Audrija Mukherjee

Project 1: Blog Series on Generative Adversarial Networks (GANs)

This blog is about a relatively new phenonemenon called GANs, i.e. Generative Adversarial Networks. Let's take this discussion one step at a time with each of the following questions.

Part 1: Background and How They Work

1. Where did they come from?
2. What are they and how do they work?
3. Why should I use them?
4. Where can I use them?

Part 2: A Toy Application

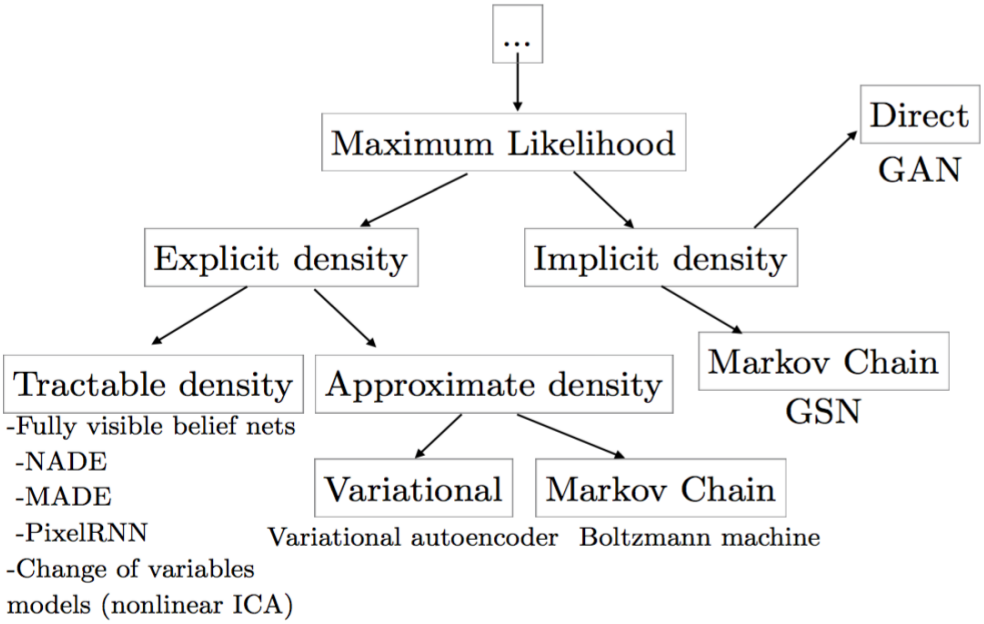
References

Part 1: Background and How They Work

Where did they come from?

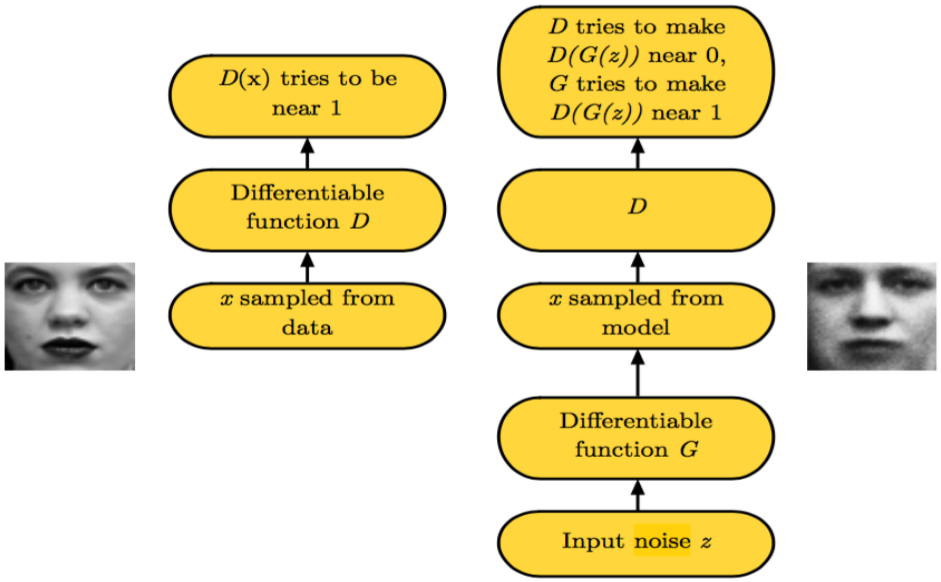
In 2014, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio, lead by Ian Goodfellow came up with Generative Adversarial Networks, and published a paper on it in the journal- Advances in neural information processing systems. The basis of GANs lies in computational graphs and game theory. The paper proves that given enough modeling power, two models fighting against each other would be able to co-train through simply backpropagation.

