Identifying RoboCup MSL Robots in distorted images, using a fast Neural Network with a minimum number of examples

Peter van Lith
Technical University Eindhoven, the Netherlands
vanlith.peter@gmail.com

April 24, 2018

Abstract

This paper describes a project to classify and localize RoboCup soccer-playing robots from their camera images, using Neural Networks. For RoboCup MSL robots it is important to extract information from their environment during a competition. The main sensor is an omnidirectional camera which provides a 360 degree view of a part of the field where the robot is operating. Among other things the robot needs to find out the location of other robots and identify them as belonging to the same team, or the opponent. In addition it needs to recognize the ball and possibly other objects like the referee.

Analysis of the omnidirectional image is currently done with industry-standard vision technology that can detect the presence of a robot but not its identity. With the use of Neural Networks we hope to be able to identify the robots on the field and collect more detailed information about the game situation that will allow the robot to learn new behaviors that will gradually improve its performance over time. This first step is aimed at collecting the necessary information to create such a system in a follow-up project.

Because the appearance of the competing robots is unknown in advance to the competition, training needs to be done on-site. There are in total 12 robots on the field and all robots of a team are similar. So there is a very limited number of samples. The onidirectional camera distorts the image, so the training program needs to create samples that are identical to these images, The robots have limited processing capabilities, so the network must be small and fast enough to identify robots moving at speeds of up to 3 m/s.

I. Introduction

In this document we describe the first step in a search for an approach based on neural networks that must learn to recognize and locate robots in the Tech United Turtle robots. The Turtle robots are competing in the RoboCup Midsize League and are 3-wheeled omni-directional robots, equipped with on-board computers, an omni-directional camera, a Kinect camera and a Jetson GPU board. The software for this platform is completely written in C++ and MatLab and uses conventional vision technology to recognize and localize robots.

Most Neural Networks are currently created using standard software platforms, classifying hundreds of classes and are being trained with millions of input examples. They usually take hours or days to train and result in large networks that require powerful computers to run on. This is very different from our case, where we basically have two types or robots to identify as well as the ball and occasionally people. The number of examples is therefore rather limited as is the available processor power of the robots. So this project is atypical in that it works with a small, shallow network, has a limited number of classes and very few examples. The main question to be answered in this paper is therefore: Is it possible to train a small Neural Network to reliably recognize a limited number of object types, given only a few examples. An additional question is if this network can be trained during a competition and then run on an embedded platform.

The main aim of the project is to create a collection of Neural Networks that in the future, will allow the Turtle robots to learn new behaviors in such a way that it forms a hybrid of existing software and a self-learning system. We want the robots to recognize unknown game situations and then allow a simulator to learn how to treat these novel situations and then transfer this new behavior to the robots.

The actual implementation of this idea will take place in a smaller version of the Turtle, called the mini-Turtle, running a Raspberry-Pi and a Movidius Neural Compute Stick, so we aim for small and fast networks that can operate in real-time on an embedded system. We will also briefly describe a roadmap for the development of the learning environment and the structure of the robot behavior, employing a merger between existing and deep learning technology. We

are proposing that by combining both technologies we can achieve a better control over when and where deep learning is appropriate, while at the same time gaining insight in the way new behaviors can be learned and integrated into the existing system.

In this first step of the project we intend to use two small Neural Networks, that will work together. The first one is a small network, based on the MNIST and CIFAR10 examples with a number of important adaptations to allow classification of the robots, the ball and people on the field. The second network locates the robots in the omni-directional image and uses the first network to identify each robot.

In most multi-object classification tasks, a variety of different objects will need to be classified, so networks are trained with 100 or more classes using many examples of different shapes, orientations and sizes. In our case there is a limited number of objects; a Ball, Robots of our own team or the opponent's team and possibly people like the referee. Robots of the same team have generally the same shape, with the possible exception of the keeper. All robots of a team wear the same kind of shirt and have a recognizable number to identify them (Figure 1).

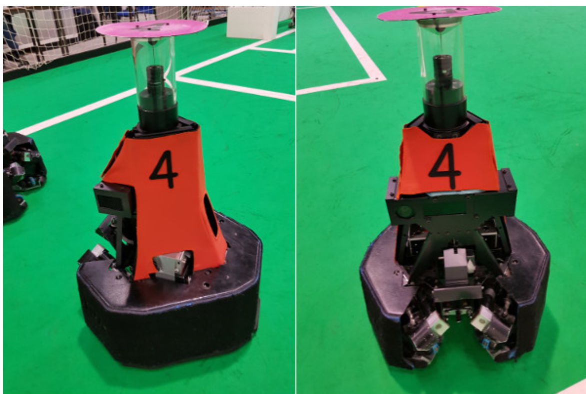

Figure 1: *MSL Turtle, side and front view*

Robots need to be classified according to the following criteria:

1. Classify robots as belonging to our own team or the opponent. This is the most significant property.
2. Determine the orientation of the robot. Is it facing us, or moving away. So we want to know if it is Front, Left, Right or Backward facing.
3. Determine the identity of a robot. This information is important when calculating a trajectory. Especially when robots move at high speed and come together closely, it may be hard to keep track of every individual robot. Having access to its identity may help in these cases. However this is a task, which is very difficult for distant robots or when occlusion occurs. Although we started out with MNIST to tackle this problem, we currently do not include this classification level.
4. In addition to this we also need to classify the Ball and People who might be moving around the field, like the referee with possibly an orientation as with the robots.

During a competition it is very important to have this kind of information available at high frequencies in order to keep track of fast moving objects. The simpler the network, the faster it will be. But in addition to identifying a robot, we also need to know its position on the field. Locating multiple objects in an image is the subject of many research projects and many proposals have been made. We will first discuss the various problems and will then describe how we tackled this problem. Existing solutions to the localization problem are discussed in the appendix.

An image recognition pipeline generally consists of three stages, and we will describe our approach to each of these stages step by step in the following sections.

1. Feature Extraction. We will describe the various possibilities and considerations for our system.
2. Object classification. We describe how our system classifies objects.
3. Object localization. This forms the main body of the current stage of the project.

II. Feature Extraction

In Computer Vision technology, the recognition of objects is done by selecting a collection of features that describe the object to be recognized. Techniques that are used often are: Haar features, Histogram Of Gradients (HOG) or Scale Invariant Feature Transform (SIFT). As a result of the success of Neural Networks, current interest concentrates more on automated feature detection capabilities of these networks.

While most of the recent work concentrates on Convolutional Neural Networks (CNN), other techniques have

been developed that are in some cases equally successful. Because we are interested in a simple and fast solution, some of these approaches might also be of interest. A very simple approach with a single layer network, using K-means [2] shows results that are on a par with deep convolutional networks. Here three criteria proved important; using feature sizes of 6x6 or less, using large numbers of features (> 1000) and the use of whitening of the input 2.

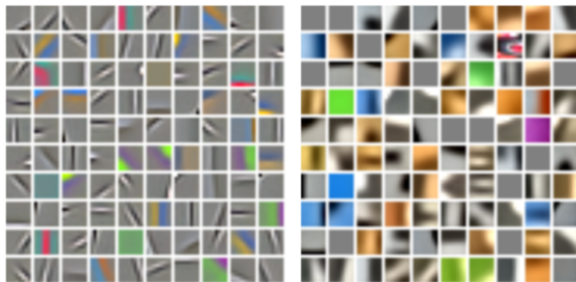

Figure 2: *Features selected with K-means with and without whitening*

We are interested in developing a simple, end-to-end training facility that can also run on a small GPU and will therefore concentrate on an implementation using a convolutional network, but will take these findings as an important consideration. For a historical overview of CNNs, see [20]

To get started we are using a small Convolutional Neural Network (CNN) that is trained to recognize robots from our own team and those from the opponent. We started out with the camera images (see figure 3) and developed a program to help with the selection of robots in bounding boxes from these images. Initially all robots had a standard number plate with the colors Cyan or Magenta. But then a change in the competition rules allowed the usage of shirts, which left us with no information about the actual appearance of the robots prior to a competition. So during a competition, we take pictures of all robots and then use these to train a new network for every match. Because the network is kept very small, it can be trained on a desktop computer with a GPU in a short amount of time. This network is then used as the basis for the localization program.

The input for the network is a collection of pictures, taken with a mobile phone camera with their background removed. This picture is then distorted to look very similar in shape to the image of the omni-directional camera and rotated at several angles, so we get about 1000 versions of every robot in different orientations (B/F/L/R), color variations and other augmentations. The network is trained with these examples and then verified against samples, taken from the Turtle camera (Figure 3).

All input pictures are scaled down to the smallest versions that are visible in the Turtle image, which in our case is 64x64 pixels. When taking examples from the Turtle images for verification, the position of the robot in the image is used to scale down the robot to this 64x64 image size. Because the aspect ratio of all robots is the same, it is easy to take a robot from any position in the image and calculate it's distance, size and rotation in the picture. This way the network only has to learn a single size robot and can concentrate on the team identity and orientation of the robots.

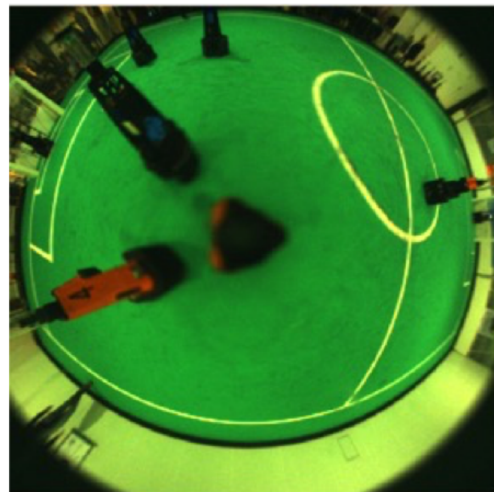

Figure 3: *Image from the Turtle camera*

To find the best architecture for this network we first developed visualization software that gives insight into what has been learned by the network. With this tool we are able to find the best structure and number of filters to classify the objects reliably. The fist step is to find the best way to learn features that properly classify the robot. Our program allows a large number of parameters to be modified and with this we run many experiments with different architectures and measure how reliable the augmented and transformed input images train a network to recognize robot images, selected from actual

Turtle images.

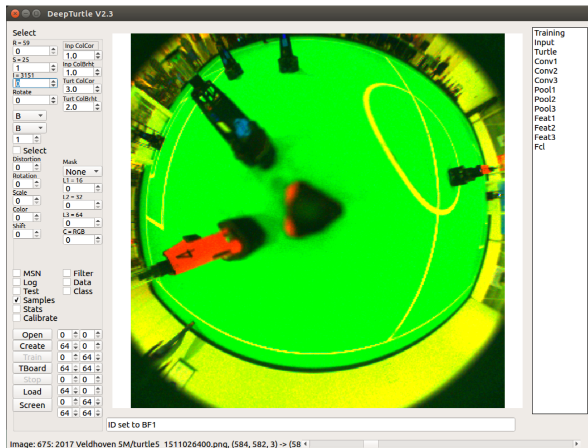

Figure 4: *User Interface of Network Training program*

Figure 4 shows the program that is used to train and test the network. During every experiment it keeps a logfile of all settings, the training time and the error rates of every training batch, the error on the testset and on the validation set. The validation set consists of examples for every object type, lifted from the Turtle camera. We use the output of this program to show the results of all experiments.

i. Hyper Parameters

When setting up a Neural Network there are many factors to consider like the number of layers, the number of features or the kind of pre-processing of the images. These settings are referred to as hyper-parameters and many different configurations are possible. The art of configuring a Neural Network is to find the best combination of hyper-parameters to train a network for a particular collection of input images.

