Conditional Generative Adversarial Network

A Generative Adversarial Network (GAN) is a type of generative model. It consists of two parts called the "generator" and the "discriminator". The generator takes random values as input and transforms them into an output that (hopefully) resembles the training data. The discriminator takes a set of samples as input and tries to distinguish the real training samples from the ones created by the generator. Both of them are trained together. The discriminator tries to get better and better at telling real from false data, while the generator tries to get better and better at fooling the discriminator.

A Conditional GAN (CGAN) allows additional inputs to the generator and discriminator that their output is conditioned on. For example, this might be a class label, and the GAN tries to learn how the data distribution varies between classes.

Colab

This tutorial and the rest in this sequence are designed to be done in Google colab. If you'd like to open this notebook in colab, you can use the following link.

Setup

To run DeepChem within Colab, you'll need to run the following cell of installation commands.

In [ ]:
```
!pip install --pre deepchem
import deepchem
deepchem.__version__
```

For this example, we will create a data distribution consisting of a set of ellipses in 2D, each with a random position, shape, and orientation. Each class corresponds to a different ellipse. Let's randomly generate the ellipses. For each one we select a random center position, X and Y size, and rotation angle. We then create a transformation matrix that maps the unit circle to the ellipse.

In [1]:
```
import deepchem as dc
import numpy as np
import tensorflow as tf

n_classes = 4
class_centers = np.random.uniform(-4, 4, (n_classes, 2))
class_transforms = []
for i in range(n_classes):
    xscale = np.random.uniform(0.5, 2)
    yscale = np.random.uniform(0.5, 2)
    angle = np.random.uniform(0, np.pi)
    m = [[xscale*np.cos(angle), -yscale*np.sin(angle)],
         [xscale*np.sin(angle), yscale*np.cos(angle)]]
    class_transforms.append(m)
class_transforms = np.array(class_transforms)
```

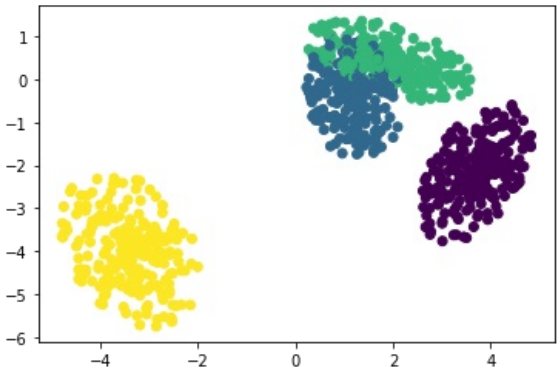
This function generates random data from the distribution. For each point it chooses a random class, then a random position in that class' ellipse.

In [2]:
```
def generate_data(n_points):
    classes = np.random.randint(n_classes, size=n_points)
    r = np.random.random(n_points)
    angle = 2*np.pi*np.random.random(n_points)
    points = (r*np.array([np.cos(angle), np.sin(angle)])).T
    points = np.einsum('ijk,ik->ij', class_transforms[classes], points)
    points += class_centers[classes]
    return classes, points
```

Let's plot a bunch of random points drawn from this distribution to see what it looks like. Points are colored based on their class label.

In [3]:
```
%matplotlib inline
import matplotlib.pyplot as plot
classes, points = generate_data(1000)
plot.scatter(x=points[:,0], y=points[:,1], c=classes)
```

Out[3]:
```
<matplotlib.collections.PathCollection at 0x1584692d0>
```



Now let's create the model for our CGAN. DeepChem's GAN class makes this very easy. We just subclass it and implement a few methods. The two most important are:

- `create_generator()` constructs a model implementing the generator. The model takes as input a batch of random noise plus any condition variables (in our case, the one-hot encoded class of each sample). Its output is a synthetic sample that is supposed to resemble the training data.
- `create_discriminator()` constructs a model implementing the discriminator. The model takes as input the samples to evaluate (which might be either real training data or synthetic samples created by the generator) and the condition variables. Its output is a single number for each sample, which will be interpreted as the probability that the sample is real training data.

In this case, we use very simple models. They just concatenate the inputs together and pass them through a few dense layers. Notice that the final layer of the discriminator uses a sigmoid activation. This ensures it produces an output between 0 and 1 that can be interpreted as a probability.

We also need to implement a few methods that define the shapes of the various inputs. We specify that the random noise provided to the generator should consist of ten numbers for each sample; that each data sample consists of two numbers (the X and Y coordinates of a point in 2D); and that the conditional input consists of `n_classes` numbers for each sample (the one-hot encoded class index).

