Automated Design - Using Machine Learning to Design Structures

Michael Cooper

Supervisor: Dr. Mehdi Kashani

Co-Supervisor: Dr Srinandan Dasmahapatra

27.04.2018

7890 words

This report is submitted in partial fulfilment of the requirements for the Civil Engineering degree program, Faculty of Engineering and the Environment, University of Southampton.

Declaration

I, Michael Cooper declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research. I confirm that: this work was done wholly while in candidature for a degree at this university; where any part of this thesis has previously been submitted for any other qualification at this University or any other institution that this has been clearly stated; where I have consulted the published work of others this is always clearly attributed; where I have quoted from the work of others the source is always given. With the exception of such quotations, this thesis is entirely my own work; I have acknowledged all main sources of help; where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself; none of this work has been published.

Acknowledgements

Dr. Mehdi Kashani - My primary supervisor.

Dr Srinandan Dasmahapatra - My co supervisor.

Sam Cooper - Help in suggesting potential network ideas, and generally helping answer some questions.

David Cooper - Proofreading the document and moral support.

David Clifford - For kindly minding over my computers whilst they trained my networks and I had to go elsewhere.

Table of Contents

Declaration ---------- 3

Acknowledgements ---------- 3

Table of Contents ---------- 4

Glossary of Terms and Abbreviations ---------- 5

1.0 - Abstract ---------- 7

2.0 - Introduction ---------- 7

2.1 - Background ---------- 8

3.0 - Method ---------- 10

3.1 - General Network Features and Procedures ---------- 10

3.2 - Hardware ---------- 12

3.3 - Testing Procedure ---------- 12

3.4 - Network Architectures and Training Procedures ---------- 13

3.4.0 - Single Box Network ---------- 13

3.4.1 - Recurrent Network ---------- 13

3.4.2 - Isolated Recurrent Networks ---------- 14

3.4.3 - YOLO Based Networks ---------- 14

4.0 - Single Box Network Results ---------- 15

4.1 - Recurrent Network Results ---------- 15

4.2 - You Only Look Once Based Network Results ---------- 17

5.0 - Discussion ---------- 21

6.0 - Conclusions ---------- 24

6.1 - Recommendations ---------- 25

7.0 - Bibliography ---------- 28

8.0 - Appendix ---------- 30

Glossary of Terms and Abbreviations

Civil engineering and architectural terms.

- Floorplan - a drawing of the architectural information for a floor, including the location of walls and rooms
- Structure - the collection of load bearing features in a building, such as beams columns and slabs.
- Rectilinear - this refers to structures which have floor plans that are rectilinear polygons, meaning that the sides of the polygons are restricted to the orthogonal grid.

General programming terms.

- GA - genetic algorithm, these typically evolve designs over time by randomly variating a set of designs from a parent, then choosing the best variant to become the new parent. Many variations exist taking a variety of inspirations from nature.
- PSO - particle swarm optimiser, these use a large set of candidate solutions which are moved around the solution space in order to find optimums.
- Algorithmic - used to describe software built on traditional rule and database based programming.
- GPU - Graphics Processing Unit, informally a graphics card.
- CPU - Central Processing Unit, informally the processor.

Machine learning terms.

- Neural Network - a type of program which in essence mimics natural brains and neurons in a mathematical way. Instead of being programmed to solve problems they 'learn' to solve problems
- Network - an umbrella term used in the report to describe all forms of neural network.
- Neuron - a mathematical representation of a biological neuron. These combine the weights of the inputs with an activation to produce an output.
- Weights - the importance of each input to a certain neuron.
- Activation - a mathematical function which combines the inputs to a neuron producing a single output.
- Layer - this is a discrete set of neurons of fixed size which interact with each other in specified ways. Networks have multiple layers.
- CNN - convolution neural network, these expand neural networks to work over higher dimensional input space using filters.
- Filters - these are layers of neurons which look over an entire image using windows without interacting with other neurons in their layer. CNN layers can have multiple filters.

- Windows - these ensure one neuron looks at only a certain part of an image or filter, rather than the whole.
- Strides - this is the distance between each window.
- RCNN - Recurrent convolutional neural network, these feed back into themselves looping round, in very simple terms the equivalent of a loop in algorithmic programming.
- Network architecture - the precise makeup of all the layers in a network. This usage of architecture is distinct from other uses of architecture in the report, which is used in its traditional building sense.
- ReLU - rectified linear unit- a type of activation function used in neurons
- Linear Activation - this is distinct from ReLu in that it allows for negative output values.
- LSTM - long short-term memory layer - this 'remembers' what it has done in previous iterations of a RCNN.
- Optimiser - all networks must learn how their weights, the optimiser is the function which performs the process of learning.
- Training - used as a more formalised term for getting a network to learn.
- Training Data - the data which a network uses to learn.
- Epoch - a run of training performed on the entire set of training data.
- Batch size - the length of the vector that a network is trained on. The mathematical formulation means that networks do not have to train on a single piece of data at any given time, but can train on multiple.
- Dropout - a process in which in any given epoch a specified proportion of neurons and their weights are removed from the network.

1.0 - Abstract

The research used a convolutional neural network (CNN) based on the YOLO network (J. Redmon S. D., 2016), to decompose arbitrary floorplans into rectangles. This was done to move further towards automated structural design; previously this has only been achieved on rectangular or rectilinear structures. This research now provides a method using CNN's which allows for a much wider range of structures to be automatically designed, by simplifying arbitrary floorplans into a set of rectangular segments, which can already have the structure automatically designed. As an offshoot, the research finds a method of solving the general problem of decomposing images in to low order shapes.

Redmon, J. D. (2016). You Only Look Once: Unified, Real-Time Object Detection. *Proceedings of the IEEE conference on computer vision and pattern recognition*, (pp. pp. 779-788).

2.0 - Introduction

Total automated building design is considered to be a process whereby a computer program would be fed a relatively small amount of parameters and subsequently perform both engineering and architectural functions, hence produce the design for a building using nothing but those parameters.

Total automated building design, has the potential to be orders of magnitude faster than designing a building by hand; this leads to a number of potentially beneficial. Firstly the productivity of engineers and architects could be substantially increased. Secondly in the time it would take for a single building to be designed by hand, a computer could iterate through many different versions of that building, exploring more options. This allows for much more optimised design, including many metrics, such as structural efficiency, energy consumption, carbon footprint and building liveability. Finally automation often leads to improved quality and hence potentially reducing the risk of errors occurring.

A large part of the process of total automated building design, is automated structural design. This is the process of submitting a small number of architectural parameters for a building in to a computer program, then using it to automatically design the structure of a building from those parameters. A likely input would be the set of floor plans for that building.

An appropriate approach to automated design is likely to consist of several distinct processes in the following order:

1. Laying out of the primary structural members.
2. Laying out of secondary structural members.
3. Formulation of the loads that the structure must withstand.
4. Designing members and connections such that they can withstand the applied loading.
5. Outputting drawings of the structure and printing out calculations for quality assurance.

Currently automated structural design can be achieved for boxes and rectilinear floorplans, but has not yet been demonstrated for anything more complex. This leads to the aim of the research project; to expand the number of structures that can be automatically designed, by presenting a method using a neural network, which decomposes arbitrary floorplans into rectangles. Through this process of segmenting into rectangles, primary members can then be laid out onto the overall floorplan in an algorithmic way, by dividing up the rectangles along their primary axis, which could be optimised for instance by a genetic algorithm,. Once the primary members have been laid out, Step 1, the remaining steps can be achieved using the current methods set out in the literature for automated design, such as knowledge based expert systems of genetic algorithms. The method allows for multi-storey structures to be designed, so long as the floorplan remains constant with height, as columns could be automatically placed at the intersection of beams, or some other appropriate place.

2.1 - Background

The background presents an overview of the work within the civil engineering community which has been done to move towards automated structural design and why design has not yet progressed beyond rectangular and rectilinear structures. In addition it presents a background into neural networks and these are suitable in solving the problem of decomposing floorplans into rectangles, hence furthering progress in automated design.

Automated structural design has been a long sought after goal, Bathurst L. L. (1965) was one of the first to propose the concept, managing to design simple buildings given loads and geometry. The software selected appropriate members and sections to use, by calculating the maximum force and moment in each member, then selecting the lightest steel sections capable of resisting these actions.

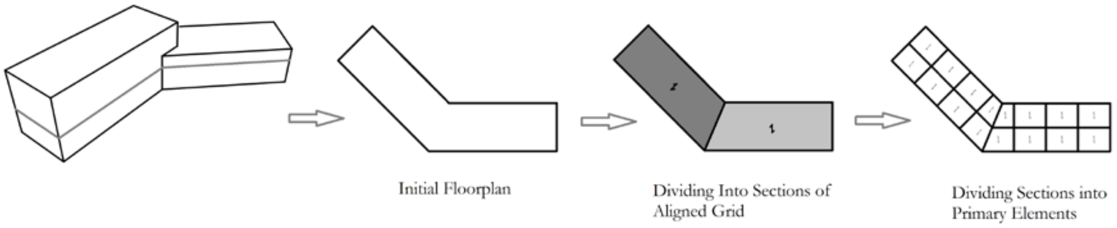
Today many commercial software packages exist which perform the same task as described in the 1965 paper, albeit at a much more complex level, for instance Tekla Structural Designer can perform wholly the design of members given the loading and geometry, using an FEA solver, then optimising the sizes of the members using an iterative algorithm (Trimble, 2018). This is largely the current state of commercial software, despite the potential benefits that more complete automation could lead to.

Following the early attempts of Bathurst, automation focused primarily on knowledge based expert systems, these are algorithmic approaches which use a set of rules and databases that are manually programmed in by human engineers. This fundamentally limits the range of designs that can be produced; all of the presented methods were limited to designing rectangular buildings, or used geometric data that was manually inputted. Some of these also used basic algorithmic optimisation approaches, to attempt to improve design. (C. M. Chan, 1995), (R. Sacks A. W., 1997), (R. Sacks A. W., 2000).

More recently genetic algorithms (GA) and particle swarm optimisers (PSO) have been used to automatically design and then optimise structures. This has allowed for slightly more complex rectilinear structures to be designed, some however, are still only working simply on rectangular structures, (K. W. Park, 1999), (A. Nimtawat, Automated layout design of beam-slab floors using a genetic algorithm, 2009) (A. Nimtawat, A genetic algorithm fo beam-slab layout design of rectilinear floors, 2010), (A. Nimtawat, Simple Particle Swarm Optimization for Solving Beam-Slab Layout Design Problems, 2011), (P. Sharafi M. N., 2012), (P. Sharafi L. H., 2015).

Critically none of the papers describe a method which can design structures which haven't got rectangular or rectilinear floorplans. This is still beneficial as many of the buildings designed today are rectangular or rectilinear, however it is nonetheless a limiting feature of these approaches. The problem is that all of the papers propose automated design systems which must have rules or cost function that are hard coded in by human engineers. To do this for every foreseeable case of arbitrary floorplans with complex shapes and geometries would be a difficult and time consuming task. If a method could be created which segments arbitrary floorplans into rectangles then the problem would be solved, as rectangular structures can already be designed using current method. More specifically once overall rectangular sections have been laid out, primary members can be laid out over the whole floorplan regularly by dividing up along the axis of each using an algorithmic approach, this would then allow the remaining design steps to be automated. Figure 1 shows this graphically. But attempting to do the segmenting using a rule or optimiser based approach would be difficult because of the need to manually program in the rules or loss functions. In this research I sought to use a wholly different approach, using neural networks, to solve this problem.