The amount of variation that is the result of the hyper-parameter settings determines the number of neurons of the network, normally referred to as the parameters of the network. The more parameters a network has, the more different properties of the input images can be learned, but more parameters also make the network larger and require more processing power. So our task is to find the best combination of the number of parameters, reliability of object classification and the time required to perform the classification.

Our experiments in selecting the best network configuration evolve around the following steps:

1. Avoiding Overfitting
2. Color Correction
3. Input Distortion
4. Input Augmentation
5. Data preparation
6. Background Selection
7. Feature Size
8. Number of Features
9. Number of layers and dimensions for every layer
10. Configuration of Fully Connected Layers (FCL)
11. Performance of Feature Selection Process
12. Training Time

ii. Avoiding Overfitting

One of the major hurdles to overcome while building a Neural Network is how to handle overfitting. Overfitting happens when a network learns the inputs too well and as a result looses the ability to recognize new examples. The way to prevent this is to provide many different examples and to limit the number of parameters of the network.

Hopefully such a combination will contain sufficient information to classify the target objects. Because our aim is to train a network based on a small number of classes, where the robots are almost identical, the risk of overfitting is large. During many of our tests the networks show the symptoms of overfitting. When the loss factor (difference between the input class and the predicted class) continues to decrease and the error on the test-set stays stable or decreases, while the validation error is getting higher, the network is overfitting. This means that it is learning to represent the input too closely, thereby losing generality and thus the ability to predict unseen examples correctly.

In Figure 5 the red line represents the loss, the green line represents the error on the test-set and the blue line the error on the validation-set. While the Loss and Test error keep getting lower, the validation error initially was in sync, but then starts to rise.

In our case the validation set consists of samples, lifted from the actual Turtle images, thereby representing the target objects that are never seen by the network during training and therefore form a very good indicator of the performance of the network. So when a test results in a high validation error, this can mean that the network has insufficient information, or that it might be overfitting.

Overfitting can be decreased by providing more examples, stopping the training at an earlier stage or by

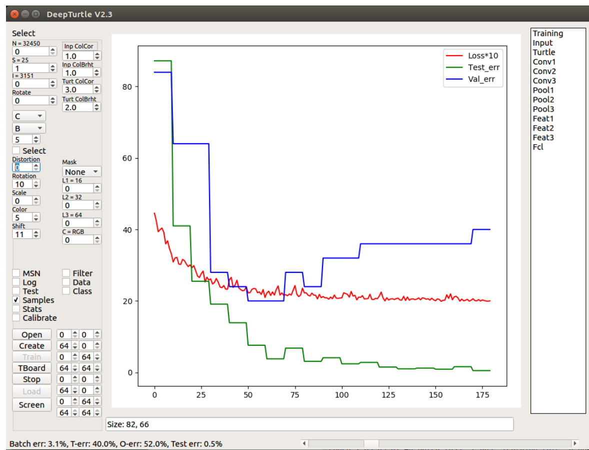

Figure 5: *Example of a network that is overfitting*

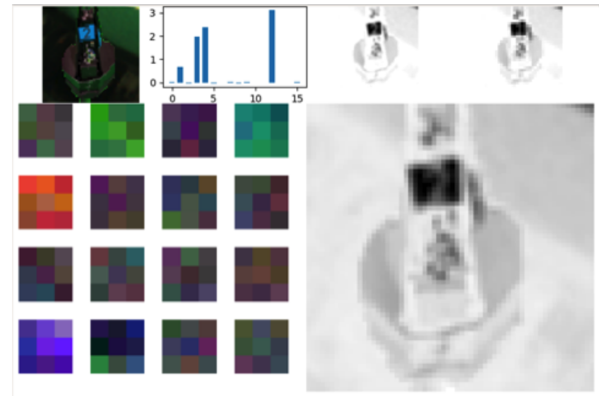

Figure 6: *Selected filters in the first layer*

decreasing the learning rate. We have tried all possibilities as part of the network design process. We cannot increase the number of examples very much, because of the limited number of robots involved in the process. More variations of the robots does not increase the information that is contained in these input images.

The process of finding the optimal network configuration is an iterative process, driven by trial and error. Almost all hyper-parameters influence each other, so the descriptions that follow are not the result of a sequential process, but of many tests, shuttling back and forth between changing the configuration of the network to find out the influence of each of the hyper-parameters.

iii. Color Correction

One of the first things that we found out is the large difference in color saturation between the phone images and the Turtle camera images. This can be clearly seen in Figure 3, where the blue number sign in the Turtle image is very faint. So we introduced variable color corrections on both the input- and Turtle images, so they were more comparable. Then we started experiments with several filter sizes and number of features. The first round confirmed reports from several of the articles, referenced in this paper, that 3x3 filters for every layer worked best. We also experimented with 'Valid' versus 'Same' padding as this is used in TensorFlow and found that 'Same' padding, which keeps the spatial dimensions of a layer constant works best.

In Figure 6 the filters for the first layer are shown. The orange of the shirt and the blue from the opponent's robots are clearly visible. The bargraph shows that there are 4 important features in this picture and the large image shows the activation layer. This information is then going through a pooling and RELU layer and becomes input the the second layer. In this example the ball was not classified correctly because the yellow of the ball is not present in the selected features. So more color corrections were required to make the these colors also being represented in the filters.

Table 1: *Color Correction*

Color Correction	Test-err	Val-err
No Correction	0.3	48.0
Input Correction	0.7	36.0
Turtle Correction	0.5	40.0
Input + Turtle Corr	0.4	16.0

Table 1 shows the difference in recognition between color correction and no color correction on both the input and the Turtle images. The results however are not fully representative because the training of the network with uncorrected data seems to be overfitting, which drives the Validation error to a higher level. We discussed this in section ii.

iv. Input Augmentation

For every robot in a team there are be four pictures, one for every orientation (Back, Front, Left and Right). We remove the background using Gimp and replace it with a solid green background. This results in a total of 20 robot pictures for every team. We add another set of pictures for the ball and some people, resulting in a total of about

60 input pictures. Because 60 pictures is not enough to train the network reliably, we use augmentation to generate between 500 and 1000 variations for every robot image. The same is done with images of the ball and of people around the field. During augmentation, the green background can be replaced with other backgrounds as described in section vii. Every robot picture may then be augmented by a number of versions of every picture, using the following parameters:

1. Distortion. Distorts the image based on the properties of the Turtle camera mirror as described later.
2. Rotation. Rotates left and right for 45 degrees to resemble images in the Turtle camera. We may add more rotations later on if this proves necessary.
3. Scale. Images may be scaled. This is only a limited scaling, because the input image is 64x64 pixels. In the camera image the robot size varies between 64x64 and 128x128 pixels, but is always scaled down to 64x64. So scaling is only done within the 64x64 images and is limited to about 10 pixels.
4. Color. This is the color variation of the input images, to represent various lighting conditions.
5. Shift. These are random shifts of the image in various directions.

Please note that we do not use flipping, since the robot orientation is an important property to be learned and flipping will also change the robot's number plate. Also up/down flipping would interfere with the rotation, caused by the Turtle mirror. The number of variations for every of the augmentations determines the total number of input images that the network will be trained on. If it proves that this is insufficient information to reliably train a network, we may create additional input images by taking more photographs of the robots. Removing the backgrounds however will involve additional preparation time during a competition, but fortunately that only applies to the robots of the competitors.

In all test-cases we measure the performance of the network through the error rate of the validation set. The second measure is the variation in feature selection, used to recognize the images. We express this as the top activation levels of all activation layers of the network. The larger the variation and activation levels, the more accurate the recognition process will be. We will be showing examples of this in section viii about Feature Sizing.

Table 2: *Distortion Correction*

MSN	Train-err	Test-err	Val-err
Undistorted Input	3.1	0.5	40.0
Distorted Input	0.5	0.5	12.0

v. Input Distortion

The Turtle's omni-directional camera uses a conical mirror to get a 360 degree image of the field around the robot. Objects in the camera field are distorted following a uniform pattern in the entire image. We therefore take the original image, taken with a mobile phone camera and distort it to make the input images similar to the images of the robots in the Turtle camera image.

For this we use an affine transformation, which is controlled by a distortion matrix. The training program provides this matrix, which can be set with a number of default values or can be manipulated manually to test different transformations. The first experiments are done to find out how much distortions influence the training and recognition behavior of the neural network. Figure 7 shows the difference between the input image and the distorted image, used to train the network. Table 2 shows the results.

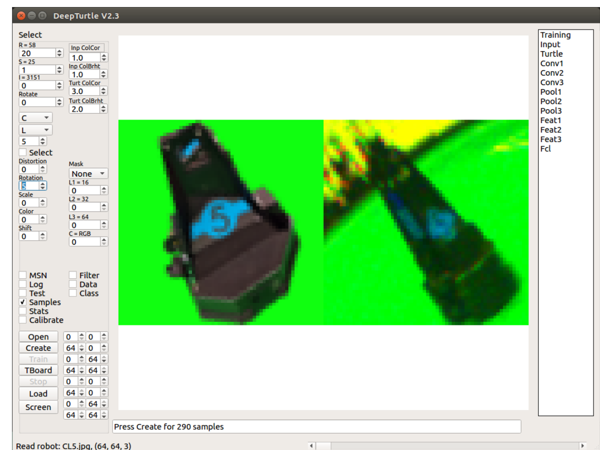

Figure 7: *Difference between input image and robot in Turtle image*

To enhance recognition further, we include 8 different variations on the distortion, each slightly varying the parameters of the distortion matrix. In Figure 8 the result of distorting the input image against the Turtle camera image is shown.

Of course all parameters influence the final results and

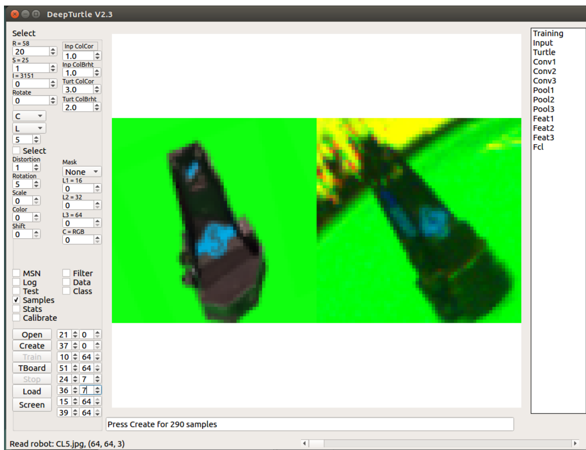

Figure 8: *Both images after distortion of the input*

the settings of the network hyperparameters are highly intertwined, so other factors are also of consideration. Anyhow, we can see that the shape of the robot has a large impact on the recognition performance and the feature selection process that forms the basis for this recognition, as we will see later on.

vi. Data preparation

There are many factors that influence the selection of features. First we concentrated on preprocessing of the images. There are three different methods commonly used: Mean Subtraction, Normalization (MSN) and Whitening. The last one is generally not used in CNNs, and also requires substantial processing time, which we cannot afford when running a competition, so we did not use this one.

