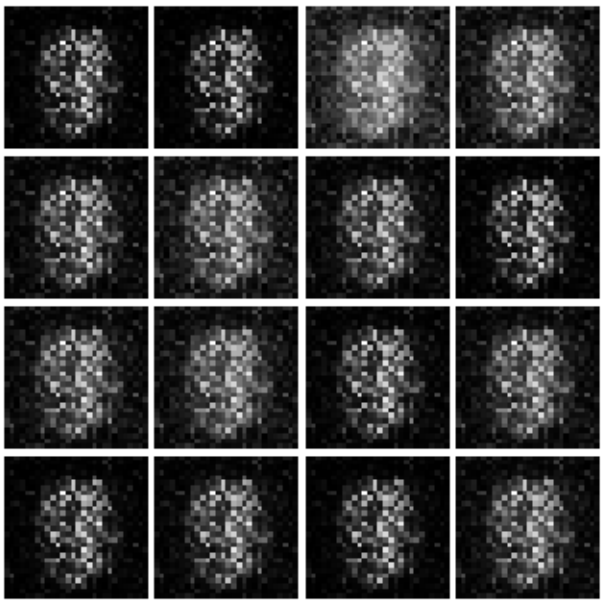
Run the cell below to show the generated images.

[ ]:
```
numIter = 0
for img in images:
    print("Iter: {}".format(numIter))
    show_images(img)
    plt.show()
    numIter += 250
    print()
```

