Neural network programming: variational autoencoders

Erik Spence

SciNet HPC Consortium

24 May 2018

Today's 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" and find the file "VAEs.pdf".

Today's class

This class will cover the following topics:

- Regular autoencoders.
- Variational autoencoders.
- KL divergence loss function.
- Example.

Please ask questions if something isn't clear.

Variational autoencoders

Variational autoencoders are another type of generative network.

- Generative adversarial networks are powerful, but are hard to train.
- It's easy enough to generate output which 'looks like' the training data, but we lack control over the details of the generated data.
- We would like to be able to specify some of the general details of our generated data: "create a face with glasses".
- This where variational autoencoders show their strength.

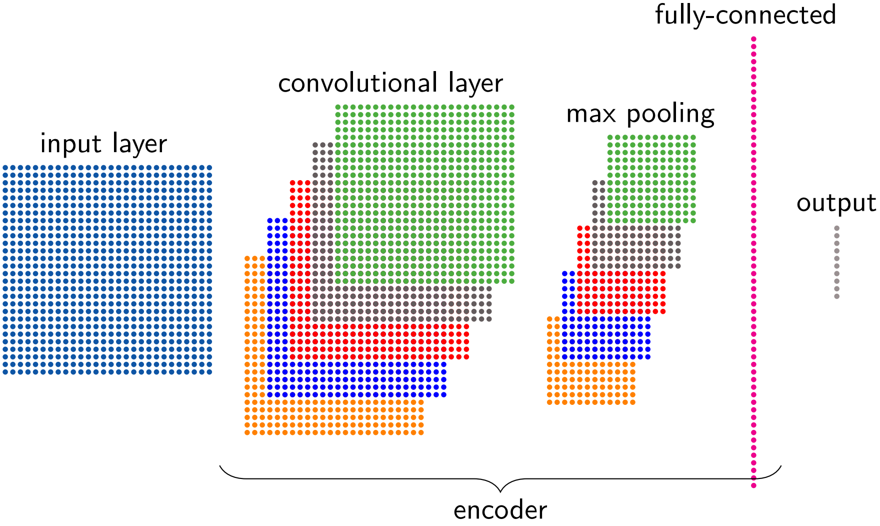
Variational autoencoders can give you a level of control over your generative networks that no other class of networks can.

Standard autoencoders

A standard autoencoder network is actually a pair of connected networks.

- These are an 'encoder' and a 'decoder'.
- The encoder network takes an input from a data set and converts it into a lower-dimensional form (it is compressed into a 'latent space'). In some sense the encoding is a compression technique, based on the data itself.
- We've seen encoders before, we just didn't call them that.
- CNNs are encoders, at least the convolution/pooling sections of the networks.
- These convert the image into a dense form that is then used for classification, for example.
- In a standard autoencoder, the encoder is similar, in that it 'encodes' the input into a form which can be used as input into a another neural network.

Our old network



Standard autoencoders, continued

Standard autoencoders take this one step further: add a second network, the decoder, whose job is to take the encoded output (latent variable) and decode it back into the original input data.

The goal is to make the generated data the same as the input data.

Standard autoencoders, continued more

Some notes about autoencoders.

- The entire network is trained as a whole.
- The loss function is usually cross entropy or mean-squared error; it's known as the "reconstruction loss".
- Unfortunately, the encoded data is lossy, meaning that information is lost (thrown away).
- This is because the input data is much larger in size than what comes out of the encoder.
- But the encoder keeps enough of the important information that the essentials of the image can be rebuilt by the decoder.

Together these two networks form an autoencoder.

