Single-Image Depth Perception in the Wild and Implementation

E4040.2012Fall.DLDP.report

Yufan Luo yl5086, Xin Bu xb2165, Zhihao Yi zy2523
Columbia University

Abstract

This project aims to analyze, implement, and reproduce a deep learning model that recovers depth from a single image taken in unconstrained settings. The design was first introduced by Weifeng Chen et al in their paper "Single-Image Depth Perception in the Wild" [1]. The proposed model takes the 3-channel images as input and outputs the 1-channel images with pixel-wise metric depth. In our paper, we used their dataset "Depth in the Wild" consisting of 495K diverse images annotated with the relative depth for each pair of randomly sampled points. And we successfully reconstructed the inception module and implemented the hourglass model, proposed by Chen et al, with a reasonable accuracy of 73.42% in 2 epochs.

1. Introduction

Depth approximation in RGB images is always a significant problem in the field of computer vision. And in recent years there has been considerable development in this area, especially for the methods relying on RGB-D datasets. Nonetheless, the majority of the existing RGB-D datasets are on man-made scenes like campus, road scenes, and indoor scenes with no human presence. In addition, the existing RGB-D datasets obtained by depth sensors are often limited by the range, resolution, and transparent objects' presence. Hence, the model relying on such existing RGB-D scenes might not be capable of generalizing the images in unconstrained settings.

Chen et al believe it is difficult and ambiguous for humans to approximate the metric depth from the image while estimating the relative depth is quite achievable [1]. With respect to that, Chen et al made a contribution to the dataset Depth in the Wild (DIW) which consists of 495K diverse images annotated with the relative depth for each pair of randomly sampled points. The other contribution Chen made is the multi-scale network model which is capable of predicting the pixel-wise metric depth and the corresponding loss function.

2. Summary of the Original Paper

2.1 Methodology of the Original Paper

The images in DIW were gathered from Flickr, and in order to obtain the relative depth annotations, Chen et al released a crowdsourcing survey on Amazon Mechanical Turk, and present the crowd worker images with two highlighted points, asking a question: "Which point is closer, point 1, point 2, or hard to tell?"

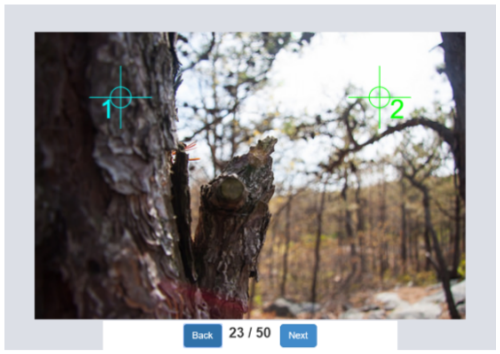

Figure 1: Crowdsourcing survey: picking the closer points [1]

Chen et al also discuss how to select the pair of points to query per image. The interesting experimental result is as follows:

1. The point at the bottom of the picture would be classified as a closer point with an accuracy rate of 85.8%.
2. For two points on the same horizontal line, the point closer to the center would be classified as a closer point with an accuracy of 71.4%

Therefore, the method chosen by Chen et al is to select two points on the same horizontal line with central symmetry. With respect to that, the probability of the left point being a closer point reached an unbiased rate of 50.3%.

The network, designed by Chen et al, is a variant of the "hourglass" structure, a type of convolutional neural network with skip connections. The network processes and passes information across multiple scales, using a series of convolutional layers for downsampling and upsampling and downsampling and skip connections to add back high-resolution features. And the variant of the inception module is applied to each layer.

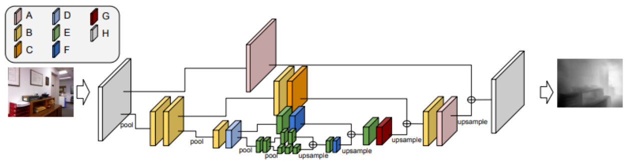

Figure 2: The "hourglass" network used by Chen et al [1]. Block H is a convolution layer with a 3x3 filter.

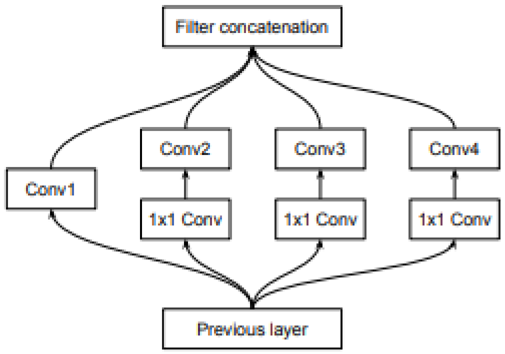

Figure 3: Variant of inception module used in the network [1]

Block Id	A	B	C	D	E	F	G
#In/#Out	128/64	128/128	128/128	128/256	256/256	256/256	256/128
Inter Dim	64	32	64	32	32	64	32
Conv1	1x1	1x1	1x1	1x1	1x1	1x1	1x1
Conv2	3x3	3x3	3x3	3x3	3x3	3x3	3x3
Conv3	7x7	5x5	7x7	5x5	5x5	7x7	5x5
Conv4	11x11	7x7	11x11	7x7	7x7	11x11	7x7

Table 1: Parameters for each type of layer in the "hourglass" network [1]

Chen et al discussed that if the dataset is large enough, the network that can fit the relative depth can predict the metric depth. In other words, if the relative depth of a picture can be determined, the metric depth can naturally be determined. And the loss function proposed by Chen et al does not depend on metric depth but on relative depth instead. The training image is represented as I and its K queries R = {(i, j, r)}, where i is the location of the first point and j is of the second point, and r ∈ {-1, 0, +1}, where +1 denotes the closer point i; -1 denotes the closer point j; and 0 denotes an equally close situation. The loss function is defined as follows:

$$L(I, R, z) = \sum_{k=1}^{K} \psi_k(I, i_k, j_k, r, z),$$

$$\psi_k(I, i_k, j_k, z) = \begin{cases} \log\left(1 + \exp(-z_{i_k} + z_{j_k})\right), & r_k = +1 \\ \log\left(1 + \exp(z_{i_k} - z_{j_k})\right), & r_k = -1 \\ (z_{i_k} - z_{j_k})^2, & r_k = 0. \end{cases}$$

where $\psi_k(I, i_k, j_k, r, z)$ is the loss for the k-th query. Hence, when the relative depth relationship of two points is "closer", the loss function makes the metric depths of the two points closer, otherwise, it makes the metric depths of the two points have a large discrepancy.

