**CS4670 Digital Image Compression**

**Final Project Report**

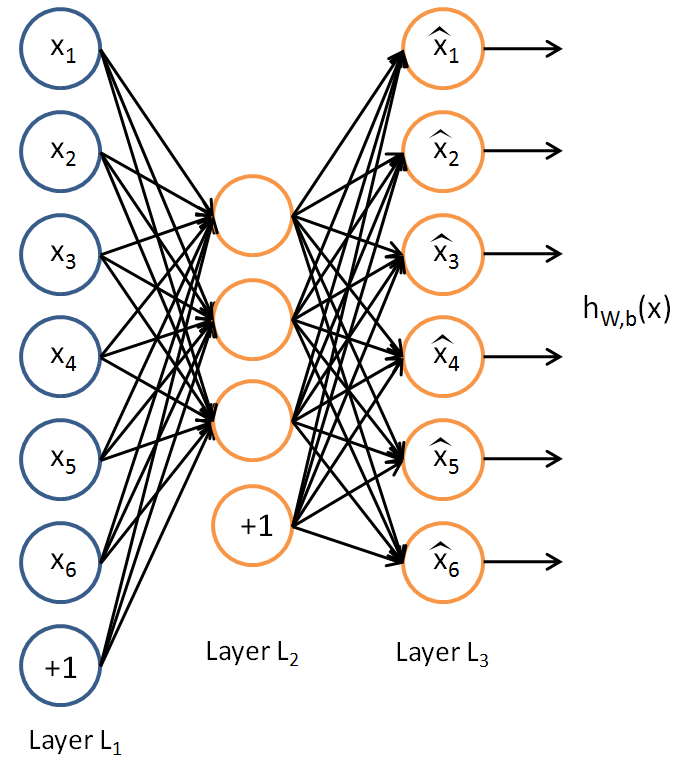
Written by

Frankie Cancino

**Artificial Neural Networks**

The basic idea that software can simulate the neocortex’s large array of neurons in an artificial “neural network” is decades old, and it has led to many breakthroughs. But because of improvements in mathematical formulas and increasingly powerful computers, computer scientists can now model many more layers of virtual neurons than ever before. With this greater depth, they are producing remarkable advances in image recognition. This is the basis of deep learning. Convolutional neural networks model visual perception, and can be applied to visual recognition tasks. One major advantage of networks is the use of shared weight in convolutional layers, which means that the same filter (weights bank) is used for each pixel in the layer; this both reduces memory footprint and improves performance

Here is a visual example of a Neural Network, specifically an autoencoder:



**Autoencoders**

An autoencoder is a neural network that is trained to attempt to copy its input to its output. Internally, it has a hidden layer *h* that describes a code used to represent the input. The network may be viewed as consisting of two parts: an encoder function *h=f(x)* and a decoder that produces a reconstruction *r=g(h).* "Autoencoding" is a data compression algorithm where the compression and decompression functions are data-specific, lossy, and learned automatically from examples rather than engineered by a human. Additionally, in almost all contexts where the term "autoencoder" is used, the compression and decompression functions are implemented with neural networks. To build an autoencoder, you need three things: an encoding function, a decoding function, and a distance function between the amount of information loss between the compressed representation of your data and the decompressed representation (i.e. a "loss" function). The encoder and decoder will be chosen to be parametric functions (typically neural networks), and to be differentiable with respect to the distance function, so the parameters of the encoding/decoding functions can be optimizing to minimize the reconstruction loss.

