OPTIMIZATION OF CONVOLUTIONAL NEURAL NETWORKS FOR IMAGE CLASSIFICATION USING GENETIC ALGORITHMS AND BAYESIAN OPTIMZATION

by

Waseem Rawat

submitted in accordance with the requirements for
the degree of

MAGISTER TECHNOLOGIAE

In the subject

Electrical Engineering

at the

University of South Africa

Supervisor: Professor Zenghui Wang

January 2018

Declaration

Name: Waseem Rawat

Student number: 58557199

Degree: Magister Technologiae (Mtech): Electrical Engineering

Exact wording of the title of the dissertation or thesis as appearing on the copies submitted for examination:

Optimization of Convolutional Neural Networks for image classification using Genetic Algorithms and Bayesian optimization

I declare that the above dissertation/thesis is my own work and that all the sources that I have used or quoted have been indicated and acknowledged by means of complete references.

_____ 15/01/2018

SIGNATURE DATE

Acknowledgements

The completion of this dissertation would not have been possible if I did not have the support of a few key individuals and therefore I would like to take this opportunity to extend my heartfelt gratitude to all of them.

Firstly, I would like to express my profound appreciation to my supervisor, Professor Zenghui Wang, from the Department of Electrical and Mining Engineering at the University of South Africa (UNISA) for his exemplary guidance and for the selfless time he made available to read through and correct my various drafts. His attention to detail and experience in the publication process continuously challenged me to improve the standard of my work and for this, I am sincerely grateful.

I also owe my thankfulness to my colleagues at Toyota South Africa Motors for their cordial support and cooperation during the lifespan of my research. Regrettably, I cannot acknowledge them by name.

Finally, I would like to extend a special thanks to my family and friends, and especially my loving wife and daughter, for their unceasing encouragement and inspiration. This dissertation would not have been possible if I did not have their full support and for this, I am eternally obliged.

Dedication

To my loving and supportive family

Abstract

Notwithstanding the recent successes of deep convolutional neural networks for classification tasks, they are sensitive to the selection of their hyperparameters, which impose an exponentially large search space on modern convolutional models. Traditional hyperparameter selection methods include manual, grid, or random search, but these require expert knowledge or are computationally burdensome. Divergently, Bayesian optimization and evolutionary inspired techniques have surfaced as viable alternatives to the hyperparameter problem. Thus, an alternative hybrid approach that combines the advantages of these techniques is proposed. Specifically, the search space is partitioned into discrete-architectural, and continuous and categorical hyperparameter subspaces, which are respectively traversed by a stochastic genetic search, followed by a genetic-Bayesian search. Simulations on a prominent image classification task reveal that the proposed method results in an overall classification accuracy improvement of 0.87% over unoptimized baselines, and a greater than 97% reduction in computational costs compared to a commonly employed brute force approach.

Key words: Deep learning, Artificial neural networks, Convolutional neural networks, Evolutionary algorithms, Genetic algorithms, Bayesian optimization, Computer vision, Image classification, Model selection, Hyperparameter optimization

Table of contents

List of figures

List of tables

List of acronyms

ANN	Artificial neural network
API	Application programming interface
CIFAR	Canadian Institute for Advanced Research
CNN	Convolutional neural network
CPU	Central processing unit
DCNN	Deep convolutional neural network
DNN	Deep neural network
EA	Evolutionary algorithm
ELU	Exponential Linear Units
GA	Genetic algorithm
GPU	Graphics processing unit
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
Max pooling	Maximum pooling
MOP	Multi-scale orderless pooling
MNIST	Mixed National Institute of Standards and Technology
MOE	Metrics Optimization Engine
NiN	Network in Network
PSO	Particle swarm optimization
RAM	Random access memory
R-CNN	Regions with CNN features
ReLU	Rectified Linear units
SELU	Scaled exponential linear units
SaaS	Software as a Service
SGD	Stochastic gradient decent
SNN	Self-normalizing neural networks
SSA	Sequential search algorithm
TI-pooling	Transformation invariant pooling

CHAPTER 1

Introduction

1.1 Background

Image classification, which can be defined as the task of categorizing images into one of several predefined classes, is a fundamental problem in computer vision. It forms the basis for other computer vision tasks such as localization, detection and segmentation (Karpathy, 2016). However, although the task can be considered second nature for humans, it is much more challenging for an automated system. Some of the complications encountered include viewpoint-dependent object variability and the high in-class variability of having many object types (Ciresan, Meier, Masci, Gambardella and Schmidhuber, 2011). Traditionally, a dual stage approach was used to solve the classification problem. Handcrafted features were first extracted from images using feature descriptors and these served as an input to a trainable classifier. The major hindrance of this approach was that the accuracy of the classification task was profoundly dependent on the design of the feature extraction stage and this usually proved to be a formidable task (LeCun, Bottou, Bengio and Haffner, 1998).

In recent years, deep learning models that exploit multiple layers of nonlinear information processing, for feature extraction and transformation as well as for pattern analysis and classification, have been shown to overcome these challenges. Amongst them, convolutional neural networks (CNNs - LeCun et al., 1989a, 1989b) have become the leading architecture for most image recognition, classification and detection tasks (LeCun, Bengio and Hinton, 2015). Despite some early successes (LeCun et al., 1989a, 1989b; LeCun et al. 1998; Simard, Steinkraus and Platt, 2003), deep CNNs (DCNNs) were brought into the limelight as a result of the deep learning renaissance (Hinton, Osindero and Teh, 2006; Hinton and Salakhutdinov, 2006; Bengio, Lamblin, Popovici and Larochelle, 2006), which was fuelled by graphics processing units (GPUs), larger datasets and better algorithms (Krizhevsky, Sutskever and Hinton, 2012; Deng and Yu, 2014; Simonyan and Zisserman, 2014; Zeiler and Fergus, 2014). Several advances such as the application of GPU's (Chellapilla, Puri and Simard, 2006), and maximum pooling (max pooling) for DCNNs (Ranzato, Huang, Boureau and LeCun, 2007) have all contributed to their recent popularity. In particular, their popularity was significantly enhanced by the now famous AlexNet architecture (Krizhevsky, Sutskever and Hinton, 2012), which won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012, and highlighted the advantages of several

modern techniques such as Rectified Linear Units (ReLUs - Nair and Hinton, 2010), Dropout regularization (Hinton, Srivastava, Krizhevsky, Sutskever and Salakhutdinov, 2012), and parallel GPU implementations. Following on from the successes of the AlexNet model, DCNNs have continued to evolve and some of their representative improvements include the Network in Network (NiN – Lin, Chen and Yan, 2013), Inception (Szegedy, Liu, et al., 2015) and ResNet (He, Zhang, Ren and Sun, 2015b) models.

These accomplishments have led researchers to progress several DCNN components, resulting in a plethora of improvements to their architecture, pooling layers, activation functions, loss functions, regularization techniques, optimization procedures, and computational characteristics, as elaborated on in Rawat and Wang (2017). In fact, in the period since 2012, the successes of modern DCNNs has been so astounding, that on certain image classification tasks, they have even surpassed human-level performance (Ciresan, Meier and Schmidhuber, 2012; Wan, Zeiler, Zhang, LeCun and Fergus, 2013; Ioffe and Szegedy, 2015; Szegedy, Vanhoucke, Ioffe, Shlens and Wojna, 2015; He, Zhang, Ren and Sun, 2015a, 2015b; Szegedy, Ioffe, and Vanhoucke, 2016).

On the other hand, DCNN successes have prompted others to scrutinize their internal mechanisms and gain a better understanding of their operation and expressive ability, resulting in research into several open issues that are still being actively researched. For example DCNNs are not invariant to large scale geometric deformations (Gong, Wang, Guo and Lazebnik, 2014), current models impose considerable storage and memory constraints averting mobile deployment (Han, Mao and Dally, 2016; Iandola et al., 2016), describing the semantic content of images is still a challenging problem (Vinyals, Toshev, Bengio and Erhan, 2015) and the lack of robustness to adversarial (Szegedy et al., 2014) and nonsense (Nguyen, Yosinski and Clune, 2015) images poses significant risks to security dependent applications. Furthermore, despite some progress (Mallat, 2012; Wiatowski and Bolcskei, 2015; Basu et al., 2016; Bengio, Mesnard, Fischer, Zhang and Wu, 2017), theoretical motivations of why DCNNs are successful are largely devoid.

1.2 Motivation

Further to the challenges described above, DCNNs require numerous architectural choices and hyperparameters, such as the number and size of the convolutional and pooling filters, the need to use or negate regularization techniques such as Dropout (Hinton et al., 2012), and the important choice of which activation function to use. The learning parameters such as the optimization technique, and its associated learning rate, the number of epochs and the size of each batch presented to the network, and the weight initialization method to adopt, also need

to be selected. When the learning choices are combined with the architectural choices, the number of possible models grow exponentially with each additional parameter, making DCNN model selection a fine art, compounded by the fact that deep models are computationally expensive to compute.

The traditional methods for model selection include grid search[1] (Pedregosa et al., 2011, random search (Bergstra and Bengio, 2012), and manual tuning; however all of these have their own challenges. Manual model selection requires expert domain knowledge or unsystematic rules of thumb (Dernoncourt and Lee, 2016), grid search is computationally burdensome (Snoek, Larochelle and Adams, 2012), and whilst random search relaxes some of the computational load imposed by grid search, it is not directed towards promoting high performing models. On the other hand, intelligent model selection strategies such as Evolutionary Algorithms (EA's), and more specifically Genetic Algorithms (GA's), have been shown to perform better than grid search techniques for support vector machines (Friedrichs and Igel, 2005; Huang and Wang 2006; Martino, Ferrucci, Gravino and Sarro, 2011) and neural networks (Ding, Li, Su, Yu and Jin, 2013), and they can be parallelized, however they are challenged when faced with complex and high dimensional search spaces, and furthermore, applications to modern DCNNs have been limited.

Divergently, Bayesian optimization has emerged as an influential solution for the automated selection of deep neural network (DNN) models (Snoek et al., 2012; Swersky, Snoek and Adams, 2013; Snoek et al., 2012; Shahriari, Swersky, Wang, Adams and de Freitas, 2016), and in particular, Bayesian optimization based on Gaussian processes (Rasmussen and Williams, 2006) is known to work well for continuous variables (Loshchilov and Hutter, 2016). Despite this, Bayesian algorithms impose a significant administrative overhead and require expert knowledge in order to obtain sensible results (Dewancker, McCourt and Clark, n.d), and furthermore, is inherently sequential in nature, thus preventing superlative parallelization (Loshchilov and Hutter, 2016).

Thus, it can be deduced that although GA's and Bayesian optimization have their advantages for model selection, both of these approaches have their own challenges. Therefore, in this dissertation, an alternative automated approach that combines using these approaches for their respective benefits, is studied. More specifically, it deals with the utilization of genetic and Bayesian search techniques to explore the near optimal DCNN model for the momentous computer vision task of image classification.

[1] Grid search is sometimes referred to as brute-force computation

1.3 Research aim and objectives

In the context of this dissertation, the aim relates to the overall driving force of the research, whilst the objectives focus on the means by which the aim will be achieved (Dawson, 2002).

Aim: *The main aim of this study is to conduct a pragmatic and firsthand evaluation into reducing the computational costs incurred during DCNN model selection, for the task of image classification, and to explore the associated fundamentals.*

Objectives: The specific objectives of this dissertation are as follows:

1) To establish and describe the fundamental concepts associated with the architecture and optimization of DCNNs for the task of image classification.
2) To investigate the motivations behind the recent renaissance in the application of DCNNs for visual tasks and to expose their open challenges, focusing on the problem of hyperparameter selection in DCNNs.
3) To propose a methodology to use a stochastic GA to search the architectural hyperparameter space of DCNNs.
4) To establish a Bayesian search on top of the GA search to traverse the learning parameter subspace.
5) To empirically evaluate the results of these optimization methods, in order to measure and improve the effectiveness (classification accuracy and ability to generalize) and efficiency (computational cost and complexity) of DCNNs.

1.4 Contribution

Whilst DCNNs have emerged as the leading models for image classification tasks (LeCun et al., 2015; Rawat and Wang, 2017), the computational encumbrance imposed by their large search spaces is still an open challenge that requires further attention. The contributions of this dissertation is aimed towards addressing this challenge, by proposing and empirically verifying an alternative automated approach to efficiently traverse the search space to find the near optimal DCNN hyperparameters, with the intention of reducing computation by removing the need for a computationally exhaustive grid search approach or the requirement for domain specific expert model selection. The specific contributions and departure points from other work are as follows:

- A survey on the motivations for DCNNs for image classification tasks, together with several of their open issues, focusing especially on their large model selection search spaces, is provided.
- The large search space of modern DCNNs is separated into discrete architectural, and categorical and continuous learning subspaces, with the intention of applying different optimization techniques to search for their optimal parameters.
- A biologically inspired stochastic genetic algorithm (GA) for the model selection problem is presented, and it is applied to efficiently search the architectural space.
- The evolutionary architectural search is combined with a state-of-the-art Bayesian search on top of the stochastic GA, and the hybrid approach is applied to efficiently traverse the learning subspace.

1.5 Research design and methodology

From a philosophical point of view, this dissertation adopts a positivist tone, since it is associated with the empirical verification of the presented optimization methods through quantitative research methods, with numerical evidence (Orlikowski and Baroudi, 1991; Knox, 2004). It involves the presentation of new algorithms to automatically search for the near optimal DCNN hyperparameters, and this is followed by the experiential substantiation of the presented methods, which are analyzed in detail. Experimental results and comparisons to other techniques are obtained using traditional computer vision benchmarks.

1.6 Dissertation outline

The remainder of this dissertation consists of six chapters, which are closely linked to the research objectives discussed in Section 1.3. They are structured as follows:

Chapter 2: This chapter introduces the application of DCNNs for image classification and provides the related theoretical background on their architectures, optimization techniques and initialization procedures.

Chapter 3: This chapter concisely surveys the reasons behind the deep learning renaissance and the motivations in favour of DCNNs for image classification tasks. It also contextualizes the focus of the research by introducing some of the remaining challenges of DCNNs, with a special emphasis on the challenge of hyperparameter optimization and the computational problems imposed by the large search space of our modern models. This is followed by the formalization of the problem and discussions on the common approaches used to address it.

Chapter 4: This chapter provides the background to the proposed optimization techniques. It begins by introducing evolutionary and genetic algorithms and several of its operations. Next, a brief survey on the use of GA's for neural network model selection is provided. Thereafter, the chapter motivates for the use of Bayesian optimization for DCNN model selection before providing a formal introduction to the technique.

Chapter 5: In this chapter a methodology of applying a stochastic GA to the task of DCNN model selection is formalized, and this is followed by several simulations to demonstrate the computational efficiency of the proposed method. The technique is analysed in detail before the chapter is concluded.

Chapter 6: In this chapter, another optimization algorithm is proposed to supplement the GA proposed in the previous chapter. More specifically a methodology of using a GA-Bayesian optimization algorithm on top of the stochastic GA presented in the previous chapter is formalized. Thereafter, the method is empirically verified and analysed, and its computational efficiency demonstrated.

Chapter 7: In Chapter 7, the study is summarized and concluded, and its limitations elaborated on. Future work and recommendations on the use of the GA-Bayesian optimization procedure for DCNN model selection are also discussed.

CHAPTER 2

Preliminaries and theoretical background

2.1 Chapter overview

In this chapter, an overview and several theoretical details associated with DCNNs, is provided. Specifically, it formally introduces several common DCNN layers including the convolutional, pooling and Dropout layers, together with other architectural considerations such as their activation functions. Thereafter, it introduces learning in DCNNs, by elaborating on a popular loss function and several modern gradient based learning approaches, in addition to the traditional gradient decent. Finally, a motivation for weight initialization for deep models is provided, and this is followed by an introduction to several modern weight initialization schemes.

2.2 Overview of CNN architecture

CNNs are feed forward networks in that information flow takes place in one direction only (from their inputs to their outputs). Like artificial neural networks (ANN) are biologically inspired, so are CNNs. The visual cortex in the brain, which consists of alternating layers of simple and complex cells (Hubel and Wiesel, 1959, 1962), motivates their architecture. CNN architectures come in several variations; however, in general, they consist of convolutional and pooling (or subsampling) layers, which are grouped into modules. Either one or more fully connected layers, as in a standard feed-forward neural network, follow these modules. Modules are often stacked on top of each other to form a deep model. Figure 2.1 illustrates typical CNN architecture for a toy image classification task. An image is input directly to the network and this is followed by several stages of convolution and pooling. Thereafter, representations from these operations feed one or more fully connected layers. Finally, the last fully connected layer, outputs the class label. Despite this being, the most popular base architecture found in the literature, several architectural changes have been proposed in recent years with the objective of improving image classification accuracy or reducing computation costs (see Rawat and Wang, 2017 for further details).

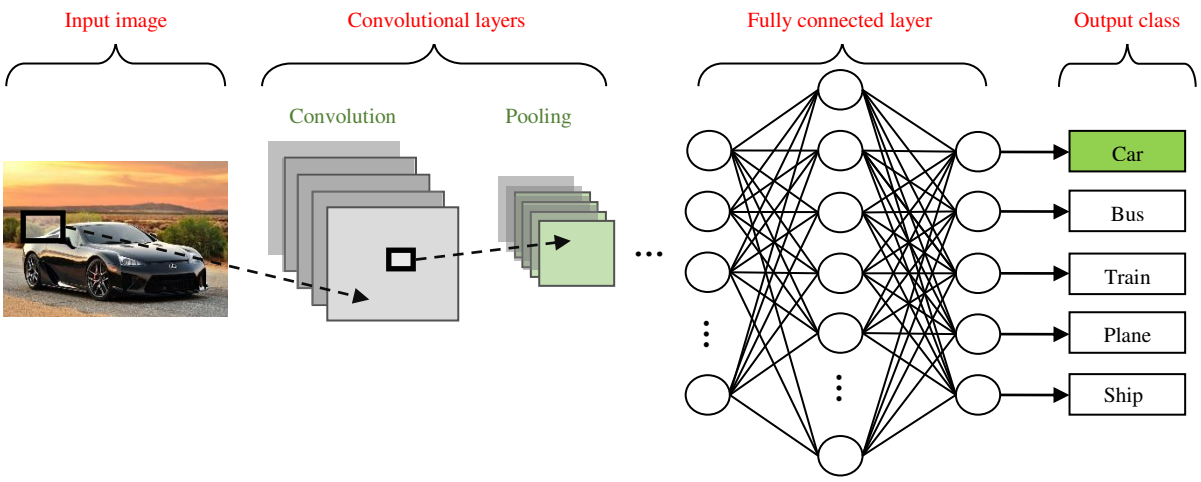


Figure 2.1: CNN image classification pipeline

2.2.1 Convolutional layers

The convolutional layers serve as feature extractors and thus they learn the feature representations of their input images. The neurons (or filters) in the convolutional layers are arranged into feature maps. Each neuron in a feature map has a receptive field, which is connected to a neighborhood of neurons in the previous layer via a set of trainable weights, sometimes referred to as a filter bank (LeCun et al., 2015). Inputs are convolved with the learned weights in order to compute a new feature map and the convolved results are sent through a nonlinear activation function. All neurons within a feature map have weights that are constrained to be equal; however, different feature maps within the same convolutional layer have different weights so that several features can be extracted at each location (LeCun et al., 1998; LeCun et al., 2015). More formally, the kth output feature map $\mathcal{Y}_k$ can be computed as:

$$\mathcal{Y}_k = f(\mathcal{W}_k * x) \tag{2.1}$$

where the input image is denoted by x, the convolutional filter related to the kth feature map is denoted by W_k, the multiplication sign in this context refers to the 2D convolutional operator, which is used to calculate the inner product of the filter model at each location of the input image and $f(\cdot)$ represents the nonlinear activation function (Yu, Wang, Chen and Wei, 2014). Nonlinear activation functions allow for the extraction of nonlinear features, and whilst traditionally, the sigmoid and hyperbolic tangent functions were used; recently, ReLUs (Nair and Hinton, 2010) have become popular (LeCun et al., 2015). The choice of activation function affects network training time and this has a significant influence on the performance of large DCNNs on large datasets (Krizhevsky et al., 2012). With this in mind, the ReLU

activation function and two of its latest variations are considered in this study; thus, in the subsections that follow, they are formally introduced and elaborated on.

2.2.1.1 Activation functions

2.2.1.1.1 ReLU activations

Traditional activation functions, such as the sigmoid or hyperbolic tangent are given by $f(x) = 1 / (1 + e^{-x})$ and $f(x) = tanh(x)$ respectively, where f is the neurons output as a function of its input x (the same notation is used for the remainder of the activation functions that follow). On the other hand the ReLU (Nair and Hinton, 2010), which is a piecewise linear function, has the simplified form $f(x) = max(x, 0)$. The ReLU only retains the positive part of the activation, by reducing the negative part to zero, whilst the integrated maximum operator promotes faster computation. The ReLU has been used in several state-of-the-art image classification systems (Lin et al., 2013; Gong et al., 2014; Zeiler and Fergus, 2014; Simonyan and Zisserman, 2014; Szegedy, Vanhoucke et al., 2015, Szegedy, Liu et al., 2015) and an in depth discussion and further motivations on them, can be found in the work presented by Glorot, Bordes and Bengio (2011).

2.2.1.1.2 ELU activations

Whilst ReLUs (Nair and Hinton, 2010), and other variants such as leaky ReLUs (Maas, Hannun and Ng, 2013) and parametric ReLUs (He et al., 2015a) are all non-saturating and thus lessen the vanishing gradient problem (Bengio, Simard and Frasconi, 1994), only ReLUs ensure a noise-robust deactivation state (Nair and Hinton, 2010; Clevert, Unterthiner and Hochreiter, 2016). However, they are non-negative and thus have a mean activation larger than zero. To deal with this, Clevert et al. (2016) proposed the Exponential Linear Unit (ELU), which has negative values to allow for activations near zero, but also saturates to a negative value with smaller arguments. Since the saturation decreases the variation of the units when deactivated, the precise deactivation argument becomes less relevant, thereby making ELUs robust against noise. Formally:

$$f(x) = max(x, 0) + \min(\vartheta(e^{x-1}), 0) \tag{2.2}$$

where ϑ is a predetermined parameter that controls the amount an ELU will saturate for negative inputs. ELUs speed up DCNN learning and lead to higher classification accuracy when compared to other activation functions such as ReLUs, and were thus considered for the current study.

2.2.1.1.3 SELU activations

Whilst Batch normalization (Ioffe and Szegedy, 2015) has developed as a robust standard to normalize DNN filter activations, training with normalization imposes the estimation of normalization parameters, and furthermore, it perturbs traditional optimization techniques such as stochastic gradient decent (SGD - Bottou, 1998, 2010) and popular regularization methods such as Dropout (Hinton et al., 2012), both of which results in high variance in the training error. This led to the very recent development of Self-normalizing neural networks (SNN), which have Scaled Exponential Linear Units (SELU's) at their core (Klambauer, Unterthiner, Mayr and Hochreiter, 2017). SELU's can avert the vanishing / exploding gradient problem (Bengio et al., 1994) by inducing self-normalizing variance stabilization characteristics on the SNN's that utilize them. More formerly, the SELU activation function is defined by:

$$f(x) = \lambda \begin{cases} x & if x > 0 \\ \alpha e^x - \alpha & if\ x \leq 0 \end{cases} \tag{2.3}$$

where α and λ denote fixed scaling factors that are not backpropagated through the network (see Klambauer et al., 2017 for the theoretical derivation). Furthermore, in addition to the SELU activation function, for SNNs to work, Klambauer et al. (2017) proposed a custom weight initialization scheme and a Dropout variant known as Alpha Dropout (see Section 2.2.4.2). Given the through theoretical analysis and promising empirical results, motivating for these freshly proposed activation functions, they are experimented with and compared to ReLU's (Nair and Hinton, 2010) and ELU's (Clevert et al., 2016), in Chapter 5.

2.2.2 Pooling layers

After the convolutional layers, the pooling layers are perhaps the most important (Rawat and Wang, 2017). They recapitulate the responses of neighboring neurons from the same kernel map and thus reduce the dimensions of their input representations. Significantly, they provide DCNNs with their spatial invariance to input distortions and translations (Krizhevsky et al., 2012; LeCun et al., 2015). Initially, it was common practice to use average pooling aggregation layers to propagate the average of all the input values, of a small neighborhood of an image, to the next layer (LeCun et al., 1989a, 1989b; LeCun et al., 1998). However, in modern models (Krizhevsky et al., 2012; Simonyan and Zisserman, 2014; Zeiler and Fergus, 2014; Szegedy, Liu et al., 2015), max pooling aggregation layers propagate the maximum

value within a receptive field to the next layer (Ranzato et al., 2007). Formally, max pooling selects the largest element within each receptive field such that:

$$y_{kij} = \max_{(p,q)\epsilon\Re_{ij}} x_{kpq} \tag{2.4}$$

where the output of the pooling operation, associated with the kth feature map, is denoted by y_{kij}, x_{kpq} denotes the element at location (p, q) contained by the pooling region $\Re_{ij}$, which embodies a receptive field around the position (i, j) (Yu et al., 2014). Figure 2.2 illustrates the difference between max pooling and average pooling. Given an input image of size 4x4, if a 2x2 filter and stride of two is applied, max pooling outputs the maximum value of each 2x2 region, whilst average pooling outputs the average rounded integer value of each subsampled region.

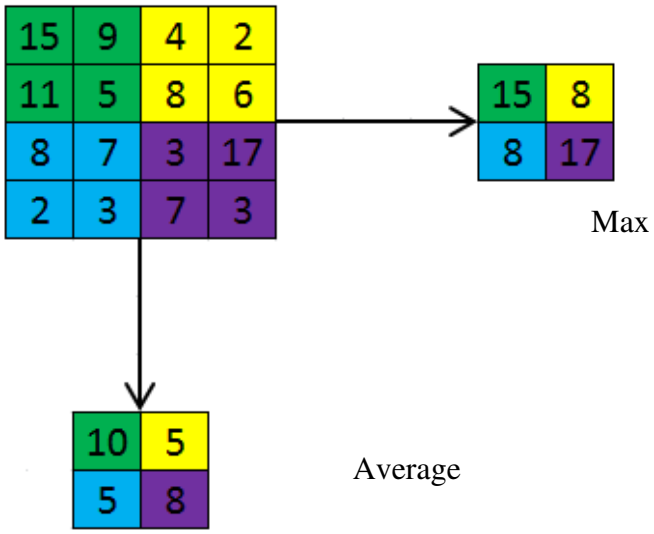


Figure 2.2: Average vs. max pooling

2.2.3 Fully connected layers

Several convolutional and pooling layers are usually stacked on top of each other to extract more abstract feature representations as you move through the network. The fully connected layers that follow these layers interpret these feature representations and perform the function of high level reasoning (Hinton et al., 2012; Simonyan and Zisserman, 2014; Zeiler and Fergus, 2014). For classification problems, it is standard to use the softmax operator (see Section 2.3.2) on top of a DCNN (Krizhevsky et al., 2012; Lin et al., 2013; Simonyan and Zisserman, 2014; Zeiler and Fergus, 2014; Szegedy, Liu et al., 2015). Whilst early success was enjoyed by using radial basis functions (RBFs), as the classifier on top of the convolutional towers (LeCun et al., 1998), Tang (2013) found that replacing the softmax

operator with a support vector machine (SVM), leads to improved classification accuracy. Moreover, given that computation in the fully connected layers is often challenged by their compute-to-data ratio, a global average-pooling layer, which feeds into a simple linear classifier, can be used as an alternative (Lin et al. 2013). In this work, the traditional fully connected scheme (LeCun et al., 1998) is adapted, however the number of filters utilized in each fully connected layer is optimized for enhanced performance (see Sections 5.3.3 and 5.3.4).

2.2.4 Dropout layers

A commonly experienced problem with training CNNs, and in particular DCNNs, is overfitting, which is poor performance on a held-out test set after the network is trained on a small or even large training set. This affects the models ability to generalize on unseen data, and is a major challenge for DCNNs; however, overfitting can be assuaged by regularization. Although the easiest and most common method to reduce overfitting is data augmentation[2] (LeCun et al., 1998; Simard et al., 2003; Ciresan et al., 2012; Krizhevsky et al., 2012; Montavon, Orr and Muller, 2012; Chatfield, Simonyan, Vedaldi and Zisserman, 2014), it requires a large memory footprint and comes at a high computational cost (Szegedy, Liu et al., 2015). This has led to the development of other regularization methods such as L_1 and L_2 regularization, stopping training early, soft-weight sharing (Nowlan and Hinton, 1992), and more recently the successful addition of Dropout (Hinton et al., 2012, Srivastava, Hinton, Krizhevsky, Sutskever and Salakhutdinov, 2014) to DCNN architecture[3] (Krizhevsky et al., 2012). In the following subsections, a formal description of Dropout is provided, and this is supplemented by a discussion on a recent variation of the technique since both the original technique and the variation are used in the current study (see Section 5.3.4).

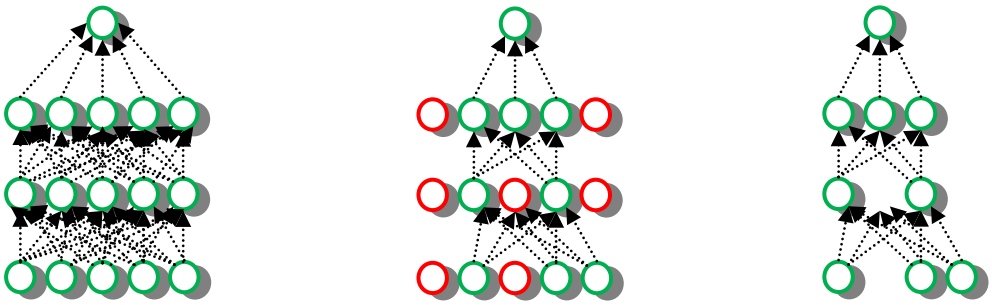
2.2.4.1 Formal introduction to Dropout

In Dropout (Hinton et al., 2012, Srivastava et al., 2014), each unit of a layers output is retained with probability p, else, it is set to zero with probability $1-p$, with 0.5 being a common value of p (Krizhevsky et al., 2012, Hinton et al., 2012). When Dropout is applied to a fully connected layer of a DCNN (or any DNN), the output of the layer $r = [r_1, r_2, \dots, r_d]^T$, can be expressed as:

[2] Data augmentation is a procedure to artificially enlarge a dataset
[3] Dropout regularization can be incorporated into DCNN architecture as additional layers prioir to the fully connected layers

$$r = m \star a(W_v) \tag{2.5}$$

where $\star$ denotes the element wise product between a binary mask vector m, and the matrix product between the input vector $v = [v_1, v_{2,} \dots, v_n]^T$ and the weight matrix W (with dimensions $d \times n$), followed by a nonlinear activation function, a. In Eqn. 2.5, the binary mask vector has size d and each element $j,$ is drawn independently from a $Bernoulli(p)$ distribution $m_j,$ whilst the biases are included in W and fixed to one for simplicity (Wan et al., 2013). The primarily benefit of Dropout is its proven ability to significantly reduce overfitting by effectively preventing feature co-adaptation (Hinton et al., 2012). Furthermore, a suitable way to regularize a DCNN is to average the results from several different networks, however, for large models, the computational resources required to do this will be astronomical. Dropout on the other hand provides a means to roughly merge an exponential number of DCNNs, in an effective manner and is thus capable of attaining model averaging (Hinton et al., 2012, Goodfellow, Warde-Farley, Mirza, Courville and Bengio, 2013; Srivastava et al., 2014), without the computational burden of traditional averaging methods. The technique is illustrated in Figure 2.3, which shows a network without Dropout, and the transitional and overall effect of the technique.



a) Fully connected network c) Nodes to be dropped c) Effective Dropout network

Figure 2.3: Difference between fully connected network layers without Dropout (a) and with Dropout (b and c)

2.2.4.2 Alpha Dropout

Dropout is well suited to ReLU activations, since it sets it's activations with probability $1 - p$ to zero, which relates to the default value for this activation and lies in an area of low variance. However, for SELU's (see Section 2.2.1.1.3) on the other hand, the low variance and normal activation value is expressed by:

$$\lim_{x \to -\infty} selu(x) = -\lambda\alpha = \alpha' \tag{2.6}$$

where λ and α are scaling factors of the related SELU activation function (see Section 2.2.1.1.3). Thus, rather than setting activations to zero, Klambauer et al. (2017) proposed a novel Dropout variant called Alpha Dropout that stochastically sets the activations to α', thereby making it well suited to SELU activations, since it stochastically sets the inputs to the negative saturation value. To ensure the self-normalizing property of Alpha Dropout, a parameterized affine transformation is applied (see Klambauer et al., 2017 for further details), and this maintains the means and variances even after the application of the technique. Initial empirical work shows that Dropout rates of $1 - p = 0.05$ and $1 - p = 0.10$ can result in suitable performance, however, this is optimized in Section 5.3.4.

2.2.4.3 Discussion

Further to the traditional and Alpha Dropout techniques, various other improvements and Dropout variants have been proposed in recent times. These are concisely surveyed in Rawat and Wang (2017). On the theoretical side, Wager, Wang and Liang (2013) highlighted its adaptive regularization characteristics, its efficiency and ensemble learning characteristics were examined by Warde-Farley, Goodfellow, Courville and Bengio (2013), whilst Baldi and Sadowski (2014) provided a detailed mathematical analysis of its static and dynamic properties, and characterized its averaging properties for DNNs by formal recursive equations. Thus, these works can be consulted for further details on the technique.

2.3 Network Training and Optimization

2.3.1 Introduction

CNNs, and ANNs in general, use learning algorithms to adjust their free parameters (i.e. the biases and weights) in order to attain the desired network output. The most common algorithm used for this purpose is backpropagation (LeCun, 1989; LeCun et al., 1998; Bengio, 2009; Deng and Yu, 2014; Deng, 2014). Backpropagation computes the gradient of an objective (also referred to as a cost / loss / performance) function to determine how to adjust a networks parameters in order to minimize errors that affect performance. In the subsections that follow, a commonly utilized loss function and several modern optimization techniques that are used to minimize this loss, all of which are optimized in Section 6.2.4 and 6.2.5 by the proposed methods, are briefly introduced.

2.3.2 The Softmax loss

The most common classification loss for DCNNs is the softmax loss (Krizhevsky et al., 2012; Lin et al., 2013; Goodfellow et al., 2013; Zeiler and Fergus, 2013, 2014; Chatfield et al., 2014; Simonyan and Zisserman, 2014; Szegedy, Liu et al., 2015; Szegedy, Vanhoucke et al., 2015, He et al., 2015a, 2015b). It consists of the softmax activation function, which is widely used in DCNNs, owing to its simplicity and probabilistic interpretation. Combining this activation function with the cross-entropy loss (or multinomial logistic regression) in the last fully connected layer of a convolutional model, results in the formation of the softmax loss. Formally, for the $i-th$ input feature x_i that has a corresponding integer label y_i, the softmax activation function is defined as the unnormalized log probabilities of the classes k, such that $P(Y=k|X=x_i)=\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$. Thus, the softmax loss can be written as:

$$L=\frac{1}{N}\sum_i L_i=\frac{1}{N}\sum_i\left[-\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)\right] \tag{2.7}$$

where the $j-th$ element ($j \epsilon [1,K], K$ is the number of classes) of the vector of class scores f, is represented by f_j, and N is the amount of training data. For this loss, f is typically the activations of a fully connected layer W, thus f_{y_i}, can be denoted as $f_{y_i}=W_{y_i}^T x_i$ in which W_{y_i} is the y_i-th column of W (Liu, Wen, Yu and Yang, 2016).

2.3.3 Gradient based learning

The softmax loss is usually minimized using some form of SGD (Bottou, 1998, 2010), during which the gradient is evaluated using the backpropagation algorithm. Whilst gradient decent, made popular by Rumelhart, Hinton and Williams (1986), was used in many of the early CNNs (LeCun et al., 1989a, 1989b; LeCun et al., 1998), increases in data size and its related computational complexity, have led to the popularization of SGD, which is a tremendous simplification of the traditional method (Bottou, 2010). In the subsections that follow these and other recent optimization advances, which are considered during the experimentation component of this dissertation, are briefly introduced.

2.3.3.1 Gradient decent

Consider a loss function $\mathcal{L}(\theta)$, parameterized by $\theta \in \mathbb{R}^d$, which denotes the model's parameters. Gradient decent provides a means to optimize (or minimize) $\mathcal{L}(\theta)$ by considering

the slope (or gradient) of $\mathcal{L}(\theta)$ with respect to θ, denoted by $\nabla_\theta \mathcal{L}(\theta)$, by updating θ in the direction converse to the gradient of $\mathcal{L}(\theta)$. Traditional gradient decent also referred to as batch gradient decent, computes the gradient of $\mathcal{L}(\theta)$, in relation to θ, for the complete training dataset, expressed formerly as:

$$\theta = \theta - \eta \cdot \nabla_\theta \mathcal{L}(\theta) \tag{2.8}$$

where the size of the steps taken to attain the local optimum, or more specifically the local minimum, is governed by the learning rate denoted by η. For modern datasets, which are too large to fit into memory, traditional gradient decent is too slow since the gradients for the entire dataset needs to be computed to perform just one update. Furthermore, it has a dependence on the observations of past iterations, and this doesn't allow it to facilitate online model updates (processing examples on the fly), thus making it intractable for modern applications. Despite this, gradient descent assures convergence to the global minimum and local minimum for convex and non-convex error surfaces, respectively.

2.3.3.2 Stochastic gradient decent

Instead of precisely computing the gradient $\nabla_\theta \mathcal{L}(\theta)$, a single[4] randomly selected sample $x^{(i)}$, with corresponding label $y^{(i)}$, is used to estimate it for each iteration, thereby making the process naturally stochastic, resulting in the technique being called (SGD - Bottou 1998, 2010). SGD can be expressed formerly as:

$$\theta = \theta - \eta \cdot \nabla_\theta \mathcal{L}(\theta;,\ x^{(i)}\ ;\ y^{(i)}\ ) \tag{2.9}$$

Divergently from traditional gradient decent, SGD relaxes the need for redundant computations, since for similar examples of a particular dataset, it re-computes the gradients prior to the parameter update. This makes it more efficient and faster, and allows it to process examples online (or on the fly), since it does not need to recall which examples were observed in past iterations (Choromanska, Henaff, Mathieu, Arous and LeCun, 2015). Furthermore, standard SGD can be implemented in parallel, across multiple GPUs, for further optimization and improved processing speeds, particularly for large scale machine learning applications (Zinkevich, Weimer, Li and Smola, 2010; Recht, Re, Wright and Niu, 2011; Dean et al., 2012; Zhuang, Chin, Juan and Lin, 2013; Bengio, 2013; Paine, Jin, Yang, Lin and Huang,

[4] In practice a mini batch of samples

2013). Despite the effectiveness of SGD, it is unable to update individual parameters, which is essential to perform updates of different scales, depending on parameter importance.

