Table of Contents

0. Executive Summary

Composite materials consist of various components that are combined to form a novel material possessing specific characteristics. The use of composite materials is on the rise in contemporary engineering and design because of the ability to customize microstructures to achieve particular material properties. However, the development of composite materials is a time-intensive and expensive process. In the last few years, materials modeling and machine learning have contributed to speeding up materials development. However, some of these frameworks do not ensure the manufacturability of their generated materials. To enhance the likelihood of creating materials that can be manufactured, one approach is to make minor adjustments to existing microstructures that can result in better chemical and physical properties. In order to improve the physical and chemical properties of materials, it is necessary to examine the composition and structure of the material to determine which attributes in the microstructures will lead to different properties. However, analyzing these microstructures is time-consuming and computationally expensive due to their complexity. Instead, we can simplify the microstructure to reduce its complexity, analyze the simplified version, select the simplified microstructure that possesses the desired properties, and then reconstruct the original version from the simplified one. Our team used deep learning models to simplify the representation of complex microstructure data. These models can analyze microstructure images and learn how to represent them in a simpler manner while retaining their important characteristics. Additionally, these models can also reconstruct the original microstructure from the simplified representation. We were able to reduce the complexity of microstructure images by more than 99%, while still producing high-quality representations that were very similar to the original data. Our proposed solution has the capability to compress 2-dimensional microstructure images to 0.16% of their original size, while still producing high-quality microstructure samples that are statistically similar to the original microstructures.

1. Introduction

A composite material is a material that is produced from two or more constituent materials. Composite materials combine the properties of their constituent materials to surpass the technical specifications of either material. However, the development of composite materials is a time intensive and expensive process. In recent years, materials modeling and machine learning have aided progress towards the development of computational workflows that have the potential to significantly accelerate the materials development process. Be that as it may, many of these frameworks do not consider the manufacturability of their models. Hence, there has been a need for frameworks that allow fast and inexpensive development of manufacturable composite materials.

In an Argonne National Laboratory (ANL) funded project, our client Dr. Noah H. Paulson and his team are developing a framework for the development of new composite materials that are also manufacturable. Dr. Paulson and his team are working on ensuring the manufacturability of new composites by reverse engineering the new materials. This reverse engineering process, better known as inverse design, consists of two main phases. Phase 1 starts by tweaking (optimizing) the material to improve the material, and then phase 2 tests the manufacturability of the tweaked material with computerized simulations. However, this inverse design process is bottlenecked by the computational complexity involved in phase 1. Therefore, our team will address this bottleneck by employing a framework that leverages artificial intelligence to reduce the computational complexity involved in phase 1 of the inverse design. The bottleneck in phase 1 is caused by the inefficiency of the machines that do the tweaking of composite data. These machines run more efficiently with data that is less complex than the current data at hand. One way to make the data less complex is by reducing its dimensionality.

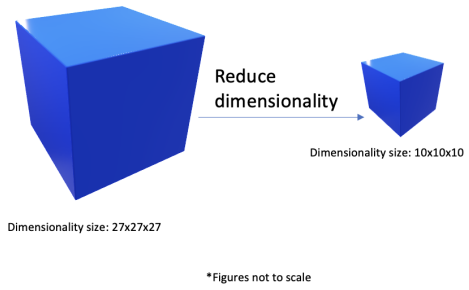


Figure 1: Dimensionality reduction

Reducing the dimensionality of data will help make the data less complex and will help machines run optimizations more efficiently on the new composite structure with smaller size. Refer to the appendix for more information on dimensionality reduction. It will also alleviate the bottleneck of phase 1 in the inverse design process. However, phase 2 of inverse design involves testing the manufacturability of materials with simulators. These simulators take data with the original (non-reduced) form. This means that we will have to revert the data from its reduced form back to its original form. Furthermore, we have to make sure when we recover the original form of the data, the data will look as identical as possible to its true original form. The figure below demonstrates a possible side-effect of recovering the original data.

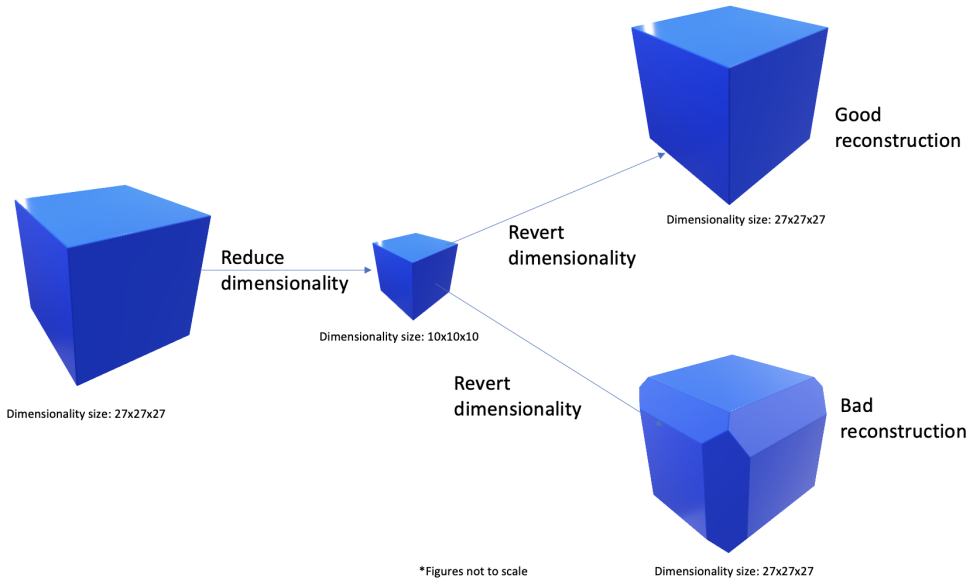


Figure 1.2: Dimensionality reduction and reconstruction

Thus, there is a need for an algorithm that transforms an input microstructure image to a lower dimensional space, also known as the latent space, to create a representation of the material, and then uses this representation to generate new microstructure examples. It is important to mention that the scope of this project was defined for implementing dimensionality reduction and reconstruction for 3-dimensional samples. However, given the difficult nature of the problem that we are trying to solve, our team has been able to successfully implement dimensionality reduction and reconstruction for 2-dimensional microstructure as of now. We will be discussing our plan for implementing a model that performs dimensionality reduction and reconstruction on 3-dimensional data in section 8: Next Steps.

The upcoming section outlines the functional requirements of this project, including the necessary functions, constraints, and other detailed requirements for the proposed design solution. Following this, we detail our methodology, which involves exploring several feasible alternative design solutions, each with their own specifications. We test each of these design alternatives and evaluate how well they meet the requirements of this project. To conclude the report, we outline the shortcomings of each design alternative and provide potential causes for these issues. Finally, we propose next steps for the project based on our findings.

2. Requirements

To create a design that performs dimensionality reduction and reconstruction on microstructure diagrams, the following section will outline the functions, constraints, and other detailed requirements of a proposed design solution.

2.1 Functions

A function describes the purpose of the proposed design and what it is supposed to accomplish. Table 2.1 outlines the primary functions of an effective solution, as well as lists the secondary functions which describe how the primary functions will be executed.

Table 2.1 - Primary and secondary functions for any proposed design solution.

Primary Function	Secondary Functions
To create a lower-dimensional representation of an inputted 3-D microstructure image comprised of voxelized data	Read voxelized microstructure data
	Reduce data to lower-dimensional space
	Output a representation of inputted microstructure in lower-dimensional space
To use a lower-dimensional representation of a microstructure to create new microstructure examples of the same composite material	Read representation of microstructure in lower-dimensional space
	Output voxelized data of similar microstructure

2.2 Objectives

An objective elaborates what a proposed solution should accomplish, along with the relevant metrics for measuring success, while using the design's functions described in the previous section as its guiding principles. Table 2.2 illustrates these objectives, their metrics, and the criteria for success in order of descending importance.

Table 2.2 - Objectives, metrics for measurement, and the target KPI for judging success.

Objective	Metric	Target
Should maintain fidelity when reducing data	Spatial statistics	Spatial statistics for specific features of the generated data should lie within 6 sigma of the spatial statistics distribution of the same features of the inputted data.
	Mean Squared Error (MSE) Loss	Minimize MSE between designs
	Fréchet Inception Distance (FID Score)	Minimize FID score between deesigns
Should create a latent space with a reduced dimensionality than the original data	Latent space dimensions	Dimensionality of magnitude to the order of 10
Should reduce and restore dimensionality in a timely manner	Time complexity of the reduction and output generation operation	Quadratic time complexity or better
Should have traceability between software requirements, and software architecture and design	Each requirement is linked to one or more software architecture design choices and implementations	100% of requirements are attributed to one or more software architecture design choices

2.3 Constraints

Table 2.3 outlines the constraints that a proposed design solution must meet to be eligible for consideration. These constraints are specific to the design being an algorithmic tool used within the existing inverse-design framework the client has for the generation of next-generation composite materials. Therefore, these constraints primarily originate from the client, including applicable standards.

Table 2.3 - Constraints, their reasoning(s), and the associated limit.

Constraint	Reason for Inclusion
Must generate both a latent space in a reduced dimensional space, as well as use this latent space to generate and reconstruct microstructure diagrams.	Some frameworks provide a latent space, and some frameworks provide a generated output. The design the team delivers must provide both.
Capable of handling 2-D microstructure diagrams as input, with possible framework extension to 3-D microstructures diagrams.	The final product must be able to work with both 2-D and 3-D data
Limited to relatively small dataset of 3D microstructure diagram	The team is working with a dataset that is at approximately 1,000 to 10,000 data points large. So the design must incorporate a framework that is data-efficient.
Limited computational resources on local machine	The team is given data that is of dimensions 21x21x21, so the designed algorithms must be able to run on local machines.

2.4 Service Environment

The following section discusses the physical and virtual environment in which the design will operate. Living things and environment are not considered because it is a software design used on a computing machine. The detailed specifications of the environment in which the design will be used is listed below in Table 2.4:

Table 2.4 - Service Environment Specifications

Type of Service Environment	Service	Reason for Inclusion	Specifications
Physical Environment	Indoor weather and conditions	The design is meant to be used on a	No specification relevant to indoor

		computer. As such, the usability characteristics of our design is considered.	conditions is required. It is just required that the computer machine is able to run.
	Computer or computational machine		Multi-core CPU processor, 8GB+ RAM is required.
Virtual Environment	Python and Relevant Libraries	The design will be created using Python, so it is important to consider the language version as well as the version of all relevant libraries that may be used.	Python 3.6+, all other up-to-date libraries.

2.5 Stakeholder Analysis

In Table 2.5 below is a list including the entities affected by this design as an algorithm in a research laboratory. These stakeholders were determined based on how much they are or will be influenced by the design over the design life-cycle.