**imageCompressionNeuralNet.py**

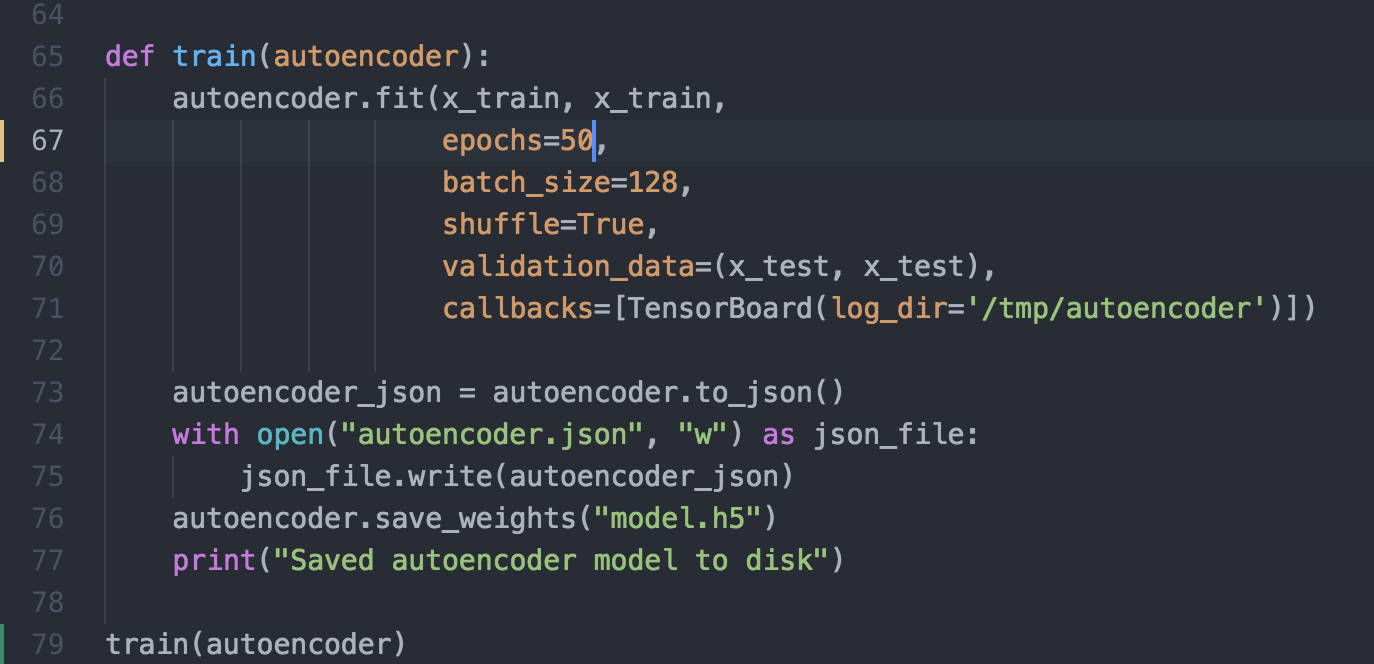
For my implementation for the Digital Image Compression final project, I went a step further from the requirements and decided to build my own Artificial Neural Network. Specifically, I built an autoencoder. As mentioned above, an autoencoder creates an algorithm to both encode and decode an image. I will go step-by-step through the model to show this project meets the requirements and utilizes an advanced method to conduct image compression. Shown below is the code at the beginning of the python file.



The first section is just a list of imports to conduct this experiment. The methods save\_mnist and load\_mnist turn the mnist data from matrices into JPEG format. Later on they are then turned into matrices for compression once again. The images are already grayscale, however there are methods used later to ensure that the images are in fact shown as grayscale images.



This next section of code now creates the Neural Network. It creates a Convolutional Neural Network. As mentioned before, Convolutional neural networks model visual perception, and can be applied to visual recognition tasks. This creates the autoencoder, meaning it creates the encoder with algorithms to apply the DCT, quantization, scanning, and coding of symbols. It also creates the decoder taking care of the decoding, inverse scan, inverse quantization, inverse DCT, and begins to plot the R-D curve that will be shown later. It then creates the mnist load data to prepare for the train and test methods.



