Generative_Adversarial_Networks_TF

May 27, 2020

[0]:
```
from google.colab import drive

drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# 'cs231n' folder containing the '.py', 'classifiers' and 'datasets'
# folders.
FOLDERNAME = 'hw6_files/'

assert FOLDERNAME is not None, "[!] Enter the foldername."

%cd drive/My\ Drive
%cp -r $FOLDERNAME ../../
%cd ../..
%cd 'hw6_files/'
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id
=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redire
ct_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20http
s%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.c
om%2fauth%2fdrive%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.reado
nly%20https%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly

Enter your authorization code:
ůůůůůůůůůů
Mounted at /content/drive
/content/drive/My Drive
/content
/content/hw6_files
```

0.0.1 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 $x \sim p_{\text{data}}$ are samples from the input data, $z \sim p(z)$ 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 aternate 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:

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

0.0.2 What else is there?

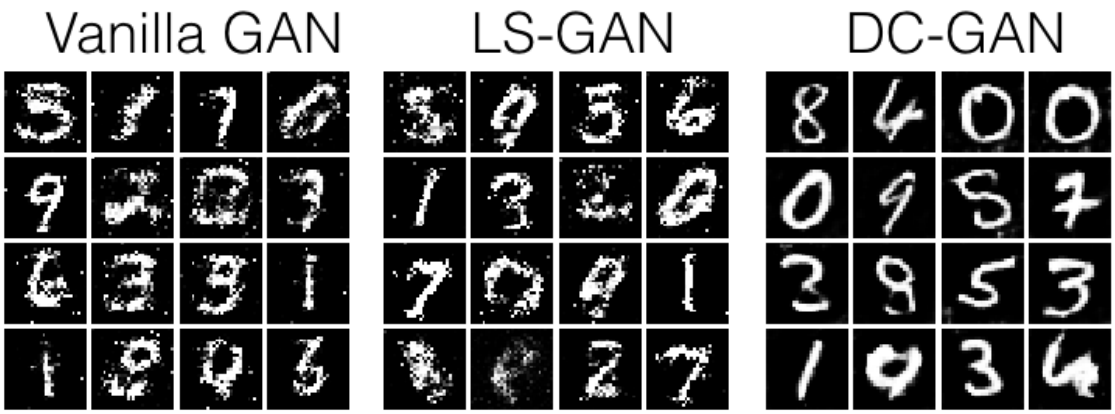
Since 2014, GANs have exploded into a huge research area, with massive workshops, and hundreds of new papers. Compared to other approaches for generative models, they often produce the highest quality samples but are some of the most difficult and finicky models to train (see this github repo that contains a set of 17 hacks that are useful for getting models working). Improving the stabiilty and robustness of GAN training is an open research question, with new papers coming out every day! For a more recent tutorial on GANs, see here. There is also some even more recent exciting work that changes the objective function to Wasserstein distance and yields much more stable results across model architectures: WGAN, WGAN-GP.

GANs are not the only way to train a generative model! For other approaches to generative modeling check out the deep generative model chapter of the Deep Learning book.

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:

[0]:
```
from IPython.display import Image
Image(filename="gan_outputs_tf.png")
```

[0]:



0.1 Setup

[0]:
```
import tensorflow as tf
import numpy as np
import os

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

%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'

# A bunch of utility functions

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

def preprocess_img(x):
    return 2 * x - 1.0

def deprocess_img(x):
    return (x + 1.0) / 2.0

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

def count_params(model):
    """Count the number of parameters in the current TensorFlow graph """
    param_count = np.sum([np.prod(p.shape) for p in model.weights])
    return param_count

answers = np.load('gan-checks-tf.npz')

NOISE_DIM = 10
NUM_SAMPLES = 10000
```

0.2 Dataset

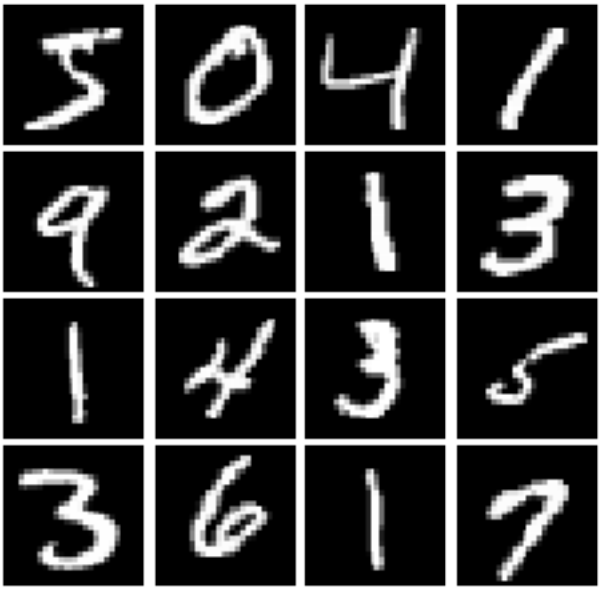
GANs are notoriously finicky with hyperparameters, and also require many training epochs. In order to make this assignment approachable without a GPU, we will be working on the MNIST dataset, which is 60,000 training and 10,000 test images. Each picture contains a centered image of white digit on black 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.

Heads-up: Our MNIST wrapper returns images as vectors. That is, they're size (batch, 784). If you want to treat them as images, we have to resize them to (batch,28,28) or (batch,28,28,1). They are also type np.float32 and bounded [0,1].

[0]:
```
class MNIST(object):
    def __init__(self, batch_size, shuffle=False):
        """
        Construct an iterator object over the MNIST data

        Inputs:
        - batch_size: Integer giving number of elements per minibatch
        - shuffle: (optional) Boolean, whether to shuffle the data on each␣
 ↪epoch
        """
        train, _ = tf.keras.datasets.mnist.load_data()
        X, y = train
        X = X.astype(np.float32)/255
```

```
        X = X.reshape((X.shape[0], -1))
        self.X, self.y = X, y
        self.batch_size, self.shuffle = batch_size, shuffle

    def __iter__(self):
        N, B = self.X.shape[0], self.batch_size
        idxs = np.arange(N)
        if self.shuffle:
            np.random.shuffle(idxs)
        return iter((self.X[i:i+B], self.y[i:i+B]) for i in range(0, N, B))
```

[0]:
```
# show a batch
mnist = MNIST(batch_size=16)
show_images(mnist.X[:16])
```



0.3 LeakyReLU

In the cell below, you should implement a LeakyReLU. See the class notes (where alpha is small number) or equation (3) in this paper. LeakyReLUs keep ReLU units from dying and are often used in GAN methods (as are maxout units, however those increase model size and therefore are not used in this notebook).

HINT: You should be able to use `tf.maximum`

[0]:
```
def leaky_relu(x, alpha=0.01):
    """Compute the leaky ReLU activation function.

    Inputs:
    - x: TensorFlow Tensor with arbitrary shape
```

```
    - alpha: leak parameter for leaky ReLU

    Returns:
    TensorFlow Tensor with the same shape as x
    """
    # TODO: implement leaky ReLU
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    return tf.maximum(alpha * x, x)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

Test your leaky ReLU implementation. You should get errors < 1e-10

[0]:
```
def test_leaky_relu(x, y_true):
    y = leaky_relu(tf.constant(x))
    print('Maximum error: %g'%rel_error(y_true, y))

test_leaky_relu(answers['lrelu_x'], answers['lrelu_y'])
```

```
Maximum error: 0
```

0.4 Random Noise

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

[0]:
```
def sample_noise(batch_size, dim):
    """Generate random uniform noise from -1 to 1.

    Inputs:
    - batch_size: integer giving the batch size of noise to generate
    - dim: integer giving the dimension of the noise to generate

    Returns:
    TensorFlow Tensor containing uniform noise in [-1, 1] with shape␣
 ↪[batch_size, dim]
    """
    # TODO: sample and return noise
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    return tf.random.uniform((batch_size, dim), minval=-1, maxval= 1)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

Make sure noise is the correct shape and type:

[0]:
```
def test_sample_noise():
    batch_size = 3
    dim = 4
    z = sample_noise(batch_size, dim)
```

```
    # Check z has the correct shape
    assert z.get_shape().as_list() == [batch_size, dim]
    # Make sure z is a Tensor and not a numpy array
    assert isinstance(z, tf.Tensor)
    # Check that we get different noise for different evaluations
    z1 = sample_noise(batch_size, dim)
    z2 = sample_noise(batch_size, dim)
    assert not np.array_equal(z1, z2)
    # Check that we get the correct range
    assert np.all(z1 >= -1.0) and np.all(z1 <= 1.0)
    print("All tests passed!")

test_sample_noise()
```

```
All tests passed!
```

0.5 Discriminator

Our first step is to build a discriminator. **Hint:** You should use the layers in `tf.keras.layers` to build the model. All fully connected layers should include bias terms. For initialization, just use the default initializer used by the `tf.keras.layers` functions.

Architecture: * Fully connected layer with input size 784 and output size 256 * LeakyReLU with alpha 0.01 * Fully connected layer with output size 256 * LeakyReLU with alpha 0.01 * Fully connected layer with output size 1

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

[0]:
```
def discriminator():
    """Compute discriminator score for a batch of input images.

    Inputs:
    - x: TensorFlow Tensor of flattened input images, shape [batch_size, 784]

    Returns:
    TensorFlow Tensor with shape [batch_size, 1], containing the score
    for an image being real for each input image.
    """
    model = tf.keras.models.Sequential([
        # TODO: implement architecture
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        tf.keras.layers.Dense(256, activation=leaky_relu, use_bias=True,␣
↪input_shape=(784,)),
        tf.keras.layers.Dense(256, activation=leaky_relu, use_bias=True),
        tf.keras.layers.Dense(1, use_bias=True),

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ])
```

```
    return model
```

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

[0]:
```
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. {0} instead of␣
 ↪{1}. Check your achitecture.'.format(cur_count,true_count))
    else:
        print('Correct number of parameters in discriminator.')

test_discriminator()
```

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

0.6 Generator

Now to build a generator. You should use the layers in `tf.keras.layers` to construct the model. All fully connected layers should include bias terms. Note that you can use the tf.nn module to access activation functions. Once again, use the default initializers for parameters.

Architecture: * Fully connected layer with inupt size tf.shape(z)[1] (the number of noise dimensions) and output size 1024 * `ReLU` * Fully connected layer with output size 1024 * `ReLU` * Fully connected layer with output size 784 * `TanH` (To restrict every element of the output to be in the range [-1,1])

[0]:
```
def generator(noise_dim=NOISE_DIM):
    """Generate images from a random noise vector.

    Inputs:
    - z: TensorFlow Tensor of random noise with shape [batch_size, noise_dim]

    Returns:
    TensorFlow Tensor of generated images, with shape [batch_size, 784].
    """
    model = tf.keras.models.Sequential([
        # TODO: implement architecture
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        tf.keras.layers.Dense(1024, activation='relu', use_bias=True,␣
 ↪input_shape=(noise_dim,)),
        tf.keras.layers.Dense(1024, activation='relu', use_bias=True),
        tf.keras.layers.Dense(784, activation='tanh', use_bias=True),

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ])
    return model
```

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

[0]:
```
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. {0} instead of {1}. Check your achitecture.'.format(cur_count,true_count))
    else:
        print('Correct number of parameters in generator.')

test_generator()
```

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

1 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: Use tf.ones and tf.zeros to generate labels for your discriminator. Use tf.keras.losses.BinaryCrossentropy to help compute your loss function.