The following taxonomic tree shows pre-existing methods before GANs. GANs are trained using the strategy from the rightmost leaf of the tree using an implicit model that samples directly from the distribution represented by the model.



What are they and how do they work?

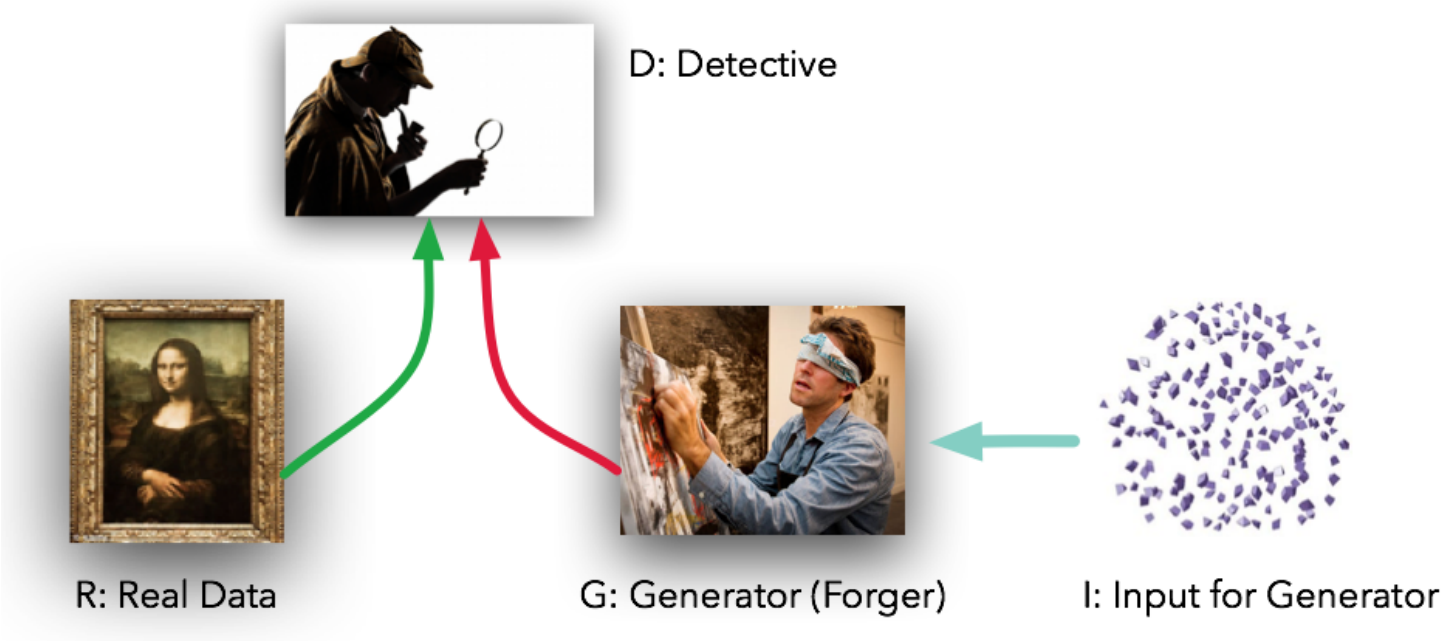
Generative Adversarial Models are exactly what the name implies. They are machine learning models, consisting of 2 adversarial networks/models: a generative model G that captures the data distribution and generates output similar to the training data, and a discriminative model D that estimates the probability that a sample came from the training data or from G. The training procedure for G is to maximize the probability of D making a mistake. The following figure shows the working of both G and D. This framework corresponds to a minimax two-player game. (References 1,2 for explain the math behind it)



The Forger-Detective Analogy

Goodfellow's metaphor for GAN was that G was like a team of forgers trying to match real paintings with their own output as closely as possible, while D was the team of detectives trying to tell the difference aka differentiate the fake ones

from the real ones. The only difference being that the forgers(G) are blind to the original data in the sense that they never see the original data. They only get the judgments of the investigators(D).



