Autoencoder:

Autoencoders are a form of neural network designed to break a sample down into a smaller representation of itself, which we refer to as the latent dimension. As the network trains, its goal is to minimize the reconstruction loss of a sample. This means that the network is trying to recreate the sample as accurately as possible from its "bare-bones" representation, and thus it must find the best diminished representation of the sample.

Autoencoders are composed of two separate neural networks: the encoder and the decoder. The encoder network has an input which is the same size and dimension as the input data. The data is then propagated through hidden layers, which can be convolutional or dense, depending on the input. The last hidden layer, which must be dense, contains the number of neurons in the latent dimension. Therefore, the network will "encode" the data by compressing it into the size of latent dimension.

The decoder network takes the encoder's output, the latent vector, and propagates it through multiple layers, which can be once again either dense or convolutional. The layers increase in size, with the decoder output layer being the same size as the encoder input. These two models are combined and trained together, with the goal of recreating a sample as accurately as possible.

Autoencoders suffer from several issues, however. The most prominent is that the network does not understand that points close to each other in the latent dimension are related; therefore, when trying to generate new, synthetic samples, the autoencoder struggles unless it has seen the exact point before. One way to combat this problem is by sampling from a distribution around our latent representation rather than a single point in the smaller dimension. This allows the autoencoder to create samples that are *similar* to those it has seen before, but not exactly

identical. In addition to this small change, we attempt to map points to distributions which are as close to the standard normal as possible. This makes the network behave in a more predictable manner. We measure the efficacy of this technique using Kullback-Liebler (KL) Divergence. The incorporation of these two techniques into the autoencoder's structure form the basis for a variational autoencoder, or VAE. Because of the performance advantages of VAEs, our annotated code document will only cover how to make this autoencoder architecture.

Generative Adversarial Networks:

Generative Adversarial Networks (GANs) are a type of neural network similar to autoencoders, but with a slightly different training process. Like an autoencoder, GANs feature two distinct networks: the generator and the discriminator. The basic idea behind a GAN is that the generator network creates fake samples from a sample of noise, and these samples are fed to the discriminator network alongside actual samples from our data. The discriminator must then determine whether a sample is fake or real. The goal of the network's training is to improve the performance of both networks over time. The discriminator will be able to accurately discern between real and fake samples, while the generator will be able to create samples so realistic that the discriminator cannot tell if they are real or fake. Eventually, once the generator is trained well enough, it should be able to produce realistic synthetic samples from a sample of noise.

The generator network has a similar structure to a decoder. It receives a random, one-dimensional sample and propagates the sample through many layers, eventually outputting a sample of the same shape as our actual data. The discriminator network has multiple inputs. It receives the generator's created samples and a batch of real samples. It takes these samples and propagates them through a series of hidden layers, eventually ending in a binary classification task where 0 is assigned to a fake sample and 1 is assigned to a real sample.

Although the construction of a GAN is relatively simple, its training process is more complex. The discriminator and generator are trained adversarially, meaning that when the generator is learning, the discriminator is not, and vice versa. When the generator is training, it creates synthetic samples and feeds them to the critic, which attempts to classify these samples as real or fake. The generator learns from the accuracy of the discriminator's classifications, and it adjusts its samples to fool the generator more often.

Once the generator has trained for a certain time period, its weights are frozen, meaning that it is not able to learn. Additionally, the discriminator's weights are unfrozen, meaning it can now learn. Now, the generator feeds the discriminator fake samples alongside real sample inputs, and the discriminator tries to classify them as real or fake once more. However, the discriminator is now the part of the network that is learning and improving over time. This process continues for the specified amount of epochs, and the convergence of the generator and discriminator are measured separately.

In recent years, two major additions have become standard in GAN architecture. The first is the use of wasserstein loss, which more accurately measures the convergence of the network. A lower wasserstein loss number indicates a more successful training process. Due to the evaluation given by this metric, in a Wasserstein GAN (WGAN), the discriminator is referred to instead as the "critic." In addition to the use of wasserstein loss, a gradient penalty is applied to the loss function. This prevents drastic fluctuations between predictions made by the critic, lowering computational time and improving convergence.

