Assessing the Potential Applications of Deep Learning in Design

Ranjeeth Mahankali

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Architecture

University of Washington

2018

Thesis Committee:

Brian R. Johnson, Chair

Alex T. Anderson

Program Authorized to Offer Degree:

Architecture

University of Washington

Abstract

Assessing the Potential Applications of Deep Learning in Design

Ranjeeth Mahankali

Chair of the Supervisory Committee:
Associate Professor Brian R. Johnson
Department of Architecture, College of Built Environments

The recent wave of developments and research in the field of deep learning and artificial intelligence is causing the border between the intuitive and deterministic domains to be redrawn. Amidst all the excitement surrounding this field, there are several prototypes being made, most of which are narrow, single purpose applications of deep learning technologies. This thesis takes a step back to establish a broader understanding of the new class of algorithms that deep learning offers. Beginning with the observation that architectural design workflow is often characterized by several representational transformations as projects grow in resolution and complexity, from sketching to detailed drawings or models, this research developed a series of deep learning prototypes that illustrate the potential application of this technology in the larger design workflow. This paper discusses the performance of these prototypes, identifies the challenges for integrating deep learning in practical design applications. This paper also suggests some ways in which these technologies might affect how the design process is carried out.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my thesis committee chair Prof. Brian Johnson, without whom this research would not have been possible. Starting almost 18 months ago, when I first started to get interested in this topic, Brian supported my ideas through independent studies and helped me turn them into a thesis. I really appreciate all his input and the intellectually stimulating conversations.

I would also like to thank my thesis committee member Prof. Alex Anderson, who helped me develop a broad appeal for the arguments I make in my thesis. Alex helped me realize several of these ideas in a concrete, written form during the research practicum course, helping me in organizing the ideas and addressing a wider audience.

I would also like to thank Dan Belcher, Nate Holland and Scott Crawford, who, through interviews helped me address some key ideas in this thesis.

Finally, I would like to thank my father Laxman Mahankali, who trusted me enough to take on a huge financial gamble in letting me pursue this degree.

DEDICATION

I dedicate this thesis and the ideas within, to a phenomenon – the eventual decline of mysticism in creative professions.

Chapter 1. INTRODUCTION

Design is a complex cognitive process preceding creation, during which a design proposition is generated, analyzed, queried, and adjusted. Questions are considered regarding the impacts, costs and performance of the proposed artifact. Some are subject to scientific investigation and can be represented through mathematical means. Others rely more on cultural sensitivity, memory, and judgement. Some remain ephemeral and effervescent, with answers arising from the habits and predilections of the designers. The design proposition itself is usually too complicated to be managed in human working memory, so designers rely on one or more external representations of the design proposal to capture and retain details. In addition to facilitating organization of the design, the external representation can be made accessible to other participants in the design process and becomes the central oracle for questions regarding the design.

Since the advent of computing technology, significant effort has gone into creating digital representations for design propositions on which computational processes can operate. Because many aspects of design propositions include a geometrical component, and because of the synergy between our understanding of computational processes and many engineering questions that arise in design, geometry capture and editing tools have tended to evolve most rapidly. Thus, computer programs have typically been deterministic, and the algorithms that drive them are essentially a set of instructions that get executed from start to finish to arrive at an output.

However, there is an alternative to this declarative/deterministic logic--associative logic. This is fundamentally different from deterministic logic and is typically difficult to explain fully with a rational approach. However, consider how a dog is trained to follow a verbal command. There is no rational explanation that connects the spoken word “sit” to the actual act of sitting. The spoken

command is just a sound, and it is different if the trainer speaks different language. The dog merely learns to associate that sound with the act of sitting without asking for an explanation. This association could have been reinforced by providing treats to the dog when it correctly obeyed the command. In the context of computer programs, this difference is very profound. If a computer can be programmed to learn by association, then it means that the person writing the program does not have to know the rational sequence of steps required to arrive at an output based on the given input. All that is needed are the input-output pairs that can then be used to train the computer and make it learn to generate outputs from inputs by association. Deep learning is essentially a mathematical technique that emulates associative learning. Deep neural networks can learn from input-output pairs by association, so it is not required to program in the exact logic of how to get from the input to the output. This lack of need to specify the explicitly the "how" of the task is what makes deep neural networks so attractive for tasks that are typically considered intuitive.

It must be noted that associations do not necessarily mean logic of the form "if this, then that". That would be direct comparison. Human designers use associations at various levels of abstraction when synthesizing new ideas (Kalay 2004). In the order of increasing abstraction, they could be precedents, symbols or metaphors. Associations made at a higher level of abstraction are more general and rich in nature. Abstract associations can remain relevant in new situations. Associations play an important role in creativity and intuition in general (Kalay 2004). For example, in character design for animated movies and video games, characters designed to be "scary" often have features inspired from carnivorous animals or taken from other elements of nature that are also "scary". These designers try to exploit the primal associations that we all are predisposed to make, to trigger those emotions in us with their designs. Several unquantifiable qualities of objects like cute, creepy, scary etc. are essentially associations. These associations can

be found in language and literature as well. For example, “Sunshine” and “Warmth” have positive associations in the English language, but they do not have the same associations in, say, a language spoken in parts South India with hot climate, where "stay cold" is a common greeting instead of "warm regards". The reason for this mismatch of notions is because English-speaking cultures originated in cold parts of the world where sunshine is welcomed. This is an important characteristic of associative logic. There is no absolute correct response or incorrect response, instead the correct response is whatever response you associate correctness with.

Deep neural networks, implemented on traditional computing hardware, mimic this behavior. In the case of a deep neural network, there is no absolute correct or incorrect output, the neural network produces an output based on whatever associations it learned from the data it was trained on. If the training data is a good representation of the data to which the network will be applied later, then the network will produce "correct" answers. This is the fundamental characteristic that differentiates deep learning from genetic algorithms and other optimization techniques in the context of design, and it is attractive precisely because it offers the potential of making headway on the more human and abstract aspects of design, where personal learning and preference and culture prevail -- the land of recognition, metaphor, and judgement.

Chapter 2. DEEP LEARNING - BACKGROUND

The Idea of creating computer programs inspired by the mechanisms of our own brain are not new and have been around for a long time (McCulloch and Pitts 1943). As covered in the literature review, the algorithms and techniques used to design and train these neural networks have been evolving over the last 70 years, but some basic concepts remain the same.

2.1 NEURAL NETWORKS IN A NUTSHELL

The figure shows a basic schematic of a neural network, where the circles represent the neurons and the arrows represent the weights of the neural network.

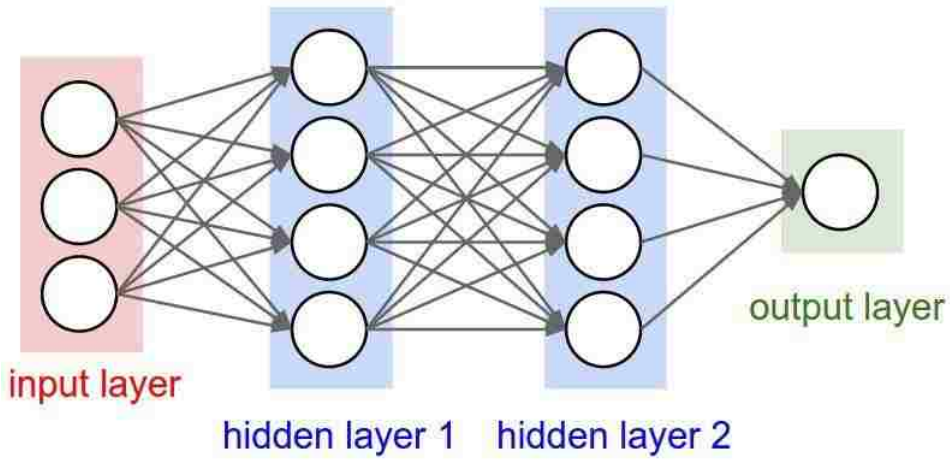

Figure 2.1: A Basic Neural Network

Each neuron receives a weighted sum of the incoming numbers from the previous layer of neurons, the weights used for the weighted sum being the arrows that connect the neurons. This weighted sum is then put through an activation function, whose output is then sent to the neurons in the next layer. Originally, the purpose of activation function was to decide whether a neuron "fires" or not. In these cases, the activation function would be a step function. If the output is higher than a

threshold the output is 1, else it is 0. But it has been observed that neural networks perform better if the activation functions are smoother rather than a step function. Some examples of such smooth activation functions are sigmoid and tanh. These two functions have an 's' shaped curve. With these smooth activation functions, the output of a neuron is not binary but rather continuous, so a neuron can fire weakly or strongly, or somewhere in between. Figure 2.2 is a plot of the sigmoid activation function.

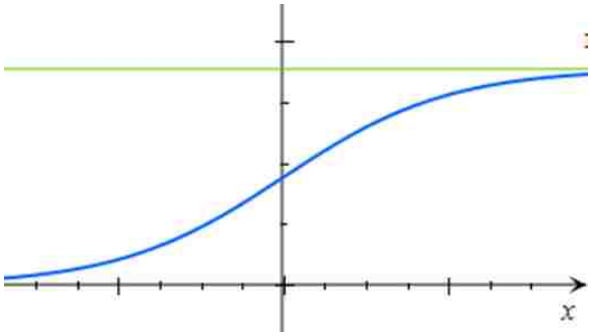

Figure 2.2: Sigmoid function - commonly used for activation

The sigmoid activation function, though continuous, tends to collapse to near 0 or near 1 values if the output is too high or too low. Rectified Linear Units function (Hinton and Nair 2010) has been found to yield much better training results. Modern deep learning libraries like TensorFlow, Caffe etc. have a readily usable collection of activation functions.

Activation Function Name	Activation Function Equation
Sigmoid	$y = \frac{1}{1 + e^{-x}}$
Tanh	$y = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$
Rectified Linear Units (ReLU)	$y = max(0, x)$
Leaky Rectified Linear Units (Leaky ReLU)	$y = max(\alpha x, x)$ for small values of $\alpha \approx 0.1$

Table 2.1: Some Common Activation Functions used in Deep Learning Models

2.2 FEED FORWARD

The mechanism by which the input is fed into a deep neural network and output is generated is known as feed forward. Using matrix multiplications and activation functions, it is possible to describe the feed forward mechanism for a simple neural network like the one in the Figure 2.1. In that neural network, the input layer has 3 neurons, hidden layer 1 has 4 neurons and there are 12 weights (3 times 4) connecting the input layer to hidden layer 1. The numbers in the input layer can be written as a matrix of the shape [1,3]. The 12 weights can be written as a matrix of shape [3,4] and when the input matrix is multiplied with the weight matrix, the resulting matrix will have a shape of [1,4]. These values are essentially weighted sums of the input values, so these 4 values correspond to the 4 neurons in the hidden layer-1. These 4 values are put through the activation function before repeating the process again for the next layer and so on until the output values are obtained, in this case, just one number. It must be noted that this example network has fully connected layers i.e. there is a weight connecting every neuron in a layer to every neuron in the next layer. So, if two consecutive layers in a network have 'm' and 'n' neurons then they have 'm x n' weights connecting them. Most typical deep neural networks have layers with several hundred or even thousands of neurons. Two layers with a thousand neurons each require a million weights to be 'fully connected', so this imposes computational challenges on training and using large networks. Convolutional Layers (discussed later in the paper) solve this problem by something called 'weight sharing'.

2.3 LOSS FUNCTIONS, LEARNING RATE AND BACKPROPAGATION

The feed forward method is the mechanism for obtaining output values of a neural network for a given set of inputs, but a neural network needs to be trained for the output to be meaningful. Neural networks are trained by backpropagation. Following are the steps in a training cycle:

1. Take the input-output example from the training dataset and pass the input data through the neural network using the feed forward mechanism. The outputs in the training set are sometimes called "targets" since that is what you would ideally expect from a trained model.
2. Once you have the outputs, calculate the loss function, which is essentially a measure of "error" between the outputs and the targets. The exact mathematical functions used to measure this error varies from case to case.
3. For each weight in your model, you calculate the partial derivative of the error with respect to the weight, which is also known as "gradient" and then use that to adjust that weight ever so slightly.
4. It is not good to apply the adjustments to the weights based on a single training example, so you scale these adjustments down by multiplying them with a small number known as "learning rate". For a good quality training process learning should be very small.
5. You repeat this process and go through the training set. It is common to have the training process make multiple passes through the training dataset.
6. Ideally, as the training progresses the accuracy of the network should increase the loss values decrease.

Calculating gradients for weights and adjusting them to minimize the loss function is known as "Gradient Descent Optimization", which is essentially a hill-climb algorithm applied parallelly to all the weights in the model. But there are several other optimization techniques (Adagrad, Adam etc.) which are improvements on the standard gradient descent, to increase the speed of training without losing the reliability. Modern deep learning frameworks have these algorithms built in for ease of use. Loss functions are a measure of the error, but the exact mathematical functions used

can affect how well the model is trained and how fast. A simple loss function would be to calculate the absolute difference between the targets and the outputs. RMS (Root Mean Squared) error is more commonly used than absolute difference. Table 2.2 shows some common loss functions.

Loss Function Name	Loss Function Equation (y is the expected output and y' is the predicted output)
RMS Loss	$\sum_i (y_i - y_i')^2$
Absolute difference	$\sum_i \lvert y_1 - y_i' \rvert^2$
Cross Entropy - when range is [0,1)	$-\sum_i y_i \, log(y_i')$

Table 2.2: Some Common Loss Functions used in Deep Learning

In most cases the output of a network is not a single number but a matrix with many numbers. So typically, the loss values are calculated element wise and then averaged or summed, the result of which is used to calculate gradients.

2.4 Quality of Training

A lot of factors affect the quality of the training process and the performance of the neural network. As discussed previously, there is no absolute measure for how intelligent or accurate a neural network is. The intelligence or accuracy of the neural network depends on two things: (a) What data is used to train the neural network? (b) Is that data representative of the real-world cases where you intend to put the neural network to use? For this reason, it is important to have a large and diverse dataset, at least as diverse as the intended use cases which you expect the neural network to perform on. For example, if you are building a model to recognize apples and all the apple images in your training datasets have the apples placed on wooden table tops, your model might learn to recognize apples by looking for the texture of wood. Your accuracy while training goes

up, but your model would fail to recognize apples in other situations. Larger and more diverse training set would solve this problem. Google's image recognition model is trained on millions of images.

Learning rate is another factor that can affect the quality of the training process. A large learning rate means that you are making large adjustments to your weights based on individual training examples. This is not good for the quality of training. You want the final model to be trained stably on the entire dataset. So, small learning rates are typically good. But if the learning rates are too small, though your training process is safe, it will take too long. So ideally, you want to have an optimum learning rate where your training speed is fast without compromising the quality of the training. There are several guidelines on how to choose the learning rates. These guidelines can be used as a starting point but the optimum learning rate for an individual case could vary greatly and can be figured out by some trial and error.

