An Analysis and Implementation of Various Topics in Computer Vision

Joseph Kawiecki

Abstract

The representation of scene geometry for computer systems is a challenging topic within the field of computer vision. As someone interested in working a career within robotics, this is a problem I am excited to investigate and hopefully contribute towards. In the following, I examine, implement, and extend a promising solution to this problem, titled *CodeSLAM - Learning a Compact, Optimisable Representation for Dense Visual SLAM* (Bloesch et al., 2018). This implementation is preceded by an analysis of three different methods within computer vision all aiming to solve unique problems. The first method, already discussed, titled CodeSLAM was from a paper submitted to the Conference on Computer Vision and Pattern Recognition (CVPR) in 2018. The second method, titled *Binary TTC: A Temporal Geofence for Autonomous Navigation* (Badki et al., 2021), is from a paper submitted to the Conference on Computer Vision and Pattern Recognition (CVPR) in 2021. Binary TTC aims to tackle the issue of detecting rapid depth changes in vehicles, such as during a collision, through the use of a binary geofence. The third method, titled *SinGAN: Learning a Generative Model from a Single Natural Image* (Shaham et al., 2019), is from a paper submitted to the International Conference on Computer Vision (ICCV) in 2019. SinGAN aims to improve General Adversarial Networks (GANs) by presenting a method to produce a high-performance model capable of generating fake images of quality to fool humans through only a single training image. The following will include a deeper dive into each of these methods followed by a model implementation and improvement of CodeSLAM.

1 CodeSLAM

1.1 CodeSLAM - Summary

The efficient, high-quality, and real-time depiction of 3D geometry for image perception systems is a critical, yet challenging problem in the area of computer vision. There are two primary approaches that include constructing dense geometric mappings and sparse mappings. As one can infer, dense mappings contain significantly more geometric information, yet are computationally costly relative to sparse mappings that only capture partial scene data. CodeSLAM aims to bridge this gap by generating a compact, yet dense geometric scene representation for a single image, single vision-based simultaneous localization and mapping algorithm, or SLAM, environment.

To accomplish this, CodeSLAM utilizes image intensity data fed into an autoencoder neural network specifically to generate predictions for pixel depth and uncertainty. These outputs are then entered into a cost function in order to continually optimize the network. If overlapping images are used, further optimization will be performed by smoothing out the warped edges. Finally, CodeSLAM makes use of the generated depth values along with camera position and orientation data to construct a Structure from Motion (SfM) model. This model can then be used to infer scene geometry and motion in a dense SLAM mapping.

In terms of results, CodeSLAM was successfully able to represent image scenes optimizing for both geometry and motion. However, future work aims to move beyond a single image 3D geometry approach to a real-time approach and eventually real-time 3D object recognition.

1.2 CodeSLAM - Review

CodeSLAM appears to be a very promising development in computer vision, yet is still very early. In the current stage, CodeSLAM does not seem to be very practical in the scope of the real-world expected use cases of dynamic vehicles due to its limited abilities with single image keyframes. As such, it was important for the authors to mention future plans including real-time 3D dense geometry representation and object recognition. Another strength of CodeSLAM is its integration of both geometry and motion representation in SLAM. By tracking the camera position and orientation along with the depth outputs from the encoder network, CodeSLAM is able to track the current camera position against the last single keyframe image. This approach alternates between tracking and mapping, which is accomplished via depth inference through the encoder network.

One question I had while reading is how does a user know how many single image keyframes are optimal to represent the 3D geometry. I assume there is a tradeoff between the accuracy of geometric representation (measured via RMS error) and computational cost, although this is not mentioned in the paper. Computational cost is likely not a significant concern in the early development of CodeSLAM, but I would think with the progression to real-time geometric recognition that it would be. Furthermore, on this topic, where in the scene is the optimal placement of the single image keyframes relative to the master frame? Should they be nearby to leverage overlap optimization or further away to gain a greater view of the scene?

Another potential weakness and question I had while reading was the implementation of overlap optimization. As I understand, if two views are overlapping, their overlapping depth predictions can be optimized to produce better estimates. However, the authors describe that this requires computing correspondences between the two views, which is computationally expensive. As such, would it be advantageous to spread out the selection of single image keyframes so they don't overlap? Further, what would the performance impact be of entirely removing the overlap optimization? I assume accuracy would decrease, but by what factor compared with the resulting increase in computation speed?

