Development and Optimization of Monocular Depth Estimation Models

I. Leela Sai Abhiram
Contributors: Dr. Sandeep Paul

June 20, 2024

Abstract

This project focuses on developing and optimizing monocular depth estimation models. Initially, foundational concepts in computer vision and OpenCV were explored for one week. Pretrained models like MiDaS and DepthAnyThing were tested for monocular depth estimation. Following this, a custom dataset was created using a RealSense D455 camera, and a simple UNet model was developed and trained, showing promising but preliminary results.

Collaboration with Dr. Sandeep Paul provided access to different models and DIODE and NYU Depth datasets. Due to computational constraints, focus shifted to the DIODE dataset (approx. 3GB). Various optimization strategies were explored, including integrating Inception blocks into multiple models (UNet, Attention-UNet, DWT-UNet, etc.) and implementing pruning techniques to improve efficiency. Evaluation metrics such as FLOPs, MACs, MAE, RMSE, and threshold measures were used to assess model performance, detailed in an Excel report comparing standard and Inception block versions.

1 Introduction

1.1 Background

Monocular depth estimation is a crucial task in the field of computer vision, involving the prediction of depth information from a single RGB image. Traditional methods for depth estimation typically rely on expensive equipment such as LiDAR sensors or stereo cameras, which provide high accuracy but at a significant cost. However, many applications, such as augmented reality, robotics, and autonomous driving, do not require such precise measurements and can benefit from more cost-effective solutions.

1.2 Objectives

Monocular depth estimation offers a viable alternative by using a single camera to estimate depth, making it accessible and practical for a broader range of applications. Recent advancements in deep learning have significantly improved the accuracy and reliability of monocular depth estimation models, making them a compelling choice for various real-world tasks. The primary objectives of this project are:

- To gain a comprehensive understanding of computer vision (CV) and monocular depth estimation techniques.
- To develop and train a monocular depth estimation model.
- To optimize the performance of different models using different methods.

2 Literature Review

2.1 Overview of Depth Estimation Techniques

Depth estimation is a critical aspect of computer vision, enabling machines to perceive the world in three dimensions. Traditional methods for depth estimation include stereo vision, where depth is inferred from the disparity between two or more images taken from slightly different viewpoints. Techniques such as Structure from Motion (SfM) and Simultaneous Localization and Mapping (SLAM) leverage multiple views over time to reconstruct 3D scenes. Time-of-Flight (ToF) cameras and LiDAR sensors provide direct depth measurements by emitting signals and measuring the return time. Each method has its strengths and limitations, with stereo vision and SfM requiring good feature matching and sufficient texture, while ToF and LiDAR offer high precision but at higher costs and complexity.

2.2 Deep Learning for Depth Estimation

Deep learning has revolutionized depth estimation by enabling monocular depth prediction from a single image, which was previously a challenging task. Convolutional Neural Networks (CNNs) are commonly used due to their ability to learn spatial hierarchies of features. Networks like DispNet, Monodepth, and DORN (Depth from a Single Image by Predicting Relative Depth) have shown impressive results by learning from large datasets with ground truth depth information. These models often employ encoder-decoder architectures, where the encoder captures high-level features and the decoder reconstructs the depth map. Techniques such as supervised learning, self-supervised learning, and semi-supervised learning are utilized, with losses designed to enforce consistency between predicted and actual depth values.

2.3 Pretrained Models and Their Performance

Pretrained models for depth estimation leverage large datasets and significant computational resources, offering state-of-the-art performance without the need for extensive retraining. Models such as Monodepth2, MegaDepth, and MiDaS (Mixed Depth Scales) have been trained on diverse datasets like KITTI, NYU Depth, and ReDWeb. These models can generalize well across different environments and conditions. For instance, MiDaS is known for its robustness in predicting relative depth across various scenarios, including indoor, outdoor, and synthetic scenes. Performance metrics such as root mean squared error (RMSE), mean absolute error (MAE), and scale-invariant logarithmic error (SILog) are commonly used to evaluate these models, demonstrating their accuracy and reliability.

2.4 Challenges in Monocular Depth Estimation

Monocular depth estimation, despite its advancements, faces several challenges. One significant issue is scale ambiguity, where the absolute scale of objects cannot be determined from a single image. This leads to difficulties in accurately predicting distances. Additionally, the lack of ground truth depth data for diverse environments limits the generalization capabilities of models. Occlusions and textureless regions pose further challenges, as they can result in inaccurate depth predictions. Lighting variations and dynamic scenes also affect the robustness of depth estimation models. Addressing these challenges requires innovative solutions such as incorporating geometric constraints, enhancing data augmentation techniques, and developing more sophisticated neural network architectures to better capture the complexities of real-world scenes.

3 Methodology

3.1 Approaches and Techniques

The project followed a systematic approach to develop and optimize the monocular depth estimation models. Here are the steps and techniques used:

- **Initial Learning Phase:** Spent one week learning the fundamentals of computer vision and familiarizing myself with OpenCV.
- **Exploring Pretrained Models:** Tested pretrained models such as MiDaS and DepthAnyThing for monocular depth estimation.
- **Custom Dataset Creation:** Created a custom dataset using the RealSense D455 camera.
- **Model Development:** Developed a simple UNet model and trained it using the custom dataset.
- **Collaboration and Data Acquisition:** Collaborated with Dr. Sandeep Paul, who provided access to his thesis on monocular depth estimation including different models and datasets like DIODE (Dense Indoor and Outdoor Depth) and NYU Depth datasets. Due to computational resource limitations, focused on the DIODE dataset (approximately 3GB).

- **Model Optimization:** Researched and integrated Inception blocks into various models: UNet, Attention-UNet, DWT-UNet, Attention-DWT-UNet, UNet++, DWT-UNet++, and DWT-Attention-UNet++. Investigated pruning techniques and implemented changes to nbfilter sizes.
- **Evaluation and Analysis:** Evaluated models using metrics such as FLOPs, MACs, MAE, RMSE, and various threshold measures. Compiled results into a comprehensive Excel sheet for both standard and Inception block versions of each model.

3.2 Tools and Technologies

- **Programming Languages:** Python
- **Development Environments:** Google Colab, Visual Studio Code (VS Code)
- **Libraries and Frameworks:** OpenCV, TensorFlow, Keras, PyTorch etc..
- **Hardware:** RealSense D455 camera , Laptop
- **Datasets:** DIODE, NYU Depth Dataset

3.3 Data Collection and Analysis

- **Custom Dataset:** Collected using RealSense D455 camera.
- **Training and Evaluation:** Models were trained on the DIODE dataset, and performance metrics were calculated to assess the efficiency of each model.

4 Model Architectures

4.1 UNet

This function constructs a U-Net model for monocular depth estimation. It uses a series of convolutional layers with increasing filter sizes, followed by max-pooling layers for down-sampling, creating a contracting path. After reaching the bottleneck, it applies transposed convolutions for up-sampling, concatenating features from corresponding layers in the contracting path to form an expansive path. The final output layer uses a 1x1 convolution followed by a sigmoid activation to produce the depth map.

