In [1]:
```
# Make TFGAN models and TF-Slim models discoverable.
import sys
import os
# This is needed since the notebook is stored in the `tensorflow/models/gan`
sys.path.append('..')
sys.path.append(os.path.join('..', 'slim'))
```

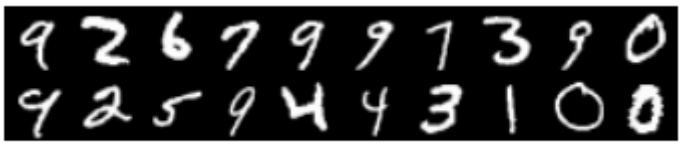
In [2]:
```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import time
import functools
from six.moves import xrange  # pylint: disable=redefined-builtin

import tensorflow as tf

# Main TFGAN library.
tfgan = tf.contrib.gan

# TFGAN MNIST examples from `tensorflow/models`.
from mnist import data_provider
from mnist import util

# TF-Slim data provider.
from datasets import download_and_convert_mnist

# Shortcuts for later.
queues = tf.contrib.slim.queues
layers = tf.contrib.layers
ds = tf.contrib.distributions
framework = tf.contrib.framework
```

In [3]:
```
leaky_relu = lambda net: tf.nn.leaky_relu(net, alpha=0.01)


def visualize_training_generator(train_step_num, start_time, data_np):
    """Visualize generator outputs during training.

    Args:
        train_step_num: The training step number. A python integer.
        start_time: Time when training started. The output of `time.time()`.
            python float.
        data: Data to plot. A numpy array, most likely from an evaluated Tens
            tensor.
    """
    print('Training step: %i' % train_step_num)
    time_since_start = (time.time() - start_time) / 60.0
    print('Time since start: %f m' % time_since_start)
    print('Steps per min: %f' % (train_step_num / time_since_start))
    plt.axis('off')
    plt.imshow(np.squeeze(data_np), cmap='gray')
    plt.show()

def visualize_digits(tensor_to_visualize):
    """Visualize an image once. Used to visualize generator before training.

    Args:
        tensor_to_visualize: An image tensor to visualize. A python Tensor.
    """
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        with queues.QueueRunners(sess):
            images_np = sess.run(tensor_to_visualize)
    plt.axis('off')
    plt.imshow(np.squeeze(images_np), cmap='gray')

def evaluate_tfgan_loss(gan_loss, name=None):
    """Evaluate GAN losses. Used to check that the graph is correct.

    Args:
        gan_loss: A GANLoss tuple.
        name: Optional. If present, append to debug output.
    """
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        with queues.QueueRunners(sess):
            gen_loss_np = sess.run(gan_loss.generator_loss)
            dis_loss_np = sess.run(gan_loss.discriminator_loss)
    if name:
        print('%s generator loss: %f' % (name, gen_loss_np))
        print('%s discriminator loss: %f'% (name, dis_loss_np))
    else:
        print('Generator loss: %f' % gen_loss_np)
        print('Discriminator loss: %f'% dis_loss_np)
```

In [4]:
```
MNIST_DATA_DIR = '/tmp/mnist-data'

if not tf.gfile.Exists(MNIST_DATA_DIR):
    tf.gfile.MakeDirs(MNIST_DATA_DIR)

download_and_convert_mnist.run(MNIST_DATA_DIR)
```

```
Dataset files already exist. Exiting without re-creating them.
```

In [5]:
```
tf.reset_default_graph()

# Define our input pipeline. Pin it to the CPU so that the GPU can be reserv
# for forward and backwards propogation.
batch_size = 32
with tf.device('/cpu:0'):
    real_images, _, _ = data_provider.provide_data(
        'train', batch_size, MNIST_DATA_DIR)

# Sanity check that we're getting images.
check_real_digits = tfgan.eval.image_reshaper(
    real_images[:20,...], num_cols=10)
visualize_digits(check_real_digits)
```



