UDACITY

‹ Return to "Deep Learning" in the classroom

Generate Faces

REVIEW

CODE REVIEW

HISTORY

Requires Changes

2 SPECIFICATIONS REQUIRE CHANGES

Kudos ! I think you've done a perfect job of implementing a Deep Convolutional GAN to Generate Faces. It's very clear that you have a good understanding of the basics.
There were some minor errors/tips which I am sure you can improve for your next submission !

Looking forward to your next submission. :)

Further Reading: One of the biggest problem GAN researchers face (you yourself must have experienced) with standard loss function is, the quality of generated images do not correlate with loss of either G or D. Since both the network are competing against each other, the losses fluctuate *a lot*. This problem was solved in early 2017 with introduction of Wasserstein GANs. With WGAN, the loss function directly correlate with how good your model is, and tracking decrease in loss a good idea. Do read it up.

Finally, have a look at this amazing library by Google for training and evaluating Generative Adversarial Networks.

Required Files and Tests

The project submission contains the project notebook, called “dlnd_face_generation.ipynb”.

The `iPythonNB` and helper files are included.

All the unit tests in project have passed.

Great work. Unit testing is one of the most reliable methods to ensure that your code is free from all bugs without getting confused with the interactions with all the other code. If you are interested, you can read up more and I hope that you will continue to use unit testing in every module that you write to keep it clean and speed up your development.

But always keep in mind, that unit tests cannot catch every issue in the code. So your code could have bugs even though unit tests pass.

Build the Neural Network

The function model_inputs is implemented correctly.

Correct.
Placeholders is the building block in computation graph of any neural net (especially in **tensorflow**).

Often I find students confused between `tf.Variable` and `tf.placeholder`. This answer gives correct usecase for both.

The function discriminator is implemented correctly.

Overall you did a fine job implementing the `Discriminator` as a simple convolution network.

Let me illustrate the pros of the architecture you chose.

Pros

- `tf.variable_scope('discriminator', reuse=reuse)` was essential to this part for two reasons. Firstly, to make sure all the variable names start with start with `discriminator`. This will help out later when training the separate networks. Secondly, the discriminator will need to share variables between the fake and real input images using reuse.
- You chose not to use pooling layers to decrease the spatial size. Max pooling generates sparse gradients, which affects the stability of GAN training. We generally use Average Pooling or Conv2d + stride.
- Correctly used Leaky ReLU. As explained above we never want sparse gradients (~ 0 gradients). Therefore, we use a leaky ReLU to allow gradients to flow backwards through the layer unimpeded.
- Used Batch normalization. We initialize the BatchNorm Parameters to transform the input to zero mean/unit variance distributions but as the training proceeds it can learn to transform to `x` mean and `y` variance, which might be better for the network. This post is an awesome read to understand BatchNorm to it's core.
- Using a sigmoid for output layer.

Tips

- Using dropout in discriminator so that it is less prone to learning the data distribution.

- Use custom weight initialization. Xavier init is proposed to work best when working with GANs.

The function generator is implemented correctly.

Most of the suggestions are same for both Generator and Discriminator.

Let me (again) illustrate the pros of the architecture you chose.

Pros

- `Tanh` as the last layer of the generator output. This means that we'll have to normalize the input images to be between -1 and 1.

Tips

- Try setting leak for `leaky_relu` a bit lower. Did you tried 0.1 ?
- Try decreasing the width of layers from 512 -> 64. In context of GANs, a sharp decline in number of filters for Generator helps produce better results.

The function model_loss is implemented correctly.

Perfect.
Now that was the trickiest part (and my personal favorite in GAN :)

Tips

- Use **One Sided Label Smoothing** for Discriminator loss, will help it generalize better. If you have two target labels: Real=1 and Fake=0, then for each incoming sample, if it is real, then replace the label with a random number between 0.7 and 1.2, and if it is a fake sample, replace it with 0.0 and 0.3 (for example).
- A simple change like `labels = tf.ones_like(d_logits_real) * np.random.uniform(0.7, 1.2)` will help with the training process. This is known as label smoothing, typically used with classifiers to improve performance.
- However, only-one-sided label smoothing is recommended to weaken the D **and not** G.

The function model_opt is implemented correctly.

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

Perfect job implementing `train`, except for one small piece.

- Tanh is used as the last layer of the Generator output.
- Meaning the generated images by Generator will be between -1 and 1.
- Therefore, we would need to rescale the true images, between -1 to 1 too.
- But currently, `batch_images` lies between -0.5 to 0.5.
- Use, `batch_images = batch_images*2` to normalize the input to Discriminator.

You might want to watch up this talk on How to train a GAN by one of the author of original DCGAN paper and corresponding write-up.

The parameters are set reasonable numbers.

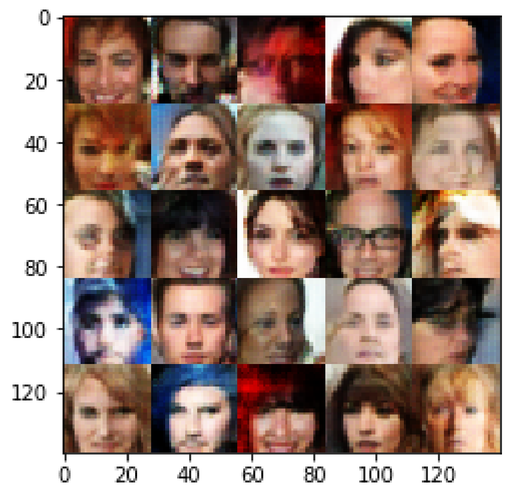
Given your network architecture, the choice of hyper-parameter are reasonable.

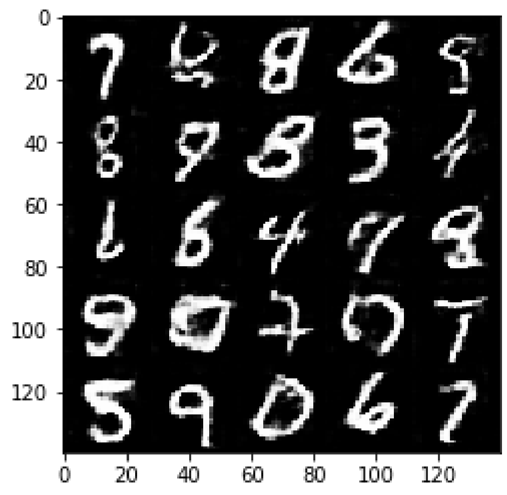
Tips

- You selected a good value for `beta1`. Here's a good post explaining the importance of beta values and which value might be empirically better. Also try lowering it even further, ~0.1 might even produce better results.
- An important point to note is, batch size and learning rate are linked. If the batch size is too small then the gradients will become more unstable and would need to reduce the learning rate.

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

Good work, but output images are not realistic faces.
However, once you fix the issues mentioned above, you will be able to obtain results similar to as shown below.





RESUBMIT

DOWNLOAD PROJECT