Diagram of Member Layout Process - Figure 1


Initial Floorplan
Dividing Into Sections of Aligned Grid
Dividing Sections into Primary Elements

Neural networks consist of multiple layers of artificial neurons, which are mathematical representations of biological neurons (W. S. McCulloch, 1943). Artificial neurons, specifically the linear kind I have used, output a single variable which is the weighted sum of all of the inputs (Vinod Nair, 2010). This allows a collection of artificial neurons, a network to perform functions, assuming that their weights are correct. Neural networks are specifically useful because unlike a typical program, where parameters are programmed in, methods exist to get networks to learn their weights, and hence learn how to solve problems (Schmidhuber, 2015) (D. P. Kingma, 2014). This process of learning weights is called training, and requires examples of the correct outputs from specified inputs referred to as training data. Consequently I decided to use a neural networks to approach the problem. Instead of having to manually program in all of the rules to decompose floor plans into rectangles, a neural network would learn these rules, referred to normally as features, by itself which is an automated process.

In addition recent advances in the field have led to networks generalising much better. Generalising means that a network can predict results accurately for inputs it has never seen before, for instance a human only has to be able to see an object once to be able to recognise that object elsewhere, this is exactly what generalisation is. For the problem of decomposing floorplans into rectangles, this is useful, as it means that only a relatively small sample of examples ought to be required for a network that can generalise, to start robustly performing on all possible floorplans (Srivastava, 2014).

Specifically the problem of decomposing floorplans into rectangles can be framed as an image classification problem; attempting to classify boxes within an image of the floorplan. A specific type of neural network has been designed for this task, a convolutional neural network (CNN). Neurons in these networks do not look at an entire image, but rather a small segment of an image, greatly increasing efficiency (Schmidhuber, 2015) (A. Krizhevsky, 2012) (Y. LeCun, 1998). These networks were inspired by the visual cortices of real organisms. A CNN is the precise kind of network that was used within this research to solve the problem of decomposing floorplans, as it is well suited to the task.

This leads to the specific objective of research, within the slightly broader aim as set out in 1.0, which is to use a CNN to decompose images of arbitrary floorplans into a set of appropriate boxes, specifically outputting the size, location and rotation of these boxes.

3.0 - Method

The method outlines the precise details of each network and how training data was created, such that the research may be reproduced. The wider rationale as to why each network was created is presented in the results section. In total eight key networks were created, one version of the single box network, four versions of the recurrent network and three versions of the YOLO based network.

3.1 - General Network Features and Procedures

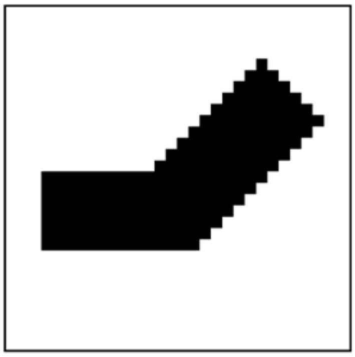
Many of the features and procedures, especially in terms of the automated generation of training data, key parameters and the format of the data remained the same for all networks. These are presented within this section, primarily because many of these methods were successful and did not need to be changed.

All programming was performed using Python 3.6.0 (Python Software Foundation, 2018), as it has a very wide range of prebuilt modules and interfaces and is widely used within the scientific community. The mathematical suite used was Numpy (Travis E, 2006), as this is the standard array and matrix module used within Python and is required to interface with Keras. All of the networks were constructed using the Keras frontend python module (Chollet, 2015), and the Tensorflow backend was used (M. Abadi, 2015). Keras provides a convenient machine learning interface to Tensorflow, which is one of the most widely used backends for machine learning. All graphs were plotted using Matplotlib, the standard graph plotting module for Python (Hunter, 2007).

In total I wrote over 4000 lines of code for the report, including networks, data generation, and testing. An example of some of the code written is attached to the appendix, section 8.0. The example specifically is the code for the final network, BTBnet.

All of the networks took single channel images as inputs. The images consisted of binary pixels, either 0 to represent the absence of floor or 1 for the presence of floor in a hypothetical floorplan, this is the simplest possible representation of a floorplan, which allows image convolution.

Figure 2 - Example Floorplan for a 32x32 Image.



The lines of X = 0, X = 1, Y =0 and Y = 1 represent the boundary box. The bottom left corner of the input image represented the coordinate of (0+Pw/2, 0+Pw/2) and the top right represented the coordinate of (1-Pw/2, 1-Pw/2), with Pw being the pixel width. This normalisation is a practicality which allows for direct comparison between the networks regardless of the input image size. The addition of the pixel width term forces a slight offset, meaning that the pixels are not on the border of the coordinate system. This prevents redundant pixels, which unnecessarily bloat the network size hence decreasing computational efficiency.

The dimension output vectors for all of the networks were of length 5: width, height, X-position of the centroid, Y-position of the centroid and rotation, in that specific order. The dimension vector is all that is required to describe any arbitrary box over the specified image.

The training data was all automatically created by randomly generating boxes using a separate program written in Python. For all random boxes generated, the X- position and Y-position were created with a random uniform distribution between 0 and 1; the width and height were created with a random uniform distribution between 0 and 1.5; the rotations were created with a random uniform distribution between 0 and pi/2. These values allow for all possible boxes that lie within the bounding box, yet prevent rotation duplicates. It also ensures most boxes created lie within the bounding box as boxes were rejected if they lay outside the boundary box. This prevents an unnecessary number of boxes being rejected, hence increase the speed of box generation.

All networks were trained using the Keras inbuilt ADAM optimiser, as it has proven to be efficient, and does not require any hyperparameters (D. P. Kingma, 2014). An efficient optimiser is key as it can speed up training significantly. In addition the lack of required hyperparameters means that lots of training runs need not be done, simply to find the optimal hyperparameters.

All network layers used ReLU (Rectified Linear Unit) activations except when specified. These have been shown to be effective, not suffering from drop off in addition to being computationally efficient (Vinod Nair, 2010). Drop off is a phenomenon in certain activation functions, where values get too large or too small and a neuron ceases to learn. Due to the linear nature of ReLU neurons, they do not cease to learn, this prevents ineffective neurons, increasing network efficiency. In addition due to the fact the activation is a very simple calculation, the computational speed of the network is increased, decreasing training time.

3.2 - Hardware

The following is a summary of the hardware that was used to train the networks, generate the data and perform all other general programming tasks.

Standard desktop pc's provided by the University of Southampton were used, these had 16GB of RAM and quad core i7 3770 CPUs. The networks were trained on the CPU rather than the GPU as is standard recently. This proved adequate for the initial networks, though proved limiting for the later more sophisticated networks. Ideally a large GPU would have been used for training as these are well optimised for training neural networks. This was not however possible.

Ultimately the hardware proved especially limiting when moving onto the full BTBnetb as using 160x160 pixel images, each training epoch took 1hr40mins. As discussed in section 6.1, if better hardware were available, higher resolution images would have been trained on.

3.3 - Testing Procedure

In order to test the networks and compare them against each other, a standard testing procedure was developed and this is set out in this section. Note that the 8 networks all have different regimes which they operate over. This presented some difficulties as the input and outputs are different for each. As far as possible testing was standardised however this was not always possible.

The 8 Networks were tested on a standard data set that was generated. The data set consisted of 4000 box sequences, equally split between sequences of length one, two, three and four boxes. This standardised data set allows for comparison between them. The 8 networks were trained using the relevant portions of the training data.

The actual error between the predicted values and the ground truth was then calculated. This was only done for the vector of dimensions. As all networks were trained on normalised data, this is comparable. The actual error was calculated by subtracting the predicted value from the actual value for each parameter in the dimensional vector across every single applicable data point for each network. Five box plots, Graphs 1 to 5, were created to display the actual error for each network over the values in the vector of dimensions. Note that Graph 1 shows a truncated box, this is on purpose so that the remaining data is well scaled, the truncated box obviously shows a bad result and does not need to be shown fully to understand this.

For each network the standard error of each value in the dimension vector was calculated. The standard deviation of error for the 5 parameters in the dimension vector were then averaged, leading to a single value showing the spread of the data for each network. For each network the modal value for each parameter in the dimension vector was also found and averaged, producing a single mean model value showing the average systematic error in each network. This provides a single unified measure of spread and systematic error with which to compare the networks. Graph 6 is a bar chart displaying this data.

The three YOLO based networks also predict the number of boxes and their location within the grid. The prediction for the networks is composed of two parts, firstly the confidence of an object being present in a given cell, secondly the confidence of this being a box; these two values are then multiplied and if this is above a given threshold value then a box is considered as having been predicted. To test this the predicted boxes were compared against the actual boxes over a range of different threshold values. The number of correct matches was then recorded for each given threshold and plotted in Graph 7. This allows a comparison of the classification accuracy for these three networks

3.4 - Network Architectures and Training Procedures

For each network a precise architecture had to be designed and training performed, and this section sets out the more precise details of the 8 networks. Note that the loss functions have been described within the results section as they are directly relevant to how each network performs.

3.4.0 - Single Box Network

The single box network performed 20 epochs of training on 100,000 automatically generated boxes, using a batch size of 500. Table 1 displays the precise network architecture used. The input consisted of 16x16 images of a single box, outputting a single dimensional vector as set out in 4.1. This was then tested using the set of sequences of length 1 from the training data. The final fully connected layers have associated dropout, this is an effective way to prevent overfitting, a phenomenon that means networks only learn the presented data ceasing to generalise (Srivastava, 2014).

3.4.1 - Recurrent Network

For the recurrent network, training data was again automatically generated; this time a sequence of three random boxes was generated and the images created. The logical union of the images formed a single image, and then for each following box, the respective pixels were subtracted, leaving a sequence of boxes and images.

Recurrent networks normally train on sequences of data and in this case it would mean a sequence of three boxes. If this were done, it would mean that the first and second in the sequence would be left without earlier versions. This is normally negated by zero padding, a process whereby images with nothing in them are slotted into the sequence before the first. This however reduces computational efficiency. To sidestep this problem, the sequences were sliced into singular data points, sequences of length one, then fed into the network.

Table 2 displays the precise network architecture. The network was trained on 30,000 pieces of data for 50 epochs. It was tested on the sequences of length three from the test data set.

3.4.2 - Isolated Recurrent Networks

Ultimately three versions of the networks used to isolate the issues with the recurrent network were created: the first, which was trained on the first box in a sequence of two, **labelled Isolated A in the graphs**; the second training on the second box in the same sequence, these two were trained for 20 epochs over 100,000 pieces of data, **labelled Isolated B in the graphs**; finally to rule out a lack of training the third was created, this was trained on the first box in a two box sequence, for 50 epochs over 500,000 pieces of data, **labelled Isolated C in the graphs**. The three networks were tested on the corresponding parts of the test data set. The number of epochs was set to ensure convergance.

All three versions had the same architecture, shown in Table 3. The network architecture was reverted back to that of the single box network. The input image resolution was quadrupled to a 32 x 32 pixel image in an attempt to increase accuracy. In addition the data for the network was cleaned up from the previous networks. Specifically it was noted that a large proportion of unlikely images were being created, namely ones where boxes were either not intersecting at all or a smaller box was enveloped wholly in a larger box; this would slow training significantly. These particular cases were barred from being automatically generated. The method of creating the sequences remained identical to the recurrent network.

3.4.3 - YOLO Based Networks

For all three versions of the YOLO based network, the output array has depth 13: a dimension vector and C for the first box, a dimension vector and C for the second box and finally P. With C being the respective confidence of a box being there and P the confidence that it is a box. Although for these networks P is not required, it is left in the code to allow for potential further extension of the network to predict more shapes by simply adding more P variables. As the network only predicts on a single image rather than a sequence, the only image required is the union image of a set of random boxes. This change in requirements of the data allowed for a significant speed up of the automatic generation of the data.