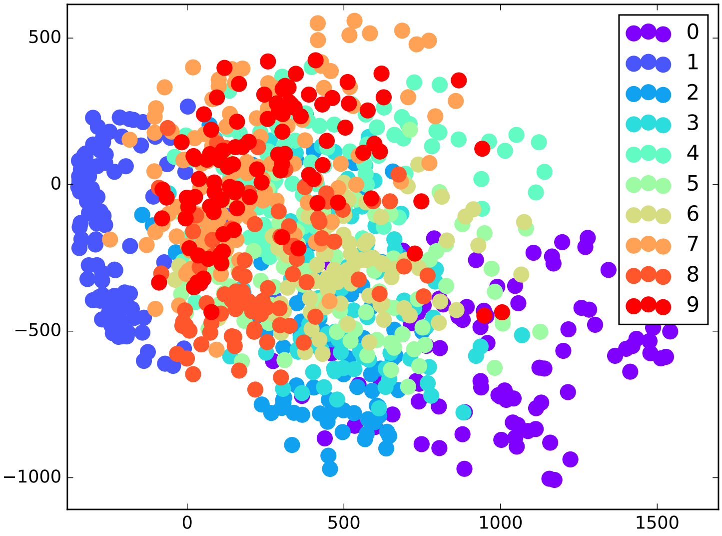
Problems with standard autoencoders

But standard autoencoders have some shortcomings.

- Standard autoencoders can be trained to generate encoded representations of the input data.
- But other than a few niche applications, they aren't that commonly used.
- Why? Because the encodings which are generated by the encoder, which are 1D vectors, are often not continuous between one input and another.
- Why is that a problem? Because we don't want to just replicate the original data. We want to be able to create variations on the original data, continuously.
- If the encoded output is clustered, meaning that similar inputs end up getting encoded "close together", and the clusters are not close together, the decoder is not going to know what to do if I give it, as input, an encoding vector which is a combination of two inputs.

Creating new variations on the original data set is not possible in this scenario. The decoder just won't know what to do.

Problems with standard autoencoders, continued



Variational autoencoders

Variational autoencoders are similar to standard autoencoders, but with a twist.

- By design, the space into which the encoder encodes is continuous.
- This allows the decoder to be able to generate output which combines different features of the data from the original data set.
- We can to this by feeding the decoder combinations of the encodings of different inputs.
- How is the encoder forced to encode continuously? By not outputting just a single encoding vector, but rather by outputing two vectors:
  - a vector of mean values
  - a vector of standard deviations
- We now sample from the distributions represented by these means and standard deviations to create an input vector, which is fed to the decoder.

This means that, even if we use the same input to the encoder, the decoder will see a different input due to the random sampling.

Variational autoencoders, continued

Variational autoencoders add randomness to the input of the decoder.

- For a given input to the encoder, the decoder ends up sampling a range of values which get associated with the decoder's output.
- This local variation of the input to the decoder removes some of the discontinuity in the decoder's input space, at least for a class of inputs.
- But this still doesn't get us continuity between different classes of input.
- We need the inputs to all be as close as possible to each other, while still being distinguishable.

How is that accomplished?

Variational autoencoders, continued more

To force this, we need to revisit a topic we touched on in the GANs class: KL divergence.

- KL divergence (Kullback-Leibler divergence) is a measure of the distance between two probability distributions.
- To minimize this distance, in this case, means to optimize the probability distribution parameters which are output by the encoder, to closely resemble that of the target distribution.
- To do this we need to calculate a new cost function.

This will require some math to explain properly.

KL divergence

Let:

- X be our target data (our data set)
- z is our latent variable (the output of the encoder).
- $P(X)$ is the probability distribution of the data set.
- $P(X|z)$ the distribution of the data given the latent variable.

The goal is to get $P(X) = \int P(X|z)P(z)$, so that we can model the data set. To this end, we would like to determine $P(z|X)$, so that we can get $P(z)$. Using a VAE, we infer $P(z|X)$ using a Bayesian technique called "Variational Inference".