Table 2.5 - Impact by/on Stakeholder for Design - Stakeholder Analysis

Stakeholder	Impact by/on Stakeholder
Argonne National Laboratory (ANL)	The design will be used within a department at this research institution. Any progress and achievement within a team at ANL may result in more funding and better recognition for the critical work done at ANL.
Dr. Noah Paulson's Team at ANL	The design will be used within our client's existing framework for the inverse design of next-generation composite materials. The design will provide Dr. Paulson's team with a new and more efficient way to perform optimization computations to determine microstructure properties.
US Department of Energy	The design will be used within a team at ANL, which is a research center operated by the US Department of Energy. Therefore, the US Department has a stake and will benefit from any work that is done which presents

	progress in research relevant to the Department.
UChicago Argonne LLC	The design will be implemented on a team that consists of collaborators from the University of Chicago. The UChicago Argonne LLC is a joint program between the university and ANL to collaborate on state-of-the-art research. Therefore, since our design team is working collaboratively with members of the UChicago Argonne LLC program, any progress or achievement from the proposed design will also impact the members of the UChicago Argonne LLC at ANL.

2.6 Design for Excellence

The following section outlines the methodologies and practices that will be utilized to create a deliverable that incorporates the Design for Excellence (DFx) frameworks. Each principle along with its technique for incorporation will help the team increase the value of its design by meeting defined requirements and optimizing the design development cycle. Therefore, the specific DFx principles are listed below in Table 2.6, along with the reason for inclusion of each principle and the technique of how each principle will be incorporated into the development cycle.

Table 2.6 - DFx Principles for Design

Type of DFx Principle	Reason for Inclusion of DFx	Technique for DFx Principle
Design for Cost	This principle aims to optimize the overall cost of the design and its lifecycle processes. The team intends to incorporate this DFx principle to generate designs that limit the cost and time to completion, as well as optimize the computational cost associated with the design.	- Developing alternative designs and choosing winner through comparative models - Simplification and modularization of code
Design for Innovation	This principle aims to incorporate new ideas in a design to drive solutions. The team intends to incorporate this DFx principle to generate designs that add value and	- Start with multiple options when exploring technologies and explore alternative ways to architect the product - Competitive research -

	enhance the capabilities of pre-existing technologies.	understand how similar products are defined, designed, manufactured
Design for Quality	This principle aims to add quality checks within the product development cycle which measure and evaluate key metrics to deliver a good quality design. The team intends to incorporate this DFx principle by measuring and comparing key quality-related metrics between designs.	- Quality checks that measure “performance”, “durability”, and “cost”
Design for Serviceability	This principle focuses on making the design easy to maintain. The team intends to incorporate this DFx by measuring the efficiency of maintaining the full functionality of the design.	- Use OOP principles in writing software - Include logging, assertion statements, and catch statements to ensure errors are correctly identified and quickly addressed
Design for Reliability	This principle aims to build reliability into designs to determine the ability of the system to perform its functions over a period of time. The team intends to incorporate this DFx to ensure designs can handle an acceptable level of throughout data over a sustained period of time.	- Incorporate methodologies to identify, assess and reduce the risk of product failures - Test on appropriate level of throughput data, with variations to ensure proper error handling is incorporated into software

3. Problem Solving & Research Methodology

As a research-intensive project, the team followed an ad hoc design implementation strategy which involved three phases; research, implementation, and iteration. The research stage involved researching various machine learning models that would fit the problem and their associated limitations with respect to the problem. The implementation stage involved implementing the model in code, as well as conducting all the tests necessary with

non-microstructure data, such as the CIFAR-10 dataset. Finally, the iteration stage involved improving the model and altering it to work with 2D microstructure data.

4. Methods

To develop candidate design solutions, the team explored implementations of state-of-the-art generative deep-learning based frameworks which achieve all the objectives and satisfy the constraints set in Section 2.3. Feasibility and robustness of frameworks were then later determined by comparing input data from the dataset and the output microstructure diagrams.

4.1 Data Processing

Our 2-dimensional models were trained using a dataset of raw 200x200-pixel inorganic composite microstructure images, comprising of 60 different classes, each with 12 instances. Therefore, our initial dataset only had 720 images. As such, the team had to apply multiple rounds of data preprocessing techniques to allow this data to be usable and sufficient for neural network training. To expand this dataset, the team applied several transformations such as rotations, flips, and random blurring, effectively increasing the dataset size by a factor of 8. In addition, to standardize the images, the team ensured that each image contained three color channels, as some classes had only one. The team then labeled each image with its corresponding class and created a PyTorch Dataset object. Finally, we transformed the dataset object into a PyTorch DataLoader object, which is ready to be used as input for our PyTorch deep learning models.

4.2 Models

Within each deep-learning based framework that was chosen, one consistency was the use of convolutional layers in the models. Convolutional layers are a fundamental building block in convolutional neural networks (CNNs), which are a type of deep learning model widely used for image and video analysis tasks. A convolutional layer applies a set of filters (also called kernels or weights) to the input image or feature map, producing a set of output feature maps. Each filter slides over the input image, performing a dot product operation between its weights and a local patch of the image at each position, and generating a single value in the output

feature map. The output feature map is typically smaller than the input image, as the filter's receptive field "sees" only a portion of the input at each position. The advantage of using convolutional layers is that they can capture local patterns and spatial dependencies in microstructure images, while sharing weights across different regions, which reduces the number of parameters and makes the model more efficient. Additionally, by stacking multiple convolutional layers on top of each other, the model can learn increasingly complex and abstract features from microstructure images.

4.2.1 Deep Convolutional Variational Autoencoder (C-VAE)

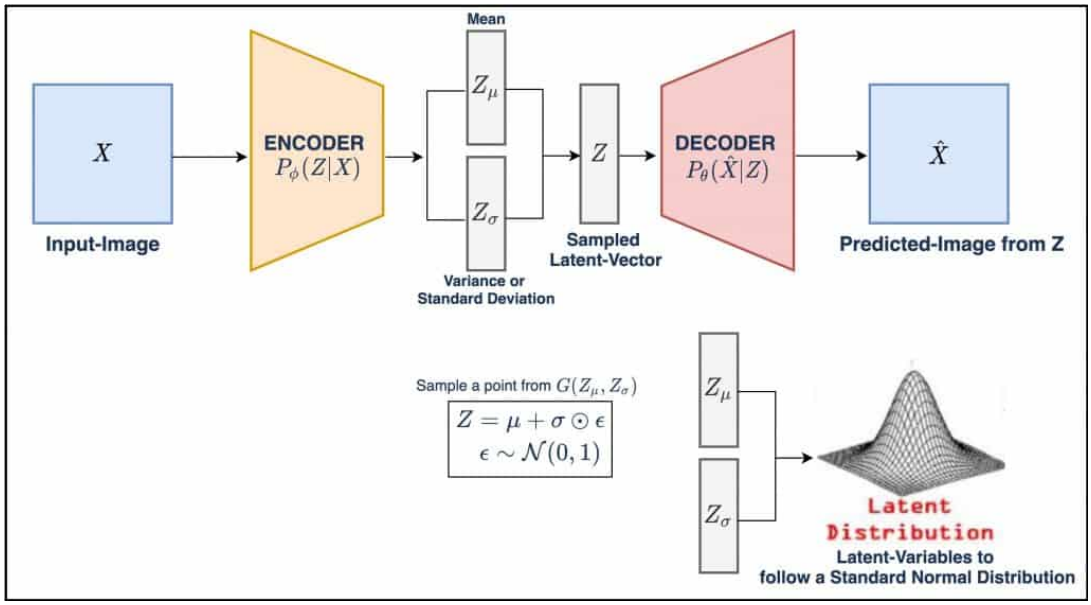


Figure 4.2.1: Deep convolutional variational autoencoder diagram. Note that the encoder and decoder are convolutional layers. [1].

The first feasible framework is Convolutional VAEs, which are a subcategory of Variational Autoencoders (VAEs) that consist of convolutional layers. They are a type of neural network that can learn to represent complex data in a compact and structured way. It does this by training a model to encode high-dimensional input data, such as images, into a lower-dimensional latent space, and then decoding their latent representations back into the original input space. One key advantage of this framework is that it generates a continuous latent vector between the encoder and decoder components which can be used for interpolation and generation of new composite microstructures. Specific to the model that the team implemented, the C-VAE was able to learn a compressed and meaningful representation of the microstructures,

while ensuring that the latent representation of the microstructures maintains its probabilistic characteristics. This allowed us to generate new and diverse microstructures that preserved the statistical properties of the original data to some extent.

Although the initial raw dataset was expanded using numerous preprocessing techniques mentioned in Section 4.1, additional steps needed to be taken to achieve meaningful results. Due to the overall limited data points, it was simply not enough to train a C-VAE framework from scratch to a point where it could show decent performance. As such, one key technique which was also applied to all other frameworks later on was the use of transfer learning. Transfer learning is a technique where the weights of a pre-trained model are used as starting points for a new, related task. Instead of training the model on microstructure data from scratch, the team used the weights of a model trained on a larger, similar dataset - ImageNet - to be the starting point instead of starting for scratch. The ImageNet dataset consists of more than 14 million images, labeled into numerous classes of both animate and inanimate objects found in real life. The team hypothesized that the skills and representations that a model could learn from the ImageNet dataset may be helpful and in somewhat of a similar task to learning a representation of microstructure data. This hypothesis proved to be true, and transfer learning proved to be a worthwhile technique to boost model performance. As such, this technique was applied on all subsequent frameworks as well, as the dataset of just microstructure data proved to be insufficient when trying to train a neural network model from scratch with it. Because we included the pretrained resnet-152 model, the architecture is rather deep. Hence, we placed it in the appendix B section.

4.2.2 Deep Convolutional Generative Adversarial Network (DCGAN)

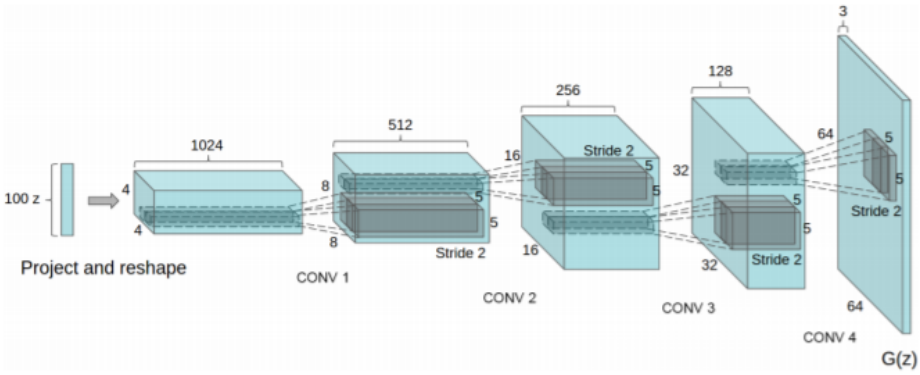


Figure 4.2.2.1: DCGAN generator diagram. [2]

Figure 4.2.2.2: DCGAN discriminator diagram. [2]

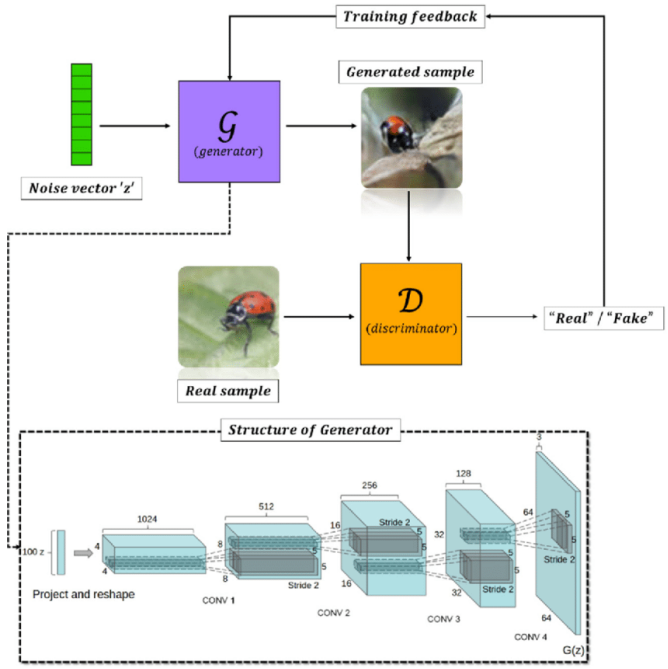


Figure 4.2.2.3: DCGAN diagram. [3]

The second feasible design framework is a DCGAN, which is a subcategory of Generative Adversarial Networks (GANs) which consist of deep convolutional layers in addition to the base neural network layers. DCGANs consist of two components - the generator and the discriminator. The generator component is responsible for generating new data samples given inputs, while the discriminator tries to distinguish whether the samples that the generator has created are real or fake given both the real and fake images that were generated. As the generator improves at creating more realistic samples that fool the discriminator, the discriminator gets

better at distinguishing real from fake data. The ultimate goal of the GAN framework is to generate data that is so realistic that it becomes indistinguishable from real data.