2 Binary TTC

2.1 Binary TTC - Summary

Regardless of the application, path planning in computer vision often significantly relies on depth information. Many technologies exist to accurately collect depth information, one being single view or monocular camera vision. However, estimating the depth of a pixel is a challenging problem often requiring specific filtering or strong prior training data. For the specific situation of predicting head-on collisions, from a practical standpoint, an object approaching the observer is much more important than an object moving away. Thus it makes sense and is significantly easier, in this case, to use time to contact (TTC) instead of depth to predict collisions. Time to contact (TTC) is the amount of time before the collision between an object and observer. Binary TTC is a new approach to first estimate the TTC of each object pixel and then determine from a reference if the object pixel size has increased, meaning the object is approaching, or decreased, indicating the object is moving away. Through this data for each pixel, a temporary geofence can be built around the observer to predict TTC.

Binary TTC builds this geofence by taking in two sequential images and a scale factor. Computing each object pixel's TTC, these values are then fed into a network to compare against a scaled image TTC reference to determine where the object has moved. The network then outputs a probability map that with thresholds can be classified into a binary TTC to predict future object movement. Furthermore, using the same method architecture described, a continuous TTC can be found by computing multiple binary TTC's in parallel pending the hardware used.

In terms of results, Binary TTC has a similar level of performance to competitors but is at least 25x faster. Furthermore, with parallelization hardware, Binary TTC is able to rapidly produce geofences to determine a continuous TTC. Expected use cases for Binary TTC include autonomous vehicles, robotics, and more.

2.2 BinaryTTC - Review

Binary TTC appears to be a great tool for autonomous vehicles in detecting collisions. In terms of strengths, the paper discussed comparisons to competitors to emphasize the simplicity and practicality of the Binary TTC method. The paper began with an insight into the different technologies used to predict depth including lidar, stereo, and a monocular camera. The authors argued that to measure depth, lidar was too expensive, stereo required specific calibrations, and that while achievable, it would be challenging on a monocular camera. It was then argued that in the case of a collision in front of the monocular camera, the most important aspect would be the object approaching the camera which could be measured via TTC. Practicality, simplicity, and efficiency are the three main advantages that Binary TTC brings relative to competitors.

However, one of the most prominent issues the paper fails to address is the fact that a vehicle may be struck from anywhere. In the current implementation, the geofence that is computed only protrudes from the front of the monocular camera. By detecting changes in pixel size ratio, I would imagine it to be challenging and require significantly more effort to establish reliable geofencing on the sides of the vehicle. Potentially in the future, the authors may look into stitching camera views together, but this may present issues in accurately detecting a change in pixel size ratio from different viewpoints of an object. Binary TTC is still a very advantageous development in the scope of autonomous vehicles. But in its current implementation, it would need to be paired with other technologies to help overcome its shortcomings with a 360-degree viewpoint.

It appears from the Binary TTC methodology that the lack of training data concerning ground truth TTC was a significant factor influencing design and development. So much so, that another similar method called binary optical flow was modified and implemented as an auxiliary tool in order to pre-train the Binary TTC network. While the binary optical flow auxiliary function also serves function to build strong prior biases to help the network, I wonder if with better training data it would need to even be included. How much of an impact of binary optical flow does the pre-training function have relative to its use to help build stronger biases? In essence, with better ground truth TTC training data, could the binary optical flow function be removed and still achieve similar performance?

3 SinGAN

3.1 SinGAN - Summary

General Adversarial Networks (GANs) have been a large success in modeling visual data. But it is still very challenging and often requires conditioning or specific training in order to model highly diverse datasets. SinGAN is an unconditional model that is able to produce new, variable sample images of an arbitrary size and ratio with the same global structure utilizing only a single training image. SinGAN has been shown to be capable of producing fake images that are of enough quality to fool humans.

SinGAN operates on a patch of an image at a time. Each patch is fed into a pyramid setup of GANs including a generator that outputs a fake patch in an attempt to fool the discriminator. At each stage of the pyramid, the generator is fed random noise to introduce variability into the image patch. As the the image patch ascends the GAN pyramid, the image is continually refined.

