Spring 6-8-2016

Machine Learning on the Cloud for Pattern Recognition

Tien Nguyen
San Jose State University

Writing Project

Machine Learning on the Cloud for Pattern Recognition

Final Report

Author

Tien Nguyen

CS 298

May 2016

Advisor

Dr. Chris Tseng

A Writing Project Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment of the Requirements for the Degree: Master of Science

The Designated Committee Approves the Master's Project Titled

Machine Learning on the Cloud for Pattern Recognition

By

Tien Nguyen

Approved for the Department of Computer Science

San José State University

May 2016

Acknowledgements

I would like to express my sincere gratitude to my project advisor, Dr. Chris Tseng, for his expertise, guidance, encouragement, and time. I would like to thank my committee members, Dr. Tsau Young Lin and Dr. Duc Than Tran, for their inputs and suggestions. Lastly, I would like to thank my family and friends for their kind support.

ABSTRACT

Pattern recognition is a field of machine learning with applications to areas such as text recognition and computer vision. Machine learning algorithms, such as convolutional neural networks, may be trained to classify images. However, such tasks may be computationally intensive for a commercial computer for larger volumes or larger sizes of images. Cloud computing allows one to overcome the processing and memory constraints of average commercial computers, allowing computations on larger amounts of data. In this project, we developed a system for detection and tracking of moving human and vehicle objects in videos in real time or near real time. We trained various classifiers to identify objects of interest as either vehicular or human. We then compared the accuracy of different machine learning algorithms, and we compared the training runtime between a commercial computer and a virtual machine on the cloud.

Table of Contents

Table of Figures

1 Project Description

1.1 Introduction

In this project we perform a comparison of several machine learning algorithms on the task of object classification in video surveillance. Among the algorithms we used are feed-forward neural networks, support vector machines, and convolutional neural networks. Video surveillance is the use of video cameras to watch over a location. Smart video surveillance is an automated form of video surveillance that integrates object detection, classification, object tracking, or behavior recognition [1]. Many object tracking and video surveillance-related techniques have been studied in literature [2] [3] [4] [5]. We trained the algorithms to classify images and then we applied the algorithms to regions of interest in frames in a video.

Feed-forward neural networks (NNs), or multi-layer perceptrons (MLPs) when the NNs have multiple computational layers, are non-cyclical networks of units called neurons between the input and output layers and can be used to predict output from some given input [6]. They are known to be vulnerable to a phenomenon called "overfitting" wherein a NN trained on a small dataset will perform poorly on new unknown data [6]. Various techniques exist that can reduce or prevent overfitting, such as random dropout, wherein hidden units are individually randomly ignored [6].

A support vector machine (SVM) is a binary classification algorithm that learns a high-dimensional decision boundary called the maximum margin hyperplane to classify inputs [7].

A convolutional neural network (CNN) is a type of feed-forward neural network that uses a type of network layer called a convolution layer. Convolutional layers are network layers in which the connections between inputs and neurons are defined by two-dimensionally regions [8]. The two-dimensional regions are known as local receptive fields and represent rectangular regions of an input source, such as an image. The convolution layers produce feature maps as output, which are usually smaller than the source input [8]. The size of a convolutional layer's output is determined by the size of the strides and the size of the feature maps. The stride indicates the amount of pixel distance between each step of the convolution computation used to produce the feature maps. Feature maps are also known as kernels. CNN architectures often also utilize pooling layers to reduce the complexity of the feature maps produced by the convolution layers [9] [8]. CNNs are able to learn spatial structures and, thus, are useful for image classification [8].

Histogram of oriented gradients (HOG) is a feature description technique that represents images as histograms of the orientations of edges within the image [10] [11]. HOG is an effective technique for detection of humans in images [10].

1.2 Literature Review

The use of CNN for computer vision has been extensively studied in literature [12] [13] [14] [15] . CNNs have been applied to a variety of problems, such as optical character recognition, bank check reading systems, and airport video surveillance [14]. The application of CNNs to the problem of classifying high-resolution images has also been studied [15]. In [12], a CNN was used for video surveillance, where the network was trained to estimate the position and size of objects in consecutive video frames. In many other existing object tracking approaches, pre-trained or online algorithms determine object candidates and other algorithms were used to track the candidates. When tracking objects, these other approaches did not utilize previously known information such as the object's previous position or size, which led to the tracker often making false positives when many similar object candidates were near a target object. The approach discussed in [12] took advantage of previously known information to reduce false positives.

In literature, there exist many studies on different approaches to the challenges of human detection and video surveillance [2] [3] [4] [5] [10]. The Histogram of Oriented Gradients technique was shown to be effective for detecting humans in images [10]. A technique for clustering tracked blobs in video using spatio-temporal information has been developed that allows for the tracking of multiple moving objects in a scene [2]. A vehicle tracking technique utilizing a genetic algorithm and particle filters has been studied that successfully track vehicles even in the event of occlusion [4]. A technique for segmenting or extracting objects from a sequence of images has also been studied [5].

1.3 Problem Statement and Project Goal

In our video surveillance problem, we required a means of processing video frames so that we could apply classification on. A video is composed of a sequence of frames, which are static images. We applied computer vision techniques to process the frames of the video before applying machine learning. We compared the accuracy of various machine learning algorithms to decide which those algorithms had the best performance.

In addition to the surveillance system, we provide a comparison of the training run time of the learning algorithms on two different machines. We used a 64-bit Windows 7 machine with an Intel® Core™ 2 Duo CPU T6500 @ 2.10GHz 2.10 GHz processor (PC) and an Ubuntu 14 virtual machine with an Intel® Xeon® processor E5 v3 family processor under the Microsoft Azure service's G-Series (VM) [16]. Microsoft Azure offers a variety of cloud computing services that include virtual machines, databases, and analytics [17].

Our project's primary goal was to build a video surveillance system that could automatically determine whether a moving object on-camera was either a human or vehicle. Such a system could be useful for driveway surveillance, for instance. Concretely, we required motion detection, object tracking, and object classification in the surveillance system. Thus, we found that computer vision and machine learning techniques were applicable to our problem.

We structure our report into sections that separate the details of the project design, implementation, workflow, and results. Section 1 describes the project's overall purpose. Section 2 explains our design for the system. Section 3 discusses our implementation of the design. Section 4 provides details on the workflow of using our implemented code to run the system on a trained CNN. Section 5 discusses the results we obtained, and Section 6 presents our analysis of our results. Finally, Section 7 concludes our report.

2 Project Design

This section describes the design details of our system.

2.1 Video Surveillance System Design

The goal for the video surveillance system was to implement the following key features: motion detection, object tracking, and object classification. We used motion detection to find objects of interest, which we then applied object tracking and object classification techniques on.

Figure 1 summarizes the cycle in which the video surveillance system operates. First, the system reads a frame from the source video. Next, the system searches for track-able feature points and updates any currently tracked feature points. Features are patterns in an image, but track-able feature points are features that distinguishable and relatively easy to locate in an image, such as the corner of an object [19]. After finding and updating feature points, the system found candidate objects by performing a background subtraction algorithm [19]. We consider a moving object to be a candidate if its pixel size was large enough such that it could be a human or vehicle. A candidate object had a pixel position on the screen, pixel width and height bounds, an image snapshot of the object, and a hue histogram. Objects that had sizes below the threshold were considered noise. This size threshold must be adjusted before running the video surveillance system. The object candidates are compared to currently tracked objects to determine if the tracked objects have moved or not. If a candidate had a high enough similarity to a tracked object, then the tracked object was assumed to be the same as the candidate and the tracked object was updated with the candidate object's information such as position and bounds. Similarity was considered based on color, velocity of tracked feature points, relative position, and proximity. Any candidates not associated to an already tracked object are then tracked by the system. Afterwards, the system checks for lost objects and removes them from the list of tracked objects. An object is considered lost if it has not moved significantly for a set amount of time. This time threshold is adjustable. The system then performs a classification algorithm on each of the currently tracked objects and displays the results, such as the classification label and a tracking identifier, onto the screen above the appropriate objects. The system then repeats the process until either the video has no more frames to be read or the user terminates the program.

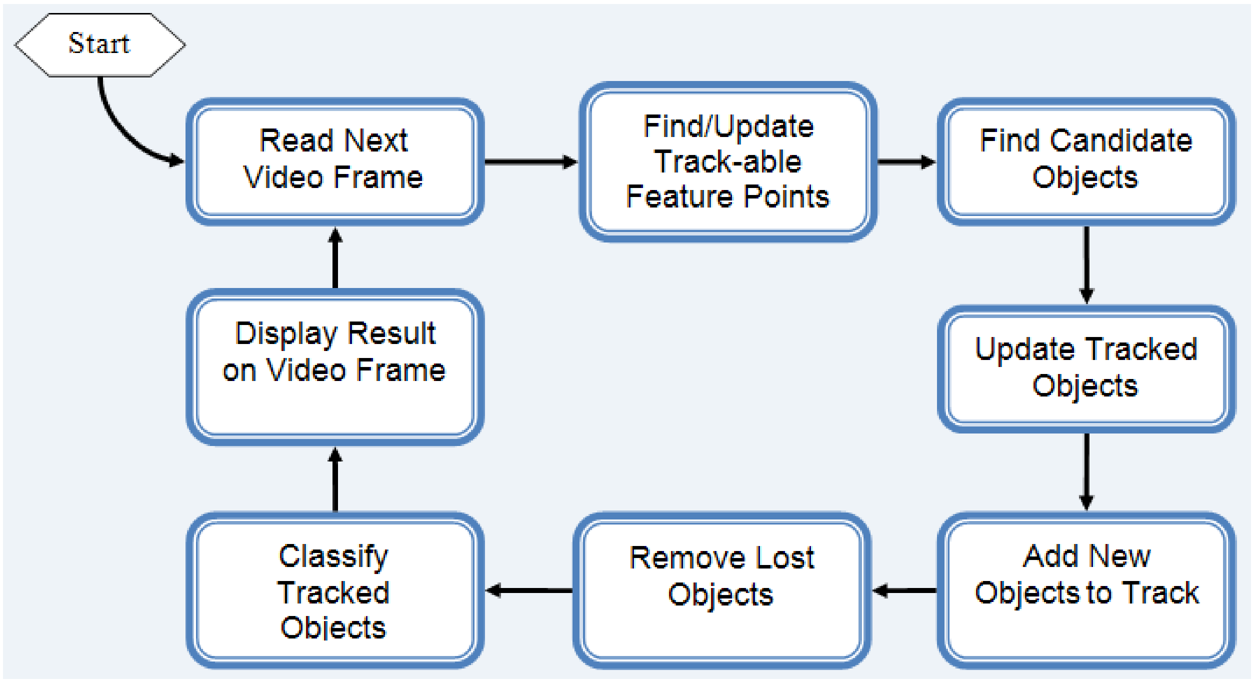


Figure 1 Video Surveillance Process Flowchart

Figure 2 shows snapshots of the surveillance system applied to a pre-recorded video (from [20]). These sample show that as the woman in the background enters the scene, the system detects, classifies, and tracks her. Furthermore, the identification numbers of tracked objects tend to stay consistent.