For all three networks, $\lambda_{dims} = 5$, $\lambda_{coords} = 5$, $\lambda_{theta} = 10$, $\lambda_{obj} = 0.5$, $\lambda_{noobj} = 0.5$ and $\lambda_{p} = 1.0$. The introduction of lambda terms, or weightings means that the loss function can penalise individually each term. Consequently more emphasis can be placed on the various specific terms if required. This was done for the rotational term, λ_{theta}, as it was discovered with the earlier networks that the rotation was predicted very poorly.

The test version used an input image of 16 x 16 pixels, and outputted to a grid of 4 x 4, with each grid cell predicting the vector of depth 13. It was trained over 100,000 data points and 50 epochs. The 100,000 data points were split equally over sets of one, two, three and four boxes.

The network following the test network was expanded to use higher fidelity data in the form of a 160 x 160 pixel image. In addition the output grids were expanded to 5 x 5 cells. The architecture was increased to have a few more layers, Table 6 displays the precise details. This was trained over 4000 data points and 100 epochs, again with an equal split between sets of one, two, three and four boxes. The large dimensionality increase of the training data led to significant RAM shortage problems and a slower training, hence the small data set size.

A final network was created, BTBnet, with an architecture following almost exactly the YOLO network's architecture. The number of layers was increased to improve the accuracy of the network. In addition the YOLO architecture is proven to work for its more general task. Table 7 shows the precise network architecture. The loss function and training data dimensions remained the same from the previous network version. Several changes were made to the loss function code for BTBnet to make it use less RAM, this allowed for substantially more data to be loaded. The final network was trained on 20,000 data points again with an equal split between sets of one, two, three and four boxes. It was trained for 100 epochs.

4.0 - Single Box Network Results

To assess the feasibility of using a CNN to predict the location, size and rotation of boxes on floorplans I created an initial network which predicted the parameters of a single box from an image of the box.

The network was designed to predict the location, size and rotation of boxes from 16x16 pixel images of those boxes. The network used a mean squared error loss function. The mean squared error is the mean of the quadratic loss across the output vector. The quadratic loss is appropriate for calculating precise values. However by simply averaging across the output vector, individual weightings cannot be given to each set of values. The final layer uses a linear activation, this allows for all possible values to be output.

The network consisted of 3 convolutional layers and 3 fully connected layers. It was trained for 30 epochs to ensure convergence. **This network is labelled as Single Box in the graphs**.

The network worked well. The networks prediction accuracy for the width, height and x, y, positions of the boxes in the test data set is very good as shown in Graphs 1-4. The network however does not perform well on predicting box rotation as shown in Graph 5. Visual inspections confirmed that the network was working as expected, accurately predicting the location of boxes from given images. As a result of the success with this network, it had been shown that the task was feasible, as at a minimum it is possible to get a CNN to predict a single box. Consequently I moved on to try and create a network to predict more than one box.

4.1 - Recurrent Network Results

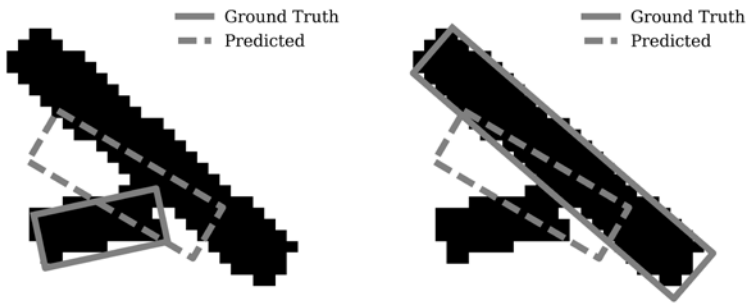
To predict multiple boxes a slightly different approach was taken. A recurrent neural network predicts sequentially, it is akin to a loop in standard algorithmic programming. The idea was to use a recurrent convolutional neural network, a recurrent version of a CNN, to sequentially predict one box after the first. Meaning that for a given arbitrary floorplan the network starts by predicting the first box, then removing this box from the image of the floorplan, and repeating this until no boxes are left.

I used the same network architecture as before, however replacing the first layer with a long short-term memory layer. An LSTM layer in essence 'remembers' what the network has done on previous iterations. This should allow the network to remember that on the previous iteration it had placed the first box and so forth, this ought to improve the accuracy, as it should know where it is in a sequence (Sepp Hochreiter, 1997).

The network was trained for 50 epochs to ensure convergence. **This network is labelled as "Recurrent" in the graphs**.

The network was not performing very well in any of the given metrics as shown in Graphs 1-6. In order to explore this further I created several standard CNN networks which isolated features of the networks as standard CNNs ought to be a good approximation for a single layer of the network. These are the isolated networks described in section 3.4.2. It was discovered that there was a fundamental limitation of the model; a phenomenon which prevented training. With simple visual observation of a number of the networks predictions the problem became clear.

Recc Prediction Problem - Figure 3



It is clear that the predicted rectangles often hover in the centre of the presented images rather than choosing one rectangle in particular to predict. This is caused by there being no one rectangle that the network should always choose to predict first. For instance the network may see two identical images which have their respective boxes to predict flipped, as exampled in Figure 3. This would cause the network to train to pull the predicted box to the bottom in the first image, yet towards the top in the second image. A similar thing happens with both the width, height and rotation. This eventually leads to an overall result that the first box and so on is always predicted as an 'average' of the boxes to be predicted.

4.2 - You Only Look Once Based Network Results

Recent studies have created networks which accurately predict multiple bounding boxes for multiple objects accurately. In reality the prediction of bounding boxes is similar to the problem of predicting the location of rectangles within images of floorplans. This means that a network based on one of these ought to solve the problem (J. Redmon S. D., 2016) (K. He, 2017).

I sought to base a newer version of the network based on one of these networks, specifically the You Only Look Once (YOLO) network (J. Redmon S. D., 2016). The network has several useful properties to it. The way in which it predicts the bounding boxes, by simply predicting the values of the box parameters is very similar to what was previously attempted, meaning that the automated data generation barely had to be modified from the previous networks. It is able to predict a very large number of objects simultaneously, unlike a recurrent net of the type previously attempted, which has to predict sequentially. This condenses the process down to a single step rather than many; it also has the capacity to predict and classify a wide variety of objects. Hence a YOLO based network could be able to predict a variety of different shapes, for instance ovals and triangles in addition to boxes.

The network functions by dividing up an image into a grid. Each grid cell predicts a set number of objects, providing the confidence of that object being there and the bounding box variables. In addition each grid cell predicts a further value, the confidence of object classification, i.e. is it a box or a triangle? Ultimately the network predicts an object by multiplying the object classification by the confidence of the object being there, and checking if this is above a predetermined threshold value.

One particular desired property is not present in the network it cannot predict the rotation of boxes. Fortunately the output of the network can be readily expanded to include a rotational term, and the loss function modified to accommodate this.

To ascertain the viability of this approach and test the methods to be used I created an initial version of the network using 16x16 image inputs, with an output grid of 4x4. I reused the architecture from the previous networks, but reformatted the inputs and outputs, additionally rewriting the loss function to match that of the YOLO network, with the addition of a bounding box rotation term.

$$\boldsymbol{BTBS\ Loss} = \boldsymbol{\lambda_{dims}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \boldsymbol{Mask_{i,j}^{obj}} \left(\left(\sqrt{\boldsymbol{w_i}} - \sqrt{\boldsymbol{\hat{w}_i}}\right)^2 + \left(\sqrt{\boldsymbol{h_i}} - \sqrt{\boldsymbol{\hat{h}_i}}\right)^2 \right)$$

$$+ \boldsymbol{\lambda_{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \boldsymbol{Mask_{i,j}^{obj}} \left((\boldsymbol{x_i} - \boldsymbol{\hat{x}_i})^2 + (\boldsymbol{y_i} - \boldsymbol{\hat{y}_i})^2 \right)$$

$$+ \boldsymbol{\lambda_{theta}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \boldsymbol{Mask_{i,j}^{obj}} \left((\boldsymbol{\theta_i} - \boldsymbol{\hat{\theta}_i})^2 \right)$$

$$+ \boldsymbol{\lambda_{obj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \boldsymbol{Mask_{i,j}^{obj}} \left((\boldsymbol{C_i} - \boldsymbol{\hat{C}_i})^2 \right)$$

$$+ \boldsymbol{\lambda_{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \boldsymbol{NotMask_{i,j}^{obj}} \left((\boldsymbol{C_i} - \boldsymbol{\hat{C}_i})^2 \right)$$

$$+ \boldsymbol{\lambda_{p}} \sum_{i=0}^{S^2} \boldsymbol{Mask_{i,j}^{obj}} \left((\boldsymbol{p_i(c)} - \boldsymbol{\hat{p}_i(c)})^2 \right)$$

The Test YOLO based network loss function. The loss function is an exact copy of the YOLO loss function (J. Redmon S. D., 2016), aside from the addition of the term for the rotation of the bounding boxes, and all terms having a lambda value. The hat above a value denotes that it is the predicted value. The values in the loss function are as follows: w, box width; h, box height; x, box x position; y, box y position; C, the confidence of there being a box in a cell; p, the confidence of that objet being a box; , λ_{dims}, the scale for w and h; λ_{coords}, the scale for x and y; λ_{theta}, the scale for the rotation; λ_{obj}, the scale for the object predictor term; λ_{noobj}, the term for the logical not predictor term; λ_{p}, the scale for the classifier term; B, the vector of the boxes in each cell; S^2, the array of the cells; Mask penalises if and object is not present and should be, and finally NotMask penalises if an object is present and should not be.

The network was trained for 50 epochs to ensure convergence. **This network is labelled as "Test YOLO" in the graphs**.

The performance metrics were particularly poor as shown in Graphs 1-7, however visual inspection of results were much more promising, specifically the network did not have the "box averaging" problem from the recurrent network. As the initial results seemed promising, I increased the size of the inputs substantially to aim for greater precision in the model.

The network that was used to test the larger image resolution increased it to 160x160 pixels, 100 times larger than what was used in the previous network, the output grid was also increased to 5x5. The network architecture was largely kept the same, though batchnormalisation was added to the end of each convolutional layer as this increases network performance (J. Redmon S. D., 2016). I also reformulated the loss function slightly to increase computational performance.

The loss function was reformulated to penalise the coordinates, dimensions and angle only linearly as opposed to quadratically. This ensures that penalties don't become very small with small errors as the goal is to aim for numerical accuracy. The width and height were also no longer penalised by their square root, as this was deemed to be an unnecessary step, and over complicated calculations.

$$\boldsymbol{BTBnet\ Loss} = \boldsymbol{\lambda_{dims}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \boldsymbol{Mask_{i,j}^{obj}} \left((\boldsymbol{w_i} - \boldsymbol{\hat{w}_i}) + (\boldsymbol{h_i} - \boldsymbol{\hat{h}_i}) \right)$$

$$+\lambda_{coord}\sum_{i=0}^{S^2}\sum_{j=0}^{B} Mask_{i,j}^{obj}\left((x_i-\hat{x}_i)+(y_i-\hat{y}_i)\right)$$

$$+\lambda_{theta}\sum_{i=0}^{S^2}\sum_{j=0}^{B} Mask_{i,j}^{obj}\left((\theta_i-\hat{\theta}_i)\right)$$

$$+\lambda_{obj}\sum_{i=0}^{S^2}\sum_{j=0}^{B} Mask_{i,j}^{obj}\left((C_i-\hat{C}_i)^2\right)$$

$$+\lambda_{noobj}\sum_{i=0}^{S^2}\sum_{j=0}^{B} NotMask_{i,j}^{obj}\left((C_i-\hat{C}_i)^2\right)$$

$$+\lambda_{p}\sum_{i=0}^{S^2} Mask_{i,j}^{obj}\left((p_i(c)-\hat{p}_i(c))^2\right)$$

The BTBnet loss function. Significant modifications have been made from the YOLO loss function (J. Redmon S. D., 2016). As in the test network loss, a rotation term is added and all terms have a lambda value. In addition the dimensional values are linearly penalised, and the box size is no longer penalised by the square root. The hat above a value denotes that it is the predicted value. The values in the loss function are as follows: w, box width; h, box height; x, box x position; y, box y position; C, the confidence of there being a box in a cell; p, the confidence of that objet being a box; , λ_{dims}, the scale for w and h; λ_{coords}, the scale for x and y; λ_{theta}, the scale for the rotation; λ_{obj}, the scale for the object predictor term; λ_{noobj}, the term for the logical not predictor term; λ_{p}, the scale for the classifier term; B, the vector of the boxes in each cell; S^2, the array of the cells; Mask penalises if and object is not present and should be, and finally NotMask penalises if an object is present and should not be.

