Giorgio Mendoza
CS549-E24-E01
Dr. Alexandros Lioulemes

Week 10 Lab

PART 1

Objective: Part 1 goal is to train a Deep Convolutional Generative Adversarial Network to generate new, "fake" images of handwritten digits. This task involves following a tutorial provided by TensorFlow, which guides the implementation and training process of the DCGAN model on the MNIST dataset. The end goal is to evaluate the model's ability to produce realistic digit images that mimic the style and structure of the original dataset.

Code Description: In this part, we implement a DCGAN to generate new, "fake" images of handwritten digits from the MNIST dataset. The program begins by loading and preprocessing the dataset, followed by defining the generator and discriminator models using TensorFlow and Keras.

The generator creates new images from random noise, while the discriminator attempts to distinguish between real and fake images. The models are trained in tandem using a custom training loop that includes functions for calculating loss and optimizing both models.

The code also includes functionality to save model checkpoints during training and to generate and save images at each epoch. At the end, a GIF is obtained from the images produced during the training process, showcasing the progression of the generated images over time.

Conclusion: The output from this part of the project met the expectations, with some generated images closely resembling real digits, reflecting the style in which they were trained. It was observed that increasing the number of epochs improved the quality of the generated images, though it also resulted in longer processing times. This highlights the trade-off between model performance and computational efficiency.

Figure 1. Fake Numbers Progession

PART 2

Objective: In this part, the objective is to use the Fashion-MNIST dataset to train a DCGAN for generating new "fake" images of fashion items. The goal is to implement a similar Python script as before that follows the process of creating and training the DCGAN model. The model will be trained to generate realistic-looking images of clothing items, similar to those found in the Fashion-MNIST dataset.

Code Description: This program generates new "fake" images of fashion items using the Fashion-MNIST dataset. The code begins by loading and preprocessing the dataset, where the images are reshaped and normalized to fit the input requirements of the model.

The generator model is defined using several layers, including Dense, BatchNormalization, LeakyReLU, and Conv2DTranspose. This model takes in a random noise vector and generates a 28x28 grayscale image of a fashion item. The discriminator model, on the other hand, is designed to classify images as either real or fake using Conv2D, LeakyReLU, and Dropout layers, followed by a Dense layer for the final classification.

During training, the generator and discriminator models are optimized using the Adam optimizer. The generator aims to produce realistic images to fool the discriminator, while the discriminator tries to correctly classify real and fake images. The training loop runs for 50 epochs, and during each epoch, the models are updated using the computed gradients from the loss functions. Checkpoints are saved periodically to allow for later continuation of training.

After each epoch, generated images are saved and displayed, and at the end of the training process to visually verify the output.

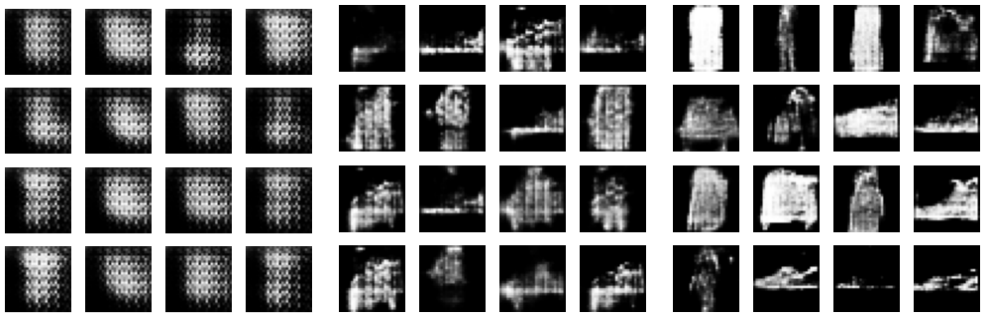


Figure 2. Fake Clothes Progression

Conclusion: In this part of the lab, a DCGAN was successfully trained on the Fashion-MNIST dataset to generate new, "fake" fashion-item images as shown above. Over the course of 50 epochs, the model demonstrated a clear progression in the quality and realism of the generated images. The results indicate that the DCGAN effectively learned to capture the underlying patterns of the fashion items, progressively improving its ability to generate coherent and recognizable images. This experiment showcases the GANs ability in generating synthetic data that resembles real-world inputs.

PART 3:

Objective: The objective of this part is to train a Neural Radiance Field (NeRF) network on a dataset of Bulldozer images, following the provided tutorial. NeRF is an advanced approach to generating novel views of complex 3D scenes by optimizing an underlying 3D representation using only 2D images.