[0]:
```
def discriminator_loss(logits_real, logits_fake):
    """
    Computes the discriminator loss described above.

    Inputs:
    - logits_real: Tensor of shape (N, 1) giving scores for the real data.
    - logits_fake: Tensor of shape (N, 1) giving scores for the fake data.

    Returns:
    - loss: Tensor containing (scalar) the loss for the discriminator.
    """
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****


    bce = tf.keras.losses.BinaryCrossentropy(from_logits = True, reduction=tf.keras.losses.Reduction.NONE)
    left = bce(tf.ones(tf.shape(logits_real)), logits_real)
    #print(left, logits_real)
    right = bce(tf.zeros(tf.shape(logits_fake)), logits_fake)
```

```
    loss = tf.reduce_mean(left) + tf.reduce_mean(right)
    #print(logits_real.shape, left.shape, right.shape)

    #D_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(labels=tf.
 ↪ones_like(logits_real), logits=logits_real)) + tf.reduce_mean(tf.nn.
 ↪sigmoid_cross_entropy_with_logits(labels=tf.zeros_like(logits_fake),␣
 ↪logits=logits_fake))

    #print(loss - D_loss)
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return loss

def generator_loss(logits_fake):
    """
    Computes the generator loss described above.

    Inputs:
    - logits_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.

    Returns:
    - loss: PyTorch Tensor containing the (scalar) loss for the generator.
    """
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    bce = tf.keras.losses.BinaryCrossentropy(from_logits = True, reduction=tf.
 ↪keras.losses.Reduction.NONE)
    loss = tf.reduce_mean( bce(tf.ones(tf.shape(logits_fake)), logits_fake))

    #G_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(labels=tf.
 ↪ones_like(logits_fake), logits=logits_fake))

    #print(G_loss - loss)
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return loss
```

Test your GAN loss. Make sure both the generator and discriminator loss are correct. You should see errors less than 1e-8.

[0]:
```
def test_discriminator_loss(logits_real, logits_fake, d_loss_true):
    d_loss = discriminator_loss(tf.constant(logits_real),
                                tf.constant(logits_fake))
    print("Maximum error in d_loss: %g"%rel_error(d_loss_true, d_loss))

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

```
Maximum error in d_loss: 3.31013e-10
```

[0]:
```
def test_generator_loss(logits_fake, g_loss_true):
    g_loss = generator_loss(tf.constant(logits_fake))
    print("Maximum error in g_loss: %g"%rel_error(g_loss_true, g_loss))

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

```
Maximum error in g_loss: 1.34417e-09
```

2 Optimizing our loss

Make an `Adam` optimizer with a 1e-3 learning rate, beta1=0.5 to mininize G_loss and D_loss separately. The trick of decreasing beta was shown to be effective in helping GANs converge in the Improved Techniques for Training GANs paper. In fact, with our current hyperparameters, if you set beta1 to the Tensorflow default of 0.9, there's a good chance your discriminator loss will go to zero and the generator will fail to learn entirely. In fact, this is a common failure mode in GANs; if your D(x) learns too fast (e.g. loss goes near zero), your G(z) is never able to learn. Often D(x) is trained with SGD with Momentum or RMSProp instead of Adam, but here we'll use Adam for both D(x) and G(z).

[0]:
```
# TODO: create an AdamOptimizer for D_solver and G_solver
def get_solvers(learning_rate=1e-3, beta1=0.5):
    """Create solvers for GAN training.

    Inputs:
    - learning_rate: learning rate to use for both solvers
    - beta1: beta1 parameter for both solvers (first moment decay)

    Returns:
    - D_solver: instance of tf.optimizers.Adam with correct learning_rate and␣
 ↪beta1
    - G_solver: instance of tf.optimizers.Adam with correct learning_rate and␣
 ↪beta1
    """
    D_solver = None
    G_solver = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    D_solver = tf.keras.optimizers.Adam(
    learning_rate=learning_rate, beta_1=beta1)

    G_solver = tf.keras.optimizers.Adam(
    learning_rate=learning_rate, beta_1=beta1)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return D_solver, G_solver
```

3 Training a GAN!

Well that wasn't so hard, was it? After the first epoch, you should see fuzzy outlines, clear shapes as you approach epoch 3, and decent shapes, about half of which will be sharp and clearly recognizable as we pass epoch 5. In our case, we'll simply train D(x) and G(z) with one batch each every iteration. However, papers often experiment with different schedules of training D(x) and G(z), sometimes doing one for more steps than the other, or even training each one until the loss gets "good enough" and then switching to training the other.

[0]:
```
# a giant helper function
def run_a_gan(D, G, D_solver, G_solver, discriminator_loss, generator_loss,\
              show_every=20, print_every=20, batch_size=128, num_epochs=10,\
              noise_size=NOISE_DIM):
    """Train a GAN for a certain number of epochs.

    Inputs:
    - D: Discriminator model
    - G: Generator model
    - D_solver: an Optimizer for Discriminator
    - G_solver: an Optimizer for Generator
    - generator_loss: Generator loss
    - discriminator_loss: Discriminator loss
    Returns:
        Nothing
    """
    mnist = MNIST(batch_size=batch_size, shuffle=True)

    iter_count = 0
    for epoch in range(num_epochs):
        for (x, _) in mnist:
            with tf.GradientTape() as tape:
                real_data = x
                logits_real = D(preprocess_img(real_data))

                g_fake_seed = sample_noise(batch_size, noise_size)
                fake_images = G(g_fake_seed)
                logits_fake = D(tf.reshape(fake_images, [batch_size, 784]))

                d_total_error = discriminator_loss(logits_real, logits_fake)
                d_gradients = tape.gradient(d_total_error, D.
↪trainable_variables)
                D_solver.apply_gradients(zip(d_gradients, D.
↪trainable_variables))

            with tf.GradientTape() as tape:
                g_fake_seed = sample_noise(batch_size, noise_size)
                fake_images = G(g_fake_seed)
```

```
                gen_logits_fake = D(tf.reshape(fake_images, [batch_size, 784]))
                g_error = generator_loss(gen_logits_fake)
                g_gradients = tape.gradient(g_error, G.trainable_variables)
                G_solver.apply_gradients(zip(g_gradients, G.
↪trainable_variables))

            #if (iter_count % show_every == 0):
                #print('Epoch: {}, Iter: {}, D: {:.4}, G:{:.4}'.format(epoch,␣
↪iter_count,d_total_error,g_error))
                #imgs_numpy = fake_images.cpu().numpy()
                #show_images(imgs_numpy[0:16])
                #plt.show()
            iter_count += 1

    # random noise fed into our generator
    z = sample_noise(batch_size, noise_size)
    # generated images
    G_sample = G(z)
    print('Final images')
    show_images(G_sample[:16])
    plt.show()
```

Train your GAN! This should take about 10 minutes on a CPU, or about 2 minutes on GPU.

[0]:
```
# Make the discriminator
D = discriminator()

# Make the generator
G = generator()

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

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

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
Final images
```

4 Least Squares GAN

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

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

and the discriminator loss:

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

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

[0]:
```
def ls_discriminator_loss(scores_real, scores_fake):
    """
    Compute the Least-Squares GAN loss for the discriminator.

    Inputs:
    - scores_real: Tensor of shape (N, 1) giving scores for the real data.
    - scores_fake: Tensor of shape (N, 1) giving scores for the fake data.

    Outputs:
    - loss: A Tensor containing the loss.
    """
```

```
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    loss = 0.5 * tf.reduce_mean(tf.math.square(scores_real - 1)) + 0.5 * tf.
 ↪reduce_mean(tf.math.square(scores_fake))

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return loss

def ls_generator_loss(scores_fake):
    """
    Computes the Least-Squares GAN loss for the generator.

    Inputs:
    - scores_fake: Tensor of shape (N, 1) giving scores for the fake data.

    Outputs:
    - loss: A Tensor containing the loss.
    """
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    loss = 0.5 * tf.reduce_mean(tf.math.square(scores_fake - 1))

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return loss
```

Test your LSGAN loss. You should see errors less than 1e-8.

[0]:
```
def test_lsgan_loss(score_real, score_fake, d_loss_true, g_loss_true):

    d_loss = ls_discriminator_loss(tf.constant(score_real), tf.
 ↪constant(score_fake))
    g_loss = ls_generator_loss(tf.constant(score_fake))
    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: 0
Maximum error in g_loss: 0
```

Create new training steps so we instead minimize the LSGAN loss:

[0]:
```
# Make the discriminator
D = discriminator()

# Make the generator
```

```
G = generator()

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

# Run it!
run_a_gan(D, G, D_solver, G_solver, ls_discriminator_loss, ls_generator_loss)
```

```
Final images
```

5 Deep 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 as our discriminators and generators.

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. *Be sure to check the dimensions of x and reshape when needed*, fully connected blocks expect [N,D] Tensors while conv2d blocks expect [N,H,W,C] Tensors. Please use `tf.keras.layers` to define the following architecture:

Architecture: * Conv2D: 32 Filters, 5x5, Stride 1, padding 0 * Leaky ReLU(alpha=0.01) * Max Pool 2x2, Stride 2 * Conv2D: 64 Filters, 5x5, Stride 1, padding 0 * 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

Once again, please use biases for all convolutional and fully connected layers, and use the default parameter initializers. Note that a padding of 0 can be accomplished with the 'VALID' padding option.

[0]:
```
def discriminator():
    """Compute discriminator score for a batch of input images.

    Inputs:
    - x: TensorFlow Tensor of flattened input images, shape [batch_size, 784]

    Returns:
    TensorFlow Tensor with shape [batch_size, 1], containing the score
    for an image being real for each input image.
    """
    model = tf.keras.models.Sequential([
        # TODO: implement architecture
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        tf.keras.layers.Reshape((28, 28, 1), input_shape=(784,)),
        tf.keras.layers.Conv2D(32, 5, strides=(1, 1), activation=leaky_relu),
        tf.keras.layers.MaxPool2D(pool_size=(2, 2), strides=2),
        tf.keras.layers.Conv2D(64, 5, strides=(1, 1), activation=leaky_relu),
        tf.keras.layers.MaxPool2D(pool_size=(2, 2), strides=2),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(4*4*64, activation=leaky_relu),
        tf.keras.layers.Dense(1),
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ])
    return model

model = discriminator()
test_discriminator(1102721)
```

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

Generator For the generator, we will copy the architecture exactly from the InfoGAN paper. See Appendix C.1 MNIST. Please use `tf.keras.layers` for your implementation. You might find the documentation for tf.keras.layers.Conv2DTranspose useful. The architecture is as follows.

Architecture: * Fully connected with output size 1024 * `ReLU` * BatchNorm * Fully connected with output size 7 x 7 x 128 * `ReLU` * BatchNorm * Resize into Image Tensor of size 7, 7, 128 * Conv2D^T (transpose): 64 filters of 4x4, stride 2 * `ReLU` * BatchNorm * Conv2d^T (transpose): 1 filter of 4x4, stride 2 * `TanH`

Once again, use biases for the fully connected and transpose convolutional layers. Please use the default initializers for your parameters. For padding, choose the 'same' option for transpose convolutions. For Batch Normalization, assume we are always in 'training' mode.

[0]:
```
def generator(noise_dim=NOISE_DIM):
    """Generate images from a random noise vector.

    Inputs:
    - z: TensorFlow Tensor of random noise with shape [batch_size, noise_dim]

    Returns:
    TensorFlow Tensor of generated images, with shape [batch_size, 784].
    """
    model = tf.keras.models.Sequential([
    # TODO: implement architecture
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    tf.keras.layers.Dense(1024, activation='relu', input_shape = (noise_dim,)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(7 * 7 * 128, activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Reshape((7, 7, 128)),
    tf.keras.layers.Conv2DTranspose(64, (4,4), strides = (2,2), padding='same',␣
 ↪activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2DTranspose(1, (4,4), strides = (2,2), padding='same',␣
 ↪activation='tanh'),
    tf.keras.layers.Flatten()
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ])
    return model
test_generator(6595521)
```

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

We have to recreate our network since we've changed our functions.

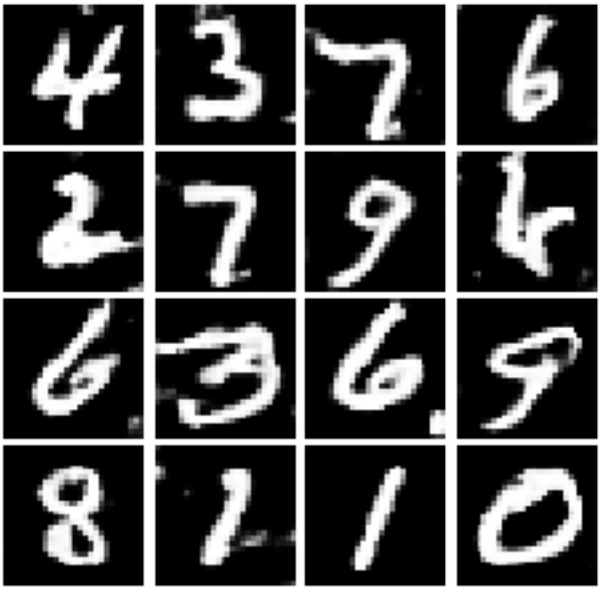
5.0.1 Train and evaluate a DCGAN

This is the one part of A3 that significantly benefits from using a GPU. It takes 3 minutes on a GPU for the requested five epochs. Or about 50 minutes on a dual core laptop on CPU (feel free to use 3 epochs if you do it on CPU).