2.3.3.3 Adagrad

To compensate for this, Adagrad (Duchi, Hazan and Singer, 2011), which acclimates the learning rate η to the parameters, was proposed. This adaptation facilities performing minor (or smaller) updates for the frequently used parameters, and major (or larger) updates for their infrequent counterparts. Whilst the previous optimization methods update every parameter θ_i using the same learning rate η, Adagrad adapts it based on the results computed for past gradients for θ_i. Adagrad's update rule can be expressed formerly as:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \Big/ \sqrt{G_{t,ii} + \mathcal{E}} \cdot g_{t,i} \tag{2.10}$$

where $g_{t,i}$ denotes the per-parameter update, at every time step t, and $G_t \in \mathbb{R}^{dxd}$ is a diagonal matrix, with diagonal elements i, i, which denotes the sum of the gradients squared, in relation to θ_i, up to t, and $\mathcal{E}$ represents a smoothing term that prevents division by zero (Ruder, 2017). Adagrad has been shown to significantly improve the robustness of SGD, when used for large models (Dean et al., 2012), and furthermore, its adaptive scaling properties make it ideally suited for sparse data based applications. The downside of the technique is a belligerent, monotonically decreasing learning rate (Ruder, 2017).

2.3.3.4 Adadelta and RMSprop

In order to mitigate the drastically dimensioning learning rate of the Adagrad optimization technique, the RMSprop[5] and Adadelta (Zeiler, 2012) adaptive learning rate schedulers were independently proposed. Since RMSprop and the first update vector of Adadelta are identical, we experiment only with RMSprop in the simulations (see Sections 6.2.4 and 6.2.5); thus, only this technique is elaborated on here. The update rule for RMSprop can be formerly expressed as:

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2 \tag{2.11}$$

$$\theta_{t+1} = \theta_t - \eta \Big/ \sqrt{E[g^2]_t + \mathcal{E}} \cdot g_t \tag{2.12}$$

[5] Unpublished, proposed by Hinton, available from: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

where $E[g^2]_t$ is the running average of the gradients, at time step t, and the learning rate η is recommended as 0.001, for acceptable results. RMSprop and Adadelta both divide, the learning rate, by an exponentially decaying average of the gradients squared (Ruder, 2017).

2.3.3.5 Adam and AdaMax

Whilst Adadelta and RMSprop both accumulate exponentially decreasing averages of the gradients squared, they don't accumulate past gradients. To assure this, another per-parameter adaptive learning rate scheduler, called Adaptive Moment Estimation (Adam) was proposed (Kingma and Ba, 2014). Expressed formerly, Adam accumulates an exponentially decaying average of past gradients, such that:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2.13}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) {g_t}^2 \tag{2.14}$$

where m_t denotes the mean (i.e. the estimate of the first moment), and un-centered variance (i.e. the estimate of the second moment) of the gradients. Similar to Adadelta and RMSprop, the parameters are updated via the following rule:

$$\theta_{t+1} = \theta_t - {\eta}/{\sqrt{\widehat{v_t}} + \mathcal{E}} \cdot \widehat{m}_t \tag{2.15}$$

where $\widehat{m}_t = m_t / 1 - \beta_1^t$ and $\widehat{v_t} = v_t / 1 - \beta_2^t$ denote bias corrected (away from zero) moment estimates. The Adam optimizer was also generalized to a variant called AdaMax, which utilized the infinity norm (L_∞) for numerical stability, rather than the L_2 norm utilized by Adam. Expressed formerly, AdaMax's infinity-constrained second moment is:

$$\acute{v}_t = \beta_2^\infty v_{t-1} + (1 - \beta_2^\infty) |g_t|^\infty = \max(\beta_2 v_{t-1}, |g_t|) \tag{2.16}$$

Substituting $\sqrt{\widehat{v_t}} + \mathcal{E}$ with $\acute{v}_t$ in Eqn. 2.15 (Adam's update rule), we have the following update rule for the AdaMax optimizer:

$$\theta_{t+1} = \theta_t - {\eta}/{\acute{v}_t} \cdot \widehat{m}_t \tag{2.17}$$

where $\acute{v}_t$ relies on the max operator and thus does not incline towards zero biases like with Eqns. 2.13 and 2.14 of Adam. Therefore, there is no need to compute a bias correction for

Eqn. 2.16. Intuitively, Adam can be perceived as a combination of Momentum[6] (Qian, 1999) and RMSprop, where the Momentum *attribute* contributes to the exponentially decaying average of the past gradients, and the RMSprop *attribute* facilitates the computation of the past gradients squared (Ruder, 2017).

2.3.3.6 Nadam

Further to AdaMax, Adam was also extended to a Nesterov-accelerated Adaptive Moment Estimation (Nadam – Dozat, 2016) optimizer, which combines Adam and Nesterov accelerated gradient (Nesterov, 1983), the former of which is an improvement to the traditional Momentum technique. Expressed formerly, the Nadam update rule is:

$$\theta_{t+1} = \theta_t - {\eta}\Big/{\sqrt{\widehat{v_t}} + \mathcal{E}} \cdot \beta_1 \widehat{m}_t + {(1-\beta_1) g_t}\Big/{1-\beta_1^t} \tag{2.18}$$

where $\widehat{m}_t$ is the bias-corrected estimate of the momentum vector (see Dozat, 2016, for derivation).

2.3.3.7 Discussion

In this subsection, the optimization techniques that will be utilized in the simulations (see Sections 6.2.4 and 6.2.5) were briefly introduced, whilst Ruder (2017), can be referred to for further details. A detailed analysis on SGD is presented in Bottou (1998, 2010), whilst several alternatives are compared and surveyed in Sutskever, Martens, Dahl and Hinton (2013); which can be referenced for further examination.

2.4 Initialization

DCNNs are generally difficult to train due to their inherently large number of parameters (weights and biases), and the fact that their loss functions are non-convex (Choromanska et al., 2015). They are further affected by the vanishing gradient problem, in which the lower layers have gradients near zero because higher layers are almost saturated (Bengio et al., 1994). To alleviate these concerns and promote rapid convergence, initialization of their weight matrices is extremely momentous (Sutskever et al., 2013; Mishkin and Matas, 2016). Whilst several weight initialization schemes have been proposed (see Section 2.4.5), only those considered during simulations are briefly introduced in the subsections that follow.

[6] This is a technique developed to accelerate SGD and dampen its known oscillations

2.4.1 Random initialization

Initializing all the weights to zero will mean that if every filter in the network computes the same output, the same gradients will be computed during backpropagation and all the filters will receive the same update. Consequently, there will be no source of asymmetry between the different filters of the DCNN model (Karpathy, 2016). However it is still desirable to have the initial weights close to zero since it's conceivable that once the model is fully trained, approximately half the weights will be positive and the other half negative. Thus, a common symmetry breaking solution is to use small random numbers to initialize the weights, since filters with random, yet inimitable weights will compute unique updates and thus be able to integrate themselves as unique parts of the complete network. The random weights for this scheme can be drawn from either a normal or a uniform distribution, with a zero-mean Gaussian distribution, being a popular choice (Krizhevsky et al., 2012).

2.4.2 Glorot initialization

The problem with the previous randomly initialized filter scheme is that as the number of inputs grows, the variance of the output distribution will grow as well. Glorot and Bengio (2010) considered this and proposed a normalized initialization scheme that essentially adopted a balanced distribution for weight initialization (He et al., 2015a). For this initialization scheme, the initial weights are drawn from a uniform or normal distribution, with a zero-mean and precise variance. As recommended by Glorot and Bengio (2010), the following variance can be used:

$$Var(W\_Init) = \frac{2}{n_x + n_y} \tag{2.19}$$

where W_Init represents a specific neurons distribution at initialization, n_x is the number of neurons feeding into the variance, and n_y represents the number of neurons furnished by its output. Thus, for this technique, the number of input and output neurons controls the degree of initialization. This technique is often referred to *Xavier* initialization or *Glorot* initialization, the latter of which used in this dissertation.

2.4.3 He initialization

Glorot initialization promotes the propagation of signals deep into DNNs (DCNNs included), and has been shown to lead to substantively faster convergence (Glorot and Bengio, 2010).

However, its main limitation is that its derivation is based on the assumption that activations are linear, thus making it inappropriate for ReLU (Nair and Hinton, 2010) and PReLU activations (He et al., 2015a). To circumvent this, He et al. (2015a) derived a theoretically sound initialization that considered these nonlinear activations. Specifically, for this scheme the weights are initialized from a zero-mean truncated normal[7] or limited uniform distribution whose standard deviation is $\sqrt{2/n_l}$, where n is the number of connections of the response and l is the layer index. Furthermore, they initialize the biases to zero. Compared to *Glorot* initialization, this scheme is well suited to training extremely deep models, and is referred to as *He* initialization for the remainder of this study.

2.4.4 LeCun initialization

Another similar, yet much older initialization scheme which also draws the initial weights from a normal or uniform distribution, with zero mean, was proposed as far back as 1998 (LeCun, Bottou, Orr and Müller, 1998; Klambauer et al., 2017). However, instead of using a variance of $\sqrt{2/n_l}$, which was specifically implemented to counter the effects of the modern ReLU activation function (Nair and Hinton, 2010; Klambauer et al., 2017), a variance of $\sqrt{1/n_l}$ was utilized. Whilst this scheme was initially designed to work with the sigmoid activation function, it was recently also utilized successfully with the freshly proposed SELU activation (Klambauer et al., 2017), which is introduced in Section 2.2.1.1.3. Following Chollet et al. (2015), this initialization scheme is referred to as *Lecun* initialization for the remainder of this study.

2.4.5 Discussion

Like with the optimization techniques discussed in the previous section, in this section, the initialization methods that will be utilized in the simulations (see Sections 6.2.4 and 6.2.5) are briefly motivated for and introduced. Further details on these schemes can be found in the original works (LeCun et al., 1998; Glorio and Bengio, 2010; He et al., 2015a), whilst other initialization schemes are presented in Saxe, McClelland and Ganguli (2013), Sussillo and Abbott (2014), Hinton, Vinyals and Dean (2015), Romero et al., 2015 and Srivastava, Greff and Schmidhuber (2015a).

[7]A normal distribution is used in this study to facilitate equal comparison with the other schemes

2.5 Conclusion

This chapter briefly highlighted some of the fundamental aspects related to the basic building blocks of CNNs. It included discussions on its convolutional, pooling and Dropout layers, and this was followed by an introduction to the challenge of overfitting for DCNNs, and two related regularization techniques, namely Dropout and Alpha Dropout. Thereafter, gradient-based learning and several modern optimization techniques were introduced. Next, the complexity of training DCNNs and the related challenge of vanishing gradients were established, and this was pursued by discussions on several solutions in the form of weight initialization schemes. Whilst this chapter provided a concise introduction to several momentous mechanisms of modern DCNNs, a detailed analysis on these mechanisms were published in Rawat and Wang (2017), and thus excluded for brevity. In the next chapter, the motivation for using DCNNs for image classification is provided, together with discussions on several open challenges, which leads to an elaboration and formalization on the problem of DCNN model selection.

CHAPTER 3

Motivation for DCNNs and the problem of DCNN model selection

3.1 Chapter overview

This chapter motivates for the use of DCNNs for visual tasks, by covering their successes since their early development until their recent domination of image classification related applications. The remaining challenges preventing the large-scale deployment of our current models are also expounded on, with the intention of emphasizing the challenge of the hyperparameter search and computational problems imposed by the large search space and depth of our modern DCNNs. Thereafter, a formalism of the hyperparameter problem for the presented work is elaborated on, and this is followed by discussions on the details and challenges of the problem. Finally, current approaches to the challenge are introduced and the proposed methods located, before the chapter is concluded.

3.2 History of deep learning and CNNs

3.2.1 Early developments based on neuroscience

Biology has inspired several artificial intelligence techniques such as ANNs, evolutionary algorithms and cellular automata (Floreano and Mattiussi, 2008). However, perhaps the greatest success story amongst them, are CNNs (Goodfellow, Bengio and Courville, 2016). Their history began with the neurobiological experiments conducted by Hubel and Wiesel (Hubel and Wiesel, 1959, 1962) from as early as 1959. The main contribution of their work was the discovery that neurons in different stages of the visual system, responded strongly to specific stimulus patterns, whilst ignoring others. More specifically, they found that neurons in the early stages of the primary visual cortex responded strongly to precisely orientated patterns of light, such as bars, but ignored other more complex patterns of the input stimulus that resulted in strong responses from neurons in later stages. They also found that the visual cortex consisted of simple cells that had local receptive fields, and complex cells, which were invariant to shifted or distorted inputs, arranged in a hierarchical fashion. These works provided the early inspiration to model our automated vision systems based on characteristics of the central nervous system.

In 1979, a novel multi-layered neural network model, nicknamed the neocognitron, was proposed (Fukushima, 1979). Modelled based on the findings of Hubel and Wiesel (1959, 1962), it also consisted of simple and complex cells, cascaded together in a hierarchical

manner. With this architecture, the network proved successful at recognizing simple input patterns irrespective of a shift in the position or a considerable distortion in the shape of the input pattern (Fukushima, 1980; Fukushima and Miyake, 1982). Significantly, the neocognitron laid the groundwork for the development of CNNs. In fact, CNNs were derived from the neocognitron, and hence they have a similar architecture (LeCun et al., 2015).

3.2.2 Brief history of backpropagation and the first CNNs

Backpropagation was derived in the 1960s, and in particular, S.E Dreyfus (1962) derived a simplified version of the algorithm that used the chain rule alone. Nevertheless, the early versions of backpropagation were inefficient since they backpropagated derivative information from one layer to the preceding layer without openly addressing direct links across layers. Furthermore, they did not consider potential efficiency gains due to network sparseness (Schmidhuber, 2015). The modern efficient form of the algorithm that addressed these issues was derived in 1970 (Linnainmaa, 1970), however, there was no mention of its use for ANNs. Preliminary discussions for its use for ANNs date back to 1974 (Werbos, 1974), however the first known application of efficient backpropagation, specifically for ANNs, was described in 1981 (Werbos, 1982), but this remained relatively unknown. Nevertheless, it was "significantly popularized" (Schmidhuber, 2015) due to a seminal paper in 1986, by D. E. Rumelhart et al. (1986), which demonstrated that by using the backpropagation learning algorithm, the internal hidden neurons of an ANN could be trained to represent important features of the task domain.

In 1989, LeCun et al. (1989a, 1989b) proposed the first multi-layered CNNs and successfully applied these large-scale networks, to real (hand written digits and zip codes) image classification problems. These initial CNNs were reminiscent of the neocognitron (Fukushima, 1979, 1980; Fukushima and Miyake, 1982). However, the key difference was that they were trained in a fully supervised fashion using backpropagation, which was in contrast to the unsupervised reinforcement scheme used by their predecessor. This allowed them to rely more profoundly on automatic learning rather than hand-designed pre-processing for feature extraction (LeCun et al., 1989a, 1989b; LeCun, 1989), which previously proved to be extremely challenging, and hence they form an essential component of many modern, competition-winning, DCNNs (Krizhevsky et al., 2012; (Simonyan and Zisserman, 2014; Zeiler and Fergus, 2014; Szegedy, Liu et al., 2015).

3.2.3 Introduction of the LeNet architecture and MNIST dataset

In 1998, the CNNs described earlier (LeCun et al., 1989a, 1989b), were improved upon and used for the task of individual character classification, in a document recognition application. This work was published in a detailed seminal paper (LeCun et al., 1998), which highlighted the main advantages of CNNs when compared to traditional ANNs: they require fewer free parameters (because of weight sharing), and they consider the spatial topology of the input data, thereby allowing them to deal with the variability of 2D shapes.

In addition to the proposed CNNs, which included the famous LeNet-5 model, LeCun et al. (1998) also introduced the popular Mixed National Institute of Standards and Technology (MNIST) dataset of 70000 handwritten digits, which has since become very popular for computer vision tasks. Further details on the MNIST dataset and LeNet-5 architecture can be found in Sections 5.3.2.2 and 5.3.3.1, respectively, which deals with their application and adaptation for the work presented in this dissertation.

3.2.4 Early CNN successes despite perceived issues with gradient decent

In the late 1990s and early 2000s, neural network research had diminished (Simard et al., 2003; LeCun et al., 2015), since they were hardly utilized for machine learning tasks, whilst computer vision and speech recognition tasks overlooked them. It was widely believed that learning useful multistage feature extractors, with little prior knowledge, was infeasible due to issues with the popular optimization algorithm, gradient decent. Specifically, it was thought that basic gradient decent would not recover from poor weight configurations that inhibited the reduction of the average backpropagated error, a phenomenon known as poor local minima (LeCun et al., 2015). In contrast, other statistical methods and in particular SVMs, became popular due to their successes (Decoste and Schölkopf, 2002).

Contrary to this trend, although their applications were very scattered, CNNs continued to produce encouraging results and some of their representative successes in this period were reported by Simard et al. (2003), LeCun, Huang and Bottou (2004), Chopra, Hadsell and LeCun (2005), Muller, Ben, Cosatto, Flepp and LeCun (2005), Huang and LeCun (2006), and Chellapilla, Shilman and Simard (2006).

3.2.5 The deep learning renaissance

The first feed forward multi-layered neural networks were trained in 1965 (Ivakhnenko and Lapa, 1965), and although they did not use backpropagation, they were perhaps the first deep learning systems (Schmidhuber, 2015). Although deep-learning-like algorithms have a long

history, the term deep learning became a catchphrase around 2006, when deep belief networks (DBNs) and autoencoders trained in an unsupervised fashion, were used to initialize DNNs, trained using backpropagation (Hinton et al., 2006; Hinton and Salakhutdinov, 2006; Bengio et al., 2006). Prior to this, it was taught that deep multi-layered networks (including DCNNs) were too hard to train, due to issues with gradient decent, and thus they were not popular (Bengio et al., 2006; Bengio, 2009; Deng and Yu, 2014; Schmidhuber, 2015; Goodfellow et al., 2016). Conversely, CNNs were a notable exception and proved easier to train when compared to fully connected networks (Simard et al., 2003, Bengio, 2009; LeCun et al., 2015; Goodfellow et al., 2016) - their early successes are discussed in the preceding sections.

However, since neural network research had slowed down in the late 1990s and early 2000s (Simard et al., 2003; LeCun et al., 2015), CNN development was also hindered but revived around 2006. Using an energy based model to extract sparse features, which has several applications, which include classification and segmentation, and then using the resultant output to initialize the first layer of a DCNN, Ranzato, Poultney, Chopra and LeCun (2006) slightly improved the previous best-reported classification result (Simard et al., 2003) on the MNIST dataset (LeCun et al., 1998). Citing Hinton et al. (2006), their DCNN model which had a similar architecture to LeCun et al. (1998), but used a considerably larger number of feature maps to produce sparse features, was pre-trained in an unsupervised fashion and consisted of an encoder-decoder system that initialized the first layer weights of the DCNN. This work was the first to use DCNNs initialized by unsupervised training techniques during the period of the deep learning renaissance (Hinton et al., 2006; Hinton and Salakhutdinov, 2006; Bengio et al., 2006), and led to several other unsupervised pre-training attempts between 2006 and 2011 (Ranzato et al., 2006; Ranzato et al., 2007; Lee, Grosse, Ranganath, and Ng, 2009; Jarrett, Kavukcuoglu and LeCun, 2009, LeCun, Kavukcuoglu, and Farabet, 2010). In addition to unsupervised pre-training, the deep learning renaissance was characterized by the introduction of GPU's for neural network (Oh and Jung, 2004; Steinkrau, Simard and Buck, 2005), and in particular DCNN computation[8] (Chellapilla et al., 2006). This was a significant advance since GPUs have become a momentous facet of most award winning or state-of-the-art DCNNs (Ciresan et al., 2011; Krizhevsky et al., 2012; Hinton et al., 2012; Zeiler and Fergus, 2013, 2014; Simonyan and Zisserman, 2014; Szegedy, Vanhoucke et al., 2015; He et al., 2015a).

Furthermore, another characteristic of the renaissance, was the advancement of previous algorithms, such average pooling (LeCun et al., 1989a, 1989b, LeCun et al., 1998), which was

[8] During program execution, convolution operations are computationally costly and thus make DCNNs significantly slower too evaluate when compared to standard ANNs of the same magnitude

succeeded by the introduction of max-pooling. Max-pooling was used in conjugation with backpropagation for the first time for a DCNN like architecture in 2007 (Ranzato et al., 2007). Thereafter, in 2010, Scherer, Müller and Behnke (2010) showed, empirically, that the max pooling operation was vastly superior for capturing invariance in image-like data and could lead to improved generalization and faster convergence when compared to a subsampling operation. Continuing with the empirical work, Jarrett et al. (2009) highlighted further advantages over average pooling, such as the relaxation of rectification layers for models that use the max-pooling approach. Since max pooling was only designed for feed forward networks, Lee et al. (2009) introduced and applied probabilistic max pooling to convolutional DBNs with the aim of scaling DBNs (Hinton et al., 2006) to full sized, high dimensional images, resulting in "translation invariant hierarchical generative models". Further motivation for the technique is provided in Section 5.3.3.1.

3.2.6 The changing point in the application of DCNNs for computer vision tasks

The deep learning renaissance of 2006 (Hinton et al., 2006; Hinton and Salakhutdinov, 2006; Bengio et al., 2006), spurred on several successful applications of DCNNs to a wide variety of tasks. These included image and object classification and recognition (Chellapilla et al., 2006; Ranzato et al., 2007; Weston, Ratle, Mobahi, Collobert, 2008; Jarrett et al., 2009; Lee et al., 2009; LeCun et al., 2010; Scherer et al., 2010; Boureau, Ponce and LeCun (2010); Masci, Meier, Ciresan and Schmidhuber, 2011), face detection (Nasse, Thurau and Fink, 2009) and image segmentation (Turaga et al,. 2010). Furthermore, they also found interesting applications in scene parsing (Farabet, Couprie, Najman and LeCun, 2012), vision for autonomous off-road driving (Hadsell et al., 2009), and hand gesture recognition (Nagi et al., 2011). However, despite these accomplishments, they were still largely discarded by the mainstream computer vision and machine learning communities (LeCun et al., 2015). This changed after the ILSVRC 2012 (Russakovsky et al., 2015), when a fully supervised DCNN achieved record-breaking classification results on a subset of the ImageNet dataset (Krizhevsky et al., 2012). This work revolutionized the field of computer vision, and as a result, DCNNs have since become the leading architecture for most visual tasks, and in particular, for image classification related applications.

3.3 The AlexNet architecture

Central to their success, they implemented several novel and unusual techniques, which included the use ReLU's (Nair and Hinton, 2010), over traditional sigmoid or hyperbolic

tangent activation functions (see Section 2.2.1.1), a parallel GPU configuration to mitiagte computation of their large model, which was inspired by Ciresan et al. (2012), and the application of local response normalization. Denoted mathematically, if a kernel i, at position $(x; y)$ is used to compute the activity of a neuron denoted by $a^i_{x,y}$, and the ReLU nonlinearity is than applied, the response-normalized activity $b^i_{x,y}$ can be expressed as:

$$\boldsymbol{b^i_{x,y}} = \boldsymbol{a^i_{x,y}} / (\boldsymbol{k} + \boldsymbol{\alpha} \sum_{\boldsymbol{j=max(0,i-n/2)}}^{\boldsymbol{min(N-1,i+n/2)}} (\boldsymbol{a^j_{x,y}})^{\boldsymbol{2}})^{\boldsymbol{\beta}} \quad (3.1)$$

where N is the total number of kernels in the layer and the sum runs over n "adjacent" kernel maps at the same spatial position. This scheme aided generalization and reduced their networks classification error rates. They further reduced the classification error by overlapping the networks max pooling layers. Figure 3.1 illustrates the revolutionary architecture presented by Krizhevsky et al. (2012), which consisted of five convolutional layers, three of which were followed by overlapping max pooling layers, and three fully connected layers. The various layer-parts in the top half of the figure ran on one GPU, whilst the layer-parts at the bottom ran on the second GPU, with the GPU's only interacting with each other at specific layers.

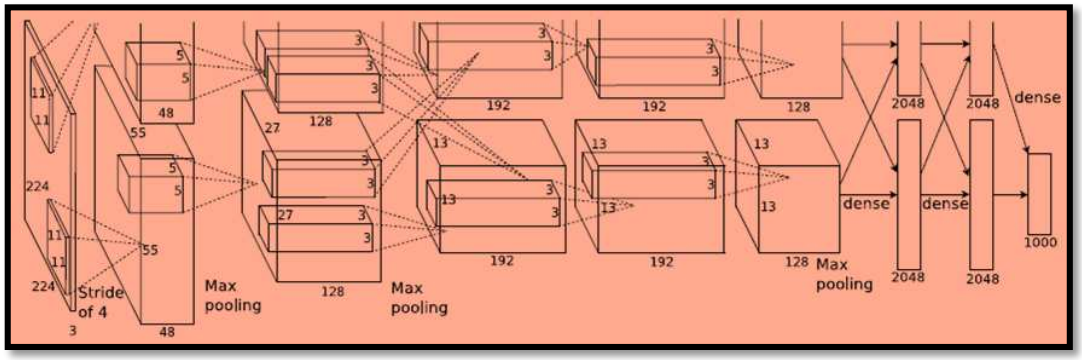


Figure 3.1: DCNN architecture split over two GPUs (Krizhevsky et al., 2012)

The model was regularized by Dropout (Hinton et al., 2012 - see Sections 2.2.4), and a specific form of data augmentation that applied translations and horizontal reflections to the data, and altered the intensities of its channels, by performing principal component analysis (PCA) on their pixel values, both of which led to improved classification performance. The model of Krizhevsky et al. (2012) has been used extensively for various purposes since its development. Numerous researches have used it to benchmark their models against, or as a base model to test new algorithms. Furthermore, their model has inspired several DCNN works and has become one of the major contributors to the recent rise in DCNN technology for image classification related applications.

3.4 Representative improvements exemplify DCNN dominance

The ground-breaking work of Krizhevsky et al. (2012), was followed by DCNN dominance in image classification tasks, as highlighted by their successes in subsequent ILSVRC's (Russakovsky et al., 2015). In an attempt to understand them and derive ways to improve their performance, Zeiler and Fergus (2014) introduced a new visualization technique, using a multi-layered deconvolutional network (Zeiler, Taylor and Fergus, 2011), that provided vision into the intermediate feature extraction layers of the network and they used this in a diagnostic role to improve the DCNN architecture and performance of Krizhevsky et al. (2012), winning the 2013 challenge.

In 2014, Szegedy, Liu et al. (2015) introduced the Inception DCNN architecture and a particular incarnation of it, called GoogLeNet, produced outstanding image classification and object detection results, winning both the ImageNet classification and detection challenges in 2014 (Russakovsky et al., 2015). Their success was brought about by using a very large network, consisting of twenty-two layers. Since the cost of this is a larger number of parameters, which makes the network more prone to overfitting, and a considerably larger computational burden, they used a carefully engineered design that used 1x1 convolutions heavily, which was inspired by Lin et al. (2013), to perform two functions. Most significantly, they served as dimension reduction blocks prior to the more computationally costly 3x3 and 5x5 convolutions and they included the use of rectified linear activations (Nair and Hinton, 2010), thus making them dual purpose. Subsequently, they were able to increase the depth and width of their network, whilst only marginally increasing the computational cost.

Similar to Szegedy, Liu et al. (2015), Simonyan and Zisserman (2014), the runners up in the same classification contest, ILSVRC 2014 (Russakovsky et al., 2015), also used a very DCNN, which consisted of nineteen layers compared to the twenty-two of their competitors. However, asserting that the Inception model was too complex, they kept all the parameters of their DCNN architecture constant, and steadily increased the depth alone. This was made feasible by using smaller sized convolutional (3x3) filters throughout the network, which was inspired by Ciresan et al. (2011), who already used smaller kernels, albeit for shallower networks applied to simpler tasks.

The winners of the ILSVRC 2015 (Russakovsky et al. 2015), He et al. (2015b) used an even deeper DCNN, when compared to their predecessors. In fact, their model was ultra-deep in that it consisted of 152 layers. Since deeper models are harder to train and suffer from

degradation[9] (of training and thus test accuracy) (He et al., 2015b; He and Sun, 2015; Srivastava et al., 2015), they introduced a new residual learning framework, in which layers learnt residual functions, with reference to their preceding layer inputs. This allowed errors to be propagated directly to the preceding units, and thus made these networks easier to optimize, and although they were ultra-deep, easier to train.

Notwithstanding degradation, deeper models are generally more accurate and thus produce better empirical results, however as depth increases so does computational costs. With this in mind, the representative work discussed here have led to several recent attempts to improve the classification accuracy of DCNNs by modifying their architecture for improved performance, without losing sight of the computational burden imposed on such models.

3.5 Recent DCNN advances

In addition to the revolutionary work of Krizhevsky et al. (2012) and the further symbolic improvements described in the preceding section (Simonyan and Zisserman, 2014; Zeiler and Fergus, 2014; Szegedy, Liu et al., 2015; He et al., 2015b), several other improvement attempts related to network architecture, nonlinear activation functions, supervision components, regularization mechanisms, optimization techniques and swifter processing of DCNNs has supplemented the advancement of DCNNs, and brought them to the forefront of the deep learning renaissance. A detailed survey of these advancements is provided in Rawat and Wang (2017), which was published in conjunction to the write up of this dissertation, and can be referred to for further details.

3.6 Open challenges

Despite the promising image classification results obtained by DCNNs, there are still challenges that need to be addressed and further to the key challenges described in the following sections, other known challenges include degradation (see Section 3.4), internal covariate shift[10] (Ioffe and Szegedy, 2015) and a deeper image understanding (Vinyals et al., 2015; Karpathy and Fei-Fei, 2016). Further details on these together with selected intrinsic trends from recent publications can be found in Rawat and Wang (2017).

[9] Degradation is caused by the poor propagation of activations and gradients because of stacking several nonlinear transformations on top of each other

[10] Internal covariate shift refers to changes to the distribution of each layer's inputs because of parameter changes in the previous layer

3.6.1 Theoretical proof

Despite the empirical successes of DCNNs, the theoretical proof of why they succeed is still lacking. To this end, Mallat (2012) proved translation invariance and deformation stability of the features extracted by scattering convolutional networks, whilst Wiatowski and Bölcskei (2015) persisted with the mathematical analysis of the features extracted by DCNNs and they theoretically established deformation stability and vertical translation invariance. Further theoretical analysis was conducted by Basu et al. (2016). Others have turned their attention to the internal operation and performance of DCNNs, such as the commonly cited feature visualization technique presented by Zeiler and Fergus (2014), supplemented by other visualization attempts by Girshick, Donahue, Darrell and Malik (2014) and Yu, Yang, Bai, Yao and Rui (2014a, b), which focused on understanding the internal mechanisms of our current models. Thus, further progress is dependent on both sound theoretical proof and practical investigations that lead to improved understanding and performance.

3.6.2 Geometric invariance

Although DCNNs are robust against small-scale deformations (Lee et al., 2009), their final representations are not geometrically invariant (Ciresan et al., 2011; Gong et al., 2014; Razavian, Azizpour, Sullivan, Carlsson, 2014). Specifically, they are sensitive to global translations, rotations and scaling (Gong et al., 2014). To address translation variances, Lee et al. (2009) proposed probabilistic max pooling, whilst the multi-scale ordeless pooling (MOP) scheme of Gong et al. (2014) was shown to be robust against several geometric variances. Recently, the transformation invariant pooling (TI-pooling) scheme presented by Laptev, Savinov, Buhmann and Pollefeys (2016) efficiently handled rotations and scale changes, and thus built transformation invariance into DCNN architecture, whilst the spatial transformer module proposed by Jaderberg, Simonyan and Zisserman (2015) learned translation, scale, rotation and warping invariance. With this regard, interesting future directions include investigating if further fundamental changes to DCNN architectures are required to improve their universal robustness, supplemented by investigations into novel datasets that facilitate the training of more robust models (Rawat and Wang, 2017).

3.6.3 Adversarial examples

The intriguing discovery of adversarial examples (Szegedy et al., 2014), is another open and fascinating challenge for DCNNs and classifiers in general. Adversarial examples are small, yet intentional, perturbations applied to images with the aim of misleading or fooling the

recognition / classification system. When these perturbations are used to alter an image, humans are easily able to classify the image correctly (Goodfellow, Shlens and Szegedy, 2015; Uličný, Lundstrom and Byttner, 2016), whilst classifiers see the image as being from a different class.

So far the most promising attempts to solve this issue focus on different training techniques such as adversarial training (Goodfellow et al. 2015) and distillation (Papernot, McDaniel, Wu, Jha and Swami, 2016), generative pre-processing methods such as the use of denoising autoencoders (Uličný et al., 2016), and changing DCNN architecture to make it more non-linear or to penalize unusual signals (Zhao and Griffin, 2016; Jin, Dundar, and Culurciello 2016).

Since these adversarial (Szegedy et al., 2014) images highlight a vast gap between the vision capabilities of humans and computer vision systems, finding these intriguing properties have brought about several questions regarding the generalisation, function approximation and security features of deep networks (Szegedy et al., 2014; Goodfellow et al., 2015; Gu and Rigazio, 2015; Zhao and Griffin, 2016; Uličný et al., 2016; Jin et al., 2016; Papernot et al., 2016). This has opened up a completely new area of research focusing on the generation of these images and the design of systems, and in particular DCNNs that are robust against them.

3.6.4 *Unsupervised learning*

Despite the contribution of unsupervised pre-training to the deep learning renaissance (Hinton et al., 2006; Hinton and Salakhutdinov, 2006; Bengio et al., 2006), current DCNNs mostly utilize the supervised learning paradigm and thus they are not able to exploit the massive amounts of unlabelled data available on the internet, stored in cloud based systems or even captured by mobile devices. Furthermore, human learning is naturally unsupervised (LeCun et al., 2015), and thus it is expected that future DCNN models will attempt to mimic nature more than our current models do (Rawat and Wang, 2017). Recent attempts along these lines include works by Goodfellow et al. (2014) Kingma and Welling (2014), Bengio, Thibodeau-Laufer, Alain and Yosinski (2014), Kulkarni, Whitney, Kohli and Tenenbaum (2015) and more recently Bachman (2016), all of which utilize promising generative based modelling techniques.

3.6.5 Computational cost

It is well known that larger datasets have contributed to the successes of deep learning (Krizhevsky et al., 2012; Deng and Yu, 2014; Zeiler and Fergus, 2014). However, the

downside, particularly during training, is a greater computational burden. Coupled to this, is the fact that DCNN models have a tremendous amount of parameters (weights and biases), which has a negative effect on their storage and memory requirements (Krizhevsky et al., 2012; Wan et al., 2013; Simonyan and Zisserman, 2014; Szegedy, Vanhoucke et al., 2015; Szegedy, Liu et al., 2015; He et al., 2015b; Szegedy et al., 2016). Thus, a considerable amount of research has gone in to reduce the computational costs and storage space requirements of DCNNs. Recurrent computational cost reduction themes include parallel computing approaches, exploiting the convolution theorem and circular projections, and matrix manipulations. Furthermore, some of the latest advances include DCNN compression and weight quantization (Han et al., 2016), fast algorithms (Lavin and Gray, 2016), GPU clusters with truncated representations (Dettmers, 2016) and FPGA accelerated advances (Qiao et al., 2016). Both the recurrent themes and latest advances are surveyed in the associated survey paper, and be referred to for further details.

3.6.6 Model selection and hyperparameter optimization

Further to the computational cost of training individual DCNN models, and their large number of model parameters, tuning deep models, and in particular DCNNs, is further challenged by the requirement to tune its hyperparameters. Whilst the model parameters such as the filter weights are optimized internally through the selected optimization algorithm, which is more often than not SGD (Bottou, 1998, 2010), the models external settings also need to be tuned. The internal behavior of the model is controlled by the selection of these hyperparameters, and determining the correction combination is critical to the resultant accuracy. This challenge applies to machine leaning models in general, and selecting the appropriate algorithm and hyperparameter combinations can significantly affect the models performance, and in certain instances, cause accuracy changes from 1% to 95% (Thornton, Hutter, Hoos, Leyton-Brown, 2013). Furthermore, with specific regards to convolutional networks, the hyperparameter space is very large and this is compounded by the fact that deeper and wider models require a larger array of hyperparameters.

Hyperparameters can be of the discrete, continuous or categorical type. For DCNNs, typical discrete hyperparameters include the number and size of the convolutional filters, the number and size of the pooling filters, the Dropout rate if Dropout is used (Hinton et al., 2012), and the number of filters for the fully connected layers, whilst the type of activation function for each layer, is categorical in nature. These hyperparameters are related to the models architecture and are referred to as there architectural hyperparameters for the rest of this dissertation. The inherent association of these hyperparameters with the models

architecture means that their number will grow exponentially, as the architectural depth or width of the model is expanded. For example, if an additional convolutional layer is added to a model's architecture, then additional hyperparameters such as the number, size and stride of the convolutional filters will need to be tuned.

On the other hand, whilst the type of optimizer to use can also be regarded as a categorical learning hyperparameter, its associated learning rate is inherently continuous in nature. Other typical hyperparameters include the size of each batch of data fed to the network and the total number of epochs required to train the network (see Table 5.1 for a comprehensive example). Furthermore, specific optimizers have their own hyperparameters in addition to their learning rates that also require tuning. For example if SGD is used with momentum (Qian, 1999), the momentum rate and the learning rate decay also require tuning. These types of hyperparameters are related to the models learning mechanisms and are termed their learning parameters for the reminder of the dissertation. Dissimilar to the architectural hyperparameters, they are global in nature and thus are not directly affected as a models depth or width is increased. From the above, it can be concluded that for DCNN models, there are various aspects that require configuration, and these configurations can have fluctuating effects on the performance of the resulting model.

The question that follows is: how do we go about selecting the correct combination of hyperparameters for a specific task? The most obvious approach will be to search the entire space and consider all the possible combinations available (commonly referred to as grid search or brute-force approach – Pedregosa et al., 2011), however given the computational cost to validate a single combination for a deep model – see Section 3.6.5, this is not practicable and in certain instances, for example on large labelled datasets, not even conceivable. Another approach is to manually tune the models parameters using experience and rules of thumb (Hsu, Chang and Lin, 2003; Hinton, 2012; Bengio, 2012). However, these are subjective or arbitrary approaches and their reproducibility is often challenged. Furthermore, for models with a large number of hyperparameters, these approaches are not practical (Claesen, Simm, Popovic, Moreau, and DeMoor, 2014). This has led researches to consider other alternatives such as random search to mitigate computation and automated guided search techniques such as particle swarm optimization (PSO), GA's, Bayesian optimization and reinforcement learning based techniques. In the sections that follow, the hyperparameter search problem for DCNNs is formalized, and this is followed by an introduction to some of the challenges associated with the task. Thereafter, a brief synopsis of some of the current research into mitigating the problem for convolutional models, is

introduced, with the intention of positioning the search methods presented in Chapters 5 and 6.