By training the model on the Bulldozer dataset, the aim is to synthesize photorealistic renderings of the Bulldozer from different viewpoints. The final deliverables will include the trained model and the rendered outputs after various epochs, demonstrating the model's learning and rendering capabilities.

Code Description: This implementation focuses on training a Neural Radiance Field (NeRF) model using TensorFlow and Keras to synthesize novel views of a 3D scene from 2D images of a Bulldozer. The process begins with data preparation, where the TinyNeRF dataset is loaded and preprocessed. This

dataset contains images and corresponding camera poses, which are split into training and validation sets. A data pipeline is then created to handle the batching and shuffling of the data efficiently.

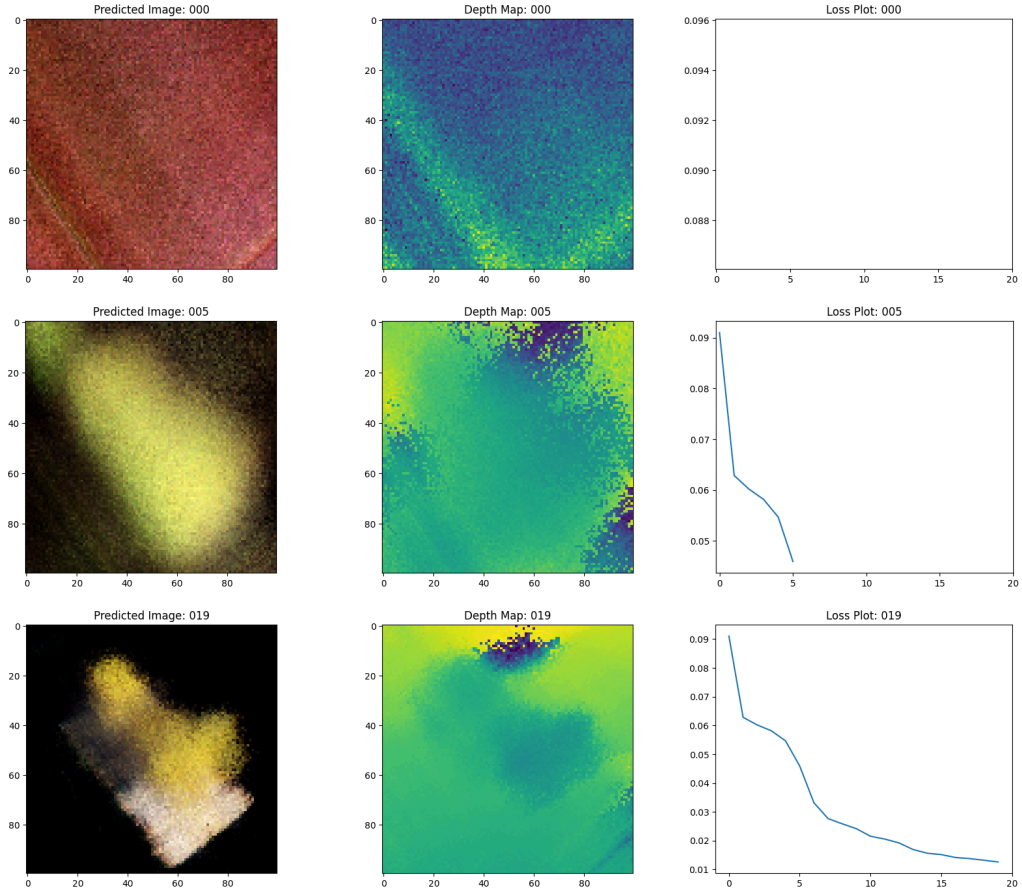


Figure 3. Predicted Images Progression

The core of the implementation is the NeRF model, which is built as a Multi-Layer Perceptron using Keras. This model takes encoded ray positions as input and predicts the RGB color and volume density for each point in the 3D space. The training process is managed within a custom Keras model class, enabling the use of `model.fit` for training. The Mean Squared Error loss function is utilized, and the optimization is carried out using the Adam optimizer. Additionally, a custom callback is defined to monitor the training process and save intermediate results, such as generated images and depth maps.