Why should I use them?

Apart from Yann LeCun's statement, here are a few more reasons.

Generative models can be trained with missing data and can provide pre- dictions on inputs that are missing data. It is much easier to obtain large amounts of unlabeled data in the real world, than labelled data. GANs are good at performing semi-supervised learning, since they can improve their generalization by studying a large number of unlabeled examples.

Unlike MLE based models, GANs can work with multimodal outputs i.e. they can produce multiple correct answers. Applications of multi-modal outputs include generating the next frame of a video

GANs to generate images (DCGAN)

DCGAN network from Radford et al. takes as input 100 random numbers drawn from a uniform distribution and outputs an image of size 64x64x3. The following animation shows that as the code is changed incrementally, the generated images change too. The samples from the generator start out noisy and chaotic, and over time converge to more realistic images.

Here is the crux of it. Let's assume we have to train a GAN model for ImageNet which is about 200 GB in size. GAN models typically have only about 100 million parameters, so the model has to learn to (lossily) compress 200GB of pixel data into about 100MB of weights. So it must discover the most salient features of the data to do a good job; for example, it will likely learn that spatially close pixels are likely to have similar color, or that most things have horizontal or vertical edges, or blobs of different colors. Eventually, the model may discover many more complex regularities.

Where can I use them? (Examples)

Single image super-resolution: Take a low-resolution image and synthesize a high-resolution equivalent i.e. impute the missing information.Choosing an image that is the average of all possible images outputs one that is too blurry and GANs work well in this case.

Generative Adversarial Text to Image Synthesis: Input a text description of an image and a GAN will generate a realistic image of that scene. This is a particularly interesting application presented in 2016 by Scott Reed, et all of University of Michigan.

Generating Synthetic Art: Images that combine the content of a photograph with the style of several wellknown artworks. The images were created by finding an image that simultaneously matches the content representation of the photograph and the style representation of the artwork. In the following set of images, the first one is the original image. For each of the remaining images, the bottom left corner shows the painting which provided style for its generation

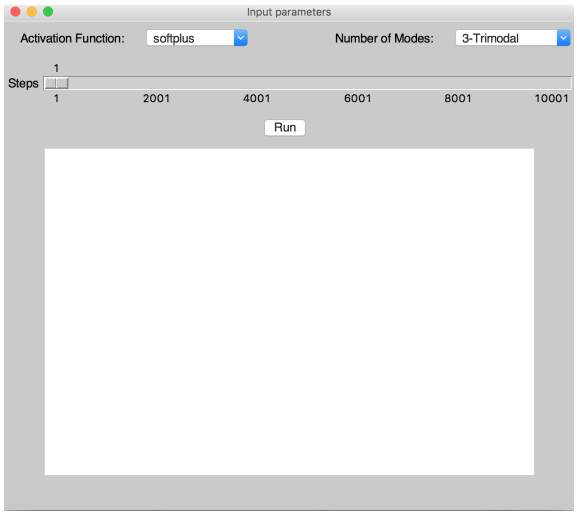
Part 2: A Toy Application

As mentioned above, GANs have a lot of applications. But for understanding how GANs work, all of those examples are a bit too complex and require several tweaks to produce good results. So I came up with a literal toy example. Through a GUI, the user gets to choose an input distribution,the activation function for the generator and the number of steps for the model to run. Now the model will try to learn to approximate the input. The goal is for people who aren't too familar with GANs to be able to see how varying the activation function, input distribution, number of steps, etc affect the output. It's a "toy" application in terms of actually being able to toy with the GAN!

Prerequisites: Tensorflow 0.11, appjar 0.52, developed on Python 2.7

Steps to run:

1. Run the application
2. A GUI pops up. The original screen looks like this.



3. Select the type of distribution you'd like to approximate on the top-right of the window; options are Unimodal, Bimodal and Trimodal.
4. Select the activation function and Number of Steps (1-10001) to train the model for. (A run for 10000 steps will take some time)
5. Press Run. While the code is running you'll see a loading screen.
6. The output is generated and displayed in the space below the Run button.