[0]:
```
# Make the discriminator
D = discriminator()

# Make the generator
G = generator()

# Use the function you wrote earlier to get optimizers for the Discriminator␣
 ↪and the Generator
D_solver, G_solver = get_solvers()
```

```
# Run it!
run_a_gan(D, G, D_solver, G_solver, ls_discriminator_loss, ls_generator_loss,␣
 ↪num_epochs=5)
```

```
Final images
```



5.0.2 Inception score

[0]:
```
batch_size = 128
num_classes = 10
epochs = 20

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
```

```
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(512, activation='relu'))
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(num_classes, activation='softmax'))

model.summary()

model.compile(loss='categorical_crossentropy',
              optimizer=tf.keras.optimizers.RMSprop(),
              metrics=['accuracy'])

history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1,
                    validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
60000 train samples
10000 test samples
Model: "sequential_11"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_28 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout (Dropout)            (None, 512)               0
_________________________________________________________________
dense_29 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_30 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
_________________________________________________________________
Epoch 1/20
469/469 [==============================] - 3s 6ms/step - loss: 0.2458 -
```

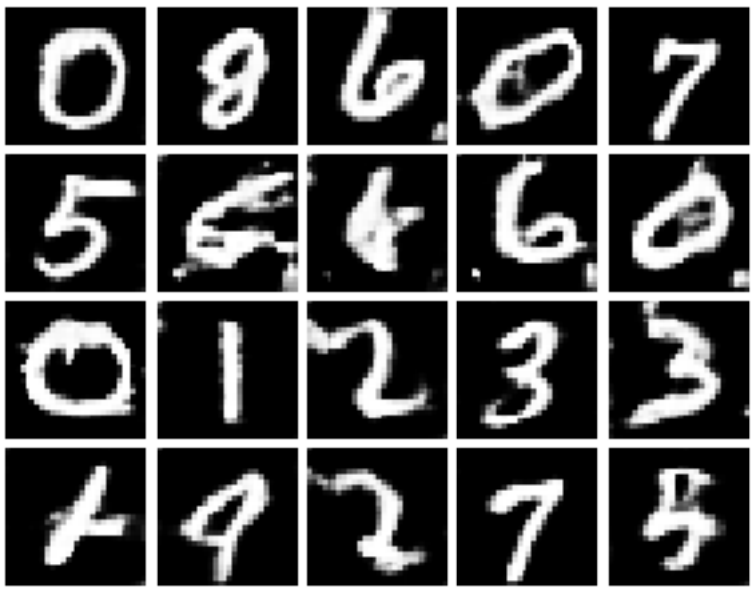
```
accuracy: 0.9238 - val_loss: 0.1070 - val_accuracy: 0.9670
Epoch 2/20
469/469 [==============================] - 3s 6ms/step - loss: 0.1018 -
accuracy: 0.9693 - val_loss: 0.1032 - val_accuracy: 0.9668
Epoch 3/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0740 -
accuracy: 0.9780 - val_loss: 0.0813 - val_accuracy: 0.9757
Epoch 4/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0606 -
accuracy: 0.9822 - val_loss: 0.0752 - val_accuracy: 0.9799
Epoch 5/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0489 -
accuracy: 0.9853 - val_loss: 0.0686 - val_accuracy: 0.9805
Epoch 6/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0438 -
accuracy: 0.9870 - val_loss: 0.0811 - val_accuracy: 0.9801
Epoch 7/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0390 -
accuracy: 0.9888 - val_loss: 0.0802 - val_accuracy: 0.9802
Epoch 8/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0342 -
accuracy: 0.9897 - val_loss: 0.0811 - val_accuracy: 0.9820
Epoch 9/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0308 -
accuracy: 0.9910 - val_loss: 0.0929 - val_accuracy: 0.9817
Epoch 10/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0293 -
accuracy: 0.9917 - val_loss: 0.0865 - val_accuracy: 0.9833
Epoch 11/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0240 -
accuracy: 0.9930 - val_loss: 0.1064 - val_accuracy: 0.9833
Epoch 12/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0275 -
accuracy: 0.9925 - val_loss: 0.0996 - val_accuracy: 0.9830
Epoch 13/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0231 -
accuracy: 0.9934 - val_loss: 0.1003 - val_accuracy: 0.9822
Epoch 14/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0219 -
accuracy: 0.9942 - val_loss: 0.0905 - val_accuracy: 0.9845
Epoch 15/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0195 -
accuracy: 0.9945 - val_loss: 0.1000 - val_accuracy: 0.9838
Epoch 16/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0189 -
accuracy: 0.9948 - val_loss: 0.1193 - val_accuracy: 0.9827
Epoch 17/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0209 -
```

```
accuracy: 0.9948 - val_loss: 0.1135 - val_accuracy: 0.9838
Epoch 18/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0187 -
accuracy: 0.9952 - val_loss: 0.1328 - val_accuracy: 0.9823
Epoch 19/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0157 -
accuracy: 0.9957 - val_loss: 0.1204 - val_accuracy: 0.9836
Epoch 20/20
469/469 [==============================] - 3s 6ms/step - loss: 0.0179 -
accuracy: 0.9954 - val_loss: 0.1145 - val_accuracy: 0.9841
Test loss: 0.11450298875570297
Test accuracy: 0.9840999841690063
```

5.0.3 Verify the trained classifier on the generated samples

Generate samples and visually inspect if the predicted labels on the samples match the actual digits in generated images.

[0]:
```
z = sample_noise(NUM_SAMPLES, NOISE_DIM)
G_sample = G(z)

show_images(G_sample[:20])
plt.show()
```



[0]:
```
G_sample = G_sample.numpy()
G_sample = G_sample.reshape(NUM_SAMPLES, 784)
G_sample = G_sample.astype('float32')
G_sample = deprocess_img(G_sample)
np.argmax(model.predict(G_sample[:20]), axis=-1)
```

[0]:
```
array([0, 8, 6, 0, 7, 5, 6, 6, 6, 0, 0, 1, 2, 3, 3, 2, 9, 2, 7, 5])
```

5.0.4 Implement the inception score

Implement Equation 1 in the reference [3]. Replace expectation in the equation with empirical average of `num_samples` samples. Don't forget the exponentiation at the end. You should get Inception score of at least 8.5

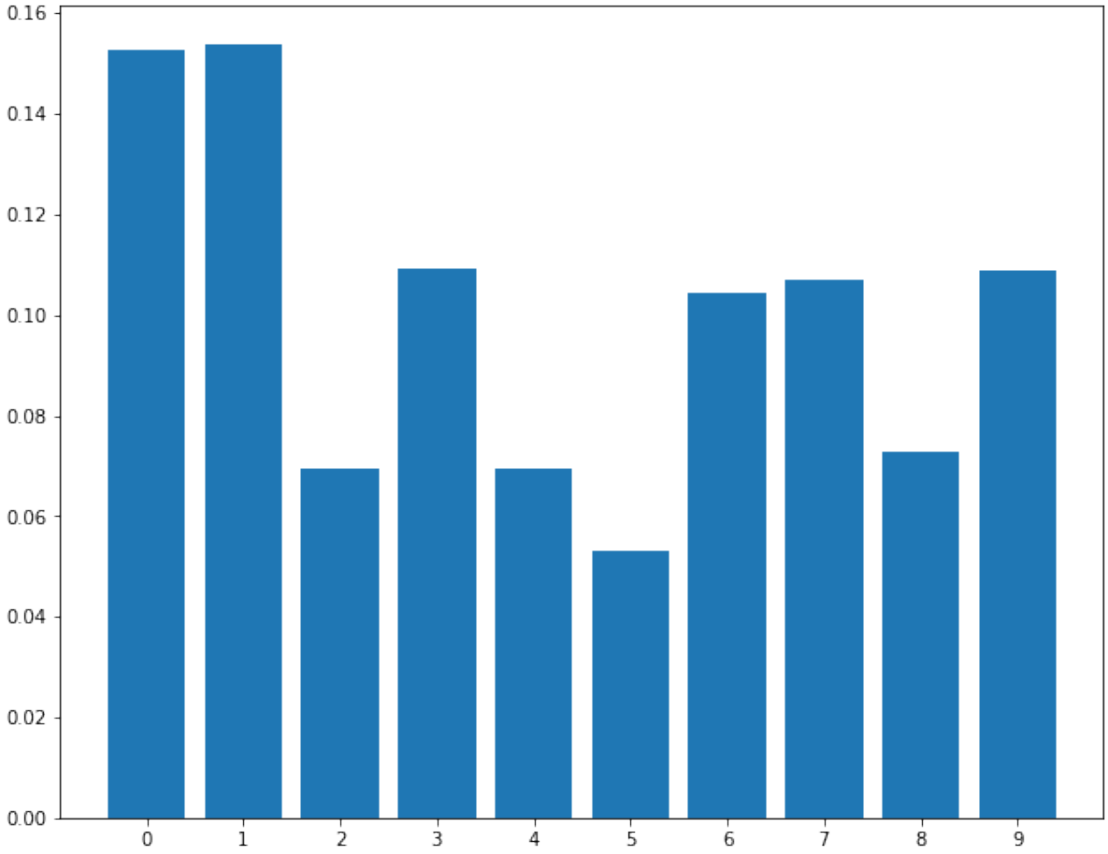
[0]:
```
kld_obj = tf.keras.losses.KLDivergence()
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

image = G_sample
predicted_y = model.predict(image) #p(y|x)
true_y  =  np.ones ((NUM_SAMPLES, 1)) * np.mean(predicted_y, axis = 0)

score = np.exp(kld_obj(predicted_y, true_y))
print('Inception score: {:.4}'.format(score))
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
Inception score: 9.146
```

5.0.5 Plot the histogram of predicted labels

Let's additionally inspect the class diversity of the generated samples.

[0]:
```
plt.hist(np.argmax(model.predict(G_sample), axis=-1),
         bins=np.arange(11)-0.5, rwidth=0.8, density=True)
plt.xticks(range(10))
plt.show()
```



5.1 INLINE QUESTION 1

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.

5.1.1 Your answer:

$ y_{t+1} = y_t + x_t$

$x_{t+1} = x_t - y_{t+1} = -y_t $

$\min_x \max_y f(x, y) = -xy - y^2$

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

5.2 INLINE QUESTION 2

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

5.2.1 Your answer: no. (x_0, y_0) = (x_6, y_6). It will go through step, (alternating, oscillating).

5.3 INLINE QUESTION 3

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.

