LINKÖPING UNIVERSITY

MASTER THESIS

30 ECTS | LIU-IEI-TEK-A–18/03082—SE

A Framework for Generative Product Design Powered by Deep Learning and Artificial Intelligence

APPLIED ON EVERYDAY PRODUCTS

Authors:
Alexander NILSSON
Martin THÖNNERS

Supervisor:
Johan PERSSON

Examiner:
Johan ÖLVANDER

Department of Management and Engineering
Division of Machine Design

June 14, 2018

Copyright

aleni766@student.liu.se
marth489@student.liu.se

"All models are wrong, but some are useful."

George E. P. Box

Abstract

In this master's thesis we explore the idea of using artificial intelligence in the product design process and seek to develop a conceptual framework for how it can be incorporated to make user customized products more accessible and affordable for everyone.

We show how generative deep learning models such as Variational Auto Encoders and Generative Adversarial Networks can be implemented to generate design variations of windows and clarify the general implementation process along with insights from recent research in the field.

The proposed framework consists of three parts: (1) A morphological matrix connecting several identified possibilities of implementation to specific parts of the product design process. (2) A general step-by-step process on how to incorporate generative deep learning. (3) A description of common challenges, strategies and solutions related to the implementation process. Together with the framework we also provide a system for automatic gathering and cleaning of image data as well as a dataset containing 4564 images of windows in a front view perspective.

Acknowledgements

We would like to thank everyone at SkyMaker AB for interesting discussions and support, especially Kristofer Skyttner for giving us the opportunity to work on this thesis and Jonathan Brandtberg for help and supervision throughout the project.

At Linköping University we would like to thank our supervisor Johan Persson and our examiner Johan Ölvander for the freedom entrusted in us and the guidance and support provided when needed.

We would also like to thank friends who have contributed to our work in different ways, with a special thank you to Axel Skyttner for insightful exchanges of ideas on the subject of deep learning.

Finally, we want to thank our opponents Anna Bengtsson and Camilla Wehlin for valuable thoughts and feedback to improve our work.

Alexander Nilsson & Martin Thönners
Linköping, June 14, 2018

Contents

List of Figures

List of Tables

List of Abbreviations

AAE	**A**dversarial **A**uto **E**ncoder
AE	**A**uto **E**ncoder
AI	**A**rtificial **I**ntelligence
API	**A**pplication **P**rogramming **I**nterface
CNN	**C**onvolutional **N**eural **N**etwork
DL	**D**eep **L**earning
DRN	**D**eep **R**esidual **N**etwork
FC	**F**ully **C**onnected
FFNN	**F**eed **F**orward **N**eural **N**etwork
GAN	**G**enerative **A**dversarial **N**etwork
GD	**G**enerative **D**esign
GDP	**G**enerative **D**esign **P**rocesses
ML	**M**achine **L**earning
NN	**N**eural **N**etwork
PDP	**P**roduct **D**esign **P**rocess
RNN	**R**ecurrent **N**eural **N**etwork
VAE	**V**ariational **A**uto **E**ncoder

Terminology

Vanishing Gradients	A problem encountered during backpropagation where the gradients for several consecutive layers are very close to zero causing the back propagated error to go to zero.
Exploding Gradients	A problem encountered during back propagation where the gradients for several consecutive layers are far above unity causing the back propagated error to go to infinity.
Shallow (Network)	A neural network with no more than a single hidden layer.
Deep (Network)	A neural network with more than one hidden layer.
Sparse (Network)	A neural network where the majority of the connective weights are set to zero.
Latent Space	The space $\mathbb{R}^n$ in which the encoded data $z \in \mathbb{R}^n$ lies in the bottleneck layer (with n nodes) of an Auto Encoder.
Feature	A measurable property or attribute of data.
Label	Describes a class which an associated piece of data belongs to.
Overfitting	When a model learns unwanted features such as noise from training data, often occurs when the model is to complex or the quantity of training data is to low. The model performs well at training data but does not generalize well to unseen data.
Underfitting	Occurs when the model is to simple to fit enough data to notice trends in features, causing the model to perform badly at training data as well as unseen data.
Hyperparameters	The parameters used to tune the network, such as learning rate and network size.
Dataset	A collection of data, usually split into three subsets for training, testing and validation data.

Batch	A collection of data samples from the dataset.
Epoch	One complete training run (of a neural network) through the entire dataset.

Notations

a	Scalar
$\mathbf{a}$	Vector
$\mathbf{A}$	Matrix
$a^{(L)}$	Layer index
x	Input
y	Output
α	Regularization weight factor
$\boldsymbol{\theta}$	Collection of parameters
η	Learning Rate
$\circ$	Element-wise vector product

Chapter 1

Introduction

1.1 Context

This project constitutes a master thesis in Design & Product Development at Linköping University, carried out in cooperation with SkyMaker AB [1]. SkyMaker is a company located in Linköping, Sweden, developing product generators and solutions to automate the process from user customized product all the way down to production.

1.2 Motivation

Mass production in fixed quantities are no longer the only mean to produce commercial products. Thanks to advancements in technologies such as additive manufacturing (3D-printing) and other highly automated manufacturing techniques a growing market for mass production of user customized products has emerged, locally and on demand. But customized products require customized models and machine instructions to produce, which is not always a trivial process. While open-source software and free CAD-tools have enabled some users to produce their own products it is still too complicated for the average customer, and hiring someone to do it for you can be very expensive.

Product generators partially solves this problem by presenting a set of know design parameters for the customer to play with. But as the design space grows larger so does the underlying system which needs to cope with all intermediate constraints.

Over the last couple of decades technological advancements have made it possible to collect and process large quantities of data. This is something that enables a more precise training of *Artificial Intelligence* using *Deep Learning* and has led to major breakthroughs in what areas artificial intelligence can be used. Since Google 2012 adapted the usage of deep neural networks to their voice search, they reported a big increase to the speech recognition accuracy (Sak et al., 2015). Other examples are IBM's Watson, a machine that played Jeopardy against human top performers and managed to win (Ferrucci et al., 2010), or Google's driverless car, Waymo [2], to mention a few. By using different AI-algorithms for generative design perhaps it would be possible to create customized products, without the cost of a designer or an engineer.

1.3 Aim

The goal with this thesis is to develop a framework for how deep learning and generative design can be applied to make the product design process faster, simpler or

[1] https://www.skymaker.se/
[2] https://waymo.com/

more effective; in order to create mass customized products, available and affordable for everyone. The work aims to guide and inspire companies to implement artificial intelligence in their product design process and act as a stepping stone which helps companies taking the first step into a new way of designing products.

1.4 Research Questions

By developing, implementing and validating a conceptual framework for generative design based on deep learning, implemented on a common construction product, the following research questions will be evaluated and answered:

1. How can generative design, powered by deep learning, be incorporated in the product design process?
2. How can the data required to train generative neural networks be obtained?
3. How can challenges in the implementation process be addressed?

1.5 Approach

On a general level the project work is split into four distinct phases: *Theory Study*, *Concept Development*, *Implementation* and *Closure*. Each phase builds upon the work in the previous phase and contributes to the final framework. A flow chart of the project process and how each phase relates to the framework can be seen in figure 1.1.

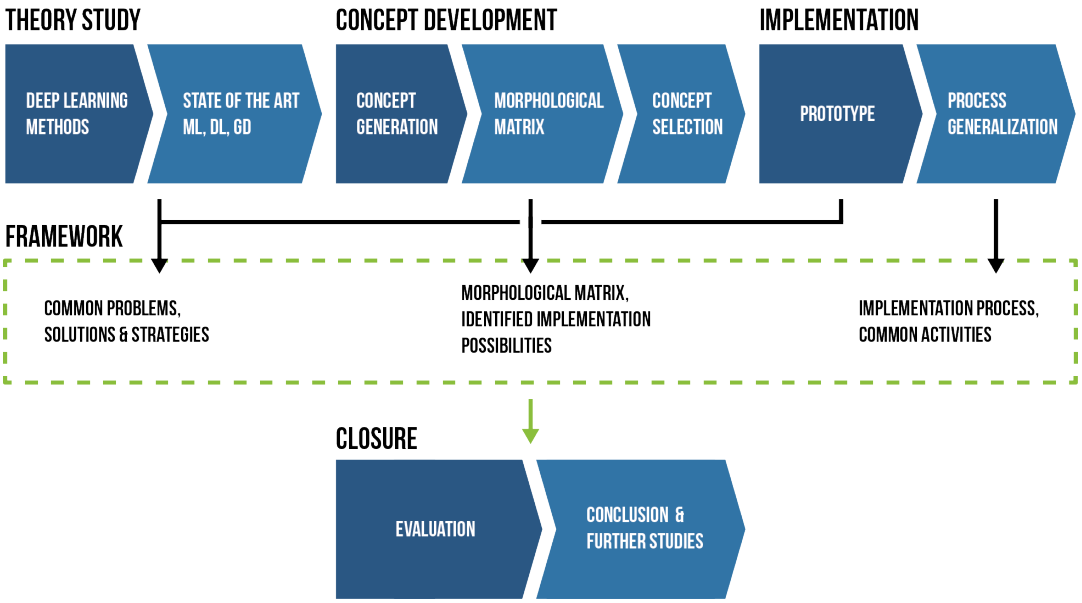

FIGURE 1.1: Flow chart of the project process at a high level with its four phases (Theory Study, Concept Development, Implementation and Closure) and how each part contribute to the framework (Note that the size of the blocks in this flowchart are arbitrary and bear no resemblance of the actual work required by each task.)

1.5.1 Theory Study

The *Theory Study* phase lays the foundation to the project through an in-depth study of existing technology and state of the art research within the field of Artificial Intelligence (AI), Machine Learning (ML) and Generative Design (GD) to form a knowledge base to ground the project on. The theory study also yields important results to the framework in form of identified common problems, solutions and strategies which is of value for the framework.

1.5.2 Concept Development

The *Concept Development* phase is an exploratory phase where potential applications of Generative Design Processes (GDP) and Deep Learning (DL) within the Product Design Process (PDP) are postulated and listed based on state of the art research and existing technologies. These potential implementations form a greater picture of how GDP and AI could automate and enhance the PDP as a whole, and which algorithms or methods that are used within each application or area today. These results are summarized in a morphological matrix of identified implementation possibilities which is another important result for the framework. Morphological matrix is a good starting point for end to end automation and to find appropriate solutions to different PDP tasks.

1.5.3 Implementation

The *Implementation* phase is the key development phase where a selected concept is further developed and implemented in a prototype demonstrating some use case and benefit of GDP targeted for a product design process. Based on the work of the prototype a general implementation process is formulated with the common activities and steps required to take when building a system of this kind. This process is one of the primary results for the final framework. The work with the prototype will also yield further understanding of the challenges in deep learning, how they are being handled today.

1.5.4 Closure

Lastly, the *Closure* phase wraps up the conceptual framework with the results from previous steps as well as evaluating the prototype. The overall approach and results are discussed and future studies and areas of interest are stated.

Chapter 2

Theory

The three main areas of theory relevant for this thesis are *Product Development* (section 2.1), *Generative Design* (section 2.2) and *Artificial Intelligence* (section 2.3) which are summarized in respective sections. The current state of the art and bleeding edge research within artificial intelligence lies in the area of *Deep Learning* with *Neural Networks*. Therefore the majority of the theory study will be focused on that subject covering the essentials of neural networks in section 2.4, types of deep learning in section 2.5 and common neural network architectures in section 2.6. Section 2.7 also present some recent implementations of AI related to product design.

Deep learning and neural networks overlaps and builds upon many other existing fields of mathematics and computer science, such as: *Linear Algebra, Calculus, Graph Theory, Statistics, Probability Theory, Pattern Recognition, Data Mining, Data Processing, Optimization and Visualization*.These topics will not be covered but are encouraged to be explored further by the reader. The most technical theories and terms in this chapter is however described in a simplified manner, making it understandable with only the fundamental knowledge.

2.1 Product Development

Product development is a very time consuming process and it often takes several years to develop a functional, attractive and working product that meet the needs of the customer (Ulrich and Eppinger, 2012). There are countless different strategies on how to make the product development process more efficient, depending on the conditions and objectives; some of them, such as SCRUM, involves an agile approach to counter fast changes (Ovesen, 2012); Others, such as Stage-Gate, has a more planned approach with different checkpoints (gates) to make sure that the development is going as planned and enable room for changes if needed (Cooper, 1990). Other commonly used strategies are the Design for "X" strategies such as Design for Environment (DFE) or Design for Manufacture (DFM) (Ulrich and Eppinger, 2012) . However all strategies seem to have some main activities and processes in common, even though their order, magnitude and implementation may differ. These activities and processes make up the main, general, building blocks for product development and are necessary in order to create a new product: Product specifications based on customer needs, concept development, design and functionality development, manufacturing and market release. By breaking down the building blocks into smaller blocks of commonly used generic activities you get the general blocks of product development shown in figure 2.1.

Throughout history different technical advancements have made parts of product development easier by introducing new tools or even automated assistance in

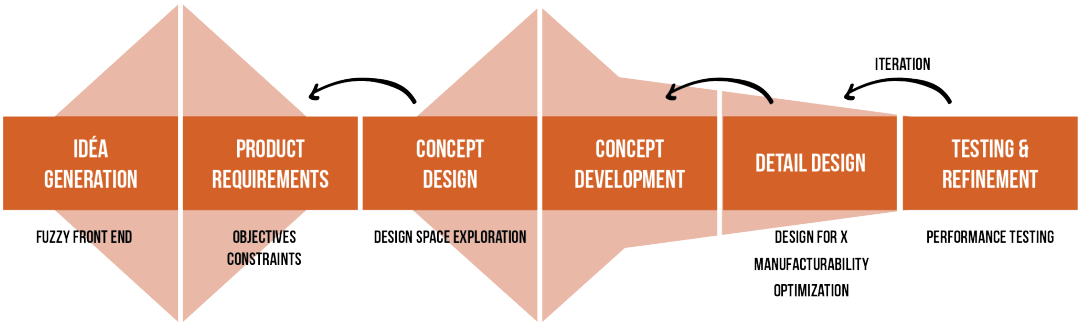

FIGURE 2.1: An example of a generic Product Development Process

design and manufacturing. Tasks previously done by hand are now done by machines. The first industrial revolution in the 18th to 19th century was highly fueled by the major technological breakthrough of the steam engine. Since then there has been two more industrial revolutions. The second came with the usage of oil and electrical power in the late 19th century and led to the invention of, for example, the light-bulb and the telephone. The third revolution (the digital revolution) began around 1980 and is still going on by the development of internet, computers and different technologies for information availability. We now stand on the edge towards the fourth industrial revolution. The fourth revolution comes with the use of artificial intelligence, nanotechnology, quantum computing, 3D-printing, internet-of-things (IoT) and much more. (Liu, 2017)

2.2 Generative Design

The subject of generative design is very broad and shows up in a range of different applications and fields such as art, architecture and product development. Despite the spread it is difficult to find a coherent definition of what it is, but listed below are some of the most common ones which we find relevant to the context of product development.

"Generative design systems are aimed at creating new design processes that produce spatially novel yet efficient and buildable designs through exploitation of current computing and manufacturing capabilities" - Kristina Shea

"Generative design is not about designing the building – Its' about designing the system that builds a building." – Lars Hesellgren

"An over arching computational method; in essence an incremental specification of design logic in a computational form that eventually yields with a design space open for exploration of alternatives and their variations." - Halil Erhan

Autodesk [1] have divides generative design into four categories: Form Synthesis, Lattice and surface optimization, Topology optimization and Trabecular Structures (Autodesk, 2018). The applications and use cases presented by Autodesk within these domains are primarily implemented with the objective of finding the most efficient geometrical structure for a certain situation, using as little material as possible. These geometries usually evolves into very complex and organic looking structures which, while perfectly solving the problem at hand, are not always optimal from

[1] https://www.autodesk.com/

a manufacturing or aesthetic standpoint, see figure 2.2. These optimization based methods are often powered by cloud computing to quickly cycle through and evaluate many thousands of design iterations in search for the perfect one.

FIGURE 2.2: Demonstration of a light weight vehicle part created with generative design by General Motors in cooperation with Autodesk (Danon, 2018)

Generative design is also often used to generate seemingly complex but beautiful art and designs by allowing seed data to evolve according to simple underlying rules, often resulting in structures similar to the ones we see in nature, see figure 2.3. A small change in initial conditions and tweaks to the ruleset can have dramatic effect on the end result and yield countless design variations.

FIGURE 2.3: 3D printed sculpture created with generative design by Nervous System, 2014

The challenge with generative design is to provide enough freedom for the system to explore and generate new designs, while coming up with the appropriate constraints and objectives for the design to be feasible.

2.3 Artificial Intelligence

AI is an umbrella term for any algorithm or system behaviour thought of as intelligent, such as pattern recognition, decision making and planning. Or as defined by Oxford Dictionaries (2018):

artificial intelligence [noun] *The theory and development of computer systems able to perform tasks normally requiring human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages.*

AI is commonly divided into two distinct categories: *Weak* (Applied) AI and *Strong* (General) AI where Weak AI encompasses and express only domain or task specific intelligence while Strong AI is the true general self aware AI commonly depicted in science fiction. To this day only weak AI has been created. Within the context of AI one of the most important fields of research is Machine Learning (ML) (section 2.3.1) which is the discipline of building computer systems with the ability to learn and effectively improve its performance on a task over time. The majority of all research related to AI is currently being developed within a sub section of ML called Deep Learning (DL) (section 2.3.2), which is currently outperforming any previous techniques. Artificial General Intelligence (AGI) (section 2.3.3) or Strong AI will likely develop from the research in DL but may also develop from other areas entirely which are yet to be discovered. An illustration of how these fields relate to each other can be seen in figure 2.4.

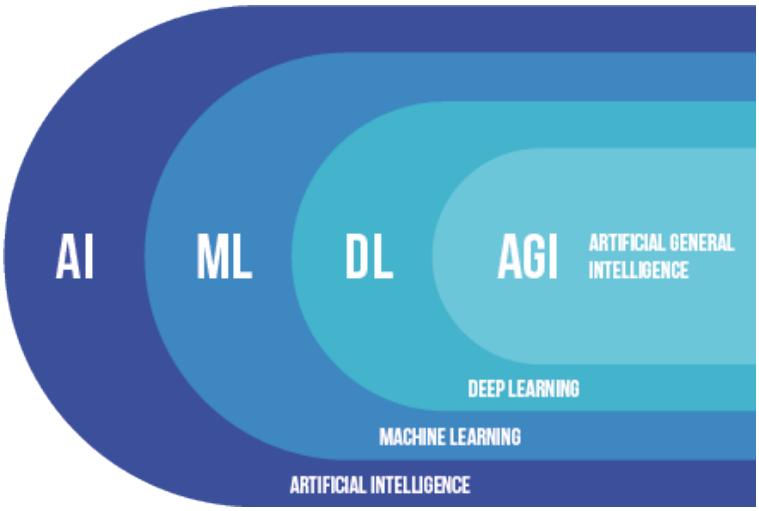

FIGURE 2.4: From AI to AGI and how it relates to ML and DL

2.3.1 Machine Learning

Ability to learn a specific task without being explicitly programmed. Algorithms whose performance improve as they are exposed to more data over time e.g: spam filters, chat-bots, search engines, recommendation systems, etc. Some common methods and techniques for ML are: Decision trees, Expert Systems, Support Vector Machines, Genetic Algorithms and Neural networks.

2.3.2 Deep Learning

Deep learning is a subsection of machine learning utilizing deep (artificial) neural networks, containing more than one hidden layer, allowing systems to learn more abstract patterns and concepts in data to excel at specific tasks. e.g: Image Recognition, Auto captioning, Speech synthesis, Natural Language Processing, etc. It is first in recent years ($>$ 2010) that deep learning has become truly feasible as a solution, thanks to faster and cheaper processors and distributed cloud computing. Some wide spread systems utilizing deep learning today are: self driving cars, voice assistants, search engines, with others.

2.3.3 Artificial General Intelligence

General versatile intelligence capable of adapting and solving many different problems in different domains, similar to humans. Is considered to be the holy grail of artificial intelligence. No such systems exist today but a lot of research is being made towards it by organizations such as DeepMind [2], OpenAI [3] and many others.

2.4 Neural Networks

Artificial neural networks are general function approximators (see section 2.4.1), constructed to mimic the functionality of the biological neural networks in our brain. An (artificial) neural network consists of a collection of nodes (neurons) and a set of links connecting the nodes to form a network. The typical node of an artificial neural network consist of three main components; a weighted sum of the input values, a bias and an activation function, see figure 2.5. The input values from each of the previous connected neurons are multiplied with a weight, unique for each connection, and then summed. To control how easily a node is activated a bias (positive or negative) is added to the weighted sum. Finally the bias and the weighted sum are passed to an activation function (e.g. Sigmoid, σ, section 2.4.2) returning the final value of the node, see equation 2.1. Both weights and biases are network parameters which are initially set and later tuned during training as the network learns. The full parameter set of a network is usually denoted by θ. In the common case every connection between any two nodes has an associated weight w and each node has an associated bias b.

$$y = \sigma(\sum_{i=1}^{n} w_i x_i + b) \tag{2.1}$$

[2] https://deepmind.com/
[3] https://openai.com/

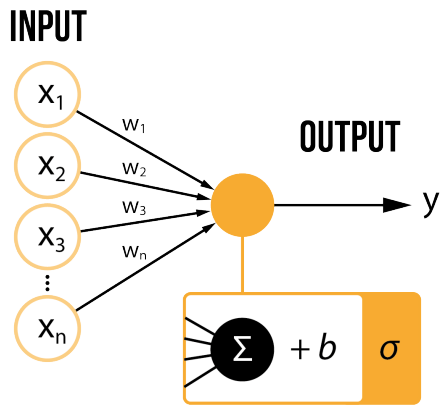

FIGURE 2.5: Illustration of an artificial neuron

The nodes are usually arranged in layers and a typical neuron network has at least three layers; one input layer, one output layer, and one or more hidden layers connecting the input to the output (see figure 2.6 for an example with two hidden layers). The activation of a full layer can be written in compact form as shown in equation 2.2, where $\mathbf{W}$ is a weight matrix containing all the weights associated with all connections to the previous layer and $\mathbf{b}$ a vector containing all the biases for each node in the layer.

$$\mathbf{y} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2.2}$$

The function for the entire network can then be written as a composed function as in equation 2.3, with superscripts representing the layer number (input layer = 0). One complete calculation of a networks output given some inputs are often referred to as a forward-pass, feed-forward operation or forward propagation.

$$\mathbf{y} = f(\mathbf{x}) = \sigma(\mathbf{W}^{(1)}\sigma(\mathbf{W}^{(2)}\sigma(...) + \mathbf{b}^{(2)}) + \mathbf{b}^{(1)}) \tag{2.3}$$

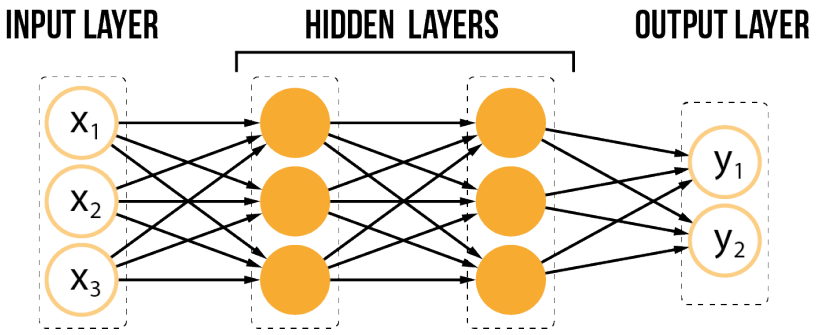

FIGURE 2.6: Illustration of a fully connected neural network with two hidden layers, input x and output y

2.4.1 The Universal Approximation Theorem

It has been formally proven by Cybenkot (1989) that neural networks with at least one hidden layer are capable of approximating *any* continuous function to any desired accuracy given enough neurons to work with. This implies that *any* problem possible to formulate as a function: regardless of complexity: could be solved given sufficiently large neural networks.

2.4.2 Activation Functions

The activation function is used to decide how much a neuron should be activated depending on the input value. If the activation function is linear there is no point having several layers in the network, since the last layer essentially will be a linear representation of the previous layers. Therefore the activation functions for deeper networks introduce a non-linearity to model more complex functions. Depending on the desired characteristics and what the network is designed to do different activation functions are used.

Listed below are some of the (as of 2018-03) most commonly used activation functions in their basic appearance. To solve specific flaws with each function several alternative versions of them have been created.

Sigmoid

The logistic Sigmoid (equation 2.4) has a s-shaped curve, see figure 2.7 and is commonly used for models that predict probabilities as it returns values between $(0, 1)$. The Sigmoid do however have some problems with vanishing gradients due to weak derivative, causing slow convergence and training in deeper networks (Maas, Hannun, and Ng, 2013).

$$Sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

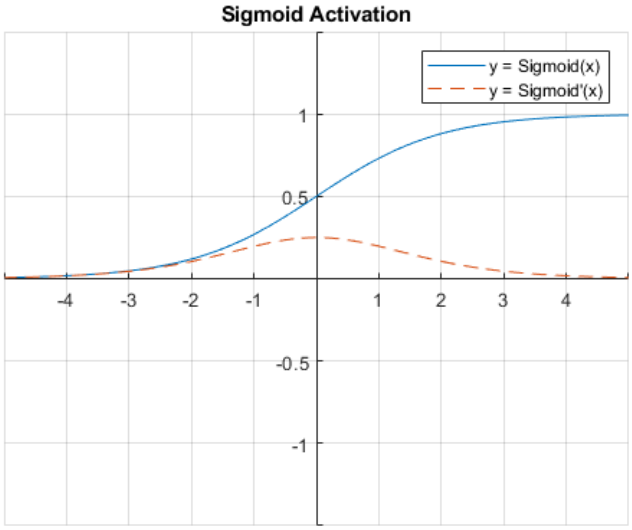

FIGURE 2.7: Graph of the Sigmoid function *(blue, continuous)* and its derivative *(orange, dotted)*

Tanh

Tanh (equation 2.5) is also a sigmoidal activation function and very similar to logistic Sigmoid with a s-shaped curve, but is centered at the origin without output range $(-1, 1)$ and a stronger gradient, see figure 2.8. But just like the logistic Sigmoid it

still has problem with vanishing gradients (Maas, Hannun, and Ng, 2013).

$$Tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.5}$$

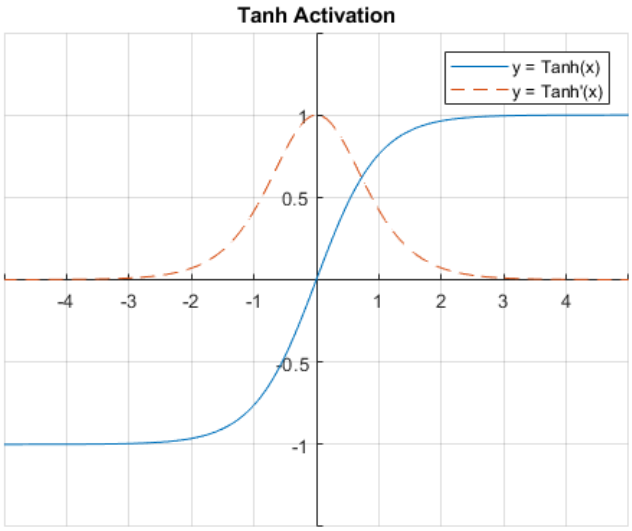

FIGURE 2.8: Graph of the Tanh function *(blue, continuous)* and its derivative *(orange, dotted)*

Rectified Linear Unit (*ReLU*)

The Rectified Linear Unit, *ReLU*, (equation 2.6) returns an output of zero for all negative input values and for all other values the output is equal to the input. *ReLU* reaches convergence much faster than logistic Sigmoid and Tanh and have no problem with vanishing gradients, which has made it one of the most used activation function for deep neural networks (He et al., 2015b). *ReLU* has also been proven by Maas, Hannun, and Ng (2013) and Nair and Hinton (2010), among others, to improve several models performances compared to sigmoidal functions. Because the gradient of all inactive neurons is zero, *ReLU* sometimes get a problem with "dead neurons", meaning that some neurons, once deactivated, may never be activated again (Maas, Hannun, and Ng, 2013). A graph of the *ReLU* function is shown in figure 2.9.

$$ReLU(x) = max(x, 0) = \begin{cases} x & x > 0 \\ 0 & \text{else} \end{cases} \tag{2.6}$$

Leaky Rectified Linear Unit (*LeakyReLU*)

LeakyReLU (equation 2.7) was introduced by Maas, Hannun, and Ng (2013) as a way to solve the dead neuron problem with *ReLU*. It is similar to *ReLU* but with an added slope, $\alpha = 0.01$, for all negative values, see figure 2.10.

$$LeakyReLU(x) = max(x, 0) = \begin{cases} x & x > 0 \\ \alpha * x & \text{else} \end{cases} \tag{2.7}$$

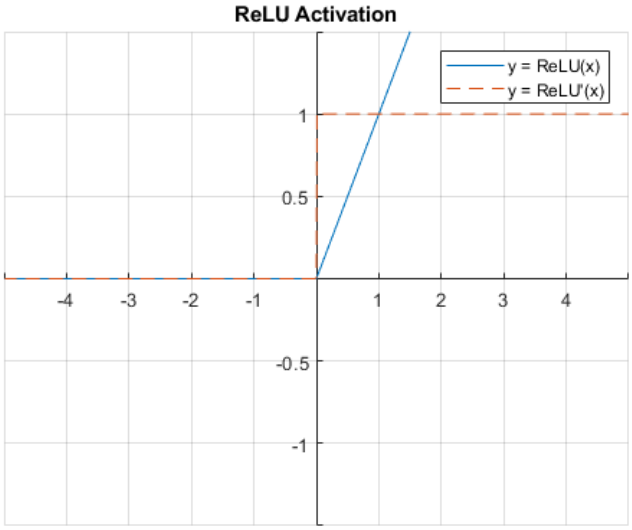

FIGURE 2.9: Graph of the *ReLU* function *(blue, continuous)* and its derivative *(orange, dotted)*

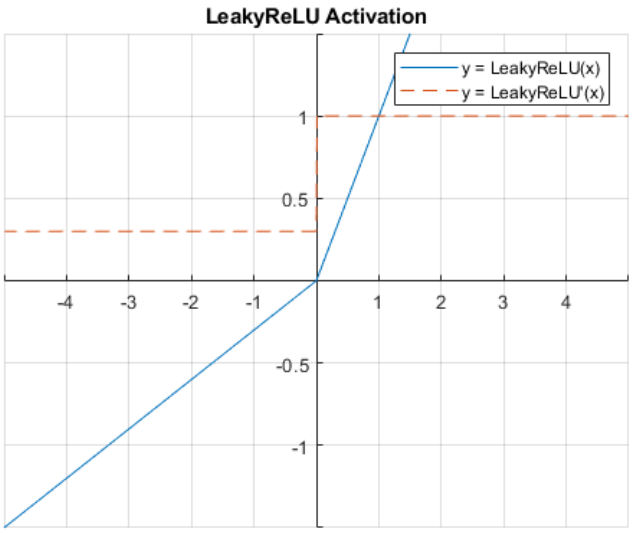

FIGURE 2.10: Graph of the *LeakyReLU* function *(blue, continuous)* and its derivative *(orange, dotted)*

SoftMax

The SoftMax function (equation 2.8) is a layer based activation function which normalizes all values to the range $(0, 1)$ and to a total sum of 1. These characteristics make SoftMax a good function for multi-class classification as the values can represent the probability of each outcome. For example: $SoftMax([2, 1, 0.1]) \rightarrow [0.7, 0.2, 0.1]$

$$SoftMax(x_k) = \frac{e^{x_k}}{\sum_{i=1}^{n} e^{x_i}} \tag{2.8}$$

2.4.3 How Neural Networks Learn

The network learns by optimizing its parameters $\boldsymbol{\theta}$ to either minimize or maximize the networks objective function. While this could be done using any optimization technique the most common approach is through gradient descent (or ascent) and variations of it. With the reason of being simple, computationally cheap, memory efficient and scalable even when the number of network parameters grow to orders of 10^7 and beyond which is very common.