3.7 Formalization of the architectural and learning hyperparameter search

The aim of using DCNNs for the image classification tasks described in this dissertation can be summarized as training a convolutional model $\mathbb{C}\mathcal{M}$, to minimize or maximize, on a known test dataset $\mathcal{D}^{(\mathrm{ts})}$, some predefined loss function denoted by $\mathcal{L}(\mathcal{D}^{(\mathrm{ts})}; \mathbb{C}\mathcal{M})$. Appropriate objective functions for the classification task include means squared error, error rate and classification accuracy, the latter of which is selected for the methods proposed in Chapter 5 (see Section 5.2.3.1). The maximization (since we dealing with accuracy) of the objective function results in finding the optimal parameters (example weights and biases) for the classification task. The $\mathbb{C}\mathcal{M}$ is itself parameterized by a set of architectural hyperparameters θ, such that $\mathbb{C}\mathcal{M}(\mathcal{D}^{(\mathrm{tr})}; \theta)$. A learning algorithm $\mathbb{L}\mathring{\mathrm{A}}$ is used to construct the convolutional model CM, through a supervised training procedure on the training dataset $\mathcal{D}^{(\mathrm{tr})}$. Typical DCNN architectural hyperparameters include the number of convolutional filters and their respective sizes, i.e. $\theta = [f_n, f_s]$. The $\mathbb{L}\mathring{\mathrm{A}}$ may also require parameterization by another set of parameters (referred to as the learning hyperparameters), denoted by λ, such that $CM = \mathbb{L}\mathring{\mathrm{A}}(\mathcal{D}^{(\mathrm{tr})}; \theta; \lambda)$. In this regard, representative learning hyperparameters include optimization learning rates and training batches sizes, i.e. $\lambda = [lr, bs]$.

Thus considering the architectural and learning space parameterization, the overall objective of the hyperparameter search over the combined space, will be to find a set of architectural and learning hyperparameters, denoted by $\theta^*$and λ^*respectively, that results in an optimal convolutional model $\mathbb{C}\mathcal{M}^*$, which maximizes the objective $\mathcal{L}(\mathcal{D}^{(\mathrm{ts})}; \mathbb{C}\mathcal{M})$. Using similar notation as Claesen et al. (2014), this can be expressed formerly as:

$$\theta^*, \lambda^* = arg\max_{\theta,\ \lambda} \mathcal{L}(\mathcal{D}^{(\mathrm{ts})}; \mathbb{L}\mathring{\mathrm{A}}\left(\mathcal{D}^{(\mathrm{tr})}\right); \theta,\ \lambda) = \arg\max_{\theta,\ \lambda} \mathcal{F}(\theta,\ \lambda;\ \mathbb{L}\mathring{\mathrm{A}}, \mathcal{D}^{(\mathrm{tr})}, \mathcal{D}^{(\mathrm{ts})}, \mathcal{L}) \quad (3.2)$$

where $\mathcal{F}$ represents tuples of architectural hyperparameters θ and learning hyperparameters λ, and returns an associated cost $\mathcal{L}$. For the architectural search presented in Chapter 5 and the learning parameter search presented in Chapter 6, a labelled academic dataset, split into $\mathcal{D}^{(\mathrm{tr})}$ and $\mathcal{D}^{(\mathrm{ts})}$ is used (see Section 5.3.2.2), and the classification accuracy, which represents the loss function $\mathcal{L}$ is computed through the softmax loss function (see Section 5.2.3.1) of the learning algorithm LA.

3.8 *Challenges in the optimization of hyperparameters*

As illustrated by Eqn. 3.2, the hyperparameter search for a particular problem is dependent on the learning algorithm $\mathbb{L}\text{Å}$, the selected loss function $\mathcal{L}$ and the dataset $\mathcal{D}^{(\mathrm{tr})}$ and $\mathcal{D}^{(\mathrm{ts})}$. In general, the hyperparameter search is tackled as a single-objective, non-differentiable, optimization problem, over a domain with different types of variables (discrete, categorical and continuous types are considered in this study). In the subsections that follow, some of the challenges in hyperparameter optimization, with regards to the DCNNs studied in this work, are briefly introduced (Claesen and Moor, 2015).

3.8.1 Costly individual model evaluations

Each evaluation of a DCNN model's accuracy objective requires validating the models performance after it has been trained on the training set, with a set of architectural and learning hyperparameters. The time to accomplish each evaluation is governed by the available computational resources, the nature of the learning algorithm and the size of the dataset. In particular, training large DCNNs on the ImageNet dataset (Russakovsky et al., 2015) can run into days, and in some cases even weeks (Krizhevsky et al., 2012; Simonyan and Zisserman, 2014). Furthermore, as illustrated in Chapter 5 and 6, training time is further dependent on the architectural and learning parameters selected. For instance training a model with fewer filters requires less computation time compared to training one that has more filters. The costly objective function evaluation challenge is exacerbated when the search space is large, and there are numerous different hyperparameter combinations, since each combination will require training and evaluating an individual model. Thus, it is of significant importance, especially with constrained hardware resources, to devise efficient ways to search the hyperparameter space, with reduced objective function evaluations.

3.8.2 Non-determinism

The non-determinism of the objective function is another challenge in hyperparameter optimization. For DCNNs, the stochasticity is induced by several random aspects of the training process and these include the weight initialization procedures, optimization techniques, and regularization mechanisms. At times, the non-determinism can be mitigated by using machine learning techniques, such as using ensembles (see Section 7.3, in recommendations), however these further add to the computational load of the system, since several individual evaluations are required. The intrinsic stochasticity of the training process, directly infers that after a hyperparameter search for a predefined set of evaluations, the best

set of architectural and learning hyperparameters selected by an intelligent search or even an exhaustive brute-force approach, may not be the true optimal set of hyperparameters. Thus, an efficient search procedure, should densely sample around the empirical optimum in order to ascertain if the selected best is an outlier or not. Furthermore, the procedure should also search other areas of the space to identify other potential optimal results (see analysis in Section 6.2.5.3, which highlights these attributes for the presented methods).

3.8.3 Complex and high dimensional search spaces

Hyperparameter optimization is further complicated by the dimensionality and complexity of the search space. As mentioned in Section 3.6.6, DCNNs have a large combined space which consists of architectural and learning parameters, and the dimensionality and complexity of the space increases exponentially as more layers get stacked on top of each other. Previous empirical work has demonstrated that certain hyperparameters are more important to performance than others, however advance identification without costly objective function evaluations is often challenging (Bengio, 2012). The hyperparameter search space complexity also poses challenges to the search method. Specifically, with regards to DCNNs, their hyperparameters are of the discrete, categorical and continuous type (see Section 3.6.6) and thus indicate the need to use methods capable of solving mixed-type optimization tasks.

3.9 Common hyperparameter search approaches

A wide array of optimization methods have been used for tuning the hyperparameters of DCNNs, and neural networks in general. The most commonly employed methods are manual tuning, grid search and random search, all of which will be briefly introduced in the following subsections.

3.9.1 Grid and random search

The grid search approach (Pedregosa et al., 2011) is the simplest and most direct method to tackle the hyperparameter optimization problem. It entails defining a set of hyperparameters, and training individual DCNNs on all their possible combinations. At the end of the search, the best performing set of hyperparameters is selected. Consider a simple search space that consists of just three hyperparameters, with ten possible values for each hyperparameter. A grid search will require training and evaluating $3^{10} = 59049$ individual CNN models, which is a very large number of evaluations especially for costly objective function evaluations such as DCNNs. Thus, whilst a grid search approach explores all the possible hyperparameter

combinations, and is ideal for cheap objective function evaluations, the downside is an extremely high computational cost for costly objectives.

Another approach is random search (Bergstra and Bengio, 2012), which randomly samples through the search space using a probability distribution that's uniform in nature. Overall computational times compared to grid search can be reduced since a predetermined number of samples less than an exhaustive grid search can be set. The downsides are that the entire space is not explored, and since the technique does not have recollection of the hyperparameters previously searched, resampling of the same point is conceivable. Furthermore, there is no direction towards indorsing the top performing hyperparameter combinations.

3.9.2 Manual search

Manual model tuning techniques rely on domain knowledge and expert experience to conduct and guide the hyperparameter search. Intuitively, manual model tuning works by identifying regions in the hyperparameter space, in which promising performance is achieved and then based on this, formulating the required intuition to select the near optimal set of hyperparameters. A major problem with this is that the reproduction of results, which is essential for scientific advancement and ease of application for basic users, is often difficult (Bergstra and Bengio, 2012).

3.10 Discussion

The challenges associated with these common hyperparameter approaches have led researches to investigate other avenues such as reinforcement learning based techniques (Li and Malik, 2016; Andrychowicz et al., 2016) evolutionary algorithms like particle swarm optimization (Lorenzo, Nalepa, Kawulok, Ramos and Pastor, 2017) and genetic algorithms (Xie and Yuille, 2017; Miikkulainen et al., 2017), physical algorithms such as simulated annealing (Souza, Suykens, Vandewalle and Bolle, 2010), and Bayesian optimization (Snoek et al., 2012; Swersky et al., 2013; Snoek et al., 2015). In particular Bayesian optimization has received a lot of attention recently due to it's efficacy with regards to objective function evaluations (Claesen and Moor, 2015). However, Bayesian approaches are complex to implement, and this has resulted in the exploration of other avenues for DCNN hyperparameter optimization, such as basic and straightforward to implement genetic algorithms. The presented work described in the remaining chapters, explores the advantages and challenges of these promising directions and presents a novel GA-Bayesian search to sequentially tackle the

model selection problem. In the chapters that follow, GA and Bayesian optimization based hyperparameter search approaches are surveyed in detail, with the intention of positioning the proposed hybrid method in the literature.

3.11 Conclusion

This chapter provided a concise survey of and motivation for deep learning CNNs for image classification, from their early development up to their role in the deep learning renaissance, and their significant popularization due to several successes from 2012 - 2017. This was followed by detailed discussions on their remaining challenges, with special focus on their computation, and high dimensional and complex search spaces. Thereafter, a formalization of the hyperparameter search problem for DCNNs, and the formal separation of the complete space into separate architectural and learning subspaces, followed. Some of the major challenges of the hyperparameter search problem, with reference to DCNNs, were also highlighted, and this was followed by a critical view on the commonly used hyperparameter search techniques. Finally, a brief introduction on recent advances to tackle the hyperparameter search challenge was provided, with the aim of locating the methods that will be presented in the remainder of this study. More specifically, in Chapter 5, a stochastic GA is presented to search the architectural space of modern DCNN models, and this is followed by a combined GA-Bayesian search of the learning subspace in the Chapter 6. Thus, in the next chapter, a concise background to these algorithms will be provided.

CHAPTER 4

Background to the proposed Genetic and Bayesian Algorithms

4.1 Chapter overview

This chapter provides the background to the algorithms presented in the rest of the dissertation. It begins by introducing evolutionary algorithms with the intention of locating the GA amongst other evolutionary-based optimization methods. Thereafter, a brief introduction to the GA and several of its representative operators is provided, before the presented work is positioned in the literature. This is followed by a section that motivates for the specific use of Bayesian optimization for the learning parameter search, and this is pursued by a formal introduction to Bayesian optimization for DCNN model selection. Discussions on several representative characteristics of the Bayesian driven search follow, before the chapter is closed out.

4.2 Evolutionary-Based Optimization Algorithms

Evolutionary algorithms (EAs), a subclass of evolutionary computation, which are general population-based, higher level, optimization processes, conduct search and optimization techniques by following natural evolutionary principles. They are dissimilar from traditional search and optimization techniques in a variety of ways.

EAs imitate natural biological evolution and/or the social behavior of organic species, making them stochastic search methods. Some examples of biological evolution include gemone evolution or more specifically, the evolution of chromosomes in living organisms such as human beings. On the other hand, the techniques used by birds or a shoal of fish to find their destination during migration or the methods used by ants to determine the shortest route to a source of food, are typical examples of social behavior of organic species (Elbeltagi, Hegazy and Grierson, 2005). The behavior of such species is guided by learning, adaptation, and evolution (Lovbjerg, 2002).

To mimic the well-organized behavior of these species, various computational systems that pursue fast and robust solutions to complex optimization problems, have been proposed (Elbeltagi et al., 2005). EAs are able to automatically find solutions to challenging optimization problems because of their ability to evolve and auxiliary tasks include the discovery of novel computer programs, the improvement of object shapes, the design of electronic circuits, and exploration of numerous other areas that are usually addressed by

human design (Floreano and Mattiussi, 2008). The three main characteristics (Yu and Gen, 2010) of EA's are that they are:

1) Population based – EA's uphold a population or groups of individual solutions, which they utilize to learn or optimize a given problem. This is a fundamental aspect of evolution.

2) Fitness orientated – The solutions in a population are governed by a genetic representation (or code) and are referred to as individuals. The foundation of EA's optimization and convergence properties are its preference towards fitter individuals. All the individuals in a population are subjected to a performance evaluation resulting in its fitness value.

3) Variation driven – The solutions or individuals in a population are exposed to operations that imitate genetic gene modifications, and this is essential to traversing the solution space optimally.

4.2.1 The Genetic Algorithm

Although several evolutionary-based techniques have been introduced over time, the first case in the literature was that of the genetic algorithm (GA), which was inspired by the Darwinian principle of survival of the fittest and the natural process of evolution through reproduction or breeding (Holland, 1975). The GA technique has been used to solve science and engineering related optimization applications, predominately due to its established ability to reach near-optimal solutions to complex problems (Hegazy, 1999; Al-Tabtabai and Alex, 1999; Grierson and Khajehpour, 2002). Briefly, for this technique a form of string, known as a chromosome, is used to represent the solution to a specified optimization problem. A set of elements, known as genes, which hold the set of values for the optimization variables, encompasses each chromosome (Goldberg, 1989). The chromosomes are also referred as a population of solutions, or parents and are usually random in nature. An objective function is usually used to assess the suitability (or fitness) of each solution. The genetic operations of selection, mutation and crossover are used to produce offspring chromosomes (or children) and this simulates the natural process of survival of the fittest. During this procedure the best chromosomes exchange information. If the offspring chromosomes provide better solutions compared to other weaker chromosomes, they are used to evolve the population. The process is iterative and usually sustained for a predetermined number of generations or until a stopping criterion is met.

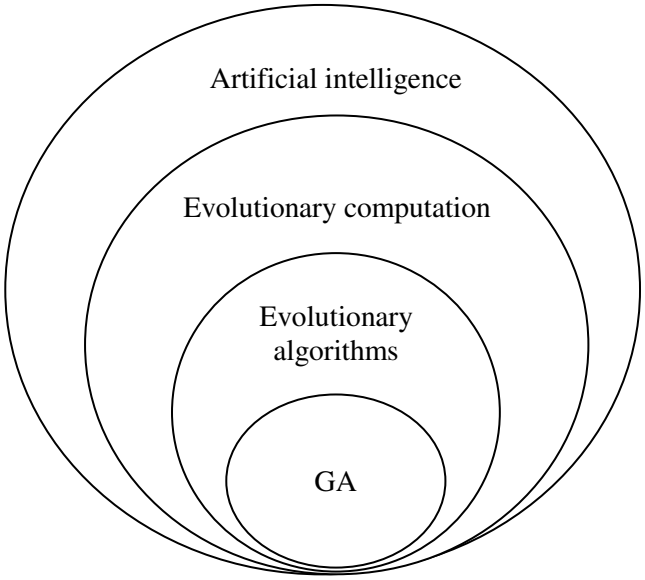


Figure 4.1: Venn diagram contextualizing the GA. The figure illustrates how the GA forms part of evolutionary algorithms, which forms part of the evolutionary computational branch of artificial intelligence.

While other parameters are required, the performance of the GA is principally governed by the population size, number of generations, crossover rate and mutation rate. As the population size and number of generations increase, the probability of finding an optimal solution is also increased, however this comes with an increase in computational costs (Elbeltagi et al., 2005). Thus, in most cases the available computational budget governs the size of the initial population and subsequent generations of evolution. For applications where real world evaluations are required to ascertain the fitness of individuals, smaller populations are often used (see Sections 5.3.3.3.1 and 5.3.4.1).

Similar to mutation in biological systems, the genetic operation of mutation contributes to preserving genetic diversity between generations. Biologically, the initial state of chromosome gene values are altered by the mutation operation and thus the derived solution may be completely diverse from the original. The result is that mutation stochastically introduces new content to the process of evolution, and its application to the GA for optimization problems can avert stagnation around local minima / maxima and promote better solutions in highly converged populations, often when crossover has a negligible effect (Elbeltagi et al., 2005, Floreano and Mattiussi, 2008). Formally, given an individual $\mathrm{I}_{t,n}$, where n and t represent the n-th individual and t-th generation respectively, the process of mutation involves randomly changing the contents of the individual (genes in biological sense

and bits in relation to an algorithm) with probability p_m[11]. Thus, mutation allows the newly derived individual to have novel characteristics, whilst still maintaining the good characteristics of the original individual. p_m is usually set low to prevent turning the GA search into a basic random search, and probabilities, per position, in the order of 0.01 are common (Floreano and Mattiussi, 2008). Several mutation operators exists and typical examples include bit flip mutation, normal mutation, uniform / non-uniform mutation, correlated / uncorrelated mutation, and boundary mutation, all of which are detailed in Yu and Gen (2010).

On the other hand, analogous to crossover and reproduction in biological systems, crossover in GA's is the process of taking more than a single parent and generating offspring (or children) from them. In GA's, crossover looks at selected individuals and creates pairwise crossovers (or recombination's) of their genomes, and this facilitates the generation of offspring that inherent the features and characteristics of their parents. Crossover is also commonly referred to as recombination.

Formally, using similar notation to mutation, given a pair of individuals $\mathrm{I}_{t,2n-1}$, $\mathrm{I}_{t,2n}$ the process of crossover simultaneously looks at two individuals, and uses their characteristics to create a new individual $\mathrm{I}_{t,n}$, which shares their attributes. Thus, children are produced by the interchange and recombination of their parent's information. Generally, the probability of executing a crossover operation is controlled by the crossover rate p_c, which is usually set between 0.6 to 1.0 (Elbeltagi et al., 2005). Several crossover operators exist, and these are usually tailored to suite the genetic representations that they are designed to evolve. Typical examples include one and multi-point crossover, uniform crossover, arithmetic crossover, differential crossover and swarm crossover (see Floreano and Mattiussi, 2008; Yu and Gen, 2010; Orive, Sorrosal, Borges, Martin and Alonso-Vicario, 2014, for further details). Figure 4.2, briefly introduces the workings of a typical GA, however further details on its mechanisms can be found in (Goldberg 1989; Al-Tabtabai and Alex, 1999). For an introduction to schema theory, which was formulated to formally show how GAs efficiently explore the high dimensional search space for progressively better solutions, the original book that introduced GAs is highly recommended (Holland, 1975), whilst further applications on GAs, coupled with a detailed description on schema theory and its limitations are presented in Mitchell (1996).

[11] p_m is also usually referred to as the mutation rate

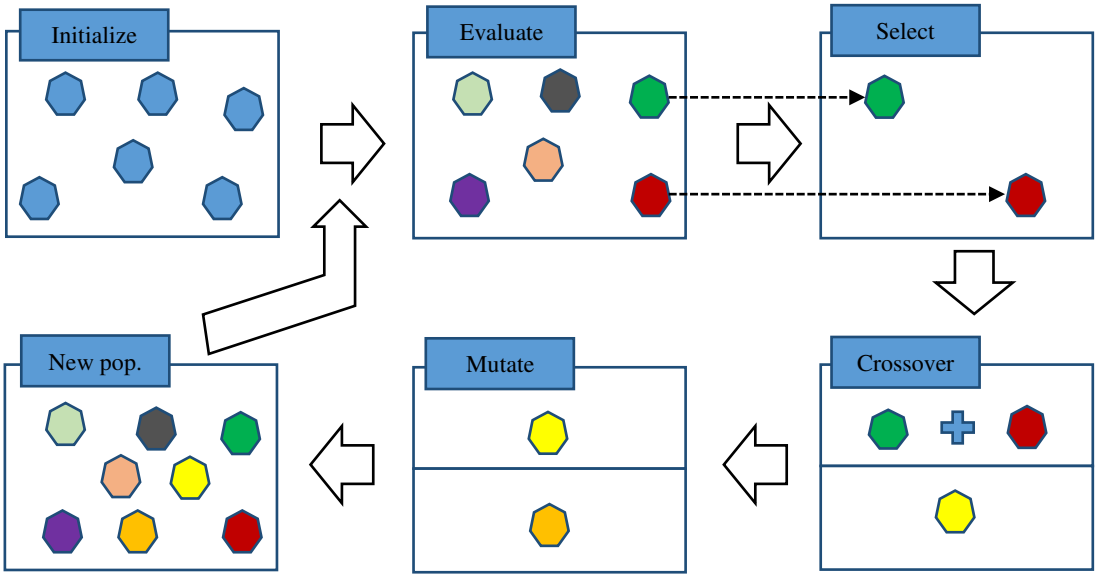


Figure 4.2: Graphical toy illustration of the genetic process and several of its representative operators

4.3. Related work

Previously, neuroevolution, which entails applying evolutionary processes to evolve the structure and architecture of both recurrent and feed forward neural networks, has seen several applications (Yao, 1999; Stanley and Miikkulainen, 2002; Gomez, Schmidhuber and Miikkulainen, 2008; Bayer, Wierstra, Togelius and Schmidhuber, 2009; Stanley, D'Ambrosio, and Gauci, 2009; Ding et al., 2013). However, despite the successes of several neuroevolutional-based techniques, their adaptation to DCNNs has not been studied extensively in the past, probably because of the complicated structure, large model size, and significant computational burden imposed by modern DCNNs (Desell, 2017). On the unsupervised front, Koutnik et al. (2014) used a CNN as an input to a recurrent neural network, which was evolved using the algorithm from Gomez et al. (2008); however, they held the CNN architecture fixed. Lately, an evolutionary strategy for deep auto-encoders, which also use the convolutional operator, was also proposed (Fernando et al., 2016), whilst other related hyperparameter and topological optimization approaches for CNNs, make use of recurrent networks and reinforcement learning motivated policy iterations (Zoph and Le, 2016).

More closely related to the proposed work, recent studies have begun focusing on the optimization of supervised DNNs. For example, Loshchilov and Hutter (2016) optimized the hyperparameters of existing DCNNs, in a large-scale parallel setting using the Covariance Matrix Adaptation Evolution Strategy (Hansen and Ostermeier, 2001); however, the

application required 30 GPUs, which is not practical for large-scale deployment. Other yet to be published manuscripts have appeared on the topic of using evolution for automatically learning the structure and hyperparameters of CNNs, illustrating that the topic is at the forefront of current DCNN advances. For instance, Xie and Yuille (2017) freshly proposed encoding CNNs as binary strings so that they can be subject to a standard GA. However, they evolve the structure of the convolutional operator (i.e. the number of convolutional nodes that give optimal performance), whilst keeping the number of filters and filter sizes fixed. Furthermore, other architectural details like the use of Dropout (Hinton et al., 2012) and the Dropout rate are also fixed. These parameters are essential to achieve good classification performance and are thus optimized in the proposed work.

Miikkulainen et al. (2017) freshly proposed using evolution to learn the topology and hyperparameters of deep models, by extending the neuroevolution technique proposed by Stanley and Miikkulainen (2002). Nevertheless, the results yielded structures that were compound and unprincipled, and models that did not promote the reuse of basic components or building blocks, which is in contrast to traditional deep models. Co-evolving the network topology, model components, and hyperparameters, resulted in the evolution of improved structures; however, the improved method is complicated since it utilizes a complex crossover operator that requires the evolution of a dual population of modules and blueprints that are recombined into larger assembled networks. Other very recent work (Desell, 2017) proposed using a distributed network of over 5000 computers, and over two months of computation, to evolve the architectural and learning parameters of DCNNs, but like the work presented by Loshchilov and Hutter (2016), large-scale adaptation is not conceivable purely because of the required hardware infrastructure.

In general, evolving the structure (number of convolutional and pooling layers or operations) of DCNNs for optimal classification performance, like the fresh work presented by Xie and Yuille (2017) and Miikkulainen et al. (2017), has its merits, but is a computationally burdensome procedure, which is further compounded when combined with hyperparameter optimization, and thus, is not tackled here. Even with the structure fixed, traversing the parameter search space to select the optimal model parameters (hereafter referred to as model selection) for modern DCNNs using GA's or other evolutionary strategies requires excessive computation, since each member of the population represents an individual DCNN that needs to be trained and scored. Furthermore, if the architectural (number of filters, filter sizes, activation functions, the use of Dropout and Dropout rate) and learning parameters (optimizer, learning rate, batch size and weight initialization) both form part of the search

space, the number of possible models grow exponentially with each additional parameter, thus making a GA based search intractable.

Considering these challenges, a traditional, yet highly stochastic GA, is presented to find the near optimal architectural parameters of a DCNN with a fixed structure in Chapter 5. Unlike the complicated approaches of others (Loshchilov and Hutter, 2016; Miikkulainen et al., 2017), the presented method shows that such sophistication is unnecessary and that standard, yet highly stochastic, evolutionary processes (see Section 5.2.3) can be used for model selection. Furthermore, previous work relied on elaborate computing power (Loshchilov and Hutter 2016; Desell, 2017) or at least the use of GPU's (Xie and Yuille, 2017; Miikkulainen et al., 2017) to merge GA's with DCNNs, however, here it is shown that a stochastically orientated GA guided search, can lead to classification improvements over baseline models, even with computation constrained to a central processing unit (CPU) alone.

Moreover, to prevent using large GA populations and running the GA for numerous iterations, both of which will add to computation, the model selection search space is efficiently partitioned into architectural and learning subspaces, which are optimized separately. Specifically, the stochastically inclined GA is used to optimize the architectural space, as detailed in the rest of this chapter. Its optimized parameters then form the basis of a second optimization process, using Bayesian optimization, which is applied to the models learning subspace, as detailed in Chapter 6.

4.4 Motivation for Bayesian optimization

In general, evaluating the model parameters of deep models, and DCNNs in particular is computationally expensive since each model needs to be fitted and evaluated from scratch; however, parameter tuning is essential for good performance. The traditional methods for model selection include grid search (or brute force computation), random search (Bergstra and Bengio, 2012) and manual tuning; yet all of these have their own challenges. Manual model selection requires expert domain knowledge or unsystematic rules of thumb (Dernoncourt and Lee, 2016), grid search (Pedregosa et al., 2011) is computationally burdensome (Snoek et al., 2012), and whilst random search (Bergstra and Bengio, 2012) relaxes some of the computational load imposed by grid search, it is not directed towards endorsing high performing models. On the other hand, GA's such as the algorithm presented in the next chapter alleviates some of these challenges; however, as the model selection search space increases, to search for near optimal solutions will require several runs of evolution with extremely large population sizes, which will significantly hinder computation (Elbeltagi et al. 2005). Furthermore, although GA's are well suited to search discrete or categorical

parameters, such as the options of the architectural search conducted in the next chapter, traditional GA's, are not suited for continuous parameters. Thus, as the dimensionality and complexity of the search space increases, it can be computationally beneficial to use other methods that efficiently seek for the best model parameters.

In recent times, Bayesian optimization (Mockus, Tiesis, and Zilinskas, 1978) has emerged as a sophisticated, yet effective and powerful, solution to the model selection problem (Snoek et al., 2012). Bayesian optimization has been shown to efficiently find near optimal parameters for a diverse range of models including statistical methods such as Markov chain Monte Carlo models (Hamze, Wang and de Freitas, 2013; Mahendran, Wang, Hamze, and de Freitas, 2012), deep belief networks (Bergstra, Bardenet, Bengio, and K´egl, 2011) and most significantly DCNNs (Snoek et al., 2012; Swersky et al., 2013; Snoek et al., 2015). Whilst there is current research into deriving ways to improve the performance of Bayesian optimization techniques including new sampling and searching strategies (McLeod, Osborne, and Roberts 2017), and integrating Bayesian neural networks into Bayesian optimization (Snoek et al., 2015), here the focus is to adapt existing approaches and combine them with GA's, for DCNN model selection. More specifically, whilst the next chapter deals with the optimization of the architectural parameters through a stochastic GA, the focus in Chapter 6 is to optimize the learning parameters, some of which are of the continuous type thereby making them intractable for a genetic search. Thus, whilst Bayesian optimization has been applied to the model selection problem for DCNNs previously (Snoek et al., 2012; Swersky et al., 2013; Snoek et al., 2015), the approach presented here aims at combining it with GA's, which has not been studied in prior work.

Bayesian optimization (introduced formally in the next section) incorporates previous information about the problem to direct the search, thus promoting a trade-off between the exploration and exploitation of the search space. This inherent characteristic is fundamental to its efficiency. Bayesian optimization derives its name from the established “Bayes' theorem”, which briefly asserts that the *posterior*[12] probability of a model M, provided evidence E, is relative to the probability of E, given M increased by the *prior*[1] probability of M (Brochu, Cora, de Freitas, 2010):

$$P(M|E) \propto P(E|M)P(M) \tag{4.1}$$

[12] The *posterior* in Bayesian optimization captures the updated confidence regarding the unknown objective function, whilst the *priori* signifies the confidence regarding the space of potential objectives

4.5 Formal introduction to Bayesian optimization for model selection

Similar to the GA optimization approach discussed in the next chapter, and other typical types of optimization, in the framework of Bayesian optimization, we are interested in searching for the global maximum (or minimum) of an unspecified objective function. Formerly, for the maximum case, we have:

$$\mathrm{x}^* = \arg\max_{\mathrm{x} \in \chi} f(\mathrm{x}) \tag{4.2}$$

where f is the objective function and χ is a bounded set or the search space of interest, which can be conceived as a subset of $\mathbb{R}^d$. For general optimization techniques, χ is more often than not a compact subset of $\mathbb{R}^d$, however in the Bayesian case, it can be generalized to more uncommon spaces that consider conditional or categorical inputs, or several of these inputs in the case of combinatorial search spaces. Moreover, Bayesian optimization accepts that f is unknown (often referred to as a blackbox function) and has no basic closed form, however, at x, which represents any random point of query in the domain, it can be analysed. This analysis or evaluation produces stochastic (or noise corrupted) outputs of the form $\mathrm{y} \in \mathbb{R}$, so that $\mathbb{E}[y|f(x)] = f(x)$. This implies that point-wise, unbiased and stochastic observations y are the only insights into the function f. Consider a sequential search algorithm (SSA) that queries f at a selected location x_{i+1}, where i represents the iteration number and y_{i+1} the point of observation. In this context, after a specified number of queries I, a final recommendation $\overline{\mathrm{x}}_{\mathrm{I}}$, which illustrates the best estimate of the optimizer, is made by the algorithm. More specifically, for an application that is characterized by large amounts of input data such as an object or image recognition task, the blackbox function f can represent the recognition system (DCNN with regards to the presented work), with selectable parameters x (for e.g., the architectural and learning parameters), with a nondeterministic discernible classification accuracy $y = f(\mathrm{x})$, on a specific dataset such as the MNIST dataset (LeCun et al., 1998). The framework of Bayesian optimization is extremely data efficient, thus resulting in it being of particular benefit for costly evaluations of f, such as training and evaluating a modern DCNN. Furthermore, for such an application, Bayesian optimization ensures that the search maintains a high efficacy, by manipulating the complete history of the optimization process.

Essentially, Bayesian optimization approaches a particular search problem by utilizing a sequential model-based optimization (SMBO) approach to search for the global optimum represented by Eqn. (4.2). More specifically, after a prior confidence over all the potential objective functions is prescribed, the model is sequentially refined based on the outcome of a

Bayesian posterior update procedure, which represents the updated confidence based on given data, on the probable objective being minimized (or maximized). This type of probabilistic surrogate model equips Bayesian optimization with the ability to sequentially persuade acquisition functions (see Section 4.5.2) of the form $\alpha_i: \chi \longrightarrow \mathbb{R}$, which direct the exploration of the search by leveraging the uncertainty in the posterior. Thus, by evaluating the usefulness of possible points for the succeeding evaluation of f, the acquisition function allows the selection of x_{i+1}, by maximizing α_i. In this context, the index i represents the implied reliance on the previous points where the objective function was evaluated and the related stochastic outputs (Shahriari et al., 2016). Thus, intuitively, acquisition functions promote efficient sampling by providing decisions that represent an automated trade-off between exploration[13] and exploitation[14] (Brochu et al., 2010; Shahriari et al., 2016). Once this trade-off is complete, by maximizing these acquisition functions, algorithms that use Bayesian optimization then determine the next point of enquiry resulting in the techniques efficiency. Thus, from the above introduction, it can be concluded that Bayesian optimization has two fundamental components. Firstly, it constructs a surrogate regression model, which is inherently probabilistic, to capture the confidence regarding the behavior of the blackbox objective function. To achieve this, the regression model must consist of a prior distribution (Shahriari et al., 2016). In other words, the assumptions about the function being optimized, are expressed by a selected prior over functions (Snoek et al., 2012). Furthermore, an observational model defines the mechanism that generates the data. The three most common types of regression models used for Bayesian optimization are Gaussian processes (Rasmussen and Williams, 2006), Random Forests (Breiman, Friedman, Olshen and Stone, 1984; Breiman, 2001) and tree-structured Parzen Estimators (Bergstra et al., 2011; Bergstra, Yamins, and Cox, 2013). Secondly, to conceptualize a useful function from the posterior of the model, and thus permit the search to determine its next point of evaluation, an acquisition function needs to be selected (Snoek et al., 2012). Over the years several acquisition functions have been proposed, with some of the representative functions being the Probability of improvement over the best current assessment (Kushner, 1964), the Expected Improvement over the best current assessment (Jones, Schonlau and Welch, 1998), and a Lower confidence bound (for minimization) on Gaussian processes (Srinivas, Seeger, Kakade and Krause, 2009). Furthermore, other recent acquisition functions, based on information-theory, such as the Entropy search acquisition (Hennig and Schuler, 2012), and a fast approximation of it referred to as the Predicative entropy acquisition (Hernández-Lobato, Hoffman, and,

[13] Sampling in areas with significantly high levels of uncertainty
[14] Sampling in areas where there is a likely high level of model predication

Ghahramani, 2014), have also been proposed. In the following sections, both of these fundamental components of Bayesian optimization will be briefly introduced.

4.5.1 Surrogate regression models

Although several regression models have been proposed in the Bayesian context, the surrogate must be able to define a predictive distribution $p(y|\mathrm{x}, \mathcal{D})$, where $\mathcal{D}$ represents a historical set of values, to be able to work with a traditional SMBO algorithm. This type of distribution facilitates the reconstruction of the objective function by recapturing the uncertainty in the regression model (Eggensperger et al., 2013; Shahriari et al., 2016; Dewancker et al., n.d.). In the following section, three popular Bayesian regression models are briefly introduced.

4.5.1.1 Gaussian processes

In the Bayesian optimization framework, Gaussian processes (Rasmussen and Williams, 2006), are non-parametric models that are flexible and controllable, thus making them a suitable and powerful probabilistic regression model for estimating objective functions (Snoek et al., 2012). Formally, Gaussian processes take the form $f : \chi \longrightarrow \mathbb{R}$. They are defined by the characteristic that a multivariate Gaussian distribution on $\mathbb{R}^N$, is induced by any predetermined set of N points represented by $\{\mathrm{x}_n \in \chi\}_{n=1}^{N}$, for which the function cost of $f(\mathrm{x}_n)$ is determined by the nth point. Furthermore, the inherent marginalization characteristics of the Gaussian distribution promotes the computation of conditionals and marginal's in closed form (Rasmussen and Williams, 2006; Snoek et al., 2012). Moreover, the Gaussian process is fully categorized by $m: \chi \longrightarrow \mathbb{R}$, which is its prior mean function, and its covariance[15] function $K: \chi \times \chi \longrightarrow \mathbb{R}$. The covariance function K regulates the smoothness and amplitude of the samples from the Gaussian process and it provides it with the ability to express an extensive distribution on functions, whilst the prior mean m introduces a conceivable offset (Snoek et al., 2012; Shahriari et al., 2016). Despite the successes of Gaussian surrogates, exact inference in Gaussian model regression is computationally costly, and this has led to other sparse Gaussian alternatives (Seeger, Williams and Lawrence, 2003; Quinonero-Candela, Rasmussen and Figueiras-Vidal, 2010) and an alternative to Gaussian process surrogates, as introduced in the next section.

[15] Sometimes referred to as the Gaussian processes positive-definite kernel

4.5.1.2 Random Forests

Random forests or ensembles of regression trees (Breiman et al., 1984), first proposed in 2001 (Breiman, 2001) have seen several practical successes due to their scalability and parallelism, as regression models (Criminisi, Shotton and Konukoglu, 2011). When applied to Bayesian optimization, the surrogate model consists of regression trees that learn random subsamples of the data (Breiman, 2001). The predictions of the individual trees are then averaged to compute a precise response surface. More formally, using the notation from Dewancker et al., (n.d.) the predictive distribution is usually constructed by assuming a Gaussian of the form $N(y|\hat{\mu}\hat{\sigma}^2)$, where the parameters $\hat{\mu}$ and $\hat{\sigma}$ are selected as the experiential mean and variance of the regression values, denoted by $r(\mathrm{x})$. Thus, the mean and variance can be respectively denoted by the following:

$$\hat{\mu} = \frac{1}{|T|}\sum_{r\epsilon T} r(\mathrm{x}) \tag{4.3}$$

$$\hat{\sigma}^2 = \frac{1}{|T|-1}\sum_{r\epsilon T}(r(\mathrm{x}) - \hat{\mu})^2 \tag{4.4}$$

where T denotes the set of regression trees in the forest, from where the regression values are drawn. Whilst Gaussian process modeled Bayesian techniques don't usually support conditional[16] variables (Hutter, Hoos, Leyton-Brown and Stützle, 2009; Bergstra et al., 2011), regression trees inherently support them, thus possessing an attractive characteristic.

4.5.1.3 Tree Parzen Estimators

Whilst Gaussian process based Bayesian optimization models $p(y|x)$ directly, the tree-structured Parzen estimator technique (Bergstra et al., 2011) models $p(x|y)$ and $p(y)$ by using non-parametric densities to replace the distributions of the priori configuration, thus transmuting the generative process. Intuitively, this is dissimilar from the traditional SMBO approach, since a predictive distribution over the objective function is not defined. On the other hand, the tree-structured Parzen estimator technique acts as a generative modeling process for all domain variables by generating two hierarchal processes (or densities) that model the variables when the objective function is on either side of a specific quantile y^*. Formerly, they can be represented by the following densities:

$$p(x|y) = \begin{cases} l(x)\ if\ y < y^* \\ g(x)\ if\ y > y^* \end{cases} \tag{4.5}$$

[16] Conditional variables in this context refer to variables that exist as a result of ranges or configurations imposed by other variables

where the density formed by the observation $\{x^{(i)}\}$ is denoted by $l(x)$ and the density formed by the remaining observations, denoted by $g(x)$. The tree Parzen estimator approach facilitates the optimization of categorical, conditional, and continuous hyperparameters, complimented by the ability to support priors for each hyperparameter over the range of expected optimal performance (Eggensperger et al., 2013). However, in contrast to Gaussian processes and random forests, which consider the combined variable condition to model the objective function, it may not be able to apprehend other variable interdependencies. Despite this, it has seen several recent successes, such as those presented in Thornton et al. (2013); Bergstra et al. (2013); and Bergstra and Cox (2013).