In our project, we employed convolutional-based GANs to learn a latent representation of microstructures. As the model was fed real 2-dimensional microstructure data, the generator generates fake 2-dimensional microstructure images, which are then fed to the discriminator along with the real microstructure images. The discriminator tries to correctly classify the images as real and fake, and its loss is backpropagated to update its weights. At the same time, the generator's loss is based on how well it fools the discriminator, and its weights are updated based on this loss. This allowed us to create highly realistic-looking microstructures, as the generator and discriminator learn to work together to produce high-quality, realistic images. Below is the GAN architecture that we trained.

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

Figure 4.2.2.4: Generator PyTorch architecture

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

Figure 4.2.2.5: Discriminator PyTorch architecture

4.2.3 Latent Diffusion Models (LDM)

Lastly, the third feasible design framework is the Stable Latent Diffusion Model (LDM) [4], a subcategory of Diffusion Models (DMs) which take in latent representations as input data instead of the original image when training the diffusion model. This framework consists of two trainable components: 1) a KL-based Autoencoder, and 2) U-Net diffusion model. LDMs function in a way where first, the autoencoder component is trained using the dataset so that it has learned a dynamic latent representation for the input data. Next, the diffusion model first uses this learned autoencoder to create an encoding for the input data it receives. Then, it performs the process of diffusion on the encoded latent representation of the input data instead of the original image itself. This is the key distinction between a base DM and an LDM. The architecture of the LDM framework that was used for this design can be seen below in Figure 4.2.3, where the code and detailed model architecture can be found on the github repository.

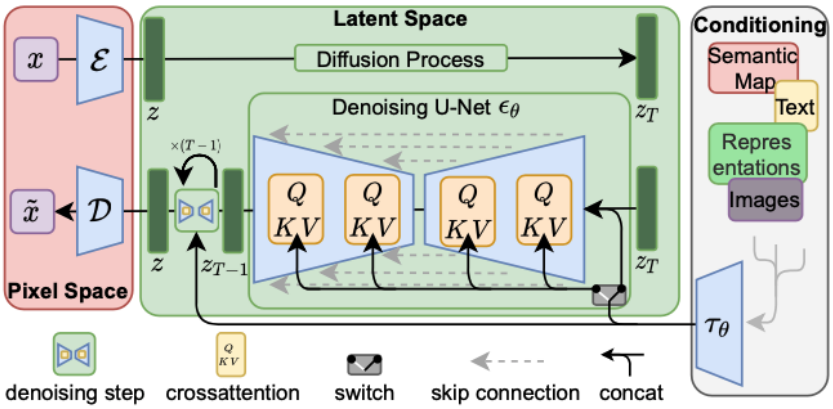


Figure 4.2.3: Architecture of LDM Framework [4]

Diffusion models are a type of generative model that learn to model the probability distribution of a dataset. Whereas typically these models operate in the pixel space, DMs can require large amounts of compute resources. To optimize the enablement of diffusion models on limited computational resources, the model can instead be applied in the latent space where the input data is first encoded and then the process of diffusion is applied on the encoded latent vector. Diffusion models learn by iteratively applying a process called diffusion to the input data. In specific to this design framework, the diffusion process is applied in the latent space and works by adding a small amount of noise over time. During training, the diffusion model learns to predict the original encoded microstructure data from the noisy encoded microstructure input.

This diffusion model consists of a set of non-linear transformations that map the input latent data to a further reduced latent space, and then back into the output space. These transformations are learned during training by optimizing the likelihood of the data under the model.

5. Results

This section presents a detailed analysis of the results obtained from each design solution, which includes a comparison between the original input data and the generated microstructure samples. Additionally, we provide information on the latent dimensionality of the encoded microstructure representation, as well as metrics on the similarity between input and output images using spatial statistics.

Spatial statistics is a branch of statistics that focuses on analyzing and modeling spatially correlated data. This technique is commonly used in material science to analyze and model spatially correlated data such as the microstructure of materials. By analyzing the spatial distribution and connectivity of different phases, grains, pores, and other microstructural features in materials, researchers can better understand the mechanical, electrical, thermal, and other properties of materials, as the microstructure can greatly affect these properties [6].

To ensure that our models produce microstructures that are statistically similar, we compare the mean absolute error (MAE) of the spatial statistics of original images against the reconstructed images. Additionally, we have visualized the encoded latent vectors for the three models to show that the latent representations of different classes of microstructures differ from each other. Furthermore, we have included the input image that was encoded into the latent representation, as well as the reconstructed image that was decoded from the latent space using the latent vector.

We also report the dimensionality of the latent space for each model. The smaller the dimensionality is, the better it would be for the task of optimization. However, we should also note that reducing the dimensionality of the latent space can result in the degradation of the model performance.

One of the main applications of our project is to produce diverse and novel microstructures that are not present in the training data. This is achieved through latent space interpolation, which involves selecting two or more latent vectors that correspond to different microstructures and linearly interpolating between them to create new latent vectors. These new

latent vectors can then be decoded by the generative model to generate new microstructure images that are intermediate between the original microstructures.

For each model, we provide a visualization of linearly interpolating between two preselected microstructures to highlight the models' ability in producing diverse and novel microstructures that visually looks like a combination of the two pre-selected microstructures. A successful latent space interpolation shows that the model has learned the underlying distribution of the data and can generate new data by modeling the underlying distribution. Conversely, an unsuccessful interpolation shows that the model has memorized the training data but has not learned anything about the distribution of the data.

5.1 C-VAE Results

During the training of the convolutional variational autoencoder, we achieved results that closely resemble the images of the chemical inorganic composite microstructures. This model performed well with a latent space dimensionality of 256. To illustrate the capability of the model, we present an example where a given image is encoded into a latent vector and then reconstructed back from this latent vector. It is important to note that the model struggles with reconstructing high quality images, which will be discussed further in the limitations section.