Listing 1: UNet Architecture

```
def UNet(img_rows, img_cols, color_type=3, num_class=1): # filter_num -16,
    dropout = 0.1
    nb_filter = [32,64,128,256,512]
    bn_axis = 3
    img_input = Input(shape=(img_rows, img_cols, color_type), name='
        main_input')

    conv1_1 = standard_unit1(img_input, stage='11', nb_filter=nb_filter[0])
    pool1 = MaxPooling2D((2, 2), strides=(2, 2), name='pool1')(conv1_1)
    #pool1 = MaxPooling2D(pool_size=(2, 2))(conv1_1)

    conv2_1 = standard_unit1(pool1, stage='21', nb_filter=nb_filter[1])
    pool2 = MaxPooling2D((2, 2), strides=(2, 2), name='pool2')(conv2_1)
    #pool2 = MaxPooling2D(pool_size=(2, 2))(conv2_1)
    conv3_1 = standard_unit1(pool2, stage='31', nb_filter=nb_filter[2])
    pool3 = MaxPooling2D((2, 2), strides=(2, 2), name='pool3')(conv3_1)
    #pool3 = MaxPooling2D(pool_size=(2, 2))(conv3_1)
    conv4_1 = standard_unit1(pool3, stage='41', nb_filter=nb_filter[3])
    pool4 = MaxPooling2D((2, 2), strides=(2, 2), name='pool4')(conv4_1)
    #pool4 = MaxPooling2D(pool_size=(2, 2))(conv4_1)

  # bottleneck
```

```
conv5_1 = standard_unit1(pool4, stage='51', nb_filter=nb_filter[4])

# Upsampling layers

up41 = layers.Conv2DTranspose(nb_filter[3], (2, 2), strides=(2, 2),
    name='up41', padding='same')(conv5_1)
#up41 = Conv2D(filters=nb_filter[3], kernel_size=(3, 3), padding='same
    ', name="conv41")(up41)
#up41 = LeakyReLU(alpha=0.2)(up41)
conv41 = layers.concatenate([up41, conv4_1], name='merge41', axis=
    bn_axis)
conv41 = standard_unit1(conv41, stage='41', nb_filter=nb_filter[3])

up31 = layers.Conv2DTranspose(nb_filter[2], (2, 2), strides=(2, 2),
    name='up31', padding='same')(conv41)
#up31 = Conv2D(filters=nb_filter[2], kernel_size=(3, 3), padding='same
    ', name="conv31")(up31)
#up31 = LeakyReLU(alpha=0.2)(up31)
conv31 = layers.concatenate([up31, conv3_1], name='merge31', axis=
    bn_axis)
conv31 = standard_unit1(conv31, stage='31', nb_filter=nb_filter[2])

up21 = layers.Conv2DTranspose(nb_filter[1], (2, 2), strides=(2, 2),
    name='up21', padding='same')(conv31)
#up21 = Conv2D(filters=nb_filter[1], kernel_size=(3, 3), padding='same
    ', name="conv21")(up21)
#up21 = LeakyReLU(alpha=0.2)(up21)
conv21 = layers.concatenate([up21, conv2_1], name='merge21', axis=
    bn_axis)
conv21 = standard_unit1(conv21, stage='21', nb_filter=nb_filter[1])

up11 = layers.Conv2DTranspose(nb_filter[0], (2, 2), strides=(2, 2),
    name='up11', padding='same')(conv21)
#up11 = Conv2D(filters=nb_filter[0], kernel_size=(3, 3), padding='same
    ', name="conv11")(up11)
#up11 = LeakyReLU(alpha=0.2)(up11)
conv11 = layers.concatenate([up11, conv1_1], name='merge11', axis=
    bn_axis)
conv11 = standard_unit1(conv11, stage='11', nb_filter=nb_filter[0])

# 1*1 convolutional layers
#conv_final = layers.Conv2D(num_class, (1, 1), activation='sigmoid',
    name='output_1', kernel_initializer = 'he_normal', padding='same',
    kernel_regularizer=l2(1e-4))(conv11)
conv_final = layers.Conv2D(num_class, kernel_size=(1,1),
    kernel_initializer = 'he_normal', padding='same', kernel_regularizer
    =l2(1e-4))(conv11)
conv_final = layers.BatchNormalization(axis=3)(conv_final)
conv_final = layers.Activation('sigmoid', name='UNET')(conv_final)
#conv_final = Conv2D(filters=1, kernel_size=3, strides=(1,1),
    activation='sigmoid', padding='same', name='UNET')(conv11)
# Model
model = Model(img_input, [conv_final],name="U-Net")
return model
```

4.2 Attention-UNet

The AUNet model is designed for monocular depth estimation using an attention mechanism integrated into a U-Net architecture. It consists of downsampling layers with convolutional and pooling operations followed by upsampling layers with transposed convolutions. Attention blocks are employed to selectively highlight informative features at different scales, enhancing depth map accuracy. The final layer utilizes a 1x1 convolution followed by batch normalization and sigmoid activation to produce the predicted depth map, encapsulating the model's emphasis on spatial context and feature importance.

Listing 2: Attention-UNet Architecture

```
def AUNet(img_rows, img_cols, color_type=3, num_class=1): # filter_num -16,
    dropout = 0.1
     nb_filter = [32,64,128,256,512]
     bn_axis = 3
     img_input = Input(shape=(img_rows, img_cols, color_type), name='
         main_input')
 # Downsampling layers
     conv1_1 = standard_unit1(img_input, stage='11', nb_filter=nb_filter[0])
         #128
     pool1 = layers.MaxPooling2D((2, 2), strides=(2, 2), name='pool1')(
         conv1_1) #64

     conv2_1 = standard_unit1(pool1, stage='21', nb_filter=nb_filter[1]) #
         64, 32
     pool2 = MaxPooling2D((2, 2), strides=(2, 2), name='pool2')(conv2_1)

     conv3_1 = standard_unit1(pool2, stage='31', nb_filter=nb_filter[2]) #
         32,16
     pool3 = MaxPooling2D((2, 2), strides=(2, 2), name='pool3')(conv3_1)

     conv4_1 = standard_unit1(pool3, stage='41', nb_filter=nb_filter[3])#
         16,8
     pool4 = MaxPooling2D((2, 2), strides=(2, 2), name='pool4')(conv4_1)

     conv5_1 = standard_unit1(pool4, stage='51', nb_filter=nb_filter[4]) #
         conv_8

     # Upsampling layers
     gs41=gating_signal(conv5_1, nb_filter[3]) #16
     a41= attention_block(conv4_1, gs41, nb_filter[3])
     up41 = layers.Conv2DTranspose(nb_filter[3], (2, 2), strides=(2, 2),
         name='up41', padding='same')(conv5_1)
     conv41 = layers.concatenate([up41, a41], name='merge41', axis=bn_axis)
     uconv41 = standard_unit1(conv41, stage='41', nb_filter=nb_filter[3])

     gs31=gating_signal(conv4_1, nb_filter[2]) #32
     a31= attention_block(conv3_1, gs31, nb_filter[2])
     up31 = layers.Conv2DTranspose(nb_filter[2], (2, 2), strides=(2, 2),
         name='up31', padding='same')(uconv41)
     conv31 = layers.concatenate([up31, a31], name='merge31', axis=bn_axis)
     uconv31 = standard_unit1(conv31, stage='31', nb_filter=nb_filter[2])

     gs21=gating_signal(conv3_1, nb_filter[1]) #64
     a21= attention_block(conv2_1, gs21, nb_filter[1])
     up21 = layers.Conv2DTranspose(nb_filter[1], (2, 2), strides=(2, 2),
         name='up21', padding='same')(uconv31)
     conv21 = layers.concatenate([up21, a21], name='merge21', axis=bn_axis)
     uconv21 = standard_unit1(conv21, stage='21', nb_filter=nb_filter[1])
     gs11=gating_signal(conv2_1, nb_filter[0]) #128
     a11= attention_block(conv1_1, gs11, nb_filter[0])
     up11 = layers.Conv2DTranspose(nb_filter[0], (2, 2), strides=(2, 2),
         name='up11', padding='same')(uconv21)
     conv11 = layers.concatenate([up11, a11], name='merge11', axis=bn_axis)
     uconv11 = standard_unit1(conv11, stage='11', nb_filter=nb_filter[0])

     # 1*1 convolutional layers
     #conv_final = layers.Conv2D(num_class, (1, 1), activation='sigmoid',
         name='output_1', kernel_initializer = 'he_normal', padding='same',
         kernel_regularizer=l2(1e-4))(conv11)
     conv_final = layers.Conv2D(num_class, kernel_size=(1,1),
         kernel_initializer = 'he_normal', padding='same', kernel_regularizer
         =l2(1e-4))(uconv11)
     conv_final = layers.BatchNormalization(axis=3)(conv_final)
     conv_final = layers.Activation('sigmoid', name='AUNET')(conv_final)

     # Model
     model = Model(img_input, [conv_final], name="Attn-UNet")
     return model
```

4.3 DWT-UNet

The Udwt model employs a U-Net architecture enhanced with discrete wavelet transform (DWT) pooling and inverse wavelet transform (IWT) upsampling operations for monocular depth estimation. It begins with standard convolutional units followed by DWT pooling for down-sampling. The up-sampling path utilizes IWT and convolutional layers to recover spatial resolution, integrating residual connections to preserve fine-grained details. A final 1x1 convolutional layer with batch normalization and sigmoid activation generates the depth map, leveraging wavelet transforms to capture multi-scale features effectively. I experimented decreasing the nb_filter size and included the results in the excel sheet as pruned model.

