Pyramid Stereo Matching Network Implementation

Patrick Williams
patrickwilliams@wustl.edu

Abstract

Vision Stereo is an important task within Computer Vision. Many different approaches have been taken to calculate this disparity map, but Pyramid Stereo Matching Network is one of the first neural networks to successfully create disparity maps with state of the art results. Pyramid Stereo Matching uses a convolutional neural network and spatial pyramid pooling to create cost volumes fed into a 3-d stacked hourglass convolutional neural network architecture before upsampling and disparity regression to create a disparity map from the image pair. This paper implements the Pyramid Stereo Matching Network using Keras models to test the abilities of the Pyramid Stereo Matching Network.

1 Introduction

Depth estimation is a critical task in the area of Computer Vision. Knowing the depth element of a pixel within an image provides you with much more information to perform object detection, 3D model reconstruction, among other tasks. Knowing the depth of a pixel is very useful information in itself as well. For example, depth estimation is very useful within autonomous vehicles so that the vehicle can know how far away it is from other objects so it can avoid hitting other objects and follow traffic-laws to reach its destination.

In order to perform depth estimation, there are several methods that have been used to acquire a disparity map. LIDAR (light detection and ranging) is one technique used today that acquired depth information by sending out lasers and measuring the reflected light with a sensor to determine distance. LIDAR has been proven to be effective, but it is costly. Many methods have relied on the abilities of Computer Vision to perform epipolar geometry.

A point in space will have light reflect off of it that can be captured by a camera. The camera will therefore know where that pixel is on a line in space, but it is not able to tell where that pixel is on the line. Using two cameras where we know the orientation between the camera allows us to use epipolar geometry by matching a pixel in one camera to a pixel in another camera. Since those cameras put that pixel's origin point on two different lines in space, we can find the intersection of those lines to find the point's location in space.

A common use of epipolar geometry is when we have two cameras calibrated in the y and z directions so that they only vary in the x direction. This setup allows the disparity of a pixel between the two cameras to only differ in the positive x direction from the left image to the right image so the possibilities to match pixels between is much lower. For each point in the left image (x,y), we want to find the corresponding pixel in the right image at (x+d,y). Every pixel will have its own value for ”d”, so a disparity map for the left image can be formed with the value ”d” represented at each location. Similarly, a disparity map for the right image can be found by matching the point (x,y) in the right image with the point (x-d,y) in the left image. There are occasions where one pixel in the left image will not have an accurate corresponding pixel in the right image. This happens when the disparity puts the pixel out of the frame of the right image or for occlusions, when there is a pixel in one image blocked off by some object in-front of that pixel's point in space in the other image. These pixels should still have a value for disparity in the images disparity map, but are much harder to deal with since there is no pixel to accurately match to in the pair's image.

Up until now, the field of Computer Vision has mostly relied on algorithms to build a disparity map to match pixels between a left and right image. Typically, matching pixels from one image to another is based on some form of matching cost that usually considers matching pixel intensity and a smoothness factor. This matching cost can then be used to make a cost volume with a disparity dimension where you try to minimize the cost of matching pixels across the disparity dimension.

Convolutional networks have had much success across many vision tasks, but have not been used very successfully to form disparity maps yet.

The Pyramid Stereo Matching Network [1] attempts to apply a convolutional neural network architecture to solve for a disparity map with a unique approach. The network attempts to use convolutional layers to extract features from the images at different size levels and learn a cost volume for the disparities with a neural network architecture. The Pyramid Stereo Matching Network has six main steps:

1. A Convolutional Network to pull out features within the images
2. A Spatial Pyramid Pooling Module that looks for features at different levels within the image by average pooling the image to different sizes.
3. Convolutional layers to upsample the pyramid to the same size and concatenate them together in the channel domain
4. Performing operations 1-3 continually shifting the left and right images to form a cost volume for different disparities.
5. A 3-dimensional convolutional network that tries to learn to minimize this cost volume and upsampling the output to the original dimensions of the image.
6. A softmax layer and disparity regression to add up the probabilities that a pixel in the disparity map has each disparity multiplied by that disparity. Outputs disparity guess at every pixel to form disparity map.