For our purposes, we will be working with a GAN specifically designed to handle Microbiome Data known as the MBGAN. The major change in the MBGAN is the inclusion of two layers in the critic: a phylogenetic transformation layer and a logarithmic normalization

layer. The phylogenetic transformation layer calculates the relationships between taxa in our dataset using distance measures, and it factors these relationships in the critic's calculations. The logarithmic normalization layer is used to normalize our input data, which is typically full of 0 values, causing it to be skewed right. These two changes make the MBGAN the generative network that is best suited to handle microbiome data.

Applying VAE Architecture to our Data

In order to use VAE architecture, we must import the VariationalAutoencoder class from the MicrobeVAE.py file. After importing this class, along with the necessary packages, we can begin to instantiate the VAE. First, it is important to preprocess the data. This step varies, as input data can come in many different formats. However, it is crucial for the VAE input that *each row represents one sample, and each column represents one taxa.* Unfortunately, it is not feasible to include metadata in the VAE, since it is better equipped to handle numeric data.

After preprocessing the data, we must split our data into a training and testing dataset. It is recommended to partition 80% of the data for training, and 20% for testing.

Once the data has been split into training and testing sets, we can begin to create the VAE architecture. This is done by calling the VariationalAutoencoder class and specifying hyperparameters as arguments. We specify the input shape of our data, a list of values representing the number of neurons in each encoder layer, a list representing the number of neurons in the decoder layer, and the size of the latent dimension. When creating a new model, we recommend the certain values for each parameter. The number of neurons in each encoder and decoder layer should be less than the input size, but greater than the latent dimension. The latent dimension should be smaller than the input size; a good rule of thumb is around 1/10 of the input size, but this can be modified and tested. After defining these hyperparameters, we can

summarize the encoder and decoder in order to see their architectures. We can also use a process called gridsearch to test hyperparameter combinations after we create a functional network. This involves defining a "grid" of different combinations of hyperparameters and testing the results of each combination.

Once we have specified the network's hyperparameters, we can train it. First, we define the learning rate, reconstruction loss weight, number of epochs, and batch size. The learning rate should be rather small; we recommend values between 0.01 and 0.0001. We recommend a reconstruction loss weight of 10,000, 1000 epochs, and a batch size of 16. We can then compile the VAE with the learning rate and reconstruction loss weight, and we can train the model with our training dataset, number of epochs, and batch size. After training is complete, we can evaluate the network's accuracy by gauging its reconstruction loss and KL divergence. In an optimal scenario, these metrics should be as low as possible.

Applying WGANGP Architecture to our Data

Applying a basic WGANGP to our data is a similar process to the application of the VAE. Like with the VAE, we must import the WGANGP class from our *MicrobeWGANGP.py* file. After doing this and importing other necessary packages, we can begin the preprocessing step.

The input data format for the WGANGP is exactly the same as the input format for our VAE. Once again, each row represents one sample, and each column represents one taxa. After the data has been preprocessed, we once again perform an 80-20 Train-Test split on our data.

Instantiating the WGANGP requires the specification of a larger number of parameters. For both the critic and generator, we must define the the number of dense neurons per layer (in list format), the batch normalization momentum, the activation to use, the learning rate, and the

dropout rate. We must also specify the input dimension of the data, the size of the input dimension for the generator, the model's optimizer, the gradient penalty weight, latent dimension size, and batch size. The recommended values for all parameters are specified in our *FrackingWGANGP* jupyter notebook. After specifying all of these parameters, we can instantiate our model.

After summarizing the model architecture, we can begin to define the training process. We must include the number of epochs to train for, the ratio of critic to generator training steps, and the batch size of the model. We can then call the train method on our GAN, with our input data and these specifications as arguments.

After training has completed, the best way for us to evaluate the effectiveness of the model is to examine predicted sample outputs, its loss metrics, and pcoa plots. If synthetic samples are distributed similarly to the original, loss metrics are sufficiently low, and pcoa plots are sufficiently similar, we can further proceed with our simulated data.