In terms of results, SinGAN was able to confuse human test subjects at over a 40% rate in determining if a single image was real or fake at the highest variability scale. With a lower degree of variability, a greater level of confusion followed. Some use cases of SinGAN include editing, super resolution on a single image, integrating a pasted image with a background, short video animations from a single image, and converting a painting to a realistic image.

3.2 SinGAN - Review

SinGAN is a interesting tool with lots of real world application in the realm of photoshop and editing. A strength of the model was the ability to control and scale the variability in the images. It was important for a user to be able to set on a scale how many differences were desired in the output image relative to the input. In each step of the GAN pyramid, this scale will enable a user to control just how much random noise is input to the generator of the fake image. Another strength of SinGAN was the literal elimination of training datasets. Often times, the question of how well a model will work is strongly tied to the quantity and quality of the training dataset. But since SinGAN is able to function on just a single image, this is not a factor.

However, this also brings up a weakness in the implementation. As the model is only trained on a single image, this then limits the amount of feature diversity that can be captured in the generated image. For example, if an image contains a horse, as SinGAN is only trained on a single image, it will not be able to produce an output with a different breed of horse.

One question I had that was not covered is the optimal amount of generator and discriminator pyramid levels for ideal refinement. I would assume that at some point, there is no meaningful difference in refinement between pyramid levels and was wondering what that level would be. However, this question may also depend on the use case as I would think for super-resolution that a greater level of refinement would be ideal relative to say a use case in animating an image.

4 CodeSLAM Implementation

4.1 Goals and Ideas

Of the three papers analyzed, CodeSLAM was the most interesting and relevant to my future career. Outside of class, I conduct work for a club related to robotic vision and navigation. Part of the reason I selected CodeSLAM to further implement was to potentially incorporate the method into the robots I work with. Furthermore, robotics and computer vision systems are two fields of high interest I could see myself pursuing during my career. CodeSLAM has applications tailored to both these fields and in all was a great learning experience.

As this particular paper already had multiple prior implementations in various repositories online, my plan was to implement the completed work and then extend the suggested ideas presented. The primary extension proposed by the authors and which I had hoped to solve was improving from single-image geometry to a real-time approach. Another idea for extending this work was to examine the overlap optimization feature mentioned in the paper. The authors mention that if two of the selected image keyframes overlap, they are optimized to produce better depth estimates. However, this optimization can be computationally expensive. The last idea for extension was investigating the convolution pyramid used to refine depth and uncertainty in the model. CodeSLAM first computes image uncertainty along with five mapping features in a U-Net. These mapping features are then fed into a variational autoencoder. From there, the VAE computes depth. But before uncertainty is returned from the U-Net and depth returned from the VAE, each is input into separate, four-component convolutional pyramids. The authors mention that these pyramids are used to evaluate the error across multiple resolutions. However, similar to the optimization overlap, these pyramids also could be computationally expensive.

4.2 Checkpoint Implementation

At the point of the second checkpoint, I had been able to successfully reproduce CodeSLAM by sourcing code from a few online repositories with some modifications. The major features consist of a data preprocessor, a data loader, the model comprised of a U-Net and VAE, a loss model, an optimizer, a learning rate scheduler, a training function, and some additional tools such as a model checkpoint saver, parser, and plotter. I relied on the online repository to implement the U-Net and VAE networks, loss model, optimizer, scheduler, and training function. I adapted online examples with significant changes to construct the data loading and initial preprocessing largely myself. Regarding the dataset, a very small subsection of SceneNet RGB-D was used for training as recommended by the authors. In terms of results, I had been able to train only on a single epoch with a small fraction of the total iterations recommended due to the limited data at the time and extensive training times. While the model was able to compile and train, at that stage, the model was lacking a testing function for evaluation. Nevertheless, the log outputs showed a clear decrement in the loss value on most iterations, indicating that at least something seemed to be working.

Even with online aid, the initial implementation step was significantly harder than expected. Preprocessing a small subset of the SceneNet RGB-D dataset and configuring the format to run with PyTorch data loaders was a challenge. The tedious process involved writing bash scripts to split, rename, and reorganize the downloaded data. Once processed, the data was then uploaded to Colab which took a painstakingly long time.