2 Background & Related Work

Stereo algorithms typically use matching cost computations, aggregating over some form of cost volume, minimizing the cost over the cost volume through some form of optimization, and then some disparity refinements. Recently, state of the art stereo algorithms have begun to use some form of convolutional neural network for calculating the matching cost computations and then using Semi-Global Matching [2] to calculate the disparity map. Zbontar and LeCun [3] produced a convolutional neural network architecture that can learn similarities between 9x9 patches of an image which can then be useful for a patch matching algorithm before disparity refinements.

More Recently, some network architectures have been made to handle end-to-end disparity mapping. End-to-end means the networks can handle all parts of the algorithm from receiving the image pair to creating the disparity map. Mayer et al. proposed network architecture DispNet [4] is capable of handling this using convolution, upsampling, downsampling, and add layers. Kendall et al. goes a step further in GC-Net [5] by using 3-d convolutions. SharpMask [6] and RefineNet [7] were some early studies into an stacking upsampling, downsampling architecture with residual layers - referred to as encoder-decoder networks or an hourglass structure - to look for top-down and bottom-up information via skip connections with add layers.

Pyramid Pooling, used in this network, is not unprecedented. ParseNet [8] and others have been able to use the pyramid pooling structure to find information at multiple levels of an image for semantic segmentation while SPyNet [9] and PWC-Net [10] utilize the pyramid pooling structure in this network to improve optical flow, a very similar task to disparity except there is usually a one dimensional constraint to distance in stereo and there is a two-dimensional contract to movement in optical flow.

Pyramid Stereo Matching Network is unique however. Pyramid pooling had not yet been successfully applied to stereo and end-to-end neural networks for stereo were still rare at its time. The Pyramid Stereo Matching Network is very smart in using many different methods to efficiently extract features at different sizes within the image and with different structures by using many different techniques. The Pyramid Stereo Matching Network uses downsampling and dilation, the spatial pyramid pooling module to extract features at different pooling sizes within the image, shifting the left and right images in order to build cost volumes, and then using a stacked hourglass structure to learn the best way to minimize the resulting cost volumes for disparity.

3 Method

3.1 Convolutional Neural Network

First, the Pyramid Stereo Matching Network has convolutional layers. All convolutional layers use a 3x3 filter and are followed by Batch Normalization and Leaky ReLU layers. The convolutional layers downsample to half the original size of the image and is then passed through a few more convolutional layers. Then there are four different residual blocks for learning unary features. In the first block the image is kept at half the size of the original image and put through three convolutional layers. In the second block the image is downsampled to one-fourth the original image size and put through sixteen convolutional layers. The output of this second block is saved aside for use in the Spatial Pyramid Pooling Module before the layer is fed into the third block. The first block had 32 channels per convolution, 64 channels for the second block, and that is further increased to 128 channels per convolutional layer with three layers for the third block as the convolutional neural network is further increasing the feature field to look into different features. The third block has 3 convolutional layers and uses a dilation of 2 so the filters are element-wise multiplication with alternating pixels around a central location in the image. Using dilation in this block allows the block to look into a larger receptive field for potential image features to learn from. The fourth block has an even bigger receptive field by using a dilation of 4 with 128 channels per convolutional layers for 3 layers.

3.2 Spatial Pyramid Pooling Module

The Spatial Pyramid Pooling Module takes in the output of the second and fourth block of the convolutional neural network. The fourth block is attached to the SPP average pooling output from the convolutional layers by different pool sizes (8, 16, 32, 64). Doing so allows the neural network to look for object features at different levels so the model does not only look for local features. The different pooling sizes are then fed into a convolutional layer with a 3x3 filter and 32 channels before upsampling with bilinear interpolation so the image is back to one-fourth the original size.

