A Lightweight Convolutional Neural Network model for Object Size Estimations in 2D images

Amanda Ho Shan Rui
amandahsr9@gmail.com

***Abstract* — The ability to estimate object sizes in 2D images accurately have far-reaching implications – from measuring abnormalities such as tumors in medical imaging for effective diagnostics to detecting infrastructure damages for severity assessment and risk management. Using images and videos to aid in object size estimations are often impeded by perspective-induced optical illusions, shadows or the angle of an object. Precisely measuring objects as such, is not always feasible and is often a difficult task. Hence, this project seeks to build and train a deep-learning model to detect and estimate the size of objects in 2D images, with a focus on lightweight models that balance speed and performance.**

I. Introduction

Estimating object size is a task that cuts across multiple industries, where deep learning has proved to be especially useful in situations where physical sensors or measurements are impractical or difficult to leverage on. In a 2019 study that researched on methods to predict survival time of high-grade gliomas patients, a multi-channel 3D convolutional neural network (CNN) was proposed to extract predictive features such as tumor sizes [1]. The 3D CNN achieved a high accuracy of 90.66% in predicting patient survival time, showing the importance of accurate tumor size detection in medical images in helping patient prognosis and treatment. Deep learning has also been utilized successfully to characterize critical infrastructure damage in bridges and transportation networks. The Segment Anything Model (SAM) was used by Kopiika et al. to segment bridges and transport networks in post-disaster images in Ukraine as part of its multi-tiered approach to identifying important structures for damage asseessment [2], which included utilizing other deep learning models like Stable Diffusion to enhance image quality for better segmentation. The authors found that deep-learning models were crucial in helping detect damages more efficiently, aiding in rapid decision-making during disaster recovery. These instances highlight the value of deploying computationally efficient deep learning models that balance speed and performance for object size estimations in 2D images.

II. Literature review

One of the major challenges in building accurate object size estimation models is the limited availability of training datasets that provide diversity in object types and 3D annotations of object sizes. Current datasets with 3D annotations of object sizes include the ObjectNet3D dataset, containing 90,127 images with 100 object categories for training and testing [3]. Although the dataset contains a wide variety of objects alongside their x and y coordinates to compute width and height, it lacks true object size annotations and requires further computation to convert pixels to physical units for object size estimation. The Falling Things dataset was published by NVIDIA to address the lack of available images for object detection related tasks, synthetically generated to be photorealistic and annotated with true object sizes [4]. The dataset contains 60,000 annotated images of 21 household object types, but is not easily accessible due to its large file size and storage on a google drive that necessitates manual downloading. The Common Objects in Context (MS COCO) is another popular dataset to utilize for object characterization due to its large size of over 330,000 images spanning over 80 object categories focusing on well-known common objects [5]. However, the dataset also lacks physical units for its object size annotations, requiring additional computation to convert pixels to physical units. Lastly, the KITTI Vision Benchmark Suite (KITTI) dataset is geared towards improving autonomous driving models, containing objects such as cars, cyclists and pedestrians in its images [6]. While this dataset has a smaller size of 7,481 images, it boasts comprehensive annotations of object types, object dimensions and true 3D object sizes.

While the number of suitable datasets is limited, many model architectures suit the task of object size estimation. Any CNN regression model designed to extract visual features from 2D images to predict 3D sizes can be utilized for the task at hand, where deeper models typically provide better performance but take longer to train. Many CNN models have evolved to produce good performance on a relatively small number of parameters, designed to be repurposed for portable applications that do not have sufficient computational resources to host state-of-the-art models. The MobileNetV2 model was developed by Google for utility on mobile and embedded vision applications, integrating depthwise convolutions to reduce the number of computational operations executed, and skip connections to connect across layers for faster convergence [7, Fig. 1]. The model contains only approximately 3.4 million parameters but achieves an accuracy rate of 72% on ImageNet, comparable to the much more complex ResNet-50 model that achieved 76%.

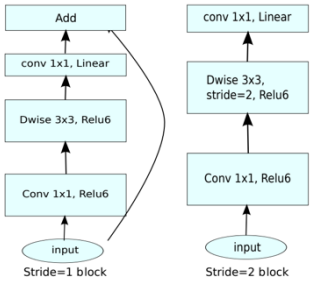

Fig. 1: Model architecture of the MobileNetV2 model.

Tiny YOLOv3 (You Only Look Once) is a Single Shot Detector that predicts bounding boxes and classes in a single pass of the network [8, Fig. 2], making it faster and more efficient than other networks. The model creatively optimizes its speed by using default bounding boxes, focusing on predicting the offsets of the boxes to avoid computing absolute coordinates from scratch. While Tiny YOLOv3 does not outperform previous YOLO-based models in terms of accuracy, it runs almost 4x faster than its predecessor YOLOv3.

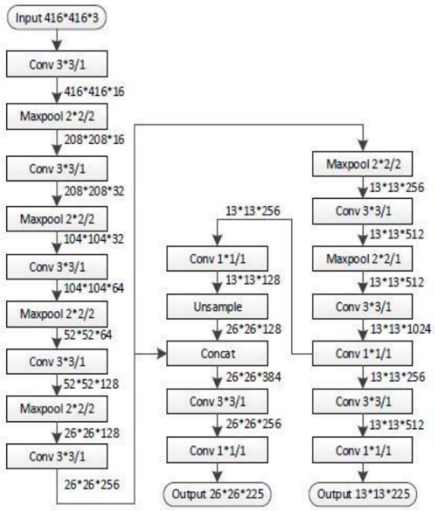

Fig. 2: Model architecture of the Tiny YOLOv3 model.