5.1.1 Reconstruction Results for C-VAE

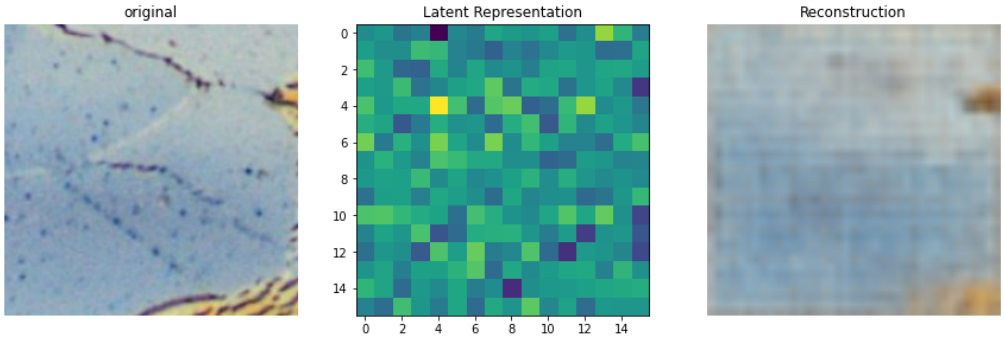


Figure 5.1.1.1: Reconstruction of sample from microstructure class 1

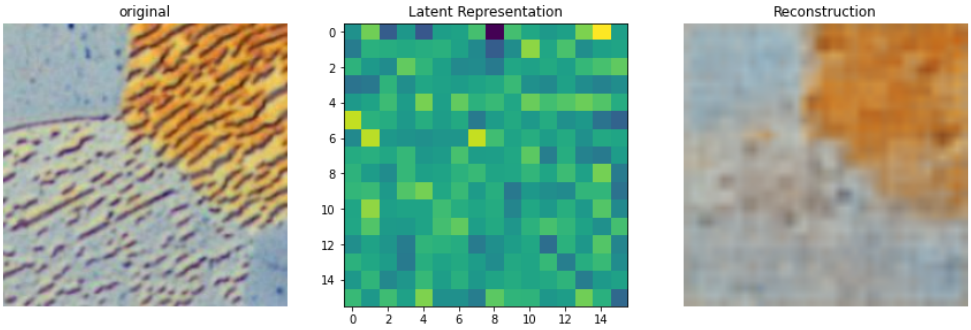


Figure 5.1.1.2: Reconstruction of sample from microstructure class 1

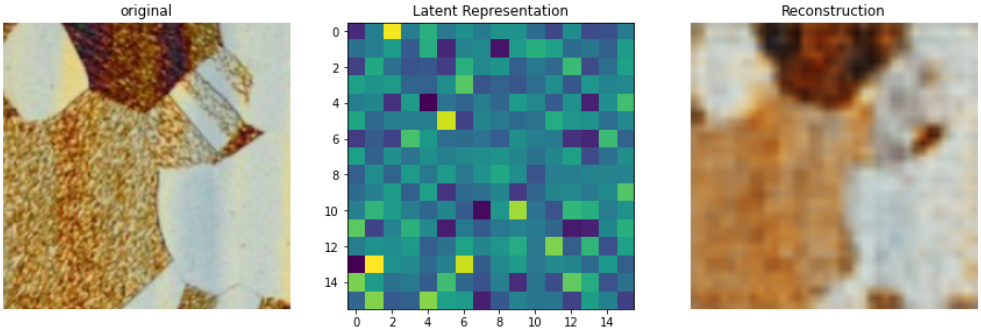


Figure 5.1.1.3: Reconstruction of sample from microstructure class 2

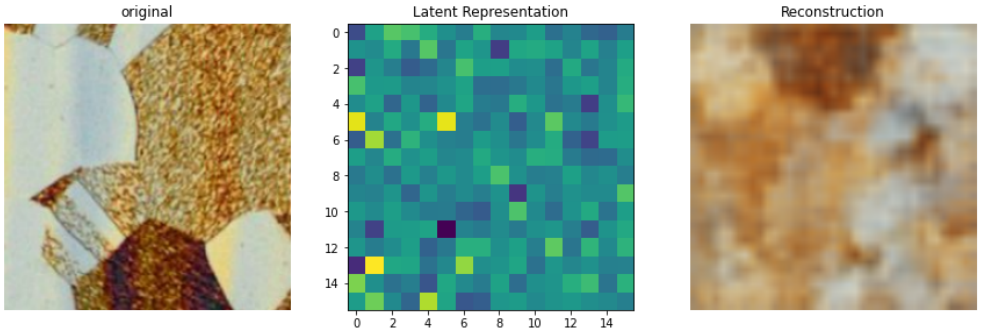


Figure 5.1.1.4: Reconstruction of sample from microstructure class 2

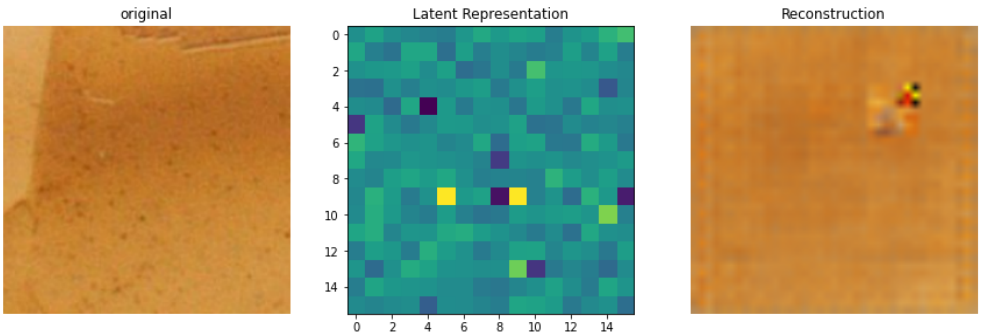


Figure 5.1.1.5: Reconstruction of sample from microstructure class 4

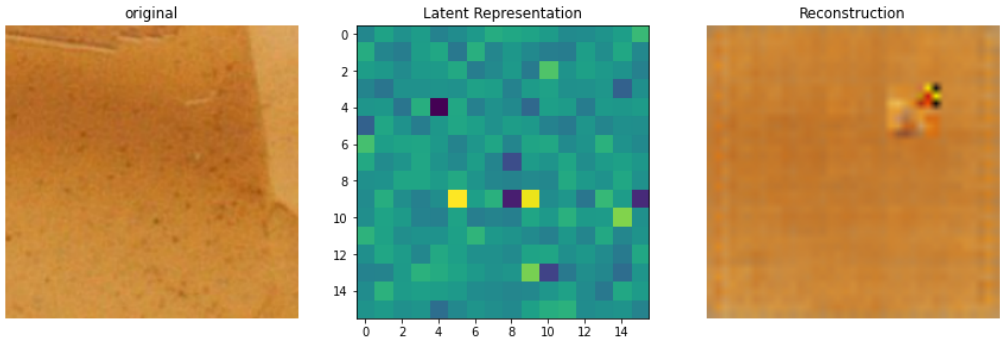


Figure 5.1.1.6: Reconstruction of sample from microstructure class 4

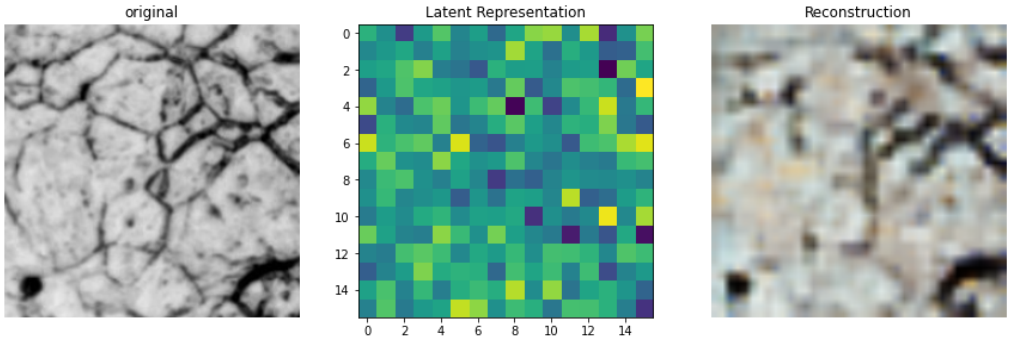


Figure 5.1.1.7: Reconstruction of sample from microstructure class 7

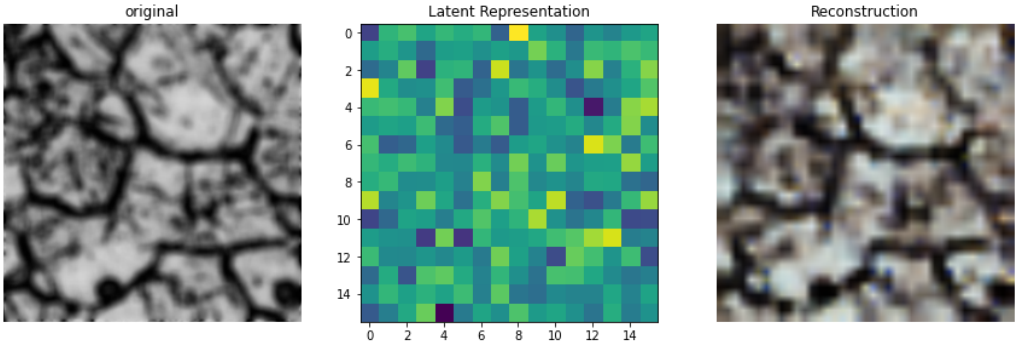


Figure 5.1.1.8: Reconstruction of sample from microstructure class 7

Based on the results obtained, it is evident that the model has the ability to generate samples that resemble the original images. A noteworthy observation is that the same reconstruction is produced for rotations of the same image. This suggests that the model has learned that an image and its rotation are the same, as can be seen in Figures 5.1.1.1 and 5.1.1.2. However, it is clear that the model is not capable of producing high-quality results. In order to assess whether the

model has learned the underlying distribution of the materials, we carry out an interpolation in the latent space, which will be discussed in the next section.

5.1.2 Interpolation Results for C-VAE

Below, we present interpolations between two selected images that belong to the same class but have different appearances. The purpose of this is to evaluate whether the model can generate distinct variations of the same material class. The hypothesis is that if the model has learned the underlying distribution of the material class, it will be able to produce samples that do not exactly resemble the two provided samples; rather, the produced samples should look like variations of the same material from that class. To perform interpolation, we first encode two original samples from a given class into the VAE latent space. Next, we linearly interpolate between the two points in the latent space and select 8 equidistant points from the interpolation. Finally, we feed the interpolated points into the model decoder to reconstruct microstructure images.

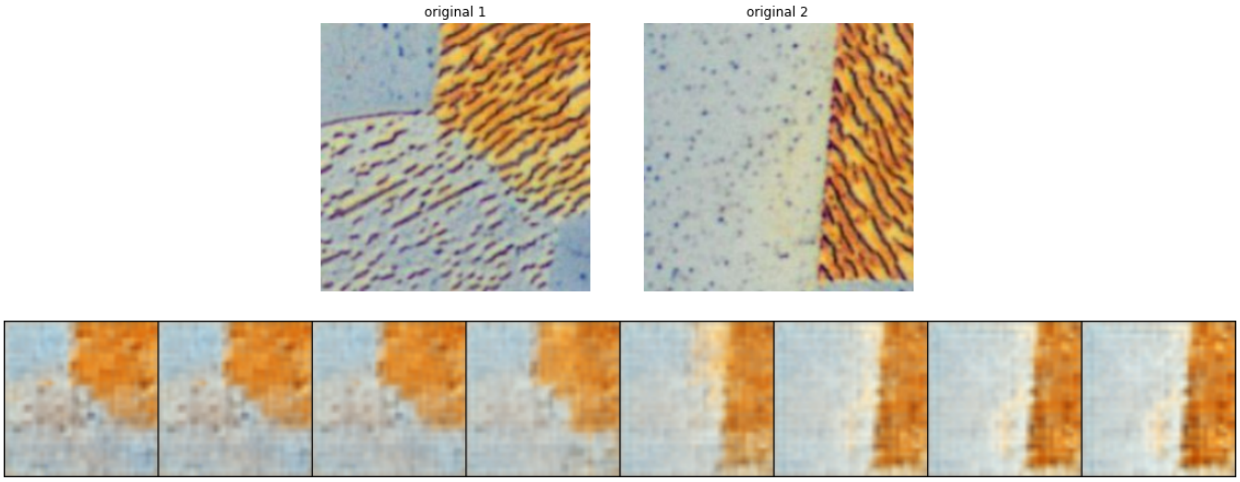
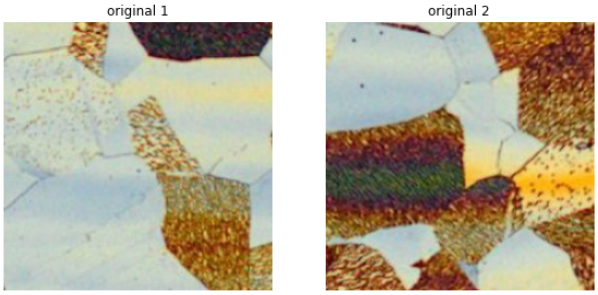


