Convolutional and Generative Adversarial Networks on the MNIST dataset

Laurent Poirier McClure

ECON 420: Machine Learning

December 24th, 2018

The first part of this paper shows the results of training a convolutional network for image recognition. The second part trains a Generative Adversarial Network to reproduce images. The third part trains a Variational Autoencoder to reproduce images as well. The dataset used is the MNIST handwritten digit database. The training set consists of 60,000 handwritten digits, and the validation set consists of 10,000 such digits. The sample is reshaped to a size of 28x28, and images are black and white.

Part I: Convolutional Network (Lecun et al., 1998)

Experiment 1

The present convolutional network is made of three layers covering a two-dimensional input. The first convolutional layer consists of a 3x3 kernel with 64 nodes. The layer's output goes through a Rectified Linear Unit activation function.The second layer is similar, but with 32 nodes. Since the convolutional layers produce 2D outputs, a ‘’flattening’’ layer is applied to prepare the data for the classifying layer. The model makes use of the Adam optimizer and the cross entropy loss function.The classification decision is made through a softmax function. The classifying layer is set up as a one-hot vector: each input is classified into a vector whose entry is 1 for the chosen category and 0 for other entries. For example, the digit "5" should be classified into the vector whose sixth entry is a one, and whose other entries are zero.

The model's average accuracy over the first three epochs was 33.03% for the training set. The model performed slightly better on the validation set, with a 35.34% accuracy rate. This suggests that the model has not overfit yet, as results are better for the validation than for the training set. Due to the nature of the model, there seems to be an underfitting problem that can be remedied by adding convolutional layers. The average execution time per epoch is 378 seconds.

The second round of training exhibits a slight increase in performance, with an average of 40.67% accuracy on the training set and 41.00% on the validation set. The average execution time per epoch is down to 193.67 seconds

Experiment 2

The second version of the model adds an intermediate convolutional layer to remedy the underfitting problem. The new layer has 48 nodes, and its kernel feature map is kept at a 3x3 size.

This added depth sharply increases the accuracy of the model, now at 96.91% over three epochs for the training set, and 98.00% for the validation set. The increasing trend from training to validation suggests that despite very high accuracy, there is still room for learning. On the other hand, additional depth trades accuracy for execution time, which now increases to 515 seconds per epoch.

The second round of training presents an accuracy of 99.20% over the training set, and 98.15% over the validation set. This is the first time the model performs better over the training set than the validation set, suggesting the model is close to the point of overfitting. The fact that the model is near its optimal predicting power is reinforced by a decrease in the running time, now at 310 seconds per epoch. The model now has learned enough to operate with less effort and equal accuracy.

Experiment 3

The next modification aims at improving runtime, since improvements to the accuracy seem to have reached efficient marginal benefits. The third iteration of the model drops the intermediate convolutional layer to reduce computation, but decreases the feature map's size to 1x1 in order to tamper the decrease in accuracy. Training over the first three epochs gives an average accuracy of 87.40% for the training set, and 90.49% for the validation set. While accuracy decreased, the running time is now at 120 seconds. This is slightly more than a third of the previous running time. The second round yields no noticeable change. It thus seems that adding an additional convolutional layer provides near perfect accuracy at the expense of almost tripling the runtime.

Experiment 4

The dropout function is now added to the two-layer model in order to further reduce computational burden. The dropout rate is set at 50%. Accuracy results are similar to the two-layer model. On the other hand, run time drops to an average of 79 seconds for both sets of training. It thus seems that dropout with a 1x1 kernel size is a net improvement, cutting runtime almost by half.

Experiment 5

To consider whether the dropout technique is a net improvement across models, a dropout rate of 50% was also applied to the 3-layer model that previously yielded near-perfect accuracy at 515 seconds per epoch. This dropout rate cut runtime by half to 331 seconds per epoch. However, the results were underwhelming: 15.90% accuracy on the training set for the first three epochs, and 17.70% for the validation set. The second round of training yields no noticeable change in any respect.

This suggests that the additional layer presents a liability if dropout is applied to a limited amount of epochs. It seems that the additional layer’s explanatory contribution is outweighed by the subtracted inputs from that layer. Thus the dropout strategy only becomes efficient when a sufficiently large number of iterations is available, for then the missing information can reasonably be statistically inferred.

Experiment 6

We now cut the dropout rate by half to see whether the 3-layer convolutional network could benefit from this technique. If the added layer only loses a fourth of its inputs, perhaps its explanatory power is still marginally beneficial. A dropout rate of 25% boosts accuracy on the training set to 97.00% on average over all six epochs, and to 97.89% for the validation set. On the other hand, runtime sharply increased to 683 seconds per epoch.

Experiment 7

Early stopping was also applied to the 3-layer model, replacing dropout.The value monitored was the validation set's loss step. The threshold for early stopping was a loss reduction less than 0.01. Data suggests early stopping is both more faster and more accurate than dropout rates of 25% and 50%. The first round of training on three epochs yielded no stoppage, performing similarly to the 3-layer convolutional network above. Stoppage occurred during the second round of training, after the fifth epoch. This method results in an average running time of 342 seconds over the 5 epochs. The accuracy at the monitored threshold was 99.20% over the training set, and 98.07% over the validation set. This suggests early stopping works well on models with more depth. Its attractiveness comes from its property of preventing overfitting and reducing runtime, which complements the 3-layer model well.

Part II: Generative Adversarial Network (Goodfellow et al., 2014)

We now move on to a model based on Generative Adversarial Networks (GAN). In this model, there are two components: a discriminator and a generator. The idea is to train the generator to create fake inputs in order to fool the discriminator. In turn, the discriminator is trained on its ability to distinguish between the true training set and generated inputs. The discriminator and the generator train simultaneously. In theory, this is a two-player minimax game, where the unique solution is for the discriminator to distinguish the fake from the real input half of the time. Through this process, the generator learns to create inputs that are closer and closer to the real data.

The present GAN trains on the MNIST dataset. The dataset's inputs were rescaled to -1 and 1 in order to fit the hyperbolic tangent activation function.

The model starts with the generator, which receives inputs Z. Inputs Z are simply random noise at the start of training, and adjust as the model learns. They are channeled through a leaky Rectified Linear Units activation function. The leaky ReLU is chosen to smoothen backpropagation. The leaky ReLU's output is then fed to a hyperbolic tangent activation function. The alpha parameter for the hyperbolic tangent activation function is set to 0.01. The output of the hyperbolic tangent activation function then goes through the discriminator. The discriminator takes inputs from two different sources: the generator and the MNIST dataset. After going through a leaky ReLU, the real and fake inputs are classified by a sigmoid function, which outputs 1 (real) or 0 (fake). To improve performance of the sigmoid output layer, the label of real inputs is smoothened to 0.9 instead of 1. Figure 1 shows a graph of the model.

Losses are computed through sigmoid cross-entropy. The discriminator's loss is the sum of losses from predictions of real and fake images. The loss from real images is computed as the sigmoid cross-entropy between the prediction vector from a real input and a vector of 1's (smoothened to 0.9). The loss from fake images is computed in a similar way, but with a vector of 0's instead of 1's. On the other hand, the generator's loss is computed by the sigmoid cross-entropy between the discriminator's prediction vector from fake input and a vector of 1's, as it wants to maximize the error of the discriminator. The Adam optimizer is used to minimize the loss for the discriminator and the generator separately.

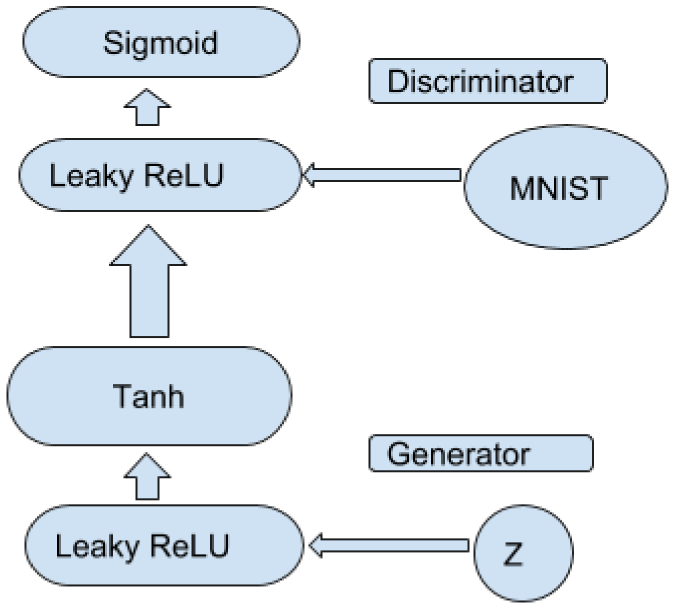

FIGURE 1: Graph of the Generative Adversarial Model

The model iterates over 100 epochs. From figure 2, data suggests the training loss has high variance up until the 10th epoch, and stabilizes thereafter. The generator's training loss hovers around 2 for the rest of the training, and the discriminator's training loss hovers around 1. Figure 3 shows images generated throughout the training, every 10 epoch. One can see the input Z is simply noise at the beginning. The generator then learns to fool the discriminator by creating shapes that look like numbers. By the end of the training, shapes like 6 and 9 look authentic, while shapes like 4 still look blurry. Figure 4 shows inputs generated after the training. The same figures are reproduced for 1000 epochs from figure 5 to 7.

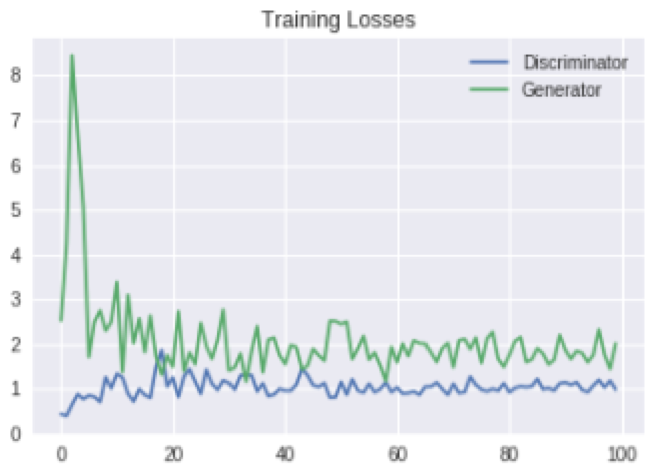

FIGURE 2: Loss as a function of epochs

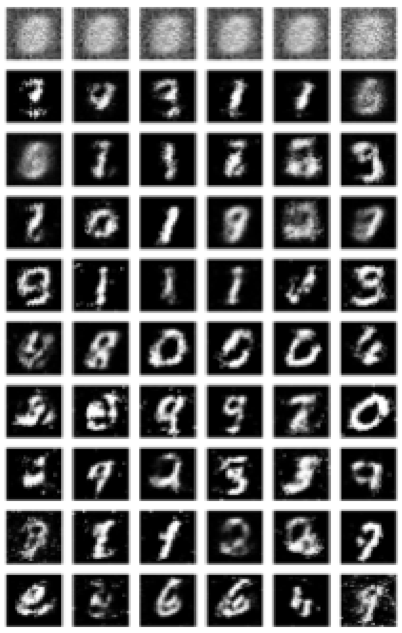

FIGURE 3: Evolution of the generator's output throughout epochs.

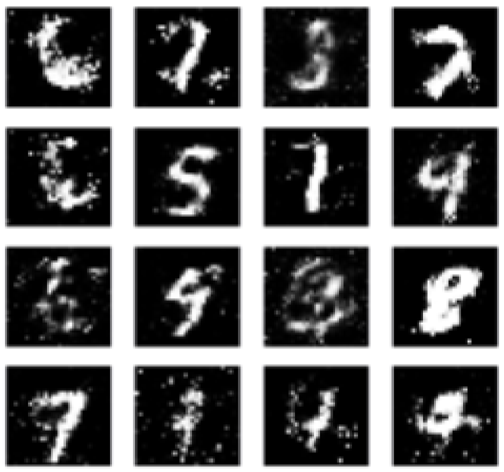

FIGURE 4: Generator's output after 100 epochs

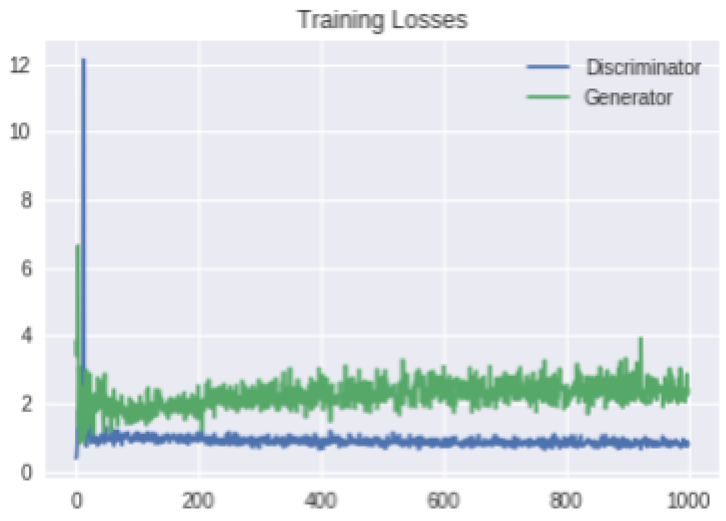

FIGURE 5: Loss as a function of epochs

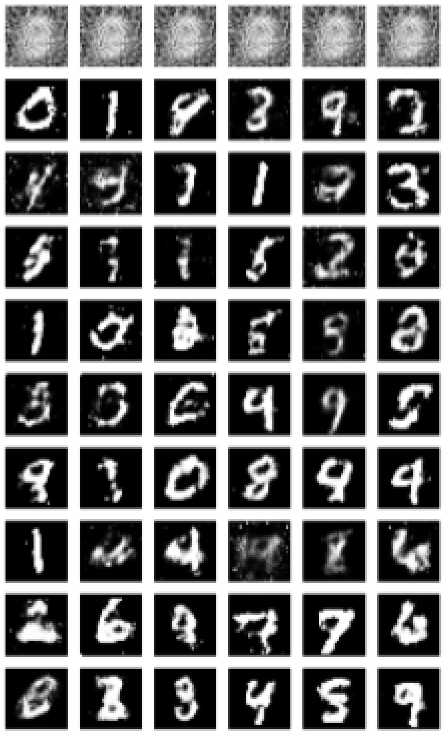

FIGURE 6: Evolution of the generator's output throughout epochs, 1000 epochs

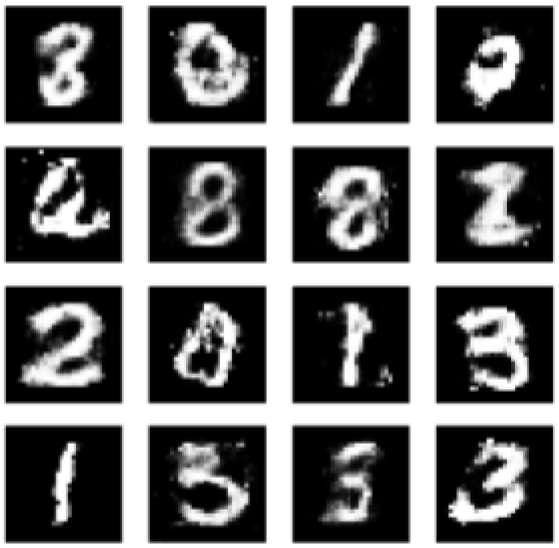

FIGURE 7: Generator's output after 1000 epochs

Part III: Variational Autoencoders (Kingma and Welling, 2013)

This section deals variational autoencoders (VAE), which are an alternative method for reproducing images. While a generative adversarial network accomplishes such tasks through generators and discriminators, a VAE uses probabilistic encoders and decoders. The model takes an input X and channels it through an encoder. This encoder will then generate inputs Z through a probability distribution conditional on input X. Those encoded inputs are then transformed by a decoder that tries to reproduce the original output.

In order to encode the dataset, we use variational inference to approximate a family of Gaussian distributions, which are indexed by their mean and variance. To avoid inferring means and variances for every single datapoint, the encoder amortizes inference by sharing parameters locally. The encoder produces inputs Z in the form of pairwise mean and variance. Since the model is trained on the MNIST dataset, the data consists of black and white images with pixels taking the value of 0 or 1. Thus the decoder takes the Bernoulli distribution. The parameters of these distributions are optimized in the loss function. Figure 8 presents a graph of the model.

The loss function of this model has two components: one relating to the encoder, and another relating to the decoder. The encoder's loss of information is expressed as the discrepancy between its generated distribution and a standard normal distribution, through the Kullback-Leibler divergence. This incentivizes the encoder to keep representations of similar numbers close together. The loss associated with the decoder is expressed as the log likelihood of a multivariate Bernoulli distribution.

The method used for optimization is stochastic gradient descent. We take the expectation of the gradient of each component of the loss function. This expectation is approximated with Monte-Carlo estimates. The optimization is over the two functions of the encoded input Z in the loss function. In order to optimize these two components, the generated sample is parametrized to make the stochasticity independent of the parameters. Since Z is distributed by a normal distribution, we assume Z equals the mean plus the element-wise multiplication of the standard deviation with an error term whose distribution is standard normal. The loss function's components of Z are now deterministic with respect to the the parameters of the distribution, paving the way for optimization. Figure 9 and 10 show the manifold and latent space representation.After 40 epochs, the algorithm starts to distinguish between rounded figures such 3, 5 and 9, but much of the digits are still blurry. It thus seems that the generative adversarial network learns handwritten digits faster.

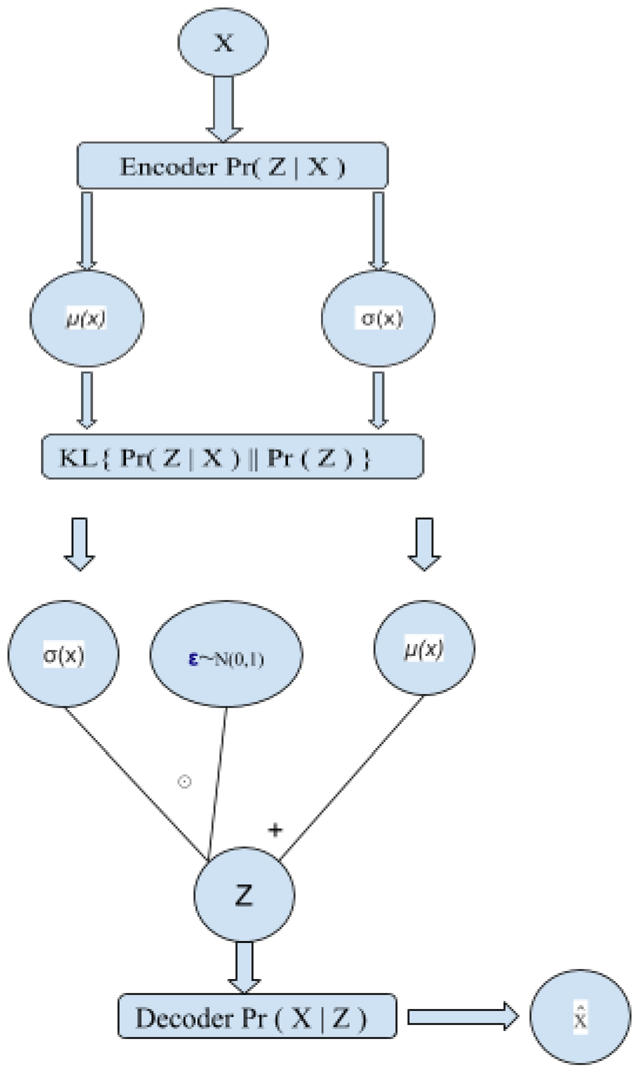

FIGURE 8: Variational autoencoder graph

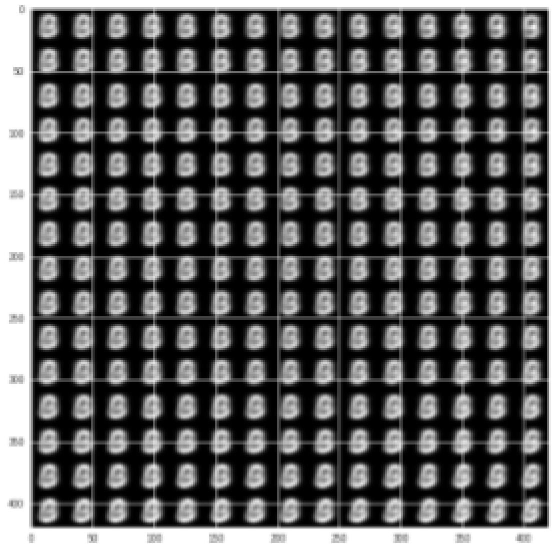

FIGURE 9: Learned two-dimensional manifold, 40 epochs

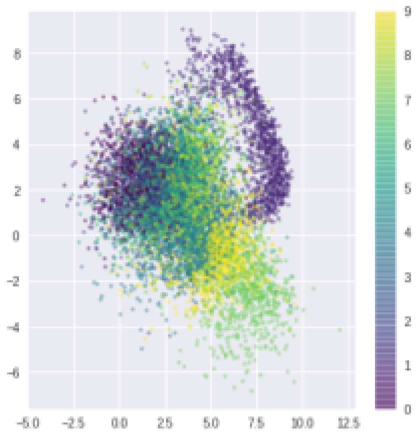

FIGURE 10: Encoder's mapping of the input onto Z-space, 40 epochs

References

Journal Articles

Goodfellow, Ian, et al. "Generative adversarial nets." Advances in neural information processing systems. 2014.

Kingma, Diederik P., and Max Welling. " Auto-encoding variational bayes."

arXiv preprint arXiv:1312.6114 (2013),

LeCun, Yann, et al. “Gradient-based learning applied to document recognition.” Proceedings of the IEEE 86.11 (1998): 2278-2324.

Websites

“Building a Convolutional Neural Network (CNN) in Keras” *Towards Data Science*, 16/09/2018,
https://towardsdatascience.com/building-a-convolutional-neural-network-cnn-in-keras-329fbbadc5f5.

“GAN — Introduction and Implementation — PART1: Implement a simple GAN in TF for MNIST handwritten digit generation” *Towards Data Science*, 06/27/17,
https://towardsdatascience.com/gan-introduction-and-implementation-part1-implement-a-simple-gan-in-tf-for-mnist-handwritten-de00a759ae5c

“A Tutorial on Variational Autoencoders with a Concise Keras Implementation”, 18/01/18,
https://tiao.io/post/tutorial-on-variational-autoencoders-with-a-concise-keras-implementation/