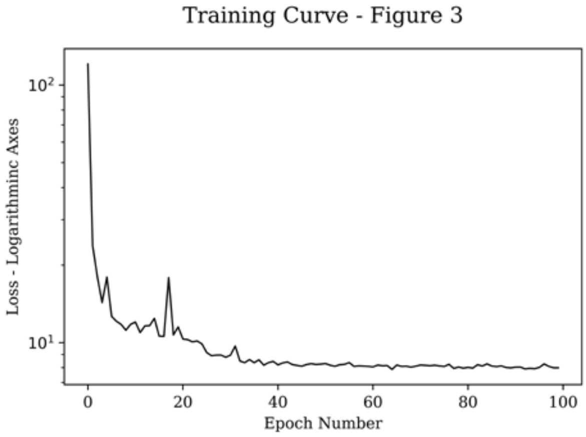
This larger network was trained for 100 epochs, to ensure convergence. **This network is labelled as "Large YOLO" in the graphs**.

The performance metrics were very good coming close to that of the single box network. In addition the revised network predicted the rotation much better than the other networks and this is largely due to the much greater emphasis placed on the rotational term as I noticed that this is something the other networks were particularly poor at predicting. Visual inspections of results seemed to corroborate the overall metrics, demonstrating robust predictions. The classification accuracy is not particularly good, specifically as shown in Graph 7, the network has a very narrow threshold peak, meaning the classification accuracy is very sensitive to changes in threshold.

To increase the accuracy of the model, especially the classification accuracy, I substantially increased the network size, to match more closely that of the already proven DarkNet (J. Redmon S. D., 2016), which is the network architecture used by YOLO. This consists of 19 convolutional layers as opposed to the 8 for the previous networks. This new network became the final network, BTBnet.

In addition improvements were made to the code for the loss function, freeing up substantial RAM, This meant that I could train on a larger batch size and a larger data set.

The network was trained for 100 epochs, to ensure convergence, this is shown in Figure 4.



BTBnet has a smaller interquartile range and overall range in all metrics as shown in Graphs 1 -5. This suggests overall a better accuracy than the previous network, which had a smaller architecture. It even performs better than the single box network at predicting box width and height, something none of the other networks do. BTBnet has larger spread on its outliers than the previous network and this leads to a higher mean standard deviation. This is a consequence of the standard deviation being sensitive to outliers unlike the range and interquartile range. For most of the data, BTBnet is better than the previous network, however for some examples it makes very inaccurate guesses.

Importantly BTBnet, unlike every other network makes no systematic error in any of the predicted parameters, as the distribution around zero is almost exactly symmetrical, aside from a few outliers. This leads to a mean modal value of zero for the network. This is a part of achieving good accuracy.

Graph 7 plots the number of correct box predictions for given threshold values over the three YOLO based networks. Note that the number of data points is 4000, hence this would be a perfect prediction. The difference in prediction accuracy is clear. The first YOLO based network performs very badly compared to the others, achieving a maximum correct prediction of 6.9% at a threshold of 0.315. The Large YOLO network achieves a prediction maximum of 36.5% at a threshold of 0.12. BTBnet achieves a better result of 42.2% at a threshold of 0.323. It also has a substantially broader peak than the previous network meaning that variation in the confidence value will affect the performance of BTBnet less.

5.0 - Discussion

Graphs 1 through 5 show the box plots of the actual error; the closer the modal value to zero, the better. This is because a perfect prediction would have an error of zero. Large differences between the modal value and zero suggest systematic errors. Closer spreads of the data demonstrate better accuracy. The ideal to aim for is tight data with a modal value of zero.

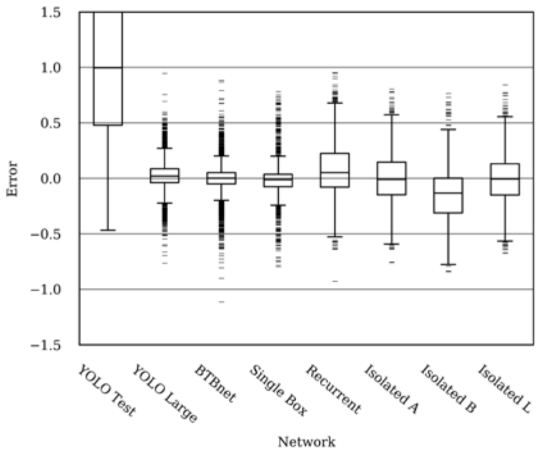
Graph 6 shows aggregate values of the results; specifically the mean average standard distribution, to show data spread for each network, and mean modal value, to show the average systematic error in each network. With Graph 6, the lower the mean standard deviation the better and the closer the mean modal value to zero, the better.

Note that in Graph 5, although appearing similar to Graphs 1 through 4, the scale on the vertical axis is doubled, hence the errors appear in contrast smaller than they actually are.

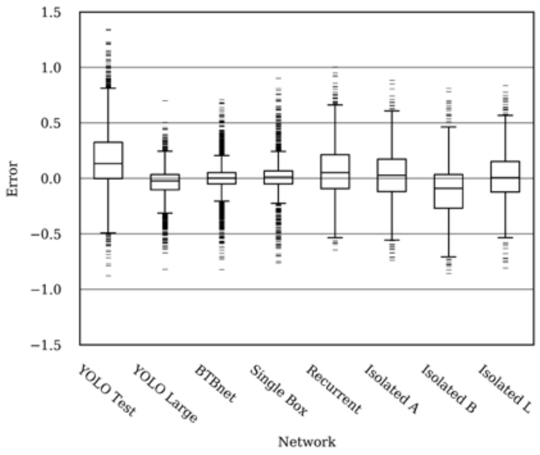
Graph 7 plots the number of correct box predictions for given threshold values over the three YOLO based networks. The more correct predictions, the better.

Comparing all 8 networks, the single box network still performs comparatively well, aside from the prediction of rotation. In fact it predicts the location of boxes better than even BTBnet, the final network. This is as a consequence of its much easier task of predicting a single box. The four versions of the recurrent network perform quite badly, especially in relation to the two final networks and the single box network. Unsurprisingly, they all perform equally as badly, as a direct consequence of the box averaging phenomenon. The test version of the YOLO based network performs the worst overall as it is trying to predict boxes simultaneously, yet uses a very low resolution input image. The final two versions of the YOLO based network are overall outperforming every other network significantly, as discussed in section 2.3. BTBnet is performing better than its earlier version, leading to the conclusion that overall BTBnet is performing the best.

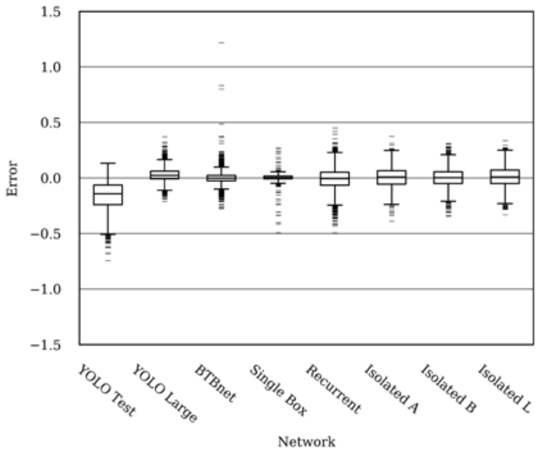
Width Error Distruibution - Graph 1



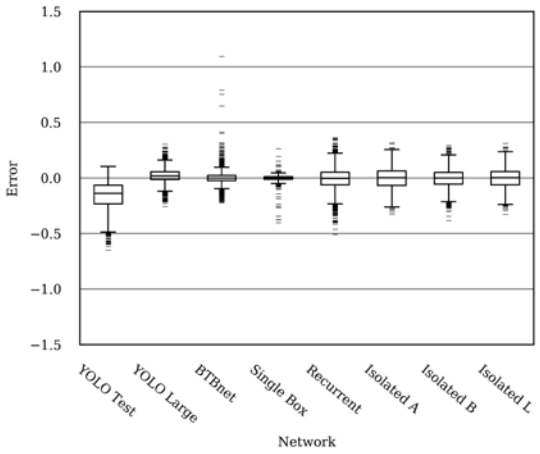
Height Error Distruibution - Graph 2



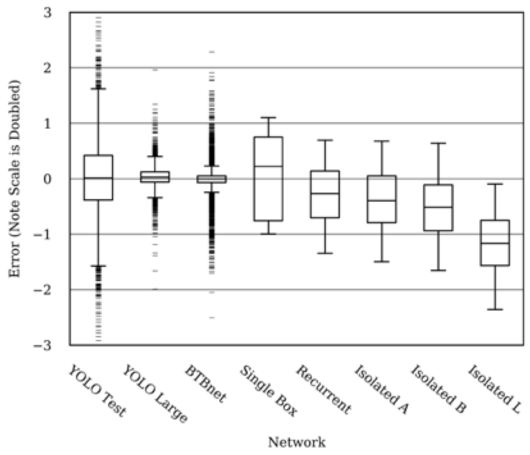
X Positional Error Distruibution - Graph 3



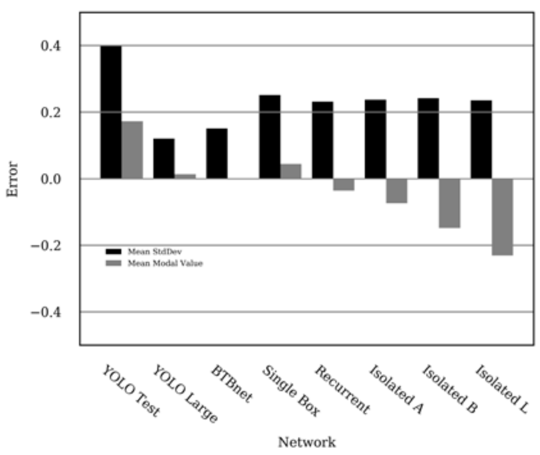
Y Positional Error Distruibution - Graph 4



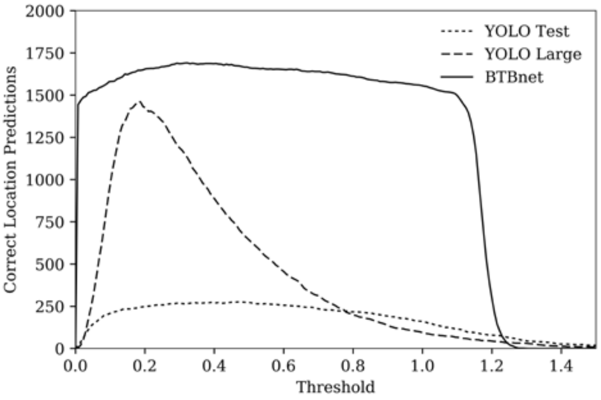
Angular Error Distruibution - Graph 5



Mean StdDev and Mean Modal Value - Graph 6



Correct Box Loaction Predictions - Graph 7

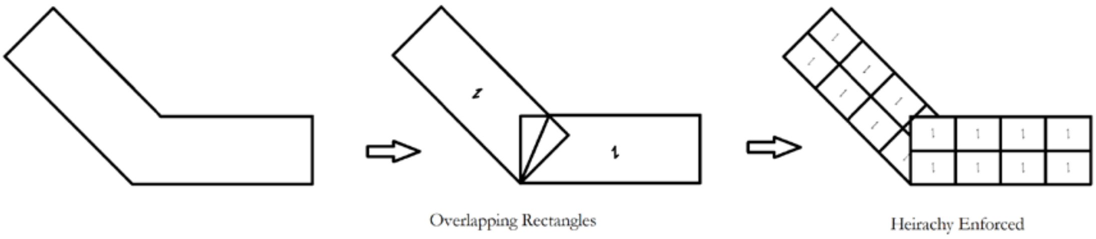


The appendix, section 8.0, contains enlarged versions of Graphs 1 through 7 for convenience.