Overfitting is another issue that could potentially result in poor performance of a network. Typically, when training a model, a set of examples is kept aside for testing purposes. These examples are not used in the training process and is referred to as the "testing set" (There are other more advanced methods for organizing your datasets which are not discussed here). Because they are not used in the training process, they are essentially new for the network and if the network performs well on this set, then that means the training process was successful. The accuracy of the network on the training data is called the training accuracy and that on the testing data is called the testing accuracy. If the training accuracy is larger than the testing accuracy, then the model is considered overfit. This means that the model has learn to perform on the training data but did not learn the general patterns required to perform on new data. The model is fit to the specific quirks of the training dataset and is not generalized. This happens if your model is too large (too many

layers and too many neurons) compared to the complexity of the task it is trying to accomplish and ends up memorizing the data instead of learning generalized patterns from it. Measures to prevent this include decreasing the size of the model (number of layers and number of neurons), increasing the size and the diversity of the dataset, use dropout or regularization (beyond the scope of this paper). Dropout is an excellent technique to prevent overfitting. In this, during every training cycle, the activations of a randomly selected subset of neurons is set to zero. Without dropout, there is a chance that the network might learn to rely on few activations and reinforce only those over time. But with dropout, because neurons are being randomly turned off, the network cannot rely on few features, so it is forced to learn redundancies.

Chapter 3. LITERATURE REVIEW

The developments in this area of computer science were not continuous. The initial wave of excitement did not last due to lack of reliable ways to train a neural network. But there have been more waves of developments triggered by changes in technology.

3.1 THE SECOND WAVE

The second wave of developments in this field was triggered by the advent of backpropagation techniques to train the neural networks. With this new technique, they seemed to exhibit promising performances in the areas of speech and handwriting recognition (Rumelhart, Hinton and Williams 1986). The output values generated by neural networks trained during this period were binary since the improved activation functions did not come into the picture until much later. In addition to activation functions, there were also limitations in terms of the available computation power and training data.

Though neural networks at this time were very rudimentary, they showed some promise. The strong relation between how neural networks function and associative reasoning was identified early on (R. Coyne 1990). Coyne draws a contrast between (1) HyperCard system of storing discreet records and searching through them and (2) storing abstracted patterns in a trained neural network to demonstrate the strong connection between how neural networks work and associative reasoning. These different ways of storing data and searching through the data can be organized on a spectrum ranging from literal to abstract. The HyperCard system that Coyne discusses is a literal one, but he also points out how the layered structure of neural networks relates to various levels of abstractions. The first input layer receives the most literal form of information, and the subsequent layers receive increasingly abstracted forms of information. A trained network would

have weights at each of these layers that are designed to handled information at that level of abstraction. Coyne further points out that the number of literal bytes required to store a neural network is constant independent of how much data it was trained on, unlike a HyperCard like system, whose size would increase with increase in number of cards stored. This is possible through, again abstraction. Because, neural networks find patterns that are common between the examples on which it was trained on, it does not need to remember each of the examples separately. It can be argued that abstraction is a form of lossy compression which is driving a large body of modern research into using deep learning as a way of compressing information. Coyne also points out more similarities between how neural networks work and how human brains work, like how the information is not only abstracted, but is also decentralized. Destruction of a small number of weights of a neural network tends to uniformly decrease the performance of the neural network rather than affect a single aspect of its performance. Coyne suggests parallels between this and macro level cognitive behavior in humans. Another similarity explains the phenomenon called standardizing, or overregularizing (in the context of modern deep learning, this is known as overfitting), where a neural network trained on a narrow dataset assumes the patterns that it learned across a more diverse dataset and hence suffers from decreased accuracy. Coyne draws a parallel between overregularizing and an experiment (Rumelhart and McCIelIand 1986) that showed the tendency of children to overregularize when learning past tenses of verbs. This paper also goes on to use a rudimentary neural network on binary bitmaps, to train them on a small set of bitmaps and then have them tweak new, arbitrary bitmaps to make them resemble the ones that they were trained on.

During this period, neural networks were also called Parallel Distributed Computing Systems (PDP Systems), because of some hardware configurations that were thought to be the best way to emulate

the functionality of a neural network. There were also other attempts to exploit the power of neural networks and connect them to design problems (Petrovic 1996) taking a different approach by treating individual nodes in the network as agents, and the weights as a collection of interactions between these agents. It is unclear if these differences are only semantic. This program works with a 3 x 3 grid of size 4.2m, replicated in 2 floors and an attached 3d visualizer visualizes the output of the neural network as a 3d model. This uses a "semantic differential matrix" as input, which essentially tries to quantify subjective characteristics of the design. Though they showed some promise, the performance of these prototypes was seriously bottlenecked by computing power and the size of the training dataset.

The potential of neural networks (or PDP systems as they were referred to at the time) in analogical and associative reasoning, especially where the problems were ill-defined, as they often are in design, was clearly identified. Despite that, the practical applications of these prototypes did not persist and the consensus of scholars in architecture fell back to skepticism by the late 90s. Even in speech and handwriting recognition, the neural networks could not live up to their theoretical promise. This was mainly due to lack of training data and processing power, even though a lot of advances were made in training techniques. This meant that the excitement surrounding this field within the architecture discipline was also downhill by early 2000s. This was reflected in the consensus of the period among the scholars in architecture and neural networks were seen mostly as pattern recognizers that had limitations with no feasible potential in the foreseeable future (Kalay 2004).

3.2 THE THIRD WAVE

The last five years have seen a new wave of developments and excitement. This wave is fueled by several developments. Between the early 2000s and the present, there has been an explosion of data that came with the internet and the ubiquity of computers and other devices. Thanks to the gaming industry, computing power became more inexpensive and compact, especially GPUs which are better suited to do massive numbers of parallel computations, something that neural networks could take advantage of. There have been successful efforts to create large reliable labelled datasets through crowd sourcing, ImageNet (Deng, et al. 2009) being the best example of this. Better activation functions were developed (Hinton and Nair 2010). Techniques such as dropout (Srivastava, et al. 2014) were developed which greatly improved the quality of training. All these developments were the setup which led to the breakthrough performance of neural networks on image recognition tasks (Krizhevsky, Sutskever and Hinton 2012), outdoing every other machine learning model. This triggered the wave of development over the last five years. The growth has been explosive, attracting interest from places far beyond academia. Today we have neural networks that can predict the future frames in a video, generate text captions describing images, generate images from verbal descriptions… the list does not end here, and new breakthrough developments have been surfacing every few months. These developments compel us to rethink our stance on intuition and creativity and redraw the line that separates what machines can and cannot do. Of course, even the best neural networks we have are nowhere near general intelligence and probably more rudimentary than the brains of worms. But there is plenty of evidence to suggest that, though at a very rudimentary level, these algorithms operate in the same domain as that of human intelligence.

3.2.1 Recurrent Neural Networks

The size of the input for a given deep neural network is fixed, for example the neural network in Figure 2.2 has 3 input nodes so it can accept three numbers as input. But depending on what the application is, it might be necessary for a neural network to be able to process inputs of varying sizes. For example, a network intended to translate sentences from one language to another would be expected to accept sentences of varying length as input. Recurrent neural networks are used to accomplish this. The variable length input is broken down into smaller chunks of equal size and they are fed into the network one by one, but the RNN can process the input as a whole because of its structure. In an RNN, output of the last hidden layer is plugged back into the input of the first hidden layer, to be combined with the chunk of input coming next. This essentially allows the network to, if trained to do so, retain relevant information from the past. The feed forward and backpropagation processes in recurrent networks essentially happen through time. Because they are compatible with input of variable size, most neural networks in natural language processing use recurrent neural networks.

3.2.2 Convolutional Layers

Recognizing objects in images is a popular application for deep learning. One way to do it would be to interpret the all the pixel values as individual input neurons. Fully connected layers can become computationally heavy as the number of neurons increase. This can be solved by sharing a small number of weights across many connections. This is done for images using kernel filters. Applying kernel filters to images is essentially calculating weighted sums of pixel values. But because the same kernel is moved around the image, the same weights are being shared across the image. This is what a convolutional layer does. Convolutional layers not only make the computations lighter, but they perform better for recognizing objects in images. Because the same

kernel with same weights is being moved over the image, the patterns that the networks learns are not dependent on spatial location. This means that a convolutional neural network (CNN) trained to recognize cats will recognize a cat no matter which part of the image it is in. This property of convolutional neural networks is known as "Translational Invariance". All layers in a convolutional neural network do not have to be convolutional layers. Most CNNs make use of a few fully connected layers along with the convolutional layers. These networks typically accept images (i.e. normalized pixel values) as input and output a vector, which could be for object recognition or some other task.

3.2.3 Autoencoders, Generative Models and Embeddings

Autoencoders came from efforts to use neural networks for data compression. They consist of an encoder which compresses the input data by mapping it to a vector (smaller in size than the original data), and a decoder which takes this vector and regenerates the original data from it. These two parts of the autoencoder are typically trained together as one composite neural network. The decoder is essentially a generative model because it is up-sampling the data. This idea can be applied to convolutional neural networks to generate images. This idea is fully exploited in generative adversarial networks (GAN), which can generate photorealistic images.

GANs are made of two neural networks. One is a convolutional network known as the discriminator, and the other is a deconvolutional network known as the generator. The generator, as the name suggests, generates images, which before training are of course just noise. The discriminator takes an image as input and outputs the probability that the image is from the training set and not generated by the generator. The loss functions for these two networks are defined separately. The discriminator is penalized every time it guesses wrong, i.e. every time it is fooled by the generator (or if it guesses real images as generated). The generator is penalized every time

it fails to fool the discriminator. These two networks are then trained together, and they both get better together. The discriminator keeps getting better at identifying fake images and generator keeps learning ways to fool discriminator until you have a generator that is generating images that are near photo realistic.

Autoencoders and generative models do not require input-output pairs and they can instead learn patterns from a corpus of data. This type of training is broadly classified as unsupervised learning. Embeddings is another form of training that falls under the umbrella term of unsupervised learning. Training embeddings is a technique that can be used to create more meaningful representations of human readable data for computers, which preserves the semantic understanding that humans see in the original data. A trained set of embeddings can then be used as a lookup table to map human readable data to a form where the semantic aspects of the data are accessible to a computer, which in turn can be used as an input for another application. For example, Word2Vec (Mikolov, et al. 2013) is a set of word embeddings that is trained on large bodies of text. It is essentially a dictionary that maps every word (in a large set of words) to a vector. Each of those vectors represents a point in an n-dimensional vector space and they are referred to as the embeddings for the words. But at the beginning, the embeddings are randomly initialized. The training process tries to minimize the distance between the points corresponding to the words that often appear together in the body of text on which the model is being trained on. So, after the training is finished, the embedding vectors corresponding to words that are closely related are also spatially closer to each other. A trained set can display characteristics such as the difference between the vectors corresponding to the words “king” and “man” being the same as the difference between the vectors corresponding to the words “queen” and “woman”. This is one example of several semantic relationships between words that

can be "embedded" in these vectors. These vectors can then be used as a lookup table for any other text processing deep learning model like translation, chat bots etc.

3.3 Third Wave Applications: Deep Learning

In 2012 (Krizhevsky, Sutskever and Hinton 2012), a convolutional neural network which implements dropout, outperformed every other machine learning model with a margin of over ten percent. This network was named "Alex Net". This was trained on the labelled image dataset known as the ImageNet and could classify an image across 1000 different labels. In the following years, Google went on to develop their own deep convolutional neural network which performed even better on the same dataset. Google called its image recognition network "Inception" (Szegedy, Liu, et al. 2014). The 'top-5 error rate' (i.e. the correct label for the image is not in the top 5 guesses from the network) for the 3rd incremental improvement of Inception (Inception v3) reaches 3.46%, which already beats humans at the same task (top 5 error rate of 5.1%). Since then the Inception model was made open for anyone to download and use for their own classification tasks. This works by retraining the top layer of the Inception model. Since the Inception model was already trained to map images to 1000 different labels, the lower levels of the model are trained to identify edges, shapes, textures and other high-level features of an image. This ability can be used and applied to your own dataset and your own set of labels just by training the final layer of Inception. This is called "transfer learning".

The issue that is often used in arguments against deep learning is that they are essentially black boxes with no way to clearly understand what is going on inside them when they are recognizing images (or other tasks). Engineers at Google, to open this black box, tried to visualize the data in the intermediate layers of the image recognition network. These attempts revealed visualizations

that are very similar to the descriptions of hallucinations that humans recount after being under the influence of hallucinogenic substances. This suggested that though very rudimentary, these models operate in the same domain as that of the visual processing in the human brain. Because the exact mechanisms inside a deep neural network are still mostly a mystery, it is not possible to definitively prove these claims.

Figure 3.1: Images Generated by a GAN from a provided Verbal Description

Generative adversarial networks were first proposed in 2014 (Goodfellow, et al. 2014). They caused many ripples in the deep learning community because most other deep learning models were essentially interpreters which generated data, but GANs can generate data. A Deconvolutional GAN trained on a captioned image dataset could learn the visual concepts behind the words used in the captions of the images in the dataset. This model was then able to generate new photorealistic images based on verbal descriptions. These images are not in the training set. It could generate the image of a flower with "big droopy yellow petals with burgundy streaks and a yellow stigma". In theory, the network could learn the visual representations of concepts behind words "yellow" or "petals" etc. and was able to put together a totally new image. There is no

definitive way to confirm this because of the "black box" nature of the deep learning models. The "black box" nature of the deep learning models did not stop them from being improved dramatically within a span of few years and being deployed in applications. There are mobile apps that are available today which can take the self portrait of the user and then manipulate it to make them smile, or other variations in expressions, hair etc. The results produced by these algorithms are comparable to the photo editing and manipulating skills of a novice human editor.

As previously discussed, Word2Vec is a popular word embedding model, and such embeddings are being exploited to build several deep learning models which are categorized as "natural language processing". There are also attempts to apply the same concept of embeddings to different sounds instead of words, and train them on hours and hours of audio. There are other generative models which can generate speech that sounds completely natural, as opposed to other speech synthesis algorithms which essentially assemble the sound from pieces. “Wave Net” is one such model, and its audio is completely continuous and is undistinguishable from natural speech. Similar technologies are being deployed in sound editing software, which can learn a person's voice from a 20-minute audio clip of that person's speech and then read any sentence in that person's voice, and it sounds undistinguishably natural (Adobe Voco - Wikipedia n.d.). This is not the end of the list of various deep learning models. The breakthroughs in this area are happening so fast and so frequently that they are very inconvenient for academic documentation.

In the light of these rapid developments, it is necessary to re-assess the potential of neural networks in architecture and design, to anticipate and adapt to the changes that AI might bring up on this discipline and to exploit it to make better software tools. One clear observation that can be made from all these breakthroughs is that, in the last five years, they expanded more horizontally than vertically. There is a lot of energy and excitement directed towards realizing and prototyping the

breadth of possible applications for deep learning. Even when they are deployed in applications, most of them are in the form of virtual assistants, intelligent searching, fun mobile apps. It is difficult to predict how long it will take for these techniques to enter more specialized fields like Architecture. That would require a collaboration between experts in both areas.

3.4 The Mathematics of Associations

Given the close relation between capacity for association and creativity (R. Coyne 1990), together with the fact that deep learning exposes mathematical techniques that emulate human-like association, a broad understanding of these mathematical techniques, their meaning and scope is essential to assess the potential of these tools in design workflows.

