Attempting to Regulate Image Generation in a Generative Adversarial Net using an External Ranker and Linear Interpolation

Alan Lau

MComp (hons) Computer Science
University of Bath
May 2017

Attempting to Regulate Image Generation in a Generative Adversarial Net using an External Ranker and Linear Interpolation

Submitted by: Alan Lau

COPYRIGHT

DECLARATION

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

Abstract

This project aims to investigate whether we can regulate the generative process of a Generative Adversarial Net (GAN). Various attempts have been taken to try and regulate the process. We aim to utilise a ranker and linear interpolation to achieve this

Our investigation has found that our method shows promise but requires more work. Demonstrating in a synthetic digit dataset, we confirm that linear interpolation in the latent space helps fill the gap required within images. However, on more complex datasets, futher investigation, such as the positioning of the ranker, shall provide better results.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Human has an incredible brain that is capable of collating different life experiences into memory, and can combine the intricate details between pieces of memories to create new, unseen content. Researchers in machine learning have long been attempting to replicate this through developing algorithms that can fully exploit these intricate details to generating new, realistic content.

Deep convolutional neural network have seen great success in supervised learning, especially in discriminative tasks such as classification, localisation and detection. Massive improvements have been achieved and demonstrated in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC).

Generating new images is an unsupervised learning problem. The objective is to extract useful hidden details in some given images and use them to create new datapoints. Well-known unsupervised learning algorithms such as variations of Boltzmann Machines (pp. 654-687 [1]), Gaussian Mixture Models (e.g. citegmm) and Gaussian Process [2] require intractable optimisation and approximation methods, which make them difficult to compute over image datasets which are of high dimensions. Hence, more efficient and effective methods utilising the structure of deep neural networks have been developed.

Utilising the benefits of deep convolutional neural networks, Generative Adversarial Nets (GAN) [3] is developed and has become a hot topic in generative models. LeCun, the major contributor to Convolutional Neural Networks (CNN), reckons that GAN is the next major development in generative models [1]. Although being able to generate new, near natural images [4], it is difficult to regulate how the generated image looks like. Little work is done comparing different deep learning generative models in their generative properties and the ability to regulate the outcome with some metrics.

[1] Answer by LeCun on Quora on the question "What are some recent and potentially upcoming breakthroughs in deep learning?": *link [Accessed 24 April 2017]*

1.1 Aim

We want to attempt to regulate the generation of images through a pairwise comparison ranker. We ask the question, "Is it possible to generate a desired image, say of a certain level of smiliness, rather than any new image randomly?" This type of criteria is subjective, which means it is hard for a user to quantify by inspection. We will investigate if a ranker, for instance, a pairwise comparison ranker, is suitable for our models.

Our goal is to develop a way for a user to generate a desired image given some subjective metrics of a bounded scale. For instance, a user might want to generate a 'more smiling' image or of 'a smiliness level of 0.5'. Together with a ranker, we attempt to investigate if a simple method such as linear interpolation enables this.

To limit the scope of the project, we will focus on two major generative algorithms, namely *autoencoder* and *Deep Convolutional Generative Adversarial Network (DCGAN)* [4]. These will be further detailed in Chapter 2.

1.2 Structure

We first discuss the related concepts and the development of deep learning in images applications in Chapter 2. We then describe the methods we have taken in Chapter 3 to reach our results. And subsequently, we discuss the results in detail in Chapter 4. Finally, we conclude the findings of this project in Chapter 5.

Chapter 2

Literature Review

Traditionally, supervised learning methods such as Support Vector Machine and Random Forest have taken the centre stage in machine learning. However, it requires meticulous feature engineering and manual fine-tuning in order to obtain good results.

When it comes to problems involving complex datasets such as computer vision or graphics-related problems, not only is it difficult to handle the high dimensionality and complexity of the raw dataset manually, but it is also challenging to produce an appropriate feature vector format that is effective in capturing useful and generalised information. Deep Neural Network, in particular convolutional neural network (CNN), has been shown to be far more effective than conventional supervised machine learning approaches in image applications in practise.

In this review, we explore the current state-of-the-art deep learning algorithms, focusing on those designed for image classification and generation. Through reviewing these notable implementations, we will reveal the properties that makes them useful, and explain the components to pursue the aims of this project.

2.1 Neural Network

Machine learning and pattern analysis have been a key research area in the past few decades. Traditionally, to make machine learning algorithms work, a lot of effort has to put into feature engineering, i.e. the process of transforming raw data, such as image pixels, into a suitable structure that can be used effectively to perform tasks such as object classification or object detection. My third-year individual project [5] demonstrated that despite a clear and logical approach to feature engineering, it may not be the optimal format that provides the highest quality of results. Neural networks aim to discover this representation purely through its structures and a set of methods. This is called representation learning.

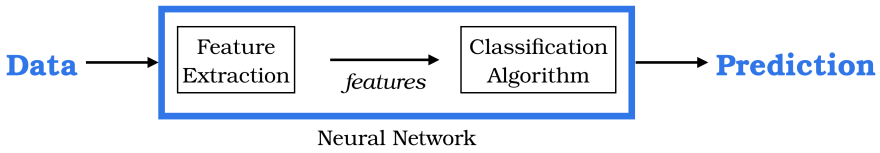

Figure 2.1: *A neural network works as a 'black box' to map input to output. It automatically performs feature extraction and classification internally.*

The key differentiator between traditional machine learning techniques and deep neural networks is that deep neural networks learns its features through some defined, general-purpose functions, rather than manually defined through labour-intensive feature engineering [6]. In our studied case, it can learn an adequate representation as feature extraction and classification are jointly performed.

2.1.1 Structure of Neural Network

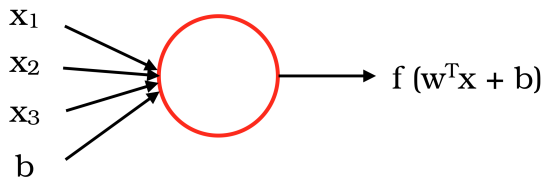

Figure 2.2: *A neuron with inputs x_i and bias b.*

The units that form a neural network is called neurons. Each neuron takes some input and maps it to the output using a non-linear mathematical function. In other words, the purpose of a neuron is to map an input to an output given the input value, weight and bias.

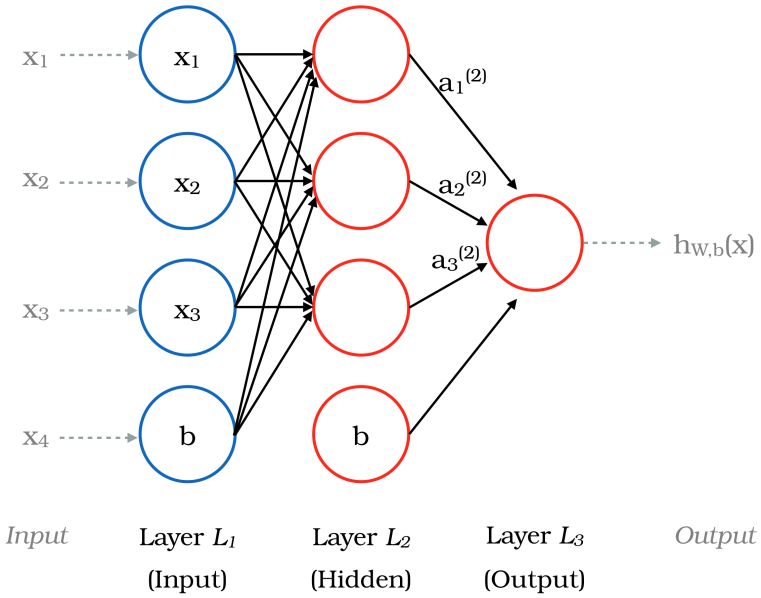

Figure 2.3: *This shows a simple neural network with three layers (of which one is hidden) using logistic regression and sigmoid activation as an example. We display the inputs as neurons so to demonstrate the various types of layers in the network. (Adapted from [7].)*

Figure 2.3 displays the common architecture of a feedforward neural network, where the first layer is an input layer and the last layer is an output layer. There can be as many neurons in each of these layers, including the output layer where we might expect multiple output options, and also as many hidden layers as required. Each hidden layer takes the output of the previous layer as an input and computes an output. In other words, there is no cycle in this type of networks, and information only flows from one end to another.

The three layers in Figure 2.3 represent each of the three types of layers in a common neural network:

- **Layer** L_1: This is the *input layer*. It takes the initial input and feed it into the network.
- **Layer** L_2: This is the *hidden layer*. Their output are produced by the neurons and are not usually directly observed. Each hidden layer projects into another space.
- **Layer** L_3: This is the *output layer*. It gives the final output of the network.

There can be ambiguity in how the network structure is referred to. Here, we define that this is a *1-layer network*, i.e. with respect to the number of *hidden* layers. However, when we refer to the mathematical elements of the network, we take into account the

input and output layers. For instance, the output of the hidden layer is $a_i^{(2)}$, denoting that it is from *layer* L_2, instead of *layer* L_1.

2.1.2 Training Phases

There are two steps in training a neural network. The first step is the forward pass, where the input is passed to produce an output using the initial parameters. The second is the backward pass, where a method called backpropagation is used to allow the results of the cost function to be passed backwards from output to input, so that new parameter values can be calculated.

2.1.2.1 Step 1: Forward Pass

As shown in Figure 2.2, each neuron takes some input values and produces an output. In a multi-layer neural network, the output of a neuron acts as the input for another, until the final output is created.

$$h_{w,b}(x) = f\left(w^T x + b\right) = f(z) \tag{2.1}$$

Mathematically, each neuron acts as a transformation unit that calculates an activation given some input. We take Figure 2.2 as an example. The neuron takes x_1, x_2 and x_3 as input, with associated weights W_i and a bias b for the whole input. We denote W as a vector containing all the weights associated to the input. This gives a hypothesis $h_{W,b}(x)$ that is used to fit to the input. The output is created by using a non-linear activation function, generally in the form $f : \mathbb{R} \mapsto \mathbb{R}$, to map from input to output, as shown in Equation (2.1) and visualised in Figure 2.2. We will discuss more about activation functions in Section 2.2.6.

The initialisation of the weights are crucial. If all weight is identical initially, all neuron will give the same output, which defeats the purpose of a neural network where it combines differing parts to obtain an output [7, 8]. In other words, 'symmetry breaking' is maintained by making each weight different. Although larger weights allow a greater symmetry breaking effect, this can cause the activation functions in neurons to saturate, making them permanently disabled [1].

We can expand this to cover the whole network shown in Figure 2.3. The neural network now has parameters $(W, b) = \left(W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}\right)$. We denote the association of weights between units of two adjacent layers by $W_{ij}^{(l)}$, i.e. the connection between neuron j in layer l and i in layer $l+1$. Note that this is in a 'reverse' order that we will be useful for Step 2 of the training process. Similarly for biases, we use b_i^l to represent the bias associated with unit i in layer l.

Hence, we can calculate the activation for each neuron in the network, denoted as $a_i^{(l)}$ to represent i-$^{\text{th}}$ neuron output at layer l:

$$a_1^{(2)} = f\left(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}\right) \tag{2.2}$$

$$a_2^{(2)} = f\left(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}\right) \tag{2.3}$$

$$a_3^{(2)} = f\left(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}\right) \tag{2.4}$$

$$h_{W,b}(x) = f\left(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)}\right) \tag{2.5}$$

In the same vein as we defined z in Equation (2.1), we can simply by defining $z^{(l)} = W^{(l-1)}a^{(l-1)} + b^{(l-1)}$, realising that for every subsequent layer in a feedforward layer, we calculate its activation by:

$$z^{(l+1)} = W^{(l)}a^{(l)} + b^{(l)} \tag{2.6}$$

$$a^{(l+1)} = f\left(z^{(l+1)}\right) \tag{2.7}$$

This generalisation enables networks with varying number of hidden layers and neurons in each hidden and output layer possible.

2.1.2.2 Step 2: Backward Pass

In order to update the parameters (weights and biases) of the network, we require a cost function, an optimisation algorithm and an effective method that extends the effect of the optimisation across the network. Let us first describe cost functions and the optimisation algorithms briefly:

- **Cost Function**
  A cost function is used to calculate the error of the output compared to the ground truth. There exists many loss functions for different purposes. Variations of *cross-entropy* are popular choices for classification applications [8], and *mean square error* (MSE) is used for regression problems. We aim to minimise this function through an optimisation algorithm.

- **Optimisation [9]**
  Variations of gradient descent is used to optimise the cost function in a deep neural network. Gradient descent mathematically guarantees that we find the direction of steepest descend, so that we can alter the weight accordingly [8]. *Mini-batch gradient descent (MGD)* is the general choice, as it combines the benefits offered by *batch gradient descent (BGD)* and *stochastic gradient descent (SGD)*.

  BGD computes the gradient of the cost function for the whole dataset at each parameter update, which takes a long time and requires all the data to be available in memory. On the other hand, *SGD* updates the parameters for each datapoint, making the cost function to fluctuate greatly as it moves towards convergence.

MGD combines the two by performing an update for each small batch, B, which can be computed efficiently when compared to *BGD* and a more stable convergence than *SGD*.

We usually employ an optimisation algorithm for the learning rate of gradient descent, as it is difficult to pick an optimal value that maximises the chance of a timely convergence. Used in notable networks such as deep convolutional generative adversarial network [4], Adam [10] is a recommended algorithm [8] that attempts to maximise the learning step size while keeping it stable [10].

A common setup setup is to use a mean-squared error (MSE) cost function, backpropagation to calculate gradients across the network, and mini-batch gradient descent with batch size of B as the optimisation method to learn from the calculated gradients so to optimise the cost function. Equation (2.8) demonstrates the MSE example. We generalise any cost function with Equation (2.9), where $\mathcal{L}$ demonstrates any cost function with network output and ground truth as input.

$$J(W,b) = \frac{1}{B}\sum_{k=1}^{B}\left(\frac{1}{2}\|h_{W,b}\left(x^{(k)}\right) - y^{(k)}\|^2\right) \tag{2.8}$$

$$J(W,b) = \frac{1}{B}\sum_{k=1}^{B}\left(\mathcal{L}\left[h_{W,b}\left(x^{(k)}\right), y^{(k)}\right]\right) \tag{2.9}$$

The backward pass aims to minimise this function. We use mini-batch gradient descent to perform this optimisation. Although MSE and other cost functions could reach a local optima due to the shape of the function [7], practical experiences such as [11] have shown that it general performs fine.

Backpropagation provides an efficient way to perform such optimisation. Originally proposed as an efficient and generic method to compute gradients, Rumelhart et al. [12] demonstrated that it allows much faster training time in neural networks than previous methods. The method uses partial derivatives to maintain its efficiency.

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha\frac{\partial}{\partial W_{ij}^{(l)}}J\left(W,b\right) \tag{2.10}$$

$$\frac{\partial}{\partial W_{ij}^{(l)}}J(W,b) = \frac{1}{B}\sum_{k=1}^{B}\frac{\partial}{\partial W_{ij}^{(l)}}J\left(W,b;x^{(k)},y^{(k)}\right) \tag{2.11}$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J\left(W, b\right) \tag{2.12}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b) = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial}{\partial b_i^{(l)}} J\left(W, b; x^{(k)}, y^{(k)}\right) \tag{2.13}$$

We aim to update the parameters using gradient descent as shown in Equations (2.10) and (2.12). They are updated by deducting partial derivates of the cost function. Backpropagation computes these derivates efficiently using Equations (2.11) and (2.13).

Combined with the forward pass (step 1), backpropagation works as follow:

1. Forward pass (step 1) computes activations for neurons in all n_l layers
2. For each neuron in the output layer n_l, we can simply compute the error by comparing the output activation and the true value. Mathematically, we utilises the partial derivatives of the activation functions:

$$\begin{aligned} \delta_i^{(n_l)} &= \frac{\partial}{\partial z_i^{(n_l)}} \mathcal{L}\left[h_{W,b}\left(x^{(k)}\right), y^{(k)}\right] \\ &= -\left(y_i - a_i^{(n_l)}\right) \cdot f'\left(z_i^{(n_l)}\right) \end{aligned} \tag{2.14}$$

3. Working backwards, we compute the error for each neuron in the hidden layers layers $n_l - 1$, $n_l - 2$, ..., 2. The error of a hidden layer neuron is calculated using a weighted average of the error terms that use its activation as an input. Hence, in mathematical notation,

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_l+1} W_{ji}^{(l)} \delta_j^{(l+1)}\right) f'\left(z_i^{(l)}\right)$$

4. With these error rates and activations, we can then compute the partial derivatives of the cost function for each datapoint. This is then used by gradient descent to update the parameters of the network, as seen in Equations (2.10) and (2.12).

$$\frac{\delta}{\delta W_{ij}^{(l)}} J(W, b; x, y) = a_j^{(l)} \delta_i^{(l+1)} \tag{2.15}$$

$$\frac{\delta}{\delta b_i^{(l)}} J(W, b; x, y) = \delta_i^{(l+1)} \tag{2.16}$$

Variations of Gradient Descent and Backgropagation have seen great success across supervised (e.g. [11, 13]) and unsupervised applications (e.g. [4]), especially in image-related networks.

2.2 Convolutional Neural Network (CNN)

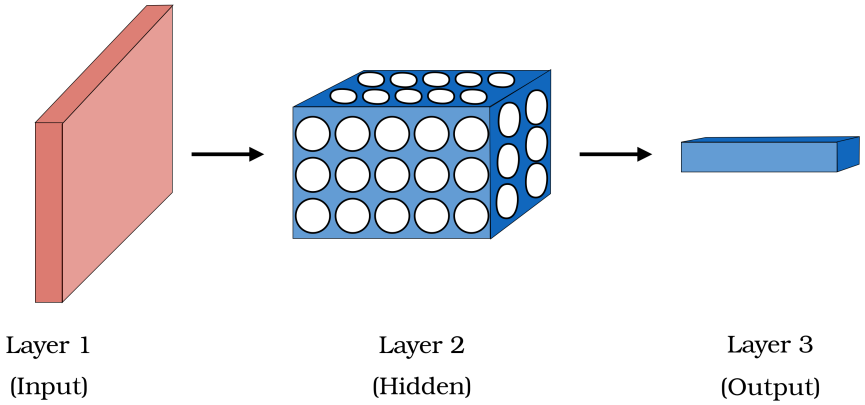

Figure 2.4: *A one-hidden layer Convolutional Neural Network. A colour image is three-dimensional (height, width, colour depth). Each convolutional layer transforms this input into another three-dimensional space of neuron activation. (Adapted from [8].)*

For a problem involving images, a conventional neural network as shown in Figure 2.3 is not sufficient. Its input is represented in a single vector form. If we simply reshape an image to a single vector, we lose details about the internal structure such as dimensions and correlation between pixels. Also, the large amount of parameters required can easily overwhelm a traditional neural network, causing it to overfit. It increases quadratically as the image size grows. For instance, given a colour image of size $200 \times 200 \times 3$, it requires 120 000 weights for each neuron. More generally, given the number of colour channels of c and image dimensions of $n \times m$, we require weights $c \times n \times m$, which can be reduced to a $\alpha c \times n^2$. As a network would require more than a few neurons in a hidden layer, this means the weights required increases even more rapidly [8].

On a higher level, we require to be invariant to properties such as translation and rotation. We want a network that is capable of understanding the underlying structure that represents a particular group of objects, rather than learning the specific position of which an object is placed.

A different approach is proposed. In the 1980s, Fukushima [14] demonstrated how a Convolutional Neural Network (CNN) can be used to recognise pattern within simple digit and letter images, without the need of manual feature engineering. LeCun et al. [15] further extended the existing work and demonstrated outstanding results with simple image datasets, such as hand-written digits, on classification tasks and face detection, which led to a highly accurate US zip code recogniser that handles a single image of a zip code without requiring explicit segmentation [16].

2.2.1 Layers

2.2.1.1 Convolutional Layer

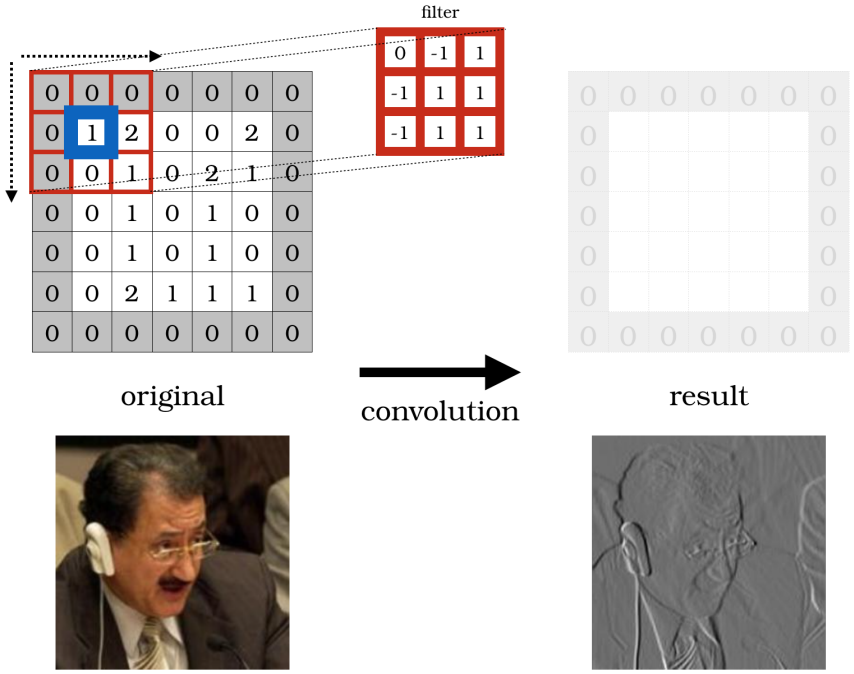

Figure 2.5: *An overview of the results of a convolution. A filter visits each pixel to calculate a new value by doing element-wise multiplication and summation. The results of a convolution varies using different filters. The bottom shows an example where convolution is used to detect edges. (Adapted from [8])*

Convolution is the underlying technique that makes this possible. The purpose of convolution is to capture the internal structures of images. It is difficult to distinguish between two categories of objects under very similar settings of two categories of objects simply by inspecting the individual pixels. Using many different filters in each hidden layers allows various features to be captured.

Combined with a neural network structure and making it deep, it identifies the invariance hidden underneath the raw pixels of input images. It starts from high to low level. Low level detail includes the shape or edges of a particular object within the scene, whereas high level detail may show an area of interest.

2.2.2 The Concept of Deconvolutional Network (DeConvNet)

Zeiler and Fergus identified that there is a lack of understanding about the internals as to the reasons that AlexNet performed significantly better than conventional machine learning algorithms. Through extending upon AlexNet, they introduce the DeConvNet.

It acts as a visualisation tool that traces the details extracted from an input image

through the network. Given a trained CNN, a DeConvNet is attached to a convolutional layer.

Activations are obtained by passing an image through this network. For a chosen layer and a feature map in the CNN, it takes its strongest activation and set all other activations in the layer as 0. This feature map is then passed into the DeConvNet.

Through reversing the process of the CNN in the DeConvNet, picking the top activations, and the same filters and pooling settings (or, 'switches') as the CNN, visualisation of the invariance learnt by each convolutional layer becomes possible [17]. This enables further improvements to the construction of CNNs through visual inspection of their inner workings.

2.2.2.1 Max Pooling

Max pooling layers are used to reduce the spatial size of the activations along the network. This reduction in size limits overfitting as the number of parameters reduces through this process [8]. It is usually used once between two convolutional operations as a downsampling method, and eventually creating a single vector called a latent variable. This is how a CNN automatically create a feature from an input.

2.2.3 Weight Initialisation

Network initialisation is important in training neural networks. For instance, if each neuron is given the same initial weight, they will all produce the same output and creates the same gradient, which defeats the purpose of a neural network where it aims to obtain different details through activating different neurons. [8].

A simple method to overcome this is by assigning small random numbers as weights for each neuron, but requires normalisation to the weights to keep the variance of the output of neurons to 1 [8]. However, this careful tuning requires much effort and is hard to get right [18].

2.2.4 Weight Normalisation

A popular solution is to use Batch Normalisation (BatchNorm) developed by Ioffe and Szegedy [18]. Consider a CNN trained using backpropagation and mini-batch stochastic gradient descent, a BatchNorm layer is placed before just after a convolutional layer but before non-linear transformation. It shows it achieves efficient and effective training performance that requires less concise initialisation of hyperparameters, and at times regularisation such as Dropout is not needed [18].

$$\mu_B = \frac{1}{B}\sum_{k=1}^{B} x_k \tag{2.17}$$

$$\sigma_B^2 = \frac{1}{B}\sum_{k=1}^{B} (x_k - \mu_B)^2 \tag{2.18}$$

$$\hat{x}_k = \frac{x_k - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{2.19}$$

$$\mathrm{BN}_{\gamma,\beta}(x_k) = \gamma \hat{x}_k + \beta \tag{2.20}$$

It aims to keeps the distribution of non-linearity activations stable throughout the network by enforcing a unit Gaussian distribution [8]. In this way, training is less affected by the value of hyperparameters such as learning rate.

In essence, the algorithm (Equation (2.20)) aims to learn the scale γ and shift β required in a mini-batch $B = \{x_1 \dots x_k\}$, through computing the normalisation (Equation (2.19)) using the mean (Equation (2.17)) and variance (Equation (2.18)). [18]

2.2.5 Regularisation

Regularisation methods are used to avoid overfitting. Two popular methods are *L2 regularisation* and *max norm constraints*. *L2 regularisation* adds the term $\frac{1}{2}\lambda w^2$ to penalises the cost function each weight w, where λ signifies the regularisation strength. This cause the cost function to decay linearly towards zero, such that training is stopped when the performace starts to deteriorate [19]. The advantage of doing so is that it encourages the network to use all inputs [8].

Max norm constraints use a different strategy to perform regularisation, in that it keeps an upper bound on the weight w for each neuron [8]. This protects the network from saturating if it is initialised with inappropriate parameters, such as too high a learning rate.

These methods are often used to alongside *Dropout*, created by Srivastava et al. [19], that is efficient and simple to run [8]. One can think of *Dropout* as a way of adding a Bernoulli noise $\mathbf{r}$ to the hidden units in a subnetwork before non-linear transformations. It aims to minimise overfitting by training an exponential number of subnetworks where each omits some units randomly determined by probability p, with network parameters updated according to the training dataset. This prevents the network from fitting on sampling noise, rely on other hidden units to rectify mistakes and believe them as the complex relationship within the training dataset [19]. Mathematically, we simply incorporate this noise to Equations (2.6) and (2.7) to achieve *Dropout*, by replacing a with $\hat{a}$, an element-wise product between the input and r, i.e. $\hat{a} = \mathbf{r} * a$.

2.2.6 Activation Functions

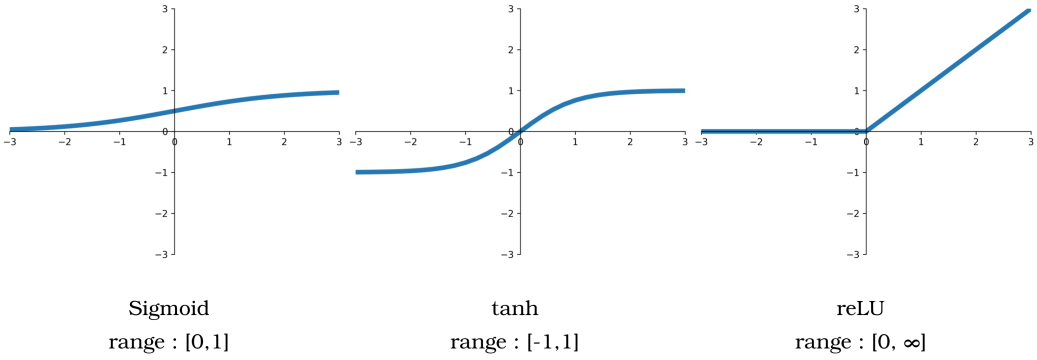

Figure 2.6: *Three commonly used activation function is used, namely (left to right), sigmoid, hyperbolic tangent (tanh) and rectified linear unit (ReLU), which exhibit different behaviour.*

The power of deep neural networks lies in the fact that non-linear activations are used to map raw input data to output. Complex datasets are often non-linear in nature. Using non-linear functions enables intricate details within the data to be captured, and preserve translational invariance especially in high dimensional datapoints [20].

Also, we want to project the data into a spce that is easy to work with. Through bounding the data within an interval avoids calculus overflow. More importantly, neural netowrks work by stacking the results of the hidden layers. Without the use of non-linear activation functions, stacking these layers together has the same effect of having only one hidden layer, since $W_1 W_2 x = Wx$. Hence, the effort of stacking multiple layers becomes wasteful.

Function	**Range**	**Equation**	**Derivative**
sigmoid	[0,1]	$f(z) = \frac{1}{1+e^{-z}}$	$f'(z) = f(z)(1 - f(z))$
Hyperbolic tangent ($tanh$)	[-1,1]	$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$f'(z) = 1 - (f(z))^2$
Rectified Linear Unit ($ReLU$)	[0, ∞]	$f(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases}$	$f'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{otherwise} \end{cases}$

Table 2.1: *Mathematical properties of common activation functions.*

Figure 2.6 shows three popular non-linear activations being used in deep neural networks. Each possesses characteristics that make them useful. Not only are activation functions used to automatically create latent variables, it is an integral part in training the network.

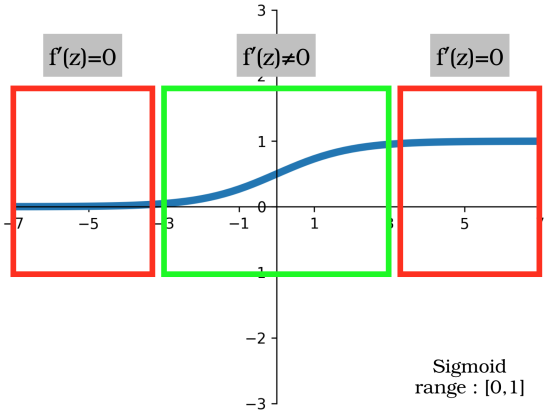

Figure 2.7: *The red boxes indicate the saturation boundaries of a sigmoid function where the problem of 'vanishing gradient' occurs. The green box indicates the section that is not affected by the problem.*

sigmoid was used in early neural network models but its downsides have seen it replaced by *tanh*. More recently, *ReLU* has become very popular, especially in neural networks that are designed to handle images [8]. The main reason that it is the least favourable activation function is that it tends to a zero gradient rapidly as it approaches the tails of the function, a problem known as *vanishing gradient* when it reaches the saturation boundaries, where $f'(z) \to 0$ during training. Gradient is essential in training a neural network, as seen in the backward pass section (Section 2.1.2.2). This would stop the training of the network.

tanh, a rescaled version of *sigmoid*, is preferred over sigmoid as it is centred in zero. This reduces the chance in which the parameters becoming all positive or negative, which makes the weight (gradient) update happen in a 'zigzag'. This allows the network to be trained more efficiently. However, it still suffers from the problem of vanishing gradient, as it is still a *sigmoid*-based function.

ReLU performs better than *sigmoid*-related functions, most notably in CNNs. It has shown the best practical results and which is the most popular choice [6].

There are a few reasons that *ReLU* has become popular. Firstly, it retains the properties of a linear function and does not have the inherent saturation problem in *sigmoid* and *tanh* [8]. Together with the fact that no exponential calculation is required, these enable a much faster training speed as evidenced in AlexNet [11].

To avoid neurons being neglected if there occurs a large gradient propagating through the network during training, *Leaky ReLU* aims to improve upon *ReLU* by giving a small negative slope (e.g. $0.01z$) rather than 0. Mathematically, we have:

$$f(z) = \begin{cases} \alpha z & \text{if } z < 0 \quad (\alpha \geq 0) \\ z & \text{otherwise} \end{cases}$$

However, there are inconsistent reports of success of using *Leaky ReLU*, so care must

be taken when choosing this as the activation function of a network [8].

2.3 Applications in Classification

Algorithm	Top-5 Error Rate	Year	Remarks
AlexNet [11]	15.4%	2012	• First major success of deep CNN • Next best had an error of 26.2% • Use of ReLU as activation
ZF Net [17]	11.2%	2013	• Introduces DeConvNet to visualise hidden layers
VGG Net [21]	7.3%	2014	• Only use 3×3 filters • Consecutive convolutions before max pooling to simulate larger filters while significantly reducing the number of parameters • Similar results to GoogLeNet but much simpler structure
GoogLeNet [13]	6.7%	2015	• Introduces inception module to create a concatenated output of multiple operations to maximise the detail captured with minimal impact on number of parameters
ResNet [22]	3.6%	2015	• Very deep network – 152 layers • Introduces residual block to calculate small changes with regards to input, which is easier to optimise and addresses the saturation problem as more layers are stacked together

Table 2.2: *Top performers of ImageNet Large Scale Visual Recognition Challenge (ILSVRC) that have made major contributions to the development of CNNs since 2012.*

In recent years, major improvements to computational power leads to the use of graphical processing units (GPUs) to handle machine learning tasks [23]. All the networks mentioned in Table 2.2 utilises multiple GPUs to train on large datasets over multiple days. This also leads to the rapid development and discovery of the capabilities of CNNs to handle much more complicated imagery datasets, as seen in breakthroughs in classification [11] and artificial intelligence agents in computer games [24]. The work of LeCun laid the fundamentals and underlying theories that are still relevant today.

Deshpande [25] provides a clear summary about some significant contributions made towards the development of classifying CNNs in the past 5 years. These major research effort are realised through the *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC) competition, where teams try to build the best machine learning algorithm to handle tasks such as classification and detection.

Although our project is focused on the generative prospect of CNNs, reviewing the current state of supervised applications enables us to understand the winning strategies which informs unsupervised applications.

2.3.1 Strategies

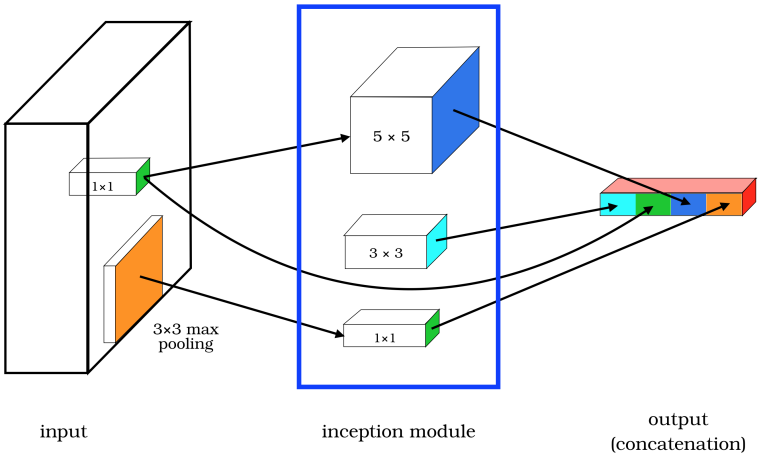

Figure 2.8: *An inception module that consists of three convolution* – 1×1, 3×3 *and* 5×5. *Their results are concatenated to form the output. (Adapted from [13].)*

These successful networks use different strategies to achieve better efficiency. For instance, GoogLeNet introduces the inception module (illustrated in Figure 2.8) to efficiently capture more details in one layer by concatenating the results of multiple operations. It would require many more parameters if they were layered out sequentially. VGG Net uses multiple small filters before downsampling to achieve a similar effect of a large filter, which would otherwise require more parameters. ResNet uses a very deep network but introduces the residual block to combat the inefficient and untraceable learning by directly mapping from input to output in a normal network.

Sometimes a simple structure could match the performance of one that is much more complex. Efficiency apart, VGG Net uses a much simpler network structure and performed similarly to GoogLeNet.

It is generally believed that the deeper a network is, the better it will perform, as more intricate details can be captured. However, it is found that a degradation problem occurs (not caused by overfitting) when the network starts to converge. This overwhelms the accuracy and makes it decrease rapidly again [22]. This causes a higher training

error, hence inherently higher testing error. Inception units in GoogLeNet and residual block in ResNet attempts to fix them.

2.4 Applications in Generation

Generative neural networks learn in an unsupervised manner, which goal is to learn to generate images that closely relates to real-life images. It is difficult to capture intricate details within a real-life image, such as the lighting and natural skin tone. More technically, historical generative models based on Boltzmann Machine requires approximating the values derived from intractable algorithms such as Maximum Likelihood and Markov chain [1, 3].

There are three main approaches to generative models — autoencoder, autoregression and adversarial. They can be CNN-based or with strong mathemtical background that utilises Markov chains. In particular, the adversarial framework, Generative Adversarial Nets (GAN) [3], and its applications such as Deep Convolutional GAN (DC-GAN) [4] have shown good promise. In the following, we shall discuss their properties and some of their successful applications.

2.4.1 Autoencoder

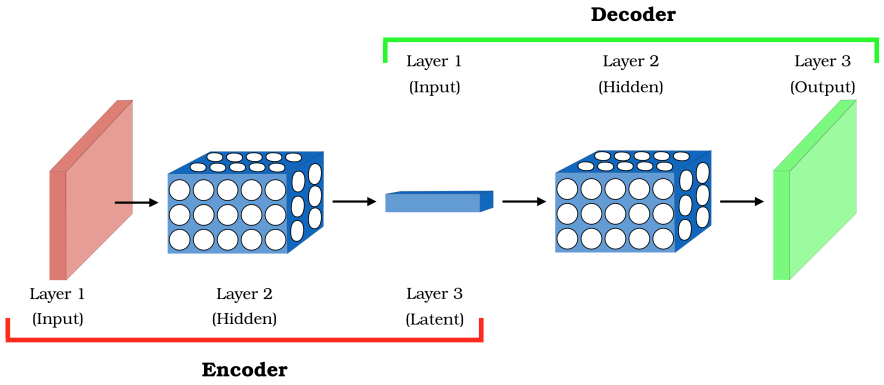

Figure 2.9: *A convolutional autoencoder with a 1-hidden layer encoder and decoder. The output of the encoder becomes the input of the decoder.*

There is a long history of autoencoders for the purpose of dimensionality reduction or feature learning. With more research into the theoretical connection between latent variable models and autoencoders, autoencoders have become a candidate for effective generative modelling [1].

The goal of an autoencoder is to generate an output that closely resembles the input. An autoencoder contains an encoder and a decoder. The encoder contains one or more hidden layers and is used to generate a latent variable (code) that describes the input,

i.e. $\mathbf{h} = f(\mathbf{x})$. This can be thought of as a 'compression' process, where it tries to represent the input in a smaller space.

The decoder then takes latent variables of this form and passes them through its hidden layers, i.e. $\mathbf{r} = g(\mathbf{h})$. It aims to 'decompress' the code and create a reconstruction of the input.

At training, autoencoders are designed to only be able to closely resemble but not perfectly copying the training input, as the aim is to learn about the intricate detail hidden within the training data using convolution [1]. If an autoencoder can perfectly replicate the training input, no learning is achieved, and is likely caused by overfitting. By learning useful details from the training data, given some new latent variable, the decoder should then be able to create some new output that has never been seen by the autoencoder before.

Deep Convolutional Inverse Graphics Network (DC-IGN) designed by Kulkarni et al. [26] that can interpret individual elements of an image so to generate new images of the same object but different poses and lighting. In a similar fashion as Figure 2.9 and cues from GoogLeNet [13] as discussed in Section 2.3.1, it disentangles the latent variable into separate features, such as pose and direction of light. They demonstrates that this creates much better reconstructions, as it is easier to control individual elements than correlated values within a vector [26].

2.4.2 Generative Adversarial Nets (GAN)

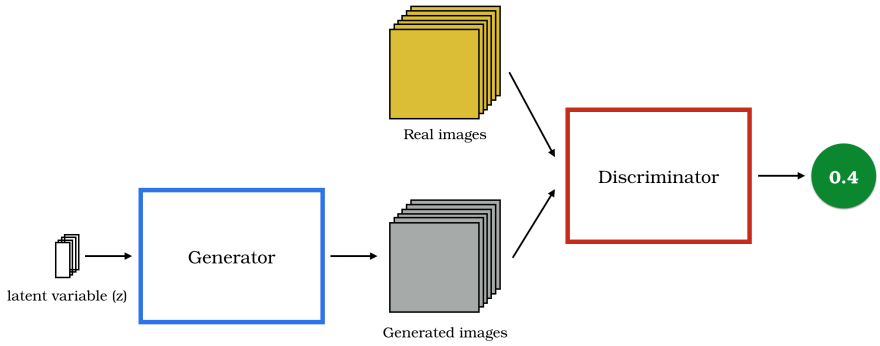

Figure 2.10: *The diagram denotes the structure of a GAN model, where there is a generator G (blue) that generates images, and a discriminator D (red) that determines how likely a given image is real.*

The popularity and success of CNN in discriminative applications leads to the research done by Szegedy et al. [27]. They found that neural networks in general are unstable with image inputs that are visually invariant but with small variations in the underlying data, even with a state-of-the-art deep CNN at the time such as AlexNet [11].

Also, by exploiting the fact that the smoothness assumption does not hold in neural

networks due to their non-linearity, they showed that it is possible to optimise the perturbations required to trigger a misclassification. Applying the findings of [27], Goodfellow et al. [3] describes the Generative Adversarial Nets (GAN) to train generative models using an adversarial approach.

There are two models in this framework — a generative model, G and a discriminative model, D. The two models are trained together and aimed to counteract each other. The purpose of G is to generate new images that aim to cause the discriminator to believe it is from the training dataset, and D acts as a classifier that calculates the probability that a sample is from the training data rather than from G [3].

Mathematically (all from [3]), both G and D are in the form of a multi-layer neural network. We define the generator as $G(\mathbf{z};\theta_g)$, which takes noise variables $\mathbf{z}$ with prior $p_{\mathbf{z}}(\mathbf{z})$, and parameters θ_g. It aims to learns the distribution p_g over training dataset $\mathbf{x}$, and maps the noise input to the training data space to generate adversarial examples. For the discriminator, we define it as $D(\mathbf{x};\theta_d)$. It maps the input space to a single value that denotes the probability $D(\mathbf{x})$ that the sample is from $\mathbf{x}$.

At the same time, we aim to maximise the probability of assigning the correct label with both $\mathbf{x}$ and $G(\mathbf{z};\theta_g)$ inputs, and create samples that *trick* D by minimising $\log\left(1 - D\left(G\left(\mathbf{z}\right)\right)\right)$.

This can be demonstrated as a minmax game between G and D for which the function V reveals the optimal:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[\log D\left(\mathbf{x}\right)\right] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}\left[\log\left(1 - D\left(G\left(\mathbf{z}\right)\right)\right)\right] \tag{2.21}$$

GAN converges when then generator has perfectly learnt the distribution of the input data $p_g = p_{\text{data}}$, i.e. $D(\mathbf{x}) = \frac{1}{2}$ [3]. Mini-batch stochastic gradient descent and back-propagation are used to update the weights as demonstrated in Section 2.1.2.2. Given a batch of B samples per batch, a hyperparameter n and learning rate η, we perform the following updates over some number of iterations:

1. For each iteration and for n steps, we obtain a new mini-batch of B samples from both the noise and training data. We then ascend the parameters of the discriminator D:

$$\theta_D = \theta D - \eta \nabla_{\theta_d} \frac{1}{B} \sum_{k=1}^{B} \left[\log D\left(x^{(k)}\right) + \log\left(1 - D\left(G\left(\mathbf{z}^k\right)\right)\right)\right] \tag{2.22}$$

2. For each iteration, we obtain a new mini-batch of B samples from the noise. We then descend the generator G:

$$\theta_G = \theta G - \eta \nabla_{\theta_g} \frac{1}{B} \sum_{k=1}^{B} \log\left(1 - D\left(G\left(\mathbf{z}^k\right)\right)\right) \tag{2.23}$$

GAN stands out as an important milestone for generative modelling because it can be trained just with backpropagation to compute the gradients, without the need to approximate the Markov chain values, which makes it much easier to train in practise [3].

However, convergence is not guaranteed and can cause underfitting in GAN [3]. It is possible that G and D causes V to fluctuate rather than reaching the optimal [1]. Also, GAN algorithms require careful hyperparameter tuning to achieve stability [1].

2.4.2.1 Deep Convolutional Generative Adversarial Network (DC-GAN)

GAN is difficult to scale for larger images and deeper generative models [4], DC-GAN [4] aim to improve this through various techniques learnt from how discriminative CNNs are built.

There are five main characteristics of DC-GAN:

- Contrary to traditional discriminative CNNs, DC-GAN incorporates the idea from the all convolutional net [28] that all pooling (in D) and unpooling (in G) layers are replaced by convolutional (strided convolutional) and deconvolutional (fractional-strided convolutional) layers. where no pooling and unpooling layer is used in G and D, as it is found that having the network learn its own upsampling and downsampling respectively.
- No fully connected layer is used, but the latent input to G and output of the last convolutional layer of D are simply reshaped to fit the next layer [4].
- BatchNorm (as discussed in Section 2.2.4) is used to enable a stable learning.
- Mainly *ReLU*, *Leaky ReLU* activations are used to produce high quality results.
- Together with backpropagation, mini-batch stochastic gradient descent with Adam and tuned parameters are used to learn the weights throughout the networks.

It is interesting to note that CNN structures in many notable networks, such as DC-GAN, are defined through performing experiments. For instance, DC-GAN would not work without BatchNorm but there is no proven explanation for this, apart from the fact that it.

2.5 Ranking

Our project aims to provide map concrete values to some subjective metric that is hard to quantify. Ranking is a method that we may use. It is used to create a hierarchy within some given datapoints. It is an important task for information retrieval tasks such as search. For instance, the success of a search engine relies on its ability to locate the most relevant documents governed by some input keywords.

Broadly, there are three main approaches to learning a ranking model [29]: pointwise, pairwise and listwise:

- **The pointwise approach**
  The approach takes the feature vector of one single document and uses a scoring function f to give it a predicted relevance value to signify its ordering. It only considers each document individually without taking into account their dependencies amongst each other. This poses as an disadvantage to tasks that relies on the close consideration between items, such as a search engine.
- **The pairwise approach**
  Pairs of feature vectors are compared between themselves. '1' is usually assigned to signify the more relevant of the pair and '-1' to the other. In mathematical terms, given a pair of vectors (x_u, x_v), there exists a scoring function f and an indicator function I_A that outputs 1 if the comparison A, for example $f(x_u) \geq f(x_v)$, is valid or 0 otherwise. This is performed for each pair elements to obtain $y_i = 2 \cdot I_A - 1$. This comparison enables relationships between the pair to be captured. It enables the direct application of classification methods to classify between pairs.
- **The listwise approach** The approach takes ranked lists of feature vectors as input and learn a function that orders the lists. It aims to minimise the loss funciton that maps the list to their orders.

A pairwise approach seems to be most fitting to our problem, as there would not be exact values that users can specify. We can provide a fixed scale which can limit the ambiguity if otherwise an exact value is required. We shall further investigate this in the next section.

2.6 Summary

In this literature review, we looked at the state of CNNs. The rapid development of CNN is driven by a few key factors, including the availability of a *large* labelled image dataset and the vast increase in computational power offered by GPUs.

We also discovered that the design of the architecture of a network is a creative process. The number and type of layers, and the structural composition depend on some asserted assumptions or through experimentation with various settings. And, we have briefly looked at the three approaches to ranking which would guide us to pick the right ranking algorithm that fits our task.

Chapter 3

Methodology

In the previous chapter, we look at the current state of CNN and how they are used in supervised problems such as classification, and unsupervised problems such as image generation. We saw that GAN has become a popular type of generative model. DC-GAN and other implementations have demonstrated that GAN can be used to generate high quality and sharp images. However, they lack the ability to control the generated images. By passing generated images from the network to a ranker, we aim to be able to impose an order on the latent variable about a subjective metric, and therefore produce a desired new image based on a given score not known to the ranker before.

In essence, we plan to train two DC-GANs on two different datasets with associated rankers to enable us to rank their latent variables. Then, we aim to take an 'iterative linear interpolation' approach to generate new images with a requested level of a metric.

In this chapter, we describe the setup we have and the steps taken in attempt to govern the production of new images.

3.1 Datasets

Dataset	Datapoints	Chosen Attribute	Datapoints with Required Attribute	Type	Complexity
Digits	10 000	Rotation	10 000	Synthetic	Simple
LFW	13 235	Smiling	13 143	Real	Difficult

Table 3.1: *Summary of datasets used.*

The chosen datasets are a digits-of-1 dataset (*'Digits'*) [30] and an aligned version of the Labelled Faces in the Wild dataset (*'LFW-a'*) [31, 32]. They provide us with a large enough dataset to test our proposed method of iterative linear interpolation within the constraints of this project, and attribute labels that is helpful for training our rankers. By investigating how our method performs on a simple, synthetic dataset enables us to establish how it may be useful, while attempting this on a more complicated, real-life dataset enables us to explore the extend of which the method would work.

Figure 3.1: *Sample images taken from the Digits dataset.*

Digits contains digits of '1' of various rotation and thickness against a black background. We choose rotation as the attribute that we want to control image generation on. As seen in Figure 3.1, it is the most prominent feature that can be observed, which makes it easier to decide on whether one image has more 'positive' (clockwise) or 'negative' (anti-clockwise) rotation.

On the other hand, LFW-a is much more complicated in that it contains more noise with varying foreground and background objects. The alignment causes hard black border around the edges, which could be problematic as they may be picked up as being prominent features of the image. Here, we pick smiliness as the metric we want to control our image generation on, as it is an observable though subjective metric.

Figure 3.2: *Sample images takes from the LFW-a dataset [32], which is the LFW dataset [31] aligned with a commercial-grade system.*

3.2 Data Pre-processing

Despite a CNN is designed to perform automatic feature engineering to produce latent variables on their own, some data pre-processing is still crucial to enable the network to learn the information we want it to. This is particularly true for our ranker as we will see later in Sections 3.4 and 4.2.

3.2.1 Cropping

Figure 3.3: *Each image is cropped with 60 pixels removed from each edge to reduce background noise.*

Cropping enables the features we desire to be in the centre and removes unwanted noise, so that the network and ranker can pick up on the correct features. If there are elaborate objects apart from our subject, this could hinder the performance of our training, especially in the ranker due to its ability to scale with increasing dimension. However, this has to be performed carefully. Removing too much of an image away could reduce diversity in the dataset. Without diversity, the network would not be able to learn much meaningful information to generate new pictures.

3.2.2 Normalisation

Image data are normalised in two ways to ensure that it is centred at zero and within a given range. For all datasets, we normalise the image data at training and testing time so that they fall into the range [-1, 1]. It enables us to work on the same scale across images so to allow the network take them with equal importance.

3.3 The Generative Model

We use DC-GAN [4] as our model, with slight modifications to the code to handle data pre-processing, specific input paths and visual outcome. However, we avoid changing the hyperparameter settings, such as the batch size and image size. As researched, we found out that the success of a CNN is usually through experimenting with various structures and settings, rather than with strong mathematical proofs.

Dataset	**Epochs**	**Iterations (Epoch $\times$ Images / Batch Size)**
Digits	400	$400 * \frac{8000}{64} = 50\,000$
LFW-a	1000	$1000 * \frac{10\,950}{64} \approx 117\,000$

Table 3.2: *Summary of DC-GANs trained.*

The number of iterations is one of the main values that govern the amount of information captured by the network through the training set. The general idea is to have as many iterations as possible. Through visual inspection of the generated images of networks with different number of epochs (as we keep the given batch size from [4]) given time and other constraints, we find these value produces the most visually plausible results.

Figure 3.4: *Overfitting occurred while training DC-GAN on Digits.*

However, the effect of the number of iterations can be limited by the number of training datapoints and diversity in the dataset. For instance, we attempted to train the Digits

network with more iterations but it caused overfitting and was unable to generate any digits of 1.

After training the model, we remove the discriminator and just use the generator, as we are only interested in generating new images.

3.4 The Ranker

With the help of the discriminator, the model fits noises sampled from the normal distribution to plausible outputs. So far, we have a trained network simply by providing it with pre-processed images with an appropriate number of iterations, which enables us to generate new, unseen images (see Epoch 400 of Figure 3.4).

However, there is not a way that allows us to create an image governed by a specific metric, especially one that is hard to quantify through inspection. As mentioned, we aim to achieve such generation of images through imposing an order in the latent space with the help of a ranker, so to provide users with a quantitative way to specify 'how much' of a metric they want to observe in a generated image.

3.4.1 RankSVM

We pick RankSVM as our ranker algorithm for a few reasons. The algorithm provides a generic interface that works with different types of data. Also, as it is based on Support Vector Machines (SVM), it inherits its ability to handle higher dimension data and being generally stable [29].

We use the efficient implementation of Chapelle et al. [33]. It uses the Newton method to improve efficiency so that it works on large number of comparisons, as opposed to the original implementation by Joachims [34] which struggles to scale [33]. This implementation requires three inputs:

- **A**: a sparse matrix of pairwise comparisons of the input data
- **C**: a vector of hyperparameters for each pair that indicates the amount of training error penalisation
- **X**: a matrix containing 1-dimensional feature vectors (feature vectors derived from our training datasets in our case)

By using the ground attribute labels provided by our chosen datasets, we can compute **A** through comparing them in pairs exhaustively. For instance, in the Digits case, we have labels on the predefined rotations of the given dataset. Each row of **A** should contain exactly a '1' for the larger pair element and '-1' for the smaller with no other values. We use 1000 datapoints to train each of our rankers due to memory limitation.

As SVM is a supervised classification algorithm, we can perform qualitative evaluation to measure its performance. In other words, this enables us to perform cross-validation to locate the best hyperparameter c out of a subset that we tested with. Briefly, cross-validation subdivides the training set into a training and a validation set. We perform this 10 times (i.e. a 10-fold cross validation) to obtain the average training error against the validation set. We then train on the whole set with the chosen c value, which is given to each pair to form $\mathbf{C}$.

We require some methods to extract features to form the data matrix $\mathbf{X}$. Like other traditional machine learning methods, careful feature extraction is key to the success of the model. Trivially, one might simply flatten an image into a vector to use as the input, but the correlations between pixels, rows and columns are lost, making it difficult to generalise. Also, the size of each feature will increase quickly if such method is used. Here, we use the GIST descriptor as our feature extractor [35]:

- **GIST Descriptor**

  GIST provides an efficient way to extract features of an image without requiring elaborate processing such as segmentation and dealing with each object and regions in the scene [35]. It captures information of an image at a low dimensional space through convolutions using Gabor filters, with the resultant vector encapsulating perceptual information such as naturalness and ruggedness. This information turns out to reliably capture and describe the information stored within an image.

  It produces a vector of 512 elements regardless of the size of the input image. This enables us to control the scale whilst capturing useful and precise information that we can use to create a ranking.

With these inputs, we train a ranker which provides us with a vector w. To obtain a score from the ranker, we simply have to multiply a gist feature vector by w.

3.5 Ranking Latent Variables

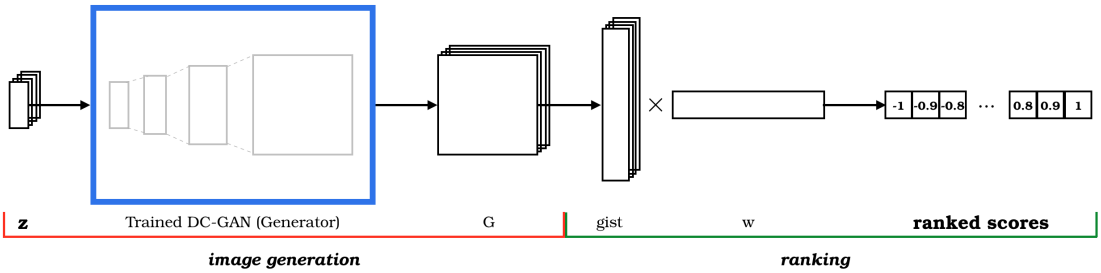

Figure 3.5: *Figure showing how we will incorporate the ranker with the generator to order the latent space.*

After obtaining a trained network and a ranker, we can now consider using them together to achieve the ranking step of our goal towards regulating image generation. We require to rank in the latent space. In order to do so, the ranker acts as an extended part of the generative model.

We first use new latent variables sampled from the normal distribution and pass z into the generator to obtain new generated images. With GIST, we obtain new features and multiply it with the learnt ranker weights w. By ordering the scores, we obtain a rank of the latent variables.

3.6 Latent Exploration

A brute force method for obtaining images of all ranks would simply be computing an infinite amount of images so that we can have an image for each score. However, this is not achievable so alternative methods have to used. The use of a simple algorithm can be beneficial as it is quick and easy to perform.

In order to explore in the latent space, we first perform linear interpolation between the extrema latents found in the ranked values. Interpolation enables a smooth transition from one extreme to another, acting to fill in the gaps. As the dimensions of the latent variable is rather low with respect to that of an image, interpolation within the latent manifold should enable a smooth transition from one extrema to another. Then, we plan to adopt an iterative approach in performing interpolation to obtain an image closet to the score requested.

The iterative approach algorithm is as follows:

1. Find the neighbouring latents (z_A and z_B) closest to the target
2. Perform interpolation between z_A and z_B to obtain the interpolation matrix $\mathbf{I}$

3. Obtain ranked scores for $\mathbf{I}$
4. Repeat from Step 1 until one of the neighbours reaches a score of $\epsilon(5, \text{target})$

3.7 Evaluating Quality of Network and its Outcome

With traditional or mathematical-proven machine learning models, there exists many quantitative methods in evaluating the outcome. For instance, precision-recall rates are used to examine the performance of a supervised model of classification. For our ranker, as discussed in Section 3.4.1, we perform cross-validation as a method to optimise our parameter that gives the highest accuracy rate.

However, GAN is difficult to evaluate as there is no meaningful metrics that can provide a value to describe the subjectivity of whether an image looks 'good'. Visual inspection remains the method used to evaluate the quality of the output images.

Chapter 4

Results

In this chapter, we discuss the results of what we have attempted to perform. This can be divided into three main parts:

- Network Performance
- Ranker Performance
- Data Exploration

In this chapter, we explain the results we have achieved and discuss about how this can be useful to inform future work.

4.1 Network Performance

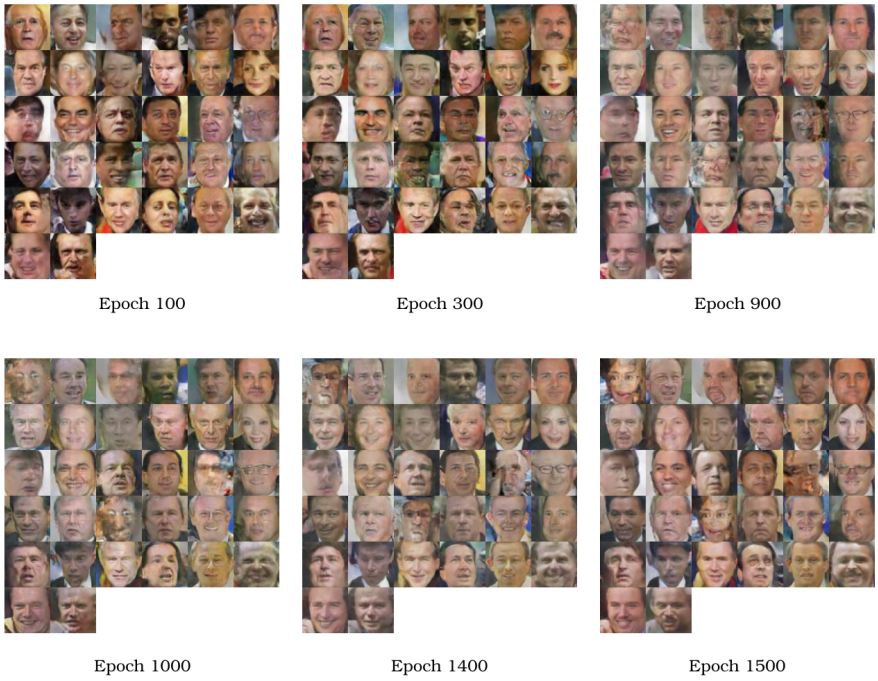

Figure 4.1: *Some generated images by the LFW-a generative model as the number of training epochs (which informs the number of iterations) grows.*

Visual inspection, although subjective, is the way used to evaluating the 'quality' of the output from a GAN. As it is seen, with more than 10 000 and a batch size of 64, we would have already run more than 234 000 iterations, there are still learning to be done. Due to hardware access limitation, we have only trained until 1500 epoch before reaching a near optimal network.

For our investigation, this should not pose a critical problem, as the inspected generated images are mostly plausible with a visible smiliness.

4.2 Ranker Performance

Dataset	Cropping (all sides)	Best C value	Cross-validation Accuracy
Digits	N/A	10	98.80%
LFW-a	N/A	0.01	68.15%
LFW-a	30	0.01	72.43%
LFW-a	40	0.01	74.21%
LFW-a	50	0.01	77.69%
LFW-a	60	0.01	80.87%

Table 4.1: *Summary of cross-validation results for training the RankSVM ranker for Digits and LFW-a. An LFW-a image is* 250×250 *at its native image dimension.*

We train a ranker for each of the two datasets we use. As mentioned, we perform cross-validation to quantify the performance of the ranker. With Digits, we achieve a ranker with 98.8% accuracy at cross-validation. This is expected due to the simplistic nature of the dataset.

On the contrary, the LFW-a ranker requires data pre-processing to achieve a *good enough* accuracy. We achieve an accuracy rate of 80.87% at cross-validation when the images are cropped to show only the face, as shown in Figure 3.3.

4.2.1 Ranking Test Images

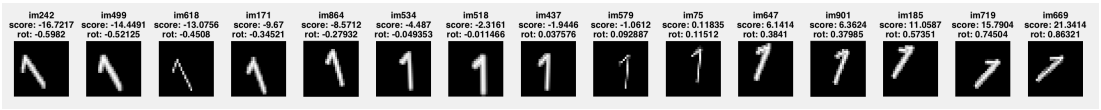

Figure 4.2: *From the test set of 1000 Digits images, we compute their gist descriptors and create an ordering with the trained ranker. 15 scores are picked randomly from the ranked values. Here we show the images corresponding to the scores chosen.*

Figure 4.3: *From the test set of 1193 LFW-a images, we compute their gist descriptors and create an ordering with the trained ranker. 15 scores are randomly picked from the ranked values. Here we show the images corresponding to the scores chosen.*

Expectedly, the Digits benefits from its simplicity and a highly accurate ranker. We observe that the digits nicely rotate clockwise from left to right. Given the complexity of the subtle differences in smiliness, the LFW-a ranker performs well in general. We can see that 'high smiliness' faces tend to the positive scale, whereas 'low smiliness' images tend to the left.