4.5.2 Acquisition functions for Bayesian optimization

As introduced in Section 4.5, acquisition functions define a control mechanism that balances exploration of novel areas in the search space and exploitation of areas that are known to provide promising results. Despite the existence of several acquisition functions (Kushner, 1964; Jones et al., 1998; Srinivas et al., 2009), the expected improvement (Jones et al., 1998) function is the most popular (Brochu et al., 2010; Eggensperger et al., 2013) and is used by all the surrogate regression models discussed in the previous sections. Formerly, for some model M of $f: \chi \longrightarrow \mathbb{R}$, the expected improvement is the expectancy that $f(x)$ will negatively exceed some quantile (or threshold) y^*. Thus, the expected improvement can be computed by:

$$EI_{y^*}(x) \coloneqq \int_{-\infty}^{\infty} \max(y^* - y, 0) p(y|x) dy. \tag{4.6}$$

where at a specific location x, y represents the best formerly observed objective value (Bergstra et al., 2011). The expected improvement criterion has been shown to demonstrate superior behavior to the probability of behavior criterion, and it does not require parameter tuning like Gaussian process confidence bound acquisitions (Srinivas et al., 2009; Snoek et al., 2012). Furthermore, this criterion has intuitive characteristics and has been shown to work well in a variety of applications (Bergstra et al., 2011).

4.6 Conclusion

This chapter gave an overview of EA's, followed by an elaboration on the GA and several of its operators. A brief survey of using evolutionary processes for neural networks structural and architectural choices, with special focus around CNNs, was conducted with the intention of locating the proposed application of a stochastic GA for CNN model selection. Next, an introduction to Bayesian optimization and several of its representative components was

provided, and this was preceded by a formal introduction to Bayesian optimization for the task of model selection. In the next chapter, a stochastic GA is proposed and applied to tackle the model selection problem.

CHAPTER 5

GA inspired DCNN model selection

5.1 Chapter overview

This chapter presents an application of the GA to the computationally intensive task of DCNN model selection. It commences by describing and formalizing a methodology of applying the GA to the given task. Several traditional genetic operators are adapted and utilized in an algorithm that applies evolutionary processes to efficiently traverse the parametric search space of modern DCNNs. More specifically, the search space is efficiently separated into architectural and learning subspaces, with the GA applied to the former. Thereafter, experiments using the GA for model selection against a brute-force approach are conducted, before the method is extended to traverse a larger model selection search space. Finally, the results are discussed and analyzed before the chapter is closed out with a brief conclusion.

5.2 CNN model selection using GA's

5.2.1 Introduction

In this section, a standard stochastic GA is used to design, by evolution, the network topology and hyperparameters of a CNN. First, the methodology to obtain a near optimal CNN architecture is described. Next, the algorithm together with its corresponding genetic operations is introduced and this is followed by the evaluation of the approach to a standard image classification related application. For the remainder of this work, the GA is used to propose new architectural details and hyperparameters, whilst the networks free parameters (i.e. the biases and weights) are computed through standalone traditional neural network training using backpropagation (see Section 2.3.1).

5.2.2 Methodology

As mentioned in Section 4.2, EA's maintain a population of solutions that traverse a solution space and they use evolutionary processes to obtain near optimal solutions. Each solution is then evaluated and based on the score or fitness of the individual solutions, the population is evolved. During the evolutionary process, selection takes place and the fittest members of the population are subjected to crossover and mutation in order to evolve the population by improving its overall fitness and thus generate feasible solutions to the optimization problem.

With this in mind, a population of CNNs are evolved with the intention of finding the best architecture and hyperparameters for a traditional image classification related application. However, unlike previous applications of the GA to the hyperparameter problem, to promote convergence in minimal iterations, with small population sizes, compounded by constrained hardware, the search space is effectively detached into architectural and learning subspaces, optimized, respectively, by a stochastic GA and state-of-the-art Bayesian technique, as described in the next chapter.

The architectural parameters that are optimized using the GA are the number of convolutional filters for each layer, the size of the convolutional filters for each layer, the number of filters for the fully connected layers, the decision to use or negate Dropout (Hinton et al., 2012), the Dropout rate, if Dropout is used and the activation function of all the layers excluding the softmax layer[17]. The learning parameters, such as the optimizer to use, the learning rate, the batch size and the weight initialization method are separately optimized using Bayesian optimization. For the proposed GA, each member of the population subjected to evolutionary processes constitutes a set of topological choices, such those illustrated by Table 5.4 and Table 5.9, and thus an individual CNN model. Therefore, using the topological choices available in Table 5.9 as an example, an individual $\mathrm{I}_{t,n}$, may have the architecture illustrated by Table 5.1.

[17] The softmax activation function ensures that a vector of real-valued scores are squashed to a vector that contains values between zero and one that add up to one. Thus, it ensures that the output of the last fully connected layer has a sum of probabilities equal to one.

Table 5.1: Typical DCNN architectural selection space and sample parameter selection

Layer	Hyperparameter	Sample hyperparameter value
Convolutional layer 1	Number of filters	16
	Kernel size	3*3
Convolutional layer 2	Number of filters	32
	Kernel size	5*5
Fully connected layer 1	Number of filters	128
	Decision on Dropout use	No Dropout
	Dropout rate	N/A
Fully connected layer 2	Number of filters	128
	Decision on Dropout use	Dropout
	Dropout rate	0.5
All layers except softmax layer	Activation function	ReLU

5.2.3 Evolutionary process

5.2.3.1 Initialization, Selection and Retention

Figure 4.2 illustrates the genetic process, whilst Algorithm 5.1 summarizes the details. Formerly, a set of randomized individual CNN models $\{I_{0,n}\}_{n=1}^{N}$ are used to initialize the population of CNNs. Each network is then trained on a subset (training set $\mathcal{D}_{tr}$) of an image classification dataset $\mathcal{D}$, before being evaluated on its test set $\mathcal{D}_{ts}$. Since the fitness function of

the GA channels the evolutionary process, and is application specific, it's imperative to use an appropriate fitness function (Lessmann, Stahlbock and Crone, 2005), and given that the task is image classification, classification accuracy is selected. Here accuracy takes the notation $a_{t-1,n}$, as the evaluation of the *n*-th individual CNN $I_{t-1,n}$ takes place before crossover of the *t*-th generation. From an implementation perspective, notwithstanding the interpretability and importance of using accuracy as the fitness function, it cannot be directly used during the training of the CNN population, since the backpropagation algorithm necessitates the use of a metric that is differentiable. Thus, the classification accuracy is calculated by computing the softmax loss (see Section 2.3.2) in the final fully connected layer of the individual CNNs. The training and evaluation process is computationally expensive, as each model is trained and evaluated from scratch, and thus, this step is the bottleneck of the evolutionary process. The networks are then categorized according to their classification accuracy, and only the top performing individual CNNs $\{\mathrm{I}_{t,n}\}_{n=1}^{r_p}$, where r_p represents the predetermined percentage of models to be retained, are selected to evolve the population via reproduction and become part of the next generation $\{\mathrm{I}_{t,n}^{'}\}_{n=1}^{N}$. To prevent getting trapped in local maxima, a subset of the poor performing CNNs, are also retained, with random probability p_r.

5.2.3.2 Crossover

Initialization, selection and retention, is followed by random crossover c_r, in which children CNNs are breed from randomly selected pairs of parent members $\mathrm{I}_{t,2n-1}$, $\mathrm{I}_{t,2n}$ of the retained population (top performing CNN's and the retained poor performers). The number of children that are breed is dependent on the number of individuals *N* in the initial population $\{\mathrm{I}_{0,n}\}_{n=1}^{N}$, and the number of retained models. For example if $N = 16$, in the initial population, and 25% of the top performers were retained, plus another two of the weaker CNNs, ten children will need to be breed in order to maintain the original population size for the next generation $\{\mathrm{I}_{t,n}^{'}\}_{n=1}^{N}$. With this scheme, there is a possibility of an individual CNN appearing in different generations, since *N* remains unchanged from the initialized population $\{\mathrm{I}_{0,n}\}_{n=1}^{N}$. During crossover randomly selected topological choices from parent CNNs are crossed over to children CNNs, as illustrated by Table 5.2, where the selected parameters are represented by the shaded blocks.

Table 5.2: Illustration of crossover between parent CNN_A and CNN_B resulting in a child CNN_C

Conv. 1		Conv. 1		Fully connected layer 1			Fully connected layer 1			Activation
Filter no.	Filter size	Filter no.	Filter size.	Filter no.	Dropout use?	Dropout rate	Filter no.	Dropout use?	Dropout rate	Function to use?
Parent CNN_A										
64	3*3	32	5*5	16	No	0.25	256	No	N/A	ELU
Parent CNN_B										
32	5*5	16	4*4	128	Yes	0.5	512	Yes	0.75	ReLU
Child CNN_C										
32	3*3	16	5*5	128	Yes	0.5	256	No	N/A	ReLU

5.2.3.3 Mutation

Crossover is followed by mutation of the children, where the rate of mutation is controlled by p_m. The lack of mutation can cause a population to lack diversity and devolve, and thus mutation is imperative to promote diversity, augment the capability of the population and facilitate propagation (Floreano and Mattiussi, 2008, Yu and Gen, 2010). To implement mutation, a child CNN is selected with probability p_m and a randomly selected topographical feature of it is replaced with another arbitrarily selected feature, resulting in a mutated population $\{I_{t,n}^{'}\}_{n=1}^{N}$. The effect of mutation on a given CNN architecture is illustrated in Table 5.3, where the shaded blocks represent the mutated topographical parameters. Mutation in this fashion facilitates the retention of the majority of the strong topographical characteristics of the selected CNN, whilst also providing a chance of evaluating new CNN architectures. The entire evolutionary process is repeated for a predetermined number of generations T.

Table 5.3: Illustration of mutation in the offspring after crossover has taken place, resulting in a mutated CNN_ M´

Conv. 1		Conv. 2		Fully connected layer 1			Fully connected layer 2			Activation
Filter no.	Filter size	Filter no.	Filter size.	Filter no.	Dropout use?	Dropout rate	Filter no.	Dropout use?	Dropout rate	Function to use?
Child CNN_M – Before mutation										
64	3*3	32	5*5	16	Yes	0.25	256	No	N/A	ReLU
Child CNN_M´ – After mutation										
64	3*3	32	5*5	16	Yes	0.25	256	No	N/A	ELU

Algorithm 5.1: CNN model selection flow, using genetic processes of selection, crossover, and mutation

Algorithm 5.1: CNN model selection using the GA
Data: Reference classification training set $\mathfrak{D}_{tr}$ and test set $\mathfrak{D}_{ts}$ **Genetic process inputs:** The number of CNNs in the initial and subsequent generations *N*, the maximum number of generations *T*, the percentage of top performing networks in each generations r_p and the probability p_r of random poor performers being retained, random crossover c_r, and the rate of mutation p_m.
1: GA initialization: Generate a random initial population of CNNs $\{\mathrm{I}_{0,n}\}_{n=1}^{N}$
2: Train and evaluate the initial population: Train each individual $I_{t-1,n}$ on $\mathfrak{D}_{tr}$ and evaluate its accuracy $a_{t-1,n}$ on $\mathfrak{D}_{tr}$
for each generation, t = 1; 2; 3; 4; :: ; T, **repeat** the following genetic operations:
3: Selection: Select a percentage r_p of the top performing CNNs $\{\mathrm{I}_{t,n}\}_{n=1}^{r_p}$ to retain for the next generation, plus add random poor performing models with probability p_r, to form the next generation $\{\mathrm{I}'_{t,n}\}_{n=1}^{N}$.
4: Crossover: Perform random crossover c_r, for each CNN pair $\mathrm{I}_{t,2n-1}$, $\mathrm{I}_{t,2n}$ to maintain the population at *N*
5: Mutation: Select children randomly with probability p_m, and mutate a randomly selected topographical choice
6: Evaluation: Repeat step 2 for each generation
until the predetermined number of generations *T* is complete
Result: The final generation $\{\mathrm{I}_{T,n}\}_{n=1}^{N}$ of CNNs with their classification accuracies.

5.3 Experimentation

5.3.1 Overview

In the sections that follow, the experimental setup is introduced, and this is followed by the derivation of a base model. Thereafter, a brute-force computational approach is applied to a limited search space, and the results are compared to the application of the GA to the same model selection space. Thereafter, the algorithm is used to search a much larger model selection search space, which consists of filter numbers and sizes, activation functions, and Dropout rates. Each computational approach is followed by a detailed analysis and comparison of the results achieved.

5.3.2 Experimental Setup

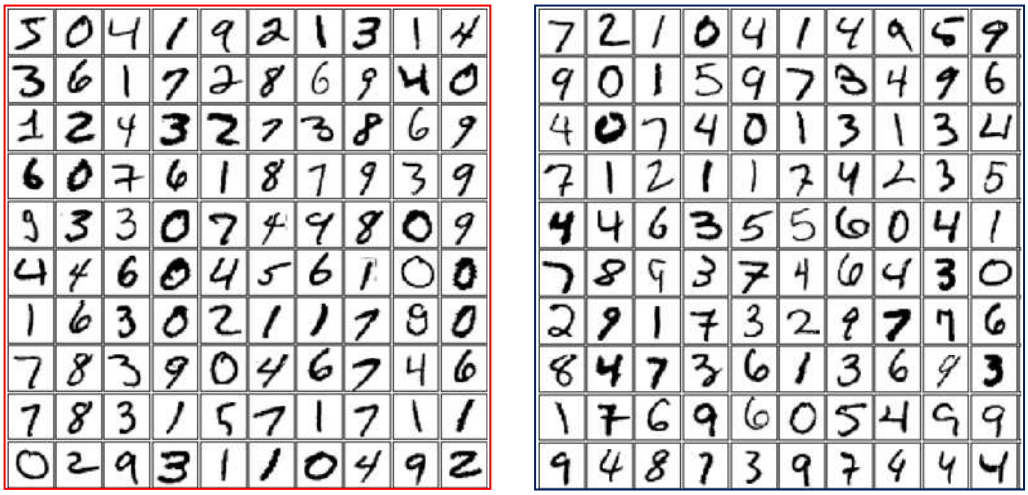
5.3.2.1 Development environment

All simulations were conducted on an 8-core Intel i7-6700k CPU, clocked at 4.0 GHz (4.2 GHz maximum frequency), with an 8 MB cache, and 16GB DDR4 random access memory (RAM). All the necessary image data was stored on a 512 GB solid state drive for swift access during training and evaluation. The software experiments were implemented in Python using the Keras (Chollet et al., 2015) application programming interface (API) whilst TensorFlow (Abadi et al., 2015) was used as the backend. Thus, Keras served as the high-level API in which the CNN models were created, and sat on top of TensorFlow, which did the required numerical computations using data flow graphs. Other Python libraries and dependencies included numPy and sciPy, whilst matplotlib, was used for generating several of the figures.

5.3.2.2 Data

The proposed GA was used to evolve the architecture of traditional CNNs, in order to maximize the classification performance on the popular MNIST dataset (LeCun et al., 1998). MNIST, which was constructed out of the NIST[18] dataset, consists of a collection of greyscale handwritten digits. Specifically, there are 60000 training images and 10000 test images, both drawn from the same data distribution over ten categories (i.e. images of digits from zero to nine). The first one hundred training and test set images are illustrated by Figure 5.1 a) and 5.1 b), respectively. All the images are size normalized, have dimensions of 28 * 28 pixels and the centre of gravity of their intensity is located around the midpoint of each image. Each image has a sample vector of $28 * 28 = 784$ dimensions, where individual elements are represented in binary (Deng, 2012). The dataset was first introduced in 1998, and has since been used extensively for several computer vision tasks and, in particular, for image classification and recognition problems. In particular, CNNs have outperformed several other machine learning and pattern recognition models on this popular benchmark, and whilst a brief analysis on the classifiers that have used it can be found in Deng (2012), the performances of several state of the art CNNs on the benchmark, are detailed in (Rawat and Wang, 2017).

[18] National Institute of Standards and Technology



a) b)

Figure 5.1: Grid view of the first hundred training (a) and test (b) set images of the MNIST dataset (LeCun et al., 1998), generated in Python using matplotlib.

The traditional train-test split of 60000-10000, was maintained for all simulations and the images were converted to single precision floating-point format (float32 in Python). The data values (pixels) were normalized to the range $[0, 1]$, before being fed to the network. Furthermore, the data was randomly shuffled for each epoch. The original MNIST dataset[19] is provided in four files, which consists of the training set images and its corresponding labels, and the testing set images and its corresponding labels. The data from the label files are represented by 1-dimensional arrays and thus are not split into the ten (zero to nine) different class labels, thus they were converted to 10-dimensional class matrices from 1-diminsional class arrays. No further preprocessing or data augmentation was considered at this point. Given the computational resources available at the time, and the fact the presented GA requires training each member of the population (each CNN) from scratch, evaluating the technique on large datasets like the challenging ImageNet (Russakovsky et al., 2015) image classification benchmark, is not practicable and thus left for future work.

5.3.2.3 Accuracy computation

Whilst several performance metrics such as the sparse categorical and top_k categorical metric functions are available in Keras[20] (Chollet et al., 2015), the categorical accuracy metric was selected for this study, since it was the most appropriate measure for the selected MNIST dataset. The categorical accuracy metric computes the accuracy of a pair of true and predicted

[19] Available from: http://yann.lecun.com/exdb/mnist/
[20] The performance metrics are available from: https://keras.io/metrics/

labels and checks if the predicted class is the same as the true class. It does this by using a maximum operator to determine the highest value of the set of predications, which it uses as the current prediction and compares this to the comparative set of true labels. This metric, which Keras defines as a function, is used to compute the overall accuracy of the test set, utilizing the following conventional expression:

$$\mathrm{A_i} = \frac{S_c}{S_n} * 100 \tag{5.1}$$

where $\mathrm{A_i}$ is the classification accuracy, S_c denotes the total number of samples (or images in this case), which are correctly classified, and S_n denotes the total number of samples.

5.3.3 Commissioning the GA for CNN model selection

5.3.3.1 Base model selection and improvement

The selected base model for GA model selection was derived from the popular LeNet-5 model (LeCun et al., 1998). The LeNet architecture consists of alternating layers of convolutional and subsampling operations, followed by three fully connected layers, as illustrated by Figure 5.2. Although the model has been reworked several times, it still forms an an essential component of many recent competition-winning DCNNs (Krizhevsky et al., 2012; Simonyan and Zisserman, 2014; Zeiler and Fergus, 2014; Szegedy, Liu, et al., 2015).

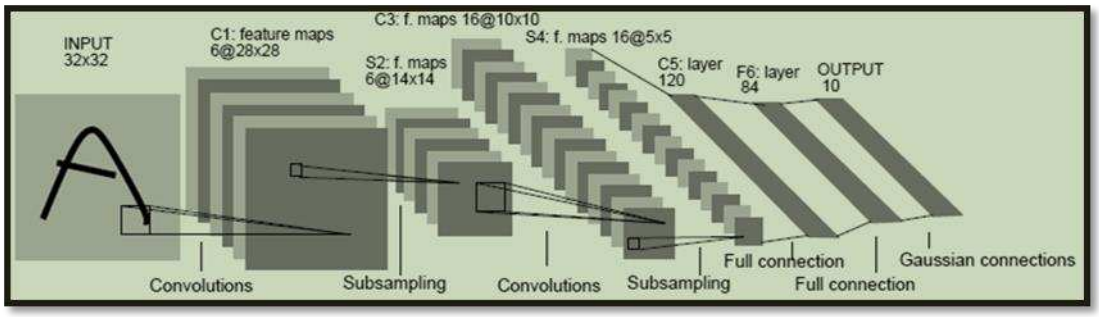


Figure 5.2: Architectural choices of the LeNet-5 model (LeCun et al., 1998)

To improve its performance, prior to using it as a base model (CNN_B1) for CNN evolution, several modern architectural changes were made to the original model. Firstly, the average pooling layers were replaced with max-pooling layers. Max pooling, introduced by Ranzato et al. (2007) has been shown to be a vastly superior operation for capturing invariance in image-like data and can lead to improved generalization and faster convergence when compared to an averaging operation (Scherer et al., 2010). Jarrett et al. (2009) showed that max pooling alleviated the need for a rectification layer, which is not usually part of

DCNN architecture; however, they found that average pooling does not enjoy the same benefit and thus suffers from cancellation effects between neighboring filter outputs. Furthermore, Boureau et al. (2010) also carried a detailed theoretical analysis and further empirical work to uncover the advantages of using max pooling over average pooling. Thus, given the above motivation, max pooling with a 2*2 stride was used to replace the average pooling layers in the modified LeNet model.

The original 5*5 convolutional filter sizes were used, although this parameter was optimized later. The original LeNet used 6 filters in the first convolutional layer and 16 in the second layer. In the base model prior to CNN evolution, these were changed to 32, and 64, respectively for improved performance, whilst the number of filters in the first fully connected layer was also increased to 128 filters for the same reason. These adjustments followed the default, yet expertly designed Keras (Chollet et al., 2015) CNN for MNIST (LeCun et al., 1998). To regularize the base model and thus prevent over-fitting, Dropout (Hinton et al., 2012 - see Section 2.2.4.1 for a formal introduction) was applied to the fully connected layers of the model (Krizhevsky et al., 2012; Hinton et al., 2012; Srivastava et al., 2014). Specifically, when each training case was presented to the network during the training phase, each hidden neuron was randomly omitted from the network with a probability of 0.5 (this probability was optimized later). Thus, hidden neurons could not rely on other hidden neurons being present, and this prevented complex coadaptation of features on the training data. At test time, all of the hidden neurons were used, but their outputs were multiplied by 0.5 to compensate for the fact that double the number of neurons were now active, thus resulting in a strong regularization effect that significantly reduced overfitting. Whilst Appendix A illustrates the DCNN structure used during all simulations, Figure 5.3 illustrates the difference in performance of the base model with the original number of filters and no Dropout (Le-Net derived model named CNN_B01) compared to its performance with modern filter choices and Dropout (model named CNN_B1). As illustrated, CNN_B1 slightly outperforms CNN_B01, when trained for a constrained number of epochs. After twenty epochs, CNN_B01 reaches an accuracy of 98.36%, whilst the accuracy of CNN_B1 supersedes it and achieves 98.60%. The best-seen accuracy for CNN_B01 is 98.36%, whilst CNN_B1 accomplishes a best-seen accuracy of 98.70%. Both models can achieve higher accuracy if trained for more epochs, but the focus for the rest of this chapter is to accomplish high accuracies, with limited computation and then transfer the results to the final models - see Section 5.3.4.2.

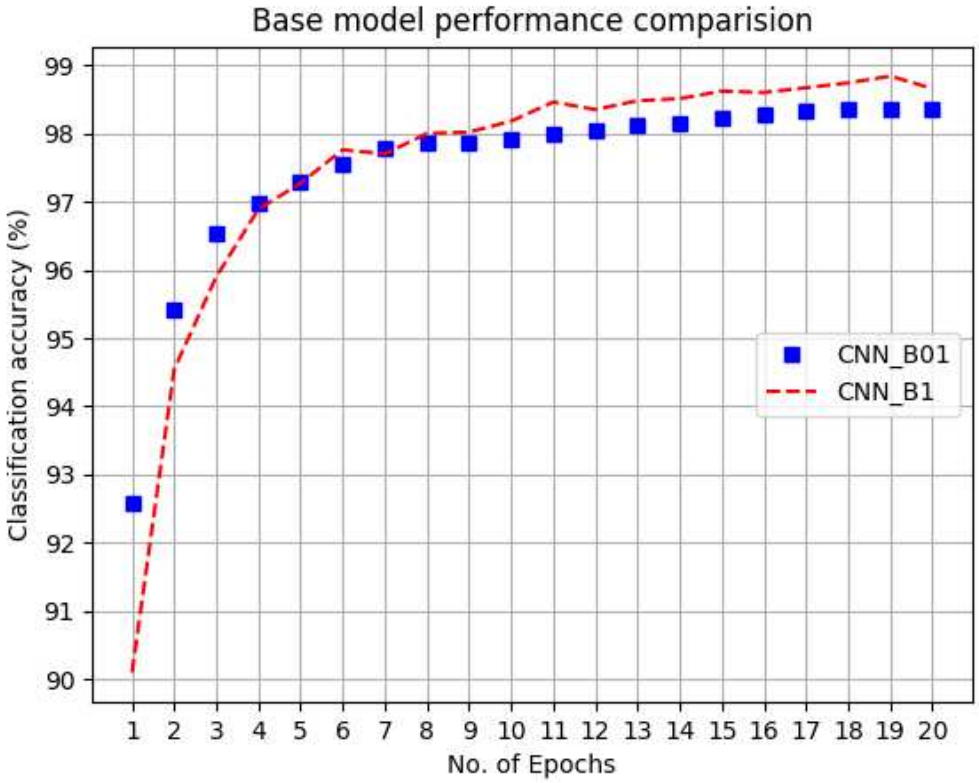


Figure 5.3: Performance comparison between CNN_B01 and CNN_B1

The popular softmax loss function (Krizhevsky et al., 2012; Goodfellow et al., 2013; Lin et al., 2013; Zeiler and Fergus, 2013; Simonyan and Zisserman, 2014; Chatfield et al., 2014; Szegedy, Liu, et al., 2015; Szegedy, Vanhoucke et al., 2015; He et al., 2015a, 2015b) was used as the objective function and it was minimized using the traditional and widely accepted SGD (Bottou, 1998, 2010) algorithm. The learning rate was set to 0.01, as recommended by (Bengio, 2012), whilst the batch size was set to 64. The convolutional and fully connected filters were initialized using the scheme presented in (Glorot and Bengio, 2010). No further learning parameter optimization was performed at this stage. The final base architecture, prior to optimization, CNN_B1, and its variant subjected to optimization CNN_GA_1.X, can be seen in Table 5.4.

5.3.3.2 Brute-force model selection

To test the GA technique for CNN model selection against a brute force approach, where all the models are examined, the hyperparameter choices for the first round of experimentation were limited to the number of filters for each layer of the base model - see Table 5.4. Thus, considering the number of filter choices available for each of the four variable layers, resulted in $4 * 3^3 = 108$ CNN models, which were independently verified. To save on computation, the number of epochs for each model was constrained to twenty epochs (i.e. the model was

shown the training set of 60000 images twenty times). On the hardware described in Section 5.3.2.1, this took a total of 1044 minutes, with an average of 9.66 minutes to train each network (training times varied since models with more filters take longer to train). The results of the top performing models from this brute force approach are illustrated in Table 5.5 and Figure 5.6. As illustrated, there is significant redundancy amongst the different parameter combinations and thus a brute force approach to evaluate all the models is a waste of computational resources. However, the top performing accuracy is noted and will be used as a benchmark for the results that will be obtained from the evolutionary process.

Table 5.4: The architectural choices of the base model CNN_B1 and the choices exposed to the GA search, illustrated by CNN_GA_1.X

Layer	**Hyperparameter**	**CNN_B1**	**CNN_GA_1.X**
Convolutional layer 1	Number of filters	32	Choice between: {8, 16, 32, 64}
	Kernel size	5*5	5*5
Max pooling layer 1	Filter size	2*2	2*2
Convolutional layer 2	Number of filters	64	Choice between: {16, 32, 64}
	Kernel size	5*5	5*5
Max pooling layer 2	Filter size	2*2	2*2
Fully connected layer 1	Number of filters	128	Choice between: {64, 128, 256}
	Dropout rate	0.5	0.5
Fully connected layer 1	Number of filters	84	Choice between: {64, 128, 256}
	Dropout rate	0.5	0.5
Global parameters	Activation Function - All layers except softmax layer	ReLU / Softmax in the softmax layer	ReLU / Softmax in the softmax layer
	Optimization	SGD	SGD

Table 5.5: The architectural choices and respective accuracies of the top performing models computed during the brute force run

Model	No. of filters Conv. 1	No. of filters Conv. 2	No. of filters FC. 1	No. of filters FC.2	Accuracy (%)
1	64	64	128	64	98.98
2	16	32	256	256	98.96
3	32	64	256	256	98.96
4	64	64	256	128	98.96
5	32	64	128	128	98.94
6	64	64	64	256	98.94
7	8	64	256	64	98.93
8	32	64	128	256	98.93
9	32	32	256	64	98.92
10	64	64	256	256	98.92

5.3.3.3 Limited GA model selection

5.3.3.3.1 Computational experiment

Next, the GA described by Algorithm 5.1 was applied to select the filter choices illustrated by CNN_GA_1.X in Table 5.4. To reduce the computational burden imposed by the brute-force approach, a small population with $N = 12$ individual CNNs was created and maintained for $T = 10$ generations of evolution. Given this small population, the percentage of the top performing models to retain was set as $r_p = 50\%$, whilst the probability of a poor performing model being retained was set to $p_r = 10\%$. Random crossover c_r was used to maintain the population size at $N = 12$, for each generation. This resulted in the training and evaluation of 12 models in the initial generation and 49 further models for the other nine generations. Thus, in total, $12 + 49 = 61$ CNNs were trained and evaluated during ten runs of evolution, since the models that are retained were not evaluated again to preserve computational resources. Table 5.6 shows the results of applying selection to the entire population and random retention to the rest of the poor performers. The probability of mutation p_m was set high to a value of 0.3, to promote the generation of models that represented new filter choices. To allow comparison with the brute force approach, training was also constrained to twenty

epochs (see Section 5.3.4.2, which illustrates the enhanced performance by training the optimized models for more epochs).

Table 5.6: The results of the genetic operations of selection, retention, and crossover

Generation	Retained no. of CNNs ($r_p = 50\%$)	Retained no. of poor performers ($p_r = 10\%$)	No. of crossed over CNNs	Population Size
T = 1	12	N/A	N/A	N = 12
T = 2	6	1	5	N = 12
T = 3	6	0	6	N = 12
T = 4	6	1	5	N = 12
T = 5	6	1	5	N = 12
T = 6	6	2	4	N = 12
T = 7	6	0	6	N = 12
T = 8	6	0	6	N = 12
T = 9	6	0	6	N = 12
T = 10	6	0	6	N = 12
Total	**66**	**5**	**49**	**N/A**

Table 5.7: The architectural choices and respective accuracies of the top performing models computed during the evolutionary process

Model	No. of filters Conv. 1	No. of filters Conv. 2	No. of filters FC. 1	No. of filters FC.2	Accuracy (%)
1	32	64	256	64	99.02
2	32	32	256	64	99.01
3	64	64	256	64	99
4	8	64	256	64	98.99
5	64	64	128	128	98.99
6	32	64	256	64	98.96
7	64	64	256	128	98.94
8	64	64	256	64	98.93
9	64	64	256	64	98.91
10	32	32	256	64	98.91
11	64	64	64	128	98.89
12	64	64	256	256	98.88

5.3.3.3.2 Experimental results and comparison to brute-force

Table 5.7 and Figure 5.6 illustrate the performance of the last generation of the evolutionary process, whilst the mean accuracy for each generation is depicted in Figure 5.5. The best performing model of the evolutionary process (CNN_GA_1.1) achieved an accuracy of 99.02%, which is slightly higher (stochastic nature of weight initialization, SGD and random Dropout does not guarantee determinism[21]) than the best model (CNN_BF) computed during the brute-force approach.

Although the accuracies and filter choices between the GA and brute-force approach are not exactly identical, which can be attributed to the non-deterministic nature of the computation, there are various similarities and positive signs between the two approaches. Firstly, amongst the final generation of the evolutionary process, 41.67% of the models share exactly the same parameters (4/4 filter choices) as the top ten models computed during the brute force run. Secondly, the remaining models (58.33%) of the GA derived run selected very similar parameters (3/4 filter choices) compared to their brute-force counterparts. Comparing the mean accuracies of the final generation and top sixty one models of the evolutionary process (total number of models evolved) to their top performing brute force counterparts (top twelve and top sixty one models respectively) shows very comparable general performance, albeit for different parameter choices and accuracies, between the two approaches, as highlighted by Table 5.8.

Table 5.8: Performance analysis between the base, brute-force, and GA approaches

Evaluation criteria	**Brute-force approach**	**GA-CNN**
Mean accuracy of top 12 models / last generation ($N = 12$)	98.94	98.95
Mean accuracy of top 61 models / all GA models	98.81	98.86
Top performing model	98.98	99.02

Furthermore, as illustrated in Table 5.7 and following the brute-force approach (Table 5.5), there is significant redundancy amongst the different parameter combinations. Notwithstanding these redundancies and the stochastic nature of the computation, as expected, as the training proceeded the top performing GA model also achieved better accuracy compared to the baseline model (CNN_B1 - which was not optimized). Furthermore, it also

[21] NB: This result is after an improvement in reproducibility was observed by setting random seeds to ensure all the core Python, Keras and TensorFlow random numbers were started in well-defined initial states, and the Python seed set to facilitate deterministic hash-based operations

accomplished very similar accuracies compared to the brute force model, as illustrated by Figure 5.4, which illustrates the performance of CNN_B1, CNN_BF and CNN_GA_1.1.

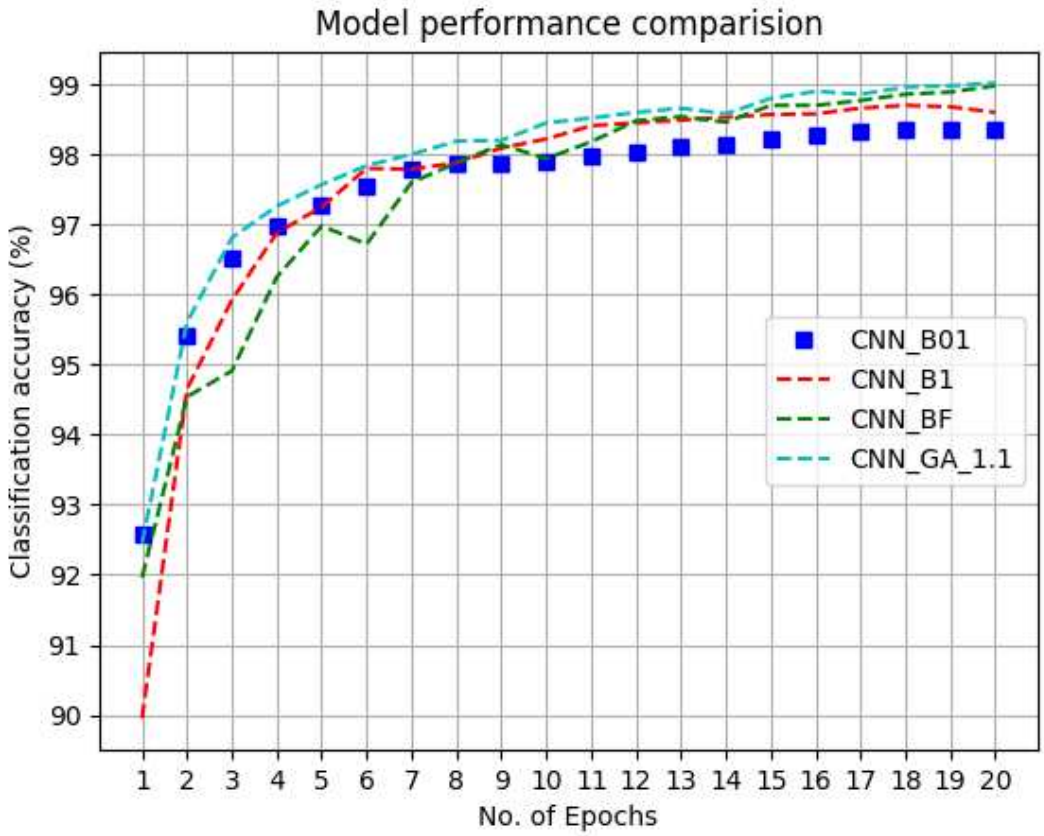


Figure 5.4: Performance comparison between CNN_B01, CNN_B1, CNN_BF and CNN CNN_GA_1.1

Generally, although experimentation over numerous generations is usually constrained by the computational budget available, in this case, the GA was only tested over $T = 10$ generations to force the total number of computed models to be below the total brute-force limit (108 models). However, the general trend observed in Figure 5.5 is that the GA leads to higher mean accuracies for succeeding generations. This is significant since it ensures that the GA promotes the development of fitter individuals, which is one of the three fundamental characteristics of an EA (Yu and Gen, 2010).

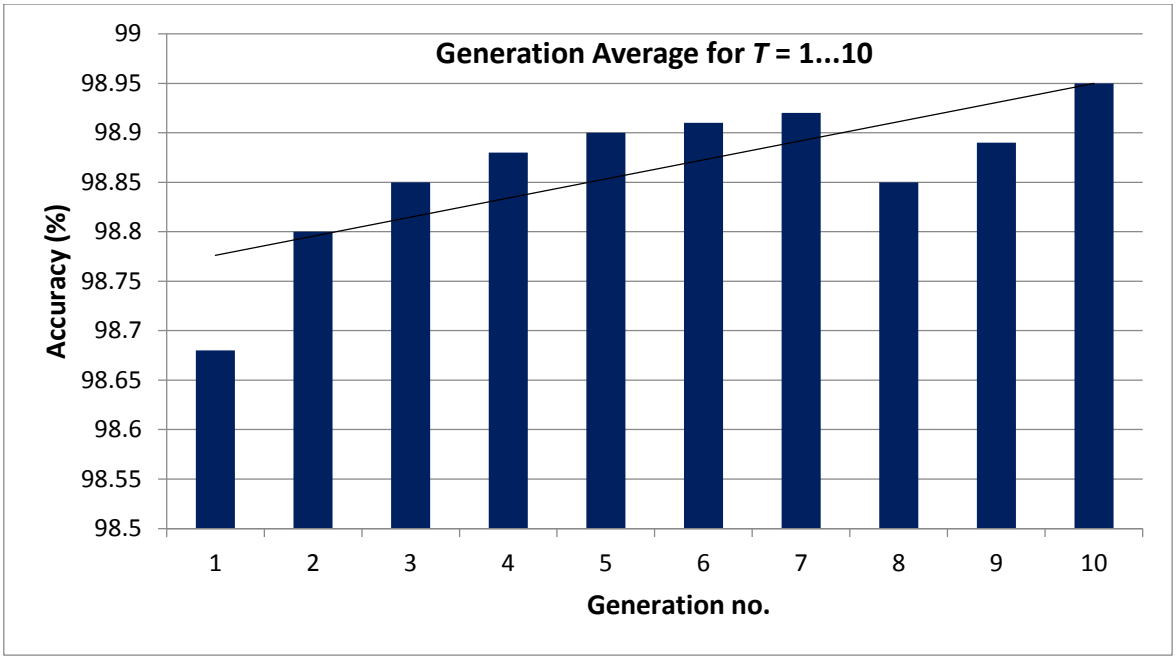


Figure 5.5: Graphical performance representation of the average classification accuracy of all the CNNs with reference to the generation in which they were computed

5.3.3.3.3 Analysis

As described in Section 5.3.3.2, the computational time required by the brute force approach was 1044 minutes, for 108 CNN models. In contrast, the 61 models trained and evaluated by the GA, was accomplished in 620 minutes, which is a greater than 40% improvement in computational time, for very similar results (see Table 5.8). From the results obtain from both the techniques, and the accuracy similarities between slightly different parameter choices, it can be concluded that the limited (filter numbers) model selection search space is full of redundant search points that need not be evaluated via a computationally exorbitant brute-force approach. However, using the genetic operations of selection, crossover, and mutation to search this space can promote the exploration of CNN models that achieve higher accuracies compared to unoptimized models, and accuracies very similar to brute-force computation, whist mitigating the need to pursue searching around poor performing solutions, subsequently resulting in a drastic reduction of unnecessary computational resources. The consequence of the stochastic components (random operations) of the GA guided search presented here is similar to a random search (Bergstra and Bengio, 2012), however, the selection and retention mechanism, guides the GA towards fitter individuals, which is an attribute that random search is devoid of. Furthermore, it's conceivable that the GA guided search can provide exponential model tuning speed gains as the architectural and parameter

complexity increases, and this leads to its application to evolve nine parameters of the same model, as detailed in the next section.

