Advanced neural networks: generative adversarial networks

Erik Spence

SciNet HPC Consortium

30 October 2017

Today's data, code and slides

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

`https://support.scinet.utoronto.ca/education`

Click on the link for the class, and look under "File Storage".

Discriminative versus generative networks

Let's examine a distinction we haven't yet made: discriminative versus generative networks.

- A discriminative network is trained to detect whether some input data is a member of a given class. Examples include the standard networks we've come to know and love, such as fully-connected, CNNs and LSTMs.
- In probabilistic terms, given the input data x, and a desired label y, the discriminative network calculates the conditional probability $P(y|\mathrm{x})$.
- In contrast, a generative network is trained to calculate the joint probability of the data and labels simultaneously, $P(\mathrm{x}, y)$.
- The joint probability can be converted into a posterior, $P(y|\mathrm{x})$, and thus used as a classifier.
- Alternatively, the generative abilities of the network can be used to create new (x, y) pairs.

Generative networks

There are several types of generative networks you may run into.

- PixelCNN: an auto-regressive model, the conditional distribution of each pixel is modeled given the left and above pixels.
- Variational Autoencoders: one network (the encoder) casts the input data into a lower-dimensional representation; a second network (the decoder) reconstructs the input from the low-D representation.
- Generative Adversarial networks: two networks are trained simultaneously, one to generate fake data, and one to identify the fake data, when compared to real data.
- Boltzmann Machines, Fully Visible Belief Networks, Generative Stochastic Networks, and others.

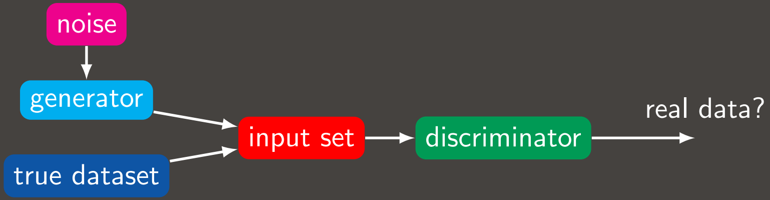
Today we'll be discussing Generative Adversarial Networks.

Generative Adversarial Networks

What are Generative Adversarial Networks (GANs)?

- A GAN consists of two coupled networks, the "discriminator" and the "generator".
- The generator takes 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 alternatively. Eventually (if all goes well) the output of the generator will become very similar to that of the input dataset.

GAN schematic



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

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 $\boldsymbol{\theta}_G$.
- The discriminator wishes to minimize the cost function $C_D(\boldsymbol{\theta}_D, \boldsymbol{\theta}_G)$, but only has control over $\boldsymbol{\theta}_D$.
- Similarly, the generator wishes to minimize the cost function $C_G(\boldsymbol{\theta}_D, \boldsymbol{\theta}_G)$, but only has control over $\boldsymbol{\theta}_G$.
- Formally, because the two networks are trying to reach an equillibrium, rather than a minimum, the goal is to find a Nash equillibrium.

Training GANs, continued

The original algorithm called for the usual 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 $\mathbf{z}$.
- We then perform two steps alternatively.
  - We update θ_D to reduce C_D, based on both real and fake input.
  - We update θ_G to reduce C_G.
- In the original GAN algorithm, the cost function for the discriminator is always the same, cross-entropy:

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

We have assumed $2N$ 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.

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' (its score goes to zero), 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.

Wasserstein GANs

The inherent difficulties with training GANs led to research into how to stabilize them. Nine months ago the "Wasserstein GAN" was introduced.

- Let the probability distribution of the real data be P_r, and the distribution we are trying to match to it be P_θ.
- The original GAN training algorithm attempted to optimize P_θ using the maximum likelihood estimation, which is equivalent to minimizing the KL divergence (a measure of the "distance" between two probability distributions).
- Wasserstein GANs instead are trained by minimizing the Wasserstein distance, which is a different measure of how far P_r and P_θ are from each other.
- This is more stable, since even if there is minimal overlap between P_r and P_θ at the beginning of training, you don't end up with the usual training problems.

See the original Wasserstein GAN paper for the details.

Wasserstein GANs, continued

How are Wasserstein GANs (WGANs) different from regular GANs?