3.4.1 High Dimensional Data and Flattening

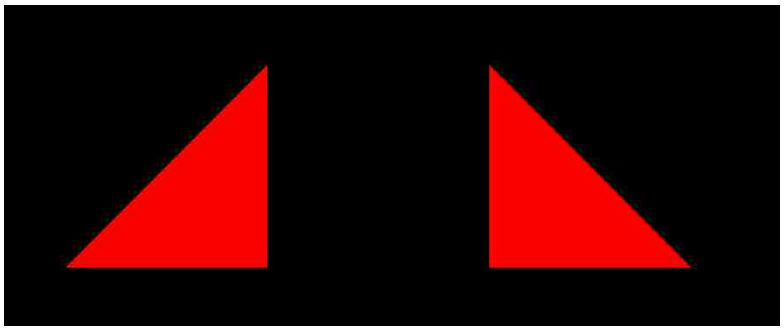

Figure 3.2: Left: Right pixel darker; Right: Both pixels together not brighter than 255

To understand what happens to the data as it flows through the layers and weights of a neural network, it is important to understand what high dimensional data is and how can we visualize it in a way that makes it more intuitive for humans. Consider a grayscale image of size 2x1, which has only 2 pixels, each containing a value in the range [0, 255]. If the value in the first pixel were to be interpreted as ‘x’ and the second pixel as ‘y’ the point (x, y) can be plotted on a 2D plane. So, every possible image of this size can be represented by a point on this plane. Now, consider

the subset of images which qualify the filter statement: “The left pixel should always be brighter than the right pixel”, the red triangle in the left plot of Figure 3.2 shows the area in which all such images exist. Similarly, the plot on the right shows all the images where the two pixels together are not brighter than 255. Now consider another grayscale image of size 3x1. Now we have a third pixel whose value can be interpreted as ‘z’ and point (x, y, z) can be plotted in 3D space. Every possible grayscale image of size 3x1 can be represented as a point in this 3D space. This way of looking at images as points in high dimensional space and collections of images lying inside boundaries can be extended to larger resolution images as well. If an image is of size 100 x 100, then every possible image of that resolution, can be represented as a single point in a 10,000-dimensional hyperspace. Images that are very similar to each other would correspond to points that are close to each other in this hyperspace. As the input data (images) pass through the layers of the neural network, the dimensionality of the data (length of the vector) can change. So, passing the data through a trained network can be a way of reducing the dimensionality of the data, making it easier to visualize. Mathematical techniques like t-SNE (Maaten and Hinton 2008) can further improve the visualization of high dimensional data by reducing the dimensionality to 2 or 3, by “unrolling” the data. A good analogy to understand how this unrolling works is map projections. If one were to take the globe and project a shadow of all the land masses, it would result in a very confusing 2d representation, and would not be very useful. But most types of maps work by, in some way, unrolling the surface of the globe. This might not preserve the precise shapes in the high dimensional hyperspace, but this preserves proximity and spatial relations, while reducing the high dimensional data into an interpretable form for humans. T-SNE works in a very similar way (Visualizing with t-SNE 2015). MNIST is a dataset of hand written digits which is a commonly used as a benchmark in deep learning. In this dataset, each image is of 28 x 28 pixel

resolution, and contains a hand-written digit. This dataset also contains labels corresponding to each image, which indicate the digit the image contains, these labels are used to train an NN. Given the resolution of the input images, they have 784 pixels each, which means that each MNIST image can be represented as a point in the 784-dimensional space. Figure 3.3 shows one such plot of the MNIST data, flattened to 2 dimensions.

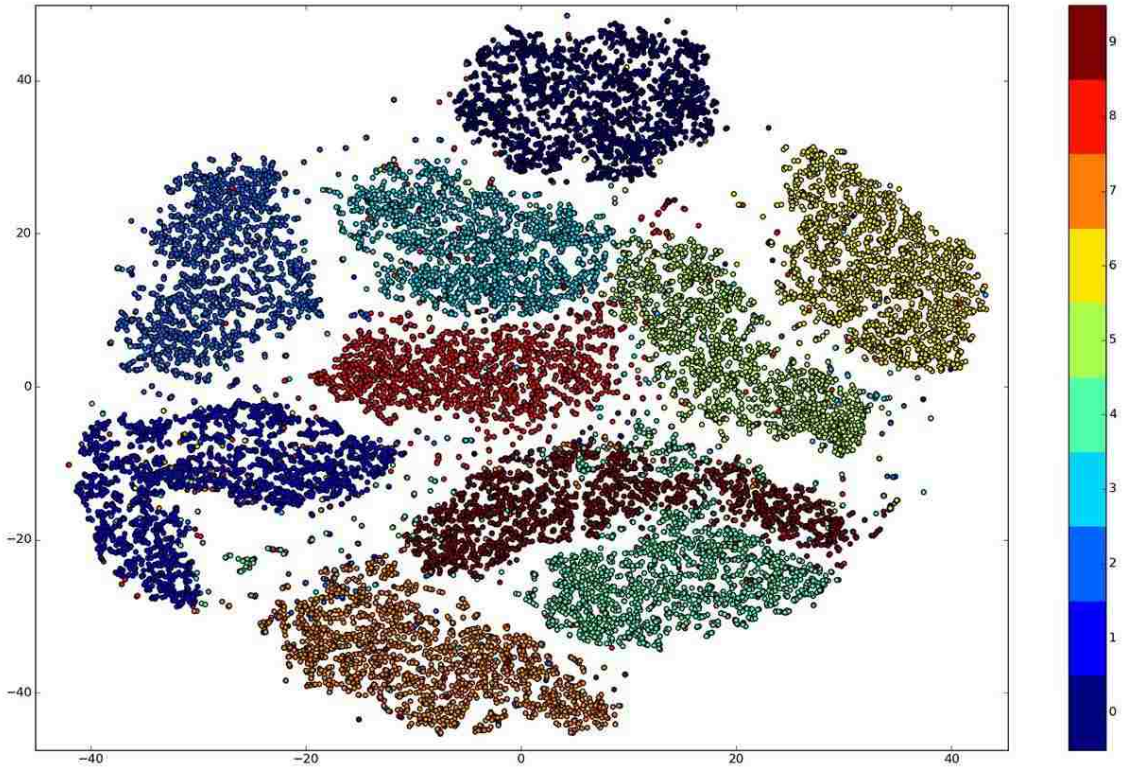

Figure 3.3: MNIST data plotted using t-SNE (Visualizing with t-SNE 2015)

The plot clearly shows that the images corresponding to the same digit form a clear ‘shape’ in the hyperspace. When a neural network is trained on this dataset, the weights in the neural network learn these shapes to distinguish between the digits. The weights of a neural network correlate to these shapes in a way analogous to how the coefficients of a polynomial correlate to the shape of a plot. For example, the green dots corresponding to digit ‘4’ bleed over onto the brown dots

corresponding to digit ‘9’, and it is understandable because in some cases, handwritten 4 and 9 can be similar.

Figure 3.4: 4s and 9s sampled from MNIST dataset

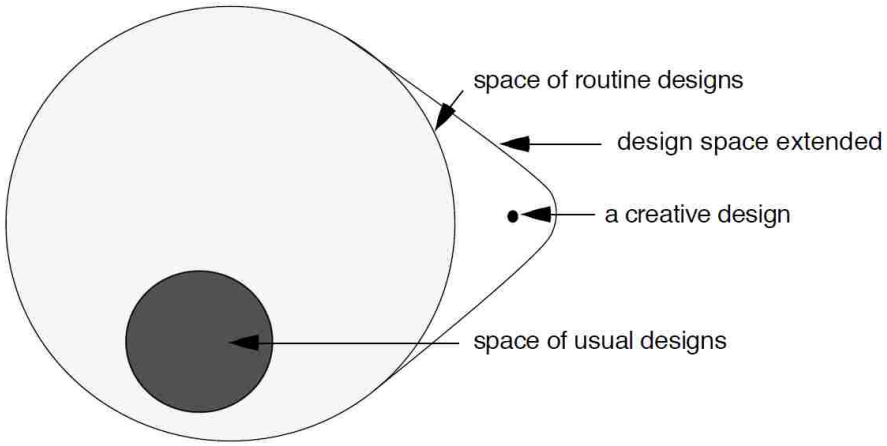

Figure 3.5: Designs existing in state spaces (Gero and Maher 1991)

Given the weights of a neural network that ‘learned’ these shapes in hyperspace, they can be used to sample a point in that hyperspace, which lies in the distribution corresponding to, say, digit ‘5’ that does not exist in the training data. Essentially the network is writing a new digit ‘5’. This is what GANs do. They learn the distributions in the training data and use the learned weights to generate new samples in the same distribution. This technique can be used to do more interesting things, like sampling the distributions to generate data with specific qualities. For example, in the MNIST data, we can sample a point in the hyperspace that is somewhere between the two digits – resulting in an ambiguous looking handwritten digit. Or, we can generate, say, a digit ‘5’ which

maximizes it's quintessential five-ness. In the hyperspace, this would correspond to a point in the '5' cloud, that is farthest from all other digit clouds.

This idea of hyperspaces, distributions and sampling these distributions can be extended to other datasets, like faces, animals etc., and these techniques can be used to generate images that do not exist, like some of the projects discussed in Section 3.3. Though these domains are simple and rudimentary, these mathematical tools give us a way to understand creative processes and how new images (and other forms of data) can be "created" by sampling from higher dimensional distributions. If a neural network can learn the shapes in the hyperspace from the training data and generate new data that stretches the boundary of these high dimensional spaces, that network would be emulating a creative process, albeit a very rudimentary one. Even though the field of computer science arrived at this notion from a mathematical point of view, this is not so far from how designers think about creative processes. They are connected at least at a metaphoric level. The notion of designs existing in 'State Spaces' (Gero and Maher 1991) is analogous to the idea of hyperspace and stretching the boundaries of these spaces with creativity is analogous to how networks sample distributions. At present we are limited to rudimentary datasets because of the computational requirements to do the higher dimensional math. Compute power is one challenge that needs to be overcome to apply these techniques to practical design applications, but there are other major challenges related to data formats.

The examples used to demonstrate these concepts are often related to image datasets. Images occupy the distinct position of being visual and yet being homogeneous in terms of how the data is stored. The same cannot be said about the design representations used in other fields of design, specially architecture, engineering and construction (AEC). This issue of homogeneous data is discussed in much greater detail in the later in Section 5.4.

Chapter 4. PROTOTYPE DEEP NEURAL NETWORKS

Though it's possible to find broad similarities between the associative capabilities of a neural network and that of human designers, a narrow and tractable problem in this domain is needed to test this idea. One task that almost all design workflows involve, and one that relies heavily on associative learning is the translation of design from one representation to another. At any stage of a design process, there is a representation at hand the designer is using. And as the design process progresses, the design moves from one representation to another, often gaining more and more detail as the designer continues to make decisions. The representations at the beginning of a design process are often very abstract and contain limited detail, this allows the designer to explore options and iterate through them efficiently. But as the designer makes decisions, and adds detail to the design, the design moves to representations that are more concrete.

Figure 4.1: A Sketch that requires learned conventions to interpret spatially

The different levels of abstractions in each representation make them non-equivalent; in other words, it is not possible to have a functional one-to-one mapping between any two representations. That is what makes it challenging for computers to translate designs from one representation to another. So how do the human designers do it? The sketch in Figure 4.1 shows how humans rely on associations to perform these translations. To a human looking at this sketch, it is very easy to visualize the 3-dimensional composition that it represents. But there no possible one to one mapping between a set of 2d positions (in the sketch) and 3d positions (scene visualized by the humans looking at the sketch), to overcome this obstacle; humans rely on associations. People are likely to look at the white, shaded circle on the top right and think that it is the moon, but in theory it could just as easily be an oddly shaded ping pong ball, floating close to the camera. But people do not typically think like that because they are not used seeing ping pong balls floating arbitrarily, that is not an association that they normally form. These associations are so ubiquitous in design processes that we usually go unnoticed. Of course, this is a general example and similar associative reasoning can be found in other places, for example, a designer might use a very specific style of sketching making his/her sketches interpretable only by their assistants who have learned these representations by experience and formed the required associations.

The rest of this chapter will discuss new prototypes which were developed as part of this research, intended to show that deep neural networks can learn the associations necessary to translate designs from one representation to another.

4.1 Generating Voxel Maps with SketchTo3D

The ability to interpret 2D sketches in terms of 3D geometry is an essential design skill and has been the subject of analytical approaches. This is distinct from the use of 2D UI elements such as

construction planes to render screen input meaningful in 3D modeling, as it requires interpretation of a finished 2D drawing without reference to existing 3D data. SketchTo3D is a deep neural network that demonstrates the ability of neural networks to understand spatial information from 2D representations. As demonstrated in the above discussion, this task involves translating information between two non-equivalent representations and requires use of learned conventions.

The input images for the network are grayscale bitmaps containing wireframe diagrams (perspective view) of cuboids of random sizes (1 or 2 in number) placed randomly in a scene. The network outputs a voxel map of the scene. A voxel map in this context is the volume of the scene divided into small cells, where each cell is represented either a '0' for an empty cell or a '1' for an occupied cell. The performance of the network is measured based on how closely the generated voxel map approximates the real voxel map corresponding to the actual scene shown in the input bitmap.

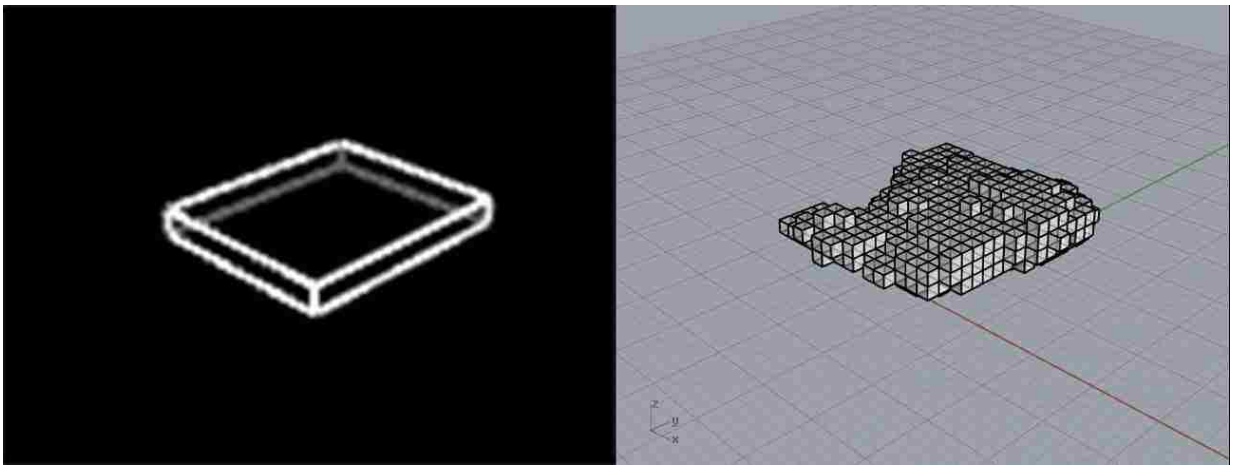

Figure 4.2: Voxel map (right) generated by the NN based on the input image (left)

The network was composed of two fully connected initial layers, followed by two convolutional layers, followed by two deconvolutional layers, which generate the voxel map as the output. This

network achieved 93.5% accuracy. Figure 4.2 shows one input-output pair (from a randomly sampled collection) and nine more such pairs sampled randomly from the test-dataset can be found in Appendix A. The generated voxel maps showed reasonable understanding of proportions of the boxes in the scene and their position, but they all have very poorly defined edges and often do not resemble cuboids.

