**ECE8443 Pattern Recognition Project Report**

**Student:** Alexander W Feild

* I use the royal “we” to refer to myself as a project team in this report

**MSU NetID:** awf97

**MSU ID:** 902290841

# **Introduction**

The purpose of this effort was to develop a method that can train a model to produce stereo disparity maps without ground truth disparity maps. The unsupervised metric used was image reconstruction. Additionally, reinforcement learning was used and the problem formulated as a non-linear optimization problem. Our goal is to evolve this method to non-aligned image pairs and optical flow. However, stereo pairs are a simpler case and we used KITTI 2015 for this project. This project report compares results to a state-of-the-art method that relies on labelled data, Hierarchical Neural Architecture Search for Deep Stereo Matching. We discuss the primary problem in reinforcement learning, predicting future reward, and limitation of our unsupervised reconstruction metric. We outline which steps and potential improvements we will experiment with next. Code for this project can be found here: <https://github.com/feildawproton/StereoRL>

# **Background**

The production of disparity maps and optical flow maps have historically been done with optimization methods and manual feature descriptors. Deep learning has dominated vision classification tasks for about 8 years now. Since 2016 it has dominated stereo disparity and optical flow tasks. However, most methods require ground truth for training which reduces the availability of data. More recent research has been done into unsupervised methods. We explore an unsupervised method in the form of reinforcement learning.

Hsueh-Ying Lai et al. in their 2019 paper “Bridging Stereo Matching and Optical Flow via Spatiotemporal Correspondence” [1] used and unsupervised metric for producing stereo disparity and optical flow maps. They train a model to jointly learn stereo disparity and optical flow. They introduce geometric constraints on the learning process and utilized in their objective for unsupervised training. They use a warping function applied to the images and their 2-Warp loss measure consistency between disparity and optical flow tasks. The unsupervised loss functions they use include: self-supervised reconstruction loss, smoothness loss, left-right consistency loss, and 2-Warp consistency loss. The reconstruction loss they use is occlusion aware. The smoothness loss encourages local smoothness but allows edges. This method is state-of-the-art and we wanted to get it working but were not able to do so in time

Ryosuke Furuta and Toshihiko Yamasaki introduced PixelRL [6] to address the problem of action space size when applying reinforcement learning to image tasks. The use a fully convolutional neural network (FCN) that copies parameter across all pixels, including the action space for each pixel. This greatly reduced the memory requirement of the model (compared to one that had action spaces that addressed each pixel separately (instead of sharing weights)). PixelRL formalized each pixel as having and agent. We were greatly inspired by PixelRL to reduce the memory cost when using reinforcement learning for image operations. We don’t formalize each pixel as having an agent, but in-the-end the effect is quite similar.

Junyuan Xie et al. in their 2016 paper “Deep3D: Fully Automatic 2d-to-3D Video Conversion with Deep Convolutional Neural Networks” [2] introduce a method to produce stereo image pairs from single 2D images. What is noteworthy about their approach is that it is trained in a unsupervised manner on 3D movies. This means that there is a huge number of training samples. This paper is of note to us because they also use an unsupervised image reconstruction metric in order to train the production of disparity maps.

Wenjie Luo et al. in their 2016 paper “Efficient Deep Learning for Stereo Matching” [8] demonstrate what is now considered an archetypal supervised disparity map deep learning method. This method using Siamese networks on image pairs and ground truth labels. We investigated this method because it is considered now typical and the code is openly available. We tried getting their code working but this would have required extensive porting.

Moritz Menze and Andreas Geiger introduce the KITTI 2015 in their 2015 paper “Object Scene Flow for Autonomous Vehicles” [4]. They use LIDAR and fitting 3D models to objects in scenes in order to produce ground truth disparity and optical flow maps. This is the dataset that we use because many methods rely on ground truth data, even though our method does not. The benchmark is available here: <http://www.cvlibs.net/datasets/kitti/eval_scene_flow.php?benchmark=stereo> . Test disparity and optical flow maps can be submitted to the benchmark for evaluation. Alternative dataset exist such as those introduce by Nikolaus Mayer et al [5]. These datasets use computer generated graphics to produce datasets with absolutely accurate ground truth. These dataset can be found here: <https://lmb.informatik.uni-freiburg.de/resources/datasets/SceneFlowDatasets.en.html>

Xuelian Cheng et al. introduced LEAStereo in their 2020 paper “Hierarchical Neural Architecture Search for Deep Stereo Matching” [9]. This method implements a Neural Architecture Search (NAS) that searches a set of network operations in order to find the best architecture for a given task. In this case it has been applied to stereo matching, including for the KITTI 2015 dataset. This method requires ground truth. We were able to get the code for this method working. Additionally, the team provide their best results which we used for comparison to our method. This method is the most performant method on KITTI that also has its code available for download.

Saad Merrouche et al. describe the most common methods used to evaluate disparity maps in their 2020 paper “Objective Image Quality Measures for Disparity Maps Evaluation” [7]. Even though we did not address the quality of our disparity maps directly in this project, our disparity maps will be assessed when we submit them to benchmarks that rely on labeled, ground truth data.

Kun Zhou et al. review the state-of-the-art in stereo matching algorithms in their 2020 paper “Review of Stereo Matching Algorithms Based on Deep Learning” [3]. The go over traditional optimization and search methods but focus on how deep learning is used today. The identify three categories: 1) using deep learning to augment a traditional algorithm, such as substituting CNNs in for feature learning only; 2) Deep learning methods relying on labeled data, and; 3) Unsupervised methods. They note that all unsupervised methods rely on minimizing error between warped versions of the input image. This can be reformulated to describe our approach of image reconstruction.

# **Methods**

## Overview

The primary ideas behind this project are: 1) to use an unsupervised reward/error function for disparity/flow map production, in order to enable learning and testing on unlabeled data; 2) to allow for online learning/lifelong learning and better fitment to testing or real world data because of the unsupervised reward function; 3) to use reinforcement learning, so that the agent model can make iterative improvements to its disparity map using the reward function; 4) to minimize the size of the action space using a fully convolution neural network.

Based on our goals we cannot rely on ground truth disparity maps; the unsupervised reward function used was image reconstruction. The model produces iterative changes to a disparity map. Disparity maps index the pixels on one image to the pixels on another image. In our case the disparity map is aligned with the left image and points to the right image. Therefore we can hypothesize that reconstruction of the left image, using the disparity map and right image, is an unsupervised stand-in for the disparity map.