5.3.1 Your answer: No. Discriminator would have high variance despite good G. It means Generator is generating garbage data that fools Discriminator, so hight loss for D.

white-box-attack

May 27, 2020

0.1 White-box attack exercise

In this exercise, you will implement the following white-box attacks. 1. Fast Gradient Sign Method (FGSM) 2. Projected Gradient Descent (PGD)

[0]:
```
from google.colab import drive

drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# 'cs231n' folder containing the '.py', 'classifiers' and 'datasets'
# folders.
# e.g. 'cs231n/assignments/assignment1/cs231n'
FOLDERNAME = "hw6_files"

assert FOLDERNAME is not None, "[!] Enter the foldername."

%cd /content/drive/My\ Drive/$FOLDERNAME/data
!bash download_data.sh
%cd ../models
!bash download_models.sh
%cd ..
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id
=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redire
ct_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20http
s%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.c
om%2fauth%2fdrive%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.reado
nly%20https%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly

Enter your authorization code:
ůůůůůůůůůů
Mounted at /content/drive
/content/drive/My Drive/hw6_files/data
--------------------------------
Cifar-10 dataset already exits
--------------------------------
/content/drive/My Drive/hw6_files/models
```

```
--------------------------------------
Naturally-trained model already exists
--------------------------------------
----------------------------------------
Adversarially-trained model already exists
----------------------------------------
/content/drive/My Drive/hw6_files
```

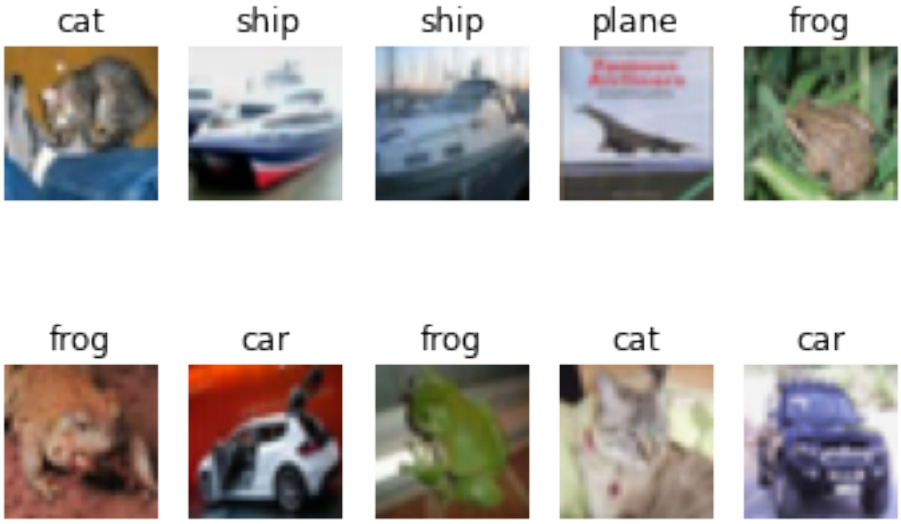
[0]:
```
import math
import matplotlib.pyplot as plt
import numpy as np
import tensorflow.compat.v1 as tf
tf.experimental.output_all_intermediates(True)
tf.disable_eager_execution()

from cifar10_input import CIFAR10Data
from model import Model

%load_ext autoreload
%autoreload 2
```

0.2 Loading Cifar-10 test dataset

[0]:
```
# Clear previously loaded data
try:
   del eval_data
   print('Clearing previously loaded data')
except:
   pass

# Load Cifar-10 test set
print('Loading Cifar-10 test dataset')
DATA_DIR = './data/cifar-10-batches-py'
eval_data = CIFAR10Data(DATA_DIR).eval_data

# Print the number of samples in the test set
print('The number of the test data: {}'.format(eval_data.n))

# Print the first 10 samples
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)
    sample_image = eval_data.xs[i]
    sample_label = eval_data.ys[i]
    plt.imshow(sample_image.astype('uint8'))
    plt.axis('off')
```

```
    plt.title(classes[sample_label])
```

```
Loading Cifar-10 test dataset
The number of the test data: 10000
```



0.3 Restoring a naturally-trained ResNet classifier

[0]:
```
# Reset all graphs and session
print('Clearing all graphs')
tf.reset_default_graph()

# Create a naturally-trained model
print('Creating a ResNet model')
model = Model(mode='eval')
sess = tf.Session()

# Restore parameters
print('Restoring parameters')
NAT_MODEL_DIR = './models/naturally_trained'
model_file = tf.train.latest_checkpoint(NAT_MODEL_DIR)

var_list = {}
with tf.variable_scope('', reuse=True):
  for var in tf.train.list_variables(model_file)[1:]:
    if 'Momentum' not in var[0]:
      var_list[var[0]] = tf.get_variable(name=var[0].replace('BatchNorm',␣
 ↪'batch_normalization'))
```

```
saver = tf.train.Saver(var_list=var_list)
saver.restore(sess, model_file)
```

```
Clearing all graphs
Creating a ResNet model
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/tensorflow/python/ops/resource_variable_ops.py:1666: calling
BaseResourceVariable.__init__ (from tensorflow.python.ops.resource_variable_ops)
with constraint is deprecated and will be removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
WARNING:tensorflow:From /content/drive/My Drive/hw6_files/model.py:108:
batch_normalization (from tensorflow.python.layers.normalization) is deprecated
and will be removed in a future version.
Instructions for updating:
Use keras.layers.BatchNormalization instead.  In particular,
`tf.control_dependencies(tf.GraphKeys.UPDATE_OPS)` should not be used (consult
the `tf.keras.layers.BatchNormalization` documentation).
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/tensorflow/python/layers/normalization.py:336: Layer.apply (from
tensorflow.python.keras.engine.base_layer_v1) is deprecated and will be removed
in a future version.
Instructions for updating:
Please use `layer.__call__` method instead.
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/tensorflow/python/util/deprecation.py:507: UniformUnitScaling.__init__
(from tensorflow.python.ops.init_ops) is deprecated and will be removed in a
future version.
Instructions for updating:
Use tf.initializers.variance_scaling instead with distribution=uniform to get
equivalent behavior.
Restoring parameters
INFO:tensorflow:Restoring parameters from
./models/naturally_trained/checkpoint-70000
```

0.4 Evaluating the model

Before implementing attack methods, we have to evaluate the model for the following reasons. 1. To check whether the model is successfuly restored. 2. To get samples that are correctly classified. We don't have to attack misclassified samples.

Note that the indices of the first 100 samples are stored in a variable named `correct_indices`. You will use it later.

[0]:
```
def evaluate(model, sess, data, indices, attack_method=None):
    """
    Given the data specified by the indices, evaluate the model.

    Args:
```

```
        model: TensorFlow model
        sess: TensorFlow session
        data: Cifar-10 test dataset
        indices: Indices that specifies the data
        attack_method (optional): Instance of attack method, If it is not None,␣
↪the attack method is applied before
        evaluation.

    Returns:
        correct_prediction: NumPy array with the same shape as the indices.␣
↪Given an index, 1 if the corresponding
        sample is correctly classifed, 0 otherwise.
    """

    correct_predictions = np.zeros([0], np.int32)

    num_images = len(indices)
    batch_size = 100
    num_batches = int(math.ceil(num_images/batch_size))

    # Run batches
    for batch in range(num_batches):
        # Construct batch
        bstart = batch*batch_size
        bend = min(bstart+batch_size, num_images)
        image_batch = data.xs[indices[bstart:bend]]
        image_batch = np.int32(image_batch)
        label_batch = data.ys[indices[bstart:bend]]
        # Attack batch
        if attack_method is not None:
            image_batch = attack_method.perturb(image_batch, label_batch, sess)
        # Evaluate batch
        feed_dict = {
            model.x_input: image_batch,
            model.y_input: label_batch
        }
        correct_prediction = sess.run(model.correct_prediction,␣
↪feed_dict=feed_dict)
        correct_predictions = np.concatenate([correct_predictions,␣
↪correct_prediction], axis=0)

    return correct_predictions

# Evaluate the naturally-trained model on the first 1000 samples in the test␣
↪dataset
indices = np.arange(0, 1000)
```

```
print('Evaluating naturally-trained model')
correct_predictions = evaluate(model, sess, eval_data, indices)
accuracy = np.mean(correct_predictions)*100
print('Accuracy: {:.1f}%'.format(accuracy))

# Select the first 100 samples that are correctly classified.
correct_indices = np.where(correct_predictions==1)[0][:100]
```

```
Evaluating naturally-trained model
Accuracy: 96.2%
```

0.5 Fast Gradient Sign Method (FGSM)

Now, you will implement Fast Gradient Sign Method under ℓ_∞ constraint, the first method of generating adversarial examples proposed by Goodfellow et al.. The algorithm is as follows.

$x_{adv} = x + \epsilon \cdot \text{sgn}(\nabla_x L(x, y, \theta))$

where x, y are an image and the corresponding label, L is a loss function, and ϵ is a maximum perturbation. Usually, Cross-Entropy loss is used for L. However, there might be many possible choices for L, such as Carlini-Wagner loss (https://arxiv.org/abs/1608.04644)

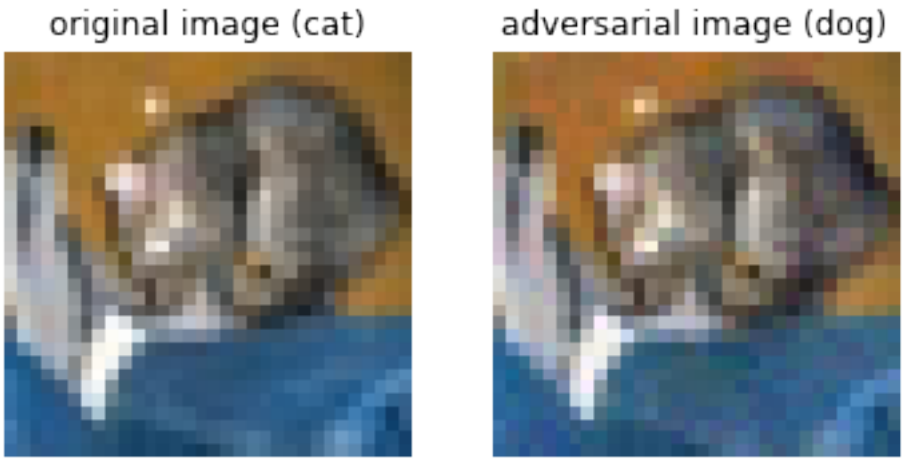
Your code for this section will all be written inside `attacks/fgsm_attack.py`.

[0]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id
=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redire
ct_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20http
s%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.c
om%2fauth%2fdrive%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.reado
nly%20https%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly

Enter your authorization code:
··········
Mounted at /content/drive
```

[0]:
```
# First implement Fast Gradient Sign Method.
# Open attacks/fgsm_attack.py and follow instructions in the file.

from attacks.fgsm_attack import FGSMAttack

# Check if your implementation is correct.

# Default attack setting
epsilon = 8
loss_func = 'cw'

# Create an instance of FGSMAttack
fgsm_attack = FGSMAttack(model, epsilon, loss_func)
```

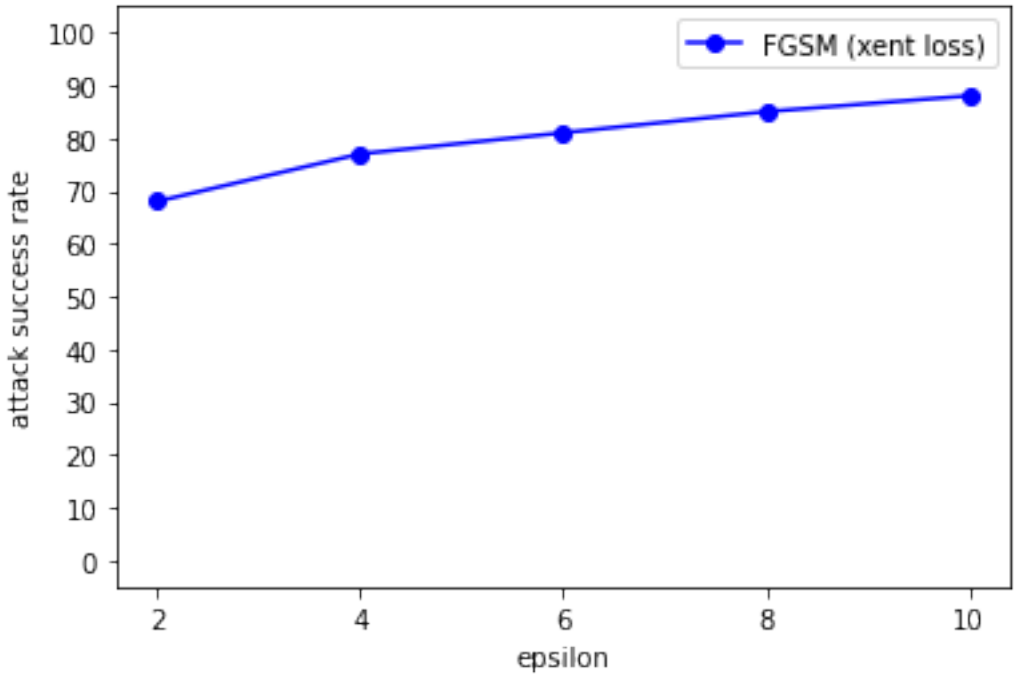
```
# Run FGSM attack on a sample
sample_image = eval_data.xs[correct_indices[0]]
sample_image = np.int32(sample_image) # please convert uint8 to int32
sample_image = np.expand_dims(sample_image, axis=0)
sample_label = eval_data.ys[correct_indices[0]]
sample_label = np.expand_dims(sample_label, axis=0)
sample_adv_image = fgsm_attack.perturb(sample_image, sample_label, sess)
sample_adv_label = sess.run(model.predictions, feed_dict={model.x_input:␣
 ↪sample_adv_image})