Figure 5.1.2.1: Interpolation of two samples from microstructure class 1



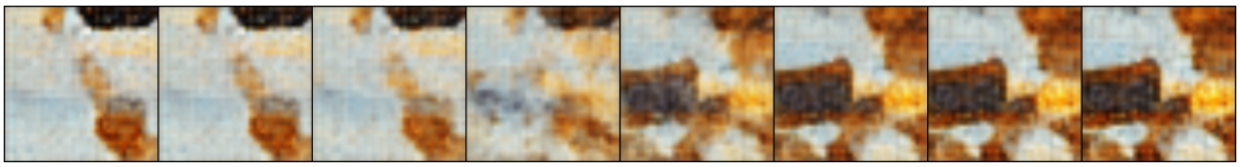


Figure 5.1.2.2: Interpolation of two samples from microstructure class 2

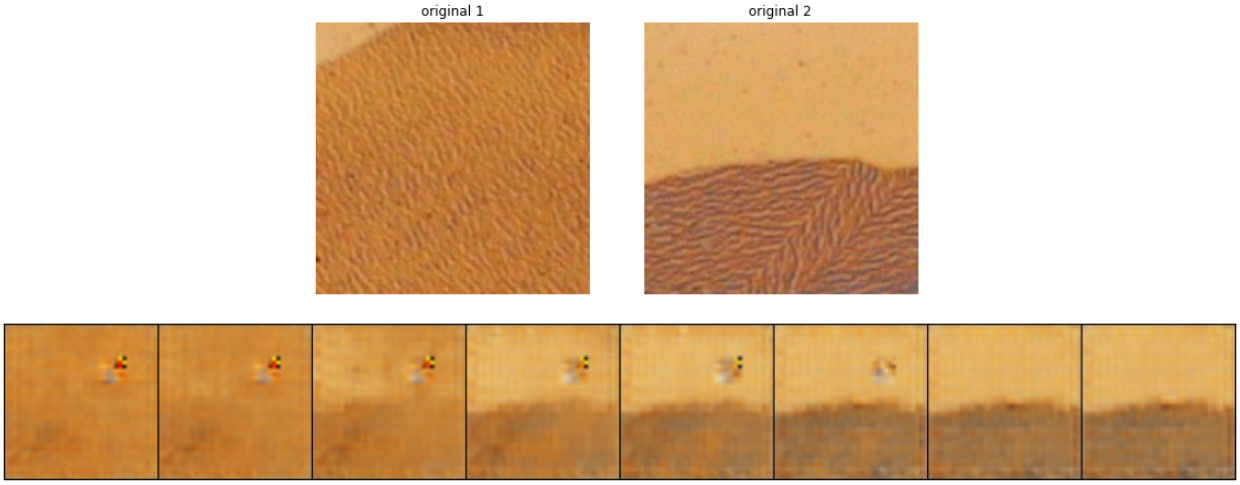


Figure 5.1.2.3: Interpolation of two samples from microstructure class 4

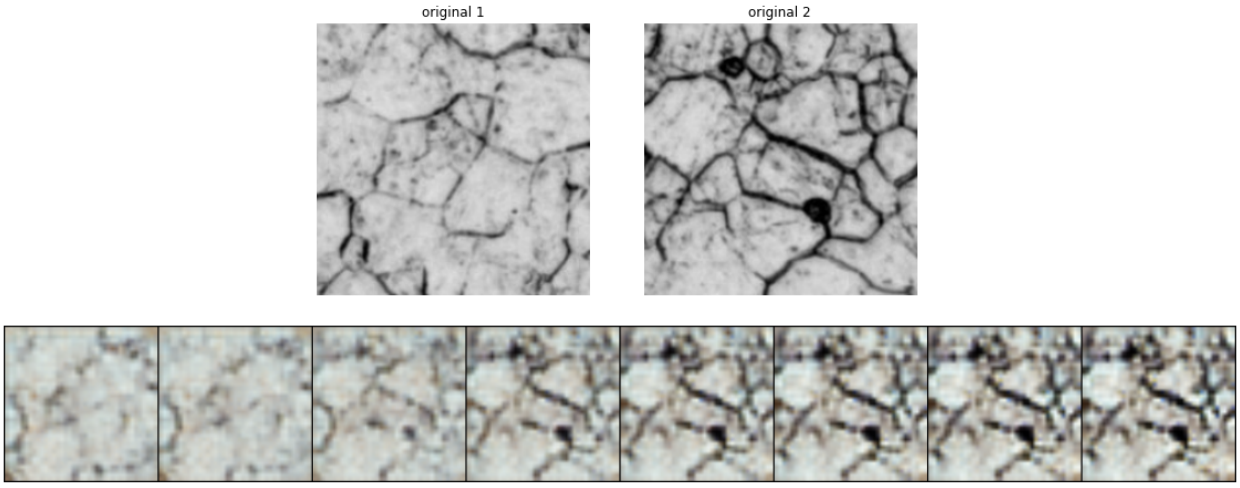


Figure 5.1.2.4: Interpolation of two samples from microstructure class 7

The primary goal of VAE is to generate diverse images, not merely reconstruct the same image. However, it seems that the model is struggling with robust interpolation as the generated images are nearly identical to the original data. This raises concerns that the model has overfit to the training data and lacks useful generalizations regarding the microstructures. To address this

issue, we need to modify the model's architecture and its training process. One potential solution is to reduce the KL regularization term, also known as the beta term, to less than 1. Currently, it is set at 1, which could be limiting the model's ability to generate diverse images. Additionally, we should consider adding more augmented data and increasing the size of the training dataset to provide the model with a more extensive range of data to learn from. Another approach is to simplify the features used by the model. One option is to preselect image classes that are mostly grayscale, rather than mixing color and black-and-white classes together. This can help to reduce the complexity of the model and make it easier to generate diverse images. In summary, to enhance the performance of the VAE model, we need to modify its architecture, training process, and the data it learns from. By doing so, we can help the model generate a broader range of diverse images that better generalize to new microstructures.

5.1.3 Original vs Reconstructed Spatial Statistics for C-VAE

We compared the spatial statistics of the original samples with those of the reconstructed samples. When we calculate spatial statistics on a tensor, it results in another tensor of the same shape. To get a scalar metric for the spatial statistic tensor, we calculated its mean. We have plotted a bar plot below that shows the average of the mean values of the spatial statistics of 101 original samples against that of their corresponding reconstructions.

Note that the height of the blue bars represents the average of the means of spatial statistics of each original sample, and the height of the orange bars represents the average of the means of spatial statistics of each reconstruction of a given sample. The error bars represent the standard deviation of the mean of spatial statistic tensors.

Inferencing similarity of means from a barplot with error bars can be done by visually inspecting the overlap of the error bars between the two groups being compared. If the error bars overlap significantly, then it suggests that the means are not significantly different and the groups being compared are likely to have similar means. On the other hand, if the error bars do not overlap or overlap minimally, then it suggests that the means are significantly different and the groups being compared are likely to have different means.

From the plot, we can observe that we were only able to reconstruct statistically similar images for microstructure class 1, and the reconstructions of the other classes are not statistically similar to the original data. This could be due to the fact that the model is not capable of

producing high-quality reconstructions, and as a result, its performance is low on spatial statistic scores.

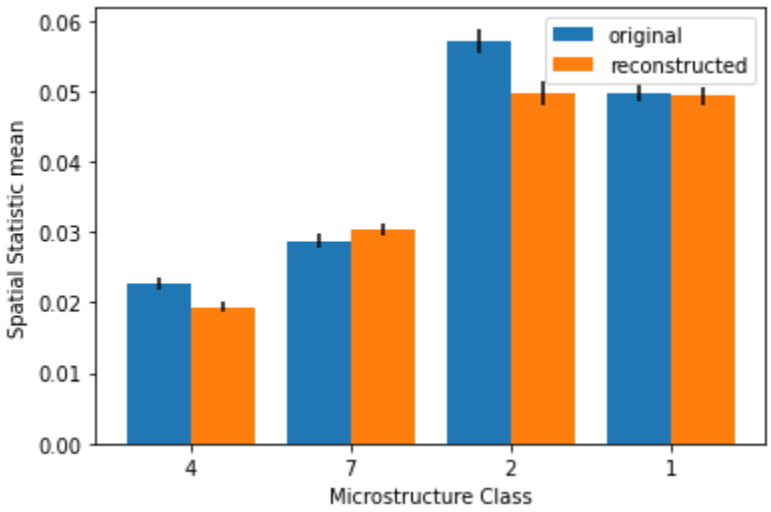


Figure 5.1.3: Average of the mean of spatial statistic of original samples compared to the average of the mean of spatial statistic of reconstructed samples for 4 classes of microstructures

5.2 DCGAN Results

After 16000 iterations of training, the DCGAN produces results that closely resemble the original images. The GAN is able to learn to generate samples from a 100 dimensional latent space. Note that this latent space is smaller than the latent space of the C-VAE. The dimensionality of this latent space is 1% of the dimensionality of the original data. Overall, the model performs better than the VAE in reconstructing the details of the original images for some classes. But, there are limitations that we will discuss in the limitations section. It is also important to note that a GAN does not have an encoder. Moreover, the architecture of our framework does not allow for the generation of samples conditioned on microstructure class. As a result, one can only generate arbitrary samples from the latent space. This can be done by decoding a random gaussian noise vector of size 100. In the visualizations below, we have resized the noise vectors of size 100 to vectors of size (10, 10) for better visualizations.

5.2.1 Reconstruction Results for DCGAN

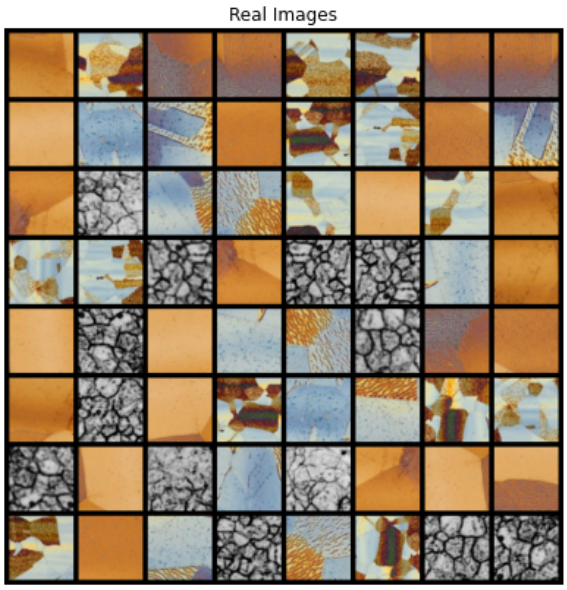


Figure 5.2.1.1: original microstructure images

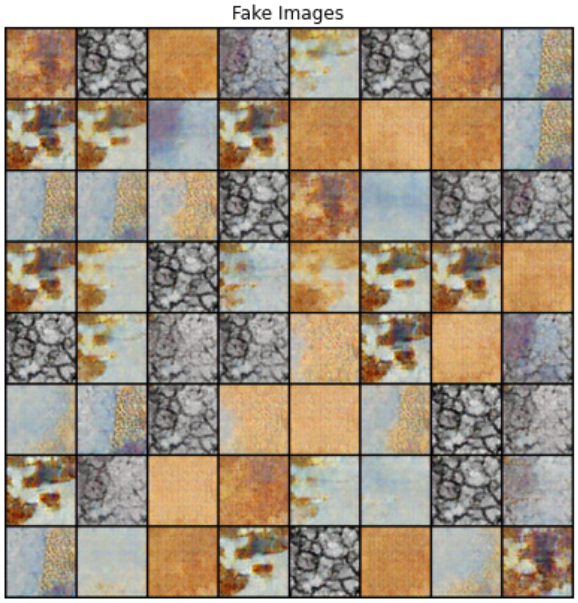


Figure 5.2.1.2: generated microstructure samples by the GAN

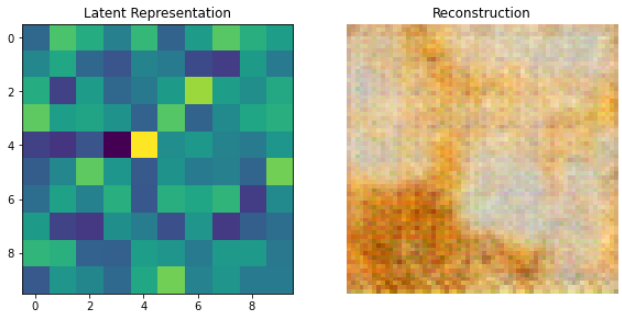


Figure 5.2.1.3: generated sample microstructure by the GAN from a given latent representation

