# Optical Flow and Convolutional Neural Network

## ENPM 673 Project 4, Spring 2021

Diane Ngo, 117554960

# Table of Contents

# Introduction

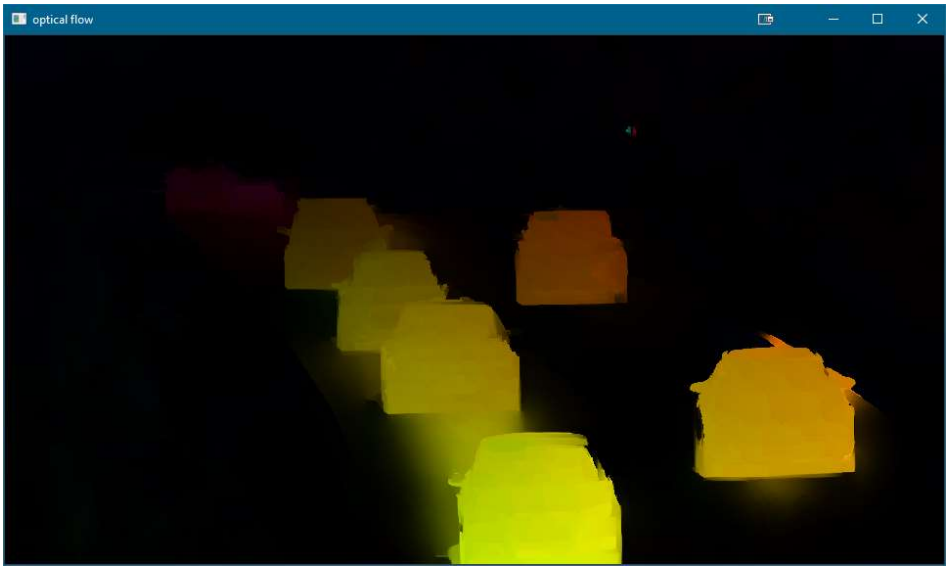
Project 4 consists of using optical flow on a video of cars on a highway and creating a Convolutional Neural Network to train and classify nine different types of seafood.

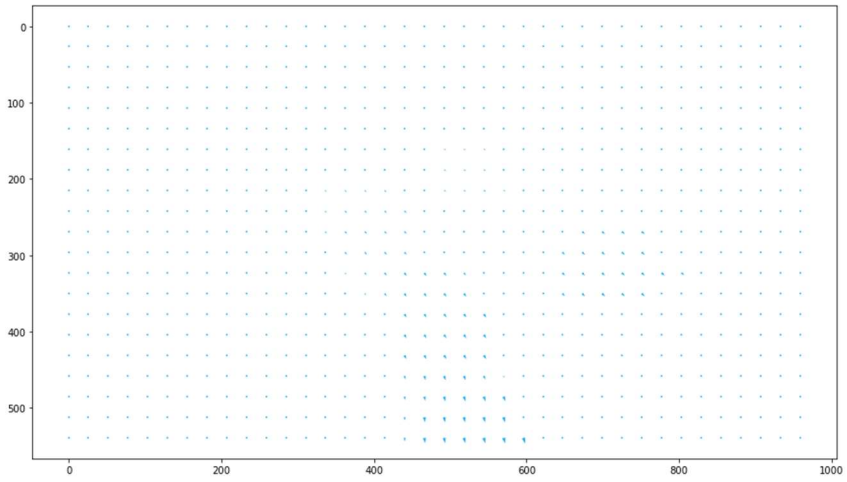
## Problem 1: Optical Flow

Problem 1 is given a video of cars on the highway. The tasks for this problem are to plot a vector field of the optical flow for every frame, and then remove the background in the video to only show the movement of the cars on the highway. I referenced the link provided from the instructions for Optical Flow. I used the Optical Flow Sparse to Dense function from OpenCV, so it displays the cars with a rainbow color in a black background. The main issues I had with this part was the calculation of the optical flow and displaying it was very choppy. To remedy this, I resized the video to half its original size and it performs better. The video has a lot of cars, resulting in a lot of movement. The optical flow's lighting seems inconsistent with some flashing. After the optical flow is finished displaying, the vector field plotting will start in another window. The vector field properly plots every 25 pixels however it is very small. Example of the optical flow and the vector field is below.

