Zebravision 4.0 Object Detection

Alexander Allen and Ben Decker

A Zebravision Labs White Paper:

Brought to you by Team 900:

http://team900.org

Introduction

Team 900 worked on developing a generalized method of detecting objects that do not have retroreflective tape. We accomplished this by training neural networks to locate objects within images. Neural networks are, at the most basic level, mathematical models which are applied to incoming information to determine features of that information. In our case, these models are mathematical processes that are applied to matrices containing RGB (Red, Green, Blue) values of each pixel in the input image. This model outputs a single value, the probability of the input image being a boulder, the game piece for the 2016 FRC Game. We can use this to process image input from cameras and locate objects within images.

We use a combination of the Caffe[1] library from the “BerkleyVision”[2] team and the OpenCV library to create, train, and use the networks. Our code can implement networks which can locate any type of object in an image given proper training of the network. Our particular approach was taken from a paper written by researchers at the Stevens Institute of Technology and Adobe Research[3]. Their method uses several sets of neural networks to detect objects. A simple, fast net is used to quickly screen out parts of the image which are obviously not the object while the remaining parts of the image are passed to progressively slower and more accurate nets for final classification.

Ü ˙a{{u©µµMV_Z ªUZxi ZjZ†ƒbybnl ªnx` µ˙

Ý ˙a{{u©µµ„ „ „ ªZZVyªUZxi ZjZ†ªZX|µ?ZyZMÆVaµ<xnhZV{yµ! @µƒbybnl µ˙

Þ ˙

a{{u©µµ„ „ „ ªVƒ¹_n|l XM{bnl ªnx` µnuZl MXWZyyµVnl {Zl {¼Vƒux¼ÝÛÜàµuMuZxyµ1b¼ ¼ nl ƒnj|{bnl Nj¼3Z|xNj¼ÝÛÜà¼ G<?¼uMu ZxªuX_˙

Neural Network Introduction

A neural network, in general, is an optimized mathematical model that processes a fixed size input, such as our images, and outputs confidence scores for a set of labels used for classification of the input data. The confidence scores are normalized between 0 and 1 and can basically be thought of as a likelihood that the input fits in a particular labeled class.

For example, in our case a given image will be run through the net and give us two confidence scores. One will be the percent confidence that the object is a boulder and the other will be a percent confidence that the object is not. The results are normalized to add to 100% so in practice with only two classes we have all the information we need from the "object is a boulder" confidence score.

At a high level, a neural net used for image classification has two basic parts. The first is a set of convolutional filters and the second is a fully connected network. The convolutional filters work over a small subset of the image and identify features in the image at a given location - lines, blobs of color, and so on. The fully connected layers look at the complete set of feature outputs and use that classify the image.

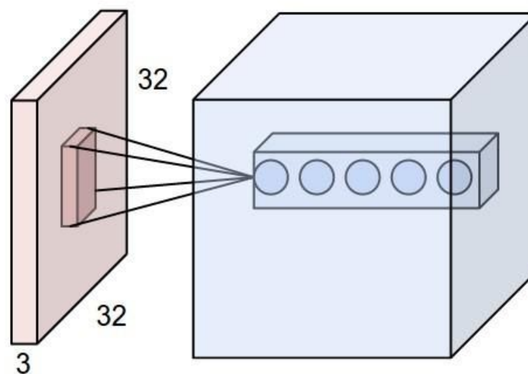
This is similar to how other object classification approaches work with one significant difference. Older classification schemes use hand-crafted fixed feature extractors that are defined separately from the code which used their output. Neural nets, on the other hand, "learn" both the filters and the fully connected classifier simultaneously when trained with input data. In practice this makes them much more capable than approaches where the two are separate.

Creating the Networks

We used the software DIGITS[4] from NVIDIA to facilitate building and training the networks. This software runs as a server on a computer with an NVIDIA GPU. This software uses a configuration file to define each layer of the neural network and its properties. These layers define what algorithms the Caffe library should use to process the data as it moves from layer to layer.

