Generative Sampling of real-world traffic scenes on the Tegra X1

Menna El-Shaer, Fusun Ozguner, Keith Redmill
The Ohio State University

Abstract

Urban traffic scenes present a challenge to vision perception systems due to the dynamic interactions among participants whether they are pedestrians, bicyclists, or other vehicles. Object detection deep learning models have proved successful in classifying or identifying objects on the road, but do not allow for the probabilistic reasoning and learning that traffic situations require. Deep Generative Models that learn the data distribution of training sets are capable of generating samples from the trained model that better represent sensory data, which leads to better feature representations and eventually better perception systems. Learning such models is computationally intensive so we decide to utilize Graphics Processing chips designed for vision processing. In this paper, we present a small image dataset collected from different types of busy intersections on The Ohio State University campus along with our CUDA implementations of training a Restricted Boltzmann Machine on NVIDIA GTX1080 GPU, and its generative sampling inference on an NVIDIA Tegra X1 SoC module. We demonstrate the sampling capability of a simple RBM network trained on a subset of the dataset, along with profiling results from experiments done on the Jetson TX1 platform.
Keywords: Automated Driving; Intelligent Vehicles; Vision Perception; Real-time Scene Understanding; Probabilistic Graphical Models; Deep Learning; NVIDIA Graphics Processing Units

1. Introduction

The recent success of deep learning in the autonomous vehicle industry is largely attributed to how well those systems can classify objects in the traffic environment. Usually the most studied network architectures e.g. Convolutional Neural Networks (CNNs) are discriminative learning models where a vast amount of human-annotated i.e. labeled data is needed for training and learning features in the data. The system then learns from the labeled examples and can detect or classify those objects on its own. While this is important in solving a subset of situational awareness problems of self-driving cars, identifying the class of an object in the environment, e.g. classifying the object that runs in front of the car as a small child or a small animal, is irrelevant as the car is supposed to react in the same way. Thus, training these deep networks in the supervised manner used is not very useful in challenging urban traffic environments that need reasoning built into the learning process. A class of deep networks that are better suited for reasoning tasks are generative learning models that don't require labeled examples and generally learn the joint distribution

of the training data. Thus, by training these networks with different traffic situations, they learn a traffic model with specific parameters related to the traffic situation e.g. a moving object is in front of the car, which is then used to drive the vehicle to take the appropriate action e.g. reduce speed to avoid collision.

Two points need to be noted here: one is that the traditional algorithms used for training these deep networks are very computationally intensive, if at all tractable; second is that the structure of these deep networks maps very easily to parallel computing architectures that use multiple units for processing at a time. This helps greatly to make such algorithms transferrable to practical situations and is the main reason behind such advances in the field of intelligent vehicles. AI supercomputers are now ubiquitously available e.g. the NVIDIA DRIVE platform, and their use is helping in accelerating research and development in the field of real-time scene understanding.

In this article, we will present an experiment done at The Ohio State University where we use the integral part of the NVIDIA Drive PX systems: the NVIDIA Tegra X1 SoC, with a suite of GigE Flea3 Point Grey cameras that are aligned at different angles to capture full views of different types of traffic intersections e.g. four-way stop, stop light or a round-about, to do the real-time inference of the trained generative model. Training is done offline on the NVIDIA Pascal architecture GTX 1080 GPU. We will utilize the CUDA programming framework to do both training and inference of the model. Several optimization strategies, especially for the Tegra X1 system where real-time performance is critical will be discussed, along with profiling program analysis that will help identify performance bottlenecks.

2. Background

We start by giving a brief review of two seemingly unconnected topics in sections 2.1 and 2.2 that are fundamental to understand our methods and implementations that follow.

2.1. Generative Modeling

Even though generative models have been researched for many years, their application in autonomous vehicle research has not been well studied or documented (except for their classification abilities in [1], which just scratches the surface). What follows is a review of the energy-based generative model we used. Inspired by statistical mechanics, the Restricted Boltzmann Machine, a probabilistic graphical model's energy with bipartite connections between its stochastic variables $v \in [0,1]^V$ and $h \in \{0,1\}^H$ shown in figure 1 [2], [3], [4] is defined with model parameters θ as:

$$E(v,h) = -bv - ch - h^T W v \,;\, \theta \,\epsilon\, \{b, c, W\} \tag{1}$$

Interactions between nodes are represented by the weight matrix W, while b and c are bias terms for visible nodes v and hidden nodes h respectively.

The joint probability distribution between the variables is modeled as:

$$p(v,h) = \frac{e^{-E(v,h)}}{Z} \tag{2}$$

where Z is the partition function or probability normalizing constant.

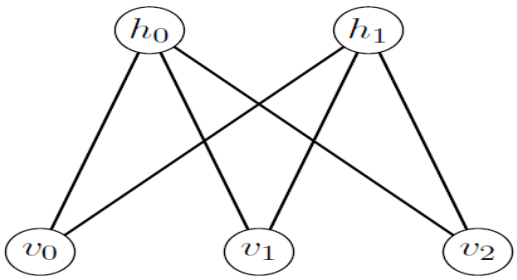


Figure 1: A graphical representation of a Restricted Boltzmann Machine

Given input variables $v \in [0,1]^V$, model parameters θ can be learned by maximizing the log-likelihood $F(v)$ as shown in equations 3 and 4.

$$F(v) = -log(p(v,h)) = -log(\sum_h e^{-E(v,h)}) \tag{3}$$

$$-\frac{\partial log(p(v))}{\partial \theta} = \frac{\partial F(v)}{\partial \theta} - \sum_v p(v)\frac{\partial F(v)}{\partial \theta} \tag{4}$$

The second negative term in equation 4 which is the expectation over all possible input data configurations is intractable in practice, to make computations feasible we will approximate the likelihood by fixing the number of model samples to a finite N as shown in equation 5 and use repeated sampling i.e. Markov Chain Monte Carlo (MCMC) which results in N samples drawn from a distribution similar to the marginal distribution $p(v)$.

