Neural network programming: generative adversarial networks

Erik Spence

SciNet HPC Consortium

14 May 2024

Today's code and slides

You can get the slides and code for today's class at the SciNet Education web page.

`https://scinet.courses/1327`

Click on the link for the class, and look under "Lectures", click on "GANs".

Today's class

This class will cover the following topics:

- Generative adversarial networks.
- Example.

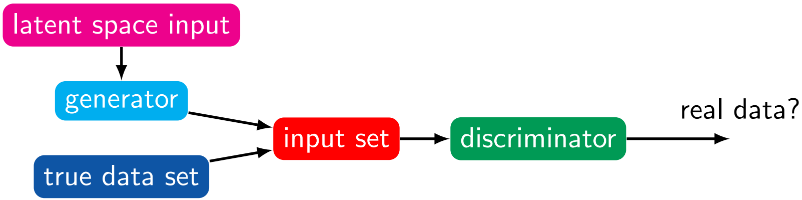
Please ask questions if something isn't clear.

Generative Adversarial Networks (2014)

What are Generative Adversarial Networks (GANs)?

- GANs are another type of generative network, introduced by Goodfellow and collaborators, U. de Montréal.
- A GAN consists of two coupled networks, the "discriminator" and the "generator".
- The generator takes a latent space vector (random noise) as input, and generates fake data to be fed into the discriminator.
- The discriminator is a standard discriminating neural network.
- The system is called "adversarial" because the two networks are treated as adversaries:
  - The discriminator is trained to learn whether a given input, $\mathbf{x}$, is authentic data from a real data set, rather than fake data created by the generator.
  - The generator is trained to try to fool the discriminator into thinking its output comes from the real data set.
- The two networks are trained alternately. Eventually (if all goes well) the output of the generator will become very similar to that of the input data set.

GAN schematic



The discriminator is given a mixed data set of real data from the true data set and fake data from the generator.

GANs can do amazing things

https://thispersondoesnotexist.com

Training GANs

Training both networks simultaneously must require coupling them together. How is this done?

- Let the discriminator, D, take as its input $\mathbf{x}$ and has weights and biases $\boldsymbol{\theta_D}$.
- Let the generator, G, take as its input $\mathbf{z}$ and has weights and biases $\boldsymbol{\theta_G}$.
- We wish to minimize the discriminator's cost function $C_D(\boldsymbol{\theta_D}, \boldsymbol{\theta_G})$, but the discriminator only has control over $\boldsymbol{\theta_D}$.
- Similarly, we wish to minimize the generator's cost function $C_G(\boldsymbol{\theta_D}, \boldsymbol{\theta_G})$, but the generator only has control over $\boldsymbol{\theta_G}$.
- Formally, because the two networks are trying to reach an equilibrium, rather than a minimum, the goal is to find a Nash equilibrium.

Training GANs, continued

The original algorithm called for Stochastic Gradient Descent (SGD) to train the networks.

- At each step, two minibatches are sampled.
  - A batch of $\mathbf{x}$ values from the true data set.
  - A batch of random values $\mathbf{z}$, which are then used to generate fake data, using the generator.
- We then perform two steps alternatively.
  - We update $\boldsymbol{\theta_D}$ to reduce $\boldsymbol{C_D}$, based on both real and fake data.
  - We update $\boldsymbol{\theta_G}$ to reduce $\boldsymbol{C_G}$.
- In the original GAN algorithm, the cost function for the discriminator is always the same, cross-entropy:

$$\boldsymbol{C_D(\theta_D, \theta_G) = -\frac{1}{2}\sum_i^N \log\left(D(x_i)\right) - \frac{1}{2}\sum_i^N \log\left(1 - D(G(z_i))\right)}$$

We have assumed $\boldsymbol{2N}$ data points in each minibatch, half of which are from the real data set.

Training GANs, continued more

What cost function do we use for the generator? Several have been proposed.

- One option is the "zero-sum game": $C_G = -C_D$.
- Another option is to flip the target used to construct the cross-entropy:
  $$C_G = -\frac{1}{2}\sum_i^N \log\left(D(G(z_i))\right).$$
- The motivation for this function is to ensure that the losing side has a strong gradient.
- Maximum likelihood: $C_G = -\frac{1}{2}\sum_i^N e^{\sigma^{-1}(D(G(z_i)))}$

Where σ is the usual sigmoid function.

