Homework 2 - Building Autoencoders using DNNs, CNNs, and LSTMs

Ben Fox
University of California, Santa Barbara

May 5, 2019

Abstract

This paper demonstrates the encoding and decoding of a Pokemon image dataset utilizing three models, a vanilla deep neural network (DNN), a convolutional neural network (CNN), and a long short term memory recurrent neural network (LSTM). All models were built using Python and Keras. Overall, the three classifiers produced poor accuracy results and decent mean squared error (mse) results. The DNN, with one encoding and one decoding layer, had accuracy of 38.03% and mse of 0.034. The CNN also performed poorly from an accuracy stand point. The CNN encoder contained two convolutional and two max pooling layers, and the decoder had two up sampling and two convolutional layers. It had a test accuracy of 37.76% and mse of 0.032. Lastly, the LSTM performed poorly. The LSTM encoder contained one LSTM layer with 512 hidden states. The decoder consisted of one LSTM layer with 1200 hidden states followed by a dense layer. It had a test accuracy of 37.83% and mse of 0.042. Image reconstructions were performed and compared to original images for each model. Overall, this was a great exercise in building a variety of autoencoder models, comparing each one, and visualizing the results.

1 Data

For this assignment, the images used were Pokemon images (found here) comprised of 792 images of different Pokemon. The images were split into a train and test set, utilizing 713 images for training and 79 images for testing (10% test set). This dataset is fun and interesting to apply autoencoders to regenerate Pokemon that are not ”real” Pokemon. An example of a Pokemon image from the dataset is shown in figure 1.

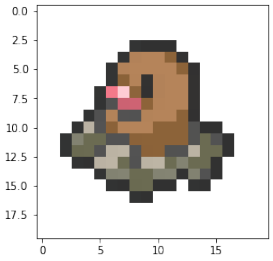


Figure 1: Example image from Pokemon Dataset

Fortunately, this data is mostly pre-proecessed. Images were originally 40x40x3; however, were cropped to 20x20x3 to eliminate whitespace. Additionally, each image was normalized by dividing by 255 to represent pixels between range zero and one. The dataset is summarized in table 1.

Original Images	**Processed Images**	**Train Split**	**Test Split**
792x40x40x3	792x20x20x3	713x20x20x3	79x20x20x3

Table 1: Pokemon Dataset Image Summary,

Overall, this dataset was enjoyable to work with because it consisted of interesting and fun images and provided me the opportunity to delve into the Keras framework to build autoencoders utilizing a variety of techniques without extensive training times.

2 Network

2.1 Vanilla DNN Autoencoder

The DNN built was a one layer neural network for the encoder that compressed the Pokemon images from a 1200 dimensional vector to a 512 dimensional vector. Thus, the layer was comprised of a total of 512 neurons. The layer compressed the input by 57%. The decoder rescaled this image back to 1200 dimensions (neurons) to reconstruct the image. The associated network graph, scaled down by 100, is shown in figure 2. The encoder layer was batch normalized and utilized the Relu activation function. The decoder layer was batch normalized and utilized a sigmoid activation function. Biases and weights were initialized to be zero. The optimizer used was the Adams optimizer and the loss func-

tion used was binary cross entropy, since all values predicted are in the range [0,1]. An overview of the encoder and decoder layers are seen in table 2.

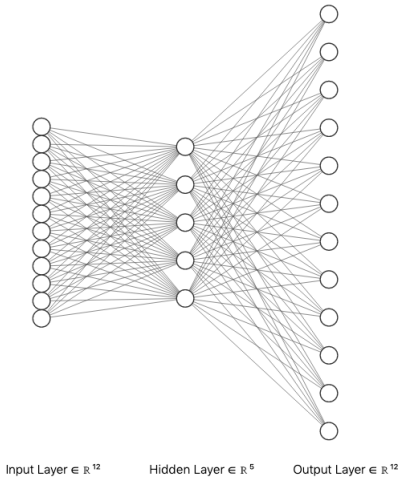


Figure 2: Vanilla DNN Autoencoder Graph

Layer	Number of Neurons	Activation
Input	1200	NA
Encoder	512	Relu
Decoder	1200	Sigmoid