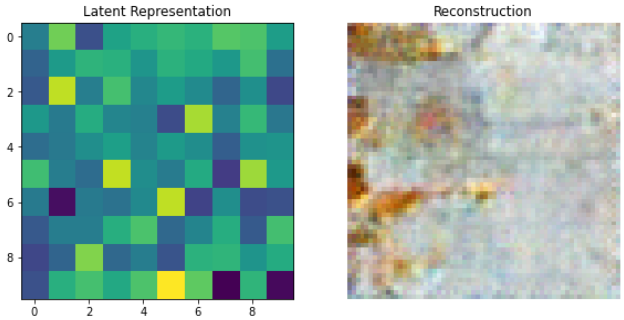


Figure 5.2.1.4: generated sample microstructure by the GAN from a given latent representation

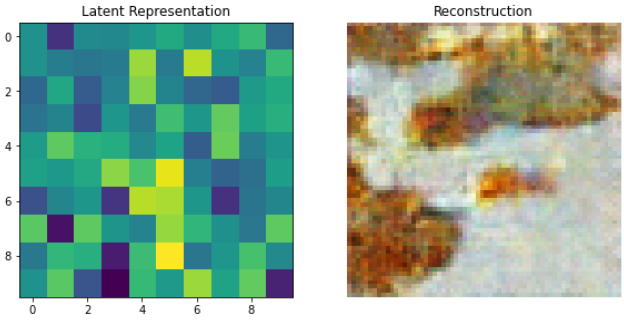


Figure 5.2.1.5: generated sample microstructure by the GAN from a given latent representation

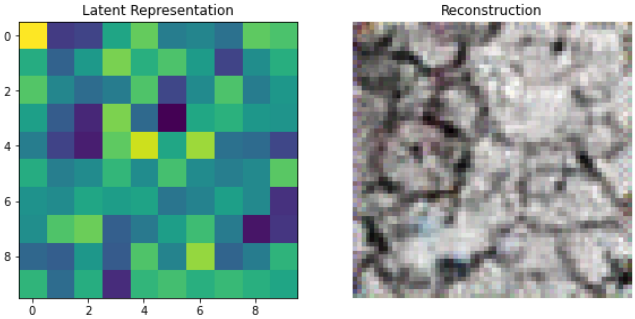


Figure 5.2.1.6: generated sample microstructure by the GAN from a given latent representation

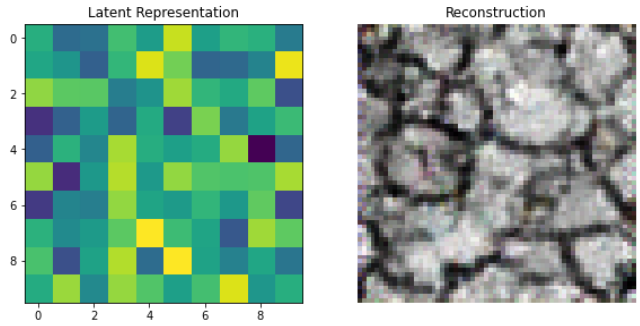


Figure 5.2.1.7: generated sample microstructure by the GAN from a given latent representation

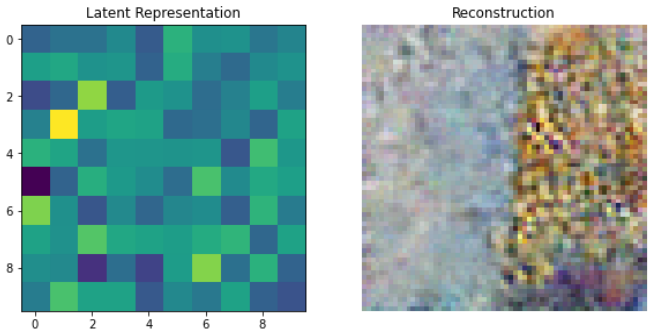


Figure 5.2.1.8: generated sample microstructure by the GAN from a given latent representation

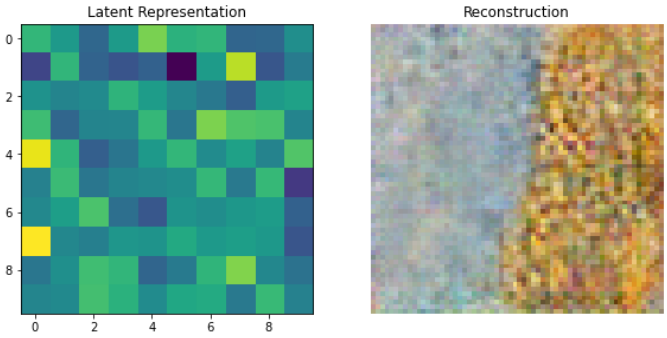


Figure 5.2.1.9: generated sample microstructure by the GAN from a given latent representation

Due to the absence of an encoder in the GAN architecture, generating samples requires decoding noise vectors with the generator. Unfortunately, this approach is not ideal since it does not allow for the encoding of microstructure images into the latent space.

5.2.2 Interpolation Results for DCGAN

In this section, we will present the results of linearly interpolating in the latent space of the GAN. It is important to note that unlike the VAE model, we cannot simply interpolate between any two given images since the GAN does not have an encoder. Therefore, to perform interpolation, we must select two arbitrary points that are known to be decoded to a microstructure of some form, and interpolate between the two.

To start, we generate two noise vectors of size 100 and decode them to see what their corresponding microstructures look like. If the two decoded results are the microstructures that we want to interpolate between, we linearly interpolate between their noise vectors in the latent space and select 8 equidistant points from the interpolation. Finally, we feed the interpolated points into the model decoder to reconstruct microstructure images.

It is worth noting that the quality of the generated images may vary depending on the chosen noise vectors and the GAN's architecture. Nonetheless, linear interpolation in the latent space of a GAN is a powerful tool that can be used to generate new images that lie between two given images in a meaningful way.

Below are interpolated images that lie between two samples of the same microstructure classes.

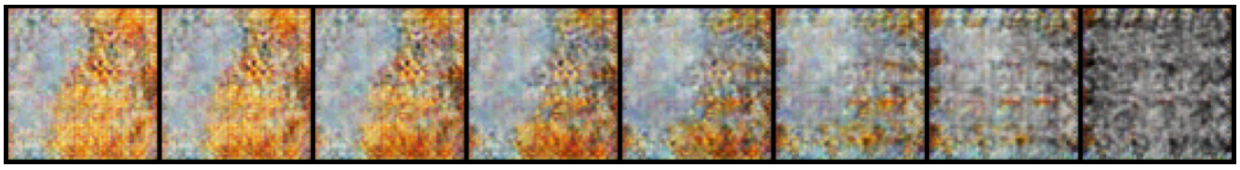


Figure 5.2.2.1: Interpolation of two samples from microstructure class 1

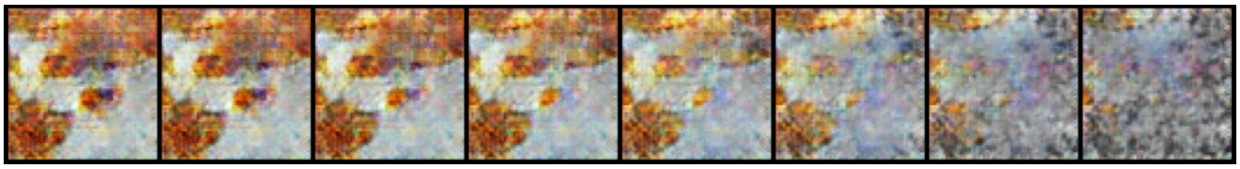


Figure 5.2.2.2: Interpolation of two samples from microstructure class 2

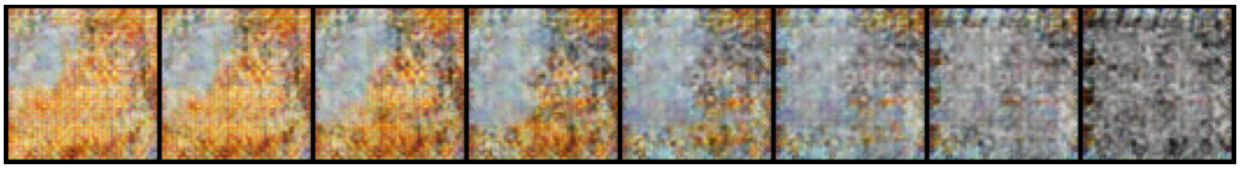


Figure 5.2.2.3: Interpolation of two samples from microstructure class 4

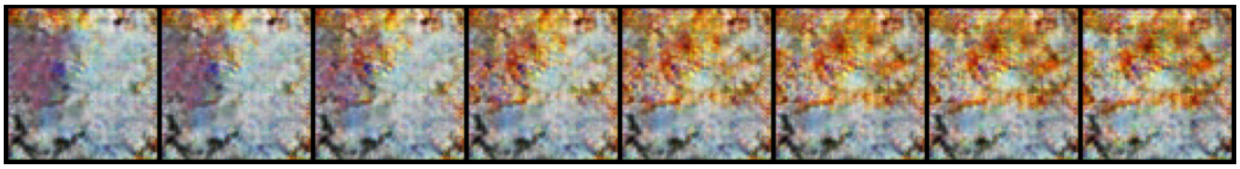


Figure 5.2.2.4: Interpolation of two samples from microstructure class 7

The analysis of the reconstructed interpolated points reveals a notable discrepancy with the original images of the same class. One potential reason for this disparity could be attributed to the GAN training data that includes both colored and black and white images, which might have led to confusion in the model's learning process. Another plausible explanation could be the insufficiency of the training data size for the GAN model, indicating a possible need for a larger and more diverse dataset to improve its performance. Furthermore, the GAN may be stuck at a local optimum and that could be a potential reason why results did not improve any further after 2000 epochs of training.

Comparing original vs. reconstructed spatial statistics for the GAN is not trivial because the GAN produces arbitrary samples. The samples can be from any class, and one has to implement a framework that clusters generated samples according to their class label. We leave this task for now, as the current GAN architecture will likely not be an option that will be explored any further.

5.3 LDM Results

Training our LDM required a lot more computational resources than the frameworks tested above in Sections 5.1 and 5.2. To enable the models to learn, we employed the additional cloud GPU power available with the Google Colab Pro+ subscription. This allowed the team access to increased CPU and GPU RAM to be able to train this framework. After training to ~150 epochs across two classes of microstructures over the course of a night, the LDM frameworks produced results which are very close to the original image. This model performed really well, with a latent space dimensionality of just 64. As such, the team was able to reduce the original dimensions of the images from 256x256 to an 8x8 pixel representation. It is important to note that the latent representations are also different based on the input microstructure image, showing the framework understands the underlying differences between these inorganic composite microstructure classes of data.

5.3.1 Reconstruction Results for LDM

To illustrate the capability of this framework, there are a few examples where a given ‘original’ input image is encoded into a ‘latent representation’, and then lastly reconstructed back from this latent vector. Compared to the input images, the outputs are very similar but with slight amounts of blurriness. Additionally, the output reconstructions also do not contain the very small grains and dots that the input images contain, showing that the model is likely not overfitting to the data. These example reconstructions and latent representations can be seen below from Figure 5.3.1.1 to Figure 5.3.1.5.