The next steps at the time included writing a testing function to better evaluate the model results, processing and uploading more of the SceneNet RGB-D dataset to Colab, and beginning extension from single-image to real-time geometry.

4.3 Final Implementation

At this stage, much has changed since the last checkpoint. I was able to successfully adapt and implement a testing function for the model to evaluate and deliver results. I also built a better bash script that while still very tedious to operate, was able to process the SceneNet data and upload it to Colab. In terms of the model extension, I struggled significantly in converting to a real-time approach and ended up transitioning to analyzing the convolutional pyramid.

The plan all along for my extension was to upgrade CodeSLAM from single-image scene geometry to predict scene geometry in real-time. This would require a significant reduction in the model latency. Quickly after checkpoint two, I found a paper from the same authors as CodeSLAM titled, *CodeMapping: Real-time dense mapping for sparse slam using compact scene representations*(Matsuki et al., 2021).The paper built on the ideas of CodeSLAM and outlined a method for real-time dense mapping operating in conjunction with spare SLAM systems. CodeSLAM was a dense mapping framework built for eventual use with real-time mapping and localization in the form of a SLAM algorithm. However, the authors argue that dense mapping and localization simultaneously are not as successful as spare methods because dense mappings contain a large number of scene parameters making real-time, joint optimization difficult. Thus CodeMap proposes a real-time dense mapping method that works in tandem with an existing sparse SLAM system. The actual network associated with CodeMap is a U-Net and VAE very similar to CodeSLAM that is conditioned on image intensity, sparse depth, and new input in reprojection error. The SLAM system and network operate in parallel to produce a real-time depth mapping of the scene. The basic operation is displayed in Figure 1. Input data images are fed into the SLAM thread that uses the ORB-SLAM3 software to perform sparse tracking and mapping. From there, the SLAM system feeds a sparse depth map, camera poses, and the reprojection error of a keyframe onto its neighbor into the dense mapping thread. The dense mapping then leverages this input data with a U-Net and VAE to build a depth prediction for the scene in real time. The dense mapping thread essentially is a slight upgrade over the CodeSLAM network that works in parallel with the SLAM system. The paper was almost exactly what I was looking for to extend CodeSLAM, but I quickly ran into issues. I began setting up ORB-SLAM3 following the documentation listed on the GitHub page but was having problems installing the necessary software and building locally, much less on Colab. Another issue encountered was that while the paper provided a description of reprojection error, I struggled to wrap my head around it much less implement it in code. The final straw was when the paper mentioned ORB-SLAM3 ran in C++ and required a C++ API to convert the U-Net, VAE python model. All these issues paired with the lack of any online code base factored into my decision to change extension approaches.

While I did examine a few other similar papers for converting CodeSLAM to real-time operation, I eventually decided on analyzing the convolutional pyramid feature present in CodeSLAM. As mentioned above, after exiting the U-Net and VAE, uncertainty and depth respectively are fed into a four-network convolutional pyramid that finds uncertainty and depth at four different pixel resolutions before entering the loss function. While computing these features at multiple resolutions is nice, it likely does not help reduce the latency needed for an eventual real-time approach. I was curious about the impacts on performance and timing and decided to further investigate.

I first made a copy of the current CodeSLAM model and made a few changes to test upon. I changed the input image channel to three instead of one. While I did find this change online in a few repositories, I made the change as I thought that training on RGB images instead of grayscale could potentially improve depth and uncertainty accuracy. This required reconfiguring a few of the U-Net and VAE blocks in order to account for the layer change. The next significant change made was the removal of the convolu-