Table 2: Vanilla DNN Autoencoder Model,

2.2 CNN Autoencoder

The CNN built was a four layer encoder neural network, consisting of two convolutional and max pooling layers. The decoder network was four layers as well, consisting of two up-sampling and convolutional layers. The CNN compressed the images from a 20x20x3 representation to a 5x5x32 representation, thus having a compression factor of 33%. The associated network graph is shown in figure 3. The encoder layers were both batch normalized and utilized the Relu activation function. The decoder, inner CNN layer was batch normalized and utilized a Relu activation function, and the final convolutional layer was

batch normalized and utilized a sigmoid activation function. The layers' filter sizes are detailed in table 3. As in the DNN, biases and weights were initialized to zero. The optimizer used was the Adams optimizer and the loss function used was binary cross entropy.

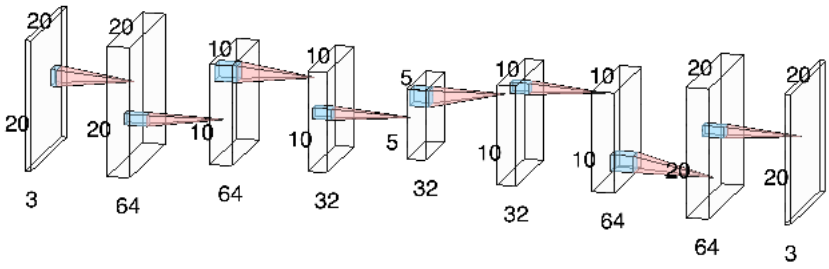


Figure 3: CNN Autoencoder Graph

Layer	Size	Filter Size	Activation
Input Image	[20,20,3]	NA	NA
Encoder CNN 1	[20,20,64]	[3,3]	Relu
Encoder Maxpool 1	[10,10,64]	[2,2]	NA
Encoder CNN 2	[10,10,32]	[3,3]	Relu
Encoder Maxpool 2	[5,5,32]	[2,2]	NA
Decoder Up-Sample 1	[10,10,32]	[2,2]	NA
Decoder CNN 1	[10,10,64]	[3,3]	Relu
Decoder Up-Sample 2	[20,20,64]	[2,2]	NA
Decoder CNN 2	[20,20,3]	[3,3]	Sigmoid

Table 3: CNN Autoencoder Model

2.3 LSTM Autoencoder

The LSTM built was a sequence to sequence pixel autoencoder, which took a pixel sequence, compressed it into a lower dimensional vector, and decoded that vector to reconstruct an image. The encoder consisted of a one layer LSTM, which took a sequence of 1200 pixels with one time step (20x20x3 = 1200) and compressed the pixel representation into a final state vector of dimension 512 (the number of hidden layers in the LSTM). The

compression factor for the LSTM was thus 57%. The decoder network was a one layer LSTM, followed by one dense layer. The decoder LSTM took the 512 dimensional vector and predicted a sequence of pixels. The associated network graph is shown in figure 4. The encoder layer utilized the Relu activation function. The decoder layer utilized a Relu activation function, and the final dense layer was batch normalized and utilized a sigmoid activation function. The layers' filter sizes are detailed in table 4. As in the DNN and CNN, biases and weights were initialized to zero. The optimizer used was the Adams optimizer and the loss function used was binary cross entropy.

