Synthetic Dataset Impact on Monocular Depth Estimation Model Robustness

ICS 635: Final Project Report

https://github.com/julietteraubolt/ICS635-Final-Project
Mahie Crabbe [26943503], Christine Nakano [24937421], Juliette Raubolt [29029148]

1 Introduction

The purpose of our project is to see if it is possible to improve the robustness of monocular depth estimation (MDE) models by fine-tuning them on synthetically generated datasets. While MDE models such as ZoeDepth have achieved strong performance on standard benchmarks, they often struggle to generalize to scenes that differ significantly from their original training distribution. This project evaluates whether fine-tuning ZoeDepth on synthetic data (randomized, abstract scene structures) improves its robustness and performance.

ZoeDepth is a unified, scalable model for zero-shot monocular depth estimation. It is designed to produce dense depth predictions from a single RGB input across a wide variety of domains. ZoeDepth builds upon advancements in depth estimation by introducing a modular architecture that combines strong feature encoding with flexible decoding strategies. Given an input image, ZoeDepth outputs a per-pixel depth prediction, mapping two-dimensional visual features to three-dimensional spatial information. In this project, we approximate a function that infers scene geometry, object boundaries, and relative distances directly from pixel intensity patterns. Machine learning is appropriate for this task because traditional geometric methods, such as stereo vision or structure-from-motion, require multiple viewpoints or explicit motion information, which are unavailable in single-image scenarios. Moreover, handcrafted feature-based methods struggle with challenges such as textureless regions, reflections, and occlusions, conditions that learning-based models can overcome by using large-scale statistical patterns learned from data.

Overall, this project aims to improve Single-image Depth Estimation (SIDE). Single-image depth estimation (SIDE) is important because it allows machines to understand 3D structures from just one 2D image, without needing extra hardware like stereo cameras or LiDAR. This capability is critical for making depth sensing cheaper, lighter, and more accessible, especially in fields like robotics, autonomous driving, augmented reality (AR), and mobile devices where space, power, and cost are limited. In short, SIDE enables 3D perception using only standard cameras, opening up 3D scene understanding to many more platforms and applications.

2 Data

Our dataset can be found here: https://huggingface.co/datasets/julietter/final_project_synthetic_depth_v1/tree/main

The dataset developed for this project consists of 3,000 labeled samples, where each sample is a pair of an RGB image and its corresponding depth map (6,000 total images). It was constructed by generating 200 distinct scenes within Unity, with each scene composed of a randomly selected background (color or skybox) and populated by 50–75 randomly chosen objects selected from a library of around 30 materials and 30 prefabs. For each scene, 15 images were captured from slightly different camera angles and varying lighting, resulting in a diverse set of views and configurations that collectively form the full dataset. The synthetic dataset was structured into two directories (rgb/ and depth/).

All labels in the dataset are computationally generated and therefore completely trustworthy. The depth maps are captured directly alongside each rendered RGB image using Unity's internal depth capture functionality, ensuring pixel-perfect ground truth without human error. The randomness of scene composition, ranging from object placement to object size, material, and background, was intentional to expose models to a wide variety of non-structured visual cues, thus promoting robustness and adaptability.

The dataset is designed to be easily expandable. Future iterations could add additional scenes, materials, backgrounds, and object types to further enhance variability. Because the dataset generation process is automated within Unity, increasing the dataset size would primarily involve generating new random scenes and re-running the capture script. The current dataset version is publicly hosted on Hugging Face for accessibility and relevant scripts can be found on GitHub for reproducibility.

To conduct the fine-tuning and evaluation of the selected models, we plan to use the KOA High Performance Computing (HPC) Cluster at the University of Hawaiʻi. KOA is a large-scale distributed system consisting of multiple interconnected nodes, designed to handle computational workloads too intensive for standard desktop machines. Our experimental workflow will involve developing the code locally, likely using an integrated development environment such as PyCharm, and then transferring the scripts, models, and dataset to KOA for training. By utilizing KOA's computational resources, we aim to efficiently fine-tune the large models and conduct comprehensive evaluations across the synthetic dataset, while ensuring reproducibility and scalability of our experiments.

3 Methodology

ZoeDepth is a transformer-based monocular depth estimation model that combines elements of supervised and self-supervised learning. It is built on a modular architecture that separates the encoder and decoder components for flexibility and scalability. The encoder is a vision transformer (ViT) with some CNN backbone pre-trained on ImageNet, which extracts high-level visual features from the input image. The decoder translates these features into dense depth predictions at the pixel level. Internally, ZoeDepth uses domain-specific adapters and combines both relative and metric depth supervision. The model outputs a dictionary of predictions, with "metric_depth" providing the dense, real-world depth map. This decoupled structure enables zero-shot transfer to various scene types and supports fine-tuning with small datasets, making it well-suited for our synthetic training data.