$$-\frac{\partial log(p(v))}{\partial \theta} \approx \frac{\partial F(v)}{\partial \theta} - \frac{1}{N}\sum_{v \epsilon N} \frac{\partial F(v)}{\partial \theta} \tag{5}$$

The bipartite structure of the RBM allows for repeated alternating Block Gibbs Sampling [2], [3], [4] between the two layers as shown in equations 6.

$$\begin{aligned} p(h_j = 1 \,|\, v) &= g(c_j + \sum_i v_i w_{ij}) \\ p(v_i = 1 \,|\, h) &= g(b_i + \sum_j h_j w_{ij}) \end{aligned} \tag{6}$$

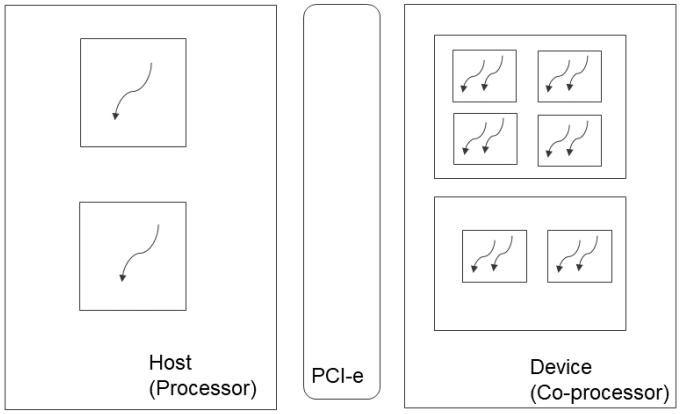


Figure 2: The Heterogeneous Computing Model

Such approximation defined by Hinton in [5], known as constrastive divergence (equation 7) has been shown to work well with $N = 1$ [5], [2].

$$\triangle w_{ij} = \lambda \,.\, (E(v_i, h_j)_{\text{data}} - E(v_i, h_j)_{\text{reconstruction}}) \tag{7}$$

where λ is the learning rate of the stochastic gradient descent of equation 5

2.2. Heterogeneous Processing Model

GPUs are an example of heterogeneous processors, where two separate different architecture processors are connected via a PCIe bus. Each system has its own DRAM and resources. The GPU is comprised of a scalable array of *Streaming Multiprocessors* or SMs. Parallelism is achieved using NVIDIA's Single Instruction Multiple Thread (SIMT) model, where multiple threads are execute the same instruction in a group called *warp* [6], [7], [8]. The SMs are responsible for processing thread blocks of the kernel grid launched by the CPU co-processor which the programmer gets to control. Figure 2 shows the heterogeneous programming structure [8].

2.3. The KITTI Dataset

We decided to collect and use our own dataset to study the real-time image processing capability of the Jetson TX1 platform connected with multiple machine vision cameras. We note that the KITTI dataset [**?**

], [**?**], [**?**], [**?**] is another potential source of urban traffic scene images that could be used for training, testing and benchmarking various algorithms.

3. Materials and Methods

Two sets of images were used in this experimental section. The first dataset was collected to test the experimental setup, while the second dataset was collected to model and reconstruct traffic scene intersections. Algorithm implementation and testing were done on both datasets, and are discussed in sections 3.1 and 3.2 respectively.

3.1. Dataset 1: Bike Path Images

3.1.1. System Setup

Figure 3 shows the experimental setup to collect image data using an electric golf cart as the moving vehicle. Images were captured using one StereoLabs ZED camera [9] mounted on the bottom right-side of the windshield connected using USB 3.0 to an NVIDIA Jetson TX1 development board [10]. Images were recorded using ZED SDK 1.0.

Figure 3: Experimental Setup used to collect real-world images along a gravel bike path using an electric golf cart

3.1.2. Data Collection

We drove the golf cart along the gravel bike path and captured images of walking, running and jogging pedestrians along with some on bicycles. A collection of seven sample videos in .svo format were recorded. Example images from a sample video are shown in figure 4.

3.2. Dataset 2: Urban Traffic Intersections

3.2.1. System Setup

Images were captured using two Point Grey GigE Flea3 cameras [11] mounted on the car's dashboard angled 45° to the left and right, connected using Intel Network GigE PCIe Adapter 82576 to an NVIDIA Jetson TX1 development board [10]. Image Acquisition was done synchronously using both cameras.

Figure 4: Sample images recorded from an .svo video file using ZED SDK 1.0

3.2.2. Data Collection

Images of real-life traffic intersections were collected while driving around campus. We focused on including images of the following five types of intersections in the dataset:

- Four-way stop sign controlled
- Traffic light signal controlled
- Cross-walks without stop signs
- Roundabout-type intersections
- Three-way intersections

Total number of images in the dataset were 5859 images from each camera. We divided the dataset into independent training and test sets in the typically used ratio of 5:1.

3.3. Model Training

All image frames in an .svo video were used to train a Restricted Boltzmann Machine as an example of a probabilistic one-layer unsupervised network. Images were cropped from 720×1280 to 360×960 to remove redundant pixels e.g. sky regions. 2D Image data used were normalized grayscale values stored as *double* data type of 8 bytes. This was done to best match the RBM model that uses probability values for its input visible layer. One container for all training data was created indexed by the frame number as the container's row pointer; this speeded up the copying process i.e. an image was copied as in blocks of its rows.