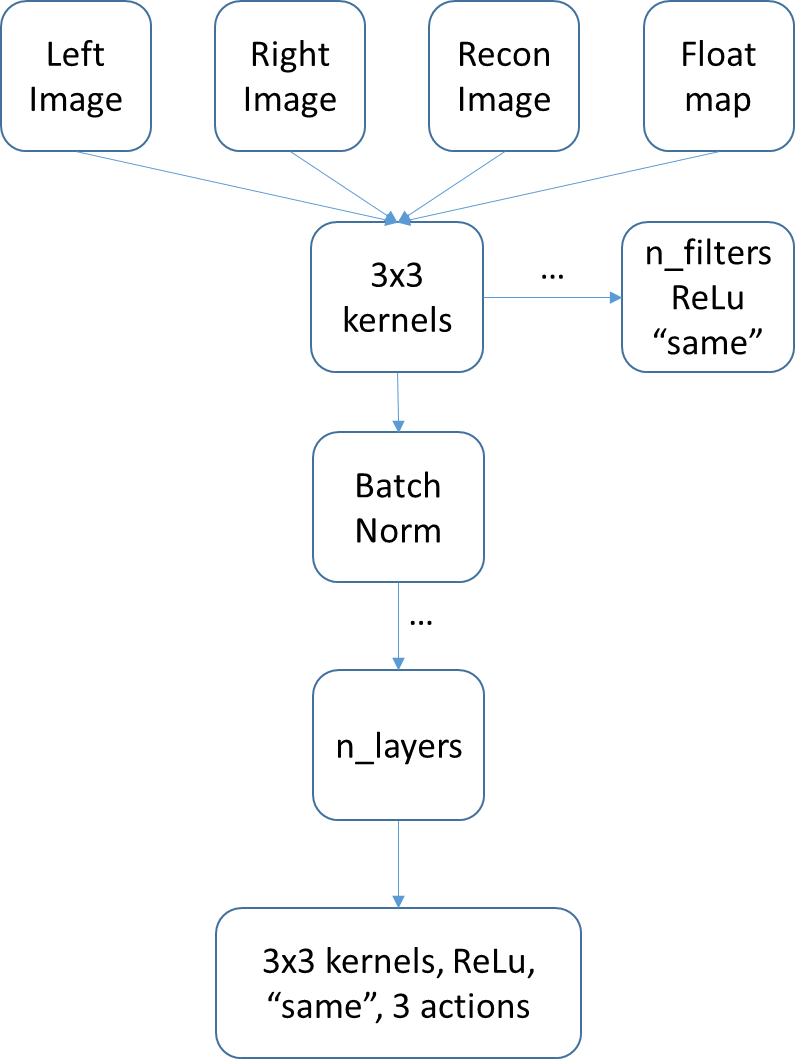
Because we are using an unsupervised reward function we can enable online learning. This allows the model iterate and improve over test/deployed datasets. As long as the model does not over-fit and lose generality an agent model can perform better over time with more iterations over user data.

Reinforcement learning is a natural fit for iterative improvements and unsupervised reward functions. While to agent model in reinforcement learning is optimized by its reward function, the agent itself is an optimizer. The agent optimizes the environment, through its action space, and makes adjustments based on the environment and its internal representation of the reward function. This makes reinforcement learning useful for optimization-like problems.

In our case we formulated to production of disparity maps as an optimization problem. RL agents cannot produced something multivalued like a disparity map. Instead they produce probabilities for taking certain actions. In our case these actions were changes to the disparity map. The disparity map starts as all zero indices (pointing to their own locations) and the agent outputs probabilities for changes to every pixel. These probabilities are maxed and each pixel is either incremented (index increased by one pixel location, decrement (index decreased by one pixel location), or nothing occurs. We don’t report experimental results for allowing the model to do nothing vs not. However, anecdotal experience suggests that allowing the model to do nothing is very important to stable learning.

One of the problems with reinforcement learning is the size of the action space. As the action space grows, memory demands increase enormously. We potentially have an unwieldly action space with potential actions for each pixel. We adopt the fully convolution neural network, inspired by PixelRL [6], in order to enable complete weight sharing. In this way weights are shared between every pixel, and every pixel effective has an agent. Because of this weight sharing, we can have much larger inputs and outputs than would otherwise be possible. The size of the model still has an effect on GPU memory when it is deployed for training due to copying into cache. Also, image size still has an effect on memory consumption, but its exponent is much reduced. For example: before moving to a FCN model a single copy of a model taking in 84x84 pixel images would take 8GB of GPU memory on our computer. Now, with a FCN, we can hold batches of 64 84x278 images in the same amount of GPU memory.

## Model Architecture

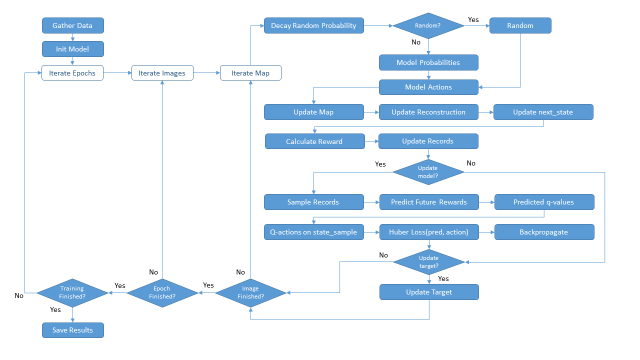


We use 3x3 convolutional kernels to reduce memory use and to follow PixelRL. Along with shallow depth this could result in the output indices not seeing enough of the image. Padding is adjusted to keep all layers the same size. ReLu activation is used on outputs since they represent probability and reward estimate jointly. PixelRL sometimes uses a convolution recurrent neural network, in the form of ConvGRU as well. We were not able to get this working in time for this project. We will implement ConvLSTM in the future. In this case the model takes in the left image, right image, reconstructed image, and an adjusted version of the map. The right image and the map are given as input state in order to enable to model to be aware of past action and better interpret current state, especially since we don’t include a recurrent portion to the model. Anecdotally, using just the left image and the reconstruction as input resulted in an inability to learn and would crash to zero reward on all tests.

## Training Architecture

While we will expand this method to non-aligned cameras we will assume aligned stereo pairs for this project. Additionally we will only be producing disparity maps that are aligned on the left image. These disparity values are enforced to be negative (i.e. index to the left). The disparity map then recreates the left image using the right image as a source of pixel values (this is where the indices point). Each entry in the map contains a relative index. Relative indices must be added to each agent’s location to get global indices. If global indices are out of range of the right image, then nothing is drawn. A recreation of the left image, from the right image and using the map is what is used for calculation reward.

Since we use q-learning, the agents cannot output a map directly. Instead, they output the probability of a reward value for taking a particular action. In our case the particular action is a step that updates the disparity map. Therefore, the map is iteratively updated. This is a challenge to learning because the reward for moving in a particular direction is most likely non-linear. That means reward will go down before it goes up



First, the training algorithm gathers the data and initializes two models. The target model is updated less often (to match the base model) and is used to calculate predicted future rewards. The base model is used to calculated q-action values. Each epoch is made up of every training image. Each image is iterated over a fixed number of times.

