Network_Visualization

April 14, 2025

[ ]:
```
# 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. 'cs231n/assignments/assignment3/'
FOLDERNAME = None
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/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

1 Network Visualization

In this notebook, we will explore the use of *image gradients* for generating new images.

When training a model, we define a loss function which measures our current unhappiness with the model's performance. We then use backpropagation to compute the gradient of the loss with respect to the model parameters and perform gradient descent on the model parameters to minimize the loss.

Here we will do something slightly different. We will start from a CNN model which has been pretrained to perform image classification on the ImageNet dataset. We will use this model to define a loss function which quantifies our current unhappiness with our image. Then we will use backpropagation to compute the gradient of this loss with respect to the pixels of the image. We will then keep the model fixed and perform gradient descent *on the image* to synthesize a new image which minimizes the loss.

We will explore three techniques for image generation.

Saliency Maps. We can use saliency maps to tell which part of the image influenced the classification decision made by the network.

Fooling Images. We can perturb an input image so that it appears the same to humans but will be misclassified by the pretrained network.

Class Visualization. We can synthesize an image to maximize the classification score of a particular class; this can give us some sense of what the network is looking for when it classifies images of that class.

[ ]:
```
# Setup cell.
import torch
import torchvision
import numpy as np
import random
import matplotlib.pyplot as plt
from PIL import Image
from cs231n.image_utils import SQUEEZENET_MEAN, SQUEEZENET_STD
from cs231n.net_visualization_pytorch import *

%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
```

2 Pretrained Model

For all of our image generation experiments, we will start with a convolutional neural network which was pretrained to perform image classification on ImageNet. We can use any model here, but for the purposes of this assignment we will use SqueezeNet [1], which achieves accuracies comparable to AlexNet but with a significantly reduced parameter count and computational complexity.

Using SqueezeNet rather than AlexNet or VGG or ResNet means that we can easily perform all image generation experiments on CPU.

[1] Iandola et al, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5MB model size", arXiv 2016

[ ]:
```
# Download and load the pretrained SqueezeNet model.
model = torchvision.models.squeezenet1_1(pretrained=True)

# We don't want to train the model, so tell PyTorch not to compute gradients
# with respect to model parameters.
for param in model.parameters():
    param.requires_grad = False
```

2.1 Loading ImageNet Validation Images

We have provided a few example images from the validation set of the ImageNet ILSVRC 2012 Classification dataset. Since they come from the validation set, our pretrained model did not see these images during training. Run the following cell to visualize some of these images along with their ground-truth labels.

[ ]:
```
from cs231n.data_utils import load_imagenet_val
X, y, class_names = load_imagenet_val(num=5)

plt.figure(figsize=(12, 6))
for i in range(5):
    plt.subplot(1, 5, i + 1)
    plt.imshow(X[i])
    plt.title(class_names[y[i]])
    plt.axis('off')
plt.gcf().tight_layout()
```

3 Saliency Maps

Using this pretrained model, we will compute class saliency maps as described in Section 3.1 of [2].

A **saliency map** tells us the degree to which each pixel in the image affects the classification score for that image. To compute it, we compute the gradient of the unnormalized score corresponding to the correct class (which is a scalar) with respect to the pixels of the image. If the image has shape `(3, H, W)` then this gradient will also have shape `(3, H, W)`; for each pixel in the image, this gradient tells us the amount by which the classification score will change if the pixel changes by a small amount. To compute the saliency map, we take the absolute value of this gradient, then take the maximum value over the 3 input channels; the final saliency map thus has shape `(H, W)` and all entries are nonnegative.

[2] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.

3.0.1 Hint: PyTorch `gather` method

Recall in Assignment 1 you needed to select one element from each row of a matrix; if `s` is an numpy array of shape `(N, C)` and `y` is a numpy array of shape `(N,)` containing integers `0 <= y[i] < C`, then `s[np.arange(N), y]` is a numpy array of shape `(N,)` which selects one element from each element in `s` using the indices in `y`.

In PyTorch you can perform the same operation using the `gather()` method. If `s` is a PyTorch Tensor of shape `(N, C)` and `y` is a PyTorch Tensor of shape `(N,)` containing longs in the range `0 <= y[i] < C`, then

`s.gather(1, y.view(-1, 1)).squeeze()`

will be a PyTorch Tensor of shape `(N,)` containing one entry from each row of `s`, selected according to the indices in `y`.

run the following cell to see an example.

You can also read the documentation for the gather method and the squeeze method.

```
[ ]: # Example of using gather to select one entry from each row in PyTorch
def gather_example():
    N, C = 4, 5
    s = torch.randn(N, C)
    y = torch.LongTensor([1, 2, 1, 3])
    print(s)
    print(y)
    print(s.gather(1, y.view(-1, 1)).squeeze())
gather_example()
```

Implement `compute_saliency_maps` function inside `cs231n/net_visualization_pytorch.py`

Once you have completed the implementation above, run the following to visualize some class saliency maps on our example images from the ImageNet validation set:

```
[ ]: def show_saliency_maps(X, y):
    # Convert X and y from numpy arrays to Torch Tensors
    X_tensor = torch.cat([preprocess(Image.fromarray(x)) for x in X], dim=0)
    y_tensor = torch.LongTensor(y)

    # Compute saliency maps for images in X
    saliency = compute_saliency_maps(X_tensor, y_tensor, model)

    # Convert the saliency map from Torch Tensor to numpy array and show images
    # and saliency maps together.
    saliency = saliency.numpy()
    N = X.shape[0]
    for i in range(N):
        plt.subplot(2, N, i + 1)
        plt.imshow(X[i])
        plt.axis('off')
        plt.title(class_names[y[i]])
        plt.subplot(2, N, N + i + 1)
        plt.imshow(saliency[i], cmap=plt.cm.hot)
        plt.axis('off')
        plt.gcf().set_size_inches(12, 5)
    plt.show()

show_saliency_maps(X, y)
```

4 Inline Question 1

A friend of yours suggests that in order to find an image that maximizes the correct score, we can perform gradient ascent on the input image, but instead of the gradient we can actually use the saliency map in each step to update the image. Is this assertion true? Why or why not?

Your Answer:

5 Fooling Images

We can also use image gradients to generate "fooling images" as discussed in [3]. Given an image and a target class, we can perform gradient **ascent** over the image to maximize the target class, stopping when the network classifies the image as the target class. Implement the following function to generate fooling images.

[3] Szegedy et al, "Intriguing properties of neural networks", ICLR 2014

Implement `make_fooling_image` function inside `cs231n/net_visualization_pytorch.py`

Run the following cell to generate a fooling image. You should ideally see at first glance no major difference between the original and fooling images, and the network should now make an incorrect prediction on the fooling one. However you should see a bit of random noise if you look at the 10x magnified difference between the original and fooling images. Feel free to change the `idx` variable to explore other images.

[ ]:
```
idx = 0
target_y = 6

X_tensor = torch.cat([preprocess(Image.fromarray(x)) for x in X], dim=0)
X_fooling = make_fooling_image(X_tensor[idx:idx+1], target_y, model)

scores = model(X_fooling)
assert target_y == scores.data.max(1)[1][0].item(), 'The model is not fooled!'
```

After generating a fooling image, run the following cell to visualize the original image, the fooling image, as well as the difference between them.

[ ]:
```
X_fooling_np = deprocess(X_fooling.clone())
X_fooling_np = np.asarray(X_fooling_np).astype(np.uint8)

plt.subplot(1, 4, 1)
plt.imshow(X[idx])
plt.title(class_names[y[idx]])
plt.axis('off')

plt.subplot(1, 4, 2)
plt.imshow(X_fooling_np)
plt.title(class_names[target_y])
plt.axis('off')

plt.subplot(1, 4, 3)
X_pre = preprocess(Image.fromarray(X[idx]))
diff = np.asarray(deprocess(X_fooling - X_pre, should_rescale=False))
plt.imshow(diff)
plt.title('Difference')
```

```
plt.axis('off')

plt.subplot(1, 4, 4)
diff = np.asarray(deprocess(10 * (X_fooling - X_pre), should_rescale=False))
plt.imshow(diff)
plt.title('Magnified difference (10x)')
plt.axis('off')

plt.gcf().set_size_inches(12, 5)
plt.show()
```

6 Class Visualization

By starting with a random noise image and performing gradient ascent on a target class, we can generate an image that the network will recognize as the target class. This idea was first presented in [2]; [3] extended this idea by suggesting several regularization techniques that can improve the quality of the generated image.

Concretely, let I be an image and let y be a target class. Let $s_y(I)$ be the score that a convolutional network assigns to the image I for class y; note that these are raw unnormalized scores, not class probabilities. We wish to generate an image I^* that achieves a high score for the class y by solving the problem

$$I^* = \arg\max_I (s_y(I) - R(I))$$

where R is a (possibly implicit) regularizer (note the sign of $R(I)$ in the argmax: we want to minimize this regularization term). We can solve this optimization problem using gradient ascent, computing gradients with respect to the generated image. We will use (explicit) L2 regularization of the form

$$R(I) = \lambda \|I\|_2^2$$

and implicit regularization as suggested by [3] by periodically blurring the generated image. We can solve this problem using gradient ascent on the generated image.

[2] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.

[3] Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML 2015 Deep Learning Workshop

In `cs231n/net_visualization_pytorch.py` complete the implementation of the `class_visualization_update_step` used in the `create_class_visualization` function below. Once you have completed that implementation, run the following cells to generate an image of a Tarantula:

[ ]:
```
def create_class_visualization(target_y, model, dtype, **kwargs):
    """
```

```
    Generate an image to maximize the score of target_y under a pretrained␣
↪model.

    Inputs:
    - target_y: Integer in the range [0, 1000) giving the index of the class
    - model: A pretrained CNN that will be used to generate the image
    - dtype: Torch datatype to use for computations

    Keyword arguments:
    - l2_reg: Strength of L2 regularization on the image
    - learning_rate: How big of a step to take
    - num_iterations: How many iterations to use
    - blur_every: How often to blur the image as an implicit regularizer
    - max_jitter: How much to gjitter the image as an implicit regularizer
    - show_every: How often to show the intermediate result
    """
    model.type(dtype)
    l2_reg = kwargs.pop('l2_reg', 1e-3)
    learning_rate = kwargs.pop('learning_rate', 25)
    num_iterations = kwargs.pop('num_iterations', 100)
    blur_every = kwargs.pop('blur_every', 10)
    max_jitter = kwargs.pop('max_jitter', 16)
    show_every = kwargs.pop('show_every', 25)

    # Randomly initialize the image as a PyTorch Tensor, and make it requires␣
↪gradient.
    img = torch.randn(1, 3, 224, 224).mul_(1.0).type(dtype).requires_grad_()

    for t in range(num_iterations):
        # Randomly jitter the image a bit; this gives slightly nicer results
        ox, oy = random.randint(0, max_jitter), random.randint(0, max_jitter)
        img.data.copy_(jitter(img.data, ox, oy))
        class_visualization_update_step(img, model, target_y, l2_reg,␣
↪learning_rate)
        # Undo the random jitter
        img.data.copy_(jitter(img.data, -ox, -oy))

        # As regularizer, clamp and periodically blur the image
        for c in range(3):
            lo = float(-SQUEEZENET_MEAN[c] / SQUEEZENET_STD[c])
            hi = float((1.0 - SQUEEZENET_MEAN[c]) / SQUEEZENET_STD[c])
            img.data[:, c].clamp_(min=lo, max=hi)
        if t % blur_every == 0:
            blur_image(img.data, sigma=0.5)

        # Periodically show the image
        if t == 0 or (t + 1) % show_every == 0 or t == num_iterations - 1:
```

```
            plt.imshow(deprocess(img.data.clone().cpu()))
            class_name = class_names[target_y]
            plt.title('%s\nIteration %d / %d' % (class_name, t + 1,␣
↪num_iterations))
            plt.gcf().set_size_inches(4, 4)
            plt.axis('off')
            plt.show()

    return deprocess(img.data.cpu())
```

[ ]:
```
dtype = torch.FloatTensor
model.type(dtype)

target_y = 76 # Tarantula
# target_y = 78 # Tick
# target_y = 187 # Yorkshire Terrier
# target_y = 683 # Oboe
# target_y = 366 # Gorilla
# target_y = 604 # Hourglass
out = create_class_visualization(target_y, model, dtype)
```

Try out your class visualization on other classes! You should also feel free to play with various hyperparameters to try and improve the quality of the generated image, but this is not required.

[ ]:
```
# target_y = 78 # Tick
# target_y = 187 # Yorkshire Terrier
# target_y = 683 # Oboe
# target_y = 366 # Gorilla
# target_y = 604 # Hourglass
target_y = np.random.randint(1000)
print(class_names[target_y])
X = create_class_visualization(target_y, model, dtype)
```

[ ]:
```

```

RNN_Captioning

April 14, 2025

[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. 'cs231n/assignments/assignment3/'
FOLDERNAME = 'COMP4471/assignments/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/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/COMP4471/assignments/assignment3/cs231n/datasets
/content/drive/My Drive/COMP4471/assignments/assignment3
```

1 Image Captioning with RNNs

In this exercise, you will implement vanilla Recurrent Neural Networks and use them to train a model that can generate novel captions for images.

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

from cs231n.gradient_check import eval_numerical_gradient,␣
↪eval_numerical_gradient_array
```

```python
from cs231n.rnn_layers import *
from cs231n.captioning_solver import CaptioningSolver
from cs231n.classifiers.rnn import CaptioningRNN
from cs231n.coco_utils import load_coco_data, sample_coco_minibatch,␣
 ↪decode_captions
from cs231n.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

For this exercise, we will use the 2014 release of the COCO dataset, a standard testbed for image captioning. The dataset consists of 80,000 training images and 40,000 validation images, each annotated with 5 captions written by workers on Amazon Mechanical Turk.

Image features. We have preprocessed the data and extracted features for you already. For all images, we have extracted features from the fc7 layer of the VGG-16 network pretrained on ImageNet, and these features are stored in the files `train2014_vgg16_fc7.h5` and `val2014_vgg16_fc7.h5`. To cut down on processing time and memory requirements, we have reduced the dimensionality of the features from 4096 to 512 using Principal Component Analysis (PCA), and these features are stored in the files `train2014_vgg16_fc7_pca.h5` and `val2014_vgg16_fc7_pca.h5`. The raw images take up nearly 20GB of space so we have not included them in the download. Since all images are taken from Flickr, we have stored the URLs of the training and validation images in the files `train2014_urls.txt` and `val2014_urls.txt`. This allows you to download images on-the-fly for visualization.

Captions. Dealing with strings is inefficient, so we will work with an encoded version of the captions. Each word is assigned an integer ID, allowing us to represent a caption by a sequence of integers. The mapping between integer IDs and words is in the file `coco2014_vocab.json`, and you can use the function `decode_captions` from the file `cs231n/coco_utils.py` to convert NumPy arrays of integer IDs back into strings.

Tokens. There are a couple special tokens that we add to the vocabulary, and we have taken care of all implementation details around special tokens for you. We prepend a special `<START>` token and append an `<END>` token to the beginning and end of each caption respectively. Rare words are replaced with a special `<UNK>` token (for "unknown"). In addition, since we want to train with minibatches containing captions of different lengths, we pad short captions with a special `<NULL>` token after the `<END>` token and don't compute loss or gradient for `<NULL>` tokens.

You can load all of the COCO data (captions, features, URLs, and vocabulary) using the `load_coco_data` function from the file `cs231n/coco_utils.py`. Run the following cell to do so:

[3]:
```
# Load COCO data from disk into a dictionary.
# We'll work with dimensionality-reduced features for the remainder of this␣
↪assignment,
# but you can also experiment with the original features on your own by␣
↪changing the flag below.
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/My
Drive/COMP4471/assignments/assignment3/cs231n/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
```

2.1 Inspect the Data

It is always a good idea to look at examples from the dataset before working with it.

You can use the `sample_coco_minibatch` function from the file `cs231n/coco_utils.py` to sample minibatches of data from the data structure returned from `load_coco_data`. Run the following to sample a small minibatch of training data and show the images and their captions. Running it multiple times and looking at the results helps you to get a sense of the dataset.

[5]:
```
# Sample a minibatch and show the images and captions.
# 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.
batch_size = 3

captions, features, urls = sample_coco_minibatch(data, batch_size=batch_size)
for i, (caption, url) in enumerate(zip(captions, urls)):
    plt.imshow(image_from_url(url))
    plt.axis('off')
```

```
    caption_str = decode_captions(caption, data['idx_to_word'])
    plt.title(caption_str)
    plt.show()
```

Output hidden; open in https://colab.research.google.com to view.

3 Recurrent Neural Network

As discussed in lecture, we will use Recurrent Neural Network (RNN) language models for image captioning. The file `cs231n/rnn_layers.py` contains implementations of different layer types that are needed for recurrent neural networks, and the file `cs231n/classifiers/rnn.py` uses these layers to implement an image captioning model.

We will first implement different types of RNN layers in `cs231n/rnn_layers.py`.

NOTE: The Long-Short Term Memory (LSTM) RNN is a common variant of the vanilla RNN. `LSTM_Captioning.ipynb` is optional extra credit, so don't worry about references to LSTM in `cs231n/classifiers/rnn.py` and `cs231n/rnn_layers.py` for now.

4 Vanilla RNN: Step Forward

Open the file `cs231n/rnn_layers.py`. This file implements the forward and backward passes for different types of layers that are commonly used in recurrent neural networks.

First implement the function `rnn_step_forward` which implements the forward pass for a single timestep of a vanilla recurrent neural network. After doing so run the following to check your implementation. You should see errors on the order of e-8 or less.

[6]:
```
N, D, H = 3, 10, 4

x = np.linspace(-0.4, 0.7, num=N*D).reshape(N, D)
prev_h = np.linspace(-0.2, 0.5, num=N*H).reshape(N, H)
Wx = np.linspace(-0.1, 0.9, num=D*H).reshape(D, H)
Wh = np.linspace(-0.3, 0.7, num=H*H).reshape(H, H)
b = np.linspace(-0.2, 0.4, num=H)

next_h, _ = rnn_step_forward(x, prev_h, Wx, Wh, b)
expected_next_h = np.asarray([
  [-0.58172089, -0.50182032, -0.41232771, -0.31410098],
  [ 0.66854692,  0.79562378,  0.87755553,  0.92795967],
  [ 0.97934501,  0.99144213,  0.99646691,  0.99854353]])

print('next_h error: ', rel_error(expected_next_h, next_h))
```

```
next_h error:  6.292421426471037e-09
```

5 Vanilla RNN: Step Backward

In the file `cs231n/rnn_layers.py` implement the `rnn_step_backward` function. After doing so run the following to numerically gradient check your implementation. You should see errors on the order of `e-8` or less.