Options of Activation functions are:

1. Relu: max (features, 0)
2. Relu6: min (max (features, 0), 6)
3. Sigmoid: 1 / (1 + exp(-x))
4. Softplus: log(exp(features) + 1)
5. Tanh: (1 - exp(-2x)) / (1 + exp(-2x))

Code Description

The GUI sets the "loading" screen when the "Run" button is pressed and calls the main_call function to create and train the model with the parameters chosen by the user. In case of any error, it will display an Error screen.

```
%example code- GUI
def on_run(x):
app.setImage("Result",'running.gif')
func = app.getOptionBox("A")
modes = app.getOptionBox("N")
num_steps = app.getScale("Steps")
print ("Running....", num_steps,func,modes)
tf.reset_default_graph()
if (main_call(num_steps=num_steps,num_modes=dict_modes[modes],act_func=dict_act[func])):
    app.setImage("Result", 'op.jpeg')
else:
    app.setImage("Result", 'error.jpeg')
```

The generator's first layer depends on the user input if activation function. The output layer is simply a linear function.

```
%example code- generator
if act_func==1:
    h0 = tf.nn.relu(linear(input, h_dim, 'g0'))
elif act_func==2:
    h0 = tf.nn.relu6(linear(input, h_dim, 'g0'))
elif act_func == 3:
    h0 = tf.tanh(linear(input, h_dim, 'g0'))
```

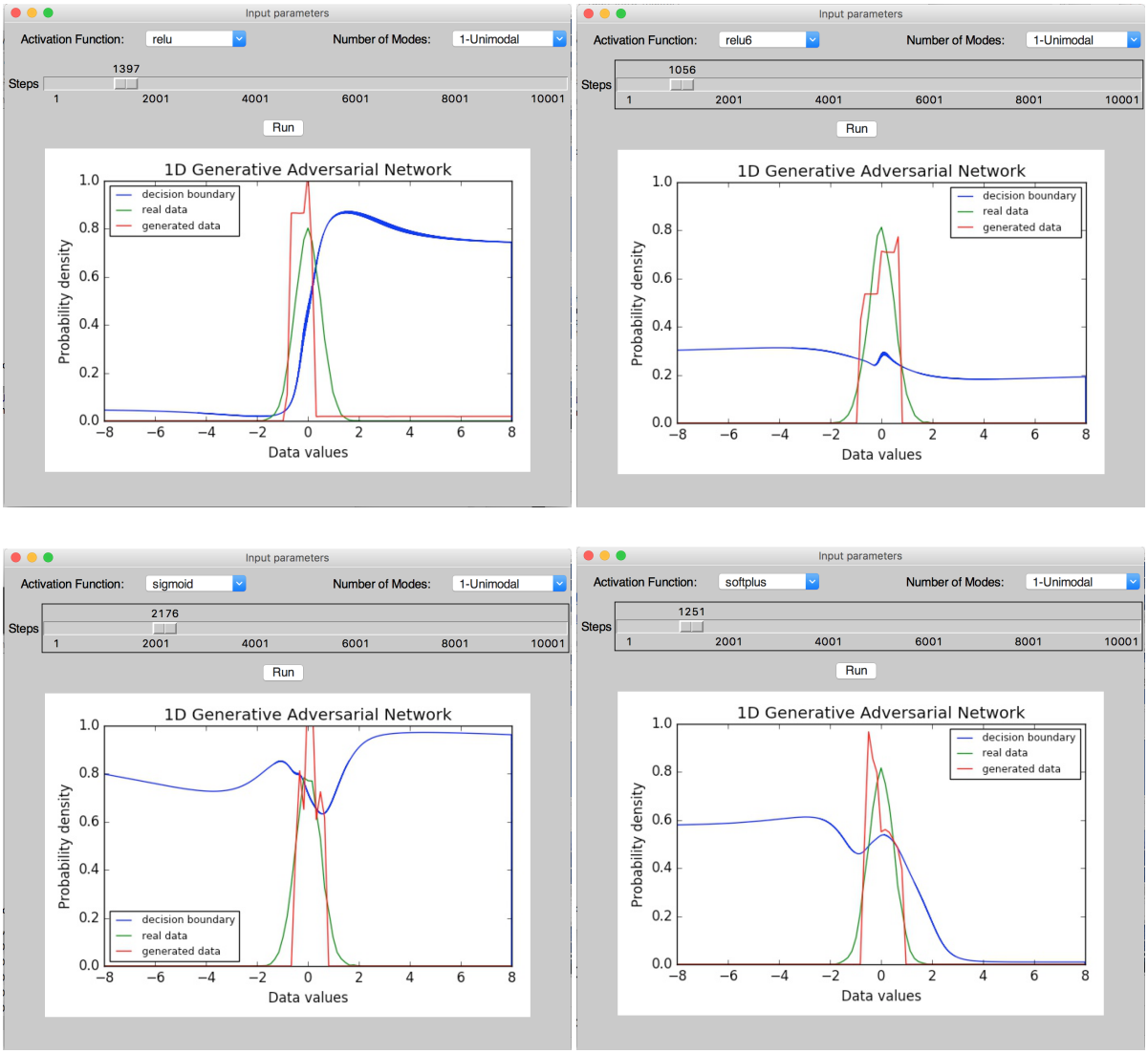
```
elif act_func == 4:
    h0 = tf.nn.softplus(linear(input, h_dim, 'g0'))
else:
    h0 = tf.sigmoid(linear(input, h_dim, 'g0'))
h3 = linear(h0, 1, 'g3')
return h3
```