Listing 3: DWT-UNet Architecture

```
def Udwt(img_rows, img_cols, color_type=3, num_class=1): # filter_num -16,
    dropout = 0.1
    nb_filter = [32,64,128,256,512]
    bn_axis = 3
    img_input = Input(shape=(img_rows, img_cols, color_type), name='
        main_input')

    conv1_1 = standard_unit1(img_input, stage='11', nb_filter=nb_filter[0])
    pool1 = DWT_Pooling()(conv1_1)

    conv2_1 = standard_unit1(pool1, stage='21', nb_filter=nb_filter[1])
    pool2 = DWT_Pooling()(conv2_1)

    conv3_1 = standard_unit1(pool2, stage='31', nb_filter=nb_filter[2])
    pool3 = DWT_Pooling()(conv3_1)

    conv4_1 = standard_unit1(pool3, stage='41', nb_filter=nb_filter[3])
    pool4 = DWT_Pooling()(conv4_1)
    ##############changing to stdunit1
    conv5_1 = standard_unit(pool4, stage='51', nb_filter=nb_filter[4])

    up41 = IWT_UpSampling()(conv5_1)
    up41 = layers.Conv2D(filters = nb_filter[3], kernel_size = (1, 1),\
              kernel_initializer = 'he_normal', padding = 'same')(up41)
    conv41 = Add()([up41, conv4_1])
    conv41 = standard_unit1(conv41, stage='12', nb_filter=nb_filter[3])
    up31 = IWT_UpSampling()(conv41)
    up31 = layers.Conv2D(filters = nb_filter[2], kernel_size = (1, 1),\
                kernel_initializer = 'he_normal', padding = 'same')(up31)
    conv31 = Add()([up31, conv3_1])
    conv31 = standard_unit1(conv31, stage='12', nb_filter=nb_filter[2])

    up21 = IWT_UpSampling()(conv31)
    up21 = layers.Conv2D(filters = nb_filter[1], kernel_size = (1, 1),\
                kernel_initializer = 'he_normal', padding = 'same')(up21)
    conv21 = Add()([up21, conv2_1])
    conv21 = standard_unit1(conv21, stage='12', nb_filter=nb_filter[1])

    up11 = IWT_UpSampling()(conv21)
    up11 = layers.Conv2D(filters = nb_filter[0], kernel_size = (1, 1),\
                kernel_initializer = 'he_normal', padding = 'same')(up11)
    conv11 = Add()([up11, conv1_1])
    conv11 = standard_unit1(conv11, stage='12', nb_filter=nb_filter[0])

    # 1*1 convolutional layers
    #conv_final = layers.Conv2D(num_class, (1, 1), activation='sigmoid',
        name='output_1', kernel_initializer = 'he_normal', padding='same',
        kernel_regularizer=l2(1e-4))(conv11)
    conv_final = layers.Conv2D(num_class, kernel_size=(1,1),
        kernel_initializer = 'he_normal', padding='same', kernel_regularizer
        =l2(1e-4))(conv11)
    conv_final = layers.BatchNormalization(axis=3)(conv_final)
    conv_final = layers.Activation('sigmoid', name='UDWT')(conv_final)

    # Model
    model = Model(img_input, [conv_final],name="DWT-UNet2")
    return model
```

4.4 UNET++

The Nest_Net model implements a variant of the U-Net architecture for monocular depth estimation, featuring nested skip connections to enhance feature integration across multiple scales. It employs standard convolutional units followed by max-pooling for downsampling and transposed convolutions for upsampling. Each stage incorporates skip connections merging feature maps from previous layers, culminating in a deep hierarchy of features. The final output layer utilizes a 1x1 convolution with sigmoid activation to produce the depth map.I experimented decreasing the nb_filter size and included the results in the excel sheet as pruned model.

Listing 4: UNET++ Architecture

```
def Nest_Net(img_rows, img_cols, color_type=3, num_class=1,
    deep_supervision=False):
    #added 16
    nb_filter = [16,32,64,128,256,512]

    bn_axis = 3
    img_input = Input(shape=(img_rows, img_cols, color_type), name='
        main_input')

    conv1_1 = standard_unit(img_input, stage='11', nb_filter=nb_filter[0])
    pool1 = layers.MaxPooling2D((2, 2), strides=(2, 2), name='pool1')(
        conv1_1)

    conv2_1 = standard_unit(pool1, stage='21', nb_filter=nb_filter[1])
```

```
pool2 = MaxPooling2D((2, 2), strides=(2, 2), name='pool2')(conv2_1)

up1_2 = layers.Conv2DTranspose(nb_filter[0], (2, 2), strides=(2, 2),
    name='up12', padding='same')(conv2_1)
conv1_2 = layers.concatenate([up1_2, conv1_1], name='merge12', axis=
    bn_axis)
conv1_2 = standard_unit(conv1_2, stage='12', nb_filter=nb_filter[0])

conv3_1 = standard_unit(pool2, stage='31', nb_filter=nb_filter[2])
pool3 = MaxPooling2D((2, 2), strides=(2, 2), name='pool3')(conv3_1)

up2_2 = layers.Conv2DTranspose(nb_filter[1], (2, 2), strides=(2, 2),
    name='up22', padding='same')(conv3_1)
conv2_2 = layers.concatenate([up2_2, conv2_1], name='merge22', axis=
    bn_axis)
conv2_2 = standard_unit(conv2_2, stage='22', nb_filter=nb_filter[1])

up1_3 = layers.Conv2DTranspose(nb_filter[0], (2, 2), strides=(2, 2),
    name='up13', padding='same')(conv2_2)
conv1_3 = layers.concatenate([up1_3, conv1_1, conv1_2], name='merge13',
     axis=bn_axis)
conv1_3 = standard_unit(conv1_3, stage='13', nb_filter=nb_filter[0])

conv4_1 = standard_unit(pool3, stage='41', nb_filter=nb_filter[3])
pool4 = MaxPooling2D((2, 2), strides=(2, 2), name='pool4')(conv4_1)

up3_2 = layers.Conv2DTranspose(nb_filter[2], (2, 2), strides=(2, 2),
    name='up32', padding='same')(conv4_1)
conv3_2 = layers.concatenate([up3_2, conv3_1], name='merge32', axis=
    bn_axis)
conv3_2 = standard_unit(conv3_2, stage='32', nb_filter=nb_filter[2])

up2_3 = layers.Conv2DTranspose(nb_filter[1], (2, 2), strides=(2, 2),
    name='up23', padding='same')(conv3_2)
conv2_3 = layers.concatenate([up2_3, conv2_1, conv2_2], name='merge23',
     axis=bn_axis)
conv2_3 = standard_unit(conv2_3, stage='23', nb_filter=nb_filter[1])

up1_4 = layers.Conv2DTranspose(nb_filter[0], (2, 2), strides=(2, 2),
    name='up14', padding='same')(conv2_3)
conv1_4 = layers.concatenate([up1_4, conv1_1, conv1_2, conv1_3], name='
    merge14', axis=bn_axis)
conv1_4 = standard_unit(conv1_4, stage='14', nb_filter=nb_filter[0])

conv5_1 = standard_unit(pool4, stage='51', nb_filter=nb_filter[4])

up4_2 = layers.Conv2DTranspose(nb_filter[3], (2, 2), strides=(2, 2),
    name='up42', padding='same')(conv5_1)
conv4_2 = layers.concatenate([up4_2, conv4_1], name='merge42', axis=
    bn_axis)
conv4_2 = standard_unit(conv4_2, stage='42', nb_filter=nb_filter[3])

up3_3 = layers.Conv2DTranspose(nb_filter[2], (2, 2), strides=(2, 2),
    name='up33', padding='same')(conv4_2)
conv3_3 = layers.concatenate([up3_3, conv3_1, conv3_2], name='merge33',
     axis=bn_axis)
conv3_3 = standard_unit(conv3_3, stage='33', nb_filter=nb_filter[2])
```

```
    up2_4 = layers.Conv2DTranspose(nb_filter[1], (2, 2), strides=(2, 2),
        name='up24', padding='same')(conv3_3)
    conv2_4 = layers.concatenate([up2_4, conv2_1, conv2_2, conv2_3], name='
        merge24', axis=bn_axis)
    conv2_4 = standard_unit(conv2_4, stage='24', nb_filter=nb_filter[1])

    up1_5 = layers.Conv2DTranspose(nb_filter[0], (2, 2), strides=(2, 2),
        name='up15', padding='same')(conv2_4)
    conv1_5 = layers.concatenate([up1_5, conv1_1, conv1_2, conv1_3, conv1_4
        ], name='merge15', axis=bn_axis)
    conv1_5 = standard_unit(conv1_5, stage='15', nb_filter=nb_filter[0])

    #nestnet_output_1 = layers.Conv2D(num_class, (1, 1), activation='
        sigmoid', name='output_1', kernel_initializer = 'he_normal', padding
        ='same', kernel_regularizer=l2(1e-4))(conv1_2)
    #nestnet_output_2 = layers.Conv2D(num_class, (1, 1), activation='
        sigmoid', name='output_2', kernel_initializer = 'he_normal', padding
        ='same', kernel_regularizer=l2(1e-4))(conv1_3)
    #nestnet_output_3 = layers.Conv2D(num_class, (1, 1), activation='
        sigmoid', name='output_3', kernel_initializer = 'he_normal', padding
        ='same', kernel_regularizer=l2(1e-4))(conv1_4)
    nestnet_output_4 = layers.Conv2D(num_class, (1, 1), activation='sigmoid
        ', name='UNETP', kernel_initializer = 'he_normal', padding='same',
        kernel_regularizer=l2(1e-4))(conv1_5)

    if deep_supervision:
        model = Model(img_input, [nestnet_output_1,nestnet_output_2,
            nestnet_output_3,nestnet_output_4])
    else:
        model = Model(img_input, [nestnet_output_4], name="UNetp")

    return model
```