Video for Problem 1:

https://drive.google.com/file/d/10wt43d1Q535M1jnSfZB8DqYBG-MsIleK/view?usp=sharing






## Problem 2: Convolutional Neural Network

Problem 2 is given a dataset of nine different types of seafood, each with 1000 augmented images. The nine different species are: gilt head bream, red sea bream, sea bass, horse mackerel, black sea sprat, striped red mullet, trout, and shrimp. The task for this problem is to create a Convolutional Neural Network (CNN) using VGG-16 architecture in order to classify the different type of seafood. The suggestion was to use Google Colab with Pytorch if students did not have access to a GPU, and use Tensorflow, Pytorch, or Keras.

Please note that I did not use Google Colab and Pytorch. My computer has a powerful GPU, Nvidia GTX 1080Ti and 32GB of RAM, so training the models was rather fast. More on results will be covered later.

It is important when running my code, the fish dataset directory needs to be structured a certain way. There are two ways to set up the directory. The first is manually changing the directory for the Fish Dataset inside the folder where the program is being run. I was told Google Colab had a certain way to import files and because I didn't use it, I had to work around that. The original dataset directory was Fish Dataset -> Fish Name -> Fish Name and Fish Name GT (ground truth). There is a cell in my code with import shutil, run that cell to delete the ground truth folders (these folders caused me a lot of problems for training, and since it is not necessary for training, I just deleted it. After deleting these GT files, please merge the Fish Names folder so that the directory should be Fish Dataset -> Fish Name -> image files. The easier method, is just to use the zip file provided from the google drive link, please download this and place in the folder where the program code is, I know the instructions said to avoid including the dataset provided.

I am using Jupyter notebook with tensorflow's keras library, numpy, matplotlib, and PIL for images. After setting up the directory the cell after all the imports prints out the directory list. I've also created a cell in order to filter out corrupted files and delete them, since sometimes I would run the training, I would get an error with "model.fit can only use jpg, png, etc … images". You can probably disregard this cell, but it is there just in case. After setting up the directory and introductory material, the general pipeline of the code can be explained. In general, a model is created under the VGG-16 architecture and then fit.

## Pipeline

1. Set the batch size, scale down the image size, create the training and validation dataset through keras preprocessing dataset function.
2. Plot 9 random images from the datasets and display their respective class name.
3. Configure the training and validation sets for performance through autotune.
    a. This keeps the images in memory and avoids a bottleneck during training.
4. Create the model through Sequential.
    a. Standardize the data through rescaling to [-1, 1].
    b. Create VGG-16 Architecture.
5. Compile model with adam optimizer and SpareCategoricalCrossEntropy loss calculator.
6. Fit model with training and validation data, with number of epochs.
7. Plot 2 graphs, accuracy and loss between training and validation.

Figure 1 shows the plot of 9 randomly selected seafood from the dataset. This is to show that the program can properly extract the data from the directory and it is set up correctly.