In [6]:
```
def generator_fn(noise, weight_decay=2.5e-5, is_training=True):
    """Simple generator to produce MNIST images.

    Args:
        noise: A single Tensor representing noise.
        weight_decay: The value of the l2 weight decay.
        is_training: If `True`, batch norm uses batch statistics. If `False`
            norm uses the exponential moving average collected from populati
            statistics.

    Returns:
        A generated image in the range [-1, 1].
    """
    with framework.arg_scope(
        [layers.fully_connected, layers.conv2d_transpose],
        activation_fn=tf.nn.relu, normalizer_fn=layers.batch_norm,
        weights_regularizer=layers.l2_regularizer(weight_decay)),\
    framework.arg_scope([layers.batch_norm], is_training=is_training,
                        zero_debias_moving_mean=True):
        net = layers.fully_connected(noise, 1024)
        net = layers.fully_connected(net, 7 * 7 * 256)
        net = tf.reshape(net, [-1, 7, 7, 256])
        net = layers.conv2d_transpose(net, 64, [4, 4], stride=2)
        net = layers.conv2d_transpose(net, 32, [4, 4], stride=2)
        # Make sure that generator output is in the same range as `inputs`
        # ie [-1, 1].
        net = layers.conv2d(net, 1, 4, normalizer_fn=None, activation_fn=tf.t

        return net
```

In [7]:
```
def discriminator_fn(img, unused_conditioning, weight_decay=2.5e-5,
                     is_training=True):
    """Discriminator network on MNIST digits.

    Args:
        img: Real or generated MNIST digits. Should be in the range [-1, 1].
        unused_conditioning: The TFGAN API can help with conditional GANs, wh
            would require extra `condition` information to both the generato
            discriminator. Since this example is not conditional, we do not u
            argument.
        weight_decay: The L2 weight decay.
        is_training: If `True`, batch norm uses batch statistics. If `False`
            norm uses the exponential moving average collected from populatio
            statistics.

    Returns:
        Logits for the probability that the image is real.
    """
    with framework.arg_scope(
        [layers.conv2d, layers.fully_connected],
        activation_fn=leaky_relu, normalizer_fn=None,
        weights_regularizer=layers.l2_regularizer(weight_decay),
        biases_regularizer=layers.l2_regularizer(weight_decay)):
        net = layers.conv2d(img, 64, [4, 4], stride=2)
        net = layers.conv2d(net, 128, [4, 4], stride=2)
        net = layers.flatten(net)
        with framework.arg_scope([layers.batch_norm], is_training=is_trainin
            net = layers.fully_connected(net, 1024, normalizer_fn=layers.bat
        return layers.linear(net, 1)
```

In [8]:
```
noise_dims = 64
gan_model = tfgan.gan_model(
    generator_fn,
    discriminator_fn,
    real_data=real_images,
    generator_inputs=tf.random_normal([batch_size, noise_dims]))

# Sanity check that generated images before training are garbage.
check_generated_digits = tfgan.eval.image_reshaper(
    gan_model.generated_data[:20,...], num_cols=10)
visualize_digits(check_generated_digits)
```

In [9]:
```
# We can use the minimax loss from the original paper.
vanilla_gan_loss = tfgan.gan_loss(
    gan_model,
    generator_loss_fn=tfgan.losses.minimax_generator_loss,
    discriminator_loss_fn=tfgan.losses.minimax_discriminator_loss)

# We can use the Wasserstein loss (https://arxiv.org/abs/1701.07875) with th
# gradient penalty from the improved Wasserstein loss paper
# (https://arxiv.org/abs/1704.00028).
improved_wgan_loss = tfgan.gan_loss(
    gan_model,
    # We make the loss explicit for demonstration, even though the default i
    # Wasserstein loss.
    generator_loss_fn=tfgan.losses.wasserstein_generator_loss,
    discriminator_loss_fn=tfgan.losses.wasserstein_discriminator_loss,
    gradient_penalty_weight=1.0)

# We can also define custom losses to use with the rest of the TFGAN framewo
def silly_custom_generator_loss(gan_model, add_summaries=False):
    return tf.reduce_mean(gan_model.discriminator_gen_outputs)
def silly_custom_discriminator_loss(gan_model, add_summaries=False):
    return (tf.reduce_mean(gan_model.discriminator_gen_outputs) -
            tf.reduce_mean(gan_model.discriminator_real_outputs))
custom_gan_loss = tfgan.gan_loss(
    gan_model,
    generator_loss_fn=silly_custom_generator_loss,
    discriminator_loss_fn=silly_custom_discriminator_loss)

# Sanity check that we can evaluate our losses.
for gan_loss, name in [(vanilla_gan_loss, 'vanilla loss'),
                       (improved_wgan_loss, 'improved wgan loss'),
                       (custom_gan_loss, 'custom loss')]:
    evaluate_tfgan_loss(gan_loss, name)
```