If the decoder's distribution is $Q(X|z)$, then the KL divergence is given by:

$$D_{KL}\left[Q(z|X)||P(z|X)\right] = \sum_z Q(z|X) \log\left(\frac{Q(z|X)}{P(z|X)}\right)$$

KL divergence, continued

$$\begin{aligned}
D_{KL}\left[Q(z|X)||P(z|X)\right] &= \sum_z Q(z|X)\log\left(\frac{Q(z|X)}{P(z|X)}\right)\\
&= E\left[\log\left(\frac{Q(z|X)}{P(z|X)}\right)\right]\\
&= E\left[\log\left(Q(z|X)\right)-\log\left(P(z|X)\right)\right]\\
&= E\left[\log\left(Q(z|X)\right)-\log\left(\frac{P(X|z)P(z)}{P(X)}\right)\right]\\
&= E\left[\log\left(Q(z|X)\right)-\log\left(P(X|z)\right)\right.\\
&\qquad\left.-\log\left(P(z)\right)+\log\left(P(X)\right)\right]\\
&= E\left[\log\left(Q(z|X)\right)-\log\left(P(X|z)\right)\right.\\
&\qquad\left.-\log\left(P(z)\right)\right]+\log\left(P(X)\right)
\end{aligned}$$

Where we have used Bayes' rule in the third-last line, and $P(X)$ has been taken out of the expectation value.

KL divergence, continued more

$$
\begin{aligned}
D_{KL}\left[Q(z|X)||P(z|X)\right] &= E\left[\log\left(Q(z|X)\right) - \log\left(P(X|z)\right)\right. \\
&\qquad \left. - \log\left(P(z)\right)\right] + \log\left(P(X)\right)
\end{aligned}
$$

We now rearrange some terms.

$$
\begin{aligned}
\log\left(P(X)\right) - D_{KL}\left[Q(z|X)||P(z|X)\right] &= E\left[\log\left(P(X|z)\right) - \log\left(Q(z|X)\right)\right. \\
&\qquad \left. + \log\left(P(z)\right)\right] \\
&= E\left[\log\left(P(X|z)\right)\right] - E\left[\log\left(Q(z|X)\right)\right. \\
&\qquad \left. - \log\left(P(z)\right)\right] \\
&= E\left[\log\left(P(X|z)\right)\right] - D_{KL}\left[Q(z|X)||P(z)\right]
\end{aligned}
$$

And there it is, the VAE loss function!

KL divergence, continued even more

$$\log\left(P(X)\right) - D_{KL}\left[Q(z|X)||P(z|X)\right] = E\left[\log\left(P(X|z)\right)\right] - D_{KL}\left[Q(z|X)||P(z)\right]$$

How do we interpret this?

- $\log\left(P(X)\right)$ represents the data set.
- Which has some error associated with it, $D_{KL}\left[Q(z|X)||P(z|X)\right]$.

We should maximize $E\left[\log\left(P(X|z)\right)\right]$ and minimize the difference between $Q(z|X)$ and $P(z)$.

- We can approximate $P(X|z)$ using the decoder half of the network.
- Our regular loss function will maximize that already.

KL divergence, implementation

How do minimize the difference between $Q(z|X)$ and $P(z)$?

- $P(z)$ is the latent variable distribution. We can model this using $N(0,1)$ (Gaussian with $\mu = 0$ and $\sigma = 1$).
- We model $Q(z|X)$ as a Gaussian with parameters $\mu(X)$ and $\Sigma(X)$.
- Then the KL divergence between those two pieces can be calculated exactly.

$$D_{KL}\left[N(\mu(X), \Sigma(X)) || N(0,1)\right] = \frac{1}{2} \sum_k \left(\Sigma(X) + \mu^2(X) - 1 - \log\left(\Sigma(X)\right)\right)$$

Where k is the number of dimensions in z, and $\Sigma(X)$ is the diagonal of the usual covariance matrix. We use this, combined with the regular loss function, to train the network.

The algorithm

So what is the approach? Create a network with the following architecture.