- Remove all non-linear functions from the discriminator output. Instead just use the output of the nodes straight-up.
- The discriminator weights must lie in a "compact space". To enforce this the values are clipped so they remain inside some fixed range. (This is obviously not ideal; improvements to this have been developed.)
- The discriminator is trained for additional iterations between generator training sessions.
- Training algorithms which do not include 'momentum' are used (SGC, RMSProp).
- The Wasserstein loss function is used, which is the mean of the actual values (y) multiplied by the predicted ($\hat{y}$).

$$C = \frac{1}{m}\sum_{i}^{m} y_i \hat{y}_i$$

WGAN example

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

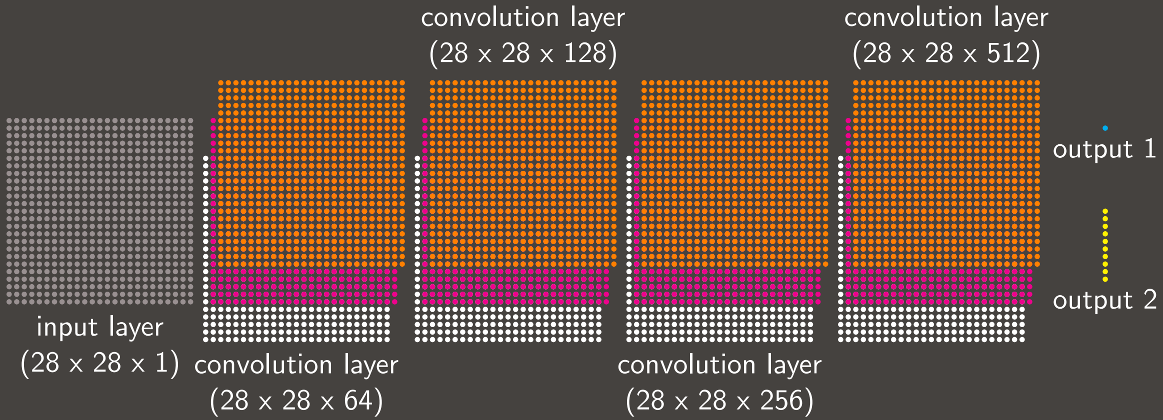
- Let's work on our old friend, the MNIST data set.
- As you recall, these are 50000 28 x 28 pixel images of hand-written digits, in greyscale.
- There are many many types of GANs out there. This one will be a Deep Convolutional GAN (DCGAN), of the Wasserstein variety.
- The goal will be for the network to generate images of hand-written digits which are convincing.

Our discriminator

First we need a discriminator. A deep one.

- 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, but with different strides 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 two output layers:
  - single output node, which is used to indicate whether the input image is real or fake.
  - a fully-connected layer of 10 nodes, used to indicate which digit is in the image.

Our discriminator, continued



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

Our discriminator, the code

```python
# MNIST_wgan.py
import keras.initializers as ki
import keras.models as km
import keras.layers as kl
import keras.optimizers as ko

def add_D_layers(in, fm_num, stride):

  w_init = ki.RandomNormal(
    stddev = 0.02)

  x = kl.Conv2D(fm_num, (5, 5),
    strides = stride, padding = "same",
    kernel_initializer = w_init)(in)
  x = kl.LeakyReLU(alpha = 0.2)(x)
  x = kl.Dropout(0.4)(x)

  return x
```

```python
# Create the discriminator.
def create_D():

  input_image = kl.Input(shape = (28, 28, 1))

  x = add_D_layers(input_image, 64, 2)
  x = add_D_layers(x, 128, 2);   x = add_D_layers(x, 256, 2)
  x = add_D_layers(x, 512, 1);   last = kl.Flatten()(x)

  output_status = kl.Dense(1, activation = "linear")(last)
  output_class = kl.Dense(10, activation = "softmax")(last)

  model = km.Model(inputs = input_image, name = 'D',
    outputs = [ouput_status, output_class])

  model.compile(optimizer = ko.RMSprop(lr = 5e-5),
    loss = [wasserstein, 'sparse_categorical_crossentropy'])
  return model
```

Other activation functions: leaky ReLU

Two commonly-used functions:

- Rectifier Linear Units (ReLUs):

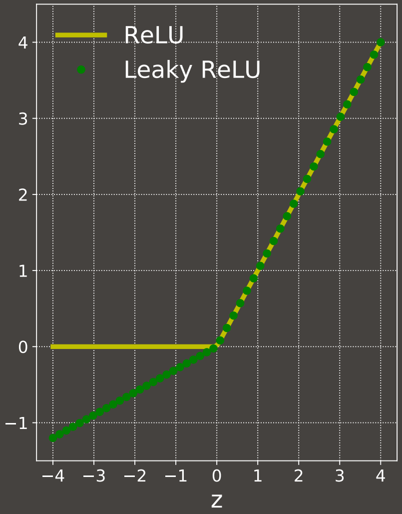
  $$f(z) = \max(0, z).$$

- 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.