4.5 Inception Blocks

Listing 5: Attention-UNet Architecture

4.6 DWT UNET++

The DWTNestNet architecture integrates wavelet pooling and inverse wavelet transformations within a nested U-Net framework for monocular depth estimation. It employs convolutional units with increasing filter sizes and wavelet pooling layers for down-sampling, facilitating effective feature extraction across multiple scales. The model incorporates skip connections with additive merging of feature maps from previous layers during up-sampling to preserve spatial information. The final depth map is generated using a 1x1 convolution followed by batch normalization and a sigmoid activation function. This approach enables the DWTNestNet to leverage both spatial and frequency domain information, enhancing its ability to capture detailed depth cues in complex scenes. I decreased the nb_filter size to decrease the parameters due to hardware limitations and included results as pruned in excel sheet.

Listing 6: DWT UNET++ Architecture

```
def DWTNestNet(img_rows, img_cols, color_type=1, num_class=1,
    deep_supervision=False):

    nb_filter = [32,64,128,256,512,1024,2048]
    bn_axis = 3
    batch_norm= True
```

```
img_input = Input(shape=(img_rows, img_cols, color_type), name='
    main_input')

conv1_1 = standard_unit1(img_input, stage='11', nb_filter=nb_filter[0])
pool1 = DWT_Pooling()(conv1_1)

conv2_1 = standard_unit1(pool1, stage='21', nb_filter=nb_filter[1])
pool2 = DWT_Pooling()(conv2_1)

#a11=        conv1_1, conv2_1, nb_filter[0] (x,g 32)
#gating_11 = gating_signal(conv2_1, nb_filter[0],batch_norm)
#a11 = attention_block(conv1_1, gating_11, nb_filter[0])
up1_2 = IWT_UpSampling()(conv2_1)
up1_2 = layers.Conv2D(filters = nb_filter[0], kernel_size = (1, 1),\
           kernel_initializer = 'he_normal', padding = 'same')(up1_2)
print(up1_2.shape)
print(conv1_1.shape)
conv1_2 = Add()([up1_2, conv1_1])
conv1_2 = standard_unit1(conv1_2, stage='12', nb_filter=nb_filter[0])

conv3_1 = standard_unit1(pool2, stage='31', nb_filter=nb_filter[2])
pool3 = DWT_Pooling()(conv3_1)

up2_2 = IWT_UpSampling()(conv3_1)
up2_2 = layers.Conv2D(filters = nb_filter[1], kernel_size = (1, 1),\
           kernel_initializer = 'he_normal', padding = 'same')(up2_2)
print(up2_2.shape)
print(conv2_1.shape)
conv2_2 = Add()([up2_2, conv2_1])
conv2_2 = standard_unit1(conv2_2, stage='22', nb_filter=nb_filter[1])

up1_3 = IWT_UpSampling()(conv2_2)
up1_3 = layers.Conv2D(filters = nb_filter[0], kernel_size = (1, 1),\
           kernel_initializer = 'he_normal', padding = 'same')(up1_3)
print(up1_3.shape)
print(conv1_1.shape)
print(conv1_2.shape)
conv1_3 = Add()([up1_3, conv1_1, conv1_2])
conv1_3 = standard_unit1(conv1_3, stage='13', nb_filter=nb_filter[0])

conv4_1 = standard_unit1(pool3, stage='41', nb_filter=nb_filter[3])
pool4 = DWT_Pooling()(conv4_1)

up3_2 = IWT_UpSampling()(conv4_1)
up3_2 = layers.Conv2D(filters = nb_filter[2], kernel_size = (1, 1),\
           kernel_initializer = 'he_normal', padding = 'same')(up3_2)
print(up3_2.shape)
print(conv3_1.shape)
conv3_2 = Add()([up3_2, conv3_1])
conv3_2 = standard_unit1(conv3_2, stage='32', nb_filter=nb_filter[2])

up2_3 = IWT_UpSampling()(conv3_2)
up2_3 = layers.Conv2D(filters = nb_filter[1], kernel_size = (1, 1),\
           kernel_initializer = 'he_normal', padding = 'same')(up2_3)
print(up2_3.shape)
print(conv2_1.shape)
print(conv2_2.shape)
conv2_3 = Add()([up2_3, conv2_1, conv2_2])
```

```
conv2_3 = standard_unit1(conv2_3, stage='23', nb_filter=nb_filter[1])

up1_4 = IWT_UpSampling()(conv2_3)
up1_4 = layers.Conv2D(filters = nb_filter[0], kernel_size = (1, 1),\
          kernel_initializer = 'he_normal', padding = 'same')(up1_4)
print(up1_4.shape)
print(conv1_1.shape)
print(conv1_2.shape)
print(conv1_3.shape)
conv1_4 = Add()([up1_4, conv1_1, conv1_2, conv1_3])
conv1_4 = standard_unit1(conv1_4, stage='14', nb_filter=nb_filter[0])

conv5_1 = standard_unit1(pool4, stage='51', nb_filter=nb_filter[4])

up4_2 = IWT_UpSampling()(conv5_1)
up4_2 = layers.Conv2D(filters = nb_filter[3], kernel_size = (1, 1),\
          kernel_initializer = 'he_normal', padding = 'same')(up4_2)
print(up4_2.shape)#(None, 30, 40, 2048)

print(conv4_1.shape)#(None, 30, 40, 2048)
####################################
conv4_2 = Add()([up4_2, conv4_1])
conv4_2 = standard_unit1(conv4_2, stage='42', nb_filter=nb_filter[3])

#a32= conv3_2, conv4_2, nb_filter[2] (x,g 128)
gating_32 = gating_signal(conv4_2, nb_filter[0], batch_norm)
a32 = attention_block(conv3_2, gating_32, nb_filter[2])
up3_3 = IWT_UpSampling()(conv4_2)
up3_3 = layers.Conv2D(filters = nb_filter[2], kernel_size = (1, 1),\
          kernel_initializer = 'he_normal', padding = 'same')(up3_3)
print(up3_3.shape)
print(conv3_1.shape)
print(conv3_2.shape)
conv3_3 = Add()([up3_3, conv3_1, conv3_2])
conv3_3 = standard_unit1(conv3_3, stage='33', nb_filter=nb_filter[2])

up2_4 = IWT_UpSampling()(conv3_3)
up2_4 = layers.Conv2D(filters = nb_filter[1], kernel_size = (1, 1),\
          kernel_initializer = 'he_normal', padding = 'same')(up2_4)
print(up2_4.shape)
print(conv2_1.shape)
print(conv2_2.shape)
print(conv2_3.shape)
conv2_4 = Add()([up2_4, conv2_1, conv2_2, conv2_3])
conv2_4 = standard_unit1(conv2_4, stage='24', nb_filter=nb_filter[1])

up1_5 = IWT_UpSampling()(conv2_4)
up1_5 = layers.Conv2D(filters = nb_filter[0], kernel_size = (1, 1),\
          kernel_initializer = 'he_normal', padding = 'same')(up1_5)
print(up1_5.shape)
print(conv1_1.shape)
print(conv1_2.shape)
print(conv1_3.shape)
print(conv1_4.shape)
conv1_5 = Add()([up1_5, conv1_1, conv1_2, conv1_3, conv1_4])
conv1_5 = standard_unit1(conv1_5, stage='15', nb_filter=nb_filter[0])
```

```
    #nestnet_output_1 = layers.Conv2D(num_class, (1, 1), activation='
        sigmoid', name='output_1', kernel_initializer = 'he_normal', padding
        ='same', kernel_regularizer=l2(1e-4))(conv1_2)
    #nestnet_output_2 = layers.Conv2D(num_class, (1, 1), activation='
        sigmoid', name='output_2', kernel_initializer = 'he_normal', padding
        ='same', kernel_regularizer=l2(1e-4))(conv1_3)
    #nestnet_output_3 = layers.Conv2D(num_class, (1, 1), activation='
        sigmoid', name='output_3', kernel_initializer = 'he_normal', padding
        ='same', kernel_regularizer=l2(1e-4))(conv1_4)
    #nestnet_output_4 = layers.Conv2D(num_class, (1, 1), activation='
        sigmoid', name='output_4', kernel_initializer = 'he_normal', padding
        ='same', kernel_regularizer=l2(1e-4))(conv1_5)
    nestnet_output_4 = layers.Conv2D(num_class, kernel_size=(1,1),
        kernel_initializer = 'he_normal', padding='same', kernel_regularizer
        =l2(1e-4))(conv1_5)
    nestnet_output_4 = layers.BatchNormalization(axis=3)(nestnet_output_4)
    nestnet_output_4 = layers.Activation('sigmoid', name='NDWT')(
        nestnet_output_4)
    #if deep_supervision:
    #    model = Model(img_input, [nestnet_output_1,nestnet_output_2,
        nestnet_output_3,nestnet_output_4])
    #else:
    model = Model(img_input, [nestnet_output_4],name="DWT-UNetP")

    return model
```