- Build an encoder.
- As we mentioned earlier, the encoder will output two vectors, a vector of mean values ($\boldsymbol{\mu}(\boldsymbol{X})$), and standard deviations ($\boldsymbol{\Sigma}(\boldsymbol{X})$).
- To create the latent variable, $\boldsymbol{z}$, to input into the decoder, we sample randomly from the Gaussians with means of $\boldsymbol{\mu}(\boldsymbol{X})$ and standard deviations $\boldsymbol{\Sigma}(\boldsymbol{X})$.
- This $\boldsymbol{z}$ is fed into the decoder, which generates an output.
- To train, we use two loss functions:
  - $\boldsymbol{L}_{\text{reconstruction}} = \frac{1}{2}\left(\boldsymbol{X}_{\text{in}} - \boldsymbol{X}_{\text{out}}\right)^2$
  - $\boldsymbol{L}_{\text{KL}} = \frac{1}{2}\sum_k \left(\boldsymbol{\Sigma}(\boldsymbol{X}) + \boldsymbol{\mu}^2(\boldsymbol{X}) - 1 - \log\left(\boldsymbol{\Sigma}(\boldsymbol{X})\right)\right)$
- Train the network in the usual way.

But, as you might expect, there is a slight problem with what's been described here.

Backpropagation over a sampled distribution

How do you do backpropagation through a function which is sampled?

- Sampling doesn't have a gradient.
- There is a way around this, called the "reparameterization" trick.

Suppose that x is sampled from $N(\mu(X), \Sigma(X))$, and suppose that we standardize it, so that $\mu = 0$ and $\Sigma = 1$.

$$x_{\text{std}}^2 = (x - \mu)\,\Sigma^{-1}\,(x - \mu)^T$$

This leaves $x = \mu + \Sigma^{\frac{1}{2}} x_{\text{std}}$. We do the same to sample our latent variable:

$$z = \mu(X) + \Sigma^{\frac{1}{2}}(X)\epsilon$$

where ϵ is sampled from $N(0, 1)$. This moves the non-differentiable part of the operation out of the network, so that the network can still be trained in the usual way.

VAE example

Let's do an example. We'll use our old friend, MNIST.

- We will use convolutional and transpose convolutional layers.
- We will have a latent space size of 100.
- We will use mean squared error as our reconstruction cost function, combined with the KL divergence.
- Rather than have the encoder calculate the standard deviation of the mean values it outputs, we will instead have it calculate the log of the standard deviations (it is more numerically stable to calculate an exponential than a logarithm).
- We will build the encoder and decoder as two separate networks, so that they can be saved, and later used, separately. They will be trained together, obviously.
- We normalize the input data, and then use sigmoid on the output of the decoder.

VAE example, the code

```
# mnist_vae.py
import keras.models as km
import keras.layers as kl
import keras.backend as K
from keras.datasets import mnist

def sampling(args):
  z_mean, z_log_var = args
  batch = K.shape(z_mean)[0]
  dim = K.int_shape(z_mean)[1]
  eps = K.random_normal(shape = (batch, dim))
  return z_mean + K.exp(0.5 * z_log_var) * eps

(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train = x_train.reshape(60000, 28, 28, 1) / 255.
x_test = x_test.reshape(10000, 28, 28, 1) / 255.
```

```
# mnist_vae.py, continued
input_img = km.Input(shape = (28, 28, 1))
x = kl.Conv2D(16, kernel_size = (3, 3),
  activation = "relu",
  padding = "same")(input_img)
x = kl.MaxPooling2D(pool_size = (2, 2),
  stride = 2)(x)
x = kl.Conv2D(32, kernel_size = (3, 3),
  activation = "relu", padding = "same")(x)
x = kl.MaxPooling2D(pool_size = (2, 2),
  stride = 2)(x)

x = kl.Flatten()(x)

z_mean = kl.Dense(latent_dim,
  activation = "linear")(x)
z_log_var = kl.Dense(latent_dim,
  activation = "linear")(x)
```