The next step in the process of automated design from the output of BTBnet would be laying out the primary members within the original floorplan inputted into the network. The predicted rectangles would be divided up into a set number of divisions in each of their respective orthogonal axes. Primary beams would then be laid out along these divisions, and columns placed on the intersections. The model of a single floorplan and its associated columns would then be copied vertically to the buildings desired number of stories. If designing using a solid slab, columns would be placed at the same locations as when designing using beams. The members would then be designed to resist imposed loads. The members and number of divisions would then be optimised, for example using the GA based method of P. Sharafi (2012).

An artefact of this method, which seems problematic is overlapping rectangles which are generated by the CNN, every single image in Figure 6 has this problem. The problem however can be solved by enforcing a simple hierarchy, whereby one rectangle is deemed “more important” than the others, hence laying its beams out first. The order of hierarchy would most likely be dictated by the area of each rectangle, the larger the area the more important. This would reflect manual practises. The process is shown in Figure 5.

Algorithmic Heirachy Enforcement - Figure 5



6.0 - Conclusions

The goal of the research was to expand the number of structures that can be automatically designed, by presenting a method using a neural network, which decomposes arbitrary floorplans into rectangles. This allows a much wider range of structures to be automatically designed, as beams can be placed at regular intervals along the axis of the rectangles.

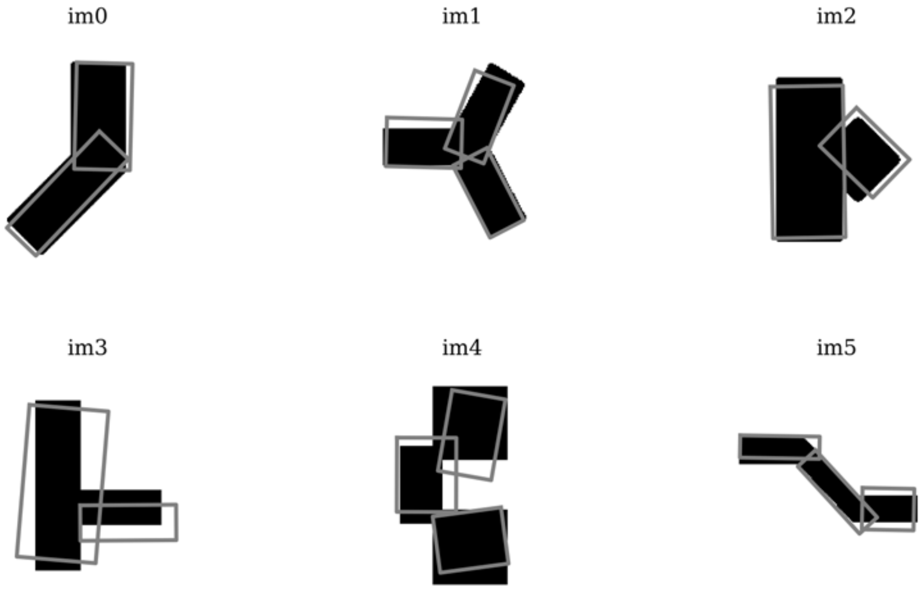
A number of CNN and RCNN networks have been progressively developed leading to the final BTBnet, which is accurately and robustly predicting correct boxes to paint on images. The network performs the best overall out of the networks tested, relying on knowledge gained from testing the previous networks and using a proven architecture from YOLO.

Figure 6 provides visual examples of BTBnet predicting on a selection of test images, which were hand drawn to imitate potential floor plans. Although this is a small dataset the network appears to be behaving in a logical and robust manner, predicting what a human might interpret, despite only having seen automatically generated data. The overall method for the research, using neural networks to decompose floorplans into boxes seems to be effective; certainly BTBnet is performing this task well.

Despite the relatively accurate predictions shown in Figure 6, the network still makes a substantial number of errors. Although correctly classifying 42.2% of boxes to be within the correct grid cell, it classifies incorrectly 57.8% of the time which is the majority. Though correct predictions may be made despite the output vectors of the classification not necessarily matching, this is unlikely to be a large amount. This high rate of errors is an issue, as commercial software must be robust enough to work in almost every single case, thus leaving a lot to be desired of BTBnet.

For the method to be adopted in software that could actually be used, especially given this is for civil engineering which necessitates very high accuracy to ensure the safety of structures, BTBnet would have to at least match and probably exceed human capability. It is clear even from the examples shown in Figure 6 that a human would perform much better than BTBnet in its current state.

Examples From BTBA - Figure 6



im0

im1

im2

im3

im4

im5

Interestingly though, in solving the problem of decomposing arbitrary floorplans into simple geometric shapes, I have created a method of decomposing high dimensional images into simple geometric shapes. This is a much more general problem than the specific structural engineering problem I have focussed on. This has potentially much wider applications than simply enabling the automated design of a wider range of structures, for example within urban mapping, it could be used to map outlines of buildings in cities, automating this as a process.

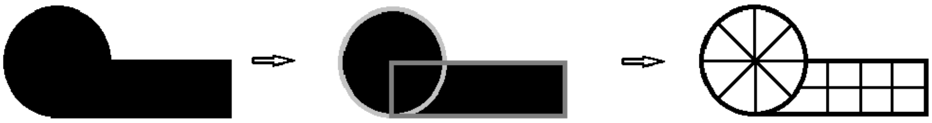
6.1 - Recommendations

BTBnet shows potential and it demonstrates that the method may eventually be capable of automatically decomposing floorplans into rectangles with an accuracy that is good enough to be used commercially, but clearly is not capable with the current network. In order to achieve this the network has to improve its accuracy substantially, certainly close to that of a humans.

There are several potential improvements that could be made with relative ease, that combined might make enough of a difference. Foremost the image size should be increased, BTBnet uses images of 160x160 pixels, yet the YOLO network uses images of 448x448, 8 times the resolution. This specifically would have been done if it were not for the hardware limitations, which would have made the process of training too slow. In addition the improvements described in the further YOLO network versions, YOLO9000 (J. Redmon, 2016) and YOLO V3 (J. Redmon, YOLOv3: An Incremental Improvement, 2018) should be implemented. These improvements combined may be able to increase accuracy to a level where the network is suitably reliable enough for commercial software. This is something which should be investigated further.

The network has further problems; at the moment it only decomposes into rectangles, this is useful for a large number of different floorplans. However many potential floorplans may not be well represented by a series of rectangles. For instance in Figure 7, the left hand side of the geometry is clearly circular in nature, this would be poorly approximated by rectangles. Instead the network should recognize this as a circle, and lay out primary members in a radial fashion as opposed to along the axis of a rectangle. This is at least what a human engineer would do.

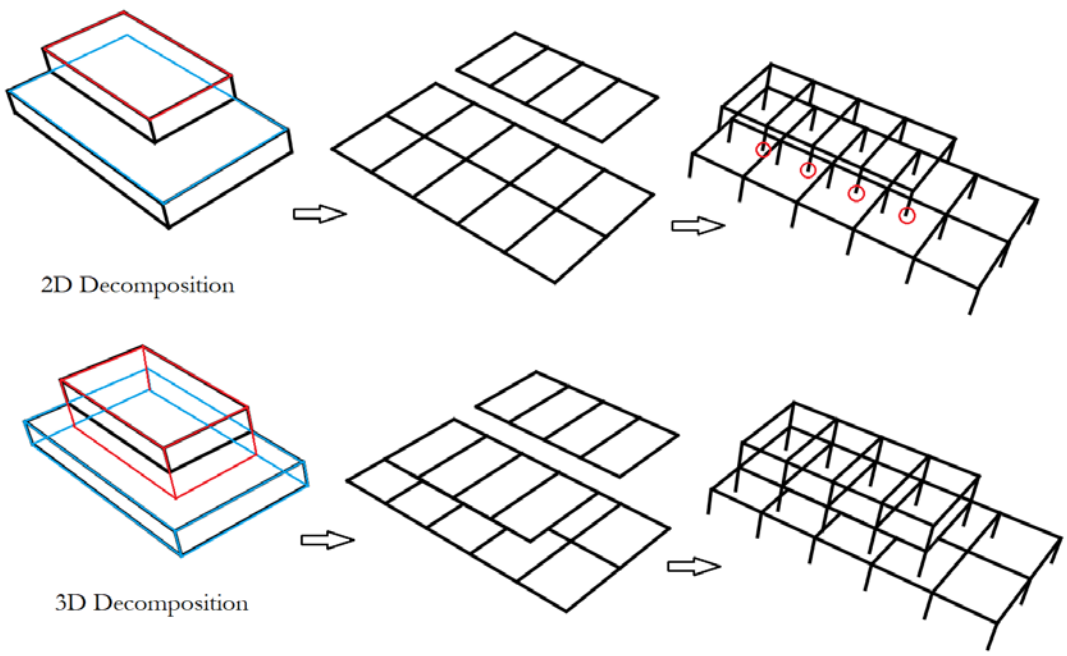
Demonstration of Need to Decompose into Different Shapes - Figure 7



Fortunately because the BTBnet is based on the YOLO network, which classifies multiple different objects, there is no reason that BTBnet could not do the same. To do this the output vector would need to be increased to accommodate for more shapes, and training data would have to be widened to include those different shapes. This ought to lead to a much more useful network that can decompose a much wider array of floorplans in a logical manner.

The method as it currently stands suffers from another critical problem as it is only able to design buildings which have a floorplan that is uniform over the stories of a building. It might seem that to design buildings which change floorplan over height would be straight forward using the current method. Simply feed the floorplans for each story, lay out the primary beams, and then put columns at every beam intersection? This naive approach could quickly lead to problems as shown in Figure 8, specifically with a simple 2D decomposition in which columns are placed at beam intersections. This can lead to column ends floating in mid-air. However if a 3D decomposition approach were taken, whereby 3D boxes were then predicted, with beams laid on horizontal divisions and columns lying on the intersection of beams, then the errors can be eliminated.

Demonstration of Need for 3D Decomposition - Figure 8



This could feasibly be implemented with modification to the current method of using a CNN. The network instead of being fed a single floorplan, would be fed all floorplans simultaneously. The network would convolve over the 3D input image, and would then predict the location of various 3D shapes. An appropriate hierarchy would be formulated, likely placing taller shapes over shorter ones, then these shapes would be appropriately divided into beams and columns, using an algorithmic approach.

7.0 - Bibliography

A. Krizhevsky, I. S. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, (pp. 1097-1105).

A. Nimtawat, P. N. (2009). Automated layout design of beam-slab floors using a genetic algorithm. *Computers & Structures*, 87(21-22), 1308-1330.

A. Nimtawat, P. N. (2010). A genetic algorithm fo beam-slab layout design of rectilinear floors. *Engineering Structures*, 32(11), 3488-3500.

A. Nimtawat, P. N. (2011). Simple Particle Swarm Optimization for Solving Beam-Slab Layout Design Problems. *Procedia Engineering*, 14, 1392-1398.