More variation in pooling size could further build the complexity of the pyramid and add missing information that is interpolated, however the four pool sizes that we have seem sufficient for our task.

3.3 Convolutional Layers

The four different pooling sizes in the spatial pyramid pooling module are then concatenated with the second and fourth block of the convolutional neural network in the channels domain.

The second block from the convolutional neural network has looked for some smaller features within the original image while the fourth block has looked for larger features due to its bigger receptive field. The four pooling sizes then look for features at different sizes within the image by changing the resolutions of the image to different sizes. Concatenating these six different layers together allows us to look for many different types of features within the image where we can gain both local and global knowledge about matching pixels. The layer is then put through convolutional layers to reduce the channel size down to 32. This output is then concatenated in the channel domain with its image pair so that we can look at different feature information in the different images to compare.

It is important to note the convolutional network, spatial pyramid pooling, and convolutional layers all share weights between the varying image inputs.

3.4 Forming Cost Volume

To make more layers of the cost volume for different disparity, you will shift both the left and right images to the right. The left image will be shifted by removing the first four columns and adding four zero-padding columns to the left. The right image will be shifted by removing the last four columns and adding zero-padding columns to the left. Doing these operations allow us to build our convolutional network and spatial pyramid pooling to form a cost volume at the disparity of four. Use the "fusion" focused convolutional layers to add on to the disparity dimension of the cost volume and repeat by continuing to shift the images four until your cost volume has a disparity dimension one-fourth the size of the maximum disparity. Later on we will be able to use bilinear interpolation to upsample and fill in the gaps. Since the channel domain has 64 channels, there are 64 different cost volumes that are made by our network architecture to try and learn the best cost volume.

3.5 3D-Convolutional Neural Network and Upsampling

The networks cost volumes are then fed into a 3-dimensional convolutional neural network where we will be applying 3-dimensional convolutional layers across the disparity, height, and width dimensions. Starting this part of the network architecture these three dimensions are one-fourth the final size with 64 channels. There are two different potential implementations for this network:

1. Stacked Hourglass 3D CNN
2. Basic 3D CNN

The stacked hourglass network combines downsampling and upsampling with residual layers. Residual layers add in previous layers of the same size so that the intermediate steps only have to learn a difference between the two layers instead of the whole layer itself. The network does this downsampling and upsampling architecture structure three times while setting aside an output after each step so there are three different outputs of the network. The first output is concatenated into the second output to become part of the second output and the second output is concatenated into the third output to become part of the third output. The three outputs are then upsampled with bilinear interpolation so they have a full disparity, height, and width dimensions. Each of the three outputs will be used to produce a disparity map so back-propagation will happen at three different points in the network

The basic 3d-convolutional neural network is much more simple and does not involve any strides for downsampling. The basic 3d-convolutional neural network however still does use residual layers so that layers in-between only have to learn the residual between two layers. After 12 3d-convolutional layers with four add layers residual steps in-between, the final layer is upsampled to a full disparity, height, width dimension.

3.6 Disparity Regression

After we have our DxHxW output, we then perform disparity regression. To do this, we first feed our output into a softmax layer across the disparity dimension so that there is a probability outputted that a pixel for our disparity map has each disparity. We then perform regression for every pixel we will multiply the probability of a disparity by that disparity for every disparity and sum that number up. This gives you an expected value for the disparity of that pixel based on the output of the softmax layer.

4 Experimental Results

For my own experiment, I build my own Pyramid Stereo Matching Network Architecture in Google Colaboratory using a Keras Model. For my dataset, I chose to use the KITTI-2015 Stereo dataset since this dataset was small enough for me to be able to use in Google Colaboratory easily and because the dataset is used as a benchmark for stereo. My goal was to build the full model architecture and be able to train it so that I could have my model accurately predict disparity maps of the data.

