Image Generation Using an Autoencoder
Math 156 Group Project

Amaael Antonini, Krystian Galik, Salman Shah, and Zixin Zhang

June 13, 2018

1 Introduction

Ever since their invention, neural networks have been widely used to solve many kinds of problems. An autoencoder is a specific type of neural networks that is designed to encode a given class of information into a significantly lower dimensional space and decode the vectors in that space to reproduce the original image. In this project, we investigate how well an autoencoder can encode graphical information and discover the optimal architecture for our problem through experimentation. Then we further our investigation by asking the following question: "Given a dataset of images of different categories, can we build a machine learning model that can be trained to generate new images pertaining to a certain category?"

2 Model

2.1 Dataset

We use the MNIST dataset because it is the perfect dataset for our problem. This is a dataset with 60,000 labeled 28x28 pixel images of handwritten integers from 0 to 9. These images are grayscale, allowing us to represent each pixel as its intensity value from 0 to 1, and to represent each image as a vector with 784 entries containing all the intensity values.

2.2 Approach

In order to determine the best model for our problem, we sequentially narrow our choices by a number of variables. First, we determine the architecture of the autoencoder by experimenting with the number of hidden layers. Next, we experiment with loss functions to determine the loss function that produces the least value for validation loss. Finally, we experiment with probability distributions that allow us to generate new examples of images with the maximum amount of variance while maintaining good accuracy.

2.3 Architecture

2.3.1 Dimension of the Layers

The input layer of the autoencoder must have 784 nodes since that is the size of our input vectors. The output layer must also have 784 nodes. This is because we measure the performance accuracy by simply comparing the output image to the input image.

The complexity of an autoencoder is determined by its number of hidden layers. We define the encoding dimension to be 32 (ie: the network reduces the dimensionality of the image and stores it in a 32 dimensional vector). We built three different variations of our autoencoder: the first with a single hidden layer, the second with 5 hidden layers, and the third with 9 hidden layers. The number of nodes in each layer is given respectively below:

- 1 hidden layer: 32
- 5 hidden layers: 512, 128, 32, 128, 512
- 9 hidden layers: 512, 256, 128, 64, 32, 64, 128, 256, 512

Notice that the encoder portion of the network steadily decrements the number of nodes towards the encoding dimension while the decoder portion of the network steadily increments the number of nodes back to the dimension of our input image. This choice of reduced dimensionality for the encoder takes into account the following two considerations: first, using a consistently dimension-reducing encoder forces the neural network to avoid learning the identity function at any stage of training; second, the autoencoder can reduce the input data to a lower-dimensional space where implementing a probability distribution is computationally more feasible. The symmetric structure is implemented for convenience.

2.3.2 Number of Training Epochs and Number of Layers

After testing with three different architectures, we discovered that in the range of 100 epochs, the 5 layer network had the best performance. However, we noticed that the 9 layer network continued to improve when we continued training the networks for higher number of epochs, whereas the 5 layer and the 1 layer networks stopped showing improvements. For this reason, we chose the 9 layer network and trained it for 1000 epochs to use for our model. See Figure.1 and Figure.2 for the resulting outputs.

2.3.3 Loss Function

We experimented with 3 different loss functions: (1) mean squared error, (2) binary cross entropy, and (3) Euclidean distance loss. Recall that these loss functions are given by:

$$E_1(X) = \frac{1}{d}\sum_{i=1}^{d}(F(X)_i - X_i)^2 \tag{1}$$

$$E_2(X) = -\sum_{i=1}^{d} X_i \log(F(X)_i) - (1 - X_i)\log(1 - F(X)_i) \tag{2}$$

$$E_3(X) = \|F(X) - X\|^2 \tag{3}$$

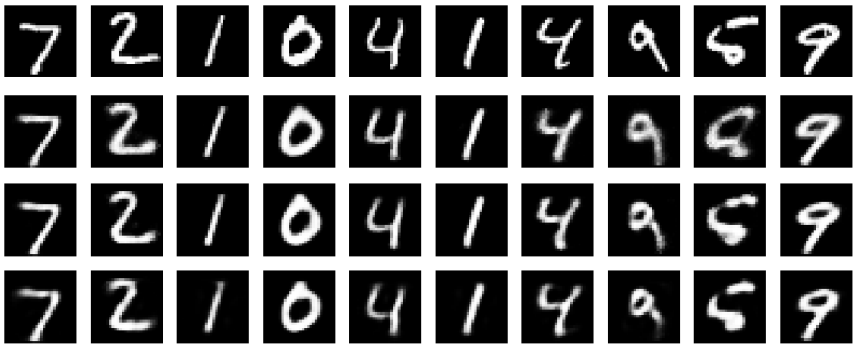

Figure 1: Autoencoder results after 100 epochs. Original (1st row), 9 layer (2nd row), 5 layer (3rd row), 1 layer (last row)

Figure 2: Autoencoder results for 9 layer network after 1000 epochs. Input (top), output (bottom)

---

where $X \in \mathbb{R}^d$ is the input, F is the autoencoder, and $d = 784$ is the input dimension

Through experimentation, we discovered that out of these three loss functions, binary cross entropy produced the least error values over hundreds of epochs, and hence, we chose this loss function for optimizing our model.