Applying MBGAN Architecture to Our Data

In order to use generative adversarial networks on our data, one needs to understand the file structure and creation process of these networks. To create the Microbiome GAN (MB-GAN), we use two python files: *MBGAN.py* and *utils.py*. The *MBGAN.py* file is a python file that lets us create the architecture and graph for the network, while the *utils.py* file contains necessary background and utility functions for running and evaluating the network. These modules are imported into the main python file along with all other necessary modules. In order for the MB-GAN to work properly, it is important to disable eager execution in tensorflow, as seen in the annotated code document.

After importing necessary packages in the main document, we define a python function to save the network's output. This allows us to compare samples at different points in the network's training. Relevant literature suggests saving a batch of synthetic samples every 1000 epochs. The function creates a directory, saves the generated samples to this directory, and returns the sparsity and Shannon's entropy metrics for the sample.

After defining the save function, we load our data into the main file. The format of the data is very important: data should only contain taxa abundance information, and it should be separated by each case. Therefore, there will be a separate generation dataset for each classification group. Furthermore, each row should correspond to one sample, while each column should correspond to one species. Once the data has been properly loaded into the main python file, we will use the *expand_phylo* function from *utils.py* to expand taxa to a higher order. This function returns a matrix of phylogenetic relationships in the data along with a dictionary of indices for each taxa. We use these outputs to create a dense matrix of phylogenetic relationships in our data, which is utilized further during the training process of the network.

After completing these brief preprocessing steps, we must specify hyperparameters for our model and its training process. We do so by defining two dictionaries: one named *model_config* and one name *train_config*. *Model_config* contains four entries: *ntaxa* specifies the number of taxa in our dataset, *latent_dim* specifies the size of the latent dimension from which we will generate our synthetic samples, *generator* specifies the number of channels for the hidden layers in the generator, and *critic* specifies the number of channels in the critic hidden layers, the dropout rate of the critic, the matrix containing phylogenetic information, and the weighting scale for critic calculations. *Train_config* contains two entries: *generator* specifies the generator's optimizer and its learning rate, while *critic* specifies the critic's loss weights,

optimizer, and learning rate. Recommended parameter values are found in the *mbgan_test* jupyter notebook.

After creating the dictionaries which specify the GAN's architecture for our data, we can now instantiate the model. We do so by calling the MBGAN class (from *MBGAN.py*) and inputting our model's name and its training and configuration dictionaries. After this step, we can now train the model. We have to specify several arguments, including the input data, epochs, batch size, save function, directory to export to, and save interval. Once these arguments are specified, the model should function correctly.

Known Code Bugs/Future Code Directions

Troubleshooting neural networks often requires a deep understanding of the mathematics behind the model. In order to understand why our loss is higher than we would like or why the model fails to converge, we need to have background knowledge of the computations occurring in the network. Because of the ambiguous nature of neural network tuning, fixing bugs in the model can be difficult.

The largest issue we have faced in implementing the MBGAN is receiving NaN (not a number) loss values during training. If the network's loss is NaN, the training just didn't work, so its structure must be fixed. Unfortunately, the reasons for NaN loss can be broad. Some examples are exploding gradients, vanishing gradients, or code errors.

After examining the MBGAN network's architecture, the problem appears to come from one layer: the RandomWeightedAverage layer. The RandomWeightedAverage layer is constructed to inherit from the _Merge class in Keras, which is now deprecated. This means that the layer must be constructed to inherit from a different class, such as the Layer, Average, or Concatenate class. Unfortunately, when inheriting from these classes, the recommended sample

size of four dimensions produces an incompatible input shape for the rest of the network. When lowering the sample size to become compatible with the rest of the network, It appears that problems in loss calculations occur, inducing NaN loss values. However, we are not fully confident that the error stems from this layer; while there is a high likelihood, it is also worthwhile to examine other possible causes and solutions to this problem.