So we concentrate on Mean Subtraction (also referred to as Zero Centering) and Normalization. The total number of input images is split into two sets; a training set and a test set. Additionally a third set is generated used for validation. This can be either a split-off from the training set, but in most cases we take samples from the Turtle images, since those are the target of the training process.

All input sets are converted from RGB pixel values to floating point numbers ranging between -1 and +1.

We then calculate the mean value of the training set and the standard deviation. We subtract the mean from all three sets and divide all three by the standard deviation. This causes zero centering and normalization of the input. According to the literature this process makes the input more easy to train.

Table 3: *Normalization*

MSN	Test-err	Val-err
Without MSN	2.1	20.0
With MSN	0.1	76.0

The influence of this process is expressed as the error rate on the test- and validation sets and the speed of training in table 3.

It seems that using MSN in our case makes things worse. One reason could be that all input images are created in the same indoor environment, so they are no natural pictures. The validation error started out high and remained high, so the network was not overfitting, while the version without MSN reached a low error rate sooner but raised later on a little bit. So for now we are using the raw input without any mean subtraction or normalization.

vii. Background Selection

In order to let the system concentrate on the features in the input images, we included a facility to select different backgrounds. We can replace the green background with other backgrounds to see which type has the highest recognition factor. By eliminating the background from the image and replacing it, we make sure, the network does not learn the background as part of the image. We included the following options:

1. No background. We leave the solid green background untouched.
2. Black background. By gving a black background, the background is eliminated. This may have an influence since all robots are black, with the exception of their shirts.
3. Random Field background. This takes random patches from Turtle image backgrounds. It includes typical backgrounds that are different for every generated image. We must take care not to include other robots or objects as part of the background.
4. White background. A white background may create a higher contrast with the robot part of the image.
5. Random Pixel background. This generates random noise as the background.

Table 4: *Background Selection*

Background	Test-err	Val-err	Remark
None	3.7	16.0	
Random	0.8	24.0	Constant level
White	0.1	32.0	
Black	1.3	80.0	Constant High
Field	1.6	88.0	Growing higher

Table 5: *Feature Sizes and Layers*

Features	T-err	O-err	Remark
5x08-3x16	16.0	44.0	
5x16-3x32	24.0	52.0	
3x08-3x16	36.0	56.0	
3x16-3x32	28.0	52.0	Slight overfit
3x32-3x64	40.0	60.0	High loss
5x08-3x16-3x32	48.0	76.0	Starts high
3x08-3x16-3x32	20.0	44.0	
3x16-3x32-3x64	12.0	40.0	
5x16-3x32-3x64	28.0	40.0	
3x32-3x64-3x128	36.0	48.0	Slight Overfit
3x08-3x16-3x32-3x40	10.3	60.3	* Third best
3x08-3x16-3x16-3x16	11.1	70.4	
3x08-3x16-3x32-3x64	17.6	67.6	
3x16-3x32-3x32-3x32	4.1	65.3	* Best
3x16-3x32-3x32-3x48	7.4	61.8	* Second Best
3x16-3x32-3x64-3x128	10.3	63.2	

viii. Feature Size and Number of features and Number of Layers

Based on the best performing settings we then started varying the number of features and feature size first in a network with two layers.

Table 5 shows that the four-layer versions with all 3x3 features performs best, while the two-layer version with 5x5 features in the first layer also perform reasonably. We will therefore further concentrate on these versions while looking at more alternatives.

Not only the performance of the network is of interest but we also want to know what the network actually learns and what kind of information is used to make the selection. Also the size of the Fully Connected Layer is of importance. We also see that the error rate on the orientation remains around 60% and we would like to get that down also.

So we want to have a good way to find out what the features that the network has learned actually represent. We have developed a set of visualization facilities to make this clear. First we will look at the features, selected in the first layer.

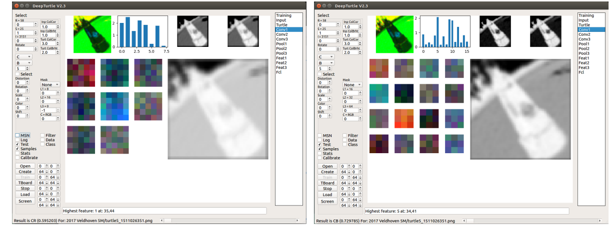

Figure 9: *First level features 5x5 and 3x3*

In Figure 9 The features for the first level are shown. Although the 5x5 features lead to a better recognition, they seem not to represent the actual colors in the images. The 3x3 features doe a much better job here. Also note the little graphs at the top, that represent the activation levels of the first layer.

ix. Adding new objects

The network correctly classifies the robots, with some errors on the orientation of some robots and problems when multiple robots were seen in a single image. But we also need to identify the ball and perhaps people on the field. When we added these new objects, the whole recognition system collapsed. It proved that the yellow ball was too much like the lines on the field, while the dark colors of the opponent's robots made it very difficult to distinguish between people and robots. Images of people disturbed the recognition process very much, so we eliminated these for all following tests. In order to make the recognition more fine-grained, we added another convolutional layer to the network. That did not seem to help much and as a result we noticed many dead features. The Sudden Death Syndrome (SDS) is caused by negative activation values, that get eliminated by the RelU layer. A possible solution to this problem is to decrease the learning rate, so we made this into another hyper-parameter and retrained the new network. This was an improvement, but the color differences between the ball and the lines on the field are still a big influence.

Another thing that does not help is the fact that in the distorted images the distinction between left and right on the Cyan robots is too small to notice, even to the

trained eye. So mis-classifications on this type of robots is unavoidable.

x. Training Time

One of the reasons for starting this project is to develop a fast Neural Network, that can be trained on-site during a competition. Because we do not know the appearance of the robots of our competitors in advance, we can only train the network once the competition starts and we can collect pictures of all opposing teams.

We will have to preprocess all pictures and then train an network for every team that we will be playing against. This should be able to be done in a small time frame. Our aim for training the network was about 15-20 minutes. Luckily with the small network, running a GTX-1070 GPU we achieve training times, considerably less than this time. Currently training time is about one minute, which compares favorably with the training time of transfer-learning models, that can take hours or even longer.

Selection of the background seems to influence training time. The system needs to learn many more parameters, which apparently takes time. But as described before, using a neutral background achieved the best results.

III. Identifying Objects

To identify individual robots on the soccer field, we need to use the Turtle images and use the trained network to classify every robot in the image. The small network was trained with the intention to learn the features to identify robots and we want to apply these learned filters to the full Turtle image. To better understand what the learned feature filters represent, we visualize the contents of the filters by implementing a simplified version of the CNNVis system. [26, 27]

i. Performance of the Feature Selection Process

It becomes much clearer when we can have an overview of all activations of a layer, side by side, so we can see what features are actually selected.

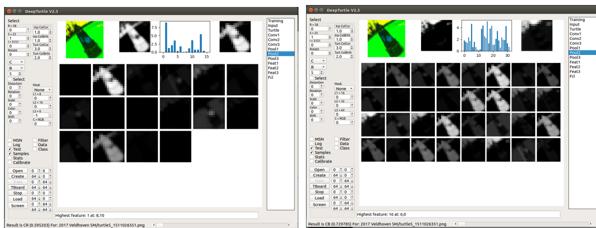

Figure 10: *Activation map of the 3x16 and the 3x32 layers*

In the 3x16 activations we see that the dominant feature (shown in the small image in the top) is the black shape of the robot, while in the 3x32 layer the dominant feature seems to be the top left corner of the image, where no robot is visible. To find out what is going on we would like to understand what these network features actually represent. To analyze this we use another visualization tool, that randomly takes 1000 samples of the generated input images and finds the 9 highest activations for every feature. It then shows these for the selected layer, so we can see what the network is looking for.

In Figure 11 is becomes clear that the orange and cyan number plates are important selected features as well as the black from the robot bodies. The yellow from the ball seems to be missing. The 5x8 features however also show combinations of green and orange and green and yellow. Because every layer combines the features from the previous layer, we need to find out what these other features represent.

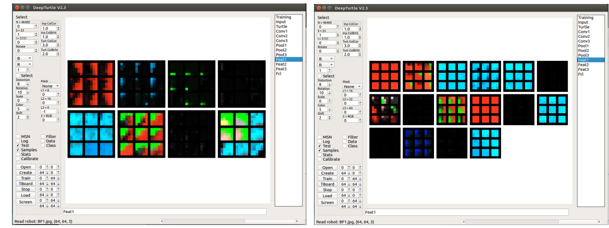

Figure 11: *Selected features for the 5x8 layer and the 3x16 layer*

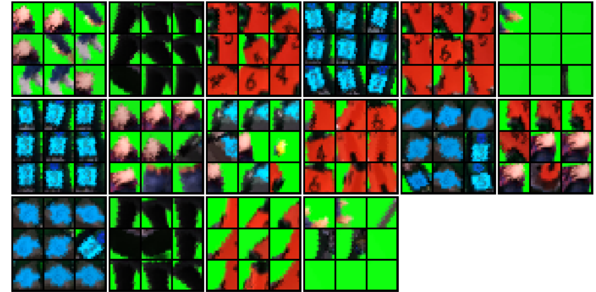

Figure 12: *Selected features for the 3x16 layer*

Figure 12 shows the most important 9 activations for the 16 features of the second layer. Here we clearly see for instance that one feature selects the front cyan number plate, while another one selects the left or right cyan number plate. The same holds true for the orange shirts. For deeper (and bigger) layers we expect many more features to be selected, as we can see in Image 13

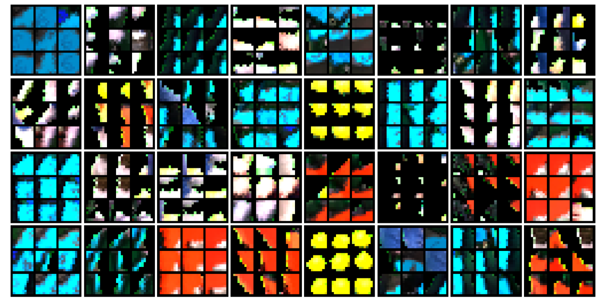

Figure 13: *Selected features for the deepest 3x32 layer*

Here we see many more features and it is very well possible that here a single feature does represent an entire class. Therefore we also calculate the class variation for all features. The number in orange shows the average class number, indicating that the feature is shared by several classes. A blue number indicates that all selected features belong to the same class and that the feature

actually represents the entire class. If all classes can be represented this way, there is a good chance that we will not need an additional Fully Connected Layer, but are able to implement a Fully Convolutional Network instead. This is shown in Figure 14

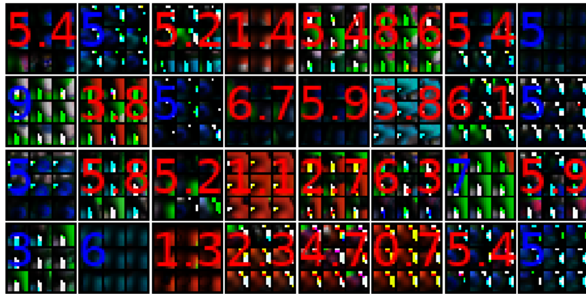

Figure 14: *Class indicators for the deepest 3x32 layer*

IV. Locating Objects

Identifying robots is not sufficient, we also need to know their location. Because we know the properties of the convex mirror, we can calculate the real-world coordinates of every robot, given its position in the image. So we want to find out where every robot in the picture is located. In order to do that we want to apply the learned network to the bigger Turtle image.

In most of the studied approaches some kind of search is made in the image to find possible objects and then these are classified, using the trained network. Region Proposals are commonly used for this [9, 8, 23, 21, 18]. But we have seen that the feature maps select properties of the robots that might help in finding the Regions Of Interest (ROI). Can we use the small network to show the feature maps of robots in the larger Turtle Images.

i. Conversion to Fully Convolutional Network

The purpose of the Fully Connected Layers is to perform the actual classification. The last Pooling layer of the network connects all its neurons to an intermediate layer, which in turn connects to the final layer, which has one neuron for every class. Through this mechanism the training process can determine how every feature in all layers needs to be updated to let the network recognize the objects.

However, in the localization stage we will need to classify all objects in an image, using these learned features. The design of the FCL makes scaling the small network to the much larger Turtle images impossible. The solution is to convert these layers to a convolutional version. Fortunately, FCL layers are equivalent to Convolutional Layers [5], so we can easily convert them and this way we get a scalable network. .

This conversion turns the network into a Fully Convolutional Network (FCN) [19]. One of the properties of a Fully Convolutional Network is that it is scalable to different input sizes, so we do not have to train a much larger network with different sized objects, which would require considerably more time, since the Turtle images are 582x582 pixels as opposed to the 64x64 pixels of the small trained network.

So we define a new FCN with the same number of layers as the original, but with the full size of the Turtle image, and apply the learned filters to it. With this modification we now have a way to visualize the Turtle images layer by layer and by selecting the proper features, we can show the different activations for every filter in all layers of the Turtle image network. But when running an inference on this network, it still classifies just a single robot in its final layer.

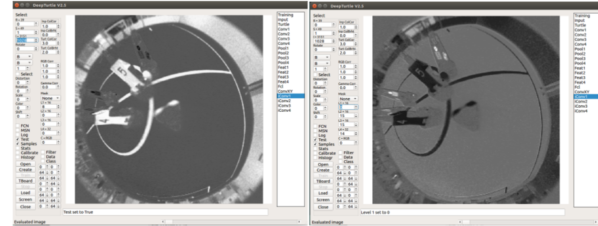

Figure 15: *Activation map for selected features of a Turtle image*

Because these learned features do not represent a classification, but a selection of parts of images that occur frequently, we need to find a way to make the classification of features represent a class. There are two methods mentioned in the literature, that address this problem:

1. Class Activation Maps. [29]
2. Pixel Based Image Segmentation. [19]

ii. Using the feature maps as class selectors

We have seen that the small trained FCN is capable of activating the filters on all robots in a Turtle image. The task of the final layers is to combine all these features into a classification. But could we not use the learned filters directly to create a classification. This is actually being done by combining all feature activations for a single class into a Class Activation Map [29]

So in our network we create a Class Activation Map as part of the visualization process. These CAMs are now used instead of the final classification layers. By using the CAM, we can run an inference on the network and see for every class, where the robots are located in the image. All we need to do is to find the center of the activated pixels. There is however a problem. The deeper we get into the network, the smaller the image gets and the best classification if achieved on the deepest level. Another method is to classify every pixel in the image and create a segmentation this way as in [19]

In both we need to take the information of a deep level and up-sample this to a higher resolution and then

apply the CAM to this input. To upscale the image we use the Fractionally Strided Convolution (FSC), called transposed convolution in TensorFlow. With the up-sampled Class Activation Maps we can now find the location of every class of objects in an image.

We are actually using a third method. During the visualization as described before viii we analyze which features in each layer are used to discriminate every class. By combining this information, we are able to create a complete list of all features of all layers that contribute to the classification, as an alternative to the final layers of the network. This makes it necessary to combine the various activation maps at different levels. So we combine the activation maps for all features of a class. First we upscale all feature maps to the same resolution and then we combine them into one activation map. This is done efficiently using the á trous algorithm [10]. This Class Activation Map is then used with the Turtle Network to classify every robot, resulting in a number of classified layers. In every layer we find the location of the highest activation and treat this as the center of the robot.

Using this classification we generate one picture, that represents the relative positions of all robots that have been found in the Turtle Image, which represents the current Game State and is used for the learning environment.

iii. Interface with the Learning environment

The recognition of robots in the entire image is finally converted to world coordinates in a bitmap with stylized object markers to serve as input to the learning algorithm as shown in 16.

This image serves as input to the learning environment and serves as in indication of the game state, the robot is in. In this learning environment the game state is compared to the existing State Transition Network (STN), that is present in the Turtle robots and which controls the actions the robot is taking. This network is hand-coded as are all actions. The purpose of the learning environment is to determine if the current Game State is a known state in the STN. If this is a new state, the learning environment will be trained to find the best solution for this game state and show its solution in the form of a sequence of steps, leading to a better, preferably known game state. These steps are shown by the

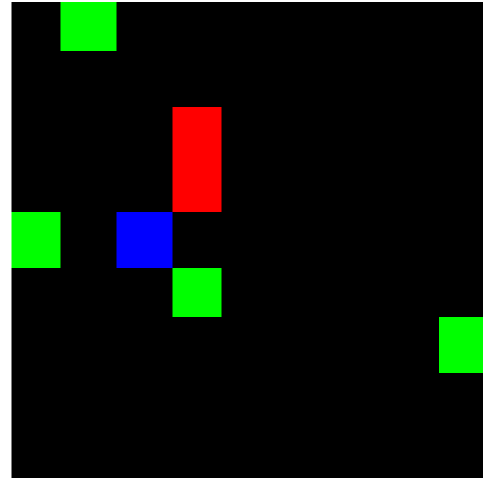

Figure 16: *Game State Image, showing position of nearby robots*

learning environment, so the developers can decide if they find the learned solution acceptable. If this is the case, the learned behavior is transferred to the Turtle robot, along with the new game state.

This allows a hybrid solution between the current, hand-coded environment and a learning environment, allowing both approaches to co-exist in the same robot. This permits the execution of well established and highly tuned game scenarios along newly learned behaviors. There is also the possibility to take the learned behavior and hand-code this for more efficient execution. It also gives full control over what is being learned by the robots, providing both insight in what the robot will do in given circumstances as well as the ability to exclude undesired automatic learned behaviors.

However this part of the proposed system is in a very early stage and will be the subject of a future paper.

V. Conclusions and Further work

We have shown that we can train a small Fully Convolutional Network (FCN) with a limited number of examples to correctly classify robots in the distorted image of a RoboCup MSL Soccer Playing robot. By using the learned filters and combining the learned activation maps, we can both classify and locate robots in the image and from that calculate the positions of the competing robots in the field.

This information is used to identify the Game State of the robot and will be input for a learning facility that will find out how to act in the given circumstances. This work is part of an ongoing project, where the next step will be to match game situations with a Game State Network, constructed with hand-crafted code. This will allow newly trained behaviors to be merged with existing conventional behaviors and resulting in a hybrid solution, where existing technology can co-exist with self-learning systems.

This project was started out of interest in combining Good-Old-Fahioned-AI technology with modern self-learning systems. In addition it will allow getting better control over what is being learned and facilitates the merger between existing hand-coded systems with learning systems. We would like to thank the Technical University of Eindhoven and the community of RoboCup Soccer teams in the region for their help and support.

References

[1] François Chollet. Xception: Deep learning with depthwise separable convolutions. Google Inc, 2016.

[2] Adam Coates, Honglak Lee, and Andrew Y. Ng. An analysis of single-layer networks in unsupervised feature learning. Computer Science Department Stanford University.

[3] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks. 2016.

[4] Daniel C. Dennett. *From Bacteria to Bach and back, The Evalution of Minds*. W.W. Norton and Company Ltd., 2017.

[5] An Equivalence, of Fully, Connected Layer, and Convolutional Layer. Technical report. 2017.

[6] Cheng-Yang Fu, Wei Liu, Ananth Ranga, Ambrish Tyagi, and Alexander C. Berg. Dssd : Deconvolutional single shot detector.

[7] Edelman Gerald M. *Neural Darwinism, The Theory of Neuronal Group Selection*. Oxford University Press, 1989.

[8] Ross Girshick. Fast r-cnn. 2015.

[9] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation tech report (v5). 2013.

[10] M. González-Audícana, X. Otazu, O. Fors, and A. Seco. Comparison between mallat's and the à trous discrete wavelet transform based algorithms for the fusion of multispectral and panchromatic images. *International Journal of Remote Sensing*, 26(3):595–614, feb 2005.

[11] Kaiming He, Georgia Gkioxari, Piotr DollaÂŕr, and Ross Girshick. Mask r-cnn. 2017.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition.

[13] G. E. Hinton and A. Krizhevsky & S. D. Wang. Transforming auto-encoders. 2017.

[14] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. 2017.

[15] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size. *arXiv:1602.07360*, 2016.

[16] Ming Liang and Xiaolin Hu. Recurrent convolutional neural network for object recognition. State Key Laboratory of Intelligent and Technology and Systems.

[17] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. 2013.

[18] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. 2015.

[19] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. 2012.

[20] Dhruy Parthasarathy. A brief history of cnns in image segmentation.

[21] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection.

[22] Joseph Redmon, Ali Farhadi, University of Washington, and Allen Institute for AI. Yolo9000: Better, faster, stronger.

[23] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 1 faster r-cnn: Towards real-time object detection with region proposal networks. 2016.

[24] Matus Telgarsky. Benefits of depth in neural networks. In *JMLR: Workshop and Conference Proceedings vol 49:1âĂŞ23, 2016*. MTELGARS@CS.UCSD.EDU, 2016.

[25] Andreas Veit, Michael Wilber, and Serge Belongie. Residual networks behave like ensembles of relatively shallow networks. Department of Computer and Science and Cornell Tech, 2016.

[26] Jason Yosinski, YOSINSKI@CS.CORNELL.EDU, and Cornell University. Understanding neural networks through deep visualization. 2015.

[27] Matthew D. Zeiler and zeiler@cs.nyu.edu. Visualizing and understanding convolutional networks. 2013.

[28] Yundong Zhang, Haomin Peng, and Pan Hu. Cs341 final report: Towards real-time detection and camera triggering. Technical report.

[29] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. 2012.

VI. Appendix A - the Mini-Turtle Network

Our goal is to create a small Neural Network that is trained with pictures from 2 or more robot teams and that is capable of recognizing robots in the omni directional Turtle images. We train a small neural network that is able to recognize robots of the same size but in different orientations and will attempt to use this network to classify and localize robots that appear in the larger Turtle image.