[7]:
```
from cs231n.rnn_layers import rnn_step_forward, rnn_step_backward
np.random.seed(231)
N, D, H = 4, 5, 6
x = np.random.randn(N, D)
h = np.random.randn(N, H)
Wx = np.random.randn(D, H)
Wh = np.random.randn(H, H)
b = np.random.randn(H)

out, cache = rnn_step_forward(x, h, Wx, Wh, b)

dnext_h = np.random.randn(*out.shape)

fx = lambda x: rnn_step_forward(x, h, Wx, Wh, b)[0]
fh = lambda prev_h: rnn_step_forward(x, h, Wx, Wh, b)[0]
fWx = lambda Wx: rnn_step_forward(x, h, Wx, Wh, b)[0]
fWh = lambda Wh: rnn_step_forward(x, h, Wx, Wh, b)[0]
fb = lambda b: rnn_step_forward(x, h, Wx, Wh, b)[0]

dx_num = eval_numerical_gradient_array(fx, x, dnext_h)
dprev_h_num = eval_numerical_gradient_array(fh, h, dnext_h)
dWx_num = eval_numerical_gradient_array(fWx, Wx, dnext_h)
dWh_num = eval_numerical_gradient_array(fWh, Wh, dnext_h)
db_num = eval_numerical_gradient_array(fb, b, dnext_h)

dx, dprev_h, dWx, dWh, db = rnn_step_backward(dnext_h, cache)

print('dx error: ', rel_error(dx_num, dx))
print('dprev_h error: ', rel_error(dprev_h_num, dprev_h))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))
```

```
dx error:  2.7795541640745535e-10
dprev_h error:  2.732467428030486e-10
dWx error:  9.709219069305414e-10
dWh error:  5.034262638717296e-10
db error:  1.708752322503098e-11
```

6 Vanilla RNN: Forward

Now that you have implemented the forward and backward passes for a single timestep of a vanilla RNN, you will combine these pieces to implement a RNN that processes an entire sequence of data.

In the file `cs231n/rnn_layers.py`, implement the function `rnn_forward`. This should be implemented using the `rnn_step_forward` function that you defined above. After doing so run the following to check your implementation. You should see errors on the order of `e-7` or less.

[8]:
```
N, T, D, H = 2, 3, 4, 5

x = np.linspace(-0.1, 0.3, num=N*T*D).reshape(N, T, D)
h0 = np.linspace(-0.3, 0.1, num=N*H).reshape(N, H)
Wx = np.linspace(-0.2, 0.4, num=D*H).reshape(D, H)
Wh = np.linspace(-0.4, 0.1, num=H*H).reshape(H, H)
b = np.linspace(-0.7, 0.1, num=H)

h, _ = rnn_forward(x, h0, Wx, Wh, b)
expected_h = np.asarray([
  [
    [-0.42070749, -0.27279261, -0.11074945,  0.05740409,  0.22236251],
    [-0.39525808, -0.22554661, -0.0409454,   0.14649412,  0.32397316],
    [-0.42305111, -0.24223728, -0.04287027,  0.15997045,  0.35014525],
  ],
  [
    [-0.55857474, -0.39065825, -0.19198182,  0.02378408,  0.23735671],
    [-0.27150199, -0.07088804,  0.13562939,  0.33099728,  0.50158768],
    [-0.51014825, -0.30524429, -0.06755202,  0.17806392,  0.40333043]]])
print('h error: ', rel_error(expected_h, h))
```

```
h error:  7.728466151011529e-08
```

7 Vanilla RNN: Backward

In the file `cs231n/rnn_layers.py`, implement the backward pass for a vanilla RNN in the function `rnn_backward`. This should run back-propagation over the entire sequence, making calls to the `rnn_step_backward` function that you defined earlier. You should see errors on the order of `e-6` or less.

[9]:
```
np.random.seed(231)

N, D, T, H = 2, 3, 10, 5

x = np.random.randn(N, T, D)
h0 = np.random.randn(N, H)
Wx = np.random.randn(D, H)
Wh = np.random.randn(H, H)
b = np.random.randn(H)
```

```python
out, cache = rnn_forward(x, h0, Wx, Wh, b)

dout = np.random.randn(*out.shape)

dx, dh0, dWx, dWh, db = rnn_backward(dout, cache)

fx = lambda x: rnn_forward(x, h0, Wx, Wh, b)[0]
fh0 = lambda h0: rnn_forward(x, h0, Wx, Wh, b)[0]
fWx = lambda Wx: rnn_forward(x, h0, Wx, Wh, b)[0]
fWh = lambda Wh: rnn_forward(x, h0, Wx, Wh, b)[0]
fb = lambda b: rnn_forward(x, h0, Wx, Wh, b)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
dh0_num = eval_numerical_gradient_array(fh0, h0, dout)
dWx_num = eval_numerical_gradient_array(fWx, Wx, dout)
dWh_num = eval_numerical_gradient_array(fWh, Wh, dout)
db_num = eval_numerical_gradient_array(fb, b, dout)

print('dx error: ', rel_error(dx_num, dx))
print('dh0 error: ', rel_error(dh0_num, dh0))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))
```

```
dx error:  1.5354482248401769e-09
dh0 error:  3.3830821485562176e-09
dWx error:  7.23583883274483e-09
dWh error:  1.3049601378601992e-07
db error:  1.5197668388626435e-10
```

8 Word Embedding: Forward

In deep learning systems, we commonly represent words using vectors. Each word of the vocabulary will be associated with a vector, and these vectors will be learned jointly with the rest of the system.

In the file `cs231n/rnn_layers.py`, implement the function `word_embedding_forward` to convert words (represented by integers) into vectors. Run the following to check your implementation. You should see an error on the order of `e-8` or less.

[10]:
```python
N, T, V, D = 2, 4, 5, 3

x = np.asarray([[0, 3, 1, 2], [2, 1, 0, 3]])
W = np.linspace(0, 1, num=V*D).reshape(V, D)

out, _ = word_embedding_forward(x, W)
expected_out = np.asarray([
 [[ 0.,          0.07142857,  0.14285714],
```

```
  [ 0.64285714,  0.71428571,  0.78571429],
  [ 0.21428571,  0.28571429,  0.35714286],
  [ 0.42857143,  0.5,         0.57142857]],
 [[ 0.42857143,  0.5,         0.57142857],
  [ 0.21428571,  0.28571429,  0.35714286],
  [ 0.,          0.07142857,  0.14285714],
  [ 0.64285714,  0.71428571,  0.78571429]]])

print('out error: ', rel_error(expected_out, out))
```

```
out error:  1.0000000094736443e-08
```

9 Word Embedding: Backward

Implement the backward pass for the word embedding function in the function `word_embedding_backward`. After doing so run the following to numerically gradient check your implementation. You should see an error on the order of `e-11` or less.

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

N, T, V, D = 50, 3, 5, 6
x = np.random.randint(V, size=(N, T))
W = np.random.randn(V, D)

out, cache = word_embedding_forward(x, W)
dout = np.random.randn(*out.shape)
dW = word_embedding_backward(dout, cache)

f = lambda W: word_embedding_forward(x, W)[0]
dW_num = eval_numerical_gradient_array(f, W, dout)

print('dW error: ', rel_error(dW, dW_num))
```

```
dW error:  3.2774595693100364e-12
```

10 Temporal Affine Layer

At every timestep we use an affine function to transform the RNN hidden vector at that timestep into scores for each word in the vocabulary. Because this is very similar to the affine layer that you implemented in assignment 2, we have provided this function for you in the `temporal_affine_forward` and `temporal_affine_backward` functions in the file `cs231n/rnn_layers.py`. Run the following to perform numeric gradient checking on the implementation. You should see errors on the order of `e-9` or less.

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

# Gradient check for temporal affine layer
```

```
N, T, D, M = 2, 3, 4, 5
x = np.random.randn(N, T, D)
w = np.random.randn(D, M)
b = np.random.randn(M)

out, cache = temporal_affine_forward(x, w, b)

dout = np.random.randn(*out.shape)

fx = lambda x: temporal_affine_forward(x, w, b)[0]
fw = lambda w: temporal_affine_forward(x, w, b)[0]
fb = lambda b: temporal_affine_forward(x, w, b)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
dw_num = eval_numerical_gradient_array(fw, w, dout)
db_num = eval_numerical_gradient_array(fb, b, dout)

dx, dw, db = temporal_affine_backward(dout, cache)

print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
dx error:  2.9215945034030545e-10
dw error:  1.5772088618663602e-10
db error:  3.252200556967514e-11
```

11 Temporal Softmax Loss

In an RNN language model, at every timestep we produce a score for each word in the vocabulary. We know the ground-truth word at each timestep, so we use a softmax loss function to compute loss and gradient at each timestep. We sum the losses over time and average them over the minibatch.

However there is one wrinkle: since we operate over minibatches and different captions may have different lengths, we append `<NULL>` tokens to the end of each caption so they all have the same length. We don't want these `<NULL>` tokens to count toward the loss or gradient, so in addition to scores and ground-truth labels our loss function also accepts a `mask` array that tells it which elements of the scores count towards the loss.

Since this is very similar to the softmax loss function you implemented in assignment 1, we have implemented this loss function for you; look at the `temporal_softmax_loss` function in the file `cs231n/rnn_layers.py`.

Run the following cell to sanity check the loss and perform numeric gradient checking on the function. You should see an error for dx on the order of `e-7` or less.

[13]:
```
# Sanity check for temporal softmax loss
from cs231n.rnn_layers import temporal_softmax_loss
```

```
N, T, V = 100, 1, 10

def check_loss(N, T, V, p):
    x = 0.001 * np.random.randn(N, T, V)
    y = np.random.randint(V, size=(N, T))
    mask = np.random.rand(N, T) <= p
    print(temporal_softmax_loss(x, y, mask)[0])

check_loss(100, 1, 10, 1.0)   # Should be about 2.3
check_loss(100, 10, 10, 1.0)  # Should be about 23
check_loss(5000, 10, 10, 0.1) # Should be within 2.2-2.4

# Gradient check for temporal softmax loss
N, T, V = 7, 8, 9

x = np.random.randn(N, T, V)
y = np.random.randint(V, size=(N, T))
mask = (np.random.rand(N, T) > 0.5)

loss, dx = temporal_softmax_loss(x, y, mask, verbose=False)

dx_num = eval_numerical_gradient(lambda x: temporal_softmax_loss(x, y,␣
 ↪mask)[0], x, verbose=False)

print('dx error: ', rel_error(dx, dx_num))
```

```
2.3027781774290146
23.025985953127226
2.2643611790293394
dx error:  2.583585303524283e-08
```

12 RNN for Image Captioning

Now that you have implemented the necessary layers, you can combine them to build an image captioning model. Open the file `cs231n/classifiers/rnn.py` and look at the `CaptioningRNN` class.

Implement the forward and backward pass of the model in the `loss` function. For now you only need to implement the case where `cell_type='rnn'` for vanilla RNNs; you will implement the LSTM case later. 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-10` or less.

[14]:
```
N, D, W, H = 10, 20, 30, 40
word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
V = len(word_to_idx)
T = 13
```

```
model = CaptioningRNN(
    word_to_idx,
    input_dim=D,
    wordvec_dim=W,
    hidden_dim=H,
    cell_type='rnn',
    dtype=np.float64
)

# Set all model parameters to fixed values
for k, v in model.params.items():
    model.params[k] = np.linspace(-1.4, 1.3, num=v.size).reshape(*v.shape)

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

loss, grads = model.loss(features, captions)
expected_loss = 9.83235591003

print('loss: ', loss)
print('expected loss: ', expected_loss)
print('difference: ', abs(loss - expected_loss))
```

```
loss:  9.832355910027387
expected loss:  9.83235591003
difference:  2.6130209107577684e-12
```

Run the following cell to perform numeric gradient checking on the `CaptioningRNN` class; you should see errors around the order of `e-6` or less.

[15]:
```
np.random.seed(231)

batch_size = 2
timesteps = 3
input_dim = 4
wordvec_dim = 5
hidden_dim = 6
word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
vocab_size = len(word_to_idx)

captions = np.random.randint(vocab_size, size=(batch_size, timesteps))
features = np.random.randn(batch_size, input_dim)

model = CaptioningRNN(
    word_to_idx,
    input_dim=input_dim,
    wordvec_dim=wordvec_dim,
    hidden_dim=hidden_dim,
```

```
    cell_type='rnn',
    dtype=np.float64,
)

loss, grads = model.loss(features, captions)

for param_name in sorted(grads):
    f = lambda _: model.loss(features, captions)[0]
    param_grad_num = eval_numerical_gradient(f, model.params[param_name],␣
 ↪verbose=False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('%s relative error: %e' % (param_name, e))
```

```
W_embed relative error: 2.331071e-09
W_proj relative error: 9.974425e-09
W_vocab relative error: 4.274378e-09
Wh relative error: 1.313259e-08
Wx relative error: 8.455229e-07
b relative error: 9.727212e-10
b_proj relative error: 1.934807e-08
b_vocab relative error: 7.087090e-11
```

13 Overfit RNN Captioning Model on Small Data

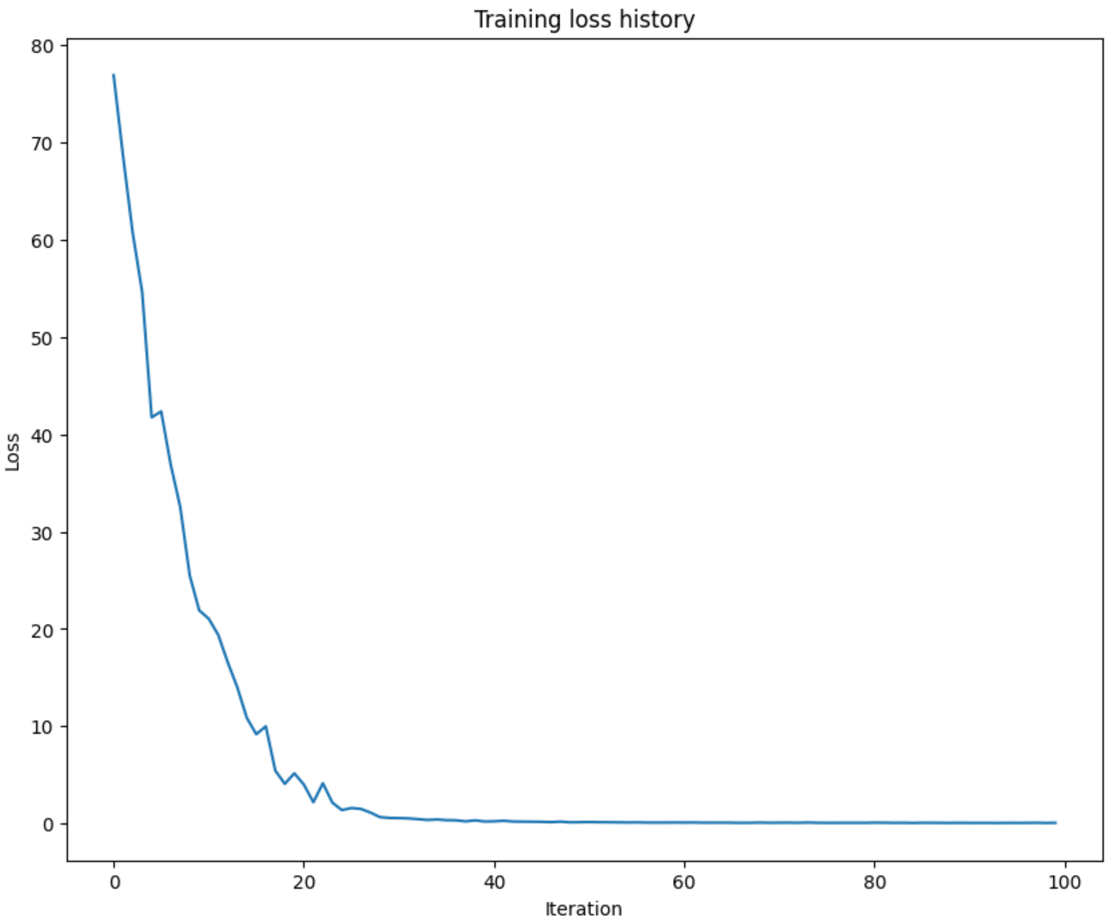
Similar to the `Solver` class that we used to train image classification models on the previous assignment, on this assignment we use a `CaptioningSolver` class to train image captioning models. Open the file `cs231n/captioning_solver.py` and read through the `CaptioningSolver` class; it should look very familiar.

Once you have familiarized yourself with the API, run the following to make sure your model overfits a small sample of 100 training examples. You should see a final loss of less than 0.1.

[16]:
```
np.random.seed(231)

small_data = load_coco_data(max_train=50)

small_rnn_model = CaptioningRNN(
    cell_type='rnn',
    word_to_idx=data['word_to_idx'],
    input_dim=data['train_features'].shape[1],
    hidden_dim=512,
    wordvec_dim=256,
)

small_rnn_solver = CaptioningSolver(
    small_rnn_model, small_data,
    update_rule='adam',
```

```
    num_epochs=50,
    batch_size=25,
    optim_config={
     'learning_rate': 5e-3,
    },
    lr_decay=0.95,
    verbose=True, print_every=10,
)

small_rnn_solver.train()

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

```
base dir  /content/drive/My
Drive/COMP4471/assignments/assignment3/cs231n/datasets/coco_captioning
(Iteration 1 / 100) loss: 76.913487
(Iteration 11 / 100) loss: 21.062620
(Iteration 21 / 100) loss: 4.016257
(Iteration 31 / 100) loss: 0.567198
(Iteration 41 / 100) loss: 0.239397
(Iteration 51 / 100) loss: 0.161970
(Iteration 61 / 100) loss: 0.111524
(Iteration 71 / 100) loss: 0.097583
(Iteration 81 / 100) loss: 0.099063
(Iteration 91 / 100) loss: 0.073966
```



Print final training loss. You should see a final loss of less than 0.1.

[17]:
```
print('Final loss: ', small_rnn_solver.loss_history[-1])
```

```
Final loss:  0.08206947465317423
```

14 RNN Sampling at Test Time

Unlike classification models, image captioning models behave very differently at training time vs. at test time. At training time, we have access to the ground-truth caption, so we feed ground-truth words as input to the RNN at each timestep. At test time, we sample from the distribution over the vocabulary at each timestep and feed the sample as input to the RNN at the next timestep.

In the file `cs231n/classifiers/rnn.py`, implement the `sample` method for test-time sampling. After doing so, run the following to sample from your overfitted model on both training and validation data. The samples on training data should be very good. The samples on validation data, however, probably won't make sense.

[18]:
```
# 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(small_data, split=split, batch_size=2)
    gt_captions, features, urls = minibatch
    gt_captions = decode_captions(gt_captions, data['idx_to_word'])

    sample_captions = small_rnn_model.sample(features)
    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()
```

```
Output hidden; open in https://colab.research.google.com to view.
```

15 Inline Question 1

In our current image captioning setup, our RNN language model produces a word at every timestep as its output. However, an alternate way to pose the problem is to train the network to operate over *characters* (e.g. 'a', 'b', etc.) as opposed to words, so that at it every timestep, it receives the previous character as input and tries to predict the next character in the sequence. For example, the network might generate a caption like

'A', ' ', 'c', 'a', 't', ' ', 'o', 'n', ' ', 'a', ' ', 'b', 'e', 'd'

Can you describe one advantage of an image-captioning model that uses a character-level RNN? Can you also describe one disadvantage? HINT: there are several valid answers, but it might be useful to compare the parameter space of word-level and character-level models.

Your Answer: An advantage is that the vocabulary size becomes much smaller which significantly reduces the parameter count in the embedding and output layers. This makes the model more memory-efficient and easier to train, especially for words not in the vocabulary since characters can be combined to form any word.

A disadvantage is that that sequences become much longer because words are split into individual characters. This forces the RNN to learn longer-range dependencies and thus slows training/inference and makes it harder to capture high-level meanings.

[ ]:
```
```

Transformer_Captioning

April 14, 2025

[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. 'cs231n/assignments/assignment3/'
FOLDERNAME = 'COMP4471/assignments/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/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/COMP4471/assignments/assignment3/cs231n/datasets
/content/drive/My Drive/COMP4471/assignments/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 cs231n.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs231n.transformer_layers import *
from cs231n.captioning_solver_transformer import CaptioningSolverTransformer
from cs231n.classifiers.transformer import CaptioningTransformer
from cs231n.coco_utils import load_coco_data, sample_coco_minibatch,␣
 ↪decode_captions
from cs231n.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/My
Drive/COMP4471/assignments/assignment3/cs231n/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 $\{v_1, \dots, v_n\}, v_i \in \mathbb{R}^d$, and a set of key vectors $\{k_1, \dots, k_n\}, k_i \in \mathbb{R}^d$, 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:

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

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:

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

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

Implement multi-headed scaled dot-product attention in the `MultiHeadAttention` class in the file `cs231n/transformer_layers.py`. The code below will check your implementation. The relative error should be less than `e-3`.

[4]:
```
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 `cs231n/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.