4.7 DWT Atten UNET++

The ADWTNestNet architecture extends the DWTNestNet by integrating attention mechanisms into its nested U-Net structure for enhanced monocular depth estimation. The model utilizes progressively increasing filter sizes across convolutional units and incorporates wavelet pooling layers for efficient feature extraction at different scales. Attention blocks are employed to selectively enhance feature maps based on contextual information, improving the model's ability to capture intricate depth details. Skip connections with additive merging preserve spatial information during up-sampling stages. The final depth map is generated using a 1x1 convolution followed by batch normalization and a sigmoid activation function.I decreased the nb_filter size to decrease the parameters due to hardware limitations and included results as pruned in excel sheet.

Listing 7: DWT Atten UNET++ Architecture

```
def ADWTNestNet(img_rows, img_cols, color_type=1, num_class=1,
    deep_supervision=False):
    #added 32 to nb filter
    nb_filter = [32,64,128,256,512,1024,2048]
    bn_axis = 3
    batch_norm= True
    img_input = Input(shape=(img_rows, img_cols, color_type), name='
        main_input')

    conv1_1 = standard_unit1(img_input, stage='11', nb_filter=nb_filter[0])
    pool1 = DWT_Pooling()(conv1_1)

    conv2_1 = standard_unit1(pool1, stage='21', nb_filter=nb_filter[1])
    pool2 = DWT_Pooling()(conv2_1)

    #a11=         conv1_1, conv2_1, nb_filter[0] (x,g 32)
    gating_11 = gating_signal(conv2_1, nb_filter[0],batch_norm)
    a11 = attention_block(conv1_1, gating_11, nb_filter[2])#0
    print(a11.shape)
```

```
up1_2 = IWT_UpSampling()(conv2_1)
up1_2 = layers.Conv2D(filters = nb_filter[0], kernel_size = (1, 1),\
            kernel_initializer = 'he_normal', padding = 'same')(up1_2)
print(up1_2.shape)
conv1_2 = Add()([up1_2, a11])
conv1_2 = standard_unit1(conv1_2, stage='12', nb_filter=nb_filter[0])

conv3_1 = standard_unit1(pool2, stage='31', nb_filter=nb_filter[2])
pool3 = DWT_Pooling()(conv3_1)

#a21= conv2_1, conv3_1, nb_filter[0] (x,g 64)
gating_21 = gating_signal(conv3_1, nb_filter[1], batch_norm)
a21 = attention_block(conv2_1, gating_21, nb_filter[2])#2
up2_2 = IWT_UpSampling()(conv3_1)
up2_2 = layers.Conv2D(filters = nb_filter[1], kernel_size = (1, 1),\
            kernel_initializer = 'he_normal', padding = 'same')(up2_2)
print(a21.shape)
print(up2_2.shape)
conv2_2 = Add()([up2_2, a21])
conv2_2 = standard_unit1(conv2_2, stage='22', nb_filter=nb_filter[1])

#a12= conv1_2, conv2_2, nb_filter[0] (x,g 32)
gating_12 = gating_signal(conv2_2, nb_filter[0], batch_norm)
a12 = attention_block(conv1_2, gating_12, nb_filter[0])
up1_3 = IWT_UpSampling()(conv2_2)
up1_3 = layers.Conv2D(filters = nb_filter[0], kernel_size = (1, 1),\
            kernel_initializer = 'he_normal', padding = 'same')(up1_3)
print(a11.shape)
print(a12.shape)
print(up1_3.shape)

conv1_3 = Add()([up1_3, a11, a12])
conv1_3 = standard_unit1(conv1_3, stage='13', nb_filter=nb_filter[0])

conv4_1 = standard_unit1(pool3, stage='41', nb_filter=nb_filter[3])
pool4 = DWT_Pooling()(conv4_1)

#a31= conv3_1, conv4_1, nb_filter[2] (x,g 128)
gating_31 = gating_signal(conv4_1, nb_filter[2], batch_norm)
a31 = attention_block(conv3_1, gating_31, nb_filter[2])
up3_2 = IWT_UpSampling()(conv4_1)
up3_2 = layers.Conv2D(filters = nb_filter[2], kernel_size = (1, 1),\
            kernel_initializer = 'he_normal', padding = 'same')(up3_2)
print(a31.shape)
print(up3_2.shape)
conv3_2 = Add()([up3_2, a31])
conv3_2 = standard_unit1(conv3_2, stage='32', nb_filter=nb_filter[2])


#a22= conv2_2, conv3_2, nb_filter[1] (x,g 64)
gating_22 = gating_signal(conv3_2, nb_filter[1], batch_norm)
a22 = attention_block(conv2_2, gating_22, nb_filter[1])
up2_3 = IWT_UpSampling()(conv3_2)
up2_3 = layers.Conv2D(filters = nb_filter[1], kernel_size = (1, 1),\
            kernel_initializer = 'he_normal', padding = 'same')(up2_3)
print(a21.shape)
print(a22.shape)
print(up2_3.shape)
```

```
conv2_3 = Add()([up2_3, a21, a22])
conv2_3 = standard_unit1(conv2_3, stage='23', nb_filter=nb_filter[1])

#a13= conv1_3, conv2_3, nb_filter[0] (x,g 32)
gating_13 = gating_signal(conv2_3, nb_filter[0], batch_norm)
a13 = attention_block(conv1_3, gating_13, nb_filter[0])
up1_4 = IWT_UpSampling()(conv2_3)
up1_4 = layers.Conv2D(filters = nb_filter[0], kernel_size = (1, 1),\
          kernel_initializer = 'he_normal', padding = 'same')(up1_4)
print(up1_4.shape)
print(a11.shape)
print(a12.shape)
print(a13.shape)
conv1_4 = Add()([up1_4, a11,a12, a13])
conv1_4 = standard_unit1(conv1_4, stage='14', nb_filter=nb_filter[0])

conv5_1 = standard_unit1(pool4, stage='51', nb_filter=nb_filter[4])

#a41= conv4_1, conv5_1, nb_filter[2] (x,g 256)
gating_41 = gating_signal(conv5_1, nb_filter[2], batch_norm)
a41 = attention_block(conv4_1, gating_41, nb_filter[2])
up4_2 = IWT_UpSampling()(conv5_1)
up4_2 = layers.Conv2D(filters = nb_filter[3], kernel_size = (1, 1),\
          kernel_initializer = 'he_normal', padding = 'same')(up4_2)
print(up4_2.shape)
print(a41.shape)
conv4_2 = Add()([up4_2, a41])
conv4_2 = standard_unit1(conv4_2, stage='42', nb_filter=nb_filter[3])

#a32= conv3_2, conv4_2, nb_filter[2] (x,g 128)
gating_32 = gating_signal(conv4_2, nb_filter[2], batch_norm)
a32 = attention_block(conv3_2, gating_32, nb_filter[2])
up3_3 = IWT_UpSampling()(conv4_2)
up3_3 = layers.Conv2D(filters = nb_filter[2], kernel_size = (1, 1),\