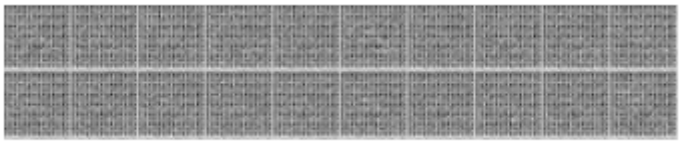
```
vanilla loss generator loss: -1.450439
vanilla loss discriminator loss: 1.674405
improved wgan loss generator loss: 0.539317
improved wgan loss discriminator loss: -0.020071
custom loss generator loss: -0.347576
custom loss discriminator loss: 0.024440
```

In [10]:
```
generator_optimizer = tf.train.AdamOptimizer(0.001, beta1=0.5)
discriminator_optimizer = tf.train.AdamOptimizer(0.0001, beta1=0.5)
gan_train_ops = tfgan.gan_train_ops(
    gan_model,
    improved_wgan_loss,
    generator_optimizer,
    discriminator_optimizer)
```

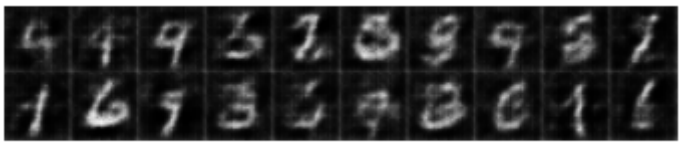
In [11]:
```
num_images_to_eval = 500
MNIST_CLASSIFIER_FROZEN_GRAPH = './mnist/data/classify_mnist_graph_def.pb'

# For variables to load, use the same variable scope as in the train job.
with tf.variable_scope('Generator', reuse=True):
    eval_images = gan_model.generator_fn(
        tf.random_normal([num_images_to_eval, noise_dims]),
        is_training=False)

# Calculate Inception score.
eval_score = util.mnist_score(eval_images, MNIST_CLASSIFIER_FROZEN_GRAPH)

# Calculate Frechet Inception distance.
with tf.device('/cpu:0'):
    real_images, _, _ = data_provider.provide_data(
        'train', num_images_to_eval, MNIST_DATA_DIR)
frechet_distance = util.mnist_frechet_distance(
    real_images, eval_images, MNIST_CLASSIFIER_FROZEN_GRAPH)

# Reshape eval images for viewing.
generated_data_to_visualize = tfgan.eval.image_reshaper(
    eval_images[:20,...], num_cols=10)
```

In [12]:
```
# We have the option to train the discriminator more than one step for every
# step of the generator. In order to do this, we use a `GANTrainSteps` with
# desired values. For this example, we use the default 1 generator train ste
# for every discriminator train step.
train_step_fn = tfgan.get_sequential_train_steps()

global_step = tf.train.get_or_create_global_step()
loss_values, mnist_scores, frechet_distances  = [], [], []

with tf.train.SingularMonitoredSession() as sess:
    start_time = time.time()
    for i in xrange(1601):
        cur_loss, _ = train_step_fn(
            sess, gan_train_ops, global_step, train_step_kwargs={})
        loss_values.append((i, cur_loss))
        if i % 200 == 0:
            mnist_score, f_distance, digits_np = sess.run(
                [eval_score, frechet_distance, generated_data_to_visualize])
            mnist_scores.append((i, mnist_score))
            frechet_distances.append((i, f_distance))
            print('Current loss: %f' % cur_loss)
            print('Current MNIST score: %f' % mnist_scores[-1][1])
            print('Current Frechet distance: %f' % frechet_distances[-1][1])
            visualize_training_generator(i, start_time, digits_np)
```

```
Current loss: -0.744440
Current MNIST score: 1.000062
Current Frechet distance: 339.406494
Training step: 0
Time since start: 0.065372 m
Steps per min: 0.000000
```



```
Current loss: -3.240865
Current MNIST score: 2.691993
Current Frechet distance: 166.751053
Training step: 200
Time since start: 1.576845 m
Steps per min: 126.835563
```