[5]:
```
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. Allows the model to focus on different types of information in parallel. Without multiple heads, it would be a lot more difficult to capture all dependencies as all patterns would compete for the same representation space.

2. Dividing creates numerical stability by counteracting the effect of vanishing gradients caused by dot products growing large in magnitude (which would hinder learning).

3. The linear transformation layer is used to mix ouput information among heads in order to allow for a better representation of feature patterns.

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 `cs231n/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.

[6]:
```
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
```

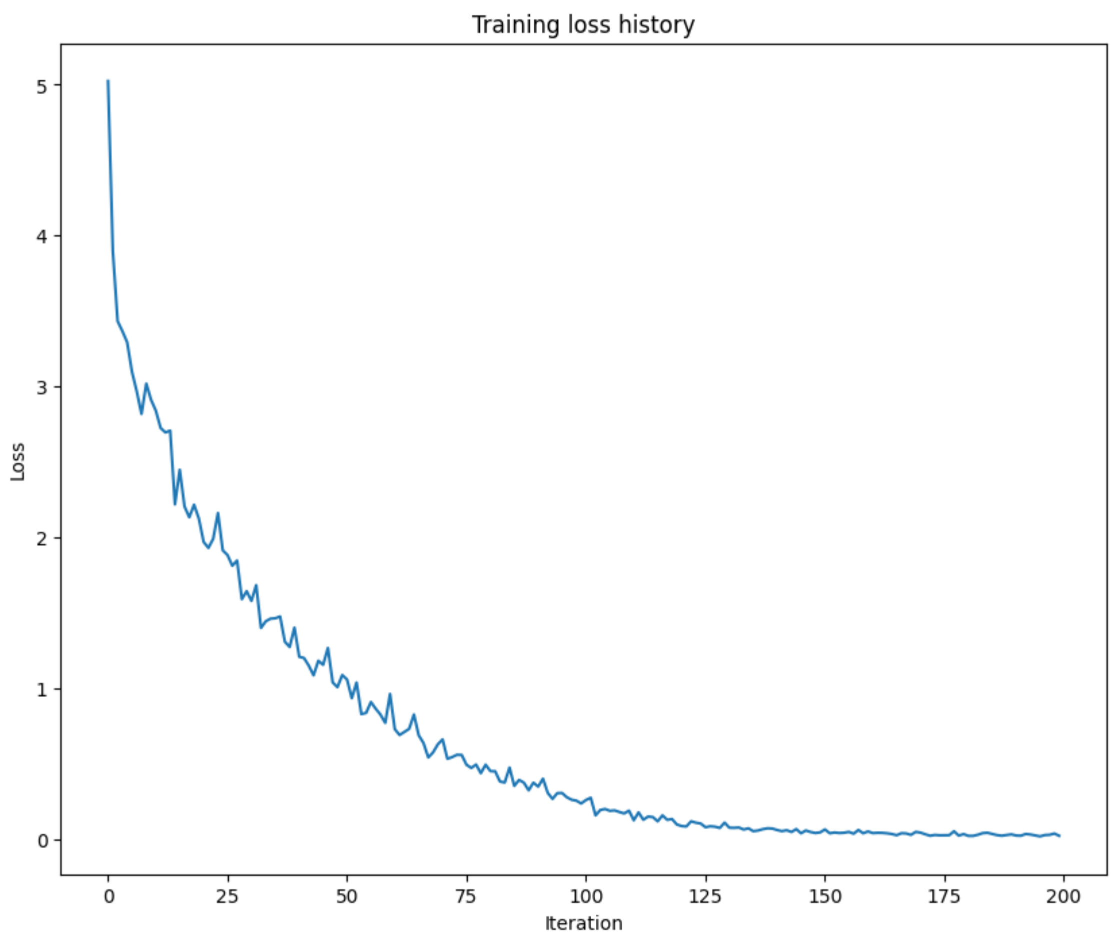
8 Overfit Transformer Captioning Model on Small Data

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

[7]:
```
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/My
Drive/COMP4471/assignments/assignment3/cs231n/datasets/coco_captioning
(Iteration 1 / 200) loss: 5.023862
(Iteration 11 / 200) loss: 2.838942
(Iteration 21 / 200) loss: 1.969214
(Iteration 31 / 200) loss: 1.578393
(Iteration 41 / 200) loss: 1.207860
(Iteration 51 / 200) loss: 1.057564
(Iteration 61 / 200) loss: 0.728179
(Iteration 71 / 200) loss: 0.660102
(Iteration 81 / 200) loss: 0.450094
```

```
(Iteration 91 / 200) loss: 0.347460
(Iteration 101 / 200) loss: 0.258647
(Iteration 111 / 200) loss: 0.124202
(Iteration 121 / 200) loss: 0.085938
(Iteration 131 / 200) loss: 0.075018
(Iteration 141 / 200) loss: 0.059534
(Iteration 151 / 200) loss: 0.063559
(Iteration 161 / 200) loss: 0.039298
(Iteration 171 / 200) loss: 0.042274
(Iteration 181 / 200) loss: 0.020821
(Iteration 191 / 200) loss: 0.023106
```



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

[8]:
```python
print('Final loss: ', transformer_solver.loss_history[-1])
```

```
Final loss:  0.02125324
```

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.

[9]:
```
# 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 bottles and a in it <END>

GT:<START> a bedroom with a bed desk and <UNK> <UNK> <END>

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

[ ]:

Generative_Adversarial_Networks

April 14, 2025

[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. 'cs231n/assignments/assignment3/'
FOLDERNAME = 'COMP4471/assignments/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/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/COMP4471/assignments/assignment3/cs231n/datasets
/content/drive/My Drive/COMP4471/assignments/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 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:

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

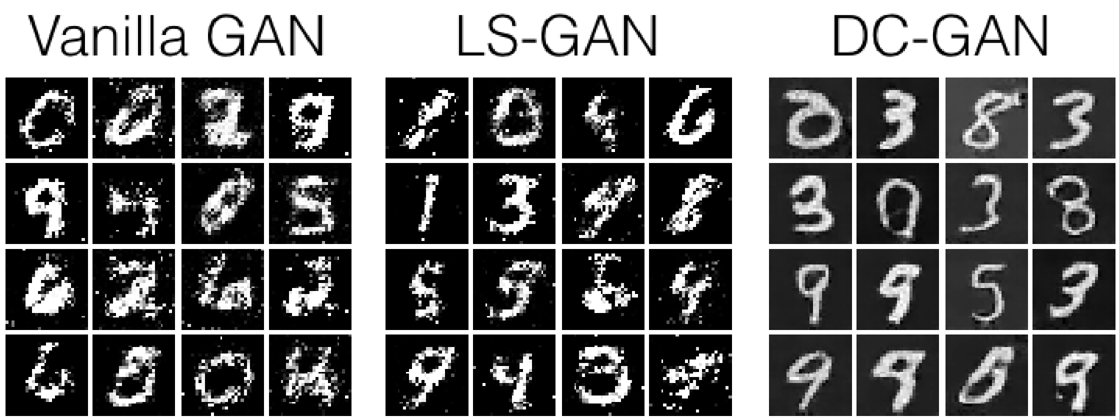
2. Update the discriminator (D), to maximize the probability of the discriminator making the correct choice on real and generated data:

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

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:

[2]:
```python
# Run this cell to see sample outputs.
from IPython.display import Image
Image('images/gan_outputs_pytorch.png')
```

[2]:



[2]:
```python
# 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 cs231n.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
```

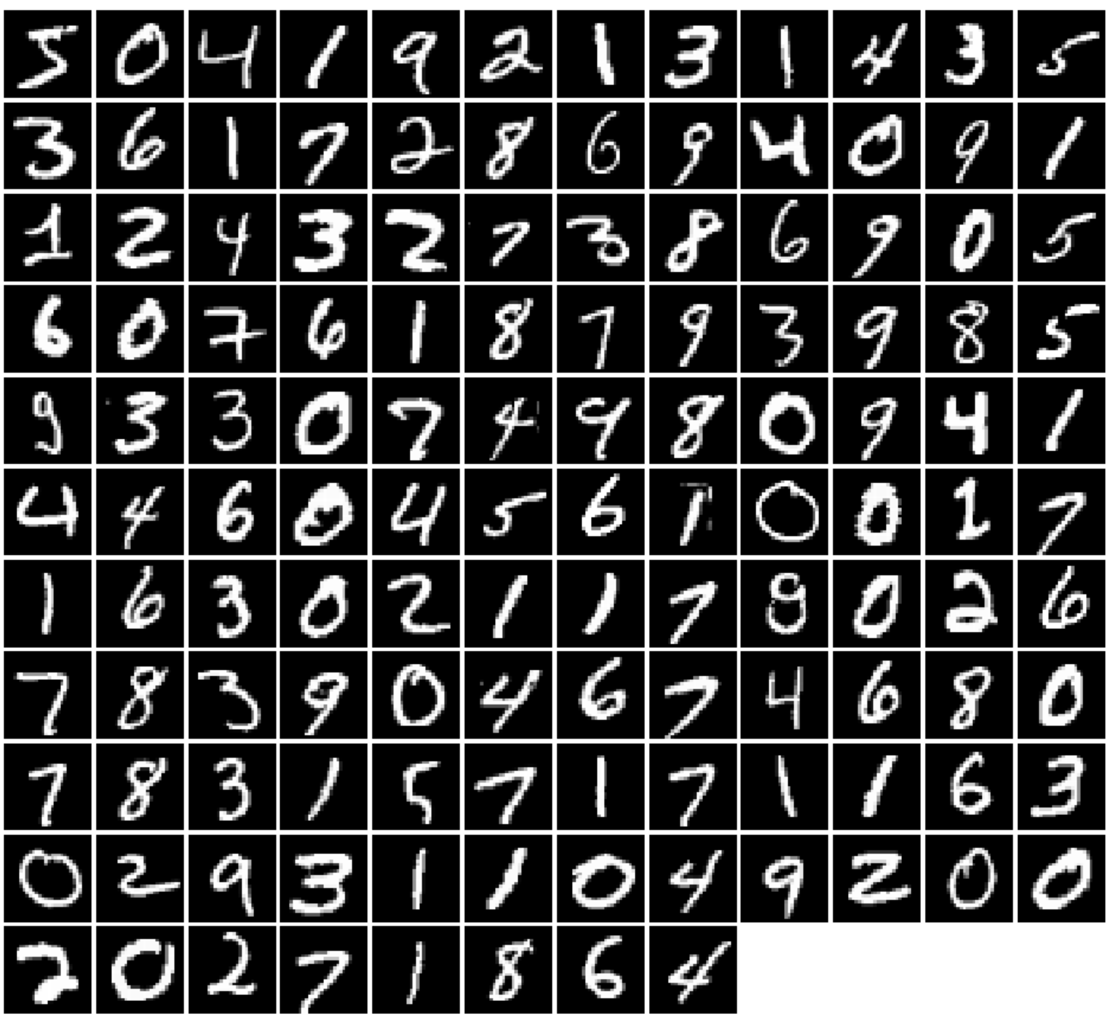
1.1 Dataset

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`.

[3]:
```
NUM_TRAIN = 50000
NUM_VAL = 5000

NOISE_DIM = 96
batch_size = 128

mnist_train = dset.MNIST(
    './cs231n/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(
    './cs231n/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)
```



1.2 Random Noise

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

Implement `sample_noise` in `cs231n/gan_pytorch.py`.

Hint: use `torch.rand`.

Make sure noise is the correct shape and type:

[5]:
```python
from cs231n.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.

[7]:
```python
from cs231n.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 cs231n/gan_pytorch.py

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

[12]:
```
from cs231n.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 cs231n/gan_pytorch.py

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

[14]:
```
from cs231n.gan_pytorch import generator

def test_generator(true_count=1858320):
    model = 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_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 $\log \left(1 - D(G(z)) \right)$, 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 `cs231n/gan_pytorch.py`

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

[16]:
```
from cs231n.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))

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

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

[17]:
```
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 `cs231n/gan_pytorch.py`

6 Training a GAN!

We provide you the main training loop. You won't need to change `run_a_gan` in `cs231n/gan_pytorch.py`, but we encourage you to read through it for your own understanding.

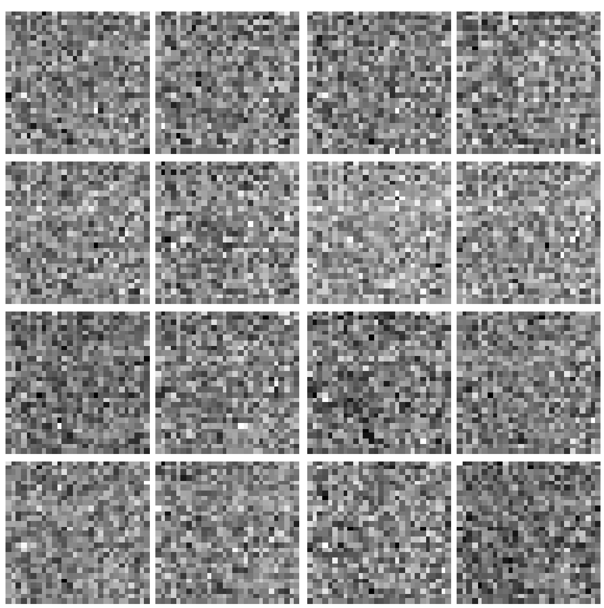
[18]:
```
from cs231n.gan_pytorch import get_optimizer, run_a_gan

# 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: 1.377, G:0.6985
Iter: 250, D: 1.047, G:1.039
Iter: 500, D: 0.9343, G:0.9997
Iter: 750, D: 1.368, G:0.9887
Iter: 1000, D: 1.113, G:1.067
Iter: 1250, D: 1.376, G:0.8773
Iter: 1500, D: 1.166, G:0.8978
Iter: 1750, D: 1.303, G:0.9913
Iter: 2000, D: 1.193, G:0.8939
Iter: 2250, D: 1.354, G:0.8321
Iter: 2500, D: 1.249, G:0.8264
Iter: 2750, D: 1.332, G:0.8083
Iter: 3000, D: 1.313, G:0.8032
Iter: 3250, D: 1.409, G:0.8666
Iter: 3500, D: 1.35, G:0.8036
Iter: 3750, D: 1.324, G:0.7617
```

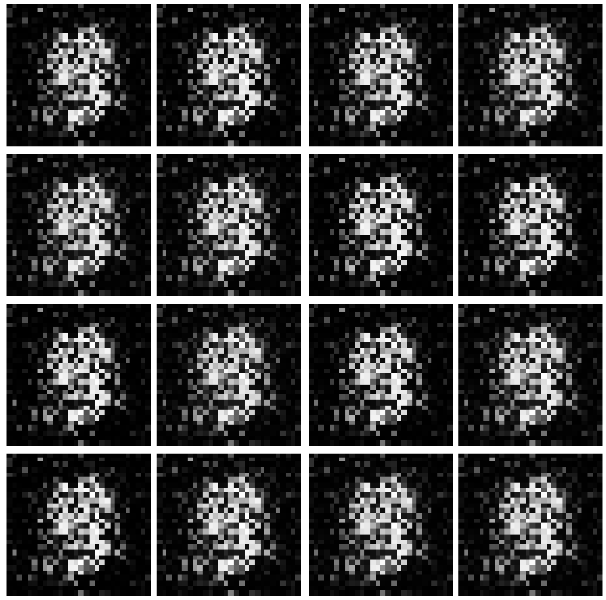
Run the cell below to show the generated images.

[12]:
```
numIter = 0
for img in images:
    print("Iter: {}".format(numIter))
    show_images(img)
    plt.show()
    numIter += 250
    print()
```