So, the performance was just satisfactory, and it is understandable that the network could not learn to interpret edges to guess the volume of the scene because that is a very analytical task. As previously discussed, NNs are good at forming associations and not reasoning analytically, so this network needs to be tested on a different dataset, one that relies less on analytical reasoning than this example. So, as a second attempt, I created another dataset, in which each scene contains a sphere. The sphere can be on the ground or offset above the ground by some distance. In scenes where the sphere is off the ground, there is a thin line that is drawn under the sphere. This can be thought of as a representation convention, where the sphere with a line represents the volume of the sphere being off the ground, like a tree maybe and the absence of the line represents an object on the ground. When trained on this dataset (named the "ball dataset"), the same NN learned this convention and achieved 99.3% accuracy. The measurement of accuracy is the same for both datasets, by counting the number of voxels predicted correctly. The Figure 4.3 shows two examples from the test dataset. Both input images have similarly located spheres, but the second image has a line below the sphere. In the side elevations of the generated voxel maps, the first example volume is placed on the XY plane, whereas the second example volume is off the XY plane. For more details about this prototype, see Appendix A.

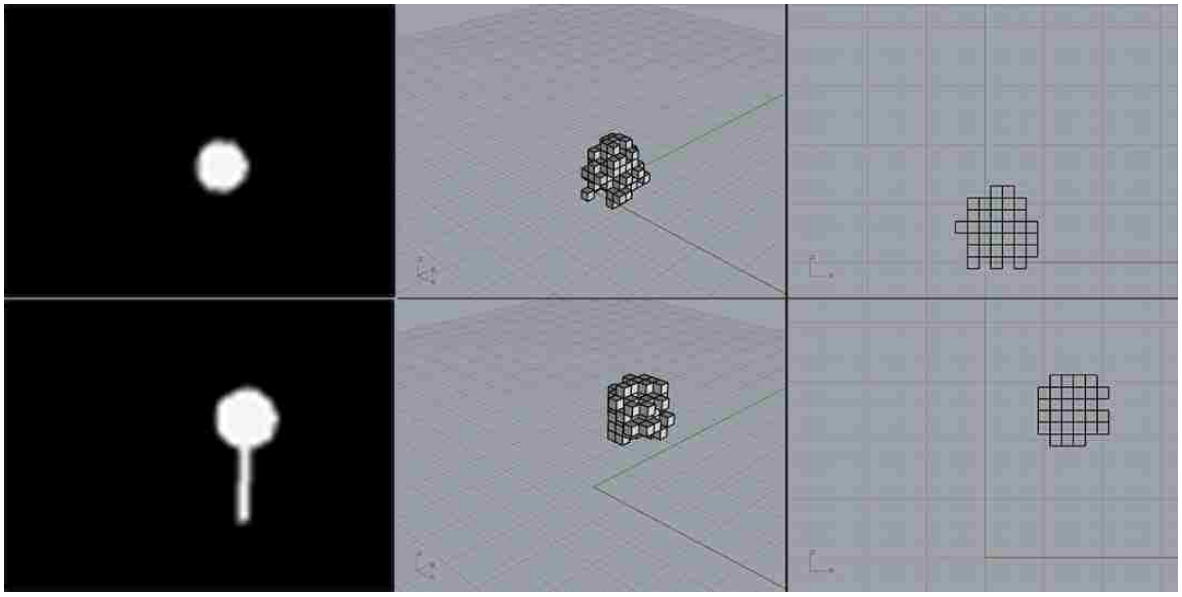

Figure 4.3: Ball dataset examples sampled from the test set

4.1.1 Creating the Dataset

The initial cuboid dataset was created using Rhinoceros 5, with a Python script. The ‘scene’ in this context refers to a fixed volume – a cuboid. The script then populates the scene with either one or two boxes, randomly sized. The script then captures a snapshot of the perspective view and then the scene is divided into a grid of 24 x 24 x 16 and each cell in this grid is assigned a value of either 0 or 1 depending on whether its empty or occupied, these data are essentially the voxel map, which is saved as an array. The perspective view snapshot (the image) and the voxel map (the array) are then saved as a training example pair. The script repeats this process to generate a large dataset, in this case 5000 instances, with an additional test set of 500, used only for testing and not training.

The second training attempt, used a different dataset named “ball dataset”. This was also generated in Rhino using a Python script. This time, each scene consisted a sphere, positioned randomly in plan and then positioned either on the XY plane (Z = 0) or off the XY plane with a random offset.

When the sphere was placed off the XY plane, the script added a vertical line to the scene below the sphere, to represent its distance above the XY plane. Perspective views and the voxel maps were captured in the same way as the cuboid dataset before.

4.1.2 The Special Loss Function

As explained briefly before, the voxel map corresponding to a scene consists of 9216 voxels (24 x 24 x 16). The accuracy of the network is measured as the percentage of the 9216 voxels which are predicted correctly as occupied or empty. The loss function against which the network was trained also uses a similar logic, counting the number of voxels whose predictions are incorrect. This loss function makes sense on the paper, but the first training attempts resulted in problems.

Equation	Explanation
$sig(x) = \frac{1}{1+e^{-x}}\ ;\ f = sig(v_p - v_t)\ ;$	Sig is a normal sigmoid function, v_p and v_t are the number of occupied voxels in the prediction and in the true voxel map.
$L = f \cdot L_e + (1-f) \cdot L_o;$	L_e & L_o are the losses measured for the empty and occupied voxels separately.

Table 4.1: The equations used for the Special Loss Function

Consider a scene with a 6x6x3 box, which would have 108 occupied voxels of the 9216 total voxels. Because the volume of the scene is a degree 3 function of the size of the boxes, the percentage of occupied voxels is often very small. During the first training attempts the network learned quickly to predict all the cells in the voxel map as empty, and still get an average accuracy of 90%. This results in the network becoming trapped in a local minimum (of loss) where it predicts empty voxels for all input images, as a "safe guess" and does not learn the spatial relations. The problem is with the way loss and accuracy functions were defined, where every empty scene is rewarded with 90% accuracy. So, a special loss function was required, which would not reward

the empty voxel output with a 90% accuracy. A special loss function was implemented in the later attempts, shown by the equations in Table 4.1. Instead of measuring the loss together for all voxels, the loss for empty voxels and the loss for occupied voxels is measured separately as two quantities. The loss for empty voxels (L_e) would account for all the empty voxels which are predicted incorrectly as occupied. The loss for occupied voxels (L_o) would account for all the occupied voxels which are predicted incorrectly as empty. If these two losses were to be added, it would result in the same loss function that was used in the first training attempt. But instead of adding them together directly, the special loss function introduces a bias (f). As evident from the equations in Table 4.1, this bias will be greater than 0.5 if the predicted map consists of more occupied voxels than the true map and it will be less than 0.5 if the predicted map consists of fewer occupied voxels than the true map. Since the bias is calculated with a sigmoid function, as the difference between the number of occupied voxels in the predicted and true voxel maps increases, the bias quickly approaches either 0 or 1. The equation for the special loss function (L) takes advantage of this bias, to selectively penalize the network. When the network predicts fewer occupied pixels, it gets penalized heavily for the occupied voxels predicted as empty (L_o) and not so heavily for the empty voxels predicted as occupied (L_e). This constantly changing loss function prevents the network from getting trapped in a local minimum. The final version of the network used this loss function and worked well in both the box dataset and the ball dataset. The version of the special loss function implemented in the code was slightly different, where the input of the sigmoid function (to calculate the bias) was scaled to make the transition of the sigmoid around 0 more sudden, making it more like a step function, but the idea behind the special loss function remains the same.

4.1.3 Discussion

The performance of the network was excellent for the ball dataset, because all it had to do in that case was to learn a convention in the 2D representations and convert that into a spatial interpretation. But the results on the box dataset were not as good. The network learned to interpret the location and the volume of the boxes in the scene, but it did not learn to interpret the shapes and edges. When humans interpret edges to infer spatial information from them, they are basing it on geometric primitives that they are familiar with and are surprisingly less open to other possible spatial configuration that can result in the same 2D view. This bias is what leads to illusions like the Ames Room (Ames 1952). It seems likely that neural networks might perform better if they must interpret a 2D scene as a composition of objects that are predefined, like furniture, which eliminates the load of guessing the exact shape of these objects and interpreting their edges.

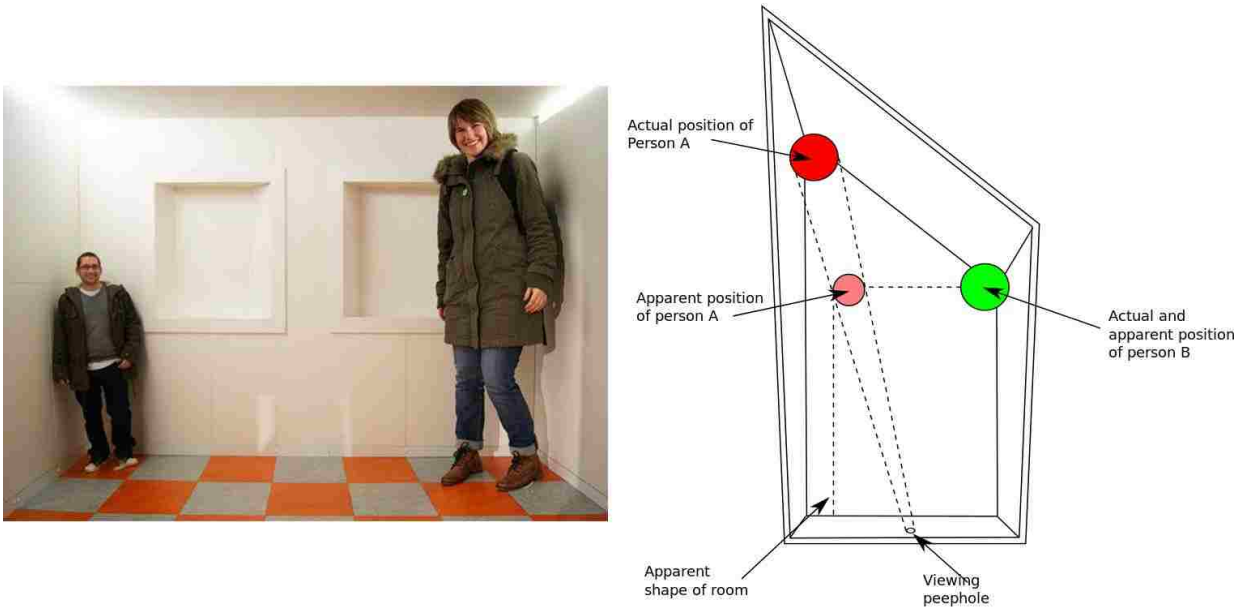

Figure 4.4: Left: Ames room illusion; Right: explanation of the Ames room illusion

4.1.4 Compose3D

The Ames Room illusion proves that humans are not really interpreting the exact geometric information in a sketch, but rather recognizing the familiar objects in the sketch. So, to see if a convolutional neural network can do the same, a modified version of SketchTo3D network, titled Compose3D was made and trained on a new dataset. In this dataset, each scene had a cube (with a circle drawn on one corner to break symmetry) was randomly positioned and oriented on a horizontal rectangular plane (floor plane). Compose3D was trained to recognize the object's position and orientation from the perspective view of the scene. The network is not actually trying to predict the geometry of the object; the geometry of the object is provided to the network, which predicts the position and orientation of the object and uses that information to place the provided geometry to approximate the scene shown in the sketch. To measure the accuracy of the network, the predicted position and the orientation of the object were considered correct if they were within a certain tolerance of the original values. With that accuracy metric, the accuracy of the network was 85.2% with tolerance set to 5% of the size of the floor plane of the scene. When the tolerance was changed to 10%, the accuracy was 92.4%.

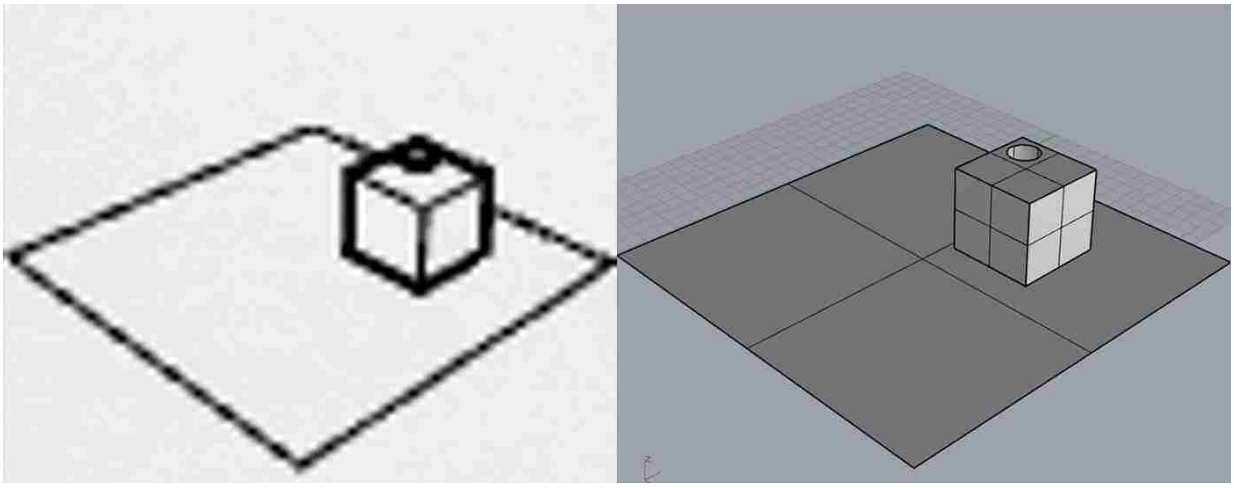

Figure 4.5: Left: Input scene; Right: Object placed at predicted position and rotation

Figure 4.5 shows an example bitmap that was interpreted by the trained Compose3D network; on the right is the object placed in the scene according to the predicted position and rotation of the object. A network like this could be trained to recognize, say, furniture and a designer can use it for fast prototyping of furniture layouts, where the designer is quickly sketching various options and the network is interpreting the sketches and creating 3D scenes that match the sketch. A network like this would also have more serious implications, which are discussed in Section 5.4.

4.2 Interpreting Raster Plans with GraphMapper

GraphMapper is a neural network trained to read raster floor plans and interpret connectivity of spaces. Beyond just interpreting connectivity of spaces, the purpose of this prototype is to demonstrate that neural networks can interpret data from design representations and provide real-time feedback to the designer. There were two reasons behind choosing raster images as the data format for the input floor plans: (1) Raster input formats allow us to use deep convolutional neural networks, which are a well-researched and developed neural network architecture. (2) Floor plans in raster format, in theory, can be sketched by hand. So, a prototype that can accept raster images as input can be used in workflows that need fast prototyping with quick feedback from the neural network. GraphMapper does not yet achieve that due to the obvious lack of proper datasets for this purpose, but it does serve as a proof of concept. This network was trained with a dataset of randomized raster plans that were generated using K-d trees technique (Knecht and Reinhard 2010) described in detail in Section 4.2.1.