I had to make some small tweaks from the original network in order to build the model with the memory allocated in Google Colaboratory so that I could use their GPU's. I changed the amount of filters in convolutional layers before the cost volume to one-fourth the recommended amount so that less memory would be used. I also made it so the maximum disparity was 128 instead of 192 and the batch size was 4 instead of 12. These changes needed to be made in order to train the model in Google Colaboratory. The input to my model was 128x512 after resizing the image from 376x1240 to 188x720 and then cropping to the bottom left.

My network came out to have 556,309 total parameters where 482,899 are trainable. Before I made it so the original convolutional layers before the cost volume shared weights the parameter count was in the dozens of millions. I used Adam optimizer with a 0.0005 learning rate and everything else set to default. For my loss function, I implemented the L1-Smooth loss recommended in the Pyramid Stereo Matching Network paper.

$$smooth_{L1}(x) \quad = \quad \begin{cases} 0.5x^2 & if\ |x| < 1, \\ |x| - 0.5 & otherwise \end{cases} \tag{1}$$

Due to limited Disk storage in Google Colaboratory and time shortage I skipped pre-training the network on the large Scene Flow dataset and trained the network completely on the 2015 KITTI dataset. From the dataset containing 200 training image pairs, I split the dataset into a 180/20 training-validation split.

From the results of training 100 epochs, some positive results can be seen from the network. The disparity maps outputted from the Pyramid Stereo Matching Network appear to still be blurry, but did learn a lot of the actual structure of the true disparity maps. One thing that I noticed however was that the neural network seemed to be fast at forming disparity maps when performing the prediction operation on my Keras model. The Pyramid Stereo Matching Network did not lose much performance for being such a large neural network. A fast performance time is very important when looking at the practicality of an algorithm as algorithms with too long of a performance time are impractical in the real world due to many disparity maps needing to be computed, time constraints needed when calculating disparity maps may be imposed (especially for tasks such as autonomous vehicles), and algorithms with longer performance times are more computationally expensive.

I continued to train the neural network for fifty more epochs. The improvements in the accuracy of the output are very small as the network is likely close to as much as the network can learn on the given dataset with the given hyper-parameters. I would have liked to decrease the learning rate and continue, however re-compiling the network takes several hours and Google Colaboratory crashed before I had the chance.

The difference in the output images of the training and validation do not visually appear that different, although the training loss from the validation set ended up being about 1/3rd higher than that of the training set. The algorithm does appear to generalize to the dataset and learn a representation based on disparity within the image instead of just learning the data.

While the accuracy of pixels within 3 pixels of the actual disparity is relatively small at 36.23%, the network appears to correctly learn the general structures of the true disparity map visually and the within 8 pixel disparity accuracy is much higher at 64.29%

Given that the Pyramid Stereo Matching Network in the original paper had a within 3px validation error of 2.33%, my network clearly lagged a bit behind. There are a few reasons that could have contributed to my neural network not achieving the same level of efficiency as the one in the paper

1. The Network was not pre-trained on a larger Scene-Flow network containing 35,454 training images and 4,370 test images. Due to the KITTI-2015 Dataset being small at only 200 images and the network only being trained on 180 of those images, there is only so much the network can learn about extracting features from an image to form a cost volume and minimize the loss from error to the disparity map. A larger dataset is needed in order to train a complex neural network architecture to state of the art results

2. The neural network architecture was much more simple than the one used in the original Pyramid Stereo Matching Network. Due to limited memory in Google Colaboratory, the network used in this paper had one-fourth as many channels in convolutional layers leading up to the cost volume as the network in the original Pyramid Stereo Matching Network. Hav-