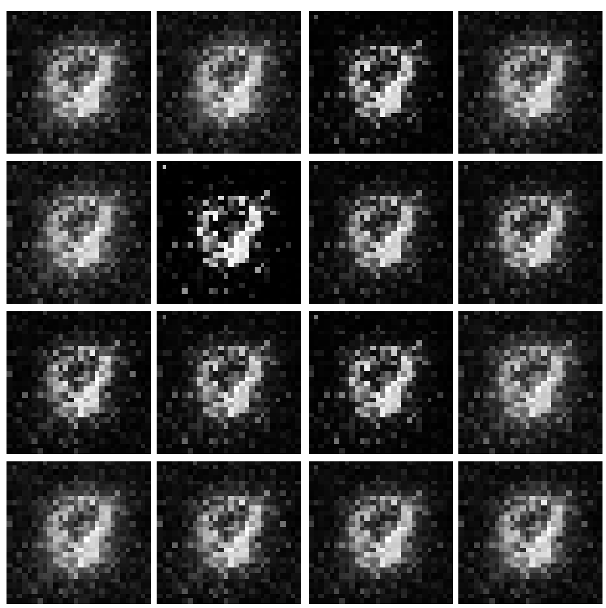
```
Iter: 0
```



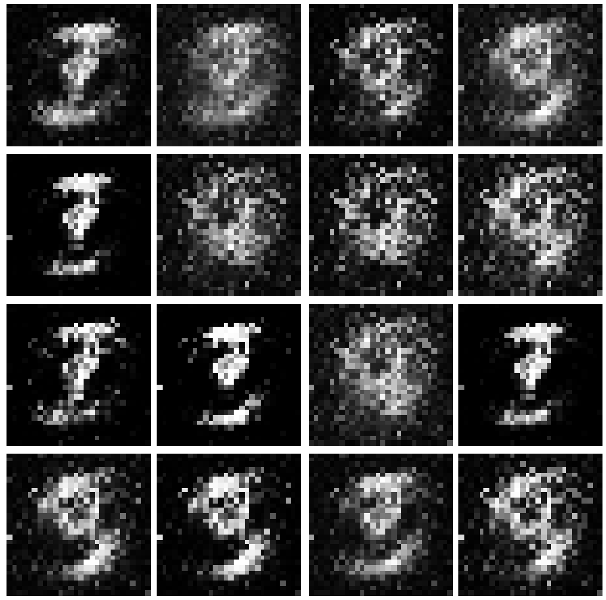
```
Iter: 250
```



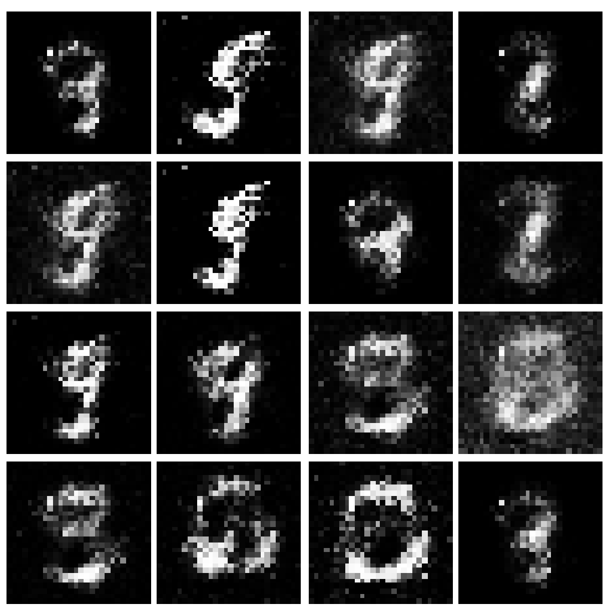
```
Iter: 500
```



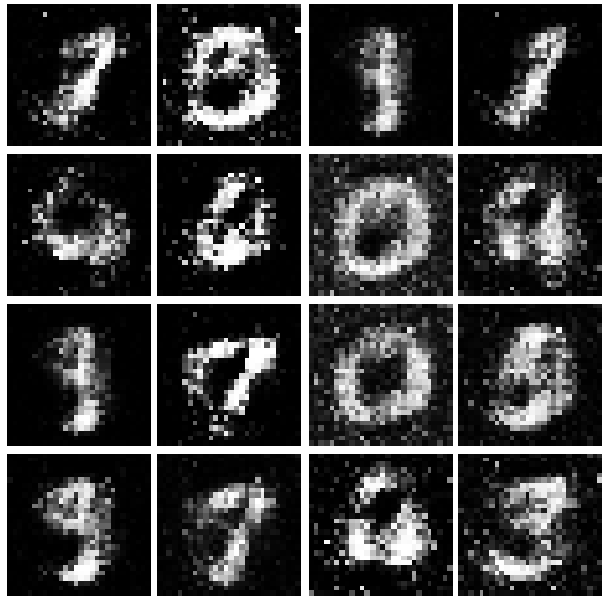
Iter: 750



Iter: 1000

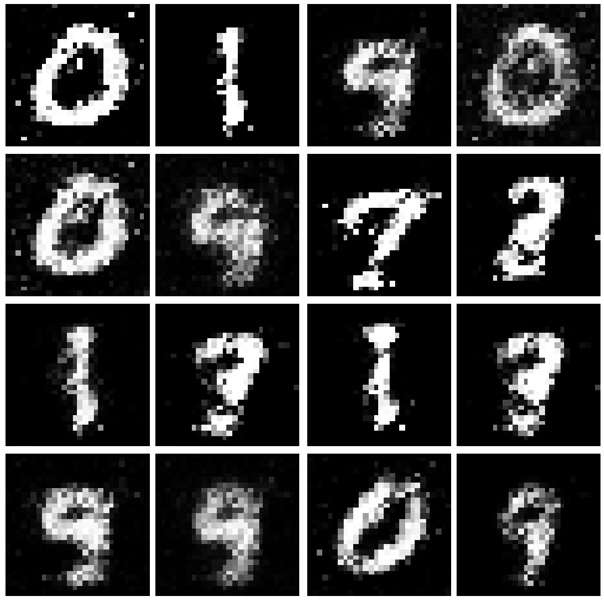


Iter: 1250

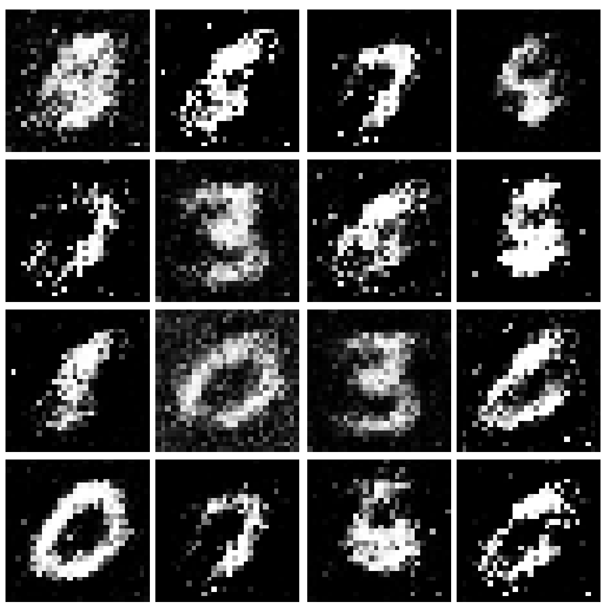


Iter: 1500

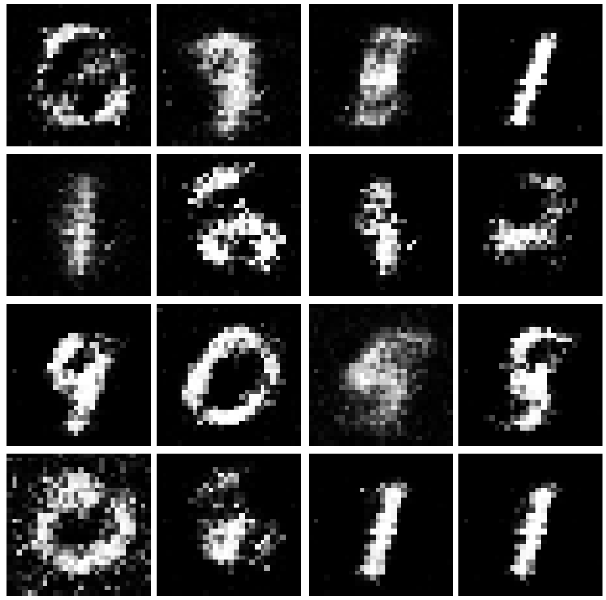
Iter: 1750



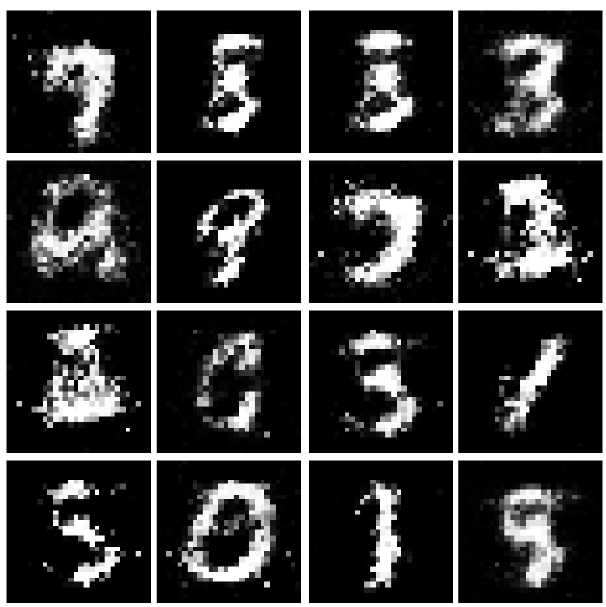
Iter: 2000



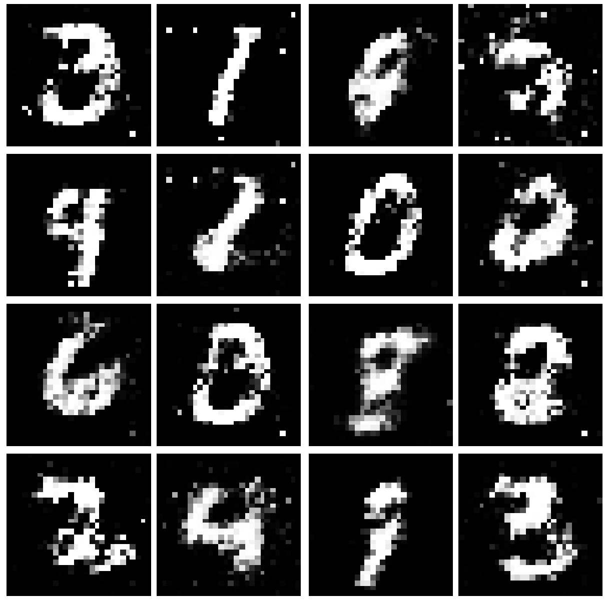
Iter: 2250



Iter: 2500

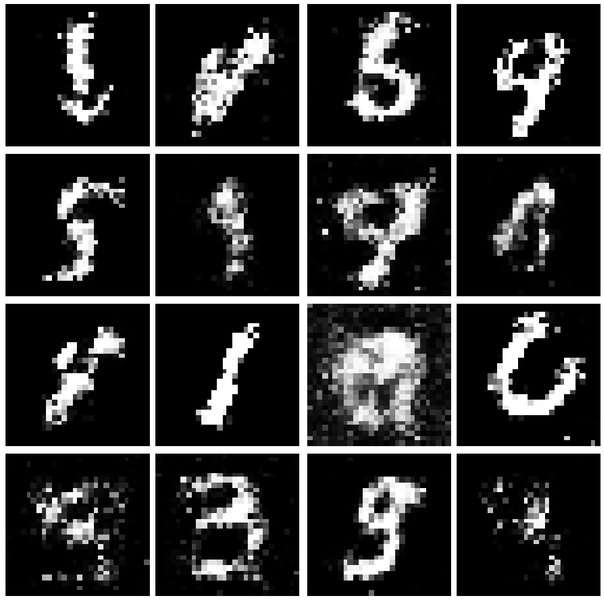


Iter: 2750

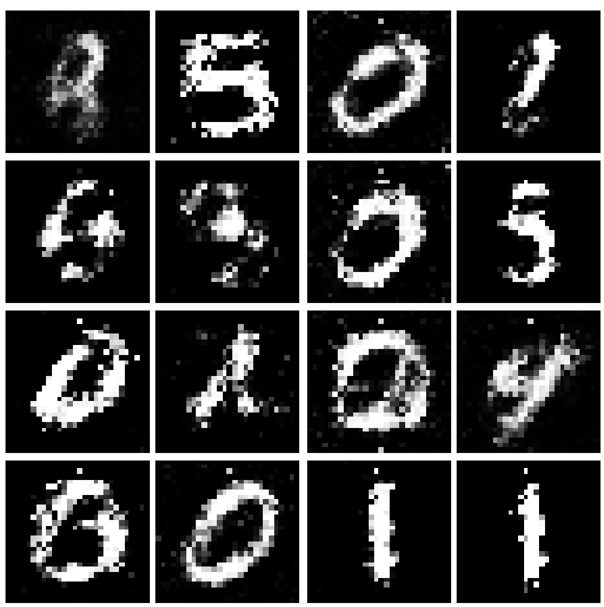


Iter: 3000

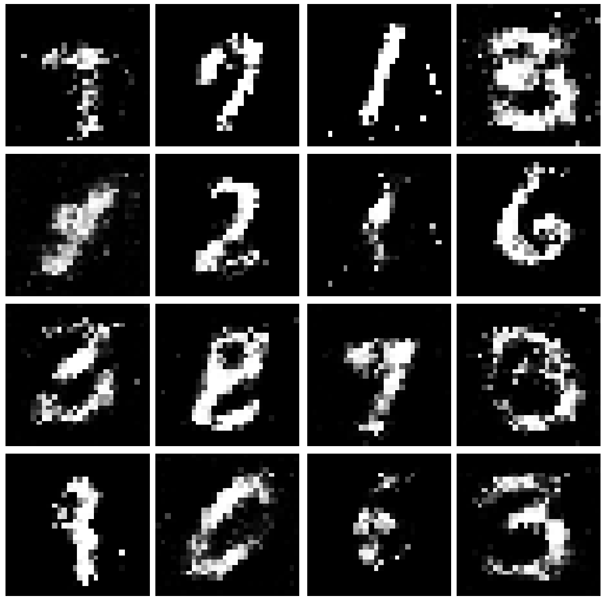
Iter: 3250



Iter: 3500



Iter: 3750



6.1 Inline Question 1

What does your final vanilla GAN image look like?

[13]:
```
# 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:

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

and the discriminator loss:

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

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 `cs231n/gan_pytorch.py`

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

[14]:
```python
from cs231n.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!

[15]:
```
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: 0.5689, G:0.51
Iter: 250, D: 0.0663, G:0.53
Iter: 500, D: 0.135, G:0.338
Iter: 750, D: 0.377, G:0.4284
Iter: 1000, D: 0.1298, G:0.3204
Iter: 1250, D: 0.1435, G:0.3777
Iter: 1500, D: 0.1488, G:0.2437
Iter: 1750, D: 0.189, G:0.1811
Iter: 2000, D: 0.2077, G:0.3095
Iter: 2250, D: 0.2262, G:0.1409
Iter: 2500, D: 0.2009, G:0.2069
Iter: 2750, D: 0.2329, G:0.2056
Iter: 3000, D: 0.2196, G:0.1726
Iter: 3250, D: 0.2347, G:0.1353
```

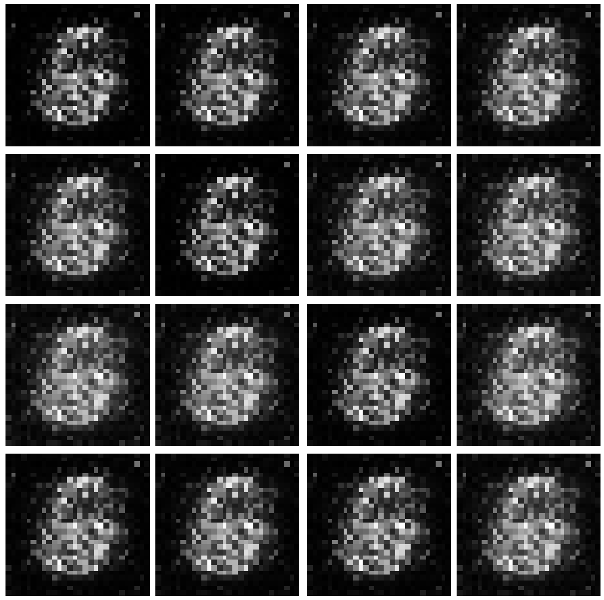
```
Iter: 3500, D: 0.207, G:0.1815
Iter: 3750, D: 0.2229, G:0.1625
```

Run the cell below to show generated images.

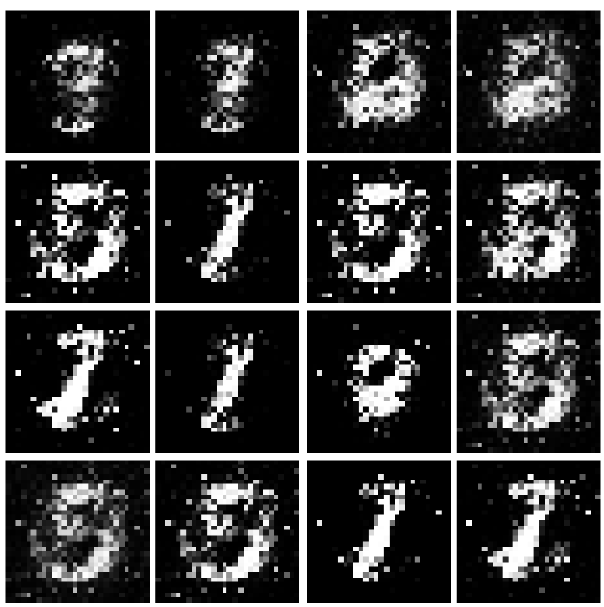
[16]:
```
numIter = 0
for img in images:
    print("Iter: {}".format(numIter))
    show_images(img)
    plt.show()
    numIter += 250
    print()
```

```
Iter: 0
```

