UDACITY

‹ Return to "Deep Learning" in the classroom

# Generate Faces

审阅 | 代码审阅 | HISTORY

## Meets Specifications

Congratulations ! 🎉 This is a wonderful submission. Your understanding and clarity of concepts of how GANs work is vividly depicted here. You've done a great job of implementing a Deep Convolutional GAN.

You've done the hard part of building the neural network, as well as that of the daunting hyper-parameter tuning. Some more hyper-parameter tuning is possible as well which I've suggested.

Anyway, were you not amazed by the results that were produced, the generator network was able to produce human-like handwritten numbers and faces without ever seeing any of them? The generator was just told weather its solution was right or wrong. Amazing, right ?!

If you wish to study the subject further, here are few of my recommendations:

1. Wasserstein GAN as it's GAN with an objective and thus has a convergence criterion.
2. Generative Models in general
3. DiscoGAN, Discover Cross-Domain Relations with Generative Adversarial Networks
4. GAN stability
5. MNIST GAN with Keras which shows that how much concise can one be when modelling neural networks.

And Congratulations 👏🎉 once again, as you've completed this project as well as this Deep Learning Nanodegree. I hope you had great experience with Udacity and will continue taking other Nanodegree too. I would suggest you to go for AIND.

### 项目中是否有需要的文件、相关函数是否通过了单元测试

**该项目提交应包含项目中名为 “dlnd_face_generation.ipynb” 的 .ipynb 文件。**

The required notebook and the helper files are included. However, the report (ipython notebook exported as HTML) is not included. Please do include it from the next time.

**项目中所有的单元测试都已通过。**

Well done, all units passed! 😄

However, unit testing does not assure perfect code or results. There could still be some unresolved issues. The purpose of unit testing is to help avoid silly mistakes and improve code quality.

Moreover, unit testing(tutorial) is very good practice to ensure that our code is free from bugs & prevents us from wasting a lot of time while debugging minor problems as well as improves our code standards.

I recommend you to continue using unit testing in every module that you write in future, to keep it clean and speed up your development. Unit testing is highly motivated in industries.

### 建立神经网络

**`model_inputs` 函数被正确实现。**

Great Start! 🎉
Placeholders are the building blocks in computational graphs of any neural network (especially in *tensorflow*, for better understanding refer the link) as they allow us to feed values into the tensorflow graph.

**`discriminator` 函数被正确实现。**

😄 You did a near perfect job of implementing the `discriminator` with the following key points to note:

✅ `discriminator` as a sequence of `conv` layers.

✅ using `conv2d` with `strides` to avoid making sparse gradients instead of max-pooling layers as they make the model unstable.

✅ using `leaky_relu` and avoiding ReLU for the same reason of avoiding sparse gradients as `leaky_relu` allows gradients to flow backwards unimpeded.

✅ using `sigmoid` as output layer

✅ BatchNorm to avoid "internal covariate shift" as batch normalisation minimises the effect of weights and parameters in successive forward and back pass on the initial data normalisation done to make the data comparable across features.

✅ Use a **smaller model** for the `discriminator` relative to `generator` as stated in this review of GANs as generation is a much harder task and requires more parameters, so the generator should be significantly bigger.

✅ Used **weight initialization:** Xavier initialization (A good blog link to help understand what it's) is recommended so as to break symmetry and thus, help converge faster as well as prevent local minima

- If the weights in a network start too small, then the signal shrinks as it passes through each layer until it's too tiny to be useful.
- If the weights in a network start too large, then the signal grows as it passes through each layer until it's too massive to be useful.

Xavier initialization makes sure the weights are 'just right', keeping the signal in a reasonable range of values through many layers.
A possible implementation in tensorflow is to pass `tf.contrib.layers.xavier_initializer()` as the value for the `kernel_initializer` paramter in `tf.layers.conv2d`

✅ Used **Dropouts** in `discrimnator` so as to make it less prone to the mistakes the generator can exploit instead of learning the data distribution as mentioned here. Possible tensorflow implementation can be achieved by simply passing the outputs from last layer into the `tf.nn.dropout` with a high `keep_probability`.

**`generator` 函数被正确实现。**

A superb attempt implementing generator as "deconvolution" network with mostly the same key points as mentioned in the previous `discriminator` block as well as good job implementing:

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

✅ Used a **smaller model** for the `discriminator` relative to `generator`.

💭 Use **Dropouts**(50%) in `generator` in both train and test phase so as to provide noise and apply on several layers of generator at both training and test time. This was first introduced in a image tanslation paper called pix2pix for which you can check out an awesome demo here.

**`model_loss` 函数被正确实现。**

Bravo! 🎉 You did a fantastic job here as the loss function for GANs can be super convoluted and confusing. For more tips and hacks refer this GAN Hacks link by Soumith

Chintala, one of the co-authors of the DCGAN as well as of Wasserstein GAN.

✅ Great job implementing **label smoothing**(one-sided). It's done to prevent `discriminator` from being too strong as well as to help it generalise better by clipping the one-vector labels.

`model_opt` 函数被正确实现。

## 神经网络训练

**训练函数被正确实现。**

- 它应使用 `model_inputs`、`model_loss` 及 `model_opt` 创建了模型。
- 同时它使用 `show_generator_output` 函数展示了 `generator` 的输出。

Brilliant Work!! You have done commendable work here combining all the parts together and making it a **Generative Adversarial Network**.

✅ Fantastic job providing `z` between -1 and 1 and using `np.random.uniform(-1, 1, size=(batch_size, z_dim))`.

❕ It's recommended to normalise the input_images rather than normalising the output. Though conceptually it makes no difference. This is because instead of tinkering with the models later, one simply plays around with the pre-processing and training portion of the networks. Also, the resultant from `tanh` would always be between -1 and 1. However, inputs images can vary, thus, it's recommended to normalise the dynamic part here instead of the static part.

**这些超参数被设置到了合理的数值。**

For your network architecture, the choice of hyperparameters is mostly reasonable.

✅ **Learning Rate**: The current rate is very reasonable.

✅ **Beta1**: Your chosen value for beta1 is good.

✅ Good value of alpha/ **leak** parameter has been chosen. However, still try lowering to 0.1 .

✅ **Z-dim**: 100/128 is a reasonably good choice here.

✅ **Batch Size**: The chosen batch_size is appropriate (~32 to 64) because

- If you choose a batch size too small then the gradients will become more unstable and would need to reduce the learning rate. So batch size and learning rate are linked.
- Also if one use a batch size too big then the gradients will become less noisy but it will take longer to converge.

✅ a good value of **label smoothing** ~ random(0.7, 1.2) to prevent the discriminator from being too strong for the generator.

✅ **Number of Filters/ depth of each layer**: Good choice.

**项目代码生成的面部图像比较逼真，至少看上去像是一些人脸。**

下载项目

返回 PATH