Data Splitting

The dataset was partitioned into three subsets following an 80:10:10 split: 80% of the samples were used for training, 10% for validation, and 10% for testing. The validation set was reserved for hyperparameter tuning and model selection, while the test set was used only for final model evaluation. This careful split ensures that model performance metrics are reported on completely unseen data, helping to accurately assess generalization and prevent overfitting.

Data Pre-processing

As mentioned earlier, this project required us to generate an entirely new synthetic data set in Unity. Each of the 200 generated scenes was randomly created from an assortment of:

- 37 prefabricated objects
- 32 different materials (i.e. glass, metal, wood, fabric, etc.)
- 50:50 choice between solid color background or skybox
    - 7 different solid colors & 6 different sky box options
- Every scene had 50 to 75 random objects varying in size, material, and orientation (position & rotation)

Transformations

In order to preprocess the RGB images, a series of transformations were applied to ensure the data aligned with the model's expected input format. Each RGB image was first resized to a fixed resolution of 256×256 pixels using bilinear interpolation, which preserves visual smoothness and minimizes aliasing artifacts during resizing. After resizing, the images were converted into PyTorch tensors, scaling the pixel values from their original 0–255 range down to a normalized 0–1 range. To match the training conditions of the model's encoder backbone (originally trained on ImageNet), the images were then normalized using the standard ImageNet mean and standard deviation values for each color channel. This normalization step ensures that the input distribution remains consistent with the model's pretraining, which is important for achieving stable fine-tuning performance.

For the depth maps, a separate but similarly structured transformation pipeline was used. Each depth image was resized to 256×256 pixels using nearest-neighbor interpolation, which preserves the original discrete depth values without introducing interpolation artifacts that could distort true depth measurements. After resizing, the images were explicitly converted to grayscale with a single output channel to match the expected input shape. They were then converted into PyTorch tensors, scaling their pixel values to the [0, 1] range. Since the depth values originated from normalized grayscale images, an additional manual rescaling was applied after loading to map the depth values into real-world metric units, by multiplying the tensor by a maximum depth value (e.g., 10 meters). This preprocessing ensures that both the RGB and depth data maintain physical meaning and appropriate scaling, providing a reliable foundation for effective fine-tuning of ZoeDepth.

Missing data methods

Given that the grayscale PNG depth maps saved from Unity store a value between 0 and 255 for each pixel, the data set required us to rescale (augment) the images to store real-world metric units for each pixel in order for ZoeDepth (and most depth estimation models) to properly interpret the depths, learn from correct mappings, and ultimately improve the model's performance. We address this in the code such that the depth data is in the proper format. Considerably if you look at the resulting images, it is seen that our depth maps are inverted which do not pose any problem here and are still usable for analysis. Though this problem stems from this concept here.

Augmentation

Each of the 200 generated scenes was augmented with 15 distinct captures (3,000 samples total), each differing in viewing angle, spatial position, and lighting configuration. The primary objective of this augmentation process was to introduce sufficient variability into the dataset, enabling the model to better generalize across a wide range of environmental conditions and mitigate overfitting to specific scene structures.

4 Implementations

This section outlines the technical details of our implementation for fine-tuning ZoeDepth on a synthetic monocular depth estimation dataset. We used PyTorch as our primary deep learning framework, supplemented by torchvision for image transformations, timm for model loading utilities, and Unity for synthetic dataset generation. All training was performed on the KOA High Performance Computing Cluster at the University of Hawai'i due to the resource demands of fine-tuning large models.

We defined all key hyperparameters at the start of the training script (Code Snippet 1). These include learning rate, batch size, image dimensions, weight decay, and early stopping parameters. ZoeDepth's "ZoeD_N" model variant was selected for its balance between performance and computational efficiency.

```python
MODEL_ID      = "ZoeD_N"
OUTPUT_DIR    = "zoedepth-finetuned"
ROOT_DIR      = ""
IMG_SIZE      = 256
BATCH_SIZE    = 2
LR            = 1e-4   # Keeping this kind of low on purpose
WEIGHT_DECAY  = 1e-4   # Our data is super clean and so we want to avoid overfitting
EPOCHS        = 20
SPLIT         = 0.1
DEVICE        = "cuda" if torch.cuda.is_available() else "cpu"
FREEZE        = True
MAX_DEPTH     = 10.0
PATIENCE      = 5
IMAGENET_MEAN = np.array([0.485, 0.456, 0.406], dtype=np.float32)
IMAGENET_STD  = np.array([0.229, 0.224, 0.225], dtype=np.float32)
```

Code Snippet 1: Hyperparameters

We also configured cosine annealing for learning rate scheduling and used the AdamW optimizer to improve stability with small batch sizes and transformer backbones.