2.4 Data Generation

To implement the data generating portion of our model, we decided to fit the encoded space of the autoencoder with a probability distribution for each category. More specifically, for each $x \in \{0, 1, ..., 9\}$, the model includes a probability distribution P_x in the 32-dimensional encoded space and samples a point from that distribution. Then, the sampled point is decoded with the decoder trained in section 2.3.

The fundamental assumption is that digits that are visually similar are encoded close together in the encoded space. Making this assumption is convenient because it allows us to use Gaussian distributions to sample points from the encoded space. At this stage, we have the option of using a single Gaussian or a mixture of Gaussians for each digit. We investigate both results in the next section.

2.4.1 Single Gaussian Implementation

Our first attempt at sampling points from the encoded space involves using standard Gaussian distributions. For each $x \in \{0, 1, ..., 9\}$, we define a Gaussian distribution $N_x(\mu_x, \Sigma_x)$ based on

all the encoded data points labeled x in the encoded space, using the sample mean as μ_x and the sample covariance matrix as Σ_x.

Given that our assumption is correct, we expect that this will capture part of the region in the encoded space that represents the digit x. Moreover, due to the fact that the shape of that region may be arbitrary, we scale down the covariance matrix to maintain a high likelihood of choosing a sample closer to the mean.

2.4.2 Gaussian Mixture Implementation

However, restricting our sample to a region close to the sample mean is likely to produce images that are very similar to the mean. Furthermore, it is possible that the shape of each region representing a digit could be quite complex and uncapturable by a single Gaussian. To bypass these issues, we implement a mixture of Gaussians which is based on the following equation:

$$P(x) = \sum_{i=1}^{K} \pi_i N(x|\mu_i, \Sigma_i) \tag{4}$$

where K represents the total number of Gaussians used to fit the data, π_i represents the probability of sampling from the i^th Gaussian, μ_i represents the mean of the i^th Gaussian and Σ_i represents the covariance matrix of the i^th Gaussian.

The GaussianMixture function from sklearn.mixture is used to implement the mixture and the initial conditions are generated through the k-means method.

3 Results

Through experimentation, we discovered that a deeper network generally took longer to train but had better overall performance. The results from decoding newly sampled points from the encoded space verify that our initial fundamental assumption is correct: images of the same digit are indeed stored closer together in the encoded space.

3.1 Results of Single Gaussian Model

The results of the single Gaussian model with scaled-down covariance matrices are shown in the left picture in Figure 3. For each digit, 10 decoded samples are generated. As shown in the figure, the single Gaussian model did not generate a great deal of variations.

The single Gaussian model with full covariance matrices was also tested and the generated results include many inrecognizable digits, which may be due to that there are overlaps among digits with similar features, such as 3 and 8.

3.2 Results of Gaussian Mixtures Model

In order to obtain more variations and capture the regions of the encoded space more accurately, we proceeded to use a mixture of Gaussians. We picked $K = 250$ in Equation 4. The results are shown in the right picture of Figure 3.

As we expected, the Gaussian mixture models produced far more variety in terms of handwriting styles than the single Gaussian model, but the distribution occasionally sampled points from regions

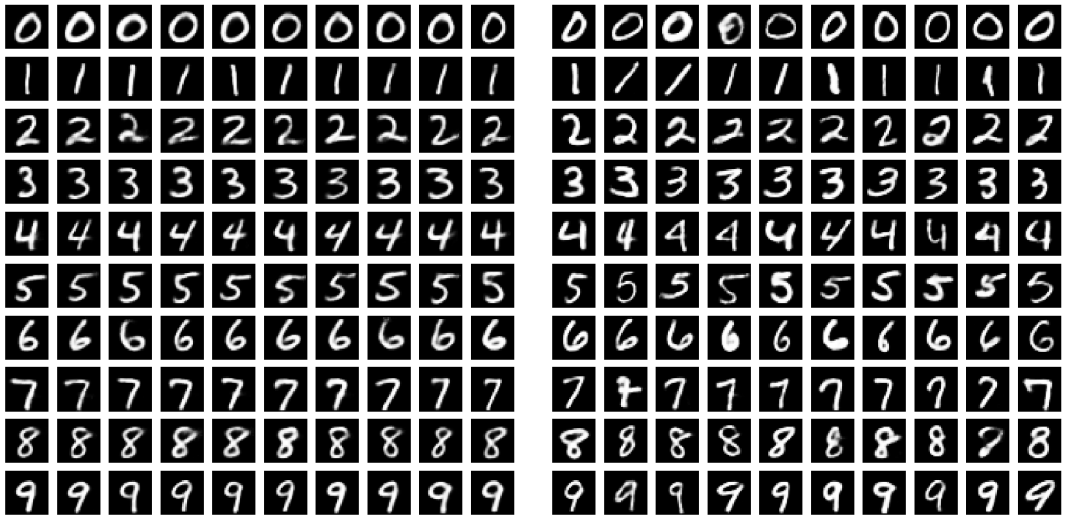

Figure 3: Results from Single Gaussian Model (left) and Gaussian Mixtures Model (right)

