A DEEP LEARNING APPROACH TO COARSE ROBOT LOCALIZATION

by

LUC ALEXANDRE BETTAIEB

Dr. Wyatt S. Newman
Thesis Advisor

Submitted in partial fulfillment of the requirements
for the degree of Master of Science

Department of Electrical Engineering and Computer Science
CASE WESTERN RESERVE UNIVERSITY

August, 2017

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis of

Luc Alexandre Bettaieb

candidate for the degree of Master of Science.

Committee Chair

Wyatt S. Newman

Committee Member

M. Cenk Çavuşoğlu

Committee Member

Gregory S. Lee

Date of Defense

April 18th, 2017

Table of Contents

List of Tables

List of Figures

A Deep Learning Approach to Coarse Robot Localization

Abstract

by

LUC ALEXANDRE BETTAIEB

This thesis explores the use of deep learning for robot localization with applications in re-localizing a mislocalized robot. Seed values for a localization algorithm are assigned based on the interpretation of images. A deep neural network was trained on images acquired in and associated with named regions. In application, the neural net was used to recognize a region based on camera input. By recognizing regions from the camera, the robot can be localized grossly, and subsequently refined with existing techniques. Explorations into different deep neural network topologies and solver types are discussed. A process for gathering training data, training the classifier, and deployment through a robot operating system (ROS) package is provided.

1 Introduction

Mobile robots and mobile manipulators are being increasingly employed in applications such as automated storage and retrieval, kitting, and warehouse logistics. In such applications, it is important that the robot has a sufficiently precise estimate of its pose in the environment. Typically, this is enabled by map-making and matching against planar laser rangefinder (LIDAR) scans. However, many such environments can have ambiguous or featureless regions, as interpreted by LIDAR. To address this issue, one could consider the interpretation of richer sensing, such as using color cameras. A human may be able to easily interpret camera images to determine a location within an environment. However, even though humans are able to capitalize on the richness of information within these types of images, traditional techniques used in robotics are unable to compete. Interpreting these images using conventional means may be a daunting task.

Recent progress in deep learning [1] has shown that it is now possible for computers to interpret color images and classify them in categories with dramatically better accuracy than in the past. This thesis explores the use of deep learning to recognize labeled regions based on camera input. With success in this approach, it would be possible to localize a robot coarsely (by region), and with a suitable seed value for estimated pose, refine the pose with the interpretation of LIDAR.

A robot can become confused in its localization when the environment is ambiguous as sensed by planar LIDAR – as well as in situations where the robot is moved in an un-trackable way while its localization algorithms are running. These problems are similar as they lead to the robot becoming mislocalized, and this situation is colloquially referred to as the “kidnapped robot problem”.

While this term may bring to mind images of black vans and questionable individuals, the issue is thankfully more benign in nature. When the robot’s sensors that enable it to know its location are covered up, disconnected, or disabled in any way – many algorithms that are designed to keep track of the robot’s position will fail.

As an example, picture a small, lightweight mobile robot navigating its environment. Suddenly, network communication between the wheel encoders that help track its position, and the laser scanner that helps determine what is around it cease communication with the main controller. Unfortunately, the low-level controller is also affected by this issue and never knows to stop commanding the motors to travel forward.

After communication is re-established, the algorithm that was keeping track of the robot’s position assumes that the next packet of information from the wheel encoders is sequential to the last, and the latest laser scan simply looks significantly different. This malicious combination causes the algorithm to fail, as the continuity the sensors previously kept between the algorithm’s model it used to track the robot and reality was broken.

When mobile robots that are deployed in real-life scenarios, such as those meant for industrial logistics, become mislocalized, which is what happens when a robot is kidnapped, disaster ensues. If the robot is unable to know exactly where it is, it would be unsafe for it to operate. Therefore, the robot must be taken offline and re-localized; potentially manually. This will cause downtime that will ultimately lead to loss of profits.

Depending on the algorithm, different techniques may be employed to remedy the situation. A common algorithm for localizing the robot in its environment, Monte Carlo Localization (MCL) [2], is able to recover from kidnapping via a global initialization technique. In this technique, hundreds of initial hypotheses of where the robot may exist are evenly distributed across free space in the map. As the algorithm progresses, the hope is that these hypotheses converge to the true position of the robot. Unfortunately, in practice, this is not always the case. When using the popular robotics middleware platform, Robot Operating System (ROS) [3] implementation of MCL, AMCL [4] (Adaptive Monte Carlo Localization), convergence is not guaranteed, and if it does occur, can often take unacceptably long times. This is because the initial search space for the robot is much too broad. If the search space could be reduced to a smaller, more manageable size, the algorithm would have an inherently easier time at re-localizing the robot. This motivates the use of image interpretation to help seed localization to lead to a coarse localization estimate.

To reduce the size of this search space, a deep convolutional neural network classifier is trained on training data consisting of images collected from separate environments in a known map space. This type of classifier belongs to a category of machine learning algorithms called deep learning. Deep learning algorithms are artificial neural networks with topologies that can range anywhere between five layers of neuronal connections, to hundreds. Their power has been proven by setting records at various machine learning competitions [5][6]. The classifier was trained using the deep learning library Caffe [7] via NVidia's deep learning training system, DIGITS [8].

After training, the classifier was then deployed in a node, a program designed for use with ROS, using a library written to interface with the Caffe C++ API. The existing AMCL package was modified to be able to place particles in the classified areas using commands from the ROS service protocol. These services were able to be triggered by a separate node, which continually checked for the current classification, assigned it to a state variable, then called the required service. This action can be triggered by a separate ROS service. All of these steps work together to help provide a better solution to the kidnapped robot problem.

The important contribution that this thesis provides is the machine learning classifier and general approach to narrowing down the global initialization technique to effectively solve the kidnapped robot problem. By being able to classify the room or area within which the robot is located, the kidnapped robot problem is able to be more effectively solved.

1.1 History and Background

Mobile robots have been the dream of mankind for generations. In ancient Greece, Hephaestus, the god of the forge, created the bronze giant Talos that was used to defend the city of Crete from pirates and invaders. Other such legends existed, describing beings of metal created by gods and men alike to assist mankind with all sorts of tasks. However, it was not until 1920 that the term “robot” became a reality when Karel Čapek wrote his play *Rossumovi Univerzální Roboti*, or “Rossum’s Universal Robots”.

Even though Čapek’s robots would most likely be classified as clones by modern standards, they left a lasting legacy on the robotics community by coining the term ‘robot’. Taken from the Slavic word *robota*, meaning ‘forced laborer’, the word stuck around to become the most common descriptor of the autonomous machines that society has become accustomed to.

The first programmable robot was designed in 1956 by engineer George Devol and businessman Joseph Engelberger. This robot, UNIMATE, was introduced into the American workforce in 1961 [9] as the first industrial robot arm and could be programmed on a magnetic disk to perform a myriad of tasks such as welding, drilling, and spraying. In 1966, the first mobile robot from Stanford Research Institute (now SRI International) was built. Called “Shakey” [10], the robot ran on code from a mainframe computer with sensor data and motor commands being sent over a radio link.

The type of algorithms that the Stanford researchers developed to run on Shakey are the groundwork of modern day robotics research. Since Shakey was introduced, there have been great strides in the field of mobile robotics. Robots such as the Adept Pioneer[1], the iRobot Roomba[2], Willow Garage’s PR2[3], and Fetch and Freight[4] from Fetch Robotics have brought the state of the art a great distance. However, even modern day robots still have to deal with problems like the ones Shakey did.

All mobile robots have the common problem of needing to know their position within their environment. This is called the localization problem, and there exist several techniques to solve it. A modern way to address this is through software packages written for the Robot Operating System (ROS) [3]. Using ROS software packages allows

[1] http://www.mobilerobots.com/ResearchRobots/PioneerP3DX.aspx
[2] http://www.irobot.com/For-the-Home/Vacuuming/Roomba.aspx
[3] http://www.willowgarage.com/pages/pr2/overview
[4] http://fetchrobotics.com/platforms-research-development/

for the easy development of new robotic systems and has become a standard in the robotics community.

1.2 Robot Operating System (ROS)

It is certainly compelling to apply deep learning to robots. However, robotic systems are increasingly complex, and it is important that the use of deep learning techniques be compatible with existing architectures. Towards this end, the approach taken here will be to encapsulate a trained neural network within a ROS node. In this manner, a trained network can be integrated within a complex robot software architecture. Such encapsulation is a contribution of this thesis.

Many open-source software packages exist within the Robot Operating System (ROS) ecosystem, all offering solutions to common robotics problems. ROS is an open-source robot middleware platform that provides a layer for asynchronous communication among programs performing a myriad of different tasks [3]. For example, when using ROS, a computer that is on the robot and connected directly to the robot's sensors may be running the "master node" of the system. On the master, programs may run to act as sensor drivers, taking in raw values over different communication protocols and converting them for use by other ROS programs, or nodes. Other nodes may subscribe to data that is published over the ROS master and transmitted via UDP.

In this fashion, a node might subscribe to information being published by another node and use that information for its own purposes. As an example, one node might contain code to handle communications with a serial interface to which motor encoders are attached. Once the encoder ticks are acquired by this node, they may be published and advertised on the ROS master. In this style, another node could subscribe to this data and use the raw encoder information to calculate the robot's odometry: the robot's velocity and an estimate of position from an integral of velocity over time.

ROS has now become the most popular robotics middleware platform, allowing thousands of researchers and industry professionals alike to build complex robots without having to worry about much of the software overhead required. Pre-existing software packages are available for download directly to one's robot. With some configuration, one can, for example, convert the equivalent of a remote-controlled car to a truly autonomous vehicle within hours. Successful commercial deployments of ROS-enabled robots include Fetch and Freight from Fetch Robotics [11], Relay from Savioke[5], Kuri from Mayfield Robotics[6], and Baxter[7] and Sawyer[8] from Rethink Robotics.

While ROS does a phenomenal job at getting one started writing software for robots, there exist fundamental problems within robotics research that have not been entirely solved. A robot must be able to make a map of its environment using its sensors – the mapping problem. Furthermore, a robot must be able to locate, or localize, itself within that map – the localization problem [12]. Doing these operations together is yet another problem called simultaneous localization and mapping, commonly referred to as SLAM. Once a robot is able to localize itself successfully in its environment, more advanced processes can take place such as path and motion planning.

1.3 Robot Localization

Robot localization is a fundamental issue within mobile robotics. Without the robot knowing where it is, it is difficult for it to do anything useful. The problem of

[5] http://www.savioke.com/
[6] https://www.heykuri.com/
[7] http://www.rethinkrobotics.com/baxter/
[8] http://www.rethinkrobotics.com/sawyer/

localization, at its core, is a state estimation problem. The state of the robot is what is being estimated and its variables correspond to the robot's pose in 2 or 3-D space. A ground-based mobile robot is usually localized within a 2D map, so its pose is an $< X, Y, \theta >$ position, or transform, with respect to the frame of the map. Other robots, such as aerial drones and underwater robots, may have their state vector extended to 6 dimensions—three displacements and three orientation variables.

1.3.1 Odometry

To localize a robot within a 2-D environment, two transforms are needed. The first transform is the map to the odometry frame and the second is from the odometry frame to the base link of the robot itself. Odometry is an estimate of the robot's pose and velocity estimated from sensed displacement. Wheel measurements are usually made by a series of sensors called encoders that are attached the drive train of the robot. A robot's pose in the "odometry frame" is calculated through these measurements, and is defined from wherever it first began its odometric computations.

In order to keep track of the motion of the robot, a mathematical model is needed. The simplest motion model is the differential drive model, in which two wheels are placed at a fixed distance in the center of a circular robot base. The encoders are then used to estimate the rotation of each wheel. Using this information as well as the radii of the wheels, the lateral and longitudinal distance traveled by the robot may be estimated.

While this method is useful for estimating the position of the robot, it is prone to a tremendous accumulation of error because of intrinsic properties in the encoders, wheel backlash, wheel slip, and an imperfect model of the robot itself. In order to correct for much of this error, the map to odometry frame transform is defined.

When using ROS, the estimate of the pose with respect to the odometry frame is considered to be accurate, whereas the odometry frame is considered to drift in space. By using a global localization algorithm, one attempts to estimate the pose of the robot's base link frame with respect to a map frame. Then, by computing the transformation from the map frame to the odometry frame, the locally accurate pose estimate provided by the odometry may be corrected for drift. The end result is an estimate of the robot's pose in the map frame, together with fast and smooth velocity and incremental pose updates from the odometry computations.

1.3.2 Global Localization

There are two "classic" techniques used to globally localize a robot in its mapped environment. The first is called grid localization, a technique that uses a histogram filter to represent the posterior belief of the robot's position. Depending on the grid size used, this algorithm can become computationally intractable for precise measurements [12]. Due to this issue, the more common approach to global localization is to use a technique called Monte Carlo localization (MCL) [2].

MCL uses a sampling-based approach called a particle filter to continually estimate and re-estimate the position of the robot. This technique has been highly popularized due to its success in real-world deployment and due to the Robot Operating System (ROS) software package called AMCL [4], or adaptive Monte Carlo localization.

Traditional MCL works by placing a fixed number of "particles" that function as hypotheses of the robot's pose around its starting pose and waiting for the robot to make its first move. Once the robot has moved, it moves all of the particles in a similar fashion as the robot, with some error added to account for the possibility of odometric error.

MCL models the robot's movement through a continuous stream of discrete motions that attempt to keep track of the robot's actual motion. For a differential drive robot, the motion model consists of three primitive movements, an initial rotation, a displacement, and a final rotation. These three movements, applied over a small window, are able to accurately model the differential drive robot.

After the particles have all moved with error, a sensor measurement is taken from the robot and each particle is tested to see if that measurement is in agreement with a simulated measurement from the particle. A helpful way to think about this step is with robots that use laser range scanners – a common sensor in mobile robotics. When the sensor measurement comes back, its data will correspond to contours of parts of the map. If that measurement and corresponding base-to-laser transform is then transformed based on the hypothetical robot pose from each particle, an error can be determined by scan-matching the transformed scan relative to expectation from the map. This operation creates a criterion by which each particle can be scored. This approach is called the likelihood field model.

After the particles are scored, a resampling technique is then used to put more particles around those with a higher score, and remove particles that have a low score. A popular technique to resample particles after they have been scored is the low variance sampling algorithm [12].

```
list<Particle> low_variance_sampler(X_t, W_t) {
  list<Particle> X_t_bar = {};
  double r = rand(0, 1/n_particles);
  double c = W_t[0];
  int i = 1;
  double u;
  for (int m = 1; m <= n_particles; m++) {
    u = r + (m - 1) * (1/n_particles);
    while (u > c) {
      i++;
      c = c + W_t[i];
    }
    X_t_bar.push_back(X_t[i]);
  }
  return X_t_bar;
}
```
Figure 1: C-Style Pseudocode for The Low Variance Sampling Algorithm [12]

In the algorithm, the initial set of particles is evaluated in terms of their corresponding scores. Then, new particles are selected and added to a new set if their score is less than a certain threshold. In this fashion, more particles keep being placed around highly scored particles, and particles with lesser importance are not chosen again. In the case of AMCL, the number of particles is changed to be in proportion to the confidence of the state estimation. This is what makes AMCL "adaptive" MCL.

1.3.3 The Kidnapped Robot Problem

The MCL algorithm works very well for keeping track of a robot, given its initial pose and that its motion model holds true. However, if the initial pose is not given, or if the robot is kidnapped – that is, moved to some unknown position in a way that its sensors cannot track its displacement – the algorithm will fail. Both of these problems can be solved over time using a global initialization approach to the particle filter, where particles are evenly distributed across the entire map. However, depending on the size of the map and the number of features that the scan-matching algorithm can pick up on, this process can take a long time and is not guaranteed to converge on the correct solution.

Global particle initialization is therefore generally not used when trying to localize a robot in any kind of large environment, especially when that robot is performing an industrial task. If a robot used in a commercial setting were operating without knowing its exact location within its environment, it would be unable to perform its duties effectively or safely.

In addition to MCL needing to be initialized with an estimate of the initial position of the robot, or re-initialized if the robot is kidnapped, the algorithm can, under rare circumstances, lose track of the robot. Depending on the inherent features in the robot's environment captured by the map, the drive train of the robot, wheel slippage and skidding, and gain parameters within the motion model, the particle filter may diverge by splitting into a multimodal distribution. For example, if a robot being tracked by a particle filter takes a sudden turn around a very sharp corridor-dividing corner, such as around a row of shelves, the particle filter may, in rare occasions, diverge into two distributions.

While this situation is indeed an edge case that does not happen with any regularity, it still should be addressed if a system's software that encounters the issue is to be considered robust. However, a more common issue for mobile robots is that the robot may be transported to an unknown starting pose before its sensors are on and collecting data. When the robot's sensors are re-enabled, the sensor model is unable to effectively match scans, since the robot's actual position does not match the particles' estimate. This causes the scores to be low and resampling to fail.

For lightweight robots, this situation can occur by picking up the robot and setting it back down in a different location. During this motion, the wheel encoders cannot sense the robot's motion. The result is an explosion of particles that cannot be recovered from, unless the filter is globally re-initialized, which, as previously mentioned, is impractical for large-scale systems.

One attempt to solve this fundamental problem was by Romanian researchers at the Technical University of Cluj-Napoca. Their approach was to create a separate map based on extracted features from the speeded-up robust features (SURF) algorithm [13]. These kinds of features are traditionally characterized as invariant regions of interest on a given input. The SURF algorithm detects points of interests by using an integer approximation of the determinant of Hessian blob detector. What this means is that it essentially looks for sequences of data that stand out based on the Hessian descriptor, which can be described as a patch of data that is fundamentally different than data surrounding it.

After the separate feature map was built and connected with the original *a priori* map, the robot was able to use the Random Sample and Consensus (RANSAC) algorithm to estimate the robot's pose. This algorithm works as a generic parameter estimation technique, but in this case, the input was the current image, the parameters were the pose of the robot, and the search space was the feature-map.

Although a large amount of conclusive evidence was not provided, the researchers indicated that the technique was able to successfully orient the robot within a correct environment. On waking up, the kidnapped robot initially believes it is in the center of its *a priori* map; but after the re-localization algorithm is run, it is able to correctly localize itself to the room it is in.

While specific error metrics were not provided in the paper, the approach seems to localize the robot within a room. Since it is not an exact localizer, this approach could work well to act as a re-initializer for the particle filter to place particles around the hypothesis pose. If the hypothesis pose was at least within some appreciable distance of the true pose of the robot, the particle filter would converge as if it were manually initialized.

2 Deep Learning and Artificial Neural Networks

Feature extraction within images is a fundamental problem within computer vision. Given the characteristic features that are within an image, complex associations may be created to enable a machine learning classifier to successfully identify images of similar classes. A new and exciting area of machine learning called deep learning uses thousands of tunable parameters to identify 'deeply learned' features that help a classifier identify an image with extreme robustness. However, in order to understand the complex structures used in deep learning classification architectures, one must first understand the foundations.

2.1 Perceptrons

The start of artificial neural networks should be with their base unit, the perceptron. A perceptron is a binary classifier that takes a vector of inputs, does some operation on them, and outputs a positive or negative number to classify the input. The mathematical structure of the output of a perceptron is as follows:

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + \sum_{i=1}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$$

Where w_0 represents a bias term of the perceptron and the $w_{1:n}$ terms correspond to the weights of the network's inputs. To put this equation in perspective, the topological structure of the corresponding perceptron is the following:

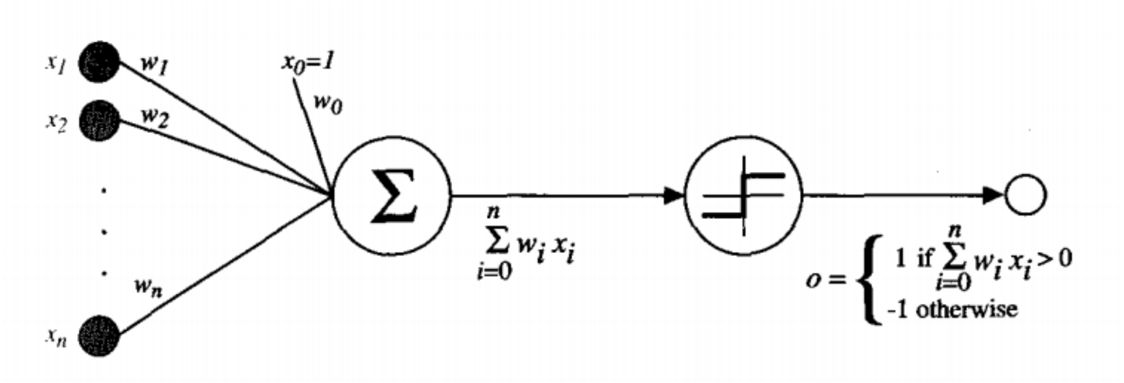

x_1
x_2
x_n
w_1
w_2
w_n
$x_0=1$
w_0
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

Figure 2: The Perceptron [14]

Using this structure, the vector of weights may be a learned parameter that helps create a linear decision boundary in correspondence with the output equation. To find the appropriate weights for this supervised learning problem, a training rule must be established. The weights are usually initialized to random variables within an accepted range and then updated using the following rule:

$$w_i \leftarrow w_i + \eta(t - o)x_i$$

Where w_i is the weight associated with the ith element of the input, η is the learning rate usually set to a small value such as 0.1, $(t - o)$ is an error term corresponding to the target value minus the output value, and x_i is the ith element of the input vector. Iterating through this equation for each input and weight for a set number of epochs or until the error term is zero or within some small tolerance [14].

Perceptrons are able to easily represent the AND, OR, NAND, and NOR boolean functions for any number of inputs, as they are linearly separable on a decision surface. However, the XOR problem poses a difficulty as the decision boundary is not linearly separable. Marvin Minsky, one of the forefathers of the perceptron model and artificial intelligence itself, famously halted research into artificial neural networks for decades by saying that the XOR problem could not be solved using the perceptron technique. In some sense he was true, as a single layer perceptron can not solve this problem, but by adding a second layer, as one may do with logic gates, the XOR problem can be addressed quite simply. Once this was addressed, work in multilayer artificial neural networks began.

2.2 Artificial Neural Networks

Multilayer perceptrons, or neural networks, have evolved over the years to include many different “layer types” to perform different operations. One of the first multi-layer neural network structures was the feedforward neural network. This network type uses a single “hidden” layer to uncover features that are able to approximate a nonlinear target function to successfully classify an input as one of several outputs. The hidden layer is “fully connected” to the inputs and the outputs so the weights associated with each neuron inside the hidden layer may have an effect on the output of the network.

When using a perceptron, the output of the summation becomes binary after being passed through the Heaviside function – returning 1 if the summation is above a static threshold, and a -1 if it is below. This acts as the ‘activation function’ for the previous summation of weights and added bias. For multilayer perceptrons, this output type is insufficient as it is not differentiable. Since a multi-layered system of linear activations cannot approximate a nonlinear target function, a different function called the sigmoidal activation function is used. The new base unit for feedforward neural networks becomes the following:

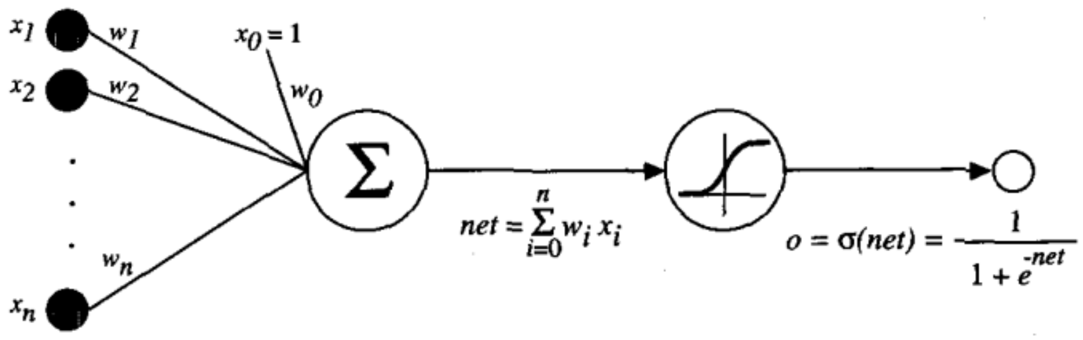

x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

Figure 3: The Feedforward Neural Network Base Unit [14]

$$y = \sum_{i=0}^{n} w_i x_i$$

$$o = \sigma(y) = \frac{1}{1 + e^{-y}}$$

The fully-connected layer remains an important structure within artificial neural networks. However, with the advent of multilayer neural network topologies, different layer types and activation functions have emerged that can affect the output of networks in different ways. One layering technique crucial to the development of deep neural networks applied to image classification is the convolutional layer.

In mathematics, a convolution is defined as a function derived from two given functions by integration that expresses how the shape of one is modified by the other. This convolutional operation, when applied in a deep neural network, is a method of sweeping over an input image using a fixed size kernel in order to extract information from that image [15]. Each kernel slides over the given input data with a certain size and stride. The size is a dimension consisting of a height, width, and depth with a smaller height and width than the input image and the same number of depth channels. The stride acts as the padding used while the kernel is sliding over the input. The kernel acts as a kind of filter by producing a 2D activation map of maximally excited responses that end up becoming a type of visual feature recognizer. These feature descriptors can be anything from primitive features such as lines and shapes to more complex features such as body parts or other objects. A depiction of a network using convolutional layers is below in figure four.

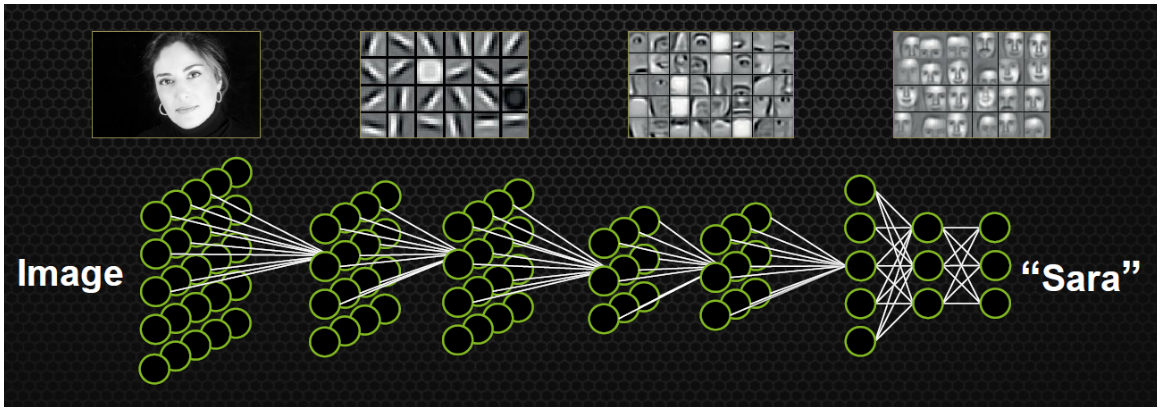


Figure 4: A Depiction of Convolutional Layers and Their Features in a ConvNet [16]

After the convolutional layer, there is typically an entire layer dedicated to applying an activation function to its input. In deep neural networks, this function is oftentimes the Rectified Linear Unit (ReLU). The ReLU function as a function of x is defined as the following:

$$f(x) = \max(0, x)$$

Which simply normalizes any negative input to zero and returns the input if it is greater than zero.

After the ReLU layer, there is typically a pooling layer to down-sample the output. Here, the kernel size and stride indicate the window to be sampled from. When the kernel is over some data, the max of that data is chosen to propagate through to the next layer. After down sampling, more convolutions may occur to pick up on deeper features within the image. Topologies can range from just a few layers, such as LeNet-5, to hundreds such as Microsoft's ResNet.

The last layer worth mentioning is the softmax classification layer. This layer normalizes neural excitations in a final layer to be in between zero and one, with their sum adding to one. The number associated with each output directly corresponds to the probability of the class being the correct classification for the propagated input. The softmax function that operates on network values "z" is the following:

$$\sigma(\bar{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

2.3 Neural Network Training

In order to train all of the weights inside of any kind of multilayer perceptron or neural network, an algorithm must be used to find a solution that minimizes the error between the desired output of training data and the actual output. Traditionally, this problem has been handled by use of the back-propagation algorithm [17]. This algorithm works by propagating an input vector with a known result through the network. When the output layer gives a result, the resulting value and the target value are passed through a pre-defined loss function, yielding an error. Based on partial derivatives, sensitivities are calculated corresponding to the influence of each synaptic weight in the network. The sensitivity computations are back-propagated through the network, and each weight is adjusted according to computed sensitivities. Pseudocode for the backpropagation algorithm could be as follows:

```
initialize_network_weights();
double error;
uint n_epochs = 0;
while(error >= error_thresh || n_epochs < epoch_thresh) {
  for (auto training_examples : ex) {
    prediction = forward_pass(ex);
    actual = ex.true_value();
    error = loss_function(prediction, actual);
    list<weight_updates> d_w_vec;
    for (auto network_layers.reverse() : layer) {
      // backward pass, output to input
      d_w_vec += compute_d_w(layer, error);
    }
    update_weights(network_layers, d_w_vec);
  }
  n_epochs++;
}
```

Figure 5: C-Style Pseudocode for the Backpropagation Algorithm [17]

In order to determine how to update each weight, or parameter, inside the network, a solver must be used. The solver's ideal job is to compute the gradient of the objective function the network is attempting to learn, and incrementally move the parameters towards the global minimum of that function. In practice, the solver may find itself in local minima, which can sometimes be good enough for the purposes of the classifier. The effect of this training process is to cause the loss function to approach zero. This process breaks down into an optimization problem that is commonly solved by a gradient descent method.