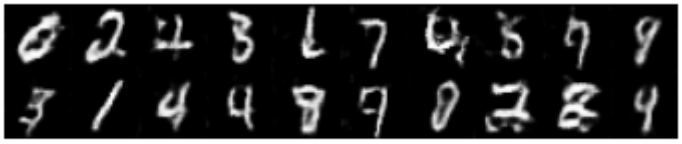


```
Current loss: -3.095253
Current MNIST score: 4.904757
Current Frechet distance: 93.779289
Training step: 400
```

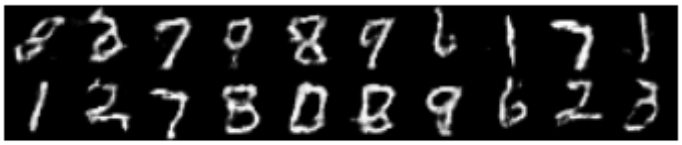
Time since start: 3.092882 m
Steps per min: 129.329208

Current loss: -3.088769
Current MNIST score: 6.725487
Current Frechet distance: 39.744904
Training step: 600
Time since start: 4.608689 m
Steps per min: 130.188869

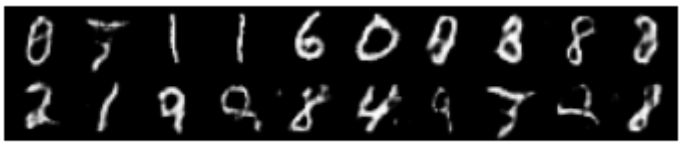
Current loss: -3.074484
Current MNIST score: 7.534400
Current Frechet distance: 19.210173
Training step: 800
Time since start: 6.130447 m
Steps per min: 130.496196



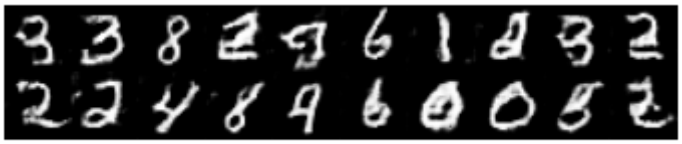
Current loss: -2.957786
Current MNIST score: 7.315704
Current Frechet distance: 24.819916
Training step: 1000
Time since start: 7.653988 m
Steps per min: 130.650852



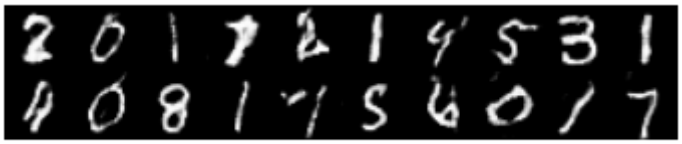
Current loss: -2.981731
Current MNIST score: 6.797893
Current Frechet distance: 53.053493
Training step: 1200
Time since start: 9.164964 m
Steps per min: 130.933413



```
Current loss: -3.221900
Current MNIST score: 7.398132
Current Frechet distance: 33.662426
Training step: 1400
Time since start: 10.689862 m
Steps per min: 130.965214
```