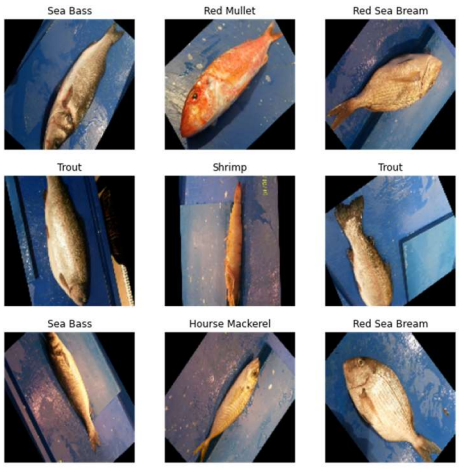
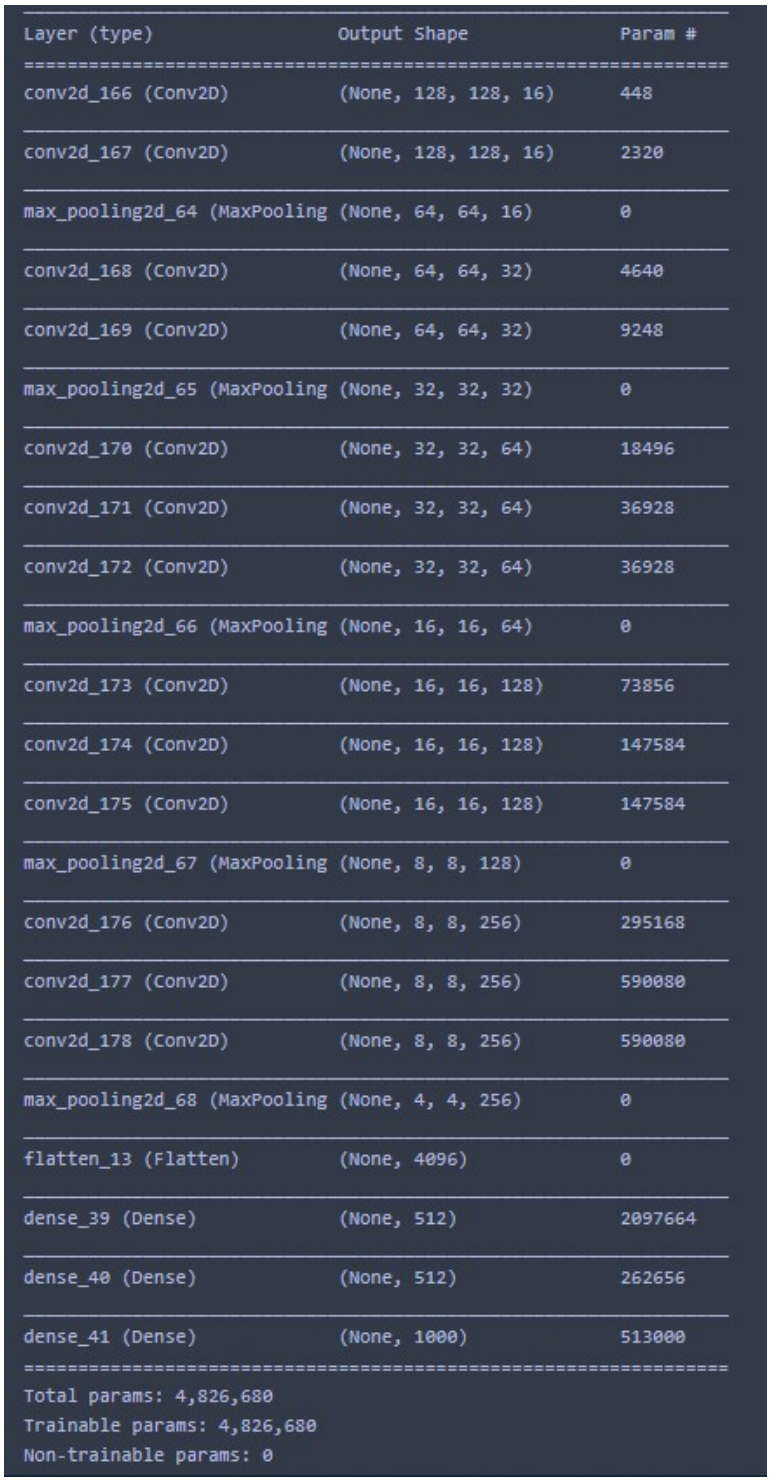



*Figure 1. Data Plot*

Github link for Problem 2 in case program cannot be run:
https://github.com/dngo13/ENPM673/blob/main/Project%204/cnn_prj4.ipynb

The VGG-16 Architecture is displayed below (Figure 2) from the keras model summary.



| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d_166 (Conv2D) | (None, 128, 128, 16) | 448 |
| conv2d_167 (Conv2D) | (None, 128, 128, 16) | 2320 |
| max_pooling2d_64 (MaxPooling | (None, 64, 64, 16) | 0 |
| conv2d_168 (Conv2D) | (None, 64, 64, 32) | 4640 |
| conv2d_169 (Conv2D) | (None, 64, 64, 32) | 9248 |
| max_pooling2d_65 (MaxPooling | (None, 32, 32, 32) | 0 |
| conv2d_170 (Conv2D) | (None, 32, 32, 64) | 18496 |
| conv2d_171 (Conv2D) | (None, 32, 32, 64) | 36928 |
| conv2d_172 (Conv2D) | (None, 32, 32, 64) | 36928 |
| max_pooling2d_66 (MaxPooling | (None, 16, 16, 64) | 0 |
| conv2d_173 (Conv2D) | (None, 16, 16, 128) | 73856 |
| conv2d_174 (Conv2D) | (None, 16, 16, 128) | 147584 |
| conv2d_175 (Conv2D) | (None, 16, 16, 128) | 147584 |
| max_pooling2d_67 (MaxPooling | (None, 8, 8, 128) | 0 |
| conv2d_176 (Conv2D) | (None, 8, 8, 256) | 295168 |
| conv2d_177 (Conv2D) | (None, 8, 8, 256) | 590080 |
| conv2d_178 (Conv2D) | (None, 8, 8, 256) | 590080 |
| max_pooling2d_68 (MaxPooling | (None, 4, 4, 256) | 0 |
| flatten_13 (Flatten) | (None, 4096) | 0 |
| dense_39 (Dense) | (None, 512) | 2097664 |
| dense_40 (Dense) | (None, 512) | 262656 |
| dense_41 (Dense) | (None, 1000) | 513000 |