Section 3.3 discusses Google's Inception architecture for image recognition and the idea of "transfer learning" where only the final layer of an already trained neural network is trained on a new dataset. This way, instead of retraining from scratch, the network's ability to recognize basic

shapes and patterns in the image is being reused on a new dataset, by using the initial layers of the network and only training the final layer. This technique was used in the development of the GraphMapper. The raster plans were first passed through the pretrained Inception network, and the activations of the penultimate layer of the network were collected and saved as a sort of transformed dataset. These data are called the bottleneck data, referring to the penultimate layer of the pretrained network as the bottleneck. Each bottleneck value is a vector that was supposed to be passed to the Inception's final layers for categorizing the image. It's not quite clear what these bottleneck values contain, but it is known that these values help the last few layers of Inception classify an image into 1 of 1000 categories, so presumably the bottleneck values also contain all the information necessary to interpret our raster plans embedded in them. These bottleneck data are greatly simplified compared to the original dataset, but since the network is pretrained, information regarding the low-level features in the images like edges, shapes and textures is preserved, which is what the final layers would use to categorize the image. These bottleneck data are then used as training data for a simple three-layer fully connected deep neural network, which is trained against the output data containing the connectivity graph of the original plan – the graph showing which rooms are connected to which. Once trained, when using this network, the raster floor plans are first passed through the pretrained Inception network and then the resulting bottleneck data is passed through the trained fully connected neural network. The accuracy of the network is measured based on the number of spaces and the connections between them that are identified correctly. GraphMapper achieved 95.1% accuracy on the test data, which is satisfactory for a proof of concept, but does not represent the real-world usability since the dataset was generated procedurally as opposed to real world plans, which might be hand drawn in real time by a designer prototyping a layout.

4.2.1 Creating the Dataset

As mentioned previously, the dataset for this prototype was created using the K-d tree algorithm (Knecht and Reinhard 2010), which uses a random distribution of points to draw walls and generate floor plans. The original algorithm was slightly modified for this purpose. The following sequence of steps describes the logic generating the dataset:

1. Start with a blank canvas of fixed size and initialize a random collection of ‘n’ points in this space, by randomly sampling coordinates within the bounds of the canvas. The number ‘n’ here is the number of desired spaces in the final plan. This results in a random distribution of points over a space that, at this stage is the entire canvas,
2. Now, calculate the centroid of all the points in the area, which is the entire canvas in the first iteration, and draw a wall through that point. The orientation of the wall is decided between vertical and horizontal based on the aspect ratio of the area that is being divided, and its chosen to produce two spaces with aspect ratios close to 1.
3. Once the wall is drawn, there are two smaller areas each containing their own group of points. Find the centroid of each group and divide their container area the same way and keep repeating this process for all the areas, including the newly created areas until each area contains only one point and hence cannot be subdivided any further.
4. Fill the area of each space with a unique solid color so that the neural network can recognize which spaces are connected to which.
5. Now that all the walls are drawn, and the total space is divided into ‘n’ rooms, randomly decide how many doors to place, which walls to place them on, and where to place them. And create openings corresponding to those doors.

6. Save the raster image to the disk, and then save the corresponding information of doors to be used as a target during the training process. Repeat this process to generate a training set of 100,000 instances and an additional test set of 10,000 instances.

4.2.2 Collecting Bottlenecks and Training

The Inception network used for this prototype was Inception v3 from December 2015, and the bottleneck values collected are from the layer "pool_3/reshape:0" which is a vector of length 2048. These bottleneck vectors were serialized and saved to the disk and used as training input data for a three layer fully connected neural network that uses ReLU, tanh and Sigmoid activations and Cross Entropy loss function (Table 2.2). This network was trained against the originally saved connectivity information (Step 6 in Section 4.2.1) and achieved 95.1% accuracy at detecting doors and interpreting the spaces that the doors are connecting in the plan. For more details about the prototype, see Appendix B.

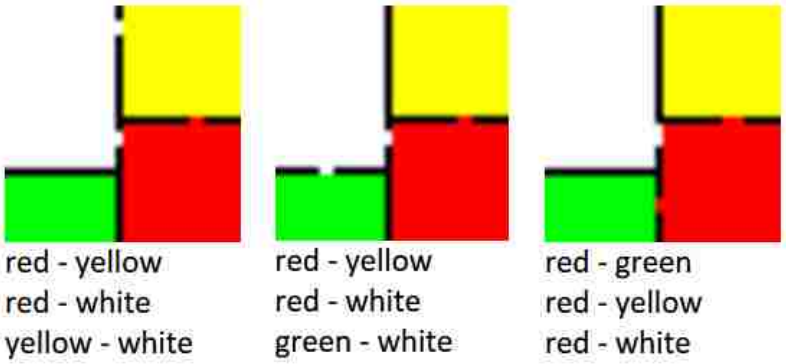

Figure 4.6: Example output from test set showing the connections between spaces

4.2.3 Discussion

To use this same network architecture and training techniques on real world applications, a dataset of raster plans, maybe hand-drawn, and the corresponding connectivity information would be required. There are no such datasets currently and creating such datasets would require humans to interpret hundreds of thousands of floor-plans and label them with connectivity information that they interpreted. This brings the discussion back to the recurring theme of datasets not being readily available and the challenge of creating these datasets, and especially, being able to create these large datasets in a time frame that does not make the original application obsolete.

The ability to retrain a pretrained network with the transfer learning technique can also be very useful. This reduces the need for large datasets, and it reduces the training time and the computational requirement of the training process by essentially recycling the training that these pretrained models went through to learn the basic features like edges, textures and shapes in the data. And by training only the final layers, we are having the pretrained network learn the new data as combinations of the basic features which it has already learned. This means the same trained network can be retrained for several different purposes, faster and with smaller datasets. This is a little close to how people learn new concepts. If someone is learning about a new fruit that they never knew before, they are not starting from scratch, they are understanding that fruit as a combination of basic features like colors, textures, taste and other abstract low-level features that can be interpreted from the sensory data. This reduces the training time and the number of training examples required. The transfer learning technique allows that to happen in the deep neural networks. But to make this work, we would need a pretrained model that went through high quality training, on an unbiased dataset, which requires a lot more curation and iteration than normal training processes that are narrower in their purpose. Inception is one such reliable pretrained

network, but more networks for other purposes need to be trained. This topic is discussed in more detailed in Chapter 5.

4.3 Building a Vocabulary of Design with BIMToVec

Given the recurring challenge of creating datasets in the prototypes previously discussed, it might be worthwhile to look in the most data-dense area of today's design workflows – BIM design applications. BIMToVec is an attempt to apply deep learning techniques to BIM data models. Of all the previously developed prototypes in this thesis, this is the one with the broadest implications and has the most data available for training. BIMToVec uses the same algorithms used by the Natural Language Processing (NLP) models and maps the objects in a BIM model to embedding vectors. These embedding vectors capture the design semantics of these objects.

4.3.1 Parallels with Natural Language Processing

The deep learning algorithms used in natural language processing store and organize data in fundamentally different ways compared to, say, convolutional networks. The core idea behind other deep learning techniques is that any real-world entity can be reduced to a feature vector – a list of numbers that describe that entity. And all such possible entities exist in the continuous "state space" of the features being measured. To elaborate on the idea of continuous feature space, consider image data. All grayscale images of, say, 100 x 100 resolution have exactly 10,000 features – the pixel intensities. If you have an image of a cat and a dog, there exists a linear interpolation between these two images in the feature space that is completely continuous. If that linear interpolation were to be played out like a video, it would look like a fade effect from one image to another. And every image within this interpolation, even if it cannot be classified as a dog or a cat confidently, is technically still an image. A convolutional neural network can be

trained on a dataset of images irrespective of what images are in the dataset, since they all lie in the feature space of images. This also makes it easy to serialize all possible images into an array of a different instances of a singular object called “pixel”. But all real-world data does not exist in such a convenient format.

Natural language processing is one such exception. Spoken and written language involves stream of words and symbols, all from a superset known as the vocabulary. Each word in the vocabulary can further be treated as a list of characters, chosen from, for English, a collection of 26 letters. To convert language into feature vectors for deep learning models, we need to find a way to quantify them – convert words into streams of numbers in a meaningful way. Say, if we assign all the letters in the alphabet numbers from 1 to 26 and treat words as numbers in a base-26 representation, that would be one way to quantify words. But that representation results in a “sparse distribution” which is not good for deep learning purposes for the following reasons: (1) The largest estimate of number of the number of words in English language comes from the Oxford Dictionary 2nd edition (WolframAlpha n.d.), at 600,000 words. The average length of an English word is estimated at 5.1 characters (WolframAlpha n.d.). The total number of possible words that are 5.1 characters long is $26^{5.1} \approx 16{,}457{,}507$. This is what makes text a very sparse distribution. Each possible character combination is not a meaningful word, which means that the feature space is not continuous but is sparsely populated with discrete entities. And more importantly, these entities are not arranged in a way that makes numerical sense. For example, the words “egg” and “yolk” are closely related in their meaning, but the word “ear” is numerically closer to “egg” than “yolk” in our base-26 representation. So even if we built a model that can learn this sparse distribution, it would be very difficult for that model to learn how words are related by meaning. These problems can be overcome by using an “embeddings” model. The technique of training embeddings is

perfectly suited for this kind of data, which is composed of discrete symbols from a vocabulary. An embeddings set can map each word in a vocabulary to an n-dimensional vector. The ‘n’ is typically a large number, for example, the Word2Vec set mentioned in Section 3.2.3 maps words to 128 dimensional vectors.

Slightly modified versions of the algorithms used to train word embeddings (Word2Vec) were used to train BIMToVec prototype. Word embeddings are typically trained on a large corpus of text, in a way that the final trained set of embeddings reflects the semantics of the vocabulary, i.e. two vectors that are very close to each other (dot product close to 1) in the n-dimensional space, should ideally correspond to two words that are very close to each other semantically. This gives the deep learning model an understanding of the meaning of the text rather than just the data used to represent the text. To apply a similar technique in the BIMToVec prototype, a collection of BIM (IFC – Industry Foundation Classes) models, rather than a corpus of text was used as a data source. Text is a one-dimensional composition of discrete entities (words). A designed BIM model is a three-dimensional composition of discreet objects (building design entities like doors, walls, etc.). An embeddings set is trained on these models to map each building object and building material to a 64-dimensional vector, in such a way that proximity in the vector space implies a semantic proximity in a design sense, of the corresponding building objects. For example, since doors almost always occur inside walls in designed buildings, one can say that doors are closely related to walls.

4.3.2 Creating the Dataset

The starting point for creating the dataset for BIMToVec was a collection of 142 IFC files downloaded from some open online sources (DURAARK n.d.) (Open IFC Model Repository n.d.) totaling in size to approximately 2.9 GB. These IFC files were sampled for groups of labels (object names and material names) that are closely related to each other semantically, with the intention

of training the embeddings set against these names. There are multiple ways to decide when two labels are related and how. So, to create the dataset, three different sampling logics were used:

1. Spatial Sampling – Objects that are in spatial proximity to each other are probably related to each other. For example, ceilings are probably related closely to slabs.
2. Material Sampling – Object names and material names associated with that object are probably closely related. For example, windows are probably related to glass.
3. Containment Tree Sampling – In an IFC file (and in most other BIM data models), building elements are organized in a containment tree. The top of this tree has the site, which contains one or more buildings, which contains one or more spaces, walls and so on. Since this containment is deliberately designed to make sense of building models, the proximity of objects in this tree could be used as a measure of how closely they are related to each other.

The sampling itself was done using OpenBIM's .NET IFC toolkit called xBim (OpenBIM n.d.). xBim was used to parse the IFC file and load its contents into memory. The in-memory models were then sampled using the three samplers described above, identifying groups of related object labels and material names, saving all possible pairs of labels as part of the dataset.

4.3.3 Training BIM Embeddings

An embedding set was trained on the final dataset with the vocabulary size of 1325, consisting of object names and material names from the IFC files, and frequently appearing substrings (for example, 'wood' might appear in 'wood floor' and 'wood siding'). The dataset contained 7.8 billion pairs of related words. The training was stopped when a satisfactory level of proximity between related words was reached. Figure 4.7 shows the final 64-dimensional embedding vectors,

flattened to 2 dimensions using t-SNE and plotted with their corresponding labels. Plotting all 1325 words would make the image unclear, so this plot only shows randomly sampled 650 embeddings. See Appendix C for more details about this prototype.

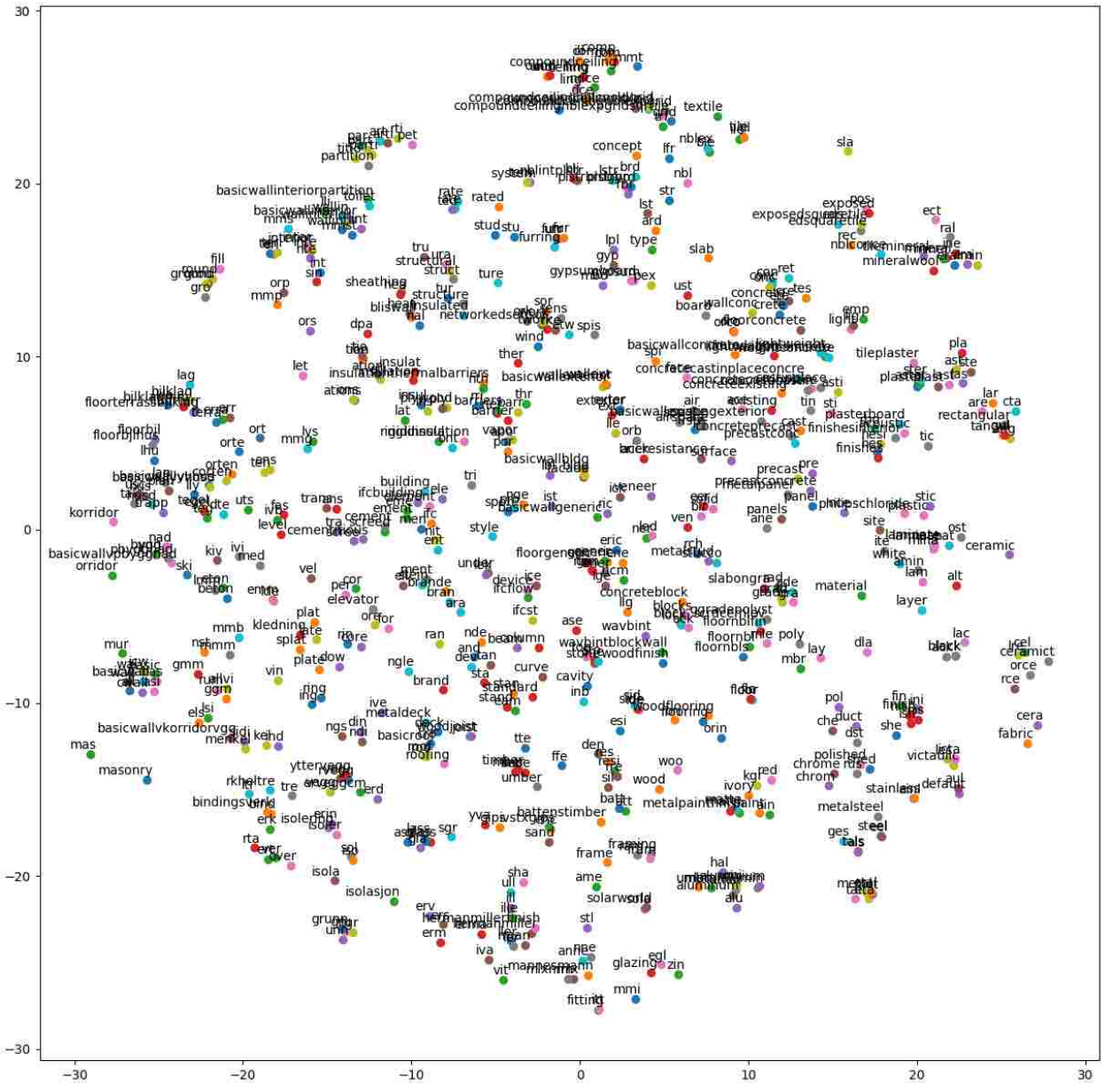

Figure 4.7: Subset of Trained Embeddings Set, plotted in 2-d, flattened using t-SNE

It is evident from the plot in Figure 4.7 that the embeddings set captures some obvious semantic relations between building object labels. Different wall materials are clustered together near the top center, and gypsum board, for example, is close to the wall cluster. All of the metals, steel, chrome etc. are all clustered near the bottom right. Now, because wall, concrete block, and gypsum board are close to each other in the 64-dimensional space, the model knows that they are all related closely semantically, and from the perspective of the designer.