Lastly, the ShuffleNetV2 makes use of skip connections, depthwise convolutions and channel shuffling as part of its unique architecture [9, Fig. 3]. Shuffling permutates the channels of feature maps, enabling interactions between different groups of channels to facilitate more diverse feature learning. The model consists of only 2.3M parameters but boasts an accuracy of 70.9% on ImageNet.

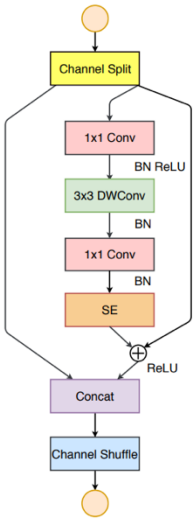

Fig. 3: Model architecture of the ShuffleNetV2 model.

III. Dataset and Model Architecture

Among the 4 available datasets, the KITTI dataset was the only dataset available that provided 3D object size annotations in physical units and was easily accessible for training from TensorFlow. It was used to train and evaluate the ShuffleNetV2 model, chosen due to its relatively lightweight architecture and high accuracy on ImageNet among the three models researched.

The KITTI dataset contains multi-object images alongside the object type, the coordinates of bounding boxes (ymin, xmin, ymax, xmax), and the true dimensions of objects in meters (height, width, depth). To simplify the complexity of the training and evaluation process, the dataset was preprocessed to generate single-object images for training and testing of ShuffleNetV2. Images are cropped based on bounding box labels to include only one object, before being resized to original image sizes of (224, 224, 3) to standardize input sizes. They are then normalized to standardize pixel values with their corresponding object dimension labels extracted, before being randomly split into training, validation and testing datasets. In total, 6,347 training images, 423 validation images and 711 testing images were obtained for training and evaluation of ShuffleNetV2.

The ShuffleNetV2 model was originally designed for classification tasks, thus requiring modifications to the model architecture to include a regression instead of a classification output layer. The input layer was configured to correspond to KITTI image sizes of (224, 224, 3), and the final output layer was configured to generate height, width and depth classes for object size estimations. The remaining ShuffleNetV2 regression model architecture retained the original model's design to provide as close a representation to the original model design.

IV. Base Model Training

Due to computational resource constraints, the ShuffleNetV2 regression model was trained using the KITTI training and validation dataset on 15 epochs and a batch size of 64. Training, validation and testing datasets were cropped to single-object images to reduce the complexity of the size estimation task, before they were normalized and resized to original image dimensions to standardize input image sizes. The Mean Squared Error (MSE) function was utilized to compute loss and the Adam optimizer was chosen to adjust model weights and biases during training. Training of the regression model took approximately 21 minutes per epoch, or 5.25 hours of training across 15 epochs.

Despite training on only 15 epochs and a small set of images, the model was able to fine-tune its model weights significantly overtime. A gradual decrease in learning curve on the training dataset over the epochs as shown in Fig. 4 indicates performance improvement in predicting object sizes that are closer to the true sizes. A fluctuating learning curve on the validation dataset suggest overfitting, suggesting that adopting a smaller learning rate might help to mitigate poor generalization to images beyond the training dataset. Additionally, both training and validation learning curves have not yet plateaued, suggesting that more rounds of training epochs might further improve model performance in object size estimations.

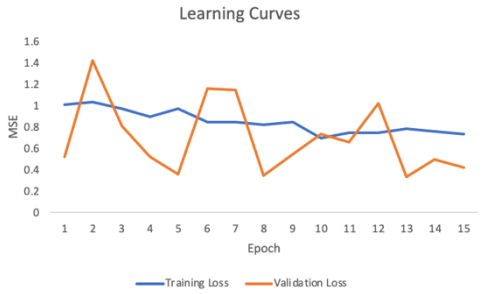

Fig. 4: Learning curves on KITTI training and validation datasets.

V. Base Model Evaluation and Results

The model was evaluated using the KITTI testing dataset on several regression evaluation metrics including Mean Squared Error (MSE), Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE) and R-Squared (R^2). The MSE measures the average squared differences between predicted and target values in the dataset, where a lower MSE indicates better model performance. It is calculated as follows,

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \quad (1)$$

where N is the number of samples, y_i is the target value and $\hat{y}_i$ is the predicted value. The MAE measures the absolute differences between predicted and target values in the dataset, where a lower MAE indicates better model performance. It is calculated as follows,

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \quad (2)$$

where N is the number of samples, y_i is the target value and $\hat{y}_i$ is the predicted value. The MAPE measures the average absolute percentage differences between predicted and target values in the dataset, where a lower MAPE indicates better model performance. It is calculated as follows,

$$MAPE = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100 \quad (3)$$

Where N is the number of samples, y_i is the target value and $\hat{y}_i$ is the predicted value. Lastly, the R^2 measures the variance in target values that can be explained by predicted values in the dataset, where a higher R^2 indicates better model performance. It is calculated as follows,

$$R^2 = 1 - \frac{SS_{res}}{SS_{total}} \quad (3)$$

where SS_{res} is the sum of squared residuals between predicted and true values, and SS_{total} is the total sum of squared residuals between target values and their mean.

The results obtained by the ShuffleNetV2 regression model across the four evaluation metrics are shown in Tab. 1.

Tab. 1: MAE, MSE, MAPE and R^2 values across object height, width and depth obtained by the ShuffleNetV2 regression model on the KITTI dataset.

Dimension	MAE	MSE	MAPE (%)	R^2
Height	0.2140	0.1221	12.1404	0.0430
Width	0.2746	0.1207	22.5665	0.1886
Depth	1.3850	12.8540	46.4655	0.1136

The model achieved relatively low MAE, MSE, MAPE and R^2 values across its predictions for object heights and widths. This suggests that the ShuffleNetV2 regression model is able to predict 2D object dimensions to a high precision but failed to learn most of the variation in the values. This might be caused by a lack of diversity in object dimensions in the training dataset, as the KITTI dataset contained object types such as cars and cyclists that tend to have standardized dimensions. The regression model likely learned to predict 2D dimensions based on average dimensions, resulting in predictions that closely resemble target values but do not explain differences between different objects.

While the model had a high precision in detecting 2D dimensions, it is unsurprising that it obtained much higher MAE, MSE and MAPE values for object depths. The MAPE in particular, suggests that the model on average, predicted depth values that had differences close to 50% of the actual depth. This poor performance highlights the inherent difficulty in predicting depth from 2D images, which requires spatial information that is lost when images are scaled down in dimension. This finding corroborates with conclusions from current literature focusing on depth estimations from images, as there are multiple ways for the same 3D object to be rendered in 2D plane and be further distorted by occlusions and perspectives [10].

VI. Improving 3D Object Size Estimation

Convolutional Block Attention Module (CBAM) is a lightweight attention module that can be easily added into various types of model architecture with negligible computational overhead [11]. The module consists of a channel attention map that uses average and max pooling to extract the most distinctive features from images, and a spatial attention map on the channel axis to identify information-rich regions [11, Fig. 5]. Models integrated with CBAM has been shown to outperform its base models and SE-integrated models significantly on various benchmarking datasets such as the ImageNet-1K, MS COCO and VOC 2007 on GFLOPs, Top-1 and Top-5 error rate metrics. These suggest that the module can be used for a variety of recognition tasks beyond image classification, and highlights its adaptability in improving performance in a variety of model architectures.

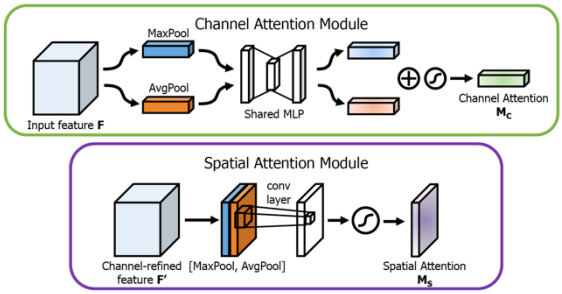

Fig. 5: Channel and Spatial Attention Module architecture in CBAM.

Additionally, the module has been shown to facilitate models in finding regions of interest and feature extraction more effectively, where images predicted with CBAM models had regions with more comprehensively encapsulated objects of interest [11, Fig. 6]. CBAM's lightweight architecture and it's negligible computational overhead makes it a potential candidate to improve ShuffleNetV2's 3D size estimations by providing better feature extraction and identification of more important regions.

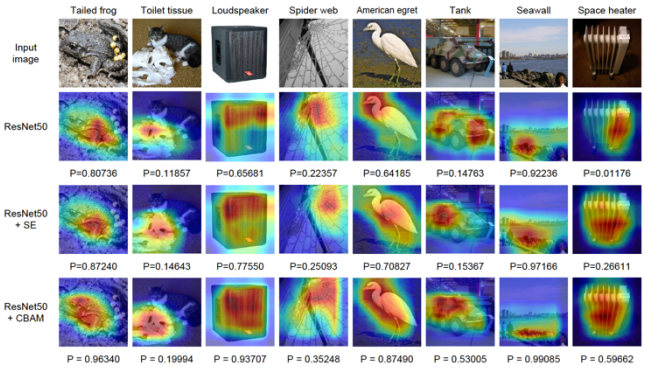

Fig. 6: Grad-CAM masks of base, base+SE, base+CBAM output.

Beyond model architecture improvements, image enhancement techniques can also be employed to improve image quality for better subsequent object size estimations. Image denoising is one such method that reduces graininess and decolorization in images by removing noise caused during storage, transmission or storage [12]. Denoising is typically based on three kinds of techniques, namely spatial filtering, temporal accumulation and machine learning or deep learning reconstruction [13]. Spatial filtering modifies specific areas of the image by utilizing neighboring pixels that are similar in color and brightness. However, this technique typically introduces blurriness and visual imperfections to the image, where modified areas can be visibly distinguished from unaltered areas. Temporal accumulation leverages on previous frames to identify and correct any artifacts in current frames. Unlike spatial filtering, temporal accumulation does not introduce blurriness or additional artifacts to the enhanced image. Lastly, machine learning or deep learning reconstruction leverage on neural networks that have been trained on a diversity of noisy images to reconstruct the true signal.

As the KITTI training dataset used to train the ShuffleNetV2 model was cropped into single object images and resized to original image dimensions, training images were highly pixelated and significantly blurred compared to the original training dataset. Considering the ease of implementation and computational overhead, employing the spatial filtering method might greatly improve the visual details of image quality for better object size estimations. The Non-Local Means (NLMeans) algorithm is one such method that denoises images by replacing pixel colors with the average color of similar pixels [14, Fig. 7].

Fig. 7: Original image (left), noisy image (middle), NLMeans output (right).

Finally, utilizing convolutional kernels that are able to enhance specific aspects of image quality is another way to improve object size estimation performance without adding significant computational overhead to the model. Djik and Croon examined various neural networks in an autonomous driving scenario, aiming to identify the image characteristics that are most important in helping networks predict depth in 2D images [15]. From object position and apparent sizes to the coordinates of ground contact point to the object, the study found that even when object interiors are removed, networks were still able to estimate depth effectively for most objects. This suggests that object edges are the primary source of guidance for networks to estimate depth more accurately. As such, kernels that focus on edge enhancements might help the models distinguish objects of interest more effectively and improve object size prediction, particularly in the worst performing object depth class where the base model achieved an MAPE of 46.4655%.

VII. Improved Model Training

The KITTI training, validation and testing dataset followed the same cropping and resizing processes as that of the base model training in section IV, with the addition of denoising and edge enhancement convolutional kernels applied before resizing for the training dataset. The NLMeans Denoising algorithm was implemented to denoise the images using the *fastNlMeansDenoisingColored* function available through the OpenCV python library. A standard 3x3 edge enhancement kernel consisting of -1 values and a 9 value core was implemented after denoising to emphasize object edges in images. After enhancements, images were found to be less grainy, contain less fluctuations in color hues across pixels and more prominent object edges as shown in Fig. 8. In addition to image enhancements, the base ShuffleNetV2 model was modified to include the CBAM after each convolutional layer using a 7x7 kernel for the spatial attention module.

Fig. 8: Original image (left), denoised + edge-enhanced image (right).

In total, two variations of the base model were trained separately for evaluation. The CBAM-integrated model is trained on the same unenhanced KITTI training and validation dataset with the same training parameters used for the base model in section IV. Training of the CBAM-integrated model took approximately 25 minutes per epoch, or 6.18 hours of training across 15 epochs. The model was able to fine-tune its model weights significantly overtime. A gradual decrease in learning curve on the training and validation dataset over the epochs as shown in Fig. 9 indicates performance improvement in predicting object sizes that are closer to the true sizes. The plateau in both training and validation loss suggests that 15 rounds of epoch might be sufficient for the CBAM-integrated model to minimize its loss function.

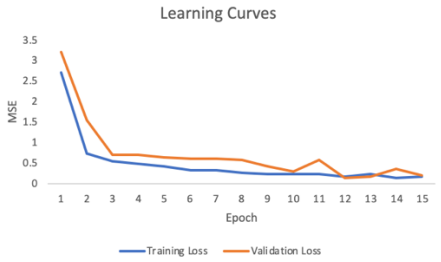

Fig. 9: Learning curves of CBAM-integrated model on unenhanced KITTI training and validation datasets.

Separately, a CBAM-integrated model is trained on the enhanced KITTI training dataset with the same training parameters used for the base model in section IV. Training of this model took approximately 27.5 minutes per epoch, or 6.92 hours of training across 15 epochs. The model was also able to fine-tune its model weights significantly overtime. A gradual decrease in learning curve on the training and validation dataset over the epochs as shown in Fig. 10 indicates performance improvement in predicting object sizes that are closer to the true sizes. The plateau in both training and validation loss also indicates that 15 rounds of epoch might be sufficient for this model to minimize its loss function.

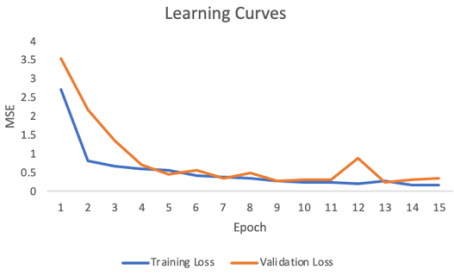

Fig. 10: Learning curves of CBAM-integrated model on enhanced KITTI training and validation datasets.

VIII. Improved Model Evaluation and Results

The two additional models were evaluated on the same unenhanced KITTI testing dataset as the base model evaluation to generate testing results that were comparable. Using the

MSE, MAE, MAPE and R^2 evaluation regression metrics, testing results for the base model, CBAM-integrated model with unenhanced training dataset, and CBAM-integrated model with enhanced training dataset were obtained. Results for object height, width and depth estimations are shown in Tab. 2, Tab. 3 and Tab. 4 respectively.

TAB. 2: MAE, MSE, MAPE and R^2 values for object height obtained by the base model, CBAM model and CBAM model + image enhancements.

Model	MAE	MSE	MAPE (%)	R^2
Base	0.2140	0.1221	12.1404	0.0430
Base + CBAM	0.2304	0.1584	12.6755	0.2415
Base + CBAM + Image enhancements	0.2069	0.1333	11.4622	0.3516

TAB. 3: MAE, MSE, MAPE and R^2 values for object width obtained by the base model, CBAM model and CBAM model + image enhancements.

Model	MAE	MSE	MAPE (%)	R^2
Base	0.2746	0.1207	22.5665	0.1886
Base + CBAM	0.1777	0.0750	13.1368	0.4961
Base + CBAM + Image enhancements	0.1683	0.0451	12.1623	0.6170

TAB. 4: MAE, MSE, MAPE and R^2 values for object depth obtained by the base model, CBAM model and CBAM model + image enhancements.