The Objective Function

In order to measure how well a network is performing on a certain task an objective function is formulated. This function is commonly formulated as a loss function, L, which the network should try to minimize or as a reward function, R, which the network should seek to maximize. E.g. reconstruction loss, classification error etc. Some common functions include Mean Square Error (MSE) (equation 2.9) and (binary) Cross Entropy (log loss) (equation 2.10).

$$MSE(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|^2 \tag{2.9}$$

$$H(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{n} (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)) \tag{2.10}$$

Gradient Descent

Gradient descent is an iterative optimization method to minimize an objective function, $L(\boldsymbol{\theta})$, parameterized by $\boldsymbol{\theta} \in \mathbb{R}^n$. Each timestep the parameters are updated by following the gradient of the objective function with respect to the parameters (assuming the objective function is differentiable), see equation 2.11. This can be thought of as standing on a mountain (the objective function) and then taking a small step in the steepest direction downhill (see figure 2.11). After several iterations the parameters eventually converge to a local or global minima of the objective function. The size of each step is controlled by η, which in the context of machine learning is referred to as the learning rate. Selecting an appropriate learning rate (and tuning it while training) is one of the challenges in machine learning. (Ruder, 2017)

$$\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta} + \eta \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) \tag{2.11}$$

Many variations and extensions to the basic gradient descent has been developed to either improve accuracy, avoid stagnation around saddle points, increase computational efficiency and more. Some of the more popular includes SGD (Stochastic Gradient Descent), RMSprop (Hinton, Srivastava, and Swersky, 2014), Adam (Adaptive Momentum Estimation) (Kingma and Ba, 2015) and Nadam (Nesterov-accelerated Adaptive Moment Estimation) (Dozat, 2015). A more in-depth comparison of these algorithms and how they are implemented can be found in the paper: *An overview of gradient descent optimization algorithms* by Ruder (2017).

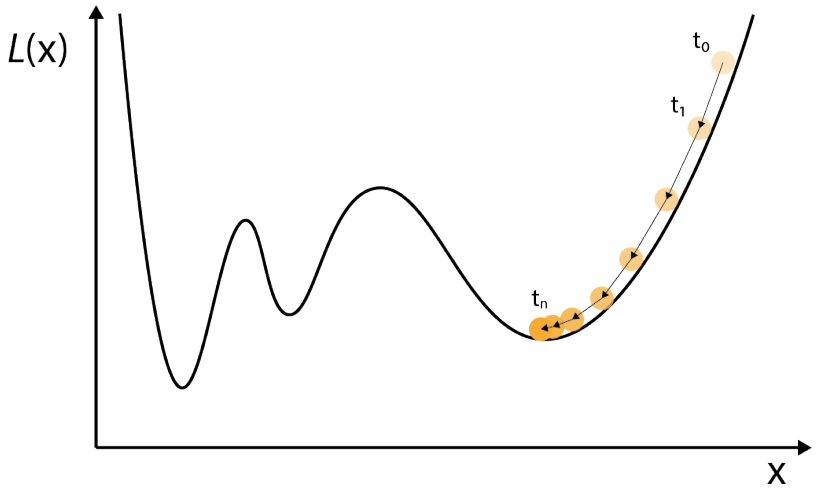

FIGURE 2.11: Visualization of gradient descent

Local Minima in Higher Dimensions

One of the challenges in optimization, especially with gradient decent is the problem of getting stuck in a local minima. In figure 2.11 it is easy to see how different initial values may result in convergence in one of several local minima. While this is a prominent problem in low dimensions it is not as big of a problem in higher dimensions R^n, as the likelihood of the function being convex in all dimensions at the same time at a point p decreases. In fact local minima are exponentially rare in high dimensions and the more prominent problem is instead saddle points as the ratio of saddle points to local minima increase exponentially with dimensionality (Dauphin et al., 2014). Dauphin et al. (2014) shows how the local minima of loss functions in high dimensions tend to cluster close to the global optimum and decrease exponentially in frequency away from it with network size. Dauphin et al. (2014) also argues that it is undesired to find the true global minima as it often leads to overfitting of the network and that a close local minima is to prefer.

Backpropagation

The Backpropagation algorithm is the backbone of todays deep learning system and is what allow deep artificial neural networks to learn. The goal with backpropagation is to calculate the partial derivatives of the networks objective function in respect to any parameter in the network (Rojas, 1996). This is not a trivial thing to do as the objective function depends on the networks output, which in turn depends on each layer-wise operation performed by the network. So to figure out how much a specific parameter in a specific layer far back in the network should be tuned to decrease the overall loss function or increase the reward is difficult. Backpropagation solves this by propagating the error (or desired change of the output y) back through the network and calculating the partial derivative using the chain rule. This is then used to update all parameters using gradient descent. This requires both the objective function and the entire network to be differentiable.

If the objective (loss) function is denoted by L and the target network is parameterized by $\boldsymbol{\theta} \in \mathbb{R}^k$, the partial derivative of the objective function with respect to a parameter $\boldsymbol{\phi} \in \boldsymbol{\theta}$ can be written as equation 2.12. Where superscripts denote layer indexes, subscripts the specific node in the layer, a the nodes activation function, and

the nodes value function prior the activation (which for a dense network is defined such that $a(z)$ corresponds to equation 2.1).

$$\frac{\partial L}{\partial \phi_j^{(L)}} = \sum_{i=1}^{n^{(L+1)}} \frac{\partial L}{\partial a_i^{(L+1)}} \frac{\partial a_i^{(L+1)}}{\partial z_i^{(L+1)}} \frac{\partial z_i^{(L+1)}}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial \phi_j^{(L)}} \tag{2.12}$$

2.4.4 Hyperparameters

Hyperparameters are a collective name for the parameters controlling the architecture and behaviour of the neural network or deep learning system but which are not learned. E.g. learning rate, batch size and layer sizes. Some parameters can be tuned during runtime such as the learning rate but others are usually left static. How to set, tune and optimize these hyperparameters effectively are on-going research which currently are primarily limited by computational complexity, as large networks take long time to train and many iterations are required to perform an optimization. Some state of the art optimization methods for hyperparameters include Hyperband (Li et al., 2017) and Bayesian optimization methods.

2.4.5 Data

The data passed to a neural network can either be on the form of samples (e.g. images, text strings) or continuous (e.g. video, music, text feed) depending on the type of network. A collection of samples are called a dataset. Networks operating on samples, such as image classifiers, are usually implemented to handle multiple forward-passes in parallel, in these cases the entire dataset is passed to the network, and a matrix of all classification evaluations are returned. If the dataset is very large (in memory), containing many thousand of datapoints, it is common to divide the dataset into smaller chunks called batches to take up less system memory at once. Radiuk (2017) showed that large batch sizes greatly increase the accuracy of CNNs on image recognition. Larger batch sizes also allow the system to go through the dataset faster resulting in faster training, so the batch size is usually set empirically to the highest value possible without running out of memory. The dataset is typically split into a training set and a test set, where the model trains on the training set and the test set is used to evaluate how well the model has generalized to new data. In some cases the data is also split to a third set, a validation set which is used when tuning the model architecture. The training dataset is typically about $60 - 90\%$ of the the full dataset with the rest used for testing and validation.

A list of some commonly used public datasets, especially for object detection are listed in table 2.1, but many more exist.

Dataset	Description	Classes	Img Size	Samples
MNIST	Handwritten digits	10	28x28	60k + 10k
COCO	Common objects in context	80	misc	330k
Open Images V4	Very diverse set of objects	>600	misc	>1.74M
CIFAR-100	Common objects	100	32x32	50k+10k
CIFAR-10	Common objects	10	32x32	50k+10k

TABLE 2.1: Some examples of commonly used public datasets available online

2.4.6 Regularization

Regularization are a collective name for any modification that is made to a learning algorithm with the intent to reduce the algorithms generalization error but not its training error (Goodfellow, Bengio, and Courville, 2016). A regularized objective function $\tilde{J}$ is typically written on the format in equation 2.13; with J being the regular objective which reduces the training error, Ω denoting a regularization function and α a hyperparameter determining the relative contribution of the regularization term to the objective function (Goodfellow, Bengio, and Courville, 2016).

$$\tilde{J}(\boldsymbol{\theta}, \mathbf{x}, \mathbf{y}) = \mathbf{J}(\boldsymbol{\theta}, \mathbf{x}, \mathbf{y}) + \alpha\Omega(\boldsymbol{\theta}) \tag{2.13}$$

Regularization is one way to solve the overfitting problem that can occur when training a machine learning model, meaning it learns unwanted features in the training data and therefore generalizes poorly outside the dataset. Below are some commonly used and well performing ways of regularizing a neural network.

Dropout

The dropout technique presented by Hinton et al. (2012) is a way to reduce overfitting in a neural network through regularization. Dropout means that each hidden neuron is given a probability that the output will be set to zero, and thereby dropped out. This leads to a different network architecture every time the input changes. Which in turn forces the neurons to learn features based on the input from several different neuron since they cannot depend on a specific neuron being active. An illustration of Dropout is shown in figure 2.12.

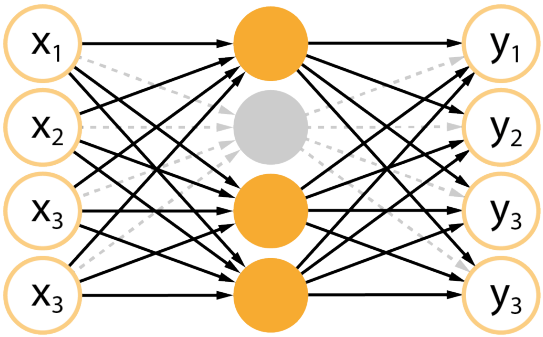

FIGURE 2.12: Illustration of Dropout with dropped node and connections in grey

DropConnect

DropConnect, introduced by Wan et al. (2013), builds on Dropout presented by Hinton et al. (2012), but instead of setting the neuron output to zero DropConnect randomly breaks connection between neurons, see figure 2.13. Since there often are more connections than neurons in a network DropConnect gives the possibility to get even more different architectures. Wan et al. (2013) shows that DropConnect in many cases perform better but is a bit slower than Dropout.

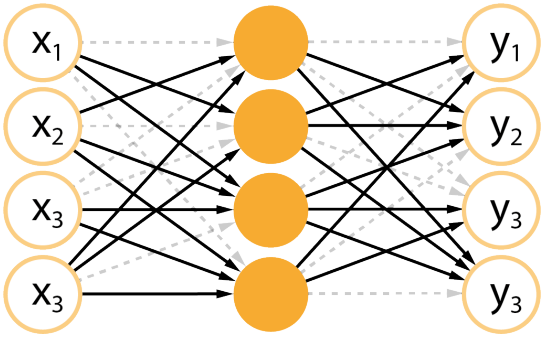

FIGURE 2.13: Illustration of DropConnect with dropped connections in grey

Kullback-Leibler Divergence

Kullback-Leibler (KL) divergence (also called relative entropy) is defined according to equation 2.14 and is a measurement of the divergence of one probability distribution to another (Kullback and Leibler, 1951). The KL Divergence of a distribution P from a reference distribution Q over the same variable x is written $D_{KL}(P||Q)$ and is often used as a regularizing term in the objective function of neural networks to drive a learned distribution towards a desired distribution (e.g. in VAEs, section 2.6.12).

$$D_{KL}(P(x)||Q(x)) = \sum_{x \in X} P(x) \log \frac{Q(x)}{P(x)} \tag{2.14}$$

If both distributions are equal the KL Divergence is zero. While the KL Divergence often is referred to as a distance it is important to keep in mind that the measure is non-metric. The function is also asymmetric, so $KL(p||q)$ does not necessarily equal $KL(q||p)$. The KL Divergence can also be written on continuous form as in equation 2.15. (Kullback and Leibler, 1951)

$$D_{KL}(P(x)||Q(x)) = \int_{-\infty}^{\infty} P(x) \log \frac{Q(x)}{P(x)} dx \tag{2.15}$$

L^2 Parameter Norm

L^2 Parameter Norm (equation 2.16), also known as *weight decay*, is a regularization strategy to drive the center of the weights distribution of the network close to zero by penalizing the squared magnitude of the weights vector, $\mathbf{w}$, (any weight matrix $\mathbf{W}$ can be reshaped to a vector, $\mathbf{w}$). (Goodfellow, Bengio, and Courville, 2016)

$$\Omega(\theta) = \frac{1}{2}||\mathbf{w}||_2^2 = \frac{1}{2}\mathbf{w}^T\mathbf{w} \tag{2.16}$$

L^1 Parameter Norm

L^1 Parameter Norm (equation 2.17) is another regularization strategy, which similar to L^2 is meant to reduce the magnitude of the weights, $\mathbf{w}$. But in comparison, L^1 Norm instead penalizes the sum of the absolute values of the individual parameters. L^1 Norm has been shown to have a sparsifying effect causing a subset of the weights to become zero. (Goodfellow, Bengio, and Courville, 2016) (Goodfellow, Bengio, and Courville, 2016)

$$\Omega(\theta) = ||\mathbf{w}||_1 = \sum_{\forall i} |w_i| \tag{2.17}$$

2.4.7 Input Normalization

Unless a certain set of input features are known to be more/less significant than others it is recommended to normalize all features to the same range so they can be compared properly. If the orders of magnitude between two features would differ a lot one would easily diminish the other and slow down learning if the small value is significant. Convergence usually happens faster if the average of each input variable over the training set is close to zero. It has also been shown that training converges faster if the inputs also are scaled to have approximately the same covariance (commonly set to one). (LeCun et al., 1998b)

Input normalization is commonly implemented by Min-Max scaling, equation 2.18, or z-score normalization, equation 2.19, but other variations exists as well (Li, Chen, and Huang, 2000).

$$z_i = \left(\frac{x_i - min(\mathbf{x})}{max(\mathbf{x}) - min(\mathbf{x})}\right)(max(\mathbf{z}) - min(\mathbf{z})) + min(\mathbf{z}) \tag{2.18}$$

$$z_i = \frac{x_i - E(\mathbf{x})}{\sqrt{Var(\mathbf{x})}} \tag{2.19}$$

2.4.8 Batch Normalization

Batch Normalization is a technique to accelerate deep network training by reducing Internal Covariate Shift (Ioffe and Szegedy, 2015) (reducing how much internal values vary given new data). By normalizing input data across the batch to a mean of β and a variance γ between layers (see equations 2.20 and 2.21), succeeding layers will be less dependent on changes in the earlier layers. This weakens the coupling between layers and allow deeper architectures to learn quicker. Both β and γ are trainable parameters allowing the network to learn the optimal distribution. Batch normalization also has a weak regularizing effect as equation 2.20 effectively introduces noise in the data, which makes the model less likely to overfit. But its effect is far less than Dropout (section 2.4.6) and is also more prominent with smaller batch sizes.

$$y_i = \gamma_i \hat{x}_i + \beta_i \tag{2.20}$$

$$\hat{x}_i = \frac{x_i - E[x_i]}{\sqrt{Var[x_i]}} \tag{2.21}$$

2.5 Deep Learning Methods

Several different techniques on how to tackle the machine learning problem has been proposed in the past and new ones are still emerging. In general they can be divided into three types of learning: *Supervised*, *Unsupervised* and *Reinforcement* based learning.

2.5.1 Supervised Learning

This is the most common learning technique. The concept of supervised learning is to present the AI to a labeled dataset, meaning that some information about the data is given. For example an image of a dog where the dog is marked with a square. The AI then predicts if there is a dog in the image and where it is located and finds out if it was right. If the prediction was wrong the weights are adjusted through backpropagation, see section 2.4.3. By iterating this through many labeled images the AI learns to recognize features connected to a dog and can make more qualified predictions.

2.5.2 Unsupervised Learning

Unsupervised learning differs from supervised learning since it does not require the data to be labeled. In unsupervised learning the AI is presented to an unlabeled dataset, which it "analyzes" to find patterns and similarities between data. The concept builds on the idea that the AI does its own categorization and clustering of features and thereby more complex features and patterns can be found. An example of an algorithm using unsupervised learning is the Auto Encoder, see section 2.6.11.

2.5.3 Reinforcement Learning

Reinforcement learning is a technique that tries to mimic the way humans learn. The AI is given an environment with which it can interact. The goal for the AI is to get as high reward as possible and depending on what action it performs, based on the current state, different rewards are given. Through trial and error the AI progressively learns what actions to take in any given situation in order to maximize the reward. The decision is typically controlled by a policy $\pi(\mathbf{a}|\mathbf{s})$ which returns the probability of taking a certain action $\mathbf{a}$ given a state $\mathbf{s}$. This policy is continuously updated by maximizing expected future reward. In reinforcement learning the AI is commonly referred to as an agent (see figure 2.14 for an illustration of the reinforcement learning process). Reinforcement learning is currently the bleeding edge of machine learning where a lot of research is being made and new results are frequently being published (e.g. OpenAI [4]).

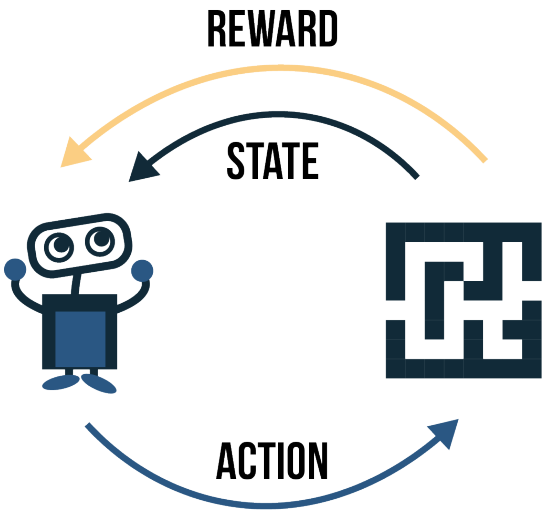

FIGURE 2.14: Illustration of the reinforcement learning process

[4] https://openai.com/

2.6 Common Neural Network Architectures

Neural networks exist in many shapes and forms with different strengths and weaknesses. The research to improve performances is intense and fast-moving, leading to variations and combinations of networks being published every other week. In the following sections the most commonly used generative architectures are presented; which many others derive from.

2.6.1 Feed Forward Neural Networks (FFNN)

A Feed Forward Neural Network is the simplest type of neural network consisting of an input layer, an output layer, and one or more fully connected hidden layers (see figure 2.15, right). Data flows only in the forward direction, and the network is usually trained through backpropagation. The simplest, still practical, form of a neural network is called a Perceptron and consists of only two input nodes directly tied to an output node (see figure 2.15, left). Given sufficient hidden nodes, a FFNN can theoretically approximate any arbitrary function mapping x to y, see section 2.4.1.

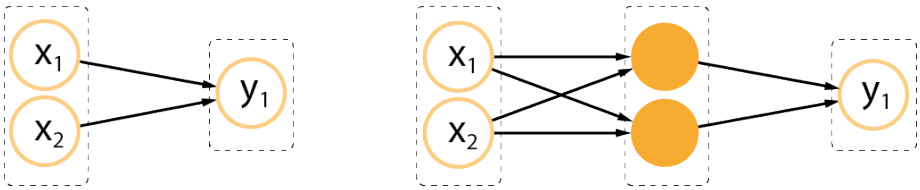

FIGURE 2.15: Example network topology of a Perceptron (left), and a shallow Feed Forward Neural Network with one hidden layer (right)

2.6.2 Deep Residual Network (DRN)

Deep Residual Networks was presented 2015 by He et al. (2015a) as a way to tackle a common problem that occurs in really deep networks. When adding many hidden layers to a deep network it can be a problem with degradation of the training accuracy, which leads to a higher training error if more layers are added.

By adding shortcut connections (see figure 2.16), that perform an identity mapping, a few layers apart throughout the network He et al. (2015a) managed to avoid the degradation problem and created a network (ResNet-152) with 152 layers; which outperformed the current state of the art networks in image recognition, and still had a lower computational complexity. After further research He et al. (2016) presented a network with 1000 layers that further improved accuracy. The structure of the model presented also showed a linear computational complexity, meaning that the difficulties of training of a deep network does not increase exponentially when adding layers.

2.6.3 Recurrent Neural Networks (RNN)

Recurrent Neural Networks are FFNNs where the hidden layer activations not only depends on the input from the previous layer but also from its own activation last time it fired, see figure 2.17. This feedback loop allows the network to encode time and sequence dependent information; which is great for handling streams of data where the action according to the current state depends on what has happened in the

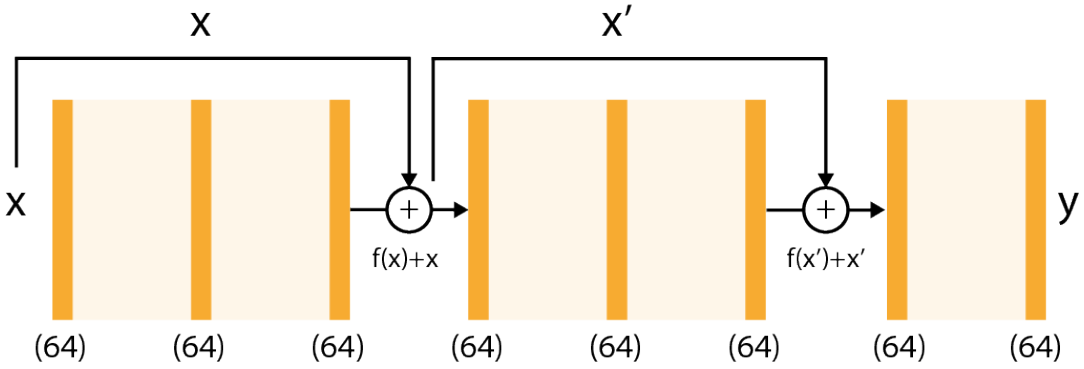

FIGURE 2.16: Example network topology of a Deep Residual Network with shortcut connections and FC layers containing 64 nodes each

past; such as meaning of speech, text and video. One known problem with RNNs is that the feedback loop causes *vanishing* and *exploding* gradients during training and operation; which causes the network to loose time dependent information too quickly, similar to the *vanishing gradients problem* faced in backpropagation of deep neural networks. Some insight into these problems as well as potential solutions within the area of music prediction and language modelling have been proposed by Pascanu, Mikolov, and Bengio (2012).

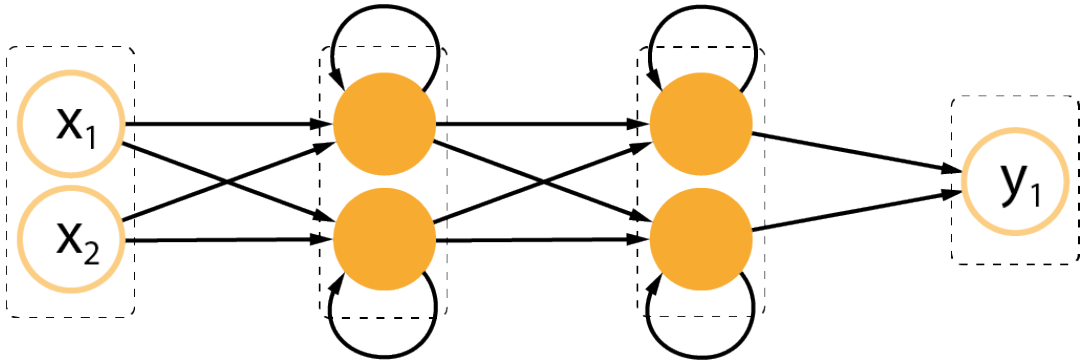

FIGURE 2.17: Example network topology of a Recurrent Neural Network with two hidden layers

2.6.4 Convolutional Neural Networks (CNN)

The Convolutional Network, LeNet, was introduced by LeCun et al. (1998a) and was used for pattern recognition in handwritten characters. It presented a new structure of the neural network, which since has been one of the most, if not the most, commonly used structure for image classification, see figure 2.18. Basically a Convolutional Network consists of four main parts: First convolution layer(-s) (see section 2.6.5), followed by a non-linear activation function (see section 2.4.2), then a pooling layer (see section 2.6.6), or a sub-sampling layer (see section 2.6.7), and finally one or more fully connected layers (LeCun et al., 1998a). The convolution layers are used to extract features from the image and can be thought of as detecting consecutively higher and higher orders of features for each layer (e.g. edges, groups of edges, nose/eyes, a face, a person). The final fully connected layer are used for classification of the image (LeCun et al., 1998a). Deep convolutional architectures can

also be used without fully connected layers at the end for image parsing or image generation (Radford, Metz, and Chintala, 2015).

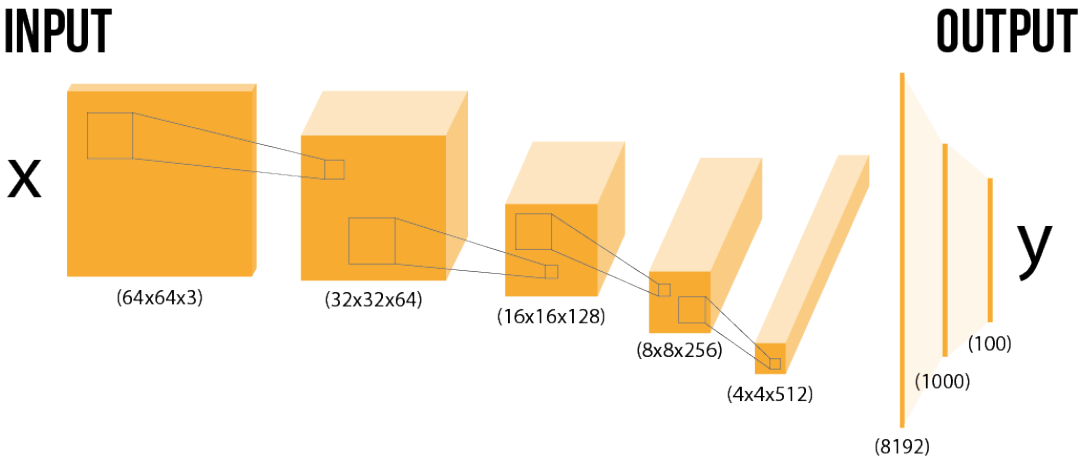

FIGURE 2.18: Example topology of a Convolutional Neural Network with four convolutional layers followed by flattening and three FC layers

2.6.5 Convolutional Layers

Compared to flat fully connected layers a convolutional layer operates on tensors instead of vectors. These tensors could be of any shape but are typically three dimensional volumes (m, n, c) representing mxn images in c channels. An RGB image can then be represented with a $(m, n, 3)$ tensor.

The convolutional layer is composed of a series of filters (convolution kernels), which are convolved with the input tensor to calculate feature maps; one for each filter. Each filter has a small receptive field only covering a portion of the input data and performs a dot product with its own weights and adds a bias to calculate a value for the feature map. The filter then moves and repeats the operation until the input tensor has been covered (see figure 2.19). The shape of the filter is typically square, $(3, 3)$ or $(5, 5)$ and are covering all channels, but other shapes can be used. How much the filter is translated between calculations is called Stride and is typically set to $(1, 1)$ (see figure 2.20). (Gu et al., 2017)

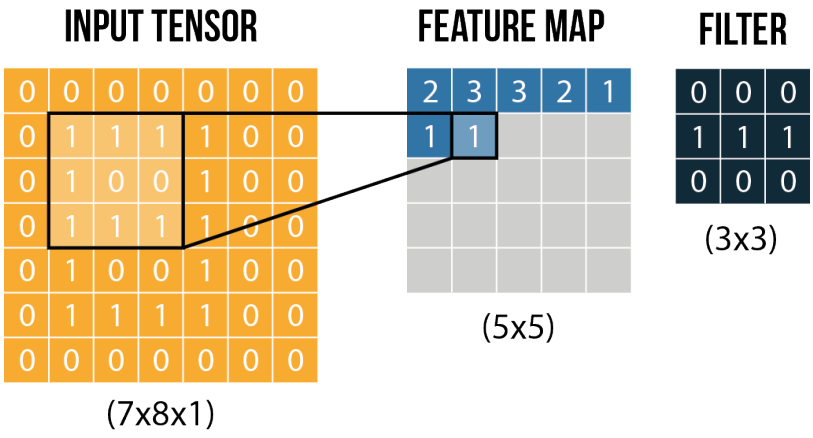

FIGURE 2.19: Explanatory example of a convolution operation using a single filter with a stride of one, no padding and a bias of 0

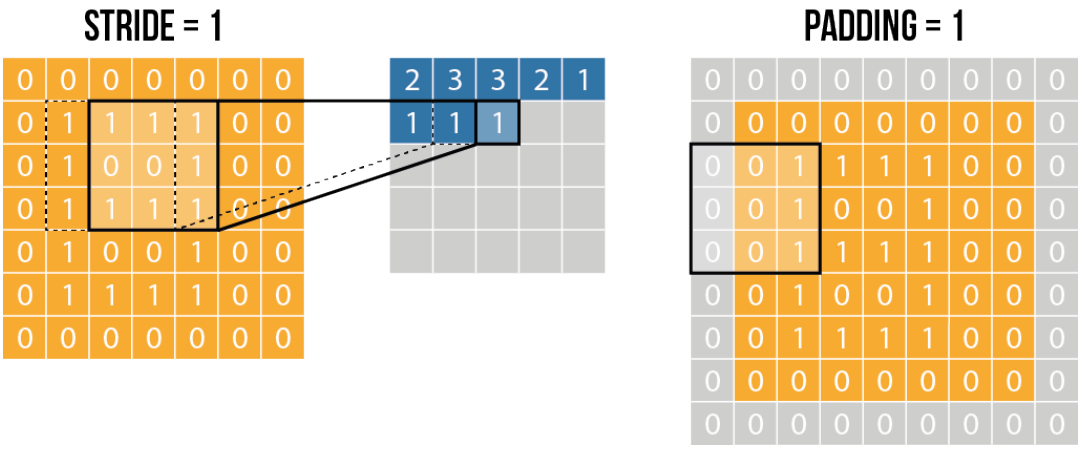

FIGURE 2.20: Explanatory example of striding and padding. Stride (left) defines how much the filter moves each step, padding (right) defines how much the filter can overlap outside the input tensor. The example to the right shows a zero-padding of size 1.

A single filter with a kernel of $(3,3)$ and a stride of $(1,1)$ applied to a $(10,10,3)$ image tensor results in a single feature map of shape $(8,8,1)$, similarly 12 filters would result in an output tensor of shape $(8,8,12)$. To prevent the resulting tensor from decreasing in width or height a padding can be added, which allows the filter to overlap the tensor boundary (see figure 2.20) (Dumoulin, Visin, and Box, 2018). A zero-padding of $(1,1)$ would in our previous example yield a final output shape of $(10,10,12)$. The calculation of the activation $a_{i,j,f}$ in layer L and feature map f with a kernel of size k, weights $\boldsymbol{W}$ and bias b can be expressed as equation 2.22 where $\circ$ is the element-wise Hadamard product.

$$a_{i,j,f}^{(L)} = \sum \mathbf{X}_{i\pm k,j\pm k}^{(L)} \circ \mathbf{W}_{f}^{(L)} + b_{f}^{(L)} \tag{2.22}$$

Convolutional layers benefit greatly from GPU accelerated training as the feature map calculations can be done in parallel (Brown, 2015).

2.6.6 Pooling Layers

Pooling layers are used to reduce the spatial size of a feature representation; to reduce the amount of parameters and computations required by the network. A pooling operation of size $(2,2)$ groups input data in shapes of $(2,2)$ and then calculates a single value for each group for the output. E.g. an input of shape $(6,6,3)$ will result in an output of shape $(3,3,3)$. The most common form of pooling is Max Pooling (equation 2.24) which returns the largest value from each group, but other variations exist as well, such as Average Pooling (equation 2.23) (Dumoulin, Visin, and Box, 2018). An example of Max Pooling can be seen in figure 2.21. A Pooling layer is typically added after one or more convolutional layers (see section 2.6.5).

$$AveragePooling : a_{i,j,f}^{(L)} = \frac{1}{k^2} \sum \mathbf{X}_{i\pm k,j\pm k}^{(L)} \tag{2.23}$$

$$MaxPooling : a_{i,j,f}^{(L)} = max(\mathbf{X}_{i\pm k,j\pm k}^{(L)}) \tag{2.24}$$

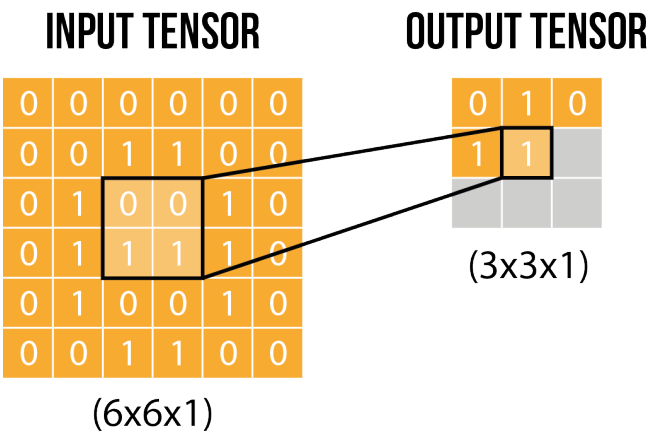

FIGURE 2.21: Example of a Max Pooling operation to a $(6, 6, 1)$ tensor with a pooling size of $(2, 2)$

2.6.7 Strided Convolutions

An alternative of using Pooling layers are to use Convolutional layers but with a Stride > 1. This is commonly referred to as a *strided convolution*. By using a Stride of 2 and add zero-padding where necessary a strided convolutional layer can effectively reduce the layer size from e.g. $(6, 6, 1)$ to $(3, 3, 1)$ just like the pooling layer would (see figure 2.22). The key benefit of strided convolutions over pooling is that it has trainable parameters, meaning that the network can learn how to perform the pooling operation (Springenberg et al., 2015), which has been shown to produce better results (Radford, Metz, and Chintala, 2015).

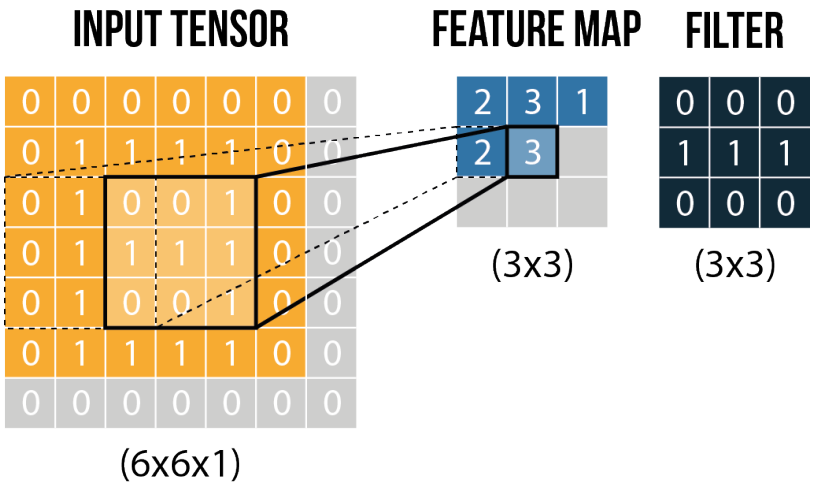

FIGURE 2.22: Example of a convolution operation with stride of 2 resulting in a downsampling by a factor of 4 similar to a pooling operation. Here using a single filter of size $(3, 3)$ with $b = 0$ and zero-padding one unit right and bottom.

2.6.8 Upsampling Layers

Upsampling layers are prominent in Generator type of networks where the desire is to return an output of larger shape than the input, e.g. $(8, 8, 3)$ to $(16, 16, 3)$. This can be achieved in several different ways; one being Unpooling which is an approximate

inverse of the Pooling operation (section 2.6.6). Two types of unpooling are repeated values (used by Keras [5]) and zero-padding, which are both shown in figure 2.23. It is also possible to use transpose strided convolutions as a trainable unpooling (Dumoulin, Visin, and Box, 2018), (Radford, Metz, and Chintala, 2015).

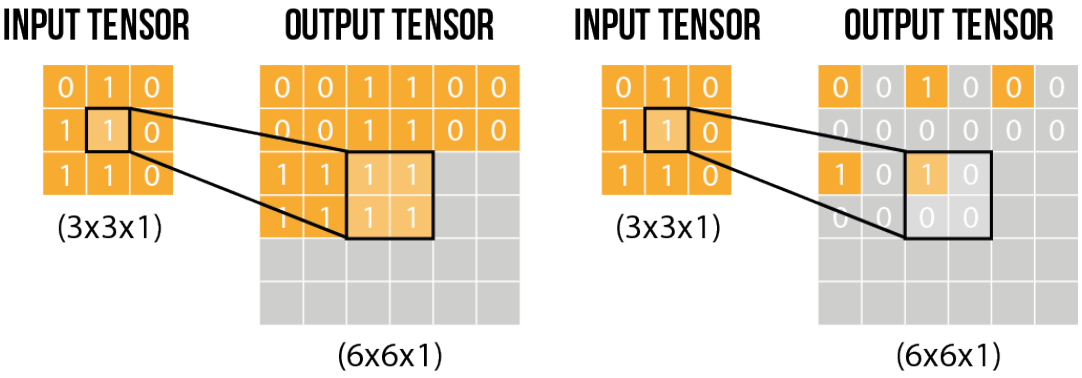

FIGURE 2.23: Example of Unpooling using repeated values (left) and zero-padding (right). Both using an unpooling size of $(2, 2)$.

2.6.9 Generative Adversarial Network (GAN)

Generative Adversarial Networks (GAN) was introduced by Goodfellow et al. (2014) as an new approach to creating generative models. A Generative Adversarial Network consists of two networks that are pitted against each other. The first network (the Generator) is a generative model trying to generate a counterfeit sample of the training data. The other network (the Discriminator) is a discriminative model trying to distinguish between the real data and the samples generated by the Generator. The Generator network takes a random noise vector $\boldsymbol{z} \sim p_z(\boldsymbol{z})$ as input and returns a generated data sample as output. The Discriminator network takes a data sample as input and returns a probability estimate of its validity ($p \in [0, 1]$). GANs operating on images are typically built as CNNs (see section 2.6.4) but it is not a requirement.

Training GANs

The two networks are trained simultaneous and as the Discriminator gets better at differentiating between the generated and the real data the Generator is forced to create better counterfeits in order to fool the Discriminator, see figure 2.24.

In practise this is done in two steps:

1. First the Discriminator network is trained to discriminate between real and generated data by outputting the correct probability of it being real (0 for generated, and 1 for real). This is done by minimizing the cross-entropy loss (like with binary classification) using a half batch of real samples $\boldsymbol{x} \sim p_x(\boldsymbol{x})$ and a half batch of generated data $\boldsymbol{x}' = G(\boldsymbol{z} \sim p_z(\boldsymbol{z}))$, as expressed in equation 2.25. During this step the Generator network is frozen and only the parameters of the Discriminator are updated through backpropagation.

2. Secondly the Generator is trained to fool the Discriminator to label the generated images as real. This is done by minimizing the negative of the Discriminators loss function (see equation 2.26) with all parameters of the Discriminator

[5] https://keras.io/layers/convolutional/#upsampling2d

network frozen and only updating the parameters of the Generator. Essentially using the gradient of the Discriminator to improve the Generator.

This Discriminator vs Generator game can be summarized as a ***minimax*** game of the value function $V(\boldsymbol{\theta_D}, \boldsymbol{\theta_G})$, see equation 2.27 as expressed in the original paper by Goodfellow et al. (2014).

$$L_D(\boldsymbol{\theta_D}, \boldsymbol{\theta_G}) = -\frac{1}{2}E_{\boldsymbol{x}\sim p_x(\boldsymbol{x})}[\log[D(\boldsymbol{x})]] - \frac{1}{2}E_{z\sim p_z(z)}[\log(1 - D(G(\boldsymbol{z})))] \tag{2.25}$$

$$L_G(\boldsymbol{\theta_D}, \boldsymbol{\theta_G}) = -L_G(\boldsymbol{\theta_D}, \boldsymbol{\theta_G}) \tag{2.26}$$

$$\min_G \max_D V(D, G) = E_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + E_{z\sim p_z(z)}[\log(1 - D(G(\boldsymbol{z})))] \tag{2.27}$$

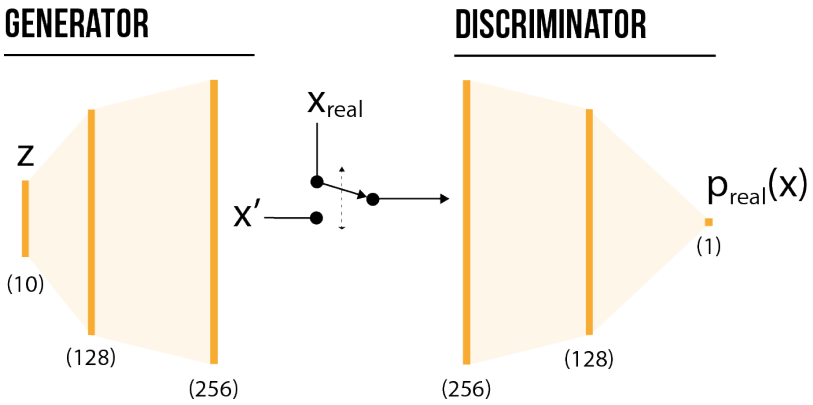

FIGURE 2.24: Example topology of a Generative Adversarial Network with fully connected layers

GANs are Unstable

For training to work properly it is important that the Generator does not outperform the Discriminator. If it happens, the Generator may start to converge towards only producing a handful of data samples, which are known to fool the Generator (Goodfellow et al., 2014) (a behaviour called *mode-collapse*). This is typically handled by updating the Discriminator k times before updating the Generator, allowing it to stay ahead. But, it is not allowed to become too good either, because then L_D falls to zero and the gradients for the Generator vanishes. This makes GANs notoriously difficult to train as the two networks need to stay synchronized even though they, by definition, are trying to outperform each other. Thankfully new improved versions have been developed which attempts to solve this caveat once and for all and one of those improved models is the Wasserstein GAN.

2.6.10 Wasserstein GAN (W-GAN)

The Wasserstein GAN (W-GAN) by Arjovsky, Chintala, and Bottou (2017) improves upon the original GAN architecture by introducing an improved objective function

and architecture. These changes allow the GANs to be trained stably, without the need for careful synchronization between Generator and Discriminator networks.

These improvements also drastically reduce the mode-collapse problem of ordinary GANs and makes it more robust to variations of network architecture. The W-GAN loss correlates with the observed sample quality from the Generator, which gives a practical metric of how well the W-GAN is doing and improves over time, a measure ordinary GANs do not have. (Arjovsky, Chintala, and Bottou, 2017)

W-GAN does not use a Discriminator to output a probability of validity. Instead it uses a Critic network which approximates the Earth Mover distance $W(p_r, p_g)$ between a fake distribution $p_g(\mathbf{x})$ (for a batch of generated samples) to the real distribution $p_r(\mathbf{x})$ (for a batch of real samples). The Earth Mover (EM) distance (also known as Wasserstein distance), defined as equation 2.28, is the minimal cost (work) of moving mass from x to y in order to turn the distribution p_r into p_g (Arjovsky and Bottou, 2017).

$$W(p_r, p_g) = \inf_{\gamma \prod(p_r, p_g)} E_{(x,y)\sim\gamma}[||x - y||] \tag{2.28}$$

Arjovsky, Chintala, and Bottou (2017) shows how the EM distance outperform similar metrics, such as KL divergence, and is the only function of the ones tested which is continuous and differentiable almost everywhere in the context of GANs, which is necessary for backpropagation to be applied successfully. Calculating the EM Distance with a neural network is not trivial but Arjovsky, Chintala, and Bottou (2017) show how it can be approximated as equation 2.29; with C_w being the function approximated by the Critic (with weights $\mathbf{w}$) and K a constant. With the constraint that the critic function has to be of class 1-Lipschitz which means that slope of the function can not be larger than 1 i.e. $\nabla_{\mathbf{x}} C_{\mathbf{w}}(\mathbf{x}) \leq 1 \forall \mathbf{x}$. Arjovsky, Chintala, and Bottou (2017) enforces this constraint by clipping the weights of the Critic to a fixed value ($[-c, c]$) after each update.

$$\max_{w} E_{\mathbf{x}\sim p_r}[C_{\mathbf{w}}(\mathbf{x})] - E_{\mathbf{x}\sim p_g}[C_{\mathbf{w}}(\mathbf{x})] \leq K \cdot W(p_r, p_g) \tag{2.29}$$

The Critic network can then be trained until convergence and still keep a linear (non-zero) gradient for the Generator to learn from, hence solving the stability problem of GANs.

Training W-GANs

The W-GAN is trained in two phases just like ordinary GANs. Firstly, the parameters of the Generator is frozen and the Critic is trained until convergence by minimizing the loss function (equation 2.30). Secondly, after clipping the weights of the Critic, the Critic network is frozen and the Generator is trained to fool the Critic by minimizing equation 2.31. This repeats until the Generator converges. (Arjovsky, Chintala, and Bottou, 2017).

$$L = E_{\hat{\mathbf{x}}\sim p_g}[D(\hat{\mathbf{x}})] - E_{\mathbf{x}\sim p_r}[D(\mathbf{x})] \tag{2.30}$$

$$L = -E_{\mathbf{z}\sim p_z}[C_{\mathbf{w}}(G_{\boldsymbol{\theta}}(\mathbf{z}))] \tag{2.31}$$

Improved W-GAN

Gulrajani et al. (2017) improves the W-GAN further by applying a regularizing term to the Critics loss function, forcing the norm of the Critics gradient to be ≤ 1 without using weight clipping, resulting in even better results. With this improvement in place the Critics objective function is defined as equation 2.32.

$$L = E_{\hat{\mathbf{x}}\sim p_g}[D(\hat{\mathbf{x}})] - E_{\mathbf{x}\sim p_r} + \alpha E_{\hat{\mathbf{x}}\sim p_g}[(||\nabla_{\hat{\mathbf{x}}} D(\hat{\mathbf{x}})||_2 - 1)^2] \tag{2.32}$$

2.6.11 Auto Encoders (AE)

The concept of an Auto Encoder (also known as an Auto Associator) was first introduced by Rumelhart, McClelland, and University of California (1986) to address a problem of unsupervised learning. An Auto Encoder is a neural network in the shape of an hour-glass, see figure 2.25, where the middle most hidden layer is smaller than the input/output layers. By training the network to replicate its input, and make the output as indistinguishable from the input as possible, the input is effectively encoded in the middle layer in a much denser format. The benefit with this approach is that the network can learn patterns and features in data on its own without the need for training data to be labeled. After training the Encoder and Decoder networks can be separated and used separately to encode respectively decode data of the class it was trained on like a specialized compression algorithm. The network is trained with backpropagation by minimizing the reconstruction error. An example of a typical loss function is expressed in equation 2.33, using the mean squared error of the reconstruction.

$$L(\mathbf{x}, \mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2 \tag{2.33}$$

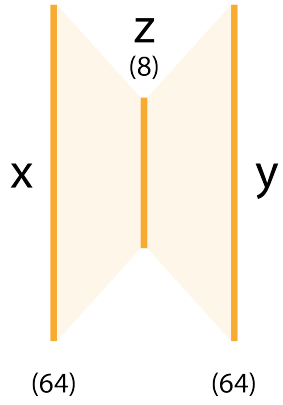

FIGURE 2.25: Example network topology of an Auto Encoder

2.6.12 Variational Auto Encoders (VAE)

A Variational Auto Encoder as defined by Kingma and Welling (2013) and Rezende, Mohamed, and Wierstra (2014) is topologically similar to the Auto Encoder but instead of learning a function representing the data, it learns parameters representing the distribution of the data (mean and variance). It also forces the latent space $p_{\mathbf{z}}(\mathbf{z})$ to take the shape of a known distribution $q(z)$ (commonly Normal $N(0, 1)$), by adding KL divergence (2.4.6) as a regularizing term in the loss function. This allows

encoded data points of similar data (which in a regular Auto Encoder could map to random points in latent space) to map to points close to each other, forming clusters of similar data. This enables relations between high dimensional data to be visualized in lower dimensional space, as well as new data to be generated by sampling new points from the latent distribution and feeding it through the Generator part of the network. Figure 2.26 shows an example of a Variational Auto Encoder which learns vectors $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ representing the mean and standard deviation of the latent space distribution $p_{\mathbf{z}}(\mathbf{z})$. $\mathbf{z}$ in turn is a sampling layer which samples $\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})$ using a random variable $\epsilon \sim N(0, 1)$ in equation 2.36.

The network is trained like an ordinary Auto Encoder but with the addition of the KL divergence term to the loss function, see equation 2.34. This can be approximated by equation 2.35 if the target distribution $q(\mathbf{z})$ is a normal distribution $N(0, 1)$ (Kingma and Welling, 2013).

$$L(\mathbf{x}, \mathbf{y}, \mathbf{z}) = \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2 + D_{KL}(p_{\mathbf{z}}(\mathbf{z})||q(\mathbf{z})) \tag{2.34}$$

$$L(\mathbf{x}, \mathbf{y}, \mathbf{z}, \boldsymbol{\sigma}, \boldsymbol{\mu}) \simeq \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2 + \frac{1}{2}\sum_{i=1}^{n}(1 + \log(\sigma_i^2)) - \mu_i^2 - \sigma_i^2) \tag{2.35}$$

$$z_i = \mu_i + \epsilon\sigma_i \tag{2.36}$$

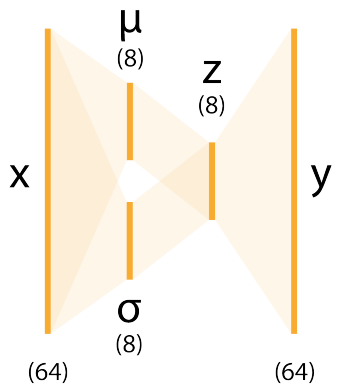

FIGURE 2.26: Example network topology of a Variational Auto Encoder

2.6.13 Adversarial Auto Encoders (AAE)

Adversarial Auto Encoders (AAE), introduced by Makhzani et al. (2015), can can be seen as an Auto Encoder (section 2.6.11) incorporating the GAN (section 2.6.9) method of using an adversarial discriminative counterpart. The Discriminator is connected to the latent space vector, $\mathbf{z}$, see figure 2.27 (left), and fills a similar function as the (KL divergence) regularization term for the VAE, (section 2.6.12), by forcing $p_{\mathbf{z}}(\mathbf{z})$ to take the shape of a desired distribution. Makhzani et al. (2015) showed that the AAE managed to better fill the desired distribution in the latent space, where the VAE left gaps and unfilled areas, hence the AAE can sample more realistic images from those areas.

Training AAEs

The AAE is trained in two phases, *Reconstruction* and *Regularization*. In the reconstruction phase the Encoder and Decoder are trained, to encode and reconstruct the input image and their weights are updated. In the regularization phase the Discriminator is first trained and updated, to differ between the wanted and the generated distribution. Then then the Encoder is trained and updated to better fit the distribution. Makhzani et al. (2015) also showed that it is possible to incorporate label classification in the distribution, through semi-supervised training. This is done by adding a one-hot vector, containing encoded labels, to the Decoders input and use a second Discriminator to ensure that the one-hot vector only contains label and no style information, see figure 2.27 (right).

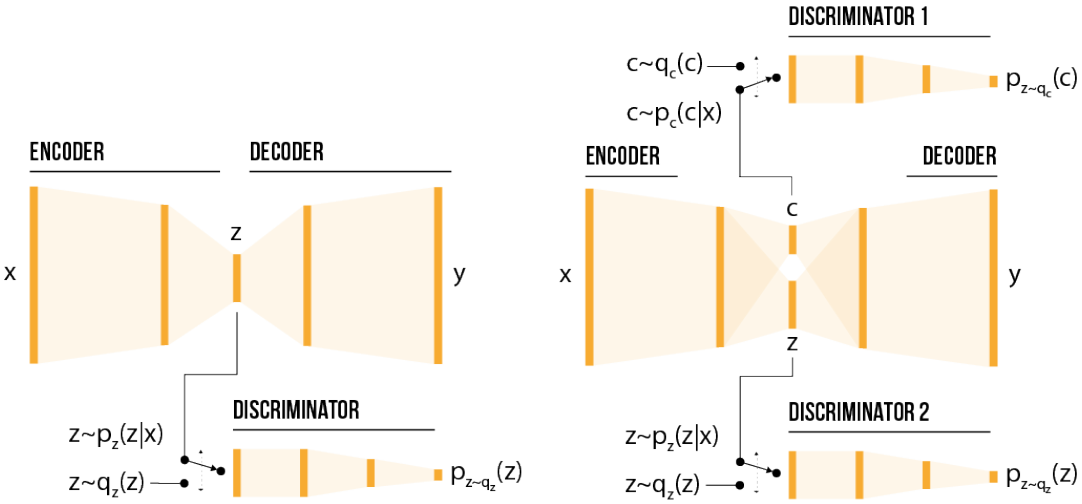

FIGURE 2.27: Example network topology of a basic Adversarial Auto Encoder (left), and a modified version incorporating label classification (right)

2.6.14 Denoising Auto Encoders (DAE)

By applying random noise to the input data of an Auto Encoder and train it to output data as close as possible to the clean input data without noise, the network learns to generalize the important features of the data and ignore overlaying noise. When fully trained, this type of network can be used to remove noise from similar data fed into the network. Vincent et al. (2008) shows how this technique can be applied to images in order to successfully reduce picture artifacts and noise. If the networks main purpose is to remove noise from the input, the hidden layer is usually kept in the same size as the input and output layers or larger to avoid compression losses. With the encoding half of the network denoted $E(\mathbf{x})$ and the decoding half $D(\mathbf{z})$ this operation can be written as equation 2.37, where ϵ is some sampled noise.

$$\mathbf{x} \simeq \mathbf{y} = D(E(\mathbf{x} + \epsilon)) \tag{2.37}$$

2.6.15 Sparse Auto Encoders (SAE)

Sparse Auto Encoders are similar to regular Auto Encoders but instead of encoding data in lower dimensional space it encodes data in higher dimensional space; with the hidden layer containing more nodes than the input. This allows the network

to pick up on smaller details and patterns in the data and learn a greater collection of these compared to an ordinary AE. To force the network to learn these patterns and to prevent the network from choosing the trivial solution, where the input is fed directly to the output, a sparsifying component is added; to limit the number of hidden nodes that are activated at once. For example by adding a sparsity regularization term to the cost function as shown by Cho and KyungHyun (2013), or by modifying the activation function from the hidden layer to only pass on the k strongest activations as shown by Makhzani and Frey (2013).

2.7 Recent Implementations Related to Product Design

There has been a lot of work in the field of DL which could be implemented in different stages of the product design process. Presented below are some promising areas and findings.

2.7.1 Reinforcement Learning

Mnih et al. (2013) presented a CNN model using reinforcement learning to learn how to play several different ATARI games, without the need to modify network architecture or hyperparameters between the games and still reach current state of the art result. This achievement may be one of the things that has risen an interest in the potential of reinforcement learning to enable a self improving and self learning neural network, making decisions depending on its environment, similar to how humans function. Silver et al. (2017) trained a network (AlphaGo Zero) to master the ancient game Go, learning novel tactics and defeating the human world champion, without providing the network with any knowledge of the game except the rules. Reinforcement learning can be used to teach industrial robots how to perform a task without explicit programming, enabling the robot to choose the correct action to perform depending on the circumstances (Zhu et al., 2018). This can allow a person without programming knowledge to use a robot for a custom task. Schmitt et al. (2018) recently showed that it is possible to let neural network learn from other previously trained networks, making the learning process quicker and achieving better performance.

2.7.2 Image Generation

By using a GAN or VAE based model it is possible to change the value of the latent space vector and thereby create a great variation of similar but different images, as shown by White (2016), Shang, Sohn, and Tian (2017) and Bojanowski et al. (2017) among others. Ha and Eck (2017) recently published a model that through vector based actions generates drawings instead of pixel based actions, making the process more similar to how humans paint and draw images.

2.7.3 3D Model Generation

There are many different approaches on how to use neural networks to generate a 3D model. Some produces voxel based model through a GAN (Sharma, Grau, and Fritz, 2016) or VAE (Brock et al., 2016) while others produce point cloud models (Lin, Kong, and Lucey, 2017) (Achlioptas et al., 2017). One of the current state of the art networks, uses a 2D image as input and produces a mesh model by deforming a preexisting mesh step by step using a CNN (Wang et al., 2018).

2.7.4 Style Transfer

Gatys, Ecker, and Bethge (2016) showed that it is possible to separate style and content with in neural networks by used a CNN and then transfer the style from one image to another while preserving the content of the target image. Since then there has been several studies on how to improve the transfer and recently Zhang, Zhang, and Cai (2017) presented a network using two encoders that separately extract style respectively content, which generalizes better. Style transfer can also be used to transfer the style from an image to a 3D model. Kato, Ushiku, and Harada (2017) presented a method that can reconstruct a 3D mesh into the style of an image by editing the textures and vertices of the mesh, see figure 2.28 for an example of the methods result.

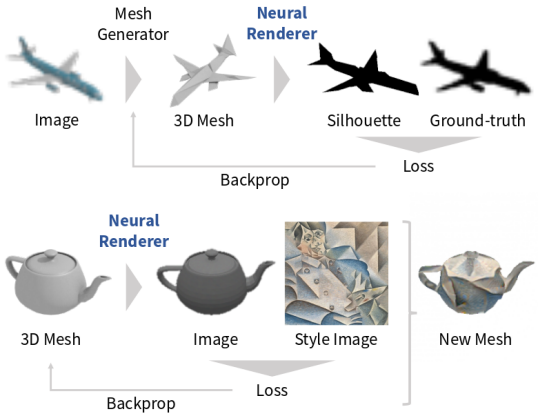

FIGURE 2.28: Image style transfer to mesh model (results from Kato, Ushiku, and Harada (2017))

Chapter 3

Method

The following section describes the methodology used in each phase of the project and how it relates to the project goal to formulate a conceptual framework for generative product design based on machine learning. For a description of the overall approach for this thesis work see section 1.5.

3.1 Theory Studies

The goal with the theory study is to form an understanding of the field at hand and what it can be used for today and in a near future. This includes identifying common methodologies and techniques, their advantages, disadvantages and applications as well as known problems, challenges and strategies to address them.

This information is collected by searching for and studying research papers, publications and literature on the subject and by following recommendations and citations from public figures in the field. A flow chart of this approach can be seen in figure 3.1.

Publications is primarily searched for through Google's search engine Google Scholar [1] and through the white paper database ArXiv [2]. Relevant topics are identified from past searches from these sources as well as new trending topics within the field of deep learning published on e.g. Reddit [3]. Cutting edge research is also located by following news and publications from organizations such as OpenAI [4] and DeepMind [5].

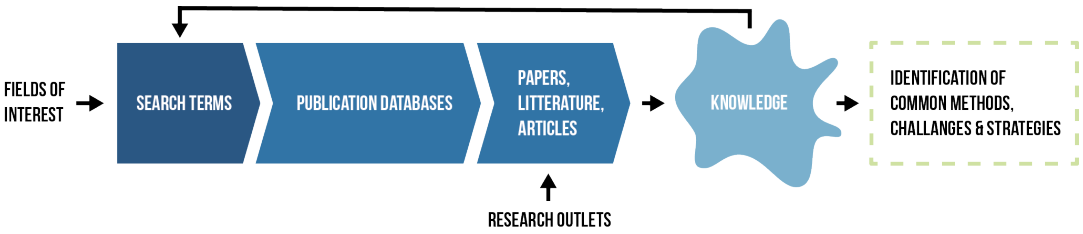

FIGURE 3.1: Flow chart of the theory study process

[1] https://scholar.google.se/
[2] https://arxiv.org/
[3] https://www.reddit.com/r/deeplearning/
[4] https://openai.com/
[5] https://deepmind.com/

3.2 Concept Development

The method for identifying and generating the concepts consists of discussions and brainstorming sessions on how the research presented in chapter 2 can be modified and implemented to fill other functions. When a possible implementation is identified, it is discussed in more detail in order to clarify what architectures are best fitted and how great the impact of a successful implementation would be on the product design process, described in 2.1.

3.2.1 Morphological Analysis

To generate and identify possible top level architectures for AI-powered generative design a Morphological Matrix is used. Each row in the matrix corresponds to a sub-function of the product design process related to generative design, and each column corresponds to a certain machine learning method, algorithm or concept. When populated with possible solutions the matrix can be used to identify holistic solutions as shown in figure 3.2.