To feed it into the model, we implemented a custom DepthDataset class (Code Snippet 2) and corresponding RGB and Depth transforms. Each RGB image was paired with its corresponding depth map, both of which underwent appropriate transformations. Depth maps were rescaled from 8-bit grayscale [0–255] to metric depth values in meters.

```python
def __getitem__(self, idx):
        # Get the filenames for the rgb and depth images
        rgb_filename = self.rgb_files[idx]
        depth_filename = rgb_filename.replace("rgb", "depth")

        # Get the full path of the images
        rgb_path = os.path.join(self.rgb_dir, rgb_filename)
        depth_path = os.path.join(self.depth_dir, depth_filename)

        # Open the images and make suer rgb is in proper format
        rgb_image = Image.open(rgb_path).convert('RGB')
        depth_image = Image.open(depth_path)

        # Apply the transforms
        rgb_tensor = self.transform_rgb(rgb_image)
        depth_tensor = self.transform_depth(depth_image).squeeze(0)

        # Apply the proper scaling for the depth tensor
        depth_tensor = depth_tensor * MAX_DEPTH

        return {
            'rgb': rgb_tensor,
            'depth': depth_tensor
        }
```

Code Snippet 2: Dataset Loading Class

This design allows for scalable data loading using PyTorch's DataLoader, ensuring proper pairing and transformation of RGB and depth inputs.

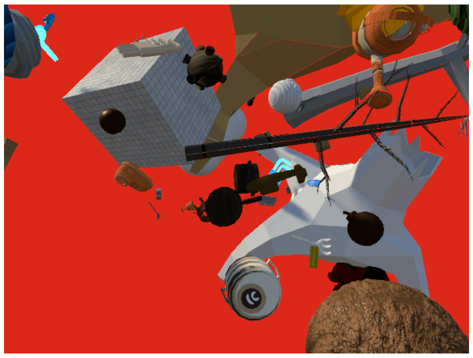
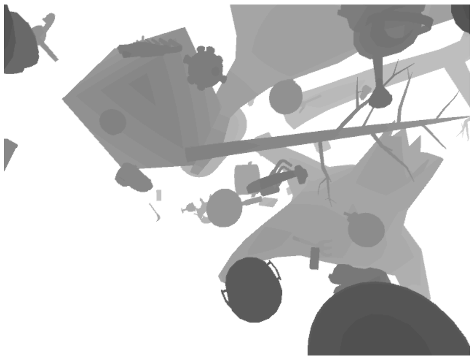
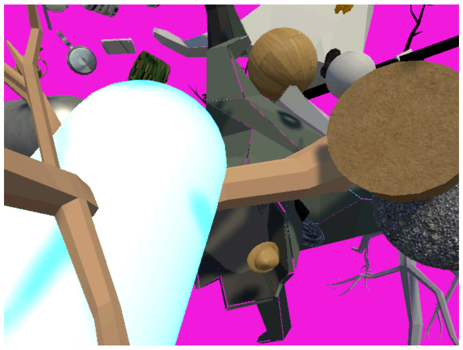
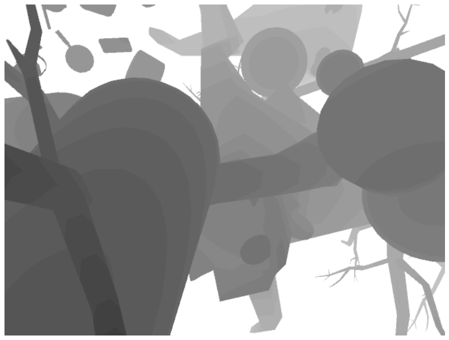
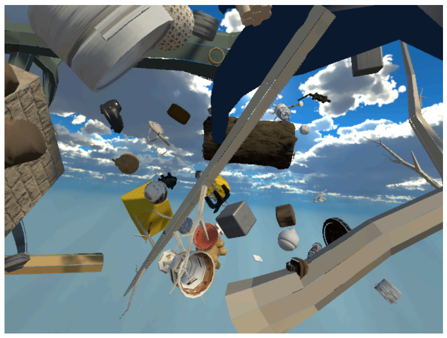
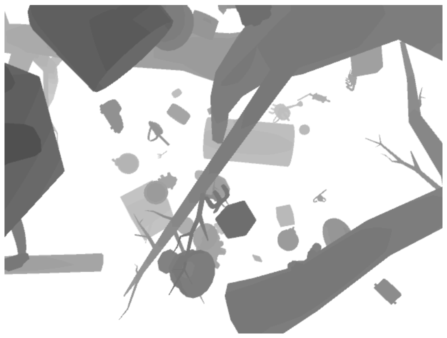
We implemented the Scale-Invariant Logarithmic Loss (SILog) function, a common loss function for depth detection, which penalizes relative errors and is robust to global scale differences, which is perfect for arbitrary scenes and synthetic domains.

```python
def forward(self, input, target, mask=None, interpolate=True):
    input = extract_key(input, KEY_OUTPUT)

    if input.shape[-2:] != target.shape[-2:] and interpolate:
        input = nn.functional.interpolate(input, size=target.shape[-2:],
mode='bilinear', align_corners=True)

    if target.ndim == 3:
        target = target.unsqueeze(1)

    with torch.amp.autocast(device_type=DEVICE, enabled=False):
        eps = 1e-6
        input = input.clamp(min=eps)
        target = target.clamp(min=eps)
        g = torch.log(input) - torch.log(target)
        Dg = torch.var(g) + self.beta * (torch.mean(g) ** 2)
        loss = 10 * torch.sqrt(Dg + eps)
    return loss
```

Code Snippet 3: Loss Function

This loss encourages the model to predict correct relative shapes even when the absolute scale may vary, aligning with our goals for robust generalization.