Model	MAE	MSE	MAPE (%)	R^2
Base	1.3850	12.8540	46.4655	0.1136
Base + CBAM	1.1901	14.1387	33.3469	0.0250
Base + CBAM + Image enhancements	1.0518	15.5069	27.1680	0.0693

The CBAM-integrated model with unenhanced training dataset performed worse than the base model when predicting object height, obtaining slightly higher MAE, MSE and MAPE values. Surprisingly, it showed an increase in R^2 value by almost 0.2. These results suggest that the CBAM-integrated model was able to capture more of the underlying pattern that contributed to variation in object height, aided by CBAM's ability to extract better features and identify more information-dense regions in images. However, the model's ability to better learn variation in object heights might have resulted in a trade-off with the ability to predict them more accurately within the same number of training iterations. When predicting object widths, the model showed a marked improvement across MAE, MSE, MAPE and R^2 values, indicating width estimates that were closer to true widths while also capturing more of the underlying variation. These results indicate that CBAM might have extracted features and regions that were more crucial to estimating widths, more so than for object height or depth. Lastly, the CBAM-integrated model also achieved higher MAE and MAPE but lower MSE and R^2 values compared to the base model when estimating object depth. Interestingly, this suggests that the model has optimized predictions that reduce smaller differences in estimated and target depths at the cost of making the occasional prediction that had large differences to target values. A lower R^2 also indicates that the addition of CBAM resulted in the model learning less of the variation in object depth, perhaps to prioritize feature and region extraction that were more predictive of height and width.

With the addition of image enhancements, the CBAM-integrated model showed improved performance with lower MAE, MSE and MAPE, and higher R^2 value for object height and width. On the other hand, the model obtained a lower MAE, MAPE and R^2 value but higher MSEt for object depth. These results indicate that image enhancements facilitated improvements in predicting object height and width but not necessarily for object depth estimations, which gives further evidence that features and regions that were more predictive of height and width were prioritized at the cost of those for depth. It is noted that image enhancements did help with producing depth estimates that were closer to target values, although the model struggled to capture underlying patterns in depth values.

IX. CONCLUSION

This paper has provided a comprehensive review of current research and development surrounding object size estimations, including available training datasets that contain 3D object dimensions and popular computationally efficient CNN models that are suitable for object size estimations. In particular, a regression model using the ShuffleNetV2 architecture was trained and tested using the KITTI dataset. The model was found to predict 2D object dimensions with high precisions but struggle with depth estimations, highlighting the inherent difficulty in predicting 3D dimensions from 2D images. Further improvements regarding model architecture and image enhancement techniques including the addition of CBAM, denoising and edge enhancements for training images were discussed and evaluated for their performance. The addition of these improvements was found to improve size prediction performance of the base model by facilitating better feature extraction and providing training images that had clearer demarcation of objects for prediction. 3D object size prediction on 2D imagesa remains a challenging task, particularly in the area of object depth prediction. Follow-up studies can be done on evaluating 3D size prediction performance using video datasets for training, as videos provide more spatial information regarding shapes and positions to enhance depth estimations.

Additionally, retraining and evaluating the ShuffleNetV2 model on another dataset with more object type diversity might enhance model performance to better learn patterns within object sizes.

REFERENCES

[1] Nie, D. *et al.* (2019) 'Multi-channel 3D deep feature learning for survival time prediction of brain tumor patients using multi-modal neuroimages', *Scientific Reports*, 9(1). doi:10.1038/s41598-018-37387-9.

[2] Kopiika, N. *et al.* (2025) 'Rapid post-disaster infrastructure damage characterisation using remote sensing and deep learning technologies: A tiered approach', *Automation in Construction*, 170, p. 105955. doi:10.1016/j.autcon.2024.105955.

[3] Xiang, Y. *et al.* (2016) 'ObjectNet3D: A large scale database for 3D object recognition', *Lecture Notes in Computer Science*, pp. 160–176. doi:10.1007/978-3-319-46484-8_10.

[4] Tremblay, J., To, T. and Birchfield, S. (2018) 'Falling things: A synthetic dataset for 3D object detection and pose estimation', *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)* [Preprint]. doi:10.1109/cvprw.2018.00275.

[5] Lin, T.-Y. *et al.* (2014) 'Microsoft Coco: Common Objects in Context', *Lecture Notes in Computer Science*, pp. 740–755. doi:10.1007/978-3-319-10602-1_48.

[6] Geiger, A., Lenz, P. and Urtasun, R. (2012) 'Are we ready for autonomous driving? The Kitti Vision Benchmark Suite', *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3354–3361. doi:10.1109/cvpr.2012.6248074.

[7] Sandler, M. *et al.* (2018) 'MobileNetV2: Inverted residuals and linear bottlenecks', *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4510–4520. doi:10.1109/cvpr.2018.00474.

[8] Adarsh, P., Rathi, P. and Kumar, M. (2020) 'Yolo v3-Tiny: Object Detection and recognition using one stage improved model', *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)* [Preprint]. doi:10.1109/icaccs48705.2020.9074315.

[9] Ma, N. *et al.* (2018) 'ShuffleNet V2: Practical guidelines for efficient CNN Architecture Design', *Lecture Notes in Computer Science*, pp. 122–138. doi:10.1007/978-3-030-01264-9_8.

[10] Mertan, A., Duff, D.J. and Unal, G. (2022) 'Single Image Depth Estimation: An overview', *Digital Signal Processing*, 123, p. 103441. doi:10.1016/j.dsp.2022.103441.

[11] S. Woo, J. Park, J.-Y. Lee, and I. S. Kweon, "CBAM: Convolutional Block Attention Module," Lecture Notes in Computer Science, pp. 3–19, 2018. doi:10.1007/978-3-030-01234-2_1

[12] R. Rajni and A. Anutam, "Image denoising techniques - an overview," International Journal of Computer Applications, vol. 86, no. 16, pp. 13–17, Jan. 2014. doi:10.5120/15069-3436

[13] J. Kim, "What is denoising?," NVIDIA Blog, https://blogs.nvidia.com/blog/what-is-denoising/ (accessed May 11, 2025).