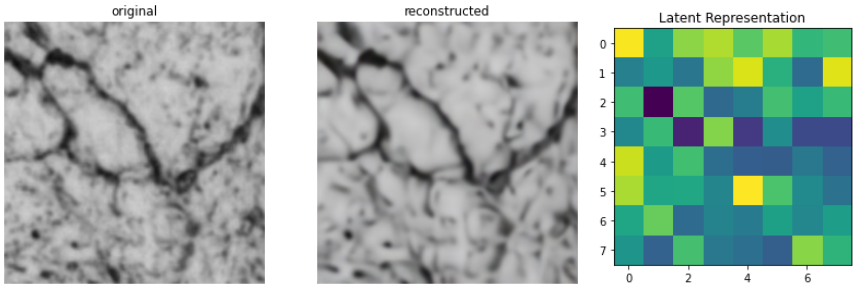


Figure 5.3.1.1: Reconstruction of Sample from Microstructure Class 9

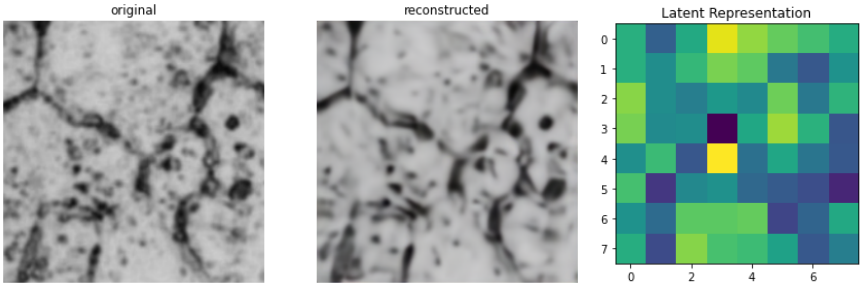


Figure 5.3.1.2: Reconstruction of Sample from Microstructure Class 9

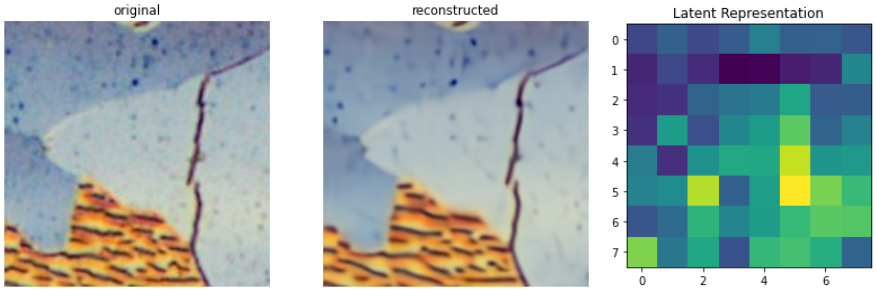


Figure 5.3.1.3: Reconstruction of Sample from Microstructure Class 1

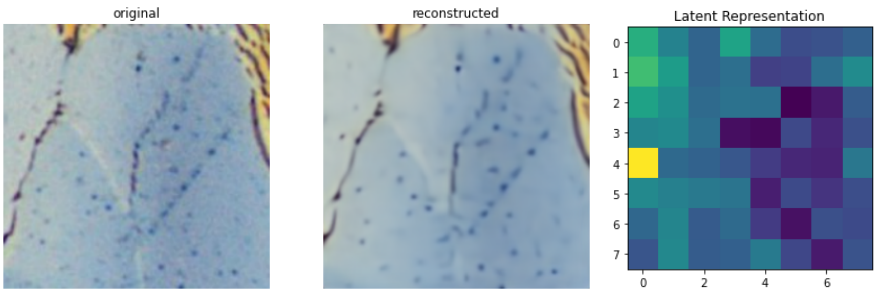


Figure 5.3.1.4: Reconstruction of Sample from Microstructure Class 1

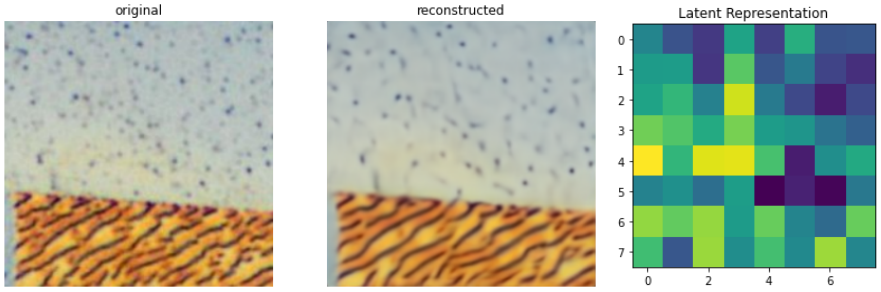


Figure 5.3.1.1: Reconstruction of Sample from Microstructure Class 1

5.3.2 Interpolation Results for LDM

Similar to Section 5.1.1, below are also results from interpolations performed between two selected real images, with the two real images from the dataset being on opposite sides of the figures. In between these original images are the interpolations. These interpolations are between images that belong to the same class but have different physical appearances. To state again, the purpose of this exercise was to determine whether the model can generate distinct variations of the same microstructure class. These interpolations were performed using the same method as in Section 5.1.2, where first we encode two original samples from a given class and generate a latent representation of each image. Next, we linearly interpolate between the two points in the latent space and select eight equidistant points from the interpolation. Finally, each of these eight equidistant points are fed into the framework to reconstruct microstructure images using these modified latent representations. The results from the interpolations can be seen below from Figure 5.3.2.1 to Figure 5.3.2.3.

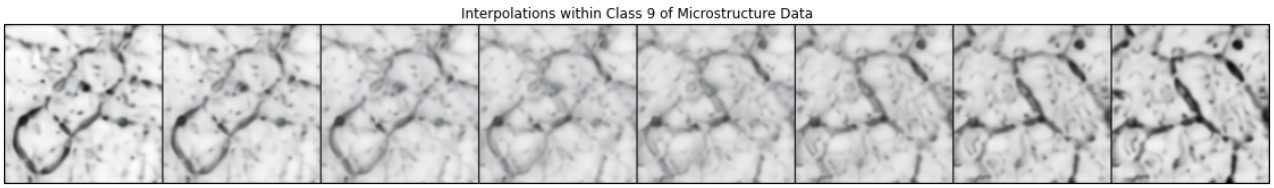


Figure 5.3.2.1: Interpolation of two samples from microstructure class 9

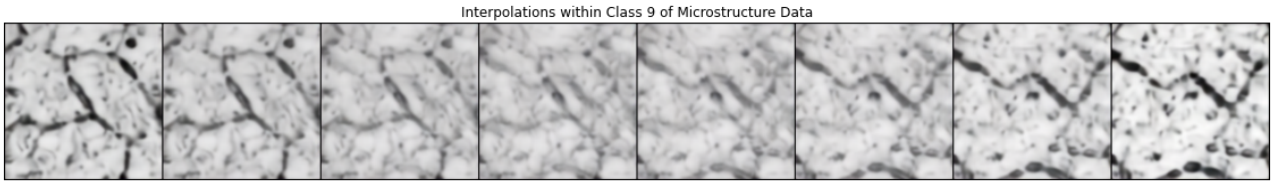


Figure 5.3.2.2: Interpolation of two samples from microstructure class 9

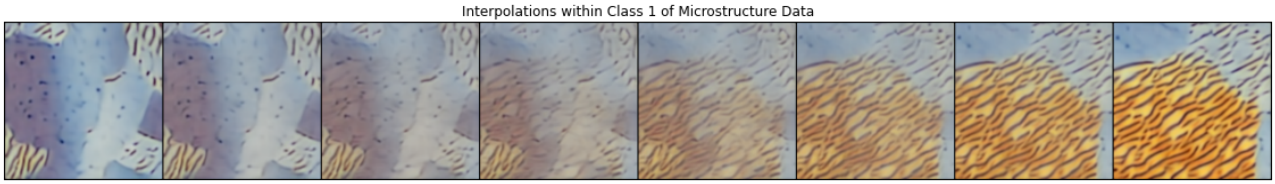


Figure 5.3.2.3: Interpolation of two samples from microstructure class 1

As can be seen from the interpolation results, the model is not as robust in performing interpolations as it is in reconstructing microstructure images. Nonetheless, the interpolations are somewhat unique at each of the eight equidistant points. The interpolations for class 9 are somewhat realistic, as the model extracts features from each of the two original images in its interpolated results. However, the interpolations have increased blur and lack clarity. The interpolations for class 1 are also somewhat appropriate but lack robustness as it does not seem that the interpolations are unique images - just a mix of the two original images.

5.3.3 Original vs Reconstruction Spatial Statistics for LDM

The last metric that the team looked at is a comparison between the original and reconstructed image in spatial statistics as referenced by Khosravani [6]. In addition to microstructures having the same physical appearance, it is also important for microstructures within the same class to be statistically similar. Similar to Section 5.1.3, Figure 5.3.3 below shows a comparison between the mean error of the original and reconstructed microstructure images in their spatial statistical space. Calculating spatial statistics on a tensor results in another tensor of the same shape. To get a scalar metric for the spatial tensor, we can calculate the mean of this spatial statistic tensor. The blue bars in Figure 5.3.3 below show the average mean error in the mean spatial statistic of the original microstructure images for each class, whereas the orange bars show the average mean error in the mean spatial statistic of the reconstructed microstructure image.

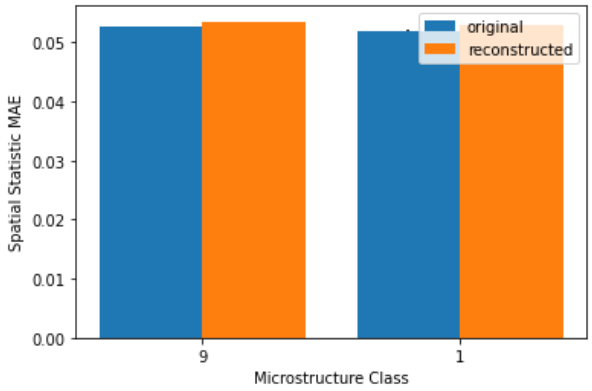


Figure 5.3.3: Mean Error Between Transformed Original and Reconstructed Microstructure Images using Spatial Statistics Between Class 1 and 9

From this Figure, we can see that the original microstructure images and the reconstructed microstructure images are also statistically the same. As such, this result shows that not only are the reconstructures from the LDM framework physically the same, but also statistically from the same class of microstructures as well.

6. Limitations

Although the candidate design solutions explored in this project achieve the objectives while adhering to the constraints, there are a few limitations and drawbacks of each framework. While these limitations do not make any of the frameworks infeasible, it is important to bring them to light so that further work can focus on addressing these limitations.

6.1 C-VAE Limitations

The main drawback of the C-VAE is the low reconstruction quality of its generated outputs. Published work around variational autoencoders has listed the blurry, poor output quality of VAE outputs to be a key limitation with this framework. This drawback also appears in the team's results, which does not allow the results from this framework to be as robust as the team would have liked. However, many researchers have hinted at studies and papers where this output from the VAE can be made more robust by adding a GAN discriminator component to the architecture which helps the outputs look more realistic and clear [5].