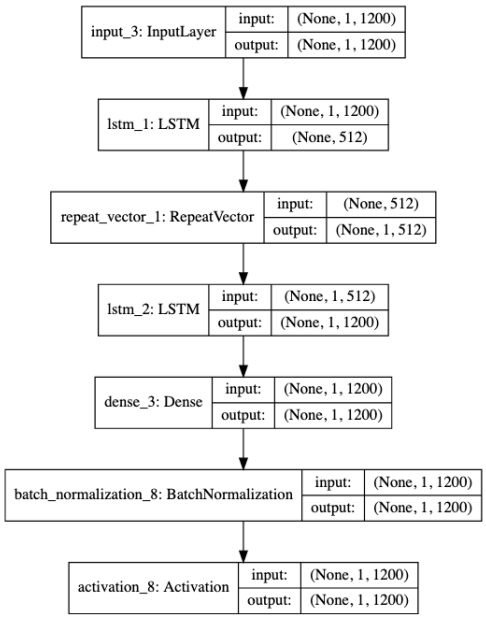


Figure 4: LSTM Autoencoder Graph

Layer	**Size**	**Activation**
Input Image	[1,1200]	NA
Encoder LSTM	[1,512]	Relu
Decoder LSTM	[1,1200]	Relu
Decoder Dense	[1,1200]	Sigmoid

Table 4: LSTM Autoencoder Model

3 Training

For all models, each batch size was 32 images. A batch size of 32 was chosen because of the small sample size and because smaller batch sizes tend to generalize better. Further, 100 epochs of training were done for each model. The corresponding loss per epoch are shown in figure 5 below for all three models.

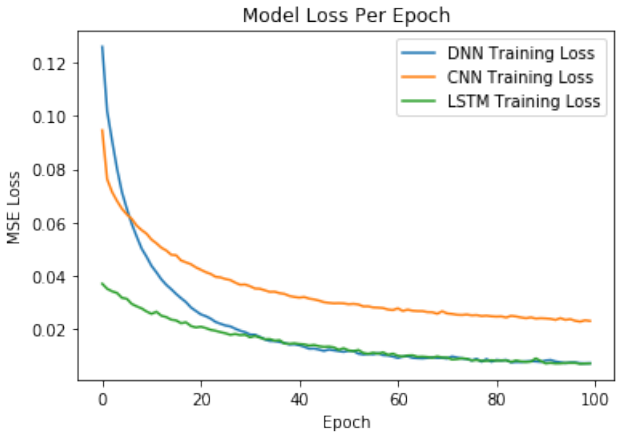


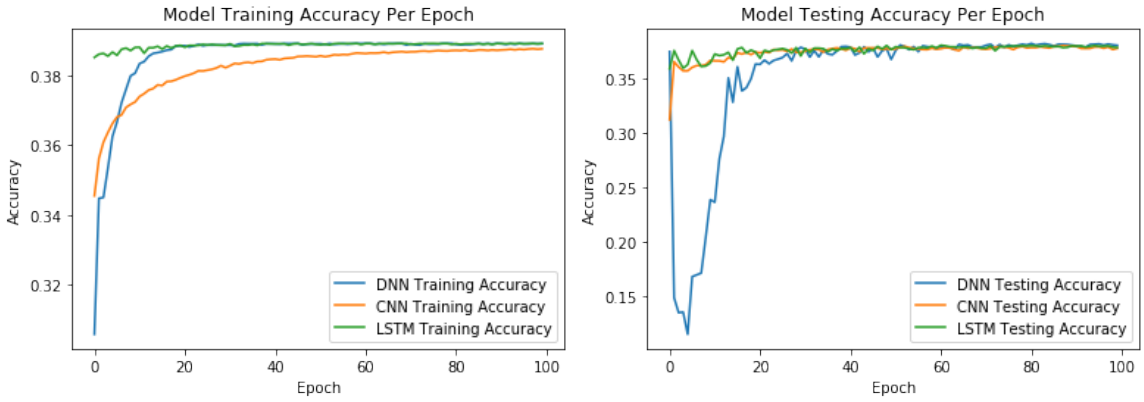
Figure 5: Model MSE Loss

4 Validation

For all models, accuracy was low and maxed out after about 30 epochs of training. Figure 6 below shows the accuracy measure for all three models per epoch. There was no indication of overtraining. Additionally, accuracy and mse measures are reported for the test set over all models. See table 5.