The first layers define how the network takes input. It defines the batch size for training iterations, the size of the image (width, height, color depth), and any data augmentation (such as including the mirror of all images in training).

The next layer is the first layer of the actual neural network, which is the convolutional layer. This layer takes in a set of input data and transforms it such that it easier for the fully connected layers to classify the image. This step identifies individual features of the image so that they are recognizable by the other layers of the network.



8]U[fUa ˙cZ=bdi hg#Ci hdi hgicZ7dbj cˋi hjcbU ˙@UhYf ˙

=a U[YgZfca \Hd.##Ag&' %b'[]h\i V']c#˙

ß ˙a{{uy©µµ` b{a| UªVnk µ3G*ž *˚ µž *(*A@

It works is by passing a “filter kernel” over the image in which the size of the filter in pixels is the “kernel size” and the amount of pixels it jumps on each iteration is the “stride”. Each of these filters is a 3 dimensional matrix with the width and height corresponding to the size of the filter and the depth corresponding to the number of input channels in the image. That is, the filter values applied to each channel could be different. This allows, for example, a particular filter to look for a certain color while ignoring others.

As each filter is passed across the image it builds an output matrix whose depth corresponds to the number of filters used. This is analogous to the creation of a new image with the number of channels in the image corresponding to the number of filters used in the convolutional layer. Each “pixel” in this new image is representative of how well the set of pixels in the initial image “match” the filter that is being passed over it. The stronger the match the larger the output value.

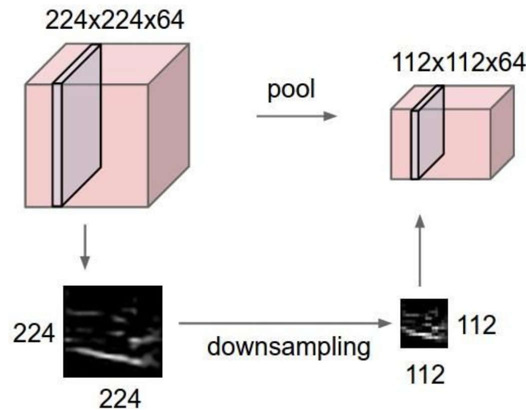
These results are often visualized using heat maps where “hotter” colors show areas of the input image which more closely fit a given filter. An example is shown on the third set of images in the figure on the right of page 9. Filter #4 in particular is a very good match to the text running down the middle of boulder.

In general concept the convolutional filters are the same as those used in many other image processing applications[5]. The main difference is that instead of fixed values, during training the values in the filter matrices are optimized to produce the ideal result for the next layers to properly classify the image.

---

[5] [illegible]

The next layers of the network prepare the convolutional output to be sent into the fully connected network. The first layer of this is the ReLU (rectified linear unit) layer which simply sets all negative values in the data set to 0. This introduces some non-linearities into the calculation which has been found to increase training speed[6].



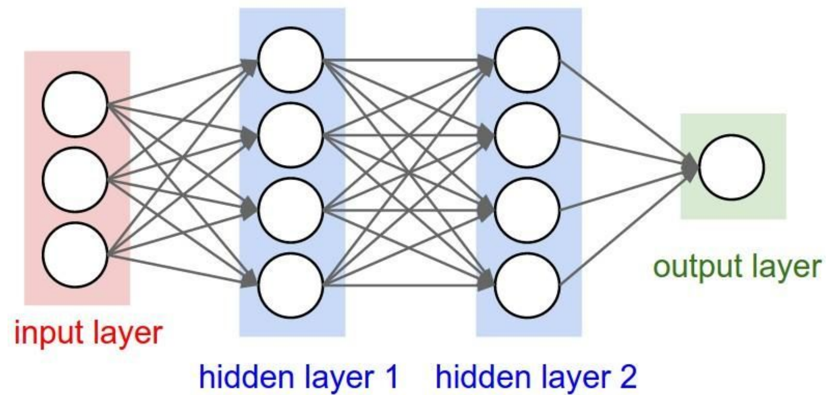
8UrUUbXJ]gj U F YdfYgldrUr]cb cZDcc]b[@UnYf

The next layer is the pooling layer which down samples the data to a lower resolution so that the fully connected layer can run faster. In our network we keep this to a minimum as the images are already rather small, but we must use it to ensure the fully connected layers run reasonably fast. Pooling also helps the network to not be affected by a rotated or translated object, because multiple pixels are being pooled into one these small changes will not trip up the network. Also because this normalizes the images during the training process it helps prevent the network from “overfitting” the images and create a more generalized definition of what it is searching for.