We use a decaying probability to determine if a random step is taken or if the agent takes an action. We didn’t perform experiments on how to include random actions. However, random search is likely to be important part of early optimization for this method. Also, random movement will help with the highly non-linear reward function. Therefore we reset the probability of taking random actions every image. Random actions are shared among all the pixels in order to not scramble the reconstruction. A random action is taken if:

Otherwise a model actions is take. The model takes the current state as input (a combination of left image, right image, reconstruction (which starts as the right image), and scaled version of the map). This model produces probabilities of taking an action. These probabilities are maxed in order to find the action to be take. This action is referenced as a number representing the index of the action with max probability.

The disparity map is updated with these actions. Actions can be to decrement, do nothing, or increment the disparity index. The disparity map and the right image is used to create a new left image reconstruction. The next state is define as the left image, right image, disparity map, and the recreation.

Reward is calculated using one of a few similarity metrics. If reward is the seen so far for a given image, we store state for saving. There are a number of records, used for predicting future reward and q-values that are updated. These include action history, state history, state next history, reward history, and map history. The state is also advanced for the next iteration.

If it is time to update the model (UPDATE\_AFTER\_ACTIONS) and there is enough for a batch then we proceed to update the model. Future rewards are predicted using a sample of next states and the target model. Predicted q-values are calculated using a reward sample and the predicted future rewards.

GAMMA is a discount factor weighing the influence of the predicted future rewards. Through the predicted future rewards and the discount factor the model learns to include future reward into its current action. With more iterations the model can predict farther into the future. In this way, a model can get over non-linearity in its reward and take actions that may result in a better late state than greedily seeking the maximum reward each step.

The model is used to calculate q-values and then q-actions from a state sample. Huber loss is calculated from the predicted q-values and q-action values. These loss values are then backpropagated through the base model.

Every so often we need to update the target model (UPDATE\_TARGET\_NETWORK). Weights are copied from the base models. When appropriate states, models, and statistics are saved.

## Software

This project was done in Python using Tensorflow and Keras. We started with the Keras tutorial on Deep Q-Learning as and outline: <https://keras.io/examples/rl/deep_q_network_breakout/> . We also use values from the PyTorch tutorial on Deep Q-Learning: <https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html> . This tutorial uses a convolutional neural network to make the same problem more difficult and fundamental. Almost nothing about these tutorial were left unchanged. Even though this work references ideas from PixelRL we did not use their codebase, even though it is open and available. PixelRL uses Chainer and a complex project structure that we did not have time to figure out. It was much faster (necessary for this project) to re-implement some of the ideas in Tensorflow and Keras.

This project also uses Numba CUDA for the mapping functions that recreate the left image from the right image and a disparity map. Numba CUDA allows us to write CUDA kernels directly in python using the decorator @cuda.jit. This was necessary because the mapping function is very parallelizable and data intensive and would be unworkable slow in python and on the CPU. Also, it was not possible to write the mapping algorithm using Tensorflow’s GPU based parallel math functions. Numba CUDA is also much easier than writing in CUDA C then wrapping and importing the modules. Numba CUDA requires wrapping kernels with a function to copy data (a similar pattern to data copy and bind in CUDA C).

# **Experiments**

## Overview

All experiments involved training an agent model to iteratively produce disparity map, through action probabilities, from stereo disparity pairs. Rewards values came from reconstruction similarity metrics. The reconstructed image is supposed to approximate the left image, given the disparity map (aligned to the left image), and the right image (which the disparity map points to). Ground truth disparity is not used to training.

After training we test each model on a test dataset. Since this is reinforcement learning and the similarity metric is unsupervised, we can leave the training parameter on. For the most part, we limited the number of attempts at the testing dataset to 1 epoch. It would be perfectly possible for this method to iterate many times over a deployed dataset and let the agent model get better at its job. We performed one test with multiple training iterations over the testing data. It may seem illegitimate to train on the test dataset. However, this is a natural result of the optimization like nature of reinforcement learning algorithms. This is simply letting the agent model optimize on a given problem.

## Dataset

We used the KITTI 2015 dataset from [4] for these experiments (website: <http://www.cvlibs.net/datasets/kitti/eval_scene_flow.php?benchmark=stereo>). This dataset contains left and right stereo aligned cameras and ground truth disparity maps for training. Ground truth disparity maps exist for testing data but they are not given to researchers (so that nobody cheats). This datasets also contains temporal optical flow images and ground truth optical flow maps, though we didn’t use them here. There are 400 training stereo pairs and 400 testing stereo pairs. Per the benchmark, testing only occurs on 200 of the 400 stereo pairs, since half of the dataset is used only for optical flow. Since the method we compared against (LEAStereo) only uses 200 testing pairs, we did the same.

In the end we used 400 training stereo pairs, 200 testing stereo pairs, and no training disparity maps (for our method). We also reduced resolution of the images for this project. This was done to allow for faster iteration and less care given to memory management. Once we have a promising method and model we will move to full resolution. Full resolution is required to compete in the online benchmark.

## Compute Platform

Experiments were ran on a single NVidia GTX 1080 with 8GB of memory. Limits on the number of filters per convolutional layer, filter size, batch size, and image size had to be observed in order to not run out of memory. Additionally, limits on the number of epochs per model in order for runs to be completed in time for updates. Learning rate for the Adam optimizer was also kept high, perhaps higher than optimum, for the same reason.

## Other Methods for Comparison

We wanted to get Bridging Stereo Matching and Optical Flow via Spatiotemporal Correspondence [1] working using the paper’s corresponding github repository. This required rewriting kernels to update to a newer version of CUDA than what the reference project uses. Additionally this required changing scripts and compiling C packages. However, dependencies were not resolved in time for results analysis.

We tried getting the Efficient Deep Learning for Stereo Matching [reference] repository working. However, this would have required updating the whole project to the latest version of Tensorflow. We did not have update completed in time for analysis.

We did get LEAStereo from [9] working here. In addition to that we got the best KITTI 2015 that the LEAStereo team was able to produce. We decided to use these for comparison. We reduced the resolution to make the comparison fairer. LEAStereo relies on ground truth disparity maps for training.