# Check if the adversarial image is valid
assert np.amax(np.abs(sample_image-sample_adv_image)) <= epsilon
assert np.amin(sample_adv_image) >= 0
assert np.amax(sample_adv_image) <= 255

# Plot the original image
plt.subplot(1, 2, 1)
plt.imshow(sample_image[0, ...].astype('uint8'))
plt.axis('off')
plt.title('original image ({})'.format(classes[sample_label[0]]))

# Plot the adversarial image
plt.subplot(1, 2, 2)
plt.imshow(sample_adv_image[0, ...].astype('uint8'))
plt.axis('off')
plt.title('adversarial image ({})'.format(classes[sample_adv_label[0]]));
```



[0]:
```
loss_func = 'xent'
epsilons = [2, 4, 6, 8, 10]
attack_success_rates = []
```

```
for epsilon in epsilons:
    fgsm_attack = FGSMAttack(model, epsilon, loss_func)
    correct_predictions = evaluate(model, sess, eval_data, correct_indices,␣
 ↪attack_method=fgsm_attack)
    attack_success_rate = np.mean(1-correct_predictions)*100
    attack_success_rates.append(attack_success_rate)
    print('Epsilon: {}, Attack success rate: {:.1f}%'.format(epsilon,␣
 ↪attack_success_rate))

plt.plot(epsilons, attack_success_rates, '-bo', label='FGSM (xent loss)')
plt.ylim(-5, 105)
plt.xticks(epsilons)
plt.yticks(np.arange(0, 110, 10))
plt.xlabel('epsilon')
plt.ylabel('attack success rate')
plt.legend();
```

```
Epsilon: 2, Attack success rate: 68.0%
Epsilon: 4, Attack success rate: 77.0%
Epsilon: 6, Attack success rate: 81.0%
Epsilon: 8, Attack success rate: 85.0%
Epsilon: 10, Attack success rate: 88.0%
```



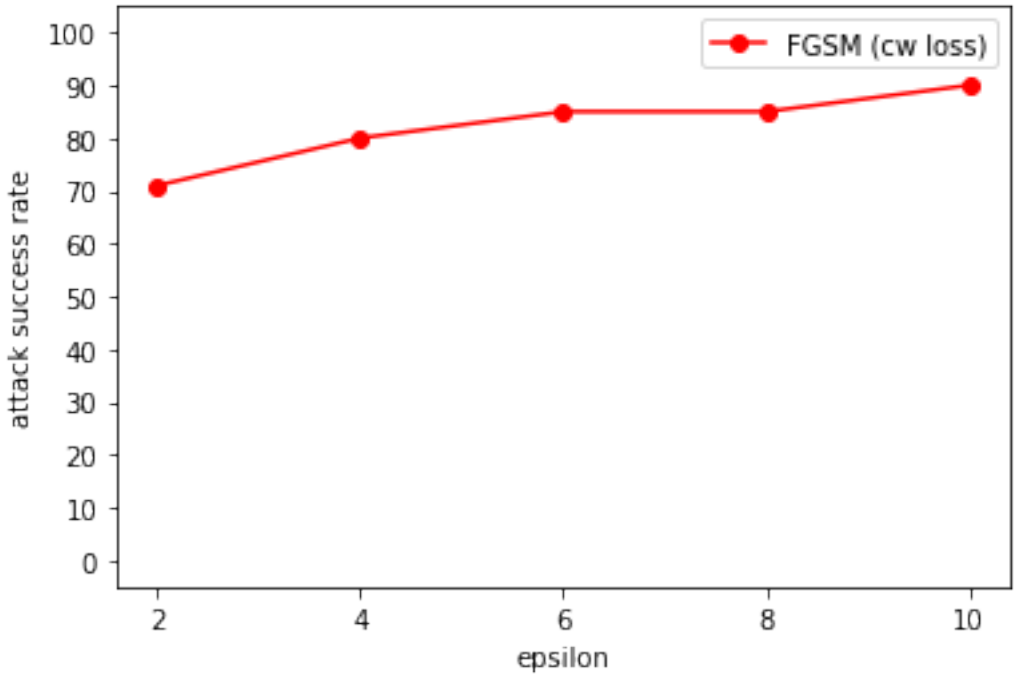
0.6 Evaluating the performance of FGSM with Carlini-Wagner loss

In this section, you will evaluate the performance of FGSM using Carlini-Wagner loss. Repeat the procedure in the previous section and compare the results.

If correctly implemented, the success rate will be 80% or higher on epsilon 8.

[0]:
```
loss_func = 'cw'
epsilons = [2, 4, 6, 8, 10]
attack_success_rates = []

for epsilon in epsilons:
    fgsm_attack = FGSMAttack(model, epsilon, loss_func)
    correct_predictions = evaluate(model, sess, eval_data, correct_indices,␣
 ↪attack_method=fgsm_attack)
    attack_success_rate = np.mean(1-correct_predictions)*100
    attack_success_rates.append(attack_success_rate)
    print('Epsilon: {}, Attack success rate: {:.1f}%'.format(epsilon,␣
 ↪attack_success_rate))

plt.plot(epsilons, attack_success_rates, '-ro', label='FGSM (cw loss)')
plt.ylim(-5, 105)
plt.xticks(epsilons)
plt.yticks(np.arange(0, 110, 10))
plt.xlabel('epsilon')
plt.ylabel('attack success rate')
plt.legend();
```

```
Epsilon: 2, Attack success rate: 71.0%
Epsilon: 4, Attack success rate: 80.0%
Epsilon: 6, Attack success rate: 85.0%
Epsilon: 8, Attack success rate: 85.0%
Epsilon: 10, Attack success rate: 90.0%
```



0.7 Projected Gradient Descent (PGD)

Next, you will implement Projected Gradient Descent under ℓ_∞ constraint, which is considered as the strongest white-box attack method proposed by Madry et al.. The algorithm is as follows.

$x^0 = x$

$x^{t+1} = \prod_{B_\infty(x,\epsilon)}[x^t + \alpha \cdot \text{sgn}(\nabla_x L(x^t, y, \theta))]$

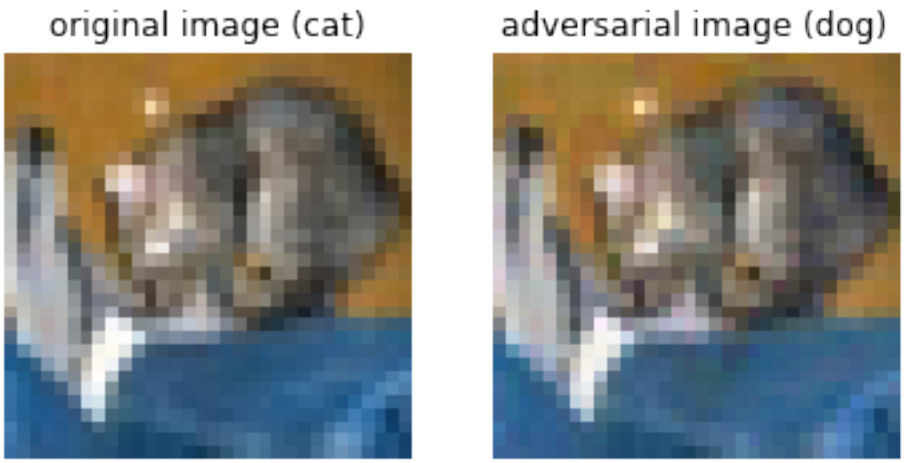
where x, y are an image and the corresponding label, L is a loss function, α is a step size, ϵ is a maximum perturbation, and $B_\infty(x, \epsilon)$ is a ℓ_∞ ball of radius ϵ centered at x.

Your code for this section will all be written inside `attacks/pgd_attack.py`.

[0]:
```
# First implement Projected Gradient Descent.
# Open attacks/pgd_attack.py and follow instructions in the file.

from attacks.pgd_attack import PGDAttack

# Check if your implementation is correct.

# Default attack setting
epsilon = 8
step_size = 2
num_steps = 20
loss_func = 'xent'

# Create an instance of FGSMAttack
pgd_attack = PGDAttack(model, epsilon, step_size, num_steps, loss_func)
```

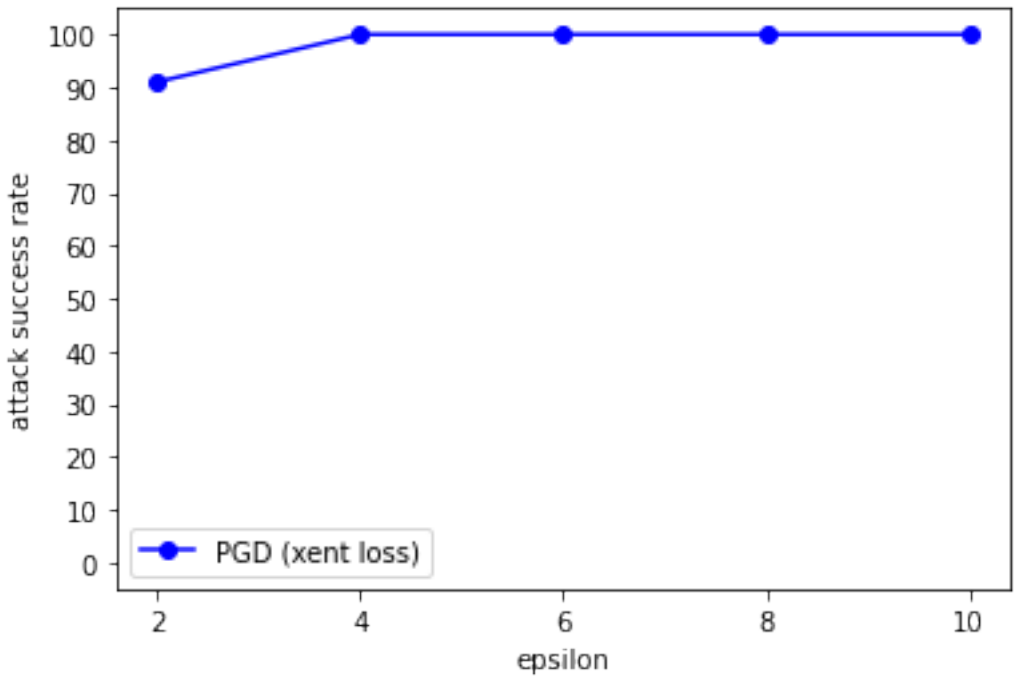
```
# Run PGD attack on a sample
sample_image = eval_data.xs[correct_indices[0]]
sample_image = np.int32(sample_image) # please convert uint8 to int32
sample_image = np.expand_dims(sample_image, axis=0)
sample_label = eval_data.ys[correct_indices[0]]
sample_label = np.expand_dims(sample_label, axis=0)
sample_adv_image = pgd_attack.perturb(sample_image, sample_label, sess)
sample_adv_label = sess.run(model.predictions, feed_dict={model.x_input:␣
 ↪sample_adv_image})

# Check if the adversarial image is valid
assert np.amax(np.abs(sample_image-sample_adv_image)) <= epsilon
assert np.amin(sample_adv_image) >= 0
assert np.amax(sample_adv_image) <= 255

# Plot the original image
plt.subplot(1, 2, 1)
plt.imshow(sample_image[0, ...].astype('uint8'))
plt.axis('off')
plt.title('original image ({})'.format(classes[sample_label[0]]))

# Plot the adversarial image
plt.subplot(1, 2, 2)
plt.imshow(sample_adv_image[0, ...].astype('uint8'))
plt.axis('off')
plt.title('adversarial image ({})'.format(classes[sample_adv_label[0]]));
```

```
WARNING:tensorflow:From /content/drive/My
Drive/hw6_files/attacks/fgsm_attack.py:27: softmax_cross_entropy_with_logits
(from tensorflow.python.ops.nn_ops) is deprecated and will be removed in a
future version.
Instructions for updating:

Future major versions of TensorFlow will allow gradients to flow
into the labels input on backprop by default.

