Analyzing 50k fonts using deep neural networks

For some reason I decided one night I wanted to get a bunch of fonts. A lot of them. An hour later I had a bunch of scrapy scripts pulling down fonts and a few days later I had more than 50k fonts on my computer.

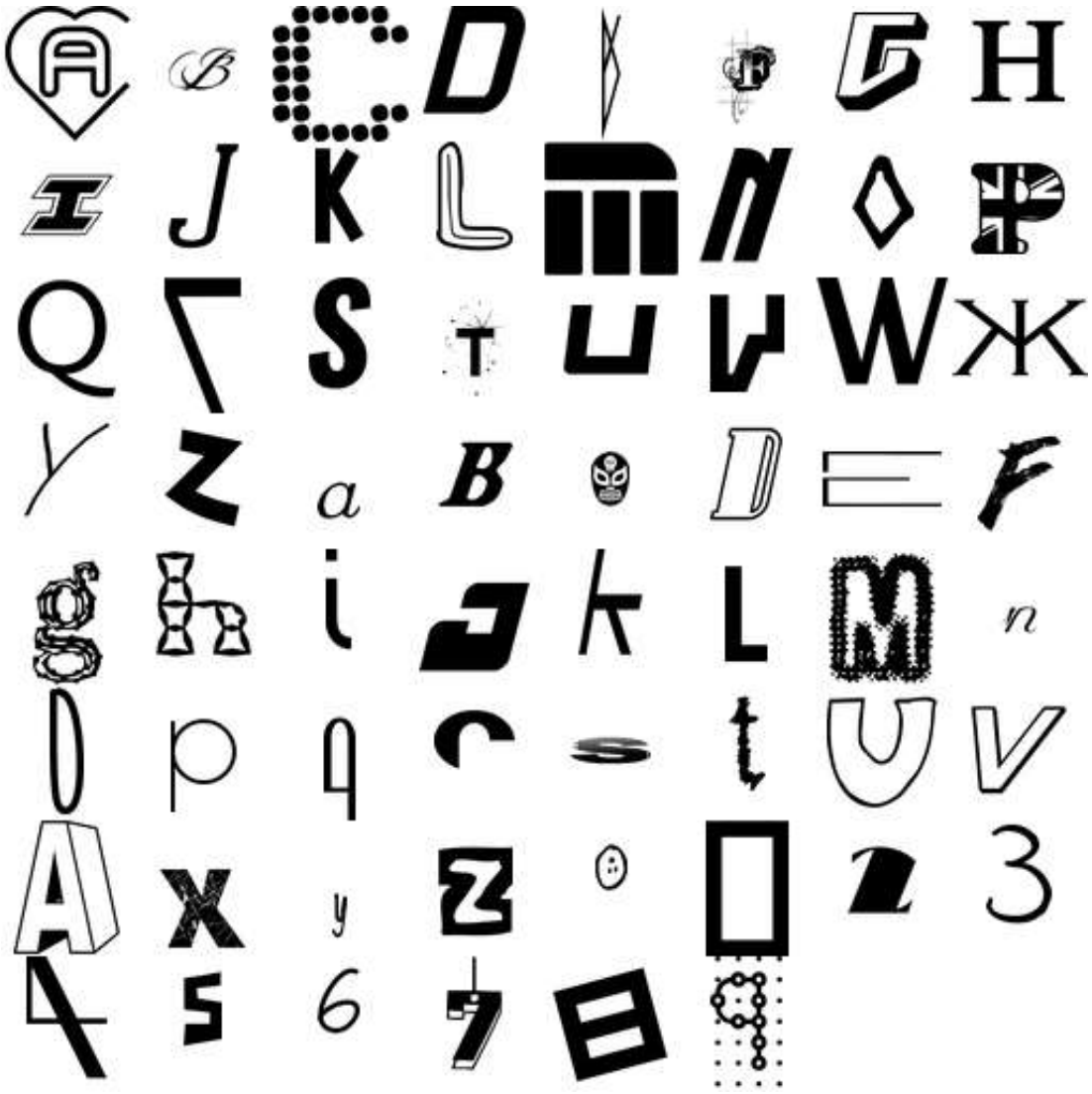
I then decided to convert it to bitmaps. It turns out this is a bit trickier than it might seem like. You need to crop in such a way that each character of a font is vertically aligned, and scale everything to fit the bitmap. I started with 512 * 512 bitmaps of all character. For every font you find the max y and min y of the bounding box, and the same thing for each individual letter. After some more number juggling I was able to scale all characters down to 64 * 64.