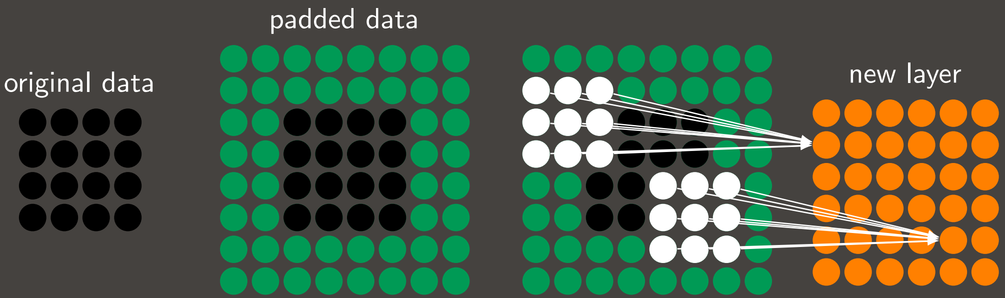
Deconvolution layers

Our generator takes noise and creates an image. As you might imagine, to do this we need to deconvolve the data. How does that work?

- Suppose we start with a 1D vector of random noise.
- Reshape the vector into a square.
- Convolution involves taking a, say, (5 x 5) square of an image and processing it into a (1 x 1) square (a single point).
- Deconvolution involves taking a say, (2 x 2) square of an image and processing it into something larger, (4 x 4) for example.
- There are several ways to do this. The "transpose convolution" technique involves either
  - padding the image with zeros around the outside,
  - or 'upsampling' the image to double its size in each dimension,

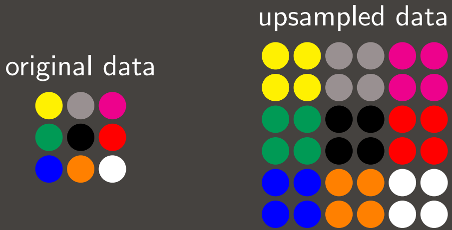
  and then doing the usual convolution.

Deconvolution layers, padding with zeros



The green padding consists of zeros. The deconvolution is the convolution of a padded version of the input data. It is equivalent to the transpose of the convolution of a (6 x 6) layer using a (3 x 3) filter, resulting in a (4 x 4) layer. The weights and biases of each deconvolution feature map are the same.

Deconvolution layers, upsampling



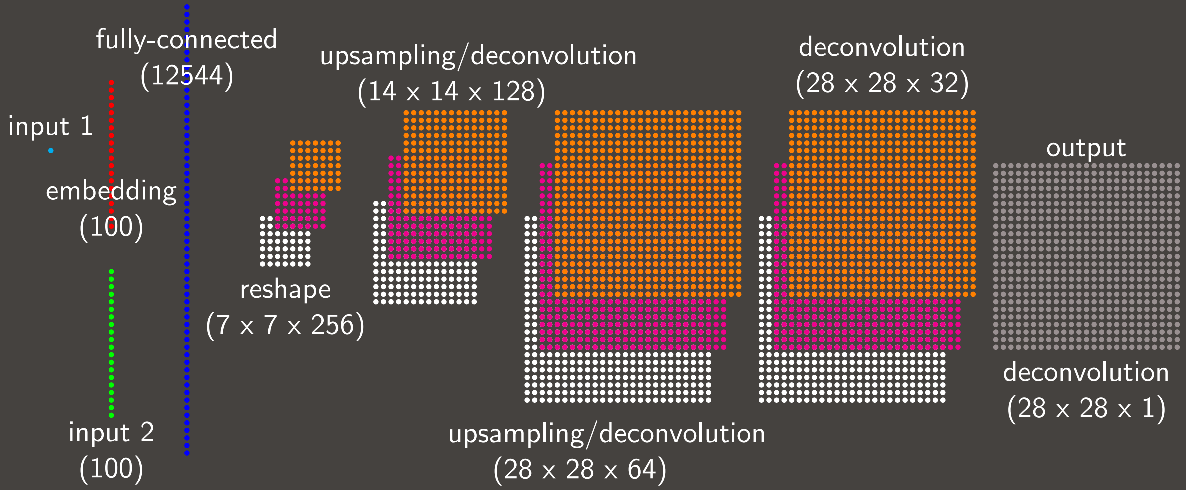
The default Keras upsampling layer doubles all data in all dimensions. Upsampling is often used instead of just padding with zeros, since there's more information available.