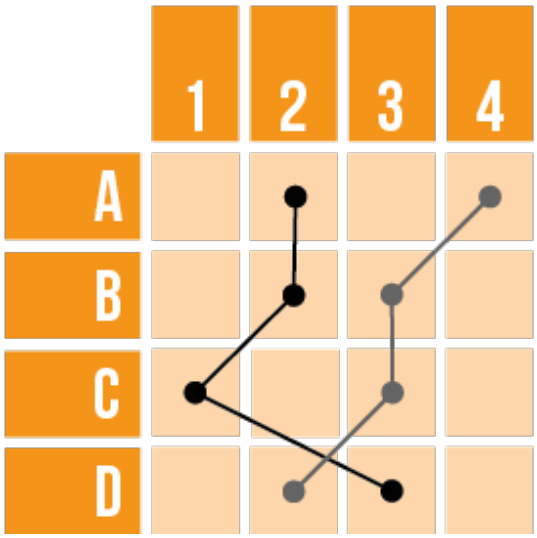

FIGURE 3.2: A Morphological Matrix constructs a grid of part solutions organized after problem or function on one axis and principle or concept on the other. Any complete subset of solutions may form a conceptual holistic solution.

3.3 Implementation

3.3.1 Prototype

The prototype is developed with the programming language Python using TensorFlow and Keras for neural network construction, training and computation.

TensorFlow

TensorFlow (Abadi et al., 2016) is an open source software library for numerical computation developed for machine learning and neural network research. It is primarily developed for Python but is also available for other languages such as Java, Go and C.

Keras

Keras (Chollet and Others, 2015) is a deep learning library capable of running on top of libraries such as TensorFlow to offering a high-level neural networks API for fast and easy prototyping. It is written in Python.

Exploring the Latent Space

In order to probe how well the training is going and what the network is learning over time a callback is added to the training loop. The callback plots and saves a grid of images generated by the network, sampled from different parts of the latent space. The resulting image provides a rough perception of the current generated design distribution $p_g(\boldsymbol{x}|\boldsymbol{z})$ and how it changes over time as the network improves. These images are most intuitive when the sampled z has two dimensions, as both axis can be represented in the plot. For higher dimensional z (e.g. 100) only a change in two of the variables can be meaningfully represented in the plot, but the quality of the samples in the grid can still indicate the networks performance just not how well it is distributed. Similar images are also generated from completely random samples of z, to probe the sampling capability and how well the network can generate new data.

3.3.2 Auto Gathering of Image Training Data

Gathering and preprocessing training data is a tiresome task: relevant data needs to be collected, correctly labeled and cropped to appropriate size. To automate the process of gathering training data we propose the following system, built around a web scraper and an object detection model (see figure 3.3).

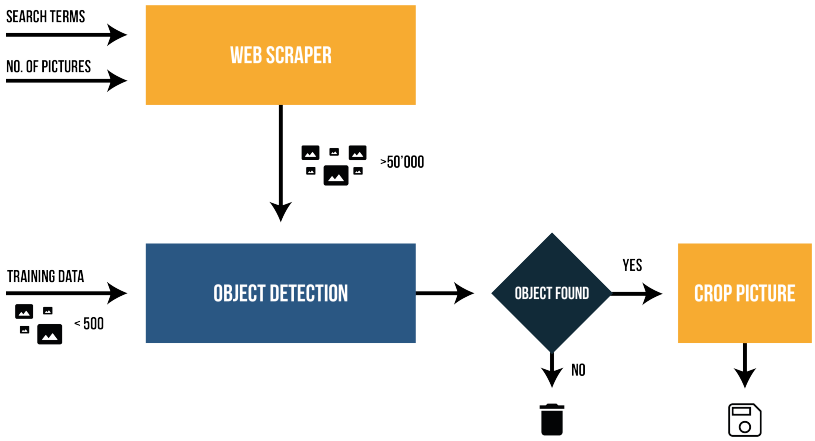

FIGURE 3.3: Flow chart of an automatic data gathering system

The web scraper compiles large sets of images from selected sources based on appropriate keywords. Multiple search terms on multiple languages can be used to assure a diversity in the results. The goal is to collect as much relevant existing data as possible from relevant public sources.

After collection the data is passed on to the object detection model which tries to determine weather the image is a false positive or not. The model is also capable of marking in which region(s) the object(s) of interest is/are positioned, making it possible to automatically crop these images to the region of interest.

An object detection model like this also needs training data before it can operate but a subset of 100-500 images are usually enough to achieve acceptable results in

identifying a certain class. Far less than the 50'000+ often required for generative and more precise models. Positive results from the object detection model can also be used to further train the model over time to improve its accuracy.

Chapter 4

Concept Development

4.1 Concepts

Based on the theory study in chapter 2, the following concepts were identified as possible to implement in the product design process, described in section 2.1. Each concept describes a tool, method or process which based on different machine learning techniques could potentially accelerate and/or automate parts of the PDP.

4.1.1 Identification of Product Requirements and Constraints

The user provides a set of images or a video showcasing a product function or use case. The system identifies the types of objects in the scene, their relation and constructs an internal model based on weight, force, size assumptions from known data. The system then generates design alternatives based on this understanding.
Applicable neural network architectures: CNN, RNN, Reinforcement Learning

4.1.2 Contextual Design Suggestion

A system where the user receive generated product design suggestions based on provided context. The context could be one or more images uploaded by the user. See figure 4.1.
Applicable neural network architectures: GAN, VAE, CNN

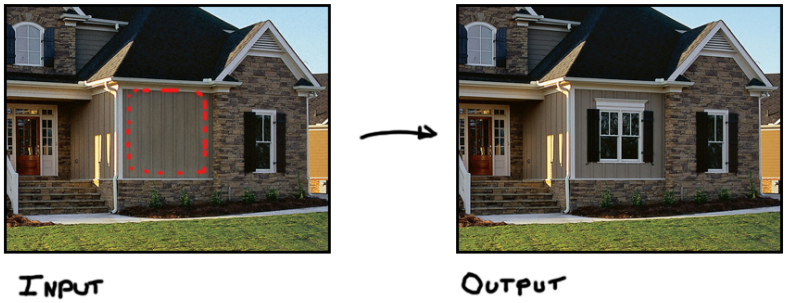

FIGURE 4.1: Illustration of concept 4.1.2 Contextual Design Suggestion

4.1.3 Abstract Design Combination

A system where a product design is generated based on a mixture of traits. Especially interesting are traits which are distinct but difficult to define analytically, such as *French*. These could also be physical properties such as *see through* and *openable*,

or different products entirely such as *window* and *chair*. The image generation works by interpolating between learned features in the encoded latent space of a learned distribution of products, see figure 4.2.
Applicable neural network architectures: GAN, VAE, AAE

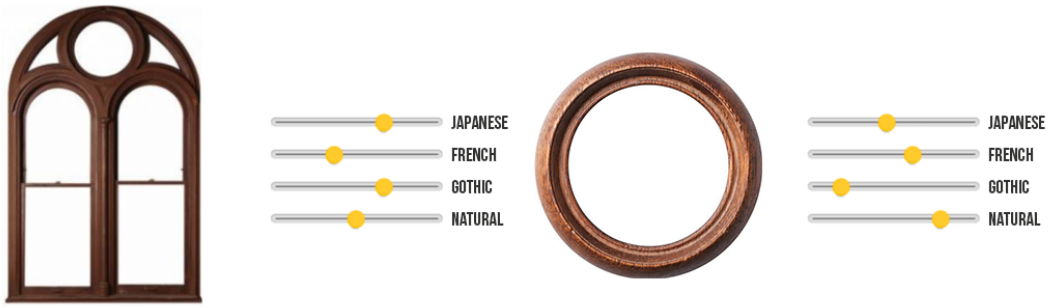

FIGURE 4.2: Illustration of concept 4.1.3 Abstract Design Combination

4.1.4 Design Suggestions from Inspiration

As described in section 2.7.4 it is possible to separate style from content using a CNN. By combining this with a generative network such as a VAE or a GAN it is possible to show the network some inspirational images to extract design features from and then generate a model or an image of a product with that design.
Applicable neural network architectures: GAN, VAE, CNN

4.1.5 Design Variation of Concept Sketches and 3D Models

The user provides one or several sketches or 3D models as input for the network. The network then generates multiple, similar, variations of the input by drawing samples from the latent space surrounding the provided input. It is also possible for the user to interpolate between different values to get combinations of different concepts and designs.
Applicable neural network architectures: GAN, VAE

4.1.6 2D to 3D

The user provides a 2D design, such as a sketch, image or blueprint and the network generates the corresponding 3D model.
Applicable neural network architectures: FFNN, RNN, CNN

4.1.7 3D Model to Components

The user specifies the available materials, tools, constraints and optimization objective. The system provides a manufacture assembly solution fulfilling the requirements.
Applicable neural network architectures: a combination of RNN, FFNN, CNN, Reinforcement Learning

4.1.8 Auto complete CAD

By using reinforcement learning to train a neural network, similar to the works described in section 2.7.1 but with different environment and rewards, it can learn to optimize design parameters of a product. It can then help to quickly come to a feasible design to fit certain specification, without time consuming analyzes. When the designer is building a model in CAD the trained network could come with suggestions on how to proceed in order to, for example, make the model more durable, easier to manufacture or cheaper.
Applicable neural network architectures: FFNN, CNN, Reinforcement Learning

4.1.9 Choice of Materials

The user provides a list of available materials and a list of components or a 3D model and the network optimizes the choice of materials for each component based on, for example, pricing, durability, how the materials affect each other, appearance or environmental impact.
Applicable neural network architectures: FFNN, CNN, Reinforcement Learning

4.1.10 Choice of Processing Method and Tools Selection

A system that monitor and analyzes the operations of all available machines and tools and uses that information to optimize what tool and processing method to use to manufacture the provided product.
Applicable neural network architectures: FFNN, CNN, Reinforcement Learning

4.1.11 Topology and Material Optimization

A system that optimizes the material consumption by modifying the topology of an existing model while maintaining its manufacturability.
Applicable neural network architectures: FFNN, CNN, Reinforcement Learning

4.1.12 Evaluate a Product in a Virtual Environment

By using reinforcement learning to train a neural network in a physics engine environment and allowing it to interact with different objects it can be thought to quickly make good estimations about the objects durability, maneuverability, manufacturability or user interaction aspects. The user then provides a 3D model of a product and the network comes with suggestions on how to improve it.
Applicable neural network architectures: Reinforcement Learning

4.2 Morphological Matrix

Table 4.1 shows the connection between the concepts and the part of the product design process where they can be implemented.

Morphological Matrix					
Idea Generation	4.1.2	4.1.3	4.1.4	4.1.5	4.1.6
Product Requirements	4.1.1				
Concept Design	4.1.2	4.1.3	4.1.4	4.1.5	4.1.6
Concept Development	4.1.5	4.1.6	4.1.7	4.1.8	
Detail Design	4.1.7	4.1.8	4.1.9	4.1.10	4.1.11
Testing & Refinement	4.1.8	4.1.12			

TABLE 4.1: The concepts connection to the product design process summarized in a morphological matrix

4.3 Concept Choice

From the concepts listed in section 4.1 one is selected for the creation of a prototype. The concept we have selected is concept Abstract Design Combination (section 4.1.3). The reasons for selecting this concept over the others are the following:

4.3.1 Can be Reduced to 2D

The selected concept could potentially be applied to both 2D and 3D and in both scenarios the underlying techniques is assumed to be very similar. Thus by implementing a system for the 2D case and draw conclusions from that, it is not far fetched that the same conclusions conclusions could be extrapolated or transferred into the 3D domain.

4.3.2 Easy to Visualize

Design variations in a visual context are easy to visualize in picture format, and allow for results to be easily interpreted and judged based on quality and accuracy in comparison to existing images.

4.3.3 May Utilize Many State of the Art Techniques

The selected prototype could be implemented using a mixture of many state of the art techniques and generative network architectures, such as different flavours of Auto Encoders and Generative Adversarial Networks. Many of the research papers we have encountered regarding generative neural networks in our studies, see chapter 2, points towards different variations of GAN and VAE as the most promising areas. By working with these state of the art models we hope to form a fair big picture of the challenges an implementation like this or similar entails.

4.3.4 Reasonable Complexity

Based on recent research in the area of generative models (GAN and Auto Encoders), it is clear that this type of system is feasible with the current state of the technology, and therefore reasonable for us to pick up within the scope of this project.

4.4 Target Product

The target product to tailor the prototype for is upon request from SkyMaker AB (section 1.1) chosen to be *Windows*. Windows come in a variety of shapes and form

and are well recognized by the public and well understood. While windows do have complex 3D geometry up close and could be designed with much more complex mullion design than they are currently, the overall shape and style of a window at a distance is well approximated in 2D. The 2D design could then be used as a high fidelity sketch of the design as a reference for creating a 3D model, manually or generated by AI.

Chapter 5

Concept Implementation

The following chapter covers the implementation phase which was laid out in figure 1.1 with the ultimate goal of building a prototype of the concept Abstract Design Combination, selected in section 4.3. This phase is divided into two parts: The first part; Auto Gathering of Image Training Data (section 5.1), covers the creation of a dataset and a system to automate the data gathering and the preparation process. The second part; Prototype (section 5.2), covers the development of the prototype using the dataset created in section 5.1 to generate new design variations.

5.1 Auto Gathering of Image Training Data

To automate the process of gathering and labeling training data a system is developed according to the process proposed in 3.3.2 (see figure 3.3); using a web scraper in tandem with an object detection model to automatically crop and label images according to selected search terms. The system is built with Python as a command line application for Windows. In the section below we first discuss the underlying problem and goal with the system, followed by a description of the overall system architecture as well as the implementation and results of each sub-system.

5.1.1 The Problem

Generative models require immense amounts of data in order to perform sufficiently. One of the challenging and most prominent problem in machine learning related to both classifiers and generative models is the problem of overfitting (Caruana, Lawrence, and Giles, 2001). Overfitting occurs when the model misinterpret random occurring patterns as a correlation to the data or when the data to network size allows the network to fully remember the dataset. In both cases the model fails to generalize beyond the training data which is (in most cases) essential for the model to be useful. We can see that in the case of latent space interpolation between existing designs, generalization is less of a problem, but that the lack of a diverse dataset instead affect the models ability of generating "intermediates" and "filling in the blanks" which a model trained on a larger dataset is much more capable of. The consensus is that models trained on larger datasets perform more accurately and are more robust than models trained on smaller datasets (Ajiboye et al., 2015).

More data does not necessarily translate into better models, the data needs to be of good quality and coherent with the task at hand. It may still be possible for a model to learn from noisy data, many argue that it is necessary for the model to be robust in the long run. But such a model will require much more time and computing to converge compared to the simple case with carefully washed data. And at this point processing is the primary bottleneck.

5.1.2 Goal

The goal with the system is to be able to gather a large and diverse set of image data from the web on one or more specified topics in a time efficient way that outperforms doing the same task manually. When built, the system will be a tool to gather crude data for generative models based on existing image data, a tool which saves the user both time and effort.

One could argue that it takes longer to develop this system than doing the task in the first place, which is true and a valid argument if the data gathering was only done once. Throughout this project we end up using this system a couple of times which might still not justify the creation of it. But we are certain that this tool will come to good use for us and for others in the future when building similar systems. This system with its code is all open source and will be accessible for everyone.

5.1.3 System Architecture

A schematic of the system architecture can be seen in figure 5.1. The system follows the primary structure of the method proposed in section 3.3.2. In addition it has been extended with a translation module, to offer greater diversity in the search results, as well as several intermediate processing steps, to further automate the work of training the object detection model.

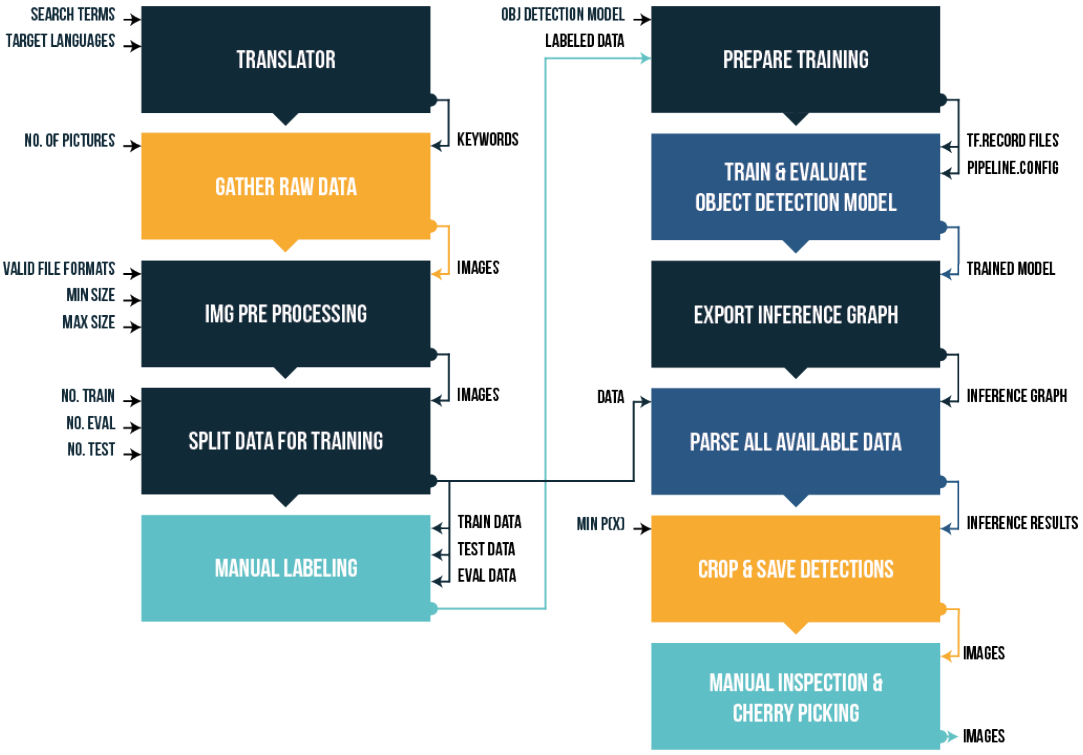

FIGURE 5.1: Flow chart of the finished Data Gathering System

The system is developed with modularity and flexibility in mind, allowing modules to be changed or inserted in the dataflow with ease and different parts of the system to be run in any desired sequence, as long as expected inputs and outputs are fulfilled. Each tile in figure 5.1 represents a module.

Web Scraper

The web scraper part of the system is built around the Google's Custom Search API [1] allowing multiple search queries to be executed programmatically for text or images. The API provides several ways of narrowing the search space and is free to use for up to 100 queries per day. Search results are returned in JSON-format including URLs to the results and are easily downloaded with Python. The speed of the Scraper is primarily limited by the speed of the internet connection, as the Custom Search API responds under a second and no further processing is performed. This module can be used on its own and has been released as a stand alone Python module [2]. Expected input is a list of strings to search for, the max number of images to scrape per string and a target output directory.

Translation of Search Terms

To provide a greater diversity in the search result and more data a translation module is added to the system, which translates provided search terms to any set of requested languages. This module is just a wrapper around Google's Translation API [3] allowing us to interact with it easily through our Python system. With this module we are able to search for any number of strings translated to any number of languages (provided they are supported by Google Translate) all automatic. This works in principle but there are some caveats to watch out for; especially "false friends", words in one language meaning something completely different in another language as these will return unwanted search results. This module takes a list of search strings (in English by default) and a list of target languages as input and returns an associative array of translations.

Object Detection Model

For Object Detection the system implements the TensorFlow Object Detection API [4] developed by Huang et al., 2016; an open source framework built on top of Google's TensorFlow, simplifying the process of constructing, training and deploying object detection models. The framework provides several pretrained networks of different architectures, which can be used as they are or as starting point to greatly reduce the training time (Gupta, 2017). It is also possible to extend an existing model in any way or only reuse part of it if desired (by dropping, appending and freezing network layers). All models return predicted object bounding-boxes or object masks, indicating where in the image one or multiple instances of an object is located, as well as a class prediction and associated uncertainty.

Automatically Crop & Save Images

Positive detections from the object detection model are automatically cropped and saved using the returned predicted bounding boxes converted back to pixel space. To eliminate false positives only bounding boxes with an inference probability above a threshold of 60% are considered. If the image contains multiple instances of the desired object or objects this process results in multiple images, one for each instance.

[1] https://developers.google.com/custom-search/json-api/v1/overview
[2] https://pypi.org/project/gscraper
[3] https://cloud.google.com/translate
[4] https://github.com/tensorflow/models/tree/master/research/object_detection

Auxiliary Functions

The system also contains auxiliary functions to automate intermediate tasks such as data splitting, preprocessing and file conversion. All images from the web scraper are passed through a preprocessing step, removing corrupt files, files considered to be too small and files of undesired file format. Images considered too large are shrunk and all duplicates are removed. Before manual labeling a subset of the scraped data is automatically split into train, test and validation batches. After labeling, the labelmap and annotation files (in xml format) are automatically compiled together with the image data to the correct format for the object detection model to train on (.record files). Object detection configuration files (pipeline.config) are also generated automatically in the process. All auxiliary functions have high level parameters, which can be set by the user to control its execution. Together it creates an easy framework to gather new data and train and test different object detection models.

5.1.4 Model Selection

When selecting a pretrained model to fine tune for a specific object detection case there are multiple things one might want to consider; such as which dataset the model was trained on, expected input/output, accuracy, computational complexity, architecture, size, etc. Depending on application a trade-off between speed and accuracy might look different and especially in mobile applications model size and complexity plays a more prominent role. The dataset matters because it defines the feature-space which the model is able to learn. A network trained on clothes is more likely to yield better results when tuned for dresses than a model trained on furniture. For this reason it may be beneficial to start experimenting with a model containing the correct domain if possible. Otherwise it is recommended to select one trained on a more generic dataset such as COCO [5] (Common Objects in Context), the Open Images Dataset [6] or any other similar subset. The accuracy of fine tuned pretrained models has also been shown to increase logarithmically with the size of the pretrained dataset (Sun et al., 2017), so a model trained on a larger dataset is to be preferred. Besides this the selection is basically a trade-off between speed and accuracy. A comparison of available pretrained object detection models from the TensorFlow Object Detection Model Zoo [7] can be seen in figure 5.2.

We chose to work with three of the pretrained models available in the Tensorflow Object Detection Model Zoo: *ssd_mobilenet_v2_coco* [8], *faster_rcnn_inception_v2_coco* [9] and *faster_rcnn_nas_lowproposals_coco* [10], see table 5.1. The first model was selected for being the fastest, the third model was selected for having highest accuracy, and the second model was selected arbitrary from the Pareto Optimal points located in between the other two (see figure 5.2).

- Ssd_mobilenet_v2 (Sandler et al., 2018) is a state of the art model specifically designed for mobile and resource constrained environments.

[5] *http://cocodataset.org/#home*
[6] *https://storage.googleapis.com/openimages/web/download.html*
[7] *https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md*
[8] *http://download.tensorflow.org/models/object_detection/ssdlite_mobilenet_v2_coco_2018_05_09.tar.gz*
[9] *http://download.tensorflow.org/models/object_detection/faster_rcnn_inception_v2_coco_2018_01_28.tar.gz*
[10] *http://download.tensorflow.org/models/object_detection/faster_rcnn_nas_lowproposals_coco_2018_01_28.tar.gz*

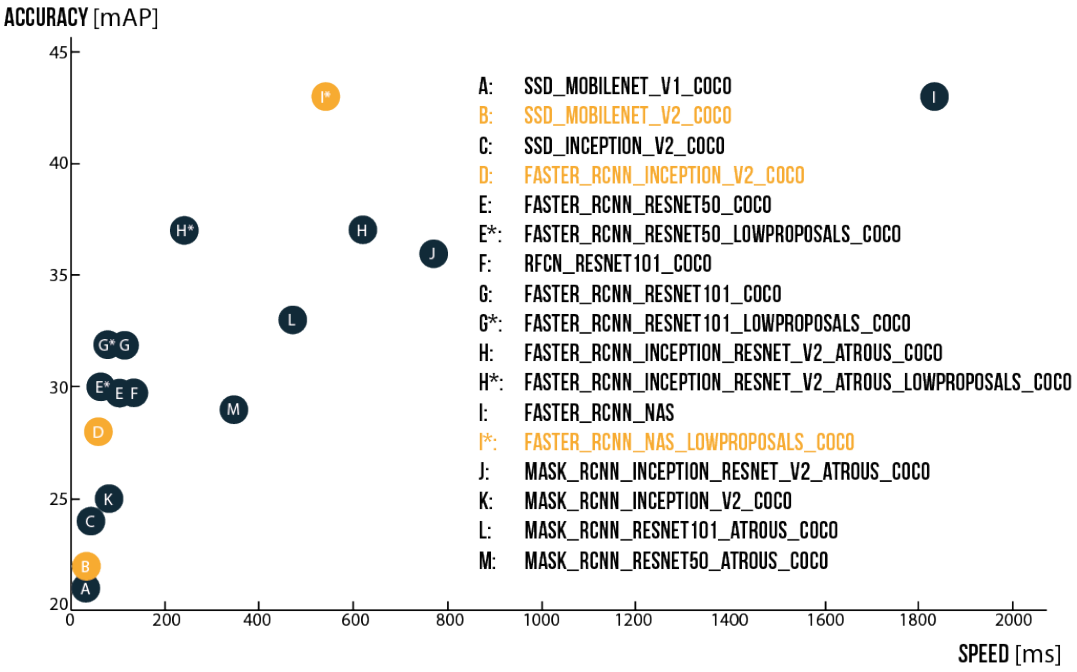

FIGURE 5.2: Comparison of available pretrained object detection models in the TensorFlow Object Detection Model Zoo (as of 2018-05)

Model Name	Speed (ms)	mAP	Outputs	Released
ssd_mobilenet_v2_coco	31	22	Boxes	18-04-02 (Sandler et al., 2018)
faster_rcnn_inception_v2_coco	58	28	Boxes	15-12-02 (Szegedy et al., 2015)
faster_rcnn_nas_lowproposals_coco	540	43	Boxes	17-07-21 (Zoph et al., 2017)

TABLE 5.1: Selected object detection models

- Faster_rcnn_inception_v2 is an architecture combining techniques from both faster_rcnn (Ren et al., 2017) and the inception_v2 (Szegedy et al., 2016) architectures which both have proved to be very successful. The resulting model has a good trade-off between speed and accuracy.
- Faster_rcnn_nas_lowproposals is an accelerated version of the Faster_rcnn_nas model (Zoph et al., 2017) returning a maximum of 50 (instead of 200) detections per image (which is more than enough of what we expect to require from a typical picture of a window or building).

5.1.5 Model Training (tuning)

All models are trained (using the pretrained model as a starting point) on a dataset of approximately 1000 windows of different shape and form, manually labeled from a collection of 500 images collected by the Web Scraper (section 5.1.3) using the keywords and the language list described in appendix A (Scrape Parameters). The labeling was performed with the open source tool *LabelImg* [11] (Tzutalin, 2015) which allows regions of an image to be marked and labeled in an easy manner. Both models was trained one after another on a single GeForce 1070 GPU with 8GB VRAM until convergence.

[11] https://github.com/tzutalin/labelImg

Training Results

The training results for the three models are summarized in table 5.2 and the total loss over the training time can be seen plotted in figures 5.3, 5.4 and 5.5 for the three models respectively.

Model	Steps	GPU time	Convergence	Time / Step	Final Loss
ssd_mobilenet_v2_coco	37'600	15 h	10 h	1.43 s	0.95
faster_rcnn_inception_v2_coco	17'000	30 min	30 min	0.1 s	0.1
faster_rcnn_nas_lowproposals_coco	8'340	9 h	6 h	3.88s	0.9

TABLE 5.2: Training results of selected object detection models

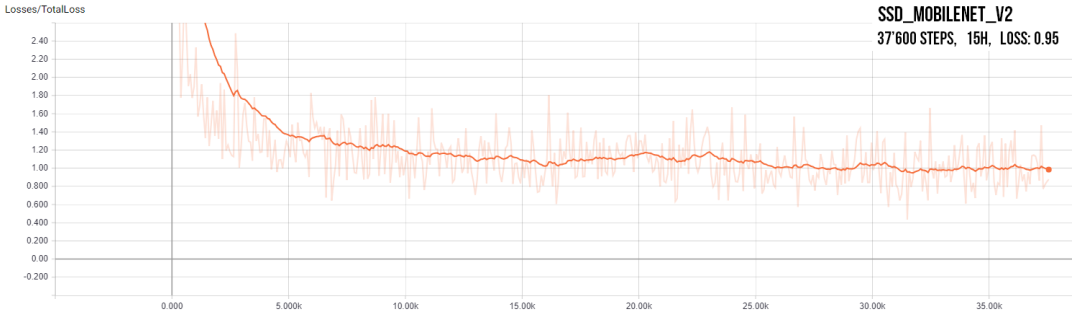

FIGURE 5.3: Plot of the training losses from ssd_mobilenet_v2

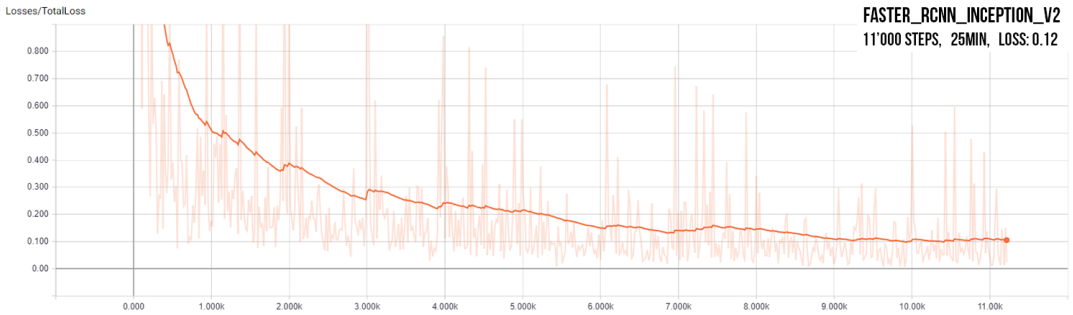

FIGURE 5.4: Plot of the training losses from faster_rcnn_inception_v2

FIGURE 5.5: Plot of the training losses from faster_rcnn_nas_lowproposals

Faster_rcnn_inception_v2 was surprisingly fast to train compared to ssd_mobilenet_v2. Given that ssd_mobilenet_v2 is running 50% faster in the forward pass (31ms vs 58ms) it is easy to assume that the back-propagation and learning time should correlate similarly but that is clearly not the case. The differences in training time are most likely due to differences in the model architectures e.g. number of trainable parameters, types of layers and types of activation functions. When comparing the size (in memory) of the three models (see table 5.3) it is clear that faster_rcnn_inception_v2 in fact is the smallest of the three, which hints it has the least number of trainable network parameters and thus should be the fastest one to train.

Model	size in memory
ssd_mobilenet_v2_coco	68'349 kb
faster_rcnn_inception_v2_coco	56'479 kb
faster_rcnn_nas_lowproposals_coco	416'849 kb

TABLE 5.3: Size in memory of selected object detection models

Comparison of trained models

The trained models are compiled and deployed with the TensorFlow Object Detection framework to be used with new sample data. For each image passed to the network it returns an inference vector of objects it believe to have detected, as well as associated probabilities and relative bounding boxes of those objects.