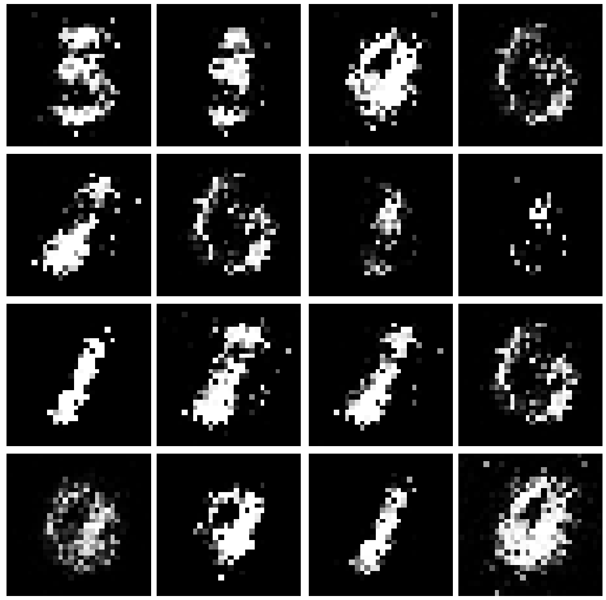
```
Iter: 250
```



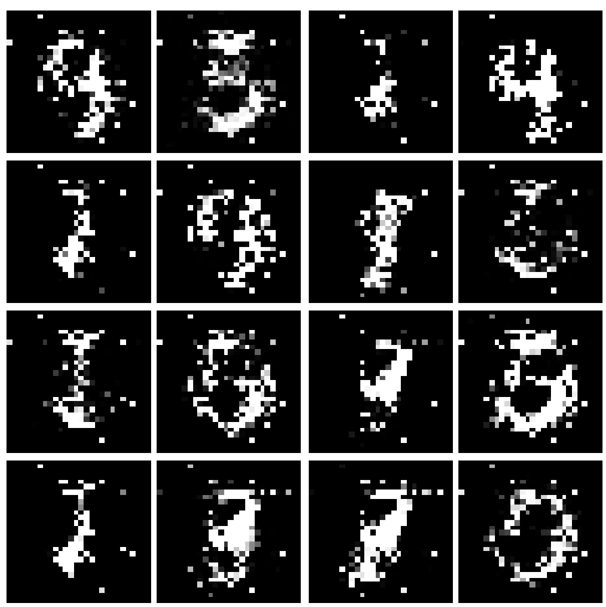
Iter: 500



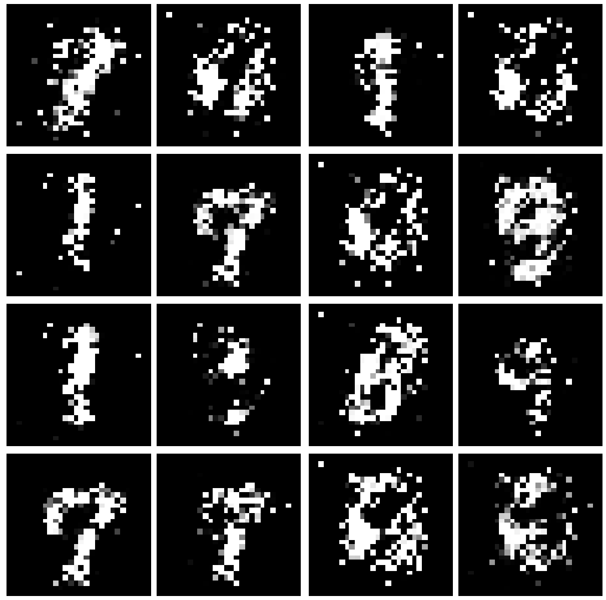
Iter: 750



Iter: 1000

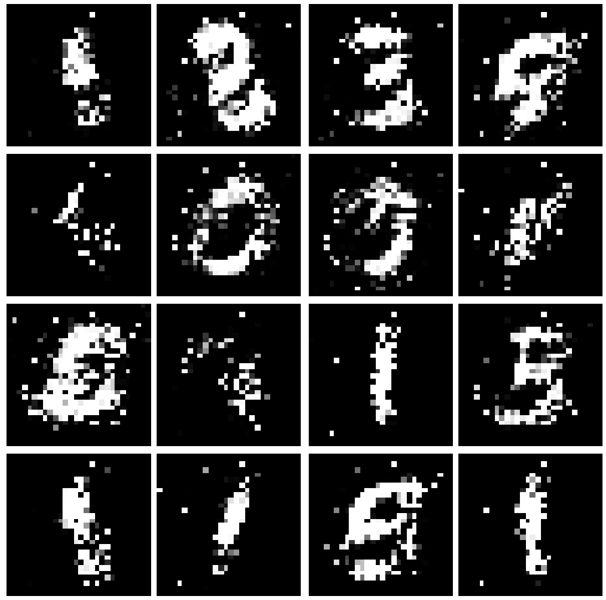


Iter: 1250

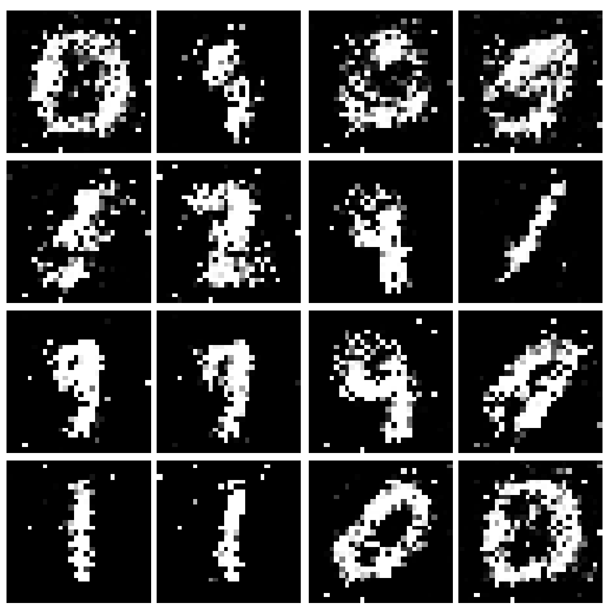


Iter: 1500

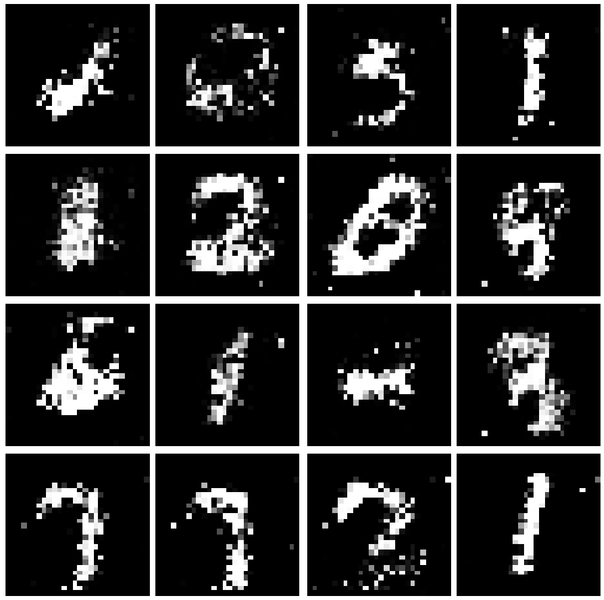
Iter: 1750



Iter: 2000

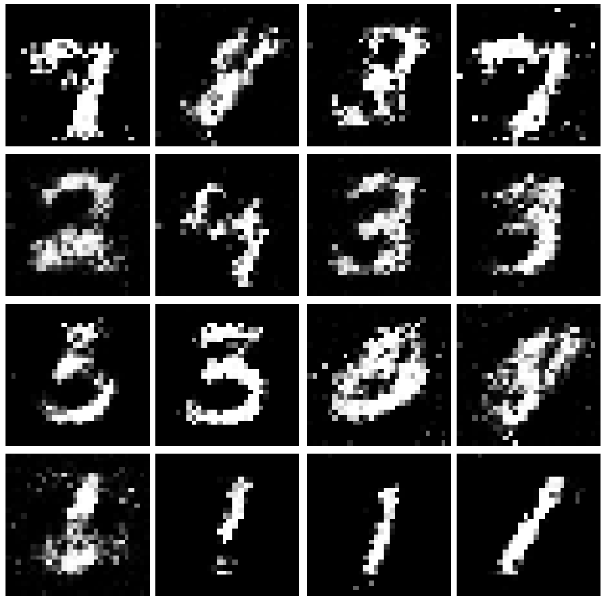


Iter: 2250



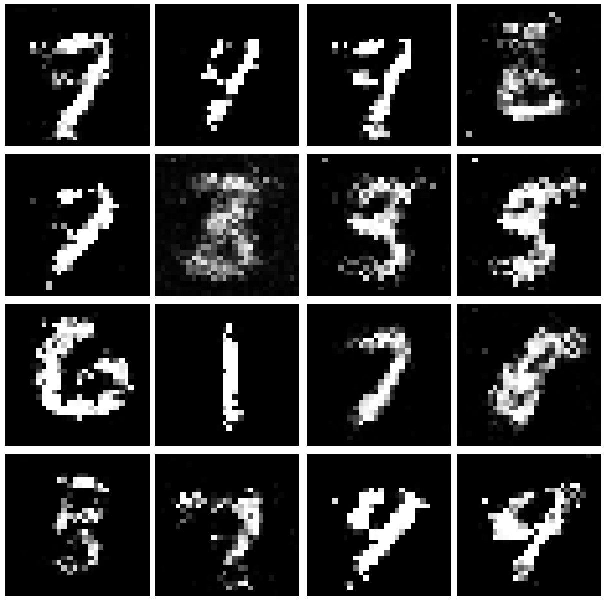
Iter: 2500

Iter: 2750

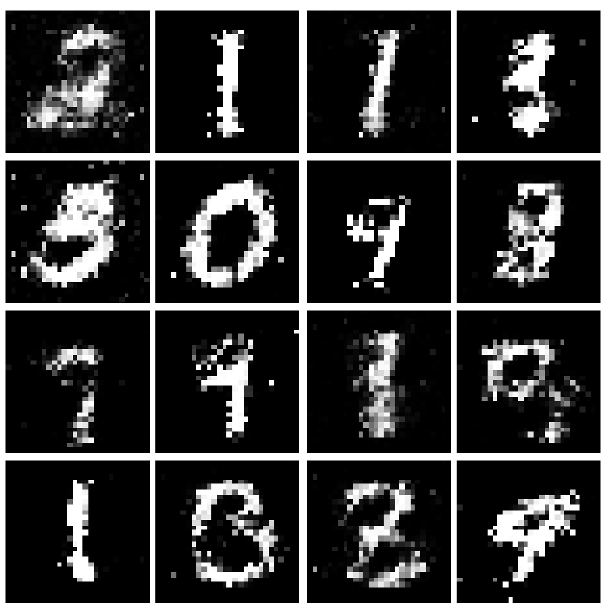


Iter: 3000

Iter: 3250



Iter: 3500



Iter: 3750

7.1 Inline Question 2

What does your final LSGAN image look like?

[17]:
```
# 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 `cs231n/gan_pytorch.py`

[4]:
```python
from cs231n.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:

[5]:
```
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 `cs231n/gan_pytorch.py`

[8]:
```
from cs231n.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()
```

[8]:
```
torch.Size([128, 784])
```

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

[9]:
```
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.
```

[19]:
```
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
)
```

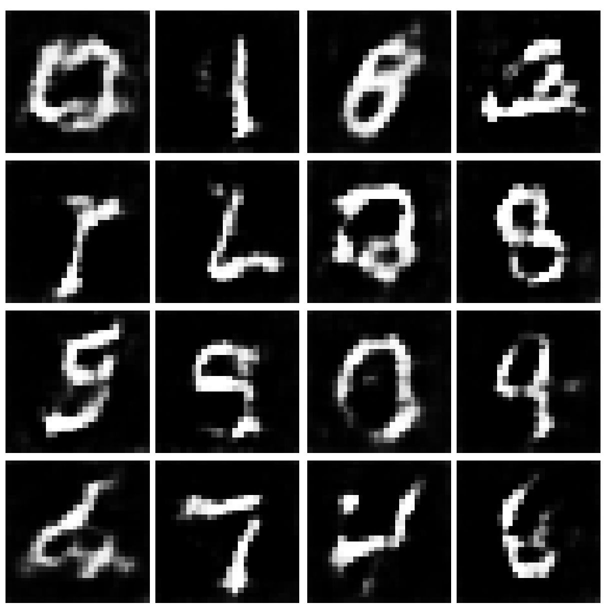
```
Iter: 0, D: 1.373, G:0.8811
Iter: 250, D: 1.251, G:0.775
Iter: 500, D: 1.168, G:1.033
Iter: 750, D: 1.226, G:1.053
Iter: 1000, D: 1.299, G:1.032
Iter: 1250, D: 1.291, G:0.9126
Iter: 1500, D: 1.132, G:1.079
Iter: 1750, D: 1.351, G:0.7687
```

Run the cell below to show generated images.

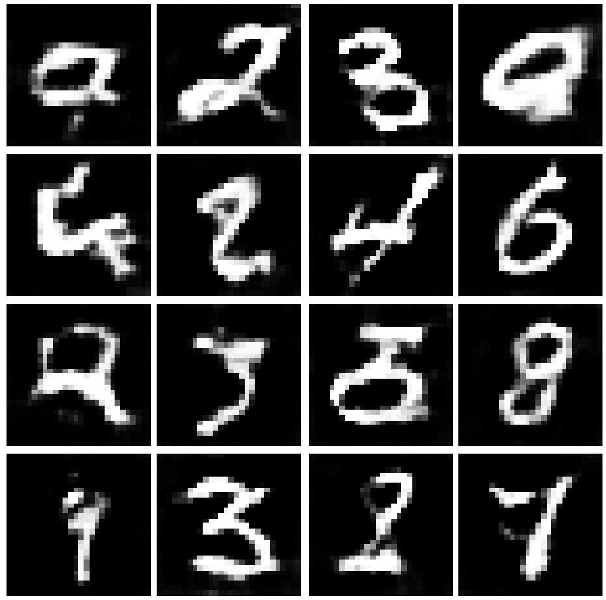
[20]:
```
numIter = 0
for img in images:
    print("Iter: {}".format(numIter))
    show_images(img)
    plt.show()
    numIter += 250
    print()
```

```
Iter: 0
```

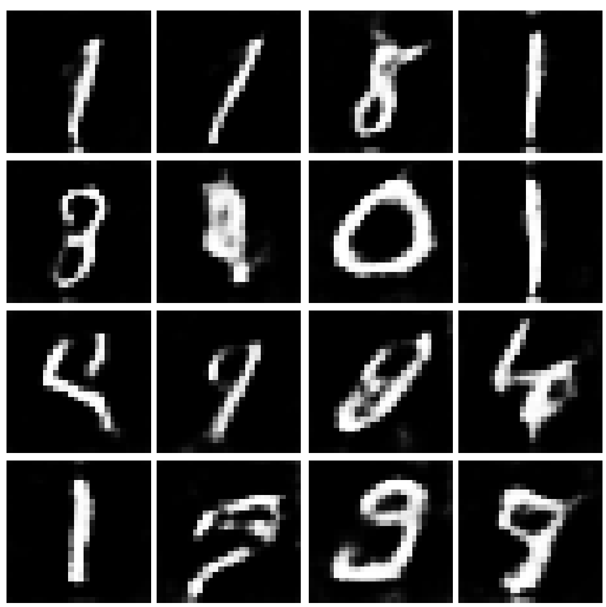
Iter: 250



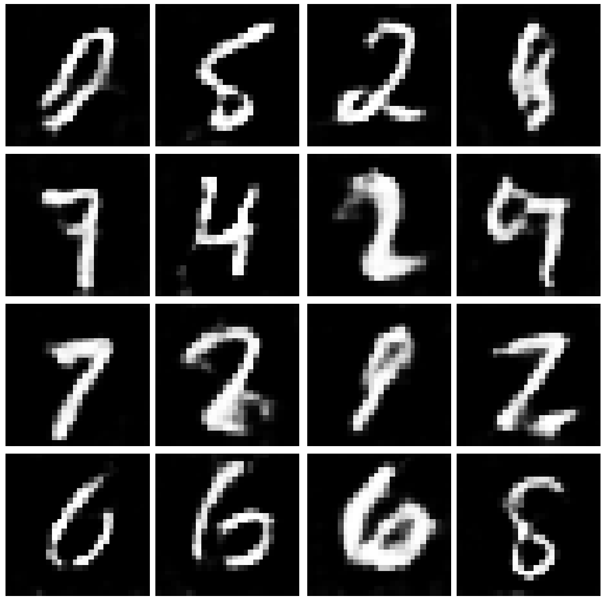
Iter: 500



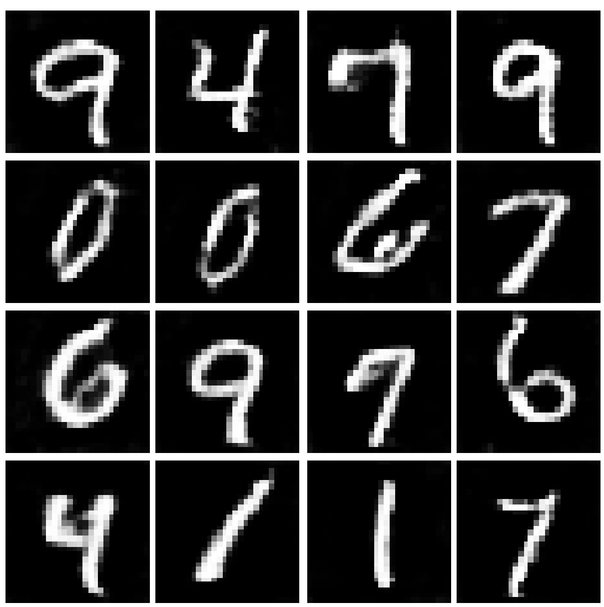
Iter: 750



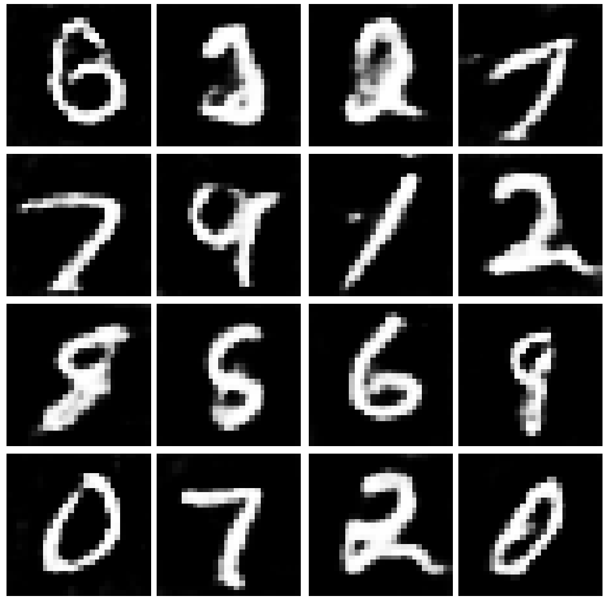
Iter: 1000



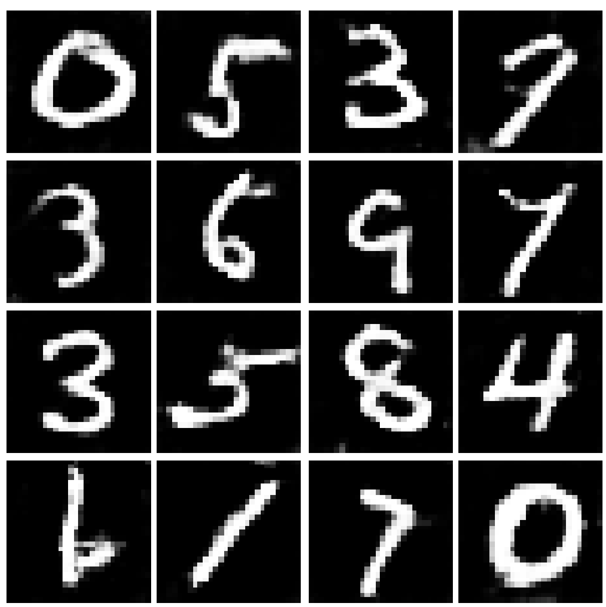
Iter: 1250



Iter: 1500



Iter: 1750

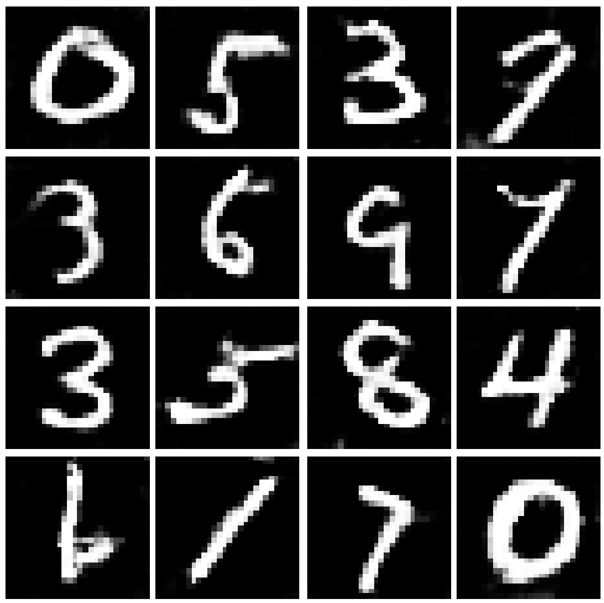


8.1 Inline Question 3

What does your final DCGAN image look like?

[21]:
```
# 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 the optimal value will never be reached because the variables oscillate indefinitely without converging, as seen in the table where we get (1,1) again after 6 steps, so the same pattern will continue.

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:

No. If the discriminator loss stays constantly high, it means the discriminator is failing to learn meaningful distinctions between real and fake data. The discriminator should improve over time by having its loss decrease. Also, a decreasing generator loss suggests the generator is finding ways to fool the discriminator that is not learning. This indicates the generator is not actually improving and that it is just exploiting a weak discriminator.

[ ]:

```

```

Self_Supervised_Learning

April 14, 2025

[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. 'cs231n/assignments/assignment3/'
FOLDERNAME = 'COMP4471/assignments/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/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/COMP4471/assignments/assignment3/cs231n/datasets
/content/drive/My Drive/COMP4471/assignments/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 Self-Supervised Learning

1.1 What is self-supervised learning?

Modern day machine learning requires lots of labeled data. But often times it's challenging and/or expensive to obtain large amounts of human-labeled data. Is there a way we could ask machines to automatically learn a model which can generate good visual representations without a labeled dataset? Yes, enter self-supervised learning!

Self-supervised learning (SSL) allows models to automatically learn a “good” representation space using the data in a given dataset without the need for their labels. Specifically, if our dataset were a bunch of images, then self-supervised learning allows a model to learn and generate a “good” representation vector for images.

The reason SSL methods have seen a surge in popularity is because the learnt model continues to perform well on other datasets as well i.e. new datasets on which the model was not trained on!

1.2 What makes a “good” representation?

A “good” representation vector needs to capture the important features of the image as it relates to the rest of the dataset. This means that images in the dataset representing semantically similar entities should have similar representation vectors, and different images in the dataset should have different representation vectors. For example, two images of an apple should have similar representation vectors, while an image of an apple and an image of a banana should have different representation vectors.