4.3.4 Discussion

Because the embeddings set provides a semantic understanding behind the concepts associated with the building elements and materials, this allows designers to delegate tasks to the design application, which would not be possible otherwise with deterministic logic, suggesting a number of potential applications:

1. Since the embedding vectors represent concepts, they could allow designers to enforce fuzzy rules on their models. For example, since the model understands the concept of, say, walls, the designer could decide that there should not be any furniture close to the walls. And the application would be able to enforce this rule without the designer having to list all the walls and all the furniture.
2. The same fuzzy rules can be extended to automate clash detection. If the designer specifies that a bolt intersecting with a beam is a valid clash, the application can then extrapolate that exception to validate clashes between other types of fasteners and objects, assuming the embeddings set is trained well enough that it captured all the fasteners together in a cluster.
3. In natural language processing, word embeddings are used as a dictionary to translate human concepts into vectors interpretable by computers. And these can then be used to

build any number of other applications, like translation apps, chatbots etc. The same rule applies to BIM embeddings. It can be used to train a model to identify anomalies in the model without depending on a rigid rule sets that are hard to maintain. It would be possible to train a model to guess the building type just by analyzing the embedding vectors of the building elements in the model. For example, if it has lots of beds, it's probably a hotel or a hospital.

4. When a 3D view of a building model is rendered in a BIM application, objects that are visible in a view port are calculated and the corresponding pixels are assigned the right color for that object. This results in a bitmap of colors – an image. A color is the same as a length 3 vector so, a different bitmap could be rendered in the same way, by calculating the visible objects in the view port, and then assigning the embedding vector of the object to the corresponding pixel. This would result in a bitmap of concepts instead of colors. Each pixel of such an image, instead of being 3-deep like in an RGB image, would be as deep as the number dimensions of the embedding vectors. This concept bitmap would allow the application to see what the user sees – actual building elements. This could be used to build an over-the-shoulder virtual assistant.

Despite the potential of an embeddings set, BIMToVec embeddings trained as part of this prototype are not quite up to the required quality. This is because there is no standard vocabulary defined for building elements and this embeddings set picked up all the loose naming conventions and spellings used by the users who built the source IFC models. These IFC models were not built with the intention of training embeddings. To train a reliable, general set of embeddings that would work across all IFC models, they would have to be trained on a carefully curated collection of IFC models, where object and layer naming conventions, material names etc. are all controlled.

Furthermore, there is the problem of embeddings picking up the bias in the dataset. It could pick up the habits and styles of the specific designers who designed the IFC models used in the dataset. This issue is discussed in detail in Chapter 5.

4.4 PROTOTYPES - SUMMARY

Figure 4.8: Stages in a typical design workflow

A typical design workflow might consist of several stages, starting with abstract representations that allow changes with relative ease like, say, concept sketches, then move on to more detailed representations like massing and layouts, drawings, models and working toward a final detailed model, maybe a BIM model, with all the construction details needed for the execution of the project. Though the chain of representations seems linear, the design process itself is hardly linear, as options are being explored, the designer goes back and forth along this path.

Starting with an abstract representation like a concept sketch is helpful, because such representations do not flood the designer with too many questions too soon, like, “what materials, dimensions, profiles and so on?” An abstract representation also has a low cost of change, i.e. changes are easier to make in an abstract representation. As the designer makes more decisions, he / she moves on to more detailed representations which begin to answer more questions about the design. This process typically continues until the construction drawings are produced, but along

the way, at any stage, the designer might decide to change a previous decision and might have to go back to a previous representation, make the change and reflect that change in the more detailed representations. The movement along this chain of representations can happen in two directions: (1) From abstract to detail and (2) From detail to abstract. Because each representation has different level of detail, there cannot exist a clear mapping between two representations.

For example, a 3D Mesh in two different modeling applications has roughly the same amount of information and the same type of information, which makes it possible to export such geometry from one application to another. Another example is that of Autodesk Revit which saves its models as RVT files which can be exported as IFC files. There exists a mapping between these two file formats, which maps each class of objects to a class of objects in IFC, each property in the RVT file is mapped to a property in the IFC file. The export logic in the application looks up the target object classes and properties and creates the IFC file with the data from RVT file. This mapping is possible between these two formats because they contain roughly the same level of detail. Even if not so direct, file formats with a similar level of detail tend to have a reasonable mapping between them that allows automatic translation. Now consider two representations, a perspective sketch and a BIM model: the BIM model is supposed to contain a lot more detail than a sketch can provide. There is no way to create more information from less information by following a series of deterministic instructions, like a conventional computer program does. Hence, it’s not possible to convert a sketch into a BIM model with deterministic logic. When humans interpret sketches to create BIM models, they can create more information from less information by using conventions that they learned through experience. This phenomenon was discussed at the beginning of this chapter.

The prototypes built as part of this thesis were intended to test the potential of deep learning to translate designs from one representation to another by learning such conventions. Table 4.2 shows the directions in which each prototype tries to translate the design representations. Though the prototypes are not good enough to take on real world applications due to the challenges mentioned in the discussion section of each prototype, they show promise that automating this part of the design workflow might be possible in the future. Given that translation between representations requires humans to use their experience and learned conventions, this is hard to automate and hence introduces bottlenecks and kinks in the workflow slowing it down. If this task were to be automated, the design workflows would function much faster and the cost of iteration would be reduced, allowing the designers to explore more ideas without worrying about feasibility. If one were to imagine a software ecosystem that would allow such a workflow, it might consist of several trained deep neural networks, one for every pair of representations that might require translations between them. At least, that was the thought process in the initial phases of this research. But having built the prototypes and having gone through the obstacles to get them working, the lessons learned in this process can be used to speculate about a more robust software ecosystem for designers that uses deep learning and significantly affects how designs are created. Section 5.4 explores one such possibility.

Prototype	Direction of translation
SketchTo3D	Sketches are interpreted with learned conventions to create 3D representations – so design is translated from abstract to detail
GraphMapper	Spaces and the connections between the spaces are interpreted from raster plans – translating from detail to abstract
BIMToVec	Hundreds of IFC models are being sampled to create an embeddings set that captures the semantics of these entities – translating from detail to abstract.

Table 4.2: The Prototypes and the direction in which they translate representations

Chapter 5. FUTURE OF DESIGNING WITH DEEP LEARNING

Despite the abundance of evidence to suggest that deep learning will enable designers to delegate intuitive tasks to a computer in the future; at present, the abilities of deep learning models are rudimentary. The prototypes built as part of this thesis try to explore the ability of deep learning models to translate designs from one representation to another. If these algorithms were to be packaged in the form of an over the shoulder virtual assistant, that would greatly reduce the cost of iteration in the design workflow, since the bottlenecks in most design workflows involve having to translate the design between different representations. Challenges that were encountered in the process of building these prototypes can be further extrapolated to predict the obstacles that will need to be overcome to integrate this technology into real-world design workflows. This chapter addresses these obstacles.

5.1 BUILDING THE DATASETS

One of the most frequently encountered obstacles when developing the prototypes for this thesis was the dearth of datasets. It must be noted that "data" is different from "dataset", especially in the context of deep learning. A Dataset is generally a well formatted, curated and controlled set of data. Typical requirements of deep learning include the need for datasets to be normalized – which means the size, units and formats are homogeneous throughout the data set. A dataset built from a niche source of data tends to be biased to that specific context, which is less desirable than a dataset without any biases.

Take the example of handwriting recognition, the MNIST dataset is arguably the most popular dataset used to benchmark handwriting recognition models. It contains 100,000 handwritten digits, split into two sets, 90,000 for training and 10,000 for testing. Each of these images is a 28 x 28

grayscale bitmap, with the background in black and the digit written in white, with the brightness values scaled down to be in the range of 0 to 1. Each of these images also comes with a corresponding target or the "correct answer" which can be used to evaluate the performance of any handwriting recognition model. This target is provided in the form of a "one-hot" vector. Any recognition model working with MNIST dataset should be able to classify a given handwritten digit into 10 possible digits (0…9), so a one-hot vector is a vector of length 10, where every value except for the one in the position corresponding to the correct classification is a '0' and the value in the position corresponding to the correct classification is a '1'. For example, if the correct classification for the digit is 3, then the one-hot vector will be as follows: [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]. This shows how well-formatted and organized datasets need to be to make deep learning possible, as opposed to an archive of scanned manuscripts, for example, which might be a good target application for the recognition model once it is trained but is of no use for the purposes of training itself.

This differentiation between data and datasets points out a major challenge that needs to be addressed before virtual design assistants can become a reality. Consider the GraphMapper prototype and the dataset that was generated for it. If that network were to be trained on a real-world dataset, then it would need a very large collection of scanned plans (at least in the order of 10,000), and all the plans should follow similar graphic styles and conventions, and they must be normalized to a standard format of storage (like the MNIST images). Each of those plans would also need an accompanying plan-graph of spaces and their connections to train against. As of today, building that dataset would require a lot of human interpretation and labeling to have something to train against. Similar challenges were encountered by computer vision researchers nearly two decades ago when they were trying to build image recognition models. These

challenges were eventually overcome by crowd-sourcing the workload of labeling images and adding images to the dataset. There are some efforts being made in the AEC field to build such datasets and make them accessible to everyone (DURAARK n.d.), but these datasets are limited in their size and quality. To build deep learning models that can be confidently deployed on real-world applications, we need more homogeneous, curated, high quality datasets.

Building datasets for AEC applications is a more vertically integrated task than creating a dataset of labeled images. It requires involvement of a lot of experts, which makes the possibility of crowd-sourcing less viable. The AEC field also has a component of creativity and ownership of designs, which makes the task of creating curated datasets even more complicated. But there is still hope. As explained in Section 5.3, improvements in the deep learning algorithms can mean that training could be accomplished with smaller datasets, which makes the task of building them more feasible.

5.2 Human Designers and Virtual Assistants

Assuming the problem of creating datasets will be overcome in the future, it is still very important to understand the effects that this new technology could have on design workflows and to understand what the dynamic could be between a human designer and a virtual assistant. Compared to the computational design tools that exist today, deep learning makes it possible for humans to delegate more intuitive, judgement dependent tasks to the computer. This raises questions about the division of labor, biases being trained into deep learning models and the black box nature of the deep learning models that exist today. Can a virtual assistant interpret high level instructions and explain the decisions made? Can a human designer and a virtual assistant learn and evolve together as they work on various projects?

5.2.1 The Black Box Problem

Despite the progress made in deep learning in the last few years in terms of neural network architectures, training techniques etc., deep learning models are still largely black boxes. The only way to evaluate them is with a test dataset, and nothing can be clearly understood by looking at the learned weights inside the network. Most applications that use deep learning today are general utilities that are not often critical; that is, if they fail there are other, albeit inconvenient, reliable ways to accomplish the task. Good examples for these are the virtual assistants that come with most smartphone operating systems today. And the most effective solution to the black-box problem so far has been to tweak the dataset, retrain the network, and essentially go through a lot of trial and error to get the performance to a satisfactory level. This could be a problem for the AEC industry, and could prevent the widespread adoption of these new technologies on real world projects.

In the AEC industry, projects often involve hundreds of people splitting up the workload, sharing responsibilities and coordinating with each other to make the project a reality. Anything that upsets this workflow has implications in real life, especially monetary implications that can be traced back to the responsible entity in the chain. A deep learning model could be trained so well that it works amazingly well on 9 out of 10 instances of real world data, but in the AEC industry, that would heavily discourage designers from adopting these new technologies. If a human assistant produces an unsatisfactory result, he/she can be held responsible for it, questioned, and if there is a misunderstanding, it can be cleared up with the human assistant, so that the same mistake is avoided in the future. An equivalent interaction would not be possible with a black-box deep learning model. If the model produces a sub-par result, there is no way to clearly know what went wrong, no way to go inside the model's weights and edit them to fix the problem. The best option

is to guess what went wrong, tweak the hyperparameters of the model or the dataset, and then retrain the model without any guarantee that the problem is fixed. Section 5.3 discusses some of the latest research happening in deep learning, which suggest that deep neural networks might not remain black-boxes in the future.

Professionals in the AEC industry were interviewed as part of this thesis to understand their view on deep learning. The popular view on the black box issue seems to be that in the early stages of adoption of these technologies in AEC projects, the tasks assigned to deep learning models will be small enough that when they produce unexpected results, the causes will be obvious, and it will be feasible to fix them (Belcher 2017, Holland 2018). But a more important concern expressed by the professionals was related to the data formats used by deep learning and other artificial intelligence models.

5.2.2 Human Editable Data Formats

Many of the breakthrough developments that have happened in deep learning in the last few years have had something to do with images, whether it was recognizing images, generating images, labeling images, etc. This is made possible because of the weight-sharing strategy employed by the convolutional neural networks. Despite the diversity of tasks that convolutional layers can do in all these networks, all images in all the datasets have one thing in common — pixels. A pixel is the only primitive on which images rely for complete definition. This makes image data very homogeneous. The same homogeneity of data can be found in other datasets used by other neural networks. This homogeneity is the key to why neural networks work so well, the weights and activations inside the neural network are tweaked for this one primitive type, and they do not have to change or adapt.

This compatibility of neural networks with homogeneous data formats also explains why the first SketchTo3D prototype had to generate voxel maps instead of 3D models with surface definitions. Surface definitions involve defining primitives, like extrusions, sweeps, revolution surfaces, etc., as opposed to a binary voxel map, which is just a homogeneous list of 0s and 1s. A mesh is another example of homogeneous 3D data. If mesh data were to be prepared for a deep learning model, it could be interpreted as a homogeneous lists of vertex coordinates and faces. But this introduces a problem for designers who work with 3D models; these homogeneous data formats are very inconvenient for human designers to work on. Human designers need clearly defined labels and primitives that make it easy for them to communicate their intentions to other people. For example, when working with geometry primitives one designer can say to another something like, “The depth of that extrusion is not the correct value.” But if they were working with a model that had nothing but meshes, it not only makes it harder to communicate, it is also harder to edit. The primitives defined in most 3D modeling software try to capture the intentions of the humans creating these 3D models. When we humans think about 3D shapes, we think about the logic behind how those shapes are created, hence we have concepts like sweeps and extrusions. It is very unnatural for humans to think about 3D geometry as lists of vertex coordinates and faces. Editing meshes involves manipulating data in bulk, and it makes the process more difficult, harder to find and fix mistakes and harder to communicate. This is a very serious dilemma because the fact that homogeneous data formats do not need a library of definitions to be interpreted is what makes them so convenient for neural networks, but that same reason is what makes them so unfriendly for workflows involving humans. This topic is explored further in Section 5.4.