[14] Buades, B. Coll, and J.-M. Morel, "Non-local means denoising," Image Processing On Line, vol. 1, pp. 208–212, Sep. 2011. doi:10.5201/ipol.2011.bcm_nlm

[15] T. Van Dijk and G. De Croon, "How do neural networks see depth in single images?," 2019 IEEE/CVF International Conference on Computer Vision (ICCV), pp. 2183–2191, Oct. 2019. doi:10.1109/iccv.2019.00227

Appendix A

The code snippet below is used to preprocess an unenhanced and enhanced KITTI training, validation and testing datasets:

```
from numpy import np
from typing import Dict, List
import cv2
import tensorflow_datasets as tfds
import tensorflow as tf
from tensorflow.data.experimental import cardinality


class KittiDataset:
    def __init__(self):
        self.input_img_shape: List[int, int] = (224, 224, 3)
        self.img_dtype = tf.int32
        self.batch_size: int = 64
        self.shuffle_seed: int = 15

        self.training_dataset: List[Dict] = None
        self.validation_dataset: List[Dict] = None
        self.testing_dataset: List[Dict] = None
        self.enhanced_training_dataset: List[Dict] = None

        self.load_dataset()
        self.process_datasets()

    def load_dataset(self) -> None:
        """
        Loads Falling Things training and testing datasets.
        """
        self.training_dataset = tfds.load("kitti", split="train", as_supervised=False)
        self.validation_dataset = tfds.load(
            "kitti", split="validation", as_supervised=False
        )
        self.testing_dataset = tfds.load("kitti", split="test", as_supervised=False)

    def image_enhance(img):
        """
        Denoises and applies filter to enhance edges in image.
        """
        deblurred_img = cv2.fastNlMeansDenoisingColored(img.numpy(), None, 11, 11, 5, 7)
        sharpen_kernel = np.array([[-1, -1, -1], [-1, 9, -1], [-1, -1, -1]])
        sharpen_img = cv2.filter2D(deblurred_img, -1, sharpen_kernel)

        return sharpen_img
```

```
def process_img(self, img: np.ndarray, enhanced: bool):
    """
    Crops and normalizes img to single objects, and returns object size annotations.
    """
    # Extract first object if more than one in img
    ori_height = tf.cast(tf.shape(img["image"])[0], tf.float32)
    ori_width = tf.cast(tf.shape(img["image"])[1], tf.float32)

    # Convert pixel coordinates to img sizes for first annotated obj
    obj_bboxes = img["objects"]["bbox"]
    bounding_boxes = (
        obj_bboxes[0][0] * ori_height,
        obj_bboxes[0][1] * ori_width,
        obj_bboxes[0][2] * ori_height,
        obj_bboxes[0][3] * ori_width,
    )

    # Extract object dimesnions for first annotated obj
    obj_sizes = img["objects"]["dimensions"]
    sizes_3d = tf.stack([obj_sizes[0][0], obj_sizes[0][1], obj_sizes[0][2]])

    # Crop img to single object, and resize/normalize
    cropped_img = tf.image.crop_to_bounding_box(
        img["image"],
        offset_height=tf.cast(bounding_boxes[0], self.img_dtype),
        offset_width=tf.cast(bounding_boxes[1], self.img_dtype),
        target_height=tf.cast(
            tf.maximum(bounding_boxes[2] - bounding_boxes[0], 1.0), self.img_dtype
        ),
        target_width=tf.cast(
            tf.maximum(bounding_boxes[1] - bounding_boxes[3], 1.0), self.img_dtype
        ),
    )

    if enhanced:
        # Apply enhancements
        img_enhanced = tf.py_function(
            func=self.image_enhance, inp=[cropped_img], Tout=tf.uint8
        )
        img_enhanced.set_shape(cropped_img.shape)
        cropped_img = img_enhanced

    norm_img = tf.image.resize(cropped_img, list(self.input_img_shape[:-1])) / 255.0
    if enhanced:
        # Set image shape
        norm_img.set_shape(self.input_img_shape)

    return norm_img, sizes_3d
```

```
def process_datasets(self) -> None:
    """
    Processes training, validation and testing datasets for object size estimations.
    """
    # Process enhanced datasets
    training_dataset = self.training_dataset.map(
        lambda x: self.process_img(x, False)
    )
    training_dataset = training_dataset.cache()
    total_samples = cardinality(training_dataset).numpy()
    training_dataset = training_dataset.shuffle(
        buffer_size=total_samples, seed=self.shuffle_seed
    )
    training_dataset = training_dataset.batch(self.batch_size)

    validation_dataset = self.validation_dataset.map(
        lambda x: self.process_img(x, False)
    )
    validation_dataset = validation_dataset.cache()
    total_samples = cardinality(validation_dataset).numpy()
    validation_dataset = validation_dataset.shuffle(
        buffer_size=total_samples, seed=self.shuffle_seed
    )
    validation_dataset = validation_dataset.batch(self.batch_size)

    testing_dataset = self.testing_dataset.map(lambda x: self.process_img(x, False))
    testing_dataset = testing_dataset.cache()
    total_samples = cardinality(testing_dataset).numpy()
    testing_dataset = testing_dataset.shuffle(
        buffer_size=total_samples, seed=self.shuffle_seed
    )
    testing_dataset = testing_dataset.batch(self.batch_size)

    # Process enhanced training dataset
    enhanced_training_dataset = self.training_dataset.map(
        lambda x: self.process_img(x, True)
    )
    enhanced_training_dataset = enhanced_training_dataset.cache()
    total_samples = cardinality(enhanced_training_dataset).numpy()
    enhanced_training_dataset = enhanced_training_dataset.shuffle(
        buffer_size=total_samples, seed=self.shuffle_seed
    )
    enhanced_training_dataset = enhanced_training_dataset.batch(self.batch_size)

    self.training_dataset = training_dataset
    self.validation_dataset = validation_dataset
    self.testing_dataset = testing_dataset
    self.enhanced_training_dataset = enhanced_training_dataset
```