We will use a different approach, where we use the Discriminator's cost function by training the Generator through the Discriminator.

Training failures

As you might at first intuitively expect, training GANs is non-trivial.

- Rather than minimizing a cost function, we're trying to balance two competing minimizations.
- This is, more often than not, unstable.
  - The generator can 'collapse' (fail to generate convincing data) resulting in the discriminator getting a perfect score.
  - The discriminator can converge to zero, and the generator stops training.
- Overcoming these problems requires extremely careful choice of hyperparameters.

GANs also suffer from other training problems:

- mode collapse: the generator latches on to a single feature of the input data and ignores all others.
- convergence ambiguity: how do we tell if things are converging? There's no single metric; the loss values don't help.

GAN example

Let's build a GAN. What problem will we tackle?

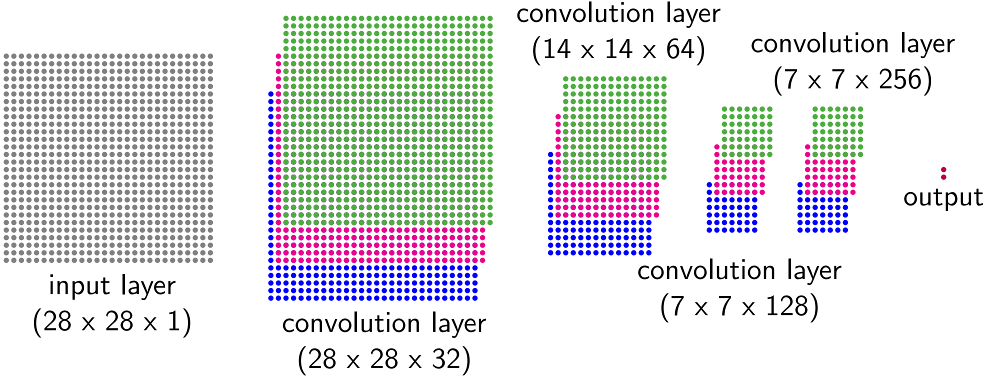
- Let's work on our old friend, the MNIST data set.
- As you recall, these are 60000 28 x 28 pixel images of hand-written digits, in greyscale.
- There are many many types of GANs out there. This one will be a Deep Convolutional GAN (DCGAN).
- The goal will be for the network to generate images of hand-written digits which are convincing.

Our discriminator

First we need a discriminator.

- The input data is (28 x 28 x 1) (greyscale).
- We then put in 4 convolution layers, each of which has a 5 x 5 filter, with strides of 1 or 2, and different numbers of feature maps.
- We use the leaky ReLU as the activation function.
- Dropout is used on all the layers.
- We then flatten the last layer and input it into the output layers, containing 2 neurons.
- Recall that the discriminator just needs to indicate whether the input image is real or fake.

Our discriminator, continued



The number of convolutional layer feature maps is given by the third number in the brackets.

Our discriminator, the code

```
# MNIST_gan.py
import tensorflow.keras.models as km
import tensorflow.keras.layers as kl
import tensorflow.keras.utils as ku
import tensorflow.keras.optimizers as ko

def add_D_layers(in, fm_num, stride):

  x = kl.Conv2D(fm_num,
    kernel_size = (5, 5),
    strides = stride,
    padding = "same")(in)

  x = kl.LeakyReLU()(x)
  x = kl.Dropout(0.3)(x)

  return x
```

```
# Create the discriminator.
def create_D():
  input_image = kl.Input(shape = (28, 28, 1))
  x = add_D_layers(input_image, 32, 1)
  x = add_D_layers(x, 64, 2)
  x = add_D_layers(x, 128, 2)
  x = add_D_layers(x, 256, 1)

  last = kl.Flatten()(x)
  output = kl.Dense(2, activation = "softmax")(last)

  model = km.Model(inputs = input_image, name = 'D',
    outputs = output)

  model.compile(optimizer = ko.Adam(1e-4),
    loss = 'categorical_crossentropy')

  return model
```

Other activation functions: leaky ReLU

Two commonly-used functions:

- Rectifier Linear Units (ReLUs):