          kernel_initializer = 'he_normal', padding = 'same')(up3_3)
print(up3_3.shape)
print(a31.shape)
print(a32.shape)
conv3_3 = Add()([up3_3, a31, a32])
conv3_3 = standard_unit1(conv3_3, stage='33', nb_filter=nb_filter[2])

#a23= conv2_3, conv3_3, nb_filter[1] (x,g 64)
gating_23 = gating_signal(conv3_3, nb_filter[1], batch_norm)
a23 = attention_block(conv2_3, gating_23, nb_filter[1])
up2_4 = IWT_UpSampling()(conv3_3)
up2_4 = layers.Conv2D(filters = nb_filter[1], kernel_size = (1, 1),\
          kernel_initializer = 'he_normal', padding = 'same')(up2_4)
print(up2_4.shape)
print(a21.shape)
print(a22.shape)
print(a23.shape)
conv2_4 = Add()([up2_4, a21, a22, a23])
conv2_4 = standard_unit1(conv2_4, stage='24', nb_filter=nb_filter[1])

#a14= conv1_4, conv2_4, nb_filter[0] (x,g 32)
gating_14 = gating_signal(conv2_4, nb_filter[0], batch_norm)
a14 = attention_block(conv1_4, gating_14, nb_filter[0])
up1_5 = IWT_UpSampling()(conv2_4)
```

```
    up1_5 = layers.Conv2D(filters = nb_filter[0], kernel_size = (1, 1),\
               kernel_initializer = 'he_normal', padding = 'same')(up1_5)
    print(up1_5.shape)
    print(a11.shape)
    print(a12.shape)
    print(a13.shape)
    print(a14.shape)
    conv1_5 = Add()([up1_5, a11, a12, a13, a14])
    conv1_5 = standard_unit1(conv1_5, stage='15', nb_filter=nb_filter[0])

    #nestnet_output_1 = layers.Conv2D(num_class, (1, 1), activation='
        sigmoid', name='output_1', kernel_initializer = 'he_normal', padding
        ='same', kernel_regularizer=l2(1e-4))(conv1_2)
    #nestnet_output_2 = layers.Conv2D(num_class, (1, 1), activation='
        sigmoid', name='output_2', kernel_initializer = 'he_normal', padding
        ='same', kernel_regularizer=l2(1e-4))(conv1_3)
    #nestnet_output_3 = layers.Conv2D(num_class, (1, 1), activation='
        sigmoid', name='output_3', kernel_initializer = 'he_normal', padding
        ='same', kernel_regularizer=l2(1e-4))(conv1_4)
    #nestnet_output_4 = layers.Conv2D(num_class, (1, 1), activation='
        sigmoid', name='output_4', kernel_initializer = 'he_normal', padding
        ='same', kernel_regularizer=l2(1e-4))(conv1_5)
    nestnet_output_4 = layers.Conv2D(num_class, kernel_size=(1,1),
        kernel_initializer = 'he_normal', padding='same', kernel_regularizer
        =l2(1e-4))(conv1_5)
    nestnet_output_4 = layers.BatchNormalization(axis=3)(nestnet_output_4)
    nestnet_output_4 = layers.Activation('sigmoid', name='NDWT')(
        nestnet_output_4)
    #if deep_supervision:
    #    model = Model(img_input, [nestnet_output_1,nestnet_output_2,
        nestnet_output_3,nestnet_output_4])
    #else:
    model = Model(img_input, [nestnet_output_4],name="AttnDWT-UNetP")

    return model
```

4.8 Inception Block

The standard_unit function mentioned below implements an Inception module, integrating multiple convolutional pathways to enhance feature extraction capabilities within neural network architectures. It consists of parallel convolutional layers: a 1x1 convolution for dimensionality reduction and feature mapping, a 3x3 convolution for capturing spatial information, a 5x5 convolution for broader receptive field coverage, and a 3x3 max pooling operation followed by 1x1 convolution for downsampling and feature extraction. Each pathway includes batch normalization for stabilizing training and ReLU activation functions to introduce non-linearity. The outputs from these pathways are concatenated along the channel axis to form a composite feature map that captures both fine-grained details and high-level contextual information. This module facilitates effective learning of complex patterns in data, enabling models to extract rich hierarchical representations and improve overall performance in various computer vision tasks.

Listing 8: Inception Block Architecture

```
def standard_unit(input_tensor, stage, nb_filter, kernel_size = 3,
    batchnorm = True):
    # 1x1 convolution
    conv_1x1 = layers.Conv2D(filters=nb_filter, kernel_size=(1, 1),
                             kernel_initializer='he_normal', padding='same'
                             )(input_tensor)
    conv_1x1 = layers.BatchNormalization()(conv_1x1)
```

```
conv_1x1 = layers.Activation('relu')(conv_1x1)

# 3x3 convolution
conv_3x3 = layers.Conv2D(filters=nb_filter, kernel_size=(3, 3),
                         kernel_initializer='he_normal', padding='same'
                         )(input_tensor)
conv_3x3 = layers.BatchNormalization()(conv_3x3)
conv_3x3 = layers.Activation('relu')(conv_3x3)

# 5x5 convolution
conv_5x5 = layers.Conv2D(filters=nb_filter, kernel_size=(5, 5),
                         kernel_initializer='he_normal', padding='same'
                         )(input_tensor)
conv_5x5 = layers.BatchNormalization()(conv_5x5)
conv_5x5 = layers.Activation('relu')(conv_5x5)

# 3x3 max pooling followed by 1x1 convolution
pool_proj = layers.MaxPooling2D(pool_size=(3, 3), strides=(1, 1),
    padding='same')(input_tensor)
pool_proj = layers.Conv2D(filters=nb_filter, kernel_size=(1, 1),
                          kernel_initializer='he_normal', padding='same
                          ')(pool_proj)
pool_proj = layers.BatchNormalization()(pool_proj)
pool_proj = layers.Activation('relu')(pool_proj)

# Concatenate all filters
x = layers.concatenate([conv_1x1, conv_3x3, conv_5x5, pool_proj], axis
    =-1)
return x
```