Finally, the fully connected layer is next. Labelled “Inner Product” in Caffe, this is the primary processing layer of the network and it converts the many values in the output of the pooling layer into “scores” for each possible classification.

á a{{uy©µuMlZxyªl buyªWµuMlZxµßãÝß¹bk MlZl Z{¹VjMyb_bMlbnl ¹„ b{a¹XZZu¹Vnl ƒnj|{bnl Njl'l Z| xNjl'l Z{„ nxi yªuX_

In our case, our fully connected layers have two outputs for “boulder” and “not a boulder”. The fully connected layer represents large weight and bias matrices which are applied to all of the output pixels from the previous convolution layers. Unlike convolution which looks for specific features of the image, the fully connected network looks at the entire image to classify it. When the inner (dot) product is taken of the data set with one of these filters it results in a scalar (single) value.



8]U[fUa cZ: i ``m7cbbYWhYXBYhkcf_

The first fully connected layer of two compiles the data into a series of values, then the second fully connected layer compiles the series of outputs from the first into the two scores which are reported in the end.

In between the two networks are a ReLU layer to zero out negative values in the nodes of the intermediate hidden layer. This non-linearity helps the network better handle non-linear boundaries between object classes.

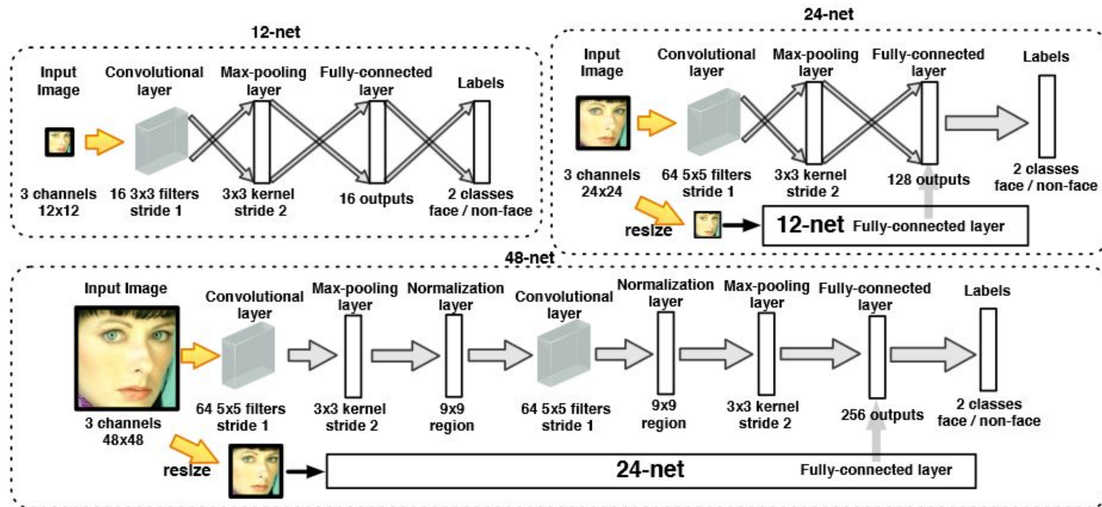
Dropout layer which randomly drops out a percentage of the connections in the network. The dropout layers helps to prevent overfitting as the fully connected networks[7], especially the latter

---

â ˙a{{uy©µµ„ „ „ ªVyª{nxnl {nªZX| µÉalb {nl µMUyuyµ/2 1?Xxnun| {ªuX_

stages, can become too strict and will decrease the probability of the net finding any given boulder.