Unlike standard PyTorch models, ZoeDepth returns a dictionary of outputs, not a single tensor. The main prediction we use is accessed via the key "metric_depth", which provides a dense metric (real-world) depth map. This design allows ZoeDepth to support multiple output types simultaneously (e.g., relative depth, auxiliary maps), but also requires explicit extraction before computing the loss. We handle this in our Loss Function class with a helper function that extracts the key.

RGB	**DEPTH**
	
	
	

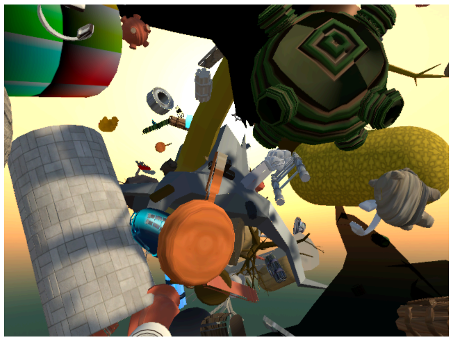
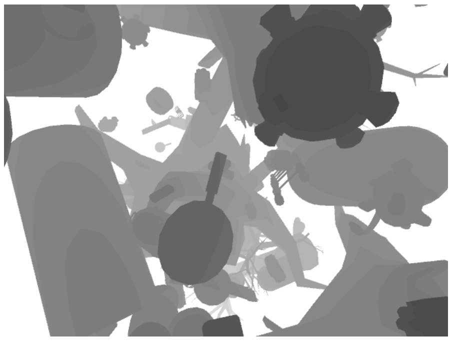
	

Table 1: Dataset Examples

5 Experiments and Results

Tools

The tools used throughout this project include Unity for the dataset development, KOA for the high processing resources, and Jupyter Notebook for implementing the fine-tuning and analysis script.

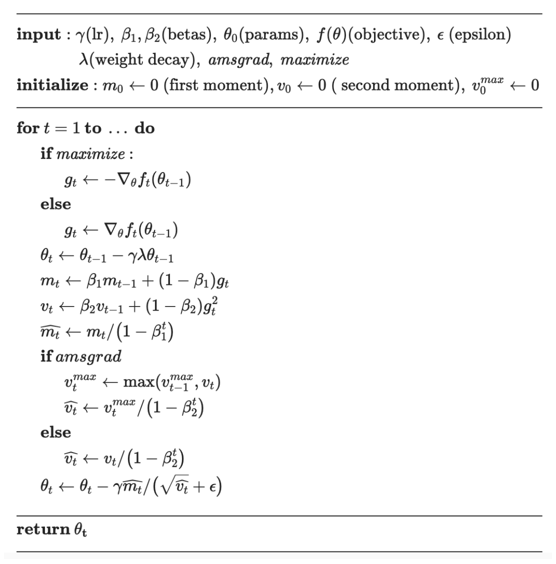
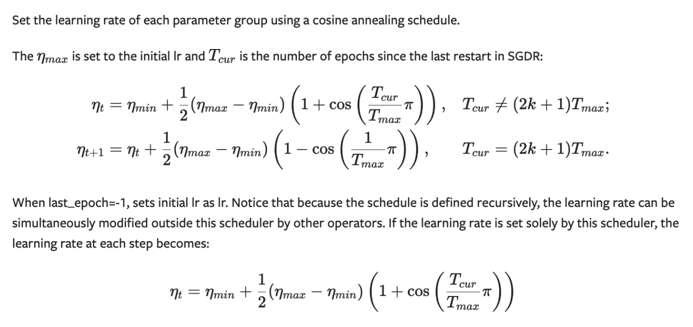
Specify Hyperparameter Optimization

Specific hyperparameter decisions can be seen in Table 2: Hyperparameter details. Additionally, we use the AdamW optimizer and CosineAnnealingLR scheduler. Details and reasoning are explained in Table 3: Fine-Tuning Optimizer and Scheduler