5.3.4 Extending the GA approach to a larger model selection search space

5.3.4.1 The search space and corresponding evolutionary process

Motivated by the results obtained by the GA in the previous section, Algorithm 5.1 was applied to select several other architectural choices as illustrated by CNN_GA_2.X in Table 5.9. The search space included filter size and number choices for all layers, the option to use or negate Dropout and type of activation function to use. If the decision to use Dropout was made for models that selected the freshly proposed SELU activation function, its associated Dropout version referred to as Alpha Dropout and its corresponding Dropout rates (see Section 2.2.4.2), were used, as recommended by the original SELU paper (Klambauer et al., 2017). For ELU's and ReLU's traditional Dropout was decided on or against and utilized. Given the larger search space, a larger population $N = 40 > N = 12$ was created and maintained for $T = 20$ generations of evolution. The percentage of the top performing models to retain was set as $r_p = 25\%$, whilst the probability of a poor performing model being retained was set to $p_r = 10\%$. Again, random crossover c_r was used to maintain the population size at $N = 40$, for each generation. This resulted in the training and evaluation of 40 models in the initial generation and 443 further models for the other nineteen generations. Thus, in total, $40 + 443 = 483$ CNNs were trained and evaluated during twenty runs of evolution. Like with CNN_GA_1.X models previously evaluated were not evaluated again to preserve computational resources. The effect of applying selection to the population, coupled by random retention, is illustrated in Table 5.10. Following the initial genetic run, the probability of mutation p_m was set high to a value of 0.3, to promote the generation of models with novel parameters.

Table 5.9: The architectural choices of the best model from the first evolutionary process, CNN_GA_1.1 and the choices exposed to the extended GA search, illustrated by CNN_GA_2.X

Layer	Hyperparameter	CNN_GA_1.1	CNN_GA_2.X
Convolutional layer 1	Number of filters	32	Choice between: {16, 32, 64}
	Kernel size	5*5	Choice between: {3*3; 4*4; 5*5}
Max pooling layer 1	Filter size	2*2	2*2
Convolutional layer 2	Number of filters	64	Choice between: {16, 32, 64}
	Kernel size	5*5	Choice between: {3*3; 4*4; 5*5}
Max pooling layer 2	Filter size	2*2	2*2
Fully connected layer 1	Number of filters	256	Choice between: {64, 128, 256}
	Dropout rate / Alpha Dropout rate	0.5	Choice between: {0, 0.25, 0.5, 0.75} / {0, 0.025, 0.05, 0.1}
Fully connected layer 2	Number of filters	64	Same as Fully connected layer 1
	Dropout rate / Alpha Dropout rate	0.5	Choice between: {0, 0.25, 0.5, 0.75} / {0, 0.025, 0.05, 0.1}
Global parameters	Activation Function - All layers except softmax layer	ReLU / Softmax in the softmax layer	Choice between: {ReLU; ELU; SELU} / Softmax
	Optimization	SGD	SGD

Table 5.10: The results of the genetic operations of selection, retention, and crossover

Generation	Retained no. of CNNs ($r_p = 35\%$)	Retained no. of poor performers ($p_r = 10\%$)	No. of crossed over CNNs	Population Size
T = 1	40	N/A	N/A	N = 40
T = 2	14	2	24	N = 40
T = 3	14	3	23	N = 40
T = 4	14	2	24	N = 40
T = 5	14	0	26	N = 40
T = 6	14	3	23	N = 40
T = 7	14	2	24	N = 40
T = 8	14	1	25	N = 40
T = 9	14	4	22	N = 40
T = 10	14	4	22	N = 40
T = 11	14	2	24	N = 40
T = 12	14	3	23	N = 40
T = 13	14	4	22	N = 40
T = 14	14	3	23	N = 40
T = 15	14	2	24	N = 40
T = 16	14	5	21	N = 40
T = 17	14	3	23	N = 40
T = 18	14	5	21	N = 40
T = 19	14	1	25	N = 40
T = 20	14	2	24	N = 40
Total	**306**	**51**	**443**	**N/A**

Considering the vast search space of CNN_GA_2.X, which totals $3^6 * 4^2 = 11664$ CNN models, using a brute force approach is unconceivable given the available resources, and thus was not considered. Instead, the baseline CNN_B1 and best performing model from CNN_GA_1.X were used as benchmarks to test the performance of CNN_GA_2.X. To measure quantifiable success and facilitate a meaningful analysis, naive accuracy improvement targets of 0.5% and 0.2%, over CNN_B1 and CNN_GA_1.1, were respectively, set. Furthermore, to facilitate comparison with the initial GA approach, training was also constrained to twenty epochs, and the same learning parameters (batch size, learning rate, optimizer, and initialization methods) maintained, however, these are optimized in the next chapter.

5.3.4.2 Results and analysis

The results of the top ten models from the final generation of evolution are illustrated in Figure 5.6 and Table 5.11, whilst the mean accuracy for each generation is depicted in Figure 5.8. Analysing the results, all the models that made it into the final generation of evolution favoured the use of Dropout (Hinton et al., 2012) and a general trend towards using ELU (Clevert et al., 2016) and SELU (Klambauer et al., 2017) activations, further highlighting the advantages of these recent advancements. Like with the brute-force and CNN_GA_1.X computational runs, there is noticeable redundancy in the accuracies obtained by the models in CNN_GA_2.X. The best performing model of the evolutionary process (CNN_GA_2.1) achieved an accuracy of 99.17%, which is greater than a 0.5% improvement over the unoptimized CNN_B1 and slightly under 0.2% improvement over the model optimized (CNN_GA_1.1) during the initial evolutionary process, both of which served as performance benchmarks. Thus, the targeted improvements set in Section 5.3.4.1, were achieved. Furthermore, performance comparisons between CNN_B1, CNN_BF, CNN_GA_1.1 and CNN_GA_2.1 are illustrated in Figure 5.7, and summarized by Table 5.12, which shows that CNN_GA_2.1 accomplishes the best results from all the experimentation thus far.

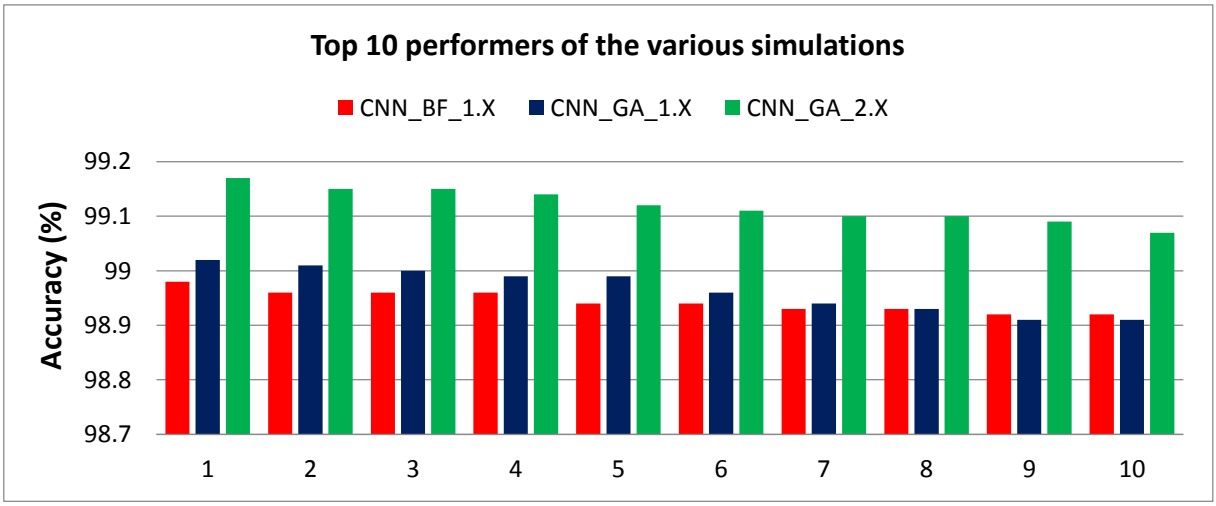


Figure 5.6: Graphical performance representation of the top models computed during the grid (CNN_BF_1.X), and first (CNN_GA_1.X) and second (CNN_GA_2.X) GA searches

Table 5.11: The architectural choices and respective accuracies of the top performing models computed during the evolutionary process

CNN	Filter no. C_1	Filter size C_1	Filter no. C_2	Filter size C_2	Filter no. FC_1	D-O use / rate	Filter no. FC_2	D-O use / rate	Act.	Acc. (%)
1	64	5	64	5	256	Use / 0.25	256	Use / 0.25	ReLU	99.17
2	64	5	64	5	256	Use / 0.25	256	Use / 0.5	ELU	99.15
3	64	5	64	5	256	Use / 0.25	256	Use / 0.25	ELU	99.15
4	64	5	128	5	256	Use / 0.25	256	Use / 0.5	ReLU	99.14
5	64	5	128	5	256	Use / 0.05	256	Use / 0.1	SELU	99.12
6	64	4	128	5	256	Use / 0.05	256	Use / 0.1	SELU	99.11
7	64	5	128	5	256	Use / 0.025	256	Use / 0.05	SELU	99.1
8	64	5	128	5	256	Use / 0.05	256	Use / 0.05	SELU	99.1
9	32	5	64	5	256	Use / 0.25	256	Use / 0.5	ELU	99.09
10	32	4	128	5	256	Use / 0.25	256	Use / 0.5	ELU	99.07

Table 5.12: Performance analysis between the base, brute-force, and different GA approaches

Model	Description	Acc. (%) - 20 Epochs	Acc. (%) - 50 Epochs	No. of para-meters
CNN_B1	Initial base model with no optimization	98.60	99.13	194,982
CNN_BF	Brute force filter number exploration	98.98	99.26	244,234
CNN_GA_1.1	GA optimized filter number exploration	99.02	99.27	311,594
CNN_GA_2.1	GA optimized 9 parameter exploration	99.18	99.30	434,890

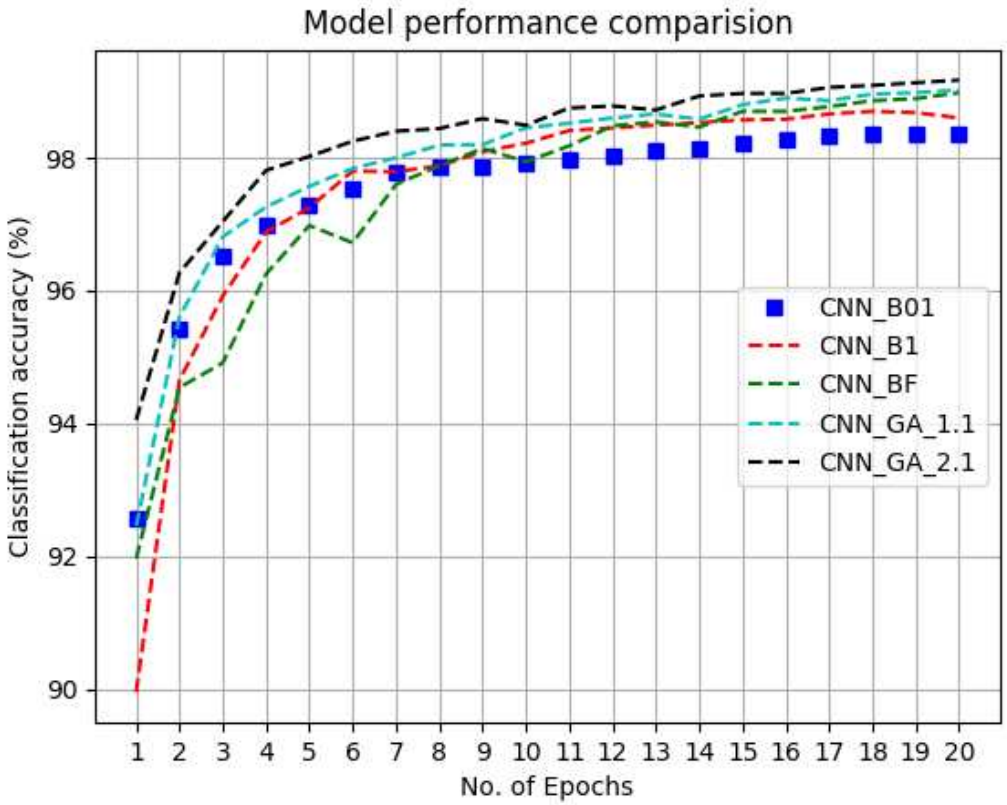


Figure 5.7: Performance comparison between CNN_B01, CNN_B1, CNN_BF, CNN_GA_1.1 and CNN_GA_2.1

Like described for CNN_GA_1.X, the computational burden of running the GA for several generation's limited experimentation, however, given the larger search space exposed to CNN_GA_2.X, the number of generations was set to $T = 20$ generations, which is double the number of iterations of CNN_GA_1.X . Notwithstanding this constraint, the general trend observed in Figure 5.8 is that the GA leads to higher mean accuracies for each succeeding generation, thereby ensuring that the algorithm promotes the development of fitter individuals, which is one of the three essential features of the evolutionary process (Yu and Gen, 2010).

Using the computation times from Section 5.3.3.2 as an estimation, the computational time required for a brute force approach for 11664 CNN models would be an astronomical $11664 * 9.66 = 1877.904$ hours. In contrast, the 483 models trained and evaluated by the GA, was accomplished in 139.25 hours, which is a greater than 92.5% improvement in computational time, for a greater than 0.5% increase in accuracy against baseline. Thus, even for large search spaces, using the evolutionary processes of selection, crossover and mutation, promotes the exploration of high performing CNN models and alleviates the need to pursue searching around poor performing solutions. Consequently, a drastic reduction in the

computational burden imposed by training CNNs to search for their near optimal characteristics can be achieved.

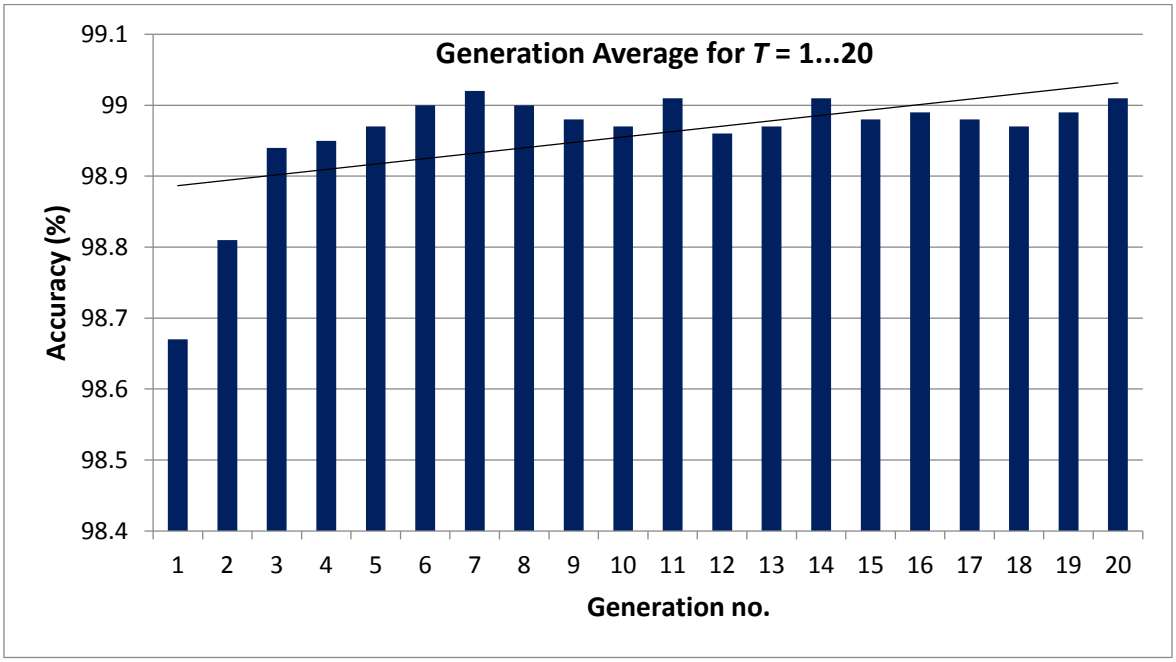


Figure 5.8: Graphical performance representation of the average classification accuracy of all the CNNs with reference to the generation in which they were computed

5.4 Conclusion

This chapter began by proposing and elaborating on a methodology to utilize the GA for selecting the near optimal architectural parameters of a population of CNNs. The methodology was applied to several simulations, before the results were discussed and analyzed. This chapter partitioned the search space of modern DCNNs into architectural and learning subspaces, and focused on applying the GA to maximize the classification accuracy of a population of CNNs with numerous architectural parameter choices, while maintaining the learning parameters fixed. This facilitated a small population size and reduced iterations of the GA, thus minimizing overall computation. In the next chapter, the optimization of the large search space imposed by DCNNs will continue and whilst the optimized architectural parameters obtained here will be fixed, Bayesian optimization will be used to optimize the learning subspace, thus culminating in a GA-Bayesian optimized model.

CHAPTER 6

GA-DCNN learning parameter optimization using a Bayesian approach

6.1 Chapter overview

This chapter continues with the computationally intensive task of DCNN model selection by extending the optimization techniques of the previous chapter by applying Bayesian optimization to the learning parameter subspace of the GA inspired models computed previously. It commences by describing and formalizing a methodology of applying Bayesian optimization to the learning parameter problem of the GA derived DCNNs from the previous chapter. Next, experiments using the Bayesian learning parameter search on top of the GA inspired architectures, are compared against a brute-force approach, before the method is extended to traverse a larger, continuous learning parameter subspace. Finally, the results are discussed and analyzed before the chapter is concluded.

6.2 Learning parameter selection with Bayesian optimization

6.2.1 Introduction

In this section Bayesian optimization is used to search for the near optimal learning parameters of the genetically derived CNNs. First, a methodology, culminating in the derivation of GA-Bayesian algorithm to obtain the near optimal global learning parameters on top of the GA selected model, is described. Thereafter, the SMBO algorithm is applied to the same image classification related application introduced in the previous chapter. For the remainder of this work, the Bayesian search is used to propose new global learning choices and hyperparameters, whilst the networks free parameters (i.e. the biases and weights) are computed through standalone traditional neural network training using backpropagation, following a similar procedure to the genetic search.

6.2.2 Methodology

As discussed in Section 5.2.2, the extensive search space imposed by the image classification DCNNs studied here was partitioned into separate architectural and learning subspaces, to promote the convergence of the stochastic GA search for the near optimal architectural parameters, with constrained computational resources. During the GA search, the learning

parameters were held fix, whilst the architectural choices were optimized. On the other hand, in the sections that follow, the GA optimized models are utilized as a fixed base to explore the learning subspace using Bayesian optimization, with the intention of further saving on computation and improving classification accuracy. Bayesian optimization, introduced in Section 4.5, explores the search space of a given domain, through deliberating exploring new areas and exploiting areas where good performance has been perceived, by using previous observations of the blackbox objective function to define the next point of observation. More specifically, the learning parameter optimization procedure for the given image classification task on the MNIST benchmark (LeCun et al., 1998) can be formalized by Algorithm 6.1. Here the unknown blackbox function denoted by f, represents the GA inspired DCNN model, with selectable learning parameters x, and stochastic and independently computed accuracy $y = f(\mathrm{x})$. In this context, the Bayesian algorithm is used to query f, for a designated set of learning parameters, denoted by the point x_{i+1} and the results are observed at a tentative observation point y_{i+1}, which represents the accuracy on the validation set, computed after training the model on the training set. The sequential queries of the Bayesian optimized learning subspace continue for a predetermined number of maximum iterations I, which is set by a specified computational budget. When I is reached, a final set of learning parameters, denoted by $\overline{\mathrm{x}}_{\mathrm{I}}$ is proposed by the Bayesian algorithm, and this represents the last GA derived architectural, and Bayesian optimized learning parameter recommendation. The best recommendation of the search is denoted by x_{B}, since it is possible that $\overline{\mathrm{x}}_{\mathrm{I}} \neq \mathrm{x}_{\mathrm{B}}$. The learning parameters that are optimized using Bayesian optimization are the weight initialization procedure, the batch size used for training the network, the optimization method and its associated learning rate. The GA inspired architectural parameters such as the number and size of the convolutional filters in the convolutional layers, the number of the fully connected filters in the fully connected layers, the activation function for the network and the decision to use or negate Dropout and the Dropout rate are held constant. For the proposed GA-Bayesian model selection procedure, the top performing GA derived model, is subject to a Bayesian search, where each iteration i of the Bayesian loop represents a set of learning parameters, such as those illustrated by Table 6.2 and Table 6.3, and thus an individual CNN model. Therefore, using the learning choices available in Table 6.2 as an example, a designated set of hyper parameters, denoted by the Bayesian search point x_{i+1}, may have the learning parameter array illustrated by Table 6.1, in which the shaded blocks represent the parameters that are held constant.

Algorithm 6.1: CNN model learning parameter selection, using Bayesian optimization

Algorithm 6.1: GA-CNN learning parameter selection using Bayesian optimization **Data:** Reference classification training set $\mathcal{D}_{tr}$ and test set $\mathcal{D}_{ts}$ **Bayesian optimization inputs:** The iteration number i for each sequential Bayesian search, the learning parameters x_0 for the initial Bayesian search, and subsequent parameters x_{i+1}, the observed classification accuracy $y = f(\mathrm{x})$ observed at an initial point y_0, and the succeeding points of observation, denoted by y_{i+1}.
1: Bayesian loop initialization: Get an initial learning parameter suggestion x_0 from the Bayesian loop
2: Train the initial CNN: Train the top performing GA derived model on the training set $\mathcal{D}_{tr}$, with the learning parameters x_0, suggested by the Bayesian loop
3: Observe the results: Validate the model on the test set $\mathcal{D}_{ts}$, at y_0 by observing the classification accuracy $y = f(\mathrm{x})$.
for each iteration, i = 1; 2; 3; 4; :: ; I, **repeat** the following steps:
4: Subject the results to a Bayesian search: Pass the observed accuracy $y = f(\mathrm{x})$, from step 3 to the Bayesian optimization loop, and obtain the next set of learning parameters
5: Model parameter fitment and training reiteration: Fit the GA selected DCNN from step 2, with the new Bayesian suggested learning parameters x_{i+1}, and retrain the model on $\mathcal{D}_{tr}$.
6: Evaluation: Repeat step 3, and observe the results on $\mathcal{D}_{ts}$ for each iteration i , at y_{i+1}
until the predetermined number of iterations I is complete
Result: The final Bayesian recommend learning parameter $\overline{\mathrm{x}}_{\mathrm{I}}$, and the best found Bayesian selection x_{B}, for the GA optimized architecture, and their associated classification accuracies y_I and y_B.

Table 6.1: Typical DCNN learning parameter distinction, and architectural similarity between GA inspired and Bayesian searched models

Layer / Global hyperparameter	Hyperparameter	Sample GA derived model selection	Sample Bayesian learning parameter search results
Convolutional layer 1	Number of filters	64	64
	Kernel size	5*5	5*5
Convolutional layer 2	Number of filters	128	128
	Kernel size	5*5	5*5
Fully connected layer 1	Number of filters	256	256
	Decision on Dropout use	Dropout	Dropout
	Dropout rate	0.25	0.25
Fully connected layer 2	Number of filters	256	256
	Decision on Dropout use	Dropout	Dropout
	Dropout rate	0.5	0.5
All layers except softmax layer	Activation function	ELU	ELU
Global learning parameters	Weight initialization technique	Glorot_uniform	LeCun_normal
	Batch size	64	128
	Optimization procedure	SGD	Adam
	Learning rate	0.01	0.002

6.2.3 Experimental setup

6.2.3.1 Development environment and data

All simulations were conducted using the hardware and data, introduced in Sections 5.3.2. Briefly, the same 8-core machine, with 16GB DDR4 RAM was used to train and validate all the required networks. Keras (Chollet et al., 2015) and TensorFlow (Abadi et al., 2015) were respectively used to build the DCNN models and carry out the required computation, whilst the other dependencies remained the same (see Section 5.3.2.1). Furthermore, the established MNIST dataset of handwritten digits (LeCun et al., 1998), split into a train-test spilt of 60000-10000 images, was used for all the experimentation in order to maintain consistency with the results obtained in the previous chapter.

6.2.3.2 Bayesian optimization implementation

Bayesian optimization was carried out using the SigOpt API[22], which is a Software as a Service (SaaS) optimization platform. SigOpt, which was derived from the open source Metrics Optimization Engine (MOE - Clark et al., 2014), utilizes an ensemble of state-of-the art Bayesian optimization techniques to find the optimal parameters for a wide variety of tasks, which amongst others, includes finding the optimal parameters of machine learning models (Snoek et al., 2012; Eggensperger et al., 2013; Dewancker et al., 2016a), such as the DCNNs studied here. More specifically, it aims to maximize a blackbox objective function f (Dewancker et al., 2016b), by utilizing Gaussian processes to model it (Dewancker et al., 2016c) and the expected improvement acquisition function to trade-off exploration and exploitation of the search space. Selecting the optimal Bayesian optimization technique for a specific problem is generally a non-intuitive process, and this is compounded by the fact that significant Bayesian optimization domain expertise is required to get suitable results. However, the SigOpt API mitigates this by incorporating several state-of-the-art Bayesian techniques around a straightforward API, thus promoting uncomplicated model tuning, with state-of-art results as analyzed in several recent works that focus on analyzing different Bayesian optimization techniques (Dewancker et al. 2016a, 2016b, 2016c).

[22] https://sigopt.com

6.2.4 Commissioning Bayesian learning parameter search for the GA selected model

6.2.4.1 Base model selection

The best performing model from the second stochastic GA search (see Section 5.3.4), CNN_GA_2.1, was used as the base model for the Bayesian learning parameter search. The optimized architecture of CNN_GA_2.1 consists of alternating layers of convolutional and max pooling operations, followed by three fully connected layers, as illustrated by Figure 6.1. Furthermore, with the exception of the softmax layer (see Section 5.2.2), the GA based model selection process favoured the use of the ReLU activation function (Nair and Hinton, 2010) for the convolutional and fully connected layers, whilst it also evolved towards Dropout (Hinton et al., 2012 - see Section 2.2.4.1 for a formal introduction) rates of $p = 0.25$ for the second and third fully connected layers. For the remainder of the simulations, these GA inspired architectural choices are held fix.

Figure 6.1: GA selected base model for Bayesian learning parameter optimization

6.2.4.2 Brute-force learning parameter search

To explore the learning subspace and use this exploration to verify the efficiency of the Bayesian optimization search for the near optimal learning parameters of the GA optimized architecture, an exhaustive grid search (brute-force approach) was first conducted. The learning subspace exposed to the grid and subsequent Bayesian search consisted of the following global parameters:

1) The kernel / filter initialization method (see Section 2.4);

2) The selection of the optimizer to use (see Section 2.3);

3) The associated learning rate of the optimizer (see Section 2.3).

Given the computational burden imposed by this approach, and the available hardware (see Section 5.3.2), the grid-search space was limited to 3-dimensions. It consisted of three

weight initialization methods, three optimizers and ten learning rates, as illustrated by Table 6.2.

Table 6.2: Learning parameter search space exposed to grid and Bayesian search

Global parameter	Search space – CNN_BA_1.X
Initialization method	Random_normal; Lecun_normal; He_normal; Glorio_normal
Learning rate	0.001 – 0.01 (step size = 0.01)
Optimizer	Adam; RMSprop; SGD

DCNNs are usually characterized by a high number of parameters, which can even run into the millions for large models (Krizhevsky et al., 2012; Simonyan and Zisserman, 2014; Szegedy, Liu, et al., 2015). Although this promotes their expressive ability (i.e. their ability to generalize), if their parameters and in particular their weights, are not initialized adequately the learning process can be hindered, due to issues with vanishing gradients (Bengio et al., 1994). The top performing model of the genetic search had a total of 434,890 parameters (see Table 5.12), and to avoid issues with vanishing gradients, it was initialized by the uniform scheme presented in (Glorio and Bengio, 2010), which is the default initialization method for both the convolutional and fully connected layers in Keras (Chollet et al., 2015). To further promote convergence with minimum computation, this learning hyperparameter is optimized by allowing the grid search to select from the initialization selection available in Table 6.2. Whilst these initialization schemes are introduced in Section 2.4 further details can be found in LeCun et al. (1998), Glorio and Bengio (2010) and He et al. (2015a).

Furthermore, notwithstanding the successes of SGD (Bottou, 1998, 2010), it also has some shortfalls such as the need to manually tune its learning rates, and its poor performance for sparse data, and this has lead others to develop other gradient based optimization schemes such as Adagrad (Duchi et al., 2011), AdaDelta (Zeiler, 2012), RMSprop[23], Adam and several of its variants such as AdaMax (Kingma and Ba, 2015) and Nadam (Dozat, 2016). These optimizers have adaptive learning rates and can lead to rapid convergence (Ruder, 2017), which is one of the fundamental objectives of the models discussed here. Thus, to obtain suitable accuracy improvements over the GA inspired models, without impacting too much on the computational load, the near optimal optimizer is searched from the choices available in Table 6.2, whilst other optimizers are explored in a second Bayesian search as detailed in

[23] Unpublished, proposed by Hinton, available from: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Section 6.2.5. Finally, the learning rate, which can be regarded as the most important hyperparameter to optimize (Bengio, 2012), of the selected optimizer, was also searched from the range illustrated in Table 6.2.

Thus, considering the learning hyperparameters available in Table 6.2 resulted in the training of $3 * 4 * 10 = 120$ CNN models, which were independently verified. To maintain consistency with the GA search, and to save on computation, the number of epochs for each model was constrained to twenty epochs (i.e. the model was shown the training set of 60000 images twenty times). On the hardware described in Section 5.3.2.1, this took a total of 1580 minutes, with an average of 13.17 minutes to train each network (training times varied since models with different optimizers and initialization methods took different times to train). The results of the top performing models from this brute force approach are illustrated in Table 6.3 and Figure 6.6. Following the architectural subspace search described in the previous chapter, significant redundancy amongst the different learning parameter combinations is also observed, highlighting the waste of computational resources consumed by an exhaustive grid search. However, similar to the grid search conducted to benchmark the GA search, the top performing accuracy is noted and will be used as a benchmark for the results that will be obtained from the Bayesian search for the near optimal learning parameters.

Table 6.3: The learning parameter choices and respective accuracies of the top performing models computed during the brute force run

Model	Initialization method	Optimization technique	Learning rate	Accuracy (%)
1	Random_normal	Adam	0.001	99.37
2	Lecun_normal	Adam	0.001	99.3
3	He_normal	Adam	0.002	99.26
4	Glorot_normal	Adam	0.001	99.21
5	He_normal	SGD	0.01	99.17
6	He_normal	Adam	0.001	99.15
7	Random_normal	Adam	0.002	99.15
8	He_normal	SGD	0.008	99.14
9	He_normal	SGD	0.007	99.12
10	Glorot_normal	SGD	0.009	99.12

6.2.4.3 Limited Bayesian parameter selection and comparison to grid search

6.2.4.3.1 Computational experiment

Next, the Bayesian approach described by Algorithm 6.1 was applied to select the learning parameters illustrated by CNN_BA_1.X in Table 6.2. To reduce the computational burden imposed by the brute-force approach, a constrained observational budget of $i = 70$ observations was set. This resulted in the training and evaluation of 70 individual models, with 70 associated recommendations for their learning parameters from the Bayesian optimization loop. These recommendations consisted of one initial recommendation x_0, 68 further exploration / exploitation traded-off recommendations, which included the best found selection x_B, and one final recommendation $\overline{\mathrm{x}}_\mathrm{I}$. To allow comparison with the brute force approach, training was also constrained to twenty epochs.

6.2.4.3.2 Experimental results and comparison to grid search

Table 6.4 and Figure 6.6 illustrate the performance of the top ten Bayesian optimized learning parameter recommendations, with their relative accuracies, whilst the search improvement history is illustrated in Figure 6.3. The best performing model of the Bayesian search (CNN_BA_1.1) achieved an accuracy of 99.38%, which is slightly higher (stochastic nature of weight initialization, gradient based optimization and random Dropout does not guarantee determinism) than the best model (CNN_BF_2.1) computed during the brute-force approach.

Table 6.4: The learning parameter choices and respective accuracies of the top performing models computed during the Bayesian search

Model	Initialization method	Optimization technique	Learning rate	Accuracy (%)
1	Random_Normal	Adam	0.002	99.38
2	Lecun_normal	Adam	0.001	99.36
3	Glorot_normal	Adam	0.009	99.34
4	Lecun_normal	Adam	0.001	99.33
5	Random_Normal	Adam	0.008	99.33
6	He_normal	Adam	0.001	99.31
7	Random_Normal	Adam	0.001	99.31
8	He_normal	Adam	0.004	99.29
9	Random_Normal	Adam	0.008	99.28
10	Glorot_normal	Adam	0.001	99.28

Although the accuracies and hyperparameter choices between the Bayesian and brute-force approach are not exactly identical, which can be attributed to the non-deterministic nature of the computation, there are various similarities and positive signs between the two approaches. Firstly, as illustrated by Tables 6.3 and 6.4 the top performing models of both approaches have only a 0.01% difference in accuracy, for very similar parameters (only the learning rate differed slightly). Secondly, amongst the top ten performing models of both approaches, 50% of the models share exactly the same initialization, optimizer and learning rate parameters (3/3). Furthermore, from the remaining models, 20% selected either the same initialization and optimizer or initialization and learning rate parameters (2/3), whilst 20% selected the same learning rates (1/3), leaving only one totally unique selection.

Moreover, as illustrated in Table 6.4 and following the brute-force approach (Table 6.3), there is notable redundancy amongst the different parameter combinations. Notwithstanding these redundancies and the stochastic nature of the computation, as expected, the top performing model found through Bayesian optimization (CNN_BA_1.1) achieved better accuracy compared to the baseline model (CNN_GA_2.1 – which only had its architecture and not its learning parameters optimized) and very similar accuracies compared to the brute force model, as illustrated by Figure 6.2, which illustrates the performance of CNN_GA_2.1, CNN_BF_2.1 and CNN_BA_1.1.

Generally, although experimentation over numerous Bayesian observations is usually constrained by the computational budget available, in this case, the Bayesian search was limited to $i = 70$ observations to force the total number of computed models to be below the total grid search limit (120 models). However, notwithstanding this limit, the Bayesian search achieves superior performance since it is still able to reach a near optimal value within this constraint, as illustrated by the best seen trace of these approaches, which is shown in Figure 6.3. Furthermore, by analysing the results of the Bayesian search, it can be noted that the most important parameter for optimal performance is the optimzer, as indicated by Figure 6.4, which illsutrates the Bayesian predicted complexity of the classification accuracy relative to perturbations on the different parameters being optimized. This is further illustrated by Figure 6.5, which illustrates the relative performance of the three optimizers that were searched. More specifically, the Bayesian search selected the Adam optimizer 36 out of 70 times, compared to the SGD and RMSprop optimizers, which were selected for the remaining 34 recommendations. This is significant, since it clearly highlights the Bayesian exploration / exploitation trade-off between areas of possibly high model performance (i.e. models that use

the Adam optimizer) and areas with significantly high levels of uncertainty (i.e. models that use the RMSprop and to a slightly lesser degree the SGD optimizer).