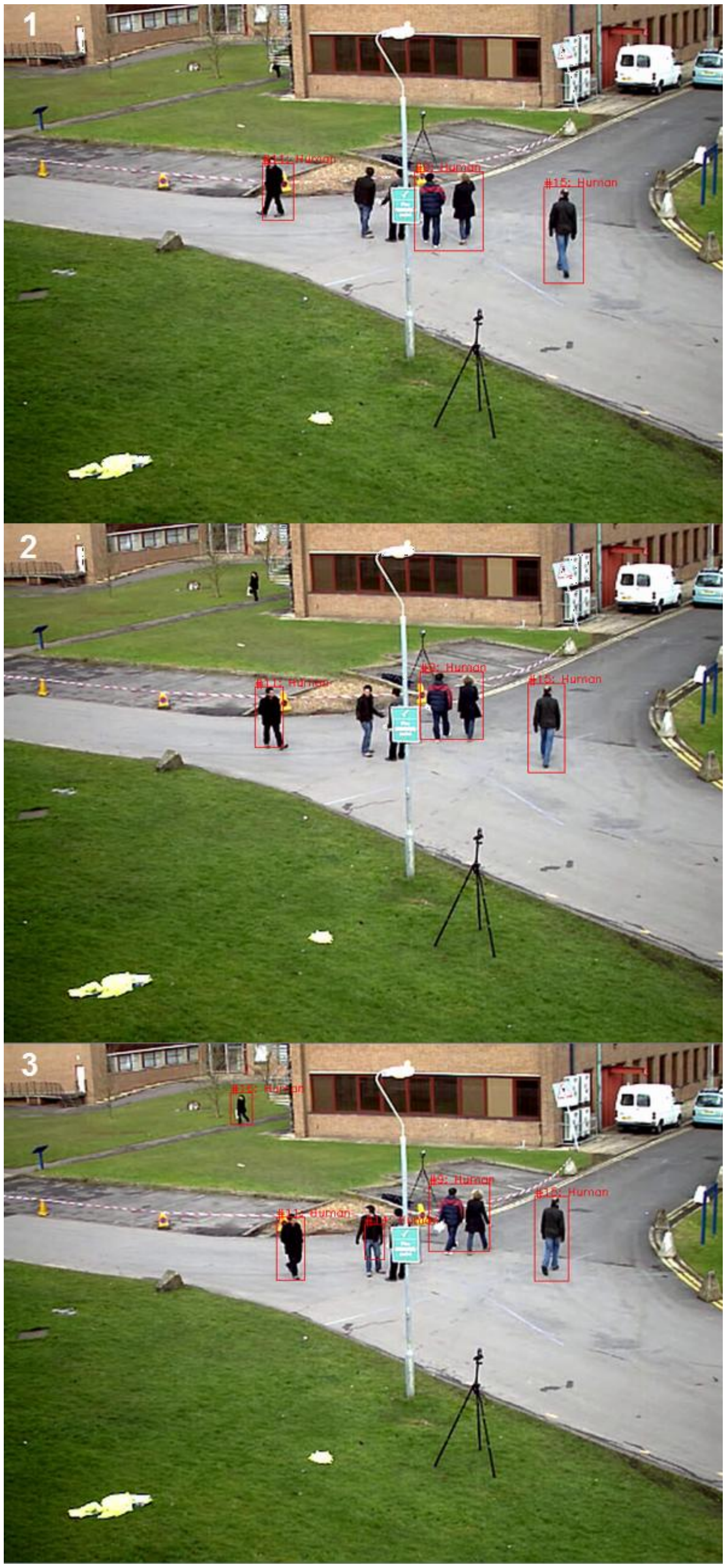


Figure 2 Surveillance System Snapshots

3 Project Implementation

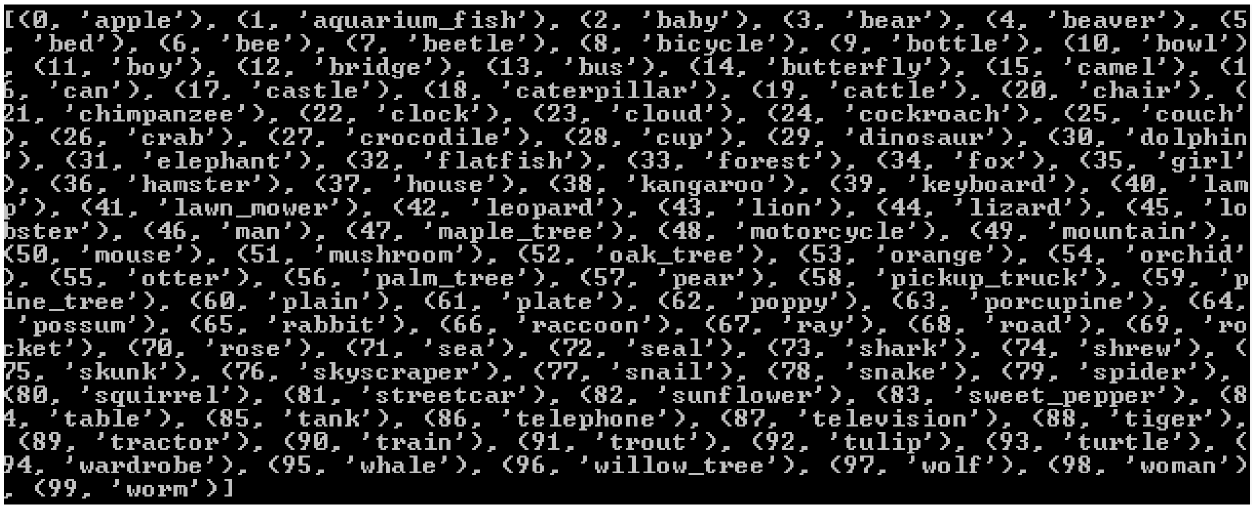
This section discusses the implementation of the video surveillance system through software tools such as OpenCV. In Section 3.1, we provide an overview on the image dataset we used for training. Section 3.2 provides our chosen classes for classification. In Section 3.3, we describe the installation and setup of software tools we used on both the VM and PC. In Section 3.4, we describe how we preprocessed the data used for training the machine learning classifiers. In Section 3.5, we describe the machine learning classifiers. In Section 3.6, we describe implementation details of the video surveillance system with respect to motion detection, object classification, and object tracking.

3.1 Dataset and Data Format

In order to train our machine learning algorithm to be able to identify objects, we required a training set of images. Many image datasets for research purposes can be found online, such as Caltech 101, PASCAL VOC, and Stanford Dogs [17]. However, depending on how we decided to feed the images to the algorithm, large images could potentially result in slow processing times. Furthermore, we required images of various vehicles and humans separated by type. Thus, we chose the CIFAR-100 dataset, which contained small 32x32 color images that included categories such as men, women, bicycles, and buses [18]. A sample of the CIFAR-100 images is shown in Figure 3. The images were originally in a unique format in which the image data and corresponding labels were contained within a Python dictionary. The dictionary had an entry containing a numpy array of uint8 values, where each row in the array represented one image. The first 1024 columns held the values for the red color channel, the next 1024 held the green color channel, and the last 1024 held the blue color channel [18]. We describe our method for processing the images in Section 3.4.

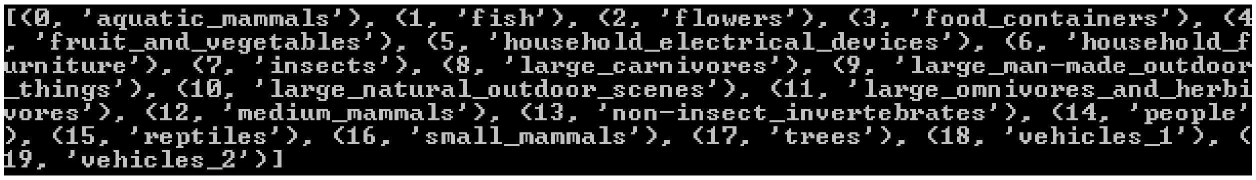
Figure 3 Samples of CIFAR-100 Images

The CIFAR-100 dataset contained exactly 100 types (classes) of images and 600 images of each class for a total of 60,000 images [18]. These classes described the images with a "fine" granularity for labels, with labels being such as beaver, orchids, man, and dinosaur. The metadata provided for CIFAR-100 also contained a "coarse" labeling for the images, which represented the "superclasses" of the images. The coarse labels included fish, reptiles, and people, among others. The dataset contained 20 superclasses, each encompassing 5 fine class labels. For example, the "people" superclass label encompassed the following classes: baby, boy, girl, man, and woman. The full list of superclasses and classes can be found in [18]. The metadata of the CIFAR-100 dataset identified classes and superclasses by integer values. The original values for the classes and superclasses paired with their corresponding names are shown in Figure 4 and Figure 5, respectively.



```
[(0, 'apple'), (1, 'aquarium_fish'), (2, 'baby'), (3, 'bear'), (4, 'beaver'), (5
, 'bed'), (6, 'bee'), (7, 'beetle'), (8, 'bicycle'), (9, 'bottle'), (10, 'bowl')
, (11, 'boy'), (12, 'bridge'), (13, 'bus'), (14, 'butterfly'), (15, 'camel'), (1
6, 'can'), (17, 'castle'), (18, 'caterpillar'), (19, 'cattle'), (20, 'chair'), (
21, 'chimpanzee'), (22, 'clock'), (23, 'cloud'), (24, 'cockroach'), (25, 'couch'
), (26, 'crab'), (27, 'crocodile'), (28, 'cup'), (29, 'dinosaur'), (30, 'dolphin
'), (31, 'elephant'), (32, 'flatfish'), (33, 'forest'), (34, 'fox'), (35, 'girl'
), (36, 'hamster'), (37, 'house'), (38, 'kangaroo'), (39, 'keyboard'), (40, 'lam
p'), (41, 'lawn_mower'), (42, 'leopard'), (43, 'lion'), (44, 'lizard'), (45, 'lo
bster'), (46, 'man'), (47, 'maple_tree'), (48, 'motorcycle'), (49, 'mountain'),
(50, 'mouse'), (51, 'mushroom'), (52, 'oak_tree'), (53, 'orange'), (54, 'orchid'
), (55, 'otter'), (56, 'palm_tree'), (57, 'pear'), (58, 'pickup_truck'), (59, 'p
ine_tree'), (60, 'plain'), (61, 'plate'), (62, 'poppy'), (63, 'porcupine'), (64,
 'possum'), (65, 'rabbit'), (66, 'raccoon'), (67, 'ray'), (68, 'road'), (69, 'ro
cket'), (70, 'rose'), (71, 'sea'), (72, 'seal'), (73, 'shark'), (74, 'shrew'), (
75, 'skunk'), (76, 'skyscraper'), (77, 'snail'), (78, 'snake'), (79, 'spider'),
(80, 'squirrel'), (81, 'streetcar'), (82, 'sunflower'), (83, 'sweet_pepper'), (8
4, 'table'), (85, 'tank'), (86, 'telephone'), (87, 'television'), (88, 'tiger'),
 (89, 'tractor'), (90, 'train'), (91, 'trout'), (92, 'tulip'), (93, 'turtle'), (
94, 'wardrobe'), (95, 'whale'), (96, 'willow_tree'), (97, 'wolf'), (98, 'woman')
, (99, 'worm')]
```

Figure 4 CIFAR-100 Classes and Corresponding Label Values



```
[(0, 'aquatic_mammals'), (1, 'fish'), (2, 'flowers'), (3, 'food_containers'), (4
, 'fruit_and_vegetables'), (5, 'household_electrical_devices'), (6, 'household_f
urniture'), (7, 'insects'), (8, 'large_carnivores'), (9, 'large_man-made_outdoor
_things'), (10, 'large_natural_outdoor_scenes'), (11, 'large_omnivores_and_herbi
vores'), (12, 'medium_mammals'), (13, 'non-insect_invertebrates'), (14, 'people'
), (15, 'reptiles'), (16, 'small_mammals'), (17, 'trees'), (18, 'vehicles_1'), (
19, 'vehicles_2')]
```

Figure 5 CIFAR-100 Superclasses and Corresponding Label Values

3.2 Human and Vehicle Classification

Although the CIFAR-100 dataset contained 100 classes of images, only a subset of those images were of interest for our system. Concretely, we only needed the classes shown in Table 1.