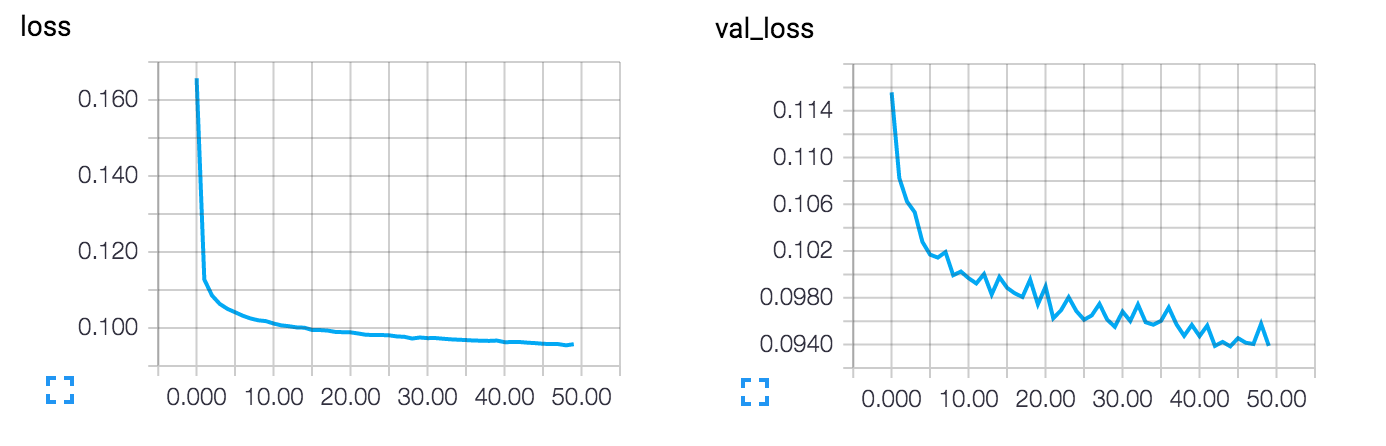
This portion of the code creates the train function. This trains the autoencoder with 60,000 image samples that are grayscale jpeg images. It does this 50 times (hence the 50 epochs). It then sends the R-D curve information through Tensorboard via the log directory (log\_dir=’/tmp/autoencoder’). Also shown in this portion of the code, is the save function. Saving the model makes sure we don’t have to train the model every time which takes approximately 2 hours for 50 epochs. This saves the model and the weights so we can send images through without taking much time. Then the train function is called when needed to train the data.