To probe visually how well the three models are performing compared to one another a test-set of six images are used. This set includes three (positive) images containing windows similar to the training data and three (negative) images including objects which the network was trained on originally: cat, dog, kite and person. The idea is to use these images to tell how well the networks have learned to map previously learned feature detections to detect our new class, while ignore previously learned classes. The results from these tests can be seen in figures 5.6, 5.7 and 5.8, where predicted bounding boxes have been overlaid in green with suggested label and probability score.

FIGURE 5.6: Detection results from ssd_mobilenet_v2

FIGURE 5.7: Detection results from faster_rcnn_inception_v2

FIGURE 5.8: Detection results from faster_rcnn_nas_lowproposals

Comparing these three images we see that ssd_mobilenet_v2 is the most accurate model of the three. It only marks a few of the windows but the ones marked are all correct. Faster_rcnn_nas_lowproposals does in comparison marks the same number of windows but is also incorrectly marks both a dog and a door. Faster_rcnn-_inception_v2 successfully marks almost all windows in the images but also both dogs and the cat.

Training Conclusion

Of the three models tested we choose to use faster_rcnn_inception_v2 for further object detection. Primarily because it managed to mark all windows of interest in the test images and is very fast to train. Mislabeling dogs, cats and other things

as windows are not good but assuming that the scraper primarily collect images of windows and buildings the number of miss-classifications should be minor. With such a fast training time it is also feasible to train the model further using a smaller learning rate and/or a larger set of training data to improve the models performance. It would also be easy to switch to any of the other two models would so be required.

5.1.6 Results

The system for auto gathering of training data resulted in a collection of almost 20'000 images of windows, which after manual selection resulted in a dataset of 4564 images. Details of the processing time and results from each of the sub-systems are presented in table 5.4. For further details on the Scrape Parameters, see appendix A. Some samples of images from the dataset before and after being parsed by the trained object detection and cropping system can be seen in figures 5.9 and 5.10 respectively. Figure 5.11 also show some samples from the final cherry picked dataset.

State	Processing Time	Resulting Images
Multilingual scrape: windows	5 h	28'000
Preprocessing filter	10 min	24'000
Window Detection & Crop	30 min	19'889
Cherry Picking	6 h	4'564

TABLE 5.4: Results of the data gathering process on the subject of windows

FIGURE 5.9: 24 example images randomly selected dataset of raw images collected by the web scraper

FIGURE 5.10: 24 example images randomly selected from the windows dataset after using the object detection model to crop and keep detections

FIGURE 5.11: 24 example images randomly selected from the final windows dataset after cherry picking

5.2 Prototype

Implementing the target concept with deep learning requires three things:

1. A model. This is the neural network architecture making the brain of the application.
2. A dataset containing a diverse set of data points for the model to train on.
3. A shell application. The shell application wraps the model with a user interface enabling user interaction and hosts the model with minimal overheads.

The model is created in Python using Keras with TensorFlow. The dataset is created using the Data Gathering System built in section 5.1. The shell application is left for future studies but could potentially be a web application using Keras.js[12] or Tensorflow.js[13] to deploy the finished Keras model. The goal is to generate new design variations of windows represented by images, preferably in color and at least 64x64px in size.

The following section (5.2.1) covers our interpretation of the deep learning problem of generating new designs based on existing ones and clarifies what we are trying to do in more technical terms. In the section after that (section 5.2.2) the choices of DL method and architecture are discussed, arguing for using a VAE primarily and a GAN secondarily. Sections 5.2.3, 5.2.4, 5.2.5 and 5.2.6 then covers the creation, training and evaluation of three different VAE architectures and one GAN architecture. Results, preliminary discussion and conclusions related to each architecture is presented in each section respectively.

5.2.1 The Challenge of Generating New Designs From Old

Let $\mathbb{D}$ denote the true design space of a product (windows in our case) and $\mathbf{X} = \{\boldsymbol{x}_i\}_{i \in I} \subset \mathbb{D}$ be the set of all known designs $\boldsymbol{x}_i$ (represented by our dataset). The known design space (see figure 5.12) can be interpreted as the convex hull of all known design $S_d = hull(\mathbf{X})$. A new novel design can then be noted as $\widetilde{\boldsymbol{x}} \in \mathbf{X}^c \in \mathbb{D}$ drawn from the distribution $p_{\mathbb{D}}$ of true designs (In our case $\widetilde{\boldsymbol{x}}$ may represent an image). Sadly both $p_{\mathbb{D}}$ and $\mathbb{D}$ are unknown so we can not sample from it directly, but if $|\mathbf{X}|$ is large enough its probability distribution of samples $p_{\mathbf{X}}$ begins to approximate the true distribution $p_{\mathbb{D}}$. While not providing ground breaking new designs $\widetilde{\boldsymbol{x}} \sim p_{\mathbf{X}}(\boldsymbol{x})$ might still be valuable and yield novel designs.

What we are looking for is a function, call it G (*Generator*), which given some random variable $\boldsymbol{z}$ generates a new, preferably unique design $\widetilde{\boldsymbol{x}}$, see equation 5.1.

$$G(\boldsymbol{z}) : \boldsymbol{z} \sim p_z(\boldsymbol{z}) \rightarrow \widetilde{x} \in \mathbb{D} \tag{5.1}$$

Where $\boldsymbol{z} \in \mathbb{R}^k$ and $x \in \mathbb{R}^n$ such that $k < n$.
G is also continuous and differentiable, a criteria for backpropagation to work as it depends on the functions gradient.

From the universal approximation theorem (section 2.4.1) follows that any function (continuous and differentiable), such as G, can be approximated by a neural network, provided it is large enough and we train it on a dataset which is large and diverse enough to approximate S_d and $p_{\mathbf{X}}(\boldsymbol{x})$. The Generator is considered good when the distribution of generated samples $p_G(\boldsymbol{x}|\boldsymbol{z})$ approximates the distribution of the known design space $p_{\mathbf{X}}(\boldsymbol{x})$ well.

[12] https://github.com/transcranial/keras-js
[13] https://js.tensorflow.org/

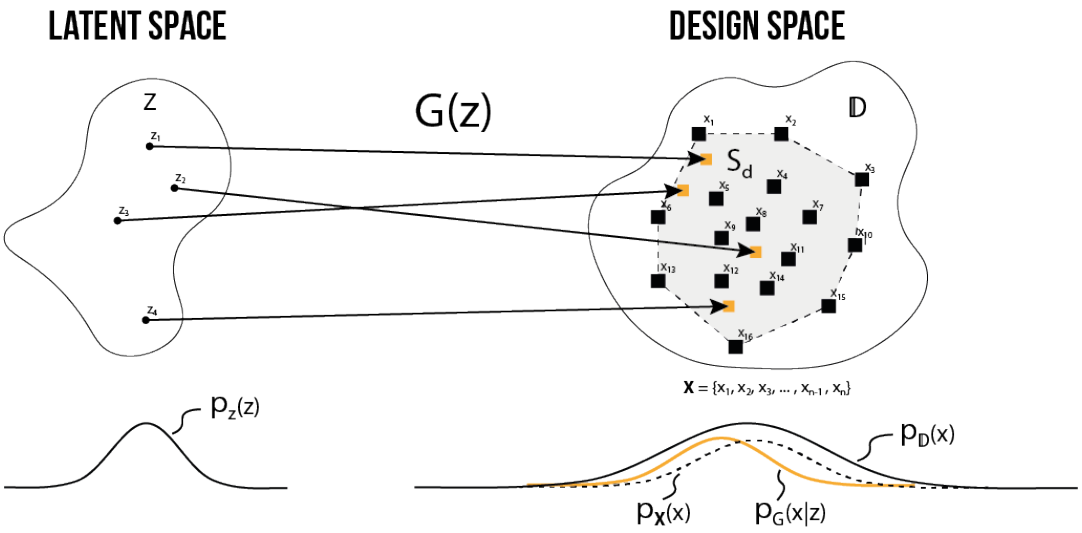

FIGURE 5.12: Illustration of the generator as a function, mapping points from the latent space to points in the design space

5.2.2 Selecting a Deep Learning Method

Training the Generator network can potentially be done through either supervised or unsupervised learning.

Supervised or Unsupervised Learning

One supervised approach would be to first generate a unique z to pair with every sample in the dataset and then train the network to output that specific sample given a specific known z. This could work, but how about generating new samples? There is no mechanism in this method to enforce other values of z to give us anything useful at all. A better method would be to use unsupervised learning where no labels needs to be in place at all and the model can learn the z-encoding on its own. Two state of the art deep learning methods for this type of application are Generative Adversarial Networks (GANs) and Auto Encoders (AEs), Variational Auto Encoders (VAEs) in particular. These architectures are described in sections 2.6.9, 2.6.11 and 2.6.12 respectively but a short summary of each is presented below.

On the subject of GANs

In a GAN new z are continuously drawn at random from a predefined $p_z(z)$ for the Generator, and the network is learning to generate better and better samples through an adversarial game with a Discriminator. $p_z(z)$ is commonly chosen as $N(0,1)$, a normal (also known as Gaussian) distribution with mean 0 and variance 1. When the Generator is trained one can easily generate new data points simply by passing new z, sampled from the normal distribution.

On the subject of AEs

In an AE the network learns the latent z encoding through reconstruction, where the network first encodes a datapoint x to z using an Encoder network $E(x)$ and then reconstructs it using a Decoder $D(z)$. (The Decoder in this case is equivalent to the previous Generator, just named differently in this context)

The VAE improves on this method by introducing a regularizing term (KL Divergence, see section 2.4.6) which forces the $p_z(z)$ distribution to approximate a normal distribution (or any other chosen distribution), allowing for new samples to be generated by sampling from the known distribution and feeding it through the decoding part of the network.

GANs vs VAEs

A known advantages with GANs over VAEs are their ability to generate sharp valid samples from arbitrarily sampled points $z \sim p_z(z)$ compared to VAEs which more often results in slightly blurred representations (Dosovitskiy and Brox, 2016). This is due to the fact that GANs are trained exclusively from $z \sim p_z(z)$ and to generate convincing samples for every z to fool the Discriminator while the VAE is trained to only reconstruct a closed set of known data samples x. But while GANs are able to generate realistic details on the small scale they often fail on a large scale, introduce artifacts which makes the sample look unrealistic (Zhao et al., 2017) unless starting small and trained to generate progressively larger images as shown by Karras et al. (2017). While said artifacts may be undesired when trying to generate faces or animals like Radford, Metz, and Chintala (2015) and Karras et al. (2017) as the correct anatomy is so deeply rooted in us humans, they may instead be desired in the domain of product design as a way to provide creative mutations to what already exists. A disadvantage with GANs compared to VAEs though is that it is impossible to know beforehand which z encoding corresponds to which type of image without trial and error, while it is easy to just encode a sample and yield the corresponding z in an AE or VAE. A function which is very useful in case we want to interpolate between any two known samples to find intermediate designs, and to interpolate between particular features. For this reason we decide to focus primarily on VAEs but will explore a GAN implementation as well for comparison and to possibly generate more diverse "*creative*" designs.

5.2.3 Building a VAE: Fully Connected Architecture

As a first test a very simple VAE model is created, using only fully connected layers to train on the MNIST [14] dataset of 28x28px grayscale images with values in the domain $[0, 1]$. The network takes flattened 28x28px images as input (784px) and encodes a z-vector in two dimensions. It uses *ReLU* activation for speed and strong gradients and 20% dropout in the Decoders layers for robustness. The activation function for the μ and σ layers are linear to allow for any value; and the activation function on the final layer of the Decoder is Sigmoid, which squashes the pixel values back to the domain $[0, 1]$. The dense layer sizes of 200 and 100 are chosen arbitrary to form a funnel from 784 to z. A visual representation of the network architecture can be seen in figure 5.13 and full details of the Encoder and Decoder architectures are listed in tables 5.5 and 5.6 respectively.

[14] http://yann.lecun.com/exdb/mnist/

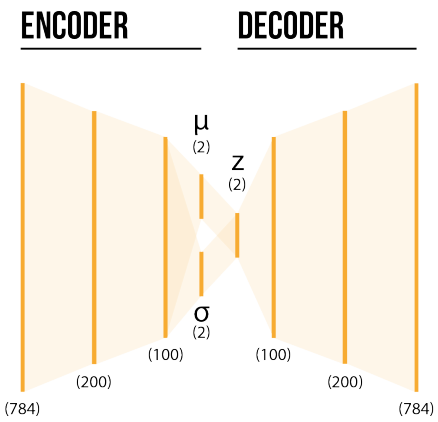

FIGURE 5.13: Illustration of the fully connected VAE architecture

Encoder E(x)				
Layer	Params	Kernel	Stride	Output Shape
Input				(784)
Dense *ReLU*	size = 200			(200)
Dense *ReLU*	size = 100			(100)
2 x Dense Linear	size = 2			2 x (2)

TABLE 5.5: Encoder architecture of the fully connected VAE in figure 5.13

Decoder D(x)				
Layer	Value	Kernel	Stride	Output Shape
Input				(2)
Dense *ReLU* Dropout	size = 100 amount = 0.2			(100)
Dense *ReLU* Dropout	size = 200 amount = 0.2			(200)
Dense Sigmoid	size = 784			(784)

TABLE 5.6: Decoder architecture of the fully connected VAE in figure 5.13

Training

The network is trained on the MNIST training set (50'000 28x28px grayscale images of hand drawn digits from 0-9), to minimize the VAE loss function (equation 2.35) constituting of reconstruction loss and KL-divergence. The network trained for 1000 epochs with a batch size of 500 on a 2.60 GHz Intel Core i5-3320M processor.

Results

The result after approximately 2 hours (1000 epochs) of training is a final loss of 6945. Figure 5.14 shows samples generated from latent vectors $z = (x, y)$ in the range $[-1, 1]$ on both axis. The model has successfully learned to represent the diversity in the samples as well as producing relatively sharp images. The model is also successful in transitioning between different samples.

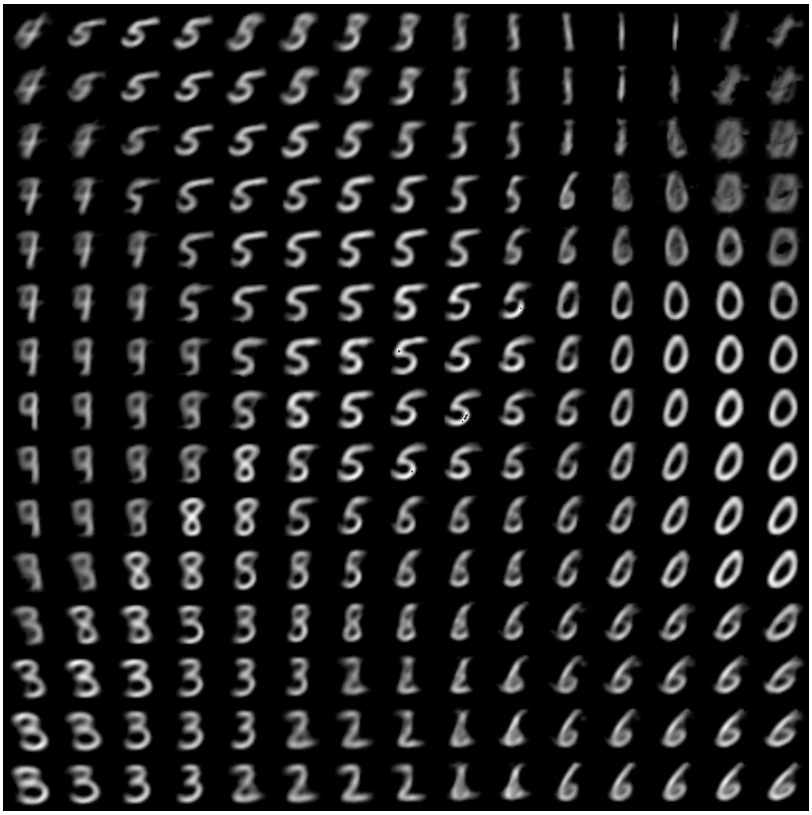

FIGURE 5.14: Sampled images from the fully connected VAE after 1000 epochs of training

Conclusion

This first experiment was successful but in order to generate larger images, also in color, a larger network is required. This poses some difficulties as the number of parameters in a dense network grows exponentially with layer size. To combat this problem the next natural step is to look at convolutional architectures and add such layers to our model. Deep convolutional architectures has also been shown to outperform regular fully connected networks, especially in image classification tasks, as described in section 2.6.4.

5.2.4 Building a VAE: Deep Convolutional Architecture

This architecture extends the previous (fully connected) architecture by adding convolutional layers before and after the dense Auto Encoder, together with MaxPooling and UpSampling layers to decrease and increase layer sizes. All convolutional layers have 16 filters and shape (3 x 3) kernels. MaxPooling and UpSampling layers

uses a factor of 2 to half and double the layer sizes respectively. The Dense bottleneck layers are kept at 200, 100 and with a z-layer of 2 as before but with the addition of dropout now also in the encoding part of the network. The network is setup to take 64x64px images in grayscale but a variation of the network with 3 channel output for color is also tested. A visual representation of the network architecture can be seen in figure 5.15 and full details of the Encoder and Decoder architectures are listed in tables 5.7 and 5.8 respectively.

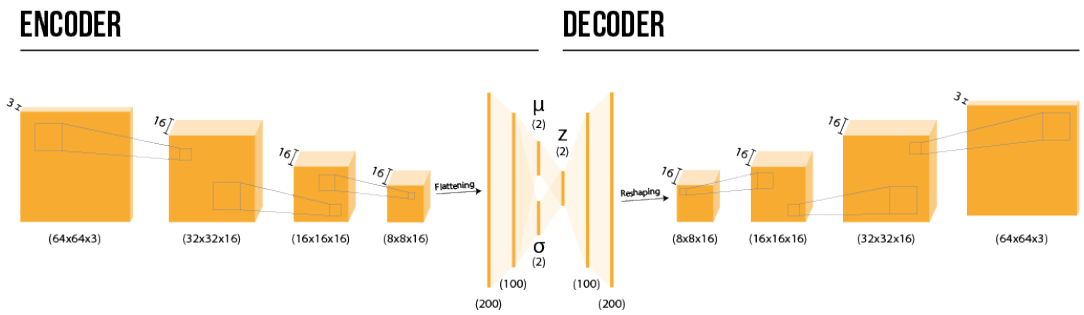

FIGURE 5.15: Illustration of the deep convolutional VAE architecture

Encoder E(x)				
Layer	Params	Kernel	Stride	Output Shape
Input				(64, 64, 3)
Conv2D MaxPool *ReLU*	filters = 16	[3 x 3] [2 x 2]	1	(64, 64, 16) (32, 32, 16)
Conv2D MaxPool *ReLU*	filters = 16	[3 x 3] [2 x 2]	1	(32, 32, 16) (16, 16, 16)
Conv2D MaxPool *ReLU*	filters = 16	[3 x 3] [2 x 2]	1	(16, 16, 16) (8, 8, 16)
Flatten Dropout	amount = 0.4			(1024)
Dense *ReLU*	size = 200			(200)
Dense *ReLU*	size = 100			(100)
2 x Dense Linear	size = 2			2 x (2)

TABLE 5.7: Encoder architecture of the deep convolutional VAE in figure 5.15

Decoder D(x)				
Layer	Params	Kernel	Stride	Output Shape
Input				(2)
Dense Dropout *ReLU*	size = 100 amount = 0.2			(100)
Dense Dropout *ReLU*	size = 200 amount = 0.2			(200)
Dense *ReLU* Reshape	size = 1024			(1024) (8, 8, 16)
Conv2D UpSampling *ReLU*	filters = 16	[3 x 3] [2 x 2]	1	(8, 8, 16) (16, 16, 16)
Conv2D UpSampling *ReLU*	filters = 16	[3 x 3] [2 x 2]	1	(16, 16, 16) (32, 32, 16)
Conv2D UpSampling *ReLU*	filters = 16	[3 x 3] [2 x 2]	1	(32, 32, 16) (64, 64, 16)
Conv2D Sigmoid	filters = 3	[3 x 3]	1	(32, 32, 3)

TABLE 5.8: Decoder architecture of the deep convolutional VAE in figure 5.15

Training

The network is first trained on our windows dataset in grayscale, scaled to 64x64px. The MNIST dataset has not been used for comparison since those images are much smaller in size and carry less than 25% as much information. A modified version of the network with color output is then trained on a small dataset of furniture. The furniture dataset is used to verify that the network is able to output colored images in a meaningful way. Finally the network is trained on the windows dataset in color. All training sessions were performed on a GeForce 1070 GPU with 6GB VRAM. Details of each training session can be seen in table 5.9.

Training Details: Deep Convolutional VAE							
Dataset	Channels	Samples	Batch Size	Optimizer	η	Epochs	Time
Windows	grayscale	4'564		Adam	2E-04	10'000	48h
Furniture	RGB	50	50	Adam	2E-04	10'000	15h
Windows	RGB	4'564	400	Adam	2E-04	2'000	20h

TABLE 5.9: Summary of training parameters used for each training session for the deep convolutional VAE

Results

Results of the first training run of grayscale windows can be seen in figure 5.16 showing how the learned distributions is changing between epochs. Each image show

samples generated from latent vectors $z = (x, y)$ in the range $[-1, 1]$ on both axis. A larger sample of the learned distribution of grayscale images can be seen in figure 5.17. Images for the second run of colored furniture and third run of colored windows can be seen in figures 5.18, 5.19, and 5.20, 5.21 accordingly.

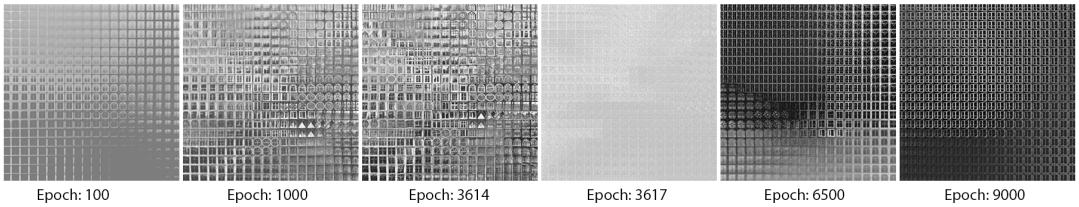

FIGURE 5.16: Images sampled on different epochs during training of the deep convolutional VAE on the windows dataset in grayscale, showing the changes in distribution (larger version can be seen in appendix B)

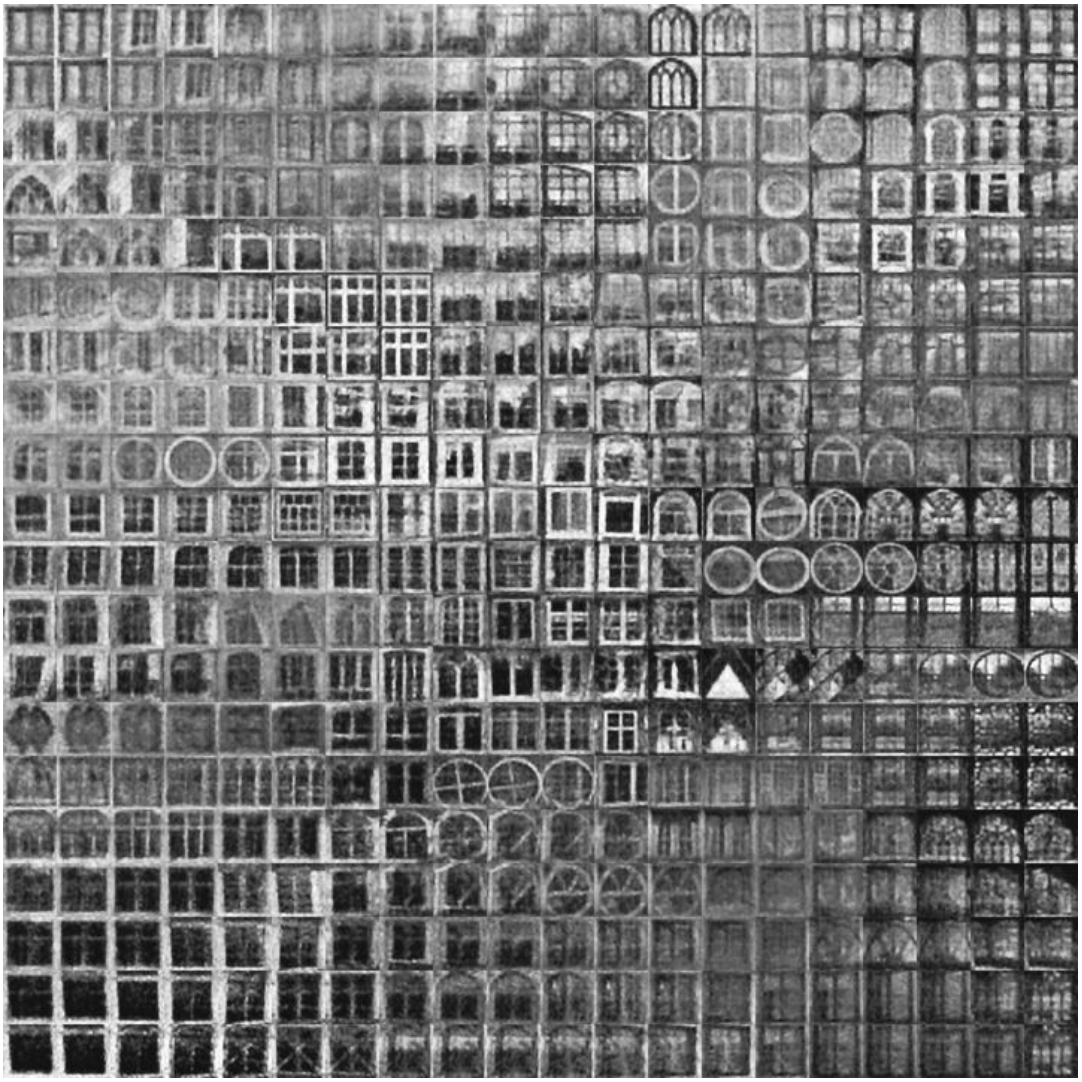

FIGURE 5.17: Sampled images from the deep convolutional VAE, trained on the windows dataset in grayscale, showing the learned distribution

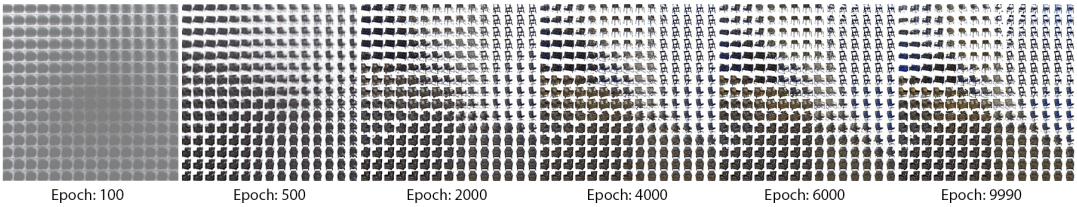

FIGURE 5.18: Images sampled on different epochs during training of the deep convolutional VAE on the furniture dataset in color, showing the changes in distribution (larger version can be seen in appendix B)

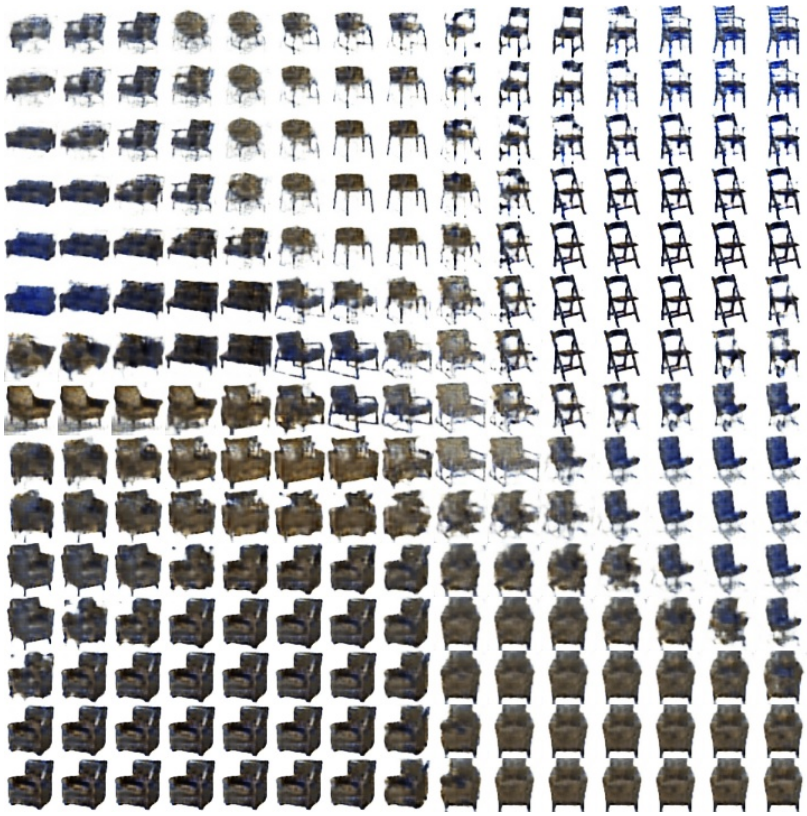

FIGURE 5.19: Sampled images from the deep convolutional VAE, trained on the furniture dataset in color, showing the learned distribution

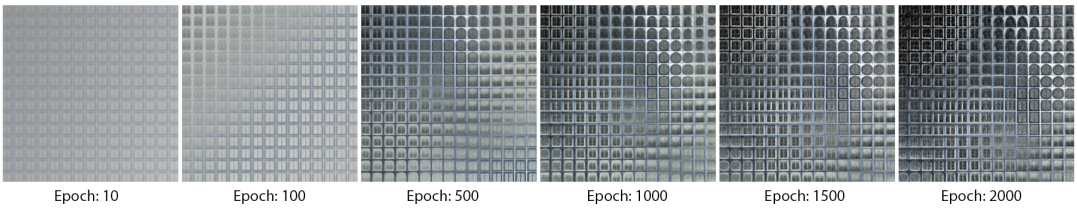

FIGURE 5.20: Images sampled on different epochs during training of the deep convolutional VAE on the windows dataset in color, showing the changes in distribution (larger version can be seen in appendix B)

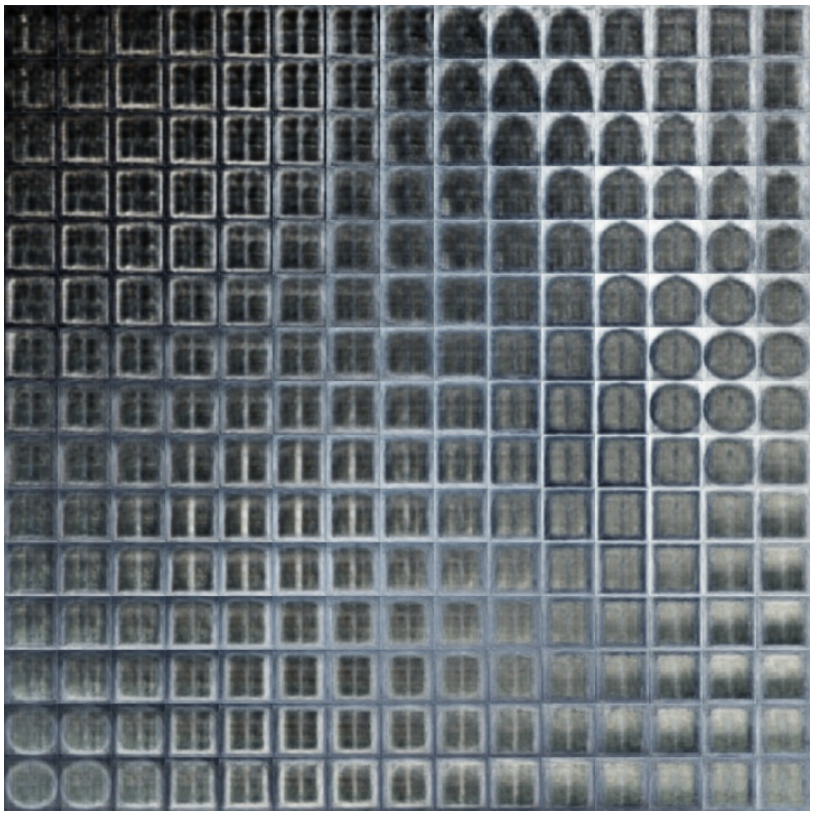

FIGURE 5.21: Sampled images from the deep convolutional VAE, trained on the windows dataset in color, showing the learned distribution

Conclusion

This experiment produced some promising results. The training run on grayscale windows resulted in a very diverse distribution of windows and the network has managed to generalize quite well. Viewing the generated distribution in figure 5.17, one can see that windows which are similar to one another are located close to each other (not as good as we had hoped at this point but still promising) similar to the MNIST results in figure 5.14. In the top-left corner we can see samples of windows typically taken from outside, while in the lower-right corner we can see windows as typically viewed from the inside, with some exceptions. Noteworthy is that the learning converged quite early, at around 2000 epochs, to later disrupt completely, at epoch 3615, and never recover. It is difficult to say what exactly caused this disruption but our belief is that it might have been caused by a high learning rate and some oscillation in the optimization, which caused great parts of the network to die out. Since *ReLU* activations are used, nodes with negative activations is set and kept at 0, which might explain why the network never fully recovered.