During training, much like the convolutional layer, the values in the weight matrices are optimized so accurate scores are reported. This is the last layer that processes the data.

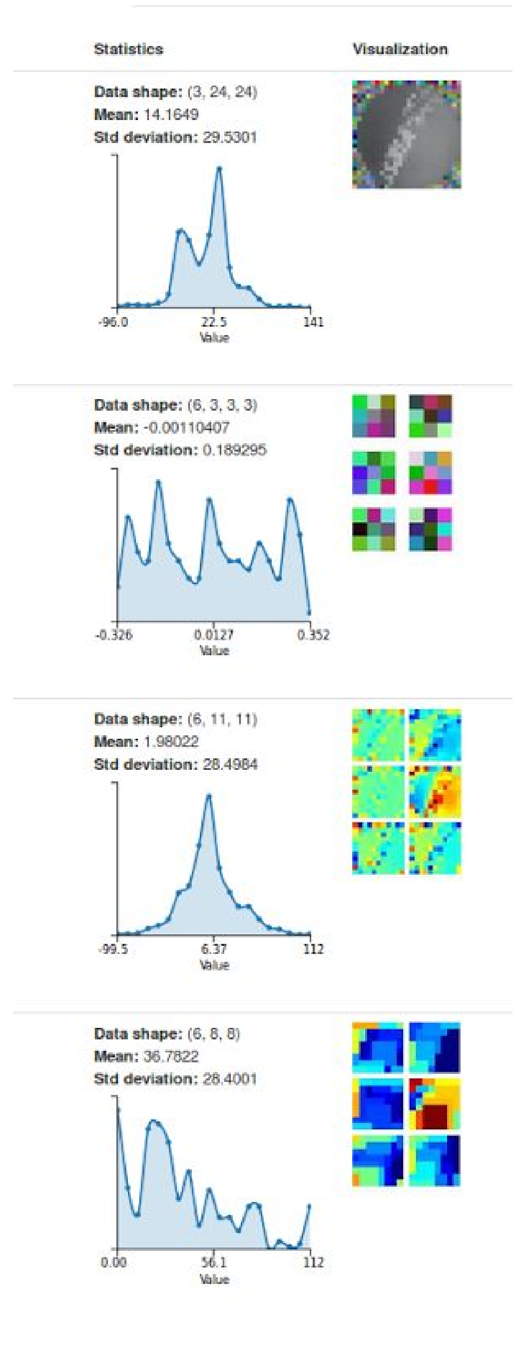


9I Ua dYB Ylk cf_ Gfi Wi f Ygk]h Ci hdi h@UWg

.

The “softmax” layer takes in the two “scores” from the fully connected layer and converts them into probabilities that are assigned to each possible classification (boulder or negative in this case). During training an additional layer, “accuracy”, also takes the output of the fully connected layers and compares the result with the expected value, the information from this is then passed back into the training function to modify the weight matrices.

J]gj U F Ydf Yg\dfcbcZdcHidf]U :]HYfg#K Y][\hiA Uf]Vig



Ultimately it is the accuracy value which the training process is attempting to optimize. Overall our network had relatively very few filters (compared to the thousands of filters on established models such as LeNet and AlexNet). This is because, one, we are only trying to detect a single object, and two, this object is relatively simple with very few distinct features other than the general shape and color which meant our network was very prone to overfitting unless the number of filters were decreased.

Data Collection

Our neural nets are trained on two sets of images. For one various images of the boulders are collected[8]. Other images without boulder were also collected from random sources. These non-boulder images are negatives - they allow the net to learn the difference between boulders and “not boulders”. After the images have been collected and pre-processed they are sorted into folders dependent on the objects which they contain. For

[8] a{{uy©µµ, „ „ ªVabZ_XZjuabªVnk µk ZXbVµuMuZxyµÞÝáä

tracking boulders only, our folders were “ball”, containing images of the ball/boulder and “negative”, containing images of all sorts of other random objects. The directory names correspond to the labels which the neural net will apply to an image after processing it.