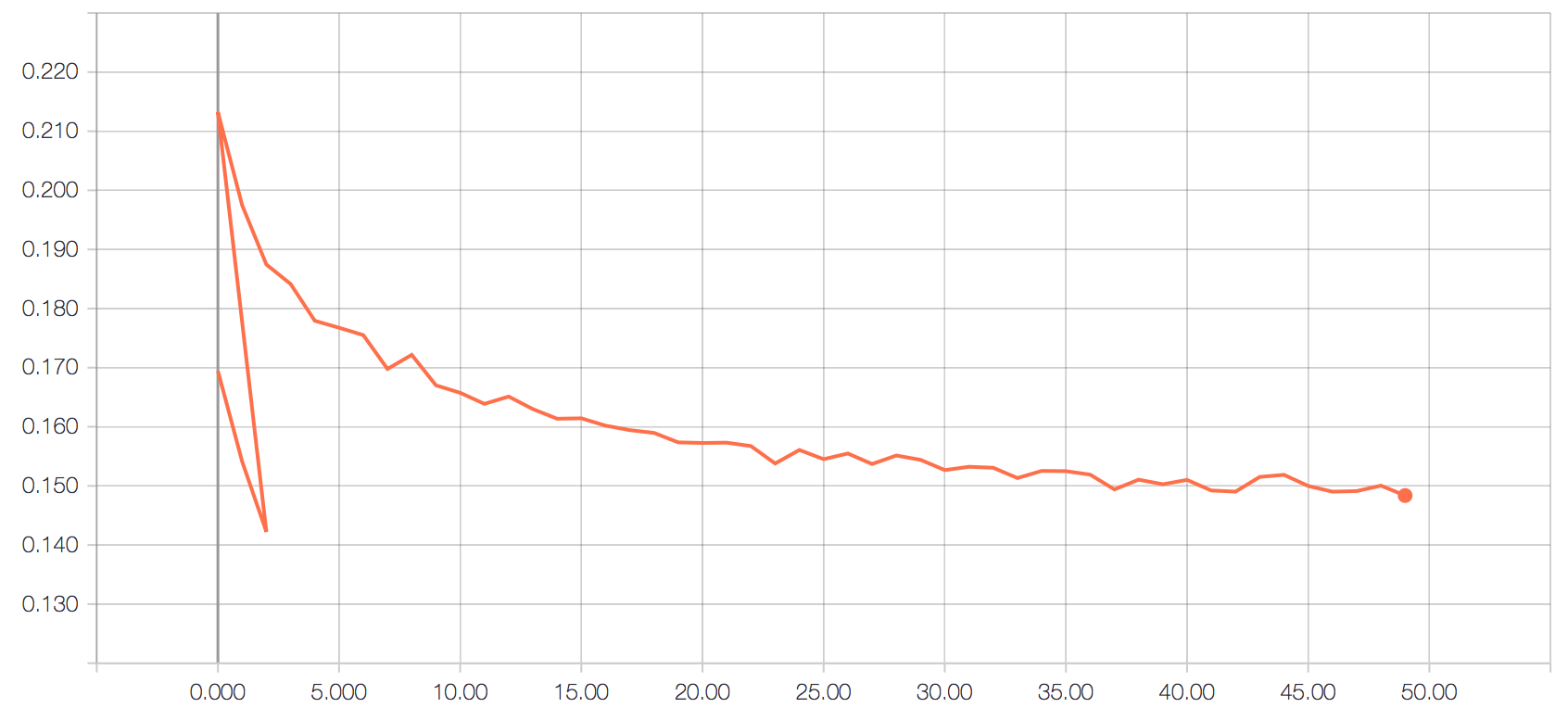
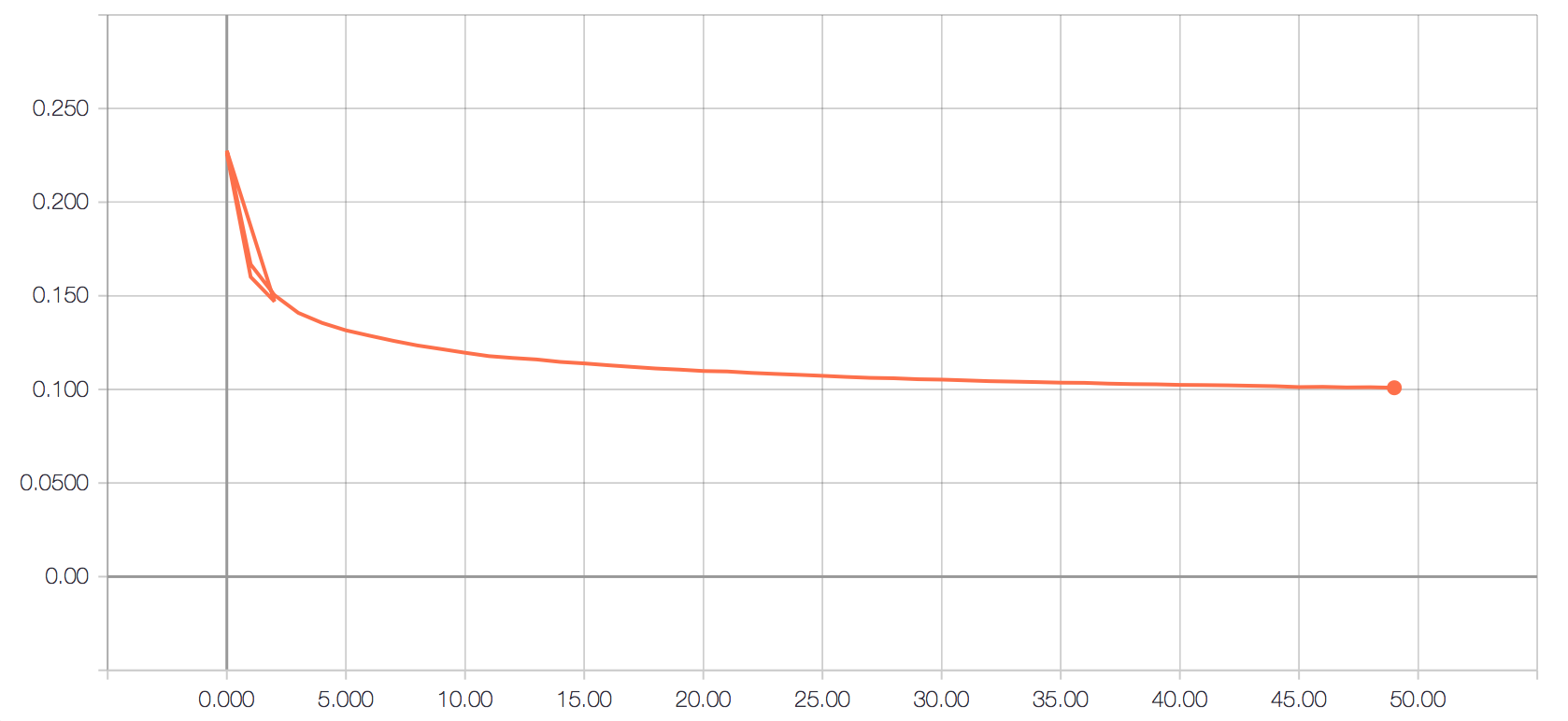