2.4 Neural Network Solver Types

Many gradient descent algorithms exist, with the two traditional algorithms being batch gradient descent and stochastic gradient descent [18]. These two algorithms work by calculating the gradient, multiplying it by a learning rate (set to something less than one), and then setting the parameter or parameters to their new values. Batch gradient descent calculates the gradient of the objective function at each current parameter value.

Then, once the vector of gradient values has been calculated, the parameter update is applied in the following fashion:

$$\underline{\theta} = \underline{\theta} - \eta \cdot \underline{\nabla_\theta J(\theta)}$$

Where theta represents the parameters, eta the learning rate, and J the objective function. This technique is the optimal way to perform gradient descent, but is prone to redundancies when computing similar gradients and can take a very long period of time to run on the large numbers of parameters found in deep neural networks. To combat the runtime issues of batch gradient descent, stochastic gradient descent performs the same operation iteratively, updating one parameter at a time:

$$\theta = \theta - \eta \cdot \nabla_\theta J\left(\theta; x^i; y^i\right)$$

In its fashion, stochastic gradient descent is able to converge to a solution much more quickly than its batch-based sibling. However, because of its iterative nature, it may continually overshoot the global minimum of the objective function. This is because, with each step, the function only proceeds in the minimization direction of one parameter; not taking into account all of them. A technique to combat the overshooting is to make the learning rate slowly decrease over time, leading to lesser influence by later parameter updates. However, this method is still prone to becoming trapped in local minima, yielding a sub-optimal solution to the objective function.

A combination of these two techniques that is the ‘de-facto’ deep learning approach to using stochastic gradient descent, is called mini-batch gradient descent [18]. This technique attempts to combine the best features of batch gradient descent and stochastic gradient descent into one by performing an iterative gradient descent using batches of parameters and their gradients. The parameter update equation for mini-batch is:

$$\theta = \theta - \eta \cdot \nabla_\theta J\left(\theta; x^{i:i+n}; y^{i:i+n}\right)$$

Where n is the 'batch size' that will be used for each iteration. Many commonly used deep learning libraries use mini-batch gradient descent for their "normal stochastic gradient descent" implementation, referred to as SGD. For example, the deep learning library Caffe [7] does this, with an additional term added in for "momentum" – a term added to the weight update equation to help SGD avoid getting stuck in ravines around local minima.

As the solver iterates further and further through the objective function, the momentum term increases the step size until a maximum is reached. This helps the solver converge faster with minimal oscillation. The parameter update equation with added momentum is broken up into two equations, one defining the update, and the other defining the modified gradient approach:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$

However, as the step size increases, one may imagine that convergence on a global solution may be more difficult. Nesterov's accelerated gradient (NAG) is an approach that seeks to give the momentum term some knowledge about its previous weight update in order to dampen momentum when convergence is beginning to occur [18]. By computing the difference between the current weight and the momentum multiplied by the previous update term and using that as the value for the result of the objective function to compute the gradient at, NAG is able to slow down the momentum when required. The update now looks like this:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$

$$\theta = \theta - v_t$$

The next logical step in improving the performance of this solver is to have it take into account each individual parameter in the update step, increasing the step size more in one direction of the parameter-space if the corresponding parameter is somehow more important than the others.

The first algorithm that attempts to tackle this problem is Adagrad, or adaptive gradient descent [18]. Since it is able to update its steps coinciding with the importance of one or more parameters, it and other algorithms of its class are better suited for dealing with sparse data. The way Adagrad handles the weighting of different parameters is by giving each parameter its own learning rate. Then, for each iteration, the learning weight for the parameter in question is modified by the square root of the sum of the squares of the previous gradients up to the current time step. This is all handled using matrix math for efficiency.

$$g_{t,i} = \nabla_\theta J(\theta_i)$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \odot g_{t,i}$$

Where G is a diagonal matrix of the squared sums of all previous gradients corresponding to the parameter in question and epsilon is a very small smoothing term that prevents division by zero. The $\odot$ operator is an element-wise matrix-vector multiplication that allows for efficient calculation of the parameter update. Since deep neural networks contain a huge number of parameters to be tuned, the matrix G can oftentimes grow to be extremely huge, leading to an inefficient use of space. Furthermore, as the denominator increases, the learning rate approaches zero, causing the algorithm to stop updating the parameters – often before a reasonable solution is found.

The next algorithm, Adadelta, addresses these issues [18]. By keeping a decaying running average of the squares of the past gradients within a fixed-size window, the algorithm is able to be much more efficient.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2$$

To be distinct, the change in parameter equation that was previously v_t in NAG is now written as $\Delta\theta_t$ to indicate the parameter change. To show the change from Adagrad to Adadelta, the current parameter update equation is:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

The denominator is actually the root mean squared (RMS) error with respect to the gradient, so it can be conveniently replaced by an RMS expression:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$$

In the derivation of the Adadelta parameter update equation, the units do not match between the previous parameter and the updated one. To address this, the authors define another exponentially decaying average of squared parameter updates that also turns out to be an RMS error:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta^2$$

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta]_t + \epsilon}$$

Finally, the RMS of the current parameter is estimated using the value of the RMS of the parameter in the previous time step. This yields a final Adadelta update equation of:

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

The next algorithm worth considering is RMSProp [18]. While technically unpublished, it was derived around the same time as Adadelta and has very similar functionality to it. The algorithm works almost identically to that of Adadelta but seems to ignore the inconsistency in units. The update equation is identical to the first update of Adadelta:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

The final algorithm that will be discussed is Adam, or Adaptive Moment Estimation [18]. It has similar functionality to that of Adagrad and Adadelta as it keeps track of past squared gradients, but it also keeps track of a decaying running average of the gradients themselves to be used as an adaptive momentum term:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

The term m_t is an estimate of the first moment of the summed gradients and the second term v_t is an estimate their second moment. The terms are initialized as zero, so in order to avoid zero-biasing, they are re-written with bias correcting terms as:

$$\widehat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Lastly, the parameter update equation comes out to be:

$$\theta_{t-1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

Adam has been shown to outperform other adaptive gradient descent algorithms since it is able to estimate both the first and second moment of the gradients and use them to appropriately step the weights in the correct direction. In general, adaptive techniques are very useful when dealing with sparse data as they are able to adjust learning rates for the smaller amounts of features available to the algorithm [18].

All of these solver types are included in the Caffe distribution within NVidia's DIGITS [8] software that was used to train three different pre-existing neural network architectures for the verification of the hypothesis. The different architectures were Lent [19] by LeCun et al., AlexNet [20] by Krizhevsky et al., and GoogLeNet [21] by Szegedy et al. Consequently, and for simplicity's sake, these were the networks used for proof-of-concept development, testing, and later deployment.

2.5 Significant Deep Neural Network Topologies

The following three networks were chosen for training to eventually classify the current region the robot is in for use in coarse localization. LeNet-5, AlexNet, and GoogLeNet are oftentimes used as a de facto standard when testing out different neural network applications, so they are well suited for this application.

The first convolutional neural network (ConvNet) architecture to be discussed is LeNet-5 developed by Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. It is the fifth evolution of the LeNet convolutional neural network architecture, with the first being introduced in 1989 for recognizing digits from the MNIST dataset [22]. Compared to the deep neural networks of today its architecture is relatively small – only having seven layers.

This is fitting for the time as the corresponding paper was published in 1998 and described the network as being better suited than any other algorithm at the time for recognizing hand-written characters. The network's size is appropriate for the task at hand as the features which it needed to discover are from a limited set of primitive shapes. Consequently, LeNet-5 performed extremely well on the MNIST dataset – achieving about 99% test accuracy without overtraining. It was even able to recognize heavily distorted numbers as well as numbers with heavy noise added. (On his website, LeCun characterizes these cases as the network being able to classify "weird" images.)

LeCun provides a visualization of his network which allows for one to understand what each layer is doing to the image:

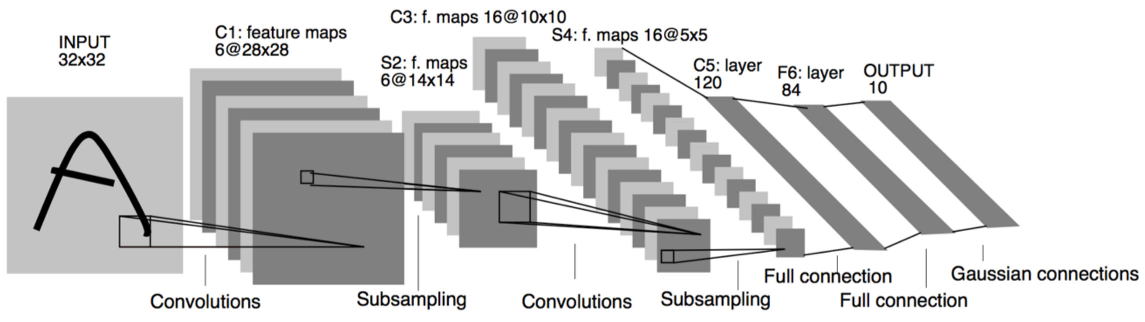


Figure 6: The LeNet-5 Network Architecture [19]

The first convolutional layer creates a six-layer feature map, followed by max pooling subsampling to reduce the size of the feature maps by half. These subsampled feature maps go through one pass of convolution and max pooling. Then, they are passed through a series of two fully connected layers of width 120 and 84 respectively. Finally, they go through a gaussian connection layer that is, in fact, a softmax layer. This allows the fully connected weights to be normalized to between zero and one, allowing the probabilities of each class to be shown.

LeNet-5 was published in 1998 when the processing power of computers was far from what it is today. For this reason, work on convolutional neural networks slowed to a very shallow pace of development for quite some time. However, with the advent of better NVidia graphical processing units (GPUs), associated CUDA (Compute Unified Device Architecture) libraries, and finally, the CudNN[9] library for building and training deep artificial neural networks, more progress began. CUDA and CudNN allow for code to be executed directly on a GPU, allowing for a large amount of parallelized processes to be run at the same time. This technique allows for all of the algorithms associated with training these deep architectures to be run significantly faster than on a traditional processor.

As a result, it was not until the late 2000's and early 2010's when much deeper ConvNets became feasible. In 2012, one of the first truly deep convolutional neural network architectures, published by Krizhevsky et al., was able to correctly classify ImageNet, a database with over 15 million labeled high-resolution images in over 22,000 categories, to 84.7% accuracy. Their deep convolutional neural network, AlexNet [20] was able to beat the previous state-of-the-art in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) [23] by using their novel topology.

Krizhevsky et al. were able to accomplish their feat by making use of several, now common, techniques. The first was the ReLU nonlinear activation function which they used to replace the normally used hyperbolic tangent or sigmoid. By using ReLU

[9] https://developer.nvidia.com/cudnn

layers as their activation functions, they were able to decrease the error rate of the solver from 75% to 25% in 30 fewer epochs. This corresponds to about a sevenfold increase in performance – which adds up quickly when the training duration is often hours, sometimes days.

Another tactic the team used to speed up training time was using multiple GPUs to train their network. Since modern GPUs are able to read and write directly from each other's memory, Krizhevsky et al. were able to write a highly efficient, parallelized convolution operation that helped them further increase the accuracy of the network, and speed up training time. They were able to increase the accuracy by putting half of the trainable neurons on one GPU and the other half on the other, and only allowing them to communicate error rates at certain layers.

A third tactic, employed to increase generalization and avoid overfitting within the classifier, was local response normalization. Normally, ReLU nonlinearities in a network help make sure that it generalizes between classes. However, the researchers found that normalizing the activity of a certain neuron before applying the activation function using the local response normalization function increased the network's accuracy. The function which applies this normalization is the following:

$$b_{x,y}^{i} = a_{x,y}^{i} / \left(k + \alpha \sum_{j=\max\left(0, i-\frac{n}{2}\right)}^{\min\left(N-1, i+\frac{n}{2}\right)} \left(a_{x,y}^{j}\right)^{2} \right)^{\beta}$$

Where $a_{x,y}^{i}$ is the activity of a neuron at coordinate (x, y) from kernel i, $b_{x,y}^{i}$ is the cooresponding output, n is the number of adjacent kernel maps, N is the total number of kernels in the layer, and the other variables are learned hyperparameters determined using a validation set.

The final technique that was used to make AlexNet perform so well compared to its predecessors was overlapping pooling. This was a slight change in how the max pooling layers within the network operated. Normally, each max pooling output does not overlap, but by combining the output of pooling layers in an overlapping fashion, the error rate was decreased by 0.4%. One may speculate that this was due to some kind of additional generalization created by "smudging" the values together. AlexNet also uses the principal of "dropout", which occasionally "drops out" units by setting weights to zero.

The version of AlexNet which Krizhevsky et al. trained to compete in the ILSVRC had parallelized layers running on different GPUs. The first operation was a convolution, followed by max pooling, then five more iterations of the same. Finally, there were two levels of fully connected layers followed by a softmax classifier. A figure of the topology is as follows:

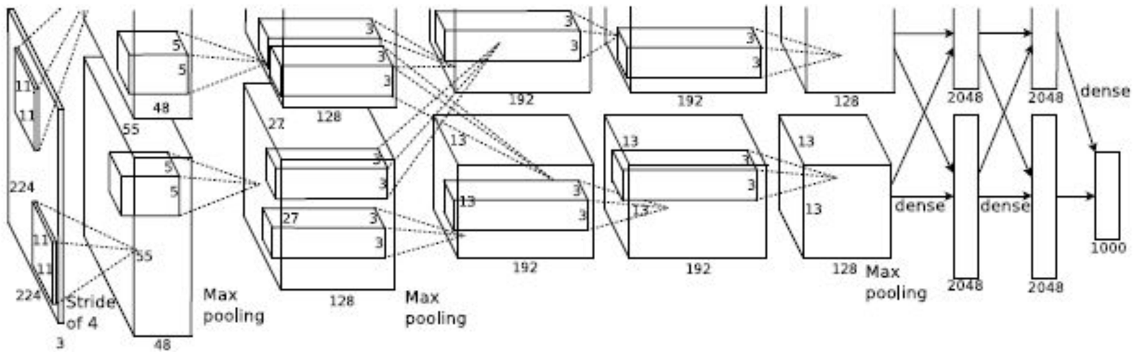


Figure 7: The AlexNet Network Architecture [20]

AlexNet was able to successfully categorize images up to 84.7% accuracy. This was a major advancement from the previous state-of-the-art by about 10% error [23]. The power of deep convolutional neural networks was proven, but there was and is still progress to make.

In 2014, researchers from Google, The University of North Carolina at Chapel Hill, and University of Michigan, Szegedy et al., set yet another state of the art with a new deep topology, codenamed Inception [21]. They were able to set the new state-of-the-art at the ILSVRC with a twenty-two layer deep network called GoogLeNet based on the principles of Hebbian learning, that is, "neurons that fire together wire together". The team was able to create their network topology in such a way that they used 12x fewer hyperparameters than AlexNet, but still were able to get significantly better results on the order of 6.67% error.

Typically, deep neural networks are made better by simply increasing their size. However, because of the increase in the number of hyperparameters, these larger networks are also more prone to overfitting. As these networks scale, they also require more processing power to train; lengthening the time needed to do the required iterative testing needed by researchers.

To combat this, fewer network connections would be required for both fully connected and convolutional layers, which as it turns out, is in agreement with a theory proposed by Arora et al. indicating that a sparse deep network may be constructed statistically by observing the correlations between layers. This theory also resonates well with the principle of Hebbian learning [24]. The fact that this theory would combat two major problems in deep learning, and is in agreement with a theory of how neuropathways in our own brains actually are formed, is encouraging. The Inception network created by Szegedy et al. was an attempt to extrapolate this theory into something usable, with much success being found.

Despite the successes of the network architecture, the authors warn that more research will need to be performed before anyone might be able to call their model-generation technique a catch-all. More tests will need to be done for classifying other types of data in different situations using the same approach to create completely different looking topologies.

The architecture of GoogLeNet is comprised of “inception modules”, spatially repeated units of optimal local construction that were discovered by high correlation analysis between layers. The correlation analysis between layers corresponds to an area of an image with respect to a subsection of that image, so a high correlation may imply an important feature. Convolutional kernel filter sizes used were restricted to 1x1, 3x3, and 5x5 pixels to avoid patch alignment issues in which a decreasing number of highly correlated patches of image decrease over larger and larger regions.

The overall architecture is made up of these discovered Inception submodules which consist of the small kernel size convolutional layers, intertwined with max pooling, followed by a “depth concatenation” layer which resizes the output to begin again. There are also auxiliary classifiers at different stages of the network that allow for different softmax outputs to be viewed at different stages of learning and execution. The unique topology of GoogLeNet was able to advance it to the top of the ILSVRC and revolutionize the deep learning community, inspiring several more iterations of the network.

2.6 Deep Neural Networks in Scene, Place, and Pose Recognition

The first three deep architectures presented, LeNet-5, AlexNet, and GoogLeNet were all designed to classify images based off of their type. While LeNet-5 classified

MNIST numbers, AlexNet and GoogLeNet were able to classify a vast number of labeled images from churches to slugs to staplers with ever-increasing accuracy. However, this begs the question of whether or not these advanced machine learning classifiers may be applied to other realms of classification. As it turns out, the answer to this leading question is yes; and much success has been seen in extracting information from images other than a naïve label.

A very interesting problem that could bridge the gap between deep learning and mobile robotic localization is the challenge of scene, place, and pose recognition. While the problems do sound similar, each is actually a separate task that influences the next. The challenge of scene recognition details the task of having a classifier identify multiple items within a single image and making sense of what the entire *scene* implies. For example, an image of a couple sitting at a pizza place eating pizza, according to GoogLeNet, may have the label "pizza restaurant", whereas a scene classifier may actually give the end user some idea as to what is going on in the image. That is, a couple sitting at a pizza place enjoying a meal, or perhaps even on a date based off of some other visual cues.

The second and third tasks are slightly more connected. Place recognition is the task given to a classifier to locate where an image may be located given some search space. For example, given an image of the Eiffel tower, the place recognizer may return "Paris, France". Pose recognition gets even more precise by training a deep neural net to return the exact six degree-of-freedom $< X, Y, Z, R, P, Y >$ pose of the camera that took the photo. This, if deployed on a mobile robotic system, could have the capability to solve the localization problem if the algorithm was robust enough and could localize the robot exactly in its environment.

PoseNet [25] attempts to solve this problem, with surprisingly accurate results. Cambridge researchers Kendall, Grimes, and Cipolla were able to re-use the majority of GoogLeNet to solve just this problem. By stripping the final softmax layer and replacing it with a layer capable of returning a 6-DOF pose, the team was able to obtain approximately 2 meter and 3° accuracy for large-scale outdoor scenes and 0.5 meter and 5° accuracy for indoor scenes.

The main contribution that Kendall et al. made to the deep learning community was the use of an affine regressor to produce a pose that replaced the softmax layer. Their network was able to perform a substantial task, but the localization error is still too great for a small indoor mobile robot to know its location when performing critical operations in an industrial setting.

Training data specific to PoseNet comes from a video taken by a smartphone while walking around the city of Cambridge in the United Kingdom. Structure-from-motion visual odometry was used to give each image a ground truth pose label for training. The video was subsampled at 2Hz to match the processing speed of the visual odometry algorithm.

Kendall et al. also describe using transfer learning to accelerate the robustness and number of examples in the network. According to their findings, a network pre-trained with features from a large database increases test accuracy and reduces time to convergence on an acceptable solution. They do not specify how this data is used within their network, but it would either be intermingled with training data, or a network would be pre-trained with it, having final layers being trained on only the data needed to regress the camera pose.

But, do we need exact pose? If a robot were able to recognize the place or location it was in, perhaps that could be enough to help already-robust algorithms do their jobs even better. This would be definitely advantageous as the accuracy level developed by PoseNet, while certainly impressive, was not enough for a mobile robot to localize exactly.

Scene recognition is another area of research in which deep learning has been applied successfully. In 2014, Zhou et al. trained a convolutional neural network on a new dataset titled "Places", a set of seven million labeled scenes [26] [27]. The network was of the same topology as the Caffe reference network, AlexNet. By using AlexNet trained on a new database, an accuracy of 56.2% was achieved. This is far from a fantastic result, but it led to more work being performed on a greater number of datasets. On a newer version of the Places database, Microsoft's ResNet [28] achieved 85.07% accuracy.

In 2015, Liao et al. developed a scene classifier that leveraged the power of deep learning to model human scene understanding based off of object knowledge [29]. By training the classifier on items common to an office, rest space, kitchen, and other similar categories, and giving the associations of those objects meaning, the researchers were able to achieve very successful results classifying scenes in the SUN RGB-D dataset [6], a Princeton Vision and Robotics group dataset for benchmarking scene classification. By training their network to recognize images within certain environments, and coordinating those images with the kind of environment they imply, the team was able to perform better than the previous state-of-the-art.

The techniques applied to scene recognition can also be applied to place recognition. If a robotic system were able to use a deep learning classifier to output a location to a scene captured by a camera. One could extrapolate that like in [13], if a classifier running on a robot was able to find meaning from a scene, its navigation software may be able to use that information in order to localize the robot.

However, place recognition requires the neural network to be able to recognize the scene or scenes corresponding to an actual place with a certain amount of invariance associated with the classification. The place must be recognized from different angles, with objects being slightly rearranged within the environment, with people obstructing the visibility of perhaps more static features [30]. Fortunately, convolutional neural networks are good at addressing all of these issues. Deeply learned features in convolutional neural networks tend to be more robust in these scenarios than their hand-engineered counterparts [30].

According to research performed by Sünderhauf et al. of the Queensland University of Technology [30] features from higher layers of ConvNets encode semantic information about a place and partition the search space within the network. Features from middle layers are robust to appearance changes such as the time of day and seasons, and features from the final layers are robust to viewpoint changes. The ConvNet they used to make these findings was AlexNet, a network originally created for use in the ILSVRC, and not to identify scenes or places.

2.7 Transfer Learning

Mentioned earlier in its application with PoseNet, transfer learning is the application of the machine learning process of knowledge transfer to deep neural networks. Its intent is to use a classifier that was previously trained on, perhaps, a related dataset, and re-train or re-use some of its learned knowledge to classify something new.

According to Gutstein et al. of The University of Texas at El Paso [31], deep neural networks are particularly well suited for this task. In their research, they found that by stripping off layers further down in the network, they were able to leverage the deeply learned features in earlier layers to increase the performance of the network when applied to a new dataset of lesser size. For their 2008 experiment, they used an architecture similar to LeNet-5, trained on a NIST dataset similar to that of the MNIST one. Afterward, 0-5 layers were 'held back' and left alone while the other layers that were not static from the initial trained were trained using 1, 5, 10, 20, and 40 examples from the next set.

Results showed that at least 10 examples worked best in this scenario. Little improvement arose when more than this amount of training examples was used. Furthermore, it showed that 3-4 layers should be kept static, and the other 2-3 should be trained on with the new data. Future work spoke about holding back specific neurons in the layers and not just the layers themselves.

3 Acquisition of Labelled Training Data

The acquisition process to gather labelled training data is a paramount step in creating any type of methodology to train and deploy a machine learning classifier in order to perform a task. As the classifier was deployed onboard a robot, specialized software was written to use the robot's sensors to gather the required data.

3.1 Robot Hardware

For this project, an Autonomous.ai brand Deep Learning Robot[10] was used. This robot is a low-cost clone of the Turtlebot, a commonly-used inexpensive mobile robot designed by Willow Garage. Since the "Deep Learning Robot" neither trains neural networks, nor is it much different than a Turtlebot, it will be referred to as a Turtlebot for the remainder of this thesis. This Turtlebot consists of a Kobuki[11] mobile base, manufactured by Yujin Robot, a NVidia Jetson TK-1[12] ARM development board running Ubuntu 14.04 and ROS Indigo, and an Asus Xtion Pro[13] RGB-D camera. The robot also came with a Bluetooth speaker that was not used in the project.

Furthermore, the robot's shelving levels were configured differently than suggested by Autonomous.ai, with the second smaller level immediately following the first, then followed by the tallest level and the top of the robot. The robot came with no mounting equipment for the Asus Xtion pro camera, which is important if a static transform is to be derived for use by the ROS robot model. So, a laser cut-able mount[14]

[10] https://www.autonomous.ai/deep-learning-robot
[11] http://kobuki.yujinrobot.com/
[12] http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html
[13] https://www.asus.com/3D-Sensor/Xtion_PRO/
[14] http://www.thingiverse.com/thing:16761

was used to attach the camera on bottom of second level of the robot.

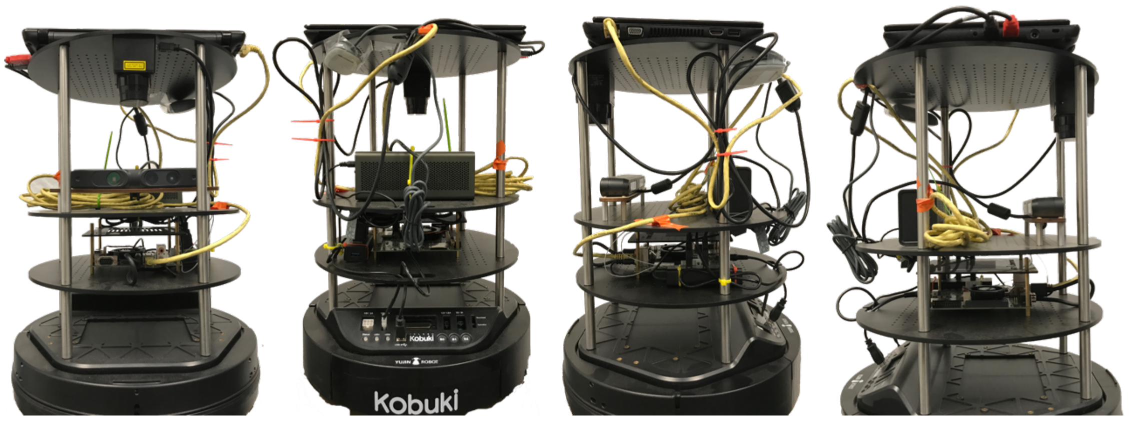


Figure 8: Front, Back, Left, and Right Views of the Turtlebot

In addition to the base components, an Asus Aspire one A0756-Q1VZC netbook was mounted to the top of the robot in a networked slave configuration to the ROS master running on the NVidia Jetson TK1. A Hokuyo URG-04LX-UG01 LIDAR was also mounted underneath the top shelf. Typically, a real spinning LIDAR is not required on Turtlebot as there are ROS nodes designed to convert the depth image or point cloud published by the RGB-D camera to a simulated laser scan. Unfortunately, testing showed that the ARM processor on the Jetson TK1 was insufficient to run the ROS driver for the Kobuki base, the *OpenNI2* drivers for the Asus Xtion Pro, *robot_pose_ekf* for performing sensor fusion on the odometry for more accurate results, as well as a laser scan generating package. So, to lighten the load on the TK1 with the added benefit of more accurate scans, the Hokuyo LIDAR was added and plugged into the Asus netbook for processing.

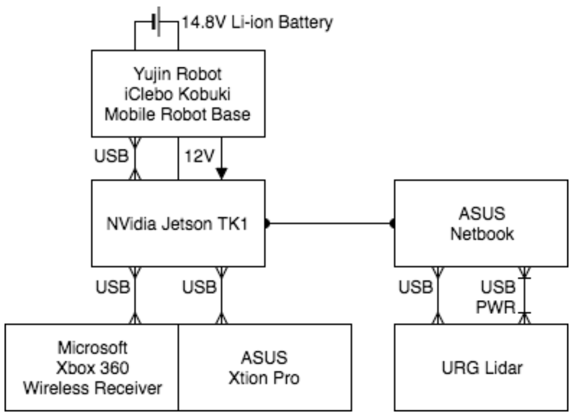


Figure 9: The Hardware Diagram

Upon receiving the Turtlebot, assembling it, and turning it on, it became quickly apparent that it did not come with any supporting software. So, in order to use the robot properly, a new ROS stack specific to Case Western Reserve University was created called *cwru_deeplearning_robot*. Within the stack, there exist several custom packages that help support the robot's full functionality.

3.2 Robot-specific Software

The first custom package that was developed is the *deeplearning_description* package, which is a copy of the *turtlebot_description* package maintained by the official Turtlebot development team. This package contains 3D meshes for visualizing the robot's components in RViz, the ROS robot visualization program, universal robot definition files (URDF) that describe how the robot's linkages and parts are positioned and oriented with respect to a common frame, and robot-specific "xacro" files that stitch all of the URDF files together. This type of package is absolutely crucial when developing a robot using ROS as when the URDF is loaded into the ROS parameter server, all of its transforms are available to programs running on the same master. This is important for calculating the robot's odometry as well as calculating the transform between the base frame of the robot to the map frame, the task which global localization programs such as AMCL attempt to perform.