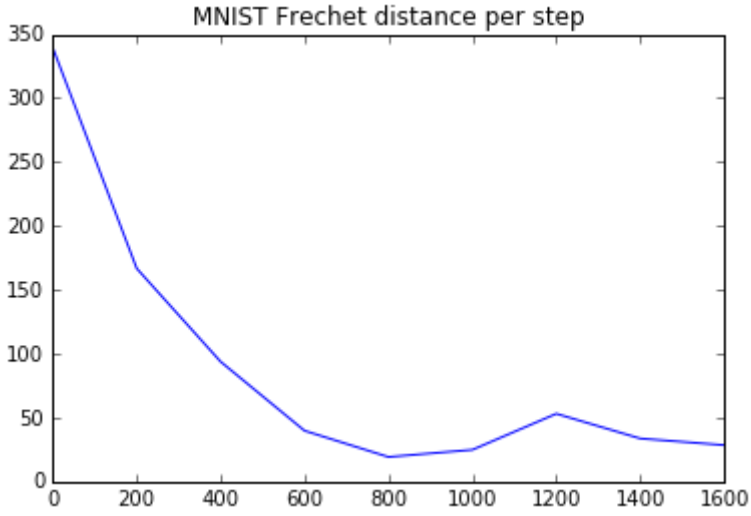


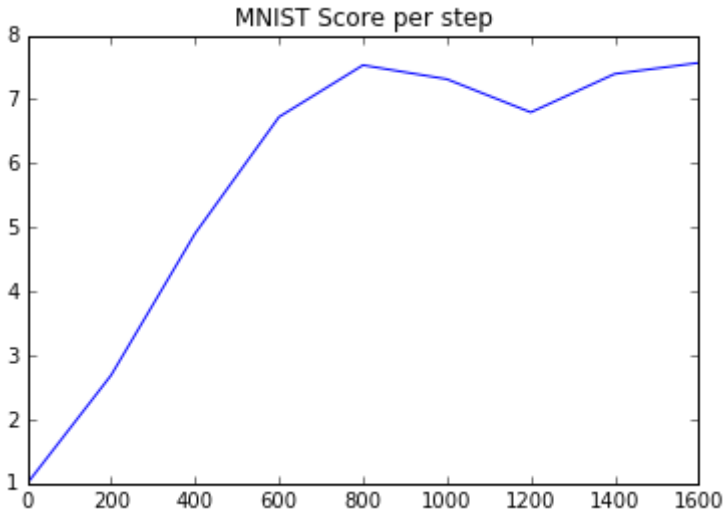
```
Current loss: -3.512912
Current MNIST score: 7.570933
Current Frechet distance: 28.488590
Training step: 1600
Time since start: 12.209630 m
Steps per min: 131.044103
```

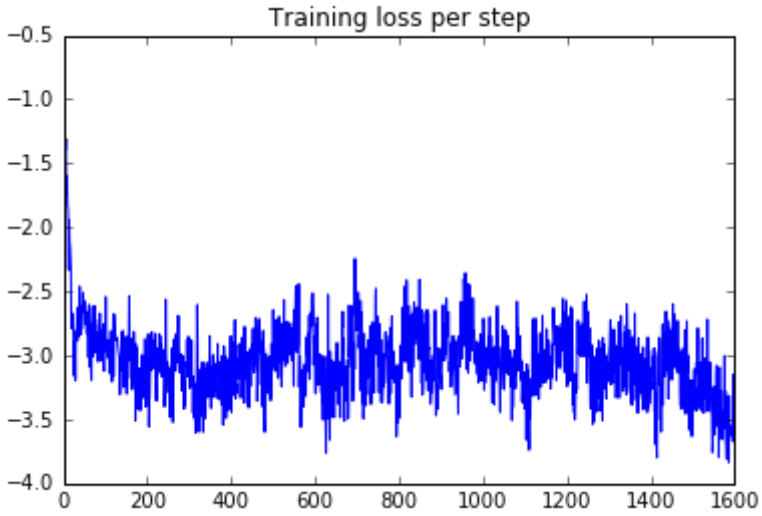


In [13]:
```
# Plot the eval metrics over time.
plt.title('MNIST Frechet distance per step')
plt.plot(*zip(*frechet_distances))
plt.figure()
plt.title('MNIST Score per step')
plt.plot(*zip(*mnist_scores))
plt.figure()
plt.title('Training loss per step')
plt.plot(*zip(*loss_values))
```

Out[13]:
```
[<matplotlib.lines.Line2D at 0x7f7a689b46d8>]
```






Training loss per step
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
−4.0
0
200
400
600
800
1000
1200
1400
1600

In [ ]: