Variational Autoencoder (VAE)

Model Architectures:

The general idea behind VAEs is density estimation. This is essentially the act of learning the distribution of certain features so that you can sample from these learned distributions.

VAEs are very similar in their architecture compared to traditional AEs. They achieve the same goal of condensing/encoding the input information into a hidden representation/latent space. This latent space is then expanded/decoded into being as similar to the original input as possible. Generally they use Dense layers, shrinking the amount of nodes in each following layer until it reaches the desired size of the hidden representation of the original image. An important distinction between this and an AE is that the hidden representation is a probabilistic representation. It is represented as a mean and the associated log variance. Some noise is then injected into this latent representation. This noise enforces that slightly different hidden representations should still produce similar output. That idea is similar to how penalizing partial derivatives work in autoencoders to force the model to have slight differences in the input not cause huge changes in the output. It then expands back up, reversing the Dense layers from before until it reaches the size of the original input again.

VAEs are usually used on image data so instead of simply using Dense layers they could utilize Conv2D layers increasing the number of filters with each subsequent layer. One thing to note is that instead of MaxPooling layers like in traditional CNNs, VAEs utilize strides to shrink the amount of data. This is because in CNNs you want to know what objects are in the image while in VAEs you would want to know where the objects are in the image, and MaxPooling simply carries the information of what is in the image.

Inputs/Outputs:

The input of VAEs is a training set of what you would like to formulate distributions for. This data that is fed in is then transformed into a probabilistic latent space. This probabilistic latent space is then expanded back into the form of the input, and ideally this output should be as close to the input as possible.

Training:

With the encoder learning the probabilistic representation and the decoder upsampling that distribution back into the format of the original image we can start thinking about how training the model works.

The encoder learns the mean and log variance of the given training data specifically. Then a sample is grabbed from the distribution by adding the mean to the variance multiplied by a random noise value, epsilon. This random sample is then fed into the decoder to attempt to

recreate the input. Keep in mind that different random samples should still be recreated into similar images.

Now the loss function of a VAE still includes the reconstruction loss of an original AE. We still want the input and output to be as similar as possible, enforcing representation in the latent space. However, the new addition to the loss function is to include KL Divergence. This penalty checks that the found distribution loses as little information as possible when representing the standard normal distribution. That enforces normality of the latent space. These two penalties create a latent space where similar inputs are very close together and have normal distributions. These two combined enforce that when sampling between two types of representations it would represent a 'real' looking shape. This allows us to smoothly transition between different types of data.

Generating Samples:

Samples are generated by first randomly sampling from the probabilistic latent space. These random samples are then upscaled by feeding them to the decoder. The output of this process is a randomly sampled image.

Generative Adversarial Networks (GAN)

Model Architectures:

GANs perform the act of sample generation which is different from density estimation that VAEs perform. This sample generation is done by creating two separate neural networks that work against each other. They play a zero sum game, which can be thought of as a game of tug of war where they both are constantly improving and 'pulling' the loss in one direction.

These two different models are called the Generator and the Discriminator. The generator's purpose is to be given random noise and produce fake images that look like the real images in the training set. The discriminator's purpose is to distinguish between which images are real and which are fake. They are both constantly battling over each other trying to learn how to trick the other model, or better discern fake images from real.

The loss function is a combination of the discriminator's confidence that real images are real and the discriminator's confidence that a generated image is fake.

The generator's exact architecture includes a combination of dense layers and Conv2DTranspose layers (using stride) to upsample using LeakyRelu activations, and ending in a tanh activation layer.

The discirminaror's architecture is really just a glorified classifier. It contains several Conv2D layers with LeakyRelu activations ending in a single Dense node.

Inputs/Outputs:

The generator and discriminator have different inputs and outputs. The generator has only random noise as input. This information is then upsampled into producing generated images. These generated images that are intended to look like the training samples are the output. As for the discriminator, the input is both real samples and fake samples. These samples then go through all the layers, outputting in a value indicating the confidence level of the sample being either real or fake.

Training:

When training a GAN, you have to hold either the Discriminator or the Generator constant while you train the other one. This process continues until convergence occurs. It begins with holding the generator constant while you train the discriminator. You update the discriminator params using gradient ASCENT, which is different from gradient descent. This is because the discriminator wants to maximize the loss function. Once the discriminator is updated, you hold it constant and begin to train the generator. The generator is updated using the traditional gradient descent since it wants to minimize the loss function.

This process of training is very unstable and there are several common issues:

1. The discriminator may perform too well and cause the gradient of the generator to vanish. If all samples of the generator are deemed fake by the discriminator with high confidence, the generator can't learn what works well to trick the discriminator and the gradient vanishes. This is a common problem since generally identifying as fake or real is a much simpler task than generating a whole new image based on noise.
    a. A suggestion to solve this issue is to adjust the loss function so that the generator wants to maximize an updated function instead of minimizing. This is called a non saturating GAN loss.
2. Mode collapse is a huge issue where the generator simply hones in on a specific example that the discriminator can't detect as fake. This causes a lack of variety in the generations of the generator which we don't want. We want to the generator encapsulate and generate samples that could look like various different looking images in the real training dataset.
3. Lack of convergence occurs when no equilibrium is reached between the two models. Both models become stuck in a loop of undoing each other's progress, meaning no new information is really learned and the GAN becomes ineffective.

A solution to many of these issues in training is by using a Wasserstein GAN. This WGAN replaces the loss function completely. It renames the discriminator to be called the critic. The loss function makes this critic want to maximize the equation D(x) - D(G(z)), which essentially means it wants to give real samples a higher score than fake ones. The generator ALSO

maximizes, but maximizes a different function. It maximizes D(G(z)), which means that it wants to maximize the score that fake samples get, tricking the discriminator to think that they are real. The loss function becomes a continuous score instead of a confidence value between 0 and 1.This is one of several changes that the WGAN makes to the original model type.

Generating Samples:

The samples are generated solely from the generator model. Once the discriminator and generator have been trained together and reached convergence, the discriminator is no longer used. In order to create new samples, we just input random noise into the generator and the output is the generated sample. One interesting thing that could be done is to input the same noise and see overtime how the model improves upon actually generating real looking samples over time as it gets trained.

Conditional Generative Adversarial Networks (CGAN)

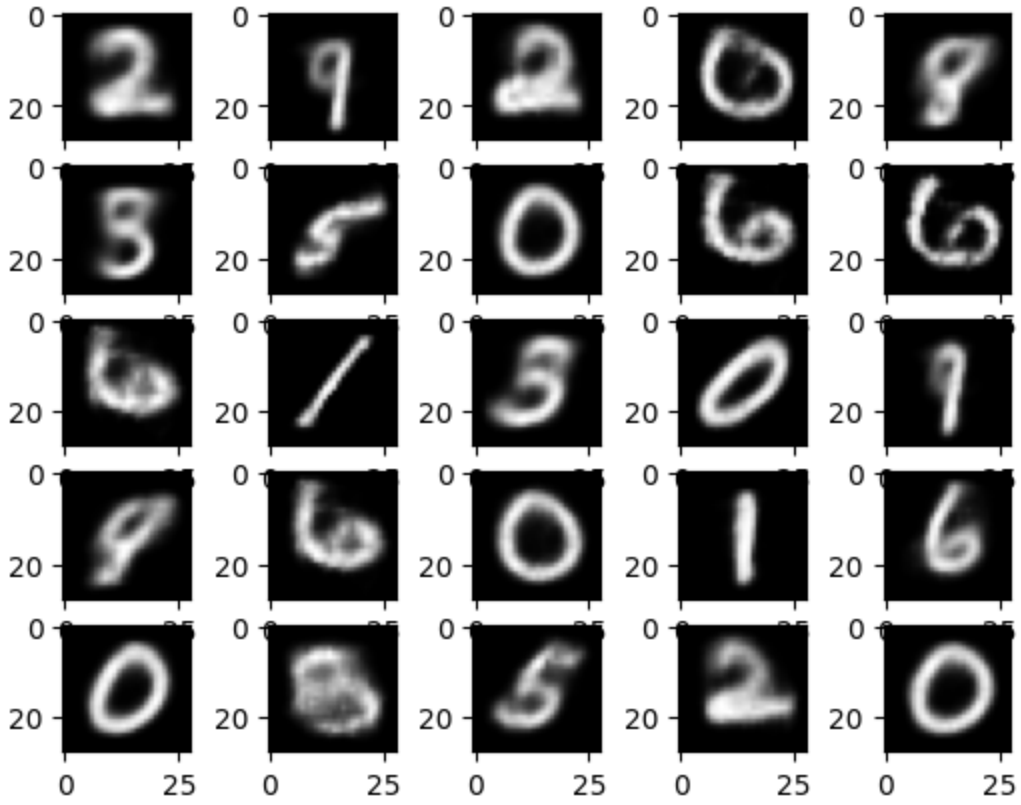
CGANs act very similarly to normal GANs, with a slight adjustment. They produce the same outputs, but they have additional information about the training set. The generator is told to generate an image given a certain condition. The discriminator is told to classify the image as real or fake given a certain condition. This condition in the case of the MNIST data set could be the class of the image. That means the GAN knows what digit it should be trying to generate, or which digit it is looking at to classify as real or fake. This class information can be fed in as one hot encoded vectors.

This additional information can make the models more useful. We can interpolate images smoothly between different classes of digits. This can help to show what a digit would look like that is a cross between a 2 and a 5.

They have the same training methods as normal GANs just with the new information added as inputs. The generated samples are generated the same as well, just with the class information given to the generator along with the random noise.

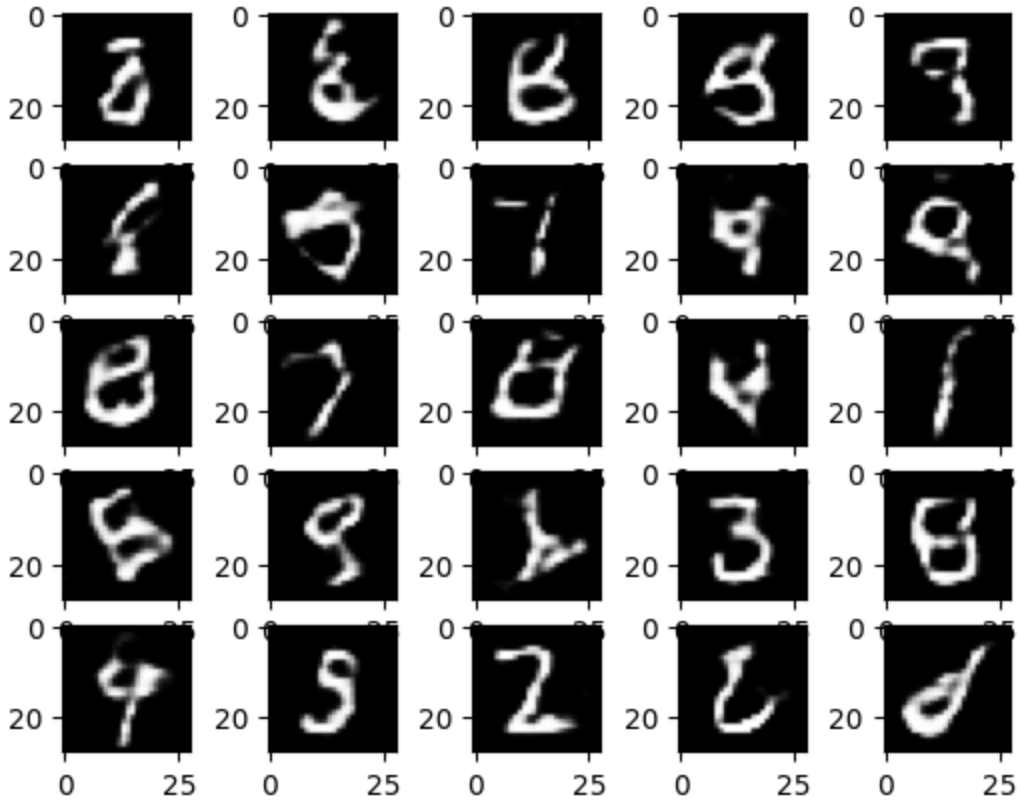
Generated Examples

VAE:



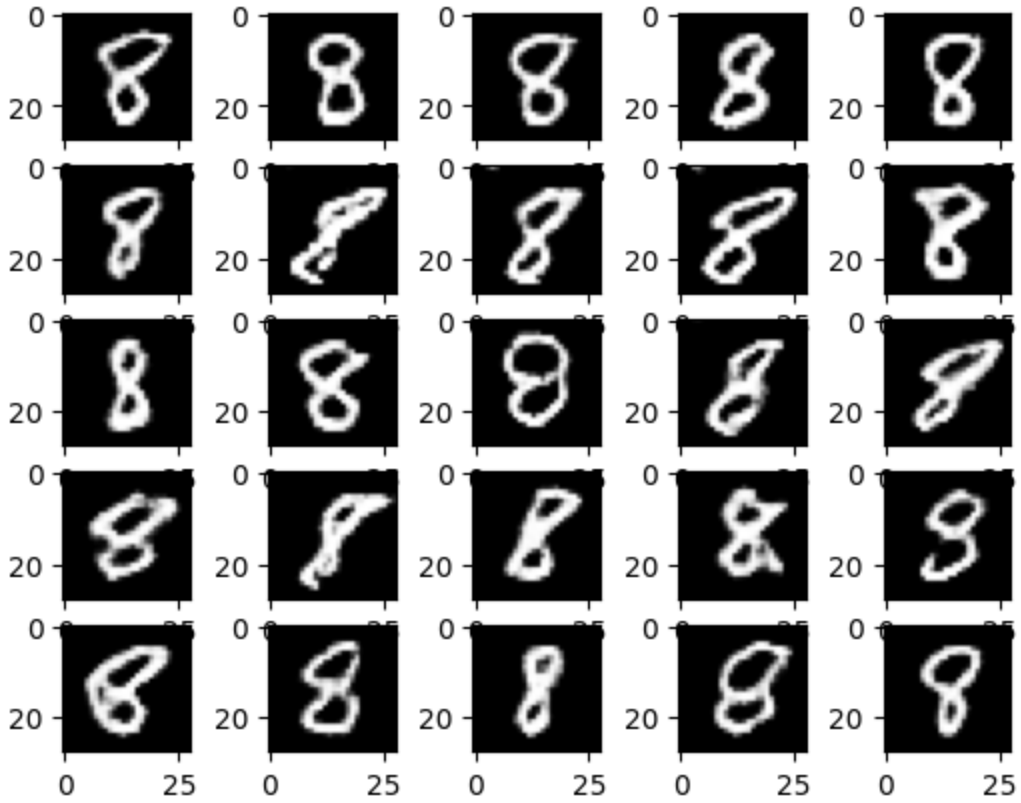
The generated samples from the VAE are generally very blurry. Some of them are even not not distinguishable as any number, which is a huge issue. Outside of those few samples, these look like really good images. It seemed to me like interpolation was impossible with this architecture. These samples were the simplest of the three models to generate.

GAN:



This was the worst of the three in terms of readability and realism. This is likely due to the fact that it doesn't contain the class information as input. The model seems to be generating samples that fall in the latent space between two real classes, in which that don't come out as readable. The best way to describe these images is not as blurry, but as looking faded and not complete. Generating these samples is also pretty simple.

CGAN:



All of these images were generated with the given class of the sample to be the digit 8. As you can see it performed really well. There isn't really blurriness, just a few images that seemed to be a sloppily handwritten 8. It was extremely easy for me to generate samples of a specific class. In my opinion this is the best of the three models. However, these samples were relatively difficult to generate compared to the other two models, yet interpolation was very intuitive.