The training run with the furniture dataset in color also gave some good results. We never expected the generated distribution to be very diverse, nor show any smooth transitions, as the dataset is so small (only 50 samples) but in terms of color reconstruction of the images it were provided it is doing a good job. In figure 5.19 it is also possible to see how the learned color distribution has been transferred to other furniture as well such as red chairs top- and mid-right.

The final training session on colored windows also converged quite early around epochs 1500-2000 to produce the distribution seen in figure 5.21. The diversity expressed in this distribution is far less compared to the results in figure 5.17 and in terms of color diversity it is nowhere near the supplied dataset. These flaws aside the network is managing to generalize quite well to generate the most common types of windows with decent transitions. This lack of detail compared to the grayscale run is probably due to a saturation in the network, where the number of convolution filters provided are too few to preserve details in all three color channels while it was enough for single channel. It could also be that the 2-dimensional z-layer is too small, so it might be worth to move up in dimensions even though it would be harder to visualize.

5.2.5 Building a VAE: Improved Deep Convolutional Architecture

To improve upon the previous results we adapt a modified version of the DCGAN architecture (see figure 5.22) by Radford, Metz, and Chintala (2015) which has proven to be very successful in its domain. The DCGAN architecture is commonly used as the baseline when comparing new state-of-the-art GAN networks. In the DCGAN paper Radford, Metz, and Chintala (2015) presents the following guidelines for stable Deep Convolutional GANs:

1. Replace any pooling layers with strided convolutions and upsampling layers with fractional-strided convolutions.
2. Use batch normalization throughout both networks (Generator and Discriminator).
3. Remove fully connected, dense layers for deeper architectures.
4. Use *ReLU* activation in Generator and *LeakyReLU* in Discriminator.

While these insights may not necessarily be applicable for VAE training they could still be worth a try since it may show promising results in the VAE application as well.

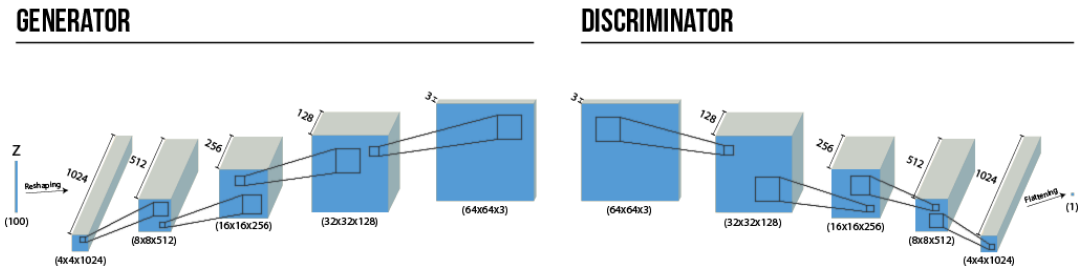

FIGURE 5.22: Illustration of the DCGAN architecture

By adapting the DCGAN's Discriminator as our Encoder and the DCGANs Generator as our Decoder, with some minor adjustments to keep our z sampling layers in the bottleneck, we end up with the following architecture (see tables 5.10 and 5.11), which can be seen in figure 5.23. The number of filters used for each convolutional layer has been increased to be closer in range to the DCGAN architecture. The final activation is kept to logistic Sigmoid (instead of Tanh as used in the DCGAN's Generator) to stay consistent with our previous networks and allow us to use the

same visualization functions as before. The sampling layers (μ, σ, z) have been extended from 2 nodes to 100 to allow for a richer encoding and hopefully much better reconstruction capabilities.

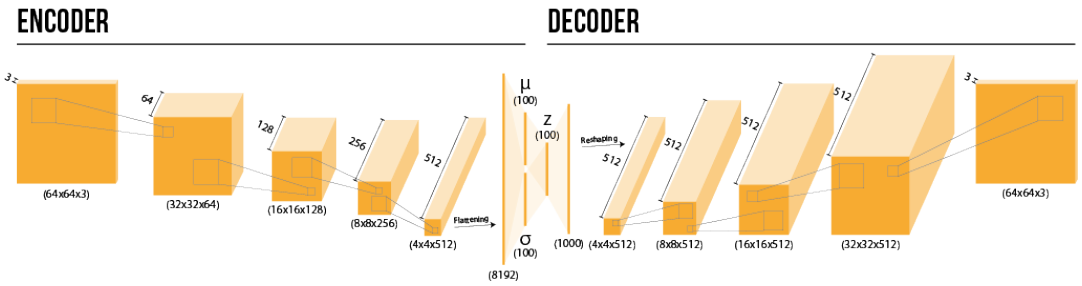

FIGURE 5.23: Illustration of our VAE architecture after adapting a modified version of the DCGAN architecture

Encoder E(x)				
Layer	Params	Kernel	Stride	Output Shape
Input				(64, 64, 3)
Conv2D *LeakyReLU* BatchNorm	filters = 64 alpha = 0.2 momentum = 0.8	[5 x 5]	2	(32, 32, 64)
Conv2D *LeakyReLU* BatchNorm	filters = 128 alpha = 0.2 momentum = 0.8	[5 x 5]	2	(16, 16, 128)
Conv2D *LeakyReLU* BatchNorm	filters = 256 alpha = 0.2 momentum = 0.8	[5 x 5]	2	(8, 8, 256)
Conv2D *LeakyReLU* BatchNorm	filters = 512 alpha = 0.2 momentum = 0.8	[5 x 5]	2	(4, 4, 512)
Flatten Dropout	 amount = 0.4			(1, 1, 8192)
2 x Dense Linear	size = 100			2 x (1, 1, 100)

TABLE 5.10: Encoder architecture of the deep convolutional VAE in figure 5.23

Decoder D(z)				
Layer	Params	Kernel	Stride	Output Shape
Input				(100)
Dense Tanh Dropout Reshape	size = 1000 amount = 0.4 			(1000) (1, 1, 1000)
Conv2DTransp *ReLU* BatchNorm	filters = 512 momentum = 0.8	[4 x 4]	1	(4, 4, 512)
Conv2DTransp *ReLU* BatchNorm	filters = 512 momentum = 0.8	[8 x 8]	2	(8, 8, 512)
Conv2DTransp *ReLU* BatchNorm	filters = 512 momentum = 0.8	[5 x 5]	2	(16, 16, 512)
Conv2DTransp *ReLU* BatchNorm	filters = 512 momentum = 0.8	[5 x 5]	2	(32, 32, 512)
Conv2DTransp Sigmoid	filters = 3	[5 x 5]	2	(64, 64, 3)

TABLE 5.11: Decoder architecture of the deep convolutional VAE in figure 5.23

Training

This network is trained on the windows dataset in color scaled to 64x64px in size. Training was performed on a single Nvidia Titan X GPU with 12GB VRAM and the learning rate was reduced after every 2'000 epochs. The model convergence after approximately 10'000 epochs and 80 hours of training. Precise training details are listed in table 5.12.

Training Details: Improved Deep Convolutional VAE						
Dataset	Channels	Samples	Batch Size	Optimizer	η	Epochs
Windows	RGB	4'564	415	Adam	2E-05	0 - 3'000
Windows	RGB	4'564	415	Adam	1E-05	3'000 - 5'000
Windows	RGB	4'564	415	Adam	1E-06	5'000 - 7'500
Windows	RGB	4'564	415	Adam	1E-07	7'500 - 11'000
Total time elapsed: ∼80h (∼26s / epoch)						

TABLE 5.12: Details on the training parameters used to train the improved deep convolutional VAE

Results

Because the z-vector contains 100 variables this time it is no longer possible to visualize the learned distribution in all axis in a regular 2D plot, only two at a time if the rest is kept constant. Figure 5.24 show samples generated from latent vector $z = (x, y, n_3, n_4, .., n_{99}, n_{100})$ with x, y in the range $[-1, 1]$ on both axis and $n_{i \in [3,100]} =$

0 for selected epochs. While not showing the true distribution and what is being learned in the other 98 dimensions this is still enough to tell how well the network is learning in terms of accuracy and sharpness in the samples produced.

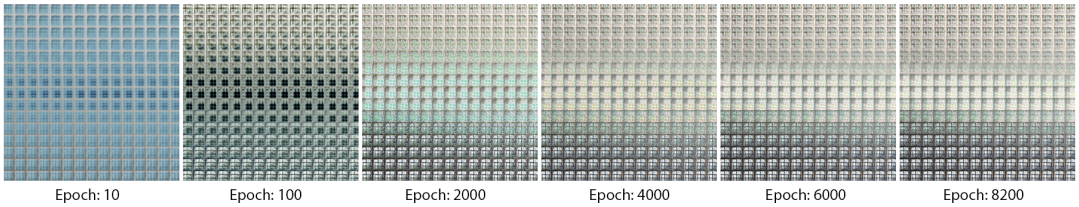

FIGURE 5.24: 100 randomly sampled images from different epochs during the training of the improved deep convolutional VAE (larger version can be seen in appendix B)

To better probe the diversity of the learned distribution from the entire z-space the following image is generated (figure 5.25) which shows 100 random samples generated from $\boldsymbol{z} \sim N(0, 2) \in \mathbb{R}^{100}$.

FIGURE 5.25: 100 randomly sampled images from the improved deep convolutional VAE, trained on the windows dataset in color (larger version can be seen in appendix B)

Reconstruction of Encoded Data

So what can we do with this model now when it is trained? One thing we can try is reconstruction of input data. By passing images through the encoder network corresponding z-vectors are obtained which when parsed by the decoding half of the network reconstructs the original image. Figure 5.26 show how well the network is performing when it comes to reconstructing 50 random samples from the training set. Note that these results are expected to be good as it was the task the network

was trained to do. Images are presented in pairs with generated image on the top and the real image on the row below.

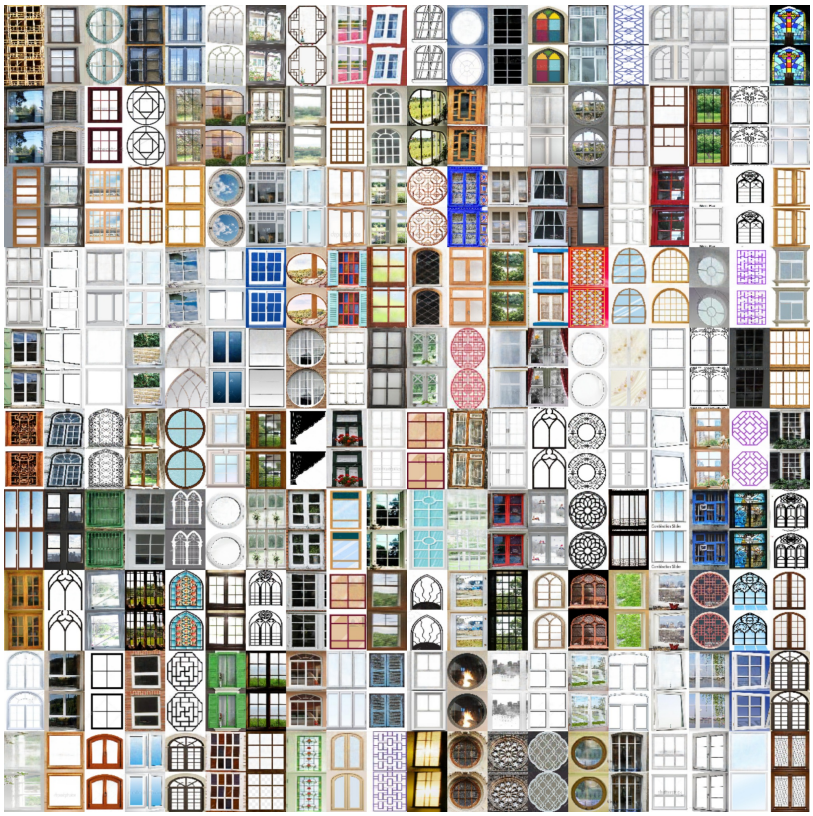

FIGURE 5.26: 50 random images from the dataset (bottom rows) and the reconstructions (top rows) generated from the improved deep convolutional VAE

How about reconstruction of new windows never seen before? Figure 5.27 shows a reconstruction of 8 windows not included in the training set. The network is expected to be able to reconstruct the general shape and color of the window frame but not necessarily what is going on in the background. Images are presented in pairs with generated image on the top and the reference image on the row below.

FIGURE 5.27: The improved deep convolutional VAEs generated reconstructions of images outside the dataset (top rows) and their corresponding original images (bottom rows)

Latent Space Interpolation

By interpolating between two latent vectors several intermediate designs can be generated. Figure 5.28 shows linear interpolation between five randomly selected pairs of windows from the dataset (in 49 steps from top-left to top-right).

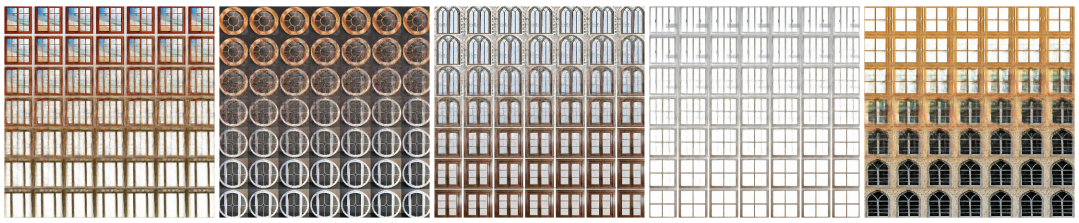

FIGURE 5.28: Linear latent space interpolations between two randomly selected windows from the dataset (top left to bottom right) (larger version can be seen in appendix B)

Note that this is done in the latent (z) space on the encoding before generating the images which result in more meaningful intermediate designs than from interpolating in pixel space (an example of this difference can be seen in figure 5.29).

Latent Space Arithmetic

It has also been shown by Bojanowski et al. (2017), Radford, Metz, and Chintala (2015) and others that the learned latent space achieves to encode higher level features and concepts in a way that vector arithmetic in the latent space can produce meaningful results. What happens if we encode a square window with mullions, subtract the encoding of a square window without mullion, and then add the encoding of a circular window without mullions, and then generate an image from that ($D(z_1 - z_2 + z_3)$)? Figure 5.30 show how the answer actually is a circular window

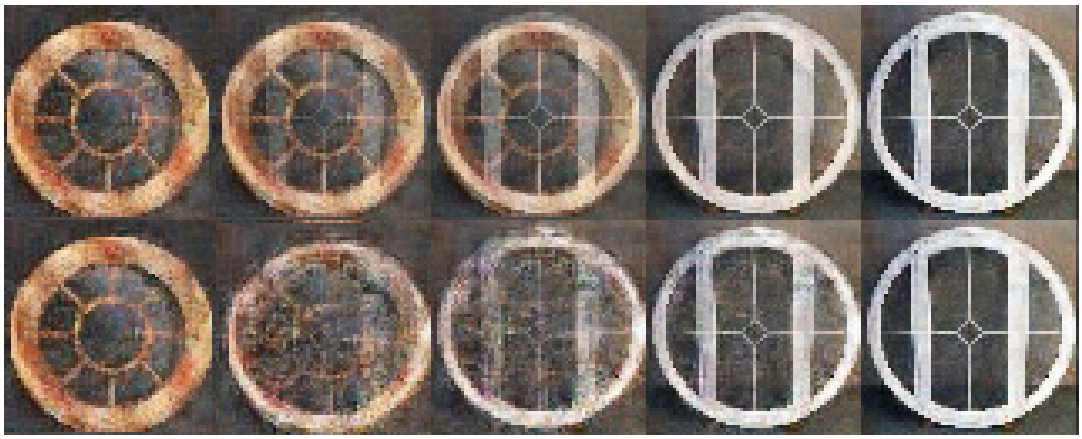

FIGURE 5.29: Interpolation in pixel space (top) compared to interpolation in encoded latent space (bottom)

with mullions. This can be compared to how Radford, Metz, and Chintala (2015) managed to extract the meaning of sunglasses from data of faces with and without sunglasses, and then add sunglasses to faces without. The bottom row in figure 5.30 shows the output from the same arithmetic but using the averaged encoding of the ones above, which yield slightly better results. Interpolating the mullion amount results in figure 5.31.

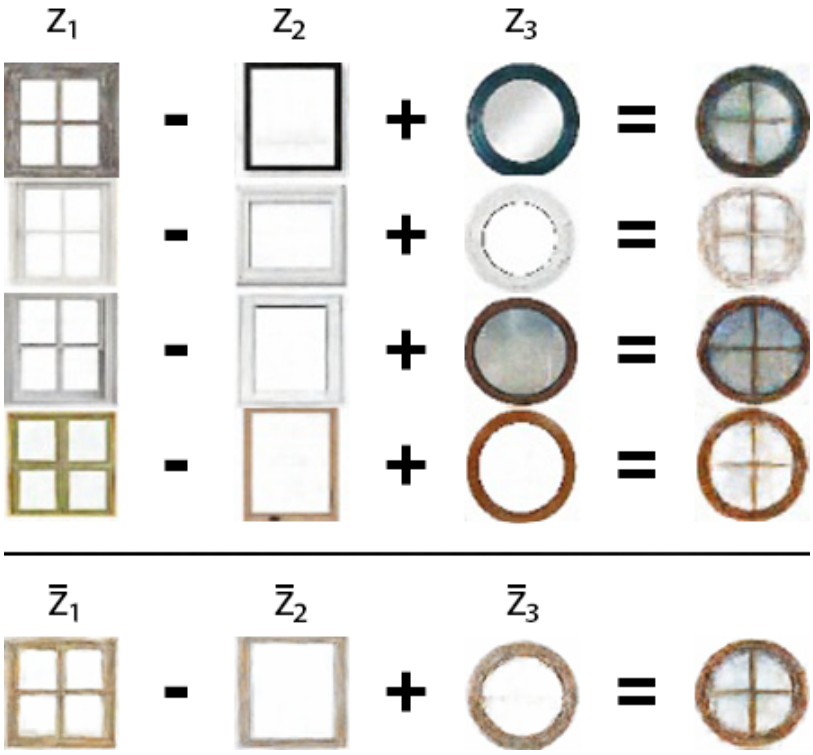

FIGURE 5.30: The successful clustering of meaningful features shown through latent space arithmetic. $result = D(E(\boldsymbol{z_2}) - E(\boldsymbol{z_1}) + E(\boldsymbol{z_3}))$

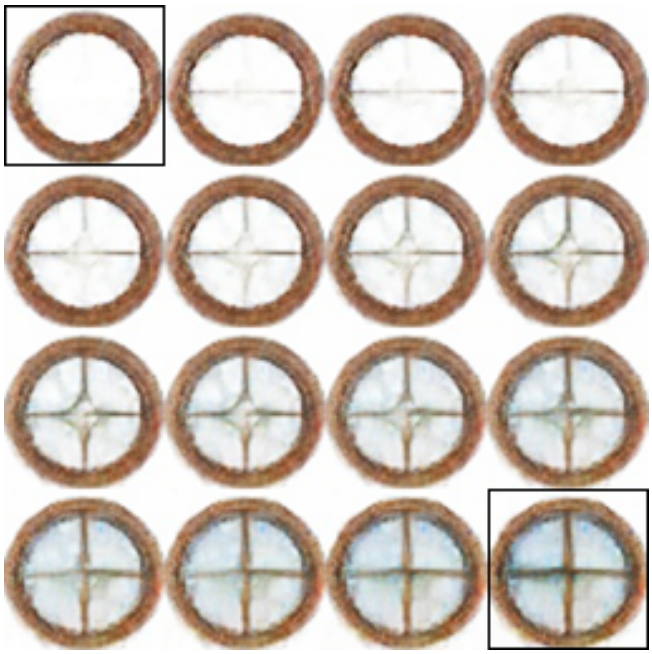

FIGURE 5.31: Generated images from latent space interpolation of a window without mullions (top-left) to a window with mullion (bottom-right), where the window with mullion was generated by adding the representation of a square window with mullions and subtracting a square window without mullions.

Conclusion

The results achieved from the final version of the VAE show a great diversity in both shape, style and color which show that the network has learned a good representation of the data distribution. It also shows exceptional reconstruction of the training samples and decent generalization to validation samples. It is possible that the network might have managed to generalize even better to validation samples with a larger dataset which would have reduced the amount of overfitting. The successful results from latent space interpolation and latent space vector arithmetic also shows that the model has learned some sort of higher level representation of the data. Many of the intermediate samples in the interpolations are blurry which likely is due to a combination of overfitting to known data and limitations in the VAE architecture.

5.2.6 Building a GAN: Improved W-GAN Architecture

In an attempt to generate more creative new design suggestions of windows which are also sharper we implement a Wasserstein GAN for comparison. Both Generator and Discriminator (Critic) are built as deep convolutional networks similar to the DCGAN (figure 5.22) architecture but with some slight variations in filter and layer sizes. As an extra experiment we also construct this network extra deep to produce samples of much higher resolution: 256x256px instead of 64x64. This is done by extending both Generator and Critic with two additional upsampling and convolutional layers. Following the advice given by Radford, Metz, and Chintala (2015) with the DCGAN architecture the network uses batch normalization after all convolutional layers except input and output. The Generator uses *ReLU* activation and the Critic uses *LeakyReLU*. The final activation is *Tanh* for the Generator and is *linear* for the Critic. A visual representation of the network architecture can be seen

in figure 5.32 and full details of the Generator and Critic architectures are listed in tables 5.13 and 5.14 respectively.

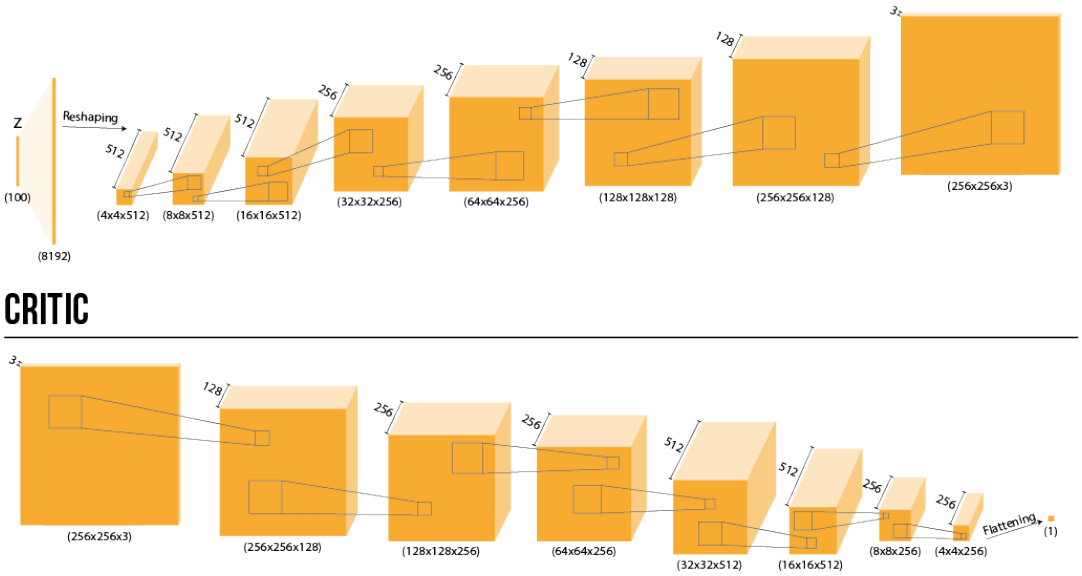

FIGURE 5.32: Illustration of the implemented W-GAN architecture

Generator G(x)				
Layer	Params	Kernel	Stride	Output Shape
Input				(100)
Dense *ReLU* Reshape	size = 8192			(8192) (4, 4, 512)
UpSampling2D Conv2D BatchNorm *ReLU*	filters = 512 momentum = 0.8	[2 x 2] [4 x 4]	1	(8, 8, 512)
UpSampling2D Conv2D BatchNorm *ReLU*	filters = 512 momentum = 0.8	[2 x 2] [4 x 4]	1	(16, 16, 512)
UpSampling2D Conv2D BatchNorm *ReLU*	filters = 256 momentum = 0.8	[2 x 2] [4 x 4]	1	(32, 32, 512) (32, 32, 256)
UpSampling2D Conv2D BatchNorm *ReLU*	filters = 256 momentum = 0.8	[2 x 2] [4 x 4]	1	(64, 64, 256)
UpSampling2D Conv2D BatchNorm *ReLU*	filters = 128 momentum = 0.8	[2 x 2] [4 x 4]	1	(128, 128, 256) (128, 128, 128)
UpSampling2D Conv2D BatchNorm *ReLU*	filters = 128 momentum = 0.8	[2 x 2] [4 x 4]	1	(256, 256, 128)
Conv2D BatchNorm	filters = 3 momentum = 0.8	[4 x 4]	1	(256, 256, 128)
Conv2DTransp Tanh	filters = 3	[4 x 4]	1	(256, 256, 3)

TABLE 5.13: Generator architecture of the W-GAN in figure 5.32

Critic C(z)				
Layer	Params	Kernel	Stride	Output Shape
Input				(256, 256, 3)
Conv2D *LeakyReLU* Dropout	filters = 128 alpha = 0.2 amount = 0.25	[3 x 3]	2	(128, 128, 128)
Conv2D BatchNorm *LeakyReLU* Dropout	filters = 256 momentum = 0.8 alpha = 0.2 amount = 0.25	[3 x 3]	2	(64, 64, 256)
Conv2D BatchNorm *LeakyReLU* Dropout	filters = 256 momentum = 0.8 alpha = 0.2 amount = 0.25	[3 x 3]	2	(32, 32, 256)
Conv2D BatchNorm *LeakyReLU* Dropout	filters = 512 momentum = 0.8 alpha = 0.2 amount = 0.25	[3 x 3]	2	(16, 16, 512)
Conv2D BatchNorm *LeakyReLU* Dropout	filters = 512 momentum = 0.8 alpha = 0.2 amount = 0.25	[3 x 3]	2	(8, 8, 512)
Conv2D BatchNorm *LeakyReLU* Dropout	filters = 512 momentum = 0.8 alpha = 0.2 amount = 0.25	[3 x 3]	2	(4, 4, 256)
Flatten Dense Linear	size = 1			(2048) (1)

TABLE 5.14: Critic architecture of the W-GAN in figure 5.32

Training

The Wasserstein GAN is trained on the windows dataset in color scaled to 256x256px in size. Training was performed on a single Nvidia Titan X GPU with 12GB VRAM using RMSprop as optimizer, a batch size of 16 and a learning rate of 10e-5. Other parameters were set as proposed by the improved W-GAN paper (Gulrajani et al., 2017). The training was stopped after 90 hours and 41500 epochs due to time constrains. Training results can be seen in section 5.2.7 below.

5.2.7 Training Results

Figure 5.33 shows a collection of 25 samples generated by feeding the Generator with noise vectors $z \sim N(0, 1)$ sampled from a normal distribution.

FIGURE 5.33: 25 randomly sampled images from the W-GAN, trained on the windows dataset in color (more samples can be seen in appendix B)

5.2.8 Conclusion

As we can see by the generated samples in figure 5.33 the Wasserstein GAN has sure learned to generate some window like images. These images are impressively sharp and look very different from the ones generated by the VAE in figure 5.25 with much more unique designs. The Generator seem to have grasped the concept of mullions and window frames quite well, managing to generate continuous closed structures with sharp edges. The model also generates some very organic designs which are more asymmetric than anything generated by the VAE previously. These global artifacts were expected since it is a known problem with deep GANs (producing images much larger than 64x64px) which are not trained progressively. But in this case these artifacts worked out quite nice in our favor by producing more creative designs. These designs are far from perfect, many of the images still show a lot of high frequency noise and some are still very blurry but they show great potential of what could be done. It is also not unlikely that the model could have improved to generate even better and more realistic samples if it was left to train further.

Chapter 6

Framework

The following section presents a framework for how deep learning and generative models can be selected, developed and implemented to automate parts of the product design process. The framework consists of three parts which follows from the Theory Study in chapter 2, the Concept Development in chapter 4 and the Concept implementation in chapter 5 as it was laid forward in section 1.5:

1. A Morphological Matrix of identified potential implementations in the PDP
2. A General Implementation Process
3. A summary of Common Problems, Challenges, Strategies and Solutions

6.1 Morphological Matrix

The result of the Concept Development phase in chapter 4 is the morphological matrix shown in table 4.1, connecting concepts of implementation possibilities to different parts of the product design process, to use as inspiration and a starting point when incorporating deep learning in the product design process. The matrix is meant to be expanded upon by the user.

6.2 General Implementation Process

Through the Concept Implementation phase in chapter 5 a generalized implementation process of seven key steps has been identified. This process with its sub-steps is presented in figure 6.1.