The creation of the data set pre-processes the images to the form which the network will use to train, sorts them by label, and compresses them into a database. During this process the images are split into two categories, “training” and “validation”. It does this so the network can be trained on one set of images but tested on a different set. This verifies that the network is actually learning general features of the object classes it is trying to detect rather than learning the individual images themselves. The latter problem is known as overfitting and results in a network which is so specific that it can only identify the exact images used to train it.

Training the Networks

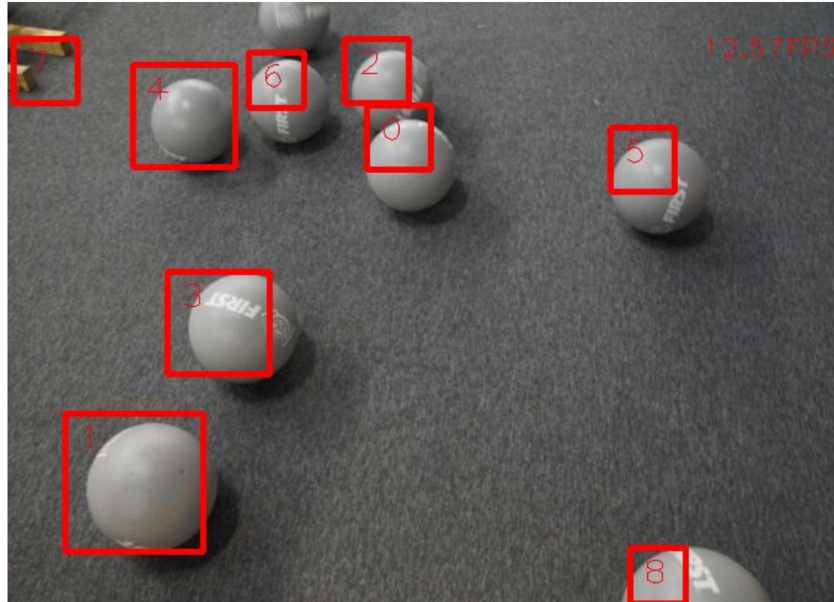
After the networks were modeled, DIGITS was once again used to complete the training process itself. In training, the computer initializes and optimizes the aforementioned weight filters in such a way that when they are applied to an image with the target object in them, they produce a higher “score” than when applied to images that don’t have the object. To do this, we used a general technique known as stochastic descent. This involves initializing the filters, running them on a subset of the images, checking how often the network made the wrong classification (the error function), then tweaking the filters accordingly and running them again in an iterative process to reach the lowest possible error.

The training rates, which determine how aggressive the training algorithm is in correcting the error, were set extremely low ($1x10^{-5}$) to prevent overfitting and the learning rate decay model was set to sigmoid with the decay starting at 85% to keep the learning rate high throughout the training process. We determined through repeated experimentation that extremely low training

rates that were kept mostly consistent through the majority of the training process produced the best results. When training a network we trained 100 epochs to give us a wide range of iterations to test.

Pulling it all Together

Our detection code is based upon a paper from a collaboration between a group at Stevens Institute of Technology and Adobe Research to create a neural network that detects faces. We made a few changes to this approach as we were detecting geometrically simpler objects than faces. Additionally, the network needed to be lightweight enough to run efficiently on our robot's NVIDIA TX1. To do this we eliminated the proposed 48x48 network and left just a 12x12 network and a 24x24 network which we found produced adequate accuracy for our purposes. Additionally, we reduced the amount of filters used in the convolutional and fully connected networks as we had a reduced number of features to detect and too many filters would both slow down the network and introduce overfitting. Finally, the face recognition system uses intermediate results from one size of network as inputs to larger nets. We decided that the benefit gained was not worth the additional complication of our network.



9I UadYNWfUJ]gjcb'Ci hdi h

Sliding Windows

The first step in processing a frame of video is generating a set of inputs to the neural net. Since the net handles a fixed sized input and is looking for a boulder which fills up the entire input image, the code couldn't just pass the full frame. Doing so would only identify a boulder that filled up the entire input image.