Because we know the exact mapping from any position in the image to its size and orientation, we could create a number of layers, each with different downsampling rates to use the small network to classify those places where a robot appears to be. To detect robots, we will be using a feature map, trained with the original robot images and create a hotmap of all places where a robot might be present. These locations will be the Regions of Interest where the classification will take place.

After the study of existing approaches, their ideas and backgrounds, we are now in a position to define the design considerations for our own network architecture. At this time we do not yet know if our approach is feasible, so this first draft only lists the things to consider and then we will start a series of experiments in which different architectures will be tested.

The most important factors will be the speed of recognition and the accuracy of the location where the robots appear in the image. When we talk about robots, we also include other objects like the ball and people that might be on or around the field. Robots must be classified as belonging to a team and should further be identified by their orientation and possibly by their number plate. We shall see how far we will get with this idea.

So the most important design criteria can be summed up as:

1. The Input.
2. Input Image Properties.
3. Selecting Regions.
4. Classifying Objects.
5. Defining Robot Locations.
6. Other Objects.
7. Interface with the Learning environment.

i. The Input

The input for the recognition process is the Omnidirectional Camera image. We will take successive samples of the input, one image at a time, immediately after processing the previous image. As soon as we know the exact processing time of the Neural Network, we can set a steady stream, where we can use double buffering to get the next image ahead of time. Images may have to be pre-processed, first to remove the borders and just leave a circular image to prevent the surroundings of the field to influence the recognition. Secondly we may need to apply Local Contrast Normalization. The resulting image will be an RGB image with enough resolution to perform the recognition task, where the size of a robot at a distance to the border of the field will be approximately 64x64 pixels.

ii. Input Image Properties

Because the image is formed by a conical mirror, all robots will be rotated between 0 and 360 degrees. To prevent having to learn all possible rotations for a robot, we propose to split the image into 4 segments, where each part will be rotated 90 degrees, leaving us with images or robots rotated at $\pm 45^{\circ}$. That way we only need to train rotations of 45 degrees. The size of robots vary between 64x64 pixels and 128x128 pixels. So we need some ranges of pixel sizes between these limits as image size pyramids or a number of convolutional layers with varying sizes. In order to find out how size-sensitive the trained neural network is to sizes, we will start with 6 layers, varying from 64, 80, 96, 112 to 128 and see how far we get with that. Input will always be 64x64 and the image is scaled down from the given sizes at 16 bit intervals vertically and with a stride of 1 horizontally. So we will have an average error of 8 bits, which equals to about 16 cm vertically.

iii. Selecting Regions

There are two methods we can use to select regions. If we use temporal input, then we use the differences with the previous image. We need to take into account that this will create false positives for objects on the field that do not move like lines and static robots. Also we cannot distinguish between a robots own movement and that of other robots. However this approach may save a lot of

time, since we do not have to scan the image for possible objects.

The second approach is to use a moving window approach, where we use a convolution over the image and use the small network to classify each possible object. We can do this by using all activation maps and find the hotspot. Using that spot we can classify part of the image for the top X hotspots of all activation maps. We could set a threshold for the activation level. If all activation maps are normalized, this might be a workable solution.

iv. Classifying Objects

We first train a small network based on 64x64 input images. This network now has learned the feature maps for the robots and other objects on the field. We then take the entire image and generate activation maps by replicating the learned features over the entire image, which is 582x582 pixels. Using a 9x9 matrix over the entire image we get a resolution of 576x576. We create an activation map of 9x9 input images and this get the heatmap for the entire image. From this heatmap we then select all top 5 activation spots and classify all these spots, using the small network. Using a threshold we now get all possible locations where a robot might be found. At various levels we need to down-sample the original input, but we might also train a 128x128 network and up-sample the low resolution regions. It all depends on how fast this network proves to be.

v. Identifying Objects

To identify individual robots on the soccer field, we use the small Convolutional Neural Network (CNN) that was described in the previous section. This network has learned a number of filters, that extract information from the image and classifies an object in its final layer. We intend use these filters and apply them to the entire omni-directional Turtle image, so we can identify multiple robots. The problem is the that the final Fully Connected Layers are not scalable and they only classify a single robot.

vi. Localizing Objects

There are many different ways of finding an object's position in an image. One of the most used subjects is number plate recognition in cars. Here, in general a program first looks for a rectangle in an image and then tries to locate the numbers or letters in that rectangle. But how do you find a rectangle in an image? The simplest way is to scan the picture left to right and top to bottom. This however is a very time consuming way. The challenge is to do this in such a way that we take as little time as possible. There are several ways to achieve this:

1. Only look at a limited area of the picture by eliminating the area where it will generally not be found, like in the sky. In our case that would be only inside the field, so eliminate the corners, since the camera image is always circular because of the mirror.
2. Start looking in the most likely place or the place of most interest. This could be the center of the image or in our case in the vicinity of the place where the robot was last time.
3. Do the search in parallel, using multiple threads or with the help of a GPU. Basically this is what a Convolutional Neural Network does, scanning the image and doing that in parallel.

The first two approaches are too slow. The third one depends on the speed and capacity of the GPU. So in most cases some combination of all three is the right approach. Several Neural Network architectures have been proposed to solve this problem and we will describe each of them in a little bit more detail.

However, our particular problem is slightly different from the problem that has been tackled with these approaches. First of all, they all concentrate on multiple classes of objects, which generally are part of a standard, upright and undistorted picture. Our problem on the other hand deals with a limited number of objects, mostly of the same shape and color combination. As you can see in the figure 3, their size and orientation depends mostly on their position, so that provides information that is not available in the other kind of object localization. We know exactly how a robot will be resized and rotated at every location in the picture. So we intend to use this extra information to simplify and speed up the localization task.

That is the subject of the next section of this paper

vii. Important factors for localization

There are a number of properties of the camera image that we need to take into account, when designing a Neural Network architecture, and we will list all of them with their prospective considerations.

In most of the existing systems the approach is taken to design a general-purpose solution that can be used for many applications. Because designing and training Neural Networks is hard, the idea of Transfer Learning emerged in which a general purpose set of images is used to make a feature extractor that may be used in other domains as is done with ImageNet, Inception, VGG, ResNet or COCO.

Our problem is simpler and much more specific. For instance we always handle the same kinds of objects; Robots, a Soccer Field, Ball and People. Their properties are well know, like their sizes and aspect ratios. Also, because of the properties of the Omni-directional Camera mirror, we know the relationship between the apparent size of a robot and its distance and rotation in the image.

Many of the properties that are otherwise learned by the network are known in advance in our case. This should make it possible to create a dedicated network architecture that is both compact and fast enough to allow quick decisions during a competition and that is also a lot simpler in structure than existing approaches. If this idea proves feasible it might also offer insights for other, dedicated applications, but our main goal is to design a dedicated Neural Network that is supporting the decision making process of the robots and that may also be used to learn optimal behaviors, preferably in real time.

So in summary, our goal is to design a simple and fast recognition procedure that forms the basis for a real-time learning mechanism for soccer playing robots. The following characteristics are relevant to design such a system:

1. The center of an image is always the robot itself and must be ignored.
2. The rotation of the robot is always from the center through its heart. This could be calculated and the apparent size of the robot in the picture also determines its distance. So we can train a network to only recognize a limited number of rotations. So we could do the following:

   (a) For every location calculate the distance and rotation. Then normalize that part of the image so we always have images of the same size and rotation variation. That simplifies processing but is that possible with existing Neural Networks?

   (b) We could split the entire picture into four quadrants and rotate each part, so all four of them always have the same orientation. That eliminates the rotation element and leaves only the scaling factor.

   (c) Scale parts of these images based on their apparent distance, so they all get the same input size.
3. Let the Neural Network do a lot of this preprocessing, so it can be done in parallel by a GPU or a group of processors.

viii. The Learning Environment

The Learning Environment is of such a large magnitude that we will dedicate a separate paper to this. Here we will describe the general ideas and considerations for design of this part.

In early AI systems in the good old-fashioned days of AI (GOFAI), all systems were based on heuristics, operating on the Symbolic Level and based on explicit knowledge. With the great successes of Neural Network technology most processing is now done with implicit knowledge which has the disadvantage that it is hard to find out what has been learned and where the knowledge is located.

This creates the problem that we do not know what these systems might do, creating some fear for runaway systems that may become uncontrollable. Dennett [4] suggests that we should not employ autonomous robots unless their creators are held responsible for their actions. They can only do so if they fully understand what these self-learning systems are doing. Of a much lesser consequence is that we have no idea how to create systems that have to learn many different actions, based on information from their environment.

End-to-end learning makes it unnecessary or even impossible to know what a system is learning and how it is able to perform its tasks. If we consider playing soccer as an example, we can imagine that during a game there are many game situations and that a player learns the best way to behave for every game situation. Every time that the robot encounters a new game situation it should learn how to handle. It also means that in every case it must be able to determine if this situation is one that it has encountered before or that it is very similar to such a situation.

Early AI systems employed a variety of situation-action pairs to determine what the most appropriate

action would be. Systems that implemented this type of approach are rule-based systems (RBS) with heuristic rules, or augmented-transition-networks (ATN), or hidden markov models (HMM). What they all share is some kind of pattern matching that determines the state the system is in, and the action to take. Many different names for these were employed like the IF clause or the left-hand-side or the situation or state and the THEN clause or right-hand-side or the action part. In many cases some kind of probability as assigned to one or both sides by things like Certainty Factors, Bayesian probability or Fuzzy Logic.

In current systems game situations are mostly described as state-transfer-diagrams, where in every state the entry- and escape conditions are described. In the early 90s Rodney Brooks designed the Subsumption Architecture that allowed a gradual buildup of basic behaviors that could be extended by adding higher conceptual levels, without disturbing earlier, already proven layers. It had the advantage that if something went wrong in higher levels, the older, proven layers would take over to create a robust system.

When employing a Neural Network, patterns can be learned that classify the game state and could directly be coupled to actions. However this approach loses the insight and structure that would otherwise be gained by manually building such a system. What we are proposing is to create a hybrid of the two approaches, marrying the symbolic level of situation-action pairs with Neural Networks. We let the Neural Network learn patterns and match these with the hand-crafted system. All behaviors that have been programmed into the existing system will be used when appropriate. They are fast and efficient and give a good insight into what is happening. When a pattern is detected that is not present in the current structure, the system will generate the appropriate classification parameters and learns the behavior that is associated with this situation. This combination is then added to the symbolic structure. This way we have control over every game situation and also have insight into the gaps that exist in the current model. A self learning system could explore this state space and learn to fill in the missing parts or could detect unknown new situations during a competition. It could also exercise in a simulation environment and find out new situations without actually playing a competition.

With this approach we can blend the symbolic with the deep learning level and combine hand-crafted solutions with learned behaviors. In addition it allows both real-time and off-line learning of new behaviors, while at the same time keep close control over what is being learned.