Before training the network, a set of initial weights and biases had to be computed.Theoretically, any random weights and biases will do but since usually there are thousands of visible and hidden neurons, any hints on where to start the search in the weight space could lead to faster convergence in the Markov chain. The procedure used for initializing weights was similar to the training procedure: all training data was divided into batches, and each complete round constituted an epoch. We followed the procedure in [12]: initial weights were allowed to vary using the harmonic mean of the weight matrix dimensions; this was done using the fact that summing the dot product of neuron activations and a weight vector depends on the length of that vector i.e. its dimension. Weights were initialized using the following formula:

$$W_{ij}^{Initial} = \frac{4 * rand(0,1)}{\sqrt{\sqrt{N_V * N_H}}} * (rand(0,1) - 0.5) \tag{8}$$

. To compensate for a potential unbalance in the random weight set, we add an initial bias $a_j = -\sum_i \bar{x_i} W_{ji}$; where $\bar{x_i}$ is the mean of the input dataset, to each hidden neuron such that for the average training set, its net input is zero i.e. distribution is centered. To ensure a somewhat small reconstruction error, setting an

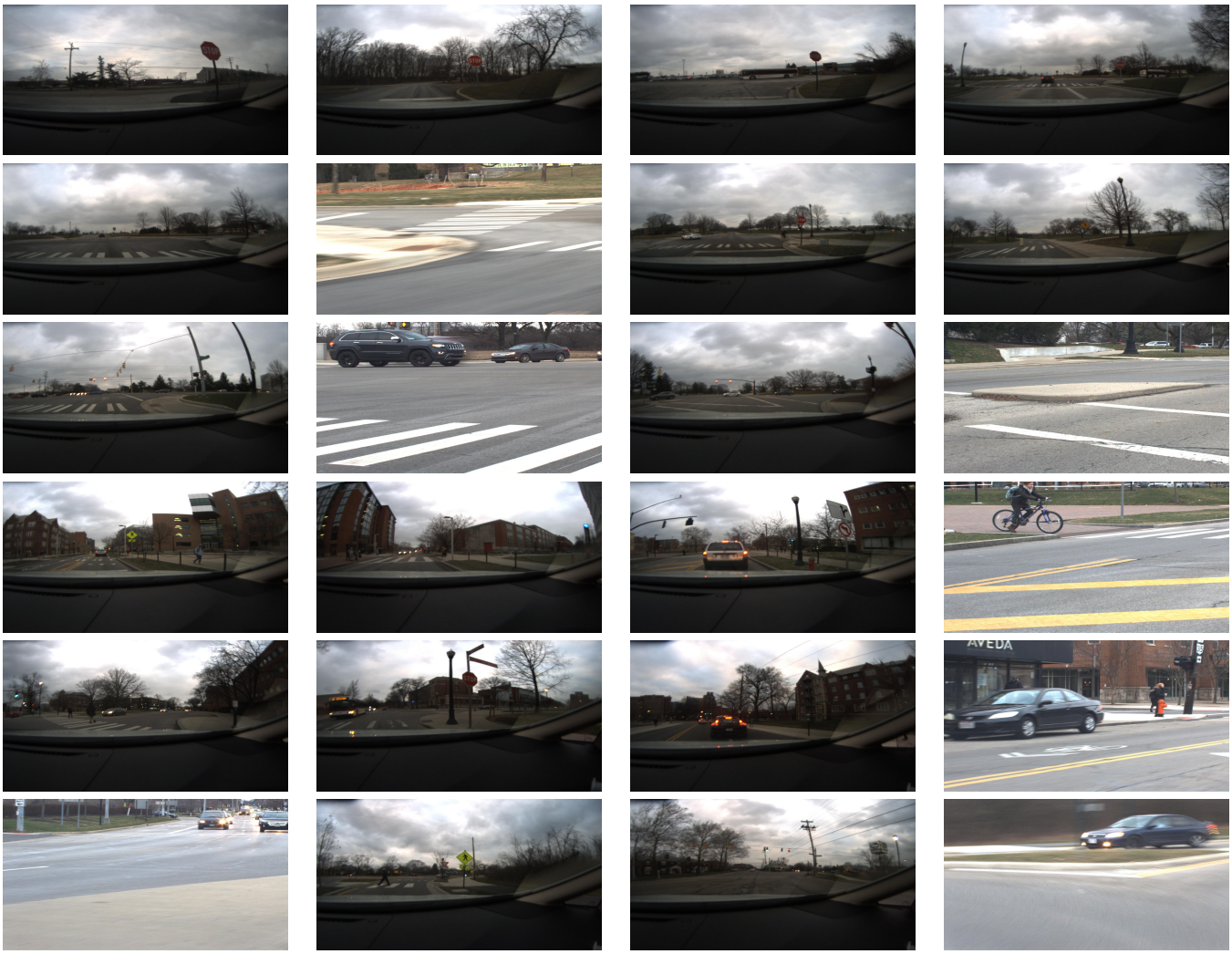


Figure 5: Sample images recorded from the collected dataset of traffic intersections