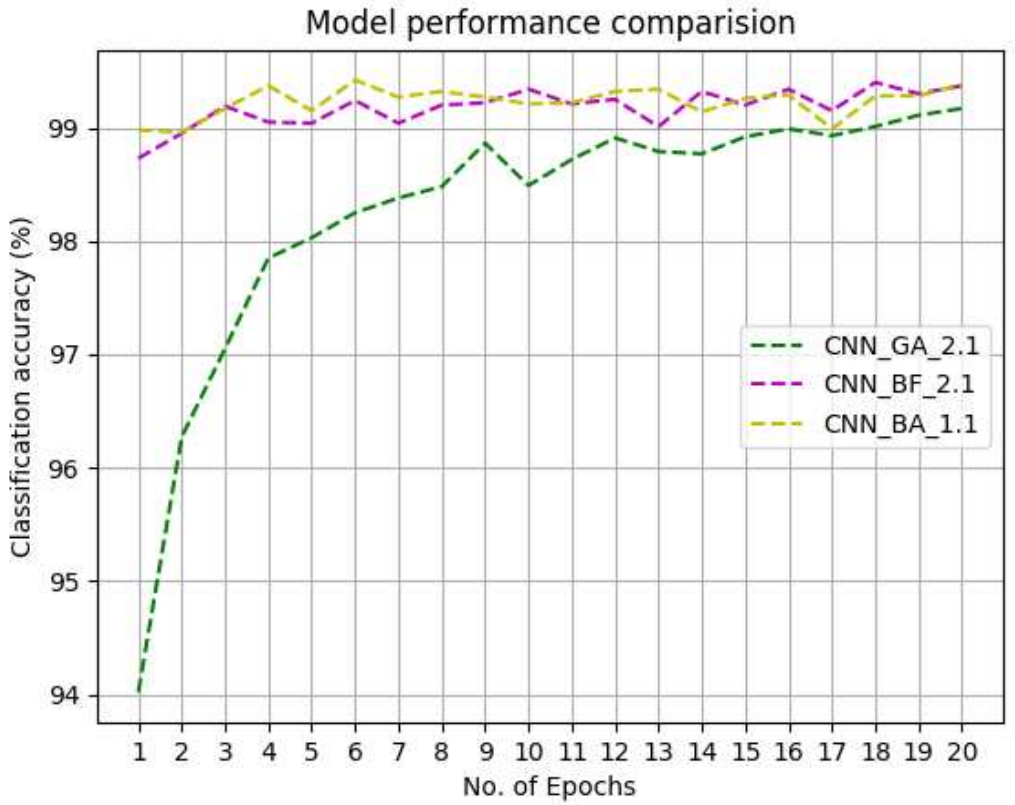


Figure 6.2: Performance comparison between CNN_GA_2.1, CNN_BF_2.1 and CNN_BA_1.1

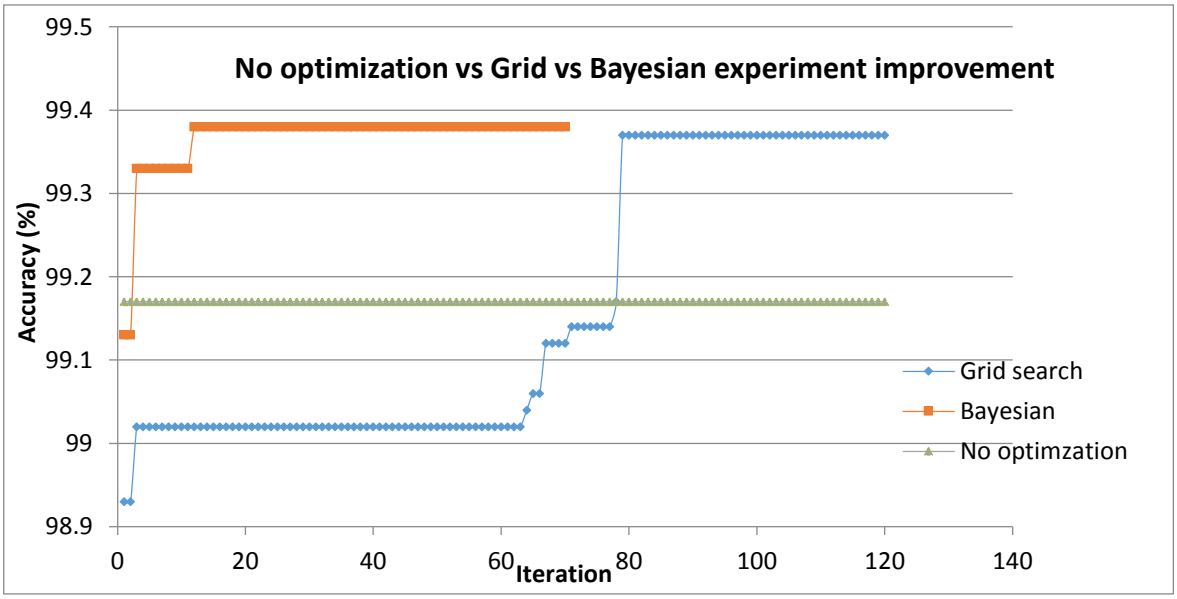


Figure 6.3: The best seen trace of the Bayesian and grid search, illustrating a greater than 0.2% improvement in accuracy, when compared to GA derived model without learning parameter optimization

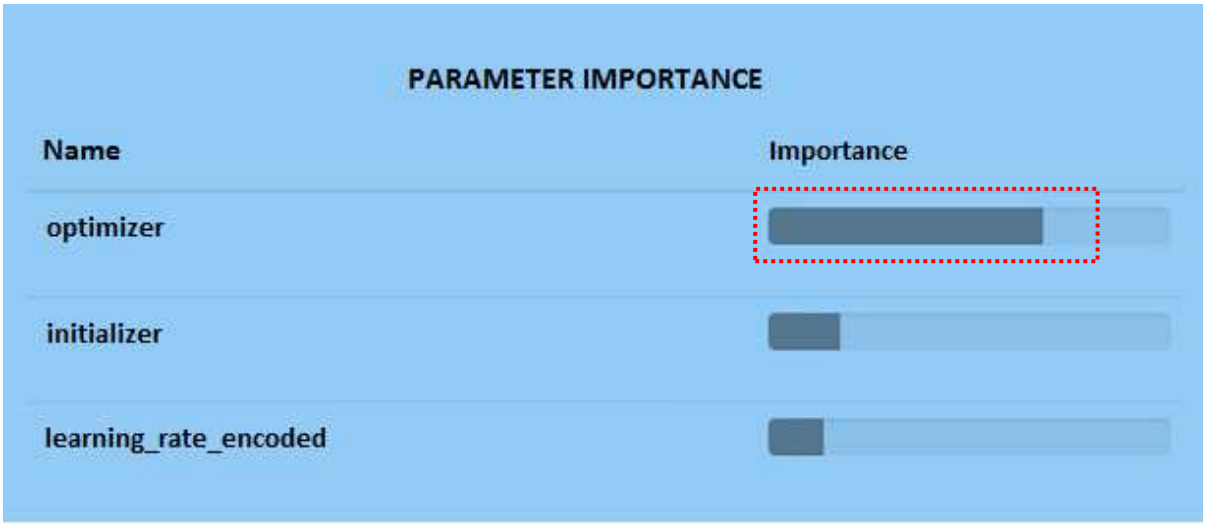


Figure 6.4: Graphical representation of parameter importance from the Bayesian search, illustrating the dominance of the optimizer

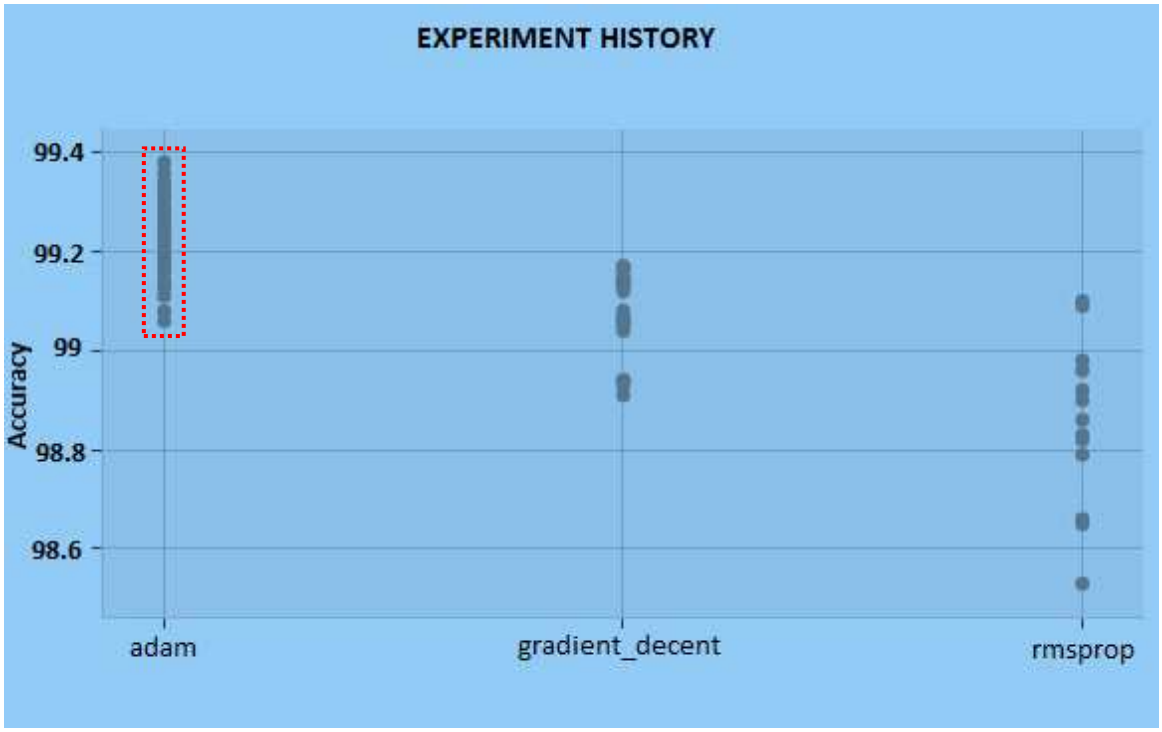


Figure 6.5: Graphical historical performance comparison of the different optimizer's during the Bayesian search, illustrating the specific dominance of the Adam optimizer

6.2.4.3.3 Analysis

As described in Section 6.2.4.2 the computational time required by the brute force approach was 1580 minutes, for 120 CNN models. In contrast, the 70 iterations of the Bayesian loop, took 950 minutes, which is a greater than 40% improvement in computational time, for very similar results (see Table 6.5). From the results obtained from both the techniques, and the

accuracy similarities between slightly different parameter choices, it can be concluded that like the architectural search space, the learning parameter selection search space is full of redundant search points that need not be evaluated via a computationally exorbitant brute-force approach. However, using Bayesian optimization to search this learning subspace can promote the exploration of CNN models that achieve higher accuracies compared to unoptimized models, and accuracies very similar to brute-force computation, whilst mitigating the need to pursue searching around poor performing solutions, subsequently resulting in a drastic reduction of unnecessary computational resources. The consequence of the Bayesian guided learning search, on top of the GA optimized architecture, described here is similar to a random search (Bergstra and Bengio, 2012), however, the Bayesian search is guided by its exploration / exploitation trade-off attribute, a feature random search is devoid of, and this prevents needless searching around areas of poor performance. Furthermore, it's conceivable that the GA-Bayesian guided search can provide exponential model tuning speed gains as the learning parameter space complexity increases (variables treated as continuous variables), something the GA search alone is not cable of achieving, and this leads to its application to search a more complex search space of the same model, as detailed in the next section.

Table 6.5: Performance analysis between the base, brute-force, and Bayesian approaches

Model name	**Model description**	**Accuracy (%)**	**Total learning subspace exploration time (minutes)**
CNN_GA_2.1	GA inspired architecture - no learning parameter optimization	99.17	N/A
CNN_BF_2.1	Brute force learning parameter search	99.37	1580
CNN_BA_1.1	Bayesian optimized learning parameter search	99.38	950

6.2.5 Extending the Bayesian approach to a larger model selection search space

6.2.5.1 The search space and corresponding Bayesian search

Motivated by the results obtained by the Bayesian learning parameter search in the previous section, Algorithm 6.1 was applied to select several other learning parameter choices. More specifically, the Bayesian algorithm was exposed to further optimization options, continuous learning rates and a large discrete range of batch sizes as illustrated by CNN_BA_2.X in Table 6.6. Given the larger search space, a larger observational budget of $i = 120 > i = 70$ Bayesian observations was set. This resulted in the training and evaluation of 120 individual models, with 120 associated recommendations for their learning parameters from the Bayesian optimization loop. These recommendations consisted of one initial recommendation x_0, 118 further exploration / exploitation traded-off recommendations, which included the best found selection x_B, and one final recommendation $\bar{x}_I$. To allow comparison with the brute force approach and initial Bayesian search (CNN_BA_1.X), training was also constrained to twenty epochs. The total required computational time, bottlenecked by model training, was 2410 minutes, with an average of just over 20 minutes / model.

Table 6.6: Learning parameter choices exposed to the second Bayesian search

Global parameter	Search space – CNN_BA_2.X
Initialization method	Random_normal; Lecun_normal; He_normal; Glorio_normal
Learning rate	0.001 – 0.01 (continuous)
Optimizer	RMSprop; SGD; Adagrad; Adam; Adamax; Nadam
Batch size	32 - 64 (with step size = 1)

The large discrete range[24] of the batch sizes, and the continuous learning rates, coupled with the categorical optimizers and initialization methods, illustrates the exponential dimensionality of the search space exposed to CNN_BA_2.X. Thus, using a brute force approach or even a genetic search is unconceivable, with the available hardware, and thus was not considered. However, the baseline CNN_GA_2.1 and best performing model from CNN_BA_1.X were used as benchmarks to test the performance of CNN_BA_2.X. Like with the second genetic search (see Section 5.3.4.1), to measure quantifiable improvement and

[24] In practice, large discrete parameter ranges behave like a continuous hyperparameters (Loshchilov and Hutter, 2016)

support a meaningful analysis, naive accuracy improvement targets of 0.25% and 0.1%, over CNN_GA_2.1 and CNN_BA_1.1, were respectively, set. Furthermore, to facilitate comparison with the initial Bayesian search, training was also constrained to twenty epochs, whilst the same architecture used for CNN_BA_1.X, as illustrated by Figure 6.1, was also maintained.

6.2.5.2 Results

Despite the computational resources imposing a computational budget constraint of 120 observations, the second Bayesian search achieved superior performance compared to the other techniques discussed here, since it was still able to reach near optimal values within this constraint, as illustrated by the best seen trace (improvement history) of these approaches, which is shown in Figure 6.8. Furthermore, the results of the top ten Bayesian optimized learning parameter recommendations, and their associated accuracies, are illustrated in Figure 6.6 and Table 6.7. Like with the brute-force and CNN_BA_1.X computational runs, there is noticeable redundancy in the accuracies obtained by the models in CNN_BA_2.X. The best performing model of the second Bayesian learning parameter search (CNN_BA_2.1) achieved an accuracy of 99.47%, which is a 0.3% improvement over the unoptimized CNN_GA_2.1 (its learning parameters were held fixed) and slightly under 0.1% improvement over the model optimized (CNN_BA_1.1) during the initial Bayesian search, both of which served as performance benchmarks. Thus, the targeted improvements set in the previous section, were achieved. Performance comparisons between CNN_GA_2.1, CNN_BA_1.1, and CNN_BA_2.1 are illustrated in Figure 6.7 and summarized by Table 6.8, which shows that CNN_BA_2.1/2 accomplishes the preeminent results from all the simulations discussed and analysed in Chapter's 5 and 6.

Table 6.7: The learning parameter choices and respective accuracies of the top performing models computed during the Bayesian search

Model	Batch size	Initialization method	Optimization technique	Learning rate	Accuracy (%)
1	46	Random_Normal	Adamax	0.001012	99.47
2	42	Random_Normal	Adamax	0.001012	99.47
3	46	He_normal	Adamax	0.01	99.45
4	60	Random_Normal	Adagrad	0.01	99.45
5	58	Random_Normal	Adamax	0.004725	99.44
6	48	Random_Normal	Adamax	0.001	99.44
7	59	Random_Normal	Adagrad	0.008025	99.44
8	40	Random_Normal	Adamax	0.01	99.44
9	60	Random_Normal	Adagrad	0.007699	99.43
10	40	Random_Normal	Adamax	0.001012	99.43

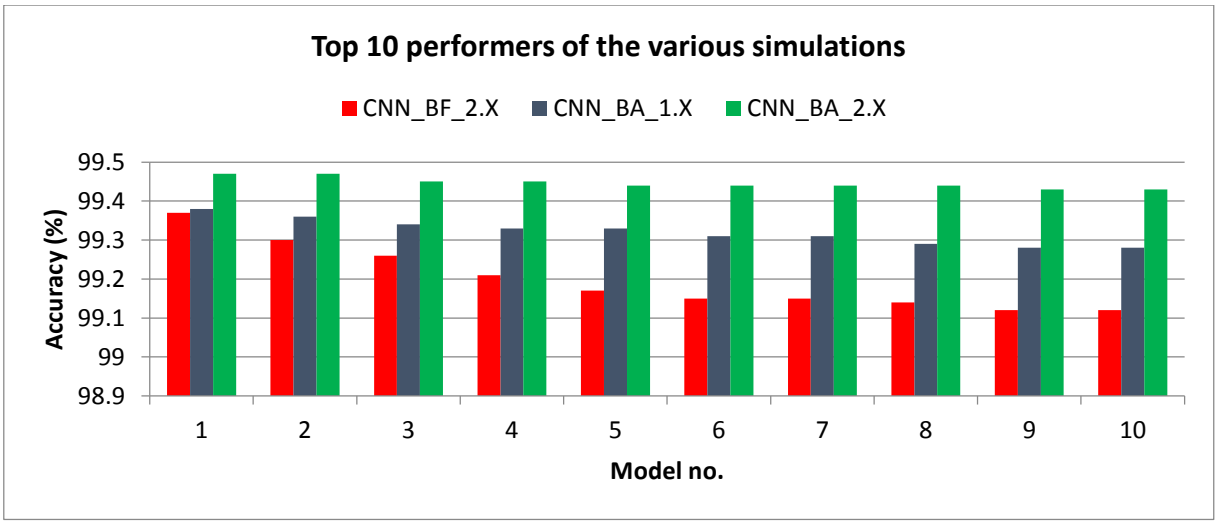


Figure 6.6: Graphical performance representation of the top models computed during the grid (CNN_BF_2.X), and first (CNN_BA_1.X) and second (CNN_BA_2.X) Bayesian searches

6.2.5.3 Analysis

By analysing the results of the second Bayesian search, it can be noted that the most important parameter for optimal performance is the optimzer (this result is consistant with the first Bayesian search), as indicated by Figure 6.9, which illsutrates the Bayesian predicted complexity of the classification accuracy relative to perturbations on the different parameters being optimized. This is further illustrated by Figure 6.10, which illustrates the relative

performance of the six optimizers that were searched. More specifically, the Bayesian search was dominated by the high performing Adamax optimizer, which was recommended 49 out of 120 times, and to a slightly lesser degree, the Adagrad optimizer, which was recommended 31 out of 120 times, compared to the other optimizers, which were selected for the remaining 40 recommendations. Moreover, from Figure 6.9, it can be further noted that following the optimizer, selecting a suitable batch size is also an important factor that needs to be considered, and to this end the Bayesian search pursued batch sizes ranging from 40 - 60 images, for near optimal performance, as illustrated by Figure 6.11. Similar results follow for the third most significant parameter, which is the initialization method. For this parameter, the Bayesian search favored random initialization 50% of the time, compared to the combined 50% of the other schemes. Figures 6.9 and 6.12 illustrate this. The results of this analysis is noteworthy, given that it clearly emphasizes the Bayesian exploration / exploitation trade-off between areas of possibly high model performance (i.e. models that use the Adamax / Adagrad optimizers, batch sizes within a certain band, and random initialization) and areas with significantly high levels of uncertainty (i.e. models that use the other optimizers, very small or large batch sizes, and the rest of the initialization techniques).

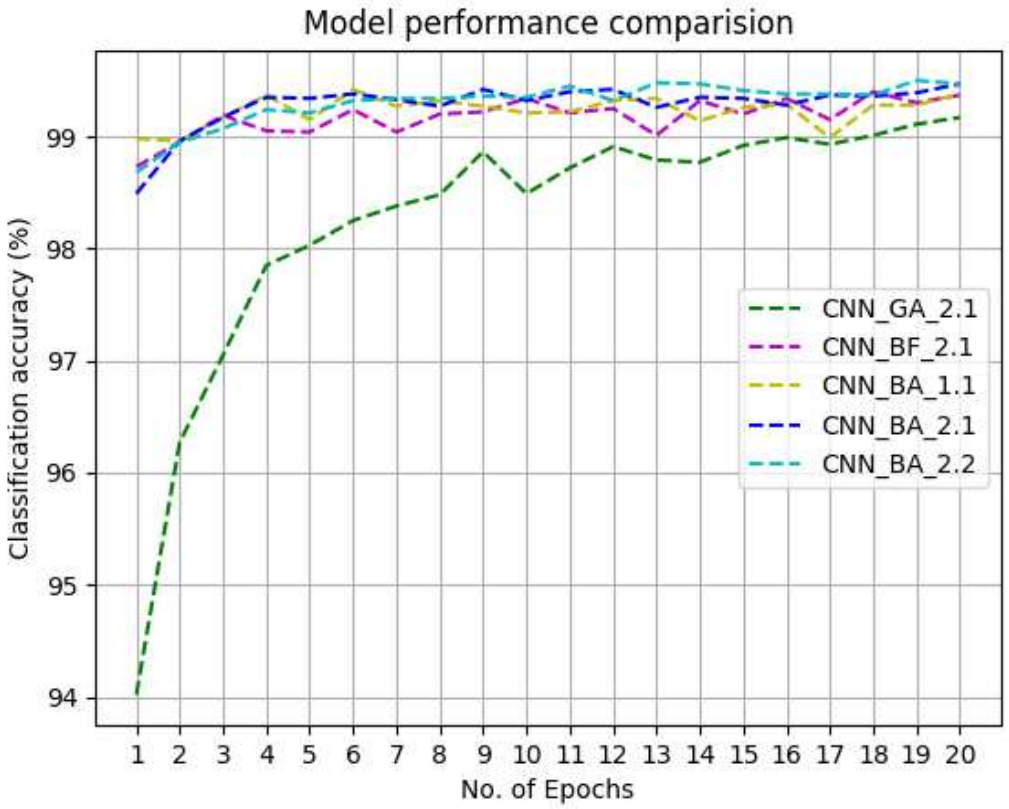


Figure 6.7: Performance comparison between CNN_GA_2.1, CNN_BF_2.1, CNN_BA_1.1, CNN_BA_2.1 and CNN_BA_2.2

Table 6.8: Performance analysis between the base, brute-force, and different Bayesian approaches

Model	Description	Acc. (%) - 20 Epochs
CNN_B1	GA inspired architecture - no learning parameter optimization	99.17
CNN_BF	Limited brute force learning parameter search	99.37
CNN_BA_1.1	Limited Bayesian optimized learning parameter search	99.38
CNN_BA_2.1/2	Extended Bayesian optimized learning parameter search	99.47

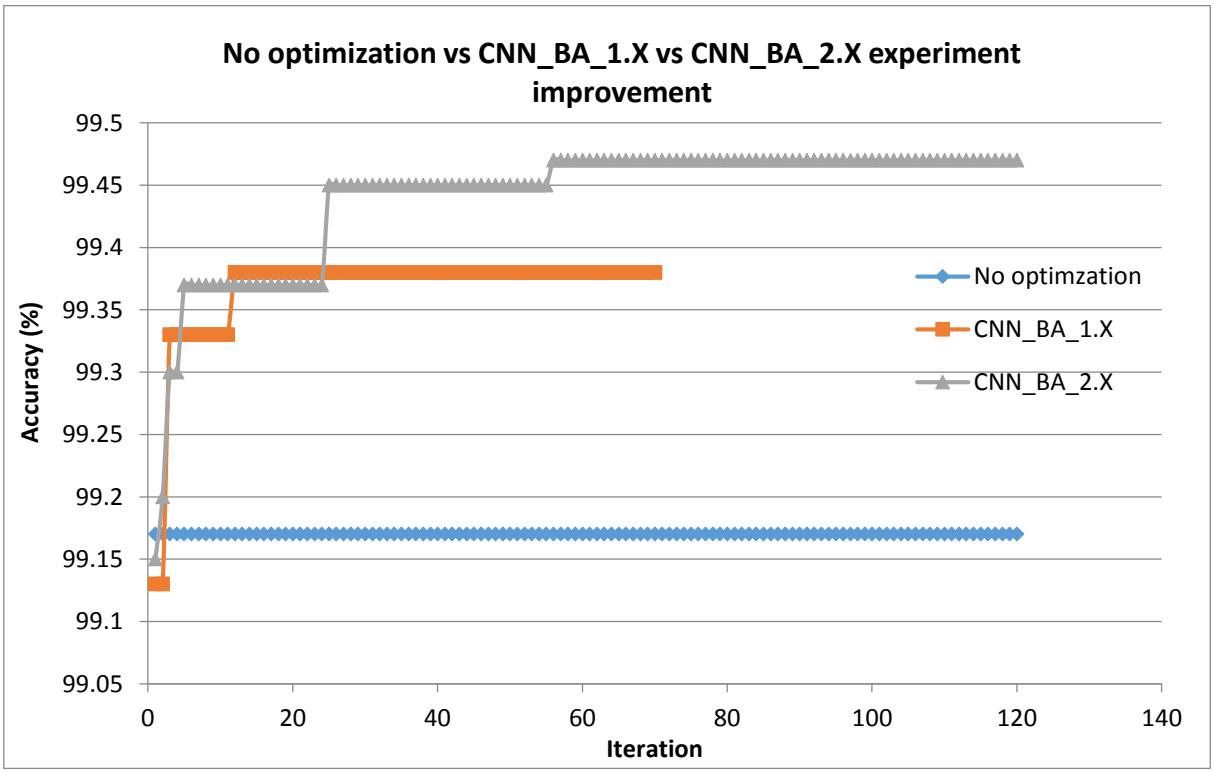


Figure 6.8: The best seen trace of the first (CNN_BA_1.X) and second (CNN_BA_2.X) Bayesian searches, illustrating a 0.3%, improvement in accuracy, when compared to GA derived model without learning parameter optimization

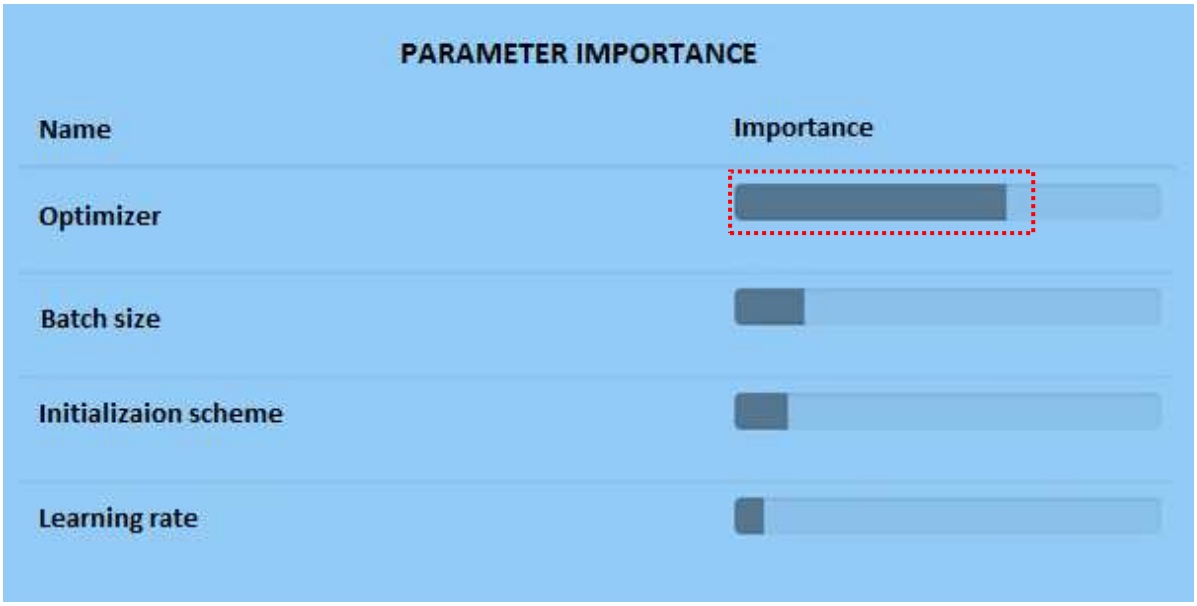


Figure 6.9: Graphical representation of parameter importance from the Bayesian search, illustrating the dominance of the optimizer

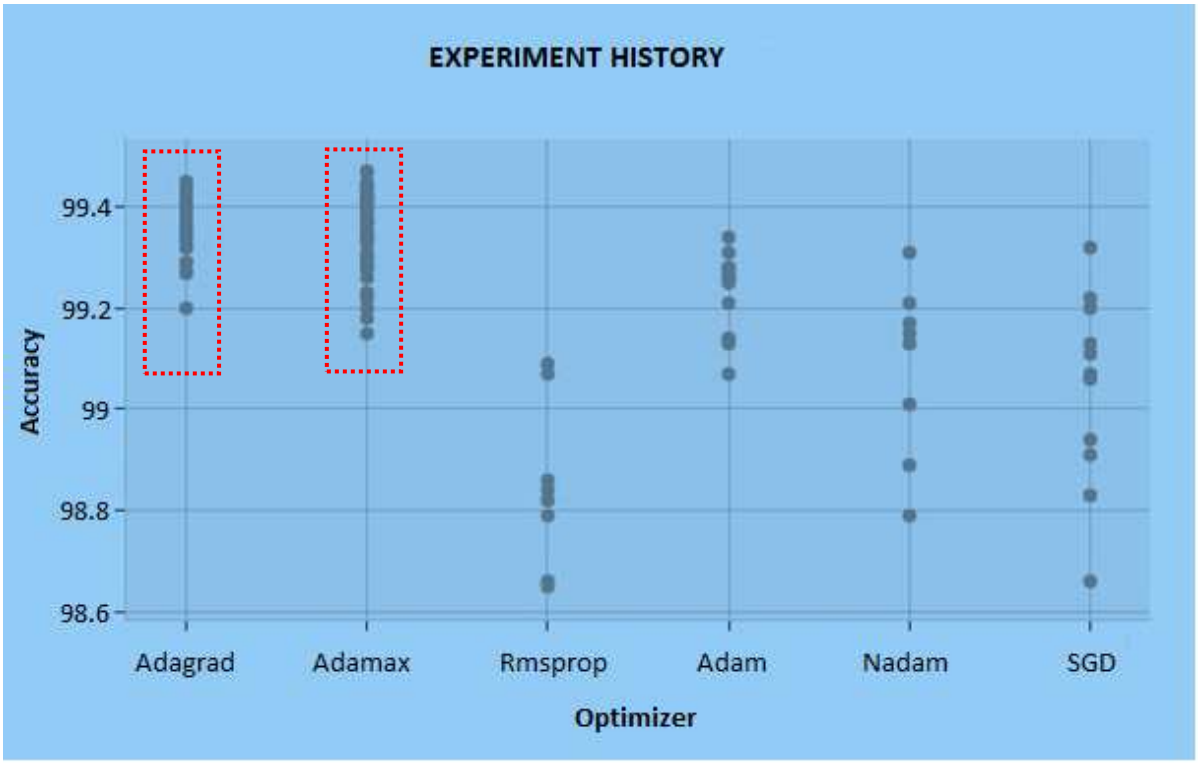


Figure 6.10: Graphical historical performance comparison of the different optimizer's used during the Bayesian search, illustrating the specific dominance of the Adamax and to a slightly lesser degree Adagrad optimizers

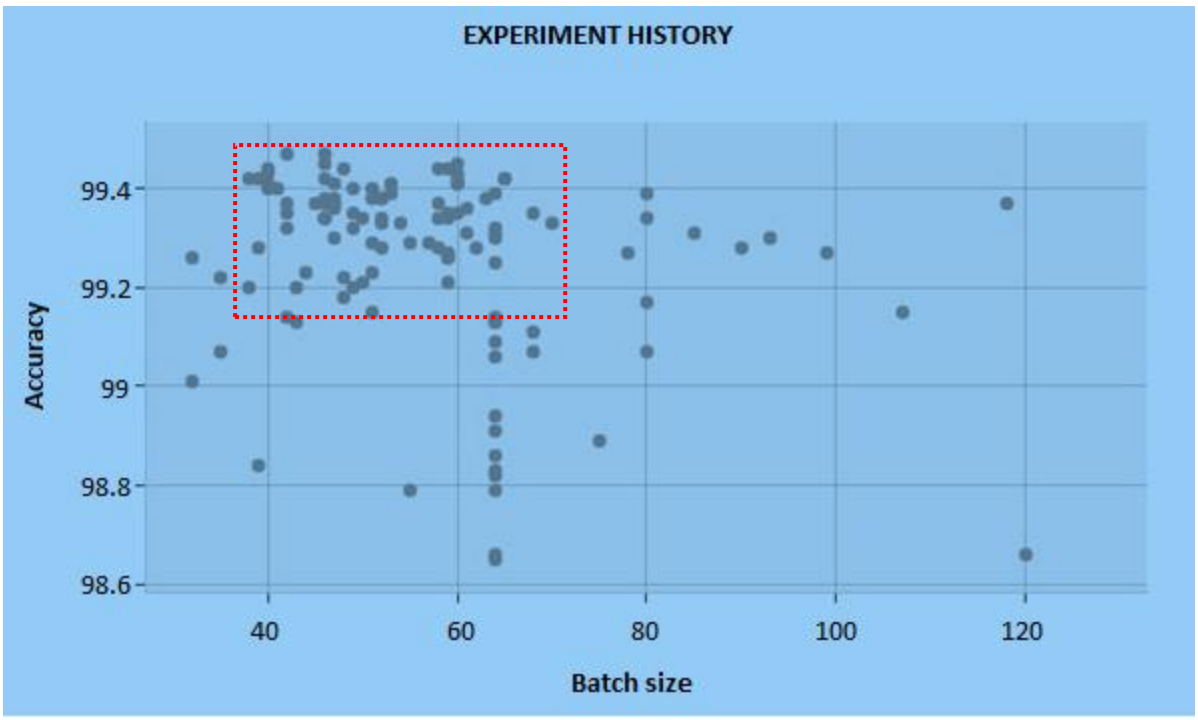


Figure 6.11: Graphical historical performance comparison of the different batch sizes used during the Bayesian search, illustrating the Bayesian preference towards batch sizes ranging from approximately 40 - 60, images per batch

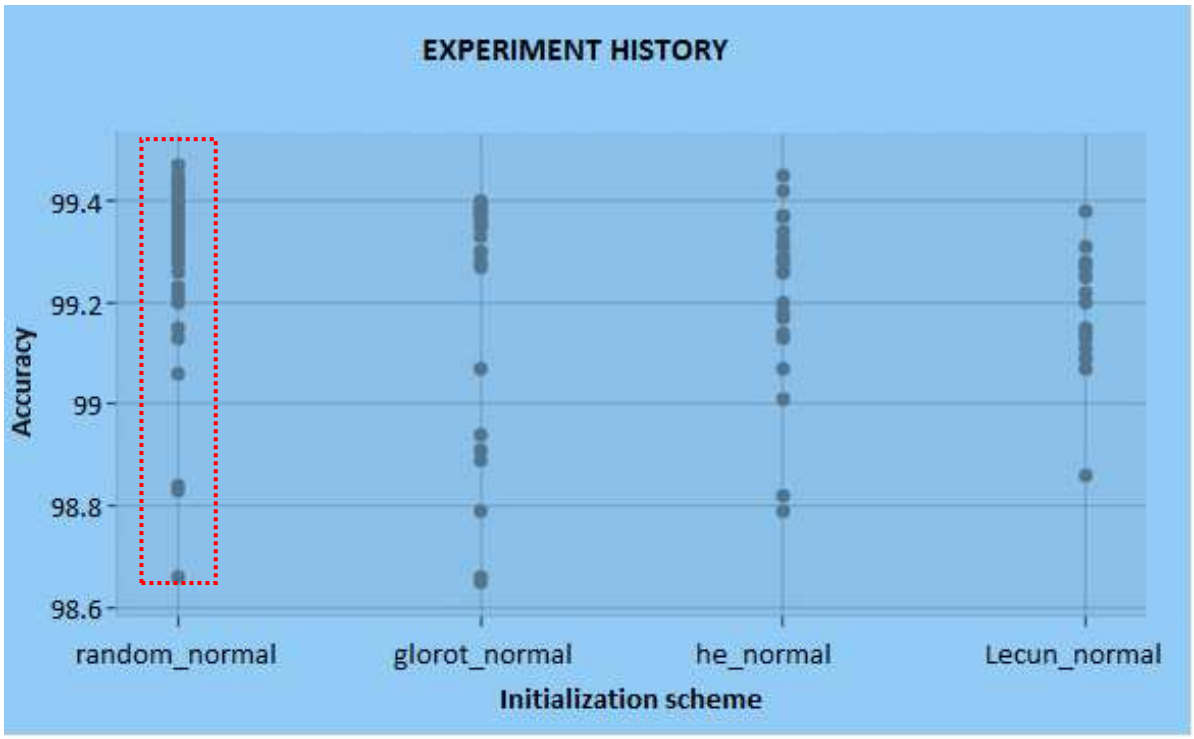


Figure 6.12: Graphical historical performance comparison of the different initialization schemes used during the Bayesian search, illustrating the Bayesian preference towards random kernel initialization

Given, the continuous nature of the learning rate parameter, a computation cost comparison to a grid search approach would be intractable, thus for comparison purposes, let's assume that the learning parameter was in fact a discrete parameter, like in the first grid search (see Table 6.2). This would require the training and evaluation of $10 * 4 * 6 * 32 = 7680$ models, totaling an exorbitant:

$$((7680\ models * 20\ minutes/model)/(60 minutes/hour)) = 2560\ hours$$

given an average computational time of 20 minutes / model (see Section 6.2.5.1). In contrast, the 120 models trained and evaluated during the Bayesian search, was accomplished in 40.2 hours, which is a greater than 98% improvement in computational time, for a 0.3% increase in accuracy against baseline. Thus, even for partitioned search spaces, using Bayesian optimization on top of a genetic search, can lead to the selection of high performing CNN models. Intuitively, the GA-Bayesian hybrid approach discussed here promotes the exploration of high performing CNN models and alleviates the need to pursue searching around poor performing solutions. It accomplishes this by separating the search space into architectural and learning parameter subspaces, to respectively promote the convergence of small evolutionary populations in minimal generations, and to facilitate a Bayesian search for continuous and continuous-like learning parameters. Subsequently, a radical reduction in the computational burden imposed by training CNNs to search for their near optimal characteristics can be achieved.

6.3 Conclusion

This chapter commences with a methodology to utilize the Bayesian optimization approach for selecting the near optimal learning parameters of the GA derived models from the previous chapter. The methodology, which culminated in the development of a GA-Bayesian algorithm, was applied to several simulations, before the results were discussed and analyzed. This chapter continued with the partitioning of the high dimensional search space of modern DCNNs into architectural and learning subspaces, and focused on applying Bayesian optimization on top of the GA to maximize the classification accuracy of the GA derived models, by considering numerous learning parameter choices, while maintaining the architectural parameters fixed. In the next chapter, the study is culminated with discussions on its limitations, recommendations, and directions for future work.

CHAPTER 7

Conclusions, limitations and future work

7.1 Summary

The aim of this dissertation was to investigate alternative ways to reduce the computational costs associated with DCNN model selection and to explore the related fundamentals. In order to accomplish this through the objectives set in Section 1.3, Chapter 2 introduced and established several fundamental concepts associated with DCNNs for image classification tasks. Specifically, it formally described their architectural details such as their convolutional, pooling, activation and Dropout layers (Hinton et al., 2012), and it also elaborated on several of their optimization and initialization techniques. Chapter 3 provided a succinct survey on DCNNs (see Rawat and Wang, 2017 - for a comprehensive survey), motivating for their application to the task of image classification, and whilst it highlighted its successes, it also exposed their representative open challenges. This led to a discussion on the problem of DCNN model selection and positioned the work presented in this dissertation. In this chapter, the search space imposed by modern convolutional networks was also separated into architectural and learning subspaces, and this led to a formalization of the hyperparameter search problem for these subspaces. Chapter 4 provided the background and motivation for the algorithms proposed to tackle the model selection and hyperparameter optimization challenge. It began by contextualizing and motivating for the presented evolutionary method by introducing EA's and GA's, and surveying the use of GA's for the task of evolving neural networks and convolutional networks in particular. Several random genetic operations such as selection, crossover and mutation were also introduced. In this chapter, Bayesian optimization was also formerly introduced as a prelude to the algorithm proposed in Chapter 6.