To develop such an environment we need to do the following things:

1. Create a small network to classify robots and the ball
2. Create a network to find the position of all robots in a camera image
3. Create a world model based on the robot field of vision
4. Create a classifier for the game situation for every robot
5. Create a state transition network (STN) with state classifiers
6. Create a network that learns optimal behaviors for a certain state
7. Create a simulation environment to exercise game situations
8. Make an interface between the robot and the simulator
9. Let the robot report unknown game situations
10. Integrate the Subsumption Architecture with the STN

VII. Appendix B - Different Existing Neural Network Approaches

In order to understand the network that we will be using to localize the robots in a Turtle image, it is important to first study the various approaches that have been developed so far and see how most of the technical challenges have been solved. This may lead to a deeper understanding of the properties of the various Neural Network architectures. Because this study is mostly included for completeness and to explain how we can design a new network architecture, this section may be safely skipped if you are mostly interested in the way that our Neural Network operates.

Because there are so many different Neural Network types, we include a schematic overview of existing architectures, found at the site of the Asimov Institute, called the Neural Network Zoo.

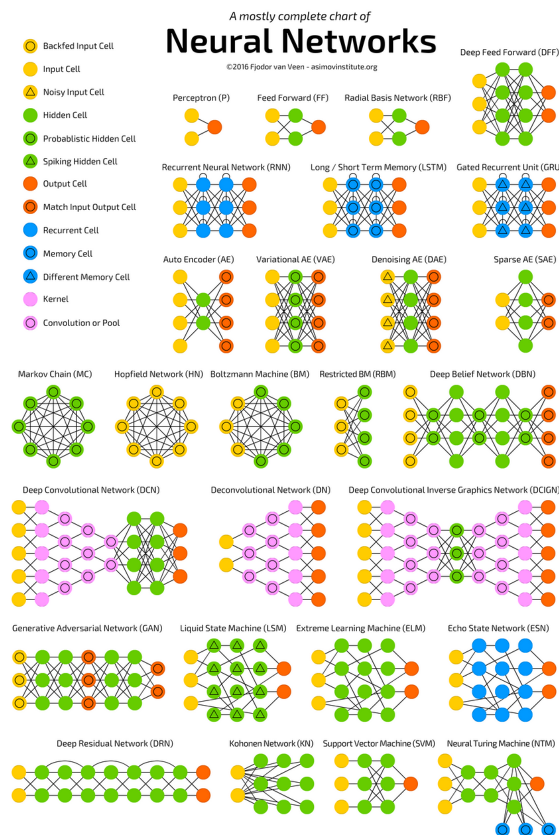

Figure 17: *The Neural Network Zoo*

We assume that Neural Network architectures for whole image recognition, like Convolutional Neural Networks (CNN) are already well understood and are mainly concentrating on localizing multiple objects in entire images. So apart from the elementary moving window approach, a number of CNN solutions have been proposed. A general introduction to some of these networks is given in [20]. We will give a brief overview of the most important approaches and comment on each of them, leading to the selection of a target solution on which we will be concentrating:

i. Recurrent CNN [16]

Recurrent Neural Networks are generally used to learn about time series, like in speech recognition, music or in action sequences. They certainly have a place in learning robot behaviors, but the application of them to object recognition is something that begs for an explanation. According to the paper, the neocortex where recognition happens has many more recurrent connections than feed-forward connections. It is thought that this information adds context to the recognition ability of the brain. This type of network learns to better classify parts of an image, using input from its neighboring pixels and adds context in this way. With fewer parameters these networks achieve better performance, which could help speed up our target networks. The paper also stresses the importance of RELU to suppress the vanishing gradients, while dropout is used to prevent the network from overfitting. It also mentions the Network In Network (NIN) [17] approach, which is something our proposed network will most probably employ as well. Instead of a Convolutional Layer, this type of network uses Recurrent Convolutional Layers (RCL).

This approach is consistent with the current trend to increase the depth of networks, without increasing the number of parameters, which we also see in other network approaches. The recurrent aspect has the advantage that it also learns to take information around the object into consideration, thereby adding context. It is also interesting that all layers have the same number of feature maps and filter sizes, which makes the architecture simpler to design. We see many of these ideas again in the other approaches that follow. The paper also indicates that dropout played a significant role in achieving the results of this network against other architectures. Another note from this article is that all tests were performed on MNIST and CIFAR while preprocess-

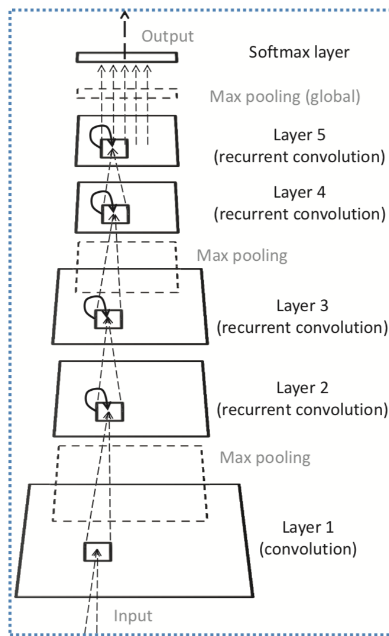

Figure 18: *Structure of a Recurrent Convolutional network*

ing all input images by subtracting the mean pixel value from all pixels, called Local Contrast Normalization. We need to try if that also influences the recognition in our models.

ii. Region Based CNN (R-CNN) [9]

Convolutional Neural Networks are mostly used for classification. Sliding window detectors to test if there is an object at a specific location was the most used way of finding the position of an object. The R-CNN approach added so-called Region Proposals and is embedded into a CNN network, so it is able to both classify and locate multiple objects in an image. Making region proposals reduces the number of places where tests for an object are made compared with a sliding window approach. These tests are performed using a Support Vector Machine (SVM), a more conventional classification method, not based on Neural Networks. The main disadvantage of this approach is its speed. Classifying a single image takes about one minute. The paper also explains that the contribution of the Fully Connected Layers (FCL) is considerably lower than expected and that Fully Convolutional Networks are equally able to classify objects. This method achieved substantial performance gains over earlier conventional classification methods, like HOG pr Haar features. It also combines the power of Convolutional Neural Networks with the classical Computer Vision tools. However it is slow and not fit for real-time image processing.

iii. Fast R-CNN [8]

With this approach, also a CNN is trained for feature extraction. But other than R-CNN two independent classification streams replace the last Fully Connected Layer, one for the selection of Region Of Interest (ROI) feature vectors and a second one for bounding box regression. Both of them are implemented with Stochastic Gradient Descent (SGD) and are therefore end-to-end trainable by the network in a single pass. This results in a considerable saving in time and additionally a better recognition performance. Where the R-CNN test time was about one minute per image, this approach takes only 0.2 second. When using an experimental truncated Singular Value Decomposition (SVD) this time is reduced to 0.08 seconds. Training time is also much lower, with 2 hours compared to 28 hours with R-CNN. So this approach is much more suitable for real-time applications. Please note that these reported times do exclude the time for generating the region proposals, which are still considerable and not done as part of the Neural Network. Another approach developed about the same time was SPPnet and achieves comparable results. However the literature continued concentrating on the Fast and later Faster R-CNN approaches.

iv. Faster R-CNN [23]

With this improvement over Fast R-CNN the Region Proposals that were previously a separate step are now generated from the same CNN and therefore adds no extra cost. It achieves image detection with a frame rate of 5 fps, using a fast GPU.

This type of network can be trained end-to-end and also detects multiple objects in near real-time. So this is an important improvement over Fast R-CNN. By re-using the CNN to generate region proposals, a new type of network is introduced, dubbed a Region Proposal Network (RPN).

It shares the same Convolutional layers that are used by the CNN already at virtually no extra cost. The RPN is a Fully Convolutional Network (FCN). So this type of network can be seen as a Fast R-CNN combined with a RPN. We will describe a bit more details about this RPN

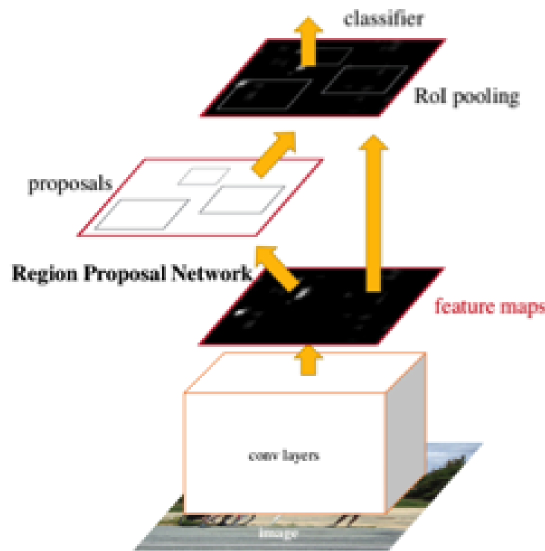

Figure 19: *Structure of Faster R-CNN network*

and its architecture as it directly applies to our goal of creating a fast and small network for detection of MSL robots.

A small network detects features in the feature map and outputs two fully connected layers from the last Convolutional Feature Map. One layer generates a box regression layer (REG), the other generates a box classification layer (CLS). The mini-network is in our case the single robot classification network, trained with photographs of the actual robots, distorted, rotated and warped so they resemble robots in the Turtle omni-directional camera image.

The REG and CLS layers are implemented as 1x1 convolutional layers. In Faster R-CNN these points are called Anchors and are proposals of different scales and aspect ratios. In our case however, the location in the image actually determines the size as a result of the distortion properties of the omni-directional conical mirror. The aspect ratios are known and do not change. So our region proposals will be very sparse and much more reliable. What is important in this approach is to design a loss function that properly defines the reliability of the region proposals, combined with the classifications, so the network learns to optimize both at the same time. This paper gives useful examples on how to do that.

v. Mask R-CNN [11]

This approach is an extension to Faster R-CNN and is designed for image segmentation. Classification is now done on a pixel-by-pixel basis so that full object maps can be generated. The Faster R-CNN network is extended with several extra layers to achieve this and create a small overhead compared to the original implementation. The approach does not seem relevant for our purpose at this time.

vi. You Only Look Once [21]

The networks we investigated so far all attempt to decrease the time spent in finding out where objects are located in an image. The main strategy is to find Regions Of Interest (ROI) and several methods are used for that. They all share the idea of region proposals, where the network checks if there actually is an object that can be classified. In most cases this means using feature detectors like Haar, HOG or SIFT.

The Yolo approach takes a different view by creating a Fully Convolutional Network in which a number of layers is looking at parts of an image in smaller and smaller sizes. Each of these layers is evaluated in parallel, so it completely eliminated earlier region proposals, but looks at them all at the same time. Hence the name You Only Look Once (YOLO).

This increases the size of the network and also relies on a fixed size for the various layers. It is considerably faster than previous methods but depending on the depth of the network it may be less accurate in determining the location of an object. Please note that this speed is mainly achieved by the use of a GPU, because many calculations are required.

For classification it uses a small network like a NIN and for location it trains input bounding boxes as a regression task. Generally Neural Networks are used either for Classification or Regression. Classification returns the class of an object, regression returns a value.