In [4]:
```
from tensorflow.keras.layers import Concatenate, Dense, Input

class ExampleGAN(dc.models.GAN):

  def get_noise_input_shape(self):
    return (10,)

  def get_data_input_shapes(self):
    return [(2,)]

  def get_conditional_input_shapes(self):
    return [(n_classes,)]

  def create_generator(self):
    noise_in = Input(shape=(10,))
    conditional_in = Input(shape=(n_classes,))
    gen_in = Concatenate()([noise_in, conditional_in])
    gen_dense1 = Dense(30, activation=tf.nn.relu)(gen_in)
    gen_dense2 = Dense(30, activation=tf.nn.relu)(gen_dense1)
    generator_points = Dense(2)(gen_dense2)
    return tf.keras.Model(inputs=[noise_in, conditional_in], outputs=[generator_points])

  def create_discriminator(self):
    data_in = Input(shape=(2,))
    conditional_in = Input(shape=(n_classes,))
    discrim_in = Concatenate()([data_in, conditional_in])
    discrim_dense1 = Dense(30, activation=tf.nn.relu)(discrim_in)
    discrim_dense2 = Dense(30, activation=tf.nn.relu)(discrim_dense1)
    discrim_prob = Dense(1, activation=tf.sigmoid)(discrim_dense2)
    return tf.keras.Model(inputs=[data_in, conditional_in], outputs=[discrim_prob])

gan = ExampleGAN(learning_rate=1e-4)
```

Now to fit the model. We do this by calling `fit_gan()`. The argument is an iterator that produces batches of training data. More specifically, it needs to produce dicts that map all data inputs and conditional inputs to the values to use for them. In our case we can easily create as much random data as we need, so we define a generator that calls the `generate_data()` function defined above for each new batch.

In [5]:
```
def iterbatches(batches):
  for i in range(batches):
```

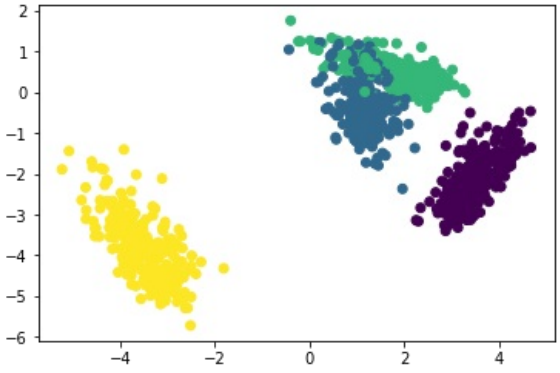
```
    classes, points = generate_data(gan.batch_size)
    classes = dc.metrics.to_one_hot(classes, n_classes)
    yield {gan.data_inputs[0]: points, gan.conditional_inputs[0]: classes}

gan.fit_gan(iterbatches(5000))
```

```
Ending global_step 999: generator average loss 0.87121, discriminator average loss 1.08472
Ending global_step 1999: generator average loss 0.968357, discriminator average loss 1.17393
Ending global_step 2999: generator average loss 0.710444, discriminator average loss 1.37858
Ending global_step 3999: generator average loss 0.699195, discriminator average loss 1.38131
Ending global_step 4999: generator average loss 0.694203, discriminator average loss 1.3871
TIMING: model fitting took 31.352 s
```

Have the trained model generate some data, and see how well it matches the training distribution we plotted before.

In [6]:
```
classes, points = generate_data(1000)
one_hot_classes = dc.metrics.to_one_hot(classes, n_classes)
gen_points = gan.predict_gan_generator(conditional_inputs=[one_hot_classes])
plot.scatter(x=gen_points[:,0], y=gen_points[:,1], c=classes)
```

Out[6]:
```
<matplotlib.collections.PathCollection at 0x160dedf50>
```



Congratulations! Time to join the Community!

Congratulations on completing this tutorial notebook! If you enjoyed working through the tutorial, and want to continue working with DeepChem, we encourage you to finish the rest of the tutorials in this series. You can also help the DeepChem community in the following ways:

Star DeepChem on GitHub

This helps build awareness of the DeepChem project and the tools for open source drug discovery that we're trying to build.

Join the DeepChem Discord

The DeepChem Discord hosts a number of scientists, developers, and enthusiasts interested in deep learning for the life sciences. Join the conversation!