Model	**Accuracy**	**MSE**
DNN	38.03%	0.034
CNN	37.76%	0.032
LSTM	37.83%	0.042

Table 5: Model Evaluations



(a) Train Accuracy (b) Test Accuracy

Figure 6: Model Train and Test Accuracies

5 Visualization

Reconstructed images are shown and compared to 10 test images for each model below in figures 7,8, 9.

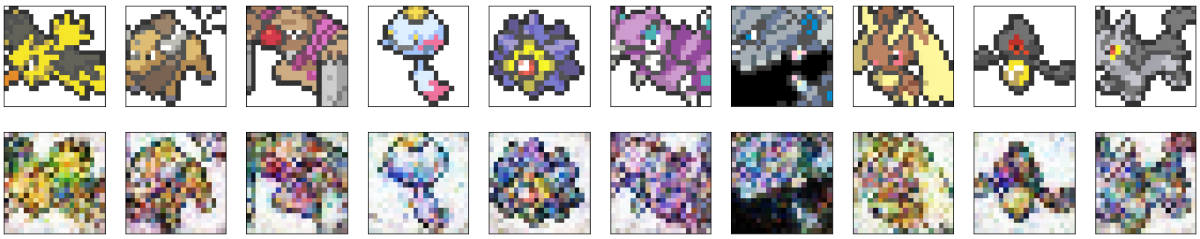


Figure 7: LSTM Reconstructions

6 Discussion

Overall, this was an interesting and stimulating assignment. Building the three different models provided me with the opportunity to understand the differences between models and use cases of each model. For example, CNNs and DNNs reduce loss more in autoencoder cases than LSTM, which leads me to believe that for images, CNNs and DNNs are better. This is likely because CNNs are able to understand the multidimensionality of

Figure 8: CNN Reconstructions

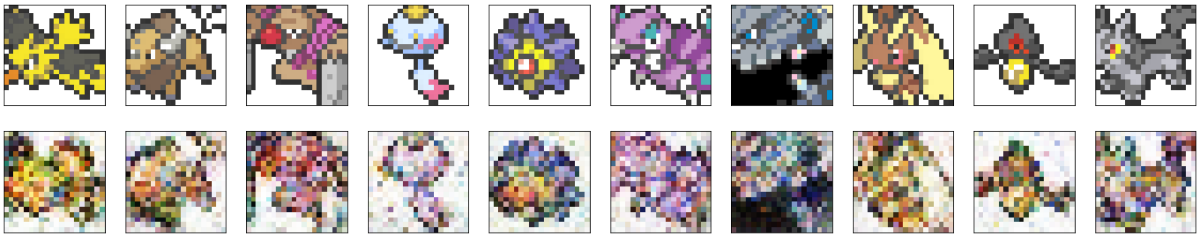


Figure 9: LSTM Reconstructions

images as well as neigborhoods of pixels better than LSTM. DNNs are also very good at learning non-linear relationships, which likely contributed to its lower loss.

Additionally, I learned that the accuracy measure for autoencoders may not be the best metric to evaluate a model. As I was trying a variety of different methods to increase accuracy, I noticed that even low accuracy techniques (not reported) had good reconstructed images. This is likely because the accuracy of a pixel does not have to be an exact match, and the relationships between the pixels is more important. This is why the neighborhood of pixel selection is so important when building autoencoders.

Lastly, throughout my many different designs for the three models trained, I noticed that batch normalization and sigmoid activation provided the best results. This is likely because the pixel representations (numbers) are consistently being normalized and re normalized to lie within the range of [0,1]. The sigmoid activation function then predicts a number in this range as well, which is why it was the best activation function for these models.