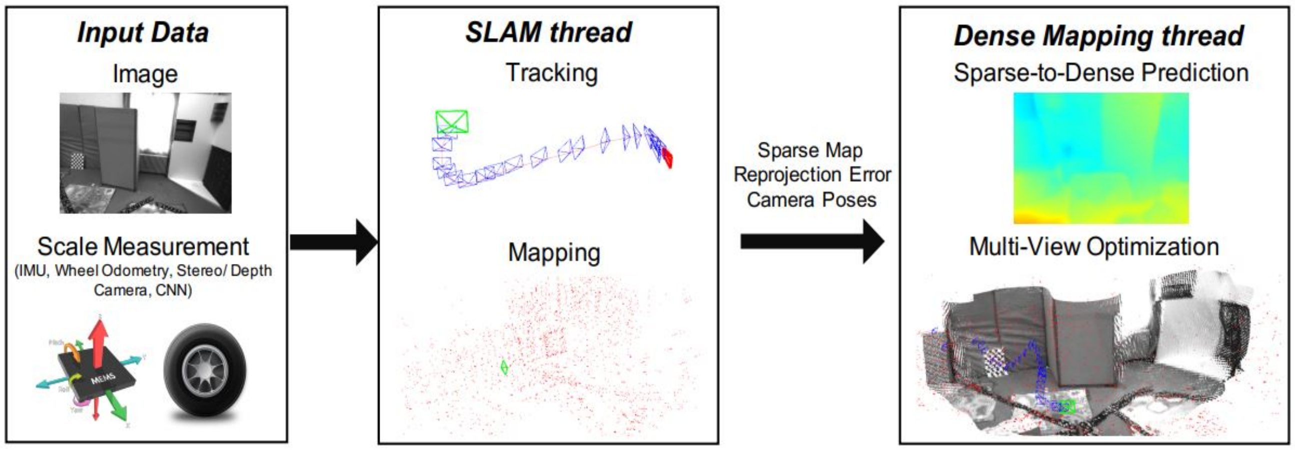

Figure 1: CodeMap system architecture: SLAM thread and Dense Mapping thread operate in parallel.

tional pyramids. This also required some reshuffling in the U-Net and VAE functions. It also allowed the new model to calculate reconstruction loss in one go instead of an iterative sum accounting for multiple uncertainties and depth outputs for the different resolutions. With changes in place, code compiling cleanly, and the evaluation functions upgraded for the new model, I was ready to test.

4.4 Evaluation and Results

Evaluation metrics include the root mean square error (RMSE) between prediction and ground truth, the logarithm of prediction and logarithm of ground-truth RMSE, evaluation time, and visual inspection.

Two comparisons were run on the different CodeSLAM models with varying epochs and iterations. The first comparison was mainly a proof of concept and thus ran on only one epoch with 100 iterations on a batch size of 16. Both models were trained with 3000 depth and photo images from SceneNet RGB-D. For testing, each model was evaluated against 1500 different depth and photo images from the same dataset. Results are displayed in Table 1, a sample output of the original model in Figure 2, and a sample output of the new model in Figure 3.

	RMSE	LogRMSE	Time (s)
Original	1.4767	6.8737	52.262
New	**0.2282**	**0.4221**	**49.951**

Table 1: Results of evaluation one.

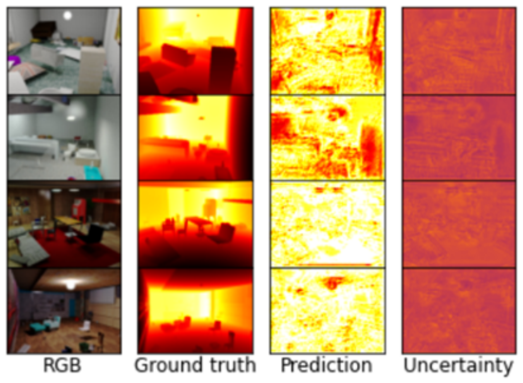

Figure 2: Original CodeSLAM model after evaluation one.

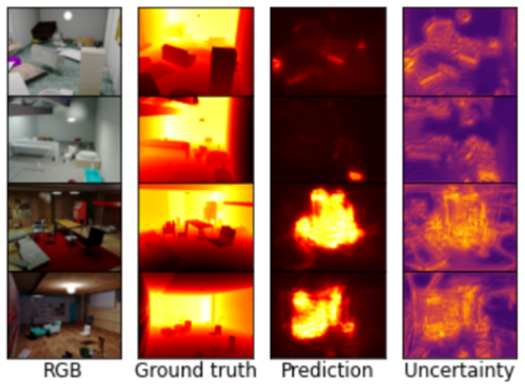

Figure 3: New CodeSLAM model after evaluation one.