Yolo works by first training a network with images for which the ground truth defines both the bounding box and the class it belongs to. Once the network is trained, input images are split into a grid of image parts. At each part of the grid, the network checks if there is a recognizable image. The output is a class probability and a regression of the bounding box. There may be a partial image in a grid cell and so several grid cells together may form an entire object. This evaluation is

done for the entire image in a single CNN, consisting of several layers, each with different grid sizes.

Figure 20: *Working of a YOLO network*

So the entire network consists of a number of Convolutional Layers for feature extraction and a number of Convolutional Layers with different grid sizes, that each classify all elements of grids of smaller sizes. Together these layers provide the bounding boxes and classifications for objects found in the entire image.

Figure 21: *Structure of a YOLO network*

Three other systems exist that contain elements that YOLO is based on, called Deep MultiBox, OverFeat and OverGrasp. The regression system of YOLO is based on OverGrasp.

vii. Yolo9000 [22]

YOLO-V2 is an improved version of YOLO. Although YOLO is fast, it made many errors on localizing smaller objects. The improved version of YOLO deals with these problems by removing some of the pooling layers, thereby increasing the image resolution of the classifiers. Also the fully connected layers are removed and replaced by anchor boxes. By making sure that all layers have a predefined size, that has a clear center position, the final layer can be made into a 13x13 feature map. Instead of predicting class and bounding boxes for every grid cell, it now generates a bounding box and classification for all anchor boxes. This approach was later replaced by a method to predict five bounding boxes for every grid cell, to increase accuracy.

Another improvement was to introduce a hierarchical classification system, called WordTree. This makes classification much simpler. In our current system we already use a hierarchical classification scheme, using Object Type, Team, Orientation and ID as subcategories.

Figure 22: *Stucture of WordTree*

Because Yolo is based on Darknet (a special implementation of a Neural Network), all pre-trained networks need to be translated to another format to be usable in TensorFlow or Caffe. There are many example implementations available on GitHub. One that is particularly interesting is Tiny-Yolo that runs on smaller machines, like iPhone, Android, Raspberry-pi or the Movidius Neural Compute Stick (NCS), which we will describe later in this paper.

viii. Single Shot Detector SSD [18]

The Single Shot Detector is a Fully Convolutional Network (FCN) that generates bounding boxes over different scales and aspect ratios. It does not generate region proposals but finds the best match for every feature map. So far near real-time speeds have only been achieved at the cost of decreased accuracy. The system receives

ground truth bounding boxes and trains a network with these. It then defines a number of convolutional feature map layers with decreasing grid sizes. This is different from Overfeat and YOLO which use single scale feature maps.

In each of the layers it defines a number of default boundary boxes. During training then the offsets of the ground truth boxes from the default boxes are calculated and learned as part of the network. The Network calculates the loss between the actual bounding boxes and the default boxes and learns the actual sizes in this way. In the same way it also learns the aspect ratio of each box, so both the offset and aspect ratios are learned.

Figure 23: *Single Shot Detector architecture*

Just as with other networks that use region proposals, the ground truth box sizes need to be assigned to specific outputs. Therefore the selection of the default boxes becomes critical and the selection of scales and aspect ratios are an important part of designing the network. Because we are using objects with well-known properties, this should bot present any real problem. This approach is a high contender for our intended network design.

ix. Deconvolutional SSD [6]

As we have seen with the Fully Convolutional Network (FCN), upsampling of pixels is an important part of some network architectures. This is done mostly by deconvolution. Actually the term deconvolution is a misnomer, the actual process is transposition and is therefore called Transpose in TensorFlow. In a Fully Convolutional Neural Network, deconvolution is used to get back the original input pixels and this improves the accuracy of the last layer. Deconvolutional SSD is therefore a combination of the FCN and SSD approach.

In Figure 24 we see the so-called hourglass model in which successively smaller layers create a bottleneck where in the narrowest part the most compressed information about the image is concentrated. From this a number of layers with increasing size are used to find

Figure 24: *Deconvolutional Single Shot Detector (DSSD) architecture*

the location of objects in the proper scale. This upscaling from lower layers is done with deconvolution. This type of network architecture is also referred to as the Encoder and Decoder stage of the network.

Another thing that becomes clear in all of these studies is that they depend on transfer learning, where large generic models are used as feature extractors. Also many experiments are done to find out what the most common aspect ratios and object sizes are. This information is then used to optimize the networks. In our case we always know in advance that the aspect ratios of all objects are the same with exception of the ball. Also the size of each object varies with its position in the image with a well-known factor, so we are in a much better position to design these parameters than networks which rely on generic classes and unknown objects. We will talk more about that in the section about the architecture of our own model.

x. Deep Residual Networks [12]

Deep Residual Networks are one of the most important innovations in Neural Network research of recent years. There are two important aspects to this approach that deal with improving the internal validity of the weight matrices of a network layer. It is desirable for a network layer to represent a function that accurately reproduces the properties of the input. This is also know as the identity mapping of a layer. The difference between the input and the output of the function is the residual and the purpose of the approach is to let each layer learn an identity mapping as well as its residual.

This is achieved by constructing a layer out of several smaller layers (mostly 2 or 3) that are connected in such a way that multiple paths between the input and the output exist. This gives such a network a lot of flexibility in learning and makes learning not only more reliable but also faster, in spite of its added depth. In Figure 25 part B these different paths are clearly visible.

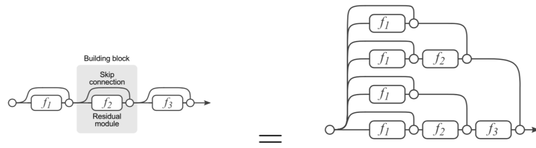

Figure 25: *Residual Network Ensemble Block Layout*

Deep Residual Networks are designed to allow the construction of networks that are considerably deeper than previous networks, but are easier to train. As an example the CIFAR 10 dataset is given, which usually has 3-5 convolutional layers. With a Residual Network it may have more than 100 layers. There has always been the suspicion that deeper networks lead to better recognition [24], but attempts to create deeper networks have always suffered from the vanishing gradients problem, where the weights in deeper layer became so small that they no longer contributed to the recognition task.

The Residual Networks are based on building blocks, each with two or more layers and a shortcut from the input to the output layer as can be seen in Figure 26. In the example with three layers, a bottleneck design is used, where the input is first scaled down and then expanded again in 1x1, 3x3 and 1x1 convolutions. In spite of the fact that there are considerably more layers, the total number of parameters and thus training and recall time remains the same, while achieving a higher accuracy.

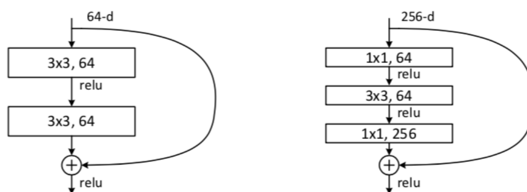

Figure 26: *Residual Net Building Blocks*

One thing that also became clear from this paper is that it is customary to process the input images by subtracting the mean pixel value for every pixel. So that is an important issue to also try in our target networks.

Neural Networks are very much inspired by their biological examples, as described earlier in R-CNN [9]. Here we seen another example, where brain research has inspired new developments with the introduction of Residual Network Ensembles, based on the seminal work of Gerald Edelman in Neural Darwinism [7]. Residual Network Ensembles are described in [25]. It explains clearly why deeper Residual Networks are just as efficient as their shallower counterparts.

In this paper some interesting experiments are done, where is shown that dropping layers in a Residual Network has a very low impact on performance, while a CNN is severely impacted by such modifications. So there is clearly something important going on in Residual Neural Networks, that we have to take into account.

It should be noted that GoogLeNet uses a further development of this idea by defining ensembles of different micro-architectures, called inception modules. The ensembles and inception modules are examples of these micro-architectures. This forms the basis of the very popular Inception system that is used for many transfer learning projects.

Further developments also take place in macro-architectures where more pathways between layers are being created, like is done in Highway Networks. Here more skips between layers are used to allow the network to learn more diverse functions. In all these newer approaches more depth is introduced to achieve higher accuracy while maintaining efficiency. These approaches are referred to as bypass connections.

So we see a further trend in the following areas:

1. Micro Architectures with bottleneck designs and multiple paths.
2. Macro Architecture with more levels and skip layers.
3. Bypass Connections to connect higher layers with deeper layers.

xi. Region Based FCN (R-FCN) [3]

Region Based Neural Networks like Fast CNN or Faster CNN rely on costly region proposals. The Region-FCN approach uses a Fully Convolutional Network to classify and localize object in entire images. There is always the contradictory goal of translation-invariance in recognition and translation-variance in object detection. The method uses position sensitive score maps to overcome this problem. It is based on Residual Networks and adds some new features that improve on the Residual

Network approach. Instead of adding region specific layers, this method adds a region-specific pooling layer to every convolutional layer. The method still employs a Region Proposal Network (RPN) but this is entirely convolutional and shares the features with the entire network.

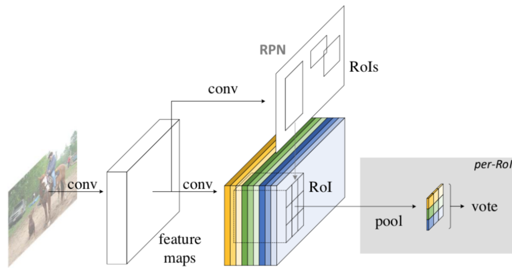

Figure 27: *R-FCN Network architecture with RPN*

This method is based on ResNet and removes the last layers and replaces them with the ROI pooling layer.

xii. Capsule Nets [13]

One of the major shortcomings of Object Recognition with Neural Networks is their inability to maintain spatial integrity in the final layers of a network. In every layer a part of this spatial information is lost. For instance to recognize a face, the network looks for eyes, a nose and a mouth. But if the position of any one of these parts is in the wrong place, the network does not notice this and still recognizes the object. This problem is being addressed in Hinton's recent work with Capsules.

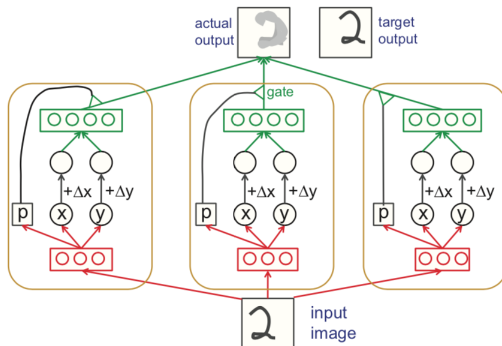

Figure 28: *Capsules in Convolutional Networks*

Instead of Pooling lLayers or a Fully Connected layer in the later stages of a network, Capsules are used, that learn the orientation of features of a particular layer, using an auto-encoder. The network will not only learn to recognize the features in a Capsule, but also its spacial dependency, which makes it much simpler to recognize an object by its parts. Features that are recognized as a whole are then propagated to deeper layers, resulting in full object recognition. This also helps in determining the orientation of the object, because the network keeps information about the spatial dependency of all recognized Capsules.