A sliding window method was used generate a set of candidates to test. A 12x12 pixel window starts at the upper left of the input and represents the first sub-image to test. The window is then moved to the right a fixed number of pixels to generate the next sub-image. This is repeated until the window hits the right side of the image. At that point, the window is moved down a fixed set of pixels and reset to the far left. This process is repeated over the entire image.

That approach works if we could guarantee the boulder is roughly 12x12 in every case. Since it won't be we had to use a multi-scale approach. The code makes multiple copies of the input image, each resized to different scales. The size of smallest and largest rescaled images are

calculated based on the maximum and minimum sizes of objects to detect. Intermediate sizes are at fixed ratio steps between the two. Passing fixed sized sliding windows over all of the scaled input images has the net effect of searching for variable-sized objects in a fixed-sized input but is much simpler computationally.

Our code used a fixed step size of 4 pixels in both the x and y direction. In practice this approach generates a large number of candidate windows per image. Variables in this system, such, the min and max size of the sliding window are dynamically controlled with sliders to balance performance with detection accuracy.

Implementing the Networks

In the ZebraVision software, the networks were loaded into the Caffe API. The output from the sliding window was then passed to the Caffe API with the 12x12 network which then returned a likelihood that each individual input window was a boulder. Any windows that achieved a specific confidence value were passed into the 24x24 network for verification. If the window passed both networks it is displayed on the screen and passed into the tracking system and communication system. This system allows the faster 12x12 networks to remove the majority of detection windows that are definitely not a boulder. The smaller subset of windows can be run on the 24x24 network which is much more accurate, but slower.

In a typical run, we end up with thousands of initial sub-images to feed into the 12x12 network. The 12x12 network eliminated all but a few hundred of these initial images. Given this, the 12x12 network dominates the run time. Tuning this network and the threshold confidence values to be just accurate enough but still quick was a major effort. The application can load

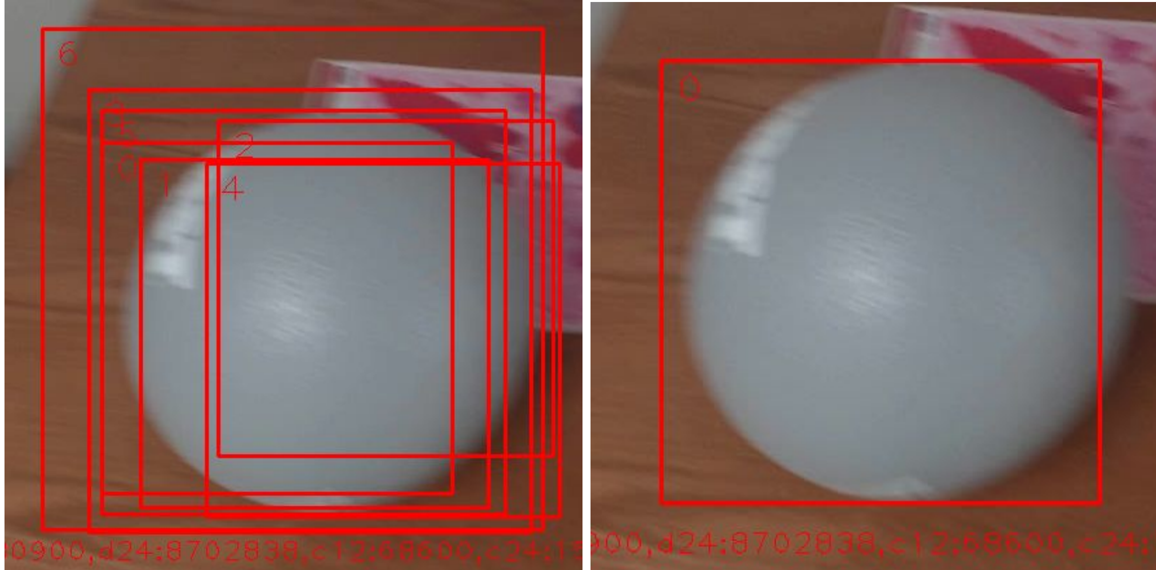
different epochs of each network in real-time to visualize the level of training needed to achieve the best results.