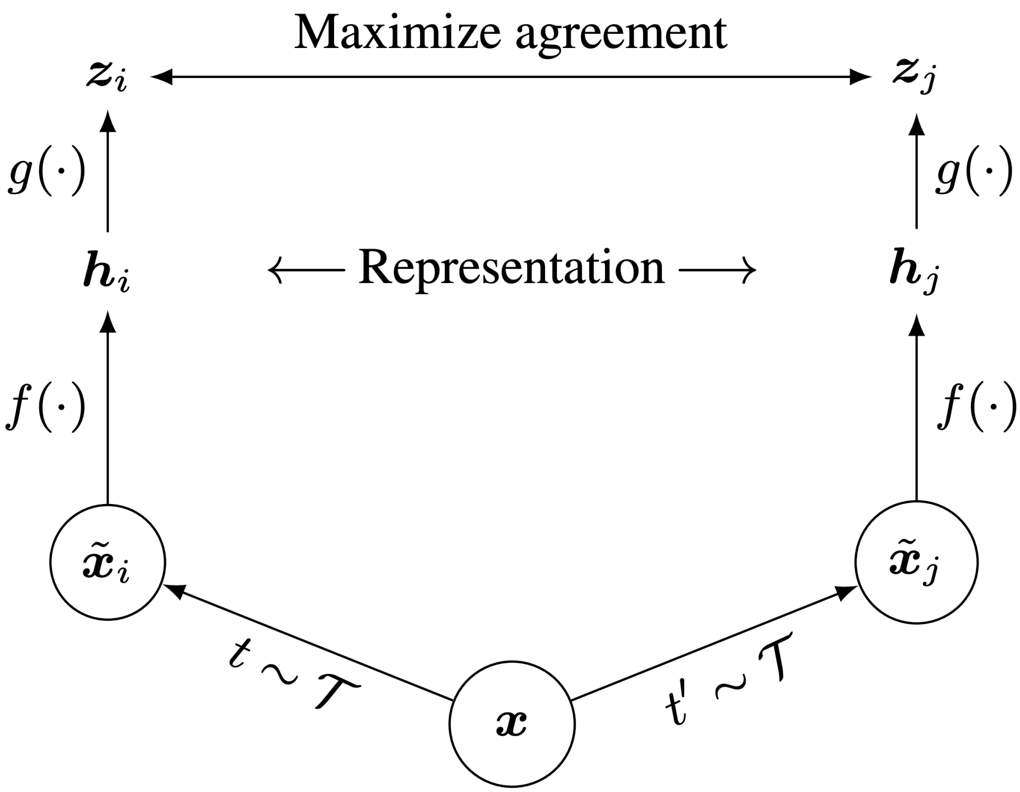
1.3 Contrastive Learning: SimCLR

Recently, SimCLR introduces a new architecture which uses **contrastive learning** to learn good visual representations. Contrastive learning aims to learn similar representations for similar images and different representations for different images. As we will see in this notebook, this simple idea allows us to train a surprisingly good model without using any labels.

Specifically, for each image in the dataset, SimCLR generates two differently augmented views of that image, called a **positive pair**. Then, the model is encouraged to generate similar representation vectors for this pair of images. See below for an illustration of the architecture (Figure 2 from the paper).

[2]:
```
# Run this cell to view the SimCLR architecture.
from IPython.display import Image
Image('images/simclr_fig2.png', width=500)
```

[2]:



Given an image **x**, SimCLR uses two different data augmentation schemes **t** and **t'** to generate the positive pair of images $\tilde{x}_i$ and $\tilde{x}_j$. f is a basic encoder net that extracts representation vectors from the augmented data samples, which yields h_i and h_j, respectively. Finally, a small neural network projection head g maps the representation vectors to the space where the contrastive loss is applied. The goal of the contrastive loss is to maximize agreement between the final vectors $z_i = g(h_i)$ and $z_j = g(h_j)$. We will discuss the contrastive loss in more detail later, and you will get to implement it.

After training is completed, we throw away the projection head g and only use f and the representation h to perform downstream tasks, such as classification. You will get a chance to finetune a layer on top of a trained SimCLR model for a classification task and compare its performance with a baseline model (without self-supervised learning).

1.4 Pretrained Weights

For your convenience, we have given you pretrained weights (trained for ~18 hours on CIFAR-10) for the SimCLR model. Run the following cell to download pretrained model weights to be used later. (This will take ~1 minute)

[3]:
```
%%bash
DIR=pretrained_model/
```

```
if [ ! -d "$DIR" ]; then
    mkdir "$DIR"
fi

URL=http://downloads.cs.stanford.edu/downloads/cs231n/pretrained_simclr_model.
 ↪pth
FILE=pretrained_model/pretrained_simclr_model.pth
if [ ! -f "$FILE" ]; then
    echo "Downloading weights..."
    wget "$URL" -O "$FILE"
fi
```

[4]:
```
# Setup cell.
%pip install thop
import torch
import os
import importlib
import pandas as pd
import numpy as np
import torch.optim as optim
import torch.nn as nn
import random
from thop import profile, clever_format
from torch.utils.data import DataLoader
from torchvision.datasets import CIFAR10
import matplotlib.pyplot as plt
%matplotlib inline

%load_ext autoreload
%autoreload 2

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

```
Collecting thop
  Downloading thop-0.1.1.post2209072238-py3-none-any.whl.metadata (2.7 kB)
Requirement already satisfied: torch in /usr/local/lib/python3.11/dist-packages
(from thop) (2.6.0+cu124)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-
packages (from torch->thop) (3.18.0)
Requirement already satisfied: typing-extensions>=4.10.0 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (4.13.1)
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-
packages (from torch->thop) (3.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages
(from torch->thop) (3.1.6)
Requirement already satisfied: fsspec in /usr/local/lib/python3.11/dist-packages
(from torch->thop) (2025.3.2)
```

```
Collecting nvidia-cuda-nvrtc-cu12==12.4.127 (from torch->thop)
  Downloading nvidia_cuda_nvrtc_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-runtime-cu12==12.4.127 (from torch->thop)
  Downloading nvidia_cuda_runtime_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-cupti-cu12==12.4.127 (from torch->thop)
  Downloading nvidia_cuda_cupti_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cudnn-cu12==9.1.0.70 (from torch->thop)
  Downloading nvidia_cudnn_cu12-9.1.0.70-py3-none-
manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cublas-cu12==12.4.5.8 (from torch->thop)
  Downloading nvidia_cublas_cu12-12.4.5.8-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cufft-cu12==11.2.1.3 (from torch->thop)
  Downloading nvidia_cufft_cu12-11.2.1.3-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-curand-cu12==10.3.5.147 (from torch->thop)
  Downloading nvidia_curand_cu12-10.3.5.147-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cusolver-cu12==11.6.1.9 (from torch->thop)
  Downloading nvidia_cusolver_cu12-11.6.1.9-py3-none-
manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cusparse-cu12==12.3.1.170 (from torch->thop)
  Downloading nvidia_cusparse_cu12-12.3.1.170-py3-none-
manylinux2014_x86_64.whl.metadata (1.6 kB)
Requirement already satisfied: nvidia-cusparselt-cu12==0.6.2 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (0.6.2)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (2.21.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (12.4.127)
Collecting nvidia-nvjitlink-cu12==12.4.127 (from torch->thop)
  Downloading nvidia_nvjitlink_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Requirement already satisfied: triton==3.2.0 in /usr/local/lib/python3.11/dist-
packages (from torch->thop) (3.2.0)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/dist-
packages (from torch->thop) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/usr/local/lib/python3.11/dist-packages (from sympy==1.13.1->torch->thop)
(1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.11/dist-packages (from jinja2->torch->thop) (3.0.2)
Downloading thop-0.1.1.post2209072238-py3-none-any.whl (15 kB)
Downloading nvidia_cublas_cu12-12.4.5.8-py3-none-manylinux2014_x86_64.whl (363.4
MB)
```

```
                         363.4/363.4 MB
4.3 MB/s eta 0:00:00
Downloading nvidia_cuda_cupti_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl (13.8 MB)
                         13.8/13.8 MB
104.0 MB/s eta 0:00:00
Downloading nvidia_cuda_nvrtc_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl (24.6 MB)
                         24.6/24.6 MB
85.6 MB/s eta 0:00:00
Downloading nvidia_cuda_runtime_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl (883 kB)
                         883.7/883.7 kB
55.7 MB/s eta 0:00:00
Downloading nvidia_cudnn_cu12-9.1.0.70-py3-none-manylinux2014_x86_64.whl
(664.8 MB)
                         664.8/664.8 MB
1.1 MB/s eta 0:00:00
WARNING: Retrying (Retry(total=4, connect=None, read=None,

redirect=None, status=None)) after connection broken by

'ProtocolError('Connection aborted.', RemoteDisconnected('Remote end closed

connection without response'))': /packages/27/94/3266821f65b92b3138631e9c8e7fe1f

b513804ac934485a8d05776e1dd43/nvidia_cufft_cu12-11.2.1.3-py3-none-

manylinux2014_x86_64.whl

Downloading nvidia_cufft_cu12-11.2.1.3-py3-none-manylinux2014_x86_64.whl
(211.5 MB)
                         211.5/211.5 MB
6.0 MB/s eta 0:00:00
Downloading nvidia_curand_cu12-10.3.5.147-py3-none-
manylinux2014_x86_64.whl (56.3 MB)
                         56.3/56.3 MB
12.1 MB/s eta 0:00:00
Downloading nvidia_cusolver_cu12-11.6.1.9-py3-none-
manylinux2014_x86_64.whl (127.9 MB)
                         127.9/127.9 MB
7.6 MB/s eta 0:00:00
Downloading nvidia_cusparse_cu12-12.3.1.170-py3-none-
manylinux2014_x86_64.whl (207.5 MB)
                         207.5/207.5 MB
6.2 MB/s eta 0:00:00
Downloading nvidia_nvjitlink_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl (21.1 MB)
                         21.1/21.1 MB
83.9 MB/s eta 0:00:00
```

```
Installing collected packages: nvidia-nvjitlink-cu12, nvidia-curand-cu12,
nvidia-cufft-cu12, nvidia-cuda-runtime-cu12, nvidia-cuda-nvrtc-cu12, nvidia-
cuda-cupti-cu12, nvidia-cublas-cu12, nvidia-cusparse-cu12, nvidia-cudnn-cu12,
nvidia-cusolver-cu12, thop
  Attempting uninstall: nvidia-nvjitlink-cu12
    Found existing installation: nvidia-nvjitlink-cu12 12.5.82
    Uninstalling nvidia-nvjitlink-cu12-12.5.82:
      Successfully uninstalled nvidia-nvjitlink-cu12-12.5.82
  Attempting uninstall: nvidia-curand-cu12
    Found existing installation: nvidia-curand-cu12 10.3.6.82
    Uninstalling nvidia-curand-cu12-10.3.6.82:
      Successfully uninstalled nvidia-curand-cu12-10.3.6.82
  Attempting uninstall: nvidia-cufft-cu12
    Found existing installation: nvidia-cufft-cu12 11.2.3.61
    Uninstalling nvidia-cufft-cu12-11.2.3.61:
      Successfully uninstalled nvidia-cufft-cu12-11.2.3.61
  Attempting uninstall: nvidia-cuda-runtime-cu12
    Found existing installation: nvidia-cuda-runtime-cu12 12.5.82
    Uninstalling nvidia-cuda-runtime-cu12-12.5.82:
      Successfully uninstalled nvidia-cuda-runtime-cu12-12.5.82
  Attempting uninstall: nvidia-cuda-nvrtc-cu12
    Found existing installation: nvidia-cuda-nvrtc-cu12 12.5.82
    Uninstalling nvidia-cuda-nvrtc-cu12-12.5.82:
      Successfully uninstalled nvidia-cuda-nvrtc-cu12-12.5.82
  Attempting uninstall: nvidia-cuda-cupti-cu12
    Found existing installation: nvidia-cuda-cupti-cu12 12.5.82
    Uninstalling nvidia-cuda-cupti-cu12-12.5.82:
      Successfully uninstalled nvidia-cuda-cupti-cu12-12.5.82
  Attempting uninstall: nvidia-cublas-cu12
    Found existing installation: nvidia-cublas-cu12 12.5.3.2
    Uninstalling nvidia-cublas-cu12-12.5.3.2:
      Successfully uninstalled nvidia-cublas-cu12-12.5.3.2
  Attempting uninstall: nvidia-cusparse-cu12
    Found existing installation: nvidia-cusparse-cu12 12.5.1.3
    Uninstalling nvidia-cusparse-cu12-12.5.1.3:
      Successfully uninstalled nvidia-cusparse-cu12-12.5.1.3
  Attempting uninstall: nvidia-cudnn-cu12
    Found existing installation: nvidia-cudnn-cu12 9.3.0.75
    Uninstalling nvidia-cudnn-cu12-9.3.0.75:
      Successfully uninstalled nvidia-cudnn-cu12-9.3.0.75
  Attempting uninstall: nvidia-cusolver-cu12
    Found existing installation: nvidia-cusolver-cu12 11.6.3.83
    Uninstalling nvidia-cusolver-cu12-11.6.3.83:
      Successfully uninstalled nvidia-cusolver-cu12-11.6.3.83
Successfully installed nvidia-cublas-cu12-12.4.5.8 nvidia-cuda-cupti-
cu12-12.4.127 nvidia-cuda-nvrtc-cu12-12.4.127 nvidia-cuda-runtime-cu12-12.4.127
nvidia-cudnn-cu12-9.1.0.70 nvidia-cufft-cu12-11.2.1.3 nvidia-curand-
cu12-10.3.5.147 nvidia-cusolver-cu12-11.6.1.9 nvidia-cusparse-cu12-12.3.1.170
```

```
nvidia-nvjitlink-cu12-12.4.127 thop-0.1.1.post2209072238
```

2 Data Augmentation

Our first step is to perform data augmentation. Implement the `compute_train_transform()` function in `cs231n/simclr/data_utils.py` to apply the following random transformations:

1. Randomly resize and crop to 32x32.
2. Horizontally flip the image with probability 0.5
3. With a probability of 0.8, apply color jitter (see `compute_train_transform()` for definition)
4. With a probability of 0.2, convert the image to grayscale

Now complete `compute_train_transform()` and `CIFAR10Pair.__getitem__()` in `cs231n/simclr/data_utils.py` to apply the data augmentation transform and generate $\tilde{x}_i$ and $\tilde{x}_j$.

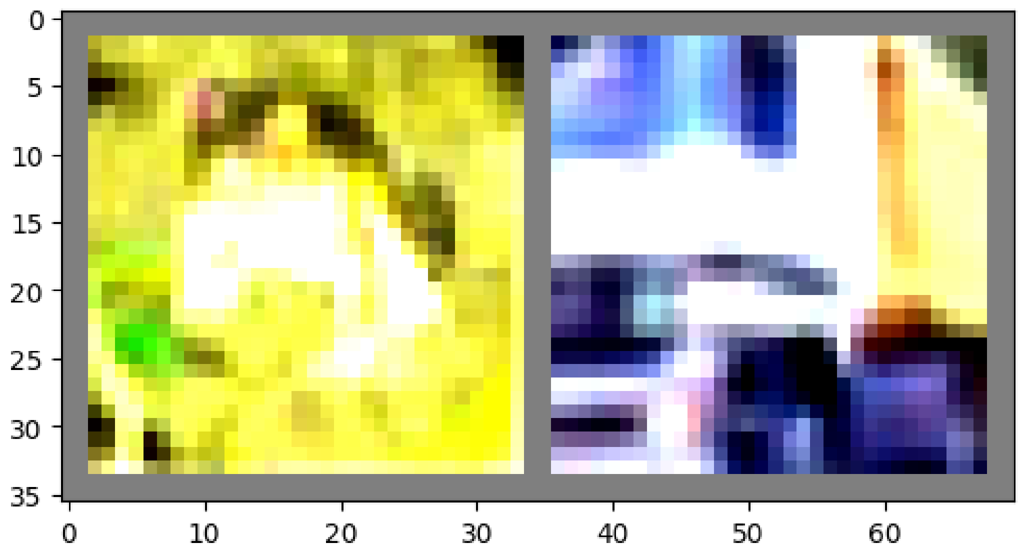
Test to make sure that your data augmentation code is correct:

[5]:
```python
from cs231n.simclr.data_utils import *
from cs231n.simclr.contrastive_loss import *

answers = torch.load('simclr_sanity_check.key')
```

[6]:
```python
from PIL import Image
import torchvision
from torchvision.datasets import CIFAR10

def test_data_augmentation(correct_output=None):
    train_transform = compute_train_transform(seed=2147483647)
    trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
 ↪download=True, transform=train_transform)
    trainloader = torch.utils.data.DataLoader(trainset, batch_size=2,
 ↪shuffle=False, num_workers=2)
    dataiter = iter(trainloader)
    images, labels = next(dataiter)
    img = torchvision.utils.make_grid(images)
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
    output = images

    print("Maximum error in data augmentation: %g"%rel_error( output.numpy(),
 ↪correct_output.numpy()))

# Should be less than 1e-07.
test_data_augmentation(answers['data_augmentation'])
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
```

```
RGB data ([0..1] for floats or [0..255] for integers). Got range
[-0.4334516..1.8768656].
```




```
Maximum error in data augmentation: 0
```

3 Base Encoder and Projection Head

The next steps are to apply the base encoder and projection head to the augmented samples $\tilde{x}_i$ and $\tilde{x}_j$.

The base encoder f extracts representation vectors for the augmented samples. The SimCLR paper found that using deeper and wider models improved performance and thus chose ResNet to use as the base encoder. The output of the base encoder are the representation vectors $h_i = f(\tilde{x}_i)$ and $h_j = f(\tilde{x}_j)$.

The projection head g is a small neural network that maps the representation vectors h_i and h_j to the space where the contrastive loss is applied. The paper found that using a nonlinear projection head improved the representation quality of the layer before it. Specifically, they used a MLP with one hidden layer as the projection head g. The contrastive loss is then computed based on the outputs $z_i = g(h_i)$ and $z_j = g(h_j)$.

We provide implementations of these two parts in `cs231n/simclr/model.py`. Please skim through the file and make sure you understand the implementation.

4 SimCLR: Contrastive Loss

A mini-batch of N training images yields a total of $2N$ data-augmented examples. For each positive pair (i, j) of augmented examples, the contrastive loss function aims to maximize the agreement of vectors z_i and z_j. Specifically, the loss is the normalized temperature-scaled cross entropy loss and aims to maximize the agreement of z_i and z_j relative to all other augmented examples in the batch:

$$l\ (i,j) = -\log \frac{\exp(\ \text{sim}(z_i, z_j)\ /\ \tau)}{\sum_{k=1}^{2N} \mathbb{1}_{k \neq i} \exp(\ \text{sim}(z_i, z_k)\ /\ \tau)}$$

where $\mathbb{1} \in \{0, 1\}$ is an indicator function that outputs 1 if $k \neq i$ and 0 otherwise. τ is a temperature parameter that determines how fast the exponentials increase.

$\text{sim}(z_i, z_j) = \frac{z_i \cdot z_j}{||z_i||||z_j||}$ is the (normalized) dot product between vectors z_i and z_j. The higher the similarity between z_i and z_j, the larger the dot product is, and the larger the numerator becomes. The denominator normalizes the value by summing across z_i and all other augmented examples k in the batch. The range of the normalized value is $(0, 1)$, where a high score close to 1 corresponds to a high similarity between the positive pair (i, j) and low similarity between i and other augmented examples k in the batch. The negative log then maps the range $(0, 1)$ to the loss values $(\inf, 0)$.

The total loss is computed across all positive pairs (i, j) in the batch. Let $z = [z_1, z_2, ..., z_{2N}]$ include all the augmented examples in the batch, where $z_1 ... z_N$ are outputs of the left branch, and $z_{N+1} ... z_{2N}$ are outputs of the right branch. Thus, the positive pairs are (z_k, z_{k+N}) for $\forall k \in [1, N]$.

Then, the total loss L is:

$$L = \frac{1}{2N} \sum_{k=1}^{N} [\ l(k,\ k+N) + l(k+N,\ k)\ ]$$

NOTE: this equation is slightly different from the one in the paper. We've rearranged the ordering of the positive pairs in the batch, so the indices are different. The rearrangement makes it easier to implement the code in vectorized form.

We'll walk through the steps of implementing the loss function in vectorized form. Implement the functions `sim`, `simclr_loss_naive` in `cs231n/simclr/contrastive_loss.py`. Test your code by running the sanity checks below.

[7]:
```
from cs231n.simclr.contrastive_loss import *
answers = torch.load('simclr_sanity_check.key')
```

[8]:
```
def test_sim(left_vec, right_vec, correct_output):
    output = sim(left_vec, right_vec).cpu().numpy()
    print("Maximum error in sim: %g"%rel_error(correct_output.numpy(), output))