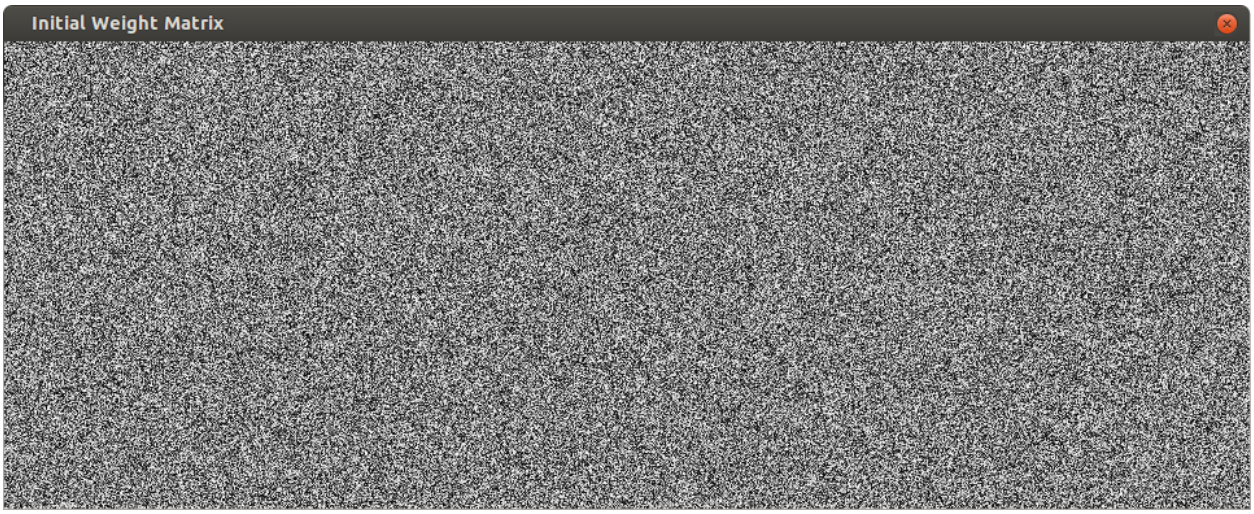


Figure 6: Initial weight matrix displayed as 360 x 960 image. This weight matrix is for the first hidden neuron in the RBM layer of 200 hidden neurons

initial bias b_i for the visible neuron activation that depends on the average activation of the hidden layer q was done.

$$c_j = log(\frac{q}{1-q}) - a_j \tag{9}$$

$$b_i = log(\frac{\bar{x_i}}{1-\bar{x_i}}) - q\sum_j W_{ji} \tag{10}$$

Figure 6 displays the weight matrix of the first hidden neuron as an image.

Alternatively, one could follow the recipe described in [13], and use small random values chosen from a distribution $N(0, 0.01)$, zero hidden biases, and bias for visible unit i as $log[p_i/(1-p_i)]$; where p_i is the proportion of training samples that turn unit i on.

Training was done in batches on an NVIDIA GTX-1080 [14] which is based on the Pascal GPU architecture. Since the card follows the typical heterogeneous memory architecture, data has to be copied into device memory space before processing. Training hyperparameters were declared in *constant memory* since they are *read-only*, while pointers to dynamic device memory were used for data and weight arrays. At the beginning of each training epoch, data was shuffled using a standard Fisher-Yates shuffle [15] and indices were copied to the device. There are usually some serial correlations in the data so shuffling helps to vary the contents of batches throughout training epochs.

A training epoch thus becomes a loop through all batches of the following:

1. Start batch (loop)

    1.1. Get visible unit values from the data

1.2. Compute the hidden probabilities without sampling from the visible layer values

1.3. Start Markov chain

1.3.1. Sample hidden activation values

1.3.2. Compute the visible layer values from the hidden activation values

1.3.3. Compute the reconstruction error for the current chain

1.3.4. Use the latest visible layer probability values to get the hidden layer probabilities

1.4. Update Parameters

1.4.1. Update the visible bias vector

1.4.2. Update the hidden bias vector

1.4.3. Update the weight matrix

1.5. Compute weight gradients (and gradient differences for adjusting the learning rate heuristically – if desired)

2. Next batch (loop)

3. Test for convergence by measuring the largest weight gradient computed during the current epoch relative to the largest weight magnitude

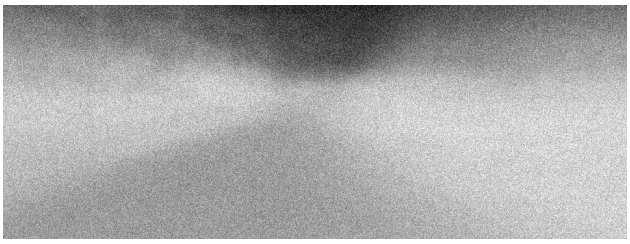
4. Adjust momentum and learning values for the next epoch – if needed to prevent large deviations when near convergence

Each training step was implemented on the device as a CUDA kernel. Columns of working vectors were mapped to grid blocks along x-dimension, while rows were mapped to the y-dimension. Each element in the working vector was assigned to a thread, launching the kernel with a number of threads per block that is a multiple of the warpsize i.e. 32 threads, to ensure coalesced memory reads/writes and general concurrent instruction execution. Parallel reduction using shared memory was used to compute the weight gradients and weight gradient differences using partial sums, and the maximum weight magnitude using partial maximums.

3.4. Inference

To test the trained model on new image samples, we performed a round of Gibbs sampling. Test images were provided from another recorded .svo video from the first dataset, pre-processed in the same manner as the training images, and fed to the trained network in a random order, to assess the generative ability of the training by generating feature samples.

Inference was done on the Tegra X1 SoC [10], which is based on the older Maxwell GPU architecture. In contrast to how we used the system memory for training; no explicit data copies were done on the TX1. Network hyperparameters were defined as static variables in managed device memory space; working vectors



(a) Weight matrix relating h_0, the first hidden neuron to the input. A total of 5 hidden neurons were used in this network

(b) Weight matrix relating h_0, the first hidden neuron to the input. A total of 200 hidden neurons were used in this network

Figure 7: Computed weight matrices computed after training using the bike path dataset

and data were dynamically allocated on the TX1's unified memory. A one-step Gibbs sampling is the process of first sampling the hidden layer activations using the input visible neurons followed by sampling the visible layer from the computed hidden values. As stated in multiple previous work [2], [4], [16] we found the one-step constrastive divergence i.e. one-step sampling, to be sufficient.

Gibbs sampling was done on the device using two CUDA kernels, one for each sampling phase. Thread mapping was done in a similar case to our training implementation. 2D image data were treated as 32-bit floats (FP32).

The same inference procedure was done for the second dataset except that images from both cameras were used, which enriched the data, but meant that a sampling step could contain two images captured simultaneously.

4. Results and Discussion

4.1. Training

A total number of 2110 images from the first dataset were used in training, while 4883 combined images from both cameras were used from the second dataset. Each dataset was trained separately. Final weights and biases were copied back to the host and displayed. Figures 7 and 8 shows sample weight matrices from trained networks on both datasets. Network weights can be interpreted as filter coefficients that influence the activation of hidden units. Please refer to the appendix for hyperparameter values used while training.