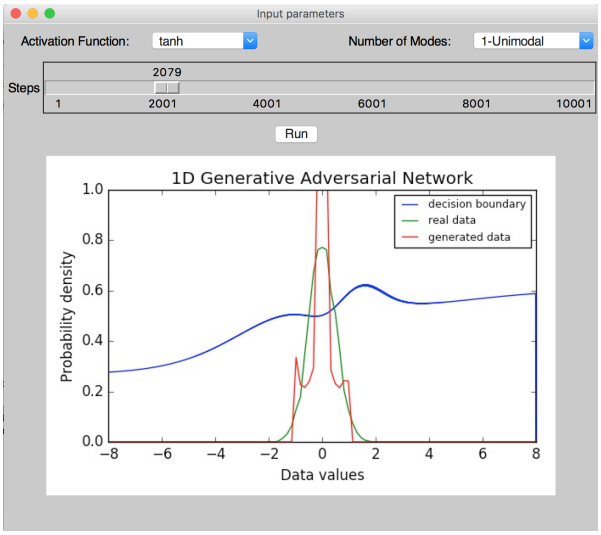
The discriminator has fixed layers with the first 2 being of tanh activation, the third layer based on the minibatch discrimination technique described by Tim Saliman et. all The output layer has sigmoid activation.

```
%example code- discriminator
h0 = tf.tanh(linear(input, h_dim * 2, 'd0'))
h1 = tf.tanh(linear(h0, h_dim * 2, 'd1'))
h2 = minibatch(h1)
h3 = tf.sigmoid(linear(h2, 1, scope='d3'))
return h3
```