Table 1 CIFAR-100 Superclasses and Classes of Interest

Superclass	Class
People	Baby, boy, girl, man, woman
Vehicles 1	Bicycle, bus, motorcycle, pickup truck

3.3 Software Installation and Setup

In order to implement our system, we utilized various libraries for version 2.7.10 of the Python programming language. The main libraries we used were OpenCV (version 2.4.11), Lasagne (version 0.1), Theano (version 0.7.0.dev), and Numpy (version 1.9.3). OpenCV is an open source library for processing and manipulating images for computer vision [20]. Its features also include various machine learning algorithms that are commonly used in computer vision. However, OpenCV does not provide implementations of CNNs, so we used the CNNs provided by the Lasagne library. Lasagne is a library that allows customization of feed forward neural networks such as CNN and recurrent networks [21]. Lasagne is built on top of Theano [21], which is a library for efficiently evaluating multi-dimensional array-based mathematical expressions [22] [23] [24]. Numpy is a library that provides N-dimensional array functionality [25], which was useful for representing our image data.

We installed and set up the previously mentioned software on two platforms: Windows 7 and Ubuntu Server 14.04 LTS (Ubuntu). On Windows 7, we used the 64-bit WinPython version 2.7.10.2 to run python. On Ubuntu, we ran python using the Anaconda Python 2.7 distribution.

3.3.1 Setup and Installation on Windows 7

We started our system implementation by setting up the environment on our Windows 7 machine (PC). We first followed the instructions on the WinPython website [26] to install WinPython. Next, we installed OpenCV, Theano, and Lasagne. Numpy came pre-installed with WinPython.

3.3.1.1 Install OpenCV

OpenCV was installed through the following instructions.

1. Download the OpenCV self-extracting archive from their download page on Sourceforge.
2. Unpack the self-extracting archive.
3. Add OpenCV to your system's user path variables.
4. Add the OpenCV binary path to your system's PATH variable.

First, we downloaded version 2.4.11 of the OpenCV self-extracting archive from OpenCV's Sourceforge download page (http://sourceforge.net/projects/opencvlibrary/files/opencv-win/). This file is approximately 280 MB in size.

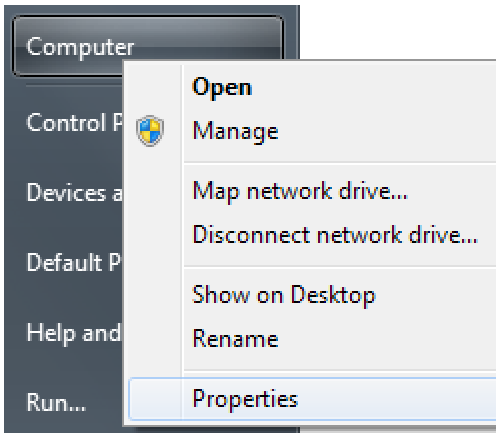
Next, we extracted the archive to your desired directory. The extracted directory was larger than 3.70 GB in size.

Afterwards, we added OpenCV to the user path variables of our system. Since our 64-bit Windows 7 machine had Visual Studio 2013, we ran the following command in a Windows terminal.

```
setx OPENCV_DIR D:\OpenCV\Build\x64\vc12
```

Finally, we added the OpenCV binary directory to our system's PATH variable using the following instructions.

1. Click the "Start" button on the task bar.



2. Hover over the "Computer" item on the Start Menu and right click.

3. Select Properties on the menu.

4. If prompted, enter your administrator's credentials and click "OK".

5. On the new window, click the "Advanced" tab, then click the "Environment Variables…" button.

6. Under the “System variables” section, select the “Path” variable and click the “Edit…” button.

7. If the Path variable already has contents, append a semicolon to the end if there is not one yet.

8. Append `%OPENCV_DIR%\bin` to the end of the Path variable.

9. Click the "OK" button on the following windows: “Edit System Variable”, “Environment Variables”, and “System Properties”.

3.3.1.2 Install Theano

Since Lasagne required a more recent version of Theano than the release version [27], we ran the following command in the WInPython Command Prompt to upgrade the version of Theano that was already installed on WinPython to version 0.7.0.dev-5429c30a5c74877bf06ad6654aa40c21971bf3f7.

```
pip install --upgrade --no-deps git+git://github.com/Theano/Theano.git
--user
```

3.3.1.3 Install Lasagne

We installed Lasagne with the following command in the WinPython Command Prompt.

```
pip install Lasagne
```

3.3.1.4 Additional OpenCV Setup for WinPython

There were several other instructions that we followed so that OpenCV would be usable in our Python scripts and so that OpenCV could access video files.

1. Go to the *build/python/2.7/x64 directory* of the OpenCV installation.
2. Copy the cv2.pyd file to *python-2.7.10.amd64/Lib/site-packages* directory of your WinPython installation.
3. Go to the *sources/3rdparty/ffmpeg* directory of your OpenCV installation.
4. Copy the *opencv_ffmpeg.dll* and *opencv_ffmpeg_64.dll* files to the *python-2.7.10.amd64* directory of your WinPython installation.
5. In the *python-2.7.10.amd64* directory of your WinPython installation, rename the *opencv_ffmpeg2411.dll* and *opencv_ffmpeg_64.dll* files to *opencv_ffmpeg.dll* and *opencv_ffmpeg2411_64.dll*, respectively.

3.3.2 Setup and Installation on Ubuntu

The process of installing OpenCV on Ubuntu differed from that of the installation on Windows 7.

3.3.2.1 Install OpenCV

Installation of OpenCV requires various dependencies and options [20] [28] [29] [30].

We ran the following commands in the command line to install dependencies.

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install build-essential
sudo apt-get install cmake git libgtk2.0-dev pkg-config libavcodec-dev
libavformat-dev libswscale-dev
sudo apt-get install python-dev python-numpy libtbb2 libtbb-dev
libjpeg-dev libpng-dev libtiff-dev libjasper-dev libdc1394-22-dev
sudo apt-get install libatlas-base-dev gfortran
sudo apt-get install python-dev
```

Afterwards, we needed to install Python 2.7.10 separate from our Anaconda distribution using the following commands [31].

```
sudo apt-get install build-essential checkinstall
sudo apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-dev
libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev
cd /usr/src
wget https://www.python.org/ftp/python/2.7.10/Python-2.7.10.tgz
tar xzf Python-2.7.10.tgz
cd Python-2.7.10
sudo ./configure --enable-shared
sudo make altinstall
```

We used the `--enable-shared` option when running `configure` because we found that it was needed in our VM setup. Next, we run the following commands to download and unzip OpenCV, where <opencv_dir> is the directory in which we installed OpenCV into.

```
wget "http://sourceforge.net/projects/opencvlibrary/files/opencv-
unix/2.4.11/opencv-2.4.11.zip"
unzip opencv-2.4.11.zip –d <opencv_dir>
cd <opencv_dir>
mkdir realease
cd release
cmake -DCMAKE_BUILD_TYPE=RELEASE -DCMAKE_INSTALL_PREFIX=/usr/local -
DBUILD_NEW_PYTHON_SUPPORT=ON -DBUILD_opencv_python=ON –
DINSTALL_PYTHON_EXAMPLES=ON -DWITH_CUDA=ON -
DPYTHON_INCLUDE_DIRS=/usr/local/include/python2.7 -
DPYTHON_LIBRARY=/usr/local/lib/python2.7/config/libpython2.7.a ..
make -j4
sudo make install
export PYTHONPATH=$PYTHONPATH:/usr/local/lib/python2.7/site-packages
```

3.3.2.2 Install Theano and Lasagne

We installed Theano and Lasagne using commands similar to what we used on the Windows 7 machine.

```
pip install --upgrade --no-deps git+git://github.com/Theano/Theano.git
--user
pip install Lasagne
```

3.4 Data Preprocessing

3.4.1 CIFAR-100

We applied a few techniques to the CIFAR-100 dataset to make it easier to work with. First, we used the code shown in Figure 6 to convert CIFAR-100's array format to a grayscale format easily usable with OpenCV functions. Although OpenCV can handle both color and grayscale images, we chose to work with grayscale because of the reduced amount of color channels and because lighting variation interfered with learning [32].

```
# Get the ith image from the CIFAR-100 dataset file.
# Make it grayscale and usable with OpenCV.
dict = self.unpickle(dataset_file)
newshape = (32, 32, 3)
cifar_img = dict['data'][i]:
img = np.ndarray(newshape, dtype=np.uint8, buffer=cifar_img,
order='F')
img = img.transpose((1,0,2))
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Figure 6 Code for Converting CIFAR-100 Images to Grayscale

Another preprocessing technique that we applied to the images was histogram of oriented gradients (HOG). HOG is a feature descriptor that can represent an entire object--such as a human--in a vector that "summarizes" the image gradients of the image [11] [33] [10]. HOG descriptors can be much smaller than the images they describe. We apply HOG in only two of our classifiers.

When observing several of the images, we noticed that certain classes in the human superclass contained images that we thought were too different from the other classes.

Specifically, the "baby" class contained many images that we thought would be confusing to the classifier, so we omitted it. A sample of various images of people found in the CIFAR-100 dataset is shown in Figure 7.

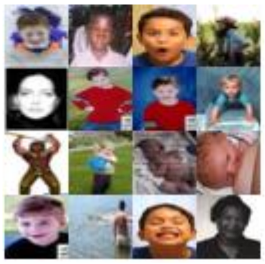


Figure 7 Human Samples from CIFAR-100

We also tried separating the classes in the "vehicles 1" class into two new superclasses. Bicycles and motorcycles would belong in one class, which we called "bikes", and buses and pickup trucks would belong in another class, which we called "vehicles".

3.4.2 Surveillance Images

As an alternative to the CIFAR-100 images, we created our own dataset of human and vehicle images taken from surveillance videos. We noticed that a significant amount of the human samples in the CIFAR-100 image dataset, such as those shown in Figure 7, were in poses that did not closely resemble those of people found in the surveillance videos that we used. For instance, many of the human samples in CIFAR-100 were portraits. The surveillance footage we used for testing, however, feature people whose whole bodies were visible. Figure 8 shows a snapshot from a surveillance video (from [20] and [19]), where the whole bodies of people are visible. Figure 9 shows numerous sample images of people taken from the same surveillance video. In addition to issue with the human images, the only cars found in the CIFAR-100 dataset were pickup trucks. To increase the variety of vehicles to match those found in surveillance videos, we included cars with sizes ranging from sub-compact to minivan into our surveillance image dataset. We note that the surveillance image dataset contains multiple images of the same objects in different poses or angles.