See `tf.nn.softmax_cross_entropy_with_logits_v2`.
```



[0]:
```
step_size = 2
num_steps = 20
loss_func = 'xent'
epsilons = [2, 4, 6, 8, 10]
attack_success_rates = []

for epsilon in epsilons:
    pgd_attack = PGDAttack(model, epsilon, step_size, num_steps, loss_func)
    correct_predictions = evaluate(model, sess, eval_data, correct_indices,␣
 ↪attack_method=pgd_attack)
    attack_success_rate = np.mean(1-correct_predictions)*100
    attack_success_rates.append(attack_success_rate)
    print('Epsilon: {}, Attack success rate: {:.1f}%'.format(epsilon,␣
 ↪attack_success_rate))

plt.plot(epsilons, attack_success_rates, '-bo', label='PGD (xent loss)')
plt.ylim(-5, 105)
plt.xticks(epsilons)
plt.yticks(np.arange(0, 110, 10))
plt.xlabel('epsilon')
plt.ylabel('attack success rate')
plt.legend();
```

```
Epsilon: 2, Attack success rate: 91.0%
Epsilon: 4, Attack success rate: 100.0%
Epsilon: 6, Attack success rate: 100.0%
Epsilon: 8, Attack success rate: 100.0%
Epsilon: 10, Attack success rate: 100.0%
```



0.8 Evaluating the performance of PGD with Carlini-Wagner loss

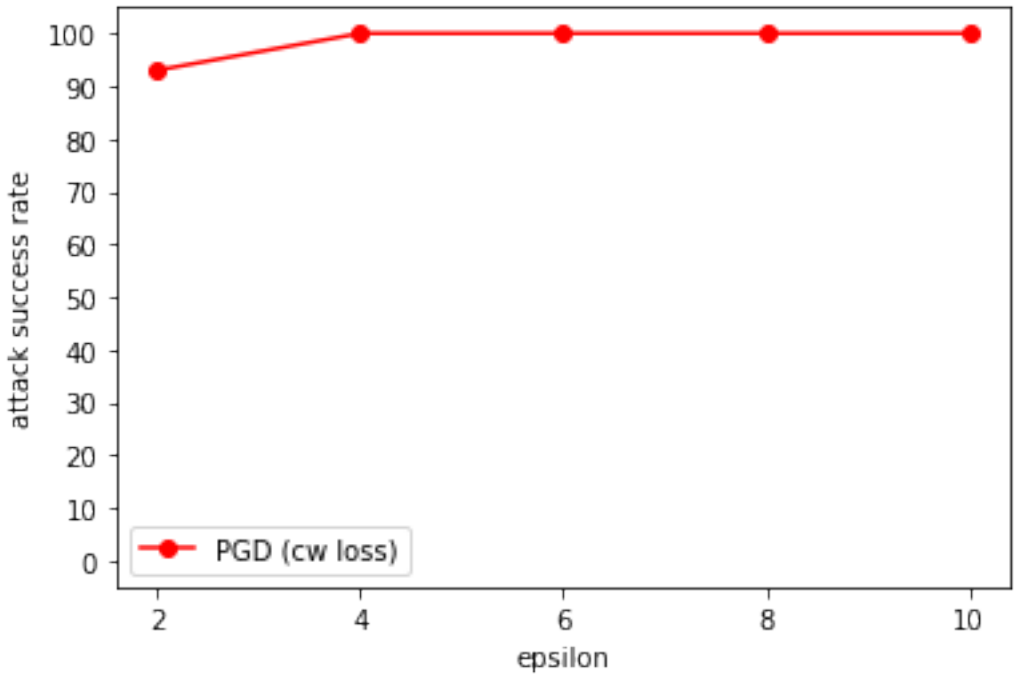
In this section, you will evaluate the performance of PGD using Carlini-Wagner loss. Repeat the procedure in the previous section and compare the results.

If correctly implemented, the success rate will be 100% on epsilon 8.

[0]:
```
step_size = 2
num_steps = 20
loss_func = 'cw'
epsilons = [2, 4, 6, 8, 10]
attack_success_rates = []

for epsilon in epsilons:
    pgd_attack = PGDAttack(model, epsilon, step_size, num_steps, loss_func)
    correct_predictions = evaluate(model, sess, eval_data, correct_indices,␣
 ↪attack_method=pgd_attack)
    attack_success_rate = np.mean(1-correct_predictions)*100
    attack_success_rates.append(attack_success_rate)
    print('Epsilon: {}, Attack success rate: {:.1f}%'.format(epsilon,␣
 ↪attack_success_rate))

plt.plot(epsilons, attack_success_rates, '-ro', label='PGD (cw loss)')
plt.ylim(-5, 105)
plt.xticks(epsilons)
plt.yticks(np.arange(0, 110, 10))
```

```
plt.xlabel('epsilon')
plt.ylabel('attack success rate')
plt.legend();
```

```
Epsilon: 2, Attack success rate: 93.0%
Epsilon: 4, Attack success rate: 100.0%
Epsilon: 6, Attack success rate: 100.0%
Epsilon: 8, Attack success rate: 100.0%
Epsilon: 10, Attack success rate: 100.0%
```



Inline Question
Which is better, Cross-Entropy loss or Carlini-Wagner loss?
Your Answer It seems to be similar.

0.9 Attacks on adversarially-trained model

As you can see, naturally-trained neural networks are vulnerable to adversarial attacks. There are several ways to improve adversarial robustness of neural networks. One example is adversarial training, which uses adversarial samples to train a neural network. It constitutes the current state-of-the-art in the adversarial robustness.

PGD adversarial training, proposed by Madry et al., utilizes Projected Gradient Descent to train a network. It has been shown that PGD adversarial training on MNIST and Cifar-10 can defend white-box attack successfully.

[0]:
```
# Reset all graphs and session
print('Clearing all graphs')
```

```
tf.reset_default_graph()

# Create a naturally-trained model
print('Creating a ResNet model')
model = Model(mode='eval')
sess = tf.Session()

# Restore parameters
print('Restoring parameters')
ADV_MODEL_DIR = './models/adv_trained'
model_file = tf.train.latest_checkpoint(ADV_MODEL_DIR)

var_list = {}
with tf.variable_scope('', reuse=True):
  for var in tf.train.list_variables(model_file)[1:]:
    if 'Momentum' not in var[0]:
      var_list[var[0]] = tf.get_variable(name=var[0].replace('BatchNorm',␣
 ↪'batch_normalization'))

saver = tf.train.Saver(var_list=var_list)
saver.restore(sess, model_file)
```

```
Clearing all graphs
Creating a ResNet model
Restoring parameters
INFO:tensorflow:Restoring parameters from ./models/adv_trained/checkpoint-70000
```

0.10 Evaluating the model

Before implementing attack methods, we have to evaluate the model for the following reasons. 1. To check whether the model is successfuly restored. 2. To get samples that are correctly classified. We don't have to attack misclassified samples.

Note that the indices of the first 100 samples are stored in a variable named `correct_indices`. You will use it later.

[0]:
```
# Evaluate the adversarially-trained model on the first 1000 samples in the␣
 ↪test dataset
indices = np.arange(0, 1000)

print('Evaluating adversarially-trained model')
correct_predictions = evaluate(model, sess, eval_data, indices)
accuracy = np.mean(correct_predictions)*100
print('Accuracy: {:.1f}%'.format(accuracy))

# Select the first 100 samples that are correctly classified.
correct_indices = np.where(correct_predictions==1)[0][:100]
```

```
Evaluating adversarially-trained model
```

```
Accuracy: 87.3%
```

Inline Question

Is the accuracy of adversarially-trained model higher than that of naturally-trained model, or lower? Explain why they are different.

Your answer

Lower. Adversarially trained model is trained with some adversarial samples to improve robustness to adversarial data. The accuracy on normal data would be lower compared to that of normally trained model.

Useful material

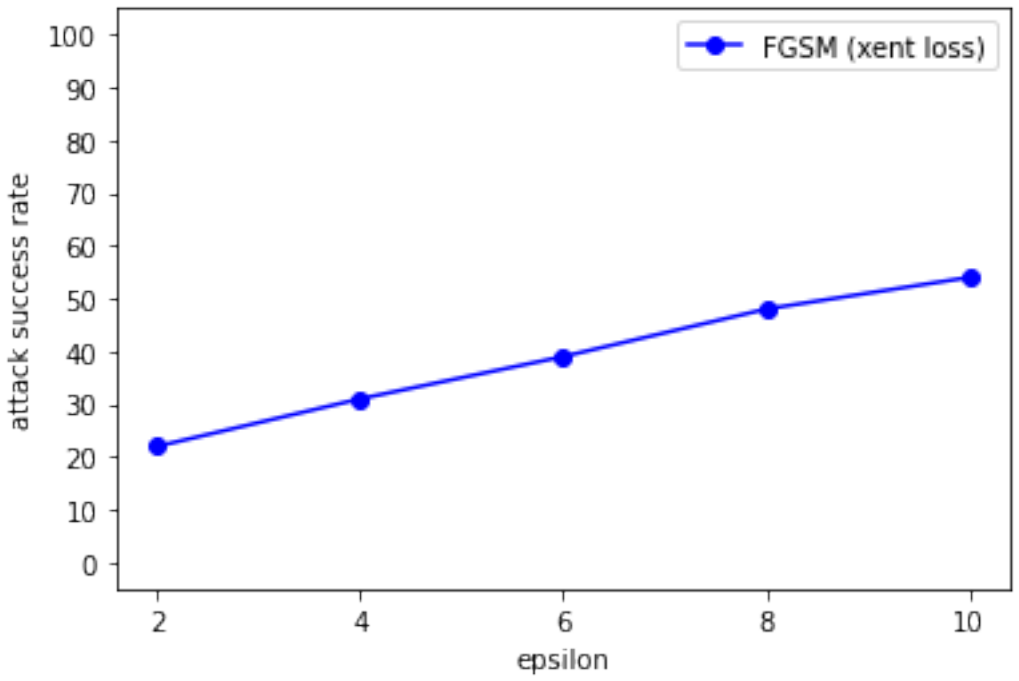
For those who are curious about this phenomenon, see https://arxiv.org/abs/1805.12152.

0.11 Evaluating the performance of FGSM on adversarially-trained model

Now, we will evaluate the the performance of FGSM on adversarially-trained model. In this section, you will use Cross-Entropy loss as L.

[0]:
```
loss_func = 'xent'
epsilons = [2, 4, 6, 8, 10]
fgsm_attack_success_rates = []

for epsilon in epsilons:
    fgsm_attack = FGSMAttack(model, epsilon, loss_func)
    correct_predictions = evaluate(model, sess, eval_data, correct_indices,␣
 ↪attack_method=fgsm_attack)
    attack_success_rate = np.mean(1-correct_predictions)*100
    fgsm_attack_success_rates.append(attack_success_rate)
    print('Epsilon: {}, Attack success rate: {:.1f}%'.format(epsilon,␣
 ↪attack_success_rate))

plt.plot(epsilons, fgsm_attack_success_rates, '-bo', label='FGSM (xent loss)')
plt.ylim(-5, 105)
plt.xticks(epsilons)
plt.yticks(np.arange(0, 110, 10))
plt.xlabel('epsilon')
plt.ylabel('attack success rate')
plt.legend();
```