4.2. Inference

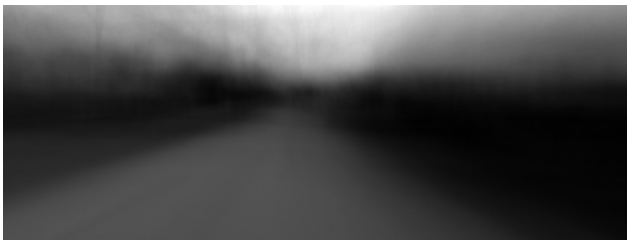
Samples from both datasets are shown in figure 9 respectively. No data copying between processing systems was needed here to display sample results, since both the CPU and GPU share the memory space. Total number of test images in both datasets were 817 and 976 images respectively.

Generative Capability:. Images in figure 9 represent the hidden features found after training the model with 20 hidden units, which is equivalent to representing the data in a 20 hidden (latent) dimensions. Grayscale

(a) Weight matrix relating h_0, the first hidden neuron to the input. A total of 20 hidden neurons were used in this network

(b) Weight matrix relating h_1, the second hidden neuron to the input. A total of 20 hidden neurons were used in this network

Figure 8: Computed weight matrices computed after training using the urban intersections dataset



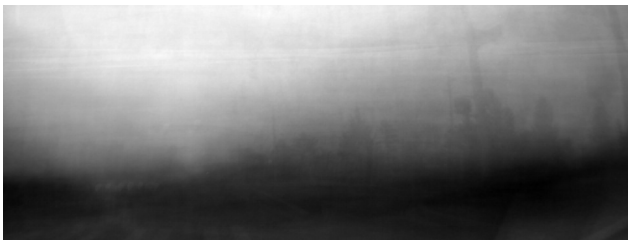


Figure 9: Generated Samples from the trained model in section 3.3

images (0 to 255) are mapped from the 0 to 1 probability space of the RBM. Visible neurons in the RBM represent discrete input pixel values of images while hidden neurons represent the lower dimensional feature space learned by the model. The goal of generative models is to find the representation space that best describes real-world image features, so the hypothesis is the more hidden neurons used in the implementation, the better image representation we get, which is what we showed. It is our future goal to find an alternative memory implementation of the network parameters, instead of traditional dynamically allocating them on the Tegra's unified memory, since the CUDA API function we used *cudaMallocManaged()* can only allocate up to half the size of the physical RAM, so a implementing a swap memory strategy is our next future research step.

Image Perception Evaluation:. To test the generative capability of the model, we used the Perception-based Image QUality Evaluator (PIQUE) metric defined in [17]. PIQUE calculates the quality score of an image using a block-wise distortion map of local features that are extracted from perceptually significant spatial regions. Figure 10 illustrates the algorithm steps.

Our average perception scores were 28.3107% and 42.9174% for each dataset respectively. We measured the mean score for 20 generative samples from each dataset independently.

4.3. Profiling on the TX1

Figure 11 shows our implementation of gibbs sampling as CUDA kernels on the TX1. We expect our *vis_to_hid* kernel to be more computationally expensive than the *hid_to_vis* kernel, since we there are more

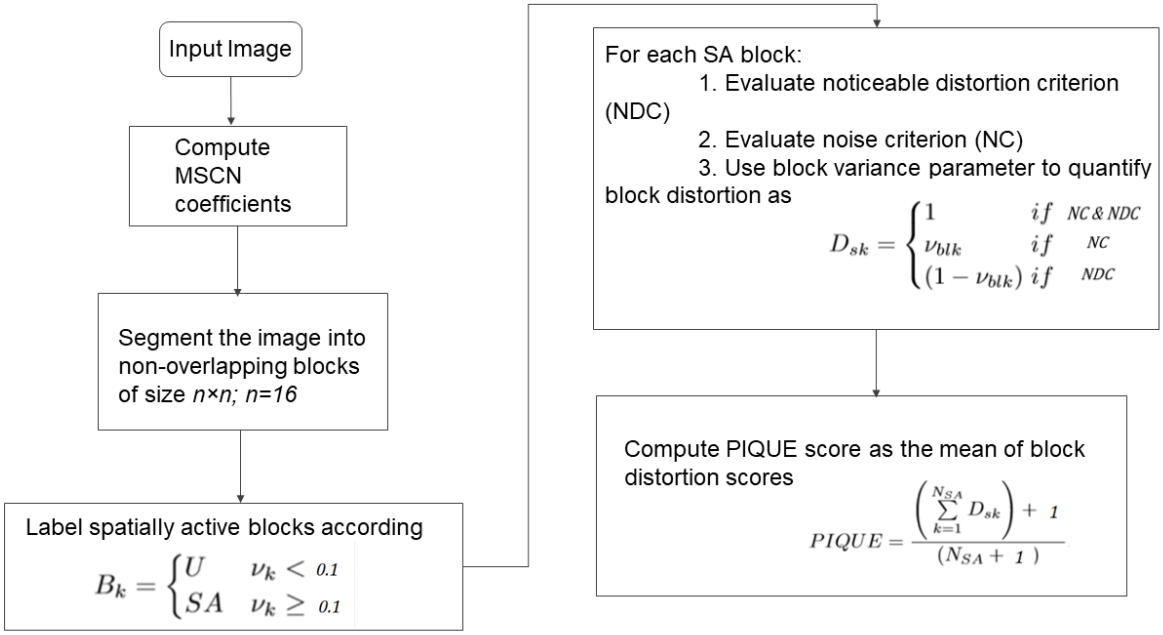


Figure 10: Algorithm steps used to compute PIQUE perception scores of images

visible than hidden neurons. We measured *vis_to_hid* kernel runtimes with four different block sizes of 32, 64, 128 and 1024 threads, and plotted them in figure 12. Figure 13 shows a profiler plot of the number of warps issued per SM on the Maxwell GPU with different block sizes. Figures 14 and 15 show the utilization of each function unit, and the percentage of execution of each instruction type respectively. We follow with a discussion of the optimization strategies utilized in our implementation.