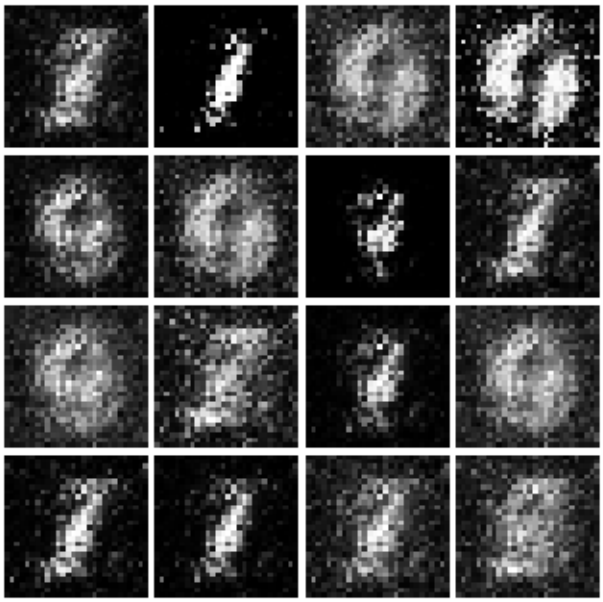
```
Iter: 0
```



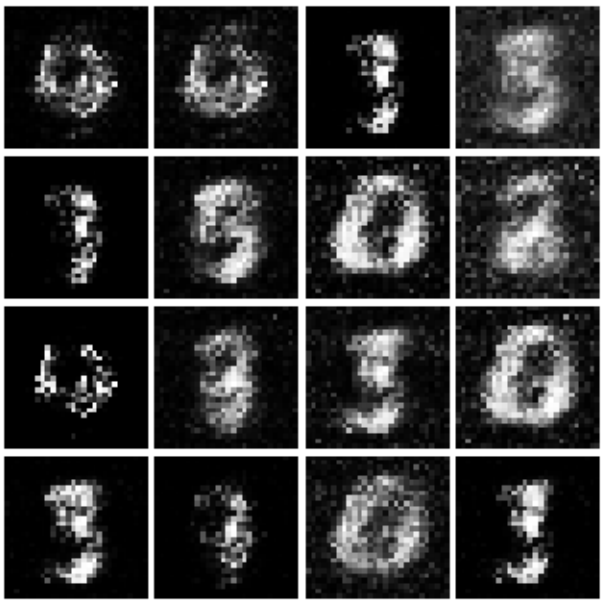
```
Iter: 250
```



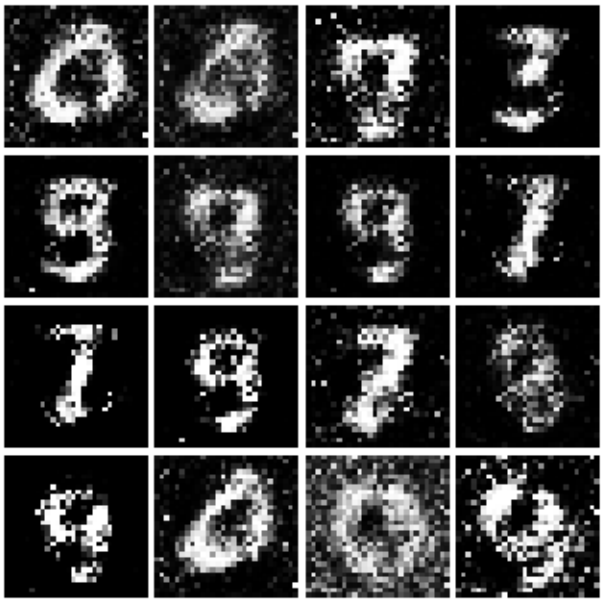
Iter: 500



Iter: 750

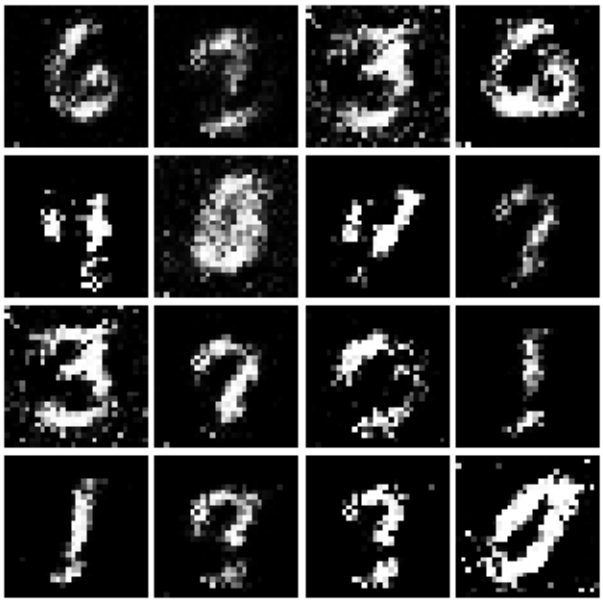


Iter: 1000

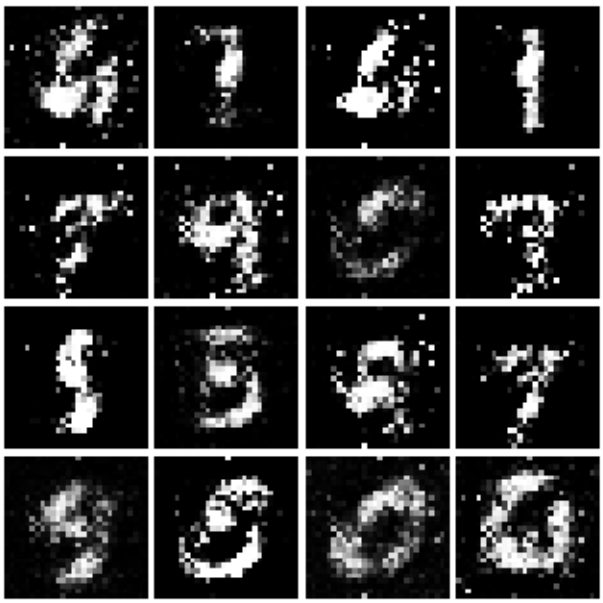


Iter: 1250

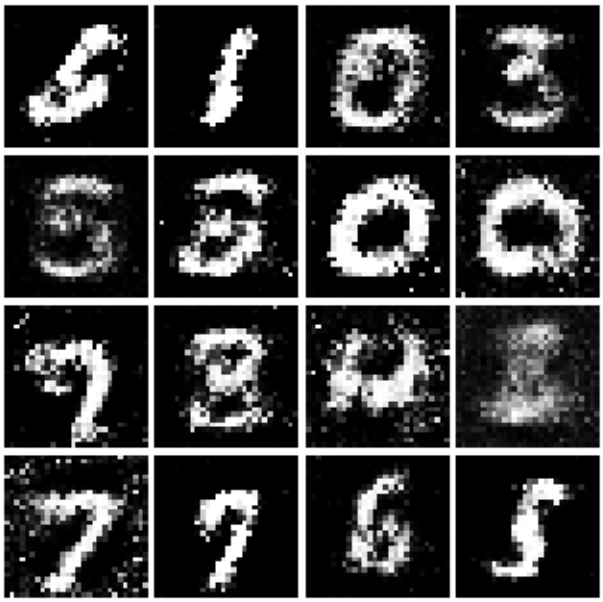
Iter: 1500



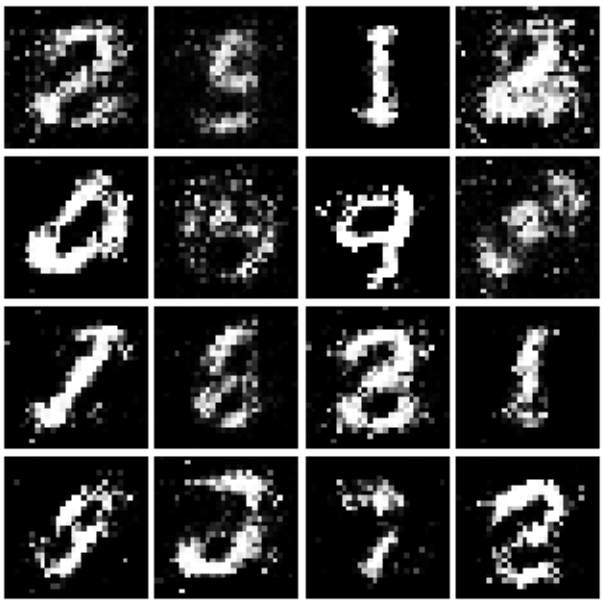
Iter: 1750



Iter: 2000



Iter: 2250



Iter: 2500

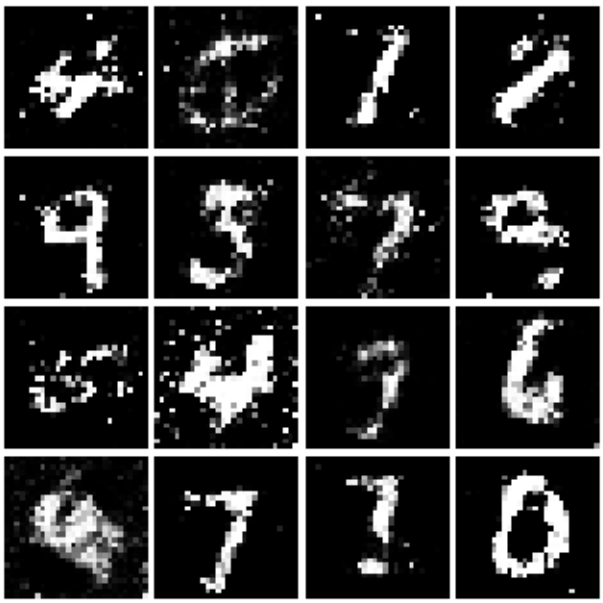
Iter: 2750

Iter: 3000

Iter: 3250

Iter: 3500

Iter: 3750

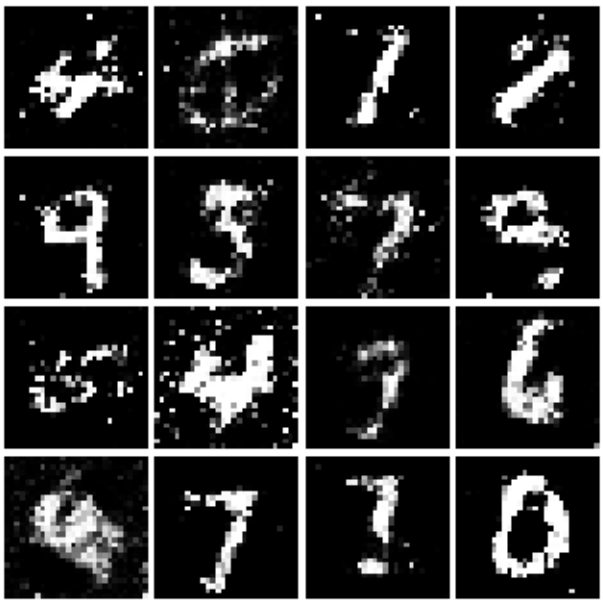


6.1 Inline Question 1

What does your final vanilla GAN image look like?

[ ]:
```
# This output is your answer.
print("Vanilla GAN final image:")
show_images(images[-1])
plt.show()
```

```
Vanilla GAN final image:
```



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.

7 Least Squares GAN

We'll now look at Least Squares GAN, a newer, more stable alernative to the original GAN loss function. For this part, all we have to do is change the loss function and retrain the model. We'll implement equation (9) in the paper, with the generator loss:

$$\ell_G = \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[(D(G(z))-1)^2\right]$$

and the discriminator loss:

$$\ell_D = \frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}}\left[(D(x)-1)^2\right] + \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[(D(G(z)))^2\right]$$

HINTS: Instead of computing the expectation, we will be averaging over elements of the minibatch, so make sure to combine the loss by averaging instead of summing. When plugging in for $D(x)$ and $D(G(z))$ use the direct output from the discriminator (`scores_real` and `scores_fake`).

Implement `ls_discriminator_loss`, `ls_generator_loss` in `cs682/gan_pytorch.py`

Before running a GAN with our new loss function, let's check it:

[ ]:
```
from cs682.gan_pytorch import ls_discriminator_loss, ls_generator_loss

def test_lsgan_loss(score_real, score_fake, d_loss_true, g_loss_true):
    score_real = torch.Tensor(score_real).type(dtype)
```

```
    score_fake = torch.Tensor(score_fake).type(dtype)
    d_loss = ls_discriminator_loss(score_real, score_fake).cpu().numpy()
    g_loss = ls_generator_loss(score_fake).cpu().numpy()
    print("Maximum error in d_loss: %g"%rel_error(d_loss_true, d_loss))
    print("Maximum error in g_loss: %g"%rel_error(g_loss_true, g_loss))

test_lsgan_loss(
    answers['logits_real'],
    answers['logits_fake'],
    answers['d_loss_lsgan_true'],
    answers['g_loss_lsgan_true']
)
```

```
Maximum error in d_loss: 1.53171e-08
Maximum error in g_loss: 2.7837e-09
```

Run the following cell to train your model!

[ ]:
```
D_LS = discriminator().type(dtype)
G_LS = generator().type(dtype)

D_LS_solver = get_optimizer(D_LS)
G_LS_solver = get_optimizer(G_LS)

images = run_a_gan(
    D_LS,
    G_LS,
    D_LS_solver,
    G_LS_solver,
    ls_discriminator_loss,
    ls_generator_loss,
    loader_train
)
```

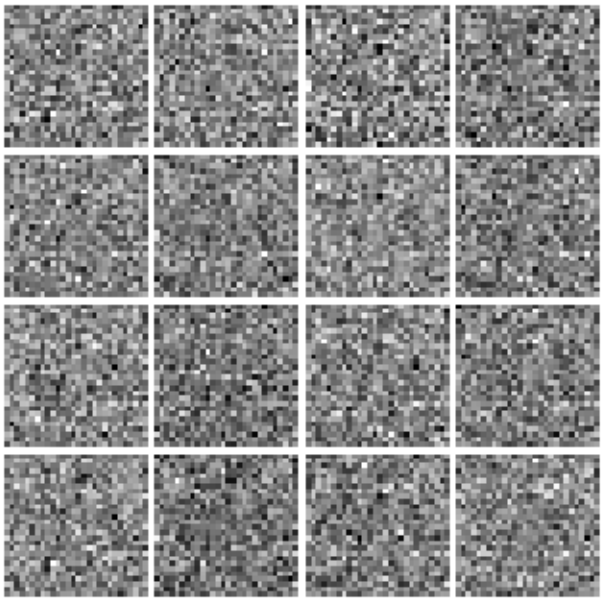
```
Iter: 0, D: 0.5124, G:0.4988
Iter: 250, D: 0.1808, G:0.3755
Iter: 500, D: 0.1216, G:0.4679
Iter: 750, D: 0.1431, G:0.3375
Iter: 1000, D: 0.1682, G:0.2342
Iter: 1250, D: 0.1817, G:0.2205
Iter: 1500, D: 0.1954, G:0.2406
Iter: 1750, D: 0.1912, G:0.1718
Iter: 2000, D: 0.2384, G:0.1717
Iter: 2250, D: 0.2297, G:0.1646
Iter: 2500, D: 0.2122, G:0.214
Iter: 2750, D: 0.2341, G:0.1425
Iter: 3000, D: 0.2183, G:0.1781
Iter: 3250, D: 0.2523, G:0.2211
```

```
Iter: 3500, D: 0.2132, G:0.1528
Iter: 3750, D: 0.2218, G:0.1924
```

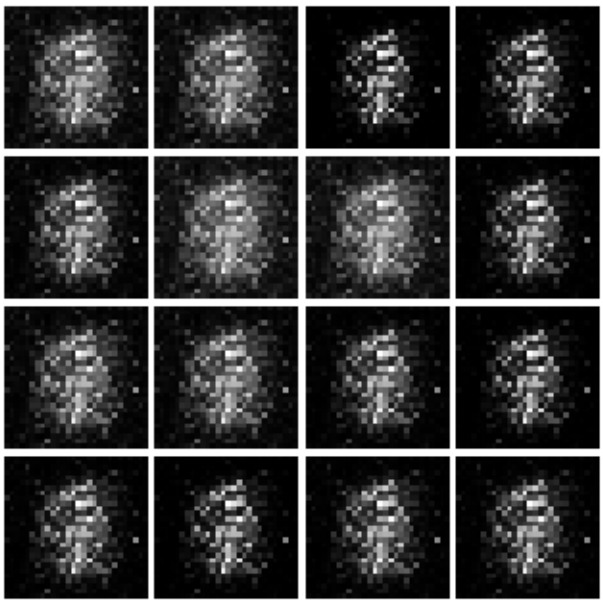
Run the cell below to show generated images.