The result is a tensor of size 56443 * 62 * 64 * 64. Exercise for the reader: where does the number 62 come from? I stored it as a tiny little (13GB) HDF5 file that you can download here: fonts.hdf5.

If you take the average of all fonts, here's what you get:

Hopefully by now it should be clear where the number 62 came from. The median is a lot less blurry than the average:

Both mean and median are well-formed and legible! However individual fonts are all over the place:



I guess I practically begged for it, stealing fonts from various sketchy places all over the web. In particular most of the fonts don't even have lower case versions of the letters. A minority of fonts miss certain characters and will just output rectangles instead. And look at the ridiculous Power Ranger figure for the lower case "c"!

Training a neural network

Now, let's train a neural network that generates characters! Specifically what I wanted to do is to create a "font vector" that is a vector in latent space that "defines" a certain font. That way we embed all fonts in a space where similar fonts have similar vectors.

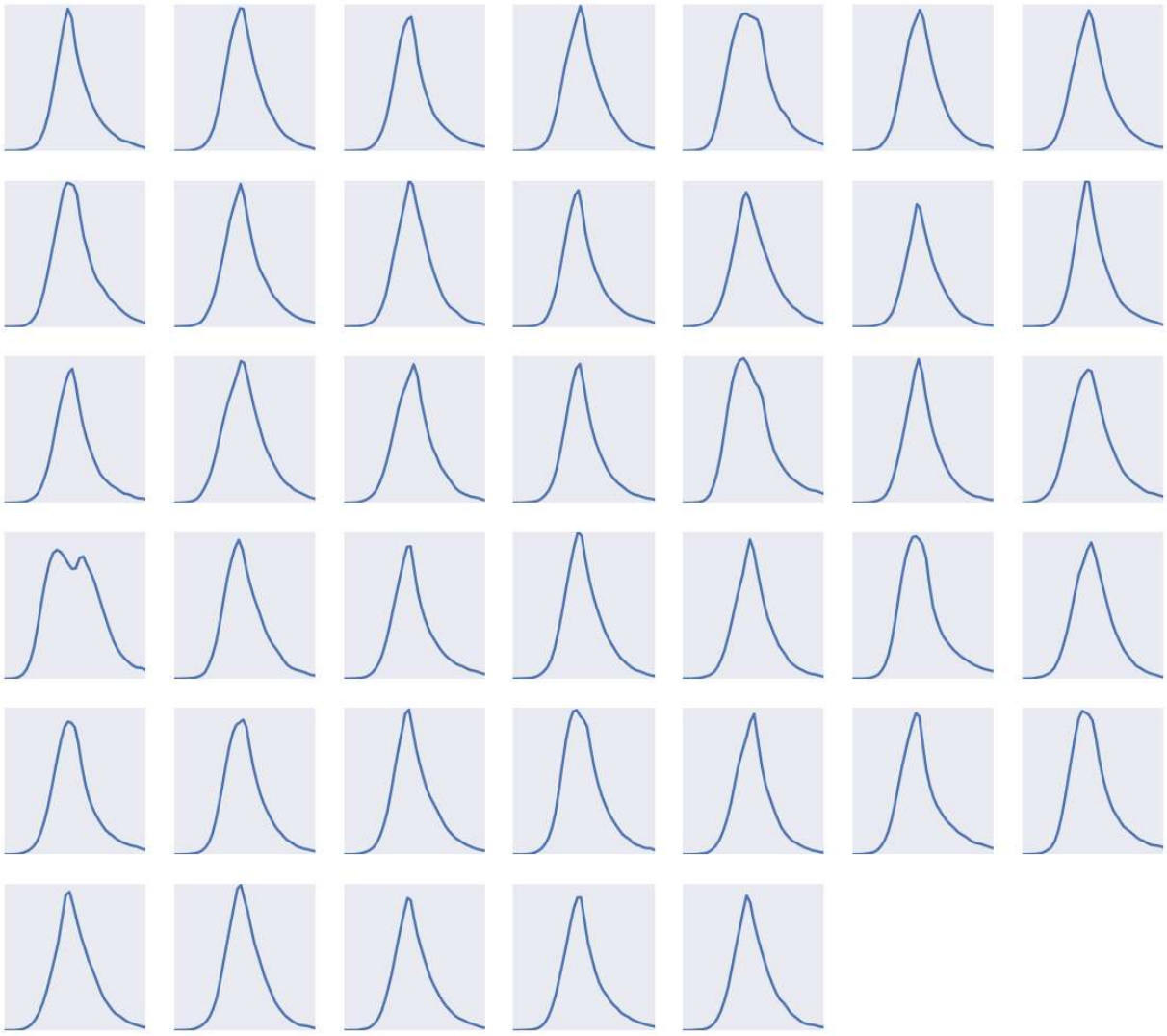
I built a simple neural network using Lasagne/Theano — check out the code here. It took an insane amount of time to converge, probably because there's so much data and parameters. After weeks of running, the model converges to something that looks decent.