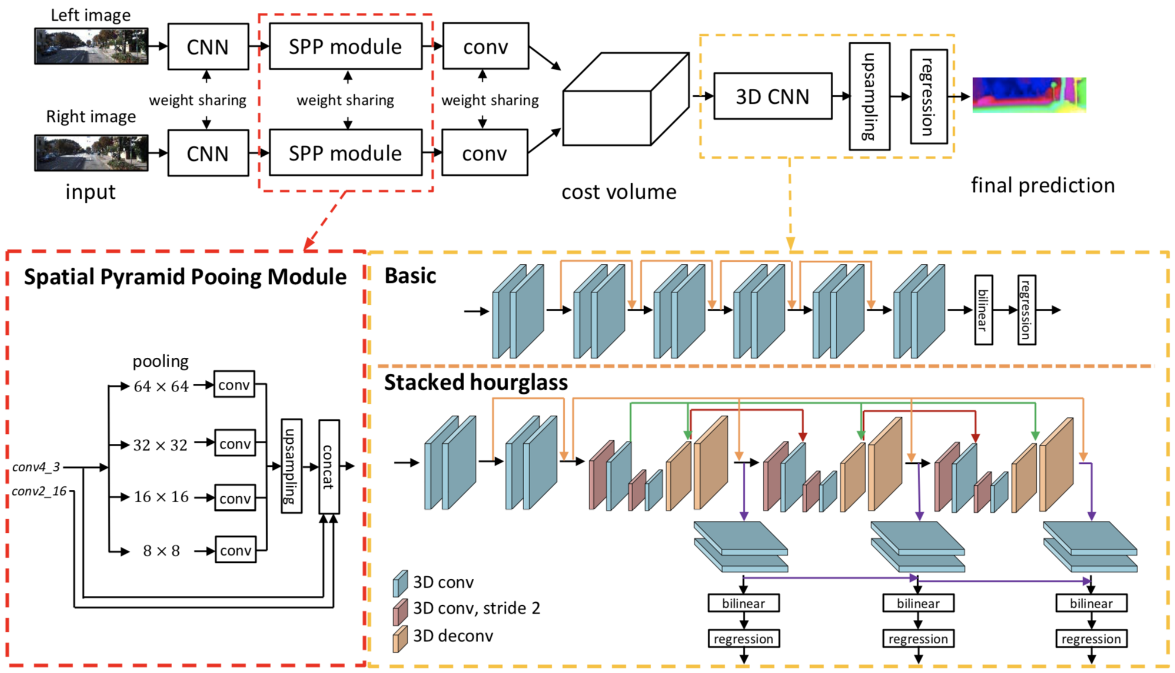
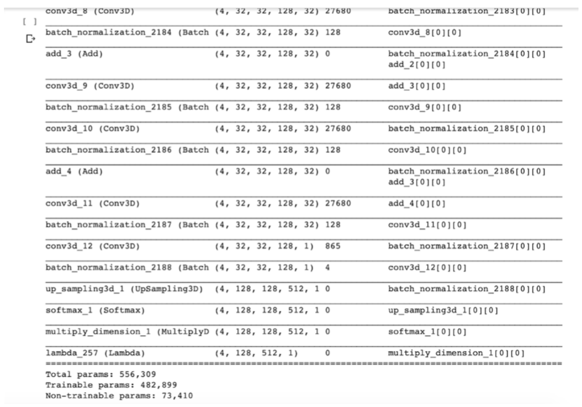
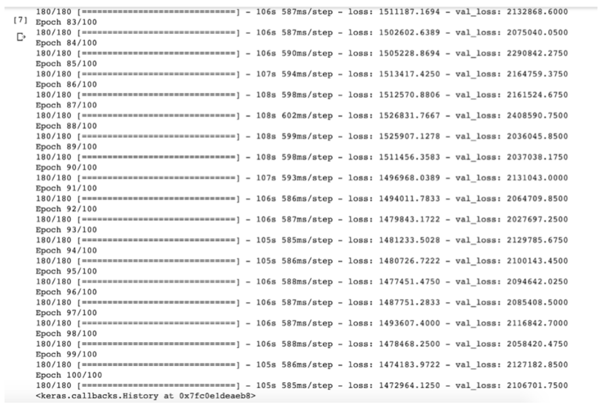