Non-maximum Suppression

The detection nets tend to produce several overlapping detections for each boulder. As the detection window is moved near a boulder it will catch some of the boulder but not all of it. This will give a moderate confidence level for that window. As the window gets closer to being centered on the boulder the nets will continue to produce good confidences until the window moves off the boulder. There is a similar problem when searching the image at multiple scales : the object will be detected at several similar sizes with various levels of confidence.

It is difficult to choose a confidence threshold which distinguishes boulders from non-boulders to eliminate all but the center image. Setting the threshold too high will prevent the nets from identifying boulders under marginal conditions (bad lighting, slightly blocked by other objects, etc). Setting the threshold too low will leave multiple "hits" for a given object.

The solution to this problem is called non-maximum suppression. Basically, this code looks for overlapping detection rectangles. For each set of overlapping rectangles, it keeps only the one with the highest confidence. The amount of overlap allowed can be adjusted in our code for fine-tuning. The images below show a before and after example of the code in action.



9lUadYcZbcb!aU]ai a gjddfYggjcb

Calibration Networks

Another feature, which we did not get to fully implement this season but is currently being worked on is a calibration network after the output of each detection network. The purpose of a calibration network is to better align the detection windows so that they are centered on the object that is being detected.

The calibration network is structured very similarly to the detection networks. Instead of the network determining whether it is a ball or not, it begins with the assumption that it is a ball and it determines how the image it is given has been shifted or resized from what the networks has learned is an “ideal” boulder. If a given class has a high enough confidence the code will shift and/or resize the detection window by the amount corresponding to that class.

With these new offsets the rectangle is ideally centered on the object in question, in our case, the boulder. This is important because, when passing data to the RoboRio to drive to a specific location to pick up a boulder, the location in question should ideally be in the center of the ball. The advantage of running the calibration network between the 12x12 network and the 24x24

network is, if the detection window is actually a ball, the calibration network will center the window before sending it to the 24x24 network. This allows us to set the detection threshold much higher as detection windows that are actually balls should be detected with extremely high confidence as they will be correctly centered like training images.

7U]VfUh]cb bYhg]bUh]cb"?M`ck ]gh\Ycf][]bU XYhYWh]cb"F YX]gh\YWU]VfUh]cb bYhk cf_ UX^ ghYXXYhYWh]cb`

`cWUh]cb"

Training Calibration Networks

Calibration networks require shifted training data to train the calibration nets to quantify how much an object was shifted. We used 3 separate x and y shifts, and 5 size shifts (including a value that indicates “no change needed”). For each permutation of these shift/resize values, we used a formula given by the aforementioned research paper in section 3.2.2 to shift the bounding box of the ball, to show the neural network what the relative locations look like.

The images of the ball we had are fairly tightly fitted. This caused problems with resizing or shifting off the edge of our input images. To solve this, we expanded the image by the image’s

width on every side, resulting in a new image with the ball in the center of an image 9x the initial size. We then set the rectangle of interest to the original image's area, and shifted the rectangle based on the formula above.

Each possible shift permutation was labeled with a number 0-44. We wrote each separate shifted/scaled output image into a subdirectory corresponding to the correct label. This produced a net with 45 different outputs. The confidence for a given output label was how much the net believed a given input image was misaligned by that shift/scale value. The output of the network can then be directly used to correct the initial bounding box output by the 12x12 network.

[illegible]

[illegible] *Computer Vision Foundation.* [illegible]

[illegible] *Stanford University.* [illegible]

[illegible]

Thanks to our vision team mentors Eric Blau and Kevin Jaget! They were instrumental to guiding us in creating a functional neural network system. Their mentorship and dedication throughout the year has been most appreciated.

Also a huge thank you to NVIDIA whose sponsorship and continued support for our team's work in vision has been incredibly helpful. They've been great at giving us support for their mobile computing solutions, the Jetson TK1 and TX1, which we've been using on our robot for the past two years and with which this level of computing was made possible. Additionally, thank you to