Total params: 4,826,680
Trainable params: 4,826,680
Non-trainable params: 0

*Figure 2. VGG-16 Architecture Summary from Keras*

## Results

For testing purposes while creating the model, I have tested a few different scenarios. I started testing with a batch size of 32 and 10 epochs. I created 3 different models, one with 10 layers, 13 layers, and then 16 layers (following VGG-16). The first two were mainly just testing configuration and if training and classification gave consistent results. For the three different layer scenarios, I have tested each with standardization and without through data rescaling. I used a kernel size of (3, 3) for each test. For pooling, I used a pool size of (2, 2), and strides of (2, 2). The filter size for each convolution increases from 16, 32, 64, 128, and 256.

The 10-layer model consists of 2 convolution > pooling > 2 convolution > pooling > 3 convolution > pooling > flatten > 3 dense layer. Figures 3 and 4 are the plots for accuracy and loss for the 10-layer, 32 batch size, and rescaling/no rescaling. The data for these tests were unstable to a sense. The rescaling on this model made the training accuracy less than without the standardization. The model did reach an accuracy of 95% however the training accuracy and the validation accuracy did not seem to converge.

The 13-layer model consists of 2 convolution > pooling > 2 convolution > pooling > 3 convolution > pooling > 3 convolution > pooling > flatten > 3 dense layer. Figures 5 and 6 are the plots for accuracy and loss for the 13-layer, 32 batch size, and rescaling/no rescaling. The results for the 13-layer testing was unexpected compared to the 10-layer one. The data did not converge well and reached an overall lower accuracy than its predecessor. The validation data accuracy with standardization was at 90%.

The 16-layer model consists of 2 convolution > pooling > 2 convolution > pooling > 3 convolution > pooling > 3 convolution > pooling > 3 convolution > pooling > flatten > 3 dense layer. Figures 7 and 8 are the plots for accuracy and loss for the 16-layer, 32 batch size, and rescaling/no rescaling. I have also done an extra test with the 16 layer and a batch size of 64. The plots are in Figure 9 and 10. These have an epoch size of 20 because it took longer to reach the desired accuracy. The test with the batch size of 32 at 16-layers was by far the best results. The training accuracy and validation accuracy converged nicely around 95%, and the plot was stable. At a batch size of 64, the graph was unstable and did not reach a similar accuracy. Table 1 shows the data for Fig. 7/8.

**Table 1**. Comparing the Accuracy and Loss for 16-layer, 32 batch size

| Epoch Number | Training Acc. | Training Loss | Validation Acc. | Validation Loss |
|---|---|---|---|---|
| 1 | 0.1159 | 3.3550 | 0.2361 | 1.9376 |
| 2 | 0.3049 | 1.7975 | 0.5772 | 1.2059 |
| 3 | 0.5979 | 1.0878 | 0.7544 | 0.6752 |
| 4 | 0.7685 | 0.6265 | 0.8611 | 0.3746 |
| 5 | 0.8499 | 0.4116 | 0.8856 | 0.3412 |
| 6 | 0.8908 | 0.3037 | 0.9094 | 0.2882 |
| 7 | 0.9219 | 0.2258 | 0.9339 | 0.2135 |
| 8 | 0.9399 | 0.1646 | 0.9322 | 0.1932 |
| 9 | 0.9606 | 0.1189 | 0.9317 | 0.2105 |
| 10 | 0.9563 | 0.1290 | 0.9350 | 0.1954 |