Chapter 5 build on the theoretical background provided in Chapter 4 and presented a stochastic GA to tackle the model selection problem. The random genetic operations introduced in Chapter 4 were tied together into a stochastic GA that was used to traverse the architectural search space. The proposed method was compared to a computationally exhaustive grid search approach, and several simulations and detailed analyses were presented to illustrate the computational superiority of the presented technique.

Chapter 6 presented a GA-Bayesian search technique to further improve the computational performance whilst searching for the optimal DCNN model. Building on the

background provided in latter parts of Chapter 4, a methodology, culminating in the derivation of a GA-Bayesian optimization algorithm, to further improve the performance of the GA presented in the previous chapter, was presented. Divergently from Chapter 5, in this chapter the architectural variables were held fixed, whilst the learning subspace variables were optimized through the GA-Bayesian technique. Comprehensive simulations and elaborate analyses were used to compare and contrast the presented method to other approaches, and its computational efficiency demonstrated. Thus, through these detailed chapters, the objectives set out in Section 1.3 to attain the overall aim of this study, were accomplished.

7.2 Conclusions and deductions

A grid search of the architectural search space dissects the possible topological model selection choices into equivalently sized (with regards to each dimension) grids, resulting in a uniform sampling of all the possible architectures, and this entails training and validating complete DCNN models at each intersection of the partitioned space. The downside is that this requires searching over an exponential number of dimensions, which is computationally exorbitant given the cost of training a single model. Whilst some of these computational costs can be reduced if a random search (Bergstra and Bengio, 2012) is conducted, random search is not directed towards top performing models. On the other hand, the stochastic GA presented in Chapter 5 uses several random operations to promote a type of random search of the architectural space, however, it poses the additional benefit of directing the search towards the selection of high performing models. Specifically, the randomness is accomplished by the random generation of the initial generation, and the stochastic breeding (crossover) and mutation mechanism of the GA, whilst the "direction" attributes that guides the search towards promoting the elite of the population of DCNNs is achieved by the selection and retention mechanisms, the latter of which is also stochastic in nature.

Whilst the GA search significantly reduced the computational costs of searching for the near optimal DCNN architecture as a direct result of its random and direction orientated attributes, searching the learning parameter subspace, which consists of continuous and continuous-like variables, also necessitated optimization for optimal classification performance. Like with the grid search of the architectural space, although uniform sampling is accomplished, the computational load is exorbitant, and the computational load is further compounded by the continuous nature of the batch size and learning rate parameters. Moreover, whilst the GA search did well for discrete parameters such as the filter sizes and numbers, it would be intractable to use for the continuous type variables. To assuage these

concerns, the space was searched using a Bayesian search on top of the GA derived models, and whilst the GA-Bayesian search exhibited randomness with regards to the initial Bayesian, and subsequent exploration related samples, it was focussed towards the exploitation of the top performing DCNN models. Subsequently, a significantly reduced computational load for searching the near optimal learning parameters was achieved. Furthermore, the exploration of the combined architectural and learning subspaces, conducted by the GA-Bayesian search highlighted the redundancies of the search space imposed on our modern models, further illustrating the need for techniques that traverse the space using automated intelligence, to mitigate unnecessary searching around low performing areas of the space, and this can result in notable computational gains, over well-established techniques such as the computationally exorbitant grid search approach.

Intuitively, the hybrid GA-Bayesian approach presented in this dissertation is similar to a random search, however, the search is guided by genetic operations which promote high performing models, and the exploration / exploitation trade-off attribute of the Bayesian component, which are features that random search is devoid of, and this alleviates the need to pursue searching around poor performing solutions. It accomplishes this by separating the search space into architectural and learning parameter subspaces, to respectively promote the convergence of small evolutionary populations in minimal generations, and to facilitate a Bayesian search for continuous and continuous-like learning parameters. Subsequently, a radical reduction in the computational burden imposed by training CNNs to search for their near optimal characteristics can be achieved.

7.3 Limitations and associated recommendations

Despite the promising results obtained, the presented GA-Bayesian technique suffers from a few drawbacks. Firstly, certain discrete architectural parameters such as the convolutional and pooling strides, the convolutional and pooling padding and the pooling filter sizes were held constant. Similarly, certain continuous learning hyperparameters such as the decay and epsilon parameters for the optimizers, and the standard deviation values for certain initialization schemes, were also held constant. Whilst it would be interesting to search for the unoptimized discrete architectural hyperparameters using the GA component of the search, the Bayesian component will be well suited for the unoptimized continuous learning parameters.

Secondly, the available computational constraints, limited experimentation to the MNIST dataset (LeCun et al., 1998), however, it's conceivable that with greater computational resources, the technique could be extended to natural image datasets such as

the CIFAR[25]-10, CIFAR-100 (Krizhevsky, 2009) and ImageNet (Russakovsky et al., 2015) benchmarks. Whilst to obtain acceptable classification performance on these datasets will require deeper models, with more convolutional, pooling, activation and Dropout layers, the inherent depth of these DCNNs will pose additional architectural choices to optimize. Although it's plausible that the stochastic GA presented in Chapter 5 could be generalized to search for these additional hyperparameters, the extension of the search space will require larger population sizes and a greater number of generations to ensure convergence, both of which will be realizable with the additional computational resources.

Thirdly, the simulations in Chapter 5 and 6 highlighted the inherent non-determinism (see Section 3.8.2) of our current machine learning models (Dalessandro, 2013). As discussed in Section 3.8.2, the source of the non-determinism can be attributed to several factors that include random weight initialization, random Dropout, certain hash-based operations and the stochastic nature of SGD (Bottou, 1998, 2010). To mitigate the non-determinism, one option is to run a single model multiple times and then average the results, an ensemble machine learning technique referred to as bootstrap aggregation or more commonly known as bagging. The downside of such an approach is the additional computational costs, which can be mitigated by additional computational resources.

Finally, the presented method was commissioned on the task of image classification, which has been dominated by DCNNs in recent years (Rawat and Wang, 2017). However, DCNNs have also been shown to work well for object detection and segmentation tasks (Girshick et al., 2014; Girshick, 2015; Ren, He, Girshick and Sun, 2016; He, Gkioxari, Doll´ar and Girshick, 2017). Further to requiring different architectural hyperparameters compared to image classification tasks, the techniques presented for object detection and segmentation tasks, such Regions with CNN features (R-CNN – Girshick et al., 2014), have their own hyperparameters that require tuning. These hyperparameters include region warping padding, bounding box-regression and non-maximum suppression threshold choices. Thus, given the computational gains of using the proposed GA-Bayesian searching strategy, and the fact that the method works independently of model training and validation, it is conceivable that the presented optimization methods can be generalized to search for the hyperparameters for object detection, segmentation and other similar computer vision related tasks. These recommendations are left for future work.

[25] Canadian Institute for Advanced Research

7.4 Further interesting directions and future work

Further to the recommendations described in the previous section, several other new directions presented themselves during the course of this dissertation. These are briefly summarized below.

7.4.1 Comparing the GA to PSO and other evolutionary algorithms

Whilst the proposed method used a stochastic GA for the architectural search, other modern metaheuristics such as the PSO algorithm (Kennedy and Eberhart, 1995) have certain advantages over GA's. These include more effective memory capabilities, and the ability to maintain greater diversity in the swarm compared to the population retention mechanisms of the GA (Coello, Lamont and Veldhuizen, 2007). Furthermore, recent work has seen PSO being applied to the DCNN model selection problem (Lorenzo et al., 2017; Ye, 2017). Thus, an interesting direction will be to compare the performance of PSO and other evolutionary algorithms to the stochastic GA for the task of model selection.

7.4.2 GA based model parameter training and structure evolution

During the course of the simulations, which formed the core of this dissertation, network training was conducted using backpropagation, optimized with some version of SGD. Thus, the models weights and biases were learnt using these traditional techniques. However, despite the heavy use of backpropagation and gradient based learning for image classification tasks, it is still unknown if these techniques are intrinsically flawed leading to some of the known challenges with DCNNs, such as those introduced in Section 3.6. Thus, whilst further work is required to understand the inner mechanisms of our current optimization techniques, an alternative would be to use the GA to train the CNN filters (Ouellette, Browne and Hirasawa, 2004; Lamos-Sweeney, 2012; Tirumala, 2014), thus using evolutionary procedures for both model selection and model parameter tuning.

Like discussed in Section 4.3, the presented GA was only used to search for the near optimal DCNN model, whilst the network structure was held fixed. Whilst others have considered utilizing GA's to search for the optimal DCNN structure (Xie and Yuille, 2017), they held certain architectural parameters fixed. Thus, another possible extension of the presented GA would be to adapt it to search for the optimal structure and architecture simultaneously. Naturally, this will require larger population sizes and more generations of evolution, and thus impose a greater computational burden.

7.4.3 Parallelization of the Bayesian search component

Whilst the GA component of the GA-Bayesian searching strategy can be parallelized (all the individual DCNNs in each generation can be computed concurrently), the Bayesian component of the search, which is based on Gaussian processes, is inherently sequential in nature, since Bayesian optimization is generally presented as a SMBO algorithm (Shahriari et al., 2016). Notwithstanding its success in finding the near optimal learning parameters with significantly reduced computational time, it may be pragmatically beneficial to run multiple Bayesian evaluations in parallel. We leave this, and the other directions described in this section, to future work.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... & Ghemawat, S. (2016). *TensorFlow: Large-scale machine learning on heterogeneous distributed systems.* arXiv preprint arXiv:1603.04467.

Al-Tabtabai, H., & Alex, A. P. (1999). Using genetic algorithms to solve optimization problems in construction. *Engineering Construction and Architectural Management*, *6*(2), 121-132.

Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., & de Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems* (pp. 3981-3989).

Bachman, P. (2016). An architecture for deep, hierarchical generative models. In D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, & R.Garnett (Eds.), *Advances in neural information processing systemsm, 29* (pp. 4826–4834). Red Hook, NY: Curran.

Baldi, P., and Sadowski, P. (2014). The dropout learning algorithm. *Artificial Intelligence*, *210*, 78–122.

Basu, S., Karki, M., DiBiano, R., Mukhopadhyay, S., Ganguly, S., Nemani, R., & Gayaka, S. (2016). *A theoretical analysis of deep neural networks for texture classification*. arXiv 1605.02699.

Bayer, J., Wierstra, D., Togelius, J., & Schmidhuber, J. (2009). Evolving memory cell structures for sequence learning. *Artificial Neural Networks–ICANN 2009*, 755-764.

Bengio, Y. (2009). Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, *2*(1), 1–127.

Bengio, Y. (2012). *Practical Recommendations for Gradient-Based Training of Deep Architectures.* arXiv preprint arXiv:1206.5533.

Bengio, Y. (2013). Deep learning of representations: Looking forward. In *Proceedings of International Conference on Statistical Language and Speech Processing* (pp. 1–37). Berlin: Springer.

Bengio, Y., Lamblin, P., Popovici, D., & Larochelle, H. (2006).Greedy layer-wise training of deep networks. In J. C. Platt, D. Koller, Y. Singer, & S. T. Roweis (Eds.), *Advances in neural information processing systems, 19* (pp. 2814–2822). Red Hook, NY: Curran.

Bengio, Y., Mesnard, T., Fischer, A., Zhang, S., & Wu, Y. (2017). STDP-compatible approximation of backpropagation in an energy-based model. *Neural Computation*, *29*(3), 555–577.

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, *5*(2), 157–166.

Bengio, Y., Thibodeau-Laufer, E., Alain, G., & Yosinski, J. (2014). Deep generative stochastic networks trainable by backprop. In *Proceedings of the 31st International Conference Machine Learning* (pp. 226–234). N.p.: International Machine Learning Society.

Bergstra, J., Bardenet, R., Bengio, Y., & Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems* (pp. 2546-2554).

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, *13*(Feb), 281-305.

Bergstra, J., & Cox, D. D. (2013). *Hyperparameter Optimization and Boosting for Classifying Facial Expressions: How good can a" Null" Model be?* arXiv preprint arXiv:1306.3476.

Bergstra, J., Yamins, D., & Cox, D. (2013). Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International Conference on Machine Learning* (pp. 115-123).

Bottou, L. (1998). Online learning and stochastic approximations. *On-Line Learning in Neural Networks, 17*(9), 142-177.

Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In *Proceedings of the International Conference on Computational Statistics (COMPSTAT)*, 177-186. Berlin: Physica-Verlag Heidelberg.

Boureau, Y., Ponce, J., & LeCun, Y. (2010). A theoretical analysis of feature pooling in visual recognition. In *Proceedings of the 27th International Conference on Machine Learning (ICML),* 111-118. N.p.: International Machine Learning Society.

Breiman, L. (2001). Random forests. *Machine learning*, *45*(1), 5-32.

Breiman, L., Friedman, J., Stone, C. J., & Olshen, R. A. (1984). *Classification and regression trees*. CRC press.

Brochu, E., Cora, V. M., & De Freitas, N. (2010). *A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning*. arXiv preprint arXiv:1012.2599.

Chatfield, K., Simonyan, K., Vedaldi, A., & Zisserman, A. (2014). *Return of the devil in the details: Delving deep into convolutional nets.* ArXiv Preprint arXiv:1405.3531.

Chellapilla, K., & Puri, S., & Simard, P. (2006). High performance convolutional neural networks for document processing. In *Proceedings of the 10th International Workshop on Frontiers in Handwriting Recognition.* Los Alamitos, CA: IEEE Computer Society.

Chollet, F. (2015). Keras. *URL http://keras. io*.

Chopra, S., Hadsell, R., & LeCun, Y. (2005). Learning a similarity metric discriminatively, with application to face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 539–546). Los Alamitos, CA: IEEE Computer Society.

Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., & LeCun, Y. (2015). The loss surfaces of multilayer networks. In *Proceedings 18th International Conference on Artificial Intelligence and Statistics* (pp. 192–204). www.jmlr.org/proceedings/papers/v38/choromanska15.pdf

Ciresan, D. C., Meier, U., Masci, J., Maria Gambardella, L., & Schmidhuber, J. (2011). Flexible, high performance convolutional neural networks for image classification. In *Proceedings of the International Joint Conference on Artificial Intelligence* (vol. 1, pp. 1237–1242). Menlo Park, CA: AAAI Press.

Ciresan, D.C., Meier, U., & Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 3642–3649). Red Hook, NY: Curran.

Claesen, M., & De Moor, B. (2015). *Hyperparameter Search in Machine Learning*. arXiv preprint arXiv:1502.02127.

Claesen, M., Simm, J., Popovic, D., Moreau, Y., & De Moor, B. (2014). E*asy hyperparameter search using Optunity.* arXiv preprint arXiv:1412.1114.

Clark, S., Liu, E., Frazier, P., Wang, J., Oktay, D., & Vesdapunt, N. (2014). *MOE: A global, black box optimization engine for real world metric optimization*. https://github.com/Yelp/MOE, 2014.

Clevert, D., Unterthiner, T., & Hochreiter, S. (2016). Fast and accurate deep network learning by exponential linear units (ELUs). In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, 1-14. N.p.: Computational and Biological Learning Society.

Coello, C. A. C., Lamont, G. B., & Van Veldhuizen, D. A. (2007). *Evolutionary algorithms for solving multi-objective problems* (Vol. 5). New York: Springer.

Criminisi, A., Shotton, J., & Konukoglu, E. (2011). Decision forests for classification, regression, density estimation, manifold learning and semi-supervised learning. *Microsoft Research Cambridge, Tech. Rep. MSRTR-2011-114*, *5*(6), 12.

Dalessandro, B. (2013). Bring the noise: Embracing randomness is the key to scaling up machine learning algorithms. *Big Data*, *1*(2), 110-112.

Dawson, C. (2002). Practical research methods: A user friendly guide to research. Oxford, UK: How to Books Ltd.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., . . . Le, Q. V. (2012). Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems, 25* (pp. 1223–1231). Red Hook, NY: Curran.

Decoste, D., & Scholkopf, B. (2002). Training invariant support vector machines. *Machine Learning, 46(1–3), 161–190.*

Deng, L. (2012). The MNIST database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, *29*(6), 141-142.

Deng, L. (2014). A tutorial survey of architectures, algorithms, and applications for deep learning. *APSIPA Transactions on Signal and Information Processing*, *3*(2), 1–29.

Deng, L., & Yu, D. (2014). Deep learning: Methods and applications. *Foundations and Trends in Signal Processing*, *7*(3–4), 197–387.

Dernoncourt, F., & Lee, J. Y. (2016, December). Optimizing neural network hyperparameters with gaussian processes for dialog act classification. In *Spoken Language Technology Workshop (SLT), 2016 IEEE* (pp. 406-413). IEEE.

Desell, T. (2017*). Large Scale Evolution of Convolutional Neural Networks Using Volunteer Computing*. arXiv preprint arXiv:1703.05422.

Dettmers, T. (2016). 8-bit approximations for parallelism in deep learning. In *Proceedings of the 4th International Conference on Learning Representations* (pp. 1–14). N.p.: Computational and Biological Learning Society.

Dewancker, I., McCourt, M., Clark, S. (n.d.). *Bayesian Optimization Primer* [White paper]. Retrieved August 22, 2017, from SigOpt: https://sigopt.com/static/pdf/SigOpt_Bayesian_Optimization_Primer.pdf

Dewancker, I., McCourt, M., Clark, S., Hayes, P., Johnson, A., & Ke, G. (2016a). *A Stratified Analysis of Bayesian Optimization Methods.* arXiv preprint arXiv:1603.09441.

Dewancker, I., McCourt, M., Clark, S., Hayes, P., Johnson, A., & Ke, G. (2016b). *Evaluation System for a Bayesian Optimization Service.* arXiv preprint arXiv:1605.06170.

Dewancker, I., McCourt, M., Clark, S., Hayes, P., Johnson, A., & Ke, G. (2016c). A Strategy for Ranking Optimization Methods using Multiple Criteria. In *ICML Workshop on Automatic Machine Learning* (pp. 11-20).

Ding, S., Li, H., Su, C., Yu, J., & Jin, F. (2013). Evolutionary artificial neural networks: a review. *Artificial Intelligence Review*, 1-10.

Dozat, T. (2016). Incorporating Nesterov Momentum into Adam. In *ICLR Workshop*. 2013-2016.

Dreyfus, S. (1962). The numerical solution of variational problems. *Journal of Mathematical Analysis and Applications*, *5*(1), 30–45.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, *12*(Jul), 2121-2159.

Eggensperger, K., Feurer, M., Hutter, F., Bergstra, J., Snoek, J., Hoos, H., & Leyton-Brown, K. (2013). Towards an empirical foundation for assessing Bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*. Volume 10.

Elbeltagi, E., Hegazy, T., & Grierson, D. (2005). Comparison among five evolutionary-based optimization algorithms. *Advanced engineering informatics*, *19*(1), 43-53.

Farabet, C., Couprie, C., Najman, L., & LeCun, Y. (2012). *Scene parsing with multiscale feature learning, purity trees, and optimal covers*. arXiv 1202.2160.

Fernando, C., Banarse, D., Reynolds, M., Besse, F., Pfau, D., Jaderberg, M., ... & Wierstra, D. (2016, July). Convolution by evolution: Differentiable pattern producing networks.

In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference* (pp. 109-116). ACM.

Floreano, D., Mattiussi, C., & Intelligence, Bio-Inspired Artificial Intelligence (2008). Theories, Methods, and Technologies.

Friedrichs, F., & Igel, C. (2005). Evolutionary tuning of Multiple SVM Parameters. *Neurocomputing, 64*, 107-117.

Fukushima, K. (1979). Self-organization of a neural network which gives position invariant response. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence* (vol. 1, pp. 291–293). San Francisco: Morgan Kaufmann.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, *36*(4), 193–202.

Fukushima, K., & Miyake, S. (1982). Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position. *Pattern Recognition*, 15(6), 455–469.

Girshick, R. (2015, December). Fast R-CNN. In *2015 IEEE International Conference on Computer Vision (ICCV).* (pp. 1440-1448). IEEE.

Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 580–587). Red Hook, NY: Curran.

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (pp. 249-256).

Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics* (pp. 315–323).www.jmlr.org/proceedings/papers/v15/glorot11a/glorot11a.pdf

Goldberg, D. E. (1989). Genetic algorithms in search, optimization, and machine learning, 1989. *Reading: Addison-Wesley*.

Gomez, F., Schmidhuber, J., & Miikkulainen, R. (2008). Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, *9*(May), 937-965.

Gong, Y.,Wang, L., Guo, R., & Lazebnik, S. (2014, September). Multi-scale orderless pooling of deep convolutional activation features. In *Proceedings of the European Conference on Computer Vision* (pp. 392–407). Berlin: Springer.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. Cambridge, MA:MIT Press.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., . . . Bengio, Y. (2014). Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, & K. Q.Weinberger (Eds.), *Advances in neural information processing systems, 27* (pp. 2672–2680). Red Hook, NY: Curran.

Goodfellow, I. J., Shlens, J., & Szegedy, C. (2015). Explaining and harnessing adversarial examples. In *Proceedings of the 3rd International Conference on Learning Representations* (pp. 1–11). N.p.: Computational and Biological Learning Society.

Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A. C., & Bengio, Y. (2013). Maxout networks. In *Proceedings of the 30th International Conference Machine Learning (ICML),* 1319-1327. N.p.: International Machine Learning Society.

Grierson, D. E., & Khajehpour, S. (2002). Method for conceptual design applied to office buildings. *Journal of computing in civil engineering*, *16*(2), 83-103.

Gu, S., & Rigazio, L. (2014). *Towards deep neural network architectures robust to adversarial examples*. arXiv 1412.5068.

Hadsell, R., Sermanet, P., Ben, J., Erkan, A., Scoffier,M., Kavukcuoglu, K., . . . LeCun, Y. (2009). Learning long-range vision for autonomous off-road driving. *Journal of Field Robotics*, *26*(2), 120–144.

Hamze, F., Wang, Z., & de Freitas, N. (2013). Self-avoiding random dynamics on integer complex systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, *23*(1), 9.

Han, S.,Mao, H., & Dally, W. J. (2016). Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding. In *Proceedings of the 3rd International Conference on Learning Representations* (pp. 1–14).N.p.: Computational and Biological Learning Society.

Hansen, N., & Ostermeier, A. (2001). Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, *9*(2), 159-195.

He, K., & Sun, J. (2015). Convolutional neural networks at constrained time cost. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 5353–5360). Red Hook, NY: Curran.

He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). *Mask R-CNN*. arXiv preprint arXiv:1703.06870.

He, K., Zhang, X., Ren, S., & Sun, J. (2015a). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV),* 1026-1034. Red Hook, NY: Curran Associates.

He, K., Zhang, X., Ren, S., & Sun, J. (2015b). *Deep residual learning for image recognition.* ArXiv Preprint arXiv: 1512.03385.

Hegazy, T. (1999). Optimization of construction time-cost trade-off analysis using genetic algorithms. *Canadian Journal of Civil Engineering*, *26*(6), 685-697.

Hennig, P., & Schuler, C. J. (2012). Entropy search for information-efficient global optimization. *Journal of Machine Learning Research*, *13*(Jun), 1809-1837.

Hernández-Lobato, J. M., Hoffman, M. W., & Ghahramani, Z. (2014). Predictive entropy search for efficient global optimization of black-box functions. In *Advances in neural information processing systems* (pp. 918-926).

Hinton, G. E. (2012). A practical guide to training restricted Boltzmann machines. In *Neural networks: Tricks of the trade* (pp. 599-619). Springer, Berlin, Heidelberg.

Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, *313*(5786), 504–507.

Hinton, G. E., Osindero, S., & Teh, Y. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, *18*(7), 1527–1554.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012). *Improving neural networks by preventing co-adaptation of feature detectors.* ArXiv Preprint arXiv: 1207.0580.

Hinton, G., Vinyals, O., & Dean, J. (2015). *Distilling the knowledge in a neural network*. arXiv: 1503.02531.

Holland, J. H. (1975). Adaptation in Natural and Artificial Systems, Univ. of Mich. Press. *Ann Arbor*.

Hsu, C.W., Chang, C.C., & Lin, C.J. 2003. A practical guide to support vector classification. *Department of Computer Science, National Taiwan University. Tech. rep.*

Huang, C. L., & Wang, C. J. (2006). A GA-based feature selection and parameters optimization for support vector machines. *Expert Systems with applications*, *31*(2), 231-240.

Huang, F. J., & LeCun, Y. (2006). Large-scale learning with SVM and convolutional nets for generic object categorization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 284–291). Red Hook, NY: Curran.

Hubel, D. H., & Wiesel, T. N. (1959). Receptive fields of single neurons in the cat's striate cortex. *Journal of Physiology*, *148*(1), 574–591.

Hubel, D. H., & Wiesel, T. N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *Journal of Physiology*, *160*(1), 106–154.

Hutter, F., Hoos, H. H., Leyton-Brown, K., & Stützle, T. (2009). ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, *36*(1), 267-306.

Iandola, F. N., Moskewicz, M. W., Ashraf, K., Han, S., Dally, W. J., & Keutzer, K. (2016). *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size*. arXiv: 1602.07360.

Ioffe, S., and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference Machine Learning* (pp. 448–456). N.p.: International Machine Learning Society.

Ivakhnenko, A. G., & Lapa, V. G. (1966). *Cybernetic predicting devices.* New York: CCM Information Corp.

Jaderberg, M., Simonyan, K., and Zisserman, A. (2015). Spatial transformer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Advances in neural information processing systems, 28* (pp. 2017–2025). Red Hook, NY: Curran.

Jarrett, K., Kavukcuoglu, K., & Lecun, Y. (2009). What is the best multi-stage architecture for object recognition? In *Proceedings of the IEEE International Conference on Computer Vision (ICCV),* 2146-2153. Red Hook, NY: Curran Associates.

Jones, D. R., Schonlau, M., & Welch, W. J. (1998). Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, *13*(4), 455-492.

Karpathy, A. (2016). *CS231n: Convolutional neural networks for visual recognition*. http://cs231n.github.io/classification/

Karpathy, A., & Fei-Fei, L. (2016). Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 3128–3137). Red Hook, NY: Curran.

Kennedy, J., & Eberhart, R. C. (1995, October).Anew optimizer using particle swarm theory. In *Proceedings of the 6th International Symposium on Micro Machine and Human Science* (pp. 39–43). Piscataway, NJ: IEEE.

Kingma, D. P., & Welling, M. (2014). *Auto-encoding variational Bayes*. arXiv 1312.6114v10.

Kingma, D.P., and Ba, J. (2014). *Adam: A method for stochastic optimization*. arXiv 1412.6980.

Klambauer, G., Unterthiner, T., Mayr, A., & Hochreiter, S. (2017). *Self-Normalizing Neural Networks*. arXiv preprint arXiv:1706.02515.

Knox, K. T. (2004). A researcher's dilemma-philosophical and methodological pluralism. *The Electronic Journal of Business Research Methods*, *2*(2), 119-128.

Koutník, J., Schmidhuber, J., & Gomez, F. (2014, July). Evolving deep unsupervised convolutional networks for vision-based reinforcement learning. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation* (pp. 541-548). ACM.

Krizhevsky, A. (2009). *Learning multiple layers of features from tiny images.* Master's thesis, University of Toronto, Canada.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In F. Pereira, C.J.C. Burges, L. Bottou & K.Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems 25 (NIPS),* 1097-1105. Red Hook, NY: Curran Associates.

Kulkarni, T. D., Whitney, W. F., Kohli, P., & Tenenbaum, J. (2015). Deep convolutional inverse graphics network. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Advances in neural information processing systems, 28* (pp. 2539–2547). Red Hook, NY: Curran.

Kushner, H. J. (1964). A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. *Journal of Basic Engineering*, *86*(1), 97-106.

Lamos-Sweeney, J. D. (2012). *Deep learning using genetic algorithms*. Master's thesis, Rochester Institute of Technology.

Laptev, D., Savinov, N., Buhmann, J. M., and Pollefeys, M. (2016). *TI-POOLING: Transformation-invariant pooling for feature learning in convolutional neural networks*. arXiv 1604.06318.

Lavin, A., & Gray, S. (2016). Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 4013–4021). Red Hook, NY: Curran.

LeCun, Y. (1989). Generalization and network design strategies. In R. Pfeifer, Z. Schreter, F. Fogelman, & L. Steels (Eds.), *Connections in perspective* (pp. 143–155). Zurich, Switzerland: Elsevier.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989a). Handwritten digit recognition with a back-propagation network. In D. S. Touretzky (Ed.), *Advances in neural information processing systems, 2* (pp. 396–404). Cambridge, MA: MIT Press.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989b). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, *1*(4), 541–551.

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

LeCun, Y., Bottou, L., Orr, G. B., and Müller, K. R. (1998). Efficient backprop. In *Neural networks: Tricks of the trade* (pp. 9-50). Springer, Berlin, Heidelberg.

LeCun, Y., Huang, F. J., & Bottou, L. (2004). Learning methods for generic object recognition with invariance to pose and lighting. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 97–104). Red Hook, NY: Curran.

LeCun, Y., Kavukcuoglu, K., & Farabet, C. (2010). Convolutional networks and applications in vision. In *Proceedings of the IEEE International Symposium on Circuits and Systems* (pp. 253–256). Red Hook, NY: Curran.

Lee, H., Grosse, R., Ranganath, R., & Ng, A. Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the*

26th International Conference Machine Learning (pp. 609–616).N.p.: International Machine Learning Society.

Lessmann, S., Stahlbock, R., & Crone, S. F. (2005). Optimizing hyperparameters of support vector machines by genetic algorithms. In *IC-AI* (pp. 74-82).

Li, K., & Malik, J. (2016). *Learning to optimize*. arXiv preprint arXiv:1606.01885.

Lin, M., Chen, Q., & Yan, S. (2013). *Network in network*. ArXiv Preprint arXiv:1312.4400.

Linnainmaa, S. (1970). *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors*. Master's thesis, University of Helsinki, Finland.

Liu, W., Wen, Y., Yu, Z., & Yang, M. (2016, June). Large-Margin Softmax Loss for Convolutional Neural Networks. In *ICML* (pp. 507-516).

Lorenzo, P. R., Nalepa, J., Kawulok, M., Ramos, L. S., & Pastor, J. R. (2017, July). Particle swarm optimization for hyper-parameter selection in deep neural networks. In *Proceedings of the Genetic and Evolutionary Computation Conference* (pp. 481-488). ACM.

Loshchilov, I., & Hutter, F. (2016). *CMA-ES for Hyperparameter Optimization of Deep Neural Networks.* arXiv preprint arXiv:1604.07269.

Lovbjerg, M. (2002). Improving particle swarm optimization by hybridization of stochastic search heuristics and self-organized criticality. *Master's thesis, Department of Computer Science, University of Aarhus*.

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proceedings of the 30th International Conference Machine Learning* (pp. 1–8). N.p.: International Machine Learning Society.

Mahendran, N., Wang, Z., Hamze, F., & de Freitas, N. (2012, March). Adaptive MCMC with Bayesian optimization. In *Artificial Intelligence and Statistics* (pp. 751-760).

Mallat, S. (2012). Group invariant scattering. *Communications on Pure and Applied Mathematics, 65*(10), 1331–1398. doi:10.1002/cpa.21413

Martino, S., Ferrucci, F., Gravino, C., & Sarro, F. (2011, June). A genetic algorithm to configure support vector machines for predicting fault-prone components. In *International Conference on Product Focused Software Process Improvement* (pp. 247-261). Springer, Berlin, Heidelberg.

Masci, J., Meier, U., Ciresan, D., & Schmidhuber, J. (2011). Stacked convolutional autoencoders for hierarchical feature extraction. In *Proceedings of the 21th International Conference on Artificial Neural Networks* (pp. 52–59). Berlin: Springer.

McLeod, M., Osborne, M. A., & Roberts, S. J. (2017). *Practical Bayesian Optimization for Variable Cost Objectives*. arXiv preprint arXiv:1703.04335.

Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., ... & Hodjat, B. (2017*). Evolving Deep Neural Networks*. arXiv preprint arXiv:1703.00548.

Mishkin, D., & Matas, J. (2016). All you need is a good init. In *Proceedings of the 4th International Conference on Learning Representations* (pp. 1–13). N.p.: Computational and Biological Learning Society.

Mitchell, M. (1996). *An introduction to genetic algorithms.* Cambridge, MA: MIT Press.

Mockus, J., Tiesis, V., & Zilinskas, A. (1978). The Application of Bayesian Methods for Seeking the Extremum. In *Towards Global Optimization*, volume 2. (pp. 117-128). Amsterdam, Elsevier.

Montavon, G., Orr, G. B., & Muller, K. (Eds.). (2012). *Neural networks: Tricks of the trade* (2nd ed.). Berlin: Springer.

Muller, U., Ben, J., Cosatto, E., Flepp, B., & LeCun, Y. (2005). Off-road obstacle avoidance through end-to-end learning. In Y.Weiss, P. B. Scholkopf, & J. C. Platt (Eds.), *Advances in neural information processing systems, 18* (pp. 739–746). Cambridge, MA: MIT Press.

Nagi, J., Ducatelle, F., Di Caro, G. A., Ciresan, D., Meier, U., Giusti, A., . . . Gambardella, L. M. (2011). Max-pooling convolutional neural networks for vision based hand gesture recognition. In *Proceedings of the IEEE International Conference on Signal and Image Processing Applications* (pp. 342–347). Red Hook, NY: Curran.

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning* (pp. 807–814). N.p.: International Machine Learning Society.

Nasse, F., Thurau, C., & Fink, G. A. (2009). Face detection using GPU-based convolutional neural networks. In *Proceedings of the 13th International Conference on Computer Analysis of Images and Patterns* (pp. 83–90). Berlin: Springer.

Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence O (1/k^ 2). In *Doklady AN USSR* (Vol. 269, pp. 543-547).

Nguyen, A., Yosinski, J., & Clune, J. (2015). Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 427–436). Los Alamitos, CA: IEEE Computer Society.

Nowlan, S. J., & Hinton, G. E. (1992). Simplifying neural networks by soft weight sharing. *Neural Computation*, *4*(4), 473–493.

Oh, K., & Jung, K. (2004). GPU implementation of neural networks. *Pattern Recognition*, *37*(6), 1311–1314.

Orive, D., Sorrosal, G., Borges, C.E., Martin, C., Alonso-Vicario, A. (2014). Evolutionary algorithms for hyperparameter tuning on neural networks models. In Affenzeller, Bruzzone, Jiménez, Longo, Merkuryev, Zhang (Eds.), *Proceedings of the European Modeling and Simulation Symposium,* 402-409.

Orlikowski, W. J., & Baroudi, J. J. (1991). Studying information technology in organizations: Research approaches and assumptions. *Information systems research*, *2*(1), 1-28.

Oullette, R., Browne, M., & Hirasawa, K. (2004, June). Genetic algorithm optimization of a convolutional neural network for autonomous crack detection. In *Evolutionary Computation, 2004. CEC2004. Congress on* (Vol. 1, pp. 516-521). IEEE.

Paine, T., Jin, H., Yang, J., Lin, Z., & Huang, T. (2013). *GPU asynchronous stochastic gradient descent to speed up neural network training*. arXiv: 1312.6186.

Papernot, N., McDaniel, P., Wu, X., Jha, S., & Swami, A. (2016). Distillation as a defense to adversarial perturbations against deep neural networks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy* (pp. 1–16). Los Alamitos, CA: IEEE Computer Society.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*(Oct), 2825-2830.

Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural Networks*, *12*, 145–150.

Qiao, Y., Shen, J., Xiao, T., Yang, Q., Wen, M., & Zhang, C. (2016). FPGA-accelerated deep convolutional neural networks for high throughput and energy efficiency. *Concurrency and Computation: Practice and Experience*. doi:10.1002/cpe.3850

Quinonero-Candela, J., Rasmussen, C. E., & Figueiras-Vidal, A. R. (2010). Sparse spectrum Gaussian process regression. *Journal of Machine Learning Research*, *11*(Jun), 1865-1881.

Ranzato, M. A., Huang, F. J., Boureau, Y., & LeCun, Y. (2007). Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition (CVPR),* 1-8. Los Alamitos, CA: IEEE Computer Society.

Ranzato, M., Poultney, C., Chopra, S., & LeCun, Y. (2006). Efficient learning of sparse representations with an energy-based model. In P. B. Scholkopf, J. C. Platt, & T. Hoffman (Eds.), *Advances in neural information processing systems, 19* (pp. 1137–1144). Cambridge, MA: MIT Press.

Rasmussen, C. E., & Williams, C. K. (2006). *Gaussian processes for machine learning*, volume 1. Cambridge: MIT press.

Rawat, W., & Wang, Z. (2017). Deep convolutional neural networks for image classification: A comprehensive review. *Neural Computation, 29*(9), 2352-2449. Doi: 10.1162/neco_a_00990

Razavian, A., Azizpour, H., Sullivan, J., & Carlsson, S. (2014). CNN features off-the shelf: An astounding baseline for recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops* (pp. 806–813). Los Alamitos, CA: IEEE Computer Society.

Recht, B., Re, C., Wright, S., & Niu, F. (2011). Hogwild: Alock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, & K. Q.Weinberger (Eds.), *Advances in neural information processing systems*, *24* (pp. 693–701). Red Hook, NY: Curran.

Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems* (pp. 91-99).

Romero, A., Ballas, N., Kahou, S. E., Chassang, A., Gatta, C., & Bengio, Y. (2015). Fitnets: Hints for thin deep nets. In *Proceedings of the 3rd International Conference on Learning Representations* (pp. 1–13). N.p.: Computational and Biological Learning Society.

Ruder, S. (2017). *An overview of gradient descent optimization algorithms*. arXiv preprint arXiv:1609.04747v2.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., . . . Bernstein, M. (2015). ImageNet large scale visual recognition challenge. *International Journal of Computer Vision, 115*(3), 211-252.

Saxe, A. M., McClelland, J. L., and Ganguli, S. (2013). *Exact solutions to the nonlinear dynamics of learning in deep linear neural networks*. arXiv 1312.6120.

Scherer, D., Müller, A., & Behnke, S. (2010). Evaluation of pooling operations in convolutional architectures for object recognition. In *Proceedings of the 20th International Conference on Artificial Neural Networks (ICANN),* 92-101. Heidelberg, Berlin, Germany: Springer.

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

Seeger, M., Williams, C., & Lawrence, N. (2003). Fast forward selection to speed up sparse Gaussian process regression. In *Artificial Intelligence and Statistics 9*. EPFL-CONF-161318.

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., & de Freitas, N. (2016). Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, *104*(1), 148-175.

Simard, P. Y., Steinkraus, D., & Platt, J. C. (2003). Best practices for convolutional neural networks applied to visual document analysis. In *Proceedings of the 7th International Conference on Document Analysis and Recognition* (vol. 3, pp. 958–963). Washington, DC: IEEE Computer Society.