Figure 1: Pyramid Stereo Matching Network Architecture [1]



```
conv3d_8 (Conv3D)              (4, 32, 32, 128, 32) 27680       batch_normalization_2183[0][0]
__________________________________________________________________________________________________
batch_normalization_2184 (Batch (4, 32, 32, 128, 32) 128         conv3d_8[0][0]
__________________________________________________________________________________________________
add_3 (Add)                     (4, 32, 32, 128, 32) 0           batch_normalization_2184[0][0]
                                                                 add_2[0][0]
__________________________________________________________________________________________________
conv3d_9 (Conv3D)               (4, 32, 32, 128, 32) 27680       add_3[0][0]
__________________________________________________________________________________________________
batch_normalization_2185 (Batch (4, 32, 32, 128, 32) 128         conv3d_9[0][0]
__________________________________________________________________________________________________
conv3d_10 (Conv3D)              (4, 32, 32, 128, 32) 27680       batch_normalization_2185[0][0]
__________________________________________________________________________________________________
batch_normalization_2186 (Batch (4, 32, 32, 128, 32) 128         conv3d_10[0][0]
__________________________________________________________________________________________________
add_4 (Add)                     (4, 32, 32, 128, 32) 0           batch_normalization_2186[0][0]
                                                                 add_3[0][0]
__________________________________________________________________________________________________
conv3d_11 (Conv3D)              (4, 32, 32, 128, 32) 27680       add_4[0][0]
__________________________________________________________________________________________________
batch_normalization_2187 (Batch (4, 32, 32, 128, 32) 128         conv3d_11[0][0]
__________________________________________________________________________________________________
conv3d_12 (Conv3D)              (4, 32, 32, 128, 1)  865         batch_normalization_2187[0][0]
__________________________________________________________________________________________________
batch_normalization_2188 (Batch (4, 32, 32, 128, 1)  4           conv3d_12[0][0]
__________________________________________________________________________________________________
up_sampling3d_1 (UpSampling3D)  (4, 128, 128, 512, 1 0           batch_normalization_2188[0][0]
__________________________________________________________________________________________________
softmax_1 (Softmax)             (4, 128, 128, 512, 1 0           up_sampling3d_1[0][0]
__________________________________________________________________________________________________
multiply_dimension_1 (MultiplyD (4, 128, 128, 512, 1 0           softmax_1[0][0]
__________________________________________________________________________________________________
lambda_257 (Lambda)             (4, 128, 512, 1)     0           multiply_dimension_1[0][0]
==================================================================================================
Total params: 556,309
Trainable params: 482,899
Non-trainable params: 73,410
```