Hyperparameter	Description	Purpose
MODEL_ID = "ZoeD_N"	Specifies which variant of ZoeDepth to use. "ZoeD_N" likely refers to ZoeDepth-Nano, a lightweight version	Choosing the model size impacts performance vs. speed tradeoffs. Nano is faster and less memory-intensive, but less accurate
OUTPUT_DIR = "zoedepth-finetuned"	Directory where fine-tuned model checkpoints and logs are saved	Organizes your outputs, allows resuming training, and tracks progress over time
ROOT_DIR = ""	Root directory where your dataset (RGB + depth) is stored	The model looks here to find training and validation data. Needs to be correct or data loading will fail
IMS_SIZE = 256	The size (height × width) to which images are resized before input to the model.	Controls memory usage, training speed, and detail retention. 256 is small, so faster training but less fine detail
BATCH_SIZE = 2	Number of image-depth pairs processed together per forward/backward pass	Affects training stability and speed. Batch size of 1 is common when GPU memory is limited
LR = 1e-4	Learning rate: controls how much the model updates its weights at each step	A low LR (like 1e-4) is appropriate for fine-tuning to avoid overriding useful pretrained features
WEIGHT_DECAY = 1e-4	L2 regularization strength. Penalizes large weights to prevent overfitting	Lower weight decay is appropriate when training on clean, synthetic data with minimal noise
EPOCHS = 20	Total number of times the model will iterate	Too few = underfitting; too many =

	over the entire training dataset	overfitting. 20 is a good default for fine-tuning on medium-sized data
SPLIT = 0.1	Fraction of the dataset to reserve for validation (not training)	Ensures you can evaluate model performance on unseen data during training
DEVICE = “cuda” if torch.cuda.is_available () else “cpu”	Automatically uses a GPU if available, else falls back to CPU	Deep learning is vastly faster on GPU. This makes the script hardware-adaptive
FREEZE = True	Whether to freeze the encoder (e.g., ViT backbone) during training	Freezing keeps pretrained features intact and fine-tunes only the decoder/head. Useful when training on small or synthetic datasets
MAX_DEPTH = 10.0	The maximum depth (in meters) represented in your training data. Used to rescale depth maps	Critical when depth maps are normalized to [0–1]; this defines what “1.0” actually means in physical space
PATIENCE = 5	Number of epochs to wait without improvement in validation loss before early stopping	Helps avoid overfitting by stopping training once progress stalls
IMAGENET_MEAN = [0.485, 0.456, 0.406]	Mean pixel values per RGB channel, used for normalization	Ensures inputs are in the distribution expected by the ImageNet-pretrained encoder
IMAGENET_STD = [0.229, 0.224, 0.225]	Standard deviations per channel for normalization	Complements the mean — helps keep network activations balanced during training

Table 2: Hyperparameter Details

AdamW	**CosineAnnealingLR**
	
- Handles adaptive learning rates per parameter (good for fine-tuning large models like ZoeDepth). - Correctly decouples weight decay from gradient updates (important for transformers and modern	- Smoothly lowers learning rate over time, helping you start learning fast and finish carefully without overshooting. - Perfect for small datasets and few epochs (like your 3,000 images and 20-epoch training plan).

```
input : $\gamma$(lr), $\beta_1, \beta_2$(betas), $\theta_0$(params), $f(\theta)$(objective), $\epsilon$ (epsilon)
        $\lambda$(weight decay), amsgrad, maximize
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ ( second moment), $v_0^{max} \leftarrow 0$

for $t = 1$ to ... do
    if maximize :
        $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
    else
        $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma\lambda\theta_{t-1}$
    $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)g_t^2$
    $\widehat{m_t} \leftarrow m_t / (1-\beta_1^t)$
    if amsgrad
        $v_t^{max} \leftarrow \max(v_{t-1}^{max}, v_t)$
        $\widehat{v_t} \leftarrow v_t^{max} / (1-\beta_2^t)$
    else
        $\widehat{v_t} \leftarrow v_t / (1-\beta_2^t)$
    $\theta_t \leftarrow \theta_t - \gamma\widehat{m_t} / (\sqrt{\widehat{v_t}} + \epsilon)$

return $\theta_t$
```

Set the learning rate of each parameter group using a cosine annealing schedule.

The η_{max} is set to the initial lr and T_{cur} is the number of epochs since the last restart in SGDR:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right), \quad T_{cur} \neq (2k+1)T_{max};$$
$$\eta_{t+1} = \eta_t + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 - \cos\left(\frac{1}{T_{max}}\pi\right)\right), \quad T_{cur} = (2k+1)T_{max}.$$

When last_epoch=-1, sets initial lr as lr. Notice that because the schedule is defined recursively, the learning rate can be simultaneously modified outside this scheduler by other operators. If the learning rate is set solely by this scheduler, the learning rate at each step becomes:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right)$$

models). - Stabilizes small-batch training (since your batch size is low, AdamW is much safer than plain SGD).	- Requires no manual tuning during training — no abrupt jumps or schedule tweaking needed.