[ ]:
```
numIter = 0
for img in images:
    print("Iter: {}".format(numIter))
    show_images(img)
    plt.show()
    numIter += 250
    print()
```

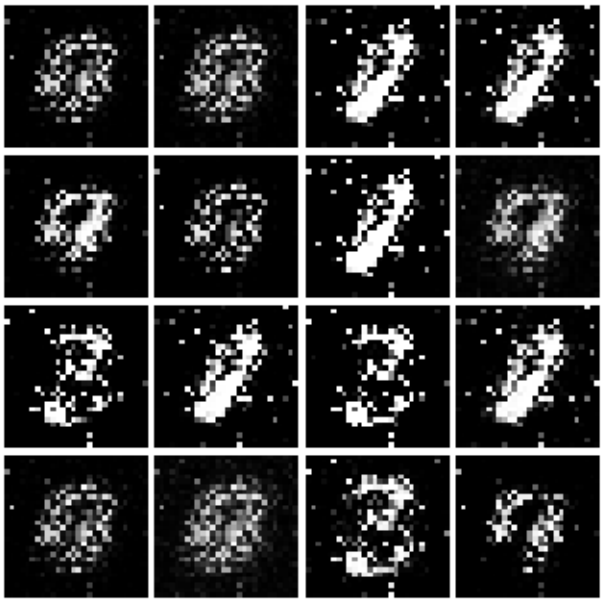
```
Iter: 0
```



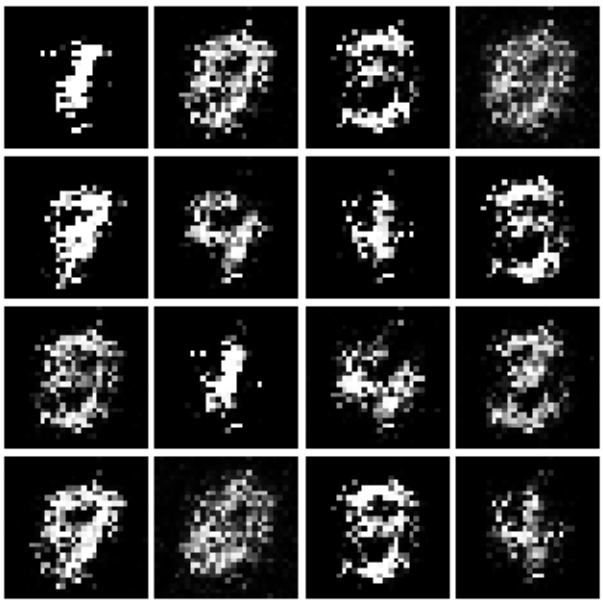
```
Iter: 250
```



Iter: 500

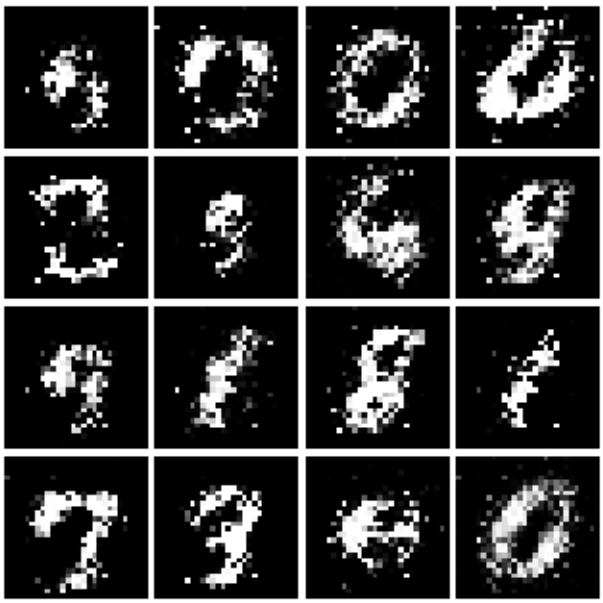


Iter: 750



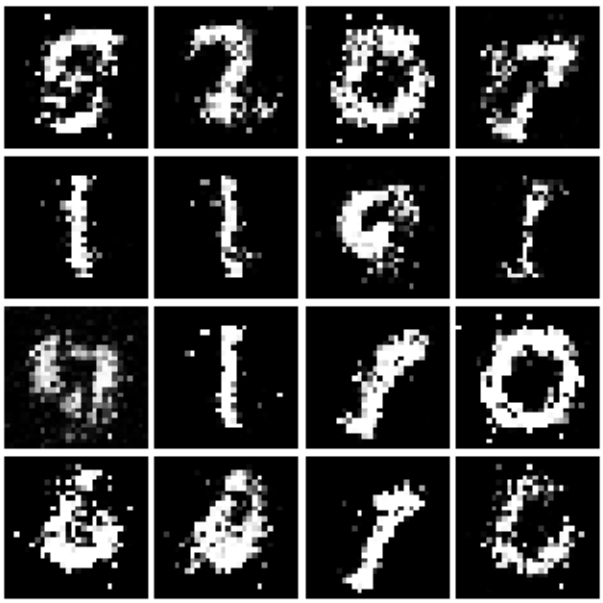
Iter: 1000

Iter: 1250

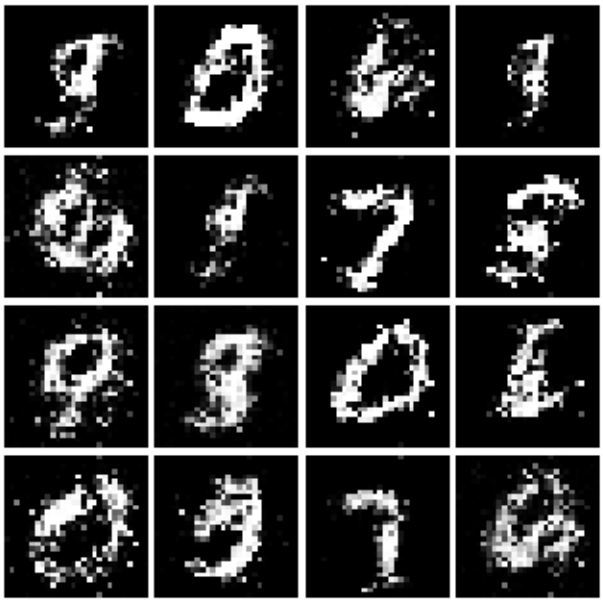


Iter: 1500

Iter: 1750



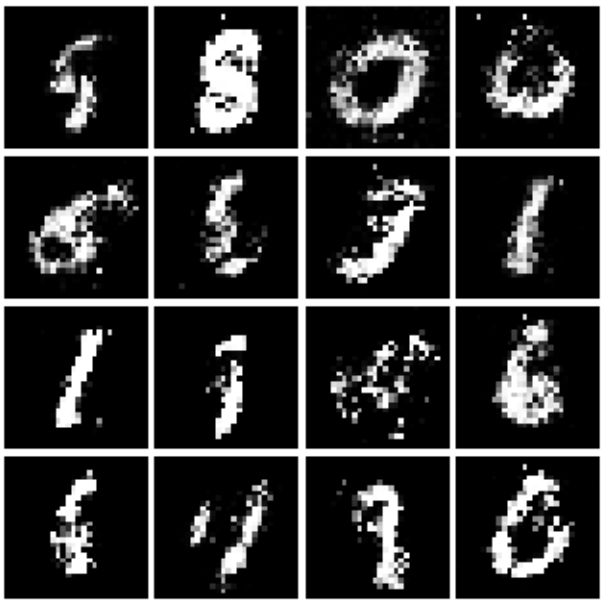
Iter: 2000



Iter: 2250

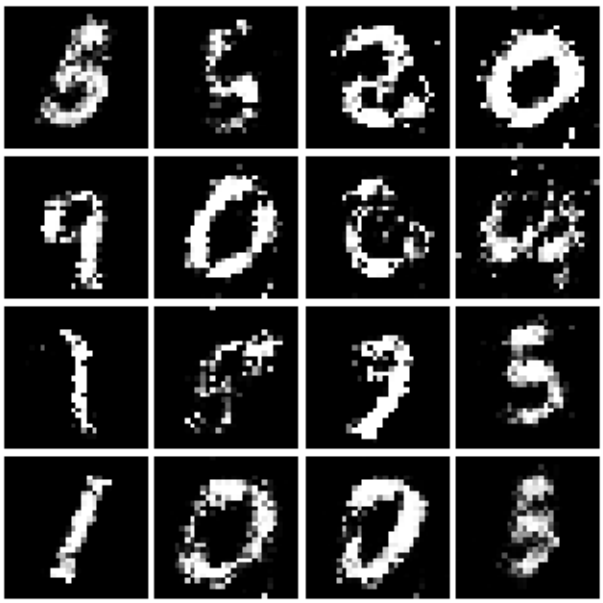
Iter: 2500

Iter: 2750

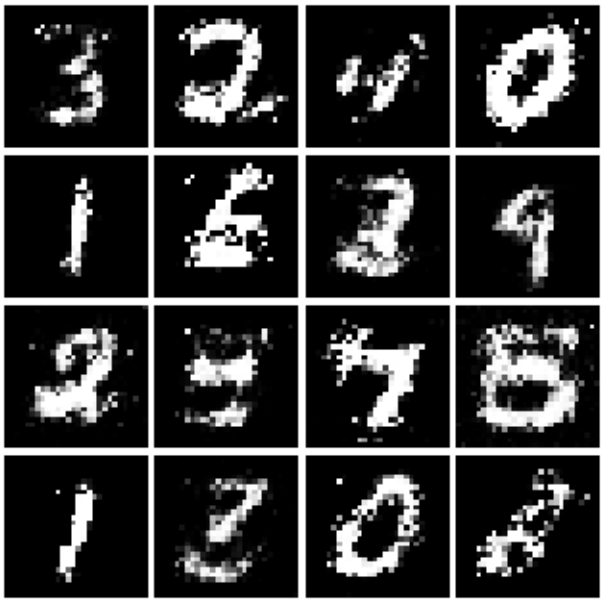


Iter: 3000

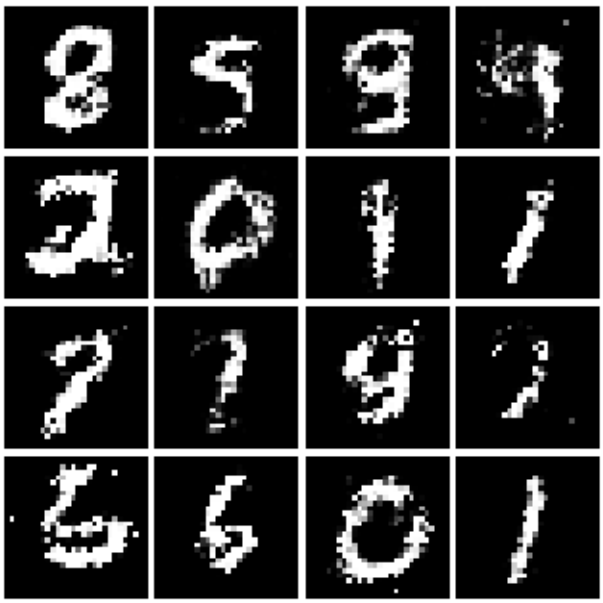
Iter: 3250



Iter: 3500



Iter: 3750

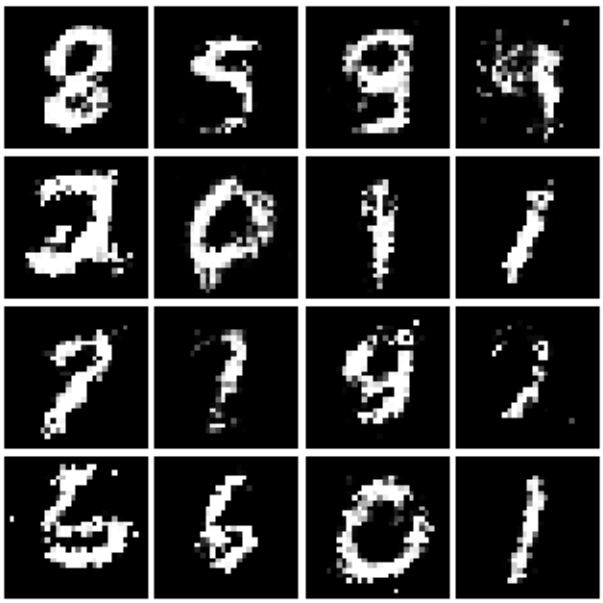


7.1 Inline Question 2

What does your final LSGAN image look like?

[ ]:
```
# This output is your answer.
print("LSGAN final image:")
show_images(images[-1])
plt.show()
```

```
LSGAN final image:
```



8 Deeply Convolutional GANs

In the first part of the notebook, 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. * 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

Implement `build_dc_classifier` in `cs682/gan_pytorch.py`

[ ]:
```
from cs682.gan_pytorch import build_dc_classifier

data = next(enumerate(loader_train))[-1][0].type(dtype)
b = build_dc_classifier(batch_size).type(dtype)
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(batch_size)
    cur_count = count_params(model)
    if cur_count != true_count:
        print('Incorrect number of parameters in classifier. Check your␣
 ↪achitecture.')
    else:
        print('Correct number of parameters in classifier.')

test_dc_classifer()
```

```
Correct number of parameters in classifier.
```