(We assume block size of 1024 threads from here on out)

Global Memory Bandwidth Calculations:. We calculated the effective memory bandwidth used by the *vis_to_hid* kernel by calculating the number of bytes read and written by the kernel over its run time. A memory bandwidth of 2.35GB/s was achieved.

We utilized the unified memory model in our inference implementation. This avoided unnecessary data copies between host and device, in addition to utilizing the L2- cache, in contrast with zero-copy implementations where caching behavior is disabled [18]. Row-major order global memory reads and writes were coalesced, which ensured no unnecessary memory transactions were issued [8], [19], [20]. Achieved occupancy was 23.4% versus 31.2% for the 32 thread block kernel.

Active warps:. Warp execution efficiency i.e. the average percentage of active threads in the executed warps for the *vis_to_hid* kernel was 62.5% due to the presence of intra-warp divergence in the if-branching statements.

```
__global__ void vis_to_hid_kernel()
{
   int ihid = blockIdx.x * blockDim.x +
threadIdx.x;
   int ichain = blockIdx.y;

   float sum = hid_bias[ihid];
   int randnum;

   for (int ivis = 0; ivis < n_vis; ivis++)
       sum += wtr[ivis*n_hid+ihid] *

       vis_layer[ichain*n_vis+ivis];

   float act_Q = 1.0f / (1.0f + __expf(-
sum));
   hid_layer[ichain*n_hid+ihid] = (frand <
act_Q) ? 1.0 : 0.0;
}
```

```
__global__ void hid_to_vis_kernel()
{
      int ivis = blockIdx.x *
blockDim.x + threadIdx.x;
      int ichain = blockIdx.y;
      float sum = vis_bias[ivis];
      for (int ihid = 0; ihid < n_hid;
ihid++){
            sum += w[ihid*n_vis+ivis]
* hid_layer[ichain*n_hid+ihid];

      vis_layer[ichain*n_vis+ivis] =
1.0 / (1.0 + __expf(-sum));
}
```

Figure 11: Implementation of Gibbs sampling kernels

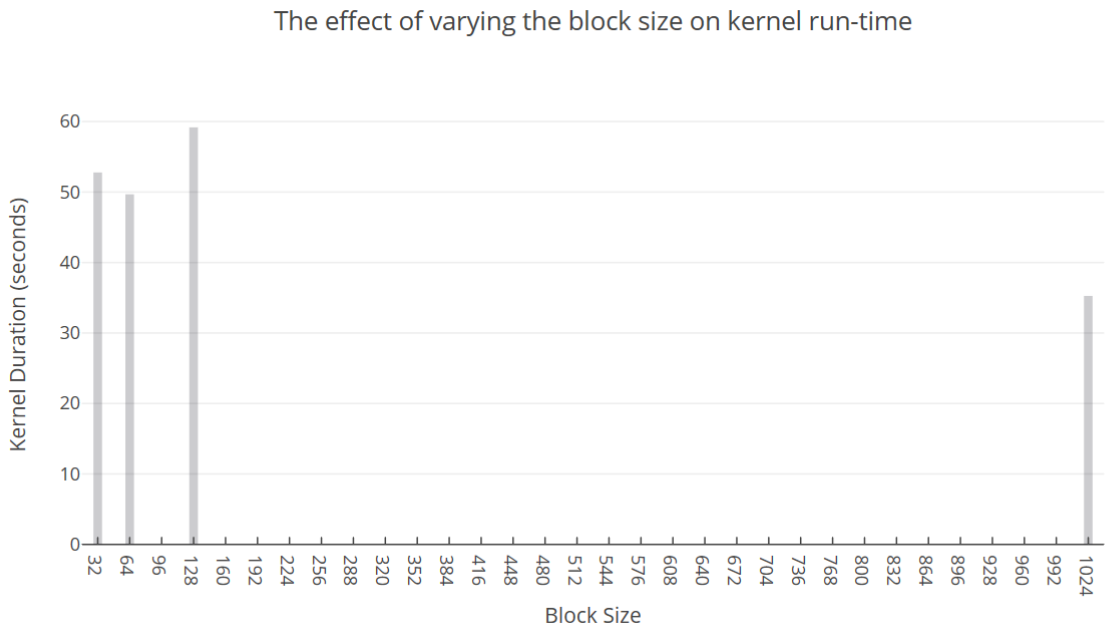


Figure 12: A plot of vis_to_hid kernel duration times ran with block sizes of 32, 64, 128 and 1024 threads. Running the kernel with maximum number of threads produced the shortest run time