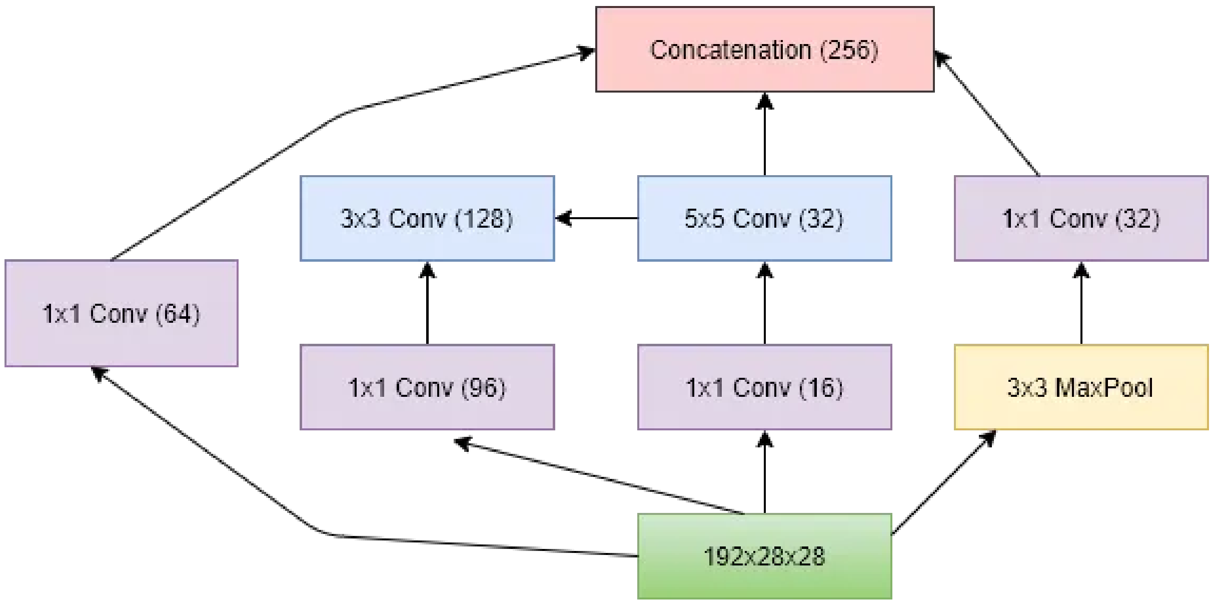


Figure 1: Inception Block Architecture

5 Training and Optimization

5.1 Training Process

The inception module has been integrated into all the models architecture by making changes to original models and trained on DIODE dataset .

5.2 Optimization Techniques

The optimisations I did were decreasing the nb_filters and also tried pruning but didn't succeed due to time constraints , so I didn't include those results, additionally there are many optimisation techniques like pruning , changing weights , etc on which I read so many research papers. Changing the data set and using a much larger dataset like NYU dataset will increase the models performance drastically , which i couldnt do because of hardware limitations .

6 Results and Analysis

6.1 Evaluation Metrics

The evaluation metrics I used are TRAINABLE PARAMS, NON TRAINABLE PARAMS, TOTAL PARAMS, FLOPs, MACs, MAE, RMSE, Threshold < 1.25, Threshold $< 1.25^2$, Threshold $< 1.25^3$, Tot imgs evaluated, Evaluation time (sec).

Out of them TRAINABLE PARAMS, NON TRAINABLE PARAMS, TOTAL PARAMS, FLOPs, MACs are used to analyse the complexity of the model , rest are used to analyse the accuracy of the models.

- Trainable Parameters are the number of model parameters that are updated during training and can be directly from the model summary in TensorFlow.
- Non-Trainable Parameters are the number of model parameters that are NOT updated during training and can be directly from the model summary in TensorFlow.
- Total Parameters are the Sum of trainable and non-trainable parameters obtained from the model summary.
- FLOPs (Floating Point Operations) are the total number of floating point operations performed during the forward pass of the model.

  Computed using a profiling function (*get_flops*) that analyzes the computational graph of the model.

  FLOPs provide insight into the computational complexity of the model. Models with lower FLOPs are generally more efficient, consuming fewer computational resources.
- MACs (Multiply-Accumulate Operations) are the total number of multiply-accumulate operations, often used as a proxy for the actual computational workload of the model.MACs give a more concrete estimate of the computational workload compared to FLOPs. They are useful for understanding the model's energy consumption and inference speed on hardware.
- MAE measures the average magnitude of errors between predicted values and true values. It gives a sense of how far off predictions are from the actual values, on average. It is computed by taking the absolute difference between each predicted and true value, averaging these absolute differences across all samples.

  $$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|$$

  where y_i are the true values and $\hat{y}_i$ are the predicted values for each sample i A lower MAE indicates that, on average, predictions are closer to the true values. It is particularly useful when errors should be penalized equally across all predictions.
- RMSE provides a measure of the average magnitude of the error between predicted and true values, taking into account the square of errors to penalize larger errors more heavily than MAE. RMSE

is computed by taking the square root of the average of squared differences between predicted and true values across all samples.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

RMSE gives a more nuanced view than MAE, emphasizing larger errors more strongly due to the squaring operation. It is useful when larger errors should be penalized more than smaller errors.

- Threshold Metrics (Threshold < 1.25, Threshold $< 1.25^2$, Threshold $< 1.25^3$) evaluate the percentage of predictions within a specified ratio threshold of the true value. They are often used in applications where relative accuracy is more critical than absolute error.

$$\text{Threshold} < 1.25 = \frac{1}{n}\sum_{i=1}^{n} I\left(\max\left(\frac{y_i}{\hat{y}_i}, \frac{\hat{y}_i}{y_i}\right) < 1.25\right)$$

$$\text{Threshold} < 1.25^2 = \frac{1}{n}\sum_{i=1}^{n} I\left(\max\left(\frac{y_i}{\hat{y}_i}, \frac{\hat{y}_i}{y_i}\right) < 1.25^2\right)$$

$$\text{Threshold} < 1.25^3 = \frac{1}{n}\sum_{i=1}^{n} I\left(\max\left(\frac{y_i}{\hat{y}_i}, \frac{\hat{y}_i}{y_i}\right) < 1.25^3\right)$$

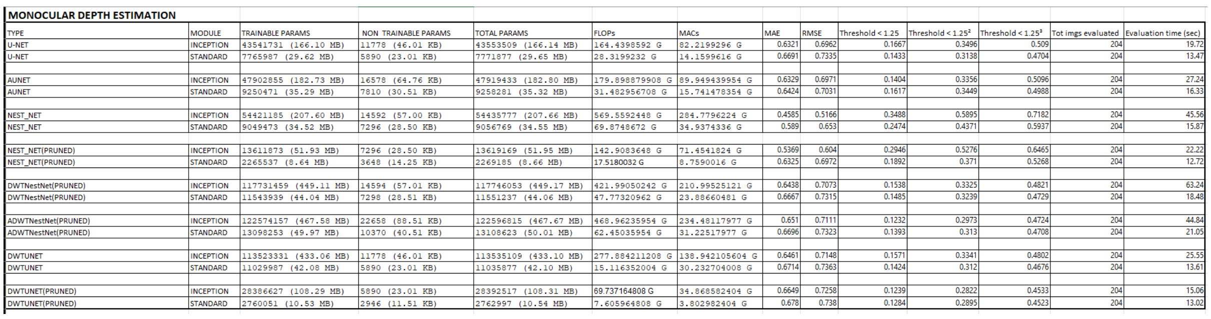
For each sample i, compute the ratio of true value y_i to predicted value $\hat{y}_i$ and vice versa. Check how many ratios fall below the specified thresholds (1.25, 1.25^2, 1.25^3). Average these across all samples.

These metrics indicate the percentage of predictions that are within ±25%, ±56.25%, and ±97.65% of the true value. They are useful for evaluating models in tasks where relative error is more critical than absolute error.

- Total Images Evaluated are the total number of images processed during the evaluation of the model.

- Evaluation Time (seconds) is the elapsed time to evaluate the model on the entire dataset.