6.2 DCGAN Limitations

Next, one key limitation and drawback of the GAN framework is the possibility for the model to collapse while training. This can happen due to mode collapse, which is a common problem where the generator learns to produce only a limited set of samples, ignoring the rest of the input data. This can result in the generator producing repetitive or low-quality samples that lack diversity. This is evident from some generated samples this design produced, which look exactly like the original data. Another key limitation for the GAN architecture is the instability in training. GANs are prone to instability during training, which can make it difficult to train them to generate high-quality samples consistently. This instability can manifest as oscillations in the loss function or the generator and discriminator getting stuck in a sub-optimal state. Moreover, the GAN does not have an encoder and will not allow for the encoding of microstructure images into the latent space. Because of this, the GAN may not be an ideal solution to this problem.

6.3 LDM Limitations

A key limitation on the LDM design framework is the amount of computational resources still needed to train the models. Although the diffusion model requires relatively less compute since the diffusion process is happening in the latent space, training the autoencoder requires a lot of compute power. This is due to its depth, self-attention mechanisms within the framework, as well as the loss calculations.

One key limitation of all the design solutions however is the need to be trained on a lot more data than what is currently available to the team. The design solutions presented above show that it is possible to generate both a latent space and sample microstructures given input microstructure diagrams, and that the frameworks chosen do work. However, the dataset on which these models are trained on needs to vastly expand for these models to be truly valuable in their robust application to varying microstructure classes of input data.

7. Implementation Plan

The LDM model will be designed to be used by our client. It has been designed to perform three main functions. Firstly, it is capable of being trained on a given set of microstructure classes, allowing researchers to feed in their own data and train the model on a specific set of materials. Secondly, the user will be able to encode microstructure images into a latent representation, providing the Argonne team with the ability to perform optimizations on the latent space representation of samples. Finally, the software can also be used to reconstruct microstructure samples when given a latent representation, allowing researchers to decode optimized microstructure representations. By offering these three key features, the LDM model provides material scientists with a powerful tool for deep learning analysis of microstructures.

8. Next Steps

There are a few next steps which can be taken following the takeaways from this project. Each takeaway builds on one or more of the candidate design frameworks discussed above, with an emphasis on steps that can be taken to further improve results.

The first next step is to implement Score Based Generative Modelling in Latent Space (LSGM). LSGM is a variation of the VAE that includes a Score Based Model to enable improved performance [7].

Moreover, we can add conditioning to the diffusion model in the LDM design framework. The true strength of diffusion models is being able to generate new and realistic microstructures after having learned the probabilistic distribution of various microstructure classes, which can be altered using conditioning. The paper implements conditioning using the *Embedding* module within PyTorch. However, for the purposes of this design and project, the conditioning component was zeroed out to simplify learning and generated outputs. As such, one key next step would be to add in the conditioning mechanism back into the model and allow conditioned generations using desired physical and chemical properties.

Furthermore, with regards to the dataset itself, the 2D microstructure dataset can be expanded. Currently, there are only 12 samples per class. This can cause the model to overfit on the limited data it has trained on. Latent diffusion models are particularly more prone to overfitting due to their complexity relative to other models. As such, adding to the dataset can help alleviate guard against the possibility of overfitting.

Finally, the next step towards building a more comprehensive model includes testing with 3D microstructure data. This would involve the associated changes to the model itself to be able to accommodate the data, as well as conducting the relevant iterations on the model to optimize it for 3D data. However, implementing this step is contingent on having sufficient computational resources available, as training a model with 3D microstructure data will be very computationally taxing.

9. Conclusion

We conclude this report by summarizing our project achievements. Our goal was to implement a framework capable of compressing microstructure images and reconstructing the original dimensionality while maintaining high reconstruction quality. We successfully accomplished this task on 2-dimensional microstructure images by reducing the dimensionality by 99.8% with the help of latent diffusion models while maintaining a very high reconstruction quality.

Our project's success shows the potential of applying deep learning techniques to address complex material science problems. This framework can be used to optimize material designs, simulate material properties, and enable more efficient data storage and sharing in the field of material science. Future work includes expanding the framework to handle higher-dimensional microstructure images and further improving the reconstruction quality.

References

[1] A. Sharma, "Variational Autoencoder in TensorFlow (python code)," LearnOpenCV, 19-Oct-2021. [Online]. Available: https://learnopencv.com/variational-autoencoder-in-tensorflow/. [Accessed: 24-Mar-2023].

[2] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative Adversarial Networks," arXiv.org, 07-Jan-2016. [Online]. Available: https://arxiv.org/abs/1511.06434. [Accessed: 24-Mar-2023].

[3] "DCGAN (deep convolutional generative Adversarial Network) generator ..." [Online]. Available: https://www.researchgate.net/figure/DCGAN-Deep-Convolutional-Generative-Adversarial-Network-generator-used-for-LSUN_fig1_340884113. [Accessed: 25-Mar-2023].

[4] Rombach, R., Blattmann, A., "High-Resolution Image Synthesis with Latent Diffusion Models" April 2022. Arxiv. [Online]. Accessed: 21-Oct-2022. Available: https://arxiv.org/abs/2112.10752

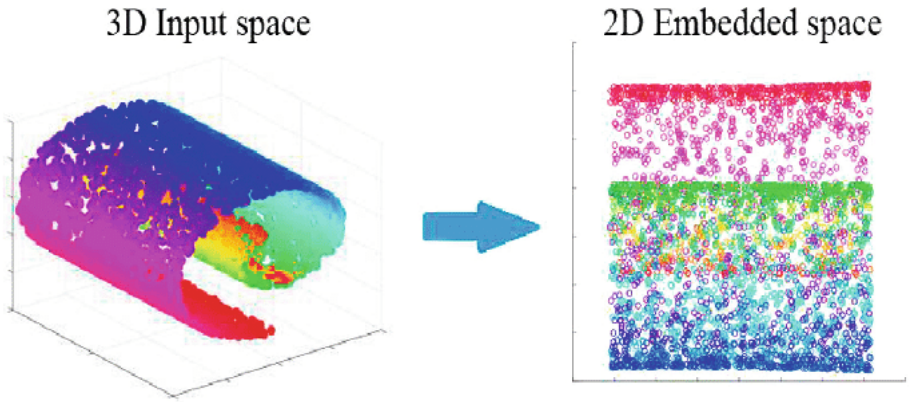
[5] Rombach, R., "Adversarial Variational Auto Encoder" Dec 2020. Arxiv. [Online]. Accessed: 11-Dec-2022. Available: https://arxiv.org/abs/2012.11551

[6] Khosravani, A., Cecen, A., "Development of high throughput assays for establishing process-structure-property linkages in multiphase polycrystalline metals: Application to dual-phase steels", Science Direct, 2017. [Online]. Accessed: 17-Oct-2022. Available: https://www.sciencedirect.com/science/article/abs/pii/S135964541630800X

[7] A. Vahdat, K. Kreis, and J. Kautz, "Score-based generative modeling in Latent Space," arXiv.org, 02-Dec-2021. [Online]. Available: https://arxiv.org/abs/2106.05931. [Accessed: 24-Mar-2023].

Appendix A

The figure below demonstrates an example of reducing dimensionality from 3D to 2D.



The picture on the right-hand-side is the result of reducing the dimensionality of the 3D data in the picture on the left-hand-side. Reducing data dimensionality makes data analysis and visualization easier. As it can be seen, the segmentation of different coloured regions in the 2D data is much easier than the segmentation of different colored regions in the 2D data.

3D papers

In another framework, the authors used a voxel-based variational autoencoder (VAE) for latent space modeling and a voxel-based deep CNN to perform object classification [3]. Their methods take into account challenges specific to voxel representations, and demonstrate the viability of voxel representations in discriminative tasks. Their model also learns to smoothly interpolate between arbitrary, previously unseen shapes. Latent space interpolation is one application that our client is interested in.

Another framework leveraged a learning-based method for interpolating and manipulating 3D shapes represented as point clouds that is explicitly designed to minimize information loss in the latent space. Their approach is based on constructing two autoencoders that enable shape synthesis and, at the same time, provides links to the intrinsic shape information. Their method avoids expensive optimization and the strong regularization provided by our dual latent space approach also helps to improve shape recovery in challenging settings from noisy point clouds across different datasets. Experiments show that their method results in more realistic and smoother interpolations compared to baselines.

Appendix B

```
ResNet_VAE(
  (resnet): Sequential(
    (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (4): Sequential(
      (0): Bottleneck(
        (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (downsample): Sequential(
          (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (1): Bottleneck(
        (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      )
      (2): Bottleneck(
        (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      )
    )
    (5): Sequential(
      (0): Bottleneck(
        (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (downsample): Sequential(
          (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
```

```
)
(1): Bottleneck(
  (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(2): Bottleneck(
  (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(3): Bottleneck(
  (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(4): Bottleneck(
  (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(5): Bottleneck(
  (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(6): Bottleneck(
  (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(7): Bottleneck(
  (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
)
(6): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (2): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (3): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (4): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (5): Bottleneck(
```

```
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(6): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(7): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(8): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(9): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(10): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(11): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(12): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(13): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(14): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(15): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(16): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(17): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
```

```
(18): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(19): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(20): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(21): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(22): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(23): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(24): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

```
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(25): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(26): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(27): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(28): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(29): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(30): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
```

```
  )
  (31): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (32): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (33): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (34): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (35): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
)
(7): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
```

```
      (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     )
    )
    (1): Bottleneck(
     (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
     (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
     (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
     (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
     (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
     (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
     (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
     (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (relu): ReLU(inplace=True)
    )
   )
   (8): AdaptiveAvgPool2d(output_size=(1, 1))
  )
  (fc1): Linear(in_features=2048, out_features=1024, bias=True)
  (bn1): BatchNorm1d(1024, eps=1e-05, momentum=0.01, affine=True, track_running_stats=True)
  (fc2): Linear(in_features=1024, out_features=1024, bias=True)
  (bn2): BatchNorm1d(1024, eps=1e-05, momentum=0.01, affine=True, track_running_stats=True)
  (fc3_mu): Linear(in_features=1024, out_features=256, bias=True)
  (fc3_logvar): Linear(in_features=1024, out_features=256, bias=True)
  (fc4): Linear(in_features=256, out_features=1024, bias=True)
  (fc_bn4): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc5): Linear(in_features=1024, out_features=1024, bias=True)
  (fc_bn5): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (convTrans6): Sequential(
   (0): ConvTranspose2d(64, 32, kernel_size=(3, 3), stride=(2, 2))
   (1): BatchNorm2d(32, eps=1e-05, momentum=0.01, affine=True, track_running_stats=True)
   (2): ReLU(inplace=True)
  )
  (convTrans7): Sequential(
   (0): ConvTranspose2d(32, 8, kernel_size=(3, 3), stride=(2, 2))
   (1): BatchNorm2d(8, eps=1e-05, momentum=0.01, affine=True, track_running_stats=True)
   (2): ReLU(inplace=True)
  )
  (convTrans8): Sequential(
   (0): ConvTranspose2d(8, 3, kernel_size=(3, 3), stride=(2, 2))
   (1): BatchNorm2d(3, eps=1e-05, momentum=0.01, affine=True, track_running_stats=True)
   (2): Sigmoid()
  )

)
```