## Hyperparameters

We did not vary as many hyperparameters as we would like due to time and compute limits. Hyperparameters that we did up varying and testing include:

* Number of Epochs (NUM\_EPOCHS)
* Number of hidden layers (NUM\_LAYERS)
* Whether we batch normalize or not (BATCH\_NORMALIZE)
* Number of filters per layer (FILTERS\_PER\_LAYER)
* Reconstruction Similarity Method (\_similarity\_error)
* Number of inputs (not reported here) (NUM\_IMAGE\_INPUTS + 1)
* Batch size (BATCH\_SIZE)

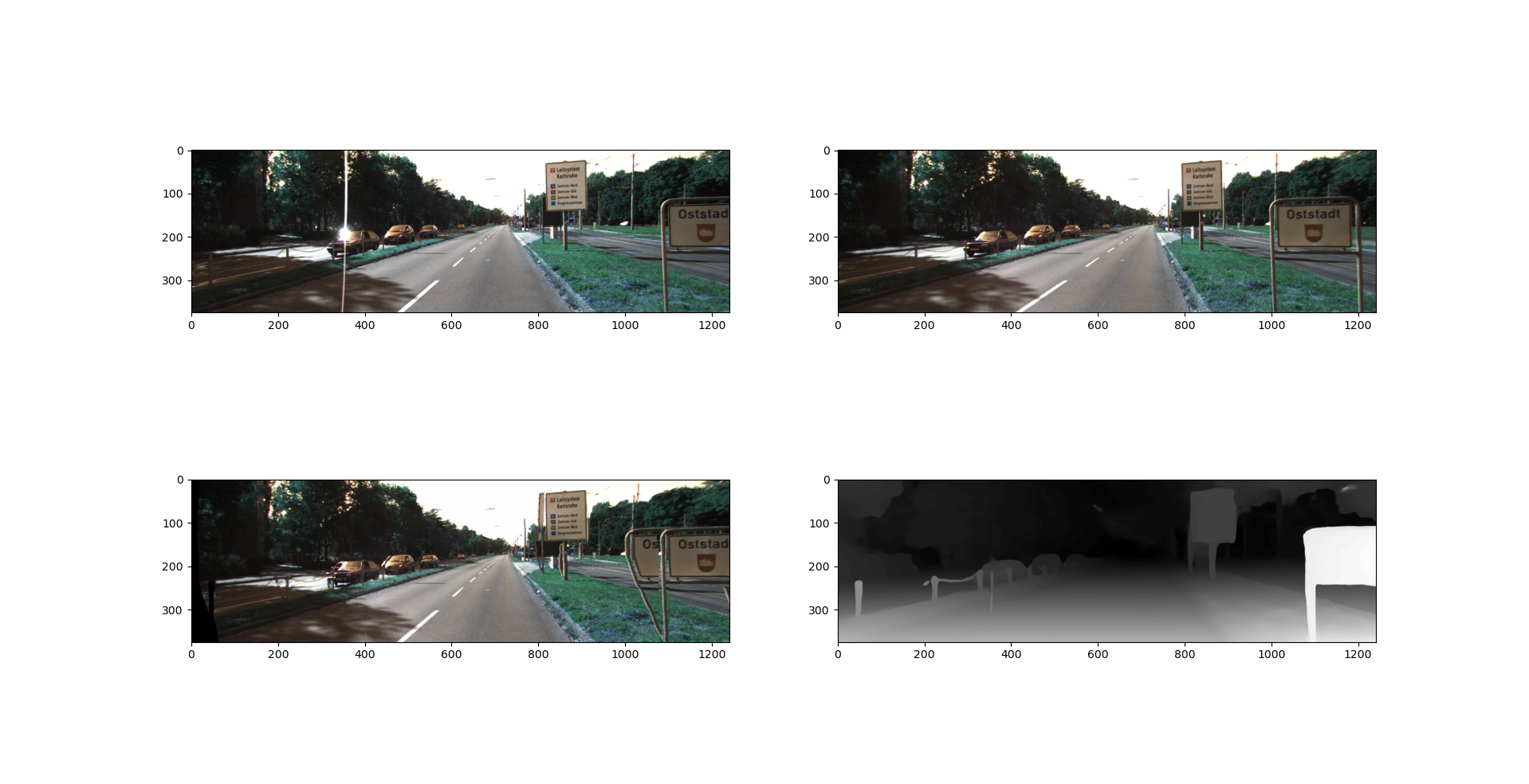
Many other hyperparameters were worth testing but we did not. Those include:

* Discount factor for predicting future rewards (GAMMA): 0.99
* How often to update target network (UPDATE\_TARGET\_NETWORK): len(images) / 10
* Loss function (LOSS\_FUNCTION): Huber
* Epsilon start value for decaying probability of random action (EPS\_START): 0.9
* Epsilon end value for decaying probability of random action (EPS\_END): .05
* Number of iteration steps for a single image (MAX\_STEPS\_PER\_IMAGE): 512
* Epsilon decay for decaying probability of random action (EPS\_Decay):
  + MAX\_STEPS\_PER\_IMAGE / 4
* How often to perform backpropagation (UPDATE\_AFTER\_ACTIONS): 8
* Learning rate for Adam optimizer (LEARNING\_RATE): 0.001
* Kernel x,y dimensions (KERNEL\_DIMS): 3X3

# **Results Analysis**

## Reconstruction Similarity Function

We used the results from LEAStereo [9] for comparison to our results. We also used LEAStereo to baseline the utility of our reconstruction similarity function. LEAStereo is the most performant of the methods tested on the KITTI 2015 dataset that also has its code readily available. It relies on ground truth data. LEAStereo outputs positive valued disparity maps that are aligned to the left image and point to the right image. This means that disparity values will need to be made negative. Additionally, disparity maps are stored as 16 bit pngs. Depending on the package used to open disparity images, the values may need to be scalled by 1/256. This is the case with Tensorflow’s image package but not with openCV. Indices are relative to their own pixel location; relative indices must be added to disparity pixel location to get the location on the right image.



Example result from LEAStereo on a test image pair. The top images are the left and right cameras. The lower right image is the disparity map (aligned with the left image and pointing to the right). The lower left image is the reconstruction of the left image using the map and the right image. Note the black on the left side of the reconstruction; this is expected. Also, not the error in duplicating the Otrstadt sign.

Assuming the results from LEAStereo to be good, we wanted to compare them against the reconstruction similarity between simply the left and right image. If the disparity estimation method works, and this one does, and our reconstruction similarity metric is valid, the similarity between the left image and the reconstruction must be higher than the similarity between the left and right images.

|  |  |  |  |
| --- | --- | --- | --- |
| LEAStereo Results | | full resolution | reduced res |
| left/right | Mean | 0.98879545 | 0.984352664 |
| Naïve | Stddev | 0.005035799 | 0.006459251 |
| Recreation | Mean | 0.964195463 | 0.968944835 |
| Naïve | Stddev | 0.01262049 | 0.011797016 |
| Recreation | Mean | 0.976425758 | 0.980948733 |
| Boost = 0.5 | Stddev | 0.008160826 | 0.007083238 |
| Recreation | Mean | 0.982540904 | 0.986950679 |
| Boost = 0.75 | Stddev | 0.00647343 | 0.005189422 |
| Recreation | Mean | 0.984008538 | 0.988391142 |
| Boost = 0.81 | Stddev | 0.006174637 | 0.00484361 |
| Recreation | Mean | 0.986209995 | 0.990551848 |
| Boost = 0.90 | Stddev | 0.005832285 | 0.004448236 |

In the table above we see the Left-Right similarity using a naïve metric was better than the reconstruction that was derived from the LEAStereo disparity map. This surely means our naïve metric is not a good one for disparity map and image reconstruction optimization. This is surely from pixel indices that go out of range. Parts of the left image will not be in the right image because they do no overlap. In the image reconstruction these pixels appear black. Our naïve metric is simply cosine similarity of color values. While values will be poorly matched between left and right images, it must be better (on average) than having a large number of black/blank pixels.