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 * Use `Unflatten()` to reshape into Image Tensor of shape 7, 7, 128 * ConvTranspose2d: 64 filters of 4x4, stride 2, 'same' padding (use `padding=1`) * `ReLU` * BatchNorm * ConvTranspose2d: 1 filter of 4x4, stride 2, 'same' padding (use `padding=1`) * `TanH` * Should have a 28x28x1 image, reshape back into 784 vector (using `Flatten()`)

Implement `build_dc_generator` in `cs682/gan_pytorch.py`

[ ]:
```
from cs682.gan_pytorch import build_dc_generator

test_g_gan = build_dc_generator().type(dtype)
test_g_gan.apply(initialize_weights)

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

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

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

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

```
Correct number of parameters in generator.
```

[ ]:
```
D_DC = build_dc_classifier(batch_size).type(dtype)
D_DC.apply(initialize_weights)
G_DC = build_dc_generator().type(dtype)
G_DC.apply(initialize_weights)

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

images = run_a_gan(
    D_DC,
    G_DC,
    D_DC_solver,
    G_DC_solver,
    discriminator_loss,
    generator_loss,
    loader_train,
    num_epochs=5
)
```

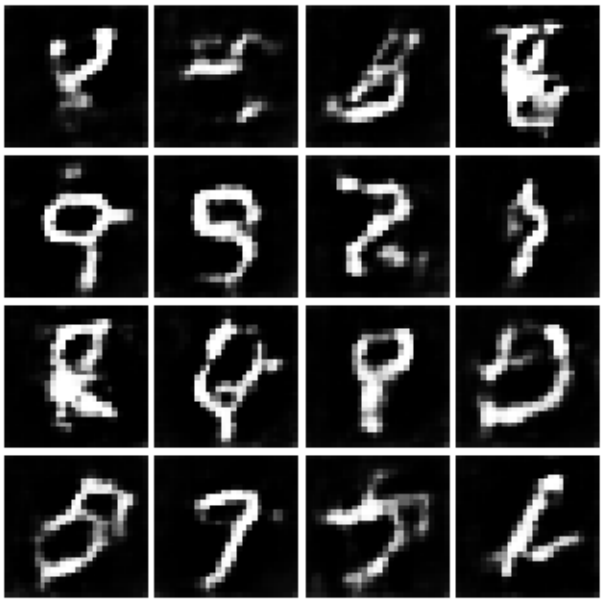
```
Iter: 0, D: 1.4, G:0.4541
Iter: 250, D: 1.348, G:1.028
Iter: 500, D: 1.083, G:1.27
Iter: 750, D: 1.218, G:1.254
Iter: 1000, D: 1.252, G:0.9445
Iter: 1250, D: 1.249, G:0.9214
Iter: 1500, D: 1.139, G:1.125
Iter: 1750, D: 1.123, G:0.9784
```

Run the cell below to show generated images.

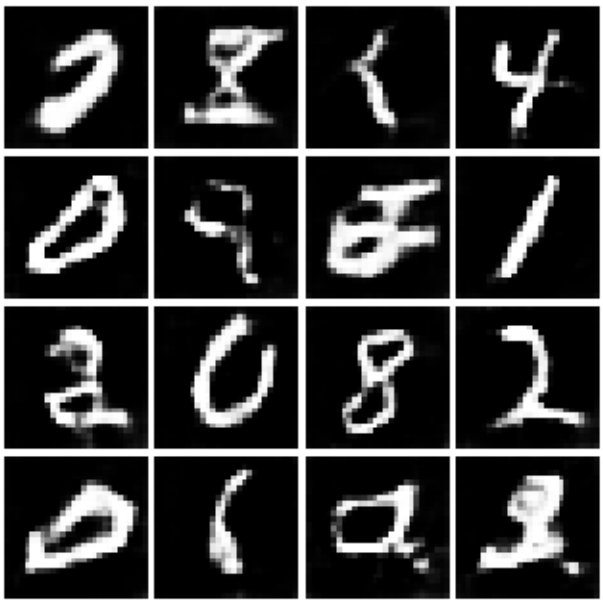
[ ]:
```
numIter = 0
for img in images:
    print("Iter: {}".format(numIter))
    show_images(img)
    plt.show()
    numIter += 250
    print()
```

```
Iter: 0
```

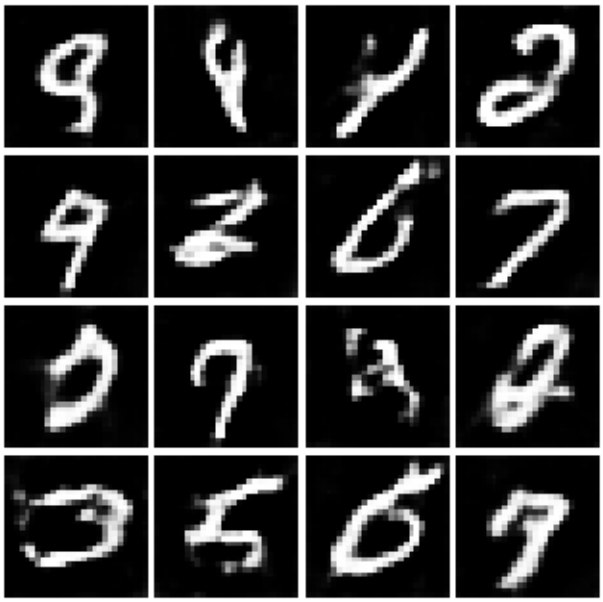
Iter: 250



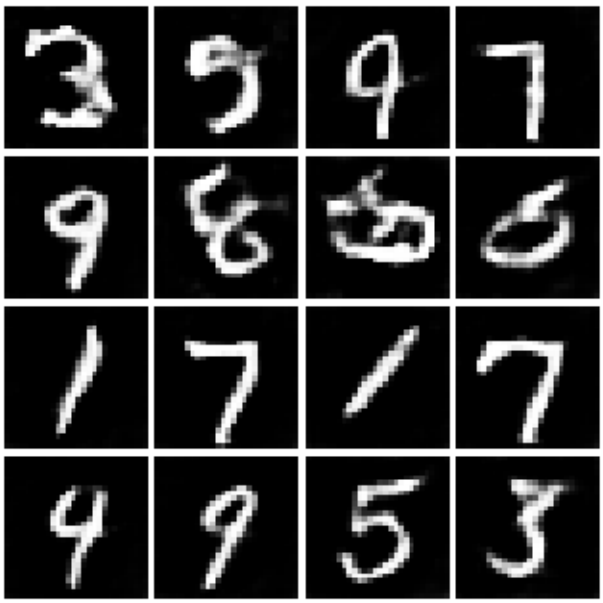
Iter: 500



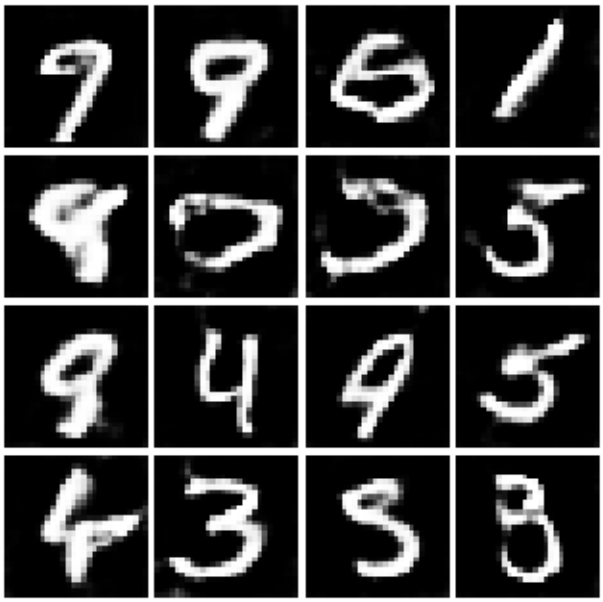
Iter: 750



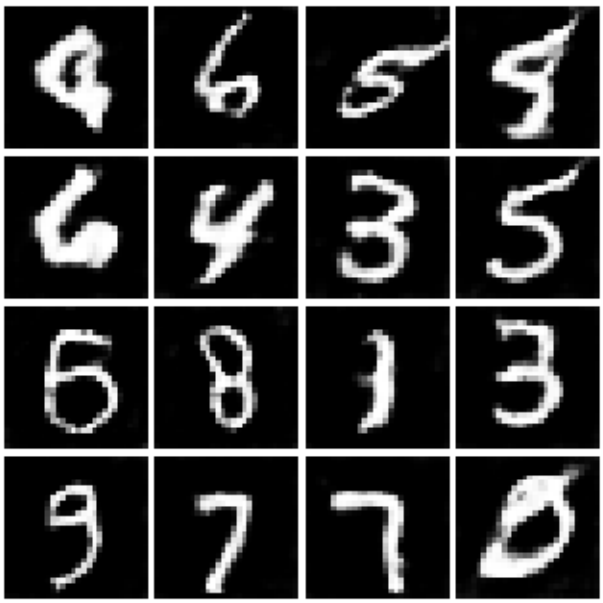
Iter: 1000



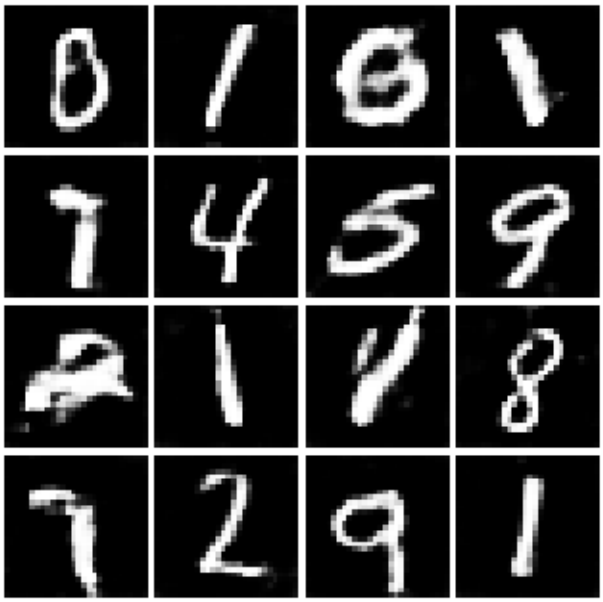
Iter: 1250



Iter: 1500



Iter: 1750

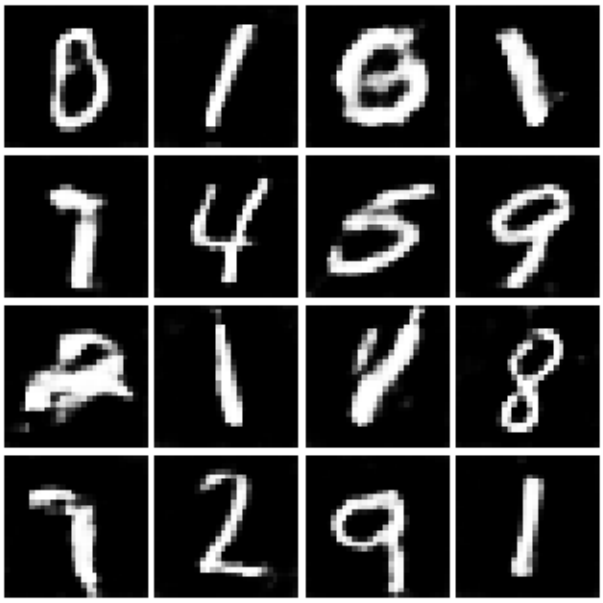


8.1 Inline Question 3

What does your final DCGAN image look like?

```
[ ]: # This output is your answer.
print("DCGAN final image:")
show_images(images[-1])
plt.show()
```

```
DCGAN final image:
```



8.2 Inline Question 4

We will look at an example to see why alternating minimization of the same objective (like in a GAN) can be tricky business.

Consider $f(x, y) = xy$. What does $\min_x \max_y f(x, y)$ evaluate to? (Hint: minmax tries to minimize the maximum value achievable.)

Now try to evaluate this function numerically for 6 steps, starting at the point $(1, 1)$, by using alternating gradient (first updating y, then updating x using that updated y) with step size 1. **Here step size is the learning_rate, and steps will be learning_rate * gradient.** You'll find that writing out the update step in terms of $x_t, y_t, x_{t+1}, y_{t+1}$ will be useful.

Breifly explain what $\min_x \max_y f(x, y)$ evaluates to and record the six pairs of explicit values for (x_t, y_t) in the table below.

8.2.1 Your answer:

y_0	y_1	y_2	y_3	y_4	y_5	y_6
1	2	1	-1	-2	-1	1
x_0	x_1	x_2	x_3	x_4	x_5	x_6
1	-1	-2	-1	1	2	1

8.3 Inline Question 5

Using this method, will we ever reach the optimal value? Why or why not?

8.3.1 Your answer:

No. We'll not reach the optimal value because we get stuck in a cycle of similar value (x, y) after a number of iterations (cyclic behavior). x, and y are thus oscillating, and thus resulting in an infinite loop.

8.4 Inline Question 6

If the generator loss decreases during training while the discriminator loss stays at a constant high value from the start, is this a good sign? Why or why not? A qualitative answer is sufficient.

8.4.1 Your answer:

It's not a good sign.

Discriminiator loss being high means that the discriminator hasn't learned the structure of the real data effectively, leading it to pass fake images as real ones. Since it stays at a constant high value, it doesn't get a chance to be good enough to provide a meaningful signal to train the generator.

In turn, generator will suffer. Generator loss decreasing means that it is being able to fool the discriminator easily, and as it generates noisy images, these can end up being classified as real ones by the discriminator due to its high loss. This results in drop in generator loss even though it hasn't learnt anything and generated images are nowhere close to real distribution.

[1]:

```
  File "<ipython-input-1-265d8e84bd5c>", line 1
    It is not a good sign.
                ^
SyntaxError: invalid syntax
```

StyleTransfer

December 3, 2023

[1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs682/assignments/assignment3/'
FOLDERNAME = 'cs682/assignment3'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the COCO dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs682/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs682/assignment3/cs682/datasets
/content/drive/My Drive/cs682/assignment3
```

0.1 Using GPU

Go to `Runtime > Change runtime type` and set `Hardware accelerator` to `GPU`. This will reset Colab. **Rerun the top cell to mount your Drive again.**

1 Style Transfer

In this notebook we will implement the style transfer technique from "Image Style Transfer Using Convolutional Neural Networks" (Gatys et al., CVPR 2015).

The general idea is to take two images, and produce a new image that reflects the content of one but the artistic "style" of the other. We will do this by first formulating a loss function that matches

the content and style of each respective image in the feature space of a deep network, and then performing gradient descent on the pixels of the image itself.

The deep network we use as a feature extractor is SqueezeNet, a small model that has been trained on ImageNet. You could use any network, but we chose SqueezeNet here for its small size and efficiency.

Here's an example of the images you'll be able to produce by the end of this notebook:

1.1 Setup

[37]:
```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as T
import PIL

import numpy as np

from imageio import imread
from collections import namedtuple
import matplotlib.pyplot as plt

from cs682.image_utils import SQUEEZENET_MEAN, SQUEEZENET_STD
%matplotlib inline
```

We provide you with some helper functions to deal with images, since for this part of the assignment we're dealing with real JPEGs, not CIFAR-10 data.

[38]:
```
def preprocess(img, size=512):
    transform = T.Compose([
        T.Resize(size),
        T.ToTensor(),
        T.Normalize(mean=SQUEEZENET_MEAN.tolist(),
                    std=SQUEEZENET_STD.tolist()),
        T.Lambda(lambda x: x[None]),
    ])
    return transform(img)
```

```
def deprocess(img):
    transform = T.Compose([
        T.Lambda(lambda x: x[0]),
        T.Normalize(mean=[0, 0, 0], std=[1.0 / s for s in SQUEEZENET_STD.
 ↪tolist()]),
        T.Normalize(mean=[-m for m in SQUEEZENET_MEAN.tolist()], std=[1, 1, 1]),
        T.Lambda(rescale),
        T.ToPILImage(),
    ])
    return transform(img)

def rescale(x):
    low, high = x.min(), x.max()
    x_rescaled = (x - low) / (high - low)
    return x_rescaled

def rel_error(x,y):
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))

def features_from_img(imgpath, imgsize):
    img = preprocess(PIL.Image.open(imgpath), size=imgsize)
    img_var = img.type(dtype)
    return extract_features(img_var, cnn), img_var

# Older versions of scipy.misc.imresize yield different results
# from newer versions, so we check to make sure scipy is up to date.
def check_scipy():
    import scipy
    vnum = int(scipy.__version__.split('.')[1])
    major_vnum = int(scipy.__version__.split('.')[0])

    assert vnum >= 16 or major_vnum >= 1, "You must install SciPy >= 0.16.0 to␣
 ↪complete this notebook."

check_scipy()

answers = dict(np.load('style-transfer-checks.npz'))
```

As in the last assignment, we need to set the dtype to select either the CPU or the GPU

[39]:
```
dtype = torch.FloatTensor
# Uncomment out the following line if you're on a machine with a GPU set up for␣
 ↪PyTorch!
# dtype = torch.cuda.FloatTensor
```

[40]:
```
# Load the pre-trained SqueezeNet model.
cnn = torchvision.models.squeezenet1_1(pretrained=True).features
```

```
cnn.type(dtype)

# We don't want to train the model any further, so we don't want PyTorch to␣
 ↪waste computation
# computing gradients on parameters we're never going to update.
for param in cnn.parameters():
    param.requires_grad = False

# We provide this helper code which takes an image, a model (cnn), and returns␣
 ↪a list of
# feature maps, one per layer.
def extract_features(x, cnn):
    """
    Use the CNN to extract features from the input image x.

    Inputs:
    - x: A PyTorch Tensor of shape (N, C, H, W) holding a minibatch of images␣
 ↪that
      will be fed to the CNN.
    - cnn: A PyTorch model that we will use to extract features.

    Returns:
    - features: A list of feature for the input images x extracted using the␣
 ↪cnn model.
      features[i] is a PyTorch Tensor of shape (N, C_i, H_i, W_i); recall that␣
 ↪features
      from different layers of the network may have different numbers of␣
 ↪channels (C_i) and
      spatial dimensions (H_i, W_i).
    """
    features = []
    prev_feat = x
    for i, module in enumerate(cnn._modules.values()):
        next_feat = module(prev_feat)
        features.append(next_feat)
        prev_feat = next_feat
    return features

#please disregard warnings about initialization
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208:
UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be
removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223:
UserWarning: Arguments other than a weight enum or `None` for 'weights' are
deprecated since 0.13 and may be removed in the future. The current behavior is
```

```
equivalent to passing `weights=SqueezeNet1_1_Weights.IMAGENET1K_V1`. You can
also use `weights=SqueezeNet1_1_Weights.DEFAULT` to get the most up-to-date
weights.
  warnings.warn(msg)
```

1.2 Computing Loss

We're going to compute the three components of our loss function now. The loss function is a weighted sum of three terms: content loss + style loss + total variation loss. You'll fill in the functions that compute these weighted terms below.

1.3 Content loss

We can generate an image that reflects the content of one image and the style of another by incorporating both in our loss function. We want to penalize deviations from the content of the content image and deviations from the style of the style image. We can then use this hybrid loss function to perform gradient descent **not on the parameters** of the model, but instead **on the pixel values** of our original image.

Let's first write the content loss function. Content loss measures how much the feature map of the generated image differs from the feature map of the source image. We only care about the content representation of one layer of the network (say, layer ℓ), that has feature maps $A^\ell \in \mathbb{R}^{1\times C_\ell \times H_\ell \times W_\ell}$. C_ℓ is the number of filters/channels in layer ℓ, H_ℓ and W_ℓ are the height and width. We will work with reshaped versions of these feature maps that combine all spatial positions into one dimension. Let $F^\ell \in \mathbb{R}^{C_\ell \times M_\ell}$ be the feature map for the current image and $P^\ell \in \mathbb{R}^{C_\ell \times M_\ell}$ be the feature map for the content source image where $M_\ell = H_\ell \times W_\ell$ is the number of elements in each feature map. Each row of F^ℓ or P^ℓ represents the vectorized activations of a particular filter, convolved over all positions of the image. Finally, let w_c be the weight of the content loss term in the loss function.

Then the content loss is given by:

$L_c = w_c \times \sum_{i,j}(F_{ij}^\ell - P_{ij}^\ell)^2$

[41]:
```
def content_loss(content_weight, content_current, content_original):
    """
    Compute the content loss for style transfer.

    Inputs:
    - content_weight: Scalar giving the weighting for the content loss.
    - content_current: features of the current image; this is a PyTorch Tensor␣
 ↪of shape
        (1, C_l, H_l, W_l).
    - content_target: features of the content image, Tensor with shape (1, C_l,␣
 ↪H_l, W_l).

    Returns:
    - scalar content loss
    """
```

```
    content_loss = content_weight*(torch.sum(torch.pow(content_current -␣
↪content_original, 2)))
    return content_loss
    # pass
```

Test your content loss. You should see errors less than 0.0001.

[42]:
```
def content_loss_test(correct):
    content_image = 'styles/tubingen.jpg'
    image_size =  192
    content_layer = 3
    content_weight = 6e-2

    c_feats, content_img_var = features_from_img(content_image, image_size)

    bad_img = torch.zeros(*content_img_var.data.size()).type(dtype)
    feats = extract_features(bad_img, cnn)

    student_output = content_loss(content_weight, c_feats[content_layer],␣
↪feats[content_layer]).cpu().data.numpy()
    error = rel_error(correct, student_output)
    print('Maximum error is {:.3f}'.format(error))

content_loss_test(answers['cl_out'])
```

```
Maximum error is 0.000
```

1.4 Style loss

Now we can tackle the style loss. For a given layer ℓ, the style loss is defined as follows:

First, compute the Gram matrix G which represents the correlations between the responses of each filter, where F is as above. The Gram matrix is an approximation to the covariance matrix – we want the activation statistics of our generated image to match the activation statistics of our style image, and matching the (approximate) covariance is one way to do that. There are a variety of ways you could do this, but the Gram matrix is nice because it's easy to compute and in practice shows good results.

Given a feature map F^ℓ of shape (C_ℓ, M_ℓ), the Gram matrix has shape (C_ℓ, C_ℓ) and its elements are given by:

$$G^\ell_{ij} = \sum_k F^\ell_{ik} F^\ell_{jk}$$

Assuming G^ℓ is the Gram matrix from the feature map of the current image, A^ℓ is the Gram Matrix from the feature map of the source style image, and w_ℓ a scalar weight term, then the style loss for the layer ℓ is simply the weighted Euclidean distance between the two Gram matrices:

$$L_s^\ell = w_\ell \sum_{i,j} \left(G_{ij}^\ell - A_{ij}^\ell\right)^2$$

In practice we usually compute the style loss at a set of layers $\mathcal{L}$ rather than just a single layer ℓ; then the total style loss is the sum of style losses at each layer:

$$L_s = \sum_{\ell \in \mathcal{L}} L_s^\ell$$

Begin by implementing the Gram matrix computation below:

[43]:
```
def gram_matrix(features, normalize=True):
    """
    Compute the Gram matrix from features.

    Inputs:
    - features: PyTorch Tensor of shape (N, C, H, W) giving features for
      a batch of N images.
    - normalize: optional, whether to normalize the Gram matrix
        If True, divide the Gram matrix by the number of neurons (H * W * C)

    Returns:
    - gram: PyTorch Tensor of shape (N, C, C) giving the
      (optionally normalized) Gram matrices for the N input images.


    """
    pass

    N, C, H, W = features.size()

    features_reshaped = features.view(N, C, -1)
    gram = torch.bmm(features_reshaped, features_reshaped.transpose(1, 2))
    if normalize:
      return gram / (H * W * C)
    else:
      return gram
```

Test your Gram matrix code. You should see errors less than 0.0001.

[44]:
```
def gram_matrix_test(correct):
    style_image = 'styles/starry_night.jpg'
    style_size = 192
    feats, _ = features_from_img(style_image, style_size)
    student_output = gram_matrix(feats[5].clone()).cpu().data.numpy()
    error = rel_error(correct, student_output)
    print('Maximum error is {:.3f}'.format(error))
```

```
gram_matrix_test(answers['gm_out'])
```

```
Maximum error is 0.000
```

Next, implement the style loss:

[45]:
```
# Now put it together in the style_loss function...
def style_loss(feats, style_layers, style_targets, style_weights):
    """
    Computes the style loss at a set of layers.

    Inputs:
    - feats: list of the features at every layer of the current image, as␣
 ↪produced by
      the extract_features function.
    - style_layers: List of layer indices into feats giving the layers to␣
 ↪include in the
      style loss.
    - style_targets: List of the same length as style_layers, where␣
 ↪style_targets[i] is
      a PyTorch Tensor giving the Gram matrix of the source style image␣
 ↪computed at
      layer style_layers[i].
    - style_weights: List of the same length as style_layers, where␣
 ↪style_weights[i]
      is a scalar giving the weight for the style loss at layer style_layers[i].

    Returns:
    - style_loss: A PyTorch Tensor holding a scalar giving the style loss.
    """
    # Hint: you can do this with one for loop over the style layers, and should
    # not be very much code (~5 lines). You will need to use your gram_matrix␣
 ↪function.
    style_loss = 0
    for i in range(len(style_layers)):
      gram = gram_matrix(feats[style_layers[i]])
      # print("Gram shape", gram.shape)
      # print("Style targets shape", style_targets[i].shape)
      style_loss += style_weights[i]*torch.sum(torch.pow(gram-style_targets[i],␣
 ↪2))

    return style_loss
```

Test your style loss implementation. The error should be less than 0.0001.