Our generator

How shall we construct our generator?

- The inputs will be a vector of Gaussian noise, and the digit we would like generated.
- Because we would like to be able to specify which digit (0-9) we would like the generator to generate, we create an embedding layer which will combine the noise input with weights for the specified digit.
- Feed this into a fully-connected layer.
- Reshape the layer's output into a square.
- Upsample the data, and then apply transposed convolution to it.
- Repeat the above step, using a different number of feature maps.
- Apply the transpose convolution again, without upsampling.
- And do it again, but with a single layer as the output.

Our generator, continued



Embedding layers

SciNet

- Embedding layers were developed for text analysis. We could have (and should have) used one in last class' example.
- By matrix multiplication, the layer transforms one-hot-encoded words into their corresponding "word embedding".
- The matrix of weights in the embedding layer is of the shape (vocabulary_size, embedding_dimension). These weights are learned as part of the NN training.
- As we saw last class, when each word in a sentence is one-hot-encoded the input becomes a matrix of shape (sentence_length, vocabulary_size).
- The embedding layer transforms each word-index i into the ith row of the embedding weights matrix, resulting in a matrix of shape (sentence_length, embedding_dimension).
- In our case there are 10 'words', the digits 0-9.
- When a digit is inputed into the generator, that particular row of weights is returned by the embedding layer, which is then multiplied by the noise input.

Our generator, the code

```
# MNIST_wgan.py, continued
def add_G_layers(in, fm_num, upsample):

  w_init = ki.RandomNormal(stddev = 0.02)
  if upsample:
    x = kl.UpSampling2D(size = (2, 2))(in)
  else::  x = in

  x = kl.Conv2DTranspose(fm_num, (5, 5),
    padding = "same",
    kernel_initializer = w_init)(x)

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

  return x
```

```
# Create the generator.
def create_G():
  input_class = kl.Input(shape = (1,))
  input_z = kl.Input(shape = (100,))

  e = kl.Embedding(10, 100)(input_class)
  embedding_class = kl.Flatten()(e)

  h = kl.multiply([input_z, embedded_class])
  x = kl.Dense(256 * 7 * 7)(h)
  x = kl.LeakyReLU()(x)
  x = kl.Reshape((7, 7, 256))(x)
  x = add_G_layers(x, 128, True)
  x = add_G_layers(x, 64, True)
  x = add_G_layers(x, 32, False)
  x = add_G_layers(x, 1, False)
  return km.Model(inputs = [input_z, input_class],
    outputs = x)
```

Training our WGAN

The algorithm for training the WGAN is as follows.

- Create the two input layers for the discriminator.
- Create the discriminator (D) and generator (G).
- Create a combined discriminator-generator (DG) network.
- Turn off training of the discriminator.
- Compile the DG network.
- Now iterate:
  - Train D on real and new fake data. Clip the weights. Repeat several times until D is well converged.
  - 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 WGAN, the code

```python
# MNIST_wgan.py, continued
import keras.backend as K
import numpy as np
import numpy.random as npr

def wasserstein(y_true, y_pred):
  return K.mean(y_true * y_pred)

# ################################

# Create the input layers.
input_z = kl.Input(shape = (100,))
input_class = kl.Input(shape = (1,),
  dtype = 'int32')

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

```python
# MNIST_wgan.py, continued

# Create the combined network.
output_status, output_class =
  D(G(inputs = [input_z, input_class]))
DG = km.Model(inputs = [input_z, input_class],
  outputs = [output_status, output_class])