It might be possible to find a data format that is somewhere between meshes and primitive based surface models, something like NURBS surfaces. They are defined by control points, weights,

knots and an integer degree. Though not as flattened and homogeneous as a mesh data format, it is still better than a primitive based format for deep learning purposes. But using NURBS surfaces with deep learning raises other concerns and opportunities. Surface modeling practices is already a popular topic among the surface modeling community (Belcher 2017), which includes boat hull designers, shoe designers etc. It is possible to achieve the same surface, within some degree of tolerance of course, with two or more completely different distributions of control points and their corresponding weights. But, an experienced surface modeler will know which one of those ways is the best, because it makes editing the surface model easier in the future. For example, for a boat hull, the likely edits that will be made to that surface model in the future can be anticipated by an experienced designer, so he/she would know which distribution of control points would make the most sense. Any given surface can be modeled, within some degree of tolerance, using either degree-3 surfaces or degree-4 or degree-5 surfaces, the distribution of control points and weights will be different in each case. But industrial designers and automobile designers prefer using higher degree curves (degrees 5 to 7) because they give them the precision required to manufacture these objects (Paquette 2008). The knowledge these designers have about choosing a distribution of control points to make the surface editable in the future, or to use a degree for the curve or surface for manufacturable objects, are things that they learned from experience. The implications that these modeling techniques have later in the workflow cannot be inferred directly from the shapes produced from this geometry. Since they are learned from experience and cannot be defined rationally, this is associative knowledge, and neural networks might be able to learn these patterns and might be able to create the surface models with the correct degree for the curve and the correct distribution of the control points. Such a network could be used to, say, fix the surface model

created by an inexperienced designer, or to interpret sketches of a designer and convert them into a human editable surface model.

5.3 Capsule Networks: Better AI

In November 2017, a new neural network architecture was introduced -- capsule networks. It is claimed that capsule networks can do “inverse graphics” (Sabour, Frosst and Hinton 2017). Where computer graphics is mainly concerned with looking at the data and all the objects present in it, then figuring out how they would appear on screen and rendering them, inverse graphics does the opposite; it is the ability to look at a rendered image and then estimate the probabilities of what component objects are present in the picture and what the subject of the picture is. Experts in the field claim that this network architecture is far more potent than convolutional networks. Capsule networks can perform recognition tasks, but they can also cope with changes in the viewport, camera angle, etc. In other words, they can recognize the same object even if the orientation of the camera is slightly different. This would have some implications for the design community trying to incorporate deep learning into their workflows. Capsule networks also show signs that with better neural network architecture, we might be able to pose the question “why” for the output generated by the neural network, which can potentially solve the black-box issue that was previously discussed.

5.3.1 Transparency of Capsule Networks

If the final activations of a capsule network classify an image as, say, the interior of a bedroom, then the layer of capsules before the final layer would have identified objects like bed, lamp, etc. The layers before that would have identified things like, bed post, mattress, lamp shade, etc. Not only are these objects identified, these activations can be visualized and traced back, essentially

allowing us to ask the network why it thinks the picture is of a bedroom. The key difference here, compared to convolutional networks, is that the activations of intermediate layers in convolutional networks are feature maps that do not necessarily correlate to human concepts or vocabulary. But the capsule networks allow us to put human understandable labels on these activations, making them more transparent. That is why they are considered to perform inverse graphics.

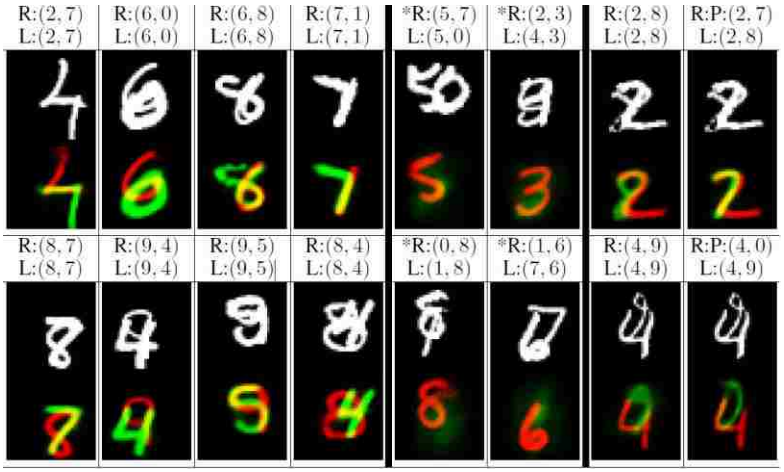

Figure 5.1: Reconstructions of digits from Caps-Net on Multi-MNIST dataset

In theory, though it sounds like the black box problem is completely fixed, the actual proof that we have today of the capabilities of capsule networks is very rudimentary. The Multi-MNIST dataset contains two handwritten digits per image overlaid on each other and the performance of a model on this set is measured as the ability of the model to predict both the digits correctly. Caps-Net achieved 95% accuracy on this dataset. But as explained before, Caps-Net can do more than just classify images. Figure 5.1, taken from the original paper (Sabour, Frosst and Hinton 2017) shows the reconstructions of the two identified digits from the corresponding capsule activations. For example, in the second image from the top-left, the capsule network predicted 6 and 0. When the capsule activations are used to reconstruct the digits, it is analogous to asking the network why

it thinks they are 6 and 0, which is answered in the reconstruction with the red and green color overlays.

5.3.2 Human Interpretable Features

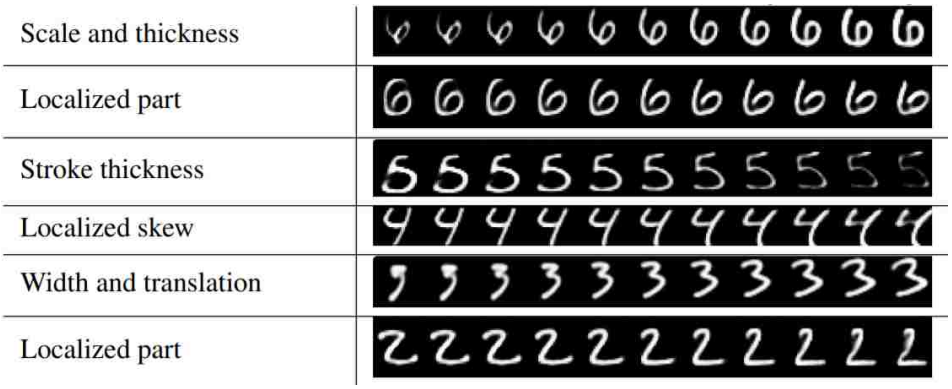

Figure 5.2: Digits generated by tweaking the dimensions capsule activations

The capsule activations contain the probability that a component object is present in the image (like the lamp in the bedroom) and information about how the object is positioned and oriented. Each dimension of these capsule activation vectors seems to be representing a human interpretable qualitative aspect of the graphic. The column on the left shows the qualities that these dimensions are affecting, and on the right are the images generated by tweaking these dimensions. It is evident that these dimensions can change qualitative aspects of the digit like width, stroke thickness, etc. Though this is a very rudimentary proof that uses the MNIST dataset, in theory, if Caps-Net were to be scaled to work on more complex data, we could have a workflow that involves a human designer and a Caps-Net, where the human designer can ask the network to change certain characteristics of its output, much like a human assistant. It is not completely clear at this point whether capsule networks would scale for larger, more complex data and still retain all these advantages, but according to Geoffrey Hinton, they will. That at least, gives us promise that the

black-box problem discussed in Section 5.2.1 can be fixed with this new neural network architecture.

5.4 The Elusive Design Pixel

Section 5.2.2 discusses the problem of data formats, which is a result of the fact that deep learning prefers homogeneous data formats, while human designers prefer data formats with discrete labels and primitives that do a better job of capturing intentions. So many breakthroughs in deep learning have something to do with processing images and videos because of the homogeneity of this data, which is composed of pixels. Pixels make it possible to represent any graphic design in a homogeneous way. This section explores the idea of a "design pixel", which would be to a designer in the AEC field what a conventional pixel is to a graphic designer – a single primitive, arrangements of instances of which can represent any possible design. There are several complexities in building/product design that need to be taken into consideration, which make the task of coming up with a design pixel very challenging, hence its elusive character. For example, the range of representations used in a design workflow creates one of the challenges. There is no single canvas that a building designer can use from the beginning until the end of the design process, for which we can define a 'pixel'. This is further complicated by the fact that at every stage of a design, the designer is concerned with different aspects of the design.

5.4.1 Imagining a Design Pixel with Emulated Homogeneity

One way to interpret designs in a homogeneous way is to think of them as compositions of concepts and ideas. With adequate knowledge of the design field and the conventions used, it would be possible to convert a composition of ideas into any representation. An embeddings set like the one discussed in Section 4.3, if trained on a high quality curated dataset, would be able to capture

design entities as concepts, with their semantics embedded in the high dimensional space. Such an embedding set, together with other deep learning algorithms can be used to emulate homogeneity in design data. The prototypes (discussed in Chapter 4.) demonstrate the challenges encountered in their development process, and the lessons learned lead us to imagine a better ecosystem of deep learning models that can change how designs are created.

The prototypes themselves were made to translate design data between two representations, but in hindsight, that strategy will require training of a lot more prototypes, compared to, say, having a central, persistent data format that each deep learning model maps a representation to. For example, with the deep learning technologies that exist today, with the right dataset, it would be possible to create a model that can interpret images and map them to building elements. The prototype discussed in 4.2 (GraphMapper) is a proof of concept for being able to interpret plans and recognize building elements in them. There are deep learning models like Google's QuickDraw (Google n.d.) which made suggestive drawing a reality, albeit in the domain of simple drawings and doodles. The application can recognize what the user is trying to draw. Google's "Auto Draw" (Statt 2017) can take this even further. The application can not only recognize what the user is trying to draw, but then replace it with a cleaner version of the drawing fetched from a library of existing drawings. This technology as it exists today, trained with the correct datasets, can be very useful in prototyping building plans, and furniture layouts etc., as demonstrated by the Compose3D prototype discussed in Section 4.1.4. This capacity to recognize and assist, if combined with a well-trained building element embeddings set, can be packaged as a family of deep neural networks that can map different representations (sketches, plans, bubble diagrams etc.) to a central homogeneous data format. Since the embeddings can represent objects and ideas, a central data format that incorporates embeddings together with spatial composition of objects

would be able to persist a virtual building as a composition of ideas and objects rather than 3D geometry. If a family of deep neural networks can map all representations to a central homogeneous data format, then the user can work on any representation, but still be working on the same persistent central data format. This means the user could switch between representations any time and continue working seamlessly. Figure 5.3 is a schematic diagram depicting this ecosystem of software that would support such a workflow. Of course, for this process to feel seamless, the conversion of data from each representation to the central data format must happen fast – comparable to how graphic editing software today can seamlessly convert vector data into raster data for on screen display, and this is done fast enough that the user does not notice it at all. So, it is clear that such a software ecosystem with trained deep learning models integrated with an elusive homogeneous data format for building design will likely not become a reality for a long time, given the challenges of processing power that deep learning requires.

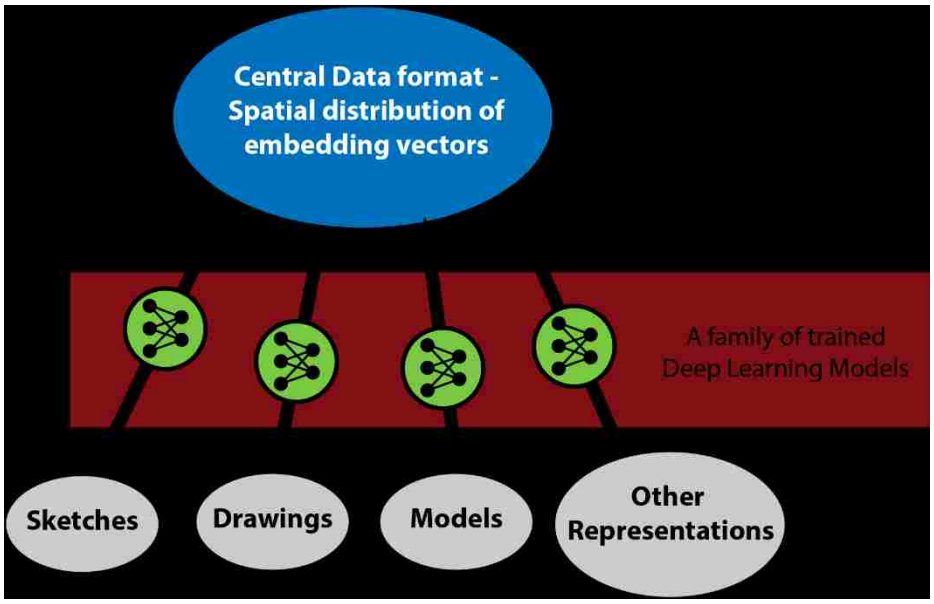

Figure 5.3: Schematic: Elusive Design Pixel

5.4.2 A Composition of Ideas

Despite all the pieces not quite falling into their places and not having a concrete implementation of the Elusive Design Pixel data format, it is possible to explore the implications of such a software ecosystem. At this point, it is not completely clear how spatial information and embedding vectors could be integrated to create this data format, but the significance of using embedding vectors can be demonstrated by taking existing data formats and extrapolating them to use embeddings. Figure 5.4 shows one such example, extrapolating images to use embeddings instead of colors. A color is essentially a vector – [R, G, B] of length 3 so in a typical image each pixel has depth 3 (or 24 bits if each color is 8 bits). Instead of using color, embedding vectors can be used, making the pixel as deep as the length of the embedding vectors. This would make the image a bitmap of concepts rather than a bitmap of colors. Such a bitmap would also be compatible with existing deep learning techniques that are designed to work with images.