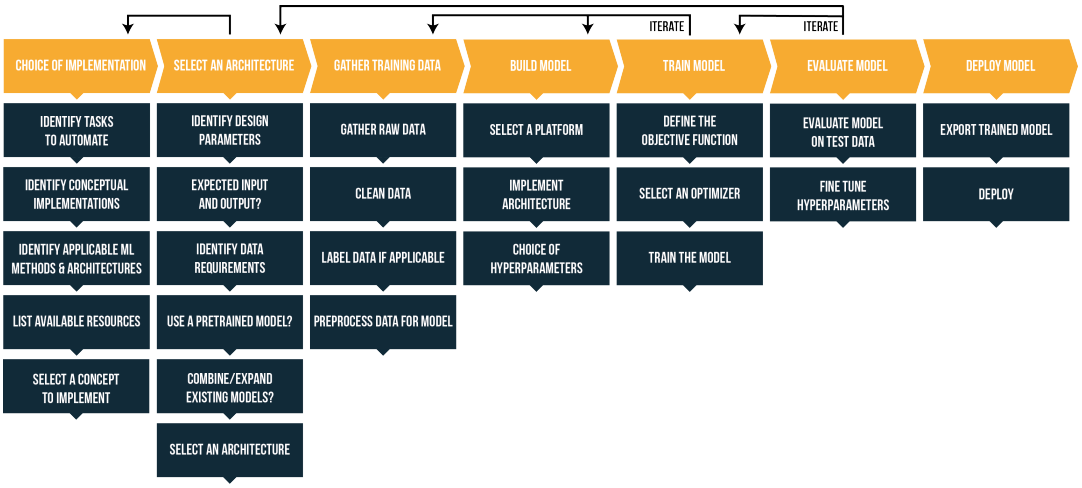

FIGURE 6.1: General Implementation Process of DL-based models to automate parts of the PDP (see appendix B for a larger version)

A more detailed description of each sub-step as well as common problems, challenges, strategies and solutions are presented below.

6.2.1 Step 1 - Choice of Implementation

Before implementing AI it is important to identify and clarify what is profited by doing so: does it save time, cut costs or in some other way improve the process? The following steps creates a summary of factors which affect the conditions of implementation. With the summary it is easier to decide if, how and where AI could benefit the process and where it is worth starting.

Step 1.a - Identify Tasks to Automate

Begin by breaking down your PDP into smaller parts or *sub-processes* and place them in a list. Create two new columns with *Time* and *Complexity* (the need for special competences). Estimate how great the time or complexity benefits of automation would be for each part and add it to respective column.

Step 1.b - Identify Conceptual Implementations

Identify possible implementations in a conceptual form through usage of existing techniques, either as they are or by modifying some parameters and connect the concepts to parts of the process, or the process as a whole. The morphological matrix presented in section 6.1 is a good starting point to build from. Add concepts and modify the matrix to fit the current process with its sub-processes.

Step 1.c - Identify Applicable ML Methods and Architectures

Decide on a DL method to use and identify which neural network architectures could be used in order to implement the concepts. Since different architectures can be trained in different ways it is important to consider what kind of training data is required and how accessible that data is. The availability of data will effect the choice of DL method. Add three more columns to the list with *Architecture*, *Training Data* and *Data Availability*.

Step 1.d - List Available Resources

List the resources that are available for the project, such as time, expertise and computing power. Weigh the benefits of using computational resources in-house versus renting cloud solutions.

Step 1.e - Select a Concept to Implement

The list now shows a collection of possible implementations, see table 6.1, their benefits to the process, their applicable architectures, what training data is required and the availability to that data. These are the major factors to consider before implementation. What kind of architecture and data is used greatly affect the training difficulties and processing power required to train. Weigh the benefits against the difficulties and if necessary break down the process further to find a good implementation.

Sub-processes	Time	Complexity	Architectures	Data	Data Availability
Design Ideas	Minor	Average	VAE, GAN	Images	Available
...	...	Low	RNN	Videos	Gather
...	...	...	...	...	...
...	...	...	...	...	...
...	...	...	...	...	...
Modelling	Average	High	CNN, FFNN	3D Mesh	Create

TABLE 6.1: Example: list of factors that affect the conditions of implementation

6.2.2 Step 2 - Select an Architecture

Selecting what type of neural network to implement and how to modify it to solve a specific problem is a challenging task. One of the best things to do is to look at what others have done in research related to the problem at hand, to identify pros and cons of different methods. The following steps covers some key points which are worth to consider.

Step 2.a - Identify Design Parameters

Do the selected implementation have any design parameters? Clarify what they are and reason about how they could be incorporated into the system. Some parameters might be feasible to feed directly to the deep learning system as a condition while others are parsed by ordinary functions first.

Step 2.b - Expected Input and Output?

What are the expected input and outputs of the system? If the problem can be rephrased as a function you know it can be solved using a neural network, according to the universal approximation theorem (see section 2.4.1), given enough data and the appropriate network. By doing this you may also identify weather or not you can break down the problem into sub-problems which could be solved individually with machine learning or implemented analytically.

Step 2.c - Identify Data Requirements

What data are we processing? Different types of architectures are developed to work with different types of data, such as images, binary data, 3D mesh representations etc. By identifying what data the implementation will work with or might need to work with, unfeasible network architectures can be discarded unless there are potential means to modify them.

Depending on the size of the input and output data and what it represent the depth and size of the neural network will also be effected as more or less up/down-sampling layers may be required.

Step 2.d - Use a Pretrained Model?

Also consider the possibility of using an existing model with pretrained weights. Using a ***pretrained model*** saves much time, since the model already has learned to extract some features, but puts some restraints on what architectures are possible to use. In order to load the weights, the layersizes of both models must be of equal size. It is however possible to only load weights into some of the layers and add more layers without pretrained weights. If choosing to use pretrained weights it is important to keep in mind what kind of dataset has been used for that training. For example, a model trained on generating chairs is easier to train to generating tables than a model trained to generate animals.

Step 2.e - Combine / Expand Existing Models?

It is also worth to consider combining multiple existing models or use one or more models as a base for building deeper or more complex architectures. Existing models are usually well developed and can form a stable starting point to not start from scratch.

Step 2.f - Select an Architecture

According to the considerations above select the architecture or architectures which feel the most promising. After implementation and testing it is not uncommon to iterate on this original decision and try slight variations of the architecture to improve the results.

6.2.3 Step 3 - Gather Training Data

In order to train the model training data is required. Depending on what kind of learning method (section 2.5) and architecture (section 2.6) is chosen, the type and amount of training data required differ.

Step 3.a - Gather Raw Data

Many existing datasets can be used for free but in many cases these datasets might not have the required data for a custom area of implementation. Then the data needs to be either created or gathered, cleaned and preprocessed. In the case where gathering of a vast amount of data is required it takes a long time to gather manually. Then an automatic gathering system, as the one created in section 5.1.3, is very useful. Similar systems can be built in order to collect other types of data, such as 3D models, videos or text documents.

Step 3.b - Clean Data

It is important to have good data since the result after training of a neural network is based on the data it trains on. For that reason once the data is gathered it could be worth taking the time to clean the data by going through the dataset and remove undesired entries or clean the representation. This process can of course also be automated by implementing other deep learning systems or algorithms.

Step 3.c - Label Data if Applicable

If labeled data is required, as conditional data or for supervised learning based methods, and is not available it need to be labeled either manually or automatically by a system. Consider to train a classifier on a subset of manually labeled data if a lot of labeled data is required.

Step 3.d - Preprocess Data for Model

Depending on the selected network architecture it may be required to preprocess the data to fit the required shape of the input. It can also be worth augmenting the dataset by generating copies of the data with slight perturbations, to expand the dataset and make the model more robust to small changes in the target data. It is also common to split the dataset into three: one training set, one evaluation set and one test set.

6.2.4 Step 4 - Build Model

While a deep learning system can be implemented from scratch in any programming language it is commonly more convenient to use one of the many existing frameworks and libraries for machine learning and data analysis to do so. Among the current most popular alternatives are TensorFlow[1], Theano[2], Keras[3], Torch[4], Caffe2[5] and CNTK[6]. Libraries like Keras (which runs with either TensorFlow, Theano or CNTK in the background) comes with the essential building blocks for a neural network (e.g. layers, optimizers, loss and activation functions) predefined to easily be used in any project, enabling non-expert users access to powerful deep learning tools.

Step 4.a - Select a Platform

Depending on which platform to build the model on and in which programming language, select any appropriate library or framework from the ones listed above or other alternatives.

[1] https://www.tensorflow.org/
[2] http://deeplearning.net/software/theano/
[3] https://keras.io/
[4] http://torch.ch/
[5] https://caffe2.ai/
[6] https://docs.microsoft.com/en-gb/cognitive-toolkit/

Step 4.b - Implement Architecture

Implement the selected model architecture in the selected library or framework using the programming language of choice. Many existing implementations of popular architectures using various libraries can be found online (e.g. on GitHub) as reference.

Step 4.c - Choice of Hyperparameters

Select appropriate hyperparameters following research recommendations and existing implementations. These parameters, such as layer sizes, batch size, regularization factors and learning rate, are commonly fine tuned in between training sessions to find more optimal settings.

6.2.5 Step 5 - Train Model

Different models and architecture are trained differently and has its respective challenges, strengths and weaknesses. However there are several similarities in training challenges. We recommend trying the strategies and methods presented below to solve the problems, which it does in most cases, before looking for new state of the art solutions to specific problems.

Step 5.a - Define the Objective Function

The ***Objective Function*** (commonly a loss function) is probably most important since it is an expression of what to optimize. Choosing what objective function to use is completely depending on what the network is going to be used for (e.g. Cross Entropy for classification and Mean Square Error for reconstruction). A good way to find suitable objective functions is to look at how state of the art research, with similar model architectures and use, has defined their objective function and incorporate it. These objective functions can then tweaked with custom regularizing terms until the desired behaviour is achieved.

Step 5.b - Select an Optimizer

When compiling the model before training an important choice is what ***Optimizer*** to use. As mentioned in section 2.4.3 most optimizing techniques in ML are different gradient descent based operations and new ones are frequently published. As of now there are no consensus on what optimizer is the best. We do however recommend using *Adam* which offers an adaptive learning rate, momentum and a bias correction of the moment estimates. This makes Adam robust to the choice of hyperparameters and easy to work with, especially for someone not yet familiar with the tuning of hyperparameters. The Adam optimizers default learning rate of $\eta = 0.001$ works well in most cases. If it does not, some intuition about learning rate can be helpful. The ***Learning Rate*** is essentially the size of each step of gradient descent. A large step makes the training faster but is less accurate and increases the risk of overshooting the minima and the training may not converge. With a small step on the other hand, being slower and more accurate, may lead to the training being stuck in a badly placed local minima. If the training stagnates it could be either that the learning rate is to high or to low. Begin by lowering the learning rate and if that does not solve it, then try a higher learning rate.

Step 5.c - Train the Model

Train the model until the measured objective converges while keeping an eye on the validation loss.

There several ways to increase the ***Training Speed*** or making it less computational heavy. The *ReLU* activation function can speed up the training as well as decrease the amount of computing power required because of its simple derivative. Applying *Batch Normalization* (section 2.4.8) is another way to accelerate the training speed, by limiting variation in the layer input values. The choice of the *Batch Size* is also one thing to consider. The best results in accuracy as well as training speed are, as described in section 2.4.5, gotten by using the biggest batch size possible, depending on the memory of the hardware being used. For most networks the training is much faster on GPUs than CPUs, thanks to multi-processing, which makes GPU the obvious choice when training neural networks. Worth noting is that Tensor Processing Units (TPU) which are designed specifically for neural networks are currently being developed and made available through cloud services, such as Google Cloud [7].

When working with very deep neural networks there can be problems with degradation of the training accuracy and that it demands huge computing power in order to train. As described in section 2.6.2, this can be solved by adding Shortcut Connections throughout the network. Another important thing to consider is what kind of ***Activation Function*** to use. The choice of activation function depends on what the network is meant to do. The Sigmoid and Tanh functions are commonly used and works well for predictive tasks, with the main difference that Tanh has stronger gradients than Sigmoid and has the range $(-1, 1)$ instead of $(0, 1)$. As mentioned in section 2.4.2, they do both have problem with vanishing gradients, causing the network to stop learning or learning really slow. To solve this problem, use the *ReLU* activation on all layers except the last one. Sometimes *ReLU* has a problem with dead neurons during training causing some parts of the network to become permanently inactive (dead). If this happens it can be solved by replacing the *ReLU* with *LeakyReLU*. If the network needs to be able to classify multiple classes the last layer of the network should have a Softmax function instead of Sigmoid or Tanh.

6.2.6 Step 6 - Evaluate Model

After the model has converged in its training it is time to test it to see how well it approximates the desired function. The model can also be evaluated continuously while training or in between training epochs but with the cost of slowing down the training procedure.

Step 6.a - Evaluate Model on Test Data

To properly evaluate the model and see how well it has generalized to new data a test dataset is used. It is important that the test data is not part of the training data to give a fair measure of how well the model is actually performing.

If the model performs good on the training data but poorly on the test data (while coming from the same original dataset of similar data) the model is most likely overfit.

[7] https://cloud.google.com/tpu/

Overfitting can be reduced in multiple ways. Since overfitting usually occurs when the network has learned unwanted features or noise in the data it could be that the *Network is to Advanced* for its purpose or that the *Dataset is to Small*. The simplest way of minimizing the risk of overfitting is probably to apply *Early Stopping*, meaning that the training is terminated when the validation error no longer improves rather than the convergence of the training error. Since early stopping might not bring the loss down to a wanted value it is common to add some sort of *Regularization* to the network, see section 2.4.6. Dropout is an effective and easily implemented regularizing method. Dropout makes each neuron less dependant of the values from individual neurons and instead forces inputs from several different neurons to be used. Regularization can also be applied by parameter penalization, such as L^1 or L^2 regularization. L^1 regularization forces small weights to 0, making the model less advanced and gives a sparse solution. L^2 regularization forces all weights to small values, giving all weights similar strength, which results in a similar effect as Dropout where a neuron is less dependant on inputs from specific neurons. Another thing to consider is to use a larger dataset, which means more information and less room for overfitting. One can also add small perturbations to the input data by applying a small amount of Gaussian noise to the input which forces the network to find a more general representation.

Step 6.b - Fine Tune Hyperparameters

If the model is performing well on both training and evaluation data but not fully as good as hoped for you can try to fine tune the hyperparameters. One thing to try is to continue training but with a lower learning rate or a different optimizer, if the model still does not improve you can try adjusting other parameters such as the layer size or how much regularizing terms are weighted in the objective function. Some optimization techniques can be applied to find the best configuration of the hyperparameters but it is still under research how to do it properly and effectively without being too computationally expensive (Murugan, 2017). If fine tuning of the hyperparameters prove no luck or if the hyperparameters already have been optimized your final option is to change the architecture, which layers are used as well as the depth of the network. It may also be worth looking into other deep learning methods entirely.

When fine tuning the hyperparameters or the network architecture it is common practise to use a validation dataset. The valuation dataset is like the test dataset split from the same original dataset as the training set. This way the model is trained by the training data, the architecture is fine tuned unbiased by the training on the validation set and the model can finally be tested unbiased by the fine tuning on the test set.

6.2.7 Step 7 - Deploy Model

After the model has been trained and tuned and is working as expected the final step is to deploy the model for use in a target application.

Step 7.a - Export Trained Model

Export the model with its trained parameters according to the documentation of the used machine learning library. For models built in Keras models with weights can be exported in HDF5 format and in TensorFlow as a (checkpoint) .ckpt file.

Step 7.b - Deploy

Deploy the model by incorporating it in a shell application or parent system. How this is done in practise depends on what programming language and machine learning library have been used and is typically well documented. Exported TensorFlow models can be deployed to a web application using Tensorflow.js [8], same goes for Keras models but with Keras.js [9].

[8] https://js.tensorflow.org/
[9] https://transcranial.github.io/keras-js/

Chapter 7

Discussion

In this thesis we have explored how artificial intelligence, specifically in the field of deep learning, can be incorporated into the product design process to make tasks faster, simpler or more effective. The aim was that this exploration would lead to general guidelines and allow us to develop a framework to inspire, guide and help companies interested in taking a first step in using artificial intelligence to develop new products.

7.1 Method

The work process began with a very wide scope to get familiar with the field of deep learning. When sufficient knowledge had been gained, by reading different papers and research findings, the scope was narrowed to focus on the most relevant areas, presented in section 2.6. Since artificial intelligence and deep learning are fields of research yet to reach maturity, new findings are published frequently and it is difficult to intuitively form an opinion on what network types are best suited for certain tasks without initially having a wider scope. However, a narrow initial scope might have benefited the creation of the prototypes by enabling us to spend more time on it. One could also argue that a wide initial scope, which shows a bigger picture, could later help generalizing the implementation process and find challenges and difficulties not encountered while building the prototype.

7.1.1 The Choice of Machine Learning Library

We have throughout the implementation phase been using Keras and TensorFlow to build and train neural networks in Python. The choice of using these libraries over others was established early based on popularity, community sizes and for being open source. They are also very well documented with several examples and tutorials, which have been a great resource along the way. We are still very happy with this choice but believe that it would have been possible to land in similar results using any other machine learning library or framework.

7.1.2 Sources of Information

The main sources of information has been research papers on different subjects within machine learning. Initially we read books written on the subject but as we got familiar with the fundamentals and read how many of the current state of the art techniques work, we switched to instead reading research papers written in the last

couple of years. Deep learning is a field which in the recent years had some major breakthroughs and since it takes some time to publish a book we consider research papers a better source of recent and relevant information. ArXiv [1] publish most of the recent findings about machine learning and has been a great asset in our work, among with proceedings of the international conference on machine learning from the last years. We believe these references are reliable sources as most papers are written by prominent research groups and researchers within the field of deep learning and have been peer reviewed.

7.1.3 The Concept Development Phase

The method used to generate concepts of potential AI and machine learning systems for the PDP was Morphological Analysis, brainstorming and discussions around recent results presented in research papers. We find this method to have been sufficient for our needs in this context as the objective was not to find the optimal implementation, nor all possible solutions, but rather to find examples as inspiration and to find a concept reasonable for us to build a prototype of within the scope of this thesis. We found Morphological Analysis to be a good method for reasoning about concepts in a bigger picture and how different concepts might be able to interplay, and use that to come up with other potential concepts. We also believe that the Morphological Matrix can be a great tool in forming more holistic solutions, automating several steps of the PDP at once, but it is something we just discussed briefly and leave for future studies.

7.2 Result

7.2.1 The Concept Development Phase

The concepts presented in the concept development phase are quite broad and hypothetical but are all rooted in current research and based on the material covered in the theory study phase. As the focus in the theory phase was primarily deep learning and generative models the concepts presented are primarily based on these techniques. The concepts presented are also very general in the sense that they could be applied to a variety of products and tasks. This is due to the fact that we did not have a specific PDP or product in mind when doing the concept development phase but desired to keep it general such that it could be applicable and relatable by as many readers as possible. We believe that more precise concepts could be formulated if the target PDP and product class are known from the beginning as each type of product come with its own challenges. But this is left for future work and for anyone using the suggested framework described in chapter 6.

7.2.2 Auto Gathering of Image Data

The system for auto gathering of image data, described in section 5.1, followed by some manual cherry picking, resulted in a dataset with 4564 images of windows. About 1/4 of the images provided by the system ended up in the dataset while 3/4 was discarded during the manual selection. The reason for doing the manual selection at all was with the objective to improve our chances of training a generative model by limiting the data to the desired perspective (front view) and without distracting objects occluding the window.

[1] https://arxiv.org/

The primary bottleneck of the web scraper is the download speed of the images. It took us five hours with a $10Mbit/s$ download speed to run the scraper so with a faster internet connection this time could be drastically shortened. As the preprocessing step took ten minutes and the object detection and cropping procedure 30 minutes the total data gathering speed could have been decreased by 80% to about 60 minutes if say a $100Mbit/s$ connection had been used.

If we make an estimation that it takes about 90 seconds to manually download and crop five images, it would take about 23 hours to manually create a dataset of the same size. However, that does not include data cleaning and preprocessing for the network, a task our system can do automatically. Our system remove files that are undesirably small, corrupt and of undesired file type, it also shrink images considered too large and removes any duplicates from the set, all of this very quickly. Doing the same task manually would be very tedious and take a very long time to do.

There are several factors which could have decreased the amount of images that had to be discarded through the manual selection process. The primary factor is the labeled dataset the object detection model used to detect, crop and save windows from pictures was trained. The labeled dataset contained a very broad set of windows from various angles and partly occluded by objects. If this model had instead been trained only on images of windows like the ones we later cherry picked, it might have been more restrictive and ignored windows in undesired perspective or positioning. It would have been possible to train a second object detection model to parse the first collected dataset according to a few manually cherry picked examples to automate this task as well. However, due to time constrains and no guarantee that it would work we decided to just do the final selection manually instead. Another factor that played a big role is the type of object recognition model that were used. Of the three we tested we chose to work with the one that was fastest to train and detected the most windows on our test images. This despite detecting some false positives. By selecting another more accurate object detection model (like *ssd_mobilenet_v*2 or $faster\_rcnn\_nas\_lowproposals$) we might have ended up with more accurate detections. It is also possible that the number of vague detections of windows, especially in crude perspectives could have been reduced if we had used a higher threshold (>60%) for the minimal inference probability when deciding which detections to save.

There are also models available in the TensorFlow Object Detection Zoo that returns a masks of the detected object and not just a bounding box. By using a masking model it would be possible to remove the background of the images and only keep the desired object, and by that obtain a much cleaner dataset without varying background. This was considered but as they require much more elaborated prelabeled data compared to the bounding box models we decided to not use them. But it is something that could be worth looking into in the future.

7.2.3 Prototype

Using the windows dataset created with the data gathering system we explored generative deep learning methods in an attempt to generate design variations of windows. Based on the theory study we identified Variational Auto Encoders (VAEs) and Generative Adversarial Networks (GANs) to be two of the most promising methods to do so. We implemented three different VAE architectures, each improving upon the previous, and one GAN architecture, using Keras with TensorFlow as the backend. Preliminary discussion of these, intermediate results and how one led

to the other can be found under Prototypes (section 5.2) in the Concept Implementation, chapter 5.

The goal of these implementations were to create a prototype of the selected concept Abstract Design Combination, which utilizes latent space interpolation to mix different traits of known products to produce new designs. While we did not manage to interpolate between abstractions such as style and function (like in the concept description), we did manage to interpolate between existing windows and different mullion designs.

We still believe it would be possible to interpolate between different styles and functions as well but the challenge is to identify the corresponding transformations in the latent space to do so. One technique to do so is to use averaged encoding of known samples to identify the centers of respective style or function clusters in the latent space and then use vector arithmetic to extract and reapply that feature on something else, as we show with mullions in figure 5.30; Or interpolate freely between these pin-pointed classes. We only tried this with mullions as they are easy to manually filter from the existing dataset, pin-pointing a style class is more vague and would be easier to do using the automatic data gathering system on targeted search terms. Due to time constraints and for not directly contributing to the creation of the conceptual framework we leave this for further studies.

Besides interpolating between traits we also explored how to use the models to generate new design suggestions arbitrarily from noise. The Improved Convolutional VAE, built in section 5.2.5, show great potential on this by providing a diverse set of designs which, while looking like the training data, has some unique variations to them. While the quality and size of these images are not perfect we believe they still could be used as a source of inspiration when designing products. It is also not unfeasible that an improved system, with the ability to generate high resolution images, could be put in the hands of a customer to select a unique desired design as a template; which a designer and/or engineer then can realize into a true user customized product. The Wasserstein GAN, which was implemented in section 5.2.6, continues on the same line of thought with the goal of generating larger and much sharper images. This was done successfully and the resulting images also show a very interesting variation of designs, which stand out from the training dataset with more asymmetric and organic features. The model was still improving when we stopped it after 90 hours of training so it is possible that more realistic looking designs could have been generated if the model was left to train for longer. It is also possible that better results could have been achieved from both the VAE and W-GAN if a larger dataset had been used, to reduce overfitting and to better approximate the existing design distribution.

It is also possible to combine the strengths from both the VAE and GAN networks to build an Adversarial Auto Encoder (see section 2.6.13) with or without class conditioning to possibly reach even better results.

Limitations

The creation of the prototype was greatly limited by both available computing power and time for the training procedure. Had these two factors not been an issue the prototype could have been made more complex with either deeper architecture or more neurons in the layers allowing more complex functions to be approximated by the network. It is difficult to know in advance how changes in the hyperparameters will effect the training process and the resulting output. After x epochs the effects of the changes becomes clearer, however with our limited resources these epochs takes

considerable time to run and hence it takes time to try different configurations. A computer dedicated and built for deep learning would save a lot of time and allow for faster iterations. Another limitation is our previous knowledge in Python programming. Neither of us have had much experience with Python but since it is supported by many good resources for artificial intelligence and deep learning implementation, such as Keras (section 3.3.1) and TensorFlow (section 3.3.1), we decided to use Python. Fortunately it is a common and popular programming language with many great tutorials and resources to learn from. It is worth noting how the accessibility to these kind of resources has increased and how open the research in the field of artificial intelligence is (many publish their code used in research projects, free and open source, on GitHub [2]. Thanks to this openness and accessibility it is nowadays possible for anyone, even non experts, to implement artificial intelligence and build their own deep neural networks to solve specific problems.

7.2.4 Framework

The general implementation process (section 6.2) shows the main steps required in order to implement DL to the PDP. Though when using reinforcement learning to train a network the process is different than with supervised or unsupervised learning. Then the environment setup and reward function could be one of the biggest challenges, compared to the challenge of getting clean and good training data to the other learning methods. The general process we present is based on how we implemented our prototypes (section 5.2) along with how the implementation has been described in the different research papers referenced to throughout the theory study (chapter 2). Hence our general process is more directed towards the supervised and unsupervised learning methods and does not take reinforcement learning into account as much. Connected to the implementation process we have covered a series of challenges, which commonly occur when getting started with deep learning, along with solutions and strategies on how to overcome the difficulties. The challenges presented are directed towards supervised and unsupervised training but are not restricted to specific network architectures. The solutions and strategies should give the basic intuition on deep learning and neural networks needed to start building neural networks and implementing it to a custom task.

One could argue that we have not covered or discovered near to all implementation possibilities. Surely this is true, however the morphological matrix (table 4.1), where the concepts are connected to certain parts of the product design process, is not made to be a complete collection of the best possible ways to implement deep learning. The number of possibilities keeps rising along with the development of new techniques and methods of building and training neural networks. Due to this the matrix is made to be used by people interested in incorporating deep learning in their own design processes; either as a starting point to choose applications from and add new implementations to, or as inspiration of what can be done with artificial intelligence. Thus, we believe the concepts fill their purpose. The limitations in time and availability to computing power in this thesis work prohibited us from validating all of the concepts. They do however, build on previously done research with similar output results as those presented in the concept descriptions, with the main difference in what data the network is trained on. With that in mind it is reasonable to assume the concepts are realizable and possible to implement in the product design process.

[2] https://github.com/

The framework build on research published in the last couple of years but since the research is progressing so rapidly there will surely be several new architectures with their own difficulties and solutions published in the years to come. Despite that, our belief is that this framework will be a big help for someone getting started with deep learning and facilitate in their understanding of research being published in the future.

7.3 The work in a wider context

Since this thesis aims to support and inspire the use of artificial intelligence it is worth considering the possible long term effects. The automation provided by artificial intelligence, powered by deep learning, could indeed lead to the discontinuation of several common professions. This is something that goes hand in hand with technical advancements. Similar to the assembly line which made Henry Ford extremely successful in the car industry due to automation and innovative methods. Surely this was causing people to dread a huge unemployment issue, but which in the end led to a redistribution of professions and not to a decrease in the number of available jobs. Though the transitioning might not have been as smooth as it could have. Perhaps policies and institutions of a country could provide programs to help with job transitions, to keep high levels of employment and productivity despite increased automation. Since the effects of artificial intelligence on different professions is uncertain, the transition requires a high level of flexibility and preparation for different outcome scenarios in order to be smooth. In the near future we are likely to experience the automation of many tasks and it is important to reflect on the effects of it and how it affect people in order to support the transition.

7.3.1 Future Work

Since deep learning is such a rapidly growing field, with multiple approaches to solve different problems, there are many areas which we did not have time or opportunity to explore. There will always be new ways to incorporate artificial intelligence into product design as long as at least one of the two areas are advancing or changes in some other way. It is just most recently that some mesh based networks has emerged on the subject of 3D recreation, see section 2.7.3. One of the primary setbacks is that the way we are currently representing neural networks requires that the shape of the input data is constant, and consistent. If we build a network to parse 64x64px images, that is what it can do. If we want to parse a 128x83px image we first have to rescale it to 64x64. This poses some problems for CAD and similar high level representations as the number of features and datapoints vary greatly from model to model and has no necessary correlation to the represented size of the model. Recent successes with mesh-based networks worked around this by using fixed resolution meshes, e.g. an input mesh with a fixed number of vertices. There are some types of networks that can handle varying input though, and that is recurrent neural networks, which instead of parsing all data at once, takes a stream of data, which could be of undefined length. This is how state of the art systems today are processing text and sound data. One could thus in theory feed a CAD representation to a neural network in chunks, feature by feature, while the network internally builds up a representation of what it "sees", perhaps with the help of reservoir computing. One could also try to implement other types of recurrent neural network structures such

as recursive neural tensor nets, which would then work feature by feature recursively. The solution is perhaps somewhere in between utilizing all of this and other techniques still not invented. With access to more computational power it would be interesting to focus more on generation of 3D objects. Reinforcement learning (section 2.5.3) is an exciting approach on how a neural network can learn and opens up new possibilities for what neural networks can be used for. In this thesis we have not explored this approach enough to be able to draw conclusions on common problems and solutions in the implementation process using reinforcement learning. It would however be worth investigating the potential of reinforcement learning on the product design process and how it can be combined with different networks. Because of the many rapid changes and development in the field of deep learning there will surely be several new interesting prospects of implementation, which together can identify costumer needs, design and optimize a product as well as plan and streamline the production and manufacturing process.

Chapter 8

Conclusions

In this thesis we have explored the possibility to implement deep learning in the product design process. We have developed a conceptual framework containing three parts: (1) A morphological matrix connecting implementation possibilities to respective parts of the design process, to be used for inspiration and as something to build on. (2) A generalized implementation process focusing on supervised and unsupervised learning methods. (3) We have also identified challenges and difficulties commonly faced when implementing deep learning along with solutions and strategies to overcome them. In the process of creating this framework we have explored generative deep learning models and shown how VAEs and GANs can be used to generate design variations of windows. Together with the framework we also present a system for auto gathering of image data [1], greatly reducing the time and effort required to create a custom training dataset. We also release a diverse dataset [2], including 4564 images of different types of windows from a front view perspective.

[1] https://github.com/AlexNilsson/autolabeler
[2] https://github.com/AlexNilsson/windows-dataset

Bibliography

Abadi, Martin et al. (2016). "TensorFlow: A system for large-scale machine learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

Achlioptas, Panos et al. (2017). "Learning Representations and Generative Models for 3D Point Clouds". URL: http://arxiv.org/abs/1707.02392.