The next important software package in the stack is *deeplearning_bringup*. The purpose of this package is to load the URDF into the parameter server and launch all of the necessary drivers required for the robot to begin operating. The main launch file, *deeplearning.launch*, contains includes for the Kobuki base drivers, the *robot_pose_ekf* package for performing sensor fusion on the raw odometry and IMU data from the Kobuki base, the *OpenNI2* package for running the Asus Xtion Pro image streams, a line to load the URDF, and lastly a line to start teleoperation of the robot via an Xbox 360 controller and the ROS package *joy*. This launch file leverages a popular technique by hiding component-specific includes through separate launch files within a subdirectory. These separate launch files have an additional suffix of .xml in order to render them unrecognizable by the roslaunch tool, but includable by other launch files. This allows for the main launch file to be more readable, and component-specific launch files to exist and be maintained separately.

The last package that is a part of the *cwru_deeplearning_robot* software stack is the *deeplearning_launch* package. The purpose of this package is to house utility launch files specific to the Turtlebot used for the project. Currently, there exists a launch file to start *gmapping*, a SLAM algorithm typically used to make an occupancy grid map of the robot's environment, and another to load a map of the lab's office area onto the map server and start AMCL.

All of these software packages had to be created in order to use the Deep Learning Robot as if it were a normal ROS-compatible robot. They are open source and available online on the CWRU Robotics GitHub page[15].

3.3 Data Collection Process

In order to train any neural network, let alone one that will assist in a localization task, training data must be collected. Images were collected using the onboard Asus Xtion Pro camera. Even though the camera is able to supply RGB-D data, only RGB data was use to keep consistent with the deep learning techniques that were evaluated.

Data for the proof-of-concept testing phase was collected using a pipeline of ROS tools and a custom script. First, the robot was placed in an area to be recognized later. Next, the rosbag utility was started in order to capture every single frame being published to the "/camera/rgb/image_raw/downsized" topic.

Rosbag is a ROS utility that records data on specified topics and writes it in a binary format to a special ROS 'bag' file. These bag files are then able to be played on to a ROS core at a later time to re-publish all of the file's saved data. A bag file is recorded for each area to be later classified and should encompass images of the whole area. A useful utility to run during data collection is *image_view*, which allows the user to view the live image being recorded to the bag file.

Downsized images were being published to this topic at 30Hz by a separate node, *img_downsize*. This node works by subscribing to the full-size image, converting it to a different size using values stored on the ROS parameter server, and republishing it on the same topic with "/downsized" added on.

[15] https://github.com/cwru-robotics/cwru_deeplearning_robot

Even though the images were downsized from their original 640x480, they were only reduced to 400x400, so each image could have copies at different resolutions saved. Since the images were still rather large, the bag file grew rather quickly and a locally networked master-slave system was used to have a netbook with a larger hard drive capture all of the data instead of the Turtlebot's onboard computer. Once the area had been sufficiently explored, the rosbag utility is terminated and the bag file saved. This process is repeated for each new area that would like to be recognized later on.

After bag files for each are collected, the *extract_images* node from the ROS package *image_view* is used to extract JPEG encoded images from the bag. To make this convenient, a new package was created called *bag_img_extractor*. In this package, there contains a launch file that plays the bag from which the images are to be extracted, and runs the extractor with parameters for the image topic to subscribe to as well as the sampling rate. To use the launch file, one must edit the path to the bag in question within the launch file, then launch it. After is has finished running, all of the images will be saved in the ROS_HOME directory, a Linux environment variable that is usually set a hidden ".ros" folder in the home directory.

Once the full-size images are within the ".ros" directory, they can be moved to a separate directory easily using the bash command "mv". Since all of the image files are JPEGs and no other JPEGs normally exist within the ROS_HOME folder, they may be moved in bulk to a folder with their class label using the wildcard operator, that is, "mv *.jpg /new/path/label". It is important to create a folder with the same name as the class label of the images to house all of the images of that class. This is a convention for many deep learning libraries that allows them to easily identify the label of each piece of training data, regardless of the image's filename. Grouping all of the full-size images in each of their class label folders inside another directory is also beneficial for post-processing.

After the full-size images are moved into a proper directory, they are now ready to be shrunk to different resolutions for testing with different neural network topologies. For this, a script called *generate_training_data.py* was written. This script takes in as an argument the root directory where the class-labeled folders are located, loops through them, and converts the training images within them to a variety of different sizes. For each specified size of image to be generated, the script creates a folder within the label directory to store the downsampled images. Then, using the fork of the Python Image Library (PIL) called Pillow[16], the program converts each file to each requested size, placing the new image inside the appropriate folder. Lastly, after some manual re-arrangement of the folders, the new dataset for the areas to be recognized later on is finished.

After the process of generating all of the different sizes of training data, DIGITS is used to create a training set database which its models may then be trained using. To carry out this procedure, one must simply navigate to the DIGITS interface webpage, select the 'images' drop down menu underneath 'new dataset', select 'classification', and then fill in all the required details for creating the database.

3.4 Dataset Creation

Within the DIGITS system, there is functionality to create a new dataset or create a new model based on some existing dataset. For this project, three main datasets were

[16] https://pillow.readthedocs.io/en/4.1.x/

created. The first was a dataset of 300x300 images of four different classes. The classes the images corresponded to were “lab”, “machineshop”, “office”, and “serverroom”. This dataset was meant for use by the AlexNet and GoogLeNet topologies provided by the DIGITS system for easy deployment. The next dataset that was created was exactly the same, but with 32x32 images for use with the LeNet-5. This 4-class dataset consisted of 1435 images in total, with 482 being part of the “lab” label, 487 having the “office” label, 243 with the “machineshop” label, and 223 having the “serverroom” label.

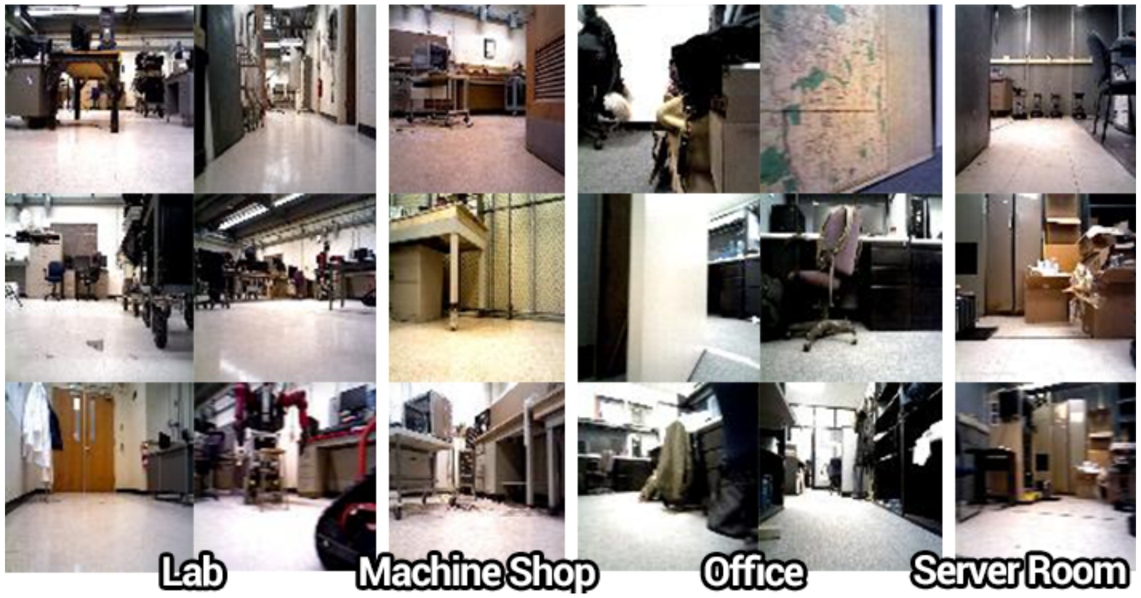


Figure 10: Sample Images from the 4-class Dataset

The last dataset was similar to the first, but with three additional classes of “entryway”, “circuitslab”, and “lounge”. All data was collected in the second and third floors of the Glennan building at Case Western Reserve University in Cleveland, Ohio. This dataset had a total of 2449 images, with the intersection of the two different sets having the same content, and 172 of the images belonging to the “entryway” class, 515 belonging to the “circuitslab” class, and 327 belonging to “lounge”. For both the 4-class series of datasets and the 7-class dataset, a separate validation set was taken by both driving and carrying the robot around the different areas it was trained around. The same method for creating the training sets was used to create the test set.

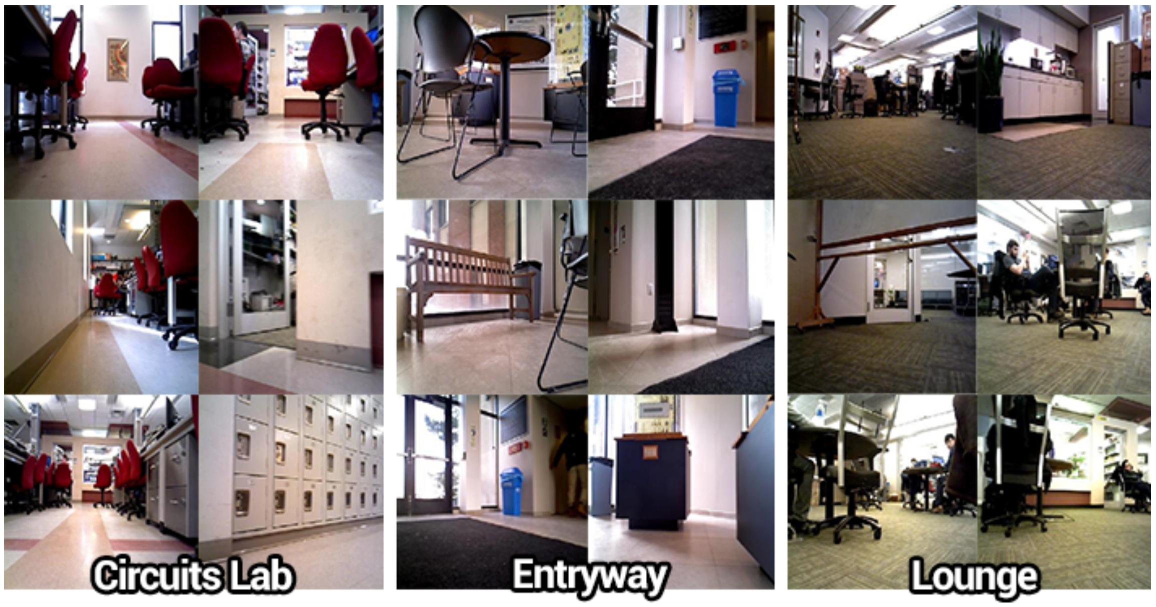


Figure 11: Selected Images from the Additional Three Classes for the 7-class Dataset

4 Deep Neural Network Training and Analysis

To speed up deep learning training and analysis, NVidia's easy to use deep learning training program, DIGITS [8] was used. DIGITS, or the Deep Learning GPU Training System, is a web app that can be installed on a computer with a NVidia GPU that allows for deep learning models to be quickly created and tested. After installing CUDA and Caffe [7] to the computer or device the end-user wishes to use DIGITS on, they can download a Debian package from NVidia's website to install DIGITS. After installation, DIGITS runs as a web server on the computer and is accessible locally by opening a browser and navigating to http://localhost, or remotely by navigating to the IP address or hostname of the computer from any browser on the same network.

The deep learning libraries supported by DIGITS are Caffe and Torch[17]. For this project, Caffe was used. Caffe supports all of the major solver types for training deep neural networks, and for each topology, all solver types were evaluated. The topologies evaluated were LeNet-5, AlexNet, and GoogLeNet.

4.1 Training Procedure

The training procedure using DIGITS is very intuitive. After a training database has been created using images gathered from the robot, all that needs to be done is navigating to the 'images' drop down menu underneath 'new model', and selecting the 'classification' option. In the next screen, the user can then specify all of the training details using the on-screen user interface. Details to be specified on this screen include

[17] http://torch.ch/

the dataset to use, the number of training epochs, and type of solver to use, as well as the network to use and the model name.

DIGITS comes with the three built-in network types of LeNet-5, AlexNet, and GoogLeNet. However, it is also easy to define your own network architecture using prototxt, a mark-up language text description of the architecture for use with Google's protocol buffer library. Using this language, the user can define the properties of each layer in the network to create new experimental topologies.

After the network and all of the other training parameters are selected, training may begin on one or more GPUs (or CPU) on the host computer. Then, after a variable amount of time depending on the network's topology, the number of classes to train, and the hardware; the model will cease training. At this point, the model can be downloaded and tested at each snapshot interval recorded to disk by the training system. This is useful if a case arises where the validation accuracy dips down at a later point in the training process, as evidenced when using NAG with AlexNet.

When downloading the model, many files are produced but only four are needed for deployment. First, there is the model prototxt file that defines the model architecture to the deployment library. Next, there are the weights for the model that are stored within the Caffe model file. Then, there is the mean file which corresponds to the mean of the dataset to be subtracted from each image run through the network. Lastly, there is the label file which contains all of the candidate labels for classification. These are the four files required for network deployment.

4.2 Training LeNet-5 on the 4-class Dataset

LeNet-5 was trained on the 4-class dataset with each evaluated solver type. Training graphs were produced and test validation was performed on the classifiers with the best validation accuracy. On these same networks, an inspection of the features and resulting spatial activations was also performed.

4.2.1 Adadelta Results

The first network to be evaluated was LeNet-5 with the Adadelta solver. After 500 epochs of training, the network converged to 84.89% validation accuracy over the validation set. For such a small network topology, the level of validation accuracy gained from the small amount of training data for each class was quite impressive. However, this also seems appropriate with the limited number of classes being trained upon and inherent visual differences of the areas.

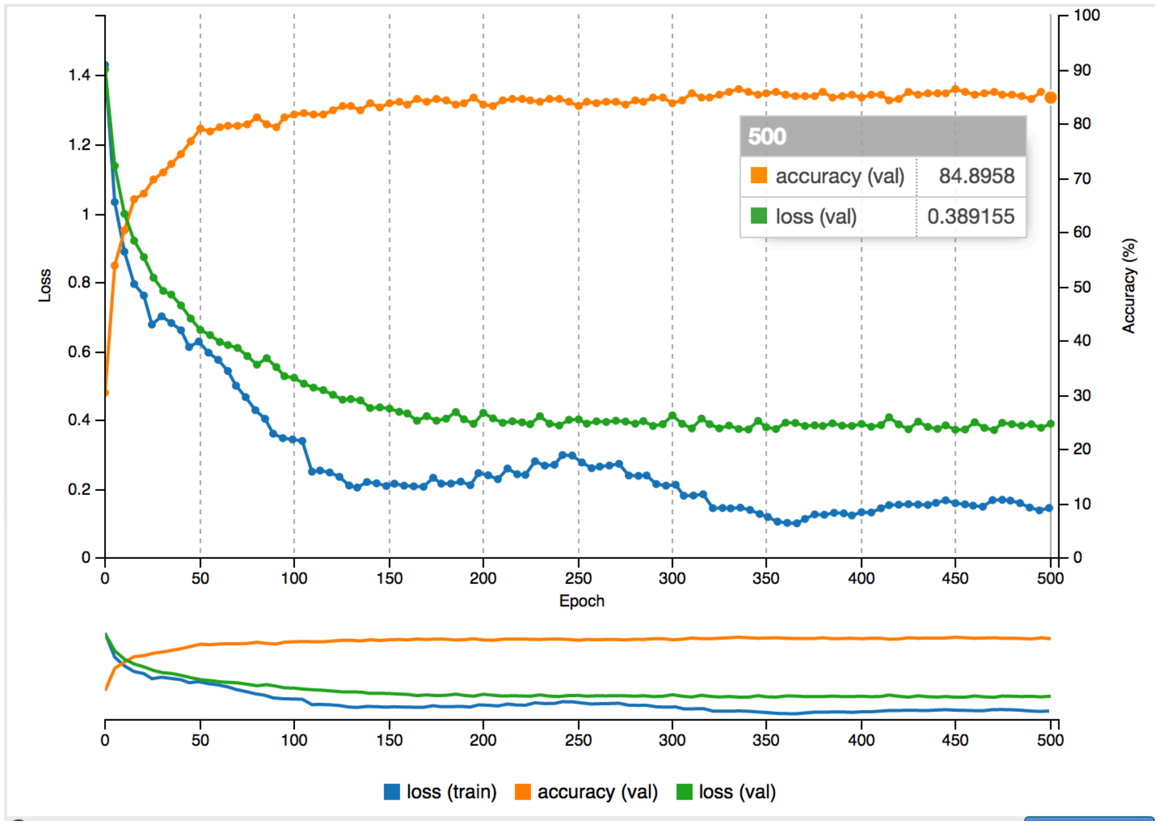


Figure 12: Adadelta Training Graph for LeNet-5 (Generated by DIGITS)

4.2.2 Adagrad Results

After only 25 epochs out of the original 500, the Adagrad algorithm was able to get up to 93% validation accuracy on classifying the five classes. This quick training time could be useful for training, deployment, and potential retraining on machines with lower processing abilities.

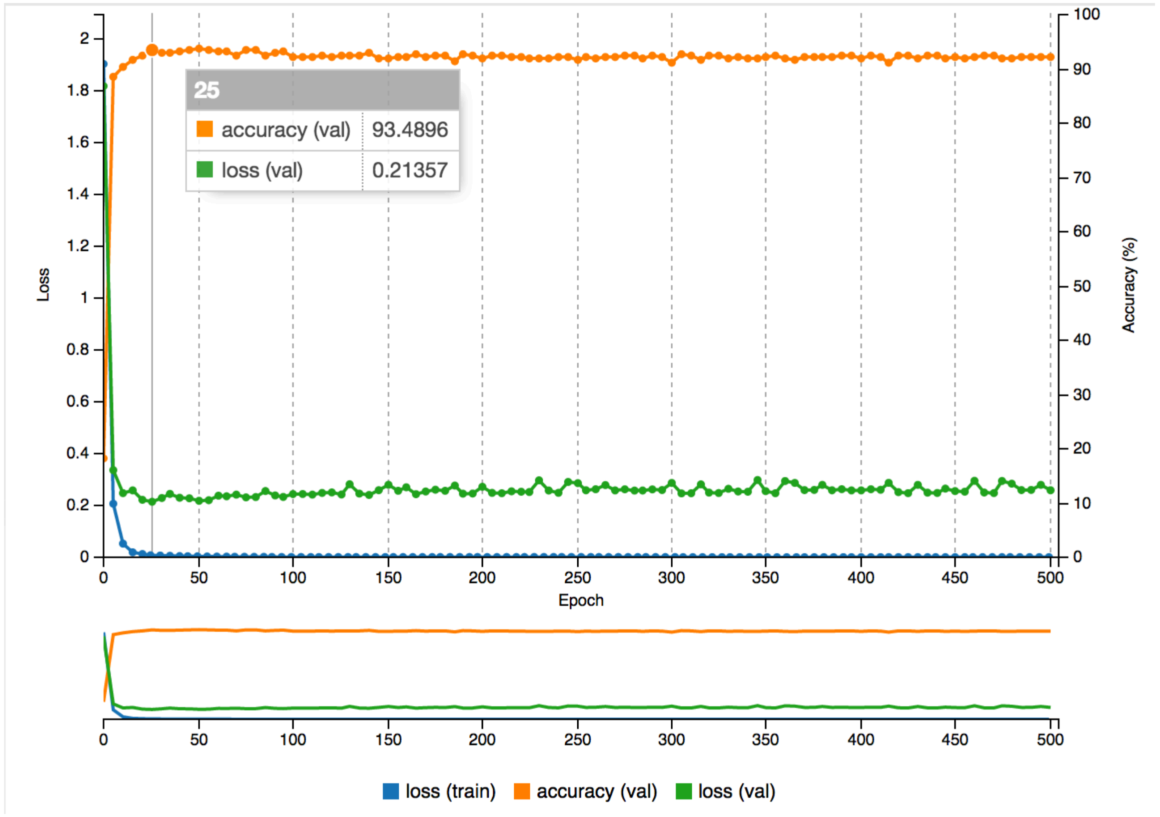


Figure 13: Adagrad Training Graph for LeNet-5 (Generated by DIGITS)

4.2.3 Adam Results

After 15 epochs of training, Adam converged to a high of 91% validation accuracy, however, after it peaked at this result, it fluctuated immensely and shot down to around 33% where is remained, steadily oscillating by about 3% until training terminated. One may assume that for such a small network when compared to AlexNet and

GoogLeNet, the implementation of adaptive moment estimation failed after 15 epochs.

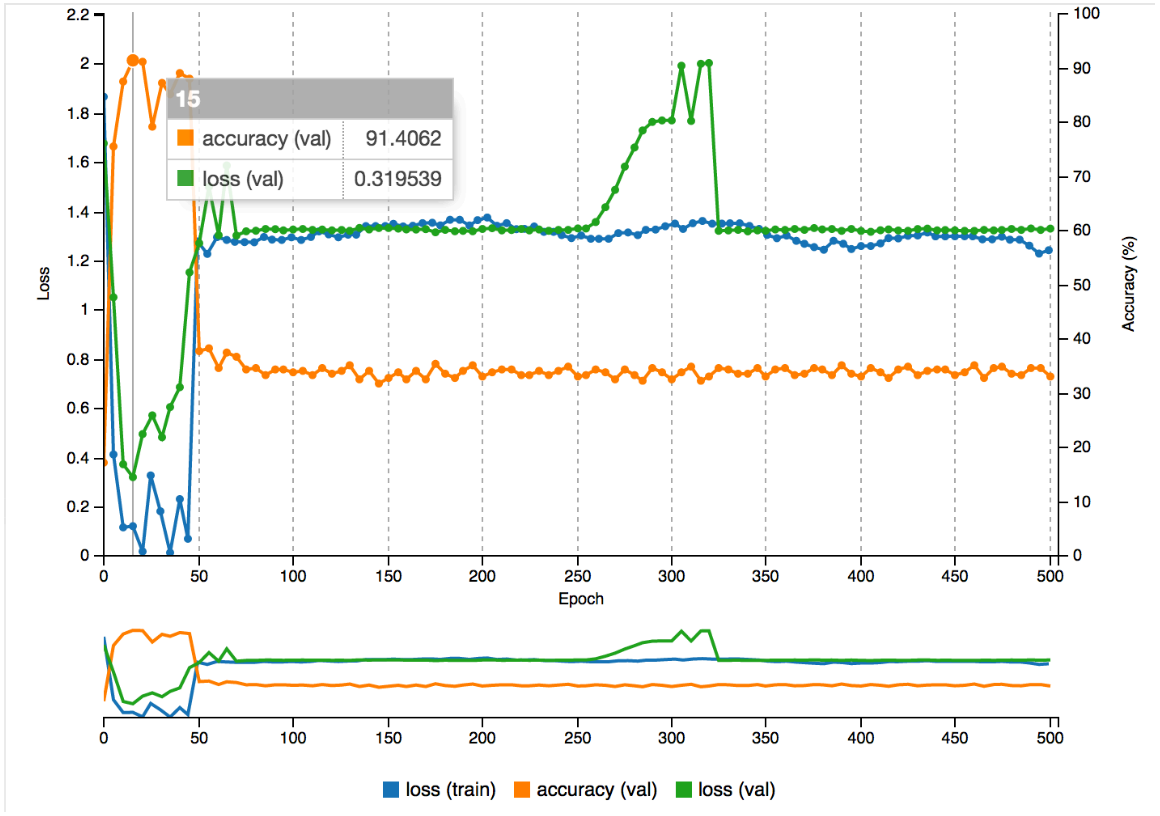


Figure 14: Adam Training Graph for LeNet-5 (Generated by DIGITS)

4.2.4 NAG Results

After 35 epochs, Nesterov's accelerated gradient descent reached 92% validation accuracy, where it remained semi-steadily until its training had finished. Like Adagrad, this quick training time could be useful for embedded applications.

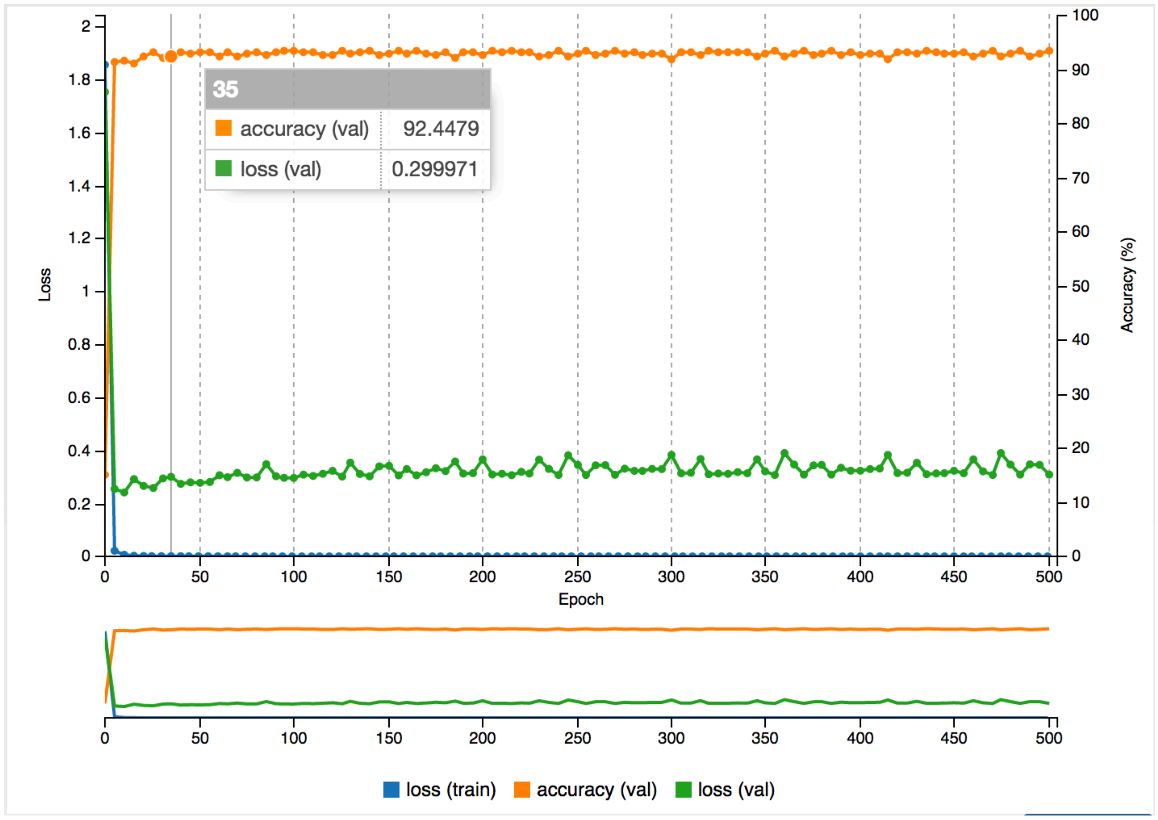


Figure 15: NAG Training Graph for LeNet-5 (Generated by DIGITS)

4.2.5 RMSProp Results

RMSProp solving the LeNet-5 architecture yielded some interesting results. For about the first 150 epochs, the accuracy climbed semi-significantly but suffered great drawbacks approximately every other iteration. After the first 150 epochs were finished, the accuracy climbed steadily, hitting what seemed to be an asymptote at around 94.5%

validation accuracy, a fantastic result for the small network.

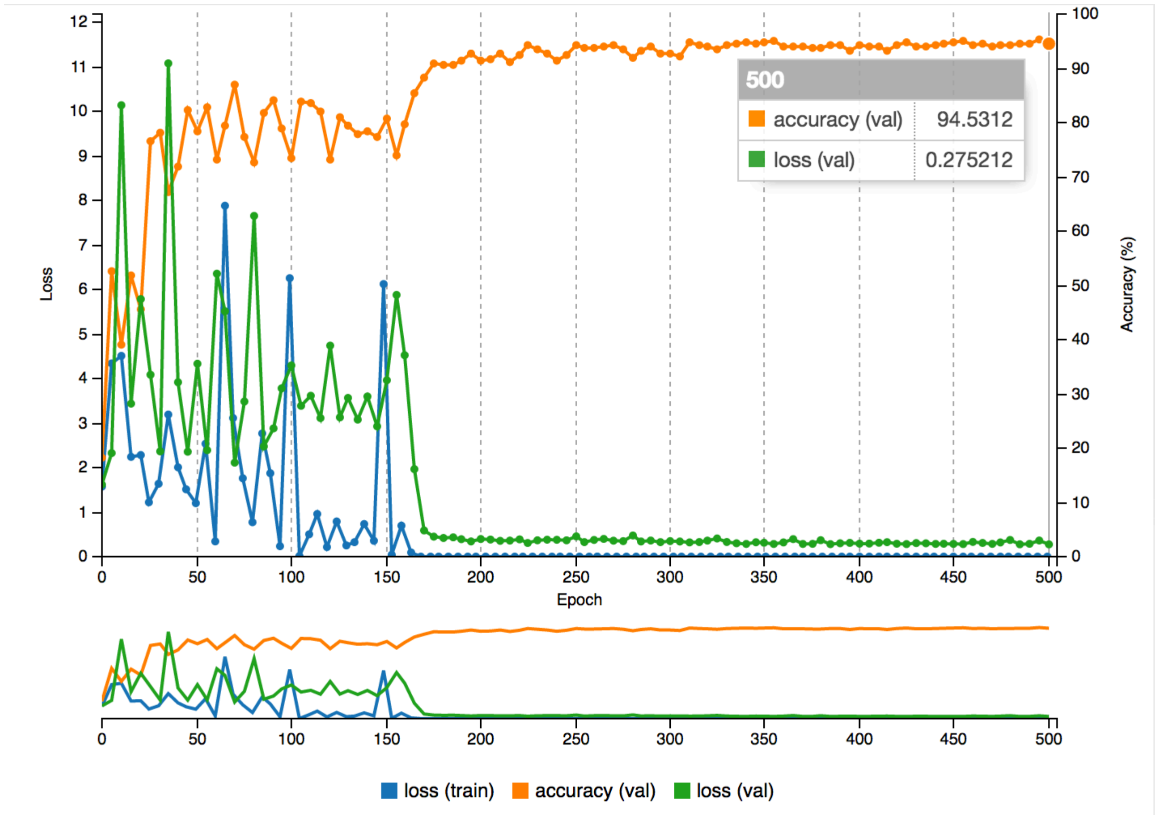


Figure 16: RMSProp Training Graph for LeNet-5 (Generated by DIGITS)

4.2.6 SGD Results

Lastly, stochastic gradient descent was used to train LeNet-5 on the 4-class data. Surprisingly, the vanilla technique was the one to perform the best – only taking 40 epochs to reach an accuracy of 95%. The rest of the training iterations left in the session only made a negligible improvement in the accuracy. This short training time matched with fantastic results would prove very useful if deployed on an embedded system.

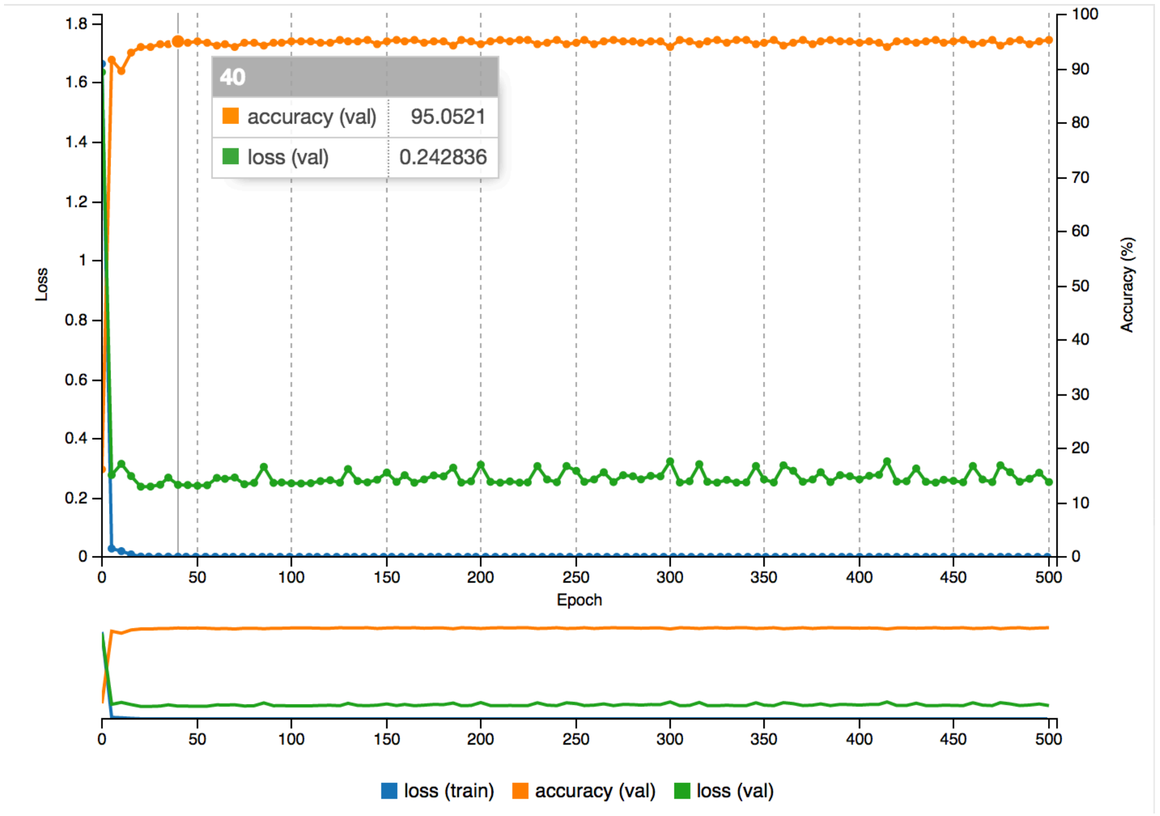


Figure 17: SGD Training Graph for LeNet-5 (Generated by DIGITS)

Since this solver produced the network with the highest validation accuracy out of all of the trials, it was run on a test set. Out of 619 images, it achieved 87.08% accuracy on classifying them correctly. A summary of the results for each category is below in table one.

LeNet-5 4-class Test Accuracy	"lab"	"office"	"machineshop"	"serverroom"
87.08%	88.90%	97.80%	84.38%	78.95%

Table 1: 4-class Test Accuracies for LeNet-5

4.2.7 SGD Features

Since stochastic gradient descent yielded the best results on LeNet-5, the weights and activations of the SGD trained network were chosen to be visualized. First, the data is scaled to the appropriate network size which corresponds to the “scaled” image. The mean is then subtracted from the image which corresponds to the “data” image. Next, the data proceeds through the convolutional layer. The image that is being passed through the network is a fellow researcher sitting on a chair in the “office” environment. The red areas shown in the activation images correspond to the parts of the data input with the highest levels of excitement from the features in the convolutional layer.

After the output of the first convolutional layer has been downsampled by a pooling layer, weights from the second convolutional layer activate different areas of the input, furthering the classification process. Next, the data goes through two inner product layers that are then followed by a softmax classification layer. In the visualization, the blue square of the softmax layer corresponds to a correct result of the “office” label.

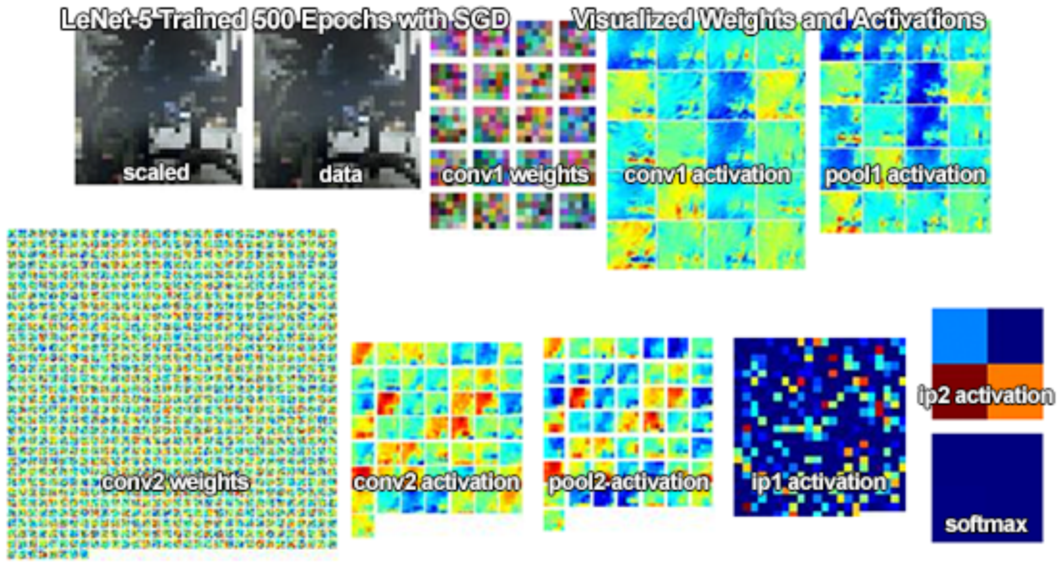


Figure 18: SGD-derived Features in LeNet-5 For an Image in the "office" Environment

A second image with the class of "lab" was run through the same trained network. It was correctly labeled. The network visualizations are provided for an additional reference.

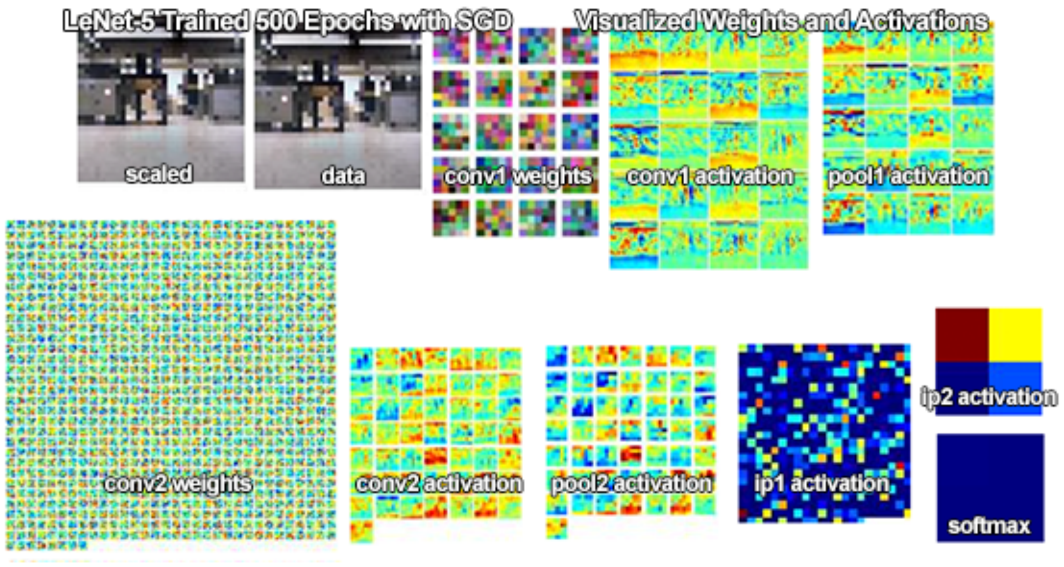


Figure 19: SGD-derived Features in LeNet-5 For an Image in the "lab" Environment

4.3 Training AlexNet on the 4-class Dataset

Training AlexNet effectively proved to be more challenging than LeNet-5. Longer training times were required to achieve results that were oftentimes not as good as LeNet-5's. Furthermore, some of the implementations of the different solvers simply did not work with this architecture. The longer training times and worse validation performance on the much smaller database may be explained by the fact that AlexNet was originally designed to classify a large range of different images and a much more topologically complex network intended to capture deeper features than perhaps required.

4.3.1 Adadelta Results

Adadelta created a good asymptotic convergence pattern that approached its final result of 85%. There were no hiccups along the way, but given the size of the network and the amount of time spent training, the level of validation accuracy was disappointing.

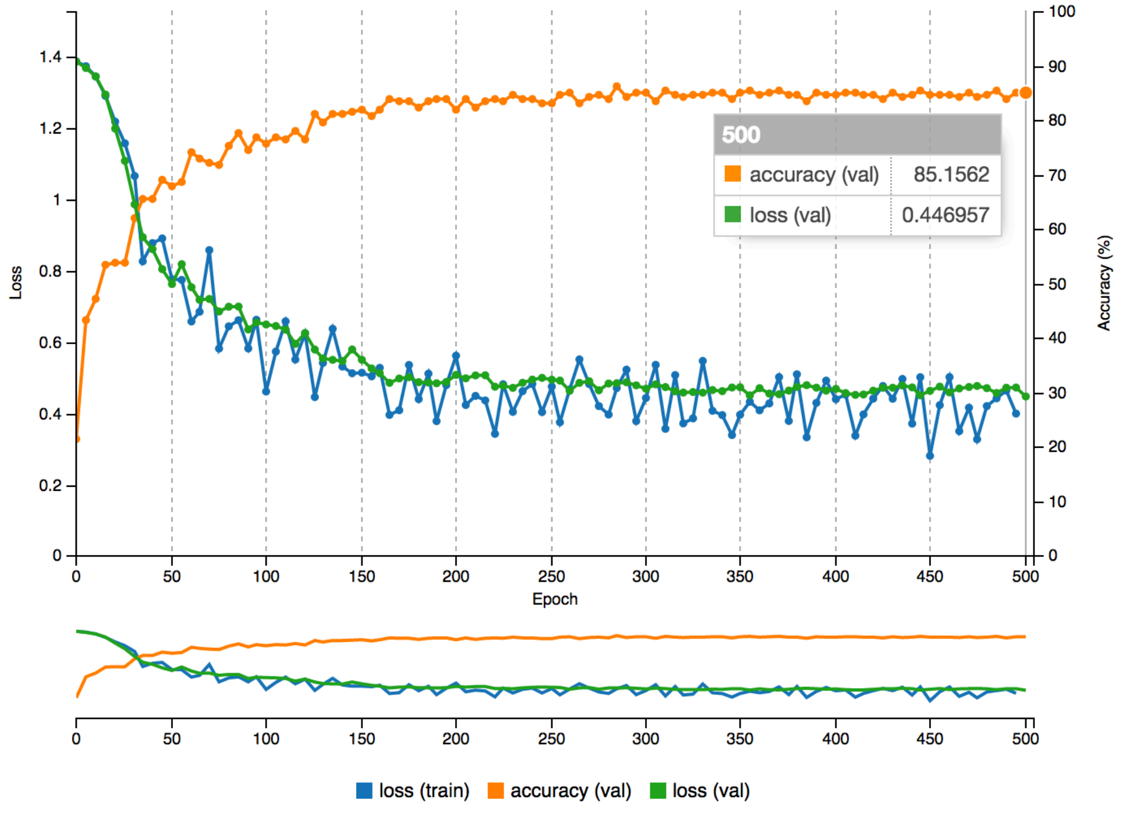


Figure 20: Adadelta Training Graph for AlexNet (Generated by DIGITS)

4.3.2 Adagrad Results

Adagrad did not perform well at all when attempting to solve the weights for AlexNet. The algorithm started seemingly well at an initial 'guess' accuracy level of slightly about 20% but failed to make much progress at all, inching up over the span of

500 epochs to a meager 65% validation accuracy.

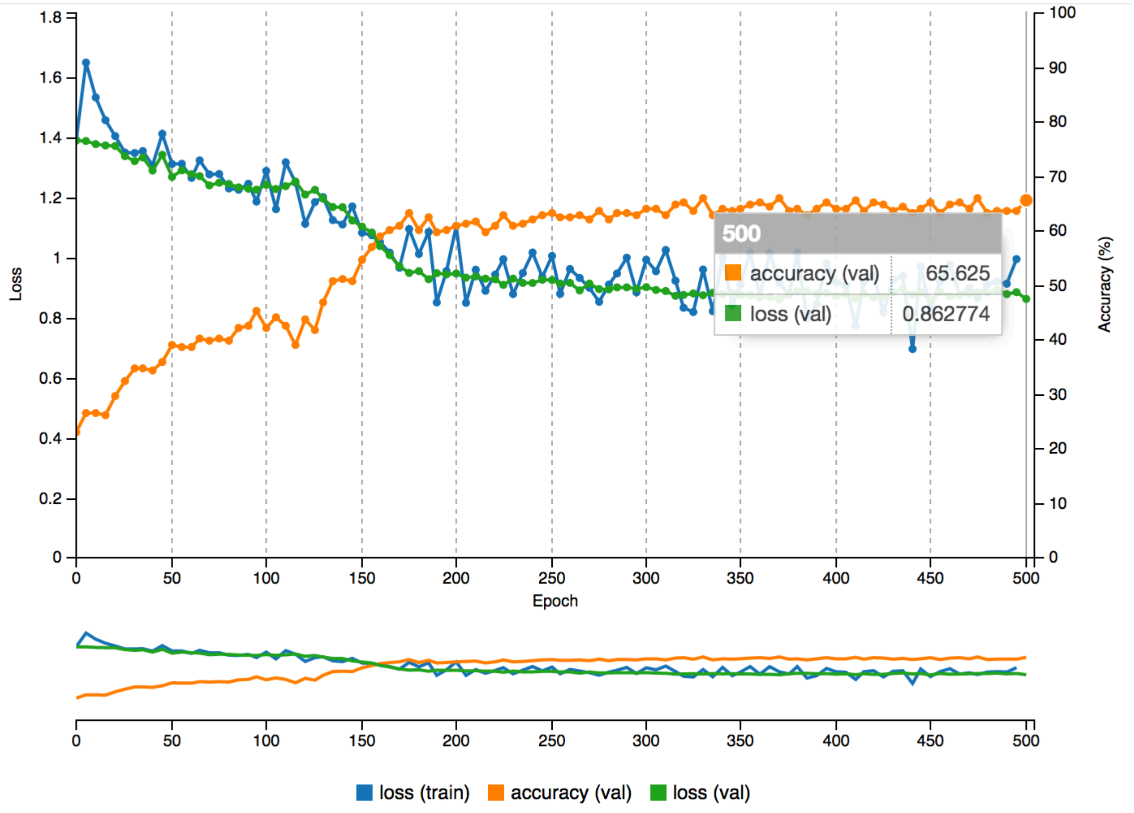


Figure 21: Adagrad Training Graph for AlexNet (Generated by DIGITS)

4.3.3 Adam Results

The implementation of Adam used by Caffe did not work with AlexNet at all and remained around the initial random seed value of 25% for the entirety of training.

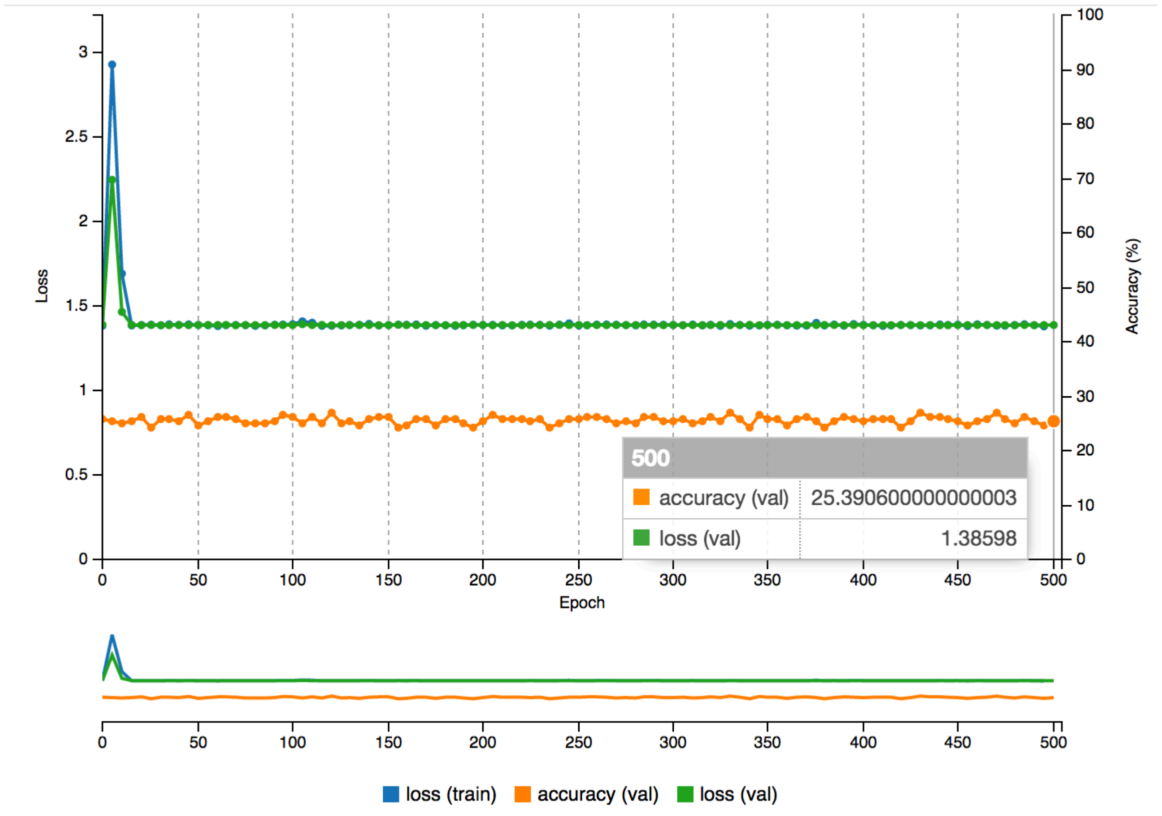


Figure 22: Adam Training Graph for AlexNet (Generated by DIGITS)

4.3.4 NAG Results

Nesterov's performed rather oddly on AlexNet. The training started off rather well, rapidly increasing to a peak of 95% at epoch 60%, a good indication of successful learning. However, immediately after the 60th iteration, the validation accuracy tanked down to the original random initialization value and stayed there, fluctuating for the remainder of the epochs. This may have been because of an implementation bug with using NAG on AlexNet.

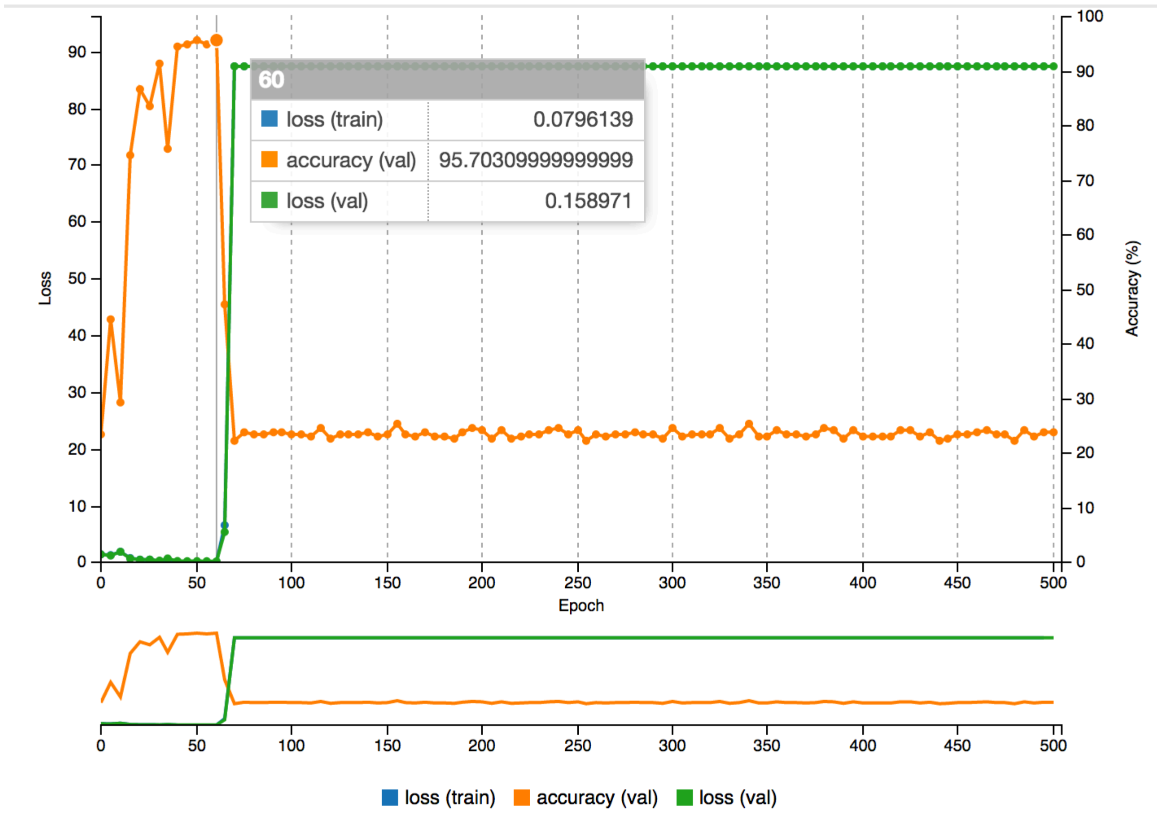


Figure 23: NAG Training Graph for AlexNet (Generated by DIGITS)

4.3.5 RMSProp Results

The results of using the RMSProp solver on AlexNet were identical to those of Adam, with the validation accuracy continually fluctuating around the initial seed value for the entirety of the training session.

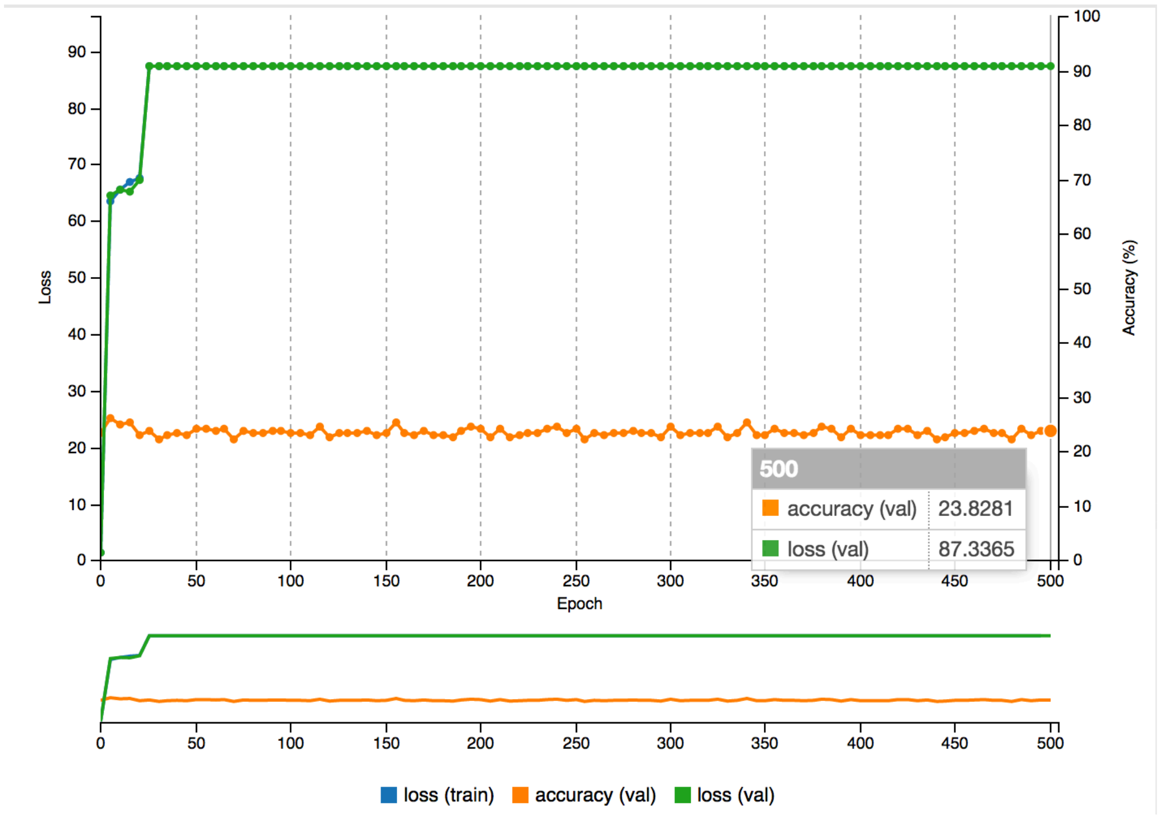


Figure 24: RMSProp Training Graph for AlexNet (Generated by DIGITS)

4.3.6 SGD Results

Once again, this solver did the best by achieving a validation accuracy of 96.48% after 160 iterations. In fact, the model converged to a high result very quickly, jumping up to above 90% within the first 50 epochs. After the 500 epochs were finished, training ended with a final validation accuracy of 96.875%.

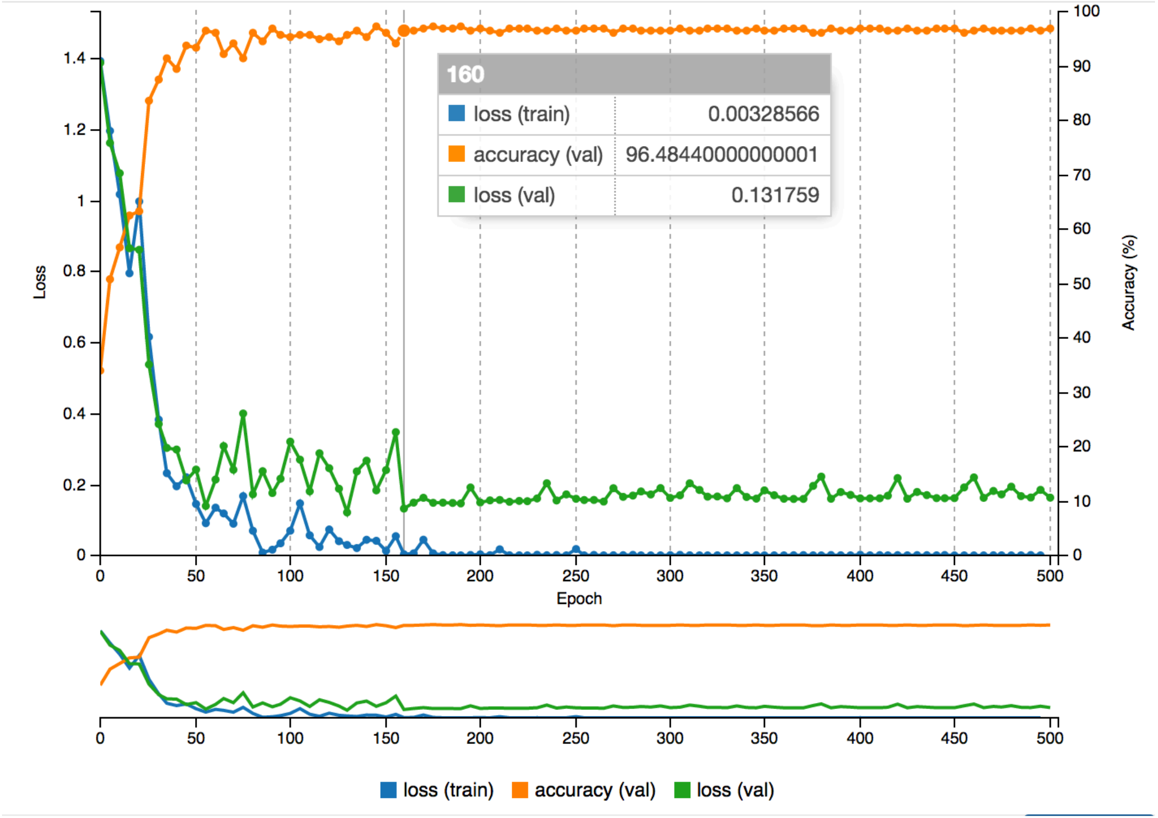


Figure 25: SGD Training Graph for AlexNet (Generated by DIGITS)

Since this solver produced the network with the highest validation accuracy out of all of the trials, it was run on a test set. Out of 619 images, it achieved 94.03% accuracy on classifying them correctly. A summary of the results for each category is below in table two.

AlexNet 4-class Test Accuracy	"lab"	"office"	"machineshop"	"serverroom"
94.03%	93.20%	98.94%	93.75%	85.53%

Table 2: 4-class Test Accuracies for AlexNet

4.3.7 SGD Features

Since SGD generated the best results from AlexNet, the features and activations that it found and generated will be examined. From the data layer, the picture with the correct class label of 'office' is passed through the first convolutional layer, activating weights corresponding to patterns in the image. Those are then normalized, pooled, and passed through a second convolutional layer – resulting in more activations. This process is repeated again, followed by two final convolutional layers. Lastly, pooling is performed and that result is fed into a fully connected layer. Not pictured is the final softmax classifier that outputs the correct class label for the input.

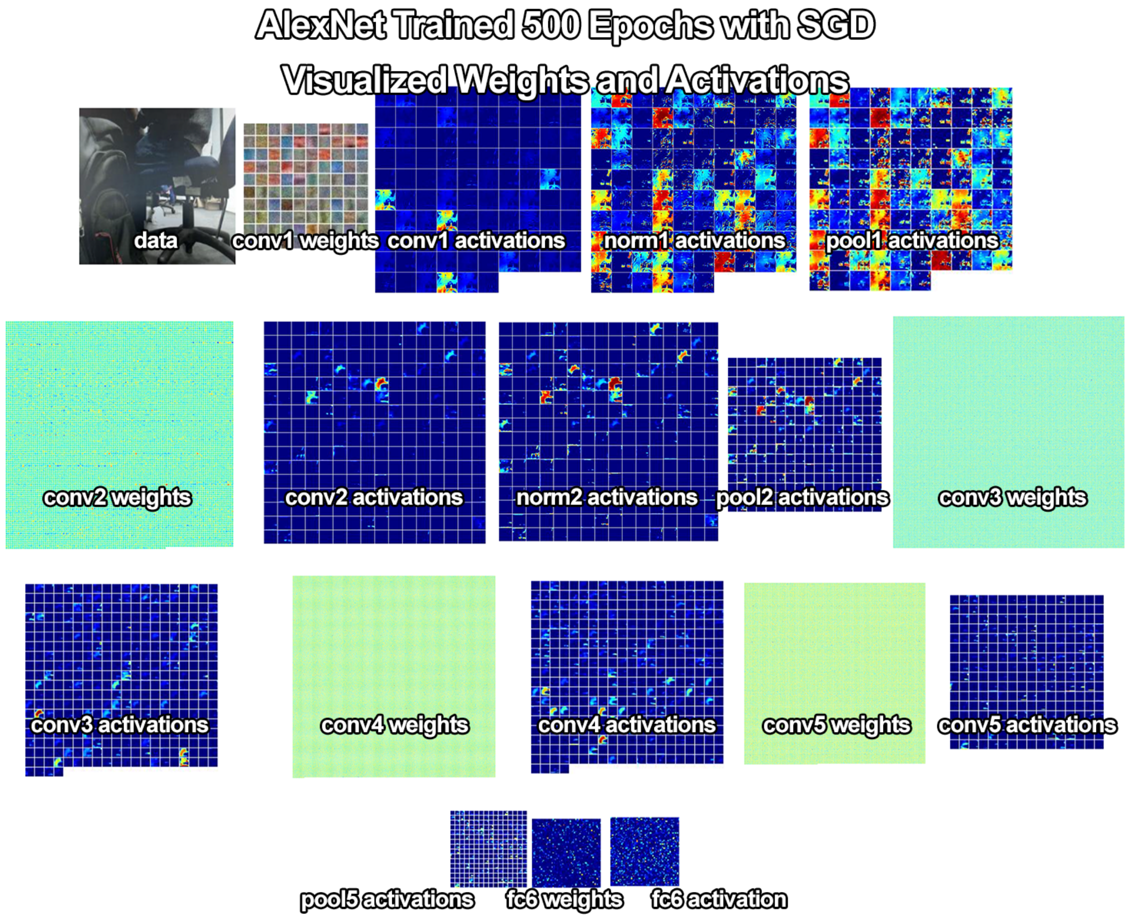


Figure 26: SGD-derived Features in AlexNet For an Image in the "office" Environment

A second piece of input data with the correct label of 'lab' is provided for further visualization purposes.

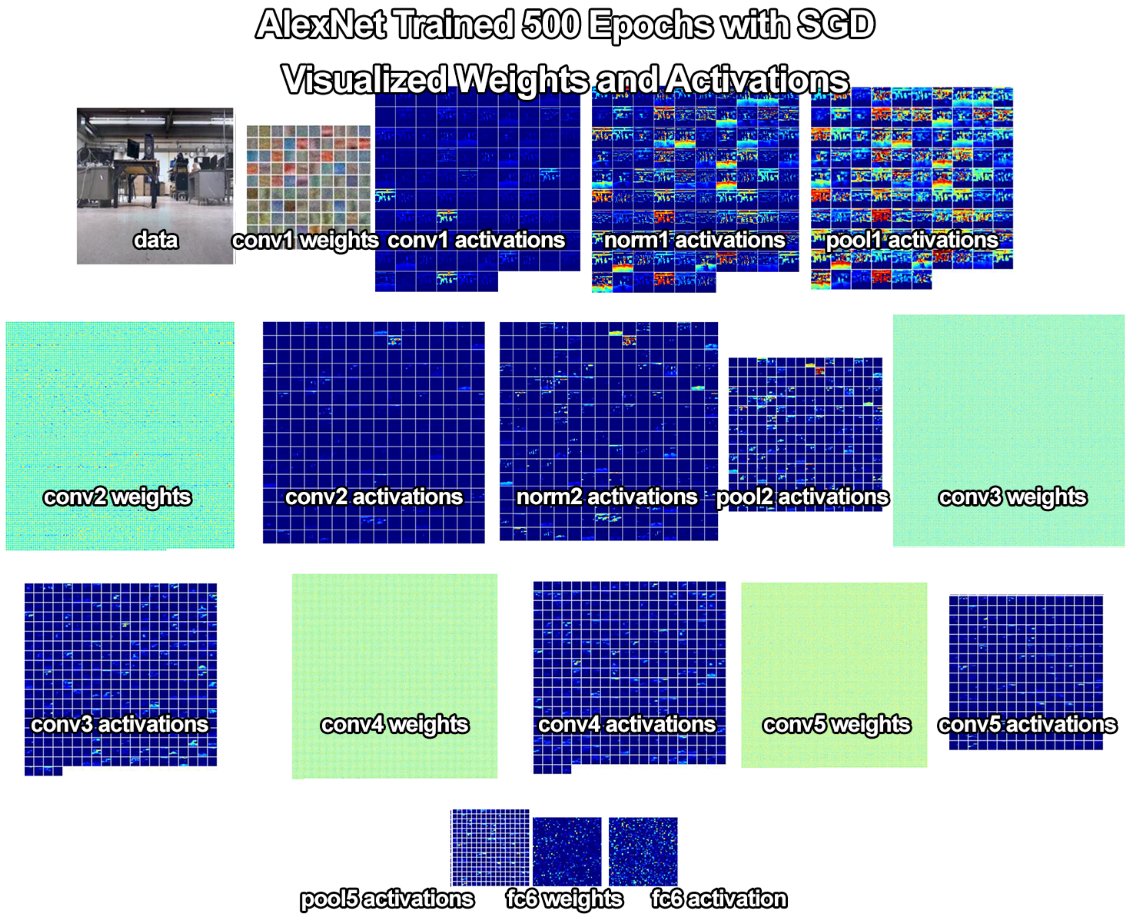


Figure 27: SGD-derived Features in AlexNet For an Image in the "lab" Environment

4.4 Training GoogLeNet on the 4-class Dataset

Training GoogLeNet took time, but the results were promising. The power of the deeply learned features encoded in GoogLeNet's layers were able to bring the classifier to some of the highest accuracy levels yet. However, like AlexNet, some of the solvers, Adam and RMSProp, failed to converge on any solution. This is likely due to an implementation issue with the Caffe deep learning library. The long training times required for GoogLeNet in coordination with its gigantic topology indicate that it would be capable of learning more classes. For only four classes, applying GoogLeNet may be more than required, but the accuracy levels of the results are still spectacular.

4.4.1 Adadelta Results

Adadelta allowed for GoogLeNet to achieve 90.00% validation accuracy after about 175 epochs. The training accuracy approached the end result steadily and asymptotically.

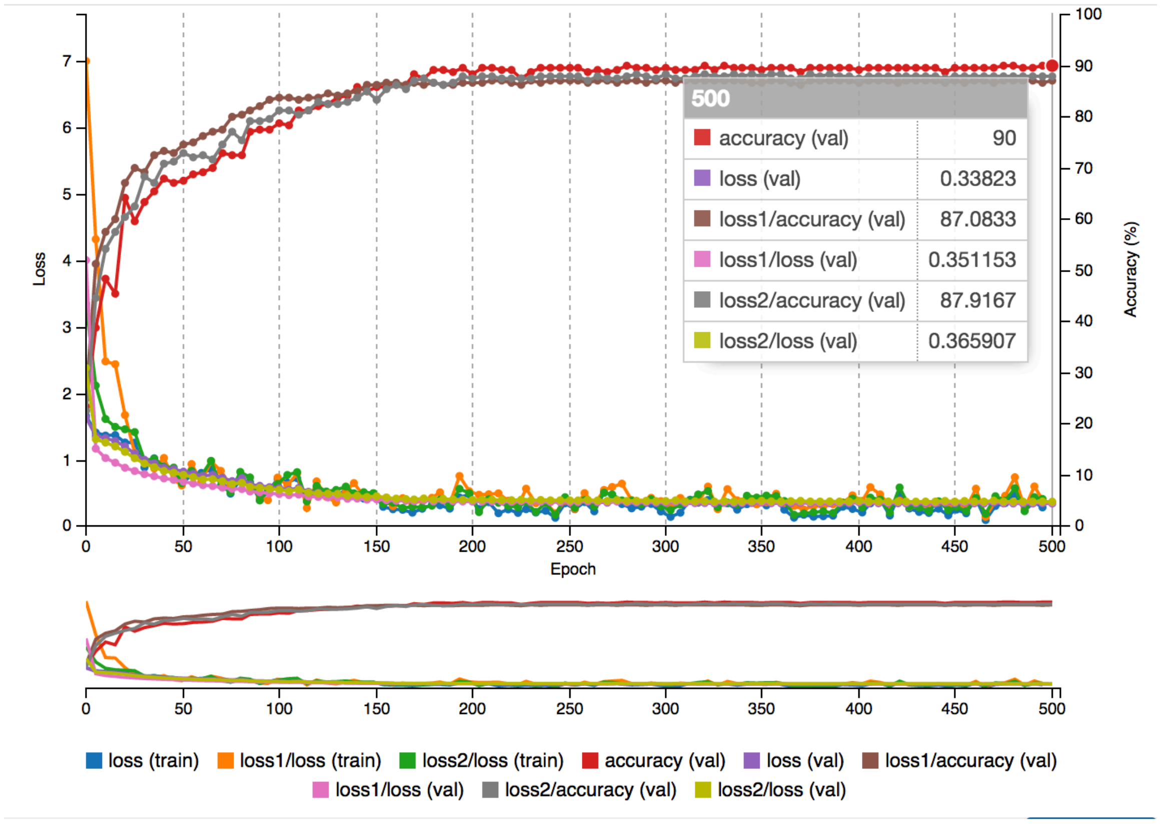


Figure 28: Adadelta Training Graph for GoogLeNet (Generated by DIGITS)

4.4.2 Adagrad Results

In only 165 epochs, Adagrad was able to achieve 95.41% validation accuracy. The other iterations did not make any improvements to the already stellar performance of the classifier.

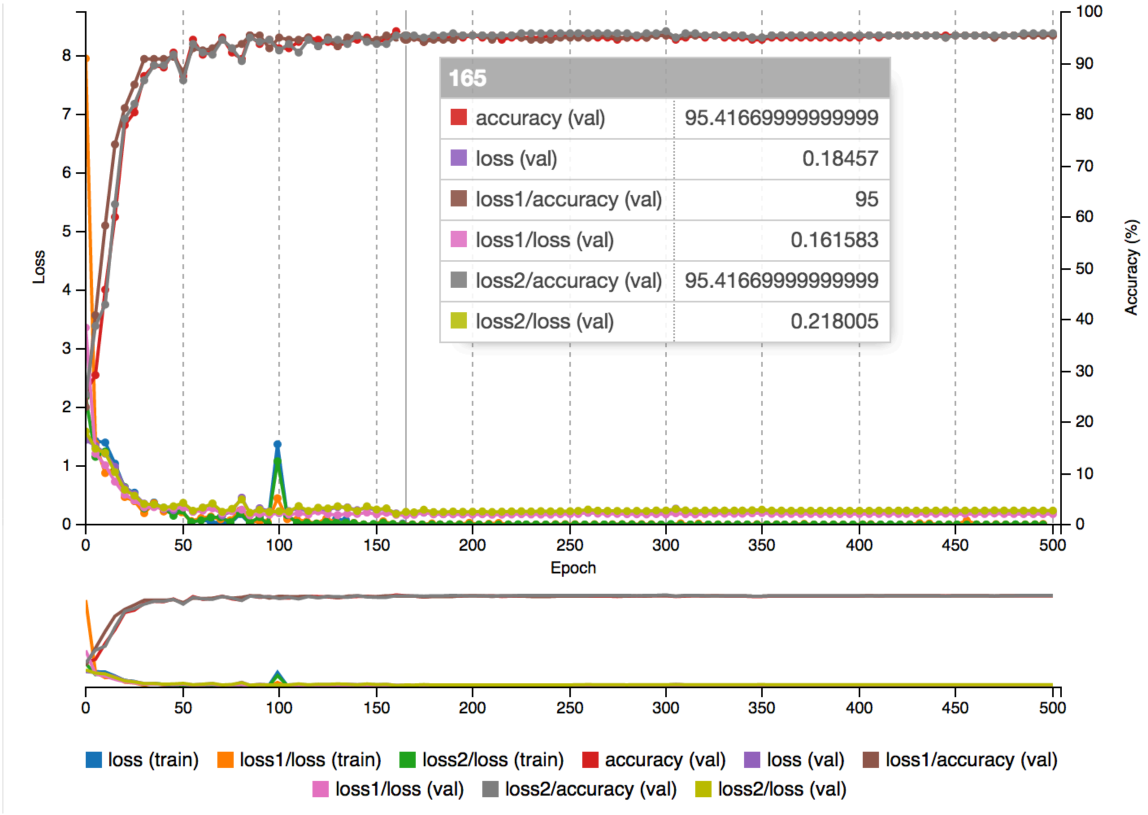


Figure 29: Adagrad Training Graph for GoogLeNet (Generated by DIGITS)

4.4.3 Adam Results

Adam failed to converge on any solution and merely fluctuated around the initial random seed value.

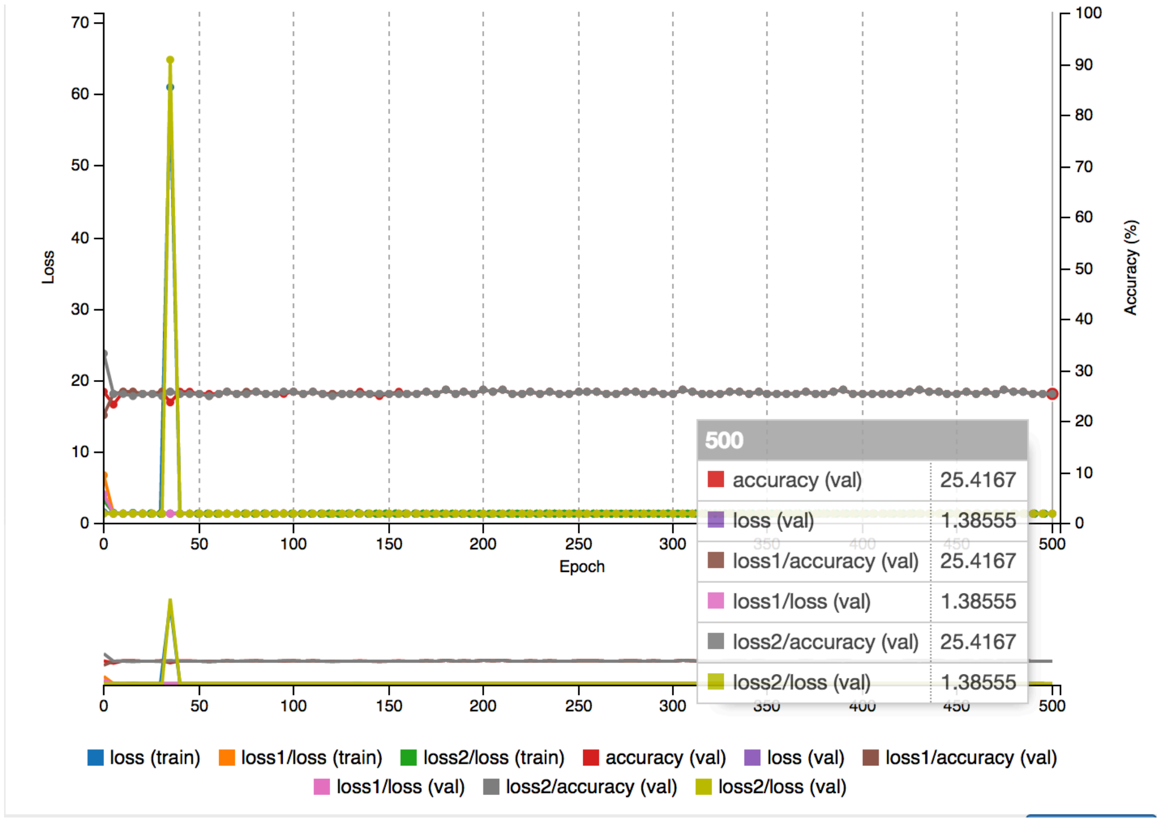


Figure 30: Adam Training Graph for GoogLeNet (Generated by DIGITS)

4.4.4 NAG Results

Nesterov's approximated gradient descent method achieved a very high result of 97.08% after only 165 iterations. Like the previous solver technique, the rest of the epochs made no difference in the level of validation accuracy.

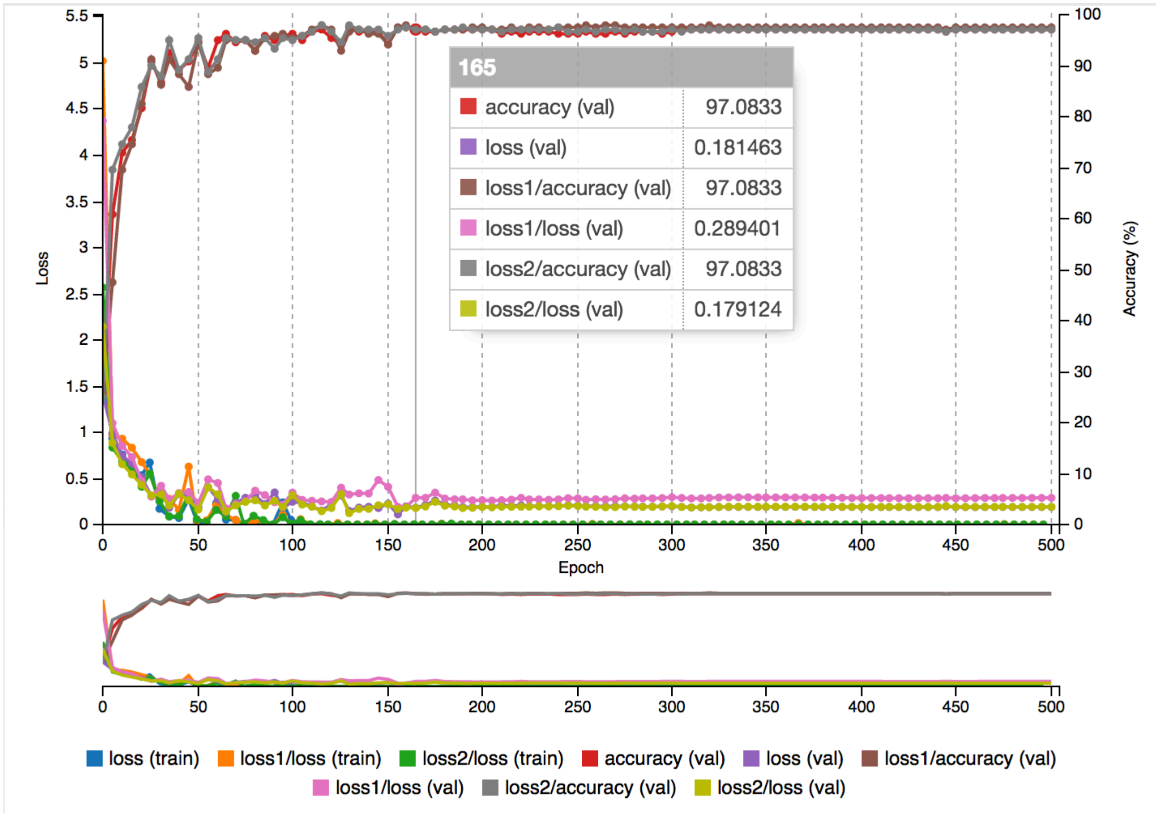


Figure 31: NAG Training Graph for GoogLeNet (Generated by DIGITS)

4.4.5 RMSProp Results

Like Adam and as mentioned previously, RMSProp failed to converge on any solution.

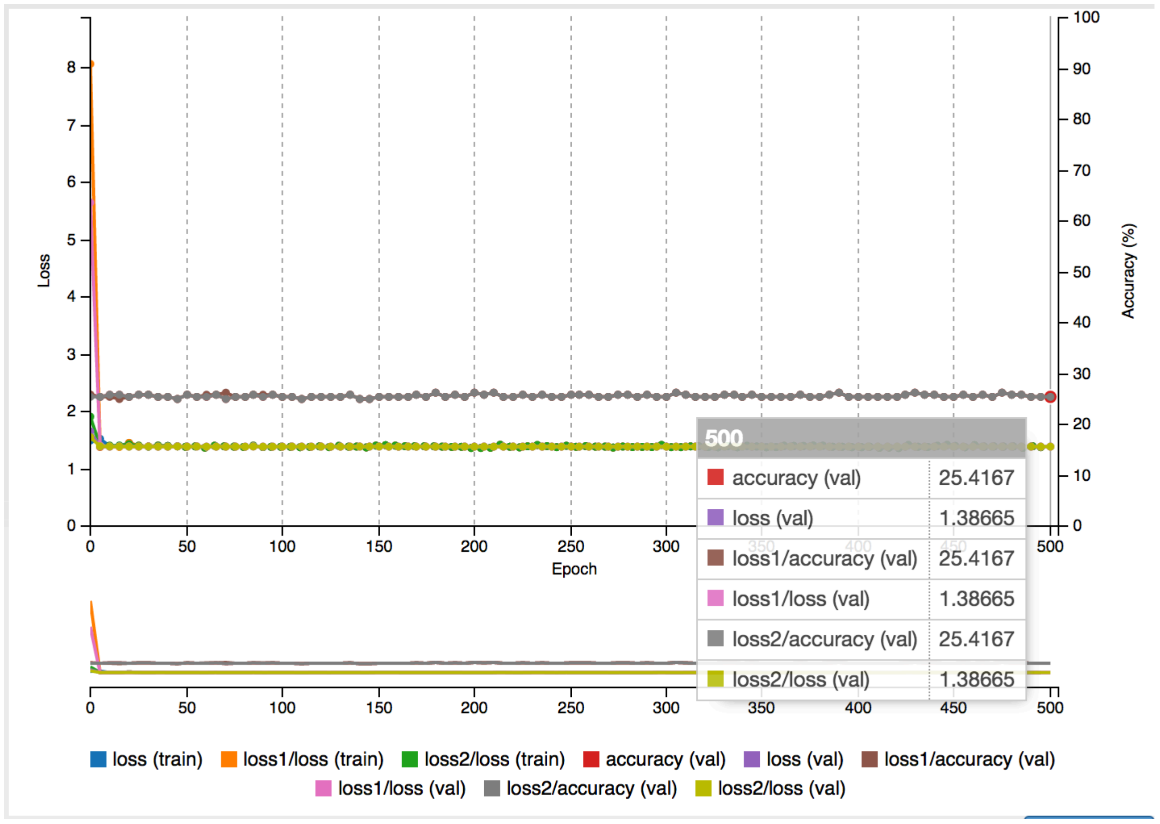


Figure 32: RMSProp Training Graph for GoogLeNet (Generated by DIGITS)

4.3.6 SGD Results

After 305 iterations, SGD achieved the same level of validation accuracy as the NAG solver, 97.08%. However, as the iterations progressed to 500, the level of validation accuracy reached the highest rate yet – 97.91%.

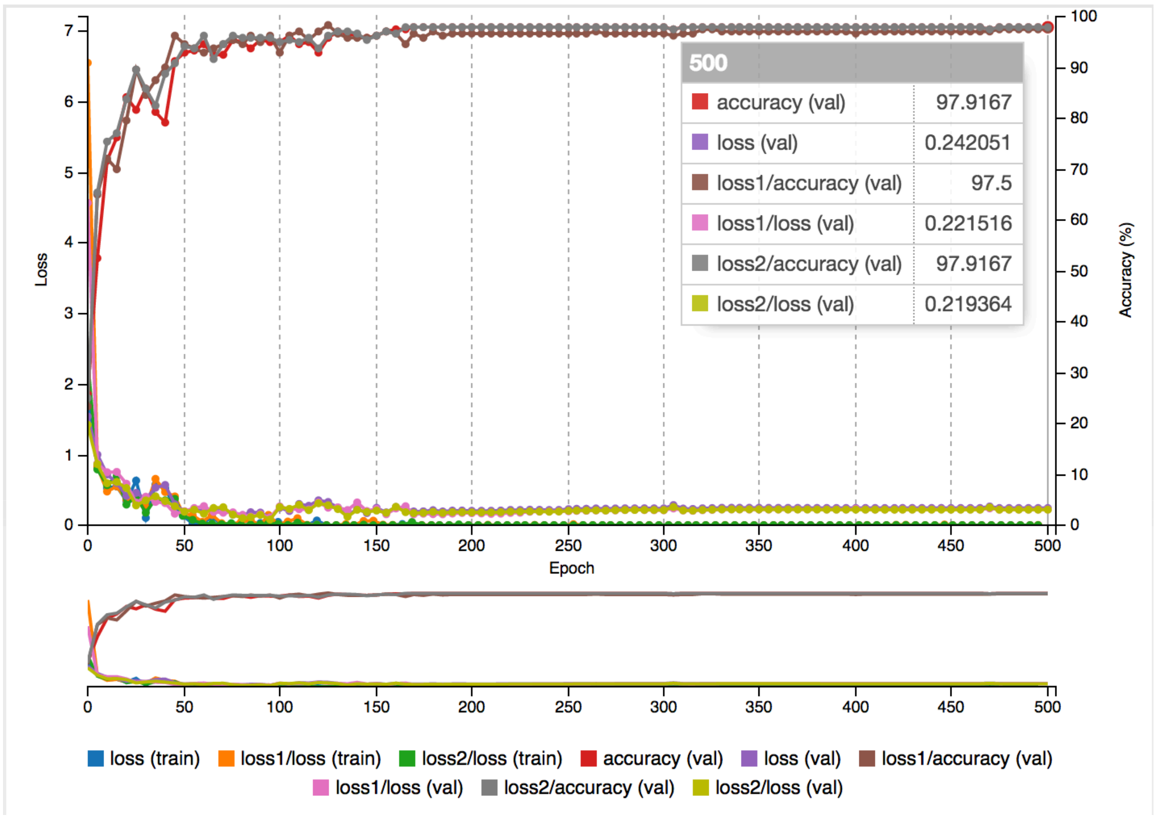


Figure 33: SGD Training Graph for GoogLeNet (Generated by DIGITS)

Since this solver produced the network with the highest validation accuracy out of all of the trials, it was run on a test set. Out of 619 images, it achieved 89.34% accuracy on classifying them correctly. A summary of the results for each category is below in table three.