2.2 Key Results of the Original Paper

Chen et al evaluate their method by comparing their ordinal error measures (disagreement rate with ground-truth depth ordering) and metric error measures to Zoran et al's [2] , Eigen et al's [3] and other's models on NYU Depth [4], which consists of indoor scenes with ground-truth Kinect depth.

Method	WKDR	$\text{WKDR}^{=}$	$\text{WKDR}^{\neq}$
Ours	**35.6%**	**36.1%**	**36.5%**
Zoran [14]	43.5%	44.2%	41.4%
rand_12K	**34.9%**	**32.4%**	**37.6%**
rand_6K	36.1%	32.2%	39.9%
rand_3K	35.8%	28.7%	41.3%
Ours_Full	**28.3%**	**30.6%**	**28.6%**
Eigen(A) [8]	37.5%	46.9%	32.7%
Eigen(V) [8]	34.0%	43.3%	29.6%

Table 2: Ordinal error measures on NYU Depth [1]

Method	RMSE	RMSE (log)	RMSE [a] (s.inv)	absrel	sqrrel
Ours	1.13	0.39	0.26	0.36	0.46
Ours_Full	1.10	0.38	0.24	0.34	0.42
Zoran [14]	1.20	0.42	-	0.40	0.54
Eigen(A) [8]	0.75	0.26	0.20	0.21	0.19
Eigen(V) [8]	**0.64**	**0.21**	**0.17**	**0.16**	**0.12**
Wang [28]	0.75	-	-	0.22	-
Liu [6]	0.82	-	-	0.23	-
Li [10]	0.82	-	-	0.23	-
Karsch [1]	1.20	-	-	0.35	-
Baig [40]	1.0	-	-	0.3	-

Table 3: Metric error measures on NYU Depth [1]

In Table 2, Chen et al report WKDR, the weighted disagreement rate between the predicted ordinal relations and ground-truth ordinal relations, $\text{WKDR}^{=}$ (disagreement rate on pairs whose ground-truth relations =) and $\text{WKDR}^{\neq}$ (disagreement rate on pairs whose ground-truth relations are < or >).

According to Table 2, the network Chen et al proposed outperforms all three metrics trained with the same data. And according to Table 3, since the network relies on the relative depth without the presence of ground=truth depth value, the RMSE (the root mean squared error) is higher than other networks but still outperforms the model proposed by Zoran et al.

3. Methodology (of the Students' Project)

3.1. Objectives and Technical Challenges

The original paper was written in PyTorch. The main objective of our project is to implement the paper through Tensorflow 2.4 and achieve a result as good as the original.

The technical challenges are mainly two parts: the achievement of the architecture of the network and the wrangling of the huge data for training. The hourglass network consists of 24 inception modules, and each inception module is designed to have 7 layers. In practice, we also add batch normalization layers in the inception module. The network contains 5,375,969 parameters in total. The training and validation data consist of 404,880 pictures, and the annotations need to be format correct to extract information for loss computation and training process.

3.2.Problem Formulation and Design Description

The problem is to predict pixel-wise depth with respect to an RGB input image. To solve this problem, the deep network is formulated to take an RGB image of any size as input, with the output being a depth map of size 240*320*1. The label contains the coordinates of one pair of points and their relative depth.

The first task is to load the images and the labels. Unlike the assignments we have done before, the volume of the images is too large to be loaded as a whole, so we have to load them by name every time a new training batch is needed. As for labels, they are much smaller in volume and thus can be retrieved from given .csv files thoroughly and then stored in a Python dictionary where the filename becomes the key and the relative depth relationship becomes the value.

The second task is to build the model. The structure of the deep network follows the idea proposed by Chen et al's, which utilized the advantages of the hourglass network, inception modules, and skip connections to achieve high capacity as well as high trainability. Our inception module is implemented by subclassing the base class keras.layers.Layer, and our hourglass model is based on keras.models.Model. The complete structure of our model can be found in Appendix, Figure 14.

The third task is to define a loss function and a metrics function so that the model can be trained and evaluated. Here we adopt the mathematical principles behind the ranking loss function demonstrated by Chen et al and provide a loss function on our own. The loss function can get the coordinates of the point pair, and calculates the loss accordingly. Similarly, the metric function also gets the coordinates of the point pair, and justifies whether the depth is correctly predicted.

The last but one task is to train the model over the training dataset. Since the dataset cannot be loaded as a whole, we cannot simply use the model.fit() method given by Keras, and thus need to write our own training step and training loop. Notably, with the presence of a standalone yet abundant testing dataset, we did not carry out validation during training.

The final task is to evaluate the model over the testing dataset, which is done by a testing loop similar to the training loop. Chen et al evaluated the model with both ordinal error and metric error based on both ordinal ground truth and metric ground truth. However, we have no access to the metric ground truth, so only the ordinal error will be presented.

4. Implementation

4.1 Data

Intrinsic Images in the Wild: 404,880 images from Flickr. Collected by random query keywords sampled from an English dictionary and exclude artificial images such as drawings and clip art. Those images are taken with no constraints on cameras, locations, scenes, and objects.

Annotations: The annotations of those images are stored in DIW_train_val.csv. It is crowdsourced by Chen et al using Amazon Mechanical Turk. Each picture has one pair of points annotation. Those pairs of points are unconstrained pairs and symmetric pairs. The unconstrained pairs are chosen randomly with no constraint. The symmetric pairs are chosen to sample two points uniformly from a random horizontal line symmetrically. Each annotation has two rows. The first row is the relative path of the picture, the second row contains the coordinates of the pair of points, the ordinal relations between the pair of points (>, =, <), and the size of the picture.

4.2 Deep Learning Network

The network, designed by Chen et al, is a variant of the “hourglass” structure. The model uses the inception modules as the basic unit. The inception modules consist of multiple convolution layers to extract features. The inception module's multiple convolution layers are designed to be parallel to reduce parameters and avoid gradient vanishing while ensuring the capability to extract features.