Because of this we tested boosting the reconstruction value of out of range pixels. Instead of zero (0) they returned the value “Boost” in the table above. In theory this number should be a balance between exploration (letting the model throw the correct pixels out of range), and preventing the model from gaming the reward function and throwing too many pixels out of range. This boost value should also result in better reconstruction similarity than left-right similarity. We went on to test a boost value of 0.9.

## Number of Epochs

We were not able to run a large number of epochs on our models. The most we were able to overnight was 5. The tables below compares 3 and 4 epochs and 4 and 5 epochs. We don’t see a significant difference. Perhaps this is to be expected. In the future we should run thousands of epochs with a lower learning rate.

|  |  |  |
| --- | --- | --- |
| Comparing Num Epochs (Naïve similarity, 4 inputs, 4 Normed Layers, 64 filters, 3x3 Kernel Dims, .001 LR) | | |
| Epcochs | Mean Similarity | StdDev Similaity |
| 3 | 0.985606322 | 0.006059277 |
| 4 | 0.98576417 | 0.006091498 |

|  |  |  |
| --- | --- | --- |
| Comparing Num Epochs (Naïve similarity, 4 inputs, 5 Normed Layers, 32 filters, 3x3 Kernel Dims, .001 LR, batch size 32) | | |
| Epcochs | Mean Similarity | StdDev Similaity |
| 4 | 0.985653708 | 0.006047823 |
| 5 | 0.985138951 | 0.006138407 |

## Number of layers

Our models had at most 5 hidden convolutional layer, along with batch normalization layers. This is not a lot for deep learning. This combined with our small kernels means that each pixel agent does not see a lot of the input images. Depths greater than 5 hidden layers were not attempted because of out-of-memory issues. We did not observe a difference depending on the number of hidden layers (again, perhaps because the difference was not significant).

|  |  |  |
| --- | --- | --- |
| Comparing Num Layers (Naïve similarity, 3 epochs, 4 inputs, 3x3 Kernel dims, 64 filters, .001 Learning Rate) | | |
| Normed Layers | Mean Similarity | StdDev Similaity |
| 3 | 0.985701586 | 0.006042564 |
| 4 | 0.985606322 | 0.006059277 |

## Batch Normalization

We wanted to see if batch normalization made a difference for our models. If batch normalization was used, batch normalization layers were inserted after every hidden convolutional layer. The table below shows a noticeable, though not significant, difference between using batch normalization and not.

|  |  |  |
| --- | --- | --- |
| Batch Normalization vs not (Naïve similarity, 3 epochs, 4 inputs, 3 layers, 3x3 kernel, 64 filters, .001 Learning Rate) | | |
| Batch Normed | Mean Similarity | StdDev Similaity |
| No | 0.984587062 | 0.006383839 |
| Yes | 0.985701586 | 0.006042564 |

## Number of Filters

For simplicities sake, and borrowed from PixelRL, every hidden layer used the same number of filters. The table bellow compares 32 and 64 filters per batch normalized layer. A significant difference was not observed. At depths greater than 4 layers, we could not use more than 32 filters or we would run out of memory on our GTX 1080.

|  |  |  |
| --- | --- | --- |
| Comparing Num Filters (4 epochs, 4 inputs, 4 Normed Layers, 3x3 Kernel, .001 Learning Rate) | | |
| Num Filters | Mean Similarity | StdDev Similaity |
| 32 | 0.985622946 | 0.005981981 |
| 64 | 0.98576417 | 0.006091498 |

## Similarity Metrics

As noted above our naïve similarity metric punished out of bounds pixels. However, this is inevitable. We tested not accounting for out of bounds pixels, boosting their reconstruction similarity to 0.9, and setting them to white in the reconstruction. The goal is to balance algorithm and model exploration with exploitation. We either did not achieve this balance with these metrics or there was some other problem that made the similarity metric irrelevant.

|  |  |  |
| --- | --- | --- |
| Compairing Similarity Methods (5 epochs, 4 inputs, 5 normed layers, 3x3 kernels, 32 filters, batch size 32, .001 Learning Rate) | | |
| Similarity Method | Mean Similarity | StdDev Similaity |
| Naïve | 0.985138951 | 0.006138407 |
| 0.9 Boost | 0.985942968 | 0.006033917 |
| White Background | 0.985977651 | 0.005776801 |

## Comparison to Nothing and the State-of-the-Art

In the table below we report the similarity metric for our method, LEAStereo, and the Left-right similarity. Note these are for reconstruction, not for disparity estimation. Our method is noticeably better than doing nothing, though not significantly so. LEAStereo, a best in class KITTI 2015 disparity map estimator, did the best. The mean similarities of our method and doing nothing are outside the first standard deviation of LEAStereo’s reconstruction similarities.

|  |  |  |
| --- | --- | --- |
| Method | Mean Similarity | StdDev Similarity |
| Left-Right similarity | 0.984352664 | 0.006459251 |
| Our method 0.9 Boost | 0.985942968 | 0.006033917 |
| LEAStereo 0.9 Boost | 0.990551848 | 0.004448236 |