Some notes on the model

- 4 hidden layers of fully connected layers of width 1024.
- The final layer is a 4096 layer (64 * 64) with sigmoid nonlinearity so that the output is between 0 (white) and 1 (black).
- L1 loss between predictions and target. This works much better than L2 which generates very "gray" images — you can see qualitatively in the pictures above.
- Pretty strong L2 regularization of all parameters.
- Leaky rectified units (alpha=0.01) of nonlinearity on each layer.
- The first layer is 102D — each font is a 40D vector joined with a 62D binary one-hot vector of what is the character.
- Learning rate is 1.0 which is shockingly high — seemed to work well. Decrease by 3x when no improvements on the 10% test set is achieved in any epoch.
- Minibatch size is 512 — seemed like larger minibatches gave faster convergence for some weird reason.
- No dropout, didn't seem to help. I did add some moderate Gaussian noise (of sigma 0.03) to the font vector and qualitatively it seemed to help a bit.
- Very simple data augmentation by blurring the input randomly with sigma sampled from [0, 1]. My theory was that this would help fitting charactersÂ that have thin lines.

All of the code is available in the erikbern/deep-fonts repo on Github.

After convergence, we end up having a nice 40D embedding of all 50k fonts. LooksÂ like it ends up being roughly a multivariate normal — here's the distribution of each of the 40 dimensions:



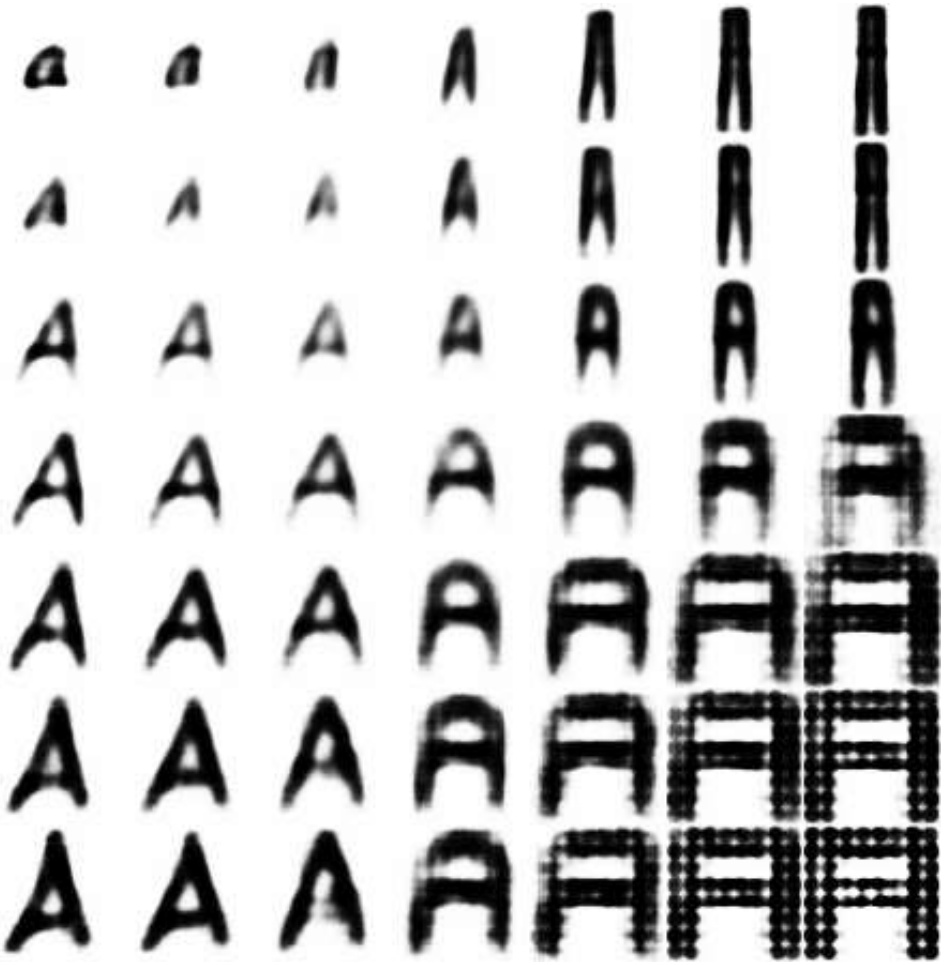
Playing around with the model

To start with, let's recreate real font characters with characters generated from the network. Let's plot the real character together with the model outputs. For each pair below, the real character is on the left, the model output on the right.