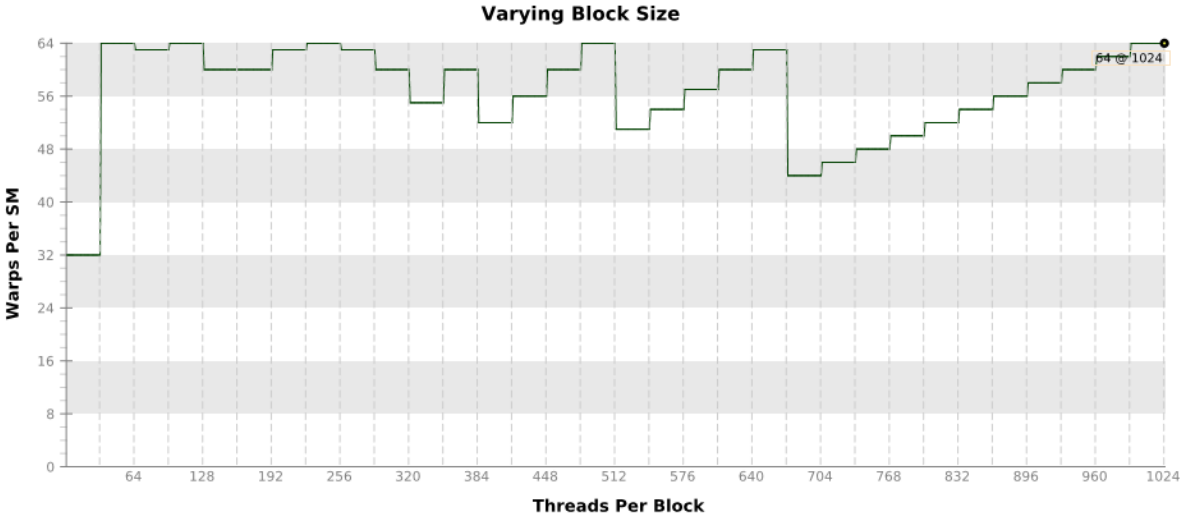


Figure 13: How the number of warps vary per Maxwell SM with different block sizes. There are 2 SMs on the Maxwell GPU architecture. This plot can be used to determine the scheduled number of warps each SM gets for different block sizes or numbers of threads present in a block

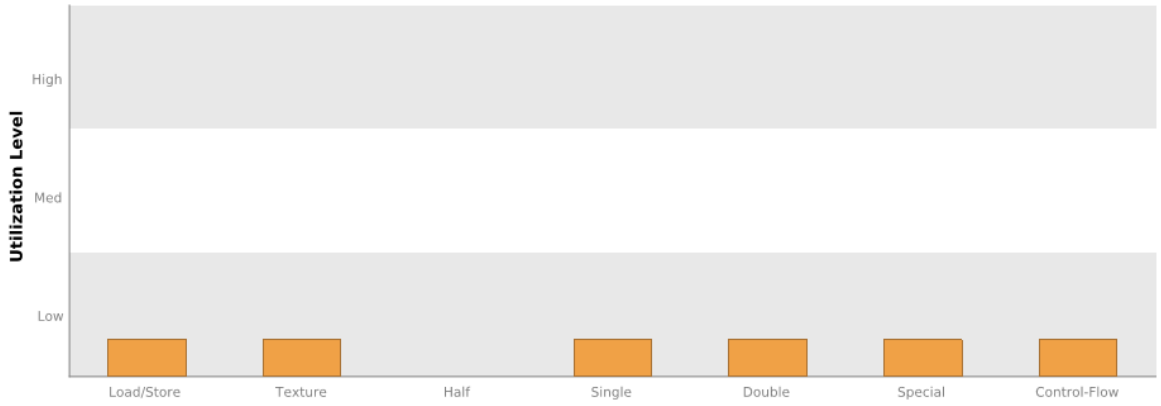


Figure 14: Function Unit utilization of the vis_to_hid kernel. This plot displays how the kernel utilized an SM's function units. Different implementations will result in different utilizations, thus varying the performance of the kernel

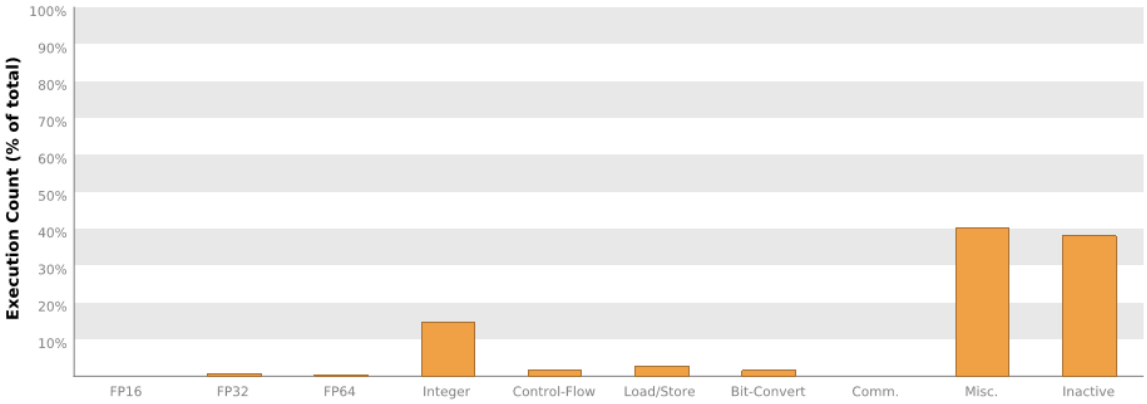


Figure 15: Types of instructions and their percentage of execution in the vis_to_hid kernel. Instructions are grouped into classes and the "inactive" class shows the amount of cycles a thread did not execute any instruction due to divergence or predication

5. Conclusion

This research addresses the problem of real-time scene understanding and environment perception in the context of traffic scenes for different Advanced Driving Assistance systems (ADAS) and automated driving applications. It concerns finding suitable representations for image data by using probabilistic generative methods to model the hidden or latent variables in the data. The claim here is that if we can find the optimal space representation, classification (e.g. labeling vehicles, pedestrians and other objects in the traffic scene), regression tasks and inference become easier and more accurate. Traffic image data from camera sensors have proven to be complex and thus require multiple stages for best feature extraction. This, in addition to the amount of available sensory data become bottlenecks for the real-time processing requirement of traffic applications.

The availability of parallel computing architectures e.g. GPUs has helped improve the work complexity of these feature learning algorithms and contributed to their applications in automotive safety products. The availability of real-world traffic scenes datasets is crucial for developing and testing those learning algorithms. To summarize, our contributions can be listed as follows:

1. Collection of real-world images of traffic intersections and a dataset that could always be expanded in the future.

2. Implementing and training a probabilistic generative network on GPUs, utilizing and studying its architecture for best computational processing power.

3. Implementing an inference model on embedded SoCs and studying performance optimization strategies.

Acknowledgement

This work was partially funded by NSF grant number CNS-1446735.