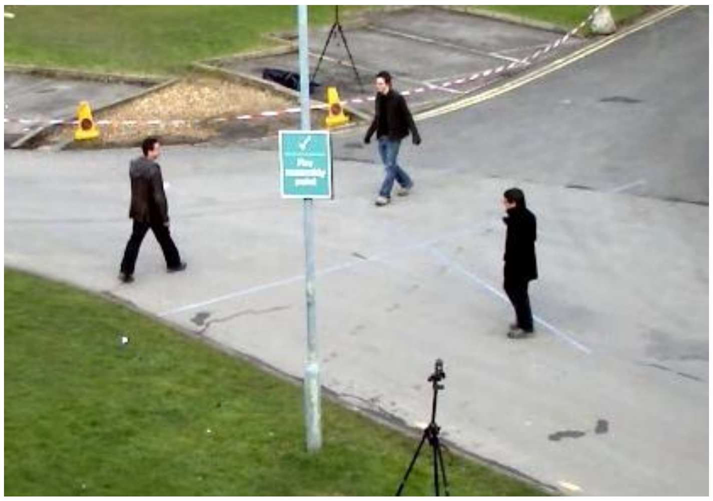


Figure 8 Surveillance Video Snapshot

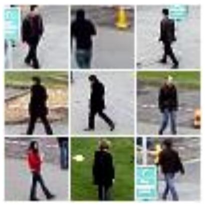


Figure 9 Human Samples from Surveillance Images

3.5 Training Machine Learning Classifiers

In order to perform classification on the CIFAR-100 images, we tried various machine learning algorithms. Among these algorithms are multi-layer perceptrons, convolutional neural networks, and support vector machines. This section discusses the settings we used for each machine learning algorithm. Each machine learning algorithm was trained on the subset of CIFAR-100 alone, the surveillance images, and when possible, pre-trained with the subset of CIFAR-100 before re-training with surveillance images.

Before we trained the algorithms, we needed to adjust the datasets. For the CIFAR-100 dataset we split the object classes into two classes: vehicle and human. These classes corresponded to class label 0 and class label 1, respectively, from the CIFAR-100 dataset. Table 2 summarizes the mapping of CIFAR-100 image classes to integer class labels. In preparation of training, we split our dataset into 3200 training images, 800 validation images, and 800 testing images. Each of these image sets had an even distribution of each of the eight image classes. The human images made up 50 percent of each sample, and the vehicle images made up the rest of the 50 percent. We used the same amounts and distributions of samples for the surveillance dataset.

Table 2 Vehicle and Human Class Labels

Class labels	Image classes from CIFAR-100
0	Bicycle, motorcycle, bus, pickup truck
1	Boy, girl, man, woman

Table 3 Dataset Split

Set	Amount of Samples
Training	3200
Validation	800
Testing	800

3.5.1 Convolutional Neural Network Training

We used the CNN with the following model shown in Table 4 [34]. Unless otherwise specified, stride of the convolution operation is 1x1, and the stride of the pooling operation is the same as the pool size.

Table 4 CNN Architecture

Layer	Settings
2D Convolutional	Amount of filters = 64, filter size = 5x5, rectify activation function, Glorot Uniform weight distribution for filter weights
Max-Pooling	Size = 2x2
2D Convolutional	Amount of filters = 32, filter size = 5x5, rectify activation function, Glorot Uniform weight distribution for filter weights
Max-Pooling	Size = 2x2
Fully Connected	Dropout = 50%, number of units = 256, rectify activation function
Fully Connected	Dropout = 50%, number of units = 2, softmax activation function

3.5.2 Multi-layer Perceptron with Histogram of Oriented Gradients

Many techniques exist for training a MLP on images. A simple method would be to convert each image into grayscale, where values range from 0 to 255, and feed the grayscale images to the MLP for training and classification. However, in doing so each of our input images would be composed of 1024 features. To reduce the amount of features per image, we use the HOG feature descriptor with 16 bins to convert a 32x32 pixel grayscale image into 64 features [35]. The layer architecture of our MLP is summarized in Table 5. We used the same class-labeling as in Table 2 and the same amount of training, validation, and testing samples as in Table 3.

Table 5 MLP Architecture

Layer	**Settings**
Hidden layer	Number of units = 129; rectifier activation function
Output layer	Number of units = 2; softmax activation function

3.5.3 Support Vector Machine with Histogram of Oriented Gradients

In a similar manner to the approach used with the MLP, we used HOG to compute the features that we fed to the SVM for training. The same class labeling and dataset splitting as in Table 2 and Table 3, respectively, were used for training and testing the SVM. We used the following settings for the SVM:

- Each input sample has 64 features.
- Kernel: Linear
- Type: SVM_C_SVC
- C = 2.67
- Gamma = 5.383

3.6 Smart Video Surveillance System Details

This section discusses our implementation of the system design described in Section 2.1. When implementing the system, we considered the cycle to occur in four phases, as shown in Figure 10. The following subsections describe the implementation of the phases.

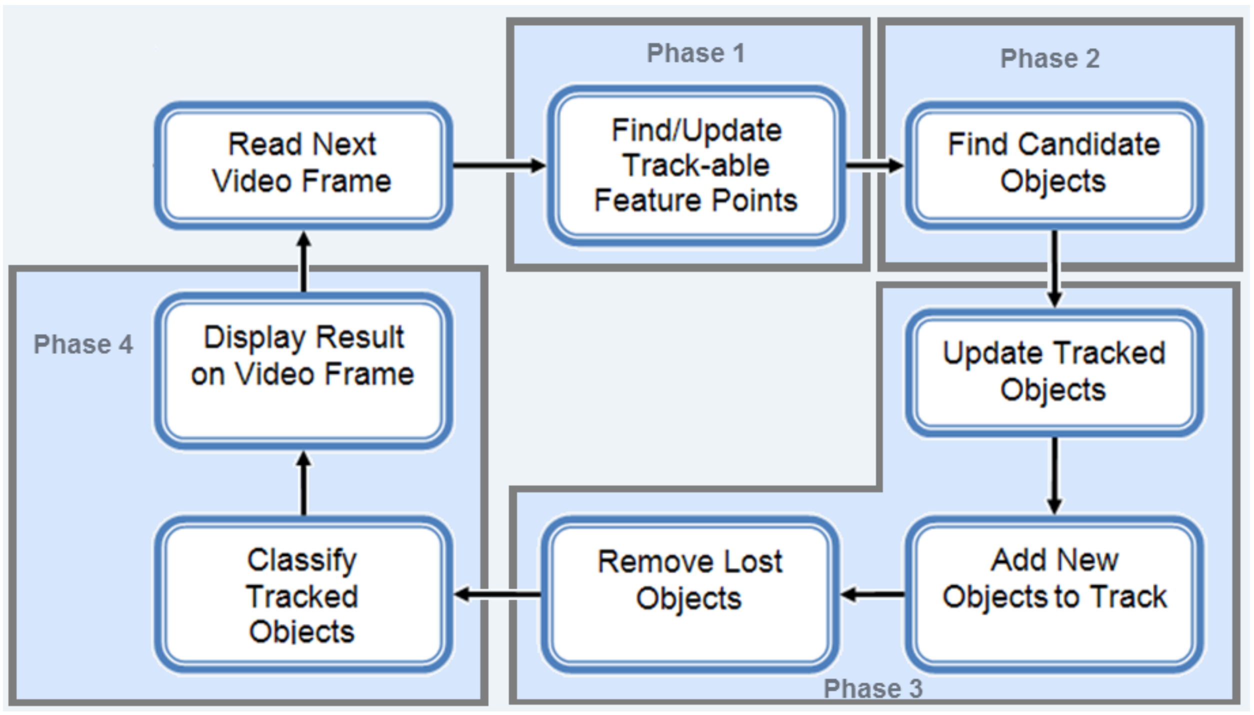


Figure 10 Four Phases of the System Cycle

3.6.1 Phase 1: Update Tracked Feature Points

In Phase 1, the system updated previously tracked feature points and found new feature points. We used the Lucas-Kanade method via the calcOpticalFlowPyrLK function in OpenCV to obtain feature points. When tracking a feature point, the system kept a record of the point's positions since the time at which it was first found. The new feature points were used as one aspect of the object tracking mechanism and were assigned to candidate objects in Phase 2.

3.6.2 Phase 2: Find Candidates

Phase 2 dealt with finding enough information about candidate objects such that tracking would be effective. We used motion detection in our project to locate objects of interest within a particular frame of video. During each frame, we applied Gaussian Mixture-based Background/Foreground Segmentation, called MOG2 in OpenCV [19]. MOG2 returned what was known as a foreground mask, which was a black and white image where white indicated a foreground object and black indicated the background of a video frame [19]. When enough sequential frames are fed to MOG2, the algorithm is able to separate moving objects from the background. Figure 11 shows an example of a video frame (from [19], [20]) and the corresponding foreground mask at that point in the video.

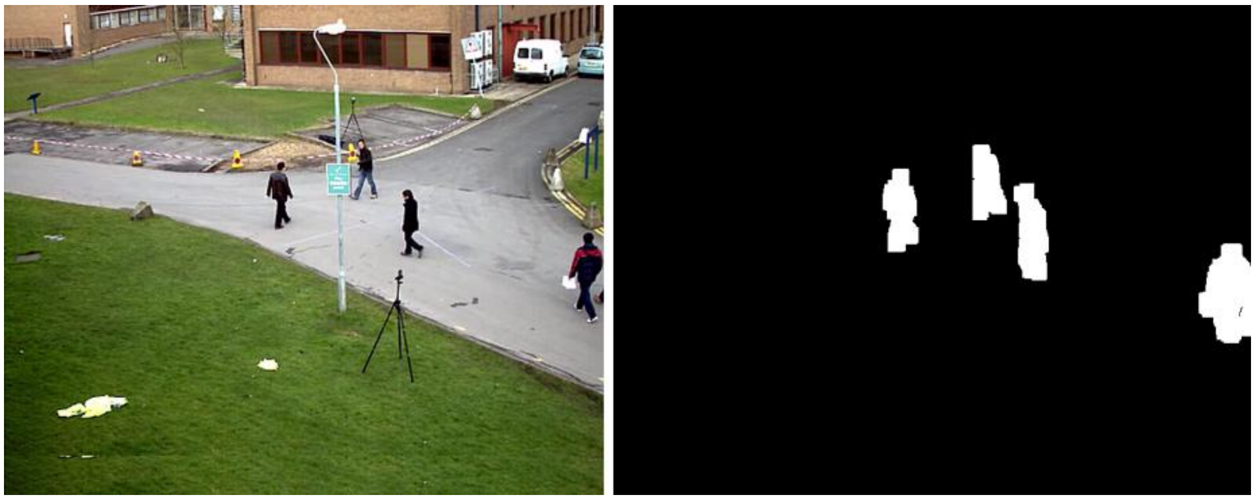


Figure 11 Video Frame (left) and MOG2 Foreground Mask (right)

After producing the foreground mask, we used the findContours function of OpenCV to obtain the contours, or pixel position and pixel size information about the white regions from the foreground mask. We assumed each white region, or contour, to represent a separate object. We used the pixel positions and pixel dimensions of each contour to determine regions on the frame to use for object classification and tracking. For each region of interest, we created a square sub-image containing the region at the center and having a length equal to the larger of the region's width and height. We scaled the sub-image to a 32x32 pixel image. Our concept of a candidate object included the contour of the object, the 32x32 pixel image, a hue histogram of the image before it was converted to grayscale, and the white region of the object on the frame.