*Figure 3. Accuracy vs. Loss, Batch Size 32, 10 layers, and No Rescaling*

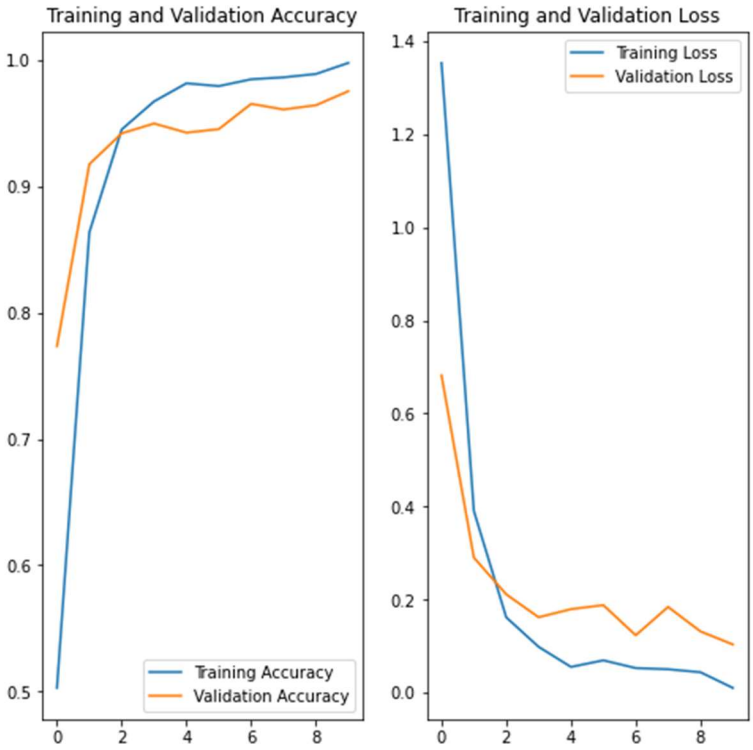



*Figure 4. Accuracy vs. Loss, Batch Size 32, 10 layers, with Rescaling*

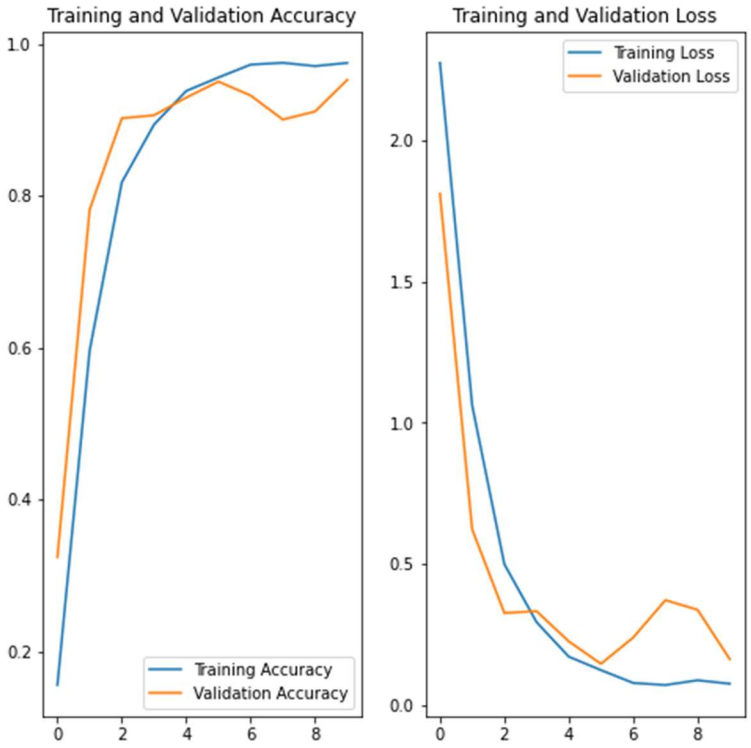



*Figure 5. Accuracy vs. Loss, Batch Size 32, 13 layers, and No Rescaling*


*Figure 6. Accuracy vs. Loss, Batch Size 32, 13 layers, and Rescaling*

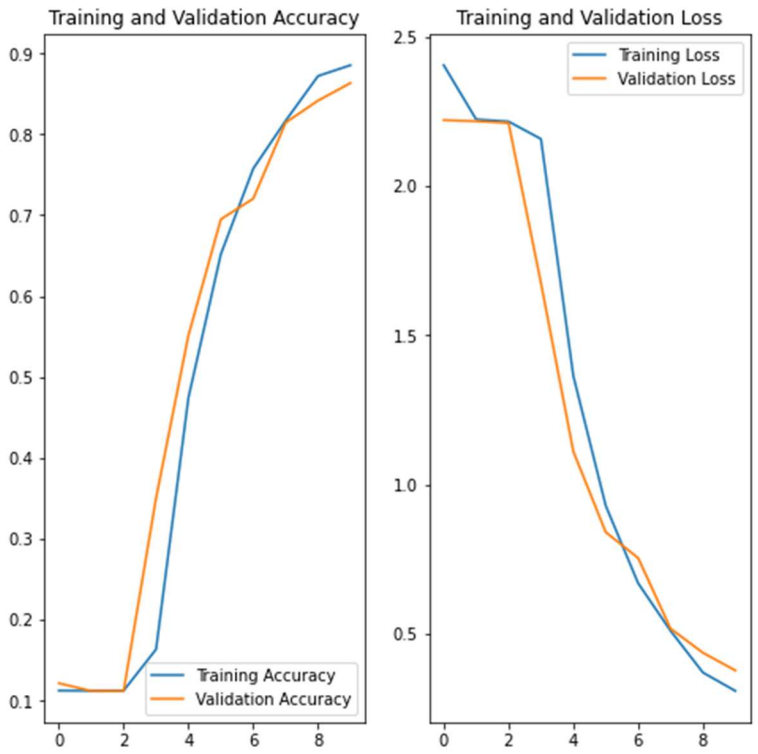