Table 3: Fine-Tuning Optimizer and Scheduler

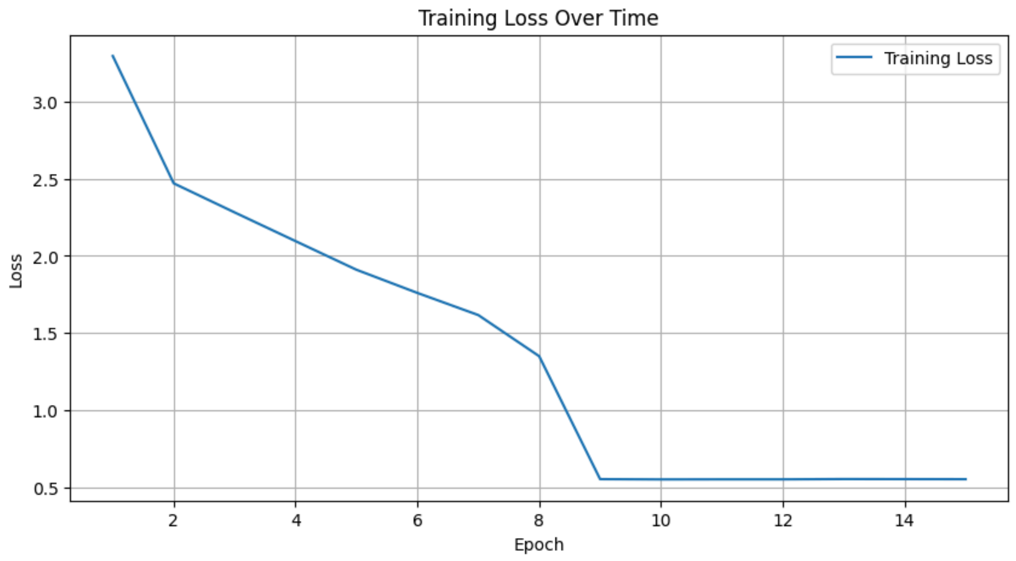
Evaluation Metrics

The evaluation metrics included in the project are explained in Table 4: Evaluation Metrics Details. We wrote a helper function which when called goes through and computes the chosen evaluation metrics. Such that we can measure these metrics on the original model and the fine-tuned model when given the test dataset.

Metric	Meaning	How it works	Purpose
d1	δ1 accuracy: percentage of predicted depths where the prediction error is "close enough" — specifically, less than 1.25× away from ground truth.	For each pixel, compute max(pred/gt, gt/pred). If it's < 1.25, count it as correct. Then report the percentage of correct pixels.	For each pixel, compute max(pred/gt, gt/pred). If it's < 1.25, count it as correct. Then report the percentage of correct pixels. Measures how often your model is acceptably close, ignoring big outliers. It rewards "mostly right" predictions even if the exact value isn't perfect.
abs_rel	Absolute Relative Error: average of	Predicted - ground_truth	divided by ground_truth
rmse	Root Mean Squared Error: square root of average (predicted-ground_truth)^2 across all pixels.	Compute (pred - gt)^2 per pixel, average, then take the square root.	Harshly penalizes big mistakes. Very sensitive to outliers — a few bad pixels can dominate RMSE.
mae	Mean Absolute Error: average of	Predicted - ground_truth	Across all pixels
silog	Scale-Invariant Log Error: combines error in log-space to be insensitive to global scale shifts.	Roughly: take the log of predictions and ground truth, compute variance of the difference, plus the squared mean difference.	Super important for monocular depth: if the entire prediction is off by a constant factor (e.g., twice too big), SILog won't punish it as hard. Focuses more on relative depth shapes than absolute scale.

Table 4: Evaluation Metrics Details

Experiment Results:



Plot 1: Loss Function Plot

Model	d1	abs_rel	rmse	mae	silog
Fine-Tuned	0.01623	1.34804	5.4793	5.3177	0.63528
Original	0.01624	1.34799	5.47911	5.31749	0.63526
Improvement	0	-0.00005	-0.00019	-0.0002	-0.00001