VAE example, the code, continued

```
# mnist_vae.py, continued

z = kl.Lambda(sampling)([z_mean, z_log_var])

def vae_loss(input_img, output_img):

  kl_loss = 1 + z_log_var
    - K.square(z_mean) - K.exp(z_log_var)
  kl_loss = -0.5 * K.sum(kl_loss, axis = -1)

  r_loss = 784 * K.mean(K.square(input_img -
    output_img)

  return r_loss + kl_loss
```

```
# mnist_vae.py, continued

decoder_input = km.Input(shape = (latent_dim,))

x = kl.Dense(7 * 7 * 32,
  activation = "relu")(decoder_input)
x = kl.Reshape((7, 7, 32))(x)

x = kl.Conv2DTranspose(32, kernel_size = (3, 3),
  activation = "relu", padding = "same")(x)
x = kl.UpSampling2D(size = (2, 2))(x)
x = kl.Conv2DTranspose(16, kernel_size = (3, 3),
  activation = "relu", padding = "same")(x)
x = kl.UpSampling2D(size = (2, 2))(x)
decoded = kl.Conv2DTranspose(1, kernel_size = (3, 3),
  activation = "sigmoid", padding = "same")(x)
```

VAE example, the code, continued more

```
# mnist_vae.py, continued

encoder = km.Model(inputs = input_img,
  outputs = [z_mean, z_log_var, z])

decoder = km.Model(inputs = decoder_input,
  outputs = decoded)

output_img = decoder(encoder(input_img)[2])
vae = km.Model(inputs = input_img,
  outputs = output_img)

vae.compile(loss = vae_loss,
  optimizer = "rmsprop",
  metrics = ["accuracy"])
```

```
# mnist_vae.py, continued

fit = vae.fit(x_train, x_train, epochs = 200,
  batch_size = 64, verbose = 2)

score = vae.evaluate(x_test, x_test)
print "score is", score

# Save the final models.
vae.save('mnist_vae.h5')
encoder.save('mnist_encoder.h5')
decoder.save('mnist_decoder.h5')
```

VAE example, running the code

```
ejspence@mycomp ~> python mnist_vae.py
Using Theano backend.
13s - loss:  43.2175 - acc:  0.7972
Epoch 2/200
8s - loss:  35.0138 - acc:  0.8034
Epoch 3/200
9s - loss:  32.4862 - acc:  0.8058
⋮
Epoch 198/200
8s - loss:  28.3887 - acc:  0.8092
Epoch 199/200
8s - loss:  28.3741 - acc:  0.8093
Epoch 200/200
9s - loss:  28.3841 - acc:  0.8093
9600/10000 [============================>..]  - ETA: 0s
score is [28.383084182739257, 0.8069316199302673]
ejspence@mycomp ~>
```

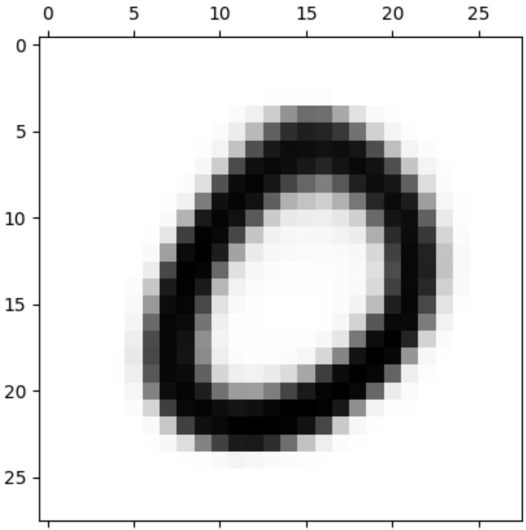
What can we do with it?

Because we know the labels of the input data (the digits 0-9), we can play around with encoder and decoder to interesting effect. For example:

- Using the encoder, we can average over the encoder-output means of all the inputs of the same label (all the sixes, for example).
- We then have a set of 10, latent vectors which represent each digit.
- We can then linearly interpolate from one latent vector to another, and run those interpolations through the decoder.
- If the decoder is any good, as we transition from one latent vector to the next we should smoothly transition from one digit to the next.

In this way, we can manipulate data or images of different types, blending, subtracting and modifying images in a predictable way.

Manipulating digits



Linky goodness

Variational autoencoders:

- https://wiseodd.github.io/techblog/2016/12/10/variational-autoencoder
- https://blog.keras.io/building-autoencoders-in-keras.html
- http://kvfrans.com/variational-autoencoders-explained
- https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf
- https://jaan.io/what-is-variational-autoencoder-vae-tutorial