GoogLeNet 4-class Test Accuracy	"lab"	"office"	"machineshop"	"serverroom"
89.34%	91.96%	97.35%	90.62%	93.43%

Table 3: 4-class Test Accuracies for GoogLeNet

4.4.7 SGD Features

Like the other two networks tested, SGD yielded the best results out of any solver type. Since GoogLeNet is so massive, displaying all of the visualized features and activations would be impractical. Generally, they take on the same form as the other convolutional layers in the other networks. However, the greater number of features and the millions of learned hyperparameters allow for the network to encode a greater amount of learned information and to create a more complex decision surface in the parameter space.

4.5 Results of the 4-class Dataset and Discussion

GoogLeNet, by far, had the best validation results out of any other neural network tested on the 4-class dataset. The 97.91% overall validation accuracy is a very high result that speaks to the power of the neural network used. However, AlexNet was able to surpass GoogLeNet in three out of the four evaluated classes with test accuracy, averaging out at an overall test accuracy of 94.03%, about five percent higher than GoogLeNet's at 89.34%. It seems that at least for the test set used to benchmark the classifier, along with the number of classes used, AlexNet is the best option.

One may ask the question of whether or not getting such a high accuracy matters so much for the application at hand. As long as the accuracy is somewhat high, and the user can assume that the robot is not on the edge of two rooms, an algorithm could be used to run multiple images through the classifier and cast a weighted vote on the result. Multiple images could be recorded by the camera at different rotations – a safe movement to make for a differential drive robot with a circular chassis. By using this voting method, more accurate results to be used to assist the localization problem may be acquired.

Even though AlexNet had the best results, it still took a significant amount of time to train, especially when compared to LeNet-5. Since LeNet-5 was able to converge to a result that allowed for it to correctly identify 87.08% of test examples correctly in only two minutes, the network may be able to provide close to the level that AlexNet can provide with a small fraction of time spent training.

For training the neural network, a common GPU, the NVidia GeForce GTX 745 was used. Training time for the 4-class dataset on LeNet-5 took only two minutes to converge with all of the solvers used. For AlexNet, it took about three hours, and for GoogLeNet, it took about four hours.

When comparing these different algorithms and keeping in mind that the level of accuracy, as long as it is within some acceptable threshold, it seems that the best "bang for your buck" may come out of re-using LeNet-5 for the small number of classes needed. This small network holds a great deal of power given the small number of classes to be differentiated between. It had the best accuracy to training time ratio over the other networks.

Name	Training Time	Solver	Test Accuracy	Accuracy to Training Ratio
LeNet-5	2 minutes	SGD	87.08%	87.08 / 2 = **43.54**
AlexNet	180 minutes	SGD	94.03%	94.03 / 180 = **0.522**
GoogLeNet	240 minutes	SGD	89.34%	89.34 / 240 = **0.372**

Table 4: Performances of Different Network Topologies on the 4-class Dataset

Even though LeNet-5 has the best characteristics for few classes, it could be inferred that when adding more classes, and therefore creating a more difficult to find decision surface, the other networks would perform better.

4.6 Testing the 7-class Dataset

A new dataset was created in the same fashion as the previous ones to include the classes of "lab", "machineshop", "serverroom", "office", "entryway", "circuitslab", and "lounge". This was to test how the networks reacted to a more complex decision surface being generated to achieve acceptable accuracy. The three networks, LeNet-5, AlexNet, and GoogLeNet we all tested using SGD as it produced the best results for the 4-class data.

For testing the networks, a test set was created by driving the Turtlebot around to each that was within a drive-able region. For regions that were upstairs, the robot was kidnapped (picked up), and brought to the areas. This set was acquired in the same fashion as the training set, but with one continuous bag file.

4.6.1 Training LeNet-5 on the 7-class Dataset Using SGD

LeNet-5 was able to converge to 92.65% validation accuracy after only 40 iterations. Throughout its 500-epoch training time, it did not make any improvements past this value. With adding only 3 more classes, the decrease in accuracy was only about 3%. Training time was also very quick, only lasting about 30 minutes on the same graphics card that was used to train the other networks.

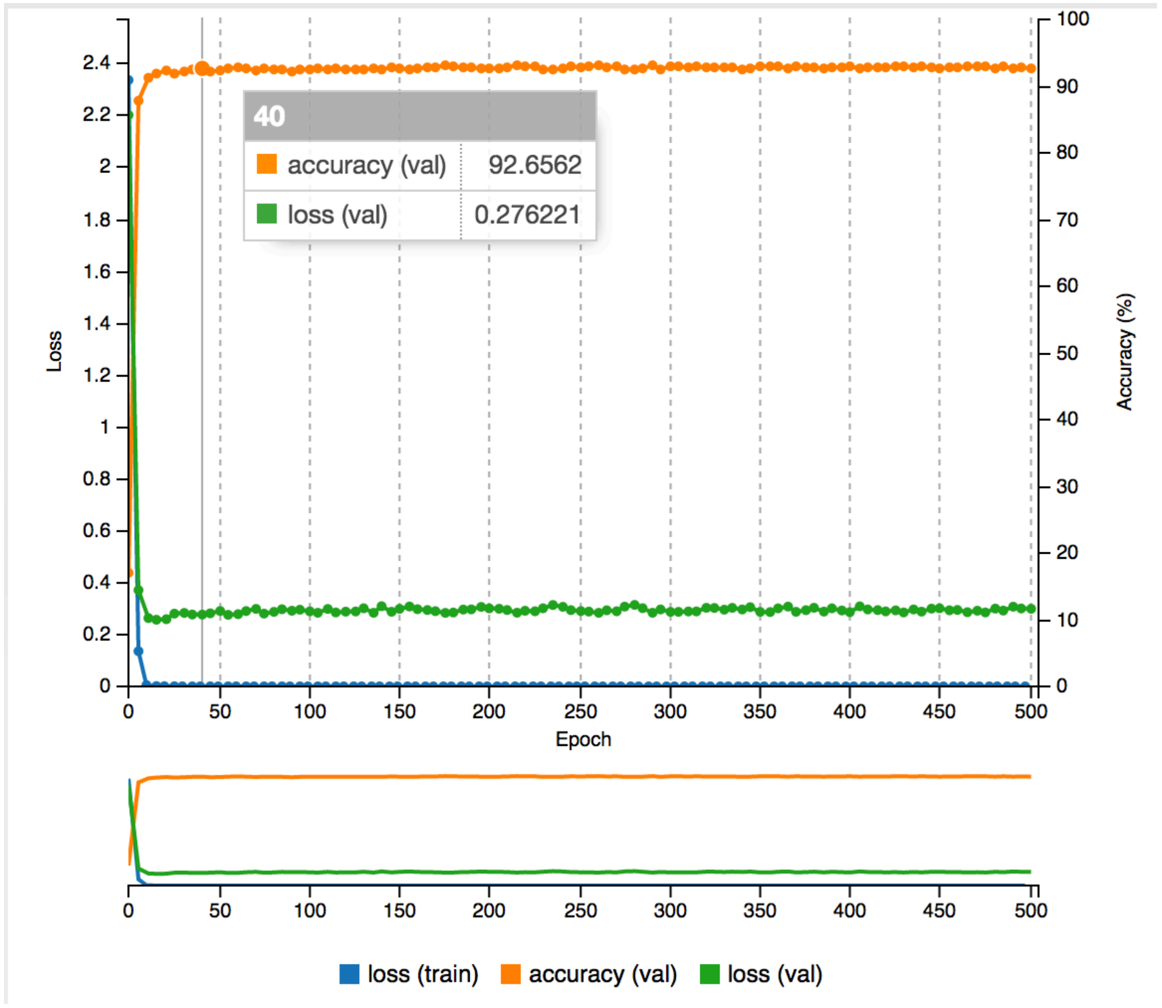


Figure 34: 7-class SGD Training Graph for LeNet-5 (Generated by DIGITS)

Since this was the only solver type used to train LeNet-5 with the 7-class dataset, the classifier was tested against a separate test set. It did not perform quite well, especially when compared to the 4-class dataset. On the test set, it achieved an overall accuracy of 60.43%. This likely had to do with the increased number of classes, as well as the construction of the test set. A complete breakdown of how the classifier performed against all of the different classes is below in table five.

LeNet-5 7-class Test Accuracy	60.43%
“lab”	91.21%
“office”	100%
“machineshop”	83.49%
“serverroom”	75.0%
“lounge”	79.92%
“circuitslab”	37.27%
“entryway”	70.96%

Table 5: 7-class Test Accuracies for LeNet-5

4.6.2 Training AlexNet on the 7-class Dataset Using SGD

As expected, AlexNet performed better than previously when more encode-able information was provided to the network. With four classes, AlexNet achieved an accuracy of around 96% after around 160 iterations. With the additional three classes, AlexNet performed even better, attaining a result of 98.125% after only 100 iterations. For the rest of the 400 out of the 500 total training iterations, the network converged to a final validation accuracy of 98.59%.

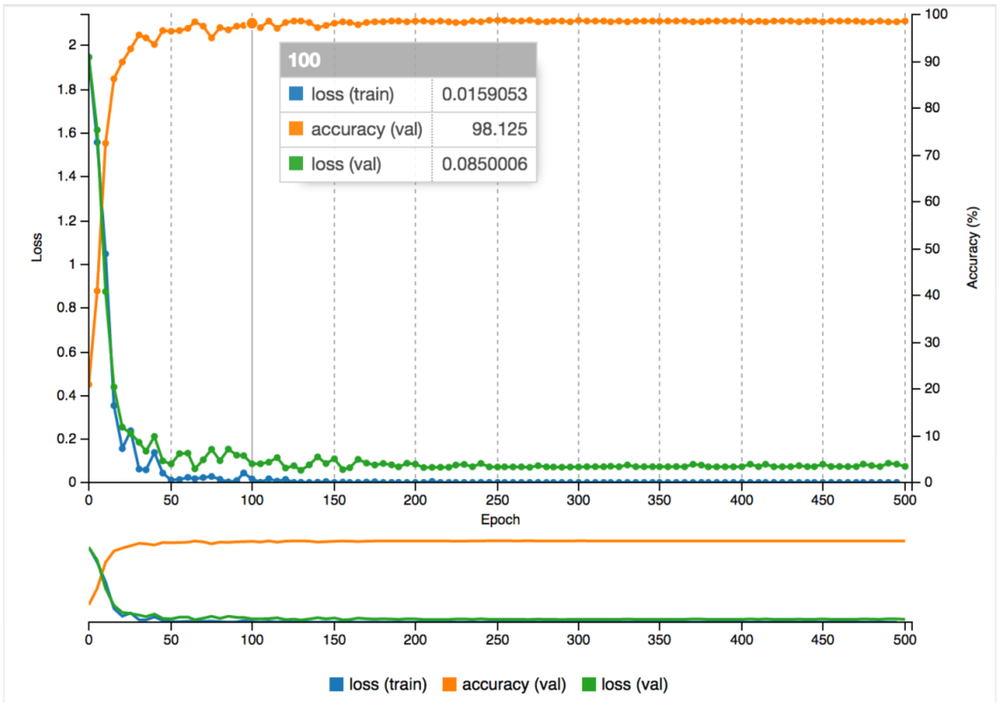


Figure 35: 7-class SGD Training Graph for AlexNet (Generated by DIGITS)

On the same test set for the 7-class data, AlexNet achieved a much better result of 84.85% test accuracy. This likely was because of its deeper architecture, and ability to make more complex of a decision surface within the feature space. A summary of how AlexNet performed against all of the different classes in the dataset is below in table six.

AlexNet 7-class Test Accuracy	84.85%
"lab"	93.91%
"office"	100%
"machineshop"	97.29%
"serverroom"	84.61%
"lounge"	88.84%
"circuitslab"	43.63%
"entryway"	98.38%

Table 6: 7-class Test Accuracies for AlexNet

4.6.3 Training GoogLeNet on the 7-class Dataset Using SGD

Like AlexNet, the larger network topology of GoogLeNet allowed for more descriptive information to be encoded within the weights of the network. The more complex features that were discovered to differentiate between the seven classes were able to allow GoogLeNet to achieve the fantastic level of 99.19% validation accuracy after only 55 epochs. The rest of the iterations made only a small difference, with the end result being 99.36% validation accuracy.

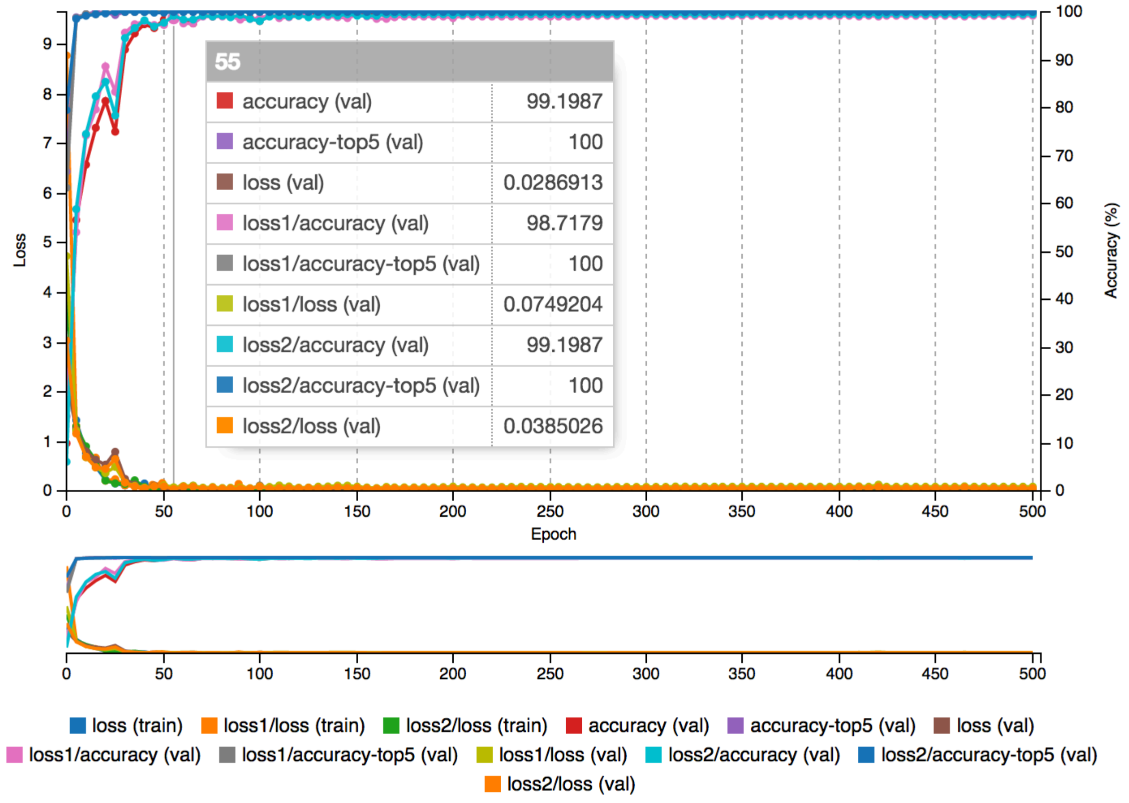


Figure 36: 7-class SGD Training Graph for GoogLeNet (Generated by DIGITS)

On the test set, GoogLeNet performed even better than AlexNet, yielding an overall test accuracy of 89.08%. This surely had to do with its more complex topology, as the decision surface the network created was better than the last. A summary of the results for each class is below in table seven.

GoogLeNet 7-class Test Accuracy	89.08%
“lab”	99.32%
“office”	100%
“machineshop”	97.29%
“serverroom”	82.69%
“lounge”	91.82%
“circuitslab”	57.27%
“entryway”	100%

Table 7: 7-class Test Accuracies for GoogLeNet

4.6.4 Results of the 7-class Dataset and Discussion

The networks that were trained with the additional two classes collectively performed slightly worse than their 4-class counterparts, except for GoogLeNet, which stayed somewhat consistent around the 89% range. This was likely due to a few different reasons. The first is that when adding in more classes, the decision surface that must be created becomes inherently more difficult to create. The additional two dimensions provided two extra layers of complexity to the problem.

Furthermore, while AlexNet and GoogLeNet are networks that were created for use with complex images, LeNet-5 was not. The LeNet-5 topology was originally created for use with recognizing handwritten digits, not for classifying places. AlexNet and GoogLeNet may be better re-purposed for the task at hand because of their more complex topologies. This allows them to create more complex decision surfaces among classes that are more complex and harder to distinguish between.

Another reason is the low performance of the “circuitslab” class between all classifiers. This, especially for LeNet-5, brought down the overall accuracy by a large amount. The low accuracy on the test set may have been because of poor coverage in the training set. This could likely be addressed by developing a heuristic to measure how much an area has been covered when collecting training data.

The fact that GoogLeNet maintained its accuracy relatively well is to be expected for such a complex and deep network. The repeated “inception units” that GoogLeNet is comprised of allow it to create a complex decision surface with a truly variable amount of data and classes provided.

4.7 Transfer Learning with Inception in TensorFlow

TensorFlow [32] is a deep learning library created by Google that is written in C++ but is primarily intended for use with Python. The library contains a C and C++ API, but at the time of the writing of this thesis, both were not nearly as well supported as the Python one. Within the TensorFlow examples, there exists a convenient script for performing a type of transfer learning with a version of GoogLeNet, Inception v3, that has been pre-trained on the ImageNet [5] database. The script is called *retrain.py*, and is able to retrain the final layer of the Inception v3 network for use with new image classes.

The idea of retraining on top of the ImageNet database would be to leverage the power of the deeply learned features that were realized from all of the images in the database and to apply them to entirely new classifications. The theory of how this operates is that if the early-layer features that were discovered by the neural network could be robust enough to help differentiate between the huge number of classes in ImageNet, then only the outer layers could be trained to distinguish between the new classes. The features which the network discovered early-on would be very rudimentary, consisting of edge detectors, texture and color detectors, and other kinds of features that can only be described through actual visualization.

The process of using the transfer learning script is relatively straightforward. First, training data is collected and put in folders corresponding to the appropriate class label. TensorFlow guidelines suggest that there should be at least five-hundred examples for each separate class in order for retraining to successfully occur. Next, the script scrapes each directory for the images, and passes them one at a time through the network to generate what the TensorFlow developers coin to be ‘bottleneck files’. These files are a collection of the weights at the penultimate layer of the network, which are then used during actual training to refer to the neural excitations that the image produces. Finally, training occurs in which the final classification layer of the Inception v3 network is re-trained from scratch to classify between the new categories.

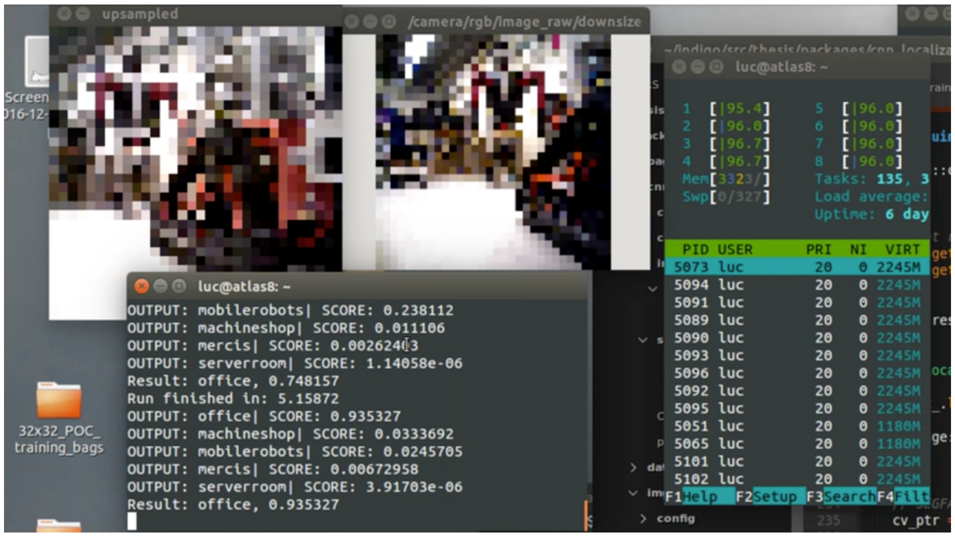
After the script has finished retraining the final layer to whatever number of iterations requested, the final accuracy is displayed and the TensorFlow graph definition file is saved to disk in a binary protocol buffer[18] (protobuf) format. A label file, as well as retraining logs, are also saved. The protobuf graph definition file may then be used later in deployment through the TensorFlow API.

Several attempts were made to use the retraining script to perform transfer learning with the intent of leveraging Inception v3's power to classify many locations. Each attempt, as long as there was significant data, yielded positive results with TensorFlow reporting high (>90%) accuracy levels on validation sets. The datasets tested ranged from the 4-class data used to successfully train LeNet-5, AlexNet, and GoogLeNet to high accuracy levels, to less spread out datasets consisting of different areas within the 'office' environment alone. When performing the retraining operation, all of the tested datasets claimed to have high levels of accuracy.

Unfortunately, when the models were tested with live camera data, it was quickly found that something was awry. The trained models performed terribly when fed with live camera data through a custom ROS package designed to use the graph definition and corresponding labels with the TensorFlow C++ API. The cause of the issue likely had to do with the usage of the C++ API as its documentation is far from fully developed.

[18] https://developers.google.com/protocol-buffers/

Furthermore, the network took an uncharacteristically long time to propagate an image forward through the structure. The larger the image, the slower the classification took. This led to the usage of 32x32 images being used to train and run through the network. On a dataset consisting of the five classes “office, “mercis”, “mobilerobots”, “machineshop”, and “serverroom” (“mobilerobots” and “mercis” later turned into the general label of “lab”), the classifier performed badly, hardly classifying anything correctly – even though its reported accuracy was high.



```
OUTPUT: mobilerobots| SCORE: 0.238112
OUTPUT: machineshop| SCORE: 0.011106
OUTPUT: mercis| SCORE: 0.00262403
OUTPUT: serverroom| SCORE: 1.14058e-06
Result: office, 0.748157
Run finished in: 5.15872
OUTPUT: office| SCORE: 0.935327
OUTPUT: machineshop| SCORE: 0.0333692
OUTPUT: mobilerobots| SCORE: 0.0245705
OUTPUT: mercis| SCORE: 0.00672958
OUTPUT: serverroom| SCORE: 3.91703e-06
Result: office, 0.935327
```

```
  1  [|95.4]      5  [|96.0]
  2  [|96.0]      6  [|96.0]
  3  [|96.7]      7  [|96.0]
  4  [|96.7]      8  [|96.0]
  Mem[3323/]      Tasks: 135, 3
  Swp[0/327]      Load average:
                  Uptime: 6 day

  PID USER      PRI  NI  VIRT
 5073 luc        20   0 2245M
 5094 luc        20   0 2245M
 5091 luc        20   0 2245M
 5089 luc        20   0 2245M
 5090 luc        20   0 2245M
 5093 luc        20   0 2245M
 5096 luc        20   0 2245M
 5092 luc        20   0 2245M
 5095 luc        20   0 2245M
 5051 luc        20   0 1180M
 5065 luc        20   0 1180M
 5101 luc        20   0 2245M
 5102 luc        20   0 2245M
F1Help  F2Setup F3SearchF4Filt
```

Figure 37: Incorrect Result of "office" for Actual "lab" Label Using TensorFlow Retraining Deployment

Even though the deployment results were unsuccessful, transfer learning still has an important role in this area of research. It was demonstrated that this technique performed well in PoseNet [25] and certainly could prove useful in later work. The limiting factor was almost certainly the confusing C++ API provided by Google. In the future, when a better and more understandable version of the API is released, the work likely could be continued without any issue.

5 ROS Deployment and Performance

There are three main phases to deploying a deep learning classifier. To begin, there must be some way of acquiring data with which to train the classifier. This is the data collection phase. Next, there is the training phase in which the gathered data is used to train the deep neural network. Lastly, the deployment phase is where the classifier is used to perform its required duty within some application. Apart from the main phases of data collection, training and deployment is the *cwru_deeplearning_robot* package which launches ROS nodes that support overall robot operation.

Software for the deployment of the neural networks was written in C++ using the ROS middleware framework. Multiple packages were developed that work together to allow for a robot to initialize MCL within a known environment. Previous phases and their corresponding packages have already been discussed in their relevant sections. A visualization of the entire software architecture is as follows:

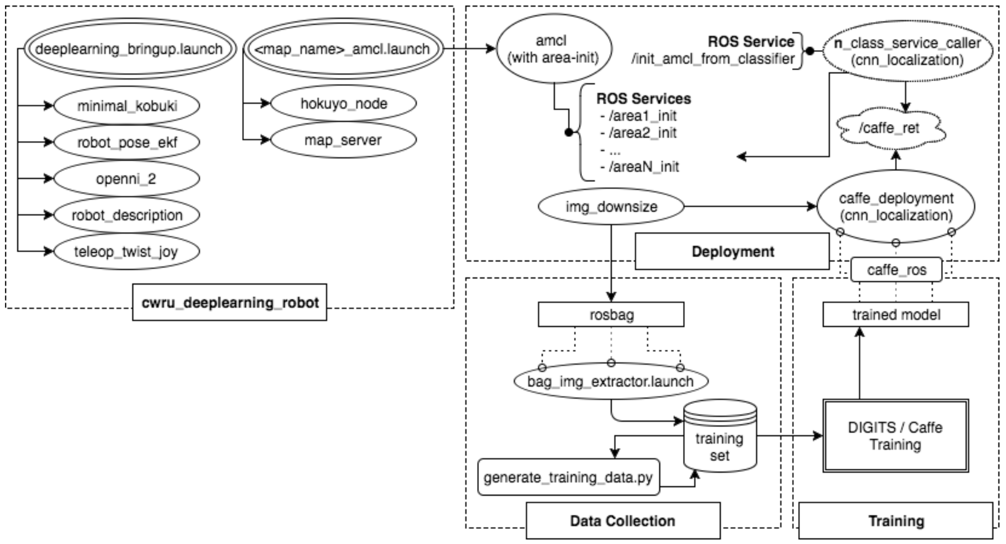


Figure 38: The Software Architecture for Caffe Deployment

In the present work, the trained classifier was deployed as a ROS node written using the Caffe C++ API. This was run together with a modification of AMCL edited to initialize particles in the different trained areas. The edited AMCL node uses a series of newly advertised ROS services based off of the traditional ‘global initialization’ call to put particles over pre-defined polygonal sections of the map instead of all over it.

While AMCL is attempting to track the position of the robot, *cnn_localization*’s *caffe_deployment* is constantly running the current image through the trained model. The *caffe_deployment* node uses a custom library called *caffe_ros* to load a Caffe classifier and other supporting files into memory. This library is heavily influenced by the popular *ros_caffe* library written by TzuTa Lin[19] with the main advantage of the new version being the ease of linking and instantiation.

The *caffe_deployment* node also takes advantage of a multiple classification estimate using a simple version of a soft voting majority rule ensemble classifier [33]. Since the robot is constantly broadcasting new images of the area it is currently in, the data may be taken advantage of by running each new image through the classifier. For each image classified, the probabilities of each class are added up and then averaged over the amount of additions performed.

For example, if this method were to use the five latest images to vote on where the robot was located, all of the respective probabilities for each area would be added together for each of the five images received. Then, those probabilities would be divided by five, with the highest probability being the output class. This method works well because of the sequential continuity of the classes of the images received and helps get rid of spurious incorrect classifications by letting the true class outweigh them.

[19] https://github.com/tzutalin/ros_caffe

The *caffe_deployment* node publishes this ensemble estimation of the current location of the robot on a set topic, usually set to “/caffe_ret”. Additionally, a verbose output that can be used to gain additional insight into how the classifier is doing using the voting method is available on the topic’s “verbose” channel (i.e. “/caffe_ret/verbose”). Lastly, the consistent, unfiltered output of the classifier is available on the “all” channel (i.e. “/caffe_ret/all”).

As the *caffe_deployment* node constantly classifies the current location of the robot in the map, another node, written specifically to test the 4-class categorization, runs in tandem. The *four_class_service_caller* constantly monitors the ensemble classifier’s output and saves it to an internal state variable. The node also advertises a service to initialize AMCL with the current estimate of the robot’s position using that state variable. On request, the node will use the classification information published to the ROS core to call the appropriate ROS service advertised by the edited version of AMCL to initialize particles over the appropriate corresponding region on the map. AMCL is then able to more accurately converge on the true position of the robot when compared to a global initialization.