6.2 Performance Comparison



MONOCULAR DEPTH ESTIMATION

TYPE	MODULE	TRAINABLE PARAMS	NON TRAINABLE PARAMS	TOTAL PARAMS	FLOPs	MACs	MAE	RMSE	Threshold < 1.25	Threshold < 1.25²	Threshold < 1.25³	Tot imgs evaluated	Evaluation time (sec)
U-NET	INCEPTION	43541731 (166.10 MB)	11778 (46.01 KB)	43553509 (166.14 MB)	164.4398592 G	82.2199296 G	0.6321	0.6962	0.1667	0.3496	0.509	204	19.72
U-NET	STANDARD	7765987 (29.62 MB)	5890 (23.01 KB)	7771877 (29.65 MB)	28.3199232 G	14.1599616 G	0.6691	0.7335	0.1433	0.3138	0.4704	204	13.47
AUNET	INCEPTION	47902855 (182.73 MB)	16578 (64.76 KB)	47919433 (182.80 MB)	179.898879908 G	89.949439954 G	0.6329	0.6971	0.1404	0.3356	0.5096	204	27.24
AUNET	STANDARD	9250471 (35.29 MB)	7810 (30.51 KB)	9258281 (35.32 MB)	31.482956708 G	15.741478354 G	0.6424	0.7031	0.1617	0.3449	0.4988	204	16.33
NEST_NET	INCEPTION	54421185 (207.60 MB)	14592 (57.00 KB)	54435777 (207.66 MB)	569.5592448 G	284.7796224 G	0.4585	0.5166	0.3488	0.5895	0.7182	204	45.56
NEST_NET	STANDARD	9049473 (34.52 MB)	7296 (28.50 KB)	9056769 (34.55 MB)	69.8748672 G	34.9374336 G	0.589	0.653	0.2474	0.4371	0.5937	204	15.87
NEST_NET(PRUNED)	INCEPTION	13611873 (51.93 MB)	7296 (28.50 KB)	13619169 (51.95 MB)	142.9083648 G	71.4541824 G	0.5369	0.604	0.2946	0.5276	0.6465	204	22.22
NEST_NET(PRUNED)	STANDARD	2265537 (8.64 MB)	3648 (14.25 KB)	2269185 (8.66 MB)	17.5180032 G	8.7590016 G	0.6325	0.6972	0.1892	0.371	0.5268	204	12.72
DWTNestNet(PRUNED)	INCEPTION	117731459 (449.11 MB)	14594 (57.01 KB)	117746053 (449.17 MB)	421.99050242 G	210.99525121 G	0.6438	0.7073	0.1538	0.3325	0.4821	204	63.24
DWTNestNet(PRUNED)	STANDARD	11543939 (44.04 MB)	7298 (28.51 KB)	11551237 (44.06 MB)	47.77320962 G	23.88660481 G	0.6667	0.7315	0.1485	0.3239	0.4729	204	18.48
ADWTNestNet(PRUNED)	INCEPTION	122574157 (467.58 MB)	22658 (88.51 KB)	122596815 (467.67 MB)	468.96235954 G	234.48117977 G	0.651	0.7111	0.1232	0.2973	0.4724	204	44.84
ADWTNestNet(PRUNED)	STANDARD	13098253 (49.97 MB)	10370 (40.51 KB)	13108623 (50.01 MB)	62.45035954 G	31.22517977 G	0.6696	0.7323	0.1393	0.313	0.4708	204	21.05
DWTUNET	INCEPTION	113523331 (433.06 MB)	11778 (46.01 KB)	113535109 (433.10 MB)	277.884211208 G	138.942105604 G	0.6461	0.7148	0.1571	0.3341	0.4802	204	25.55
DWTUNET	STANDARD	11029987 (42.08 MB)	5890 (23.01 KB)	11035877 (42.10 MB)	15.116352004 G	30.232704008 G	0.6714	0.7363	0.1424	0.312	0.4676	204	13.61
DWTUNET(PRUNED)	INCEPTION	28386627 (108.29 MB)	5890 (23.01 KB)	28392517 (108.31 MB)	69.737164808 G	34.868582404 G	0.6649	0.7258	0.1239	0.2822	0.4533	204	15.06
DWTUNET(PRUNED)	STANDARD	2760051 (10.53 MB)	2946 (11.51 KB)	2762997 (10.54 MB)	7.605964808 G	3.802982404 G	0.678	0.738	0.1284	0.2895	0.4523	204	13.02

Figure 2: Inception Block Architecture

The key findings and outcomes of the project are summarized below:

- **Model Performance:** The initial UNet model trained on the custom dataset showed promising results but required further tuning.

- **Inception Block Integration:** Integration of Inception blocks into various models improved their performance.

- **Pruning Techniques:** Successfully implemented changes to nbfilter sizes, enhancing model efficiency.

- **Evaluation Metrics:** Models were evaluated using metrics such as FLOPs, MACs, MAE, RMSE, and thresholds (1.25, 1.25^2, 1.25^3).
- **Comparative Analysis:** Compiled an Excel sheet comparing the performance of standard and Inception block versions of each model.
- **Model Efficiency:**The NESTNET(UNet++) INCEPTION model has the lowest MAE and RMSE values, indicating high accuracy. However, it requires significantly more computational resources, as reflected in its high FLOPs and MACs.
- **Pruning Effectiveness:**Pruned models, particularly NESTNET(UNet++) STANDARD, show a good balance between reduced computational cost and performance, with only a slight increase in error metrics compared to their non-pruned counterparts.
- **Inception vs. Standard:**Models with the INCEPTION module are having better accuracy but also higher computational demands compared to the STANDARD module models.
- **Best Overall Performance:**Considering both accuracy and computational efficiency, the NESTNET(UNet++) Pruned INCEPTION model stands out with relatively low error rates and moderate computational requirements.

7 Discussion

7.1 Comparison with Initial Objectives

The project met its objectives by successfully developing a basic monocular depth estimation model and optimizing some models. The integration of Inception blocks and pruning techniques demonstrated improvements in model performance.

7.2 Unexpected Outcomes and Challenges

- Faced so many compatibility issues with different libraries and many other errors in the initial days for work.
- Computational resource limitations hindered the training of models on the NYU Depth dataset.
- Integrating Inception blocks required careful tuning to balance model complexity and performance.

7.3 Contribution to the Field

The optimized models provide a cost-effective solution for depth estimation, making the technology more accessible. The comparative analysis offers valuable insights into the benefits of integrating Inception blocks and pruning the depth estimation models.

8 Conclusion

8.1 Main Takeaways

Monocular depth estimation is a viable alternative to expensive depth estimation methods. Optimizing models with Inception blocks and pruning techniques can significantly enhance performance.

8.2 Recommendations for Future Work

- Explore additional optimization techniques such as advanced pruning methods and quantization.
- Investigate the use of transfer learning to improve model performance with limited computational resources.
- Expand the evaluation to include real-world applications and scenarios.

9 References

- Creating a simple UNet model: Link
- Understanding Inception: Simplifying the Network Architecture. Available at: Link
- MiDaS: Link
- Monocular Depth Face Tracker: Link
- Depth Anything Release: Link
- Self-Supervised Depth Estimation Paper: Link
- ZoeDepth Using Torch Hub: Link
- Distance From Camera: Link
- Face Distance Measurement Course: Link
- MiDaS Depth Estimation Main V1: Link
- Video Depthify: Link
- Reddit on Monocular Self-Supervised Depth: Link
- Monodepth2: Link
- PackNet-SfM: Link
- PyTorch From Source: Link
- Self-Supervised Learning in Depth: Link
- KITTI Dataset: Link
- Metric3D V2 Paper: Link
- Metric3D GitHub: Link
- Metric3D GitHub Alternative: Link
- Deep Learning for Monocular Depth Estimation Review: Link
- Depth Anything README: Link
- Train Monodepth2 Example: Link
- Kitti-Dataset GitHub: Link
- DepthFM Article: Link
- MonoVAN GitHub: Link
- Monodepth2 GitHub: Link
- PyTorch Vision: Link
- RGB2Depth GitHub: Link
- Monocular Depth Estimation with Transformers: Link
- Transformer-based Depth Estimation Paper: Link
- Self-Supervised Monocular Depth Estimation: Link
- ICCV 2019 Self-Supervised Depth Estimation: Link
- Depth Estimation Survey Paper: Link
- StackOverflow on Batch Size and Epochs: Link

- Paper on Depth Estimation with Transformers: Link
- Inception Module: Link
- Inception v4 Paper: Link
- Inception ResNet Paper: Link
- Inception v4 Paper on Papers with Code: Link
- Inception v4 GitHub: Link
- Inception ResNet Paper PDF: Link
- NYU Depth V2 Dataset: Link
- GeeksforGeeks on Inception Network: Link
- TensorFlow Pruning with Keras: Link
- Stride in Convolutional Neural Networks: Link

A Github Link for Code

- https://github.com/leela4821u/Monocular-Depth-Estimation.git

B Links for saved files

- Saved test images
- Saved training graphs
- Saved models