*Figure 7. Accuracy vs. Loss, Batch Size 32, 16 layers, and No Rescaling*

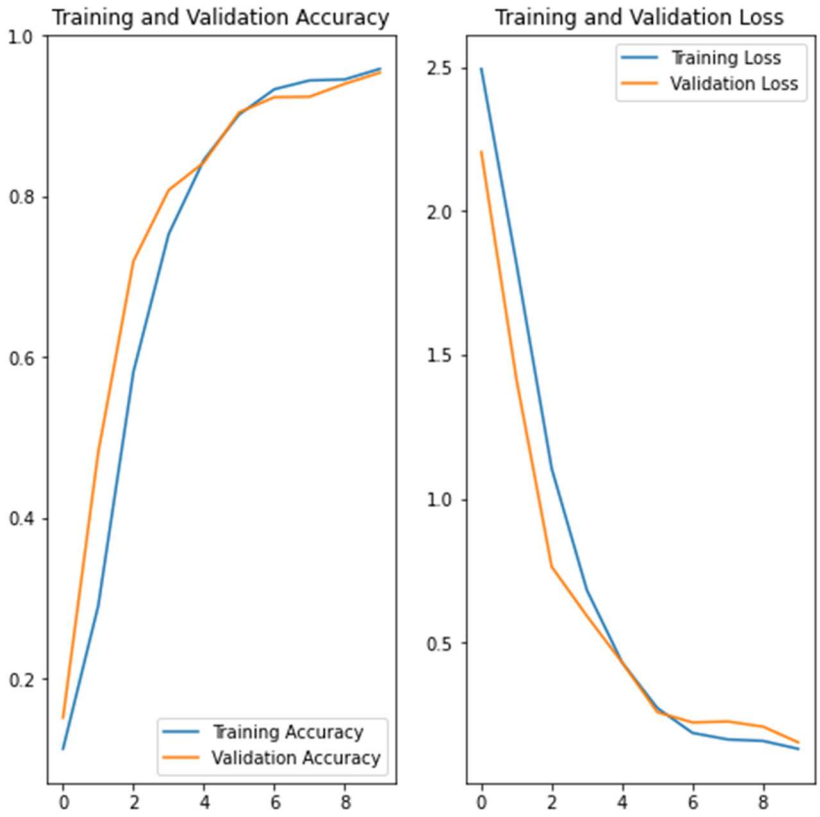



*Figure 8. Accuracy vs. Loss, Batch Size 32, 16 layers, and Rescaling*

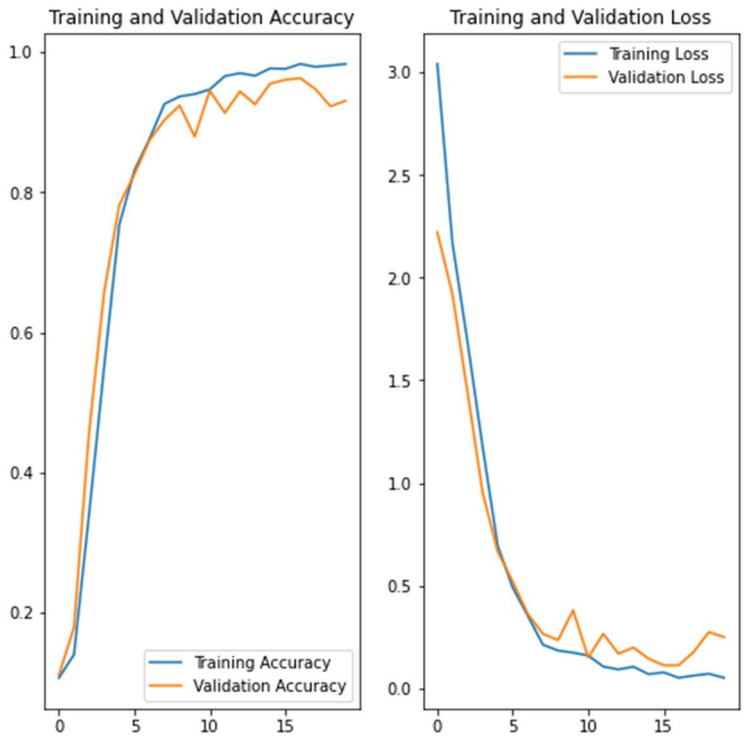



*Figure 9. Accuracy vs. Loss, Batch Size 64, 16 layers, and No Rescaling*

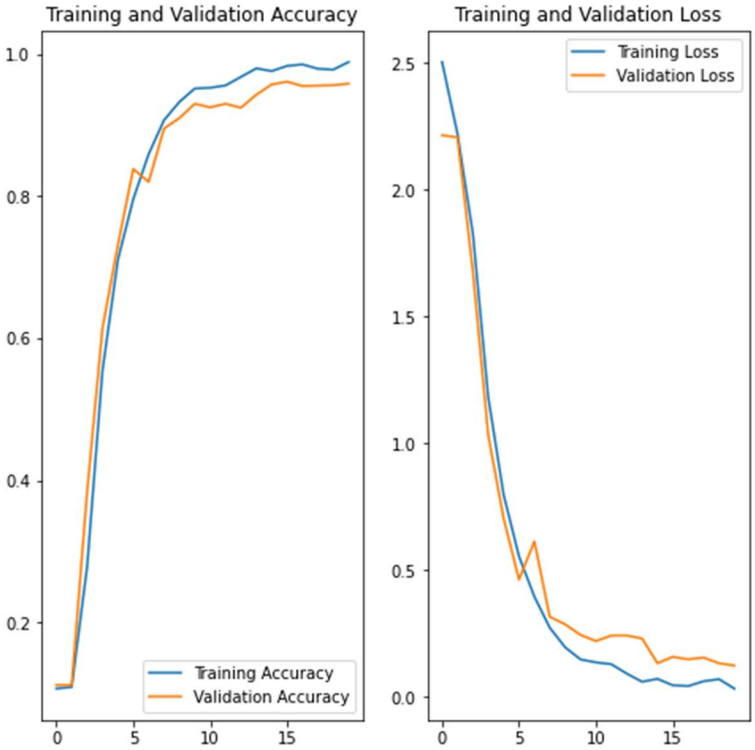



*Figure 10. Accuracy vs. Loss, Batch Size 64, 16 layers, and Rescaling*

# References

O.Ulucan, D.Karakaya, and M.Turkan.(2020) A large-scale dataset for fish segmentation and classification. In Conf. Innovations Intell. Syst. Appli. (ASYU)

https://learnopencv.com/optical-flow-in-opencv/

https://www.tensorflow.org/tutorials/load_data/images

https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image_dataset_from_directory

https://www.tutorialspoint.com/matplotlib/matplotlib_quiver_plot.htm