Capsules seem a promising way to allow recognition of rotated and scaled objects, while maintaining localization information like we need in our project. So we will test this approach as part of the network we are developing.

xiii. SqueezeNet [15]

All previous methods concentrated on accuracy of recognition and speed of detection but always based on powerful computers with multiple GPUs. Here speeds of more than 40 fps have been achieved. There are however more and more applications where there are no fast systems of GPUs are available, like smart-phones, tablets, drones and robots. In our case the robots have a reasonable processor and a small GPU, but because they are battery operated their power consumption must be as low as possible.

Therefore several new approaches have been developed in which the recognition task is to be done on small, mobile and power restricted devices. The first one we will investigate is SqueezeNet.

The main trick to increase speed and lower power consumption is to decrease the number of parameters and model size, without degrading performance too much. Basically there are two approaches to achieve this goal. First of all there is network pruning, in which all values below a certain value are set to zero and then retrain the network with all zero values removed, The second method is quantization where all floating point values are reduced to small integers, thus saving space and computation time.

SqueezeNet uses a micro-architecture with modules called Fires, based on 1x1 and 3x3 convolutions. Downsampling is delayed to deeper layers to keep the total number of parameters low.

In addition it uses Deep Compression, which consists of both network pruning and quantization to 8 bit in-

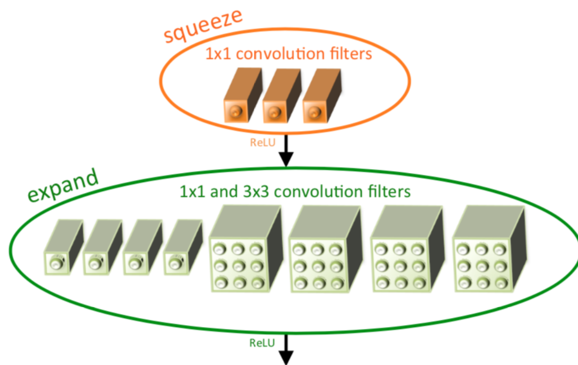

Figure 29: *Squeeze and Expand strategy of SqueezeNet*

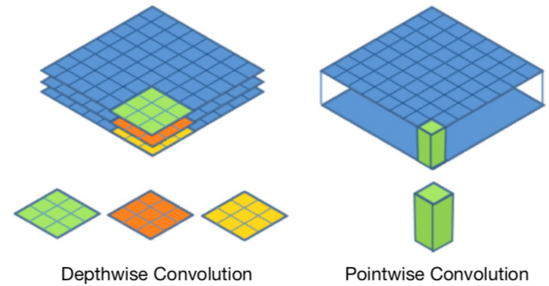

Figure 30: *Depthwise Separable Convolution structure*

tegers. A special version of 8 bit quantization, called Ristretto is also used, which we will not describe here in any more detail. Ristretto achieves a less than 1% drop in accuracy with 8 bits compared to 32 bit values.

On the macro level, SqueezeNet introduces several bypass strategies between Fire levels. These bypasses make sure that a residual function is learned between the layers that are connected by the bypass. Simple bypasses require that the number of input and output channels are the same and therefore do not create any additional parameters.

xiv. MobileNets [14]

MobileNets is another approach to create condensed networks that can run on mobile devices. It uses a technique called depthwise separable convolutions [1] that we saw before in the ResNet summary. [25] This paper also refers to some other approaches to make networks smaller and faster like Vector Quantization, Huffman coding, Structured Transform Networks, Deep Fried Networks, Network Distillation and Low Bit Networks. We will not go into details about these, but instead concentrate on the ideas behind MobileNets.

Depthwise separable convolutional layers consist of a dense 1x1 pointwise convolution and a 3x3 depthwise convolution. The pointwise convolutions are much more efficient in calculation and therefore save a lot of processing time, since 95% of the time is spent in these layers.

The MobileNets also introduce two hyper parameters that allow an easy selection between speed and accuracy of the network, without having to redesign the architecture. These are called a width multiplier and a resolution multiplier. By varying these two hyperparameters the size/speed and accuracy of the network can be changed.

xv. Realtime Detection on Raspberry Pi [28]

Most neural networks are developed for high-end CPU and GPU systems. This paper describes the development of a network targeted for a Raspberry Pi, as we intend to use for the mini-Turtle robot. The performance of the proposed system is measured in three dimensions, as is the case in most other systems: mean Average Precision (mAP), detection speed in frames per second (fps) and model size (mostly in mB or number of parameters).

This work is based on a combination of SSD [18] and MobileNets [14] and achieves a performance of 20x faster and 15x smaller than Yolo [21].

This network is using the same Depthwise Separable Convolutions (DSC) as MobileNets, but other than all systems before, this system relies on a steady input stream and makes region proposals based on the differences between successive frames. This temporal detection model is well suited to our intended use case. But if no change occurs, the region proposals will not be generated. This saves a lot of time in finding Regions Of Interest (ROI), but introduces another problem in making sure that steady objects that were detected earlier are kept in memory somewhere.

Although this model does not appear to deliver real-world applicability, it generates a number of interesting ideas that we will include in the considerations for our own model.

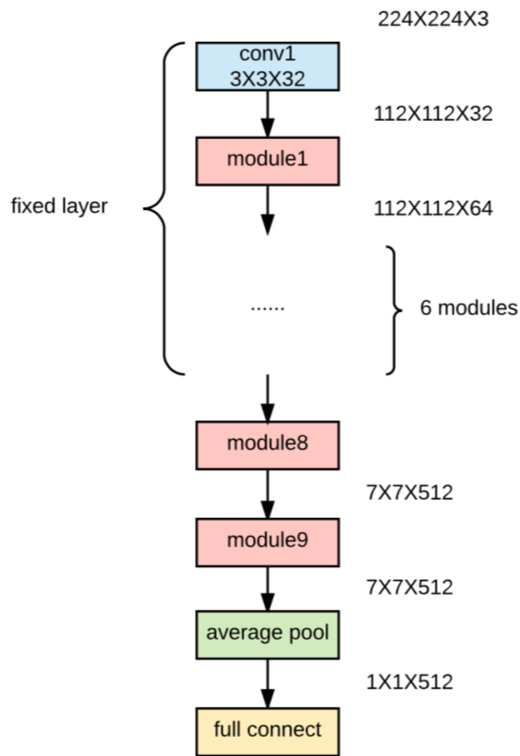

Figure 31: *DSC Architecture*

VIII. Appendix C - Considered Localization Approaches

The approaches listed in this appendix are the ones that inspire our work most. The ideas and sometimes the examples were used in building our own network.

i. Weakly Supervised Learning with FCN [19]

ii. Semantic Segmentation [19]

iii. Learning Deep Features for Discriminative Localization [19]

iv. Understanding Neural Networks Through Visualization [19]

v. Fully Convolutional NN [19]

Most Neural Networks for classification and segmentation consist of a number of Convolutional Layers, followed by one or more Fully Connected Layers (FCL). In this approach there are no Fully Connected Layers, but all layers are of the Convolutional type.

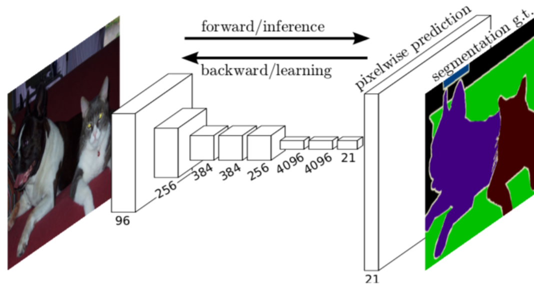

Figure 32: *Fully Convolutional Network (FCN)*

This type of network is most useful for image segmentation, where the features in each layer are used in the deeper layers to make a pixel-by-pixel prediction about the class of every pixel, thus leading to very accurate image segmentation. This could be important for both classification and localization of objects in an image. It does not need region proposals nor dense classification layers.

In most previous approaches multi-scale pyramids are used to handle different sizes of objects. In this FCN approach these pyramids are no longer needed. The model is trained on whole images with whole-image ground truth information. The main difference with earlier networks is that these learn nonlinear *functions* to classify data, while a convnet learns deep nonlinear *filters*. To be able to define object boundaries on the deepest layers, we need to get back to the original pixel level. This is done by a Deconvolutional Network that up-samples the coarse pixels in the last layers back to the input level.

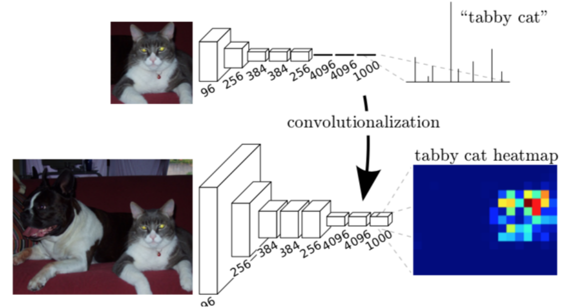

Figure 33: *Comparing FCL to FCN setup*

What we see in Figure 33 is the difference between the two approaches. In the top part the last FCL layer classifies the image as a cat. In the FCN version the output is a heat-map of the cat, thus both classifying and locating the object. The layers in between are all convolutional with in the last part some Deconvolutional layers that are up-sampling the image to gain access to the input layer.

Another advantage is that the FCN approach is generally more than 5 times faster than previous approaches. The up-sampling however takes additional time, which can be gained back by using the so-called à trous wavelet transform. [10]

vi. Network In Network (NIN) [17]

This approach differs very much from other architectures in that is uses a number of stacked 1x1 convolutional layers. This allows the combination of spatial properties of a layer and transferring this information to the next layer. Although it may seem that this contradicts the principles used in earlier approaches it does allow the combination of convolutional features in a better way than just adding more layers to a network.

In section xiii about creating smaller networks like SqueezeNet we will describe these convolutional layers in more detail.

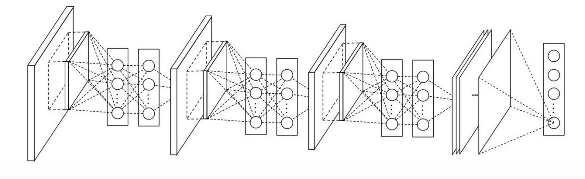

Figure 34: *Architecture of NiN network with 1x1 Conv Layers*

IX. Appendix D - The developed software

In this section we describe the developed software and show some examples of the system in operation. Because this is part of an ongoing project, new versions may become available, so refer to the GitHub repository for the most recent information.

X. Further Considerations

1. ATN's
2. Subsumption Architecture
3. Speed vs Accuracy.
4. Network Architecture.
5. Scaling and Aspect ratios.
6. Depth of the network.
7. Subtracting mean pixel value, Local Contrast Normalization.
8. Temporal Region Proposals
9. Frame subtraction, movement detection
10. Neural Darwinism Gerald Edelman
11. Stochastic Gradient Descent - Random behavior
12. Asymptotically - with reference to an insignificantly large difference
13. a Trous algorithm - Wavelet transform
14. Network In Network
15. Things learned from the literature.
16. Structure of learned network derived from body structure
17. Learning to identify real-world situations
18. Explaining learned behavior
19. Dennets robot requirements
20. Region Proposal Network from Faster-R-CNN
21. Movidius Neural Compute Stick
22. SSD is likely candidate
23. The use of Ensembles
24. Region Based FCN
25. MobileNets
26. Properties of Movidius compiler