On the second comparison, models were run on four epochs on a batch size of 16. Both models were trained on 6000 depth and photo images and tested on 1500 different data points, again from SceneNet RGB-D. For this second comparison, evaluation was performed at regular intervals in training in addition to on the complete model after training. Results are displayed in Table 2, a sample output of the original model in Figure 4, and a sample output of the new model in Figure 5.

	RMSE	LogRMSE	Time (s)
Original	1.2599	4.8878	36.094
New	**0.2177**	**0.2737**	**34.097**

Table 2: Results of evaluation two.

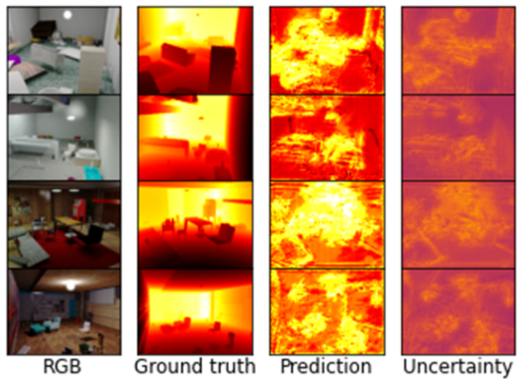

Figure 4: Original CodeSLAM model after evaluation two.

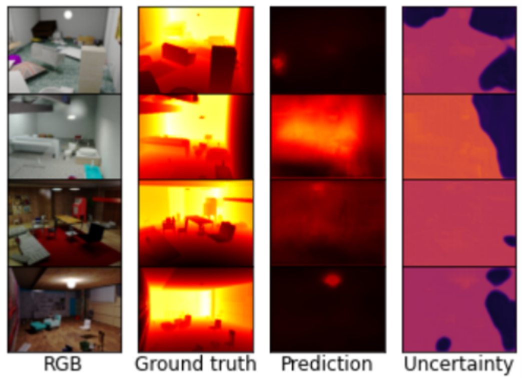

Figure 5: New CodeSLAM model after evaluation two.

4.5 Discussion

As seen in the result tables, the new CodeSLAM implementation beats out the original implementation in every metric measured. The new model is able to predict the scene depth much closer to the ground truth than the original. Furthermore, the new model is slightly faster than the original model. This makes sense as the new model has four fewer convolutional steps and reconstruction loss iterations to go through than the original with the depth and uncertainty pyramid.

Regarding the visual inspection, both models don't seem to perform very well. In terms of the original CodeSLAM model, both evaluation outputs predict a significantly brighter image than the new CodeSLAM model. This could be due to the fact that the original CodeSLAM model gives input images only one channel while the new model allocates three. This allows the original model to make a better prediction on inputs closer to grayscale than RGB. Such a case occurs with the top two image inputs in which the original model seems to produce a relatively close output to the ground truth. The new model seems to do reasonably on inputs that are more grayscale and RGB.

Even though the new model performs better in all metrics, one potential downside if applied to a real-world scenario is the case of different input resolutions. As all the training and testing images are of the same resolution coming from the same dataset, it makes sense that removing the convolutional pyramid that calculates for different resolutions would improve performance. However, if a vehicle is processing real-time data, the input images to the network may not be all the same resolution which could potentially negatively impact the model performance.

References

Badki, A., Gallo, O., Kautz, J., and Sen, P. Binary ttc: A temporal geofence for autonomous navigation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12946–12955, 2021. URL https://arxiv.org/pdf/2101.04777.pdf.

Bloesch, M., Czarnowski, J., Clark, R., Leutenegger, S., and Davison, A. J. Codeslam—learning a compact, optimisable representation for dense visual slam. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2560–2568, 2018. URL https://arxiv.org/pdf/1804.00874.pdf.

Matsuki, H., Scona, R., Czarnowski, J., and Davison, A. J. Codemapping: Real-time dense mapping for sparse slam using compact scene representations. *IEEE Robotics and Automation Letters*, 6(4):7105–7112, 2021. URL https://arxiv.org/pdf/2107.08994.pdf.

Shaham, T. R., Dekel, T., and Michaeli, T. Singan: Learning a generative model from a single natural image. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 4570–4580, 2019. URL https://arxiv.org/pdf/1905.01164.pdf.