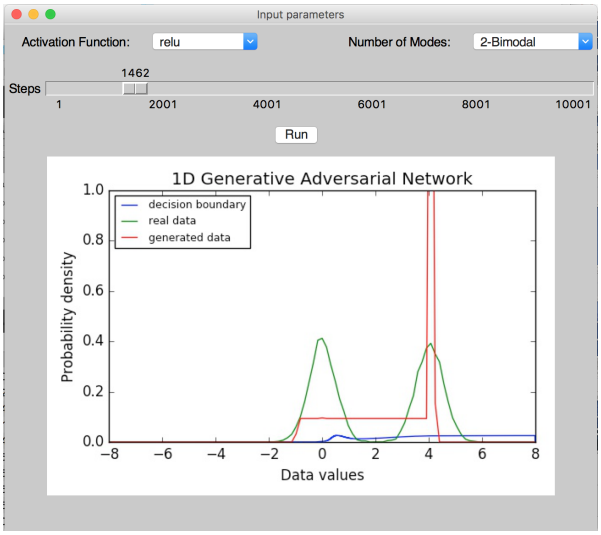
Results

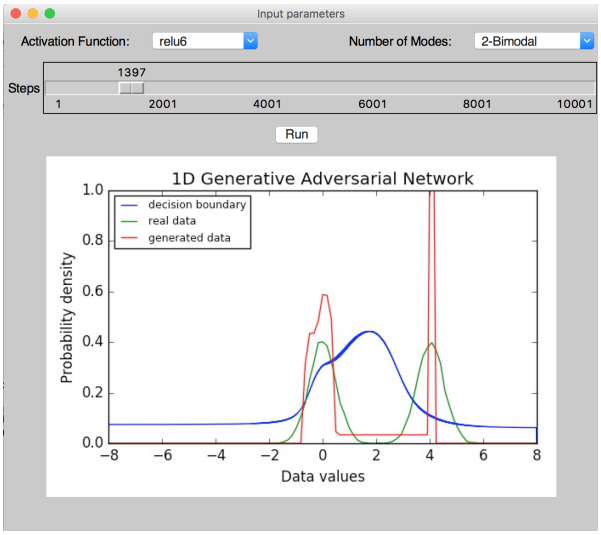
Sample runs using different activation functions for Unimodal Distribution

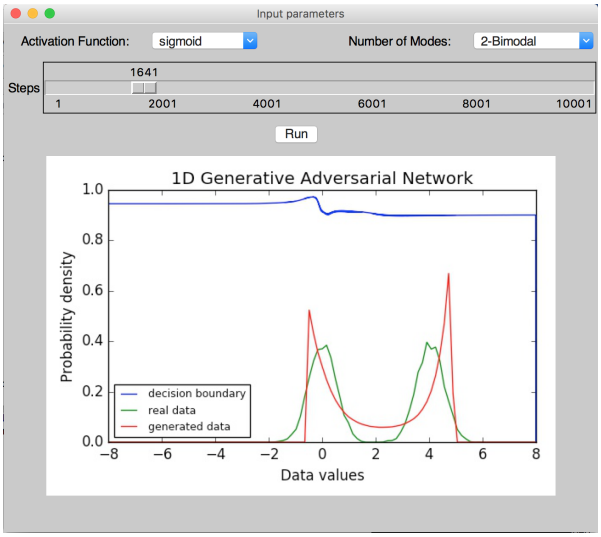


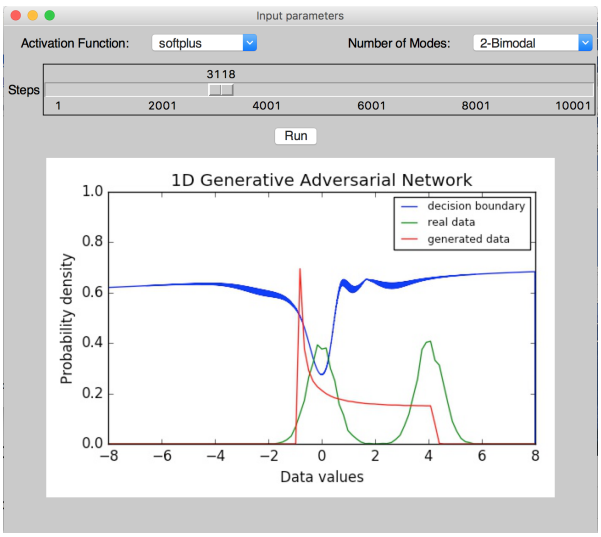


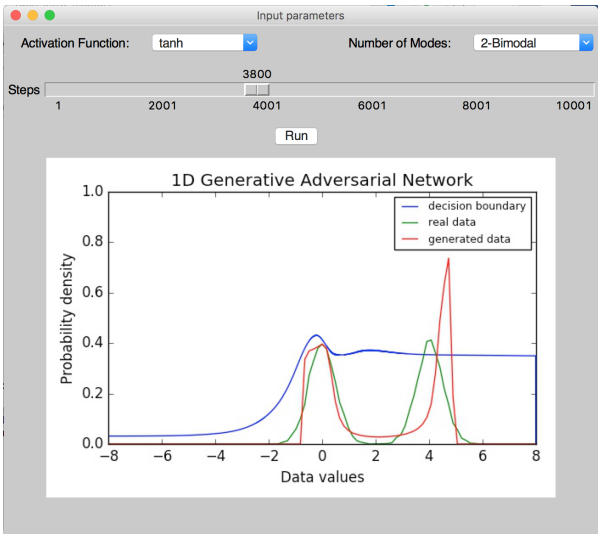
Sample runs using different activation functions for Bimodal Distribution