Table 5: Test Data Results

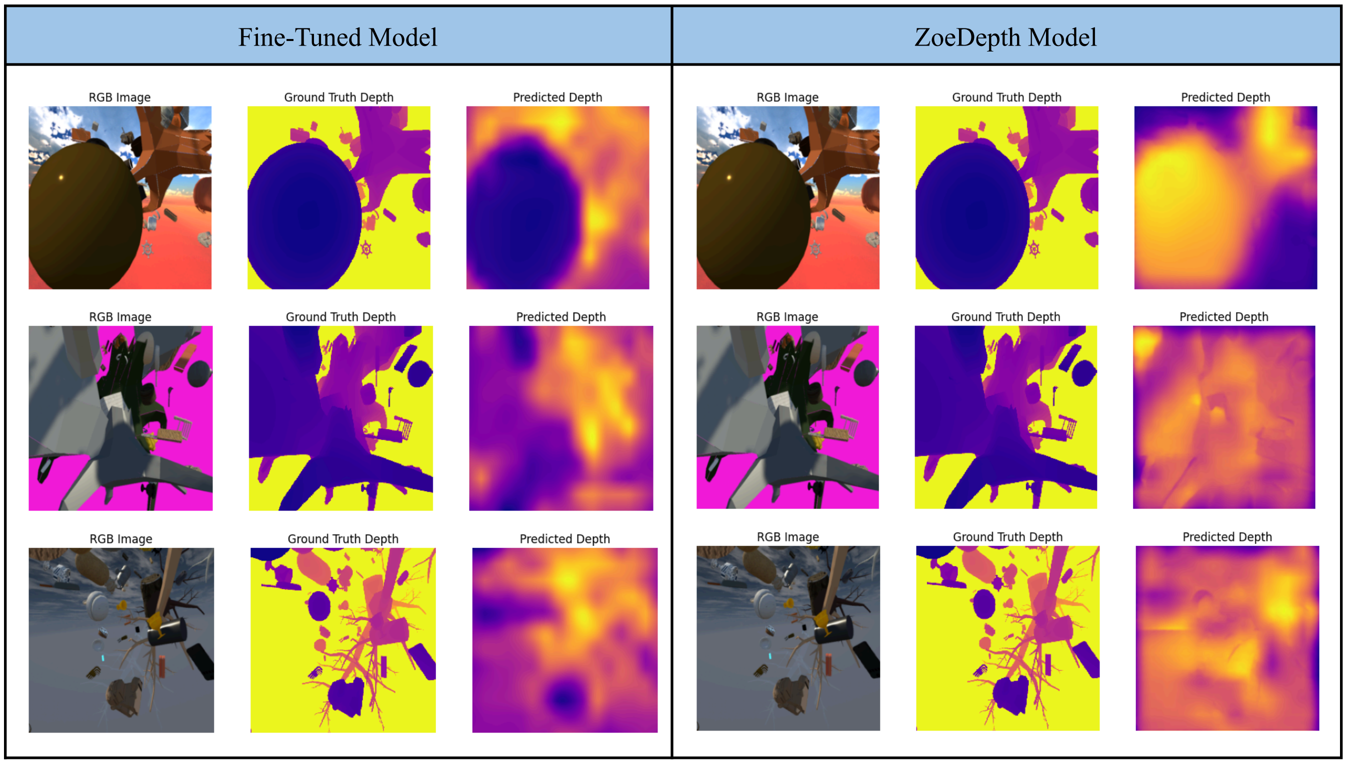


Table 6: Depth Map Prediction Results

Data Analysis

Training Behavior

The training loss curve in Plot 1 shows a consistent decline over the first nine epochs. It started at about 3.2 and stabilized around 0.55 by epoch 9. This is good because this means that the loss function we implemented was working as it should, suggesting that the model was effectively optimizing its parameters and learning from the dataset. We utilized early stopping here which stopped after 5 epochs of no improvements. The low learning rate used and the early stopping were useful to prevent overfitting which was a concern for fine-tuning with synthetic data.

Evaluation Metrics

After running the pre-trained ZoeDepth model and our fine-tuned version of the model on the test set we can see a comparison of the evaluation metrics (seen in Table 5: Test Data Results). Looking at the improvement row it is apparent that there was close to no change. This indicates that fine-tuning did not really have any impact on the model (positive or negative). In essence, the fine-tuned model performs nearly identically to the original ZoeDepth model.

The only real changes that can be seen in the results come from the generated predicted depth maps outlined in Table 6: Depth Map Prediction Results. Note that our predicted depth maps are inverted, this is not an actual issue here as addressed in the Methodology section, it is just a visual flip. Nonetheless what can be observed from the results is that our predicted depth maps are a bit more blurry, but pick up more features as compared to the depths predicted on the pre-trained model are sharper.

The near-zero change in evaluation metrics suggests that the synthetic dataset used for fine-tuning did not introduce sufficiently novel information to modify the model’s behavior in a meaningful way. This may be attributed to factors such as:

1. **Insufficient Dataset:** While the synthetic dataset includes variations in lighting, camera angle, and scene geometry, it may still lack the complexity to make an impact on the models robustness in adapting to new scenes. In essence, the data was potentially too small and too simple.
2. **Frozen Encoder Limitation:** In our implementation we freeze the encoder in an attempt to maintain the strong feature extraction capabilities of the model and only influence the depth detection part. While this is a good idea considering the scope of our dataset, this may have been overkill keeping the entire encoder frozen the whole time. If the new data requires adjustments at the feature level (e.g., recognizing new texture patterns or synthetic artifacts), the frozen backbone would have limited how the model would learn this. Perhaps freezing the encoder in the first few epochs and then unfreezing some would have been better.

Future Directions

Based on the analysis and potential shortcomings future work should:

- Improve upon the dataset by adding more randomness and more data. This would look like adding more prefabs, more materials, more variations in lighting
- As more epochs go one in the fine-tuning, unfreeze parts of the encoder to allow it to adapt and learn from the new dataset such that it gains that crucial information in addition to how to estimate depth.

6 Challenges and Adaptations

Initially, our project planned to fine-tune and compare three monocular depth estimation models: Marigold, HybridDepth, and Distill-Any-Depth. However, during implementation, several challenges led us to adjust the scope and focus exclusively on fine-tuning ZoeDepth. A major technical barrier arose with HybridDepth, which requires focal stack inputs rather than single RGB images. Generating focal stacks would have required creating multiple images at varying focus levels for each scene, significantly increasing the size and complexity of the dataset (from approximately 3,000 samples to potentially over 15,000 samples). This preprocessing step was beyond the resources and time available for the project timeline.