Appendix B

The code snippet below is used to generate a CBAM architecture:

```
import tensorflow as tf
from keras.models import Sequential
from keras.layers import (
    Dense,
    Conv2D,
    GlobalAveragePooling2D,
    GlobalMaxPooling2D,
    Concatenate,
    Multiply,
)


class CBAM(tf.keras.layers.Layer):
    def __init__(self, reduction_ratio=16, kernel_size=7):
        super(CBAM, self).__init__()
        self.reduction_ratio = reduction_ratio
        self.kernel_size = kernel_size
        self.channel_activation_function = "relu"
        self.spatial_activation_function = "sigmoid"

    def build(self, input_shape):
        channels = input_shape[-1]

        # Channel Attention Block
        self.shared_mlp = Sequential(
            [
                Dense(
                    channels // self.reduction_ratio,
                    activation=self.channel_activation_function,
                ),
                Dense(channels),
            ]
        )
        self.global_avg_pool = GlobalAveragePooling2D(keepdims=True)
        self.global_max_pool = GlobalMaxPooling2D(keepdims=True)
        self.sigmoid = tf.keras.activations.sigmoid

        # Spatial Attention Block
        self.conv = Conv2D(
            1,
            kernel_size=self.kernel_size,
            padding="same",
            activation=self.spatial_activation_function,
        )
```

```
def call(self, x):
    # Channel Attention
    avg_pool = self.global_avg_pool(x)
    max_pool = self.global_max_pool(x)
    avg_out = self.shared_mlp(avg_pool)
    max_out = self.shared_mlp(max_pool)
    channel_attention = self.sigmoid(avg_out + max_out)
    x = Multiply()([x, channel_attention])

    # Spatial Attention
    avg_pool = tf.reduce_mean(x, axis=-1, keepdims=True)
    max_pool = tf.reduce_max(x, axis=-1, keepdims=True)
    concat = Concatenate(axis=-1)([avg_pool, max_pool])
    spatial_attention = self.conv(concat)
    x = Multiply()([x, spatial_attention])

    return x
```