Figure 2: Part of Network Created



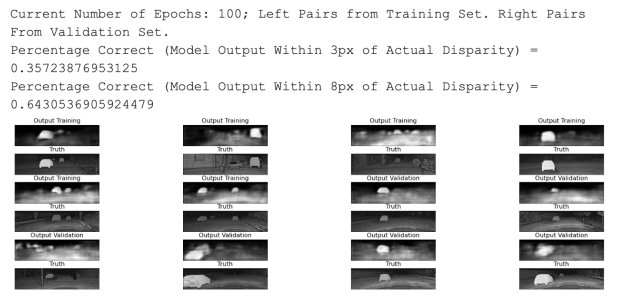
```
180/180 [==============================] - 106s 587ms/step - loss: 1511187.1694 - val_loss: 2132868.6000
Epoch 83/100
180/180 [==============================] - 106s 587ms/step - loss: 1502602.6389 - val_loss: 2075040.0500
Epoch 84/100
180/180 [==============================] - 106s 590ms/step - loss: 1505228.8694 - val_loss: 2290842.2750
Epoch 85/100
180/180 [==============================] - 107s 594ms/step - loss: 1513417.4250 - val_loss: 2164759.3750
Epoch 86/100
180/180 [==============================] - 108s 598ms/step - loss: 1512570.8806 - val_loss: 2161524.6750
Epoch 87/100
180/180 [==============================] - 108s 602ms/step - loss: 1526831.7667 - val_loss: 2408590.7500
Epoch 88/100
180/180 [==============================] - 108s 599ms/step - loss: 1525907.1278 - val_loss: 2036045.8500
Epoch 89/100
180/180 [==============================] - 108s 598ms/step - loss: 1511456.3583 - val_loss: 2037038.1750
Epoch 90/100
180/180 [==============================] - 107s 593ms/step - loss: 1496968.0389 - val_loss: 2131043.0000
Epoch 91/100
180/180 [==============================] - 106s 586ms/step - loss: 1494011.7833 - val_loss: 2064709.8500
Epoch 92/100
180/180 [==============================] - 106s 587ms/step - loss: 1479843.1722 - val_loss: 2027697.2500
Epoch 93/100
180/180 [==============================] - 105s 585ms/step - loss: 1481233.5028 - val_loss: 2129785.6750
Epoch 94/100
180/180 [==============================] - 105s 586ms/step - loss: 1480726.7222 - val_loss: 2100143.4500
Epoch 95/100
180/180 [==============================] - 106s 588ms/step - loss: 1477451.4750 - val_loss: 2094642.0250
Epoch 96/100
180/180 [==============================] - 106s 587ms/step - loss: 1487751.2833 - val_loss: 2085408.5000
Epoch 97/100
180/180 [==============================] - 106s 587ms/step - loss: 1493607.4000 - val_loss: 2116842.7000
Epoch 98/100
180/180 [==============================] - 106s 588ms/step - loss: 1478468.2500 - val_loss: 2058420.4750
Epoch 99/100
180/180 [==============================] - 105s 586ms/step - loss: 1474183.9722 - val_loss: 2127182.8500
Epoch 100/100
180/180 [==============================] - 105s 585ms/step - loss: 1472964.1250 - val_loss: 2106701.7500
<keras.callbacks.History at 0x7fc0e1deaeb8>
```

Figure 3: Network Training. Note: Epoch takes roughly 105s to run

ing so few channels prevents the network from being able to learn many different features and can result in a more blurry output as the network is not complex enough to handle the problem.

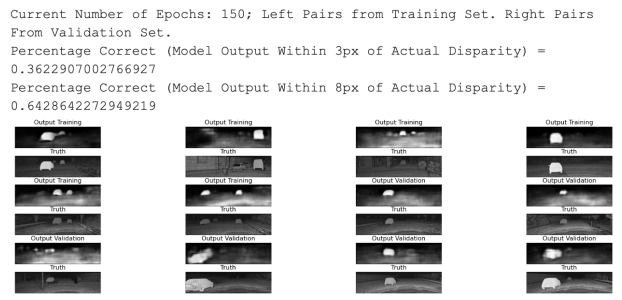
3. The batch size used in this paper was 4, whereas it was 12 in the original paper. A batch size of 4 is very small and can inhibit the ability of a complex neural network architecture to reach a local minimum. From testing, I saw a noticeable difference in the results of using a batch size of 1 versus a batch size of 2 and a batch size of 4. The network in this paper was limited due to memory limits in Google Colaboratory, but the batch size was likely too small.

4. A fourth reason why my network had less success than the one in the paper was because my maximum disparity was 128 instead of 192. Most of the disparities within the KITTI 2015 dataset were less than or equal to 128, however there were a few outliers that had disparities greater than 128 while all disparities within the dataset were less than 192. This means the



```
Current Number of Epochs: 100; Left Pairs from Training Set. Right Pairs
From Validation Set.
Percentage Correct (Model Output Within 3px of Actual Disparity) =
0.35723876953125
Percentage Correct (Model Output Within 8px of Actual Disparity) =
0.6430536905924479
```

Figure 4: Output Disparity Maps vs. Ground Truth After 100 Epochs



```
Current Number of Epochs: 150; Left Pairs from Training Set. Right Pairs
From Validation Set.
Percentage Correct (Model Output Within 3px of Actual Disparity) =
0.3622907002766927
Percentage Correct (Model Output Within 8px of Actual Disparity) =
0.6428642272949219
```

Figure 5: Output Disparity Maps vs. Ground Truth After 150 Epochs

model made in the original paper allowed for all disparities in the dataset to be correctly recorded while I could not learn a disparity greater than 128 with my network architecture.