Bathurst, L. L. (1965). Automated Building Design. *In Proceedings of the SHARE design automation project* (pp. 17.1 - 17.7). ACM.

C. M. Chan, D. E. (1995). Automatic Optimal Design of Tall Steel Building Frameworks. *Journal of Structural Engineering*, 121(5), 838-847.

Chollet, F. e. (2015). *Keras*. Retrieved from https://keras.io

D. P. Kingma, J. B. (2014). Adam: A Method for Stochastic Optimization. *arXiv:1412.6980*.

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 90--95.

J. Redmon, A. F. (2016). YOLO9000: Better, Faster, Stronger. *arXiv:1612.08242*.

J. Redmon, A. F. (2018). YOLOv3: An Incremental Improvement. *arXiv:1804.02767*.

J. Redmon, S. D. (2016). You Only Look Once: Unified, Real-Time Object Detection. *Proceedings of the IEEE conference on computer vision and pattern recognition*, (pp. 779-788).

K. He, G. G. (2017). Mask R-CNN. *IEEE International Conference on Computer Vision* (pp. 2980-2988). IEEE.

K. W. Park, D. E. (1999). Pareto-Optimal Conceptual Design of the Structural Layout of buildings Using a Multicriteria Genetic Algorithm. *Computer-Aided Civil and Infrastructure Engineering*, 14(3), 163-170.

M. Abadi, A. A. (2015). *TensorFlow: Large-scale machine learning on heterogeneous systems*. Retrieved from tensorflow.org.

P. Sharafi, L. H. (2015). Conceptual design optimization of rectilinear building frames: A knapsack problem approach. *Engineering Optimization, Volume 47*.

P. Sharafi, M. N. (2012). Heuristic Approach for Optimum Cost and Layout Design of 3D Reinforced Concrete Frames. *Journal of Structural Engineering, Volume 138, Issue 7*.

Python Software Foundation. (2018). Retrieved from https://www.python.org/

R. Sacks, A. W. (1997). A project model for an automated building system. *Automation in Construction, Volume 7*, 21-34.

R. Sacks, A. W. (2000). Structural Design in an automated building system. *Automation in Construction, Volume 10*, 181-197.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks*, 61, 85-117.

Sepp Hochreiter, J. S. (1997). Long Short-Term Memory. *Neural Computation Volume 9*, 1735 - 1780.

Srivastava, N. H. (2014). Dropout: A simple way to prevent neural networks from overfitting. *he Journal of Machine Learning Research*, 5(1), 1929-1958.

Travis E, O. (2006). *A guide to NumPy.* USA: Trelgol Publishing.

Trimble. (2018). Tekla Structural Designer.

Vinod Nair, G. E. (2010). Rectified linear units improve restricted boltzmann machines. *ICML'10 Proceedings of the 27th International Conference on International Conference on Machine Learning* (pp. 807 n-814). Omnipress.

W. S. McCulloch, W. P. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4), 115-133.

Y. LeCun, L. B. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278-2324.

8.0 - Appendix

Layer	Filters/ Nodes	Window Size	Strides	Activation	Dropout	Output Dims
X						16 x 16
2D Convolutional	50	4 x 4	1 x 1	ReLU		16 x 16
MaxPool		2 x 2	2 x 2			8 x 8
2D Convolutional	100	4 x 4	1 x 1	ReLU		
2D Convolutional	100	4 x 4	1 x 1	ReLU		
MaxPool		2 x 2	1 x 1			4 x 4
Connected	500			ReLU	0.25	500 x 1
Connected	250			ReLU	0.25	250 x 1
Connected	5			Linear		5 x 1
Y						5 x 1

Table 1 – Single Box Network Architecture

Layer	Filters/ Nodes	Window Size	Strides	Activation	Dropout	Output Dims
X						16 x 16
2D Conv LSTM	50	4 x 4	1 x 1	ReLU		16 x 16
MaxPool		2 x 2	2 x 2			8 x 8
2D Convolutional	100	4 x 4	1 x 1	ReLU		
2D Convolutional	100	4 x 4	1 x 1	ReLU		
MaxPool		2 x 2	1 x 1			4 x 4
Connected	500			ReLU	0.25	500 x 1
Connected	250			ReLU	0.25	250 x 1
Connected	5			Linear		5 x 1
Y						5 x 1

Table 2 – Recurrent Network Architecture

Layer	Filters/ Nodes	Window Size	Strides	Activation	Dropout	Output Dims
X						32 x 32
2D Convolutional	50	4 x 4	1 x 1	ReLU		32 x 32
MaxPool		2 x 2	2 x 2			16 x 16
2D Convolutional	100	4 x 4	1 x 1	ReLU		
2D Convolutional	100	4 x 4	1 x 1	ReLU		
MaxPool		2 x 2	1 x 1			8 x 8
Connected	500			ReLU	0.25	500 x 1
Connected	250			ReLU	0.25	250 x 1
Connected	5			Linear		5 x 1
Y						5 x 1

Table 3 – Isolated Recurrent Network Architectures

Layer	Filters/	Window Size	Strides	Activation	Output Dims
X					16 x 16
2D Convolutional	50	4 x 4	1 x 1	ReLU	16 x 16
MaxPool		2 x 2	2 x 2		8 x 8
2D Convolutional	200	4 x 4	1 x 1	ReLU	
2D Convolutional	200	4 x 4	1 x 1	ReLU	
MaxPool		2 x 2	1 x 1		4 x 4
2D Convolutional	100	4 x 4	1 x 1	ReLU	
2D Convolutional	100	4 x 4	1 x 1	ReLU	
2D Convolutional	100	4 x 4	1 x 1	ReLU	
2D Convolutional	13	4 x 4	1 x 1	Linear	
Y					4 x 4 x 13

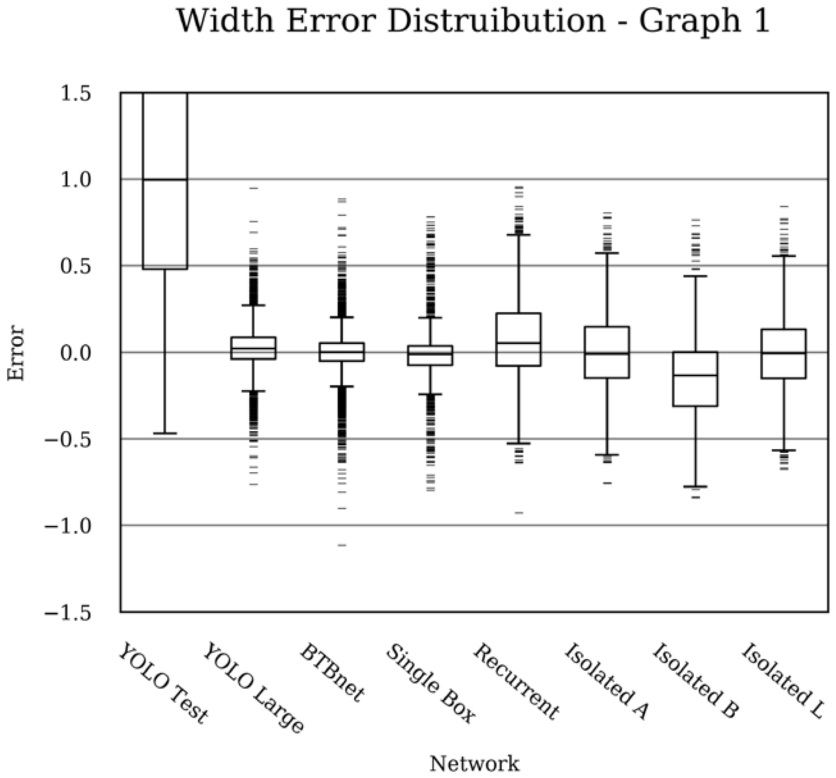
Table 4 – Test Version of YOLO Based Network Architecture

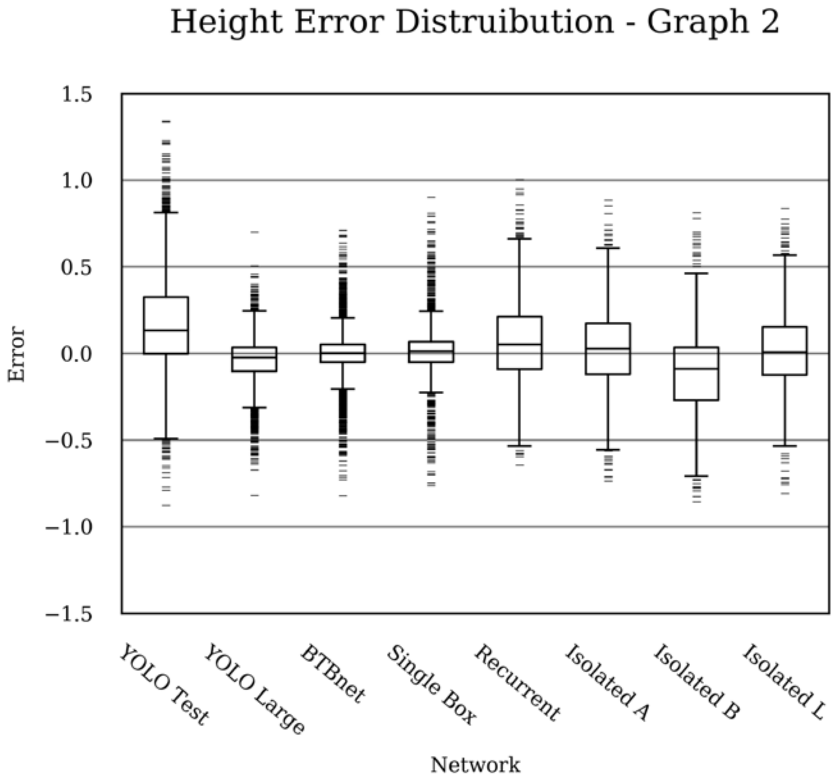
Layer	Filters	Window Size	Strides	Activation	Output Dims
X					160 x 160
2D Convolutional	50	4 x 4	1 x 1	ReLU	160 x 160
MaxPool		4 x 4	4 x 4		40 x 40
2D Convolutional	100	4 x 4	1 x 1	ReLU	
MaxPool		2 x 2	1 x 1		20 x 20
2D Convolutional	200	4 x 4	1 x 1	ReLU	
2D Convolutional	200	4 x 4	1 x 1	ReLU	
MaxPool		2 x 2	1 x 1		10 x 10
2D Convolutional	100	4 x 4	1 x 1	ReLU	
2D Convolutional	100	4 x 4	1 x 1	ReLU	
MaxPool		2 x 2	1 x 1		5 x 5
2D Convolutional	100	4 x 4	1 x 1	ReLU	
2D Convolutional	13	4 x 4	1 x 1	Linear	
Y					5 x 5 x 13