These are all characters drawn from the test set, so the network hasn't seen any of them during training. All we're telling the network is (a) what font it is (b) what character it is. The model has seen other characters of the same font during training, so what it does is to infer from those training examples to the unseen test examples.

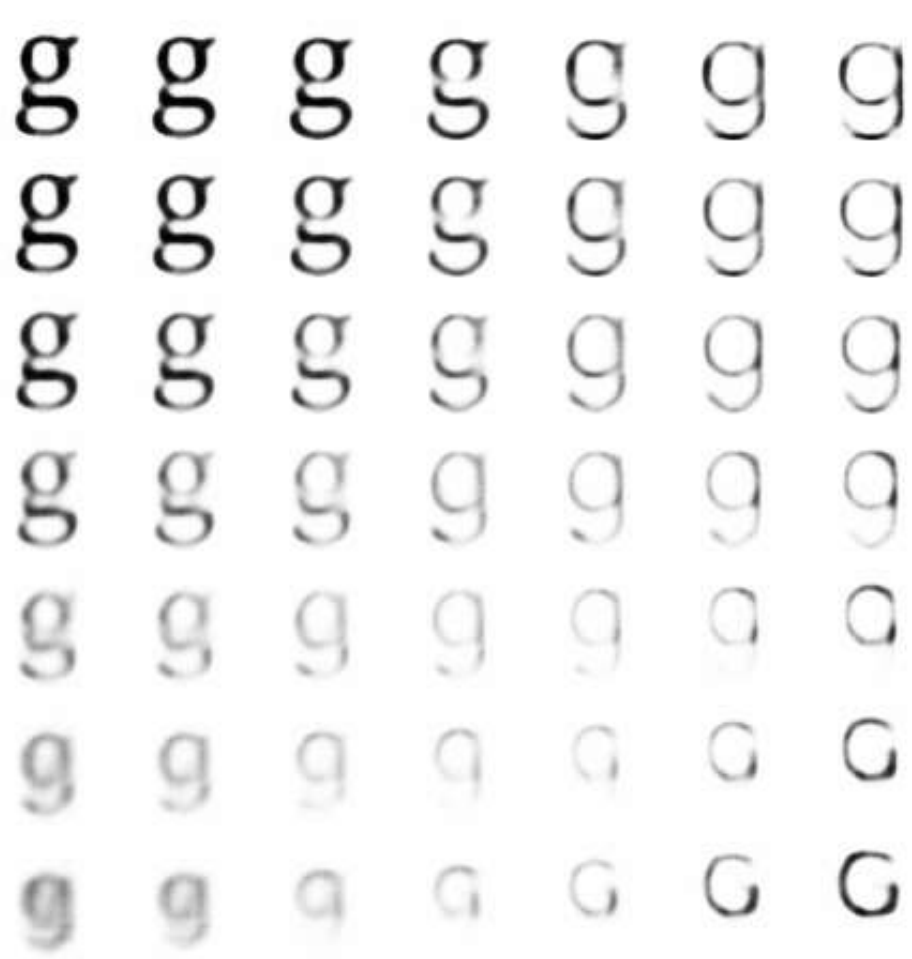
The network does a decent job at most of the characters, but gives up on some of the more difficult ones. For instance, characters with thin black lines are very hard to predict for the model, since if it renders the line just a few pixel to the side, that's twice the loss of just rendering whitespace.

We can also interpolate between different fonts in continuous space. Since every font is a vector, we can create arbitrary font vectors and generate output from it. Let's sample four fonts and put them in the corners of a square, then interpolate between them!