Figure 4.4: *This shows the relationship between the ground truth smiliness metric and the score calculated by the ranker of 1193 test images from the LFW-a dataset.*

Whilst the calculated score increases, the corresponding ground truth labels increases, though we observe that there is a large variance within the original scores. We do not observe this range of variance for Digits. This could be explained by the complexity of the dataset. The faces in LFW-a contains noisy backgrounds and various poses that would affect the features extracted by the ranker.

4.2.2 Ranking Latent Variables

Figure 4.5: *Using 1000 latent variables sampled from the normal distribution, we generate 1000 new images with the Digits GAN. 1000 scores can therefore be computed. From the sorted scores, we pick 15 samples and display their corresponding image.*

Figure 4.6: *Using 1000 latent variables sampled from the normal distribution, we generate 1000 new images with the LFW-a GAN. 1000 scores can therefore be computed. From the sorted scores, we pick 15 samples and display their corresponding image.*

We rank the latent variable through generating its corresponding images with the generator, and then passing it onto the ranker, as demonstrated with Figure 3.5. We see that for both datasets that the rankers are working well, where we can see the extrema at both ends. This also suggests that it is possible to rank at the latent space, which is part of our aim to enable us to generate a desired image.

4.3 Data Exploration

We perform linear interpolation in the latent space to try to explore if it is possible to create plausible intermediate images between two given images. We first perform an extrema interpolation, i.e. between the most negatively and positively scored latent variables. We then explain how an iterative interpolation approach could help to achieve our goal.

4.3.1 Extrema Interpolation

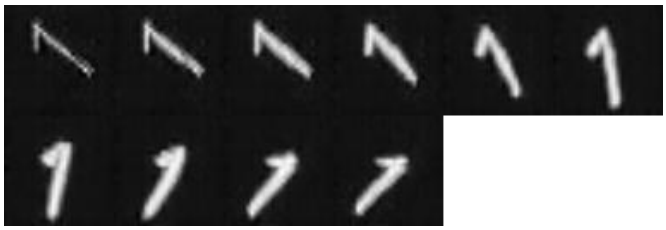

Figure 4.7: *Interpolation between two extrema of latents from generated Digits images.*

Figure 4.8: *Interpolation between two extrema of latents from generated LFW-a images.*

We interpolate between the latent variables that produces the lowest and highest scores obtained through the generated images, from 1000 latent variables sampled from the normal distribution.

In the Digits example, we observe that the interpolation works to generate the intermediate intervals. The most prominent feature within these images is rotation and there are little noise due to the near uniform black background across all images. This means that it is easy to control this visible feature.

We can also see the smiliness transformation and smooth transition occurring between the two extrema in the LFW-a interpolations. However, this smooth transition has some side effects. For instance, it has created a mapping between the two images , such that as the image becomes more like the extremum it is heading towards.

4.3.2 Iterative Interpolation

Supported by the results above, we can deduce an interactive interpolation approach in finding the best match to a given value of the criterion, which is detailed in Section 3.6. Using the linear interpolation between the neighbouring images of a target value provided by the user, we can reduce our search space and attempt to achieve the closest images desired by the user.

4.4 Summary

We have trained two plausible rankers on two contrasting datasets – a simple, synthetic dataset called Digits, and a difficult, real-life dataset called LFW-a. Despite the fact that the faces images are much more complex with the variations within the datasets, it obtains an acceptable level of pairwise comparison accuracy.

We have established that the simple algorithm of linear interpolation can be useful for generating intermediate images between two given images through their latent variables.

Looking at the LFW-a dataset – whilst we are able to rank and generate images with varying smiliness through linear interpolation, it also alters other correlated factors as a side effect. There is no control available within the latent space in this setup to govern individual metric while keep others constant. Other methods should be explored.

Chapter 5

Conclusion

5.1 Achievements

We have studied whether it is possible to regulate image production in GAN through the means of an external ranker and linear interpolation. On one hand, our approach shows that it is possible to rank images, even more complex, real-life images to some extend, and utilise linear interpolation to create plausible intermediate images with an increasing level of the given metric. We see this clearly displayed in the Digits datset. However, when an image becomes complex with many internal correlations, it becomes difficult to regulate just one metric in the latent space.

Whilst inconclusive, our investigation has led us to believe that simple methods such as linear interpolation can be useful when used with other methods.

5.2 Future Work

Building on top of our understanding of generative models and our investigation, there are other methods to govern the generative process of images. One method is to incorporate the ranker into a GAN model. By concatenating ranker score information to the latent variable, we attempt to impose further restrictions of the information learnt by the network. This enables us to segment the information into chunks that can then be regulated independently. We have seen examples that the concatenation of multiple parts of information together have produced good results in classficiation tasks with GoogLeNet [13] and in Deep Convolutional Inverse Graphics Network [26].

With more work, we believe the system we aim to produce can help with generating new 3-dimensional features such as body shape and height. This can also be useful in motion capture scenarios where new movements can be generated.

5.3 Final Thoughts

Regulating the generative process of a GAN is a complicated task. Our investigation has found that an external ranker and linear interpolation may not be sufficient in dealing with complex data, such as those with a lot of noise.

In this project, we reviewed the current state of convolutional neural networks in classification and generation. We then discussed our methodology in an attempt to regulate the generation of images in a GAN. We have discussed the strengths and limitations of our method.

Bibliography

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT Press, 2016.

[2] Carl Edward Rasmussen. "Gaussian processes for machine learning". In: (2006).

[3] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[4] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks". In: *arXiv preprint arXiv:1511.06434* (2015).

[5] Alan Lau. "Using Supervised Machine-learning Techniques to Identify Objects Classes in Images with Depth Data". In: *unpublished manuscript* (2016). URL: `https://www.melaus.xyz/files/dissertation.pdf`.

[6] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444.

[7] Andrew Ng. "Sparse autoencoder". In: *CS294A Lecture notes* 72.2011 (2011), pp. 1–19.

[8] Andrej Karpathy. "Cs231n: Convolutional neural networks for visual recognition". In: *Online Course* (2016).

[9] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

[10] Diederik Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[12] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *Cognitive modeling* 5.3 (1988), p. 1.

[13] Christian Szegedy et al. "Going deeper with convolutions". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 1–9.

[14] Kunihiko Fukushima and Sei Miyake. "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition". In: *Competition and cooperation in neural nets*. Springer, 1982, pp. 267–285.

[15] Yann LeCun et al. "Generalization and network design strategies". In: *Connectionism in perspective* (1989), pp. 143–155.

[16] Ofer Matan et al. "Reading handwritten digits: A zip code recognition system". In: *Computer* 25.7 (1992), pp. 59–63.

[17] Matthew D Zeiler and Rob Fergus. "Visualizing and understanding convolutional networks". In: *European conference on computer vision*. Springer. 2014, pp. 818–833.

[18] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[19] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting." In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[20] Quoc V Le et al. *A Tutorial on Deep Learning Part 1: Nonlinear Classifiers and The Backpropagation Algorithm*. 2015.

[21] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[22] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 770–778.

[23] Dave Steinkraus, I Buck, and PY Simard. "Using GPUs for machine learning algorithms". In: *Document Analysis and Recognition, 2005. Proceedings. Eighth International Conference on*. IEEE. 2005, pp. 1115–1120.

[24] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[25] Adit Deshpande. *The 9 Deep Learning Papers You Need To Know About (Understanding CNNs Part 3)*. 2016. URL: `https://adeshpande3.github.io/adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html`.

[26] Tejas D Kulkarni et al. "Deep convolutional inverse graphics network". In: *Advances in Neural Information Processing Systems*. 2015, pp. 2539–2547.

[27] Christian Szegedy et al. "Intriguing properties of neural networks". In: *arXiv preprint arXiv:1312.6199* (2013).

[28] Jost Tobias Springenberg et al. "Striving for simplicity: The all convolutional net". In: *arXiv preprint arXiv:1412.6806* (2014).

[29] Tie-Yan Liu et al. "Learning to rank for information retrieval". In: *Foundations and Trends in Information Retrieval* 3.3 (2009), pp. 225–331.

[30] Kwang In Kim, Florian Steinke, and Matthias Hein. "Semi-supervised regression using hessian energy with an application to semi-supervised dimensionality reduction". In: *Advances in Neural Information Processing Systems*. 2009, pp. 979–987.

[31] Gary B. Huang et al. *Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments*. Tech. rep. 07-49. University of Massachusetts, Amherst, 2007.

[32] Yaniv Taigman, Lior Wolf, Tal Hassner, et al. "Multiple One-Shots for Utilizing Class Label Information." In: *BMVC*. Vol. 2. 2009, pp. 1–12.

[33] Olivier Chapelle and S Sathiya Keerthi. "Efficient algorithms for ranking with SVMs". In: *Information Retrieval* 13.3 (2010), pp. 201–215.

[34] Thorsten Joachims. "Optimizing search engines using clickthrough data". In: *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2002, pp. 133–142.

[35] Aude Oliva and Antonio Torralba. "Modeling the shape of the scene: A holistic representation of the spatial envelope". In: *International journal of computer vision* 42.3 (2001), pp. 145–175.