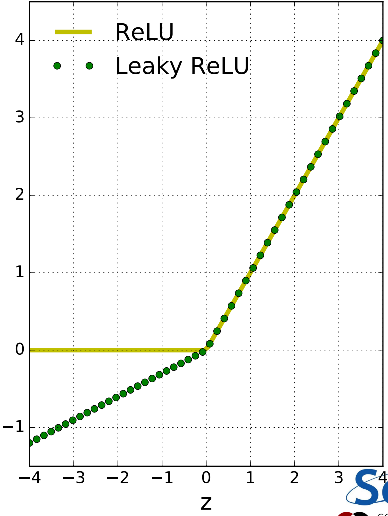
  $$f(z) = \max(0, z).$$

- Leaky ReLU:

$$f(z) = \begin{cases} z & z > 0 \\ \alpha z & z \leq 0 \end{cases}$$

for $\alpha > 0$.

Leaky ReLUs have gradients for $z < 0$, which is usually advantageous.

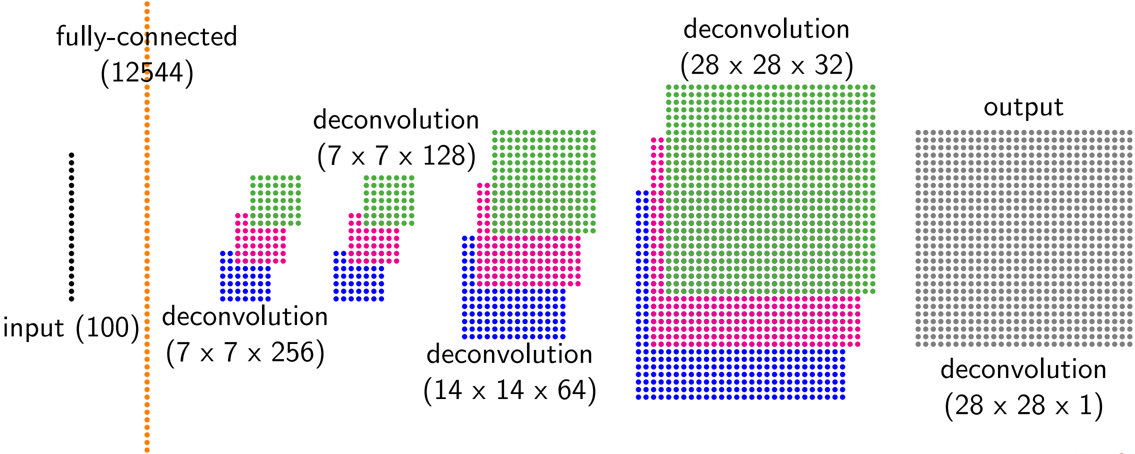


Our generator

How shall we construct our generator?

- We have a single input, the latent space input (a vector of Gaussian noise).
- Feed this into a fully-connected layer.
- Reshape the layer's output into a square.
- Repeatedly apply transposed convolution to it, while shrinking the number of feature maps, until we get to (28 x 28 x 1).

Our generator, continued



Our generator, the code

```
# MNIST_gan.py, continued

def add_G_layers(in, fm_num, stride):

  x = kl.Conv2DTranspose(fm_num,
    kernel_size = (5, 5),
    padding = "same",
    strides = stride)(in)

  x = kl.BatchNormalization()(x)
  x = kl.LeakyReLU()(x)

  return x
```

```
def create_G():
  input_z = kl.Input(shape = (100,))

  x = kl.Dense(256 * 7 * 7)(input_z)
  x = kl.BatchNormalization()(x)
  x = kl.LeakyReLU()(x)

  x = kl.Reshape((7, 7, 256))(x)

  x = add_G_layers(x, 256, 1)
  x = add_G_layers(x, 128, 1)
  x = add_G_layers(x, 64, 2)
  x = add_G_layers(x, 32, 2)

  x = kl.Conv2DTranspose(1, (5, 5), padding = "same",
    activation = "tanh")(x)

  return km.Model(inputs = input_z, outputs = x)
```

Training our GAN

The algorithm for training the GAN is as follows.

- Create the input layer for the discriminator.
- Create the discriminator (D) and generator (G).
- Create a combined discriminator-generator (DG) network.
- Turn off the training of the discriminator.
- Compile the DG network.
- Now iterate:
  - Create fake data, using G.
  - Train D on real and new fake data.
  - Turn off training of D.
  - Train the combined DG network so as to train G to create authentic images.
  - Turn training for D back on.