Overall, my results show that the Pyramid Stereo Matching Network can work to form a disparity map. However, a larger dataset than the KITTI-2015 dataset is likely required in order to train such a complex neural network architecture to learn a disparity map.

5 Conclusion

Vision stereo is an important field in Computer Vision that continually sees more research done to it and improvements. Pyramid Stereo Matching Network at the time of its publishing was a state of the art approach to vision stereo that showed that neural networks could be implemented in a style resembling the techniques used by algorithms before neural networks to achieve state of the art results in Computer Vision. The Pyramid Stereo Matching Network, while reasonably low in trainable weights for a deep convolutional neural network architecture, is very high in training complexity due to the network using the same convolutional and spatial pyramid pooling weights many different times to build a cost volume. The network architecture that I built was roughly one-fourth the size of the one in the original paper and took hours to compile on a Google Colaboratory GPU. Given training a batch size of four on the KITTI-2015 dataset 0.43 seconds (average Epoch time around 105s for 45 batches in Epoch) on my smaller Pyramid Stereo Matching Network, training a single Epoch of the SceneFlow dataset on my smaller Pyramid Stereo Matching Network likely would have taken just over one-hour. The original paper trained a larger network on the SceneFlow dataset for 10 Epochs and then the KITTI-2015 dataset for 300 Epochs. Doing such takes intense training time and is very computationally expensive to specialize a disparity map algorithm for a dataset. Although there is intense training time, the actual runtime performance of the Pyramid Stereo Matching Network is very good for state of the art results where there is a within 3px validation accuracy of 2.33%. The Pyramid Stereo Matching Networks uses many useful strategies to extract useful information from images at different levels and then use old vision stereo strategies of building a cost volume to make an disparity map.

Acknowledgments

Used Keras documentation to help make custom layers and Google Colaboratories GPU's to build and train model.

References

[1] J. Chang and Y. Chen. Pyramid Stereo Matching Network, 2018.

[2] A. Sekiand and M. Pollefeys. SGM-Nets:Semi-globalmatching with neural networks. In The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), July 2017.

[3] J. Zbontar and Y. LeCun. Stereo matching by training a convolutional neural network to compare image patches. Journal of Machine Learning Research, 17(1-32):2, 2016.

[4] N. Mayer, E. Ilg, P. Hausser, P. Fischer, D. Cremers, A. Dosovitskiy, and T. Brox. A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. In The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2016

[5] A. Kendall, H. Martirosyan, S. Dasgupta, P. Henry, R. Kennedy, A. Bachrach, and A. Bry. End-to-end

learning of geometry and context for deep stereo regression. In The IEEE International Conference on Computer Vision (ICCV), Oct 2017.

[6] P. O. Pinheiro, T.-Y. Lin, R. Collobert, and P. Dollar. Learning to refine object segments. In European Conference on Computer Vision, pages 75–91. Springer, 2016

[7] G. Lin, A. Milan, C. Shen, and I. Reid. RefineNet: Multi- path refinement networks for high-resolution semantic segmentation. In The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), July 2017.

[8] W. Liu, A. Rabinovich, and A. C. Berg. ParseNet: Looking wider to see better. arXiv preprint arXiv:1506.04579, 2015.

[9] A. Ranjan and M. J. Black. Optical flow estimation using a spatial pyramid network. In IEEE Conference on Computer Vision and Pattern Recognition (CVPR), volume 2, 2017.

[10] D. Sun, X. Yang, M.-Y. Liu, and J. Kautz. PWC-Net: CNNs for optical flow using pyramid, warping, and cost volume. arXiv preprint arXiv:1709.02371, 2017.