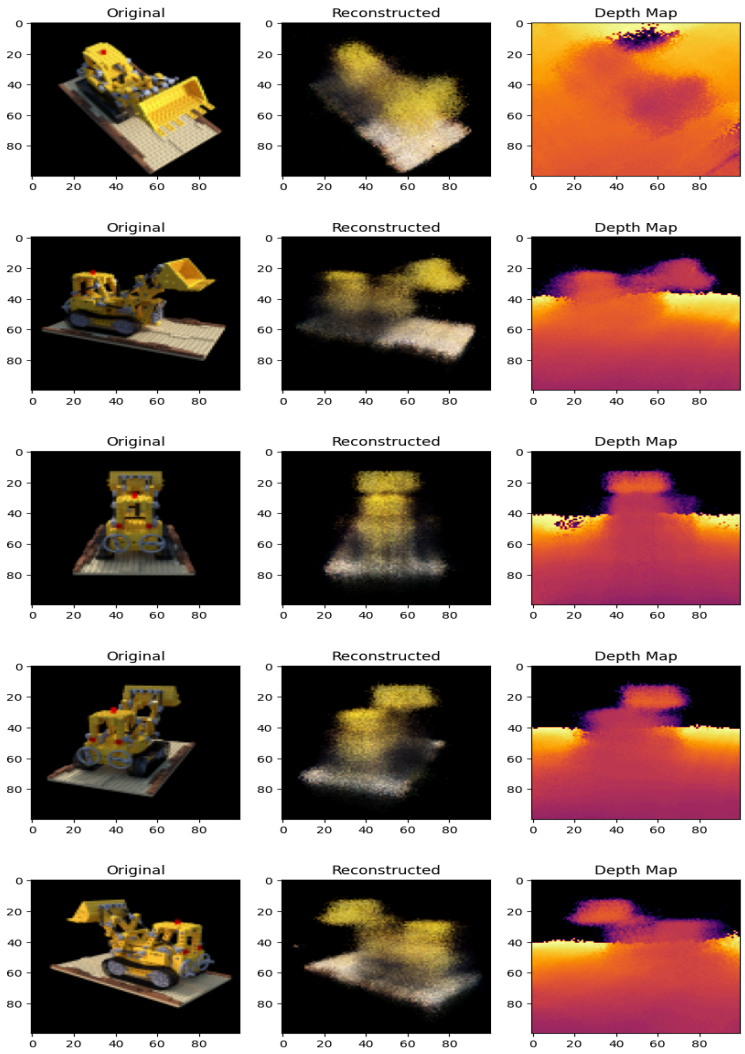


Figure 4. Reconstructed Images Progression & Depth Map

As the model trains, it renders images and depth maps by predicting the RGB colors and densities along sampled rays. These predictions are combined to produce the final output images, which are visualized and saved for further analysis. After training, the model is employed to generate novel views of the scene by varying the camera angle, demonstrating the NeRF model's ability to create high-quality 3D reconstructions from 2D images.

Conclusion: By closely following the steps outlined in the tutorial, the NeRF model was successfully trained to reconstruct a 3D representation of the Bulldozer using the provided images. The model's ability to synthesize realistic views from novel angles was validated by generating a video that smoothly transitions around the object, showcasing the reconstructed scene.

Additionally, the code allowed for visualizing individual frames and depth maps, offering further insight into the model's internal representations. The results clearly demonstrate the effectiveness of NeRF in generating high-quality 3D scenes from 2D images, with the final outputs, including the video for further analysis.

Figure 5. Final Output

PART 4:

Objective: The goal of this task is to extend the implementation of the Neural Radiance Fields (NeRF) algorithm by training it on a custom dataset. The task involves capturing a series of images of an object from various angles, ensuring that the object has a matte texture with minimal reflections, and that the scene is uniformly lit with a black or white background.

The collected images will be used to train the NeRF algorithm to synthesize a 3D scene. Ground truth camera poses will be obtained using tools such as COLMAP or VisualSfM.

Code Description: In this part of the project, VisualSfM was used to generate an NVM file, which contains camera parameters like focal lengths, positions, and orientations, along with 3D coordinates of points in the scene. To simplify the implementation, the camera matrices derived from the NVM file were directly coded. These matrices are essential for accurately simulating the scene from various viewpoints in the NeRF model.