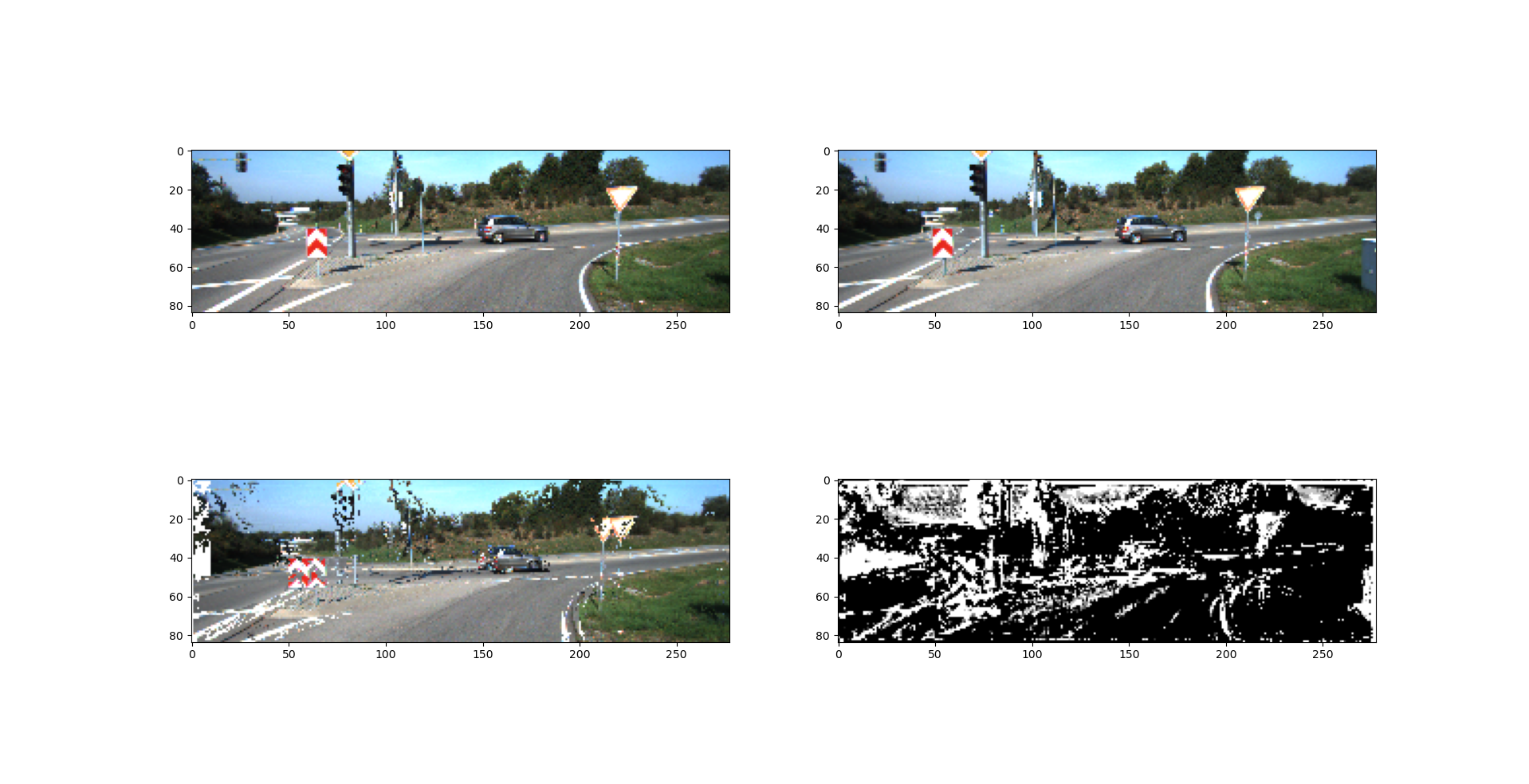
## Online Learning Iterations

Since out method relies on reinforcement learning and an unsupervised reward function it is possible to enable online learning. In this case the agent model is allowed to update itself to receive better reward on a test dataset or after the model is deployed. In this manner a model could get better at the task a user wants or simply make improvements to its disparity estimations with more iterations. In the tables below we compare 1 iteration vs 5 iterations over the test dataset for two models with different similarity metrics. We see a noticeable, if not significant, improvement in mean similarity. With more iterations, and compute power, this may become significant.

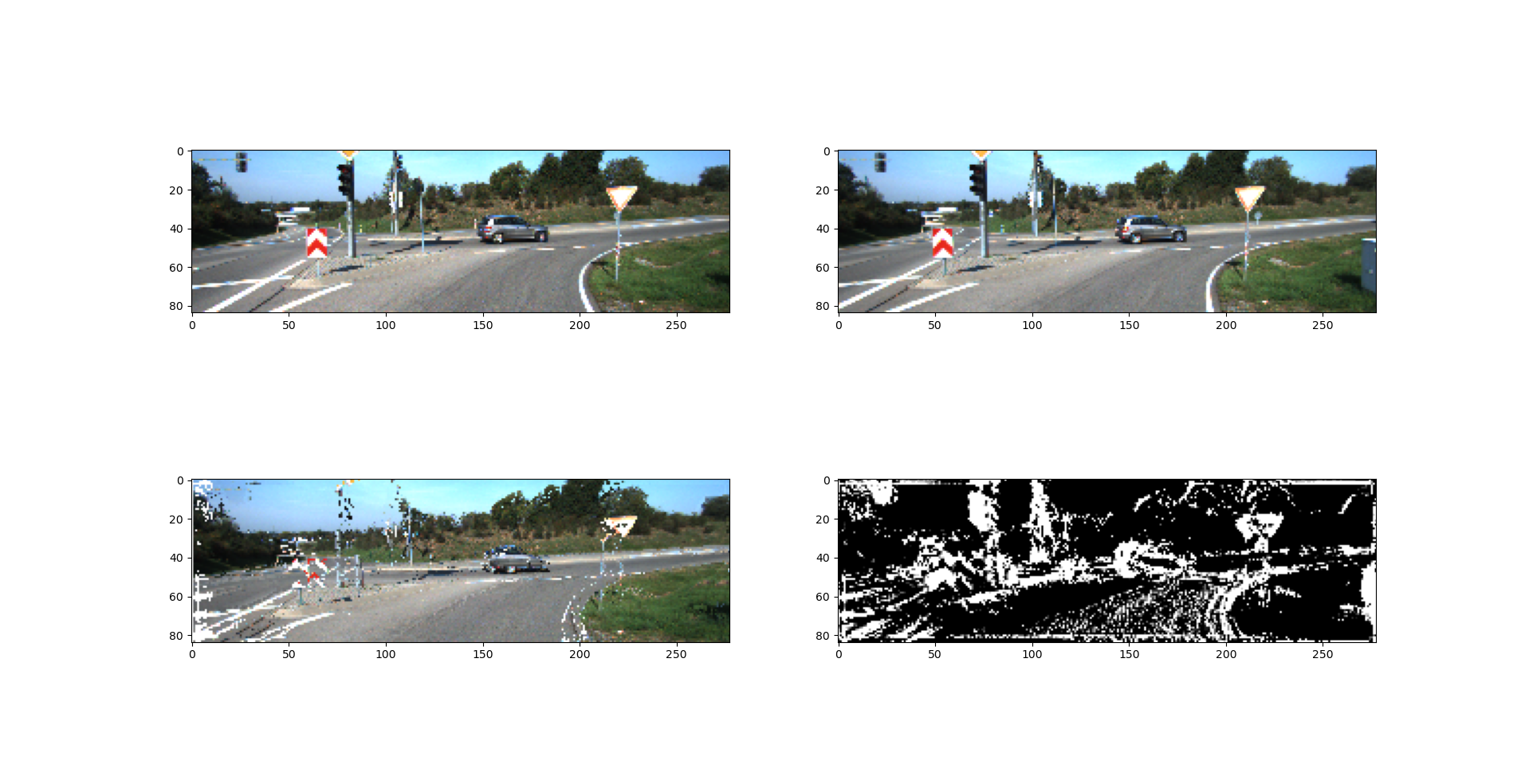
|  |  |  |
| --- | --- | --- |
| Comparing iterations online learning iterations over training data for white-background methods. | | |
| Similarity Method | Mean Similarity | StdDev Similaity |
| 1 iter | 0.985977651 | 0.005776801 |
| 5 iter | 0.986205734 | 0.005946595 |