Training our GAN, the code

```
# MNIST_gan.py, continued

import tensorflow.keras.backend as K
import numpy as np
import numpy.random as npr

# Create the generator input layers.
input_z = kl.Input(shape = (100,))

# Create the networks.
D = create_D()
G = create_G()
```

```
# MNIST_gan.py, continued

# Create the combined network.
output = D(G(inputs = input_z))

DG = km.Model(inputs = input_z, outputs = output)

# Turn off D before compiling.
DG.get_layer("D").trainable = False

# Compile the generator.
DG.compile(optimizer = ko.Adam(lr = 1e-4),
  loss = "categorical_crossentropy")
```

Training our GAN, the code, continued

```
# MNIST_gan.py, continued
for it in range(num_epochs):
  for image_batch in train_dataset:

    # Turn on D.
    D.trainable = True
    for l in D.layers: l.trainable = True

    # Create some fake images.
    zz = npr.normal(0., 1., (batch_size, 100))
    f_images = G.predict(zz)

    all_images = np.concatenate([f_images,
      image_batch])

    all_cats = np.contatenate([np.zeros(batch_size),
      np.ones(image_batch.shape[0])])
    all_cats = ku.to_categorical(all_cats, 2)
```

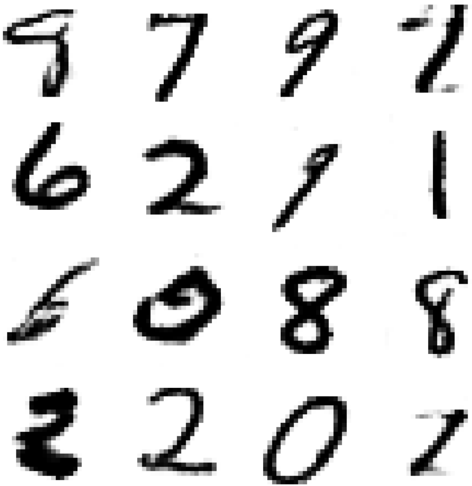
```
# MNIST_gan.py, continued

    # Train on the mages.
    D_loss = D.train_on_batch(all_images,
      all_cats)

    # We are done training D. Now train G.
    D.trainable = False
    for l in D.layers: l.trainable = False

    # Create some input.
    zz = npr.normal(0., 1., (batch_size, 100))

    # Train DG on the fake images.
    DG_loss = DG.train_on_batch(zz,
      ku.to_categorical(np.ones(batch_size),2))

    # Now save the losses and images.
```

Training our GAN, running

This takes about 3 hours on a GPU.

```
ejspence@mycomp ~>
ejspence@mycomp ~> python MNIST_gan.py
0: [D loss:  0.039339] [DG loss:  0.055060]
1: [D loss:  0.020271] [DG loss:  0.150696]
2: [D loss:  0.038817] [DG loss:  5.117784]
3: [D loss:  0.365811] [DG loss:  2.477790]
.
.
.
496: [D loss:  0.617432] [DG loss:  1.407433]
497: [D loss:  0.625149] [DG loss:  1.447843]
498: [D loss:  0.621429] [DG loss:  1.181722]
499: [D loss:  0.596228] [DG loss:  1.274543]
ejspence@mycomp ~>
```

Our GAN, results



Some final GAN notes

Some notes about the example, and GANs.

- This took many attempts to get to work. Training failures aren't uncommon.
- Since the GAN paper was published, man better GAN techniques have been introduced.
- There are zillions of variations on the GAN. Check out the "GAN zoo" if you're interested.
- There is talk of using GANs to replace regular HPC.

The movement in the community is now away from GANs, and toward diffusion networks instead.

Linky goodness

GANs:

- https://arxiv.org/abs/1701.00160
- https://blog.openai.com/generative-models
- https://deephunt.in/the-gan-zoo-79597dc8c347
- http://arxiv.org/abs/1511.06434
- https://medium.com/towards-data-science/gan-by-example-using-keras-on-tensorflow-backend-1a6d515a60d0
- https://arxiv.org/abs/1606.03498
- https://arxiv.org/abs/1701.07875