Results of the deployment of the neural network were promising. Compared to global initialization of the algorithm, the timeliness and overall ability for the algorithm to converge on the position of the robot was improved. When globally initializing the filter, AMCL not only takes a long time to converge, but also is not guaranteed to converge on the robot, perhaps converging on some other position instead (even if that position is outside of the feasible area of the robot to exist within). Placing particles over a smaller area where the robot allows for the particles to be resampled more effectively and leads to overall better results.

Using the weighted majority classifier with a total of five votes, the robot was initialized successfully in each environment. Below, in table eight, are the trials carried out.

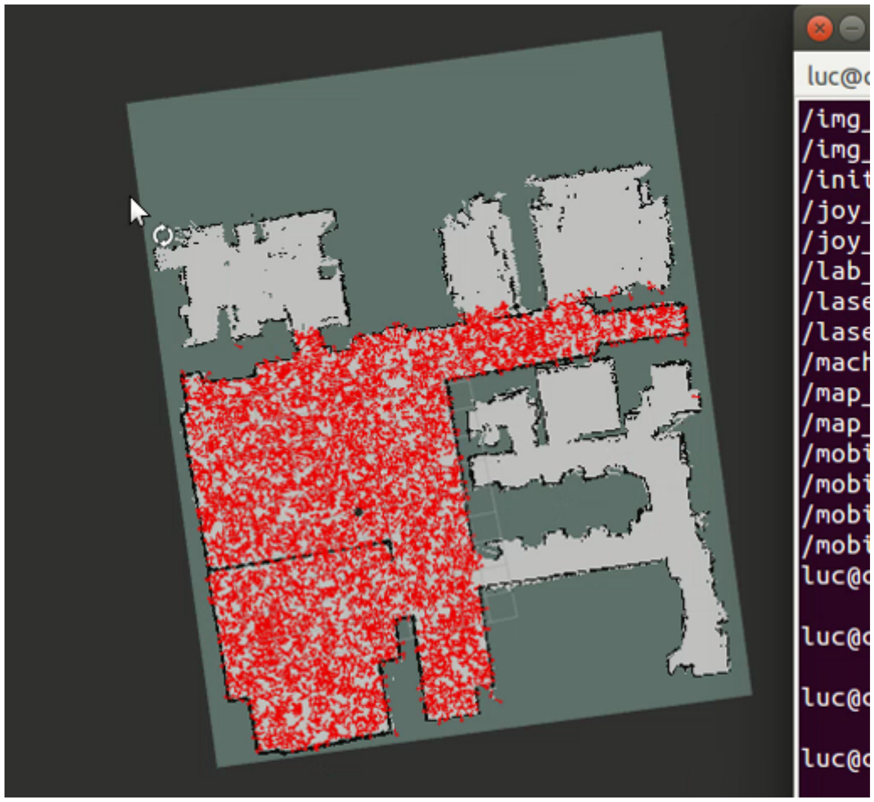
Initialization Type	**Result**	**Time to determine result (seconds)**
Global	Fail	16.78
Global	Fail	14.8
"office"	Fail	27.35
"office"	Partial success[20]	32.38
"serverroom"	Success	12.79
"serverroom"	Success	20.1
"lab"	Success	28.77
"lab"	Fail	20.18
"machineshop"	Success	14.17
"machineshop"	Success	20.27

Table 8: Particle Filter Initialization Trials Using Weighted Majority Classifier with Five Votes

It should be noted that failures occurred more often when initialization took place in areas that were rather large. This was partly due to AMCL's inability to throw away particles that traveled off of the surface of the map, which would be an impossible movement for the robot, but also because it is simply a more difficult search space. Each time the robot was initialized within the "serverroom" and "machineshop" environments, the particles converged on the position of the robot without issue.

The software development process was tracked online using the open source software "git"[21]. All software written for this thesis is available online with an open source license on GitHub[22].

[20] Correct convergence occurred, but subsequently failed.
[21] https://git-scm.com/
[22] https://github.com/lucbettaieb/thesis



```
/img_
/img_
/init
/joy_
/joy_
/lab_
/lase
/lase
/mach
/map_
/map_
/mobi
/mobi
/mobi
/mobi
luc@c

luc@c

luc@c

luc@c
```

Figure 39: ConvNet-derived AMCL initialization of the robot in the "lab"

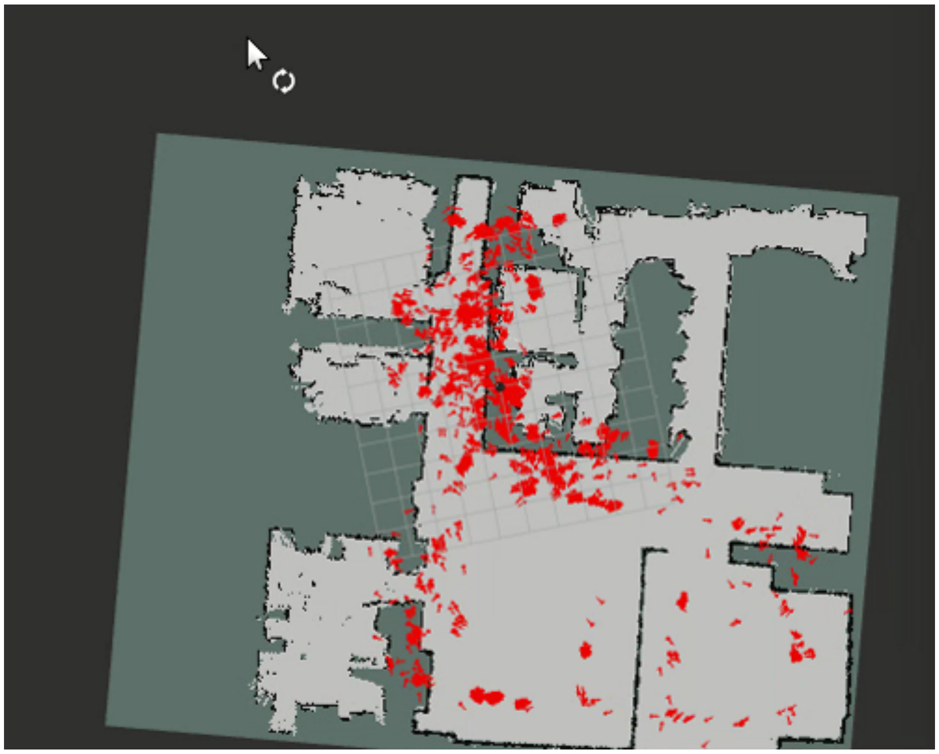


Figure 40: AMCL beginning to converge on the true position as the robot moves

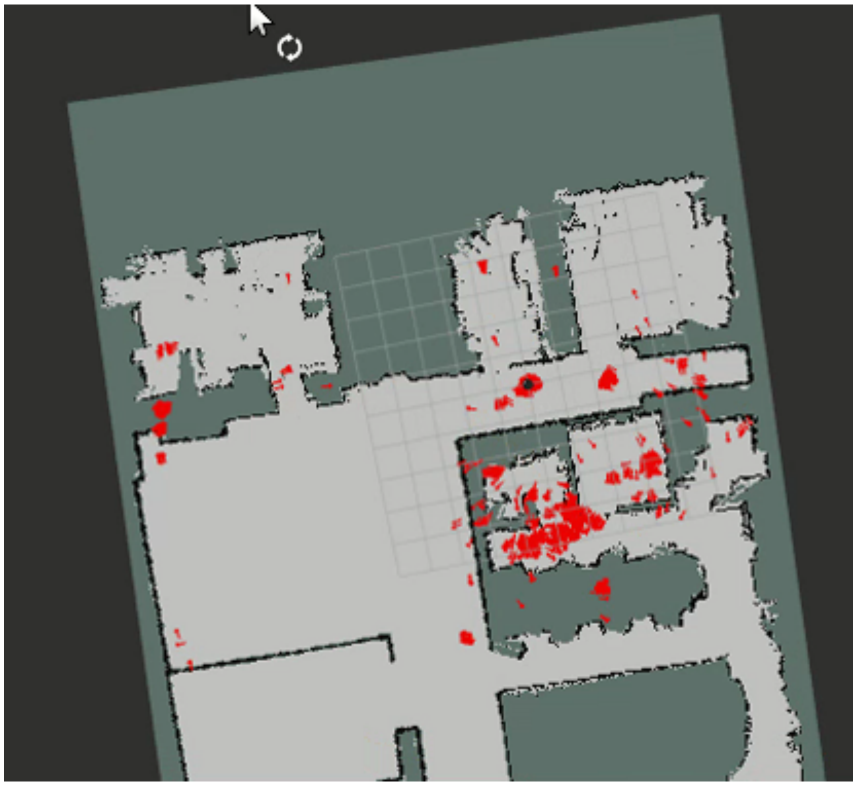


Figure 41: Plausible hypothesis of the robot's location - almost finished converging

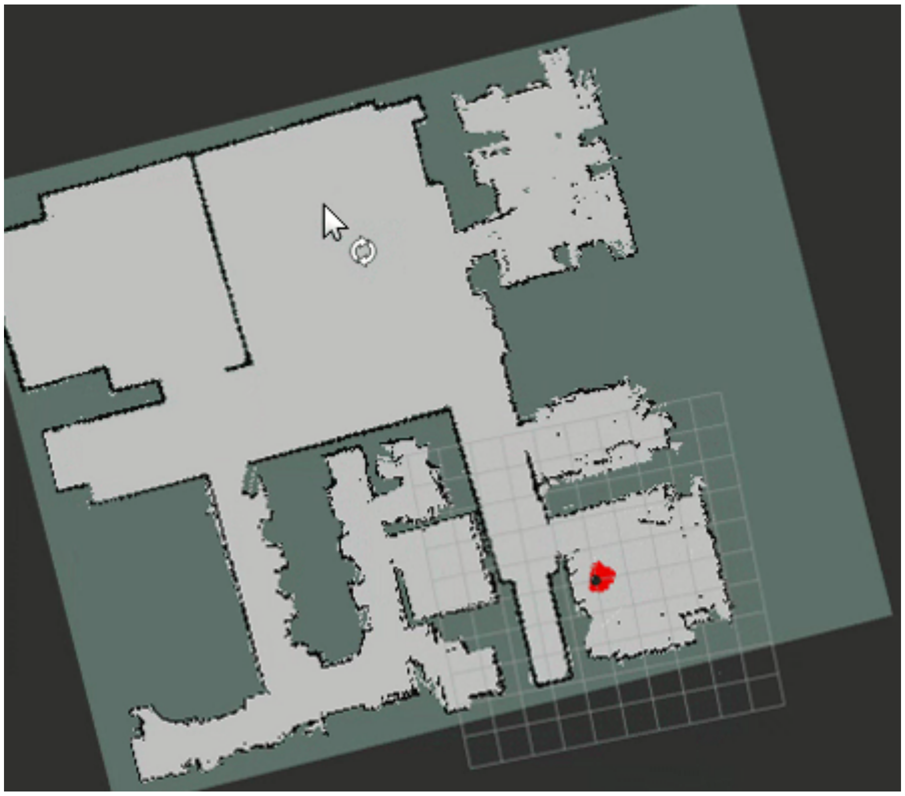


Figure 42: AMCL converged on the robot's position as it traveled to the 'machineshop' location

To improve performance, a weighted majority classifier was considered. Evaluation of this alternative deployment was performed using a Python script that used Caffe to run randomly selected test data through the pre-trained neural network. These images were then evaluated using voting among individual classifications. This script began by setting up the neural network using Caffe's Python API, then loading the first class of images from a labeled directory within a base directory. Then, for N between 1 and 9 where N is the number of images to be passed through the network and used to voting, using each class, one hundred separate trials were conducted in order to get an estimate of the network's performance. The results were generally promising, with some exceptions.

LeNet-5 was able to see significant improvements when using the weighted majority classifier, approaching near perfect accuracy for all classes when the number of classifications reached 9. The office class seemed to be the most difficult for the network, perhaps indicating that more training data could be useful in order to differentiate between that environment and parts of others that could look similar.

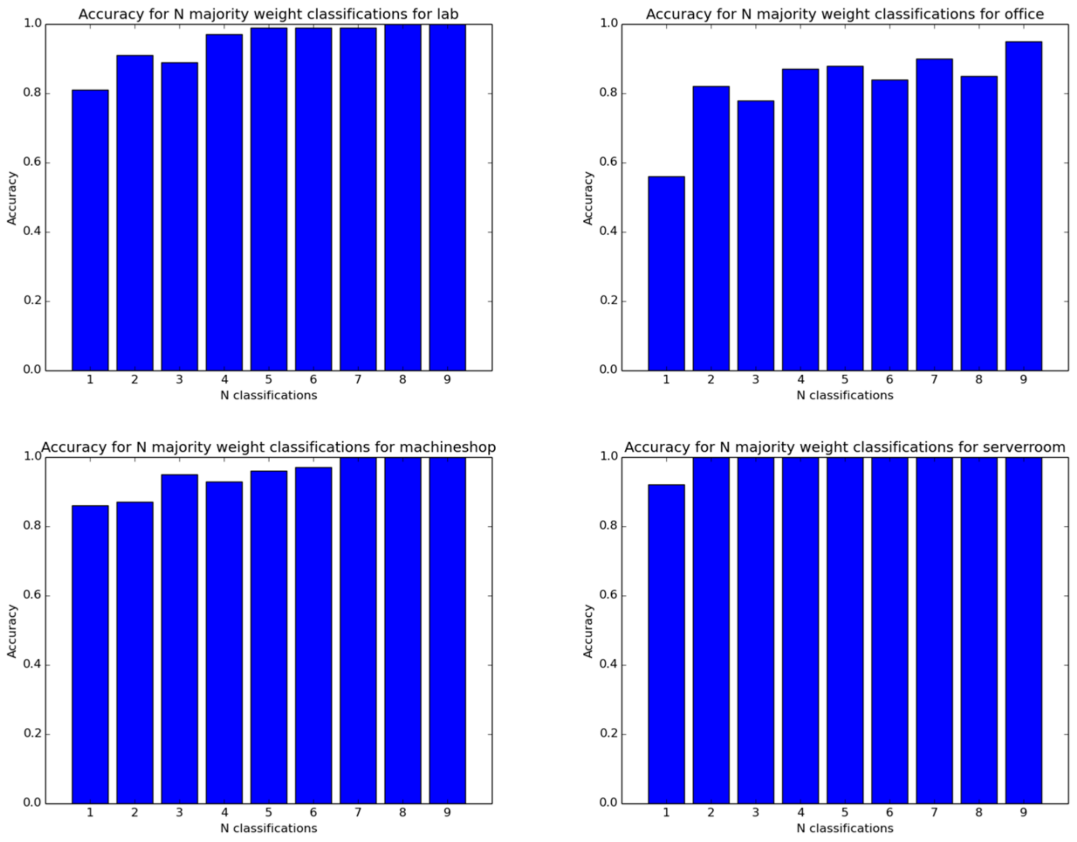


Figure 43: Weighted Majority Classifier Performance using LeNet-5 for the 4-class Dataset

AlexNet was evaluated as well using a majority-vote classifier. When the number of votes reached five, the weighted majority classifier was able to achieve perfect results. However, even after one vote was added in to the classification, significant improvements were seen.

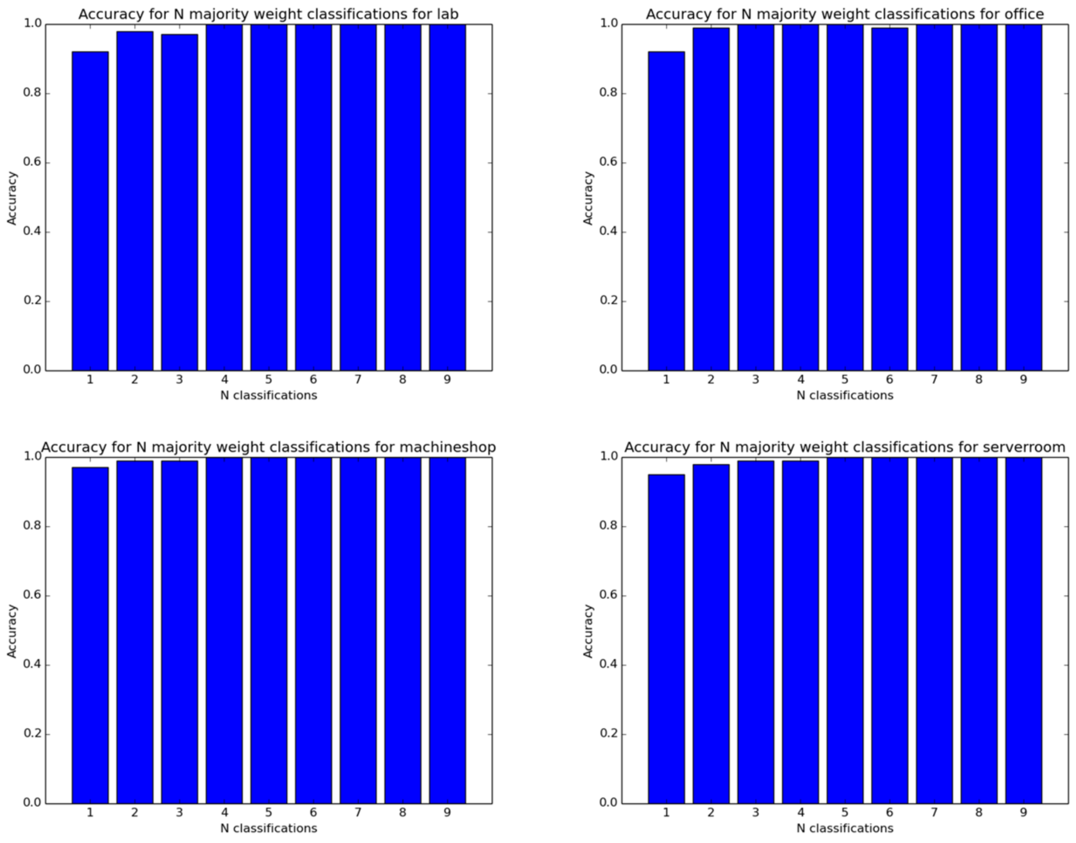


Figure 44: Weighted Majority Classifier Performance using AlexNet for the 4-class Dataset

GoogLeNet was similarly evaluated. After four votes, the weighted majority classifier was able to help GoogLeNet achieve near perfect results – however, it was not until seven votes were used that the network was able to achieve apparent perfection.

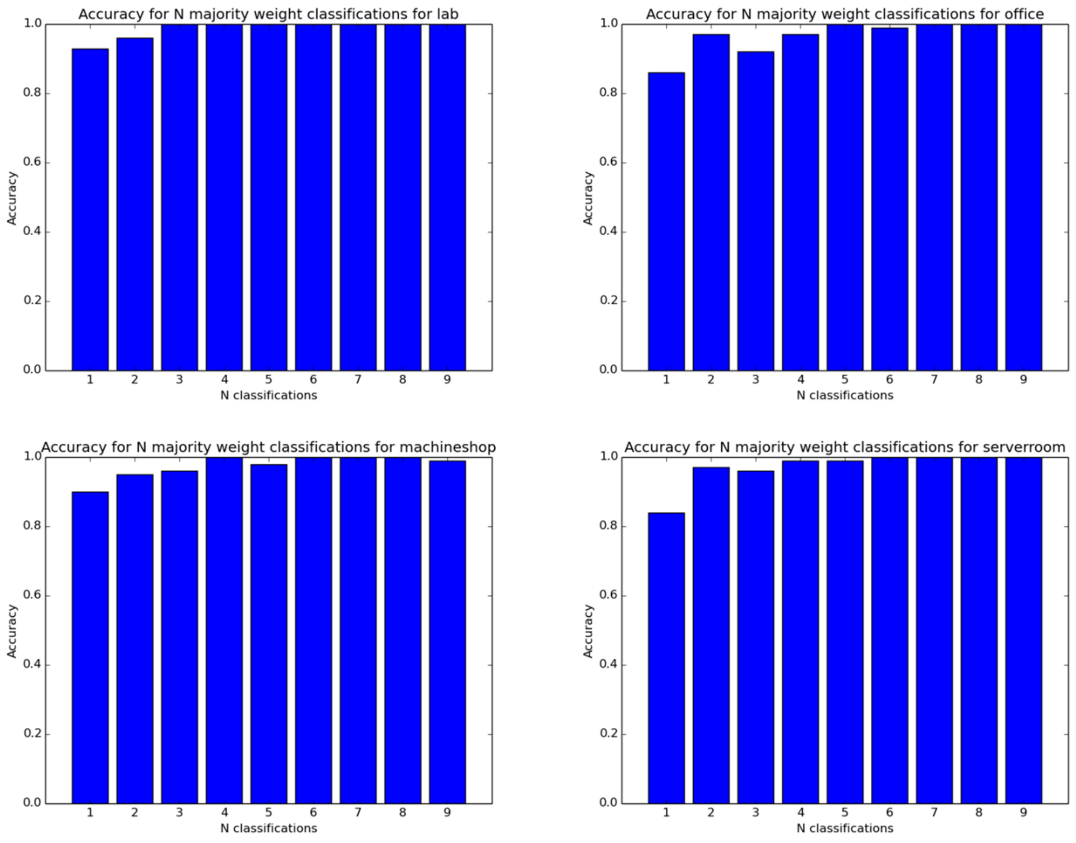


Figure 45: Weighted Majority Classifier Performance using GoogLeNet for the 4-class Dataset

For the 7-class dataset, LeNet-5 was able to see improvements within every single category except for the circuits lab and entryway, which both performed poorly overall. However, even as they did perform poorly, it seemed as if adding more votes to the classification was able to help slightly in raising the accuracy. For the circuits lab, using three votes was able to achieve the best results at around 53% accuracy. For the entryway, nine votes produced the best classification accuracy, at around 27%.

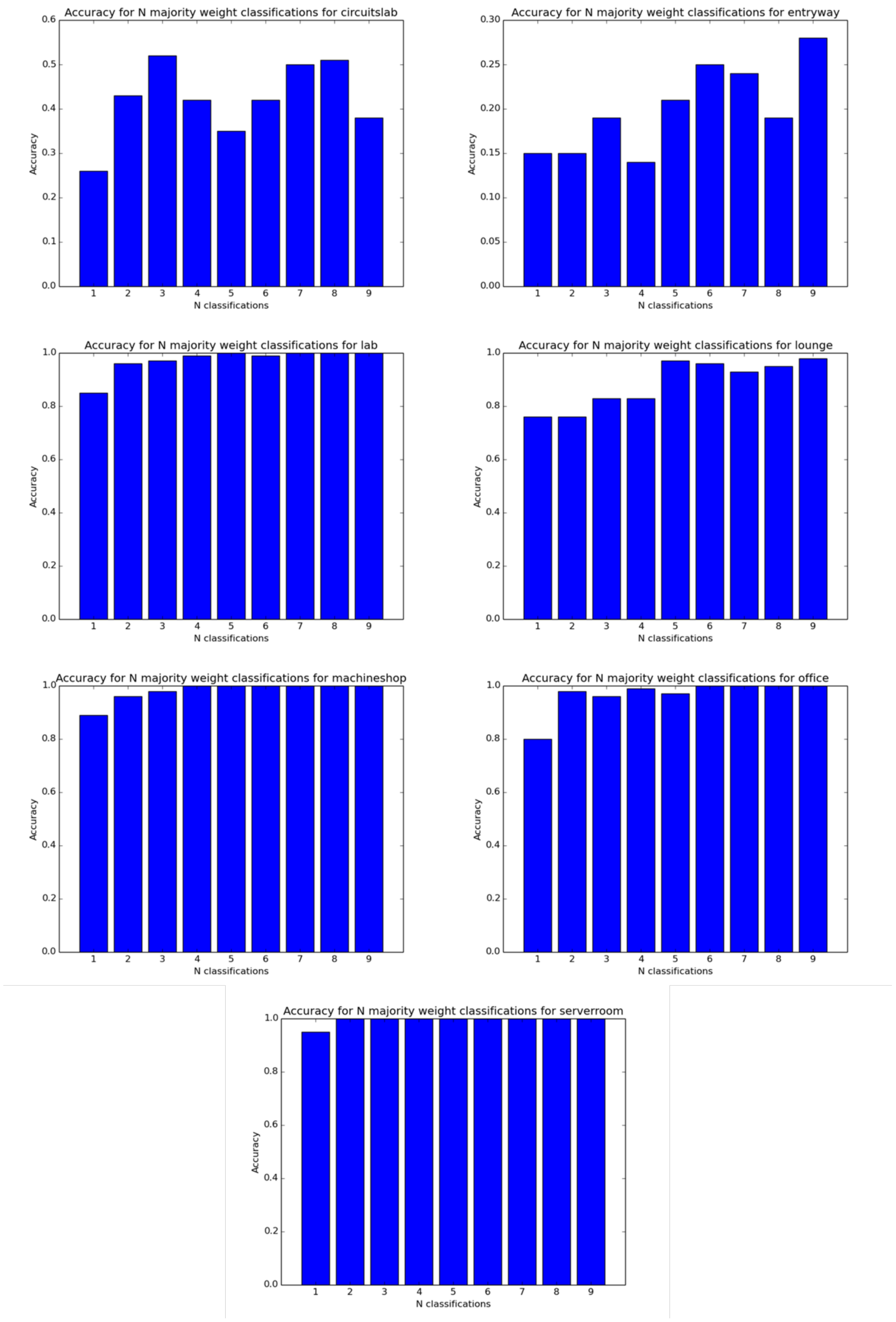


Figure 46: Weighted Majority Classifier Performance using LeNet-5 for the 7-class Dataset

AlexNet was re-evaluated for the 7-class dataset. It suffered greatly in classifying the circuits lab when using the weighted majority classifier. It was unable to perform well using its regular classification of one vote, but adding more only made the accuracy worse. This may be related to poor training data collection for the area which would support the creation of a heuristic to determine when enough data was collected. For the circuits lab, it achieved around 14% accuracy on the set with one vote. After adding more votes in, the accuracy eventually tanked to 0%. However, for all of the other classes – especially the entryway, adding the additional votes helped the accuracy rise to near perfection for seven votes.

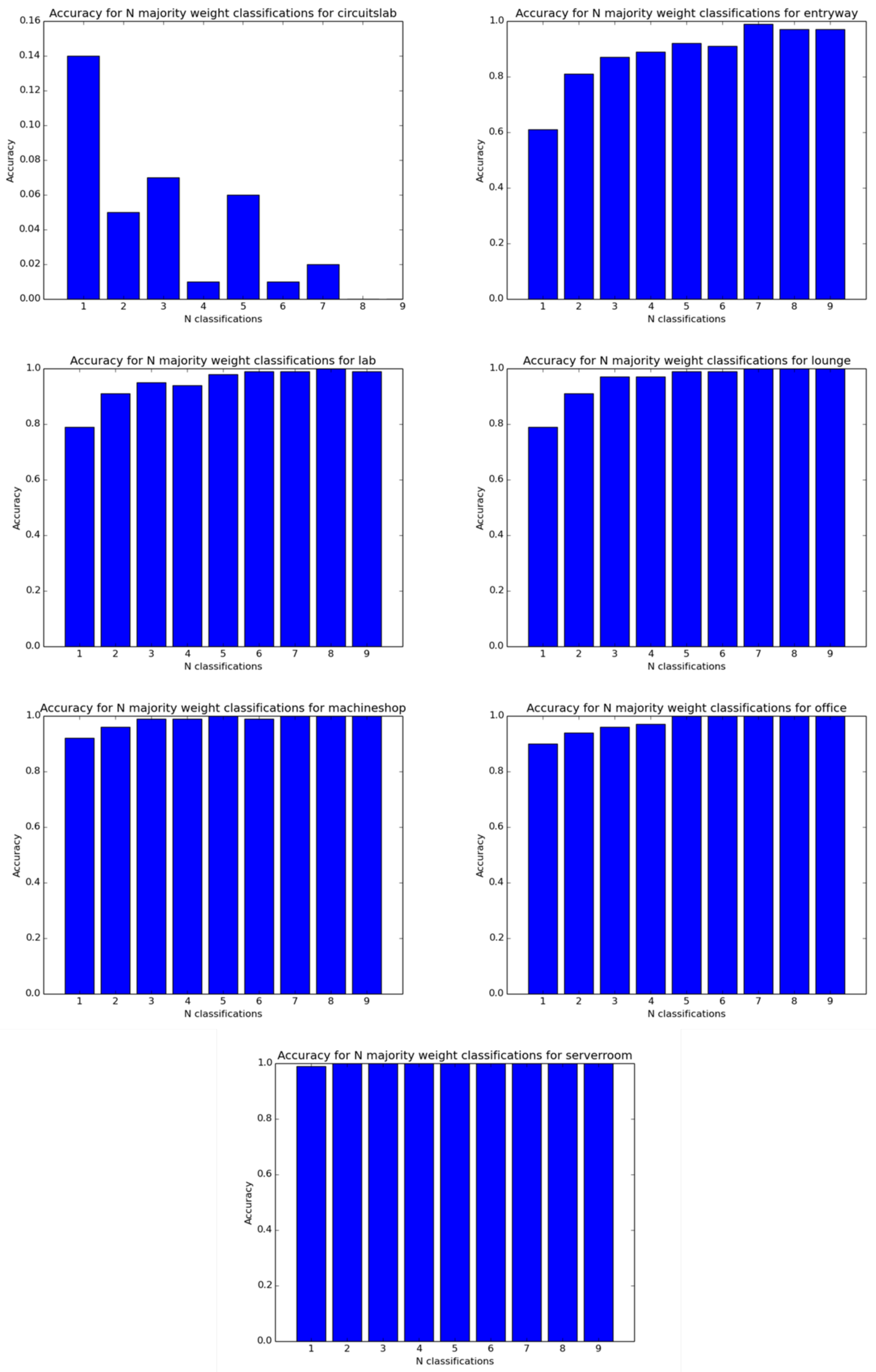


Figure 47: Weighted Majority Classifier Performance using AlexNet for the 7-class Dataset

GoogLeNet, like AlexNet, did not fare well in identifying the circuits lab. Adding additional votes into the mix only decreased accuracy for this class. This further supports the claim that not enough training data was collected for this area. Again, for all of the other classes, test accuracy increased with the number of votes used, nearing perfection after five votes.

