UDACITY

PROJECT

# Generate Faces

A part of the Deep Learning Nanodegree Foundation Program

PROJECT REVIEW

CODE REVIEW

NOTES

SHARE YOUR ACCOMPLISHMENT!

## Meets Specifications

Wow, you are actually generating faces without having seen any of them. Remember your network has seen some faces, but not the faces that it generated. This was an awesome project. You are working on cutting edge applications of GANs.
Now let's see those faces

Here are some additinal resources tht you may find interesting:

- Generative Adversarial Networks Explained with a Classic Spongebob Squarepants Episode
- GitHub - zhangqianhui/AdversarialNetsPapers: The classical papers and codes about generative adversarial nets
- generative-models/GAN at master · wiseodd/generative-models · GitHub
- magenta/GAN.md at master · tensorflow/magenta · GitHub
- Image Completion with Deep Learning in TensorFlow
- An introduction to Generative Adversarial Networks (with code in TensorFlow) - AYLIEN
- MNIST Generative Adversarial Model in Keras – O'Shea Research
- Fantastic GANs and where to find them
- ARXIV_1
- ARXIV_2
- ARXIV_3

There is a small issue. Your fake images are 32x32x3 instead of 28x28x3. But it's the seasons to forgive but NOT forget. So I have passed your notebook, but make sure you make that change before you move on from the project. You don't want to put the wrong notebook on your resume and get grilled in an interview. So please make that update.

### Required Files and Tests

**The project submission contains the project notebook, called "dlnd_face_generation.ipynb".**

Yes. Found it.

```
-rwxr-xr-x@ 1 XXXX staff 3671514 Jul 24 10:26 dlnd_face_generation_submission_1.ipynb
```

**All the unit tests in project have passed.**

Good job. Unit tests are good way to sanitize your code in chunks. This helps isolate problems to easier to code debug code chunks.
Always a good practice to write/break your code in small functions to do something focused, granular and easily measured and write unit tests to check that the function.
Here are some good attributes of a good function:

```
short code size
high efficiency
low memory footprint
high reliability
high generality
```

### Build the Neural Network

**The function model_inputs is implemented correctly.**

**The function discriminator is implemented correctly.**

- You can use a more memory efficient form of leaky ReLu as referenced here

```
def lrelu(x, leak=0.2, name="lrelu"):
   with tf.variable_scope(name):
       f1 = 0.5 * (1 + leak)
       f2 = 0.5 * (1 - leak)
       return f1 * x + f2 * abs(x))
```

- Instead of this `flat = tf.reshape(h3, (-1, 4*4*256))` you can use `flat = tf.contrib.layers.flatten(h3)` So that you don't need to remember the shape.
- Finally, its always a good idea to use a kernel_initializer in the conv2d. You can use `kernel_initializer=tf.random_normal_initializer(stddev=stddev))`
- Good values for stddev are 0.02 to start with. You can also use the `xavier_initialize/tf.contrib.layers.xavier_initializer()` as the value for the kernel_initializer parameter in `tf.layers.conv2d`. For more see here

**The function generator is implemented correctly.**

- Your output images are going to be 32x32x3 and not 28x28x3 as we need (since the real images are 28x28x3). Normally I would make this "requires changes" but we have been asked to be more lenient since its end of the term. So I will let it go. But please change your parameters so that you get 28x28x3
- The same feedback about leaky ReLus as before. You can use the more memory efficient version.
- It's a good idea to use the `tf.contrib.layers.xavier_initializer()` in the `conv2d_transpose`.
- Also, as pointed out here it is important to make the Generator to be more powerful than the Discriminator. So please see if you can add one more layer and increase channel depth and the input. You may get better final results.
- Also, note that we are using a `tanh` as the final activation in the generator, which means you have to scale the real images to be between -1 and 1 if not done for you already before you start training.
  `out = tf.tanh(logits)`

**The function model_loss is implemented correctly.**

- You can use a smoothing function. Something similar to this :
  `d_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_real, labels=tf.ones_like(d_model_real) * np.random.uniform(0.7, 1`
- You can use a a similar way for the zeros as well.
  `d_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake, labels=tf.ones_like(d_model_fake) * np.random.uniform(0.0, 0`
- As mentioned on the GANHACKs it is better to have a number close to 0 and 1 as the labels instead of a strict 0 or 1.* The idea is not let the discriminator efficiently learn the one's and thus prevent it from overfitting. We introduce stochasticity in training process an idea similar to adding dropouts.

**The function model_opt is implemented correctly.**

```
d_train_opt = tf.train.AdamOptimizer(
    learning_rate, beta1=beta1).minimize(d_loss, var_list=d_vars)

ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
g_updates = [opt for opt in ops if opt.name.startswith('generator')]
with tf.control_dependencies(g_updates):
    g_train_opt = tf.train.AdamOptimizer(
        learning_rate, beta1).minimize(g_loss, var_list=g_vars)
```

- You should do the same for the Discriminator as well.

## Neural Network Training

**The function train is implemented correctly.**

- It should build the model using `model_inputs`, `model_loss`, and `model_opt`.
- It should show output of the `generator` using the `show_generator_output` function

These are the two most important steps in this function to get right to get good results. And you have them correctly done.

```
                batch_images *= 2.0
                z_sample = np.random.uniform(-1, 1, (batch_size, z_dim))
```

**The parameters are set reasonable numbers.**

```
batch_size = 64
z_dim = 100
learning_rate = 0.002
beta1 = 0.5
```

Yeah. Look good!

- Usually a good value for z_dim is 100. Why? That's what I have seen in most papers.
- You can refer to section 3 of this paper to see that values they used.
  The few most important parameters that you can tune are:

  ```
  batch_size
  learning_rate
  beta1
  ```

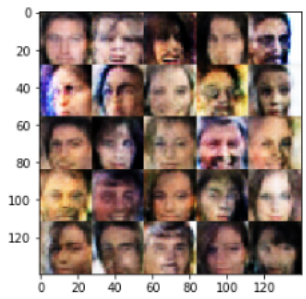
- Here is the effect of each of them:

  ```
  • Smaller batches usually find a relatively good solution more quickly, as they make a smaller pass through the dat
  a before performing an update however they can be influenced by most recent examples more heavily.
  • Larger batches can be more stable, as they are not overly influenced by the most recently seen training examples
   but take longer to update the model parameters.
  ```

- Good choices of batch sizes may also depend on epochs. If you have fewer epochs to train it makes more sense to have smaller batches. This allows weight updates more frequently than if the batches were larger. But try to keep batch sizes as a power of 2. But smaller batches also mean that you will see large changes from batch to batch.
- The learning rate will determine if the learning is stable or unstable. Most of these networks can become unstable easily and hence a low learning rate is preferred. Your choice of learning rate is fine, but I encourage you ti play around with a learning that is one order or two orders larger and one order smaller. What happens to the images?
- Finally to the see the effect of beta1 look @ the page for AdamOptimizer. Play around with the values of beta1 using values 0.9, 0.5 and 0.2 and see the effect.

**The project generates realistic faces. It should be obvious that images generated look like faces.**

Looks ok. But they seem like they haven't slept for many days. 😄




```
Epoch 1/1... Discriminator Loss: 1.0085... Generator Loss: 0.8846
Epoch 1/1... Discriminator Loss: 1.2728... Generator Loss: 0.5930
```

DOWNLOAD PROJECT

RETURN TO PATH

Student FAQ