[46]:
```
def style_loss_test(correct):
    content_image = 'styles/tubingen.jpg'
    style_image = 'styles/starry_night.jpg'
    image_size =  192
    style_size = 192
    style_layers = [1, 4, 6, 7]
    style_weights = [300000, 1000, 15, 3]

    c_feats, _ = features_from_img(content_image, image_size)
    feats, _ = features_from_img(style_image, style_size)
    style_targets = []
    for idx in style_layers:
        style_targets.append(gram_matrix(feats[idx].clone()))

    student_output = style_loss(c_feats, style_layers, style_targets,␣
↪style_weights).cpu().data.numpy()
    error = rel_error(correct, student_output)
    print('Error is {:.3f}'.format(error))


style_loss_test(answers['sl_out'])
```

```
Error is 0.000
```

1.5 Total-variation regularization

It turns out that it's helpful to also encourage smoothness in the image. We can do this by adding another term to our loss that penalizes wiggles or "total variation" in the pixel values.

You can compute the "total variation" as the sum of the squares of differences in the pixel values for all pairs of pixels that are next to each other (horizontally or vertically). Here we sum the total-variation regualarization for each of the 3 input channels (RGB), and weight the total summed loss by the total variation weight, w_t:

$$L_{tv} = w_t \times \left(\sum_{c=1}^{3}\sum_{i=1}^{H-1}\sum_{j=1}^{W}(x_{i+1,j,c} - x_{i,j,c})^2 + \sum_{c=1}^{3}\sum_{i=1}^{H}\sum_{j=1}^{W-1}(x_{i,j+1,c} - x_{i,j,c})^2\right)$$

In the next cell, fill in the definition for the TV loss term. To receive full credit, your implementation should not have any loops.

[47]:
```
def tv_loss(img, tv_weight):
    """
    Compute total variation loss.

    Inputs:
    - img: PyTorch Variable of shape (1, 3, H, W) holding an input image.
    - tv_weight: Scalar giving the weight w_t to use for the TV loss.

    Returns:
    - loss: PyTorch Variable holding a scalar giving the total variation loss
```

```
      for img weighted by tv_weight.
    """
    # Your implementation should be vectorized and not require any loops!
    pass

    w_variance = torch.sum(torch.pow(img[:, :, :, :-1] - img[:, :, :, 1:], 2))
    h_variance = torch.sum(torch.pow(img[:, :, :-1, :] - img[:, :, 1:, :], 2))

    loss = tv_weight * (h_variance + w_variance)
    return loss
```

Test your TV loss implementation. Error should be less than 0.0001.

[48]:
```
def tv_loss_test(correct):
    content_image = 'styles/tubingen.jpg'
    image_size =  192
    tv_weight = 2e-2

    content_img = preprocess(PIL.Image.open(content_image), size=image_size)

    student_output = tv_loss(content_img, tv_weight).cpu().data.numpy()
    error = rel_error(correct, student_output)
    print('Error is {:.3f}'.format(error))

tv_loss_test(answers['tv_out'])
```

```
Error is 0.000
```

Now we're ready to string it all together (you shouldn't have to modify this function):

[49]:
```
def style_transfer(content_image, style_image, image_size, style_size,␣
 ↪content_layer, content_weight,
                   style_layers, style_weights, tv_weight, init_random = False):
    """
    Run style transfer!

    Inputs:
    - content_image: filename of content image
    - style_image: filename of style image
    - image_size: size of smallest image dimension (used for content loss and␣
 ↪generated image)
    - style_size: size of smallest style image dimension
    - content_layer: layer to use for content loss
    - content_weight: weighting on content loss
    - style_layers: list of layers to use for style loss
    - style_weights: list of weights to use for each layer in style_layers
    - tv_weight: weight of total variation regularization term
    - init_random: initialize the starting image to uniform random noise
```

```
    """

    # Extract features for the content image
    content_img = preprocess(PIL.Image.open(content_image), size=image_size)
    feats = extract_features(content_img, cnn)
    content_target = feats[content_layer].clone()

    # Extract features for the style image
    style_img = preprocess(PIL.Image.open(style_image), size=style_size)
    feats = extract_features(style_img, cnn)
    style_targets = []
    for idx in style_layers:
        style_targets.append(gram_matrix(feats[idx].clone()))

    # Initialize output image to content image or nois
    if init_random:
        img = torch.Tensor(content_img.size()).uniform_(0, 1).type(dtype)
    else:
        img = content_img.clone().type(dtype)

    # We do want the gradient computed on our image!
    img.requires_grad_()

    # Set up optimization hyperparameters
    initial_lr = 3.0
    decayed_lr = 0.1
    decay_lr_at = 180

    # Note that we are optimizing the pixel values of the image by passing
    # in the img Torch tensor, whose requires_grad flag is set to True
    optimizer = torch.optim.Adam([img], lr=initial_lr)

    f, axarr = plt.subplots(1,2)
    axarr[0].axis('off')
    axarr[1].axis('off')
    axarr[0].set_title('Content Source Img.')
    axarr[1].set_title('Style Source Img.')
    axarr[0].imshow(deprocess(content_img.cpu()))
    axarr[1].imshow(deprocess(style_img.cpu()))
    plt.show()
    plt.figure()

    for t in range(200):
        if t < 190:
            img.data.clamp_(-1.5, 1.5)
        optimizer.zero_grad()
```

```
        feats = extract_features(img, cnn)

        # Compute loss
        c_loss = content_loss(content_weight, feats[content_layer],␣
↪content_target)
        s_loss = style_loss(feats, style_layers, style_targets, style_weights)
        t_loss = tv_loss(img, tv_weight)
        loss = c_loss + s_loss + t_loss

        loss.backward()

        # Perform gradient descents on our image values
        if t == decay_lr_at:
            optimizer = torch.optim.Adam([img], lr=decayed_lr)
        optimizer.step()

        if t % 100 == 0:
            print('Iteration {}'.format(t))
            plt.axis('off')
            plt.imshow(deprocess(img.data.cpu()))
            plt.show()
    print('Iteration {}'.format(t))
    plt.axis('off')
    plt.imshow(deprocess(img.data.cpu()))
    plt.show()
```

1.6 Generate some pretty pictures!

Try out `style_transfer` on the three different parameter sets below. Make sure to run all three cells. Feel free to add your own, but make sure to include the results of style transfer on the third parameter set (starry night) in your submitted notebook.

- The `content_image` is the filename of content image.
- The `style_image` is the filename of style image.
- The `image_size` is the size of smallest image dimension of the content image (used for content loss and generated image).
- The `style_size` is the size of smallest style image dimension.
- The `content_layer` specifies which layer to use for content loss.
- The `content_weight` gives weighting on content loss in the overall loss function. Increasing the value of this parameter will make the final image look more realistic (closer to the original content).
- `style_layers` specifies a list of which layers to use for style loss.
- `style_weights` specifies a list of weights to use for each layer in style_layers (each of which will contribute a term to the overall style loss). We generally use higher weights for the earlier style layers because they describe more local/smaller scale features, which are more important to texture than features over larger receptive fields. In general, increasing these weights will make the resulting image look less like the original content and more distorted towards the appearance of the style image.

- `tv_weight` specifies the weighting of total variation regularization in the overall loss function. Increasing this value makes the resulting image look smoother and less jagged, at the cost of lower fidelity to style and content.

Below the next three cells of code (in which you shouldn't change the hyperparameters), feel free to copy and paste the parameters to play around them and see how the resulting image changes.

[50]:
```
# Composition VII + Tubingen
params1 = {
    'content_image' : 'styles/tubingen.jpg',
    'style_image' : 'styles/composition_vii.jpg',
    'image_size' : 192,
    'style_size' : 512,
    'content_layer' : 3,
    'content_weight' : 5e-2,
    'style_layers' : (1, 4, 6, 7),
    'style_weights' : (20000, 500, 12, 1),
    'tv_weight' : 5e-2
}

style_transfer(**params1)
```


```
Iteration 0
```

Iteration 100

```
Iteration 199
```

[51]:
```
# Scream + Tubingen
params2 = {
    'content_image':'styles/tubingen.jpg',
    'style_image':'styles/the_scream.jpg',
    'image_size':192,
    'style_size':224,
    'content_layer':3,
    'content_weight':3e-2,
    'style_layers':[1, 4, 6, 7],
    'style_weights':[200000, 800, 12, 1],
    'tv_weight':2e-2
}

style_transfer(**params2)
```

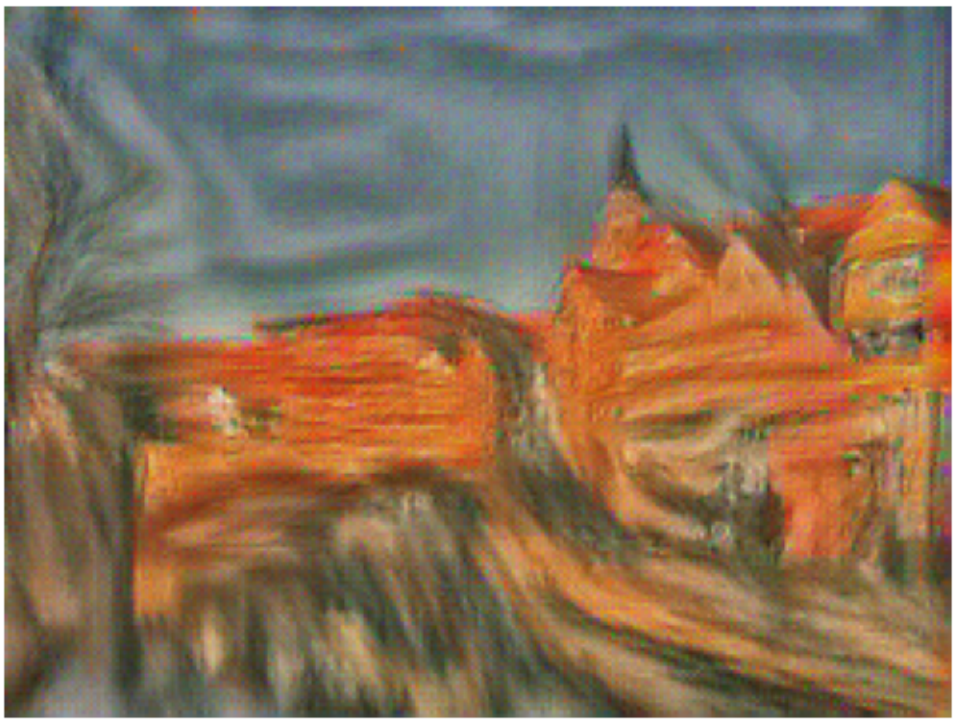
Content Source Img.