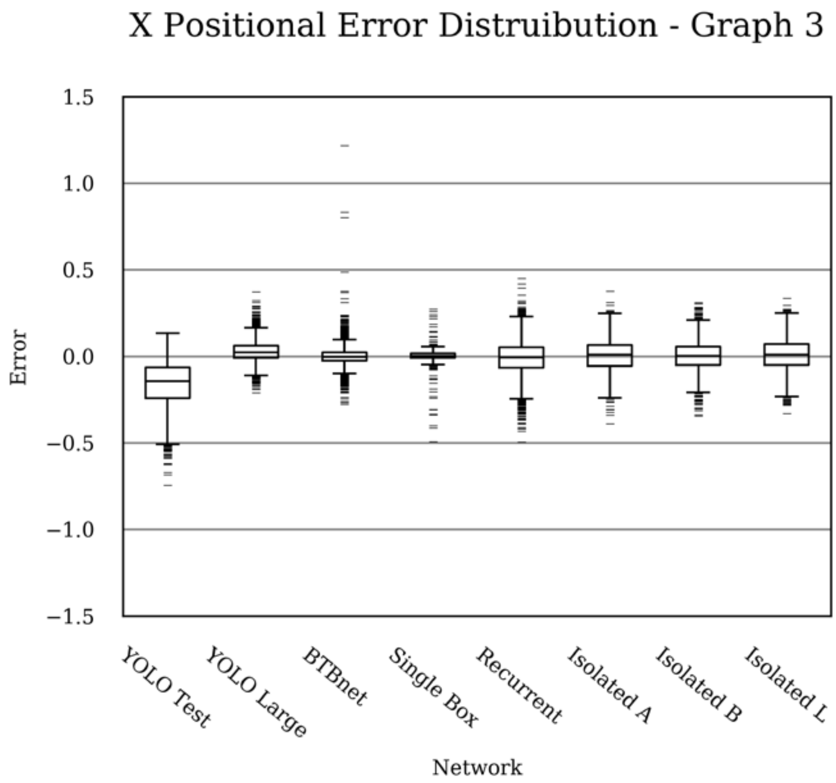
Table 5 – YOLO Based Network Architecture

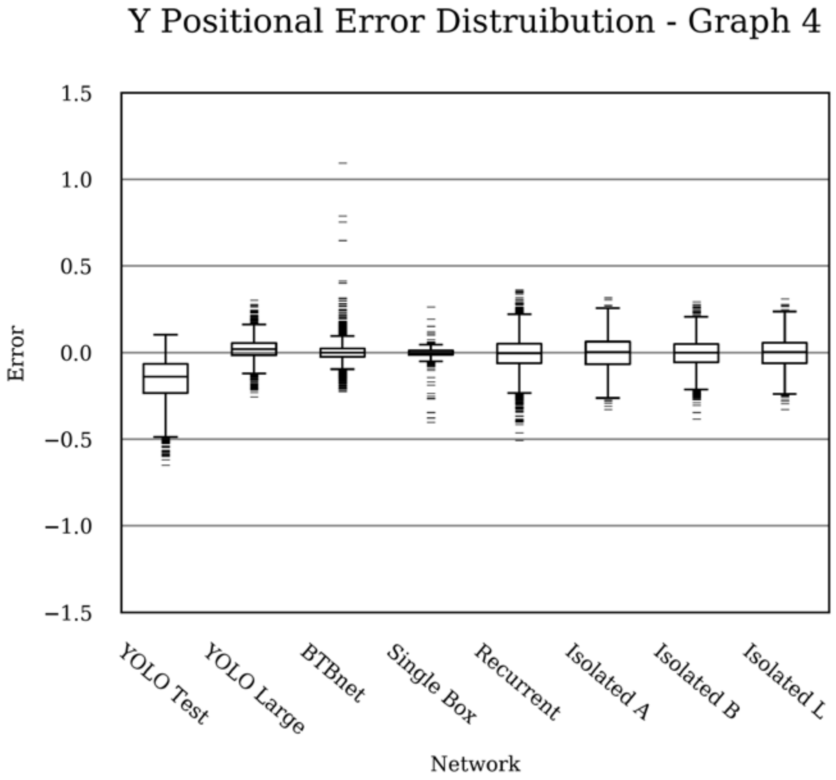
Layer	Filters	Window Size	Strides	Activation	Output Dims
X					160 x 160
2D Convolutional	25	3 x 3	1 x 1	ReLU	160 x 160
MaxPool		2 x 2	2 x 2		80 x 80
2D Convolutional	50	3 x 3	1 x 1	ReLU	
MaxPool		2 x 2	1 x 1		40 x 40
2D Convolutional	100	3 x 3	1 x 1	ReLU	
2D Convolutional	50	1 x 1	1 x 1	ReLU	
2D Convolutional	100	3 x 3	1 x 1	ReLU	
MaxPool		2 x 2	1 x 1		20 x 20
2D Convolutional	200	3 x 3	1 x 1	ReLU	
2D Convolutional	100	1x 1	1 x 1	ReLU	
2D Convolutional	200	3 x 3	1 x 1	ReLU	
MaxPool		2 x 2	1 x 1		10 x 10
2D Convolutional	400	3 x 3	1 x 1	ReLU	
2D Convolutional	200	1 x 1	1 x 1	ReLU	
2D Convolutional	400	3 x 3	1 x 1	ReLU	
2D Convolutional	200	1 x 1	1 x 1	ReLU	
2D Convolutional	400	3 x 3	1 x 1	ReLU	
MaxPool		2 x 2	1 x 1		5 x 5
2D Convolutional	800	3 x 3	1 x 1	ReLU	
2D Convolutional	400	1 x 1	1 x 1	ReLU	
2D Convolutional	800	3 x 3	1 x 1	ReLU	
2D Convolutional	400	1 x 1	1 x 1	ReLU	
2D Convolutional	800	3 x 3	1 x 1	ReLU	
2D Convolutional	13	1 x 1	1 x 1	ReLU	
Y					5 x 5 x 13

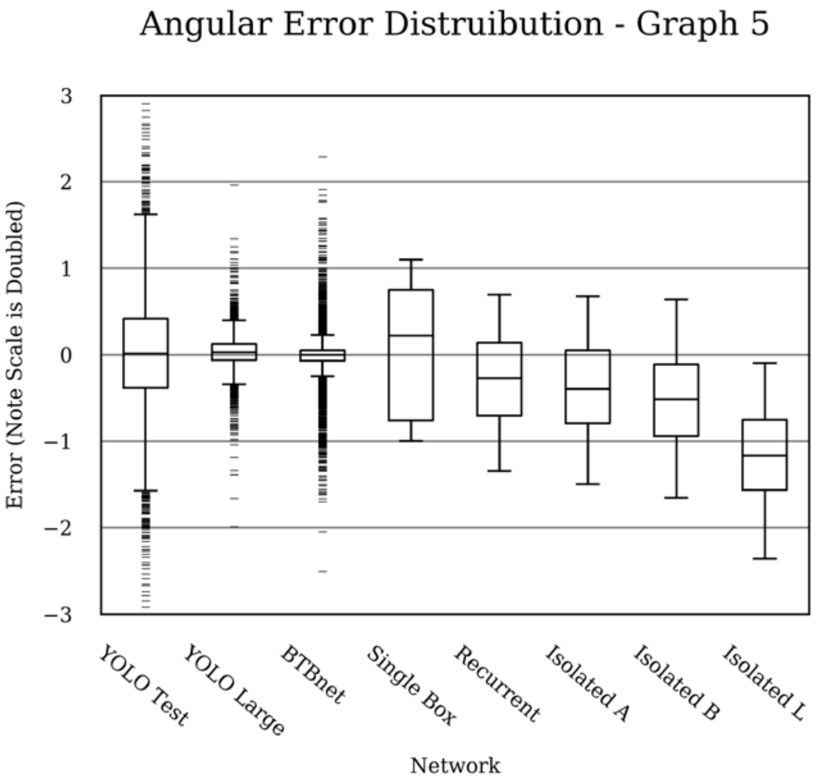
Table 6 – BTBnet Architecture

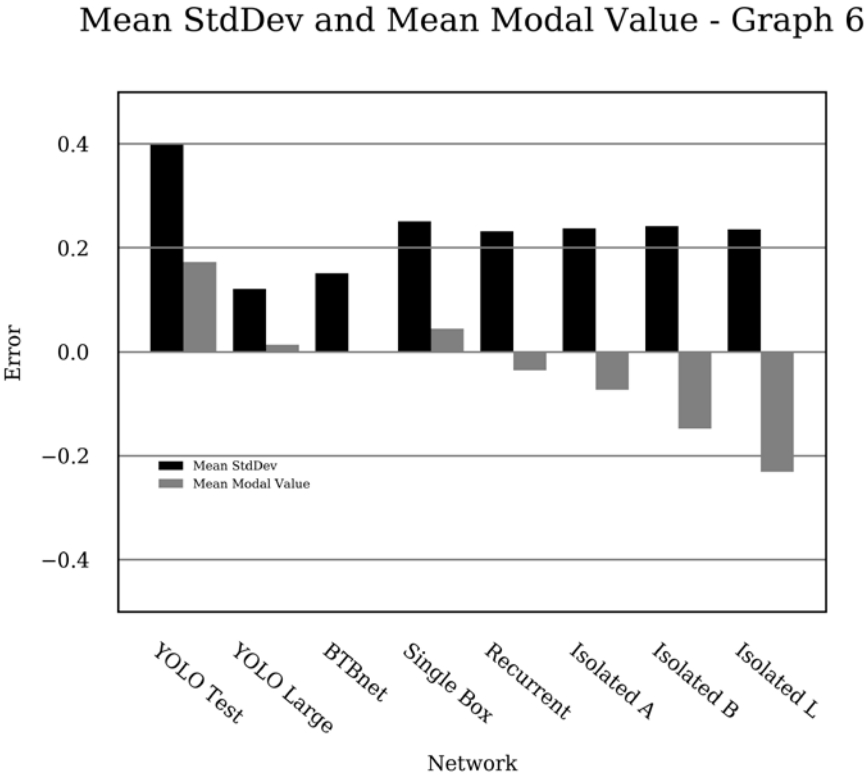

Width Error Distruibution - Graph 1
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
Error
YOLO Test
YOLO Large
BTBnet
Single Box
Recurrent
Isolated A
Isolated B
Isolated L
Network

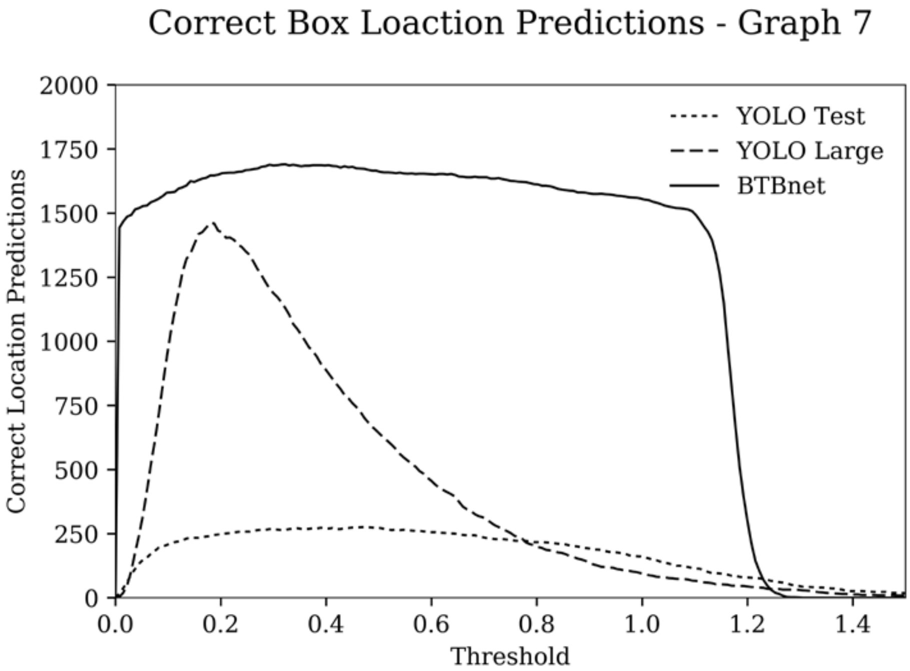

Height Error Distruibution - Graph 2
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
Error
YOLO Test
YOLO Large
BTBnet
Single Box
Recurrent
Isolated A
Isolated B
Isolated L
Network


X Positional Error Distruibution - Graph 3
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
Error
YOLO Test
YOLO Large
BTBnet
Single Box
Recurrent
Isolated A
Isolated B
Isolated L
Network