# Should be less than 1e-07.
test_sim(answers['left'][0], answers['right'][0], answers['sim'][0])
test_sim(answers['left'][1], answers['right'][1], answers['sim'][1])
```

```
Maximum error in sim: 3.81097e-08
Maximum error in sim: 0
```

[9]:
```
def test_loss_naive(left, right, tau, correct_output):
    naive_loss = simclr_loss_naive(left, right, tau).item()
    print("Maximum error in simclr_loss_naive: %g"%rel_error(correct_output,␣
 ↪naive_loss))
```

```
# Should be less than 1e-07.
test_loss_naive(answers['left'], answers['right'], 5.0, answers['loss']['5.0'])
test_loss_naive(answers['left'], answers['right'], 1.0, answers['loss']['1.0'])
```

```
Maximum error in simclr_loss_naive: 0
Maximum error in simclr_loss_naive: 5.65617e-08
```

Now implement the vectorized version by implementing `sim_positive_pairs`, `compute_sim_matrix`, `simclr_loss_vectorized` in `cs231n/simclr/contrastive_loss.py`. Test your code by running the sanity checks below.

[10]:
```
def test_sim_positive_pairs(left, right, correct_output):
    sim_pair = sim_positive_pairs(left, right).cpu().numpy()
    print("Maximum error in sim_positive_pairs: %g"%rel_error(correct_output.
 ↪numpy(), sim_pair))

# Should be less than 1e-07.
test_sim_positive_pairs(answers['left'], answers['right'], answers['sim'])
```

```
Maximum error in sim_positive_pairs: 3.81097e-08
```

[11]:
```
def test_sim_matrix(left, right, correct_output):
    out = torch.cat([left, right], dim=0)
    sim_matrix = compute_sim_matrix(out).cpu()
    assert torch.isclose(sim_matrix, correct_output).all(), "correct: {}. got:␣
 ↪{}".format(correct_output, sim_matrix)
    print("Test passed!")

test_sim_matrix(answers['left'], answers['right'], answers['sim_matrix'])
```

```
Test passed!
```

[12]:
```
def test_loss_vectorized(left, right, tau, correct_output):
    vec_loss = simclr_loss_vectorized(left, right, tau, device=left.device).
 ↪item()
    print("Maximum error in loss_vectorized: %g"%rel_error(correct_output,␣
 ↪vec_loss))

# Should be less than 1e-07.
test_loss_vectorized(answers['left'], answers['right'], 5.0, answers['loss']['5.
 ↪0'])
test_loss_vectorized(answers['left'], answers['right'], 1.0, answers['loss']['1.
 ↪0'])
```

```
Maximum error in loss_vectorized: 0.6
Maximum error in loss_vectorized: 0.6
```

5 Implement the train function

Complete the `train()` function in `cs231n/simclr/utils.py` to obtain the model's output and use `simclr_loss_vectorized` to compute the loss. (Please take a look at the `Model` class in `cs231n/simclr/model.py` to understand the model pipeline and the returned values)

[13]:
```
from cs231n.simclr.data_utils import *
from cs231n.simclr.model import *
from cs231n.simclr.utils import *
```

5.0.1 Train the SimCLR model

Run the following cells to load in the pretrained weights and continue to train a little bit more. This part will take ~10 minutes and will output to `pretrained_model/trained_simclr_model.pth`.

NOTE: Don't worry about logs such as '*[WARN] Cannot find rule for ...*'. These are related to another module used in the notebook. You can verify the integrity of your code changes through our provided prompts and comments.

[14]:
```
# Do not modify this cell.
feature_dim = 128
temperature = 0.5
k = 200
batch_size = 64
epochs = 1
temperature = 0.5
percentage = 0.5
pretrained_path = './pretrained_model/pretrained_simclr_model.pth'

# Prepare the data.
train_transform = compute_train_transform()
train_data = CIFAR10Pair(root='data', train=True, transform=train_transform,␣
 ↪download=True)
train_data = torch.utils.data.Subset(train_data, list(np.
 ↪arange(int(len(train_data)*percentage))))
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True,␣
 ↪num_workers=16, pin_memory=True, drop_last=True)
test_transform = compute_test_transform()
memory_data = CIFAR10Pair(root='data', train=True, transform=test_transform,␣
 ↪download=True)
memory_loader = DataLoader(memory_data, batch_size=batch_size, shuffle=False,␣
 ↪num_workers=16, pin_memory=True)
test_data = CIFAR10Pair(root='data', train=False, transform=test_transform,␣
 ↪download=True)
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False,␣
 ↪num_workers=16, pin_memory=True)

# Set up the model and optimizer config.
```

```
model = Model(feature_dim)
model.load_state_dict(torch.load(pretrained_path, map_location='cpu'),␣
 ↪strict=False)
model = model.to(device)
flops, params = profile(model, inputs=(torch.randn(1, 3, 32, 32).to(device),))
flops, params = clever_format([flops, params])
print('# Model Params: {} FLOPs: {}'.format(params, flops))
optimizer = optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-6)
c = len(memory_data.classes)

# Training loop.
results = {'train_loss': [], 'test_acc@1': [], 'test_acc@5': []} #<< -- output

if not os.path.exists('results'):
    os.mkdir('results')
best_acc = 0.0
for epoch in range(1, epochs + 1):
    train_loss = train(model, train_loader, optimizer, epoch, epochs,␣
 ↪batch_size=batch_size, temperature=temperature, device=device)
    results['train_loss'].append(train_loss)
    test_acc_1, test_acc_5 = test(model, memory_loader, test_loader, epoch,␣
 ↪epochs, c, k=k, temperature=temperature, device=device)
    results['test_acc@1'].append(test_acc_1)
    results['test_acc@5'].append(test_acc_5)

    # Save statistics.
    if test_acc_1 > best_acc:
        best_acc = test_acc_1
        torch.save(model.state_dict(), './pretrained_model/trained_simclr_model.
 ↪pth')
```

```
/usr/local/lib/python3.11/dist-packages/torch/utils/data/dataloader.py:624:
UserWarning: This DataLoader will create 16 worker processes in total. Our
suggested max number of worker in current system is 2, which is smaller than
what this DataLoader is going to create. Please be aware that excessive worker
creation might get DataLoader running slow or even freeze, lower the worker
number to avoid potential slowness/freeze if necessary.
  warnings.warn(
```

```
[INFO] Register count_convNd() for <class 'torch.nn.modules.conv.Conv2d'>.
[INFO] Register count_normalization() for <class
'torch.nn.modules.batchnorm.BatchNorm2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.activation.ReLU'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.container.Sequential'>.
[INFO] Register count_adap_avgpool() for <class
'torch.nn.modules.pooling.AdaptiveAvgPool2d'>.
[INFO] Register count_linear() for <class 'torch.nn.modules.linear.Linear'>.
[INFO] Register count_normalization() for <class
```

```
'torch.nn.modules.batchnorm.BatchNorm1d'>.
# Model Params: 24.62M FLOPs: 1.31G
```

```
Train Epoch: [1/1] Loss: 416.9781: 100%|     | 390/390 [02:38<00:00,
2.46it/s]
Feature extracting: 100%|     | 782/782 [00:47<00:00, 16.44it/s]
Test Epoch: [1/1] Acc@1:83.44% Acc@5:99.34%: 100%|     | 157/157
[00:12<00:00, 12.55it/s]
```

6 Finetune a Linear Layer for Classification!

Now it's time to put the representation vectors to the test!

We remove the projection head from the SimCLR model and slap on a linear layer to finetune for a simple classification task. All layers before the linear layer are frozen, and only the weights in the final linear layer are trained. We compare the performance of the SimCLR + finetuning model against a baseline model, where no self-supervised learning is done beforehand, and all weights in the model are trained. You will get to see for yourself the power of self-supervised learning and how the learned representation vectors improve downstream task performance.

6.1 Baseline: Without Self-Supervised Learning

First, let's take a look at the baseline model. We'll remove the projection head from the SimCLR model and slap on a linear layer to finetune for a simple classification task. No self-supervised learning is done beforehand, and all weights in the model are trained. Run the following cells.

NOTE: Don't worry if you see low but reasonable performance.

[15]:
```python
class Classifier(nn.Module):
    def __init__(self, num_class):
        super(Classifier, self).__init__()

        # Encoder.
        self.f = Model().f

        # Classifier.
        self.fc = nn.Linear(2048, num_class, bias=True)

    def forward(self, x):
        x = self.f(x)
        feature = torch.flatten(x, start_dim=1)
        out = self.fc(feature)
        return out
```

[16]:
```python
# Do not modify this cell.
feature_dim = 128
temperature = 0.5
k = 200
batch_size = 128
```

```python
epochs = 10
percentage = 0.1

train_transform = compute_train_transform()
train_data = CIFAR10(root='data', train=True, transform=train_transform, download=True)
trainset = torch.utils.data.Subset(train_data, list(np.arange(int(len(train_data)*percentage))))
train_loader = DataLoader(trainset, batch_size=batch_size, shuffle=True, num_workers=16, pin_memory=True)
test_transform = compute_test_transform()
test_data = CIFAR10(root='data', train=False, transform=test_transform, download=True)
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False, num_workers=16, pin_memory=True)

model = Classifier(num_class=len(train_data.classes)).to(device)
for param in model.f.parameters():
    param.requires_grad = False

flops, params = profile(model, inputs=(torch.randn(1, 3, 32, 32).to(device),))
flops, params = clever_format([flops, params])
print('# Model Params: {} FLOPs: {}'.format(params, flops))
optimizer = optim.Adam(model.fc.parameters(), lr=1e-3, weight_decay=1e-6)
no_pretrain_results = {'train_loss': [], 'train_acc@1': [], 'train_acc@5': [],
           'test_loss': [], 'test_acc@1': [], 'test_acc@5': []}

best_acc = 0.0
for epoch in range(1, epochs + 1):
    train_loss, train_acc_1, train_acc_5 = train_val(model, train_loader, optimizer, epoch, epochs, device='cuda')
    no_pretrain_results['train_loss'].append(train_loss)
    no_pretrain_results['train_acc@1'].append(train_acc_1)
    no_pretrain_results['train_acc@5'].append(train_acc_5)
    test_loss, test_acc_1, test_acc_5 = train_val(model, test_loader, None, epoch, epochs)
    no_pretrain_results['test_loss'].append(test_loss)
    no_pretrain_results['test_acc@1'].append(test_acc_1)
    no_pretrain_results['test_acc@5'].append(test_acc_5)
    if test_acc_1 > best_acc:
        best_acc = test_acc_1

# Print the best test accuracy.
print('Best top-1 accuracy without self-supervised learning: ', best_acc)
```

```
[INFO] Register count_convNd() for <class 'torch.nn.modules.conv.Conv2d'>.
[INFO] Register count_normalization() for <class
```

```
'torch.nn.modules.batchnorm.BatchNorm2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.activation.ReLU'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.container.Sequential'>.
[INFO] Register count_adap_avgpool() for <class
'torch.nn.modules.pooling.AdaptiveAvgPool2d'>.
[INFO] Register count_linear() for <class 'torch.nn.modules.linear.Linear'>.
# Model Params: 23.52M FLOPs: 1.31G

Train Epoch: [1/10] Loss: 2.5539 ACC@1: 10.70% ACC@5: 51.30%: 100%|      |
40/40 [00:10<00:00,  3.85it/s]
Test Epoch: [1/10] Loss: 2.3212 ACC@1: 11.48% ACC@5: 51.60%: 100%|      |
79/79 [00:11<00:00,  7.15it/s]
Train Epoch: [2/10] Loss: 2.4299 ACC@1: 10.88% ACC@5: 51.42%: 100%|      |
40/40 [00:09<00:00,  4.36it/s]
Test Epoch: [2/10] Loss: 2.7025 ACC@1: 10.18% ACC@5: 55.10%: 100%|      |
79/79 [00:11<00:00,  7.09it/s]
Train Epoch: [3/10] Loss: 2.3950 ACC@1: 11.70% ACC@5: 53.12%: 100%|      |
40/40 [00:10<00:00,  3.90it/s]
Test Epoch: [3/10] Loss: 2.5049 ACC@1: 10.24% ACC@5: 53.42%: 100%|      |
79/79 [00:10<00:00,  7.31it/s]
Train Epoch: [4/10] Loss: 2.4029 ACC@1: 12.44% ACC@5: 54.02%: 100%|      |
40/40 [00:08<00:00,  4.47it/s]
Test Epoch: [4/10] Loss: 2.5870 ACC@1: 10.34% ACC@5: 52.39%: 100%|      |
79/79 [00:11<00:00,  6.81it/s]
Train Epoch: [5/10] Loss: 2.4127 ACC@1: 12.24% ACC@5: 54.48%: 100%|      |
40/40 [00:10<00:00,  3.73it/s]
Test Epoch: [5/10] Loss: 2.7166 ACC@1: 14.82% ACC@5: 54.43%: 100%|      |
79/79 [00:10<00:00,  7.49it/s]
Train Epoch: [6/10] Loss: 2.3939 ACC@1: 12.44% ACC@5: 54.02%: 100%|      |
40/40 [00:08<00:00,  4.61it/s]
Test Epoch: [6/10] Loss: 2.3872 ACC@1: 13.67% ACC@5: 54.45%: 100%|      |
79/79 [00:11<00:00,  7.00it/s]
Train Epoch: [7/10] Loss: 2.3648 ACC@1: 13.10% ACC@5: 54.66%: 100%|      |
40/40 [00:09<00:00,  4.31it/s]
Test Epoch: [7/10] Loss: 2.4616 ACC@1: 11.80% ACC@5: 55.54%: 100%|      |
79/79 [00:10<00:00,  7.36it/s]
Train Epoch: [8/10] Loss: 2.3864 ACC@1: 11.86% ACC@5: 55.12%: 100%|      |
40/40 [00:10<00:00,  3.81it/s]
Test Epoch: [8/10] Loss: 2.4651 ACC@1: 14.32% ACC@5: 59.70%: 100%|      |
79/79 [00:10<00:00,  7.42it/s]
Train Epoch: [9/10] Loss: 2.3793 ACC@1: 13.22% ACC@5: 56.60%: 100%|      |
40/40 [00:08<00:00,  4.57it/s]
Test Epoch: [9/10] Loss: 2.6685 ACC@1: 10.05% ACC@5: 57.28%: 100%|      |
79/79 [00:11<00:00,  6.79it/s]
Train Epoch: [10/10] Loss: 2.4030 ACC@1: 12.96% ACC@5: 57.64%: 100%|      |
40/40 [00:10<00:00,  3.91it/s]
Test Epoch: [10/10] Loss: 2.4337 ACC@1: 15.30% ACC@5: 58.28%: 100%|      |
79/79 [00:10<00:00,  7.28it/s]
```

```
Best top-1 accuracy without self-supervised learning:  15.299999999999999
```

6.2 With Self-Supervised Learning

Let's see how much improvement we get with self-supervised learning. Here, we pretrain the SimCLR model using the simclr loss you wrote, remove the projection head from the SimCLR model, and use a linear layer to finetune for a simple classification task.

[17]:
```python
# Do not modify this cell.
feature_dim = 128
temperature = 0.5
k = 200
batch_size = 128
epochs = 10
percentage = 0.1
pretrained_path = './pretrained_model/trained_simclr_model.pth'

train_transform = compute_train_transform()
train_data = CIFAR10(root='data', train=True, transform=train_transform,␣
 ↪download=True)
trainset = torch.utils.data.Subset(train_data, list(np.
 ↪arange(int(len(train_data)*percentage))))
train_loader = DataLoader(trainset, batch_size=batch_size, shuffle=True,␣
 ↪num_workers=16, pin_memory=True)
test_transform = compute_test_transform()
test_data = CIFAR10(root='data', train=False, transform=test_transform,␣
 ↪download=True)
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False,␣
 ↪num_workers=16, pin_memory=True)

model = Classifier(num_class=len(train_data.classes))
model.load_state_dict(torch.load(pretrained_path, map_location='cpu'),␣
 ↪strict=False)
model = model.to(device)
for param in model.f.parameters():
    param.requires_grad = False

flops, params = profile(model, inputs=(torch.randn(1, 3, 32, 32).to(device),))
flops, params = clever_format([flops, params])
print('# Model Params: {} FLOPs: {}'.format(params, flops))
optimizer = optim.Adam(model.fc.parameters(), lr=1e-3, weight_decay=1e-6)
pretrain_results = {'train_loss': [], 'train_acc@1': [], 'train_acc@5': [],
           'test_loss': [], 'test_acc@1': [], 'test_acc@5': []}

best_acc = 0.0
for epoch in range(1, epochs + 1):
```

```python
    train_loss, train_acc_1, train_acc_5 = train_val(model, train_loader,␣
↪optimizer, epoch, epochs)
    pretrain_results['train_loss'].append(train_loss)
    pretrain_results['train_acc@1'].append(train_acc_1)
    pretrain_results['train_acc@5'].append(train_acc_5)
    test_loss, test_acc_1, test_acc_5 = train_val(model, test_loader, None,␣
↪epoch, epochs)
    pretrain_results['test_loss'].append(test_loss)
    pretrain_results['test_acc@1'].append(test_acc_1)
    pretrain_results['test_acc@5'].append(test_acc_5)
    if test_acc_1 > best_acc:
        best_acc = test_acc_1

# Print the best test accuracy. You should see a best top-1 accuracy of >=70%.
print('Best top-1 accuracy with self-supervised learning: ', best_acc)
```