```
Epsilon: 2, Attack success rate: 22.0%
Epsilon: 4, Attack success rate: 31.0%
Epsilon: 6, Attack success rate: 39.0%
Epsilon: 8, Attack success rate: 48.0%
Epsilon: 10, Attack success rate: 54.0%
```



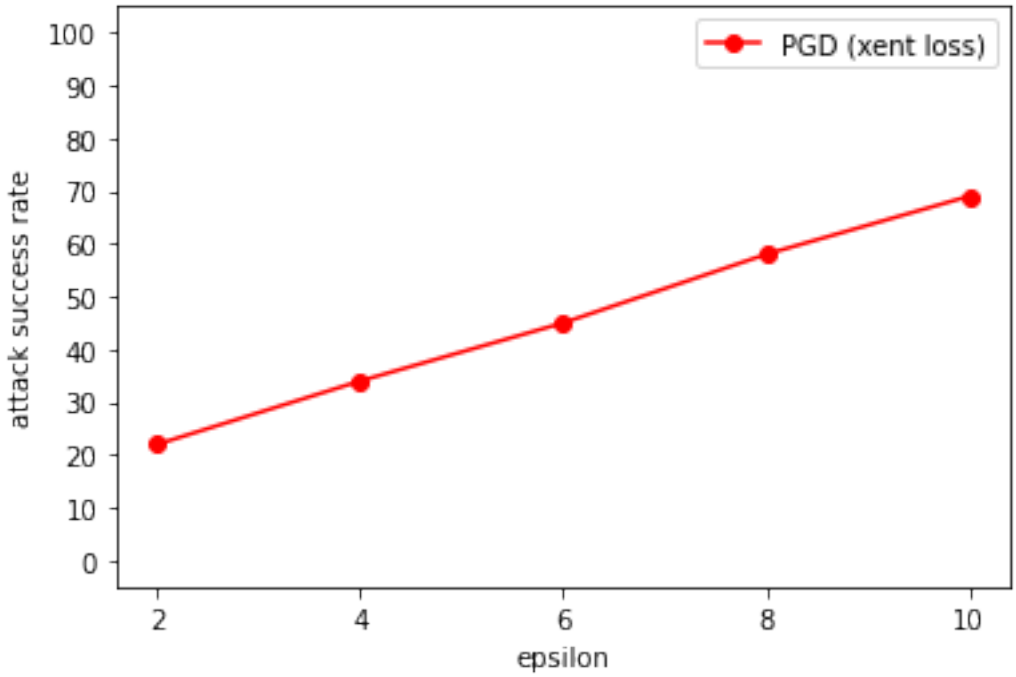
0.12 Evaluating the performance of PGD on adversarially-trained model

Now, we will evaluate the the performance of PGD on adversarially-trained model. In this section, you will use Cross-Entropy loss as L.

[0]:
```
step_size = 2
num_steps = 20
loss_func = 'xent'
epsilons = [2, 4, 6, 8, 10]
pgd_attack_success_rates = []

for epsilon in epsilons:
    pgd_attack = PGDAttack(model, epsilon, step_size, num_steps, loss_func)
    correct_predictions = evaluate(model, sess, eval_data, correct_indices,␣
 ↪attack_method=pgd_attack)
    attack_success_rate = np.mean(1-correct_predictions)*100
    pgd_attack_success_rates.append(attack_success_rate)
    print('Epsilon: {}, Attack success rate: {:.1f}%'.format(epsilon,␣
 ↪attack_success_rate))

plt.plot(epsilons, pgd_attack_success_rates, '-ro', label='PGD (xent loss)')
plt.ylim(-5, 105)
plt.xticks(epsilons)
plt.yticks(np.arange(0, 110, 10))
plt.xlabel('epsilon')
```

```
plt.ylabel('attack success rate')
plt.legend();
```

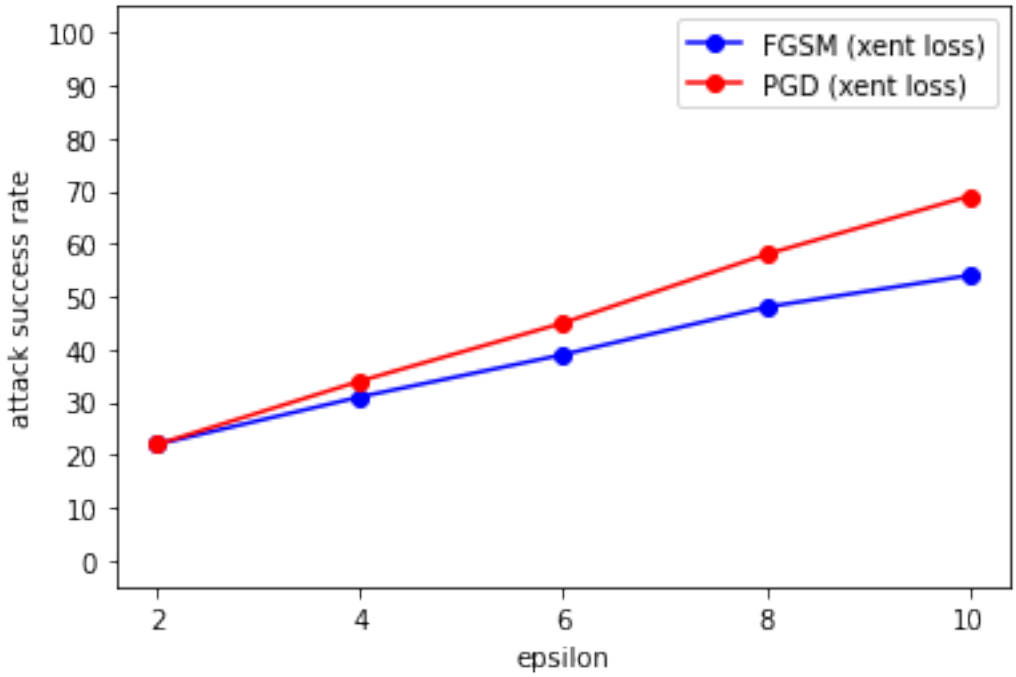
```
Epsilon: 2, Attack success rate: 22.0%
Epsilon: 4, Attack success rate: 34.0%
Epsilon: 6, Attack success rate: 45.0%
Epsilon: 8, Attack success rate: 58.0%
Epsilon: 10, Attack success rate: 69.0%
```



0.13 Comparing the performance of FGSM and PGD

Finally, we compare the performace of FGSM and PGD on adversarially-trained model. Just overlay the plots drawn in the two previous sections.

[0]:
```
epsilons = [2, 4, 6, 8, 10]

plt.plot(epsilons, fgsm_attack_success_rates, '-bo', label='FGSM (xent loss)')
plt.plot(epsilons, pgd_attack_success_rates, '-ro', label='PGD (xent loss)')
plt.ylim(-5, 105)
plt.xticks(epsilons)
plt.yticks(np.arange(0, 110, 10))
plt.xlabel('epsilon')
plt.ylabel('attack success rate')
plt.legend();
```



Inline question

Describe the result above.

Your answer

Attack success rate on adversarially-trained model is remarkably lower than that on normally-trained model. Also, the attack success rate increases as epsilon size increases.

Attack success rate of PGD is slightly higher than that of FGSM. The performance of PGD seems to be better than that of FGSM.

black-box-attack

May 27, 2020

0.1 Black-box attack exercise

In this exercise, you will implement the following black-box attack. 1. NES attack (NES)

[1]:
```
from google.colab import drive

drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# 'cs231n' folder containing the '.py', 'classifiers' and 'datasets'
# folders.
# e.g. 'cs231n/assignments/assignment1/cs231n'
FOLDERNAME = "hw6_files"

assert FOLDERNAME is not None, "[!] Enter the foldername."

%cd /content/drive/My\ Drive/$FOLDERNAME/data
!bash download_data.sh
%cd ../models
!bash download_models.sh
%cd ..
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id
=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redire
ct_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20http
s%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.c
om%2fauth%2fdrive%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.reado
nly%20https%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly

Enter your authorization code:
ůůůůůůůůůů
Mounted at /content/drive
/content/drive/My Drive/hw6_files/data
------------------------------
Cifar-10 dataset already exits
------------------------------
/content/drive/My Drive/hw6_files/models
-------------------------------------
```

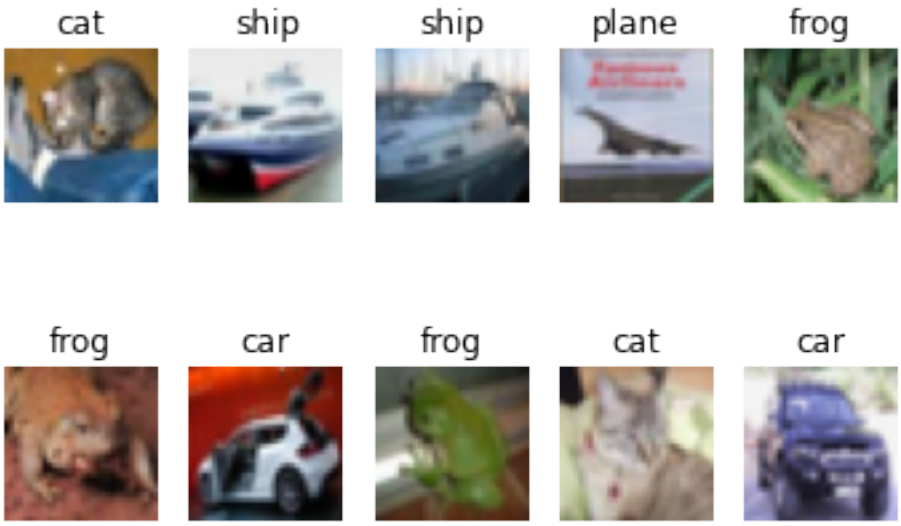
```
Naturally-trained model already exists
--------------------------------------
----------------------------------------
Adversarially-trained model already exists
----------------------------------------
/content/drive/My Drive/hw6_files
```

[0]:
```
import math
import matplotlib.pyplot as plt
import numpy as np
import tensorflow.compat.v1 as tf
tf.disable_eager_execution()

from cifar10_input import CIFAR10Data
from model import Model

%load_ext autoreload
%autoreload 2
```

0.2 Loading Cifar-10 test dataset

[3]:
```
# Clear previously loaded data
try:
   del eval_data
   print('Clearing previously loaded data')
except:
   pass

# Load Cifar-10 test set
print('Loading Cifar-10 test dataset')
DATA_DIR = './data/cifar-10-batches-py'
eval_data = CIFAR10Data(DATA_DIR).eval_data

# Print the number of samples in the test set
print('The number of the test data: {}'.format(eval_data.n))

# Print the first 10 samples
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)
    sample_image = eval_data.xs[i]
    sample_label = eval_data.ys[i]
    plt.imshow(sample_image.astype('uint8'))
    plt.axis('off')
    plt.title(classes[sample_label])
```

```
Loading Cifar-10 test dataset
The number of the test data: 10000
```



0.3 Restoring a naturally-trained ResNet classifier

[4]:
```
# Reset all graphs
print('Clearing all graphs')
tf.reset_default_graph()

# Create a naturally-trained model
print('Creating a ResNet model')
model = Model(mode='eval')
sess = tf.Session()

# Restore parameters
print('Restoring parameters')
NAT_MODEL_DIR = './models/naturally_trained'
model_file = tf.train.latest_checkpoint(NAT_MODEL_DIR)

var_list = {}
with tf.variable_scope('', reuse=True):
  for var in tf.train.list_variables(model_file)[1:]:
    if 'Momentum' not in var[0]:
      var_list[var[0]] = tf.get_variable(name=var[0].replace('BatchNorm',␣
 ↪'batch_normalization'))

saver = tf.train.Saver(var_list=var_list)
saver.restore(sess, model_file)
```

```
Clearing all graphs
Creating a ResNet model
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/tensorflow/python/ops/resource_variable_ops.py:1666: calling
BaseResourceVariable.__init__ (from tensorflow.python.ops.resource_variable_ops)
with constraint is deprecated and will be removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
WARNING:tensorflow:From /content/drive/My Drive/hw6_files/model.py:108:
batch_normalization (from tensorflow.python.layers.normalization) is deprecated
and will be removed in a future version.
Instructions for updating:
Use keras.layers.BatchNormalization instead.  In particular,
`tf.control_dependencies(tf.GraphKeys.UPDATE_OPS)` should not be used (consult
the `tf.keras.layers.BatchNormalization` documentation).
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/tensorflow/python/layers/normalization.py:336: Layer.apply (from
tensorflow.python.keras.engine.base_layer_v1) is deprecated and will be removed
in a future version.
Instructions for updating:
Please use `layer.__call__` method instead.
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/tensorflow/python/util/deprecation.py:507: UniformUnitScaling.__init__
(from tensorflow.python.ops.init_ops) is deprecated and will be removed in a
future version.
Instructions for updating:
Use tf.initializers.variance_scaling instead with distribution=uniform to get
equivalent behavior.
Restoring parameters
INFO:tensorflow:Restoring parameters from
./models/naturally_trained/checkpoint-70000
```

0.4 Evaluating the model

Before implementing attack methods, we have to evaluate the model for the following reasons. 1. To check whether the model is successfuly restored. 2. To get samples that are correctly classified. We don't have to attack misclassified samples.

[5]:
```
def evaluate(model, sess, data, indices, attack_method=None):
    """
    Given the data specified by the indices, evaluate the model.