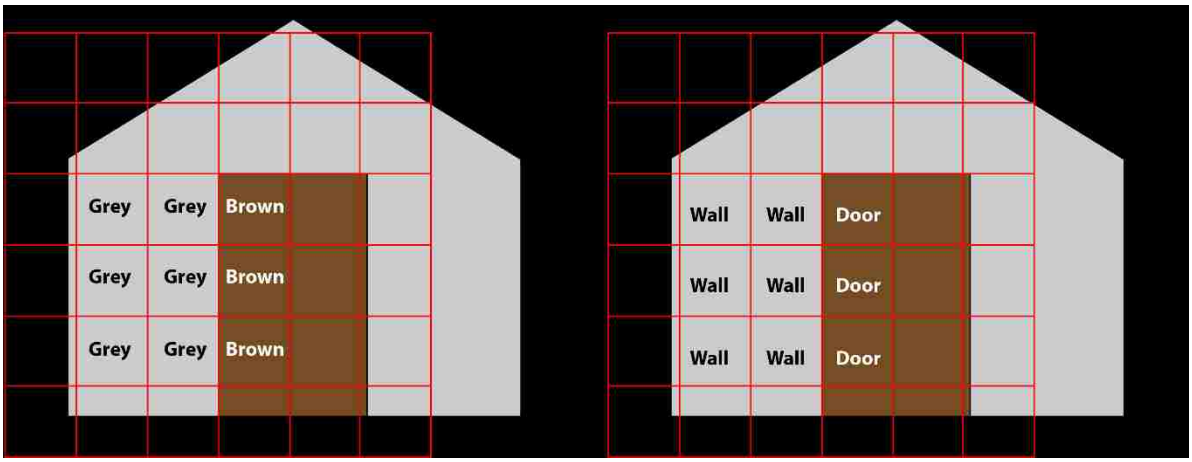

Figure 5.4: Left: Bitmap of colors; Right: Bitmap of concepts

5.4.3 Constructing Ideas, not Models

As discussed previously, an ecosystem of deep neural networks and an embeddings-based central data format can allow the designer to edit a single spatial composition of ideas from several

representations. But too many parts of this picture are missing to properly understand the implications of such a possibility. It might help to use the similar developments occurring in the field of natural language processing as an analogy. In NLP, there is a concept of a "thought vector" (Kiros, et al. 2015) – sentences are converted into lists of embeddings using a trained embeddings set like Word2Vec and this list is then passed to a recurrent neural network which outputs a final vector representing its state at the end of the sentence. Recurrent neural networks are used here for two reasons: (1) The lengths of sentences are variable and RNNs can handle that kind of data (2) the meaning of the sentence is sensitive to the order of the words. The final state vector of the RNN at the end of the sentence is known as a thought vector, which supposedly represents the meaning of the sentence in vector space. These thought vectors have been proven to be very effective in translation apps that use deep learning. A sentence is converted into a list of embedding vectors, passed through an encoder network which outputs the thought vector which is then passed to the decoder network which outputs a series of embedding vectors, corresponding words to which are then looked up in a embeddings set of a different language, finishing the translation process. Some of the best translation applications today use versions of this technique (Bahdanau, Cho and Bengio 2014). This thought vector essentially represents the idea behind the input sentence, which brings us to the similarity that this technique has to the elusive design pixel data format that was discussed before. In both cases the idea behind a concrete representation, say, a drawing in the case of the elusive design pixel and a written sentence in the case of NLP, is being converted into a vector that represents the core idea behind the concrete representation. As speculated earlier, a designer being able to move from representation to representation during the workflow, while the data persists in a central embeddings-based data format is analogous to editing text in one language, then later editing the same text in a different language, while the ideas in the text persist

as thought vectors. But the fact is that even the best translation models today cannot do high enough quality translation to make a multi-language editing workflow feasible. Though the thought vector models today work well for one-way translation of text, the quality of the text after a two-way translation is greatly reduced, and often the meaning is not preserved well. But that is just the present technology, and we might see improvements in the future. So, it must be noted that editing a design across multiple representations might be a possibility in the future, but it is too soon to make any concrete speculations about how that might work. The deep learning techniques today show signs that hint toward such technologies, but their quality is not good enough yet. They will likely have to go through a long process of evolution and adaptation, likely starting from a narrow niche in the design field where the vocabulary is fairly limited, like shoe design, or hull design and slowly expanding to broader areas of design – all of that in addition to other challenges discussed in Chapter 5. Deep learning may not have any immediate paradigm shifting effects on the AEC field, but it allows us to envision a future where designers construct ideas and not just models.

5.4.4 Challenges

There are some fundamental, almost philosophical questions that representation of designs as idea maps and text as thought vectors bring up. Can an idea exist in isolation from concrete representations that are used to express the idea? The answer to this question is not clear at this point but, given how different representations all can point to the same idea during a design workflow, it might be possible for ideas to exist in isolation from concrete representations as higher dimensional vectors. These questions are especially difficult to think about given how counterintuitive high dimensional vector space operations are to think about.

A strong argument against the claim of an idea existing in isolation from concrete representations can be made with the example of languages and cultural ideas that are embedded in language.

There are several languages with words and concepts that do not have direct translations in English. One such example is the Finnish word “Sisu”, which does not have a proper translation in English (Wikipedia n.d.). Similar ideas were discussed in Chapter 1. But the existence of concepts in cultures that cannot be translated completely to other cultures is not a valid counter example against existence of ideas in isolation from the concrete representations used to express the ideas. It must be noted that different cultures evolved languages, words and values in different contexts and settings. This is analogous to neural networks (or embedding vectors) being trained with different datasets. For example, if one were to study the culture, values and history of Finland, one might develop better appreciation of the word “Sisu”. This means that for ideas to exist in isolation from concrete representations, the embedding vectors and the family of neural networks all need to be trained on a standardized dataset.

It is difficult to make concrete speculations about the effects of deep learning far into the future. Deep learning will most likely be adopted by narrow domains in design, like, shoe designers or hull designers, etc. It will also likely gain momentum in the field of graphic design, mainly because graphic data already has a pixel that is not elusive. Deep learning might gradually spread to other areas of design. The eventual effects of deep learning in building design might not manifest in the exact way as speculated in this thesis, but there is plenty of evidence to suggest that deep learning will have a significant effect on how designers work.

REFERENCES

n.d. *Adobe Voco - Wikipedia.* Accessed 2017. https://en.wikipedia.org/wiki/Adobe_Voco.

Ames, Adelbert. 1952. "The Ames Demonstrations in Perception." *Hafner Publishing.* Accessed 2018. https://en.wikipedia.org/wiki/Ames_room.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. 2014. "Neural Machine Translation by Jointly Learning to Align and Translate." *Computing Research Repository.*

Belcher, Daniel, interview by Ranjeeth Mahankali. 2017. *Developer at McNeel* (November 30).

Coyne, R. D. 1997. "Creativity as Commonplace." *Design Studies 18*, 135-141.

—. 1990. "Tools for Exploring Associative Reasoning in Design." *CAAD futures Digital Proceedings.* MIT Press.

Coyne, R. D., and A. G. Postmus. 1990. "Spatial Applications of Neural Networks in Computer Aided Design." *Artificial Intelligence in Engineering* Vol 5, Issue 1, Pages 9-22.

Coyne, R. D., M. A. Rosenman, A. D. Radford, M. Balachandran, and J. S. Gero. 1989. *Knowledge-Based Design Systems.* Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Deng, J., W. Dong, R. Socher, L. J. Li, K. Li, and Fei-Fei Li. 2009. "ImageNet: A Large-Scale Hierarchical Image Database." *IEEE Computer Vision and Pattern Recognition (CVPR).*

DURAARK. n.d. *DURAARK Datasets.* Accessed June 2017. http://duraark.eu/data-repository/#ifc.

Gero, J. S., and M. Yan. 1994. "Shape Emergence by Symbolic Reasoning." *Environment and Planning B: Urban Analytics and City Science* Vol 21, Issue 2, pp. 191 - 212.

Gero, J. S., and Mary Lou Maher. 1991. "Mutation and Analogy to support Creativity in Computer Aided Design." *CAAD futures Digital Proceedings* 261 to 270.

Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengjo. 2014. "Generative Adversarial Nets." *Advances in Neural Information Processing Systems 27* 2672-2680.

Google. n.d. *Quick, Draw!* Accessed January 2018. https://quickdraw.withgoogle.com/.

Hinton, Geoffrey, and Vinod Nair. 2010. "Recitified Linear Units Improve Restricted Boltzmann Machines." *Proceedings of the 27th International Conference on Maching Learning (ICML-10)* 807 - 814.

Hinton, Geoffrey, Li Deng, Dong Yu, George Dahl, Abdel Rahman Mohamed, Navdeep Jaitly, Andrew Senior, et al. 2012. "Deep Neural Networks for Acoustic Modeling in Speech Recognition." *IEEE Signal Processing Magazine,* 82 - 97.

Holland, Nate, interview by Ranjeeth Mahankali. 2018. *Design Computation Leader at NBBJ* (February 1).

Kalay, Yehuda. 2004. *Architecture's New Media: Principles, Theories and Methods of Computer Aided Design.* MIT Press.

Kiros, Ryan, Yukun Zhu, Ruslan Salakhutdinov, Richard S. Zemel, Antonio Torralba, Raquel Urtasun, and Sanja Fidler. 2015. "Skip-Thought Vectors." *Computing Research Repository.*

Knecht, Knecht, and Koineg Reinhard. 2010. "Generating FLoor Plan Layouts with K-d Trees and Evolutionary Algorithms." *13th Generative Art Conference.* 238-253.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey Hinton. 2012. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in Neural Information Processing Systems 25* 1097-1105.

Maaten, Laurens van der, and Geoffrey Hinton. 2008. "Visualizing Data using t-SNE." *Journal of Machine Learning Research 9* 2579-2605.

McCaig, Greame, Steve DiPaola, and Liane Gabora. 2016. "Deep Convolutional Networks as Models of Generalization and Blending." *Computing Research Repository.*

McCulloch, Warren S., and Walter Pitts. 1943. "A Logical Calculus of the Idea immanent in Nervous Activity." *The Bulletin of Mathematical Biophysics* 5.4: 115-133.

Metz, Luke. 2015. *Visualizing with t-SNE.* August 25. Accessed January 2018. https://indico.io/blog/visualizing-with-t-sne/.

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. "Distributed Representations of Words and Phrases and their Compositionality." *Computing Research Repository.*

Mordinstev, Alexander, Christopher Olah, and Mike Tyka. 2015. *Inceptionism: Going Deeper into Neural Networks - Google Research Blog.* June 17. https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html.

Open IFC Model Repository - Univ. of Auckland. n.d. *Open IFC Model Repository.* Accessed June 2017. http://openifcmodel.cs.auckland.ac.nz/.

OpenBIM. n.d. *The Open Toolbox for BIM.* Accessed June 2017. http://www.openbim.org/.

Paquette, Andrew. 2008. "Nurbs Curves." In *Computer Graphics for Artists: An Introduction,* by Andrew Paquette, 185-198. London: Springer.

Petrovic, Ivan K. 1996. "Computer Design Agents and Creative Interfaces." *Automation in Construction 5* 151-159.

Rumelhart, D. E., and J. L. McCIelIand. 1986. "On learning the past tense of English verbs." *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 2, Psychological and Biological Models* 216-271.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. 1986. "Learning Representations by Back-Propagating Errors." *Nature Vol. 323* 533-536.

Sabour, Sara, Nicholas Frosst, and Geoffrey E. Hinton. 2017. "Dynamic Routing Between Capsules." *Computing Research Repository.*

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research, Volume 15* 1929-1958.

Statt, Nick. 2017. *Google's AI doodle bot will transform your crude drawings into glorious clip art.* April 11. Accessed February 18, 2018. https://www.theverge.com/2017/4/11/15263434/google-ai-autodraw-doodle-bot-drawing-image-recognition.

Szegedy, Christian, Sergey Ioffe, Jonathan Shlens, Zbigniew Wojna, and Vincent Vanhoucke. 2015. "Rethinking the Inception Architecture for Computer Vision." *Computing Research Repository.*

Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. "Going Deeper With Convolutions." *Computing Research Repository, Volume ABS/1409.4842.*

Wikipedia. n.d. *Sisu: Wikipedia.* Accessed March 3rd, 2018. https://en.wikipedia.org/wiki/Sisu.

WolframAlpha. n.d. *Average Word Length in English.* Accessed Jan 27, 2017. http://www.wolframalpha.com/input/?i=average+english+word+length.

—. n.d. *Total Words in English Language.* Accessed Jan 27, 2017. http://www.wolframalpha.com/input/?i=total+words+in+english.

APPENDIX A: SKETCHTO3D

The SketchTo3D network was built using the Tensorflow(r1.2) library. The network was designed with 2 fully connected layers, followed by 2 convolutional layers, followed by 2 deconvolutional layers. All layers, except for the last two, use the Rectified Linear Units activation function. The penultimate deconvolutional layer uses a Hyperbolic Tangent activation function and the final layer uses a sigmoid activation function.

For the loss function, separate training attempts were made with the custom loss function discussed in Section 4.1.2 and with a sigmoid loss function that is commonly used in multi label classifiers. The custom loss function yielded the best results, so the final version of the network uses the custom loss function. The Adam Optimizer built into the Tensorflow library was used for training the network, with a learning rate of 0.0001.

To test this prototype, the source code can be downloaded from a public GitHub repository at: https://github.com/ranjeethmahankali/sketchTo3d. The repository contains the following:

- model.py and model_0.py: definitions of two different versions of the neural network.
- dataGen.py: Rhino-Python script to generate the dataset for training and testing.
- trainModel.py and useModel.py: Python files to train the model and test the trained model.
- results: This folder contains more examples of input-output pairs for the trained sketchTo3D network.

The code also relies on other python libraries like Pillow – a windows adaption of the Python Imaging Library for processing images, NumPy for manipulating vector data. The list of external dependencies is not limited to these libraries.

APPENDIX B: GRAPHMAPPER

As discussed in the main text, this network uses the transfer learning technique. To generate the bottleneck values for the input images in the dataset, a December 2015 version (v2) of Inception was used. The bottleneck values were captured by inputting the images into the "DecodeJpeg:0" layer of inception and saving the activations from the "pool_3/_reshape :0" layer.

A separate neural network, with 3 fully connected layers was trained on the generated bottleneck values, hence implementing transfer learning. The three layers in the network use Rectified Linear Units, Hyperbolic Tangent and Sigmoid for activation functions respectively. The network was trained to minimize the Cross-Entropy loss using TensorFlow's built in Adam Optimizer, with learning rate set to 0.0001. The network also uses L2 regularization, with the alpha-factor set to 0.002 (used to scale the L2 loss before adding it to the original cross-entropy loss) to avoid overfitting.

All of the source code for this prototype is available in a public GitHub repository, which can be found at: https://github.com/ranjeethmahankali/graphMapper. The repository contains the following:

- spaceGraph.py and spaceGraph_ext.py: a prewritten class used to store the nested connectivity information of spaces in a floor plan. The extension class handles exporting the plans as geometry as well.

- planeVec.py: 2D geometry and vector functions that are used by the space graph class to generate floor plans.

- dataGen.py: Implements the k-d tree algorithm to generate floor plans, using the space graph class, to generate the dataset.

- inceptionPrep.py: Uses the inception model to generate bottleneck values for each image in the dataset.

- inceptionModel.py: The three-layer deep neural network definition, designed to interpret the bottleneck values generated from inception.

- trainModel.py and testModel.py: Python files for training and testing the model.

The model also uses Tensorflow(r1.2), NumPy and Pillow libraries at various points to generate data and manipulate vector data, though the list of external dependencies is not limited to these libraries.

APPENDIX C: BIMTOVEC

The source code for this prototype is available on a public GitHub repository at the following link: https://github.com/ranjeethmahankali/BIMToVec. The repository contains the following:

- BIMToVecDotNet – a Visual Studio solution with two projects – IfcSampler for sampling the IFC files using xBIM, and the RevitBIMToVec which is the Revit add-in which integrates BIMToVec with Revit.

- create_atom_dataset.py and atomize_vocab.py: to extract atoms – frequently occurring substrings in the main vocabulary, from the vocabulary exported by the IfcSampler.

- model_embeddings.py and atom_embeddings.py: embedding definitions for the main vocabulary and the atoms.

- train_atoms.py and train_embeddings.py: to train the embedding vectors for the atoms and the main vocabulary.

- model_server.py: to establish a TCP client-server relationship between the trained embeddings model loaded into a python environment and the .NET Revit add-in.

- word2Vec_reference.pdf: copy of the Notebook demonstrating how to train and test the word2Vec embeddings.

- embeddingCalc.py: miscellaneous collection of functions related to mathematics of embedding vectors.