|  |  |  |
| --- | --- | --- |
| Comparing iterations online learning iterations over training data for 0.9 boosted methods. | | |
| Similarity Method | Mean Similarity | StdDev Similaity |
| 1 iter | 0.985942968 | 0.006033917 |
| 5 iter | 0.986209498 | 0.006020598 |

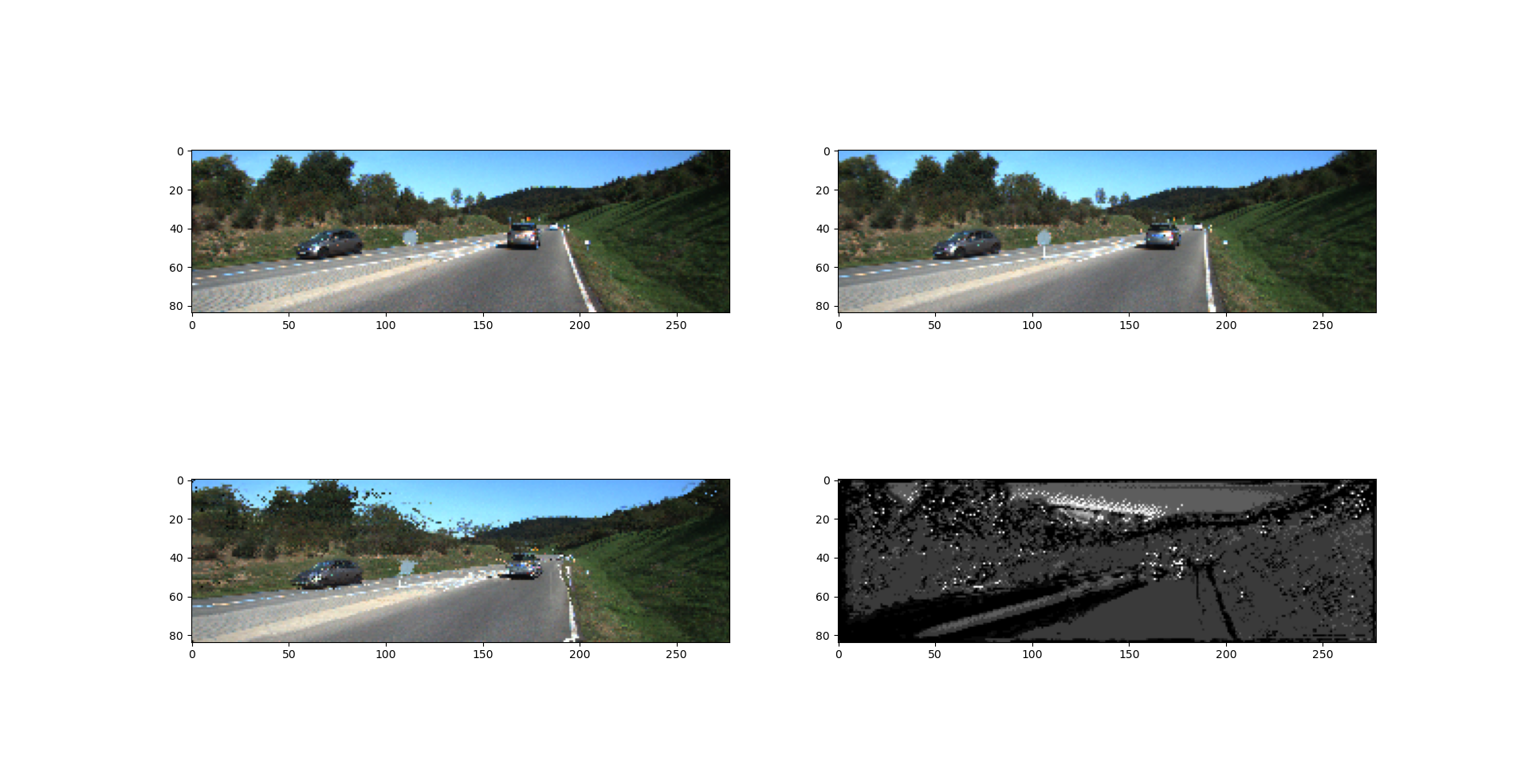
## Subjective Analysis of Some of Our Reconstructions