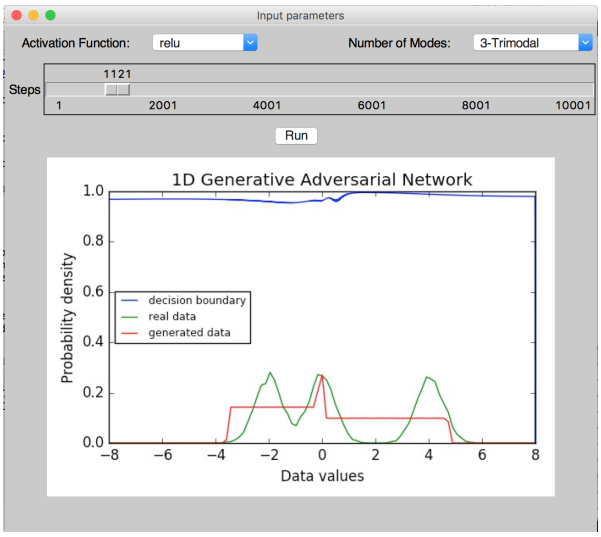


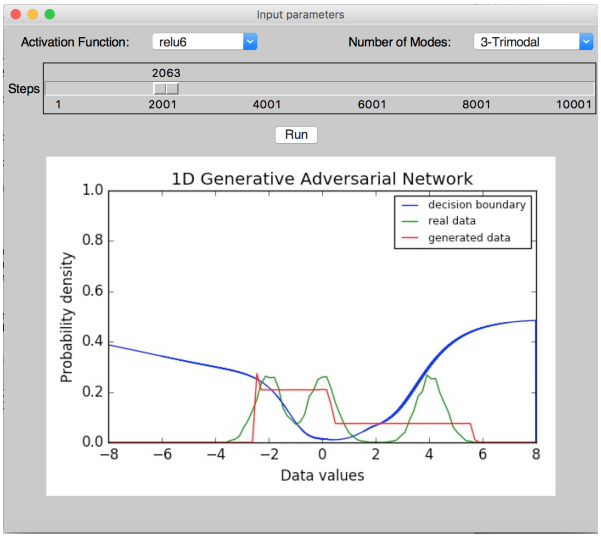


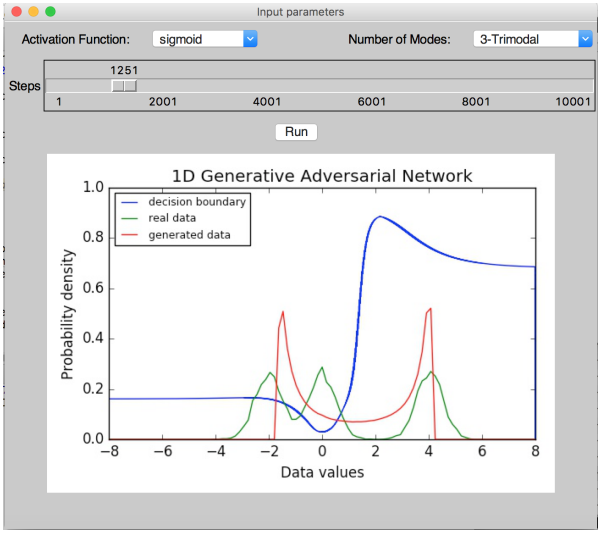


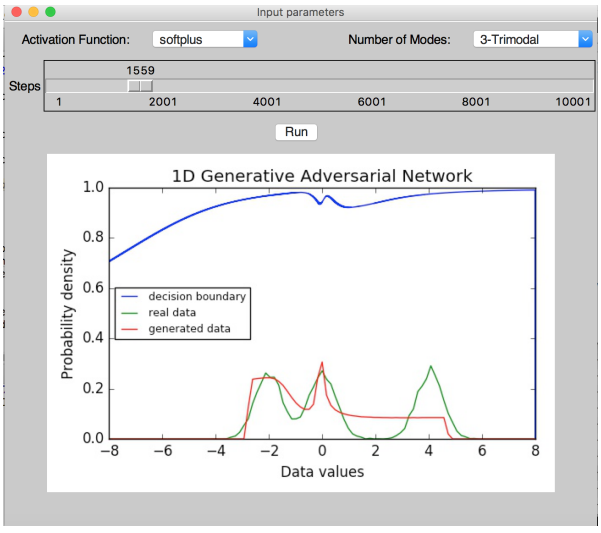


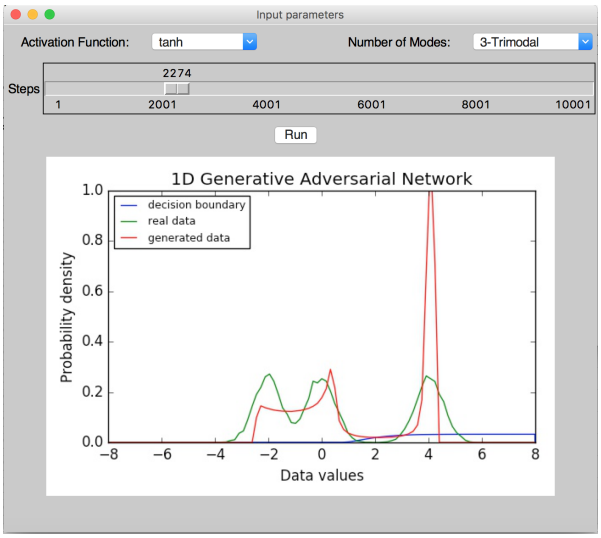
Sample runs using different activation functions for Trimodal Distribution











Observations

As seen from the results, most activation functions work all right for unimodal input. The relu and relu6 activations' outputs show the expected 'step' like features in the output. From the sample runs, it seems like softplus works well for Unimodal distributions.

We see a completely different scenario for bimodal distributions. Relu, relu6 and softplus activation functions don't seem to work very well. Relu and softplus do not recognize both modes of the distribution properly, while relu6 and relu6 output a distribution which is very narrow around the 2nd mode. Sigmoid and tanh activations give relatively better output since they recognize both the modes well.

Observations for trimodal distribution are rather interesting. Relu and Relu6 don't perform very well(as expected). But the sigmoid activation seems to only recognize 2 modes of the 3!. On the other hand although softplus activation recognizes only 2 of the 3 modes, the output seems to conform much better to the input distribution. Tanh activation performs relatively better by somewhat recognizing all the 3 modes, but it makes the output very narrow at the third mode.