Based on the inception module, the model is designed with two main parts, the upper and lower part. The upper part is the skip connection, which is usually used in ResNet to make a connection between the deep network. The skip connection makes sure the information is maintained and avoids gradient vanishing. In our scenario, the skip connection passes the input figure information from the start part of the model to the end part of the model to reduce the information loss and avoid gradient vanishing. The lower part is the variant of the hourglass model, it can be seen as two components: encoders and decoders. The encoders are a series of convolution inception modules and pooling layers. The encoders are used to extract the figure features. Each encoder would shrink the size of the feature while extracting more features, which is also known as "downsampling". The decoders are the upsampling layers, each of which increases the size of the image while recovering more details.

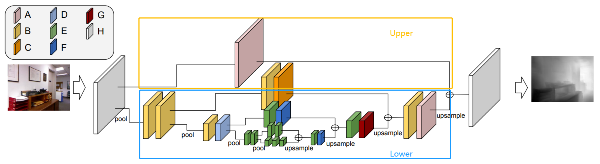

Figure 4: The "hourglass" network marked with upper and lower components

The model is trained and tested in a Jupyter notebook, namely Training.ipynb, it imports the model and functions from other python files that we have written and stored in the utils folder. It imports the Hourglass model from model.py, imports read_depth_info from read_depth_info.py, imports load_images, load_labels functions from the loader.py, and imports loss_fn function from loss.py.

In Training.py, the read_depth_info function reads depth information from the annotation file (.csv format). Then we initialize the Hourglass model instance, set up an optimizer as Adam, and instantiate the loss function as loss_fn function. Create two lists to record the train loss and accuracy. Take out an example figure for showing its depth map during each training step. At the start of the training process, extract and shuffle all the input figures' names from the depth information obtained from the read_depth_info function. Take out the names of batch size figures and use load_images to resize figures and gain input information, use load_labels to get labels. Forward the batch, calculate loss and adjust weights through gradient descent, and record the training loss and accuracy. Display the loss value after every 1000 samples and show the example output.

Annotation.csv is used as depth information, 404,880 pictures are fed to the training loop batch by batch.

The first image of the dataset, namely "dc4b5a857a519b705dafb01354765e171495 -9a11.thumb" is used as the example image.

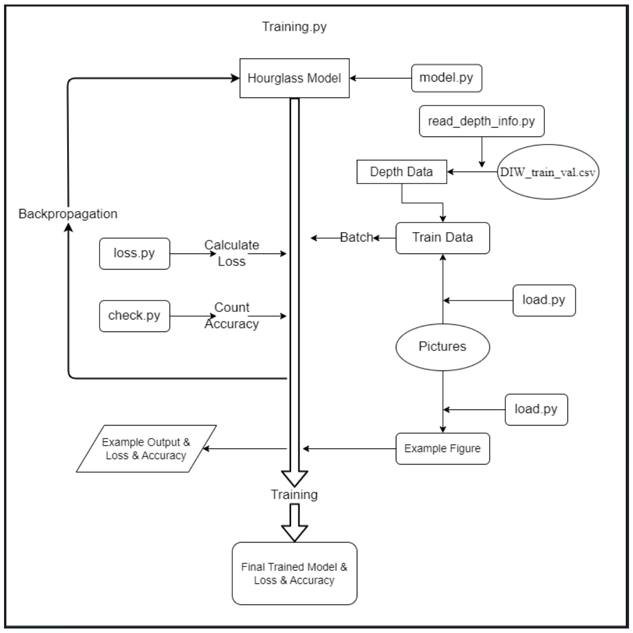

Figure 5: Training process flowchart

4.3 Software Design

Link to project Github: https://github.com/ecbme4040/e4040-2022fall-project-DLDP-xb2165-yl5086-zy2523

Description for *read_depth_info.py* implementation:

This file contains the read_depth_info function. The read_depth_info function reads annotations from a .csv file and processes them into a dictionary. The function takes in two optional arguments: input_height and input_width, which are set to 240 and 320 by default. The function requires one mandatory argument: path, which is a string that represents the path of the .csv file to be read. The function starts by creating an empty dictionary called depth_info. It then opens the .csv file located at the path specified by the path argument and reads all the lines from the file. It stores these lines in a list called info. The function then defines a rule for changing the label notation, which maps the values '<', '=', and '>' to -1, 0, and 1, respectively. The function then iterates through the lines in the info list and processes them into the desired format. For each iteration, the function gets the filename from the current line and gets the point pair information from the next line. It applies the rule for changing the label notation to the label value in the point pair information. It then scales the point pair (A and B) according to the real shape of the image, which is not always 320x240. The function then projects the normalized A and B positions to 320x240 and stores the resulting point pair in a temporary list called temp. It then appends a new key-value pair to the depth_info dictionary, where the key is the filename and the value is the temp list. Finally, the function prints a message indicating the number of labels it has successfully processed and returns the depth_info dictionary.

Flowchart for *read_depth_info.py*:

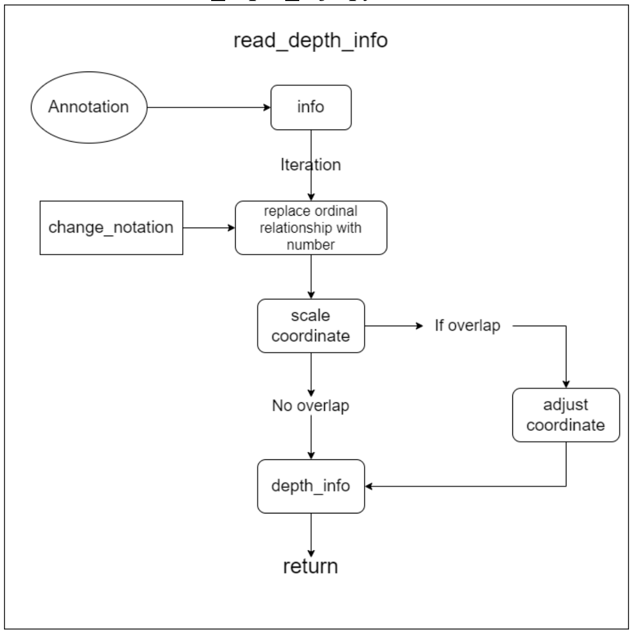

Figure 6: read_depth_info flowchart

Pseudo code for *read_depth_info.py*:

```
def read_depth_info(path, input_height=240, input_width=320):
    initialize an empty dict for labels named depth_info

    initialize an empty list named info
    standardize the filename and pair_info in path and store them to info,
    where pair_info: [yA, xA, yB, xB, relation_label, image_width,
image_height]

    quantify relation_label in info:
        ‘ < ‘ → -1
        ‘ = ‘ → 0
        ‘ > ‘ → 1

    for every pair of filename and pair_info in info:
        ori_img_width  ←  pair_info.image_width
            ori_img_height ←  pair_info.image_height

            normalize ori_img_width to input_width = 320
            normalize ori_img_height to input_height = 240

            ori_pointA ← [pair_info.xA, pair_info.yA]
            ori_pointB ← [pair_info.xB, pair_info.yB]

            normalize ori_pointA to 320x240
            normalize ori_pointB to 320x240

            if normalized ori_pointA = normalized ori_pointB:
                move ori_pointA slightly

            relation ← pair_info.relation_label
            initialize an empty list named temp
            append [normalized ori_pointA, normalized ori_pointB,-
             relation to temp
            depth_info[filename] = temp

    return depth_info
```

Description for *loss.py* implementation:

This file contains the loss function and loss_fn class. The loss function calculates the loss for a given pair of predicted depths (za and zb) and a ground-truth ordinal relation (relation). The function returns a value representing the loss, which is calculated based on the value of relationships. If the relation is 1, the function returns the logarithm of 1 plus the exponent of the difference between za and zb. If the relation is -1, the function returns the logarithm of 1 plus the exponent of the difference between zb and za. If the relation is 0, the function returns the square of the difference between za and zb. The class loss_fn is a subclass of the Loss class from the tf.keras.losses module in TensorFlow. The loss_fn class has a single method called call, which takes in two arguments: y_true and y_pred. The call method calculates the total loss for a batch of data by iterating through each image in the batch and extracting the ground-truth ordinal relation, and the predicted depths to each pair of points (A and B). It then calls the loss function for each pair of points and sums the resulting losses for all pairs in the batch. Finally, the call method returns the total loss for the batch.

Flowchart for *loss.py*:

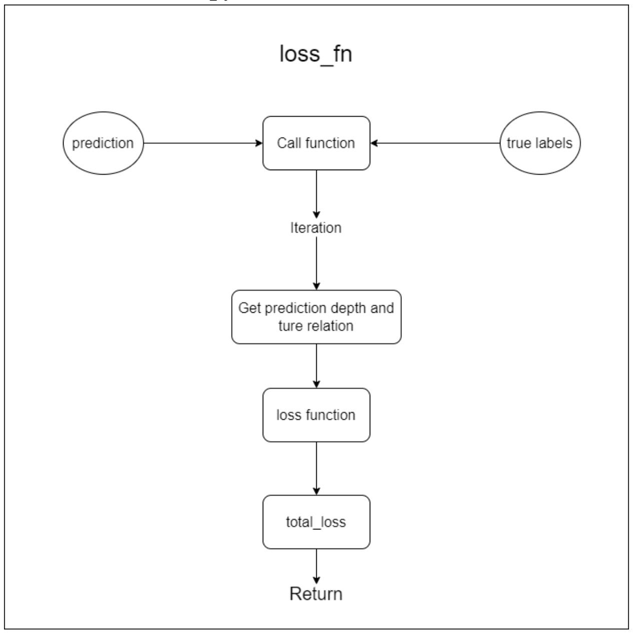

Figure 7: loss flowchart

Pseudo code for *loss.py*:

```
def loss(za, zb, relation):
    if relation = +1:
            loss = log(1 + exp(-za + zb))
    else if relation = -1:
          loss = log(1 + exp(za - zb))
    else:
            loss = (za - zb)^2


class loss_fn(tf.keras.losses.Loss):
    def init('name'):
        initialization function of the class

    def call(y_true, y_pred):
        initialize the total_loss
            remove the last dimension of y_pred(channel)

            for every image in the batch:
                get point A and point B and the relation from y_true
                get prediction za(point A) and zb(point B) from y_pred

                total_loss ← total_loss + loss(za, zb, relation)

    return total_loss
```

Description for *loader.py* implementation:

loader.py: This file contains load_images function and load_labels function. The load_images function takes in a list of filenames, a path to the directory where the images are stored, and the desired width and height of the images. It loads the images with the given filenames from the specified directory, resizes them to the desired width and height, converts them to NumPy arrays, and returns a NumPy array containing all the loaded and processed images. It also checks if an image is grayscale and, if so, converts it to a 3-channel image by replicating the grayscale channel three times. The load_labels function takes in a list of filenames and a dictionary of depth information. It returns a NumPy array of labels corresponding to the given filenames by extracting the depth information from the dictionary for each filename. Both functions use the Python Imaging Library (PIL) to load the images.

Flowchart for *loader.py*:

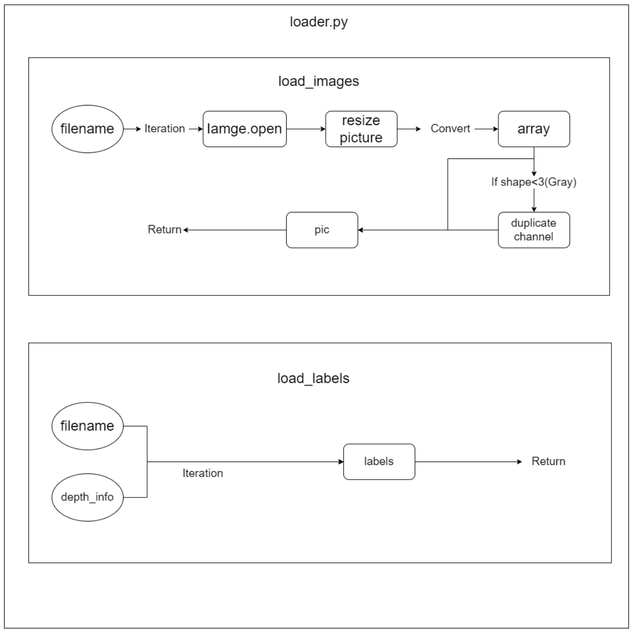

Figure 8: loader flowchart

Pseudo code for *loader.py*:

```
def load_images(filename_list, DIW_PATH, input_width, input_height):
    initialize an empty list named images

    for every filename in filename_list:
        load (DIW_PATH + filename) as an image named pic
            resize pic to (input_width, input_height)
            convert pic to a float type

            if pic is a gray image:
                make pic a three-channel image
                make every channel identical to the original pic

            append pic to images

    stack every images.pic at axis=0

    return images


def load_labels(filename_list, depth_info):
    initialize an empty list named labels

    for every filename in filename_list:
            #call customized depth_info function
        append depth_info[filename] to labels

    make labels into array type

    return labels
```

Description for *check.py* implementation:

This file contains the check function and check_fn function. The check function takes in two predicted depths and a ground-truth relation and returns 1 if the predicted relation is correct according to the ground-truth relation, and 0 otherwise. The relation can be closer (+1), further (-1), or equal (0). The check_fn function takes in a list of labels and the output of a model and returns the number of correctly classified images. It processes the output by removing the last dimension (channel), and then loops through the labels to get the ground-truth relation and the coordinates of points A and B. It then gets the predictions for the depths of points A and B and calls the check function to determine if the predicted relation is correct. It counts the number of correctly classified images and returns this count.

Flowchart for *check.py*:

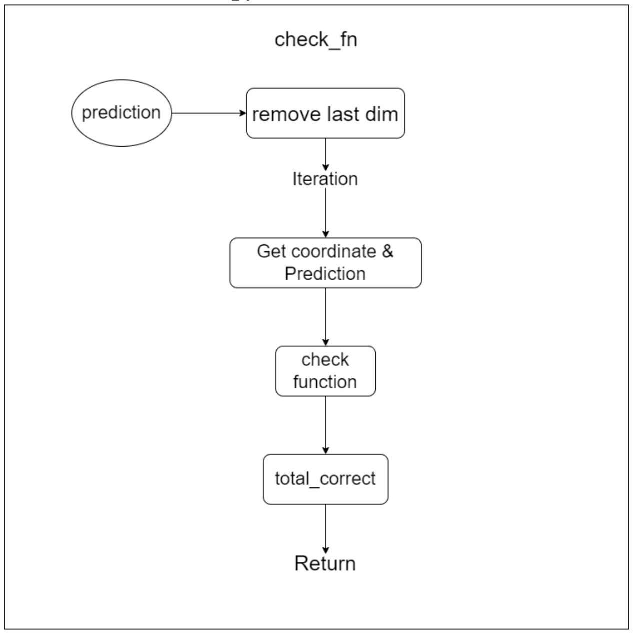

Figure 9: check_fn flowchart

Pseudo code for *check.py*:

```
def check(za, zb, relation):
    if (relation = 1) and (za < zb):
        return 1
    else if (relation = -1) and (za > zb):
        return 1
    else if (relation = 0) and (za = zb):
        return 1
    else:
            return 0


def check_fn(labels, output):
    # labels: [batch, [1, 5] = [yA, xA, yB, xB, relation]]

    initialize the counter named total_correct
    remove the last dimension of output (channel)

    for every label[1, 5] in labels:
        pointA = [label.xA, label.yA]
            pointB = [label.xB, label.yB]
            ground_truth = label.relation

            za = the corresponding prediction in output at pointA
            zb = the corresponding prediction in output at pointB

            total_correct ← total_correct + check(za, zb, ground_truth)
```

Description for *model.py* implementation:

This file contains the hourglass model, inception module. The implementation of an Inception class is a custom layer inherited from tf.keras.layer.Layer. The Inception layer consists of four blocks of convolutional layers. Each block has a convolutional layer with a different filter size, and some blocks also have a 1x1 convolutional layer. The four blocks are designed to capture different features of the input, and the outputs of the four blocks are concatenated and returned as the output of the Inception layer. The Hourglass class is inherited from tf.keras.models.Model. The hourglass model has an encoder part, which consists of a series of Inception layers followed by a max pooling layer. It also has a decoder part, which consists of a series of upsampling and Inception layers followed by a convolutional layer. To make the model manageable, the name of each layer in the hourglass init method can be found in Appendix, Table 6. The encoder part reduces the spatial resolution of the input, while the decoder part increases the spatial resolution and reconstructs the original size of the input. The output of the model is the prediction of the depth map of the input image. The call method of the Hourglass class defines the forward pass of the hourglass model, which consists of the encoding and decoding process. The my_summary method takes in two optional arguments, input_width and input_height, which represent the width and height of the input images to the model. It creates a dummy input tensor with the specified dimensions and uses it to generate a summary of the model by calling the call method on it. The plot_network method also takes in the optional input_width and input_height arguments, as well as an optional name argument, which specifies the name of the file to save the plot to. It creates a dummy input tensor and uses it to plot the model using the plot_model function from TensorFlow's utils module. The show_shapes and show_layer_names arguments are set to True to include shape information and layer names in the plot.

Flowchart for *model.py*:

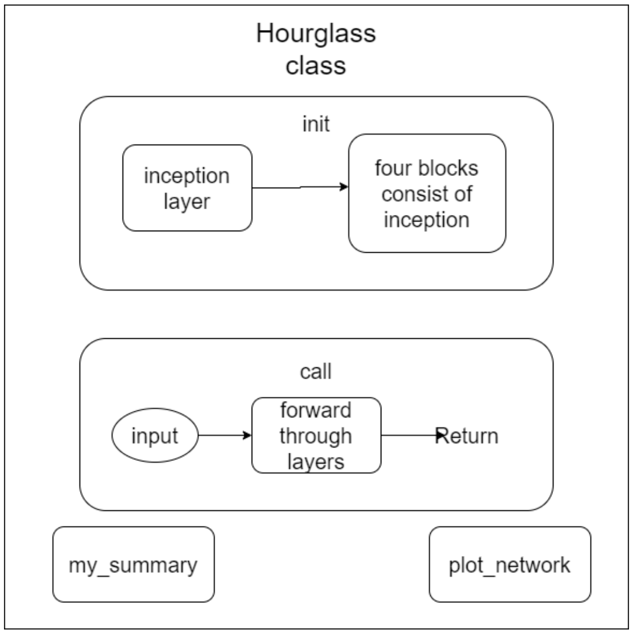

Figure 10: model flowchart

Pseudo code for *model.py*:

```
class Inception(layers.Layer):
    initialization function:
        define all 4 parallel convolution blocks
    call function:
        connect all defined blocks

class Hourglass(models.Model):
    initialization function:
        define all 22 inception modules and 2 conv2d layers

    call function:
        connect all inception modules and conv2d layers from input to
output accordingly

    my_summary function:
        show model summary

    plot_network function:
        plot the structure of the model
```