The program implements a workflow for training a Neural Radiance Fields model on a custom dataset. We start by setting global variables like batch size, the number of samples, and the number of positional encoding dimensions.

```
4145.75732422 0.758623812598 -0.647597910559 -0.0481619343853 -0.0528039806917 -0.57979766101 -0.00331523648927 1.28509022728 -0.194859420378 0
4129.03173828 0.755033849016 -0.655257921985 -0.0198042020556 -0.0130093347474 0.0599650755664 0 1.70663820684 -0.191261467262 0
4172.72753906 0.7464482351 -0.659688403675 -0.0616571129673 -0.0618378052673 -1.08046104842 0.0489413572262 1.65483007223 -0.182582892704 0
4148.46337891 0.755518661094 -0.645345855067 -0.0880937077128 -0.0785426699978 -1.57678784001 -0.0975716595425 1.29181683462 -0.205603149392 0
4124.29199219 0.733156170361 -0.672386266891 -0.0683705886149 -0.0755249057089 -1.16197829822 -0.272155529 2.24350875448 -0.177449291786 0
4131.56396484 0.738639692799 -0.671420792683 -0.0329712455649 -0.0501831701096 -0.126492832249 -0.116717685747 2.15234193909 -0.174967678889 0
4164.06347656 0.74257750715 -0.655361837421 -0.101395437775 -0.0938022632763 -2.35339167134 -0.0836421589215 1.77675987011 -0.210984870517 0
4166.92382813 0.752580193737 -0.64250216932 -0.108082478346 -0.0955636311948 -2.37304200599 -0.226072431927 1.30097274709 -0.208111071675 0
4101.05322266 0.728295098715 -0.672885506235 -0.0745158643361 -0.106104424622 -1.91987685187 -0.533633840678 2.28697443698 -0.184732709348 0
4223.29541016 0.769159042337 -0.630007984748 -0.0829159835009 -0.0678949552546 -1.89471354833 4.4408920985e-016 0.699623388684 -0.205207446801 0
4201.23242188 0.770505550914 -0.633040128949 -0.0537419450258 -0.051897034136 -0.995631442565 -0.00333861708323 0.717111782597 -0.183786884016 0
4151.76318359 0.766519167782 -0.640476247728 -0.0303328629389 -0.0363021835922 -0.123656776365 -0.0996707255232 0.824938063414 -0.177053556251 0
4138.23144531 0.758623744449 -0.651209015458 -0.00837559898786 -0.0186216807075 0.707629549281 -0.00660720769958 1.3523192443 -0.188815949586 0
4141.14697266 0.752122908574 -0.658961748541 0.00417230959828 0.00794908450673 1.01534439481 -0.045810906291 1.72693197694 -0.180934409861 0
4133.94677734 0.737540887538 -0.67529523274 0.00311353722561 8.88647107287e-005 1.10379163087 -0.0835638814891 2.16285256585 -0.175834133793 0
4163.68457031 0.765132727021 -0.643772153919 0.0111124181076 -0.00238748925243 1.06153463126 4.4408920985e-016 0.911948284645 -0.172729491332 0
4166.53125 0.758011386054 -0.651889389703 0.0211571609935 0.00335548565078 1.7935226425 -0.000354836607018 1.39507969884 -0.173785667855 0
4139.42431641 0.760318762227 -0.649192566944 0.0203925441372 0.00695794663732 1.6333654544 -0.140334719684 1.07633762949 -0.169666979027 0
4162.82177734 0.740881052168 -0.670643679691 0.0279236941287 0.0235105522652 1.91772229273 -0.101216828308 1.78632026385 -0.163804225099 0
4163.56542969 0.732874719567 -0.679029009316 0.0305489566031 0.0296811216703 2.07973970831 -0.0943726251173 2.06338421619 -0.168660281129 0
```

Figure 6. NVM file Parameters

Data preprocessing involves loading images from a specified directory, resizing them to a fixed height and width, and processing camera poses obtained from an external source, converting them into 4x4 transformation matrices. This prepares the dataset for training.

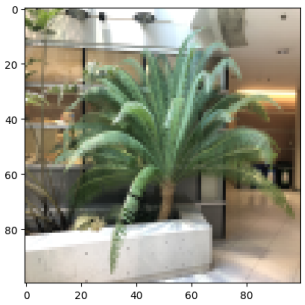


Figure 7. Low-res input image

The script includes a positional encoding function, encode_position, which transforms input coordinates into a higher-dimensional space. This step is critical in NeRF for capturing fine details in the scene.

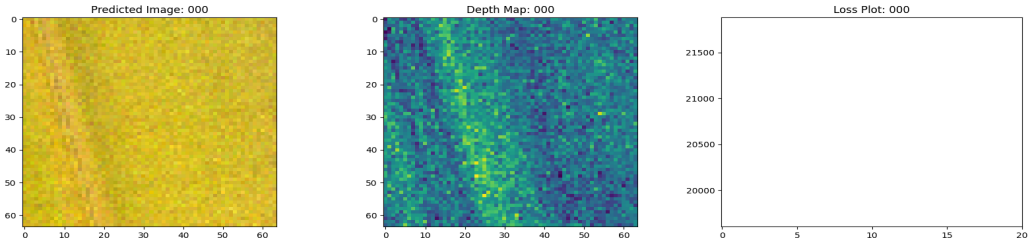
Ray generation is handled by the get_rays function, which generates rays corresponding to each pixel in the image based on camera intrinsics and extrinsics. These rays are sampled at multiple points along their paths, a key process in volume rendering.