Simonyan, K., & Zisserman, A. (2014). *Very deep convolutional networks for large-scale image recognition.* ArXiv Preprint arXiv: 1409.1556.

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems, 25* (pp. 2951–2959). Red Hook, NY: Curran.

Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., ... & Adams, R. (2015). Scalable bayesian optimization using deep neural networks. In *International Conference on Machine Learning* (pp. 2171-2180).

Srinivas, N., Seeger, M., Kakade, S. M., & Krause, A. (2009). *Gaussian process bandits without regret: An experimental design approach.* arXiv preprint arXiv: 0912.3995.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research, 15*(1), 1929-1958.

Srivastava, R. K., Greff, K., & Schmidhuber, J. (2015a). Training very deep networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Advances in neural information processing systems, 28* (pp. 2377–2385). Red Hook, NY: Curran.

Srivastava, R. K., Greff, K., & Schmidhuber, J. (2015b). *Highway networks*. arXiv 1505.00387.

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, *10*(2), 99-127

Stanley, K. O., D'Ambrosio, D. B., & Gauci, J. (2009). A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, *15*(2), 185-212.

Steinkrau, D., Simard, P. Y., & Buck, I. (2005). Using GPUs for machine learning algorithms. In *Proceedings of the 8th International Conference on Document Analysis and Recognition* (pp. 1115–1119).Washington, DC: IEEE Computer Society.

Sussillo, D., and Abbott, L. (2014). *Random walk initialization for training very deep feedforward networks*. arXiv 1412.6558.

Sutskever, I., Martens, J., Dahl, G. E., & Hinton, G. E. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference Machine Learning* (pp. 1139–1147). N.p.: International Machine Learning Society.

Swersky, K., Snoek, J., & Adams, R. P. (2013). Multi-task bayesian optimization. In *Advances in neural information processing systems* (pp. 2004-2012)

Szegedy, C., Ioffe, S., & Vanhoucke, V. (2016). *Inception-v4, Inception-ResNet and the impact of residual connections on learning*. arXiv 1602.07261.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., . . . Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition (CVPR),* 1-9. Los Alamitos, CA: IEEE Computer Society.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2015). *Rethinking the Inception architecture for computer vision*. ArXiv Preprint arXiv: 1512.00567.

Szegedy, C., Zaremba,W., Sutskever, I., Bruna, J., Erhan, D.,Goodfellow, I., and Fergus,R. (2014). Intriguing properties of neural networks. In *Proceedings of the 1st International Conference on Learning Representations* (pp. 1–10). N.p.: Computational and Biological Learning Society.

Tang, Y. (2013). *Deep learning using linear support vector machines*. arXiv 1306.0239.

Thornton, C., Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2013, August). Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 847-855). ACM.

Tirumala, S. S. (2014, November). Implementation of Evolutionary Algorithms for Deep Architectures. In *AIC* (pp. 164-171).

Turaga, S. C., Murray, J. F., Jain, V., Roth, F., Helmstaedter, M., Briggman, K., . . .Seung, H. S. (2010). Convolutional networks can learn to generate affinity graphs for image segmentation. *Neural Computation*, *22*(2), 511–538.

Uličný, M., Lundstrom, J., & Byttner, S. (2016). Robustness of deep convolutional neural networks for image recognition. In *Proceedings of the 1st International Symposium on Intelligent Computing Systems* (pp. 16–30). Switzerland: Springer International Publishing.

Vinyals, O., Toshev, A., Bengio, S., & Erhan, D. (2015). Show and tell: A neural image caption generator. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 3156–3164). Red Hook, NY: Curran.

Wager, S., Wang, S., & Liang, P. S. (2013). Dropout training as adaptive regularization. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems, 26* (pp. 351–359). Red Hook, NY: Curran.

Wan, L., Zeiler, M., Zhang, S., LeCun, Y., & Fergus, R. (2013). Regularization of neural networks using Dropconnect. In *Proceedings of the 30th International Conference Machine Learning* (pp. 1058–1066). N.p.: International Machine Learning Society.

Warde-Farley, D., Goodfellow, I. J., Courville, A., & Bengio, Y. (2013). *An empirical analysis of dropout in piecewise linear networks*. arXiv 1312.6197.

Werbos, P. (1974). *Beyond regression: New tools for prediction and analysis in the behavioral sciences*. Doctoral dissertation, Harvard University.

Werbos, P. J. (1982). Applications of advances in nonlinear sensitivity analysis. In R. F. Drenick & F. Kozin (Eds.), *System modeling and optimization* (pp. 762–770). Berlin: Springer.

Weston, J., Ratle, F., Mobahi, H., & Collobert, R. (2008). Deep learning via semi-supervised embedding. In *Proceedings of the 25th International Conference on Machine Learning* (pp. 1168–1175). N.p.: International Machine Learning Society.

Wiatowski, T., & Bolcskei, H. (2015). *A mathematical theory of deep convolutional neural networks for feature extraction*. arXiv 1512.06293.

Xavier-de-Souza, S., Suykens, J. A., Vandewalle, J., & Bollé, D. (2010). Coupled simulated annealing. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, *40*(2), 320-335.

Xie, L., & Yuille, A. (2017*). Genetic CNN*. arXiv preprint arXiv:1703.01513.

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, *87*(9), 1423-1447.

Ye, F. (2017). Particle swarm optimization-based automatic parameter selection for deep neural networks and its applications in large-scale and high-dimensional data. *PloS one*, *12*(12), e0188746.

Yu, D., Wang, H., Chen, P., & Wei, Z. (2014). Mixed pooling for convolutional neural networks. In *Proceedings of the 9th International Conference on Rough Sets and Knowledge Technology* (pp. 364–375). Berlin: Springer.

Yu, W., Yang, K., Bai, Y., Yao, H., and Rui, Y. (2014a). DNN flow: DNN feature pyramid based image matching. In *Proceedings of the British Machine Vision Conference* (pp. 1–10). Durham, UK: BMVA Press.

Yu, X., & Gen, M. (2010). *Introduction to evolutionary algorithms*. Springer Science & Business Media.

Yu, W., Yang, K., Bai, Y., Yao, H., & Rui, Y. (2014b). *Visualizing and comparing convolutional neural networks*. arXiv 1412.6631.

Zeiler, M. D. (2012). *ADADELTA: an adaptive learning rate method*. arXiv preprint arXiv:1212.5701.

Zeiler, M. D., & Fergus, R. (2014). Visualizing and understanding convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV),* 818-833. Heidelberg, Berlin, Germany: Springer.

Zeiler, M. D., Taylor, G.W., and Fergus, R. (2011). Adaptive deconvolutional networks for mid and high level feature learning. In *Proceedings of the IEEE International Conference on Computer Vision* (pp. 2018–2025). Red Hook, NY: Curran.

Zhao, Q., and Griffin, L. D. (2016). *Suppressing the unusual: Towards robust CNNs using symmetric activation functions*. arXiv 1603.05145v1.

Zhuang, Y., Chin, W., Juan, Y., & Lin, C. (2013). A fast parallel SGD for matrix factorization in shared memory systems. *Proceedings of the 7th ACM Conference on Recommender Systems* (pp. 249–256). New York: ACM.

Zinkevich, M., Weimer, M., Li, L., and Smola, A. J. (2010). Parallelized stochastic gradient descent. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, & A. Culotta (Eds.), *Advances in neural information processing systems (NIPS), 23* (pp. 2595–2603). Red Hook, NY: Curran.

Zoph, B., & Le, Q. V. (2016). *Neural architecture search with reinforcement learning*. arXiv preprint arXiv:1611.01578.

Appendix A: Sample python source codes

A1: Listing one

```
"""
File: GA_Optimzation.py

Author: W. Rawat

Description: Stochastic Genetic algorithm code for DCNN architectural selection

Detailed description: This section of code defines the core of the Genetic Algorithm used for
DCNN model selection. It defines all the genetic operations including selection, mutation and
crossover.

Credit: Based on the Multi-Layer Perceptron (MLP) evolution code from:
https://github.com/harvitronix/neural-network-genetic-algorithm
"""
# Initial library and module imports
import numpy as np

# Set python random seed to improve reproducibility
seed = 1337
np.random.seed(seed)

# Script specific python imports
from functools import reduce
from operator import add
import random
from network import Network

# GA Assignments
# Set the percentage of DCNN's to retain for each new generation (float)
retention_rate = retention_rate
# Set the probability of a discarded DCNN forming part of the new generation (float)
stochastic_keep = stochastic_keep
# Set the probability that a DCNN will be randomly mutated (float)
stochastic_mutate = stochastic_mutate

class GA_Optimzation():

    """This is the class that implements the stochastic GA for DCNN architectural
    optimization"""

  def __init__(self, dcnn_architectural_space, retention_rate=retention_rate,
          stochastic_keep=stochastic_keep, stochastic_mutate=stochastic_mutate):

    self.stochastic_mutate = stochastic_mutate
    self.stochastic_keep = stochastic_keep
    self.retention_rate = retention_rate
```

```
        self.dcnn_architectural_space = dcnn_architectural_space

    def create_population(self, count):

        """Create a randomly selected population of DCNN's
        Passes:
            count: Defines the size of the population, i.e. the number of networks to create (int)
        Returns:
            The population of DCNN network objects (list)
        """
        pop = []
        for _ in range(0, count):

            # Generate a random DCNN.
            network = Network(self.dcnn_architectural_space)  # Network is a class that
                                                              # represents an individual DCNN
            network.create_random()
            # Add the network to the population of DCNN's
            pop.append(network)

        return pop

    @staticmethod

    def fitness(network):

        """Return the classification accuracy of each DCNN model"""

        return network.accuracy


    def average_grade(self, pop):

        """Compute the mean accuracy of the generation of DCNN's
        Passes:
            pop: The population of DCNN's (list)
        Returns:
            The mean accuracy of the population of DCNN's (float)
        """

        summed = reduce(add, (self.fitness(network) for network in pop))

        return summed / float((len(pop)))

    def reproduce(self, parent1, parent2):

        """From the parents, make two offspring.
        Passes:
            parent1: Network parameters (dict)
            parent2: Network parameters (dict)
        Returns:
            Two DCNN network objects (list)
```

```
        """
        offsprings = []
        for _ in range(2):
            offspring = {}

            # Scan through the DCNN architectural space and select the architecture of the
            # offspring
            for param in self.dcnn_architectural_space:
                offspring[param] = random.choice(
                    [parent1.network[param], parent2.network[param]]
                )

            # Generate a DCNN network object
            network = Network(self.dcnn_architectural_space)
            network.create_set(offspring)
            # Perform stochastic mutation on a selection of the offspring
            if self.stochastic_mutate > random.random():
                network = self.mutate(network)
            # Add the mutated offspring to the population of DCNN's
            offsprings.append(network)

        return offsprings

    def mutate(self, network):

        """Stochastically mutate a part of the architecture
        Passes:
            network: The architectural parameters to mutate (dict)
        Returns:
            (Network): A stochastically mutated architectural object
        """

        # Random key selection
        mutation = random.choice(list(self.dcnn_architectural_space.keys()))
        # Stochastic parameter mutation
        network.network[mutation] = random.choice(self.dcnn_architectural_space[mutation])

        return network

    def evolution(self, pop):

        """Carry out the evolution of the population of DCNN's
        Passes:
            pop: A list of network parameters (list)
        Returns:
            The evolved population of DCNN's (list)
        """

        # Get classification accuracy for each DCNN
        average_graded = [(self.fitness(network), network) for network in pop]
```

```
# Filter the DCNN's in decending order - Arrange from highest to lowest
average_graded = [x[1] for x in sorted(average_graded, key=lambda x: x[0],
reverse=True)]

# Determine the predetermined number to be part of the next generation
retention_number = int(len(average_graded)*self.retention_rate)

# The DCNN's retained, represent the parents for the new generation
parents = average_graded[:retention_number]

# Stochastically keep DCNN's from those that are discarded
for individual in average_graded[retain_number:]:
    if self.stochastic_keep > random.random():
        parents.append(individual)

# Determine the requirement to maintain the population at the original population size
parents_number = len(parents)
required_number = len(pop) - parents_number
offsprings = []

# From the two remaining DCNN's produce offspring
while len(offsprings) < required_number:

    # Stochasitically select the parents
    mother = random.randint(0, parents_number-1)
    father = random.randint(0, parents_number-1)

    # If they are not distinct, correct
    if mother  == father:
        mother = parents[mother]
        father = parents[father]

        # Carryout recombination operation
        new_borns = self. reproduce(mother, father)

        # Add the offsprings one at a time.
        for new_born in new_borns:

            # Add newborns to population and prevent growing the population to larger than
            # the required size
            if len(offsprings) < required_number:
                offsprings.append(new_born)

parents.extend(offsprings)

return parents
```

A2: Listing two

```
"""
File: Genetic_CNN_training.py

Author: W. Rawat

Description: DCNN data and architecture configuration and training

Detailed description: This section of code loads and configures the MNIST dataset, defines
the DCNN architecture, conducts the training, and displays the model diagram and accuracy
performance

"""

# Import the print function
from __future__ import print_function

# Initial library and module imports
import numpy as np
import tensorflow as tf
import random

# Import os and configure python seed to stabilize hash-based operations
import os
os.environ['PYTHONHASHSEED'] = '0'

# Set Numpy, python and TensorFlow random seeds to improve reproducibility
np.random.seed(42)
random.seed(12345)
tf.set_random_seed(1234)

# Import the various Keras related modules
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, AlphaDropout
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
from keras import optimizers, initializers

# Assignments specific to the MNIST dataset

# Assign the number of classes
num_classes = 10

# Assign the input image dimensions
img_rows, img_cols = 28, 28

# Load the preshuffled MNIST dataset as its constituent train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
# Rearrange the data for use in Keras
if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

# Convert the data to float and normalize to the range [0-1]
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# Print the shape to make sure everything is in order before passing to the first fully connected
# layer
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Process labels into distinct class types
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# Pass the network parameters from the network class
def compile_model(network):                                    # Comment this line and the return
                                                               # line, and manually set
                                                               # assignments below to run any
                                                               # arbitrary individual architecture
    """Build and compile the model
     Passes:
        network: The architecture of the network (dict)

     Returns:
         An individually compiled architecture
    """
    # Obtain the network parameters

    # Activation options passed to the network class - [RELU; ELU; SELU
    activation = activation

    # Filter no. options for the first convolutional layer - [16; 32; 64]
    nb_of_filters_c1 = network['nb_of_filters_c1']

    # Filter sizes for the first convolutional layer - [3; 4; 5]
    filter_sizes_c1 = network['filter_sizes_c1']

    # Filter no. options for the second convolutional layer - [16; 32; 64]
    nb_of_filters_c2 = network['nb_of_filters_c2']
```

```
# Filter sizes for the second convolutional layer - [3; 4; 5]
filter_sizes_c2 = network['filter_sizes_c2']

# Filter number options for the first and second fully connected layers   # - [64; 128; 256]
nb_of_filters_fc1 = network['nb_of_filters_fc1']

# Dropout probability for the first Dropout layer – [0; 0.25; 0.5; 0.75
# / 0; 0.025; 0.05; 0.01]
dropout_fc1 = network['dropout_fc1']

# Dropout probability for the second Dropout layer – [0; 0.25; 0.5; 0.75
# / 0; 0.025; 0.05; 0.01]
dropout_fc2 = network['dropout_fc2']

# Define a Keras sequential model that specifies the model architecture
model = Sequential()

    # For the first convolutional layer: Initialize the convolutional kernel    # using Le_Cun
    initialization; pass the assigned value to the activation    # function, number of filters and
    kernel size
    model.add(Conv2D(network['nb_of_filters_c1'],
                     kernel_initializer='lecun_normal',
                     kernel_size=(filter_sizes_c1, filter_sizes_c1),
                     activation=activation,
                     input_shape=input_shape))

    # Carry out max-pooling with fixed filter size
    model.add(MaxPooling2D(pool_size=(2, 2)))

    # For the second convolutional layer: Initialize the convolutional kernel using Le_Cun
    # initialization; pass the assigned value to the   # activation function, number of filters
    # and kernel size
    model.add(Conv2D(network['nb_of_filters_c2'],
                     kernel_initializer='lecun_normal',
                     kernel_size=(filter_sizes_c2, filter_sizes_c2), activation=activation))


    # Carry out max-pooling with fixed filter size
    model.add(MaxPooling2D(pool_size=(2, 2)))

    # Make the convoltuional layers 1-Dimensional before passing them to the fully
    # connected layers
    model.add(Flatten())

    # For the first fully connected layer: Pass the assigned filter number and activation
    # function;  set the initialization scheme to Le_Cun
    model.add(Dense(network['nb_of_filters_fc1'],
                     kernel_initializer='lecun_normal',
                     activation=activation))

    # Apply a Dropout layer and pass the assigned Dropout probability
    model.add(Dropout(network['dropout_fc1']))
```

```
        # For the second fully connected layer: Pass the assigned filter number and activation
        # function; set the initialization scheme to Le_Cun
        model.add(Dense(network['nb_of_filters_fc2'],
                        kernel_initializer='lecun_normal',
                        activation=activation))

        # Apply another Dropout layer and pass the assigned Dropout probability
        model.add(Dropout(network['dropout_fc2']))

        # For the last fully connected layer: Pass the number of classes and set
        # the activation function as the Softmax activation
        model.add(Dense(nb_classes, activation='softmax'))


        # Select SGD as the optimizer and set the fixed learning rate
        sgd = optimizers.SGD(lr=0.01)

       # Compile the model and select the crossentropy loss function, optimized
       # using SGD. Set the objective function as accuracy
       model.compile(loss='categorical_crossentropy', optimizer='sgd',
                metrics=['accuracy'])

       return model

# Fit the model on the training data: Set the fixed number of epochs, the  # fixed batch size
and the ask for the data to be shuffled
model.fit(x_train, y_train,
          batch_size=64,
          epochs=20,
          verbose=2,
          shuffle=True)

# Validate the model on the test data
score = model.evaluate(x_test, y_test, verbose=0)

# Display the classification accuracy and the loss
print('Test loss:', score[0])
print('Test accuracy:', score[1])

# Display the model summary to ascertain the number of parameters per layer
model.summary()

# Plot the model using Keras visualization
# Import associated libraries and modules
import pydot, graphviz
from keras.utils import plot_model

# Plot the model found in Appendix B and save to file
plot_model(model, to_file='DCNN_model.png')
```

A3: Listing three

```
"""
File: CNN_BA_1.X.py

Author: W. Rawat

Description: Learning parameter optimization for first Bayesian search

Detailed description: This code creates the connection with the SigOpt SaaS, creates the
experimental variables for the first Bayesian search, and defines the learning parameters and
their values. Its passes these values to the GA inspired architecture for training.
"""
# Section 1: Bayesian experiment configuration

# API related python imports

import argparse
from sigopt.sigopt import Connection

# Establish connection with SigOpt

conn =
Connection(client_token="KLXCNIJFFYLRRHTCJPMPNSOHHONUTDVMFBXIKQJQV
NPAEDIG")

# Setup the Bayesian optimization experiment

experiment = conn.experiments().create(

 name=" Bayesian_CNN_1",

 parameters=[

  dict(

   name="initializer",                          # Define the initialization options

   categorical_values=[

    dict(

     name="RandomNormal",

     enum_index=1

     ),

    dict(

     name="lecun_normal",

     enum_index=2

     ),

    dict(

     name="glorot_normal",
```

```
        enum_index=3
        ),
      dict(
        name="he_normal",
        enum_index=4
        )
      ],
    type="categorical"                    # Set initialization options as categorical type
    ),
  dict(
    name="learning_rate_encoded",         # Define the discretely encoded learning rate
                                          # 1 = 0.001; 10 = 0.01.
    bounds=dict(
      min=1,
      max=10
      ),
    default_value=1,
    type="int"
    ),
  dict(
    name="optimizer",                     # Define the optimization options
    categorical_values=[
      dict(
        name="adam",
        enum_index=1
        ),
      dict(
        name="rmsprop",
        enum_index=2
        ),
      dict(
        name="gradient_descent",
        enum_index=3
        )
```

```
        ],
    type="categorical"                  # Set optimization options as categorical type
    )
  ],
metrics=[                               # Define the objective as classification accuracy
        name="Accuracy"
  dict()
  ],
num_solutions=1,                        # API limited number of concurrent
                                        # Bayesian recommendations
observation_budget=70,                  # Specify the predetermined number of Bayesian
                                        # evaluations to 70 evaluations
parallel_bandwidth=1,
type="offline"
)
```

 # Section 2: Keras model definition and actual DCNN training

```
# Import the print function
from __future__ import print_function

# Initial library and module imports
import numpy as np
import tensorflow as tf
import random

# Import os and configure python seed to stabilize hash-based operations
import os
os.environ['PYTHONHASHSEED'] = '0'

# Set Numpy, python and TensorFlow random seeds to improve reproducibility
np.random.seed(42)
random.seed(12345)
tf.set_random_seed(1234)

# Import the various Keras related modules
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, AlphaDropout
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
from keras import optimizers, initializers
```

```
# Assignments specific to the MNIST dataset

# Assign the number of classes
num_classes = 10

# Assign the input image dimensions
img_rows, img_cols = 28, 28

# Load the preshuffled MNIST dataset as its constituent train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Rearrange the data for use in Keras
if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)


# Convert the data to float and normalize to the range [0-1]
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# Print the shape to make sure everything is in order before passing to the first layer
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Process labels into distinct class types
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# Configure the optimizers
OPTIMIZERS = {
    'adam': optimizers.Adam,
    'rmsprop': optimizers.RMSprop,
    'gradient_descent': optimizers.SGD,
}

# Pass the network parameters from the Bayesian assignments

"""Build, compile and fit the model
    Passes:
        assignments: The learning parameter assignment (dict)
    Returns:
        An individually compiled set of parameters
    """
```

```
def create_model(assignments):

    # Define a Keras sequential model that specifies the model architecture

    model = Sequential()

    # For the first convolutional layer: Pass the initializer and set GA inspired architecture
    model.add(Conv2D(64, kernel_initializer=assignments['initializer'],
                     kernel_size=(5, 5), activation='ReLU', input_shape=input_shape))

    # Carry out max-pooling with fixed filter size
    model.add(MaxPooling2D(pool_size=(2, 2)))


    # For the second convolutional layer: Pass the initializer and set GA inspired architecture
    model.add(Conv2D(64, kernel_initializer=assignments['initializer'],
                     kernel_size=(5, 5), activation='ReLU'))

    # Carry out max-pooling with fixed filter size
    model.add(MaxPooling2D(pool_size=(2, 2)))

    # Make the convoltuional layers 1-Dimensional
    model.add(Flatten())

    # For the first fully connected layer: Pass the initializer and set GA inspired architecture
    model.add(Dense(256, kernel_initializer=assignments['initializer'],
                     activation='ReLU'))

    # Apply GA inspired Dropout rate
    model.add(Dropout(0.25))

    # For the second fully connected layer: Pass the initializer and set GA inspired
    architecture
    model.add(Dense(256, kernel_initializer=assignments['initializer'],
                     activation='ReLU'))

    # Apply GA inspired Dropout rate
    model.add(Dropout(0.25))

    # For the last fully connected layer: Pass the number of classes and set the softmax
    # activation

    model.add(Dense(num_classes, activation='softmax'))

    # Pass the optimizer and compile the model; select the crossentropy loss. Set the
    # objective function as accuracy.
    model.compile(
        optimizer=OPTIMIZERS[assignments['optimizer']](lr=(assignments['learning_rate_e
        ncoded'])),
        loss='categorical_crossentropy',
        metrics=['accuracy'],
    )
```

```
        # Fit the model on the shuffled training data and set the batch size and number of epochs
        model.fit(x_train, y_train,
                  batch_size=64,
                  epochs=20,
                  verbose=2,
                  shuffle=True)

        return model

# Validate the model on the test data
def evaluate_model(assignments):
    model = create_model(assignments)
    return model.evaluate(x_test, y_test, verbose=0)[1]

# Section 3: Automate the process for the predetermined number of evaluations

for _ in range(experiment.observation_budget):
    suggestion = conn.experiments(experiment.id).suggestions().create()
    assignments = suggestion.assignments
    value = evaluate_model(assignments)

    conn.experiments(experiment.id).observations().create(
        suggestion=suggestion.id,
        value=value
    )

assignments = conn.experiments(experiment.id).best_assignments().fetch().data[0].
assignments

# Display the parameters
print(assignments)
```

A4: Listing four

```
"""
File: CNN_BA_2.X.py

Author: W. Rawat

Description: Learning parameter optimization for second Bayesian search

Detailed description: This code creates the connection with the SigOpt SaaS, creates the
experimental variables for the first Bayesian search, and defines the learning parameters and
their values. Its passes these values to the GA inspired architecture for training.
"""
# Section 1: Bayesian experiment configuration
# API related python imports
import argparse
from sigopt.sigopt import Connection

# Establish connection with SigOpt
conn =
Connection(client_token="KLXCNIJFFYLRRHTCJPMPNSOHHONUTDVMFBXIKQJQV
NPAEDIG")
# Setup the Bayesian optimization experiment
experiment = conn.experiments().create(
 name="Bayesian_CNN_2",
 parameters=[
  dict(
   name="Batch size",                    # Define the batch size variable
   bounds=dict(
    min=32,
    max=128
    ),
   default_value=64,
   type="int"                            # Set as integer type
   ),
  dict(
   name="Initialization scheme",         # Define the initialization options
   categorical_values=[
    dict(
```

```
        name="Lecun_normal",
        enum_index=5
        ),
      dict(
        name="glorot_normal",
        enum_index=6
        ),
      dict(
        name="he_normal",
        enum_index=7
        ),
      dict(
        name="random_normal",
        enum_index=8
        )
      ],
    default_value="random_normal",
    type="categorical"                    # Set initialization options as categorical type

    ),
  dict(
    name="Learning rate",                 # Define the continuous learning rate variable
    bounds=dict(
      min=-6.907755279,                   # -6.907755279 = 0.001
      max=-4.605170186                    # -4.605170186 = 0.01
      ),
    type="double"                         # Set encoded learning rate as double
    ),
  dict(
    name="Optimizer",                     # Define the optimization options and include
    categorical_values=[                  # Adagrad; AdaMax and Nadam options
      dict(
        name="SGD",
        enum_index=3
```

```
        ),
        dict(
          name="Rmsprop",
          enum_index=5
        ),
        dict(
          name="Adam",
          enum_index=6
        ),
        dict(
          name="Adamax",
          enum_index=7
        ),
        dict(
          name="Nadam",
          enum_index=8
        ),
        dict(
          name="Adagrad",
          enum_index=10
        )
      ],
      type="categorical"                    # Set optimization options as categorical type
    )
  ],
  metrics=[
    dict(
      name="Accuracy"                       # Define the objective as classification accuracy
    )
  ],
  num_solutions=1,                          # API limited number of concurrent
                                            # Bayesian recommendations
  observation_budget=120,                   # Specify the predetermined number of
                                            # Bayesian evaluations to 120 evaluations
```

```
 parallel_bandwidth=1,
 type="offline"
 )
# Section 2: Keras model definition and actual DCNN training
# Import the print function
from __future__ import print_function

# Initial library and module imports
import numpy as np
import tensorflow as tf
import random

# Import os and configure python seed to stabilize hash-based operations
import os
os.environ['PYTHONHASHSEED'] = '0'

# Set Numpy, python and TensorFlow random seeds to improve reproducibility
np.random.seed(42)
random.seed(12345)
tf.set_random_seed(1234)

# Import the various Keras related modules
import math
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, AlphaDropout
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
from keras import optimizers, initializers

# Assignments specific to the MNIST dataset

# Assign the number of classes
num_classes = 10

# Assign the input image dimensions
img_rows, img_cols = 28, 28

# Load the preshuffled MNIST dataset as its constituent train and test sets

(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Rearrange the data for use in Keras

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
```

```
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

# Convert the data to float and normalize to the range [0-1]
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# Print the shape to make sure everything is in order before passing to the first layer
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Process labels into distinct class types
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# Configure the optimizers
OPTIMIZERS = {
    'adam': optimizers.Adam,
    'rmsprop': optimizers.RMSprop,
    'gradient_descent': optimizers.SGD,
    'nadam': optimizers.Nadam,
    'adamax': optimizers.Adamax,
    'adagrad': optimizers.Adagrad,

}

# Pass the network parameters from the Bayesian assignments

"""Build, compile and fit the model
    Passes:
        assignments: The learning parameter assignment (dict)
    Returns:
        An individually compiled set of parameters
    """

def create_model(assignments):

        # Define a Keras sequential model that specifies the model architecture
        model = Sequential()

        # For the first convolutional layer: Pass the initializer and set GA inspired architecture
        model.add(Conv2D(64, kernel_initializer=assignments['initializer'],
                         kernel_size=(5, 5), activation='ReLU', input_shape=input_shape))

        # Carry out max-pooling with fixed filter size
        model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
    # For the second convolutional layer: Pass the initializer and set GA inspired architecture
    model.add(Conv2D(64, kernel_initializer=assignments['initializer'],
                     kernel_size=(5, 5), activation='ReLU'))

    # Carry out max-pooling with fixed filter size
    model.add(MaxPooling2D(pool_size=(2, 2)))

    # Make the convoltuional layers 1-Dimensional
    model.add(Flatten())

    # For the first fully connected layer: Pass the initializer and set GA inspired architecture
    model.add(Dense(256, kernel_initializer=assignments['initializer'],
                     activation='ReLU'))

    # Apply GA inspired Dropout rate
    model.add(Dropout(0.25))

    # For the second fully connected layer: Pass the initializer and set GA inspired
    architecture
    model.add(Dense(256, kernel_initializer=assignments['initializer'],
                     activation='ReLU'))

    # Apply GA inspired Dropout rate
    model.add(Dropout(0.25))

    # For the last fully connected layer: Pass the number of classes and set the softmax
    # activation
    model.add(Dense(num_classes, activation='softmax'))

    # Pass the optimizer and log encoded learning rate, and compile the model; select the
    # crossentropy loss. Set the objective function as accuracy.
    model.compile(
        optimizer=OPTIMIZERS[assignments['optimizer']](
        lr=math.exp(assignments['Learning rate'])),
        loss='categorical_crossentropy',
        metrics=['accuracy'],
    )

    # Fit the model on the shuffled training data and set the batch size and number of epochs
    model.fit(x_train, y_train,
                batch_size= assignments['Batch size'],
                epochs=20,
                verbose=2,
                shuffle=True)

    return model

# Validate the model on the test data
def evaluate_model(assignments):
    model = create_model(assignments)
    return model.evaluate(x_test, y_test, verbose=0)[1]
```

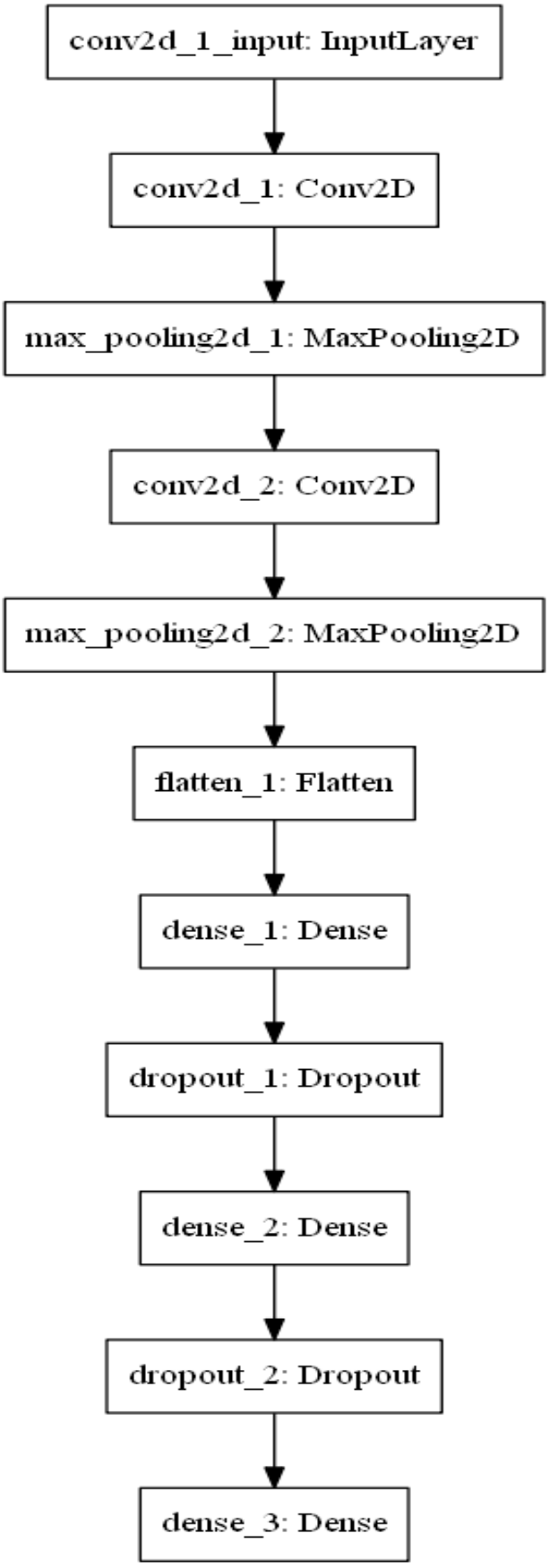
```
# Section 3: Automate the process for the predetermined number of evaluations

for _ in range(experiment.observation_budget):
    suggestion = conn.experiments(experiment.id).suggestions().create()
    assignments = suggestion.assignments
    value = evaluate_model(assignments)

    conn.experiments(experiment.id).observations().create(
        suggestion=suggestion.id,
        value=value
    )

assignments = conn.experiments(experiment.id).best_assignments().fetch().data[0].
assignments

# Display the parameters
print(assignments)
```

A5: Listing five

```
"""
File: MNSIT_Image_Plotter.py

Author: W. Rawat

Description: Plot images from the MNIST dataset

Detailed description: Training set images are plotted in a 10*10 = 100 image grid.

"""
# Import required libraries and modules
import matplotlib.pyplot as plt
from keras.datasets import mnist
from keras.utils import np_utils

# Configure the required image size
fig = plt.figure(figsize=(6, 6))                    # Value is in inches


# Load the MNIST dataset in its constituent training and test sets
 (x_train, y_train), (x_test, y_test) = mnist.load_data()

# Configure the image subplots
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)

# For each of the 100 images
for i in range(100):

    # Initialize the subplots and add a 10 by 10 grid

    wr = fig.add_subplot(12, 12, i + 1, xticks=[], yticks=[])

    # Display an image at the indexed position

    wr.imshow(X_test[i], cmap=plt.cm.binary, interpolation='nearest')


# Display the images in grid format
plt.show()
```

Appendix B: DCNN model visualization

DCNN model used for all simulations – Plotted using Keras model visualization



Appendix C: Approved Ethical Clearance

UNISA university of south africa

UNISA SCHOOL OF ENGINEERING (SOE) ETHICS REVIEW COMMITTEE

Date: 26/10/2017

NHREC Registration # : (If applicable) N/A

ERC Reference # **:2017/CSET_SOE/WR/001**

Name : Waseem Rawat

Student #: 58557199

Staff # : N/A

Dear Mr Waseem Rawat

Decision: Ethics Approval from 26/10/2017 to 26/10/2020

Researcher(s): Name: Mr Waseem Rawat
Address: 93 sahara Sands, 25 Playfair Road, North beach, Durban
E-mail address: wrawat@toyota.co.za, telephone # 083 646 6587

Supervisor (s): Name: Prof Zenghui Wang
E-mail address: wangz@unisa.a.cza, telephone # 011 471 3513

Working title of research:

Optimization of Neural Networks using Genetic Algorithms and Bayesian Optimization

Qualification: MTech Engineering Electrical

Thank you for the application for research ethics clearance by the Unisa SOE Ethics Review Committee for the above mentioned research. Ethics approval is granted for 3 years.

*The **low risk application** was **reviewed** by the SOE Ethics Review Committee on 26/10/2017 in compliance with the Unisa Policy on Research Ethics and the Standard Operating Procedure on Research Ethics Risk Assessment. The decision was approved on 26/10/2017.*

The proposed research may now commence with the provisions that:

1. The researcher(s) will ensure that the research project adheres to the values and principles expressed in the UNISA Policy on Research Ethics.

University of South Africa
Preller Street, Muckleneuk Ridge, City of Tshwane
PO Box 392 UNISA 0003 South Africa
Telephone: +27 12 429 3111 Facsimile: +27 12 429 4150
www.unisa.ac.za

2. Any adverse circumstance arising in the undertaking of the research project that is relevant to the ethicality of the study should be communicated in writing to the SOE Committee.
3. The researcher(s) will conduct the study according to the methods and procedures set out in the approved application.
4. Any changes that can affect the study-related risks for the research participants, particularly in terms of assurances made with regards to the protection of participants' privacy and the confidentiality of the data, should be reported to the Committee in writing, accompanied by a progress report.
5. The researcher will ensure that the research project adheres to any applicable national legislation, professional codes of conduct, institutional guidelines and scientific standards relevant to the specific field of study. Adherence to the following South African legislation is important, if applicable: Protection of Personal Information Act, no 4 of 2013; Children's act no 38 of 2005 and the National Health Act, no 61 of 2003.
6. Only de-identified research data may be used for secondary research purposes in future on condition that the research objectives are similar to those of the original research. Secondary use of identifiable human research data require additional ethics clearance.
7. No field work activities may continue after the expiry date (xxx). Submission of a completed research ethics progress report will constitute an application for renewal of Ethics Research Committee approval.

Note:

*The reference number **2017/CSET_SOE/WR/001** should be clearly indicated on all forms of communication with the intended research participants, as well as with the Committee.*

Yours sincerely,

.....

Dr T Sithebe

Acting Chair of SOE ERC

E-mail: sithet@unisa.ac.za

Tel: (012) 429-3864

.....

Prof B B Mamba

Executive Dean : XXX

E-mail:mambabb@unisa.ac.za

Tel: (011) 670 9230

University of South Africa
Preller Street, Muckleneuk Ridge, City of Tshwane
PO Box 392 UNISA 0003 South Africa
Telephone: +27 12 429 3111 Facsimile: +27 12 429 4150
www.unisa.ac.za

Appendix D: List of publications and other contributions

Peer reviewed international journals:

1. **Rawat, W**., & Wang, Z. (2017). Deep convolutional neural networks for image classification: A comprehensive review. *Neural Computation, 29*(9), 2352-2449. doi:10.1162/neco_a_00990. (**ISI Master indexed journal**)

Manuscripts under review in international journals:

2. **Rawat, W.**, & Wang, Z. (2017). Hybrid stochastic GA-Bayesian search for Deep Convolutional Neural Network model selection, *submitted to Neural Computation.*

Other contributions

3. **Research compilation, editing and survey work (6/7 chapters complete):**

   Sun. Y., (2017). Neural Network-based Swarm Optimization. *Manuscript in preparation*.