The new feature points found in Phase 1 were assigned to candidate objects found via motion detection. Each newly found feature point was assigned to the candidate in which the point's position was located on the candidate's white region.

Each candidate object was classified as either human or vehicle via the trained convolutional neural network. The images of each candidate object were converted to grayscale so that their dimensionality would match the dimensionality and amount of color channels as the images used to train the classifier, since the machine learning algorithms required that the input images be of the same dimensions. The CNN returned the class label of each image, which the candidate object's classification was then set to.

3.6.3 Phase 3: Update List of Tracked Objects

In Phase 3, the system was responsible for updating the list of tracked objects. This task included assigning candidates to similar tracked objects and tracking objects that were not yet tracked. Tracked objects that were considered lost were removed from the list of tracked objects. In the following paragraphs, we say that a tracked object "consumes" a candidate when the candidate is assigned to the tracked object.

As mentioned in Section 2.1, we used a measure of similarity to determine that a tracked object should consume a candidate. Similarity was based on relative position, proximity, color, and velocity of tracked feature points. To determine if a candidate object was possibly the same as a tracked object, the system compared the distance between the two objects and their color. Color similarity was implemented via a comparison between the color histograms of two object's images. Each object's color image was first converted to HSV color space before their histogram could be computed through OpenCV's *cv2.calcHist* function. The color histograms were compared using the *cv2.compareHist* function using the Bhattacharyya distance method. Since the Bhattacharyya distance method returns 0.0 for exactly similar images and numbers close to 1.0 for dissimilar images, we subtracted the return value of *cv2.calcHist* from 1.0 to obtain a number that was higher for higher similarity and lower for lower similarity. If the resultant comparison value was above a threshold, the candidate and tracked object were considered to have the same colors. The velocity of a tracked object was used to reduce the likelihood that a candidate is assigned to a tracked object that is moving in a different direction. We observed that a moving object would generally move in the same direction that it was moving in the previous frame. To calculate the velocity of a previously tracked object, we took the difference between a tracked point previous position and its current position. The average of up to 3 of the newest positions for each tracked point were considered, and the average of each tracked point's velocity was used as the tracked object's velocity.

When a tracked object consumed a candidate its classification, position, bounds, list of tracked feature points, and list of contours were updated with the information of the candidate. The classification of the candidate object was added to the tracked object's list of classifications up to a certain amount. The x-coordinate of the tracked object was set to the average between the candidate's x-coordinate and the tracked object's y-coordinate. The y-coordinate was calculated similarly. The candidate's list of feature points and list of contours were added to the tracked object's lists. The tracked object's width was set to `avg(tracked_width, max(tracked_x, candidate_x) - min(tracked_x, candidate_x))`, where tracked_width is the width of the tracked object, tracked_x is the tracked object's pixel x-coordinate, and candidate_x is the candidate object's pixel x-coordinate. The *avg* function returns the average of the two arguments, the *max* function returns the maximum of the two arguments, and the *min* function returns the minimum of the two arguments. The tracked object's height was set to `avg(tracked_height, max(tracked_y, candidate_y) - min(tracked_y, candidate_y))`, where tracked_height, tracked_y, and candidate_y correspond to the tracked object's height, the tracked object's pixel y-coordinate, and the candidate's y-coordinate, respectively.

To determine whether a tracked object was lost or not, the system gave each tracked object a counter that we called staleness. The system considered an object to be lost only if it has not moved after a set number of frames. The staleness counter was used to count the number of frames in which a tracked object has not moved. At the beginning of Phase 3, each tracked object's staleness was incremented. If a tracked object consumed a candidate object, then the object's staleness was reset to zero, since the object was not lost.

At the end of this phase, the system performed clean up operations. Tracked objects whose edges touched the edge of the frame had their staleness counters increased by a large amount, since it was likely that the object was moving outside of the view. Objects whose staleness reached a certain threshold were removed from the list of tracked objects. Any feature points found to be outside of their tracked object's bounds were removed from the tracked object's list of tracked feature points.

3.6.4 Phase 4: Display Results

At the end of the cycle, the information of the tracked objects were displayed to the screen. Track objects that had a staleness counter that exceeded a certain threshold or that were considered lost were ignored. Furthermore, the system ignored displaying a tracked object if the object did not exceed a certain amount of same-class classifications. The system displayed the tracking ID number and the classification of each tracked object above the corresponding tracked object, and the system drew a rectangle around each tracked object with the same coordinates and dimensions as the object.

4 Workflow

In this section, we describe the workflow from training the CNN to running the video surveillance system. We also include the commands for training the NN and SVM, but our video surveillance code does not currently support NN or SVM. A link to our source code can be found in the appendix in Section 9.1.

4.1 Train the Classifiers

Our source code contains programs to train the CNN, NN, and SVM.

4.1.1 Train a Convolutional Neural Network

Figure 12 shows the terminal command for running the code to train a CNN on the human and vehicles samples from the CIFAR-100 dataset. In the command,

- *<CIFAR100_TRAIN>* is the pickled training dataset file from CIFAR-100 [18].
- *<CIFAR100_TEST>* is the pickled test dataset file from CIFAR-100 [18].
- *<DEST>* is the destination file to save the trained CNN's weights to.

```
python project/ml/train_cnn.py -f <CIFAR100_TRAIN> -t
<CIFAR100_TEST> -d <DEST>
```

Figure 12 Command for Training CNN on CIFAR-100

Figure 13 shows the terminal command for running the code to train a CNN on the human and vehicles samples from the surveillance dataset. In the command,

- *<CONFIG_FILE>* is a JSON file containing a single JSON object with properties specifying the parameters to be used by the program:
  - *"src_dir"* - string, directory containing directory of images. These inner directories should have same name as the integer class labels, e.g. 0 or 1.
  - *"dest"* - string, destination file path to save trained network to.
  - *"num_classes*" - int, amount of classes.
  - *"num_epochs"* - int, number of epochs to run training for.
  - *"predictionOutputFile"* - string indicating desired filepath to output prediction output to, or null for none.

```
python project/ml/train_cnn_surveillance.py --config
<CONFIG_FILE>
```

Figure 13 Command for Training CNN on Surveillance Images

Figure 14 shows the command used for loading a CNN trained on the CIFAR-100 samples and re-training the CNN on the surveillance images. Similar to the previous command, the <CONFIG_FILE> parameter is the filename of the JSON file with configuration settings:

- *"cnn_file"* - string, path to the weights of the CNN trained on the CIFAR-100 samples, as an .npy file.
- *"cifar_test_data"* - string, path to the test dataset file from CIFAR-100 [18]. This is used for testing only.
- *"arch"* - string, the identifier for the CNN architecture as indicated in project/ml/lasagna_cnn.py. We leave this set to "ex".
- *"src_dir"* - string, directory containing directory of images. These inner directories should have same name as the integer class labels.
- *"dest"* - string, destination filepath to save trained network to.
- *"num_classes*" - int, amount of classes.
- *"num_epochs"* - int, number of epochs to run training for.
- *"predictionOutputFile"* - string indicating desired filepath to output prediction output to, or null for none.

```
python project/ml/train_cnn_cifar_surveillance.py --config
<CONFIG_FILE>
```

Figure 14 Command for Re-Training CNN on Surveillance Images

4.1.2 Train a Neural Network

Figure 15 shows the terminal command for running the code to train a NN on the human and vehicles samples from the CIFAR-100 dataset. The argument names are identical to those for CNN discussed in Section 4.1.1. above:

- *<CIFAR100_TRAIN>* is the pickled training dataset file from CIFAR-100 [18].
- *<CIFAR100_TEST>* is the pickled test dataset file from CIFAR-100 [18].

- *<DEST>* is the destination file to save the trained NN's weights to.

```
python project/ml/train_nn.py -f <CIFAR100_TRAIN> -t
<CIFAR100_TEST> -d <DEST>
```

Figure 15 Command for Training NN on CIFAR-100

Figure 16 shows the terminal command for running the code to train a NN on the human and vehicles samples from the surveillance dataset. In the command,

- *<CONFIG_FILE>* is a JSON file containing a single JSON object with properties specifying the parameters to be used by the program:
    - *"src_dir"* - string, directory containing directory of images. These inner directories should have same name as the integer class labels, e.g. 0 or 1.
    - *"dest"* - string, destination file path to save trained network to.
    - *"num_classes*" - int, amount of classes.
    - *"num_epochs"* - int, number of epochs to run training for.
    - *"predictionOutputFile"* - string indicating desired filepath to output prediction output to, or null for none.

```
python project/ml/train_nn_surveillance.py --config <CONFIG_FILE>
```

Figure 16 Command for Training NN on Surveillance Images

Figure 17 shows the command used for loading a NN trained on the CIFAR-100 samples and re-training the NN on the surveillance images. The arguments are similar to the analogous command for CNN discussed in Section 4.1.1, except the "arch" property should be set to "1". The "cnn_file" property for NN shares the same name as the "cnn_file" property for CNN. The properties for the configuration file are as follows:

- *"cnn_file"* - string, path to the weights of the NN trained on the CIFAR-100 samples, as an .npy file.
- *"cifar_test_data"* - string, path to the test dataset file from CIFAR-100 [18]. This is used for testing only.
- *"arch"* - string, the identifier for the NN architecture as indicated in project/ml/lasagna_nn.py. We leave this set to "1".

- *"src_dir"* - string, directory containing directory of images. These inner directories should have same name as the integer class labels.
- *"dest"* - string, destination filepath to save trained network to.
- *"num_classes*" - int, amount of classes.
- *"num_epochs"* - int, number of epochs to run training for.
- *"predictionOutputFile"* - string indicating desired filepath to output prediction output to, or null for none.

```
python project/ml/train_nn_cifar_surveillance.py --config
<CONFIG_FILE>
```

Figure 17 Command for Re-Training NN on Surveillance Images

4.1.3 Train a Support Vector Machine

The terminal command for running the code to train a SVM on the human and vehicles samples from the CIFAR-100 dataset, shown in Figure 18, is similar to the command for training a as CNN discussed in Section 4.1.1. above. The parameters are as follows:

- <*CIFAR100_TRAIN*> is the pickled training dataset file from CIFAR-100 [18].
- <*CIFAR100_TEST*> is the pickled test dataset file from CIFAR-100 [18].
- <*DEST*> is the destination file to save the trained SVM's weights to.

```
python project/ml/train_svm.py -f <CIFAR100_TRAIN> -t
<CIFAR100_TEST> -d <DEST>
```

Figure 18 Command for Training SVM on CIFAR-100 Images

Figure 19 shows the terminal command for running the code to train a NN on the human and vehicles samples from the surveillance dataset. In the command, where <*CONFIG_FILE*> is a JSON file containing a single JSON object with properties specifying the parameters to be used by the program:

- *"src_dir"* - string, directory containing directory of images. These inner directories should have same name as the integer class labels, e.g. 0 or 1.
- *"dest"* - string, destination file path to save trained SVM to.
- *"num_classes*" - int, amount of classes.