```
[INFO] Register count_convNd() for <class 'torch.nn.modules.conv.Conv2d'>.
[INFO] Register count_normalization() for <class
'torch.nn.modules.batchnorm.BatchNorm2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.activation.ReLU'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.container.Sequential'>.
[INFO] Register count_adap_avgpool() for <class
'torch.nn.modules.pooling.AdaptiveAvgPool2d'>.
[INFO] Register count_linear() for <class 'torch.nn.modules.linear.Linear'>.
# Model Params: 23.52M FLOPs: 1.31G
```

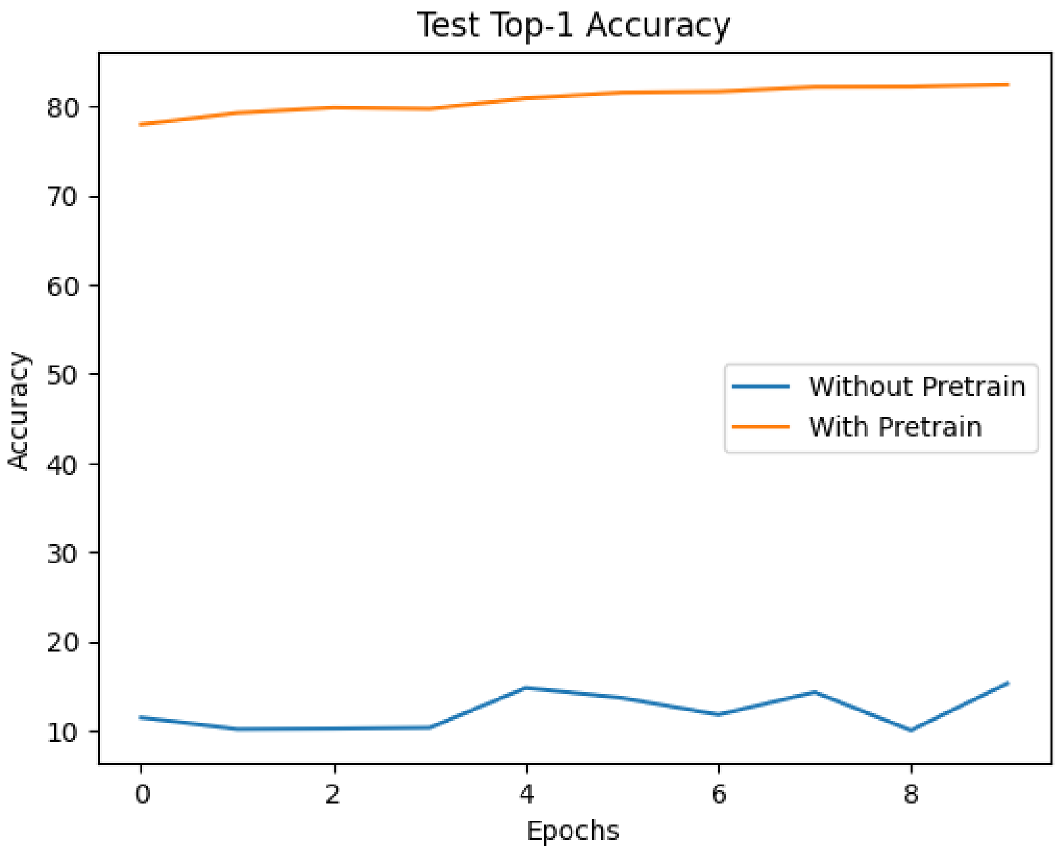
```
Train Epoch: [1/10] Loss: 1.8226 ACC@1: 64.36% ACC@5: 93.74%: 100%|      |
40/40 [00:08<00:00,  4.59it/s]
Test Epoch: [1/10] Loss: 1.3337 ACC@1: 77.95% ACC@5: 98.15%: 100%|      |
79/79 [00:10<00:00,  7.20it/s]
Train Epoch: [2/10] Loss: 1.1894 ACC@1: 76.00% ACC@5: 97.48%: 100%|      |
40/40 [00:10<00:00,  3.77it/s]
Test Epoch: [2/10] Loss: 0.9404 ACC@1: 79.24% ACC@5: 98.25%: 100%|      |
79/79 [00:10<00:00,  7.43it/s]
Train Epoch: [3/10] Loss: 0.9362 ACC@1: 76.26% ACC@5: 97.98%: 100%|      |
40/40 [00:08<00:00,  4.63it/s]
Test Epoch: [3/10] Loss: 0.7820 ACC@1: 79.84% ACC@5: 98.64%: 100%|      |
79/79 [00:11<00:00,  6.78it/s]
Train Epoch: [4/10] Loss: 0.8409 ACC@1: 77.00% ACC@5: 97.70%: 100%|      |
40/40 [00:09<00:00,  4.02it/s]
Test Epoch: [4/10] Loss: 0.7101 ACC@1: 79.70% ACC@5: 98.57%: 100%|      |
79/79 [00:10<00:00,  7.40it/s]
Train Epoch: [5/10] Loss: 0.7679 ACC@1: 77.94% ACC@5: 97.82%: 100%|      |
40/40 [00:09<00:00,  4.39it/s]
Test Epoch: [5/10] Loss: 0.6468 ACC@1: 80.88% ACC@5: 98.88%: 100%|      |
79/79 [00:10<00:00,  7.55it/s]
Train Epoch: [6/10] Loss: 0.7362 ACC@1: 77.70% ACC@5: 97.94%: 100%|      |
40/40 [00:08<00:00,  4.56it/s]
```

```
Test Epoch: [6/10] Loss: 0.6099 ACC@1: 81.52% ACC@5: 98.86%: 100%|      |
79/79 [00:11<00:00,  6.62it/s]
Train Epoch: [7/10] Loss: 0.6972 ACC@1: 78.20% ACC@5: 98.32%: 100%|      |
40/40 [00:10<00:00,  3.82it/s]
Test Epoch: [7/10] Loss: 0.5897 ACC@1: 81.63% ACC@5: 98.93%: 100%|      |
79/79 [00:10<00:00,  7.51it/s]
Train Epoch: [8/10] Loss: 0.6784 ACC@1: 78.28% ACC@5: 98.32%: 100%|      |
40/40 [00:09<00:00,  4.42it/s]
Test Epoch: [8/10] Loss: 0.5677 ACC@1: 82.15% ACC@5: 99.00%: 100%|      |
79/79 [00:11<00:00,  7.08it/s]
Train Epoch: [9/10] Loss: 0.6683 ACC@1: 78.58% ACC@5: 98.28%: 100%|      |
40/40 [00:09<00:00,  4.42it/s]
Test Epoch: [9/10] Loss: 0.5568 ACC@1: 82.18% ACC@5: 98.95%: 100%|      |
79/79 [00:10<00:00,  7.39it/s]
Train Epoch: [10/10] Loss: 0.6441 ACC@1: 79.16% ACC@5: 98.22%: 100%|      |
40/40 [00:11<00:00,  3.54it/s]
Test Epoch: [10/10] Loss: 0.5406 ACC@1: 82.42% ACC@5: 98.91%: 100%|      |
79/79 [00:10<00:00,  7.50it/s]

Best top-1 accuracy with self-supervised learning:  82.42
```

6.2.1 Plot your Comparison

Plot the test accuracies between the baseline model (no pretraining) and same model pretrained with self-supervised learning.

[18]:
```
plt.plot(no_pretrain_results['test_acc@1'], label="Without Pretrain")
plt.plot(pretrain_results['test_acc@1'], label="With Pretrain")
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Test Top-1 Accuracy')
plt.legend()
plt.show()
```


Test Top-1 Accuracy
Accuracy
Epochs
Without Pretrain
With Pretrain
80
70
60
50
40
30
20
10
0
2
4
6
8

[ ]:

LSTM_Captioning

April 14, 2025

[ ]:
```
# 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. 'cs231n/assignments/assignment3/'
FOLDERNAME = None
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/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

1 Image Captioning with LSTMs

In the previous exercise, you implemented a vanilla RNN and applied it to image captioning. In this notebook, you will implement the LSTM update rule and use it for image captioning.

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

from cs231n.gradient_check import eval_numerical_gradient,␣
↪eval_numerical_gradient_array
from cs231n.rnn_layers import *
from cs231n.captioning_solver import CaptioningSolver
from cs231n.classifiers.rnn import CaptioningRNN
```

```
from cs231n.coco_utils import load_coco_data, sample_coco_minibatch,␣
↪decode_captions
from cs231n.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 notebook, we will use the COCO dataset for captioning.

[ ]:
```
# 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))
```

3 LSTM

A common variant on the vanilla RNN is the Long-Short Term Memory (LSTM) RNN. Vanilla RNNs can be tough to train on long sequences due to vanishing and exploding gradients caused by repeated matrix multiplication. LSTMs solve this problem by replacing the simple update rule of the vanilla RNN with a gating mechanism as follows.

Similar to the vanilla RNN, at each timestep we receive an input $x_t \in \mathbb{R}^D$ and the previous hidden state $h_{t-1} \in \mathbb{R}^H$; the LSTM also maintains an H-dimensional *cell state*, so we also receive the previous cell state $c_{t-1} \in \mathbb{R}^H$. The learnable parameters of the LSTM are an *input-to-hidden* matrix $W_x \in \mathbb{R}^{4H\times D}$, a *hidden-to-hidden* matrix $W_h \in \mathbb{R}^{4H\times H}$ and a *bias vector* $b \in \mathbb{R}^{4H}$.

At each timestep we first compute an *activation vector* $a \in \mathbb{R}^{4H}$ as $a = W_x x_t + W_h h_{t-1} + b$. We then divide this into four vectors $a_i, a_f, a_o, a_g \in \mathbb{R}^H$ where a_i consists of the first H elements of a, a_f is the next H elements of a, etc. We then compute the *input gate* $g \in \mathbb{R}^H$, *forget gate* $f \in \mathbb{R}^H$, *output gate* $o \in \mathbb{R}^H$ and *block input* $g \in \mathbb{R}^H$ as

$$i = \sigma(a_i) \qquad f = \sigma(a_f) \qquad o = \sigma(a_o) \qquad g = \tanh(a_g)$$

where σ is the sigmoid function and tanh is the hyperbolic tangent, both applied elementwise.

Finally we compute the next cell state c_t and next hidden state h_t as

$$c_t = f \odot c_{t-1} + i \odot g \qquad\qquad h_t = o \odot \tanh(c_t)$$

where $\odot$ is the elementwise product of vectors.

In the rest of the notebook we will implement the LSTM update rule and apply it to the image captioning task.

In the code, we assume that data is stored in batches so that $X_t \in \mathbb{R}^{N\times D}$ and will work with *transposed* versions of the parameters: $W_x \in \mathbb{R}^{D\times 4H}$, $W_h \in \mathbb{R}^{H\times 4H}$ so that activations $A \in \mathbb{R}^{N\times 4H}$ can be computed efficiently as $A = X_t W_x + H_{t-1} W_h$

4 LSTM: Step Forward

Implement the forward pass for a single timestep of an LSTM in the `lstm_step_forward` function in the file `cs231n/rnn_layers.py`. This should be similar to the `rnn_step_forward` function that you implemented above, but using the LSTM update rule instead.

Once you are done, run the following to perform a simple test of your implementation. You should see errors on the order of `e-8` or less.

[ ]:
```
N, D, H = 3, 4, 5
x = np.linspace(-0.4, 1.2, num=N*D).reshape(N, D)
prev_h = np.linspace(-0.3, 0.7, num=N*H).reshape(N, H)
prev_c = np.linspace(-0.4, 0.9, num=N*H).reshape(N, H)
Wx = np.linspace(-2.1, 1.3, num=4*D*H).reshape(D, 4 * H)
Wh = np.linspace(-0.7, 2.2, num=4*H*H).reshape(H, 4 * H)
b = np.linspace(0.3, 0.7, num=4*H)

next_h, next_c, cache = lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)

expected_next_h = np.asarray([
    [ 0.24635157,  0.28610883,  0.32240467,  0.35525807,  0.38474904],
    [ 0.49223563,  0.55611431,  0.61507696,  0.66844003,  0.7159181 ],
    [ 0.56735664,  0.66310127,  0.74419266,  0.80889665,  0.858299  ]])
expected_next_c = np.asarray([
    [ 0.32986176,  0.39145139,  0.451556,    0.51014116,  0.56717407],
    [ 0.66382255,  0.76674007,  0.87195994,  0.97902709,  1.08751345],
    [ 0.74192008,  0.90592151,  1.07717006,  1.25120233,  1.42395676]])

print('next_h error: ', rel_error(expected_next_h, next_h))
print('next_c error: ', rel_error(expected_next_c, next_c))
```

5 LSTM: Step Backward

Implement the backward pass for a single LSTM timestep in the function `lstm_step_backward` in the file `cs231n/rnn_layers.py`. Once you are done, run the following to perform numeric gradient checking on your implementation. You should see errors on the order of `e-7` or less.

[ ]:
```
np.random.seed(231)

N, D, H = 4, 5, 6
x = np.random.randn(N, D)
prev_h = np.random.randn(N, H)
prev_c = np.random.randn(N, H)
Wx = np.random.randn(D, 4 * H)
Wh = np.random.randn(H, 4 * H)
b = np.random.randn(4 * H)

next_h, next_c, cache = lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)

dnext_h = np.random.randn(*next_h.shape)
dnext_c = np.random.randn(*next_c.shape)

fx_h = lambda x: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]
fh_h = lambda h: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]
fc_h = lambda c: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]
fWx_h = lambda Wx: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]
fWh_h = lambda Wh: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]
fb_h = lambda b: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]

fx_c = lambda x: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]
fh_c = lambda h: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]
fc_c = lambda c: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]
fWx_c = lambda Wx: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]
fWh_c = lambda Wh: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]
fb_c = lambda b: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]

num_grad = eval_numerical_gradient_array

dx_num = num_grad(fx_h, x, dnext_h) + num_grad(fx_c, x, dnext_c)
dh_num = num_grad(fh_h, prev_h, dnext_h) + num_grad(fh_c, prev_h, dnext_c)
dc_num = num_grad(fc_h, prev_c, dnext_h) + num_grad(fc_c, prev_c, dnext_c)
dWx_num = num_grad(fWx_h, Wx, dnext_h) + num_grad(fWx_c, Wx, dnext_c)
dWh_num = num_grad(fWh_h, Wh, dnext_h) + num_grad(fWh_c, Wh, dnext_c)
db_num = num_grad(fb_h, b, dnext_h) + num_grad(fb_c, b, dnext_c)

dx, dh, dc, dWx, dWh, db = lstm_step_backward(dnext_h, dnext_c, cache)

print('dx error: ', rel_error(dx_num, dx))
print('dh error: ', rel_error(dh_num, dh))
```

```python
print('dc error: ', rel_error(dc_num, dc))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))
```

6 LSTM: Forward

In the function `lstm_forward` in the file `cs231n/rnn_layers.py`, implement the `lstm_forward` function to run an LSTM forward on an entire timeseries of data.

When you are done, run the following to check your implementation. You should see an error on the order of `e-7` or less.

[ ]:
```python
N, D, H, T = 2, 5, 4, 3
x = np.linspace(-0.4, 0.6, num=N*T*D).reshape(N, T, D)
h0 = np.linspace(-0.4, 0.8, num=N*H).reshape(N, H)
Wx = np.linspace(-0.2, 0.9, num=4*D*H).reshape(D, 4 * H)
Wh = np.linspace(-0.3, 0.6, num=4*H*H).reshape(H, 4 * H)
b = np.linspace(0.2, 0.7, num=4*H)

h, cache = lstm_forward(x, h0, Wx, Wh, b)

expected_h = np.asarray([
 [[ 0.01764008,  0.01823233,  0.01882671,  0.0194232 ],
  [ 0.11287491,  0.12146228,  0.13018446,  0.13902939],
  [ 0.31358768,  0.33338627,  0.35304453,  0.37250975]],
 [[ 0.45767879,  0.4761092,   0.4936887,   0.51041945],
  [ 0.6704845,   0.69350089,  0.71486014,  0.7346449 ],
  [ 0.81733511,  0.83677871,  0.85403753,  0.86935314]]])

print('h error: ', rel_error(expected_h, h))
```

7 LSTM: Backward

Implement the backward pass for an LSTM over an entire timeseries of data in the function `lstm_backward` in the file `cs231n/rnn_layers.py`. When you are done, run the following to perform numeric gradient checking on your implementation. You should see errors on the order of `e-8` or less. (For `dWh`, it's fine if your error is on the order of `e-6` or less).

[ ]:
```python
from cs231n.rnn_layers import lstm_forward, lstm_backward
np.random.seed(231)

N, D, T, H = 2, 3, 10, 6

x = np.random.randn(N, T, D)
h0 = np.random.randn(N, H)
Wx = np.random.randn(D, 4 * H)
```

```
Wh = np.random.randn(H, 4 * H)
b = np.random.randn(4 * H)

out, cache = lstm_forward(x, h0, Wx, Wh, b)

dout = np.random.randn(*out.shape)

dx, dh0, dWx, dWh, db = lstm_backward(dout, cache)

fx = lambda x: lstm_forward(x, h0, Wx, Wh, b)[0]
fh0 = lambda h0: lstm_forward(x, h0, Wx, Wh, b)[0]
fWx = lambda Wx: lstm_forward(x, h0, Wx, Wh, b)[0]
fWh = lambda Wh: lstm_forward(x, h0, Wx, Wh, b)[0]
fb = lambda b: lstm_forward(x, h0, Wx, Wh, b)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
dh0_num = eval_numerical_gradient_array(fh0, h0, dout)
dWx_num = eval_numerical_gradient_array(fWx, Wx, dout)
dWh_num = eval_numerical_gradient_array(fWh, Wh, dout)
db_num = eval_numerical_gradient_array(fb, b, dout)

print('dx error: ', rel_error(dx_num, dx))
print('dh0 error: ', rel_error(dh0_num, dh0))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))
```

8 LSTM Captioning Model

Now that you have implemented an LSTM, update the implementation of the `loss` method of the `CaptioningRNN` class in the file `cs231n/classifiers/rnn.py` to handle the case where `self.cell_type` is `lstm`. This should require adding less than 10 lines of code.

Once you have done so, run the following to check your implementation. You should see a difference on the order of `e-10` or less.

[ ]:
```
N, D, W, H = 10, 20, 30, 40
word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
V = len(word_to_idx)
T = 13

model = CaptioningRNN(
    word_to_idx,
    input_dim=D,
    wordvec_dim=W,
    hidden_dim=H,
    cell_type='lstm',
```

```
    dtype=np.float64
)

# Set all model parameters to fixed values
for k, v in model.params.items():
  model.params[k] = np.linspace(-1.4, 1.3, num=v.size).reshape(*v.shape)

features = np.linspace(-0.5, 1.7, num=N*D).reshape(N, D)
captions = (np.arange(N * T) % V).reshape(N, T)

loss, grads = model.loss(features, captions)
expected_loss = 9.82445935443

print('loss: ', loss)
print('expected loss: ', expected_loss)
print('difference: ', abs(loss - expected_loss))
```

9 Overfit LSTM Captioning Model on Small Data

Run the following to overfit an LSTM captioning model on the same small dataset as we used for the RNN previously. You should see a final loss less than 0.5.

[ ]:
```
np.random.seed(231)

small_data = load_coco_data(max_train=50)

small_lstm_model = CaptioningRNN(
    cell_type='lstm',
    word_to_idx=data['word_to_idx'],
    input_dim=data['train_features'].shape[1],
    hidden_dim=512,
    wordvec_dim=256,
    dtype=np.float32,
)

small_lstm_solver = CaptioningSolver(
    small_lstm_model, small_data,
    update_rule='adam',
    num_epochs=50,
    batch_size=25,
    optim_config={
     'learning_rate': 5e-3,
    },
    lr_decay=0.995,
    verbose=True, print_every=10,
)
```

```
small_lstm_solver.train()

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

Print final training loss. You should see a final loss of less than 0.5.

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

10 LSTM Sampling at Test Time

Modify the `sample` method of the `CaptioningRNN` class to handle the case where `self.cell_type` is `lstm`. This should take fewer than 10 lines of code.

When you are done run the following to sample from your overfit LSTM model on some training and validation set samples. As with the RNN, training results should be very good, and validation results probably won't make a lot of sense (because we're overfitting).

[ ]:
```
# 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(small_data, split=split, batch_size=2)
    gt_captions, features, urls = minibatch
    gt_captions = decode_captions(gt_captions, data['idx_to_word'])

    sample_captions = small_lstm_model.sample(features)
    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()
```

[ ]:
```

```