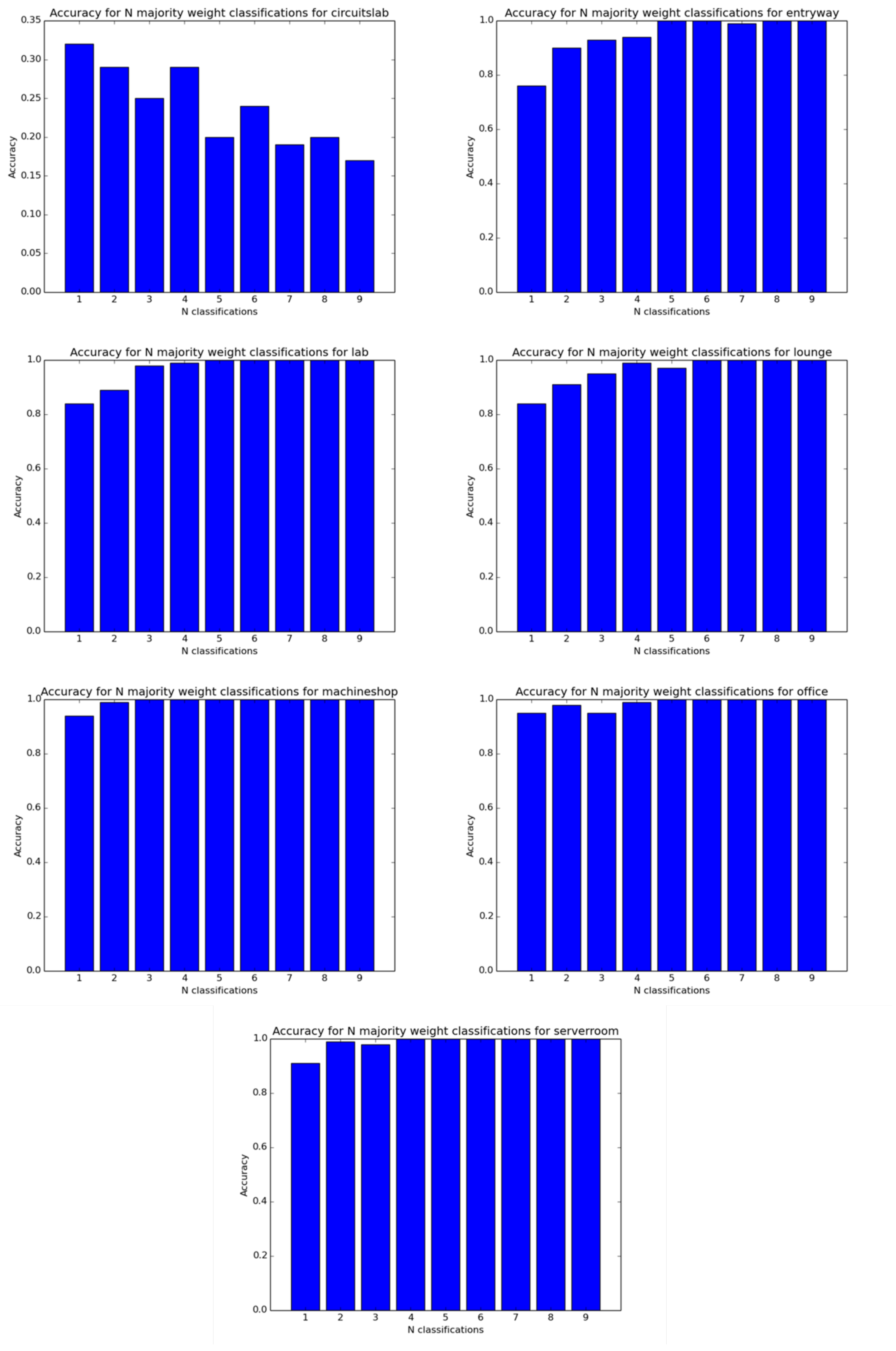


Figure 48: Weighted Majority Classifier Performance using GoogLeNet for the 7-class Dataset

6 Conclusion and Future Work

In this thesis, a deep neural network classifier was used to determine the rough location of a mobile robot in a known environment. To facilitate this, the robot operating system (ROS) was used as well as the NVidia Deep Learning GPU Training System (DIGITS). Training data was collected using an Asus Xtion Pro camera attached to a Turtlebot. Three deep neural network topologies were trained using DIGITS and Caffe and evaluated for their usefulness.

By testing each topology and each solver type, it was found that LeNet-5 provided a speedy training time with a very acceptable accuracy, but it began to suffer when more classes were added. Conversely, GoogLeNet performed very well for many different classes, but it required a long time to train. This is likely because of the greater number of learned parameters paired with the increased complexity of the decision boundaries in the feature space. Out of all of the solver types, stochastic gradient descent (SGD) seemed to consistently perform the best. Overall, GoogleNet trained with SGD was the best choice for any number of classes applied to the network.

After training, Caffe was able to generate files that allowed the network to be deployed. ROS code was written to load the Caffe network into memory and continually classify the camera's images. A weighted majority classifier was used to classify multiple images using a vote of multiple classifications. By doing so, gross localization of the robot is revealed, and this information was used to initialize or re-initialize a particle filter used to track the robot's position. This system was tested and successfully deployed on the Turtlebot, allowing for it to initialize and re-initialize its particle filter with faster localization times than if initialized using global initialization. With respect to previous claims that global localization with a particle filter solves the kidnapped robot problem, this approach is able to solve it in a better way, since it allows for much faster re-localization.

The contents of this thesis describe much of the work done on this project, but there could always be improvements. Some work was done on supplemental software packages that allow for the automatic collection of training data simply by driving the robot around the known environment.

To assist data labeling, the map may be segmented into different regions using a standard interface defined in the *map_segmentation* package. This interface allows for different implementations of segmentation algorithms to be loaded in using the ROS pluginlib[23] interface. After segmentation, a distance heuristic may be used as long as AMCL has a low covariance to automatically assign incoming frames to a label corresponding to the segmented area on the map. Using the *data_collector* package, the robot could be driven around until enough data was gathered for each area on the map. This metric could either be numerical or defined according to a more advanced heuristic, perhaps including the pose of the image taken.

These automatic data collection and labeling features would be quite valuable, especially if this research project was ever to be made robust and developed into a working ROS package for widespread use. Additionally, many more features that have not been developed at all would be likely needed.

When the neural network is deployed, it uses data that was gathered at a previous point in time. This has an inherent downside for a robot that operates in dynamic

[23] http://wiki.ros.org/pluginlib

environments that constantly change over time. This would be especially true of robots that operate around many people. To address this, it would be beneficial to implement a deployment and learning system that could continually learn and improve its interpretation of the environment.

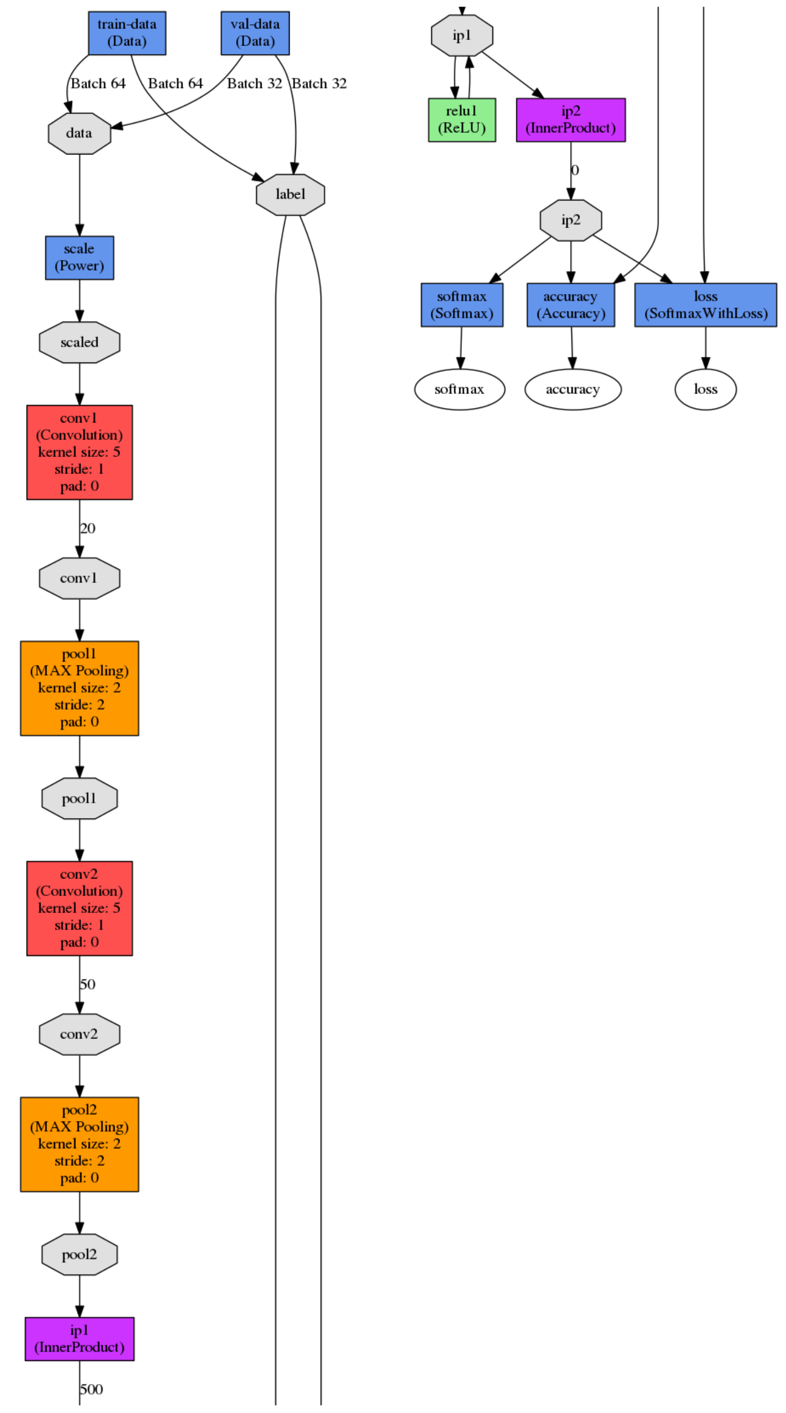
Furthermore, if the robot perhaps encountered a new environment where the class label was unknown, instead of naively assigning it to one of the previously trained class labels, it could create an arbitrary class label for it and add it to its network. At a later time, a human teacher would be able to assign a proper class label to the area. Note, though, that if the label were simply being used to assist in the localization of the robot, its actual name would not be truly required.

There are many limitations to developing software with AMCL. When the node itself was edited to provide service calls to initialize particles in different regions of the map, a very round-about approach was used to get it to work. Sometimes, particles also make movements that are physically impossible for a robot to make. Monitoring particle movement and giving a low score to particles that go through walls or exist off the map would be beneficial. To this note, a more modular and extensible version of a particle filter is desired. Currently, no open source projects within the ROS ecosystem exist that fulfill this need, so AMCL would likely need an overhaul to be properly used for the continuation of this project.

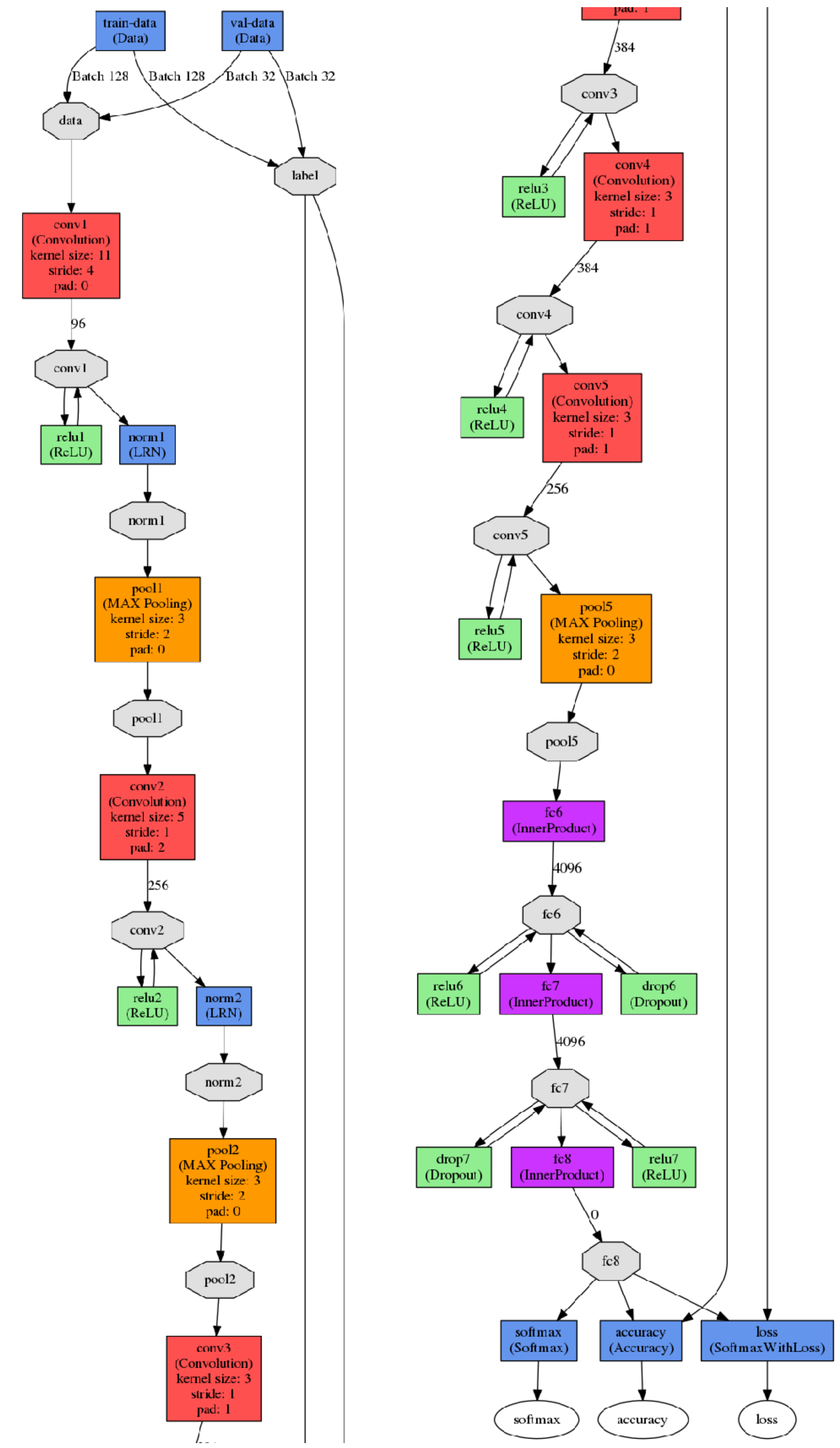
Another challenge that could be addressed by future work is the optimal allocation of resources between the Jetson TK1 on the Turtlebot, and the supplemental netbook. In reality, if a more powerful computer could be attached to the robot instead of the TK1, the problem would likely solve itself.

Lastly, more research could be performed in the area of actually returning a pose estimation, rather than merely a room label, e.g. as in PoseNet [25]. If there were a streamlined approach to generating training data for a mobile robot to train a network like PoseNet that could return an estimate of the robot's position in an indoor environment, it would perform the functionality of this thesis with much greater accuracy and usability. An error term could be generated between the particle filter's estimated pose and the pose returned by the neural network. With this information, an appropriate action could be taken, e.g. re-initialization, an adjustment of the map to odometry transform, or further training data being added back into the network.

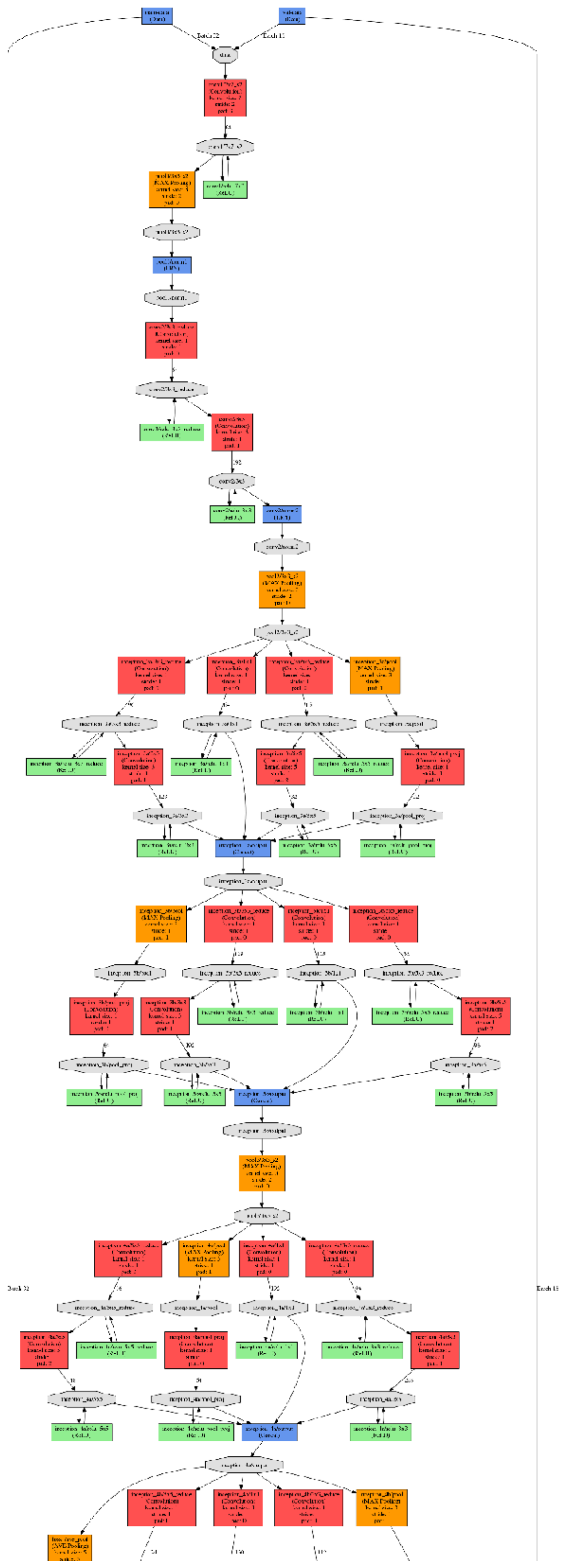
Appendix A: LeNet-5 Structure

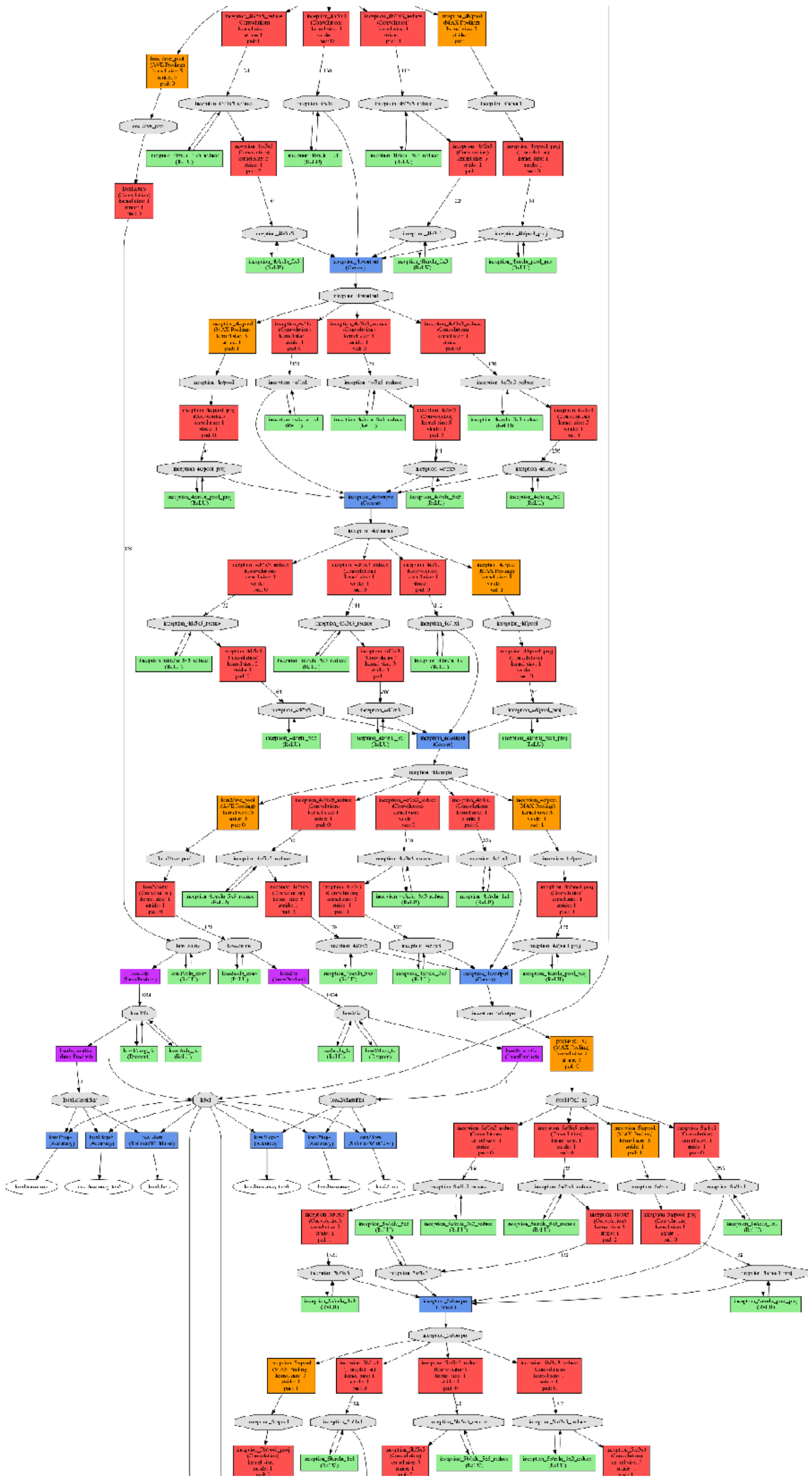


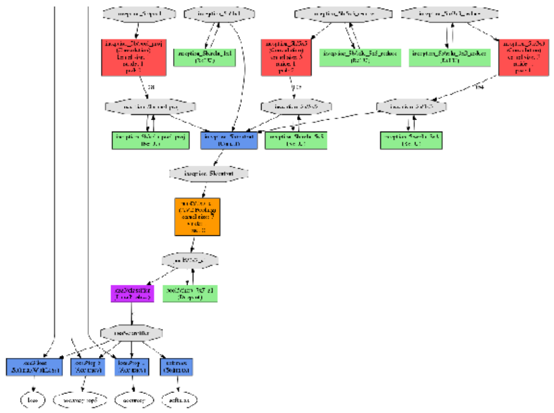
Appendix B: AlexNet Structure



Appendix C: GoogLeNet Structure






loss
accuracy
softmax

Bibliography

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] D. Fox, W. Burgard, F. Dellaert, and S. Thrun, "Monte Carlo Localization: Efficient Position Estimation for Mobile Robots," *Aaai-99*, no. Handschin 1970, pp. 343–349, 1999.

[3] M. Quigley *et al.*, "ROS: an open-source Robot Operating System," *Icra*, vol. 3, no. Figure 1, p. 5, 2009.

[4] B. Gerkey, "amcl." [Online]. Available: http://wiki.ros.org/amcl. [Accessed: 21-Apr-2017].

[5] O. Russakovsky *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2015.

[6] S. Song, S. P. Lichtenberg, and J. Xiao, "SUN RGB-D: A RGB-D scene understanding benchmark suite," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 07–12–June, pp. 567–576, 2015.

[7] Y. Jia *et al.*, "Caffe: Convolutional Architecture for Fast Feature Embedding," *ACM Int. Conf. Multimed.*, pp. 675–678, 2014.

[8] L. Yeager, "DIGITS : the Deep learning GPU Training System," *ICML AutoML Work.*, 2015.

[9] "UNIMATE: The First Industrial Robot," *Robotic Industries Association*. [Online]. Available: http://www.robotics.org/joseph-engelberger/unimate.cfm. [Accessed: 04-Aug-2017].

[10] N. J. Nilsson, "Shakey the Robot," *Tech. Note 323*, no. April, 1984.

[11] M. Wise, M. Ferguson, D. King, E. Diehr, and D. Dymesich, “Fetch & Freight : Standard Platforms for Service Robot Applications,” pp. 2–7, 2016.

[12] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. MIT Press, 2005.

[13] A. Majdik, M. Popa, L. Tamas, I. Szoke, and G. Lazea, “New approach in solving the kidnapped robot problem,” *Robot. (ISR), 2010 41st Int. Symp. 2010 6th Ger. Conf. Robot.*, pp. 1–6, 2010.

[14] T. M. Mitchell, *Machine Learning*. McGraw-Hill, 1997.

[15] A. Karpathy, “cs231n: Convolutional Neural Networks (CNNs / ConvNets).” [Online]. Available: http://cs231n.github.io/convolutional-networks/.

[16] L. Brown and NVidia, “Accelerate Machine Learning with the cuDNN Deep Neural Network Library.” [Online]. Available: https://devblogs.nvidia.com/parallelforall/accelerate-machine-learning-cudnn-deep-neural-network-library/. [Accessed: 12-Apr-2017].

[17] P. J. Werbos, “Backpropagation Through Time: What It Does and How to Do It,” *Proc. IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[18] S. Ruder, “An overview of gradient descent optimization algorithms,” *Web Page*, pp. 1–12, 2016.

[19] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-Based Learning Applied to Document Recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” *Adv. Neural Inf. Process. Syst.*, pp. 1–9, 2012.

[21] C. Szegedy *et al.*, "Going deeper with convolutions," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 07–12–June, pp. 1–9, 2015.

[22] Y. LeCun *et al.*, "Handwritten Digit Recognition with a Back-Propagation Network," in *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, Ed. Morgan-Kaufmann, 1990, pp. 396–404.

[23] Stanford Vision Lab, "Large Scale Visual Recognition Challenge 2011 (ILSVRC2011)." [Online]. Available: http://image-net.org/challenges/LSVRC/2011/results. [Accessed: 12-Apr-2017].

[24] S. Arora, A. Bhaskara, R. Ge, and T. Ma, "Provable Bounds for Learning Some Deep Representations," *Int. Conf. Mach. Learn.*, vol. 32, p. 18, 2014.

[25] A. Kendall, M. Grimes, and R. Cipolla, "PoseNet: A Convolutional Network for Real-Time 6-DOF Camera Relocalization," pp. 2938–2946.

[26] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva, "Learning Deep Features for Scene Recognition using Places Database," *Adv. Neural Inf. Process. Syst. 27*, pp. 487–495, 2014.

[27] B. Zhou, A. Khosla, A. Lapedriza, A. Torralba, and A. Oliva, "Places: An Image Database for Deep Scene Understanding," *ArXiv*, pp. 1–12, 2016.

[28] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *Arxiv.Org*, vol. 7, no. 3, pp. 171–180, 2015.

[29] Y. Liao, S. Kodagoda, Y. Wang, L. Shi, and Y. Liu, "Understand scene categories by objects: A semantic regularized scene classifier using Convolutional Neural Networks," *Proc. - IEEE Int. Conf. Robot. Autom.*, vol. 2016–June, pp. 2318–2325, 2016.

[30] N. Sünderhauf, S. Shirazi, F. Dayoub, B. Upcroft, and M. Milford, “On the performance of ConvNet features for place recognition,” *IEEE Int. Conf. Intell. Robot. Syst.*, vol. 2015–Decem, pp. 4297–4304, 2015.

[31] S. Gutstein, O. Fuentes, and E. Freudenthal, “Knowledge Transfer in Deep Convolutional Neural Nets,” *Int. J. Artif. Intell. Tools*, vol. 17, no. 3, p. 555, 2008.

[32] GoogleResearch, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015.

[33] S. Raschka, “EnsembleVoteClassifier.” [Online]. Available: http://rasbt.github.io/mlxtend/user_guide/classifier/EnsembleVoteClassifier/. [Accessed: 12-Apr-2017].