Y Positional Error Distruibution - Graph 4
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
Error
YOLO Test
YOLO Large
BTBnet
Single Box
Recurrent
Isolated A
Isolated B
Isolated L
Network


Angular Error Distruibution - Graph 5
Error (Note Scale is Doubled)
3
2
1
0
−1
−2
−3
YOLO Test
YOLO Large
BTBnet
Single Box
Recurrent
Isolated A
Isolated B
Isolated L
Network


Mean StdDev and Mean Modal Value - Graph 6
Error
0.4
0.2
0.0
−0.2
−0.4
Mean StdDev
Mean Modal Value
YOLO Test
YOLO Large
BTBnet
Single Box
Recurrent
Isolated A
Isolated B
Isolated L
Network


Correct Box Loaction Predictions - Graph 7
Correct Location Predictions
Threshold
0
250
500
750
1000
1250
1500
1750
2000
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
YOLO Test
YOLO Large
BTBnet

Code Example

Specifically this builds then trains BTBnet.

```python
"""
"Bot The Builder"

Neural network based on the YOLO network architecture.

Created - 03.04.2018
Author  - Michael Cooper
"""

import keras
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, BatchNormalization
import keras.backend as K
import numpy as np
import Utils

################################################################################

def createmodel():
    """
    Creates the BTB network architecture.

    In  - [Z,160,160, 1]
    Out - [Z, 5, 5,13]
    """

    model = Sequential()

    # 160x106 Layers

    model.add(Conv2D(filters = 25, kernel_size = (3, 3), strides = (1,1),
                     activation = 'relu', padding = 'same',
                     input_shape=(160,160,1)))
    model.add(BatchNormalization())

    # 80x80 Layers

    model.add(MaxPooling2D(pool_size = (2, 2), strides = (2,2)))

    model.add(Conv2D(filters = 50, kernel_size = (3, 3), strides = (1,1),
                     activation = 'relu', padding = 'same'))
    model.add(BatchNormalization())

    # 40x40 Layers

    model.add(MaxPooling2D(pool_size = (2, 2), strides = (2,2)))

    model.add(Conv2D(filters = 100, kernel_size = (3, 3), strides = (1,1),
                     activation = 'relu', padding = 'same'))
    model.add(BatchNormalization())

    model.add(Conv2D(filters = 50, kernel_size = (1, 1), strides = (1,1),
                     activation = 'relu', padding = 'same'))
    model.add(BatchNormalization())

    model.add(Conv2D(filters = 100, kernel_size = (3, 3), strides = (1,1),
                     activation = 'relu', padding = 'same'))
    model.add(BatchNormalization())

    # 20x20 Layers

    model.add(MaxPooling2D(pool_size = (2, 2), strides = (2,2)))

    model.add(Conv2D(filters = 200, kernel_size = (3, 3), strides = (1,1),
                     activation = 'relu', padding = 'same'))
    model.add(BatchNormalization())

    model.add(Conv2D(filters = 100, kernel_size = (1, 1), strides = (1,1),
                     activation = 'relu', padding = 'same'))
    model.add(BatchNormalization())

    model.add(Conv2D(filters = 200, kernel_size = (3, 3), strides = (1,1),
                     activation = 'relu', padding = 'same'))
    model.add(BatchNormalization())

    # 10x10 Layers

    model.add(MaxPooling2D(pool_size = (2, 2), strides = (2,2)))

    model.add(Conv2D(filters = 400, kernel_size = (3, 3), strides = (1,1),
                     activation = 'relu', padding = 'same'))
    model.add(BatchNormalization())

    model.add(Conv2D(filters = 200, kernel_size = (1, 1), strides = (1,1),
                     activation = 'relu', padding = 'same'))
    model.add(BatchNormalization())

    model.add(Conv2D(filters = 400, kernel_size = (3, 3), strides = (1,1),
                     activation = 'relu', padding = 'same'))
    model.add(BatchNormalization())

    model.add(Conv2D(filters = 200, kernel_size = (1, 1), strides = (1,1),
                     activation = 'relu', padding = 'same'))
    model.add(BatchNormalization())

    model.add(Conv2D(filters = 400, kernel_size = (3, 3), strides = (1,1),
                     activation = 'relu', padding = 'same'))
    model.add(BatchNormalization())

           # 5x5 Layers

           model.add(MaxPooling2D(pool_size = (2, 2), strides = (2,2)))

           model.add(Conv2D(filters = 800, kernel_size = (3, 3), strides = (1,1),
                            activation = 'relu', padding = 'same'))
           model.add(BatchNormalization())

           model.add(Conv2D(filters = 400, kernel_size = (1, 1), strides = (1,1),
                            activation = 'relu', padding = 'same'))
           model.add(BatchNormalization())

           model.add(Conv2D(filters = 800, kernel_size = (3, 3), strides = (1,1),
                            activation = 'relu', padding = 'same'))
           model.add(BatchNormalization())

```

```
        model.add(Conv2D(filters = 400, kernel_size = (1, 1), strides = (1,1),
                         activation = 'relu', padding = 'same'))
        model.add(BatchNormalization())

        model.add(Conv2D(filters = 800, kernel_size = (3, 3), strides = (1,1),
                         activation = 'relu', padding = 'same'))
        model.add(BatchNormalization())

        # Output Layers

        model.add(Conv2D(filters = 13, kernel_size = (1, 1), strides = (1,1),
                         activation = 'relu', padding = 'same'))
        model.add(BatchNormalization())

        # Compiler

        model.compile(loss=BTBloss,
                  optimizer=keras.optimizers.Adam(),
                  metrics=['accuracy'])
        return model

    def train(Xfilename,Yfilename,model,run,epochs,batchsize):
        """
        trains the model given training data.
        """
        X = np.load(Xfilename)
        Y = np.load(Yfilename)
        print(np.shape(X))
        print(np.shape(Y))
        checkpointer = keras.callbacks.ModelCheckpoint(
                   run+'BTB{epoch:02d}.hdf5',save_weights_only=True)
        model.fit(X, Y, batch_size = batchsize, epochs = epochs,
              callbacks=[checkpointer])

    def loadmodel(filename, model):
        """
        loads saved weights into the model.
        """
        return model.load_weights(filename)

    def evaluate(Xfilename,Yfilename,model):
        """
        Evaluates the model returning the loss of a testing data set.
        """
        X = np.load(Xfilename)
        Y = np.load(Yfilename)
        print(np.shape(X))
        print(np.shape(Y))
        return model.evaluate(X,Y)

    #########################################################################

    def BTBloss(y_true,y_pred):
        """
        Defines a custom loss function for the model.  The loss function is
        identical to the YOLO loss function with the addition of an extra
        term
        relating to the rotation of the bounding boxes.
        """

    # Dimensions are all linear error.
    objMask0 = y_true[:,:,:,5]
    MaskShape0 = K.shape(objMask0)
    Zeros0 = K.zeros(MaskShape0)
    notMask0 = K.equal(y_true[:,:,:,5],Zeros0)
    notMask0 = K.cast(notMask0, 'float32')

    objMask1 = y_true[:,:,:,11]
    MaskShape1 = K.shape(objMask1)
    Zeros1 = K.zeros(MaskShape1)
    notMask1 = K.equal(y_true[:,:,:,11],Zeros1)
    notMask1 = K.cast(notMask1, 'float32')

    lamcoord = 5
    lamdims = 5
    lamtheta = 10
    lamobj = 0.5
    lamnoobj = 0.5
    lamp = 1.0

    coords = ( objMask0*(K.abs(y_true[:,:,:,2]-y_pred[:,:,:,2])
                        +K.abs(y_true[:,:,:,3]-y_pred[:,:,:,3]))+
               objMask1*(K.abs(y_true[:,:,:,8]-y_pred[:,:,:,8])
                        +K.abs(y_true[:,:,:,9]-y_pred[:,:,:,9])) )
    coords = K.sum(coords,1)
    coords = lamcoord*K.sum(coords,1)

    dims = ( objMask0*(K.abs(y_true[:,:,:,0]-y_pred[:,:,:,0])+
                       K.abs(y_true[:,:,:,1]-y_pred[:,:,:,1]))+
             objMask1*(K.abs(y_true[:,:,:,6]-y_pred[:,:,:,6])+
                       K.abs(y_true[:,:,:,7]-y_pred[:,:,:,7])) )
    dims = K.sum(dims,1)
    dims = lamdims*K.sum(dims,1)

    theta = ( objMask0*K.abs(y_true[:,:,:,4]-y_pred[:,:,:,4])+
              objMask1*K.abs(y_true[:,:,:,10]-y_pred[:,:,:,10]) )
    theta = K.sum(theta,1)
    theta = lamtheta*K.sum(theta,1)

    # The predictors are all quadratic clasification error.
    conf = ( objMask0*(y_true[:,:,:,5]-y_pred[:,:,:,5])**2+
             objMask1*(y_true[:,:,:,11]-y_pred[:,:,:,11])**2 )
    conf = K.sum(conf,1)
    conf = lamobj*K.sum(conf,1)

    cnot = ( notMask0*(y_true[:,:,:,5]-y_pred[:,:,:,5])**2+
             notMask1*(y_true[:,:,:,11]-y_pred[:,:,:,11])**2 )
    cnot = K.sum(cnot,1)
    cnot = lamnoobj*K.sum(cnot,1)

    p = (y_true[:,:,:,12]-y_pred[:,:,:,12])**2
    p = K.sum(p,1)
    p = lamp*K.sum(p,1)

    return (coords + dims + theta + conf + cnot + p)

########################################################################

def test(model,threshold,Z):
    """
    Provides a graphical based way of quickly checking the models
    performance.
    """
    Data = Utils.DataGrid(160,5)
    Xtrue,Ytrue = Data.createdata(Z,2)
    Ypred = model.predict(Xtrue)
    Ypred = Ypred[0]
    Ypredlist = []
    Ytrue = Ytrue[0]
    Ytruelist = []
    for i in Ypred:
        for j in i:
            k = list(j)
            out0 = k[:5]
            out0.append(k[5]*k[12])
            out1 = k[6:11]
            out1.append(k[11]*k[12])
            Ypredlist.append(out0)
            Ypredlist.append(out1)
    for i in Ytrue:
        for j in i:
            k = list(j)
            out0 = k[:5]
            out0.append(k[5]*k[12])
            out1 = k[6:11]
            out1.append(k[11]*k[12])
            Ytruelist.append(out0)
            Ytruelist.append(out1)
    for i in Ytruelist:
        if i[-1] > 0.5:
            print('w:'+str(round(i[0],3))+',  h:'+str(round(i[1],3))+
               ',  x:'+str(round(i[2],3))+',  y:'+str(round(i[3],3))+
               ',  t:'+str(round(i[4],3))+',  C:'+str(round(i[5],3))+
               '   Ground Truth')
            vecs = Data.boxverticies(i[0],i[1],i[2],i[3],i[4])
            Utils.DataPlot.plotbox(160,vecs,colour='b')
    for i in Ypredlist:
        if i[-1] > threshold:
            print('w:'+str(round(i[0],3))+',  h:'+str(round(i[1],3))+
               ',  x:'+str(round(i[2],3))+',  y:'+str(round(i[3],3))+
               ',  t:'+str(round(i[4],3))+',  C:'+str(round(i[5],3))+
               '   Predicted')
            vecs = Data.boxverticies(i[0],i[1],i[2],i[3],i[4])
            Utils.DataPlot.plotbox(160,vecs,colour='r')

#########################################################################

"""
Training and testing
"""


BTB = createmodel()

loadmodel('Version2NBTB10.hdf5',BTB)

train('X20K-5.npy','Y20K-5.npy',BTB,'Version2O',20,100)

results = evaluate('X1K-5','Y1K-5',BTB)

print(results)

test(BTB,0.5,[1])
```