It is undestandable when the model generates a narrow curve around the mode of the input distribution because generating values close the the mean would mean that the discriminator can't tell the real and generated data apart.

Discussion

Changing the architecture in terms of number of layers in the generator and discriminator, and varying with the number of steps will definitely give different results. It is likely that a deeper network is required for the generator to be able to generate more complex distributions. The application can easily be extended to play around with the number of layers in the generator and discriminator as well as the batch size for minibatch discrimination and activation for the discriminator layers.

References

1. Generative Adversarial Nets by Ian J. Goodfellow, et all: arXiv:1406.2661v1 [stat.ML] 10 Jun 2014
2. NIPS 2016 Tutorial: Generative Adversarial Networks by Ian Goodfellow: arXiv:1701.00160v3 [cs.LG] 9 Jan 2017
3. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks by Alec Radford, Luke Metz, Soumith Chintala: arXiv:1511.06434v2 [cs.LG] 7 Jan 2016

4. Generative Adversarial Text to Image Synthesis by Scott Reed, Zeynep Akata, Xinchen Yan, Lajanugen Logeswaran,Bernt Schiele, Honglak Lee: arXiv:1605.05396v2 [cs.NE] 5 Jun 2016
5. https://blog.openai.com/generative-models/
6. http://blog.aylien.com/introduction-generative-adversarial-networks-code-tensorflow/
7. https://medium.com/@devnag/generative-adversarial-networks-gans-in-50-lines-of-code-pytorch-e81b79659e3f