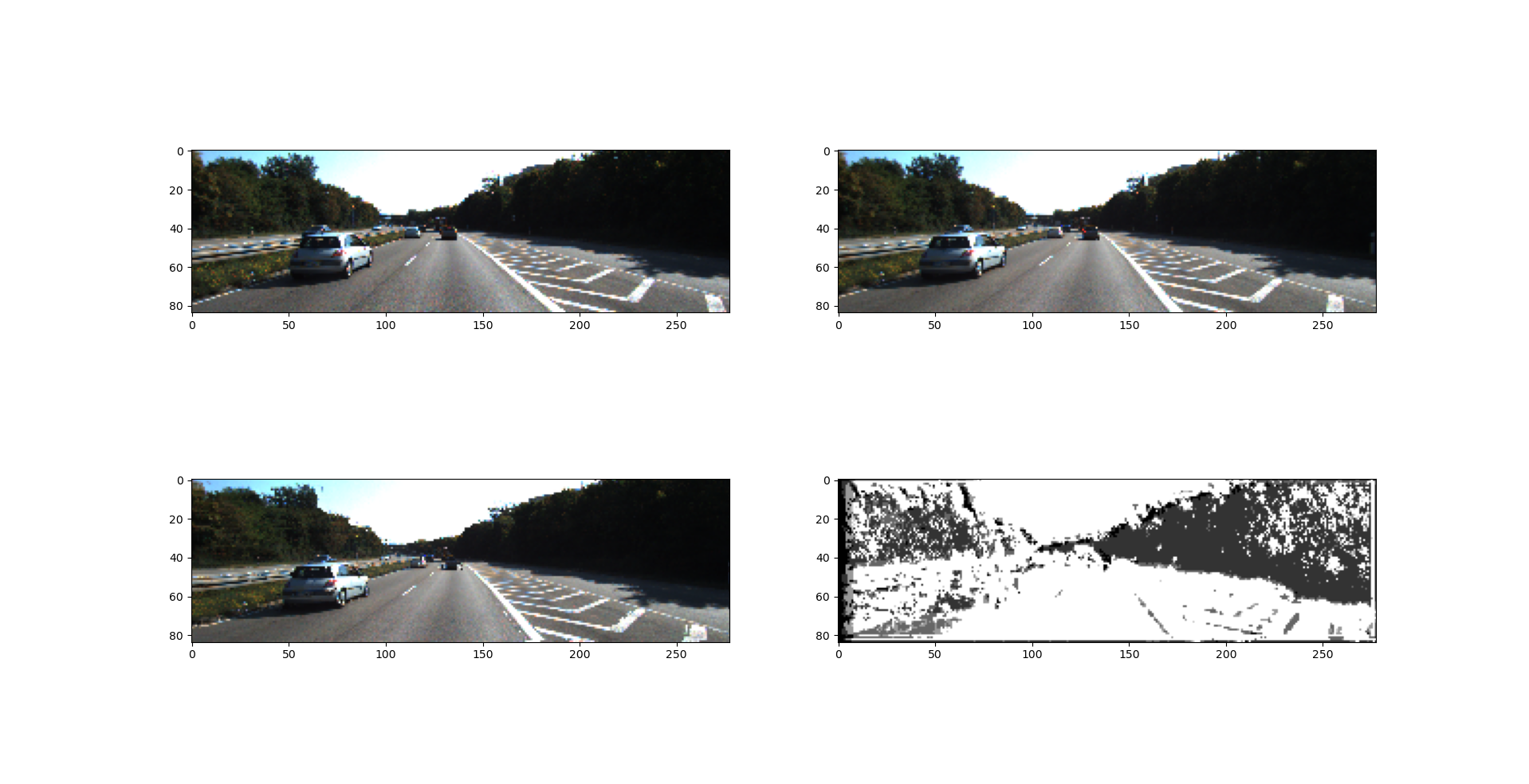
This is an example test result from a 5 normed layer model with a similarity metric that substituted white pixels for out of range indices. Notice in the reconstruction that about half of the each street sign and some of the road marks are quite well aligned, while the others left in their original location. Perhaps this model would have done better with more iterations over these images or with more training epochs. We believe this clearly demonstrates learning the objective and that we are on a decent path.



Same input scene and model as above. However, the agent model was afforded 5 iterations over the testing dataset. The model was allowed to learn online since the reward function is unsupervised and this is likely how the model would be deployed. This is an example of online/lifelong learning.



This is an example from testing a 5 normalized hidden layer model, trained with 0.9 boost to out-of-range-indices for 5 epochs. Though the image is scramble a bit we can see that the agent did a decent job of aligning most of the image. Perhaps this model would have benefited from more training or more parameters.



This is an example test result from a model trained with 4 normalized layers with 64 filters each for 4 epochs. This agent did a decent job of moving the car over for alignment.

# **Conclusions**

* more epochs
  + better inclusion of future reward
* lower learning rate
  + combined with more epochs and we may get a better search
* more depth
  + especially for kernel size 3x3 (borrowed from PixelRL)
    - this means that each agent did not see a lot of the images
  + On a better compute platform perhaps (HPC?)
* More GPU memory
* Recurrent NN
  + In the form of ConvLSTM probably
  + Likely lead to better prediction of future reward
* Better thought out reconstruction metric
  + But don’t want to encourage the model to throw indices out of range because that is easier and gets sufficient reward
* Basically need a better estimate of future reward
* Need to test how random steps should be implemented
  + Compare over image, over epoch, over entire training
* This methods does not relying on camera alignment and should expand to optical flow (x, y indexing) fairly easily, though this would require more training and computation. This method also does not relying on knowing camera properties or even camera calibration. Cameras could even have separate properties, making this method useful in situations that more traditional methods would not be. This method would be a first good step in correspondence matching
* This method is also theoretically less sensitive to textures. Moving forward we could test this.
* Should we test disparity and flow smoothness. Is this a natural consequence of sharing the weights between all pixel agents?
* We could also compare to a classifier that outputs complete maps each iteration. We will still use the same unsupervised recreation metric.
* Perhaps during training we could compare to ground truth disparity for our sake. We would not let the model or training algorithm see this data. It would be for human use in iterative model and method adjustments only.
* More iterations over testing/deployed images
  + Online learning
  + Results indicate that this would work
* Explore more kernels
* Explore more random actions and/or more iterations over image.

# **References**

[1] Hsueh-Ying Lai, Yi-Hsuan Tsai, Wei-Chen Chiu, “Bridging Stereo Matching and Optical Flow via Spatiotemporal Correspondence” *arXiv:1905.09265v1 [cs.CV]* 2019 May 22

[2] Junyuan Xie, Ross Girshick, Ali Farhadi, “Deep3D: Fully Automatic 2D-to-3D Video Conversion with Deep Convvolutional Neural Networks” *arXiv:1604.03650v1 [cs.CV]* 2016 April 13

[3] Kun Zhou, Ziangxi Meng, Bo Cheng, “Review of Stereo Matching Algorithms Based on Deep Learning” *Hindawi, Computational Intelligence and Neuroscience, Volume 2020, Article Id 8562323* 2020 March 23

[4] Moritz Menze, Andreas Geiger, “Object Scene Flow for Autonomous Vehicles” 2015

[5] Nikolaus Mayer, Eddy Ilg, Philip Hausser, Philipp Fischer, Daniel Cremers, Alexey Dosovitskiy, Thomas Brox, “A large Dataset to Train Convolutional Networks for Disparity, Optical Flow, and Scene Flow Estimation”

[6] Ryosuke Furuta, Toshihiko Yamasaki, “PixelRL: Fully Convolutional Network with Reinforcement Learning for Image Processing” *arXiv:1912.07190v1 [cs.CV]* 2019 Dec 16

[7] Saad Merrouche, Milenko Andric, Boban Bondzulic, Dimitric Bujakovic “Objective Image Quality Measure for Disparity Maps Evaluation” *MDPI Electronics* 2020 Oct 02

[8] Wenjie Luo, Alexander G. Schwing, Raquel Urtasun, “Efficient Deep Learning for Stereo Matching” *ct.toronto.edu* 2016

[9] Xuelian Cheng, Yiran Zhong, Mehrtrash Harandi, Yuchao Dai, Xiaojun Chang, Tom Drummond, Hongdong Li, Zongyuan Ge, “Hierachical Neural Architecture Search for Deep Stereo Matching” *34th Conference on Neural Information Processing Systems*, 2020, Vancouver, Canada