Appendix C

The code snippet below is used to generate a ShuffleNetV2 architecture, with parameters for CBAM and enhanced training:

```
from dataset import KittiDataset
from channelshuffle import ChannelShuffle
from keras.layers import (
    Input,
    Dense,
    Conv2D,
    BatchNormalization,
    GlobalAveragePooling2D,
    DepthwiseConv2D,
    ReLU,
    Add,
    Concatenate,
)
from keras.models import Model
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from typing import List
import numpy as np
from tqdm import tqdm
from CBAM import CBAM

class ShuffleNetV2:
    def __init__(self, CBAM_status: bool, train_enhanced: bool):
        self.num_output_class: int = 3
        self.loss_function: str = "mse"
        self.optimizer: str = "adam"
        self.eval_metrics: List[str] = ["mae"]
        self.dataset: KittiDataset = KittiDataset()

        self.depthwise_kernel_size: int = 3
        self.conv_kernel_size: int = 1
        self.padding: str = "same"
        self.activation_function = "relu"

        self.layer1_filters: int = 24
        self.layer1_kernel_size: int = 3
        self.layer1_strides: int = 2
        self.block2_filters: int = 116
        self.block2_strides: int = 1
        self.block3_filters: int = 232
        self.block3_strides: int = 2
        self.dense4_units: int = 128
        self.CBAM_status: bool = CBAM_status

        self.model = self.initialize_model()
```

```
    # Training/Evaluation metrics
    self.num_epochs: int = 15
    self.trained_model = None
    self.train_enhanced = train_enhanced

def initialize_shufflenet_block(self, inputs, filters, stride):
    """
    Returns a single ShuffleNetV2 block.
    """
    x = DepthwiseConv2D(
        kernel_size=self.depthwise_kernel_size, strides=stride, padding=self.padding
    )(inputs)
    x = BatchNormalization()(x)
    x = ReLU()(x)

    x = Conv2D(
        filters=filters // 2,
        kernel_size=self.conv_kernel_size,
        padding=self.padding,
    )(x)
    x = BatchNormalization()(x)

    x = ChannelShuffle()(x)

    if stride == 1 and inputs.shape[-1] == filters:
        if self.CBAM_status:
            shortcut = CBAM()(inputs)
        x = Add()([x, inputs])

    else:
        shortcut = DepthwiseConv2D(
            kernel_size=self.depthwise_kernel_size,
            strides=stride,
            padding=self.padding,
        )(inputs)
        shortcut = BatchNormalization()(shortcut)
        shortcut = Conv2D(
            filters=filters // 2,
            kernel_size=self.conv_kernel_size,
            padding=self.padding,
        )(shortcut)
        shortcut = BatchNormalization()(shortcut)
        if self.CBAM_status:
            shortcut = CBAM()(shortcut)
        x = Concatenate()([x, shortcut])

    x = ReLU()(x)

    return x
```

```
def initialize_model(self):
    """
    Initializes the ShuffleNetV2 regression model.
    """
    inputs = Input(shape=self.dataset.input_img_shape)

    # Layer 1
    x = Conv2D(
        self.layer1_filters,
        kernel_size=self.layer1_kernel_size,
        strides=self.layer1_strides,
        padding=self.padding,
    )(inputs)
    x = BatchNormalization()(x)
    x = ReLU()(x)

    # Block 2
    x = self.initialize_shufflenet_block(
        x, filters=self.block2_filters, stride=self.block2_strides
    )

    # Block 3
    x = self.initialize_shufflenet_block(
        x, filters=self.block3_filters, stride=self.block2_strides
    )

    # Output layer
    x = GlobalAveragePooling2D()(x)
    x = Dense(self.dense4_units, activation=self.activation_function)(x)
    outputs = Dense(self.num_output_class)(x)

    model = Model(inputs, outputs)

    model.compile(
        loss=self.loss_function, optimizer=self.optimizer, metrics=self.eval_metrics
    )

    print("Model architecture:")
    model.summary()

    return model
```

```
def train_model(self) -> None:
    """
    Trains model using training dataset and training parameters.
    """
    if self.train_enhanced:
        self.trained_model = self.model.fit(
            self.dataset.enhanced_training_dataset,
            epochs=self.num_epochs,
            validation_data=self.dataset.validation_dataset,
        )
    else:
        self.trained_model = self.model.fit(
            self.dataset.training_dataset,
            epochs=self.num_epochs,
            validation_data=self.dataset.validation_dataset,
        )

def mean_absolute_percentage_error(self, y_true, y_pred):
    """
    Calculates MAPE, handling zero values in y_true to avoid division by zero.
    """
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    non_zero_mask = y_true != 0
    if np.any(non_zero_mask):
        return (
            np.mean(
                np.abs(
                    (y_true[non_zero_mask] - y_pred[non_zero_mask])
                    / y_true[non_zero_mask]
                )
            )
            * 100
        )
    else:
        return np.inf

def evaluate_model_metrics(self):
    """
    Returns MAE, MSE, MAPE and R-squared values of output classes.
    """
    true_vals = []
    pred_vals = []
```

```
# Extract predictions
print("Collecting predictions...")
for imgs, labels in tqdm(self.dataset.testing_dataset):
    preds = self.trained_model.predict(imgs, verbose=0)
    true_vals.append(labels.numpy())
    pred_vals.append(preds)

true_vals = np.concatenate(true_vals, axis=0)
pred_vals = np.concatenate(pred_vals, axis=0)

metrics = {"MAE": [], "MSE": [], "MAPE": [], "R2": []}

for i in range(self.num_output_class):
    print(f"calculating class {i+1}")
    true_dim = true_vals[:, i]
    pred_dim = pred_vals[:, i]

    # Calculate metrics
    mae = mean_absolute_error(true_dim, pred_dim)
    mse = mean_squared_error(true_dim, pred_dim)
    mape = self.mean_absolute_percentage_error(true_dim, pred_dim)
    r2 = r2_score(true_dim, pred_dim)

    # Store metrics
    metrics["MAE"].append(mae)
    metrics["MSE"].append(mse)
    metrics["MAPE"].append(mape)
    metrics["R2"].append(r2)

return metrics
```

Appendix D

The code snippet below is the main execution script to train and evaluate models:

```
from shufflenetv2 import ShuffleNetV2
from typing import List

output_classes_names: List[str] = ["Height", "Width", "Length"]

# Base Model train + eval
print("Train and evaluating base model...")
model = ShuffleNetV2(CBAM_status=False, train_enhanced=False)
model.train_model()
model.trained_model.save("trained_basemodel.keras")
metrics = model.evaluate_model_metrics()
for i in range(output_classes_names):
    print(f"Metrics for {output_classes_names[i]}:")
    print(f"MAE: {metrics[i]['mae']}")
    print(f"MSE: {metrics[i]['mse']}")
    print(f"MAPE: {metrics[i]['mape']}%")
    print(f"R-squared: {metrics[i]['r2']}")

# CBAM Model unenhanced train + eval
print("Train and evaluating CBAM model...")
model = ShuffleNetV2(CBAM_status=True, train_enhanced=False)
model.train_model()
model.trained_model.save("trained_CBAMmodel.keras")
metrics = model.evaluate_model_metrics()
for i in range(output_classes_names):
    print(f"Metrics for {output_classes_names[i]}:")
    print(f"MAE: {metrics[i]['mae']}")
    print(f"MSE: {metrics[i]['mse']}")
    print(f"MAPE: {metrics[i]['mape']}%")
    print(f"R-squared: {metrics[i]['r2']}")

# CBAM Model enhanced train + eval
print("Train and evaluating CBAM + image enhancements model...")
model = ShuffleNetV2(CBAM_status=True, train_enhanced=True)
model.train_model()
model.trained_model.save("trained_CBAMEnhancedmodel.keras")
metrics = model.evaluate_model_metrics()
for i in range(output_classes_names):
    print(f"Metrics for {output_classes_names[i]}:")
    print(f"MAE: {metrics[i]['mae']}")
    print(f"MSE: {metrics[i]['mse']}")
    print(f"MAPE: {metrics[i]['mape']}%")
    print(f"R-squared: {metrics[i]['r2']}")
```