- *"predictionOutputFile"* - string indicating desired filepath to output prediction output to, or null for none.

```
python project/ml/train_svm_surveillance.py --config
<CONFIG_FILE>
```

Figure 19 Command for Training SVM on Surveillance Images

4.2 Extract Surveillance Images

To obtain images of human and vehicle samples taken directly from surveillance video footage, we used a modification of our video surveillance system. We trained a CNN to classify human and vehicle objects with reasonable accuracy, which we used for identifying the class of found objects. Our image extraction program created 32x32 pixel images based on the bounds of the detected objects and separated the images by their classifications. After we obtained the surveillance images, we manually double checked the images and corrected misclassifications while discarding any bad images. Afterwards, we trained another set of machine learning classifiers on the surveillance images and compared the accuracy to the classifiers trained on the CIFAR-100 samples.

```
python project/cv/videoscraper_main.py --config
config_scrape/conf_scraper.json
```

Figure 20 Sample Command for Extracting Surveillance Images

Figure 20 shows an example of the command we used to run the image extraction program. The program takes a configuration file, which contains a single JSON object with the following properties:

- "source" - either string, video file source; or integer, the webcam index number, usually 0
- "cnn_file" - string, the .npy file containing the weights of the trained CNN.
- "arch" - string, the identifier for the CNN architecture as indicated in project/ml/lasagna_cnn.py. We leave this set to "ex".
- "empty_first_frame" - boolean, whether the first frame has no foreground objects or not.
- "obj_classes" - list of strings, the names of the classes in order of their integer labels.

- "cnoise_top" - integer, minimum pixel height of an object positioned at the top of the image. Objects shorter than the threshold shall be considered noise.
- "cnoise_bottom" - integer, minimum pixel height of an object positioned at the bottom of the image. Objects shorter than the threshold shall be considered noise.
- "scrape_dest" - string, path to a directory in which to write the extracted images to. The images will be placed in another directory within that directory, named after the numeric label for the corresponding class.
- "scrape_prefix" - string, a prefix to use when naming the images.
- "scrape_override_class" - integer or null, the class label to use as the classification of the images. If not null, this setting will override the video surveillance system's classification.

An example configuration file is shown in Figure 21.

```
{
    "source" : "path/to/video.mp4",
    "cnn_file" : "path/to/trained_models/cifar_cnn.npy",
    "arch" : "ex",
    "empty_first_frame" : false,
    "obj_classes": ["Vehicle", "Human"],
    "cnoise_top" : 10,
    "cnoise_bottom" : 70,
    "scrape_dest": "path/to/datasets/scraped_imgs",
    "scrape_prefix": "video_",
    "scrape_override_class": null
}
```

Figure 21 Sample Surveillance Image Extraction Configuration File

4.3 Run the Video Surveillance System

Our video surveillance system can be run from the command line with either a video file source or a webcam source. Figure 22 shows the command to run the system, where <CONFIG_FILE> is a JSON file specifying the configuration settings. The configuration file contains a single JSON object with the following properties:

- "source" - either string, video file source; or integer, the webcam index number, usually 0
- "cnn_file" - string, the .npy file containing the weights of the trained CNN.
- "arch" - string, the identifier for the CNN architecture as indicated in project/ml/lasagna_cnn.py. We leave this set to "ex".
- "empty_first_frame" - boolean, whether the first frame has no foreground objects or not.
- "obj_classes" - list of strings, the names of the classes in order of their integer labels.
- "cnoise_top" - integer, minimum pixel height of an object positioned at the top of the image. Objects shorter than the threshold shall be considered noise.
- "cnoise_bottom" - integer, minimum pixel height of an object positioned at the bottom of the image. Objects shorter than the threshold shall be considered noise.

```
python project/cv/peopledetector_main.py --config <CONFIG_FILE>
```

Figure 22 Command for Running Video Surveillance System

5 Results

This section summarizes the results obtained in our classifier training and testing stages. To quantify the results of the CNN, we used an accuracy measure and the loss function and an accuracy measure. We also measured the validation accuracy of the MLP and SVM approaches. Accuracy is the percent of correct predictions made for a particular dataset; concretely, accuracy is given by the total number of correct predictions divided by the total number of elements in the dataset. Loss is the average of the cross entropy between all predictions and targets in the dataset, given by the following formula [36] [21]:

$$\text{Loss}(w) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j} t_{i,j}\log\left(p_{i,j}\right)$$

The value w is the weight vector containing N samples, $t_{i,j}$ is the target value corresponding to the ith sample and the jth model, and $p_{i,j}$ is the predicted value corresponding to the ith sample and the jth model.

5.1 Convolutional Neural Network

By training the CNN for for more than 100 epochs, the model reached a high level of accuracy. We ran the CNN algorithm for 150 epochs and achieved an accuracy of 90.6% on our test set. On our PC the training took more than 3 hours, whereas on the VM training lasted for only 1.21 hours. We gathered two measures on the training process for the CIFAR-100 images: loss and accuracy. These measures were gathered from both the training and the validation process and are summarized in Figure 23, Figure 24, and Figure 25. We observe that validation accuracy begins to plateau at around epoch 90, suggesting that we could have stopped the training process early. Our results for training on CIFAR-100 images, training on surveillance images, pre-training on CIFAR-100 and re-training on surveillance images are summarized in Table 6. We observe that the validation accuracy of the CNN trained on the surveillance images and validated on surveillance images scored nearly 100 percent; we attribute this high accuracy due to the fact that the surveillance image dataset contained multiple images of the same objects at different angles.

Table 6 CNN Validation Accuracies

Training Images	**Epochs**	**Validation Accuracy on CIFAR-100 Images**	**Validation Accuracy on Surveillance Images**
CIFAR-100	150	90.6%	83.20%
Surveillance	90	64.00%	99.60%
Pre-trained on CIFAR-100, re-trained on surveillance	90	79.00%	98.40%