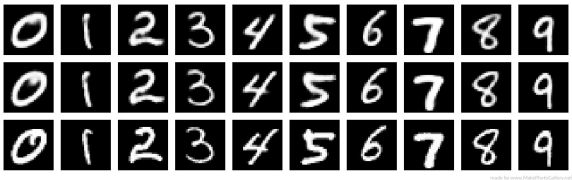

Figure 4: Comparisons for Gaussian mixture generated digits. Generated digit (1st row), Best matching reconstructed image (2nd row), Original image (3rd row)

that were outside the region for the digit we desired. We further expected that increasing the value of K would give us progressively more accurate models in terms of describing the actual distribution of data. We observed that this was not necessarily the case, although for $K = 1$, the model consistently produced poor results.

Furthermore, in order to determine the degree of overfitting, comparisons among the generated digits, the corresponding best matching reconstructed digits of the autoencoder and the corresponding original digits are performed and shown in Figure 4. The method used to determine the best matching reconstructed digits is the binary cross-entropy. We can see that the generated digits are distinguishable from the best matching reconstructed digits and the original digits in Figure 4, which illustrates the originality of the generated digits by the Gaussian mixture method.

Figure 5: Standard basis vectors of the 32-dimensional encoded space

Figure 6: Convex Combination of Sample Means for 1 and 7.

3.3 Interesting Observations

3.3.1 Standard Basis Vectors

Decoded versions of the standard basis vectors are shown in Figure.5. Notice that most of the vectors are vaguely recognizable (by humans) as poorly handwritten numbers. This indicates that the network is not encoding the input information by recognizing patterns (such as loops or edges), but simply by clustering them in space.

3.3.2 Convex Combinations of Sample Means

To observe and analyze the spaces between the labeled clusters in the encoded space, we implemented pairwise convex combinations of sample means of every digit. The resulting image warps from one digit to another in a non-randomized fashion. Figure.6 shows the evolution between digit 1 and 7, which demonstrates that the decoder performs non-linear operations which capture more features than simple linear combinations.

3.3.3 Principal Component Analysis (PCA)

In our last attempt to visualize the encoded space, we projected the space into its first two principal components. The left picture in Figure. 7 shows a plot of the eigenvalues of the sample covariance matrix in the encoded space. Notice that the eigenvalues decrease steadily (i.e. there is no sudden drop off for the values), implying that much of the data is using the available 32 dimensions. Therefore, projecting the data onto two dimensions may greatly misrepresent its actual representation

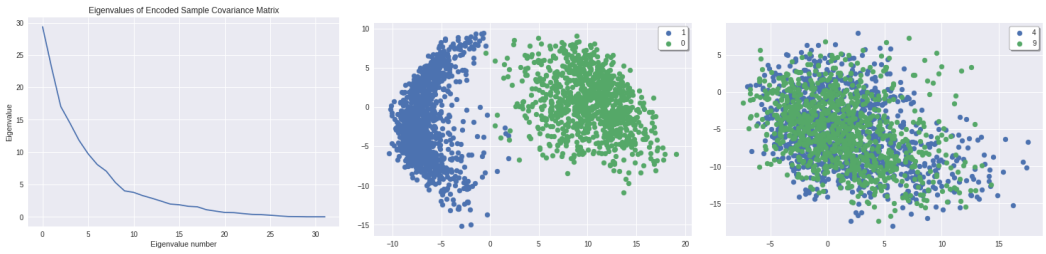

Figure 7: PCA: eigenvalues (left), projection of 1 and 0 (center), projection of 4 and 9 (right)

and exclusivity from other clusters. Nevertheless, comparing the middle plot and and the rightmost plot in Figure. 7, we can see that digits with very different features, such as 0 and 1, are already separated clearly in the projected subspace, while digits with similar features, such as 4 and 9, have many overlaps in the projected subspace.

4 Conclusion

In this project, we trained an autoencoder to encode and decode handwritten digits using the MNIST dataset. We also created models for generating digits from the encoded space. A slightly different (and perhaps better) approach would have been to create a variational autoencoder to sample digits. Other potential improvements include implementing sparsity or using disentangled representation. Although we took a fairly simple approach to sampling images, we still obtained good results.

5 References

- Bishop, Christopher M. *Pattern Recognition and Machine Learning*. Springer, 2013.
- Goodfellow, Ian, et al. *Deep Learning*. MIT Press, 2017.
- Chollet, Francois. Building Autoencoders in Keras. *The Keras Blog*, MkDocs, 14 May 2016, blog.keras.io/building-autoencoders-in-keras.html.