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

# Compile the generator.
DG.compile(optimizer = ko.RMSprop(lr = 5e-5),
  loss = [wasserstein,
  "sparse_categorical_crossentropy"])
```

Training our WGAN, the code, continued

```
# MNIST_wgan.py, continued
for it in range(25000):
  for d_it in range(100):

    # Turn on D, and clip weights.
    D.trainable = True
    for l in D.layers:
      l.trainable = True
      weights = l.get_weights()
      weights = [np.clip(w, -0.01, 0.01)
        for w in in weights]
      l.set_weights(weights)

    # Pick some random real images.
    indices = npr.choice(len(X_train,
      batch_size, replace = False)
    r_images = X_train[indices]
    r_classes = y_train[indices]
```

```
# MNIST_wgan.py, continued

    # Train on the real images.
    D_loss = D.train_on_batch(r_images,
      [-np.ones(batch_size), r_classes])

    # Create some fake images.
    zz = npr.normal(0., 1., (batch_size, 100))
    f_classes = npr.randint(0, 10, batch_size)
    f_images = G.predict([zz, f_classes])

    # Train on the fake images.
    D_loss = D.train_on_batch(f_images,
      [np.ones(batch_size), f_classes])
```

Training our WGAN, the code, continued more

Note that the discriminator is trained for many iterations for each training of the generator.

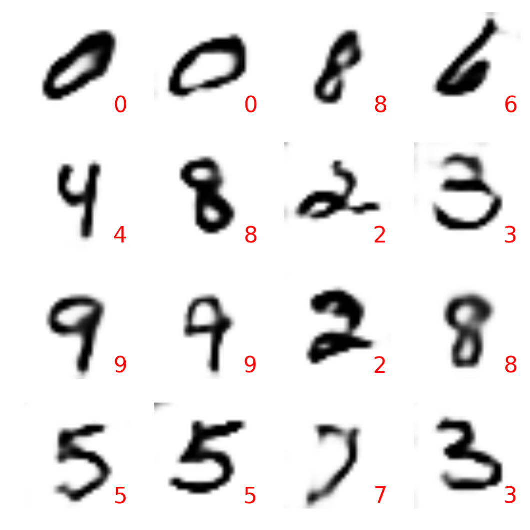
```
# MNIST_wgan.py, continued

  # 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))
  f_classes = npr.randint(0, 10, batch_size)

  # Train DG on the fake images.
  DG_loss = DG.train_on_batch(
    [zz, f_classes],
    [-np.ones(batch_size), f_classes])

  # Now save the losses and images.
```

Training our WGAN, running

This takes about 3 hours on a GPU.

```
ejspence@mycomp ~>
ejspence@mycomp ~> python MNIST_wgan.py
Using Tensorflow backend.
0: [D true loss: -2837.692] [D fake loss: 3.050] [DG loss: 2.636] [Total loss: -2834.641]
50: [D true loss: -1849.921] [D fake loss: 1408.130] [DG loss: -1282.885] [Total loss: -441.791]
100: [D true loss: -544.046] [D fake loss: -372.518] [DG loss: 513.699] [Total loss: -916.565]
150: [D true loss: -4682.79] [D fake loss: 3663.87] [DG loss: -3454.65] [Total loss: -1018.92]
.
.
.
24750: [D true loss: 0.1663] [D fake loss: 0.0420] [DG loss: 0.0335] [Total loss: 0.2083]
24800: [D true loss: 0.1830] [D fake loss: 0.0254] [DG loss: 0.0297] [Total loss: 0.2084]
24850: [D true loss: 0.0796] [D fake loss: 0.0384] [DG loss: 0.0538] [Total loss: 0.1181]
24900: [D true loss: 0.1067] [D fake loss: 0.0285] [DG loss: 0.0774] [Total loss: 0.1352]
24950: [D true loss: 0.0455] [D fake loss: 0.0729] [DG loss: 0.0829] [Total loss: 0.1185]
ejspence@mycomp ~>
```

Our WGAN, results



Some final GAN notes

Some notes about the example, and GANs.

- Even using the Wasserstein loss function this took many attempts to get to work.
- Since the WGAN paper was published, even better ("improved") WGAN 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.

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

WGAN:

- https://arxiv.org/abs/1701.07875 (original WGAN paper)
- https://arxiv.org/abs/1704.00028
- http://www.alexirpan.com/2017/02/22/wasserstein-gan.html