    Args:
        model: TensorFlow model
        sess: TensorFlow session
        data: Cifar-10 test dataset
        indices: The indices that specify the data
```

```
        attack_method (optional): Instance of an attack method, If it is not␣
↪None, the attack method is applied before
        evaluation.

    Return
        correct_prediction: NumPy array with the same shape as the indices.␣
↪Given an index, 1 if the corresponding
        sample is correctly classifed, 0 otherwise.
    """

    correct_predictions = np.zeros([0], np.int32)

    num_images = len(indices)
    # batch_size set to 1 for NES attack
    batch_size = 1
    num_batches = int(math.ceil(num_images/batch_size))

    # Run batches
    for batch in range(num_batches):
        # Construct batch
        bstart = batch*batch_size
        bend = min(bstart+batch_size, num_images)
        image_batch = data.xs[indices[bstart:bend]]
        image_batch = np.int32(image_batch)
        label_batch = data.ys[indices[bstart:bend]]
        # Attack batch
        if attack_method is not None:
            image_batch = attack_method.perturb(image_batch, label_batch, sess)
        # Evaluate batch
        feed_dict = {
            model.x_input: image_batch,
            model.y_input: label_batch
        }
        correct_prediction = sess.run(model.correct_prediction,␣
↪feed_dict=feed_dict)
        correct_predictions = np.concatenate([correct_predictions,␣
↪correct_prediction], axis=0)

    return correct_predictions

# Evaluate the naturally-trained model on the first 1000 samples in the test␣
↪dataset
indices = np.arange(0, 1000)

print('Evaluating naturally-trained model')
correct_predictions = evaluate(model, sess, eval_data, indices)
accuracy = np.mean(correct_predictions)*100
```

```
print('Accuracy: {:.1f}%'.format(accuracy))

# Select the first 100 samples that are correctly classified.
correct_indices = np.where(correct_predictions==1)[0][:100]
```

```
Evaluating naturally-trained model
Accuracy: 96.2%
```

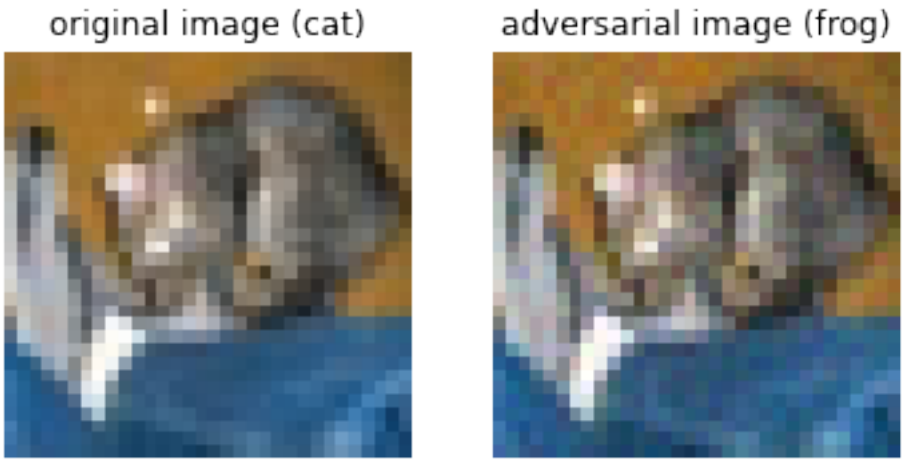
0.5 Black-box attack with NES gradient estimation (NES)

Now, we will implement NES attack, a black-box attack method proposed by Ilyas et al., which uses vector-wise gradient estimation technique called NES and then performs PGD with those estimated gradients.

NES estimates the gradient by
$\nabla_x L(\theta, x, y) \approx \frac{1}{\sigma n} \sum_i^n (L(x + \sigma u_i) - L(x - \sigma u_i)) u_i$
where each u_i are image size random vectors sampled from standard normal distribution.
Your code for this section will all be written inside `attacks/nes_attack`.

[6]:
```
# First implement NES attack.
# Open attacks/nes_attack.py and follow instructions in the file.
from attacks.nes_attack import NESAttack

epsilon = 8
step_size = 2
num_steps = 20
loss_func = 'xent'

nes_attack = NESAttack(model, epsilon, step_size, num_steps, loss_func)

sample_image = eval_data.xs[correct_indices[0]]
sample_image = np.int32(sample_image)
sample_image = np.expand_dims(sample_image, axis=0)
sample_label = eval_data.ys[correct_indices[0]]
sample_label = np.expand_dims(sample_label, axis=0)
sample_adv_image = nes_attack.perturb(sample_image, sample_label, sess)
feed_dict = {
    model.x_input: sample_adv_image
}
sample_adv_label = sess.run(model.predictions, feed_dict=feed_dict)

# Check if the adversarial image is valid
assert np.amax(np.abs(sample_image-sample_adv_image)) <= epsilon
assert np.amin(sample_adv_image) >= 0
assert np.amax(sample_adv_image) <= 255

# Plot the original image
plt.subplot(1, 2, 1)
plt.imshow(sample_image[0, ...].astype('uint8'))
```

```
plt.axis('off')
plt.title('original image ({})'.format(classes[sample_label[0]]))

# Plot the adversarial image
plt.subplot(1, 2, 2)
plt.imshow(sample_adv_image[0, ...].astype('uint8'))
plt.axis('off')
plt.title('adversarial image ({})'.format(classes[sample_adv_label[0]]));
```



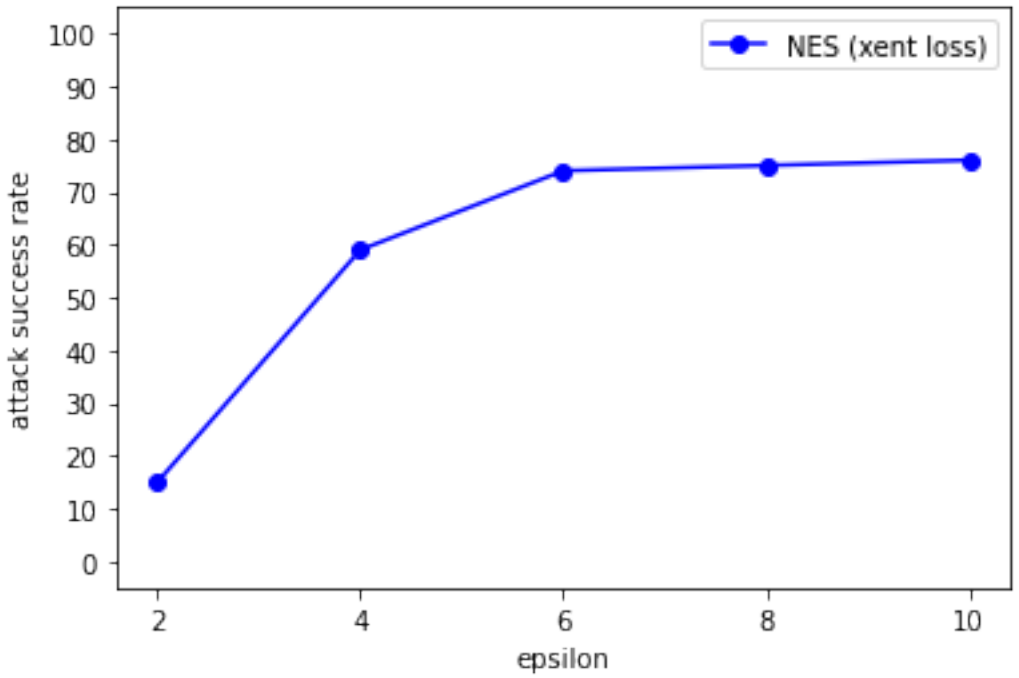
1 Evaluate performance on a naturally-trained model

Let's measure your attack's performance to check if you implemented it right. Also watch the attack success rate change as epsilon gets larger. If correctly implemented, the success rate will be about 75% or higher on epsilon 8. (Keep in mind that NES attack in our implementation attacks one image at a time, so the evaluation will take much longer than FGSM or PGD. Evaluation on a single epsilon may take up to 10 min.)

[7]:
```
epsilons = [2, 4, 6, 8, 10]
attack_success_rates = []

for epsilon in epsilons:
    nes_attack = NESAttack(model, epsilon, step_size, num_steps, loss_func)
    correct_predictions = evaluate(model, sess, eval_data, correct_indices,␣
 ↪attack_method=nes_attack)
    attack_success_rate = np.mean(1-correct_predictions)*100
    attack_success_rates.append(attack_success_rate)
    print('Epsilon: {}, Attack success rate: {:.1f}%'.format(epsilon,␣
 ↪attack_success_rate))

plt.plot(epsilons, attack_success_rates, '-bo', label='NES (xent loss)')
```

```
plt.ylim(-5, 105)
plt.xticks(epsilons)
plt.yticks(np.arange(0, 110, 10))
plt.xlabel('epsilon')
plt.ylabel('attack success rate')
plt.legend();
```

```
Epsilon: 2, Attack success rate: 15.0%
Epsilon: 4, Attack success rate: 59.0%
Epsilon: 6, Attack success rate: 74.0%
Epsilon: 8, Attack success rate: 75.0%
Epsilon: 10, Attack success rate: 76.0%
```



2 Attacks on adversarially-trained model

[8]:
```
# Reset all graphs
tf.reset_default_graph()

# Create naturally-trained model
model = Model(mode='eval')
sess = tf.InteractiveSession()

# Restore parameters
```

```
ADV_MODEL_DIR = './models/adv_trained'
model_file = tf.train.latest_checkpoint(ADV_MODEL_DIR)

var_list = {}
with tf.variable_scope('', reuse=True):
  for var in tf.train.list_variables(model_file)[1:]:
    if 'Momentum' not in var[0]:
      var_list[var[0]] = tf.get_variable(name=var[0].replace('BatchNorm',␣
 ↪'batch_normalization'))

saver = tf.train.Saver(var_list=var_list)
saver.restore(sess, model_file)
```

```
INFO:tensorflow:Restoring parameters from ./models/adv_trained/checkpoint-70000
```

[9]:
```
# Check whether the model is successfully restored.
indices = np.arange(0, 1000)
correct_predictions = evaluate(model, sess, eval_data, indices)
accuracy = np.mean(correct_predictions)*100
print('Accuracy: {:.1f}%'.format(accuracy))

# Select the indices of the first 100 images that are correctly classified.
correct_indices = np.where(correct_predictions==1)[0][:100]
```

```
Accuracy: 87.3%
```

3 Evaluate performance on an adversarially-trained model

This time you will check the same attack's performance on an adversarially-trained model. Check for differences on the success rate.

[10]:
```
epsilons = [2, 4, 6, 8, 10]
attack_success_rates = []

for epsilon in epsilons:
    nes_attack = NESAttack(model, epsilon, step_size, num_steps, loss_func)
    correct_predictions = evaluate(model, sess, eval_data, correct_indices,␣
 ↪attack_method=nes_attack)
    attack_success_rate = np.mean(1-correct_predictions)*100
    attack_success_rates.append(attack_success_rate)
    print('Epsilon: {}, Attack success rate: {:.1f}%'.format(epsilon,␣
 ↪attack_success_rate))

plt.plot(epsilons, attack_success_rates, '-bo', label='NES (xent loss)')
plt.ylim(-5, 105)
plt.xticks(epsilons)
plt.yticks(np.arange(0, 110, 10))
```

```
plt.xlabel('epsilon')
plt.ylabel('attack success rate')
plt.legend();
```

```
Epsilon: 2, Attack success rate: 5.0%
Epsilon: 4, Attack success rate: 8.0%
Epsilon: 6, Attack success rate: 12.0%
Epsilon: 8, Attack success rate: 18.0%
Epsilon: 10, Attack success rate: 24.0%
```