Certain characters have multiple forms that we can interpolate between, eg. lowercase g:

Erik Bernhardsson



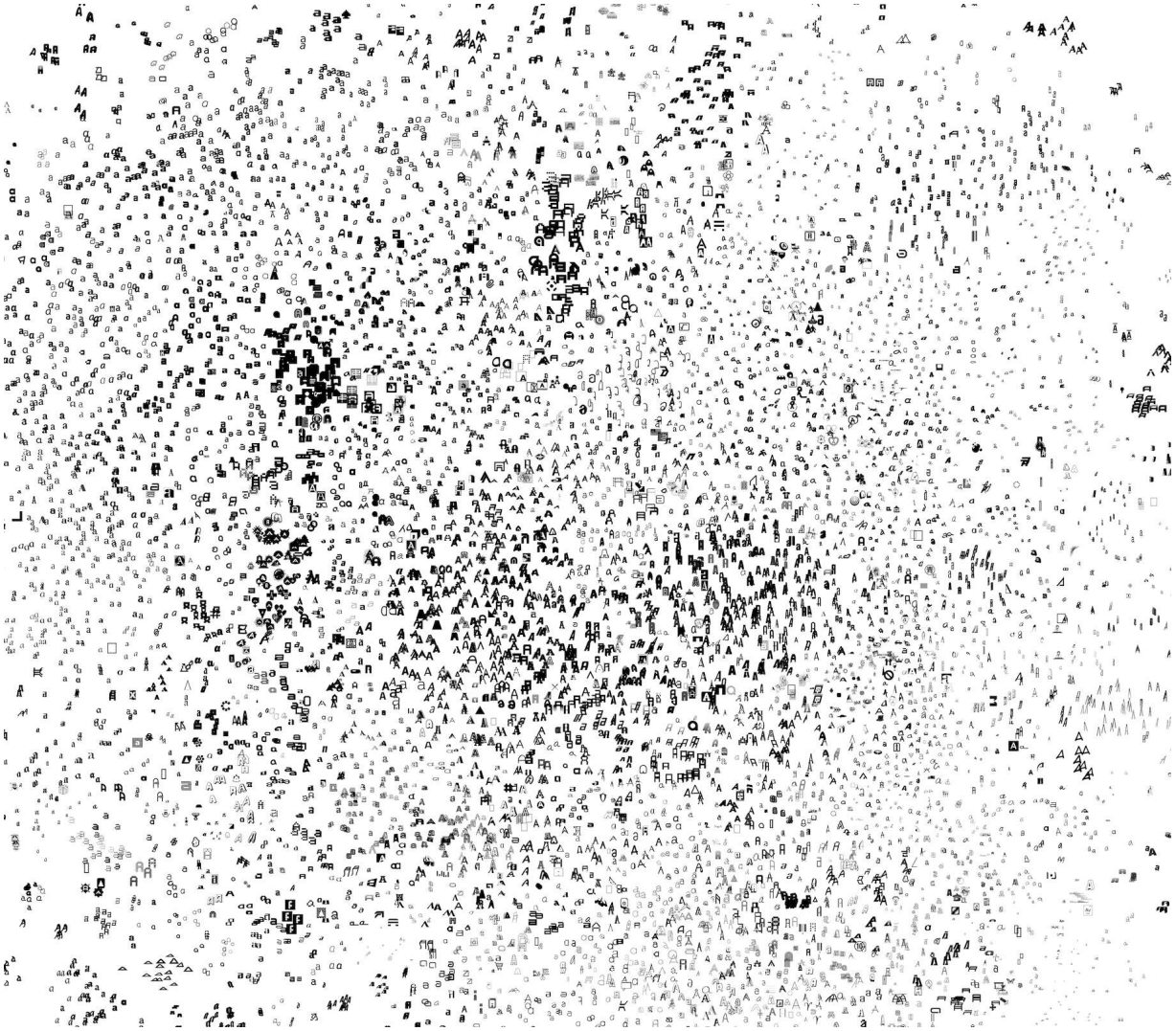
We can also pick a font vector and generate new fonts from random perturbations:

(btw internet god — please forgive me for wasting bandwidth on all the animated gifs in this blog post!)

We can also generate completely new fonts. If we model the distribution of font vectors as a multivariate normal, we can sample random vectors from it and look at the fonts they generate. I'm interpolating between a few of thoseÂ vectors in the picture below:

An interesting thing here is that the model has learned that many fonts use upper case characters for the lower case range — the network interpolates between Q and q seamlessly. Here's an example of the network interpolating very slowly between two fonts where this is the main difference:

Another cool thing we can do since we have all fonts in a continuous space is to run t-SNE on them and embed all fonts into the 2D plane. Here's a small excerpt of such an embedding:



Final remarks

There are many other fun things you can do. It's clear that there's some room for improvement here. In particular, if I had more time, I would definitely explore generative adversarial models, which seems better at generating pictures. Another few things should be relatively easy to implement, such as batch normalization and parametric leaky rectifications. And finally the network architecture itself could probably benefit from doing deconvolutions instead of fully connected layers

Feel free to download the data and play around with it if you're interested!

January 21, 2016

erikbern