Style Source Img.

```
Iteration 0
```

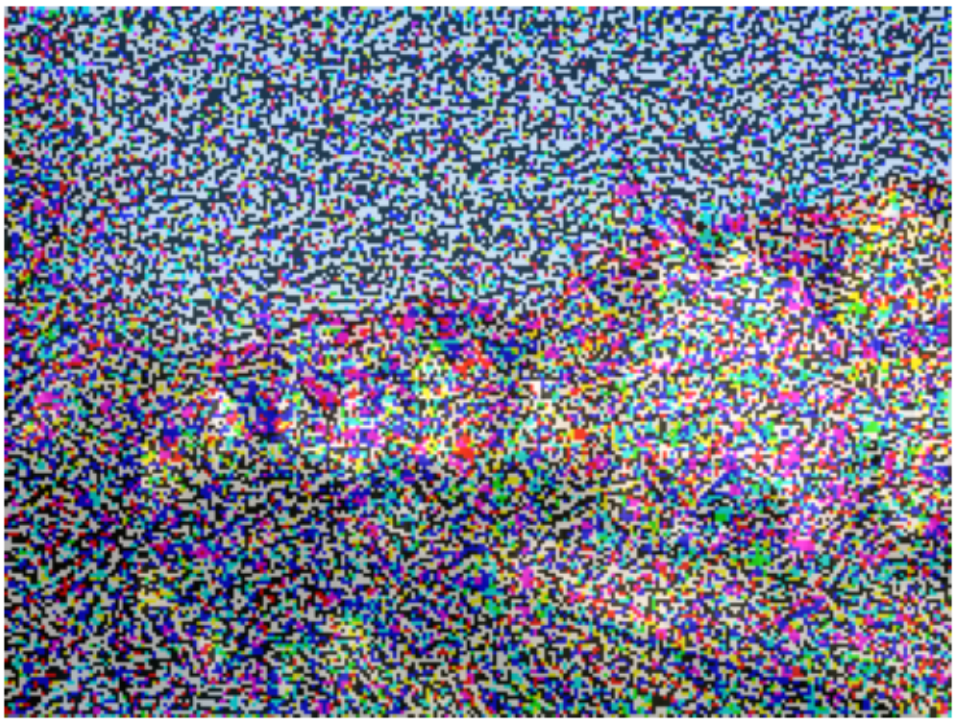
```
Iteration 100
```

Iteration 199



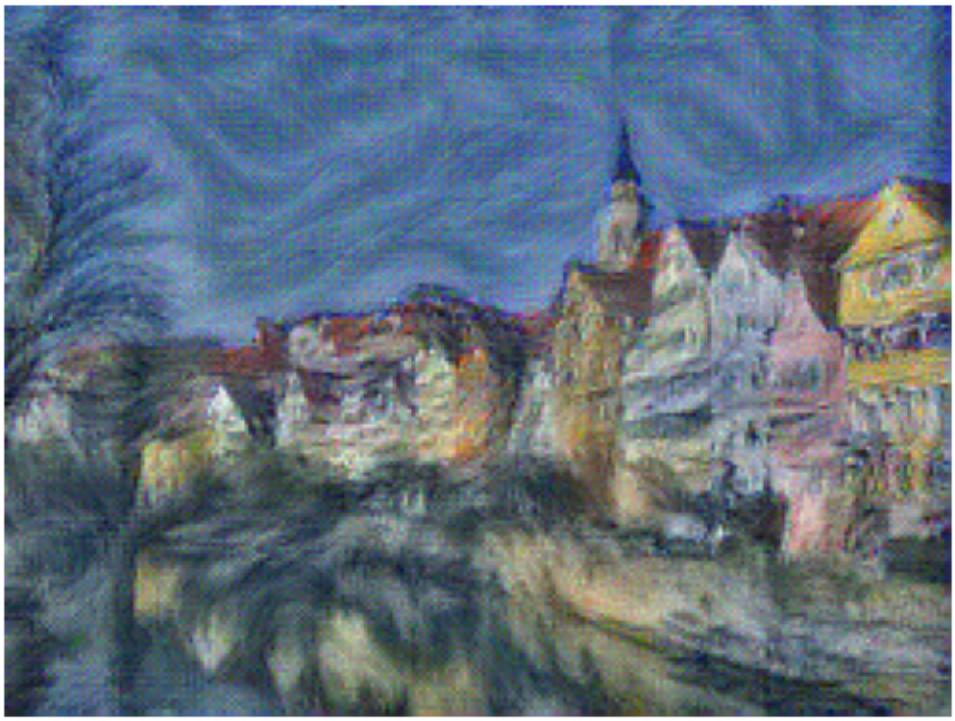
[52]:
```
# Starry Night + Tubingen
params3 = {
    'content_image' : 'styles/tubingen.jpg',
    'style_image' : 'styles/starry_night.jpg',
    'image_size' : 192,
    'style_size' : 192,
    'content_layer' : 3,
    'content_weight' : 6e-2,
    'style_layers' : [1, 4, 6, 7],
    'style_weights' : [300000, 1000, 15, 3],
    'tv_weight' : 2e-2
}

style_transfer(**params3)
```


```
Iteration 0
```



Iteration 100

```
Iteration 199
```



1.7 Feature Inversion

The code you've written can do another cool thing. In an attempt to understand the types of features that convolutional networks learn to recognize, a recent paper [1] attempts to reconstruct an image from its feature representation. We can easily implement this idea using image gradients from the pretrained network, which is exactly what we did above (but with two different feature representations).

Now, if you set the style weights to all be 0 and initialize the starting image to random noise instead of the content source image, you'll reconstruct an image from the feature representation of the content source image. You're starting with total noise, but you should end up with something that looks quite a bit like your original image.

(Similarly, you could do "texture synthesis" from scratch if you set the content weight to 0 and initialize the starting image to random noise, but we won't ask you to do that here.)

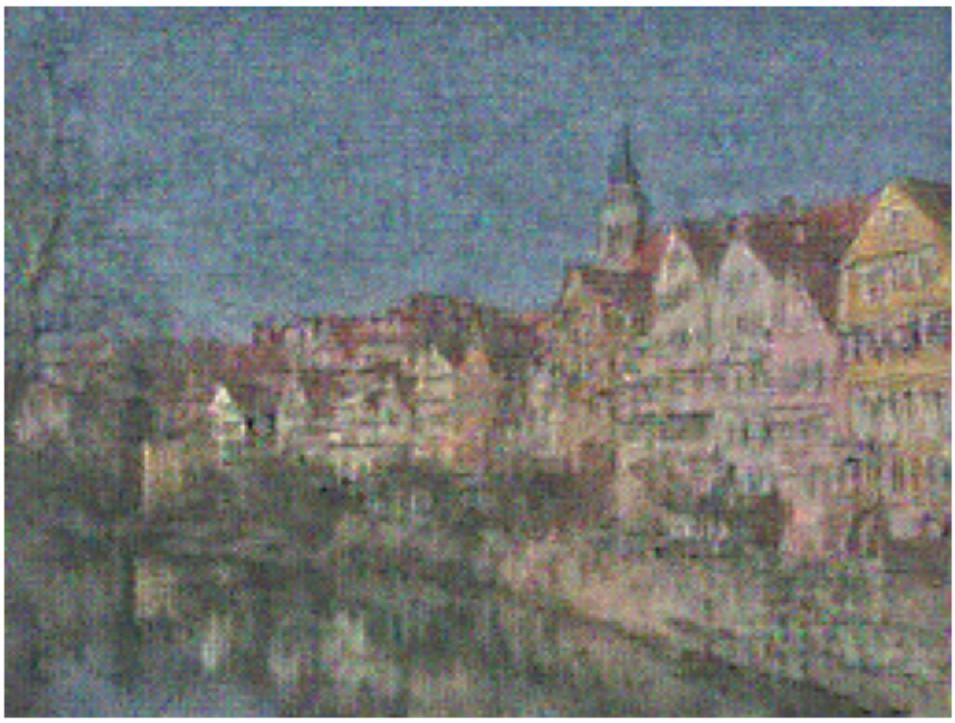
Run the following cell to try out feature inversion.

[1] Aravindh Mahendran, Andrea Vedaldi, "Understanding Deep Image Representations by Inverting them", CVPR 2015

[53]:
```
# Feature Inversion -- Starry Night + Tubingen
params_inv = {
    'content_image' : 'styles/tubingen.jpg',
    'style_image' : 'styles/starry_night.jpg',
    'image_size' : 192,
    'style_size' : 192,
    'content_layer' : 3,
    'content_weight' : 6e-2,
    'style_layers' : [1, 4, 6, 7],
    'style_weights' : [0, 0, 0, 0], # we discard any contributions from style␣
 ↪to the loss
    'tv_weight' : 2e-2,
    'init_random': True # we want to initialize our image to be random
}

style_transfer(**params_inv)
```


```
Iteration 0
```

Iteration 100



Iteration 199