5. Results

5.1 Project Results

We train our model in only 2 epochs due to time constraints, whereas it yielded a considerable training accuracy of 71.14% and 73.42% after the first and second epochs, respectively. The accuracy improvement over two epochs is not substantial but is expected to increase and converge in a reasonable number of epochs. In addition, based on the sample images periodically generated by the model, we can assert that our model is trained in an expected direction, and the loss does decrease over the first and second epochs. After the two epochs of training, we tested our model on 50k testing images, and yielded an accuracy of 74.12% in telling the relative depth of the given point pair, which shows the model was functioning well without overfitting.

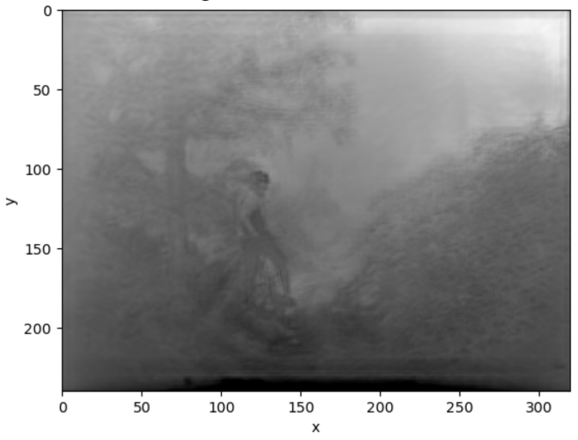

Figure 11: The metric depth by our model after two epochs of training

	1st quartile of samples	2nd quartile of samples	3rd quartile of samples	4th quartile of samples
accuracy	0.6753	0.6965	0.7053	0.7114
cumulative loss	57985	113982	170696	229608

Table 4: The accuracy and cumulative loss over samples for the first epoch

	1st quartile of samples	2nd quartile of samples	3rd quartile of samples	4th quartile of samples
accuracy	0.7299	0.7316	0.7328	0.7342
cumulative loss	55372	111114	166570	224798

Table 5: The accuracy and cumulative loss over samples for the second epoch

5.2 Comparison of the Results Between the Original Paper and Students' Project

Based on the sample image trained from the model after the second epoch, we can claim our model is capable of estimating the metric depth to some degree even though it does not perform as good as Chen et al's model.

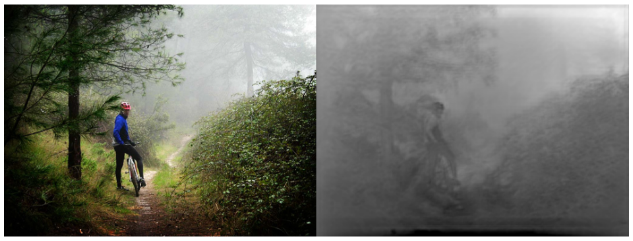
Figure 12: Qualitative results on DIW by our model. Left image: the original image. Right image: our depth

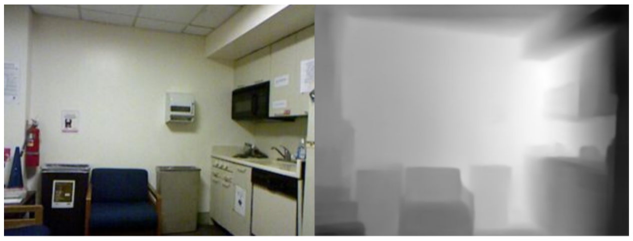
Figure 13: Qualitative results on NYU Depth by Chen et al's model [1]. Left image: the original image. Right image: Chen et al's depth

5.3 Discussion / Insights Gained

In this paper, we evaluate our model on two different distributions of initial weights, the default initializer glorot_random and the customized one glorot_uniform. The model with the initializer glorot_uniform does converge faster than the glorot_random with the ReLU activation function, which matches the empirical results from Huimin Li et al's work [5]. And the comparison of the two sets of initializers we conduct is not addressed in Chen et al's work. Another factor that would affect the convergence speed is whether to apply gaussian smoothing to the input image before feeding the model. After evaluating a small portion of the dataset, we find out that applying gaussian smoothing to the input image before feeding the model would actually make the convergence speed faster. With respect to that, we apply the gaussian smoothing in the pre-training stage to determine the hyperparameters of the model. Since the original dataset contains a large number of samples, we do not apply the gaussian smoothing to the actual training process.

Neither vanishing gradient nor exploding gradient occurs in our model training process, and our model is inspired by Chen et al. The "hourglass" model proposed by Chen et al contains a series of convolutional layers and skip connections and the variant of the inception module is applied to each layer. Contributing to the nature of the inception module and skip connection, the model is guaranteed to avoid the issues of vanishing gradient and exploding gradient. And the design of the "hourglass" structure ensures the image features can be thoroughly extracted in each layer. Hence, the overall model we conduct can successfully predict the depth of the images as Chen et al proposed.

Due to the time constraints and the volume of the dataset (containing 405k images), we trained our model in only 2 epochs, and it took 4 Nvidia T4 GPUs nearly 16 hours to finish one epoch. The accuracies after completing the first and second epoch are 71.14% and 73.42%, respectively. And the ultimate accuracy is expected to be higher in a reasonable number of epochs.