The render_flat_rays function flattens the sampled rays into a format suitable for input to the NeRF model, incorporating random noise to improve training robustness. The script then splits the images and poses into training and validation datasets, creating TensorFlow datasets for efficient data loading and batching.

The NeRF model is defined using the get_nerf_model function, which constructs a Multi-Layer Perceptron with residual connections. This model takes the flattened rays as input and outputs RGB values and density (sigma) for volume rendering.

Training and monitoring are managed by the NeRF class, which extends keras.Model, defining the training and evaluation steps. A TrainMonitor callback is implemented to visualize results at the end of each epoch, saving predicted images, depth maps, and loss plots.

Finally, the script compiles and trains the NeRF model using the prepared datasets, with visualization and logging integrated to monitor the model's progress throughout the training process. This comprehensive implementation allows the user to train a neural network to synthesize novel views of a scene based on a collection of images and corresponding camera poses.



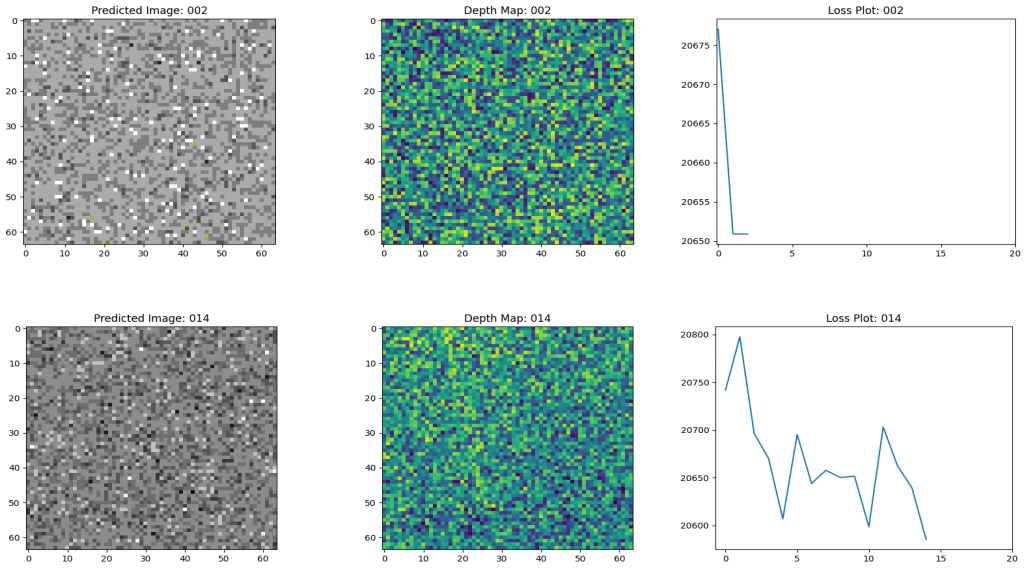


Figure 8. Predicted Image results

Conclusion: Unfortunately, the program was unable to produce accurate image predictions, even though the intermediate matrices were appropriately sized. The primary issue may stem from inadequate model training, which was further complicated by the deactivation of the validation step due to an unresolved AttributeError: 'NoneType' object has no attribute 'items'. This error prevented the program from properly validating the model during training.

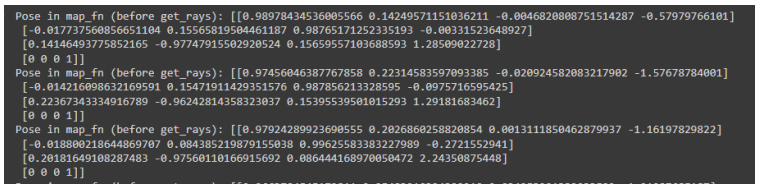


Figure 9. Predicted Image results

A plausible explanation for the high error could be related to the mismatch between the camera matrices and their corresponding images. In the npz file for the bulldozer, the rotation and translation matrices are consistently paired with their respective images. However, in this case, the matrices were manually implemented and the images were imported in sequential order. This mismatch might have led to the significant errors observed in the predictions.

Further refinement of the data preparation and model training process could be necessary to achieve better results. However, I ran out of time.