In addition, we encountered difficulties in setting up and integrating the other models. Marigold and Distill-Any-Depth each had their own unique dependencies, training requirements, and environment setups, which made simultaneous management of three separate pipelines impractical given our available time and computational constraints. Ultimately, focusing on fine-tuning a single, strong model, ZoeDepth, was the most practical and strategic decision. ZoeDepth was selected because it works with standard single RGB images, aligns well with our synthetic dataset format, and offers a robust baseline for investigating generalization improvements.

The project faced several challenges within the Unity development environment as well. There was a limited selection of prefabricated outdoor and indoor scenes that met the project requirements, and many available free materials and prefabs were incompatible with the rendering pipeline or did not function as intended. Object placement within the camera's field of view also required careful management to ensure consistency. For processing and quality control purposes, captured images were subsequently resized to 256 × 256 pixels. To accurately capture depth information, it was necessary to develop a custom shader, as existing solutions did not produce satisfactory results.

Further, throughout the development process, several technical obstacles were encountered. Significant time was spent resolving issues related to loading the dataset into the KOA environment, which required deep integration with existing frameworks. Initially we planned on loading the dataset from Hugging Facebut the python dataset loader script would not run which was crucial in setting up the dataset for practical use. We eventually resulted to compressing the images and manually uploading them to KOA where we then defined our Hugging Face dataset loader script at the class referenced in Code Snippet 2. Implementing models from the Hugging Face platform posed additional challenges, particularly in adapting pre-trained architectures to the project's specific requirements. Furthermore, identifying and selecting an appropriate Monocular Depth Estimation (MDE) model involved extensive evaluation to achieve the desired balance of accuracy, computational efficiency, and compatibility with the overall system design.

Considerably, the greatest challenge of all was when putting together the various issues faced and how we overcame them we had to sacrifice design decisions to ensure that the fine-tuning script would work, but at the cost of performance. For instance limited prefabs available in Unity could impact the quality of the dataset and we needed to decrease the batch size to 2 which has its own consequences introducing potential noisy gradients. Nonetheless this reveals potential future ablation studies which work to better overcome and address these issues given more resources.

7 Conclusion

The fine-tuning results demonstrate that while we were able to successfully fine-tune ZoeDepth with a decreasing loss function, the final evaluation metrics remained nearly unchanged from the original pretrained model. This suggests that our synthetic dataset did not effectively improve the model's representations. We predict that this is due to shortcomings in the dataset and diminishing returns of freezing the encoder.

Initially, our project aimed to fine-tune and compare three monocular depth estimation (MDE) models—Marigold, HybridDepth, and Distill-Any-Depth—on a custom synthetic dataset. Adaptations had to be made, and ZoeDepth was ultimately selected as the model of focus because it aligned best with our dataset format (single RGB images), had relatively accessible training and inference pipelines, and served as a strong benchmark for exploring generalization via synthetic fine-tuning. Ultimately, the need to make practical design compromises, such as reducing scene diversity and shrinking batch and image size, may have limited the fine-tuning script's effectiveness.

Nonetheless, these trade-offs highlight promising avenues for future work. With greater access to resources, more diverse and realistic synthetic datasets could be generated, larger batch sizes could be supported, and encoder unfreezing or domain adaptation methods could be explored. These adjustments may improve performance and better investigate the impacts of synthetic fine-tuning for monocular depth estimation.

Appendix

"AdamW — PyTorch 2.7 Documentation." Pytorch.org, 2024, pytorch.org/docs/stable/generated/torch.optim.AdamW.html. Accessed 29 Apr. 2025.

Bhat, S. F., Birkl, R., Wofk, D., Wonka, P., & Müller, M. (2023). Zoedepth: Zero-shot transfer by combining relative and metric depth. arXiv preprint arXiv:2302.12288.

"CosineAnnealingLR — PyTorch 2.7 Documentation." Pytorch.org, 2024, pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.CosineAnnealingLR.html. Accessed 29 Apr. 2025.

Ganj, A., Su, H., & Guo, T. (2024). HybridDepth: Robust Metric Depth Fusion by Leveraging Depth from Focus and Single-Image Priors. arXiv preprint arXiv:2407.18443.

He, X., Guo, D., Li, H., Li, R., Cui, Y., & Zhang, C. (2025). Distill Any Depth: Distillation Creates a Stronger Monocular Depth Estimator. arXiv preprint arXiv:2502.19204.

Hu, X., Zhang, C., Zhang, Y., Hai, B., Yu, K., & He, Z. (2024). Learning to adapt clip for few-shot monocular depth estimation. In Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (pp. 5594-5603).

Ke, B., Obukhov, A., Huang, S., Metzger, N., Daudt, R. C., & Schindler, K. (2024). Repurposing diffusion-based image generators for monocular depth estimation. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 9492-9502).