References

[1] R. Hadsell, A. Erkan, P. Sermanet, M. Scoffier, U. Muller, Y. Lecun, Deep belief net learning in a long-range vision system for autonomous off-road driving, 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems`doi:10.1109/iros.2008.4651217`.

[2] R. R. Salakhutdinov, Learning deep generative models, Annual Review of Statistics and its Application 2 (2015) 361–385.

[3] Y. Bengio, A. Courville, P. Vincent, Representation learning: A review and new perspectives, IEEE Trans. Pattern Anal. Mach. Intell. 35 (8) (2013) 1798–1828. `doi:10.1109/TPAMI.2013.50`.

[4] Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle, Greedy layer-wise training of deep networks, in: Proceedings of the 19th International Conference on Neural Information Processing Systems, NIPS'06, MIT Press, Cambridge, MA, USA, 2006, pp. 153–160.

[5] G. E. Hinton, S. Osindero, A fast learning algorithm for deep belief nets, Neural Computation 18.

[6] J. Sanders, E. Kandrot, CUDA by example : An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010.

[7] D. B. Kirk, W. W. Hwu, Programming Massively Parallel Processors: A Hands On Approach, 2nd Edition, Elsevier Science and Technology, San Francisco, CA, 2012.

[8] CUDA C programming guide, Available at `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html` (2018/04/16).

[9] Stereolabs ZED camera.
URL `{https://www.stereolabs.com/zed}(6/7/2018)`

[10] NVIDIA Jetson TX1 development platform.
URL `{https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules}(6/7/2018)`

[11] Point Grey FLEA3 cameras.
URL `https://www.ptgrey.com/flea3-13-mp-color-gige-vision-cs-mount-sony-icx445-camera(6/24/2018)`

[12] T. Masters, Deep Belief Nets in C++ and CUDA C, 1st Edition, CreateSpace Independent Publishing Platform, 2015.

[13] G. Hinton, A Practical Guide to Training Restricted Boltzmann Machines, 2010.

[14] NVIDIA GTX1080 graphics cards series.
URL https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080/(6/19/2018)

[15] R. Durstenfeld, Algorithm 235: Random permutation, Commun. ACM 7 (7) (1964) 420–. doi:10.1145/364520.364540.

[16] Y. Bengio, Learning deep architectures for AI 2 (2009) 1–55.

[17] N. Venkatanath, D. Praneeth, B. M. Chandrasekhar, S. S. Channappayya, S. S. Medasani, Blind image quality evaluation using perception based features, 2015 Twenty First National Conference on Communications (NCC)doi:10.1109/ncc.2015.7084843.

[18] CUDA for Tegra, Tech. rep., NVIDIA Corporation (Feb 2018).

[19] CUDA C best practices guide, Available at http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html (2018/04/16).

[20] J. Cheng, M. Grossman, T. McKercher, Professional CUDA C Programming, Wrox, 2014.

Appendix

The development environment

Two machines were used for developing, each with its own compatible binaries and libraries for developing and testing the algorithms in question.

The Jetson System was flashed with Linux For Tegra (L4T) release 24.2.1 which runs a sample file system derived from Ubuntu v16.04 LTS. Flashing was done through an Ubuntu Linux x86_64 v14.04.

The Host System was a hexcore Intel Core i7-6700K CPU which needed aarch64 compilation libraries for cross-compilation on the Jetson.

CUDA 8.0 was used in the experiments on both machines.

The image processing part was done on the Jetson due to SDK compatibility issues with the ZED camera, CUDA 8.0 version and the Ubuntu 14.04 host machine.

Table 1: Hyperparameters used while training our experimental networks

Hyperparameter		
Name	Description	Value
Markov Chain Start	Beginning index of markov chain	1
Markov Chain End	Ending index of markov chain	4
Markov Chain Rate	Exponential Smoothing Rate of markov chain	0.5
Mean Field	Use meanfield activations = 0; use random sampling = 1	1
Greedy Mean Field	if == 0 ? use for training	1
Max Epochs	Maximum number of epochs for training	5
Max No Improvments	Converge if this many epochs without ratio improvements	5
Convergence Criterion	Convergence heuristic	0.5
Learning Rate	Stochastic Gradient Descent Rate	0.1
Momentum Start	Network's momentum start value	0.005
Momentum End	Network's momentum end value	1
Weight Penalty	Cost [a] assigned to weights	0.005
Sparsity Penalty	Sparsity [b] cost	0.1
Sparsity Target	Sparsity target value	0.1

[a] introduced parameter to discourage large weights associated with poor mixing rates
[b] average probability of a neuron being active

Hyperparameters Constants used in training experiments

We used the recommended values for setting training hyperparameters from [12] and [13]. Table 1 shows their descriptions and values used.

NVIDIA Tegra X1 Device Specifications

[0] NVIDIA Tegra X1

Compute Capability	5.3
Max. Threads per Block	1024
Max. Threads per Multiprocessor	2048
Max. Shared Memory per Block	48 KiB
Max. Shared Memory per Multiprocessor	96 KiB
Max. Registers per Block	32768
Max. Registers per Multiprocessor	65536
Max. Grid Dimensions	[2147483647, 65535, 65535]
Max. Block Dimensions	[1024, 1024, 64]
Max. Warps per Multiprocessor	64
Max. Blocks per Multiprocessor	32
Half Precision FLOP/s	36.864 GigaFLOP/s
Single Precision FLOP/s	36.864 GigaFLOP/s
Double Precision FLOP/s	1.152 GigaFLOP/s
Number of Multiprocessors	2
Multiprocessor Clock Rate	72 MHz
Concurrent Kernel	true
Max IPC	6
Threads per Warp	32
Global Memory Bandwidth	204 MB/s
Global Memory Size	3.901 GiB
Constant Memory Size	64 KiB
L2 Cache Size	256 KiB
Memcpy Engines	1

Figure 16: TX1 Device Specifications