6. Future Work

With respect to the impact of customized initializers in the training stage, for future work the first direction is to explore the impact of different initializers and activation functions in convergence speed. Indeed, our findings do agree with Li et al's result which indicates that changes in weight initializers and activation functions do influence the convergence speed significantly.

We reproduce a network containing 5,375,969 parameters in total which is computationally expensive. With respect to that, for future work, the second direction is to explore the simpler structure of the model containing fewer parameters while still maintaining the performance.

With the consideration of the fact that our model was not thoroughly trained due to time constraints, for future work the third direction is to train the model in a reasonable number of epochs, making the accuracy of the model converge.

7. Conclusion

Chen et al proposed a network that can predict the metric depth in a single image taken in unconstrained settings, and the model was trained entirely on the DIW dataset only containing a pair of points and the corresponding relative depth. We successfully reproduced and implemented the model proposed by Chen et al, and the model archives an accuracy of 73.42%. The accuracy seems not good enough, but considering it was the result in only 2 epochs the accuracy is quite reasonable and expected to converge in a reasonable number of epochs.

The convergence of the accuracy is guaranteed if more epochs are applied. With respect to that, the direction for future research in this paper is to discover a simpler network structure that can significantly reduce the computational complexity while still maintaining good performance and to explore the impact of different weights initializers and activation functions which can notably influence the convergence speed in the training speed.

In the paper, we learned how to construct a complex network using the inception module that can predict the metric depth in a single image. Rather than a naive three layers network, a well-designed network with a complex structure is widely used in either academic or industrial. As well-educated engineers in the future, we are supposed to obtain the engineering competence to overcome the problem, and that is what we are trying to achieve.

8. Acknowledgement

We would like to express our deepest appreciation to Professor Kostic for his dedicated teaching in the course *Neural Networks and Deep Learning*, and Chen et al for their thorough research in the metric depth estimation which motivated us to reproduce their work. In addition, we are also thankful for Qinbo Li's open source on Github by which some interfaces of our code are inspired [6].

9. References

[1] Chen, Weifeng, et al. "Single-Image Depth Perception in the Wild." University of Michigan, Ann Arbor, in Neural Information Processing Systems (NIPS), 2016.

[2] D. Zoran, P. Isola, D. Krishnan, and W. T. Freeman, "Learning ordinal relationships for mid-level vision," in ICCV, 2015.

[3] D. Eigen and R. Fergus, "Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture," in ICCV, 2015.

[4] N. Silberman, D. Hoiem, P. Kohli, and R. Fergus, "Indoor segmentation and support inference from rgbd images," in ECCV, Springer, 2012.

[5] Li, Huimin, et al. "A Comparison of Weight Initializers in Deep Learning-based Side-channel Analysis," Delft University of Technology, Delft, The Netherlands, part of the Lecture Notes in Computer Science book series (LNSC, volume 12418), 2020

[6] Li, Qinbo, DIW_TF_Implementation, 2020, GitHub repository, https://github.com/Turmac/DIW_TF_Implementation

8. Appendix

Layer	***Output Shape***	***Params #***
Input_image (InputLayer)	*(None, 240, 320, 3)*	*0*
H1 (Conv2D)	*(None, 240, 320, 128)*	*3584*
C4_pooling	*(None, 120, 160, 128)*	*0*
C4_B1 (Inception)	*(None, 120, 160, 128)*	*102496*
C4_B1 (Inception)	*(None, 120, 160, 128)*	*102496*
C3_pooling	*(None, 60, 80, 128)*	*0*
C3_B2 (Inception)	*(None, 60, 80, 128)*	*102496*
C3_D1 (Inception)	*(None, 60, 80, 256)*	*192224*
C2_pooling	*(None, 30, 40, 256)*	*0*
C2_E2 (Inception)	*(None, 30, 40, 256)*	*212704*
C2_E3 (Inception)	*(None, 30, 40, 256)*	*212704*
C1_pooling	*(None, 15, 20, 256)*	*0*
C1_E1 (Inception)	*(None, 15, 20, 256)*	*212704*
C1_E2 (Inception)	*(None, 15, 20, 256)*	*212704*
C1_E4 (Inception)	*(None, 30, 40, 256)*	*212704*
C1_E3 (Inception)	*(None, 15, 20, 256)*	*212704*
C1_E5 (Inception)	*(None, 30, 40, 256)*	*212704*
C1_up	*(None, 30, 40, 256)*	*0*
C1_add (Add)	*(None, 30, 40, 256)*	*0*
C2_E4 (Inception)	*(None, 30, 40, 256)*	*212704*

continuing table		
Layer	***Output Shape***	***Params #***
C2_E1 (Inception)	*(None, 60, 80, 256)*	*212704*
C2_F2 (Inception)	*(None, 30, 40, 256)*	*800960*
C2_F1 (Inception)	*(None, 60, 80, 256)*	*800960*
C2_up	*(None, 60, 80, 256)*	*0*
C2_add (Add)	*(None, 60, 80, 256)*	*0*
C3_E1 (Inception)	*(None, 60, 80, 256)*	*212704*
C3_B1 (Inception)	*(None, 120, 160, 128)*	*102496*
C3_G1 (Inception)	*(None, 60, 80, 128)*	*118880*
C3_C1 (Inception)	*(None, 120, 160, 128)*	*396864*
C3_up	*(None, 60, 80, 256)*	*0*
C3_add (Add)	*(None, 60, 80, 256)*	*0*
C4_B3 (Inception)	*(None, 120, 160, 128)*	*102496*
C4_A2 (Inception)	*(None, 120, 160, 64)*	*211200*
C4_A1 (Inception)	*(None, 240, 320, 64)*	*211200*
C4_up	*(None, 120, 160, 128)*	*0*
C4_add (Add)	*(None, 120, 160, 128)*	*0*
H2(Conv2D)	*(None, 240, 320, 1)*	*577*