Finally, the last of the code loads the machine learning model in order to use the saved model. Again, in order to not take up 2 hours of time to train the data each time. Once loaded it uses the decoded images to plot the images. Using matplotlib, it displays the original images and the decoded images next to each other.

**R-D Curve**

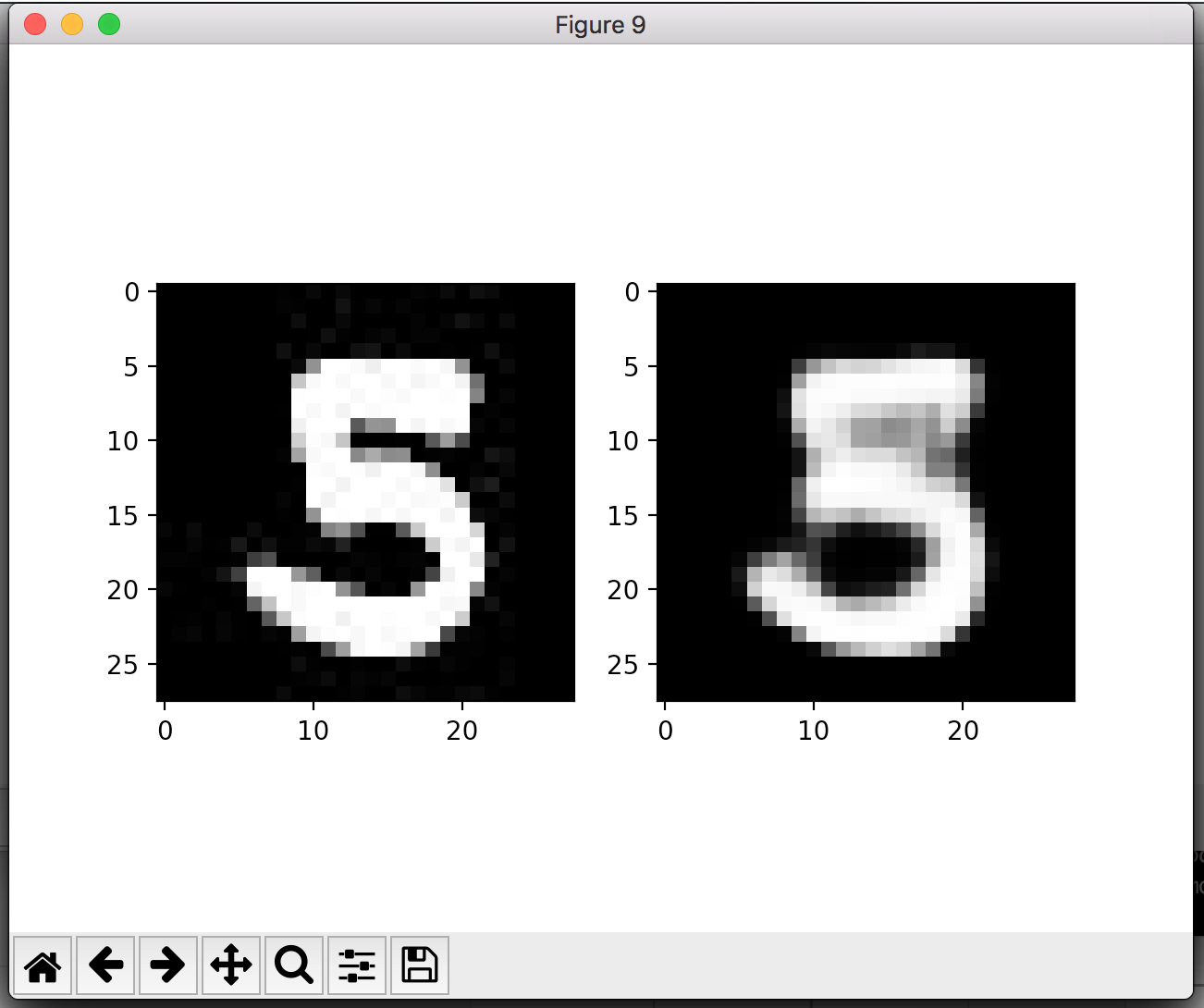
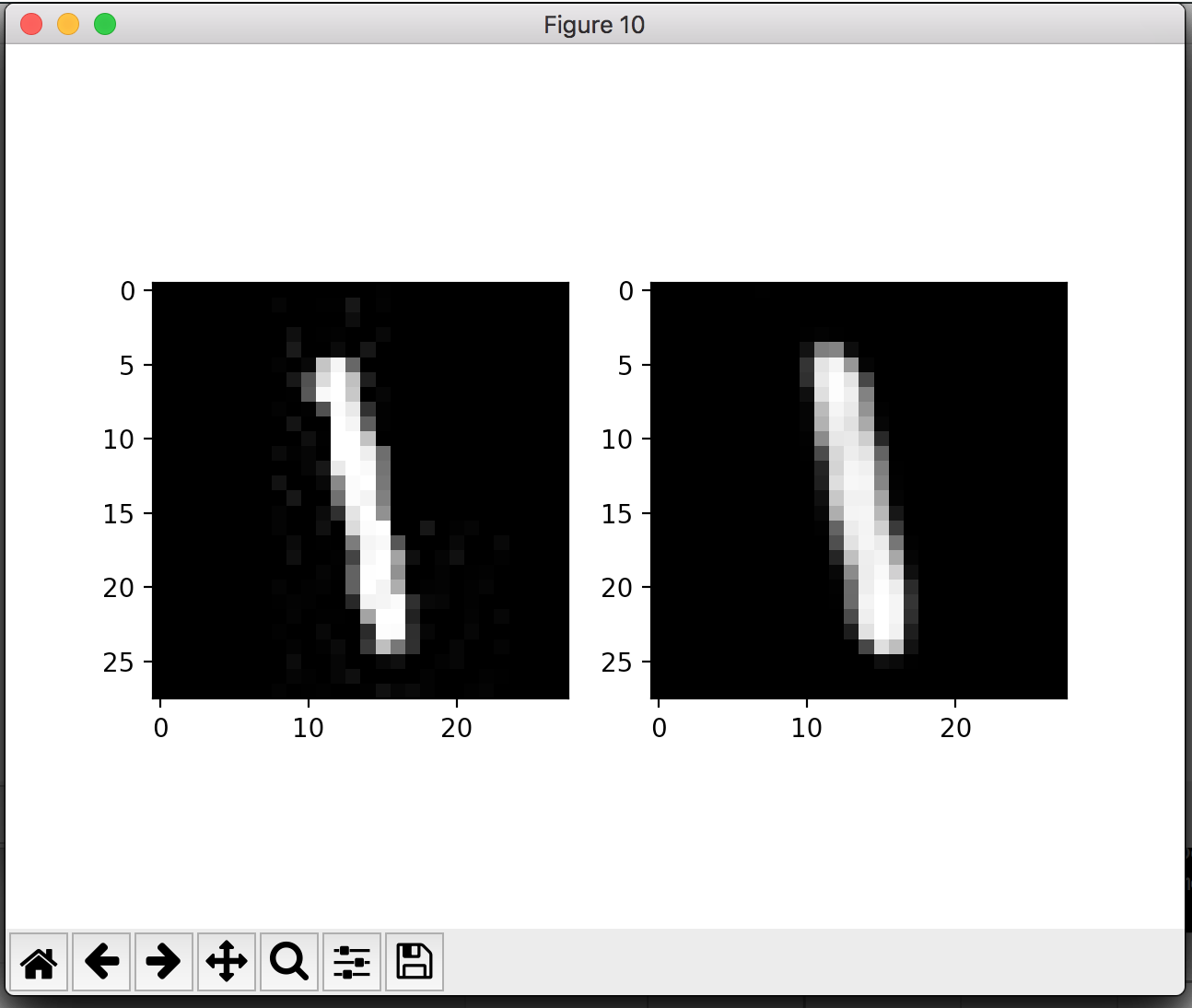
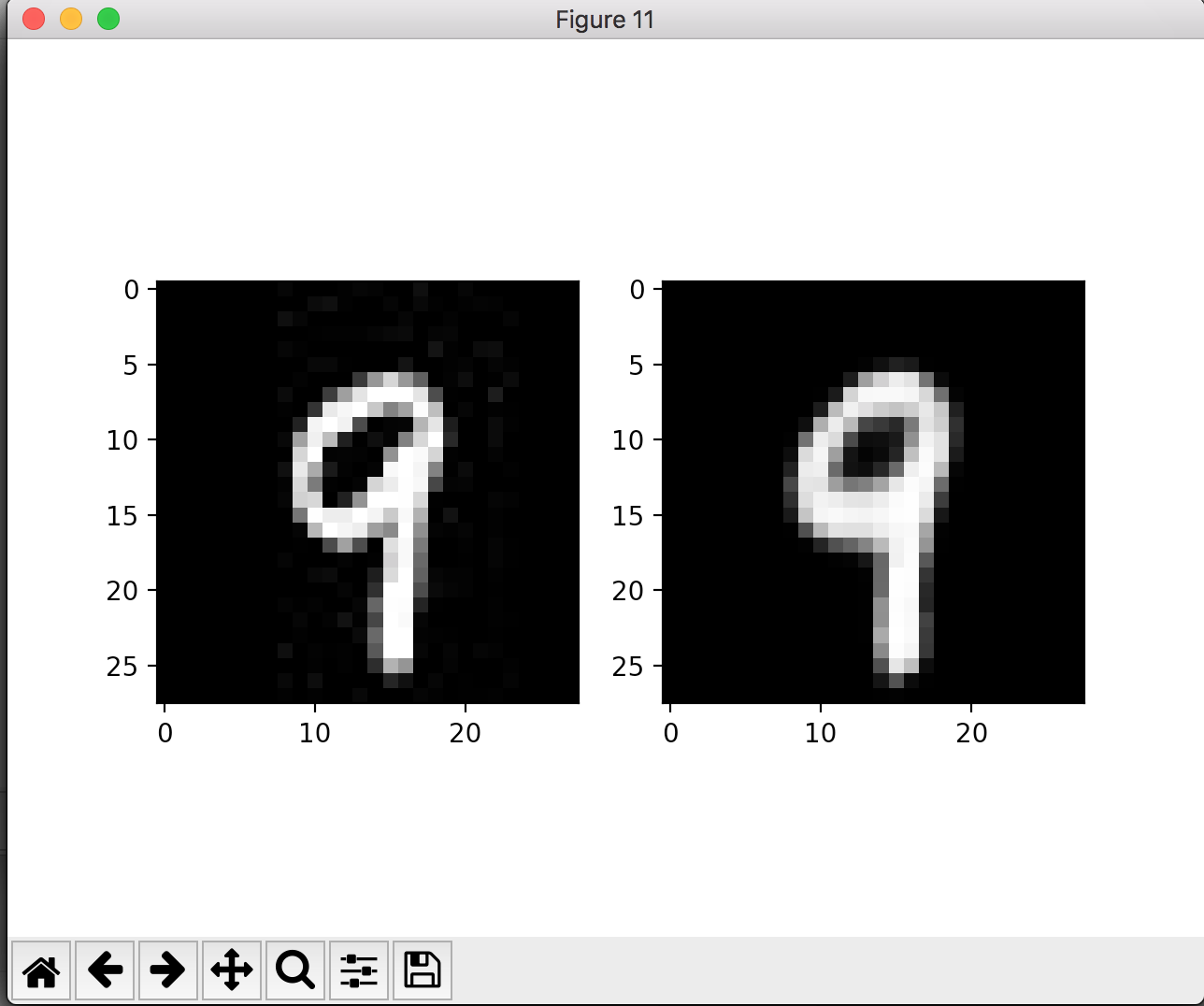
Below is the R-D curve shown after running the autoencoder 50 times (50 epochs) with 60,000 sample images. The error rate tailors off around 0.094. It uses binary cross entropy to help calculate this curve after compiling the model. The image shown is a sample, also shown below that is the actual R-D curve I found when running my model multiple times with different number of epochs. Which is why the curve has variations.





**Output**

*Images:*

****

*Encoder Output:*