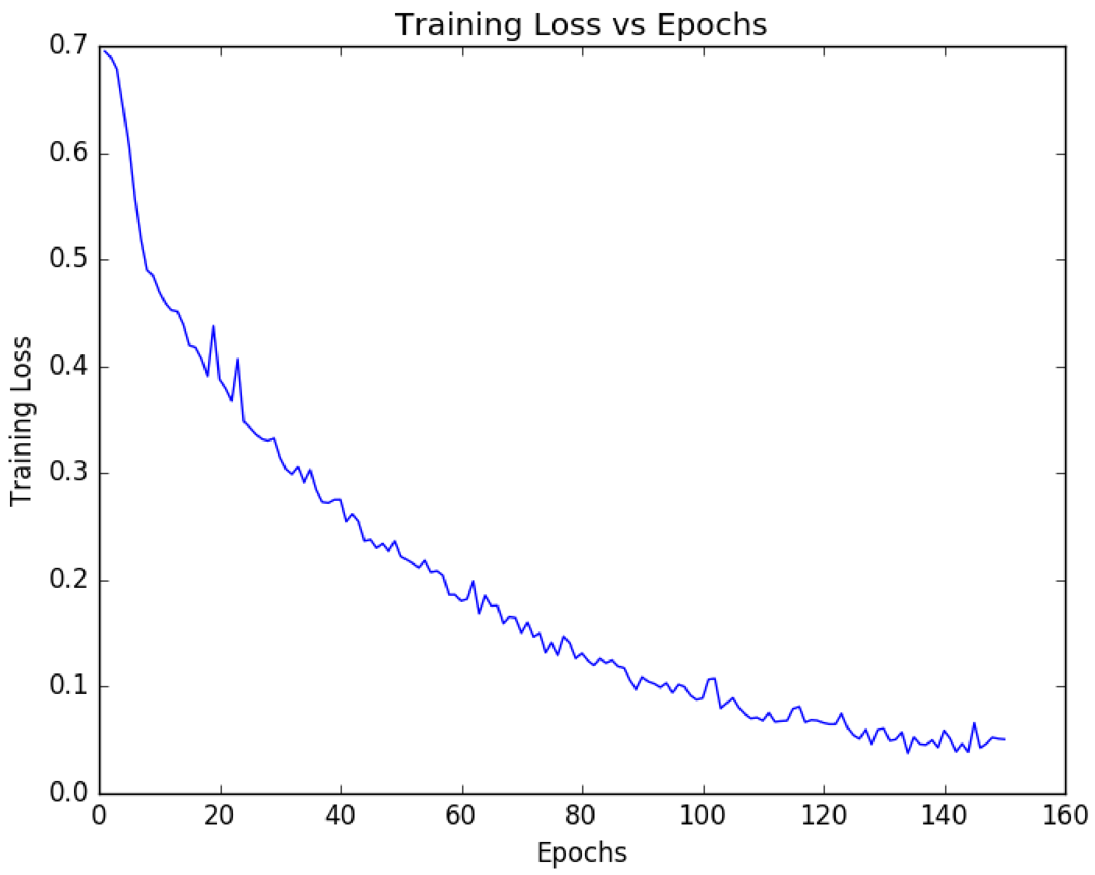


Figure 23 CNN 1 Training Loss vs. Epochs

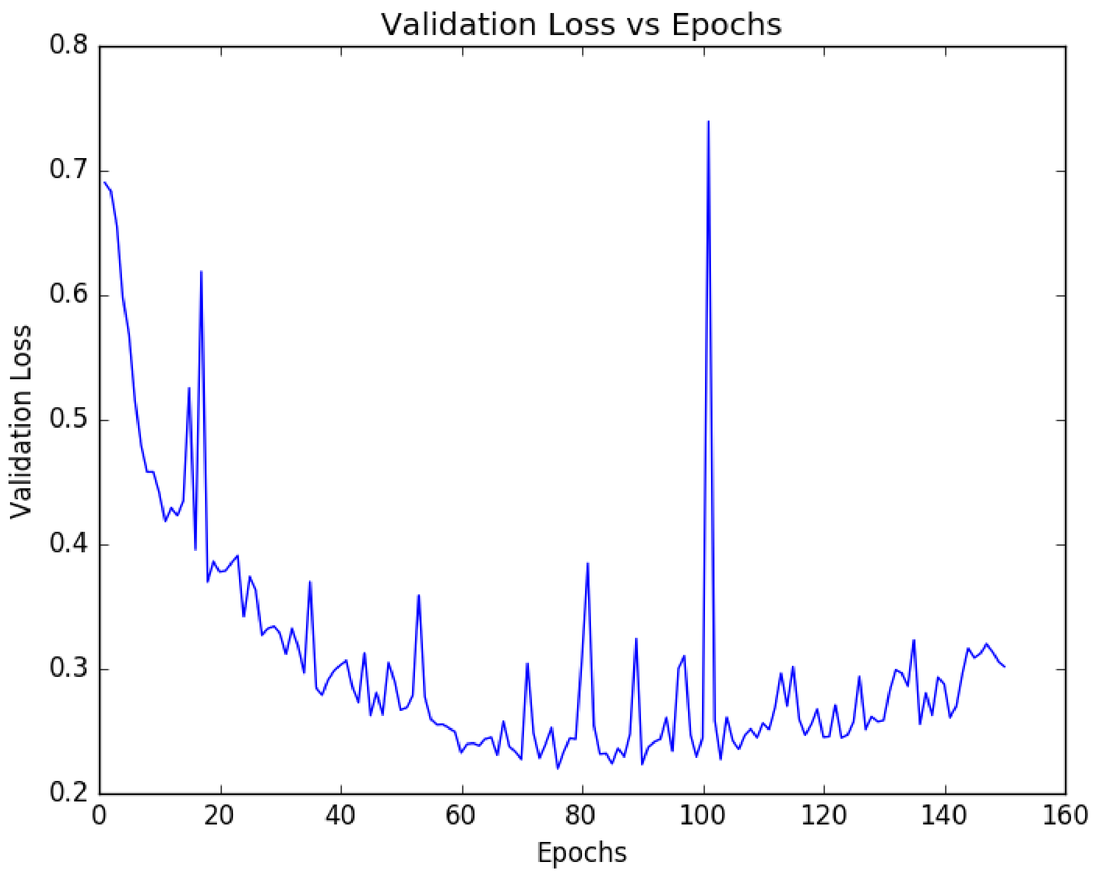


Figure 24 CNN 1 Validation Loss vs. Epochs

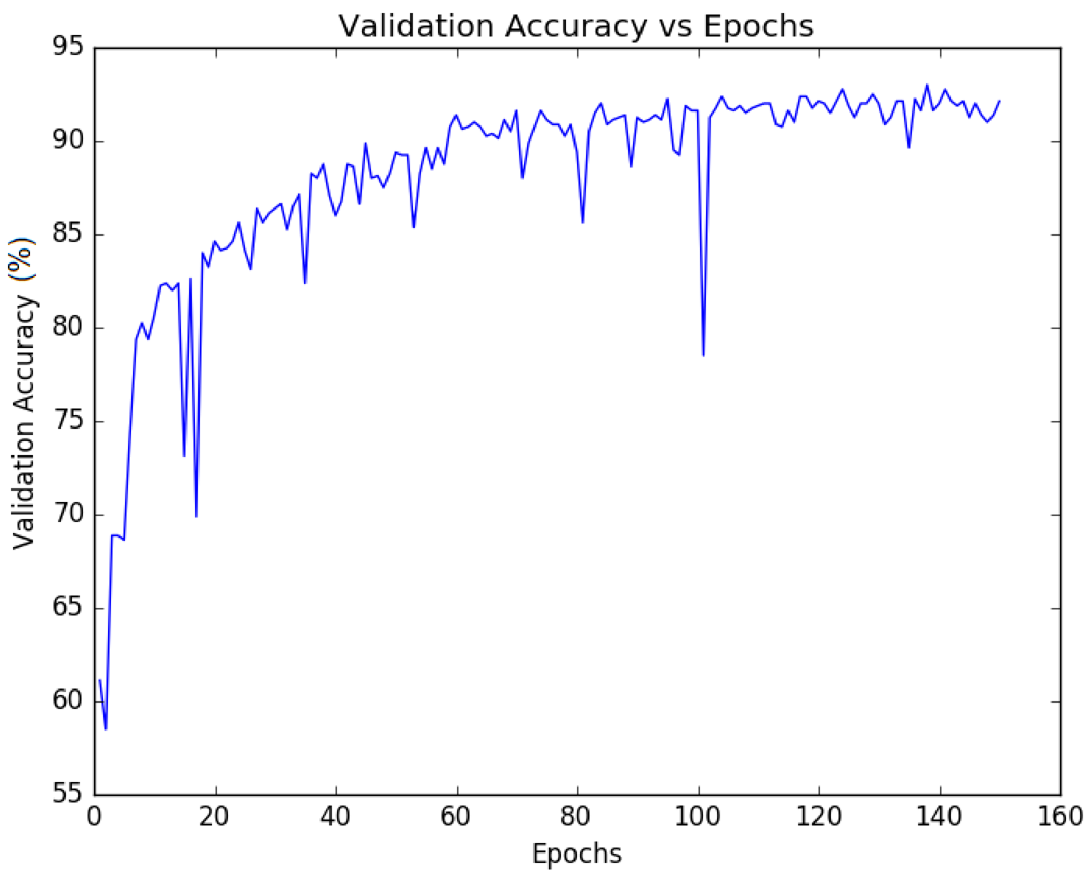


Figure 25 CNN 1 Validation Accuracy vs. Epochs

We inspected the feature maps of the CNN to check that features were being learned. The learned feature maps of the first and second convolutional layers are shown in Figure 26 and Figure 27, respectively. The non-random structure of the feature maps suggests that the CNN is learning information about the spatial structure of the images [8].

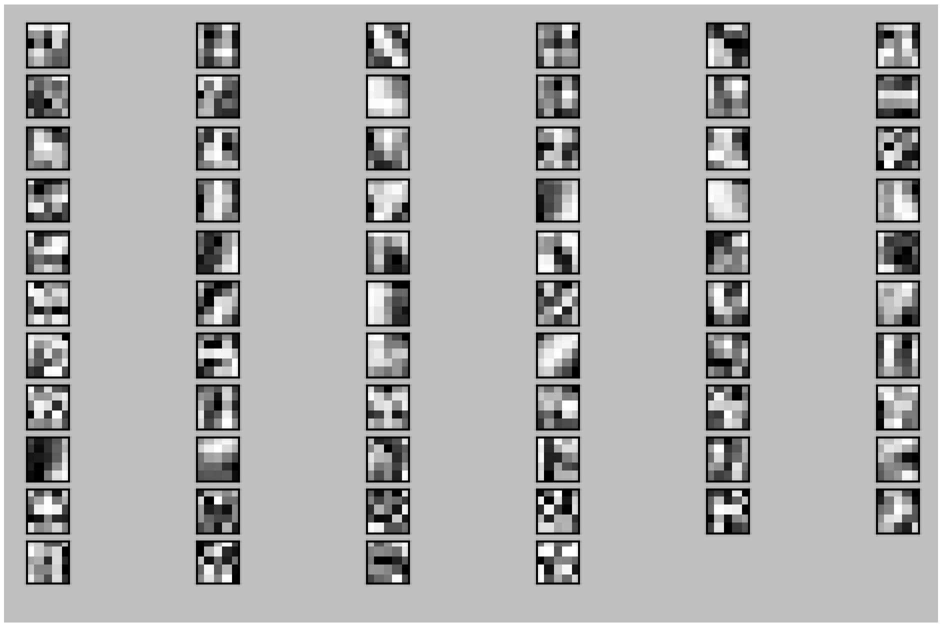


Figure 26 CNN 1 Feature Maps of First Convolutional Layer

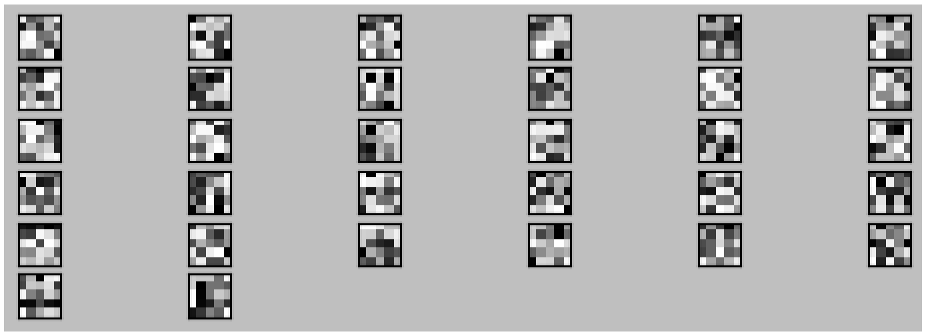


Figure 27 CNN 2 Feature Maps of Second Convolutional Layer

5.2 Multi-layer Perceptron with Histogram of Oriented Gradients

We ran the training process using CIFAR-100 images for 600 epochs with a learning rate of 0.00075. On our PC, this took 24.71 seconds to train, whereas on the VM it took 16.41 seconds. The validation accuracy achieved was 81.25%. Our results for the cases of training on CIFAR-100, training on surveillance, and pre-training on CIFAR-100 and retraining on surveillance are summarized in Table 7. Like with the results of training the CNN, the NN with HOG trained on the surveillance images achieved much lower accuracy when validated using the CIFAR-100 image set instead of using the surveillance image set. We observe that this approach always had less than 50 percent validation accuracy when the validation set was from an image dataset that the learning algorithm did not train on. In the case of pre-training on CIFAR-100 images and re-trianing on the surveillance images, validation accuracy was roughly the same as when the NN was trained on surveillance images.

Table 7 MLP+HOG Validation Accuracies

Training Images	**Epochs**	**Validation Accuracy on CIFAR-100 Images**	**Validation Accuracy on Surveillance Images**
CIFAR-100	600	81.25%	20.0%
Surveillance	90	45.60%	78.38%
Pre-trained on CIFAR-100, re-trained on surveillance	90	45.40%	80.80%

5.3 Support Vector Machines with Histogram of Oriented Gradients

The SVM approach achieved a validation accuracy of 77.0%. Training completed in 0.688 seconds on the PC and took 0.365 seconds on the VM. The validation accuracies for training on CIFAR-100 and training on surveillance images are summarized in Table 8. The SVM approach had strange results when the surveillance images were involved.

Table 8 SVM+HOG Validation Accuracies

Training Images	**Validation Accuracy on CIFAR-100 Images**	**Validation Accuracy on Surveillance Images**
CIFAR-100	77.0%	0.75%
Surveillance	3.5%	26.13%

6 Analysis

6.1 Training Results

We found that the CNN approach had a significantly higher validation accuracy (90.6 %) than the MLP and SVM approaches (81.25 % and 77.0 %, respectively). This result motivated our decision to use the trained CNN for classification in our object detection part of our project.

When training and validating using the surveillance image datasets, we found that validation accuracies tended to be similar or better than when we trained and validated using the CIFAR-

100 dataset. When validating on an image set different from the training image set, the accuracies were always worse.

The SVM approach had unusual results, as summarized in Table 8. A validation accuracy of around 50 percent would have meant that the algorithm was making predictions as accurate as random guessing; the results of the SVM approach, however, were significantly below 50 percent. This meant that if we swapped all of the classifications made by the trained SVM, the validation accuracies would have been competitive with the CNN and MLP approaches. A possible explanation for the strange results could have been that the surveillance images were mislabeled when passed to the SVM.

6.2 System Results

We found our system to be effective under only certain situations. Due to the background subtraction technique used, our system required that the camera be static and the lighting be consistent, otherwise moving objects may not correctly be detected. Furthermore, we found that moving objects with large regions of flat colors and flat textures, such as the surface of a single-colored van, resulted in poor motion detection. The background subtraction technique that the system used also had the side effect of considering nearby moving objects to be considered as being the same entity. The system has no means of separating conjoined objects; these objects may not always belong to the same class, resulting in situations where an object may hide itself within another moving object.