Table 6: The summary of the deep network

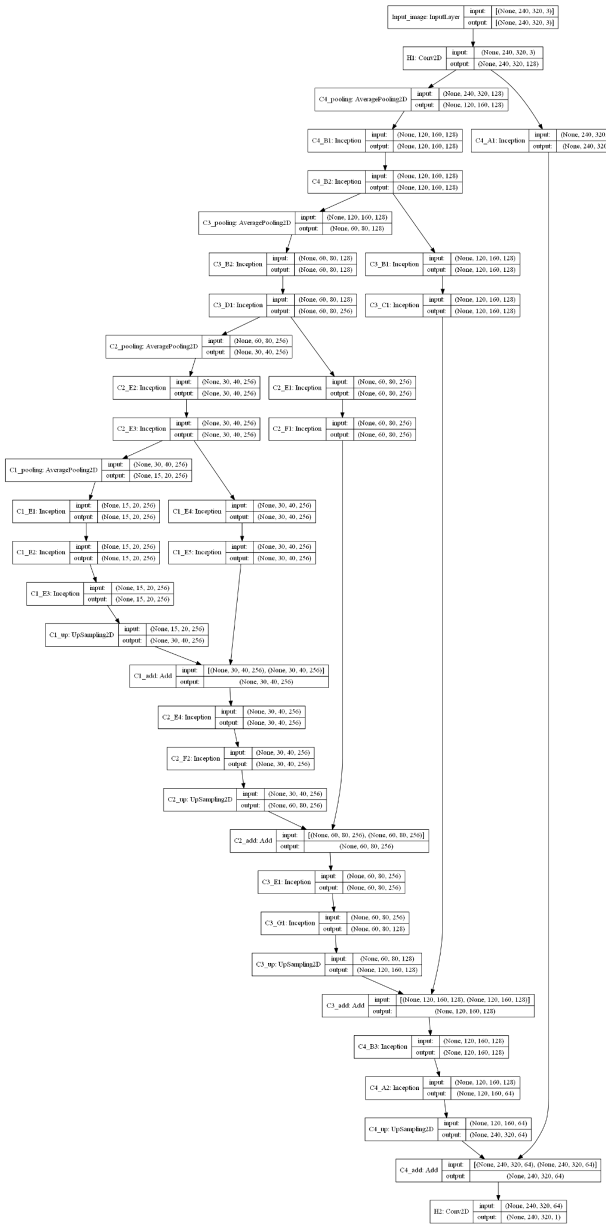
Input_image: InputLayer input: [(None, 240, 320, 3)] output: [(None, 240, 320, 3)]
H1: Conv2D input: (None, 240, 320, 3) output: (None, 240, 320, 128)
C4_pooling: AveragePooling2D input: (None, 240, 320, 128) output: (None, 120, 160, 128)
C4_B1: Inception input: (None, 120, 160, 128) output: (None, 120, 160, 128)
C4_A1: Inception input: (None, 240, 320, 128) output: (None, 240, 320, 64)
C4_B2: Inception input: (None, 120, 160, 128) output: (None, 120, 160, 128)
C3_pooling: AveragePooling2D input: (None, 120, 160, 128) output: (None, 60, 80, 128)
C3_B2: Inception input: (None, 60, 80, 128) output: (None, 60, 80, 128)
C3_B1: Inception input: (None, 120, 160, 128) output: (None, 120, 160, 128)
C3_D1: Inception input: (None, 60, 80, 128) output: (None, 60, 80, 256)
C3_C1: Inception input: (None, 120, 160, 128) output: (None, 120, 160, 128)
C2_pooling: AveragePooling2D input: (None, 60, 80, 256) output: (None, 30, 40, 256)
C2_E2: Inception input: (None, 30, 40, 256) output: (None, 30, 40, 256)
C2_E1: Inception input: (None, 60, 80, 256) output: (None, 60, 80, 256)
C2_E3: Inception input: (None, 30, 40, 256) output: (None, 30, 40, 256)
C2_F1: Inception input: (None, 60, 80, 256) output: (None, 60, 80, 256)
C1_pooling: AveragePooling2D input: (None, 30, 40, 256) output: (None, 15, 20, 256)
C1_E1: Inception input: (None, 15, 20, 256) output: (None, 15, 20, 256)
C1_E4: Inception input: (None, 30, 40, 256) output: (None, 30, 40, 256)
C1_E2: Inception input: (None, 15, 20, 256) output: (None, 15, 20, 256)
C1_E5: Inception input: (None, 30, 40, 256) output: (None, 30, 40, 256)
C1_E3: Inception input: (None, 15, 20, 256) output: (None, 15, 20, 256)
C1_up: UpSampling2D input: (None, 15, 20, 256) output: (None, 30, 40, 256)
C1_add: Add input: [(None, 30, 40, 256), (None, 30, 40, 256)] output: (None, 30, 40, 256)
C2_E4: Inception input: (None, 30, 40, 256) output: (None, 30, 40, 256)
C2_F2: Inception input: (None, 30, 40, 256) output: (None, 30, 40, 256)
C2_up: UpSampling2D input: (None, 30, 40, 256) output: (None, 60, 80, 256)
C2_add: Add input: [(None, 60, 80, 256), (None, 60, 80, 256)] output: (None, 60, 80, 256)
C3_E1: Inception input: (None, 60, 80, 256) output: (None, 60, 80, 256)
C3_G1: Inception input: (None, 60, 80, 256) output: (None, 60, 80, 128)
C3_up: UpSampling2D input: (None, 60, 80, 128) output: (None, 120, 160, 128)
C3_add: Add input: [(None, 120, 160, 128), (None, 120, 160, 128)] output: (None, 120, 160, 128)
C4_B3: Inception input: (None, 120, 160, 128) output: (None, 120, 160, 128)
C4_A2: Inception input: (None, 120, 160, 128) output: (None, 120, 160, 64)
C4_up: UpSampling2D input: (None, 120, 160, 64) output: (None, 240, 320, 64)
C4_add: Add input: [(None, 240, 320, 64), (None, 240, 320, 64)] output: (None, 240, 320, 64)
H2: Conv2D input: (None, 240, 320, 64) output: (None, 240, 320, 1)

Figure 14: The structure of the deep network

8.1 Individual Student Contributions in Fractions

	xb2165	yl5086	zy2523
Last Name	Bu	Luo	Yi
Fraction of contribution	1/3	1/3	1/3
What I did 1	Inception class	Model class	Model class

What I did 2	load functions	load functions	read .csv functions
What I did 3	loss functions	training and testing	metric functions
What I did 4	report	report	report