Ajiboye, A R et al. (2015). "Evaluating the Effect of Dataset Size on Predictive Model Using Supervised Learning Technique". In: *International Journal of Software Engineering & Computer Sciences (IJSECS)* 1, pp. 75–84. DOI: 10.15282/ijsecs.1.2015.6.0006. URL: http://ijsecs.ump.edu.my/images/archive/vol1/06Ajiboye{_}IJSECS.pdf.

Arjovsky, Martin and Léon Bottou (2017). "Towards Principled Methods for Training Generative Adversarial Networks". URL: https://arxiv.org/pdf/1701.04862.pdf.

Arjovsky, Martin, Soumith Chintala, and Léon Bottou (2017). "Wasserstein GAN". URL: https://arxiv.org/pdf/1701.07875.pdf.

Autodesk (2018). *What Is Generative Design*. URL: https://www.autodesk.com/solutions/generative-design (visited on 03/15/2018).

Bojanowski, Piotr et al. (2017). "Optimizing the Latent Space of Generative Networks". URL: http://arxiv.org/abs/1707.05776.

Brock, Andrew et al. (2016). "Generative and Discriminative Voxel Modeling with Convolutional Neural Networks". In: *arXiv_org_e-Print_Archive*. DOI: 10.5281/ZENODO.802261. arXiv: 1608.04236. URL: http://arxiv.org/abs/1608.04236.

Brown, Larry (2015). *Deep Learning with GPUs*. URL: http://www.nvidia.com/content/events/geoInt2015/LBrown{_}DL.pdf.

Caruana, Rich, Steve Lawrence, and Abstract Lee Giles (2001). "Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping". In: *Advances in Neural Information Processing Systems 13 (NIPS 2000)*. Ed. by T. K. Leen Leen, T. G. Dietterich, and V. Tresp. MIT Press, pp. 402–408. URL: https://papers.nips.cc/paper/1895-overfitting-in-neural-nets-backpropagation-conjugate-gradient-and-early-stopping.pdf.

Cho and KyungHyun (2013). "Simple sparsification improves sparse denoising autoencoders in denoising highly noisy images". In: *ICML'13 Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*. Ed. by Sanjoy Dasgupta and David McAllester. Atlanta: JMLR.org, pp. III–432. URL: https://dl.acm.org/citation.cfm?id=3042985.

Chollet, François and Others (2015). *Keras*. \url{https://keras.io}.

Cooper, Robert G. (1990). "Stage-gate systems: A new tool for managing new products". In: *Business Horizons* 33.3, pp. 44–54. ISSN: 0007-6813. DOI: 10.1016/0007-6813(90)90040-I. URL: https://www-sciencedirect-com.e.bibl.liu.se/science/article/pii/000768139090040I.

Cybenkot, G (1989). "Mathematics of Control, Signals, and Systems Approximation by Superpositions of a Sigmoidal Function*". In: *Math. Control Signals Systems* 2, pp. 303–314. URL: http://bit.ly/2JLHzy1.

Danon, Bill (2018). *How GM and Autodesk are using generative design for vehicles of the future*. URL: http://blogs.autodesk.com/inthefold/how-gm-and-autodesk-use-generative-design-for-vehicles-of-the-future/ (visited on 05/22/2018).

Dauphin, Yann N et al. (2014). "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization". URL: https://arxiv.org/pdf/1406.2572.pdf.

Dosovitskiy, Alexey and Thomas Brox (2016). "Generating Images with Perceptual Similarity Metrics based on Deep Networks". In: arXiv: 1602.02644. URL: http://arxiv.org/abs/1602.02644.

Dozat, Timothy (2015). *Incorporating Nesterov Momentum into Adam*. URL: http://cs229.stanford.edu/proj2015/054{_}report.pdf.

Dumoulin, Vincent, Francesco Visin, and George E P Box (2018). "A guide to convolution arithmetic for deep learning". In: arXiv: arXiv:1603.07285v2. URL: https://arxiv.org/pdf/1603.07285.pdf.

Ferrucci, David et al. (2010). "Building Watson: An Overview of the DeepQA Project". In: *AI Magazine* 31.3, p. 59. ISSN: 0738-4602. DOI: 10.1609/aimag.v31i3.2303. URL: https://aaai.org/ojs/index.php/aimagazine/article/view/2303.

Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge (2016). "Image Style Transfer Using Convolutional Neural Networks". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, pp. 2414–2423. ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.265. URL: http://ieeexplore.ieee.org/document/7780634/.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep learning*. MIT Press, p. 775. ISBN: 0262035618. URL: http://www.deeplearningbook.org.

Goodfellow, Ian J. et al. (2014). "Generative Adversarial Networks". In: *arXiv_org_e-Print_Archive*, p. 9. arXiv: 1406.2661. URL: http://arxiv.org/abs/1406.2661.

Gu, Jiuxiang et al. (2017). "Recent Advances in Convolutional Neural Networks". URL: https://arxiv.org/pdf/1512.07108.pdf.

Gulrajani, Ishaan et al. (2017). "Improved Training of Wasserstein GANs Montreal Institute for Learning Algorithms". URL: https://arxiv.org/pdf/1704.00028.pdf.

Gupta, Dishashree (2017). *Transfer learning & The art of using Pre-trained Models in Deep Learning*. URL: https://www.analyticsvidhya.com/blog/2017/06/transfer-learning-the-art-of-fine-tuning-a-pre-trained-model/.

Ha, David and Douglas Eck (2017). "A Neural Representation of Sketch Drawings". URL: http://arxiv.org/abs/1704.03477.

He, Kaiming et al. (2015a). "Deep Residual Learning for Image Recognition". In: *arXiv_org_e-Print_Archive*, p. 12. arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

— (2015b). "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *2015 IEEE International Conference on Computer Vision (ICCV)*. IEEE, pp. 1026–1034. ISBN: 978-1-4673-8391-2. DOI: 10.1109/ICCV.2015.123. URL: http://ieeexplore.ieee.org/document/7410480/.

— (2016). "Identity Mappings in Deep Residual Networks". In: *arXiv.org e-Print Archive*, p. 15. arXiv: 1603.05027. URL: http://arxiv.org/abs/1603.05027.

Hinton, Geoffrey, Ni@sh Srivastava, and Kevin Swersky (2014). *Neural Networks for Machine Learning Lecture 6a Overview of mini--batch gradient descent*. Toronto. URL:

http://www.cs.toronto.edu/{~}tijmen/csc321/slides/lecture{_}slides{_}lec6.pdf.

Hinton, Geoffrey E. et al. (2012). "Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv_org_e-Print_Archive*. arXiv: 1207.0580. URL: http://arxiv.org/abs/1207.0580.

Huang, Jonathan et al. (2016). "Speed/accuracy trade-offs for modern convolutional object detectors". In: arXiv: 1611.10012. URL: http://arxiv.org/abs/1611.10012.

Ioffe, Sergey and Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32th International Conference on Machine Learning*. Ed. by David Blei Francis Bach. Lille, France: PMLR, pp. 448–456. URL: http://proceedings.mlr.press/v37/ioffe15.pdf.

Karras, Tero et al. (2017). "Progressive Growing of GANs for Improved Quality, Stability, and Variation". URL: http://arxiv.org/abs/1710.10196.

Kato, Hiroharu, Yoshitaka Ushiku, and Tatsuya Harada (2017). "Neural 3D Mesh Renderer". URL: http://arxiv.org/abs/1711.07566.

Kingma, Diederik P. and Jimmy Ba (2015). "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations (ICLR 2015)*. San Diego: Ithaca, NY: arXiv.org. arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

Kingma, Diederik P and Max Welling (2013). "Auto-Encoding Variational Bayes". In: arXiv: 1312.6114. URL: http://arxiv.org/abs/1312.6114.

Kullback, S and R A Leibler (1951). *On Information and Sufficiency*. 1st ed. Vol. 22, pp. 79–86. URL: http://www.jstor.org/stable/2236703.

LeCun, Y. et al. (1998a). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE*. Vol. 86. 11, pp. 2278–2324. DOI: 10.1109/5.726791. URL: http://ieeexplore.ieee.org/document/726791/.

LeCun, Yann et al. (1998b). "Efficient BackProp". In: *Neural Networks: Tricks of the Trade, this book is an outgrowth of a 1996 NIPS workshop*. London, UK, pp. 9–50. URL: http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf.

Li, Hongxing, C.L. Philip Chen, and Han-Pang Huang (2000). *Fuzzy Neural Intelligent Systems: Mathematical Foundation and the Applications in Engineering*. CRC Press, pp. 255–265. ISBN: 9780849323607.

Li, Lisha et al. (2017). "Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits". In: *5th International Conference on Learning Representations*. URL: http://bit.ly/2JKjJCR.

Lin, Chen-Hsuan, Chen Kong, and Simon Lucey (2017). "Learning Efficient Point Cloud Generation for Dense 3D Object Reconstruction". URL: http://arxiv.org/abs/1706.07036.

Liu, Chen (2017). "International Competitiveness and the Fourth Industrial Revolution". In: *Entrepreneurial Business and Economics Review* 5.4, pp. 111–133. ISSN: 2353883X. DOI: 10.15678/EBER.2017.050405. URL: https://eber.uek.krakow.pl/index.php/eber/article/view/307.

Maas, Andrew L, Awni Y Hannun, and Andrew Y Ng (2013). "Rectifier Nonlinearities Improve Neural Network Acoustic Models". In: *ICML Workshop on Deep Learning for Audio, Speech, and Language Processing (WDLASL 2013)*.

Makhzani, Alireza and Brendan Frey (2013). "k-Sparse Autoencoders". In: arXiv: 1312.5663. URL: http://arxiv.org/abs/1312.5663.

Makhzani, Alireza et al. (2015). "Adversarial Autoencoders". In: *arXiv.org e-Print Archive*. arXiv: 1511.05644. URL: http://arxiv.org/abs/1511.05644.

Mnih, Volodymyr et al. (2013). *Playing Atari with Deep Reinforcement Learning*. Tech. rep. arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

Murugan, Pushparaja (2017). "Hyperparameters Optimization in Deep Convolutional Neural Network / Bayesian Approach with Gaussian Process Prior". URL: http://arxiv.org/abs/1712.07233.

Nair, Vinod and Geoffrey E Hinton (2010). "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *ICML 2010 - Proceedings, 27th International Conference on Machine Learning*, pp. 807–814. URL: http://www.cs.utoronto.ca/{~}hinton/absps/reluICML.pdf.

Nervous System (2014). *Floraform sculptures*. URL: https://n-e-r-v-o-u-s.com/projects/albums/floraform-sculptures/ (visited on 03/15/2018).

Ovesen, Nis (2012). *The Challenges of Becoming Agile: Implementing and Conducting Scrum in Integrated Product Development*. URL: http://vbn.aau.dk.e.bibl.liu.se/da/publications/the-challenges-of-becoming-agile(80b45789-6f75-45d2-9697-ecc750439f21).html{\#}?.

Oxford Dictionaries (2018). *Definition of artificial intelligence in English by Oxford Dictionaries*. URL: https://en.oxforddictionaries.com/definition/artificial{_}intelligence (visited on 03/15/2018).

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio (2012). "On the difficulty of training Recurrent Neural Networks". In: arXiv: 1211.5063. URL: http://arxiv.org/abs/1211.5063.

Radford, Alec, Luke Metz, and Soumith Chintala (2015). "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks". In: *arXiv_org_e-Print_Archive*. arXiv: 1511.06434. URL: http://arxiv.org/abs/1511.06434.

Radiuk, Pavlo M. (2017). "Impact of Training Set Batch Size on the Performance of Convolutional Neural Networks for Diverse Datasets". In: *Information Technology and Management Science* 20.1. ISSN: 2255-9094. DOI: 10.1515/itms-2017-0003. URL: http://content.sciendo.com/view/journals/itms/20/1/article-p20.xml.

Ren, Shaoqing et al. (2017). "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.6, pp. 1137–1149. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2016.2577031. URL: http://ieeexplore.ieee.org/document/7485869/.

Rezende, Danilo Jimenez, Shakir Mohamed, and Daan Wierstra (2014). "Stochastic Backpropagation and Approximate Inference in Deep Generative Models". In: arXiv: 1401.4082. URL: http://arxiv.org/abs/1401.4082.

Rojas, Raúl (1996). "The Backpropagation Algorithm". In: *Neural Networks*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 149–182. DOI: 10.1007/978-3-642-61068-4_7. URL: http://link.springer.com/10.1007/978-3-642-61068-4{_}7.

Ruder, Sebastian (2017). "An overview of gradient descent optimization algorithms *". Dublin. URL: https://arxiv.org/pdf/1609.04747.pdf.

Rumelhart, David E., James L. McClelland, and San Diego. PDP Research Group. University of California (1986). *Parallel distributed processing : explorations in the microstructure of cognition*. MIT Press, pp. 318–362. ISBN: 026268053X. URL: https://dl.acm.org/citation.cfm?id=104293{\&}preflayout=flat.

Sak, Haşim et al. (2015). *Google voice search: faster and more accurate*. URL: https://ai.googleblog.com/2015/09/google-voice-search-faster-and-more.html.

Sandler, Mark et al. (2018). "MobileNetV2: Inverted Residuals and Linear Bottlenecks". In: *CoRR* abs/1801.0. arXiv: 1801.04381. URL: https://arxiv.org/pdf/1801.04381.pdf.

Schmitt, Simon et al. (2018). "Kickstarting Deep Reinforcement Learning". URL: http://arxiv.org/abs/1803.03835.

Shang, Wenling, Kihyuk Sohn, and Yuandong Tian (2017). "Channel-Recurrent Autoencoding for Image Modeling". URL: http://arxiv.org/abs/1706.03729.

Sharma, Abhishek, Oliver Grau, and Mario Fritz (2016). "VConv-DAE: Deep Volumetric Shape Learning Without Object Labels". In: *arXiv_org_e-Print_Archive*. arXiv: 1604.03755. URL: http://arxiv.org/abs/1604.03755.

Silver, David et al. (2017). "Mastering the game of Go without human knowledge." In: *Nature* 550.7676, pp. 354–359. ISSN: 1476-4687. DOI: 10.1038/nature24270. URL: http://www.nature.com/doifinder/10.1038/nature24270http://www.ncbi.nlm.nih.gov/pubmed/29052630.

Springenberg, Jost Tobias et al. (2015). "Striving for Simplicity: The All Convolutional Net". Freiburg, Germany. URL: https://arxiv.org/pdf/1412.6806.pdf.

Sun, Chen et al. (2017). "Revisiting Unreasonable Effectiveness of Data in Deep Learning Era". In: *IEEE International Conference on Computer Vision (ICCV)*. URL: https://arxiv.org/pdf/1707.02968.pdf.

Szegedy, Christian et al. (2015). "Rethinking the Inception Architecture for Computer Vision". In: *2016 {IEEE} Conference on Computer Vision and Pattern Recognition*, pp. 2818–2826. arXiv: 1512.00567. URL: http://arxiv.org/abs/1512.00567.

Szegedy, Christian et al. (2016). "Rethinking the Inception Architecture for Computer Vision". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, pp. 2818–2826. ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.308. URL: http://ieeexplore.ieee.org/document/7780677/.

Tzutalin (2015). *LebelImg*. URL: https://github.com/tzutalin/labelImg.

Ulrich, Karl T. and Steven D. Eppinger (2012). *Product design and development*. McGraw-Hill/Irwin, p. 415. ISBN: 9780071086950. URL: http://www.ulrich-eppinger.net/.

Vincent, Pascal et al. (2008). "Extracting and composing robust features with denoising autoencoders". In: *Proceedings of the 25th international conference on Machine learning - ICML '08*. New York, New York, USA: ACM Press, pp. 1096–1103. ISBN: 9781605582054. DOI: 10.1145/1390156.1390294. URL: http://portal.acm.org/citation.cfm?doid=1390156.1390294.

Wan, Li et al. (2013). "Regularization of Neural Networks using DropConnect". In: *Proceedings of the 30th International Conference on Machine Learning*. Vol. 28, pp. 1058–1066. URL: http://proceedings.mlr.press/v28/wan13.html.

Wang, Nanyang et al. (2018). "Pixel2Mesh: Generating 3D Mesh Models from Single RGB Images". URL: http://arxiv.org/abs/1804.01654.

White, Tom (2016). "Sampling Generative Networks". In: *arXiv_org_e-Print_Archive*. arXiv: 1609.04468. URL: http://arxiv.org/abs/1609.04468.

Zhang, Yexun, Ya Zhang, and Wenbin Cai (2017). "Separating Style and Content for Generalized Style Transfer". URL: http://arxiv.org/abs/1711.06454.

Zhao, Bo et al. (2017). "Multi-View Image Generation from a Single-View". URL: http://arxiv.org/abs/1704.04886.

Zhu, Yuke et al. (2018). "Reinforcement and Imitation Learning for Diverse Visuomotor Skills". URL: http://arxiv.org/abs/1802.09564.

Zoph, Barret et al. (2017). "Learning Transferable Architectures for Scalable Image Recognition". URL: http://arxiv.org/abs/1707.07012.

Appendix A

Scrape Parameters

Search Terms
window
window white background
window frame white background
window styles
window style
window frame
window frame design
round window frame
wood window frame
chinese window frame white background
chinese window frame
gothic window frame white background
gothic window frame
isolated window
window mullion design
circular window
circular window design
japanese window
japan window
glass window

TABLE A.1: The strings to search for

Search Languages
en
ar
sv
de
es
el
fr
it
jw
ko
pl
tr
no
ru

TABLE A.2: Languages to which the search terms are translated

Appendix B

Latent Space Sampled Images

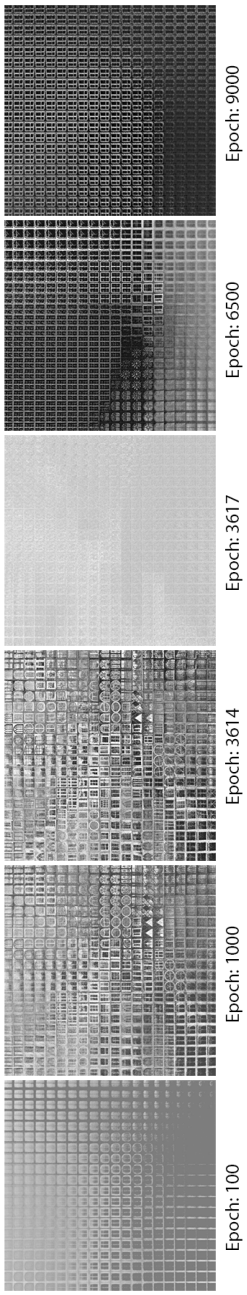

FIGURE B.1: Figure 5.16 in a larger format

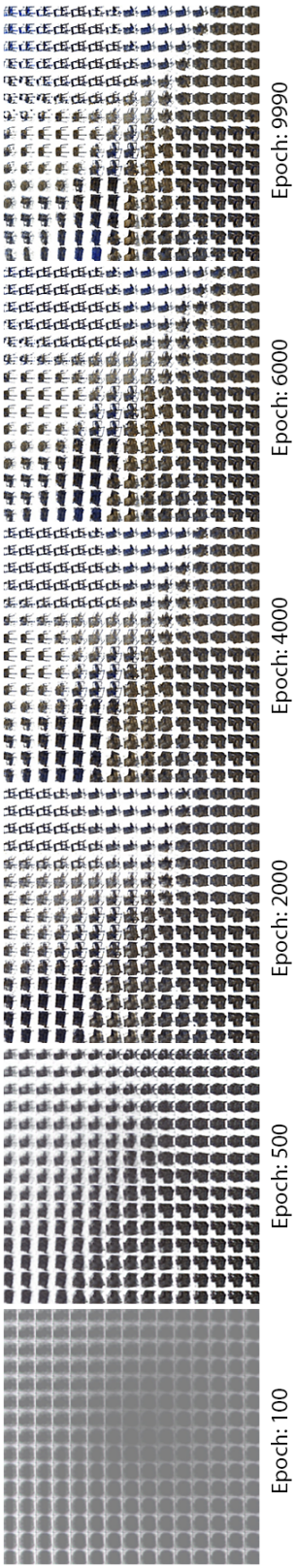

FIGURE B.2: Figure 5.18 in a larger format

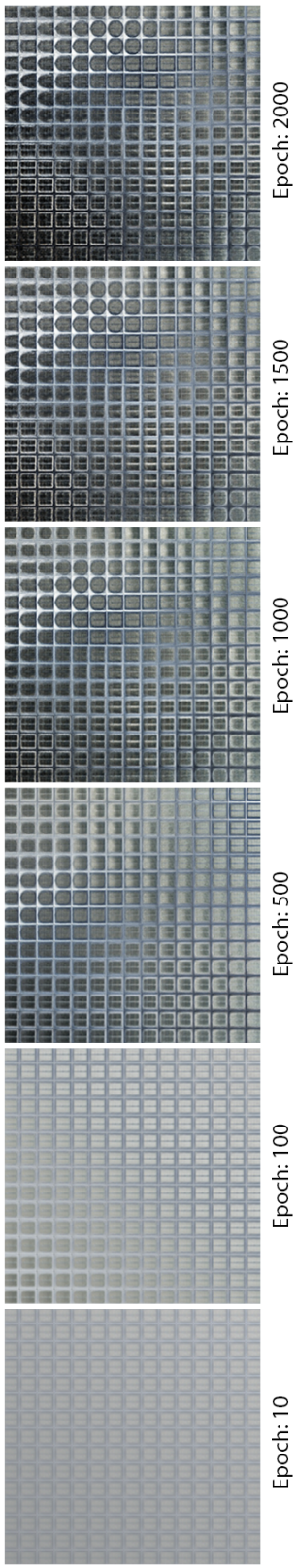

FIGURE B.3: Figure 5.20 in a larger format

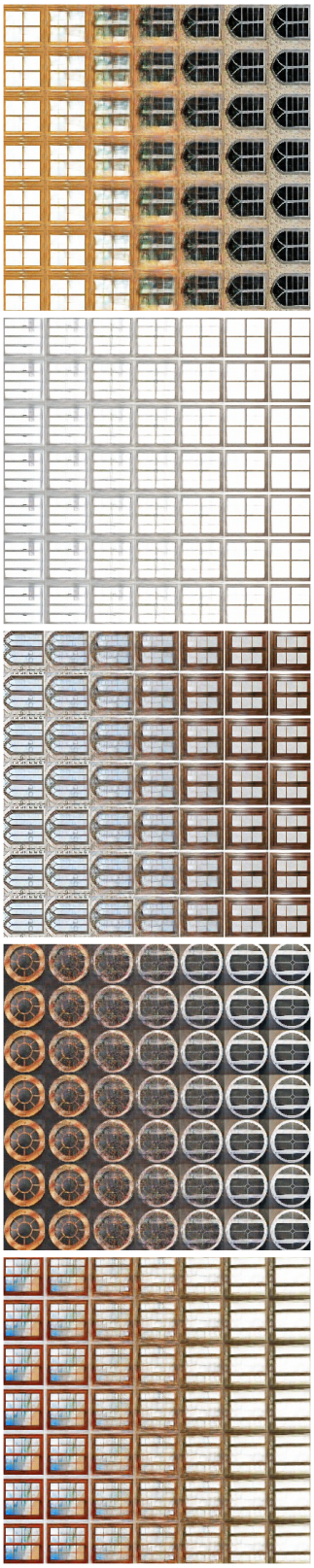

FIGURE B.4: Figure 5.28 in a larger format

FIGURE B.5: Figure 5.25 in a larger format

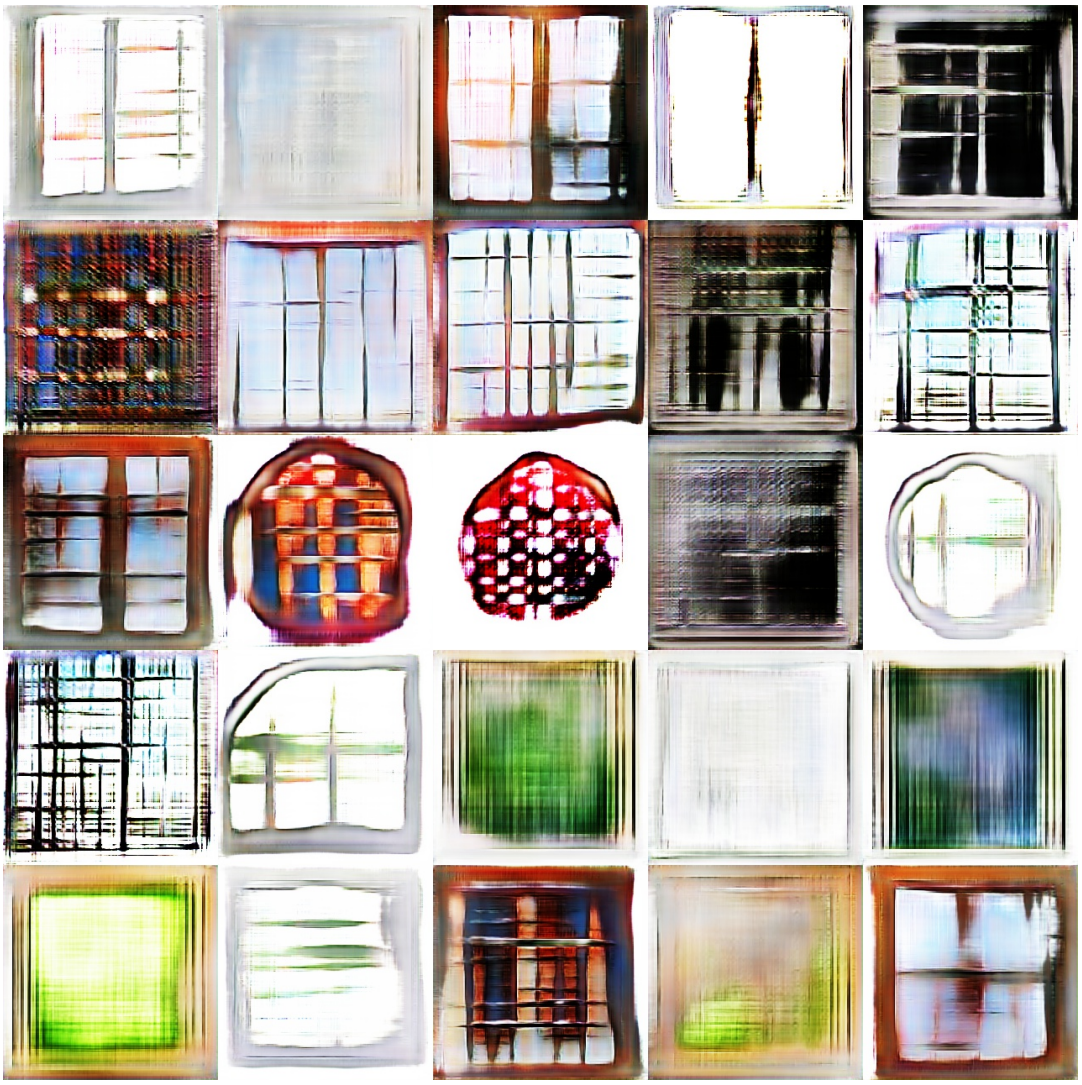

FIGURE B.6: 25 randomly sampled (256x256) images from the W-GAN model after 37800 epochs of training

FIGURE B.7: 25 randomly sampled (256x256) images from the W-GAN model after 37930 epochs of training

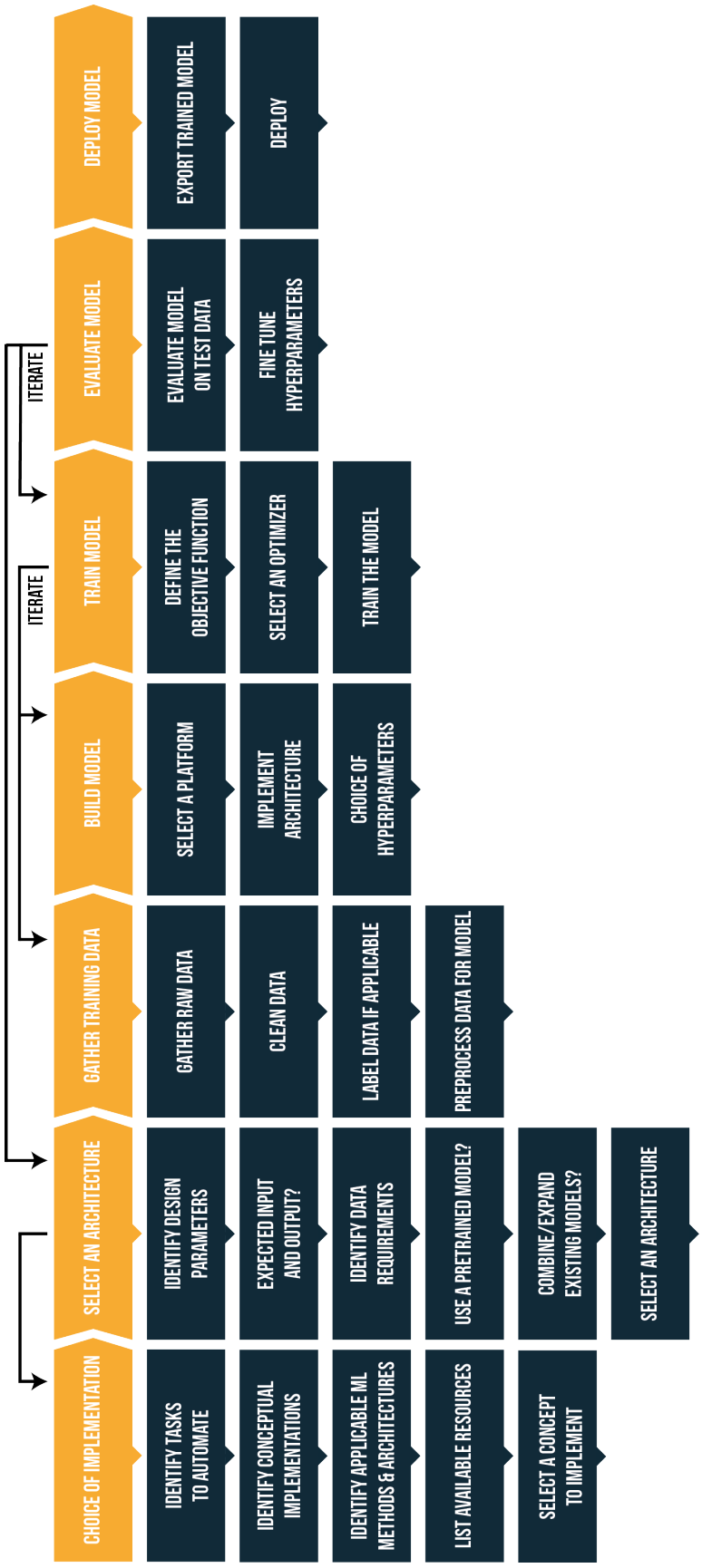

FIGURE B.8: Figure 6.1 in a larger format