A Musing Upon Variational Autoencoders

Joseph Suarez

Abstract

We conducted several hundred experiments in the process of implementing a variational autoencoder. Some of these yielded significant performance gains, training speedups, or increases in stability. Using the same parameters as in the original paper by Kingma et al, we were able to come close to their reported test error, but there remained a significant gap. We expect this is due to some unreported optimizations in the original paper. As such, we performed some of our own optimizations and achieve more comparable results. We present many of these optimizations below.

Introduction

While the original paper details the VAE algorithm in extreme mathematical detail, I would like to step back for a moment and present my own understanding of the algorithm, which takes the form of a small modification to a standard neural network. To begin, consider a standard neural network. For the purpose of example, consider a fairly standard 784-500-500-500-784 architecture for learning the identity on MNIST. This is a large network that will quickly overfit the fairly small MNIST train set.

Suppose we want to ensure the network does not overfit by explicitly requiring the network to learn a small feature representation of the data. We can do this by decreasing the size of, say, the middle layer, such that we obtain a 784-500-20-500-784 architecture. This dramatically lessens the overfitting model, as the network must learn to reconstruct digits from a representation in only 20 dimension. This is a standard autoencoder.

Now we take the idea of latent representation one step further. We will not only require that the network learn a latent representation, but that that representation be specified as a probability distribution; digits must be reconstructed from only a sample of the distribution over latents space.

Of course, we cannot merely throw away the first half of the network (encoder) and expect the second half (decoder) to learn anything useful by sampling, say, a standard normal distribution. Instead, we will use the outputs of the encoder to specify the means and (log) standard deviations of the distribution. In my implementation, I achieve this as follows. First send the input digit through the encoder to generate a latent vector with an even number of dimensions. Split this vector in half. Use the first half as means and the second half as log standard deviations for a multivariate gaussian. Draw a single sample (with dimensionality equal to half that of the encode vector) and use this as input to the decoder. Output a prediction as normal.

Experiments

We first use the original parameters of the paper, with one small exception: our encoder has 1000 units instead of 500. This is to compensate for the fact that we generate the encode vector using a single network, then split it in half. In the original paper, the encode vector is generated with two separate networks. The approaches should be equivalent, accounting for the factor of 2 in the hidden dimension. Using a 784-1000-3-500-784 architecture with Adagrad, learning rate 0.01, minibatch size 100, we achieve summed KL and cross entropy train error of 145 and test error of 147.3 after training on 12 million examples. For comparison, the original paper reports error of approximately 136 for both train and test on this task.

State of the art implementations almost always involve a large number of tweaks. We make a few of our own. First, we blame the optimizer and switch to Adam with learning rate 0.001. This reduces train and test errors to 138.5 and 145.3 respectively.

Second, we blame the RELU activations and switch to ELUs. This updates train/test errors to and 141.7 and 144.1, respectively. Interestingly, the train error increased while the test error decreased, which suggests ELUs help combat overfitting.

We then tried a larger architecture: 784-2000-6-1000-784. This resulted in decreased train/test errors of 138 and 142.

With the difference in error seen above, even with my optimizations, I began wondering if I had a minor bug in my implementation. Or perhaps this error difference is due to some “special sauce” in the original paper’s implementation and not all that significant. We test with z=20 and obtain train/test errors of 94.25 and 96.1, which is around 6 points lower than reported in the paper for ~15 million training examples. Interestingly, our improvement on the original paper’s error for z=20 is approximately the same as their advantage on our implementation for z=3. And as a sanity check, our reconstructions look quite good.

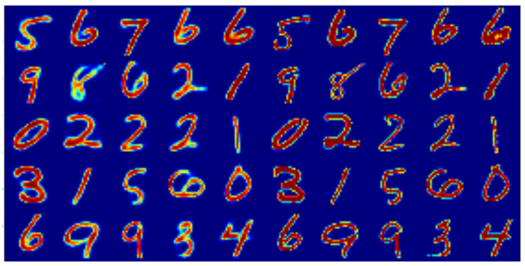


Figure 1. Sample reconstructions for z=40 after training on 14 million examples. The right hand side shows the targets.

When we reduce z to 2, as required to render manifolds, error increases markedly. After 14 million training examples, we obtain train and test errors of 159.5 and 160.5 respectively. However, the error seemed highly unstable, so we reduced the learning rate to 0.0001 and continued training. Note that early reconstruction results are surprisingly good for such high error.

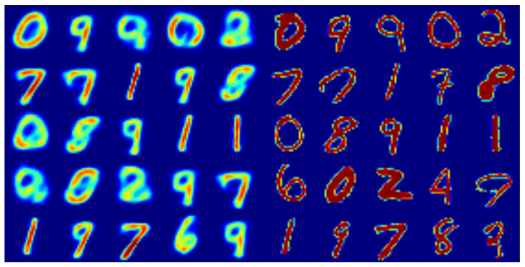


Figure 2: Sample reconstruction with z=2 after training on 14 million examples.

We train for another 10 million examples and obtain train/test errors of 154.6 and 161.4. We are beginning to overfit, thus we stop.

We present two images of the manifolds produced by sampling latent sapce. The first, as in the original paper, was obtained by transforming linear spacing with the inverse normal CDF. The second, which we prefer, was obtained via a linear distribution between -5 and 5. The bottom half of our manifold is severely distorted—it appears the algorithm failed to utilize this portion of the space. However, the top half of the manifold does indeed represent a reasonable representation of the digits in latent space. We present both, for comparision.

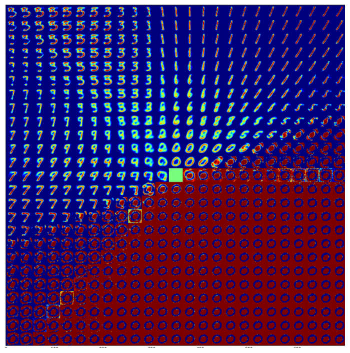
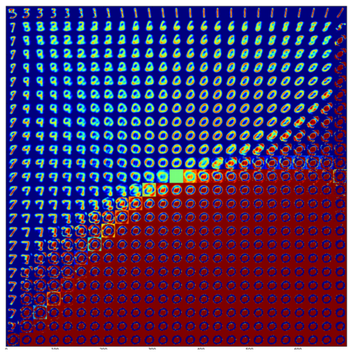


Figure 3. Manifold created by sampling latent space in 2 dimensions with linear sample spacing.


0
100
200
300
400
500
600