The object tracking capabilities of the system was constrained by several factors. The feature detection technique used did not seem to perform well when the video quality was not smooth and when objects moved too fast. In addition, multiple objects that moved close to one another tended to become merged into a single tracked entity. The color similarity technique required that background objects not share similar colors to moving objects.

7 Conclusion

We found that convolutional neural networks had the highest accuracy amongst the machine learning algorithms that we tested. Especially when trained on surveillance images, the CNN approach performed with nearly 100 percent accuracy. The surveillance image validation dataset, however, was composed of mostly similar objects, meaning that the CNN could have been overfitted on those images. Thus, further tests need to be carried out on surveillance images that do not come from the same sources as the ones used for training.

Although our video surveillance system had high classification accuracy, the detection and tracking capabilities of the system were limited. Our video surveillance system had many weaknesses, which included the requirement that the camera be static and that objects not have flat textures. Many of the weaknesses of the system could possibly be mitigated or avoided if an alternative motion detection technique to Gaussian Mixture-based Background Segmentation was used. The tracking system did not perform consistently in a crowded scene with many objects moving close to one another.

Our possible future work may include improving our object detection strategy and performing further tests on the training process. It would be of interest to use an object detection technique that allowed for a moving camera. To improve upon our CNN training results, we would test the trained CNN on surveillance images taken from videos that do not contain images of objects used in the training stage.

8 References

[1] M. H. Sedky, M. Moniri and C. C. Chibelushi, "Classification of smart video surveillance systems for commercial applications," in *IEEE Conference on Advanced Video and Signal Based Surveillance, 2005.*, Como, 2005.

[2] G. Harit and S. Chaudhury, "Video Shot Characterization Using Principles of Perceptual Prominence and Perceptual Grouping in Spatio–Temporal Domain," in *IEEE Transactions on Circuits and Systems for Video Technology*, 2007.

[3] K. Liu, B. Liu, C. Chen and C. W. Chen, "A hierarchical anti-occlusion tracking algorithm based on DMPF and ORB," in *Image Processing (ICIP), 2015 IEEE International Conference on*, Quebec City, QC, 2015.

[4] K. T. K. Teo, R. K. Y. Chin, N. S. V. K. Rao, F. Wong and W. L. Khong, "Vehicle Tracking Using Particle Filter for Parking Management System," in *Artificial Intelligence with Applications in Engineering and Technology (ICAIET), 2014 4th International Conference on*, Kota Kinabalu, 2014.

[5] T. Kirubarajan, Y. Bar-Shalom, K. R. Pattipati and L. M. Loew, "Interacting segmentation

and tracking of overlapping objects from an image sequence," in *Decision and Control, 1997., Proceedings of the 36th IEEE Conference on*, San Diego, 1997.

[6] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors.," 3 July 2012. [Online]. Available: http://arxiv.org/abs/1207.0580. [Accessed 15 November 2015].

[7] P.-N. Tan, M. Steinbach and V. Kumar, Introduction to Data Mining, Pearson, 2005.

[8] M. A. Nielsen, Neural Networks and Deep Learning, Determination Press, 2015.

[9] A. Ng, J. Ngiam, C. Y. Foo, Y. Mai, C. Suen, A. Coates, A. Maas, A. Hannun, B. Huval, T. Wang and S. Tandon, "UFLDL Tutorial," 7 January 2015. [Online]. Available: http://ufldl.stanford.edu/tutorial/. [Accessed 15 November 2015].

[10] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, San Diego, 2005.

[11] C. McCormick, "HOG Person Detector Tutorial," 9 May 2013. [Online]. Available: chrisjmccormick.wordpress.com/2013/05/09/hog-person-detector-tutorial/. [Accessed 15 November 2015].

[12] J. Fan, W. Xu, Y. Wu and Y. Gong, "Human Tracking Using Convolutional Neural Networks," in *Neural Networks, IEEE Transactions on*, 2010.

[13] I. Arel, D. Rose and T. Karnowski, "Deep Machine Learning - A New Frontier in Artificial Intelligence Research [Research Frontier]," *Computational Intelligence Magazine, IEEE,* vol. 5, no. 4, pp. 13-18, 2010.

[14] Y. LeCun, K. Kavukcuoglu and C. Farabet, "Convolutional Networks and Applications in Vision," *Computational Intelligence Magazine, IEEE* , vol. 5, no. 4, pp. 13-18, 2010.

[15] A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, Curran Associates, Inc., 2012, pp. 1097-1105.

[16] Microsoft, "Virtual Machines Pricing," [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/virtual-machines/#Linux. [Accessed 29 September 2015].

[17] Microsoft, "What is Azure—the Best Cloud Service from Microsoft | Microsoft Azure," Microsoft, 2016. [Online]. Available: https://azure.microsoft.com/en-us/. [Accessed 21 May 2016].

[18] Opencv dev team, "Welcome to opencv documentation!," Itseez, 10 November 2014. [Online]. Available: http://docs.opencv.org/3.0-beta/index.html. [Accessed 1 May 2015].

[19] Itseez, "OpenCV," Itseez, 2015. [Online]. Available: http://opencv.org/. [Accessed 15 November 2015].

[20] T. Kang, "Using Neural Networks for Image Classification," 18 May 2015. [Online]. Available: http://scholarworks.sjsu.edu/etd_projects/395. [Accessed 15 November 2015].

[21] A. Krizhevsky, "The CIFAR-10 and CIFAR-100 datasets," 12 December 2013. [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html. [Accessed 2015 15 November].

[22] Lasagne contributors, "Welcome to Lasagne," Lasagne contributors, 2015. [Online]. Available: http://lasagne.readthedocs.org/en/latest/index.html. [Accessed 15 November 2015].

[23] LISA lab, "Welcome -- Theano 0.7 documentation," LISA lab, 2015. [Online]. Available: http://deeplearning.net/software/theano/. [Accessed 15 November 2015].

[24] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley and Y. Bengio, *Theano: new features and speed improvements*, NIPS, 2012.

[25] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley and Y. Bengio, "Theano: A CPU and GPU Math Expression Compiler," in *Proceedings of the Python for Scientific Computing Conference (SciPy) 2010*, Austin, 2010.

[26] Numpy developers, "Numpy," Numpy developers, 23 October 2013. [Online]. Available: http://www.numpy.org/. [Accessed 15 November 2015].

[27] P. Raybaut, "WinPython," 29 October 2015. [Online]. Available: https://winpython.github.io/. [Accessed 29 November 2015].

[28] Lasagne contributors, "Installation -- Lasagne 0.2.dev1 documentation," Lasagne contributors, 2015. [Online]. Available:

http://lasagne.readthedocs.org/en/latest/user/installation.html. [Accessed 30 November 2015].

[29] Waqas, "Why cv2.so missing after opencv installed?," 14 April 2013. [Online]. Available: http://stackoverflow.com/questions/15790501/why-cv2-so-missing-after-opencv-installed/16003545#16003545. [Accessed 30 November 2015].

[30] J. Timmerman, "cmake finds wrong python libs," 21 March 2012. [Online]. Available: http://stackoverflow.com/questions/7660001/cmake-finds-wrong-python-libs/9810796#9810796. [Accessed 30 November 2015].

[31] A. Rosebrock, "Install OpenCV 3.0 and Python 2.7+ on Ubuntu," 22 June 2015. [Online]. Available: http://www.pyimagesearch.com/2015/06/22/install-opencv-3-0-and-python-2-7-on-ubuntu/. [Accessed 30 November 2015].

[32] Rahul, "How to Install Python 2.7.10 on Ubuntu & LinuxMint," 30 July 2015. [Online]. Available: tecadmin.net/install-python-2-7-on-ubuntu-and-linuxmint/. [Accessed 30 November 2015].

[33] A. Dundar, "Convolutional Neural Networks," 13 January 2013. [Online]. Available: https://www.youtube.com/watch?v=n6hpQwq7Inw. [Accessed 15 November 2015].

[34] C. McCormick, "Gradient Vectors," 7 May 2013. [Online]. Available: https://chrisjmccormick.wordpress.com/2013/05/07/gradient-vectors/. [Accessed 15 November 2015].

[35] Lasagne contributors, "Tutorial," 2015. [Online]. Available: http://lasagne.readthedocs.org/en/latest/user/tutorial.html. [Accessed 12 2 2016].

[36] A. Mordvintsev and A. K, "OCR of Hand-written Data using SVM," 26 January 2016. [Online]. Available: http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_ml/py_svm/py_svm_opencv/py_svm_opencv.html. [Accessed 13 February 2016].

[37] K. Murphy, Machine Learning: A Probabilistic Perspective, MIT, 2012.

[38] A. Vedaldi and A. Zisserman, "Efficient Additive Kernels via Explicit Feature Maps," *Pattern Analysis and Machine Intelligence, IEEE Transactions on,* vol. 34, no. 3, pp. 480-492, 2012.

[39] Y. Amit and P. Felzenszwalb, "Object Detection," in *Computer Vision, A Reference Guide,* Springer US, 2014, pp. 537-542.

[40] RockTheStar, "What is the correct architecture for convolutional neural network?," 17 October 2014. [Online]. Available: http://stackoverflow.com/questions/26434325/what-is-the-correct-architecture-for-convolutional-neural-network. [Accessed 15 November 2015].

[41] S. McCann and J. Reesman, "Object Detection using Convolutional Neural Networks," Stanford University, 2013.

[42] J. Mairal, Z. H. Piotr Koniusz and C. Schmid, "Convolutional Kernel Networks," in *Advances in Neural Information Processing Systems (NIPS)*, Montreal, 2014.

[43] LISA lab, "Convolutional Neural Networks (LeNet)," n.d.. [Online]. Available: http://deeplearning.net/tutorial/lenet.html. [Accessed 15 November 2015].

[44] G. E. Hinton, S. Osindero and Y.-W. Teh, "A Fast Learning Algorithm for Deep Belief Nets," *Neural Computation*, vol. 18, no. 7, pp. 1527-1554, 2006.

[45] Y. Kim and T. Moon, "Human Detection and Activity Classification Based on Micro-Doppler Signatures Using Deep Convolutional Neural Networks," *Geoscience and Remote Sensing Letters, IEEE,* vol. PP, no. 99, pp. 1-5, 2015.

[46] T. Rui, J.-c. Fei, P. Cui, Y. Zhou and H.-s. Fang, "Head detection based on convolutional neural network with multi-stage weighted feature," in *Signal and Information Processing (ChinaSIP), 2015 IEEE China Summit and International Conference on* , Chengdu, 2015.

[47] J. Wang, Q. Hou, N. Liu and S. Zhang, "Model of Human Visual Cortex Inspired Computational Models for Visual Recognition," in *Multimedia Big Data (BigMM), 2015 IEEE International Conference on*, Beijing, 2015.

[48] S. Kothiya and K. Mistree, "A review on real time object tracking in video sequences," in *Electrical, Electronics, Signals, Communication and Optimization (EESCO), 2015 International Conference on* , Visakhapatnam, 2015.

[49] Continuum Analytics, "Download Anaconda now!," Continuum Analytics, Inc., 2015. [Online]. Available: https://www.continuum.io/downloads. [Accessed 14 November 2015].

9 Appendix

9.1 Source Code

The source code for our project is publicly available at github.com/tn9900/cs298.